The MySQL Cluster API Developer Guide

Version 2.0 (2009-06-02)

The MySQL Cluster API Developer Guide: Version 2.0 (2009-06-02)

The MySQL Cluster API Developer Guide

Document generated on: 2009-06-02 (revision: 15165)

This guide provides information for developers wishing to use: The low-level C/C++-language NDB API for the MySQL NDBCLUSTER storage engine, the C-language MGM API for communicating with and controlling MySQL Cluster management servers, and other APIs used with MySQL in the context of MySQL Cluster. This Guide includes concepts, terminology, class and function references, practical examples, common problems, and tips for using these APIs in applications. It also contains information about NDB internals that may be of interest to developers working with NDBCLUSTER, such as communication protocols employed between nodes, filesystems used by data nodes, and error messages.

The information presented in this guide is current for recent MySQL Cluster NDB 6.2, NDB 6.3, and NDB 7.0 releases. You should be aware that there have been significant changes in the NDB API, MGM API, and other particulars in MySQL Cluster versions since MySQL 5.1.12.

Table of Contents

Chapter 1. Overview and Concepts

This chapter provides a general overview of essential MySQL Cluster, NDB API, and MGM API concepts, terminology, and programming constructs.

1.1. Introduction

This section introduces the NDB Transaction and Scanning APIs as well as the NDB Management (MGM) API for use in building applications to run on MySQL Cluster. It also discusses the general theory and principles involved in developing such applications.

1.1.1. The NDB API

The *NDB API* is an object-oriented application programming interface for MySQL Cluster that implements indexes, scans, transactions, and event handling. NDB transactions are ACID-compliant in that they provide a means to group together operations in such a way that they succeed (commit) or fail as a unit (rollback). It is also possible to perform operations in a "no-commit" or deferred mode, to be committed at a later time.

NDB scans are conceptually rather similar to the SQL cursors implemented in MySQL 5.0 and other common enterprise-level database management systems. These allow for high-speed row processing for record retrieval purposes. (MySQL Cluster naturally supports set processing just as does MySQL in its non-Cluster distributions. This can be accomplished via the usual MySQL APIs discussed in the MySQL Manual and elsewhere.) The NDB API supports both table scans and row scans; the latter can be performed using either unique or ordered indexes. Event detection and handling is discussed in Section 2.3.11, “The `NdbEventOperation` Class”, as well as Section 2.4.7, “NDB API Event Handling Example”.

In addition, the NDB API provides object-oriented error-handling facilities in order to provide a means of recovering gracefully from failed operations and other problems. See Section 2.4.3, “Handling Errors and Retrying Transactions”, for a detailed example.

The NDB API provides a number of classes implementing the functionality described above. The most important of these include the `Ndb`, `Ndb_cluster_connection`, `NdbTransaction`, and `NdbOperation` classes. These model (respectively) database connections, cluster connections, transactions, and operations. These classes and their subclasses are listed in Section 2.3, “NDB API Classes, Interfaces, and Structures”. Error conditions in the NDB API are handled using `NdbError`, a structure which is described in Section 2.3.30, “The `NdbError` Structure”.

1.1.2. The MGM API

The *MySQL Cluster Management API*, also known as the *MGM API*, is a C-language programming interface intended to provide administrative services for the cluster. These include starting and stopping Cluster nodes, handling Cluster logging, backups, and restoration from backups, as well as various other management tasks. A conceptual overview of MGM and its uses can be found in Chapter 3, *The MGM API*.

The MGM API's principal structures model the states of individual modes (`ndb_mgm_node_state`), the state of the Cluster as a whole (`ndb_mgm_cluster_state`), and management server response messages (`ndb_mgm_reply`). See Section 3.4, “MGM Structures”, for detailed descirptions of these.

1.2. Terminology

Provides a glossary of terms which are unique to the NDB and MGM APIs, or have a specialised meaning when applied therein.

The following terms are useful to an understanding of MySQL Cluster, the NDB API, or have a specialised meaning when used in one of these contexts. In addition, you may find the MySQL Manual's MySQL Cluster Glossary to be useful as well.

- **Backup**: A complete copy of all cluster data, transactions and logs, saved to disk.
- **Restore**: Returning the cluster to a previous state as stored in a backup.
- **Checkpoint**: Generally speaking, when data is saved to disk, it is said that a checkpoint has been reached. When working with the `NDB` storage engine, there are two sorts of checkpoints which work together in order to ensure that a consistent view of the cluster's data is maintained:
  - **Local Checkpoint (LCP)**: This is a checkpoint that is specific to a single node; however, LCPs take place for all nodes in the cluster more or less concurrently. An LCP involves saving all of a node's data to disk, and so usually occurs every few minutes, depending upon the amount of data stored by the node.

    More detailed information about LCPs and their behaviour can be found in the MySQL Manual, in the sections Defining MySQL Cluster Data Nodes, and Configuring MySQL Cluster Parameters for Local Checkpoints.

    - **Global Checkpoint (GCP)**: A GCP occurs every few seconds, when transactions for all nodes are synchronized and the REDO log is flushed to disk.

      A related term is *GCI*, which stands for "Global Checkpoint ID". This marks the point in the REDO log where a GCP took place.

- **Node**: A component of MySQL Cluster. 3 node types are supported:

    - *Management (MGM) node*: This is an instance of `ndb_mgmd`, the cluster management server daemon.

    - *Data node* (sometimes also referred to as a "storage nodes", although this usage is now discouraged): This is an instance of `ndbd`, and stores cluster data.

    - *API node*: This is an application that accesses cluster data. *SQL node* refers to a `mysqld` process that is connected to the cluster as an API node.

  For more information about these node types, please refer to Section 1.3.3, "Review of MySQL Cluster Concepts", or to MySQL Cluster Programs, in the MySQL Manual.

- **Node Failure**: MySQL Cluster is not solely dependent upon the functioning of any single node making up the cluster, which can continue to run even when one node fails.

- **Node Restart**: The process of restarting a cluster node which has stopped on its own or been stopped deliberately. This can be done for several different reasons, including the following:

    - Restarting a node which has shut down on its own (when this has occurred, it is known as *forced shutdown* or *node failure*; the other cases dicussed here involve manually shutting down the node and restarting it)

    - To update the node's configuration

    - As part of a software or hardware upgrade

    - In order to defragment the node's `DataMemory`

- **Initial Node Restart**: The process of starting a cluster node with its file system removed. This is sometimes used in the course of software upgrades and in other special circumstances.

- **System Crash** (or **System Failure**): This can occur when so many cluster nodes have failed that the cluster's state can no longer be guaranteed.

- **System Restart**: The process of restarting the cluster and reinitialising its state from disk logs and checkpoints. This is required after either a planned or an unplanned shutdown of the cluster.

- **Fragment**: Contains a portion of a database table; in other words, in the `NDB` storage engine, a table is broken up into and stored as a number of subsets, usually referred to as fragments. A fragment is sometimes also called a *partition*.

- **Replica**: Under the `NDB` storage engine, each table fragment has number of replicas in order to provide redundancy.

- **Transporter**: A protocol providing data transfer across a network. The NDB API supports 4 different types of transporter connections: TCP/IP (local), TCP/IP (remote), SCI, and SHM. TCP/IP is, of course, the familiar network protocol that underlies HTTP, FTP, and so forth, on the Internet. SCI (Scalable Coherent Interface) is a high-speed protocol used in building multiprocessor systems and parallel-processing applications. SHM stands for Unix-style shared memory segments. For an informal introduction to SCI, see this essay at dolphinics.com.

- `NDB`: This originally stood for "Network Database". It now refers to the storage engine used by MySQL AB to enable its MySQL Cluster distributed database.

- **ACC**: Access Manager. Handles hash indexes of primary keys providing speedy access to the records.

- **TUP**: Tuple Manager. This handles storage of tuples (records) and contains the filtering engine used to filter out records and attributes when performing reads and/or updates.

- **TC**: Transaction Coordinator. Handles co-ordination of transactions and timeouts; serves as the interface to the NDB API for indexes and scan operations.

1.3. The `NDBCLUSTER` Transaction and Scanning API

This section defines and discusses the high-level architecture of the NDB API, and introduces the NDB classes which are of greatest use and interest to the developer. It also covers most important NDB API concepts, including a review of MySQL Cluster

Concepts.

1.3.1. Core NDB API Classes

The NDB API is a MySQL Cluster application interface that implements transactions. It consists of the following fundamental classes:

- `Ndb_cluster_connection` represents a connection to a cluster.

  See Section 2.3.24, "The `Ndb_cluster_connection` Class".

- `Ndb` is the main class, and represents a connection to a database.

  See Section 2.3.8, "The `Ndb` Class".

- `NdbDictionary` provides meta-information about tables and attributes.

  See Section 2.3.10, "The `NdbDictionary` Class".

- `NdbTransaction` represents a transaction.

  See Section 2.3.19, "The `NdbTransaction` Class".

- `NdbOperation` represents an operation using a primary key.

  See Section 2.3.15, "The `NdbOperation` Class".

- `NdbScanOperation` represents an operation performing a full table scan.

  See Section 2.3.18, "The `NdbScanOperation` Class".

- `NdbIndexOperation` represents an operation using a unique hash index.

  See Section 2.3.12, "The `NdbIndexOperation` Class".

- `NdbIndexScanOperation` represents an operation performing a scan using an ordered index.

  See Section 2.3.13, "The `NdbIndexScanOperation` Class".

- `NdbRecAttr` represents an attribute value.

  See Section 2.3.16, "The `NdbRecAttr` Class".

In addition, the NDB API defines an `NdbError` structure, which contains the specification for an error.

It is also possible to receive events triggered when data in the database is changed. This is accomplished through the `NdbEventOperation` class.

> **Important**
>
> The `NDB` event notification API is not supported prior to MySQL 5.1. (Bug#19719)

For more information about these classes as well as some additional auxiliary classes not listed here, see Section 2.3, "NDB API Classes, Interfaces, and Structures".

1.3.2. Application Program Basics

The main structure of an application program is as follows:

1. Connect to a cluster using the `Ndb_cluster_connection` object.
2. Initiate a database connection by constructing and initialising one or more `Ndb` objects.
3. Identify the tables, columns, and indexes on which you wish to operate, using `NdbDictionary` and one or more of its subclasses.
4. Define and execute transactions using the `NdbTransaction` class.

5. Delete `Ndb` objects.
6. Terminate the connection to the cluster (terminate an instance of `Ndb_cluster_connection`).

1.3.2.1. Using Transactions

The procedure for using transactions is as follows:

1. Start a transaction (instantiate an `NdbTransaction` object).
2. Add and define operations associated with the transaction using instances of one or more of the `NdbOperation`, `NdbScanOperation`, `NdbIndexOperation`, and `NdbIndexScanOperation` classes.
3. Execute the transaction (call `NdbTransaction::execute()`).
4. The operation can be of two different types — `Commit` or `NoCommit`:
   - If the operation is of type `NoCommit`, then the application program requests that the operation portion of a transaction be executed, but without actually committing the transaction. Following the execution of a `NoCommit` operation, the program can continue to define additional transaction operations for later execution.

     `NoCommit` operations can also be rolled back by the application.
   - If the operation is of type `Commit`, then the transaction is immediately committed. The transaction must be closed after it has been committed (even if the commit fails), and no further operations can be added to or defined for this transaction.

   See Section 2.3.19.1.3, "The `NdbTransaction::ExecType` Type".

1.3.2.2. Synchronous Transactions

Synchronous transactions are defined and executed as follows:

1. Begin (create) the transaction, which is referenced by an `NdbTransaction` object typically created using `Ndb::startTransaction()`. At this point, the transaction is merely being defined; it is not yet sent to the NDB kernel.
2. Define operations and add them to the transaction, using one or more of the following:
   - `NdbTransaction::getNdbOperation()`
   - `NdbTransaction::getNdbScanOperation()`
   - `NdbTransaction::getNdbIndexOperation()`
   - `NdbTransaction::getNdbIndexScanOperation()`

   along with the appropriate methods of the respective`NdbOperation` class (or possibly one or more of its subclasses). Note that, at this point, the transaction has still not yet been sent to the NDB kernel.
3. Execute the transaction, using the `NdbTransaction::execute()` method.
4. Close the transaction by calling `Ndb::closeTransaction()`.

For an example of this process, see Section 2.4.1, "Using Synchronous Transactions".

To execute several synchronous transactions in parallel, you can either use multiple `Ndb` objects in several threads, or start multiple application programs.

1.3.2.3. Operations

An `NdbTransaction` consists of a list of operations, each of which is represented by an instance of `NdbOperation`, `NdbScanOperation`, `NdbIndexOperation`, or `NdbIndexScanOperation` (that is, of `NdbOperation` or one of its child classes).

Some general information about cluster access operation types can be found in MySQL Cluster Interconnects and Performance, in the MySQL Manual.

1.3.2.3.1. Single-row operations

After the operation is created using NdbTransaction::getNdbOperation() or NdbTransaction::getNdbIndexOperation(), it is defined in the following three steps:

1. Specify the standard operation type using `NdbOperation::readTuple()`.
2. Specify search conditions using `NdbOperation::equal()`.
3. Specify attribute actions using `NdbOperation::getValue()`.

Here are two brief examples illustrating this process. For the sake of brevity, we omit error handling.

This first example uses an `NdbOperation`:

```
// 1. Retrieve table object
myTable= myDict->getTable("MYTABLENAME");

// 2. Create an NdbOperation on this table
myOperation= myTransaction->getNdbOperation(myTable);

// 3. Define the operation's type and lock mode
myOperation->readTuple(NdbOperation::LM_Read);

// 4. Specify search conditions
myOperation->equal("ATTR1", i);

// 5. Perform attribute retrieval
myRecAttr= myOperation->getValue("ATTR2", NULL);
```

For additional examples of this sort, see Section 2.4.1, “Using Synchronous Transactions”.

The second example uses an `NdbIndexOperation`:

```
// 1. Retrieve index object
myIndex= myDict->getIndex("MYINDEX", "MYTABLENAME");

// 2. Create
myOperation= myTransaction->getNdbIndexOperation(myIndex);

// 3. Define type of operation and lock mode
myOperation->readTuple(NdbOperation::LM_Read);

// 4. Specify Search Conditions
myOperation->equal("ATTR1", i);

// 5. Attribute Actions
myRecAttr = myOperation->getValue("ATTR2", NULL);
```

Another example of this second type can be found in Section 2.4.5, “Using Secondary Indexes in Scans”.

We now discuss in somewhat greater detail each step involved in the creation and use of synchronous transactions.

1. **Define single row operation type.** The following operation types are supported:

   - `NdbOperation::insertTuple()`: Inserts a non-existing tuple.
   - `NdbOperation::writeTuple()`: Updates a tuple if one exists, otherwise inserts a new tuple.
   - `NdbOperation::updateTuple()`: Updates an existing tuple.
   - `NdbOperation::deleteTuple()`: Deletes an existing tuple.
   - `NdbOperation::readTuple()`: Reads an existing tuple using the specified lock mode.

   All of these operations operate on the unique tuple key. When `NdbIndexOperation` is used, then each of these operations operates on a defined unique hash index.

   > **Note**
   >
   > If you want to define multiple operations within the same transaction, then you need to call `NdbTransaction::getNdbOperation()` or `NdbTransaction::getNdbIndexOperation()` for each operation.

2. **Specify Search Conditions.** The search condition is used to select tuples. Search conditions are set using `NdbOperation::equal()`.

3. **Specify Attribute Actions.** Next, it is necessary to determine which attributes should be read or updated. It is important to re-

member that:

- Deletes can neither read nor set values, but only delete them.
- Reads can only read values.
- Updates can only set values. Normally the attribute is identified by name, but it is also possible to use the attribute's identity to determine the attribute.

`NdbOperation::getValue()` returns an `NdbRecAttr` object containing the value as read. To obtain the actual value, one of two methods can be used; the application can either

- Use its own memory (passed through a pointer *`aValue`*) to `NdbOperation::getValue()`, or
- receive the attribute value in an `NdbRecAttr` object allocated by the NDB API.

The `NdbRecAttr` object is released when `Ndb::closeTransaction()` is called. For this reason, the application cannot reference this object following any subsequent call to `Ndb::closeTransaction()`. Attempting to read data from an `NdbRecAttr` object before calling `NdbTransaction::execute()` yields an undefined result.

1.3.2.3.2. Scan Operations

Scans are roughly the equivalent of SQL cursors, providing a means to perform high-speed row processing. A scan can be performed on either a table (using an `NdbScanOperation`) or an ordered index (by means of an `NdbIndexScanOperation`).

Scan operations have the following characteristics:

- They can perform read operations which may be shared, exclusive, or dirty.
- They can potentially work with multiple rows.
- They can be used to update or delete multiple rows.
- They can operate on several nodes in parallel.

After the operation is created using `NdbTransaction::getNdbScanOperation()` or `NdbTransaction::getNdbIndexScanOperation()`, it is carried out as follows:

1. Define the standard operation type, using `NdbScanOperation::readTuples()`.

   Note

   See Section 2.3.18.2.1, “`NdbScanOperation::readTuples()`”, for additional information about deadlocks which may occur when performing simultaneous, identical scans with exclusive locks.

2. Specify search conditions, using `NdbScanFilter`, `NdbIndexScanOperation::setBound()`, or both.
3. Specify attribute actions using `NdbOperation::getValue()`.
4. Execute the transaction using `NdbTransaction::execute()`.
5. Traverse the result set by means of successive calls to `NdbScanOperation::nextResult()`.

Here are two brief examples illustrating this process. Once again, in order to keep things relatively short and simple, we forego any error handling.

This first example performs a table scan using an `NdbScanOperation`:

```
// 1. Retrieve a table object
myTable= myDict->getTable("MYTABLENAME");

// 2. Create a scan operation (NdbScanOperation) on this table
myOperation= myTransaction->getNdbScanOperation(myTable);

// 3. Define the operation's type and lock mode
myOperation->readTuples(NdbOperation::LM_Read);

// 4. Specify search conditions
NdbScanFilter sf(myOperation);
sf.begin(NdbScanFilter::OR);
```

```
sf.eq(0, i);   // Return rows with column 0 equal to i or
sf.eq(1, i+1); // column 1 equal to (i+1)
sf.end();

// 5. Retrieve attributes
myRecAttr= myOperation->getValue("ATTR2", NULL);
```

The second example uses an NdbIndexScanOperation to perform an index scan:

```
// 1. Retrieve index object
myIndex= myDict->getIndex("MYORDEREDINDEX", "MYTABLENAME");

// 2. Create an operation (NdbIndexScanOperation object)
myOperation= myTransaction->getNdbIndexScanOperation(myIndex);

// 3. Define type of operation and lock mode
myOperation->readTuples(NdbOperation::LM_Read);

// 4. Specify search conditions
// All rows with ATTR1 between i and (i+1)
myOperation->setBound("ATTR1", NdbIndexScanOperation::BoundGE, i);
myOperation->setBound("ATTR1", NdbIndexScanOperation::BoundLE, i+1);

// 5. Retrieve attributes
myRecAttr = MyOperation->getValue("ATTR2", NULL);
```

Some additional discussion of each step required to perform a scan follows:

1. **Define Scan Operation Type.** It is important to remember that only a single operation is supported for each scan operation (`NdbScanOperation::readTuples()` or `NdbIndexScanOperation::readTuples()`).

   > **Note**
   >
   > If you want to define multiple scan operations within the same transaction, then you need to call `NdbTransaction::getNdbScanOperation()` or `NdbTransaction::getNdbIndexScanOperation()` separately for *each* operation.

2. **Specify Search Conditions.** The search condition is used to select tuples. If no search condition is specified, the scan will return all rows in the table. The search condition can be an `NdbScanFilter` (which can be used on both `NdbScanOperation` and `NdbIndexScanOperation`) or bounds (which can be used only on index scans - see `NdbIndexScanOperation::setBound()`). An index scan can use both `NdbScanFilter` and bounds.

   > **Note**
   >
   > When NdbScanFilter is used, each row is examined, whether or not it is actually returned. However, when using bounds, only rows within the bounds will be examined.

3. **Specify Attribute Actions.** Next, it is necessary to define which attributes should be read. As with transaction attributes, scan attributes are defined by name, but it is also possible to use the attributes' identities to define attributes as well. As discussed elsewhere in this document (see Section 1.3.2.2, “Synchronous Transactions”), the value read is returned by the `NdbOperation::getValue()` method as an `NdbRecAttr` object.

1.3.2.3.3. Using Scans to Update or Delete Rows

Scanning can also be used to update or delete rows. This is performed by

1. Scanning with exclusive locks using `NdbOperation::LM_Exclusive`.
2. (*When iterating through the result set*:) For each row, optionally calling either `NdbScanOperation::updateCurrentTuple()` or `NdbScanOperation::deleteCurrentTuple()`.
3. (*If performing `NdbScanOperation::updateCurrentTuple()`*:) Setting new values for records simply by using `NdbOperation::setValue()`. `NdbOperation::equal()` should not be called in such cases, as the primary key is retrieved from the scan.

   > **Important**
   >
   > The update or delete is not actually performed until the next call to `NdbTransaction::execute()` is made, just as with single row operations. `NdbTransaction::execute()` also must be called before any locks are released; for more information, see Section 1.3.2.3.4, “Lock Handling with Scans”.

Features Specific to Index Scans. When performing an index scan, it is possible to scan only a subset of a table using `NdbIndexScanOperation::setBound()`. In addition, result sets can be sorted in either ascending or descending order, using `NdbIndexScanOperation::readTuples()`. Note that rows are returned unordered by default unless *`sorted`* is set to `true`. It is also important to note that, when using `NdbIndexScanOperation::BoundEQ()` on a partition key, only fragments containing rows will actually be scanned. Finally, when performing a sorted scan, any value passed as the `NdbIndexScanOperation::readTuples()` method's *`parallel`* argument will be ignored and maximum parallelism will be used instead. In other words, all fragments which it is possible to scan are scanned simultaneously and in parallel in such cases.

1.3.2.3.4. Lock Handling with Scans

Performing scans on either a table or an index has the potential to return a great many records; however, Ndb locks only a predetermined number of rows per fragment at a time. The number of rows locked per fragment is controlled by the batch parameter passed to `NdbScanOperation::readTuples()`.

In order to allow the application to handle how locks are released, `NdbScanOperation::nextResult()` has a Boolean parameter *`fetchAllowed`*. If `NdbScanOperation::nextResult()` is called with *`fetchAllowed`* equal to `false`, then no locks may be released as result of the function call. Otherwise the locks for the current batch may be released.

This next example shows a scan delete that handles locks in an efficient manner. For the sake of brevity, we omit error-handling.

```
int check;

// Outer loop for each batch of rows
while((check = MyScanOperation->nextResult(true)) == 0)
{
  do
  {
    // Inner loop for each row within the batch
    MyScanOperation->deleteCurrentTuple();
  }
  while((check = MyScanOperation->nextResult(false)) == 0);

  // When there are no more rows in the batch, execute all defined deletes
  MyTransaction->execute(NoCommit);
}
```

For a more complete example of a scan, see Section 2.4.4, “Basic Scanning Example”.

1.3.2.3.5. Error Handling

Errors can occur either when operations making up a transaction are being defined, or when the transaction is actually being executed. Catching and handling either sort of error requires testing the value returned by `NdbTransaction::execute()`, and then, if an error is indicated (that is, if this value is equal to `-1`), using the following two methods in order to identify the error's type and location:

- `NdbTransaction::getNdbErrorOperation()` returns a reference to the operation causing the most recent error.
- `NdbTransaction::getNdbErrorLine()` yields the method number of the erroneous method in the operation, starting with `1`.

This short example illustrates how to detect an error and to use these two methods to identify it:

```
theTransaction = theNdb->startTransaction();
theOperation = theTransaction->getNdbOperation("TEST_TABLE");
if(theOperation == NULL)
  goto error;

theOperation->readTuple(NdbOperation::LM_Read);
theOperation->setValue("ATTR_1", at1);
theOperation->setValue("ATTR_2", at1);  //  Error occurs here
theOperation->setValue("ATTR_3", at1);
theOperation->setValue("ATTR_4", at1);

if(theTransaction->execute(Commit) == -1)
{
  errorLine = theTransaction->getNdbErrorLine();
  errorOperation = theTransaction->getNdbErrorOperation();
}
```

Here, `errorLine` is `3`, as the error occurred in the third method called on the `NdbOperation` object (in this case, `theOperation`). If the result of `NdbTransaction::getNdbErrorLine()` is `0`, then the error occurred when the operations were executed. In this example, `errorOperation` is a pointer to the object `theOperation`. The `NdbTransaction::getNdbError()` method returns an `NdbError` object providing information about the error.

Note

Transactions are *not* automatically closed when an error occurs. You must call `Ndb::closeTransaction()` or `NdbTransaction::close()` to close the transaction.

See Section 2.3.8.1.9, "`Ndb::closeTransaction()`", and Section 2.3.19.2.7, "`NdbTransaction::close()`".

One recommended way to handle a transaction failure (that is, when an error is reported) is as shown here:

1. Roll back the transaction by calling `NdbTransaction::execute()` with a special `ExecType` value for the *`type`* parameter.

   See Section 2.3.19.2.5, "`NdbTransaction::execute()`" and Section 2.3.19.1.3, "The `NdbTransaction::ExecType` Type", for more information about how this is done.

2. Close the transaction by calling `NdbTransaction::closeTransaction()`.

3. If the error was temporary, attempt to restart the transaction.

Several errors can occur when a transaction contains multiple operations which are simultaneously executed. In this case the application must go through all operations and query each of their `NdbError` objects to find out what really happened.

Important

Errors can occur even when a commit is reported as successful. In order to handle such situations, the NDB API provides an additional `NdbTransaction::commitStatus()` method to check the transaction's commit status.

See Section 2.3.19.2.10, "`NdbTransaction::commitStatus()`".

1.3.3. Review of MySQL Cluster Concepts

This section covers the NDB Kernel, and discusses MySQL Cluster transaction handling and transaction coordinators. It also describes NDB record structures and concurrency issues.

The *NDB Kernel* is the collection of data nodes belonging to a MySQL Cluster. The application programmer can for most purposes view the set of all storage nodes as a single entity. Each data node is made up of three main components:

- **TC**: The transaction coordinator.
- **ACC**: The index storage component.
- **TUP**: The data storage component.

When an application executes a transaction, it connects to one transaction coordinator on one data node. Usually, the programmer does not need to specify which TC should be used, but in some cases where performance is important, the programmer can provide "hints" to use a certain TC. (If the node with the desired transaction coordinator is down, then another TC will automatically take its place.)

Each data node has an ACC and a TUP which store the indexes and data portions of the database table fragment. Even though a single TC is responsible for the transaction, several ACCs and TUPs on other data nodes might be involved in that transaction's execution.

1.3.3.1. Selecting a Transaction Coordinator

The default method is to select the transaction coordinator (TC) determined to be the "nearest" data node, using a heuristic for proximity based on the type of transporter connection. In order of nearest to most distant, these are:

1. SCI
2. SHM
3. TCP/IP (localhost)
4. TCP/IP (remote host)

If there are several connections available with the same proximity, one is selected for each transaction in a round-robin fashion.

Optionally, you may set the method for TC selection to round-robin mode, where each new set of transactions is placed on the next data node. The pool of connections from which this selection is made consists of all available connections.

As noted in Section 1.3.3, “Review of MySQL Cluster Concepts”, the application programmer can provide hints to the NDB API as to which transaction coordinator should be uses. This is done by providing a table and a partition key (usually the primary key). If the primary key as the partition key, then the transaction is placed on the node where the primary replica of that record resides. Note that this is only a hint; the system can be reconfigured at any time, in which case the NDB API chooses a transaction coordinator without using the hint. For more information, see Section 2.3.1.2.16, “`Column::getPartitionKey()`”, and Section 2.3.8.1.8, “`Ndb::startTransaction()`”. The application programmer can specify the partition key from SQL by using this construct:

```
CREATE TABLE ... ENGINE=NDB PARTITION BY KEY (attribute_list);
```

For additional information, see Partitioning, and in particular `KEY` Partitioning, in the MySQL Manual.

1.3.3.2. NDB Record Structure

The `NDBCLUSTER` storage engine used by MySQL Cluster is a relational database engine storing records in tables as with other relational database systems. Table rows represent records as tuples of relational data. When a new table is created, its attribute schema is specified for the table as a whole, and thus each table row has the same structure. Again, this is typical of relational databases, and `NDB` is no different in this regard.

Primary Keys. Each record has from 1 up to 32 attributes which belong to the primary key of the table.

Transactions. Transactions are committed first to main memory, and then to disk, after a global checkpoint (GCP) is issued. Since all data are (in most NDB Cluster configurations) synchronously replicated and stored on multiple data nodes, the system can handle processor failures without loss of data. However, in the case of a system-wide failure, all transactions (committed or not) occurring since the most recent GCP are lost.

Concurrency Control. `NDBCLUSTER` uses *pessimistic concurrency control* based on locking. If a requested lock (implicit and depending on database operation) cannot be attained within a specified time, then a timeout error results.

Concurrent transactions as requested by parallel application programs and thread-based applications can sometimes deadlock when they try to access the same information simultaneously. Thus, applications need to be written in a manner such that timeout errors occurring due to such deadlocks are handled gracefully. This generally means that the transaction encountering a timeout should be rolled back and restarted.

Hints and Performance. Placing the transaction coordinator in close proximity to the actual data used in the transaction can in many cases improve performance significantly. This is particularly true for systems using TCP/IP. For example, a Solaris system using a single 500 MHz processor has a cost model for TCP/IP communication which can be represented by the formula

```
[30 microseconds] + ([100 nanoseconds] * [number of bytes])
```

This means that if we can ensure that we use “popular” links we increase buffering and thus drastically reduce the costs of communication. The same system using SCI has a different cost model:

```
[5 microseconds] + ([10 nanoseconds] * [number of bytes])
```

This means that the efficiency of an SCI system is much less dependent on selection of transaction coordinators. Typically, TCP/IP systems spend 30 to 60% of their working time on communication, whereas for SCI systems this figure is in the range of 5 to 10%. Thus, employing SCI for data transport means that less effort from the NDB API programmer is required and greater scalability can be achieved, even for applications using data from many different parts of the database.

A simple example would be an application that uses many simple updates where a transaction needs to update one record. This record has a 32-bit primary key which also serves as the partitioning key. Then the `keyData` is used as the address of the integer of the primary key and `keyLen` is `4`.

1.3.4. The Adaptive Send Algorithm

Discusses the mechanics of transaction handling and transmission in MySQL Cluster and the NDB API, and the objects used to implement these.

When transactions are sent using `NdbTransaction::execute()`, they are not immediately transferred to the NDB Kernel. Instead, transactions are kept in a special send list (buffer) in the `Ndb` object to which they belong. The adaptive send algorithm decides when transactions should actually be transferred to the NDB kernel.

The NDB API is designed as a multi-threaded interface, and so it is often desirable to transfer database operations from more than one thread at a time. The NDB API keeps track of which `Ndb` objects are active in transferring information to the NDB kernel and

the expected number of threads to interact with the NDB kernel. Note that a given instance of `Ndb` should be used in at most one thread; different threads should *not* share the same `Ndb` object.

There are four conditions leading to the transfer of database operations from `Ndb` object buffers to the NDB kernel:

1. The NDB Transporter (TCP/IP, SCI, or shared memory) decides that a buffer is full and sends it off. The buffer size is implementation-dependent and may change between MySQL Cluster releases. When TCP/IP is the transporter, the buffer size is usually around 64 KB. Since each `Ndb` object provides a single buffer per data node, the notion of a “full” buffer is local to each data node.

2. The accumulation of statistical data on transferred information may force sending of buffers to all storage nodes (that is, when all the buffers become full).

3. Every 10 ms, a special transmission thread checks whether or not any send activity has occurred. If not, then the thread will force transmission to all nodes. This means that 20 ms is the maximum amount of time that database operations are kept waiting before being dispatched. A 10-millisecond limit is likely in future releases of MySQL Cluster; checks more frequent than this require additional support from the operating system.

4. For methods that are affected by the adaptive send algorithm (such as `NdbTransaction::execute()`), there is a *`force`* parameter that overrides its default behaviour in this regard and forces immediate transmission to all nodes. See the individual NDB API class listings for more information.

> **Note**
>
> The conditions listed above are subject to change in future releases of MySQL Cluster.

Chapter 2. The NDB API

This chapter contains information about the NDB API, which is used to write applications that access data in the NDBCLUSTER storage engine.

2.1. Getting Started with the NDB API

This section discusses preparations necessary for writing and compiling an NDB API application.

2.1.1. Compiling and Linking NDB API Programs

This section provides information on compiling and linking NDB API applications, including requirements and compiler and linker options.

2.1.1.1. General Requirements

To use the NDB API with MySQL, you must have the NDB client library and its header files installed alongside the regular MySQL client libraries and headers. These are automatically installed when you build MySQL using the `--with-ndbcluster` `configure` option or when using a MySQL binary package that supports the NDBCLUSTER storage engine.

> **Note**
>
> MySQL 4.1 does not install the required NDB-specific header files. You should use MySQL 5.0 or later when writing NDB API applications, and this Guide is targeted for use with MySQL 5.1.
>
> The library and header files were not included in MySQL 5.1 binary distributions prior to MySQL 5.1.12; beginning with 5.1.12, you can find them in `/usr/include/storage/ndb`. This issue did not occur when compiling MySQL 5.1 from source.

2.1.1.2. Compiler Options

Header Files. In order to compile source files that use the NDB API, you must ensure that the necessary header files can be found. Header files specific to the NDB API are installed in the following subdirectories of the MySQL `include` directory:

- `include/mysql/storage/ndb/ndbapi`
- `include/mysql/storage/ndb/mgmapi`

Compiler Flags. The MySQL-specific compiler flags needed can be determined using the `mysql_config` utility that is part of the MySQL installation:

```
$ mysql_config --cflags
-I/usr/local/mysql/include/mysql -Wreturn-type -Wtrigraphs -W -Wformat
-Wsign-compare -Wunused  -mcpu=pentium4 -march=pentium4
```

This sets the include path for the MySQL header files but not for those specific to the NDB API. The `--include` option to `mysql_config` returns the generic include path switch:

```
shell> mysql_config --include
-I/usr/local/mysql/include/mysql
```

It is necessary to add the subdirectory paths explicitly, so that adding all the needed compile flags to the CXXFLAGS shell variable should look something like this:

```
CFLAGS="$CFLAGS "`mysql_config --cflags`
CFLAGS="$CFLAGS "`mysql_config --include`/storage/ndb
CFLAGS="$CFLAGS "`mysql_config --include`/storage/ndb/ndbapi
CFLAGS="$CFLAGS "`mysql_config --include`/storage/ndb/mgmapi
```

> **Tip**
>
> If you do not intend to use the Cluster management functions, the last line in the previous example can be omitted. However, if you are interested in the management functions only, and do not want or need to access Cluster data except from MySQL, then you can omit the line referencing the `ndbapi` directory.

2.1.1.3. Linker Options

NDB API applications must be linked against both the MySQL and NDB client libraries. The NDB client library also requires some

functions from the `mystrings` library, so this must be linked in as well.

The necessary linker flags for the MySQL client library are returned by `mysql_config --libs`. For multithreaded applications you should use the `--libs_r` instead:

```
$ mysql_config --libs_r
-L/usr/local/mysql-5.1/lib/mysql -lmysqlclient_r -lz -lpthread -lcrypt
-lnsl -lm -lpthread -L/usr/lib -lssl -lcrypto
```

Formerly, to link an NDB API application, it was necessary to add `-lndbclient`, `-lmysys`, and `-lmystrings` to these options, in the order shown, and adding all the required linker flags to the `LDFLAGS` variable looked something like this:

```
LDFLAGS="$LDFLAGS "`mysql_config --libs_r`
LDFLAGS="$LDFLAGS -lndbclient -lmysys -lmystrings"
```

Beginning with MySQL 5.1.24-ndb-6.2.16 and MySQL 5.1.24-ndb-6.3.14, it is necessary only to add `-lndbclient` to `LD_FLAGS`, as shown here:

```
LDFLAGS="$LDFLAGS "`mysql_config --libs_r`
LDFLAGS="$LDFLAGS -lndbclient"
```

(For more information about this change, see Bug#29791.)

2.1.1.4. Using Autotools

It is often faster and simpler to use GNU autotools than to write your own makefiles. In this section, we provide an autoconf macro `WITH_MYSQL` that can be used to add a `--with-mysql` option to a configure file, and that automatically sets the correct compiler and linker flags for given MySQL installation.

All of the examples in this chapter include a common `mysql.m4` file defining `WITH_MYSQL`. A typical complete example consists of the actual source file and the following helper files:

- `acinclude`
- `configure.in`
- `Makefile.m4`

`automake` also requires that you provide `README`, `NEWS`, `AUTHORS`, and `ChangeLog` files; however, these can be left empty.

To create all necessary build files, run the following:

```
aclocal
autoconf
automake -a -c
configure --with-mysql=/mysql/prefix/path
```

Normally, this needs to be done only once, after which `make` will accommodate any file changes.

Example 1-1: `acinclude.m4`.

```
m4_include([../mysql.m4])
```

Example 1-2: `configure.in`.

```
AC_INIT(example, 1.0)
AM_INIT_AUTOMAKE(example, 1.0)
WITH_MYSQL()
AC_OUTPUT(Makefile)
```

Example 1-3: `Makefile.am`.

```
bin_PROGRAMS = example
example_SOURCES = example.cc
```

Example 1-4: `WITH_MYSQL` source for inclusion in `acinclude.m4`.

```
dnl
dnl configure.in helper macros
dnl

AC_DEFUN([WITH_MYSQL], [
  AC_MSG_CHECKING(for mysql_config executable)

  AC_ARG_WITH(mysql, [  --with-mysql=PATH path to mysql_config binary or mysql prefix dir], [
```

```
  if test -x $withval -a -f $withval
    then
      MYSQL_CONFIG=$withval
    elif test -x $withval/bin/mysql_config -a -f $withval/bin/mysql_config
    then
     MYSQL_CONFIG=$withval/bin/mysql_config
    fi
  ], [
  if test -x /usr/local/mysql/bin/mysql_config -a -f /usr/local/mysql/bin/mysql_config
    then
      MYSQL_CONFIG=/usr/local/mysql/bin/mysql_config
    elif test -x /usr/bin/mysql_config -a -f /usr/bin/mysql_config
    then
      MYSQL_CONFIG=/usr/bin/mysql_config
    fi
  ])

  if test "x$MYSQL_CONFIG" = "x"
  then
    AC_MSG_RESULT(not found)
    exit 3
  else
    AC_PROG_CC
    AC_PROG_CXX

    # add regular MySQL C flags
    ADDFLAGS=`$MYSQL_CONFIG --cflags`

    # add NDB API specific C flags
    IBASE=`$MYSQL_CONFIG --include`
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb"
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb/ndbapi"
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb/mgmapi"

    CFLAGS="$CFLAGS $ADDFLAGS"
    CXXFLAGS="$CXXFLAGS $ADDFLAGS"

    LDFLAGS="$LDFLAGS "`$MYSQL_CONFIG --libs_r`" -lndbclient -lmystrings -lmysys"
    LDFLAGS="$LDFLAGS "`$MYSQL_CONFIG --libs_r`" -lndbclient -lmystrings"

    AC_MSG_RESULT($MYSQL_CONFIG)
  fi
])
```

2.1.2. Connecting to the Cluster

This section covers connecting an NDB API application to a MySQL cluster.

2.1.2.1. Include Files

NDB API applications require one or more of the following include files:

- Applications accessing Cluster data via the NDB API must include the file `NdbApi.hpp`.
- Applications making use of both the `NDB` API and the regular MySQL client API also need to include `mysql.h`.
- Applications that use cluster management functions need the include file `mgmapi.h`.

2.1.2.2. API Initialisation and Cleanup

Before using the NDB API, it must first be initialised by calling the `ndb_init()` function. Once an NDB API application is complete, call `ndb_end(0)` to perform a cleanup.

2.1.2.3. Establishing the Connection

To establish a connection to the server, it is necessary to create an instance of `Ndb_cluster_connection`, whose constructor takes as its argument a cluster connectstring; if no connectstring is given, `localhost` is assumed.

The cluster connection is not actually initiated until the `Ndb_cluster_connection::connect()` method is called. When invoked without any arguments, the connection attempt is retried each 1 second indefinitely until successful, and no reporting is done. See Section 2.3.24, “The `Ndb_cluster_connection` Class”, for details.

By default an API node will connect to the “nearest” data node — usually a data node running on the same machine, due to the fact that shared memory transport can be used instead of the slower TCP/IP. This may lead to poor load distribution in some cases, so it is possible to enforce a round-robin node connection scheme by calling the `set_optimized_node_selection()` method with `0` as its argument prior to calling `connect()`. (See Section 2.3.24.1.6, “`Ndb_cluster_connection::set_optimized_node_selection()`”.)

The `connect()` method initiates a connection to a cluster management node only — it does not wait for any connections to data nodes to be made. This can be accomplished by using `wait_until_ready()` after calling `connect()`. The `wait_until_ready()` method waits up to a given number of seconds for a connection to a data node to be established.

In the following example, initialisation and connection are handled in the two functions `example_init()` and `example_end()`, which will be included in subsequent examples via the file `example_connection.h`.

Example 2-1: Connection example.

```
#include <stdio.h>
#include <stdlib.h>
#include <NdbApi.hpp>
#include <mysql.h>
#include <mgmapi.h>

Ndb_cluster_connection* connect_to_cluster();
void disconnect_from_cluster(Ndb_cluster_connection *c);

Ndb_cluster_connection* connect_to_cluster()
{
  Ndb_cluster_connection* c;

  if(ndb_init())
    exit(EXIT_FAILURE);

  c= new Ndb_cluster_connection();

  if(c->connect(4, 5, 1))
  {
    fprintf(stderr, "Unable to connect to cluster within 30 seconds.\n\n");
    exit(EXIT_FAILURE);
  }

  if(c->wait_until_ready(30, 0) < 0)
  {
    fprintf(stderr, "Cluster was not ready within 30 seconds.\n\n");
    exit(EXIT_FAILURE);
  }
}

void disconnect_from_cluster(Ndb_cluster_connection *c)
{
  delete c;

  ndb_end(2);
}

int main(int argc, char* argv[])
{
  Ndb_cluster_connection *ndb_connection= connect_to_cluster();

  printf("Connection Established.\n\n");

  disconnect_from_cluster(ndb_connection);

  return EXIT_SUCCESS;
}
```

2.1.3. Mapping MySQL Database Object Names and Types to NDB

This section discusses NDB naming and other conventions with regard to database objects.

Databases and Schemas. Databases and schemas are not represented by objects as such in the NDB API. Instead, they are modelled as attributes of `Table` and `Index` objects. The value of the `database` attribute of one of these objects is always the same as the name of the MySQL database to which the table or index belongs. The value of the `schema` attribute of a `Table` or `Index` object is always '`def`' (for “default”).

Tables. MySQL table names are directly mapped to `NDB` table names without modification. Table names starting with '`NDB$`' are reserved for internal use>, as is the `SYSTAB_0` table in the `sys` database.

Indexes. There are two different type of NDB indexes:

- *Hash indexes* are unique, but not ordered.
- *B-tree indexes* are ordered, but allow duplicate values.

Names of unique indexes and primary keys are handled as follows:

- For a MySQL `UNIQUE` index, both a B-tree and a hash index are created. The B-tree index uses the MySQL name for the index; the name for the hash index is generated by appending '`$unique`' to the index name.

- For a MySQL primary key only a B-tree index is created. This index is given the name `PRIMARY`. There is no extra hash; however, the uniqueness of the primary key is guaranteed by making the MySQL key the internal primary key of the `NDB` table.

Column Names and Values. `NDB` column names are the same as their MySQL names.

Datatypes. MySQL datatypes are stored in `NDB` columns as follows:

- The MySQL `TINYINT`, `SMALLINT`, `INT`, and `BIGINT` datatypes map to `NDB` types having the same names and storage requirements as their MySQL counterparts.
- The MySQL `FLOAT` and `DOUBLE` datatypes are mapped to `NDB` types having the same names and storage requirements.
- The storage space required for a MySQL `CHAR` column is determined by the maximum number of characters and the column's character set. For most (but not all) character sets, each character takes one byte of storage. When using UTF-8, each character requires three bytes. You can find the number of bytes needed per character in a given character set by checking the `Maxlen` column in the output of `SHOW CHARACTER SET`.
- In MySQL 5.1, the storage requirements for a `VARCHAR` or `VARBINARY` column depend on whether the column is stored in memory or on disk:
  - For in-memory columns, the `NDBCLUSTER` storage engine supports variable-width columns with 4-byte alignment. This means that (for example) a the string `'abcde'` stored in a `VARCHAR(50)` column using the `latin1` character set requires 12 bytes — in this case, 2 bytes times 5 characters is 10, rounded up to the next even multiple of 4 yields 12. (This represents a change in behaviour from Cluster in MySQL 5.0 and 4.1, where a column having the same definition required 52 bytes storage per row regardless of the length of the string being stored in the row.)
  - In Disk Data columns, `VARCHAR` and `VARBINARY` are stored as fixed-width columns. This means that each of these types requires the same amount of storage as a `CHAR` of the same size.
- Each row in a Cluster `BLOB` or `TEXT` column is made up of two separate parts. One of these is of fixed size (256 bytes), and is actually stored in the original table. The other consists of any data in excess of 256 bytes, which stored in a hidden table. The rows in this second table are always 2000 bytes long. This means that record of *`size`* bytes in a `TEXT` or `BLOB` column requires
  - 256 bytes, if *`size`* `<= 256`
  - `256 + 2000 * ((`*`size`*` – 256) \ 2000) + 1)` bytes otherwise

2.2. The NDB API Object Hierarachy

This section provides a hierarchical listing of all classes, interfaces, and structures exposed by the NDB API.

- `Ndb`
  - `Key_part_ptr`
  - `PartitionSpec`
- `NdbBlob`
- `Ndb_cluster_connection`
- `NdbDictionary`
  - `AutoGrowSpecification`
  - `Dictionary`
    - `List`
      - `Element`
  - `Column`
  - `Object`

        - Datafile
        - Event
        - Index
        - LogfileGroup
        - Table
        - Tablespace
        - Undofile
    - RecordSpecification
- NdbError
- NdbEventOperation
- NdbInterpretedCode
- NdbOperation
    - NdbIndexOperation
    - NdbScanOperation
        - NdbIndexScanOperation
            - IndexBound
- NdbRecAttr
- NdbRecord
- NdbScanFilter
- NdbTransaction

2.3. NDB API Classes, Interfaces, and Structures

This section provides a detailed listing of all classes, interfaces, and stuctures defined in the NDB API.

Each listing includes:

- Description and purpose of the class, interface, or structure.
- Pointers, where applicable, to parent and child classes.
- A diagram of the class and its members.

  > **Note**
  >
  > The sections covering the NdbDictionary and NdbOperation classes also include entity-relationship diagrams showing the hierarchy of inner classes, subclasses, and public type descending from them.

- Detailed listings of all public members, including descriptions of all method parameters and type values.

Class, interface, and structure descriptions are provided in alphabetical order. For a hierarchical listing, see Section 2.2, “The NDB API Object Hierarachy”.

2.3.1. The Column Class

This class represents a column in an NDB Cluster table.

Parent class. NdbDictionary

Child classes. *None*

Description. Each instance of the `Column` is characterised by its type, which is determined by a number of type specifiers:

- Built-in type
- Array length or maximum length
- Precision and scale (*currently not in use*)
- Character set (applicable only to columns using string datatypes)
- Inline and part sizes (applicable only to `BLOB` columns)

These types in general correspond to MySQL datatypes and their variants. The data formats are same as in MySQL. The NDB API provides no support for constructing such formats; however, they are checked by the `NDB` kernel.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getName()`	Gets the name of the column
`getNullable()`	Checks whether the column can be set to `NULL`
`getPrimaryKey()`	Check whether the column is part of the table's primary key
`getColumnNo()`	Gets the column number
`equal()`	Compares `Column` objects
`getType()`	Gets the column's type (`Type` value)
`getLength()`	Gets the column's length
`getCharset()`	Get the character set used by a string (text) column (not applicable to columns not storing character data)
`getInlineSize()`	Gets the inline size of a `BLOB` column (not applicable to other column types)
`getPartSize()`	Gets the part size of a `BLOB` column (not applicable to other column types)
`getStripeSize()`	Gets a BLOB column's stripe size (not applicable to other column types)
`getSize()`	Gets the size of an element
`getPartitionKey()`	Checks whether the column is part of the table's partitioning key
`getArrayType()`	Gets the column's array type
`getStorageType()`	Gets the storage type used by this column
`getPrecision()`	Gets the column's precision (used for decimal types only)
`getScale()`	Gets the column's scale (used for decimal types only)
`Column()`	Class constructor; there is also a copy constructor
`~Column()`	Class destructor
`setName()`	Sets the column's name
`setNullable()`	Toggles the column's nullability
`setPrimaryKey()`	Determines whether the column is part of the primary key
`setType()`	Sets the column's `Type`
`setLength()`	Sets the column's length
`setCharset()`	Sets the character set used by a column containing character data (not applicable to non-textual columns)
`setInlineSize()`	Sets the inline size for a `BLOB` column (not applicable to non-`BLOB` columns)
`setPartSize()`	Sets the part size for a `BLOB` column (not applicable to non-`BLOB` columns)
`setStripeSize()`	Sets the stripe size for a `BLOB` column (not applicable to non-`BLOB` columns)
`setPartitionKey()`	Determines whether the column is part of the table's partitioning key
`setArrayType()`	Sets the column's `ArrayType`
`setStorageType()`	Sets the storage type to be used by this column

Method	Purpose / Use
`setPrecision()`	Sets the column's precision (used for decimal types only)
`setScale()`	Sets the column's scale (used for decimal types only)

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.1.2, "`Column` Methods".

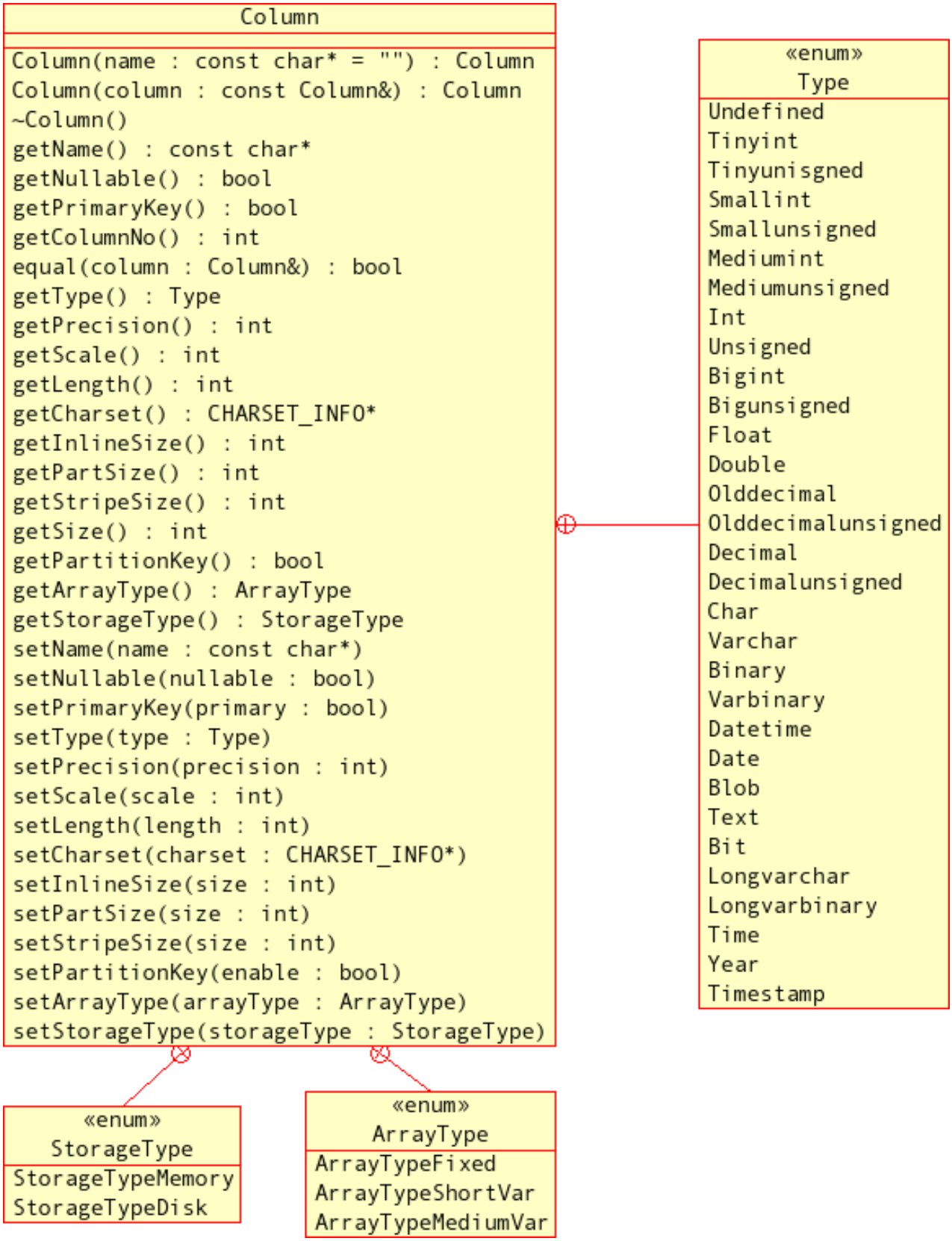
> **Important**
>
> In the NDB API, column names are handled in case-sensitive fashion. (This differs from the MySQL C API.) To reduce the possibility for error, it is recommended that you name all columns consistently using uppercase or lowercase.

Types. These are the public types of the `Column` class:

Type	Purpose / Use
`ArrayType`	Specifies the column's internal storage format
`StorageType`	Determines whether the column is stored in memory or on disk
`Type`	The column's datatype. `NDB` columns have the datatypes as found in MySQL

For a discussion of each of these types, along with its possible values, see Section 2.3.1.1, "`Column` Types".

Class diagram. This diagram shows all the available methods and enumerated types of the `Column` class:



2.3.1.1. `Column` Types

This section details the public types belonging to the `Column` class.

2.3.1.1.1. The `Column::ArrayType` Type

This type describes the `Column`'s internal attribute format.

Description. The attribute storage format can be either fixed or variable.

Enumeration values.

Value	Description
`ArrayTypeFixed`	stored as a fixed number of bytes
`ArrayTypeShortVar`	stored as a variable number of bytes; uses 1 byte overhead
`ArrayTypeMediumVar`	stored as a variable number of bytes; uses 2 bytes overhead

The fixed storage format is faster but also generally requires more space than the variable format. The default is `ArrayTypeShortVar` for `Var*` types and `ArrayTypeFixed` for others. The default is usually sufficient.

2.3.1.1.2. The `Column::StorageType` Type

This type describes the storage type used by a `Column` object.

Description. The storage type used for a given column can be either in memory or on disk. Columns stored on disk mean that less RAM is required overall but such columns cannot be indexed, and are potentially much slower to access. The default is `StorageTypeMemory`.

Enumeration values.

Value	Description
`StorageTypeMemory`	Store the column in memory
`StorageTypeDisk`	Store the column on disk

2.3.1.1.3. `Column::Type`

`Type` is used to describe the `Column` object's datatype.

Description. Datatypes for `Column` objects are analogous to the datatypes used by MySQL. The types `Tinyint`, `Tinyintunsigned`, `Smallint`, `Smallunsigned`, `Mediumint`, `Mediumunsigned`, `Int`, `Unsigned`, `Bigint`, `Bigunsigned`, `Float`, and `Double` (that is, types `Tinyint` through `Double` in the order listed in the Enumeration Values table) can be used in arrays.

Enumeration values.

Value	Description
`Undefined`	Undefined
`Tinyint`	1-byte signed integer
`Tinyunsigned`	1-byte unsigned integer
`Smallint`	2-byte signed integer
`Smallunsigned`	2-byte unsigned integer
`Mediumint`	3-byte signed integer
`Mediumunsigned`	3-byte unsigned integer
`Int`	4-byte signed integer
`Unsigned`	4-byte unsigned integer
`Bigint`	8-byte signed integer
`Bigunsigned`	8-byte signed integer
`Float`	4-byte float
`Double`	8-byte float
`Olddecimal`	Signed decimal as used prior to MySQL 5.0
`Olddecimalunsigned`	Unsigned decimal as used prior to MySQL 5.0
`Decimal`	Signed decimal as used by MySQL 5.0 and later
`Decimalunsigned`	Unsigned decimal as used by MySQL 5.0 and later
`Char`	A fixed-length array of 1-byte characters; maximum length is 255 characters

Value	Description
`Varchar`	A variable-length array of 1-byte characters; maximum length is 255 characters
`Binary`	A fixed-length array of 1-byte binary characters; maximum length is 255 characters
`Varbinary`	A variable-length array of 1-byte binary characters; maximum length is 255 characters
`Datetime`	An 8-byte date and time value, with a precision of 1 second
`Date`	A 4-byte date value, with a precision of 1 day
`Blob`	A binary large object; see Section 2.3.9, “The `NdbBlob` Class”
`Text`	A text blob
`Bit`	A bit value; the length specifies the number of bits
`Longvarchar`	A 2-byte `Varchar`
`Longvarbinary`	A 2-byte `Varbinary`
`Time`	Time without date
`Year`	1-byte year value in the range 1901-2155 (same as MySQL)
`Timestamp`	Unix time

> **Caution**
>
> Do not confuse `Column::Type` with `Object::Type` or `Table::Type`.

2.3.1.2. `Column` Methods

This section documents the public methods of the `Column` class.

> **Note**
>
> The assignment (`=`) operator is overloaded for this class, so that it always performs a deep copy.

> **Warning**
>
> As with other database objects, `Column` object creation and attribute changes to existing columns done using the NDB API are not visible from MySQL. For example, if you change a column's datatype using `Column::setType()`, MySQL will regard the type of column as being unchanged. The only exception to this rule with regard to columns is that you can change the name of an existing column using `Column::setName()`.
>
> Also remember that the NDB API handles column names in case-sensitive fashion.

2.3.1.2.1. `Column` Constructor

Description. You can create a new `Column` or copy an existing one using the class constructor.

> **Warning**
>
> A `Column` created using the NDB API will *not* be visible to a MySQL server.
>
> The NDB API handles column names in case-sensitive fashion. For example, if you create a column named “myColumn”, you will not be able to access it later using “Mycolumn” for the name. You can reduce the possibility for error, by naming all columns consistently using only uppercase or only lowercase.

Signature. You can create either a new instance of the `Column` class, or by copying an existing `Column` object. Both of these are shown here.

- Constructor for a new `Column`:

```
Column
    (
      const char* name = ""
    )
```

- Copy constructor:

```
Column
    (
```

```
      const Column& column
    )
```

Parameters. When creating a new instance of `Column`, the constructor takes a single argument, which is the name of the new column to be created. The copy constructor also takes one parameter — in this case, a reference to the `Column` instance to be copied.

Return Value. A `Column` object.

Destructor. The `Column` class destructor takes no arguments and *None*.

2.3.1.2.2. `Column::getName()`

Description. This method returns the name of the column for which it is called.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion. For example, if you retrieve the name “myColumn” for a given column, attempting to access this column using “Mycolumn” for the name fails with an error such as `COLUMN IS NULL` or `TABLE DEFINITION HAS UNDEFINED COLUMN`. You can reduce the possibility for error, by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the column.

2.3.1.2.3. `Column::getNullable()`

Description. This method is used to determine whether the column can be set to `NULL`.

Signature.

```
bool getNullable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A Boolean value: `true` if the column can be set to `NULL`, otherwise `false`.

2.3.1.2.4. `Column::getPrimaryKey()`

Description. This method is used to determine whether the column is part of the table's primary key.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
bool getPrimaryKey
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A Boolean value: `true` if the column is part of the primary key of the table to which this column belongs, otherwise `false`.

2.3.1.2.5. `Column::getColumnNo()`

Description. This method gets the number of a column — that is, its horizontal position within the table.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion, “myColumn” and “Mycolumn” are not considered to be the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
int getColumnNo
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column number as an integer.

2.3.1.2.6. `Column::equal()`

Description. This method is used to compare one `Column` with another to determine whether the two `Column` objects are the same.

Signature.

```
bool equal
    (
      const Column& column
    ) const
```

Parameters. `equal()` takes a single parameter, a reference to an instance of `Column`.

Return Value. `true` if the columns being compared are equal, otherwise `false`.

2.3.1.2.7. `Column::getType()`

Description. This method gets the column's datatype.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion, “myColumn” and “Mycolumn” are not considered to be the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The `Type` (datatype) of the column. For a list of possible values, see Section 2.3.1.1.3, “`Column::Type`”.

2.3.1.2.8. `Column::getPrecision()`

Description. This method gets the precision of a column.

> **Note**
>
> This method is applicable to decimal columns only.

Signature.

```
int getPrecision
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's precision, as an integer. The precision is defined as the number of significant digits; for more information, see the discussion of the `DECIMAL` datatype in Numeric Types, in the MySQL Manual.

2.3.1.2.9. `Column::getScale()`

Description. This method gets the scale used for a decimal column value.

> **Note**
>
> This method is applicable to decimal columns only.

Signature.

```
int getScale
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The decimal column's scale, as an integer. The scale of a decimal column represents the number of digits that can be stored following the decimal point. It is possible for this value to be `0`. For more information, see the discussion of the `DECIMAL` datatype in Numeric Types, in the MySQL Manual.

2.3.1.2.10. `Column::getLength()`

Description. This method gets the length of a column. This is either the array length for the column or — for a variable length array — the maximum length.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
int getLength
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The (maximum) array length of the column, as an integer.

2.3.1.2.11. `Column::getCharset()`

Description. This gets the character set used by a text column.

> **Note**
>
> This method is applicable only to columns whose `Type` value is `Char`, `Varchar`, or `Text`.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
CHARSET_INFO* getCharset
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to a `CHARSET_INFO` structure specifying both character set and collation. This is the same as a MySQL `MY_CHARSET_INFO` data structure; for more information, see `mysql_get_character_set_info()`,in the MySQL Manual.

2.3.1.2.12. `Column::getInlineSize()`

Description. This method retrieves the inline size of a blob column — that is, the number of initial bytes to store in the table's blob attribute. This part is normally in main memory and can be indexed.

> **Note**
>
> This method is applicable only to blob columns.

Signature.

```
int getInlineSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The blob column's inline size, as an integer.

2.3.1.2.13. `Column::getPartSize()`

Description. This method is used to get the part size of a blob column — that is, the number of bytes that are stored in each tuple of the blob table.

> **Note**
>
> This method is applicable to blob columns only.

Signature.

```
int getPartSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's part size, as an integer. In the case of a `Tinyblob` column, this value is `0` (that is, only inline bytes are stored).

2.3.1.2.14. `Column::getStripeSize()`

Description. This method gets the stripe size of a blob column — that is, the number of consecutive parts to store in each node group.

Signature.

```
int getStripeSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's stripe size, as an integer.

2.3.1.2.15. `Column::getSize()`

Description. This function is used to obtain the size of a column.

> **Important**
>
> The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong letter-case for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
int getSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The column's size in bytes (an integer value).

2.3.1.2.16. `Column::getPartitionKey()`

Description. This method is used to check whether the column is part of the table's partitioning key.

> **Note**
>
> A *partitioning key* is a set of attributes used to distribute the tuples onto the `NDB` nodes. This key a hashing function specific to the `NDBCLUSTER` storage engine.
>
> An example where this would be useful is an inventory tracking application involving multiple warehouses and regions, where it might be good to use the warehouse ID and district id as the partition key. This would place all data for a specific district and warehouse in the same database node. Locally to each fragment the full primary key will still be used with the hashing algorithm in such a case.
>
> For more information about partitioning, partitioning schemes, and partitioning keys in MySQL 5.1, see Partitioning, in the MySQL Manual.

> **Important**
>
> The only type of user-defined partitioning that is supported for use with the `NDBCLUSTER` storage engine in MySQL 5.1 is key partitioning.

Signature.

```
bool getPartitionKey
    (
      void
    ) const
```

Parameters. *None*.

Return Value. `true` if the column is part of the partitioning key for the table, otherwise `false`.

2.3.1.2.17. `Column::getArrayType()`

Description. This method gets the column's array type.

Signature.

```
ArrayType getArrayType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `ArrayType`; see Section 2.3.1.1.1, “The `Column::ArrayType` Type” for possible values.

2.3.1.2.18. `Column::getStorageType()`

Description. This method obtains a column's storage type.

Signature.

```
StorageType getStorageType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `StorageType` value; for more information about this type, see Section 2.3.1.1.2, "The `Column::StorageType` Type".

2.3.1.2.19. `Column::setName()`

Description. This method is used to set the name of a column.

> **Important**
>
> `setName()` is the only `Column` method whose result is visible from a MySQL Server. MySQL cannot see any other changes made to existing columns using the NDB API.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. This method takes a single argument — the new name for the column.

Return Value. This method *None*.

2.3.1.2.20. `Column::setNullable()`

Description. This method allows you to toggle the nullability of a column.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setNullable
    (
      bool nullable
    )
```

Parameters. A Boolean value. Using `true` makes it possible to insert `NULL`s into the column; if *`nullable`* is `false`, then this method performs the equivalent of changing the column to `NOT NULL` in MySQL.

Return Value. *No return value*.

2.3.1.2.21. `Column::setPrimaryKey()`

Description. This method is used to make a column part of the table's primary key, or to remove it from the primary key.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.
>
> The NDB API handles column names in case-sensitive fashion; "myColumn" and "Mycolumn" are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
void setPrimaryKey
    (
      bool primary
    )
```

Parameters. This method takes a single Boolean value. If it is `true`, then the column becomes part of the table's primary key; if `false`, then the column is removed from the primary key.

Return Value. *No return value*.

2.3.1.2.22. `Column::setType()`

Description. This method sets the `Type` (datatype) of a column.

> **Important**
>
> `setType()` resets *all* column attributes to their (type dependent) default values; it should be the first method that you call when changing the attributes of a given column.
>
> Changes made to columns using this method are not visible to MySQL.
>
> The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong letter-case for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
void setType
    (
      Type type
    )
```

Parameters. This method takes a single parameter — the new `Column::Type` for the column. The default is `Unsigned`. For a listing of all permitted values, see Section 2.3.1.1.3, “`Column::Type`”.

Return Value. *No return value.*

2.3.1.2.23. `Column::setPrecision()`

Description. This method can be used to set the precision of a decimal column.

> **Important**
>
> This method is applicable to decimal columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPrecision
    (
      int precision
    )
```

Parameters. This method takes a single parameter — precision is an integer, the value of the column's new precision. For additional information about decimal precision and scale, see Section 2.3.1.2.8, “`Column::getPrecision()`”, and Section 2.3.1.2.9, “`Column::getScale()`”.

Return Value. *No return value.*

2.3.1.2.24. `Column::setScale()`

Description. This method can be used to set the scale of a decimal column.

> **Important**
>
> This method is applicable to decimal columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setScale
    (
      int scale
    )
```

Parameters. This method takes a single parameter — the integer *`scale`* is the new scale for the decimal column. For additional information about decimal precision and scale, see Section 2.3.1.2.8, “`Column::getPrecision()`”, and Section 2.3.1.2.9, “`Column::getScale()`”.

Return Value. *No return value.*

2.3.1.2.25. `Column::setLength()`

Description. This method sets the length of a column. For a variable-length array, this is the maximum length; otherwise it is the array length.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.
>
> The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong letter-case by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
void setLength
    (
      int length
    )
```

Parameters. This method takes a single argument — the integer value *`length`* is the new length for the column.

Return Value. *No return value.*

2.3.1.2.26. `Column::setCharset()`

Description. This method can be used to set the character set and collation of a `Char`, `Varchar`, or `Text` column.

> **Important**
>
> This method is applicable to `Char`, `Varchar`, and `Text` columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setCharset
    (
      CHARSET_INFO* cs
    )
```

Parameters. This method takes one parameter. *`cs`* is a pointer to a `CHARSET_INFO` structure. For additional information, see Section 2.3.1.2.11, “`Column::getCharset()`”.

Return Value. *No return value.*

2.3.1.2.27. `Column::setInlineSize`

Description. This method gets the inline size of a `BLOB` column — that is, the number of initial bytes to store in the table's blob attribute. This part is normally kept in main memory, and can be indexed and interpreted.

> **Important**
>
> This method is applicable to `BLOB` columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setInlineSize
    (
      int size
    )
```

Parameters. The integer *`size`* is the new inline size for the `BLOB` column.

Return Value. *No return value.*

2.3.1.2.28. `Column::setPartSize()`

Description. This method sets the part size of a `BLOB` column — that is, the number of bytes to store in each tuple of the `BLOB` table.

> **Important**
>
> This method is applicable to `BLOB` columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPartSize
    (
      int size
    )
```

Parameters. The integer *`size`* is the number of bytes to store in the `BLOB` table. Using zero for this value allows only inline bytes to be stored, in effect making the column's type `TINYBLOB`.

Return Value. *No return value*.

2.3.1.2.29. `Column::setStripeSize()`

Description. This method sets the stripe size of a `BLOB` column — that is, the number of consecutive parts to store in each node group.

> **Important**
>
> This method is applicable to `BLOB` columns only.
>
> Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setStripeSize
    (
      int size
    )
```

Parameters. This method takes a single argument. The integer *`size`* is the new stripe size for the column.

Return Value. *No return value*.

2.3.1.2.30. `Column::setPartitionKey()`

Description. This method makes it possible to add a column to the partitioning key of the table to which it belongs, or to remove the column from the table's partitioning key.

> **Important**
>
> Changes made to columns using this method are not visible to MySQL.
>
> For additional information, see Section 2.3.1.2.16, “`Column::getPartitionKey()`”.

Signature.

```
void setPartitionKey
    (
      bool enable
    )
```

Parameters. The single parameter *`enable`* is a Boolean value. Passing `true` to this method makes the column part of the table's partitioning key; if *`enable`* is `false`, then the column is removed from the partitioning key.

Return Value. *No return value*.

2.3.1.2.31. `Column::setArrayType()`

Description. Sets the array type for the column.

Signature.

```
void setArrayType
    (
      ArrayType type
    )
```

Parameters. A `Column::ArrayType` value. See Section 2.3.1.1.1, "The `Column::ArrayType` Type", for more information.

Return Value. *None*.

2.3.1.2.32. `Column::setStorageType()`

Description. Sets the storage type for the column.

Signature.

```
void setStorageType
    (
      StorageType type
    )
```

Parameters. A `Column::StorageType` value. See Section 2.3.1.1.2, "The `Column::StorageType` Type", for more information.

Return Value. *None*.

2.3.2. The `Datafile` Class

This section covers the `Datafile` class.

Parent class. Object

Child classes. *None*

Description. The `Datafile` class models a Cluster Disk Data datafile, which is used to store Disk Data table data.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support datafiles; thus the `Datafile` class is unavailable for NDB API applications written against these MySQL versions.

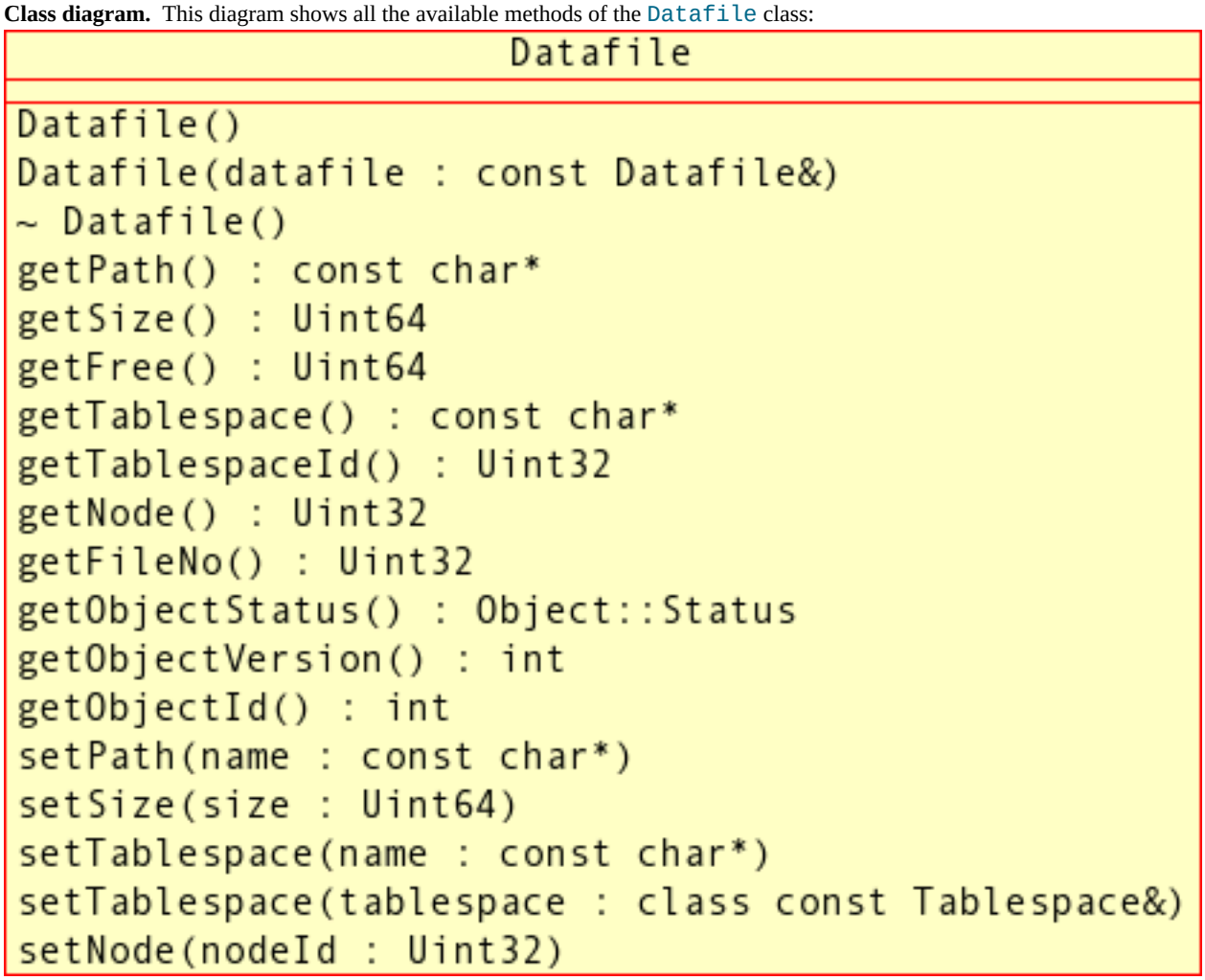
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Datafile()`	Class constructor
`~Datafile()`	Destructor
`getPath()`	Gets the file system path to the datafile
`getSize()`	Gets the size of the datafile
`getFree()`	Gets the amount of free space in the datafile
`getNode()`	Gets the ID of the node where the datafile is located
`getTablespace()`	Gets the name of the tablespace to which the datafile belongs
`getTablespaceId()`	Gets the ID of the tablespace to which the datafile belongs
`getFileNo()`	Gets the number of the datafile in the tablespace
`getObjectStatus()`	Gets the datafile's object status
`getObjectVersion()`	Gets the datafile's object version
`getObjectId()`	Gets the datafile's object ID
`setPath()`	Sets the name and location of the datafile on the file system
`setSize()`	Sets the datafile's size
`setTablespace()`	Sets the tablespace to which the datafile belongs
`setNode()`	Sets the Cluster node where the datafile is to be located

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.2.1, "`Datafile` Methods".

Types. The `Datafile` class defines no public types.

Class diagram. This diagram shows all the available methods of the `Datafile` class:



2.3.2.1. `Datafile` Methods

This section provides descriptions of the public methods of the `Datafile` class.

2.3.2.1.1. `Datafile` Class Constructor

Description. This method creates a new instance of `Datafile`, or a copy of an existing one.

Signature. To create a new instance:

```
Datafile
    (
      void
    )
```

To create a copy of an existing `Datafile` instance:

```
Datafile
    (
      const Datafile& datafile
    )
```

Parameters. New instance: *None*. Copy constructor: a reference to the `Datafile` instance to be copied.

Return Value. A `Datafile` object.

2.3.2.1.2. `Datafile::getPath()`

Description. This method returns the file system path to the datafile.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. *None.*

Return Value. The path to the datafile on the data node's file system, a string (character pointer).

2.3.2.1.3. `Datafile::getSize()`

Description. This method gets the size of the datafile in bytes.

Signature.

```
Uint64 getSize
    (
      void
    ) const
```

Parameters. *None.*

Return Value. The size of the data file, in bytes, as an unsigned 64-bit integer.

2.3.2.1.4. `Datafile::getFree()`

Description. This method gets the free space available in the datafile.

Signature.

```
Uint64 getFree
    (
      void
    ) const
```

Parameters. *None.*

Return Value. The number of bytes free in the datafile, as an unsigned 64-bit integer.

2.3.2.1.5. `Datafile::getTablespace()`

Description. This method can be used to obtain the name of the tablespace to which the datafile belongs.

> **Note**
>
> You can also access the associated tablespace's ID directly. See Section 2.3.2.1.6, "`Datafile::getTablespaceId()`".

Signature.

```
const char* getTablespace
    (
      void
    ) const
```

Parameters. *None.*

Return Value. The name of the associated tablespace (as a character pointer).

2.3.2.1.6. `Datafile::getTablespaceId()`

Description. This method gets the ID of the tablespace to which the datafile belongs.

> **Note**
>
> You can also access the name of the associated tablespace directly. See Section 2.3.2.1.5, "`Datafile::getTablespace()`".

Signature.

```
Uint32 getTablespaceId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The is method returns the tablespace ID as an unsigned 32-bit integer.

2.3.2.1.7. `Datafile::getNode()`

Description. This method retrieves the ID of the Cluster node on which the datafile resides.

Signature.

```
Uint32 getNode
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The node ID as an unsigned 32-bit integer.

2.3.2.1.8. `Datafile::getFileNo()`

Description. This method gets the number of the file within the associated tablespace.

Signature.

```
Uint32 getFileNo
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The file number, as an unsigned 32-bit integer.

2.3.2.1.9. `Datafile::getObjectStatus()`

Description. This method is used to obtain the datafile's object status.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The datafile's `Status`. See Section 2.3.20.1.3, “The `Object::Status` Type”.

2.3.2.1.10. `Datafile::getObjectVersion()`

Description. This method retrieves the datafile's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The datafile's object version, as an integer.

2.3.2.1.11. `Datafile::getObjectId()`

Description. This method is used to obtain the object ID of the datafile.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The datafile's object ID, as an integer.

2.3.2.1.12. `Datafile::setPath()`

Description. This method sets the path to the datafile on the data node's file system.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. The path to the file, a string (as a character pointer).

Return Value. *None*.

2.3.2.1.13. `Datafile::setSize()`

Description. This method sets the size of the datafile.

Signature.

```
void setSize
    (
      Uint64 size
    )
```

Parameters. This method takes a single parameter — the desired *`size`* in bytes for the datafile, as an unsigned 64-bit integer.

Return Value. *None*.

2.3.2.1.14. `Datafile::setTablespace()`

Description. This method is used to associate the datafile with a tablespace.

Signatures. `setTablespace()` can be invoked with either the name of the tablespace, as shown here:

```
void setTablespace
    (
      const char* name
    )
```

Or with a reference to a `Tablespace` object.

```
void setTablespace
    (
      const class Tablespace& tablespace
    )
```

Parameters. This method takes a single parameter, which can be either one of the following:

- The *`name`* of the tablespace (as a character pointer).
- A reference *`tablespace`* to the corresponding `Tablespace` object.

Return Value. *None*.

2.3.2.1.15. `Datafile::setNode()`

Description. Designates the node to which this datafile belongs.

Signature.

```
void setNode
    (
      Uint32 nodeId
    )
```

Parameters. The *`nodeId`* of the node on which the datafile is to be located (an unsigned 32-bit integer value).

Return Value. *None*.

2.3.3. The `Dictionary` Class

This section describes the `Dictionary` class.

Parent class. NdbDictionary

Child classes. List

Description. This is used for defining and retrieving data object metadata. It also includes methods for creating and dropping database objects.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Dictionary()`	Class constructor method
`~Dictionary()`	Destructor method
`getTable()`	Gets the table having the given name
`getIndex()`	Gets the index having the given name
`getEvent()`	Gets the event having the given name
`getTablespace()`	Gets the tablespace having the given name
`getLogfileGroup()`	Gets the logfile group having the given name
`getDatafile()`	Gets the datafile having the given name
`getUndofile()`	Gets the undofile having the given name
`getNdbError()`	Retrieves the latest error
`createTable()`	Creates a table
`createIndex()`	Creates an index
`createEvent()`	Creates an event
`createTablespace()`	Creates a tablespace
`createLogfileGroup()`	Creates a logfile group
`createDatafile()`	Creates a datafile
`createUndofile()`	Creates an undofile
`dropTable()`	Drops a table
`dropIndex()`	
`dropEvent()`	Drops an index
`dropTablespace()`	Drops a tablespace
`dropLogfileGroup()`	Drops a logfile group
`dropDatafile()`	Drops a datafile
`dropUndofile()`	Drops an undofile
`listObjects()`	Fetches a list of the objects in the dictionary
`listIndexes()`	Fetches a list of the indexes defined on a given table
`listEvents()`	Fetches a list of the events defined in the dictionary
`removeCachedTable()`	Removes a table from the local cache
`removeCachedIndex()`	Removes an index from the local cache

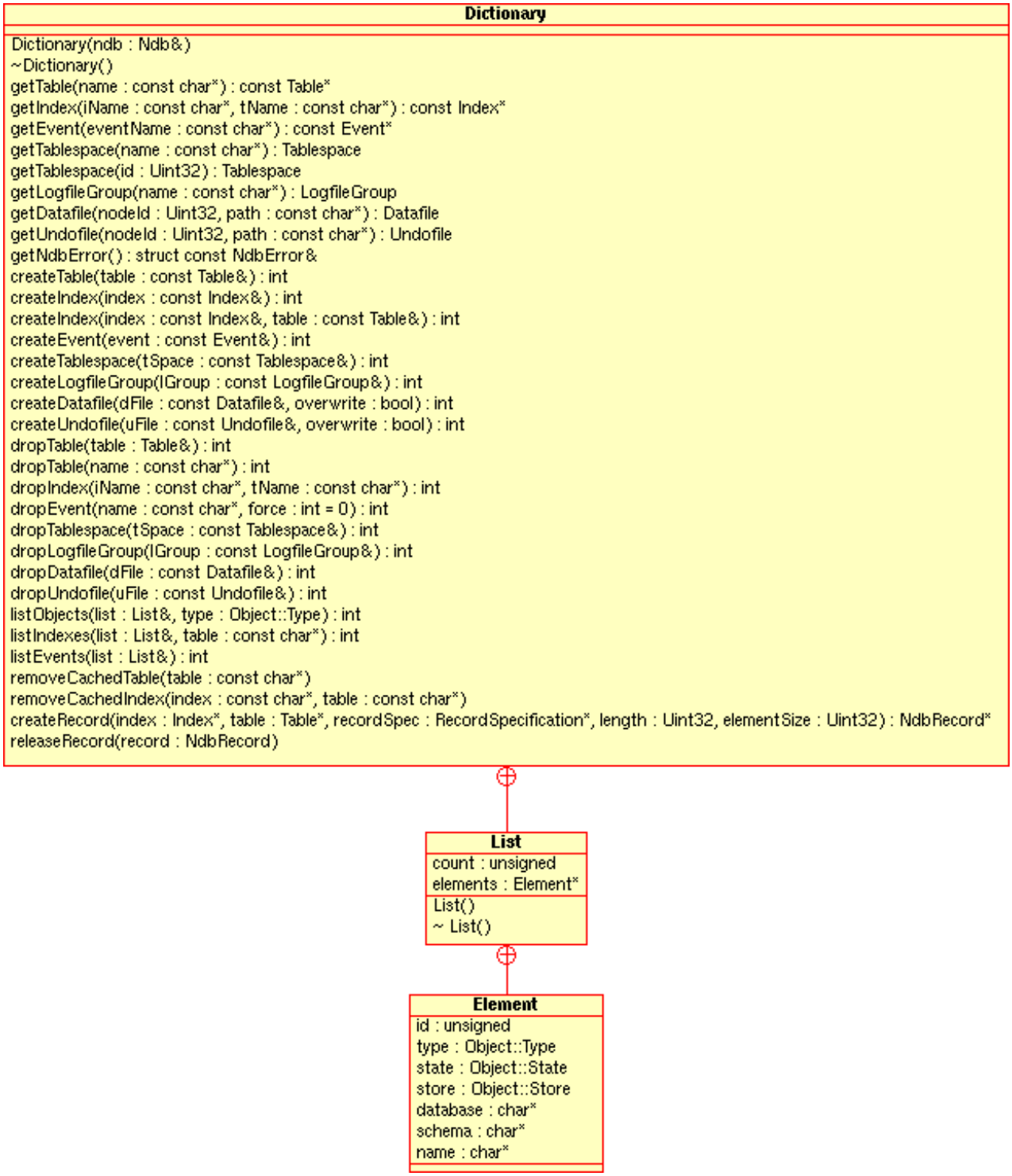
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.3.1, “`Dictionary` Methods”.

> **Important**
>
> Objects created using the `Dictionary::create*()` methods are not visible from the MySQL Server. For this reason, it is usually preferable to avoid using them.

> **Note**
>
> The Dictionary class does not have any methods for working directly with columns. You must use `Column` class methods for this purpose — see Section 2.3.1, "The `Column` Class", for details.

Types. See Section 2.3.7, "The `List` Class", and Section 2.3.27, "The `Element` Structure".

`Dictionary` Class and Subclass Diagram. This diagram shows all the public members of the `Dictionary` class and its subclasses:



2.3.3.1. `Dictionary` Methods

This section details all of the public methods of the `Dictionary` class.

2.3.3.1.1. `Dictionary` Class Constructor

Description. This method creates a new instance of the `Dictionary` class.

> **Note**
>
> Both the constructor and destructor for this class are protected methods, rather than public.

Signature.

```
protected Dictionary
    (
      Ndb& ndb
    )
```

Parameters. An `Ndb` object. See Section 2.3.8, "The `Ndb` Class".

Return Value. A `Dictionary` object.

Destructor. The destructor takes no parameters and returns nothing.

```
protected ~Dictionary
    (
      void
    )
```

2.3.3.1.2. `Dictionary::getTable()`

Description. This method can be used to access the table with a known name. See Section 2.3.21, "The `Table` Class".

Signature.

```
const Table* getTable
    (
      const char* name
    ) const
```

Parameters. The *`name`* of the table.

Return Value. A pointer to the table, or `NULL` if there is no table with the *`name`* supplied.

2.3.3.1.3. `Dictionary::getIndex()`

Description. This method retrieves a pointer to an index, given the name of the index and the name of the table to which the table belongs.

Signature.

```
const Index* getIndex
    (
      const char* iName,
      const char* tName
    ) const
```

Parameters. Two parameters are required:

- The name of the index (*`iName`*)
- The name of the table to which the index belongs (*`tName`*)

Both are string values, represented by character pointers.

Return Value. A pointer to an `Index`. See Section 2.3.5, "The `Index` Class", for information about this object.

2.3.3.1.4. `Dictionary::getEvent()`

Description. This method is used to obtain an `Event` object, given the event's name.

Signature.

```
const Event* getEvent
    (
      const char* eventName
    )
```

Parameters. The *`eventName`*, a string (character pointer).

Return Value. A pointer to an `Event` object. See Section 2.3.4, “The `Event` Class”, for more information.

2.3.3.1.5. `Dictionary::getTablespace()`

Description. Given either the name or ID of a tablespace, this method returns the corresponding `Tablespace` object.

Signatures. Using the tablespace name:

```
Tablespace getTablespace
    (
      const char* name
    )
```

Using the tablespace ID:

```
Tablespace getTablespace
    (
      Uint32 id
    )
```

Parameters. Either one of the following:

- The *`name`* of the tablespace, a string (as a character pointer)
- The unsigned 32-bit integer *`id`* of the tablespace

Return Value. A `Tablespace` object, as discussed in Section 2.3.22, “The `Tablespace` Class”.

2.3.3.1.6. `Dictionary::getLogfileGroup()`

Description. This method gets a `LogfileGroup` object, given the name of the logfile group.

Signature.

```
LogfileGroup getLogfileGroup
    (
      const char* name
    )
```

Parameters. The *`name`* of the logfile group.

Return Value. An instance of `LogfileGroup`; see Section 2.3.6, “The `LogfileGroup` Class”, for more information.

2.3.3.1.7. `Dictionary::getDatafile()`

Description. This method is used to retrieve a `Datafile` object, given the node ID of the data node where a datafile is located and the path to the datafile on that node's file system.

Signature.

```
Datafile getDatafile
    (
      Uint32      nodeId,
      const char* path
    )
```

Parameters. This method must be invoked using two arguments, as shown here:

- The 32-bit unsigned integer *`nodeId`* of the data node where the datafile is located
- The *`path`* to the datafile on the node's file system (string as character pointer)

Return Value. A `Datafile` object — see Section 2.3.2, “The `Datafile` Class”, for details.

2.3.3.1.8. `Dictionary::getUndofile()`

Description. This method gets an `Undofile` object, given the ID of the node where an undofile is located and the file system path to the file.

Signature.

```
Undofile getUndofile
    (
      Uint32      nodeId,
      const char* path
    )
```

Parameters. This method requires the following two arguments:

- The *`nodeId`* of the data node where the undofile is located; this value is passed as a 32-bit unsigned integer
- The *`path`* to the undofile on the node's file system (string as character pointer)

Return Value. An instance of `Undofile`. For more information, see Section 2.3.23, “The `Undofile` Class”.

2.3.3.1.9. `Dictionary::getNdbError()`

Description. This method retrieves the most recent `NDB` API error.

Signature.

```
const struct NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an `NdbError` object. See Section 2.3.30, “The `NdbError` Structure”.

2.3.3.1.10. `Dictionary::createTable()`

Description. Creates a table given an instance of `Table`.

Signature.

```
int createTable
    (
      const Table& table
    )
```

Parameters. An instance of `Table`. See Section 2.3.21, “The `Table` Class”, for more information.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.11. `Dictionary::createIndex()`

Description. This method creates an index given an instance of `Index` and possibly an optional instance of `Table`.

Signature.

```
int createIndex
    (
      const Index& index
    )
```

```
int createIndex
    (
      const Index& index,
      const Table& table
    )
```

Parameters. *Required*: A reference to an `Index` object. *Optional*: A reference to a `Table` object.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.12. `Dictionary::createEvent()`

Description. Creates an event, given a reference to an `Event` object.

Signature.

```
int createEvent
    (
      const Event& event
    )
```

Parameters. A reference *`event`* to an `Event` object.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.13. `Dictionary::createTablespace()`

Description. This method creates a new tablespace, given a `Tablespace` object.

Signature.

```
int createTablespace
    (
      const Tablespace& tSpace
    )
```

Parameters. This method requires a single argument — a reference to an instance of `Tablespace`.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.14. `Dictionary::createLogfileGroup()`

Description. This method creates a new logfile group, given an instance of `LogfileGroup`.

Signature.

```
int createLogfileGroup
    (
      const LogfileGroup& lGroup
    )
```

Parameters. A single argument, a reference to a `LogfileGroup` object, is required.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.15. `Dictionary::createDatafile()`

Description. This method creates a new datafile, given a `Datafile` object.

Signature.

```
int createDatafile
    (
      const Datafile& dFile
    )
```

Parameters. A single argument — a reference to an instance of `Datafile` — is required.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.16. `Dictionary::createRecord()`

Description. This method is used to create an `NdbRecord` object for use in table or index scanning operations. (See Section 2.3.25, “The `NdbRecord` Interface”.)

`Dctionary::createRecord()` is available beginning with MySQL Cluster NDB 6.2.3.

Signature. To create an `NdbRecord` for use in table operations:

```
NdbRecord* createRecord
    (
```

```
      const Table* table,
      const RecordSpecification* recordSpec,
      Uint32 length,
      Uint32 elementSize
    )
```

To create an `NdbRecord` for use in index operations, you can use either of the following:

```
NdbRecord* createRecord
    (
      const Index* index,
      const Table* table,
      const RecordSpecification* recordSpec,
      Uint32 length,
      Uint32 elementSize
    )
```

or

```
NdbRecord* createRecord
    (
      const Index* index,
      const RecordSpecification* recordSpec,
      Uint32 length,
      Uint32 elementSize
    )
```

Parameters. `Dictionary::createRecord()` takes the following parameters:

- If this `NdbRecord` is to be used with an index, a pointer to the corresponding `Index` object. If the `NdbRecord` is to be used with a table, this parameter is omitted. (See Section 2.3.5, “The `Index` Class”.)
- A pointer to a `Table` object representing the table to be scanned. If the `Ndbrecord` produced is to be used with an index, then this optionally specifies the table containing that index. (See Section 2.3.21, “The `Table` Class”.)
- A `RecordSpecification` used to describe a column. (See Section 2.3.32, “The `RecordSpecification` Structure”.
- The *`length`* of the record.
- The size of the elements making up this record.

Return Value. An `NdbRecord` for use in operations involving the given table or index.

Example. See Section 2.3.25, “The `NdbRecord` Interface”.

2.3.3.1.17. `Dictionary::createUndofile()`

Description. This method creates a new undofile, given an `Undofile` object.

Signature.

```
int createUndofile
    (
      const Undofile& uFile
    )
```

Parameters. This method requires one argument: a reference to an instance of `Undofile`.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.18. `Dictionary::dropTable()`

Description. Drops a table given an instance of `Table`.

Signature.

```
int dropTable
    (
      const Table& table
    )
```

Parameters. An instance of `Table`. See Section 2.3.21, “The `Table` Class”, for more information.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.19. `Dictionary::dropIndex()`

Description. This method drops an index given an instance of `Index` and possibly an optional instance of `Table`.

Signature.

```
int dropIndex
    (
      const Index& index
    )
```

```
int dropIndex
    (
      const Index& index,
      const Table& table
    )
```

Parameters.

- **Required.** A reference to an `Index` object.
- **Optional.** A reference to a `Table` object.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.20. `Dictionary::dropEvent()`

Description. This method drops an event, given a reference to an `Event` object.

Signature.

```
int dropEvent
    (
      const char* name,
      int         force = 0
    )
```

Parameters. This method takes two parameters:

- The *`name`* of the event to be dropped, as a string.
- By default, `dropEvent()` fails if the event specified does not exist. You can override this behavior by passing any nonzero value for the (optional) *`force`* argument; in this case no check is made as to whether there actually is such an event, and an error is returned only if the event exists but it was for whatever reason not possible to drop it.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.21. `Dictionary::dropTablespace()`

Description. This method drops a tablespace, given a `Tablespace` object.

Signature.

```
int dropTablespace
    (
      const Tablespace& tSpace
    )
```

Parameters. This method requires a single argument — a reference to an instance of `Tablespace`.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.22. `Dictionary::dropLogfileGroup()`

Description. This method drops a logfile group, given an instance of `LogfileGroup`.

Signature.

```
int dropLogfileGroup
    (
```

```
      const LogfileGroup& lGroup
    )
```

Parameters. A single argument, a reference to a `LogfileGroup` object, is required.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.23. `Dictionary::dropDatafile()`

Description. This method drops a datafile, given a `Datafile` object.

Signature.

```
int dropDatafile
    (
      const Datafile& dFile
    )
```

Parameters. A single argument — a reference to an instance of `Datafile` — is required.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.24. `Dictionary::dropUndofile()`

Description. This method drops an undofile, given an `Undofile` object.

Signature.

```
int dropUndofile
    (
      const Undofile& uFile
    )
```

Parameters. This method requires one argument: a reference to an instance of `Undofile`.

Return Value. `0` on success, `-1` on failure.

2.3.3.1.25. `DIctionary::invalidateTable()`

Description. This method is used to invalidate a cached table object.

Signature.

```
void invalidateTable
    (
      const char* name
    )
```

Parameters. The *`name`* of the table to be removed from the table cache.

Return Value. *None*.

2.3.3.1.26. `Dictionary::listObjects()`

Description. This method is used to obtain a list of objects in the dictionary. It is possible to get all of the objects in the dictionary, or to restrict the list to objects of a single type.

Signatures.

```
int listObjects
    (
      List&       list,
      Object::Type type = Object::TypeUndefined
    ) const
```

or

```
int listObjects
    (
      List&       list,
      Object::Type type = Object::TypeUndefined
    )
```

Parameters. A reference to a `List` object is required — this is the list that contains the dictionary's objects after `listOb-`

jects() is called. (See Section 2.3.7, “The List Class”.) An optional second argument *type* may be used to restrict the list to only those objects of the given type — that is, of the specified Object::Type. (See Section 2.3.20.1.5, “The Object::Type Type”.) If *type* is not given, then the list contains all of the dictionary's objects.

Return Value. 0 on success, -1 on failure.

2.3.3.1.27. Dictionary::listIndexes()

Description. This method is used to obtain a List of all the indexes on a table, given the table's name. (See Section 2.3.7, “The List Class”.)

Signature.

```
int listIndexes
    (
      List&      list,
      const char* table
) const
```

or

```
int listIndexes
    (
      List&      list,
      const char* table
    )
```

Parameters. listIndexes() takes two arguments:

- A reference to the List that contains the indexes following the call to the method
- The name of the *table* whose indexes are to be listed

Both of these arguments are required.

Return Value. 0 on success, -1 on failure.

2.3.3.1.28. Dictionary::listEvents()

Description. This method returns a list of all events defined within the dictionary.

This method was added in MySQL Cluster NDB 6.1.13.

Signature.

```
int listEvents
    (
      List& list
    )
```

or

```
int listEvents
    (
      List& list
    ) const
```

Parameters. A reference to a List object. (See Section 2.3.7, “The List Class”.)

Return Value. 0 on success; -1 on failure.

2.3.3.1.29. Dictionary::releaseRecord()

Description. This method is used to free an NdbRecord after it is no longer needed.

Signature.

```
void releaseRecord
    (
      NdbRecord* record
    )
```

Parameters. The NdbRecord to be cleaned up.

Return Value. *None*.

Example. See Section 2.3.25, “The `NdbRecord` Interface”.

2.3.3.1.30. `Dictionary::removeCachedTable()`

Description. This method removes the specified table from the local cache.

Signature.

```
void removeCachedTable
    (
      const char* table
    )
```

Parameters. The name of the *`table`* to be removed from the cache.

Return Value. *None*.

2.3.3.1.31. `Dictionary::removeCachedIndex()`

Description. This method removes the specified index from the local cache.

Signature.

```
void removeCachedIndex
    (
      const char* index,
      const char* table
    )
```

Parameters. The `removeCachedIndex()` requires two arguments:

- The name of the *`index`* to be removed from the cache
- The name of the *`table`* in which the index is found

Return Value. *None*.

2.3.4. The `Event` Class

This section discusses the `Event` class, its methods and defined types.

Parent class. NdbDictionary

Child classes. *None*

Description. This class represents a database event in a MySQL Cluster.

Methods. The following table lists the public methods of the `Event` class and the purpose or use of each method:

Method	Purpose / Use
`Event()`	Class constructor
`~Event()`	Destructor
`getName()`	Gets the event's name
`getTable()`	Gets the `Table` object on which the event is defined
`getTableName()`	Gets the name of the table on which the event is defined
`getTableEvent()`	Checks whether an event is to be detected
`getDurability()`	Gets the event's durability
`getReport()`	Gets the event's reporting options
`getNoOfEventColumns()`	Gets the number of columns for which an event is defined
`getEventColumn()`	Gets a column for which an event is defined
`getObjectStatus()`	Gets the event's object status
`getObjectVersion()`	Gets the event's object version
`getObjectId()`	Gets the event's object ID

Method	Purpose / Use
`setName()`	Sets the event's name
`setTable()`	Sets the `Table` object on which the event is defined
`addTableEvent()`	Adds the type of event that should be detected
`setDurability()`	Sets the event's durability
`setReport()`	The the event's reporting options
`addEventColumn()`	Adds a column on which events should be detected
`addEventColumns()`	Adds multiple columns on which events should be detected
`mergeEvents()`	Stes the event's merge flag

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.4.2, “`Event` Methods”.

Types. These are the public types of the `Event` class:

Type	Purpose / Use
`TableEvent`	Represents the type of a table event
`EventDurability`	Specifies an event's scope, accessibility, and lifetime
`EventReport`	Specifies the reporting option for a table event

For a discussion of each of these types, along with its possible values, see Section 2.3.4.1, “`Event` Types”.

Class diagram. This diagram shows all the available methods and enumerated types of the `Event` class:

2.3.4.1. `Event` Types

This section details the public types belonging to the `Event` class.

2.3.4.1.1. The `Event::TableEvent` Type

This section describes `TableEvent`, a type defined by the `Event` class.

Description. `TableEvent` is used to classify the types of events that may be associated with tables in the NDB API.

Enumeration values.

Value	Description
TE_INSERT	Insert event on a table
TE_DELETE	Delete event on a table
TE_UPDATE	Update event on a table
TE_DROP	Occurs when a table is dropped
TE_ALTER	Occurs when a table definition is changed
TE_CREATE	Occurs when a table is created
TE_GCP_COMPLETE	Occurs on Cluster failures
TE_CLUSTER_FAILURE	Occurs on the completion of a global checkpoint
TE_STOP	Occurs when an event operation is stopped
TE_NODE_FAILURE	Occurs when a Cluster node fails
TE_SUBSCRIBE	Occurs when a cluster node subscribes to an event
TE_UNSUBSCRIBE	Occurs when a cluster node unsubscribes from an event
TE_ALL	Occurs when any event occurs on a table (not relevant when a specific event is received)

2.3.4.1.2. The `Event::EventDurability` Type

This section discusses `EventDurability`, a type defined by the `Event` class.

Description. The values of this type are used to describe an event's lifetime or persistence as well as its scope.

Enumeration values.

Value	Description
ED_UNDEFINED	The event is undefined or of an unsupported type.
ED_SESSION	This event persists only for the duration of the current session, and is available only to the current application. It is deleted after the application disconnects or following a cluster restart. **Important** The value `ED_SESSION` is reserved for future use and is not yet supported in any MySQL Cluster release.
ED_TEMPORARY	Any application may use the event, but it is deleted following a cluster restart. **Important** The value `ED_TEMPORARY` is reserved for future use and is not yet supported in any MySQL Cluster release.
ED_PERMANENT	Any application may use the event, and it persists until deleted by an application — even following a cluster. restart

2.3.4.1.3. The `Event::EventReport` Type

This section discusses `EventReport`, a type defined by the `Event` class.

Description. The values of this type are used to specify reporting options for table events.

Enumeration values.

Value	Description
ER_UPDATED	Reporting of update events
ER_ALL	Reporting of all events, except for those not resulting in any updates to the inline parts of `BLOB` columns
ER_SUBSCRIBE	Reporting of subscription events

2.3.4.2. `Event` Methods

2.3.4.2.1. `Event` Constructor

Description. The `Event` constructor creates a new instance with a given name, and optionally associated with a table.

Signatures. Name only:

```
Event
    (
      const char* name
    )
```

Name and associated table:

```
Event
    (
      const char*                  name,
      const NdbDictionary::Table& table
    )
```

Parameters. At a minimum, a *`name`* (as a constant character pointer) for the event is required. Optionally, an event may also be associated with a table; this argument, when present, is a reference to a `Table` object (see Section 2.3.21, “The `Table` Class”).

Return Value. A new instance of `Event`.

Destructor. A destructor for this class is supplied as a virtual method which takes no arguments and whose return type is `void`.

2.3.4.2.2. `Event::getName()`

Description. This method obtains the name of the event.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the event, as a character pointer.

2.3.4.2.3. `Event::getTable()`

Description. This method is used to find the table with which an event is associated. It returns a reference to the corresponding `Table` object. You may also obtain the name of the table directly using getTableName(). (For details, see Section 2.3.4.2.4, “`Event::getTableName()`”.)

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The table with which the event is associated — if there is one — as a pointer to a `Table` object; otherwise, this method returns `NULL`. (See Section 2.3.21, “The `Table` Class”.)

2.3.4.2.4. `Event::getTableName()`

Description. This method obtains the name of the table with which an event is associated, and can serve as a convenient alternative to `getTable()`. (See Section 2.3.4.2.3, “`Event::getTable()`”.)

Signature.

```
const char* getTableName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table associated with this event, as a character pointer.

2.3.4.2.5. `Event::getTableEvent()`

Description. This method is used to check whether a given table event will be detected.

Signature.

```
bool getTableEvent
    (
      const TableEvent te
    ) const
```

Parameters. This method takes a single parameter - the table event's type, that is, a `TableEvent` value. See Section 2.3.4.1.1, "The `Event::TableEvent` Type", for the list of possible values.

Return Value. This method returns `true` if events of `TableEvent` type *`te`* will be detected. Otherwise, the return value is `false`.

2.3.4.2.6. `Event::getDurability()`

Description. This method gets the event's lifetime and scope (that is, its `EventDurability`).

Signature.

```
EventDurability getDurability
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `EventDurability` value. See Section 2.3.4.1.2, "The `Event::EventDurability` Type", for possible values.

2.3.4.2.7. `Event::getReport()`

Description. This method is used to obtain the reporting option in force for this event.

Signature.

```
EventReport getReport
    (
      void
    ) const
```

Parameters. *None*.

Return Value. One of the reporting options specified in Section 2.3.4.1.3, "The `Event::EventReport` Type".

2.3.4.2.8. `Event::getNoOfEventColumns()`

Description. This method obtains the number of columns on which an event is defined.

Signature.

```
int getNoOfEventColumns
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of columns (as an integer), or `-1` in the case of an error.

2.3.4.2.9. `Event::getEventColumn()`

Description. This method is used to obtain a specific column from among those on which an event is defined.

Signature.

```
const Column* getEventColumn
    (
      unsigned no
    ) const
```

Parameters. The number (*no*) of the column, as obtained using `getNoOfColumns()` (see Section 2.3.4.2.8, "`Event::getNoOfEventColumns()`").

Return Value. A pointer to the `Column` corresponding to *no*.

2.3.4.2.10. `Event::getObjectStatus()`

Description. This method gets the object status of the event.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object status of the event — for possible values, see Section 2.3.20.1.3, "The `Object::Status` Type".

2.3.4.2.11. `Event::getObjectVersion()`

Description. This method gets the event's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object version of the event, as an integer.

2.3.4.2.12. `Event::getObjectId()`

Description. This method retrieves an event's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID of the event, as an integer.

2.3.4.2.13. `Event::setName()`

Description. This method is used to set the name of an event. The name must be unique among all events visible from the current application (see Section 2.3.4.2.6, "`Event::getDurability()`").

> **Note**
>
> You can also set the event's name when first creating it. See Section 2.3.4.2.1, "`Event` Constructor".

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The *name* to be given to the event (as a constant character pointer).

Return Value. *None*.

2.3.4.2.14. `Event::setTable()`

Description. This method defines a table on which events are to be detected.

> **Note**
>
> By default, event detection takes place on all columns in the table. Use `addEventColumn()` to override this behaviour. For details, see Section 2.3.4.2.18, "`Event::addEventColumn()`".

Signature.

```
void setTable
    (
      const NdbDictionary::Table& table
    )
```

Parameters. This method requires a single parameter, a reference to the table (see Section 2.3.21, "The `Table` Class") on which events are to be detected.

Return Value. *None*.

2.3.4.2.15. `Event::addTableEvent()`

Description. This method is used to add types of events that should be detected.

Signature.

```
void addTableEvent
    (
      const TableEvent te
    )
```

Parameters. This method requires a `TableEvent` value. See Section 2.3.4.1.1, "The `Event::TableEvent` Type" for possible values.

Return Value. *None*.

2.3.4.2.16. `Event::setDurability()`

Description. This method sets an event's durability — that is, its lifetime and scope.

Signature.

```
void setDurability(EventDurability ed)
```

Parameters. This method requires a single `EventDurability` value as a parameter. See Section 2.3.4.1.2, "The `Event::EventDurability` Type", for possible values.

Return Value. *None*.

2.3.4.2.17. `Event::setReport()`

Description. This method is used to set a reporting option for an event. Possible option values may be found in Section 2.3.4.1.3, "The `Event::EventReport` Type".

Signature.

```
void setReport
    (
      EventReport er
    )
```

Parameters. An `EventReport` option value.

Return Value. *None*.

2.3.4.2.18. `Event::addEventColumn()`

Description. This method is used to add a column on which events should be detected. The column may be indicated either by its

ID or its name.

> **Important**
>
> You must invoke `Dictionary::createEvent()` before any errors will be detected. See Section 2.3.3.1.12, "`Dictionary::createEvent()`".

> **Note**
>
> If you know several columns by name, you can enable event detection on all of them at one time by using `addEventColumns()`. See Section 2.3.4.2.19, "`Event::addEventColumns()`".

Signature. Identifying the event using its column ID:

```
void addEventColumn
    (
      unsigned attrId
    )
```

Identifying the column by name:

```
void addEventColumn
    (
      const char* columnName
    )
```

Parameters. This method takes a single argument, which may be either one of:

- The column ID (`attrId`), which should be an integer greater than or equal to `0`, and less than the value returned by `getNoOfEventColumns()`.
- The column's *`name`* (as a constant character pointer).

Return Value. *None*.

2.3.4.2.19. `Event::addEventColumns()`

Description. This method is used to enable event detection on several columns at the same time. You must use the names of the columns.

> **Important**
>
> As with `addEventColumn()`, you must invoke `Dictionary::createEvent()` before any errors will be detected. See Section 2.3.3.1.12, "`Dictionary::createEvent()`".

Signature.

```
void addEventColumns
    (
      int          n,
      const char** columnNames
    )
```

Parameters. This method requires two arguments:

- The number of columns *`n`* (an integer).
- The names of the columns *`columnNames`* — this must be passed as a pointer to a character pointer.

Return Value. *None*.

2.3.4.2.20. `Event::mergeEvents()`

Description. This method is used to set the *merge events flag*, which is `false` by default. Setting it to `true` implies that events are merged as follows:

- For a given `NdbEventOperation` associated with this event, events on the same primary key within the same global checkpoint index (GCI) are merged into a single event.

- A blob table event is created for each blob attribute, and blob events are handled as part of main table events.
- Blob post/pre data from blob part events can be read via `NdbBlob` methods as a single value.

> **Note**
>
> Currently this flag is not inherited by `NdbEventOperation`, and must be set on `NdbEventOperation` explicitly. See Section 2.3.11, "The `NdbEventOperation` Class".

Signature.

```
void mergeEvents
    (
      bool flag
    )
```

Parameters. A Boolean *`flag`* value.

Return Value. *None*.

2.3.5. The `Index` Class

This section provides a reference to the `Index` class and its public members.

Parent class. NdbDictionary

Child classes. *None*

Description. This class represents an index on an `NDB Cluster` table column. It is a descendant of the `NdbDictionary` class, via the `Object` class. For information on these, see Section 2.3.10, "The `NdbDictionary` Class", and Section 2.3.20, "The `Object` Class".

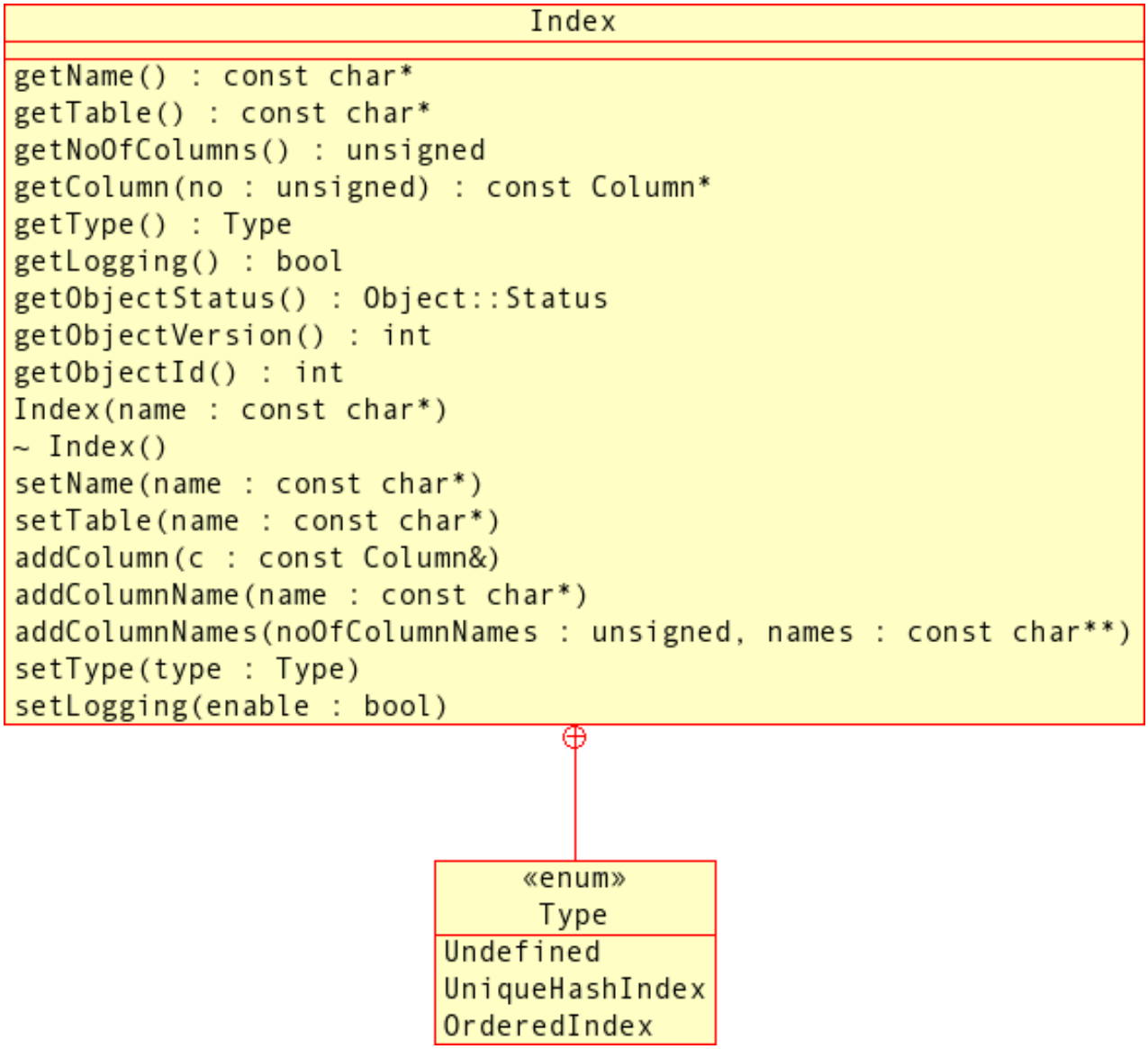
Methods. The following table lists the public methods of `Index` and the purpose or use of each method:

Method	Purpose / Use
`Index()`	Class constructor
`~Index()`	Destructor
`getName()`	Gets the name of the index
`getTable()`	Gets the name of the table being indexed
`getNoOfColumns()`	Gets the number of columns belonging to the index
`getColumn()`	Gets a column making up (part of) the index
`getType()`	Gets the index type
`getLogging()`	Checks whether the index is logged to disk
`getObjectStatus()`	Gets the index object status
`getObjectVersion()`	Gets the index object status
`getObjectId()`	Gets the index object ID
`setName()`	Sets the name of the index
`setTable()`	Sets the name of the table to be indexed
`addColumn()`	Adds a `Column` object to the index
`addColumnName()`	Adds a column by name to the index
`addColumnNames()`	Adds multiple columns by name to the index
`setType()`	Set the index type
`setLogging()`	Enable/disable logging of the index to disk

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.5.2, "`Index` Methods".

Types. Index has one public type, the `Type` type. For a discussion of `Type`, see Section 2.3.5.1, "The `Index::Type` Type".

Class diagram. This diagram shows all the available methods and enumerated types of the `Index` class:



2.3.5.1. The `Index::Type` Type

Description. This is an enumerated type which describes the sort of column index represented by a given instance of `Index`.

> **Caution**
>
> Do not confuse this enumerated type with `Object::Type`, which is discussed in Section 2.3.20.1.5, "The `Object::Type` Type", or with `Table::Type` or `Column::Type`.

Enumeration values.

Value	Description
`Undefined`	Undefined object type (initial/default value)
`UniqueHashIndex`	Unique unordered hash index (only index type currently supported)
`OrderedIndex`	Non-unique, ordered index

2.3.5.2. `Index` Methods

This section contain descriptions of all public methods of the `Index` class. This class has relatively few methods (compared to, say, `Table`), which are fairly straightforward to use.

> **Important**

If you create or change indexes using the NDB API, these modifications cannot be seen by MySQL. The only exception to this is renaming the index using `Index::setName()`.

2.3.5.2.1. `Index` Class Constructor

Description. This is used to create an new instance of `Index`.

Important

Indexes created using the NDB API cannot be seen by the MySQL Server.

Signature.

```
Index
    (
      const char* name = ""
    )
```

Parameters. The name of the new index. It is possible to create an index without a name, and then assign a name to it later using `setName()`. See Section 2.3.5.2.11, "`Index::setName()`".

Return Value. A new instance of `Index`.

Destructor. The destructor (`~Index()`) is supplied as a virtual method.

2.3.5.2.2. `Index::getName()`

Description. This method is used to obtain the name of an index.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the index, as a constant character pointer.

2.3.5.2.3. `Index::getTable()`

Description. This method can be used to obtain the name of the table to which the index belongs.

Signature.

```
const char* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table, as a constant character pointer.

2.3.5.2.4. `Index::getNoOfColumns()`

Description. This method is used to obtain the number of columns making up the index.

Signature.

```
unsigned getNoOfColumns
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An unsigned integer representing the number of columns in the index.

2.3.5.2.5. `Index::getColumn()`

Description. This method retrieves the column at the specified position within the index.

Signature.

```
const Column* getColumn
    (
      unsigned no
    ) const
```

Parameters. The ordinal position number *no* of the column, as an unsigned integer. Use the `getNoOfColumns()` method to determine how many columns make up the index — see Section 2.3.5.2.4, "`Index::getNoOfColumns()`", for details.

Return Value. The column having position *no* in the index, as a pointer to an instance of `Column`. See Section 2.3.1, "The `Column` Class".

2.3.5.2.6. `Index::getType()`

Description. This method can be used to find the type of index.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An index type. See Section 2.3.5.1, "The `Index::Type` Type", for possible values.

2.3.5.2.7. `Index::getLogging()`

Description. Use this method to determine whether logging to disk has been enabled for the index.

> **Note**
>
> Indexes which are not logged are rebuilt when the cluster is started or restarted.
>
> Ordered indexes currently do not support logging to disk; they are rebuilt each time the cluster is started. (This includes restarts.)

Signature.

```
bool getLogging
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A Boolean value:

- `true`: The index is being logged to disk.
- `false`: The index is not being logged.

2.3.5.2.8. `Index::getObjectStatus()`

Description. This method gets the object status of the index.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `Status` value — see Section 2.3.20.1.3, "The `Object::Status` Type", for more information.

2.3.5.2.9. `Index::getObjectVersion()`

Description. This method gets the object version of the index.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object version for the index, as an integer.

2.3.5.2.10. `Index::getObjectId()`

Description. This method is used to obtain the object ID of the index.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, as an integer.

2.3.5.2.11. `Index::setName()`

Description. This method sets the name of the index.

> **Note**
>
> This is the only `Index::set*()` method whose result is visible to a MySQL Server.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The desired *`name`* for the index, as a constant character pointer.

Return Value. *None*.

2.3.5.2.12. `Index::setTable()`

Description. This method sets the table that is to be indexed. The table is referenced by name.

Signature.

```
void setTable
    (
      const char* name
    )
```

Parameters. The *`name`* of the table to be indexed, as a constant character pointer.

Return Value. *None*.

2.3.5.2.13. `Index::addColumn()`

Description. This method may be used to add a column to an index.

> **Note**
>
> The order of the columns matches the order in which they are added to the index. However, this matters only with ordered indexes.

Signature.

```
void addColumn
    (
      const Column& c
    )
```

Parameters. A reference *c* to the column which is to be added to the index.

Return Value. *None*.

2.3.5.2.14. Index::addColumnName()

Description. This method works in the same way as addColumn(), except that it takes the name of the column as a parameter. See Section 2.3.5.2.5, "Index::getColumn()".

Signature.

```
void addColumnName
    (
      const char* name
    )
```

Parameters. The *name* of the column to be added to the index, as a constant character pointer.

Return Value. *None*.

2.3.5.2.15. Index::addColumnNames()

Description. This method is used to add several column names to an index definition at one time.

> **Note**
>
> As with the addColumn() and addColumnName() methods, the indexes are numbered in the order in which they were added. (However, this matters only for ordered indexes.)

Signature.

```
void addColumnNames
    (
      unsigned     noOfNames,
      const char** names
    )
```

Parameters. This method takes two parameters:

- The number of columns/names *noOfNames* to be added to the index.
- The *names* to be added (as a pointer to a pointer).

Return Value. *None*.

2.3.5.2.16. Index::setType()

Description. This method is used to set the index type.

Signature.

```
void setType
    (
      Type type
    )
```

Parameters. The *type* of index. For possible values, see Section 2.3.5.1, "The Index::Type Type".

Return Value. *None*.

2.3.5.2.17. Index::setLogging

Description. This method is used to enable or disable logging of the index to disk.

Signature.

```
void setLogging
    (
      bool enable
    )
```

Parameters. `setLogging()` takes a single Boolean parameter *`enable`*. If *`enable`* is `true`, then logging is enabled for the index; if false, then logging of this index is disabled.

Return Value. *None*.

2.3.6. The `LogfileGroup` Class

This section discusses the `LogfileGroup` class, which represents a MySQL Cluster Disk Data logfile group.

Parent class. NdbDictionary

Child classes. *None*

Description. This class represents a MySQL Cluster Disk Data logfile group, which is used for storing Disk Data undofiles. For general information about logfile groups and undofiles, see the MySQL Cluster Disk Data Tables, in the MySQL Manual.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support logfile groups; thus the `LogfileGroup` class is unavailable for NDB API applications written against these MySQL versions.

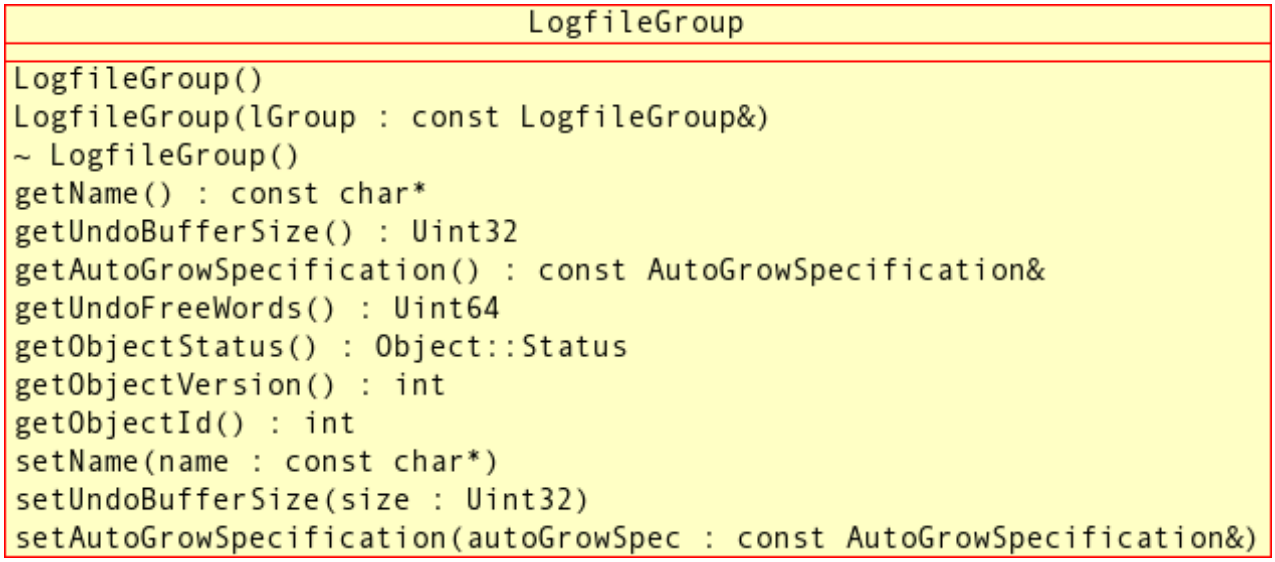
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`LogfileGroup()`	Class constructor
`~LogfileGroup()`	Virtual destructor
`getName()`	Retrieves the logfile group's name
`getUndoBufferSize()`	Gets the size of the logfile group's `UNDO` buffer
`getAutoGrowSpecification()`	Gets the logfile group's `AutoGrowSpecification` values
`getUndoFreeWords()`	Retrieves the amount of free space in the `UNDO` buffer
`getObjectStatus()`	Gets the logfile group's object status value
`getObjectVersion()`	Retrieves the logfile group's object version
`getObjectId()`	Get the object ID of the logfile group
`setName()`	Sets the name of the logfile group
`setUndoBufferSize()`	Sets the size of the logfile group's `UNDO` buffer.
`setAutoGrowSpecification()`	Sets `AutoGrowSpecification` values for the logfile group

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.6.1, “`LogfileGroup` Methods”.

Types. The `LogfileGroup` class does not itself define any public types. However, two of its methods make use of the `AutoGrowSpecification` data structure as a parameter or return value. For more information, see Section 2.3.26, “The `AutoGrowSpecification` Structure”.

Class diagram. This diagram shows all the available public methods of the `LogfileGroup` class:



2.3.6.1. `LogfileGroup` Methods

This section provides descriptions for the public methods of the `LogfileGroup` class.

2.3.6.1.1. `LogfileGroup` Constructor

Description. The LogfileGroup class has two public constructors, one of which takes no arguments and creates a completely new instance. The other is a copy constructor.

> **Note**
>
> The `Dictionary` class also supplies methods for creating and destroying `LogfileGroup` objects. See Section 2.3.3, "The `Dictionary` Class".

Signatures. New instance:

```
LogfileGroup
    (
      void
    )
```

Copy constructor:

```
LogfileGroup
    (
      const LogfileGroup& logfileGroup
    )
```

Parameters. When creating a new instance, the constructor takes no parameters. When copying an existing instance, the constructor is passed a reference to the `LogfileGroup` instance to be copied.

Return Value. A `LogfileGroup` object.

Destructor.

```
virtual ~LogfileGroup
    (
      void
    )
```

Examples.

```
[To be supplied...]
```

2.3.6.1.2. `LogfileGroup::getName()`

Description. This method gets the name of the logfile group.

Signature.

```
const char* getName
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The logfile group's name, a string (as a character pointer).

2.3.6.1.3. `LogfileGroup::getUndoBufferSize()`

Description. This method retrieves the size of the logfile group's `UNDO` buffer.

Signature.

```
Uint32 getUndoBufferSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The size of the `UNDO` buffer, in bytes.

2.3.6.1.4. `LogfileGroup::getAutoGrowSpecification()`

Description. This method retrieves the `AutoGrowSpecification` associated with the logfile group.

Signature.

```
const AutoGrowSpecification& getAutoGrowSpecification
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `AutoGrowSpecification` data structure. See Section 2.3.26, “The `AutoGrowSpecification` Structure”, for details.

2.3.6.1.5. `LogfileGroup::getUndoFreeWords()`

Description. This method retrieves the number of bytes unused in the logfile group's `UNDO` buffer.

Signature.

```
Uint64 getUndoFreeWords
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of bytes free, as a 64-bit integer.

2.3.6.1.6. `LogfileGroup::getObjectStatus()`

Description. This method is used to obtain the object status of the `LogfileGroup`.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The logfile group's `Status` — see Section 2.3.20.1.3, “The `Object::Status` Type” for possible values.

2.3.6.1.7. `LogfileGroup::getObjectVersion()`

Description. This method gets the logfile group's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object version of the logfile group, as an integer.

2.3.6.1.8. `LogfileGroup::getObjectId()`

Description. This method is used to retrieve the object ID of the logfile group.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The logfile group's object ID (an integer value).

2.3.6.1.9. `LogfileGroup::setName()`

Description. This method is used to set a name for the logfile group.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The *`name`* to be given to the logfile group (character pointer).

Return Value. *None*.

2.3.6.1.10. `LogfileGroup::setUndoBufferSize()`

Description. This method can be used to set the size of the logfile group's `UNDO` buffer.

Signature.

```
void setUndoBufferSize
    (
      Uint32 size
    )
```

Parameters. The *`size`* in bytes for the `UNDO` buffer (using a 32-bit unsigned integer value).

Return Value. *None*.

2.3.6.1.11. `LogfileGroup::setAutoGrowSpecification()`

Description. This method sets to the `AutoGrowSpecification` data for the logfile group.

Signature.

```
void setAutoGrowSpecification
    (
      const AutoGrowSpecification& autoGrowSpec
    )
```

Parameters. The data is passed as a single parameter, an `AutoGrowSpecification` data structure — see Section 2.3.26, “The `AutoGrowSpecification` Structure”.

Return Value. *None*.

2.3.7. The `List` Class

This section covers the `List` class.

Parent class. Dictionary

Child classes. *None*

Description. The `List` class is a `Dictionary` subclass that is used for representing lists populated by the methods `Dictionary::listObjects()`, `Dictionary::listIndexes()`, and `Dictionary::listEvents()`. (See Section 2.3.3.1.26, "`Dictionary::listObjects()`", Section 2.3.3.1.27, "`Dictionary::listIndexes()`", and Section 2.3.3.1.28, "`Dictionary::listEvents()`", for more information about these methods.)

Class Methods. This class has only two methods, a constructor and a destructor. Neither method takes any arguments.

Constructor. Calling the `List` constructor creates a new `List` whose `count` and `elements` attributes are both set equal to `0`.

Destructor. The destructor `~List()` is simply defined in such a way as to remove all elements and their properties. You can find its definition in the file `/storage/ndb/include/ndbapi/NdbDictionary.hpp`.

Attributes. A `List` has two attributes:

- `count`, an unsigned integer, which stores the number of elements in the list.
- `elements`, a pointer to an array of `Element` data structures contained in the list. See Section 2.3.27, "The `Element` Structure".

Types. The `List` class defines an `Element` structure, which is described in the following section.

> **Note**
>
> For a graphical representation of this class and its parent-child relationships, see Section 2.3.3, "The `Dictionary` Class".

2.3.8. The `Ndb` Class

This class represents the `NDB` kernel; it is the primary class of the NDB API.

Parent class. *None*

Child classes. *None*

Description. Any non-trivial NDB API program makes use of at least one instance of `Ndb`. By using several `Ndb` objects, it is possible to implement a multi-threaded application. You should remember that one `Ndb` object cannot be shared between threads; however, it is possible for a single thread to use multiple `Ndb` objects. A single application process can support a maximum of 128 `Ndb` objects.

> **Note**
>
> The Ndb object is multi-thread safe in that each `Ndb` object can be handled by one thread at a time. If an `Ndb` object is handed over to another thread, then the application must ensure that a memory barrier is used to ensure that the new thread sees all updates performed by the previous thread.
>
> Semaphores and mutexes are examples of easy ways to provide memory barriers without having to bother about the memory barrier concept.

It is also possible to use multiple `Ndb` objects to perform operations on different clusters in a single application. See the Note on Application-Level Partitioning for conditions and restrictions applying to such usage.

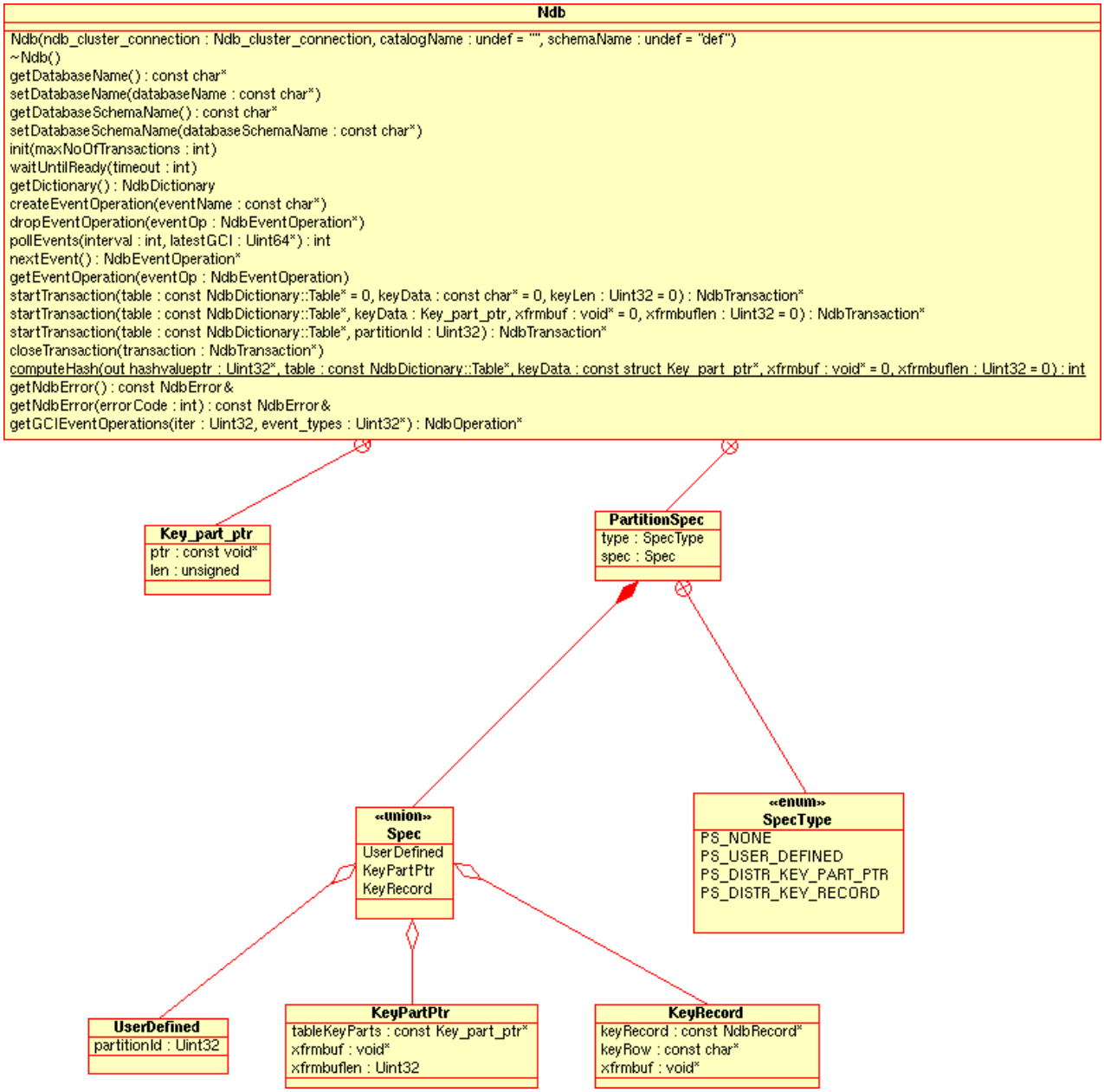
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Ndb()`	Class constructor; represents a connection to a MySQL Cluster.
`~Ndb()`	Class destructor; terminates a Cluster connection when it is no longer to be used
`init()`	Initialises an `Ndb` object and makes it ready for use.
`getDictionary()`	Gets a dictionary, which is used for working with database schema information.
`getDatabaseName()`	Gets the name of the current database.

Method	Purpose / Use
`setDatabaseName()`	Sets the name of the current database.
`getDatabaseSchemaName()`	Gets the name of the current database schema.
`setDatabaseSchemaName()`	Sets the name of the current database schema.
`startTransaction()`	Begins a transaction. (See Section 2.3.19, "The `NdbTransaction` Class".)
`closeTransaction()`	Closes a transaction.
`computeHash()`	Computes a distribution hash value.
`createEventOperation()`	Creates a subscription to a database event. (See Section 2.3.11, "The `NdbEvent-Operation` Class".)
`dropEventOperation()`	Drops a subscription to a database event.
`pollEvents()`	Waits for an event to occur.
`getNdbError()`	Retrieves an error. (See Section 2.3.30, "The `NdbError` Structure".)
`getReference()`	Retrieves a reference or identifier for the `Ndb` object instance.

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.8.1, "`Ndb` Methods".

Class diagram. This diagram shows all the available members of the `Ndb` class:



2.3.8.1. `Ndb` Methods

The sections that follow discuss the public methods of the `Ndb` class.

2.3.8.1.1. `Ndb` Class Constructor

Description. This creates an instance of `Ndb`, which represents a connection to the MySQL Cluster. All NDB API applications should begin with the creation of at least one `Ndb` object. This requires the creation of at least one instance of `Ndb_cluster_connection`, which serves as a container for a cluster connectstring.

Signature.

```
Ndb
    (
      Ndb_cluster_connection* ndb_cluster_connection,
      const char*                    catalogName = "",
      const char*                    schemaName = "def"
    )
```

Parameters. The `Ndb` class constructor can take up to 3 parameters, of which only the first is required:

- *ndb_cluster_connection* is an instance of Ndb_cluster_connection, which represents a cluster connectstring. (See Section 2.3.24, "The Ndb_cluster_connection Class".)
- *catalogName* is an optional parameter providing a namespace for the tables and indexes created in any connection from the Ndb object.

  This is equivalent to what mysqld considers "the database".

  The default value for this parameter is an empty string.
- The optional *schemaName* provides an additional namespace for the tables and indexes created in a given catalog.

  The default value for this parameter is the string "def".

Return Value. An Ndb object.

~Ndb() (Class Destructor). The destructor for the Ndb class should be called in order to terminate an instance of Ndb. It requires no arguments, nor any special handling.

2.3.8.1.2. Ndb::init()

Description. This method is used to initialise an Ndb object.

Signature.

```
int init
    (
      int maxNoOfTransactions = 4
    )
```

Parameters. The init() method takes a single parameter *maxNoOfTransactions* of type integer. This parameter specifies the maximum number of parallel NdbTransaction objects that can be handled by this instance of Ndb. The maximum permitted value for *maxNoOfTransactions* is 1024; if not specified, it defaults to 4.

> **Note**
>
> Each scan or index operation uses an extra NdbTransaction object. See Section 2.3.19, "The NdbTransaction Class".

Return Value. This method returns an int, which can be either of two values:

- **0**: indicates that the Ndb object was initialised successfully.
- **-1**: indicates failure.

2.3.8.1.3. Ndb::getDictionary()

Description. This method is used to obtain an object for retrieving or manipulating database schema information. This Dictionary object contains meta-information about all tables in the cluster.

> **Note**
>
> The dictionary returned by this method operates independently of any transaction. See Section 2.3.3, "The Dictionary Class", for more information.

Signature.

```
NdbDictionary::Dictionary* getDictionary
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An instance of the Dictionary class.

2.3.8.1.4. Ndb::getDatabaseName()

Description. This method can be used to obtain the name of the current database.

Signature.

```
const char* getDatabaseName
    (
      void
    )
```

Parameters. None.

Return Value. The name of the current database.

2.3.8.1.5. Ndb::setDatabaseName()

Description. This method is used to set the name of the current database.

Signature.

```
void setDatabaseName
    (
      const char *databaseName
    )
```

Parameters. `setDatabaseName()` takes a single, required parameter, the name of the new database to be set as the current database.

Return Value. N/A.

2.3.8.1.6. Ndb::getDatabaseSchemaName()

Description. This method can be used to obtain the current database schema name.

Signature.

```
const char* getDatabaseSchemaName
    (
      void
    )
```

Parameters. None.

Return Value. The name of the current database schema.

2.3.8.1.7. Ndb::setDatabaseSchemaName()

Description. This method allows you to set the name of the current database schema.

Signature.

```
void setDatabaseSchemaName
    (
      const char *databaseSchemaName
    )
```

Parameters. The name of the database schema.

Return Value. N/A.

2.3.8.1.8. Ndb::startTransaction()

Description. This method is used to begin a new transaction. There are three variants, the simplest of these using a table and a partition key or partition ID to specify the transaction coordinator (TC). The third variant allows you to specify the TC by means of a pointer to the data of the key.

> **Important**
>
> When the transaction is completed it must be closed using `NdbTransaction::close()` or `Ndb::closeTransaction()`. Failure to do so aborts the transaction. This must be done regardless of the transaction's final outcome, even if it fails due to an error.
>
> See Section 2.3.8.1.9, “`Ndb::closeTransaction()`”, and Section 2.3.19.2.7, “`NdbTransac-`

tion::close()”.

Signature.

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table* table = 0,
      const char* keyData = 0,
      Uint32* keyLen = 0
    )
```

Parameters. This method takes three parameters, as follows:

- *table*: A pointer to a Table object. (See Section 2.3.21, “The Table Class”.) This is used to determine on which node the transaction coordinator should run.
- *keyData*: A pointer to a partition key corresponding to *table*.
- *keyLen*: The length of the partition key, expressed in bytes.

Distribution-aware forms of startTransaction(). Beginning with MySQL Cluster NDB 6.1.4, it is possible to employ *distribution awareness* with this method — that is, to suggest which node should act as the transaction coordinator — .

Signature.

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table* table,
      const struct Key_part_ptr*  keyData,
      void*                       xfrmbuf = 0,
      Uint32                      xfrmbuflen = 0
    )
```

Parameters. When specifying the transaction coordinator, this method takes four parameters:

- A pointer to a *table* (NdbDictionary::Table object) used for deciding which node should act as the transaction coordinator.
- A null-terminated array of pointers to the values of the distribution key columns. The length of the key part is read from metadata and checked against the passed value.

  A Key_part_ptr is defined as shown in Section 2.3.29, “The Key_part_ptr Structure”.
- A pointer to a temporary buffer, used to calculate the hash value.
- The length of the buffer.

If *xfrmbuf* is NULL (the default), then a call to malloc() or free() is made automatically, as appropriate. startTransaction() fails if xfrmbuf is not NULL and *xfrmbuflen* is too small.

Example. Suppose that the table's partition key is a single BIGINT column. Then you would declare the distribution key array as shown here:

```
Key_part_ptr distkey[2];
```

The value of the distribution key would be defined as shown here:

```
unsigned long long distkeyValue= 23;
```

The pointer to the distribution key array would be set as follows:

```
distkey[0].ptr= (const void*) &distkeyValue;
```

The length of this pointer would be set accordingly:

```
distkey[0].len= sizeof(distkeyValue);
```

The distribution key array must terminate with a NULL element. This is necessary to avoid to having an additional parameter providing the number of columns in the distribution key:

```
distkey[1].ptr= NULL;
distkey[1].len= NULL;
```

Setting the buffer to NULL allows startTransaction() to allocate and free memory automatically:

```
xfrmbuf= NULL;
xfrmbuflen= 0;
```

> **Note**
>
> You can also specify a buffer to save having to make explicit `malloc()` and `free()` calls, but calculating an appropriate size for this buffer is not a simple matter; if the buffer is not `NULL` but its length is too short, then the startTransaction() call fails. However, if you choose to specify the buffer, 1 MB is usually a sufficient size.

Now, when you start the transaction, you can access the node that contains the desired information directly.

In MySQL Cluster NDB 6.2 and later, another distribution-aware version of this method is available. This variant allows you to specify a table and partition (using the partition ID) as a hint for selecting the transaction coordinator, and is defined as shown here:

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table* table,
      Uint32 partitionId
    )
```

In the event that the cluster has the same number of data nodes as it has replicas, specifying the transaction coordinator gains no improvement in performance, since each data node contains the entire database. However, where the number of data nodes is greater than the number of replicas (for example, where `NoOfReplicas` is set equal to 2 in a cluster with 4 data nodes), you should see a marked improvement in performance by using the distribution-aware version of this method.

It is still possible to use this method as before, without specifying the transaction coordinator. In either case, you must still explicitly close the transaction, whether or not the call to `startTransaction()` was successful.

Return Value. On success, an `NdbTransaction` object (see Section 2.3.19, “The `NdbTransaction` Class”). In the event of failure, `NULL` is returned.

2.3.8.1.9. `Ndb::closeTransaction()`

Description. This is one of two NDB API methods provided for closing a transaction (the other being `NdbTransaction::close()` — see Section 2.3.19.2.7, “`NdbTransaction::close()`”). You must call one of these two methods to close the transaction once it has been completed, whether or not the transaction succeeded.

> **Important**
>
> If the transaction has not yet been committed, it is aborted when this method is called. See Section 2.3.8.1.8, “`Ndb::startTransaction()`”.

Signature.

```
void closeTransaction
    (
      NdbTransaction *transaction
    )
```

Parameters. This method takes a single argument, a pointer to the `NdbTransaction` to be closed.

Return Value. N/A.

2.3.8.1.10. `Ndb::computeHash()`

Description. This method can be used to compute a distribution hash value, given a table and its keys.

> **Important**
>
> `computeHash()` can be used onlyt for tables that use native NDB partitioning.

Signature.

```
static int computeHash
    (
      Uint32*                    hashvalueptr,
      const NdbDictionary::Table* table,
      const struct Key_part_ptr*  keyData,
      void*                      xfrmbuf = 0,
      Uint32                     xfrmbuflen = 0
    )
```

Parameters. This method takes the following parameters:

- If the method call id successful, *`hashvalueptr`* is set to the computed hash value.
- A pointer to a *`table`* (see Section 2.3.21, “The `Table` Class”).
- *`keyData`* is a null-terminated array of pointers to the key parts that are part of the table's distribution key. The length of each key part is read from metadata and checked against the passed value (see Section 2.3.29, “The `Key_part_ptr` Structure”).
- *`xfrmbuf`* is a pointer to temporary buffer used to calculate the hash value.
- *`xfrmbuflen`* is the length of this buffer.

> **Note**
>
> If *`xfrmbuf`* is `NULL` (the default), then a call to `malloc()` or `free()` is made automatically, as appropriate. `computeHash()` fails if `xfrmbuf` is not `NULL` and *`xfrmbuflen`* is too small.

Return Value. 0 on success, an error code on failure. (If the method call succeeds, the computed hash value is made available via *`hashvalueptr`*.)

2.3.8.1.11. `Ndb::createEventOperation`

Description. This method creates a subscription to a database event.

Signature.

```
NdbEventOperation* createEventOperation
    (
      const char *eventName
    )
```

Parameters. This method takes a single argument, the unique *`eventName`* identifying the event to which you wish to subscribe.

Return Value. A pointer to an `NdbEventOperation` object (or `NULL`, in the event of failure). See Section 2.3.11, “The `NdbEventOperation` Class”.

2.3.8.1.12. `Ndb::dropEventOperation()`

Description. This method drops a subscription to a database event represented by an `NdbEventOperation` object.

Signature.

```
int dropEventOperation
    (
      NdbEventOperation *eventOp
    )
```

Parameters. This method requires a single input parameter, a pointer to an instance of `NdbEventOperation`.

Return Value. **0** on success; any other result indicates failure.

2.3.8.1.13. `Ndb::pollEvents()`

Description. This method waits for a GCP to complete. It is used to determine whether any events are available in the subscription queue.

Beginning with MySQL Cluster NDB 6.2.5, this method waits for the next *epoch*, rather than the next GCP. See Section 2.3.11, “The `NdbEventOperation` Class”, for more information about the differences in event handling for this and later MySQL Cluster NDB 6.2.*x* releases.

Signature.

```
int pollEvents
    (
      int     maxTimeToWait,
      Uint64* latestGCI = 0
    )
```

Parameters. This method takes two parameters, as shown here:

- The maximum time to wait, in milliseconds, before “giving up” and reporting that no events were available (that is, before the

method automatically returns **0**).

- The index of the most recent global checkpoint. Normally, this may safely be permitted to assume its default value, which is `0`.

Return Value. `pollEvents()` returns a value of type `int`, which may be interpreted as follows:

- **> 0**: There are events available in the queue.
- **0**: There are no events available.
- **< 0**: Indicates failure (possible error).

2.3.8.1.14. `Ndb::nextEvent()`

Description. Returns the next event operation having data from a subscription queue.

Signature.

```
NdbEventOperation* nextEvent
    (
      void
    )
```

Parameters. None.

Return Value. This method returns an `NdbEventOperation` object representing the next event in a subscription queue, if there is such an event. If there is no event in the queue, it returns `NULL` instead. (See Section 2.3.11, "The `NdbEventOperation` Class".)

2.3.8.1.15. `Ndb::getNdbError()`

Description. This method provides you with two different ways to obtain an `NdbError` object representing an error condition. For more detailed information about error handling in the NDB API, see Chapter 4, *MySQL Cluster API Errors*.

Signature. The `getNdbError()` method actually has two variants. The first of these simply gets the most recent error to have occurred:

```
const NdbError& getNdbError
    (
      void
    )
```

The second variant returns the error corresponding to a given error code:

```
const NdbError& getNdbError
    (
      int errorCode
    )
```

Regardless of which version of the method is used, the `NdbError` object returned persists until the next NDB API method is invoked.

Parameters. To obtain the most recent error, simply call `getNdbError()` without any parameters. To obtain the error matching a specific *`errorCode`*, invoke the method passing the code (an `int`) to it as a parameter. For a listing of NDB API error codes and corresponding error messages, see Section 4.2, "NDB API Errors and Error Handling".

Return Value. An `NdbError` object containing information about the error, including its type and, where applicable, contextual information as to how the error arose. See Section 2.3.30, "The `NdbError` Structure", for details.

2.3.8.1.16. `Ndb::getReference()`

Description. This method can be used to obtain a reference to a given `Ndb` object. This is the same value that is returned for a given operation corresponding to this object in the output of `DUMP 2350`. (See Section 5.2.3.9, "`DUMP 2350`", for an example.)

Signature.

```
Uint32 getReference
    (
      void
    )
```

Parameters. *None*.

2.3.9. The NdbBlob Class

This class represents a handle to a BLOB column and provides read and write access to BLOB column values. This object has a number of different states and provides several modes of access to BLOB data; these are also described in this section.

Parent class. *None*

Child classes. *None*

Description. An instance of NdbBlob is created using the NdbOperation::getBlobHandle() method during the operation preparation phase. (See Section 2.3.15, "The NdbOperation Class".) This object acts as a handle on a BLOB column.

BLOB Data Storage. BLOB data is stored in 2 locations:

- The header and inline bytes are stored in the blob attribute.
- The blob's data segments are stored in a separate table named NDB$BLOB_*tid*_cid, where *tid* is the table ID, and *cid* is the blob column ID.

The inline and data segment sizes can be set using the appropriate NdbDictionary::Column() methods when the table is created. See Section 2.3.1, "The Column Class", for more information about these methods.

Data Access Types. NdbBlob supports 3 types of data access:

- In the preparation phase, the NdbBlob methods getValue() and setValue() are used to prepare a read or write of a BLOB value of known size.
- Also in the preparation phase, setActiveHook() is used to define a routine which is invoked as soon as the handle becomes active.
- In the active phase, readData() and writeData() are used to read and write BLOB values having arbitrary sizes.

These data access types can be applied in combination, provided that they are used in the order given above.

BLOB Operations. BLOB operations take effect when the next transaction is executed. In some cases, NdbBlob is forced to perform implicit execution. To avoid this, you should always operate on complete blob data segments, and use NdbTransaction::executePendingBlobOps() to flush reads and writes. There is no penalty for doing this if nothing is pending. It is not necessary to do so following execution (obviously) or after next scan result is obtained. NdbBlob also supports reading post- or pre-blob data from events. The handle can be read after the next event on the main table has been retrieved. The data becomes available immediately. (See Section 2.3.11, "The NdbEventOperation Class".)

BLOBs and NdbOperations. NdbOperation methods acting on NdbBlob objects have the following characteristics:

- NdbOperation::insertTuple() must use NdbBlob::setValue() if the BLOB attribute is non-nullable.
- NdbOperation::readTuple() used with any lock mode can read but not write blob values.

  When the LM_CommittedRead lock mode is used with readTuple(), the lock mode is automatically upgraded to LM_Read whenever blob attributes are accessed.
- NdbOperation::updateTuple() can either overwrite an existing value using NdbBlob::setValue(), or update it during the active phase.
- NdbOperation::writeTuple() always overwrites blob values, and must use NdbBlob::setValue() if the BLOB attribute is non-nullable.
- NdbOperation::deleteTuple() creates implicit, non-accessible BLOB handles.
- A scan with any lock mode can use its blob handles to read blob values but not write them.

  A scan using the LM_Exclusive lock mode can update row and blob values using updateCurrentTuple(); the operation returned must explicitly create its own blob handle.

  A scan using the LM_Exclusive lock mode can delete row values (and therefore blob values) using deleteCurrentTuple(); this create implicit non-accessible blob handles.

- An operation which is returned by `lockCurrentTuple()` cannot update blob values.

See Section 2.3.15, "The `NdbOperation` Class".

Known Issues. The following are known issues or limitations encountered when working with `NdbBlob` objects:

- Too many pending `BLOB` operations can overflow the I/O buffers.
- The table and its `BLOB` data segment tables are not created atomically.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getState()`	Gets the state of an `NdbBlob` object
`getValue()`	Prepares to read a blob value
`setValue()`	Prepares to insert or update a blob value
`setActiveHook()`	Defines a callback for blob handle activation
`getVersion()`	Checks whether a blob is statement-based or event-based
`getNull()`	Checks whether a blob value is `NULL`
`setNull()`	Sets a blob to `NULL`
`getLength()`	Gets the length of a blob, in bytes
`truncate()`	Truncates a blob to a given length
`getPos()`	Gets the current position for reading/writing
`setPos()`	Sets the position at which to begin reading/writing
`readData()`	Reads data from a blob
`writeData()`	Writes blob data
`getColumn()`	Gets a blob column.
`getNdbError()`	Gets an error (an `NdbError` object)
`blobsFirstBlob()`	Gets the first blob in a list.
`blobsNextBlob()`	Gets the next blob in a list
`getBlobEventName()`	Gets a blob event name
`getBlobTableName()`	Gets a blob data segment's table name.
`getNdbOperation()`	Get a pointer to the operation (`NdbOperation` object) to which this `NdbBlob` object belonged when created.

> **Note**
>
> `getBlobTableName()` and `getBlobEventName()` are static methods.

> **Tip**
>
> Most `NdbBlob` methods (nearly all of those whose return type is `int`) return `0` on success and `-1` in the event of failure.

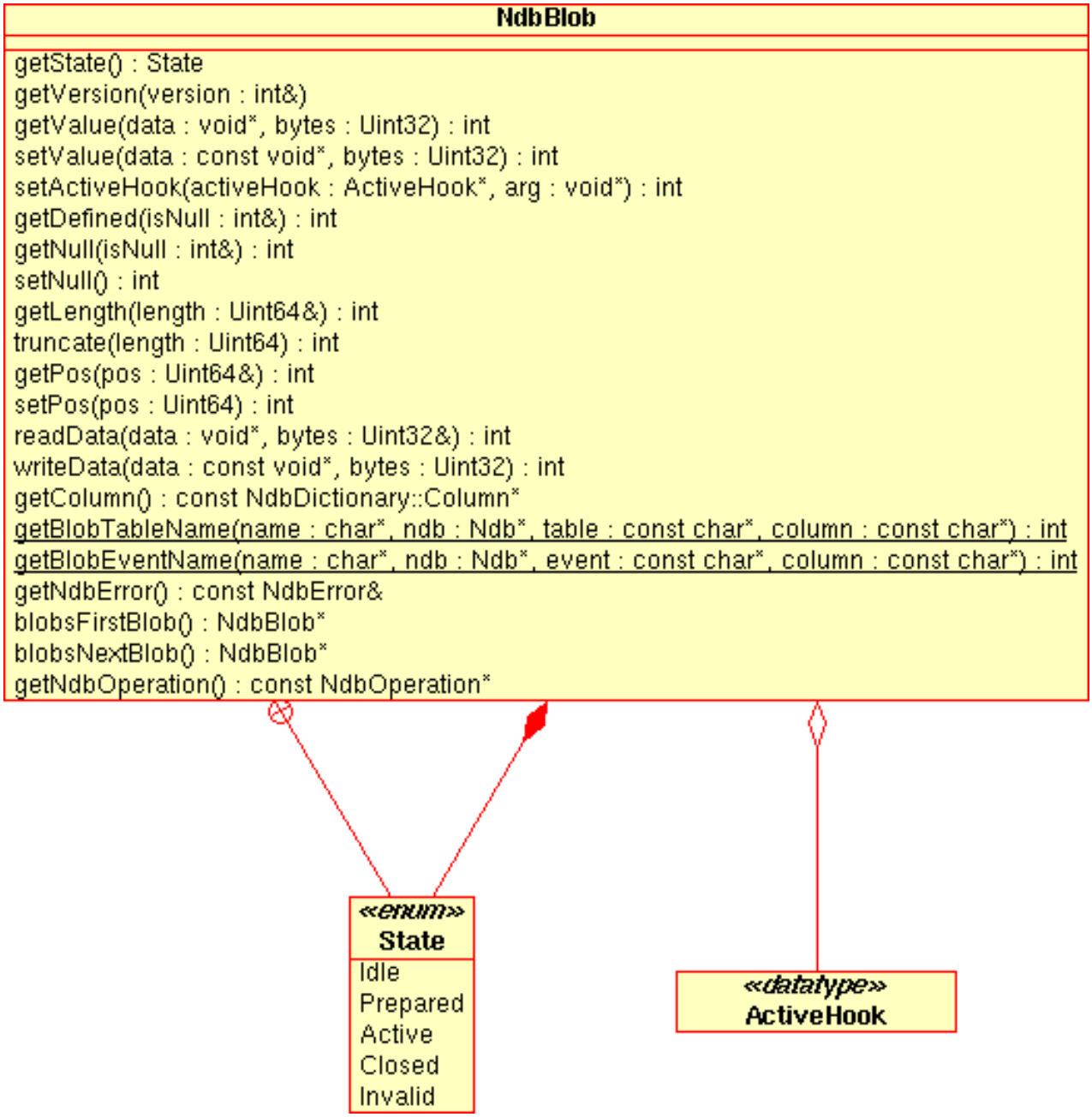
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.9.2, "`NdbBlob` Methods".

Types. The public types defined by `NdbBlob` are shown here:

Type	Purpose / Use
`ActiveHook`	Callback for `NdbBlob::setActiveHook()`
`State`	Represents the states that may be assumed by the `NdbBlob`.

For a discussion of each of these types, along with its possible values, see Section 2.3.9.1, "`NdbBlob` Types".

Class diagram. This diagram shows all the available methods and types of the `NdbBlob` class:



2.3.9.1. `NdbBlob` Types

This section details the public types belonging to the `NdbBlob` class.

2.3.9.1.1. The `NdbBlob::ActiveHook` Type

ActiveHook is a datatype defined for use as a callback for the `setActiveHook()` method. (See Section 2.3.9.2.4, "`NdbBlob::setActiveHook()`".)

Definition. `ActiveHook` is a custom datatype defined as shown here:

```
typedef int ActiveHook
    (
      NdbBlob* me,
      void*    arg
    )
```

Description. This is a callback for `NdbBlob::setActiveHook()`, and is invoked immediately once the prepared operation has been executed (but not committed). Any calls to `getValue()` or `setValue()` are performed first. The `BLOB` handle is active so `readData()` or `writeData()` can be used to manipulate the `BLOB` value. A user-defined argument is passed along with

the NdbBlob.ActiveHook() returns a nonzero value in the event of an error.

2.3.9.1.2. The NdbBlob::State Type

This is an enumerated datatype which represents the possible states of an NdbBlob instance.

Description. An NdbBlob may assume any one of these states

Enumeration values.

Value	Description
Idle	The NdbBlob has not yet been prepared for use with any operations.
Prepared	This is the state of the NdbBlob prior to operation execution.
Active	This is the BLOB handle's state following execution or the fetching of the next result, but before the transaction is committed.
Closed	This state occurs after the transaction has been committed.
Invalid	This follows a rollback or the close of a transaction.

2.3.9.2. NdbBlob Methods

This section discusses the public methods available in the NdbBlob class.

> **Important**
>
> This class has no public constructor. You can obtain a blob handle using NdbEventOperation::getBlobHandle().

2.3.9.2.1. NdbBlob::getState()

Description. This method gets the current state of the NdbBlob object for which it is invoked. Possible states are described in Section 2.3.9.1.2, "The NdbBlob::State Type".

Signature.

```
State getState
    (
      void
    )
```

Parameters. None.

Return Value. A State value. For possible values, see Section 2.3.9.1.2, "The NdbBlob::State Type".

2.3.9.2.2. NdbBlob::getValue()

Description. Use this method to prepare to read a blob value; the value is available following invocation. Use getNull() to check for a NULL value; use getLength() to get the actual length of the blob, and to check for truncation. getValue() sets the current read/write position to the point following the end of the data which was read.

Signature.

```
int getValue
    (
      void*  data,
      Uint32 bytes
    )
```

Parameters. This method takes two parameters — a pointer to the *data* to be read, and the number of *bytes* to be read.

Return Value. 0 on success, -1 on failure.

2.3.9.2.3. NdbBlob::setValue()

Description. This method is used to prepare for inserting or updating a blob value. Any existing blob data that is longer than the new data is truncated. The data buffer must remain valid until the operation has been executed. setValue() sets the current read/write position to the point following the end of the data. You can set *data* to a null pointer (0) in order to create a NULL value.

Signature.

```
int setValue
    (
      const void*  data,
      Uint32       bytes
    )
```

Parameters. This method takes two parameters:

- The *`data`* that is to be inserted or used to overwrite the blob value.
- The number of *`bytes`* — that is, the length — of the *`data`*.

Return Value. `0` on success, `-1` on failure.

2.3.9.2.4. `NdbBlob::setActiveHook()`

Description. This method defines a callback for blob handle activation. The queue of prepared operations will be executed in no-commit mode up to this point; then, the callback is invoked. For additional information, see Section 2.3.9.1.1, "The `NdbBlob::ActiveHook` Type".

Signature.

```
int setActiveHook
    (
      ActiveHook*  activeHook,
      void*        arg
    )
```

Parameters. This method requires two parameters:

- A pointer to an `ActiveHook` value; this is a callback as explained in Section 2.3.9.1.1, "The `NdbBlob::ActiveHook` Type".
- A pointer to `void`, for any data to be passed to the callback.

Return Value. `0` on success, `-1` on failure.

2.3.9.2.5. `NdbBlob::getVersion()`

Description. This method is used to distinguish whether a blob operation is statement-based or event-based.

Signature.

```
void getVersion
    (
      int& version
    )
```

Parameters. This method takes a single parameter, an integer reference to the blob version (operation type).

Return Value. One of the following three values:

- `-1`: This is a "normal" (statement-based) blob.
- `0`: This is an event-operation based blob, following a change in its data.
- `1`: This is an event-operation based blob, prior to any change in its data.

> **Note**
>
> `getVersion()` is always successful, assuming that it is invoked as a method of a valid `NdbBlob` instance.

2.3.9.2.6. `NdbBlob::getNull()`

Description. This method checks whether the blob's value is `NULL`.

Signature.

```
int getNull
    (
      int& isNull
    )
```

Parameters. A reference to an integer *`isNull`*. Following invocation, this parameter has one of the following values, interpreted as shown here:

- `-1`: The blob is undefined. If this is a non-event blob, this result causes a state error.
- `0`: The blob has a non-null value.
- `1`: The blob's value is `NULL`.

Return Value. *None*.

2.3.9.2.7. `NdbBlob::setNull()`

Description. This method sets the value of a blob to `NULL`.

Signature.

```
int setNull
    (
      void
    )
```

Parameters. None.

Return Value. `0` on success; `-1` on failure.

2.3.9.2.8. `NdbBlob::getLength()`

Description. This method gets the blob's current length in bytes.

Signature.

```
int getLength
    (
      Uint64& length
    )
```

Parameters. A reference to the length.

Return Value. The blob's length in bytes. For a `NULL` blob, this method returns `0`. to distinguish between a blob whose length is `0` blob and one which is `NULL`, use the `getNull()` method.

2.3.9.2.9. `NdbBlob::truncate()`

Description. This method is used to truncate a blob to a given length.

Signature.

```
int truncate
    (
      Uint64 length = 0
    )
```

Parameters. `truncate()` takes a single parameter which specifies the new *`length`* to which the blob is to be truncated. This method has no effect if *`length`* is greater than the blob's current length (which you can check using `getLength()`).

Return Value. `0` on success, `-1` on failure.

2.3.9.2.10. `NdbBlob::getPos()`

Description. This method gets the current read/write position in a blob.

Signature.

```
int getPos
    (
      Uint64& pos
    )
```

Parameters. One parameter, a reference to the position.

Return Value. Returns `0` on success, or `-1` on failure. (Following a successful invocation, *`pos`* will hold the current read/write position within the blob, as a number of bytes from the beginning.)

2.3.9.2.11. `NdbBlob::setPos()`

Description. This method sets the position within the blob at which to read or write data.

Signature.

```
int setPos
    (
      Uint64 pos
    )
```

Parameters. The setPos() method takes a single parameter *`pos`* (an unsigned 64-bit integer), which is the position for reading or writing data. The value of *`pos`* must be between `0` and the blob's current length.

> **Important**
>
> “Sparse” blobs are not supported in the NDB API; there can be no unused data positions within a blob.

Return Value. `0` on success, `-1` on failure.

2.3.9.2.12. `NdbBlob::readData()`

Description. This method is used to read data from a blob.

Signature.

```
int readData
    (
      void*     data,
      Uint32&   bytes
    )
```

Parameters. `readData()` accepts a pointer to the data to be read, and a reference to the number of bytes read.

Return Value. `0` on success, `-1` on failure. Following a successful invocation, *`data`* points to the data that was read, and *`bytes`* holds the number of bytes read.

2.3.9.2.13. `NdbBlob::writeData()`

Description. This method is used to write data to an `NdbBlob`. After a successful invocation, the read/write position will be at the first byte following the data that was written to the blob.

> **Note**
>
> A write past the current end of the blob data extends the blob automatically.

Signature.

```
int writeData
    (
      const void*  data,
      Uint32       bytes
    )
```

Parameters. This method takes two parameters, a pointer to the *`data`* to be written, and the number of *`bytes`* to write.

Return Value. `0` on success, `-1` on failure.

2.3.9.2.14. `NdbBlob::getColumn()`

Description. Use this method to get the `BLOB` column to which the `NdbBlob` belongs.

Signature.

```
const Column* getColumn
    (
      void
    )
```

Parameters. None.

Return Value. A Column object. (See Section 2.3.1, “The `Column` Class”.)

2.3.9.2.15. `NdbBlob::getNdbError()`

Description. Use this method to obtain an error object. The error may be blob-specific or may be copied from a failed implicit operation. The error code is copied back to the operation unless the operation already has a nonzero error code.

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. None.

Return Value. An `NdbError` object. See Section 2.3.30, “The `NdbError` Structure”.

2.3.9.2.16. `NdbBlob::blobsFirstBlob()`

Description. This method initialises a list of blobs belonging to the current operation and returns the first blob in the list.

Signature.

```
NdbBlob* blobsFirstBlob
    (
      void
    )
```

Parameters. None.

Return Value. A pointer to the desired blob.

2.3.9.2.17. `NdbBlob::blobsNextBlob()`

Description. Use the method to obtain the next in a list of blobs that was initialised using `blobsFirstBlob()`. See Section 2.3.9.2.16, “`NdbBlob::blobsFirstBlob()`”.

Signature.

```
NdbBlob* blobsNextBlob
    (
      void
    )
```

Parameters. None.

Return Value. A pointer to the desired blob.

2.3.9.2.18. `NdbBlob::getBlobEventName()`

Description. This method gets a blob event name. The blob event is created if the main event monitors the blob column. The name includes the main event name.

Signature.

```
static int getBlobEventName
    (
      char*       name,
      Ndb*        ndb,
      const char* event,
      const char* column
    )
```

Parameters. This method takes 4 parameters:

- *name*: The name of the blob event.
- *ndb*: The relevant Ndb object.
- *event*: The name of the main event.
- *column*: The blob column.

Return Value. 0 on success, -1 on failure.

2.3.9.2.19. NdbBlob::getBlobTableName()

Description. This method gets the blob data segment table name.

> **Note**
>
> This method is generally of use only for testing and debugging purposes.

Signature.

```
static int getBlobTableName
    (
      char*       name,
      Ndb*        ndb,
      const char* table,
      const char* column
    )
```

Parameters. This method takes 4 parameters:

- *name*: The name of the blob data segment table.
- *ndb*: The relevant Ndb object.
- *table*: The name of the main table.
- *column*: The blob column.

Return Value. 0 on success, -1 on failure.

2.3.9.2.20. NdbBlob::getNdbOperation()

Description. This method can be used to find the operation with which the handle for this NdbBlob is associated.

NdbBlob::getNdbOperation() is available beginning with MySQL Cluster NDB 6.2.17 and MySQL Cluster NDB 6.3.19.

Signature.

```
const NdbOperation* getNdbOperation
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to an operation.

> **Important**
>
> The operation referenced by the pointer retruned by this method may be represented by either an NdbOperation or NdbScanOperation object.
>
> See Section 2.3.15, "The NdbOperation Class", and Section 2.3.18, "The NdbScanOperation Class", for more information.

2.3.10. The NdbDictionary Class

This class provides meta-information about database objects, such as tables, columns, and indexes.

While the preferred method of database object creation and deletion is through the MySQL Server, `NdbDictionary` also permits the developer to perform these tasks via the NDB API.

Parent class. *None*

Child classes. Dictionary, Column, Object

Description. This is a data dictionary class that supports enquiries about tables, columns, and indexes. It also provides ways to define these database objects and to remove them. Both sorts of functionality are supplied via inner classes that model these objects. These include the following:

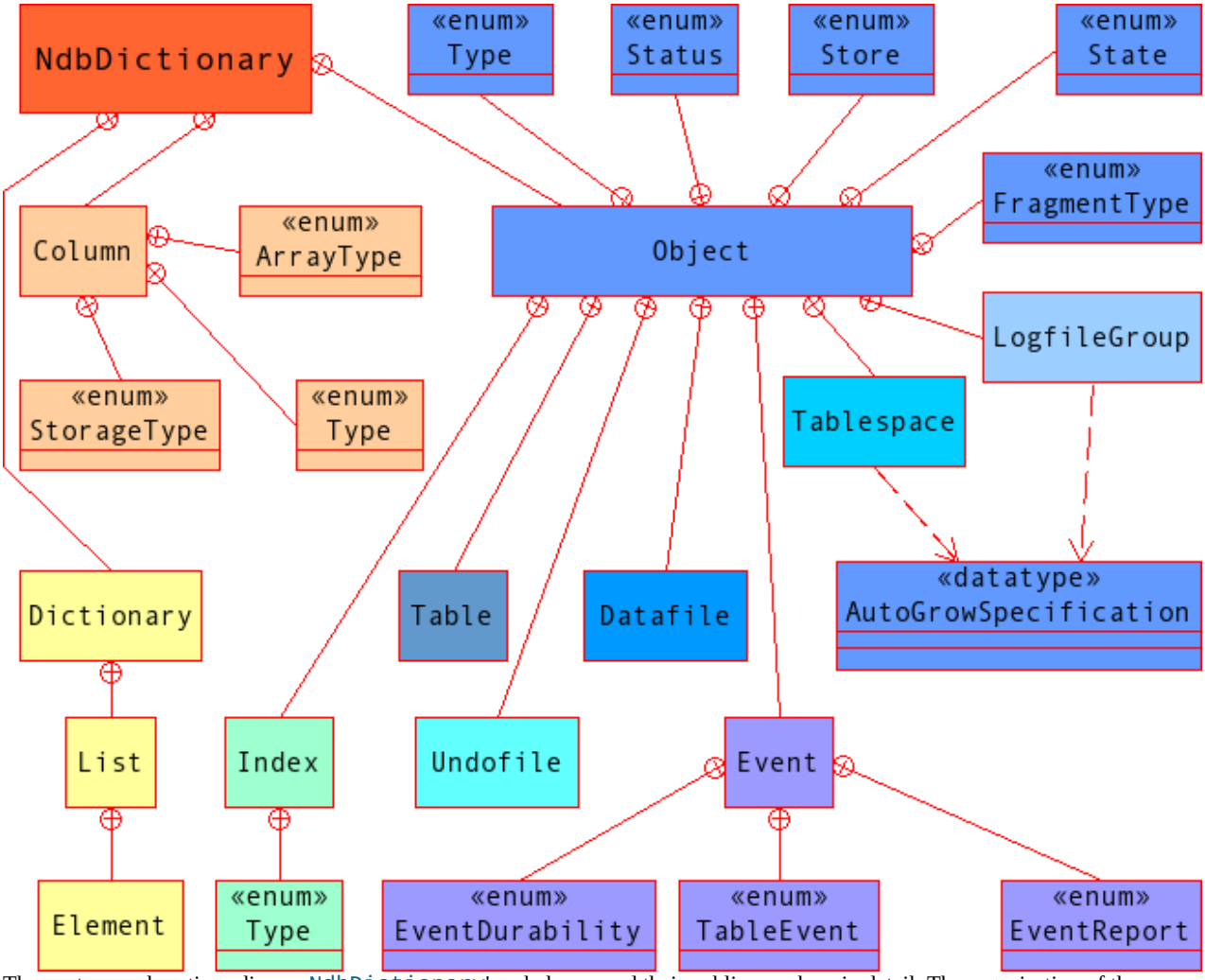
- `NdbDictionary::Object::Table` for working with tables
- `NdbDictionary::Column` for creating table columns
- `NdbDictionary::Object::Index` for working with secondary indexes
- `NdbDictionary::Dictionary` for creating database objects and making schema enquiries
- `NdbDictionary::Object::Event` for working with events in the cluster.

Additional `NdbDictionary::Object` subclasses model the tablespaces, logfile groups, datafiles, and undofiles required for working with MySQL Cluster Disk Data tables (introduced in MySQL 5.1).

> **Warning**
>
> Tables and indexes created using `NdbDictionary` cannot be viewed from the MySQL Server.
>
> Dropping indexes via the NDB API that were created originally from a MySQL Cluster causes inconsistencies. It is possible that a table from which one or more indexes have been dropped using the NDB API will no longer be usable by MySQL following such operations. In this event, the table must be dropped, and then re-created using MySQL to make it accessible to MySQL once more.

Methods. `NdbDictionary` itself has no public methods. All work is accomplished by accessing its subclasses and their public members.

`NdbDictionary` Subclass Hierarchy. This diagram shows the hierarchy made up of the `NdbDictionary` class, its subclasses, and their enumerated datatypes:



The next several sections discuss `NdbDictionary`'s subclasses and their public members in detail. The organisation of these sections reflects that of the `NdbDictionary` class hierarchy.

> **Note**
>
> For the numeric equivalents to enumerations of `NdbDictionary` subclasses, see the file `/storage/ndb/include/ndbapi/NdbDictionary.hpp` in the MySQL 5.1 source tree.

2.3.11. The `NdbEventOperation` Class

This section describes the `NdbEventOperation` class, which is used to monitor changes (events) in a database. It provides the core functionality used to implement MySQL Cluster Replication.

Parent class. *None*

Child classes. *None*

Description. `NdbEventOperation` represents a database event.

Creating an Instance of `NdbEventOperation`. This class has no public constructor or destructor. Instead, instances of `NdbEventOperation` are created as the result of method calls on `Ndb` and `NdbDictionary` objects:

1. There must exist an event which was created using `Dictionary::createEvent()`. This method returns an instance of the `NdbDictionary::Event` class. See Section 2.3.3.1.12, “`Dictionary::createEvent()`”.

   An `NdbEventOperation` object is instantiated using `Ndb::createEventOperation()`, which acts on instance of `NdbDictionary::Event`. See Section 2.3.8.1.11, “`Ndb::createEventOperation`”.

See also Section 2.3.4, “The `Event` Class”.

An instance of this class is removed by invoking `Ndb::dropEventOperation`. See Section 2.3.8.1.12, "`Ndb::dropEventOperation()`", for more information.

> **Tip**
>
> A detailed example demonstrating creation and removal of event operations is provided in Section 2.4.7, "NDB API Event Handling Example".

Known Issues. The following issues may be encountered when working with event operations in the NDB API:

- The maximum number of active `NdbEventOperation` objects is currently fixed at compile time at `2 * MaxNoOfTables`. This may become a separately configurable parameter in a future release.
- Currently, all `INSERT`, `DELETE`, and `UPDATE` events — as well as all attribute changes — are sent to the API, even if only some attributes have been specified. However, these are hidden from the user and only relevant data is shown after calling `Ndb::nextEvent()`.

  Note that false exits from `Ndb::pollEvents()` may occur, and thus the following `nextEvent()` call returns zero, since there was no available data. In such cases, simply call `pollEvents()` again.

  See Section 2.3.8.1.13, "`Ndb::pollEvents()`", and Section 2.3.8.1.14, "`Ndb::nextEvent()`".
- Event code does *not* check the table schema version. When a table is dropped, make sure that you drop any associated events.

  We intend to remedy this issue in a future release of MySQL Cluster and the NDB API.
- If you have received a complete epoch, events from this epoch are not re-sent, even in the event of a node failure. However, if a node failure has occurred, subsequent epochs may contain duplicate events, which can be identified by duplicated primary keys.

  In the MySQL Cluster replication code, duplicate primary keys on `INSERT` operations are normally handled by treating such inserts as `REPLACE` operations.

> **Tip**
>
> To view the contents of the system table containing created events, you can use the `ndb_select_all` utility as shown here:
>
> ```
> ndb_select_all -d sys 'NDB$EVENTS_0'
> ```

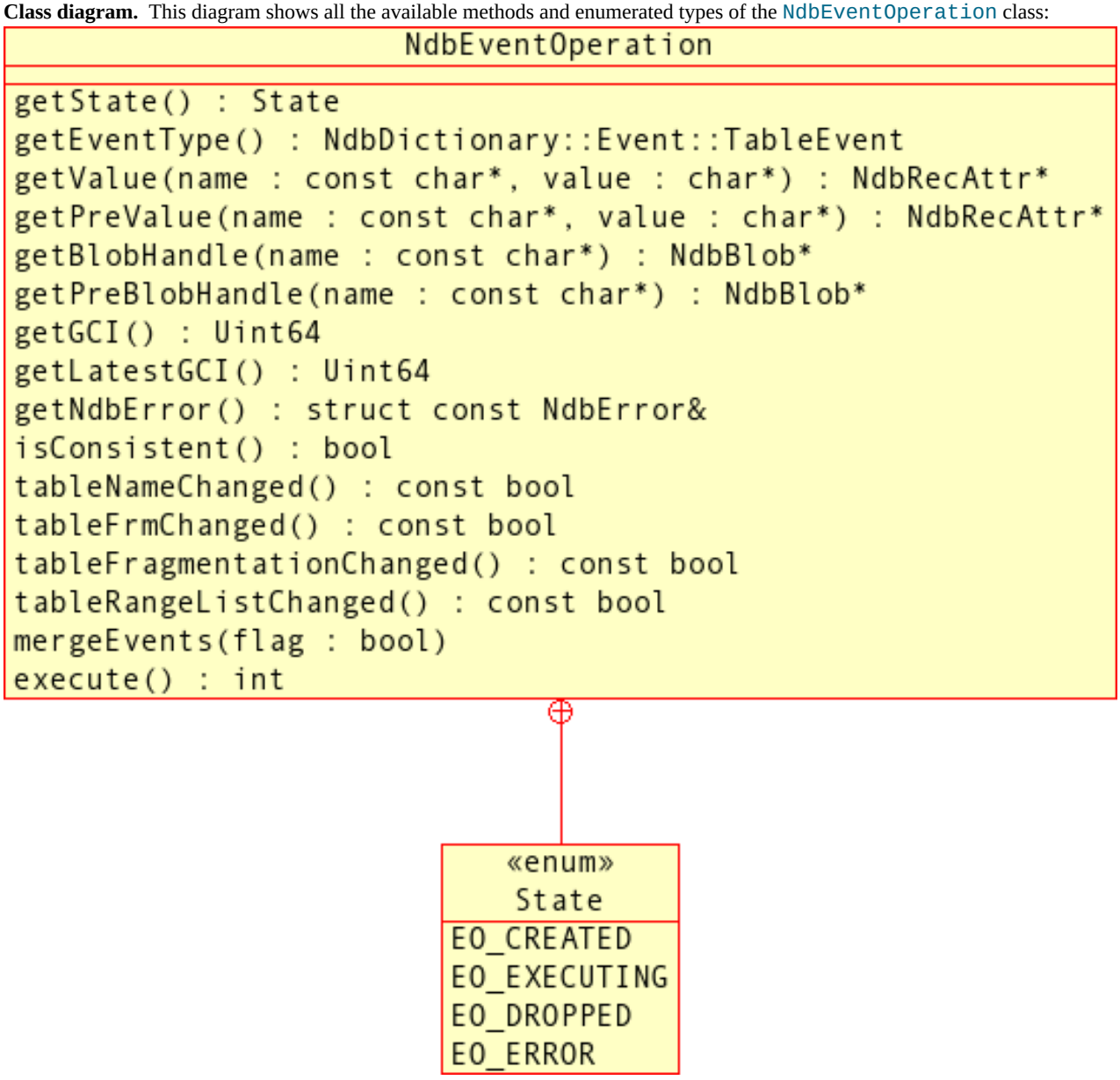
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getState()`	Gets the current state of the event operation
`getEventType()`	Gets the event type
`getValue()`	Retrieves an attribute value
`getPreValue()`	Retrieves an attribute's previous value
`getBlobHandle()`	Gets a handle for reading blob attributes
`getPreBlobHandle()`	Gets a handle for reading the previous blob attribute
`getGCI()`	Retrieves the GCI of the most recently retrieved event
`getLatestGCI()`	Retrieves the most recent GCI (whether or not the corresponding event has been retrieved)
`getNdbError()`	Gets the most recent error
`isConsistent()`	Detects event loss caused by node failure
`tableNameChanged()`	Checks to see whether the name of a table has changed
`tableFrmChanged()`	Checks to see whether a table `.FRM` file has changed
`tableFragmentationChanged()`	Checks to see whether the fragmentation for a table has changed
`tableRangeListChanged()`	Checks to see whether a table range partition list name has changed
`mergeEvents()`	Allows for events to be merged
`execute()`	Activates the `NdbEventOperation`

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.11.2, "`NdbEventOperation` Methods".

Types. `NdbEventOperation` defines one enumerated type. See Section 2.3.11.1, "The `NdbEventOperation::State` Type", for details.

Class diagram. This diagram shows all the available methods and enumerated types of the `NdbEventOperation` class:



2.3.11.1. The `NdbEventOperation::State` Type

Description. This type describes the event operation's state.

Enumeration values.

Value	Description
`EO_CREATED`	The event operation has been created, but `execute()` has not yet been called.
`EO_EXECUTING`	The `execute()` method has been invoked for this event operation.
`EO_DROPPED`	The event operation is waiting to be deleted, and is no longer usable.
`EO_ERROR`	An error has occurred, and the event operation is unusable.

A `State` value is returned by the `getState()` method. See Section 2.3.11.2.1, "`NdbEventOperation::getState()`", for more information.

2.3.11.2. `NdbEventOperation` Methods

This section contains definitions and descriptions of the public methods of the `NdbEventOperation` class.

2.3.11.2.1. `NdbEventOperation::getState()`

Description. This method gets the event operation's current state.

Signature.

```
State getState
    (
      void
    )
```

Parameters. *None*.

Return Value. A `State` value. See Section 2.3.11.1, "The `NdbEventOperation::State` Type".

2.3.11.2.2. `NdbEventOperation::getEventType()`

Description. This method is used to obtain the event's type (`TableEvent`).

Signature.

```
NdbDictionary::Event::TableEvent getEventType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `TableEvent` value. See Section 2.3.4.1.1, "The `Event::TableEvent` Type".

2.3.11.2.3. `NdbEventOperation::getValue()`

Description. This method defines the retrieval of an attribute value. The NDB API allocates memory for the `NdbRecAttr` object that is to hold the returned attribute value.

> **Important**
>
> This method does *not* fetch the attribute value from the database, and the `NdbRecAttr` object returned by this method is not readable or printable before calling the `execute()` method and `Ndb::nextEvent()` has returned a non-`NULL` value.
>
> If a specific attribute has not changed, the corresponding `NdbRecAttr` will be in the state `UNDEFINED`. This can be checked by using `NdbRecAttr::isNULL()` which in such cases returns `-1`.

***`value`* Buffer Memory Allocation.** It is the application's responsibility to allocate sufficient memory for the *`value`* buffer (if not `NULL`), and this buffer must be aligned appropriately. The buffer is used directly (thus avoiding a copy penalty) only if it is aligned on a 4-byte boundary and the attribute size in bytes (calculated as `NdbRecAttr::attrSize()` times `NdbRecAttr::arraySize()`) is a multiple of 4.

> **Note**
>
> `getValue()` retrieves the current value. Use `getPreValue()` for retrieving the previous value. See Section 2.3.11.2.4, "`NdbEventOperation::getPreValue()`".

Signature.

```
NdbRecAttr* getValue
    (
      const char* name,
      char*       value = 0
    )
```

Parameters. This method takes two parameters:

- The *`name`* of the attribute (as a constant character pointer).

- A pointer to a *value*:
  - If the attribute value is not NULL, then the attribute value is returned in this parameter.
  - If the attribute value is NULL, then the attribute value is stored only in the NdbRecAttr object returned by this method. See *value* Buffer Memory Allocation for more information regarding this parameter.

Return Value. An NdbRecAttr object to hold the value of the attribute, or a NULL pointer indicating that an error has occurred. See Section 2.3.16, "The NdbRecAttr Class".

2.3.11.2.4. NdbEventOperation::getPreValue()

Description. This method performs identically to getValue(), except that it is used to define a retrieval operation of an attribute's previous value rather than the current value. See Section 2.3.11.2.3, "NdbEventOperation::getValue()", for details.

Signature.

```
NdbRecAttr* getPreValue
    (
      const char* name,
      char*       value = 0
    )
```

Parameters. This method takes two parameters:

- The *name* of the attribute (as a constant character pointer).
- A pointer to a *value*:
  - If the attribute value is not NULL, then the attribute value is returned in this parameter.
  - If the attribute value is NULL, then the attribute value is stored only in the NdbRecAttr object returned by this method. See *value* Buffer Memory Allocation for more information regarding this parameter.

Return Value. An NdbRecAttr object to hold the value of the attribute, or a NULL pointer indicating that an error has occurred. See Section 2.3.16, "The NdbRecAttr Class".

2.3.11.2.5. NdbEventOperation::getBlobHandle()

Description. This method is used in place of getValue() for blob attributes. The blob handle (NdbBlob) returned by this method supports read operations only.

> **Note**
>
> To obtain the previous value for a blob attribute, use getPreBlobHandle().

Signature.

```
NdbBlob* getBlobHandle
    (
      const char* name
    )
```

Parameters. The *name* of the blob attribute.

Return Value. A pointer to an NdbBlob object. See Section 2.3.9, "The NdbBlob Class".

2.3.11.2.6. NdbEventOperation::getPreBlobHandle()

Description. This function is the same as getBlobHandle(), except that it is used to access the previous value of the blob attribute. See Section 2.3.11.2.5, "NdbEventOperation::getBlobHandle()".

Signature.

```
NdbBlob* getPreBlobHandle
  (
    const char* name
  )
```

Parameters. The *name* of the blob attribute.

Return Value. A pointer to an `NdbBlob`. See Section 2.3.9, "The `NdbBlob` Class".

2.3.11.2.7. `NdbEventOperation::getGCI()`

Description. This method retrieves the GCI for the most recently retrieved event.

Signature.

```
Uint64 getGCI
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The global checkpoint index of the most recently retrieved event (an integer).

2.3.11.2.8. `NdbEventOperation::getLatestGCI()`

Description. This method retrieves the most recent GCI.

Beginning with MySQL Cluster NDB 6.2.5, this method actually returns the latest epoch number, and all references to GCIs in the documentation for this method when using this or a later MySQL Cluster NDB version should be taken to mean epoch numbers instead. This is a consequence of the implementation for micro-CGPs.

> **Note**
>
> The GCI obtained using this method is not necessarily associated with an event.

Signature.

```
Uint64 getLatestGCI
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The index of the latest global checkpoint, an integer.

2.3.11.2.9. `NdbEventOperation::getNdbError()`

Description. This method retrieves the most recent error.

Signature.

```
const struct NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an `NdbError` structure. See Section 2.3.30, "The `NdbError` Structure".

2.3.11.2.10. `NdbEventOperation::isConsistent()`

Description. This method is used to determine whether event loss has taken place following the failure of a node.

Signature.

```
bool isConsistent
    (
      void
    ) const
```

Parameters. *None*.

Return Value. If event loss has taken place, then this method returns `false`; otherwise, it returns `true`.

2.3.11.2.11. NdbEventOperation::tableNameChanged()

Description. This method tests whether a table name has changed as the result of a TE_ALTER table event. (See Section 2.3.4.1.1, "The Event::TableEvent Type".)

Signature.

```
const bool tableNameChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns true if the name of the table has changed; otherwise, the method returns false.

2.3.11.2.12. NdbEventOperation::tableFrmChanged()

Description. Use this method to determine whether a table .FRM file has changed in connection with a TE_ALTER event. (See Section 2.3.4.1.1, "The Event::TableEvent Type".)

Signature.

```
const bool tableFrmChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns true if the table .FRM file has changed; otherwise, the method returns false.

2.3.11.2.13. NdbEventOperation::tableFragmentationChanged()

Description. This method is used to test whether a table's fragmentation has changed in connection with a TE_ALTER event. (See Section 2.3.4.1.1, "The Event::TableEvent Type".)

Signature.

```
const bool tableFragmentationChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns true if the table's fragmentation has changed; otherwise, the method returns false.

2.3.11.2.14. NdbEventOperation::tableRangeListChanged()

Description. Use this method to check whether a table range partition list name has changed in connection with a TE_ALTER event.

Signature.

```
const bool tableRangeListChanged
    (
      void
    ) const
```

Parameters. *None*.

Return Value. This method returns true if range or list partition name has changed; otherwise it returns false.

2.3.11.2.15. NdbEventOperation::mergeEvents()

Description. This method is used to set the merge events flag. For information about event merging, see Section 2.3.4.2.20, "Event::mergeEvents()".

> **Note**
>
> The merge events flag is false by default.

Signature.

```
void mergeEvents
    (
      bool flag
    )
```

Parameters. A Boolean *`flag`*.

Return Value. *None*.

2.3.11.2.16. `NdbEventOperation::execute()`

Description. Activates the `NdbEventOperation`, so that it can begin receiving events. Changed attribute values may be retrieved after `Ndb::nextEvent()` has returned a value other than `NULL`. One of `getValue()`, `getPreValue()`, `getBlobValue()`, or `getPreBlobValue()` must be called before invoking `execute()`.

> **Important**
>
> Before attempting to use this method, you should have read the explanations provided in Section 2.3.8.1.14, "`Ndb::nextEvent()`", and Section 2.3.11.2.3, "`NdbEventOperation::getValue()`". Also see Section 2.3.11, "The `NdbEventOperation` Class".

Signature.

```
int execute
    (
      void
    )
```

Parameters. *None*.

Return Value. This method returns `0` on success and `-1` on failure.

2.3.12. The `NdbIndexOperation` Class

This section describes the `NdbIndexOperation` class and its public methods.

Parent class. NdbOperation

Child classes. *None*

Description. `NdbIndexOperation` represents an index operation for use in transactions. This class inherits from `NdbOperation`; see Section 2.3.15, "The `NdbOperation` Class", for more information.

> **Note**
>
> `NdbIndexOperation` can be used only with unique hash indexes; to work with ordered indexes, use `NdbIndexScanOperation`. See Section 2.3.13, "The `NdbIndexScanOperation` Class".

Methods. The following table lists the public methods of this class and the purpose or use of each method:

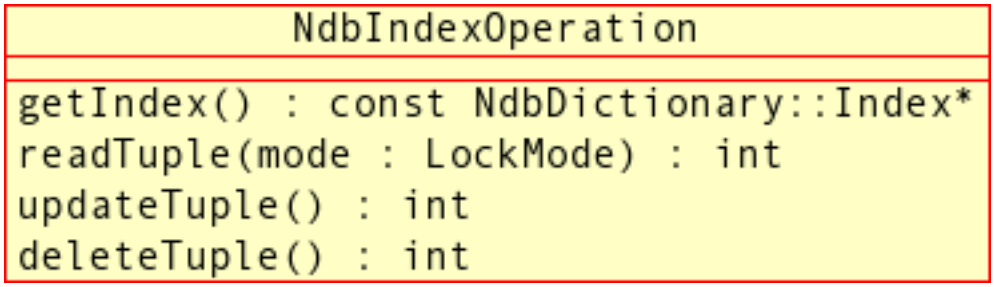
Method	Purpose / Use
`getIndex()`	Gets the index used by the operation
`readTuple()`	Reads a tuple from a table
`updateTuple()`	Updates an existing tuple in a table
`deleteTuple()`	Removes a tuple from a table

> **Note**
>
> Index operations are not permitted to insert tuples.

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.15.2, "`NdbOperation` Methods".

Types. The `NdbIndexOperation` class defines no public types of its own.

Class diagram. This diagram shows all the available methods of the `NdbIndexOperation` class:



> **Note**
>
> For more information about the use of `NdbIndexOperation`, see Section 1.3.2.3.1, "Single-row operations".

2.3.12.1. `NdbIndexOperation` Methods

This section lists and describes the public methods of the `NdbIndexOperation` class.

> **Note**
>
> This class has no public constructor. To create an instance of `NdbIndexOperation`, it is necessary to use the `NdbTransaction::getNdbIndexOperation()` method. See Section 2.3.19.2.4, "`NdbTransaction::getNdbIndexOperation()`".

2.3.12.1.1. `NdbIndexOperation::getIndex()`

Description.

Signature.

```
const NdbDictionary::Index* getIndex
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to an `Index` object. See Section 2.3.5, "The `Index` Class".

2.3.12.1.2. `NdbIndexOperation::readTuple()`

Description. This method define the `NdbIndexOperation` as a `READ` operation. When the `NdbTransaction::execute()` method is invoked, the operation reads a tuple. See Section 2.3.19.2.5, "`NdbTransaction::execute()`".

Signature.

```
int readTuple
    (
      LockMode mode
    )
```

Parameters. *`mode`* specifies the locking mode used by the read operation. See Section 2.3.15.1.3, "The `NdbOperation::LockMode` Type", for possible values.

Return Value. `0` on success, `-1` on failure.

2.3.12.1.3. `NdbIndexOperation::updateTuple()`

Description. This method defines the `NdbIndexOperation` as an `UPDATE` operation. When the `NdbTransaction::execute()` method is invoked, the operation updates a tuple found in the table. See Section 2.3.19.2.5, "`NdbTransaction::execute()`".

Signature.

```
int updateTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.12.1.4. `NdbIndexOperation::deleteTuple()`

Description. This method defines the `NdbIndexOperation` as a `DELETE` operation. When the `NdbTransaction::execute()` method is invoked, the operation deletes a tuple from the table. See Section 2.3.19.2.5, "`NdbTransaction::execute()`".

Signature.

```
int deleteTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.13. The `NdbIndexScanOperation` Class

This section discusses the `NdbIndexScanOperation` class and its public members.

Parent class. NdbScanOperation

Child classes. *None*

Description. The `NdbIndexScanOperation` class represents a scan operation using an ordered index. This class inherits from `NdbScanOperation` and `NdbOperation`. See Section 2.3.18, "The `NdbScanOperation` Class", and Section 2.3.15, "The `NdbOperation` Class", for more information about these classes.

> **Note**
>
> `NdbIndexScanOperation` is for use with ordered indexes only; to work with unique hash indexes, use `NdbIndexOperation`.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

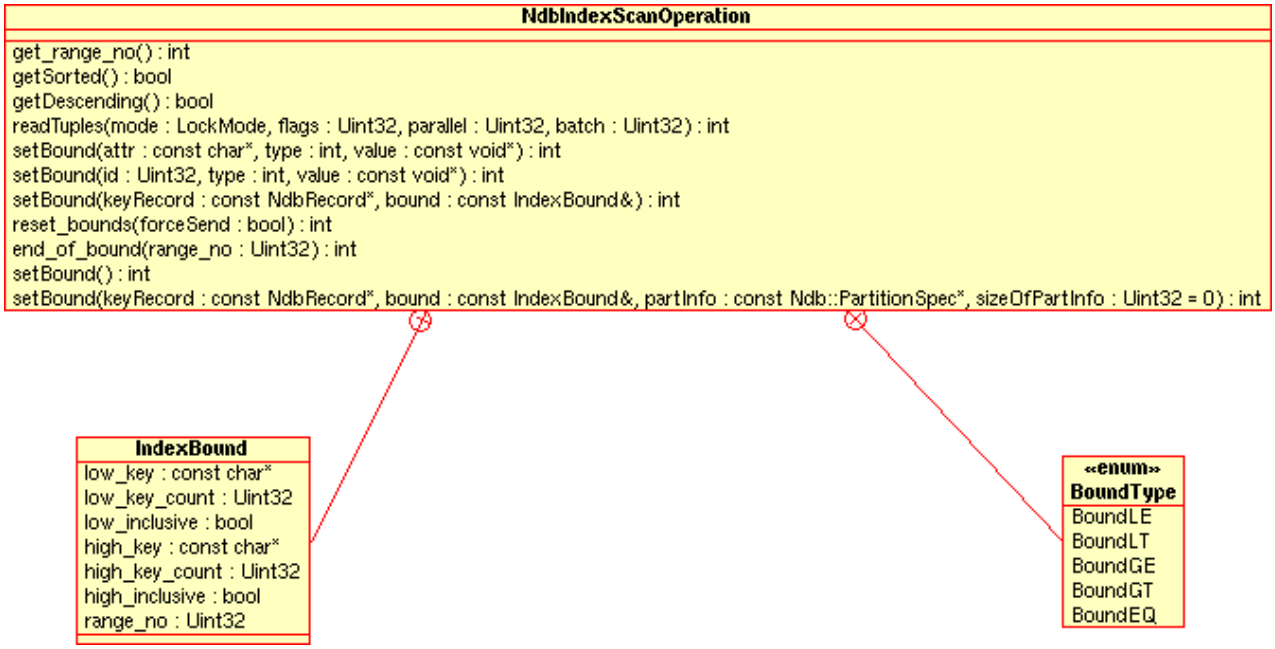
Method	Purpose / Use
`get_range_no()`	Gets the range number for the current row
`getSorted()`	Checks whether the current scan is sorted
`getDescending()`	Checks whether the current scan is sorted
`readTuples()`	Reads tuples using an ordered index
`setBound()`	Defines a bound on the index key for a range scan
`reset_bounds()`	Resets bounds, puts the operation in the send queue
`end_of_bound()`	Marks the end of a bound

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.13.2, "`NdbIndexScanOperation` Methods".

Types. The `NdbIndexScanOperation` class defines one public type. See Section 2.3.13.1, "The `NdbIndexScanOperation::BoundType` Type".

Beginning with MySQL Cluster NDB 6.2.3, this class defines an additional struct for use with operations employing `NdbRecord`; see Section 2.3.28, "The `IndexBound` Structure", and Section 2.3.25, "The `NdbRecord` Interface", for more information.

Class diagram. This diagram shows all the public members of the `NdbIndexScanOperation` class:



Note

For more information about the use of `NdbIndexScanOperation`, see Section 1.3.2.3.2, “Scan Operations”, and Section 1.3.2.3.3, “Using Scans to Update or Delete Rows”

2.3.13.1. The `NdbIndexScanOperation::BoundType` Type

Description. This type is used to describe an ordered key bound.

Tip

The numeric values are fixed in the API and can be used explicitly — in other words, it is “safe” to calculate the values and use them.

Enumeration values.

Value	Numeric Value	Description
`BoundLE`	`0`	Lower bound
`BoundLT`	`1`	Strict lower bound
`BoundGE`	`2`	Upper bound
`BoundGT`	`3`	Strict upper bound
`BoundEQ`	`4`	Equality

2.3.13.2. `NdbIndexScanOperation` Methods

This section lists and describes the public methods of the `NdbIndexScanOperation` class.

2.3.13.2.1. `NdbIndexScanOperation::get_range_no()`

Description. This method returns the range number for the current row.

Signature.

```
int get_range_no
    (
      void
    )
```

Parameters. *None*.

Return Value. The range number (an integer).

2.3.13.2.2. NdbIndexScanOperation::getSorted()

Description. This method is used to check whether the scan is sorted.

Signature.

```
bool getSorted
    (
      void
    ) const
```

Parameters. *None*.

Return Value. `true` if the scan is sorted, otherwise `false`.

2.3.13.2.3. NdbIndexScanOperation::getDescending()

Description. This method is used to check whether the scan is descending.

Signature.

```
bool getDescending
    (
      void
    ) const
```

Parameters. *None*.

Return Value. This method returns `true` if the scan is sorted in descending order; otherwise, it returns `false`.

2.3.13.2.4. NdbIndexScanOperation::readTuples()

Description. This method is used to read tuples, using an ordered index.

Signature.

```
virtual int readTuples
    (
      LockMode mode = LM_Read,
      Uint32   flags = 0,
      Uint32   parallel = 0,
      Uint32   batch = 0

    )
```

Parameters. The `readTuples()` method takes 3 parameters, as listed here:

- The lock *`mode`* used for the scan. This is a `LockMode` value; see Section 2.3.15.1.3, "The `NdbOperation::LockMode` Type" for more information, including permitted values.
- One or more scan flags; multiple *`flags`* are `OR`'ed together as they are when used with `NdbScanOperation::readTuples()`. See Section 2.3.18.1, "The `NdbScanOperation::ScanFlag` Type" for possible values.
- The number of fragments to scan in *`parallel`*; use `0` to specify the maximum automatically.
- The *`batch`* parameter specifies how many records will be returned to the client from the server by the next `NdbScanOperation::nextResult(true)` method call. Use `0` to specify the maximum automatically.

  > **Note**
  >
  > This parameter was ignored prior to MySQL 5.1.12, and the maximum was used.(Bug#20252)

Return Value. An integer: `0` indicates success; `-1` indicates failure.

2.3.13.2.5. NdbIndexScanOperation::setBound

Description. This method defines a bound on an index key used in a range scan. In MySQL Cluster NDB 6.2.3 and later, it is also sets bounds for index scans defined using `NdbRecord`.

"Old" API usage (prior to introduction of `NdbRecord`). Each index key can have a lower bound, upper bound, or both. Setting the key equal to a value defines both upper and lower bounds. Bounds can be defined in any order. Conflicting definitions gives

rise to an error.

Bounds must be set on initial sequences of index keys, and all but possibly the last bound must be non-strict. This means, for example, that “a >= 2 AND b > 3” is permissible, but “a > 2 AND b >= 3” is not.

The scan may currently return tuples for which the bounds are not satisfied. For example, a `<= 2 && b <= 3` not only scans the index up to `(a=2, b=3)`, but also returns any `(a=1, b=4)` as well.

When setting bounds based on equality, it is better to use `BoundEQ` instead of the equivalent pair `BoundLE` and `BoundGE`. This is especially true when the table partition key is a prefix of the index key.

`NULL` is considered less than any non-`NULL` value and equal to another `NULL` value. To perform comparisons with `NULL`, use `setBound()` with a null pointer (`0`).

An index also stores all-`NULL` keys as well, and performing an index scan with an empty bound set returns all tuples from the table.

Signature (“Old” API).

```
int setBound
    (
      const char* name,
      int         type,
      const void* value
    )
```

or

```
int setBound
    (
      Uint32      id,
      int         type,
      const void* value
    )
```

Parameters (“Old” API). This method takes 3 parameters:

- Either the *`name`* or the *`id`* of the attribute on which the bound is to be set.
- The bound `type` — see Section 2.3.13.1, “The `NdbIndexScanOperation::BoundType` Type”.
- A pointer to the bound *`value`* (use `0` for `NULL`).

As used with `NdbRecord` (MySQL Cluster NDB 6.2.3 and later). This method is called to add a range to an `IndexScan` operation which has been defined with a call to `NdbTransaction::scanIndex()`. To add more than one range, the index scan operation must have been defined with the `SF_MultiRange` flag set. (See Section 2.3.18.1, “The `NdbScanOperation::ScanFlag` Type”.)

> **Note**
>
> Where multiple numbered ranges are defined with multiple calls to `setBound()`, and the scan is ordered, the range number for each range must be larger than the range number for the previously defined range.

Signature (when used with `NdbRecord`). MySQL Cluster NDB 6.2.3 and later:

```
int setBound
    (
      const NdbRecord* keyRecord,
      const IndexBound& bound
    )
```

Parameters. As used with `NdbRecord` in MySQL Cluster NDB 6.2.3 and later, this method takes 2 parameters:

- *`keyRecord`*: This is an `NdbRecord` structure corresponding to the key on which the index is defined.
- The *`bound`* to add (see Section 2.3.28, “The `IndexBound` Structure”).

Starting with MySQL Cluster NDB 6.3.24 and NDB 7.0.4, an additional version of this method is available, which can be used when the application knows that rows in-range will be found only within a particular partition. This is the same as that shown previously, except for the addition of a `PartitionSpecification`. Doing so limits the scan to a single partition, improving system efficiency.

Signature (when specifying a partition).

```
int setBound
    (
      const NdbRecord* keyRecord,
      const IndexBound& bound,
      const Ndb::PartitionSpec* partInfo,
      Uint32 sizeOfPartInfo = 0
    )
```

Parameters (when specifying a partition). Beginning with MySQL Cluster NDB 6.3.24 and MySQL Cluster NDB 7.0.4, this method can be invoked with the following 4 parameters:

- *keyRecord*: This is an `NdbRecord` structure (see Section 2.3.25, "The `NdbRecord` Interface") corresponding to the key on which the index is defined.
- The *bound* to be added to the scan (see Section 2.3.28, "The `IndexBound` Structure").

  > **Note**
  >
  > *keyRecord* and *bound* are defined and used in the same way as with the 2-parameter version of this method.

- *partInfo*: This is a pointer to an `Ndb::PartitionSpec`, which provides extra information making it possible to scan a reduced set of partitions. See Section 2.3.31, "The `PartitionSpec` Structure", for more information.
- *sizeOfPartInfo*: The length of the partition specification.

Return Value. Returns `0` on success, `-1` on failure.

2.3.13.2.6. `NdbIndexScanOperation::reset_bounds()`

Description. Reset the bounds, and put the operation into the list that will be sent on the next `NdbTransaction::execute()` call.

Signature.

```
int reset_bounds
    (
      bool forceSend = false
    )
```

Parameters. Set `forceSend` to true in order to force the operation to be sent immediately.

Return Value. `0` on success, `-1` on failure.

2.3.13.2.7. `NdbIndexScanOperation::end_of_bound()`

Description. This method is used to mark the end of a bound; used when batching index reads (that is, when employing multiple ranges).

Signature.

```
int end_of_bound
    (
      Uint32 range_no
    )
```

Parameters. The number of the range on which the bound occurs.

Return Value. `0` indicates success; `-1` indicates failure.

2.3.14. The `NdbInterpretedCode` Class

This section discusses the `NdbInterpretedCode` class, which can be used to prepare and execute an NDB interpreted program.

Beginning with MySQL Cluster NDB 6.2.14 and MySQL Cluster 6.3.12, you *must* use the `NdbInterpretedCode` class (rather than `NdbScanOperation`) to write interpreted programs used for scans.

Parent class. *None*.

Child classes. *None*.

Description. `NdbInterpretedCode` represents an interpreted program for use in operations created using `NdbRecord` (see Section 2.3.25, "The `NdbRecord` Interface"), or with scans created using the old API. The `NdbScanFilter` class can also be used to generate an NDB interpreted program using this class. (See Section 2.3.17, "The `NdbScanFilter` Class".) This class was added in MySQL Cluster NDB 6.2.14 and 6.3.12.

> **Important**
>
> This interface is still under development, and so is subject to change without notice. The `NdbScanFilter` API is a more stable API for defining scanning and filtering programs. See Section 2.3.17, "The `NdbScanFilter` Class", for more information.

Using `NdbInterpretedCode`. To create an `NdbInterpretedCode` object, invoke the constructor, optionally supplying a table for the program to operate on, and a buffer for program storage and finalization. If no table is supplied, then only instructions which do not access table attributes can be used.

> **Note**
>
> Each NDB API operation applies to one table, and so does any `NdbInterpretedCode` program attached to that operation.

If no buffer is supplied, then an internal buffer is dynamically allocated and extended as necessary. Once the `NdbInterpretedCode` object is created, you can add instructions and labels to it by calling the appropriate methods as described later in this section. When the program has completed, finalize it by calling the `finalise()` method, which resolves any remaining internal branches and calls to label and subroutine offsets.

> **Note**
>
> A single finalized `NdbInterpretedCode` program can be used by more than one operation. It need not be re-prepared for successive operations.

To use the program with `NdbRecord` operations and scans, pass it at operation definition time via the `OperationOptions` or `ScanOptions` parameter. Alternatively, you can use the program with old-style API scans by passing it via the `setInterpretedProgram()` method. When the program is no longer required, the `NdbInterpretedCode` object can be deleted, along with any user-supplied buffer.

Error checking. For reasons of efficiency, methods of this class provide minimal error checking.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

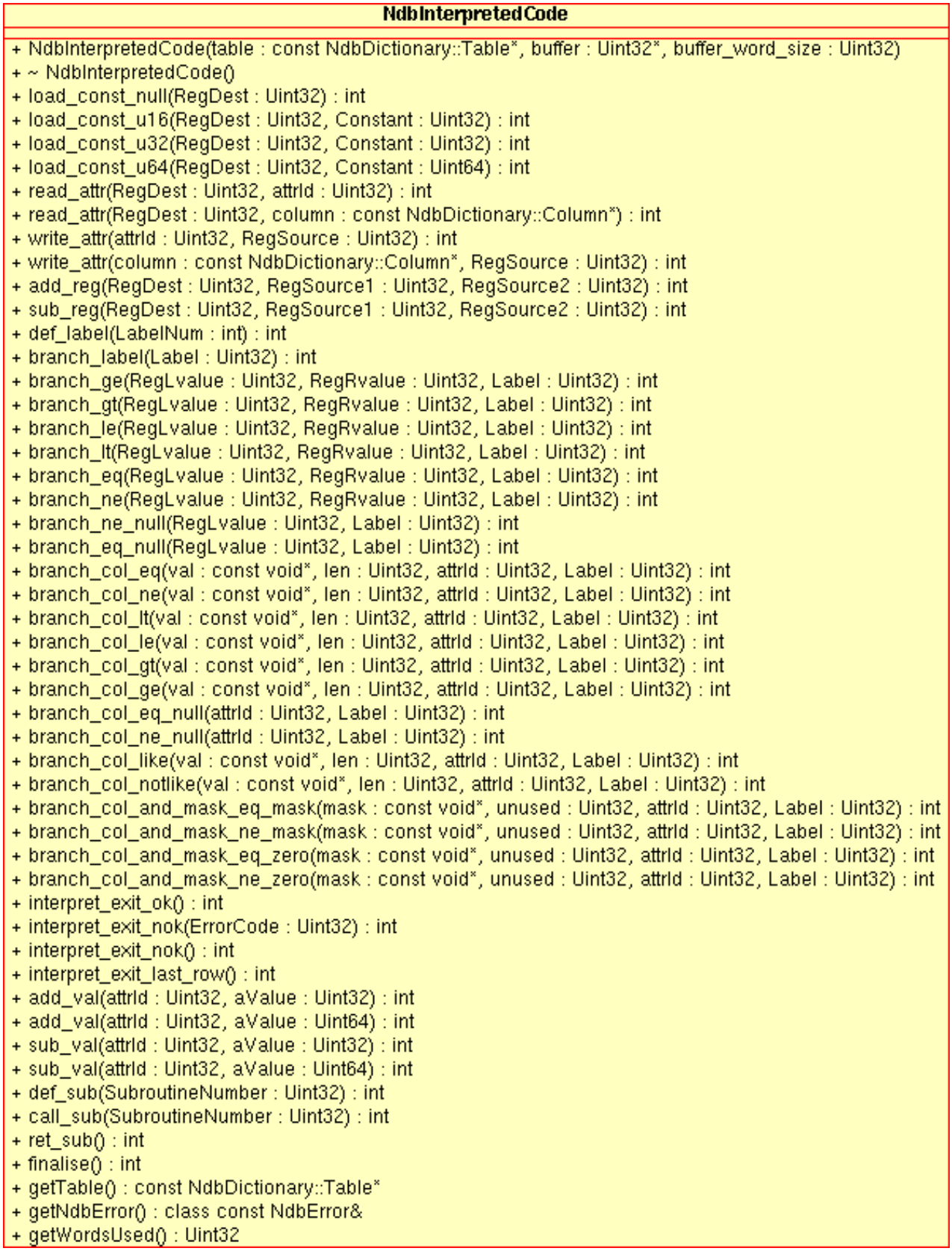
Method	Purpose / Use
`NdbInterpretedCode()`	Class constructor
`load_const_null()`	Load a `NULL` value into a register
`load_const_u16()`	Load a 16-bit numeric value into a register
`load_const_u32()`	Load a 32-bit numeric value into a register
`load_const_u64()`	Load a 64-bit numeric value into a register
`read_attr()`	Read a register value into a table column
`write_attr()`	Write a table column value into a register
`add_reg()`	Add two register values and store the result in a third register
`sub_reg()`	Subtract two register values and store the result in a third register
`def_label()`	Create a label for use within the interpreted program
`branch_label()`	Unconditional jump to a label
`branch_ge()`	Jump if one register value is greater than or equal to another
`branch_gt()`	Jump if one register value is greater than another
`branch_le()`	Jump if one register value is less than or equal to another
`branch_lt()`	Jump if one register value is less than another
`branch_eq()`	Jump if one register value is equal to another
`branch_ne()`	Jump if one register value is not equal to another
`branch_ne_null()`	Jump if a register value is not `NULL`
`branch_eq_null()`	Jump if a register value is `NULL`
`branch_col_eq()`	Jump if a column value is equal to another
`branch_col_ne()`	Jump if a column value is not equal to another

Method	Purpose / Use
branch_col_lt()	Jump if a column value is less than another
branch_col_le()	Jump if a column value is less than or equal to another
branch_col_gt()	Jump if a column value is greater than another
branch_col_ge()	Jump if a column value is greater than or equal to another
branch_col_eq_null()	Jump if a column value is NULL
branch_col_ne_null()	Jump if a column value is not NULL
branch_col_like()	Jump if a column value matches a pattern
branch_col_notlike()	Jump if a column value does not match a pattern
branch_col_and_mask_eq_mask()	Jump if a column value ANDed with a bitmask is equal to the bitmask
branch_col_and_mask_ne_mask()	Jump if a column value ANDed with a bitmask is not equal to the bitmask
branch_col_and_mask_eq_zero()	Jump if a column value ANDed with a bitmask is equal to 0
branch_col_and_mask_ne_zero()	Jump if a column value ANDed with a bitmask is not equal to 0
interpret_exit_ok()	Return a row as part of the result
interpret_exit_nok()	Do not return a row as part of the result
interpret_last_row()	Return a row as part of the result, and do not check any more rows in this fragment
add_val()	Add a value to a table column value
sub_val()	Subtract a value from a table column value
def_sub()	Define a subroutine
call_sub()	Call a subroutine
ret_sub()	Return from a subroutine
finalise()	Completes interpreted program and prepares it for use
getTable()	Gets the table on which the program is defined
getNdbError()	Gets the most recent error associated with this NdbInterpretedCode object
getWordsUsed()	Gets the number of words used in the buffer

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.14.1, “NdbInterpreted-Code Methods”.

Types. This class defines no public types.

Class diagram. This diagram shows all the available methods of the NdbInterpretedCode class:



2.3.14.1. `NdbInterpretedCode` Methods

2.3.14.1.1. `NdbInterpretedCode` Constructor

Description. This is the `NdbInterpretedCode` class constuctor.

Signature.

```
NdbInterpretedCode
    (
      const NdbDictionary::Table* table = 0,
      Uint32* buffer = 0,
      Uint32 buffer_word_size = 0
    )
```

Parameters. The `NdbInterpretedCode` constructor takes three parameters, as described here:

- The *`table`* against which which this program is to be be run. This parameter must be supplied if the program is table-specific — that is, if it reads from or writes to columns in a table.
- A pointer to a *`buffer`* of 32-bit words used to store the program.
- *`buffer_word_size`* is the length of the buffer passed in. If the program exceeds this length then adding new instructions will fail with error `4518 TOO MANY INSTRUCTIONS IN INTERPRETED PROGRAM`.

  Alternatively, if no buffer is passed, a buffer will be dynamically allocated internally and extended to cope as instructions are added.

Return Value. An instance of `NdbInterpretedCode`.

2.3.14.1.2. `NdbInterpretedCode` Methods for Loading Values into Registers

The methods described in this section are used to load constant values into `NdbInterpretedCode` program registers. The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`load_const_null()`	1	1
`load_const_u16()`	1	1
`load_const_u32()`	2	2
`load_const_u64()`	3	3

2.3.14.1.2.1. `NdbInterpretedCode::load_const_null()`

Description. This method is used to load a `NULL` value into a register.

Signature.

```
int load_const_null
    (
      Uint32 RegDest
    )
```

Parameters. This method takes a single parameter, the register into which to place the `NULL`.

Return Value. Returns `0` on success, `-1` otherwise.

2.3.14.1.2.2. `NdbInterpretedCode::load_const_u16()`

Description. This method loads a 16-bit value into the specified interpreter register.

Signature.

```
int load_const_u16
    (
      Uint32 RegDest,
      Uint32 Constant
    )
```

Parameters. This method takes two parameters:

- *`RegDest`*: The register into which the value should be loaded.
- A *`Constant`* value to be loaded

Return Value. Returns `0` on success, `-1` otherwise.

2.3.14.1.2.3. `NdbInterpretedCode::load_const_u32()`

Description. This method loads a 32-bit value into the specified interpreter register.

Signature.

```
int load_const_u32
    (
      Uint32 RegDest,
      Uint32 Constant
    )
```

Parameters. This method takes two parameters:

- *`RegDest`*: The register into which the value should be loaded.
- A *`Constant`* value to be loaded

Return Value. Returns `0` on success, `-1` otherwise.

2.3.14.1.2.4. `NdbInterpretedCode::load_const_u64()`

Description. This method loads a 64-bit value into the specified interpreter register.

Signature.

```
int load_const_u64
    (
      Uint32 RegDest,
      Uint64 Constant
    )
```

Parameters. This method takes two parameters:

- *`RegDest`*: The register into which the value should be loaded.
- A *`Constant`* value to be loaded

Return Value. Returns `0` on success, `-1` otherwise.

2.3.14.1.3. `NdbInterpretedCode` Methods for Copying Values Between Registers and Table Columns

This class provides two methods for copying values between a column in the current table row and a program register. The `read_attr()` method is used to copy a table column value into a program register; `write_attr()` is used to copy a value from a program register into a table column. Both of these methods require that the table being operated on was specified when creating the `NdbInterpretedCode` object for which they are called.

The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`read_attr()`	1	1
`write_attr()`	1	1

More detailed information may be found in the next two sections.

2.3.14.1.3.1. `NdbInterpretedCode::read_attr()`

Description. The `read_attr()` method is used to read a table column value into a program register. The column may be specified either by using its attribute ID or as a pointer to a `Column` object (Section 2.3.1, “The `Column` Class”).

Signature. Referencing the column by its attribute ID:

```
int read_attr
```

```
    (
      Uint32 RegDest,
      Uint32 attrId
    )
```

Referencing the column as a `Column` object:

```
int read_attr
    (
      Uint32 RegDest,
      const NdbDictionary::Column* column
    )
```

Parameters. This method takes two parameters:

- The register to which the column value is to be copied (*`RegDest`*).
- Either of the following references to the table column whose value is to be copied:
  - The table column's attribute ID (*`attrId`*)
  - A pointer to a *`column`* — that is, a pointer to an `NdbDictionary::Column` object referencing the table column

Return Value. `0` on success; `-1` on failure.

2.3.14.1.3.2. `NdbInterpretedCode::write_attr()`

Description. This method is used to copy a register value to a table column. The column may be specified either by using its attribute ID or as a pointer to a `Column` object (Section 2.3.1, "The `Column` Class").

Signature. Referencing the column by its attribute ID:

```
int read_attr
    (
      Uint32 attrId,
      Uint32 RegSource
    )
```

Referencing the column as a `Column` object:

```
int read_attr
    (
      const NdbDictionary::Column* column,
      Uint32 RegSource
    )
```

Parameters. This method takes two parameters:

- A reference to the table column to which the register value is to be copied. This can be either of the following:
  - The table column's attribute ID (*`attrId`*)
  - A pointer to a *`column`* — that is, a pointer to an `NdbDictionary::Column` object referencing the table column
- The register whose value is to be copied (*`RegSource`*).

Return Value. `0` on success; `-1` on failure.

2.3.14.1.4. `NdbInterpretedCode` Register Arithmetic Methods

This class provides two methods for performing arithmetic operations on registers. `add_reg()` allows you to load the sum of two registers into another register; `sub_reg()` lets you load the difference of two registers into another register.

The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`add_reg()`	1	1
`sub_reg()`	1	1

More information about these methods is presented in the next two sections.

2.3.14.1.4.1. `NdbInterpretedCode::add_reg()`

Description. This method sums the values stored in any two given registers and stores the result in a third register.

Signature.

```
int add_reg
    (
      Uint32 RegDest,
      Uint32 RegSource1,
      Uint32 RegSource2
    )
```

Parameters. This method takes three parameters. The first of these is the register in which the result is to be stored. The second and third parameters are the registers whose values are to be summed.

> **Note**
>
> It is possible to re-use for storing the result one of the registers whose values are summed; that is, *`RegDest`* can be the same as *`RegSource1`* or *`RegSource2`*.

Return Value. `0` on success; `-1` on failure.

2.3.14.1.4.2. `NdbInterpretedCode::sub_reg()`

Description. This method gets the difference between the values stored in any two given registers and stores the result in a third register.

Signature.

```
int sub_reg
    (
      Uint32 RegDest,
      Uint32 RegSource1,
      Uint32 RegSource2
    )
```

Parameters. This method takes three parameters. The first of these is the register in which the result is to be stored. The second and third parameters are the registers whose values are to be subtracted; that is, the value of register *`RegDest`* is set equal to

```
(value in register RegSource1) – (value in register RegSource2)
```

> **Note**
>
> It is possible to re-use one of the registers whose values are subtracted for storing the result; that is, *`RegDest`* can be the same as *`RegSource1`* or *`RegSource2`*.

Return Value. `0` on success; `-1` on failure.

2.3.14.1.5. `NdbInterpretedCode`: Labels and Branching

The `NdbInterpretedCode` class allows you to define labels within interpreted programs and provides a number of methods for performing jumps to these labels based on any of the following types of conditions:

- Comparison between two register values
- Comparison between a column value and a given constant
- Whether or not a column value matches a given pattern

To define a label, use the `def_label()` method. See Section 2.3.14.1.6, “`NdbInterpretedCode::def_label()`”.

To perform an unconditional jump to a label, use the `branch_label()` method. See Section 2.3.14.1.7, “`NdbInterpretedCode::branch_label()`”.

To perform a jump to a given label based on a comparison of register values, use one of the `branch_*()` methods (`branch_ge()`, `branch_gt()`, `branch_le()`, `branch_lt()`, `branch_eq()`, `branch_ne()`, `branch_ne_null()`, or `branch_eq_null()`). See Section 2.3.14.1.8, “Register-Based `NdbInterpretedCode` Branch Operations”.

To perform a jump to a given label based on a comparison of table column values, use one of the `branch_col_*()` methods (`branch_col_ge()`, `branch_col_gt()`, `branch_col_le()`, `branch_col_lt()`, `branch_col_eq()`, `branch_col_ne()`, `branch_col_ne_null()`, or `branch_col_eq_null()`). See Section 2.3.14.1.9, "Column-Based `NdbInterpretedCode` Branch Operations".

To perform a jump based on pattern-matching of a table column value, use one of the methods `branch_col_like()` or `branch_col_notlike()`. See Section 2.3.14.1.10, "Pattern-Based `NdbInterpretedCode` Branch Operations".

2.3.14.1.6. `NdbInterpretedCode::def_label()`

Description. This method defines a label to be used as the target of one or more jumps in an interpreted program.

`def_label()` uses a 2-word buffer and requires no space for request messages.

Signature.

```
int def_label
    (
      int LabelNum
    )
```

Parameters. This method takes a single parameter *`LabelNum`*, whose value must be unique among all values used for labels within the interpreted program.

Return Value. `0` on success; `-1` on failure.

2.3.14.1.7. `NdbInterpretedCode::branch_label()`

Description. This method performs an unconditional jump to an interpreted program label (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Signature.

```
int branch_label
    (
      Uint32 Label
    )
```

Parameters. This method takes a single parameter, an interpreted program *`Label`* defined using `def_label()`.

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8. Register-Based `NdbInterpretedCode` Branch Operations

Most of the methods discussed in this section are used to branch based on the results of register-to-register comparisons. There are also two methods used to compare a register value with `NULL`. All of these methods require as a parameter a label defined using the `def_label()` method (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

These methods can be thought of as performing the following logic:

```
if(register_value1 condition register_value2)
  goto Label
```

The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`branch_ge()`	1	1
`branch_gt()`	1	1
`branch_le()`	1	1
`branch_lt()`	1	1
`branch_eq()`	1	1
`branch_ne()`	1	1
`branch_ne_null()`	1	1
`branch_eq_null()`	1	1

2.3.14.1.8.1. `NdbInterpretedCode::branch_ge()`

Description. This method compares two register values; if the first is greater than or equal to the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_ge
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared — *`RegLvalue`* and *`RegRvalue`* — and the program *`Label`* to jump to if *`RegLvalue`* is greater than or equal to *`RegRvalue`*. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.2. `NdbInterpretedCode::branch_gt()`

Description. This method compares two register values; if the first is greater than the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_gt
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared — *`RegLvalue`* and *`RegRvalue`* — and the program *`Label`* to jump to if *`RegLvalue`* is greater than *`RegRvalue`*. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.3. `NdbInterpretedCode::branch_le()`

Description. This method compares two register values; if the first is less than or equal to the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_le
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared — *`RegLvalue`* and *`RegRvalue`* — and the program *`Label`* to jump to if *`RegLvalue`* is less than or equal to *`RegRvalue`*. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.4. `NdbInterpretedCode::branch_lt()`

Description. This method compares two register values; if the first is less than the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_lt
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared — *`RegLvalue`* and *`RegRvalue`* — and the program *`Label`* to jump to if *`RegLvalue`* is less than *`RegRvalue`*. *`Label`* must have been defined previ-

ously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.5. `NdbInterpretedCode::branch_eq()`

Description. This method compares two register values; if they equal, then the interpreted program jumps to the specified label.

Signature.

```
int branch_eq
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared — *`RegLvalue`* and *`RegRvalue`* — and the program *`Label`* to jump to if they are equal. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.6. `NdbInterpretedCode::branch_ne()`

Description. This method compares two register values; if they are not equal, then the interpreted program jumps to the specified label.

Signature.

```
int branch_ne
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared — *`RegLvalue`* and *`RegRvalue`* — and the program *`Label`* to jump they are not equal. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.7. `NdbInterpretedCode::branch_ne_null()`

Description. This method compares a register value with `NULL`; if the value is not null, then the interpreted program jumps to the specified label.

Signature.

```
int branch_ne_null
    (
      Uint32 RegLvalue,
      Uint32 Label
    )
```

Parameters. This method takes two parameters, the register whose value is to be compared with `NULL` (*`RegLvalue`*) and the program *`Label`* to jump to if *`RegLvalue`* is not null. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.8.8. `NdbInterpretedCode::branch_eq_null()`

Description. This method compares a register value with `NULL`; if the register value is null, then the interpreted program jumps to the specified label.

Signature.

```
int branch_eq_null
    (
      Uint32 RegLvalue,
      Uint32 Label
    )
```

Parameters. This method takes two parameters, the register whose value is to be compared with `NULL` (*`RegLvalue`*) and the program *`Label`* to jump to if *`RegLvalue`* is null. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.14.1.6, "`NdbInterpretedCode::def_label()`").

Return Value. `0` on success, `-1` on failure.

2.3.14.1.9. Column-Based `NdbInterpretedCode` Branch Operations

The methods described in this section are used to perform branching based on a comparison between a table column value and a given constant value. Each of these methods expects the attribute ID of the column whose value is to be tested rather than a reference to a `Column` object.

These methods, with the exception of `branch_col_eq_null()` and `branch_col_ne_null()`, can be thought of as performing the following logic:

```
if(constant_value condition column_value)
  goto Label
```

In each case (once again excepting `branch_col_eq_null()` and `branch_col_ne_null()`), the arbitrary constant is the first parameter passed to the method.

The space requirements for each of these methods is shown in the following table, where *`L`* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`branch_col_eq_null()`	2	2
`branch_col_ne_null()`	2	2
`branch_col_eq()`	2	2 + `CEIL(`*`L`*` / 8)`
`branch_col_ne()`	2	2 + `CEIL(`*`L`*` / 8)`
`branch_col_lt()`	2	2 + `CEIL(`*`L`*` / 8)`
`branch_col_le()`	2	2 + `CEIL(`*`L`*` / 8)`
`branch_col_gt()`	2	2 + `CEIL(`*`L`*` / 8)`
`branch_col_ge()`	2	2 + `CEIL(`*`L`*` / 8)`

> **Note**
>
> The expression `CEIL(`*`L`*` / 8)` is the number of whole 8-byte words required to hold the constant value to be compared.

2.3.14.1.9.1. `NdbInterpretedCode::branch_col_eq()`

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the values are equal.

Signature.

```
int branch_col_eq
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters:

- A constant value (*`val`*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *`val`*
- A *`Label`* (previously defined using `def_label()`) to jump to if the compared values are equal

Return Value. `0` on success, `-1` on failure.

2.3.14.1.9.2. NdbInterpretedCode::branch_col_ne()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the two values are not equal.

Signature.

```
int branch_col_ne
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the compared values are unequal

Return Value. 0 on success, -1 on failure.

2.3.14.1.9.3. NdbInterpretedCode::branch_col_lt()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is less than the column value.

Signature.

```
int branch_col_lt
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the constant value is less than the column value

Return Value. 0 on success, -1 on failure.

2.3.14.1.9.4. NdbInterpretedCode::branch_col_le()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is less than or equal to the column value.

Signature.

```
int branch_col_le
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the constant value is less than or equal to the column value

Return Value. 0 on success, -1 on failure.

2.3.14.1.9.5. NdbInterpretedCode::branch_col_gt()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is greater than the column value.

Signature.

```
int branch_col_gt
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the constant value is greater than the column value

Return Value. 0 on success, -1 on failure.

2.3.14.1.9.6. NdbInterpretedCode::branch_col_ge()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is greater than or equal to the column value.

Signature.

```
int branch_col_ge
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the constant value is greater than or equal to the column value

Return Value. 0 on success, -1 on failure.

2.3.14.1.9.7. NdbInterpretedCode::branch_col_eq_null()

Description. This method tests the value of a table column and jumps to the indicated program label if the column value is NULL.

Signature.

```
int branch_col_eq_null
    (
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method requires two parameters: the attribuite ID for the table column, and the program label to jump to if the column value is `NULL`.

Return Value. `0` on success, `-1` on failure.

2.3.14.1.9.8. `NdbInterpretedCode::branch_col_ne_null()`

Description. This method tests the value of a table column and jumps to the indicated program label if the column value is not `NULL`.

Signature.

```
int branch_col_ne_null
    (
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method requires two parameters: the attribuite ID of the table column, and the program label to jump to if the column value is not `NULL`.

Return Value. `0` on success, `-1` on failure.

2.3.14.1.10. Pattern-Based `NdbInterpretedCode` Branch Operations

The `NdbInterpretedCode` class provides two methods which can be used to branch based on a comparison between a column containing character data (that is, a `CHAR`, `VARCHAR`, `BINARY`, or `VARBINARY` column) and a regular expression pattern.

The pattern syntax allowed in the regular expression is the same as that supported by the MySQL Server's `LIKE` and `NOT LIKE` operators, including the `_` and `%` metacharacters. For more information about these, see String Comparison Functions.

> **Note**
>
> This is also the same regular expression pattern syntax supported by `NdbScanFilter`; see Section 2.3.17.2.6, “`NdbScanFilter::cmp()`”, for more information.

The table being operated upon must be supplied when the `NdbInterpretedCode` object is instantiated. The regular expression pattern should be in plain `CHAR` format, even if the column is actually a `VARCHAR` (in other words, there should be no leading length bytes).

These functions behave as shown here:

```
if (column_value [NOT] LIKE pattern)
  goto Label;
```

The space requirements for these methods are shown in the following table, where `L` represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`branch_col_like()`	2	2 + `CEIL(L / 8)`
`branch_col_notlike()`	2	2 + `CEIL(L / 8)`

> **Note**
>
> The expression `CEIL(L / 8)` is the number of whole 8-byte words required to hold the constant value to be compared.

2.3.14.1.10.1. `NdbInterpretedCode::branch_col_like()`

Description. This method tests a table column value against a regular expression pattern and jumps to the indicated program label if they match.

Signature.

```
int branch_col_like
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters, which are listed here:

- A regular expression pattern (*`val`*); see Section 2.3.14.1.10, “Pattern-Based `NdbInterpretedCode` Branch Operations”, for the syntax supported
- Length of the pattern (in bytes)
- The attribute ID for the table column being tested
- The program label to jump to if the table column value matches the pattern

Return Value. `0` on success, `-1` on failure

2.3.14.1.10.2. `NdbInterpretedCode::branch_col_notlike()`

Description. This method is similar to `branch_col_like()` in that it tests a table column value against a regular expression pattern; however it jumps to the indicated program label only if the pattern and the column value do *not* match.

Signature.

```
int branch_col_notlike
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes 4 parameters as shown here:

- A regular expression pattern (*`val`*); see Section 2.3.14.1.10, “Pattern-Based `NdbInterpretedCode` Branch Operations”, for the syntax supported
- Length of the pattern (in bytes)
- The attribute ID for the table column being tested
- The program label to jump to if the table column value does not match the pattern

Return Value. `0` on success, `-1` on failure

2.3.14.1.11. `NdbInterpretedCode` Bitwise Comparison Operations

These instructions are available beginning with MySQL Cluster NDB 6.3.20. They are used to branch based on the result of a logical `AND` comparison between a `BIT` column value and a bitmask pattern.

Use of these methods requires that the table being operated upon was supplied when the `NdbInterpretedCode` object was constructed. The mask value should be the same size as the bit column being compared. `BIT` values are passed into and out of the NDB API as 32-bit words with bits set in order from the least significant bit to the most significant bit. The endianness of the platform on which the instructions are executed controls which byte contains the least significant bits. On x86, this is the first byte (byte 0); on SPARC and PPC, it is the last byte.

The buffer length and request length for each of these methods each requires an amount of space equal to 2 words plus the column width rounded (up) to the nearest whole word.

2.3.14.1.11.1. `NdbInterpretedCode::branch_col_and_mask_eq_mask()`

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is equal to the bitmask, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_eq_mask
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method allows for 4 parameters, of which 3 are actually used:

- A pointer to a constant *`mask`* to compare the column value to
- A `Uint32` value which is currently *`unused`*.
- The *`attrId`* of the column to be compared.
- A program *`Label`* to jump to if the condition is true.

Return Value. This method returns `0` on success and `-1` on failure.

2.3.14.1.11.2. `NdbInterpretedCode::branch_col_and_mask_ne_mask()`

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is not equal to the bitmask, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_ne_mask
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method allows for 4 parameters, of which 3 are actually used:

- A pointer to a constant *`mask`* to compare the column value to.
- A `Uint32` value which is currently *`unused`*.
- The *`attrId`* of the column to be compared.
- A program *`Label`* to jump to if the condition is true.

Return Value. This method returns `0` on success and `-1` on failure.

2.3.14.1.11.3. `NdbInterpretedCode::branch_col_and_mask_eq_zero()`

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is equal to `0`, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_eq_zero
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method allows for 4 parameters, of which 3 are actually used:

- A pointer to a constant *`mask`* to compare the column value to.
- A `Uint32` value which is currently *`unused`*.

- The *attrId* of the column to be compared.
- A program *Label* to jump to if the condition is true.

Return Value. This method returns 0 on success and -1 on failure.

2.3.14.1.11.4. NdbInterpretedCode::branch_col_and_mask_ne_zero()

Description. This method is used to compare a BIT column value with a bitmask; if the column value ANDed together with the bitmask is not equal to 0, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_ne_zero
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method allows for 4 parameters, of which 3 are actually used:

- A pointer to a constant *mask* to compare the column value to.
- A Uint32 value which is currently *unused*.
- The *attrId* of the column to be compared.
- A program *Label* to jump to if the condition is true.

Return Value. This method returns 0 on success and -1 on failure.

2.3.14.1.12. NdbInterpretedCode Result Handling Methods

The methods described in this section are used to tell the interpreter that processing of the current row is complete, and — in the case of scans — whether or not to include this row in the results of the scan.

The space requirements for these methods are shown in the following table, where *L* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
interpret_exit_ok()	1	1
interpret_exit_nok()	1	1
interpret_exit_last_row()	1	1

2.3.14.1.12.1. NdbInterpretedCode::interpret_exit_ok()

Description. For a scanning operation, this method indicates that the current row should be returned as part of the results of the scan and that the program should move on to the next row. For other operations, calling this method causes the interpreted program to exit.

Signature.

```
int interpret_exit_ok
    (
      void
    )
```

Parameters. *None*.

Return Value. 0 on success, -1 on failure.

2.3.14.1.12.2. NdbInterpretedCode::interpret_exit_nok()

Description. For scanning operations, this method is used to indicate that the current row should not be returned as part of the scan, and to cause the program should move on to the next row. It causes other types of operations to be aborted.

Signature.

```
int interpret_exit_nok
    (
      Uint32 ErrorCode = 899
    )
```

Parameters. This method takes a single (optional) parameter *`ErrorCode`* which defaults to NDB error code `899` (`ROWID ALREADY ALLOCATED`). For a complete listing of NDB error codes, see Section 4.2.2, "NDB Error Codes and Messages".

Return Value. `0` on success, `-1` on failure.

2.3.14.1.12.3. `NdbInterpretedCode::interpret_last_row()`

Description. For a scanning operation, invoking this method indicates that this row should be returned as part of the scan, and that no more rows in this fragment should be scanned. For other types of operations, the method causes the operation to be aborted.

Signature.

```
int interpret_exit_last_row
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` if successful, `-1` otherwise.

2.3.14.1.13. `NdbInterpretedCode` Convenience Methods

The methods described in this section can be used to insert multiple instructions (using specific registers) into an interpreted program.

> **Important**
>
> In addition to updating the table column, these methods use interpreter registers 6 and 7, replacing any existing contents of register 6 with the original column value and any existing contents of register 7 with the modified column value. The table itself must be previously defined when instantiating the `NdbInterpretedCode` object for which the method is invoked.

The space requirements for these methods are shown in the following table, where *`L`* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`add_value()`	4	1; if the supplied value >= 2^{16}: 2; if >= 2^{32}: 3
`sub_value()`	4	1; if the supplied value >= 2^{16}: 2; if >= 2^{32}: 3

2.3.14.1.13.1. `NdbInterpretedCode::add_val()`

Description. This method adds a specified value to the value of a given table column, and places the original and modified column values in registers 6 and 7. It is equivalent to the following series of `NdbInterpretedCode` method calls, where *`attrId`* is the table column' attribute ID and *`aValue`* is the value to be added:

```
read_attr(6, attrId);
load_const_u32(7, aValue);
add_reg(7,6,7);
write_attr(attrId, 7);
```

`aValue` can be a 32-bit or 64-bit integer.

Signature.

```
int add_val
    (
      Uint32 attrId,
      Uint32 aValue
    )
```

or

```
int add_val
```

```
    (
      Uint32 attrId,
      Uint64 aValue
    )
```

Parameters. A table column attribute ID and a 32-bit or 64-bit integer value to be added to this column value.

Return Value. `0` on success, `-1` on failure.

2.3.14.1.13.2. `NdbInterpretedCode::sub_val()`

Description. This method subtracts a specified value from the value of a given table column, and places the original and modified column values in registers 6 and 7. It is equivalent to the following series of `NdbInterpretedCode` method calls, where *`attrId`* is the table column' attribute ID and *`aValue`* is the value to be subtracted:

```
read_attr(6, attrId);
load_const_u32(7, aValue);
sub_reg(7,6,7);
write_attr(attrId, 7);
```

`aValue` can be a 32-bit or 64-bit integer.

Signature.

```
int sub_val
    (
      Uint32 attrId,
      Uint32 aValue
    )
```

or

```
int sub_val
    (
      Uint32 attrId,
      Uint64 aValue
    )
```

Parameters. A table column attribute ID and a 32-bit or 64-bit integer value to be subtracted from this column value.

Return Value. `0` on success, `-1` on failure.

2.3.14.1.14. Using Subroutines with `NdbInterpretedCode`

`NdbInterpretedCode` supports subroutines which can be invoked from within interpreted programs, with each subroutine being identified by a unique number. Subroutines can be defined only following all main program instructions.

> **Important**
>
> Numbers used to identify subroutines must be contiguous; however, they do not have to be in any particular order.

The beginning of a subroutine is indicated by invoking the `def_sub()` method; all instructions after this belong to this subroutine until the subroutine is terminated with `ret_sub()`. A subroutine is called using `call_sub()`.

Once the subroutine has completed, the program resumes execution with the instruction immediately following the one which invoked the subroutine. Subroutines can also be invoked from other subroutines; currently, the maximum subroutine stack depth is 32.

2.3.14.1.14.1. `NdbInterpretedCode::def_sub()`

Description. This method is used to mark the start of a subroutine. See Section 2.3.14.1.14, “Using Subroutines with `NdbInterpretedCode`”, for more information.

Signature.

```
int def_sub
    (
      Uint32 SubroutineNumber
    )
```

Parameters. A single parameter, a number used to identify the subroutine.

Return Value. `0` on success, `-1` otherwise.

2.3.14.1.14.2. `NdbInterpretedCode::call_sub()`

Description. This method is used to call a subroutine.

Signature.

```
int call_sub
    (
      Uint32 SubroutineNumber
    )
```

Parameters. This method takes a single parameter, the number identifying the subroutine to be called.

Return Value. `0` on success, `-1` on failure.

2.3.14.1.14.3. `NdbInterpretedCode::ret_sub()`

Description. This method marks the end of the current subroutine.

Signature.

```
int ret_sub
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` otherwise.

2.3.14.1.15. `NdbInterpretedCode` Utility Methods

This section provides information about some utility methods supplied by `NdbInterepretedCode`.

2.3.14.1.15.1. `NdbInterpretedCode::finalise()`

Description. This method prepares an interpreted program, including any subroutines it might have, by resolving all branching instructions and calls to subroutines. It must be called before using the program, and can be invoked only once for any given `NdbInterpretedCode` object.

If no instructions have been defined, this method attempts to insert a single `interpret_exit_ok()` method call prior to finalization.

Signature.

```
int finalise
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` otherwise.

2.3.14.1.15.2. `NdbInterpretedCode::getTable()`

Description. This method can be used to obtain a reference to the table for which the `NdbInterpretedCode` object was defined.

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to a `Table` object. Returns `NULL` if no table object was supplied when the `NdbInterpretedCode` was instantiated.

2.3.14.1.15.3. `NdbInterpretedCode::getNdbError()`

Description. This method returns the most recent error associated with this `NdbInterpretedCode` object.

Signature.

```
const class NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an `NdbError` object. See Section 2.3.30, “The `NdbError` Structure”, for more information.

2.3.14.1.15.4. `NdbInterpretedCode::getWordsUsed()`

Description. This method returns the number of words from the buffer that have been used, whether the buffer is one that is user-supplied or the internally-provided buffer.

Signature.

```
Uint32 getWordsUsed
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of words used from the buffer.

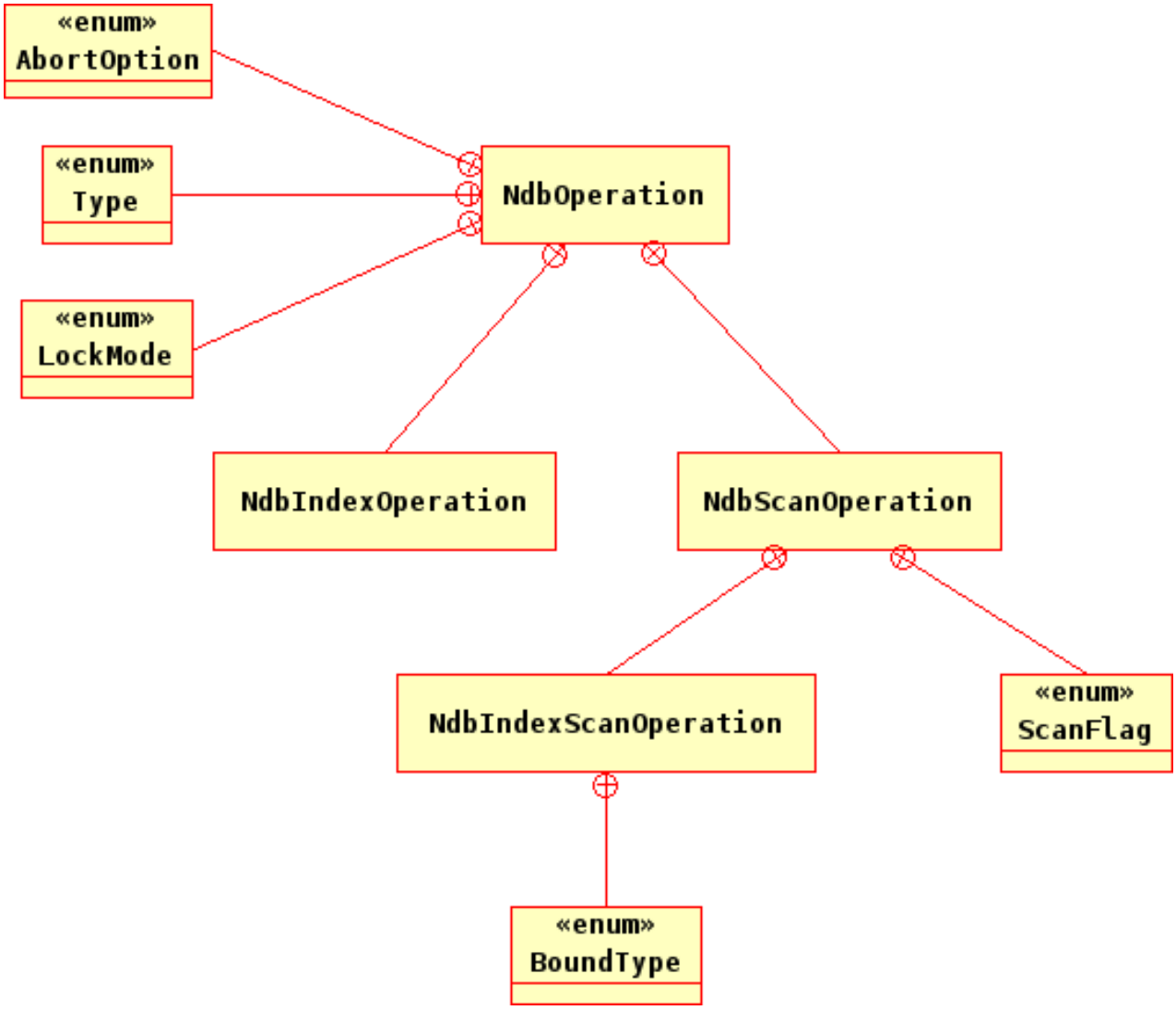
2.3.15. The `NdbOperation` Class

This section discusses the `NdbOperation` class.

Parent class. *None*

Child classes. NdbIndexOperation, NdbScanOperation

`NdbOperation` Subclasses. This diagram shows the relationships of `NdbOperation`, it subclasses, and their public types:



Description. `NdbOperation` represents a "generic" data operation. Its subclasses represent more specific types of operations. See Section 2.3.15.1.2, "The `NdbOperation::Type` Type" for a listing of operation types and their corresponding `NdbOperation` subclasses.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getValue()`	Allocates an attribute value holder for later access
`getBlobHandle()`	Used to access blob attributes
`getTableName()`	Gets the name of the table used for this operation
`getTable()`	Gets the table object used for this operation
`getNdbError()`	Gets the latest error
`getNdbErrorLine()`	Gets the number of the method where the latest error occurred
`getType()`	Gets the type of operation
`getLockMode()`	Gets the operation's lock mode
`getNdbTransaction()`	Gets the `NdbTransaction` object for this operation
`equal()`	Defines a search condition using equality
`setValue()`	Defines an attribute to set or update
`insertTuple()`	Adds a new tuple to a table
`updateTuple()`	Updates an existing tuple in a table
`readTuple()`	Reads a tuple from a table

Method	Purpose / Use
`writeTuple()`	Inserts or updates a tuple
`deleteTuple()`	Removes a tuple from a table

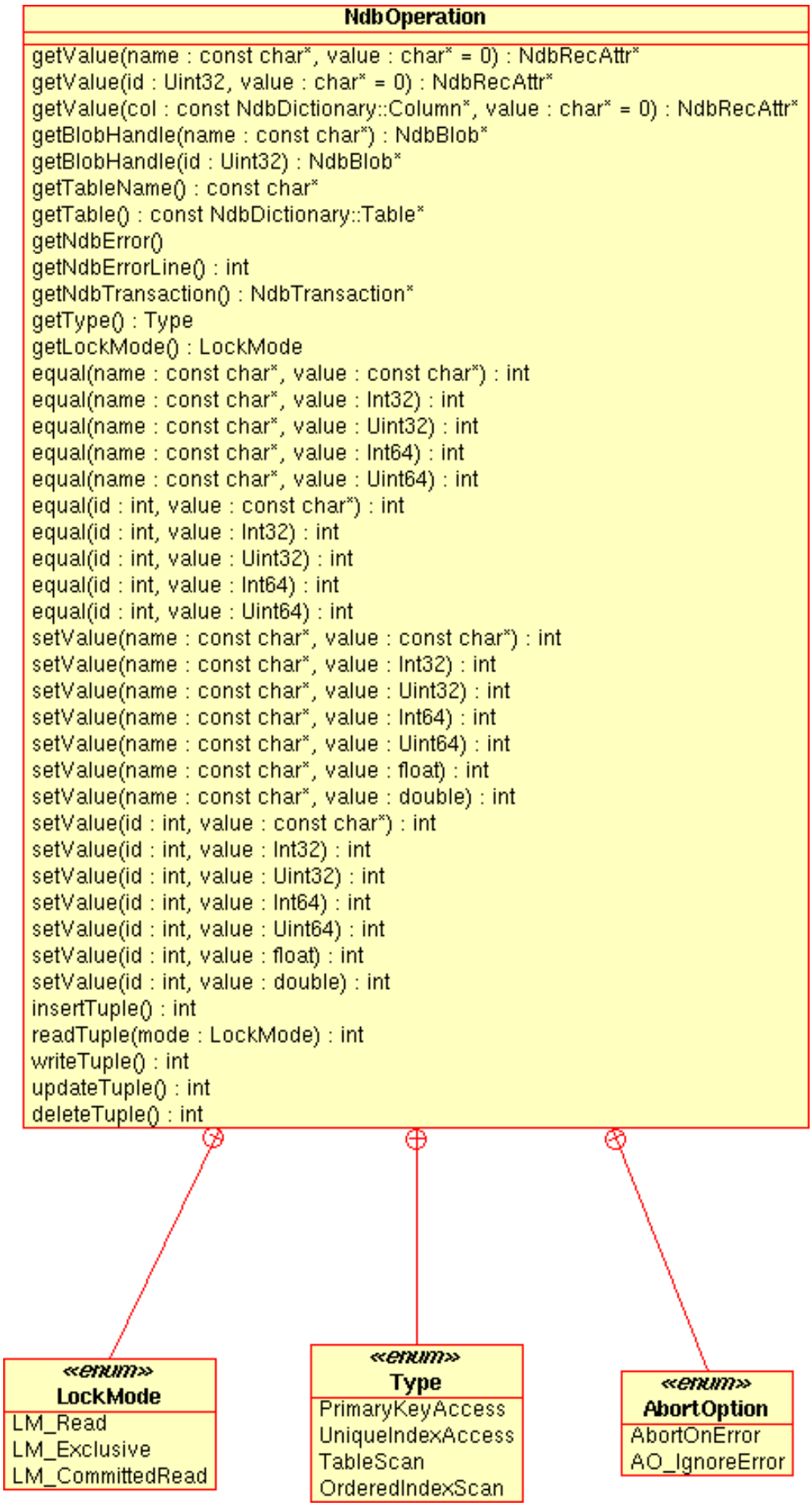
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.15.2, "`NdbOperation` Methods".

Types. The `NdbOperation` class defines two public types, as shown in the following table:

Type	Purpose / Use
`Type`	Operation access types
`LockMode`	The type of lock used when performing a read operation

For a discussion of each of these types, along with its possible values, see Section 2.3.15.1, "`NdbOperation` Types".

Class diagram. This diagram shows all the available methods and enumerated types of the `NdbOperation` class:


NdbOperation
getValue(name : const char*, value : char* = 0) : NdbRecAttr*
getValue(id : Uint32, value : char* = 0) : NdbRecAttr*
getValue(col : const NdbDictionary::Column*, value : char* = 0) : NdbRecAttr*
getBlobHandle(name : const char*) : NdbBlob*
getBlobHandle(id : Uint32) : NdbBlob*
getTableName() : const char*
getTable() : const NdbDictionary::Table*
getNdbError()
getNdbErrorLine() : int
getNdbTransaction() : NdbTransaction*
getType() : Type
getLockMode() : LockMode
equal(name : const char*, value : const char*) : int
equal(name : const char*, value : Int32) : int
equal(name : const char*, value : Uint32) : int
equal(name : const char*, value : Int64) : int
equal(name : const char*, value : Uint64) : int
equal(id : int, value : const char*) : int
equal(id : int, value : Int32) : int
equal(id : int, value : Uint32) : int
equal(id : int, value : Int64) : int
equal(id : int, value : Uint64) : int
setValue(name : const char*, value : const char*) : int
setValue(name : const char*, value : Int32) : int
setValue(name : const char*, value : Uint32) : int
setValue(name : const char*, value : Int64) : int
setValue(name : const char*, value : Uint64) : int
setValue(name : const char*, value : float) : int
setValue(name : const char*, value : double) : int
setValue(id : int, value : const char*) : int
setValue(id : int, value : Int32) : int
setValue(id : int, value : Uint32) : int
setValue(id : int, value : Int64) : int
setValue(id : int, value : Uint64) : int
setValue(id : int, value : float) : int
setValue(id : int, value : double) : int
insertTuple() : int
readTuple(mode : LockMode) : int
writeTuple() : int
updateTuple() : int
deleteTuple() : int
«enum»
LockMode
LM_Read
LM_Exclusive
LM_CommittedRead
«enum»
Type
PrimaryKeyAccess
UniqueIndexAccess
TableScan
OrderedIndexScan
«enum»
AbortOption
AbortOnError
AO_IgnoreError

> **Note**
>
> For more information about the use of `NdbOperation`, see Section 1.3.2.3.1, "Single-row operations".

2.3.15.1. `NdbOperation` Types

This section details the public types belonging to the `NdbOperation` class.

2.3.15.1.1. The `NdbOperation::AbortOption` Type

Description. This type is used to determine whether failed operations should force a transaction to be aborted. It is used as an argument to the `execute()` method — see Section 2.3.19.2.5, "`NdbTransaction::execute()`", for more information.

Enumeration values.

Value	Description
`AbortOnError`	A failed operation causes the transaction to abort.
`AO_IgnoreOnError`	Failed operations are ignored; the transaction continues to execute.
`DefaultAbortOption`	The `AbortOption` value is set according to the operation type: • Read operations: `AO_IgnoreOnError` • Scan takeover or DML operations: `AbortOnError`

`DefaultAbortOperation` is available beginning with MySQL Cluster NDB 6.2.0. See Section 2.3.19.2.5, "`NdbTransaction::execute()`", for more information.

> **Important**
>
> Previous to MySQL Cluster NDB 6.2.0, this type belonged to the `NdbTransaction` class.

2.3.15.1.2. The `NdbOperation::Type` Type

Description. `Type` is used to describe the operation access type. Each access type is supported by `NdbOperation` or one of its subclasses, as shown in the following table:

Enumeration values.

Value	Description	Class
`PrimaryKeyAccess`	A read, insert, update, or delete operation using the table's primary key	`NdbOperation`
`UniqueIndexAccess`	A read, update, or delete operation using a unique index	`NdbIndexOperation`
`TableScan`	A full table scan	`NdbScanOperation`
`OrderedIndexScan`	An ordered index scan	`NdbIndexScanOperation`

2.3.15.1.3. The `NdbOperation::LockMode` Type

Description. This type describes the lock mode used when performing a read operation.

Enumeration values.

Value	Description
`LM_Read`	Read with shared lock
`LM_Exclusive`	Read with exclusive lock
`LM_CommittedRead`	Ignore locks; read last committed

> **Note**
>
> There is also support for dirty reads (`LM_Dirty`), but this is normally for internal purposes only, and should not be

- used for applications deployed in a production setting.

2.3.15.2. `NdbOperation` Methods

This section lists and describes the public methods of the `NdbOperation` class.

> **Note**
>
> This class has no public constructor. To create an instance of `NdbOperation`, you must use `NdbTransaction::getNdbOperation()`. See Section 2.3.19.2.1, "`NdbTransaction::getNdbOperation()`", for more information.

2.3.15.2.1. `NdbOperation::getValue()`

Description. This method prepares for the retrieval of an attribute value. The NDB API allocates memory for an `NdbRecAttr` object that is later used to hold the returned attribute value.

> **Important**
>
> This method does *not* fetch the attribute value from the database, and the `NdbRecAttr` object returned by this method is not readable or printable before calling `NdbTransaction::execute()`.
>
> If a specific attribute has not changed, the corresponding `NdbRecAttr` will be in the state `UNDEFINED`. This can be checked by using `NdbRecAttr::isNULL()` which in such cases returns `-1`.
>
> See Section 2.3.19.2.5, "`NdbTransaction::execute()`", and Section 2.3.16.1.4, "`NdbRecAttr::isNULL()`".

Signature. There are three versions of this method, each having different parameters:

```
NdbRecAttr* getValue
    (
      const char* name,
      char*       value = 0
    )

NdbRecAttr* getValue
    (
      Uint32 id,
      char*  value = 0
    )

NdbRecAttr* getValue
    (
      const NdbDictionary::Column* col,
      char*                        value = 0
    )
```

Parameters. All three forms of this method have two parameters, the second parameter being optional (defaults to `0`). They differ with regard to the type of the first parameter, which can be any one of the following:

- The attribute *`name`*
- The attribute *`id`*
- The *`column`* on which the attribute is defined

In all three cases, the second parameter is a character buffer in which a non-`NULL` attribute value is returned. In the event that the attribute is `NULL`, is it stored only in the `NdbRecAttr` object returned by this method.

Return Value. An `NdbRecAttr` object to hold the value of the attribute, or a `NULL` pointer, indicating an error.

Retrieving integers. Integer values can be retrieved from both the *`value`* buffer passed as this method's second parameter, and from the `NdbRecAttr` object itself. On the other hand, character data is available from `NdbRecAttr` (see Section 2.3.16.1.19, "`NdbRecAttr::aRef()`") only if no buffer was passed in to `getValue()`. However, character data is written to the buffer only if one is provided, in which case it cannot be retrieved from the `NdbRecAttr` object that was returned. In the latter case, `NdbRecAttr::aRef()` returns a buffer pointing to an empty string.

Accessing bit values. The following example shows how to check a given bit from the *`value`* buffer. Here, `op` is an operation (`NdbOperation` object), `name` is the name of the column from which to get the bit value, and `trans` is an `NdbTransaction` object.

```
Uint32 buf[];

op->getValue(name, buf); /* bit column */
```

```
trans->execute();

if(buf[X/32] & 1 << (X & 31)) /* check bit X */
{
  /* bit X set */
}
```

2.3.15.2.2. NdbOperation::getBlobHandle()

Description. This method is used in place of `getValue()` or `setValue()` for blob attributes. It creates a blob handle (`NdbBlob` object). A second call with the same argument returns the previously created handle. The handle is linked to the operation and is maintained automatically. See Section 2.3.9, "The `NdbBlob` Class", for details.

Signature. This method has two forms, depending on whether it is called with the name or the ID of the blob attribute:

```
virtual NdbBlob* getBlobHandle
    (
      const char* name
    )

virtual NdbBlob* getBlobHandle
    (
      Uint32 id
    )
```

Parameters. This method takes a single parameter, which can be either one of the following:

- The *`name`* of the attribute
- The *`id`* of the attribute

Return Value. Regardless of parameter type used, this method return a pointer to an instance of `NdbBlob`.

2.3.15.2.3. NdbOperation::getTableName()

Description. This method retrieves the name of the table used for the operation.

Signature.

```
const char* getTableName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table.

2.3.15.2.4. NdbOperation::getTable()

Description. This method is used to retrieve the table object associated with the operation.

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An instance of `Table`. For more information, see Section 2.3.21, "The `Table` Class".

2.3.15.2.5. NdbOperation::getNdbError()

Description. This method gets the most recent error (an `NdbError` object).

Signature.

```
const NdbError& getNdbError
    (
      void
```

```
    ) const
```

Parameters. *None*.

Return Value. An `NdbError` object. See Section 2.3.30, “The `NdbError` Structure”.

2.3.15.2.6. `NdbOperation::getNdbErrorLine()`

Description. This method retrieves the method number in which the latest error occurred.

Signature.

```
int getNdbErrorLine
    (
      void
    )
```

Beginning with MySQL Cluster NDB 6.2.17 and MySQL Cluster NDB 6.3.19, this method can also be used as shown here:

```
int getNdbErrorLine
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The method number (an integer).

2.3.15.2.7. `NdbOperation::getType()`

Description. This method is used to retrieve the access type for this operation.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `Type` value. See Section 2.3.15.1.2, “The `NdbOperation::Type` Type”.

2.3.15.2.8. `NdbOperation::getLockMode()`

Description. This method gets the operation's lock mode.

Signature.

```
LockMode getLockMode
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `LockMode` value. See Section 2.3.15.1.3, “The `NdbOperation::LockMode` Type”.

2.3.15.2.9. `NdbOperation::getNdbTransaction()`

Description. Gets the `NdbTransaction` object for this operation. Available beginning with MySQL Cluster NDB 6.2.17 and MySQL Cluster NDB 6.3.19.

Signature.

```
virtual NdbTransaction* getNdbTransaction
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to an `NdbTransaction` object. See Section 2.3.19, “The `NdbTransaction` Class”.

2.3.15.2.10. `NdbOperation::equal()`

Description. This method defines a search condition with an equality. The condition is true if the attribute has the given value. To set search conditions on multiple attributes, use several calls to `equal()`; in such cases all of them must be satisfied for the tuple to be selected.

> **Important**
>
> If the attribute is of a fixed size, its value must include all bytes. In particular a `Char` value must be native-blank padded. If the attribute is of variable size, its value must start with 1 or 2 little-endian length bytes (2 if its type is `Long*`).

> **Note**
>
> When using `insertTuple()`, you may also define the search key with `setValue()`. See Section 2.3.15.2.11, "`NdbOperation::setValue()`".

Signature. There are 10 versions of `equal()`, each having slightly different parameters. All of these are listed here:

```
int  equal
    (
      const char* name,
      const char* value
    )

int  equal
    (
      const char* name,
      Int32       value
    )

int  equal
    (
      const char* name,
      Uint32      value
    )

int  equal
    (
      const char* name,
      Int64       value
    )

int  equal
    (
      const char* name,
      Uint64      value
    )

int  equal
    (
      Uint32      id,
      const char* value
    )

int  equal
    (
      Uint32 id,
      Int32  value
    )

int  equal
    (
      Uint32 id,
      Uint32 value
    )

int  equal
    (
      Uint32 id,
      Int64  value
    )

int  equal
    (
      Uint32 id,
      Uint64 value
    )
```

Parameters. This method requires two parameters:

- The first parameter can be either of the following:
  - The *`name`* of the attribute (a string)

  - The *id* of the attribute (an unsigned 32-bit integer)
- The second parameter is the attribute *value* to be tested; it can be any one of the following 5 types:
  - String
  - 32-bit integer
  - Unsigned 32-bit integer
  - 64-bit integer
  - Unsigned 64-bit integer

Return Value. Returns `-1` in the event of an error.

2.3.15.2.11. `NdbOperation::setValue()`

Description. This method defines an attribute to be set or updated.

> **Important**
>
> There are a number of `NdbOperation::setValue()` methods that take a certain type as input (pass by value rather than passing a pointer). It is the responsibility of the application programmer to use the correct types.
>
> However, the NDB API does check that the application sends a correct length to the interface as given in the length parameter. A `char*` value can contain any datatype or any type of array. If the length is not provided, or if it is set to zero, then the API assumes that the pointer is correct, and does not check it.

> **Tip**
>
> To set a `NULL` value, use the following construct:
>
> ```
> setValue("ATTR_NAME", (char*)NULL);
> ```

> **Note**
>
> When you use `insertTuple()`, the NDB API will automatically detect that it is supposed to use `equal()` instead.
>
> In addition, it is not necessary when using insertTuple() to use `setValue()` on key attributes before other attributes.

Signature. There are 14 versions of `NdbOperation::setValue()`, each with slightly different parameters, as listed here (and summarised in the *Parameters* section following):

```
int setValue
    (
      const char* name,
      const char* value
)

int setValue
    (
      const char* name,
      Int32       value
    )

int setValue
    (
      const char* name,
      Uint32      value
    )

int setValue
    (
      const char* name,
      Int64       value
    )

int setValue
    (
      const char* name,
      Uint64      value
    )

int setValue
    (
```

```
      const char* name,
      float       value
    )

int setValue
    (
      const char* name,
      double      value
    )

int setValue
    (
      Uint32      id,
      const char* value
    )

int setValue
    (
      Uint32 id,
      Int32  value
    )

int setValue
    (
      Uint32 id,
      Uint32 value
    )

int setValue
    (
      Uint32 id,
      Int64  value
    )

int setValue
    (
      Uint32 id,
      Uint64 value
    )

int setValue
    (
      Uint32 id,
      float  value
    )

int setValue
    (
      Uint32 id,
      double value
    )
```

Parameters. This method requires two parameters:

- The first parameter identified the attribute to be set, and may be either one of:
  - The attribute *`name`* (a string)
  - The attribute *`id`* (an unsigned 32-bit integer)
- The second parameter is the *`value`* to which the attribute is to be set; its type may be any one of the following 7 types:
  - String (`const char*`)
  - 32-bit integer
  - Unsigned 32-bit integer
  - 64-bit integer
  - Unsigned 64-bit integer
  - Double
  - Float

See Section 2.3.15.2.10, “`NdbOperation::equal()`”, for important information regarding the value's format and length.

Return Value. Returns `-1` in the event of failure.

2.3.15.2.12. `NdbOperation::insertTuple()`

Description. This method defines the `NdbOperation` to be an `INSERT` operation. When the `NdbTransaction::execute()` method is called, this operation adds a new tuple to the table. See Section 2.3.19.2.5, “`NdbTransac-`

tion::execute()”.

Signature.

```
virtual int insertTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.15.2.13. NdbOperation::readTuple()

Description. This method define the `NdbOperation` as a `READ` operation. When the `NdbTransaction::execute()` method is invoked, the operation reads a tuple. See Section 2.3.19.2.5, “`NdbTransaction::execute()`”.

Signature.

```
virtual int readTuple
    (
      LockMode mode
    )
```

Parameters. *`mode`* specifies the locking mode used by the read operation. See Section 2.3.15.1.3, “The `NdbOperation::LockMode` Type”, for possible values.

Return Value. `0` on success, `-1` on failure.

2.3.15.2.14. NdbOperation::writeTuple()

Description. This method defines the `NdbOperation` as a `WRITE` operation. When the `NdbTransaction::execute()` method is invoked, the operation writes a tuple to the table. If the tuple already exists, it is updated; otherwise an insert takes place. See Section 2.3.19.2.5, “`NdbTransaction::execute()`”.

Signature.

```
virtual int writeTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.15.2.15. NdbOperation::updateTuple()

Description. This method defines the `NdbOperation` as an `UPDATE` operation. When the `NdbTransaction::execute()` method is invoked, the operation updates a tuple found in the table. See Section 2.3.19.2.5, “`NdbTransaction::execute()`”.

Signature.

```
virtual int updateTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.15.2.16. NdbOperation::deleteTuple()

Description. This method defines the `NdbOperation` as a `DELETE` operation. When the `NdbTransaction::execute()` method is invoked, the operation deletes a tuple from the table. See Section 2.3.19.2.5, “`NdbTransaction::execute()`”.

Signature.

```
virtual int deleteTuple
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.16. The `NdbRecAttr` Class

The section describes the `NdbRecAttr` class and its public methods.

Parent class. *None*

Child classes. *None*

Description. `NdbRecAttr` contains the value of an attribute. An `NdbRecAttr` object is used to store an attribute value after it has been retrieved the `NDB` Cluster using the `NdbOperation::getValue()`. This object is allocated by the NDB API. A brief example is shown here:

```
MyRecAttr = MyOperation->getValue("ATTR2", NULL);

if(MyRecAttr == NULL)
  goto error;

if(MyTransaction->execute(Commit) == -1)
  goto error;

ndbout << MyRecAttr->u_32_value();
```

For more examples, see Section 2.4.1, “Using Synchronous Transactions”.

> **Note**
>
> An `NdbRecAttr` object is instantiated with its value when `NdbTransaction::execute()` is invoked. Prior to this, the value is undefined. (Use `NdbRecAttr::isNULL()` to check whether the value is defined.) This means that an `NdbRecAttr` object has valid information only between the times that `NdbTransaction::execute()` and `Ndb::closeTransaction()` are called. The value of the null indicator is `-1` until `NdbTransaction::execute()` method is invoked.

Methods. `NdbRecAttr` has a number of methods for retrieving values of various simple types directly from an instance of this class. To obtain a reference to the value, use `NdbRecAttr::aRef()`. The following table lists all of the public methods of this class and the purpose or use of each method:

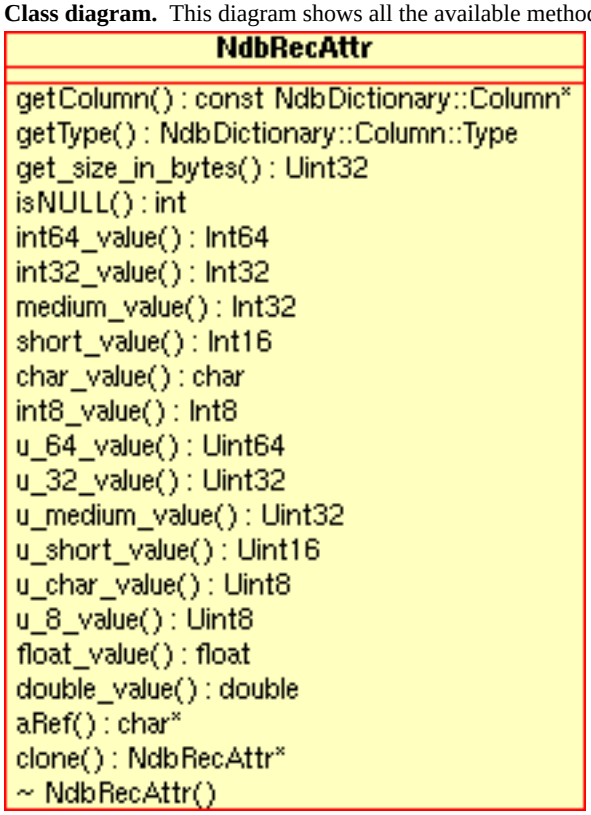
Method	Purpose / Use
`getColumn()`	Gets the column to which the attribute belongs
`getType()`	Gets the attribute's type (`Column::Type`)
`get_size_in_bytes()`	Gets the size of the attribute, in bytes
`isNULL()`	Tests whether the attribute is `NULL`
`int64_value()`	Retrieves a `Bigint` attribute value, as a 64-bit integer
`int32_value()`	Retrieves an `Int` attribute value, as a 32-bit integer
`medium_value()`	Retrieves a `Mediumint` attribute value, as a 32-bit integer
`short_value()`	Retrieves a `Smallint` attribute value, as a 16-bit integer
`char_value()`	Retrieves a `Char` attribute value
`int8_value()`	Retrieves a `Tinyint` attribute value, as an 8-bit integer
`u_64_value()`	Retrieves a `Bigunsigned` attribute value, as an unsigned 64-bit integer
`u_32_value()`	Retrieves an `Unsigned` attribute value, as an unsigned 32-bit integer
`u_medium_value()`	Retrieves a `Mediumunsigned` attribute value, as an unsigned 32-bit integer
`u_short_value()`	Retrieves a `Smallunsigned` attribute value, as an unsigned 16-bit integer
`u_char_value()`	Retrieves a `Char` attribute value, as an unsigned `char`
`u_8_value()`	Retrieves a `Tinyunsigned` attribute value, as an unsigned 8-bit integer
`float_value()`	Retrieves a `Float` attribute value, as a float (4 bytes)
`double_value()`	Retrieves a `Double` attribute value, as a double (8 bytes)
`aRef()`	Gets a pointer to the attribute value
`clone()`	Makes a deep copy of the `RecAttr` object

Method	Purpose / Use
~NdbRecAttr()	Destructor method

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.16.1, "NdbRecAttr Methods".

Types. The NdbRecAttr class defines no public types.

Class diagram. This diagram shows all the available methods of the NdbRecAttr class:



2.3.16.1. NdbRecAttr Methods

This section lists and describes the public methods of the NdbRecAttr class.

Constructor and Destructor. The NdbRecAttr class has no public constructor; an instance of this object is created using Ndb-Transaction::execute(). The destructor method, which is public, is discussed in Section 2.3.16.1.21, "~NdbRecAttr()".

2.3.16.1.1. NdbRecAttr::getColumn()

Description. This method is used to obtain the column to which the attribute belongs.

Signature.

```
const NdbDictionary::Column* getColumn
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to a Column object. See Section 2.3.1, "The Column Class".

2.3.16.1.2. NdbRecAttr::getType()

Description. This method is used to obtain the column's datatype.

Signature.

```
NdbDictionary::Column::Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `NdbDictionary::Column::Type` value. See Section 2.3.1.1.3, "`Column::Type`" for more information, including permitted values.

2.3.16.1.3. `NdbRecAttr::get_size_in_bytes()`

Description. You can use this method to obtain the size of an attribute (element).

Signature.

```
Uint32 get_size_in_bytes
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The attribute size in bytes, as an unsigned 32-bit integer.

2.3.16.1.4. `NdbRecAttr::isNULL()`

Description. This method checks whether an attribute value is `NULL`.

Signature.

```
int isNULL
    (
      void
    ) const
```

Parameters. *None*.

Return Value. One of the following 3 values:

- `-1`: The attribute value is not defined due to an error.
- `0`: The attribute value is defined, but is not `NULL`.
- `1`: The attribute value is defined and is `NULL`.

> **Important**
>
> In the event that `NdbTransaction::execute()` has not yet been called, the value returned by `isNull()` is not determined.

2.3.16.1.5. `NdbRecAttr::int64_value()`

Description. This method gets a `Bigint` value stored in an `NdbRecAttr` object, and returns it as a 64-bit signed integer.

Signature.

```
Int64 int64_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 64-bit signed integer.

2.3.16.1.6. `NdbRecAttr::int32_value()`

Description. This method gets an `Int` value stored in an `NdbRecAttr` object, and returns it as a 32-bit signed integer.

Signature.

```
Int32 int32_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit signed integer.

2.3.16.1.7. NdbRecAttr::medium_value()

Description. Gets the value of a `Mediumint` value stored in an `NdbRecAttr` object, and returns it as a 32-bit signed integer.

Signature.

```
Int32 medium_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit signed integer.

2.3.16.1.8. NdbRecAttr::short_value()

Description. This method gets a `Smallint` value stored in an `NdbRecAttr` object, and returns it as a 16-bit signed integer (short).

Signature.

```
short short_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 16-bit signed integer.

2.3.16.1.9. NdbRecAttr::char_value()

Description. This method gets a `Char` value stored in an `NdbRecAttr` object, and returns it as a `char`.

Signature.

```
char char_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `char` value.

2.3.16.1.10. NdbRecAttr::int8_value()

Description. This method gets a `Small` value stored in an `NdbRecAttr` object, and returns it as an 8-bit signed integer.

Signature.

```
Int8 int8_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An 8-bit signed integer.

2.3.16.1.11. NdbRecAttr::u_64_value()

Description. This method gets a `Bigunsigned` value stored in an `NdbRecAttr` object, and returns it as a 64-bit unsigned integer.

Signature.

```
Uint64 u_64_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 64-bit unsigned integer.

2.3.16.1.12. NdbRecAttr::u_32_value()

Description. This method gets an `Unsigned` value stored in an `NdbRecAttr` object, and returns it as a 32-bit unsigned integer.

Signature.

```
Uint32 u_32_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit unsigned integer.

2.3.16.1.13. NdbRecAttr::u_medium_value()

Description. This method gets an `Mediumunsigned` value stored in an `NdbRecAttr` object, and returns it as a 32-bit unsigned integer.

Signature.

```
Uint32 u_medium_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit unsigned integer.

Example.

```
[To be supplied...]
```

2.3.16.1.14. NdbRecAttr::u_short_value()

Description. This method gets a `Smallunsigned` value stored in an `NdbRecAttr` object, and returns it as a 16-bit (short) unsigned integer.

Signature.

```
Uint16 u_short_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A short (16-bit) unsigned integer.

2.3.16.1.15. NdbRecAttr::u_char_value()

Description. This method gets a `Char` value stored in an `NdbRecAttr` object, and returns it as an unsigned `char`.

Signature.

```
Uint8 u_char_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An 8-bit unsigned `char` value.

2.3.16.1.16. `NdbRecAttr::u_8_value()`

Description. This method gets a `Smallunsigned` value stored in an `NdbRecAttr` object, and returns it as an 8-bit unsigned integer.

Signature.

```
Uint8 u_8_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An 8-bit unsigned integer.

2.3.16.1.17. `NdbRecAttr::float_value()`

Description. This method gets a `Float` value stored in an `NdbRecAttr` object, and returns it as a float.

Signature.

```
float float_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A float (4 bytes).

2.3.16.1.18. `NdbRecAttr::double_value()`

Description. This method gets a `Double` value stored in an `NdbRecAttr` object, and returns it as a double.

Signature.

```
double double_value
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A double (8 bytes).

2.3.16.1.19. `NdbRecAttr::aRef()`

Description. This method is used to obtain a reference to an attribute value, as a `char` pointer. This pointer is aligned appropriately for the datatype. The memory is released by the NDB API when `NdbTransaction::closeTransaction()` is executed on the transaction which read the value.

Signature.

```
char* aRef
    (
      void
    ) const
```

Parameters. A pointer to the attribute value. Because this pointer is constant, this method can be called anytime after `NdbOperation::getValue()` has been called.

Return Value. *None*.

2.3.16.1.20. `NdbRecAttr::clone()`

Description. This method creates a deep copy of an `NdbRecAttr` object.

> **Note**
>
> The copy created by this method should be deleted by the application when no longer needed.

Signature.

```
NdbRecAttr* clone
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `NdbRecAttr` object. This is a complete copy of the original, including all data.

2.3.16.1.21. `~NdbRecAttr()`

Description. The `NdbRecAttr` class destructor method.

> **Important**
>
> You should delete only copies of `NdbRecAttr` objects that were created in your application using the `clone()` method. See Section 2.3.16.1.20, "`NdbRecAttr::clone()`".

Signature.

```
~NdbRecAttr
    (
      void
    )
```

Parameters. *None*.

Return Value. *None*.

2.3.17. The `NdbScanFilter` Class

This section discusses the `NdbScanFilter` class and its public members.

Parent class. *None*

Child classes. *None*

Description. `NdbScanFilter` provides an alternative means of specifying filters for scan operations.

> **Important**
>
> Prior to MySQL 5.1.14, the comparison methods of this class did not work with `BIT` values (see Bug#24503).
>
> Development of this interface continues in MySQL 5.1, and the charcateristics of the NdbScanFilter class are likely to change further in future releases.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`NdbScanFilter()`	Constructor method
`~NdbScanFilter()`	Destructor method
`begin()`	Begins a compound (set of conditions)
`end()`	Ends a compound
`istrue()`	Defines a term in a compound as `TRUE`
`isfalse()`	Defines a term in a compound as `FALSE`
`cmp()`	Compares a column value with an arbitrary value

Method	Purpose / Use
`eq()`	Tests for equality
`ne()`	Tests for inequality
`lt()`	Tests for a less-than condition
`le()`	Tests for a less-than-or-equal condition
`gt()`	Tests for a greater-than condition
`ge()`	Tests for a greater-than-or-equal condition
`isnull()`	Tests whether a column value is `NULL`
`isnotnull()`	Tests whether a column value is not `NULL`
`getNdbError()`	Provides access to error information
`getNdbOperation()`	Gets the associated `NdbOperation`

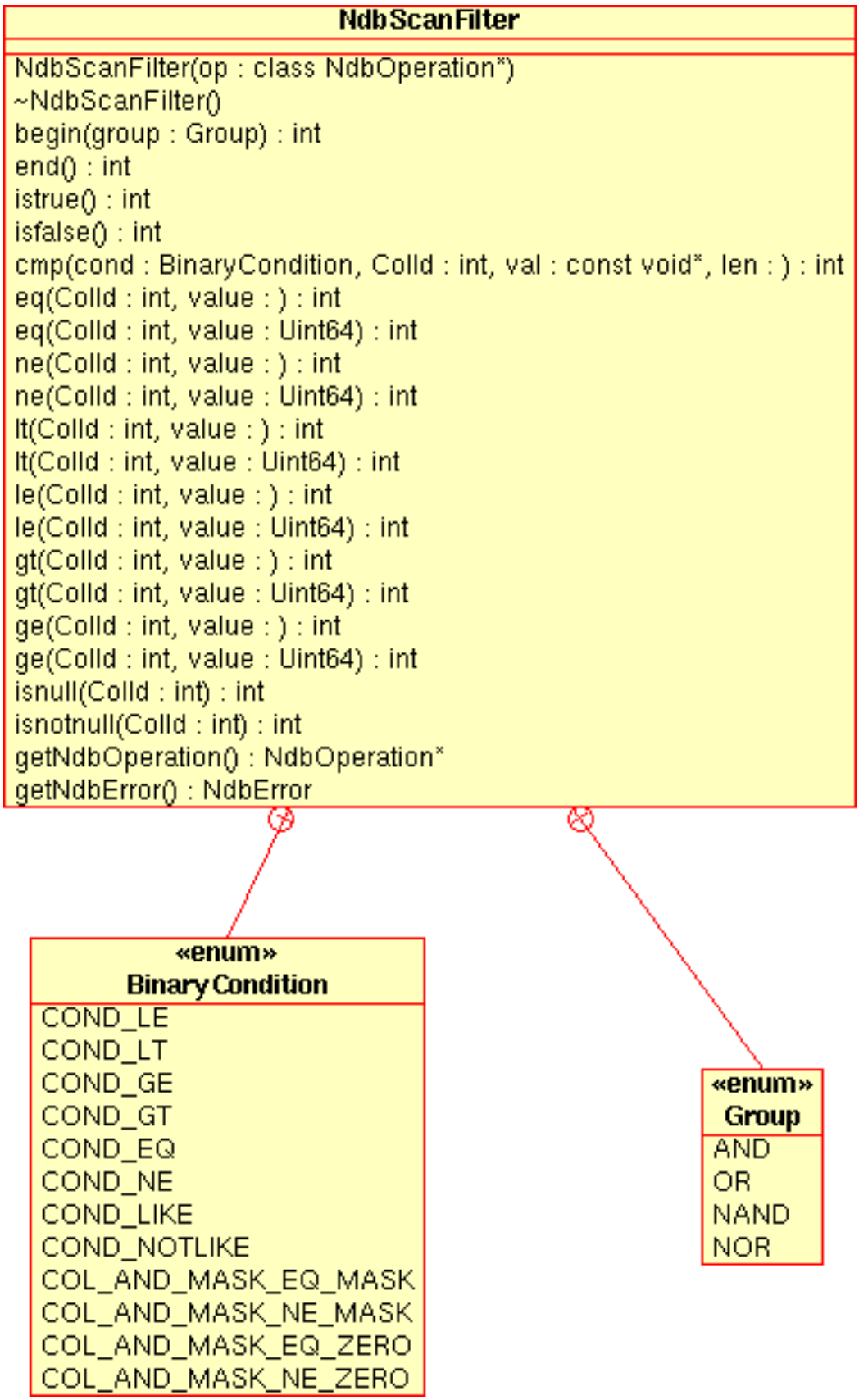
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.17.2, "`NdbScanFilter` Methods".

Types. The `NdbScanFilter` class defines two public types:

- `BinaryCondition`: The type of condition, such as lower bound or upper bound.
- `Group`: A logical grouping operator, such as `AND` or `OR`.

For a discussion of each of these types, along with its possible values, see Section 2.3.17.1, "`NdbScanFilter` Types".

Class diagram. This diagram shows all the public members of the `NdbScanFilter` class:



2.3.17.1. `NdbScanFilter` Types

This section details the public types belonging to the `NdbScanFilter` class.

2.3.17.1.1. The `NdbScanFilter::BinaryCondition` Type

Description. This type represents a condition based on the comparison of a column value with some arbitrary value — that is, a bound condition. A value of this type is used as the first argument to `NdbScanFilter::cmp()`.

Enumeration values.

Enumeration value	Description	Type of column values compared
`COND_LE`	Lower bound (`<=`)	integer
`COND_LT`	Strict lower bound (`<`)	integer
`COND_GE`	Upper bound (`>=`)	integer
`COND_GT`	Strict upper bound (`>`)	integer
`COND_EQ`	Equality (`=`)	integer
`COND_NE`	Inequality (`<>` or `!=`)	integer
`COND_LIKE`	`LIKE` condition	string
`COND_NOTLIKE`	`NOT LIKE` condition	string
`COL_AND_MASK_EQ_MASK`	Column value is equal to column value `AND`ed with bitmask	`BIT`
`COL_AND_MASK_NE_MASK`	Column value is not equal to column value `AND`ed with bitmask	`BIT`
`COL_AND_MASK_EQ_ZERO`	Column value `AND`ed with bitmask is equal to zero	`BIT`
`COL_AND_MASK_NE_ZERO`	Column value `AND`ed with bitmask is not equal to zero	`BIT`

String comparisons. Strings compared using `COND_LIKE` and `COND_NOTLIKE` can use the pattern metacharacters `%` and `_`. See Section 2.3.17.2.6, "`NdbScanFilter::cmp()`", for more information.

`BIT` comparisons. The `BIT` comparison operators `COL_AND_MASK_EQ_MASK`, `COL_AND_MASK_NE_MASK`, `COL_AND_MASK_EQ_ZERO`, and `COL_AND_MASK_NE_ZERO` were added in MySQL Cluster NDB 6.3.20. Corresponding methods are also available for `NdbInterpretedCode` and `NdbOperation` beginning with MySQL Cluster NDB 6.3.20; for more information about these methods, see Section 2.3.14.1.11, "`NdbInterpretedCode` Bitwise Comparison Operations".

2.3.17.1.2. The `NdbScanFilter::Group` Type

Description. This type is used to describe logical (grouping) operators, and is used with the `begin()` method. (See Section 2.3.17.2.2, "`NdbScanFilter::begin()`".)

Enumeration values.

Value	Description
`AND`	Logical `AND`: *`A AND B AND C`*
`OR`	Logical `OR`: *`A OR B OR C`*
`NAND`	Logical `NOT AND`: `NOT (`*`A AND B AND C`*`)`
`NOR`	Logical `NOT OR`: `NOT (`*`A OR B OR C`*`)`

2.3.17.2. `NdbScanFilter` Methods

This section lists and describes the public methods of the `NdbScanFilter` class.

2.3.17.2.1. `NdbScanFilter` Class Constructor

Description. This is the constructor method for `NdbScanFilter`, and creates a new instance of the class.

Signature.

```
NdbScanFilter
    (
      class NdbOperation* op
    )
```

Parameters. This method takes a single parameter: a pointer to the `NdbOperation` to which the filter applies.

Return Value. A new instance of `NdbScanFilter`.

Destructor. The destructor takes no arguments and does not return a value. It should be called to remove the `NdbScanFilter` object when it is no longer needed.

2.3.17.2.2. `NdbScanFilter::begin()`

Description. This method is used to start a compound, and specifies the logical operator used to group together the conditions making up the compound. The default is `AND`.

Signature.

```
int begin
    (
      Group group = AND
    )
```

Parameters. A `Group` value: one of `AND`, `OR`, `NAND`, or `NOR`. See Section 2.3.17.1.2, "The `NdbScanFilter::Group` Type", for additional information.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.3. `NdbScanFilter::end()`

Description. This method completes a compound, signalling that there are no more conditions to be added to it.

Signature.

```
int end
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.4. `NdbScanFilter::istrue()`

Description. Defines a term of the current group as `TRUE`.

Signature.

```
int istrue
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.5. `NdbScanFilter::isfalse()`

Description. Defines a term of the current group as `FALSE`.

Signature.

```
int isfalse
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.6. `NdbScanFilter::cmp()`

Description. This method is used to perform a comparison between a given value and the value of a column.

> **Note**
>
> In many cases, where the value to be compared is an integer, you can instead use one of several convenience methods provided by `NdbScanFilter` for this purpose. See Section 2.3.17.2.7, "`NdbScanFilter` Integer Comparison Methods".

Signature.

```
int cmp
    (
      BinaryCondition condition,
      int columnId,
      const void* value,
      Uint32 length = 0
    )
```

Parameters. This method takes the following parameters:

- *`condition`*: This represents the condition to be tested which compares the value of the column having the column ID *`columnID`* with some arbitrary value. The *`condition`* is a `BinaryCondition` value; for permitted values and the relations that they represent, see Section 2.3.17.1.1, "The `NdbScanFilter::BinaryCondition` Type".

  The *`condition`* values `COND_LIKE` or `COND_NOTLIKE` are used to compare a column value with a string pattern.

- *`columnId`*: This is the column's identifier, which can be obtained using the `Column::getColumnNo()` method (see Section 2.3.1.2.5, "`Column::getColumnNo()`", for details).

- *`value`*: The value to be compared, repesented as a pointer to `void`.

  Using using a `COND_LIKE` or `COND_NOTLIKE` comparison condition, the *`value`* is treated as a string pattern which can include the pattern metacharacters or "wildcard" characters `%` and `_`, which have the meanings shown here:

Metacharacter	Description
`%`	Matches zero or more characters
`_`	Matches exactly one character

  To match against a literal "%" or "_" character, use the backslash (`\`) as an escape character. To match a literal "\" character, use `\\`.

  > **Note**
  >
  > These are the same wildcard characters that are supported by the SQL `LIKE` and `NOT LIKE` operators, and are interpreted in the same way. See String Comparison Functions, for more information.

- *`length`*: The length of the value to be compared. The default value is `0`. Using `0` for the *`length`* has the same effect as comparing to `NULL`, that is using the `isnull()` method (see Section 2.3.17.2.8, "`NdbScanFilter::isnull()`").

Return Value. This method returns an integer whose value can be interpreted as shown here:

- `0`: The comparison is true.
- `-1`: The comparison is false.

2.3.17.2.7. `NdbScanFilter` Integer Comparison Methods

This section provides information about several convenience methods which can be used in lieu of the `NdbScanFilter::cmp()` method when the arbitrary value to be compared is an integer. Each of these methods is essentially a wrapper for `cmp()` that includes an appropriate value of `BinaryCondition` for that method's *`condition`* parameter; for example, `NdbScanFilter::eq()` is defined like this:

```
int eq(int columnId, Uint32 value)
{
  return cmp(BinaryCondition::COND_EQ, columnId, &value, 4);
}
```

For more information about the `cmp()` method, see Section 2.3.17.2.6, "`NdbScanFilter::cmp()`".

2.3.17.2.7.1. `NdbScanFilter::eq()`

Description. This method is used to perform an equality test on a column value and an integer.

Signature.

```
int eq
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int eq
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. This method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.7.2. `NdbScanFilter::ne()`

Description. This method is used to perform an inequality test on a column value and an integer.

Signature.

```
int ne
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int ne
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()` and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.7.3. `NdbScanFilter::lt()`

Description. This method is used to perform a less-than (strict lower bound) test on a column value and an integer.

Signature.

```
int lt
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int lt
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()`, `ne()`, and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.7.4. `NdbScanFilter::le()`

Description. This method is used to perform a less-than-or-equal test on a column value and an integer.

Signature.

```
int le
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int le
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.7.5. `NdbScanFilter::gt()`

Description. This method is used to perform a greater-than (strict upper bound) test on a column value and an integer.

Signature.

```
int gt
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int gt
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.7.6. `NdbScanFilter::ge()`

Description. This method is used to perform a greater-than-or-equal test on a column value and an integer.

Signature.

```
int ge
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int ge
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()`, `lt()`, `le()`, and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return Value. `0` on success, `-1` on failure.

2.3.17.2.8. `NdbScanFilter::isnull()`

Description. This method is used to check whether a column value is `NULL`.

Signature.

```
int isnull
    (
      int ColId
    )
```

Parameters. The ID of the column whose value is to be tested.

Return Value. `0` if the value is `NULL`.

2.3.17.2.9. `NdbScanFilter::isnotnull()`

Description. This method is used to check whether a column value is not `NULL`.

Signature.

```
int isnotnull
    (
      int ColId
    )
```

Parameters. The ID of the column whose value is to be tested.

Return Value. `0` if the value is not `NULL`.

2.3.17.2.10. `NdbScanFilter::getNdbError()`

Description. Because errors encountered when building an NdbScanFilter do not propagate to any involved `NdbOperation` object, it is necessary to use this method to access error information.

Signature.

```
const NdbError& getNdbError
    (
      void
    )
```

Parameters. *None*.

Return Value. A reference to an `NdbError`. See Section 2.3.30, “The `NdbError` Structure”, for more information.

2.3.17.2.11. `NdbScanFilter::getNdbOperation()`

Description. If the `NdbScanFilter` was constructed with an `NdbOperation`, this method can be used to obtain a pointer to that `NdbOperation` object.

Signature.

```
NdbOperation* getNdbOperation
    (
      void
    )
```

Parameters. *None.*

Return Value. A pointer to the `NdbOperation` associated with this `NdbScanFilter`, if there is one. Otherwise, `NULL`.

2.3.18. The `NdbScanOperation` Class

This section describes the `NdbScanOperation` class and its class members.

Parent class. NdbOperation

Child classes. NdbIndexScanOperation

Description. The `NdbScanOperation` class represents a scanning operation used in a transaction. This class inherits from `NdbOperation`. For more information, see Section 2.3.15, "The `NdbOperation` Class".

> **Note**
>
> Beginning with MySQL Cluster NDB 6.2.14 and MySQL Cluster 6.3.12, you must use the `NdbInterpretedCode` class instead of `NdbScanOperation` when writing interpreted programs used for scans. See Section 2.3.14, "The `NdbInterpretedCode` Class", for more information.

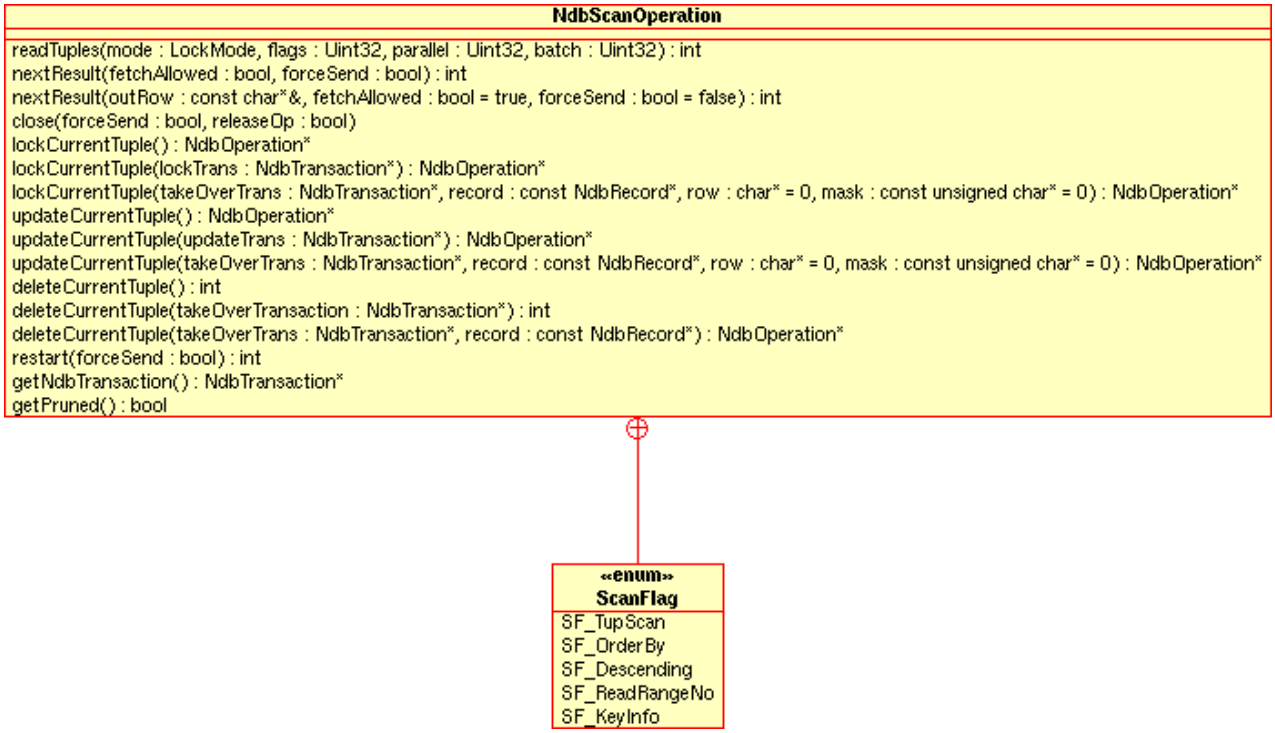
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`readTuples()`	Reads tuples
`nextResult()`	Gets the next tuple
`close()`	Closes the scan
`lockCurrentTuple()`	Locks the current tuple
`updateCurrentTuple()`	Updates the current tuple
`deleteCurrentTuple()`	Deletes the current tuple
`restart()`	Restarts the scan
`getNdbTransaction()`	Gets the `NdbTransaction` object for this scan
`getPruned()`	Used to find out whether this scan is pruned to a single partition

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.18.2, "`NdbScanOperation` Methods".

Types. This class defines a single public type `ScanFlag`. See Section 2.3.18.1, "The `NdbScanOperation::ScanFlag` Type", for details.

Class diagram. This diagram shows all the available members of the `NdbScanOperation` class:



> **Note**
>
> For more information about the use of `NdbScanOperation`, see Section 1.3.2.3.2, “Scan Operations”, and Section 1.3.2.3.3, “Using Scans to Update or Delete Rows”.
>
> Multi-Range Read scans using `NdbScanOperation` are not supported using MySQL Cluster NDB 6.2. They are supported for MySQL Cluster NDB 6.3 beginning with 6.3.17. (Bug#38791) Both NDB 6.2 and NDB 6.3 support Multi-Range Read scans using `NdbRecord`. For more information, see Section 2.3.25, “The `NdbRecord` Interface”.

2.3.18.1. The `NdbScanOperation::ScanFlag` Type

Description. Values of this type are the scan flags used with the `readTuples()` method. More than one may be used, in which case, they are `OR`'ed together as the second argument to that method. See Section 2.3.18.2.1, “`NdbScanOperation::readTuples()`”, for more information.

Enumeration values.

Value	Description
`SF_TupScan`	TUP scan
`SF_OrderBy`	Ordered index scan (ascending)
`SF_Descending`	Ordered index scan (descending)
`SF_ReadRangeNo`	Enables `NdbIndexScanOperation::get_range_no()`
`SF_KeyInfo`	Requests `KeyInfo` to be sent back to the caller

2.3.18.2. `NdbScanOperation` Methods

This section lists and describes the public methods of the `NdbScanOperation` class.

> **Note**
>
> This class has no public constructor. To create an instance of `NdbScanOperation`, it is necessary to use the `NdbTransaction::getNdbScanOperation()` method. See Section 2.3.19.2.2, “`NdbTransaction::getNdbScanOperation()`”.

2.3.18.2.1. `NdbScanOperation::readTuples()`

Description. This method is used to perform a scan.

Signature.

```
virtual int readTuples
    (
      LockMode mode = LM_Read,
      Uint32   flags = 0,
      Uint32   parallel = 0,
      Uint32   batch = 0
    )
```

Parameters. This method takes four parameters, as shown here:

- The lock *mode*; this is a `LockMode` value as described in Section 2.3.15.1.3, "The `NdbOperation::LockMode` Type".

  > **Caution**
  >
  > When scanning with an exclusive lock, extra care must be taken due to the fact that, if two threads perform this scan simultaneously over the same range, then there is a significant probability of causing a deadlock. The likelihood of a deadlock is increased if the scan is also ordered (that is, using `SF_OrderBy` or `SF_Descending`).
  >
  > The `NdbIndexScanOperation::close()` method is also affected by this deadlock, since all outstanding requests are serviced before the scan is actually closed.
  >
  > We are working to resolve this issue in a future release.

- One or more `ScanFlag` values. Multiple values are `OR`'ed together
- The number of fragments to scan in *`parallel`*; use `0` to require that the maximum possible number be used.
- The *`batch`* parameter specifies how many records will be returned to the client from the server by the next `NdbScanOperation::nextResult(true)` method call. Use `0` to specify the maximum automatically.

  > **Note**
  >
  > This parameter was ignored prior to MySQL 5.1.12, and the maximum was used. (Bug#20252)

Return Value. `0` on success, `-1` on failure.

2.3.18.2.2. `NdbScanOperation::nextResult()`

Description. This method is used to fetch the next tuple in a scan transaction. Following each call to `nextResult()`, the buffers and `NdbRecAttr` objects defined in `NdbOperation::getValue()` are updated with values from the scanned tuple.

Signature.

```
int nextResult
    (
      bool fetchAllowed = true,
      bool forceSend = false
    )
```

Beginning with MySQL Cluster NDB 6.2.3, it is also possible to use this method as shown here:

```
int nextResult
    (
      const char*& outRow,
      bool fetchAllowed = true,
      bool forceSend = false
    )
```

Parameters. This method takes two parameters:

- Normally, the NDB API contacts the `NDB` kernel for more tuples whenever it is necessary; setting *`fetchAllowed`* to `false` keeps this from happening.

  Disabling *`fetchAllowed`* by setting it to `false` forces `NDB` to process any records it already has in its caches. When there are no more cached records it returns `2`. You must then call `nextResult()` with *`fetchAllowed`* equal to `true` in order to contact `NDB` for more records.

  While `nextResult(false)` returns `0`, you should transfer the record to another transaction. When `nextResult(false)` returns `2`, you must execute and commit the other transaction. This causes any locks to be transferred to the other transaction, updates or deletes to be made, and then, the locks to be released. Following this, call `nextResult(true)` —

this fetches more records and caches them in the NDB API.

> **Note**
>
> If you do not transfer the records to another transaction, the locks on those records will be released the next time that the NDB Kernel is contacted for more records.

Disabling *fetchAllowed* can be useful when you want to update or delete all of the records obtained in a given transaction, as doing so saves time and speeds up updates or deletes of scanned records.

- *forceSend* defaults to false, and can normally be omitted. However, setting this parameter to true means that transactions are sent immediately. See Section 1.3.4, "The Adaptive Send Algorithm", for more information.

Parameters. Beginning with MySQL Cluster NDB 6.2.3, this method can also be called with the following parameters:

- Calling nextResult() sets a pointer to the next row in *outRow* (if returning 0). This pointer is valid (only) until the next call to nextResult() when *fetchAllowed* is true. The NdbRecord object defining the row format must be specified beforehand using NdbTransaction::scanTable() (or NdbTransaction::scanIndex().
- When false, *fetchAllowed* forces NDB to process any records it already has in its caches. See the description for this parameter in the previous *Parameters* subsection for more details.
- Setting *forceSend* to true means that transactions are sent immediately, as described in the previous *Parameters* subsection, as well as in Section 1.3.4, "The Adaptive Send Algorithm".

Return Value. This method returns one of the following 4 integer values:

- -1: Indicates that an error has occurred.
- 0: Another tuple has been received.
- 1: There are no more tuples to scan.
- 2: There are no more cached records (invoke nextResult(true) to fetch more records).

2.3.18.2.3. NdbScanOperation::close()

Description. Calling this method closes a scan.

> **Note**
>
> See *Scans and exclusive locks* for information about multiple threads attempting to perform the same scan with an exclusive lock and how this can affect closing the scans..

Signature.

```
void close
    (
      bool forceSend = false,
      bool releaseOp = false
    )
```

Parameters. This method takes two parameters:

- *forceSend* defaults to false; call close() with this parameter set to true in order to force transactions to be sent.
- *releaseOp* also defaults to false; set to true in order to release the operation.

Return Value. *None*.

2.3.18.2.4. NdbScanOperation::lockCurrentTuple()

Description. This method locks the current tuple.

Signature.

```
NdbOperation* lockCurrentTuple
    (
      void
    )
```

or

```
NdbOperation* lockCurrentTuple
    (
      NdbTransaction* lockTrans
    )
```

Beginning with MySQL Cluster NDB 6.2.3, the following signature is also supported for this method, when using `NdbRecord`:

```
NdbOperation *lockCurrentTuple
    (
      NdbTransaction* takeOverTrans,
      const NdbRecord* record,
      char* row = 0,
      const unsigned char* mask = 0
    )
```

Parameters. This method takes a single, optional parameter — the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Parameters when using `NdbRecord`. When using the `NdbRecord` interface (see Section 2.3.25, "The `NdbRecord` Interface"), this method takes these parameters, as described in the following list:

- The transaction (*`takeOverTrans`*) that should perform the lock; when using `NdbRecord` with scans, this parameter is not optional.
- The `NdbRecord` referenced by the scan. This is required, even if no records are being read.
- The *`row`* from which to read. Set this to `NULL` if no read is to occur.
- The *`mask`* pointer is optional. If it is present, then only columns for which the corresponding bit in the mask is set are retrieved by the scan.

> **Important**
>
> Calling an `NdbRecord` scan lock takeover on an `NdbRecAttr`-style scan is not valid, nor is calling an `NdbRecAttr`-style scan lock takeover on an `NdbRecord`-style scan.

Return Value. This method returns a pointer to an `NdbOperation` object, or `NULL`. (See Section 2.3.15, "The `NdbOperation` Class".)

2.3.18.2.5. `NdbScanOperation::updateCurrentTuple()`

Description. This method is used to update the current tuple.

Signature.

```
NdbOperation* updateCurrentTuple
    (
      void
    )
```

or

```
NdbOperation* updateCurrentTuple
    (
      NdbTransaction* updateTrans
    )
```

Beginning with MySQL Cluster NDB 6.2.3, it is also possible to employ this method, when using `NdbRecord` with scans, as shown here:

```
NdbOperation* updateCurrentTuple
    (
      NdbTransaction*      takeOverTrans,
      const NdbRecord*     record,
      const char*          row,
      const unsigned char* mask = 0
    )
```

See Section 2.3.25, "The `NdbRecord` Interface", for more information.

Parameters. This method takes a single, optional parameter — the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Parameters when using `NdbRecord`. When using the `NdbRecord` interface (beginning with MySQL Cluster NDB 6.2.3), this method takes the following parameters, as described here:

- The takeover transaction (*`takeOverTrans`*).
- The *`record`* (`NdbRecord` object) referencing the column used for the scan.
- The *`row`* to read from. If no attributes are to be read, set this equal to `NULL`.
- The *`mask`* pointer is optional. If it is present, then only columns for which the corresponding bit in the mask is set are retrieved by the scan.

Return Value. This method returns an `NdbOperation` object or `NULL`. (See Section 2.3.15, "The `NdbOperation` Class".)

2.3.18.2.6. `NdbScanOperation::deleteCurrentTuple()`

Description. This method is used to delete the current tuple.

Signature.

```
int deleteCurrentTuple
    (
      void
    )
```

or

```
int deleteCurrentTuple
    (
      NdbTransaction* takeOverTransaction
    )
```

Beginning with MySQL Cluster NDB 6.2.3, this method's signature when performing `NdbRecord`-style scans is shown here:

For more information, see Section 2.3.25, "The `NdbRecord` Interface".

Parameters. This method takes a single, optional parameter — the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Parameters when using `NdbRecord`. When used with the `NdbRecord` interface (beginning with MySQL Cluster NDB 6.2.3), this method takes two parameters, as shown here:

- The takeover transaction (*`takeOverTrans`*, a pointer to an `NdbTransaction` object) to employ.
- The *`record`* (`NdbRecord` object) referencing the scan column.

Return Value. `0` on success, `-1` on failure.

2.3.18.2.7. `NdbScanOperation::restart()`

Description. Use this method to restart a scan without changing any of its `getValue()` calls or search conditions.

Signature.

```
int restart
    (
      bool forceSend = false
    )
```

Parameters. Call this method with *`forceSend`* set to `true` in order to force the transaction to be sent.

Return Value. `0` on success, `-1` on failure.

2.3.18.2.8. `NdbScanOperation::getNdbTransaction()`

Description. Gets the `NdbTransaction` object for this scan. Available beginning with MySQL Cluster NDB 6.2.17 and

MySQL Cluster NDB 6.3.19.

Signature.

```
NdbTransaction* getNdbTransaction
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to an `NdbTransaction` object. See Section 2.3.19, "The `NdbTransaction` Class".

2.3.18.2.9. `NdbScanOperation::getPruned()`

Description. This method is used to determine whether or not a given scan operation has been pruned to a single partition. For scans defined using NdbRecord, this method can be called before or after the scan is executed. For scans not defined using `NdbRecord`, `getPruned()` is valid only after the scan has been executed.

Signature.

```
bool getPruned
    (
      void
    ) const
```

Parameters. *None*.

Return Value. `TRUE`, if the scan is pruned to a single table partition.

2.3.19. The `NdbTransaction` Class

This section describes the `NdbTransaction` class and its public members.

Parent class. *None*

Child classes. *None*

Description. A transaction is represented in the NDB API by an `NdbTransaction` object, which belongs to an `Ndb` object and is created using `Ndb::startTransaction()`. A transaction consists of a list of operations represented by the `NdbOperation` class, or by one of its subclasses — `NdbScanOperation`, `NdbIndexOperation`, or `NdbIndexScanOperation` (see Section 2.3.15, "The `NdbOperation` Class"). Each operation access exactly one table.

Using Transactions. After obtaining an `NdbTransaction` object, it is employed as follows:

- An operation is allocated to the transaction using one of these methods:
  - `getNdbOperation()`
  - `getNdbScanOperation()`
  - `getNdbIndexOperation()`
  - `getNdbIndexScanOperation()`

  Calling one of these methods defines the operation. Several operations can be defined on the same `NdbTransaction` object, in which case they are executed in parallel. When all operations are defined, the `execute()` method sends them to the `NDB` kernel for execution.
- The `execute()` method returns when the `NDB` kernel has completed execution of all operations previously defined.

  > **Important**
  >
  > All allocated operations should be properly defined before calling the `execute()` method.
- `execute()` performs its task in one of 3 modes, listed here:
  - `NdbTransaction::NoCommit`: Executes operations without committing them.
  - `NdbTransaction::Commit`: Executes any remaining operation and then commits the complete transaction.
  - `NdbTransaction::Rollback`: Rolls back the entire transaction.

`execute()` is also equipped with an extra error handling parameter, which provides two alternatives:

- `NdbOperation::AbortOnError`: Any error causes the transaction to be aborted. This is the default behaviour.
- `NdbOperation::AO_IgnoreError`: The transaction continues to be executed even if one or more of the operations defined for that transaction fails.

> **Note**
>
> In MySQL 5.1.15 and earlier, these values were `NdbTransaction::AbortOnError` and `NdbTransaction::AO_IgnoreError`.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`getNdbOperation()`	Gets an `NdbOperation`
`getNdbScanOperation()`	Gets an `NdbScanOperation`
`getNdbIndexScanOperation()`	Gets an `NdbIndexScanOperation`
`getNdbIndexOperation()`	Gets an `NdbIndexOperation`
`execute`	Executes a transaction
`refresh()`	Keeps a transaction from timing out
`close()`	Closes a transaction
`getGCI()`	Gets a transaction's global checkpoint ID (GCI)
`getTransactionId()`	Gets the transaction ID
`commitStatus()`	Gets the transaction's commit status
`getNdbError()`	Gets the most recent error
`getNdbErrorOperation()`	Gets the most recent operation which caused an error
`getNdbErrorLine()`	Gets the line number where the most recent error occurred
`getNextCompletedOperation()`	Gets operations that have been executed; used for finding errors
`readTuple()`	Read a tuple using `NdbRecord`
`insertTuple()`	Insert a tuple using `NdbRecord`
`updateTuple()`	Update a tuple using `NdbRecord`
`writeTuple()`	Write a tuple using `NdbRecord`
`deleteTuple()`	Delete a tuple using `NdbRecord`
`scanTable()`	Perform a table scan using `NdbRecord`
`scanIndex()`	Perform an index scan using `NdbRecord`

> **Important**
>
> The methods `readTuple()`, `insertTuple()`, `updateTuple()`, `writeTuple()`, `deleteTuple()`, `scanTable()`, and `scanIndex()` require the use of `NdbRecord`, which is available beginning with MySQL Cluster NDB 6.2.3.

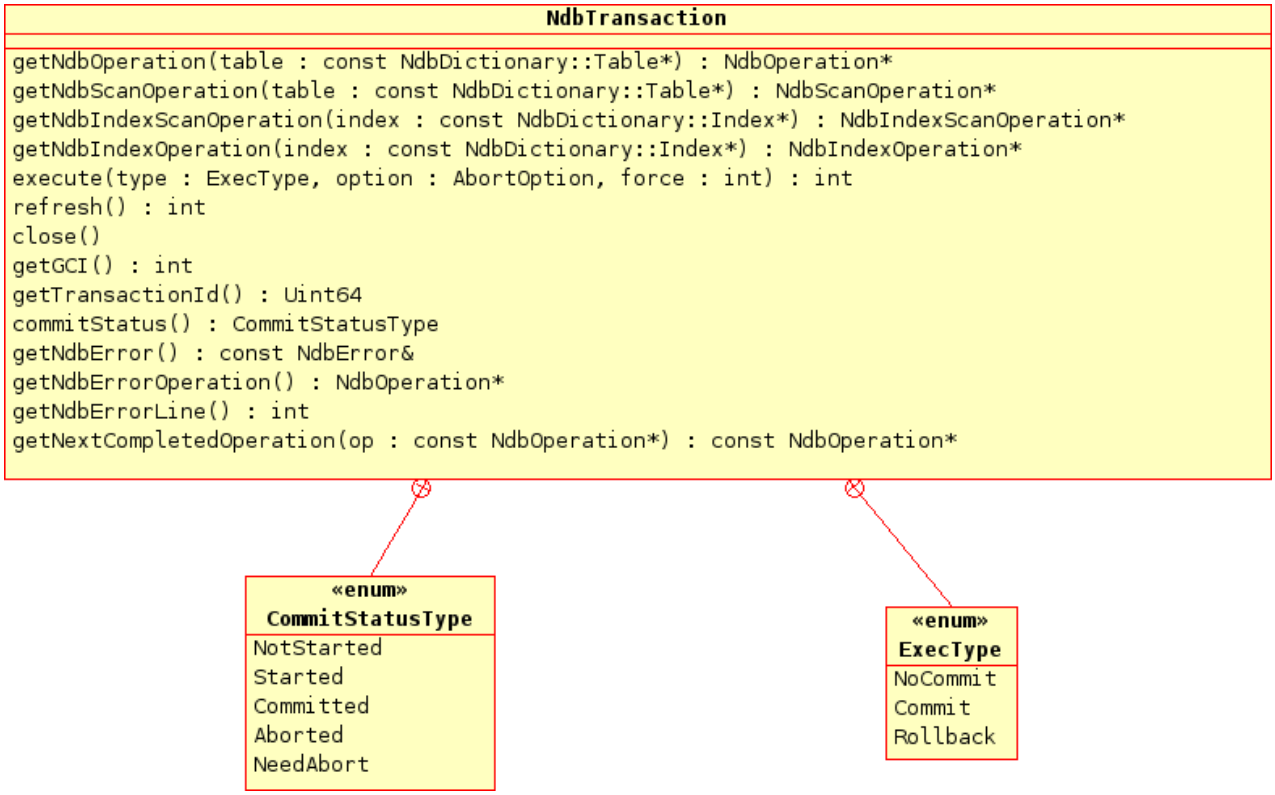
For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.19.2, "`NdbTransaction` Methods".

Types. `NdbTransaction` defines 3 public types as shown in the following table:

Type	Purpose / Use
`AbortOption`	Determines whether failed operations cause a transaction to abort
`CommitStatusType`	Describes the transaction's commit status
`ExecType`	Determines whether the transaction should be committed or rolled back

For a discussion of each of these types, along with its possible values, see Section 2.3.19.1, "`NdbTransaction` Types".

Class diagram. This diagram shows all the available methods and enumerated types of the `...` class:



> **Note**
>
> The methods `readTuple()`, `insertTuple()`, `updateTuple()`, `writeTuple()`, `deleteTuple()`, `scanTable()`, and `scanIndex()` (available beginning with MySQL Cluster NDB 6.2.3) are not shown in the diagram due to space cosniderations.

2.3.19.1. `NdbTransaction` Types

This section details the public types belonging to the `NdbTransaction` class.

2.3.19.1.1. The `NdbTransaction::AbortOption` Type

Description. This type is used to determine whether failed operations should force a transaction to be aborted. It is used as an argument to the `execute()` method — see Section 2.3.19.2.5, "`NdbTransaction::execute()`", for more information.

Enumeration values.

Value	Description
`AbortOnError`	A failed operation causes the transaction to abort.
`AO_IgnoreOnError`	Failed operations are ignored; the transaction continues to execute.

> **Important**
>
> Beginning with MySQL Cluster NDB 6.2.0, this type belongs to the `NdbOperation` class and its possible values and default behavior have changed. NDB API application code written against previous versions of MySQL Cluster that refers explicitly to `NdbTransaction::AbortOption` values must be modified to work with MySQL Cluster NDB 6.2.0 or later.
>
> In particular, this effects the use of `NdbTransaction::execute()` in MySQL Cluster NDB 6.2.0 and later. See Section 2.3.15.1.1, "The `NdbOperation::AbortOption` Type", and Section 2.3.19.2.5, "`NdbTransaction::execute()`", for more information.

2.3.19.1.2. The `NdbTransaction::CommitStatusType` Type

Description. This type is used to describe a transaction's commit status.

Enumeration values.

Value	Description
`NotStarted`	The transaction has not yet been started.
`Started`	The transaction has started, but is not yet committed.
`Committed`	The transaction has completed, and has been committed.
`Aborted`	The transaction was aborted.
`NeedAbort`	The transaction has encountered an error, but has not yet been aborted.

A transaction's commit status ca be read using the `commitStatus()` method. See Section 2.3.19.2.10, "`NdbTransaction::commitStatus()`".

2.3.19.1.3. The `NdbTransaction::ExecType` Type

Description. This type sets the transaction's execution type — that is, whether it should execute, execute and commit, or abort. It is used as a parameter to the `execute()` method. (See Section 2.3.19.2.5, "`NdbTransaction::execute()`".)

Enumeration values.

Value	Description
`NoCommit`	The transaction should execute, but not commit.
`Commit`	The transaction should execute and be committed.
`Rollback`	The transaction should be rolled back.

2.3.19.2. `NdbTransaction` Methods

This section lists and describes the public methods of the `NdbTransaction` class.

2.3.19.2.1. `NdbTransaction::getNdbOperation()`

Description. This method is used to create an `NdbOperation` associated with a given table.

> **Note**
>
> All operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbOperation* getNdbOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The `Table` object on which the operation is to be performed. See Section 2.3.21, "The `Table` Class".

Return Value. A pointer to the new `NdbOperation`. See Section 2.3.15, "The `NdbOperation` Class".

2.3.19.2.2. `NdbTransaction::getNdbScanOperation()`

Description. This method is used to create an `NdbScanOperation` associated with a given table.

> **Note**
>
> All scan operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbScanOperation* getNdbScanOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The `Table` object on which the operation is to be performed. See Section 2.3.21, "The `Table` Class".

Return Value. A pointer to the new `NdbScanOperation`. See Section 2.3.18, "The `NdbScanOperation` Class".

2.3.19.2.3. `NdbTransaction::getNdbIndexScanOperation()`

Description. This method is used to create an `NdbIndexScanOperation` associated with a given table.

> **Note**
>
> All index scan operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbIndexScanOperation* getNdbIndexScanOperation
    (
      const NdbDictionary::Index* index
    )
```

Parameters. The `Index` object on which the operation is to be performed. See Section 2.3.5, "The `Index` Class".

Return Value. A pointer to the new `NdbIndexScanOperation`. See Section 2.3.13, "The `NdbIndexScanOperation` Class".

2.3.19.2.4. `NdbTransaction::getNdbIndexOperation()`

Description. This method is used to create an `NdbIndexOperation` associated with a given table.

> **Note**
>
> All index operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbIndexOperation* getNdbIndexOperation
    (
      const NdbDictionary::Index* index
    )
```

Parameters. The `Index` object on which the operation is to be performed. See Section 2.3.5, "The `Index` Class".

Return Value. A pointer to the new `NdbIndexOperation`. See Section 2.3.12, "The `NdbIndexOperation` Class".

2.3.19.2.5. `NdbTransaction::execute()`

Description. This method is used to execute a transaction.

Signature.

```
int execute
    (
      ExecType execType,
      NdbOperation::AbortOption abortOption = NdbOperation::DefaultAbortOption,
      int force = 0
    )
```

Parameters. The execute method takes 3 parameters, as described here:

- The execution type (`ExecType` value); see Section 2.3.19.1.3, "The `NdbTransaction::ExecType` Type", for more information and possible values.
- An abort option (`NdbOperation::AbortOption` value).

> **Note**
>
> Prior to MySQL Cluster NDB 6.2.0, *`abortOption`* was of type `NdbTransaction::AbortOption`; the `AbortOption` type became a member of `NdbOperation` in MySQL Cluster NDB 6.2.0), and its default value was `NdbTransaction::AbortOnError`. See Section 2.3.15.1.1, "The `NdbOperation::AbortOption` Type", for more information.
>
> Also beginning with MySQL Cluster NDB 6.2.0, errors arising from this method are found with `NdbOpera-`

> `tion::getNdbError()` rather than `NdbTransaction::getNdbError()`. See Section 2.3.15.2.5, "`Ndb-Operation::getNdbError()`", for more information.

- A *`force`* parameter, which determines when operations should be sent to the `NDB` Kernel:
  - `0`: Non-forced; detected by the adaptive send algorithm.
  - `1`: Forced; detected by the adaptive send algorithm.
  - `2`: Non-forced; not detected by the adaptive send algorithm.

  See Section 1.3.4, "The Adaptive Send Algorithm".

Return Value. `0` on success, `-1` on failure. The fact that the transaction did not abort does not necessarily mean that each operation was successful; you must check each operation individually for errors.

In MySQL 5.1.15 and earlier versions, this method returned `-1` for some errors even when the trasnsaction itself was not aborted; beginning with MySQL 5.1.16, this method reports a failure *if and only if* the transaction was aborted. (This change was made due to the fact it had been possible to construct cases where there was no way to determine whether or not a transaction was actually aborted.) However, the transaction's error information is still set in such cases to reflect the actual error code and category.

This means, in the case where a **`NoDataFound`** error is a possibility, you must now check for it explicitly, for example:

```
Ndb_cluster_connection myConnection;

if( myConnection.connect(4, 5, 1) )
{
  cout << "Unable to connect to cluster within 30 secs." << endl;
  exit(-1);
}

Ndb myNdb(&myConnection, "test");

//  define operations...

myTransaction = myNdb->startTransaction();

if(myTransaction->getNdbError().classification == NdbError:NoDataFound)
{
  cout << "No records found." << endl;
  //  ...
}

myNdb->closeTransaction(myTransaction);
```

2.3.19.2.6. `NdbTransaction::refresh()`

Description. This method updates the transaction's timeout counter, and thus avoids aborting due to transaction timeout.

> **Note**
>
> It is not advisable to take a lock on a record and maintain it for a extended time since this can impact other transactions.

Signature.

```
int refresh
    (
      void
    )
```

Parameters. *None*.

Return Value. `0` on success, `-1` on failure.

2.3.19.2.7. `NdbTransaction::close()`

Description. This method closes a transaction. It is equivalent to calling `Ndb::closeTransaction()` (see Section 2.3.8.1.9, "`Ndb::closeTransaction()`").

> **Important**
>
> If the transaction has not yet been committed, it is aborted when this method is called. See Section 2.3.8.1.8, "`Ndb::startTransaction()`".

Signature.

```
void close
    (
      void
    )
```

Parameters. *None*.

Return Value. *None*.

2.3.19.2.8. `NdbTransaction::getGCI()`

Description. This method retrieves the transaction's global checkpoint ID (GCI).

Each committed transaction belongs to a GCI. The log for the committed transaction is saved on disk when a global checkpoint occurs.

By comparing the GCI of a transaction with the value of the latest GCI restored in a restarted NDB Cluster, you can determine whether or not the transaction was restored.

> **Note**
>
> Whether or not the global checkpoint with this GCI has been saved on disk cannot be determined by this method.

> **Important**
>
> The GCI for a scan transaction is undefined, since no updates are performed in scan transactions.

Signature.

```
int getGCI
    (
      void
    )
```

Parameters. *None*.

Return Value. The transaction's GCI, or `-1` if none is available.

> **Note**
>
> No GCI is available until `execute()` has been called with `ExecType::Commit`.

2.3.19.2.9. `NdbTransaction::getTransactionId()`

Description. This method is used to obtain the transaction ID.

Signature.

```
Uint64 getTransactionId
    (
      void
    )
```

Parameters. *None*.

Return Value. The transaction ID, as an unsigned 64-bit integer.

2.3.19.2.10. `NdbTransaction::commitStatus()`

Description. This method gets the transaction's commit status.

Signature.

```
CommitStatusType commitStatus
    (
      void
    )
```

Parameters. *None*.

Return Value. The commit status of the transaction, a `CommitStatusType` value. See Section 2.3.19.1.2, “The `NdbTransaction::CommitStatusType` Type”.

2.3.19.2.11. NdbTransaction::getNdbError()

Description. This method is used to obtain the most recent error (NdbError).

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to an NdbError object. See Section 2.3.30, "The NdbError Structure".

> **Note**
>
> For additional information about handling errors in transactions, see Section 1.3.2.3.5, "Error Handling".

2.3.19.2.12. NdbTransaction::getNdbErrorOperation()

Description. This method retrieves the operation that caused an error.

> **Tip**
>
> To obtain more information about the actual error, use the NdbOperation::getNdbError() method of the NdbOperation object returned by getNdbErrorOperation(). (See Section 2.3.15.2.5, "NdbOperation::getNdbError()".)

Signature.

```
NdbOperation* getNdbErrorOperation
    (
      void
    )
```

Parameters. *None*.

Return Value. A pointer to an NdbOperation.

> **Note**
>
> For additional information about handling errors in transactions, see Section 1.3.2.3.5, "Error Handling".

2.3.19.2.13. NdbTransaction::getNdbErrorLine()

Description. This method return the line number where the most recent error occurred.

Signature.

```
int getNdbErrorLine
    (
      void
    )
```

Parameters. *None*.

Return Value. The line number of the most recent error.

> **Note**
>
> For additional information about handling errors in transactions, see Section 1.3.2.3.5, "Error Handling".

2.3.19.2.14. NdbTransaction::getNextCompletedOperation()

Description. This method is used to retrieve a transaction's completed operations. It is typically used to fetch all operations belonging to a given transaction to check for errors.

NdbTransaction::getNextCompletedOperation(NULL) returns the transaction's first NdbOperation object; NdbTransaction::getNextCompletedOperation(*myOp*) returns the NdbOperation object defined after NdbOperation *myOp*.

Important

This method should only be used after the transaction has been executed, but before the transaction has been closed.

Signature.

```
const NdbOperation* getNextCompletedOperation
    (
      const NdbOperation* op
    ) const
```

Parameters. This method requires a single parameter *op*, which is an operation (`NdbOperation` object), or `NULL`.

Return Value. The operation following *op*, or the first operation defined for the transaction if `getNextCompletedOperation()` was called using `NULL`.

2.3.19.2.15. `NdbTransaction::readTuple()`

Description. This method reads a tuple using `NdbRecord` objects.

Signature.

```
NdbOperation* readTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* result_rec,
      char* result_row,
      NdbOperation::LockMode lock_mode = NdbOperation::LM_Read,
      const unsigned char* result_mask = 0
    )
```

Parameters. This method takes the following parameters:

- *key_rec* is a pointer to an `NdbRecord` for either a table or an index. If on a table, then the operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *key_rec* must include all columns of the key.
- The *key_row* passed to this method defines the primary or unique key of the affected tuple, and must remain valid until `execute()` is called.
- *result_rec* is a pointer to an `NdbRecord` used to hold the result
- *result_row* defines a buffer for the result data.
- *lock_mode* specifies the lock mode in effect for the operation. See Section 2.3.15.1.3, "The `NdbOperation::LockMode` Type", for permitted values and other information.
- *result_mask* defines a subset of attributes to read. Only if `mask[attrId >> 3] & (1<<(attrId & 7))` is set is the column affected. The mask is copied, and so need not remain valid after the method call returns.

Return Value. The `NdbOperation` representing this operation (can be used to check for errors).

2.3.19.2.16. `NdbTransaction::insertTuple()`

Description. Inserts a tuple using `NdbRecord`.

Signature.

```
NdbOperation* insertTuple
    (
      const NdbRecord* record,
      const char* row,
      const unsigned char* mask = 0
    )
```

Parameters. `insertTuple` takes the following parameters:

- A pointer to an `NdbRecord` indicating the *record* to be inserted.
- A *row* of data to be inserted.

- A *mask* which can be used to filter the columns to be inserted.

Return Value. The NdbOperation representing this insert operation.

2.3.19.2.17. NdbTransaction::updateTuple()

Description. Updates a tuple using an NdbRecord object.

Signature.

```
NdbOperation* updateTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* attr_rec,
      const char* attr_row,
      const unsigned char* mask = 0
    )
```

Parameters. updateTuple() takes the following parameters:

- *key_rec* is a pointer to an NdbRecord for either a table or an index. If on a table, then the operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *key_rec* must include all columns of the key.
- The *key_row* passed to this method defines the primary or unique key of the affected tuple, and must remain valid until execute() is called.
- *attr_rec* is an NdbRecord referencing the attribute to be updated.

  Note

  For unique index operations, the *attr_rec* must refer to the underlying table of the index, not to the index itself.
- *attr_row* is a buffer containing the new data for the update.
- The *mask*, if not NULL, defines a subset of attributes to be updated. The mask is copied, and so does not need to remain valid after the call to this method returns.

Return Value. The NdbOperation representing this operation (can be used to check for errors).

2.3.19.2.18. NdbTransaction::writeTuple()

Description. This method is used with NdbRecord to write a tuple of data.

Signature.

```
NdbOperation* writeTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* attr_rec,
      const char* attr_row,
      const unsigned char* mask = 0
    )
```

Parameters. This method takes the following parameters:

- *key_rec* is a pointer to an NdbRecord for either a table or an index. If on a table, then the operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *key_rec* must include all columns of the key.
- The *key_row* passed to this method defines the primary or unique key of the tuple to be written, and must remain valid until execute() is called.
- *attr_rec* is an NdbRecord referencing the attribute to be written.

  Note

  For unique index operations, the *attr_rec* must refer to the underlying table of the index, not to the index itself.
- *attr_row* is a buffer containing the new data.

- The *mask*, if not NULL, defines a subset of attributes to be written. The mask is copied, and so does not need to remain valid after the call to this method returns.

Return Value. The NdbOperation representing this write operation. The operation can be checked for errors if and as necessary.

Example.

```
[To be supplied...]
```

2.3.19.2.19. NdbTransaction::deleteTuple()

Description. Deletes a tuple using NdbRecord.

Signature.

```
NdbOperation* deleteTuple
    (
      const NdbRecord* key_rec,
      const char* key_row
    )
```

Parameters. This method takes the following parameters:

- *key_rec* is a pointer to an NdbRecord for either a table or an index. If on a table, then the delete operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *key_rec* must include all columns of the key.
- The *key_row* passed to this method defines the primary or unique key of the tuple to be deleted, and must remain valid until execute() is called.

Return Value. A pointer to the NdbOperation representing this write operation. The operation can be checked for errors if necessary.

2.3.19.2.20. NdbTransaction::scanTable()

Description. This method performs a table scan, using an NdbRecord object to read out column data.

Signature.

```
NdbScanOperation* scanTable
    (
      const NdbRecord* result_record,
      NdbOperation::LockMode lock_mode = NdbOperation::LM_Read,
      const unsigned char* result_mask = 0,
      Uint32 scan_flags = 0,
      Uint32 parallel = 0,
      Uint32 batch = 0
    )
```

Parameters. The scanTable() method takes the following parameters:

- A pointer to an NdbRecord for storing the result. This *result_record* must remain valid until after the execute() call has been made.
- The *lock_mode* in effect for the operation. See Section 2.3.15.1.3, "The NdbOperation::LockMode Type", for allowable values and other information.
- The *result_mask* pointer is optional. If it is present, only columns for which the corresponding bit (by attribute ID order) in *result_mask* is set will be retrieved in the scan. The *result_mask* is copied internally, so in contrast to *result_record* need not be valid when execute() is invoked.
- *scan_flags* can be used to impose ordering and sorting conditions for scans. See Section 2.3.18.1, "The NdbScanOperation::ScanFlag Type", for a list of permitted values.
- The *parallel* argument is the desired parallelism, or 0 for maximum parallelism (receiving rows from all fragments in parallel), which is the default.
- *batch* determines whether batching is employed. The default is 0 (off).

Return Value. A pointer to the `NdbScanOperation` representing this scan. The operation can be checked for errors if necessary.

2.3.19.2.21. `NdbTransaction::scanIndex()`

Description. This method is used to perform an index scan of a table, using `NdbRecord`. The scan may optionally be ordered.

Signature.

```
NdbIndexScanOperation* scanIndex
    (
      const NdbRecord* key_record,
      const char* low_key,
      Uint32 low_key_count,
      bool low_inclusive,
      const char* high_key,
      Uint32 high_key_count,
      bool high_inclusive,
      const NdbRecord* result_record,
      NdbOperation::LockMode lock_mode = NdbOperation::LM_Read,
      const unsigned char* result_mask = 0,
      Uint32 scan_flags = 0,
      Uint32 parallel = 0,
      Uint32 batch = 0
    )
```

> **Note**
>
> For multi-range scans, the *`low_key`* and *`high_key`* pointers must be unique. In other words, it is not permissible to reuse the same row buffer for several different range bounds within a single scan. However, it is permissible to use the same row pointer as *`low_key`* and *`high_key`* in order to specify an equals bound; it is also permissible to reuse the rows after the `scanIndex()` method returns — thatis, they need not remain valid until `execute()` time (unlike the `NdbRecord` pointers).

Parameters. This method takes the following parameters:

- The *`key_record`* describes the index to be scanned. It must be a key record on the index; that is, the columns which it specifies must include all of the key columns of the index. It must be created from the index to be scanned, and not from the underlying table.
- *`low_key`* determines the lower bound for a range scan.
- *`low_key_count`* determines the number of columns used for the lower bound when specifying a partial prefix for the scan.
- *`low_inclusive`* determines whether the lower bound is considered as a `>=` or `>` relation.
- *`high_key`* determines the upper bound for a range scan.
- *`high_key_count`* determines the number of columns used for the higher bound when specifying a partial prefix for the scan.
- *`high_inclusive`* determines whether the lower bound is considered as a `<=` or `<` relation.
- The *`result_record`* describes the rows to be returned from the scan. For an ordered index scan, *`result_record`* be a key record on the index; that is, the columns which it specifies must include all of the key columns of the index. This is because the index key is needed for merge sorting of the scans returned from each fragment.
- The *`lock_mode`* for the scan must be one of the values specified in Section 2.3.15.1.3, “The `NdbOperation::LockMode` Type”.
- The *`result_mask`* pointer is optional. If it is present, only columns for which the corresponding bit (by attribute ID order) in *`result_mask`* is set will be retrieved in the scan. The *`result_mask`* is copied internally, so in contrast to *`result_record`* need not be valid when `execute()` is invoked.
- *`scan_flags`* can be used to impose ordering and sorting conditions for scans. See Section 2.3.18.1, “The `NdbScanOperation::ScanFlag` Type”, for a list of permitted values.
- The *`parallel`* argument is the desired parallelism, or `0` for maximum parallelism (receiving rows from all fragments in parallel), which is the default.
- *`batch`* determines whether batching is employed. The default is 0 (off).

Return Value. The current `NdbIndexScanOperation`, which can be used for error checking.

2.3.20. The `Object` Class

This class provides meta-information about database objects such as tables and indexes. `Object` subclasses model these and other database objects.

Parent class. NdbDictionary

Child classes. Datafile, Event, Index, LogfileGroup, Table, Tablespace, Undofile

Methods. The following table lists the public methods of the `Object` class and the purpose or use of each method:

Method	Purpose / Use
`getObjectStatus()`	Gets an object's status
`getObjectVersion()`	Gets the version of an object
`getObjectId()`	Gets an object's ID

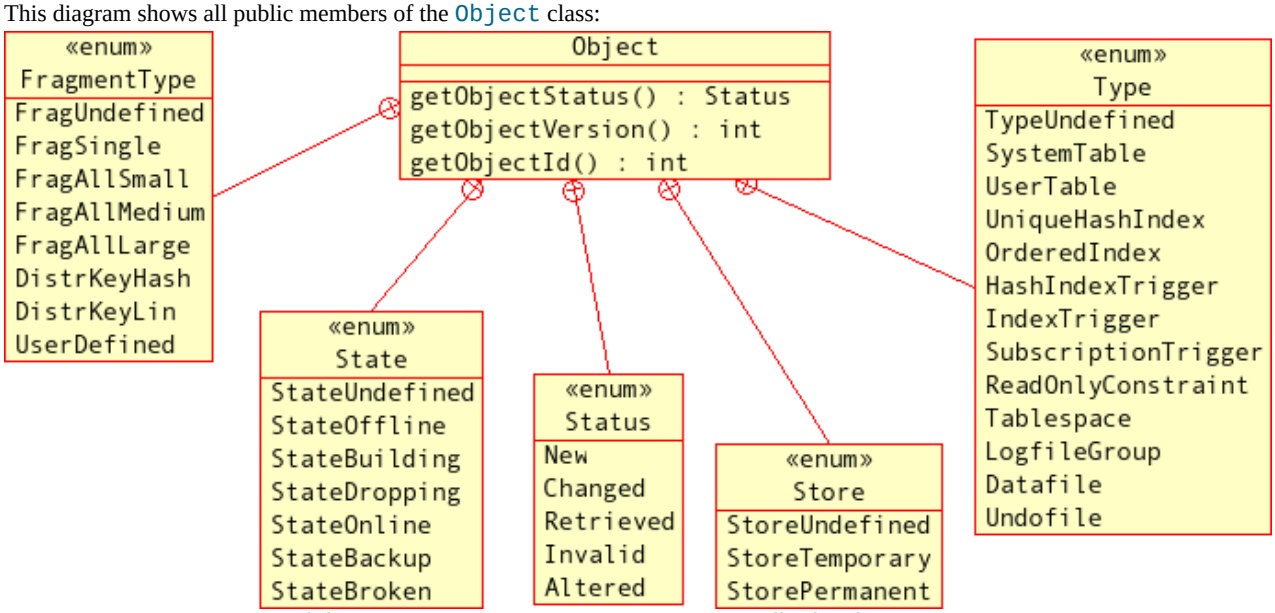
For a detailed discussion of each of these methods, see Section 2.3.20.2, "`Object` Methods".

Types. These are the public types of the `Object` class:

Type	Purpose / Use
`FragmentType`	Fragmentation type used by the object (a table or index)
`State`	The object's state (whether it is usable)
`Status`	The object's state (whether it is available)
`Store`	Whether the object has been temporarily or permanently stored
`Type`	The object's type (what sort of table, index, or other database object the `Object` represents)

For a discussion of each of these types, along with its possible values, see Section 2.3.20.1, "`Object` Class Enumerated Types".

This diagram shows all public members of the `Object` class:



For a visual representation of `Object`'s subclasses, see Section 2.3.10, "The `NdbDictionary` Class".

2.3.20.1. `Object` Class Enumerated Types

This section details the public enumerated types belonging to the `Object` class.

2.3.20.1.1. The `Object::FragmentType` Type

This type describes the `Object`'s fragmentation type.

Description. This parameter specifies how data in the table or index is distributed among the cluster's storage nodes, that is, the

number of fragments per node. The larger the table, the larger the number of fragments that should be used. Note that all replicas count as a single fragment. For a table, the default is `FragAllMedium`. For a unique hash index, the default is taken from the underlying table and cannot currently be changed.

Enumeration values.

Value	Description
`FragUndefined`	The fragmentation type is undefined or the default
`FragAllMedium`	Two fragments per node
`FragAllLarge`	Four fragments per node

2.3.20.1.2. The `Object::State` Type

This type describes the state of the `Object`.

Description. This parameter provides us with the object's state. By *state*, we mean whether or not the object is defined and is in a usable condition.

Enumeration values.

Value	Description
`StateUndefined`	Undefined
`StateOffline`	Offline, not useable
`StateBuilding`	Building (e.g. restore?), not useable(?)
`StateDropping`	Going offline or being dropped; not usable
`StateOnline`	Online, usable
`StateBackup`	Online, being backed up, usable
`StateBroken`	Broken; should be dropped and re-created

2.3.20.1.3. The `Object::Status` Type

This type describes the `Object`'s status.

Description. Reading an object's `Status` tells whether or not it is available in the `NDB` kernel.

Enumeration values.

Value	Description
`New`	The object exists only in memory, and has not yet been created in the `NDB` kernel
`Changed`	The object has been modified in memory, and must be committed in the `NDB` Kernel for changes to take effect
`Retrieved`	The object exists, and has been read into main memory from the `NDB` Kernel
`Invalid`	The object has been invalidated, and should no longer be used
`Altered`	The table has been altered in the `NDB` kernel, but is still available for use

2.3.20.1.4. The `Object::Store` Type

This type describes the `Object`'s persistence.

Description. Reading this value tells us is the object is temporary or permanent.

Enumeration values.

Value	Description
`StoreUndefined`	The object is undefined
`StoreTemporary`	Temporary storage; the object or data will be deleted on system restart
`StorePermanent`	The object or data is permanent; it has been logged to disk

2.3.20.1.5. The `Object::Type` Type

This type describes the `Object`'s type.

Description. The `Type` of the object can be one of several different sorts of index, trigger, tablespace, and so on.

Enumeration values.

Value	Description
`TypeUndefined`	Undefined
`SystemTable`	System table
`UserTable`	User table (may be temporary)
`UniqueHashIndex`	Unique (but unordered) hash index
`OrderedIndex`	Ordered (but not unique) index
`HashIndexTrigger`	Index maintenance (*internal*)
`IndexTrigger`	Index maintenance (*internal*)
`SubscriptionTrigger`	Backup or replication (*internal*)
`ReadOnlyConstraint`	Trigger (*internal*)
`Tablespace`	Tablespace
`LogfileGroup`	Logfile group
`Datafile`	Datafile
`Undofile`	Undofile

2.3.20.2. `Object` Methods

The sections that follow describe each of the public methods of the `Object` class.

> **Important**
>
> All 3 of these methods are pure virtual methods, and are reimplemented in the `Table`, `Index`, and `Event` sub-classes where needed. See Section 2.3.21, "The `Table` Class", Section 2.3.5, "The `Index` Class", and Section 2.3.4, "The `Event` Class".

2.3.20.2.1. `Object::getObjectStatus()`

Description. This method retrieves the status of the object for which it is invoked.

Signature.

```
virtual Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The current `Status` of the `Object`. For possible values, see Section 2.3.20.1.3, "The `Object::Status` Type".

2.3.20.2.2. `Object::getObjectVersion()`

Description. The method gets the current version of the object.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object's version number, an integer.

2.3.20.2.3. `Object::getObjectId()`

Description. This method retrieves the object's ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, an integer.

2.3.21. The `Table` Class

This section describes the `Table` class, which models a database table in the NDB API.

Parent class. NdbDictionary

Child classes. *None*

Description. The `Table` class represents a table in a MySQL Cluster database. This class extends the `Object` class, which in turn is an inner class of the `NdbDictionary` class.

> **Important**
>
> It is possible using the NDB API to create tables independently of the MySQL server. However, it is usually not advisable to do so, since tables created in this fashion cannot be seen by the MySQL server. Similarly, it is possible using `Table` methods to modify existing tables, but these changes (except for renaming tables) are not visible to MySQL.

Calculating Table Sizes. When calculating the data storage one should add the size of all attributes (each attribute consuming a minimum of 4 bytes) and well as 12 bytes overhead. Variable size attributes have a size of 12 bytes plus the actual data storage parts, with an additional overhead based on the size of the variable part. For example, consider a table with 5 attributes: one 64-bit attribute, one 32-bit attribute, two 16-bit attributes, and one array of 64 8-bit attributes. The amount of memory consumed per record by this table is the sum of the following:

- 8 bytes for the 64-bit attribute
- 4 bytes for the 32-bit attribute
- 8 bytes for the two 16-bit attributes, each of these taking up 4 bytes due to right-alignment
- 64 bytes for the array (64 * 1 byte per array element)
- 12 bytes overhead

This totals 96 bytes per record. In addition, you should assume an overhead of about 2% for the allocation of page headers and wasted space. Thus, 1 million records should consume 96 MB, and the additional page header and other overhead comes to approximately 2 MB. Rounding up yields 100 MB.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Table()`	Class constructor
`~Table()`	Destructor
`getName()`	Gets the table's name
`getTableId()`	Gets the table's ID
`getColumn()`	Gets a column (by name) from the table
`getLogging()`	Checks whether logging to disk is enabled for this table
`getFragmentType()`	Gets the table's `FragmentType`
`getKValue()`	Gets the table's `KValue`
`getMinLoadFactor()`	Gets the table's minimum load factor

Method	Purpose / Use
getMaxLoadFactor()	Gets the table's maximum load factor
getNoOfColumns()	Gets the number of columns in the table
getNoOfPrimaryKeys()	Gets the number of columns in the table's primary key.
getPrimaryKey()	Gets the name of the table's primary key
equal()	Compares the table with another table
getFrmData()	Gets the data from the table .FRM file
getFrmLength()	Gets the length of the table's .FRM file
getFragmentData()	Gets table fragment data (ID, state, and node group)
getFragmentDataLen()	Gets the length of the table fragment data
getRangeListData()	Gets a RANGE or LIST array
getRangeListDataLen()	Gets the length of the table RANGE or LIST array
getTablespaceData()	Gets the ID and version of the tablespace containing the table
getTablespaceDataLen()	Gets the length of the table's tablespace data
getLinearFlag()	Gets the current setting for the table's linear hashing flag
getFragmentCount()	Gets the number of fragments for this table
getTablespace()	Gets the tablespace containing this table
getTablespaceNames()	
getObjectType()	Gets the table's object type (Object::Type as opposed to Table::Type)
getObjectStatus()	Gets the table's object status
getObjectVersion()	Gets the table's object version
getObjectId()	Gets the table's object ID
getMaxRows()	Gets the maximum number of rows that this table may contain
getDefaultNoPartitionsFlag()	Checks whether the default number of partitions is being used
getRowGCIIndicator()	
getRowChecksumIndicator()	
setName()	Sets the table's name
addColumn()	Adds a column to the table
setLogging()	Toggle logging of the table to disk
setLinearFlag()	Sets the table's linear hashing flag
setFragmentCount()	Sets the number of fragments for this table
setFragmentType()	Sets the table's FragmentType
setKValue()	Set the KValue
setMinLoadFactor()	Set the table's minimum load factor (MinLoadFactor)
setMaxLoadFactor()	Set the table's maximum load factor (MaxLoadFactor)
setTablespace()	Set the tablespace to use for this table
setStatusInvalid()	
setMaxRows()	Sets the maximum number of rows in the table
setDefaultNoPartitionsFlag()	Toggles whether the default number of partitions should be used for the table
setFrm()	Sets the .FRM file to be used for this table
setFragmentData()	Sets the fragment ID, node group ID, and fragment state
setTablespaceNames()	Sets the tablespace names for fragments
setTablespaceData()	Sets the tablespace ID and version
setRangeListData()	Sets LIST and RANGE partition data
setObjectType()	Sets the table's object type
setRowGCIIndicator()	*Documentation not yet available.*
setRowChecksumIndicator()	*Documentation not yet available.*

Method	Purpose / Use
`aggregate()`	Computes aggregate data for the table
`validate()`	Validates the definition for a new table prior to creating it

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.21.2, "`Table` Methods".

Types. The `Table` class defines a single public type `SingleUserMode`. For more information, see Section 2.3.21.1, "The `Table::SingleUserMode` Type".

Class diagram. This diagram shows all the available methods of the `Table` class:

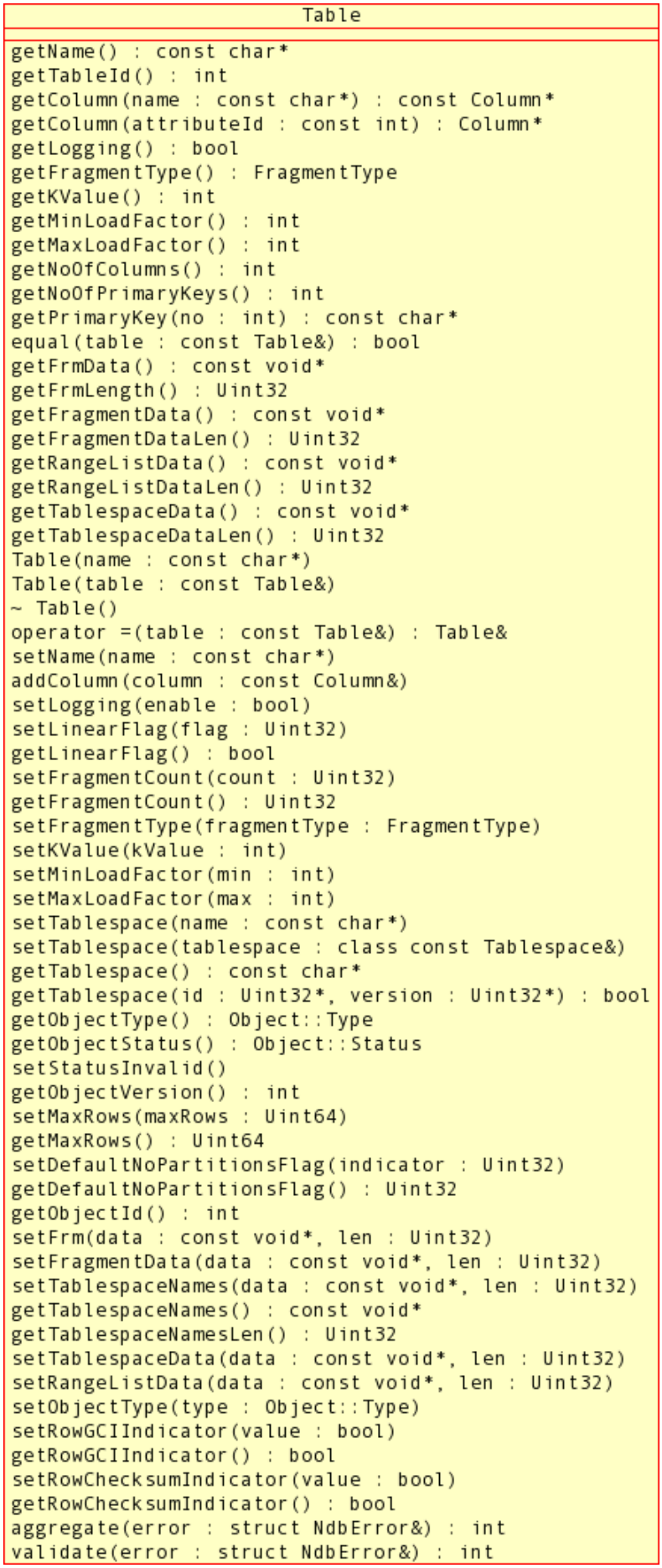


Table
getName() : const char*
getTableId() : int
getColumn(name : const char*) : const Column*
getColumn(attributeId : const int) : Column*
getLogging() : bool
getFragmentType() : FragmentType
getKValue() : int
getMinLoadFactor() : int
getMaxLoadFactor() : int
getNoOfColumns() : int
getNoOfPrimaryKeys() : int
getPrimaryKey(no : int) : const char*
equal(table : const Table&) : bool
getFrmData() : const void*
getFrmLength() : Uint32
getFragmentData() : const void*
getFragmentDataLen() : Uint32
getRangeListData() : const void*
getRangeListDataLen() : Uint32
getTablespaceData() : const void*
getTablespaceDataLen() : Uint32
Table(name : const char*)
Table(table : const Table&)
~ Table()
operator =(table : const Table&) : Table&
setName(name : const char*)
addColumn(column : const Column&)
setLogging(enable : bool)
setLinearFlag(flag : Uint32)
getLinearFlag() : bool
setFragmentCount(count : Uint32)
getFragmentCount() : Uint32
setFragmentType(fragmentType : FragmentType)
setKValue(kValue : int)
setMinLoadFactor(min : int)
setMaxLoadFactor(max : int)
setTablespace(name : const char*)
setTablespace(tablespace : class const Tablespace&)
getTablespace() : const char*
getTablespace(id : Uint32*, version : Uint32*) : bool
getObjectType() : Object::Type
getObjectStatus() : Object::Status
setStatusInvalid()
getObjectVersion() : int
setMaxRows(maxRows : Uint64)
getMaxRows() : Uint64
setDefaultNoPartitionsFlag(indicator : Uint32)
getDefaultNoPartitionsFlag() : Uint32
getObjectId() : int
setFrm(data : const void*, len : Uint32)
setFragmentData(data : const void*, len : Uint32)
setTablespaceNames(data : const void*, len : Uint32)
getTablespaceNames() : const void*
getTablespaceNamesLen() : Uint32
setTablespaceData(data : const void*, len : Uint32)
setRangeListData(data : const void*, len : Uint32)
setObjectType(type : Object::Type)
setRowGCIIndicator(value : bool)
getRowGCIIndicator() : bool
setRowChecksumIndicator(value : bool)
getRowChecksumIndicator() : bool
aggregate(error : struct NdbError&) : int
validate(error : struct NdbError&) : int

2.3.21.1. The `Table::SingleUserMode` Type

Description. Single user mode specifies access rights to the table when single user mode is in effect.

Enumeration values.

Value	Description
`SingleUserModeLocked`	The table is locked (unavailable).
`SingleUserModeReadOnly`	The table is available in read-only mode.
`SingleUserModeReadWrite`	The table is available in read-write mode.

2.3.21.2. `Table` Methods

This section discusses the public methods of the `Table` class.

> **Note**
>
> The assignment (`=`) operator is overloaded for this class, so that it always performs a deep copy.

> **Warning**
>
> As with other database objects, `Table` object creation and attribute changes to existing tables done using the NDB API are not visible from MySQL. For example, if you add a new column to a table using `Table::addColumn()`, MySQL will not see the new column. The only exception to this rule with regard to tables is that you can change the name of an existing table using `Table::setName()`.

2.3.21.2.1. `Table` Constructor

Description. Creates a `Table` instance. There are two version of the `Table` constructor, one for creating a new instance, and a copy constructor.

> **Important**
>
> Tables created in the NDB API using this method are not accessible from MySQL.

Signature. New instance:

```
Table
    (
      const char* name = ""
    )
```

Copy constructor:

```
Table
    (
      const Table& table
    )
```

Parameters. For a new instance, the name of the table to be created. For a copy, a reference to the table to be copied.

Return Value. A `Table` object.

Destructor.

```
virtual ~Table()
```

2.3.21.2.2. `Table:getName()`

Description. Gets the name of a table.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the table (a string).

2.3.21.2.3. `Table::getTableId()`

Description. This method gets a table's ID.

Signature.

```
int getTableId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer.

2.3.21.2.4. `Table::getColumn()`

Description. This method is used to obtain a column definition, given either the index or the name of the column.

Signature. Using the column ID:

```
Column* getColumn
    (
      const int AttributeId
    )
```

Using the column name:

```
Column* getColumn
    (
      const char* name
    )
```

Parameters. Either of: the column's index in the table (as would be returned by the column's `getColumnNo()` method), or the name of the column.

Return Value. A pointer to the column with the specified index or name. If there is no such column, then this method returns `NULL`.

2.3.21.2.5. `Table::getLogging()`

Description. This class is used to check whether a table is logged to disk — that is, whether it is permanent or temporary.

Signature.

```
bool getLogging
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns a Boolean value. If this method returns `true`, then full checkpointing and logging are done on the table. If `false`, then the table is a temporary table and is not logged to disk; in the event of a system restart the table still exists and retains its definition, but it will be empty. The default logging value is `true`.

2.3.21.2.6. `Table::getFragmentType()`

Description. This method gets the table's fragmentation type.

Signature.

```
FragmentType getFragmentType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `FragmentType` value, as defined in Section 2.3.20.1.1, “The `Object::FragmentType` Type”.

2.3.21.2.7. `Table::getKValue()`

Description. This method gets the KValue, a hashing parameter which is currently restricted to the value `6`. In a future release, it may become feasible to set this parameter to other values.

Signature.

```
int getKValue
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer (currently always `6`).

2.3.21.2.8. `Table::getMinLoadFactor()`

Description. This method gets the value of the load factor when reduction of the hash table begins. This should always be less than the value returned by `getMaxLoadFactor()`.

Signature.

```
int getMinLoadFactor
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer (actually, a percentage expressed as an integer — see Section 2.3.21.2.9, "`Table::getMaxLoadFactor()`").

2.3.21.2.9. `Table::getMaxLoadFactor()`

Description. This method returns the load factor (a hashing parameter) when splitting of the containers in the local hash tables begins.

Signature.

```
int getMaxLoadFactor
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer whose maximum value is 100. When the maximum value is returned, this means that memory usage is optimised. Smaller values indicate that less data is stored in each container, which means that keys are found more quickly; however, this also consumes more memory.

2.3.21.2.10. `Table::getNoOfCOlumns()`

Description. This method is used to obtain the number of columns in a table.

Signature.

```
int getNoOfColumns
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer — the number of columns in the table.

2.3.21.2.11. `Table::getNoOfPrimaryKeys()`

Description. This method finds the number of primary key columns in the table.

Signature.

```
int getNoOfPrimaryKeys
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An integer.

2.3.21.2.12. `Table::getPrimaryKey()`

Description. This method is used to obtain the name of the table's primary key.

Signature.

```
const char* getPrimaryKey
    (
      int no
    ) const
```

Parameters. *None*.

Return Value. The name of the primary key, a string (character pointer).

2.3.21.2.13. `Table::equal()`

Description. This method is used to compare one instance of `Table` with another.

Signature.

```
bool equal
    (
      const Table& table
    ) const
```

Parameters. A reference to the `Table` object with which the current instance is to be compared.

Return Value. `true` if the two tables are the same, otherwise `false`.

2.3.21.2.14. `Table::getFrmData()`

Description. The the data from the `.FRM` file associated with the table.

Signature.

```
const void* getFrmData
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A pointer to the `.FRM` data.

2.3.21.2.15. `Table::getFrmLength()`

Description. Gets the length of the table's `.FRM` file data, in bytes.

Signature.

```
Uint32 getFrmLength
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the `.FRM` file data (unsigned 32-bit integer).

2.3.21.2.16. `Table::getFragmentData()`

Description. This method gets the table's fragment data (ID, state, and node group).

Signature.

```
const void* getFragmentData
    (
      void
    ) const
```

Parameters. *None.*

Return Value. A pointer to the data to be read.

2.3.21.2.17. Table::getFragmentDataLen()

Description. Gets the length of the table fragment data to be read, in bytes.

Signature.

```
Uint32 getFragmentDataLen
    (
      void
    ) const
```

Parameters. *None.*

Return Value. The number of bytes to be read, as an unsigned 32-bit integer.

2.3.21.2.18. Table::getRangeListData()

Description. This method gets the range or list data associated with the table.

Signature.

```
const void* getRangeListData
    (
      void
    ) const
```

Parameters. *None.*

Return Value. A pointer to the data.

2.3.21.2.19. Table::getRangeListDataLen()

Description. This method gets the size of the table's range or list array.

Signature.

```
Uint32 getRangeListDataLen
    (
      void
    ) const
```

Parameters. *None.*

Return Value. The length of the list or range array, as an integer.

2.3.21.2.20. Table::getTablespaceData()

Description. This method gets the table's tablespace data (ID and version).

Signature.

```
const void* getTablespaceData
    (
      void
    ) const
```

Parameters. *None.*

Return Value. A pointer to the data.

2.3.21.2.21. Table::getTablespaceDataLen()

Description. This method is used to get the length of the table's tablespace data.

Signature.

```
Uint32 getTablespaceDataLen
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the data, as a 32-bit unsigned integer.

2.3.21.2.22. `Table::getLinearFlag()`

Description. This method retrieves the value of the table's linear hashing flag.

Signature.

```
bool getLinearFlag
    (
      void
    ) const
```

Parameters. *None*.

Return Value. `true` if the flag is set, and `false` if it is not.

2.3.21.2.23. `Table::getFragmentCount()`

Description. This method gets the number of fragments in the table.

Signature.

```
Uint32 getFragmentCount
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of table fragments, as a 32-bit unsigned integer.

2.3.21.2.24. `Table::getTablespace()`

Description. This method is used in two ways: to obtain the name of the tablespace to which this table is assigned; to verify that a given tablespace is the one being used by this table.

Signatures. To obtain the name of the tablespace:

```
const char* getTablespace
    (
      void
    ) const
```

To determine whether the tablespace is the one indicated by the given ID and version:

```
bool getTablespace
    (
      Uint32* id      = 0,
      Uint32* version = 0
    ) const
```

Parameters. The number and types of parameters depend on how this method is being used:

- When used to obtain the name of the tablespace in use by the table, it is called without any arguments.
- When used to determine whether the given tablespace is the one being used by this table, then getTablespace() takes two parameters:

  1. The tablespace *`id`*, given as a pointer to a 32-bit unsigned integer

2. The tablespace *`version`*, also given as a pointer to a 32-bit unsigned integer

The default value for both *`id`* and *`version`* is `0`.

Return Value. The return type depends on how the method is called.

- When `getTablespace()` is called without any arguments, it returns a `Tablespace` object instance. See Section 2.3.22, "The `Tablespace` Class", for more information.
- When called with two arguments, it returns `true` if the tablespace is the same as the one having the ID and version indicated; otherwise, it returns `false`.

2.3.21.2.25. `Table::getObjectType()`

Description. This method is used to obtain the table's type — that is, its `Object::Type` value

Signature.

```
Object::Type getObjectType
    (
      void
    ) const
```

Parameters. *None*.

Return Value. Returns a `Type` value. For possible values, see Section 2.3.20.1.5, "The `Object::Type` Type".

2.3.21.2.26. `Table::getObjectStatus()`

Description. This method gets the table's status — that is, its `Object::Status`.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `Status` value. For possible values, see Section 2.3.20.1.3, "The `Object::Status` Type".

2.3.21.2.27. `Table::getObjectVersion()`

Description. This method gets the table's object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The table's object version, as an integer.

2.3.21.2.28. `Table::getMaxRows()`

Description. This method gets the maximum number of rows that the table can hold. This is used for calculating the number of partitions.

Signature.

```
Uint64 getMaxRows
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The maximum number of table rows, as a 64-bit unsigned integer.

2.3.21.2.29. `Table::getDefaultNoPartitionsFlag()`

Description. This method is used to find out whether the default number of partitions is used for the table.

Signature.

```
Uint32 getDefaultNoPartitionsFlag
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A 32-bit unsigned integer.

2.3.21.2.30. `Table::getObjectId()`

Description. This method gets the table's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID is returned as an integer.

2.3.21.2.31. `Table::getTablespaceNames()`

Description. This method gets a pointer to the names of the tablespaces used in the table fragments.

Signature.

```
const void* getTablespaceNames
    (
      void
    )
```

Parameters. *None*.

Return Value. A pointer to the tablespace name data.

2.3.21.2.32. `Table::getTablespaceNamesLen()`

Description. This method gets the length of the tablespace name data returned by `getTablespaceNames()`. (See Section 2.3.21.2.31, “`Table::getTablespaceNames()`”.)

Signature.

```
Uint32 getTablespaceNamesLen
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The length of the names data, in bytes, as a 32-but unsigned integer.

2.3.21.2.33. `Table::getRowGCIIndicator()`

Description.

Signature.

```
bool getRowGCIIndicator
    (
```

```
      void
    ) const
```

Parameters. *None*.

Return Value. A `true`/`false` value.

2.3.21.2.34. `Table::getRowChecksumIndicator()`

Description.

Signature.

```
bool getRowChecksumIndicator
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A `true`/`false` value.

2.3.21.2.35. `Table::setName()`

Description. This method sets the name of the table.

> **Note**
>
> This is the only `set*()` method of `Table` whose effects are visible to MySQL.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. *`name`* is the (new) name of the table.

Return Value. *None*.

2.3.21.2.36. `Table::addColumn()`

Description. Adds a column to a table.

Signature.

```
void addColumn
    (
      const Column& column
    )
```

Parameters. A reference to the column which is to be added to the table.

Return Value. *None*; however, it does create a copy of the original `Column` object.

2.3.21.2.37. `Table::setLogging()`

Description. Toggles the table's logging state. See Section 2.3.21.2.5, “`Table::getLogging()`”.

Signature.

```
void setLogging
    (
      bool enable
    )
```

Parameters. If *`enable`* is `true`, then logging for this table is enabled; if it is `false`, then logging is disabled.

Return Value. *None*.

2.3.21.2.38. `Table::setLinearFlag()`

Description.

Signature.

```
void setLinearFlag
    (
      Uint32 flag
    )
```

Parameters. The `flag` is a 32-bit unsigned integer.

Return Value. *None*.

2.3.21.2.39. `Table::setFragmentCount()`

Description. Sets the number of table fragments.

Signature.

```
void setFragmentCount
    (
      Uint32 count
    )
```

Parameters. `count` is the number of fragments to be used for the table.

Return Value. *None*.

2.3.21.2.40. `Table::setFragmentType()`

Description. This method sets the table's fragmentation type.

Signature.

```
void setFragmentType
    (
      FragmentType fragmentType
    )
```

Parameters. This method takes one argument, a `FragmentType` value. See Section 2.3.20.1.1, "The `Object::FragmentType` Type", for more information.

Return Value. *None*.

2.3.21.2.41. `Table::setKValue()`

Description. This sets the `KValue`, a hashing parameter.

Signature.

```
void setKValue
    (
      int kValue
    )
```

Parameters. `kValue` is an integer. Currently the only permitted value is `6`. In a future version this may become a variable parameter.

Return Value. *None*.

2.3.21.2.42. `Table::setMinLoadFactor()`

Description. This method sets the minimum load factor when reduction of the hash table begins.

Signature.

```
void setMinLoadFactor
    (
      int min
    )
```

Parameters. This method takes a single parameter `min`, an integer representation of a percentage (for example, `45` represents 45

percent). For more information, see Section 2.3.21.2.8, “`Table::getMinLoadFactor()`”.

Return Value. *None*.

2.3.21.2.43. `Table::setMaxLoadFactor()`

Description. This method sets the maximum load factor when splitting the containers in the local hash tables.

Signature.

```
void setMaxLoadFactor
    (
      int max
    )
```

Parameters. This method takes a single parameter *`max`*, an integer representation of a percentage (for example, `45` represents 45 percent). For more information, see Section 2.3.21.2.9, “`Table::getMaxLoadFactor()`”.

> **Caution**
>
> This should never be greater than the minimum load factor.

Return Value. *None*.

2.3.21.2.44. `Table::setTablespace()`

Description. This method sets the tablespace for the table.

Signatures. Using the name of the tablespace:

```
void setTablespace
    (
      const char* name
    )
```

Using a `Tablespace` object:

```
void setTablespace
    (
      const class Tablespace& tablespace
    )
```

Parameters. This method can be called with a single argument of either of two types:

1. The *`name`* of the tablespace (a string).
2. A reference to an existing `Tablespace` instance.

See Section 2.3.22, “The `Tablespace` Class”.

Return Value. *None*.

2.3.21.2.45. `Table::setMaxRows()`

Description. This method sets the maximum number of rows that can be held by the table.

Signature.

```
void setMaxRows
    (
      Uint64 maxRows
    )
```

Parameters. *`maxRows`* is a 64-bit unsigned integer that represents the maximum number of rows to be held in the table.

Return Value. *None*.

2.3.21.2.46. `Table::setDefaultNoPartitionsFlag()`

Description. This method sets an indicator that determines whether the default number of partitions is used for the table.

Signature.

```
void setDefaultNoPartitionsFlag
    (
      Uint32 indicator
    ) const
```

Parameters. This method takes a single argument *`indicator`*, a 32-bit unsigned integer.

Return Value. *None*.

2.3.21.2.47. `Table::setFrm()`

Description. This method is used to write data to this table's `.FRM` file.

Signature.

```
void setFrm
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method takes two arguments:

- A pointer to the *`data`* to be written.
- The length (*`len`*) of the data.

Return Value. *None*.

2.3.21.2.48. `Table::setFragmentData()`

Description. This method writes an array of fragment information containing the following information:

- Fragment ID
- Node group ID
- Fragment State

Signature.

```
void setFragmentData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method takes two parameters:

- A pointer to the fragment *`data`* to be written
- The length (*`len`*) of this data, in bytes, as a 32-bit unsigned integer

Return Value. *None*.

2.3.21.2.49. `Table::setTablespaceNames()`

Description. Sets the names of the tablespaces used by the table fragments.

Signature.

```
void setTablespaceNames
    (
      const void* data
      Uint32      len
    )
```

Parameters. This method takes two parameters:

- A pointer to the tablespace names *data*
- The length (*len*) of the names data, as a 32-bit unsigned integer.

Return Value. *None*.

2.3.21.2.50. Table::setTablespaceData()

Description. This method sets the tablespace information for each fragment, and includes a tablespace ID and a tablespace version.

Signature.

```
void setTablespaceData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method requires two parameters:

- A pointer to the *data* containing the tablespace ID and version
- The length (*len*) of this data, as a 32-bit unsigned integer.

Return Value. *None*.

2.3.21.2.51. Table::setRangeListData()

Description. This method sets an array containing information that maps range values and list values to fragments. This is essentially a sorted map consisting of fragment ID/value pairs. For range partitions there is one pair per fragment. For list partitions it could be any number of pairs, but at least as many pairs as there are fragments.

Signature.

```
void setRangeListData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method requires two parameters:

- A pointer to the range or list *data* containing the ID/value pairs
- The length (*len*) of this data, as a 32-bit unsigned integer.

Return Value. *None*.

2.3.21.2.52. Table::setObjectType()

Description. This method sets the table's object type.

Signature.

```
void setObjectType
    (
      Object::Type type
    )
```

Parameters. The desired object *type*. This must be one of the `Type` values listed in Section 2.3.20.1.5, "The `Object::Type` Type".

Return Value. *None*.

2.3.21.2.53. Table::setRowGCIIndicator()

Description. *Documentation not yet available*

Signature.

```
void setRowGCIIndicator
    (
      bool value
    ) const
```

Parameters. A `true`/`false` *`value`*.

Return Value. *None*.

2.3.21.2.54. `Table::setRowChecksumIndicator()`

Description. *Documentation not yet available*

Signature.

```
void setRowChecksumIndicator
    (
      bool value
    ) const
```

Parameters. A `true`/`false` *`value`*.

Return Value. *None*.

2.3.21.2.55. `Table::setStatusInvalid()`

Description. Forces the table's status to be invalidated.

Signature.

```
void setStatusInvalid
    (
      void
    ) const
```

Parameters. *None*.

Return Value. *None*.

2.3.21.2.56. `Table::aggregate()`

Description. This method computes aggregate data for the table. It is required in order for aggregate methods such as `getNoOfPrimaryKeys()` to work properly before the table has been created and retrieved via `getTable()`.

> **Note**
>
> This method was added in MySQL 5.1.12. (See Bug#21690.)

Signature.

```
int aggregate
    (
      struct NdbError& error
    )
```

Parameters. A reference to an `NdbError` object.

Return Value. An integer, whose value is `0` on success, and `-1` if the table is in an inconsistent state. In the latter case, the *`error`* is also set.

2.3.21.2.57. `Table::validate()`

Description. This method validates the definition for a new table prior to its being created, and executes the `Table::aggregate()` method, as well as performing additional checks. `validate()` is called automatically when a table is created or retrieved. For this reason, it is usually not necessary to call `aggregate()` or `validate()` directly.

> **Warning**
>
> Even after the `validate()` method is called, there may still exist errors which can be detected only by the `NDB` kernel when the table is actually created.

> **Note**
>
> This method was added in MySQL 5.1.12. (See Bug#21690.)

Signature.

```
int validate
    (
      struct NdbError& error
    )
```

Parameters. A reference to an `NdbError` object.

Return Value. An integer, whose value is `0` on success, and `-1` if the table is in an inconsistent state. In the latter case, the *`error`* is also set.

2.3.22. The `Tablespace` Class

This section discusses the `Tablespace` class and its public members.

Parent class. NdbDictionary

Child classes. *None*

Description. The `Tablespace` class models a MySQL Cluster Disk Data tablespace, which contains the datafiles used to store Cluster Disk Data. For an overview of Cluster Disk Data and their characteristics, see `CREATE TABLESPACE` Syntax, in the MySQL Manual.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support tablespaces; thus the `Tablespace` class is unavailable for NDB API applications written against these MySQL versions.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Tablespace()`	Class constructor
`~Tablespace()`	Virtual destructor method
`getName()`	Gets the name of the tablespace
`getExtentSize()`	Gets the extent size used by the tablespace
`getAutoGrowSpecification()`	Used to obtain the AutoGrowSpecification structure associated with the tablespace
`getDefaultLogfileGroup()`	Gets the name of the tablespace's default logfile group
`getDefaultLogfileGroupId()`	Gets the ID of the tablespace's default logfile group
`getObjectStatus()`	Used to obtain the `Object::Status` of the `Tablespace` instance for which it is called
`getObjectVersion()`	Gets the object version of the `Tablespace` object for which it is invoked
`getObjectId()`	Gets the object ID of a `Tablespace` instance
`setName()`	Sets the name for the tablespace
`setExtentSize()`	Sets the size of the extents used by the tablespace
`setAutoGrowSpecification()`	Used to set the auto-grow characteristics of the tablespace
`setDefaultLogfileGroup()`	Sets the tablespace's default logfile group

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.22.1, “`Tablespace` Methods”.

Types. The Tablespace class defines no public types of its own; however, two of its methods make use of the `AutoGrowSpecification` data structure. Information about this structure can be found in Section 2.3.26, “The `AutoGrowSpecification` Structure”.

Class diagram. This diagram shows all the available methods and enumerated types of the `Tablespace` class:

```
                                Tablespace
                                
Tablespace()
Tablespace(tablespace : const Tablespace&)
~Tablespace()
getName() : const char*
getExtentSize() : Uint32
getAutoGrowSpecification() : const AutoGrowSpecification&
getDefaultLogfileGroup() : const char*
getDefaultLogfileGroupId() : Uint32
getObjectStatus() : Object::Status
getObjectVersion() : int
getObjectId() : int
setName(name : const char*)
setExtentSize(size : Uint32)
setAutoGrowSpecification(autoGrowSpec : const AutoGrowSpecification&)
setDefaultLogfileGroup(name : const char*)
setDefaultLogfileGroup(lGroup : class const LogfileGroup&)
```

2.3.22.1. `Tablespace` Methods

This section provides details of the public members of the NDB API's `Tablespace` class.

2.3.22.1.1. `Tablespace` Constructor

Description. These methods are used to create a new instance of `Tablespace`, or to copy an existing one.

> **Note**
>
> The `NdbDictionary::Dictionary` class also supplies methods for creating and dropping tablespaces. See Section 2.3.3, “The `Dictionary` Class”.

Signatures. New instance:

```
Tablespace
    (
      void
    )
```

Copy constructor:

```
Tablespace
    (
      const Tablespace& tablespace
    )
```

Parameters. New instance: *None*. Copy constructor: a reference *`tablespace`* to an existing `Tablespace` instance.

Return Value. A `Tablespace` object.

Destructor. The class defines a virtual destructor `~Tablespace()` which takes no arguments and returns no value.

2.3.22.1.2. `Tablespace::getName()`

Description. This method retrieves the name of the tablespace.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the tablespace, a string value (as a character pointer).

2.3.22.1.3. `Tablespace::getExtentSize()`

Description. This method is used to retrieve the *extent size* — that is the size of the memory allocation units — used by the tablespace.

> **Note**
>
> The same extent size is used for all datafiles contained in a given tablespace.

Signature.

```
Uint32 getExtentSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The tablespace's extent size in bytes, as an unsigned 32-bit integer.

2.3.22.1.4. `Tablespace::getAutoGrowSpecification()`

Description.

Signature.

```
const AutoGrowSpecification& getAutoGrowSpecification
    (
      void
    ) const
```

Parameters. *None*.

Return Value. A reference to the structure which describes the tablespace auto-grow characteristics — for details, see Section 2.3.26, "The `AutoGrowSpecification` Structure".

2.3.22.1.5. `Tablespace::getDefaultLogfileGroup()`

Description. This method retrieves the name of the tablespace's default logfile group.

> **Note**
>
> Alternatively, you may wish to obtain the ID of the default logfile group — see Section 2.3.22.1.6, "`Tablespace::getDefaultLogfileGroupId()`".

Signature.

```
const char* getDefaultLogfileGroup
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the logfile group (string value as character pointer).

2.3.22.1.6. `Tablespace::getDefaultLogfileGroupId()`

Description. This method retrieves the ID of the tablespace's default logfile group.

> **Note**
>
> You can also obtain directly the name of the default logfile group rather than its ID — see Section 2.3.22.1.5, "`Tablespace::getDefaultLogfileGroup()`".

Signature.

```
Uint32 getDefaultLogfileGroupId
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The ID of the logfile group, as an unsigned 32-bit integer.

2.3.22.1.7. `Tablespace::getObjectStatus()`

Description. This method is used to retrieve the object status of a tablespace.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `Object::Status` value — see Section 2.3.20.1.3, "The `Object::Status` Type", for details.

2.3.22.1.8. `Tablespace::getObjectVersion()`

Description. This method gets the tablespace object version.

Signature.

```
virtual int getObjectVersion
  (
    void
  ) const
```

Parameters. *None*.

Return Value. The object version, as an integer.

2.3.22.1.9. `Tablespace::getObjectId()`

Description. This method retrieves the tablespace's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, as an integer.

2.3.22.1.10. `Tablespace::setName()`

Description. This method sets the name of the tablespace.

Signature.

```
void setName
    (
      const char* name
    ) const
```

Parameters. The *`name`* of the tablespace, a string (character pointer).

Return Value. *None*.

2.3.22.1.11. `Tablespace::setExtentSize()`

Description. This method sets the tablespace's extent size.

Signature.

```
void setExtentSize
    (
      Uint32 size
    )
```

Parameters. The *`size`* to be used for this tablespace's extents, in bytes.

Return Value. *None*.

2.3.22.1.12. `Tablespace::setAutoGrowSpecification()`

Description. This method is used to set the auto-grow characteristics of the tablespace.

Signature.

```
void setAutoGrowSpecification
    (
      const AutoGrowSpecification& autoGrowSpec
    )
```

Parameters. This method takes a single parameter, an `AutoGrowSpecification` data structure. See Section 2.3.26, "The `AutoGrowSpecification` Structure".

Return Value. *None*.

2.3.22.1.13. `Tablespace::setDefaultLogfileGroup()`

Description. This method is used to set a tablespace's default logfile group.

Signature. This method can be called in two different ways. The first of these uses the name of the logfile group, as shown here:

```
void setDefaultLogfileGroup
    (
      const char* name
    )
```

This method can also be called by passing it a reference to a `LogfileGroup` object:

```
void setDefaultLogfileGroup
    (
      const class LogfileGroup& lGroup
    )
```

> **Note**
>
> There is no method for setting a logfile group as the default for a tablespace by referencing the logfile group's ID. (In other words, there is no `set*()` method corresponding to `getDefaultLogfileGroupId()`.)

Parameters. Either the *`name`* of the logfile group to be assigned to the tablespace, or a reference *`lGroup`* to this logfile group.

Return Value. *None*.

2.3.23. The `Undofile` Class

The section discusses the `Undofile` class and its public methods.

Parent class. NdbDictionary

Child classes. *None*

Description. The `Undofile` class models a Cluster Disk Data undofile, which stores data used for rolling back transactions.

> **Note**
>
> In MySQL 5.1, only unindexed column data can be stored on disk. Indexes and indexes columns continue to be stored in memory as with previous versions of MySQL Cluster.
>
> Versions of MySQL prior to 5.1 do not support Disk Data storage and so do not support undofile; thus the `Undofile` class is unavailable for NDB API applications written against these MySQL versions.

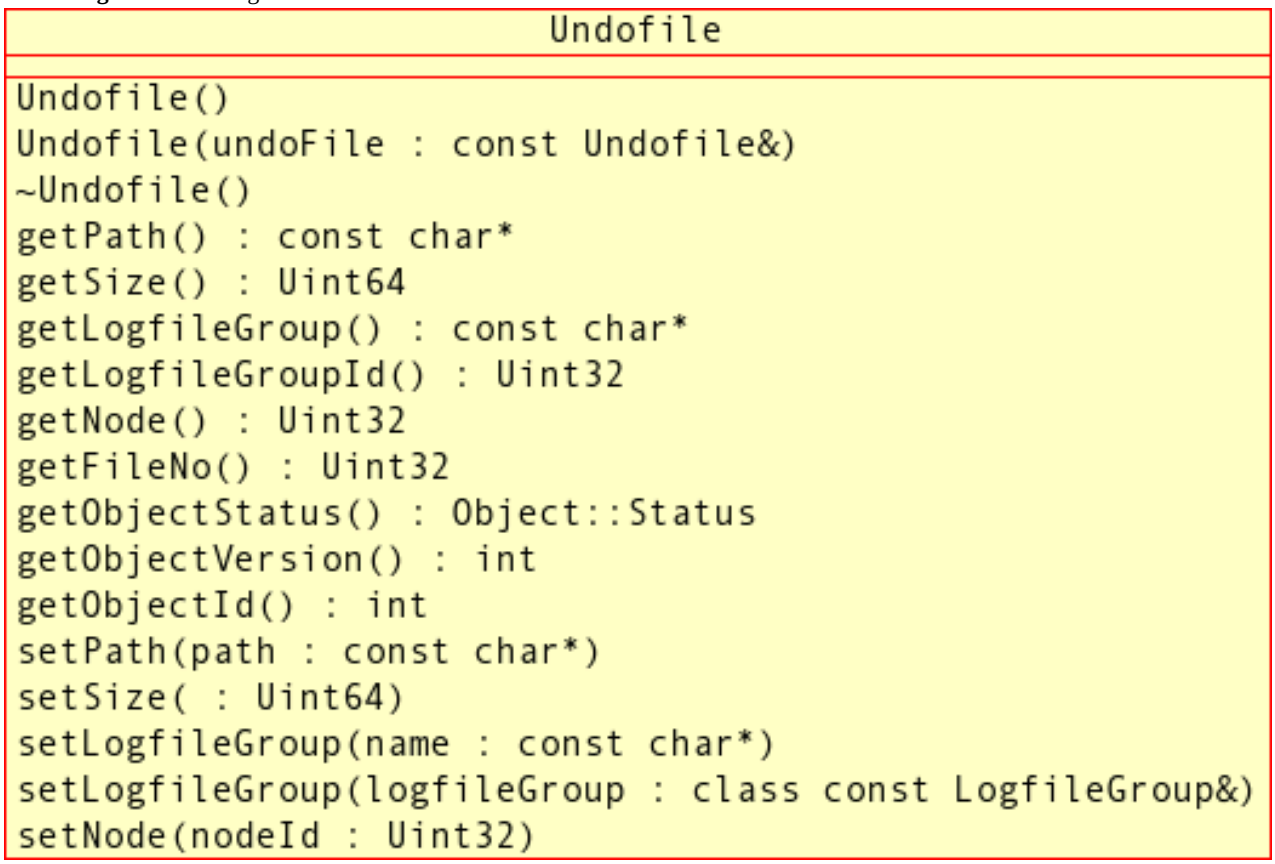
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Undofile()`	Class constructor
`~Undofile()`	Virtual destructor
`getPath()`	Gets the undofile's file system path
`getSize()`	Gets the size of the undofile
`getLogfileGroup()`	Gets the name of the logfile group to which the undofile belongs
`getLogfileGroupId()`	Gets the ID of the logfile group to which the undofile belongs
`getNode()`	Gets the node where the undofile is located
`getFileNo()`	Gets the number of the undofile in the logfile group
`getObjectStatus()`	Gets the undofile's `Status`
`getObjectVersion()`	Gets the undofile's object version
`getObjectId()`	Gets the undofile's object ID
`setPath()`	Sets the file system path for the undofile
`setSize()`	Sets the undofile's size
`setLogfileGroup()`	Sets the undofile's logfile group using the name of the logfile group or a reference to the corresponding `LogfileGroup` object
`setNode()`	Sets the node on which the undofile is located

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.23.1, “`Undofile` Methods”.

Types. The `Undofile` class defines no public types.

Class diagram. This diagram shows all the available methods of the `Undofile` class:



2.3.23.1. `Undofile` Methods

This section details the public methods of the `Undofile` class.

2.3.23.1.1. `Undofile` Constructor

Description. The class constructor can be used to create a new `Undofile` instance, or to copy an existing one.

Signatures. Creates a new instance:

```
Undofile
    (
      void
    )
```

Copy constructor:

```
Undofile
    (
      const Undofile& undoFile
    )
```

Parameters. New instance: *None*. The copy constructor takes a single argument — a reference to the `Undofile` object to be copied.

Return Value. An `Undofile` object.

Destructor. The class defines a virtual destructor which takes no arguments and has the return type `void`.

2.3.23.1.2. `Undofile::getPath()`

Description. This method retrieves the path matching the location of the undofile on the data node's file system.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The file system path, a string (as a character pointer).

2.3.23.1.3. `Undofile::getSize()`

Description. This method gets the size of the undofile in bytes.

Signature.

```
Uint64 getSize
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The size in bytes of the undofile, as an unsigned 64-bit integer.

2.3.23.1.4. `Undofile::getLogfileGroup()`

Description. This method retrieves the name of the logfile group to which the undofile belongs.

Signature.

```
const char* getLogfileGroup
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The name of the logfile group, a string value (as a character pointer).

2.3.23.1.5. `Undofile::getLogfileGroupId()`

Description. This method retrieves the ID of the logfile group to which the undofile belongs.

Note

It is also possible to obtain the name of the logfile group directly. See Section 2.3.23.1.4, “`Undofile::getLogfileGroup()`”

Signature.

```
Uint32 getLogfileGroupId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The ID of the logfile group, as an unsigned 32-bit integer.

2.3.23.1.6. `Undofile::getNode()`

Description. This method is used to retrieve the node ID of the node where the undofile is located.

Signature.

```
Uint32 getNode
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The node ID, as an unsigned 32-bit integer.

2.3.23.1.7. `Undofile::getFileNo()`

Description. The `getFileNo()` method gets the number of the undofile in the logfile group to which it belongs.

Signature.

```
Uint32 getFileNo
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The number of the undofile, as an unsigned 32-bit integer.

2.3.23.1.8. `Undofile::getObjectStatus()`

Description. This method is used to retrieve the object status of an undofile.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return Value. An `Object::Status` value — see Section 2.3.20.1.3, “The `Object::Status` Type”, for details.

2.3.23.1.9. `Undofile::getObjectVersion()`

Description. This method gets the undofile's object version.

Signature.

```
virtual int getObjectVersion
  (
    void
  ) const
```

Parameters. *None*.

Return Value. The object version, as an integer.

2.3.23.1.10. `Undofile::getObjectId()`

Description. This method retrieves the undofile's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return Value. The object ID, as an integer.

2.3.23.1.11. `Undofile::setPath()`

Description. This method is used to set the file system path of the undofile on the data node where it resides.

Signature.

```
void setPath
    (
      const char* path
    )
```

Parameters. The desired *`path`* to the undofile.

Return Value. *None*.

2.3.23.1.12. `Undofile::setSize()`

Description. Sets the size of the undofile in bytes.

Signature.

```
void setSize
    (
      Uint64 size
    )
```

Parameters. The intended *`size`* of the undofile in bytes, as an unsigned 64-bit integer.

Return Value. *None*.

2.3.23.1.13. `Undofile::setLogfileGroup()`

Description. Given either a name or an object reference to a logfile group, the `setLogfileGroup()` method assigns the undofile to that logfile group.

Signature. Using a logfile group name:

```
void setLogfileGroup
    (
      const char* name
    )
```

Using a reference to a LogfileGroup instance:

```
void setLogfileGroup
    (
      const class LogfileGroup & logfileGroup
    )
```

Parameters. The *`name`* of the logfile group (a character pointer), or a reference to a `LogfileGroup` instance.

Return Value. *None*.

2.3.23.1.14. `Undofile::setNode()`

Description. Sets the node on which the logfile group is to reside.

Signature.

```
void setNode
   (
     Uint32 nodeId
   )
```

Parameters. The *`nodeId`* of the data node where the undofile is to be placed; this is an unsigned 32-bit integer.

Return Value. *None*.

2.3.24. The `Ndb_cluster_connection` Class

This class represents a connection to a cluster of data nodes.

Parent class. *None*

Child classes. *None*

Description. An NDB application program should begin with the creation of a single `Ndb_cluster_connection` object, and typically makes use of a single `Ndb_cluster_connection`. The application connects to a cluster management server when this object's `connect()` method is called. By using the `wait_until_ready()` method it is possible to wait for the connection to reach one or more data nodes.

> **Note**
>
> There is no restriction against instantiating multiple `Ndb_cluster_connection` objects representing connections to different management servers in a single application, nor against using these for creating multiple instances of the `Ndb` class. Such `Ndb_cluster_connection` objects (and the `Ndb` instances based on them) are not required even to connect to the same cluster.
>
> For example, it is entirely possible to perform *application-level partitioning* of data in such a manner that data meeting one set of criteria are "handed off" to one cluster via an `Ndb` object that makes use of an `Ndb_cluster_connection` object representing a connection to that cluster, while data not meeting those criteria (or perhaps a different set of criteria) can be sent to a different cluster through a different instance of Ndb that makes use of an `Ndb_cluster_connection` "pointing" to the second cluster.
>
> It is possible to extend this scenario to develop a single application that accesses an arbitrary number of clusters. However, in doing so, the following conditions and requirements must be kept in mind:
>
> - A cluster management server (`ndb_mgmd`) can connect to one and only one cluster without being restarted and reconfigured, as it must read the data telling it which data nodes make up the cluster from a configuration file (`config.ini`).
> - An `Ndb_cluster_connection` object "belongs" to a single management server whose host name or IP address is used in instantiating this object (passed as the *`connectstring`* argument to its constructor); once the object is created, it cannot be used to initiate a connection to a different management server.
>
>   (See Section 2.3.24.1.1, "`Ndb_cluster_connection` Class Constructor".)
> - An `Ndb` object making use of this connection (`Ndb_cluster_connection`) cannot be re-used to connect to a different cluster management server (and thus to a different collection of data nodes making up a cluster). Any given instance of `Ndb` is bound to a specific `Ndb_cluster_connection` when created, and that `Ndb_cluster_connection` is in turn bound to a single and unique management server when it is instantiated.
>
>   (See Section 2.3.8.1.1, "`Ndb` Class Constructor".)
> - The bindings described above persist for the lifetimes of the `Ndb` and `Ndb_cluster_connection` objects in question.
>
> Therefore, it is imperative in designing and implementing any application that accesses multiple clusters in a single session, that a separate set of `Ndb_cluster_connection` and `Ndb` objects be instantiated for connecting to each cluster management server, and that no confusion arises as to which of these is used to access which MySQL Cluster.
>
> It is also important to keep in mind that no direct "sharing" of data or data nodes between different clusters is possible. A data node can belong to one and only one cluster, and any movement of data between clusters must be accomplished on the application level.
>
> For examples demonstrating how connections to two different clusters can be made and used in a single application,

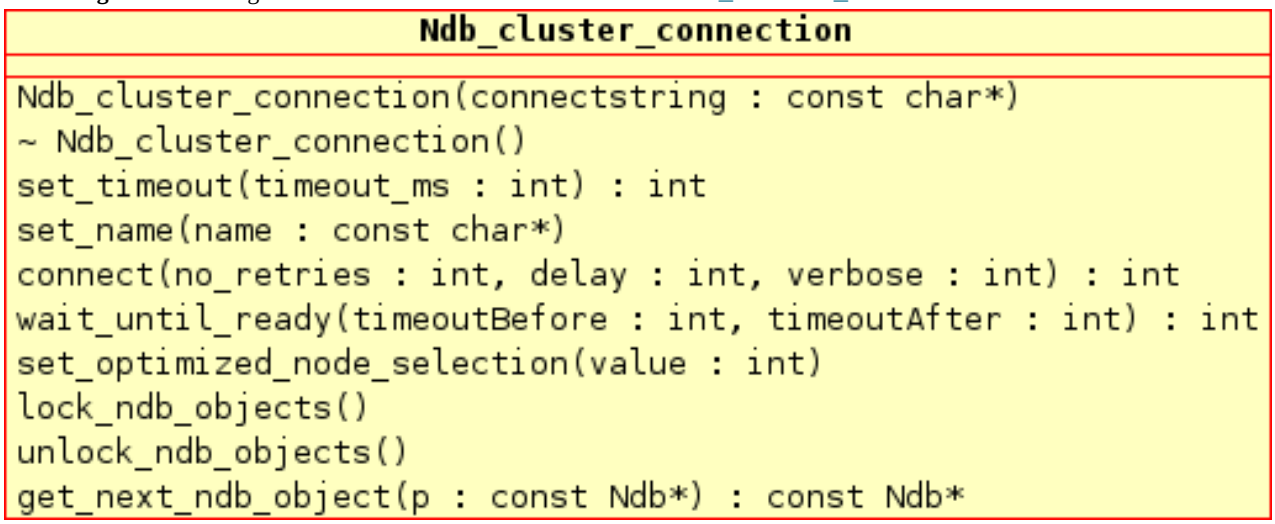
see Section 2.4.2, “Using Synchronous Transactions and Multiple Clusters”, and Section 2.4.8, “Event Handling with Multiple Clusters”.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Ndb_cluster_connection()`	Constructor; creates a connection to a cluster of data nodes.
`set_name()`	Provides a name for the connection
`set_timeout()`	Sets a connection timeout
`connect()`	Connects to a cluster management server.
`wait_until_ready()`	Waits until a connection with one or more data nodes is successful.
`set_optimized_node_selection()`	Used to control node-selection behaviour.
`lock_ndb_objects()`	Disables the creation of new `Ndb` objects.
`unlock_ndb_objects()`	Enables the creation of new `Ndb` objects.
`get_next_ndb_object()`	Used to iterate through multiple `Ndb` objects.

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.24.1, “`Ndb_cluster_connection` Methods”.

Class diagram. This diagram shows all the available methods of the `Ndb_cluster_connection` class:



2.3.24.1. `Ndb_cluster_connection` Methods

2.3.24.1.1. `Ndb_cluster_connection` Class Constructor

Description. This method creates a connection to a MySQL cluster, that is, to a cluster of data nodes. The object returned by this method is required in order to instantiate an `Ndb` object. (See Section 2.3.8, “The `Ndb` Class”.) Thus, every NDB API application requires the use of an `Ndb_cluster_connection`.

Signature.

```
Ndb_cluster_connection
    (
      const char* connectstring = 0
    )
```

Parameters. This method requires a single parameter — a *`connectstring`* pointing to the location of the management server.

Return Value. An instance of `Ndb_cluster_connection`.

2.3.24.1.2. `Ndb_cluster_connection::set_name()`

Description. Sets a name for the connection. If the name is specified, it is reported in the cluster log.

Signature.

```
void set_name
    (
      const char* name
    )
```

Parameters. The *`name`* to be used as an identifier for the connection.

Return Value. *None*.

2.3.24.1.3. `Ndb_cluster_connection::set_timeout()`

Description. Used to set a timeout for the connection, to limit the amount of time that we may block when connecting.

This method is actually a wrapper for the function `ndb_mgm_set_timeout()`. For more information, see Section 3.2.4.11, “ndb_mgm_set_timeout()”.

Signature.

```
int set_timeout
    (
      int timeout_ms
    )
```

Parameters. The length of the timeout, in milliseconds (*`timeout_ms`*). Currently, only multiples of 1000 are accepted.

Return Value. 0 on success; any other value indicates failure.

2.3.24.1.4. `Ndb_cluster_connection::connect()`

Description. This method connects to a cluster management server.

Signature.

```
int connect
    (
      int retries = 0,
      int delay   = 1,
      int verbose = 0
    )
```

Parameters. This method takes three parameters, all of which are optional:

- *`retries`* specifies the number of times to retry the connection in the event of failure. The default value (`0`) means that no additional attempts to connect will be made in the event of failure; a negative value for *`retries`* results in the connection attempt being repeated indefinitely.
- The *`delay`* represents the number of seconds between reconnect attempts; the default is `1` second.
- *`verbose`* indicates whether the method should output a report of its progress, with `1` causing this reporting to be enabled; the default is `0` (reporting disabled).

Return Value. This method returns an `int`, which can have one of the following 3 values:

- **0**: The connection attempt was successful.
- **1**: Indicates a recoverable error.
- **-1**: Indicates an unrecoverable error.

2.3.24.1.5. `Ndb_cluster_connection::wait_until_ready()`

Description. This method waits until the requested connection with one or more data nodes is successful.

Signature.

```
int wait_until_ready
    (
      int timeoutBefore,
      int timeoutAfter
    )
```

Parameters. This method takes two parameters:

- *timeoutBefore* determines the number of seconds to wait until the first “live” node is detected. If this amount of time is exceeded with no live nodes detected, then the method immediately returns a negative value.
- *timeoutAfter* determines the number of seconds to wait after the first “live” node is detected for all nodes to become active. If this amount of time is exceeded without all nodes becoming active, then the method immediately returns a value greater than zero.

If this method returns `0`, then all nodes are “live”.

Return Value. `wait_until_ready()` returns an `int`, whose value is interpreted as follows:

- **`= 0`**: All nodes are “live”.
- **`> 0`**: At least one node is “live” (however, it is not known whether *all* nodes are “live”).
- **`< 0`**: An error occurred.

2.3.24.1.6. `Ndb_cluster_connection::set_optimized_node_selection()`

Description. This method can be used to override the `connect()` method's default behaviour as regards which node should be connected to first.

Signature.

```
void set_optimized_node_selection
    (
      int value
    )
```

Parameters. An integer *value*.

Return Value. *None*.

2.3.24.1.7. `ndb_cluster_connection::get_next_ndb_object()`

Description. This method is used to iterate over a set of `Ndb` objects, retrieving them one at a time.

Signature.

```
const Ndb* get_next_ndb_object
    (
      const Ndb* p
    )
```

Parameters. This method takes a single parameter, a pointer to the last `Ndb` object to have been retrieved or `NULL`.

Return Value. Returns the next `Ndb` object, or `NULL` if no more `Ndb` objects are available.

Iterating over `Ndb` objects. To retrieve all existing `Ndb` objects:

1. Invoke the `lock_ndb_objects()` method. This prevents the creation of any new instances of `Ndb` until the `unlock_ndb_objects()` method is called.
2. Retrieve the first available `Ndb` object by passing `NULL` to `get_next_ndb_object()`. You can retrieve the second `Ndb` object by passing the pointer retrieved by the first call to the next `get_next_ndb_object()` call, and so on. When a pointer to the last available `Ndb` instance is used, the method returns `NULL`.
3. After you have retrieved all desired `Ndb` objects, you should re-enable `Ndb` object creation by calling the `unlock_ndb_objects()` method.

See also Section 2.3.24.1.8, “ndb_cluster_connection::lock_ndb_objects()”, and Section 2.3.24.1.9, “ndb_cluster_connection::unlock_ndb_objects()”.

2.3.24.1.8. ndb_cluster_connection::lock_ndb_objects()

Description. Calling this method prevents the creation of new instances of the Ndb class. This method must be called prior to iterating over multiple Ndb objects using get_next_ndb_object().

Signature.

```
void lock_ndb_objects
    (
      void
    )
```

Parameters. *None*.

Return Value. *None*.

This method was added in MySQL Cluster NDB 6.1.4. For more information, see Section 2.3.24.1.7, “ndb_cluster_connection::get_next_ndb_object()”.

2.3.24.1.9. ndb_cluster_connection::unlock_ndb_objects()

Description. This method undoes the effects of the lock_ndb_objects() method, making it possible to create new instances of Ndb. unlock_ndb_objects() should be called after you have finished retrieving Ndb objects using the get_next_ndb_object() method.

Signature.

```
void unlock_ndb_objects
    (
      void
    )
```

Parameters. *None*.

Return Value. *None*.

For more information, see Section 2.3.24.1.7, “ndb_cluster_connection::get_next_ndb_object()”.

2.3.25. The NdbRecord Interface

NdbRecord is an interface which provides a mapping to a full or a partial record stored in NDB. In the latter case, it can be used in conjunction with a bitmap to assist in access. NdbRecord is available beginning with MySQL Cluster NDB 6.2.3.

NdbRecord has no API methods of its own; rather it acts as a handle that can be passed between various method calls for use in many different sorts of operations, including these:

- Unique key reads and primary key reads
- Table scans and index scans
- DML operations involving unique keys or primary keys
- Operations involving index bounds

The same NdbRecord can be used simultaneously in multiple operations, transactions, and threads.

An NdbRecord can be created in NDB API programs by calling the createRecord() method of the NdbDictionary class. In addition, a number of NDB API methods have additional declarations in MySQL Cluster NDB 6.2.3 and later MySQL Cluster releases that allow the programmer to leverage NdbRecord:

- NdbScanOperation::nextResult()
- NdbScanOperation::lockCurrentTuple()
- NdbScanOperation::updateCurrentTuple()

- NdbScanOperation::deleteCurrentTuple()
- Dictionary::createRecord()
- Dictionary::releaseRecord()
- NdbTransaction::readTuple()
- NdbTransaction::insertTuple()
- NdbTransaction::updateTuple()
- NdbTransaction::writeTuple()
- NdbTransaction::deleteTuple()
- NdbTransaction::scanTable()
- NdbTransaction::scanIndex()

In addition, new members of NdbIndexScanOperation and NdbDictionary are introduced in MySQL Cluster NDB 6.2.3 for use with NdbRecord scans:

- NdbIndexScanOperation::IndexBound is a structure used to describe index scan bounds. See Section 2.3.28, "The IndexBound Structure".
- NdbDictionary::RecordSpecification is a structure used to specify columns and range offsets. See Section 2.3.32, "The RecordSpecification Structure".

Beginning with MySQL Cluster NDB 6.3.24 and MySQL Cluster NDB 7.0.4, you can also use NdbRecord in conjunction with the new Ndb::PartitionSpec structure to perform scans that take advantage of partition pruning, by means of a new variant of NdbIndexScanOperation::setBound(). For more information, see Section 2.3.13.2.5, "NdbIndexScanOperation::setBound", and Section 2.3.31, "The PartitionSpec Structure".

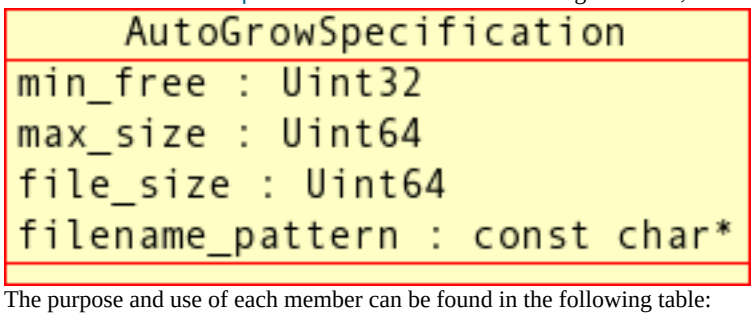
2.3.26. The AutoGrowSpecification Structure

This section describes the AutoGrowSpecification structure.

Parent class. NdbDictionary

Description. The AutoGrowSpecification is a data structure defined in the NdbDictionary class, and is used as a parameter to or return value of some of the methods of the Tablespace and LogfileGroup classes. See Section 2.3.22, "The Tablespace Class", and Section 2.3.6, "The LogfileGroup Class", for more information.

Methods. AutoGrowSpecification has the following members, whose types are as shown in the following diagram:



The purpose and use of each member can be found in the following table:

Name	Description
min_free	???
max_size	???
file_size	???
filename_pattern	???

2.3.27. The Element Structure

This section discusses the `Element` structure.

Parent class. List

Description. The `Element` structure models an element of a list; it is used to store an object in a `List` populated by the methods `Dictionary::listObjects()` and `Dictionary::listIndexes()`.

Attributes. An `Element` has the attributes shown in the following table:

Attribute	Type	Initial Value	Description
`id`	`unsigned int`	`0`	The object's ID
`type`	`Object::Type`	`Object::TypeUndefined`	The object's type — see Section 2.3.20.1.5, "The `Object::Type` Type" for possible values
`state`	`Object::State`	`Object::StateUndefined`	The object's state — see Section 2.3.20.1.2, "The `Object::State` Type" for possible values
`store`	`Object::Store`	`Object::StoreUndefined`	How the object is stored — see Section 2.3.20.1.4, "The `Object::Store` Type" for possible values
`database`	`char*`	`0`	The database in which the object is found
`schema`	`char*`	`0`	The schema in which the object is found
`name`	`char*`	`0`	The object's name

> **Note**
>
> For a graphical representation of this class and its parent-child relationships, see Section 2.3.3, "The `Dictionary` Class".

2.3.28. The `IndexBound` Structure

Parent class. NdbIndexScanOperation

Description. `IndexBound` is a structure used to describe index scan bounds for `NdbRecord` scans. It is available beginning with MySQL Cluster NDB 6.2.3.

Members. These are shown in the following table:

Name	Type	Description
`low_key`	`const char*`	Row containing lower bound for scan (or `NULL` for scan from the start).
`low_key_count`	`Uint32`	Number of columns in lower bound (for bounding by partial prefix).
`low_inclusive`	`bool`	True for `<=` relation, false for `<`.
`high_key`	`const char*`	Row containing upper bound for scan (or `NULL` for scan to the end).
`high_key_count`	`Uint32`	Number of columns in upper bound (for bounding by partial prefix).
`high_inclusive`	`bool`	True for `>=` relation, false for `>`.
`range_no`	`Uint32`	Value to identify this bound; may be read using the `get_range_no()` method (see Section 2.3.13.2.1, "`NdbIndexScanOperation::get_range_no()`"). This value must be less than 8192 (set to zero if it is not being used). For ordered scans, `range_no` must be strictly increasing for each range, or else the result set will not be sorted correctly.

For more information, see Section 2.3.25, “The NdbRecord Interface”.

2.3.29. The Key_part_ptr Structure

This section describes the Key_part_ptr structure.

Parent class. Ndb

Description. Key_part_ptr provides a convenient way to define key-part data when starting transactions and computing hash values, by passing in pointers to distribution key values. When the distribution key has multiple parts, they should be passed as an array, with the last part's pointer set equal to NULL. See Section 2.3.8.1.8, “Ndb::startTransaction()”, and Section 2.3.8.1.10, “Ndb::computeHash()”, for more information about how this structure is used.

Attributes. A Key_part_ptr has the attributes shown in the following table:

Attribute	Type	Initial Value	Description
ptr	const void*	*none*	Pointer to one or more distribution key values
len	unsigned	*none*	The length of the pointer

2.3.30. The NdbError Structure

This section discusses the NdbError data structure, which contains status and other information about errors, including error codes, classifications, and messages.

Description. An NdbError consists of six parts:

1. *Error status*: This describes the impact of an error on the application, and reflects what the application should do when the error is encountered.

   The error status is described by a value of the Status type. See Section 2.3.30.1.1, “The NdbError::Status Type”, for possible Status values and how they should be interpreted.

2. *Error classification*: This represents a logical error type or grouping.

   The error classification is described by a value of the Classification type. See Section 2.3.30.1.2, “The NdbError::Classification Type”, for possible classifications and their interpretation. Additional information is provided in Section 4.2.3, “NDB Error Classifications”.

3. *Error code*: This is an NDB API internal error code which uniquely identifies the error.

   > **Important**
   >
   > It is *not* recommended to write application programs which are dependent on specific error codes. Instead, applications should check error status and classification. More information about errors can also be obtained by checking error messages and (when available) error detail messages. However — like error codes — these error messages and error detail messages are subject to change.

   A listing of current error codes, broken down by classification, is provided in Section 4.2.2, “NDB Error Codes and Messages”. This listing will be updated periodically, or you can check the file storage/ndb/src/ndbapi/ndberror.c in the MySQL 5.1 sources.

4. *MySQL Error code*: This is the corresponding MySQL Server error code. MySQL error codes are not discussed in this document; please see Server Error Codes and Messages, in the MySQL Manual, for information about these.

5. *Error message*: This is a generic, context-independent description of the error.

6. *Error details*: This can often provide additional information (not found in the error message) about an error, specific to the circumstances under which the error is encountered. However, it is not available in all cases.

   Where not specified, the error detail message is NULL.

   > **Important**
   >
   > Specific NDB API error codes, messages, and detail messages are subject to change without notice.

Definition. The NdbError structure contains the following members, whose types are as shown:

- `Status status`

  The error status.

- `Classification classification`

  The error type (classification).

- `int code`

  The NDB API error code.

- `int mysql_code`

  The MySQL error code.

- `const char* message`

  The error message.

- `char* details`

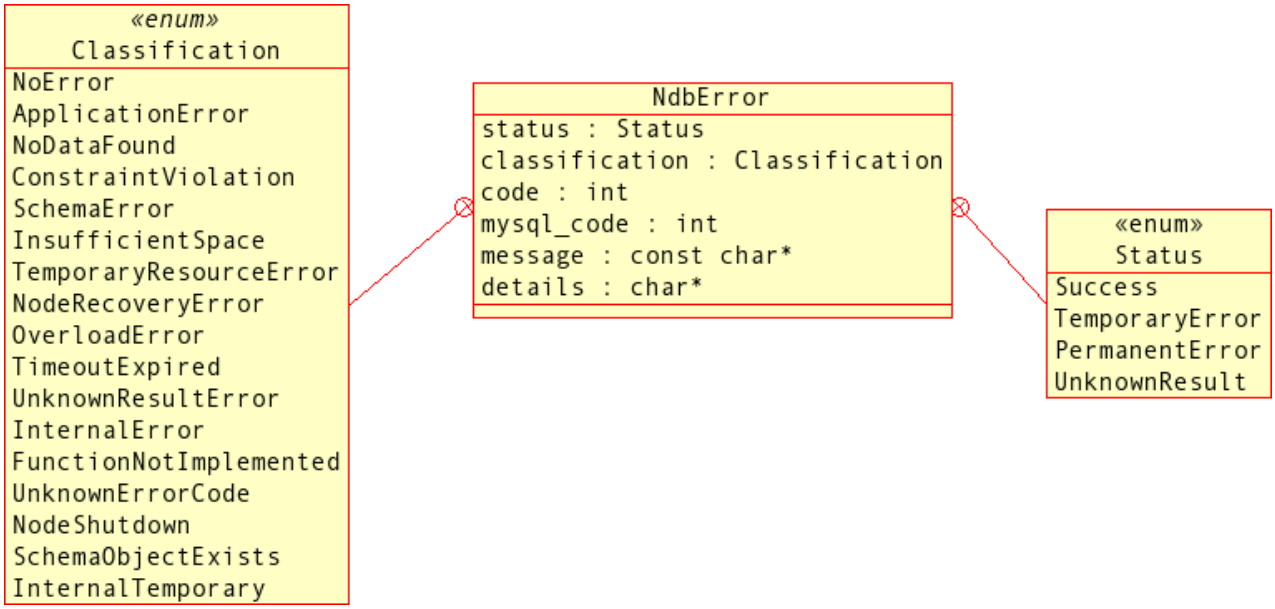
  The error detail message.

See the Description for more information about these members and their types.

Types. `NdbError` defines two datatypes:

- `Status`: The error status. See Section 2.3.30.1.1, "The `NdbError::Status` Type".
- `Classification`: The type of error or the logical grouping to which it belongs. See Section 2.3.30.1.2, "The `NdbError::Classification` Type".

Structure Diagram. This diagram shows all the available members and types of the `NdbError` structure:



2.3.30.1. `NdbError` Types

This section discusses the `Status` and `Classification` datatypes defined by `NdbError`.

2.3.30.1.1. The `NdbError::Status` Type

Description. This type is used to describe an error's status.

Enumeration values.

Value	Description
`Success`	No error has occurred
`TemporaryError`	A temporary and usually recoverable error; the application should retry the operation giving rise to the error
`PermanentError`	Permanent error; not recoverable
`UnknownResult`	The operation's result or status is unknown

> **Note**
>
> Related information specific to certain error conditions may be found in Section 4.2.3, “NDB Error Classifications”.

2.3.30.1.2. The `NdbError::Classification` Type

Description. This type describes the type of error, or the logical group to which it belongs.

Enumeration values.

Value	Description
`NoError`	Indicates success (no error occurred)
`ApplicationError`	An error occurred in an application program
`NoDataFound`	A read operation failed due to one or more missing records.
`ConstraintViolation`	A constraint violation occurred, such as attempting to insert a tuple having a primary key value already in use in the target table.
`SchemaError`	An error took place when trying to create or use a table.
`InsufficientSpace`	There was insufficient memory for data or indexes.
`TemporaryResourceError`	This type of error is typically encountered when there are too many active transactions.
`NodeRecoveryError`	This is a temporary failure which was likely caused by a node recovery in progress, some examples being when information sent between an application and `NDB` is lost, or when there is a distribution change.
`OverloadError`	This type of error is often caused when there is insufficient logfile space.
`TimeoutExpired`	A timeout, often caused by a deadlock.
`UnknownResultError`	It is not known whether a transaction was committed.
`InternalError`	A serious error has occurred in `NDB` itself.
`FunctionNotImplemented`	The application attempted to use a function which is not yet implemented.
`UnknownErrorCode`	This is seen where the `NDB` error handler cannot determine the correct error code to report.
`NodeShutdown`	This is caused by a node shutdown.
`SchemaObjectExists`	The application attempted to create a schema object that already exists.
`InternalTemporary`	A request was sent to a non-master node.

> **Note**
>
> Related information specific to certain error conditions may be found in Section 4.2.2, “NDB Error Codes and Messages”, and in Section 4.2.3, “NDB Error Classifications”.

2.3.31. The `PartitionSpec` Structure

This section describes the `PartitionSpec` structure.

Parent class. Ndb

Description. `PartitionSpec` is a structure available in MySQL Cluster NDB 6.3.24 and later, and used for describing a table

partition in terms of any one of the following:

- A specific partition ID for a table with user-defined partitioning.
- An array made up of a table's distribution key values for a table with native partitioning.
- (*MySQL Cluster NDB 7.0.4 and later*:) A row in `NdbRecord` format containing a natively partitioned table's distribution key values.

Attributes. A `PartitionSpec` has two attributes, a `SpecType` and a `Spec` which is a data structure corresponding to that `SpecType`, as shown in the following table:

`SpecType` Enumeration	**`SpecType` Value (`Uint32`)**	**Data Structure**	**Description**
`PS_NONE`	`0`	*none*	No partitioning information is provided.
`PS_USER_DEFINED`	`1`	`UserDefined`	For a table having user-defined partitioning, a specific partition is identified by its partition ID.
`PS_DISTR_KEY_PART_PTR`	`2`	`KeyPartPtr`	For a table having native partitioning, an array containing the table's distribution key values is used to identify the partition.
(*MySQL Cluster NDB 7.0.4 and later*:) `PS_DISTR_KEY_RECORD`	`3`	`KeyRecord`	The partition is identified using a natively partitioned table's distribution key values, as contained in a row given in `NdbRecord` format.

`UserDefined` structure. This structure is used when the `SpecType` is `PS_USER_DEFINED`.

Attribute	**Type**	**Description**
`partitionId`	`Uint32`	The partition ID for the desired table.

`KeyPartPtr` structure. This structure is used when the `SpecType` is `PS_DISTR_KEY_PART_PTR`.

Attribute	**Type**	**Description**
`tableKeyParts`	`const Key_part_ptr*` (see Section 2.3.29, "The `Key_part_ptr` Structure")	Pointer to the distribution key values for a table having native partitioning.
`xfrmbuf`	`void*`	Pointer to a temporary buffer used for performing calculations.
`xfrmbuflen`	`Uint32`	Length of the temporary buffer.

`KeyRecord` structure. (*MySQL Cluster NDB 7.0.4 and later*:) This structure is used when the `SpecType` is `PS_DISTR_KEY_RECORD`.

Attribute	**Type**	**Description**
`keyRecord`	`const NdbRecord*` (see Section 2.3.25, "The `NdbRecord` Interface")	A row in `NdbRecord` format, containing a table's distribution keys.
`keyRow`	`const char*`	The distribution key data.
`xfrmbuf`	`void*`	Pointer to a temporary buffer used for performing calculations.
`xfrmbuflen`	`Uint32`	Length of the temporary buffer.

Definition from `Ndb.hpp`. Because this is a fairly complex structure, we here provide the original source-code definition of `PartitionSpec`, as given in `storage/ndb/include/ndbapi/Ndb.hpp`:

```
struct PartitionSpec
{
  enum SpecType
```

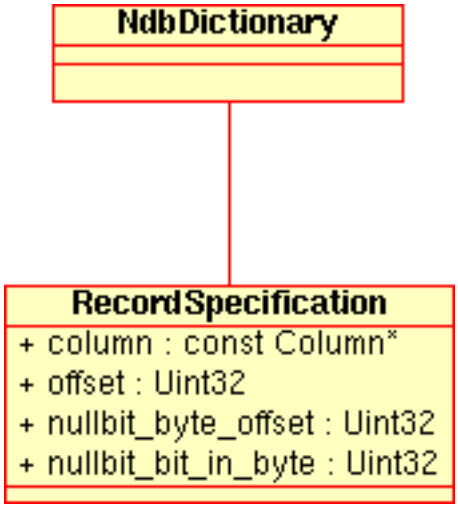
```
  {
    PS_NONE                = 0,
    PS_USER_DEFINED        = 1,
    PS_DISTR_KEY_PART_PTR  = 2

    /* MySQL Cluster NDB 7.0.4 and later: */

    PS_DISTR_KEY_RECORD    = 3
  };

  Uint32 type;

  union
  {
    struct {
      Uint32 partitionId;
    } UserDefined;

    struct {
      const Key_part_ptr* tableKeyParts;
      void* xfrmbuf;
      Uint32 xfrmbuflen;
    } KeyPartPtr;

    /* MySQL Cluster NDB 7.0.4 and later: */

    struct {
      const NdbRecord* keyRecord;
      const char* keyRow;
      void* xfrmbuf;
      Uint32 xfrmbuflen;
    } KeyRecord;
  };
};
```

2.3.32. The `RecordSpecification` Structure

Parent class. NdbDictionary

Description. This structure is used to specify columns and range offsets when creating `NdbRecord` objects.

Members. The makeup of this structure is shown here:



The individual members are described in more detail in the following table:

Name	Type	Description
`column`	`const Column *`	The column described by this entry (the column's maximum size defines the field size for the row). Even when creating an `NdbRecord` for an index, this must point to a column obtained from the underlying table, and not from the index itself.
`offset`	`Uint32`	The offset of data from the beginning of a row. For reading blobs, the blob handle (`NdbBlob*`), rather than the actual blob data, is written into the row. This means that there must be at least `sizeof(NdbBlob*)` must be available in the row.
`nullbit_byte_offset`	`Uint32`	The offset from the beginning of the row of the byte containing the `NULL` bit.
`nullbit_bit_in_byte`	`Uint32`	`NULL` bit (0-7).

Important

`nullbit_byte_offset` and `nullbit_bit_in_byte` are not used for non-`NULL`able columns.

For more information, see Section 2.3.25, "The `NdbRecord` Interface".

2.4. Practical Examples

This section provides code examples illustrating how to accomplish some basic tasks using the NDB API.

All of these examples can be compiled and run as provided, and produce sample output to demonstrate their effects.

2.4.1. Using Synchronous Transactions

This example illustrates the use of synchronous transactions in the NDB API.

The source code for this example can be found in `storage/ndb/ndbapi-examples/ndbapi_simple/ndbapi_simple.cpp` in the MySQL 5.1 tree.

The correct output from this program is as follows:

```
ATTR1  ATTR2
0      10
1       1
2      12
Detected that deleted tuple doesn't exist!
4      14
5       5
6      16
7       7
8      18
9       9
```

```
#include <mysql.h>
#include <NdbApi.hpp>
// Used for cout
#include <stdio.h>
#include <iostream>

static void run_application(MYSQL &, Ndb_cluster_connection &);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();

  // connect to mysql server and cluster and run application
  {
    char * mysqld_sock  = argv[1];
    const char *connectstring = argv[2];
    // Object representing the cluster
    Ndb_cluster_connection cluster_connection(connectstring);

    // Connect to cluster management server (ndb_mgmd)
    if (cluster_connection.connect(4 /* retries               */,
          5 /* delay between retries */,
          1 /* verbose               */))
    {
      std::cout << "Cluster management server was not ready within 30 secs.\n";
      exit(-1);
    }

    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster was not ready within 30 secs.\n";
      exit(-1);
    }

    // connect to mysql server
```

```
    MYSQL mysql;
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
          0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    // run the application code
    run_application(mysql, cluster_connection);
  }

  ndb_end(0);

  return 0;
}

static void create_table(MYSQL &);
static void drop_table(MYSQL &);
static void do_insert(Ndb &);
static void do_update(Ndb &);
static void do_delete(Ndb &);
static void do_read(Ndb &);

static void run_application(MYSQL &mysql,
         Ndb_cluster_connection &cluster_connection)
{
  /********************************************
   * Connect to database via mysql-c          *
   ********************************************/
  mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
  if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  /********************************************
   * Connect to database via NdbApi           *
   ********************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, "TEST_DB_1" );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(myNdb);
  do_update(myNdb);
  do_delete(myNdb);
  do_read(myNdb);
  drop_table(mysql);
  mysql_query(&mysql, "DROP DATABASE TEST_DB_1");
}

/*********************************************************
 * Create a table named MYTABLENAME if it does not exist *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "CREATE TABLE"
      "  MYTABLENAME"
      "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
      "     ATTR2 INT UNSIGNED NOT NULL)"
      "  ENGINE=NDB"))
    MYSQLERROR(mysql);
}

/*********************************
 * Drop a table named MYTABLENAME
 *********************************/
static void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "DROP TABLE"
      "  MYTABLENAME"))
    MYSQLERROR(mysql);
}

/***************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 ***************************************************************************/
static void do_insert(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
```

```
    myOperation->equal("ATTR1", i);
    myOperation->setValue("ATTR2", i);

    myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i+5);
    myOperation->setValue("ATTR2", i+5);

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*****************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *****************************************************************/
static void do_update(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->updateTuple();
    myOperation->equal( "ATTR1", i );
    myOperation->setValue( "ATTR2", i+10);

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*********************************************
 * Delete one tuple (the one with primary key 3) *
 *********************************************/
static void do_delete(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->deleteTuple();
  myOperation->equal( "ATTR1", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
}

/*****************************
 * Read and print all tuples *
 *****************************/
static void do_read(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());
```

```
    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    if (myTransaction->getNdbError().classification == NdbError::NoDataFound)
      if (i == 3)
        std::cout << "Detected that deleted tuple doesn't exist!" << std::endl;
      else
              APIERROR(myTransaction->getNdbError());

    if (i != 3) {
      printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb.closeTransaction(myTransaction);
  }
}
```

2.4.2. Using Synchronous Transactions and Multiple Clusters

This example demonstrates synchronous transactions and connecting to multiple clusters in a single NDB API application.

The source code for this program may be found in the MySQL 5.1 source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_simple_dual/ndbapi_simple_dual.cpp`.

```
/*
 *  ndbapi_simple_dual.cpp: Using synchronous transactions in NDB API
 *
 *  Correct output from this program is:
 *
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *
 */

#include <mysql.h>
#include <NdbApi.hpp>
// Used for cout
#include <stdio.h>
#include <iostream>

static void run_application(MYSQL &, Ndb_cluster_connection &, const char* table, const char* db);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 5)
  {
    std::cout << "Arguments are <socket mysqld1> <connect_string cluster 1> <socket mysqld2> <connect_string cluster 2>
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();
  {
    char * mysqld1_sock  = argv[1];
    const char *connectstring1 = argv[2];
    char * mysqld2_sock = argv[3];
    const char *connectstring2 = argv[4];

    // Object representing the cluster 1
    Ndb_cluster_connection cluster1_connection(connectstring1);
    MYSQL mysql1;
    // Object representing the cluster 2
```

```
    Ndb_cluster_connection cluster2_connection(connectstring2);
    MYSQL mysql2;

    // connect to mysql server and cluster 1 and run application
    // Connect to cluster 1  management server (ndb_mgmd)
    if (cluster1_connection.connect(4 /* retries               */,
            5 /* delay between retries */,
            1 /* verbose             */))
    {
      std::cout << "Cluster 1 management server was not ready within 30 secs.\n";
      exit(-1);
    }
    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster1_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster 1 was not ready within 30 secs.\n";
      exit(-1);
    }
    // connect to mysql server in cluster 1
    if ( !mysql_init(&mysql1) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql1, "localhost", "root", "", "",
              0, mysqld1_sock, 0) )
      MYSQLERROR(mysql1);


    // connect to mysql server and cluster 2 and run application

    // Connect to cluster management server (ndb_mgmd)
    if (cluster2_connection.connect(4 /* retries               */,
            5 /* delay between retries */,
            1 /* verbose             */))
    {
      std::cout << "Cluster 2 management server was not ready within 30 secs.\n";
      exit(-1);
    }
    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster2_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster 2 was not ready within 30 secs.\n";
      exit(-1);
    }
    // connect to mysql server in cluster 2
    if ( !mysql_init(&mysql2) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql2, "localhost", "root", "", "",
              0, mysqld2_sock, 0) )
      MYSQLERROR(mysql2);

    // run the application code
    run_application(mysql1, cluster1_connection, "MYTABLENAME1", "TEST_DB_1");
    run_application(mysql2, cluster2_connection, "MYTABLENAME2", "TEST_DB_2");
  }
  // Note: all connections must have been destroyed before calling ndb_end()
  ndb_end(0);

  return 0;
}

static void create_table(MYSQL &, const char* table);
static void drop_table(MYSQL &, const char* table);
static void do_insert(Ndb &, const char* table);
static void do_update(Ndb &, const char* table);
static void do_delete(Ndb &, const char* table);
static void do_read(Ndb &, const char* table);

static void run_application(MYSQL &mysql,
          Ndb_cluster_connection &cluster_connection,
          const char* table,
          const char* db)
{
  /****************************************
   * Connect to database via mysql-c          *
   ****************************************/
 char db_stmt[256];
 sprintf(db_stmt, "CREATE DATABASE %s\n", db);
  mysql_query(&mysql, db_stmt);
 sprintf(db_stmt, "USE %s", db);
  if (mysql_query(&mysql, db_stmt) != 0) MYSQLERROR(mysql);
  create_table(mysql, table);

  /****************************************
   * Connect to database via NdbApi           *
   ****************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, db );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(myNdb, table);
  do_update(myNdb, table);
```

```
  do_delete(myNdb, table);
  do_read(myNdb, table);
  /*
   * Drop the table
   */
  drop_table(mysql, table);
 sprintf(db_stmt, "DROP DATABASE %s\n", db);
  mysql_query(&mysql, db_stmt);
}

/********************************************************
 * Create a table named by table if it does not exist *
 ********************************************************/
static void create_table(MYSQL &mysql, const char* table)
{
  char create_stmt[256];

  sprintf(create_stmt, "CREATE TABLE %s \
             (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,\
              ATTR2 INT UNSIGNED NOT NULL)\
             ENGINE=NDB", table);
  if (mysql_query(&mysql, create_stmt))
    MYSQLERROR(mysql);
}

/*****************************
 * Drop a table named by table
 *****************************/
static void drop_table(MYSQL &mysql, const char* table)
{
  char drop_stmt[256];

  sprintf(drop_stmt, "DROP TABLE IF EXISTS %s", table);
  if (mysql_query(&mysql, drop_stmt))
    MYSQLERROR(mysql);
}

/**************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 **************************************************************************/
static void do_insert(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i);
    myOperation->setValue("ATTR2", i);

    myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i+5);
    myOperation->setValue("ATTR2", i+5);

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/**************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 **************************************************************/
static void do_update(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->updateTuple();
    myOperation->equal( "ATTR1", i );
    myOperation->setValue( "ATTR2", i+10);

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());
```

```
    myNdb.closeTransaction(myTransaction);
  }
}

/*************************************************
 * Delete one tuple (the one with primary key 3) *
 *************************************************/
static void do_delete(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->deleteTuple();
  myOperation->equal( "ATTR1", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
}

/****************************
 * Read and print all tuples *
 ****************************/
static void do_read(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      if (i == 3) {
  std::cout << "Detected that deleted tuple doesn't exist!" << std::endl;
      } else {
  APIERROR(myTransaction->getNdbError());
      }

    if (i != 3) {
      printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb.closeTransaction(myTransaction);
  }
}
```

2.4.3. Handling Errors and Retrying Transactions

This program demonstrates handling errors and retrying failed transactions using the NDB API.

The source code for this example can be found in `storage/ndb/ndbapi-examples/ndbapi_retries/ndbapi_retries.cpp` in the MySQL 5.1 tree.

There are many ways to program using the NDB API. In this example, we perform two inserts in the same transaction using `NdbConnection::execute(NoCommit)`.

In NDB API applications, there are two types of failures to be taken into account:

1. **Transaction failures**: If non-permanent, these can be handled by re-executing the transaction.
2. **Application errors**: These are indicated by `APIERROR`; they must be handled by the application programmer.

```
/*
 *  ndbapi_retries.cpp: Error handling and retrying transactions
 */
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <iostream>

// Used for sleep (use your own version of sleep)
#include <unistd.h>
#define TIME_TO_SLEEP_BETWEEN_TRANSACTION_RETRIES 1

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }

//
//  APIERROR prints an NdbError object
//
#define APIERROR(error) \
  { std::cout << "API ERROR: " << error.code << " " << error.message \
              << std::endl \
              << "           " << "Status: " << error.status \
              << ", Classification: " << error.classification << std::endl\
              << "           " << "File: " << __FILE__ \
              << " (Line: " << __LINE__ << ")" << std::endl \
              ; \
  }

//
//  TRANSERROR prints all error info regarding an NdbTransaction
//
#define TRANSERROR(ndbTransaction) \
  { NdbError error = ndbTransaction->getNdbError(); \
    std::cout << "TRANS ERROR: " << error.code << " " << error.message \
              << std::endl \
              << "           " << "Status: " << error.status \
              << ", Classification: " << error.classification << std::endl \
              << "           " << "File: " << __FILE__ \
              << " (Line: " << __LINE__ << ")" << std::endl \
              ; \
    printTransactionError(ndbTransaction); \
  }

void printTransactionError(NdbTransaction *ndbTransaction) {
  const NdbOperation *ndbOp = NULL;
  int i=0;

  /**************************************************************
   * Print NdbError object of every operations in the transaction *
   **************************************************************/
  while ((ndbOp = ndbTransaction->getNextCompletedOperation(ndbOp)) != NULL) {
    NdbError error = ndbOp->getNdbError();
    std::cout << "           OPERATION " << i+1 << ": "
        << error.code << " " << error.message << std::endl
        << "           Status: " << error.status
        << ", Classification: " << error.classification << std::endl;
    i++;
  }
}


//
//  Example insert
//  @param myNdb          Ndb object representing NDB Cluster
//  @param myTransaction  NdbTransaction used for transaction
//  @param myTable        Table to insert into
//  @param error          NdbError object returned in case of errors
//  @return -1 in case of failures, 0 otherwise
//
int insert(int transactionId, NdbTransaction* myTransaction,
     const NdbDictionary::Table *myTable) {
  NdbOperation   *myOperation;          // For other operations

  myOperation = myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) return -1;

  if (myOperation->insertTuple() ||
      myOperation->equal("ATTR1", transactionId) ||
      myOperation->setValue("ATTR2", transactionId)) {
    APIERROR(myOperation->getNdbError());
    exit(-1);
  }

  return myTransaction->execute(NdbTransaction::NoCommit);
}


//
//  Execute function which re-executes (tries 10 times) the transaction
//  if there are temporary errors (e.g. the NDB Cluster is overloaded).
//  @return -1 failure, 1 success
```

```
//
int executeInsertTransaction(int transactionId, Ndb* myNdb,
          const NdbDictionary::Table *myTable) {
  int result = 0;                          // No result yet
  int noOfRetriesLeft = 10;
  NdbTransaction   *myTransaction;         // For other transactions
  NdbError ndberror;

  while (noOfRetriesLeft > 0 && !result) {

    /********************************
     * Start and execute transaction *
     ********************************/
    myTransaction = myNdb->startTransaction();
    if (myTransaction == NULL) {
      APIERROR(myNdb->getNdbError());
      ndberror = myNdb->getNdbError();
      result = -1;  // Failure
    } else if (insert(transactionId, myTransaction, myTable) ||
        insert(10000+transactionId, myTransaction, myTable) ||
        myTransaction->execute(NdbTransaction::Commit)) {
      TRANSERROR(myTransaction);
      ndberror = myTransaction->getNdbError();
      result = -1;  // Failure
    } else {
      result = 1;   // Success
    }

    /********************************
     * If failure, then analyze error *
     ********************************/
    if (result == -1) {
      switch (ndberror.status) {
      case NdbError::Success:
  break;
      case NdbError::TemporaryError:
  std::cout << "Retrying transaction..." << std::endl;
  sleep(TIME_TO_SLEEP_BETWEEN_TRANSACTION_RETRIES);
  --noOfRetriesLeft;
  result = 0;   // No completed transaction yet
  break;

      case NdbError::UnknownResult:
      case NdbError::PermanentError:
  std::cout << "No retry of transaction..." << std::endl;
  result = -1;  // Permanent failure
  break;
      }
    }

    /*********************
     * Close transaction *
     *********************/
    if (myTransaction != NULL) {
      myNdb->closeTransaction(myTransaction);
    }
  }

  if (result != 1) exit(-1);
  return result;
}

/*********************************************************
 * Create a table named MYTABLENAME if it does not exist *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "CREATE TABLE"
      "  MYTABLENAME"
      "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
      "     ATTR2 INT UNSIGNED NOT NULL)"
      "  ENGINE=NDB"))
    MYSQLERROR(mysql);
}

/*********************************
 * Drop a table named MYTABLENAME
 *********************************/
static void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
      "DROP TABLE"
      "  MYTABLENAME"))
    MYSQLERROR(mysql);
}

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
```

```
  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connectstring); // Object representing the cluster

  int r= cluster_connection->connect(5 /* retries               */,
            3 /* delay between retries */,
            1 /* verbose               */);
  if (r > 0)
  {
    std::cout
      << "Cluster connect failed, possibly resolved with more retries.\n";
    exit(-1);
  }
  else if (r < 0)
  {
    std::cout
      << "Cluster connect failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs." << std::endl;
    exit(-1);
  }
  // connect to mysql server
  MYSQL mysql;
  if ( !mysql_init(&mysql) ) {
    std::cout << "mysql_init failed\n";
    exit(-1);
  }
  if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
        0, mysqld_sock, 0) )
    MYSQLERROR(mysql);

  /*****************************************
   * Connect to database via mysql-c          *
   *****************************************/
  mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
  if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  Ndb* myNdb= new Ndb( cluster_connection,
           "TEST_DB_1" );  // Object representing the database

  if (myNdb->init() == -1) {
    APIERROR(myNdb->getNdbError());
    exit(-1);
  }

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");
  if (myTable == NULL)
  {
    APIERROR(myDict->getNdbError());
    return -1;
  }
  /*********************************
   * Execute some insert transactions *
   *********************************/
  for (int i = 10000; i < 20000; i++) {
    executeInsertTransaction(i, myNdb, myTable);
  }

  delete myNdb;
  delete cluster_connection;

  drop_table(mysql);

  ndb_end(0);
  return 0;
}
```

2.4.4. Basic Scanning Example

This example illustrates how to use the NDB scanning API. It shows how to perform a scan, how to scan for an update, and how to scan for a delete, making use of the `NdbScanFilter` and `NdbScanOperation` classes.

(See Section 2.3.17, “The `NdbScanFilter` Class”, and Section 2.3.18, “The `NdbScanOperation` Class”.)

The source code for this example may found in MySQL 5.1 tree, in the file `storage/ndb/ndbapi-examples/ndbapi_scan/ndbapi_scan.cpp`.

This example makes use of the following classes and methods:

- `Ndb_cluster_connection`:
  - `connect()`

  - `wait_until_ready()`

  See Section 2.3.24, “The `Ndb_cluster_connection` Class”.

- `Ndb`:
  - `init()`
  - `getDictionary()`
  - `startTransaction()`
  - `closeTransaction()`

  See Section 2.3.8, “The `Ndb` Class”.

- `NdbTransaction`:
  - `getNdbScanOperation()`
  - `execute()`

  See Section 2.3.19, “The `NdbTransaction` Class”.

- `NdbOperation`:
  - `insertTuple()`
  - `equal()`
  - `setValue()`

  See Section 2.3.15, “The `NdbOperation` Class”.

- `NdbScanOperation`:
  - `getValue()`
  - `readTuples()`
  - `nextResult()`
  - `deleteCurrentTuple()`
  - `updateCurrentTuple()`

  See Section 2.3.18, “The `NdbScanOperation` Class”.

- `NdbDictionary`:
  - `Dictionary::getTable()`

    See Section 2.3.3, “The `Dictionary` Class”.
  - `Table::getColumn()`

    See Section 2.3.21, “The `Table` Class”.
  - `Column::getLength()`

    See Section 2.3.1, “The `Column` Class”.

- `NdbScanFilter`:
  - `begin()`
  - `eq()`
  - `end()`

  See Section 2.3.17, “The `NdbScanFilter` Class”.

```
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
// Used for cout
#include <iostream>
#include <stdio.h>

/**
 * Helper sleep function
 */
static void
milliSleep(int milliseconds){
  struct timeval sleeptime;
  sleeptime.tv_sec = milliseconds / 1000;
  sleeptime.tv_usec = (milliseconds - (sleeptime.tv_sec * 1000)) * 1000000;
  select(0, 0, 0, 0, &sleeptime);
}


/**
 * Helper sleep function
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

struct Car
{
  /**
   * Note memset, so that entire char-fields are cleared
   *   as all 20 bytes are significant (as type is char)
   */
  Car() { memset(this, 0, sizeof(* this)); }

  unsigned int reg_no;
  char brand[20];
  char color[20];
};

/**
 * Function to drop table
 */
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE GARAGE"))
    MYSQLERROR(mysql);
}


/**
 * Function to create table
 */
void create_table(MYSQL &mysql)
{
  while (mysql_query(&mysql,
      "CREATE TABLE"
      "  GARAGE"
      "    (REG_NO INT UNSIGNED NOT NULL,"
      "     BRAND CHAR(20) NOT NULL,"
      "     COLOR CHAR(20) NOT NULL,"
      "     PRIMARY KEY USING HASH (REG_NO))"
      "  ENGINE=NDB"))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "MySQL Cluster already has example table: GARAGE. "
        << "Dropping it..." << std::endl;
    /******************
     * Recreate table *
     ******************/
    drop_table(mysql);
    create_table(mysql);
  }
}

int populate(Ndb * myNdb)
{
  int i;
  Car cars[15];

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Five blue mercedes
   */
  for (i = 0; i < 5; i++)
```

```
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "Mercedes");
    sprintf(cars[i].color, "Blue");
  }

  /**
   * Five black bmw
   */
  for (i = 5; i < 10; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "BMW");
    sprintf(cars[i].color, "Black");
  }

  /**
   * Five pink toyotas
   */
  for (i = 10; i < 15; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "Toyota");
    sprintf(cars[i].color, "Pink");
  }

  NdbTransaction* myTrans = myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  for (i = 0; i < 15; i++)
  {
    NdbOperation* myNdbOperation = myTrans->getNdbOperation(myTable);
    if (myNdbOperation == NULL)
      APIERROR(myTrans->getNdbError());
    myNdbOperation->insertTuple();
    myNdbOperation->equal("REG_NO", cars[i].reg_no);
    myNdbOperation->setValue("BRAND", cars[i].brand);
    myNdbOperation->setValue("COLOR", cars[i].color);
  }

  int check = myTrans->execute(NdbTransaction::Commit);

  myTrans->close();

  return check != -1;
}

int scan_delete(Ndb* myNdb,
    int column,
    const char * color)

{

  // Scan all records exclusive and delete
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int deletedRows = 0;
  int check;
  NdbError             err;
  NdbTransaction  *myTrans;
  NdbScanOperation  *myScanOp;

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retyrMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {
    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
    << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
    if (myTrans == NULL)
    {
      const NdbError err = myNdb->getNdbError();

      if (err.status == NdbError::TemporaryError)
      {
  milliSleep(50);
  retryAttempt++;
```

```
continue;
    }
    std::cout <<  err.message << std::endl;
    return -1;
  }

 /**
  * Get a scan operation.
  */
  myScanOp = myTrans->getNdbScanOperation(myTable);
  if (myScanOp == NULL)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Define a result set for the scan.
   */
  if(myScanOp->readTuples(NdbOperation::LM_Exclusive) != 0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Use NdbScanFilter to define a search critera
   */
  NdbScanFilter filter(myScanOp) ;
  if(filter.begin(NdbScanFilter::AND) < 0  ||
     filter.cmp(NdbScanFilter::COND_EQ, column, color) < 0 ||
     filter.end() < 0)
  {
    std::cout <<  myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Start scan    (NoCommit since we are only reading at this stage);
   */
  if(myTrans->execute(NdbTransaction::NoCommit) != 0){
    err = myTrans->getNdbError();
    if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
milliSleep(50);
continue;
    }
    std::cout << err.code << std::endl;
    std::cout << myTrans->getNdbError().code << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }


 /**
  * start of loop: nextResult(true) means that "parallelism" number of
  * rows are fetched from NDB and cached in NDBAPI
  */
  while((check = myScanOp->nextResult(true)) == 0){
    do
    {
if (myScanOp->deleteCurrentTuple() != 0)
{
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  return -1;
}
deletedRows++;

/**
 * nextResult(false) means that the records
 * cached in the NDBAPI are modified before
 * fetching more rows from NDB.
 */
    } while((check = myScanOp->nextResult(false)) == 0);

    /**
     * Commit when all cached tuple have been marked for deletion
     */
    if(check != -1)
    {
check = myTrans->execute(NdbTransaction::Commit);
    }

    if(check == -1)
    {
/**
 * Create a new transaction, while keeping scan open
 */
check = myTrans->restart();
    }

    /**
```

```
       * Check for errors
       */
      err = myTrans->getNdbError();
      if(check == -1)
      {
  if(err.status == NdbError::TemporaryError)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    milliSleep(50);
    continue;
  }
      }
      /**
       * End of loop
       */
    }
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return 0;
  }

  if(myTrans!=0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
  }
  return -1;
}


int scan_update(Ndb* myNdb,
    int update_column,
    const char * before_color,
    const char * after_color)

{

  // Scan all records exclusive and update
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int updatedRows = 0;
  int check;
  NdbError              err;
  NdbTransaction  *myTrans;
  NdbScanOperation  *myScanOp;

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retyrMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {

    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
    << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
    if (myTrans == NULL)
    {
      const NdbError err = myNdb->getNdbError();

      if (err.status == NdbError::TemporaryError)
      {
  milliSleep(50);
  retryAttempt++;
  continue;
      }
      std::cout <<  err.message << std::endl;
      return -1;
    }

   /**
    * Get a scan operation.
    */
    myScanOp = myTrans->getNdbScanOperation(myTable);
    if (myScanOp == NULL)
    {
      std::cout << myTrans->getNdbError().message << std::endl;
      myNdb->closeTransaction(myTrans);
      return -1;
```

```
  }

  /**
   * Define a result set for the scan.
   */
  if( myScanOp->readTuples(NdbOperation::LM_Exclusive) )
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Use NdbScanFilter to define a search critera
   */
  NdbScanFilter filter(myScanOp) ;
  if(filter.begin(NdbScanFilter::AND) < 0  ||
     filter.cmp(NdbScanFilter::COND_EQ, update_column, before_color) <0||
     filter.end() <0)
  {
    std::cout <<  myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * Start scan    (NoCommit since we are only reading at this stage);
   */
  if(myTrans->execute(NdbTransaction::NoCommit) != 0)
  {
    err = myTrans->getNdbError();
    if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
milliSleep(50);
continue;
    }
    std::cout << myTrans->getNdbError().code << std::endl;
    myNdb->closeTransaction(myTrans);
    return -1;
  }

  /**
   * start of loop: nextResult(true) means that "parallelism" number of
   * rows are fetched from NDB and cached in NDBAPI
   */
  while((check = myScanOp->nextResult(true)) == 0){
    do {
/**
 * Get update operation
 */
NdbOperation * myUpdateOp = myScanOp->updateCurrentTuple();
if (myUpdateOp == 0)
{
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  return -1;
}
updatedRows++;

/**
 * do the update
 */
myUpdateOp->setValue(update_column, after_color);
/**
 * nextResult(false) means that the records
 * cached in the NDBAPI are modified before
 * fetching more rows from NDB.
 */
    } while((check = myScanOp->nextResult(false)) == 0);

    /**
     * NoCommit when all cached tuple have been updated
     */
    if(check != -1)
    {
check = myTrans->execute(NdbTransaction::NoCommit);
    }

    /**
     * Check for errors
     */
    err = myTrans->getNdbError();
    if(check == -1)
    {
if(err.status == NdbError::TemporaryError){
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  milliSleep(50);
  continue;
}
    }
    /**
     * End of loop
     */
  }
```

```
    /**
     * Commit all prepared operations
     */
    if(myTrans->execute(NdbTransaction::Commit) == -1)
    {
      if(err.status == NdbError::TemporaryError){
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  milliSleep(50);
  continue;
      }
    }

    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return 0;
  }


  if(myTrans!=0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
  }
  return -1;
}



int scan_print(Ndb * myNdb)
{
// Scan all records exclusive and update
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int fetchedRows = 0;
  int check;
  NdbError             err;
  NdbTransaction  *myTrans;
  NdbScanOperation  *myScanOp;
  /* Result of reading attribute value, three columns:
     REG_NO, BRAND, and COLOR
   */
  NdbRecAttr *      myRecAttr[3];

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("GARAGE");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retyrMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {

    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
    << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
    if (myTrans == NULL)
    {
      const NdbError err = myNdb->getNdbError();

      if (err.status == NdbError::TemporaryError)
      {
  milliSleep(50);
  retryAttempt++;
  continue;
      }
     std::cout << err.message << std::endl;
      return -1;
    }
    /*
     * Define a scan operation.
     * NDBAPI.
     */
    myScanOp = myTrans->getNdbScanOperation(myTable);
    if (myScanOp == NULL)
    {
      std::cout << myTrans->getNdbError().message << std::endl;
      myNdb->closeTransaction(myTrans);
      return -1;
    }

    /**
```

```
    * Read without locks, without being placed in lock queue
    */
   if( myScanOp->readTuples(NdbOperation::LM_CommittedRead) == -1)
   {
     std::cout << myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Define storage for fetched attributes.
    * E.g., the resulting attributes of executing
    * myOp->getValue("REG_NO") is placed in myRecAttr[0].
    * No data exists in myRecAttr until transaction has commited!
    */
   myRecAttr[0] = myScanOp->getValue("REG_NO");
   myRecAttr[1] = myScanOp->getValue("BRAND");
   myRecAttr[2] = myScanOp->getValue("COLOR");
   if(myRecAttr[0] ==NULL || myRecAttr[1] == NULL || myRecAttr[2]==NULL)
   {
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  return -1;
   }
   /**
    * Start scan   (NoCommit since we are only reading at this stage);
    */
   if(myTrans->execute(NdbTransaction::NoCommit) != 0){
     err = myTrans->getNdbError();
     if(err.status == NdbError::TemporaryError){
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  milliSleep(50);
  continue;
     }
     std::cout << err.code << std::endl;
     std::cout << myTrans->getNdbError().code << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * start of loop: nextResult(true) means that "parallelism" number of
    * rows are fetched from NDB and cached in NDBAPI
    */
   while((check = myScanOp->nextResult(true)) == 0){
     do {

  fetchedRows++;
  /**
   * print  REG_NO unsigned int
   */
  std::cout << myRecAttr[0]->u_32_value() << "\t";

  /**
   * print  BRAND character string
   */
  std::cout << myRecAttr[1]->aRef() << "\t";

  /**
   * print  COLOR character string
   */
  std::cout << myRecAttr[2]->aRef() << std::endl;

  /**
   * nextResult(false) means that the records
   * cached in the NDBAPI are modified before
   * fetching more rows from NDB.
   */
     } while((check = myScanOp->nextResult(false)) == 0);

   }
   myNdb->closeTransaction(myTrans);
   return 1;
  }
  return -1;

}


int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /****************************************************************
   * Connect to mysql server and create table                     *
   ****************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
```

```
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
          0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE TEST_DB");
    if (mysql_query(&mysql, "USE TEST_DB") != 0) MYSQLERROR(mysql);

    create_table(mysql);
  }

  /*************************************************************
   * Connect to ndb cluster                                    *
   *************************************************************/

  Ndb_cluster_connection cluster_connection(connectstring);
  if (cluster_connection.connect(4, 5, 1))
  {
    std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
    exit(-1);
  }
  // Optionally connect and wait for the storage nodes (ndbd's)
  if (cluster_connection.wait_until_ready(30,0) < 0)
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb myNdb(&cluster_connection,"TEST_DB");
  if (myNdb.init(1024) == -1) {      // Set max 1024  parallel transactions
    APIERROR(myNdb.getNdbError());
    exit(-1);
  }

  /****************************************
   * Check table definition                 *
   ****************************************/
  int column_color;
  {
    const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
    const NdbDictionary::Table *t= myDict->getTable("GARAGE");

    Car car;
    if (t->getColumn("COLOR")->getLength() != sizeof(car.color) ||
  t->getColumn("BRAND")->getLength() != sizeof(car.brand))
    {
      std::cout << "Wrong table definition" << std::endl;
      exit(-1);
    }
    column_color= t->getColumn("COLOR")->getColumnNo();
  }

  if(populate(&myNdb) > 0)
    std::cout << "populate: Success!" << std::endl;

  if(scan_print(&myNdb) > 0)
    std::cout << "scan_print: Success!" << std::endl  << std::endl;

  std::cout << "Going to delete all pink cars!" << std::endl;

  {
    /**
     * Note! color needs to be of exact the same size as column defined
     */
    Car tmp;
    sprintf(tmp.color, "Pink");
    if(scan_delete(&myNdb, column_color, tmp.color) > 0)
      std::cout << "scan_delete: Success!" << std::endl  << std::endl;
  }

  if(scan_print(&myNdb) > 0)
    std::cout << "scan_print: Success!" << std::endl  << std::endl;

  {
    /**
     * Note! color1 & 2 need to be of exact the same size as column defined
     */
    Car tmp1, tmp2;
    sprintf(tmp1.color, "Blue");
    sprintf(tmp2.color, "Black");
    std::cout << "Going to update all " << tmp1.color
        << " cars to " << tmp2.color << " cars!" << std::endl;
    if(scan_update(&myNdb, column_color, tmp1.color, tmp2.color) > 0)
      std::cout << "scan_update: Success!" << std::endl  << std::endl;
  }
  if(scan_print(&myNdb) > 0)
    std::cout << "scan_print: Success!" << std::endl  << std::endl;

  /**
   * Drop table
   */
  drop_table(mysql);

  return 0;
}
```

2.4.5. Using Secondary Indexes in Scans

This program illustrates how to use secondary indexes in the NDB API.

The source code for this example may be found in the MySQL 5.1 source tree, in `storage/ndb/ndbapi-examples/ndbapi_simple_index/ndbapi_simple_index.cpp`.

The correct output from this program is shown here:

```
ATTR1 ATTR2
0      10
1       1
2      12
Detected that deleted tuple doesn't exist!
4      14
5       5
6      16
7       7
8      18
9       9
```

```
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 3)
    {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /******************************************************************
   * Connect to mysql server and create table                       *
   ******************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
          0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
    if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);

    if (mysql_query(&mysql,
        "CREATE TABLE"
        "  MYTABLENAME"
        "    (ATTR1 INT UNSIGNED,"
        "     ATTR2 INT UNSIGNED NOT NULL,"
        "     PRIMARY KEY USING HASH (ATTR1),"
        "     UNIQUE MYINDEXNAME USING HASH (ATTR2))"
        "  ENGINE=NDB"))
      MYSQLERROR(mysql);
  }

  /******************************************************************
   * Connect to ndb cluster                                         *
   ******************************************************************/

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connectstring); // Object representing the cluster

  if (cluster_connection->connect(5,3,1))
  {
    std::cout << "Connect to cluster management server failed.\n";
    exit(-1);
  }
```

```
if (cluster_connection->wait_until_ready(30,30))
{
  std::cout << "Cluster was not ready within 30 secs.\n";
  exit(-1);
}

Ndb* myNdb = new Ndb( cluster_connection,
    "TEST_DB_1" );  // Object representing the database
if (myNdb->init() == -1) {
  APIERROR(myNdb->getNdbError());
  exit(-1);
}

const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");
if (myTable == NULL)
  APIERROR(myDict->getNdbError());
const NdbDictionary::Index *myIndex= myDict->getIndex("MYINDEXNAME$unique","MYTABLENAME");
if (myIndex == NULL)
  APIERROR(myDict->getNdbError());

/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
for (int i = 0; i < 5; i++) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->insertTuple();
  myOperation->equal("ATTR1", i);
  myOperation->setValue("ATTR2", i);

  myOperation = myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->insertTuple();
  myOperation->equal("ATTR1", i+5);
  myOperation->setValue("ATTR2", i+5);

  if (myTransaction->execute( NdbTransaction::Commit ) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/****************************************
 * Read and print all tuples using index *
 ****************************************/
std::cout << "ATTR1 ATTR2" << std::endl;

for (int i = 0; i < 10; i++) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->readTuple(NdbOperation::LM_Read);
  myIndexOperation->equal("ATTR2", i);

  NdbRecAttr *myRecAttr= myIndexOperation->getValue("ATTR1", NULL);
  if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

  if(myTransaction->execute( NdbTransaction::Commit ) != -1)
    printf(" %2d    %2d\n", myRecAttr->u_32_value(), i);

  myNdb->closeTransaction(myTransaction);
}

/*****************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *****************************************************************/
for (int i = 0; i < 10; i+=2) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->updateTuple();
  myIndexOperation->equal( "ATTR2", i );
  myIndexOperation->setValue( "ATTR2", i+10);

  if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/*********************************************
 * Delete one tuple (the one with primary key 3) *
```

```
   ************************************************/
  {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

    NdbIndexOperation *myIndexOperation=
      myTransaction->getNdbIndexOperation(myIndex);
    if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

    myIndexOperation->deleteTuple();
    myIndexOperation->equal( "ATTR2", 3 );

    if (myTransaction->execute(NdbTransaction::Commit) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb->closeTransaction(myTransaction);
  }

  /*****************************
   * Read and print all tuples *
   *****************************/
  {
    std::cout << "ATTR1 ATTR2" << std::endl;

    for (int i = 0; i < 10; i++) {
      NdbTransaction *myTransaction= myNdb->startTransaction();
      if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

      NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
      if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

      myOperation->readTuple(NdbOperation::LM_Read);
      myOperation->equal("ATTR1", i);

      NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
      if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

      if(myTransaction->execute( NdbTransaction::Commit ) == -1)
  if (i == 3) {
    std::cout << "Detected that deleted tuple doesn't exist!\n";
  } else {
    APIERROR(myTransaction->getNdbError());
  }

      if (i != 3) {
  printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
      }
      myNdb->closeTransaction(myTransaction);
    }
  }

  /**************
   * Drop table *
   **************/
  if (mysql_query(&mysql, "DROP TABLE MYTABLENAME"))
    MYSQLERROR(mysql);

  delete myNdb;
  delete cluster_connection;

  ndb_end(0);
  return 0;
}
```

2.4.6. Using `NdbRecord` with Hash Indexes

This program illustrates how to use secondary indexes in the NDB API with the aid of the NdbRecord interface introduced in MySQL-5.1.18-6.2.3..

The source code for this example may be found in the `mysql-5.1-telco` and `mysql-5.1-telco-6.2` source trees, in the file `storage/ndb/ndbapi-examples/ndbapi_s_i_ndbrecord/main.cpp`.

When run on a cluster having 2 data nodes, the correct output from this program is as shown here:

```
ATTR1 ATTR2
  0     0   (frag=0)
  1     1   (frag=1)
  2     2   (frag=1)
  3     3   (frag=0)
  4     4   (frag=1)
  5     5   (frag=1)
  6     6   (frag=0)
  7     7   (frag=0)
  8     8   (frag=1)
  9     9   (frag=0)
ATTR1 ATTR2
  0    10
  1     1
  2    12
Detected that deleted tuple doesn't exist!
  4    14
```

```
  5     5
  6    16
  7     7
  8    18
  9     9
```

```
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(1); }

int main(int argc, char** argv)
{
  if (argc != 3)
    {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /****************************************************************
   * Connect to mysql server and create table                     *
   ****************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
                             0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
    if (mysql_query(&mysql, "USE TEST_DB_1") != 0)
      MYSQLERROR(mysql);

    mysql_query(&mysql, "DROP TABLE MYTABLENAME");
    if (mysql_query(&mysql,
                    "CREATE TABLE"
                    "  MYTABLENAME"
                    "    (ATTR1 INT UNSIGNED,"
                    "     ATTR2 INT UNSIGNED NOT NULL,"
                    "     PRIMARY KEY USING HASH (ATTR1),"
                    "     UNIQUE MYINDEXNAME USING HASH (ATTR2))"
                    "  ENGINE=NDB"))
      MYSQLERROR(mysql);
  }

  /****************************************************************
   * Connect to ndb cluster                                       *
   ****************************************************************/

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connectstring); // Object representing the cluster

  if (cluster_connection->connect(5,3,1))
  {
    std::cout << "Connect to cluster management server failed.\n";
    exit(1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(1);
  }

  Ndb* myNdb = new Ndb( cluster_connection,
                        "TEST_DB_1" );  // Object representing the database
  if (myNdb->init() == -1) {
    APIERROR(myNdb->getNdbError());
    exit(1);
  }

  NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("MYTABLENAME");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());
  const NdbDictionary::Index *myIndex= myDict->getIndex("MYINDEXNAME$unique","MYTABLENAME");
```

```
  if (myIndex == NULL)
    APIERROR(myDict->getNdbError());

  /* Create NdbRecord descriptors. */
  const NdbDictionary::Column *col1= myTable->getColumn("ATTR1");
  if (col1 == NULL)
    APIERROR(myDict->getNdbError());
  const NdbDictionary::Column *col2= myTable->getColumn("ATTR2");
  if (col2 == NULL)
    APIERROR(myDict->getNdbError());

  /* NdbRecord for primary key lookup. */
  NdbDictionary::RecordSpecification spec[2];
  spec[0].column= col1;
  spec[0].offset= 0;
  spec[0].nullbit_byte_offset= 0;
  spec[0].nullbit_bit_in_byte= 0;
  const NdbRecord *pk_record=
    myDict->createRecord(myTable, spec, 1, sizeof(spec[0]));
  if (pk_record == NULL)
    APIERROR(myDict->getNdbError());

  /* NdbRecord for all table attributes (insert/read). */
  spec[0].column= col1;
  spec[0].offset= 0;
  spec[0].nullbit_byte_offset= 0;
  spec[0].nullbit_bit_in_byte= 0;
  spec[1].column= col2;
  spec[1].offset= 4;
  spec[1].nullbit_byte_offset= 0;
  spec[1].nullbit_bit_in_byte= 0;
  const NdbRecord *attr_record=
    myDict->createRecord(myTable, spec, 2, sizeof(spec[0]));
  if (attr_record == NULL)
    APIERROR(myDict->getNdbError());

  /* NdbRecord for unique key lookup. */
  spec[0].column= col2;
  spec[0].offset= 4;
  spec[0].nullbit_byte_offset= 0;
  spec[0].nullbit_bit_in_byte= 0;
  const NdbRecord *key_record=
    myDict->createRecord(myIndex, spec, 1, sizeof(spec[0]));
  if (key_record == NULL)
    APIERROR(myDict->getNdbError());
  char row[2][8];

  /*************************************************************************
   * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
   *************************************************************************/
  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

    /*
      Fill in rows with data. We need two rows, as the data must remain valid
      until NdbTransaction::execute() returns.
    */
    memcpy(&row[0][0], &i, 4);
    memcpy(&row[0][4], &i, 4);
    int value= i+5;
    memcpy(&row[1][0], &value, 4);
    memcpy(&row[1][4], &value, 4);

    NdbOperation *myOperation=
      myTransaction->insertTuple(attr_record, &row[0][0]);
    if (myOperation == NULL)
      APIERROR(myTransaction->getNdbError());
    myOperation=
      myTransaction->insertTuple(attr_record, &row[1][0]);
    if (myOperation == NULL)
      APIERROR(myTransaction->getNdbError());

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb->closeTransaction(myTransaction);
  }

  /****************************************
   * Read and print all tuples using index *
   ****************************************/
  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL)
      APIERROR(myNdb->getNdbError());

    memcpy(&row[0][4], &i, 4);
    unsigned char mask[1]= { 0x01 };            // Only read ATTR1
    NdbOperation *myOperation=
      myTransaction->readTuple(key_record, &row[0][0],
                               attr_record, &row[1][0],
                               NdbOperation::LM_Read, mask);
    if (myOperation == NULL)
      APIERROR(myTransaction->getNdbError());
```

```
  /* Demonstrate the posibility to use getValue() for the odd extra read. */
  Uint32 frag;
  if (myOperation->getValue(NdbDictionary::Column::FRAGMENT,
                            (char *)(&frag)) == 0)
    APIERROR(myOperation->getNdbError());

  if (myTransaction->execute( NdbTransaction::Commit,
                              NdbOperation::AbortOnError ) != -1)
  {
    int value;
    memcpy(&value, &row[1][0], 4);
    printf(" %2d    %2d   (frag=%u)\n", value, i, frag);
  }

  myNdb->closeTransaction(myTransaction);
}

/*****************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *****************************************************************/
for (int i = 0; i < 10; i+=2) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL)
    APIERROR(myNdb->getNdbError());

  memcpy(&row[0][4], &i, 4);
  int value= i+10;
  memcpy(&row[1][4], &value, 4);
  unsigned char mask[1]= { 0x02 };            // Only update ATTR2
  NdbOperation *myOperation=
    myTransaction->updateTuple(key_record, &row[0][0],
                               attr_record, &row[1][0], mask);
  if (myOperation == NULL)
    APIERROR(myTransaction->getNdbError());

  if ( myTransaction->execute( NdbTransaction::Commit ) == -1 )
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/************************************************
 * Delete one tuple (the one with unique key 3) *
 ************************************************/
{
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL)
    APIERROR(myNdb->getNdbError());

  int value= 3;
  memcpy(&row[0][4], &value, 4);
  NdbOperation *myOperation=
    myTransaction->deleteTuple(key_record, &row[0][0]);
  if (myOperation == NULL)
    APIERROR(myTransaction->getNdbError());

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/*****************************
 * Read and print all tuples *
 *****************************/
{
  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL)
      APIERROR(myNdb->getNdbError());

    memcpy(&row[0][0], &i, 4);
    NdbOperation *myOperation=
      myTransaction->readTuple(pk_record, &row[0][0],
                               attr_record, &row[1][0]);
    if (myOperation == NULL)
      APIERROR(myTransaction->getNdbError());

    if (myTransaction->execute( NdbTransaction::Commit,
                                NdbOperation::AbortOnError ) == -1)
      if (i == 3) {
        std::cout << "Detected that deleted tuple doesn't exist!\n";
      } else {
        APIERROR(myTransaction->getNdbError());
      }

    if (i != 3) {
      int value1, value2;
      memcpy(&value1, &row[1][0], 4);
      memcpy(&value2, &row[1][4], 4);
      printf(" %2d    %2d\n", value1, value2);
    }
    myNdb->closeTransaction(myTransaction);
  }
```

```
  }

  /**************
   * Drop table *
   **************/
  if (mysql_query(&mysql, "DROP TABLE MYTABLENAME"))
    MYSQLERROR(mysql);

  delete myNdb;
  delete cluster_connection;

  ndb_end(0);
  return 0;
}
```

2.4.7. NDB API Event Handling Example

This example demonstrates NDB API event handling.

The source code for this program may be found in the MySQL 5.1 source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_event/ndbapi_event.cpp`.

```
/*
 *  ndbapi_event.cpp: Illustrates event handling in the NDB API.
 */
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>
#include <unistd.h>
#ifdef VM_TRACE
#include <my_global.h>
#endif
#ifndef assert
#include <assert.h>
#endif


/**
 * Assume that there is a table which is being updated by
 * another process (e.g. flexBench -l 0 -stdtables).
 * We want to monitor what happens with column values.
 *
 * Or using the mysql client:
 *
 * shell> mysql -u root
 * mysql> create database TEST_DB;
 * mysql> use TEST_DB;
 * mysql> create table t0
 *        (c0 int, c1 int, c2 char(4), c3 char(4), c4 text,
 *        primary key(c0, c2)) engine ndb charset latin1;
 *
 * In another window start ndbapi_event, wait until properly started

   insert into t0 values (1, 2, 'a', 'b', null);
   insert into t0 values (3, 4, 'c', 'd', null);
   update t0 set c3 = 'e' where c0 = 1 and c2 = 'a'; -- use pk
   update t0 set c3 = 'f'; -- use scan
   update t0 set c3 = 'F'; -- use scan update to 'same'
   update t0 set c2 = 'g' where c0 = 1; -- update pk part
   update t0 set c2 = 'G' where c0 = 1; -- update pk part to 'same'
   update t0 set c0 = 5, c2 = 'H' where c0 = 3; -- update full PK
   delete from t0;

   insert ...; update ...; -- see events w/ same pk merged (if -m option)
   delete ...; insert ...; -- there are 5 combinations ID IU DI UD UU
   update ...; update ...;

   -- text requires -m flag
   set @a = repeat('a',256); -- inline size
   set @b = repeat('b',2000); -- part size
   set @c = repeat('c',2000*30); -- 30 parts

   -- update the text field using combinations of @a, @b, @c ...

 * you should see the data popping up in the example window
 *
 */

#define APIERROR(error) \
  { std::cout << "Error in " << __FILE__ << ", line:" << __LINE__ << ", code:" \
              << error.code << ", msg: " << error.message << "." << std::endl; \
    exit(-1); }

int myCreateEvent(Ndb* myNdb,
                  const char *eventName,
                  const char *eventTableName,
                  const char **eventColumnName,
                  const int noEventColumnName,
                  bool merge_events);
```

```
int main(int argc, char** argv)
{
  if (argc < 3)
  {
    std::cout << "Arguments are <connect_string cluster> <timeout> [m(merge events)|d(debug)].\n";
    exit(-1);
  }
  const char *connectstring = argv[1];
  int timeout = atoi(argv[2]);
  ndb_init();
  bool merge_events = argc > 3 && strchr(argv[3], 'm') != 0;
#ifdef VM_TRACE
  bool dbug = argc > 3 && strchr(argv[3], 'd') != 0;
  if (dbug) DBUG_PUSH("d:t:");
  if (dbug) putenv("API_SIGNAL_LOG=-");
#endif

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connectstring); // Object representing the cluster

  int r= cluster_connection->connect(5 /* retries               */,
                                     3 /* delay between retries */,
                                     1 /* verbose               */);
  if (r > 0)
  {
    std::cout
      << "Cluster connect failed, possibly resolved with more retries.\n";
    exit(-1);
  }
  else if (r < 0)
  {
    std::cout
      << "Cluster connect failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs." << std::endl;
    exit(-1);
  }

  Ndb* myNdb= new Ndb(cluster_connection,
          "TEST_DB");  // Object representing the database

  if (myNdb->init() == -1) APIERROR(myNdb->getNdbError());

  const char *eventName= "CHNG_IN_t0";
  const char *eventTableName= "t0";
  const int noEventColumnName= 5;
  const char *eventColumnName[noEventColumnName]=
    {"c0",
     "c1",
     "c2",
     "c3",
     "c4"
    };

  // Create events
  myCreateEvent(myNdb,
    eventName,
    eventTableName,
    eventColumnName,
    noEventColumnName,
                merge_events);

  // Normal values and blobs are unfortunately handled differently..
  typedef union { NdbRecAttr* ra; NdbBlob* bh; } RA_BH;

  int i, j, k, l;
  j = 0;
  while (j < timeout) {

    // Start "transaction" for handling events
    NdbEventOperation* op;
    printf("create EventOperation\n");
    if ((op = myNdb->createEventOperation(eventName)) == NULL)
      APIERROR(myNdb->getNdbError());
    op->mergeEvents(merge_events);

    printf("get values\n");
    RA_BH recAttr[noEventColumnName];
    RA_BH recAttrPre[noEventColumnName];
    // primary keys should always be a part of the result
    for (i = 0; i < noEventColumnName; i++) {
      if (i < 4) {
        recAttr[i].ra    = op->getValue(eventColumnName[i]);
        recAttrPre[i].ra = op->getPreValue(eventColumnName[i]);
      } else if (merge_events) {
        recAttr[i].bh    = op->getBlobHandle(eventColumnName[i]);
        recAttrPre[i].bh = op->getPreBlobHandle(eventColumnName[i]);
      }
    }

    // set up the callbacks
    printf("execute\n");
    // This starts changes to "start flowing"
```

```
  if (op->execute())
    APIERROR(op->getNdbError());

  NdbEventOperation* the_op = op;

  i= 0;
  while (i < timeout) {
    // printf("now waiting for event...\n");
    int r = myNdb->pollEvents(1000); // wait for event or 1000 ms
    if (r > 0) {
// printf("got data! %d\n", r);
while ((op= myNdb->nextEvent())) {
      assert(the_op == op);
  i++;
  switch (op->getEventType()) {
  case NdbDictionary::Event::TE_INSERT:
    printf("%u INSERT", i);
    break;
  case NdbDictionary::Event::TE_DELETE:
    printf("%u DELETE", i);
    break;
  case NdbDictionary::Event::TE_UPDATE:
    printf("%u UPDATE", i);
    break;
  default:
    abort(); // should not happen
  }
       printf(" gci=%d\n", (int)op->getGCI());
       for (k = 0; k <= 1; k++) {
         printf(k == 0 ? "post: " : "pre : ");
         for (l = 0; l < noEventColumnName; l++) {
           if (l < 4) {
             NdbRecAttr* ra = k == 0 ? recAttr[l].ra : recAttrPre[l].ra;
             if (ra->isNULL() >= 0) { // we have a value
               if (ra->isNULL() == 0) { // we have a non-null value
                 if (l < 2)
                   printf("%-5u", ra->u_32_value());
                 else
                   printf("%-5.4s", ra->aRef());
               } else
                 printf("%-5s", "NULL");
             } else
               printf("%-5s", "-"); // no value
           } else if (merge_events) {
             int isNull;
             NdbBlob* bh = k == 0 ? recAttr[l].bh : recAttrPre[l].bh;
             bh->getDefined(isNull);
             if (isNull >= 0) { // we have a value
               if (! isNull) { // we have a non-null value
                 Uint64 length = 0;
                 bh->getLength(length);
                 // read into buffer
                 unsigned char* buf = new unsigned char [length];
                 memset(buf, 'X', length);
                 Uint32 n = length;
                 bh->readData(buf, n); // n is in/out
                 assert(n == length);
                 // pretty-print
                 bool first = true;
                 Uint32 i = 0;
                 while (i < n) {
                   unsigned char c = buf[i++];
                   Uint32 m = 1;
                   while (i < n && buf[i] == c)
                     i++, m++;
                   if (! first)
                     printf("+");
                   printf("%u%c", m, c);
                   first = false;
                 }
                 printf("[%u]", n);
                 delete [] buf;
               } else
                 printf("%-5s", "NULL");
             } else
               printf("%-5s", "-"); // no value
           }
         }
         printf("\n");
       }
}
    } else
printf("timed out (%i)\n", timeout);
  }
  // don't want to listen to events anymore
  if (myNdb->dropEventOperation(the_op)) APIERROR(myNdb->getNdbError());
  the_op = 0;

  j++;
}

{
  NdbDictionary::Dictionary *myDict = myNdb->getDictionary();
  if (!myDict) APIERROR(myNdb->getNdbError());
  // remove event from database
  if (myDict->dropEvent(eventName)) APIERROR(myDict->getNdbError());
}
```

```
  delete myNdb;
  delete cluster_connection;
  ndb_end(0);
  return 0;
}

int myCreateEvent(Ndb* myNdb,
      const char *eventName,
      const char *eventTableName,
      const char **eventColumnNames,
      const int noEventColumnNames,
                  bool merge_events)
{
  NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  if (!myDict) APIERROR(myNdb->getNdbError());

  const NdbDictionary::Table *table= myDict->getTable(eventTableName);
  if (!table) APIERROR(myDict->getNdbError());

  NdbDictionary::Event myEvent(eventName, *table);
  myEvent.addTableEvent(NdbDictionary::Event::TE_ALL);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_INSERT);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_UPDATE);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_DELETE);

  myEvent.addEventColumns(noEventColumnNames, eventColumnNames);
  myEvent.mergeEvents(merge_events);

  // Add event to database
  if (myDict->createEvent(myEvent) == 0)
    myEvent.print();
  else if (myDict->getNdbError().classification ==
     NdbError::SchemaObjectExists) {
    printf("Event creation failed, event exists\n");
    printf("dropping Event...\n");
    if (myDict->dropEvent(eventName)) APIERROR(myDict->getNdbError());
    // try again
    // Add event to database
    if ( myDict->createEvent(myEvent)) APIERROR(myDict->getNdbError());
  } else
    APIERROR(myDict->getNdbError());

  return 0;
}
```

2.4.8. Event Handling with Multiple Clusters

This example illustrates the handling log events using the MGM API on multiple clusters in a single application.

The source code for this program may be found in the MySQL 5.1 source tree, in the file `storage/ndb/ndbapi-examples/mgmapi_logevent2/mgmapi_logevent2.cpp`.

```
#include <mysql.h>
#include <ndbapi/NdbApi.hpp>
#include <mgmapi.h>
#include <stdio.h>

/*
 * export LD_LIBRARY_PATH=../../../libmysql_r/.libs:../../../ndb/src/.libs
 */

#define MGMERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_mgm_get_latest_error(h), \
          ndb_mgm_get_latest_error_msg(h)); \
  exit(-1); \
}

#define LOGEVENTERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_logevent_get_latest_error(h), \
          ndb_logevent_get_latest_error_msg(h)); \
  exit(-1); \
}

int main(int argc, char** argv)
{
  NdbMgmHandle h1,h2;
  NdbLogEventHandle le1,le2;
  int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP,
       15, NDB_MGM_EVENT_CATEGORY_CONNECTION,
       15, NDB_MGM_EVENT_CATEGORY_NODE_RESTART,
       15, NDB_MGM_EVENT_CATEGORY_STARTUP,
       15, NDB_MGM_EVENT_CATEGORY_ERROR,
       0 };
  struct ndb_logevent event1, event2;

  if (argc < 3)
  {
```

```
  printf("Arguments are <connect_string cluster 1> <connect_string cluster 2> [<iterations>].\n");
  exit(-1);
}
const char *connectstring1 = argv[1];
const char *connectstring2 = argv[2];
int iterations = -1;
if (argc > 3)
  iterations = atoi(argv[3]);
ndb_init();

h1= ndb_mgm_create_handle();
h2= ndb_mgm_create_handle();
if ( h1 == 0 || h2 == 0 )
{
  printf("Unable to create handle\n");
  exit(-1);
}
if (ndb_mgm_set_connectstring(h1, connectstring1) == -1 ||
    ndb_mgm_set_connectstring(h2, connectstring1))
{
  printf("Unable to set connectstring\n");
  exit(-1);
}
if (ndb_mgm_connect(h1,0,0,0)) MGMERROR(h1);
if (ndb_mgm_connect(h2,0,0,0)) MGMERROR(h2);

if ((le1= ndb_mgm_create_logevent_handle(h1, filter)) == 0) MGMERROR(h1);
if ((le2= ndb_mgm_create_logevent_handle(h1, filter)) == 0) MGMERROR(h2);

while (iterations-- != 0)
{
  int timeout= 1000;
  int r1= ndb_logevent_get_next(le1,&event1,timeout);
  if (r1 == 0)
    printf("No event within %d milliseconds\n", timeout);
  else if (r1 < 0)
    LOGEVENTERROR(le1)
  else
  {
    switch (event1.type) {
    case NDB_LE_BackupStarted:
printf("Node %d: BackupStarted\n", event1.source_nodeid);
printf("  Starting node ID: %d\n", event1.BackupStarted.starting_node);
printf("  Backup ID: %d\n", event1.BackupStarted.backup_id);
break;
    case NDB_LE_BackupCompleted:
printf("Node %d: BackupCompleted\n", event1.source_nodeid);
printf("  Backup ID: %d\n", event1.BackupStarted.backup_id);
break;
    case NDB_LE_BackupAborted:
printf("Node %d: BackupAborted\n", event1.source_nodeid);
break;
    case NDB_LE_BackupFailedToStart:
printf("Node %d: BackupFailedToStart\n", event1.source_nodeid);
break;

    case NDB_LE_NodeFailCompleted:
printf("Node %d: NodeFailCompleted\n", event1.source_nodeid);
break;
    case NDB_LE_ArbitResult:
printf("Node %d: ArbitResult\n", event1.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event1.ArbitResult.code & 0xffff,
       event1.ArbitResult.arbit_node);
break;
    case NDB_LE_DeadDueToHeartbeat:
printf("Node %d: DeadDueToHeartbeat\n", event1.source_nodeid);
printf("  node %d\n", event1.DeadDueToHeartbeat.node);
break;

    case NDB_LE_Connected:
printf("Node %d: Connected\n", event1.source_nodeid);
printf("  node %d\n", event1.Connected.node);
break;
    case NDB_LE_Disconnected:
printf("Node %d: Disconnected\n", event1.source_nodeid);
printf("  node %d\n", event1.Disconnected.node);
break;
    case NDB_LE_NDBStartCompleted:
printf("Node %d: StartCompleted\n", event1.source_nodeid);
printf("  version %d.%d.%d\n",
       event1.NDBStartCompleted.version >> 16 & 0xff,
       event1.NDBStartCompleted.version >> 8 & 0xff,
       event1.NDBStartCompleted.version >> 0 & 0xff);
break;
    case NDB_LE_ArbitState:
printf("Node %d: ArbitState\n", event1.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event1.ArbitState.code & 0xffff,
       event1.ArbitResult.arbit_node);
break;

    default:
break;
    }
  }
```

```
    int r2= ndb_logevent_get_next(le1,&event2,timeout);
    if (r2 == 0)
      printf("No event within %d milliseconds\n", timeout);
    else if (r2 < 0)
      LOGEVENTERROR(le2)
    else
    {
      switch (event2.type) {
      case NDB_LE_BackupStarted:
  printf("Node %d: BackupStarted\n", event2.source_nodeid);
  printf("  Starting node ID: %d\n", event2.BackupStarted.starting_node);
  printf("  Backup ID: %d\n", event2.BackupStarted.backup_id);
  break;
      case NDB_LE_BackupCompleted:
  printf("Node %d: BackupCompleted\n", event2.source_nodeid);
  printf("  Backup ID: %d\n", event2.BackupStarted.backup_id);
  break;
      case NDB_LE_BackupAborted:
  printf("Node %d: BackupAborted\n", event2.source_nodeid);
  break;
      case NDB_LE_BackupFailedToStart:
  printf("Node %d: BackupFailedToStart\n", event2.source_nodeid);
  break;

      case NDB_LE_NodeFailCompleted:
  printf("Node %d: NodeFailCompleted\n", event2.source_nodeid);
  break;
      case NDB_LE_ArbitResult:
  printf("Node %d: ArbitResult\n", event2.source_nodeid);
  printf("  code %d, arbit_node %d\n",
         event2.ArbitResult.code & 0xffff,
         event2.ArbitResult.arbit_node);
  break;
      case NDB_LE_DeadDueToHeartbeat:
  printf("Node %d: DeadDueToHeartbeat\n", event2.source_nodeid);
  printf("  node %d\n", event2.DeadDueToHeartbeat.node);
  break;

      case NDB_LE_Connected:
  printf("Node %d: Connected\n", event2.source_nodeid);
  printf("  node %d\n", event2.Connected.node);
  break;
      case NDB_LE_Disconnected:
  printf("Node %d: Disconnected\n", event2.source_nodeid);
  printf("  node %d\n", event2.Disconnected.node);
  break;
      case NDB_LE_NDBStartCompleted:
  printf("Node %d: StartCompleted\n", event2.source_nodeid);
  printf("  version %d.%d.%d\n",
         event2.NDBStartCompleted.version >> 16 & 0xff,
         event2.NDBStartCompleted.version >> 8 & 0xff,
         event2.NDBStartCompleted.version >> 0 & 0xff);
  break;
      case NDB_LE_ArbitState:
  printf("Node %d: ArbitState\n", event2.source_nodeid);
  printf("  code %d, arbit_node %d\n",
         event2.ArbitState.code & 0xffff,
         event2.ArbitResult.arbit_node);
  break;

      default:
  break;
      }
    }
  }

  ndb_mgm_destroy_logevent_handle(&le1);
  ndb_mgm_destroy_logevent_handle(&le2);
  ndb_mgm_destroy_handle(&h1);
  ndb_mgm_destroy_handle(&h2);
  ndb_end(0);
  return 0;
}
```

2.4.9. Basic BLOB Handling Example

This example illustrates the manipulation of a `BLOB` column in the `NDB` API. It demonstrates how to perform insert, read, and update operations, using both inline value buffers as well as read and write methods.

The source code can be found can be found in the file `storage/ndb/ndbapi-examples/ndbapi_blob/ndbapi_blob.cpp` in the `mysql-5.1` tree.

> **Note**
>
> While the MySQL data type used in the example is actually `TEXT`, the same principles apply

```
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
/* Used for cout. */
```

```
#include <iostream>
#include <stdio.h>
#include <ctype.h>


/**
 * Helper debugging macros
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

/* Quote taken from Project Gutenberg. */
const char *text_quote=
"Just at this moment, somehow or other, they began to run.\n"
"\n"
"  Alice never could quite make out, in thinking it over\n"
"afterwards, how it was that they began:  all she remembers is,\n"
"that they were running hand in hand, and the Queen went so fast\n"
"that it was all she could do to keep up with her:  and still the\n"
"Queen kept crying 'Faster! Faster!' but Alice felt she COULD NOT\n"
"go faster, though she had not breath left to say so.\n"
"\n"
"  The most curious part of the thing was, that the trees and the\n"
"other things round them never changed their places at all:\n"
"however fast they went, they never seemed to pass anything.  'I\n"
"wonder if all the things move along with us?' thought poor\n"
"puzzled Alice.  And the Queen seemed to guess her thoughts, for\n"
"she cried, 'Faster!  Don't try to talk!'\n"
"\n"
"  Not that Alice had any idea of doing THAT.  She felt as if she\n"
"would never be able to talk again, she was getting so much out of\n"
"breath:  and still the Queen cried 'Faster! Faster!' and dragged\n"
"her along.  'Are we nearly there?'  Alice managed to pant out at\n"
"last.\n"
"\n"
"  'Nearly there!' the Queen repeated.  'Why, we passed it ten\n"
"minutes ago!  Faster!'  And they ran on for a time in silence,\n"
"with the wind whistling in Alice's ears, and almost blowing her\n"
"hair off her head, she fancied.\n"
"\n"
"  'Now!  Now!' cried the Queen.  'Faster!  Faster!'  And they\n"
"went so fast that at last they seemed to skim through the air,\n"
"hardly touching the ground with their feet, till suddenly, just\n"
"as Alice was getting quite exhausted, they stopped, and she found\n"
"herself sitting on the ground, breathless and giddy.\n"
"\n"
"  The Queen propped her up against a tree, and said kindly, 'You\n"
"may rest a little now.'\n"
"\n"
"  Alice looked round her in great surprise.  'Why, I do believe\n"
"we've been under this tree the whole time!  Everything's just as\n"
"it was!'\n"
"\n"
"  'Of course it is,' said the Queen, 'what would you have it?'\n"
"\n"
"  'Well, in OUR country,' said Alice, still panting a little,\n"
"'you'd generally get to somewhere else--if you ran very fast\n"
"for a long time, as we've been doing.'\n"
"\n"
"  'A slow sort of country!' said the Queen.  'Now, HERE, you see,\n"
"it takes all the running YOU can do, to keep in the same place.\n"
"If you want to get somewhere else, you must run at least twice as\n"
"fast as that!'\n"
"\n"
"  'I'd rather not try, please!' said Alice.  'I'm quite content\n"
"to stay here--only I AM so hot and thirsty!'\n"
"\n"
" -- Lewis Carroll, 'Through the Looking-Glass'.";

/*
  Function to drop table.
*/
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE my_text"))
    MYSQLERROR(mysql);
}


/*
  Functions to create table.
*/
int try_create_table(MYSQL &mysql)
{
  return mysql_query(&mysql,
                     "CREATE TABLE"
                     "  my_text"
                     "    (my_id INT UNSIGNED NOT NULL,"
                     "     my_text TEXT NOT NULL,"
```

```
                    "     PRIMARY KEY USING HASH (my_id))"
                    "  ENGINE=NDB");
}

void create_table(MYSQL &mysql)
{
  if (try_create_table(mysql))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "MySQL Cluster already has example table: my_text. "
              << "Dropping it..." << std::endl;
    /******************
     * Recreate table *
     ******************/
    drop_table(mysql);
    if (try_create_table(mysql))
      MYSQLERROR(mysql);
  }
}

int populate(Ndb *myNdb)
{
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->insertTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  myBlobHandle->setValue(text_quote, strlen(text_quote));

  int check= myTrans->execute(NdbTransaction::Commit);
  myTrans->close();
  return check != -1;
}


int update_key(Ndb *myNdb)
{
  /*
    Uppercase all characters in TEXT field, using primary key operation.
    Use piece-wise read/write to avoid loading entire data into memory
    at once.
  */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->updateTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());

  /* Execute NoCommit to make the blob handle active. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  Uint64 length= 0;
  if (-1 == myBlobHandle->getLength(length))
    APIERROR(myBlobHandle->getNdbError());

  /*
    A real application should use a much larger chunk size for
    efficiency, preferably much larger than the part size, which
    defaults to 2000. 64000 might be a good value.
  */
#define CHUNK_SIZE 100
  int chunk;
  char buffer[CHUNK_SIZE];
  for (chunk= (length-1)/CHUNK_SIZE; chunk >=0; chunk--)
  {
    Uint64 pos= chunk*CHUNK_SIZE;
    Uint32 chunk_length= CHUNK_SIZE;
    if (pos + chunk_length > length)
      chunk_length= length - pos;

    /* Read from the end back, to illustrate seeking. */
```

```
    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->readData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    int res= myTrans->execute(NdbTransaction::NoCommit);
    if (-1 == res)
      APIERROR(myTrans->getNdbError());

    /* Uppercase everything. */
    for (Uint64 j= 0; j < chunk_length; j++)
      buffer[j]= toupper(buffer[j]);

    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->writeData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    /* Commit on the final update. */
    if (-1 == myTrans->execute(chunk ?
                               NdbTransaction::NoCommit :
                               NdbTransaction::Commit))
      APIERROR(myTrans->getNdbError());
  }

  myNdb->closeTransaction(myTrans);

  return 1;
}


int update_scan(Ndb *myNdb)
{
  /*
    Lowercase all characters in TEXT field, using a scan with
    updateCurrentTuple().
  */
  char buffer[10000];

  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbScanOperation *myScanOp= myTrans->getNdbScanOperation(myTable);
  if (myScanOp == NULL)
    APIERROR(myTrans->getNdbError());
  myScanOp->readTuples(NdbOperation::LM_Exclusive);
  NdbBlob *myBlobHandle= myScanOp->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myScanOp->getNdbError());
  if (myBlobHandle->getValue(buffer, sizeof(buffer)))
    APIERROR(myBlobHandle->getNdbError());

  /* Start the scan. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  int res;
  for (;;)
  {
    res= myScanOp->nextResult(true);
    if (res==1)
      break;                                    // Scan done.
    else if (res)
      APIERROR(myScanOp->getNdbError());

    Uint64 length= 0;
    if (myBlobHandle->getLength(length) == -1)
      APIERROR(myBlobHandle->getNdbError());

    /* Lowercase everything. */
    for (Uint64 j= 0; j < length; j++)
      buffer[j]= tolower(buffer[j]);

    NdbOperation *myUpdateOp= myScanOp->updateCurrentTuple();
    if (myUpdateOp == NULL)
      APIERROR(myTrans->getNdbError());
    NdbBlob *myBlobHandle2= myUpdateOp->getBlobHandle("my_text");
    if (myBlobHandle2 == NULL)
      APIERROR(myUpdateOp->getNdbError());
    if (myBlobHandle2->setValue(buffer, length))
      APIERROR(myBlobHandle2->getNdbError());

    if (-1 == myTrans->execute(NdbTransaction::NoCommit))
      APIERROR(myTrans->getNdbError());
  }

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());

  myNdb->closeTransaction(myTrans);

  return 1;
}
```

```
struct ActiveHookData {
  char buffer[10000];
  Uint32 readLength;
};

int myFetchHook(NdbBlob* myBlobHandle, void* arg)
{
  ActiveHookData *ahd= (ActiveHookData *)arg;

  ahd->readLength= sizeof(ahd->buffer) - 1;
  return myBlobHandle->readData(ahd->buffer, ahd->readLength);
}

int fetch_key(Ndb *myNdb)
{
  /*
    Fetch and show the blob field, using setActiveHook().
  */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->readTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  struct ActiveHookData ahd;
  if (myBlobHandle->setActiveHook(myFetchHook, &ahd) == -1)
    APIERROR(myBlobHandle->getNdbError());

  /*
    Execute Commit, but calling our callback set up in setActiveHook()
    before actually committing.
  */
  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  /* Our fetch callback will have been called during the execute(). */

  ahd.buffer[ahd.readLength]= '\0';
  std::cout << "Fetched data:" << std::endl << ahd.buffer << std::endl;

  return 1;
}


int update2_key(Ndb *myNdb)
{
  char buffer[10000];

  /* Simple setValue() update. */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->updateTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  memset(buffer, ' ', sizeof(buffer));
  if (myBlobHandle->setValue(buffer, sizeof(buffer)) == -1)
    APIERROR(myBlobHandle->getNdbError());

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}


int delete_key(Ndb *myNdb)
{
  /* Deletion of blob row. */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
```

```
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->deleteTuple();
  myNdbOperation->equal("my_id", 1);

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}


int main(int argc, char**argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char *mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /* Connect to mysql server and create table. */
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed.\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
                             0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE TEST_DB");
    if (mysql_query(&mysql, "USE TEST_DB") != 0)
      MYSQLERROR(mysql);

    create_table(mysql);
  }

  /* Connect to ndb cluster. */

  Ndb_cluster_connection cluster_connection(connectstring);
  if (cluster_connection.connect(4, 5, 1))
  {
    std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
    exit(-1);
  }
  /* Optionally connect and wait for the storage nodes (ndbd's). */
  if (cluster_connection.wait_until_ready(30,0) < 0)
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb myNdb(&cluster_connection,"TEST_DB");
  if (myNdb.init(1024) == -1) {      // Set max 1024 parallel transactions
    APIERROR(myNdb.getNdbError());
    exit(-1);
  }

  if(populate(&myNdb) > 0)
    std::cout << "populate: Success!" << std::endl;

  if(update_key(&myNdb) > 0)
    std::cout << "update_key: Success!" << std::endl;

  if(update_scan(&myNdb) > 0)
    std::cout << "update_scan: Success!" << std::endl;

  if(fetch_key(&myNdb) > 0)
    std::cout << "fetch_key: Success!" << std::endl;

  if(update2_key(&myNdb) > 0)
    std::cout << "update2_key: Success!" << std::endl;

  if(delete_key(&myNdb) > 0)
    std::cout << "delete_key: Success!" << std::endl;

  /* Drop table. */
  drop_table(mysql);

  return 0;
}
```

2.4.10. Handling BLOBs Using NdbRecord

This example illustrates the manipulation of a BLOB column in the NDB API using the NdbRecord interface available beginning with MySQL Cluster NDB 6.2.3. It demonstrates how to perform insert, read, and update operations, using both inline value buffers as well as read and write methods. It can be found in the file storage/ndb/ndbapi-examples/ndbapi_blob_ndbrecord/main.cpp in the mysql-5.1-telco and mysql-5.1-telco-6.2 source trees.

> **Note**
>
> While the MySQL data type used in the example is actually TEXT, the same principles apply

```
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
/* Used for cout. */
#include <iostream>
#include <stdio.h>
#include <ctype.h>


/**
 * Helper debugging macros
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

/* Quote taken from Project Gutenberg. */
const char *text_quote=
"Just at this moment, somehow or other, they began to run.\n"
"\n"
"  Alice never could quite make out, in thinking it over\n"
"afterwards, how it was that they began:  all she remembers is,\n"
"that they were running hand in hand, and the Queen went so fast\n"
"that it was all she could do to keep up with her:  and still the\n"
"Queen kept crying 'Faster! Faster!' but Alice felt she COULD NOT\n"
"go faster, though she had not breath left to say so.\n"
"\n"
"  The most curious part of the thing was, that the trees and the\n"
"other things round them never changed their places at all:\n"
"however fast they went, they never seemed to pass anything.  'I\n"
"wonder if all the things move along with us?' thought poor\n"
"puzzled Alice.  And the Queen seemed to guess her thoughts, for\n"
"she cried, 'Faster!  Don't try to talk!'\n"
"\n"
"  Not that Alice had any idea of doing THAT.  She felt as if she\n"
"would never be able to talk again, she was getting so much out of\n"
"breath:  and still the Queen cried 'Faster! Faster!' and dragged\n"
"her along.  'Are we nearly there?'  Alice managed to pant out at\n"
"last.\n"
"\n"
"  'Nearly there!' the Queen repeated.  'Why, we passed it ten\n"
"minutes ago!  Faster!'  And they ran on for a time in silence,\n"
"with the wind whistling in Alice's ears, and almost blowing her\n"
"hair off her head, she fancied.\n"
"\n"
"  'Now!  Now!' cried the Queen.  'Faster!  Faster!'  And they\n"
"went so fast that at last they seemed to skim through the air,\n"
"hardly touching the ground with their feet, till suddenly, just\n"
"as Alice was getting quite exhausted, they stopped, and she found\n"
"herself sitting on the ground, breathless and giddy.\n"
"\n"
"  The Queen propped her up against a tree, and said kindly, 'You\n"
"may rest a little now.'\n"
"\n"
"  Alice looked round her in great surprise.  'Why, I do believe\n"
"we've been under this tree the whole time!  Everything's just as\n"
"it was!'\n"
"\n"
"  'Of course it is,' said the Queen, 'what would you have it?'\n"
"\n"
"  'Well, in OUR country,' said Alice, still panting a little,\n"
"'you'd generally get to somewhere else--if you ran very fast\n"
"for a long time, as we've been doing.'\n"
"\n"
"  'A slow sort of country!' said the Queen.  'Now, HERE, you see,\n"
"it takes all the running YOU can do, to keep in the same place.\n"
"If you want to get somewhere else, you must run at least twice as\n"
"fast as that!'\n"
"\n"
"  'I'd rather not try, please!' said Alice.  'I'm quite content\n"
"to stay here--only I AM so hot and thirsty!'\n"
"\n"
" -- Lewis Carroll, 'Through the Looking-Glass'.";
```

```
/* NdbRecord objects. */

const NdbRecord *key_record;                   // For specifying table key
const NdbRecord *blob_record;                  // For accessing blob
const NdbRecord *full_record;                  // All columns, for insert

/* The row is 4 bytes of primary key + space for blob handle pointer. */
#define ROWSIZE (4 + sizeof(NdbBlob *))

static void setup_records(Ndb *myNdb)
{
  NdbDictionary::RecordSpecification spec[2];

  NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("my_text");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());
  const NdbDictionary::Column *col1= myTable->getColumn("my_id");
  if (col1 == NULL)
    APIERROR(myDict->getNdbError());
  const NdbDictionary::Column *col2= myTable->getColumn("my_text");
  if (col2 == NULL)
    APIERROR(myDict->getNdbError());

  spec[0].column= col1;
  spec[0].offset= 0;
  spec[0].nullbit_byte_offset= 0;
  spec[0].nullbit_bit_in_byte= 0;
  spec[1].column= col2;
  spec[1].offset= 4;
  spec[1].nullbit_byte_offset= 0;
  spec[1].nullbit_bit_in_byte= 0;

  key_record= myDict->createRecord(myTable, &spec[0], 1, sizeof(spec[0]));
  if (key_record == NULL)
    APIERROR(myDict->getNdbError());
  blob_record= myDict->createRecord(myTable, &spec[1], 1, sizeof(spec[0]));
  if (blob_record == NULL)
    APIERROR(myDict->getNdbError());
  full_record= myDict->createRecord(myTable, &spec[0], 2, sizeof(spec[0]));
  if (full_record == NULL)
    APIERROR(myDict->getNdbError());
}

/*
  Function to drop table.
*/
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE my_text"))
    MYSQLERROR(mysql);
}


/*
  Functions to create table.
*/
int try_create_table(MYSQL &mysql)
{
  return mysql_query(&mysql,
                     "CREATE TABLE"
                     "  my_text"
                     "    (my_id INT UNSIGNED NOT NULL,"
                     "     my_text TEXT NOT NULL,"
                     "     PRIMARY KEY USING HASH (my_id))"
                     "  ENGINE=NDB");
}

void create_table(MYSQL &mysql)
{
  if (try_create_table(mysql))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "MySQL Cluster already has example table: my_text. "
              << "Dropping it..." << std::endl;
    /******************
     * Recreate table *
     ******************/
    drop_table(mysql);
    if (try_create_table(mysql))
      MYSQLERROR(mysql);
  }
}

int populate(Ndb *myNdb)
{
  char row[ROWSIZE];

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  Uint32 id= 1;
  memcpy(&row[0], &id, 4);
  NdbOperation *myNdbOperation= myTrans->insertTuple(full_record, row);
  if (myNdbOperation == NULL)
```

```
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  myBlobHandle->setValue(text_quote, strlen(text_quote));

  int check= myTrans->execute(NdbTransaction::Commit);
  myTrans->close();
  return check != -1;
}


int update_key(Ndb *myNdb)
{
  char row[ROWSIZE];

  /*
    Uppercase all characters in TEXT field, using primary key operation.
    Use piece-wise read/write to avoid loading entire data into memory
    at once.
  */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  Uint32 id= 1;
  memcpy(&row[0], &id, 4);
  NdbOperation *myNdbOperation=
    myTrans->updateTuple(key_record, row, blob_record, row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());

  /* Execute NoCommit to make the blob handle active. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  Uint64 length= 0;
  if (-1 == myBlobHandle->getLength(length))
    APIERROR(myBlobHandle->getNdbError());

  /*
    A real application should use a much larger chunk size for
    efficiency, preferably much larger than the part size, which
    defaults to 2000. 64000 might be a good value.
  */
#define CHUNK_SIZE 100
  int chunk;
  char buffer[CHUNK_SIZE];
  for (chunk= (length-1)/CHUNK_SIZE; chunk >=0; chunk--)
  {
    Uint64 pos= chunk*CHUNK_SIZE;
    Uint32 chunk_length= CHUNK_SIZE;
    if (pos + chunk_length > length)
      chunk_length= length - pos;

    /* Read from the end back, to illustrate seeking. */
    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->readData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    int res= myTrans->execute(NdbTransaction::NoCommit);
    if (-1 == res)
      APIERROR(myTrans->getNdbError());

    /* Uppercase everything. */
    for (Uint64 j= 0; j < chunk_length; j++)
      buffer[j]= toupper(buffer[j]);

    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->writeData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    /* Commit on the final update. */
    if (-1 == myTrans->execute(chunk ?
                               NdbTransaction::NoCommit :
                               NdbTransaction::Commit))
      APIERROR(myTrans->getNdbError());
  }

  myNdb->closeTransaction(myTrans);

  return 1;
}


int update_scan(Ndb *myNdb)
{
  /*
    Lowercase all characters in TEXT field, using a scan with
    updateCurrentTuple().
  */
  char buffer[10000];
  char row[ROWSIZE];
```

```
  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbScanOperation *myScanOp=
    myTrans->scanTable(blob_record, NdbOperation::LM_Exclusive);
  if (myScanOp == NULL)
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myScanOp->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myScanOp->getNdbError());
  if (myBlobHandle->getValue(buffer, sizeof(buffer)))
    APIERROR(myBlobHandle->getNdbError());

  /* Start the scan. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  const char *out_row;
  int res;
  for (;;)
  {
    res= myScanOp->nextResult(out_row, true);
    if (res==1)
      break;                                    // Scan done.
    else if (res)
      APIERROR(myScanOp->getNdbError());

    Uint64 length= 0;
    if (myBlobHandle->getLength(length) == -1)
      APIERROR(myBlobHandle->getNdbError());

    /* Lowercase everything. */
    for (Uint64 j= 0; j < length; j++)
      buffer[j]= tolower(buffer[j]);

    NdbOperation *myUpdateOp=
      myScanOp->updateCurrentTuple(myTrans, blob_record, row);
    if (myUpdateOp == NULL)
      APIERROR(myTrans->getNdbError());
    NdbBlob *myBlobHandle2= myUpdateOp->getBlobHandle("my_text");
    if (myBlobHandle2 == NULL)
      APIERROR(myUpdateOp->getNdbError());
    if (myBlobHandle2->setValue(buffer, length))
      APIERROR(myBlobHandle2->getNdbError());

    if (-1 == myTrans->execute(NdbTransaction::NoCommit))
      APIERROR(myTrans->getNdbError());
  }

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());

  myNdb->closeTransaction(myTrans);

  return 1;
}


struct ActiveHookData {
  char buffer[10000];
  Uint32 readLength;
};

int myFetchHook(NdbBlob* myBlobHandle, void* arg)
{
  ActiveHookData *ahd= (ActiveHookData *)arg;

  ahd->readLength= sizeof(ahd->buffer) - 1;
  return myBlobHandle->readData(ahd->buffer, ahd->readLength);
}

int fetch_key(Ndb *myNdb)
{
  char key_row[ROWSIZE];
  char out_row[ROWSIZE];

  /*
    Fetch and show the blob field, using setActiveHook().
  */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  Uint32 id= 1;
  memcpy(&key_row[0], &id, 4);
  NdbOperation *myNdbOperation=
    myTrans->readTuple(key_record, key_row, blob_record, out_row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  struct ActiveHookData ahd;
  if (myBlobHandle->setActiveHook(myFetchHook, &ahd) == -1)
```

```
    APIERROR(myBlobHandle->getNdbError());

  /*
    Execute Commit, but calling our callback set up in setActiveHook()
    before actually committing.
  */
  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  /* Our fetch callback will have been called during the execute(). */

  ahd.buffer[ahd.readLength]= '\0';
  std::cout << "Fetched data:" << std::endl << ahd.buffer << std::endl;

  return 1;
}


int update2_key(Ndb *myNdb)
{
  char buffer[10000];
  char row[ROWSIZE];

  /* Simple setValue() update. */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  Uint32 id= 1;
  memcpy(&row[0], &id, 4);
  NdbOperation *myNdbOperation=
    myTrans->updateTuple(key_record, row, blob_record, row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  memset(buffer, ' ', sizeof(buffer));
  if (myBlobHandle->setValue(buffer, sizeof(buffer)) == -1)
    APIERROR(myBlobHandle->getNdbError());

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}


int delete_key(Ndb *myNdb)
{
  char row[ROWSIZE];

  /* Deletion of blob row. */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  Uint32 id= 1;
  memcpy(&row[0], &id, 4);
  NdbOperation *myNdbOperation= myTrans->deleteTuple(key_record, row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}


int main(int argc, char**argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char *mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  ndb_init();
  MYSQL mysql;

  /* Connect to mysql server and create table. */
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed.\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
                             0, mysqld_sock, 0) )
      MYSQLERROR(mysql);
```

```
    mysql_query(&mysql, "CREATE DATABASE TEST_DB");
    if (mysql_query(&mysql, "USE TEST_DB") != 0)
      MYSQLERROR(mysql);

    create_table(mysql);
  }

  /* Connect to ndb cluster. */

  Ndb_cluster_connection cluster_connection(connectstring);
  if (cluster_connection.connect(4, 5, 1))
  {
    std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
    exit(-1);
  }
  /* Optionally connect and wait for the storage nodes (ndbd's). */
  if (cluster_connection.wait_until_ready(30,0) < 0)
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb myNdb(&cluster_connection,"TEST_DB");
  if (myNdb.init(1024) == -1) {      // Set max 1024 parallel transactions
    APIERROR(myNdb.getNdbError());
    exit(-1);
  }

  setup_records(&myNdb);

  if(populate(&myNdb) > 0)
    std::cout << "populate: Success!" << std::endl;

  if(update_key(&myNdb) > 0)
    std::cout << "update_key: Success!" << std::endl;

  if(update_scan(&myNdb) > 0)
    std::cout << "update_scan: Success!" << std::endl;

  if(fetch_key(&myNdb) > 0)
    std::cout << "fetch_key: Success!" << std::endl;

  if(update2_key(&myNdb) > 0)
    std::cout << "update2_key: Success!" << std::endl;

  if(delete_key(&myNdb) > 0)
    std::cout << "delete_key: Success!" << std::endl;

  /* Drop table. */
  drop_table(mysql);

  return 0;
}
```

Chapter 3. The MGM API

This chapter discusses the MySQL Cluster Management API, a C language API that is used for administrative tasks such as starting and stopping Cluster nodes, backups, and logging. It also covers MGM concepts, programming constructs, and event types.

3.1. General Concepts

Each MGM API function needs a management server handle of type `NdbMgmHandle`. This handle is created by calling the function `ndb_mgm_create_handle()` and freed by calling `ndb_mgm_destroy_handle()`.

See Section 3.2.3.1, “`ndb_mgm_create_handle()`”, and Section 3.2.3.4, “`ndb_mgm_destroy_handle()`”, for more information about these two functions.

> **Important**
>
> You should not share an `NdbMgmHandle` between threads. While it is possible to do so (if you implement your own locks), this is not recommended; each thread should use its own management server handle.

A function can return any of the following:

- An integer value, with a value of `-1` indicating an error.
- A non-constant pointer value. A `NULL` value indicates an error; otherwise, the return value must be freed by the programmer.
- A constant pointer value, with a `NULL` value indicating an error. The returned value should not be freed.

Error conditions can be identified by using the appropriate error-reporting functions `ndb_mgm_get_latest_error()` and `ndb_mgm_error()`.

Here is an example using the MGM API (without error handling for brevity's sake):

```
NdbMgmHandle handle= ndb_mgm_create_handle();
ndb_mgm_connect(handle,0,0,0);
struct ndb_mgm_cluster_state *state= ndb_mgm_get_status(handle);
for(int i=0; i < state->no_of_nodes; i++)
{
  struct ndb_mgm_node_state *node_state= &state->node_states[i];
  printf("node with ID=%d ", node_state->node_id);

  if(node_state->version != 0)
    printf("connected\n");
  else
    printf("not connected\n");
}
free((void*)state);
ndb_mgm_destroy_handle(&handle);
```

3.1.1. Working with Log Events

Data nodes and management servers regularly and on specific occasions report on various log events that occur in the cluster. These log events are written to the cluster log. Optionally an MGM API client may listen to these events using the method `ndb_mgm_listen_event()`. Each log event belongs to a category `ndb_mgm_event_category`) and has a severity `ndb_mgm_event_severity` associated with it. Each log event also has a level (0-15) associated with it.

Which log events that come out is controlled with `ndb_mgm_listen_event()`, `ndb_mgm_set_clusterlog_loglevel()`, and `ndb_mgm_set_clusterlog_severity_filter()`.

This is an example showing how to listen to events related to backup:

```
int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP, 0 };
int fd = ndb_mgm_listen_event(handle, filter);
```

3.1.2. Structured Log Events

The following steps are involved:

1. Create an `NdbLogEventHandle` using `ndb_mgm_create_logevent_handle()`.

2. Wait for and store log events using `ndb_logevent_get_next()`.

3. The log event data is available in the structure `ndb_logevent`. The data which is specific to a particular event is stored in a union between structures; use `ndb_logevent::type` to decide which structure is valid.

The following sample code demonstrates listening to events related to backups:

```
int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP, 0 };
NdbLogEventHandle le_handle= ndb_mgm_create_logevent_handle(handle, filter);
struct ndb_logevent le;
int r= ndb_logevent_get_next(le_handle, &le, 0);
if(r < 0)
  /*  error  */
else if(r == 0)
  /*  no event  */

switch(le.type)
{
  case NDB_LE_BackupStarted:
    ... le.BackupStarted.starting_node;
    ... le.BackupStarted.backup_id;
    break;
  case NDB_LE_BackupFailedToStart:
    ... le.BackupFailedToStart.error;
    break;
  case NDB_LE_BackupCompleted:
    ... le.BackupCompleted.stop_gci;
    break;
  case NDB_LE_BackupAborted:
    ... le.BackupStarted.backup_id;
    break;
  default:
    break;
}
```

For more information, see Section 3.2.1, “Log Event Functions”.

> **Note**
>
> Available log event types are listed in Section 3.3.4, “The `Ndb_logevent_type` Type”, as well as in the file `/storage/ndb/include/mgmapi/ndb_logevent.h` in the MySQL 5.1 sources.

3.2. MGM C API Function Listing

This section covers the structures and functions used in the MGM API. Listings are grouped by purpose or use.

3.2.1. Log Event Functions

This section discusses functions that are used for listening to log events.

3.2.1.1. `ndb_mgm_listen_event()`

Description. This function is used to listen to log events, which are read from the return file descriptor. Events use a text-based format, the same as in the cluster log.

Signature.

```
int ndb_mgm_listen_event
    (
      NdbMgmHandle handle,
      const int    filter[]
    )
```

Parameters. This function takes two arguments:

- An `NdbMgmHandle` *`handle`*.
- A *`filter`* which consists of a series of `{level, ndb_mgm_event_category}` pairs (in a single array) that are pushed to a file descriptor. Use `0` for the level to terminate the list.

Return Value. The file descriptor from which events are to be read.

3.2.1.2. `ndb_mgm_create_logevent_handle()`

Description. This function is used to create a log event handle.

Signature.

```
NdbLogEventHandle ndb_mgm_create_logevent_handle
    (
      NdbMgmHandle handle,
      const int    filter[]
    )
```

Parameters. This function takes two arguments:

- An `NdbMgmHandle` *`handle`*.
- A *`filter`* which consists of a series of `{level, ndb_mgm_event_category}` pairs (in a single array) that are pushed to a file descriptor. Use `0` for the level to terminate the list.

Return Value. A log event handle.

3.2.1.3. `ndb_mgm_destroy_logevent_handle()`

Description. Use this function to destroy a log event handle when there is no further need for it.

Signature.

```
void ndb_mgm_destroy_logevent_handle
    (
      NdbLogEventHandle* handle
    )
```

Parameters. A pointer to a log event *`handle`*.

Return Value. *None*.

3.2.1.4. `ndb_logevent_get_fd()`

Description. This function retrieves a file descriptor from an `NdbMgmLogEventHandle`. It was implemented in MySQL 5.1.12.

> **Warning**
>
> Do not attempt to read from the file descriptor returned by this function.

Signature.

```
int ndb_logevent_get_fd
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A `LogEventHandle`.

Return Value. A file descriptor. In the event of failure, `-1` is returned.

3.2.1.5. `ndb_logevent_get_next()`

Description. This function is used to retrieve the next log event, using the event's data to fill in the supplied `ndb_logevent` structure.

Signature.

```
int ndb_logevent_get_next
    (
      const NdbLogEventHandle handle,
      struct ndb_logevent*    logevent,
      unsigned                timeout
    )
```

Parameters. Three parameters are expected by this functions:

- An `NdbLogEventHandle`
- A pointer to an `ndb_logevent` data structure
- The number of milliseconds to wait for the event before timing out; passing `0` for this parameter causes the function to block until the next log event is received

Return Value. The value returned by this function is interpreted as follows:

- `> 0`: The event exists, and it data was retrieved into the *`logevent`*
- `0`: A timeout occurred while waiting for the event (more than *`timeout`* milliseconds elapsed)
- `< 0`: An error occurred.

If the return value is less than or equal to zero, then the *`logevent`* is not altered or affected in any way.

3.2.1.6. `ndb_logevent_get_latest_error()`

Description. This function retrieves the error code from the most recent error.

> **Note**
>
> You may prefer to use `ndb_logevent_get_latest_error_msg()` instead. See Section 3.2.1.7, “`ndb_logevent_get_latest_error_msg()`”

Signature.

```
int ndb_logevent_get_latest_error
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A log event handle.

Return Value. An error code.

3.2.1.7. `ndb_logevent_get_latest_error_msg()`

Description. Retrieves the text of the most recent error obtained while trying to read log events.

Signature.

```
const char* ndb_logevent_get_latest_error_msg
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A log event handle.

Return Value. The text of the error message.

3.2.2. MGM API Error Handling Functions

The MGM API used for Error handling are discussed in this section.

Each MGM API error is characterised by an error code and an error message. There may also be an error description that may provide additional information about the error. The API provides functions to obtain this information in the event of an error.

3.2.2.1. `ndb_mgm_get_latest_error()`

Description. This function is used to get the latest error code associated with a given management server handle.

Signature.

```
int ndb_mgm_get_latest_error
    (
      const NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. An error code corresponding to an `ndb_mgm_error` value; see Section 3.3.3, "The `ndb_mgm_error` Type". You can obtain the related error message using `ndb_mgm_get_latest_error_msg()`; see Section 3.2.2.2, "`ndb_mgm_get_latest_error_msg()`".

3.2.2.2. `ndb_mgm_get_latest_error_msg()`

Description. This function is used to obtain the latest general error message associated with an `NdbMgmHandle`.

Signature.

```
const char* ndb_mgm_get_latest_error_msg
    (
      const NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. The error message text. More specific information can be obtained using `ndb_mgm_get_latest_error_desc()`; see Section 3.2.2.3, "`ndb_mgm_get_latest_error_desc()`", for details.

3.2.2.3. `ndb_mgm_get_latest_error_desc()`

Description. Get the most recent error description associated with an `NdbMgmHandle`; this description provides additional information regarding the error message.

Signature.

```
const char* ndb_mgm_get_latest_error_desc
    (
      const NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. The error description text.

3.2.2.4. `ndb_mgm_set_error_stream()`

Description. The function can be used to set the error output stream.

Signature.

```
void ndb_mgm_set_error_stream
    (
      NdbMgmHandle handle,
      FILE*        file
    )
```

Parameters. This function requires two parameters:

- An `NdbMgmHandle`
- A pointer to the file to which errors are to be sent.

Return Value. *None*.

3.2.3. Management Server Handle Functions

This section contains information about the MGM API functions used to create and destroy management server handles.

3.2.3.1. `ndb_mgm_create_handle()`

Description. This function is used to create a handle to a management server.

Signature.

```
NdbMgmHandle ndb_mgm_create_handle
    (
```

```
      void
    )
```

Parameters. *None*.

Return Value. An `NdbMgmHandle`.

3.2.3.2. `ndb_mgm_set_name()`

Description. This function can be used to set a name for the management server handle, which is then reported in the Cluster log.

Signature.

```
void ndb_mgm_set_name
    (
      NdbMgmHandle handle,
      const char*  name
    )
```

Parameters. This function takes two arguments:

- A management server *`handle`*.
- The desired *`name`* for the *`handle`*.

Return Value. *None*.

3.2.3.3. `ndb_mgm_set_ignore_sigpipe()`

Description. Beginning with MySQL Cluster NDB 6.3.19, the MGM API by default installs a signal handler that ignores all `SIGPIPE` signals that might occur when writing to asocket that has been closed or reset. An application that provides its own handler for `SIGPIPE` should call this function after creating the management server handle and before using the handle to connect to the management server. (In other words, call this function after using `ndb_mgm_create_handle()` but before calling `ndb_mgm_connect()`, which causes the MGM API's `SIGPIPE` handler to be installed unless overridden.)

> **Note**
>
> Previous to MySQL Cluster NDB 6.3.19, MGM API applications simply exited without returning an error whenever the connection to the management server was lost. (Bug#40498)

Signature.

```
int ndb_mgm_set_ignore_sigpipe
    (
      NdbMgmHandle handle,
      int ignore = 1
    )
```

Parameters. This function takes two parameters:

- A management server handle
- An integer value which determines whether to *`ignore`* `SIGPIPE` errors. Set this to 1 (the default) to cause the MGM API to ignore `SIGPIPE`; set to zero if you wish for `SIGPIPE` to propagate to your MGM API application.

Return Value. *None*.

3.2.3.4. `ndb_mgm_destroy_handle()`

Description. This function destroys a management server handle

Signature.

```
void ndb_mgm_destroy_handle
    (
      NdbMgmHandle* handle
    )
```

Parameters. A pointer to the `NdbMgmHandle` to be destroyed.

Return Value. *None*.

3.2.4. Management Server Connection Functions

This section discusses MGM API functions that are used to initiate, configure, and terminate connections to an `NDB` management server.

3.2.4.1. `ndb_mgm_get_connectstring()`

Description. This function retrieves the connectstring used for a connection.

> **Note**
>
> This function returns the default connectstring if no call to `ndb_mgm_set_connectstring()` has been performed. In addition, the returned connectstring may be formatted slightly differently than the original in that it may contain specifiers not present in the original.
>
> The connectstring format is the same as that discussed for Section 3.2.4.9, “`ndb_mgm_set_connectstring()`”.

Signature.

```
const char* ndb_mgm_get_connectstring
    (
      NdbMgmHandle handle,
      char*        buffer,
      int          size
    )
```

Parameters. This function takes three arguments:

- An `NdbMgmHandle`.
- A pointer to a *`buffer`* in which to place the result.
- The *`size`* of the buffer.

Return Value. The connectstring — this is the same value that is pushed to the *`buffer`*.

3.2.4.2. `ndb_mgm_get_configuration_nodeid()`

Description. This function gets the ID of the node to which the connection is being (or was) made.

Signature.

```
int ndb_mgm_get_configuration_nodeid
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server handle.

Return Value. A node ID.

3.2.4.3. `ndb_mgm_get_connected_port()`

Description. This function retrieves the number of the port used by the connection.

Signature.

```
int ndb_mgm_get_connected_port
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. A port number.

3.2.4.4. ndb_mgm_get_connected_host()

Description. This function is used to obtain the name of the host to which the connection is made.

Signature.

```
const char* ndb_mgm_get_connected_host
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server *handle*.

Return Value. A host name.

3.2.4.5. ndb_mgm_is_connected()

Description. Used to determine whether a connection has been established.

> **Note**
>
> This function does not determine whether or not there is a "live" management server at the other end of the connection. Beginning with MySQL 5.1.17, you can use `ndb_mgm_check_connection()` to accomplish that task. See Section 3.2.4.6, "`ndb_mgm_check_connection()`", for more information.

Signature.

```
int ndb_mgm_is_connected
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server *handle*.

Return Value. This function returns an integer, whose value is interpreted as follows:

- `0`: Not connected to the management node.
- Any nonzero value: A connection has been established with the management node.

3.2.4.6. ndb_mgm_check_connection()

Description. This function can be used to determine whether a management server is running on a given connection from a management client.

Prior to MySQL 5.1.17, this function was available but required extremely large timeouts to be configured for it to be effective.

Signature.

```
int ndb_mgm_check_connection
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle` (see Section 3.1, "General Concepts").

Return Value. This function returns `-1` in the event of an error; otherwise it returns `0`.

3.2.4.7. ndb_mgm_number_of_mgmd_in_connect_string()

Description. This is a convenience function which provides an easy way to determine the number of management servers referenced in a connectstring as set using `ndb_mgm_set_connectstring()`.

This function was added in MySQL 5.1.18.

Signature.

```
int ndb_mgm_number_of_mgmd_in_connect_string
    (
      NdbMgmHandle handle
```

```
    )
```

Parameters. A management handle (`NdbMgmHandle`).

Return Value. On success, a non-negative integer; a negative integer indicates failure.

3.2.4.8. `ndb_mgm_set_bindaddress()`

Description. This function allows you to set a local bind address for the management server. If used, it must be called before connecting to the management server.

This function was added in MySQL 5.1.18.

Signature.

```
int ndb_mgm_set_bindaddress
    (
      NdbMgmHandle handle,
      const char*  address
    )
```

Parameters. This function takes two parameters:

- A management handle (`NdbMgmHandle`).
- A string *`address`* of the form *`host`*`[:`*`port`*`]`.

Return Value. Returns an integer:

- `0` indicates success
- Any nonzero value indicates failure (the address was not valid)

> **Important**
>
> Errors caused by binding an otherwise valid local address are not reported until the connection to the management is actually attempted.

3.2.4.9. `ndb_mgm_set_connectstring()`

Description. This function is used to set the connectstring for a management server connection to a node.

Signature.

```
int ndb_mgm_set_connectstring
    (
      NdbMgmHandle handle,
      const char*  connectstring
    )
```

Parameters. `ndb_mgm_set_connectstring()` takes two parameters:

- A management server *`handle`*.
- A *`connectstring`* whose format is shown here:

  ```
  connectstring :=
      [nodeid-specification,]host-specification[,host-specification]
  ```

  (It is possible to establish connections with multiple management servers using a single connectstring.)

  ```
  nodeid-specification := nodeid=id
  host-specification := host[:port]
  ```

  `id`, *`port`*, and *`host`* are defined as follows:

  - *`id`*: An integer greater than `0` identifying a node in `config.ini`.
  - *`port`*: An integer referring to a standard Unix port.

- *host*: A string containing a valid network host address.

Section 3.2.4.1, “ndb_mgm_get_connectstring()” also uses this format for connectstrings.

Return Value. This function returns -1 in the event of failure.

3.2.4.10. ndb_mgm_set_configuration_nodeid()

Description. This function sets the connection node ID.

Signature.

```
int ndb_mgm_set_configuration_nodeid
    (
      NdbMgmHandle handle,
      int          id
    )
```

Parameters. This function requires two parameters:

- An NdbMgmHandle.
- The id of the node to connect to.

Return Value. This function returns -1 in the event of failure.

3.2.4.11. ndb_mgm_set_timeout()

Description. Normally, network operations time out after 60 seconds. This function allows you to vary this time.

This function was introduced in MySQL 5.1.18.

Signature.

```
int ndb_mgm_set_timeout
    (
      NdbMgmHandle handle,
      unsigned int timeout
    )
```

Parameters. This function takes two parameters:

- A management server handle (NdbMgmHandle).
- An amount of time to wait before timing out, expressed in milliseconds.

> **Note**
>
> The *timeout* must be an even multiple of 1000 — that is, it must be equivalent to an integral number of seconds. Fractional seconds are not supported.

Return Value. Returns 0 on success, with any other value representing failure.

3.2.4.12. ndb_mgm_connect()

Description. This function establishes a connection to a management server specified by the connectstring set by Section 3.2.4.9, “ndb_mgm_set_connectstring()”.

Signature.

```
int ndb_mgm_connect
    (
      NdbMgmHandle handle,
      int          retries,
      int          delay,
      int          verbose
    )
```

Parameters. This function takes 4 arguments:

- A management server *`handle`*.
- The number of `retries` to make when attempting to connect. `0` for this value means that one connection attempt is made.
- The number of seconds to *`delay`* between connection attempts.
- If verbose is `1`, then a message is printed for each connection attempt.

Return Value. This function returns `-1` in the event of failure.

3.2.4.13. `ndb_mgm_disconnect()`

Description. This function terminates a management server connection.

Signature.

```
int ndb_mgm_disconnect
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return Value. Returns `-1` if unable to disconnect.

3.2.5. Cluster Status Functions

This section discusses how to obtain status information from MySQL Cluster nodes.

3.2.5.1. `ndb_mgm_get_status()`

Description. This function is used to obtain the status of the nodes in a MySQL Cluster.

> **Note**
>
> The caller must free the pointer returned by this function.

Signature.

```
struct ndb_mgm_cluster_state* ndb_mgm_get_status
    (
      NdbMgmHandle handle
    )
```

Parameters. This function takes a single parameter — a management server *`handle`*.

Return Value. A pointer to an `ndb_mgm_cluster_state` data structure. See Section 3.4.3, “The `ndb_mgm_cluster_state` Structure”, for more information.

3.2.5.2. `ndb_mgm_dump_state()`

Description. This function can be used to dump debugging information to the cluster log. The MySQL Cluster management client `DUMP` command is a wrapper for this function.

> **Important**
>
> `ndb_mgm_dump_state()`, like the `DUMP` command, can cause a running MySQL Cluster to malfunction or even to fail completely if it is used improperly. Be sure to consult the relevant documentation before using this function. For more information on the `DUMP` command, and for a listing of current `DUMP` codes and their effects, see Section 5.2, “`DUMP` Commands”.

This function became available in MySQL Cluster NDB 6.1.2.17 and MySQL Cluster NDB 6.3.19.

Signature.

```
int ndb_mgm_dump_state
    (
      NdbMgmHandle handle,
```

```
      int nodeId,
      const int* arguments,
      int numberOfArguments,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function takes the following pararemeters:

- A management server handle (`NdbMgmHandle`)
- The *`nodeId`* of a cluster data node.
- An array of *`arguments`*. The first of these is the `DUMP` code to be executed. Subsequent arguments can be passed in this array if needed by or desired for the corresponding `DUMP` command.
- The *`numberOfArguments`* to be passed.
- An `ndb_mgm_reply` which cntains a return code along with a response or error message (see Section 3.4.4, "The `ndb_mgm_reply` Structure", for more information).

Return Value. `0` on success; otherwise, an error code.

Example. The following example has the same result as running `2 DUMP 1000` in the management client:

```
//  [...]
#include <mgmapi_debug.h>
//  [...]
struct ndb_mgm_reply reply;
int args[1];
int stat, arg_count, node_id;

args[0] = 1000;
arg_count = 1;
node_id = 2;

stat = ndb_mgm_dump_state(h, node_id, args, arg_count, &reply);
```

3.2.6. Functions for Starting & Stopping Nodes

The MGM API provides several functions which can be used to start, stop, and restart one or more Cluster data nodes. These functions are discussed in this section.

Starting, Stopping, and Restarting Nodes. You can start, stop, and restart Cluster nodes using the following functions:

- **Starting Nodes.** Use `ndb_mgm_start()`.
- **Stopping Nodes.** Use `ndb_mgm_stop()`, `ndb_mgm_stop2()`, or `ndb_mgm_stop3()`.
- **Restarting Nodes.** Use `ndb_mgm_restart()`, `ndb_mgm_restart2()`, or `ndb_mgm_restart3()`.

These functions are detailed in the next few sections.

3.2.6.1. `ndb_mgm_start()`

Description. This function can be used to start one or more Cluster nodes. The nodes to be started must have been started with the no-start option (`-n`), meaning that the data node binary was started and is waiting for a `START` management command which actually enables the node.

Signature.

```
int ndb_mgm_start
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_start() takes 3 parameters:

- An `NdbMgmHandle`.

- A *number* of nodes to be started. Use 0 to start all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be started.

Return Value. The number of nodes actually started; in the event of failure, -1 is returned.

3.2.6.2. ndb_mgm_stop()

Description. This function stops one or more data nodes.

Signature.

```
int ndb_mgm_stop
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_stop() takes 3 parameters:

- An NdbMgmHandle.
- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.

Calling this function is equivalent to calling ndb_mgm_stop2(*handle, number, list,* 0). See Section 3.2.6.3, "ndb_mgm_stop2()".

Return Value. The number of nodes actually stopped; in the event of failure, -1 is returned.

3.2.6.3. ndb_mgm_stop2()

Description. Like ndb_mgm_stop(), this function stops one or more data nodes. However, it offers the ability to specify whether or not the nodes shut down gracefully.

Signature.

```
int ndb_mgm_stop2
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort
    )
```

Parameters. ndb_mgm_stop2() takes 4 parameters:

- An NdbMgmHandle.
- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.
- The value of *abort* determines how the nodes will be shut down. 1 indicates the nodes will shut down immediately; 0 indicates that the nodes will stop gracefully.

Return Value. The number of nodes actually stopped; in the event of failure, -1 is returned.

3.2.6.4. ndb_mgm_stop3()

Description. Like ndb_mgm_stop() and ndb_mgm_stop2(), this function stops one or more data nodes. Like ndb_mgm_stop2(), it offers the ability to specify whether the nodes should shut down gracefully. In addition, it provides for a way to check to see whether disconnection is required prior to stopping a node.

Signature.

```
int ndb_mgm_stop3
```

```
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort,
      int*         disconnect
    )
```

Parameters. ndb_mgm_stop3() takes 5 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- The value of *`abort`* determines how the nodes will be shut down. `1` indicates the nodes will shut down immediately; `0` indicates that the nodes will stop gracefully.
- If *`disconnect`* returns `1` (`true`), this means the you must disconnect before you can apply the command to stop. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return Value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

3.2.6.5. `ndb_mgm_restart()`

Description. This function can be used to restart one or more Cluster data nodes.

Signature.

```
int ndb_mgm_restart
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_restart() takes 3 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.

Calling this function is equivalent to calling

```
ndb_mgm_restart2(handle, number, list, 0, 0, 0);
```

See Section 3.2.6.6, “`ndb_mgm_restart2()`”, for more information.

Return Value. The number of nodes actually restarted; `-1` on failure.

3.2.6.6. `ndb_mgm_restart2()`

Description. Like `ndb_mgm_restart()`, this function can be used to restart one or more Cluster data nodes. However, `ndb_mgm_restart2()` provides additional restart options, including initial restart, waiting start, and immediate (forced) restart.

Signature.

```
int ndb_mgm_restart2
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort
    )
```

Parameters. ndb_mgm_restart2() takes 6 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- If *`initial`* is true (`1`), then each node undergoes an initial restart — that is, its file system is removed.
- If `nostart` is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *`abort`* is true, then the nodes are restarted immediately, bypassing any graceful restart.

Return Value. The number of nodes actually restarted; `-1` on failure.

3.2.6.7. `ndb_mgm_restart3()`

Description. Like `ndb_mgm_restart2()`, this function can be used to cause an initial restart, waiting restart, and immediate (forced) restart on one or more Cluster data nodes. However, `ndb_mgm_restart3()` provides additional the additional options of checking whether disconnection is required prior to the restart.

Signature.

```
int ndb_mgm_restart3
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort,
      int*         disconnect
    )
```

Parameters. ndb_mgm_restart() takes 7 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- If *`initial`* is true (`1`), then each node undergoes an initial restart — that is, its file system is removed.
- If `nostart` is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *`abort`* is true, then the nodes are forced to restart immediately without performing a graceful restart.
- If *`disconnect`* returns `1` (`true`), this means the you must disconnect before you can apply the command to restart. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return Value. The number of nodes actually restarted; `-1` on failure.

3.2.7. Cluster Log Functions

This section covers the functions available in the MGM API for controlling the output of the cluster log.

3.2.7.1. `ndb_mgm_get_clusterlog_severity_filter()`

Description. This function is used to retrieve the cluster log severity filter currently in force.

> **Important**
>
> The parameters and return type of this function changed significantly between MySQL 5.1.13 and 5.1.14. The changes are detailed in the *Signature*, *Parameters*, and *Return Type* sections that follow.
>
> These changes were done in order to make this function thread-safe. The pre-5.1.14 version is still supported for backward compatibility, but you should protect it with a mutex if you intend to call it from more than one thread.

Signature. As of MySQL 5.1.14:

```
int ndb_mgm_get_clusterlog_severity_filter
    (
      NdbMgmHandle handle,
      struct ndb_mgm_severity* severity,
      unsigned int size
    )
```

In MySQL 5.1.13 and earlier, this function took only a single parameter, as shown here:

```
const unsigned int* ndb_mgm_get_clusterlog_severity_filter
    (
      NdbMgmHandle handle
    )
```

Parameters. This function added two new parameters in MySQL 5.1.14.

- *All MySQL 5.1 releases*:

  An `NdbMgmHandle`.

- *Additionally, in MySQL 5.1.14 and later*:

  - A vector *`severity`* of seven (`NDB_MGM_EVENT_SEVERITY_ALL`) elements, each of which is an `ndb_mgm_severity` structure, where each element contains `1` if a severity indicator is enabled and `0` if not. A severity level is stored at position *`ndb_mgm_clusterlog_level`*; for example the error level is stored at position `NDB_MGM_EVENT_SEVERITY_ERROR`. The first element (position `NDB_MGM_EVENT_SEVERITY_ON`) in the vector signals whether the cluster log is disabled or enabled.

  - The *`size`* of the vector (`NDB_MGM_EVENT_SEVERITY_ALL`).

Return Value. This function's return type changed beginning with MySQL 5.1.14.

- *MySQL 5.1.13 and earlier*:

  A *severity filter*, which is a vector containing 7 elements. Each element equals `1` if the corresponding severity indicator is enabled, and `0` if it is not. A severity level is stored at position *`ndb_mgm_clusterlog_level`* — for example, the “error” level is stored at position `NDB_MGM_EVENT_SEVERITY_ERROR`. The first element in the vector (`NDB_MGM_EVENT_SEVERITY_ON`) signals whether the cluster log is enabled or disabled.

- *MySQL 5.1.14 and later*:

  The number of returned severities, or `-1` in the event of an error.

3.2.7.2. `ndb_mgm_set_clusterlog_severity_filter()`

Description. This function is used to set a cluster log severity filter.

Signature.

```
int ndb_mgm_set_clusterlog_severity_filter
    (
      NdbMgmHandle                handle,
      enum ndb_mgm_event_severity severity,
      int                         enable,
      struct ndb_mgm_reply*       reply
    )
```

Parameters. This function takes 4 parameters:

- A management server *`handle`*.
- A cluster log *`severity`* to filter.
- A flag to *`enable`* or disable the filter; `1` enables and `0` disables the filter.
- A pointer to an `ndb_mgm_reply` structure for a reply message. See Section 3.4.4, “The `ndb_mgm_reply` Structure”.

Return Value. The function returns `-1` in the event of failure.

3.2.7.3. ndb_mgm_get_clusterlog_loglevel()

Description. This function, added in MySQL 5.1.16, is used to obtain log category and level information. It replaces the older `ndb_mgm_get_loglevel_clusterlog()` function, which performed the same purpose, but was not thread-safe. (See later in this section for a brief description of the deprecated version of the function.)

Signature.

```
int ndb_mgm_get_clusterlog_loglevel
    (
      NdbMgmHandle handle,
      struct ndb_mgm_loglevel* loglevel,
      unsigned int size
    )
```

Parameters. `ndb_mgm_get_clusterlog_loglevel()` takes the following parameters:

- A management *`handle`* (`NdbMgmHandle`).
- A *`loglevel`* (log level) vector consisting of twelve elements, each of which is an `ndb_mgm_loglevel` structure and which represents a log level of the corresponding category.
- The *`size`* of the vector (`MGM_LOGLEVELS`).

Return Value. This function returns the number of returned loglevels or `-1` in the event of an error.

> **Note**
>
> Prior to MySQL 5.1.14, this function was known as `ndb_mgm_get_loglevel_clusterlog()`, and had the following signature:
>
> ```
> const unsigned int* ndb_mgm_get_loglevel_clusterlog
>     (
>       NdbMgmHandle handle
>     )
> ```
>
> This version of the function is now deprecated, but is still available for backward compatibility; however, in new applications, it is recommended that you use `ndb_mgm_get_clusterlog_loglevel()`, since it is thread-safe, and the older function is not.

3.2.7.4. ndb_mgm_set_clusterlog_loglevel()

Description. This function is used to set the log category and levels for the cluster log.

Signature.

```
int ndb_mgm_set_clusterlog_loglevel
    (
      NdbMgmHandle                handle,
      int                         id,
      enum ndb_mgm_event_category category,
      int                         level,
      struct ndb_mgm_reply*       reply)
```

Parameters. This function takes 5 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the node affected.
- An event *`category`* — this is one of the values listed in Section 3.3.7, “The `ndb_mgm_event_category` Type” .
- A logging *`level`*.
- A pointer to an `ndb_mgm_reply` structure for the *`reply`* message. (See Section 3.4.4, “The `ndb_mgm_reply` Structure”.)

Return Value. In the event of an error, this function returns `-1`.

3.2.8. Backup Functions

This section covers the functions provided in the MGM API for starting and stopping backups.

3.2.8.1. ndb_mgm_start_backup()

Description. This function is used to initiate a backup of a MySQL Cluster.

Signature.

```
int ndb_mgm_start_backup
    (
      NdbMgmHandle          handle,
      int                   wait,
      unsigned int*         id,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function requires 4 parameters:

- A management server *handle* (an NdbMgmHandle).
- A *wait* flag, with the following possible values:
  - 0: Do not wait for confirmation of the backup.
  - 1: Wait for the backup to be started.
  - 2: Wait for the backup to be completed.
- A backup *id* to be returned by the function.

  > **Note**
  >
  > No backup *id* is returned if *wait* is set equal to 0.

- A pointer to an ndb_mgm_reply structure to accommodate a *reply*. See Section 3.4.4, "The ndb_mgm_reply Structure".

Return Value. In the event of failure, the function returns -1.

3.2.8.2. ndb_mgm_abort_backup()

Description. This function is used to stop a Cluster backup.

Signature.

```
int ndb_mgm_abort_backup
    (
      NdbMgmHandle          handle,
      unsigned int          id,
      struct ndb_mgm_reply* reply)
```

Parameters. This function takes 3 parameters:

- An NdbMgmHandle.
- The *id* of the backup to be aborted.
- A pointer to an ndb_mgm_reply structure.

Return Value. In case an error, this function returns -1.

3.2.9. Single-User Mode Functions

The MGM API allows the programmer to put the cluster into single-user mode — and to return it to normal mode again — from within an application. This section covers the functions that are used for these operations.

3.2.9.1. ndb_mgm_enter_single_user()

Description. This function is used to enter single-user mode on a given node.

Signature.

```
int ndb_mgm_enter_single_user
    (
      NdbMgmHandle          handle,
      unsigned int          id,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function takes 3 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the node to be used in single-user mode.
- A pointer to an `ndb_mgm_reply` structure, used for a *`reply`* message.

Return Value. Returns `-1` in the event of failure.

3.2.9.2. `ndb_mgm_exit_single_user()`

Description. This function is used to exit single-user mode and to return to normal operation.

Signature.

```
int ndb_mgm_exit_single_user
    (
      NdbMgmHandle          handle,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function requires 2 arguments:

- An `NdbMgmHandle`.
- A pointer to an `ndb_mgm_reply`.

Return Value. Returns `-1` in case of an error.

3.3. MGM Datatypes

This section discusses the datatypes defined by the MGM API.

> **Note**
>
> The types described in this section are all defined in the file `/storage/ndb/include/mgmapi/mgmapi.h`, with the exception of `Ndb_logevent_type`, `ndb_mgm_event_severity`, `ndb_mgm_logevent_handle_error`, and `ndb_mgm_event_category`, which are defined in `/storage/ndb/include/mgmapi/ndb_logevent.h`.

3.3.1. The `ndb_mgm_node_type` Type

Description. This is used to classify the different types of nodes in a MySQL Cluster.

Enumeration values.

Value	Description
`NDB_MGM_NODE_TYPE_UNKNOWN`	Unknown
`NDB_MGM_NODE_TYPE_API`	API Node (SQL node)
`NDB_MGM_NODE_TYPE_NDB`	Data node
`NDB_MGM_NODE_TYPE_MGM`	Management node

3.3.2. The `ndb_mgm_node_status` Type

Description. This type describes a Cluster node's status.

Enumeration values.

Value	Description
NDB_MGM_NODE_STATUS_UNKNOWN	The node's status is not known
NDB_MGM_NODE_STATUS_NO_CONTACT	The node cannot be contacted
NDB_MGM_NODE_STATUS_NOT_STARTED	The node has not yet executed the startup protocol
NDB_MGM_NODE_STATUS_STARTING	The node is executing the startup protocol
NDB_MGM_NODE_STATUS_STARTED	The node is running
NDB_MGM_NODE_STATUS_SHUTTING_DOWN	The node is shutting down
NDB_MGM_NODE_STATUS_RESTARTING	The node is restarting
NDB_MGM_NODE_STATUS_SINGLEUSER	The node is running in single-user (maintenance) mode
NDB_MGM_NODE_STATUS_RESUME	The node is in resume mode

3.3.3. The ndb_mgm_error Type

Description. The values for this type are the error codes that may be generated by MGM API functions. These may be found in Section 4.1, "MGM API Errors".

See also Section 3.2.2.1, "ndb_mgm_get_latest_error()", for more information.

3.3.4. The Ndb_logevent_type Type

Description. These are the types of log events available in the MGM API, grouped by event category. (See Section 3.3.7, "The ndb_mgm_event_category Type".)

Enumeration values.

Category	Type	Description
NDB_MGM_EVENT_CATEGORY_CONNECTION		(Connection events)
	NDB_LE_Connected	The node has connected
	NDB_LE_Disconnected	The node was disconnected
	NDB_LE_CommunicationClosed	Communication with the node has been closed
	NDB_LE_CommunicationOpened	Communication with the node has been started
	NDB_LE_ConnectedApiVersion	The API version used by an API node; in the case of a MySQL server (SQL node), this is the same as displayed by SELECT VERSION()
NDB_MGM_EVENT_CATEGORY_CHECKPOINT		(Checkpoint events)
	NDB_LE_GlobalCheckpointStarted	A global checkpoint has been started
	NDB_LE_GlobalCheckpointCompleted	A global checkpoint has been completed
	NDB_LE_LocalCheckpointStarted	The node has begun a local checkpoint
	NDB_LE_LocalCheckpointCompleted	The node has completed a local checkpoint
	NDB_LE_LCPStoppedInCalcKeepGci	The lcoal checkpoint was aborted, but the last global checkpoint was preserved
	NDB_LE_LCPFragmentCompleted	Copying of a table fragment was completed
NDB_MGM_EVENT_CATEGORY_STARTUP		(Startup events)

Category	Type	Description
	NDB_LE_NDBStartStarted	The node has begun to start
	NDB_LE_NDBStartCompleted	The node has completed the startup process
	NDB_LE_STTORRYRecieved	The node received an STTORRY signal, indicating that the reading of configuration data is underway; see Section 5.5.2, "Configuration Read Phase (STTOR Phase -1)", and Section 5.5.3, "STTOR Phase 0", for more information
	NDB_LE_StartPhaseCompleted	A node start phase has been completed
	NDB_LE_CM_REGCONF	The node has received a CM_REGCONF signal; see Section 5.5.4, "STTOR Phase 1", for more information
	NDB_LE_CM_REGREF	The node has received a CM_REGREF signal; see Section 5.5.4, "STTOR Phase 1", for more information
	NDB_LE_FIND_NEIGHBOURS	The node has discovered its neighboring nodes in the cluster
	NDB_LE_NDBStopStarted	The node is beginning to shut down
	NDB_LE_NDBStopCompleted	
	NDB_LE_NDBStopForced	The node is being forced to shut down (usually indicates a severe problem in the cluster)
	NDB_LE_NDBStopAborted	The started to shut down, but was forced to continue running; this happens, for example, when a STOP command was issued in the management client for a node such that the cluster would no longer be able to keep all data available if the node were shut down
	NDB_LE_StartREDOLog	Redo logging has been started
	NDB_LE_StartLog	Logging has started
	NDB_LE_UNDORecordsExecuted	The node has read and executed all records from the redo log
	NDB_LE_StartReport	The node is issuing a start report
NDB_MGM_EVENT_CATEGORY_NODE_RESTART		(Restart events)
	NDB_LE_NR_CopyDict	The node is copying the data dictionary
	NDB_LE_NR_CopyDistr	The node is copying data distribution information
	NDB_LE_NR_CopyFragsStarted	The node is copying table fragments
	NDB_LE_NR_CopyFragDone	The node has completed copying a table fragment
	NDB_LE_NR_CopyFragsCompleted	The node has completed copying all necessary table fragments
NDB_MGM_EVENT_CATEGORY_NODE_RESTART		(Node failure and arbitration)
	NDB_LE_NodeFailCompleted	All (remaining) nodes has been notified of the failure of a data node
	NDB_LE_NODE_FAILREP	A data node has failed
	NDB_LE_ArbitState	This event is used to report on the current state of arbitration in the

Category	Type	Description
		cluster
	NDB_LE_ArbitResult	This event is used to report on the outcome of node arbitration
	NDB_LE_GCP_TakeoverStarted	The node is attempting to become the master node (to assume responsibility for GCPs)
	NDB_LE_GCP_TakeoverCompleted	The node has become the master (and assumed responsibility for GCPs)
	NDB_LE_LCP_TakeoverStarted	The node is attempting to become the master node (to assume responsibility for LCPs)
	NDB_LE_LCP_TakeoverCompleted	The node has become the master (and assumed responsibility for LCPs)
NDB_MGM_EVENT_CATEGORY_STATISTIC		(Statistics events)
	NDB_LE_TransReportCounters	This indicates a report of transaction activity, which is given approximately once every 10 seconds
	NDB_LE_OperationReportCounters	Indicates a report on the number of operations performed by this node (also provided approximately once every 10 seconds)
	NDB_LE_TableCreated	A new table has been created
	NDB_LE_UndoLogBlocked	Undo logging is blocked because the log buffer is close to overflowing
	NDB_LE_JobStatistic	
	NDB_LE_SendBytesStatistic	Indicates a report of the average number of bytes transmitted per send operation by this node
	NDB_LE_ReceiveBytesStatistic	Indicates a report of the average number of bytes received per send operation to this node
	NDB_LE_MemoryUsage	A DUMP 1000 command has been issued to this node, and it is reporting its memory usage in turn
NDB_MGM_EVENT_CATEGORY_ERROR		(Errors and warnings)
	NDB_LE_TransporterError	A transporter error has occurred; see Section 4.4, “NDB Transporter Errors”, for transporter error codes and messages
	NDB_LE_TransporterWarning	A potential problem is occurring in the transporter; see Section 4.4, “NDB Transporter Errors”, for transporter error codes and messages
	NDB_LE_MissedHeartbeat	Indicates a data node has missed a hreatbeat expected from another data node
	NDB_LE_DeadDueToHeartbeat	A data node has missed at least 3 heartbeats in succssion from another data node, and is reporting that it can no longer communicate with that data node
	NDB_LE_WarningEvent	Indicates a warning message
NDB_MGM_EVENT_CATEGORY_INFO		(Information events)
	NDB_LE_SentHeartbeat	A node heartbeat has been sent
	NDB_LE_CreateLogBytes	
	NDB_LE_InfoEvent	Indicates an informational message

Category	Type	Description
[Single-User Mode]	NDB_LE_SingleUser	The cluster has entered or exited single user mode
	NDB_LE_EventBufferStatus	This type of event indicates potentially excessive usage of the event buffer
NDB_MGM_EVENT_CATEGORY_BACKUP		(Backups)
	NDB_LE_BackupStarted	A backup has been started
	NDB_LE_BackupFailedToStart	A backup has failed to start
	NDB_LE_BackupCompleted	A backup has been completed successfully
	NDB_LE_BackupAborted	A backup in progress was terminated by the user

3.3.5. The ndb_mgm_event_severity Type

Description. These are the log event severities used to filter the cluster log by ndb_mgm_set_clusterlog_severity_filter(), and to filter listening to events by ndb_mgm_listen_event().

Enumeration values.

Value	Description
NDB_MGM_ILLEGAL_EVENT_SEVERITY	Invalid event severity specified
NDB_MGM_EVENT_SEVERITY_ON	Cluster logging is enabled
NDB_MGM_EVENT_SEVERITY_DEBUG	*Used for MySQL Cluster development only*
NDB_MGM_EVENT_SEVERITY_INFO	Informational messages
NDB_MGM_EVENT_SEVERITY_WARNING	Conditions that are not errors as such, but that might require special handling
NDB_MGM_EVENT_SEVERITY_ERROR	Non-fatal error conditions that should be corrected
NDB_MGM_EVENT_SEVERITY_CRITICAL	Critical conditions such as device errors or out of memory errors
NDB_MGM_EVENT_SEVERITY_ALERT	Conditions that require immediate attention, such as corruption of the cluster
NDB_MGM_EVENT_SEVERITY_ALL	All severity levels

See Section 3.2.7.2, "ndb_mgm_set_clusterlog_severity_filter()", and Section 3.2.1.1, "ndb_mgm_listen_event()", for information on how this type is used by those functions.

3.3.6. The ndb_logevent_handle_error Type

Description. This type is used to describe log event errors.

Enumeration values.

Value	Description
NDB_LEH_NO_ERROR	No error
NDB_LEH_READ_ERROR	Read error
NDB_LEH_MISSING_EVENT_SPECIFIER	Invalid, incomplete, or missing log event specification
NDB_LEH_UNKNOWN_EVENT_TYPE	Unknown log event type
NDB_LEH_UNKNOWN_EVENT_VARIABLE	Unknown log event variable
NDB_LEH_INTERNAL_ERROR	Internal error

3.3.7. The ndb_mgm_event_category Type

Description. These are the log event categories referenced in Section 3.3.4, "The Ndb_logevent_type Type". They are also used by the MGM API functions ndb_mgm_set_clusterlog_loglevel() and ndb_mgm_listen_event().

Enumeration values.

Value	Description
NDB_MGM_ILLEGAL_EVENT_CATEGORY	Invalid log event category
NDB_MGM_EVENT_CATEGORY_STARTUP	Log events occurring during startup
NDB_MGM_EVENT_CATEGORY_SHUTDOWN	Log events occurring during shutdown
NDB_MGM_EVENT_CATEGORY_STATISTIC	Statistics log events
NDB_MGM_EVENT_CATEGORY_CHECKPOINT	Log events related to checkpoints
NDB_MGM_EVENT_CATEGORY_NODE_RESTART	Log events occurring during node restart
NDB_MGM_EVENT_CATEGORY_CONNECTION	Log events relating to connections between cluster nodes
NDB_MGM_EVENT_CATEGORY_BACKUP	Log events relating to backups
NDB_MGM_EVENT_CATEGORY_CONGESTION	Log events relating to congestion
NDB_MGM_EVENT_CATEGORY_INFO	Uncategorised log events (severity level INFO)
NDB_MGM_EVENT_CATEGORY_ERROR	Uncategorised log events (severity level WARNING, ERROR, CRITICAL, or ALERT)

See Section 3.2.7.4, “ndb_mgm_set_clusterlog_loglevel()”, and Section 3.2.1.1, “ndb_mgm_listen_event()”, for more information.

3.4. MGM Structures

This section covers the programming structures available in the MGM API.

3.4.1. The ndb_logevent Structure

Description. This structure models a Cluster log event, and is used for storing and retrieving log event information.

Definition. ndb_logevent has 8 members, the first 7 of which are shown in the following list:

- void* *handle*: An NdbLogEventHandle, set by ndb_logevent_get_next(). This handle is used only for purposes of comparison.

  See Section 3.2.1.5, “ndb_logevent_get_next()”.

- enum Ndb_logevent_type *type*: Tells which type of event this is.

  See Section 3.3.4, “The Ndb_logevent_type Type”, for possible values.

- unsigned *time*: The time at which the log event was registered with the management server.

- enum ndb_mgm_event_category *category*: The log event category.

  See Section 3.3.7, “The ndb_mgm_event_category Type”, for possible values.

- enum ndb_mgm_event_severity *severity*: The log event severity.

  See Section 3.3.5, “The ndb_mgm_event_severity Type”, for possible values.

- unsigned *level*: The log event level. This is a value in the range of 0 to 15, inclusive.

- unsigned *source_nodeid*: The node ID of the node that reported this event.

The 8th member of this structure contains data specific to the log event, and is dependent on its type. It is defined as the union of a number of data structures, each corresponding to a log event type. Which structure to use is determined by the value of *type*, and is shown in the following table:

Ndb_logevent_type Value	Structure
NDB_LE_Connected	Connected: `unsigned node`
NDB_LE_Disconnected	Disconnected:

Ndb_logevent_type Value	Structure
	`unsigned node`
NDB_LE_CommunicationClosed	CommunicationClosed: `unsigned node`
NDB_LE_CommunicationOpened	CommunicationOpened: `unsigned node`

NDB_LE_ConnectedApiVersion	ConnectedApiVersion:
	`unsigned node`
	`unsigned version`

NDB_LE_GlobalCheckpointStarted	GlobalCheckpointStarted: `unsigned gci`
NDB_LE_GlobalCheckpointCompleted	GlobalCheckpointCompleted: `unsigned gci`

NDB_LE_LocalCheckpointStarted	LocalCheckpointStarted:
	`unsigned lci`
	`unsigned keep_gci`
	`unsigned restore_gci`

NDB_LE_LocalCheckpointCompleted	LocalCheckpointCompleted: `unsigned lci`
NDB_LE_LCPStoppedInCalcKeepGci	LCPStoppedInCalcKeepGci: `unsigned data`

NDB_LE_LCPFragmentCompleted	LCPFragmentCompleted:
	`unsigned node`
	`unsigned table_id`
	`unsigned fragment_id`

NDB_LE_UndoLogBlocked	UndoLogBlocked:
	`unsigned acc_count`
	`unsigned tup_count`

NDB_LE_NDBStartStarted	NDBStartStarted: `unsigned version`
NDB_LE_NDBStartCompleted	NDBStartCompleted: `unsigned version`
NDB_LE_STTORRYRecieved	STTORRYRecieved: *[NONE]*

NDB_LE_StartPhaseCompleted	StartPhaseCompleted:
	`unsigned phase`
	`unsigned starttype`

NDB_LE_CM_REGCONF	CM_REGCONF:
	`unsigned own_id`
	`unsigned president_id`
	`unsigned dynamic_id`

NDB_LE_CM_REGREF	CM_REGREF:

Ndb_logevent_type Value	**Structure**

	`unsigned own_id`
	`unsigned other_id`
	`unsigned cause`

NDB_LE_FIND_NEIGHBOURS	FIND_NEIGHBOURS:
	`unsigned own_id`
	`unsigned left_id`
	`unsigned right_id`
	`unsigned dynamic_id`

NDB_LE_NDBStopStarted	NDBStopStarted: `unsigned stoptype`

NDB_LE_NDBStopCompleted	NDBStopCompleted:
	`unsigned action`
	`unsigned signum`

NDB_LE_NDBStopForced	NDBStopForced:
	`unsigned action`
	`unsigned signum`
	`unsigned error`
	`unsigned sphase`
	`unsigned extra`

NDB_LE_NDBStopAborted	NDBStopAborted: *[NONE]*

NDB_LE_StartREDOLog	StartREDOLog:
	`unsigned node`
	`unsigned keep_gci`
	`unsigned completed_gci`
	`unsigned restorable_gci`

NDB_LE_StartLog	StartLog:
	`unsigned log_part`
	`unsigned start_mb`
	`unsigned stop_mb`
	`unsigned gci`

NDB_LE_UNDORecordsExecuted	UNDORecordsExecuted:
	`unsigned block`
	`unsigned data1`
	`unsigned data2`
	`unsigned data3`
	`unsigned data4`
	`unsigned data5`
	`unsigned data6`
	`unsigned data7`
	`unsigned data8`
	`unsigned data9`
	`unsigned data10`

NDB_LE_NR_CopyDict	NR_CopyDict: *[NONE]*
NDB_LE_NR_CopyDistr	NR_CopyDistr: *[NONE]*
NDB_LE_NR_CopyFragsStarted	NR_CopyFragsStarted: `unsigned dest_node`

NDB_LE_NR_CopyFragDone	NR_CopyFragDone:
	`unsigned dest_node`
	`unsigned table_id`
	`unsigned fragment_id`

Ndb_logevent_type Value	**Structure**
NDB_LE_NR_CopyFragsCompleted	NR_CopyFragsCompleted: `unsigned dest_node`

NDB_LE_NodeFailCompleted	NodeFailCompleted:
	`unsigned block`
	`unsigned failed_node`
	`unsigned completing_node`
	(For `block` and `completing_node`, `0` is interpreted as "all".)

NDB_LE_NODE_FAILREP	NODE_FAILREP:
	`unsigned failed_node`
	`unsigned failure_state`

NDB_LE_ArbitState	ArbitState:
	`unsigned code`
	`unsigned arbit_node`
	`unsigned ticket_0`
	`unsigned ticket_1`

NDB_LE_ArbitResult	ArbitResult:
	`unsigned code`
	`unsigned arbit_node`
	`unsigned ticket_0`
	`unsigned ticket_1`

NDB_LE_GCP_TakeoverStarted	GCP_TakeoverStarted: `[NONE]`
NDB_LE_GCP_TakeoverCompleted	GCP_TakeoverCompleted: `[NONE]`
NDB_LE_LCP_TakeoverStarted	LCP_TakeoverStarted: `[NONE]`

NDB_LE_TransReportCounters	TransReportCounters:
	`unsigned trans_count`
	`unsigned commit_count`
	`unsigned read_count`
	`unsigned simple_read_count`
	`unsigned write_count`
	`unsigned attrinfo_count`
	`unsigned conc_op_count`
	`unsigned abort_count`
	`unsigned scan_count`
	`unsigned range_scan_count`

NDB_LE_OperationReportCounters	OperationReportCounters: `unsigned ops`
NDB_LE_TableCreated	TableCreated: `unsigned table_id`
NDB_LE_JobStatistic	JobStatistic: `unsigned mean_loop_count`

NDB_LE_SendBytesStatistic	SendBytesStatistic:
	`unsigned to_node`
	`unsigned mean_sent_bytes`

NDB_LE_ReceiveBytesStatistic	ReceiveBytesStatistic: `unsigned from_node`

Ndb_logevent_type Value	**Structure**
	`unsigned mean_received_bytes`

NDB_LE_MemoryUsage	MemoryUsage:
	`int gth`
	`unsigned page_size_kb`
	`unsigned pages_used`
	`unsigned pages_total`
	`unsigned block`

NDB_LE_TransporterError	TransporterError:
	`unsigned to_node`
	`unsigned code`

NDB_LE_TransporterWarning	TransporterWarning:
	`unsigned to_node`
	`unsigned code`

NDB_LE_MissedHeartbeat	MissedHeartbeat:
	`unsigned node`
	`unsigned count`

NDB_LE_DeadDueToHeartbeat	DeadDueToHeartbeat: `unsigned node`
NDB_LE_WarningEvent	WarningEvent: *[NOT YET IMPLEMENTED]*
NDB_LE_SentHeartbeat	SentHeartbeat: `unsigned node`
NDB_LE_CreateLogBytes	CreateLogBytes: `unsigned node`
NDB_LE_InfoEvent	InfoEvent: *[NOT YET IMPLEMENTED]*

NDB_LE_EventBufferStatus	EventBufferStatus:
	`unsigned usage`
	`unsigned alloc`
	`unsigned max`
	`unsigned apply_gci_l`
	`unsigned apply_gci_h`
	`unsigned latest_gci_l`
	`unsigned latest_gci_h`

NDB_LE_BackupStarted	BackupStarted:
	`unsigned starting_node`
	`unsigned backup_id`

NDB_LE_BackupFailedToStart	BackupFailedToStart:
	`unsigned starting_node`
	`unsigned error`

NDB_LE_BackupCompleted	BackupCompleted:
	`unsigned starting_node`
	`unsigned backup_id`
	`unsigned start_gci`
	`unsigned stop_gci`
	`unsigned n_records`
	`unsigned n_log_records`
	`unsigned n_bytes`
	`unsigned n_log_bytes`

`Ndb_logevent_type` Value	Structure

`NDB_LE_BackupAborted`	`BackupAborted`:
	`unsigned starting_node`
	`unsigned backup_id`
	`unsigned error`

`NDB_LE_SingleUser`	`SingleUser`:
	`unsigned type`
	`unsigned node_id`

`NDB_LE_StartReport`	`StartReport`:
	`unsigned report_type`
	`unsigned remaining_time`
	`unsigned bitmask_size`
	`unsigned bitmask_data[1]`

3.4.2. The `ndb_mgm_node_state` Structure

Description. Provides information on the status of a Cluster node.

Definition. This structure contains the following members:

- `int node_id`: The cluster node's node ID.
- `enum ndb_mgm_node_type node_type`: The node type.

  See Section 3.3.1, "The `ndb_mgm_node_type` Type", for permitted values.
- `enum ndb_mgm_node_status node_status`: The node's status.

  See Section 3.3.2, "The `ndb_mgm_node_status` Type", for permitted values.
- `int start_phase`: The start phase.

  This is valid only if the `node_type` is `NDB_MGM_NODE_TYPE_NDB` and the `node_status` is `NDB_MGM_NODE_STATUS_STARTING`.
- `int dynamic_id`: The ID for heartbeats and master takeover.

  Valid only for data (`ndbd`) nodes.
- `int node_group`: The node group to which the node belongs.

  Valid only for data (`ndbd`) nodes.
- `int version`: Internal version number.
- `int connect_count`: The number of times this node has connected to or disconnected from the management server.
- `char connect_address[]`: The IP address of the node when it connected to the management server.

  This value will be empty if the management server has been restarted since the node last connected.

3.4.3. The `ndb_mgm_cluster_state` Structure

Description. Provides information on the status of all Cluster nodes. This structure is returned by `ndb_mgm_get_status()`.

Definition. This structure has the following two members;

- `int no_of_nodes`: The number of elements in the `node_states` array.
- `struct ndb_mgm_node_state node_states[]`: An array containing the states of the nodes.

  Each element of this array is an `ndb_mgm_node_state` structure. For more information, see Section 3.4.2, "The

`ndb_mgm_node_state` Structure”.

See Section 3.2.5.1, “`ndb_mgm_get_status()`”.

3.4.4. The `ndb_mgm_reply` Structure

Description. Contains response information, consisting of a response code and a corresponding message, from the management server.

Definition. This structure contains two members, as shown here:

- `int return_code`: For a successful operation, this value is `0`; otherwise, it contains an error code.

  For error codes, see Section 3.3.3, “The `ndb_mgm_error` Type”.

- `char message[256]`: contains the text of the response or error message.

See Section 3.2.2.1, “`ndb_mgm_get_latest_error()`”, and Section 3.2.2.2, “`ndb_mgm_get_latest_error_msg()`”.

Chapter 4. MySQL Cluster API Errors

This chapter discusses reporting and handling of errors potentially generated in MySQL Cluster API applications. It includes information about error codes, classifications, and messages for the MGM API (see Section 3.3.3, "The `ndb_mgm_error` Type") and NDB API (see Section 4.2, "NDB API Errors and Error Handling"). Also provided in this chapter is a listing of exit codes and messages returned by a failed `ndbd` process, in Section 4.2, "NDB API Errors and Error Handling".

4.1. MGM API Errors

The following sections list the values of `MGM` errors by type. There are six types of `MGM` errors:

1. request errors
2. node ID allocation errors
3. service errors
4. backup errors
5. single user mode errors
6. general usage errors

There is only one general usage error.

4.1.1. Request Errors

These are errors generated by failures to connect to a management server.

Value	Description
NDB_MGM_ILLEGAL_CONNECT_STRING	Invalid connectstring
NDB_MGM_ILLEGAL_SERVER_HANDLE	Invalid management server handle
NDB_MGM_ILLEGAL_SERVER_REPLY	Invalid response from management server
NDB_MGM_ILLEGAL_NUMBER_OF_NODES	Invalid number of nodes
NDB_MGM_ILLEGAL_NODE_STATUS	Invalid node status
NDB_MGM_OUT_OF_MEMORY	Memory allocation error
NDB_MGM_SERVER_NOT_CONNECTED	Management server not connected
NDB_MGM_COULD_NOT_CONNECT_TO_SOCKET	Not able to connect to socket

4.1.2. Node ID Allocation Errors

These errors result from a failure to assign a node ID to a cluster node.

Value	Description
NDB_MGM_ALLOCID_ERROR	Generic error; may be possible to retry and recover
NDB_MGM_ALLOCID_CONFIG_MISMATCH	Non-recoverable generic error

4.1.3. Service Errors

These errors result from the failure of a node or cluster to start, shut down, or restart.

Value	Description
NDB_MGM_START_FAILED	Startup failure
NDB_MGM_STOP_FAILED	Shutdown failure
NDB_MGM_RESTART_FAILED	Restart failure

4.1.4. Backup Errors

These are errors which result from problems with initiating or aborting backups.

Value	Description
NDB_MGM_COULD_NOT_START_BACKUP	UNABLE TO INITIATE BACKUP
NDB_MGM_COULD_NOT_ABORT_BACKUP	UNABLE TO ABORT BACKUP

4.1.5. Single User Mode Errors

These errors result from failures to enter or exit single user mode.

Value	Description
NDB_MGM_COULD_NOT_ENTER_SINGLE_USER_MODE	UNABLE TO ENTER SINGLE-USER MODE
NDB_MGM_COULD_NOT_EXIT_SINGLE_USER_MODE	UNABLE TO EXIT SINGLE-USER MODE

4.1.6. General Usage Errors

This is a general error type for errors which are otherwise not classifiable.

Value	Description
NDB_MGM_USAGE_ERROR	GENERAL USAGE ERROR

4.2. NDB API Errors and Error Handling

This section contains a discussion of error handling in NDB API applications as well as listing listings of the most common NDB error codes and messages, along with their classifications and likely causes for which they might be raised.

For information about the NdbError structure, which is used to convey error information to NDB API applications, see Section 2.3.30, “The NdbError Structure”.

> **Important**
>
> It is strongly recommended that you *not* depend on specific error codes in your NDB API applications, as they are subject to change over time. Instead, you should use the NdbError::Status and error classification in your source code, or consult the output of perror --ndb *error_code* to obtain information about a specific error code.
>
> If you find a situation in which you need to use a specific error code in your application, please file a bug report at http://bugs.mysql.com/ so that we can update the corresponding status and classification.

4.2.1. Handling NDB API Errors

This section describes how NDB API errors can be detected and mapped onto particular operations.

NDB API errors can be generated in either of two ways:

- When an operation is defined
- When an operation is executed

Errors raised during operation definition. Errors generated during operation definition result in a failure return code from the method called. The actual error can be determined by examining the relevant NdbOperation object, or the operation's NdbTransaction object.

Errors raised during operation execution. Errors occurring during operation execution cause the transaction of which they are a part to be aborted unless the AO_IgnoreError abort option is set for the operation.

> **Important**
>
> If you have worked with older versions of the NDB API, you should be aware that, beginning with MySQL Cluster

NDB 6.2.0, the `AbortOption` type is a member of `NdbOperation`. See Section 2.3.15.1.1, "The `NdbOperation::AbortOption` Type", for more information.

By default, read operations are run with `AO_IgnoreError`, and write operations are run with `AbortOnError`, but this can be overridden by the user. When an error during execution causes a transaction to be aborted, the `execute()` method returns a failure return code. If an error is ignored due to `AO_IgnoreError` being set on the operation, the `execute()` method returns a success code, and the user must examine all operations for failure using `NdbOperation::getNdbError()`. For this reason, the return value of `getNdbError()` should usually be checked, even if `execute()` returns success. If the client application does not keep track of `NdbOperation` objects during execution, then `NdbTransaction::getNextCompletedOperation()` can be used to iterate over them.

You should also be aware that use of `NdbBlob` can result in extra operations being added to the batches executed. This means that, when iterating over completed operations using `getNextCompletedOperation()`, you may encounter operations related to `NdbBlob` objects which were not defined by your application.

Note

A read whose `LockMode` is `CommittedRead` cannot be `AbortOnError`. In this case, it is always be `IgnoreError`.

In all cases where operation-specific errors arise, an execution error with an operation is marked against both the operation and the associated transaction object. Where there are multiple operation errors in a single `NdbTransaction::execute()` call, due to operation batching and the use of `AO_IgnoreError`, only the first is marked against the `NdbTransaction` object. The remaining errors are recorded against the corresponding `NdbOperation` objects only.

It is also possible for errors to occur during execution — such as a data node failure — which are marked against the transaction object, but *not* against the underlying operation objects. This is because these errors apply to the transaction as a whole, and not to individual operations within the transaction.

For this reason, applications should use `NdbTransaction::getNdbError()` as the first way to determine whether an `NdbTransaction::execute()` call failed. If the batch of operations being executed included operations with the `AO_IgnoreError` abort option set, then it is possible that there were multiple failures, and the completed operations should be checked individually for errors using `NdbOperation::getNdbError()`.

Implicit `NdbTransaction::execute()` calls in scan and `BLOB` methods. Scan operations are executed in the same way as other operations, and also have implicit `execute()` calls within the `NdbScanOperation::nextResult()` method. When `NdbScanOperation::nextResult()` indicates failure (that is, if the method returns `-1`), the transaction object should be checked for an error. The `NdbScanOperation` may also contain the error, but only if the error is not operation-specific.

Some `BLOB` manipulation methods also have implicit internal `execute()` calls, and so can experience operation execution failures at these points. The following `NdbBlob` methods can generate implicit `execute()` calls; this means that they also require checks of the `NdbTransaction` object for errors via `NdbTransaction::getNdbError()` if they return an error code:

- `setNull()`
- `truncate()`
- `readData()`
- `writeData()`

Summary. In general, it is possible for an error to occur during execution (resulting in a failure return code) when calling any of the following methods:

- `NdbTransaction::execute()`
- `NdbBlob::setNull()`
- `NdbBlob::truncate()`
- `NdbBlob::readData()`
- `NdbBlob::writeData()`
- `NdbScanOperation::nextResult()`

Note

This method does *not* perform an implicit `execute()` call. The `NdbBlob` methods can cause other defined opera-

tions to be executed when these methods are called; however, `nextResult()` calls do not do so.

If this happens, the `NdbTransaction::getNdbError()` method should be called to identify the first error that occurred. When operations are batched, and there are `IgnoreError` operations in the batch, there may be multiple operations with errors in the transaction. These can be found by using `NdbTransaction::getNextCompletedOperation()` to iterate over the set of completed operations, calling `NdbOperation::getNdbError()` for each operation.

When `IgnoreError` has been set on any operations in a batch of operations to be executed, the `NdbTransaction::execute()` method indicates success even where errors have actually occurred, as long as none of these errors caused a transaction to be aborted. To determine whether there were any ignored errors, the transaction error status should be checked using `NdbTransaction::getNdbError()`. *Only if this indicates success can you be certain that no errors occurred*. If an error code is returned by this method, and operations were batched, then you should iterate over all completed operations to find all the operations with ignored errors.

Example (*pseudocode*). We begin by executing a transaction which may have batched operations and a mix of `AO_IgnoreError` and `AbortOnError` abort options:

```
int execResult= NdbTransaction.execute(args);
```

Note

For the number and permitted values of *`args`*, see Section 2.3.19.2.5, “`NdbTransaction::execute()`”.

Next, because errors on `AO_IgnoreError` operations do not affect execResult — that is, the value returned by `execute()` — we check for errors on the transaction:

```
NdbError err= NdbTransaction.getNdbError();

if (err.code != 0)
{
```

An nonzero value for the error code means that an error was raised on the transaction. This could be due to any of the following conditions:

- A transaction-wide error, such as a data node failure, that caused the transaction to be aborted
- A single operation-specific error, such as a constraint violation, that caused the transaction to be aborted
- A single operation-specific ignored error, such as no data found, that did not cause the transaction to be aborted
- The first of many operation-specific ignored errors, such as no data found when batching, that did not cause the transaction to be aborted
- First of a number of operation-specific ignored errors such as no data found (when batching) before an aborting operation error (transaction aborted)

```
  if (execResult != 0)
  {
```

The transaction has been aborted. The recommended strategy for handling the error in this case is to test the transaction error status and take appropriate action based on its value:

```
    switch (err.status)
    {
      case value1:
        //  statement block handling value1 ...
      case value2:
        //  statement block handling value2 ...
        //  (etc. ...)
      case valueN:
        //  statement block handling valueN ...
    }
```

Since the transaction was aborted, it is generally necessary to iterate over the completed operations (if any) and find the errors raised by each only if you wish to do so for reporting purposes.

```
  }
  else
  {
```

The transaction itself was not aborted, but there must be one or more ignored errors. In this case, you should iterate over the operations to determine what happened and handle the cause accordingly.

```
  }
```

```
}
```

To handle a `NdbScanOperation::nextResult()` which returns `-1`, indicating that the operation failed (omitting cases where the operation was successful):

```
int nextrc= NdbScanOperation.nextResult(args);
```

> **Note**
>
> For the number and permitted values of *`args`*, see Section 2.3.18.2.2, "`NdbScanOperation::nextResult()`".

```
if (nextrc == -1)
{
```

First, you should check the `NdbScanOperation` object for any errors:

```
  NdbError err= NdbScanOperation.getNdbError();

  if (err.code == 0)
  {
```

No error was found in the scan operation; the error must belong to the transaction as whole.

```
  }
    err= NdbTransaction.getNdbError();
```

Now you can handle the error based on the error status:

```
    switch (err.status)
    {
      case value1:
        //  statement block handling value1 ...
      case value2:
        //  statement block handling value2 ...
        //  (etc. ...)
      case valueN:
        //  statement block handling valueN ...
    }
}
```

For information about NDB API error classification and status codes, see Section 4.2.3, "NDB Error Classifications". While you should not rely on a specific error code or message text in your NDB API applications — since error codes and messages are both subject to change over time — it can be useful to check error codes and messages to help determine why a particular failure occurred. For more information about these, see Section 4.2.2, "NDB Error Codes and Messages". For more about `NdbError` and the types of information which can be obtained from `NdbError` objects, see Section 2.3.30, "The `NdbError` Structure".

4.2.2. NDB Error Codes and Messages

This section contains a number of tables, one for each type of NDB API error. The error types include the following:

• No error • Application error • Scan application error • Configuration or application error (currently unused) • No data found • Constraint violation	• Schema error • User defined error • Insufficient space • Temporary Resource error • Node Recovery error • Overload error • Timeout expired	• Node shutdown • Internal temporary • Unknown result error • Unknown error code (currently unused) • Internal error • Function not implemented

The information in each table includes, for each error:

- The numeric NDB error code

- The corresponding MySQL error code
- The NDB classification code

  See Section 4.2.3, “NDB Error Classifications”, for the meanings of these classification codes.
- The text of the error message

Similar errors have been grouped together in each table.

> **Note**
>
> You can always obtain the latest error codes and information from the file `storage/ndb/src/ndbapi/ndberror.c`.

4.2.2.1. *No error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
0	**0**	*NE*	No error

4.2.2.2. *Application error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
281	**HA_ERR_NO_CONNECTION**	*AE*	Operation not allowed due to cluster shutdown in progress
299	**DMEC**	*AE*	Operation not allowed or aborted due to single user mode
763	**DMEC**	*AE*	Alter table requires cluster nodes to have exact same version
823	**DMEC**	*AE*	Too much attrinfo from application in tuple manager
831	**DMEC**	*AE*	Too many nullable/bitfields in table definition
876	**DMEC**	*AE*	876
877	**DMEC**	*AE*	877
878	**DMEC**	*AE*	878
879	**DMEC**	*AE*	879
880	**DMEC**	*AE*	Tried to read too much - too many getValue calls
884	**DMEC**	*AE*	Stack overflow in interpreter
885	**DMEC**	*AE*	Stack underflow in interpreter
886	**DMEC**	*AE*	More than 65535 instructions executed in interpreter
897	**DMEC**	*AE*	Update attempt of primary key via ndbcluster internal api (if this occurs via the MySQL server it is a bug, please report)
892	**DMEC**	*AE*	Unsupported type in scan filter
4256	**DMEC**	*AE*	Must call Ndb::init() before this function
4257	**DMEC**	*AE*	Tried to read too much - too many getValue calls
320	**DMEC**	*AE*	Invalid no of nodes specified for new nodegroup
321	**DMEC**	*AE*	Invalid nodegroup id

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
322	DMEC	AE	Invalid node(s) specified for new node-group, node already in nodegroup
323	DMEC	AE	Invalid nodegroup id, nodegroup already existing
324	DMEC	AE	Invalid node(s) specified for new node-group, no node in nodegroup is started
242	DMEC	AE	Zero concurrency in scan
244	DMEC	AE	Too high concurrency in scan
269	DMEC	AE	No condition and attributes to read in scan
874	DMEC	AE	Too much attrinfo (e.g. scan filter) for scan in tuple manager
4600	DMEC	AE	Transaction is already started
4601	DMEC	AE	Transaction is not started
4602	DMEC	AE	You must call getNdbOperation before ex-ecuteScan
4603	DMEC	AE	There can only be ONE operation in a scan transaction
4604	DMEC	AE	takeOverScanOp, to take over a scanned row one must explicitly request keyinfo on readTuples call
4605	DMEC	AE	You may only call readTuples() once for each operation
4607	DMEC	AE	There may only be one operation in a scan transaction
4608	DMEC	AE	You can not takeOverScan unless you have used openScanExclusive
4609	DMEC	AE	You must call nextScanResult before try-ing to takeOverScan
4232	DMEC	AE	Parallelism can only be between 1 and 240
4707	DMEC	AE	Too many event have been defined
4708	DMEC	AE	Event name is too long
4709	DMEC	AE	Can't accept more subscribers
4710	DMEC	AE	Event not found
4711	DMEC	AE	Creation of event failed
4712	DMEC	AE	Stopped event operation does not exist. Already stopped?
311	DMEC	AE	Undefined partition used in setPartitionId
771	HA_WRONG_CREATE_OPTION	AE	Given NODEGROUP doesn't exist in this cluster
776	DMEC	AE	Index created on temporary table must it-self be temporary
777	DMEC	AE	Cannot create a temporary index on a non-temporary table
778	DMEC	AE	A temporary table or index must be spe-cified as not logging
1306	DMEC	AE	Backup not supported in diskless mode (change Diskless)
1342	DMEC	AE	Backup failed to allocate buffers (check configuration)
1343	DMEC	AE	Backup failed to setup fs buffers (check

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			CONFIGURATION)
1344	DMEC	AE	BACKUP FAILED TO ALLOCATE TABLES (CHECK CONFIGURATION)
1345	DMEC	AE	BACKUP FAILED TO INSERT FILE HEADER (CHECK CONFIGURATION)
1346	DMEC	AE	BACKUP FAILED TO INSERT TABLE LIST (CHECK CONFIGURATION)
1347	DMEC	AE	BACKUP FAILED TO ALLOCATE TABLE MEMORY (CHECK CONFIGURATION)
1348	DMEC	AE	BACKUP FAILED TO ALLOCATE FILE RECORD (CHECK CONFIGURATION)
1349	DMEC	AE	BACKUP FAILED TO ALLOCATE ATTRIBUTE RECORD (CHECK CONFIGURATION)
1329	DMEC	AE	BACKUP DURING SOFTWARE UPGRADE NOT SUPPORTED
1701	DMEC	AE	NODE ALREADY RESERVED
1702	DMEC	AE	NODE ALREADY CONNECTED
1704	DMEC	AE	NODE TYPE MISMATCH
720	DMEC	AE	ATTRIBUTE NAME REUSED IN TABLE DEFINITION
4004	DMEC	AE	ATTRIBUTE NAME OR ID NOT FOUND IN THE TABLE
4100	DMEC	AE	STATUS ERROR IN NDB
4101	DMEC	AE	NO CONNECTIONS TO NDB AVAILABLE AND CONNECT FAILED
4102	DMEC	AE	TYPE IN NDBTAMPER NOT CORRECT
4103	DMEC	AE	NO SCHEMA CONNECTIONS TO NDB AVAILABLE AND CONNECT FAILED
4104	DMEC	AE	NDB INIT IN WRONG STATE, DESTROY NDB OBJECT AND CREATE A NEW
4105	DMEC	AE	TOO MANY NDB OBJECTS
4106	DMEC	AE	ALL NOT NULL ATTRIBUTE HAVE NOT BEEN DEFINED
4114	DMEC	AE	TRANSACTION IS ALREADY COMPLETED
4116	DMEC	AE	OPERATION WAS NOT DEFINED CORRECTLY, PROBABLY MISSING A KEY
4117	DMEC	AE	COULD NOT START TRANSPORTER, CONFIGURATION ERROR
4118	DMEC	AE	PARAMETER ERROR IN API CALL
4300	DMEC	AE	TUPLE KEY TYPE NOT CORRECT
4301	DMEC	AE	FRAGMENT TYPE NOT CORRECT
4302	DMEC	AE	MINIMUM LOAD FACTOR NOT CORRECT
4303	DMEC	AE	MAXIMUM LOAD FACTOR NOT CORRECT
4304	DMEC	AE	MAXIMUM LOAD FACTOR SMALLER THAN MINIMUM
4305	DMEC	AE	K VALUE MUST CURRENTLY BE SET TO 6
4306	DMEC	AE	MEMORY TYPE NOT CORRECT
4307	DMEC	AE	INVALID TABLE NAME
4308	DMEC	AE	ATTRIBUTE SIZE NOT CORRECT
4309	DMEC	AE	FIXED ARRAY TOO LARGE, MAXIMUM 64000 BYTES
4310	DMEC	AE	ATTRIBUTE TYPE NOT CORRECT
4311	DMEC	AE	STORAGE MODE NOT CORRECT

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4312	DMEC	AE	Null Attribute Type not correct
4313	DMEC	AE	Index only storage for non-key attribute
4314	DMEC	AE	Storage Type of attribute not correct
4315	DMEC	AE	No more key attributes allowed after defining variable length key attribute
4316	DMEC	AE	Key attributes are not allowed to be NULL attributes
4317	DMEC	AE	Too many primary keys defined in table
4318	DMEC	AE	Invalid attribute name or number
4319	DMEC	AE	createAttribute called at erroneus place
4322	DMEC	AE	Attempt to define distribution key when not prepared to
4323	DMEC	AE	Distribution Key set on table but not defined on first attribute
4324	DMEC	AE	Attempt to define distribution group when not prepared to
4325	DMEC	AE	Distribution Group set on table but not defined on first attribute
4326	DMEC	AE	Distribution Group with erroneus number of bits
4327	DMEC	AE	Distribution Group with 1 byte attribute is not allowed
4328	DMEC	AE	Disk memory attributes not yet supported
4329	DMEC	AE	Variable stored attributes not yet supported
4340	DMEC	AE	Result or attribute record must be a base table NDBRECORD, not an index NDBRECORD
4400	DMEC	AE	Status Error in NdbSchemaCon
4401	DMEC	AE	Only one schema operation per schema transaction
4402	DMEC	AE	No schema operation defined before calling execute
4410	DMEC	AE	Schema transaction is already started
4501	DMEC	AE	Insert in hash table failed when getting table information from Ndb
4502	DMEC	AE	GetValue not allowed in Update operation
4503	DMEC	AE	GetValue not allowed in Insert operation
4504	DMEC	AE	SetValue not allowed in Read operation
4505	DMEC	AE	NULL value not allowed in primary key search
4506	DMEC	AE	Missing getValue/setValue when calling execute
4507	DMEC	AE	Missing operation request when calling execute
4508	DMEC	AE	GetValue not allowed for NdbRecord defined operation
4509	DMEC	AE	Non SF_MultiRange scan cannot have more than one bound
4510	DMEC	AE	User specified partition id not allowed for scan takeover operation
4511	DMEC	AE	Blobs not allowed in NdbRecord delete result record

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4512	DMEC	AE	Incorrect combination of OperationOptions optionsPresent, extraGet/setValues ptr and numExtraGet/setValues
4513	DMEC	AE	Only one scan bound allowed for non-NdbRecord setBound() API
4514	DMEC	AE	Can only call setBound/equal() for an NdbIndexScanOperation
4515	DMEC	AE	Method not allowed for NdbRecord, use OperationOptions or ScanOptions structure instead
4516	DMEC	AE	Illegal instruction in interpreted program
4517	DMEC	AE	Bad label in branch instruction
4518	DMEC	AE	Too many instructions in interpreted program
4519	DMEC	AE	NdbInterpretedCode::finalise() not called
4520	DMEC	AE	Call to undefined subroutine
4521	DMEC	AE	Call to undefined subroutine, internal error
4522	DMEC	AE	setBound() called twice for same key
4523	DMEC	AE	Pseudo columns not supported by NdbRecord
4524	DMEC	AE	NdbInterpretedCode is for different table
4535	DMEC	AE	Attempt to set bound on non key column
4536	DMEC	AE	NdbScanFilter constructor taking NdbOperation is not supported for NdbRecord
4537	DMEC	AE	Wrong API. Use NdbInterpretedCode for NdbRecord operations
4538	DMEC	AE	NdbInterpretedCode instruction requires that table is set
4539	DMEC	AE	NdbInterpretedCode not supported for operation type
4540	DMEC	AE	Attempt to pass an Index column to createRecord. Use base table columns only
4541	DMEC	AE	IndexBound has no bound information
4200	DMEC	AE	Status Error when defining an operation
4201	DMEC	AE	Variable Arrays not yet supported
4202	DMEC	AE	Set value on tuple key attribute is not allowed
4203	DMEC	AE	Trying to set a NOT NULL attribute to NULL
4204	DMEC	AE	Set value and Read/Delete Tuple is incompatible
4205	DMEC	AE	No Key attribute used to define tuple
4206	DMEC	AE	Not allowed to equal key attribute twice
4207	DMEC	AE	Key size is limited to 4092 bytes
4208	DMEC	AE	Trying to read a non-stored attribute
4209	DMEC	AE	Length parameter in equal/setValue is incorrect
4210	DMEC	AE	Ndb sent more info than the length he specified
4211	DMEC	AE	Inconsistency in list of NdbRecAttr-ob-

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			JECTS
4212	DMEC	AE	NDB REPORTS NULL VALUE ON NOT NULL ATTRIBUTE
4213	DMEC	AE	NOT ALL DATA OF AN ATTRIBUTE HAS BEEN RECEIVED
4214	DMEC	AE	NOT ALL ATTRIBUTES HAVE BEEN RECEIVED
4215	DMEC	AE	MORE DATA RECEIVED THAN REPORTED IN TCKEYCONF MESSAGE
4216	DMEC	AE	MORE THAN 8052 BYTES IN SETVALUE CANNOT BE HANDLED
4217	DMEC	AE	IT IS NOT ALLOWED TO INCREMENT ANY OTHER THAN UNSIGNED INTS
4218	DMEC	AE	CURRENTLY NOT ALLOWED TO INCREMENT NULL-ABLE ATTRIBUTES
4219	DMEC	AE	MAXIMUM SIZE OF INTERPRETATIVE ATTRIBUTES ARE 64 BITS
4220	DMEC	AE	MAXIMUM SIZE OF INTERPRETATIVE ATTRIBUTES ARE 64 BITS
4221	DMEC	AE	TRYING TO JUMP TO A NON-DEFINED LABEL
4222	DMEC	AE	LABEL WAS NOT FOUND, INTERNAL ERROR
4223	DMEC	AE	NOT ALLOWED TO CREATE JUMPS TO YOURSELF
4224	DMEC	AE	NOT ALLOWED TO JUMP TO A LABEL IN A DIFFERENT SUBROUTINE
4225	DMEC	AE	ALL PRIMARY KEYS DEFINED, CALL SETVALUE/GETVALUE
4226	DMEC	AE	BAD NUMBER WHEN DEFINING A LABEL
4227	DMEC	AE	BAD NUMBER WHEN DEFINING A SUBROUTINE
4228	DMEC	AE	ILLEGAL INTERPRETER FUNCTION IN SCAN DEFINITION
4229	DMEC	AE	ILLEGAL REGISTER IN INTERPRETER FUNCTION DEFINITION
4230	DMEC	AE	ILLEGAL STATE WHEN CALLING GETVALUE, PROBABLY NOT A READ
4231	DMEC	AE	ILLEGAL STATE WHEN CALLING INTERPRETER ROUTINE
4233	DMEC	AE	CALLING EXECUTE (SYNCHRONOUS) WHEN ALREADY PREPARED ASYNCHRONOUS TRANSACTION EXISTS
4234	DMEC	AE	ILLEGAL TO CALL SETVALUE IN THIS STATE
4235	DMEC	AE	NO CALLBACK FROM EXECUTE
4236	DMEC	AE	TRIGGER NAME TOO LONG
4237	DMEC	AE	TOO MANY TRIGGERS
4238	DMEC	AE	TRIGGER NOT FOUND
4239	DMEC	AE	TRIGGER WITH GIVEN NAME ALREADY EXISTS
4240	DMEC	AE	UNSUPPORTED TRIGGER TYPE
4241	DMEC	AE	INDEX NAME TOO LONG
4242	DMEC	AE	TOO MANY INDEXES
4243	DMEC	AE	INDEX NOT FOUND
4247	DMEC	AE	ILLEGAL INDEX/TRIGGER CREATE/DROP/ALTER REQUEST
4248	DMEC	AE	TRIGGER/INDEX NAME INVALID

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4249	DMEC	AE	Invalid table
4250	DMEC	AE	Invalid index type or index logging option
4251	HA_ERR_FOUND_DUPP_UNIQUE	AE	Cannot create unique index, duplicate keys found
4252	DMEC	AE	Failed to allocate space for index
4253	DMEC	AE	Failed to create index table
4254	DMEC	AE	Table not an index table
4255	DMEC	AE	Hash index attributes must be specified in same order as table attributes
4258	DMEC	AE	Cannot create unique index, duplicate attributes found in definition
4259	DMEC	AE	Invalid set of range scan bounds
4264	DMEC	AE	Invalid usage of blob attribute
4265	DMEC	AE	The method is not valid in current blob state
4266	DMEC	AE	Invalid blob seek position
4335	DMEC	AE	Only one autoincrement column allowed per table. Having a table without primary key uses an autoincremented hidden key, i.e. a table without a primary key can not have an autoincremented column
4271	DMEC	AE	Invalid index object, not retrieved via getIndex()
4272	DMEC	AE	Table definition has undefined column
4275	DMEC	AE	The blob method is incompatible with operation type or lock mode
4276	DMEC	AE	Missing NULL ptr in end of keyData list
4277	DMEC	AE	Key part len is to small for column
4278	DMEC	AE	Supplied buffer to small
4279	DMEC	AE	Malformed string
4280	DMEC	AE	Inconsistent key part length
4281	DMEC	AE	Too many keys specified for key bound in scanIndex
4282	DMEC	AE	range_no not strictly increasing in ordered multi-range index scan
4283	DMEC	AE	key_record in index scan is not an index ndbrecord
4284	DMEC	AE	Cannot mix NdbRecAttr and NdbRecord methods in one operation
4285	DMEC	AE	NULL NdbRecord pointer
4286	DMEC	AE	Invalid range_no (must be < 4096)
4287	DMEC	AE	The key_record and attribute_record in primary key operation do not belong to the same table
4288	DMEC	AE	Blob handle for column not available
4289	DMEC	AE	API version mismatch or wrong

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			)
4290	DMEC	AE	Missing column specification in NdbDictionary::RecordSpecification
4291	DMEC	AE	Duplicate column specification in NdbDictionary::RecordSpecification
4292	DMEC	AE	NdbRecord for tuple access is not an index key NdbRecord
4293	DMEC	AE	Error returned from application scanIndex() callback
4294	DMEC	AE	Scan filter is too large, discarded
4295	DMEC	AE	Column is NULL in Get/SetValueSpec structure
4296	DMEC	AE	Invalid AbortOption
4297	DMEC	AE	Invalid or unsupported OperationOptions structure
4298	DMEC	AE	Invalid or unsupported ScanOptions structure
4299	DMEC	AE	Incorrect combination of ScanOption flags, extraGetValues ptr and numExtraGetValues
NO_CONTACT_WITH_PROCESS	DMEC	AE	No contact with the process (dead ?).
WRONG_PROCESS_TYPE	DMEC	AE	The process has wrong type. Expected a DB process.
SEND_OR_RECEIVE_FAILED	DMEC	AE	Send to process or receive failed.
INVALID_ERROR_NUMBER	DMEC	AE	Invalid error number. Should be >= 0.
INVALID_TRACE_NUMBER	DMEC	AE	Invalid trace number.
INVALID_BLOCK_NAME	DMEC	AE	Invalid block name
NODE_SHUTDOWN_IN_PROGESS	DMEC	AE	Node shutdown in progress
SYSTEM_SHUTDOWN_IN_PROGRESS	DMEC	AE	System shutdown in progress
NODE_SHUTDOWN_WOULD_CAUSE_SYSTEM_CRASH	DMEC	AE	Node shutdown would cause system crash
UNSUPPORTED_NODE_SHUTDOWN	DMEC	AE	Unsupported multi node shutdown. Abort option required.
NODE_NOT	DMEC	AE	The specified node is not an API node.

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
_API_NODE			
OPERATION_NOT_ALLOWED_START_STOP	DMEC	AE	Operation not allowed while nodes are starting or stopping.
NO_CONTACT_WITH_DB_NODES	DMEC	AE	No contact with database nodes

4.2.2.3. *No data found* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
626	HA_ERR_KEY_NOT_FOUND	ND	Tuple did not exist

4.2.2.4. *Constraint violation* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
630	HA_ERR_FOUND_DUPP_KEY	CV	Tuple already existed when attempting to insert
839	DMEC	CV	Illegal null attribute
840	DMEC	CV	Trying to set a NOT NULL attribute to NULL
893	HA_ERR_FOUND_DUPP_KEY	CV	Constraint violation e.g. duplicate value in unique index

4.2.2.5. *Schema error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4713	DMEC	SE	Column defined in event does not exist in table
703	DMEC	SE	Invalid table format
704	DMEC	SE	Attribute name too long
705	DMEC	SE	Table name too long
707	DMEC	SE	No more table metadata records (increase MaxNoOfTables)
708	DMEC	SE	No more attribute metadata records (increase MaxNoOfAttributes)
709	HA_ERR_NO_SUCH_TABLE	SE	No such table existed
710	DMEC	SE	Internal: Get by table name not supported, use table id.
723	HA_ERR_NO_SUCH_TABLE	SE	No such table existed
736	DMEC	SE	Unsupported array size
737	HA_WRONG_CREATE_OPTION	SE	Attribute array size too big
738	HA_WRONG_CREATE_OPTION	SE	Record too big

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
739	HA_WRONG_CREATE_OPTION	SE	Unsupported primary key length
740	HA_WRONG_CREATE_OPTION	SE	Nullable primary key not supported
741	DMEC	SE	Unsupported alter table
743	HA_WRONG_CREATE_OPTION	SE	Unsupported character set in table or index
744	DMEC	SE	Character string is invalid for given character set
745	HA_WRONG_CREATE_OPTION	SE	Distribution key not supported for char attribute (use binary attribute)
779	HA_WRONG_CREATE_OPTION	SE	Invalid undo buffer size
764	HA_WRONG_CREATE_OPTION	SE	Invalid extent size
765	DMEC	SE	Out of filegroup records
750	IE	SE	Invalid file type
751	DMEC	SE	Out of file records
752	DMEC	SE	Invalid file format
753	IE	SE	Invalid filegroup for file
754	IE	SE	Invalid filegroup version when creating file
755	HA_WRONG_CREATE_OPTION	SE	Invalid tablespace
756	DMEC	SE	Index on disk column is not supported
757	DMEC	SE	Varsize bitfield not supported
758	DMEC	SE	Tablespace has changed
759	DMEC	SE	Invalid tablespace version
761	DMEC	SE	Unable to drop table as backup is in progress
762	DMEC	SE	Unable to alter table as backup is in progress
766	DMEC	SE	Cant drop file, no such file
767	DMEC	SE	Cant drop filegroup, no such filegroup
768	DMEC	SE	Cant drop filegroup, filegroup is used
769	DMEC	SE	Drop undofile not supported, drop logfile group instead
770	DMEC	SE	Cant drop file, file is used
774	DMEC	SE	Invalid schema object for drop
241	HA_ERR_TABLE_DEF_CHANGED	SE	Invalid schema object version
283	HA_ERR_NO_SUCH_TABLE	SE	Table is being dropped
284	HA_ERR_TABLE_DEF_CHANGED	SE	Table not defined in transaction coordinator
285	DMEC	SE	Unknown table error in transaction coordinator
881	DMEC	SE	Unable to create table, out of data pages (increase DataMemory)
906	DMEC	SE	Unsupported attribute type in index
907	DMEC	SE	Unsupported character set in table or index
1224	HA_WRONG_CREATE_OPTION	SE	Too many fragments
1225	DMEC	SE	Table not defined in local query handler
1226	DMEC	SE	Table is being dropped
1227	HA_WRONG_CREATE_OPTION	SE	Invalid schema version

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1228	DMEC	SE	Cannot use drop table for drop index
1229	DMEC	SE	Too long frm data supplied
1231	DMEC	SE	Invalid table or index to scan
1232	DMEC	SE	Invalid table or index to scan
1503	DMEC	SE	Out of filegroup records
1504	DMEC	SE	Out of logbuffer memory
1508	DMEC	SE	Out of file records
1509	DMEC	SE	File system error, check if path,permissions etc
1512	DMEC	SE	File read error
1514	DMEC	SE	Currently there is a limit of one logfile group
1515	DMEC	SE	Currently there is a 4G limit of one undo/data-file in 32-bit host
773	DMEC	SE	Out of string memory, please modify StringMemory config parameter
775	DMEC	SE	Create file is not supported when Diskless=1
1407	DMEC	SE	Subscription not found in subscriber manager
1415	DMEC	SE	Subscription not unique in subscriber manager
1417	DMEC	SE	Table in suscription not defined, probably dropped
1418	DMEC	SE	Subscription dropped, no new subscribers allowed
1419	DMEC	SE	Subscription already dropped
1421	DMEC	SE	Partially connected API in NdbOperation::execute()
1422	DMEC	SE	Out of subscription records
1423	DMEC	SE	Out of table records in SUMA
1424	DMEC	SE	Out of MaxNoOfConcurrentSubOperations
1425	DMEC	SE	Subscription being defined...while trying to stop subscriber
1426	DMEC	SE	No such subscriber

4.2.2.6. *Schema object already exists* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
746	DMEC	OE	Event name already exists
4244	HA_ERR_TABLE_EXIST	OE	Index or table with given name already exists

4.2.2.7. *User defined error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1321	DMEC	UD	Backup aborted by user request

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4260	DMEC	*UD*	NdbScanFilter: Operator is not defined in NdbScanFilter::Group
4261	DMEC	*UD*	NdbScanFilter: Column is NULL
4262	DMEC	*UD*	NdbScanFilter: Condition is out of bounds

4.2.2.8. *Insufficient space* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
623	HA_ERR_RECORD_FILE_FULL	*IS*	623
624	HA_ERR_RECORD_FILE_FULL	*IS*	624
625	HA_ERR_INDEX_FILE_FULL	*IS*	Out of memory in Ndb Kernel, hash index part (increase IndexMemory)
640	DMEC	*IS*	Too many hash indexes (should not happen)
826	HA_ERR_RECORD_FILE_FULL	*IS*	Too many tables and attributes (increase MaxNoOfAttributes or MaxNoOfTables)
827	HA_ERR_RECORD_FILE_FULL	*IS*	Out of memory in Ndb Kernel, table data (increase DataMemory)
902	HA_ERR_RECORD_FILE_FULL	*IS*	Out of memory in Ndb Kernel, ordered index data (increase DataMemory)
903	HA_ERR_INDEX_FILE_FULL	*IS*	Too many ordered indexes (increase MaxNoOfOrderedIndexes)
904	HA_ERR_INDEX_FILE_FULL	*IS*	Out of fragment records (increase MaxNoOfOrderedIndexes)
905	DMEC	*IS*	Out of attribute records (increase MaxNoOfAttributes)
1601	HA_ERR_RECORD_FILE_FULL	*IS*	Out extents, tablespace full
1602	DMEC	*IS*	No datafile in tablespace
747	DMEC	*IS*	Out of event records
908	DMEC	*IS*	Invalid ordered index tree node size
1303	DMEC	*IS*	Out of resources
1412	DMEC	*IS*	Can't accept more subscribers, out of space in pool
1416	DMEC	*IS*	Can't accept more subscriptions, out of space in pool

4.2.2.9. *Temporary Resource error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
217	DMEC	*TR*	217
218	DMEC	*TR*	218
219	DMEC	*TR*	219
233	DMEC	*TR*	Out of operation records in transaction coordinator (increase MaxNoOfConcurrentOperations)
275	DMEC	*TR*	Out of transaction records for complete phase (increase MaxNoOfConcurrentTransac-

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			TIONS)
279	DMEC	TR	OUT OF TRANSACTION MARKERS IN TRANSACTION COORDINATOR
414	DMEC	TR	414
418	DMEC	TR	OUT OF TRANSACTION BUFFERS IN LQH
419	DMEC	TR	419
245	DMEC	TR	TOO MANY ACTIVE SCANS
488	DMEC	TR	TOO MANY ACTIVE SCANS
490	DMEC	TR	TOO MANY ACTIVE SCANS
805	DMEC	TR	OUT OF ATTRINFO RECORDS IN TUPLE MANAGER
830	DMEC	TR	OUT OF ADD FRAGMENT OPERATION RECORDS
873	DMEC	TR	OUT OF ATTRINFO RECORDS FOR SCAN IN TUPLE MANAGER
899	DMEC	TR	ROWID ALREADY ALLOCATED
1217	DMEC	TR	OUT OF OPERATION RECORDS IN LOCAL DATA MANAGER (INCREASE MAXNOOFLOCALOPERATIONS)
1218	DMEC	TR	SEND BUFFERS OVERLOADED IN NDB KERNEL
1220	DMEC	TR	REDO LOG FILES OVERLOADED, CONSULT ONLINE MANUAL (INCREASE FRAGMENTLOGFILESIZE)
1222	DMEC	TR	OUT OF TRANSACTION MARKERS IN LQH
4021	DMEC	TR	OUT OF SEND BUFFER SPACE IN NDB API
4022	DMEC	TR	OUT OF SEND BUFFER SPACE IN NDB API
4032	DMEC	TR	OUT OF SEND BUFFER SPACE IN NDB API
1501	DMEC	TR	OUT OF UNDO SPACE
288	DMEC	TR	OUT OF INDEX OPERATIONS IN TRANSACTION COORDINATOR (INCREASE MAXNOOFCONCURRENTINDEXOPERATIONS)
289	DMEC	TR	OUT OF TRANSACTION BUFFER MEMORY IN TC (INCREASE TRANSACTIONBUFFERMEMORY)
780	DMEC	TR	TOO MANY SCHEMA TRANSACTIONS
783	DMEC	TR	TOO MANY SCHEMA OPERATIONS
785	DMEC	TR	SCHEMA OBJECT IS BUSY WITH ANOTHER SCHEMA TRANSACTION
291	DMEC	TR	OUT OF SCANFRAG RECORDS IN TC (INCREASE MAXNOOFLOCALSCANS)
784	DMEC	TR	INVALID SCHEMA TRANSACTION STATE
788	DMEC	TR	MISSING SCHEMA OPERATION AT TAKEOVER OF SCHEMA TRANSACTION
748	DMEC	TR	BUSY DURING READ OF EVENT TABLE
1350	DMEC	TR	BACKUP FAILED: FILE ALREADY EXISTS (USE 'START BACKUP <BACKUP ID>')
1411	DMEC	TR	SUBSCRIBER MANAGER BUSY WITH ADDING/REMOVING A SUBSCRIBER
1413	DMEC	TR	SUBSCRIBER MANAGER BUSY WITH ADDING THE SUBSCRIPTION
1414	DMEC	TR	SUBSCRIBER MANAGER HAS SUBSCRIBERS ON THIS SUBSCRIPTION
1420	DMEC	TR	SUBSCRIBER MANAGER BUSY WITH ADDING/REMOVING A TABLE

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
2810	DMEC	TR	No space left on the device
2811	DMEC	TR	Error with file permissions, please check file system
2815	DMEC	TR	Error in reading files, please check file system

4.2.2.10. *Node Recovery error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
286	DMEC	NR	Node failure caused abort of transaction
250	DMEC	NR	Node where lock was held crashed, restart scan transaction
499	DMEC	NR	Scan take over error, restart scan transaction
1204	DMEC	NR	Temporary failure, distribution changed
4002	DMEC	NR	Send to NDB failed
4010	DMEC	NR	Node failure caused abort of transaction
4025	DMEC	NR	Node failure caused abort of transaction
4027	DMEC	NR	Node failure caused abort of transaction
4028	DMEC	NR	Node failure caused abort of transaction
4029	DMEC	NR	Node failure caused abort of transaction
4031	DMEC	NR	Node failure caused abort of transaction
4033	DMEC	NR	Send to NDB failed
4115	DMEC	NR	Transaction was committed but all read information was not ""received due to node crash
4119	DMEC	NR	Simple/dirty read failed due to node failure
786	DMEC	NR	Schema transaction aborted due to node-failure
1405	DMEC	NR	Subscriber manager busy with node recovery
1427	DMEC	NR	Api node died, when SUB_START_REQ reached node

4.2.2.11. *Overload error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
701	DMEC	OL	System busy with other schema operation
711	DMEC	OL	System busy with node restart, schema operations not allowed
410	DMEC	OL	REDO log files overloaded, consult online manual (decrease TimeBetweenLocalCheckpoints, and|or increase NoOfFragmentLogFiles)
677	DMEC	OL	Index UNDO buffers overloaded (increase UndoIndexBuffer)
891	DMEC	OL	Data UNDO buffers overloaded (increase

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
			NDODATABUFFER)
1221	DMEC	OL	REDO BUFFERS OVERLOADED, CONSULT ONLINE MANUAL (INCREASE REDOBUFFER)
4006	DMEC	OL	CONNECT FAILURE - OUT OF CONNECTION OBJECTS (INCREASE MAXNOOFCONCURRENTTRANSACTIONS)

4.2.2.12. *Timeout expired* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
266	HA_ERR_LOCK_WAIT_TIMEOUT	TO	TIME-OUT IN NDB, PROBABLY CAUSED BY DEADLOCK
274, HA_ERR_LOCK_WAIT_TIMEOUT, TO, "Time-out in NDB, probably caused by dead-lock" }, { 237	HA_ERR_LOCK_WAIT_TIMEOUT	TO	TRANSACTION HAD TIMED OUT WHEN TRYING TO COMMIT IT

4.2.2.13. *Node shutdown* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
280	DMEC	NS	TRANSACTION ABORTED DUE TO NODE SHUTDOWN
270	DMEC	NS	TRANSACTION ABORTED DUE TO NODE SHUTDOWN
1223	DMEC	NS	READ OPERATION ABORTED DUE TO NODE SHUTDOWN
4023	DMEC	NS	TRANSACTION ABORTED DUE TO NODE SHUTDOWN
4030	DMEC	NS	TRANSACTION ABORTED DUE TO NODE SHUTDOWN
4034	DMEC	NS	TRANSACTION ABORTED DUE TO NODE SHUTDOWN

4.2.2.14. *Internal temporary* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
702	DMEC	IT	REQUEST TO NON-MASTER
787	DMEC	IT	SCHEMA TRANSACTION ABORTED
1703	DMEC	IT	NODE FAILURE HANDLING NOT COMPLETED

4.2.2.15. *Unknown result error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4007	DMEC	*UR*	Send to NDBD node failed
4008	DMEC	*UR*	Receive from NDB failed
4009	HA_ERR_NO_CONNECTION	*UR*	Cluster Failure
4012	DMEC	*UR*	Request NDBD time-out, maybe due to high load or communication problems
4013	DMEC	*UR*	Request timed out in waiting for node failure
4024	DMEC	*UR*	Time-out, most likely caused by simple read or cluster failure

4.2.2.16. *Internal error* Messages

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
896	DMEC	*IE*	Tuple corrupted - wrong checksum or column data in invalid format
901	DMEC	*IE*	Inconsistent ordered index. The index needs to be dropped and recreated
202	DMEC	*IE*	202
203	DMEC	*IE*	203
207	DMEC	*IE*	207
208	DMEC	*IE*	208
209	DMEC	*IE*	Communication problem, signal error
220	DMEC	*IE*	220
230	DMEC	*IE*	230
232	DMEC	*IE*	232
238	DMEC	*IE*	238
271	DMEC	*IE*	Simple Read transaction without any attributes to read
272	DMEC	*IE*	Update operation without any attributes to update
276	DMEC	*IE*	276
277	DMEC	*IE*	277
278	DMEC	*IE*	278
287	DMEC	*IE*	Index corrupted
290	DMEC	*IE*	Corrupt key in TC, unable to xfrm
293	DMEC	*IE*	Inconsistent trigger state in TC block
292	DMEC	*IE*	Inconsistent index state in TC block
631	DMEC	*IE*	631
632	DMEC	*IE*	632
706	DMEC	*IE*	Inconsistency during table creation
781	DMEC	*IE*	Invalid schema transaction key from NDB API
782	DMEC	*IE*	Invalid schema transaction id from NDB API
809	DMEC	*IE*	809
812	DMEC	*IE*	812
829	DMEC	*IE*	829

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
833	DMEC	IE	833
871	DMEC	IE	871
882	DMEC	IE	882
883	DMEC	IE	883
887	DMEC	IE	887
888	DMEC	IE	888
890	DMEC	IE	890
4000	DMEC	IE	MEMORY ALLOCATION ERROR
4001	DMEC	IE	Signal Definition Error
4005	DMEC	IE	Internal Error in NdbApi
4011	DMEC	IE	Internal Error in NdbApi
4107	DMEC	IE	Simple Transaction and Not Start
4108	DMEC	IE	Faulty operation type
4109	DMEC	IE	Faulty primary key attribute length
4110	DMEC	IE	Faulty length in ATTRINFO signal
4111	DMEC	IE	Status Error in NdbConnection
4113	DMEC	IE	Too many operations received
4320	DMEC	IE	Cannot use the same object twice to create table
4321	DMEC	IE	Trying to start two schema transactions
4344	DMEC	IE	Only DBDICT and TRIX can send requests to TRIX
4345	DMEC	IE	TRIX block is not available yet, probably due to node failure
4346	DMEC	IE	Internal error at index create/build
4347	DMEC	IE	Bad state at alter index
4348	DMEC	IE	Inconsistency detected at alter index
4349	DMEC	IE	Inconsistency detected at index usage
4350	DMEC	IE	Transaction already aborted
4731	DMEC	IE	Event not found
772	HA_WRONG_CREATE_OPTION	IE	Given fragmentType doesn't exist
749	HA_WRONG_CREATE_OPTION	IE	Primary Table in wrong state
1502	DMEC	IE	Filegroup already exists
1505	DMEC	IE	Invalid filegroup
1506	DMEC	IE	Invalid filegroup version
1507	DMEC	IE	File no already inuse
1510	DMEC	IE	File meta data error
1511	DMEC	IE	Out of memory
1513	DMEC	IE	Filegroup not online
1300	DMEC	IE	Undefined error
1301	DMEC	IE	Backup issued to not master (reissue command to master)
1302	DMEC	IE	Out of backup record
1304	DMEC	IE	Sequence failure
1305	DMEC	IE	Backup definition not implemented
1322	DMEC	IE	Backup already completed
1323	DMEC	IE	1323

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
1324	DMEC	IE	Backup log buffer full
1325	DMEC	IE	File or scan error
1326	DMEC	IE	Backup abortet due to node failure
1327	DMEC	IE	1327
1340	DMEC	IE	Backup undefined error
1700	DMEC	IE	Undefined error
4263	DMEC	IE	Invalid blob attributes or invalid blob parts table
4267	DMEC	IE	Corrupted blob value
4268	DMEC	IE	Error in blob head update forced rollback of transaction
4269	DMEC	IE	No connection to ndb management server
4270	DMEC	IE	Unknown blob error
4273	DMEC	IE	No blob table in dict cache
4274	DMEC	IE	Corrupted main table PK in blob operation

4.2.2.17. *Function not implemented* Errors

NDB Error Code	MySQL Error Code	NDB Error Classification	Error Message
4003	DMEC	NI	Function not implemented yet

4.2.3. NDB Error Classifications

The following table lists the classification codes used in Section 4.2, "NDB API Errors and Error Handling", and their descriptions. These can also be found in the file `/storage/ndb/src/ndbapi/ndberror.c`.

Classification Code	Error Status	Description
NE	*Success*	No error
AE	*Permanent error*	Application error
CE	*Permanent error*	Configuration or application error
ND	*Permanent error*	No data found
CV	*Permanent error*	Constraint violation
SE	*Permanent error*	Schema error
OE	*Permanent error*	Schema object already exists
UD	*Permanent error*	User defined error
IS	*Permanent error*	Insufficient space
TR	*Temporary error*	Temporary Resource error
NR	*Temporary error*	Node Recovery error
OL	*Temporary error*	Overload error
TO	*Temporary error*	Timeout expired
NS	*Temporary error*	Node shutdown
IT	*Temporary error*	Internal temporary
UR	*Unknown result*	Unknown result error
UE	*Unknown result*	Unknown error code
IE	*Permanent error*	Internal error
NI	*Permanent error*	Function not implemented

4.3. ndbd Error Messages

This section contains exit codes and error messages given when a data node process stops prematurely.

4.3.1. ndbd Error Codes

This section lists all the error messages that can be returned when a data node process halts due to an error, arranged in most cases according to the affected NDB kernel block.

For more information about kernel blocks, see Section 5.4, "NDB Kernel Blocks"

The meanings of the values given in the **Classification** column of each of the following tables is given in Section 4.3.2, "ndbd Error Classifications".

4.3.1.1. General Errors

This section contains ndbd error codes that are either generic in nature or otherwise not associated with a specific NDB kernel block.

Error Code	Error Classification	Error Text
NDBD_EXIT_GENERIC	*XRE*	Generic error
NDBD_EXIT_PRGERR	*XIE*	Assertion
NDBD_EXIT_NODE_NOT_IN_CONFIG	*XCE*	Node id in the configuration has the wrong type, (i.e. not an NDB node)
NDBD_EXIT_SYSTEM_ERROR	*XIE*	System error, node killed during node restart by other node
NDBD_EXIT_INDEX_NOTINRANGE	*XIE*	Array index out of range
NDBD_EXIT_ARBIT_SHUTDOWN	*XAE*	Node lost connection to other nodes and can not form a unpartitioned cluster, please investigate if there are error(s) on other node(s)
NDBD_EXIT_PARTITIONED_SHUTDOWN	*XAE*	Partitioned cluster detected. Please check if cluster is already running
NDBD_EXIT_NODE_DECLARED_DEAD	*XAE*	Node declared dead. See error log for details
NDBD_EXIT_POINTER_NOTINRANGE	*XIE*	Pointer too large
NDBD_EXIT_SR_OTHERNODE-FAILED	*XRE*	Another node failed during system restart, please investigate error(s) on other node(s)
NDBD_EXIT_NODE_NOT_DEAD	*XRE*	Internal node state conflict, most probably resolved by restarting node again
NDBD_EXIT_SR_REDOLOG	*XFI*	Error while reading the REDO log
NDBD-	*XFI*	Error while reading the schema file

Error Code	Error Classification	Error Text
BD_EXIT_SR_SCHEMAFILE		
2311	*XIE*	Conflict when selecting restart type
ND-BD_EXIT_NO_MORE_UNDOLOG	*XCR*	No more free UNDO log, increase UndoIndexBuffer
ND-BD_EXIT_SR_UNDOLOG	*XFI*	Error while reading the datapages and UNDO log
ND-BD_EXIT_SINGLE_USER_MODE	*XRE*	Data node is not allowed to get added to the cluster while it is in single user mode
ND-BD_EXIT_MEMALLOC	*XCE*	Memory allocation failure, please decrease some configuration parameters
ND-BD_EXIT_BLOCK_JBUFCONGESTION	*XIE*	Job buffer congestion
ND-BD_EXIT_TIME_QUEUE_SHORT	*XIE*	Error in short time queue
ND-BD_EXIT_TIME_QUEUE_LONG	*XIE*	Error in long time queue
ND-BD_EXIT_TIME_QUEUE_DELAY	*XIE*	Error in time queue, too long delay
ND-BD_EXIT_TIME_QUEUE_INDEX	*XIE*	Time queue index out of range
ND-BD_EXIT_BLOCK_BNR_ZERO	*XIE*	Send signal error
ND-BD_EXIT_WRONG_PRIO_LEVEL	*XIE*	Wrong priority level when sending signal
ND-BD_EXIT_NDBREQUIRE	*XIE*	Internal program error (failed ndbrequire)
ND-BD_EXIT_NDBASSERT	*XIE*	Internal program error (failed ndbassert)
ND-BD_EXIT_ERROR_INSERT	*XNE*	Error insert executed
ND-BD_EXIT_INVALID_CONFIG	*XCE*	Invalid configuration received from Management Server
ND-BD_EXIT_RESOURCE_ALLOC_ERROR	*XCE*	Resource allocation error, please review the configuration
ND-BD_EXIT_OS_SIG-NAL_RECEIVED	*XIE*	Error OS signal received
ND-	*XRE*	Partial system restart causing conflicting file systems

Error Code	Error Classification	Error Text
BD_EXIT_SR_RESTARTCONFLICT		

4.3.1.2. VM Errors

This section contains ndbd error codes that are associated with problems in the VM (virtal machine) NDB kernel block.

Error Code	Error Classification	Error Text
NDBD_EXIT_OUT_OF_LONG_SIGNAL_MEMORY	*XCR*	Signal lost, out of long signal memory, please increase LongMessageBuffer
NDBD_EXIT_WATCHDOG_TERMINATE	*XIE*	WatchDog terminate, internal error or massive overload on the machine running this node
NDBD_EXIT_SIGNAL_LOST_SEND_BUFFER_FULL	*XCR*	Signal lost, out of send buffer memory, please increase SendBufferMemory or lower the load
NDBD_EXIT_SIGNAL_LOST	*XIE*	Signal lost (unknown reason)
NDBD_EXIT_ILLEGAL_SIGNAL	*XIE*	Illegal signal (version mismatch a possibility)
NDBD_EXIT_CONNECTION_SETUP_FAILED	*XCE*	Connection setup failed

4.3.1.3. NDBCNTR Errors

This section contains ndbd error codes that are associated with problems in the NDBCNTR (initialization and configuration) NDB kernel block.

Error Code	Error Classification	Error Text
NDBD_EXIT_RESTART_TIMEOUT	*XCE*	Total restart time too long, consider increasing StartFailureTimeout or investigate error(s) on other node(s)
NDBD_EXIT_RESTART_DURING_SHUTDOWN	*XRE*	Node started while node shutdown in progress. Please wait until shutdown complete before starting node

4.3.1.4. DIH Errors

This section contains ndbd error codes that are associated with problems in the DIH (distribution handler) NDB kernel block.

Error Code	Error Classification	Error Text
NDBD_EXIT_MAX_CRASHED_REPL	*XFL*	Too many crashed replicas (8 consecutive node restart failures)

Error Code	Error Classification	Error Text
ICAS		
NDBD_EXIT_MASTER_FAILURE_DURING_NR	*XRE*	Unhandled master failure during node restart
NDBD_EXIT_LOST_NODE_GROUP	*XAE*	All nodes in a node group are unavailable
NDBD_EXIT_NO_RESTORABLE_REPLICA	*XFI*	Unable to find a restorable replica

4.3.1.5. ACC Errors

This section contains ndbd error codes that are associated with problems in the ACC (access control and lock management) NDB kernel block.

Error Code	Error Classification	Error Text
NDBD_EXIT_SR_OUT_OF_INDEXMEMORY	*XCR*	Out of index memory during system restart, please increase IndexMemory

4.3.1.6. TUP Errors

This section contains ndbd error codes that are associated with problems in the TUP (tuple management) NDB kernel block.

Error Code	Error Classification	Error Text
NDBD_EXIT_SR_OUT_OF_DATAMEMORY	*XCR*	Out of data memory during system restart, please increase DataMemory

4.3.1.7. NDBFS Errors

This section contains ndbd error codes that are associated with problems in the NDBFS (filesystem) NDB kernel block.

Most of these errors will provide additional information, such as operating system error codes, when they are generated.

Error Code	Error Classification	Error Text
NDBD_EXIT_AFS_NOPATH	*XIE*	No file system path
2802	*XIE*	Channel is full
2803	*XIE*	No more threads
NDBD_EXIT_AFS_PARAMETER	*XIE*	Bad parameter
NDBD_EXIT_AFS_INVALIDPATH	*XCE*	Illegal file system path
NDBD_EXIT_AFS_MAXOPEN	*XCR*	Max number of open files exceeded, please increase MaxNoOfOpenFiles

Error Code	Error Classification	Error Text
NDBD_EXIT_AFS_ALREADY_OPEN	*XIE*	FILE HAS ALREADY BEEN OPENED
NDBD_EXIT_AFS_ENVIRONMENT	*XIE*	ENVIRONMENT ERROR USING FILE
NDBD_EXIT_AFS_TEMP_NO_ACCESS	*XIE*	TEMPORARY ON ACCESS TO FILE
NDBD_EXIT_AFS_DISK_FULL	*XFF*	THE FILE SYSTEM IS FULL
NDBD_EXIT_AFS_PERMISSION_DENIED	*XCE*	RECEIVED PERMISSION DENIED FOR FILE
NDBD_EXIT_AFS_INVALID_PARAM	*XCE*	INVALID PARAMETER FOR FILE
NDBD_EXIT_AFS_UNKNOWN	*XIE*	UNKNOWN FILE SYSTEM ERROR
NDBD_EXIT_AFS_NO_MORE_RESOURCES	*XIE*	SYSTEM REPORTS NO MORE FILE SYSTEM RESOURCES
NDBD_EXIT_AFS_NO_SUCH_FILE	*XFI*	FILE NOT FOUND
NDBD_EXIT_AFS_READ_UNDERFLOW	*XFI*	READ UNDERFLOW
NDBD_EXIT_INVALID_LCP_FILE	*XFI*	INVALID LCP
NDBD_EXIT_INSUFFICENT_NODES	*XRE*	INSUFFICENT NODES FOR SYSTEM RESTART
NDBD_EXIT_UNSUPPORTED_VERSION	*XRE*	UNSUPPORTED VERSION

4.3.1.8. Sentinel Errors

A special case, to handle unknown or previously unclassified errors. *You should always report a bug using http://bugs.mysql.com/ if you can repeat a problem giving rise to this error consistently.*

Error Code	Error Classification	Error Text
0	*XUE*	NO MESSAGE SLOGAN FOUND (PLEASE REPORT A BUG IF YOU GET THIS ERROR CODE)

4.3.2. ndbd Error Classifications

This section lists the classifications for the error messages described in Section 4.3.1, “ndbd Error Codes”.

Error Code	Error Classification	Error Text
XNE	*Success*	NO ERROR
XUE	*Unknown*	UNKNOWN
XIE	*XST_R*	INTERNAL ERROR, PROGRAMMING ERROR OR MISSING ERROR MESSAGE, PLEASE REPORT A BUG
XCE	*Permanent error, external action needed*	CONFIGURATION ERROR
XAE	*Temporary error, restart node*	ARBITRATION ERROR
XRE	*Temporary error, restart node*	RESTART ERROR
XCR	*Permanent error, external action needed*	RESOURCE CONFIGURATION ERROR
XFF	*Permanent error, external action needed*	FILE SYSTEM FULL
XFI	*Ndbd file system error, restart node initial*	NDBD FILE SYSTEM INCONSISTENCY ERROR, PLEASE REPORT A BUG
XFL	*Ndbd file system error, restart node initial*	NDBD FILE SYSTEM LIMIT EXCEEDED

4.4. NDB Transporter Errors

This section lists error codes, names, and messages that are written to the cluster log in the event of transporter errors.

Error Code	Error Name	Error Text
0x00	**TE_NO_ERROR**	NO ERROR
0x01	**TE_ERROR_CLOSING_SOCKET**	ERROR FOUND DURING CLOSING OF SOCKET
0x02	**TE_ERROR_IN_SELECT_BEFORE_ACCEPT**	ERROR FOUND BEFORE ACCEPT. THE TRANSPORTER WILL RETRY
0x03	**TE_INVALID_MESSAGE_LENGTH**	ERROR FOUND IN MESSAGE (INVALID MESSAGE LENGTH)
0x04	**TE_INVALID_CHECKSUM**	ERROR FOUND IN MESSAGE (CHECKSUM)
0x05	**TE_COULD_NOT_CREATE_SOCKET**	ERROR FOUND WHILE CREATING SOCKET(CAN'T CREATE SOCKET)
0x06	**TE_COULD_NOT_BIND_SOCKET**	ERROR FOUND WHILE BINDING SERVER SOCKET
0x07	**TE_LISTEN_FAILED**	ERROR FOUND WHILE LISTENING TO SERVER SOCKET
0x08	**TE_ACCEPT_RETURN_ERROR**	ERROR FOUND DURING ACCEPT(ACCEPT RETURN ERROR)
0x0b	**TE_SHM_DISCONNECT**	THE REMOTE NODE HAS DISCONNEC-

Error Code	Error Name	Error Text
		TED
0x0c	**TE_SHM_IPC_STAT**	UNABLE TO CHECK SHM SEGMENT
0x0d	**TE_SHM_UNABLE_TO_CREATE_SEGMENT**	UNABLE TO CREATE SHM SEGMENT
0x0e	**TE_SHM_UNABLE_TO_ATTACH_SEGMENT**	UNABLE TO ATTACH SHM SEGMENT
0x0f	**TE_SHM_UNABLE_TO_REMOVE_SEGMENT**	UNABLE TO REMOVE SHM SEGMENT
0x10	**TE_TOO_SMALL_SIGID**	SIG ID TOO SMALL
0x11	**TE_TOO_LARGE_SIGID**	SIG ID TOO LARGE
0x12	**TE_WAIT_STACK_FULL**	WAIT STACK WAS FULL
0x13	**TE_RECEIVE_BUFFER_FULL**	RECEIVE BUFFER WAS FULL
0x14	**TE_SIGNAL_LOST_SEND_BUFFER_FULL**	SEND BUFFER WAS FULL,AND TRYING TO FORCE SEND FAILS
0x15	**TE_SIGNAL_LOST**	SEND FAILED FOR UNKNOWN REASON(SIGNAL LOST)
0x16	**TE_SEND_BUFFER_FULL**	THE SEND BUFFER WAS FULL, BUT SLEEPING FOR A WHILE SOLVED
0x0017	**TE_SCI_LINK_ERROR**	THERE IS NO LINK FROM THIS NODE TO THE SWITCH
0x18	**TE_SCI_UNABLE_TO_START_SEQUENCE**	COULD NOT START A SEQUENCE, BECAUSE SYSTEM RESOURCES ARE EXUMED OR NO SEQUENCE HAS BEEN CREATED
0x19	**TE_SCI_UNABLE_TO_REMOVE_SEQUENCE**	COULD NOT REMOVE A SEQUENCE
0x1a	**TE_SCI_UNABLE_TO_CREATE_SEQUENCE**	COULD NOT CREATE A SEQUENCE, BECAUSE SYSTEM RESOURCES ARE EXEMPTED. MUST REBOOT
0x1b	**TE_SCI_UNRECOVERABLE_DATA_TFX_ERROR**	TRIED TO SEND DATA ON REDUNDANT LINK BUT FAILED
0x1c	**TE_SCI_CANNOT_INIT_LOCALSEGMENT**	CANNOT INITIALIZE LOCAL SEGMENT
0x1d	**TE_SCI_CANNOT_MAP_REMOTESEGMENT**	CANNOT MAP REMOTE SEGMENT
0x1e	**TE_SCI_UNABLE_TO_UNMAP_SEGMENT**	CANNOT FREE THE RESOURCES USED BY THIS SEGMENT (STEP 1)
0x1f	**TE_SCI_UNABLE_TO_REMOVE_SEGMENT**	CANNOT FREE THE RESOURCES USED BY THIS SEGMENT (STEP 2)
0x20	**TE_SCI_UNABLE_TO_DISCONNECT_SEGMENT**	CANNOT DISCONNECT FROM A REMOTE SEGMENT
0x21	**TE_SHM_IPC_PERMANENT**	SHM IPC PERMANENT ERROR
0x22	**TE_SCI_UNABLE_TO_CLOSE_CHANNEL**	UNABLE TO CLOSE THE SCI CHANNEL AND THE RESOURCES ALLOCATED

Chapter 5. MySQL Cluster Internals

This chapter contains information about MySQL Cluster that is not strictly necessary for running the Cluster product, but can prove useful for development and debugging purposes.

5.1. MySQL Cluster File Systems

This section contains information about the file systems created and used by MySQL Cluster data nodes and management nodes.

5.1.1. Cluster Data Node File System

This section discusses the files and directories created by MySQL Cluster nodes, their usual locations, and their purpose.

5.1.1.1. Cluster Data Node `DataDir` Files

A cluster data node's `DataDir` contains at a minimum 3 files. These are named as shown here, where *`node_id`* is the node ID:

- `ndb_`*`node_id`*`_out.log`

  Sample output:

  ```
  2006-09-12 20:13:24 [ndbd] INFO     -- Angel pid: 13677 ndb pid: 13678
  2006-09-12 20:13:24 [ndbd] INFO     -- NDB Cluster -- DB node 1
  2006-09-12 20:13:24 [ndbd] INFO     -- Version 5.1.12 (beta) --
  2006-09-12 20:13:24 [ndbd] INFO     -- Configuration fetched at localhost port 1186
  2006-09-12 20:13:24 [ndbd] INFO     -- Start initiated (version 5.1.12)
  2006-09-12 20:13:24 [ndbd] INFO     -- Ndbd_mem_manager::init(1) min: 20Mb initial: 20Mb
  WOPool::init(61, 9)
  RWPool::init(82, 13)
  RWPool::init(a2, 18)
  RWPool::init(c2, 13)
  RWPool::init(122, 17)
  RWPool::init(142, 15)
  WOPool::init(41, 8)
  RWPool::init(e2, 12)
  RWPool::init(102, 55)
  WOPool::init(21, 8)
  Dbdict: name=sys/def/SYSTAB_0,id=0,obj_ptr_i=0
  Dbdict: name=sys/def/NDB$EVENTS_0,id=1,obj_ptr_i=1
  m_active_buckets.set(0)
  ```

- `ndb_`*`node_id`*`_signal.log`

  This file contains a log of all signals sent to or from the data node.

  > **Note**
  >
  > This file is created only if the `SendSignalId` parameter is enabled, which is true only for `-debug` builds.

- `ndb_`*`node_id`*`.pid`

  This file contains the data node's process ID; it is created when the `ndbd` process is started.

The location of these files is determined by the value of the `DataDir` configuration parameter. See `DataDir`.

5.1.1.2. Cluster Data Node `FileSystemDir` Files

This directory is named `ndb_`*`nodeid`*`_fs`, where *`nodeid`* is the data node's node ID. It contains the following files and directories:

- **Files**:
  - `data-`*`nodeid`*`.dat`
  - `undo-`*`nodeid`*`.dat`
- **Directories**:
  - `LCP`: This directory holds 3 subdirectories, named `0`, `1`, and `2`, which contain local checkpoint datafiles (one per checkpoint — see Configuring MySQL Cluster Parameters for Local Checkpoints).

These subdirectories each contain a number of files whose names follow the pattern `TNFM.Data`, where `N` is a table ID and and M is a fragment number. For each table, there are `NoOfFragmentLogFiles` fragments, and thus that many files.

- Directories named `D1` and `D2`, each of which contains 2 subdirectories:
  - `DBDICT`: Contains data dictionary information. This is stored in:
    - The file `P0.SchemaLog`
    - A set of directories `T0`, `T1`, `T2`, ..., each of which contains an `S0.TableList` file.
  - Directories named `D8`, `D9`, `D10`, and `D11`, each of which contains a directory named `DBLQH`. In each case, the `DBLQH` directory contains 8 files named `S0.Fraglog`, `S1.FragLog`, ..., `S6.FragLog`, `S7.FragLog`.
  - `DBDIH`: This directory contains the file `PX.sysfile`, which records information such as the last GCI, restart status, and node group membership of each node; its structure is defined in `storage/ndb/src/kernel/blocks/dbdih/Sysfile.hpp` in the MySQL source tree. In addition, the `SX.FragList` files keep records of the fragments belonging to each table.

5.1.1.3. Cluster Data Node `BackupDataDir` Files

MySQL Cluster creates backup files in the directory specified by the `BackupDataDir` configuration parameter, as discussed in Using The MySQL Cluster Management Client to Create a Backup, and Identifying Data Nodes.

The files created when a backup is performed are listed and described in MySQL Cluster Backup Concepts.

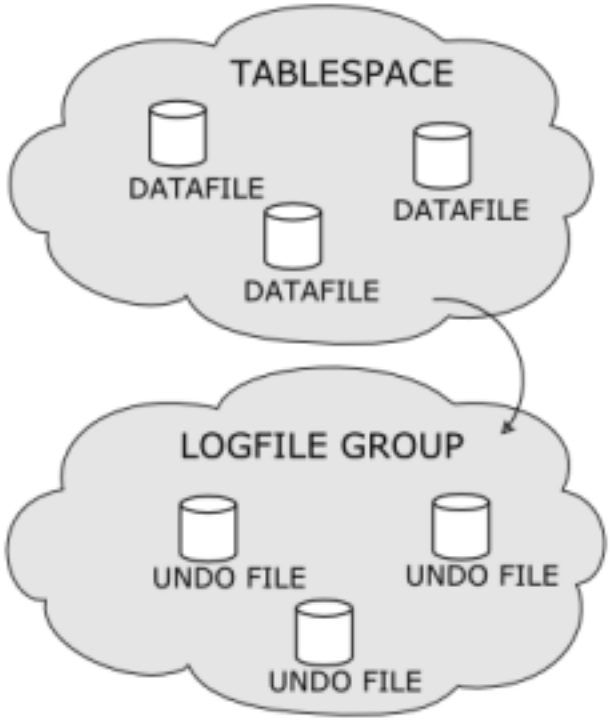
5.1.1.4. Cluster Disk Data Files

> **Note**
>
> This section applies only to MySQL 5.1 and later. Previous versions of MySQL did not support Disk Data tables.

MySQL Cluster Disk Data files are created (or dropped) by the user by means of SQL statements intended specifically for this purpose. Such files include the following:

- One or more *undo logfiles* associated with a *logfile group*
- One or more *datafiles* associated with a *tablespace* that uses the logfile group for undo logging

Both undo logfiles and datafiles are created in the data directory (`DataDir`) of each cluster data node. The relationship of these files with their logfile group and tablespace are shown in the following diagram:



Disk Data files and the SQL commands used to create and drop them are discussed in depth in MySQL Cluster Disk Data Tables.

5.1.2. Cluster Management Node File System

The files used by a MySQL Cluster management node are discussed in `ndb_mgmd`.

5.2. `DUMP` Commands

> **Warning**
>
> *Never use these commands on a production MySQL Cluster except under the express direction of MySQL Technical Support*. MySQL AB will *not* be held responsible for adverse results arising from their use under any other circumstances!

`DUMP` commands can be used in the Cluster management client (`ndb_mgm`) to dump debugging information to the Cluster log. They are documented here rather than in the MySQL Manual because:

- They are intended only for use in troubleshooting, debugging, and similar activities by MySQL developers, QA, and support personnel.
- Due to the way in which `DUMP` commands interact with memory, they can cause a running MySQL Cluster to malfunction or even to fail completely when used.
- The formats, arguments, and even availability of these commands are not guaranteed to be stable. *All of this information is subject to change at any time without prior notice*.
- For the preceding reasons, `DUMP` commands are neither intended nor warranted for use in a production environment by end-users.

General syntax:

```
ndb_mgm> node_id DUMP code [arguments]
```

This causes the contents of one or more `NDB` registers on the node with ID *`node_id`* to be dumped to the Cluster log. The registers affected are determined by the value of *`code`*. Some (but not all) `DUMP` commands accept additional *`arguments`*; these are noted and described where applicable.

Individual DUMP commands are listed by their *code* values in the sections that follow. For convenience in locating a given DUMP code, they are divided by thousands.

Each listing includes this information:

- The *code* value
- The relevant NDB kernel block or blocks (see Section 5.4, "NDB Kernel Blocks", for information about these)
- The DUMP code symbol where defined; if undefined, this is indicated using a triple dash: ---.
- Sample output; unless otherwise stated, it is assumed that each DUMP command is invoked as shown here:

  ```
  ndb_mgm> 2 DUMP code
  ```

  Generally, this is from the cluster log; in some cases, where the output may be generated in the node log instead, this is indicated. Where the DUMP command produces errors, the output is generally taken from the error log.
- Where applicable, additional information such as possible extra *arguments*, warnings, state or other values returned in the DUMP command's output, and so on. Otherwise its absence is indicated with "[N/A]".

> **Note**
>
> DUMP command codes are not necessarily defined sequentially. For example, codes 2 through 12 are currently undefined, and so are not listed. However, individual DUMP code values are subject to change, and there is no guarantee that a given code value will continue to be defined for the same purpose (or defined at all, or undefined) over time.
>
> There is also no guarantee that a given DUMP code — even if currently undefined — will not have serious consequences when used on a running MySQL Cluster.

For information concerning other ndb_mgm client commands, see Commands in the MySQL Cluster Management Client.

5.2.1. DUMP Codes 1 to 999

5.2.1.1. DUMP 1

Code	Symbol	Kernel Block(s)
1	---	QMGR

Description. Dumps information about cluster start Phase 1 variables (see Section 5.5.4, "STTOR Phase 1").

Sample Output.

```
Node 2: creadyDistCom = 1, cpresident = 2
Node 2: cpresidentAlive = 1, cpresidentCand = 2 (gci: 157807)
Node 2: ctoStatus = 0
Node 2: Node 2: ZRUNNING(3)
Node 2: Node 3: ZRUNNING(3)
```

Additional Information. [N/A]

5.2.1.2. DUMP 13

Code	Symbol	Kernel Block(s)
13	---	CMVMI, NDBCNTR

Description. Dump signal counter.

Sample Output.

```
Node 2: Cntr: cstartPhase = 9, cinternalStartphase = 8, block = 0
Node 2: Cntr: cmasterNodeId = 2
```

Additional Information. [N/A]

5.2.1.3. DUMP 14

Code	Symbol	Kernel Block(s)
14	CommitAckMarkersSize	DBLQH, DBTC

Description. Dumps free size in commitAckMarkerPool.

Sample Output.

```
Node 2: TC: m_commitAckMarkerPool: 12288 free size: 12288
Node 2: LQH: m_commitAckMarkerPool: 36094 free size: 36094
```

Additional Information. [N/A]

5.2.1.4. DUMP 15

Code	Symbol	Kernel Block(s)
15	CommitAckMarkersDump	DBLQH, DBTC

Description. Dumps information in commitAckMarkerPool.

Sample Output.

```
Node 2: TC: m_commitAckMarkerPool: 12288 free size: 12288
Node 2: LQH: m_commitAckMarkerPool: 36094 free size: 36094
```

Additional Information. [N/A]

5.2.1.5. DUMP 16

Code	Symbol	Kernel Block(s)
16	DihDumpNodeRestartInfo	DBDIH

Description. Provides node restart information.

Sample Output.

```
Node 2: c_nodeStartMaster.blockLcp = 0, c_nodeStartMaster.blockGcp = 0,
c_nodeStartMaster.wait = 0
Node 2: cstartGcpNow = 0, cgcpStatus = 0
Node 2: cfirstVerifyQueue = -256, cverifyQueueCounter = 0
Node 2: cgcpOrderBlocked = 0, cgcpStartCounter = 5
```

Additional Information. [N/A]

5.2.1.6. DUMP 17

Code	Symbol	Kernel Block(s)
17	DihDumpNodeStatusInfo	DBDIH

Description. Dumps node status.

Sample Output.

```
Node 2: Printing nodeStatus of all nodes
Node 2: Node = 2 has status = 1
Node 2: Node = 3 has status = 1
```

Additional Information. Possible node status values:

Value	Name
0	NOT_IN_CLUSTER

1	ALIVE
2	STARTING
3	DIED_NOW
4	DYING
5	DEAD

5.2.1.7. DUMP 18

Code	Symbol	Kernel Block(s)
18	DihPrintFragmentation	DBDIH

Description. Prints one entry per table fragment; lists the table number, fragment number, and list of nodes handling this fragment in order of priority.

Sample Output.

```
Node 2: Printing fragmentation of all tables --
Node 2:  Table 0 Fragment 0 - 2 3
Node 2:  Table 0 Fragment 1 - 3 2
Node 2:  Table 1 Fragment 0 - 2 3
Node 2:  Table 1 Fragment 1 - 3 2
Node 2:  Table 2 Fragment 0 - 2 3
Node 2:  Table 2 Fragment 1 - 3 2
Node 2:  Table 3 Fragment 0 - 2 3
Node 2:  Table 3 Fragment 1 - 3 2
Node 2:  Table 4 Fragment 0 - 2 3
Node 2:  Table 4 Fragment 1 - 3 2
Node 2:  Table 9 Fragment 0 - 2 3
Node 2:  Table 9 Fragment 1 - 3 2
Node 2:  Table 10 Fragment 0 - 2 3
Node 2:  Table 10 Fragment 1 - 3 2
Node 2:  Table 11 Fragment 0 - 2 3
Node 2:  Table 11 Fragment 1 - 3 2
Node 2:  Table 12 Fragment 0 - 2 3
Node 2:  Table 12 Fragment 1 - 3 2
```

Additional Information. [N/A]

5.2.1.8. DUMP 20

Code	Symbol	Kernel Block(s)
20	---	BACKUP

Description. Prints values of BackupDataBufferSize, BackupLogBufferSize, BackupWriteSize, and BackupMaxWriteSize

Sample Output.

```
Node 2: Backup: data: 2097152 log: 2097152 min: 32768 max: 262144
```

Additional Information. Can also be used to set these parameters, for example:

```
ndb_mgm> 2 DUMP 20 3 3 64 512
Sending dump signal with data:
0x00000014 0x00000003 0x00000003 0x00000040 0x00000200

Node 2: Backup: data: 3145728 log: 3145728 min: 65536 max: 524288
```

> **Warning**
>
> You must set each of these parameters to the same value on all nodes; otherwise, subsequent issuing of a START BACKUP command crashes the cluster.

5.2.1.9. DUMP 21

Code	Symbol	Kernel Block(s)
21	---	BACKUP

Description. Sends a GSN_BACKUP_REQ signal to the node, causing that node to initiate a backup.

Sample Output.

```
Node 2: Backup 1 started from node 2
Node 2: Backup 1 started from node 2 completed
StartGCP: 158515 StopGCP: 158518
#Records: 2061 #LogRecords: 0
Data: 35664 bytes Log: 0 bytes
```

Additional Information. [N/A]

5.2.1.10. DUMP 22

Code	Symbol	Kernel Block(s)
22 *backup_id*	---	BACKUP

Description. Sends a GSN_FSREMOVEREQ signal to the node. This should remove the backup having backup ID *backup_id* from the backup directory; *however, it actually causes the node to crash*.

Sample Output.

```
Time: Friday 16 February 2007 - 10:23:00
Status: Temporary error, restart node
Message: Assertion (Internal error, programming error or missing error message,
please report a bug)
Error: 2301
Error data: ArrayPool<T>::getPtr
Error object: ../../../../../storage/ndb/src/kernel/vm/ArrayPool.hpp line: 395
(block: BACKUP)
Program: ./libexec/ndbd
Pid: 27357
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.4
Version: Version 5.1.16 (beta)
```

Additional Information.

> **Warning**
>
> It appears that *any* invocation of DUMP 22 causes the node or nodes to crash.

5.2.1.11. DUMP 23

Code	Symbol	Kernel Block(s)
23	---	BACKUP

Description. Dumps all backup records and file entries belonging to those records.

> **Note**
>
> The example shows a single record with a single file only, but there may be multiple records and multiple file lines within each record.

Sample Output. With no backup in progress (BackupRecord shows as 0):

```
Node 2: BackupRecord 0: BackupId: 5 MasterRef: f70002 ClientRef: 0
Node 2:  State: 2
Node 2:  file 0: type: 3 flags: H'0
```

While a backup is in progress (BackupRecord is 1):

```
Node 2: BackupRecord 1: BackupId: 8 MasterRef: f40002 ClientRef: 80010001
Node 2:  State: 1
Node 2:  file 3: type: 3 flags: H'1
Node 2:  file 2: type: 2 flags: H'1
Node 2:  file 0: type: 1 flags: H'9
Node 2: BackupRecord 0: BackupId: 110 MasterRef: f70002 ClientRef: 0
Node 2:  State: 2
Node 2:  file 0: type: 3 flags: H'0
```

Additional Information. State values:

Value	State	Description
0	INITIAL	
1	DEFINING	Defining backup content and parameters
2	DEFINED	DEFINE_BACKUP_CONF signal sent by slave, received on master
3	STARTED	Creating triggers
4	SCANNING	Scanning fragments
5	STOPPING	Closing files
6	CLEANING	Freeing resources
7	ABORTING	Aborting backup

Types:

Value	Name
1	CTL_FILE
2	LOG_FILE
3	DATA_FILE
4	LCP_FILE

Flags:

Value	Name
0x01	BF_OPEN
0x02	BF_OPENING
0x04	BF_CLOSING
0x08	BF_FILE_THREAD
0x10	BF_SCAN_THREAD
0x20	BF_LCP_META

5.2.1.12. DUMP 24

Code	Symbol	Kernel Block(s)
24	---	BACKUP

Description. Prints backup record pool information.

Sample Output.

```
Node 2: Backup - dump pool sizes
Node 2: BackupPool: 2 BackupFilePool: 4 TablePool: 323
Node 2: AttrPool: 2 TriggerPool: 4 FragmentPool: 323
Node 2: PagePool: 198
```

Additional Information. If 2424 is passed as an argument (for example, 2 DUMP 24 2424), this causes an LCP.

5.2.1.13. DUMP 25

Code	Symbol	Kernel Block(s)
25	NdbcntrTestStopOnError	NDBCNTR

Description. Kills the data node or nodes.

Sample Output.

```
Time: Friday 16 February 2007 - 10:26:46
Status: Temporary error, restart node
Message: System error, node killed during node restart by other node
```

```
(Internal error, programming error or missing error message, please report a bug)
Error: 2303
Error data: System error 6,  this node was killed by node 2
Error object: NDBCNTR (Line: 234) 0x00000008
Program: ./libexec/ndbd
Pid: 27665
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.5
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.1.14. DUMP 70

Code	Symbol	Kernel Block(s)
70	NdbcntrStopNodes	

Description.

Sample Output.

Additional Information. [N/A]

5.2.1.15. DUMP 400

Code	Symbol	Kernel Block(s)
400	NdbfsDumpFileStat-	NDBFS

Description. Provides NDB file system statistics.

Sample Output.

```
Node 2: NDBFS: Files: 27 Open files: 10
Node 2:  Idle files: 17 Max opened files: 12
Node 2:  Max files: 40
Node 2:  Requests: 256
```

Additional Information. [N/A]

5.2.1.16. DUMP 401

Code	Symbol	Kernel Block(s)
401	NdbfsDumpAllFiles	NDBFS

Description. Prints NDB file system file handles and states (OPEN or CLOSED).

Sample Output.

```
Node 2: NDBFS: Dump all files: 27
Node 2:  0 (0x87867f8): CLOSED
Node 2:  1 (0x8787e70): CLOSED
Node 2:  2 (0x8789490): CLOSED
Node 2:  3 (0x878aab0): CLOSED
Node 2:  4 (0x878c0d0): CLOSED
Node 2:  5 (0x878d6f0): CLOSED
Node 2:  6 (0x878ed10): OPEN
Node 2:  7 (0x8790330): OPEN
Node 2:  8 (0x8791950): OPEN
Node 2:  9 (0x8792f70): OPEN
Node 2: 10 (0x8794590): OPEN
Node 2: 11 (0x8795da0): OPEN
Node 2: 12 (0x8797358): OPEN
Node 2: 13 (0x8798978): OPEN
Node 2: 14 (0x8799f98): OPEN
Node 2: 15 (0x879b5b8): OPEN
Node 2: 16 (0x879cbd8): CLOSED
Node 2: 17 (0x879e1f8): CLOSED
Node 2: 18 (0x879f818): CLOSED
Node 2: 19 (0x87a0e38): CLOSED
Node 2: 20 (0x87a2458): CLOSED
Node 2: 21 (0x87a3a78): CLOSED
```

```
Node 2: 22 (0x87a5098): CLOSED
Node 2: 23 (0x87a66b8): CLOSED
Node 2: 24 (0x87a7cd8): CLOSED
Node 2: 25 (0x87a92f8): CLOSED
Node 2: 26 (0x87aa918): CLOSED
```

Additional Information. [N/A]

5.2.1.17. DUMP 402

Code	Symbol	Kernel Block(s)
402	NdbfsDumpOpenFiles	NDBFS

Description. Prints list of NDB file system open files.

Sample Output.

```
Node 2: NDBFS: Dump open files: 10
Node 2:  0 (0x8792f70): /usr/local/mysql-5.1/cluster/ndb_2_fs/D1/DBDIH/P0.sysfile
Node 2:  1 (0x8794590): /usr/local/mysql-5.1/cluster/ndb_2_fs/D2/DBDIH/P0.sysfile
Node 2:  2 (0x878ed10): /usr/local/mysql-5.1/cluster/ndb_2_fs/D8/DBLQH/S0.FragLog
Node 2:  3 (0x8790330): /usr/local/mysql-5.1/cluster/ndb_2_fs/D9/DBLQH/S0.FragLog
Node 2:  4 (0x8791950): /usr/local/mysql-5.1/cluster/ndb_2_fs/D10/DBLQH/S0.FragLog
Node 2:  5 (0x8795da0): /usr/local/mysql-5.1/cluster/ndb_2_fs/D11/DBLQH/S0.FragLog
Node 2:  6 (0x8797358): /usr/local/mysql-5.1/cluster/ndb_2_fs/D8/DBLQH/S1.FragLog
Node 2:  7 (0x8798978): /usr/local/mysql-5.1/cluster/ndb_2_fs/D9/DBLQH/S1.FragLog
Node 2:  8 (0x8799f98): /usr/local/mysql-5.1/cluster/ndb_2_fs/D10/DBLQH/S1.FragLog
Node 2:  9 (0x879b5b8): /usr/local/mysql-5.1/cluster/ndb_2_fs/D11/DBLQH/S1.FragLog
```

Additional Information. [N/A]

5.2.1.18. DUMP 403

Code	Symbol	Kernel Block(s)
403	NdbfsDumpIdleFiles	NDBFS

Description. Prints list of NDB file system idle file handles.

Sample Output.

```
Node 2: NDBFS: Dump idle files: 17
Node 2:  0 (0x8787e70): CLOSED
Node 2:  1 (0x87aa918): CLOSED
Node 2:  2 (0x8789490): CLOSED
Node 2:  3 (0x878d6f0): CLOSED
Node 2:  4 (0x878aab0): CLOSED
Node 2:  5 (0x878c0d0): CLOSED
Node 2:  6 (0x879cbd8): CLOSED
Node 2:  7 (0x87a0e38): CLOSED
Node 2:  8 (0x87a2458): CLOSED
Node 2:  9 (0x879e1f8): CLOSED
Node 2: 10 (0x879f818): CLOSED
Node 2: 11 (0x87a66b8): CLOSED
Node 2: 12 (0x87a7cd8): CLOSED
Node 2: 13 (0x87a3a78): CLOSED
Node 2: 14 (0x87a5098): CLOSED
Node 2: 15 (0x87a92f8): CLOSED
Node 2: 16 (0x87867f8): CLOSED
```

Additional Information. [N/A]

5.2.1.19. DUMP 404

Code	Symbol	Kernel Block(s)
404	---	NDBFS

Description. Kills node or nodes.

Sample Output.

```
Time: Friday 16 February 2007 - 11:17:55
Status: Temporary error, restart node
```

```
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: ndbfs/Ndbfs.cpp
Error object: NDBFS (Line: 1066) 0x00000008
Program: ./libexec/ndbd
Pid: 29692
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.7
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.2. `DUMP` Codes 1000 to 1999

5.2.2.1. `DUMP 1000`

Code	Symbol	Kernel Block(s)
1000	`DumpPageMemory`	`DBACC`, `DBTUP`

Description. Prints data node mMemory usage (`ACC` & `TUP`), as both a number of data pages, and the percentage of `DataMemory` and `IndexMemory` used.

Sample Output.

```
Node 2: Data usage is 8%(54 32K pages of total 640)
Node 2: Index usage is 1%(24 8K pages of total 1312)
Node 2: Resource 0 min: 0 max: 639 curr: 0
```

> **Note**
>
> When invoked as `ALL DUMP 1000`, this command reports memory usage for each data node separately, in turn.

Additional Information. This is currently the only way to determine actual cluster memory usage (other than by waiting for the automatic threshold log messages to be generated).

5.2.2.2. `DUMP 1223`

Code	Symbol	Kernel Block(s)
1223	`---`	`DBDICT`

Description. Kills node.

Sample Output.

```
Time: Friday 16 February 2007 - 11:25:17
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: dbtc/DbtcMain.cpp
Error object: DBTC (Line: 464) 0x00000008
Program: ./libexec/ndbd
Pid: 742
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.10
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.2.3. `DUMP 1224`

Code	Symbol	Kernel Block(s)
1224	`---`	`DBDICT`

Description. Kills node.

Sample Output.

```
Time: Friday 16 February 2007 - 11:26:36
```

```
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: dbdih/DbdihMain.cpp
Error object: DBDIH (Line: 14433) 0x00000008
Program: ./libexec/ndbd
Pid: 975
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.11
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.2.4. DUMP 1225

Code	Symbol	Kernel Block(s)
1225	---	DBDICT

Description. Kills node.

Sample Output.

```
Node 2: Forced node shutdown completed.
Initiated by signal 6.
Caused by error 2301: 'Assertion(Internal error, programming error or
missing error message, please report a bug). Temporary error, restart node'.
- Unknown error code: Unknown result: Unknown error code
```

Additional Information. [N/A]

5.2.2.5. DUMP 1226

Code	Symbol	Kernel Block(s)
1226	---	DBDICT

Description. Prints pool objects.

Sample Output.

```
Node 2: c_obj_pool: 1332 1321
Node 2: c_opRecordPool: 256 256
Node 2: c_rope_pool: 4204 4078
```

Additional Information. [N/A]

5.2.2.6. DUMP 1332

Code	Symbol	Kernel Block(s)
1332	LqhDumpAllDefinedTabs	DBACC

Description. Prints the states of all tables known by the local query handler (LQH).

Sample Output.

```
Node 2: Table 0 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 1 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 2 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 3 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 4 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 9 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
```

```
Node 2: Table 10 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 11 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
Node 2: Table 12 Status: 0 Usage: 0
Node 2:   frag: 0 distKey: 0
Node 2:   frag: 1 distKey: 0
```

Additional Information. [N/A]

5.2.2.7. DUMP 1333

Code	Symbol	Kernel Block(s)
1333	LqhDumpNoLogPages	DBACC

Description. Reports redo log buffer usage.

Sample Output.

```
Node 2: LQH: Log pages : 256 Free: 244
```

Additional Information. The redo log buffer is measured in 32KB pages, so the sample output can be interpreted as follows:

- **Redo log buffer total.**
- **Redo log buffer free.** 7,808KB = ~7.6MB
- **Redo log buffer used.** 384KB = ~0.4MB

5.2.3. DUMP Codes 2000 to 2999

5.2.3.1. DUMP 2300

Code	Symbol	Kernel Block(s)
2300	LqhDumpOneScanRec	DBACC

Description. [Unknown]

Sample Output. [Not available]

Additional Information. [N/A]

5.2.3.2. DUMP 2301

Code	Symbol	Kernel Block(s)
2301	LqhDumpAllScanRec	DBACC

Description. Kills the node.

Sample Output.

```
Time: Friday 16 February 2007 - 12:35:36
Status: Temporary error, restart node
Message: Assertion (Internal error, programming error or missing error
message, please report a bug)
Error: 2301
Error data: ArrayPool<T>::getPtr
Error object: ../../../../../storage/ndb/src/kernel/vm/ArrayPool.hpp line: 345
(block: DBLQH)
Program: ./ndbd
Pid: 10463
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.22
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.3. DUMP 2302

Code	Symbol	Kernel Block(s)
2302	LqhDumpAllActiveScanRec	DBACC

Description. [Unknown]

Sample Output.

```
Time: Friday 16 February 2007 - 12:51:14
Status: Temporary error, restart node
Message: Assertion (Internal error, programming error or missing error message,
please report a bug)
Error: 2301
Error data: ArrayPool<T>::getPtr
Error object: ../../../../../storage/ndb/src/kernel/vm/ArrayPool.hpp line: 349
(block: DBLQH)
Program: ./ndbd
Pid: 10539
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.23
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.4. DUMP 2303

Code	Symbol	Kernel Block(s)
2303	LqhDumpLcpState	DBACC

Description. [Unknown]

Sample Output.

```
Node 2: == LQH LCP STATE ==
Node 2:  clcpCompletedState=0, c_lcpId=3, cnoOfFragsCheckpointed=0
Node 2:  lcpState=0 lastFragmentFlag=0
Node 2: currentFragment.fragPtrI=9
Node 2: currentFragment.lcpFragOrd.tableId=4
Node 2:  lcpQueued=0 reportEmpty=0
Node 2:  m_EMPTY_LCP_REQ=-1077761081
```

Additional Information. [N/A]

5.2.3.5. DUMP 2304

Code	Symbol	Kernel Block(s)
2304	---	DBLQH

Description. This command causes all fragment log files and their states to be written to the data node's out file (in the case of the data node having the node ID 1, this would be ndb_1_out.log). The number of these files is controlled by the NoFragmentLogFiles configuration parameter, whose default value is 16 in MySQL 5.1 and later releases.

Sample Output. The following is taken from ndb_1_out.log for a cluster with 2 data nodes:

```
LP 2 state: 0 WW_Gci: 1 gcprec: -256 flq: -256 currfile: 32 tailFileNo: 0 logTailMbyte: 1
  file 0(32)  FileChangeState: 0  logFileStatus: 20  currentMbyte: 1  currentFilepage 55
  file 1(33)  FileChangeState: 0  logFileStatus: 20  currentMbyte: 0  currentFilepage 0
  file 2(34)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 3(35)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 4(36)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 5(37)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 6(38)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 7(39)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 8(40)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 9(41)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 10(42)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 11(43)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 12(44)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 13(45)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 14(46)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 15(47)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
```

```
LP 3 state: 0 WW_Gci: 1 gcprec: -256 flq: -256 currfile: 48 tailFileNo: 0 logTailMbyte: 1
  file 0(48)  FileChangeState: 0  logFileStatus: 20  currentMbyte: 1  currentFilepage 55
  file 1(49)  FileChangeState: 0  logFileStatus: 20  currentMbyte: 0  currentFilepage 0
  file 2(50)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 3(51)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 4(52)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 5(53)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 6(54)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 7(55)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 8(56)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 9(57)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 10(58)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 11(59)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 12(60)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 13(61)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 14(62)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
  file 15(63)  FileChangeState: 0  logFileStatus: 1  currentMbyte: 0  currentFilepage 0
```

Additional Information. See also Section 5.2.3.6, “DUMP 2305”.

5.2.3.6. DUMP 2305

Code	Symbol	Kernel Block(s)
2305	---	DBLQH

Description. Show the states of all fragment log files (see Section 5.2.3.5, “DUMP 2304”), then kills the node.

Sample Output.

```
Time: Friday 16 February 2007 - 13:11:57
Status: Temporary error, restart node
Message: System error, node killed during node restart by other node
(Internal error, programming error or missing error message, please report a bug)
Error: 2303
Error data: Please report this as a bug. Provide as much info as possible,
expecially all the ndb_*_out.log files, Thanks. Shutting down node due to
failed handling of GCP_SAVEREQ
Error object: DBLQH (Line: 18619) 0x0000000a
Program: ./libexec/ndbd
Pid: 111
Time: Friday 16 February 2007 - 13:11:57
Status: Temporary error, restart node
Message: Error OS signal received (Internal error, programming error or
missing error message, please report a bug)
Error: 6000
Error data: Signal 6 received; Aborted
Error object: main.cpp
Program: ./libexec/ndbd
Pid: 11138
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.2
Version: Version 5.1.16 (beta)
```

Additional Information. No error message written to cluster log when the node is killed. Node failure is made evident only by subsequent heartbeat failure messages.

5.2.3.7. DUMP 2308

Code	Symbol	Kernel Block(s)
2308	---	DBLQH

Description. Kills the node.

Sample Output.

```
Time: Friday 16 February 2007 - 13:22:06
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: dblqh/DblqhMain.cpp
Error object: DBLQH (Line: 18805) 0x0000000a
Program: ./libexec/ndbd
Pid: 11640
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.8. DUMP 2315

Code	Symbol	Kernel Block(s)
2315	LqhErrorInsert5042	DBLQH

Description. [Unknown]

Sample Output. [N/A]

Additional Information. [N/A]

5.2.3.9. DUMP 2350

Code	Symbol	Kernel Block(s)
data_node_id 2350	---	---

+		

Description. Dumps all operations on a given data node or data nodes, according to the type and other parameters defined by the operation filter or filters specified.

Sample Output. Dump all operations on data node 2, from API node 5:

```
ndb_mgm> 2 DUMP 2350 1 5
2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: Starting dump of operations
2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: OP[470]:
Tab: 4 frag: 0 TC: 3 API: 5(0x8035)transid: 0x31c 0x3500500 op: SCAN state: InQueue
2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: End of operation dump
```

Additional information.

1. **Operation filter values.** The operation filter (or filters) can take on the following values:

Value	Filter
0	table ID
1	API node ID
2	2 transaction IDs, defining a range of transactions
3	transaction coordinator node ID

   In each case, the ID of the object specified follows the specifier. See the sample output for examples.

2. **Operation states.** The “normal” states that may appear in the output from this command are listed here:

   - **Transactions.**
     - **`Prepared`.** The transaction coordinator is idle, waiting for the API to proceed
     - **`Running`.** The transaction coordinator is currently preparing operations
     - **`Committing`, `Prepare to commit`, `Commit sent`.** The transaction coordinator is committing
     - **`Completing`.** The transaction coordinator is completing the commit (after commit, some cleanup is needed)
     - **`Aborting`.** The transaction coordinator is aborting the transaction
     - **`Scanning`.** The transaction coordinator is scanning
   - **Scan operations.**
     - **`WaitNextScan`.** The scan is idle, waiting for API
     - **`InQueue`.** The scan has not yet started, but rather is waiting in queue for other scans to complete
   - **Primary key operations.**
     - **`In lock queue`.** The operation is waiting on a lock
     - **`Running`.** The operation is being prepared
     - **`Prepared`.** The operation is prepared, holding an appropriate lock, and waiting for commit or rollback to complete

3. **Relation to NDB API.** It is possible to match the output of `DUMP 2350` to specific threads or `Ndb` objects. First suppose that you dump all operations on data node 2 from API node 5, using table 4 only, like this:

   ```
   ndb_mgm> 2 DUMP 2350 1 5 0 4
   2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: Starting dump of operations
   2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: OP[470]:
   Tab: 4 frag: 0 TC: 3 API: 5(0x8035)transid: 0x31c 0x3500500 op: SCAN state: InQueue
   2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: End of operation dump
   ```

   Suppose you are working with an `Ndb` instance named `MyNdb`, to which this operation belongs. You can see that this is the case by calling the `Ndb` object's `getReference()` method, like this:

   ```
     printf("MyNdb.getReference(): 0x%x\n", MyNdb.getReference());
   ```

   The output from the preceding line of code is:

```
MyNdb.getReference(): 0x80350005
```

The high 16 bits of the value shown corresponds to the number in parentheses from the `OP` line in the `DUMP` command's output (8035). For more about this method, see Section 2.3.8.1.16, "`Ndb::getReference()`".

This command was added in MySQL Cluster NDB 6.1.12 and MySQL Cluster NDB 6.2.2.

5.2.3.10. `DUMP 2352`

Code	Symbol	Kernel Block(s)
`node_id` 2352 *`operation_id`*	`---`	`---`

Description. Gets information about an operation with a given operation ID.

Sample Output. First, obtain a dump of operations. Here, we use `DUMP 2350` to get a dump of all operations on data node 2 from API node 5:

```
ndb_mgm> 2 DUMP 2350 1 5
2006-10-11 13:31:25 [MgmSrvr] INFO     -- Node 2: Starting dump of operations
2006-10-11 13:31:25 [MgmSrvr] INFO     -- Node 2: OP[3]:
Tab: 3 frag: 1 TC: 2 API: 5(0x8035)transid: 0x3 0x200400 op: INSERT state: Prepared
2006-10-11 13:31:25 [MgmSrvr] INFO     -- Node 2: End of operation dump
```

In this case, there is a single operation reported on node 2, whose operation ID is `3`. To obtain the transaction ID and primary key, we use the node ID and operation ID with `DUMP 2352` as shown here:

```
ndb_mgm> 2 dump 2352 3
2006-10-11 13:31:31 [MgmSrvr] INFO     -- Node 2: OP[3]: transid: 0x3 0x200400 key: 0x2
```

Additional Information. Use `DUMP 2350` to obtain an operation ID. See Section 5.2.3.9, "`DUMP 2350`", and the previous example.

This command was added in MySQL Cluster NDB 6.1.12 and MySQL Cluster NDB 6.2.2.

5.2.3.11. `DUMP 2400`

Code	Symbol	Kernel Block(s)
2400 *`record_id`*	`AccDumpOneScanRec`	`DBACC`

Description. Dumps the scan record having record ID *`record_id`*.

Sample Output. For `2 DUMP 1`:

```
Node 2: Dbacc::ScanRec[1]: state=1, transid(0x0, 0x0)
Node 2:  timer=0, continueBCount=0, activeLocalFrag=0, nextBucketIndex=0
Node 2:  scanNextfreerec=2 firstActOp=0 firstLockedOp=0, scanLastLockedOp=0 firstQOp=0 lastQOp=0
Node 2:  scanUserP=0, startNoBuck=0, minBucketIndexToRescan=0, maxBucketIndexToRescan=0
Node 2:  scanBucketState=0, scanLockHeld=0, userBlockRef=0, scanMask=0 scanLockMode=0
```

Additional Information. For dumping all scan records, see Section 5.2.3.12, "`DUMP 2401`".

5.2.3.12. `DUMP 2401`

Code	Symbol	Kernel Block(s)
2401	`AccDumpAllScanRec`	`DBACC`

Description. Dumps all scan records for the node specified.

Sample Output.

```
Node 2: ACC: Dump all ScanRec - size: 513
Node 2: Dbacc::ScanRec[1]: state=1, transid(0x0, 0x0)
Node 2:  timer=0, continueBCount=0, activeLocalFrag=0, nextBucketIndex=0
Node 2:  scanNextfreerec=2 firstActOp=0 firstLockedOp=0, scanLastLockedOp=0 firstQOp=0 lastQOp=0
Node 2:  scanUserP=0, startNoBuck=0, minBucketIndexToRescan=0, maxBucketIndexToRescan=0
Node 2:  scanBucketState=0, scanLockHeld=0, userBlockRef=0, scanMask=0 scanLockMode=0
```

```
Node 2: Dbacc::ScanRec[2]: state=1, transid(0x0, 0x0)
Node 2:  timer=0, continueBCount=0, activeLocalFrag=0, nextBucketIndex=0
Node 2:  scanNextfreerec=3 firstActOp=0 firstLockedOp=0, scanLastLockedOp=0 firstQOp=0 lastQOp=0
Node 2:  scanUserP=0, startNoBuck=0, minBucketIndexToRescan=0, maxBucketIndexToRescan=0
Node 2:  scanBucketState=0, scanLockHeld=0, userBlockRef=0, scanMask=0 scanLockMode=0
Node 2: Dbacc::ScanRec[3]: state=1, transid(0x0, 0x0)
Node 2:  timer=0, continueBCount=0, activeLocalFrag=0, nextBucketIndex=0
Node 2:  scanNextfreerec=4 firstActOp=0 firstLockedOp=0, scanLastLockedOp=0 firstQOp=0 lastQOp=0
Node 2:  scanUserP=0, startNoBuck=0, minBucketIndexToRescan=0, maxBucketIndexToRescan=0
Node 2:  scanBucketState=0, scanLockHeld=0, userBlockRef=0, scanMask=0 scanLockMode=0

⋮

Node 2: Dbacc::ScanRec[512]: state=1, transid(0x0, 0x0)
Node 2:  timer=0, continueBCount=0, activeLocalFrag=0, nextBucketIndex=0
Node 2:  scanNextfreerec=-256 firstActOp=0 firstLockedOp=0, scanLastLockedOp=0 firstQOp=0 lastQOp=0
Node 2:  scanUserP=0, startNoBuck=0, minBucketIndexToRescan=0, maxBucketIndexToRescan=0
Node 2:  scanBucketState=0, scanLockHeld=0, userBlockRef=0, scanMask=0 scanLockMode=0
```

Additional Information. If you want to dump a single scan record, given its record ID, see Section 5.2.3.11, “`DUMP 2400`”; for dumping all active scan records, see Section 5.2.3.13, “`DUMP 2402`”.

5.2.3.13. `DUMP 2402`

Code	Symbol	Kernel Block(s)
2402	`AccDumpAllActiveScanRec`	`DBACC`

Description. Dumps all active scan records.

Sample Output.

```
  Node 2: ACC: Dump active ScanRec - size: 513
```

Additional Information. To dump all scan records (active or not), see Section 5.2.3.12, “`DUMP 2401`”.

5.2.3.14. `DUMP 2403`

Code	Symbol	Kernel Block(s)
2403 *`record_id`*	`AccDumpOneOperationRec`	`DBACC`

Description. [Unknown]

Sample Output. (For `2 DUMP 1`:)

```
Node 2: Dbacc::operationrec[1]: transid(0x0, 0x7f1)
Node 2: elementIsforward=1, elementPage=0, elementPointer=724
Node 2: fid=0, fragptr=0, hashvaluePart=63926
Node 2: hashValue=-2005083304
Node 2: nextLockOwnerOp=-256, nextOp=-256, nextParallelQue=-256
Node 2: nextSerialQue=-256, prevOp=0
Node 2: prevLockOwnerOp=24, prevParallelQue=-256
Node 2: prevSerialQue=-256, scanRecPtr=-256
Node 2: m_op_bits=0xffffffff, scanBits=0
```

Additional Information. [N/A]

5.2.3.15. `DUMP 2404`

Code	Symbol	Kernel Block(s)
2404	`AccDumpNumOpRecs`	`DBACC`

Description. Number the number of operation records (total number, and number free).

Sample Output.

```
Node 2: Dbacc::OperationRecords: num=69012, free=32918
```

Additional Information. [N/A]

5.2.3.16. DUMP 2405

Code	Symbol	Kernel Block(s)
2405	AccDumpFreeOpRecs	

Description. Unknown: No output results if this command is called without additional arguments; if an extra argument is used, this command crashes the data node.

Sample Output. (For `2 DUMP 2405 1`:)

```
Time: Saturday 17 February 2007 - 18:33:54
Status: Temporary error, restart node
Message: Job buffer congestion (Internal error, programming error or
missing error message, please report a bug)
Error: 2334
Error data: Job Buffer Full
Error object: APZJobBuffer.C
Program: ./libexec/ndbd
Pid: 27670
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.17. DUMP 2406

Code	Symbol	Kernel Block(s)
2406	AccDumpNotFreeOpRecs	DBACC

Description. Unknown: No output results if this command is called without additional arguments; if an extra argument is used, this command crashes the data node.

Sample Output. (For `2 DUMP 2406 1`:)

```
Time: Saturday 17 February 2007 - 18:39:16
Status: Temporary error, restart node
Message: Job buffer congestion (Internal error, programming error or
missing error message, please report a bug)
Error: 2334
Error data: Job Buffer Full
Error object: APZJobBuffer.C
Program: ./libexec/ndbd
Pid: 27956
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.18. DUMP 2500

Code	Symbol	Kernel Block(s)
2500	TcDumpAllScanFragRec	DBTC

Description. Kills the data node.

Sample Output.

```
Time: Friday 16 February 2007 - 13:37:11
Status: Temporary error, restart node
Message: Assertion (Internal error, programming error or missing error
message, please report a bug)
Error: 2301
Error data: ArrayPool<T>::getPtr
Error object: ../../../../../storage/ndb/src/kernel/vm/ArrayPool.hpp line: 345
(block: CMVMI)
Program: ./libexec/ndbd
Pid: 13237
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.19. DUMP 2501

Code	Symbol	Kernel Block(s)
2501	TcDumpOneScanFragRec	DBTC

Description. No output if called without any additional arguments. With additional arguments, it kills the data node.

Sample Output. (For `2 DUMP 2501 1`:)

```
Time: Saturday 17 February 2007 - 18:41:41
Status: Temporary error, restart node
Message: Assertion (Internal error, programming error or missing error
message, please report a bug)
Error: 2301
Error data: ArrayPool<T>::getPtr
Error object: ../../../../../storage/ndb/src/kernel/vm/ArrayPool.hpp line: 345
(block: DBTC)
Program: ./libexec/ndbd
Pid: 28239
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.20. DUMP 2502

Code	Symbol	Kernel Block(s)
2502	TcDumpAllScanRec	DBTC

Description. Dumps all scan records.

Sample Output.

```
Node 2: TC: Dump all ScanRecord - size: 256
Node 2: Dbtc::ScanRecord[1]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=2
Node 2: Dbtc::ScanRecord[2]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=3
Node 2: Dbtc::ScanRecord[3]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=4

⋮

Node 2: Dbtc::ScanRecord[254]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=255
Node 2: Dbtc::ScanRecord[255]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=-256
Node 2: Dbtc::ScanRecord[255]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=-256
```

Additional Information. [N/A]

5.2.3.21. DUMP 2503

Code	Symbol	Kernel Block(s)
2503	TcDumpAllActiveScanRec	DBTC

Description. Dumps all active scan records.

Sample Output.

```
Node 2: TC: Dump active ScanRecord - size: 256
```

Additional Information. [N/A]

5.2.3.22. DUMP 2504

Code	Symbol	Kernel Block(s)
2504 *record_id*	TcDumpOneScanRec	DBTC

Description. Dumps a single scan record having the record ID *record_id*. (For dumping all scan records, see Section 5.2.3.20, "DUMP 2502".)

Sample Output. (For 2 DUMP 2504 1:)

```
Node 2: Dbtc::ScanRecord[1]: state=0nextfrag=0, nofrag=0
Node 2:  ailen=0, para=0, receivedop=0, noOprePperFrag=0
Node 2:  schv=0, tab=0, sproc=0
Node 2:  apiRec=-256, next=2
```

Additional Information. The attributes in the output of this command are described as follows:

- **ScanRecord.** The scan record slot number (same as *record_id*)
- **state.** One of the following values (found in as ScanState in Dbtc.hpp):

Value	State
0	IDLE
1	WAIT_SCAN_TAB_INFO
2	WAIT_AI
3	WAIT_FRAGMENT_COUNT
4	RUNNING
5	CLOSING_SCAN

- **nextfrag.** ID of the next fragment to be scanned. Used by a scan fragment process when it is ready for the next fragment.
- **nofrag.** Total number of fragments in the table being scanned.
- **ailen.** Length of the expected attribute information.
- **para.** Number of scan frag processes that belonging to this scan.
- **receivedop.** Number of operations received.
- **noOprePperFrag.** Maximum number of bytes per batch.
- **schv.** Schema version used by this scan.
- **tab.** The index or table that is scanned.
- **sproc.** Index of stored procedure belonging to this scan.
- **apiRec.** Reference to ApiConnectRecord
- **next.** Index of next ScanRecord in free list

5.2.3.23. DUMP 2505

Code	Symbol	Kernel Block(s)
2505	TcDumpOneApiConnectRec	DBTC

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.24. DUMP 2506

Code	Symbol	Kernel Block(s)
2506	TcDumpAllApiConnectRec	DBTC

Description. [Unknown]

Sample Output.

```
Node 2: TC: Dump all ApiConnectRecord - size: 12288
Node 2: Dbtc::ApiConnectRecord[1]: state=0, abortState=0, apiFailState=0
Node 2:  transid(0x0, 0x0), apiBref=0x1000002, scanRec=-256
Node 2:  ctcTimer=36057, apiTimer=0, counter=0, retcode=0, retsig=0
Node 2:  lqhkeyconfrec=0, lqhkeyreqrec=0, tckeyrec=0
Node 2:  next=-256
Node 2: Dbtc::ApiConnectRecord[2]: state=0, abortState=0, apiFailState=0
Node 2:  transid(0x0, 0x0), apiBref=0x1000002, scanRec=-256
Node 2:  ctcTimer=36057, apiTimer=0, counter=0, retcode=0, retsig=0
Node 2:  lqhkeyconfrec=0, lqhkeyreqrec=0, tckeyrec=0
Node 2:  next=-256
Node 2: Dbtc::ApiConnectRecord[3]: state=0, abortState=0, apiFailState=0
Node 2:  transid(0x0, 0x0), apiBref=0x1000002, scanRec=-256
Node 2:  ctcTimer=36057, apiTimer=0, counter=0, retcode=0, retsig=0
Node 2:  lqhkeyconfrec=0, lqhkeyreqrec=0, tckeyrec=0
Node 2:  next=-256

⋮

Node 2: Dbtc::ApiConnectRecord[12287]: state=7, abortState=0, apiFailState=0
Node 2:  transid(0x0, 0x0), apiBref=0xffffffff, scanRec=-256
Node 2:  ctcTimer=36308, apiTimer=0, counter=0, retcode=0, retsig=0
Node 2:  lqhkeyconfrec=0, lqhkeyreqrec=0, tckeyrec=0
Node 2:  next=-256
Node 2: Dbtc::ApiConnectRecord[12287]: state=7, abortState=0, apiFailState=0
Node 2:  transid(0x0, 0x0), apiBref=0xffffffff, scanRec=-256
Node 2:  ctcTimer=36308, apiTimer=0, counter=0, retcode=0, retsig=0
Node 2:  lqhkeyconfrec=0, lqhkeyreqrec=0, tckeyrec=0
Node 2:  next=-256
```

Additional Information. If the default settings are used, the output from this command is likely to exceed the maximum log file size.

5.2.3.25. DUMP 2507

Code	Symbol	Kernel Block(s)
2507	TcSetTransactionTimeout	DBTC

Description. Apparently requires an extra argument, but is not currently known with certainty.

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.26. DUMP 2508

Code	Symbol	Kernel Block(s)
2508	TcSetApplTransactionTimeout	DBTC

Description. Apparently requires an extra argument, but is not currently known with certainty.

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.27. DUMP 2509

Code	Symbol	Kernel Block(s)
2509	StartTcTimer	DBTC

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.28. DUMP 2510

Code	Symbol	Kernel Block(s)
2510	StopTcTimer	DBTC

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.29. DUMP 2511

Code	Symbol	Kernel Block(s)
2511	StartPeriodicTcTimer	DBTC

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.30. DUMP 2512

Code	Symbol	Kernel Block(s)
2512 [*delay*]	TcStartDumpIndexOpCount	DBTC

Description. Dumps the value of MaxNoOfConcurrentOperations, and the current resource usage, in a continuous loop. The *delay* time between reports can optionally be specified (in seconds), with the default being 1 and the maximum value being 25 (values greater than 25 are silently coerced to 25).

Sample Output. (Single report:)

```
Node 2: IndexOpCount: pool: 8192 free: 8192
```

Additional Information. There appears to be no way to disable the repeated checking of MaxNoOfConcurrentOperations once started by this command, except by restarting the data node. It may be preferable for this reason to use DUMP 2513 instead (see Section 5.2.3.31, "DUMP 2513").

5.2.3.31. DUMP 2513

Code	Symbol	Kernel Block(s)
2513	TcDumpIndexOpCount	

Description. Dumps the value of MaxNoOfConcurrentOperations, and the current resource usage.

Sample Output.

```
Node 2: IndexOpCount: pool: 8192 free: 8192
```

Additional Information. Unlike the continuous checking done by DUMP 2512 the check is performed only once (see Section 5.2.3.30, "DUMP 2512").

5.2.3.32. DUMP 2514

Code	Symbol	Kernel Block(s)
2514	---	DBTC

Description. [Unknown]

Sample Output.

```
Node 2: IndexOpCount: pool: 8192 free: 8192 - Repeated 3 times
Node 2: TC: m_commitAckMarkerPool: 12288 free size: 12288
Node 2: LQH: m_commitAckMarkerPool: 36094 free size: 36094
Node 3: TC: m_commitAckMarkerPool: 12288 free size: 12288
Node 3: LQH: m_commitAckMarkerPool: 36094 free size: 36094
Node 2: IndexOpCount: pool: 8192 free: 8192
```

Additional Information. [N/A]

5.2.3.33. DUMP 2515

Code	Symbol	Kernel Block(s)
2515	---	DBTC

Description. Appears to kill all data nodes in the cluster. Purpose unknown.

Sample Output. From the node for which the command is issued:

```
Time: Friday 16 February 2007 - 13:52:32
Status: Temporary error, restart node
Message: Assertion (Internal error, programming error or missing error
message, please report a bug)
Error: 2301
Error data: Illegal signal received (GSN 395 not added)
Error object: Illegal signal received (GSN 395 not added)
Program: ./libexec/ndbd
Pid: 14256
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

From the remaining data nodes:

```
Time: Friday 16 February 2007 - 13:52:31
Status: Temporary error, restart node
Message: System error, node killed during node restart by other node
(Internal error, programming error or missing error message, please report a bug)
Error: 2303
Error data: System error 0,  this node was killed by node 2515
Error object: NDBCNTR (Line: 234) 0x0000000a
Program: ./libexec/ndbd
Pid: 14261
Trace: /usr/local/mysql-5.1/cluster/ndb_3_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.3.34. DUMP 2550

Code	Symbol	Kernel Block(s)

data_node_id 2550 *transac-tion_filter*+	---	---

Description. Dumps all transaction from data node *data_node_id* meeting the conditions established by the transaction filter or filters specified.

Sample Output. Dump all transactions on node 2 which have been inactive for 30 seconds or longer:

```
ndb_mgm> 2 DUMP 2550 4 30
2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: Starting dump of transactions
2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: TRX[123]: API: 5(0x8035) transid: 0x31c 0x3500500 inactive: 42s stat
2006-10-09 13:16:49 [MgmSrvr] INFO     -- Node 2: End of transaction dump
```

Additional Information. The following values may be used for transaction filters. The filter value must be followed by one or more node IDs or, in the case of the last entry in the table, by the time in seconds that transactions have been inactive:

Value	Filter
1	API node ID
2	2 transaction IDs, defining a range of transactions
4	time transactions inactive (seconds)

This command was added in MySQL Cluster NDB 6.1.12 and MySQL Cluster NDB 6.2.2.

5.2.3.35. DUMP 2600

Code	Symbol	Kernel Block(s)
260	CmvmiDumpConnections	CMVMI

Description. Shows status of connections between all cluster nodes. When the cluster is operating normally, every connection has the same status.

Sample Output.

```
Node 3: Connection to 1 (MGM) is connected
Node 3: Connection to 2 (MGM) is trying to connect
Node 3: Connection to 3 (DB) does nothing
Node 3: Connection to 4 (DB) is connected
Node 3: Connection to 7 (API) is connected
Node 3: Connection to 8 (API) is connected
Node 3: Connection to 9 (API) is trying to connect
Node 3: Connection to 10 (API) is trying to connect
Node 3: Connection to 11 (API) is trying to connect
Node 4: Connection to 1 (MGM) is connected
Node 4: Connection to 2 (MGM) is trying to connect
Node 4: Connection to 3 (DB) is connected
Node 4: Connection to 4 (DB) does nothing
Node 4: Connection to 7 (API) is connected
Node 4: Connection to 8 (API) is connected
Node 4: Connection to 9 (API) is trying to connect
Node 4: Connection to 10 (API) is trying to connect
Node 4: Connection to 11 (API) is trying to connect
```

Additional Information. The message `is trying to connect` actually means that the node in question was not started. This can also be seen when there are unused `[api]` or `[mysql]` sections in the `config.ini` file nodes configured — in other words when there are spare slots for API or SQL nodes.

5.2.3.36. DUMP 2601

Code	Symbol	Kernel Block(s)
2601	CmvmiDumpLongSignalMemory	CMVMI

Description. [Unknown]

Sample Output.

```
Node 2: Cmvmi: g_sectionSegmentPool size: 4096 free: 4096
```

Additional Information. [N/A]

5.2.3.37. DUMP 2602

Code	Symbol	Kernel Block(s)
2602	CmvmiSetRestartOnErrorInsert	CMVMI

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.38. DUMP 2603

Code	Symbol	Kernel Block(s)
2603	CmvmiTestLongSigWithDelay	CMVMI

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.3.39. DUMP 2604

Code	Symbol	Kernel Block(s)
2604	CmvmiDumpSubscriptions	CMVMI

Description. Dumps current event subscriptions.

> **Note**
>
> This output appears in the ndb_*node_id*_out.log file (local to each data node) and not in the management server (global) cluster log file.

Sample Output.

```
2007-04-17 17:10:54 [ndbd] INFO     -- List subscriptions:
2007-04-17 17:10:54 [ndbd] INFO     -- Subscription: 0, nodeId: 1, ref: 0x80000001
2007-04-17 17:10:54 [ndbd] INFO     -- Category 0 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 1 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 2 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 3 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 4 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 5 Level 8
2007-04-17 17:10:54 [ndbd] INFO     -- Category 6 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 7 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 8 Level 15
2007-04-17 17:10:54 [ndbd] INFO     -- Category 9 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 10 Level 7
2007-04-17 17:10:54 [ndbd] INFO     -- Category 11 Level 15
```

Additional Information. The output lists all event subscriptions; for each subscription a header line and a list of categories with their current log levels is printed. The following information is included in the output:

- **Subscription.** The event subscription's internal ID
- **nodeID.** Node ID of the subscribing node

- **ref.** A block reference, consisting of a block ID from `storage/ndb/include/kernel/BlockNumbers.h` shifted to the left by 4 hexadecimal digits (16 bits) followed by a 4-digit hexadecimal node number. Block id `0x8000` appears to be a placeholder; it is defined as `MIN_API_BLOCK_NO`, with the node number part being 1 as expected
- **Category.** The cluster log category, as listed in Event Reports Generated in MySQL Cluster (see also the file `storage/ndb/include/mgmapi/mgmapi_config_parameters.h`).
- **Level.** The event level setting (the range being 0 to 15).

5.2.4. DUMP Codes 3000 to 3999

Currently unused.

5.2.5. DUMP Codes 4000 to 4999

Currently unused.

5.2.6. DUMP Codes 5000 to 5999

5.2.6.1. DUMP 5900

Code	Symbol	Kernel Block(s)
5900	LCPContinue	DBLQH

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.7. DUMP Codes 6000 to 6999

Currently unused.

5.2.8. DUMP Codes 7000 to 7999

5.2.8.1. DUMP 7000

Code	Symbol	Kernel Block(s)
7000	---	DBDIH

Description. Prints information on GCP state

Sample Output.

```
Node 2: ctimer = 299072, cgcpParticipantState = 0, cgcpStatus = 0
Node 2: coldGcpStatus = 0, coldGcpId = 436, cmasterState = 1
Node 2: cmasterTakeOverNode = 65535, ctcCounter = 299072
```

Additional Information. [N/A]

5.2.8.2. DUMP 7001

Code	Symbol	Kernel Block(s)
7001	---	DBDIH

Description. Prints information on the current LCP state.

Sample Output.

```
Node 2: c_lcpState.keepGci = 1
Node 2: c_lcpState.lcpStatus = 0, clcpStopGcp = 1
Node 2: cgcpStartCounter = 7, cimmediateLcpStart = 0
```

Additional Information. [N/A]

5.2.8.3. DUMP 7002

Code	Symbol	Kernel Block(s)
7002	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: cnoOfActiveTables = 4, cgcpDelay = 2000
Node 2: cdictblockref = 16384002, cfailurenr = 1
Node 2: con_lineNodes = 2, reference() = 16121858, creceivedfrag = 0
```

Additional Information. [N/A]

5.2.8.4. DUMP 7003

Code	Symbol	Kernel Block(s)
7003	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: cfirstAliveNode = 2, cgckptflag = 0
Node 2: clocallqhblockref = 16187394, clocaltcblockref = 16056322, cgcpOrderBlocked = 0
Node 2: cstarttype = 0, csystemnodes = 2, currentgcp = 438
```

Additional Information. [N/A]

5.2.8.5. DUMP 7004

Code	Symbol	Kernel Block(s)
7004	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: cmasterdihref = 16121858, cownNodeId = 2, cnewgcp = 438
Node 2: cndbStartReqBlockref = 16449538, cremainingfrags = 1268
Node 2: cntrlblockref = 16449538, cgcpSameCounter = 16, coldgcp = 437
```

Additional Information. [N/A]

5.2.8.6. DUMP 7005

Code	Symbol	Kernel Block(s)
7005	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: crestartGci = 1
```

Additional Information. [N/A]

5.2.8.7. DUMP 7006

Code	Symbol	Kernel Block(s)
7006	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: clcpDelay = 20, cgcpMasterTakeOverState = 0
Node 2: cmasterNodeId = 2
Node 2: cnoHotSpare = 0, c_nodeStartMaster.startNode = -256, c_nodeStartMaster.wait = 0
```

Additional Information. [N/A]

5.2.8.8. DUMP 7007

Code	Symbol	Kernel Block(s)
7007	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: c_nodeStartMaster.failNr = 1
Node 2: c_nodeStartMaster.startInfoErrorCode = -202116109
Node 2: c_nodeStartMaster.blockLcp = 0, c_nodeStartMaster.blockGcp = 0
```

Additional Information. [N/A]

5.2.8.9. DUMP 7008

Code	Symbol	Kernel Block(s)
7008	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: cfirstDeadNode = -256, cstartPhase = 7, cnoReplicas = 2
Node 2: cwaitLcpSr = 0
```

Additional Information. [N/A]

5.2.8.10. DUMP 7009

Code	Symbol	Kernel Block(s)
7009	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: ccalcOldestRestorableGci = 1, cnoOfNodeGroups = 1
Node 2: cstartGcpNow = 0
Node 2: crestartGci = 1
```

Additional Information. [N/A]

5.2.8.11. DUMP 7010

Code	Symbol	Kernel Block(s)
7010	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: cminHotSpareNodes = 0, c_lcpState.lcpStatusUpdatedPlace = 9843, cLcpStart = 1
Node 2: c_blockCommit = 0, c_blockCommitNo = 0
```

Additional Information. [N/A]

5.2.8.12. DUMP 7011

Code	Symbol	Kernel Block(s)
7011	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: c_COPY_GCIREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_COPY_TABREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_CREATE_FRAGREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_DIH_SWITCH_REPLICA_REQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_EMPTY_LCP_REQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_END_TOREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_GCP_COMMIT_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_GCP_PREPARE_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_GCP_SAVEREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_INCL_NODEREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_MASTER_GCPREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_MASTER_LCPREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_START_INFOREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_START_RECREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_START_TOREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_STOP_ME_REQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_TC_CLOPSIZEREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_TCGETOPSIZEREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
Node 2: c_UPDATE_TOREQ_Counter = [SignalCounter: m_count=0 0000000000000000]
```

Additional Information. [N/A]

5.2.8.13. DUMP 7012

Code	Symbol	Kernel Block(s)
7012	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: ParticipatingDIH = 0000000000000000
Node 2: ParticipatingLQH = 0000000000000000
Node 2: m_LCP_COMPLETE_REP_Counter_DIH = [SignalCounter: m_count=0 0000000000000000]
Node 2: m_LCP_COMPLETE_REP_Counter_LQH = [SignalCounter: m_count=0 0000000000000000]
Node 2: m_LAST_LCP_FRAG_ORD = [SignalCounter: m_count=0 0000000000000000]
Node 2: m_LCP_COMPLETE_REP_From_Master_Received = 0
```

Additional Information. [N/A]

5.2.8.14. DUMP 7013

Code	Symbol	Kernel Block(s)
7013	DihDumpLCPState	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: lcpStatus = 0 (update place = 9843)
Node 2: lcpStart = 1 lcpStopGcp = 1 keepGci = 1 oldestRestorable = 1
Node 2: immediateLcpStart = 0 masterLcpNodeId = 2
```

Additional Information. [N/A]

5.2.8.15. DUMP 7014

Code	Symbol	Kernel Block(s)
7014	DihDumpLCPMasterTakeOver	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: c_lcpMasterTakeOverState.state = 0 updatePlace = 11756 failedNodeId = -202116109
Node 2: c_lcpMasterTakeOverState.minTableId = 4092851187 minFragId = 4092851187
```

Additional Information. [N/A]

5.2.8.16. DUMP 7015

Code	Symbol	Kernel Block(s)
7015	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: Table 1: TabCopyStatus: 0 TabUpdateStatus: 0 TabLcpStatus: 3
Node 2:  Fragment 0: noLcpReplicas==0  0(on 2)=1(Idle) 1(on 3)=1(Idle)
Node 2:  Fragment 1: noLcpReplicas==0  0(on 3)=1(Idle) 1(on 2)=1(Idle)
Node 2: Table 2: TabCopyStatus: 0 TabUpdateStatus: 0 TabLcpStatus: 3
Node 2:  Fragment 0: noLcpReplicas==0  0(on 2)=0(Idle) 1(on 3)=0(Idle)
Node 2:  Fragment 1: noLcpReplicas==0  0(on 3)=0(Idle) 1(on 2)=0(Idle)
Node 2: Table 3: TabCopyStatus: 0 TabUpdateStatus: 0 TabLcpStatus: 3
Node 2:  Fragment 0: noLcpReplicas==0  0(on 2)=0(Idle) 1(on 3)=0(Idle)
Node 2:  Fragment 1: noLcpReplicas==0  0(on 3)=0(Idle) 1(on 2)=0(Idle)
Node 2: Table 4: TabCopyStatus: 0 TabUpdateStatus: 0 TabLcpStatus: 3
Node 2:  Fragment 0: noLcpReplicas==0  0(on 2)=0(Idle) 1(on 3)=0(Idle)
Node 2:  Fragment 1: noLcpReplicas==0  0(on 3)=0(Idle) 1(on 2)=0(Idle)
```

Additional Information. [N/A]

5.2.8.17. DUMP 7016

Code	Symbol	Kernel Block(s)
7016	DihAllAllowNodeStart	DBDIH

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.8.18. DUMP 7017

Code	Symbol	Kernel Block(s)
7017	DihMinTimeBetweenLCP	DBDIH

Description. [Unknown]

Sample Output.

```
. . .
```

Additional Information. [N/A]

5.2.8.19. DUMP 7018

Code	Symbol	Kernel Block(s)
7018	DihMaxTimeBetweenLCP	DBDIH

Description. [Unknown]

Sample Output.

```
. . .
```

Additional Information. [N/A]

5.2.8.20. DUMP 7020

Code	Symbol	Kernel Block(s)
7020	---	DBDIH

Description. This command provides general signal injection functionality. Two additional arguments are always required:

1. The number of the signal to be sent
2. The number of the block to which the signal should be sent

In addition some singals allow or require for extra data to be sent.

Sample Output.

```
. . .
```

Additional Information. [N/A]

5.2.8.21. DUMP 7080

Code	Symbol	Kernel Block(s)
7080	EnableUndoDelayDataWrite	DBACC, DBDIH, DBTUP

Description. [Unknown]

Sample Output.

```
. . .
```

Additional Information. [N/A]

5.2.8.22. DUMP 7090

Code	Symbol	Kernel Block(s)
7090	DihSetTimeBetweenGcp	DBDIH

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.8.23. DUMP 7098

Code	Symbol	Kernel Block(s)
7098	---	DBDIH

Description. [Unknown]

Sample Output.

```
Node 2: Invalid no of arguments to 7098 - startLcpRoundLoopLab - expected
2 (tableId, fragmentId)
```

Additional Information. [N/A]

5.2.8.24. DUMP 7099

Code	Symbol	Kernel Block(s)
7099	DihStartLcpImmediately	DBDIH

Description. Can be used to trigger an LCP manually.

Sample Output. In this example, node 2 is the master node and controls LCP/GCP synchronization for the cluster. regardless of the *node_id* specified, only the master node responds.

```
Node 2: Local checkpoint 7 started. Keep GCI = 1003 oldest restorable GCI = 947
Node 2: Local checkpoint 7 completed
```

Additional Information. You may need to enable a higher logging level to have the checkpoint's completion reported:

```
ndb_mgmgt; ALL CLUSTERLOG CHECKPOINT=8
```

5.2.8.25. DUMP 7901

Code	Symbol	Kernel Block(s)
7901	---	DBDIH, DBLQH

Description. Provides timings of GCPs.

Sample Output.

```
...
```

Additional Information. Available beginning with MySQL Cluster NDB 6.1.19.

5.2.9. DUMP Codes 8000 to 8999

5.2.9.1. DUMP 8004

Code	Symbol	Kernel Block(s)
8004	---	SUMA

Description. Dumps information about subscription resources.

Sample Output.

```
Node 2: Suma: c_subscriberPool  size: 260 free: 258
Node 2: Suma: c_tablePool  size: 130 free: 128
Node 2: Suma: c_subscriptionPool  size: 130 free: 128
Node 2: Suma: c_syncPool  size: 2 free: 2
Node 2: Suma: c_dataBufferPool  size: 1009 free: 1005
Node 2: Suma: c_metaSubscribers count: 0
Node 2: Suma: c_removeDataSubscribers count: 0
```

Additional Information. When `subscriberPool ... free` becomes and stays very low relative to `subscriberPool ... size`, it is often a good idea to increase the value of the `MaxNoOfTables` configuration parameter (`subscriberPool` = 2 * `MaxNoOfTables`). However, there could also be a problem with API nodes not releasing resources correctly when they are shut down. `DUMP 8004` provides a way to monitor these values.

5.2.9.2. DUMP 8005

Code	Symbol	Kernel Block(s)
8005	`---`	SUMA

Description. [Unknown]

Sample Output.

```
Node 2: Bucket 0 10-0 switch gci: 0 max_acked_gci: 2961 max_gci: 0 tail: -256 head: -256
Node 2: Bucket 1 00-0 switch gci: 0 max_acked_gci: 2961 max_gci: 0 tail: -256 head: -256
```

Additional Information. [N/A]

5.2.9.3. DUMP 8011

Code	Symbol	Kernel Block(s)
8011	`---`	SUMA

Description. Writes information about all subscribers to the cluster log.

Sample Output. (From cluster log:)

```
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1: -- Starting dump of subscribers --
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1: Table: 2 ver: 4294967040 #n: 1 (ref,data,subscription)
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1:  [ 80010004 24 0 ]
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1: Table: 3 ver: 4294967040 #n: 1 (ref,data,subscription)
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1:  [ 80010004 28 1 ]
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1: Table: 4 ver: 4294967040 #n: 1 (ref,data,subscription)
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1:  [ 80020004 24 2 ]
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 1: -- Ending dump of subscribers --
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2: -- Starting dump of subscribers --
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2: Table: 2 ver: 4294967040 #n: 1 (ref,data,subscription)
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2:  [ 80010004 24 0 ]
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2: Table: 3 ver: 4294967040 #n: 1 (ref,data,subscription)
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2:  [ 80010004 28 1 ]
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2: Table: 4 ver: 4294967040 #n: 1 (ref,data,subscription)
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2:  [ 80020004 24 2 ]
2007-11-23 13:17:31 [MgmSrvr] INFO     -- Node 2: -- Ending dump of subscribers --
```

Additional Information. Added in MySQL Cluster NDB 6.2.9.

5.2.10. DUMP Codes 9000 to 9999

5.2.10.1. DUMP 9002

Code	Symbol	Kernel Block(s)
9002	DumpTsman	TSMAN

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.10.2. DUMP 9800

Code	Symbol	Kernel Block(s)
9800	DumpTsman	TSMAN

Description. Kills data node.

Sample Output.

```
Time: Friday 16 February 2007 - 18:32:53
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: tsman.cpp
Error object: TSMAN (Line: 1413) 0x0000000a
Program: ./libexec/ndbd
Pid: 29658
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.10.3. DUMP 9801

Code	Symbol	Kernel Block(s)
9801	---	TSMAN

Description. Kills data node.

Sample Output.

```
Time: Friday 16 February 2007 - 18:35:48
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: tsman.cpp
Error object: TSMAN (Line: 1844) 0x0000000a
Program: ./libexec/ndbd
Pid: 30251
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.10.4. DUMP 9802

Code	Symbol	Kernel Block(s)
9802	---	TSMAN

Description. Kills data node.

Sample Output.

```
Time: Friday 16 February 2007 - 18:39:30
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: tsman.cpp
Error object: TSMAN (Line: 1413) 0x0000000a
Program: ./libexec/ndbd
Pid: 30482
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.10.5. DUMP 9803

Code	Symbol	Kernel Block(s)
9803	---	TSMAN

Description. Kills data node.

Sample Output.

```
Time: Friday 16 February 2007 - 18:41:32
Status: Temporary error, restart node
Message: Internal program error (failed ndbrequire) (Internal error,
programming error or missing error message, please report a bug)
Error: 2341
Error data: tsman.cpp
Error object: TSMAN (Line: 2144) 0x0000000a
Program: ./libexec/ndbd
Pid: 30712
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.2.11. DUMP Codes 10000 to 10999

5.2.11.1. DUMP 10000

Code	Symbol	Kernel Block(s)
10000	DumpLgman	---

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.12. DUMP Codes 11000 to 11999

5.2.12.1. DUMP 11000

Code	Symbol	Kernel Block(s)
11000	DumpPgman	---

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.13. DUMP Codes 12000 to 12999

5.2.13.1. DUMP 12001

Code	Symbol	Kernel Block(s)
12001	TuxLogToFile	DBTUX

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.13.2. DUMP 12002

Code	Symbol	Kernel Block(s)
12002	TuxSetLogFlags	DBTUX

Description. [Unknown]

Sample Output.

```
...
```

Additional Information. [N/A]

5.2.13.3. DUMP 12009

Code	Symbol	Kernel Block(s)
12009	TuxMetaDataJunk	DBTUX

Description. Kills data node.

Sample Output.

```
Time: Friday 16 February 2007 - 19:49:59
Status: Temporary error, restart node
Message: Error OS signal received (Internal error, programming error or
missing error message, please report a bug)
Error: 6000
Error data: Signal 6 received; Aborted
Error object: main.cpp
Program: ./libexec/ndbd
Pid: 13784
Trace: /usr/local/mysql-5.1/cluster/ndb_2_trace.log.1
Version: Version 5.1.16 (beta)
```

Additional Information. [N/A]

5.3. The NDB Protocol

This document discusses the protocol used for communication between data nodes and API nodes in a MySQL Cluster to perform various operations such as data reads and writes, committing and rolling back transactions, and handling of transaction records.

5.3.1. NDB Protocol Overview

MySQL Cluster data and API nodes communicate with one another by passing messages to one another. The sending of a message from one node and its reception by another node is referred to as a *signal*; the NDB Protocol is the set of rules governing the format of these messages and the manner in which they are passed.

An NDB message is typically either a *request* or a *response*. A request indicates that an API node wants to perform an operation involving cluster data (such as retrieval, insertion, updating, or deletion) or transactions (commit, roll back, or to fetch or relase a transaction record). A request is, when necessary, accompanied by key or index information. The response sent by a data node to this request indicates whether or not the request succeeded and, where appropriate, is accompanied by one or more data messages.

Request types. A request is represented as a REQ message. Requests can be divided into those handling data and those handling transactions:

- **Data requests.** Data request operations are of three principal types:

  1. **Primary key lookup operations.** These are performed through the exchange of TCKEY messages.

  2. **Unique key lookup operations.** These are performed through the exchange of TCINDX messages.

3. **Table or index scan operations.** These are performed through the exchange of SCANTAB messages.

Data request messages are often accompanied by KEYINFO messages, ATTRINFO messages, or both sorts of messages.

- **Transactional requests.** These may be divided into two categories:

  1. Commits and rollbacks, which are represented by TC_COMMIT and TCROLLBACK request messages, respectively.

  2. Transaction record requests — that is, transaction record acquisition and release. These requests are handled through the use of, respectively, TCSEIZE and TCRELEASE request messages.

Response types. A response indicates either the success or the failure of the request to which it is sent in reply:

- **Response indicating success.** This type of response is represented as a CONF (confirmation) message, and is often accompanied by data, which is packaged as one or more TRANSID_AI messages.
- **Response indicating failure.** This type of response is represented as a REF (refusal) message.

These message types and their relationship to one another are discussed in more detail in Section 5.3.2, "Message Naming Conventions and Structure".

5.3.2. Message Naming Conventions and Structure

This section describes the NDB Protocol message types and their structures.

Naming Conventions. Message names are constructed according to a simple pattern which should be readily apparent from the discussion of request and response types in the previous section. These are shown in the following matrix:

Operation Type	Request (**REQ**)	Response/Success (**CONF**)	Response/Failure (**REF**)
Primary Key Lookup (TCKEY)	TCKEYREQ	TCKEYCONF	TCKEYREF
Unique Key Lookup (TCINDX)	TCINDXREQ	TCINDXCONF	TCINDXREF
Table or Index Scan (SCANTAB)	SCANTABREQ	SCANTABCONF	SCANTABREF
Result Retrieval (SCAN_NEXT)	SCAN_NEXTREQ	SCANTABCONF	SCANTABREF
Transaction Record Acquisition (TCSEIZE)	TCSEIZEREQ	TCSEIZECONF	TCSEIZEREF
Transaction Record Release (TCRELEASE)	TCRELEASEREQ	TCRELEASECONF	TCRELEASEREF

CONF and REF are shorthand for "confirmed" and "refused", respectively.

Three additional types of messages are used in some instances of inter-node communication. These message types are:

1. A KEYINFO message contains information about the key used in a TCKEYREQ or TCINDXREQ message. It is employed when the key data does not fit within the request message. KEYINFO messages are also sent for index scan operations in which bounds are employed.

2. An ATTRINFO message contains non-key attribute values which does not fit within a TCKEYREQ, TCINDXREQ, or SCANTABREQ message. It is used for:

   - Supplying attribute values for inserts and updates
   - Designating which attributes are to be read for read operations
   - Specifying optional values to read for delete operations

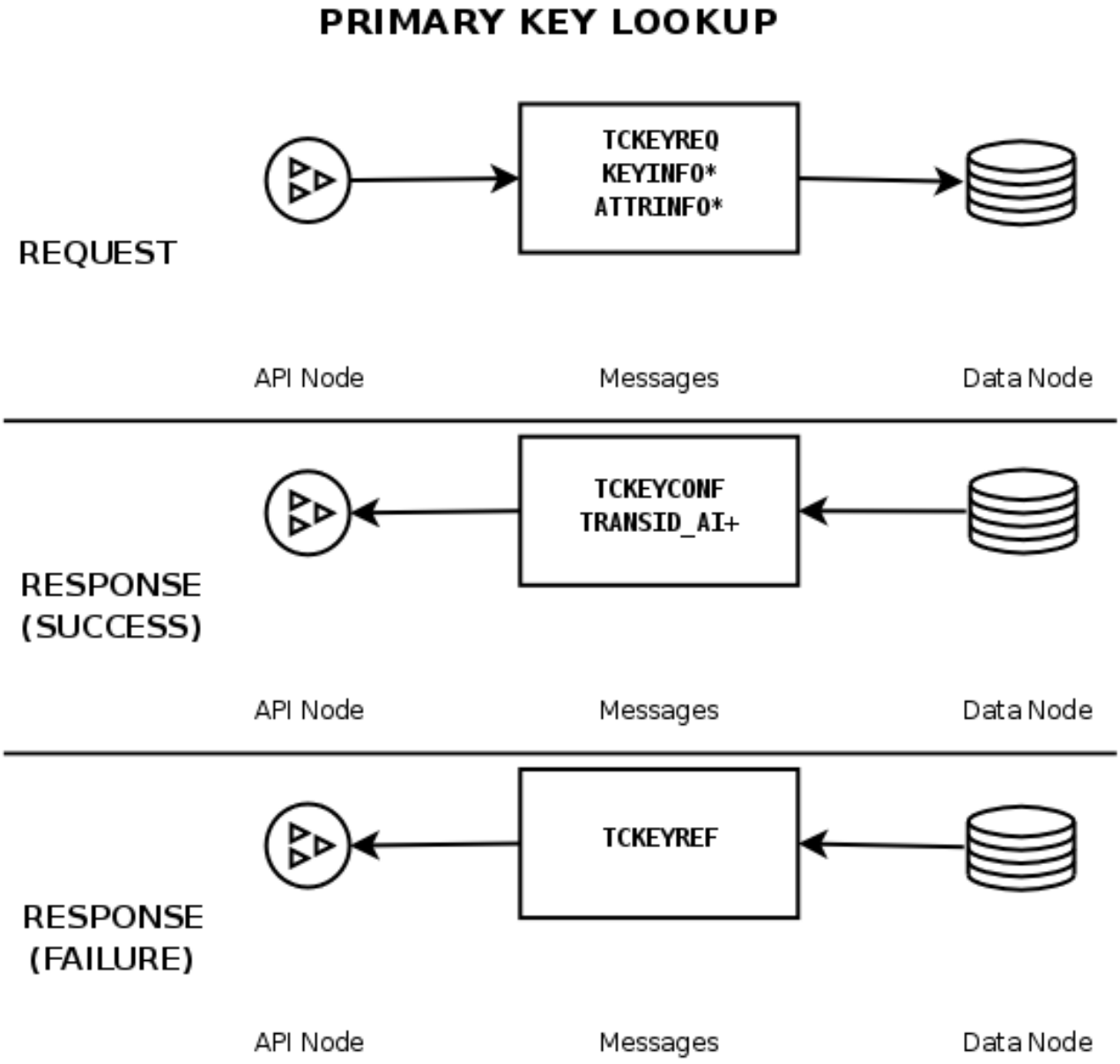
3. A TRANSID_AI message contains data returned from a read operation — in other words, it is a result set (or part of one).

5.3.3. Operations and Signals

In this section we discuss the sequence of message-passing that takes place between a data node and an API node for each of the following operations:

- Primary key lookup
- Unique key lookup
- Table scan or index scan
- Explicit commit of a transaction
- Rollback of a transaction
- Transaction record handling (acquisition and release)

Primary key lookup. An operation using a primary key lookup is performed shown in the following diagram:



Note

***** and + are used here with the meanings “zero or more” and “one or more”, respectively.

This process is explained is greater detail here:

1. The API node sends a TCKEYREQ message to the data node. In the event that the necessary information about the key to be used is too large to be contained in the TCKEYREQ, the message may be accompanied by any number of KEYINFO messages carrying the remaining key information. If additional attributes are used for the operation and exceed the space available in the TCKEYREQ, or if data is to be sent to the data node as part of a write operation, then these are sent with the TCKEYREQ as any number of ATTRINFO messages.

2. The data node then sends a message in response to the request:

   - If the operation was successful, the data node sends a TCKEYCONF message to the API node. If the request was for a read operation, then TCKEYCONF is accompanied by a TRANSID_AI message, which contains actual result data. If there is more data than can be contained in a single TRANSID_AI can carry, more than one of these messages may be sent.

   - If the operation failed, then the data node sends a TCKEYREF message back to the API node, and no more signalling takes place until the API node makes a new request.

Unique key lookup. This is performed in a manner similar to that performed in a primary key lookup:

1. A request is made by the API node using a TCINDXREQ message which may be accompanied by zero or more KEYINFO messages, zero or more ATTRINFO messages, or both.

2. The data node returns a response:

   - If the operation was a success, the message is TCINDXCONF. For a successful read operation, this message may be accompanied by one or more TRANSID_AI messages carrying the result data.

   - If the operation failed, the data node returns a TCINDXREF message.

The exchange of messages involved in a unique key lookup is illustrated here:

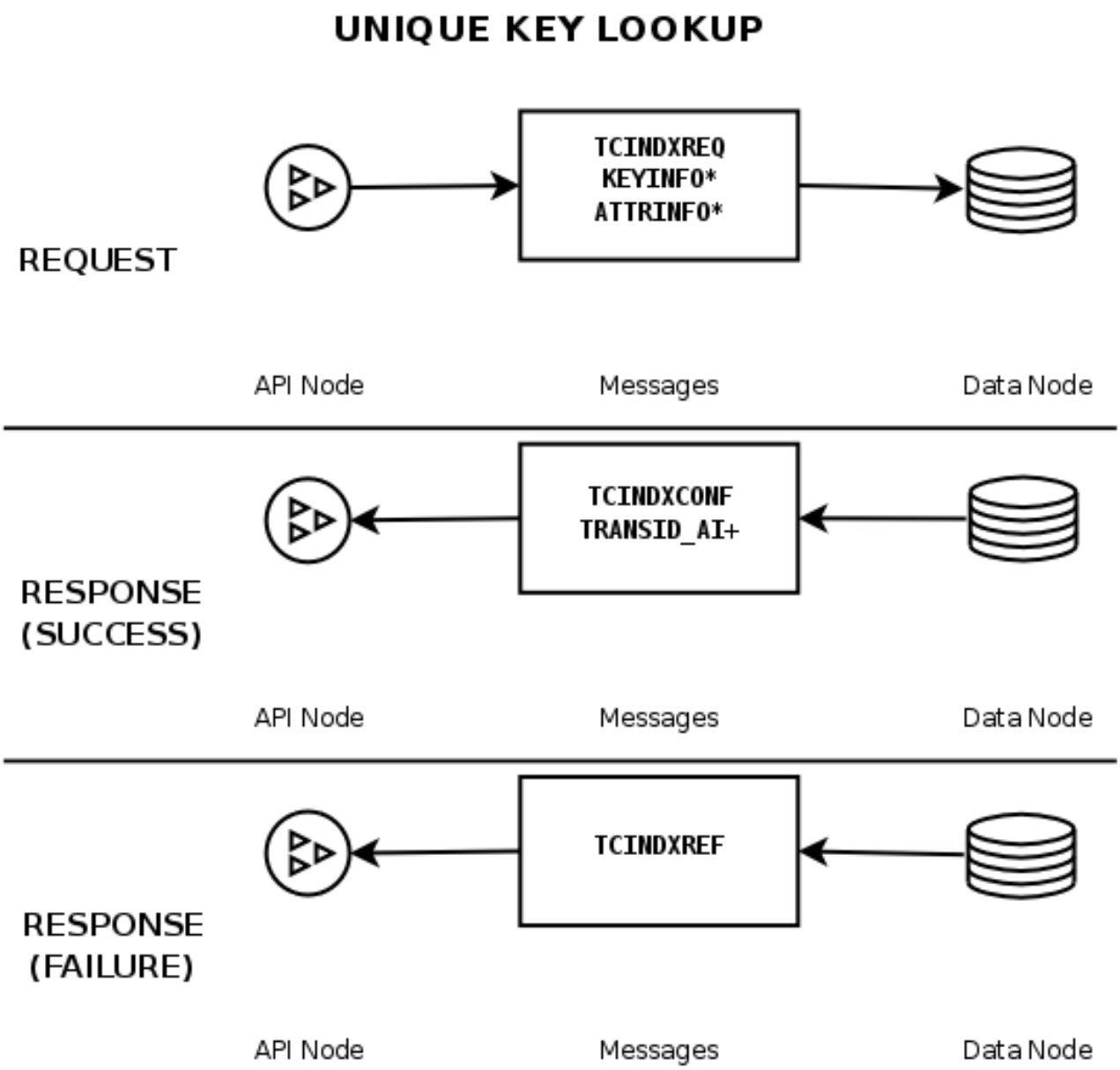
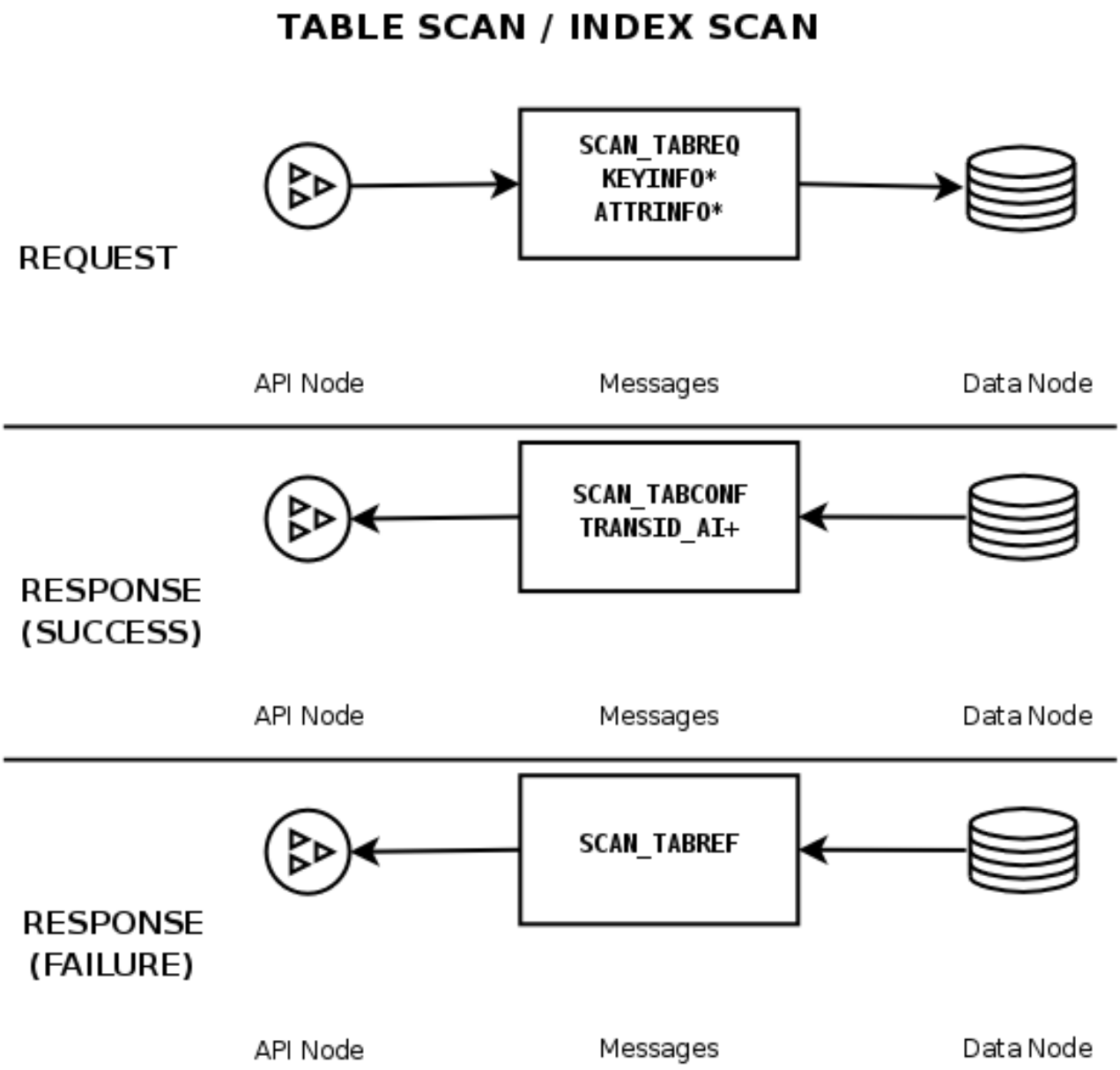


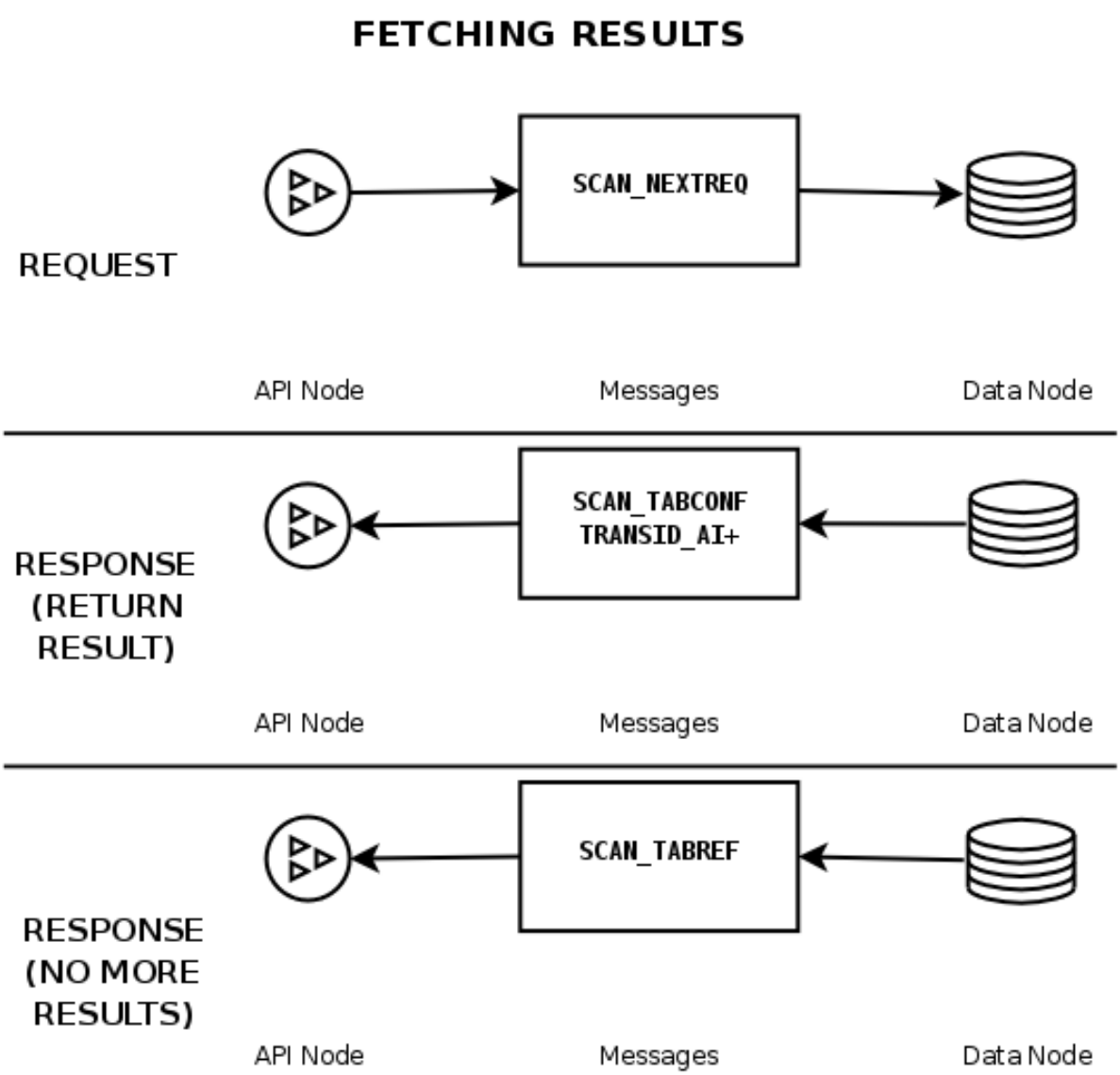
Table scans and index scans. These are similar in many respects to primary key and unique key lookups, as shown here:



1. A request is made by the API node using a `SCAN_TABREQ` message, along with zero or more `ATTRINFO` messages. `KEYINFO` messages are also used with index scans in the event that bounds are used.

2. The data node returns a response:

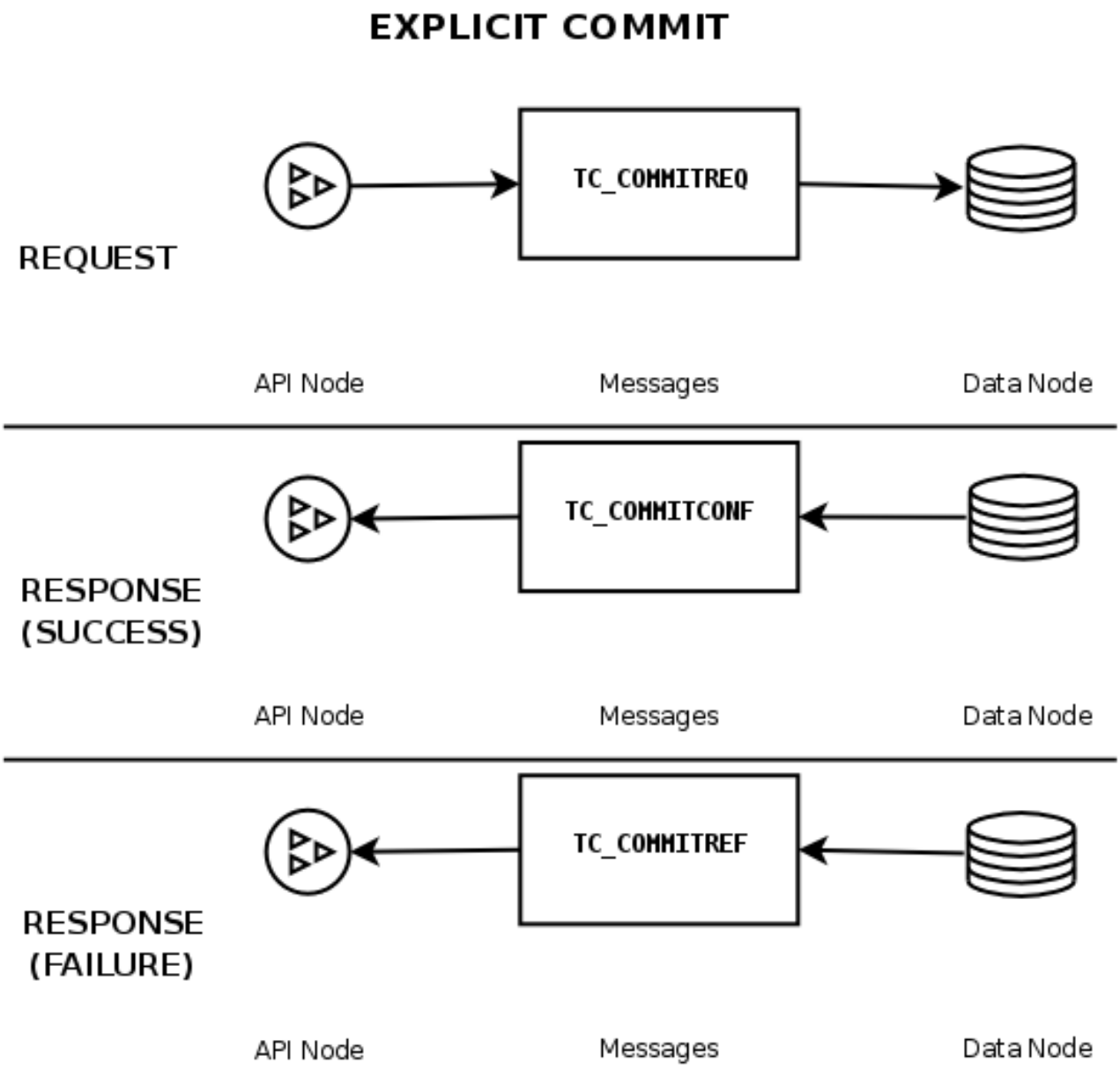
   - If the operation was a success, the message is `SCAN_TABCONF`. For a successful read operation, this message may be accompanied by one or more `TRANSID_AI` messages carrying the result data. However — unlike the case with lookups based on a primary or unique key — it is often necessary to fetch multiple results from the data node. Requests following the first are signalled by the API node using a `SCAN_NEXTREQ`, which tells the data node to send the next set of results (if there are more results). This is shown here:



- If the operation failed, the data node returns a `SCAN_TABREF` message.

  `SCAN_TABREF` is also used to signal to the API node that all data resulting from a read has been sent.

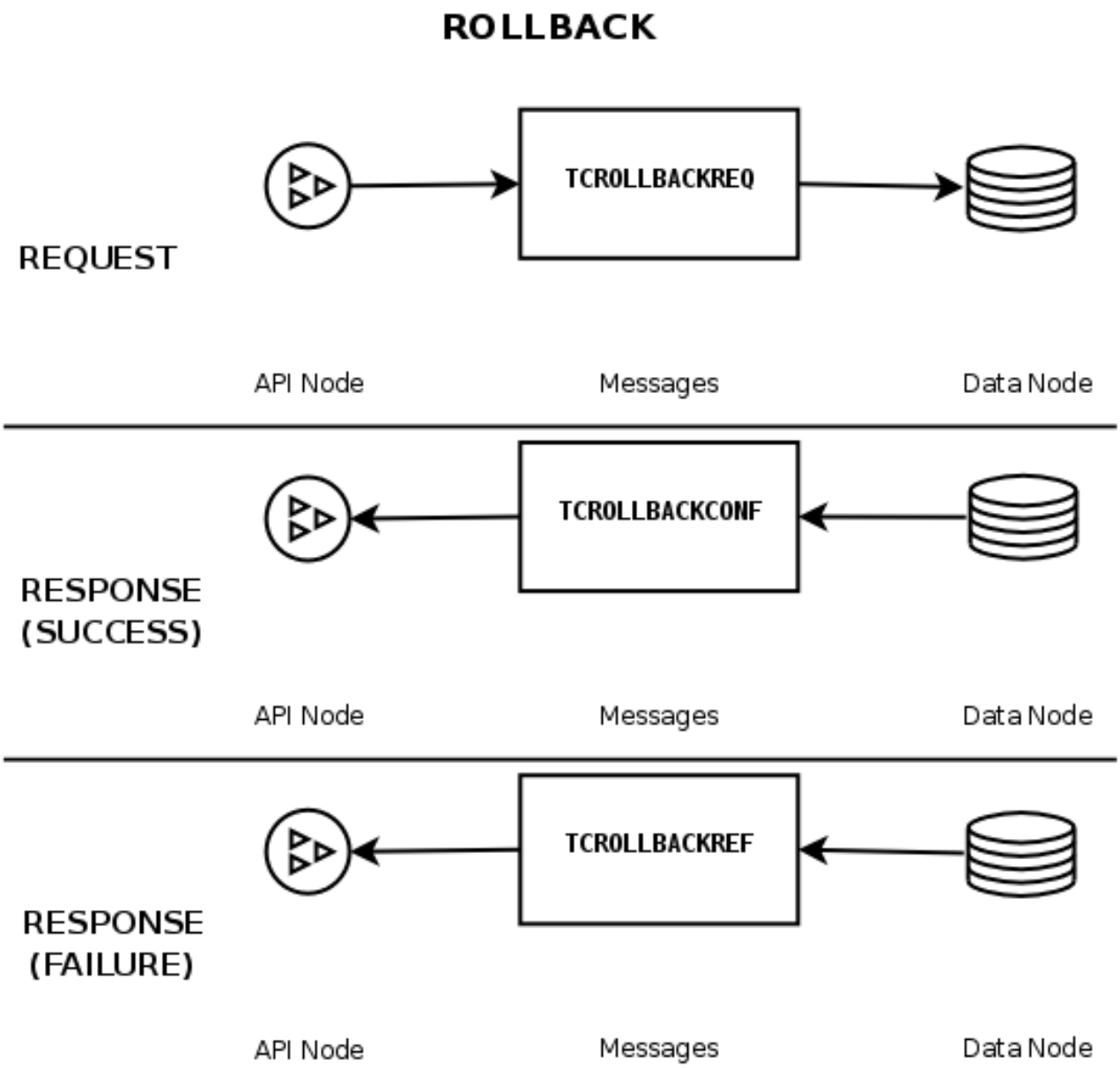
Committing and rolling back transactions. The process of performing an explicit commit follows the same general pattern as shown previously. The API node sends a `TC_COMMITREQ` message to the data node, which responds with either a `TC_COMMITCONF` (on success) or a `TC_COMMITREF` (if the commit failed). This is shown in the following diagram:



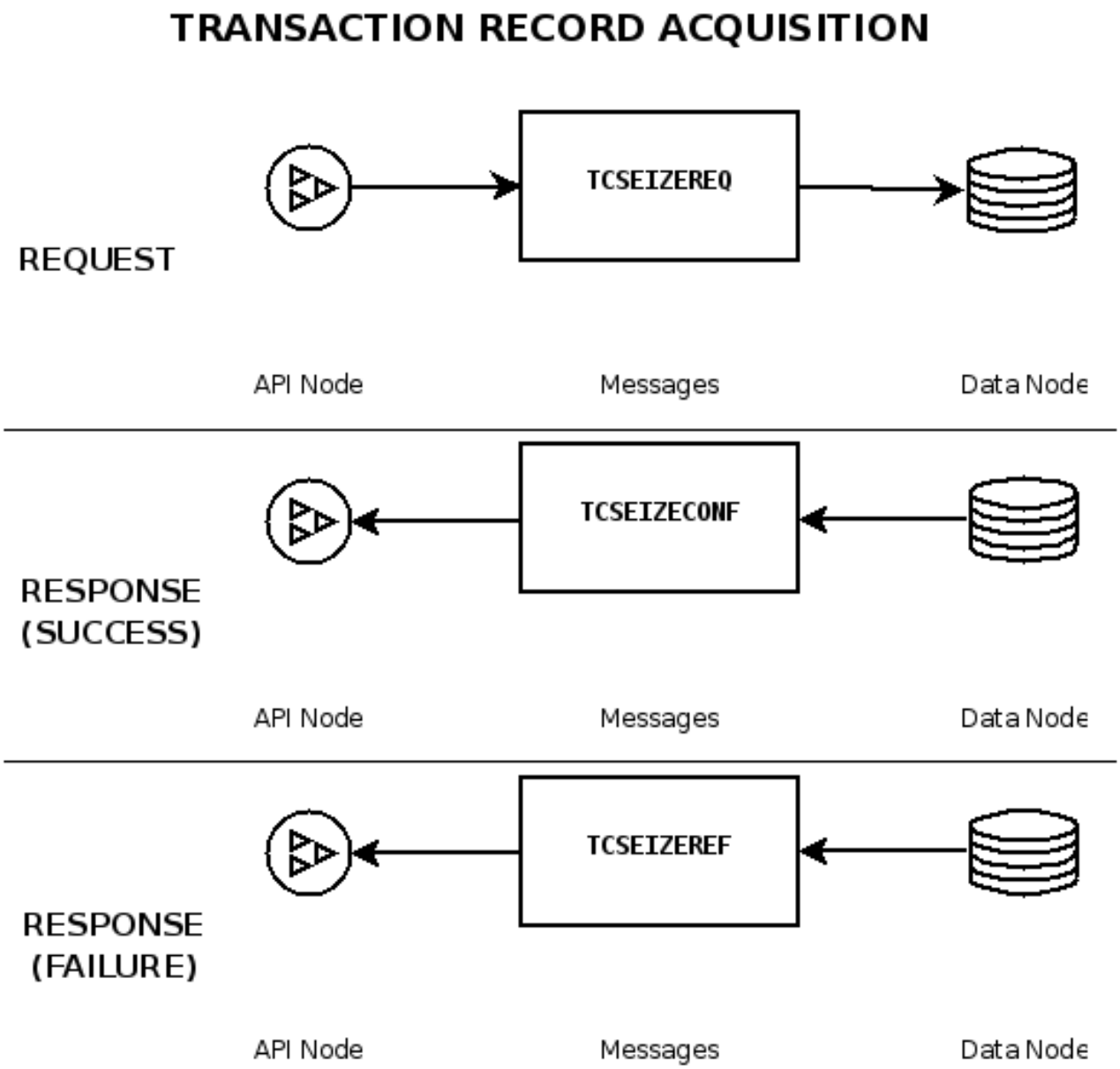
Note

Some operations perform a `COMMIT` automatically, so this is not required for every transaction.

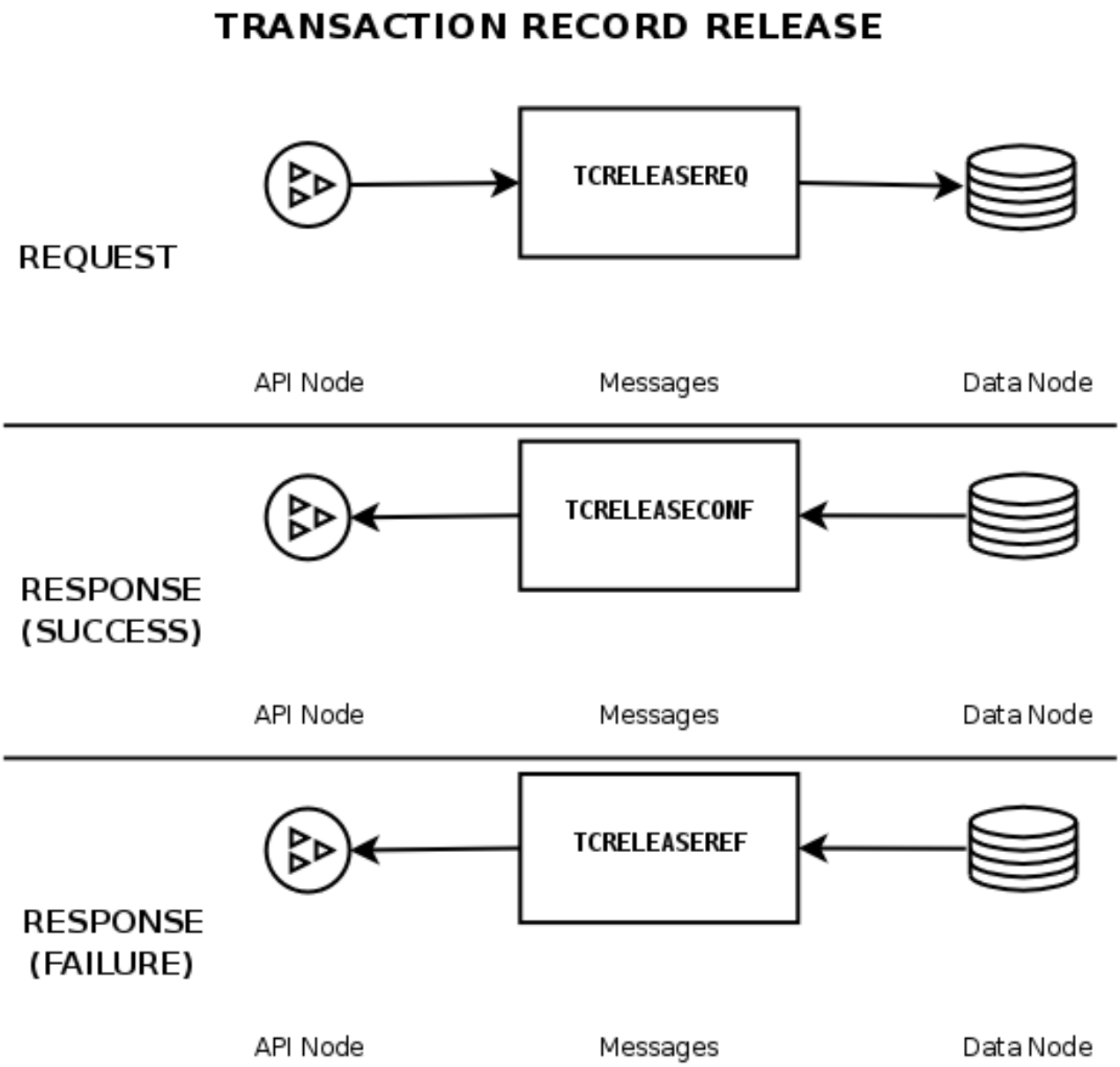
Rolling back a transaction also follows this pattern. In this case, however, the API node sends a `TCROLLBACKTREQ` message to the data node. Either a `TCROLLACKCONF` or a `TCROLLBACKREF` is sent in response, as shown here:



Handling of transaction records. Acquiring a transaction record is accomplished when an API node transmits a `TCSEIZEREQ` message to a data node and receives a `TCSEIZECONF` or `TCSEIZEREF` in return, depending on whether or not the request was successful. This is depicted here:



The release of a transaction record is also handled using the request-response pattern. In this case, the API node's request contains a `TCRELEASEREQ` message, and the data node's response uses either a `TCRELEASECONF` (indicating that the record was released) or a `TCRELEASEREF` (indicating that the attempt at release did not succeed). This series of events is illustrated in the next diagram:



5.4. NDB Kernel Blocks

The following sections list and describe the major kernel blocks found in the `NDB` source code. These are found under the directory `storage/ndb/src/kernel/blocks/` in the MySQL source code tree.

5.4.1. The BACKUP Block

This block is responsible for handling online backups and checkpoints. It is found in `storage/ndb/src/kernel/blocks/backup/`, and contains the following files:

- **`Backup.cpp`.** Defines methods for node signal handing; also provides output methods for backup status messages to user.
- **`BackupFormat.hpp`.** Defines the formats used for backup data, `.CTL`, and log files.
- **`Backup.hpp`.** Defines the `Backup` class.
- **`BackupInit.cpp`.** Actual `Backup` class constructor is found here.
- **`Backup.txt`.** Contains a backup signal diagram (text format). Somewhat dated (from 2003), but still potenitally useful to understanding the sequence of events that is followed during backups.

- **FsBuffer.hpp.** Defines the FsBuffer class, which implements the circular data buffer that is used (together with the NDB file system) for reading and writing backup data and logs.
- **read.cpp.** Contains some utility functions for reading log and checkpoint files to STDOUT.

5.4.2. The CMVMI Block

This block is responsible for configuration management between the kernel blocks and the NDB virtual machine, as well as the cluster job que and cluster transporters. It is found in storage/ndb/src/kernel/blocks/cmvmi, and contains these files:

- **Cmvmi.cpp.** Implements communication and reporting methods for the Cmvmi class.
- **Cmvmi.hpp.** Defines the Cmvmi class.

CMVMI is implemented as the Cmvmi class, defined in storage/ndb/src/kernel/blocks/cmvmi/Cmvmi.hpp.

5.4.3. The DBACC Block

Also referred to as the ACC block, this is the access control and lock management module, found in storage/ndb/src/kernel/blocks/dbacc. It contains the following files:

- **Dbacc.hpp.** Defines the Dbacc class, along with structures for operation, scan, table, and other records.
- **DbaccInit.cpp.** Dbacc class constructor and destructor; methods for initialising data and records.
- **DbaccMain.cpp.** Implements Dbacc class methods.

The ACC block handles the database index structures, which are stored in 8K pages. Database locks are also handled in the ACC block.

When a new tuple is inserted, the TUP block stores the tuple in a suitable space and returns an index (a reference to the address of the tuple in memory). ACC stores both the primary key and this tuple index of the tuple in a hash table.

Like the TUP block, the ACC block implements part of the checkpoint protocol. It also performs undo logging. It is implemented by the Dbacc class, which is defined in storage/ndb/src/kernel/blocks/dbacc/DbaccMain.hpp.

See also Section 5.4.8, “The DBTUP Block”.

5.4.4. The DBDICT Block

This block, the data dictionary block, is found in storage/ndb/src/kernel/blocks/dbdict. Data dictionary information is replicated to all DICT blocks in the cluster. This is the only block other than DBTC to which applications can send direct requests. DBDICT is responsible for managing metadata (via the cluster's master node) such as table and index definitions, as well as many Disk Data operations. This block contains the following files:

- **CreateIndex.txt.** Contains notes about processes for creating, altering, and dropping indexes and triggers.
- **Dbdict.cpp.** Implements structure for event metadata records (for NDB$EVENTS_0), as well as methods for system start and restart, table and schema file handling, and packing table data into pages. Functionality for determining node status and handling node failures is also found here. In addition, this file implements data and other initialisation routines for Dbdict.
- **DictLock.txt.** Implementation notes: Describes locking of the master node's DICT against schema operations.
- **printSchemaFile.cpp.** Contains the source for the ndb_print_schema_file utility, described in ndb_print_schema_file.
- **Slave_AddTable.sfl.** A signal log trace of a table creation operation for DBDICT on a non-master node.
- **CreateTable.txt.** Notes outlining the table creation process (dated).
- **CreateTable.new.txt.** Notes outlining the table creation process (updated version of CreateTable.txt).
- **Dbdict.hpp.** Defines the Dbdict class; also creates the NDB$EVENTS_0 table. Also defines a number of structures such as table and index records, as well as for table records.

- **DropTable.txt.** Implementation notes for the process of dropping a table.
- **Dbdict.txt.** Implementation notes for creating and dropping events and NdbEventOperation objects (see Section 2.3.11, "The NdbEventOperation Class").
- **Event.txt.** A copy of Dbdict.txt.
- **Master_AddTable.sfl.** A signal log trace of a table creation operation for DBDICT on the master node.
- **SchemaFile.hpp.** Defines the structure of a schema file.

This block is implemented as the Dbdict class, defined in storage/ndb/src/kernel/blocks/dblqh/Dbdict.hpp.

5.4.5. The DBDIH Block

This block provides data distribution (partitioning) management services. It is responsible for maintaining data fragments and replicas, handling of local and global checkpoints; it also manages node and system restarts. It contains the following files, all found in the directory storage/ndb/src/kernel/blocks/dbdih:

- **Dbdih.hpp.** This file contains the definition of the Dbdih class, as well as the FileRecordPtr type, which is used to keep storage information about a fragment and its replicas. If a fragment has more than one backup replica, then a list of the additional ones is attached to this record. This record also stores the status of the fragment, and is 64-byte aligned.
- **DbdihMain.cpp.** Contains definitions of Dbdih class methods.
- **printSysfile/printSysfile.cpp.** Older version of the printSysfile.cpp in the main dbdih directory.
- **DbdihInit.cpp.** Initializes Dbdih data and records; also contains the class destructor.
- **LCP.txt.** Contains developer noted about the exchange of messages between DIH and LQH that takes place during a local checkpoint.
- **printSysfile.cpp.** This file contains the source code for ndb_print_sys_file. For information about using this utility, see ndb_print_sys_file.
- **Sysfile.hpp.** Contains the definition of the Sysfile structure; in other words, the format of an NDB system file. See Section 5.1, "MySQL Cluster File Systems", for more information about NDB system files.

This block often makes use of BACKUP blocks on the data nodes to accomplish distributed tasks, such as global checkpoints and system restarts.

This block is implemented as the Dbdih class, whose definition may be found in the file storage/ndb/src/kernel/blocks/dbdih/Dbdih.hpp.

5.4.6. DBLQH Block

This is the local, low-level query handler block, which manages data and transactions local to the cluster's data nodes, and acts as a coordinator of 2-phase commits. It is responsible (when called on by the transaction coordinator) for performing operations on tuples, accomplishing this task with help of DBACC block (which manages the index structures) and DBTUP (which manages the tuples). It is made up of the following files, found in storage/ndb/src/kernel/blocks/dblqh:

- **Dblqh.hpp.** Contains the Dblqh class definition. The code itself includes the following modules:
  - **Start/Restart Module.** This module handles the following start phases:
    - **Start phase 1.** Load block reference and processor ID
    - **Start phase 2.** Initiate all records within the block; connect LQH with ACC and TUP
    - **Start phase 4.** Connect each LQH with every other LQH in the database system. For an initial start, create the fragment log files. For a system restart or node restart, open the fragment log files and find the end of the log files.
  - **Fragment addition and deletion module.** Used by the data dictionary to create new fragments and delete old fragments.
  - **Execution module.** This module handles the reception of LQHKEYREQ messages and all processing of operations on behalf of this request. This also involves reception of various types of ATTRINFO and KEYINFO messages, as well as communications with ACC and TUP.

- **Log module.** The log module handles the reading and writing of the log. It is also responsible for handling system restarts, and controls system restart in `TUP` and `ACC` as well.
- **Tramsaction module.** This module handles the commit and completion phases.
- **TC failure module.** Handles failures in the transaction coordinator.
- **Scan module.** This module contains the code that handles a scan of a particular fragment. It operates under the control of the transaction coordinator and orders `ACC` to perform a scan of all tuples in the fragment. `TUP` performs the necessary search conditions to insure that only valid tuples are returned to the application.
- **Node recovery module.** This is used when a node has failed, copying the effected fragment to a new fragment replica. It also shuts down all connections to the failed node.
- **LCP module.** This module handles execution and control of local checkpoints in `TUP` and `ACC`. It also interacts with `DIH` to determine which global checkpoints are recoverable.
- **GLobal checkpoint module.** Assist `DIH` in discovering when GCPs are recoverable. It handles the `GCP_SAVEREQ` message requesting `LQH` to save a given GCP to disk, and to provide a notification of when this has been done.
- **File handling module.** This includes a number of sub-modules:
  - Signal reception
  - Normal operation
  - File change
  - Initial start
  - System restart, Phase 1
  - System restart, Phase 2
  - System restart, Phase 3
  - System restart, Phase 4
  - Error

- **`DblqhInit.cpp`.** Initialises `Dblqh` records and data. Also includes the `Dblqh` class destructor, used for deallocating these.
- **`DblqhMain.cpp`.** Implements `Dblqh` functionality (class methods).
- This directory also has the files listed here in a `redoLogReader` subdirectory containing the sources for the `ndb_redo_log_reader` utility (see `ndbd_redo_log_reader`):
  - `records.cpp`
  - `records.hpp`
  - `redoLogFileReader.cpp`

This block also handles redo logging, and helps oversee the `DBACC`, `DBTUP`, `LGMAN`, `TSMAN`, `PGMAN`, and `BACKUP` blocks. It is implemented as the class `Dblqh`, defined in the file `storage/ndb/src/kernel/blocks/dblqh/Dblqh.hpp`.

5.4.7. The `DBTC` Block

This is the transaction coordinator block, which handles distributed transactions and other data operations on a global level (as opposed to `DBLQH` which deals with such issues on individual data nodes). In the source code, it is located in the directory `storage/ndb/src/kernel/blocks/dbtc`, which contains these files:

- **`Dbtc.hpp`.** Defines the `Dbtc` class and associated constructs, including the following:
  - **Trigger and index data (`TcDefinedTriggerData`).** A record forming a list of active triggers for each table. These records are managed by a trigger pool, in which a trigger record is seized whenever a trigger is activated, and released when the trigger is deactivated.
  - **Fired trigger data (`TcFiredTriggerData`).** A record forming a list of fired triggers for a given transaction.

- **Index data (`TcIndexData`).** This record forms lists of active indexes for each table. Such records are managed by an index pool, in which each index record is seized whenever an index is created, and released when the index is dropped.
- **API connection record (`ApiConnectRecord`).** An API connect record contains the connection record to which the application connects. The application can send one operation at a time. It can send a new operation immediately after sending the previous operation. This means that several operations can be active in a single transaction within the transaction coordinator, which is achieved by using the API connect record. Each active operation is handled by the TC connect record; as soon as the TC connect record has sent the request to the local query handler, it is ready to receive new operations. The `LQH` connect record takes care of waiting for an operation to complete; when an operation has completed on the `LQH` connect record, a new operation can be started on the current `LQH` connect record. `ApiConnectRecord` is always 256-byte aligned.
- **Transaction coordinator connection record (`TcConnectRecord`).** A `TcConnectRecord`) keeps all information required for carrying out a transaction; the transaction controller establishes connections to the different blocks needed to carry out the transaction. There can be multiple records for each active transaction. The TC connection record cooperates with the API connection record for communication with the API node, and with the `LQH` connection record for communication with any local query handlers involved in the transaction. `TcConnectRecord`) is permanently connected to a record in `DBDICT` and another in `DIH`, and contains a list of active `LQH` connection records and a list of started (but not currently active) `LQH` connection records. It also contains a list of all operations that are being executed with the current TC connection record. `TcConnectRecord` is always 128-byte aligned.
- **Cache record (`CacheRecord`).** This record is used between reception of a `TCKEYREQ` and sending of `LQHKEYREQ` (see Section 5.3.3, “Operations and Signals”) This is a separate record, so as to improve the cache hit rate and as well as to minimise memory storage requirements.
- **Host record (`HostRecord`).** This record contains the “alive” status of each node in the system, and is 128-byte aligned.
- **Table record (`TableRecord`).** This record contains the current schema versions of all tables in the system.
- **Scan record (`ScanRecord`).** Each scan allocates a `ScanRecord` to store information about the current scan.
- **Data buffer (`DatabufRecord`).** This is a buffer used for general data storage.
- **Attribute information record (`AttrbufRecord`).** This record can contain one (1) `ATTRINFO` signal, which contains a set of 32 attribute information words.
- **Global checkpoint information record (`GcpRecord`).** This record is used to store the globalcheckpoint number, as well as a counter, during the completion phase of the transaction. A `GcpRecord` is 32-byte aligned.
- **TC failure record (`TC_FAIL_RECORD`).** This is used when handling takeover of TC duties from a failed transaction coordinator.
-

- **`DbtcInit.cpp`.** Handles allocation and deallocation of `Dbtc` indexes and data (includes class desctructor).
- **`DbtcMain.cpp`.** Implements `Dbtc` methods.

> **Note**
>
> Any data node may act as the transaction coordinator.

The `DBTC` block is implemented as the `Dbtc` class.

The transaction coordinator is the kernel interface to which applications send their requests. It establishes connections to different blocks in the system to carry out the transaction and decides which node will handle each transaction, sending a confirmation signal on the result to the application so that the application can verify that the result received from the TUP block is correct.

This block also handles unique indexes, which must be co-ordinated across all data nodes simultaneously.

5.4.8. The `DBTUP` Block

This is the tuple manager, which manages the physical storage of cluster data. It consists of the following files found in the directory `storage/ndb/src/kernel/blocks/dbtup`:

- **`AttributeOffset.hpp`.** Defines the `AttributeOffset` class, which models the structure of an attribute, allowing for 4096 attributes, all of which are nullable.
- **`DbtupDiskAlloc.cpp`.** Handles allocation and deallocation of extents for disk space.

- **DbtupIndex.cpp.** Implements methods for reading and writing tuples using ordered indexes.
- **DbtupScan.cpp.** Implements methods for tuple scans.
- **tuppage.cpp.** Handles allocating pages for writing tuples.
- **tuppage.hpp.** Defines structures for fixed and variable size data pages for tuples.
- **DbtupAbort.cpp.** Contains routines for terminating failed tuple operations.
- **DbtupExecQuery.cpp.** Handles execution of queries for tuples and reading from them.
- **DbtupMeta.cpp.** Handle table operations for the `Dbtup` class.
- **DbtupStoredProcDef.cpp.** Module for adding and dropping procedures.
- **DbtupBuffer.cpp.** Handles read/write buffers for tuple operations.
- **DbtupFixAlloc.cpp.** Allocates and frees fixed-size tuples from the set of pages attatched to a fragment. The fixed size is set per fragment; there can be only one such value per fragment.
- **DbtupPageMap.cpp.** Routines used by `Dbtup` to map logical page IDs to physical page IDs. The mapping needs the fragment ID and the logical page ID to provide the physical ID. This part of `Dbtup` is the exclusive user of a certain set of variables on the fragment record; it is also the exclusive user of the struct for page ranges (the `PageRange` struct defined in `Dbtup.hpp`).
- **DbtupTabDesMan.cpp.** This file contains the routines making up the table descriptor memory manager. Each table has a descriptor, which is a contiguous array of data words, and which is allocated from a global array using a "buddy" algorithm, with free lists existing for each 2^N words.
- **Notes.txt.** Contains some developers' implementation notes on tuples, tuple operations, and tuple versioning.
- **Undo_buffer.hpp.** Defines the `Undo_buffer` class, used for storage of operations that may need to be rolled back.
- **Undo_buffer.cpp.** Implements some necessary `Undo_buffer` methods.
- **DbtupCommit.cpp.** Contains routines used to commit operations on tuples to disk.
- **DbtupGen.cpp.** This file contains `Dbtup` initialization routines.
- **DbtupPagMan.cpp.** This file implements the page memory manager's "buddy" algorithm. `PagMan` is invoked when fragments lack sufficient internal page space to accomodate all the data they are requested to store. It is also invoked when fragments deallocate page space back to the free area.
- **DbtupTrigger.cpp.** The routines contained in this file perform handling of `NDB` internal triggers.
- **DbtupDebug.cpp.** Used for debugging purposes only.
- **Dbtup.hpp.** Contains the `Dbtup` class definition. Also defines a number of essential structures such as tuple scans, disk allocation units, fragment records, and so on.
- **DbtupRoutines.cpp.** Implements `Dbtup` routines for reading attributes.
- **DbtupVarAlloc.cpp.**
- **test_varpage.cpp.** Simple test program for verifying variable-size page operations.

This block also monitors changes in tuples.

5.4.9. DBTUX Block

This kernel block handles the local management of ordered indexes. It consists of the following files found in the `storage/ndb/src/kernel/blocks/dbtux` directory:

- **DbtuxCmp.cpp.** Implements routines to search by key *vs* node prefix or entry. The comparison starts at a given attribute position, which is updated by the number of equal initial attributes found. The entry data may be partial, in which case `CmpUnknown` may be returned. The attributes are normalized and have a variable size, given in words.
- **DbtuxGen.cpp.** Implements initialization routines used in node starts and restarts.

- **DbtuxMaint.cpp.** Contains routines used to maintain indexes.
- **DbtuxNode.cpp.** Implements routines for node creation, allocation, and deletion operations. Also assigns lists of scans to nodes.
- **DbtuxSearch.cpp.** Provides routines for handling node scan request messages.
- **DbtuxTree.cpp.** Routines for performing node tree operations.
- **Times.txt.** Contains some (old) performance figures from tests runs on operations using ordered indexes. Of historical interest only.
- **DbtuxDebug.cpp.** Debugging code for dumping node states.
- **Dbtux.hpp.** Contains `Dbtux` class definition.
- **DbtuxMeta.cpp.** Routines for creating, setting, and dropping indexes. Also provides means of aborting these operations in the event of failure.
- **DbtuxScan.cpp.** Routines for performing index scans.
- **DbtuxStat.cpp.** Implements methods for obtaining node statistics.
- **tuxstatus.html.** 2004-01-30 status report on ordered index implementation. Of historical interest only.

5.4.10. The DBUTIL Block

This block provides internal interfaces to transaction and data operations, performing essential operations on signals passed between nodes. This block implements transactional services which can then be used by other blocks. It is also used in building online indexes, and is found in `storage/ndb/src/kernel/blocks/dbutil`, which includes these files:

- **DbUtil.cpp.** Implements `Dbutil` class methods
- **DbUtil.hpp.** Defines the `Dbutil` class, used to provide transactional services.
- **DbUtil.txt.** Implementation notes on utility protocols implemented by `DBUTIL`.

Among the duties performed by this block is the maintenance of sequences for backup IDs and other distributed identifiers.

5.4.11. The LGMAN Block

This block, the log group manager, is responsible for handling the undo logs for Disk Data tables. It consists of these files in the `storage/ndb/src/kernel/blocks` directory:

- **lgman.cpp.** Implements `Lgman` for adding, dropping, and working with log files and file groups.
- **lgman.hpp.** Contains the definition for the `Lgman` class, used to handle undo log files. Handles allocation of log buffer space.

5.4.12. The NDBCNTR Block

This is a cluster management block that handles block initialisation and configuration. During the data node startup process, it takes over from the `QMGR` block and continues the process. It also assist with graceful (planned) shutdowns of data nodes. Ths block is located in `storage/ndb/src/kernel/blocks/ndbcntr`, and contains these files:

- **Ndbcntr.hpp.** Defines the `Ndbcntr` class used to implement cluster management functions.
- **NdbcntrInit.cpp.** Initializers for `Ndbcntr` data and records.
- **NdbcntrMain.cpp.** Implements methods used for starts, restarts, and reading of configuration data.
- **NdbcntrSysTable.cpp.** `NDBCNTR` creates and initializes system tables on initial system start. The tables are defined in static structs in this file.

5.4.13. The NDBFS Block

This block provides the NDB file system abstraction layer, and is located in the directory `storage/ndb/src/kernel/blocks/ndbfs`, which contains the following files:

- **AsyncFile.hpp.** Defines the `AsyncFile` class, which represents an asynchronous file. All actions are executed concurrently with the other activities carried out by the process. Because all actions are performed in a separate thread, the result of an action is sent back through a memory channel. For the asyncronous notification of a finished request, each callincludes a request as a parameter. This class is used for writing or reading data to and from disk concurrently with other activities.
- **AsyncFile.cpp.** Defines the actions possible for an asynchronous file, and implements them.
- **Filename.hpp.** Defines the `Filename` class. Takes a 128-bit value (as a array of four longs) and makes a file name out of it. This file name encodes information about the file, such as whether it is a file or a directory, and if the former, the type of file. Possible types include data file, fragment log, fragment list, table list, schema log, and system file, among others.
- **Filename.cpp.** Implements `set()` methods for the `Filename` class.
- **MemoryChannelTest/MemoryChannelTest.cpp.** Basic program for testing reads from and writes to a memory channel (that is, reading from and writing to a circular buffer).
- **OpenFiles.hpp.** Implements an `OpenFiles` class, which implements some convenience methods for determining whether or not a given file is already open.
- **VoidFs.cpp.** Used for diskless operation. Generates a “dummy” acjknowledgement to write operations.
- **CircularIndex.hpp.** The `CircularIndex` class, defined in this file, serves as the building block for implementing circular buffers. It increments as a normal index until it reaches maximum size, then resets to zero.
- **CircularIndex.cpp.** Contains only a single `#define`, not actually used at this time.
- **MemoryChannel.hpp.** Defines the `MemoryChannel` and `MemoryChannelMultipleWriter` classes, which provide a pointer-based channel for communication between two threads. It does not copy any data into or out of the channel, so the item that is put in can not be used untill the other thread has given it back. There is no support for detecting the return of an item.
- **MemoryChannel.cpp.** “Dummy” file, not used at this time.
- **Ndbfs.hpp.** Because an `NDB` signal request can result in multiple requests to `AsyncFile`, one class (defined in this file) is responsible for keeping track of all outstanding requests, and when all are finished, reporting the outcome back to the sending block.
- **Ndbfs.cpp.** Implements initialization and signal-handling methods for the `Ndbfs` class.
- **Pool.hpp.** Creates and manages a pool of objects for use by `Ndbfs` and other classes in this block.
- **AsyncFileTest/AsyncFileTest.cpp.** Test program, used to test and benchmark functionality of `AsyncFile`.

5.4.14. The PGMAN Block

This block provides page and buffer management services for Disk Data tables. It includes these files:

- **diskpage.hpp.** Defines the `File_formats`, `Datafile`, and `Undofile` structures.
- **diskpage.cpp.** Initializes sero page headers; includes some output reoutines for reporting and debugging.
- **pgman.hpp.** Defines the `Pgman` class implementing a number of page and buffer services, including:
  - Page entries and requests
  - Page replacement
  - Page lists
  - Page cleanup
  - Other page processing

- **pgman.cpp.** Implements Pgman methods for initialization and various page management tasks.

5.4.15. The QMGR Block

This is the logical cluster management block, and handles node membership in the cluster via heartbeats. QMGR is responsible for polling the data nodes when a data node failure occurs and determining that the node has actually failed and should be dropped from the cluster. This block contains the following files, found in storage/ndb/src/kernel/blocks/qmgr:

- **Qmgr.hpp.** Defines the Qmgr class and associated structures, including those used in detection of node failure and cluster partitioning.
- **QmgrInit.cpp.** Implements data and record initilization methods for Qmgr, as well as its destructor.
- **QmgrMain.cpp.** Contains routines for monitoring of heartbeats, detection and handling of “split-brain” problems, and management of some startup phases.
- **timer.hpp.** Defines the Timer class, used by NDB to keep strict timekeeping independent of the system clock.

This block also assists in the early phases of data node startup.

The QMGR block is implemented by the Qmgr class, whose definition is found in the file storage/ndb/src/kernel/blocks/qmgr/Qmgr.hpp.

5.4.16. The RESTORE Block

This block consists of the files restore.cpp and restore.hpp, in the storage/ndb/src/kernel/blocks directory. It handles restoration of the cluster from online backups.

5.4.17. The SUMA Block

The cluster subscription manager, which handles event logging and reporting functions. It also figures prominently in MySQL CLuster Replication. SUMA consists of the following files, found in the directory storage/ndb/src/kernel/blocks/suma/:

- **Suma.hpp.** Defines the Suma class and interfaces for managing subscriptions and performing necessary communications with other SUMA (and other) blocks.
- **SumaInit.cpp.** Performs initialization of DICT, DIH, and other interfaces
- **Suma.cpp.** Implements subscription-handling routines.
- **Suma.txt.** Contains a text-based diagram illustrating SUMA protocols.

5.4.18. The TSMAN Block

This is the tablespace manager block for Disk Data tables, and includes the following files from storage/ndb/src/kernel/blocks:

- **tsman.hpp.** Defines the Tsman class, as well as structures representing data files and tablespaces.
- **tsman.cpp.** Implements Tsman methods.

5.4.19. The TRIX Block

This kernel block is responsible for the handling of internal triggers and unique indexes. TRIX, like DBUTIL, is a utility block containing many helper functions for building indexes and handling signals between nodes. It is found in the directory storage/ndb/src/kernel/blocks/trix, and includes these files:

- **Trix.hpp.** Defines the Trix class, along with structures representing subscription data and records (for communicating with SUMA) and node data and ists (needed when communicating with remote TRIX blocks).
- **Trix.cpp.** Implements Trix class methods, including those necessary for taking appropriate action in the event of node fail-

ures.

This block is implemented as the `Trix` class, defined in `storage/ndb/src/kernel/blocks/trix/Trix.hpp`.

5.5. MySQL Cluster Start Phases

5.5.1. Initialization Phase (Phase -1)

Before the data node actually starts, a number of other setup and initialization tasks must be done for the block objects, transporters, and watchdog checks, among others.

This initialization process begins in `storage/ndb/src/kernel/main.cpp` with a series of calls to `globalEmulatorData.theThreadConfig->doStart()`. When starting `ndbd` with the `-n` or `--nostart` option there is only one call to this method; otherwise, there are two, with the second call actually starting the data node. The first invocation of `doStart()` sends the `START_ORD` signal to the `CMVMI` block (see Section 5.4.2, “The CMVMI Block”); the second call to this method sends a `START_ORD` signal to `NDBCNTR` (see Section 5.4.12, “The NDBCNTR Block”).

When `START_ORD` is received by the `NDBCNTR` block, the signal is immediately transferred to `NDBCNTR`'s `MISSRA` sub-block, which handles the start process by sending a `READ_CONFIG_REQ` signals to all blocks in order as given in the array `readConfigOrder`:

1. `NDBFS`
2. `DBTUP`
3. `DBACC`
4. `DBTC`
5. `DBLQH`
6. `DBTUX`
7. `DBDICT`
8. `DBDIH`
9. `NDBCNTR`
10. `QMGR`
11. `TRIX`
12. `BACKUP`
13. `DBUTIL`
14. `SUMA`
15. `TSMAN`
16. `LGMAN`
17. `PGMAN`
18. `RESTORE`

`NDBFS` is allowed to run before any of the remaining blocks are contacted, in order to make sure that it can start the `CMVMI` block's threads.

5.5.2. Configuration Read Phase (`STTOR` Phase -1)

The `READ_CONFIG_REQ` signal provides all kernel blocks an opportunity to read the configuration data, which is stored in a global object accessible to all blocks. All memory allocation in the data nodes takes place during this phase.

Note

Connections between the kernel blocks and the NDB file system are also set up during Phase 0. This is necessary to enable the blocks to communicate easily which parts of a table structure are to be written to disk.

NDB performs memory allocations in two different ways. The first of these is by using the `allocRecord()` method (defined in `storage/ndb/src/kernel/vm/SimulatedBlock.hpp`). This is the traditional method whereby records are accessed using the `ptrCheckGuard` macros (defined in `storage/ndb/src/kernel/vm/pc.hpp`). The other method is to allocate memory using the `setSize()` method defined with the help of the template found in `storage/ndb/src/kernel/vm/CArray.hpp`.

These methods sometimes also initialize the memory, ensuring that both memory allocation and initialization are done with watchdog protection.

Many blocks also perform block-specific initialization, which often entails building linked lists or doubly-linked lists (and in some cases hash tables).

Many of the sizes used in allocation are calculated in the `Configuration::calcSizeAlt()` method, found in `storage/ndb/src/kernel/vm/Configuration.cpp`.

Some preparations for more intelligent pooling of memory resources have been made. `DataMemory` and disk records already belong to this global memory pool.

5.5.3. STTOR Phase 0

Most NDB kernel blocks begin their start phases at STTOR Phase 1, with the exception of NDBFS and NDBCNTR, which begin with Phase 0, as can be seen by inspecting the first value for each element in the ALL_BLOCKS array (defined in `src/kernel/blocks/ndbcntr/NdbcntrMain.cpp`). In addition, when the STTOR signal is sent to a block, the return signal STTORRY always contains a list of the start phases in which the block has an interest. Only in those start phases does the block actually receive a STTOR signal.

STTOR signals are sent out in the order in which the kernel blocks are listed in the ALL_BLOCKS array. While NDBCNTR goes through start phases 0 to 255, most of these are empty.

Both activities in Phase 0 have to do with initialization of the NDB file system. First, if necessary, NDBFS creates the file system directory for the data node. In the case of an initial start, NDBCNTR clears any existing files from the directory of the data node to ensure that the DBDIH block does not subsequently discover any system files (if DBDIH were to find any system files, it would not interpret the start correctly as an initial start). (See also Section 5.4.5, “The DBDIH Block”.)

Each time that NDBCNTR completes the sending of one start phase to all kernel blocks, it sends a NODE_STATE_REP signal to all blocks, which effectively updates the `NodeState` in all blocks.

Each time that NDBCNTR completes a non-empty start phase, it reports this to the management server; in most cases this is recorded in the cluster log.

Finally, after completing all start phases, NDBCNTR updates the node state in all blocks via a NODE_STATE_REP signal; it also sends an event report advising that all start phases are complete. In addition, all other cluster data nodes are notified that this node has completed all its start phases to ensure all nodes are aware of one another's state. Each data node sends a NODE_START_REP to all blocks; however, this is significant only for DBDIH, so that it knows when it can unlock the lock for schema changes on DBDICT.

Note

In the following table, and throughout this text, we sometimes refer to STTOR start phases simply as “start phases” or “Phase *N*” (where *N* is some number). NDB_STTOR start phases are always qualified as such, and so referred to as “NDB_STTOR start phases” or “NDB_STTOR phases”.

Kernel Block	Receptive Start Phases
NDBFS	0
DBTC	1
DBDIH	1
DBLQH	1, 4
DBACC	1
DBTUP	1
DBDICT	1, 3
NDBCNTR	0, 1, 2, 3, 4, 5, 6, 8, 9
CMVMI	1 (prior to QMGR), 3, 8

Kernel Block	Receptive Start Phases
QMGR	1, 7
TRIX	1
BACKUP	1, 3, 7
DBUTIL	1, 6
SUMA	1, 3, 5, 7, 100 (empty), 101
DBTUX	1,3,7
TSMAN	1, 3 (both ignored)
LGMAN	1, 2, 3, 4, 5, 6 (all ignored)
PGMAN	1, 3, 7 (Phase 7 currently empty)
RESTORE	1,3 (only in Phase 1 is any real work done)

> **Note**
>
> This table was current at the time this text was written, but is likely to change over time. The latest information can be found in the source code.

5.5.4. STTOR Phase 1

This is one of the phases in which most kernel blocks participate (see the table in Section 5.5.3, “STTOR Phase 0”). Otherwise, most blocks are involved primarily in the initialization of data — for example, this is all that DBTC does.

Many blocks initialize references to other blocks in Phase 1. DBLQH initializes block references to DBTUP, and DBACC initializes block references to DBTUP and DBLQH. DBTUP initializes references to the blocks DBLQH, TSMAN, and LGMAN.

NDBCNTR initializes some variables and sets up block references to DBTUP, DBLQH, DBACC, DBTC, DBDIH, and DBDICT; these are needed in the special start phase handling of these blocks using NDB_STTOR signals, where the bulk of the node startup process actually takes place.

If the cluster is configured to lock pages (that is, if the LockPagesInMainMemory configuration parameter has been set), CMVMI handles this locking.

The QMGR block calls the initData() method (defined in storage/ndb/src/kernel/blocks/qmgr/QmgrMain.cpp) whose output is handled by all other blocks in the READ_CONFIG_REQ phase (see Section 5.5.1, “Initialization Phase (Phase -1)”). Following these initializations, QMGR sends the DIH_RESTARTREQ signal to DBDIH, which determines whether a proper system file exists; if it does, an initial start is not being performed. After the reception of this signal comes the process of integrating the node among the other data nodes in the cluster, where data nodes enter the cluster one at a time. The first one to enter becomes the master; whenever the master dies the new master is always the node that has been running for the longest time from those remaining.

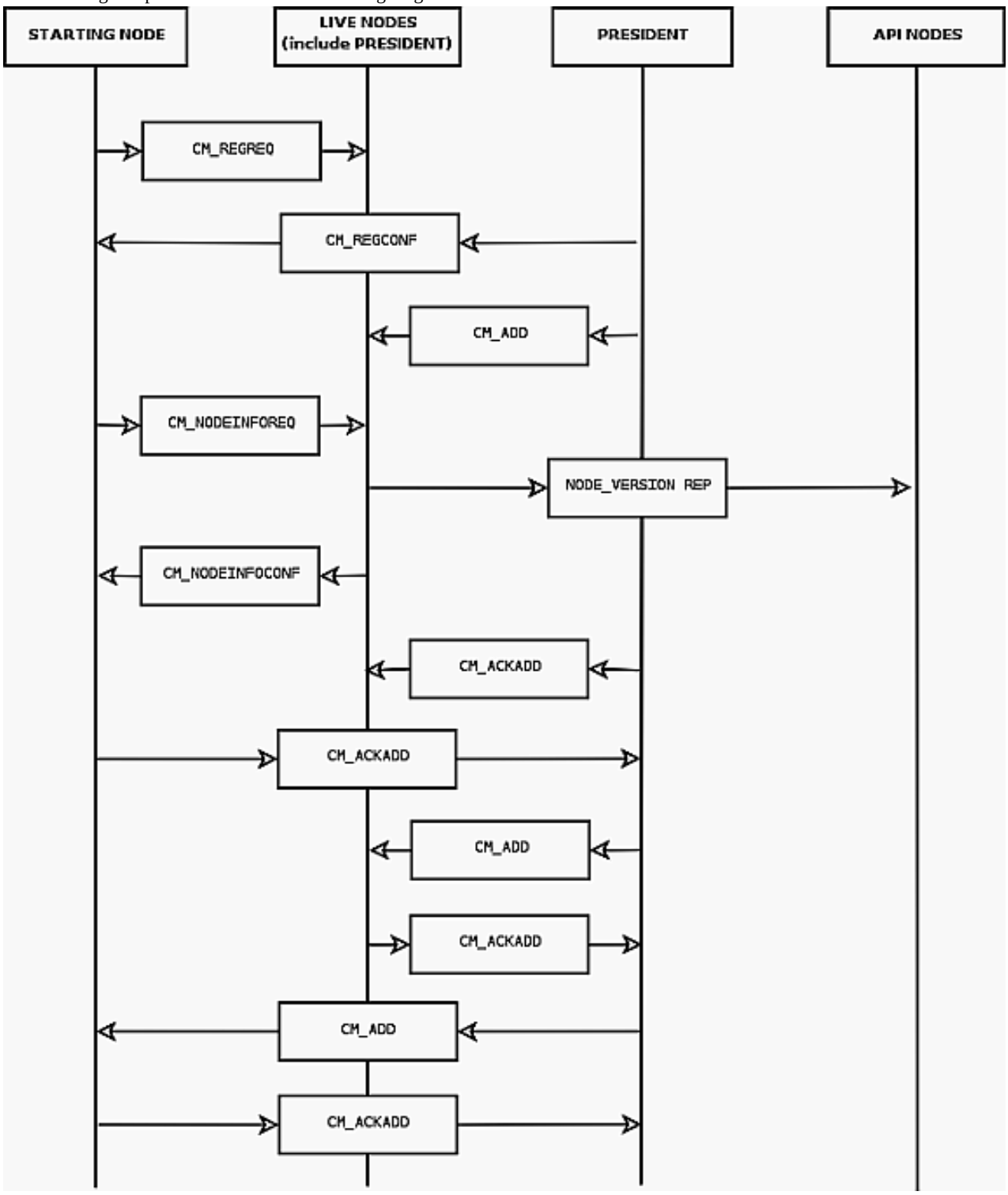
QMGR sets up timers to ensure that inclusion in the cluster does not take longer than what the cluster's configuration is set to allow (see Controlling Timeouts, Intervals, and Disk Paging for the relevant configuration parameters), after which communication to all other data nodes is established. At this point, a CM_REGREQ signal is sent to all data nodes. Only the president of the cluster responds to this signal; the president allows one node at a time to enter the cluster. If no node responds within 3 seconds then the president becomes the master. If several nodes start up simultaneously, then the node with the lowest node ID becomes president. The president sends CM_REGCONF in response to this signal, but also sends a CM_ADD signal to all nodes that are currently alive.

Next, the starting node sends a CM_NODEINFOREQ signal to all current “live” data nodes. When these nodes receive that signal they send a NODE_VERSION_REP signal to all API nodes that have connected to them. Each data node also sends a CM_ACKADD to the president to inform the president that it has heard the CM_NODEINFOREQ signal from the new node. Finally, each of the current data nodes sends the CM_NODEINFOCONF signal in response to the starting node. When the starting node has received all these signals, it also sends the CM_ACKADD signal to the president.

When the president has received all of the expected CM_ACKADD signals, it knows that all data nodes (including the newest one to start) have replied to the CM_NODEINFOREQ signal. When the president receives the final CM_ACKADD, it sends a CM_ADD signal to all current data nodes (that is, except for the node that just started). Upon receiving this signal, the existing data nodes enable communication with the new node; they begin sending heartbeats to it and including in the list of neighbors used by the heartbeat protocol.

The start struct is reset, so that it can handle new starting nodes, and then each data node sends a CM_ACKADD to the president, which then sends a CM_ADD to the starting node after all such CM_ACKADD signals have been received. The new node then opens all of its communication channels to the data nodes that were already connected to the cluster; it also sets up its own heartbeat structures and starts sending heartbeats. It also sends a CM_ACKADD message in response to the president.

The signalling between the starting data node, the already “live” data nodes, the president, and any API nodes attached to the cluster during this phase is shown in the following diagram:



As a final step, QMGR also starts the timer handling for which it is responsible. This means that it generates a signal to blocks that have requested it. This signal is sent 100 times per second even if any one instance of the signal is delayed..

The BACKUP kernel block also begins sending a signal periodically. This is to ensure that excessive amounts of data are not written to disk, and that data writes are kept within the limits of what has been specified in the cluster configuration file during and after restarts. The DBUTIL block initializes the transaction identity, and DBTUX creates a reference to the DBTUP block, while PGMAN initializes pointers to the LGMAN and DBTUP blocks. The RESTORE kernel block creates references to the DBLQH and DBTUP blocks to enable quick access to those blocks when needed.

5.5.5. STTOR Phase 2

The only kernel block that participates in this phase to any real effect is NDBCNTR.

In this phase NDBCNTR obtains the current state of each configured cluster data node. Messages are sent to NDBCNTR from QMGR reporting the changes in status of any the nodes. NDBCNTR also sets timers corresponding to the StartPartialTimeout, StartPartitionTimeout, and StartFailureTimeout configuration parameters.

The next step is for a CNTR_START_REQ signal to be sent to the proposed master node. Normally the president is also chosen as master. However, during a system restart where the starting node has a newer global checkpoint than that which has survived on the president, then this node will take over as master node, even though it is not recognized as the president by QMGR. If the starting node is chosen as the new master, then the other nodes are informed of this via a CNTR_START_REF signal.

The master withholds the CNTR_START_REQ signal until it is ready to start a new node, or to start the cluster for an initial restart or system restart.

When the starting node receives CNTR_START_CONF, it starts the NDB_STTOR phases, in the following order:

1. DBLQH
2. DBDICT
3. DBTUP
4. DBACC
5. DBTC
6. DBDIH

5.5.6. NDB_STTOR Phase 1

DBDICT, if necessary, initializes the schema file. DBDIH, DBTC, DBTUP, and DBLQH initialize variables. DBLQH also initializes the sending of statistics on database operations.

5.5.7. STTOR Phase 3

DBDICT initializes a variable that keeps track of the type of restart being performed.

NDBCNTR executes the second of the NDB_STTOR start phases, with no other NDBCNTR activity taking place during this STTOR phase.

5.5.8. NDB_STTOR Phase 2

The DBLQH block enables its exchange of internal records with DBTUP and DBACC, while DBTC allows its internal records to be exchanged with DBDIH. The DBDIH kernel block creates the mutexes used by the NDB kernel and reads nodes using the READ_NODESREQ signal. With the data from the response to this signal, DBDIH can create node lists, node groups, and so forth. For node restarts and initial node restarts, DBDIH also asks the master for permission to perform the restart. The master will ask all “live” nodes if they are prepared to permit the new node to join the cluster. If an initial node restart is to be performed, then all LCPs are invalidated as part of this phase.

LCPs from nodes that are not part of the cluster at the time of the initial node restart are not invalidated. The reason for this is that there is never any chance for a node to become master of a system restart using any of the LCPs that have been invalidated, since this node must complete a node restart — including a local checkpoint — before it can join the cluster and possibly become a master node.

The CMVMI kernel block activates the sending of packed signals, which occurs only as part of database operations. Packing must be enabled prior to beginning any such operations during the execution of the redo log or node recovery phases.

The DBTUX block sets the type of start currently taking place, while the BACKUP block sets the type of restart to be performed, if any (in each case, the block actually sets a variable whose value reflects the type of start or restart). The SUMA block remains inactive during this phase.

The PGMAN kernel block starts the generation of two repeated signals, the first handling cleanup. This signal is sent every 200 milliseconds. The other signal handles statistics, and is sent once per second.

5.5.9. STTOR Phase 4

Only the DBLQH and NDBCNTR kernel blocks are directly involved in this phase. DBLQH allocates a record in the BACKUP block, used in the execution of local checkpoints via the DEFINE_BACKUP_REQ signal. NDBCNTR causes NDB_STTOR to execute NDB_STTOR phase 3; there is otherwise no other NDBCNTR activity during this STTOR phase.

5.5.10. NDB_STTOR Phase 3

The DBLQH block initiates checking of the log files here. Then it obtains the states of the data nodes using the READ_NODESREQ signal. Unless an initial start or an initial node restart is being performed, the checking of log files is handled in parallel with a number of other start phases. For initial starts, the log files must be initialized; this can be a lengthy process and should have some progress status attached to it.

> **Note**
>
> From this point, there are two parallel paths, one continuing restart and another reading and determining the state of the redo log files.

The DBDICT block requests information about the cluster data nodes via the READ_NODESREQ signal. DBACC resets the system restart flag if this is not a system restart; this is used only to verify that no requests are received from DBTUX during system restart. DBTC requests information about all nodes by means of the READ_NODESREQ signal.

DBDIH sets an internal master state and makes other preparations exclusive to initial starts. In the case of an initial start, the non-master nodes perform some initial tasks, the master node doing once all non-master nodes have reported that their tasks are completed. (This delay is actually unnecessary since there is no reason to wait while initializing the master node.)

For node restarts and initial node restarts no more work is done in this phase. For initial starts the work is done when all nodes have created the initial restart information and initialized the system file.

For system restarts this is where most of the work is performed, initiated by sending the NDB_STARTREQ signal from NDBCNTR to DBDIH in the master. This signal is sent when all nodes in the system restart have reached this point in the restart. This we can mark as our first synchronization point for system restarts, designated WAITPOINT_4_1.

For a description of the system restart version of Phase 4, see Section 5.5.21, “System Restart Handling in Phase 4”.

After completing execution of the NDB_STARTREQ signal, the master sends a CNTR_WAITREP signal with WAITPOINT_4_2 to all nodes. This ends NDB_STTOR phase 3 as well as (STTOR) Phase 4.

5.5.11. STTOR Phase 5

All that takes place in Phase 5 is the delivery by NDBCNTR of NDB_STTOR phase 4; the only block that acts on this signal is DBDIH that controls most of the part of a data node start that is database-related.

5.5.12. NDB_STTOR Phase 4

Some initialization of local checkpoint variables takes place in this phase, and for initial restarts, this is all that happens in this phase.

For system restarts, all required takeovers are also performed. Currently, this means that all nodes whose states could not be recovered using the redo log are restarted by copying to them all the necessary data from the “live” data nodes. For node restarts and initial node restarts, the master node performs a number of services, requested to do so by sending the START_MEREQ signal to it. This phase is complete when the master responds with a START_MECONF message, and is described in Section 5.5.22, “START_MEREQ Handling”.

After ensuring that the tasks assigned to DBDIH tasks in the NDB_STTOR phase 4 are complete, NDBCNTR performs some work on its own. For initial starts, it creates the system table that keeps track of unique identifiers such as those used for AUTO_INCREMENT. Following the WAITPOINT_4_1 synchronization point, all system restarts proceed immediately to NDB_STTOR phase 5, which is handled by the DBDIH block. See Section 5.5.13, “NDB_STTOR Phase 5”, for more information.

5.5.13. NDB_STTOR Phase 5

For initial starts and system restarts this phase means executing a local checkpoint. This is handled by the master so that the other nodes will return immediately from this phase. Node restarts and initial node restarts perform the copying of the records from the primary replica to the starting replicas in this phase. Local checkpoints are enabled before the copying process is begun.

Copying the data to a starting node is part of the node takeover protocol. As part of this protocol, the node status of the starting node is updated; this is communicated using the global checkpoint protocol. Waiting for these events to take place ensures that the new node status is communicated to all nodes and their system files.

After the node's status has been communicated, all nodes are signaled that we are about to start the takeover protocol for this node.

Part of this protocol consists of Steps 3 - 9 during the system restart phase as described below. This means that restoration of all the fragments, preparation for execution of the redo log, execution of the redo log, and finally reporting back to DBDIH when the execution of the redo log is completed, are all part of this process.

After preparations are complete, copy phase for each fragment in the node must be performed. The process of copying a fragment involves the following steps:

1. The DBLQH kernel block in the starting node is informed that the copy process is about to begin by sending it a PREPARE_COPY_FRAGREQ signal.
2. When DBLQH acknowledges this request a CREATE_FRAGREQ signal is sent to all nodes notify them of the preparation being made to copy data to this replica for this table fragment.
3. After all nodes have acknowledged this, a COPY_FRAGREQ signal is sent to the node from which the data is to be copied to the new node. This is always the primary replica of the fragment. The node indicated copies all the data over to the starting node in response to this message.
4. After copying has been completed, and a COPY_FRAGCONF message is sent, all nodes are notified of the completion through an UPDATE_TOREQ signal.
5. After all nodes have updated to reflect the new state of the fragment, the DBLQH kernel block of the starting node is informed of the fact that the copy has been completed, and that the replica is now up-to-date and any failures should now be treated as real failures.
6. The new replica is transformed into a primary replica if this is the role it had when the table was created.
7. After completing this change another round of CREATE_FRAGREQ messages is sent to all nodes informing them that the takeover of the fragment is now committed.
8. After this, process is repeated with the next fragment if another one exists.
9. When there are no more fragments for takeover by the node, all nodes are informed of this by sending an UPDATE_TOREQ signal sent to all of them.
10. Wait for the next complete local checkpoint to occur, running from start to finish.
11. The node states are updated, using a complete global checkpoint. As with the local checkpoint in the previous step, the global checkpoint must be allowed to start and then to finish.
12. When the global checkpoint has completed, it will communicate the successful local checkpoint of this node restart by sending an END_TOREQ signal to all nodes.
13. A START_COPYCONF is sent back to the starting node informing it that the node restart has been completed.
14. Receiving the START_COPYCONF signal ends NDB_STTOR phase 5. This provides another synchronization point for system restarts, designated as WAITPOINT_5_2.

> **Note**
>
> The copy process in this phase can in theory be performed in parallel by several nodes. However, all messages from the master to all nodes are currently sent to single node at a time, but can be made completely parallel. This is likely to be done in the not too distant future.

In an initial and an initial node restart, the SUMA block requests the subscriptions from the SUMA master node. NDBCNTR executes NDB_STTOR phase 6. No other NDBCNTR activity takes place.

5.5.14. NDB_STTOR Phase 6

In this NDB_STTOR phase, both DBLQH and DBDICT clear their internal representing the current restart type. The DBACC block resets the system restart flag; DBACC and DBTUP start a periodic signal for checking memory usage once per second. DBTC sets an internal variable indicating that the system restart has been completed.

5.5.15. STTOR Phase 6

The NDBCNTR block defines the cluster's node groups, and the DBUTIL block initializes a number of data structures to facilitate the sending keyed operations can be to the system tables. DBUTIL also sets up a single connection to the DBTC kernel block.

5.5.16. STTOR Phase 7

In QMGR the president starts an arbitrator (unless this feature has been disabled by setting the value of the ArbitrationRank configuration parameter to 0 for all nodes — see Defining a MySQL Cluster Management Server, and Defining SQL and Other API Nodes in a MySQL Cluster, for more information; note that this currently can be done only when using MySQL Cluster Carrier Grade Edition). In addition, checking of API nodes through heartbeats is activated.

Also during this phase, the BACKUP block sets the disk write speed to the value used following the completion of the restart. The master node during initial start also inserts the record keeping track of which backup ID is to be used next. The SUMA and DBTUX blocks set variables indicating start phase 7 has been completed, and that requests to DBTUX that occurs when running the redo log should no longer be ignored.

5.5.17. STTOR Phase 8

NDB_STTOR executes NDB_STTOR phase 7; no other NDBCNTR activity takes place.

5.5.18. NDB_STTOR Phase 7

If this is a system restart, the master node initiates a rebuild of all indexes from DBDICT during this phase.

The CMVMI kernel block opens communication channels to the API nodes (including MySQL servers acting as SQL nodes). Indicate in globalData that the node is started.

5.5.19. STTOR Phase 9

NDBCNTR resets some start variables.

5.5.20. STTOR Phase 101

This is the SUMA handover phase, during which a GCP is negotiated and used as a point of reference for changing the source of event and replication subscriptions from existing nodes only to include a newly started node.

5.5.21. System Restart Handling in Phase 4

This consists of the following steps:

1. The master sets the latest GCI as the restart GCI, and then synchronizes its system file to all other nodes involved in the system restart.

2. The next step is to synchronize the schema of all the nodes in the system restart. This is performed in 15 passes. The problem we are trying to solve here occurs when a schema object has been created while the node was up but was dropped while the node was down, and possibly a new object was even created with the same schema ID while that node was unavailable. In order to handle this situation, it is necessary first to re-create all objects that are supposed to exist from the viewpoint of the starting node. After this, any objects that were dropped by other nodes in the cluster while this node was “dead” are dropped; this also applies to any tables that were dropped during the outage. Finally, any tables that have been created by other nodes while the starting node was unavailable are re-created on the starting node. All these operations are local to the starting node. As part of this process, is it also necessary to ensure that all tables that need to be re-created have been created locally and that the proper data structures have been set up for them in all kernel blocks.

   After performing the procedure described previously for the master node the new schema file is sent to all other participants in the system restart, and they perform the same synchronization.

3. All fragments involved in the restart must have proper parameters as derived from DBDIH. This causes a number of START_FRAGREQ signals to be sent from DBDIH to DBLQH. This also starts the restoration of the fragments, which are restored one by one and one record at a time in the course of reading the restore data from disk and applying in parallel the restore data read from disk into main memory. This restores only the main memory parts of the tables.

4. Once all fragments have been restored, a START_RECREQ message is sent to all nodes in the starting cluster, and then all undo logs for any Disk Data parts of the tables are applied.

5. After applying the undo logs in LGMAN, it is necessary to perform some restore work in TSMAN that requires scanning the extent headers of the tablespaces.

6. Next, it is necessary to prepare for execution of the redo log, which log can be performed in up to four phases. For each fragment, execution of redo logs from several different nodes may be required. This is handled by executing the redo logs in different phases for a specific fragment, as decided in DBDIH when sending the START_FRAGREQ signal. An EXEC_FRAGREQ signal is sent for each phase and fragment that requires execution in this phase. After these signals are sent, an EXEC_SRREQ

signal is sent to all nodes to tell them that they can start executing the redo log.

> **Note**
>
> Before starting execution of the first redo log, it is necessary to make sure that the setup which was started earlier (in Phase 4) by DBLQH has finished, or to wait until it does before continuing.

7. Prior to executing the redo log, it is necessary to calculate where to start reading and where the end of the REDO log should have been reached. The end of the REDO log should be found when the last GCI to restore has been reached.

8. After completing the execution of the redo logs, all redo log pages that have been written beyond the last GCI to be restore are invalidated. Given the cyclic nature of the redo logs, this could carry the invalidation into new redo log files past the last one executed.

9. After the completion of the previous step, DBLQH report this back to DBDIH using a START_RECCONF message.

10. When the master has received this message back from all starting nodes, it sends a NDB_STARTCONF signal back to NDBCNTR.

11. The NDB_STARTCONF message signals the end of STTOR phase 4 to NDBCNTR, which is the only block involved to any significant degree in this phase.

5.5.22. START_MEREQ Handling

The first step in handling START_MEREQ is to ensure that no local checkpoint is currently taking place; otherwise, it is necessary to wait until it is completed. The next step is to copy all distribution information from the master DBDIH to the starting DBDIH. After this, all metadata is synchronized in DBDICT (see Section 5.5.21, “System Restart Handling in Phase 4”).

After blocking local checkpoints, and then synchronizing distribution information and metadata information, global checkpoints are blocked.

The next step is to integrate the starting node in the global checkpoint protocol, local checkpoint protocol, and all other distributed protocols. As part of this the node status is also updated.

After completing this step the global checkpoint protocol is permitted to start again, the START_MECONF signal is sent to indicate to the starting node that the next phase may proceed.

5.6. NDB Internals Glossary

This section contains terms and abbreviations that are found in or useful to understanding the NDB source code.

- **ACC.** **ACC**elerator or **ACC**ess manager. Handles hash indexes of primary keys, providing fast access to records. See Section 5.4.3, “The DBACC Block”.

- **API node.** In NDB terms, this is any application that accesses cluster data using the NDB API, including mysqld when functioning as an API node. (MySQL servers acting in this capacity are also referred to as “SQL nodes”). Often abbreviated to “API”.

- **CMVMI.** Stands for **C**luster **M**anager **V**irtual **M**achine **I**nterface. An NDB kernel handling non-signal requests to the operating system, as well as configuration management, interaction with the cluster management server, and interaction between various kernel blocks and the NDB virtual machine. See Section 5.4.2, “The CMVMI Block”, for more information.

- **CNTR.** Stands for restart **C**oordi**N**a**T**o**R**. See Section 5.4.12, “The NDBCNTR Block”, for more information.

- **DBTC.** The transaction coordinator (also sometimes written simply as **TC**). See Section 5.4.7, “The DBTC Block”, for more information.

- **DICT.** The NDB data **DICT**ionary kernel block. Also **DBDICT**. See Section 5.4.4, “The DBDICT Block”.

- **DIH.** **DI**stribution **H**andler. An NDB kernel block. See Section 5.4.5, “The DBDIH Block”.

- **LGMAN.** The **L**og **G**roup **MAN**ager NDB kernel block, used for MySQL Cluster Disk Data tables. See Section 5.4.11, “The LGMAN Block”.

- **LQH.** **L**ocal **Q**uery **H**andler. NDB kernel block, discussed in Section 5.4.6, “DBLQH Block”.

- **MGM.** **M**ana**G**e**M**ent node (or management server). Implemented as the ndb_mgmd server daemon. Responsible for passing cluster configuration information to data nodes and performing functions such as starting and stopping nodes. Accessed by the

user by means of the cluster management client (ndb_mgm). A discussion of management nodes can be found in ndb_mgmd.

- **QMGR.** The cluster management block in the NDB kernel. It responsibilities include monitoring heartbeats from data and API nodes. See Section 5.4.15, "The QMGR Block", for more information.
- **RBR.** **R**ow-**B**ased **R**eplication. MySQL Cluster Replication is row-based replication. See MySQL Cluster Replication.
- **STTOR.** **ST**art **O**r **R**estart
- **SUMA.** The cluster **SU**bscription **MA**nager. See Section 5.4.17, "The SUMA Block".
- **TC**

  TC. **T**ransaction **C**oordinator. See Section 5.4.7, "The DBTC Block".
- **TRIX.** Stands for **TR**ansactions and **I**nde**X**es, which are managed by the NDB kernel block having this name. See Section 5.4.19, "The TRIX Block".
- **TSMAN.** **T**able **s**pace **m**anager. Handles tablespaces for MySQL Cluster Disk Data. See Section 5.4.18, "The TSMAN Block", for more information.
- **TUP.** **TUP**le. Unit of data storage. Also used (along with **DBTUP**) to refer to the NDB kernel's tuple management block, which is discussed in Section 5.4.8, "The DBTUP Block".

ndex

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W