MySQL Cluster

MySQL Cluster

Abstract

This is the MySQL Cluster extract from the MySQL 5.1 Reference Manual.

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

MySQL Cluster NDB 6.X/7.X

This chapter contains information about *MySQL Cluster*, which is a high-availability, high-redundancy version of MySQL adapted for the distributed computing environment. Current releases of MySQL Cluster use versions 6 and 7 of the NDBCLUSTER storage engine (also known as NDB) to enable running several computers with MySQL servers and other software in a cluster.

Beginning with MySQL 5.1.24, support for the NDBCLUSTER storage engine was removed from the standard MySQL server binaries built by MySQL. Instead, users of MySQL Cluster binaries built by MySQL should upgrade to the most recent binary release of MySQL Cluster NDB 6.2 or MySQL Cluster 6.3 for supported platforms — these include RPMs that should work with most Linux distributions. MySQL Cluster users who build from source should be aware that, also beginning with MySQL 5.1.24, NDBCLUSTER sources in the standard MySQL 5.1 tree are no longer maintained; these users should use the sources provided for MySQL Cluster NDB 6.2 or later. (Locations where the sources can be obtained are listed later in this section.)

> **Note**
>
> MySQL Cluster NDB 6.1, 6.2, and 6.3 were formerly known as "MySQL Cluster Carrier Grade Edition". Beginning with MySQL Cluster NDB 6.2.15 and MySQL Cluster NDB 6.3.14, this term is no longer applied to the MySQL Cluster software — which is now known simply as "MySQL Cluster" — but rather to a commercial licensing and support package. You can learn more about available options for commercial licensing of MySQL Cluster from *MySQL Cluster Features*, on the MySQL web site.

This chapter contains information about MySQL Cluster in MySQL 5.1 mainline releases through MySQL 5.1.23, MySQL Cluster NDB 6.2 releases through 5.1.34-ndb-6.2.18, MySQL Cluster NDB 6.3 releases through 5.1.34-ndb-6.3.26, and MySQL Cluster NDB 7.0 releases through 5.1.34-ndb-7.0.7. Currently, the MySQL Cluster NDB 6.2, MySQL Cluster NDB 6.3, and MySQL Cluster NDB 7.0 (formerly known as "MySQL Cluster NDB 6.4") release series are Generally Available (GA).

This chapter also contains historical information about MySQL Cluster NDB 6.1, although this release series is no longer in active development, and should not be used in new deployments. Users of MySQL Cluster NDB 6.1 should upgrade to a later MySQL Cluster NDB 6.x or 7.x release series as soon as possible.

Platforms supported. MySQL Cluster is currently available and supported on a number of platforms, including Linux, Solaris, Mac OS X, HP-UX, and other Unix-style operating systems on a variety of hardware. Beginning with MySQL Cluster NDB 7.0, MySQL Cluster is also available (on an experimental basis) on Microsoft Windows platforms. For exact levels of support available for on specific combinations of operating system versions, operating system distributions, and hardware platforms, please refer to the *Cluster Supported Platforms list*, maintained by the MySQL Support Team on the MySQL web site.

We are continuing to work to make MySQL Cluster available on all operating systems supported by MySQL and will update the information provided here as this work continues.

Availability. MySQL Cluster NDB 6.2, MySQL Cluster NDB 6.3, and MySQL Cluster NDB 7.0 binary and source packages are available for supported platforms from http://dev.mysql.com/downloads/cluster.

> **Note**
>
> Binary releases and RPMs were not available for MySQL Cluster NDB 6.2 prior to MySQL Cluster NDB 6.2.15.

MySQL Cluster release numbers. Starting with MySQL Cluster NDB 6.1 and 6.2, MySQL Cluster follows a somewhat different release pattern from the mainline MySQL 5.1 Cluster series of releases. In this Manual and other MySQL documentation, we identify these and later MySQL Cluster releases employing a version number that begins with "NDB". This version number is that of the NDBCLUSTER storage engine used, and not of the MySQL server version on which the MySQL Cluster release is based.

Version strings used in MySQL Cluster NDB 6.x and 7.x software. The version string displayed by MySQL Cluster NDB 6.x and 7.x software uses this format:

```
mysql-mysql_server_version-ndb-ndbcluster_engine_version
```

mysql_server_version represents the version of the MySQL Server on which the MySQL Cluster release is based. For all MySQL Cluster NDB 6.x and 7.x releases, this is "5.1". *ndbcluster_engine_version* is the version of the NDBCLUSTER storage engine used by this release of the MySQL Cluster software. You can see this format used in the mysql client, as shown here:

```
shell> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.1.34-ndb-7.0.7 Source distribution
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> SELECT VERSION()\G
*************************** 1. row ***************************
VERSION(): 5.1.34-ndb-7.0.7
1 row in set (0.00 sec)
```

This version string is also displayed in the output of the SHOW command in the ndb_mgm client:

```
ndb_mgm> SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @10.0.10.6  (5.1.34-ndb-7.0.7, Nodegroup: 0, Master)
id=2    @10.0.10.8  (5.1.34-ndb-7.0.7, Nodegroup: 0)
[ndb_mgmd(MGM)] 1 node(s)
id=3    @10.0.10.2  (5.1.34-ndb-7.0.7)
[mysqld(API)]   2 node(s)
id=4    @10.0.10.10  (5.1.34-ndb-7.0.7)
id=5 (not connected, accepting connect from any host)
```

The version string identifies the mainline MySQL version from which the MySQL Cluster release was branched and the version of the `NDBCLUSTER` storage engine used. For example, the full version string for MySQL Cluster NDB 6.2.15 (the first MySQL Cluster NDB 6.2 binary release) was `mysql-5.1.24-ndb-6.2.15`. From this we can determine the following:

- Since the portion of the version string preceding "`-ndb-`" is the base MySQL Server version, this means that MySQL Cluster NDB 6.2.15 derives from the MySQL 5.1.24, and contains all feature enhancements and bugfixes from MySQL 5.1 up to and including MySQL 5.1.24.
- Since the portion of the version string following "`-ndb-`" represents the version number of the `NDB` (or `NDBCLUSTER`) storage engine, MySQL Cluster NDB 6.2.15 uses version 6.2.15 of the `NDBCLUSTER` storage engine.

MySQL Cluster development source trees. MySQL Cluster development trees can also be accessed via https://code.launchpad.net/~mysql/:

- `MySQL Cluster NDB 6.1` (*OBSOLETE — no longer maintained*)
- `MySQL Cluster NDB 6.2` (*CURRENT*)
- `MySQL Cluster NDB 6.3` (*CURRENT*)
- `MySQL Cluster NDB 7.0` (*CURRENT*)

The MySQL Cluster development sources maintained at https://code.launchpad.net/~mysql/ are licensed under the GPL. For information about obtaining MySQL sources using Bazaar and building them yourself, see Installing from the Development Source Tree.

Currently, MySQL Cluster NDB 6.2, MySQL Cluster NDB 6.3, and MySQL Cluster NDB 7.0 releases are all Generally Available (GA). MySQL Cluster NDB 6.1 is no longer in active development. For an overview of major features added in MySQL Cluster NDB 6.x and 7.x, see Chapter 13, *MySQL Cluster Development Roadmap*.

This chapter represents a work in progress, and its contents are subject to revision as MySQL Cluster continues to evolve. Additional information regarding MySQL Cluster can be found on the MySQL AB Web site at http://www.mysql.com/products/cluster/.

Additional resources. More information may be found in the following places:

- Answers to some commonly asked questions about Cluster may be found in the MySQL 5.1 FAQ — MySQL Cluster.
- The MySQL Cluster mailing list: http://lists.mysql.com/cluster.
- The MySQL Cluster Forum: http://forums.mysql.com/list.php?25.
- Many MySQL Cluster users and some of the MySQL Cluster developers blog about their experiences with Cluster, and make feeds of these available through PlanetMySQL.
- If you are new to MySQL Cluster, you may find our Developer Zone article *How to set up a MySQL Cluster for two servers* to be helpful.

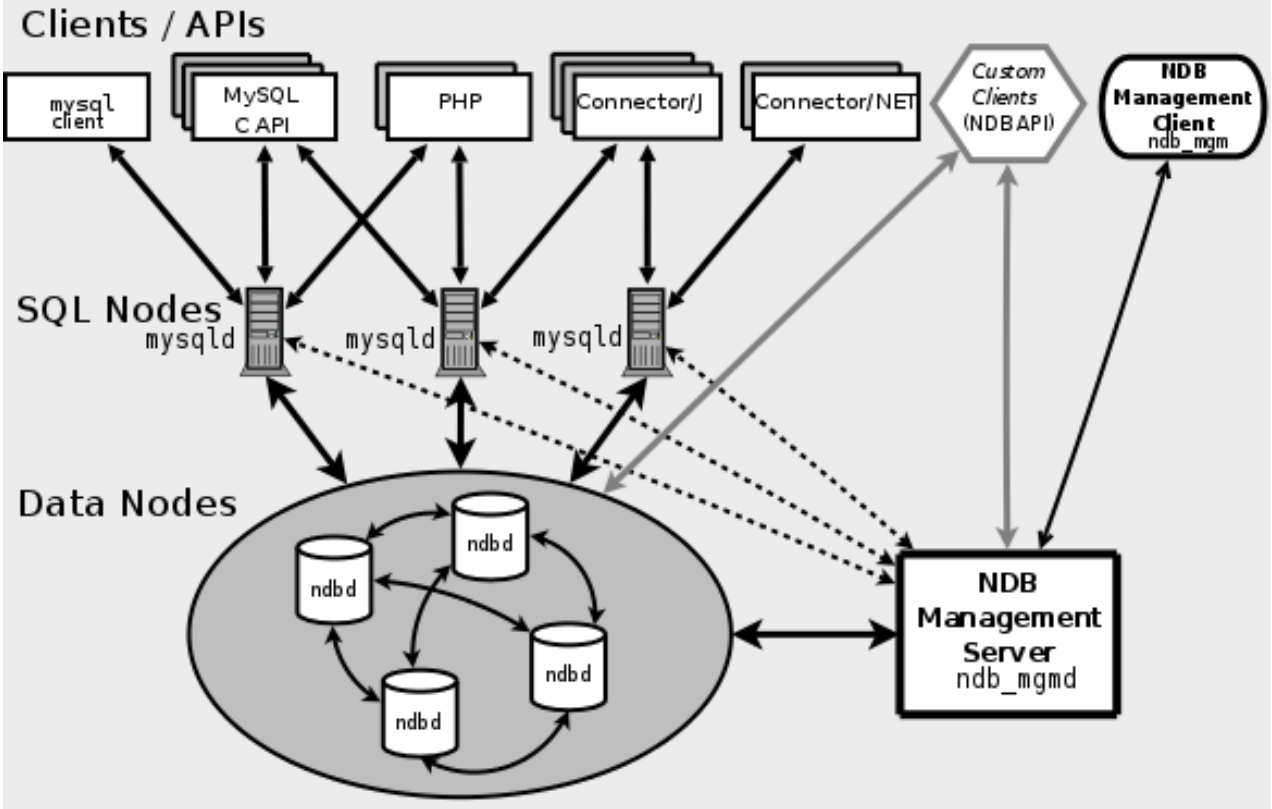
Chapter 1. MySQL Cluster Overview

MySQL Cluster is a technology that enables clustering of in-memory databases in a shared-nothing system. The shared-nothing architecture allows the system to work with very inexpensive hardware, and with a minimum of specific requirements for hardware or software.

MySQL Cluster is designed not to have any single point of failure. For this reason, each component is expected to have its own memory and disk, and the use of shared storage mechanisms such as network shares, network file systems, and SANs is not recommended or supported.

MySQL Cluster integrates the standard MySQL server with an in-memory clustered storage engine called NDB. In our documentation, the term NDB refers to the part of the setup that is specific to the storage engine, whereas “MySQL Cluster” refers to the combination of MySQL and the NDB storage engine.

A MySQL Cluster consists of a set of computers, each running one or more processes which may include a MySQL server, a data node, a management server, and (possibly) specialized data access programs. The relationship of these components in a cluster is shown here:



All these programs work together to form a MySQL Cluster. When data is stored in the NDBCLUSTER storage engine, the tables are stored in the data nodes. Such tables are directly accessible from all other MySQL servers in the cluster. Thus, in a payroll application storing data in a cluster, if one application updates the salary of an employee, all other MySQL servers that query this data can see this change immediately.

The data stored in the data nodes for MySQL Cluster can be mirrored; the cluster can handle failures of individual data nodes with no other impact than that a small number of transactions are aborted due to losing the transaction state. Because transactional applications are expected to handle transaction failure, this should not be a source of problems.

1.1. MySQL Cluster Core Concepts

NDBCLUSTER (also known as NDB) is an in-memory storage engine offering high-availability and data-persistence features.

The NDBCLUSTER storage engine can be configured with a range of failover and load-balancing options, but it is easiest to start with the storage engine at the cluster level. MySQL Cluster's NDB storage engine contains a complete set of data, dependent only on other data within the cluster itself.

The “Cluster” portion of MySQL Cluster is configured independently of the MySQL servers. In a MySQL Cluster, each part of the cluster is considered to be a *node*.

> **Note**
>
> In many contexts, the term “node” is used to indicate a computer, but when discussing MySQL Cluster it means a *process*. It is possible to run multiple nodes on a single computer; for a computer on which one or more cluster nodes are being run we use the term *cluster host*.

There are three types of cluster nodes, and in a minimal MySQL Cluster configuration, there will be at least three nodes, one of each of these types:

- *Management node* (MGM node): The role of this type of node is to manage the other nodes within the MySQL Cluster, performing such functions as providing configuration data, starting and stopping nodes, running backup, and so forth. Because this node type manages the configuration of the other nodes, a node of this type should be started first, before any other node. An MGM node is started with the command `ndb_mgmd`.

- *Data node*: This type of node stores cluster data. There are as many data nodes as there are replicas, times the number of fragments. For example, with two replicas, each having two fragments, you need four data nodes. One replica is sufficient for data storage, but provides no redundancy; therefore, it is recommended to have 2 (or more) replicas to provide redundancy, and thus high availability. A data node is started with the command `ndbd`.

- *SQL node*: This is a node that accesses the cluster data. In the case of MySQL Cluster, an SQL node is a traditional MySQL server that uses the `NDBCLUSTER` storage engine. An SQL node is a `mysqld` process started with the `--ndbcluster` and `--ndb-connectstring` options, which are explained elsewhere in this chapter, possibly with additional MySQL server options as well.

  An SQL node is actually just a specialized type of *API node*, which designates any application which accesses Cluster data. Another example of an API node is the `ndb_restore` utility that is used to restore a cluster backup. It is possible to write such applications using the NDB API. For basic information about the NDB API, see Getting Started with the NDB API.

> **Important**
>
> It is not realistic to expect to employ a three-node setup in a production environment. Such a configuration provides no redundancy; in order to benefit from MySQL Cluster's high-availability features, you must use multiple data and SQL nodes. The use of multiple management nodes is also highly recommended.

For a brief introduction to the relationships between nodes, node groups, replicas, and partitions in MySQL Cluster, see Section 1.2, “MySQL Cluster Nodes, Node Groups, Replicas, and Partitions”.

Configuration of a cluster involves configuring each individual node in the cluster and setting up individual communication links between nodes. MySQL Cluster is currently designed with the intention that data nodes are homogeneous in terms of processor power, memory space, and bandwidth. In addition, to provide a single point of configuration, all configuration data for the cluster as a whole is located in one configuration file.

The management server (MGM node) manages the cluster configuration file and the cluster log. Each node in the cluster retrieves the configuration data from the management server, and so requires a way to determine where the management server resides. When interesting events occur in the data nodes, the nodes transfer information about these events to the management server, which then writes the information to the cluster log.

In addition, there can be any number of cluster client processes or applications. These are of two types:

- **Standard MySQL clients.** MySQL Cluster can be used with existing MySQL applications written in PHP, Perl, C, C++, Java, Python, Ruby, and so on. Such client applications send SQL statements to and receive responses from MySQL servers acting as MySQL Cluster SQL nodes in much the same way that they interact with standalone MySQL servers. However, MySQL clients using a MySQL Cluster as a data source can be modified to take advantage of the ability to connect with multiple MySQL servers to achieve load balancing and failover. For example, Java clients using Connector/J 5.0.6 and later can use `jdbc:mysql:loadbalance://` URLs (improved in Connector/J 5.1.7) to achieve load balancing transparently .

- **Management clients.** These clients connect to the management server and provide commands for starting and stopping nodes gracefully, starting and stopping message tracing (debug versions only), showing node versions and status, starting and stopping backups, and so on. Such clients — such as the `ndb_mgm` management client supplied with MySQL Cluster — are written using the MGM API, a C-language API that communicates directly with one or more MySQL Cluster management servers. For more information, see The MGM API.

1.2. MySQL Cluster Nodes, Node Groups, Replicas, and Partitions

This section discusses the manner in which MySQL Cluster divides and duplicates data for storage.

Central to an understanding of this topic are the following concepts, listed here with brief definitions:

- **(Data) Node.** An `ndbd` process, which stores a *replica* —that is, a copy of the *partition* (see below) assigned to the node group of which the node is a member.

  Each data node should be located on a separate computer. While it is also possible to host multiple `ndbd` processes on a single computer, such a configuration is not supported.

  It is common for the terms "node" and "data node" to be used interchangeably when referring to an `ndbd` process; where mentioned, management (MGM) nodes (`ndb_mgmd` processes) and SQL nodes (`mysqld` processes) are specified as such in this discussion.

- **Node Group.** A node group consists of one or more nodes, and stores partitions, or sets of *replicas* (see next item).

  The number of node groups in a MySQL Cluster is not directly configurable; it is function of the number of data nodes and of the number of replicas (`NumberOfReplicas` configuration parameter), as shown here:

  ```
  [number_of_node_groups] = number_of_data_nodes / NumberOfReplicas
  ```

  Thus, a MySQL Cluster with 4 data nodes has 4 node groups if `NumberOfReplicas` is set to 1 in the `config.ini` file, 2 node groups if `NumberOfReplicas` is set to 2, and 1 node group if `NumberOfReplicas` is set to 4. Replicas are discussed later in this section; for more information about `NumberOfReplicas`, see Section 3.4.6, "Defining MySQL Cluster Data Nodes".

  > **Note**
  >
  > All node groups in a MySQL Cluster must have the same number of data nodes.

  Prior to MySQL Cluster NDB 7.0, it was not possible to add new data nodes to a MySQL Cluster without shutting down the cluster completely and reloading all of its data. In MySQL Cluster NDB 7.0 (beginning with MySQL Cluster version NDB 6.4.0), you can add new node groups (and thus new data nodes) to a running MySQL Cluster — see Section 7.8, "Adding MySQL Cluster Data Nodes Online", for information about how this can be done.

- **Partition.** This is a portion of the data stored by the cluster. There are as many cluster partitions as nodes participating in the cluster. Each node is responsible for keeping at least one copy of any partitions assigned to it (that is, at least one replica) available to the cluster.

  A replica belongs entirely to a single node; a node can (and usually does) store several replicas.

  MySQL Cluster normally partitions `NDBCLUSTER` tables automatically. However, in MySQL 5.1 and MySQL Cluster NDB 6.x, it is possible to employ user-defined partitioning with `NDBCLUSTER` tables. This is subject to the following limitations:

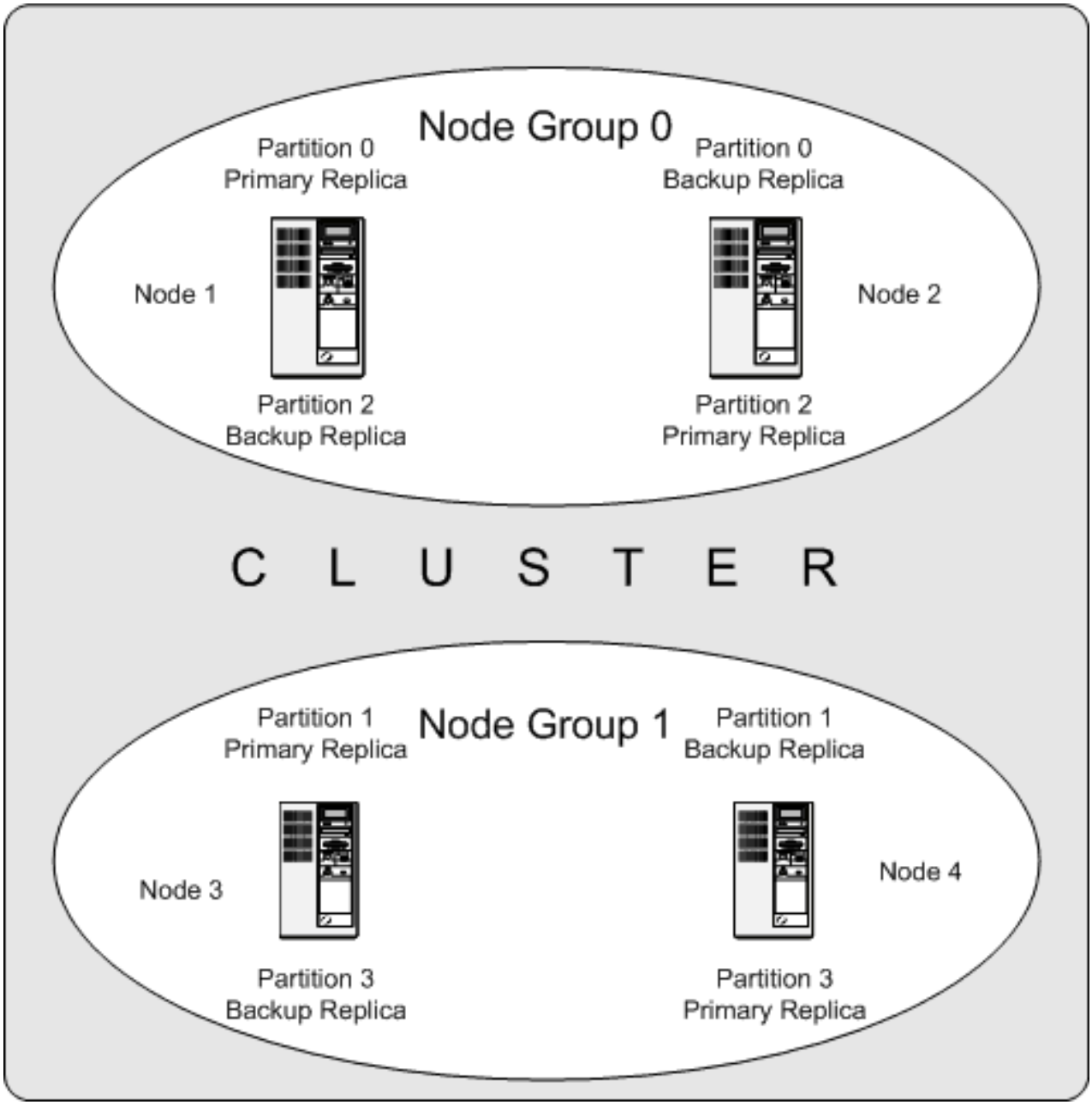
  1. Only `KEY` and `LINEAR KEY` partitioning schemes can be used with `NDBCLUSTER` tables.

  2. The maximum number of partitions that may be definied explicitly for any `NDBCLUSTER` table is 8 per node group. (The number of node groups in a MySQL Cluster is determined as discussed previously in this section.)

  For more information relating to MySQL Cluster and user-defined partitioning, see Chapter 12, *Known Limitations of MySQL Cluster*, and Partitioning Limitations Relating to Storage Engines.

- **Replica.** This is a copy of a cluster partition. Each node in a node group stores a replica. Also sometimes known as a *partition replica*. The number of replicas is equal to the number of nodes per node group.

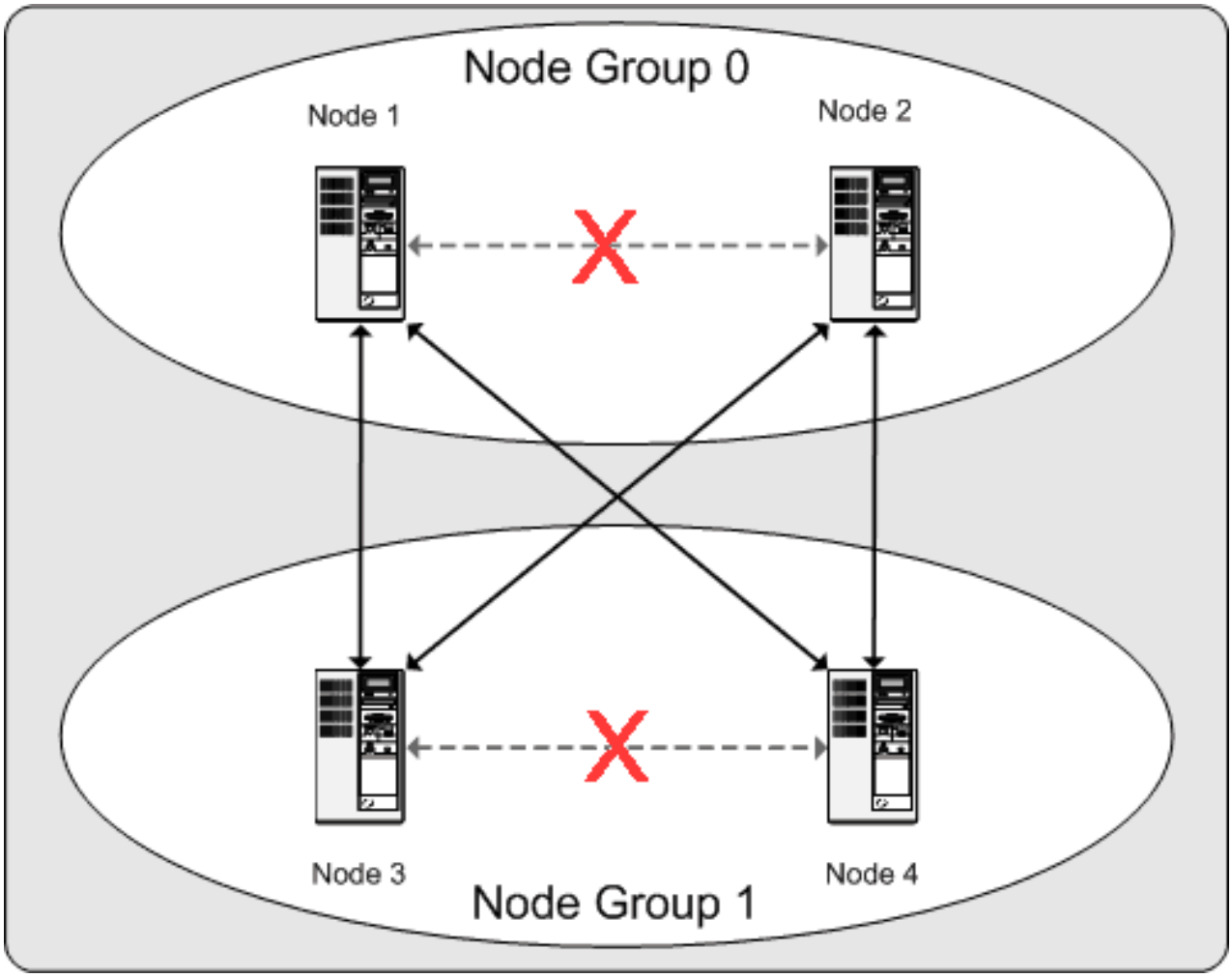
The following diagram illustrates a MySQL Cluster with four data nodes, arranged in two node groups of two nodes each; nodes 1 and 2 belong to node group 0, and nodes 3 and 4 belong to node group 1. Note that only data (`ndbd`) nodes are shown here; although a working cluster requires an `ndb_mgm` process for cluster management and at least one SQL node to access the data stored by the cluster, these have been omitted in the figure for clarity.



The data stored by the cluster is divided into four partitions, numbered 0, 1, 2, and 3. Each partition is stored — in multiple copies — on the same node group. Partitions are stored on alternate node groups:

- Partition 0 is stored on node group 0; a *primary replica* (primary copy) is stored on node 1, and a *backup replica* (backup copy of the partition) is stored on node 2.
- Partition 1 is stored on the other node group (node group 1); this partition's primary replica is on node 3, and its backup replica is on node 4.
- Partition 2 is stored on node group 0. However, the placing of its two replicas is reversed from that of Partition 0; for Partition 2, the primary replica is stored on node 2, and the backup on node 1.
- Partition 3 is stored on node group 1, and the placement of its two replicas are reversed from those of partition 1. That is, its primary replica is located on node 4, with the backup on node 3.

What this means regarding the continued operation of a MySQL Cluster is this: so long as each node group participating in the cluster has at least one node operating, the cluster has a complete copy of all data and remains viable. This is illustrated in the next diagram.



In this example, where the cluster consists of two node groups of two nodes each, any combination of at least one node in node group 0 and at least one node in node group 1 is sufficient to keep the cluster “alive” (indicated by arrows in the diagram). However, if *both* nodes from *either* node group fail, the remaining two nodes are not sufficient (shown by the arrows marked out with an **X**); in either case, the cluster has lost an entire partition and so can no longer provide access to a complete set of all cluster data.

Chapter 2. MySQL Cluster Multi-Computer How-To

This section is a “How-To” that describes the basics for how to plan, install, configure, and run a MySQL Cluster. Whereas the examples in Chapter 3, *MySQL Cluster Configuration* provide more in-depth information on a variety of clustering options and configuration, the result of following the guidelines and procedures outlined here should be a usable MySQL Cluster which meets the *minimum* requirements for availability and safeguarding of data.

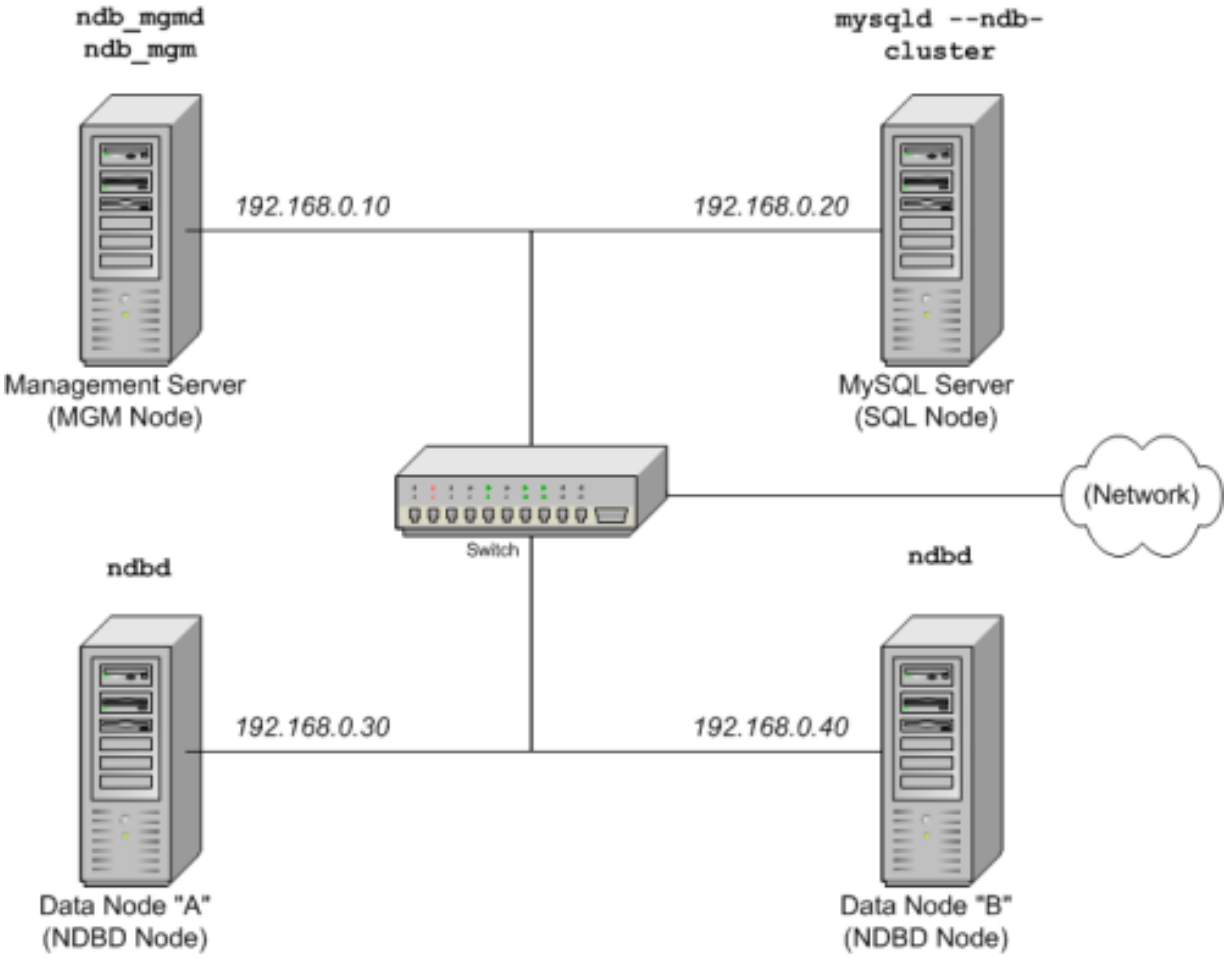
This section covers hardware and software requirements; networking issues; installation of MySQL Cluster; configuration issues; starting, stopping, and restarting the cluster; loading of a sample database; and performing queries.

Basic assumptions. This *How-To* makes the following assumptions:

1. The cluster is to be set up with four nodes, each on a separate host, and each with a fixed network address on a typical Ethernet network as shown here:

Node	IP Address
Management (MGMD) node	192.168.0.10
MySQL server (SQL) node	192.168.0.20
Data (NDBD) node "A"	192.168.0.30
Data (NDBD) node "B"	192.168.0.40

This may be made clearer in the following diagram:



In the interest of simplicity (and reliability), this *How-To* uses only numeric IP addresses. However, if DNS resolution is available on your network, it is possible to use host names in lieu of IP addresses in configuring Cluster. Alternatively, you can use the `/etc/hosts` file or your operating system's equivalent for providing a means to do host lookup if such is available.

Note

A common problem when trying to use host names for Cluster nodes arises because of the way in which some operating systems (including some Linux distributions) set up the system's own host name in the `/etc/hosts` during installation. Consider two machines with the host names `ndb1` and `ndb2`, both in the `cluster` network domain. Red Hat Linux (including some derivatives such as CentOS and Fedora) places the following entries in these machines' `/etc/hosts` files:

```
#  ndb1 /etc/hosts:
127.0.0.1   ndb1.cluster ndb1 localhost.localdomain localhost
```

```
#  ndb2 /etc/hosts:
127.0.0.1   ndb2.cluster ndb2 localhost.localdomain localhost
```

SUSE Linux (including OpenSUSE) places these entries in the machines' `/etc/hosts` files:

```
#  ndb1 /etc/hosts:
127.0.0.1       localhost
127.0.0.2       ndb1.cluster ndb1
```

```
#  ndb2 /etc/hosts:
127.0.0.1       localhost
127.0.0.2       ndb2.cluster ndb2
```

In both instances, `ndb1` routes `ndb1.cluster` to a loopback IP address, but gets a public IP address from DNS for `ndb2.cluster`, while `ndb2` routes `ndb2.cluster` to a loopback address and obtains a public address for `ndb1.cluster`. The result is that each data node connects to the management server, but cannot tell when any other data nodes have connected, and so the data nodes appear to hang while starting.

You should also be aware that you cannot mix `localhost` and other host names or IP addresses in `config.ini`. For these reasons, the solution in such cases (other than to use IP addresses for *all* `config.ini` `HostName` entries) is to remove the fully qualified host names from `/etc/hosts` and use these in `config.ini` for all cluster hosts.

2. Each host in our scenario is an Intel-based desktop PC running a common, generic Linux distribution installed to disk in a standard configuration, and running no unnecessary services. The core OS with standard TCP/IP networking capabilities should be sufficient. Also for the sake of simplicity, we also assume that the file systems on all hosts are set up identically. In the event that they are not, you will need to adapt these instructions accordingly.

3. Standard 100 Mbps or 1 gigabit Ethernet cards are installed on each machine, along with the proper drivers for the cards, and that all four hosts are connected via a standard-issue Ethernet networking appliance such as a switch. (All machines should use network cards with the same throughout. That is, all four machines in the cluster should have 100 Mbps cards *or* all four machines should have 1 Gbps cards.) MySQL Cluster will work in a 100 Mbps network; however, gigabit Ethernet will provide better performance.

   Note that MySQL Cluster is *not* intended for use in a network for which throughput is less than 100 Mbps. For this reason (among others), attempting to run a MySQL Cluster over a public network such as the Internet is not likely to be successful, and is not recommended.

4. For our sample data, we will use the `world` database which is available for download from the MySQL AB Web site. As this database takes up a relatively small amount of space, we assume that each machine has 256MB RAM, which should be sufficient for running the operating system, host NDB process, and (for the data nodes) for storing the database.

Although we refer to a Linux operating system in this How-To, the instructions and procedures that we provide here should be easily adaptable to other supported operating systems. We also assume that you already know how to perform a minimal installation and configuration of the operating system with networking capability, or that you are able to obtain assistance in this elsewhere if needed.

We discuss MySQL Cluster hardware, software, and networking requirements in somewhat greater detail in the next section. (See Section 2.1, “MySQL Cluster Hardware, Software, and Networking Requirements”.)

2.1. MySQL Cluster Hardware, Software, and Networking Requirements

One of the strengths of MySQL Cluster is that it can be run on commodity hardware and has no unusual requirements in this regard, other than for large amounts of RAM, due to the fact that all live data storage is done in memory. (It is possible to reduce this requirement using Disk Data tables — see Chapter 10, *MySQL Cluster Disk Data Tables*, for more information about these.) Naturally, multiple and faster CPUs can enhance performance. Memory requirements for other Cluster processes are relatively small.

The software requirements for Cluster are also modest. Host operating systems do not require any unusual modules, services, applications, or configuration to support MySQL Cluster. For supported operating systems, a standard installation should be sufficient. The MySQL software requirements are simple: all that is needed is a production release of MySQL 5.1.34-ndb-6.2.18 or

5.1.34-ndb-6.3.26 to have Cluster support. It is not necessary to compile MySQL yourself merely to be able to use Cluster. In this *How-To*, we assume that you are using the server binary appropriate to your platform, available via the MySQL Cluster software downloads page at http://dev.mysql.com/downloads/cluster.

For communication between nodes, Cluster supports TCP/IP networking in any standard topology, and the minimum expected for each host is a standard 100 Mbps Ethernet card, plus a switch, hub, or router to provide network connectivity for the cluster as a whole. We strongly recommend that a MySQL Cluster be run on its own subnet which is not shared with non-Cluster machines for the following reasons:

- **Security.** Communications between Cluster nodes are not encrypted or shielded in any way. The only means of protecting transmissions within a MySQL Cluster is to run your Cluster on a protected network. If you intend to use MySQL Cluster for Web applications, the cluster should definitely reside behind your firewall and not in your network's De-Militarized Zone (DMZ) or elsewhere.

  See Section 8.1, "MySQL Cluster Security and Networking Issues", for more information.

- **Efficiency.** Setting up a MySQL Cluster on a private or protected network allows the cluster to make exclusive use of bandwidth between cluster hosts. Using a separate switch for your MySQL Cluster not only helps protect against unauthorized access to Cluster data, it also ensures that Cluster nodes are shielded from interference caused by transmissions between other computers on the network. For enhanced reliability, you can use dual switches and dual cards to remove the network as a single point of failure; many device drivers support failover for such communication links.

It is also possible to use the high-speed Scalable Coherent Interface (SCI) with MySQL Cluster, but this is not a requirement. See Chapter 11, *Using High-Speed Interconnects with MySQL Cluster*, for more about this protocol and its use with MySQL Cluster.

2.2. MySQL Cluster Multi-Computer Installation

Each MySQL Cluster host computer running an SQL node must have installed on it a MySQL binary. For management nodes and data nodes, it is not necessary to install the MySQL server binary, but management nodes require the management server daemon (`ndb_mgmd`) and data nodes require the data node daemon (`ndbd`). It is also a good idea to install the management client (`ndb_mgm`) on the management server host. This section covers the steps necessary to install the correct binaries for each type of Cluster node.

MySQL AB provides precompiled binaries that support Cluster. However, we also include information relating to installing a MySQL Cluster after building MySQL from source. For setting up a cluster using MySQL's binaries, the first step in the installation process for each cluster host is to download the latest MySQL Cluster NDB 6.2 or MySQL Cluster NDB 6.3 binary archive (`mysql-cluster-gpl-6.2.18-linux-i686-glibc23.tar.gz` or `mysql-cluster-gpl-6.3.26-linux-i686-glibc23.tar.gz`, respectively) from the MySQL Cluster downloads area. We assume that you have placed this file in each machine's `/var/tmp` directory. (If you do require a custom binary, see Installing from the Development Source Tree.)

> **Note**
>
> When compiling MySQL Cluster NDB 7.0 from source, no special options are required for building multi-threaded data node binaries. On Unix platforms, configuring the build with any of the options `--plugins=max`, `--plugins=max-no-innodb`, or `--with-ndbcluster` causes `ndbmtd` to be built automatically; `make install` places the `ndbmtd` binary in the `libexec` directory along with `mysqld`, `ndbd`, and `ndb_mgm`. Similarly, on Windows, using `WITH_NDBCLUSTER_STORAGE_ENGINE` with `configure.js` causes `ndbmtd.exe` to be built automatically, and to be found in the `bin` directory of the archive created by `make_win_bin_dist`.

RPMs are also available for both 32-bit and 64-bit Linux platforms. For a MySQL Cluster, three RPMs are required:

- The **Server** RPM (for example, `MySQL-Cluster-gpl-server-6.2.18-0.sles10.i586.rpm` or `MySQL-Cluster-gpl-server-6.3.26-0.sles10.i586.rpm`), which supplies the core files needed to run a MySQL Server with `NDBCLUSTER` storage engine support (that is, as a MySQL Cluster SQL node).

  If you do not have your own client application capable of administering a MySQL server, you should also obtain and install the **Client** RPM (for example, `MySQL-Cluster-gpl-client-6.2.18-0.sles10.i586.rpm` or `MySQL-Cluster-gpl-client-6.3.26-0.sles10.i586.rpm`).

- The **Cluster storage engine** RPM (for example, `MySQL-Cluster-gpl-storage-6.2.18-0.sles10.i586.rpm` or `MySQL-Cluster-gpl-storage-6.3.26-0.sles10.i586.rpm`), which supplies the MySQL Cluster data node binary (`ndbd`).

- The **Cluster storage engine management RPM** (for example, `MySQL-Cluster-gpl-management-6.2.18-0.sles10.i586.rpm` or `MySQL-Cluster-gpl-management-6.3.26-0.sles10.i586.rpm`), which provides the MySQL Cluster management serv-

er binary (ndb_mgmd).

In addition, you should also obtain the **NDB Cluster - Storage engine basic tools** RPM (for example, MySQL-Cluster-gpl-tools-6.2.18-0.sles10.i586.rpm or MySQL-Cluster-gpl-tools-6.3.26-0.sles10.i586.rpm), which supplies several useful applications for working with a MySQL Cluster. The most important of these is the MySQL Cluster management client (ndb_mgm). The **NDB Cluster - Storage engine extra tools** RPM (for example, MySQL-Cluster-gpl-extra-6.2.18-0.sles10.i586.rpm or MySQL-Cluster-gpl-extra-6.3.26-0.sles10.i586.rpm) contains some additional testing and monitoring programs, but is not required to install a MySQL Cluster. (For more information about these additional programs, see Chapter 6, *MySQL Cluster Programs*.)

The MySQL Cluster version number in the RPM file names (shown here as 6.2.18 or 6.3.26) can vary according to the version which you are actually using. *It is very important that all of the Cluster RPMs to be installed have the same version number*. The glibc version number (if present), and architecture designation (shown here as i586) should be appropriate to the machine on which the RPM is to be installed.

See Installing MySQL from RPM Packages on Linux, for general information about installing MySQL using RPMs supplied by MySQL AB.

After installing from RPM, you still need to configure the cluster as discussed in Section 2.3, "MySQL Cluster Multi-Computer Configuration".

> **Note**
>
> After completing the installation, do not yet start any of the binaries. We show you how to do so following the configuration of all nodes.

Data and SQL Node Installation — .tar.gz Binary. On each of the machines designated to host data or SQL nodes, perform the following steps as the system root user:

1. Check your /etc/passwd and /etc/group files (or use whatever tools are provided by your operating system for managing users and groups) to see whether there is already a mysql group and mysql user on the system. Some OS distributions create these as part of the operating system installation process. If they are not already present, create a new mysql user group, and then add a mysql user to this group:

   ```
   shell> groupadd mysql
   shell> useradd -g mysql mysql
   ```

   The syntax for useradd and groupadd may differ slightly on different versions of Unix, or they may have different names such as adduser and addgroup.

2. Change location to the directory containing the downloaded file, unpack the archive, and create a symlink to the mysql directory named mysql. Note that the actual file and directory names will vary according to the MySQL Cluster version number.

   ```
   shell> cd /var/tmp
   shell> tar -C /usr/local -xzvf mysql-cluster-gpl-6.3.26-linux-i686-glibc23.tar.gz
   shell> ln -s /usr/local/mysql-cluster-gpl-6.3.26-linux-i686-glibc23.tar.gz /usr/local/mysql
   ```

3. Change location to the mysql directory and run the supplied script for creating the system databases:

   ```
   shell> cd mysql
   shell> scripts/mysql_install_db --user=mysql
   ```

4. Set the necessary permissions for the MySQL server and data directories:

   ```
   shell> chown -R root .
   shell> chown -R mysql data
   shell> chgrp -R mysql .
   ```

   Note that the data directory on each machine hosting a data node is /usr/local/mysql/data. This piece of information is essential when configuring the management node. (See Section 2.3, "MySQL Cluster Multi-Computer Configuration".)

5. Copy the MySQL startup script to the appropriate directory, make it executable, and set it to start when the operating system is booted up:

   ```
   shell> cp support-files/mysql.server /etc/rc.d/init.d/
   shell> chmod +x /etc/rc.d/init.d/mysql.server
   shell> chkconfig --add mysql.server
   ```

(The startup scripts directory may vary depending on your operating system and version — for example, in some Linux distributions, it is `/etc/init.d`.)

Here we use Red Hat's `chkconfig` for creating links to the startup scripts; use whatever means is appropriate for this purpose on your operating system and distribution, such as `update-rc.d` on Debian.

Remember that the preceding steps must be repeated on each machine where an SQL node is to reside.

SQL node installation — RPM files. On each machine to be used for hosting a cluster SQL node, install the **Server** RPM by executing the following command as the system root user, replacing the name shown for the RPM as necessary to match the name of the RPM downloaded from the MySQL AB web site:

```
shell> rpm -Uhv MySQL-Cluster-gpl-server-6.3.26-0.sles10.i586.rpm
```

This installs the MySQL server binary (`mysqld`) in the `/usr/sbin` directory, as well as all needed MySQL Server support files. It also installs the `mysql.server` and `mysqld_safe` startup scripts in `/usr/share/mysql` and `/usr/bin`, respectively. The RPM installer should take care of general configuration issues (such as creating the `mysql` user and group, if needed) automatically.

> **Note**
>
> To administer the SQL node (MySQL server), you should also install the **Client** RPM, as shown here:
>
> ```
> shell> rpm -Uhv MySQL-Cluster-gpl-client-6.3.26-0.sles10.i586.rpm
> ```
>
> This installs the `mysql` client program.

SQL node installation — building from source. If you compile MySQL with clustering support (for example, by using the `BUILD/compile-`*`platform_name`*`-max` script appropriate to your platform), and perform the default installation (using `make install` as the root user), `mysqld` is placed in `/usr/local/mysql/bin`. Follow the steps given in MySQL Installation Using a Source Distribution to make `mysqld` ready for use. If you want to run multiple SQL nodes, you can use a copy of the same `mysqld` executable and its associated support files on several machines. The easiest way to do this is to copy the entire `/usr/local/mysql` directory and all directories and files contained within it to the other SQL node host or hosts, then repeat the steps from MySQL Installation Using a Source Distribution on each machine. If you configure the build with a non-default `--prefix`, you need to adjust the directory accordingly.

Data node installation — RPM Files. On a computer that is to host a cluster data node it is necessary to install only the **NDB Cluster - Storage engine** RPM. To do so, copy this RPM to the data node host, and run the following command as the system root user, replacing the name shown for the RPM as necessary to match that of the RPM downloaded from the MySQL AB web site:

```
shell> rpm -Uhv MySQL-Cluster-gpl-storage-6.2.18-0.sles10.i586.rpm
```

The previous command installs the MySQL Cluster data node binary (`ndbd`) in the `/usr/sbin` directory.

Data node installation — building from source. The only executable required on a data node host is `ndbd` (`mysqld`, for example, does not have to be present on the host machine). By default when doing a source build, this file is placed in the directory `/usr/local/mysql/libexec`. For installing on multiple data node hosts, only `ndbd` need be copied to the other host machine or machines. (This assumes that all data node hosts use the same architecture and operating system; otherwise you may need to compile separately for each different platform.) `ndbd` need not be in any particular location on the host's file system, as long as the location is known.

Management node installation — `.tar.gz` binary. Installation of the management node does not require the `mysqld` binary. Only the MySQL Cluster management server (`ndb_mgmd`) is required; you most likely want to install the management client (`ndb_mgm`) as well. Both of these binaries also be found in the `.tar.gz` archive. Again, we assume that you have placed this archive in `/var/tmp`.

As system `root` (that is, after using `sudo`, `su root`, or your system's equivalent for temporarily assuming the system administrator account's privileges), perform the following steps to install `ndb_mgmd` and `ndb_mgm` on the Cluster management node host:

1. Change location to the `/var/tmp` directory, and extract the `ndb_mgm` and `ndb_mgmd` from the archive into a suitable directory such as `/usr/local/bin`:

   ```
   shell> cd /var/tmp
   shell> tar -zxvf mysql-5.1.34-ndb-6.3.26-linux-i686-glibc23.tar.gz
   shell> cd mysql-5.1.34-ndb-6.3.26-linux-i686-glibc23
   shell> cp bin/ndb_mgm* /usr/local/bin
   ```

   (You can safely delete the directory created by unpacking the downloaded archive, and the files it contains, from `/var/tmp` once `ndb_mgm` and `ndb_mgmd` have been copied to the executables directory.)

2. Change location to the directory into which you copied the files, and then make both of them executable:

```
shell> cd /usr/local/bin
shell> chmod +x ndb_mgm*
```

Management node installation — RPM file. To install the MySQL Cluster management server, it is necessary only to use the **NDB Cluster - Storage engine management** RPM. Copy this RPM to the computer intended to host the management node, and then install it by running the following command as the system root user (replace the name shown for the RPM as necessary to match that of the **Storage engine management** RPM downloaded from the MySQL AB web site):

```
shell> rpm -Uhv MySQL-Cluster-gpl-management-6.3.26-0.sles10.i586.rpm
```

This installs the management server binary (`ndb_mgmd`) to the `/usr/sbin` directory.

You should also install the `NDB` management client, which is supplied by the **Storage engine basic tools** RPM. Copy this RPM to the same computer as the management node, and then install it by running the following command as the system root user (again, replace the name shown for the RPM as necessary to match that of the **Storage engine basic tools** RPM downloaded from the MySQL AB web site):

```
shell> rpm -Uhv MySQL-Cluster-gpl-tools-6.3.26-0.sles10.i586.rpm
```

The **Storage engine basic tools** RPM installs the MySQL Cluster management client (`ndb_mgm`) to the `/usr/bin` directory.

> **Note**
>
> You can also install the **Cluster storage engine extra tools** RPM, if you wish, as shown here:
>
> ```
> shell> rpm -Uhv MySQL-Cluster-gpl-extra-6.3.26-0.sles10.i586.rpm
> ```
>
> You may find the extra tools useful; however the **Cluster storage engine extra tools** RPM is *not* required to install a working MySQL Cluster.

Management node installation — building from source. When building from source and running the default `make install`, the management server binary (`ndb_mgmd`) is placed in `/usr/local/mysql/libexec`, while the management client binary (`ndb_mgm`) can be found in `/usr/local/mysql/bin`. Only `ndb_mgmd` is required to be present on a management node host; however, it is also a good idea to have `ndb_mgm` present on the same host machine. Neither of these executables requires a specific location on the host machine's file system.

In Section 2.3, “MySQL Cluster Multi-Computer Configuration”, we create configuration files for all of the nodes in our example MySQL Cluster.

MySQL Cluster on Windows (*alpha*). In MySQL Cluster NDB 7.0, experimental support is added for Microsoft Windows platforms. To compile MySQL Cluster from source on Windows, you must configure the build using the `WITH_NDBCLUSTER_STORAGE_ENGINE` option before creating the Visual Studio project files. After running `make_win_bin_dist`, the MySQL Cluster binaries can be found in the `bin` directory of the resulting archive. For more information, see Installing MySQL from Source on Windows.

2.3. MySQL Cluster Multi-Computer Configuration

For our four-node, four-host MySQL Cluster, it is necessary to write four configuration files, one per node host.

- Each data node or SQL node requires a `my.cnf` file that provides two pieces of information: a *connectstring* that tells the node where to find the management node, and a line telling the MySQL server on this host (the machine hosting the data node) to enable the `NDBCLUSTER` storage engine.

  For more information on connectstrings, see Section 3.4.3, “The MySQL Cluster Connectstring”.

- The management node needs a `config.ini` file telling it how many replicas to maintain, how much memory to allocate for data and indexes on each data node, where to find the data nodes, where to save data to disk on each data node, and where to find any SQL nodes.

Configuring the Storage and SQL Nodes

The `my.cnf` file needed for the data nodes is fairly simple. The configuration file should be located in the `/etc` directory and can be edited using any text editor. (Create the file if it does not exist.) For example:

```
shell> vi /etc/my.cnf
```

> **Note**

We show `vi` being used here to create the file, but any text editor should work just as well.

For each data node and SQL node in our example setup, `my.cnf` should look like this:

```
# Options for mysqld process:
[mysqld]
ndbcluster                      # run NDB storage engine
ndb-connectstring=192.168.0.10  # location of management server
# Options for ndbd process:
[mysql_cluster]
ndb-connectstring=192.168.0.10  # location of management server
```

After entering the preceding information, save this file and exit the text editor. Do this for the machines hosting data node “A”, data node “B”, and the SQL node.

Important

Once you have started a `mysqld` process with the `NDBCLUSTER` and `ndb-connectstring` parameters in the `[mysqld]` in the `my.cnf` file as shown previously, you cannot execute any `CREATE TABLE` or `ALTER TABLE` statements without having actually started the cluster. Otherwise, these statements will fail with an error. *This is by design.*

Configuring the management node. The first step in configuring the management node is to create the directory in which the configuration file can be found and then to create the file itself. For example (running as `root`):

```
shell> mkdir /var/lib/mysql-cluster
shell> cd /var/lib/mysql-cluster
shell> vi config.ini
```

For our representative setup, the `config.ini` file should read as follows:

```
# Options affecting ndbd processes on all data nodes:
[ndbd default]
NoOfReplicas=2    # Number of replicas
DataMemory=80M    # How much memory to allocate for data storage
IndexMemory=18M   # How much memory to allocate for index storage
                  # For DataMemory and IndexMemory, we have used the
                  # default values. Since the "world" database takes up
                  # only about 500KB, this should be more than enough for
                  # this example Cluster setup.
# TCP/IP options:
[tcp default]
portnumber=2202   # This the default; however, you can use any port that is free
                  # for all the hosts in the cluster
                  # Note: It is recommended that you do not specify the port
                  # number at all and allow the default value to be used instead
# Management process options:
[ndb_mgmd]
hostname=192.168.0.10           # Hostname or IP address of management node
datadir=/var/lib/mysql-cluster  # Directory for management node log files
# Options for data node "A":
[ndbd]
                                # (one [ndbd] section per data node)
hostname=192.168.0.30           # Hostname or IP address
datadir=/usr/local/mysql/data   # Directory for this data node's data files
# Options for data node "B":
[ndbd]
hostname=192.168.0.40           # Hostname or IP address
datadir=/usr/local/mysql/data   # Directory for this data node's data files
# SQL node options:
[mysqld]
hostname=192.168.0.20           # Hostname or IP address
                                # (additional mysqld connections can be
                                # specified for this node for various
                                # purposes such as running ndb_restore)
```

Note

The `world` database can be downloaded from http://dev.mysql.com/doc/, where it can be found listed under “Examples”.

After all the configuration files have been created and these minimal options have been specified, you are ready to proceed with starting the cluster and verifying that all processes are running. We discuss how this is done in Section 2.4, “Initial Startup of MySQL Cluster”.

For more detailed information about the available MySQL Cluster configuration parameters and their uses, see Section 3.4, “MySQL Cluster Configuration Files”, and Chapter 3, *MySQL Cluster Configuration*. For configuration of MySQL Cluster as relates to making backups, see Section 7.3.3, “Configuration for MySQL Cluster Backups”.

Note

The default port for Cluster management nodes is 1186; the default port for data nodes is 2202. However, the cluster can automatically allocate ports for data nodes from those that are already free.

2.4. Initial Startup of MySQL Cluster

Starting the cluster is not very difficult after it has been configured. Each cluster node process must be started separately, and on the host where it resides. The management node should be started first, followed by the data nodes, and then finally by any SQL nodes:

1. On the management host, issue the following command from the system shell to start the management node process:

```
shell> ndb_mgmd -f /var/lib/mysql-cluster/config.ini
```

Note

`ndb_mgmd` must be told where to find its configuration file, using the `-f` or `--config-file` option. (See Section 6.4, "`ndb_mgmd` — The MySQL Cluster Management Server Daemon", for details.)

For additional options which can be used with `ndb_mgmd`, see Section 6.23, "Options Common to MySQL Cluster Programs".

2. On each of the data node hosts, run this command to start the `ndbd` process:

```
shell> ndbd
```

3. If you used RPM files to install MySQL on the cluster host where the SQL node is to reside, you can (and should) use the supplied startup script to start the MySQL server process on the SQL node.

If all has gone well, and the cluster has been set up correctly, the cluster should now be operational. You can test this by invoking the `ndb_mgm` management node client. The output should look like that shown here, although you might see some slight differences in the output depending upon the exact version of MySQL that you are using:

```
shell> ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm> SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=2    @192.168.0.30  (Version: 5.1.34-ndb-6.3.26, Nodegroup: 0, Master)
id=3    @192.168.0.40  (Version: 5.1.34-ndb-6.3.26, Nodegroup: 0)
[ndb_mgmd(MGM)] 1 node(s)
id=1    @192.168.0.10  (Version: 5.1.34-ndb-6.3.26)
[mysqld(API)]   1 node(s)
id=4    @192.168.0.20  (Version: 5.1.34-ndb-6.3.26)
```

The SQL node is referenced here as `[mysqld(API)]`, which reflects the fact that the `mysqld` process is acting as a MySQL Cluster API node.

Note

The IP address shown for a given MySQL Cluster SQL or other API node in the output of `SHOW` is the address used by the SQL or API node to connect to the cluster data nodes, and not to any management node.

You should now be ready to work with databases, tables, and data in MySQL Cluster. See Section 2.5, "Loading Sample Data into MySQL Cluster and Performing Queries", for a brief discussion.

2.5. Loading Sample Data into MySQL Cluster and Performing Queries

Working with data in MySQL Cluster is not much different from doing so in MySQL without Cluster. There are two points to keep in mind:

- For a table to be replicated in the cluster, it must use the `NDBCLUSTER` storage engine. To specify this, use the `ENGINE=NDBCLUSTER` or `ENGINE=NDB` option when creating the table:

```
CREATE TABLE tbl_name (col_name column_definitions) ENGINE=NDBCLUSTER;
```

Alternatively, for an existing table that uses a different storage engine, use `ALTER TABLE` to change the table to use `NDBCLUSTER`:

```
ALTER TABLE tbl_name ENGINE=NDBCLUSTER;
```

- Each `NDBCLUSTER` table *must* have a primary key. If no primary key is defined by the user when a table is created, the `NDBCLUSTER` storage engine automatically generates a hidden one.

  > **Note**
  >
  > This hidden key takes up space just as does any other table index. It is not uncommon to encounter problems due to insufficient memory for accommodating these automatically created indexes.)

If you are importing tables from an existing database using the output of `mysqldump`, you can open the SQL script in a text editor and add the `ENGINE` option to any table creation statements, or replace any existing `ENGINE` (or `TYPE`) options. Suppose that you have the `world` sample database on another MySQL server that does not support MySQL Cluster, and you want to export the `City` table:

```
shell> mysqldump --add-drop-table world City > city_table.sql
```

The resulting `city_table.sql` file will contain this table creation statement (and the `INSERT` statements necessary to import the table data):

```
DROP TABLE IF EXISTS `City`;
CREATE TABLE `City` (
  `ID` int(11) NOT NULL auto_increment,
  `Name` char(35) NOT NULL default '',
  `CountryCode` char(3) NOT NULL default '',
  `District` char(20) NOT NULL default '',
  `Population` int(11) NOT NULL default '0',
  PRIMARY KEY  (`ID`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
INSERT INTO `City` VALUES (1,'Kabul','AFG','Kabol',1780000);
INSERT INTO `City` VALUES (2,'Qandahar','AFG','Qandahar',237500);
INSERT INTO `City` VALUES (3,'Herat','AFG','Herat',186800);
(remaining INSERT statements omitted)
```

You need to make sure that MySQL uses the `NDBCLUSTER` storage engine for this table. There are two ways that this can be accomplished. One of these is to modify the table definition *before* importing it into the Cluster database. Using the `City` table as an example, modify the `ENGINE` option of the definition as follows:

```
DROP TABLE IF EXISTS `City`;
CREATE TABLE `City` (
  `ID` int(11) NOT NULL auto_increment,
  `Name` char(35) NOT NULL default '',
  `CountryCode` char(3) NOT NULL default '',
  `District` char(20) NOT NULL default '',
  `Population` int(11) NOT NULL default '0',
  PRIMARY KEY  (`ID`)
) ENGINE=NDBCLUSTER DEFAULT CHARSET=latin1;
INSERT INTO `City` VALUES (1,'Kabul','AFG','Kabol',1780000);
INSERT INTO `City` VALUES (2,'Qandahar','AFG','Qandahar',237500);
INSERT INTO `City` VALUES (3,'Herat','AFG','Herat',186800);
(remaining INSERT statements omitted)
```

This must be done for the definition of each table that is to be part of the clustered database. The easiest way to accomplish this is to do a search-and-replace on the file that contains the definitions and replace all instances of `TYPE=engine_name` or `ENGINE=engine_name` with `ENGINE=NDBCLUSTER`. If you do not want to modify the file, you can use the unmodified file to create the tables, and then use `ALTER TABLE` to change their storage engine. The particulars are given later in this section.

Assuming that you have already created a database named `world` on the SQL node of the cluster, you can then use the `mysql` command-line client to read `city_table.sql`, and create and populate the corresponding table in the usual manner:

```
shell> mysql world < city_table.sql
```

It is very important to keep in mind that the preceding command must be executed on the host where the SQL node is running (in this case, on the machine with the IP address `192.168.0.20`).

To create a copy of the entire `world` database on the SQL node, use `mysqldump` on the non-cluster server to export the database to a file named `world.sql`; for example, in the `/tmp` directory. Then modify the table definitions as just described and import the file into the SQL node of the cluster like this:

```
shell> mysql world < /tmp/world.sql
```

If you save the file to a different location, adjust the preceding instructions accordingly.

Running `SELECT` queries on the SQL node is no different from running them on any other instance of a MySQL server. To run queries from the command line, you first need to log in to the MySQL Monitor in the usual way (specify the `root` password at the `Enter password:` prompt):

```
shell> mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.1.34-ndb-6.2.18
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql>
```

We simply use the MySQL server's `root` account and assume that you have followed the standard security precautions for installing a MySQL server, including setting a strong `root` password. For more information, see Securing the Initial MySQL Accounts.

It is worth taking into account that Cluster nodes do *not* make use of the MySQL privilege system when accessing one another. Setting or changing MySQL user accounts (including the `root` account) effects only applications that access the SQL node, not interaction between nodes. See Section 8.2, “MySQL Cluster and MySQL Privileges”, for more information.

If you did not modify the `ENGINE` clauses in the table definitions prior to importing the SQL script, you should run the following statements at this point:

```
mysql> USE world;
mysql> ALTER TABLE City ENGINE=NDBCLUSTER;
mysql> ALTER TABLE Country ENGINE=NDBCLUSTER;
mysql> ALTER TABLE CountryLanguage ENGINE=NDBCLUSTER;
```

Selecting a database and running a `SELECT` query against a table in that database is also accomplished in the usual manner, as is exiting the MySQL Monitor:

```
mysql> USE world;
mysql> SELECT Name, Population FROM City ORDER BY Population DESC LIMIT 5;
+-----------+------------+
| Name      | Population |
+-----------+------------+
| Bombay    |   10500000 |
| Seoul     |    9981619 |
| SÃ£o Paulo |    9968485 |
| Shanghai  |    9696300 |
| Jakarta   |    9604900 |
+-----------+------------+
5 rows in set (0.34 sec)
mysql> \q
Bye
shell>
```

Applications that use MySQL can employ standard APIs to access `NDB` tables. It is important to remember that your application must access the SQL node, and not the management or data nodes. This brief example shows how we might execute the `SELECT` statement just shown by using the PHP 5.X `mysqli` extension running on a Web server elsewhere on the network:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type"
        content="text/html; charset=iso-8859-1">
  <title>SIMPLE mysqli SELECT</title>
</head>
<body>
<?php
  # connect to SQL node:
  $link = new mysqli('192.168.0.20', 'root', 'root_password', 'world');
  # parameters for mysqli constructor are:
  #   host, user, password, database
  if( mysqli_connect_errno() )
    die("Connect failed: " . mysqli_connect_error());
  $query = "SELECT Name, Population
            FROM City
            ORDER BY Population DESC
            LIMIT 5";
  # if no errors...
  if( $result = $link->query($query) )
  {
?>
<table border="1" width="40%" cellpadding="4" cellspacing ="1">
  <tbody>
  <tr>
    <th width="10%">City</th>
    <th>Population</th>
  </tr>
<?
    # then display the results...
```

```
    while($row = $result->fetch_object())
      printf("<tr>\n  <td align=\"center\">%s</td><td>%d</td>\n</tr>\n",
              $row->Name, $row->Population);
?>
  </tbody
</table>
<?
  # ...and verify the number of rows that were retrieved
    printf("<p>Affected rows: %d</p>\n", $link->affected_rows);
  }
  else
    # otherwise, tell us what went wrong
    echo mysqli_error();
  # free the result set and the mysqli connection object
  $result->close();
  $link->close();
?>
</body>
</html>
```

We assume that the process running on the Web server can reach the IP address of the SQL node.

In a similar fashion, you can use the MySQL C API, Perl-DBI, Python-mysql, or MySQL AB's own Connectors to perform the tasks of data definition and manipulation just as you would normally with MySQL.

2.6. Safe Shutdown and Restart of MySQL Cluster

To shut down the cluster, enter the following command in a shell on the machine hosting the management node:

```
shell> ndb_mgm -e shutdown
```

The `-e` option here is used to pass a command to the `ndb_mgm` client from the shell. (See Section 6.23, "Options Common to MySQL Cluster Programs", for more information about this option.) The command causes the `ndb_mgm`, `ndb_mgmd`, and any `ndbd` processes to terminate gracefully. Any SQL nodes can be terminated using `mysqladmin shutdown` and other means.

To restart the cluster, run these commands:

- On the management host (`192.168.0.10` in our example setup):

  ```
  shell> ndb_mgmd -f /var/lib/mysql-cluster/config.ini
  ```

- On each of the data node hosts (`192.168.0.30` and `192.168.0.40`):

  ```
  shell> ndbd
  ```

- On the SQL host (`192.168.0.20`):

  ```
  shell> mysqld_safe &
  ```

In a production setting, it is usually not desirable to shut down the cluster completely. In many cases, even when making configuration changes, or performing upgrades to the cluster hardware or software (or both), which require shutting down individual host machines, it is possible to do so without shutting down the cluster as a whole by performing a *rolling restart* of the cluster. For more information about doing this, see Section 5.1, "Performing a Rolling Restart of a MySQL Cluster".

Chapter 3. MySQL Cluster Configuration

A MySQL server that is part of a MySQL Cluster differs in one chief respect from a normal (non-clustered) MySQL server, in that it employs the `NDBCLUSTER` storage engine. This engine is also referred to simply as `NDB`, and the two forms of the name are synonymous.

To avoid unnecessary allocation of resources, the server is configured by default with the `NDB` storage engine disabled. To enable `NDB`, you must modify the server's `my.cnf` configuration file, or start the server with the `--ndbcluster` option.

For more information about `--ndbcluster` and other MySQL server options specific to MySQL Cluster, see Section 4.2, "`mysqld` Command Options for MySQL Cluster".

The MySQL server is a part of the cluster, so it also must know how to access an MGM node to obtain the cluster configuration data. The default behavior is to look for the MGM node on `localhost`. However, should you need to specify that its location is elsewhere, this can be done in `my.cnf` or on the MySQL server command line. Before the `NDB` storage engine can be used, at least one MGM node must be operational, as well as any desired data nodes.

3.1. Building MySQL Cluster from Source Code

`NDBCLUSTER`, the MySQL Cluster storage engine, is available in binary distributions for Linux, Mac OS X, and Solaris. We are working to make Cluster run on all operating systems supported by MySQL, including Windows. Beginning with MySQL Cluster NDB 6.4.0, there is experimental support for MySQL Cluster on Windows.

If you choose to build from a source tarball or one of the MySQL Cluster public development trees, be sure to use the `--with-ndbcluster` option when running `configure` (if building from source on Microsoft Windows platforms, use the `WITH_NDBCLUSTER_STORAGE_ENGINE` option with `configure.js`). You can also use the `BUILD/compile-pentium-max` build scripts provided for most Unix platforms. Note that this script includes OpenSSL, so you must either have or obtain OpenSSL to build successfully, or else modify `compile-pentium-max` to exclude this requirement. Of course, you can also just follow the standard instructions for compiling your own binaries, and then perform the usual tests and installation procedure. See Installing from the Development Source Tree.

`BUILD/compile-pentium-max` also includes OpenSSL, so you must either have or obtain OpenSSL to build successfully, or else modify `compile-pentium-max` to exclude this requirement. Of course, you can also just follow the standard instructions for compiling your own binaries, and then perform the usual tests and installation procedure. See Installing from the Development Source Tree.

You should also note that `compile-pentium-max` installs MySQL to the directory `/usr/local/mysql`, placing all MySQL Cluster executables, scripts, databases, and support files in subdirectories under this directory. If this is not what you desire, be sure to modify the script accordingly.

3.2. Installing MySQL Cluster Software

In the next few sections, we assume that you are already familiar with installing MySQL, and here we cover only the differences between configuring MySQL Cluster and configuring MySQL without clustering. (See Installing and Upgrading MySQL, if you require more information about the latter.)

You will find Cluster configuration easiest if you have already have all management and data nodes running first; this is likely to be the most time-consuming part of the configuration. Editing the `my.cnf` file is fairly straightforward, and this section will cover only any differences from configuring MySQL without clustering.

3.3. Quick Test Setup of MySQL Cluster

To familiarize you with the basics, we will describe the simplest possible configuration for a functional MySQL Cluster. After this, you should be able to design your desired setup from the information provided in the other relevant sections of this chapter.

First, you need to create a configuration directory such as `/var/lib/mysql-cluster`, by executing the following command as the system `root` user:

```
shell> mkdir /var/lib/mysql-cluster
```

In this directory, create a file named `config.ini` that contains the following information. Substitute appropriate values for `HostName` and `DataDir` as necessary for your system.

```
# file "config.ini" - showing minimal setup consisting of 1 data node,
# 1 management server, and 3 MySQL servers.
# The empty default sections are not required, and are shown only for
# the sake of completeness.
# Data nodes must provide a hostname but MySQL Servers are not required
```

```
# to do so.
# If you don't know the hostname for your machine, use localhost.
# The DataDir parameter also has a default value, but it is recommended to
# set it explicitly.
# Note: [db], [api], and [mgm] are aliases for [ndbd], [mysqld], and [ndb_mgmd],
# respectively. [db] is deprecated and should not be used in new installations.
[ndbd default]
NoOfReplicas= 1
[mysqld  default]
[ndb_mgmd default]
[tcp default]
[ndb_mgmd]
HostName= myhost.example.com
[ndbd]
HostName= myhost.example.com
DataDir= /var/lib/mysql-cluster
[mysqld]
[mysqld]
[mysqld]
```

You can now start the `ndb_mgmd` management server. By default, it attempts to read the `config.ini` file in its current working directory, so change location into the directory where the file is located and then invoke `ndb_mgmd`:

```
shell> cd /var/lib/mysql-cluster
shell> ndb_mgmd
```

Then start a single data node by running `ndbd`:

```
shell> ndbd
```

For command-line options which can be used when starting `ndbd`, see Section 6.23, “Options Common to MySQL Cluster Programs”.

By default, `ndbd` looks for the management server at `localhost` on port 1186.

> **Note**
>
> If you have installed MySQL from a binary tarball, you will need to specify the path of the `ndb_mgmd` and `ndbd` servers explicitly. (Normally, these will be found in `/usr/local/mysql/bin`.)

Finally, change location to the MySQL data directory (usually `/var/lib/mysql` or `/usr/local/mysql/data`), and make sure that the `my.cnf` file contains the option necessary to enable the NDB storage engine:

```
[mysqld]
ndbcluster
```

You can now start the MySQL server as usual:

```
shell> mysqld_safe --user=mysql &
```

Wait a moment to make sure the MySQL server is running properly. If you see the notice `mysql ended`, check the server's `.err` file to find out what went wrong.

If all has gone well so far, you now can start using the cluster. Connect to the server and verify that the `NDBCLUSTER` storage engine is enabled:

```
shell> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.1.36
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> SHOW ENGINES\G
...
*************************** 12. row ***************************
Engine: NDBCLUSTER
Support: YES
Comment: Clustered, fault-tolerant, memory-based tables
*************************** 13. row ***************************
Engine: NDB
Support: YES
Comment: Alias for NDBCLUSTER
...
```

The row numbers shown in the preceding example output may be different from those shown on your system, depending upon how your server is configured.

Try to create an `NDBCLUSTER` table:

```
shell> mysql
```

```
mysql> USE test;
Database changed
mysql> CREATE TABLE ctest (i INT) ENGINE=NDBCLUSTER;
Query OK, 0 rows affected (0.09 sec)
mysql> SHOW CREATE TABLE ctest \G
*************************** 1. row ***************************
       Table: ctest
Create Table: CREATE TABLE `ctest` (
  `i` int(11) default NULL
) ENGINE=ndbcluster DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

To check that your nodes were set up properly, start the management client:

```
shell> ndb_mgm
```

Use the `SHOW` command from within the management client to obtain a report on the cluster's status:

```
ndb_mgm> SHOW
Cluster Configuration
---------------------
[ndbd(NDB)]     1 node(s)
id=2    @127.0.0.1  (Version: 3.5.3, Nodegroup: 0, Master)
[ndb_mgmd(MGM)] 1 node(s)
id=1    @127.0.0.1  (Version: 3.5.3)
[mysqld(API)]   3 node(s)
id=3    @127.0.0.1  (Version: 3.5.3)
id=4 (not connected, accepting connect from any host)
id=5 (not connected, accepting connect from any host)
```

At this point, you have successfully set up a working MySQL Cluster. You can now store data in the cluster by using any table created with `ENGINE=NDBCLUSTER` or its alias `ENGINE=NDB`.

3.4. MySQL Cluster Configuration Files

Configuring MySQL Cluster requires working with two files:

- `my.cnf`: Specifies options for all MySQL Cluster executables. This file, with which you should be familiar with from previous work with MySQL, must be accessible by each executable running in the cluster.
- `config.ini`: This file, sometimes known as the *global configuration file*, is read only by the MySQL Cluster management server, which then distributes the information contained therein to all processes participating in the cluster. `config.ini` contains a description of each node involved in the cluster. This includes configuration parameters for data nodes and configuration parameters for connections between all nodes in the cluster. For a quick reference to the sections that can appear in this file, and what sorts of configuration parameters may be placed in each section, see Sections of the `config.ini` File.

Caching of configuration data. Beginning with MySQL Cluster NDB 6.4.0, MySQL Cluster uses *stateful configuration*. The global configuration file is no longer read every time the management server is restarted. Instead, the management server caches the configuration the first time it is started, and thereafter, the global confiuration file is read only when one of the following items is true:

- **The management server is started using `--initial` option.** In this case, the global configuration file is re-read, any existing cache files are deleted, and the management server creates a new configuration cache.
- **The management server is started using `--reload` option.** In this case, the management server compares its cache with the global configuration file. If they differ, the management server creates a new configuration cache; any existing configuration cache is preserved, but not used. If the management server's cache and the global configuration file contain the same configuration data, then the existing cache is used, and no new cache is created.
- **No configuration cache is found.** In this case, the management server reads the global configuration file and creates a cache containing the same configuration data as found in the file.

Configuration cache files. Beginning with MySQL Cluster 6.4.0, the management server by default creates configuration cache files in a directory named `mysql-cluster` in the MySQL installation directory. (If you build MySQL Cluster from source on a Unix system, the default location is `/usr/local/mysql-cluster`.) This can be overridden at run time by starting the management server with the `--configdir` option. Configuration cache files are binary files named according to the pattern `ndb_node_id_config.bin.seq_id`, where *`node_id`* is the management server's node ID in the cluster, and *`seq_id`* is a cache idenitifer. Cache files are numbered sequentially using *`seq_id`*, in the order in which they are created. The management server uses the latest cache file as determined by the *`seq_id`*.

Note

It is possible to roll back to a previous configuration by deleting later configuration cache files, or by renaming an earlier cache file so that it has a higher *`seq_id`*. However, since configuration cache files are written in a binary format, you should not attempt to edit their contents by hand.

For more information about the `--configdir`, `--initial`, and `--reload` options for the MySQL Cluster management server, see Section 6.24.3, "Program Options for ndb_mgmd".

We are continuously making improvements in Cluster configuration and attempting to simplify this process. Although we strive to maintain backward compatibility, there may be times when introduce an incompatible change. In such cases we will try to let Cluster users know in advance if a change is not backward compatible. If you find such a change and we have not documented it, please report it in the MySQL bugs database using the instructions given in How to Report Bugs or Problems.

3.4.1. MySQL Cluster Configuration — Basic Example

To support MySQL Cluster, you will need to update `my.cnf` as shown in the following example. You may also specify these parameters on the command line when invoking the executables.

Note

The options shown here should not be confused with those that are used in `config.ini` global configuration files. Global configuration options are discussed later in this section.

```
# my.cnf
# example additions to my.cnf for MySQL Cluster
# (valid in MySQL 5.1)
# enable ndbcluster storage engine, and provide connectstring for
# management server host (default port is 1186)
[mysqld]
ndbcluster
ndb-connectstring=ndb_mgmd.mysql.com
# provide connectstring for management server host (default port: 1186)
[ndbd]
connect-string=ndb_mgmd.mysql.com
# provide connectstring for management server host (default port: 1186)
[ndb_mgm]
connect-string=ndb_mgmd.mysql.com
# provide location of cluster configuration file
[ndb_mgmd]
config-file=/etc/config.ini
```

(For more information on connectstrings, see Section 3.4.3, "The MySQL Cluster Connectstring".)

```
# my.cnf
# example additions to my.cnf for MySQL Cluster
# (will work on all versions)
# enable ndbcluster storage engine, and provide connectstring for management
# server host to the default port 1186
[mysqld]
ndbcluster
ndb-connectstring=ndb_mgmd.mysql.com:1186
```

Important

Once you have started a `mysqld` process with the `NDBCLUSTER` and `ndb-connectstring` parameters in the `[mysqld]` in the `my.cnf` file as shown previously, you cannot execute any `CREATE TABLE` or `ALTER TABLE` statements without having actually started the cluster. Otherwise, these statements will fail with an error. *This is by design.*

You may also use a separate `[mysql_cluster]` section in the cluster `my.cnf` file for settings to be read and used by all executables:

```
# cluster-specific settings
[mysql_cluster]
ndb-connectstring=ndb_mgmd.mysql.com:1186
```

For additional `NDB` variables that can be set in the `my.cnf` file, see Section 4.3, "MySQL Cluster System Variables".

The MySQL Cluster global configuration file is named `config.ini` by default. It is read by `ndb_mgmd` at startup and can be placed anywhere. Its location and name are specified by using `--config-file=`*`path_name`* on the `ndb_mgmd` command line. If the configuration file is not specified, `ndb_mgmd` by default tries to read a file named `config.ini` located in the current working directory.

The global configuration file for MySQL Cluster uses INI format, which consists of sections preceded by section headings (surrounded by square brackets), followed by the appropriate parameter names and values. One deviation from the standard INI format is that the parameter name and value can be separated by a colon ("`:`") as well as the equals sign ("`=`"); however, the equals

sign is preferred. Another deviation is that sections are not uniquely identified by section name. Instead, unique sections (such as two different nodes of the same type) are identified by a unique ID specified as a parameter within the section.

Default values are defined for most parameters, and can also be specified in `config.ini`. (*Exception*: The `NoOfReplicas` configuration parameter has no default value, and must always be specified explicitly in the `[ndbd default]` section.) To create a default value section, simply add the word `default` to the section name. For example, an `[ndbd]` section contains parameters that apply to a particular data node, whereas an `[ndbd default]` section contains parameters that apply to all data nodes. Suppose that all data nodes should use the same data memory size. To configure them all, create an `[ndbd default]` section that contains a `DataMemory` line to specify the data memory size.

The global configuration file must define the computers and nodes involved in the cluster and on which computers these nodes are located. An example of a simple configuration file for a cluster consisting of one management server, two data nodes and two MySQL servers is shown here:

```
# file "config.ini" - 2 data nodes and 2 SQL nodes
# This file is placed in the startup directory of ndb_mgmd (the
# management server)
# The first MySQL Server can be started from any host. The second
# can be started only on the host mysqld_5.mysql.com
[ndbd default]
NoOfReplicas= 2
DataDir= /var/lib/mysql-cluster
[ndb_mgmd]
Hostname= ndb_mgmd.mysql.com
DataDir= /var/lib/mysql-cluster
[ndbd]
HostName= ndbd_2.mysql.com
[ndbd]
HostName= ndbd_3.mysql.com
[mysqld]
[mysqld]
HostName= mysqld_5.mysql.com
```

> **Note**
>
> The preceding example is intended as a minimal starting configuration for purposes of familiarization with MySQL Cluster, and is almost certain not to be sufficient for production settings. See Section 3.4.2, “Recommended Starting Configurations for MySQL Cluster NDB 6.2 and Later”, which provides more complete example starting configurations for use with MySQL Cluster NDB 6.2 and newer versions of MySQL Cluster.

Each node has its own section in the `config.ini` file. For example, this cluster has two data nodes, so the preceding configuration file contains two `[ndbd]` sections defining these nodes.

> **Note**
>
> Do not place comments on the same line as a section heading in the `config.ini` file; this causes the management server not to start because it cannot parse the configuration file in such cases.

Sections of the `config.ini` File

There are six different sections that you can use in the `config.ini` configuration file, as described in the following list:

- `[computer]`: Defines cluster hosts. This is not required to configure a viable MySQL Cluster, but be may used as a convenience when setting up a large cluster. See Section 3.4.4, “Defining Computers in a MySQL Cluster”, for more information.
- `[ndbd]`: Defines a cluster data node (`ndbd` process). See Section 3.4.6, “Defining MySQL Cluster Data Nodes”, for details.
- `[mysqld]`: Defines the cluster's MySQL server nodes (also called SQL or API nodes). For a discussion of SQL node configuration, see Section 3.4.7, “Defining SQL and Other API Nodes in a MySQL Cluster”.
- `[mgm]` or `[ndb_mgmd]`: Defines a cluster management server (MGM) node. For information concerning the configuration of MGM nodes, see Section 3.4.5, “Defining a MySQL Cluster Management Server”.
- `[tcp]`: Defines a TCP/IP connection between cluster nodes, with TCP/IP being the default connection protocol. Normally, `[tcp]` or `[tcp default]` sections are not required to set up a MySQL Cluster, as the cluster handles this automatically; however, it may be necessary in some situations to override the defaults provided by the cluster. See Section 3.4.8, “MySQL Cluster TCP/IP Connections”, for information about available TCP/IP configuration parameters and how to use them. (You may also find Section 3.4.9, “MySQL Cluster TCP/IP Connections Using Direct Connections” to be of interest in some cases.)
- `[shm]`: Defines shared-memory connections between nodes. In MySQL 5.1, it is enabled by default, but should still be considered experimental. For a discussion of SHM interconnects, see Section 3.4.10, “MySQL Cluster Shared-Memory Connections”.
- `[sci]`:Defines *Scalable Coherent Interface* connections between cluster data nodes. Such connections require software which, while freely available, is not part of the MySQL Cluster distribution, as well as specialised hardware. See Section 3.4.11, “SCI

Transport Connections in MySQL Cluster" for detailed information about SCI interconnects.

You can define `default` values for each section. All Cluster parameter names are case-insensitive, which differs from parameters specified in `my.cnf` or `my.ini` files.

3.4.2. Recommended Starting Configurations for MySQL Cluster NDB 6.2 and Later

Achieving the best performance from a MySQL Cluster depends on a number of factors including the following:

- MySQL Cluster software version
- Numbers of data nodes and SQL nodes
- Hardware
- Operating system
- Amount of data to be stored
- Size and type of load under which the cluster is to operate

Therefore, obtaining an optimum configuration is likely to be an iterative process, the outcome of which can vary widely with the specifics of each MySQL Cluster deployment. Changes in configuration are also likely to be indicated when changes are made in the platform on which the cluster is run, or in applications that use the MySQL Cluster's data. For these reasons, it is not possible to offer a single configuration that is ideal for all usage scenarios. However, in this section, we provide recommended base configurations for MySQL Cluster NDB 6.2 and 6.3 that can serve as reasonable starting points.

Starting configuration for MySQL Cluster NDB 6.2. The following is a recommended starting point for configuring a cluster running MySQL Cluster NDB 6.2.

```
# TCP PARAMETERS
[tcp default]
SendBufferMemory=2M
ReceiveBufferMemory=2M
# Increasing the sizes of these 2 buffers beyond the default values
# helps prevent bottlenecks due to slow disk I/O.
# MANAGEMENT NODE PARAMETERS
[ndb_mgmd default]
DataDir=path/to/management/server/data/directory
# It is possible to use a different data directory for each management
# server, but for ease of administration it is preferable to be
# consistent.
[ndb_mgmd]
HostName=management-server-1-hostname
# Id=management-server-A-id
[ndb_mgmd]
HostName=management-server-2-hostname
# Using 2 management servers helps guarantee that there is always an
# arbitrator in the event of network partitioning, and so is
# recommended for high availability. Each management server must be
# identified by a HostName. You may for the sake of convenience specify
# a node ID for any management server, although one will be allocated
# for it automatically; if you do so, it must be in the range 1-255
# inclusive and must be unique among all IDs specified for cluster
# nodes.
# DATA NODE PARAMETERS
[ndbd default]
NoOfReplicas=2
# This parameter has no default value; it must always must be set
# explicitly. Using 2 replicas is recommended to guarantee
# availability of data; using only 1 replica does not provide any
# redundancy, which means that the failure of a single data node causes
# the entire cluster to shut down. We do not recommend using more than
# 2 replicas, since 2 is sufficient to provide high availability, and
# we do not currently test with greater values for this parameter.
LockPagesInMainMemory=1
# On Linux and Solaris systems, setting this parameter locks data node
# processes into memory. Doing so prevents them from swapping to disk,
# which can severely degrade cluster performance.
DataMemory=3072M
IndexMemory=384M
# The values provided for DataMemory and IndexMemory assume 4 GB RAM
# per data node. However, for best results, you should first calculate
# the memory that would be used based on the data you actually plan to
# store (you may find the ndb_size.pl utility helpful in estimating
# this), then allow an extra 20% over the calculated values. Naturally,
# you should ensure that each data node host has at least as much
# physical memory as the sum of these two values.
# ODirect=1
```

```
# Enabling this parameter causes NDBCLUSTER to try using O_DIRECT
# writes for local checkpoints and redo logs; this can reduce load on
# CPUs. We recommend doing so when using MySQL Cluster NDB 6.2.3 or
# newer on systems running Linux kernel 2.6 or later.
NoOfFragmentLogFiles=300
DataDir=path/to/data/node/data/directory
MaxNoOfConcurrentOperations=100000
TimeBetweenGlobalCheckpoints=1000
TimeBetweenEpochs=200
DiskCheckpointSpeed=10M
DiskCheckpointSpeedInRestart=100M
RedoBuffer=32M
# MaxNoOfLocalScans=64
MaxNoOfTables=1024
MaxNofOfOrderedIndexes=256
[ndbd]
HostName=data-node-A-hostname
# Id=data-node-A-id
[ndbd]
HostName=data-node-B-hostname
# Id=data-node-B-id
# You must have an [ndbd] section for every data node in the cluster;
# each of these sections must include a HostName. Each section may
# optionally include an Id for convenience, but in most cases, it is
# sufficient to allow the cluster to allocate node IDs dynamically. If
# you do specify the node ID for a data node, it must be in the range 1
# to 48 inclusive and must be unique among all IDs specified for
# cluster nodes.
# SQL NODE / API NODE PARAMETERS
[mysqld]
# HostName=SQL-node-1-hostname
# Id=sql-node-A-id
[mysqld]
[mysqld]
# Each API or SQL node that connects to the cluster requires a [mysqld]
# or [api] section of its own. Each such section defines a connection
# “slot”; you should have at least as many of these sections in the
# config.ini file as the total number of API nodes and SQL nodes that
# you wish to have connected to the cluster at any given time. There is
# no performance or other penalty for having extra slots available in
# case you find later that you want or need more API or SQL nodes to
# connect to the cluster at the same time.
# If no HostName is specified for a given [mysqld] or [api] section,
# then any API or SQL node may use that slot to connect to the
# cluster. You may wish to use an explicit HostName for one connection slot
# to guarantee that an API or SQL node from that host can always
# connect to the cluster. If you wish to prevent API or SQL nodes from
# connecting from other than a desired host or hosts, then use a
# HostName for every [mysqld] or [api] section in the config.ini file.
# You can if you wish define a node ID (Id parameter) for any API or
# SQL node, but this is not necessary; if you do so, it must be in the
# range 1 to 255 inclusive and must be unique among all IDs specified
# for cluster nodes.
```

Starting configuration for MySQL Cluster NDB 6.3. The following is a recommended starting point for configuring a cluster running MySQL Cluster NDB 6.3. It is similar to the recommendation for MySQL Cluster NDB 6.2, with the addition of parameters for better control of NDBCLUSTER process threads.

```
# TCP PARAMETERS
[tcp default]
SendBufferMemory=2M
ReceiveBufferMemory=2M
# Increasing the sizes of these 2 buffers beyond the default values
# helps prevent bottlenecks due to slow disk I/O.
# MANAGEMENT NODE PARAMETERS
[ndb_mgmd default]
DataDir=path/to/management/server/data/directory
# It is possible to use a different data directory for each management
# server, but for ease of administration it is preferable to be
# consistent.
[ndb_mgmd]
HostName=management-server-1-hostname
# Id=management-server-A-id
[ndb_mgmd]
HostName=management-server-2-hostname
# Using 2 management servers helps guarantee that there is always an
# arbitrator in the event of network partitioning, and so is
# recommended for high availability. Each management server must be
# identified by a HostName. You may for the sake of convenience specify
# a node ID for any management server, although one will be allocated
# for it automatically; if you do so, it must be in the range 1-255
# inclusive and must be unique among all IDs specified for cluster
# nodes.
# DATA NODE PARAMETERS
[ndbd default]
NoOfReplicas=2
# This parameter has no default value; it must always must be set
# explicitly. Using 2 replicas is recommended to guarantee
# availability of data; using only 1 replica does not provide any
# redundancy, which means that the failure of a single data node causes
# the entire cluster to shut down. We do not recommend using more than
# 2 replicas, since 2 is sufficient to provide high availability, and
# we do not currently test with greater values for this parameter.
LockPagesInMainMemory=1
# On Linux and Solaris systems, setting this parameter locks data node
```

```
# processes into memory. Doing so prevents them from swapping to disk,
# which can severely degrade cluster performance.
DataMemory=3072M
IndexMemory=384M
# The values provided for DataMemory and IndexMemory assume 4 GB RAM
# per data node. However, for best results, you should first calculate
# the memory that would be used based on the data you actually plan to
# store (you may find the ndb_size.pl utility helpful in estimating
# this), then allow an extra 20% over the calculated values. Naturally,
# you should ensure that each data node host has at least as much
# physical memory as the sum of these two values.
# ODirect=1
# Enabling this parameter causes NDBCLUSTER to try using O_DIRECT
# writes for local checkpoints and redo logs; this can reduce load on
# CPUs. We recommend doing so when using MySQL Cluster NDB 6.2.3 or
# newer on systems running Linux kernel 2.6 or later.
NoOfFragmentLogFiles=300
DataDir=path/to/data/node/data/directory
MaxNoOfConcurrentOperations=100000
SchedulerSpinTimer=400
SchedulerExecutionTimer=100
RealTimeScheduler=1
# Setting these parameters allows you to take advantage of real-time scheduling
# of NDBCLUSTER threads (introduced in MySQL Cluster NDB 6.3.4) to get higher
# throughput.
TimeBetweenGlobalCheckpoints=1000
TimeBetweenEpochs=200
DiskCheckpointSpeed=10M
DiskCheckpointSpeedInRestart=100M
RedoBuffer=32M
# CompressedLCP=1
# CompressedBackup=1
# Enabling CompressedLCP and CompressedBackup causes, respectively, local
checkpoint files and backup files to be compressed, which can result in a space
savings of up to 50% over non-compressed LCPs and backups.
# MaxNoOfLocalScans=64
MaxNoOfTables=1024
MaxNofOfOrderedIndexes=256
[ndbd]
HostName=data-node-A-hostname
# Id=data-node-A-id
LockExecuteThreadToCPU=1
LockMaintThreadsToCPU=0
# On systems with multiple CPUs, these parameters can be used to lock NDBCLUSTER
# threads to specific CPUs
[ndbd]
HostName=data-node-B-hostname
# Id=data-node-B-id
LockExecuteThreadToCPU=1
LockMaintThreadsToCPU=0
# You must have an [ndbd] section for every data node in the cluster;
# each of these sections must include a HostName. Each section may
# optionally include an Id for convenience, but in most cases, it is
# sufficient to allow the cluster to allocate node IDs dynamically. If
# you do specify the node ID for a data node, it must be in the range 1
# to 48 inclusive and must be unique among all IDs specified for
# cluster nodes.
# SQL NODE / API NODE PARAMETERS
[mysqld]
# HostName=SQL-node-1-hostname
# Id=sql-node-A-id
[mysqld]
[mysqld]
# Each API or SQL node that connects to the cluster requires a [mysqld]
# or [api] section of its own. Each such section defines a connection
# "slot"; you should have at least as many of these sections in the
# config.ini file as the total number of API nodes and SQL nodes that
# you wish to have connected to the cluster at any given time. There is
# no performance or other penalty for having extra slots available in
# case you find later that you want or need more API or SQL nodes to
# connect to the cluster at the same time.
# If no HostName is specified for a given [mysqld] or [api] section,
# then any API or SQL node may use that slot to connect to the
# cluster. You may wish to use an explicit HostName for one connection slot
# to guarantee that an API or SQL node from that host can always
# connect to the cluster. If you wish to prevent API or SQL nodes from
# connecting from other than a desired host or hosts, then use a
# HostName for every [mysqld] or [api] section in the config.ini file.
# You can if you wish define a node ID (Id parameter) for any API or
# SQL node, but this is not necessary; if you do so, it must be in the
# range 1 to 255 inclusive and must be unique among all IDs specified
# for cluster nodes.
```

Recommended `my.cnf` options for SQL nodes. MySQL Servers acting as MySQL Cluster SQL nodes must always be started with the `--ndbcluster` and `--ndb-connectstring` options, either on the command line or in `my.cnf`. In addition, we recommend setting the following options for all `mysqld` processes in the cluster, unless your setup requires otherwise:

- `--ndb-use-exact-count=0`
- `--ndb-index-stat-enable=0`

- `--ndb-force-send=1`
- `--engine-condition-pushdown=1`

3.4.3. The MySQL Cluster Connectstring

With the exception of the MySQL Cluster management server (`ndb_mgmd`), each node that is part of a MySQL Cluster requires a *connectstring* that points to the management server's location. This connectstring is used in establishing a connection to the management server as well as in performing other tasks depending on the node's role in the cluster. The syntax for a connectstring is as follows:

```
[nodeid=node_id, ]host-definition[, host-definition[, ...]]
host-definition:
    host_name[:port_number]
```

`node_id` is an integer larger than 1 which identifies a node in `config.ini`. *`host_name`* is a string representing a valid Internet host name or IP address. *`port_number`* is an integer referring to a TCP/IP port number.

```
example 1 (long):    "nodeid=2,myhost1:1100,myhost2:1100,192.168.0.3:1200"
example 2 (short):   "myhost1"
```

`localhost:1186` is used as the default connectstring value if none is provided. If *`port_num`* is omitted from the connectstring, the default port is 1186. This port should always be available on the network because it has been assigned by IANA for this purpose (see http://www.iana.org/assignments/port-numbers for details).

By listing multiple host definitions, it is possible to designate several redundant management servers. A MySQL Cluster data or API node attempts to contact successive management servers on each host in the order specified, until a successful connection has been established.

Beginning with MySQL Cluster NDB 6.3.19, it is also possible in a connectstring to specify one or more bind addresses to be used by nodes having multiple network interfaces for connecting to management servers. A bind address consists of a hostname or network address and an optional port number. This enhanced syntax for connectstrings is shown here:

```
[nodeid=node_id, ]
    [bind-address=host-definition, ]
    host-definition[; bind-address=host-definition]
    host-definition[; bind-address=host-definition]
    [, ...]]
host-definition:
    host_name[:port_number]
```

If a single bind address is used in the connectstring *prior* to specifying any management hosts, then this address is used as the default for connecting to any of them (unless overridden for a given management server; see later in this section for an example). For example, the following connectstring causes the node to use `192.168.178.242` regardless of the management server to which it connects:

```
bind-address=192.168.178.242, poseidon:1186, perch:1186
```

If a bind address is specified *following* a management host definition, then it is used only for connecting to that management node. Consider the following connectstring:

```
poseidon:1186;bind-address=localhost, perch:1186;bind-address=192.168.178.242
```

In this case, the node uses `localhost` to connect to the management server running on the host named `poseidon` and `192.168.178.242` to connect to the management server running on the host named `perch`.

You can specify a default bind address and then override this default for one or more specific management hosts. In the following example, `localhost` is used for connecting to the management server running on host `poseidon`; since `192.168.178.242` is specified first (before any management server definitions), it is the default bind address and so is used for connecting to the management servers on hosts `perch` and `orca`:

```
bind-address=192.168.178.242,poseidon:1186;bind-address=localhost,perch:1186,orca:2200
```

There are a number of different ways to specify the connectstring:

- Each executable has its own command-line option which enables specifying the management server at startup. (See the documentation for the respective executable.)

- It is also possible to set the connectstring for all nodes in the cluster at once by placing it in a `[mysql_cluster]` section in the management server's `my.cnf` file.
- For backward compatibility, two other options are available, using the same syntax:

  1. Set the `NDB_CONNECTSTRING` environment variable to contain the connectstring.
  2. Write the connectstring for each executable into a text file named `Ndb.cfg` and place this file in the executable's startup directory.

  However, these are now deprecated and should not be used for new installations.

The recommended method for specifying the connectstring is to set it on the command line or in the `my.cnf` file for each executable.

The maximum length of a connectstring is 1024 characters.

3.4.4. Defining Computers in a MySQL Cluster

The `[computer]` section has no real significance other than serving as a way to avoid the need of defining host names for each node in the system. All parameters mentioned here are required.

- `Id`

  This is an integer value, used to refer to the host computer elsewhere in the configuration file. This is not the same as the node ID.

- `HostName`

  This is the computer's hostname or IP address.

3.4.5. Defining a MySQL Cluster Management Server

The `[ndb_mgmd]` section is used to configure the behavior of the management server. `[mgm]` can be used as an alias; the two section names are equivalent. All parameters in the following list are optional and assume their default values if omitted.

> **Note**
>
> If neither the `ExecuteOnComputer` nor the `HostName` parameter is present, the default value `localhost` will be assumed for both.

- `Id`

  Each node in the cluster has a unique identity. For a management node, this is represented by an integer value in the range 1 to 63 inclusive (previous to MySQL Cluster NDB 6.1.1), or in the range 1 to 255 inclusive (MySQL Cluster NDB 6.1.1 and later). This ID is used by all internal cluster messages for addressing the node, and so must be unique for each MySQL Cluster node, regardless of the type of node.

  > **Note**
  >
  > Data node IDs must be less than 49, regardless of the MySQL Cluster version used. If you plan to deploy a large number of data nodes, it is a good idea to limit the node IDs for management nodes (and API nodes) to values greater than 48.

- `ExecuteOnComputer`

  This refers to the `Id` set for one of the computers defined in a `[computer]` section of the `config.ini` file.

- `PortNumber`

  This is the port number on which the management server listens for configuration requests and management commands.

- `HostName`

Specifying this parameter defines the hostname of the computer on which the management node is to reside. To specify a hostname other than `localhost`, either this parameter or `ExecuteOnComputer` is required.

- `LogDestination`

  This parameter specifies where to send cluster logging information. There are three options in this regard — `CONSOLE`, `SYSLOG`, and `FILE` — with `FILE` being the default:

  - `CONSOLE` outputs the log to `stdout`:

    ```
    CONSOLE
    ```

  - `SYSLOG` sends the log to a `syslog` facility, possible values being one of `auth`, `authpriv`, `cron`, `daemon`, `ftp`, `kern`, `lpr`, `mail`, `news`, `syslog`, `user`, `uucp`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6`, or `local7`.

    Note

    Not every facility is necessarily supported by every operating system.

    ```
    SYSLOG:facility=syslog
    ```

  - `FILE` pipes the cluster log output to a regular file on the same machine. The following values can be specified:

    - `filename`: The name of the log file.
    - `maxsize`: The maximum size (in bytes) to which the file can grow before logging rolls over to a new file. When this occurs, the old log file is renamed by appending `.N` to the file name, where *`N`* is the next number not yet used with this name.
    - `maxfiles`: The maximum number of log files.

    ```
    FILE:filename=cluster.log,maxsize=1000000,maxfiles=6
    ```

    The default value for the `FILE` parameter is `FILE:filename=ndb_node_id_cluster.log,maxsize=1000000,maxfiles=6`, where *`node_id`* is the ID of the node.

  It is possible to specify multiple log destinations separated by semicolons as shown here:

  ```
  CONSOLE;SYSLOG:facility=local0;FILE:filename=/var/log/mgmd
  ```

- `ArbitrationRank`

  This parameter is used to define which nodes can act as arbitrators. Only management nodes and SQL nodes can be arbitrators. `ArbitrationRank` can take one of the following values:

  - `0`: The node will never be used as an arbitrator.
  - `1`: The node has high priority; that is, it will be preferred as an arbitrator over low-priority nodes.
  - `2`: Indicates a low-priority node which be used as an arbitrator only if a node with a higher priority is not available for that purpose.

  Normally, the management server should be configured as an arbitrator by setting its `ArbitrationRank` to 1 (the default value) and that of all SQL nodes to 0.

  Beginning with MySQL 5.1.16 and MySQL Cluster NDB 6.1.3, it is possible to disable arbitration completely by setting `ArbitrationRank` to 0 on all management and SQL nodes.

- `ArbitrationDelay`

  An integer value which causes the management server's responses to arbitration requests to be delayed by that number of milliseconds. By default, this value is 0; it is normally not necessary to change it.

-

`DataDir`

This specifies the directory where output files from the management server will be placed. These files include cluster log files, process output files, and the daemon's process ID (PID) file. (For log files, this location can be overridden by setting the `FILE` parameter for `LogDestination` as discussed previously in this section.)

The default value for this parameter is the directory in which `ndb_mgmd` is located.

- `TotalSendBufferMemory`

  This parameter is available beginning with MySQL Cluster NDB 6.4.0. It is used to determine the total amount of memory to allocate on this node for shared send buffer memory among all configured transporters.

  If this parameter is set, its minimum allowed value is 256K; the maxmimum is 4294967039. For more detailed information about the behavior and use of `TotalSendBufferMemory` and configuring send buffer memory parameters in MySQL Cluster NDB 6.4.0 and later, see Section 3.7, "Configuring MySQL Cluster Send Buffer Parameters".

> **Note**
>
> After making changes in a management node's configuration, it is necessary to perform a rolling restart of the cluster in order for the new configuration to take effect.
>
> To add new management servers to a running MySQL Cluster, it is also necessary to perform a rolling restart of all cluster nodes after modifying any existing `config.ini` files. For more information about issues arising when using multiple management nodes, see Section 12.10, "Limitations Relating to Multiple MySQL Cluster Nodes".

3.4.6. Defining MySQL Cluster Data Nodes

The `[ndbd]` and `[ndbd default]` sections are used to configure the behavior of the cluster's data nodes.

`[ndbd]` and `[ndbd default]` are always used as the section names whether you are using `ndbd` or (in MySQL Cluster NDB 6.4.0 and later) `ndbmtd` binaries for the data node processes.

There are many parameters which control buffer sizes, pool sizes, timeouts, and so forth. The only mandatory parameters are:

- Either `ExecuteOnComputer` or `HostName`, which must be defined in the local `[ndbd]` section.
- The parameter `NoOfReplicas`, which must be defined in the`[ndbd default]`section, as it is common to all Cluster data nodes.

Most data node parameters are set in the `[ndbd default]` section. Only those parameters explicitly stated as being able to set local values are allowed to be changed in the `[ndbd]` section. Where present, `HostName`, `Id` and `ExecuteOnComputer` *must* be defined in the local `[ndbd]` section, and not in any other section of `config.ini`. In other words, settings for these parameters are specific to one data node.

For those parameters affecting memory usage or buffer sizes, it is possible to use `K`, `M`, or `G` as a suffix to indicate units of 1024, 1024×1024, or 1024×1024×1024. (For example, `100K` means 100 × 1024 = 102400.) Parameter names and values are currently case-sensitive.

Information about configuration parameters specific to MySQL Cluster Disk Data tables can be found later in this section.

Beginning with MySQL Cluster NDB 6.4.0, all of these parameters also apply to `ndbmtd` (the multi-threaded version of `ndbd`). An additional data node configuration parameter `MaxNoOfExecutionThreads` applies to `ndbmtd` only, and has no effect when used with `ndbd`. For a description of this parameter and its use, see Section 6.3, "`ndbmtd` — The MySQL Cluster Data Node Daemon (Multi-Threaded)".

Identifying data nodes. The `Id` value (that is, the data node identifier) can be allocated on the command line when the node is started or in the configuration file.

- `Id`

  This is the node ID used as the address of the node for all cluster internal messages. For data nodes, this is an integer in the range 1 to 48 inclusive. Each node in the cluster must have a unique identifier.

- `ExecuteOnComputer`

  This refers to the `Id` set for one of the computers defined in a `[computer]` section.

- `HostName`

  Specifying this parameter defines the hostname of the computer on which the data node is to reside. To specify a hostname other than `localhost`, either this parameter or `ExecuteOnComputer` is required.

- `ServerPort` (*OBSOLETE*)

  Each node in the cluster uses a port to connect to other nodes. This port is used also for non-TCP transporters in the connection setup phase. The default port is allocated dynamically in such a way as to ensure that no two nodes on the same computer receive the same port number, so it should not normally be necessary to specify a value for this parameter.

- `TcpBind_INADDR_ANY`

  Setting this parameter to `TRUE` or `1` binds `IP_ADDR_ANY` so that connections can be made from anywhere (for autogenerated connections). The default is `FALSE` (`0`).

  This parameter was added in MySQL Cluster NDB 6.2.0.

- `NodeGroup`

  This parameter can be used to assign a data node to a specific node group. It is read only when the cluster is started for the first time, and cannot be used to reassign a data node to a different node group online. It is generally not desirable to use this parameter in the `[ndbd default]` section of the `config.ini` file, and care must be taken not to assign nodes to node groups in such a way that an invalid numbers of nodes are assigned to any node groups.

  The `NodeGroup` parameter is chiefly intended for use in adding a new node group to a running MySQL Cluster without having to perform a rolling restart. For this purpose, you should set it to 65535 (the maximum value). You are not required to set a `NodeGroup` value for all cluster data nodes, only for those nodes which are to be started and added to the cluster as a new node group at a later time. For more information, see Section 7.8.3, “Adding MySQL Cluster Data Nodes Online: Detailed Example”.

  This parameter was added in MySQL Cluster NDB 6.4.0.

- `NoOfReplicas`

  This global parameter can be set only in the `[ndbd default]` section, and defines the number of replicas for each table stored in the cluster. This parameter also specifies the size of node groups. A node group is a set of nodes all storing the same information.

  Node groups are formed implicitly. The first node group is formed by the set of data nodes with the lowest node IDs, the next node group by the set of the next lowest node identities, and so on. By way of example, assume that we have 4 data nodes and that `NoOfReplicas` is set to 2. The four data nodes have node IDs 2, 3, 4 and 5. Then the first node group is formed from nodes 2 and 3, and the second node group by nodes 4 and 5. It is important to configure the cluster in such a manner that nodes in the same node groups are not placed on the same computer because a single hardware failure would cause the entire cluster to fail.

  If no node IDs are provided, the order of the data nodes will be the determining factor for the node group. Whether or not explicit assignments are made, they can be viewed in the output of the management client's `SHOW` command.

  There is no default value for `NoOfReplicas`; the maximum possible value is 4. Currently, only the values 1 and 2 are actually supported (see Bug#18621).

  > **Important**
  >
  > Setting `NoOfReplicas` to 1 means that there is only a single copy of all Cluster data; in this case, the loss of a single data node causes the cluster to fail because there are no additional copies of the data stored by that node.

  The value for this parameter must divide evenly into the number of data nodes in the cluster. For example, if there are two data nodes, then `NoOfReplicas` must be equal to either 1 or 2, since 2/3 and 2/4 both yield fractional values; if there are four data nodes, then `NoOfReplicas` must be equal to 1, 2, or 4.

-

`DataDir`

This parameter specifies the directory where trace files, log files, pid files and error logs are placed.

The default is the data node process working directory.

- `FileSystemPath`

  This parameter specifies the directory where all files created for metadata, REDO logs, UNDO logs (for Disk Data tables) and data files are placed. The default is the directory specified by `DataDir`.

  Note

  This directory must exist before the `ndbd` process is initiated.

  The recommended directory hierarchy for MySQL Cluster includes `/var/lib/mysql-cluster`, under which a directory for the node's file system is created. The name of this subdirectory contains the node ID. For example, if the node ID is 2, this subdirectory is named `ndb_2_fs`.

- `BackupDataDir`

  This parameter specifies the directory in which backups are placed. If omitted, the default backup location is the directory named `BACKUP` under the location specified by the `FileSystemPath` parameter. (See above.)

Data Memory, Index Memory, and String Memory

`DataMemory` and `IndexMemory` are `[ndbd]` parameters specifying the size of memory segments used to store the actual records and their indexes. In setting values for these, it is important to understand how `DataMemory` and `IndexMemory` are used, as they usually need to be updated to reflect actual usage by the cluster:

- `DataMemory`

  This parameter defines the amount of space (in bytes) available for storing database records. The entire amount specified by this value is allocated in memory, so it is extremely important that the machine has sufficient physical memory to accommodate it.

  The memory allocated by `DataMemory` is used to store both the actual records and indexes. There is a 16-byte overhead on each record; an additional amount for each record is incurred because it is stored in a 32KB page with 128 byte page overhead (see below). There is also a small amount wasted per page due to the fact that each record is stored in only one page.

  For variable-size table attributes in MySQL 5.1, the data is stored on separate datapages, allocated from `DataMemory`. Variable-length records use a fixed-size part with an extra overhead of 4 bytes to reference the variable-size part. The variable-size part has 2 bytes overhead plus 2 bytes per attribute.

  The maximum record size is currently 8052 bytes.

  The memory space defined by `DataMemory` is also used to store ordered indexes, which use about 10 bytes per record. Each table row is represented in the ordered index. A common error among users is to assume that all indexes are stored in the memory allocated by `IndexMemory`, but this is not the case: Only primary key and unique hash indexes use this memory; ordered indexes use the memory allocated by `DataMemory`. However, creating a primary key or unique hash index also creates an ordered index on the same keys, unless you specify `USING HASH` in the index creation statement. This can be verified by running `ndb_desc -d db_name table_name` in the management client.

  The memory space allocated by `DataMemory` consists of 32KB pages, which are allocated to table fragments. Each table is normally partitioned into the same number of fragments as there are data nodes in the cluster. Thus, for each node, there are the same number of fragments as are set in `NoOfReplicas`.

  In addition, due to the way in which new pages are allocated when the capacity of the current page is exhausted, there is an additional overhead of approximately 18.75%. When more `DataMemory` is required, more than one new page is allocated, according to the following formula:

  ```
  number of new pages = FLOOR(number of current pages × 0.1875) + 1
  ```

  For example, if 15 pages are currently allocated to a given table and an insert to this table requires additional storage space, the number of new pages allocated to the table is `FLOOR(15 × 0.1875) + 1 = FLOOR(2.8125) + 1 = 2 + 1 = 3`. Now 15 + 3 = 18 memory pages are allocated to the table. When the last of these 18 pages becomes full, `FLOOR(18 ×`

`0.1875) + 1 = FLOOR(3.3750) + 1 = 3 + 1 = 4` new pages are allocated, so the total number of pages allocated to the table is now 22.

> **Note**
>
> The “18.75% + 1” overhead is no longer required beginning with MySQL Cluster NDB 6.2.3 and MySQL Cluster NDB 6.3.0.

Once a page has been allocated, it is currently not possible to return it to the pool of free pages, except by deleting the table. (This also means that `DataMemory` pages, once allocated to a given table, cannot be used by other tables.) Performing a node recovery also compresses the partition because all records are inserted into empty partitions from other live nodes.

The `DataMemory` memory space also contains UNDO information: For each update, a copy of the unaltered record is allocated in the `DataMemory`. There is also a reference to each copy in the ordered table indexes. Unique hash indexes are updated only when the unique index columns are updated, in which case a new entry in the index table is inserted and the old entry is deleted upon commit. For this reason, it is also necessary to allocate enough memory to handle the largest transactions performed by applications using the cluster. In any case, performing a few large transactions holds no advantage over using many smaller ones, for the following reasons:

- Large transactions are not any faster than smaller ones
- Large transactions increase the number of operations that are lost and must be repeated in event of transaction failure
- Large transactions use more memory

The default value for `DataMemory` is 80MB; the minimum is 1MB. There is no maximum size, but in reality the maximum size has to be adapted so that the process does not start swapping when the limit is reached. This limit is determined by the amount of physical RAM available on the machine and by the amount of memory that the operating system may commit to any one process. 32-bit operating systems are generally limited to 2–4GB per process; 64-bit operating systems can use more. For large databases, it may be preferable to use a 64-bit operating system for this reason.

- `IndexMemory`

  This parameter controls the amount of storage used for hash indexes in MySQL Cluster. Hash indexes are always used for primary key indexes, unique indexes, and unique constraints. Note that when defining a primary key and a unique index, two indexes will be created, one of which is a hash index used for all tuple accesses as well as lock handling. It is also used to enforce unique constraints.

  The size of the hash index is 25 bytes per record, plus the size of the primary key. For primary keys larger than 32 bytes another 8 bytes is added.

  The default value for `IndexMemory` is 18MB. The minimum is 1MB.

- `StringMemory`

  This parameter determines how much memory is allocated for strings such as table names, and is specified in an `[ndbd]` or `[ndbd default]` section of the `config.ini` file. A value between `0` and `100` inclusive is interpreted as a percent of the maximum default value, which is calculated based on a number of factors including the number of tables, maximum table name size, maximum size of `.FRM` files, `MaxNoOfTriggers`, maximum column name size, and maximum default column value. In general it is safe to assume that the maximum default value is approximately 5 MB for a MySQL Cluster having 1000 tables.

  A value greater than `100` is interpreted as a number of bytes.

  The default value is `5` — that is, 5 percent of the default maximum, or roughly 5 KB. (Note that this is a change from previous versions of MySQL Cluster.)

  Under most circumstances, the default value should be sufficient, but when you have a great many Cluster tables (1000 or more), it is possible to get Error 773 `Out of string memory, please modify StringMemory config parameter: Permanent error: Schema error`, in which case you should increase this value. `25` (25 percent) is not excessive, and should prevent this error from recurring in all but the most extreme conditions.

The following example illustrates how memory is used for a table. Consider this table definition:

```
CREATE TABLE example (
  a INT NOT NULL,
  b INT NOT NULL,
  c INT NOT NULL,
  PRIMARY KEY(a),
  UNIQUE(b)
) ENGINE=NDBCLUSTER;
```

For each record, there are 12 bytes of data plus 12 bytes overhead. Having no nullable columns saves 4 bytes of overhead. In addition, we have two ordered indexes on columns `a` and `b` consuming roughly 10 bytes each per record. There is a primary key hash index on the base table using roughly 29 bytes per record. The unique constraint is implemented by a separate table with `b` as primary key and `a` as a column. This other table consumes an additional 29 bytes of index memory per record in the `example` table as well 8 bytes of record data plus 12 bytes of overhead.

Thus, for one million records, we need 58MB for index memory to handle the hash indexes for the primary key and the unique constraint. We also need 64MB for the records of the base table and the unique index table, plus the two ordered index tables.

You can see that hash indexes takes up a fair amount of memory space; however, they provide very fast access to the data in return. They are also used in MySQL Cluster to handle uniqueness constraints.

Currently, the only partitioning algorithm is hashing and ordered indexes are local to each node. Thus, ordered indexes cannot be used to handle uniqueness constraints in the general case.

An important point for both `IndexMemory` and `DataMemory` is that the total database size is the sum of all data memory and all index memory for each node group. Each node group is used to store replicated information, so if there are four nodes with two replicas, there will be two node groups. Thus, the total data memory available is 2 × `DataMemory` for each data node.

It is highly recommended that `DataMemory` and `IndexMemory` be set to the same values for all nodes. Data distribution is even over all nodes in the cluster, so the maximum amount of space available for any node can be no greater than that of the smallest node in the cluster.

`DataMemory` and `IndexMemory` can be changed, but decreasing either of these can be risky; doing so can easily lead to a node or even an entire MySQL Cluster that is unable to restart due to there being insufficient memory space. Increasing these values should be acceptable, but it is recommended that such upgrades are performed in the same manner as a software upgrade, beginning with an update of the configuration file, and then restarting the management server followed by restarting each data node in turn.

Updates do not increase the amount of index memory used. Inserts take effect immediately; however, rows are not actually deleted until the transaction is committed.

Transaction parameters. The next three `[ndbd]` parameters that we discuss are important because they affect the number of parallel transactions and the sizes of transactions that can be handled by the system. `MaxNoOfConcurrentTransactions` sets the number of parallel transactions possible in a node. `MaxNoOfConcurrentOperations` sets the number of records that can be in update phase or locked simultaneously.

Both of these parameters (especially `MaxNoOfConcurrentOperations`) are likely targets for users setting specific values and not using the default value. The default value is set for systems using small transactions, to ensure that these do not use excessive memory.

- `MaxNoOfConcurrentTransactions`

  Each cluster data node requires a transaction record for each active transaction in the cluster. The task of coordinating transactions is distributed among all of the data nodes. The total number of transaction records in the cluster is the number of transactions in any given node times the number of nodes in the cluster.

  Transaction records are allocated to individual MySQL servers. Each connection to a MySQL server requires at least one transaction record, plus an additional transaction object per table accessed by that connection. This means that a reasonable minimum for this parameter is

  ```
  MaxNoOfConcurrentTransactions =
      (maximum number of tables accessed in any single transaction + 1)
      * number of cluster SQL nodes
  ```

  For example, suppose that there are 4 SQL nodes using the cluster. A single join involving 5 tables requires 6 transaction records; if there are 5 such joins in a transaction, then 5 * 6 = 30 transaction records are required for this transaction, per MySQL server, or 30 * 4 = 120 transaction records total.

  This parameter must be set to the same value for all cluster data nodes. This is due to the fact that, when a data node fails, the oldest surviving node re-creates the transaction state of all transactions that were ongoing in the failed node.

  Changing the value of `MaxNoOfConcurrentTransactions` requires a complete shutdown and restart of the cluster.

  The default value is 4096.

- `MaxNoOfConcurrentOperations`

It is a good idea to adjust the value of this parameter according to the size and number of transactions. When performing transactions of only a few operations each and not involving a great many records, there is no need to set this parameter very high. When performing large transactions involving many records need to set this parameter higher.

Records are kept for each transaction updating cluster data, both in the transaction coordinator and in the nodes where the actual updates are performed. These records contain state information needed to find UNDO records for rollback, lock queues, and other purposes.

This parameter should be set to the number of records to be updated simultaneously in transactions, divided by the number of cluster data nodes. For example, in a cluster which has four data nodes and which is expected to handle 1,000,000 concurrent updates using transactions, you should set this value to 1000000 / 4 = 250000.

Read queries which set locks also cause operation records to be created. Some extra space is allocated within individual nodes to accommodate cases where the distribution is not perfect over the nodes.

When queries make use of the unique hash index, there are actually two operation records used per record in the transaction. The first record represents the read in the index table and the second handles the operation on the base table.

The default value is 32768.

This parameter actually handles two values that can be configured separately. The first of these specifies how many operation records are to be placed with the transaction coordinator. The second part specifies how many operation records are to be local to the database.

A very large transaction performed on an eight-node cluster requires as many operation records in the transaction coordinator as there are reads, updates, and deletes involved in the transaction. However, the operation records of the are spread over all eight nodes. Thus, if it is necessary to configure the system for one very large transaction, it is a good idea to configure the two parts separately. `MaxNoOfConcurrentOperations` will always be used to calculate the number of operation records in the transaction coordinator portion of the node.

It is also important to have an idea of the memory requirements for operation records. These consume about 1KB per record.

- `MaxNoOfLocalOperations`

  By default, this parameter is calculated as 1.1 × `MaxNoOfConcurrentOperations`. This fits systems with many simultaneous transactions, none of them being very large. If there is a need to handle one very large transaction at a time and there are many nodes, it is a good idea to override the default value by explicitly specifying this parameter.

Transaction temporary storage. The next set of `[ndbd]` parameters is used to determine temporary storage when executing a statement that is part of a Cluster transaction. All records are released when the statement is completed and the cluster is waiting for the commit or rollback.

The default values for these parameters are adequate for most situations. However, users with a need to support transactions involving large numbers of rows or operations may need to increase these values to enable better parallelism in the system, whereas users whose applications require relatively small transactions can decrease the values to save memory.

- `MaxNoOfConcurrentIndexOperations`

  For queries using a unique hash index, another temporary set of operation records is used during a query's execution phase. This parameter sets the size of that pool of records. Thus, this record is allocated only while executing a part of a query. As soon as this part has been executed, the record is released. The state needed to handle aborts and commits is handled by the normal operation records, where the pool size is set by the parameter `MaxNoOfConcurrentOperations`.

  The default value of this parameter is 8192. Only in rare cases of extremely high parallelism using unique hash indexes should it be necessary to increase this value. Using a smaller value is possible and can save memory if the DBA is certain that a high degree of parallelism is not required for the cluster.

- `MaxNoOfFiredTriggers`

  The default value of `MaxNoOfFiredTriggers` is 4000, which is sufficient for most situations. In some cases it can even be decreased if the DBA feels certain the need for parallelism in the cluster is not high.

  A record is created when an operation is performed that affects a unique hash index. Inserting or deleting a record in a table with unique hash indexes or updating a column that is part of a unique hash index fires an insert or a delete in the index table. The resulting record is used to represent this index table operation while waiting for the original operation that fired it to com-

plete. This operation is short-lived but can still require a large number of records in its pool for situations with many parallel write operations on a base table containing a set of unique hash indexes.

- `TransactionBufferMemory`

  The memory affected by this parameter is used for tracking operations fired when updating index tables and reading unique indexes. This memory is used to store the key and column information for these operations. It is only very rarely that the value for this parameter needs to be altered from the default.

  The default value for `TransactionBufferMemory` is 1MB.

  Normal read and write operations use a similar buffer, whose usage is even more short-lived. The compile-time parameter `ZATTRBUF_FILESIZE` (found in `ndb/src/kernel/blocks/Dbtc/Dbtc.hpp`) set to 4000 × 128 bytes (500KB). A similar buffer for key information, `ZDATABUF_FILESIZE` (also in `Dbtc.hpp`) contains 4000 × 16 = 62.5KB of buffer space. `Dbtc` is the module that handles transaction coordination.

Scans and buffering. There are additional `[ndbd]` parameters in the `Dblqh` module (in `ndb/src/kernel/blocks/Dblqh/Dblqh.hpp`) that affect reads and updates. These include `ZATTRINBUF_FILESIZE`, set by default to 10000 × 128 bytes (1250KB) and `ZDATABUF_FILE_SIZE`, set by default to 10000*16 bytes (roughly 156KB) of buffer space. To date, there have been neither any reports from users nor any results from our own extensive tests suggesting that either of these compile-time limits should be increased.

- `MaxNoOfConcurrentScans`

  This parameter is used to control the number of parallel scans that can be performed in the cluster. Each transaction coordinator can handle the number of parallel scans defined for this parameter. Each scan query is performed by scanning all partitions in parallel. Each partition scan uses a scan record in the node where the partition is located, the number of records being the value of this parameter times the number of nodes. The cluster should be able to sustain `MaxNoOfConcurrentScans` scans concurrently from all nodes in the cluster.

  Scans are actually performed in two cases. The first of these cases occurs when no hash or ordered indexes exists to handle the query, in which case the query is executed by performing a full table scan. The second case is encountered when there is no hash index to support the query but there is an ordered index. Using the ordered index means executing a parallel range scan. The order is kept on the local partitions only, so it is necessary to perform the index scan on all partitions.

  The default value of `MaxNoOfConcurrentScans` is 256. The maximum value is 500.

- `MaxNoOfLocalScans`

  Specifies the number of local scan records if many scans are not fully parallelized. If the number of local scan records is not provided, it is calculated as the product of `MaxNoOfConcurrentScans` and the number of data nodes in the system. The minimum value is 32.

- `BatchSizePerLocalScan`

  This parameter is used to calculate the number of lock records used to handle concurrent scan operations.

  The default value is 64; this value has a strong connection to the `ScanBatchSize` defined in the SQL nodes.

- `LongMessageBuffer`

  This is an internal buffer used for passing messages within individual nodes and between nodes. Although it is highly unlikely that this would need to be changed, it is configurable. In MySQL Cluster NDB 6.4.3 and earlier, the default is 1MB; beginning with MySQL Cluster NDB 7.0.4, it is 4MB.

Memory Allocation

`MaxAllocate`

This is the maximum size of the memory unit to use when allocating memory for tables. In cases where `NDB` gives `Out of memory` errors, but it is evident by examining the cluster logs or the output of `DUMP 1000` (see `DUMP 1000`) that all available memory has not yet been used, you can increase the value of this parameter (or `MaxNoOfTables`, or both) in order to cause `NDB` to make sufficient memory available.

This parameter was introduced in MySQL 5.1.20, MySQL Cluster NDB 6.1.12 and MySQL Cluster NDB 6.2.3.

Logging and checkpointing

The following `[ndbd]` parameters control log and checkpoint behavior.

- `NoOfFragmentLogFiles`

  This parameter sets the number of REDO log files for the node, and thus the amount of space allocated to REDO logging. Because the REDO log files are organized in a ring, it is extremely important that the first and last log files in the set (sometimes referred to as the "head" and "tail" log files, respectively) do not meet. When these approach one another too closely, the node begins aborting all transactions encompassing updates due to a lack of room for new log records.

  A `REDO` log record is not removed until the required number of local checkpoints has been completed since that log record was inserted (prior to MySQL Cluster NDB 6.3.8, this was 3 local checkpoints; in later versions of MySQL Cluster, only 2 local checkpoints are necessary). Checkpointing frequency is determined by its own set of configuration parameters discussed elsewhere in this chapter.

  How these parameters interact and proposals for how to configure them are discussed in Section 3.6, "Configuring MySQL Cluster Parameters for Local Checkpoints".

  The default parameter value is 16, which by default means 16 sets of 4 16MB files for a total of 1024MB. Beginning with MySQL Cluster NDB 6.1.1, the size of the individual log files is configurable using the `FragmentLogFileSize` parameter; more information about this parameter can be found here. In scenarios requiring a great many updates, the value for `NoOfFragmentLogFiles` may need to be set as high as 300 or even higher to provide sufficient space for REDO logs.

  If the checkpointing is slow and there are so many writes to the database that the log files are full and the log tail cannot be cut without jeopardizing recovery, all updating transactions are aborted with internal error code 410 (`Out of log file space temporarily`). This condition prevails until a checkpoint has completed and the log tail can be moved forward.

  > **Important**
  >
  > This parameter cannot be changed "on the fly"; you must restart the node using `--initial`. If you wish to change this value for all data nodes in a running cluster, you can do so via a rolling node restart (using `--initial` when starting each data node).

- `FragmentLogFileSize`

  Setting this parameter allows you to control directly the size of redo log files. This can be useful in situations when MySQL Cluster is operating under a high load and it is unable to close fragment log files quickly enough before attempting to open new ones (only 2 fragment log files can be open at one time); increasing the size of the fragment log files gives the cluster more time before having to open each new fragment log file. The default value for this parameter is 16M. `FragmentLogFileSize` was added in MySQL Cluster NDB 6.1.11.

  For more information about fragment log files, see the description of the NoOfFragmentLogFiles parameter.

- `InitFragmentLogFiles`

  By default, fragment log files are created sparsely when performing an initial start of a data node — that is, depending on the operating system and file system in use, not all bytes are necessarily written to disk. Beginning with MySQL Cluster NDB 6.3.19, it is possible to override this behavior and force all bytes to be written regardless of the platform and file system type being used by mean of this parameter.

  `InitFragmentLogFiles` takes one of two values:

  - `SPARSE`. Fragment log files are created sparsely. This is the default value.
  - `FULL`. Force all bytes of the fragment log file to be written to disk.

  Depending on your operating system and file system, setting `InitFragmentLogFiles=FULL` may help eliminate I/O errors on writes to the REDO log.

- `MaxNoOfOpenFiles`

  This parameter sets a ceiling on how many internal threads to allocate for open files. *Any situation requiring a change in this parameter should be reported as a bug*.

The default value is 0. (Prior to MySQL 5.1.16, the default was 40.) However, the minimum value to which this parameter can be set is 20.

- `InitialNoOfOpenFiles`

  This parameter sets the initial number of internal threads to allocate for open files.

  The default value is 27.

- `MaxNoOfSavedMessages`

  This parameter sets the maximum number of trace files that are kept before overwriting old ones. Trace files are generated when, for whatever reason, the node crashes.

  The default is 25 trace files.

- `MaxLCPStartDelay`

  In parallel data node recovery (supported in MySQL Cluster NDB 6.3.8 and later), only table data is actually copied and synchronized in parallel; synchronization of metadata such as dictionary and checkpoint information is done in a serial fashion. In addition, recovery of dictionary and checkpoint information cannot be executed in parallel with performing of local checkpoints. This means that, when starting or restarting many data nodes concurrently, data nodes may be forced to wait while a local checkpoint is performed, which can result in longer node recovery times.

  Beginning with MySQL Cluster NDB 6.3.23 and MySQL Cluster NDB 6.4.3, it is possible to force a delay in the local checkpoint to allow more (and possibly all) data nodes to complete metadata synchronization; once each data node's metadata synchronization is complete, all of the data nodes can recover table data in parallel, even while the local checkpoint is being executed.

  To force such a delay, you can set `MaxLCPStartDelay`, which determines the number of seconds the cluster can wait to begin a local checkpoint while data nodes continue to synchronize metadata. This parameter should be set in the `[ndbd default]` section of the `config.ini` file, so that it is the same for all data nodes. The maximum value is 600; the default is 0.

Metadata objects. The next set of `[ndbd]` parameters defines pool sizes for metadata objects, used to define the maximum number of attributes, tables, indexes, and trigger objects used by indexes, events, and replication between clusters. Note that these act merely as “suggestions” to the cluster, and any that are not specified revert to the default values shown.

- `MaxNoOfAttributes`

  Defines the number of attributes that can be defined in the cluster.

  The default value is 1000, with the minimum possible value being 32. The maximum is 4294967039. Each attribute consumes around 200 bytes of storage per node due to the fact that all metadata is fully replicated on the servers.

  When setting `MaxNoOfAttributes`, it is important to prepare in advance for any `ALTER TABLE` statements that you might want to perform in the future. This is due to the fact, during the execution of `ALTER TABLE` on a Cluster table, 3 times the number of attributes as in the original table are used, and a good practice is to allow double this amount. For example, if the MySQL Cluster table having the greatest number of attributes (*`greatest_number_of_attributes`*) has 100 attributes, a good starting point for the value of `MaxNoOfAttributes` would be `6 * greatest_number_of_attributes = 600`.

  You should also estimate the average number of attributes per table and multiply this by `MaxNoOfTables`. If this value is larger than the value obtained in the previous paragraph, you should use the larger value instead.

  Assuming that you can create all desired tables without any problems, you should also verify that this number is sufficient by trying an actual `ALTER TABLE` after configuring the parameter. If this is not successful, increase `MaxNoOfAttributes` by another multiple of `MaxNoOfTables` and test it again.

- `MaxNoOfTables`

  A table object is allocated for each table, unique hash index, and ordered index. This parameter sets the maximum number of table objects for the cluster as a whole.

  For each attribute that has a `BLOB` data type an extra table is used to store most of the `BLOB` data. These tables also must be

taken into account when defining the total number of tables.

The default value of this parameter is 128. The minimum is 8 and the maximum is 1600. Each table object consumes approximately 20KB per node.

- `MaxNoOfOrderedIndexes`

  For each ordered index in the cluster, an object is allocated describing what is being indexed and its storage segments. By default, each index so defined also defines an ordered index. Each unique index and primary key has both an ordered index and a hash index.

  The default value of this parameter is 128. Each object consumes approximately 10KB of data per node.

- `MaxNoOfUniqueHashIndexes`

  For each unique index that is not a primary key, a special table is allocated that maps the unique key to the primary key of the indexed table. By default, an ordered index is also defined for each unique index. To prevent this, you must specify the `USING HASH` option when defining the unique index.

  The default value is 64. Each index consumes approximately 15KB per node.

- `MaxNoOfTriggers`

  Internal update, insert, and delete triggers are allocated for each unique hash index. (This means that three triggers are created for each unique hash index.) However, an *ordered* index requires only a single trigger object. Backups also use three trigger objects for each normal table in the cluster.

  Replication between clusters also makes use of internal triggers.

  This parameter sets the maximum number of trigger objects in the cluster.

  The default value is 768.

- `MaxNoOfIndexes`

  This parameter is deprecated in MySQL 5.1; you should use `MaxNoOfOrderedIndexes` and `MaxNoOfUniqueHashIndexes` instead.

  This parameter is used only by unique hash indexes. There needs to be one record in this pool for each unique hash index defined in the cluster.

  The default value of this parameter is 128.

Boolean parameters. The behavior of data nodes is also affected by a set of `[ndbd]` parameters taking on boolean values. These parameters can each be specified as `TRUE` by setting them equal to `1` or `Y`, and as `FALSE` by setting them equal to `0` or `N`.

- `LockPagesInMainMemory`

  For a number of operating systems, including Solaris and Linux, it is possible to lock a process into memory and so avoid any swapping to disk. This can be used to help guarantee the cluster's real-time characteristics.

  Beginning with MySQL 5.1.15 and MySQL Cluster NDB 6.1.1, this parameter takes one of the integer values `0`, `1`, or `2`, which act as follows:

  - `0`: Disables locking. This is the default value.
  - `1`: Performs the lock after allocating memory for the process.
  - `2`: Performs the lock before memory for the process is allocated.

  Previously, this parameter was a Boolean. `0` or `false` was the default setting, and disabled locking. `1` or `true` enabled locking of the process after its memory was allocated.

  Important

Beginning with MySQL 5.1.15 and MySQL Cluster NDB 6.1.1, it is no longer possible to use `true` or `false` for the value of this parameter; when upgrading from a previous version, you must change the value to `0`, `1`, or `2`.

- `StopOnError`

  This parameter specifies whether an `ndbd` process should exit or perform an automatic restart when an error condition is encountered.

  This feature is enabled by default.

- `Diskless`

  It is possible to specify MySQL Cluster tables as *diskless*, meaning that tables are not checkpointed to disk and that no logging occurs. Such tables exist only in main memory. A consequence of using diskless tables is that neither the tables nor the records in those tables survive a crash. However, when operating in diskless mode, it is possible to run `ndbd` on a diskless computer.

  Important

  This feature causes the *entire* cluster to operate in diskless mode.

  When this feature is enabled, Cluster online backup is disabled. In addition, a partial start of the cluster is not possible.

  `Diskless` is disabled by default.

- `ODirect`

  Enabling this parameter causes `NDB` to attempt using `O_DIRECT` writes for LCP, backups, and redo logs, often lowering `kswapd` and CPU usage. When using MySQL Cluster on Linux, we recommend that you enable `ODirect` if you are using a 2.6 or kernel.

  This parameter was added in the following releases:

  - MySQL 5.1.20
  - MySQL Cluster NDB 6.1.11
  - MySQL Cluster NDB 6.2.3
  - MySQL Cluster NDB 6.3.0

  `ODirect` is disabled by default.

- `RestartOnErrorInsert`

  This feature is accessible only when building the debug version where it is possible to insert errors in the execution of individual blocks of code as part of testing.

  This feature is disabled by default.

- `CompressedBackup`

  Setting this parameter to `1` causes backup files to be compressed. The compression used is equivalent to `gzip --fast`, and can save 50% or more of the space required on the data node to store uncompressed backup files. Compressed backups can be enabled for individual data nodes, or for all data nodes (by setting this parameter in the `[ndbd default]` section of the `config.ini` file).

  Important

  You cannot restore a compressed backup to a cluster running a MySQL version that does not support this feature.

  The default value is `0` (disabled).

  This parameter was introduced in MySQL Cluster NDB 6.3.7.

- `CompressedLCP`

Setting this parameter to `1` causes local checkpoint files to be compressed. The compression used is equivalent to `gzip --fast`, and can save 50% or more of the space required on the data node to store uncompressed checkpoint files. Compressed LCPs can be enabled for individual data nodes, or for all data nodes (by setting this parameter in the `[ndbd default]` section of the `config.ini` file).

> **Important**
>
> You cannot restore a compressed local checkpoint to a cluster running a MySQL version that does not support this feature.

The default value is `0` (disabled).

This parameter was introduced in MySQL Cluster NDB 6.3.7.

Controlling Timeouts, Intervals, and Disk Paging

There are a number of `[ndbd]` parameters specifying timeouts and intervals between various actions in Cluster data nodes. Most of the timeout values are specified in milliseconds. Any exceptions to this are mentioned where applicable.

- `TimeBetweenWatchDogCheck`

  To prevent the main thread from getting stuck in an endless loop at some point, a "watchdog" thread checks the main thread. This parameter specifies the number of milliseconds between checks. If the process remains in the same state after three checks, the watchdog thread terminates it.

  This parameter can easily be changed for purposes of experimentation or to adapt to local conditions. It can be specified on a per-node basis although there seems to be little reason for doing so.

  The default timeout is 6000 milliseconds (6 seconds).

- `TimeBetweenWatchDogCheckInitial`

  This is similar to the `TimeBetweenWatchDogCheck` parameter, except that `TimeBetweenWatchDogCheckInitial` controls the amount of time that passes between execution checks inside a database node in the early start phases during which memory is allocated.

  The default timeout is 6000 milliseconds (6 seconds).

  This parameter was added in MySQL 5.1.20.

- `StartPartialTimeout`

  This parameter specifies how long the Cluster waits for all data nodes to come up before the cluster initialization routine is invoked. This timeout is used to avoid a partial Cluster startup whenever possible.

  The default value is 30000 milliseconds (30 seconds). 0 disables the timeout, in which case the cluster may start only if all nodes are available.

- `StartPartitionedTimeout`

  If the cluster is ready to start after waiting for `StartPartialTimeout` milliseconds but is still possibly in a partitioned state, the cluster waits until this timeout has also passed.

  The default timeout is 60000 milliseconds (60 seconds).

- `StartFailureTimeout`

  If a data node has not completed its startup sequence within the time specified by this parameter, the node startup fails. Setting this parameter to 0 (the default value) means that no data node timeout is applied.

  For nonzero values, this parameter is measured in milliseconds. For data nodes containing extremely large amounts of data, this parameter should be increased. For example, in the case of a data node containing several gigabytes of data, a period as long as 10–15 minutes (that is, 600000 to 1000000 milliseconds) might be required to perform a node restart.

- `HeartbeatIntervalDbDb`

  One of the primary methods of discovering failed nodes is by the use of heartbeats. This parameter states how often heartbeat signals are sent and how often to expect to receive them. After missing three heartbeat intervals in a row, the node is declared dead. Thus, the maximum time for discovering a failure through the heartbeat mechanism is four times the heartbeat interval.

  The default heartbeat interval is 1500 milliseconds (1.5 seconds). This parameter must not be changed drastically and should not vary widely between nodes. If one node uses 5000 milliseconds and the node watching it uses 1000 milliseconds, obviously the node will be declared dead very quickly. This parameter can be changed during an online software upgrade, but only in small increments.

- `HeartbeatIntervalDbApi`

  Each data node sends heartbeat signals to each MySQL server (SQL node) to ensure that it remains in contact. If a MySQL server fails to send a heartbeat in time it is declared “dead,” in which case all ongoing transactions are completed and all resources released. The SQL node cannot reconnect until all activities initiated by the previous MySQL instance have been completed. The three-heartbeat criteria for this determination are the same as described for `HeartbeatIntervalDbDb`.

  The default interval is 1500 milliseconds (1.5 seconds). This interval can vary between individual data nodes because each data node watches the MySQL servers connected to it, independently of all other data nodes.

- `TimeBetweenLocalCheckpoints`

  This parameter is an exception in that it does not specify a time to wait before starting a new local checkpoint; rather, it is used to ensure that local checkpoints are not performed in a cluster where relatively few updates are taking place. In most clusters with high update rates, it is likely that a new local checkpoint is started immediately after the previous one has been completed.

  The size of all write operations executed since the start of the previous local checkpoints is added. This parameter is also exceptional in that it is specified as the base-2 logarithm of the number of 4-byte words, so that the default value 20 means 4MB (4×2^{20}) of write operations, 21 would mean 8MB, and so on up to a maximum value of 31, which equates to 8GB of write operations.

  All the write operations in the cluster are added together. Setting `TimeBetweenLocalCheckpoints` to 6 or less means that local checkpoints will be executed continuously without pause, independent of the cluster's workload.

- `TimeBetweenGlobalCheckpoints`

  When a transaction is committed, it is committed in main memory in all nodes on which the data is mirrored. However, transaction log records are not flushed to disk as part of the commit. The reasoning behind this behavior is that having the transaction safely committed on at least two autonomous host machines should meet reasonable standards for durability.

  It is also important to ensure that even the worst of cases — a complete crash of the cluster — is handled properly. To guarantee that this happens, all transactions taking place within a given interval are put into a global checkpoint, which can be thought of as a set of committed transactions that has been flushed to disk. In other words, as part of the commit process, a transaction is placed in a global checkpoint group. Later, this group's log records are flushed to disk, and then the entire group of transactions is safely committed to disk on all computers in the cluster.

  This parameter defines the interval between global checkpoints. The default is 2000 milliseconds.

- `TimeBetweenEpochs`

  This parameter defines the interval between synchronisation epochs for MySQL Cluster Replication. The default value is 100 milliseconds.

  `TimeBetweenEpochs` is part of the implementation of “micro-GCPs”, which can be used to improve the performance of MySQL Cluster Replication. This parameter was introduced in MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.2.

- `TimeBetweenEpochsTimeout`

  This parameter defines a timeout for synchronisation epochs for MySQL Cluster Replication. If a node fails to participate in a global checkpoint within the time determined by this parameter, the node is shut down. The default value is 4000 milliseconds.

  `TimeBetweenEpochsTimeout` is part of the implementation of “micro-GCPs”, which can be used to improve the performance of MySQL Cluster Replication. This parameter was introduced in MySQL Cluster NDB 6.2.7 and MySQL Cluster NDB 6.3.4.

- `MaxBufferedEpochs`

  The number of unprocessed epochs by which a subscribing node can lag behind. Exceeding this number causes a lagging subscriber to be disconnected.

  The default value of 100 is sufficient for most normal operations. If a subscribing node does lag enough to cause disconnections, it is usually due to network or scheduling issues with regard to processes or threads. (In rare circumstances, the problem may be due to a bug in the `NDB` client.) It may be desirable to set the value lower than the default when epochs are longer.

  Disconnection prevents client issues from affecting the data node service, running out of memory to buffer data, and eventually shutting down. Instead, only the client is affected as a result of the disconnect (by, for example gap events in the binlog), forcing the client to reconnect or restart the process.

- `TimeBetweenInactiveTransactionAbortCheck`

  Timeout handling is performed by checking a timer on each transaction once for every interval specified by this parameter. Thus, if this parameter is set to 1000 milliseconds, every transaction will be checked for timing out once per second.

  The default value is 1000 milliseconds (1 second).

- `TransactionInactiveTimeout`

  This parameter states the maximum time that is permitted to lapse between operations in the same transaction before the transaction is aborted.

  The default for this parameter is zero (no timeout). For a real-time database that needs to ensure that no transaction keeps locks for too long, this parameter should be set to a relatively small value. The unit is milliseconds.

- `TransactionDeadlockDetectionTimeout`

  When a node executes a query involving a transaction, the node waits for the other nodes in the cluster to respond before continuing. A failure to respond can occur for any of the following reasons:

  - The node is “dead”
  - The operation has entered a lock queue
  - The node requested to perform the action could be heavily overloaded.

  This timeout parameter states how long the transaction coordinator waits for query execution by another node before aborting the transaction, and is important for both node failure handling and deadlock detection. In MySQL 5.1.10 and earlier versions, setting it too high could cause undesirable behavior in situations involving deadlocks and node failure. Beginning with MySQL 5.1.11, active transactions occurring during node failures are actively aborted by the MySQL Cluster Transaction Coordinator, and so high settings are no longer an issue with this parameter.

  The default timeout value is 1200 milliseconds (1.2 seconds).

  Prior to MySQL Cluster NDB versions 6.2.18, 6.3.24, and 7.0.5, the effective minimum for this parameter was 100 milliseconds. (Bug#44099) Beginning with these versions, the actual minimum is 50 milliseconds.

- `DiskSyncSize`

  This is the maximum number of bytes to store before flushing data to a local checkpoint file. This is done in order to prevent write buffering, which can impede performance significantly. This parameter is *not* intended to take the place of `TimeBetweenLocalCheckpoints`.

  > **Note**
  >
  > When `ODirect` is enabled, it is not necessary to set `DiskSyncSize`; in fact, in such cases its value is simply ignored.

  The default value is 4M (4 megabytes).

  This parameter was added in MySQL 5.1.12.

- `DiskCheckpointSpeed`

The amount of data,in bytes per second, that is sent to disk during a local checkpoint.

The default value is 10M (10 megabytes per second).

This parameter was added in MySQL 5.1.12.

- `DiskCheckpointSpeedInRestart`

  The amount of data,in bytes per second, that is sent to disk during a local checkpoint as part of a restart operation.

  The default value is 100M (100 megabytes per second).

  This parameter was added in MySQL 5.1.12.

- `NoOfDiskPagesToDiskAfterRestartTUP`

  When executing a local checkpoint, the algorithm flushes all data pages to disk. Merely doing so as quickly as possible without any moderation is likely to impose excessive loads on processors, networks, and disks. To control the write speed, this parameter specifies how many pages per 100 milliseconds are to be written. In this context, a “page” is defined as 8KB. This parameter is specified in units of 80KB per second, so setting `NoOfDiskPagesToDiskAfterRestartTUP` to a value of `20` entails writing 1.6MB in data pages to disk each second during a local checkpoint. This value includes the writing of UNDO log records for data pages. That is, this parameter handles the limitation of writes from data memory. (See the entry for `IndexMemory` for information about index pages.)

  In short, this parameter specifies how quickly to execute local checkpoints. It operates in conjunction with `NoOfFragmentLogFiles`, `DataMemory`, and `IndexMemory`.

  For more information about the interaction between these parameters and possible strategies for choosing appropriate values for them, see Section 3.6, “Configuring MySQL Cluster Parameters for Local Checkpoints”.

  The default value is 40 (3.2MB of data pages per second).

  > **Note**
  >
  > This parameter is deprecated as of MySQL 5.1.6. For MySQL 5.1.12 and later versions, use DiskCheckpointSpeed and DiskSyncSize instead.

- `NoOfDiskPagesToDiskAfterRestartACC`

  This parameter uses the same units as `NoOfDiskPagesToDiskAfterRestartTUP` and acts in a similar fashion, but limits the speed of writing index pages from index memory.

  The default value of this parameter is 20 (1.6MB of index memory pages per second).

  > **Note**
  >
  > This parameter is deprecated as of MySQL 5.1.6. For MySQL 5.1.12 and later versions, use DiskCheckpointSpeed and DiskSyncSize.

- `NoOfDiskPagesToDiskDuringRestartTUP`

  This parameter is used in a fashion similar to `NoOfDiskPagesToDiskAfterRestartTUP` and `NoOfDiskPagesToDiskAfterRestartACC`, only it does so with regard to local checkpoints executed in the node when a node is restarting. A local checkpoint is always performed as part of all node restarts. During a node restart it is possible to write to disk at a higher speed than at other times, because fewer activities are being performed in the node.

  This parameter covers pages written from data memory.

  The default value is 40 (3.2MB per second).

  > **Note**
  >
  > This parameter is deprecated as of MySQL 5.1.6. For MySQL 5.1.12 and later versions, use DiskCheckpointSpeedInRestart and DiskSyncSize.

-

`NoOfDiskPagesToDiskDuringRestartACC`

Controls the number of index memory pages that can be written to disk during the local checkpoint phase of a node restart.

As with `NoOfDiskPagesToDiskAfterRestartTUP` and `NoOfDiskPagesToDiskAfterRestartACC`, values for this parameter are expressed in terms of 8KB pages written per 100 milliseconds (80KB/second).

The default value is 20 (1.6MB per second).

> **Note**
>
> This parameter is deprecated as of MySQL 5.1.6. For MySQL 5.1.12 and later versions, use DiskCheckpointSpeedInRestart and DiskSyncSize.

- `ArbitrationTimeout`

  This parameter specifies how long data nodes wait for a response from the arbitrator to an arbitration message. If this is exceeded, the network is assumed to have split.

  The default value is 1000 milliseconds (1 second).

Buffering and logging. Several `[ndbd]` configuration parameters enable the advanced user to have more control over the resources used by node processes and to adjust various buffer sizes at need.

These buffers are used as front ends to the file system when writing log records to disk. If the node is running in diskless mode, these parameters can be set to their minimum values without penalty due to the fact that disk writes are “faked” by the `NDB` storage engine's file system abstraction layer.

- `UndoIndexBuffer`

  The UNDO index buffer, whose size is set by this parameter, is used during local checkpoints. The `NDB` storage engine uses a recovery scheme based on checkpoint consistency in conjunction with an operational REDO log. To produce a consistent checkpoint without blocking the entire system for writes, UNDO logging is done while performing the local checkpoint. UNDO logging is activated on a single table fragment at a time. This optimization is possible because tables are stored entirely in main memory.

  The UNDO index buffer is used for the updates on the primary key hash index. Inserts and deletes rearrange the hash index; the NDB storage engine writes UNDO log records that map all physical changes to an index page so that they can be undone at system restart. It also logs all active insert operations for each fragment at the start of a local checkpoint.

  Reads and updates set lock bits and update a header in the hash index entry. These changes are handled by the page-writing algorithm to ensure that these operations need no UNDO logging.

  This buffer is 2MB by default. The minimum value is 1MB, which is sufficient for most applications. For applications doing extremely large or numerous inserts and deletes together with large transactions and large primary keys, it may be necessary to increase the size of this buffer. If this buffer is too small, the NDB storage engine issues internal error code 677 (`Index UNDO buffers overloaded`).

  > **Important**
  >
  > It is not safe to decrease the value of this parameter during a rolling restart.

- `UndoDataBuffer`

  This parameter sets the size of the UNDO data buffer, which performs a function similar to that of the UNDO index buffer, except the UNDO data buffer is used with regard to data memory rather than index memory. This buffer is used during the local checkpoint phase of a fragment for inserts, deletes, and updates.

  Because UNDO log entries tend to grow larger as more operations are logged, this buffer is also larger than its index memory counterpart, with a default value of 16MB.

  This amount of memory may be unnecessarily large for some applications. In such cases, it is possible to decrease this size to a minimum of 1MB.

  It is rarely necessary to increase the size of this buffer. If there is such a need, it is a good idea to check whether the disks can

actually handle the load caused by database update activity. A lack of sufficient disk space cannot be overcome by increasing the size of this buffer.

If this buffer is too small and gets congested, the NDB storage engine issues internal error code 891 (DATA UNDO BUFFERS OVERLOADED).

> **Important**
>
> It is not safe to decrease the value of this parameter during a rolling restart.

- `RedoBuffer`

  All update activities also need to be logged. The REDO log makes it possible to replay these updates whenever the system is restarted. The NDB recovery algorithm uses a "fuzzy" checkpoint of the data together with the UNDO log, and then applies the REDO log to play back all changes up to the restoration point.

  `RedoBuffer` sets the size of the buffer in which the REDO log is written. In MySQL Cluster NDB 6.4.3 and earlier, the default value is 8MB; beginning with MySQL Cluster NDB 7.0.4, the default is 32MB. The minimum value is 1MB.

  If this buffer is too small, the NDB storage engine issues error code 1221 (`REDO log buffers overloaded`).

  > **Important**
  >
  > It is not safe to decrease the value of this parameter during a rolling restart.

Controlling log messages. In managing the cluster, it is very important to be able to control the number of log messages sent for various event types to `stdout`. For each event category, there are 16 possible event levels (numbered 0 through 15). Setting event reporting for a given event category to level 15 means all event reports in that category are sent to `stdout`; setting it to 0 means that there will be no event reports made in that category.

By default, only the startup message is sent to `stdout`, with the remaining event reporting level defaults being set to 0. The reason for this is that these messages are also sent to the management server's cluster log.

An analogous set of levels can be set for the management client to determine which event levels to record in the cluster log.

- `LogLevelStartup`

  The reporting level for events generated during startup of the process.

  The default level is 1.

- `LogLevelShutdown`

  The reporting level for events generated as part of graceful shutdown of a node.

  The default level is 0.

- `LogLevelStatistic`

  The reporting level for statistical events such as number of primary key reads, number of updates, number of inserts, information relating to buffer usage, and so on.

  The default level is 0.

- `LogLevelCheckpoint`

  The reporting level for events generated by local and global checkpoints.

  The default level is 0.

- `LogLevelNodeRestart`

  The reporting level for events generated during node restart.

The default level is 0.

- `LogLevelConnection`

  The reporting level for events generated by connections between cluster nodes.

  The default level is 0.

- `LogLevelError`

  The reporting level for events generated by errors and warnings by the cluster as a whole. These errors do not cause any node failure but are still considered worth reporting.

  The default level is 0.

- `LogLevelCongestion`

  The reporting level for events generated by congestion. These errors do not cause node failure but are still considered worth reporting.

  The default level is 0.

- `LogLevelInfo`

  The reporting level for events generated for information about the general state of the cluster.

  The default level is 0.

- `MemReportFrequency`

  This parameter controls how often data node memory usage reports are recorded in the cluster log; it is an integer value representing the number of seconds between reports.

  Each data node's data memory and index memory usage is logged as both a percentage and a number of 32 KB pages of the `DataMemory` and `IndexMemory`, respectively, set in the `config.ini` file. For example, if `DataMemory` is equal to 100 MB, and a given data node is using 50 MB for data memory storage, the corresponding line in the cluster log might look like this:

  ```
  2006-12-24 01:18:16 [MgmSrvr] INFO -- Node 2: Data usage is 50%(1280 32K pages of total 2560)
  ```

  `MemReportFrequency` is not a required parameter. If used, it can be set for all cluster data nodes in the `[ndbd default]` section of `config.ini`, and can also be set or overridden for individual data nodes in the corresponding `[ndbd]` sections of the configuration file. The minimum value — which is also the default value — is 0, in which case memory reports are logged only when memory usage reaches certain percentages (80%, 90%, and 100%), as mentioned in the discussion of statistics events in Section 7.4.2, “MySQL Cluster Log Events”.

  This parameter was added in MySQL Cluster 5.1.16 and MySQL Cluster NDB 6.1.0.

Backup parameters. The `[ndbd]` parameters discussed in this section define memory buffers set aside for execution of online backups.

- `BackupDataBufferSize`

  In creating a backup, there are two buffers used for sending data to the disk. The backup data buffer is used to fill in data recorded by scanning a node's tables. Once this buffer has been filled to the level specified as `BackupWriteSize` (see below), the pages are sent to disk. While flushing data to disk, the backup process can continue filling this buffer until it runs out of space. When this happens, the backup process pauses the scan and waits until some disk writes have completed freed up memory so that scanning may continue.

  In MySQL Cluster NDB 6.4.3 and earlier, the default value is 2MB; in MySQL Cluster NDB 7.0.4 and later, it is 16MB.

- `BackupLogBufferSize`

The backup log buffer fulfills a role similar to that played by the backup data buffer, except that it is used for generating a log of all table writes made during execution of the backup. The same principles apply for writing these pages as with the backup data buffer, except that when there is no more space in the backup log buffer, the backup fails. For that reason, the size of the backup log buffer must be large enough to handle the load caused by write activities while the backup is being made. See Section 7.3.3, "Configuration for MySQL Cluster Backups".

The default value for this parameter should be sufficient for most applications. In fact, it is more likely for a backup failure to be caused by insufficient disk write speed than it is for the backup log buffer to become full. If the disk subsystem is not configured for the write load caused by applications, the cluster is unlikely to be able to perform the desired operations.

It is preferable to configure cluster nodes in such a manner that the processor becomes the bottleneck rather than the disks or the network connections.

In MySQL Cluster NDB 6.4.3 and earlier, the default value is 2MB; in MySQL Cluster NDB 7.0.4 and later, it is 16MB.

- `BackupMemory`

  This parameter is simply the sum of `BackupDataBufferSize` and `BackupLogBufferSize`.

  In MySQL Cluster NDB 6.4.3 and earlier, the default value was 2MB + 2MB = 4MB; in MySQL Cluster NDB 7.0.4 and later, it is 16MB + 16MB = 32MB.

  > **Important**
  >
  > If `BackupDataBufferSize` and `BackupLogBufferSize` taken together exceed the default value for `BackupMemory`, then this parameter must be set explicitly in the `config.ini` file to their sum.

- `BackupReportFrequency`

  This parameter controls how often backup status reports are issued in the management client during a backup, as well as how often such reports are written to the cluster log (provided cluster event logging is configured to allow it — see Section 3.4.6, "Defining MySQL Cluster Data Nodes" [35]). `BackupReportFrequency` represents the time in seconds between backup status reports.

  The default value is 0.

  This parameter was added in MySQL Cluster NDB 6.2.3.

- `BackupWriteSize`

  This parameter specifies the default size of messages written to disk by the backup log and backup data buffers.

  In MySQL Cluster 6.4.3 and earlier, the default value for this parameter was 32KB; beginning with MySQL Cluster NDB 7.0.4, it is 256KB.

- `BackupMaxWriteSize`

  This parameter specifies the maximum size of messages written to disk by the backup log and backup data buffers.

  In MySQL Cluster 6.4.3 and earlier, the default value for this parameter was 256KB; beginning with MySQL Cluster NDB 7.0.4, it is 1MB.

  > **Important**
  >
  > When specifying these parameters, the following relationships must hold true. Otherwise, the data node will be unable to start:
  >
  > - `BackupDataBufferSize >= BackupWriteSize + 188KB`
  > - `BackupLogBufferSize >= BackupWriteSize + 16KB`
  > - `BackupMaxWriteSize >= BackupWriteSize`

Realtime Performance Parameters

The [ndbd] parameters discussed in this section are used in scheduling and locking of threads to specific CPUs on multiprocessor data node hosts. They were introduced in MySQL Cluster NDB 6.3.4.

- LockExecuteThreadToCPU

  Previous to MySQL Cluster NDB 7.0. This parameter specifies the ID of the CPU assigned to handle the NDBCLUSTER execution thread. The value of this parameter is an integer in the range 0 to 65535 (inclusive). The default is 65535.

  MySQL Cluster NDB 7.0 and later (beginning with MySQL Cluster NDB 6.4.0). When used with ndbd, this parameter (now a string) specifies the ID of the CPU assigned to handle the NDBCLUSTER execution thread. When used with ndbmtd, the value of this parameter is a comma-separated list of CPU IDs assigned to handle execution threads. Each CPU ID in the list should be an integer in the range 0 to 65535 (inclusive). The number of IDs specified should match the number of execution threads determined by MaxNoOfExecutionThreads. There is no default value.

- LockMaintThreadsToCPU

  This parameter specifies the ID of the CPU assigned to handle NDBCLUSTER maintenance threads.

  The value of this parameter is an integer in the range 0 to 65535 (inclusive). This parameter was added in MySQL Cluster NDB 6.3.4. Prior to MySQL Cluster NDB 6.4.0, the default is 65535; in MySQL Cluster NDB 7.0 and later MySQL Cluster release series, there is no default value.

- RealtimeScheduler

  Setting this parameter to 1 enables real-time scheduling of NDBCLUSTER threads.

  The default is 0 (scheduling disabled).

- SchedulerExecutionTimer

  This parameter specifies the time in microseconds for threads to be executed in the scheduler before being sent. Setting it to 0 minimizes the response time; to achieve higher throughput, you can increase the value at the expense of longer response times.

  The default is 50 µsec, which our testing shows to increase throughput slightly in high-load cases without materially delaying requests.

  This parameter was added in MySQL Cluster NDB 6.3.4.

- SchedulerSpinTimer

  This parameter specifies the time in microseconds for threads to be executed in the scheduler before sleeping.

  The default value is 0.

Disk Data Configuration Parameters. Configuration parameters affecting Disk Data behavior include the following:

- DiskPageBufferMemory

  This determines the amount of space used for caching pages on disk, and is set in the [ndbd] or [ndbd default] section of the config.ini file. It is measured in bytes. Each page takes up 32 KB. This means that Cluster Disk Data storage always uses *N* * 32 KB memory where *N* is some non-negative integer.

  The default value for this parameter is 64M (2000 pages of 32 KB each).

  This parameter was added in MySQL 5.1.6.

- SharedGlobalMemory

  This determines the amount of memory that is used for log buffers, disk operations (such as page requests and wait queues), and metadata for tablespaces, log file groups, UNDO files, and data files. It can be set in the [ndbd] or [ndbd default] section of the config.ini configuration file, and is measured in bytes.

The default value is `20M`.

This parameter was added in MySQL 5.1.6.

- `DiskIOThreadPool`

  This parameter determines the number of unbound threads used for Disk Data file access. Currently, it applies to Disk Data I/O threads only, but we plan in the future to make the number of such threads configurable for in-memory data as well.

  The optimum value for this parameter depends on your hardware and configuration, and includes these factors:

  - **Physical distribution of Disk Data files.** You can obtain better performance by placing data files, undo log files, and the data node filesystem on separate physical disks. If you do this with some or all of these sets of files, then you can set `DiskIOThreadPool` higher to allow separate threads to handle the files on each disk.

  - **Disk performance and types.** The number of threads that can be accommodated for Disk Data file handling is also dependent on the speed and throughput of the disks. Faster disks and higher throughput allow for more disk I/O threads. Our test results indicate that solid-state disk drives can handle many more disk I/O threads than conventional disks, and thus higher values for `DiskIOThreadPool`.

  This parameter was added in MySQL Cluster NDB 6.4.0. Previous to MySQL Cluster NDB 6.4.3, it was named `ThreadPool`. The default value is 8

- **Disk Data filesystem parameters.** The parameters in the following list were added in MySQL Cluster NDB 6.2.17, 6.3.22, and 6.4.3 to make it easier to place MySQL Cluster Disk Data files in specific directories.

  - `FileSystemPathDD`

    If this parameter is specified, then MySQL Cluster Disk Data data files and undo log files are placed in the indicated directory. This can be overridden for data files, undo log files, or both, by specifying values for `FileSystemPathDataFiles`, `FileSystemPathUndoFiles`, or both, as explained for these parameters. It can also be overridden for data files by specifying a path in the `ADD DATAFILE` clause of a `CREATE TABLESPACE` or `ALTER TABLESPACE` statement, and for undo log files by specifying a path in the `ADD UNDOFILE` clause of a `CREATE LOGFILE GROUP` or `ALTER LOGFILE GROUP` statement. If `FileSystemPathDD` is not specified, then `FileSystemPath` is used.

    If a `FileSystemPathDD` directory is specified for a given data node (including the case where the parameter is specified in the `[ndbd default]` section of the `config.ini` file), then starting that data node with `--initial` causes all files in the directory to be deleted.

  - `FileSystemPathDataFiles`

    If this parameter is specified, then MySQL Cluster Disk Data data files are placed in the indicated directory. This overrides any value set for `FileSystemPathDD`. This parameter can be overridden for a given data file by specifying a path in the `ADD DATAFILE` clause of a `CREATE TABLESPACE` or `ALTER TABLESPACE` statement used to create that data file. If `FileSystemPathDataFiles` is not specified, then `FileSystemPathDD` is used (or `FileSystemPath`, if `FileSystemPathDD` has also not been set).

    If a `FileSystemPathDataFiles` directory is specified for a given data node (including the case where the parameter is specified in the `[ndbd default]` section of the `config.ini` file), then starting that data node with `--initial` causes all files in the directory to be deleted.

  - `FileSystemPathUndoFiles`

    If this parameter is specified, then MySQL Cluster Disk Data undo log files are placed in the indicated directory. This overrides any value set for `FileSystemPathDD`. This parameter can be overridden for a given data file by specifying a path in the `ADD UNDO` clause of a `CREATE LOGFILE GROUP` or `CREATE LOGFILE GROUP` statement used to create that data file. If `FileSystemPathUndoFiles` is not specified, then `FileSystemPathDD` is used (or `FileSystemPath`, if `FileSystemPathDD` has also not been set).

    If a `FileSystemPathUndoFiles` directory is specified for a given data node (including the case where the parameter is specified in the `[ndbd default]` section of the `config.ini` file), then starting that data node with `--initial` causes all files in the directory to be deleted.

  For more information, see Section 10.1, “MySQL Cluster Disk Data Objects”.

- **Disk Data object creation parameters.** The next two parameters enable you — when starting the cluster for the first time —

to cause a Disk Data log file group, tablespace, or both, to be created without the use of SQL statements.

- `InitialLogFileGroup`

  This parameter can be used to specify a log file group that is created when performing an initial start of the cluster. `InitialLogFileGroup` is specified as shown here:

  ```
  InitialLogFileGroup = [name=name;] [undobuffer_size=size;] file-specification-list
  file-specification-list:
      file-specification[; file-specification[; ...]]
  file-specification:
      filename:size
  ```

  The `name` of the log file group is optional and defaults to `DEFAULT_LG`. The `undobuffer_size` is also optional; if omitted, it defaults to `256M` (256 megabytes). Each *`file-specification`* corresponds to an undo log file, and at least one must be specified in the *`file-specification-list`*. Undo log files are placed according to any values that have been set for `FileSystemPath`, `FileSystemPathDD`, and `FileSystemPathUndoFiles`, just as if they had been created as the result of a `CREATE LOGFILE GROUP` or `ALTER LOGFILE GROUP` statement.

  Consider the following example:

  ```
  InitialLogFileGroup = name=LG1; undobuffer_size=128M; undo1.log:250M; undo2.log:150M
  ```

  This is equivalent to the following SQL statements:

  ```
  CREATE LOGFILE GROUP LG1
      ADD UNDOFILE 'undo1.log'
      INITIAL_SIZE 250M
      UNDO_BUFFER_SIZE 128M
      ENGINE NDBCLUSTER;
  ALTER LOGFILE GROUP LG1
      ADD UNDOFILE 'undo2.log'
      INITIAL_SIZE 150M
      ENGINE NDBCLUSTER;
  ```

  This logfile group is created when the data nodes are started with `--initial`.

  This parameter, if used, should always be set in the `[ndbd default]` section of the `config.ini` file. The behavior of a MySQL Cluster when different values are set on different data nodes is not defined.

- `InitialTablespace`

  This parameter can be used to specify a MySQL Cluster Disk Data tablespace that is created when performing an initial start of the cluster. `InitialTablespace` is specified as shown here:

  ```
  InitialTablespace = [name=name;] [extent_size=size;] [logfile_group=lg-name;] file-specification-list
  ```

  The `name` of the tablespace is optional and defaults to `DEFAULT_TS`. The `extent_size` is also optional; it defaults to `1M` (1 megabyte). The `logfile_group` is also optional, and defaults to `DEFAULT_LG`. The *`file-specification-list`* uses the same syntax as shown with the `InitialLogfileGroup` parameter, the only difference being that each *`file-specification`* used with `InitialTablespace` corresponds to a data file. At least one must be specified in the *`file-specification-list`*. Data files are placed according to any values that have been set for `FileSystemPath`, `FileSystemPathDD`, and `FileSystemPathDataFiles`, just as if they had been created as the result of a `CREATE TABLESPACE` or `ALTER TABLESPACE` statement.

  For example, consider the following line specifying `InitialTablespace` in the `[ndbd default]` section of the `config.ini` file (as with `InitialLogfileGroup`, this parameter should always be set in the `[ndbd default]` section, as the behavior of a MySQL Cluster when different values are set on different data nodes is not defined):

  ```
  InitialTablespace = name=TS1; extent_size=8M; logfile_group=LG1; data1.dat:2G; data2.dat:4G
  ```

  This is equivalent to the following SQL statements:

  ```
  CREATE TABLESPACE TS1
      ADD DATAFILE 'data1.dat'
      USE LOGFILE GROUP LG1
      EXTENT_SIZE 8M
      INITIAL_SIZE 2G
      ENGINE NDBCLUSTER;
  ALTER TABLESPACE TS1
      ADD UNDOFILE 'data2.dat'
      INITIAL_SIZE 4G
      ENGINE NDBCLUSTER;
  ```

This tablespace is created when the data nodes are started with `--initial`, and can be used whenever creating MySQL Cluster Disk Data tables thereafter.

Disk Data and `GCP STOP` errors. Errors encountered when using Disk Data tables such as `NODE *NODEID* KILLED THIS NODE BECAUSE GCP STOP WAS DETECTED` (error 2303) are often referred to as "CGP stop errors". Such errors are usually due to slow disks and insufficient disk throughput. You can help prevent these errors from occurring by using faster disks, and by adjusting the cluster configuration as discussed here:

- **MySQL Cluster NDB 6.2 and 6.3.** When working with large amounts of data on disk under high load, the default value for `DiskPageBufferMemory` may not be large enough. In such cases, you should increase its value to include most of the memory available to the data nodes after accounting for index memory, data memory, internal buffers, and memory needed by the data node host operating system. You can use this formula as a guide:

  ```
  DiskPageBufferMemory
    = 0.8
      x (
          [total memory]
            - ([operating system memory] + [buffer memory] + DataMemory + IndexMemory)
        )
  ```

  Once you have established that sufficient memory is reserved for `DataMemory`, `IndexMemory`, NDB internal buffers, and operating system overhead, it is possible (and sometimes desirable) to allocate more than the above amount of the remainder to `DiskPageBufferMemory`.

- **MySQL Cluster NDB 6.4 and 7.X.** In addition to the considerations given for `DiskPageBufferMemory` as explained in the previous item, it is also very important that the `DiskIOThreadPool` configuration parameter be set correctly; having `DiskIOThreadPool` set too high is very likely to cause GCP stop errors (Bug#37227).

Parameters for configuring send buffer memory allocation (MySQL Cluster NDB 7.0). Beginning with MySQL Cluster NDB 6.4.0, send buffer memory is allocated dynamically from a memory pool shared between all transporters, which means that the size of the send buffer can be adjusted as necessary. (Previously, the NDB kernel used a fixed-size send buffer for every node in the cluster, which was allocated when the node started and could not be changed while the node was running.) The following data node configuration parameters were added in MySQL Cluster NDB 6.4.0 to permit the setting of limits on this memory allocation; this change is reflected by the addition of the configuration parameters `TotalSendBufferMemory`, `ReservedSendBuffer-Memory`, and `OverLoadLimit`, as well as a change in how the existing `SendBufferMemory` configuration parameter is used. For more information, see Section 3.7, "Configuring MySQL Cluster Send Buffer Parameters".

- `TotalSendBufferMemory`

  This parameter is available beginning with MySQL Cluster NDB 6.4.0. It is used to determine the total amount of memory to allocate on this node for shared send buffer memory among all configured transporters.

  If this parameter is set, its minimum allowed value is 256K; the maxmimum is 4294967039.

- `ReservedSendBufferMemory`

  This optional parameter is available beginning with MySQL Cluster NDB 6.4.0. If set, it reserves an amount of memory (in bytes) for connections between data nodes, and that cannot be allocated for send buffers to be used for management or API nodes. This helps prevent API nodes from using excess send buffer memory and thereby causing communications failures in the NDB kernel.

  If this parameter is set, its minimum allowed value is 256K; the maxmimum is 4294967039.

For more detailed information about the behavior and use of `TotalSendBufferMemory` and `ReservedSendBuffer-Memory`, and about configuring send buffer memory parameters in MySQL Cluster NDB 6.4.0 and later, see Section 3.7, "Configuring MySQL Cluster Send Buffer Parameters".

> **Note**
>
> Previous to MySQL Cluster NDB 7.0, to add new data nodes to a MySQL Cluster, it is necessary to shut down the cluster completely, update the `config.ini` file, and then restart the cluster (that is, you must perform a system restart). All data node processes must be started with the `--initial` option.

Beginning with MySQL Cluster NDB 7.0, it is possible to add new data node groups to a running cluster online. See Section 7.8, "Adding MySQL Cluster Data Nodes Online", for more information.

3.4.7. Defining SQL and Other API Nodes in a MySQL Cluster

The `[mysqld]` and `[api]` sections in the `config.ini` file define the behavior of the MySQL servers (SQL nodes) and other applications (API nodes) used to access cluster data. None of the parameters shown is required. If no computer or host name is provided, any host can use this SQL or API node.

Generally speaking, a `[mysqld]` section is used to indicate a MySQL server providing an SQL interface to the cluster, and an `[api]` section is used for applications other than `mysqld` processes accessing cluster data, but the two designations are actually synonomous; you can, for instance, list parameters for a MySQL server acting as an SQL node in an `[api]` section.

Note

For a discussion of MySQL server options for MySQL Cluster, see Section 4.2, "`mysqld` Command Options for MySQL Cluster"; for information about MySQL server system variables relating to MySQL Cluster, see Section 4.3, "MySQL Cluster System Variables".

- `Id`

  The `Id` is an integer value used to identify the node in all cluster internal messages. Prior to MySQL Cluster NDB 6.1.1, the permitted range of values for this parameter was 1 to 63 inclusive. Beginning with MySQL Cluster NDB 6.1.1, the permitted range is 1 to 255 inclusive. This value must be unique for each node in the cluster, regardless of the type of node.

  Note

  Data node IDs must be less than 49, regardless of the MySQL Cluster version used. If you plan to deploy a large number of data nodes, it is a good idea to limit the node IDs for API nodes (and management nodes) to values greater than 48.

- `ExecuteOnComputer`

  This refers to the `Id` set for one of the computers (hosts) defined in a `[computer]` section of the configuration file.

- `HostName`

  Specifying this parameter defines the hostname of the computer on which the SQL node (API node) is to reside. To specify a hostname, either this parameter or `ExecuteOnComputer` is required.

  If no `HostName` or `ExecuteOnComputer` is specified in a given `[mysql]` or `[api]` section of the `config.ini` file, then an SQL or API node may connect using the corresponding "slot" from any host which can establish a network connection to the management server host machine. *This differs from the default behavior for data nodes, where `localhost` is assumed for `HostName` unless otherwise specified.*

- `ArbitrationRank`

  This parameter defines which nodes can act as arbitrators. Both MGM nodes and SQL nodes can be arbitrators. A value of 0 means that the given node is never used as an arbitrator, a value of 1 gives the node high priority as an arbitrator, and a value of 2 gives it low priority. A normal configuration uses the management server as arbitrator, setting its `ArbitrationRank` to 1 (the default) and those for all SQL nodes to 0.

  Beginning with MySQL 5.1.16 and MySQL Cluster NDB 6.1.3, it is possible to disable arbitration completely by setting `ArbitrationRank` to 0 on all management and SQL nodes.

- `ArbitrationDelay`

  Setting this parameter to any other value than 0 (the default) means that responses by the arbitrator to arbitration requests will be delayed by the stated number of milliseconds. It is usually not necessary to change this value.

- `BatchByteSize`

  For queries that are translated into full table scans or range scans on indexes, it is important for best performance to fetch records in properly sized batches. It is possible to set the proper size both in terms of number of records (`BatchSize`) and in

terms of bytes (`BatchByteSize`). The actual batch size is limited by both parameters.

The speed at which queries are performed can vary by more than 40% depending upon how this parameter is set. In future releases, MySQL Server will make educated guesses on how to set parameters relating to batch size, based on the query type.

This parameter is measured in bytes and by default is equal to 32KB.

- `BatchSize`

  This parameter is measured in number of records and is by default set to 64. The maximum size is 992.

- `MaxScanBatchSize`

  The batch size is the size of each batch sent from each data node. Most scans are performed in parallel to protect the MySQL Server from receiving too much data from many nodes in parallel; this parameter sets a limit to the total batch size over all nodes.

  The default value of this parameter is set to 256KB. Its maximum size is 16MB.

- `TotalSendBufferMemory`

  This parameter is available beginning with MySQL Cluster NDB 6.4.0. It is used to determine the total amount of memory to allocate on this node for shared send buffer memory among all configured transporters.

  If this parameter is set, its minimum allowed value is 256K; the maxmimum is 4294967039. For more detailed information about the behavior and use of `TotalSendBufferMemory` and configuring send buffer memory parameters in MySQL Cluster NDB 6.4.0 and later, see Section 3.7, “Configuring MySQL Cluster Send Buffer Parameters”.

You can obtain some information from a MySQL server running as a Cluster SQL node using `SHOW STATUS` in the `mysql` client, as shown here:

```
mysql> SHOW STATUS LIKE 'ndb%';
+-----------------------------+---------------+
| Variable_name               | Value         |
+-----------------------------+---------------+
| Ndb_cluster_node_id         | 5             |
| Ndb_config_from_host        | 192.168.0.112 |
| Ndb_config_from_port        | 1186          |
| Ndb_number_of_storage_nodes | 4             |
+-----------------------------+---------------+
4 rows in set (0.02 sec)
```

For information about these Cluster system status variables, see Server Status Variables.

> **Note**
>
> To add new SQL or API nodes to the configuration of a running MySQL Cluster, it is necessary to perform a rolling restart of all cluster nodes after adding new `[mysqld]` or `[api]` sections to the `config.ini` file (or files, if you are using more than one management server). This must be done before the new SQL or API nodes can connect to the cluster.
>
> It is *not* necessary to perform any restart of the cluster if new SQL or API nodes can employ previously unused API slots in the cluster configuration to connect to the cluster.

3.4.8. MySQL Cluster TCP/IP Connections

TCP/IP is the default transport mechanism for all connections between nodes in a MySQL Cluster. Normally it is not necessary to define TCP/IP connections; MySQL Cluster automatically sets up such connections for all data nodes, management nodes, and SQL or API nodes.

> **Note**
>
> For an exception to this rule, see Section 3.4.9, “MySQL Cluster TCP/IP Connections Using Direct Connections”.

To override the default connection parameters, it is necessary to define a connection using one or more `[tcp]` sections in the `config.ini` file. Each `[tcp]` section explicitly defines a TCP/IP connection between two MySQL Cluster nodes, and must contain at a minimum the parameters `NodeId1` and `NodeId2`, as well as any connection parameters to override.

It is also possible to change the default values for these parameters by setting them in the `[tcp default]` section.

> **Important**
>
> Any `[tcp]` sections in the `config.ini` file should be listed *last*, following all other sections in the file. However, this is not required for a `[tcp default]` section. This requirement is a known issue with the way in which the `config.ini` file is read by the MySQL Cluster management server.

Connection parameters which can be set in `[tcp]` and `[tcp default]` sections of the `config.ini` file are listed here:

- `NodeId1`, `NodeId2`

  To identify a connection between two nodes it is necessary to provide their node IDs in the `[tcp]` section of the configuration file. These are the same unique `Id` values for each of these nodes as described in Section 3.4.7, "Defining SQL and Other API Nodes in a MySQL Cluster".

- `OverloadLimit`

  Beginning in MySQL Cluster NDB 6.4.0, this parameter can be used to determine the amount of unsent data that must be present in the send buffer before the connection is considered overloaded. See Section 3.7, "Configuring MySQL Cluster Send Buffer Parameters", for more information.

- `SendBufferMemory`

  TCP transporters use a buffer to store all messages before performing the send call to the operating system. When this buffer reaches 64KB its contents are sent; these are also sent when a round of messages have been executed. To handle temporary overload situations it is also possible to define a bigger send buffer.

  Prior to MySQL Cluster NDB 7.0, this parameter determined a fixed amount of memory allocated at startup for each configured TCP connection. Beginning with MySQL Cluster NDB 6.4.0, memory is not dedicated to each transporter. Instead, the value denotes the hard limit for how much memory (out of the total available memory — that is, `TotalSendBufferMemory`) that may be used by a single transporter. For more information about configuring dynamic transporter send buffer memory allocation in MySQL Cluster NDB 7.0 and later, see Section 3.7, "Configuring MySQL Cluster Send Buffer Parameters".

  In MySQL Cluster NDB 6.4.3 and earlier, the default size of the send buffer was 256 KB; in MySQL Cluster NDB 7.0.4 and later, it is 2MB, which is the size recommended in most situations. The minimum size is 64 KB; the theoretical maximum is 4 GB.

- `SendSignalId`

  To be able to retrace a distributed message datagram, it is necessary to identify each message. When this parameter is set to `Y`, message IDs are transported over the network. This feature is disabled by default in production builds, and enabled in `-debug` builds.

- `Checksum`

  This parameter is a boolean parameter (enabled by setting it to `Y` or `1`, disabled by setting it to `N` or `0`). It is disabled by default. When it is enabled, checksums for all messages are calculated before they placed in the send buffer. This feature ensures that messages are not corrupted while waiting in the send buffer, or by the transport mechanism.

- `PortNumber` (*OBSOLETE*)

  This formerly specified the port number to be used for listening for connections from other nodes. This parameter should no longer be used.

- `ReceiveBufferMemory`

  Specifies the size of the buffer used when receiving data from the TCP/IP socket.

  In MySQL Cluster NDB 6.4.3 and earlier, the default value of this parameter was 64 KB; beginning with MySQL Cluster NDB 7.0.4, 2MB is the default. The minimum possible value is 16KB; the theoretical maximum is 4GB.

3.4.9. MySQL Cluster TCP/IP Connections Using Direct Connections

Setting up a cluster using direct connections between data nodes requires specifying explicitly the crossover IP addresses of the

data nodes so connected in the `[tcp]` section of the cluster `config.ini` file.

In the following example, we envision a cluster with at least four hosts, one each for a management server, an SQL node, and two data nodes. The cluster as a whole resides on the `172.23.72.*` subnet of a LAN. In addition to the usual network connections, the two data nodes are connected directly using a standard crossover cable, and communicate with one another directly using IP addresses in the `1.1.0.*` address range as shown:

```
# Management Server
[ndb_mgmd]
Id=1
HostName=172.23.72.20
# SQL Node
[mysqld]
Id=2
HostName=172.23.72.21
# Data Nodes
[ndbd]
Id=3
HostName=172.23.72.22
[ndbd]
Id=4
HostName=172.23.72.23
# TCP/IP Connections
[tcp]
NodeId1=3
NodeId2=4
HostName1=1.1.0.1
HostName2=1.1.0.2
```

The `HostName`*`N`* parameter, where *`N`* is an integer, is used only when specifying direct TCP/IP connections.

The use of direct connections between data nodes can improve the cluster's overall efficiency by allowing the data nodes to bypass an Ethernet device such as a switch, hub, or router, thus cutting down on the cluster's latency. It is important to note that to take the best advantage of direct connections in this fashion with more than two data nodes, you must have a direct connection between each data node and every other data node in the same node group.

3.4.10. MySQL Cluster Shared-Memory Connections

MySQL Cluster attempts to use the shared memory transporter and configure it automatically where possible. `[shm]` sections in the `config.ini` file explicitly define shared-memory connections between nodes in the cluster. When explicitly defining shared memory as the connection method, it is necessary to define at least `NodeId1`, `NodeId2` and `ShmKey`. All other parameters have default values that should work well in most cases.

> **Important**
>
> *SHM functionality is considered experimental only*. It is not officially supported in any current MySQL Cluster release, and testing results indicate that SHM performance is not appreciably greater than when using TCP/IP for the transporter.
>
> For these reasons, you must determine for yourself or by using our free resources (forums, mailing lists) whether SHM can be made to work correctly in your specific case.

- `NodeId1`, `NodeId2`

  To identify a connection between two nodes it is necessary to provide node identifiers for each of them, as `NodeId1` and `NodeId2`.

- `ShmKey`

  When setting up shared memory segments, a node ID, expressed as an integer, is used to identify uniquely the shared memory segment to use for the communication. There is no default value.

- `ShmSize`

  Each SHM connection has a shared memory segment where messages between nodes are placed by the sender and read by the reader. The size of this segment is defined by `ShmSize`. The default value is 1MB.

- `SendSignalId`

  To retrace the path of a distributed message, it is necessary to provide each message with a unique identifier. Setting this para-

meter to Y causes these message IDs to be transported over the network as well. This feature is disabled by default in production builds, and enabled in -debug builds.

- Checksum

  This parameter is a boolean (Y/N) parameter which is disabled by default. When it is enabled, checksums for all messages are calculated before being placed in the send buffer.

  This feature prevents messages from being corrupted while waiting in the send buffer. It also serves as a check against data being corrupted during transport.

- SigNum

  When using the shared memory transporter, a process sends an operating system signal to the other process when there is new data available in the shared memory. Should that signal conflict with with an existing signal, this parameter can be used to change it. This is a possibility when using SHM due to the fact that different operating systems use different signal numbers.

  The default value of SigNum is 0; therefore, it must be set to avoid errors in the cluster log when using the shared memory transporter. Typically, this parameter is set to 10 in the [shm default] section of the config.ini file.

3.4.11. SCI Transport Connections in MySQL Cluster

[sci] sections in the config.ini file explicitly define SCI (Scalable Coherent Interface) connections between cluster nodes. Using SCI transporters in MySQL Cluster is supported only when the MySQL binaries are built using --with-ndb-sci=*/your/path/to/SCI*. The *path* should point to a directory that contains at a minimum lib and include directories containing SISCI libraries and header files. (See Chapter 11, *Using High-Speed Interconnects with MySQL Cluster* for more information about SCI.)

In addition, SCI requires specialized hardware.

It is strongly recommended to use SCI Transporters only for communication between ndbd processes. Note also that using SCI Transporters means that the ndbd processes never sleep. For this reason, SCI Transporters should be used only on machines having at least two CPUs dedicated for use by ndbd processes. There should be at least one CPU per ndbd process, with at least one CPU left in reserve to handle operating system activities.

- NodeId1, NodeId2

  To identify a connection between two nodes it is necessary to provide node identifiers for each of them, as NodeId1 and NodeId2.

- Host1SciId0

  This identifies the SCI node ID on the first Cluster node (identified by NodeId1).

- Host1SciId1

  It is possible to set up SCI Transporters for failover between two SCI cards which then should use separate networks between the nodes. This identifies the node ID and the second SCI card to be used on the first node.

- Host2SciId0

  This identifies the SCI node ID on the second Cluster node (identified by NodeId2).

- Host2SciId1

  When using two SCI cards to provide failover, this parameter identifies the second SCI card to be used on the second node.

- SharedBufferSize

  Each SCI transporter has a shared memory segment used for communication between the two nodes. Setting the size of this segment to the default value of 1MB should be sufficient for most applications. Using a smaller value can lead to problems when performing many parallel inserts; if the shared buffer is too small, this can also result in a crash of the ndbd process.

- `SendLimit`

  A small buffer in front of the SCI media stores messages before transmitting them over the SCI network. By default, this is set to 8KB. Our benchmarks show that performance is best at 64KB but 16KB reaches within a few percent of this, and there was little if any advantage to increasing it beyond 8KB.

- `SendSignalId`

  To trace a distributed message it is necessary to identify each message uniquely. When this parameter is set to `Y`, message IDs are transported over the network. This feature is disabled by default in production builds, and enabled in `-debug` builds.

- `Checksum`

  This parameter is a boolean value, and is disabled by default. When `Checksum` is enabled, checksums are calculated for all messages before they are placed in the send buffer. This feature prevents messages from being corrupted while waiting in the send buffer. It also serves as a check against data being corrupted during transport.

3.5. Overview of MySQL Cluster Configuration Parameters

The next three sections provide summary tables of MySQL Cluster configuration parameters used in the `config.ini` file to govern the cluster's functioning. Each table lists the parameters for one of the Cluster node process types (`ndbd`, `ndb_mgmd`, and `mysqld`), and includes the parameter's type as well as its default, mimimum, and maximum values as applicable.

It is also stated what type of restart is required (node restart or system restart) — and whether the restart must be done with `--initial` — to change the value of a given configuration parameter. This information is provided in each table's **Restart Type** column, which contains one of the values shown in this list:

- `N`: Node Restart
- `IN`: Initial Node Restart
- `S`: System Restart
- `IS`: Initial System Restart

When performing a node restart or an initial node restart, all of the cluster's data nodes must be restarted in turn (also referred to as a *rolling restart*). It is possible to update cluster configuration parameters marked `N` or `IN` online — that is, without shutting down the cluster — in this fashion. An initial node restart requires restarting each `ndbd` process with the `--initial` option.

A system restart requires a complete shutdown and restart of the entire cluster. An initial system restart requires taking a backup of the cluster, wiping the cluster file system after shutdown, and then restoring from the backup following the restart.

In any cluster restart, all of the cluster's management servers must be restarted in order for them to read the updated configuration parameter values.

> **Important**
>
> Values for numeric cluster parameters can generally be increased without any problems, although it is advisable to do so progressively, making such adjustments in relatively small increments. However, decreasing the values of such parameters — particularly those relating to memory usage and disk space — is not to be undertaken lightly, and it is recommended that you do so only following careful planning and testing. In addition, it is the generally the case that parameters relating to memory and disk usage which can be raised using a simple node restart require an initial node restart to be lowered.

Because some of these parameters can be used for configuring more than one type of cluster node, they may appear in more than one of the tables.

> **Note**
>
> `4294967039` — which often appears as a maximum value in these tables — is defined in the `NDBCLUSTER` sources as `MAX_INT_RNIL` and is equal to `0xFFFFFEFF`, or $2^{32} - 2^8 - 1$.

3.5.1. MySQL Cluster Data Node Configuration Parameters

The following table provides information about parameters used in the `[ndbd]` or `[ndbd default]` sections of a `config.ini` file for configuring MySQL Cluster data nodes. For detailed descriptions and other additional information about each of these parameters, see Section 3.4.6, "Defining MySQL Cluster Data Nodes".

Beginning with MySQL Cluster NDB 6.4.0, these parameters also apply to `ndbmtd`, which is a multi-threaded version of `ndbd`. For more information, see Section 6.3, "`ndbmtd` — The MySQL Cluster Data Node Daemon (Multi-Threaded)".

Restart Type *Column Values*

- `N`: Node Restart
- `IN`: Initial Node Restart
- `S`: System Restart
- `IS`: Initial System Restart

See Section 3.5, "Overview of MySQL Cluster Configuration Parameters", for additional explanations of these abbreviations.

Table 3.1. MySQL Cluster Data Node Configuration Parameters

Parameter Name	Type / Units	Default Value	Minimum Value	Maximum Value	Restart Type
`ArbitrationTimeout`	milliseconds	3000	10	4294967039	N
`BackupDataBufferSize`	bytes	*MySQL Cluster 6.4.3 and earlier*: 2M; *MySQL Cluster NDB 7.0.4 and later*: 16M	0	4294967039	N
`BackupDataDir`	string	`FileSystemPath/BACKUP`	N/A	N/A	IN
`BackupLogBufferSize`	bytes	*MySQL Cluster 6.4.3 and earlier*: 2M; *MySQL Cluster NDB 7.0.4 and later*: 16M	0	4294967039	N
`BackupMemory`	bytes	*MySQL Cluster 6.4.3 and earlier*: 4M; *MySQL Cluster NDB 7.0.4 and later*: 32M	0	4294967039	N
`BackupReportFrequency` (Added in MySQL Cluster NDB 6.2.3)	seconds	0	0	4294967039	N
`BackupWriteSize`	bytes	*MySQL Cluster 6.4.3 and earlier*: 32K; *MySQL Cluster NDB 7.0.4 and later*: 256K	2K	4294967039	N
`BackupMaxWriteSize`	bytes	*MySQL Cluster 6.4.3 and earlier*: 256K; *MySQL Cluster NDB 7.0.4 and later*: 1M	2K	4294967039	N
`BatchSizePerLocalScan`	integer	64	1	992	N
`CompressedBackup` (Added in MySQL Cluster NDB 6.3.7)	boolean	0	0	1	N
`CompressedLCP` (Added in MySQL Cluster NDB 6.3.7)	boolean	0	0	1	N
`DataDir`	string	`.`	N/A	N/A	IN

DataMemory	bytes	80M	1M	1024G (subject to available system RAM and size of IndexMemory)	N
DiskCheckpointSpeed (added in MySQL 5.1.12)	integer (number of bytes per second)	10M	1M	4294967039	N
DiskCheckpointSpeedInRestart (added in MySQL 5.1.12)	integer (number of bytes per second)	100M	1M	4294967039	N
DiskIOThreadPool (Added in MySQL Cluster NDB 6.4.0; named ThreadPool prior to MySQL Cluster NDB 6.4.3)	integer	8	0	4294967039	N
Diskless	true|false (1|0)	0	0	1	IS
DiskPageBufferMemory (added in MySQL 5.1.6)	bytes	64M	4M	1024G	N
DiskSyncSize (added in MySQL 5.1.12)	integer (number of bytes)	4M	32K	4294967039	N
ExecuteOnComputer	integer				
FileSystemPath (Added in MySQL Cluster NDB 6.1.11)	string (directory path)	value specified for DataDir	N/A	N/A	IN
FileSystemPathDD (Added in MySQL Cluster NDB 6.2.17, 6.3.22, and 6.4.3)	string (directory path)	value specified for FilesystemPath, if set; otherwise, value of DataDir	N/A	N/A	IN
FileSystemPathDataFiles (Added in MySQL Cluster NDB 6.2.17, 6.3.22, and 6.4.3)	string (directory path)	value specified for FilesystemPathDD, if set; otherwise, value specified for FilesystemPath, if set; otherwise, value of DataDir	N/A	N/A	IN
FileSystemPathUndoFiles (Added in MySQL Cluster NDB 6.2.17, 6.3.22, and 6.4.3)	string (directory path)	value specified for FilesystemPathDD, if set; otherwise, value specified for FilesystemPath, if set; otherwise, value of DataDir	N/A	N/A	IN
FragmentLogFileSize	integer (bytes)	16M	4M	1G	IN

HeartbeatIntervalDbApi	milliseconds	1500	100	4294967039	N
HeartbeatIntervalDbDb	milliseconds	1500	10	4294967039	N
HostName	string	localhost	N/A	N/A	S
Id	integer	*None*	1	48	IS
IndexMemory	bytes	18M	1M	1024G (subject to available system RAM and size of DataMemory)	N
InitFragmentLogFiles (Added in MySQL Cluster NDB 6.3.19)	string	full	sparse	sparse	IN
InitialLogFileGroup (Added in MySQL Cluster NDB 6.2.17, 6.3.22, and 6.4.3)	string (see description for details)	*none*	N/A	N/A	S
InitialNoOfOpenFiles	integer	27	20	4294967039	N
InitialTablespace (Added in MySQL Cluster NDB 6.2.17, 6.3.22, and 6.4.3)	string (see description for details)	*none*	N/A	N/A	S
LockExecuteThreadToCPU (Added in MySQL Cluster NDB 6.3.4; changed in MySQL Cluster NDB 7.0)	*MySQL Cluster NDB 6.3 and earlier*: integer; *MySQL Cluster NDB 7.0 and later*: string	*MySQL Cluster NDB 6.4.3 and earlier*: 65535; *MySQL Cluster NDB 7.0 and later*: N/A	*MySQL Cluster NDB 6.4.3 and earlier*: 0; *MySQL Cluster NDB 7.0 and later*: N/A	*MySQL Cluster NDB 6.4.3 and earlier*: 65535; *MySQL Cluster NDB 7.0 and later*: N/A	N
LockMaintThreadsToCPU (Added in MySQL Cluster NDB 6.3.4)	integer	*MySQL Cluster NDB 6.3*: 65535; *MySQL Cluster NDB 7.0 and later*: N/A	0	65535	N
LockPagesInMainMemory	*As of MyS*	0	0	*As of MySQL 5.1.15 and MySQL Cluster NDB*	N

	QL 5.1.15 and MySQL Cluster NDB			*6.1.1*: 2; *previously*: 1	

	: integer ; *previ-*				

	: true|false (1|0)				
LogLevelCheckpoint	integer	0	0	15	IN
LogLevelCongestion	integer	0	0	15	N
LogLevelConnection	integer	0	0	15	N
LogLevelError	integer	0	0	15	N
LogLevelInfo	integer	0	0	15	N
LogLevelNodeRestart	integer	0	0	15	N
LogLevelShutdown	integer	0	0	15	N
LogLevelStartup	integer	1	0	15	N
LogLevelStatistic	integer	0	0	15	N
LongMessageBuffer	bytes	1M	*MySQL Cluster 6.4.3 and earlier*: 1M; *MySQL Cluster NDB 7.0.4 and later*: 4M	4294967039	N
MaxAllocate (Added in MySQL Cluster NDB 6.1.12 and MySQL Cluster NDB 6.2.3)	integer (bytes)	32M	1M	1G	N
MaxBufferedEpochs (Added in MySQL Cluster NDB 6.2.14)	integer	100	0	100000	N
MaxLCPStartDelay (Added in MySQL Cluster NDB 6.3.23 and MySQL Cluster NDB 6.4.3)	integer (seconds)	0	0	600	N
MaxNoOfAttributes	integer	1000	32	4294967039	N
MaxNoOfConcurrentIndexOperations	integer	8K	0	4294967039	N
MaxNoOfConcurrentOperations	integer	32768	32	4294967039	N
MaxNoOfConcurrentScans	integer	256	2	500	N
MaxNoOfConcurrentTransactions	integer	4096	32	4294967039	S
MaxNoOfExecutionThreads (added in MySQL Cluster NDB 6.4.0; *applies to ndbmtd only*)	integer	(as of MySQL Cluster NDB 7.0.4:) 2 (previously: *none*)	2	8	N
MaxNoOfFiredTriggers	integer	4000	0	4294967039	N
MaxNoOfIndexes (*DEPRECATED* — use MaxNoOfOrderedIndexes or MaxNoOfUniqueHashIndexes instead)	integer	128	0	4294967039	N
MaxNoOfLocalOperations	integer	UNDEFINED	32	4294967039	N
MaxNoOfLocalScans	integer	UNDEFINED (see description)	32	4294967039	N
MaxNoOfOpenFiles	integer	0 (prior to MySQL	20	4294967039	N

	teger	5.1.16: 40)			
MaxNoOfOrderedIndexes	integer	128	0	4294967039	N
MaxNoOfSavedMessages	integer	25	0	4294967039	N
MaxNoOfTables	integer	128	8	4294967039	N
MaxNoOfTriggers	integer	768	0	4294967039	N
MaxNoOfUniqueHashIndexes	integer	64	0	4294967039	N
MemReportFrequency (Added in MySQL 5.1.16 and MySQL Cluster NDB 6.1.0)	integer (seconds)	0	0	4294967039	N
NodeGroup (Added in MySQL Cluster NDB 6.4.0)	integer	UNDEFINED (normally determined by management server)	0	65535	SI
NoOfDiskPagesToDiskAfterRestartACC (*DEPRECATED* as of MySQL 5.1.6)	integer (number of 8KB pages per 100 milliseconds)	20 (= 20 * 80KB = 1.6MB/second)	1	4294967039	N
NoOfDiskPagesToDiskAfterRestartTUP (*DEPRECATED* as of MySQL 5.1.6)	integer (number of 8KB pages per 100 milliseconds)	40 (= 40 * 80KB = 3.2MB/second)	1	4294967039	N
NoOfDiskPagesToDiskDuringRestartACC (*DEPRECATED* as of MySQL 5.1.6)	integer (number of 8KB pages per 100 milliseconds)	20 (= 20 * 80KB = 1.6MB/second)	1	4294967039	N
NoOfDiskPagesToDiskDuringRestartTUP (*DEPRECATED* as of MySQL 5.1.6)	integer (number of 8KB pages per	40 (= 40 * 80KB = 3.2MB/second)	1	4294967039	N

	100 milliseconds)				
NoOfFragmentLogFiles	integer	16	3	4294967039	IN
NoOfReplicas	integer	*None*	1	4 (theoretical); 2 (supported)	IS
ODirect	boolean	0	0	1	N
RealTimeScheduler (Added in MySQL Cluster NDB 6.3.4)	boolean	0	0	1	N
RedoBuffer	bytes	*MySQL Cluster 6.4.3 and earlier*: 8M; *MySQL Cluster NDB 7.0.4 and later*: 32M	1M	4294967039	N
ReservedSendBufferMemory (added in MySQL Cluster NDB 6.4.0)	bytes	*None*	256K	4294967039	N
RestartOnErrorInsert (*DEBUG BUILDS ONLY*)	true|false (1|0)	0	0	1	N
SchedulerExecutionTimer (added in MySQL Cluster NDB 6.3.4)	seconds (integer)	50	0	11000	N
SchedulerSpinTimer (added in MySQL Cluster NDB 6.3.4)	seconds (integer)	0	0	500	N
ServerPort (*OBSOLETE*)	integer	1186	0	4294967039	N
SharedGlobalmemory (added in MySQL 5.1.6)	bytes	20M	0	65536G	N
StartFailureTimeout	milliseconds	0	0	4294967039	N
StartPartialTimeout	milliseconds	30000	0	4294967039	N
StartPartitionedTimeout	milliseconds	60000	0	4294967039	N
StopOnError	true|false (1|0)	1	0	1	N
StringMemory	integer or percentage (see description for de-	0	0	4294967039	S

	tails)				
TcpBind_INADDR_ANY (Added in MySQL Cluster NDB 6.2.0)	true|false (1|0)	1	0	0	N
TimeBetweenEpochs (Added in MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.2)	milliseconds	100	0	32000	N
TimeBetweenEpochsTimeout (Added in MySQL Cluster NDB 6.2.7 and MySQL Cluster NDB 6.3.4)	milliseconds	4000	0	32000	N
TimeBetweenGlobalCheckpoints	milliseconds	2000	10	32000	N
TimeBetweenInactiveTransactionAbortCheck	milliseconds	1000	1000	4294967039	N
TimeBetweenLocalCheckpoints	integer (number of 4-byte words as a base-2 logarithm)	20 (= 4 * 2^{20} = 4MB write operations)	0	31	N
TimeBetweenWatchDogCheck	milliseconds	6000	70	4294967039	N
TimeBetweenWatchDogCheckInitial (added in MySQL 5.1.20)	milliseconds	6000	70	4294967039	N
TotalSendBufferMemory (added in MySQL Cluster NDB 6.4.0)	bytes	*None*	256K	4294967039	N
TransactionBufferMemory	bytes	1M	1K	4294967039	N
TransactionDeadlockDetectionTimeout	milliseconds	1200	50 (*Note*: Prior to MySQL Cluster NDB 6.2.18/6.3.24/7.0.5, 100 was the effective minimum.)	4294967039	N
TransactionInactiveTimeout	milliseconds	0	0	4294967039	N
UndoDataBuffer (*OBSOLETE*)	bytes	16M	1M	4294967039	N
UndoIndexBuffer (*OBSOLETE*)	bytes	2M	1M	4294967039	N

Note

To add new data nodes to a MySQL Cluster, it is necessary to shut down the cluster completely, update the config.ini file, and then restart the cluster (that is, you must perform a system restart). All data node processes must be started with the --initial option.

Beginning in MySQL Cluster NDB 7.0, it is possible to add new data node groups to a running cluster online. For more information, see Section 7.8, "Adding MySQL Cluster Data Nodes Online".

3.5.2. MySQL Cluster Management Node Configuration Parameters

The following table provides information about parameters used in the `[ndb_mgmd]` or `[mgm]` sections of a `config.ini` file for configuring MySQL Cluster management nodes. For detailed descriptions and other additional information about each of these parameters, see Section 3.4.5, "Defining a MySQL Cluster Management Server".

Restart Type *Column Values*

- `N`: Node Restart
- `IN`: Initial Node Restart
- `S`: System Restart
- `IS`: Initial System Restart

See Section 3.5, "Overview of MySQL Cluster Configuration Parameters", for additional explanations of these abbreviations.

Table 3.2. MySQL Cluster Management Node Configuration Parameters

Parameter Name	Type / Units	Default Value	Minimum Value	Maximum Value	Restart Type
`ArbitrationDelay`	milliseconds	0	0	4294967039	N
`ArbitrationRank`	integer	1	0	2	N
`DataDir`	string	`./` (`ndb_mgmd` directory)	N/A	N/A	IN
`ExecuteOnComputer`	integer				
`HostName`	string	`localhost`	N/A	N/A	IN
`Id`	integer	*None*	1	(*MySQL Cluster NDB 6.1.0 and earlier*:) 63; (*MySQL Cluster NDB 6.1.1 and later*:) 255	IS
`LogDestination`	`CONSOLE`, `SYSLOG`, or `FILE`	`FILE` (see Section 3.4.5, "Defining a MySQL Cluster Management Server")	N/A	N/A	N
`PortNumber`	integer	1186	1	65535	S
`TotalSendBufferMemory` (added in MySQL Cluster NDB 6.4.0)	bytes	*None*	256K	4294967039	N

Note

After making changes in a management node's configuration, it is necessary to perform a rolling restart of the cluster in order for the new configuration to take effect. See Section 3.4.5, "Defining a MySQL Cluster Management Server", for more information.

To add new management servers to a running MySQL Cluster, it is also necessary perform a rolling restart of all cluster nodes after modifying any existing `config.ini` files. For more information about issues arising when using multiple management nodes, see Section 12.10, "Limitations Relating to Multiple MySQL Cluster Nodes".

3.5.3. MySQL Cluster SQL Node and API Node Configuration Parameters

The following table provides information about parameters used in the `[SQL]` and `[api]` sections of a `config.ini` file for configuring MySQL Cluster SQL nodes and API nodes. For detailed descriptions and other additional information about each of these parameters, see Section 3.4.7, "Defining SQL and Other API Nodes in a MySQL Cluster".

Note

For a discussion of MySQL server options for MySQL Cluster, see Section 4.2, "`mysqld` Command Options for MySQL Cluster"; for information about MySQL server system variables relating to MySQL Cluster, see Section 4.3, "MySQL Cluster System Variables".

Restart Type *Column Values*

- `N`: Node Restart
- `IN`: Initial Node Restart
- `S`: System Restart
- `IS`: Initial System Restart

See Section 3.5, "Overview of MySQL Cluster Configuration Parameters", for additional explanations of these abbreviations.

Table 3.3. MySQL Cluster SQL Node and API Node Configuration Parameters

Parameter Name	Type / Units	Default Value	Minimum Value	Maximum Value	Restart Type
`ArbitrationDelay`	milliseconds	0	0	4294967039	N
`ArbitrationRank`	integer	0	0	2	N
`BatchByteSize`	bytes	32K	1K	1M	N
`BatchSize`	integer	64	1	992	N
`ExecuteOnComputer`	integer				
`HostName`	string	*none*	N/A	N/A	IN
`Id`	integer	*None*	1	(*MySQL Cluster NDB 6.1.0 and earlier*:) 63; (*MySQL Cluster NDB 6.1.1 and later*:) 255	IS
`MaxScanBatchSize`	bytes	256K	32K	16M	N
`TotalSendBufferMemory` (added in MySQL Cluster NDB 6.4.0)	bytes	*None*	256K	4294967039	N

Note

To add new SQL or API nodes to the configuration of a running MySQL Cluster, it is necessary to perform a rolling restart of all cluster nodes after adding new `[mysqld]` or `[api]` sections to the `config.ini` file (or files, if you are using more than one management server). This must be done before the new SQL or API nodes can connect to the cluster.

It is *not* necessary to perform any restart of the cluster if new SQL or API nodes can employ previously unused API slots in the cluster configuration to connect to the cluster.

3.6. Configuring MySQL Cluster Parameters for Local Checkpoints

The parameters discussed in Logging and Checkpointing and in Data Memory, Index Memory, and String Memory that are used to configure local checkpoints for a MySQL Cluster do not exist in isolation, but rather are very much interdepedent on each other. In this section, we illustrate how these parameters — including `DataMemory`, `IndexMemory`, `NoOfDiskPagesToDiskAfterRestartTUP`, `NoOfDiskPagesToDiskAfterRestartACC`, and `NoOfFragmentLogFiles` — relate to one another in a working Cluster.

Important

The parameters `NoOfDiskPagesToDiskAfterRestartTUP` and `NoOfDiskPagesToDiskAfterRestartACC` were deprecated in MySQL 5.1.6. From MySQL 5.1.6 through 5.1.11, disk writes during LCPs took place at the maximum speed possible. Beginning with MySQL 5.1.12, the speed and throughput for LCPs are controlled using the parameters `DiskSyncSize`, `DiskCheckpointSpeed`, and `DiskCheckpointSpeedInRestart`. See Section 3.4.6, “Defining MySQL Cluster Data Nodes”.

In this example, we assume that our application performs the following numbers of types of operations per hour:

- 50000 selects
- 15000 inserts
- 15000 updates
- 15000 deletes

We also make the following assumptions about the data used in the application:

- We are working with a single table having 40 columns.
- Each column can hold up to 32 bytes of data.
- A typical `UPDATE` run by the application affects the values of 5 columns.
- No `NULL` values are inserted by the application.

A good starting point is to determine the amount of time that should elapse between local checkpoints (LCPs). It is worth noting that, in the event of a system restart, it takes 40-60 percent of this interval to execute the REDO log — for example, if the time between LCPs is 5 minutes (300 seconds), then it should take 2 to 3 minutes (120 to 180 seconds) for the REDO log to be read.

The maximum amount of data per node can be assumed to be the size of the `DataMemory` parameter. In this example, we assume that this is 2 GB. The `NoOfDiskPagesToDiskAfterRestartTUP` parameter represents the amount of data to be checkpointed per unit time — however, this parameter is actually expressed as the number of 8K memory pages to be checkpointed per 100 milliseconds. 2 GB per 300 seconds is approximately 6.8 MB per second, or 700 KB per 100 milliseconds, which works out to roughly 85 pages per 100 milliseconds.

Similarly, we can calculate `NoOfDiskPagesToDiskAfterRestartACC` in terms of the time for local checkpoints and the amount of memory required for indexes — that is, the `IndexMemory`. Assuming that we allow 512 MB for indexes, this works out to approximately 20 8-KB pages per 100 milliseconds for this parameter.

Next, we need to determine the number of REDO log files required — that is, fragment log files — the corresponding parameter being `NoOfFragmentLogFiles`. We need to make sure that there are sufficient REDO log files for keeping records for at least 3 local checkpoints (in MySQL Cluster NDB 6.3.8 and later, we need only allow for 2 local checkpoints). In a production setting, there are always uncertainties — for instance, we cannot be sure that disks always operate at top speed or with maximum throughput. For this reason, it is best to err on the side of caution, so we double our requirement and calculate a number of fragment log files which should be enough to keep records covering 6 local checkpoints (in MySQL Cluster NDB 6.3.8 and later, a number of fragment log files accommodating 4 local checkpoints should be sufficient).

It is also important to remember that the disk also handles writes to the REDO log, so if you find that the amount of data being written to disk as determined by the values of `NoOfDiskPagesToDiskAfterRestartACC` and `NoOfDiskPagesToDiskAfterRestartTUP` is approaching the amount of disk bandwidth available, you may wish to increase the time between local checkpoints.

Given 5 minutes (300 seconds) per local checkpoint, this means that we need to support writing log records at maximum speed for 6 * 300 = 1800 seconds (*MySQL Cluster NDB 6.3.8 and later*: 4 * 300 = 1200 seconds). The size of a REDO log record is 72 bytes plus 4 bytes per updated column value plus the maximum size of the updated column, and there is one REDO log record for each table record updated in a transaction, on each node where the data reside. Using the numbers of operations set out previously in this section, we derive the following:

- 50000 select operations per hour yields 0 log records (and thus 0 bytes), since `SELECT` statements are not recorded in the REDO log.
- 15000 `DELETE` statements per hour is approximately 5 delete operations per second. (Since we wish to be conservative in our estimate, we round up here and in the following calculations.) No columns are updated by deletes, so these statements consume only 5 operations * 72 bytes per operation = 360 bytes per second.
- 15000 `UPDATE` statements per hour is roughly the same as 5 updates per second. Each update uses 72 bytes, plus 4 bytes per column * 5 columns updated, plus 32 bytes per column * 5 columns — this works out to 72 + 20 + 160 = 252 bytes per operation, and multiplying this by 5 operation per second yields 1260 bytes per second.
- 15000 `INSERT` statements per hour is equivalent to 5 insert operations per second. Each insert requires REDO log space of 72 bytes, plus 4 bytes per record * 40 columns, plus 32 bytes per column * 40 columns, which is 72 + 160 + 1280 = 1512 bytes per operation. This times 5 operations per second yields 7560 bytes per second.

So the total number of REDO log bytes being written per second is approximately 0 + 360 + 1260 + 7560 = 9180 bytes. Multiplied by 1800 seconds, this yields 16524000 bytes required for REDO logging, or approximately 15.75 MB. The unit used for `NoOfFragmentLogFiles` represents a set of 4 16-MB log files — that is, 64 MB. Thus, the minimum value (3) for this parameter is sufficient for the scenario envisioned in this example, since 3 times 64 = 192 MB, or about 12 times what is required; the default value of 8 (or 512 MB) is more than ample in this case.

3.7. Configuring MySQL Cluster Send Buffer Parameters

Formerly, the NDB kernel used a send buffer whose size was fixed at 2MB for every node in the cluster, which was allocated when the node started. Because the size of this buffer could not be changed after the cluster was started, it was necessary to make it large enough in advance to accomodate the maximum possible load on any transporter socket. However, this was an inefficient use of memory, since much of it often went unused, and could result in large amounts of resources being wasted when scaling up to many API nodes.

Beginning with MySQL Cluster NDB 7.0, this problem is solved by employing a unified send buffer whose memory is allocated dynamically from a pool shared by all transporters. This means that the size of the send buffer can be adjusted as necessary. Configuration of the unified send buffer can accomplished by setting the following parameters:

- **`TotalSendBufferMemory`.** This parameter can be set for all types of MySQL Cluster nodes — that is, it can be set in the `[ndbd]`, `[mgm]`, and `[api]` (or `[mysql]`) sections of the `config.ini` file. It represents the total amount of memory (in bytes) to be allocated by each node for which it is set for use among all configured transporters. If set, its minimum is 256K; the maximum is 4294967039.

  In order to be backward-compatible with existing configurations, this parameter takes as its default value the sum of the maximum send buffer sizes of all configured transporters, plus an additional 32KB (one page) per transporter. The maximum depends on the type of transporter, as shown in the following table:

Transporter	Maxmimum Send Buffer Size (bytes)
TCP	`SendBufferMemory` (default = 2M)
SCI	`SendLimit` (default = 8K) plus 16K
SHM	20K

  This allows existing configurations to function in close to the same way as they did with MySQL Cluster NDB 6.3 and earlier, with the same amount of memory and send buffer space available to each transporter. However, memory that is unused by one transporter is not available to other transporters.

- **`ReservedSendBufferMemory`.** This optional data node parameter, if set, gives an amount of memory (in bytes) that is reserved for connections between data nodes; this memory is not allocated to send buffers used for communications with management servers or API nodes. This provides a way to protect the cluster against misbehaving API nodes that use excess send memory and thus cause failures in communications internally in the NDB kernel. If set, its the minimum permitted value for this parameters is 256K; the maximum is 4294967039.
- **`OverloadLimit`.** This parameter is used in the `config.ini` file `[tcp]` section, and denotes the amount of unsent data (in bytes) that must be present in the send buffer before the connection is considered overloaded. When such an overload condi-

tion occurs, transactions that affect the overloaded connection fail with NDB API Error 1218 (SEND BUFFERS OVERLOADED IN NDB KERNEL) until the overload status passes. The default value is 0; there is no defined maximum value for this parameter.

- **SendBufferMemory.** In MySQL Cluster NDB 6.3 and earlier, this TCP configuration parameter represented the amount of memory allocated at startup for each configured TCP connection. Beginning with MySQL Cluster NDB 7.0, this value denotes a hard limit for how much memory (out of the total available — that is, `TotalSendBufferMemory`) that may be used by a single transporter. However, the sum of `TotalSendBufferMemory` for all configured transporters may be greater than `SendBufferMemory`. This is a way to save memory when many nodes are in use, as long as the maximum amount of memory is never required by all transporters at the same time.

Chapter 4. MySQL Cluster Options and Variables

This section provides information about MySQL server options, server and status variables that are specific to MySQL Cluster. For general information on using these, and for other options and variables not specific to MySQL Cluster, see The MySQL Server.

For MySQL Cluster configuration parameters used in the cluster confiuration file (usually named `config.ini`), see Chapter 3, *MySQL Cluster Configuration*.

4.1. MySQL Cluster Server Option and Variable Reference

The following table provides a list of the command-line options, server and status variables applicable within `mysqld` when it is running as an SQL node in a MySQL Cluster. For a table showing *all* command-line options, server and status variables available for use with `mysqld`, see Server Option and Variable Reference.

Table 4.1. `mysqld` Command Options for MySQL Cluster

Name	Cmd-Line	Option file	System Var	Status Var	Var Scope	Dynamic
Handler_discover				Yes	Both	No
have_ndbcluster			Yes		Global	No
ndb_autoincrement_prefetch_sz	Yes	Yes	Yes		Both	Yes
ndb-batch-size	Yes	Yes	Yes		Global	No
ndb_cache_check_time	Yes	Yes	Yes		Global	Yes
ndb-cluster-connection-pool	Yes	Yes		Yes	Global	No
Ndb_cluster_node_id				Yes	Both	No
Ndb_config_from_host				Yes	Both	No
Ndb_config_from_port				Yes	Both	No
Ndb_conflict_fn_max				Yes	Global	No
Ndb_conflict_fn_old				Yes	Global	No
ndb-connectstring	Yes	Yes				
ndb_execute_count				Yes	Global	No
ndb_extra_logging	Yes	Yes	Yes		Global	Yes
ndb_force_send	Yes	Yes	Yes		Both	Yes
ndb_index_stat_cache_entries	Yes	Yes				
ndb_index_stat_enable	Yes	Yes				
ndb_index_stat_update_freq	Yes	Yes				
ndb_log_empty_epochs	Yes	Yes	Yes		Global	Yes
ndb_log_orig			Yes		Global	No
ndb-log-update-as-write	Yes	Yes	Yes		Global	Yes
ndb_log_updated_only	Yes	Yes	Yes		Global	Yes
ndb-mgmd-host	Yes	Yes				
ndb-nodeid	Yes	Yes		Yes	Global	No
Ndb_number_of_data_nodes				Yes	Global	No
ndb_optimization_delay			Yes		Global	Yes
ndb_optimized_node_selection	Yes	Yes				
ndb_pruned_scan_count				Yes	Global	No
ndb_report_thresh_binlog_epoch_slip	Yes	Yes				
ndb_report_thresh_binlog_mem_usage	Yes	Yes				
ndb_scan_count				Yes	Global	No
ndb_table_no_logging			Yes		Session	Yes
ndb_table_temporary			Yes		Session	Yes
ndb_use_copying_alter_table			Yes		Both	No

Name	Cmd-Line	Option file	System Var	Status Var	Var Scope	Dynamic
ndb_use_exact_count			Yes		Both	Yes
ndb_use_transactions	Yes	Yes				
ndb_wait_connected	Yes	Yes	Yes			No
ndbcluster	Yes	Yes				
skip-ndbcluster	Yes	Yes				
slave-allow-batching	Yes	Yes			Global	Yes
- *Variable*: slave_allow_batching			Yes		Global	Yes

4.2. `mysqld` Command Options for MySQL Cluster

This section provides descriptions of `mysqld` server options relating to MySQL Cluster. For information about `mysqld` options not specific to MySQL Cluster, and for general information about the use of options with `mysqld`, see Server Command Options.

For information about command-line options used with other MySQL Cluster processes (`ndbd`, `ndb_mgmd`, and `ndb_mgm`), see Section 6.23, "Options Common to MySQL Cluster Programs". For information about command-line options used with `NDB` utility programs (such as `ndb_desc`, `ndb_size.pl`, and `ndb_show_tables`), see Chapter 6, *MySQL Cluster Programs*.

- `--ndb-batch-size=#`

Version Introduced	5.1.23-ndb-6.3.8	
Command Line Format	`--ndb-batch-size`	
Config File Format	`ndb-batch-size`	
Variable Name	`ndb_batch_size`	
Variable Scope	Global	
Value Set	**Type**	`numeric`
	Default	`32768`
	Range	`0-31536000`

This sets the size in bytes that is used for NDB transaction batches.

- `--ndb-cluster-connection-pool=#`

Version Introduced	5.1.19-ndb-6.2.2	
Command Line Format	`--ndb-cluster-connection-pool`	
Config File Format	`ndb-cluster-connection-pool`	
Variable Name	`ndb-cluster-connection-pool`	
Variable Scope	Global	
Dynamic Variable	No	
Value Set	**Type**	`numeric`
	Default	`1`
	Range	`1-63`

By setting this option to a value greater than 1 (the default), a `mysqld` process can use multiple connections to the cluster, effectively mimicking several SQL nodes. Each connection requires its own `[api]` or `[mysqld]` section in the cluster configuration (`config.ini`) file, and counts against the maximum number of API connections supported by the cluster.

For example, suppose that you have 2 cluster host computers, each running an SQL node whose `mysqld` process was started with `--ndb-cluster-connection-pool=4`; this means that the cluster must have 8 API slots available for these connections (instead of 2). All of these connections are set up when the SQL node connects to the cluster, and are allocated to threads in a round-robin fashion.

This option is useful only when running `mysqld` on host machines having multiple CPUs, multiple cores, or both. For best results, the value should be smaller than the total number of cores available on the host machine. Setting it to a value greater than this is likely to degrade performance severely.

> **Important**
>
> Because each SQL node using connection pooling occupies multiple API node slots — each slot having its own node ID in the cluster — you must *not* use a node ID as part of the cluster connect string when starting any `mysqld` process that employs connection pooling.
>
> Setting a node ID in the connect string when using the `--ndb-cluster-connection-pool` option causes node ID allocation errors when the SQL node attempts to connect to the cluster.

This option was introduced in MySQL Cluster NDB 6.2.2. Beginning with MySQL Cluster NDB 6.2.16 and MySQL Cluster NDB 6.3.13, the value used for this option is available as a global status variable (Bug#35573).

- `--ndb-connectstring=`*`connect_string`*

Command Line Format	`--ndb-connectstring`	
Config File Format	`ndb-connectstring`	
Value Set	**Type**	`string`

When using the `NDBCLUSTER` storage engine, this option specifies the management server that distributes cluster configuration data. See Section 3.4.3, “The MySQL Cluster Connectstring”, for syntax.

- `--ndbcluster`

Command Line Format	`--ndbcluster`	
Config File Format	`ndbcluster`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

The `NDBCLUSTER` storage engine is necessary for using MySQL Cluster. If a `mysqld` binary includes support for the `NDBCLUSTER` storage engine, the engine is disabled by default. Use the `--ndbcluster` option to enable it. Use `--skip-ndbcluster` to explicitly disable the engine.

- `--ndb-nodeid=#`

Version Introduced	5.1.15	
Command Line Format	`--ndb-nodeid=#`	
Config File Format	`ndb-nodeid`	
Variable Name	`Ndb_cluster_node_id`	
Variable Scope	Global	
Dynamic Variable	No	
Value Set (>= 5.1.5)	**Type**	`numeric`
	Range	`1-255`

Set this MySQL server's node ID in a MySQL Cluster. This can be used instead of specifying the node ID as part of the connectstring or in the `config.ini` file, or allowing the cluster to determine an arbitrary node ID. If you use this option, then `--ndb-nodeid` must be specified *before* `--ndb-connectstring`. If `--ndb-nodeid` is used *and* a node ID is specified in the connectstring, then the MySQL server will not be able to connect to the cluster. In addition, if `--nodeid` is used, then either a matching node ID must be found in a `[mysqld]` or `[api]` section of `config.ini`, or there must be an “open” `[mysqld]` or `[api]` section in the file (that is, a section without an `Id` parameter specified).

Regardless of how the node ID is determined, its is shown as the value of the global status variable `Ndb_cluster_node_id` in the output of `SHOW STATUS`, and as `cluster_node_id` in the `connection` row of the output of `SHOW ENGINE NDBCLUSTER STATUS`.

For more information about node IDs for MySQL Cluster SQL nodes, see Section 3.4.7, “Defining SQL and Other API Nodes

in a MySQL Cluster".

- `--skip-ndbcluster`

Command Line Format	`--skip-ndbcluster`
Config File Format	`skip-ndbcluster`

Disable the `NDBCLUSTER` storage engine. This is the default for binaries that were built with `NDBCLUSTER` storage engine support; the server allocates memory and other resources for this storage engine only if the `--ndbcluster` option is given explicitly. See Section 3.3, "Quick Test Setup of MySQL Cluster", for an example.

4.3. MySQL Cluster System Variables

This section provides detailed information about MySQL server system variables that are specific to MySQL Cluster and the `NDB` storage engine. For system variables not specific to MySQL Cluster, see Server System Variables. For general information on using system variables, see Using System Variables.

- `have_ndbcluster`

Variable Name	`have_ndbcluster`	
Variable Scope	Global	
Dynamic Variable	No	
Value Set	**Type**	`boolean`

`YES` if `mysqld` supports `NDBCLUSTER` tables. `DISABLED` if `--skip-ndbcluster` is used.

- `multi_range_count`

Command Line Format	`--multi_range_count=#`	
Config File Format	`multi_range_count`	
Option Sets Variable	Yes, `multi_range_count`	
Variable Name	`multi_range_count`	
Variable Scope	Both	
Dynamic Variable	Yes	
Value Set	**Type**	`numeric`
	Default	`256`
	Range	`1-4294967295`

The maximum number of ranges to send to a table handler at once during range selects. The default value is 256. Sending multiple ranges to a handler at once can improve the performance of certain selects dramatically. This is especially true for the `NDBCLUSTER` table handler, which needs to send the range requests to all nodes. Sending a batch of those requests at once reduces communication costs significantly.

This variable is deprecated in MySQL 5.1, and is no longer supported in MySQL 6.0, in which arbitrarily long lists of ranges can be processed.

- `ndb_autoincrement_prefetch_sz`

Command Line Format	`--ndb_autoincrement_prefetch_sz`	
Config File Format	`ndb_autoincrement_prefetch_sz`	
Option Sets Variable	Yes, `ndb_autoincrement_prefetch_sz`	
Variable Name	`ndb_autoincrement_prefetch_sz`	
Variable Scope	Both	
Dynamic Variable	Yes	
Value Set (<= 5.1.22)	**Type**	`numeric`

	Default	32
	Range	1-256
Value Set (>= 5.1.23)	**Type**	numeric
	Default	1
	Range	1-256

Determines the probability of gaps in an autoincremented column. Set it to 1 to minimize this. Setting it to a high value for optimization — makes inserts faster, but decreases the likelihood that consecutive autoincrement numbers will be used in a batch of inserts. Default value: 32. Minimum value: 1.

Beginning with MySQL Cluster NDB 6.2.10, MySQL Cluster NDB 6.3.7, and MySQL 5.1.23, this variable affects the number of AUTO_INCREMENT IDs that are fetched between statements only. Within a statement, at least 32 IDs are now obtained at a time. The default value for ndb_autoincrement_prefetch_sz is now 1, to increase the speed of statements inserting single rows. (Bug#31956)

- ndb_cache_check_time

Command Line Format	--ndb_cache_check_time	
Config File Format	ndb_cache_check_time	
Option Sets Variable	Yes, ndb_cache_check_time	
Variable Name	ndb_cache_check_time	
Variable Scope	Global	
Dynamic Variable	Yes	
Value Set	**Type**	numeric
	Default	0

The number of milliseconds that elapse between checks of MySQL Cluster SQL nodes by the MySQL query cache. Setting this to 0 (the default and minimum value) means that the query cache checks for validation on every query.

The recommended maximum value for this variable is 1000, which means that the check is performed once per second. A larger value means that the check is performed and possibly invalidated due to updates on different SQL nodes less often. It is generally not desirable to set this to a value greater than 2000.

- ndb_extra_logging

Version Introduced	5.1.6	
Command Line Format	ndb_extra_logging=#	
Config File Format	ndb_extra_logging	
Variable Name	ndb_extra_logging	
Variable Scope	Global	
Dynamic Variable	Yes	
Value Set	**Type**	numeric
	Default	0

This variable can be used to enable recording in the MySQL error log of information specific to the NDB storage engine. It is normally of interest only when debugging NDB storage engine code.

The default value is 0, which means that the only NDB-specific information written to the MySQL error log relates to transaction handling. If the value is greater than 0 but less than 10, NDB table schema and connection events are also logged, as well as whether or not conflict resolution is in use, and other NDB errors and information. If the value is set to 10 or more, information about NDB internals, such as the progress of data distribution among cluster nodes, is also written to the MySQL error log.

This variable was added in MySQL 5.1.6.

- ndb_force_send

Command Line Format	`--ndb-force-send`	
Config File Format	`ndb_force_send`	
Option Sets Variable	Yes, `ndb_force_send`	
Variable Name	`ndb_force_send`	
Variable Scope	Both	
Dynamic Variable	Yes	
Value Set	**Type**	`boolean`
	Default	`TRUE`

Forces sending of buffers to `NDB` immediately, without waiting for other threads. Defaults to `ON`.

- `ndb_index_stat_cache_entries`

Version Removed	5.1.14	
Command Line Format	`--ndb_index_stat_cache_entries`	
Config File Format	`ndb_index_stat_cache_entries`	
Value Set	**Type**	`numeric`
	Default	`32`
	Range	`0-4294967295`

Sets the granularity of the statistics by determining the number of starting and ending keys to store in the statistics memory cache. Zero means no caching takes place; in this case, the data nodes are always queried directly. Default value: `32`.

- `ndb_index_stat_enable`

Version Removed	5.1.19	
Command Line Format	`--ndb_index_stat_enable`	
Config File Format	`ndb_index_stat_enable`	
Value Set	**Type**	`boolean`
	Default	`ON`

Use `NDB` index statistics in query optimization. Defaults to `ON`.

- `ndb_index_stat_update_freq`

Version Removed	5.1.14	
Command Line Format	`--ndb_index_stat_update_freq`	
Config File Format	`ndb_index_stat_update_freq`	
Value Set	**Type**	`numeric`
	Default	`20`
	Range	`0-4294967295`

How often to query data nodes instead of the statistics cache. For example, a value of `20` (the default) means to direct every 20th query to the data nodes.

- `ndb_optimized_node_selection`

Command Line Format	`--ndb-optimized-node-selection`	
Config File Format	`ndb_optimized_node_selection`	
Value Set	**Type**	`boolean`
	Default	`ON`

Prior to MySQL Cluster NDB 6.3.4. Causes an SQL node to use the "closest" data node as transaction coordinator. Enabled by default. Set to `0` or `OFF` to disable, in which case the SQL node uses each data node in the cluster in succession. When this option is disabled each SQL thread attempts to use a given data node 8 times before proceeding to the next one.

Beginning with MySQL Cluster NDB 6.3.4. There are two forms of optimized node selection:

1. The SQL node uses *promixity* to determine the transaction coordinator; that is, the "closest" data node to the SQL node is chosen as the transaction coordinator. For this purpose, a data node having a shared memory connection with the SQL node is considered to be "closest" to the SQL node; the next closest (in order of decreasing proximity) are: TCP connection to `localhost`; SCI connection; TCP connection from a host other than `localhost`.

2. The SQL thread uses *distribution awareness* to select the data node. That is, the data node housing the cluster partition accessed by the first statement of a given transaction is used as the transaction coordinator for the entire transaction. (This is effective only if the first statement of the transaction accesses no more than one cluster partition.)

This option takes one of the integer values `0`, `1`, `2`, or `3`. `3` is the default. These values affect node selection as follows:

- `0`: Node selection is not optimized. Each data node is employed as the transaction coordinator 8 times before the SQL thread proceeds to the next data node. (This is the same "round-robin" behavior as caused by setting this option to `0` or `OFF` in previous versions of MySQL Cluster.)

- `1`: Proximity to the SQL node is used to determine the transaction coordinator. (This is the same behavior as caused by setting this option to `1` or `ON` in previous MySQL versions.)

- `2`: Distribution awareness is used to select the transaction coordinator. However, if the first statement of the transaction accesses more than one cluster partition, the SQL node reverts to the round-robin behavior seen when this option is set to `0`.

- `3`: If distribution awareness can be employed to determine the transaction coordinator, then it is used; otherwise proximity is used to select the transaction coordinator. (This is the default behavior in MySQL Cluster NDB 6.3.4 and later.)

> **Important**
>
> Beginning with MySQL Cluster NDB 6.3.4, it is no longer possible to set `--ndb_optimized_node_selection` to `ON` or `OFF`; attempting to do so causes `mysqld` to abort with an error.

- `ndb_report_thresh_binlog_epoch_slip`

Command Line Format	`--ndb_report_thresh_binlog_epoch_slip`	
Config File Format	`ndb_report_thresh_binlog_epoch_slip`	
Value Set	**Type**	`numeric`
	Default	`3`
	Range	`0-256`

This is a threshold on the number of epochs to be behind before reporting binlog status. For example, a value of `3` (the default) means that if the difference between which epoch has been received from the storage nodes and which epoch has been applied to the binlog is 3 or more, a status message will be sent to the cluster log.

- `ndb_report_thresh_binlog_mem_usage`

Command Line Format	`--ndb_report_thresh_binlog_mem_usage`	
Config File Format	`ndb_report_thresh_binlog_mem_usage`	
Value Set	**Type**	`numeric`
	Default	`10`
	Range	`0-10`

This is a threshold on the percentage of free memory remaining before reporting binlog status. For example, a value of `10` (the default) means that if the amount of available memory for receiving binlog data from the data nodes falls below 10%, a status message will be sent to the cluster log.

- `ndb_use_copying_alter_table`

Version Introduced	5.1.12
Variable Name	`ndb_use_copying_alter_table`

Variable Scope	Both
Dynamic Variable	No

Forces NDB to use copying of tables in the event of problems with online ALTER TABLE operations. The default value is OFF.

This variable was added in MySQL 5.1.12.

- ndb_use_exact_count

Variable Name	ndb_use_exact_count	
Variable Scope	Both	
Dynamic Variable	Yes	
Value Set	**Type**	boolean
	Default	ON

Forces NDB to use a count of records during SELECT COUNT(*) query planning to speed up this type of query. The default value is ON. For faster queries overall, disable this feature by setting the value of ndb_use_exact_count to OFF.

- ndb_use_transactions

Command Line Format	--ndb_use_transactions	
Config File Format	ndb_use_transactions	
Value Set	**Type**	boolean
	Default	ON

You can disable NDB transaction support by setting this variable's values to OFF (not recommended). The default is ON.

> **Note**
>
> The setting for this variable was not honored in MySQL Cluster NDB 6.4.3 and MySQL Cluster NDB 7.0.4. (Bug#43236)

- ndb_wait_connected

Version Introduced	5.1.16-ndb-6.2.0	
Command Line Format	ndb_wait_connected	
Config File Format	ndb_wait_connected	
Option Sets Variable	Yes, ndb_wait_connected	
Variable Name	ndb_wait_connected	
Variable Scope		
Dynamic Variable	No	
Value Set	**Type**	numeric
	Default	0

This variable can be used to cause the MySQL server to wait a given period of time for connections to MySQL Cluster management and data nodes to be established before accepting MySQL client connections. The time is specified in seconds. The default value is 0.

4.4. MySQL Cluster Status Variables

This section provides detailed information about MySQL server status variables that relate to MySQL Cluster and the NDB storage engine. For status variables not specific to MySQL Cluster, and for general information on using status variables, see Server Status Variables.

- `Handler_discover`

  The MySQL server can ask the `NDBCLUSTER` storage engine if it knows about a table with a given name. This is called discovery. `Handler_discover` indicates the number of times that tables have been discovered via this mechanism.

- `Ndb_cluster_node_id`

  If the server is acting as a MySQL Cluster node, then the value of this variable its node ID in the cluster.

  If the server is not part of a MySQL Cluster, then the value of this variable is 0.

- `Ndb_config_from_host`

  If the server is part of a MySQL Cluster, the value of this variable is the host name or IP address of the Cluster management server from which it gets its configuration data.

  If the server is not part of a MySQL Cluster, then the value of this variable is an empty string.

  Prior to MySQL 5.1.12, this variable was named `Ndb_connected_host`.

- `Ndb_config_from_port`

  If the server is part of a MySQL Cluster, the value of this variable is the number of the port through which it is connected to the Cluster management server from which it gets its configuration data.

  If the server is not part of a MySQL Cluster, then the value of this variable is 0.

  Prior to MySQL 5.1.12, this variable was named `Ndb_connected_port`.

- `Ndb_execute_count`

  Provides the number of round trips to the `NDB` kernel made by operations. Added in MySQL Cluster NDB 6.3.6.

- `Ndb_number_of_data_nodes`

  If the server is part of a MySQL Cluster, the value of this variable is the number of data nodes in the cluster.

  If the server is not part of a MySQL Cluster, then the value of this variable is 0.

  Prior to MySQL 5.1.12, this variable was named `Ndb_number_of_storage_nodes`.

- `Slave_heartbeat_period`

  Shows the replication heartbeat interval (in seconds) on a replication slave.

  This variable was added in MySQL Cluster NDB 6.3.4.

- `Slave_received_heartbeats`

  This counter increments with each replication heartbeat received by a replication slave since the last time that the slave was restarted or reset, or a `CHANGE MASTER TO` statement was issued.

  This variable was added in MySQL Cluster NDB 6.3.4.

- `Ndb_pruned_scan_count`

  This variable holds a count of the number of scans executed by `NDBCLUSTER` since the MySQL Cluster was last started where `NDBCLUSTER` was able to use partition pruning.

  Using this variable together with `Ndb_scan_count` can be helpful in schema design to maximize the ability of the server to prune scans to a single table partition, thereby involving only a single data node.

  This variable was added in MySQL Cluster NDB 6.3.25 and MySQL Cluster NDB 7.0.5.

- `Ndb_scan_count`

  This variable holds a count of the total number of scans executed by `NDBCLUSTER` since the MySQL Cluster was last started.

  This variable was added in MySQL Cluster NDB 6.3.25 and MySQL Cluster NDB 7.0.5.

Chapter 5. Upgrading and Downgrading MySQL Cluster

This portion of the MySQL Cluster chapter covers upgrading and downgrading a MySQL Cluster from one MySQL release to another. It discusses different types of Cluster upgrades and downgrades, and provides a Cluster upgrade/downgrade compatibility matrix (see Section 5.2, "MySQL Cluster 5.1 and MySQL Cluster NDB 6.x/7.x Upgrade and Downgrade Compatibility"). You are expected already to be familiar with installing and configuring a MySQL Cluster prior to attempting an upgrade or downgrade. See Chapter 3, *MySQL Cluster Configuration*.

For information about upgrading or downgrading between MySQL Cluster NDB releases, or between MySQL Cluster NDB releases and mainline MySQL releases, see the changelogs relating to MySQL Cluster NDB.

This section remains in development, and continues to be updated and expanded.

5.1. Performing a Rolling Restart of a MySQL Cluster

This section discusses how to perform a *rolling restart* of a MySQL Cluster installation, so called because it involves stopping and starting (or restarting) each node in turn, so that the cluster itself remains operational. This is often done as part of a *rolling upgrade* or *rolling downgrade*, where high availability of the cluster is mandatory and no downtime of the cluster as a whole is permissible. Where we refer to upgrades, the information provided here also generally applies to downgrades as well.

There are a number of reasons why a rolling restart might be desirable:

- **Cluster configuration change.** To make a change in the cluster's configuration, such as adding an SQL node to the cluster, or setting a configuration parameter to a new value.
- **Cluster software upgrade/downgrade.** To upgrade the cluster to a newer version of the MySQL Cluster software (or to downgrade it to an older version). This is usually referred to as a "rolling upgrade" (or "rolling downgrade", when reverting to an older version of MySQL Cluster).
- **Change on node host.** To make changes in the hardware or operating system on which one or more cluster nodes are running
- **Cluster reset.** To reset the cluster because it has reached an undesirable state
- **Freeing of resources.** To allow memory allocated to a table by successive `INSERT` and `DELETE` operations to be freed for reuse by other Cluster tables

The process for performing a rolling restart may be generalised as follows:

1. Stop all cluster management nodes (`ndb_mgmd` processes), reconfigure them, then restart them
2. Stop, reconfigure, then restart each cluster data node (`ndbd` process) in turn
3. Stop, reconfigure, then restart each cluster SQL node (`mysqld` process) in turn

The specifics for implementing a particular rolling upgrade depend upon the actual changes being made. A more detailed view of the process is presented here:

<table>
<tr><th colspan="5">RESTART TYPE:</th></tr>
<tr><th>Cluster Configuration Change</th><th>Cluster Software Upgrade or Downgrade</th><th>Change on Node Host</th><th colspan="2">Cluster Reset</th></tr>
<tr><td colspan="5">A. Management node (<code>ndb_mgmd</code>) processes...</td></tr>
<tr><td rowspan="2">1. Stop all <code>ndb_mgmd</code> processes

2. Make changes in global configuration file(s)

3. Start all <code>ndb_mgmd</code> processes</td><td rowspan="2">1. Stop all <code>ndb_mgmd</code> processes

2. Replace each <code>ndb_mgmd</code> binary with new version

3. Start <code>ndb_mgmd</code> processes</td><td rowspan="2">1. Stop all <code>ndb_mgmd</code> processes

2. Make desired changes in hardware, operating system, or both

3. Start all <code>ndb_mgmd</code> processes</td><td colspan="2">(OR)</td></tr>
<tr><td>1. Stop all <code>ndb_mgmd</code> processes

2. Start all <code>ndb_mgmd</code> processes</td><td>Restart all <code>ndb_mgmd</code> processes (optional)</td></tr>
<tr><td colspan="5">B. For each data node (<code>ndbd</code>) process...</td></tr>
<tr><td colspan="2">(OR)</td><td rowspan="2">1. Stop <code>ndbd</code>

2. Replace <code>ndbd</code> binary with new version

3. Start <code>ndbd</code></td><td rowspan="2">1. Stop <code>ndbd</code>

2. Make desired changes in hardware, operating system, or both

3. Start <code>ndbd</code></td><td colspan="2">(OR)</td></tr>
<tr><td>1. Stop <code>ndbd</code>

2. Start <code>ndbd</code></td><td>Restart <code>ndbd</code></td><td>1. Stop <code>ndbd</code>

2. Start <code>ndbd</code></td><td>Restart <code>ndbd</code></td></tr>
<tr><td colspan="5">C. For each SQL node (<code>mysqld</code>) process...</td></tr>
<tr><td colspan="2">(OR)</td><td rowspan="2">1. Stop <code>mysqld</code>

2. Replace <code>mysqld</code> binary with new version

3. Start <code>mysqld</code></td><td rowspan="2">1. Stop <code>mysqld</code>

2. Make desired changes in hardware, operating system, or both

3. Start <code>mysqld</code></td><td colspan="2">(OR)</td></tr>
<tr><td>1. Stop <code>mysqld</code>

2. Start <code>mysqld</code></td><td>Restart <code>mysqld</code></td><td>1. Stop <code>mysqld</code>

2. Start <code>mysqld</code></td><td>Restart <code>mysqld</code></td></tr>
</table>

In the previous diagram, **Stop** and **Start** steps indicate that the process must be stopped completely using a shell command (such as `kill` on most Unix systems) or the management client `STOP` command, then started again from a system shell by invoking the `ndbd` or `ndb_mgmd` executable as appropriate. **Restart** indicates the process may be restarted using the `ndb_mgm` management client `RESTART` command.

Important

When performing an upgrade or downgrade of the cluster software, you *must* upgrade or downgrade the management nodes *first*, then the data nodes, and finally the SQL nodes. Doing so in any other order may leave the cluster in an unusable state.

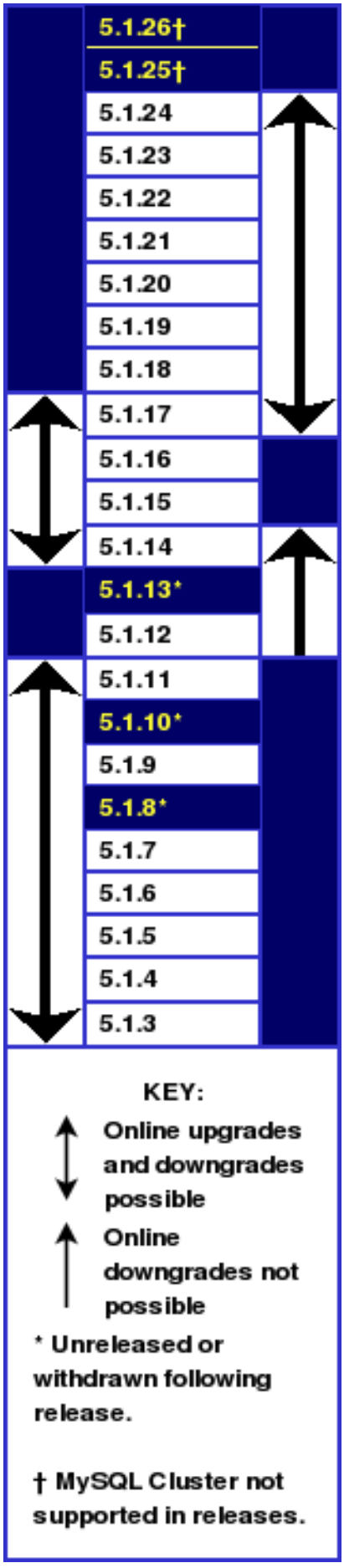
5.2. MySQL Cluster 5.1 and MySQL Cluster NDB 6.x/7.x Upgrade and

Downgrade Compatibility

This section provides information about MySQL Cluster software and table file compatibility between MySQL 5.1 and MySQL Cluster NDB 6.x releases with regard to performing upgrades and downgrades.

> **Important**
>
> Only compatibility between MySQL versions with regard to `NDBCLUSTER` is taken into account in this section, and there are likely other issues to be considered. *As with any other MySQL software upgrade or downgrade, you are strongly encouraged to review the relevant portions of the MySQL Manual for the MySQL versions from which and to which you intend to migrate, before attempting an upgrade or downgrade of the MySQL Cluster software*. See Upgrading MySQL.

The following table shows Cluster upgrade and downgrade compatibility between different releases of MySQL 5.1:


5.1.26†
5.1.25†
5.1.24
5.1.23
5.1.22
5.1.21
5.1.20
5.1.19
5.1.18
5.1.17
5.1.16
5.1.15
5.1.14
5.1.13*
5.1.12
5.1.11
5.1.10*
5.1.9
5.1.8*
5.1.7
5.1.6
5.1.5
5.1.4
5.1.3
KEY:
Online upgrades and downgrades possible
Online downgrades not possible
* Unreleased or withdrawn following release.
† MySQL Cluster not supported in releases.

Notes — MySQL 5.1.

- MySQL 5.1.3 was the first public release in this series.
- Direct upgrades or downgrades between MySQL Cluster 5.0 and 5.1 are not supported; you must dump all `NDBCLUSTER` tables using `mysqldump`, install the new version of the software, and then reload the tables from the dump.
- You cannot downgrade a MySQL 5.1.6 or later Cluster using Disk Data tables to MySQL 5.1.5 or earlier unless you convert all such tables to in-memory Cluster tables first.
- MySQL 5.1.8, MySQL 5.1.10, and MySQL 5.1.13 were not released.
- Online cluster upgrades and downgrades between MySQL 5.1.11 (or an earlier version) and 5.1.12 (or a later version) are not possible due to major changes in the cluster file system. In such cases, you must perform a backup or dump, upgrade (or downgrade) the software, start each data node with `--initial`, and then restore from the backup or dump. You can use `NDB` backup/restore or `mysqldump` for this purpose.
- Online downgrades from MySQL 5.1.14 or later to versions previous to 5.1.14 are not supported due to incompatible changes in the cluster system tables.
- Online upgrades from MySQL 5.1.17 and earlier to 5.1.18 and later are not supported for clusters using replication due to incompatible changes in the `mysql.ndb_apply_status` table. However, it should not be necessary to shut down the cluster entirely, if you follow this modified rolling restart procedure:

  1. Stop the management server, update the `ndb_mgmd` binary, then start it again. For multiple management servers, repeat this step for each management server in turn.
  2. For each data node in turn: Stop the data node, replace the `ndbd` binary with the new version, then restart the data node. It is not necessary to use `--initial` when restarting any of the data nodes.
  3. Stop *all* SQL nodes. Replace the `mysqld` binary with the new version for all SQL nodes, then restart them. It is not necessary to start them one at a time, but they must all be shut down at the same time before starting any of them again using the 5.1.18 (or later) `mysqld`. Otherwise — due to the fact that `mysql.ndb_apply_status` uses the `NDB` storage engine and is thus shared between all SQL nodes — there may be conflicts between MySQL servers using the old and new versions of the table.

  You can find more information about the changes to `ndb_apply_status` in Section 9.4, “MySQL Cluster Replication Schema and Tables”.
- The internal specifications for columns in `NDBCLUSTER` tables changed in MySQL 5.1.18 to allow compatibility with later MySQL Cluster releases that allow online adding and dropping of columns. *This change is not backward-compatible with earlier MySQL versions*.

  In order to make tables created in MySQL 5.1.17 and earlier compatible with online adding and dropping of columns (available beginning with beginning with MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.3 — see `ALTER TABLE` Syntax, for more information), it is necessary to force MySQL 5.1.18 and later to convert the tables to the new format by following this procedure:

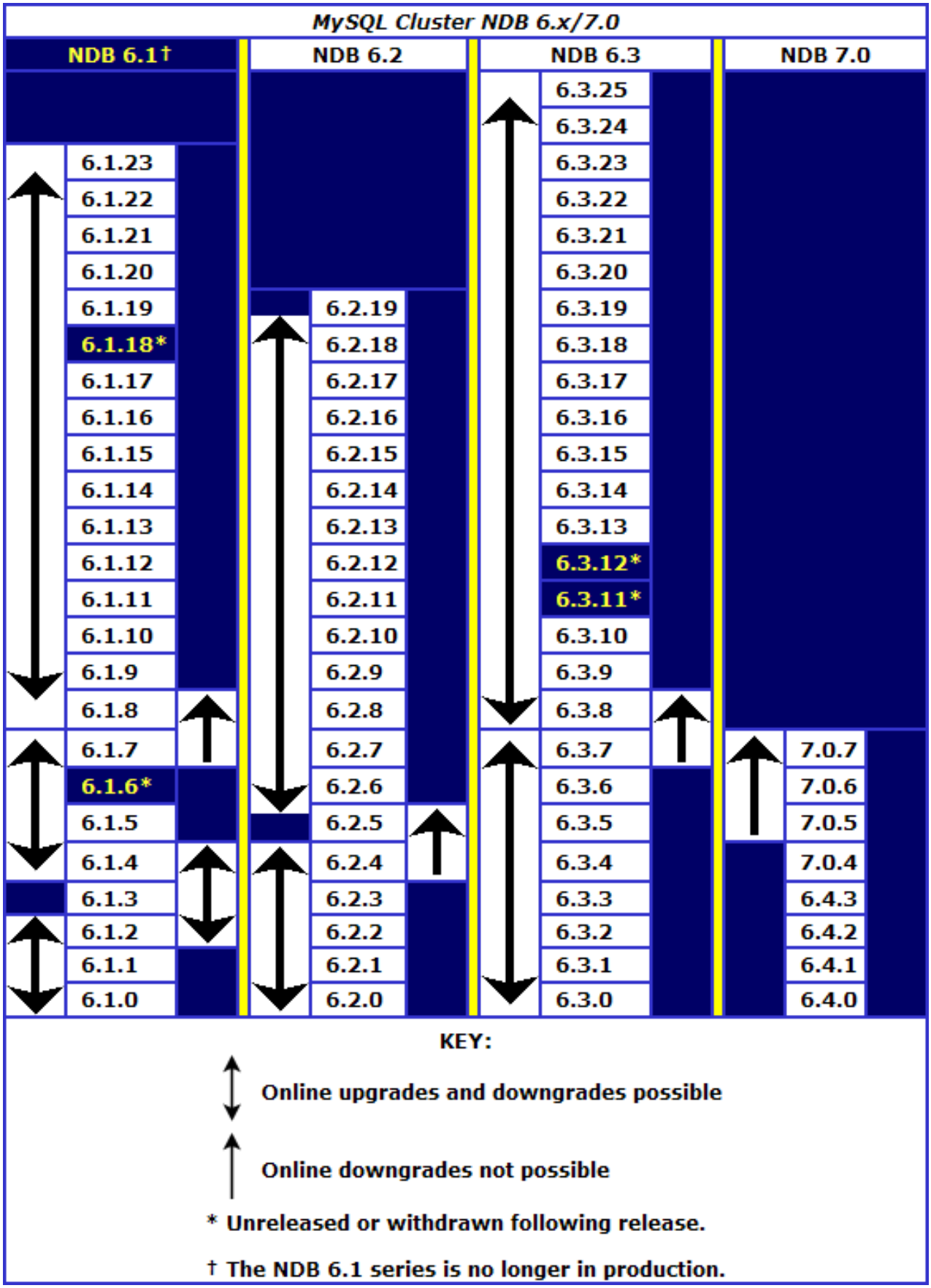
  1. Back up all `NDBCLUSTER` tables.
  2. Upgrade the MySQL Cluster software on all data, management, and SQL nodes.
  3. Shut down the cluster completely (this includes all data, management, and API or SQL nodes).
  4. Restart the cluster, starting all data nodes with the `--initial` option (to clear and rebuild the data node file systems).
  5. Restore the `NDBCLUSTER` tables from backup.

  This is not necessary for `NDBCLUSTER` tables created in MySQL 5.1.18 and later; such tables will automatically be compatible with online adding and dropping of columns (as implemented beginning with MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.2).

  In order to minimise possible later difficulties, it is strongly advised that the procedure outlined above be followed as soon as possible after to upgrading from MySQL 5.1.17 or earlier to MySQL 5.1.18 or later.

  Information about how this change effects users of MySQL Cluster NDB 6.x/7.0 is provided later in this section.
- MySQL Cluster is not supported in standard MySQL 5.1 releases, beginning with MySQL 5.1.25. If you are using MySQL Cluster in a standard MySQL 5.1 release, you should upgrade to the most recent MySQL Cluster NDB 6.2 or 6.3 release.

The following table shows Cluster upgrade and downgrade compatibility between different releases of MySQL Cluster NDB 6.x/7.0:



Notes — MySQL Cluster NDB 6.x/7.x.

- MySQL Cluster NDB 6.1 is no longer in production; if you are still using a MySQL Cluster NDB 6.1 release, you should upgrade to the most recent MySQL Cluster NDB 6.2 or 6.3 as soon as possible.
- It is not possible to upgrade from MySQL Cluster NDB 6.1.2 (or an older 6.1 release) directly to 6.1.4 or a newer NDB 6.1 release, or to downgrade from 6.1.4 (or a newer 6.1 release) directly to 6.1.2 or an older NDB 6.1 release; in either case, you must upgrade or downgrade to MySQL Cluster NDB 6.1.3 first.
- It is not possible to perform an online downgrade from MySQL Cluster NDB 6.1.8 (or a newer 6.1 release) to MySQL Cluster NDB 6.1.7 (or an older 6.1 release).
- MySQL Cluster NDB 6.1.6 and 6.1.18 were not released.
- It is not possible to perform an online upgrade or downgrade between MySQL Cluster NDB 6.2 and any previous release series (including mainline MySQL 5.1 and MySQL Cluster NDB 6.1); it is necessary to perform a dump and reload. However, it should be possible to perform online upgrades or downgrades between any MySQL Cluster NDB 6.2 release and any MySQL Cluster NDB 6.3 release up to and including 6.3.7.
- The internal specifications for columns in `NDB` tables changed in MySQL Cluster NDB 6.1.17 and 6.2.1 to allow compatibility with future MySQL Cluster releases that are expected to implement online adding and dropping of columns. This change is not backward-compatible with earlier MySQL or MySQL Cluster NDB 6.x versions.

  In order to make tables created in earlier versions compatible with online adding and dropping of columns in later versions, it is necessary to force MySQL Cluster to convert the tables to the new format by following this procedure following an upgrade:

  1. Upgrade the MySQL Cluster software on all data, management, and SQL nodes
  2. Back up all `NDB` tables
  3. Shut down the cluster (all data, management, and SQL nodes)
  4. Restart the cluster, starting all data nodes with the `--initial` option (to clear and rebuild the data node file systems)
  5. Restore the tables from backup

  In order to minimise possible later difficulties, it is strongly advised that the procedure outlined above be followed as soon as possible after to upgrading between the versions indicated. The procedure is *not* necessary for `NDBCLUSTER` tables created in any of the following versions:

  - MySQL Cluster NDB 6.1.8 or a later MySQL Cluster NDB 6.1 release
  - MySQL Cluster 6.2.1 or a later MySQL Cluster NDB 6.2 release
  - Any MySQL Cluster NDB 6.3 release

  Tables created in the listed versions (or later ones, as indicated) are already compatible with online adding and dropping of columns (as implemented beginning with MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.2).
- It was not possible to perform an online upgrade between any MySQL Cluster NDB 6.2 release and MySQL Cluster NDB 6.3.8 and later MySQL Cluster 6.3 releases. This issue was fixed in MySQL Cluster NDB 6.3.21. (Bug#41435)
- Online downgrades between MySQL Cluster NDB 6.2.5 and earlier releases are not supported.
- Online downgrades between MySQL Cluster NDB 6.3.8 and earlier releases are not supported.
- Online upgrades from any MySQL Cluster NDB 7.0 release up to and including MySQL Cluster NDB 7.0.4 (as well as all early releases numbered NDB 6.4.x) to MySQL Cluster NDB 7.0.5 or later are not possible. Upgrades to MySQL Cluster NDB 7.0.6 or later from MySQL Cluster NDB 6.3.8 or a later MySQL Cluster NDB 6.3 release, or from MySQL Cluster NDB 7.0.5 or later, are supported. (Bug#44294)

Chapter 6. MySQL Cluster Programs

Using and managing a MySQL Cluster requires several specialized programs, which we describe in this chapter. We discuss the purposes of these programs in a MySQL Cluster, how to use the programs, and what startup options are available for each of them.

These programs include the MySQL Cluster data, management, and SQL node process daemons (`ndbd`, `ndb_mgmd`, and `mysqld`) and the management client (`ndb_mgm`).

Other NDB utility, diagnostic, and example programs are included with the MySQL Cluster distribution. These include `ndb_restore`, `ndb_show_tables`, and `ndb_config`). These programs are covered later in this chapter.

The last two sections of this chapter contain tables of options used, respectively, with `mysqld` and with the various NDB programs.

6.1. MySQL Server Usage for MySQL Cluster

`mysqld` is the traditional MySQL server process. To be used with MySQL Cluster, `mysqld` needs to be built with support for the `NDBCLUSTER` storage engine, as it is in the precompiled binaries available from http://dev.mysql.com/downloads/. If you build MySQL from source, you must invoke `configure` with the `--with-ndbcluster` option to enable `NDB Cluster` storage engine support.

If the `mysqld` binary has been built with Cluster support, the `NDBCLUSTER` storage engine is still disabled by default. You can use either of two possible options to enable this engine:

- Use `--ndbcluster` as a startup option on the command line when starting `mysqld`.
- Insert a line containing `NDBCLUSTER` in the `[mysqld]` section of your `my.cnf` file.

An easy way to verify that your server is running with the `NDBCLUSTER` storage engine enabled is to issue the `SHOW ENGINES` statement in the MySQL Monitor (`mysql`). You should see the value `YES` as the `Support` value in the row for `NDBCLUSTER`. If you see `NO` in this row or if there is no such row displayed in the output, you are not running an `NDB`-enabled version of MySQL. If you see `DISABLED` in this row, you need to enable it in either one of the two ways just described.

To read cluster configuration data, the MySQL server requires at a minimum three pieces of information:

- The MySQL server's own cluster node ID
- The host name or IP address for the management server (MGM node)
- The number of the TCP/IP port on which it can connect to the management server

Node IDs can be allocated dynamically, so it is not strictly necessary to specify them explicitly.

The `mysqld` parameter `ndb-connectstring` is used to specify the connectstring either on the command line when starting `mysqld` or in `my.cnf`. The connectstring contains the host name or IP address where the management server can be found, as well as the TCP/IP port it uses.

In the following example, `ndb_mgmd.mysql.com` is the host where the management server resides, and the management server listens for cluster messages on port 1186:

```
shell> mysqld --ndbcluster --ndb-connectstring=ndb_mgmd.mysql.com:1186
```

See Section 3.4.3, “The MySQL Cluster Connectstring”, for more information on connectstrings.

Given this information, the MySQL server will be a full participant in the cluster. (We often refer to a `mysqld` process running in this manner as an SQL node.) It will be fully aware of all cluster data nodes as well as their status, and will establish connections to all data nodes. In this case, it is able to use any data node as a transaction coordinator and to read and update node data.

You can see in the `mysql` client whether a MySQL server is connected to the cluster using `SHOW PROCESSLIST`. If the MySQL server is connected to the cluster, and you have the `PROCESS` privilege, then the first row of the output is as shown here:

```
mysql> SHOW PROCESSLIST \G
*************************** 1. row ***************************
     Id: 1
   User: system user
   Host:
     db:
```

```
Command: Daemon
   Time: 1
  State: Waiting for event from ndbcluster
   Info: NULL
```

> **Important**
>
> To participate in a MySQL Cluster, the `mysqld` process must be started with *both* the options `--ndbcluster` and `--ndb-connectstring` (or their equivalents in `my.cnf`). If `mysqld` is started with only the `--ndbcluster` option, or if it is unable to contact the cluster, it is not possible to work with `NDB` tables, *nor is it possible to create any new tables regardless of storage engine*. The latter restriction is a safety measure intended to prevent the creation of tables having the same names as `NDB` tables while the SQL node is not connected to the cluster. If you wish to create tables using a different storage engine while the `mysqld` process is not participating in a MySQL Cluster, you must restart the server *without* the `--ndbcluster` option.

6.2. ndbd — The MySQL Cluster Data Node Daemon

`ndbd` is the process that is used to handle all the data in tables using the NDB Cluster storage engine. This is the process that empowers a data node to accomplish distributed transaction handling, node recovery, checkpointing to disk, online backup, and related tasks.

In a MySQL Cluster, a set of `ndbd` processes cooperate in handling data. These processes can execute on the same computer (host) or on different computers. The correspondences between data nodes and Cluster hosts is completely configurable.

The following list describes command options specific to the MySQL Cluster data node program `ndbd`.

> **Note**
>
> All of these options also apply to the multi-threaded version of this program — `ndbmtd`, which is available in MySQL Cluster NDB 7.0 — and you may substitute "`ndbmtd`" for "`ndbd`" wherever the latter occurs in this section.

For options common to all `NDBCLUSTER` programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

- `--bind-address`

Version Introduced	5.1.12	
Command Line Format	`--bind-address=name`	
Value Set	**Type**	`string`
	Default	

Causes `ndbd` to bind to a specific network interface (host name or IP address). This option has no default value.

This option was added in MySQL 5.1.12.

- `--daemon`, `-d`

Command Line Format	`--daemon`	
Value Set	**Type**	`boolean`
	Default	`TRUE`

Instructs `ndbd` to execute as a daemon process. This is the default behavior. `--nodaemon` can be used to prevent the process from running as a daemon.

- `--initial`

Command Line Format	`--initial`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

Instructs `ndbd` to perform an initial start. An initial start erases any files created for recovery purposes by earlier instances of

ndbd. It also re-creates recovery log files. Note that on some operating systems this process can take a substantial amount of time.

An --initial start is to be used *only* when starting the ndbd process under very special circumstances; this is because this option causes all files to be removed from the Cluster file system and all redo log files to be re-created. These circumstances are listed here:

- When performing a software upgrade which has changed the contents of any files.
- When restarting the node with a new version of ndbd.
- As a measure of last resort when for some reason the node restart or system restart repeatedly fails. In this case, be aware that this node can no longer be used to restore data due to the destruction of the data files.

> **Important**
>
> This option does *not* affect either of the following:
>
> - Backup files that have already been created by the affected node
> - MySQL Cluster Disk Data files (see Chapter 10, *MySQL Cluster Disk Data Tables*).

It is permissible to use this option when starting the cluster for the very first time (that is, before any data node files have been created); however, it is *not* necessary to do so.

- --initial-start

Version Introduced	5.1.11	
Command Line Format	--initial-start	
Value Set	**Type**	boolean
	Default	FALSE

This option is used when performing a partial initial start of the cluster. Each node should be started with this option, as well as --nowait-nodes.

For example, suppose you have a 4-node cluster whose data nodes have the IDs 2, 3, 4, and 5, and you wish to perform a partial initial start using only nodes 2, 4, and 5 — that is, omitting node 3:

```
ndbd --ndbd-nodeid=2 --nowait-nodes=3 --initial-start
ndbd --ndbd-nodeid=4 --nowait-nodes=3 --initial-start
ndbd --ndbd-nodeid=5 --nowait-nodes=3 --initial-start
```

This option was added in MySQL 5.1.11.

> **Important**
>
> Prior to MySQL 5.1.19, it was not possible to perform DDL operations involving Disk Data tables on a partially started cluster. (See Bug#24631.)

- --nowait-nodes=*node_id_1*[, *node_id_2*[, ...]]

Version Introduced	5.1.11	
Command Line Format	--nowait-nodes=list	
Value Set	**Type**	string
	Default	

This option takes a list of data nodes which for which the cluster will not wait for before starting.

This can be used to start the cluster in a partitioned state. For example, to start the cluster with only half of the data nodes (nodes 2, 3, 4, and 5) running in a 4-node cluster, you can start each ndbd process with --nowait-nodes=3,5. In this case, the cluster starts as soon as nodes 2 and 4 connect, and does *not* wait StartPartitionedTimeout milliseconds for nodes 3 and 5 to connect as it would otherwise.

If you wanted to start up the same cluster as in the previous example without one ndbd — say, for example, that the host machine for node 3 has suffered a hardware failure — then start nodes 2, 4, and 5 with --nowait-nodes=3. Then the cluster will start as soon as nodes 2, 4, and 5 connect and will not wait for node 3 to start.

This option was added in MySQL 5.1.9.

- --nodaemon

Command Line Format	--nodaemon	
Value Set	Type	boolean
	Default	FALSE

Instructs ndbd not to start as a daemon process. This is useful when ndbd is being debugged and you want output to be redirected to the screen.

- --nostart, -n

Command Line Format	--nostart	
Value Set	Type	boolean
	Default	FALSE

Instructs ndbd not to start automatically. When this option is used, ndbd connects to the management server, obtains configuration data from it, and initializes communication objects. However, it does not actually start the execution engine until specifically requested to do so by the management server. This can be accomplished by issuing the proper START command in the management client (see Section 7.2, "Commands in the MySQL Cluster Management Client").

ndbd generates a set of log files which are placed in the directory specified by DataDir in the config.ini configuration file.

These log files are listed below. *node_id* is the node's unique identifier. Note that *node_id* represents the node's unique identifier. For example, ndb_2_error.log is the error log generated by the data node whose node ID is 2.

- ndb_*node_id*_error.log is a file containing records of all crashes which the referenced ndbd process has encountered. Each record in this file contains a brief error string and a reference to a trace file for this crash. A typical entry in this file might appear as shown here:

```
Date/Time: Saturday 30 July 2004 - 00:20:01
Type of error: error
Message: Internal program error (failed ndbrequire)
Fault ID: 2341
Problem data: DbtupFixAlloc.cpp
Object of reference: DBTUP (Line: 173)
ProgramName: NDB Kernel
ProcessID: 14909
TraceFile: ndb_2_trace.log.2
***EOM***
```

Listings of possible ndbd exit codes and messages generated when a data node process shuts down prematurely can be found in ndbd Error Messages.

> **Important**
>
> *The last entry in the error log file is not necessarily the newest one* (nor is it likely to be). Entries in the error log are *not* listed in chronological order; rather, they correspond to the order of the trace files as determined in the ndb_*node_id*_trace.log.next file (see below). Error log entries are thus overwritten in a cyclical and not sequential fashion.

- ndb_*node_id*_trace.log.*trace_id* is a trace file describing exactly what happened just before the error occurred. This information is useful for analysis by the MySQL Cluster development team.

  It is possible to configure the number of these trace files that will be created before old files are overwritten. *trace_id* is a number which is incremented for each successive trace file.

- ndb_*node_id*_trace.log.next is the file that keeps track of the next trace file number to be assigned.
- ndb_*node_id*_out.log is a file containing any data output by the ndbd process. This file is created only if ndbd is started as a daemon, which is the default behavior.
- ndb_*node_id*.pid is a file containing the process ID of the ndbd process when started as a daemon. It also functions as a lock file to avoid the starting of nodes with the same identifier.
- ndb_*node_id*_signal.log is a file used only in debug versions of ndbd, where it is possible to trace all incoming, outgoing, and internal messages with their data in the ndbd process.

It is recommended not to use a directory mounted through NFS because in some environments this can cause problems whereby the lock on the .pid file remains in effect even after the process has terminated.

To start ndbd, it may also be necessary to specify the host name of the management server and the port on which it is listening. Optionally, one may also specify the node ID that the process is to use.

```
shell> ndbd --connect-string="nodeid=2;host=ndb_mgmd.mysql.com:1186"
```

See Section 3.4.3, “The MySQL Cluster Connectstring”, for additional information about this issue. Section 6.2, “ndbd — The MySQL Cluster Data Node Daemon”, describes other options for ndbd.

When ndbd starts, it actually initiates two processes. The first of these is called the “angel process”; its only job is to discover when the execution process has been completed, and then to restart the ndbd process if it is configured to do so. Thus, if you attempt to kill ndbd via the Unix kill command, it is necessary to kill both processes, beginning with the angel process. The preferred method of terminating an ndbd process is to use the management client and stop the process from there.

The execution process uses one thread for reading, writing, and scanning data, as well as all other activities. This thread is implemented asynchronously so that it can easily handle thousands of concurrent actions. In addition, a watch-dog thread supervises the execution thread to make sure that it does not hang in an endless loop. A pool of threads handles file I/O, with each thread able to handle one open file. Threads can also be used for transporter connections by the transporters in the ndbd process. In a multi-processor system performing a large number of operations (including updates), the ndbd process can consume up to 2 CPUs if permitted to do so.

For a machine with many CPUs it is possible to use several ndbd processes which belong to different node groups; however, such a configuration is still considered experimental and is not supported for MySQL 5.1 in a production setting. See Chapter 12, *Known Limitations of MySQL Cluster*.

6.3. ndbmtd — The MySQL Cluster Data Node Daemon (Multi-Threaded)

ndbmtd is a multi-threaded version of ndbd, the process that is used to handle all the data in tables using the NDBCLUSTER storage engine. ndbmtd is intended for use on host computers having multiple CPU cores. Except where otherwise noted, ndbmtd functions in the same way as ndbd; therefore, in this section, we concentrate on the ways in which ndbmtd differs from ndbd, and you should consult Section 6.2, “ndbd — The MySQL Cluster Data Node Daemon”, for additional information about running MySQL Cluster data nodes that apply to both the single-threaded and multi-threaded versions of the data node process.

Command-line options and configuration parameters used with ndbd also apply to ndbmtd. For more information about these options and parameters, see Section 6.24.2, “Program Options for ndbd and ndbmtd”, and Section 3.4.6, “Defining MySQL Cluster Data Nodes”, respectively.

ndbmtd is also file system-compatible with ndbd. In other words, a data node running ndbd can be stopped, the binary replaced with ndbmtd, and then restarted without any loss of data. (However, when doing this, you must make sure that MaxNoOfExecutionThreads is set to an apppriate value before restarting the node if you wish for ndbmtd to run in multi-threaded fashion.) Similarly, an ndbmtd binary can be replaced with ndbd simply by stopping the node and then starting ndbd in place of the multi-threaded binary. It is not necessary when switching between the two to start the data node binary using --initial.

> **Important**
>
> We do not currently recommend using ndbmtd with MySQL Cluster Disk Data tables in production, due to known issues which we are working to fix in a future MySQL Cluster release. (Bug#41915, Bug#44915)

Using ndbmtd differs from using ndbd in two key respects:

1. You must set an appropriate value for the MaxNoOfExecutionThreads configuration parameter in the config.ini file. If you do not do so, ndbmtd runs in single-threaded mode — that is, it behaves like ndbd.

2. Trace files are generated by critical errors in `ndbmtd` processes in a somewhat different fashion from how these are generated by `ndbd` failures.

These differences are discussed in more detail in the next few paragraphs.

Number of execution threads. The `MaxNoOfExecutionThreads` configuration parameter is used to determine the number of local query handler (LQH) threads spawned by `ndbmtd`. Although this parameter is set in `[ndbd]` or `[ndbd default]` sections of the `config.ini` file, it is exclusive to `ndbmtd` and does not apply to `ndbd`.

This parameter takes an integer value from 2 to 8 inclusive. Generally, you should set this to the number of CPU cores on the data node host, as shown in the following table:

Number of Cores	Recommended `MaxNoOfExecutionThreads` Value
2	2
4	4
8 or more	8

(It is possible to set this parameter to other values within the permitted range, but these are automatically rounded as shown in the **Value Used** column of the next table in this section.)

The multi-threaded data node process always spawns at least 4 threads:

- 1 local query handler (LQH) thread
- 1 transaction coordinator (TC) thread
- 1 transporter thread
- 1 subscription manager (SUMA) thread

Setting this parameter to a value between 4 and 8 inclusive causes additional LQH threads to be used by `ndbmtd` (up to a maximum of 4 LQH threads), as shown in the following table:

`config.ini` Value	Value Used	Number of LQH Threads Used
3	2	1
5 or 6	4	2
7	8	4

Setting this parameter outside the permitted range of values causes the management server to abort on startup with the error `ERROR LINE NUMBER: ILLEGAL VALUE VALUE FOR PARAMETER MAXNOOFEXECUTIONTHREADS`.

> **Note**
>
> In MySQL Cluster NDB 6.4.0, it is not possible to set `MaxNoOfExecutionThreads` to 2. You can safely use the value 3 instead (it is treated as 2 internally). This issue is resolved in MySQL Cluster NDB 6.4.1.

In MySQL Cluster NDB 6.4.0 through 6.4.3, the default value for this parameter was undefined, although the default behavior for `ndbmtd` was to use 1 LQH thread, as though `MaxNoOfExecutionThreads` had been set to 2. Beginning with MySQL Cluster NDB 7.0.4, this parameter has an explcit default value of 2, thus guaranteeing this default behavior.

In MySQL Cluster NDB 7.0, it is not possible to cause `ndbmtd` to use more than 1 TC thread, although we plan to introduce this capability in a future MySQL Cluster release series.

Like `ndbd`, `ndbmtd` generates a set of log files which are placed in the directory specified by `DataDir` in the `config.ini` configuration file. Except for trace files, these are generated in the same way and have the same names as those generated by `ndbd`.

In the event of a critical error, `ndbmtd` generates trace files describing what happened just prior to the error' occurrence. These files, which can be found in the data node's `DataDir`, are useful for analysis of problems by the MySQL Cluster Development and Support teams. One trace file is generated for each `ndbmtd` thread. The names of these files follow the pattern `ndb_node_id_trace.log.trace_id_tthread_id`, where *`node_id`* is the data node's unique node ID in the cluster, *`trace_id`* is a trace sequence number, and *`thread_id`* is the thread ID. For example, in the event of the failure of an `ndbmtd`

process running as a MySQL Cluster data node having the node ID 3 and with `MaxNoOfExecutionThreads` equal to 4, four trace files are generated in the data node's data directory; if the is the first time this node has failed, then these files are named `ndb_3_trace.log.1_t1`, `ndb_3_trace.log.1_t2`, `ndb_3_trace.log.1_t3`, and `ndb_3_trace.log.1_t4`. Internally, these trace files follow the same format as `ndbd` trace files.

The `ndbd` exit codes and messages that are generated when a data node process shuts down prematurely are also used by `ndbmtd`. See `ndbd` Error Messages, for a listing of these.

> **Note**
>
> It is possible to use `ndbd` and `ndbmtd` concurrently on different data nodes in the same MySQL Cluster. However, such configurations have not been tested extensively; thus, we cannot not recommend doing so in a production setting at this time.

6.4. `ndb_mgmd` — The MySQL Cluster Management Server Daemon

The management server is the process that reads the cluster configuration file and distributes this information to all nodes in the cluster that request it. It also maintains a log of cluster activities. Management clients can connect to the management server and check the cluster's status.

The following list includes options that are specific to `ndb_mgmd`. For options common to all NDB programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

- `--bind-address=host[:port]`

Version Introduced	5.1.22-ndb-6.3.2	
Command Line Format	`--bind-address`	
Value Set	**Type**	`string`
	Default	`[none]`

When specified, this option limits management server connections by management clients to clients at the specified host name or IP address (and possibly port, if this is also specified). In such cases, a management client attempting to connect to the management server from any other address fails with the error `UNABLE TO SETUP PORT: HOST:PORT!`

If the *`port`* is not specified, the management client attempts to use port 1186.

This option was added in MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.2.

- `--configdir=directory`

Version Introduced	5.1.30-ndb-6.4.0	
Command Line Format	`--configdir=directory`	
Value Set	**Type**	`filename`
	Default	`$INSTALLDIR/mysql-cluster`

Beginning with MySQL Cluster NDB 6.4.0, configuration data is cached internally rather than being read from the cluster global configuration file each time the management server is started (see Section 3.4, "MySQL Cluster Configuration Files"). This option instructs the management server to its configuration cache in the *`directory`* indicated. By default, this is a directory named `mysql-cluster` in the MySQL installation directory — for example, if you compile and install MySQL Cluster on a Unix system using the default location, this is `/usr/local/mysql-cluster`.

This behavior can be overridden using the `--initial` or `--reload` option for `ndb_mgmd`. Each of these options is described elsewhere in this section.

This option is available beginning with MySQL Cluster NDB 6.4.0.

- `--config-file=filename`, `-f filename`

Command Line Format	`-c`	
Value Set	**Type**	`filename`

	Default	`./config.ini`

Instructs the management server as to which file it should use for its configuration file. By default, the management server looks for a file named `config.ini` in the same directory as the `ndb_mgmd` executable; otherwise the file name and location must be specified explicitly.

Beginning with MySQL Cluster NDB 6.4.0, this option is ignored unless the management server is forced to read the configuration file, either because `ndb_mgmd` was started with the `--reload` or `--initial` option, or because the management server could not find any configuration cache. See Section 3.4, "MySQL Cluster Configuration Files", for more information.

- `--daemon`, `-d`

Command Line Format	`--daemon`	
Value Set	**Type**	`boolean`
	Default	`TRUE`

Instructs `ndb_mgmd` to start as a daemon process. This is the default behavior.

- `--initial`

Version Introduced	5.1.30-ndb-6.4.0	
Command Line Format	`--initial`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

Beginning with MySQL Cluster NDB 6.4.0, configuration data is cached internally rather than being read from the cluster global configuration file each time the management server is started (see Section 3.4, "MySQL Cluster Configuration Files"). Using this option overrides this behavior, by forcing the management server to delete any existing cache files, and then to re-read the configuration data from the cluster configuration file and to build a new cache.

This differs in two ways from the `--reload` option. First, `--reload` forces the server to check the configuration file against the cache and reload its data only if the contents of the file are different from the cache. Second, `--reload` does not delete any existing cache files.

If `ndb_mgmd` is invoked with `--initial` but cannot find a global configuration file, the management server cannot start.

This option was introduced in MySQL Cluster NDB 6.4.0.

- `--nodaemon`

Command Line Format	`--nodaemon`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

Instructs `ndb_mgmd` not to start as a daemon process.

- `--print-full-config`, `-P`

Command Line Format	`--print-full-config`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

Shows extended information regarding the configuration of the cluster. With this option on the command line the `ndb_mgmd`

process prints information about the cluster setup including an extensive list of the cluster configuration sections as well as parameters and their values. Normally used together with the `--config-file` (`-f`) option.

- `--reload`

Version Introduced	5.1.30-ndb-6.4.0	
Command Line Format	`--reload`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

Beginning with MySQL Cluster NDB 6.4.0, configuration data is stored internally rather than being read from the cluster global configuration file each time the management server is started (see Section 3.4, "MySQL Cluster Configuration Files"). Using this option forces the management server to check its internal data store against the cluster configuration file and to reload the configuration if it finds that the configuration file does not match the cache. Existing configuration cache files are preserved, but not used.

This differs in two ways from the `--initial` option. First, `--initial` causes all cache files to be deleted. Second, `--initial` forces the management server to re-read the global configuration file and construct a new cache.

If the management server cannot find a global configuration file, then the `--reload` option is ignored.

This option was introduced in MySQL Cluster NDB 6.4.0.

It is not strictly necessary to specify a connectstring when starting the management server. However, if you are using more than one management server, a connectstring should be provided and each node in the cluster should specify its node ID explicitly.

See Section 3.4.3, "The MySQL Cluster Connectstring", for information about using connectstrings. Section 6.4, "`ndb_mgmd` — The MySQL Cluster Management Server Daemon", describes other options for `ndb_mgmd`.

The following files are created or used by `ndb_mgmd` in its starting directory, and are placed in the `DataDir` as specified in the `config.ini` configuration file. In the list that follows, *`node_id`* is the unique node identifier.

- `config.ini` is the configuration file for the cluster as a whole. This file is created by the user and read by the management server. Chapter 3, *MySQL Cluster Configuration*, discusses how to set up this file.
- `ndb_`*`node_id`*`_cluster.log` is the cluster events log file. Examples of such events include checkpoint startup and completion, node startup events, node failures, and levels of memory usage. A complete listing of cluster events with descriptions may be found in Chapter 7, *Management of MySQL Cluster*.

  When the size of the cluster log reaches one million bytes, the file is renamed to `ndb_`*`node_id`*`_cluster.log.`*`seq_id`*, where *`seq_id`* is the sequence number of the cluster log file. (For example: If files with the sequence numbers 1, 2, and 3 already exist, the next log file is named using the number 4.)
- `ndb_`*`node_id`*`_out.log` is the file used for `stdout` and `stderr` when running the management server as a daemon.
- `ndb_`*`node_id`*`.pid` is the process ID file used when running the management server as a daemon.

6.5. `ndb_mgm` — The MySQL Cluster Management Client

The `ndb_mgm` management client process is actually not needed to run the cluster. Its value lies in providing a set of commands for checking the cluster's status, starting backups, and performing other administrative functions. The management client accesses the management server using a C API. Advanced users can also employ this API for programming dedicated management processes to perform tasks similar to those performed by `ndb_mgm`.

To start the management client, it is necessary to supply the host name and port number of the management server:

```
shell> ndb_mgm [host_name [port_num]]
```

For example:

```
shell> ndb_mgm ndb_mgmd.mysql.com 1186
```

The default host name and port number are `localhost` and 1186, respectively.

The following list includes options that are specific to `ndb_mgm`. For options common to all NDB programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

- `--try-reconnect=number`

Command Line Format	`--try-reconnect=#`	
Value Set	Type	`boolean`
	Default	`TRUE`

If the connection to the management server is broken, the node tries to reconnect to it every 5 seconds until it succeeds. By using this option, it is possible to limit the number of attempts to *`number`* before giving up and reporting an error instead.

Additional information about using `ndb_mgm` can be found in Section 6.24.4, "Program Options for `ndb_mgm`", and Section 7.2, "Commands in the MySQL Cluster Management Client".

6.6. `ndb_config` — Extract MySQL Cluster Configuration Information

This tool extracts current configuration information for data nodes, SQL nodes, and API nodes from a cluster management node (and possibly its `config.ini` file). Beginning with MySQL Cluster NDB 6.3.25 and MySQL Cluster NDB 7.0.6, it can also provide an offline dump (in text or XML format) of all configuration parameters which can be used, along with their default, maximum, and minimum values and other information (see the discussion of the `--configinfo` and `--xml` options later in this section).

Usage:

```
ndb_config options
```

The *`options`* available for this utility differ somewhat from those used with the other utilities, and so are listed in their entirety in the next section, followed by some examples.

Options:

- `--usage`, `--help`, or `-?`

Command Line Format	`--help`

Causes `ndb_config` to print a list of available options, and then exit.

- `--version`, `-V`

Command Line Format	`-V`

Causes `ndb_config` to print a version information string, and then exit.

- `--ndb-connectstring=connect_string`

Command Line Format	`--ndb-connectstring=name`	
Value Set	Type	`string`
	Default	`localhost:1186`

Specifies the connectstring to use in connecting to the management server. The format for the connectstring is the same as described in Section 3.4.3, "The MySQL Cluster Connectstring", and defaults to `localhost:1186`.

The use of `-c` as a short version for this option is supported for `ndb_config` beginning with MySQL 5.1.12.

- `--config-file=path-to-file`

  Gives the path to the management server's configuration file (`config.ini`). This may be a relative or absolute path. If the management node resides on a different host from the one on which `ndb_config` is invoked, then an absolute path must be used.

- `--query=query-options`, `-q query-options`

Command Line Format	`--query=string`	
Value Set	Type	`string`
	Default	

  This is a comma-delimited list of *query options* — that is, a list of one or more node attributes to be returned. These include `id` (node ID), type (node type — that is, `ndbd`, `mysqld`, or `ndb_mgmd`), and any configuration parameters whose values are to be obtained.

  For example, `--query=id,type,indexmemory,datamemory` would return the node ID, node type, `DataMemory`, and `IndexMemory` for each node.

  Note

  If a given parameter is not applicable to a certain type of node, than an empty string is returned for the corresponding value. See the examples later in this section for more information.

- `--host=hostname`

Command Line Format	`--host=name`	
Value Set	Type	`string`
	Default	

  Specifies the host name of the node for which configuration information is to be obtained.

- `--id=node_id`, `--nodeid=node_id`

Command Line Format	`--ndb-nodeid=#`	
Value Set	Type	`numeric`
	Default	`0`

  Used to specify the node ID of the node for which configuration information is to be obtained.

- `--nodes`

Command Line Format	`--nodes`	
Value Set	Type	`boolean`
	Default	`FALSE`

  (Tells `ndb_config` to print information from parameters defined in `[ndbd]` sections only. Currently, using this option has no affect, since these are the only values checked, but it may become possible in future to query parameters set in `[tcp]` and other sections of cluster configuration files.)

- `--type=node_type`

Command Line Format	`--type=name`	
Value Set	Type	`enumeration`
	Default	
	Valid Values	`ndbd`, `mysqld`, `ndb_mgmd`

Filters results so that only configuration values applying to nodes of the specified *`node_type`* (`ndbd`, `mysqld`, or `ndb_mgmd`) are returned.

- `--fields=`*`delimiter`*, `-f` *`delimiter`*

Command Line Format	`--fields=string`	
Value Set	**Type**	`string`
	Default	

Specifies a *`delimiter`* string used to separate the fields in the result. The default is "`,`" (the comma character).

> **Note**
>
> If the *`delimiter`* contains spaces or escapes (such as `\n` for the linefeed character), then it must be quoted.

- `--rows=`*`separator`*, `-r` *`separator`*

Command Line Format	`--rows=string`	
Value Set	**Type**	`string`
	Default	

Specifies a *`separator`* string used to separate the rows in the result. The default is a space character.

> **Note**
>
> If the *`separator`* contains spaces or escapes (such as `\n` for the linefeed character), then it must be quoted.

- `--configinfo` [`--xml`]

Version Introduced	5.1.34-ndb-7.0.6	
Command Line Format	`--configinfo`	
Value Set	**Type**	`boolean`
	Default	`false`

Version Introduced	5.1.34-ndb-7.0.6	
Command Line Format	`--configinfo --xml`	
Value Set	**Type**	`boolean`
	Default	`false`

The `--configinfo` option, added in MySQL Cluster NDB 6.3.25 and MySQL Cluster NDB 7.0.6, causes `ndb_config` to dump a list of each MySQL Cluster configuration parameter supported by the MySQL Cluster distribution of which `ndb_config` is a part, including the following information:

- A brief description of each parameter's purpose, effects, and usage
- The section of the `config.ini` file where the parameter may be used
- The parameter's data type or unit of measurement
- Where applicable, the parameter's default, minimum, and maximum values
- A brief description of the parameter's purpose, effects, and usage
- MySQL Cluster release version and build information

By default, this output is in text format. Part of this output is shown here:

```
shell> ndb_config --configinfo
****** SYSTEM ******
```

```
Name (String)
Name of system (NDB Cluster)
MANDATORY
PrimaryMGMNode (Non-negative Integer)
Node id of Primary ndb_mgmd(MGM) node
Default: 0 (Min: 0, Max: 4294967039)
ConfigGenerationNumber (Non-negative Integer)
Configuration generation number
Default: 0 (Min: 0, Max: 4294967039)
****** DB ******
MaxNoOfSubscriptions (Non-negative Integer)
Max no of subscriptions (default 0 == MaxNoOfTables)
Default: 0 (Min: 0, Max: 4294967039)
MaxNoOfSubscribers (Non-negative Integer)
Max no of subscribers (default 0 == 2 * MaxNoOfTables)
Default: 0 (Min: 0, Max: 4294967039)
…
```

You can obtain the output as XML by using the `--xml` option (also available beginning with MySQL Cluster NDB 6.3.25 and MySQL Cluster NDB 7.0.6) in addition to `--configinfo`. A portion of the resulting output is shown in this example:

```
shell> ndb_config --configinfo --xml
<configvariables protocolversion="1" ndbversionstring="mysql-5.1.34 ndb-7.0.6"
                 ndbversion="458758" ndbversionmajor="7" ndbversionminor="0"
                 ndbversionbuild="6">
  <section name="SYSTEM">
    <param name="Name" comment="Name of system (NDB Cluster)" type="string"
           mandatory="true"/>
    <param name="PrimaryMGMNode" comment="Node id of Primary ndb_mgmd(MGM) node"
           type="unsigned" default="0" min="0" max="4294967039"/>
    <param name="ConfigGenerationNumber" comment="Configuration generation number"
           type="unsigned" default="0" min="0" max="4294967039"/>
  </section>
  <section name="NDBD">
    <param name="MaxNoOfSubscriptions" comment="Max no of subscriptions (default 0 == MaxNoOfTables)"
           type="unsigned" default="0" min="0" max="4294967039"/>
    <param name="MaxNoOfSubscribers" comment="Max no of subscribers (default 0 == 2 * MaxNoOfTables)"
           type="unsigned" default="0" min="0" max="4294967039"/>
    …
  </section>
  …
</configvariables>
```

Important

The `--xml` option can be used only with the `--configinfo` option. Using `--xml` without `--configinfo` fails with an error.

Unlike the options used with this program to obtain current configuration data, `--configinfo` and `--xml` use information obtained from the MySQL Cluster sources when `ndb_config` was compiled. For this reason, no connection to a running MySQL Cluster or access to a `config.ini` or `my.cnf` file is required for these two options.

Combining other `ndb_config` options (such as `--query` or `--type`) with `--configinfo` or `--xml` is not supported. If you attempt to do so, the usual (current) result is that all other options besides `--configinfo` or `--xml` are simply ignored. However, this behavior is not guaranteed and is subject to change at any time. In addition, since `ndb_config` when used with the `--configinfo` option does not access the MySQL Cluster or read any files, trying to specify additional options such as `--ndb-connectstring` or `--config-file` with `--configinfo` serves no purpose.

Examples:

1. To obtain the node ID and type of each node in the cluster:

   ```
   shell> ./ndb_config --query=id,type --fields=':' --rows='\n'
   1:ndbd
   2:ndbd
   3:ndbd
   4:ndbd
   5:ndb_mgmd
   6:mysqld
   7:mysqld
   8:mysqld
   9:mysqld
   ```

   In this example, we used the `--fields` options to separate the ID and type of each node with a colon character (`:`), and the `--rows` options to place the values for each node on a new line in the output.

2. To produce a connectstring that can be used by data, SQL, and API nodes to connect to the management server:

   ```
   shell> ./ndb_config --config-file=usr/local/mysql/cluster-data/config.ini --query=hostname,portnumber --fields=: -
   192.168.0.179:1186
   ```

3. This invocation of `ndb_config` checks only data nodes (using the `--type` option), and shows the values for each node's ID and host name, and its `DataMemory`, `IndexMemory`, and `DataDir` parameters:

```
shell> ./ndb_config --type=ndbd --query=id,host,datamemory,indexmemory,datadir -f ' : ' -r '\n'
1 : 192.168.0.193 : 83886080 : 18874368 : /usr/local/mysql/cluster-data
2 : 192.168.0.112 : 83886080 : 18874368 : /usr/local/mysql/cluster-data
3 : 192.168.0.176 : 83886080 : 18874368 : /usr/local/mysql/cluster-data
4 : 192.168.0.119 : 83886080 : 18874368 : /usr/local/mysql/cluster-data
```

   In this example, we used the short options `-f` and `-r` for setting the field delimiter and row separator, respectively.

4. To exclude results from any host except one in particular, use the `--host` option:

```
shell> ./ndb_config --host=192.168.0.176 -f : -r '\n' -q id,type
3:ndbd
5:ndb_mgmd
```

   In this example, we also used the short form `-q` to determine the attributes to be queried.

   Similarly, you can limit results to a node with a specific ID using the `--id` or `--nodeid` option.

6.7. ndb_cpcd — Automate Testing for NDB Development

This utility is found in the `libexec` directory. It is part of an internal automated test framework used in testing and debugging MySQL Cluster. Because it can control processes on remote systems, it is not advisable to use `ndb_cpcd` in a production cluster.

The source files for `ndb_cpcd` may be found in the directory `storage/ndb/src/cw/cpcd`, in the MySQL Cluster source tree.

6.8. ndb_delete_all — Delete All Rows from an NDB Table

`ndb_delete_all` deletes all rows from the given `NDB` table. In some cases, this can be much faster than `DELETE` or even `TRUNCATE`.

Usage:

```
ndb_delete_all -c connect_string tbl_name -d db_name
```

This deletes all rows from the table named *`tbl_name`* in the database named *`db_name`*. It is exactly equivalent to executing `TRUNCATE` *`db_name.tbl_name`* in MySQL.

Additional Options:

- `--transactional`, `-t`

  Use of this option causes the delete operation to be performed as a single transaction.

  > **Warning**
  >
  > With very large tables, using this option may cause the number of operations available to the cluster to be exceeded.

6.9. ndb_desc — Describe NDB Tables

`ndb_desc` provides a detailed description of one or more `NDB` tables.

Usage:

```
ndb_desc -c connect_string tbl_name -d db_name [-p]
```

Sample Output:

MySQL table creation and population statements:

```
USE test;
CREATE TABLE fish (
```

```
    id INT(11) NOT NULL AUTO_INCREMENT,
    name VARCHAR(20),
    PRIMARY KEY pk (id),
    UNIQUE KEY uk (name)
) ENGINE=NDBCLUSTER;
INSERT INTO fish VALUES
    ('','guppy'), ('','tuna'), ('','shark'),
    ('','manta ray'), ('','grouper'), ('','puffer');
```

Output from `ndb_desc`:

```
shell> ./ndb_desc -c localhost fish -d test -p
-- fish --
Version: 16777221
Fragment type: 5
K Value: 6
Min load factor: 78
Max load factor: 80
Temporary table: no
Number of attributes: 2
Number of primary keys: 1
Length of frm data: 268
Row Checksum: 1
Row GCI: 1
TableStatus: Retrieved
-- Attributes --
id Int PRIMARY KEY DISTRIBUTION KEY AT=FIXED ST=MEMORY
name Varchar(20;latin1_swedish_ci) NULL AT=SHORT_VAR ST=MEMORY
-- Indexes --
PRIMARY KEY(id) - UniqueHashIndex
uk(name) - OrderedIndex
PRIMARY(id) - OrderedIndex
uk$unique(name) - UniqueHashIndex
-- Per partition info --
Partition  Row count  Commit count  Frag fixed memory  Frag varsized memory
2          2          2             65536              327680
1          2          2             65536              327680
3          2          2             65536              327680
NDBT_ProgramExit: 0 - OK
```

Additional Options:

- `--extra-partition-info`, `-p`

  Prints additional information about the table's partitions.

- Information about multiple tables can be obtained in a single invocation of `ndb_desc` by using their names, separated by spaces. All of the tables must be in the same database.

6.10. `ndb_drop_index` — Drop Index from an NDB Table

`ndb_drop_index` drops the specified index from an `NDB` table. *It is recommended that you use this utility only as an example for writing NDB API applications* — see the Warning later in this section for details.

Usage:

```
ndb_drop_index -c connect_string table_name index -d db_name
```

The statement shown above drops the index named *`index`* from the *`table`* in the *`database`*.

Additional Options: None that are specific to this application.

> **Warning**
>
> *Operations performed on Cluster table indexes using the NDB API are not visible to MySQL and make the table unusable by a MySQL server*. If you use this program to drop an index, then try to access the table from an SQL node, an error results, as shown here:

```
shell> ./ndb_drop_index -c localhost dogs ix -d ctest1
Dropping index dogs/idx...OK
NDBT_ProgramExit: 0 - OK
shell> ./mysql -u jon -p ctest1
Enter password: *******
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7 to server version: 5.1.12-beta-20060817
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
mysql> SHOW TABLES;
```

```
+-----------------+
| Tables_in_ctest1 |
+-----------------+
| a               |
| bt1             |
| bt2             |
| dogs            |
| employees       |
| fish            |
+-----------------+
6 rows in set (0.00 sec)
mysql> SELECT * FROM dogs;
ERROR 1296 (HY000): Got error 4243 'Index not found' from NDBCLUSTER
```

In such a case, your *only* option for making the table available to MySQL again is to drop the table and re-create it. You can use either the SQL statement`DROP TABLE` or the `ndb_drop_table` utility (see Section 6.11, "`ndb_drop_table` — Drop an NDB Table") to drop the table.

6.11. `ndb_drop_table` — Drop an NDB Table

`ndb_drop_table` drops the specified `NDB` table. (If you try to use this on a table created with a storage engine other than NDB, it fails with the error `723: No such table exists`.) This operation is extremely fast — in some cases, it can be an order of magnitude faster than using `DROP TABLE` on an `NDB` table from MySQL.

Usage:

```
ndb_drop_table -c connect_string tbl_name -d db_name
```

Additional Options: None.

6.12. `ndb_error_reporter` — NDB Error-Reporting Utility

`ndb_error_reporter` creates an archive from data node and management node log files that can be used to help diagnose bugs or other problems with a cluster. *It is highly recommended that you make use of this utility when filing reports of bugs in MySQL Cluster*.

Usage:

```
ndb_error_reporter path/to/config-file [username] [--fs]
```

This utility is intended for use on a management node host, and requires the path to the management host configuration file (`config.ini`). Optionally, you can supply the name of a user that is able to access the cluster's data nodes via SSH, in order to copy the data node log files. ndb_error_reporter then includes all of these files in archive that is created in the same directory in which it is run. The archive is named `ndb_error_report_YYYYMMDDHHMMSS.tar.bz2`, where *`YYYYMMDDHHMMSS`* is a datetime string.

If the `--fs` is used, then the data node file systems are also copied to the management host and included in the archive that is produced by this script. As data node file systems can be extremely large even after being compressed, we ask that you please do *not* send archives created using this option to MySQL AB unless you are specifically requested to do so.

Command Line Format	`--fs`	
Value Set	**Type**	`boolean`
	Default	`FALSE`

6.13. `ndb_print_backup_file` — Print NDB Backup File Contents

`ndb_print_backup_file` obtains diagnostic information from a cluster backup file.

Usage:

```
ndb_print_backup_file file_name
```

`file_name` is the name of a cluster backup file. This can be any of the files (`.Data`, `.ctl`, or `.log` file) found in a cluster backup directory. These files are found in the data node's backup directory under the subdirectory `BACKUP-#`, where `#` is the sequence number for the backup. For more information about cluster backup files and their contents, see Section 7.3.1, "MySQL Cluster Backup Concepts".

Like `ndb_print_schema_file` and `ndb_print_sys_file` (and unlike most of the other `NDB` utilities that are intended to

be run on a management server host or to connect to a management server) `ndb_print_backup_file` must be run on a cluster data node, since it accesses the data node file system directly. Because it does not make use of the management server, this utility can be used when the management server is not running, and even when the cluster has been completely shut down.

Additional Options: None.

6.14. `ndb_print_schema_file` — Print NDB Schema File Contents

`ndb_print_schema_file` obtains diagnostic information from a cluster schema file.

Usage:

```
ndb_print_schema_file file_name
```

`file_name` is the name of a cluster schema file. For more information about cluster schema files, see Cluster Data Node `FileSystemDir` Files.

Like `ndb_print_backup_file` and `ndb_print_sys_file` (and unlike most of the other `NDB` utilities that are intended to be run on a management server host or to connect to a management server) `ndb_schema_backup_file` must be run on a cluster data node, since it accesses the data node file system directly. Because it does not make use of the management server, this utility can be used when the management server is not running, and even when the cluster has been completely shut down.

Additional Options: None.

6.15. `ndb_print_sys_file` — Print NDB System File Contents

`ndb_print_sys_file` obtains diagnostic information from a MySQL Cluster system file.

Usage:

```
ndb_print_sys_file file_name
```

`file_name` is the name of a cluster system file (sysfile). Cluster system files are located in a data node's data directory (`DataDir`); the path under this directory to system files matches the pattern `ndb_#_fs/D#/DBDIH/P#.sysfile`. In each case, the *#* represents a number (not necessarily the same number). For more information, see Cluster Data Node `FileSystemDir` Files.

Like `ndb_print_backup_file` and `ndb_print_schema_file` (and unlike most of the other `NDB` utilities that are intended to be run on a management server host or to connect to a management server) `ndb_print_backup_file` must be run on a cluster data node, since it accesses the data node file system directly. Because it does not make use of the management server, this utility can be used when the management server is not running, and even when the cluster has been completely shut down.

Additional Options: None.

6.16. `ndbd_redo_log_reader` — Check and Print Content of Cluster Redo Log

Reads a redo log file, checking it for errors, printing its contents in a human-readable format, or both. `ndbd_redo_log_reader` is intended for use primarily by MySQL developers and support personnel in debugging and diagnosing problems.

This utility was made available as part of default builds beginning with MySQL Cluster NDB 6.1.3. It remains under development, and its syntax and behavior are subject to change in future releases. For this reason, it should be considered experimental at this time.

The C++ source files for `ndbd_redo_log_reader` can be found in the directory `/storage/ndb/src/kernel/blocks/dblqh/redoLogReader`.

Usage:

```
ndbd_redo_log_reader file_name [options]
```

`file_name` is the name of a cluster REDO log file. REDO log files are located in the numbered directories under the data node's data directory (`DataDir`); the path under this directory to the REDO log files matches the pattern `ndb_#_fs/D#/LCP/#/T#F#.Data`. In each case, the *#* represents a number (not necessarily the same number). For more information, see Cluster Data Node `FileSystemDir` Files.

Additional Options:

Command Line Format	-noprint	
Value Set	Type	boolean
	Default	FALSE

Command Line Format	-nocheck	
Value Set	Type	boolean
	Default	FALSE

The name of the file to be read may be followed by one or more of the options listed here:

- `-noprint`: Do not print the contents of the log file.
- `-nocheck`: Do not check the log file for errors.

Like `ndb_print_backup_file` and `ndb_print_schema_file` (and unlike most of the `NDB` utilities that are intended to be run on a management server host or to connect to a management server) `ndbd_redo_log_reader` must be run on a cluster data node, since it accesses the data node file system directly. Because it does not make use of the management server, this utility can be used when the management server is not running, and even when the cluster has been completely shut down.

6.17. `ndb_restore` — Restore a MySQL Cluster Backup

The cluster restoration program is implemented as a separate command-line utility `ndb_restore`, which can normally be found in the MySQL `bin` directory. This program reads the files created as a result of the backup and inserts the stored information into the database.

`ndb_restore` must be executed once for each of the backup files that were created by the `START BACKUP` command used to create the backup (see Section 7.3.2, “Using The MySQL Cluster Management Client to Create a Backup”). This is equal to the number of data nodes in the cluster at the time that the backup was created.

> **Note**
>
> Before using `ndb_restore`, it is recommended that the cluster be running in single user mode, unless you are restoring multiple data nodes in parallel. See Section 7.6, “MySQL Cluster Single User Mode”, for more information about single user mode.

Typical options for this utility are shown here:

```
ndb_restore [-c connectstring] -n
node_id [-s] [-m] -b backup_id -r --backup_path=/path/to/backup/files [-e]
```

The `-c` option is used to specify a connectstring which tells `ndb_restore` where to locate the cluster management server. (See Section 3.4.3, “The MySQL Cluster Connectstring”, for information on connectstrings.) If this option is not used, then `ndb_restore` attempts to connect to a management server on `localhost:1186`. This utility acts as a cluster API node, and so requires a free connection “slot” to connect to the cluster management server. This means that there must be at least one `[api]` or `[mysqld]` section that can be used by it in the cluster `config.ini` file. It is a good idea to keep at least one empty `[api]` or `[mysqld]` section in `config.ini` that is not being used for a MySQL server or other application for this reason (see Section 3.4.7, “Defining SQL and Other API Nodes in a MySQL Cluster”).

You can verify that `ndb_restore` is connected to the cluster by using the `SHOW` command in the `ndb_mgm` management client. You can also accomplish this from a system shell, as shown here:

```
shell> ndb_mgm -e "SHOW"
```

`-n` is used to specify the node ID of the data node on which the backups were taken.

The first time you run the `ndb_restore` restoration program, you also need to restore the metadata. In other words, you must re-create the database tables — this can be done by running it with the `-m` option. Note that the cluster should have an empty database when starting to restore a backup. (In other words, you should start `ndbd` with `--initial` prior to performing the restore. You should also remove manually any Disk Data files present in the data node's `DataDir`.)

It is possible to restore data without restoring table metadata. Prior to MySQL 5.1.17, `ndb_restore` did not perform any checks of table schemas; if a table was altered between the time the backup was taken and when `ndb_restore` was run, `ndb_restore` would still attempt to restore the data to the altered table.

Beginning with MySQL 5.1.17, the default behavior is for `ndb_restore` to fail with an error if table data do not match the table schema; this can be overridden using the `--skip-table-check` or `-s` option. Prior to MySQL 5.1.21, if this option is used, then `ndb_restore` attempts to fit data into the existing table schema, but the result of restoring a backup to a table schema that does not match the original is unspecified.

Beginning with MySQL Cluster NDB 6.3.8, `ndb_restore` supports limited *attribute promotion* in much the same way that it is supported by MySQL replication; that is, data backed up from a column of a given type can generally be restored to a column using a "larger, similar" type. For example, data from a `CHAR(20)` column can be restored to a column declared as `VARCHAR(20)`, `VARCHAR(30)`, or `CHAR(30)`; data from a `MEDIUMINT` column can be restored to a column of type `INT` or `BIGINT`. See Replication of Columns Having Different Data Types, for a table of type conversions currently supported by attribute promotion.

Attribute promotion by `ndb_restore` must be enabled explicitly, as follows:

1. Prepare the table to which the backup is to be restored. `ndb_restore` cannot be used to re-create the table with a different definition from the original; this means that you must either create the table manually, or alter the columns which you wish to promote using `ALTER TABLE` after restoring the table metadata but before restoring the data.

2. Invoke `ndb_restore` with the `--promote-attributes` option (short form `-A`) when restoring the table data. Attribute promotion does not occur if this option is not used; instead, the restore operation fails with an error.

In addition to `--promote-attributes`, a `--preserve-trailing-spaces` option is also available for use with `ndb_restore` beginning with MySQL Cluster NDB 6.3.8. This option (short form `-R`) causes trailing spaces to be preserved when promoting a `CHAR` column to `VARCHAR` or a `BINARY` column to `VARBINARY`. Otherwise, any trailing spaces are dropped from column values when they are inserted into the new columns.

> **Note**
>
> Although you can promote `CHAR` columns to `VARCHAR` and `BINARY` columns to `VARBINARY`, you cannot promote `VARCHAR` columns to `CHAR` or `VARBINARY` columns to `BINARY`.

The `-b` option is used to specify the ID or sequence number of the backup, and is the same number shown by the management client in the `Backup backup_id completed` message displayed upon completion of a backup. (See Section 7.3.2, "Using The MySQL Cluster Management Client to Create a Backup".)

> **Important**
>
> When restoring cluster backups, you must be sure to restore all data nodes from backups having the same backup ID. Using files from different backups will at best result in restoring the cluster to an inconsistent state, and may fail altogether.

`-e` adds (or restores) epoch information to the cluster replication status table. This is useful for starting replication on a MySQL Cluster replication slave. When this option is used, the row in the `mysql.ndb_apply_status` having `0` in the `id` column is updated if it already exists; such a row is inserted if it does not already exist. (See Section 9.9, "MySQL Cluster Backups With MySQL Cluster Replication".)

The path to the backup directory is required; this is supplied to `ndb_restore` using the `--backup_path` option, and must include the subdirectory corresponding to the ID backup of the backup to be restored. For example, if the data node's `DataDir` is `/var/lib/mysql-cluster`, then the backup directory is `/var/lib/mysql-cluster/BACKUP`, and the backup files for the backup with the ID 3 can be found in `/var/lib/mysql-cluster/BACKUP/BACKUP-3`. The path may be absolute or relative to the directory in which the `ndb_restore` executable is located.

> **Note**
>
> Previous to MySQL 5.1.17 and MySQL Cluster NDB 6.1.5, the path to the backup directory was specified as shown here, with `backup_path=` being optional:
>
> ```
> [backup_path=]/path/to/backup/files
> ```
>
> Beginning with MySQL 5.1.17 and MySQL Cluster NDB 6.1.5, this syntax changed to `--backup_path=/path/to/backup/files`, to conform more closely with options used by other MySQL programs; `--backup_id` is required, and there is no short form for this option.

It is possible to restore a backup to a database with a different configuration than it was created from. For example, suppose that a backup with backup ID `12`, created in a cluster with two database nodes having the node IDs `2` and `3`, is to be restored to a cluster

with four nodes. Then `ndb_restore` must be run twice — once for each database node in the cluster where the backup was taken. However, `ndb_restore` cannot always restore backups made from a cluster running one version of MySQL to a cluster running a different MySQL version. See Section 5.2, "MySQL Cluster 5.1 and MySQL Cluster NDB 6.x/7.x Upgrade and Downgrade Compatibility", for more information.

> **Important**
>
> It is not possible to restore a backup made from a newer version of MySQL Cluster using an older version of `ndb_restore`. You can restore a backup made from a newer version of MySQL to an older cluster, but you must use a copy of `ndb_restore` from the newer MySQL Cluster version to do so.
>
> For example, to restore a cluster backup taken from a cluster running MySQL Cluster NDB 6.2.15 to a cluster running MySQL 5.1.20, you must use a copy of `ndb_restore` from the MySQL Cluster NDB 6.2.15 distribution.

For more rapid restoration, the data may be restored in parallel, provided that there is a sufficient number of cluster connections available. That is, when restoring to multiple nodes in parallel, you must have an `[api]` or `[mysqld]` section in the cluster `config.ini` file available for each concurrent `ndb_restore` process. However, the data files must always be applied before the logs.

Formerly, when using `ndb_restore` to restore a backup made from a MySQL 5.0 cluster to a 5.1 cluster, `VARCHAR` columns were not resized and were recreated using the 5.0 fixed format. Beginning with MySQL 5.1.19, `ndb_restore` recreates such `VARCHAR` columns using MySQL Cluster 5.1's variable-width format. Also beginning with MySQL 5.1.19, this behavior can be overridden using the `--no-upgrade` option (short form: `-u`) when running `ndb_restore`.

Most of the options available for this program are shown in the following table:

Long Form	Short Form	Description	Default Value
`--backup-id`	`-b`	Backup sequence ID	*None*
`--backup_path` (added in MySQL 5.1.17 and MySQL Cluster NDB 6.1.5; previously this was `backup_path` — see Note in text)	*None*	Path to backup files	*None*
`--character-sets-dir`	*None*	Specify the directory where character set information can be found	*None*
`--connect`, `--connectstring`, or `--ndb-connectstring`	`-c` or `-C`	Set the connectstring in	`localhost:1186`

		] format	
--core-file	*None*	Write a core file in the event of an error	TRUE
--debug	-#	Output debug log	d:t:O,/tmp/ndb_restore.trace
--dont_ignore_systab_0	-f	Do not ignore system table during restore — *EXPERIMENTAL; not for production use*	FALSE
--exclude-databases=*db_list*	*None*	Do not restore the indicated database or databases (added in MySQL Cluster NDB 6.3.22 and 6.4.3)	[N/A]
--exclude-tables=*tbl_list*	*None*	Do not restore the indicated table or tables; each table must be specified using *database.table* format (added in MySQL Cluster NDB 6.3.22 and 6.4.3)	[N/A]
--help or --usage	-?	Display help message with available options and current values, then exit	[N/A]
--include-databases=*db_list*	*None*	Restore only the indicated database or databases (added in MySQL Cluster NDB 6.3.22 and 6.4.3)	[N/A]
--include-tables=*tbl_list*	*None*	Restore only the indicated table or tables; each table must be specified using *database.table* format (added in MySQL Cluster NDB 6.3.22 and 6.4.3)	[N/A]
--ndb-mgmd-host	*None*	Set the host and port in *host[:port]* format for the management server to connect to; this is the same as --connect, --connectstring, or --ndb-connectstring, but without a way to specify the nodeid	*None*
--ndb-nodegroup-map	-z	Specifies a nodegroup map — *Syntax*: list of (*source_nodegroup*, *destination_nodegroup*)	*None*
--ndb-nodeid	*None*	Specify a node ID for the ndb_restore process	0
--ndb-optimized-node-selection	*None*	Optimize selection of nodes for transactions	TRUE
--ndb-shm	*None*	Use shared memory connections when available	FALSE
--no-binlog	*None*	Do not write anything to mysqld binary logs (added in MySQL Cluster NDB 6.2.16 and 6.3.16)	FALSE (in other words, write to binary logs unless this option is used)
--no-restore-disk-objects	-d	Do not restore Disk Data objects such as tablespaces and log file groups	FALSE (in other words, restore Disk Data objects unless this option is used)
--no-upgrade	-u	Do not re-create VARSIZE columns from a MySQL 5.0 Cluster backup as variable-width columns (added in MySQL 5.1.19)	FALSE (in other words, re-create VARSIZE columns from a MySQL 5.0 Cluster backup as variable-width columns unless this option is used)
--nodeid	-n	Use backup files from node with the specified ID	0

--parallelism	-p	Set from 1 to 1024 parallel transactions to be used during the restoration process	128
--print	*None*	Print metadata, data, and log to stdout	FALSE
--print_data	*None*	Print data to stdout	FALSE
--print_log	*None*	Print log to stdout	FALSE
--print_meta	*None*	Print metadata to stdout	FALSE
--restore_data	-r	Restore data and logs	FALSE
--restore_epoch	-e	Restore epoch data into the status table; the row in the cluster.apply_status having the id 0 is inserted or updated as appropriate — this is convenient when starting up replication on a MySQL Cluster replication slave	FALSE
--restore_meta	-m	Restore table metadata	FALSE
--skip-table-check	-s	Do not check table schemas (Added in MySQL 5.1.17)	FALSE
--version	-V	Output version information and exit	[N/A]

Beginning with MySQL 5.1.18, several additional options are available for use with the --print_data option in generating data dumps, either to stdout, or to a file. These are similar to some of the options used with mysqldump, and are shown in the following table:

Long Form	Short Form	Description	Default Value
--tab	-T	Creates dumpfiles, one per table, each named *tbl_name*.txt. Takes as its argument the path to the directory where the files should be saved (required; use . for the current directory).	*None*
--fields-enclosed-by	*None*	String used to enclose all column values	*None*
--fields-option-ally-enclosed-by	*None*	String used to enclose column values containing character data (such as CHAR, VARCHAR, BINARY, TEXT, or ENUM)	*None*
--fields-terminated-by	*None*	String used to separate column values	\t (tab character)
--hex	*None*	Use hex format for binary values	[N/A]
--lines-terminated-by	*None*	String used to terminate each line	\n (linefeed character)
--append	*None*	When used with --tab, causes the data to be appended to existing files of the same name	[N/A]

> **Note**
>
> If a table has no explicit primary key, then the output generated when using the --print includes the table's hidden primary key.

Beginning with MySQL 5.1.18, it is possible to restore selected databases, or to restore selected tables from a given database using the syntax shown here:

```
ndb_restore other_options db_name_1 [db_name_2[, db_name_3][, ...] | tbl_name_1[, tbl_name_2][, ...]]
```

In other words, you can specify either of the following to be restored:

- All tables from one or more databases
- One or more tables from a single database

Beginning with MySQL Cluster NDB 6.3.22 and 6.4.3, you can (and should) use instead the options --include-databases and --include-tables for restoring only specific databases or tables, respectively. --include-databases takes a comma-delimited list of databases to be restored. --include-tables takes a comma-delimited list of tables (in *database*.*table* format) to be restored. You can use these two options together. For example, the following causes all tables in databases db1 and db2, together with the tables t1 and t2 in database db3, to be restored (and no other databases or tables):

```
shell> ndb_restore [...] --include-databases=db1,db2 --include-tables=db3.t1,db3.t2
```

(For the sake of clarity and brevity, we have omitted other, possibly required, options in the example just shown.) When `--include-databases`, `--include-tables`, or both are used, only those databases or tables specified are restored; all other databases and tables are ignored by `ndb_restore`.

Also beginning with MySQL Cluster NDB 6.3.22 and 6.4.3, it is possible to exclude from being restored one or more databases, tables, or both using the `ndb_restore` options `--exclude-databases` and `--exclude-tables`. `--exclude-databases` takes a comma-delimited list of one or more databases which should not be restored. `--exclude-tables` takes a comma-delimited list of one or more tables, using *`database.table`* format, which should not be restored. You can use these two options together. For example, the following causes all tables in all databases *except for* databases `db1` and `db2`, along with the tables `t1` and `t2` in database `db3`, *not* to be restored:

```
shell> ndb_restore [...] --exclude-databases=db1,db2 --exclude-tables=db3.t1,db3.t2
```

(Again, we have omitted other possibly necessary options in the interest of clarity and brevity from the example just shown.)

You should not use `--include-databases` or `--include-tables` together with `--exclude-databases` or `--exclude-tables`, since `--include-databases` and `--include-tables` exclude all databases and tables not explicitly named. Similarly, `--exclude-databases` and `--exclude-tables` include all databases and tables not listed in the arguments to these options.

Error reporting. `ndb_restore` reports both temporary and permanent errors. In the case of temporary errors, it may able to recover from them. Beginning with MySQL 5.1.12, it reports `Restore successful, but encountered temporary error, please look at configuration` in such cases.

> **Important**
>
> After using `ndb_restore` to initialize a MySQL Cluster for use in circular replication, binary logs on the SQL node acting as the replication slave are not automatically created, and you must cause them to be created manually. In order to cause the binary logs to be created, issue a `SHOW TABLES` statement on that SQL node before running `START SLAVE`.
>
> This is a known issue with MySQL Cluster management, which we intend to address in a future release.

6.18. `ndb_select_all` — Print Rows from an NDB Table

`ndb_select_all` prints all rows from an `NDB` table to `stdout`.

Usage:

```
ndb_select_all -c connect_string tbl_name -d db_name [> file_name]
```

Additional Options:

- `--lock=lock_type`, `-l lock_type`

  Employs a lock when reading the table. Possible values for *`lock_type`* are:

  - `0`: Read lock
  - `1`: Read lock with hold
  - `2`: Exclusive read lock

  There is no default value for this option.

- `--order=index_name`, `-o index_name`

  Orders the output according to the index named *`index_name`*. Note that this is the name of an index, not of a column, and that the index must have been explicitly named when created.

- `--descending`, `-z`

  Sorts the output in descending order. This option can be used only in conjunction with the `-o` (`--order`) option.

- `--header=FALSE`

Excludes column headers from the output.

- `--useHexFormat -x`

  Causes all numeric values to be displayed in hexadecimal format. This does not affect the output of numerals contained in strings or datetime values.

- `--delimiter=character`, `-D character`

  Causes the *`character`* to be used as a column delimiter. Only table data columns are separated by this delimiter.

  The default delimiter is the tab character.

- `--disk`

  Adds a disk reference column to the output. The column is non-empty only for Disk Data tables having non-indexed columns.

- `--rowid`

  Adds a `ROWID` column providing information about the fragments in which rows are stored.

- `--gci`

  Adds a column to the output showing the global checkpoint at which each row was last updated. See Chapter 14, *MySQL Cluster Glossary*, and Section 7.4.2, "MySQL Cluster Log Events", for more information about checkpoints.

- `--tupscan`, `-t`

  Scan the table in the order of the tuples.

- `--nodata`

  Causes any table data to be omitted.

Sample Output:

Output from a MySQL `SELECT` statement:

```
mysql> SELECT * FROM ctest1.fish;
+----+-----------+
| id | name      |
+----+-----------+
|  3 | shark     |
|  6 | puffer    |
|  2 | tuna      |
|  4 | manta ray |
|  5 | grouper   |
|  1 | guppy     |
+----+-----------+
6 rows in set (0.04 sec)
```

Output from the equivalent invocation of `ndb_select_all`:

```
shell> ./ndb_select_all -c localhost fish -d ctest1
id      name
3       [shark]
6       [puffer]
2       [tuna]
4       [manta ray]
5       [grouper]
1       [guppy]
6 rows returned
NDBT_ProgramExit: 0 - OK
```

Note that all string values are enclosed by square brackets ("`[...]`") in the output of `ndb_select_all`. For a further example, consider the table created and populated as shown here:

```
CREATE TABLE dogs (
    id INT(11) NOT NULL AUTO_INCREMENT,
    name VARCHAR(25) NOT NULL,
    breed VARCHAR(50) NOT NULL,
    PRIMARY KEY pk (id),
    KEY ix (name)
)
TABLESPACE ts STORAGE DISK
ENGINE=NDBCLUSTER;
INSERT INTO dogs VALUES
```

```
    ('', 'Lassie', 'collie'),
    ('', 'Scooby-Doo', 'Great Dane'),
    ('', 'Rin-Tin-Tin', 'Alsatian'),
    ('', 'Rosscoe', 'Mutt');
```

This demonstrates the use of several additional `ndb_select_all` options:

```
shell> ./ndb_select_all -d ctest1 dogs -o ix -z --gci --disk
GCI     id name          breed        DISK_REF
834461  2  [Scooby-Doo]  [Great Dane] [ m_file_no: 0 m_page: 98 m_page_idx: 0 ]
834878  4  [Rosscoe]     [Mutt]       [ m_file_no: 0 m_page: 98 m_page_idx: 16 ]
834463  3  [Rin-Tin-Tin] [Alsatian]   [ m_file_no: 0 m_page: 34 m_page_idx: 0 ]
835657  1  [Lassie]      [Collie]     [ m_file_no: 0 m_page: 66 m_page_idx: 0 ]
4 rows returned
NDBT_ProgramExit: 0 - OK
```

6.19. `ndb_select_count` — Print Row Counts for NDB Tables

`ndb_select_count` prints the number of rows in one or more `NDB` tables. With a single table, the result is equivalent to that obtained by using the MySQL statement `SELECT COUNT(*) FROM tbl_name`.

Usage:

```
ndb_select_count [-c connect_string] -ddb_name tbl_name[, tbl_name2[, ...]]
```

Additional Options: None that are specific to this application. However, you can obtain row counts from multiple tables in the same database by listing the table names separated by spaces when invoking this command, as shown under **Sample Output**.

Sample Output:

```
shell> ./ndb_select_count -c localhost -d ctest1 fish dogs
6 records in table fish
4 records in table dogs
NDBT_ProgramExit: 0 - OK
```

6.20. `ndb_show_tables` — Display List of NDB Tables

`ndb_show_tables` displays a list of all `NDB` database objects in the cluster. By default, this includes not only both user-created tables and `NDB` system tables, but `NDB`-specific indexes, internal triggers, and MySQL Cluster Disk Data objects as well.

Usage:

```
ndb_show_tables [-c connect_string]
```

Additional Options:

- `--loops`, `-l`

  Specifies the number of times the utility should execute. This is 1 when this option is not specified, but if you do use the option, you must supply an integer argument for it.

- `--parsable`, `-p`

  Using this option causes the output to be in a format suitable for use with `LOAD DATA INFILE`.

- `--type`, `-t`

  Can be used to restrict the output to one type of object, specified by an integer type code as shown here:

  - **1**: System table
  - **2**: User-created table
  - **3**: Unique hash index

  Any other value causes all `NDB` database objects to be listed (the default).

- `--unqualified`, `-u`

  If specified, this causes unqualified object names to be displayed.

> **Note**
>
> Only user-created Cluster tables may be accessed from MySQL; system tables such as `SYSTAB_0` are not visible to `mysqld`. However, you can examine the contents of system tables using `NDB` API applications such as `ndb_select_all` (see Section 6.18, "`ndb_select_all` — Print Rows from an NDB Table").

6.21. `ndb_size.pl` — NDBCLUSTER Size Requirement Estimator

This is a Perl script that can be used to estimate the amount of space that would be required by a MySQL database if it were converted to use the `NDBCLUSTER` storage engine. Unlike the other utilities discussed in this section, it does not require access to a MySQL Cluster (in fact, there is no reason for it to do so). However, it does need to access the MySQL server on which the database to be tested resides.

Requirements:

- A running MySQL server. The server instance does not have to provide support for MySQL Cluster.
- A working installation of Perl.
- The `DBI` module, which can be obtained from CPAN if it is not already part of your Perl installation. (Many Linux and other operating system distributions provide their own packages for this library.)
- Previous to MySQL 5.1.18, `ndb_size.pl` also required the `HTML::Template` module and an associated template file `share/mysql/ndb_size.tmpl`. Beginning with MySQL 5.1.18, `ndb_size.tmpl` is no longer needed (or included).
- A MySQL user account having the necessary privileges. If you do not wish to use an existing account, then creating one using `GRANT USAGE ON db_name.*` — where *`db_name`* is the name of the database to be examined — is sufficient for this purpose.

`ndb_size.pl` can also be found in the MySQL sources in `storage/ndb/tools`. If this file is not present in your MySQL installation, you can obtain it from the MySQL Forge project page.

Usage:

```
perl ndb_size.pl db_name|ALL] [--hostname=host[:port]] [--socket=socket] [--user=user] \
    [--password=password] [--help|-h] [--format=(html|text)] [--loadqueries=file_name] [--savequeries=file_name]
```

By default, this utility attempts to analyze all databases on the server. You can specify a single database using the `--database` option; the default behavior can be made explicit by using `ALL` for the name of the database. You can also exclude one or more databases by using the `--excludedbs` with a comma-separated list of the names of the databases to be skipped. Similarly, you can cause specific tables to be skipped by listing their names, separated by commas, following the optional `--excludetables` option. A host name (and possibly a port as well) can be specified using `--hostname`; the default is `localhost:3306`. If necessary, you can specify a socket; the default is `/var/lib/mysql.sock`. A MySQL user name and password can be specified the corresponding options shown. It also possible to control the format of the output using the `--format` option; this can take either of the values `html` or `text`, with `text` being the default. An example of the text output is shown here:

```
shell> ndb_size.pl --database=test --socket=/tmp/mysql.sock
ndb_size.pl report for database: 'test' (1 tables)
------------------------------------------------------
Connected to: DBI:mysql:host=localhost;mysql_socket=/tmp/mysql.sock
Including information for versions: 4.1, 5.0, 5.1
test.t1
-------
DataMemory for Columns (* means varsized DataMemory):
            Column Name                Type  Varsized   Key       4.1        5.0        5.1
         HIDDEN_NDB_PKEY              bigint             PRI         8          8          8
                      c2         varchar(50)         Y             52         52         4*
                      c1             int(11)                        4          4          4
                                                                   --         --         --
Fixed Size Columns DM/Row                                          64         64         12
   Varsize Columns DM/Row                                           0          0          4
DataMemory for Indexes:
               Index Name                 Type        4.1        5.0        5.1
                  PRIMARY                BTREE         16         16         16
                                                       --         --         --
      Total Index DM/Row                               16         16         16
IndexMemory for Indexes:
               Index Name        4.1        5.0        5.1
                  PRIMARY         33         16         16
                                  --         --         --
          Indexes IM/Row          33         16         16
Summary (for THIS table):
                                 4.1        5.0        5.1
    Fixed Overhead DM/Row         12         12         16
          NULL Bytes/Row           4          4          4
          DataMemory/Row          96         96         48  (Includes overhead, bitmap and indexes)
```

```
  Varsize Overhead DM/Row          0          0          8
   Varsize NULL Bytes/Row          0          0          4
       Avg Varside DM/Row          0          0         16
                  No. Rows         0          0          0
        Rows/32kb DM Page        340        340        680
Fixedsize DataMemory (KB)          0          0          0
Rows/32kb Varsize DM Page          0          0       2040
  Varsize DataMemory (KB)          0          0          0
         Rows/8kb IM Page        248        512        512
         IndexMemory (KB)          0          0          0
Parameter Minimum Requirements
------------------------------
* indicates greater than default
                Parameter          Default             4.1             5.0             5.1
          DataMemory (KB)            81920               0               0               0
       NoOfOrderedIndexes              128               1               1               1
                NoOfTables             128               1               1               1
         IndexMemory (KB)            18432               0               0               0
    NoOfUniqueHashIndexes               64               0               0               0
           NoOfAttributes             1000               3               3               3
             NoOfTriggers              768               5               5               5
```

For debugging purposes, the Perl arrays containing the queries run by this script can be read from the file specified using can be saved to a file using `--savequeries`; a file containing such arrays to be read in during script execution can be specified using `--loadqueries`. Neither of these options has a default value.

To produce output in HTML format, use the `--format` option and redirect the output to a file, as shown in this example:

```
shell> ndb_size.pl --database=test --socket=/tmp/mysql.sock --format=html > ndb_size.html
```

(Without the redirection, the output is sent to `stdout`.) This figure shows a portion of the generated `ndb_size.html` output file, as viewed in a Web browser:

MySQL Cluster analysis for world

This is an automated analysis of the DBI:mysql:database=world;host=192.168.0.176 database for migration into MySQL Cluster. No warranty is made to the accuracy of the information.

This information should be valid for MySQL 4.1 and 5.0. Since 5.1 is not a final release yet, the numbers should be used as a guide only.

Parameter Settings

NOTE the configuration parameters below do not take into account system tables and other requirements.

Parameter	4.1	5.0	5.1
DataMemory (kb)	544	544	544
IndexMemory (kb)	192	136	136
MaxNoOfTables	3	3	3
MaxNoOfAttributes	24	24	24
MaxNoOfOrderedIndexes	3	3	3
MaxNoOfUniqueHashIndexes	3	3	3
MaxNoOfTriggers	12	12	12

Memory usage because of parameters

Usage is in kilobytes. Actual usage will vary as you should set the parameters larger than those listed in the table above.

Parameter	4.1	5.0	5.1
Attributes	5	5	5
Tables	60	60	60
OrderedIndexes	30	30	30
UniqueHashIndexes	45	45	45

Table List

- City
- Country
- CountryLanguage

City

Column	Type	Size	Key	4.1 NDB Size	5.0 NDB Size	5.1 NDB Size
ID	int	11	PRI	4	4	4
District	char	20		20	20	20
Name	char	35		36	36	36
Population	int	11		4	4	4
CountryCode	char	3		4	4	4

Indexes

We assume that indexes are ORDERED (not created USING HASH). If order is not required, 10 bytes of data memory can be saved per row if the index is created USING HASH

Index	Type	Columns	4.1 IdxMem	5.0 IdxMem	5.1 IdxMem	4.1 DatMem	5.0 DatMem	5.1 DatMem
PRIMARY	BTREE	ID	29	25	25	10	10	10

DataMemory Usage

	4.1	5.0	5.1
Row Overhead	16	16	16
Column DataMemory/Row	68	68	68
Index DataMemory/Row	10	10	10
Total DataMemory/Row	94	94	94
Rows per 32kb page	347	347	347
Current number of rows	4079	4079	4079
Total DataMemory (kb)	384	384	384

IndexMemory Usage

	4.1	5.0	5.1
IndexMemory/Row	29	25	25
Rows per 8kb page	282	327	327
Current number of rows	4079	4079	4079
Total IndexMemory (kb)	120	104	104

The output from this script includes:

- Minimum values for the `DataMemory`, `IndexMemory`, `MaxNoOfTables`, `MaxNoOfAttributes`, `MaxNoOfOrderedIndexes`, `MaxNoOfUniqueHashIndexes`, and `MaxNoOfTriggers` configuration parameters required to accommodate the tables analyzed.
- Memory requirements for all of the tables, attributes, ordered indexes, and unique hash indexes defined in the database.
- The `IndexMemory` and `DataMemory` required per table and table row.

> **Note**
>
> Prior to MySQL 5.1.23, MySQL Cluster NDB 6.2.5, and MySQL Cluster NDB 6.3.7, `ndb_size.pl` was invoked as shown here:
>
> ```
> perl ndb_size.pl db_name hostname username password > file_name.html
> ```
>
> For more information about this change, see Bug#28683 and Bug#28253.

6.22. `ndb_waiter` — Wait for MySQL Cluster to Reach a Given Status

`ndb_waiter` repeatedly (each 100 milliseconds) prints out the status of all cluster data nodes until either the cluster reaches a given status or the `--timeout` limit is exceeded, then exits. By default, it waits for the cluster to achieve `STARTED` status, in which all nodes have started and connected to the cluster. This can be overridden using the `--no-contact` and `--not-started` options (see Additional Options).

The node states reported by this utility are as follows:

- `NO_CONTACT`: The node cannot be contacted.
- `UNKNOWN`: The node can be contacted, but its status is not yet known. Usually, this means that the node has received a `START` or `RESTART` command from the management server, but has not yet acted on it.
- `NOT_STARTED`: The node has stopped, but remains in contact with the cluster. This is seen when restarting the node using the management client's `RESTART` command.
- `STARTING`: The node's `ndbd` process has started, but the node has not yet joined the cluster.
- `STARTED`: The node is operational, and has joined the cluster.
- `SHUTTING_DOWN`: The node is shutting down.
- `SINGLE USER MODE`: This is shown for all cluster data nodes when the cluster is in single user mode.

Usage:

```
ndb_waiter [-c connect_string]
```

Additional Options:

- `--no-contact`, `-n`

  Instead of waiting for the `STARTED` state, `ndb_waiter` continues running until the cluster reaches `NO_CONTACT` status before exiting.

- `--not-started`

  Instead of waiting for the `STARTED` state, `ndb_waiter` continues running until the cluster reaches `NOT_STARTED` status before exiting.

- `--timeout=seconds`, `-t seconds`

  Time to wait. The program exits if the desired state is not achieved within this number of seconds. The default is 120 seconds (1200 reporting cycles).

Sample Output. Shown here is the output from `ndb_waiter` when run against a 4-node cluster in which two nodes have been shut down and then started again manually. Duplicate reports (indicated by "`...`") are omitted.

```
shell> ./ndb_waiter -c localhost
Connecting to mgmsrv at (localhost)
State node 1 STARTED
State node 2 NO_CONTACT
State node 3 STARTED
State node 4 NO_CONTACT
Waiting for cluster enter state STARTED
...
State node 1 STARTED
State node 2 UNKNOWN
State node 3 STARTED
State node 4 NO_CONTACT
Waiting for cluster enter state STARTED
...
State node 1 STARTED
State node 2 STARTING
State node 3 STARTED
State node 4 NO_CONTACT
Waiting for cluster enter state STARTED
...
State node 1 STARTED
State node 2 STARTING
State node 3 STARTED
State node 4 UNKNOWN
Waiting for cluster enter state STARTED
...
State node 1 STARTED
State node 2 STARTING
State node 3 STARTED
State node 4 STARTING
Waiting for cluster enter state STARTED
...
State node 1 STARTED
State node 2 STARTED
State node 3 STARTED
State node 4 STARTING
Waiting for cluster enter state STARTED
...
State node 1 STARTED
State node 2 STARTED
State node 3 STARTED
State node 4 STARTED
Waiting for cluster enter state STARTED
NDBT_ProgramExit: 0 - OK
```

> **Note**
>
> If no connectstring is specified, then `ndb_waiter` tries to connect to a management on `localhost`, and reports `Connecting to mgmsrv at (null)`.

6.23. Options Common to MySQL Cluster Programs

All MySQL Cluster programs (except for `mysqld`) take the options described in this section. Users of earlier MySQL Cluster versions should note that some of these options have been changed to make them consistent with one another as well as with `mysqld`. You can use the `--help` option with any MySQL Cluster program to view a list of the options which it supports.

The options in the following list are common to all MySQL Cluster executables.

For options specific to individual `NDB` programs, see Chapter 6, *MySQL Cluster Programs*.

See Section 4.2, "`mysqld` Command Options for MySQL Cluster", for `mysqld` options relating to MySQL Cluster.

- `--help --usage`, `-?`

Command Line Format	`--help`

Prints a short list with descriptions of the available command options.

- `--character-sets-dir=`*`name`*

Command Line Format	`--character-sets-dir=name`	
Value Set	**Type**	`filename`
	Default	

Tells the program where to find character set information.

- `--connect-string=connect_string`, `-c connect_string`

Command Line Format	--ndb-connectstring=name	
Value Set	Type	string
	Default	localhost:1186

connect_string sets the connectstring to the management server as a command option.

```
shell> ndbd --connect-string="nodeid=2;host=ndb_mgmd.mysql.com:1186"
```

For more information, see Section 3.4.3, “The MySQL Cluster Connectstring”.

- `--core-file`

Command Line Format	--core-file	
Value Set	Type	boolean
	Default	FALSE

Write a core file if the program dies. The name and location of the core file are system-dependent. (For MySQL Cluster programs nodes running on Linux, the default location is the program's working directory — for a data node, this is the node's `DataDir`.) For some systems, there may be restrictions or limitations; for example, it might be necessary to execute `ulimit -c unlimited` before starting the server. Consult your system documentation for detailed information.

If MySQL Cluster was built using the `--debug` option for `configure`, then `--core-file` is enabled by default. For regular builds, `--core-file` is disabled by default.

- `--debug[=options]`

Command Line Format	--debug=options

This option can be used only for versions compiled with debugging enabled. It is used to enable output from debug calls in the same manner as for the `mysqld` process.

- `--execute=command`, `-e command`

Command Line Format	--execute=name

Can be used to send a command to a Cluster executable from the system shell. For example, either of the following:

```
shell> ndb_mgm -e "SHOW"
```

or

```
shell> ndb_mgm --execute="SHOW"
```

is equivalent to

```
ndb_mgm> SHOW
```

This is analogous to how the `--execute` or `-e` option works with the `mysql` command-line client. See Using Options on the Command Line.

- `--ndb-mgmd-host=host[:port]`

Can be used to set the host and port number of the management server to connect to.

- `--ndb-nodeid=#`

  Sets this node's MySQL Cluster node ID. *The range of permitted values depends on the type of the node (data, management, or API) and the version of the MySQL Cluster software which is running on it.* See Section 12.2, "Limits and Differences of MySQL Cluster from Standard MySQL Limits", for more information.

- `--ndb-optimized-node-selection`

Command Line Format	`--ndb-optimized-node-selection`	
Value Set	**Type**	`boolean`
	Default	`TRUE`

  Optimize selection of nodes for transactions. Enabled by default.

- `--version`, `-V`

Command Line Format	`-V`

  Prints the MySQL Cluster version number of the executable. The version number is relevant because not all versions can be used together, and the MySQL Cluster startup process verifies that the versions of the binaries being used can co-exist in the same cluster. This is also important when performing an online (rolling) software upgrade or downgrade of MySQL Cluster. (See Section 5.1, "Performing a Rolling Restart of a MySQL Cluster").

6.24. Summary Tables of NDB Program Options

The next few sections contain summary tables providing basic information about command-line options used with MySQL Cluster programs.

6.24.1. Common Options for MySQL Cluster Programs

Table 6.1. `ndb_common` Option Reference

Format	Description	Introduction	Deprecated	Removed
--character-sets-dir=name	Directory where character sets are			
--ndb-connectstring=name	Set connect string for connecting to ndb_mgmd. Syntax: [nodeid=<id>;][host=]<hostname>[:<port>]. Overrides specifying entries in NDB_CONNECTSTRING and my.cnf			
--core-file	Write core on errors (defaults to TRUE in debug builds)			
--debug=options	Enable output from debug calls. Can be used only for versions compiled with debugging enabled			
--execute=name	Execute command and exit			
--help	Display help message and exit			
--ndb-mgmd-host=name	Set host and port for connecting to ndb_mgmd. Syntax: <hostname>[:<port>]. Overrides specifying entries in NDB_CONNECTSTRING and my.cnf			
--ndb-optimized-node-selection	Select nodes for transactions in a more optimal way			
--ndb-nodeid=#	Set node id for this node			
-V	Output version information and exit			

For options specific to individual MySQL Cluster programs, see Section 6.23, “Options Common to MySQL Cluster Programs”.

6.24.2. Program Options for ndbd and ndbmtd

Table 6.2. ndbd Option Reference

Format	Description	Introduction	Deprecated	Removed
--bind-address=name	Local bind address	5.1.12		
--daemon	Start ndbd as daemon (default); override with --nodaemon			
--foreground	Run ndbd in foreground, provided for debugging purposes (implies --nodaemon)			
--initial	Perform initial start of ndbd, including cleaning the file system. Consult the documentation before using this option			
--initial-start	Perform partial initial start (requires --nowait-nodes)	5.1.11		
--nodaemon	Do not start ndbd as daemon; provided for testing purposes			
--nostart	Don't start ndbd immediately; ndbd waits for command to start from ndb_mgmd			
--nowait-nodes=list	Nodes that will not be waited for during start (comma-deparated list of node IDs)	5.1.11		

For more information about ndbd, see Section 6.2, “ndbd — The MySQL Cluster Data Node Daemon”. For options common to all MySQL Cluster programs, see Section 6.23, “Options Common to MySQL Cluster Programs”.

6.24.3. Program Options for ndb_mgmd

Table 6.3. ndb_mgmd Option Reference

Format	Description	Introduction	Deprecated	Removed
--bind-address	Local bind address	5.1.22-ndb-6.3.2		
-c	Specify the cluster configuration file; in NDB-6.4.0 and later, needs --reload or --initial to override configuration cache if present			
--configdir=directory	Specify the cluster management server's configuration cache directory	5.1.30-ndb-6.4.0		
--daemon	Run ndb_mgmd in daemon mode (default)			
--initial	Causes the management server reload its configuration data from the configuration file, bypassing the configuration cache	5.1.30-ndb-6.4.0		
--interactive	Run ndb_mgmd in interactive mode (not officially supported in production; for testing purposes only)			
--mycnf	Read cluster configuration data from the my.cnf file			
--no-nodeid-checks	Do not provide any node id checks			
--nodaemon	Do not run ndb_mgmd as a daemon			
--print-full-config	Print full configuration and exit			
--reload	Causes the management server to compare the configuration file with its configuration cache	5.1.30-ndb-6.4.0		

For more information about ndb_mgmd, see Section 6.4, “ndb_mgmd — The MySQL Cluster Management Server Daemon”. For options common to all MySQL Cluster programs, see Section 6.23, “Options Common to MySQL Cluster Programs”.

6.24.4. Program Options for ndb_mgm

Table 6.4. ndb_mgm Option Reference

Format	Description	Introduction	Deprecated	Removed
--try-reconnect=#	Specify number of tries for connecting to ndb_mgmd (0 = infinite)			

For more information about `ndb_mgm`, see Section 6.5, "`ndb_mgm` — The MySQL Cluster Management Client". For options common to all MySQL Cluster programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

6.24.5. Program Options for `ndbd_redo_log_reader`

Table 6.5. `ndbd_redo_log_reader` Option Reference

Format	Description	Introduction	Deprecated	Removed
-nocheck	Do not check records for errors			
-noprint	Do not print records			

For more information about `ndbd_redo_log_reader`, see Section 6.16, "`ndbd_redo_log_reader` — Check and Print Content of Cluster Redo Log". For options common to all MySQL Cluster programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

6.24.6. Program Options for `ndb_config`

Table 6.6. `ndb_config` Option Reference

Format	Description	Introduction	Deprecated	Removed
--configinfo	Dumps information about all NDB configuration parameters in text format with default, maximum, and minimum values. Use with --xml to obtain XML output.	5.1.34-ndb-7.0.6		
--connections	Print connection information only			
--fields=string	Field separator			
--host=name	Specify host			
--mycnf	Read configuration data from my.cnf file			
--nodeid	Get configuration of node with this ID			
--nodes	Print node information only			
	Short form for --ndb-connectstring	5.1.12		
--config-file=path	Set the path to config.ini file			
--query=string	One or more query options (attributes)			
--rows=string	Row separator			
--type=name	Specify node type			
--configinfo --xml	Use with --configinfo to obtain a dump of all NDB configuration parameters in XML format with default, maximum, and minimum values.	5.1.34-ndb-7.0.6		

For more information about `ndb_config`, see Section 6.6, "`ndb_config` — Extract MySQL Cluster Configuration Information". For options common to all MySQL Cluster programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

6.24.7. Program Options for `ndb_error_reporter`

Table 6.7. `ndb_error_reporter` Option Reference

Format	Description	Introduction	Deprecated	Removed
--fs	Include file system data in error report; can use a large amount of disk space			

For more information about `ndb_error_reporter`, see Section 6.12, "`ndb_error_reporter` — NDB Error-Reporting Utility". For options common to all MySQL Cluster programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

6.24.8. Program Options for `ndb_restore`

Table 6.8. `ndb_restore` Option Reference

Format	Description	Introduction	Deprecated	Removed
--append	Append data to a tab-delimited file	5.1.18		
--backup_path=path	Path to backup files directory			
--backupid=#	Restore from the backup with the given ID			
--connect	Same as connectstring			
--restore_data	Restore table data and logs into NDB Cluster using the NDB API			
--dont_ignore_systab_0	Do not ignore system table during restore. Experimental only; not for production use			
--exclude-databases=db-list	List of one or more databases to exclude (includes those not named)	5.1.32-ndb-6.4.3		
--exclude-tables=table-list	List of one or more tables to exclude (includes those not named)	5.1.32-ndb-6.4.3		
--fields-enclosed-by=char	Fields are enclosed with the indicated character	5.1.18		
--fields-optionally-enclosed-by	Fields are optionally enclosed with the indicated character	5.1.18		
--fields-terminated-by=char	Fields are terminated by the indicated character	5.1.18		
--hex	Print binary types in hexadecimal format	5.1.18		
--include-databases=db-list	List of one or more databases to restore (excludes those not named)	5.1.32-ndb-6.4.3		
--include-tables=table-list	List of one or more tables to restore (excludes those not named)	5.1.32-ndb-6.4.3		
--lines-terminated-by=char	Lines are terminated by the indicated character	5.1.18		
--restore_meta	Restore metadata to NDB Cluster using the NDB API			
--ndb-nodegroup-map=map	Nodegroup map for NDBCLUSTER storage engine. Syntax: list of (source_nodegroup, destination_nodegroup)			
--no-binlog	If a mysqld is connected and using binary logging, do not log the restored data	5.1.24-ndb-6.3.16		
--no-restore-disk-objects	Do not restore Disk Data objects such as tablespaces and log file groups			
--no-upgrade	Do not upgrade array type for varsize attributes which do not already resize VAR data, and do not change column attributes	5.1.19		
--nodeid=#	Back up files from node with this ID			
--parallelism=#	Number of parallel transactions during restoration of data			
--preserve-trailing-spaces	Allow preservation of tailing spaces (including padding) when CHAR is promoted to VARCHAR or BINARY is promoted to	5.1.23-ndb-6.3.8		

Format	Description	Introduction	Deprecated	Removed
	VARBINARY			
--print	Print metadata, data and log to stdout (equivalent to --print_meta --print_data --print_log)			
--print_data	Print data to stdout			
--print_log	Print to stdout			
--print_metadata	Print metadata to stdout			
--progress-frequency=#	Print status of restoration each given number of seconds	5.1.?		
--promote-attributes	Allow attributes to be promoted when restoring data from backup	5.1.23-ndb-6.3.8		
--restore_epoch	Restore epoch info into the status table. Convenient on a MySQL Cluster replication slave for starting replication. The row in mysql.ndb_apply_status with id 0 will be updated/inserted.			
--skip-table-check	Skip table structure check during restoring of data	5.1.17		
--tab=path	Creates tab separated a .txt file for each table in the given path	5.1.18		
--verbose=#	Control level of verbosity in output			

For more information about `ndb_restore`, see Section 6.17, "`ndb_restore` — Restore a MySQL Cluster Backup". For options common to all NDB programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

6.24.9. Program Options for `ndb_show_tables`

Table 6.9. `ndb_show_tables` Option Reference

Format	Description	Introduction	Deprecated	Removed
--database=string	Specifies the database in which the table is found			
--loops=#	Number of times to repeat output			
--show-temp-status	Show table temporary flag			
--parsable	Return output suitable for MySQL LOAD DATA INFILE statement			
--type=#	Limit output to objects of this type			
--unqualified	Do not qualify table names			

For more information about `ndb_show_tables`, see Section 6.20, "`ndb_show_tables` — Display List of NDB Tables". For options common to all MySQL Cluster programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

6.24.10. Program Options for `ndb_size.pl`

Table 6.10. `ndb_size_pl` Option Reference

Format	Description	Introduction	Deprecated	Removed
--database=dbname	The databae or databases to examine; accepts a comma-delimited list; the default is ALL (use all databases found on the server)			
--excludedbs=db-list	Skip any databases in a comma-separated list of databases			
--excludetables=tbl-list	Skip any tables in a comma-separated list of tables			
--format=string	Set output format (text or HTML)			
--hostname[:port]	Specify host and optional port as host[:port]			
--loadqueries=file	Loads all queries from the file specified; does not connect to a database			
--password=string	Specify a MySQL user password			

Format	Description	Introduction	Deprecated	Removed
--real_table_name=table	Designates a table to handle unique index size calculations	5.1.22-ndb-6.2.5		
--savequeries=file	Saves all queries to the database into the file specified			
--socket=file	Specify a socket to connect to	5.1.22-ndb-6.2.5		
--user=string	Specify a MySQL user name			

For more information about `ndb_size.pl`, see Section 6.21, "`ndb_size.pl` — NDBCLUSTER Size Requirement Estimator". For options common to all MySQL Cluster programs, see Section 6.23, "Options Common to MySQL Cluster Programs".

Chapter 7. Management of MySQL Cluster

Managing a MySQL Cluster involves a number of tasks, the first of which is to configure and start MySQL Cluster. This is covered in Chapter 3, *MySQL Cluster Configuration*, and Chapter 6, *MySQL Cluster Programs*.

The following sections cover the management of a running MySQL Cluster.

For information about security issues relating to management and deployment of a MySQL Cluster, see Chapter 8, *MySQL Cluster Security Issues*.

There are essentially two methods of actively managing a running MySQL Cluster. The first of these is through the use of commands entered into the management client whereby cluster status can be checked, log levels changed, backups started and stopped, and nodes stopped and started. The second method involves studying the contents of the cluster log `ndb_node_id_cluster.log`; this is usually found in the management server's `DataDir` directory, but this location can be overridden using the `LogDestination` option — see Section 3.4.5, "Defining a MySQL Cluster Management Server", for details. (Recall that *`node_id`* represents the unique identifier of the node whose activity is being logged.) The cluster log contains event reports generated by `ndbd`. It is also possible to send cluster log entries to a Unix system log.

In addition, some aspects of the cluster's operation can be monitored from an SQL node using the `SHOW ENGINE NDB STATUS` statement. See `SHOW ENGINE` Syntax, for more information.

7.1. Summary of MySQL Cluster Start Phases

This section provides a simplified outline of the steps involved when MySQL Cluster data nodes are started. More complete information can be found in MySQL Cluster Start Phases.

These phases are the same as those reported in the output from the *`node_id`* `STATUS` command in the management client. (See Section 7.2, "Commands in the MySQL Cluster Management Client", for more information about this command.)

Start types. There are several different startup types and modes, as shown here:

- **Initial Start.** The cluster starts with a clean file system on all data nodes. This occurs either when the cluster started for the very first time, or when all data nodes are restarted using the `--initial` option.

  > **Note**
  >
  > Disk Data files are not removed when restarting a node using `--initial`.

- **System Restart.** The cluster starts and reads data stored in the data nodes. This occurs when the cluster has been shut down after having been in use, when it is desired for the cluster to resume operations from the point where it left off.
- **Node Restart.** This is the online restart of a cluster node while the cluster itself is running.
- **Initial Node Restart.** This is the same as a node restart, except that the node is reinitialized and started with a clean file system.

Setup and initialization (Phase -1). Prior to startup, each data node (`ndbd` process) must be initialized. Initialization consists of the following steps:

1. Obtain a node ID
2. Fetch configuration data
3. Allocate ports to be used for inter-node communications
4. Allocate memory according to settings obtained from the configuration file

When a data node or SQL node first connects to the management node, it reserves a cluster node ID. To make sure that no other node allocates the same node ID, this ID is retained until the node has managed to connect to the cluster and at least one `ndbd` reports that this node is connected. This retention of the node ID is guarded by the connection between the node in question and `ndb_mgmd`.

Normally, in the event of a problem with the node, the node disconnects from the management server, the socket used for the connection is closed, and the reserved node ID is freed. However, if a node is disconnected abruptly — for example, due to a hardware failure in one of the cluster hosts, or because of network issues — the normal closing of the socket by the operating system may not

take place. In this case, the node ID continues to be reserved and not released until a TCP timeout occurs 10 or so minutes later.

To take care of this problem, you can use `PURGE STALE SESSIONS`. Running this statement forces all reserved node IDs to be checked; any that are not being used by nodes actually connected to the cluster are then freed.

Beginning with MySQL 5.1.11, timeout handling of node ID assignments is implemented. This performs the ID usage checks automatically after approximately 20 seconds, so that `PURGE STALE SESSIONS` should no longer be necessary in a normal Cluster start.

After each data node has been initialized, the cluster startup process can proceed. The stages which the cluster goes through during this process are listed here:

- **Phase 0.** The `NDBFS` and `NDBCNTR` blocks start (see `NDB` Kernel Blocks). The cluster file system is cleared, if the cluster was started with the `--initial` option.
- **Phase 1.** In this stage, all remaining `NDB` kernel blocks are started. Cluster connections are set up, inter-block communications are established, and Cluster heartbeats are started. In the case of a node restart, API node connections are also checked.

  > **Note**
  >
  > When one or more nodes hang in Phase 1 while the remaining node or nodes hang in Phase 2, this often indicates network problems. One possible cause of such issues is one or more cluster hosts having multiple network interfaces. Another common source of problems causing this condition is the blocking of TCP/IP ports needed for communications between cluster nodes. In the latter case, this is often due to a misconfigured firewall.

- **Phase 2.** The `NDBCNTR` kernel block checks the states of all existing nodes. The master node is chosen, and the cluster schema file is initialized.
- **Phase 3.** The `DBLQH` and `DBTC` kernel blocks set up communications between them. The startup type is determined; if this is a restart, the `DBDIH` block obtains permission to perform the restart.
- **Phase 4.** For an initial start or initial node restart, the redo log files are created. The number of these files is equal to `NoOfFragmentLogFiles`.

  For a system restart:

  - Read schema or schemas.
  - Read data from the local checkpoint.
  - Apply all redo information until the latest restorable global checkpoint has been reached.

  For a node restart, find the tail of the redo log.
- **Phase 5.** Most of the database-related portion of a data node start is performed during this phase. For an initial start or system restart, a local checkpoint is executed, followed by a global checkpoint. Periodic checks of memory usage begin during this phase, and any required node takeovers are performed.
- **Phase 6.** In this phase, node groups are defined and set up.
- **Phase 7.** The arbitrator node is selected and begins to function. The next backup ID is set, as is the backup disk write speed. Nodes reaching this start phase are marked as `Started`. It is now possible for API nodes (including SQL nodes) to connect to the cluster. connect.
- **Phase 8.** If this is a system restart, all indexes are rebuilt (by `DBDIH`).
- **Phase 9.** The node internal startup variables are reset.
- **Phase 100 (*OBSOLETE*).** Formerly, it was at this point during a node restart or initial node restart that API nodes could connect to the node and begin to receive events. Currently, this phase is empty.
- **Phase 101.** At this point in a node restart or initial node restart, event delivery is handed over to the node joining the cluster. The newly-joined node takes over responsibility for delivering its primary data to subscribers. This phase is also referred to as *`SUMA` handover phase*.

After this process is completed for an initial start or system restart, transaction handling is enabled. For a node restart or initial node restart, completion of the startup process means that the node may now act as a transaction coordinator.

7.2. Commands in the MySQL Cluster Management Client

In addition to the central configuration file, a cluster may also be controlled through a command-line interface available through the management client `ndb_mgm`. This is the primary administrative interface to a running cluster.

Commands for the event logs are given in Section 7.4, "Event Reports Generated in MySQL Cluster"; commands for creating backups and restoring from them are provided in Section 7.3, "Online Backup of MySQL Cluster".

The management client has the following basic commands. In the listing that follows, *`node_id`* denotes either a database node ID or the keyword `ALL`, which indicates that the command should be applied to all of the cluster's data nodes.

- `HELP`

  Displays information on all available commands.

- `SHOW`

  Displays information on the cluster's status.

  > **Note**
  >
  > In a cluster where multiple management nodes are in use, this command displays information only for data nodes that are actually connected to the current management server.

- *`node_id`* `START`

  Brings online the data node identified by *`node_id`* (or all data nodes).

  `ALL START` works on all data nodes only, and does not affect management nodes.

  > **Important**
  >
  > To use this command to bring a data node online, the data node must have been started using `ndbd --nostart` or `ndbd -n`.

- *`node_id`* `STOP`

  Stops the data or management node identified by *`node_id`*. Note that `ALL STOP` works to stop all data nodes only, and does not affect management nodes.

  A node affected by this command disconnects from the cluster, and its associated `ndbd` or `ndb_mgmd` process terminates.

- *`node_id`* `RESTART [-n] [-i] [-a]`

  Restarts the data node identified by *`node_id`* (or all data nodes).

  Using the `-i` option with `RESTART` causes the data node to perform an initial restart; that is, the node's file system is deleted and recreated. The effect is the same as that obtained from stopping the data node process and then starting it again using `ndbd --initial` from the system shell. Note that backup files and Disk Data files are not removed when this option is used.

  Using the `-n` option causes the data node process to be restarted, but the data node is not actually brought online until the appropriate `START` command is issued. The effect of this option is the same as that obtained from stopping the data node and then starting it again using `ndbd --nostart` or `ndbd -n` from the system shell.

  Using the `-a` causes all current transactions relying on this node to be aborted. No GCP check is done when the node rejoins the cluster.

- *`node_id`* `STATUS`

  Displays status information for the data node identified by *`node_id`* (or for all data nodes).

- *`node_id`* `REPORT` *`report-type`*

  Displays a report of type *`report-type`* for the data node identified by *`node_id`*, or for all data nodes using `ALL`.

Currently, there are two accepted values for *`report-type`*:

- `BackupStatus` provides a status report on a cluster backup in progress
- `MemoryUsage` displays how much data memory and index memory is being used by each data node.

The `REPORT` command was introduced in MySQL Cluster NDB 6.2.3 and MySQL Cluster NDB 6.3.0.

- `ENTER SINGLE USER MODE` *`node_id`*

  Enters single user mode, whereby only the MySQL server identified by the node ID *`node_id`* is allowed to access the database.

  > **Important**
  >
  > Currently, it is not possible in for data nodes to join a MySQL Cluster while it is running in single user mode. (Bug#20395)

- `EXIT SINGLE USER MODE`

  Exits single user mode, allowing all SQL nodes (that is, all running `mysqld` processes) to access the database.

  > **Note**
  >
  > It is possible to use `EXIT SINGLE USER MODE` even when not in single user mode, although the command has no effect in this case.

- `QUIT`, `EXIT`

  Terminates the management client.

  This command does not affect any nodes connected to the cluster.

- `SHUTDOWN`

  Shuts down all cluster data nodes and management nodes. To exit the management client after this has been done, use `EXIT` or `QUIT`.

  This command does *not* shut down any SQL nodes or API nodes that are connected to the cluster.

- `CREATE NODEGROUP` *`nodeid`*`[, nodeid, ...]`

  Creates a new MySQL Cluster node group and causes data nodes to join it.

  This command is used after adding new data nodes online to a MySQL Cluster, and causes them to join a new node group and thus to begin participating fully in the cluster. The command takes as its sole parameter a comma-separated list of node IDs — these are the IDs of the nodes just added and started that are to join the new node group. The number of nodes must be the same as the number of nodes in each node group that is already part of the cluster (each MySQL Cluster node group must have the same number of nodes). In other words, if the MySQL Cluster has 2 node groups of 2 data nodes each, then the new node group must also have 2 data nodes.

  The node group ID of the new node group created by this command is determined automatically, and always the next highest unused node group ID in the cluster; it is not possible to set it manually.

  This command was introduced in MySQL Cluster NDB 6.4.0. For more information, see Section 7.8, “Adding MySQL Cluster Data Nodes Online”.

- `DROP NODEGROUP` *`nodegroup_id`*

  Drops the MySQL Cluster node group with the given *`nodegroup_id`*.

  This command can be used to drop a node group from a MySQL Cluster. `DROP NODEGROUP` takes as its sole argument the node group ID of the node group to be dropped.

  `DROP NODEGROUP` acts only to remove the data nodes in the effected node group from that node group. It does not stop data

nodes, assign them to a different node group, or remove them from the cluster's configuration. A data node that does not belong to a node group is indicated in the output of the management client `SHOW` command with `no nodegroup` in place of the node group ID, like this (indicated using bold text):

```
id=3    @10.100.2.67  (5.1.34-ndb-7.0.7, no nodegroup)
```

Prior to MySQL Cluster NDB 7.0.4, the `SHOW` output was not updated correctly following `DROP NODEGROUP`. (Bug#43413)

`DROP NODEGROUP` works only when all data nodes in the node group to be dropped are completely empty of any table data and table definitions. Since there is currently no way using `ndb_mgm` or in the `mysql` client to remove all data from a specific data node or node group, this means that the command succeeds only in the two following cases:

1. After issuing `CREATE NODEGROUP` in the `ndb_mgm` client, but before issuing any `ALTER ONLINE TABLE ... REORGANIZE PARTITION` statements in the `mysql` client.

2. After dropping all `NDBCLUSTER` tables using `DROP TABLE`.

   `TRUNCATE` does not work for this purpose because this removes only the table data; the data nodes continue to store an `NDBCLUSTER` table's definition until a `DROP TABLE` statement is issued that causes the table metadata to be dropped.

`DROP NODEGROUP` was introduced in MySQL Cluster NDB 6.4.0. For more information, see Section 7.8, “Adding MySQL Cluster Data Nodes Online”.

7.3. Online Backup of MySQL Cluster

This section describes how to create a backup and how to restore the database from a backup at a later time.

7.3.1. MySQL Cluster Backup Concepts

A backup is a snapshot of the database at a given time. The backup consists of three main parts:

- **Metadata.** The names and definitions of all database tables
- **Table records.** The data actually stored in the database tables at the time that the backup was made
- **Transaction log.** A sequential record telling how and when data was stored in the database

Each of these parts is saved on all nodes participating in the backup. During backup, each node saves these three parts into three files on disk:

- `BACKUP-`*`backup_id`*`.`*`node_id`*`.ctl`

  A control file containing control information and metadata. Each node saves the same table definitions (for all tables in the cluster) to its own version of this file.

- `BACKUP-`*`backup_id`*`-0.`*`node_id`*`.data`

  A data file containing the table records, which are saved on a per-fragment basis. That is, different nodes save different fragments during the backup. The file saved by each node starts with a header that states the tables to which the records belong. Following the list of records there is a footer containing a checksum for all records.

- `BACKUP-`*`backup_id`*`.`*`node_id`*`.log`

  A log file containing records of committed transactions. Only transactions on tables stored in the backup are stored in the log. Nodes involved in the backup save different records because different nodes host different database fragments.

In the listing above, *`backup_id`* stands for the backup identifier and *`node_id`* is the unique identifier for the node creating the file.

7.3.2. Using The MySQL Cluster Management Client to Create a Backup

Before starting a backup, make sure that the cluster is properly configured for performing one. (See Section 7.3.3, “Configuration for MySQL Cluster Backups”.)

The START BACKUP command is used to create a backup:

```
START BACKUP [backup_id] [wait_option] [snapshot_option]
wait_option:
WAIT {STARTED | COMPLETED} | NOWAIT
snapshot_option:
SNAPSHOTSTART | SNAPSHOTEND
```

Successive backups are automatically identified sequentially, so the *backup_id*, an integer greater than or equal to 1, is optional; if it is omitted, the next available value is used. If an existing *backup_id* value is used, the backup fails with the error BACKUP FAILED: FILE ALREADY EXISTS. If used, the *backup_id* must follow START BACKUP immediately, before any other options are used.

Prior to MySQL Cluster NDB 6.2.17, 6.3.23, and 6.4.3, backup IDs greater than 2147483648 (2^{31}) were not supported correctly. (Bug#43042) Beginning with these versions, the maximum possible backup ID is 4294967296 (2^{32}).

> **Note**
>
> Prior to MySQL Cluster NDB 7.0.5, when starting a backup using ndb_mgm -e "START BACKUP", the *backup_id* was required. (Bug#31754)

The *wait_option* can be used to determine when control is returned to the management client after a START BACKUP command is issued, as shown in the following list:

- If NOWAIT is specified, the management client displays a prompt immediately, as seen here:

  ```
  ndb_mgm> START BACKUP NOWAIT
  ndb_mgm>
  ```

  In this case, the management client can be used even while it prints progress information from the backup process.

- With WAIT STARTED the management client waits until the backup has started before returning control to the user, as shown here:

  ```
  ndb_mgm> START BACKUP WAIT STARTED
  Waiting for started, this may take several minutes
  Node 2: Backup 3 started from node 1
  ndb_mgm>
  ```

- **WAIT COMPLETED** causes the management client to wait until the backup process is complete before returning control to the user.

WAIT COMPLETED is the default.

Beginning with MySQL Cluster NDB 6.4.0, a *snapshot_option* can be used to determine whether the backup matches the state of the cluster when START BACKUP was issued, or when it was completed. SNAPSHOTSTART causes the backup to match the state of the cluster when the backup began; SNAPSHOTEND causes the backup to reflect the state of the cluster when the backup was finished. SNAPSHOTEND is the default, and matches the behavior found in previous MySQL Cluster releases.

> **Note**
>
> If you use the SNAPSHOTSTART option with START BACKUP, and the CompressedBackup parameter is enabled, only the data and control files are compressed — the log file is not compressed.

If both a *wait_option* and a *snapshot_option* are used, they may be specified in either order. For example, all of the following commands are valid, assuming that there is no existing backup having 4 as its ID:

```
START BACKUP WAIT STARTED SNAPSHOTSTART
START BACKUP SNAPSHOTSTART WAIT STARTED
START BACKUP 4 WAIT COMPLETED SNAPSHOTSTART
START BACKUP SNAPSHOTEND WAIT COMPLETED
START BACKUP 4 NOWAIT SNAPSHOTSTART
```

The procedure for creating a backup consists of the following steps:

1. Start the management client (ndb_mgm), if it not running already.

2. Execute the **START BACKUP** command. This produces several lines of output indicating the progress of the backup, as

shown here:

```
ndb_mgm> START BACKUP
Waiting for completed, this may take several minutes
Node 2: Backup 1 started from node 1
Node 2: Backup 1 started from node 1 completed
 StartGCP: 177 StopGCP: 180
 #Records: 7362 #LogRecords: 0
 Data: 453648 bytes Log: 0 bytes
ndb_mgm>
```

3. When the backup has started the management client displays this message:

   ```
   Backup backup_id started from node node_id
   ```

   backup_id is the unique identifier for this particular backup. This identifier is saved in the cluster log, if it has not been configured otherwise. *node_id* is the identifier of the management server that is coordinating the backup with the data nodes. At this point in the backup process the cluster has received and processed the backup request. It does not mean that the backup has finished. An example of this statement is shown here:

   ```
   Node 2: Backup 1 started from node 1
   ```

4. The management client indicates with a message like this one that the backup has started:

   ```
   Backup backup_id started from node node_id completed
   ```

   As is the case for the notification that the backup has started, *backup_id* is the unique identifier for this particular backup, and *node_id* is the node ID of the management server that is coordinating the backup with the data nodes. This output is accompanied by additional information including relevant global checkpoints, the number of records backed up, and the size of the data, as shown here:

   ```
   Node 2: Backup 1 started from node 1 completed
    StartGCP: 177 StopGCP: 180
    #Records: 7362 #LogRecords: 0
    Data: 453648 bytes Log: 0 bytes
   ```

It is also possible to perform a backup from the system shell by invoking `ndb_mgm` with the `-e` or `--execute` option, as shown in this example:

```
shell> ndb_mgm -e "START BACKUP 6 WAIT COMPLETED SNAPSHOTSTART"
```

When using `START BACKUP` in this way, you must specify the backup ID.

Cluster backups are created by default in the `BACKUP` subdirectory of the `DataDir` on each data node. This can be overridden for one or more data nodes individually, or for all cluster data nodes in the `config.ini` file using the `BackupDataDir` configuration parameter as discussed in Identifying Data Nodes. The backup files created for a backup with a given *backup_id* are stored in a subdirectory named `BACKUP-`*backup_id* in the backup directory.

To abort a backup that is already in progress:

1. Start the management client.
2. Execute this command:

   ```
   ndb_mgm> ABORT BACKUP backup_id
   ```

   The number *backup_id* is the identifier of the backup that was included in the response of the management client when the backup was started (in the message `Backup` *backup_id* `started from node` *management_node_id*).

3. The management client will acknowledge the abort request with `Abort of backup` *backup_id* `ordered`.

   > **Note**
   >
   > At this point, the management client has not yet received a response from the cluster data nodes to this request, and the backup has not yet actually been aborted.

4. After the backup has been aborted, the management client will report this fact in a manner similar to what is shown here:

   ```
   Node 1: Backup 3 started from 5 has been aborted. Error: 1321 - Backup aborted by user request: Permanent error: U
   ```

```
Node 3: Backup 3 started from 5 has been aborted. Error: 1323 - 1323: Permanent error: Internal error
Node 2: Backup 3 started from 5 has been aborted. Error: 1323 - 1323: Permanent error: Internal error
Node 4: Backup 3 started from 5 has been aborted. Error: 1323 - 1323: Permanent error: Internal error
```

In this example, we have shown sample output for a cluster with 4 data nodes, where the sequence number of the backup to be aborted is 3, and the management node to which the cluster management client is connected has the node ID 5. The first node to complete its part in aborting the backup reports that the reason for the abort was due to a request by the user. (The remaining nodes report that the backup was aborted due to an unspecified internal error.)

> **Note**
>
> There is no guarantee that the cluster nodes respond to an `ABORT BACKUP` command in any particular order.

The `Backup backup_id started from node management_node_id has been aborted` messages mean that the backup has been terminated and that all files relating to this backup have been removed from the cluster file system.

It is also possible to abort a backup in progress from a system shell using this command:

```
shell> ndb_mgm -e "ABORT BACKUP backup_id"
```

> **Note**
>
> If there is no backup having the ID *`backup_id`* running when an `ABORT BACKUP` is issued, the management client makes no response, nor is it indicated in the cluster log that an invalid abort command was sent.

7.3.3. Configuration for MySQL Cluster Backups

Five configuration parameters are essential for backup:

- `BackupDataBufferSize`

  The amount of memory used to buffer data before it is written to disk.

- `BackupLogBufferSize`

  The amount of memory used to buffer log records before these are written to disk.

- `BackupMemory`

  The total memory allocated in a database node for backups. This should be the sum of the memory allocated for the backup data buffer and the backup log buffer.

- `BackupWriteSize`

  The default size of blocks written to disk. This applies for both the backup data buffer and the backup log buffer.

- `BackupMaxWriteSize`

  The maximum size of blocks written to disk. This applies for both the backup data buffer and the backup log buffer.

More detailed information about these parameters can be found in Backup Parameters.

7.3.4. MySQL Cluster Backup Troubleshooting

If an error code is returned when issuing a backup request, the most likely cause is insufficient memory or disk space. You should check that there is enough memory allocated for the backup.

> **Important**
>
> If you have set `BackupDataBufferSize` and `BackupLogBufferSize` and their sum is greater than 4MB, then you must also set `BackupMemory` as well. See `BackupMemory`.

You should also make sure that there is sufficient space on the hard drive partition of the backup target.

NDB does not support repeatable reads, which can cause problems with the restoration process. Although the backup process is "hot", restoring a MySQL Cluster from backup is not a 100% "hot" process. This is due to the fact that, for the duration of the restore process, running transactions get non-repeatable reads from the restored data. This means that the state of the data is inconsistent while the restore is in progress.

> **MySQL Enterprise**
> MySQL Enterprise subscribers will find more information about Cluster backup in the Knowledge Base article, How Do I Backup my Cluster Database. Access to the MySQL Knowledge Base collection of articles is one of the advantages of subscribing to MySQL Enterprise. For more information, see http://www.mysql.com/products/enterprise/advisors.html.

7.4. Event Reports Generated in MySQL Cluster

In this section, we discuss the types of event logs provided by MySQL Cluster, and the types of events that are logged.

MySQL Cluster provides two types of event log:

- The *cluster log*, which includes events generated by all cluster nodes. The cluster log is the log recommended for most uses because it provides logging information for an entire cluster in a single location.

  By default, the cluster log is saved to a file named `ndb_node_id_cluster.log`, (where *`node_id`* is the node ID of the management server) in the same directory where the `ndb_mgm` binary resides.

  Cluster logging information can also be sent to `stdout` or a `syslog` facility in addition to or instead of being saved to a file, as determined by the values set for the `DataDir` and `LogDestination` configuration parameters. See Section 3.4.5, "Defining a MySQL Cluster Management Server", for more information about these parameters.

- *Node logs* are local to each node.

  Output generated by node event logging is written to the file `ndb_node_id_out.log` (where *`node_id`* is the node's node ID) in the node's `DataDir`. Node event logs are generated for both management nodes and data nodes.

  Node logs are intended to be used only during application development, or for debugging application code.

Both types of event logs can be set to log different subsets of events.

Each reportable event can be distinguished according to three different criteria:

- *Category*: This can be any one of the following values: `STARTUP`, `SHUTDOWN`, `STATISTICS`, `CHECKPOINT`, `NODERESTART`, `CONNECTION`, `ERROR`, or `INFO`.
- *Priority*: This is represented by one of the numbers from 1 to 15 inclusive, where 1 indicates "most important" and 15 "least important."
- *Severity Level*: This can be any one of the following values: `ALERT`, `CRITICAL`, `ERROR`, `WARNING`, `INFO`, or `DEBUG`.

Both the cluster log and the node log can be filtered on these properties.

The format used in the cluster log is as shown here:

```
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 1: Data usage is 2%(60 32K pages of total 2560)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 1: Index usage is 1%(24 8K pages of total 2336)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 1: Resource 0 min: 0 max: 639 curr: 0
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 2: Data usage is 2%(76 32K pages of total 2560)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 2: Index usage is 1%(24 8K pages of total 2336)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 2: Resource 0 min: 0 max: 639 curr: 0
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 3: Data usage is 2%(58 32K pages of total 2560)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 3: Index usage is 1%(25 8K pages of total 2336)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 3: Resource 0 min: 0 max: 639 curr: 0
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 4: Data usage is 2%(74 32K pages of total 2560)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 4: Index usage is 1%(25 8K pages of total 2336)
2007-01-26 19:35:55 [MgmSrvr] INFO     -- Node 4: Resource 0 min: 0 max: 639 curr: 0
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 4: Node 9 Connected
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 1: Node 9 Connected
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 1: Node 9: API version 5.1.15
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 2: Node 9 Connected
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 2: Node 9: API version 5.1.15
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 3: Node 9 Connected
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 3: Node 9: API version 5.1.15
2007-01-26 19:39:42 [MgmSrvr] INFO     -- Node 4: Node 9: API version 5.1.15
```

```
2007-01-26 19:59:22 [MgmSrvr] ALERT    -- Node 2: Node 7 Disconnected
2007-01-26 19:59:22 [MgmSrvr] ALERT    -- Node 2: Node 7 Disconnected
```

Each line in the cluster log contains the following information:

- A timestamp in *YYYY-MM-DD HH:MM:SS* format.
- The type of node which is performing the logging. In the cluster log, this is always [MgmSrvr].
- The severity of the event.
- The ID of the node reporting the event.
- A description of the event. The most common types of events to appear in the log are connections and disconnections between different nodes in the cluster, and when checkpoints occur. In some cases, the description may contain status information.

7.4.1. MySQL Cluster Logging Management Commands

The following management commands are related to the cluster log:

- CLUSTERLOG ON

  Turns the cluster log on.

- CLUSTERLOG OFF

  Turns the cluster log off.

- CLUSTERLOG INFO

  Provides information about cluster log settings.

- *node_id* CLUSTERLOG *category=threshold*

  Logs *category* events with priority less than or equal to *threshold* in the cluster log.

- CLUSTERLOG FILTER *severity_level*

  Toggles cluster logging of events of the specified *severity_level*.

The following table describes the default setting (for all data nodes) of the cluster log category threshold. If an event has a priority with a value lower than or equal to the priority threshold, it is reported in the cluster log.

Note that events are reported per data node, and that the threshold can be set to different values on different nodes.

Category	Default threshold (All data nodes)
STARTUP	7
SHUTDOWN	7
STATISTICS	7
CHECKPOINT	7
NODERESTART	7
CONNECTION	7
ERROR	15
INFO	7

The STATISTICS category can provide a great deal of useful data. See Section 7.4.3, “Using CLUSTERLOG STATISTICS in the MySQL Cluster Management Client”, for more information.

Thresholds are used to filter events within each category. For example, a STARTUP event with a priority of 3 is not logged unless the threshold for STARTUP is set to 3 or higher. Only events with priority 3 or lower are sent if the threshold is 3.

The following table shows the event severity levels.

> **Note**
>
> These correspond to Unix `syslog` levels, except for `LOG_EMERG` and `LOG_NOTICE`, which are not used or mapped.

1	`ALERT`	A condition that should be corrected immediately, such as a corrupted system database
2	`CRITICAL`	Critical conditions, such as device errors or insufficient resources
3	`ERROR`	Conditions that should be corrected, such as configuration errors
4	`WARNING`	Conditions that are not errors, but that might require special handling
5	`INFO`	Informational messages
6	`DEBUG`	Debugging messages used for `NDBCLUSTER` development

Event severity levels can be turned on or off (using `CLUSTERLOG FILTER` — see above). If a severity level is turned on, then all events with a priority less than or equal to the category thresholds are logged. If the severity level is turned off then no events belonging to that severity level are logged.

> **Important**
>
> Cluster log levels are set on a per `ndb_mgmd`, per subscriber basis. This means that, in a MySQL Cluster with multiple management servers, using a `CLUSTERLOG` command in an instance of `ndb_mgm` connected to one management server affects only logs generated by that management server but not by any of the others. This also means that, should one of the management servers be restarted, only logs generated by that management server are affected by the resetting of log levels caused by the restart.

7.4.2. MySQL Cluster Log Events

An event report reported in the event logs has the following format:

```
datetime [string] severity -- message
```

For example:

```
09:19:30 2005-07-24 [NDB] INFO -- Node 4 Start phase 4 completed
```

This section discusses all reportable events, ordered by category and severity level within each category.

In the event descriptions, GCP and LCP mean “Global Checkpoint” and “Local Checkpoint”, respectively.

`CONNECTION` Events

These events are associated with connections between Cluster nodes.

Event	Priority	Severity Level	Description
data nodes connected	8	`INFO`	Data nodes connected
data nodes disconnected	8	`INFO`	Data nodes disconnected
Communication closed	8	`INFO`	SQL node or data node connection closed
Communication opened	8	`INFO`	SQL node or data node connection opened

`CHECKPOINT` Events

The logging messages shown here are associated with checkpoints.

Event	Priority	Severity Level	Description
LCP stopped in calc keep GCI	0	`ALERT`	LCP stopped
Local checkpoint fragment completed	11	`INFO`	LCP on a fragment has been completed
Global checkpoint completed	10	`INFO`	GCP finished
Global checkpoint started	9	`INFO`	Start of GCP: REDO log is written to disk
Local checkpoint completed	8	`INFO`	LCP completed normally

Local checkpoint started	7	INFO	Start of LCP: data written to disk

STARTUP Events

The following events are generated in response to the startup of a node or of the cluster and of its success or failure. They also provide information relating to the progress of the startup process, including information concerning logging activities.

Event	Priority	Severity Level	Description
Internal start signal received STTORRY	15	INFO	Blocks received after completion of restart
New REDO log started	10	INFO	GCI keep *X*, newest restorable GCI *Y*
New log started	10	INFO	Log part *X*, start MB *Y*, stop MB *Z*
Node has been refused for inclusion in the cluster	8	INFO	Node cannot be included in cluster due to misconfiguration, inability to establish communication, or other problem
data node neighbors	8	INFO	Shows neighboring data nodes
data node start phase *X* completed	4	INFO	A data node start phase has been completed
Node has been successfully included into the cluster	3	INFO	Displays the node, managing node, and dynamic ID
data node start phases initiated	1	INFO	NDB Cluster nodes starting
data node all start phases completed	1	INFO	NDB Cluster nodes started
data node shutdown initiated	1	INFO	Shutdown of data node has commenced
data node shutdown aborted	1	INFO	Unable to shut down data node normally

NODERESTART Events

The following events are generated when restarting a node and relate to the success or failure of the node restart process.

Event	Priority	Severity Level	Description
Node failure phase completed	8	ALERT	Reports completion of node failure phases
Node has failed, node state was *X*	8	ALERT	Reports that a node has failed
Report arbitrator results	2	ALERT	There are eight different possible results for arbitration attempts: • Arbitration check failed — less than 1/2 nodes left • Arbitration check succeeded — node group majority • Arbitration check failed — missing node group • Network partitioning — arbitration required • Arbitration succeeded — affirmative response from node *X* • Arbitration failed - negative response from node *X* • Network partitioning - no arbitrator available • Network partitioning - no arbitrator configured
Completed copying a fragment	10	INFO	
Completed copying of dictionary information	8	INFO	
Completed copying distribution information	8	INFO	
Starting to copy fragments	8	INFO	
Completed copying all fragments	8	INFO	

GCP takeover started	7	INFO	
GCP takeover completed	7	INFO	
LCP takeover started	7	INFO	
LCP takeover completed (state = *X*)	7	INFO	
Report whether an arbitrator is found or not	6	INFO	There are seven different possible outcomes when seeking an arbitrator: • Management server restarts arbitration thread [state=*X*] • Prepare arbitrator node *X* [ticket=*Y*] • Receive arbitrator node *X* [ticket=*Y*] • Started arbitrator node *X* [ticket=*Y*] • Lost arbitrator node *X* - process failure [state=*Y*] • Lost arbitrator node *X* - process exit [state=*Y*] • Lost arbitrator node *X* <error msg> [state=*Y*]

STATISTICS Events

The following events are of a statistical nature. They provide information such as numbers of transactions and other operations, amount of data sent or received by individual nodes, and memory usage.

Event	Priority	Severity Level	Description
Report job scheduling statistics	9	INFO	Mean internal job scheduling statistics
Sent number of bytes	9	INFO	Mean number of bytes sent to node *X*
Received # of bytes	9	INFO	Mean number of bytes received from node *X*
Report transaction statistics	8	INFO	Numbers of: transactions, commits, reads, simple reads, writes, concurrent operations, attribute information, and aborts
Report operations	8	INFO	Number of operations
Report table create	7	INFO	
Memory usage	5	INFO	Data and index memory usage (80%, 90%, and 100%)

ERROR Events

These events relate to Cluster errors and warnings. The presence of one or more of these generally indicates that a major malfunction or failure has occurred.

Event	Priority	Severity	Description
Dead due to missed heartbeat	8	ALERT	Node *X* declared “dead” due to missed heartbeat
Transporter errors	2	ERROR	
Transporter warnings	8	WARNING	
Missed heartbeats	8	WARNING	Node *X* missed heartbeat #*Y*
General warning events	2	WARNING	

INFO Events

These events provide general information about the state of the cluster and activities associated with Cluster maintenance, such as logging and heartbeat transmission.

Event	Priority	Severity	Description

Sent heartbeat	12	INFO	Heartbeat sent to node *X*
Create log bytes	11	INFO	Log part, log file, MB
General information events	2	INFO	

7.4.3. Using CLUSTERLOG STATISTICS in the MySQL Cluster Management Client

The NDB management client's CLUSTERLOG STATISTICS command can provide a number of useful statistics in its output. Counters providing information about the state of the cluster are updated at 5-second reporting intervals by the transaction coordinator (TC) and the local query handler (LQH), and written to the cluster log.

Transaction coordinator statistics. Each transaction has one transaction coordinator, which is chosen by one of the following methods:

- In a round-robin fashion
- By communication proximity
- (*Beginning with MySQL Cluster NDB 6.3.4*:) By supplying a data placement hint when the transaction is started

> **Note**
>
> You can determine which TC selection method is used for transactions started from a given SQL node using the ndb_optimized_node_selection system variable.

All operations within the same transaction use the same transaction coordinator, which reports the following statistics:

- **Trans count.** This is the number transactions started in the last interval using this TC as the transaction coordinator. Any of these transactions may have committed, have been aborted, or remain uncommitted at the end of the reporting interval.

  > **Note**
  >
  > Transactions do not migrate between TCs.

- **Commit count.** This is the number of transactions using this TC as the transaction coordinator that were committed in the last reporting interval. Because some transactions committed in this reporting interval may have started in a previous reporting interval, it is possible for Commit count to be greater than Trans count.
- **Read count.** This is the number of primary key read operations using this TC as the transaction coordinator that were started in the last reporting interval, including simple reads. This count also includes reads performed as part of unique index operations. A unique index read operation generates 2 primary key read operations — 1 for the hidden unique index table, and 1 for the table on which the read takes place.
- **Simple read count.** This is the number of simple read operations using this TC as the transaction coordinator that were started in the last reporting interval. This is a subset of Read count. Because the value of Simple read count is incremented at a different point in time from Read count, it can lag behind Read count slightly, so it is conceivable that Simple read count is not equal to Read count for a given reporting interval, even if all reads made during that time were in fact simple reads.
- **Write count.** This is the number of primary key write operations using this TC as the transaction coordinator that were started in the last reporting interval. This includes all inserts, updates, writes and deletes, as well as writes performed as part of unique index operations.

  > **Note**
  >
  > A unique index update operation can generate multiple PK read and write operations on the index table and on the base table.

- **AttrInfoCount.** This is the number of 32-bit data words received in the last reporting interval for primary key operations using this TC as the transaction coordinator. For reads, this is proportional to the number of columns requested. For inserts and updates, this is proportional to the number of columns written, and the size of their data. For delete operations, this is usually zero. Unique index operations generate multiple PK operations and so increase this count. However, data words sent to describe the PK operation itself, and the key information sent, are *not* counted here. Attribute information sent to describe columns to read for scans, or to describe ScanFilters, is also not counted in AttrInfoCount.

- **`Concurrent Operations`.** This is the number of primary key or scan operations using this TC as the transaction coordinator that were started during the last reporting interval but that were not completed. Operations increment this counter when they are started and decrement it when they are completed; this occurs after the transaction commits. Dirty reads and writes — as well as failed operations — decrement this counter. The maximum value that `Concurrent Operations` can have is the maximum number of operations that a TC block can support; currently, this is `(2 * MaxNoOfConcurrentOperations) + 16 + MaxNoOfConcurrentTransactions`. (For more information about these configuration parameters, see the *Transaction Parameters* section of Section 3.4.6, "Defining MySQL Cluster Data Nodes".)

- **`Abort count`.** This is the number of transactions using this TC as the transaction coordinator that were aborted during the last reporting interval. Because some transactions that were aborted in the last reporting interval may have started in a previous reporting interval, `Abort count` can sometimes be greater than `Trans count`.

- **`Scans`.** This is the number of table scans using this TC as the transaction coordinator that were started during the last reporting interval. This does not include range scans (that is, ordered index scans).

- **`Range scans`.** This is the number of ordered index scans using this TC as the transaction coordinator that were started in the last reporting interval.

Local query handler statistics (`Operations`). There is 1 cluster event per local query handler block (that is, 1 per data node process). Operations are recorded in the LQH where the data they are operating on resides.

> **Note**
>
> A single transaction may operate on data stored in multiple LQH blocks.

The `Operations` statistic provides the number of local operations performed by this LQH block in the last reporting interval, and includes all types of read and write operations (insert, update, write, and delete operations). This also includes operations used to replicate writes — for example, in a 2-replica cluster, the write to the primary replica is recorded in the primary LQH, and the write to the backup will be recorded in the backup LQH. Unique key operations may result in multiple local operations; however, this does *not* include local operations generated as a result of a table scan or ordered index scan, which are not counted.

Process scheduler statistics. In addition to the statistics reported by the transaction coordinator and local query handler, each `ndbd` process has a scheduler which also provides useful metrics relating to the performance of a MySQL Cluster. This scheduler runs in an infinite loop; during each loop the scheduler performs the following tasks:

1. Read any incoming messages from sockets into a job buffer.
2. Check whether there are any timed messages to be executed; if so, put these into the job buffer as well.
3. Execute (in a loop) any messages in the job buffer.
4. Send any distributed messages that were generated by executing the messages in the job buffer.
5. Wait for any new incoming messages.

Process scheduler statistics include the following:

- **`Mean Loop Counter`.** This is the number of loops executed in the third step from the preceding list. This statistic increases in size as the utilization of the TCP/IP buffer improves. You can use this to monitor changes in performance as you add new data node processes.

- **`Mean send size` and `Mean receive size`.** These statistics allow you to gauge the efficiency of, respectively writes and reads between nodes. The values are given in bytes. Higher values mean a lower cost per byte sent or received; the maximum value is 64K.

To cause all cluster log statistics to be logged, you can use the following command in the `NDB` management client:

```
ndb_mgm> ALL CLUSTERLOG STATISTICS=15
```

> **Note**
>
> Setting the threshold for `STATISTICS` to 15 causes the cluster log to become very verbose, and to grow quite rapidly in size, in direct proportion to the number of cluster nodes and the amount of activity in the MySQL Cluster.

For more information about MySQL Cluster management client commands relating to logging and reporting, see Section 7.4.1, "MySQL Cluster Logging Management Commands".

7.5. MySQL Cluster Log Messages

This section contains information about the messages written to the cluster log in response to different cluster log events. It provides additional, more specific information on NDB transporter errors.

7.5.1. MySQL Cluster — Messages in the Cluster Log

The following table lists the most common NDB cluster log messages. For information about the cluster log, log events, and event types, see Section 7.4, "Event Reports Generated in MySQL Cluster". These log messages also correspond to log event types in the MGM API; see The Ndb_logevent_type Type, for related information of interest to Cluster API developers.

Log Message. Node *mgm_node_id*: Node *data_node_id* Connected **Description.** The data node having node ID *node_id* has connected to the management server (node *mgm_node_id*).	**Event Name.** Connected **Event Type.** Connection **Priority.** 8 **Severity.** INFO
Log Message. Node *mgm_node_id*: Node *data_node_id* Disconnected **Description.** The data node having node ID *data_node_id* has disconnected from the management server (node *mgm_node_id*).	**Event Name.** Disconnected **Event Type.** Connection **Priority.** 8 **Severity.** ALERT
Log Message. Node *data_node_id*: Communication to Node *api_node_id* closed **Description.** The API node or SQL node having node ID *api_node_id* is no longer communicating with data node *data_node_id*.	**Event Name.** CommunicationClosed **Event Type.** Connection **Priority.** 8 **Severity.** INFO
Log Message. Node *data_node_id*: Communication to Node *api_node_id* opened **Description.** The API node or SQL node having node ID *api_node_id* is now communicating with data node *data_node_id*.	**Event Name.** CommunicationOpened **Event Type.** Connection **Priority.** 8 **Severity.** INFO
Log Message. Node *mgm_node_id*: Node *api_node_id*: API *version* **Description.** The API node having node ID *api_node_id* has connected to management node *mgm_node_id* using NDB API version *version* (generally the same as the MySQL version number).	**Event Name.** ConnectedApiVersion **Event Type.** Connection **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Global checkpoint *gci* started **Description.** A global checkpoint with the ID *gci* has been started; node *node_id* is the master responsible for this global checkpoint.	**Event Name.** GlobalCheckpointStarted **Event Type.** Checkpoint **Priority.** 9 **Severity.** INFO
Log Message. Node *node_id*: Global checkpoint *gci* completed **Description.** The global checkpoint having the ID *gci* has been completed; node *node_id* was the master responsible for this global checkpoint.	**Event Name.** GlobalCheckpointCompleted **Event Type.** Checkpoint **Priority.** 10 **Severity.** INFO
Log Message. Node *node_id*: Local checkpoint *lcp* star-	**Event Name.** LocalCheckpointStarted

ted. Keep GCI = *current_gci* oldest restorable GCI = *old_gci* **Description.** The local checkpoint having sequence ID *lcp* has been started on node *node_id*. The most recent GCI that can be used has the index *current_gci*, and the oldest GCI from which the cluster can be restored has the index *old_gci*.	**Event Type.** Checkpoint **Priority.** 7 **Severity.** INFO
Log Message. Node *node_id*: Local checkpoint *lcp* completed **Description.** The local checkpoint having sequence ID *lcp* on node *node_id* has been completed.	**Event Name.** LocalCheckpointCompleted **Event Type.** Checkpoint **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Local Checkpoint stopped in CALCULATED_KEEP_GCI **Description.** The node was unable to determine the most recent usable GCI.	**Event Name.** LCPStoppedInCalcKeepGci **Event Type.** Checkpoint **Priority.** 0 **Severity.** ALERT
Log Message. Node *node_id*: Table ID = *table_id*, fragment *ID = fragment_id* has completed LCP on Node *node_id* maxGciStarted: *started_gci* maxGciCompleted: *completed_gci* **Description.** A table fragment has been checkpointed to disk on node *node_id*. The GCI in progress has the index *started_gci*, and the most recent GCI to have been completed has the index *completed_gci*.	**Event Name.** LCPFragmentCompleted **Event Type.** Checkpoint **Priority.** 11 **Severity.** INFO
Log Message. Node *node_id*: ACC Blocked *num_1* and TUP Blocked *num_2* times last second **Description.** Undo logging is blocked because the log buffer is close to overflowing.	**Event Name.** UndoLogBlocked **Event Type.** Checkpoint **Priority.** 7 **Severity.** INFO
Log Message. Node *node_id*: Start initiated *version* **Description.** Data node *node_id*, running NDB version *version*, is beginning its startup process.	**Event Name.** NDBStartStarted **Event Type.** StartUp **Priority.** 1 **Severity.** INFO
Log Message. Node *node_id*: Started *version* **Description.** Data node *node_id*, running NDB version *version*, has started successfully.	**Event Name.** NDBStartCompleted **Event Type.** StartUp **Priority.** 1 **Severity.** INFO
Log Message. Node *node_id*: STTORRY received after restart finished **Description.** The node has received a signal indicating that a cluster restart has completed.	**Event Name.** STTORRYRecieved **Event Type.** StartUp **Priority.** 15 **Severity.** INFO
Log Message. Node *node_id*: Start phase *phase* completed (*type*)	**Event Name.** StartPhaseCompleted **Event Type.** StartUp

Description. The node has completed start phase *phase* of a *type* start. For a listing of start phases, see Section 7.1, "Summary of MySQL Cluster Start Phases". (*type* is one of initial, system, node, initial node, or <Unknown>.)	**Priority.** 4 **Severity.** INFO
Log Message. Node *node_id*: CM_REGCONF president = *president_id*, own Node = *own_id*, our dynamic id = *dynamic_id* **Description.** Node *president_id* has been selected as "president". *own_id* and *dynamic_id* should always be the same as the ID (*node_id*) of the reporting node.	**Event Name.** CM_REGCONF **Event Type.** StartUp **Priority.** 3 **Severity.** INFO
Log Message. Node *node_id*: CM_REGREF from Node *president_id* to our Node *node_id*. Cause = *cause* **Description.** The reporting node (ID *node_id*) was unable to accept node *president_id* as president. The *cause* of the problem is given as one of Busy, Election with wait = false, Not president, Election without selecting new candidate, or No such cause.	**Event Name.** CM_REGREF **Event Type.** StartUp **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: We are Node *own_id* with dynamic ID *dynamic_id*, our left neighbour is Node *id_1*, our right is Node *id_2* **Description.** The node has discovered its neighboring nodes in the cluster (node *id_1* and node *id_2*). *node_id*, *own_id*, and *dynamic_id* should always be the same; if they are not, this indicates a serious misconfiguration of the cluster nodes.	**Event Name.** FIND_NEIGHBOURS **Event Type.** StartUp **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: *type* shutdown initiated **Description.** The node has received a shutdown signal. The *type* of shutdown is either Cluster or Node.	**Event Name.** NDBStopStarted **Event Type.** StartUp **Priority.** 1 **Severity.** INFO
Log Message. Node *node_id*: Node shutdown completed [, *action*] [Initiated by signal *signal*.] **Description.** The node has been shut down. This report may include an *action*, which if present is one of restarting, no start, or initial. The report may also include a reference to an NDB Protocol *signal*; for possible signals, refer to Operations and Signals.	**Event Name.** NDBStopCompleted **Event Type.** StartUp **Priority.** 1 **Severity.** INFO
Log Message. Node *node_id*: Forced node shutdown completed [, action]. [Occured during startphase *start_phase*.][Initiated by *signal*.][Caused by error *error_code*: '*error_message*(*error_classification*).*error_status*'. [(extra info *extra_code*)]] **Description.** The node has been forcibly shut down. The *action* (one of restarting, no start, or initial) subsequently being taken, if any, is also reported. If the shutdown occurred while the node was starting, the report includes the *start_phase* during which the node failed. If this was a result of a *signal* sent to the node, this information is also provided (see Operations and Signals, for more information). If the error causing the failure is known, this is also included; for more information about NDB error messages and classifications, see MySQL Cluster API Errors.	**Event Name.** NDBStopForced **Event Type.** StartUp **Priority.** 1 **Severity.** ALERT
Log Message. Node *node_id*: Node shutdown aborted **Description.** The node shutdown process was aborted by the user.	**Event Name.** NDBStopAborted **Event Type.** StartUp **Priority.** 1

	Severity. INFO
Log Message. Node *node_id*: StartLog: [GCI Keep: *keep_pos* LastCompleted: *last_pos* NewestRestorable: *restore_pos*] **Description.** This reports global checkpoints referenced during a node start. The redo log prior to *keep_pos* is dropped. *last_pos* is the last global checkpoint in which data node the participated; *restore_pos* is the global checkpoint which is actually used to restore all data nodes.	**Event Name.** StartREDOLog **Event Type.** StartUp **Priority.** 4 **Severity.** INFO
Log Message. *startup_message* [*Listed separately; see below.*] **Description.** There are a number of possible startup messages that can be logged under different circumstances.	**Event Name.** StartReport **Event Type.** StartUp **Priority.** 4 **Severity.** INFO
Log Message. Node *node_id*: Node restart completed copy of dictionary information **Description.** Copying of data dictionary information to the restarted node has been completed.	**Event Name.** NR_CopyDict **Event Type.** NodeRestart **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Node restart completed copy of distribution information **Description.** Copying of data distribution information to the restarted node has been completed.	**Event Name.** NR_CopyDistr **Event Type.** NodeRestart **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Node restart starting to copy the fragments to Node *node_id* **Description.** Copy of fragments to starting data node *node_id* has begun	**Event Name.** NR_CopyFragsStarted **Event Type.** NodeRestart **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Table ID = *table_id*, fragment ID = *fragment_id* have been copied to Node *node_id* **Description.** Fragment *fragment_id* from table *table_id* has been copied to data node *node_id*	**Event Name.** NR_CopyFragDone **Event Type.** NodeRestart **Priority.** 10 **Severity.** INFO
Log Message. Node *node_id*: Node restart completed copying the fragments to Node *node_id* **Description.** Copying of all table fragments to restarting data node *node_id* has been completed	**Event Name.** NR_CopyFragsCompleted **Event Type.** NodeRestart **Priority.** 8 **Severity.** INFO
Log Message. Any of the following: 1. Node *node_id*: Node *node1_id* completed failure of Node *node2_id* 2. All nodes completed failure of Node *node_id*	**Event Name.** NodeFailCompleted **Event Type.** NodeRestart **Priority.** 8 **Severity.** ALERT

3. `Node failure of node_idblock completed` **Description.** One of the following (each corresponding to the same-numbered message listed above): 1. Data node *`node1_id`* has detected the failure of data node *`node2_id`* 2. All (remaining) data nodes have detected the failure of data node *`node_id`* 3. The failure of data node *`node_id`* has been detected in the *`block`*NDB kernel block, where block is 1 of DBTC, DBDICT, DBDIH, or DBLQH; for more information, see NDB Kernel Blocks	
Log Message. `Node mgm_node_id: Node data_node_id has failed. The Node state at failure was state_code` **Description.** A data node has failed. Its state at the time of failure is described by an arbitration state code *`state_code`*: possible state code values can be found in the file `include/kernel/signaldata/ArbitSignalData.hpp`.	**Event Name.** `NODE_FAILREP` **Event Type.** `NodeRestart` **Priority.** 8 **Severity.** `ALERT`
Log Message. `President restarts arbitration thread [state=state_code]` or `Prepare arbitrator node node_id [ticket=ticket_id]` or `Receive arbitrator node node_id [ticket=ticket_id]` or `Started arbitrator node node_id [ticket=ticket_id]` or `Lost arbitrator node node_id - process failure [state=state_code]` or `Lost arbitrator node node_id - process exit [state=state_code]` or `Lost arbitrator node node_id - error_message [state=state_code]` **Description.** This is a report on the current state and progress of arbitration in the cluster. *`node_id`* is the node ID of the management node or SQL node selected as the arbitrator. *`state_code`* is an arbitration state code, as found in `include/kernel/signaldata/ArbitSignalData.hpp`. When an error has occurred, an *`error_message`*, also defined in `ArbitSignalData.hpp`, is provided. *`ticket_id`* is a unique identifier handed out by the arbitrator when it is selected to all the nodes that participated in its selection; this is used to insure that each node requesting arbitration was one of the nodes that took part in the selection process.	**Event Name.** `ArbitState` **Event Type.** `NodeRestart` **Priority.** 6 **Severity.** `INFO`
Log Message. `Arbitration check lost - less than 1/2 nodes left` or `Arbitration check won - all node groups and more than 1/2 nodes left` or `Arbitration check won - node group majority` or `Arbitration check lost - missing node group` or `Network partitioning - arbitration required` or `Arbitration won - positive reply from node node_id` or `Arbitration lost - negative reply from node node_id` or `Network partitioning - no arbitrator available` or `Network partitioning - no arbitrator configured` or `Arbitration failure - error_message [state=state_code]` **Description.** This message reports on the result of arbitration. In the event of arbitration failure, an *`error_message`* and an arbitration *`state_code`* are provided; definitions for both of these are found in `include/kernel/signaldata/ArbitSignalData.hpp`.	**Event Name.** `ArbitResult` **Event Type.** `NodeRestart` **Priority.** 2 **Severity.** `ALERT`
Log Message. `Node node_id: GCP Take over started` **Description.** This node is attempting to assume responsibility for the next global checkpoint (that is, it is becoming the master node)	**Event Name.** `GCP_TakeoverStarted` **Event Type.** `NodeRestart` **Priority.** 7

	Severity. INFO
Log Message. Node *node_id*: GCP Take over completed **Description.** This node has become the master, and has assumed responsibility for the next global checkpoint	**Event Name.** GCP_TakeoverCompleted **Event Type.** NodeRestart **Priority.** 7 **Severity.** INFO
Log Message. Node *node_id*: LCP Take over started **Description.** This node is attempting to assume responsibility for the next set of local checkpoints (that is, it is becoming the master node)	**Event Name.** LCP_TakeoverStarted **Event Type.** NodeRestart **Priority.** 7 **Severity.** INFO
Log Message. Node *node_id*: LCP Take over completed **Description.** This node has become the master, and has assumed responsibility for the next set of local checkpoints	**Event Name.** LCP_TakeoverCompleted **Event Type.** NodeRestart **Priority.** 7 **Severity.** INFO
Log Message. Node *node_id*: Trans. Count = *transactions*, Commit Count = *commits*, Read Count = *reads*, Simple Read Count = *simple_reads*, Write Count = *writes*, AttrInfo Count = *AttrInfo_objects*, Concurrent Operations = *concurrent_operations*, Abort Count = *aborts*, Scans = *scans*, Range scans = *range_scans* **Description.** This report of transaction activity is given approximately once every 10 seconds	**Event Name.** TransReportCounters **Event Type.** Statistic **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Operations=*operations* **Description.** Number of operations performed by this node, provided approximately once every 10 seconds	**Event Name.** OperationReportCounters **Event Type.** Statistic **Priority.** 8 **Severity.** INFO
Log Message. Node *node_id*: Table with ID = *table_id* created **Description.** A table having the table ID shown has been created	**Event Name.** TableCreated **Event Type.** Statistic **Priority.** 7 **Severity.** INFO
Log Message. Node *node_id*: Mean loop Counter in doJob last 8192 times = *count* **Description.**	**Event Name.** JobStatistic **Event Type.** Statistic **Priority.** 9 **Severity.** INFO
Log Message. Mean send size to Node = *node_id* last 4096 sends = *bytes* bytes **Description.** This node is sending an average of *bytes* bytes per send to node *node_id*	**Event Name.** SendBytesStatistic **Event Type.** Statistic **Priority.** 9 **Severity.** INFO

Log Message. `Mean receive size to Node = node_id last 4096 sends = bytes bytes` **Description.** This node is receiving an average of *bytes* of data each time it receives data from node *node_id*	**Event Name.** `ReceiveBytesStatistic` **Event Type.** `Statistic` **Priority.** 9 **Severity.** `INFO`
Log Message. `Node node_id: Data usage is data_memory_percentage% (data_pages_used 32K pages of total data_pages_total) / Node node_id: Index usage is index_memory_percentage% (index_pages_used 8K pages of total index_pages_total)` **Description.** This report is generated when a `DUMP 1000` command is issued in the cluster management client; for more information, see `DUMP 1000`, in MySQL Cluster Internals	**Event Name.** `MemoryUsage` **Event Type.** `Statistic` **Priority.** 5 **Severity.** `INFO`
Log Message. `Node node1_id: Transporter to node node2_id reported error error_code: error_message` **Description.** A transporter error occurred while communicating with node *node2_id*; for a listing of transporter error codes and messages, see NDB Transporter Errors, in MySQL Cluster Internals	**Event Name.** `TransporterError` **Event Type.** `Error` **Priority.** 2 **Severity.** `ERROR`
Log Message. `Node node1_id: Transporter to node node2_id reported error error_code: error_message` **Description.** A warning of a potential transporter problem while communicating with node *node2_id*; for a listing of transporter error codes and messages, see NDB Transporter Errors, for more information	**Event Name.** `TransporterWarning` **Event Type.** `Error` **Priority.** 8 **Severity.** `WARNING`
Log Message. `Node node1_id: Node node2_id missed heartbeat heartbeat_id` **Description.** This node missed a heartbeat from node *node2_id*	**Event Name.** `MissedHeartbeat` **Event Type.** `Error` **Priority.** 8 **Severity.** `WARNING`
Log Message. `Node node1_id: Node node2_id declared dead due to missed heartbeat` **Description.** This node has missed at least 3 heartbeats from node *node2_id*, and so has declared that node “dead”	**Event Name.** `DeadDueToHeartbeat` **Event Type.** `Error` **Priority.** 8 **Severity.** `ALERT`
Log Message. `Node node1_id: Node Sent Heartbeat to node = node2_id` **Description.** This node has sent a heartbeat to node *node2_id*	**Event Name.** `SentHeartbeat` **Event Type.** `Info` **Priority.** 12 **Severity.** `INFO`
Log Message. `Node node_id: Event buffer status: used=bytes_used (percent_used%) alloc=bytes_allocated (percent_available%) max=bytes_available apply_gci=latest_restorable_GCI latest_gci=latest_GCI` **Description.** This report is seen during heavy event buffer usage, for example, when many updates are being applied in a relatively short period of time; the report shows the number of bytes and the percentage of event buffer memory used, the bytes allocated and percentage still available, and the latest	**Event Name.** `EventBufferStatus` **Event Type.** `Info` **Priority.** 7 **Severity.** `INFO`

and latest restorable global checkpoints	
Log Message. `Node node_id: Entering single user mode`, `Node node_id: Entered single user mode Node API_node_id has exclusive access`, `Node node_id: Entering single user mode` **Description.** These reports are written to the cluster log when entering and exiting single user mode; *`API_node_id`* is the node ID of the API or SQL having exclusive access to the cluster (fro mroe information, see Section 7.6, "MySQL Cluster Single User Mode"); the message `Unknown single user report API_node_id` indicates an error has taken place and should never be seen in normal operation	**Event Name.** `SingleUser` **Event Type.** `Info` **Priority.** 7 **Severity.** `INFO`
Log Message. `Node node_id: Backup backup_id started from node mgm_node_id` **Description.** A backup has been started using the management node having *`mgm_node_id`*; this message is also displayed in the cluster management client when the `START BACKUP` command is issued; for more information, see Section 7.3.2, "Using The MySQL Cluster Management Client to Create a Backup"	**Event Name.** `BackupStarted` **Event Type.** `Backup` **Priority.** 7 **Severity.** `INFO`
Log Message. `Node node_id: Backup backup_id started from node mgm_node_id completed. StartGCP: start_gcp StopGCP: stop_gcp #Records: records #LogRecords: log_records Data: data_bytes bytes Log: log_bytes bytes` **Description.** The backup having the ID *`backup_id`* has been completed; for more information, see Section 7.3.2, "Using The MySQL Cluster Management Client to Create a Backup"	**Event Name.** `BackupCompleted` **Event Type.** `Backup` **Priority.** 7 **Severity.** `INFO`
Log Message. `Node node_id: Backup request from mgm_node_id failed to start. Error: error_code` **Description.** The backup failed to start; for error codes, see MGM API Errors	**Event Name.** `BackupFailedToStart` **Event Type.** `Backup` **Priority.** 7 **Severity.** `ALERT`
Log Message. `Node node_id: Backup backup_id started from mgm_node_id has been aborted. Error: error_code` **Description.** The backup was terminated after starting, possibly due to user intervention	**Event Name.** `BackupAborted` **Event Type.** `Backup` **Priority.** 7 **Severity.** `ALERT`

7.5.2. MySQL Cluster — 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)

Error Code	Error Name	Error Text
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 disconnected
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

7.6. MySQL Cluster Single User Mode

Single user mode allows the database administrator to restrict access to the database system to a single API node, such as a MySQL server (SQL node) or an instance of `ndb_restore`. When entering single user mode, connections to all other API nodes are closed gracefully and all running transactions are aborted. No new transactions are permitted to start.

Once the cluster has entered single user mode, only the designated API node is granted access to the database.

You can use the `ALL STATUS` command to see when the cluster has entered single user mode.

Example:

```
ndb_mgm> ENTER SINGLE USER MODE 5
```

After this command has executed and the cluster has entered single user mode, the API node whose node ID is `5` becomes the cluster's only permitted user.

The node specified in the preceding command must be an API node; attempting to specify any other type of node will be rejected.

> **Note**
>
> When the preceding command is invoked, all transactions running on the designated node are aborted, the connection is closed, and the server must be restarted.

The command `EXIT SINGLE USER MODE` changes the state of the cluster's data nodes from single user mode to normal mode. API nodes — such as MySQL Servers — waiting for a connection (that is, waiting for the cluster to become ready and available), are again permitted to connect. The API node denoted as the single-user node continues to run (if still connected) during and after the state change.

Example:

```
ndb_mgm> EXIT SINGLE USER MODE
```

There are two recommended ways to handle a node failure when running in single user mode:

- Method 1:

  1. Finish all single user mode transactions
  2. Issue the `EXIT SINGLE USER MODE` command
  3. Restart the cluster's data nodes

- Method 2:

  Restart database nodes prior to entering single user mode.

7.7. Quick Reference: MySQL Cluster SQL Statements

This section discusses several SQL statements that can prove useful in managing and monitoring a MySQL server that is connected to a MySQL Cluster, and in some cases provide information about the cluster itself.

- `SHOW ENGINE NDB STATUS`, `SHOW ENGINE NDBCLUSTER STATUS`

  The output of this statement contains information about the server's connection to the cluster, creation and usage of MySQL Cluster objects, and binary logging for MySQL Cluster replication.

  See `SHOW ENGINE` Syntax, for a usage example and more detailed information.

- `SHOW ENGINES`

  This statement can be used to determine whether or not clustering support is enabled in the MySQL server, and if so, whether it is active.

  See `SHOW ENGINES` Syntax, for more detailed information.

  > **Note**
  >
  > In MySQL 5.1, this statement no longer supports a `LIKE` clause. However, you can use `LIKE` to filter queries against the `INFORMATION_SCHEMA.ENGINES`, as discussed in the next item.

-

```
SELECT * FROM INFORMATION_SCHEMA.ENGINES [WHERE ENGINE LIKE 'NDB%']
```

This is the equivalent of `SHOW ENGINES`, but uses the `ENGINES` table of the `INFORMATION_SCHEMA` database (available beginning with MySQL 5.1.5). Unlike the case with the `SHOW ENGINES` statement, it is possible to filter the results using a `LIKE` clause, and to select specific columns to obtain information that may be of use in scripts. For example, the following query shows whether the server was built with `NDB` support and, if so, whether it is enabled:

```
mysql> SELECT SUPPORT FROM INFORMATION_SCHEMA.ENGINES
    ->   WHERE ENGINE LIKE 'NDB%';
+---------+
| support |
+---------+
| ENABLED |
+---------+
```

See The `INFORMATION_SCHEMA ENGINES` Table, for more information.

- `SHOW VARIABLES LIKE 'NDB%'`

  This statement provides a list of most server system variables relating to the `NDB` storage engine, and their values, as shown here:

```
mysql> SHOW VARIABLES LIKE 'NDB%';
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
| ndb_autoincrement_prefetch_sz     | 32    |
| ndb_cache_check_time              | 0     |
| ndb_extra_logging                 | 0     |
| ndb_force_send                    | ON    |
| ndb_index_stat_cache_entries      | 32    |
| ndb_index_stat_enable             | OFF   |
| ndb_index_stat_update_freq        | 20    |
| ndb_report_thresh_binlog_epoch_slip | 3     |
| ndb_report_thresh_binlog_mem_usage | 10    |
| ndb_use_copying_alter_table       | OFF   |
| ndb_use_exact_count               | ON    |
| ndb_use_transactions              | ON    |
+-----------------------------------+-------+
```

  See Server System Variables, for more information.

- `SELECT * FROM INFORMATION_SCHEMA.GLOBAL_VARIABLES WHERE VARIABLE_NAME LIKE 'NDB%';`

  This statement is the equivalent of the `SHOW` command described in the previous item, and provides almost identical output, as shown here:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.GLOBAL_VARIABLES
    ->   WHERE VARIABLE_NAME LIKE 'NDB%';
+-----------------------------------+----------------+
| VARIABLE_NAME                     | VARIABLE_VALUE |
+-----------------------------------+----------------+
| NDB_AUTOINCREMENT_PREFETCH_SZ     | 32             |
| NDB_CACHE_CHECK_TIME              | 0              |
| NDB_EXTRA_LOGGING                 | 0              |
| NDB_FORCE_SEND                    | ON             |
| NDB_INDEX_STAT_CACHE_ENTRIES      | 32             |
| NDB_INDEX_STAT_ENABLE             | OFF            |
| NDB_INDEX_STAT_UPDATE_FREQ        | 20             |
| NDB_REPORT_THRESH_BINLOG_EPOCH_SLIP | 3              |
| NDB_REPORT_THRESH_BINLOG_MEM_USAGE | 10             |
| NDB_USE_COPYING_ALTER_TABLE       | OFF            |
| NDB_USE_EXACT_COUNT               | ON             |
| NDB_USE_TRANSACTIONS              | ON             |
+-----------------------------------+----------------+
```

  Unlike the case with the `SHOW` command, it is possible to select individual columns. For example:

```
mysql> SELECT VARIABLE_VALUE
    ->   FROM INFORMATION_SCHEMA.GLOBAL_VARIABLES
    ->   WHERE VARIABLE_NAME = 'ndb_force_send';
+----------------+
| VARIABLE_VALUE |
+----------------+
| ON             |
+----------------+
```

  See The `INFORMATION_SCHEMA GLOBAL_VARIABLES` and `SESSION_VARIABLES` Tables, and Server System Variables, for more information.

- `SHOW STATUS LIKE 'NDB%'`

  This statement shows at a glance whether or not the MySQL server is acting as a cluster SQL node, and if so, it provides the MySQL server's cluster node ID, the host name and port for the cluster management server to which it is connected, and the number of data nodes in the cluster, as shown here:

```
mysql> SHOW STATUS LIKE 'NDB%';
+--------------------------+----------------+
| Variable_name            | Value          |
+--------------------------+----------------+
| Ndb_cluster_node_id      | 10             |
| Ndb_config_from_host     | 192.168.0.103 |
| Ndb_config_from_port     | 1186           |
| Ndb_number_of_data_nodes | 4              |
+--------------------------+----------------+
```

  If the MySQL server was built with clustering support, but it is not connected to a cluster, all rows in the output of this statement contain a zero or an empty string:

```
mysql> SHOW STATUS LIKE 'NDB%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| Ndb_cluster_node_id      | 0     |
| Ndb_config_from_host     |       |
| Ndb_config_from_port     | 0     |
| Ndb_number_of_data_nodes | 0     |
+--------------------------+-------+
```

  See also `SHOW STATUS` Syntax.

- `SELECT * FROM INFORMATION_SCHEMA.GLOBAL_STATUS WHERE VARIABLE_NAME LIKE 'NDB%';`

  Beginning with MySQL 5.1.12, this statement provides similar output to the `SHOW` command discussed in the previous item. However, unlike the case with `SHOW STATUS`, it is possible using the `SELECT` to extract values in SQL for use in scripts for monitoring and automation purposes.

  See The `INFORMATION_SCHEMA GLOBAL_STATUS` and `SESSION_STATUS` Tables, for more information.

7.8. Adding MySQL Cluster Data Nodes Online

This section describes how to add MySQL Cluster data nodes “online” — that is, without needing to shut down the cluster completely and restart it as part of the process. This capability is available in MySQL Cluster NDB 7.0 (beginning with MySQL Cluster NDB 6.4.0) and later MySQL Cluster release series.

> **Important**
>
> Currently, you must add new data nodes to a MySQL Cluster as part of a new node group. In addition, it is not possible to change the number of replicas (or the number of nodes per node group) online.

7.8.1. Adding MySQL Cluster Data Nodes Online: General Issues

This section provides general information about the behavior of and current limitations in adding MySQL Cluster nodes online.

Redistribution of Data. The ability to add new nodes online includes a means to reorganize `NDBCLUSTER` table data and indexes so that they are distributed across all data nodes, including the new ones. Table reorganization of both in-memory and Disk Data tables is supported. This redistribution does not currently include unique indexes (only ordered indexes are redistributed) or `BLOB` table data, but we are working to add redistibution of these in the near future. The redistribution for `NDBCLUSTER` tables already existing before the new data nodes were added is not automatic, but can be accomplished using simple SQL statements in `mysql` or another MySQL client application. However, all data and indexes added to tables created after a new node group has been added are distributed automatically among all cluster data nodes, including those added as part of the new node group.

Partial starts. It is possible to add a new node group without all of the new data nodes being started. It is also possible to add a new node group to a degraded cluster — that is, a cluster that is only partially started, or where one or more data nodes are not running. In the latter case, the cluster must have enough nodes running to be viable before the new node group can be added.

Effects on ongoing operations. Normal DML operations using MySQL Cluster data are not prevented by the creation or addition of a new node group, or by table reorganization. However, it is not possible to perform DDL concurrently with table reorganization — that is, no other DDL statements can be issued while an `ALTER TABLE ... REORGANIZE PARTITION` statement is executing. In addition, during the execution of `ALTER TABLE ... REORGANIZE PARTITION` (or the execution of any other DDL statement), it is not possible to restart cluster data nodes.

Failure handling. Failures of data nodes during node group creation and table reorganization are handled as hown in the following table:

Failure occurs during:	Failure occurs in:		
	"Old" data nodes	**"New" data nodes**	**System**

Node group creation	• **If a node other than the master fails:** The creation of the node group is always rolled forward.
	• **If the master fails:**
	  • **If the internal commit point has been reached:** The creation of the node group is rolled forward.
	  • **If the internal commit point has not yet been reached.** The creation of the node group is rolled back
		• **If a node other than the master fails:** The creation of the node group is always rolled forward.
		• **If the master fails:**
		  • **If the internal commit point has been reached:** The creation of the node group is rolled forward.
		  • **If the internal commit point has not yet been reached.** The creation of the node group is rolled back
			• **If the execution of `CREATE NODEGROUP` has reached the internal commit point:** When restarted, the cluster includes the new node group. Otherwise it without.
			• **If the execution of `CREATE NODEGROUP` has not yet reached the internal commit point:** When restarted, the cluster does not include the new node group.

Table reorganization	• **If a node other than the master fails:** The table reorganization is always rolled forward.
	• **If the master fails:**
	  • **If the internal commit point has been reached:** The table reorganization is rolled forward.
	  • **If the internal commit point has not yet been reached.** The table reorganization is rolled back.
		• **If a node other than the master fails:** The table reorganization is always rolled forward.
		• **If the master fails:**
		  • **If the internal commit point has been reached:** The table reorganization is rolled forward.
		  • **If the internal commit point has not yet been reached.** The table reorganization is rolled back.
			• **If the execution of an `ALTER ONLINE TABLE table REORGANIZE PARTITION` statement has reached the internal commit point:** When the cluster is restarted, the data and indexes belonging to *`table`* are distributed using the "new" data nodes.
			• **If the execution of an `ALTER ONLINE TABLE table REORGANIZE PARTITION` statement has not yet reached the internal commit point:** When the cluster is restarted, the data and indexes belonging to *`table`* are distributed using only the "old" data nodes.

Dropping node groups. The `ndb_mgm` client supports a `DROP NODEGROUP` command, but it is possible to drop a node group only when no data nodes in the node group contain any data. Since there is currently no way to "empty" a specific data node or node group, this command works only the following two cases:

1. After issuing `CREATE NODEGROUP` in the `ndb_mgm` client, but before issuing any `ALTER ONLINE TABLE ... REORGANIZE PARTITION` statements in the `mysql` client.
2. After dropping all `NDBCLUSTER` tables using `DROP TABLE`.

   `TRUNCATE` does not work for this purpose because the data nodes continue to store the table definitions.

7.8.2. Adding MySQL Cluster Data Nodes Online: Basic procedure

In this section, we list the basic steps required to add new data nodes to a MySQL Cluster. For a detailed example, see Section 7.8.3, “Adding MySQL Cluster Data Nodes Online: Detailed Example”.

> **Note**
>
> Beginning with MySQL Cluster NDB 7.0.4, this procedure applies whether you are using `ndbd` or `ndbmtd` binaries for the data node processes. Previously, this did not work with multi-threaded data nodes. (Bug#43108)

Assuming that you already have a running MySQL Cluster, adding data nodes online requires the following steps:

1. Edit the cluster configuration `config.ini` file, adding new `[ndbd]` sections corresponding to the nodes to be added. In the case where the cluster uses multiple management servers, these changes need to be made to all `config.ini` files used by the management servers.
2. Perform a rolling restart of all MySQL Cluster management servers.

   > **Important**
   >
   > All management servers must be restarted with the `--reload` or `--initial` option to force the reading of the new configuration.

3. Perform a rolling restart of all existing MySQL Cluster data nodes.

   > **Note**
   >
   > It is not necessary to use `--initial` when restarting the existing data nodes.

4. Perform a rolling restart of any SQL or API nodes connected to the MySQL Cluster.
5. Perform an initial start of the new data nodes.

   > **Note**
   >
   > The new data nodes may be started in any order, and can also be started concurrently, as long as they are started after the rolling restarts of all existing nodes have been completed and before proceeding to the next step.

6. Execute one or more `CREATE NODEGROUP` commands in the MySQL Cluster management client to create the new node group or node groups to which the new data nodes will belong.
7. Redistribute the cluster's data among all data nodes (including the new ones) by issuing an `ALTER ONLINE TABLE ... REORGANIZE PARTITION` statement in the `mysql` client for each `NDBCLUSTER` table.

   > **Note**
   >
   > This needs to be done only for tables already existing at the time the new node group is added. Data in tables created after the new node group is added is distributed automatically; however, data added to any given table `tbl` that existed before the new nodes were added is not distributed using the new nodes until that table has been reorganized using `ALTER ONLINE TABLE tbl REORGANIZE PARTITION`.

8. Reclaim the space freed on the “old” nodes by issuing, for each `NDBCLUSTER` table, an `OPTIMIZE TABLE` statement in the `mysql` client.

7.8.3. Adding MySQL Cluster Data Nodes Online: Detailed Example

In this section we provide a detailed example illustrating how to add new MySQL Cluster data nodes online, starting with a MySQL Cluster having 2 data nodes in a single node group and concluding with a cluster having 4 data nodes in 2 node groups.

Starting configuration. For purposes of illustration, we assume a minimal configuration, and that the cluster uses a `config.ini` file containing only the following information:

```
[ndbd default]
DataMemory = 100M
IndexMemory = 100M
NoOfReplicas = 2
DataDir = /usr/local/mysql/var/mysql-cluster
[ndbd]
Id = 1
HostName = 192.168.0.1
[ndbd]
Id = 2
HostName = 192.168.0.2
[mgm]
```

```
HostName = 192.168.0.10
Id = 10
[api]
Id=20
HostName = 192.168.0.20
[api]
Id=21
HostName = 192.168.0.21
```

We also assume that you have already started the cluster using the appropriate command line or `my.cnf` options, and that running `SHOW` in the management client produces output similar to what is shown here:

```
-- NDB Cluster -- Management Client --
ndb_mgm> SHOW
Connected to Management Server at: 192.168.0.10:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @192.168.0.1  (5.1.34-ndb-7.0.7, Nodegroup: 0, Master)
id=2    @192.168.0.2  (5.1.34-ndb-7.0.7, Nodegroup: 0)
[ndb_mgmd(MGM)] 1 node(s)
id=10   @192.168.0.10  (5.1.34-ndb-7.0.7)
[mysqld(API)]   2 node(s)
id=20   @192.168.0.20  (5.1.34-ndb-7.0.7)
id=21   @192.168.0.21  (5.1.34-ndb-7.0.7)
```

Finally, we assume that the cluster contains a single `NDBCLUSTER` table created as shown here:

```
USE n;
CREATE TABLE ips (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    country_code CHAR(2) NOT NULL,
    type CHAR(4) NOT NULL,
    ip_address varchar(15) NOT NULL,
    addresses BIGINT UNSIGNED DEFAULT NULL,
    date BIGINT UNSIGNED DEFAULT NULL
)   ENGINE NDBCLUSTER;
```

The memory usage and related information shown later in this section was generated after inserting approximately 50000 rows into this table.

> **Note**
>
> In this example, we show the single-threaded `ndbd` being used for the data node processes. However — beginning with MySQL Cluster NDB 7.0.4 (Bug#43108) — you can also apply this example if you are using the multi-threaded `ndbmtd` by substituting `ndbmtd` for `ndbd` wherever it appears in the steps that follow.

Step 1: Update configuration file. Open the cluster global configuration file in a text editor and add `[ndbd]` sections corresponding to the 2 new data nodes. (We give these data nodes IDs 3 and 4, and assume that they are to be run on host machines at addresses 192.168.0.3 and 192.168.0.4, respectively.) After you have added the new sections, the contents of the `config.ini` file should look like what is shown here, where the additions to the file are shown in bold type:

```
[ndbd default]
DataMemory = 100M
IndexMemory = 100M
NoOfReplicas = 2
DataDir = /usr/local/mysql/var/mysql-cluster
[ndbd]
Id = 1
HostName = 192.168.0.1
[ndbd]
Id = 2
HostName = 192.168.0.2
[ndbd]
Id = 3
HostName = 192.168.0.3
[ndbd]
Id = 4
HostName = 192.168.0.4
[mgm]
HostName = 192.168.0.10
Id = 10
[api]
Id=20
HostName = 192.168.0.20
[api]
Id=21
HostName = 192.168.0.21
```

Once you have made the necessary changes, save the file.

Step 2: Restart the management server. Restarting the cluster management server requires that you issue separate commands to stop the management server and then to start it again, as follows:

1. Stop the management server using the management client `STOP` command, as shown here:

```
ndb_mgm> 10 STOP
Node 10 has shut down.
Disconnecting to allow Management Server to shutdown
shell>
```

2. Because shutting down the management server causes the management client to terminate, you must start the management server from the system shell. For simplicity, we assume that `config.ini` is in the same directory as the management server binary, but in practice, you must supply the correct path to the configuration file. You must also supply the `--reload` or `--initial` option so that the management server reads the new configuration from the file rather than its configuration cache. If your shell's current directory is also the same as the directory where the management server binary is located, then you can invoke the management server as shown here:

```
shell> ndb_mgmd -f config.ini --reload
2008-12-08 17:29:23 [MgmSrvr] INFO     -- NDB Cluster Management Server. 5.1.34-ndb-7.0.7
2008-12-08 17:29:23 [MgmSrvr] INFO     -- Reading cluster configuration from 'config.ini'
```

If you check the output of `SHOW` in the management client after restarting the `ndb_mgm` process, you should now see something like this:

```
-- NDB Cluster -- Management Client --
ndb_mgm> SHOW
Connected to Management Server at: 192.168.0.10:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @192.168.0.1  (5.1.34-ndb-7.0.7, Nodegroup: 0, Master)
id=2    @192.168.0.2  (5.1.34-ndb-7.0.7, Nodegroup: 0)
id=3 (not connected, accepting connect from 192.168.0.3)
id=4 (not connected, accepting connect from 192.168.0.4)
[ndb_mgmd(MGM)] 1 node(s)
id=10   @192.168.0.10  (5.1.34-ndb-7.0.7)
[mysqld(API)]   2 node(s)
id=20   @192.168.0.20  (5.1.34-ndb-7.0.7)
id=21   @192.168.0.21  (5.1.34-ndb-7.0.7)
```

Step 3: Perform a rolling restart of the existing data nodes. This step can be accomplished entirely within the cluster management client using the `RESTART` command, as shown here:

```
ndb_mgm> 1 RESTART
Node 1: Node shutdown initiated
Node 1: Node shutdown completed, restarting, no start.
Node 1 is being restarted
ndb_mgm> Node 1: Start initiated (version 7.0.7)
Node 1: Started (version 7.0.7)
ndb_mgm> 2 RESTART
Node 2: Node shutdown initiated
Node 2: Node shutdown completed, restarting, no start.
Node 2 is being restarted
ndb_mgm> Node 2: Start initiated (version 7.0.7)
ndb_mgm> Node 2: Started (version 7.0.7)
```

> **Important**
>
> After issuing each *`X`* `RESTART` command, wait until the management client reports `Node X: Started (version ...)` *before* proceeding any further.

Step 4: Perform a rolling restart of all cluster API nodes. Shut down and restart each MySQL server acting as an SQL node in the cluster using `mysqladmin shutdown` followed by `mysqld_safe` (or another startup script). This should be similar to what is shown here, where *`password`* is the MySQL `root` password for a given MySQL server instance:

```
shell> mysqladmin -uroot -ppassword shutdown
081208 20:19:56 mysqld_safe mysqld from pid file
/usr/local/mysql/var/tonfisk.pid ended
shell> mysqld_safe --ndbcluster --ndb-connectstring=192.168.0.10 &
081208 20:20:06 mysqld_safe Logging to '/usr/local/mysql/var/tonfisk.err'.
081208 20:20:06 mysqld_safe Starting mysqld daemon with databases
from /usr/local/mysql/var
```

Of course, the exact input and output depend on how and where MySQL is installed on the system, as well as which options you choose to start it (and whether or not some or all of these options are specified in a `my.cnf` file).

Step 5: Perform an initial start of the new data nodes. From a system shell on each of the hosts for the new data nodes, start the data nodes as shown here, using the `--initial` option:

```
shell> ndbd -c 192.168.0.10 --initial
```

> **Note**

Unlike the case with restarting the existing data nodes, you can start the new data nodes concurrently; you do not need to wait for one to finish starting before starting the other.

Wait until both of the new data nodes have started before proceeding with the next step. Once the new data nodes have started, you can see in the output of the management client `SHOW` command that they do not yet belong to any node group (as indicated with bold type here):

```
ndb_mgm> SHOW
Connected to Management Server at: 192.168.0.10:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @192.168.0.1  (5.1.34-ndb-7.0.7, Nodegroup: 0, Master)
id=2    @192.168.0.2  (5.1.34-ndb-7.0.7, Nodegroup: 0)
id=3    @192.168.0.3  (5.1.34-ndb-7.0.7, no nodegroup)
id=4    @192.168.0.4  (5.1.34-ndb-7.0.7, no nodegroup)
[ndb_mgmd(MGM)] 1 node(s)
id=10   @192.168.0.10  (5.1.34-ndb-7.0.7)
[mysqld(API)]   2 node(s)
id=20   @192.168.0.20  (5.1.34-ndb-7.0.7)
id=21   @192.168.0.21  (5.1.34-ndb-7.0.7)
```

Step 6: Create a new node group. You can do this by issuing a `CREATE NODEGROUP` command in the cluster management client. This command takes as its argument a comma-separated list of the node IDs of the data nodes to be included in the new node group, as shown here:

```
ndb_mgm> CREATE NODEGROUP 3,4
Nodegroup 1 created
```

By issuing `SHOW` again, you can verify that data nodes 3 and 4 have joined the new node group (again indicated in bold type):

```
ndb_mgm> SHOW
Connected to Management Server at: 192.168.0.10:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @192.168.0.1  (5.1.34-ndb-7.0.7, Nodegroup: 0, Master)
id=2    @192.168.0.2  (5.1.34-ndb-7.0.7, Nodegroup: 0)
id=3    @192.168.0.3  (5.1.34-ndb-7.0.7, Nodegroup: 1)
id=4    @192.168.0.4  (5.1.34-ndb-7.0.7, Nodegroup: 1)
[ndb_mgmd(MGM)] 1 node(s)
id=10   @192.168.0.10  (5.1.34-ndb-7.0.7)
[mysqld(API)]   2 node(s)
id=20   @192.168.0.20  (5.1.34-ndb-7.0.7)
id=21   @192.168.0.21  (5.1.34-ndb-7.0.7)
```

Step 7: Redistribute cluster data. When a node group is created, existing data and indexes are not automatically distributed to the new node group's data nodes, as you can see by issuing the appropriate `REPORT` command in the management client:

```
ndb_mgm> ALL REPORT MEMORY
Node 1: Data usage is 5%(177 32K pages of total 3200)
Node 1: Index usage is 0%(108 8K pages of total 12832)
Node 2: Data usage is 5%(177 32K pages of total 3200)
Node 2: Index usage is 0%(108 8K pages of total 12832)
Node 3: Data usage is 0%(0 32K pages of total 3200)
Node 3: Index usage is 0%(0 8K pages of total 12832)
Node 4: Data usage is 0%(0 32K pages of total 3200)
Node 4: Index usage is 0%(0 8K pages of total 12832)
```

By using `ndb_desc` with the `-p` option, which causes the output to include partitioning information, you can see that the table still uses only 2 partitions (in the `Per partition info` section of the output, shown here in bold text):

```
shell> ndb_desc -c 192.168.0.10 -d n ips -p
-- ips --
Version: 1
Fragment type: 9
K Value: 6
Min load factor: 78
Max load factor: 80
Temporary table: no
Number of attributes: 6
Number of primary keys: 1
Length of frm data: 340
Row Checksum: 1
Row GCI: 1
SingleUserMode: 0
ForceVarPart: 1
FragmentCount: 2
TableStatus: Retrieved
-- Attributes --
id Bigint PRIMARY KEY DISTRIBUTION KEY AT=FIXED ST=MEMORY AUTO_INCR
country_code Char(2;latin1_swedish_ci) NOT NULL AT=FIXED ST=MEMORY
type Char(4;latin1_swedish_ci) NOT NULL AT=FIXED ST=MEMORY
ip_address Varchar(15;latin1_swedish_ci) NOT NULL AT=SHORT_VAR ST=MEMORY
addresses Bigunsigned NULL AT=FIXED ST=MEMORY
date Bigunsigned NULL AT=FIXED ST=MEMORY
-- Indexes --
PRIMARY KEY(id) - UniqueHashIndex
```

```
PRIMARY(id) - OrderedIndex
-- Per partition info --
Partition   Row count   Commit count  Frag fixed memory   Frag varsized memory
0           26086       26086         1572864             557056
1           26329       26329         1605632             557056
NDBT_ProgramExit: 0 - OK
```

You can cause the data to be redistributed among all of the data nodes by performing, for each NDBCLUSTER table, an ALTER ONLINE TABLE ... REORGANIZE PARTITION statement in the mysql client. After issuing the statement ALTER ONLINE TABLE ips REORGANIZE PARTITION, you can see using ndb_desc that the data for this table is now stored using 4 partitions, as shown here (with the relevant portions of the output in bold type):

```
shell> ndb_desc -c 192.168.0.10 -d n ips -p
-- ips --
Version: 16777217
Fragment type: 9
K Value: 6
Min load factor: 78
Max load factor: 80
Temporary table: no
Number of attributes: 6
Number of primary keys: 1
Length of frm data: 341
Row Checksum: 1
Row GCI: 1
SingleUserMode: 0
ForceVarPart: 1
FragmentCount: 4
TableStatus: Retrieved
-- Attributes --
id Bigint PRIMARY KEY DISTRIBUTION KEY AT=FIXED ST=MEMORY AUTO_INCR
country_code Char(2;latin1_swedish_ci) NOT NULL AT=FIXED ST=MEMORY
type Char(4;latin1_swedish_ci) NOT NULL AT=FIXED ST=MEMORY
ip_address Varchar(15;latin1_swedish_ci) NOT NULL AT=SHORT_VAR ST=MEMORY
addresses Bigunsigned NULL AT=FIXED ST=MEMORY
date Bigunsigned NULL AT=FIXED ST=MEMORY
-- Indexes --
PRIMARY KEY(id) - UniqueHashIndex
PRIMARY(id) - OrderedIndex
-- Per partition info --
Partition   Row count   Commit count  Frag fixed memory   Frag varsized memory
0           12981       52296         1572864             557056
1           13236       52515         1605632             557056
2           13105       13105         819200              294912
3           13093       13093         819200              294912
NDBT_ProgramExit: 0 - OK
```

> **Note**
>
> Normally, ALTER [ONLINE] TABLE *table_name* REORGANIZE PARTITION is used with a list of partition identifiers and a set of partition definitions to create a new partitioning scheme for a table that has already been explicitly partitioned. Its use here to redistribute data onto a new MySQL Cluster node group is an exception in this regard; when used in this way, only the name of the table is used following the TABLE keyword, and no other keywords or identifiers follow REORGANIZE PARTITION.
>
> Prior to MySQL Cluster NDB 6.4.3, ALTER ONLINE TABLE ... REORGANIZE PARTITION with no *partition_names* INTO (*partition_definitions*) option did not work correctly with Disk Data tables or with in-memory NDBCLUSTER tables having one or more disk-based columns. (Bug#42549)
>
> For more information, see ALTER TABLE Syntax.

Also, for each table, the ALTER ONLINE TABLE statement should be followed by an OPTIMIZE TABLE to reclaim wasted space. You can obtain a list of all NDBCLUSTER tables using the following query against the INFORMATION_SCHEMA.TABLES table:

```
SELECT TABLE_SCHEMA, TABLE_NAME
    FROM INFORMATION_SCHEMA.TABLES
    WHERE ENGINE = 'NDBCLUSTER';
```

> **Note**
>
> The INFORMATION_SCHEMA.TABLES.ENGINE value for a MySQL Cluster table is always NDBCLUSTER, regardless of whether the CREATE TABLE statement used to create the table (or ALTER TABLE statement used to convert an existing table from a different storage engine) used NDB or NDBCLUSTER in its ENGINE option.

You can see after performing these statements in the output of ALL REPORT MEMORY that the data and indexes are now redistributed between all cluster data nodes, as shown here:

```
ndb_mgm> ALL REPORT MEMORY
Node 1: Data usage is 5%(176 32K pages of total 3200)
Node 1: Index usage is 0%(76 8K pages of total 12832)
Node 2: Data usage is 5%(176 32K pages of total 3200)
Node 2: Index usage is 0%(76 8K pages of total 12832)
Node 3: Data usage is 2%(80 32K pages of total 3200)
Node 3: Index usage is 0%(51 8K pages of total 12832)
Node 4: Data usage is 2%(80 32K pages of total 3200)
```

```
Node 4: Index usage is 0%(50 8K pages of total 12832)
```

> **Note**
>
> Since only one DDL operation on NDBCLUSTER tables can be executed at a time, you must wait for each ALTER ONLINE TABLE ... REORGANIZE PARTITION statement to finish before issuing the next one.

It is not necessary to issue ALTER ONLINE TABLE ... REORGANIZE PARTITION statements for NDBCLUSTER tables created *after* the new data nodes have been added; data added to such tables is distributed among all data nodes automatically. However, in NDBCLUSTER tables that existed *prior to* the addition of the new nodes, neither existing nor new data is distributed using the new nodes until these tables have been reorganized using ALTER ONLINE TABLE ... REORGANIZE PARTITION.

Alternative procedure, without rolling restart. It is possible to avoid the need for a rolling restart by configuring the extra data nodes, but not starting them, when first starting the cluster. This can be accomplished by using the NodeGroup data node configuration parameter in the config.ini file, as shown here (note the section with bold text). We assume, as before, that you wish to start with two data nodes — nodes 1 and 2 — in one node group and later to expand the cluster to four data nodes, by adding a second node group consisting of nodes 3 and 4:

```
[ndbd default]
DataMemory = 100M
IndexMemory = 100M
NoOfReplicas = 2
DataDir = /usr/local/mysql/var/mysql-cluster
[ndbd]
Id = 1
HostName = 192.168.0.1
[ndbd]
Id = 2
HostName = 192.168.0.2
[ndbd]
Id = 3
HostName = 192.168.0.3
NodeGroup = 65535
[ndbd]
Id = 4
HostName = 192.168.0.4
NodeGroup = 65535
[mgm]
HostName = 192.168.0.10
Id = 10
[api]
Id=20
HostName = 192.168.0.20
[api]
Id=21
HostName = 192.168.0.21
```

In this case, you must perform the initial start of the cluster using the --nowait option with ndbd (or ndbmtd in MySQL Cluster NDB 7.0.4 and later) for each of the data nodes that you wish to have online immediately, so that the cluster does not wait for the remaining nodes to start:

```
shell> ndbd -c 192.168.0.10 --initial --nowait=3,4
```

When you are ready to add the second nodegroup, you need only perform the following additional steps:

1. Start data nodes 3 and 4, invoking the data node process once for each new node:

   ```
   shell> ndbd -c 192.168.0.10 --initial
   ```

2. Issue the appropriate CREATE NODEGROUP command in the management client:

   ```
   ndb_mgm> CREATE NODEGROUP 3,4
   ```

3. In the mysql client, issue ALTER ONLINE TABLE ... REORGANIZE PARTITION and OPTIMIZE TABLE statements for each existing NDBCLUSTER table. (As noted elsewhere in this section, existing MySQL Cluster tables cannot use the new nodes for data distribution until this has been done.)

Chapter 8. MySQL Cluster Security Issues

This section discusses security considerations to take into account when setting up and running MySQL Cluster.

Topics to be covered in this chapter include the following:

- MySQL Cluster and network security issues
- Configuration issues relating to running MySQL Cluster securely
- MySQL Cluster and the MySQL privilege system
- MySQL standard security procedures as applicable to MySQL Cluster

8.1. MySQL Cluster Security and Networking Issues

In this section, we discuss basic network security issues as they relate to MySQL Cluster. It is extremely important to remember that MySQL Cluster “out of the box” is not secure; you or your network administrator must take the proper steps to insure that your cluster cannot be compromised over the network.

Cluster communication protocols are inherently insecure, and no encryption or similar security measures are used in communications between nodes in the cluster. Because network speed and latency have a direct impact on the cluster's efficiency, it is also not advisable to employ SSL or other encryption to network connections between nodes, as such schemes will effectively slow communications.

It is also true that no authentication is used for controlling API node access to a MySQL Cluster. As with encryption, the overhead of imposing authentication requirements would have an adverse impact on Cluster performance.

In addition, there is no checking of the source IP address for either of the following when accessing the cluster:

- SQL or API nodes using “free slots” created by empty `[mysqld]` or `[api]` sections in the `config.ini` file

  This means that, if there are any empty `[mysqld]` or `[api]` sections in the `config.ini` file, then any API nodes (including SQL nodes) that know the management server's host name (or IP address) and port can connect to the cluster and access its data without restriction. (See Section 8.2, “MySQL Cluster and MySQL Privileges”, for more information about this and related issues.)

  > **Note**
  >
  > You can exercise some control over SQL and API node access to the cluster by specifying a `HostName` parameter for all `[mysqld]` and `[api]` sections in the `config.ini` file. However, this also means that, should you wish to connect an API node to the cluster from a previously unused host, you need to add an `[api]` section containing its host name to the `config.ini` file.
  >
  > More information is available elsewhere in this chapter about the `HostName` parameter. Also see Section 3.3, “Quick Test Setup of MySQL Cluster”, for configuration examples using `HostName` with API nodes.

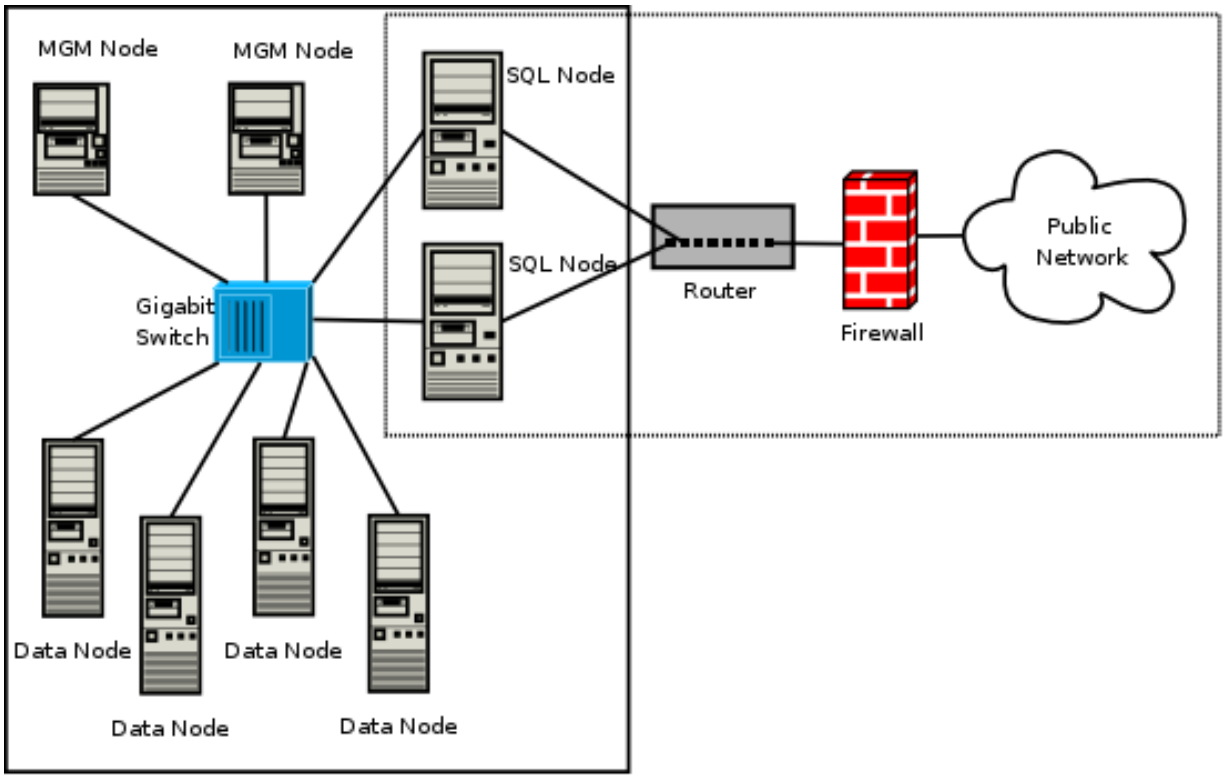
- Any `ndb_mgm` client

  This means that any cluster management client that is given the management server's host name (or IP address) and port (if not the standard port) can connect to the cluster and execute any management client command. This includes commands such as `ALL STOP` and `SHUTDOWN`.

For these reasons, it is necessary to protect the cluster on the network level. The safest network configuration for Cluster is one which isolates connections between Cluster nodes from any other network communications. This can be accomplished by any of the following methods:

1. Keeping Cluster nodes on a network that is physically separate from any public networks. This option is the most dependable; however, it is the most expensive to implement.

   We show an example of a MySQL Cluster setup using such a physically segregated network here:



This setup has two networks, one private (solid box) for the Cluster management servers and data nodes, and one public (dotted box) where the SQL nodes reside. (We show the management and data nodes connected using a gigabit switch since this provides the best performance.) Both networks are protected from the outside by a hardware firewall, sometimes also known as a *network-based firewall*.

This network setup is safest because no packets can reach the cluster's management or data nodes from outside the network — and none of the cluster's internal communications can reach the outside — without going through the SQL nodes, as long as the SQL nodes do not allow any packets to be forwarded. This means, of course, that all SQL nodes must be secured against hacking attempts.

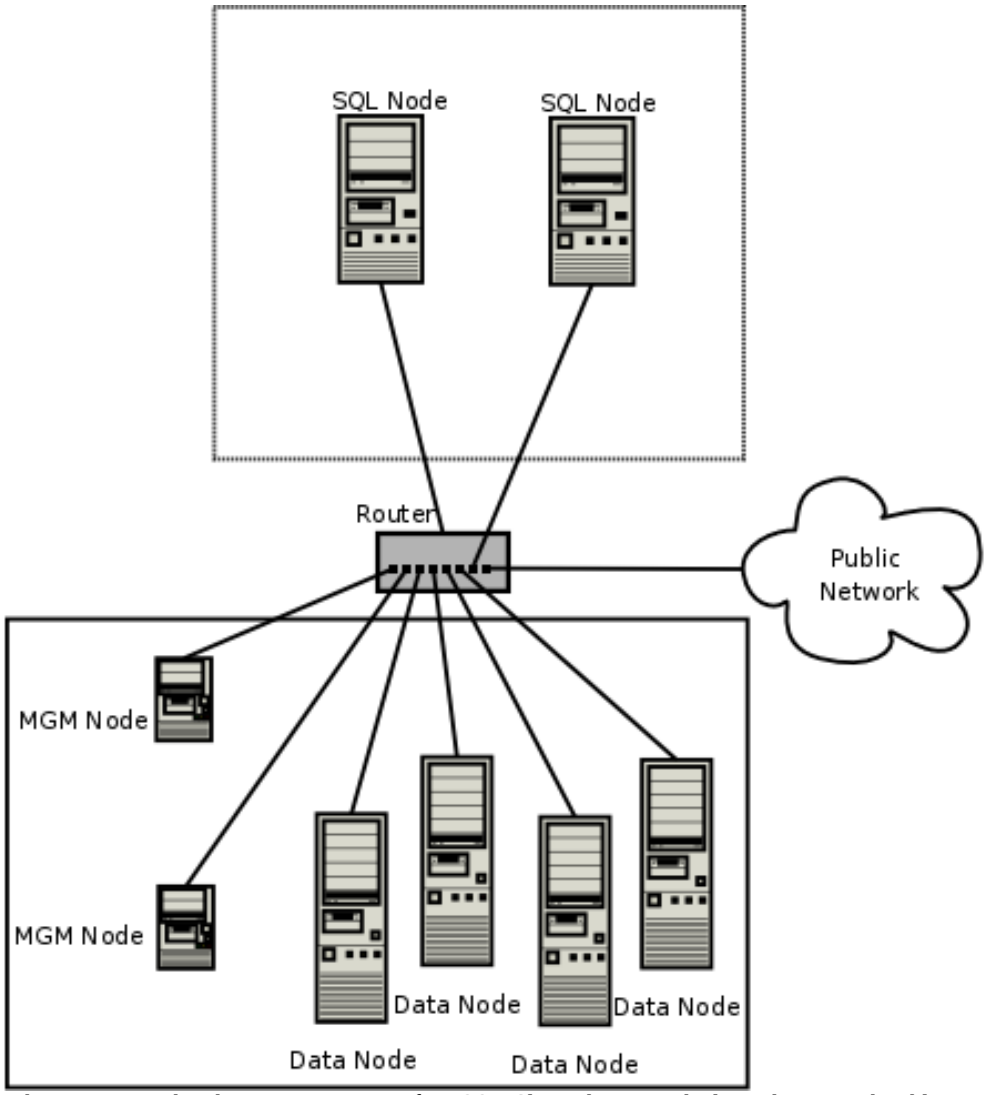
Important

With regard to potential security vulnerabilities, an SQL node is no different from any other MySQL server. See Making MySQL Secure Against Attackers, for a description of techniques you can use to secure MySQL servers.

2. Using one or more software firewalls (also known as *host-based firewalls*) to control which packets pass through to the cluster from portions of the network that do not require access to it. In this type of setup, a software firewall must be installed on every host in the cluster which might otherwise be accessible from outside the local network.

   The host-based option is the least expensive to implement, but relies purely on software to provide protection and so is the most difficult to keep secure.

   This type of network setup for MySQL Cluster is illustrated here:



Using this type of network setup means that there are two zones of MySQL Cluster hosts. Each cluster host must be able to communicate with all of the other machines in the cluster, but only those hosting SQL nodes (dotted box) can be permitted to have any contact with the outside, while those in the zone containing the data nodes and management nodes (solid box) must be isolated from any machines that are not part of the cluster. Applications using the cluster and user of those applications must *not* be permitted to have direct access to the management and data node hosts.

To accomplish this, you must set up software firewalls that limit the traffic to the type or types shown in the following table, according to the type of node that is running on each cluster host computer:

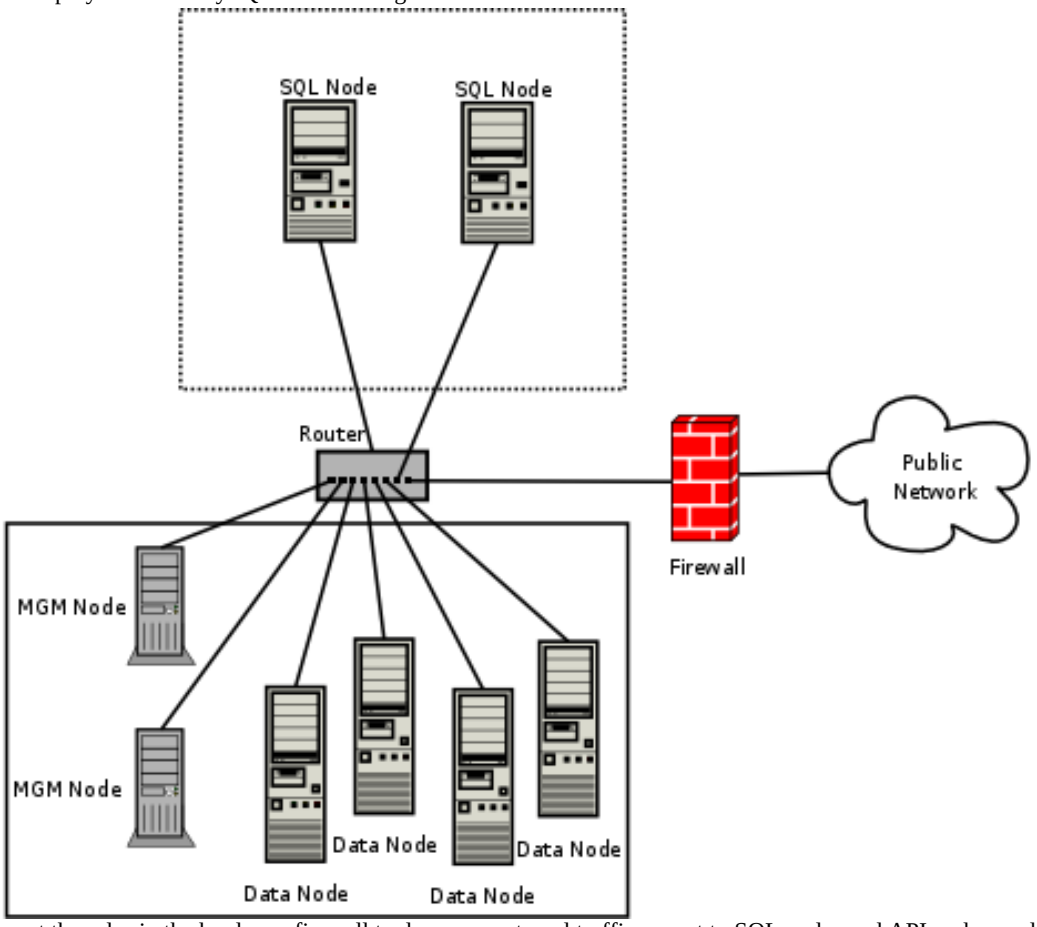
Type of Node to be Accessed	Traffic to Allow
SQL or API node	• It originates from the IP address of a management or data node (using any TCP or UDP port). • It originates from within the network in which the cluster resides and is on the port that your application is using.
Data node or Management node	• It originates from the IP address of a management or data node (using any TCP or UDP port). • It originates from the IP address of an SQL or API node.

Any traffic other than that shown in the table for a given node type should be denied.

The specifics of configuring a firewall vary from firewall application to firewall application, and are beyond the scope of this Manual. `iptables` is a very common and reliable firewall application, which is often used with `APF` as a front end to make configuration easier. You can (and should) consult the documentation for the software firewall that you employ, should you choose to implement a MySQL Cluster network setup of this type, or of a “mixed” type as discussed under the next item.

3. It is also possible to employ a combination of the first two methods, using both hardware and software to secure the cluster — that is, using both network-based and host-based firewalls. This is between the first two schemes in terms of both security level and cost. This type of network setup keeps the cluster behind the hardware firewall, but allows incoming packets to travel beyond the router connecting all cluster hosts in order to reach the SQL nodes.

   One possible network deployment of a MySQL Cluster using hardware and software firewalls in combination is shown here:



   In this case, you can set the rules in the hardware firewall to deny any external traffic except to SQL nodes and API nodes, and then allow traffic to them only on the ports required by your application.

Whatever network configuration you use, remember that your objective from the viewpoint of keeping the cluster secure remains the same — to prevent any unessential traffic from reaching the cluster while ensuring the most efficient communication between the nodes in the cluster.

Because MySQL Cluster requires large numbers of ports to be open for communications between nodes, the recommended option is to use a segregated network. This represents the simplest way to prevent unwanted traffic from reaching the cluster.

> **Note**
>
> If you wish to administer a MySQL Cluster remotely (that is, from outside the local network), the recommended way to do this is to use `ssh` or another secure login shell to access an SQL node host. From this host, you can then run the management client to access the management server safely, from within the Cluster's own local network.
>
> Even though it is possible to do so in theory, it is *not* recommended to use `ndb_mgm` to manage a Cluster directly from outside the local network on which the Cluster is running. Since neither authentication nor encryption takes place between the management client and the management server, this represents an extremely insecure means of managing the cluster, and is almost certain to be compromised sooner or later.

8.2. MySQL Cluster and MySQL Privileges

In this section, we discuss how the MySQL privilege system works in relation to MySQL Cluster and the implications of this for keeping a MySQL Cluster secure.

Standard MySQL privileges apply to MySQL Cluster tables. This includes all MySQL privilege types (`SELECT` privilege, `UPDATE` privilege, `DELETE` privilege, and so on) granted on the database, table, and column level. As with any other MySQL Server,

user and privilege information is stored in the `mysql` system database. The SQL statements used to grant and revoke privileges on `NDB` tables, databases containing such tables, and columns within such tables are identical in all respects with the `GRANT` and `REVOKE` statements used in connection with database objects involving any (other) MySQL storage engine. The same thing is true with respect to the `CREATE USER` and `DROP USER` statements.

It is important to keep in mind that the MySQL grant tables use the `MyISAM` storage engine. Because of this, those tables are not duplicated or shared among MySQL servers acting as SQL nodes in a MySQL Cluster. By way of example, suppose that two SQL nodes **A** and **B** are connected to the same MySQL Cluster, which has an `NDB` table named `mytable` in a database named `mydb`, and that you execute an SQL statement on server **A** that creates a new user `jon@localhost` and grants this user the `SELECT` privilege on that table:

```
mysql> GRANT SELECT ON mydb.mytable
    ->   TO jon@localhost IDENTIFIED BY 'mypass';
```

This user is *not* created on server **B**. In order for this to take place, the statement must also be run on server **B**. Similarly, statements run on server **A** and affecting the privileges of existing users on server **A** do not affect users on server **B** unless those statements are actually run on server **B** as well.

In other words, *changes in users and their privileges do not automatically propagate between SQL nodes*. Synchronization of privileges between SQL nodes must be done either manually or by scripting an application that periodically synchronizes the privilege tables on all SQL nodes in the cluster.

Conversely, because there is no way in MySQL to deny privileges (privileges can either be revoked or not granted in the first place, but not denied as such), there is no special protection for `NDB` tables on one SQL node from users that have privileges on another SQL node. The most far-reaching example of this is the MySQL `root` account, which can perform any action on any database object. In combination with empty `[mysqld]` or `[api]` sections of the `config.ini` file, this account can be especially dangerous. To understand why, consider the following scenario:

- The `config.ini` file contains at least one empty `[mysqld]` or `[api]` section. This means that the Cluster management server performs no checking of the host from which a MySQL Server (or other API node) accesses the MySQL Cluster.
- There is no firewall, or the firewall fails to protect against access to the Cluster from hosts external to the network.
- The host name or IP address of the Cluster's management server is known or can be determined from outside the network.

If these conditions are true, then anyone, anywhere can start a MySQL Server with `--ndbcluster --ndb-connectstring=`*`management_host`* and access the Cluster. Using the MySQL `root` account, this person can then perform the following actions:

- Execute a `SHOW DATABASES` statement to obtain a list of all databases that exist in the cluster
- Execute a `SHOW TABLES FROM `*`some_database`* statement to obtain a list of all `NDB` tables in a given database
- Run any legal MySQL statements on any of those tables, such as:
  - `SELECT * FROM `*`some_table`* to read all the data from any table
  - `DELETE FROM `*`some_table`* to delete all the data from a table
  - `DESCRIBE `*`some_table`* or `SHOW CREATE TABLE `*`some_table`* to determine the table schema
  - `UPDATE `*`some_table`*` SET `*`column1 = any_value1`* to fill a table column with “garbage” data; this could actually cause much greater damage than simply deleting all the data

    Even more insidious variations might include statements like these:

    ```
    UPDATE some_table SET an_int_column = an_int_column + 1
    ```

    or

    ```
    UPDATE some_table SET a_varchar_column = REVERSE(a_varchar_column)
    ```

    Such malicious statements are limited only by the imagination of the attacker.

  The only tables that would be safe from this sort of mayhem would be those tables that were created using storage engines other than `NDB`, and so not visible to a “rogue” SQL node.

> **Note**
>
> A user who can log in as `root` can also access the `INFORMATION_SCHEMA` database and its tables, and so obtain information about databases, tables, stored routines, scheduled events, and any other database objects for which metadata is stored in `INFORMATION_SCHEMA`.

It is also a very good idea to use different passwords for the `root` accounts on different cluster SQL nodes.

In sum, you cannot have a safe MySQL Cluster if it is directly accessible from outside your local network.

> **Important**
>
> *Never leave the MySQL root account password empty*. This is just as true when running MySQL as a MySQL Cluster SQL node as it is when running it as a standalone (non-Cluster) MySQL Server, and should be done as part of the MySQL installation process before configuring the MySQL Server as an SQL node in a MySQL Cluster.

You should never convert the system tables in the `mysql` database to use the `NDB` storage engine. There are a number of reasons why you should not do this, but the most important reason is this: *Many of the SQL statements that affect `mysql` tables storing information about user privileges, stored routines, scheduled events, and other database objects cease to function if these tables are changed to use any storage engine other than `MyISAM`*. This is a consequence of various MySQL Server internals which are not expected to change in the foreseeable future.

If you need to synchronize `mysql` system tables between SQL nodes, you can use standard MySQL replication to do so, or employ a script to copy table entries between the MySQL servers.

Summary. The two most important points to remember regarding the MySQL privilege system with regard to MySQL Cluster are:

1. Users and privileges established on one SQL node do not automatically exist or take effect on other SQL nodes in the cluster.

   Conversely, removing a user or privilege on one SQL node in the cluster does not remove the user or privilege from any other SQL nodes.

2. Once a MySQL user is granted privileges on an `NDB` table from one SQL node in a MySQL Cluster, that user can “see” any data in that table regardless of the SQL node from which the data originated.

8.3. MySQL Cluster and MySQL Security Procedures

In this section, we discuss MySQL standard security procedures as they apply to running MySQL Cluster.

In general, any standard procedure for running MySQL securely also applies to running a MySQL Server as part of a MySQL Cluster. First and foremost, you should always run a MySQL Server as the `mysql` system user; this is no different from running MySQL in a standard (non-Cluster) environment. The `mysql` system account should be uniquely and clearly defined. Fortunately, this is the default behavior for a new MySQL installation. You can verify that the `mysqld` process is running as the system user `mysql` by using the system command such as the one shown here:

```
shell> ps aux | grep mysql
root     10467  0.0  0.1   3616  1380 pts/3    S    11:53   0:00 \
  /bin/sh ./mysqld_safe --ndbcluster --ndb-connectstring=localhost:1186
mysql    10512  0.2  2.5  58528 26636 pts/3    Sl   11:53   0:00 \
  /usr/local/mysql/libexec/mysqld --basedir=/usr/local/mysql \
  --datadir=/usr/local/mysql/var --user=mysql --ndbcluster \
  --ndb-connectstring=localhost:1186 --pid-file=/usr/local/mysql/var/mothra.pid \
  --log-error=/usr/local/mysql/var/mothra.err
jon      10579  0.0  0.0   2736   688 pts/0    S+   11:54   0:00 grep mysql
```

If the `mysqld` process is running as any other user than `mysql`, you should immediately shut it down and restart it as the `mysql` user. If this user does not exist on the system, the `mysql` user account should be created, and this user should be part of the `mysql` user group; in this case, you should also make sure that the MySQL `DataDir` on this system is owned by the `mysql` user, and that the SQL node's `my.cnf` file includes `user=mysql` in the `[mysqld]` section. Alternatively, you can start the server with `--user=mysql` on the command line, but it is preferable to use the `my.cnf` option, since you might forget to use the command-line option and so have `mysqld` running as another user unintentionally. The `mysqld_safe` startup script forces MySQL to run as the `mysql` user.

> **Important**
>
> Never run `mysqld` as the system root user. Doing so means that potentially any file on the system can be read by MySQL, and thus — should MySQL be compromised — by an attacker.

As mentioned in the previous section (see Section 8.2, “MySQL Cluster and MySQL Privileges”), you should always set a root password for the MySQL Server as soon as you have it running. You should also delete the anonymous user account that is installed by default. You can accomplish these tasks via the following statements:

```
shell> mysql -u root
mysql> UPDATE mysql.user
    ->     SET Password=PASSWORD('secure_password')
```

```
    ->     WHERE User='root';
mysql> DELETE FROM mysql.user
    ->     WHERE User='';
mysql> FLUSH PRIVILEGES;
```

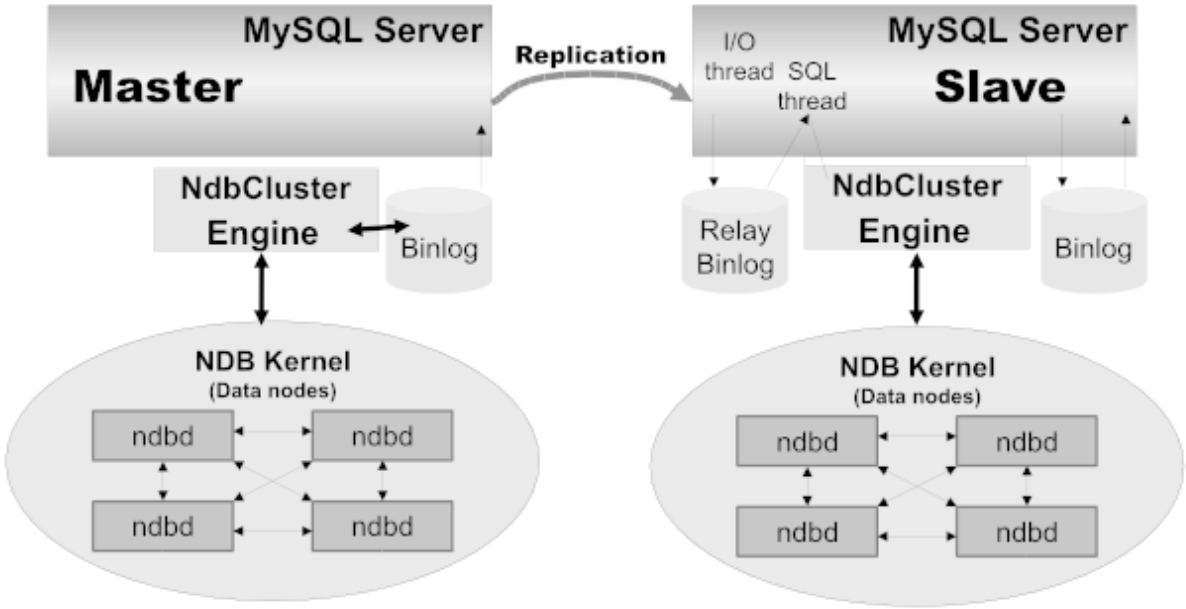
Be very careful when executing the `DELETE` statement not to omit the `WHERE` clause, or you risk deleting *all* MySQL users. Be sure to run the `FLUSH PRIVILEGES` statement as soon as you have modified the `mysql.user` table, so that the changes take immediate effect. Without `FLUSH PRIVILEGES`, the changes do not take effect until the next time that the server is restarted.

> **Note**
>
> Many of the MySQL Cluster utilities such as `ndb_show_tables`, `ndb_desc`, and `ndb_select_all` also work without authentication and can reveal table names, schemas, and data. By default these are installed on Unix-style systems with the permissions `wxr-xr-x` (755), which means they can be executed by any user that can access the `mysql/bin` directory.
>
> See Chapter 6, *MySQL Cluster Programs*, for more information about these utilities.

Chapter 9. MySQL Cluster Replication

Previous to MySQL 5.1.6, *asynchronous replication*, more usually referred to simply as “replication”, was not available when using MySQL Cluster. MySQL 5.1.6 introduces master-slave replication of this type for MySQL Cluster databases. This section explains how to set up and manage a configuration wherein one group of computers operating as a MySQL Cluster replicates to a second computer or group of computers. We assume some familiarity on the part of the reader with standard MySQL replication as discussed elsewhere in this Manual. (See Replication).

Normal (non-clustered) replication involves a “master” server and a “slave” server, the master being the source of the operations and data to be replicated and the slave being the recipient of these. In MySQL Cluster, replication is conceptually very similar but can be more complex in practice, as it may be extended to cover a number of different configurations including replicating between two complete clusters. Although a MySQL Cluster itself depends on the `NDBCLUSTER` storage engine for clustering functionality, it is not necessary to use the Cluster storage engine on the slave. However, for maximum availability, it is possible to replicate from one MySQL Cluster to another, and it is this type of configuration that we discuss, as shown in the following figure:



In this scenario, the replication process is one in which successive states of a master cluster are logged and saved to a slave cluster. This process is accomplished by a special thread known as the NDB binlog injector thread, which runs on each MySQL server and produces a binary log (`binlog`). This thread ensures that all changes in the cluster producing the binary log — and not just those changes that are effected via the MySQL Server — are inserted into the binary log with the correct serialization order. We refer to the MySQL replication master and replication slave servers as replication servers or replication nodes, and the data flow or line of communication between them as a *replication channel*.

9.1. MySQL Cluster Replication — Abbreviations and Symbols

Throughout this section, we use the following abbreviations or symbols for referring to the master and slave clusters, and to processes and commands run on the clusters or cluster nodes:

Symbol or Abbreviation	Description (Refers to...)
M	The cluster serving as the (primary) replication master
S	The cluster acting as the (primary) replication slave
shell*M*>	Shell command to be issued on the master cluster
mysql*M*>	MySQL client command issued on a single MySQL server running as an SQL node on the master cluster
mysql*M**>	MySQL client command to be issued on all SQL nodes participating in the replication master cluster
shell*S*>	Shell command to be issued on the slave cluster
mysql*S*>	MySQL client command issued on a single MySQL server running as an SQL node on the slave

	cluster
`mysqlS*>`	MySQL client command to be issued on all SQL nodes participating in the replication slave cluster
`C`	Primary replication channel
`C'`	Secondary replication channel
`M'`	Secondary replication master
`S'`	Secondary replication slave

9.2. MySQL Cluster Replication — Assumptions and General Requirements

A replication channel requires two MySQL servers acting as replication servers (one each for the master and slave). For example, this means that in the case of a replication setup with two replication channels (to provide an extra channel for redundancy), there will be a total of four replication nodes, two per cluster.

Replication of a MySQL Cluster as described in this section and those following is dependent on row-based replication. This means that the replication master MySQL server must be started with `--binlog-format=ROW` or `--binlog-format=MIXED`, as described in Section 9.6, "Starting MySQL Cluster Replication (Single Replication Channel)". For general information about row-based replication, see Replication Formats.

> **Important**
>
> If you attempt to use MySQL Cluster Replication with `--binlog-format=STATEMENT`, replication fails to work properly because the `ndb_binlog_index` table on the master and the `epoch` column of the `ndb_apply_status` table on the slave are not updated (see Section 9.4, "MySQL Cluster Replication Schema and Tables"). Instead, only updates on the MySQL server acting as the replication master propagate to the slave, and no updates from any other SQL nodes on the master cluster are replicated.
>
> In all MySQL Cluster NDB 6.x releases, the default value for the `--binlog-format` option is `MIXED`.

Each MySQL server used for replication in either cluster must be uniquely identified among all the MySQL replication servers participating in either cluster (you cannot have replication servers on both the master and slave clusters sharing the same ID). This can be done by starting each SQL node using the `--server-id=`*`id`* option, where *`id`* is a unique integer. Although it is not strictly necessary, we will assume for purposes of this discussion that all MySQL installations are the same version.

In any event, both MySQL servers involved in replication must be compatible with one another with respect to both the version of the replication protocol used and the SQL feature sets which they support; the simplest and easiest way to assure that this is the case is to use the same MySQL version for all servers involved. Note that in many cases it is not possible to replicate to a slave running a version of MySQL with a lower version number than that of the master — see Replication Compatibility Between MySQL Versions, for details.

We assume that the slave server or cluster is dedicated to replication of the master, and that no other data is being stored on it.

> **Note**
>
> It is possible to replicate a MySQL Cluster using statement-based replication. However, in this case, the following restrictions apply:
>
> - All updates to data rows on the cluster acting as the master must be directed to a single MySQL server.
> - It is not possible to replicate a cluster using multiple simultaneous MySQL replication processes.
> - Only changes made at the SQL level are replicated.
>
> These are in addition to the other limitations of statement-based replication as opposed to row-based replication; see Comparison of Statement-Based and Row-Based Replication, for more specific information concerning the differences between the two replication formats.

9.3. Known Issues in MySQL Cluster Replication

The following are known problems or issues when using replication with MySQL Cluster in MySQL 5.1:

- **Loss of master-slave connection.** Prior to MySQL 5.1.18, a MySQL Cluster replication slave `mysqld` had no way of detect-

ing that the connection from the master had been interrupted (due to, for instance, the master going down or a network failure). For this reason, it was possible for the slave to become inconsistent with the master.

Beginning with MySQL 5.1.18, the master issues a “gap” event when connecting to the cluster. When the slave encounters a gap in the replication log, it stops with an error message. This message is available in the output of `SHOW SLAVE STATUS`, and indicates that the SQL thread has stopped due to an incident registered in the replication stream, and that manual intervention is required. In order to restart the slave, it is necessary to issue the following commands:

```
SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
START SLAVE;
```

The slave then resumes reading the master binlog from the point where the gap was recorded.

> **Important**
>
> If high availability is a requirement for the slave server or cluster, then it is still advisable to set up multiple replication lines, to monitor the master `mysqld` on the primary replication line, and to fail over to a secondary line if and as necessary. For information about implementing this type of setup, see Section 9.7, “Using Two Replication Channels for MySQL Cluster Replication”, and Section 9.8, “Implementing Failover with MySQL Cluster Replication”.
>
> However, if you are replicating from a standalone MySQL server to a MySQL Cluster, one channel is usually sufficient.

- **Multi-byte character sets.** There are several known issues with regard to the use of multi-byte characters sets with MySQL Cluster Replication. See Bug#27404 (fixed in MySQL 5.1.21), Bug#29562, Bug#29563, and Bug#29564 for more information.

- **Circular replication.** Prior to MySQL 5.1.18, circular replication was not supported with MySQL Cluster replication, due to the fact that all log events created in a particular MySQL Cluster were wrongly tagged with the server ID of the MySQL server used as master and not with the server ID of the originating server.

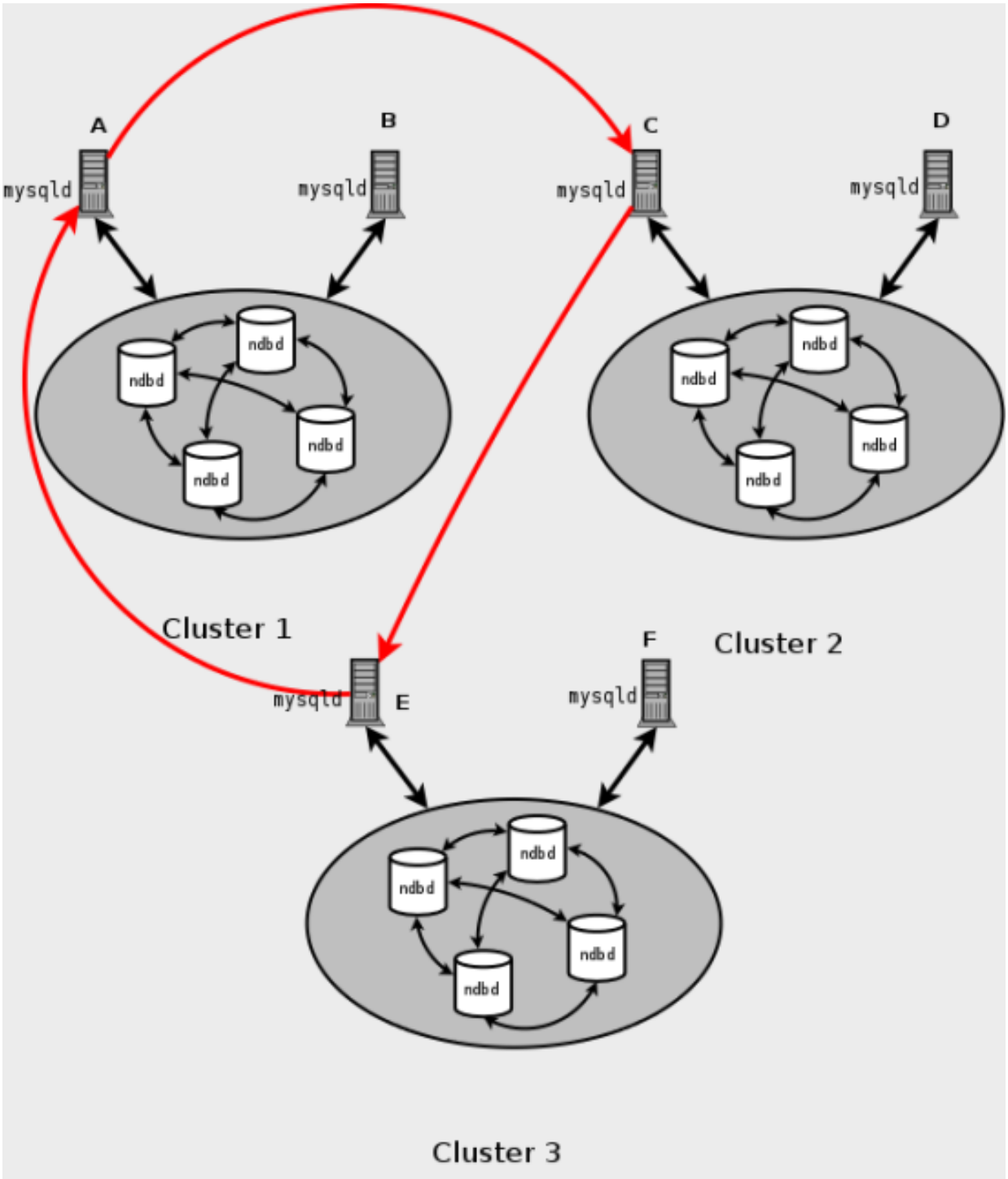
  Beginning with MySQL 5.1.18, this limitation is lifted, as discussed in the next few paragraphs, in which we consider the example of a replication setup involving three MySQL Clusters numbered 1, 2, and 3, in which Cluster 1 acts as the replication master for Cluster 2, Cluster 2 acts as the master for Cluster 3, and Cluster 3 acts as the master for Cluster 1. Each cluster has two SQL nodes, with SQL nodes A and B belonging to Cluster 1, SQL nodes C and D belonging to Cluster 2, and SQL nodes E and F belonging to Cluster 3.

  Circular replication using these clusters is supported as long as:

  - the SQL nodes on all masters and slaves are the same

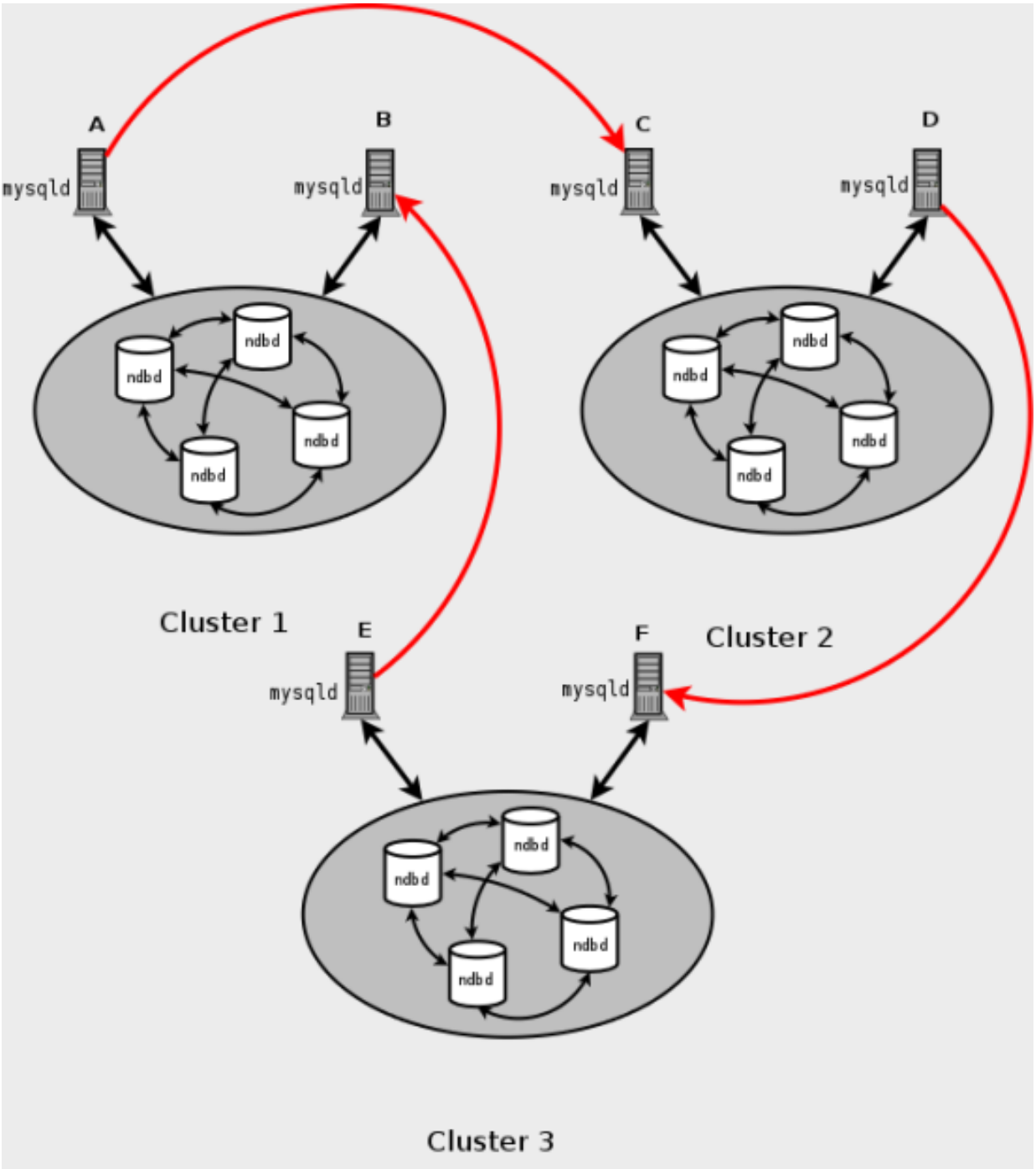
  - All SQL nodes acting as replication masters and slaves are started using the `--log-slave-updates` option

  This type of circular replication setup is shown in the following diagram:



In this scenario, SQL node A in Cluster 1 replicates to SQL node C in Cluster 2; SQL node C replicates to SQL node E in Cluster 3; SQL node E replicates to SQL node A. In other words, the replication line (indicated by the red arrows in the diagram) directly connects all SQL nodes used as replication masters and slaves.

It should also be possible to set up circular replication in which not all master SQL nodes are also slaves, as shown here:



In this case, different SQL nodes in each cluster are used as replication masters and slaves. However, you must *not* start any of the SQL nodes using `--log-slave-updates` (see the description of this option for more information). This type of circular replication scheme for MySQL Cluster, in which the line of replication (again indicated by the red arrows in the diagram) is discontinuous, should be possible, but it should be noted that it has not yet been thoroughly tested and must therefore still be considered experimental.

> **Important**
>
> Beginning with MySQL 5.1.24, you should execute the following statement before starting circular replication:
>
> ```
> mysql> SET GLOBAL slave_exec_mode = 'IDEMPOTENT';
> ```
>
> This is necessary to suppress duplicate-key and other errors that otherwise break circular replication of MySQL Cluster. `IDEMPOTENT` mode is also required for multi-master replication when using MySQL Cluster. (Bug#31609)
>
> See `slave_exec_mode`, for more information.

- **DDL statements.** The use of data definition statements, such as `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`, are recorded in the binary log for only the MySQL server on which they are issued.

- **Cluster replication and primary keys.** In MySQL 5.1.6, only those `NDB` tables having explicit primary keys could be replicated. This limitation was lifted in MySQL 5.1.7. However, in the event of a node failure, errors in replication of `NDB` tables without primary keys can still occur, due to the possibility of duplicate rows being inserted in such cases. For this reason, it is highly recommended that all `NDB` tables being replicated have primary keys.

- **Restarting with `--initial`.** Restarting the cluster with the `--initial` option causes the sequence of GCI and epoch numbers to start over from `0`. (This is generally true of MySQL Cluster and not limited to replication scenarios involving Cluster.) The MySQL servers involved in replication should in this case be restarted. After this, you should use the `RESET MASTER` and `RESET SLAVE` statements to clear the invalid `ndb_binlog_index` and `ndb_apply_status` tables. respectively.

- **`auto_increment_offset` and `auto_increment_increment` variables.** The use of the `auto_increment_offset` and `auto_increment_increment` server system variables is supported beginning with MySQL 5.1.20. Previously, these produced unpredictable results when used with `NDB` tables or MySQL Cluster replication.

- **Replication from `NDBCLUSTER` to other storage engines.** If you attempt to replicate from a MySQL Cluster to a slave that uses a storage engine that does not handle its own binary logging, the replication process aborts with the error `BINARY LOGGING NOT POSSIBLE ... STATEMENT CANNOT BE WRITTEN ATOMICALLY SINCE MORE THAN ONE ENGINE INVOLVED AND AT LEAST ONE ENGINE IS SELF-LOGGING` (Error `1595`). It is possible to work around this issue in one of the following ways:

  - **Turn off binary logging on the slave.** This can be accomplished by setting `sql_log_bin = 0`.

  - **Change the storage engine used for the `mysql.ndb_apply_status` table.** Causing this table to use an engine that does not handle its own binary logging can also eliminate the conflict. This can be done by issuing a statement such as `ALTER TABLE mysql.ndb_apply_status ENGINE=MyISAM` on the slave. It is safe to do this when using a non-`NDB` storage engine on the slave, since you do not then need to worry about keeping multiple slave SQL nodes synchronized.

  - **Filter out changes to the `mysql.ndb_apply_status` table on the slave.** This can be done by starting the slave SQL node with the option `--replicate-ignore-table=mysql.ndb_apply_status`. If you need for other tables to be ignored by replication, you might wish to use an appropriate `--replicate-wild-ignore-table` option instead.

  > **Important**
  >
  > You should *not* disable replication or binary logging of `mysql.ndb_apply_status` or change the storage engine used for this table when replicating from one MySQL Cluster to another. See *Replication and binary log filtering rules with replication between MySQL Clusters* elsewhere in this section for details.

  When replicating from `NDBCLUSTER` to a non-transactional storage engine such as `MyISAM`, you may encounter unnecessary duplicate key errors when replicating `INSERT ... ON DUPLICATE KEY UPDATE` statements. You can suppress these in MySQL Cluster NDB 6.2 by using `--ndb-log-update-as-write=0`, which forces all columns from updated rows to be sent (and not just those that were updated). For MySQL Cluster NDB 6.3.3 and later, there are additional way to determine whether or not an update to the row on the master should be applied on the slave `mysqld`; see Section 9.11, “MySQL Cluster Replication Conflict Resolution”, for more information about these methods.

- **Replication and binary log filtering rules with replication between MySQL Clusters.** If you are using any of the options `--replicate-do-*`, `--replicate-ignore-*`, `--binlog-do-db`, or `--binlog-ignore-db` to filter databases or tables being replicated, care must be taken not to block replication or binary logging of the `mysql.ndb_apply_status`, which is required for replication between MySQL Clusters to operate properly. In particular, you must keep in mind the following:

  1. Using `--replicate-do-db=`*`db_name`* (and no other `--replicate-do-*` or `--replicate-ignore-*` options) means that *only* tables in database *`db_name`* are replicated. In this case, you should also use `--replicate-do-db=mysql`, `--binlog-do-db=mysql`, or `--replicate-do-table=mysql.ndb_apply_status` to insure that `mysql.ndb_apply_status` is populated on slaves.

     Using `--binlog-do-db=`*`db_name`* (and no other `--binlog-do-db` options) means that changes *only* to tables in database *`db_name`* are written to the binary log. In this case, you should also use `--replicate-do-db=mysql`, `--binlog-do-db=mysql`, or `--replicate-do-table=mysql.ndb_apply_status` to insure that `mysql.ndb_apply_status` is populated on slaves.

  2. Using `--replicate-ignore-db=mysql` means that no tables in the `mysql` database are replicated. In this case, you should also use `--replicate-do-table=mysql.ndb_apply_status` to insure that `mysql.ndb_apply_status` is replicated.

     Using `--binlog-ignore-db=mysql` means that no changes to tables in the `mysql` database are written to the binary

log. In this case, you should also use `--replicate-do-table=mysql.ndb_apply_status` to insure that `mysql.ndb_apply_status` is replicated.

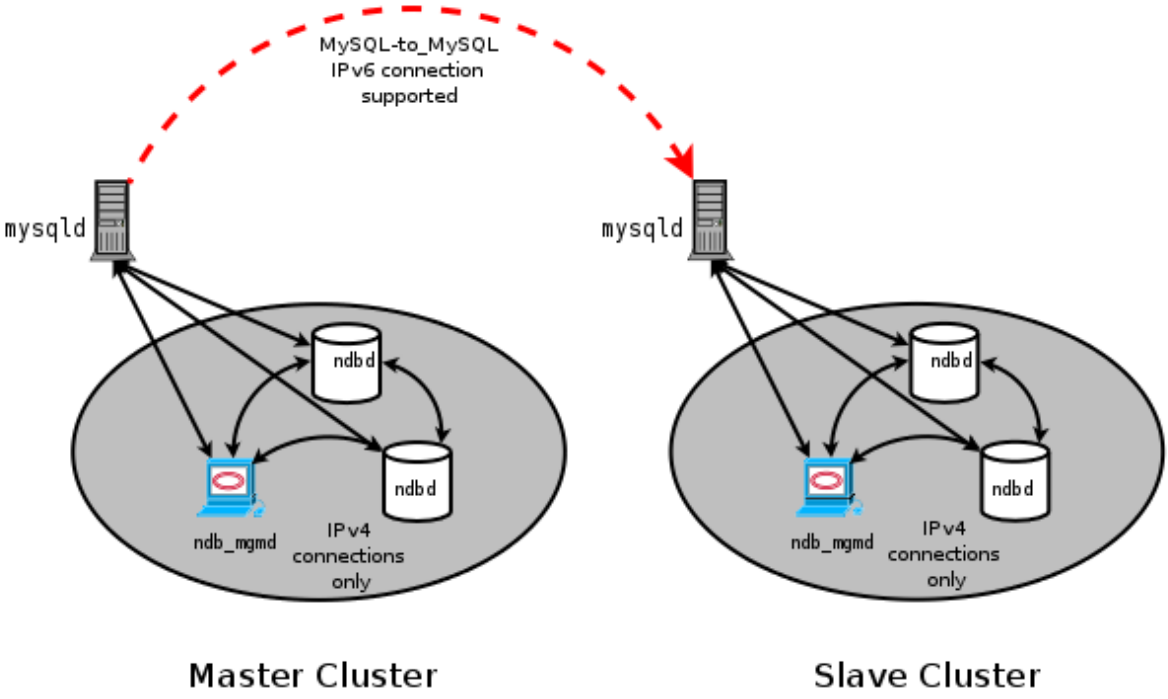
You should also remember that:

1. Each replication filtering rule requires its own `--replicate-do-*` or `--replicate-ignore-*` option, and that multiple rules cannot be expressed in a single replication filtering option. For information about these rules, see Replication and Binary Logging Options and Variables.

2. Each binary log filtering rule requires its own `--binlog-do-db` or `--binlog-ignore-db` option, and that multiple rules cannot be expressed in a single binary log filtering option. For information about these rules, see The Binary Log.

> **Note**
>
> If you are replicating a MySQL Cluster to a slave that uses a storage engine other than `NDBCLUSTER`, the considerations just given previously may not apply. See *Replication from `NDBCLUSTER` to other storage engines* elsewhere in this section for details.

- **MySQL Cluster Replication and IPv6.** Currently, the NDB API and MGM API do not support IPv6. However, beginning with MySQL Cluster NDB 6.4.1, MySQL Servers — including those acting as SQL nodes in a MySQL Cluster — can use IPv6 to contact other MySQL Servers. This means that you can replicate between MySQL Clusters using IPv6 to connect the master and slave SQL nodes as shown by the dotted arrow in the following diagram:



However, all connections originating *within* the MySQL Cluster — shown in the diagram by solid arrows — must use IPv4.

All MySQL Cluster data nodes, management servers, and management clients must be accessible from one another using IPv4. In addition, SQL nodes must use IPv4 to communicate with the cluster. There is not currently any support in the NDB and MGM APIs for IPv6, which means that any applications written using these APIs must also make all connections using IPv4.

9.4. MySQL Cluster Replication Schema and Tables

Replication in MySQL Cluster makes use of a number of dedicated tables in the `mysql` database on each MySQL Server instance acting as an SQL node in both the cluster being replicated and the replication slave (whether the slave is a single server or a cluster). These tables are created during the MySQL installation process by the `mysql_install_db` script, and include a table for storing the binary log's indexing data. Since the `ndb_binlog_index` table is local to each MySQL server and does not participate in clustering, it uses the `MyISAM` storage engine. This means that it must be created separately on each `mysqld` participating in the master cluster. (However, the binlog itself contains updates from all MySQL servers in the cluster to be replicated.) This table is defined as follows:

```
CREATE TABLE `ndb_binlog_index` (
    `Position`  BIGINT(20) UNSIGNED NOT NULL,
    `File`      VARCHAR(255) NOT NULL,
    `epoch`     BIGINT(20) UNSIGNED NOT NULL,
    `inserts`   BIGINT(20) UNSIGNED NOT NULL,
    `updates`   BIGINT(20) UNSIGNED NOT NULL,
    `deletes`   BIGINT(20) UNSIGNED NOT NULL,
    `schemaops` BIGINT(20) UNSIGNED NOT NULL,
    PRIMARY KEY (`epoch`)
) ENGINE=MYISAM  DEFAULT CHARSET=latin1;
```

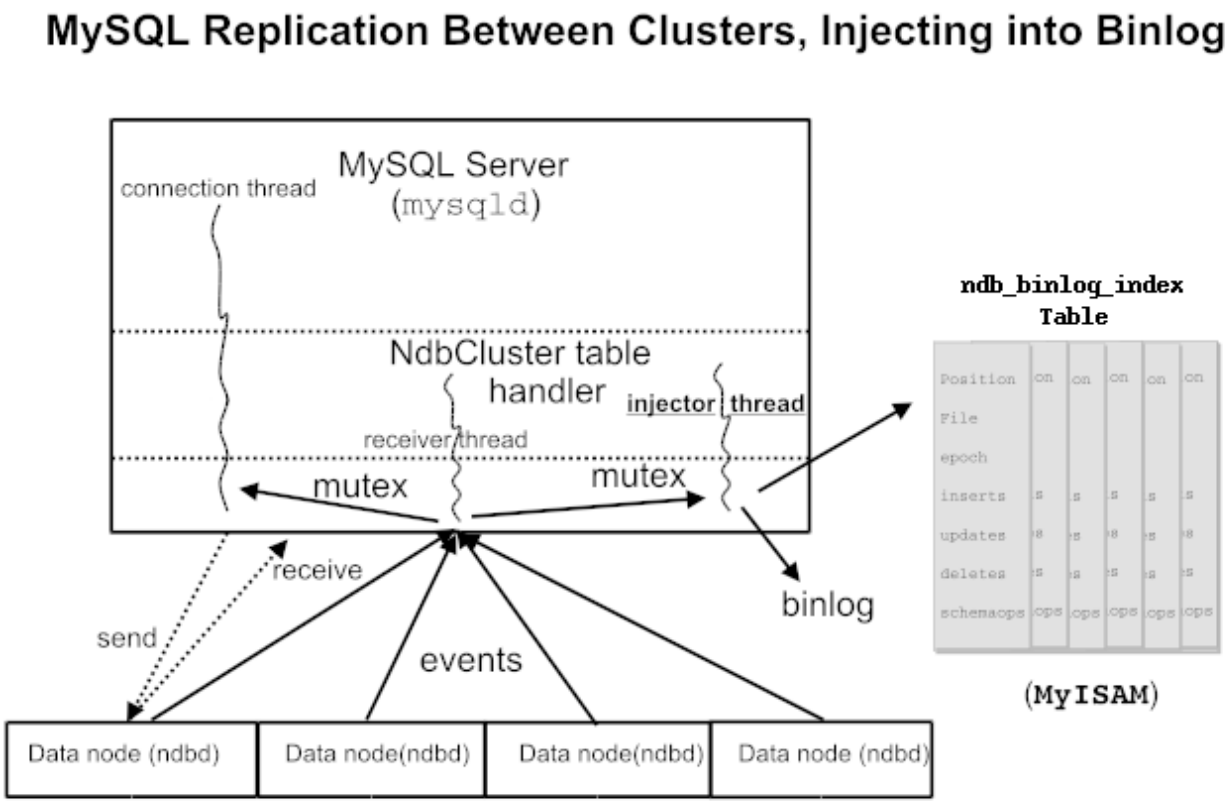
> **Important**
>
> Prior to MySQL 5.1.14, the `ndb_binlog_index` table was known as `binlog_index`, and was kept in a separate `cluster` database, which in MySQL 5.1.7 and earlier was known as the `cluster_replication` database. Similarly, the `ndb_apply_status` and `ndb_schema` tables were known as `apply_status` and `schema`, and were also found in the `cluster` (earlier `cluster_replication`) database. However, beginning with MySQL 5.1.14, all MySQL Cluster replication tables reside in the `mysql` system database.
>
> Information about how this change affects upgrades from MySQL Cluster 5.1.13 and earlier to 5.1.14 and later versions can be found in Changes in MySQL 5.1.14.

Beginning with MySQL Cluster NDB 6.3.2, this table has been changed to facilitate 3-way replication recovery. Two columns `orig_server_id` and `orig_epoch` have been added to this table; when `mysqld` is started with the `--ndb-log-orig` option, these columns store, respectively, the ID of the server on which the event originated and the epoch in which the event took place on the originating server. In addition, the table's primary key now includes these two columns. The modified table definition is shown here:

```
CREATE TABLE `ndb_binlog_index` (
    `Position` BIGINT(20) UNSIGNED NOT NULL,
    `File` VARCHAR(255) NOT NULL,
    `epoch` BIGINT(20) UNSIGNED NOT NULL,
    `inserts` INT(10) UNSIGNED NOT NULL,
    `updates` INT(10) UNSIGNED NOT NULL,
    `deletes` INT(10) UNSIGNED NOT NULL,
    `schemaops` INT(10) UNSIGNED NOT NULL,
    `orig_server_id` INT(10) UNSIGNED NOT NULL,
    `orig_epoch` BIGINT(20) UNSIGNED NOT NULL,
    `gci` INT(10) UNSIGNED NOT NULL,
    PRIMARY KEY (`epoch`,`orig_server_id`,`orig_epoch`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

The `gci` column was added in MySQL Cluster NDB 6.2.6 and MySQL Cluster NDB 6.3.2.

The following figure shows the relationship of the MySQL Cluster replication master server, its binlog injector thread, and the `mysql.ndb_binlog_index` table.



An additional table, named `ndb_apply_status`, is used to keep a record of the operations that have been replicated from the master to the slave. Unlike the case with `ndb_binlog_index`, the data in this table is not specific to any one SQL node in the (slave) cluster, and so `ndb_apply_status` can use the `NDB Cluster` storage engine, as shown here:

```
CREATE TABLE `ndb_apply_status` (
    `server_id`   INT(10) UNSIGNED NOT NULL,
    `epoch`       BIGINT(20) UNSIGNED NOT NULL,
    `log_name`    VARCHAR(255) CHARACTER SET latin1 COLLATE latin1_bin NOT NULL,
    `start_pos`   BIGINT(20) UNSIGNED NOT NULL,
    `end_pos`     BIGINT(20) UNSIGNED NOT NULL,
    PRIMARY KEY (`server_id`) USING HASH
) ENGINE=NDBCLUSTER   DEFAULT CHARSET=latin1;
```

This table is populated only on slaves; on the master, no `DataMemory` is allocated to it. However, the table is populated *from* the master. For this reason, this table must be replicated and any replication filtering or binary log filtering rules that prevent this prevent replication between clusters from operating properly. For more information about potential problems arising from such filtering rules, see Section 9.3, “Known Issues in MySQL Cluster Replication”.

The `log_name`, `start_pos`, and `end_pos` columns were added in MySQL 5.1.18.

> **Important**
>
> If you are using MySQL Cluster replication, see Section 5.2, “MySQL Cluster 5.1 and MySQL Cluster NDB 6.x/7.x Upgrade and Downgrade Compatibility” before upgrading to MySQL 5.1.18 or later from an earlier version.

The `ndb_binlog_index` and `ndb_apply_status` tables are created in the `mysql` database because they should not be replicated. No user intervention is normally required to create or maintain either of them. Both the `ndb_binlog_index` and the `ndb_apply_status` tables are maintained by the `NDB` injector thread. This keeps the master `mysqld` process updated to changes performed by the `NDB` storage engine. The `NDB` *binlog injector thread* receives events directly from the `NDB` storage engine. The `NDB` injector is responsible for capturing all the data events within the cluster, and ensures that all events which change, insert, or delete data are recorded in the `ndb_binlog_index` table. The slave I/O thread transfers the events from the master's binary log to the slave's relay log.

However, it is advisable to check for the existence and integrity of these tables as an initial step in preparing a MySQL Cluster for replication. It is possible to view event data recorded in the binary log by querying the `mysql.ndb_binlog_index` table dir-

ectly on the master. This can be also be accomplished using the `SHOW BINLOG EVENTS` statement on either the replication master or slave MySQL servers. (See `SHOW BINLOG EVENTS` Syntax.)

You can also obtain useful information from the output of `SHOW ENGINE NDB STATUS`.

The `ndb_schema` table is used to track schema changes made to `NDB` tables. It is defined as shown here:

```
CREATE TABLE ndb_schema (
    `db` VARBINARY(63) NOT NULL,
    `name` VARBINARY(63) NOT NULL,
    `slock` BINARY(32) NOT NULL,
    `query` BLOB NOT NULL,
    `node_id` INT UNSIGNED NOT NULL,
    `epoch` BIGINT UNSIGNED NOT NULL,
    `id` INT UNSIGNED NOT NULL,
    `version` INT UNSIGNED NOT NULL,
    `type` INT UNSIGNED NOT NULL,
    PRIMARY KEY USING HASH (db,name)
) ENGINE=NDB   DEFAULT CHARSET=latin1;
```

Unlike the two tables previously mentioned in this section, the `ndb_schema` table is not visible either to MySQL `SHOW` statements, or in any `INFORMATION_SCHEMA` tables; however, it can be seen in the output of `ndb_show_tables`, as shown here:

```
shell> ndb_show_tables -t 2
id    type               state    logging database     schema   name
4     UserTable          Online   Yes     mysql        def      ndb_apply_status
5     UserTable          Online   Yes     ndbworld     def      City
6     UserTable          Online   Yes     ndbworld     def      Country
3     UserTable          Online   Yes     mysql        def      NDB$BLOB_2_3
7     UserTable          Online   Yes     ndbworld     def      CountryLanguage
2     UserTable          Online   Yes     mysql        def      ndb_schema
NDBT_ProgramExit: 0 - OK
```

It is also possible to `SELECT` from this table in `mysql` and other MySQL client applications, as shown here:

```
mysql> SELECT * FROM mysql.ndb_schema WHERE name='City' \G
*************************** 1. row ***************************
     db: ndbworld
   name: City
  slock:
  query: alter table City engine=ndb
node_id: 4
  epoch: 0
     id: 0
version: 0
   type: 7
1 row in set (0.00 sec)
```

This can sometimes be useful when debugging applications.

> **Note**
>
> When performing schema changes on `NDB` tables, applications should wait until the `ALTER TABLE` statement has returned in the MySQL client connection that issued the statement before attempting to use the updated definition of the table.

The `ndb_schema` table was added in MySQL 5.1.8.

Beginning with MySQL 5.1.14, if either of the `ndb_apply_status` or `ndb_schema` tables does not exist on the slave, it is created by `ndb_restore`. (Bug#14612)

Conflict resolution for MySQL Cluster Replication requires the presence of an additional `mysql.ndb_replication` table. Currently, this table must be created manually. For details, see Section 9.11, “MySQL Cluster Replication Conflict Resolution”.

9.5. Preparing the MySQL Cluster for Replication

Preparing the MySQL Cluster for replication consists of the following steps:

1. Check all MySQL servers for version compatibility (see Section 9.2, “MySQL Cluster Replication — Assumptions and General Requirements”).

2. Create a slave account on the master Cluster with the appropriate privileges:

   ```
   mysqlM> GRANT REPLICATION SLAVE
        -> ON *.* TO 'slave_user'@'slave_host'
        -> IDENTIFIED BY 'slave_password';
   ```

   In the previous statement, *`slave_user`* is the slave account user name, *`slave_host`* is the host name or IP address of the replication slave, and *`slave_password`* is the password to assign to this account.

For example, to create a slave user account with the name "myslave," logging in from the host named "rep-slave," and using the password "53cr37," use the following GRANT statement:

```
mysqlM> GRANT REPLICATION SLAVE
     -> ON *.* TO 'myslave'@'rep-slave'
     -> IDENTIFIED BY '53cr37';
```

For security reasons, it is preferable to use a unique user account — not employed for any other purpose — for the replication slave account.

3. Configure the slave to use the master. Using the MySQL Monitor, this can be accomplished with the CHANGE MASTER TO statement:

```
mysqlS> CHANGE MASTER TO
     -> MASTER_HOST='master_host',
     -> MASTER_PORT=master_port,
     -> MASTER_USER='slave_user',
     -> MASTER_PASSWORD='slave_password';
```

In the previous statement, *master_host* is the host name or IP address of the replication master, *master_port* is the port for the slave to use for connecting to the master, *slave_user* is the user name set up for the slave on the master, and *slave_password* is the password set for that user account in the previous step.

For example, to tell the slave to replicate from the MySQL server whose host name is "rep-master," using the replication slave account created in the previous step, use the following statement:

```
mysqlS> CHANGE MASTER TO
     -> MASTER_HOST='rep-master'
     -> MASTER_PORT=3306,
     -> MASTER_USER='myslave'
     -> MASTER_PASSWORD='53cr37';
```

For a complete list of clauses that can be used with this statement, see CHANGE MASTER TO Syntax.

You can also configure the slave to use the master by setting the corresponding startup options in the slave server's my.cnf file. To configure the slave in the same way as the preceding example CHANGE MASTER TO statement, the following information would need to be included in the slave's my.cnf file:

```
[mysqld]
master-host=rep-master
master-port=3306
master-user=myslave
master-password=53cr37
```

For additional options that can be set in my.cnf for replication slaves, see Replication and Binary Logging Options and Variables.

> **Note**
>
> To provide replication backup capability, you will also need to add an ndb-connectstring option to the slave's my.cnf file prior to starting the replication process. See Section 9.9, "MySQL Cluster Backups With MySQL Cluster Replication", for details.

4. If the master cluster is already in use, you can create a backup of the master and load this onto the slave to cut down on the amount of time required for the slave to synchronize itself with the master. If the slave is also running MySQL Cluster, this can be accomplished using the backup and restore procedure described in Section 9.9, "MySQL Cluster Backups With MySQL Cluster Replication".

```
ndb-connectstring=management_host[:port]
```

In the event that you are *not* using MySQL Cluster on the replication slave, you can create a backup with this command on the replication master:

```
shellM> mysqldump --master-data=1
```

Then import the resulting data dump onto the slave by copying the dump file over to the slave. After this, you can use the mysql client to import the data from the dumpfile into the slave database as shown here, where *dump_file* is the name of the file that was generated using mysqldump on the master, and *db_name* is the name of the database to be replicated:

```
shellS> mysql -u root -p db_name < dump_file
```

For a complete list of options to use with `mysqldump`, see `mysqldump`.

> **Note**
>
> If you copy the data to the slave in this fashion, you should make sure that the slave is started with the `--skip-slave-start` option on the command line, or else include `skip-slave-start` in the slave's `my.cnf` file to keep it from trying to connect to the master to begin replicating before all the data has been loaded. Once the data loading has completed, follow the additional steps outlined in the next two sections.

5. Ensure that each MySQL server acting as a replication master is configured with a unique server ID, and with binary logging enabled, using the row format. (See Replication Formats.) These options can be set either in the master server's `my.cnf` file, or on the command line when starting the master `mysqld` process. See Section 9.6, "Starting MySQL Cluster Replication (Single Replication Channel)", for information regarding the latter option.

9.6. Starting MySQL Cluster Replication (Single Replication Channel)

This section outlines the procedure for starting MySQL CLuster replication using a single replication channel.

1. Start the MySQL replication master server by issuing this command:

```
shellM> mysqld --ndbcluster --server-id=id \
        --log-bin --binlog-format=ROW &
```

In the previous statement, *`id`* is this server's unique ID (see Section 9.2, "MySQL Cluster Replication — Assumptions and General Requirements"). This starts the server's `mysqld` process with binary logging enabled using the proper logging format.

> **Note**
>
> You can also start the master with `--binlog-format=MIXED`, in which case row-based replication is used automatically when replicating between clusters.

2. Start the MySQL replication slave server as shown here:

```
shellS> mysqld --ndbcluster --server-id=id &
```

In the previous statement, *`id`* is the slave server's unique ID. It is not necessary to enable logging on the replication slave.

> **Note**
>
> You should use the `--skip-slave-start` option with this command or else you should include `skip-slave-start` in the slave server's `my.cnf` file, unless you want replication to begin immediately. With the use of this option, the start of replication is delayed until the appropriate `START SLAVE` statement has been issued, as explained in Step 4 below.

3. It is necessary to synchronize the slave server with the master server's replication binlog. If binary logging has not previously been running on the master, run the following statement on the slave:

```
mysqlS> CHANGE MASTER TO
     -> MASTER_LOG_FILE='',
     -> MASTER_LOG_POS=4;
```

This instructs the slave to begin reading the master's binary log from the log's starting point. Otherwise — that is, if you are loading data from the master using a backup — see Section 9.8, "Implementing Failover with MySQL Cluster Replication", for information on how to obtain the correct values to use for `MASTER_LOG_FILE` and `MASTER_LOG_POS` in such cases.

4. Finally, you must instruct the slave to begin applying replication by issuing this command from the `mysql` client on the replication slave:

```
mysqlS> START SLAVE;
```

This also initiates the transmission of replication data from the master to the slave.

It is also possible to use two replication channels, in a manner simliar to the procedure described in the next section; the differences between this and using a single replication channel are covered in Section 9.7, "Using Two Replication Channels for MySQL

Cluster Replication".

Beginning with MySQL Cluster NDB 6.2.3, it is possible to improve cluster replication performance by enabling *batched updates*. This can be accomplished by starting slave `mysqld` processes with the `--slave-allow-batching` option. Normally, updates are applied as soon as they are received. However, the use of batching causes updates to be applied in 32 KB batches, which can result in higher throughput and less CPU usage, particularly where individual updates are relatively small.

> **Note**
>
> Slave batching works on a per-epoch basis; updates belonging to more than one transaction can be sent as part of the same batch.
>
> All outstanding updates are applied when the end of an epoch is reached, even if the updates total less than 32 KB.

Batching can be turned on and off at runtime. To activate it at runtime, you can use either of these two statements:

```
SET GLOBAL slave_allow_batching = 1;
SET GLOBAL slave_allow_batching = ON;
```

If a particular batch causes problems (such as a statement whose effects do not appear to be replicated correctly), slave batching can be deactivated using either of the following statements:

```
SET GLOBAL slave_allow_batching = 0;
SET GLOBAL slave_allow_batching = OFF;
```

You can check whether slave batching is currently being used by means of an appropriate `SHOW VARIABLES` statement, like this one:

```
mysql> SHOW VARIABLES LIKE 'slave%';
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| slave_allow_batching      | ON    |
| slave_compressed_protocol | OFF   |
| slave_load_tmpdir         | /tmp  |
| slave_net_timeout         | 3600  |
| slave_skip_errors         | OFF   |
| slave_transaction_retries | 10    |
+---------------------------+-------+
6 rows in set (0.00 sec)
```

9.7. Using Two Replication Channels for MySQL Cluster Replication

In a more complete example scenario, we envision two replication channels to provide redundancy and thereby guard against possible failure of a single replication channel. This requires a total of four replication servers, two masters for the master cluster and two slave servers for the slave cluster. For purposes of the discussion that follows, we assume that unique identifiers are assigned as shown here:

Server ID	Description
1	Master - primary replication channel (*M*)
2	Master - secondary replication channel (*M'*)
3	Slave - primary replication channel (*S*)
4	Slave - secondary replication channel (*S'*)

Setting up replication with two channels is not radically different from setting up a single replication channel. First, the `mysqld` processes for the primary and secondary replication masters must be started, followed by those for the primary and secondary slaves. Then the replication processes may be initiated by issuing the `START SLAVE` statement on each of the slaves. The commands and the order in which they need to be issued are shown here:

1. Start the primary replication master:

   ```
   shellM> mysqld --ndbcluster --server-id=1 \
                  --log-bin --binlog-format=row &
   ```

2. Start the secondary replication master:

   ```
   shellM'> mysqld --ndbcluster --server-id=2 \
                  --log-bin --binlog-format=row &
   ```

3. Start the primary replication slave server:

```
shellS> mysqld --ndbcluster --server-id=3 \
              --skip-slave-start &
```

4. Start the secondary replication slave:

```
shellS'> mysqld --ndbcluster --server-id=4 \
               --skip-slave-start &
```

5. Finally, initiate replication on the primary channel by executing the START SLAVE statement on the primary slave as shown here:

```
mysqlS> START SLAVE;
```

> **Warning**
>
> Only the primary channel is to be started at this point. The secondary replication channel is to be started only in the event that the primary replication channel fails, as described in Section 9.8, "Implementing Failover with MySQL Cluster Replication". Running multiple replication channels simultaneously can result in unwanted duplicate records being created on the replication slaves.

As mentioned previously, it is not necessary to enable binary logging on replication slaves.

9.8. Implementing Failover with MySQL Cluster Replication

In the event that the primary Cluster replication process fails, it is possible to switch over to the secondary replication channel. The following procedure describes the steps required to accomplish this.

1. Obtain the time of the most recent global checkpoint (GCP). That is, you need to determine the most recent epoch from the ndb_apply_status table on the slave cluster, which can be found using the following query:

```
mysqlS'> SELECT @latest:=MAX(epoch)
      ->        FROM mysql.ndb_apply_status;
```

2. Using the information obtained from the query shown in Step 1, obtain the corresponding records from the ndb_binlog_index table on the master cluster as shown here:

```
mysqlM'> SELECT
      ->     @file:=SUBSTRING_INDEX(File, '/', -1),
      ->     @pos:=Position
      -> FROM mysql.ndb_binlog_index
      -> WHERE epoch > @latest
      -> ORDER BY epoch ASC LIMIT 1;
```

These are the records saved on the master since the failure of the primary replication channel. We have employed a user variable @latest here to represent the value obtained in Step 1. Of course, it is not possible for one mysqld instance to access user variables set on another server instance directly. These values must be "plugged in" to the second query manually or in application code.

3. Now it is possible to synchronize the secondary channel by running the following query on the secondary slave server:

```
mysqlS'> CHANGE MASTER TO
      ->     MASTER_LOG_FILE='@file',
      ->     MASTER_LOG_POS=@pos;
```

Again we have employed user variables (in this case @file and @pos) to represent the values obtained in Step 2 and applied in Step 3; in practice these values must be inserted manually or using application code that can access both of the servers involved.

> **Note**
>
> @file is a string value such as '/var/log/mysql/replication-master-bin.00001', and so must be quoted when used in SQL or application code. However, the value represented by @pos must *not* be quoted. Although MySQL normally attempts to convert strings to numbers, this case is an exception.

4. You can now initiate replication on the secondary channel by issuing the appropriate command on the secondary slave mysqld:

```
mysqlS'> START SLAVE;
```

Once the secondary replication channel is active, you can investigate the failure of the primary and effect repairs. The precise actions required to do this will depend upon the reasons for which the primary channel failed.

> **Warning**
>
> The secondary replication channel is to be started only if and when the primary replication channel has failed. Running multiple replication channels simultaneously can result in unwanted duplicate records being created on the replication slaves.

If the failure is limited to a single server, it should (in theory) be possible to replicate from *M* to *S'*, or from *M'* to *S*; however, this has not yet been tested.

9.9. MySQL Cluster Backups With MySQL Cluster Replication

This section discusses making backups and restoring from them using MySQL Cluster replication. We assume that the replication servers have already been configured as covered previously (see Section 9.5, “Preparing the MySQL Cluster for Replication”, and the sections immediately following). This having been done, the procedure for making a backup and then restoring from it is as follows:

1. There are two different methods by which the backup may be started.

   - **Method A.** This method requires that the cluster backup process was previously enabled on the master server, prior to starting the replication process. This can be done by including the following line in a `[mysql_cluster]` section in the `my.cnf file`, where *`management_host`* is the IP address or host name of the `NDB` management server for the master cluster, and *`port`* is the management server's port number:

     ```
     ndb-connectstring=management_host[:port]
     ```

     > **Note**
     >
     > The port number needs to be specified only if the default port (1186) is not being used. See Section 2.3, “MySQL Cluster Multi-Computer Configuration”, for more information about ports and port allocation in MySQL Cluster.

     In this case, the backup can be started by executing this statement on the replication master:

     ```
     shellM> ndb_mgm -e "START BACKUP"
     ```

   - **Method B.** If the `my.cnf` file does not specify where to find the management host, you can start the backup process by passing this information to the `NDB` management client as part of the `START BACKUP` command. This can be done as shown here, where *`management_host`* and *`port`* are the host name and port number of the management server:

     ```
     shellM> ndb_mgm management_host:port -e "START BACKUP"
     ```

     In our scenario as outlined earlier (see Section 9.5, “Preparing the MySQL Cluster for Replication”), this would be executed as follows:

     ```
     shellM> ndb_mgm rep-master:1186 -e "START BACKUP"
     ```

2. Copy the cluster backup files to the slave that is being brought on line. Each system running an `ndbd` process for the master cluster will have cluster backup files located on it, and *all* of these files must be copied to the slave to ensure a successful restore. The backup files can be copied into any directory on the computer where the slave management host resides, so long as the MySQL and NDB binaries have read permissions in that directory. In this case, we will assume that these files have been copied into the directory `/var/BACKUPS/BACKUP-1`.

   It is not necessary that the slave cluster have the same number of `ndbd` processes (data nodes) as the master; however, it is highly recommended this number be the same. It *is* necessary that the slave be started with the `--skip-slave-start` option, to prevent premature startup of the replication process.

3. Create any databases on the slave cluster that are present on the master cluster that are to be replicated to the slave.

   > **Important**
   >
   > A `CREATE DATABASE` (or `CREATE SCHEMA`) statement corresponding to each database to be replicated must be executed on each SQL node in the slave cluster.

4. Reset the slave cluster using this statement in the MySQL Monitor:

```
mysqlS> RESET SLAVE;
```

   It is important to make sure that the slave's `apply_status` table does not contain any records prior to running the restore process. You can accomplish this by running this SQL statement on the slave:

```
mysqlS> DELETE FROM mysql.ndb_apply_status;
```

5. You can now start the cluster restoration process on the replication slave using the `ndb_restore` command for each backup file in turn. For the first of these, it is necessary to include the `-m` option to restore the cluster metadata:

```
shellS> ndb_restore -c slave_host:port -n node-id \
        -b backup-id -m -r dir
```

   `dir` is the path to the directory where the backup files have been placed on the replication slave. For the `ndb_restore` commands corresponding to the remaining backup files, the `-m` option should *not* be used.

   For restoring from a master cluster with four data nodes (as shown in the figure in Chapter 9, *MySQL Cluster Replication*) where the backup files have been copied to the directory `/var/BACKUPS/BACKUP-1`, the proper sequence of commands to be executed on the slave might look like this:

```
shellS> ndb_restore -c rep-slave:1186 -n 2 -b 1 -m \
        -r ./var/BACKUPS/BACKUP-1
shellS> ndb_restore -c rep-slave:1186 -n 3 -b 1 \
        -r ./var/BACKUPS/BACKUP-1
shellS> ndb_restore -c rep-slave:1186 -n 4 -b 1 \
        -r ./var/BACKUPS/BACKUP-1
shellS> ndb_restore -c rep-slave:1186 -n 5 -b 1 -e \
        -r ./var/BACKUPS/BACKUP-1
```

   Important

   The `-e` (or `--restore-epoch`) option in the final invocation of `ndb_restore` in this example is required in order that the epoch is written to the slave `mysql.ndb_apply_status`. Without this information, the slave will not be able to synchronize properly with the master. (See Section 6.17, “`ndb_restore` — Restore a MySQL Cluster Backup”.)

6. Now you need to obtain the most recent epoch from the `ndb_apply_status` table on the slave (as discussed in Section 9.8, “Implementing Failover with MySQL Cluster Replication”):

```
mysqlS> SELECT @latest:=MAX(epoch)
        FROM mysql.ndb_apply_status;
```

7. Using `@latest` as the epoch value obtained in the previous step, you can obtain the correct starting position `@pos` in the correct binary log file `@file` from the master's `mysql.ndb_binlog_index` table using the query shown here:

```
mysqlM> SELECT
     ->     @file:=SUBSTRING_INDEX(File, '/', -1),
     ->     @pos:=Position
     -> FROM mysql.ndb_binlog_index
     -> WHERE epoch > @latest
     -> ORDER BY epoch ASC LIMIT 1;
```

   In the event that there is currently no replication traffic, you can get this information by running `SHOW MASTER STATUS` on the master and using the value in the `Position` column for the file whose name has the suffix with the greatest value for all files shown in the `File` column. However, in this case, you must determine this and supply it in the next step manually or by parsing the output with a script.

8. Using the values obtained in the previous step, you can now issue the appropriate `CHANGE MASTER TO` statement in the slave's `mysql` client:

```
mysqlS> CHANGE MASTER TO
     ->     MASTER_LOG_FILE='@file',
     ->     MASTER_LOG_POS=@pos;
```

9. Now that the slave “knows” from what point in which `binlog` file to start reading data from the master, you can cause the slave to begin replicating with this standard MySQL statement:

```
mysqlS> START SLAVE;
```

To perform a backup and restore on a second replication channel, it is necessary only to repeat these steps, substituting the host names and IDs of the secondary master and slave for those of the primary master and slave replication servers where appropriate, and running the preceding statements on them.

For additional information on performing Cluster backups and restoring Cluster from backups, see Section 7.3, “Online Backup of MySQL Cluster”.

9.9.1. MySQL Cluster Replication — Automating Synchronization of the Replication Slave to the Master Binary Log

It is possible to automate much of the process described in the previous section (see Section 9.9, “MySQL Cluster Backups With MySQL Cluster Replication”). The following Perl script `reset-slave.pl` serves as an example of how you can do this.

```
#!/user/bin/perl -w
#  file: reset-slave.pl
#  Copyright ©2005 MySQL AB
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 2 of the License, or
#  (at your option) any later version.
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to:
#  Free Software Foundation, Inc.
#  59 Temple Place, Suite 330
#  Boston, MA 02111-1307 USA
#
#  Version 1.1
######################### Includes ###############################
use DBI;
######################### Globals ################################
my  $m_host='';
my  $m_port='';
my  $m_user='';
my  $m_pass='';
my  $s_host='';
my  $s_port='';
my  $s_user='';
my  $s_pass='';
my  $dbhM='';
my  $dbhS='';
######################## Sub Prototypes ##########################
sub CollectCommandPromptInfo;
sub ConnectToDatabases;
sub DisconnectFromDatabases;
sub GetSlaveEpoch;
sub GetMasterInfo;
sub UpdateSlave;
######################### Program Main ###########################
CollectCommandPromptInfo;
ConnectToDatabases;
GetSlaveEpoch;
GetMasterInfo;
UpdateSlave;
DisconnectFromDatabases;
################## Collect Command Prompt Info ##################
sub CollectCommandPromptInfo
{
  ### Check that user has supplied correct number of command line args
  die "Usage:\n
       reset-slave >master MySQL host< >master MySQL port< \n
                   >master user< >master pass< >slave MySQL host< \n
                   >slave MySQL port< >slave user< >slave pass< \n
       All 8 arguments must be passed. Use BLANK for NULL passwords\n"
       unless @ARGV == 8;
  $m_host  =  $ARGV[0];
  $m_port  =  $ARGV[1];
  $m_user  =  $ARGV[2];
  $m_pass  =  $ARGV[3];
  $s_host  =  $ARGV[4];
  $s_port  =  $ARGV[5];
  $s_user  =  $ARGV[6];
  $s_pass  =  $ARGV[7];
  if ($m_pass eq "BLANK") { $m_pass = '';}
  if ($s_pass eq "BLANK") { $s_pass = '';}
}
###############  Make connections to both databases #############
sub ConnectToDatabases
{
  ### Connect to both master and slave cluster databases
  ### Connect to master
  $dbhM
    = DBI->connect(
    "dbi:mysql:database=mysql;host=$m_host;port=$m_port",
    "$m_user", "$m_pass")
      or die "Can't connect to Master Cluster MySQL process!
              Error: $DBI::errstr\n";
```

```
  ### Connect to slave
  $dbhS
    = DBI->connect(
          "dbi:mysql:database=mysql;host=$s_host",
          "$s_user", "$s_pass")
    or die "Can't connect to Slave Cluster MySQL process!
            Error: $DBI::errstr\n";
}
################  Disconnect from both databases ################
sub DisconnectFromDatabases
{
  ### Disconnect from master
  $dbhM->disconnect
  or warn " Disconnection failed: $DBI::errstr\n";
  ### Disconnect from slave
  $dbhS->disconnect
  or warn " Disconnection failed: $DBI::errstr\n";
}
######################  Find the last good GCI ##################
sub GetSlaveEpoch
{
  $sth = $dbhS->prepare("SELECT MAX(epoch)
                         FROM mysql.ndb_apply_status;")
      or die "Error while preparing to select epoch from slave: ",
             $dbhS->errstr;
  $sth->execute
      or die "Selecting epoch from slave error: ", $sth->errstr;
  $sth->bind_col (1, \$epoch);
  $sth->fetch;
  print "\tSlave Epoch =  $epoch\n";
  $sth->finish;
}
#######  Find the position of the last GCI in the binlog ########
sub GetMasterInfo
{
  $sth = $dbhM->prepare("SELECT
                           SUBSTRING_INDEX(File, '/', -1), Position
                         FROM mysql.ndb_binlog_index
                         WHERE epoch > $epoch
                         ORDER BY epoch ASC LIMIT 1;")
      or die "Prepare to select from master error: ", $dbhM->errstr;
  $sth->execute
      or die "Selecting from master error: ", $sth->errstr;
  $sth->bind_col (1, \$binlog);
  $sth->bind_col (2, \$binpos);
  $sth->fetch;
  print "\tMaster bin log =  $binlog\n";
  print "\tMaster Bin Log position =  $binpos\n";
  $sth->finish;
}
##########  Set the slave to process from that location #########
sub UpdateSlave
{
  $sth = $dbhS->prepare("CHANGE MASTER TO
                         MASTER_LOG_FILE='$binlog',
                         MASTER_LOG_POS=$binpos;")
      or die "Prepare to CHANGE MASTER error: ", $dbhS->errstr;
  $sth->execute
       or die "CHANGE MASTER on slave error: ", $sth->errstr;
  $sth->finish;
  print "\tSlave has been updated. You may now start the slave.\n";
}
# end reset-slave.pl
```

9.10. MySQL Cluster Replication — Multi-Master and Circular Replication

Beginning with MySQL 5.1.18, it is possible to use MySQL Cluster in multi-master replication, including circular replication between a number of MySQL Clusters.

> **Note**
>
> Prior to MySQL 5.1.18, multi-master replication including circular replication was not supported with MySQL Cluster replication. This was because log events created in a particular MySQL Cluster were wrongly tagged with the server ID of the master rather than the server ID of the originating server.

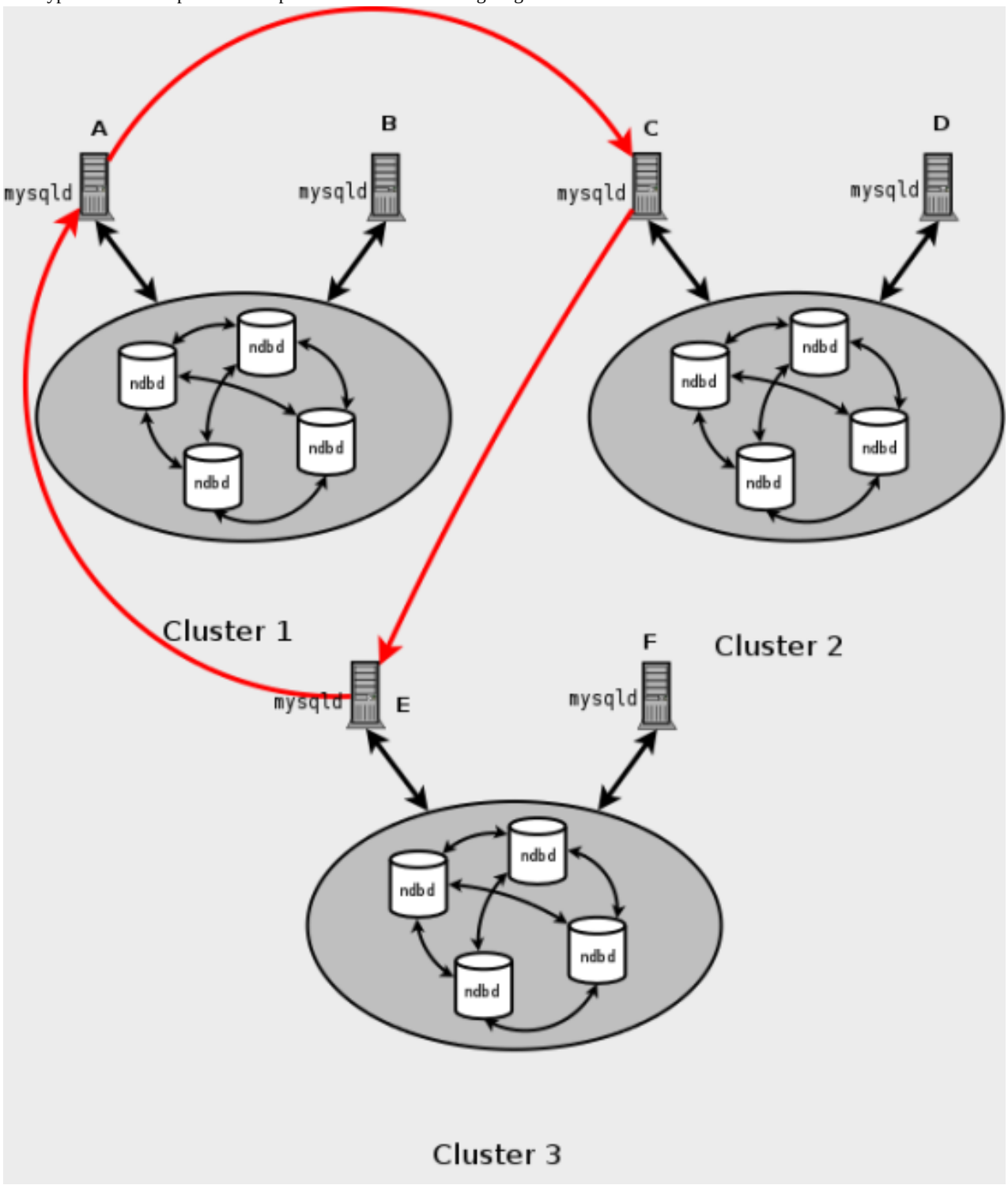
Circular replication example. In the next few paragraphs we consider the example of a replication setup involving three MySQL Clusters numbered 1, 2, and 3, in which Cluster 1 acts as the replication master for Cluster 2, Cluster 2 acts as the master for Cluster 3, and Cluster 3 acts as the master for Cluster 1. Each cluster has two SQL nodes, with SQL nodes A and B belonging to Cluster 1, SQL nodes C and D belonging to Cluster 2, and SQL nodes E and F belonging to Cluster 3.

Circular replication using these clusters is supported as long as:

- The SQL nodes on all masters and slaves are the same

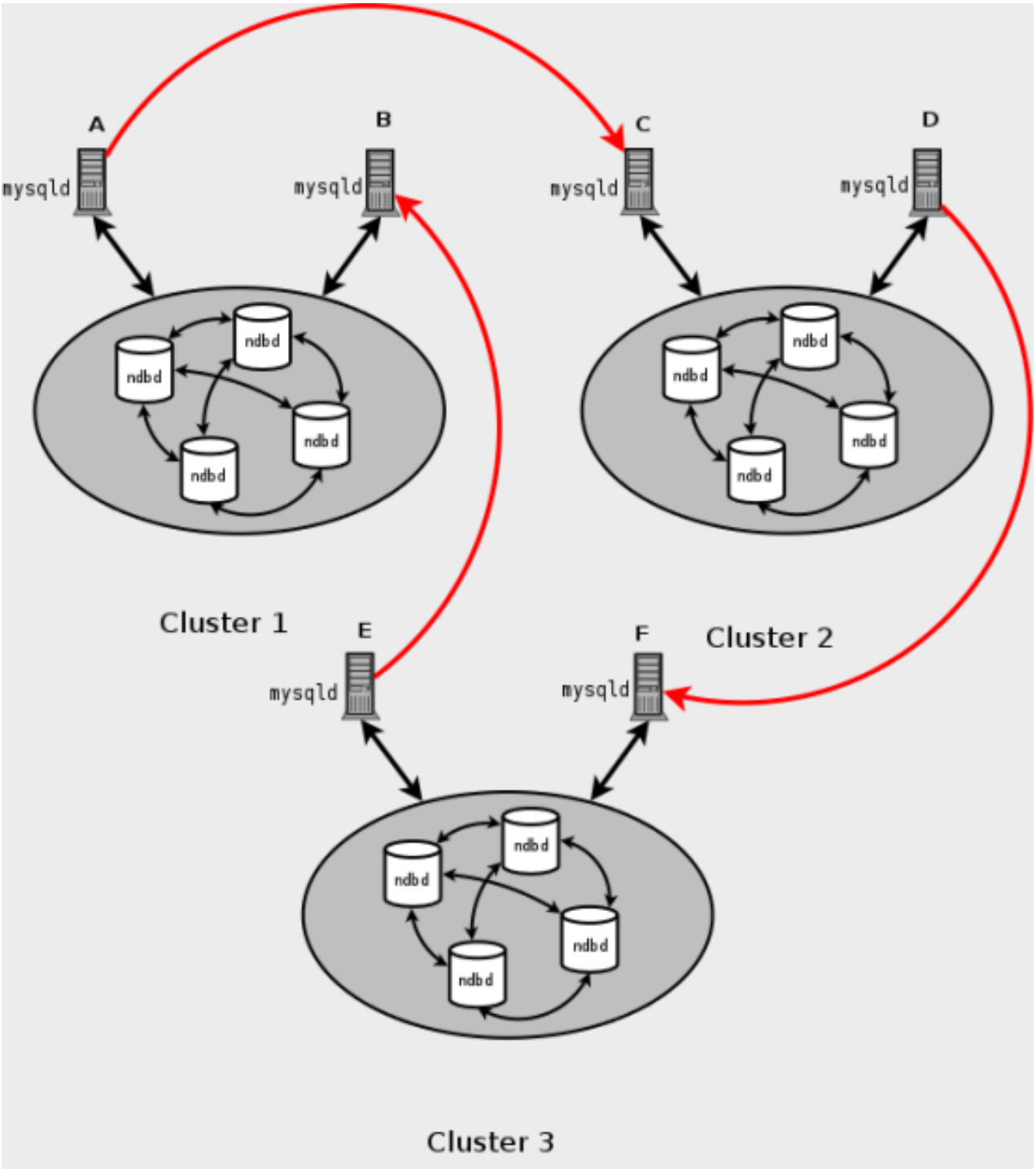
- All SQL nodes acting as replication masters and slaves are started using the `--log-slave-updates` option

This type of circular replication setup is shown in the following diagram:



In this scenario, SQL node A in Cluster 1 replicates to SQL node C in Cluster 2; SQL node C replicates to SQL node E in Cluster 3; SQL node E replicates to SQL node A. In other words, the replication line (indicated by the red arrows in the diagram) directly connects all SQL nodes used as replication masters and slaves.

It is also possible to set up circular replication in such a way that not all master SQL nodes are also slaves, as shown here:



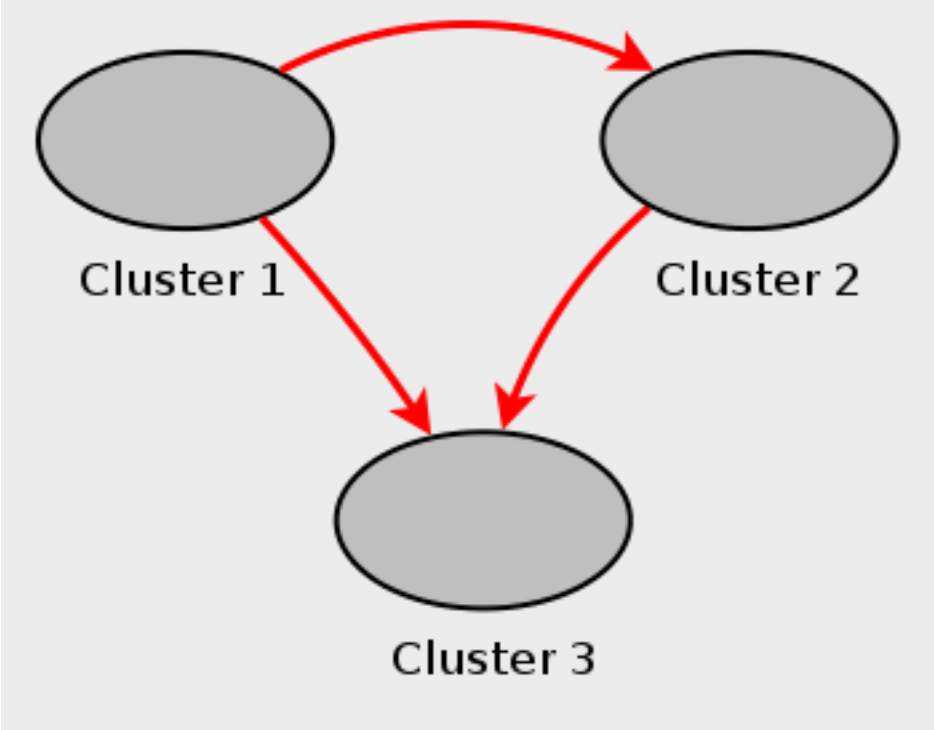
In this case, different SQL nodes in each cluster are used as replication masters and slaves. However, you must *not* start any of the SQL nodes using `--log-slave-updates` (see the description of this option for more information). This type of circular replication scheme for MySQL Cluster, in which the line of replication (again indicated by the red arrows in the diagram) is discontinuous, should be possible, but it should be noted that it has not yet been thoroughly tested and must therefore still be considered experimental.

> **Important**
>
> Beginning with MySQL 5.1.24, you should execute the following statement before starting circular replication:
>
> ```
> mysql> SET GLOBAL SLAVE_EXEC_MODE = 'IDEMPOTENT';
> ```
>
> This is necessary to suppress duplicate-key and other errors that otherwise break circular replication in MySQL Cluster. `IDEMPOTENT` mode is also required for multi-master replication when using MySQL Cluster. (Bug#31609)
>
> See `slave_exec_mode`, for more information.

Using NDB-native backup and restore to initialize a slave MySQL Cluster. When setting up circular replication, it is possible to initialize the slave cluster by using the management client `BACKUP` command on one MySQL Cluster to create a backup and then applying this backup on another MySQL Cluster using `ndb_restore`. However, this does not automatically create binary logs on the second MySQL Cluster's SQL node acting as the replication slave. In order to cause the binary logs to be created, you must issue a `SHOW TABLES` statement on that SQL node; this should be done prior to running `START SLAVE`.

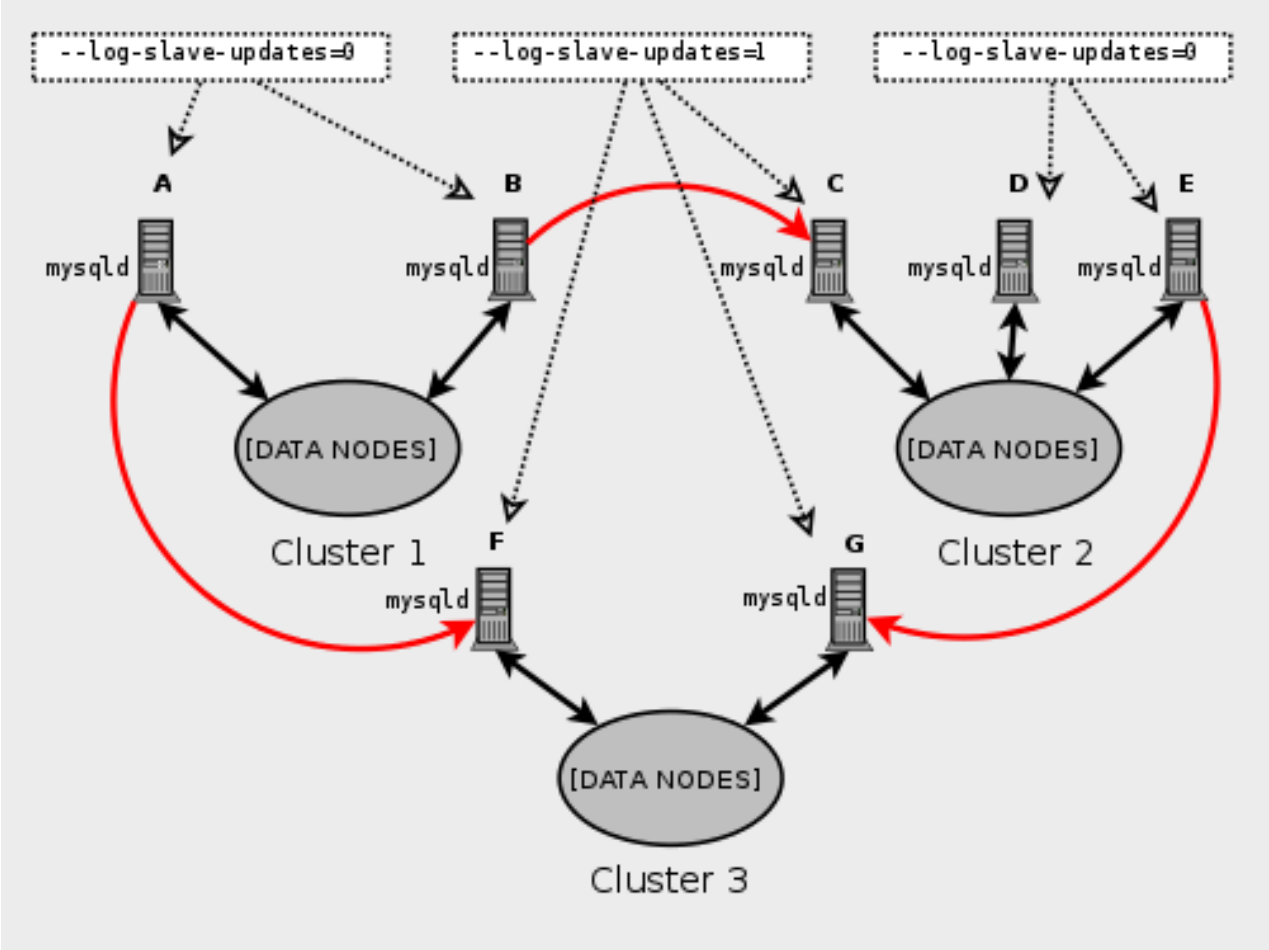
This is a known issue which we intend to address in a future release.

Multi-master failover example. In this section, we discuss failover in a multi-master MySQL Cluster replication setup with three MySQL Clusters having server IDs 1, 2, and 3. In this scenario, Cluster 1 replicates to Clusters 2 and 3; Cluster 2 also replicates to Cluster 3. This relationship is shown here:



In other words, data replicates from Cluster 1 to Cluster 3 via 2 different routes: directly, and by way of Cluster 2.

Not all MySQL servers taking part in multi-master replication must act as both master and slave, and a given MySQL Cluster might use different SQL nodes for diffferent replication channels. Such a case is shown here:



MySQL servers acting as replication slaves must be run with the `--log-slave-updates` option. Which `mysqld` processes require this option is also shown in the preceding diagram.

> **Note**
>
> Using the `--log-slave-updates` option has no effect on servers not being run as replication slaves.

The need for failover arises when one of the replicating clusters goes down. In this example, we consider the case where Cluster 1 is lost to service, and so Cluster 3 loses 2 sources of updates from Cluster 1. Because replication between MySQL Clusters is asynchronous, there is no guarantee that Cluster 3's updates originating directly from Cluster 1 are more recent than those received via Cluster 2. You can handle this by ensuring that Cluster 3 catches up to Cluster 2 with regard to updates from Cluster 1. In terms of MySQL servers, this means that you need to replicate any outstanding updates from MySQL server C to server F.

On server C, perform the following queries:

```
mysqlC> SELECT @latest:=MAX(epoch)
     ->     FROM mysql.ndb_apply_status
     ->     WHERE server_id=1;
mysqlC> SELECT
     ->     @file:=SUBSTRING_INDEX(File, '/', -1),
     ->     @pos:=Position
     ->     FROM mysql.ndb_binlog_index
     ->     WHERE orig_epoch >= @latest
     ->     AND orig_server_id = 1
     ->     ORDER BY epoch ASC LIMIT 1;
```

Copy over the values for *`@file`* and *`@pos`* manually from server C to server F (or have your application perform the equivalent). Then, on server F, execute the following `CHANGE MASTER TO` statement:

```
mysqlF> CHANGE MASTER TO
     ->     MASTER_HOST = 'serverC'
     ->     MASTER_LOG_FILE='@file',
     ->     MASTER_LOG_POS=@pos;
```

Once this has been done, you can issue a `START SLAVE` statement on MySQL server F, and any missing updates originating from server B will be replicated to server F.

9.11. MySQL Cluster Replication Conflict Resolution

When using a replication setup involving multiple masters (including circular replication), it is possible that different masters may try to update the same row on the slave with different data. Conflict resolution in MySQL Cluster Replication provides a means of resolving such conflicts by allowing a user defined resolution column to be used to determine whether or not an update to the row on a given master should be applied on the slave. (This column is sometimes referred to as a "timestamp" column, even though this column' type cannot be `TIMESTAMP`, as explained later in this section.) Different methods can be used to compare resolution column values on the slave when conflicts occur, as explained later in this section; the method used can be set on a per-table basis.

> **Important**
>
> Conflict resolution as described in this section is always applied on a row-by-row basis rather than a transactional basis. In addition, it is the application's responsibility to ensure that the resolution column is correctly populated with relevant values, so that the resolution function can make the appropriate choice when determining whether to apply an update.

Requirements. Preparations for conflict resolution must be made on both the master and the slave:

- On the master writing the binlogs, you must determine which columns are sent (all columns or only those that have been updated). This is done for the MySQL Server as a whole by applying the `mysqld` startup option `-â##ndb-log-updated-only` (described later in this section) or on a per-table basis by entries in the `mysql.ndb_replication` table.
- On the slave, you must determine which type of conflict resolution to apply ("latest timestamp wins", "same timestamp wins", or none). This is done using the `mysql.ndb_replication` system table, on a per-table basis.

If only some but not all columns are sent, then the master and slave can diverge.

> **Note**
>
> We refer to the column used for determining updates as a "timestamp" column, but the data type of this column is never `TIMESTAMP`; rather, its data type should be `INT` (`INTEGER`) or `BIGINT`. This column should be `UNSIGNED` and `NOT NULL`.

Master column control. We can see update operations in terms of "before" and "after" images — that is, the states of the table before and after the update is applied. Normally, when updating a table with a primary key, the "before" image is not of great interest; however, when we need to determine on a per-update basis whether or not to use the updated values on a replication slave, we need to make sure that both images are written to the master's binary log. This is done with the `--ndb-log-update-as-write` option for `mysqld`, as described later in this section.

> **Important**
>
> Whether logging of complete rows or of updated columns only is done is decided when the MySQL server is started, and cannot be changed online; you must either restart `mysqld`, or start a new `mysqld` instance with different logging options.

Logging full or partial rows (`--ndb-log-updated-only` option). For purposes of conflict resolution, there are two basic methods of logging rows, as determined by the setting of the `--ndb-log-updated-only` option for `mysqld`:

- Log complete rows
- Log only column data that has been updated — that is, column data whose value has been set, regardless of whether or not this value was actually changed.

It is more efficient to log updated columns only; however, if you need to log full rows, you can do so by setting `--ndb-log-updated-only` to `0` or `OFF`.

Logging changed data as updates (`--ndb-log-update-as-write` option). Either of these logging methods can be configured to be done with or without the "before" image as determined by the setting of another MySQL Server option `--ndb-log-update-as-write`. Because conflict resolution is done in the MySQL Server's update handler, it is necessary to control logging on the master such that updates are updates and not writes; that is, such that updates are treated as changes in existing rows rather than the writing of new rows (even though these replace existing rows). This option is turned on by default; to turn it off, start the server with `--ndb-log-update-as-write=0` or `--ndb-log-update-as-write=OFF`.

Conflict resolution control. Conflict resolution is usually enabled on the server where conflicts can occur. Like logging method selection, it is enabled by entries in the `mysql.ndb_replication` table.

The `ndb_replication` system table. To enable conflict resolution, it is necessary to create an `ndb_replication` table in the `mysql` system database on the master, the slave, or both, depending on the conflict resolution type and method to be employed.

This table is used to control logging and conflict resolution functions on a per-table basis, and has one row per table involved in replication. `ndb_replication` is created and filled with control information on the server where the conflict is to be resolved. In a simple master-slave setup where data can also be changed locally on the slave this will typically be the slave. In a more complex master-master (2-way) replication schema this will usually be all of the masters involved. Each row in `mysql.ndb_replication` corresponds to a table being replicated, and specifies how to log and resolve conflicts (that is, which conflict resolution function, if any, to use) for that table. The definition of the `mysql.ndb_replication` table is shown here:

```
CREATE TABLE mysql.ndb_replication  (
    db VARBINARY(63),
    table_name VARBINARY(63),
    server_id INT UNSIGNED,
    binlog_type INT UNSIGNED,
    conflict_fn VARBINARY(128),
    PRIMARY KEY USING HASH (db, table_name, server_id)
)   ENGINE=NDB
PARTITION BY KEY(db,table_name);
```

The columns in this table are described in the following list:

- **`db`.** The name of the database containing the table to be replicated.
- **`table_name`.** The name of the table to be replicated.
- **`server_id`.** The unique server ID of the MySQL instance (SQL node) where the table resides.
- **`binlog_type`.** The type of binary logging to be employed. This is determined as shown in the following table:

Value	Internal Value	Description
0	`NBT_DEFAULT`	Use server default
1	`NBT_NO_LOGGING`	Do not log this table in the binary log
2	`NBT_UPDATED_ONLY`	Only updated attributes are logged
3	`NBT_FULL`	Log full row, even if not updated (MySQL server default behavior)
4	`NBT_USE_UPDATE`	(For generating `NBT_UPDATED_ONLY_USE_UPDATE` and `NBT_FULL_USE_UPDATE` values only — not intended for separate use)
5	[*Not used*]	---
6	`NBT_UPDATED_ONLY_USE_UPDATE` (equal to `NBT_UPDATED_ONLY | NBT_USE_UPDATE`)	Use updated attributes, even if values are unchanged
7	`NBT_FULL_USE_UPDATE` (equal to `NBT_FULL | NBT_USE_UPDATE`)	Use full row, even if values are unchanged

- **`conflict_fn`.** The conflict resolution function to be applied. This function must be specified as one of the following:
  - **`NDB$MAX(column_name)`.** If the "timestamp" column value for a given row coming from the master is higher than that on the slave, it is applied; otherwise it is not applied on the slave. This is illustrated by the following pseudocode:

    ```
    if (master_new_column_value > slave_current_column_value)
      perform_update();
    ```

    This function can be used for "greatest timestamp wins" conflict resolution. This type of conflict resolution ensures that, in the event of a conflict, the version of the row that was most recently updated is the version that persists.

    Important

    The column value from the master's "after" image is used by this function.

    This conflict resolution function is available beginning with MySQL Cluster NDB 6.3.0.

  - **`NDB$OLD(column_name)`.** If the value of *`column_name`* is the same on both the master and the slave, then the update is applied; otherwise, the update is not applied on the slave and an exception is written to the log. This is illustrated by the following pseudocode:

    ```
    if (master_old_column_value == slave_current_column_value)
      perform_update();
    else
      log_exception();
    ```

This function can be used for “same value wins” conflict resolution. This type of conflict resolution ensures that updates are not applied on the slave from the wrong master.

> **Important**
>
> The column value from the master's “before” image is used by this function.

This conflict resolution function is available beginning with MySQL Cluster NDB 6.3.4.

- `NULL`: Indicates that conflict resolution is not to be used for the corresponding table

.

Status information. Beginning with MySQL Cluster NDB 6.3.3, a server status variable `Ndb_conflict_fn_max` provides a count of the number of times that a row was not applied on the current SQL node due to “greatest timestamp wins” conflict resolution since the last time that `mysqld` was started.

Beginning with MySQL Cluster NDB 6.3.4, the number of times that a row was not applied as the result of “same timestamp wins” conflict resolution on a given `mysqld` since the last time it was restarted is given by the global status variable `Ndb_conflict_fn_old`. In addition to incrementing `Ndb_conflict_fn_old`, the primary key of the row that was not used is inserted into an *exceptions table*, as explained later in this section.

Additional requirements for “Same timestamp wins” conflict resolution. To use the `NDB$OLD()` conflict resolution function, it is also necessary to create an exceptions table corresponding to each `NDB` table for which this type of conflict resolution is to be employed. The name of this table is that of the table for which “same timestamp wins” conflict resolution is to be applied, with the string `$EX` appended. (For example, if the name of the original table is `mytable`, the name of the corresponding exception table name should be `mytable$EX`.) This table is created as follows:

```
CREATE TABLE original_table$EX  (
    server_id INT UNSIGNED,
    master_server_id INT UNSIGNED,
    master_epoch BIGINT UNSIGNED,
    count INT UNSIGNED,
    original_table_pk_columns,
    [additional_columns,]
    PRIMARY KEY(server_id, master_server_id, master_epoch, count)
) ENGINE=NDB;
```

The first four columns are required. Following these columns, the columns making up the original table's primary key should be copied in the order in which they are used to define the primary key of the original table.

> **Note**
>
> The names of the first four columns and the columns matching the original table's primary key columns are not critical; however, we suggest for reasons of clarity and consistency, that you use the names shown here for the `server_id`, `master_server_id`, `master_epoch`, and `count` columns, and that you use the same names as in the original table for the columns matching those in the original table's primary key.
>
> The data types for the columns duplicating the primary key columns of the original table should be the same as for (or larger than) the original columns.

Additional columns may optionally be defined following these columns, but not before any of them; any such extra columns cannot be `NOT NULL`. The exception table's primary key must be defined as shown. The exception table must use the `NDB` storage engine. An example of use for `NDB$OLD()` and an exception table is given later in this section.

> **Important**
>
> The `mysql.ndb_replication` table is read when a data table is set up for replication, so the row corresponding to a table to be replicated must be inserted into `mysql.ndb_replication` *before* the table to be replicated is created.

Examples. The following examples assume that you have already a working MySQL Cluster replication setup, as described in Section 9.5, “Preparing the MySQL Cluster for Replication”, and Section 9.6, “Starting MySQL Cluster Replication (Single Replication Channel)”.

- **`NDB$MAX()` example.** Suppose you wish to enable “greatest timestamp wins” conflict resolution on table `test.t1`, using column `mycol` as the “timestamp”. This can be done using the following steps:

  1. Make sure that you have started the master `mysqld` with `-â##ndb-log-update-as-write=OFF`.

  2. On the master, perform this `INSERT` statement:

```
INSERT INTO mysql.ndb_replication
    VALUES ('test', 't1', 0, NULL, 'NDB$MAX(mycol)');
```

Inserting a 0 into the `server_id` indicates that all SQL nodes accessing this table should use conflict resolution. If you want to use conflict resolution on a specific `mysqld` only, use the actual server ID.

Inserting `NULL` into the `binlog_type` column has the same effect as inserting 0 (`NBT_DEFAULT`); the server default is used.

3. Create the `test.t1` table:

```
CREATE TABLE test.t1 (
    columns
    mycol INT UNSIGNED,
    columns
) ENGINE=NDB;
```

Now, when updates are done on this table, conflict resolution will be applied, and the version of the row having the greatest value for `mycol` will be written to the slave.

Note

Other `binlog_type` options — such as `NBT_UPDATED_ONLY_USE_UPDATE` should be used in order to control logging on the master via the `ndb_replication` table rather than by using command-line options.

- **`NDB$OLD()` example.** Suppose an `NDB` table such as the one defined here is being replicated, and you wish to enable "same timestamp wins" conflict resolution for updates to this table:

```
CREATE TABLE test.t2  (
    a INT UNSIGNED NOT NULL,
    b CHAR(25) NOT NULL,
    columns,
    mycol INT UNSIGNED NOT NULL,
    columns,
    PRIMARY KEY pk (a, b)
)   ENGINE=NDB;
```

The following steps are required, in the order shown:

1. First — and *prior* to creating `test.t2` — you must insert a row into the `mysql.ndb_replication` table, as shown here:

```
INSERT INTO mysql.ndb_replication
    VALUES ('test', 't2', 0, NULL, 'NDB$OLD(mycol)');
```

Possible values for the `binlog_type` column are shown earlier in this section. The value `'NDB$OLD(mycol)'` should be inserted into the `conflict_fn` column.

2. Create an appropriate exceptions table for `test.t2`. The table creation statement shown here includes all required columns; any additional columns must be declared following these columns, and before the definition of the table's primary key.

```
CREATE TABLE test.t2$EX  (
    server_id SMALLINT UNSIGNED,
    master_server_id INT UNSIGNED,
    master_epoch BIGINT UNSIGNED,
    count BIGINT UNSIGNED,
    a INT UNSIGNED NOT NULL,
    b CHAR(25) NOT NULL,
    [additional_columns,]
    PRIMARY KEY(server_id, master_server_id, master_epoch, count)
)   ENGINE=NDB;
```

3. Create the table `test.t2` as shown previously.

These steps must be followed for every table for which you wish to perform conflict resolution using `NDB$OLD()`. For each such table, there must be a corresponding row in `mysql.ndb_replication`, and there must be an exceptions table in the same database as the table being replicated.

Chapter 10. MySQL Cluster Disk Data Tables

Beginning with MySQL 5.1.6, it is possible to store the non-indexed columns of `NDB` tables on disk, rather than in RAM as with previous versions of MySQL Cluster.

As part of implementing MySQL Cluster Disk Data work, a number of improvements were made in MySQL Cluster for the efficient handling of very large amounts (terabytes) of data during node recovery and restart. These include a “no-steal” algorithm for synchronising a starting node with very large data sets. For more information, see the paper *Recovery Principles of MySQL Cluster 5.1*, by MySQL Cluster developers Mikael RonstrÃ¶m and Jonas Oreland.

MySQL Cluster Disk Data performance can be influenced by a number of configuration parameters. For information about these parameters and their effects, see *MySQL Cluster Disk Data configuration parameters* and *MySQL Cluster Disk Data storage and `GCP STOP` errors*

The performance of a MySQL Cluster that uses Disk Data storage can also be greatly improved by separating data node file systems from undo log files and tablespace data files, which can be done using symbolic links. For more information, see Section 10.2, “Using Symbolic Links with Disk Data Objects”.

10.1. MySQL Cluster Disk Data Objects

MySQL Cluster Disk Data storage is implemented using a number of *Disk Data objects*. These include the following:

- *Tablespaces* act as containers for other Disk Data objects.
- *Undo log files* undo information required for rolling back transactions.
- One or more undo log files are assigned to a *log file group*, which is then assigned to a tablespace.
- *Data files* store Disk Data table data. A data file is assigned directly to a tablespace.

Undo log files and data files are actual files in the filesystem of each data node; by default they are placed in `ndb_node_id_fs` in the `DataDir` specified in the MySQL Cluster `config.ini` file, and where `node_id` is the data node's node ID. It is possible to place these elsewhere by specifying either an absolute or relative path as part of the filename when creating the undo log or data file. Statements that create these files are shown later in this section.

MySQL Cluster tablespaces and log file groups are not implemented as files.

> **Important**
>
> Although not all Disk Data objects are implemented as files, they all share the same namespace. This means that *each Disk Data object* must be uniquely named (and not merely each Disk Data object of a given type). For example, you cannot have a tablespace and a log file group both named `dd1`.

Assuming that you have already set up a MySQL Cluster with all nodes (including management and SQL nodes) running MySQL 5.1.6 or newer, the basic steps for creating a Cluster table on disk are as follows:

1. Create a log file group, and assign one or more undo log files to it (an undo log file is also sometimes referred to as an *undofile*).

   > **Note**
   >
   > In MySQL 5.1 and later, undo log files are necessary only for Disk Data tables. They are no longer used for `NDB-CLUSTER` tables that are stored only in memory.

2. Create a tablespace; assign the log file group, as well as one or more data files, to the tablespace.
3. Create a Disk Data table that uses this tablespace for data storage.

Each of these tasks can be accomplished using SQL statements in the `mysql` client or other MySQL client application, as shown in the example that follows.

1. We create a log file group named `lg_1` using `CREATE LOGFILE GROUP`. This log file group is to be made up of two undo log files, which we name `undo_1.log` and `undo_2.log`, whose initial sizes are 16 MB and 12 MB, respectively. (The de-

fault initial size for an undo log file is 128 MB.) Optionally, you can also specify a size for the log file group's undo buffer, or allow it to assume the default value of 8 MB. In this example, we set the UNDO buffer's size at 2 MB. A log file group must be created with an undo log file; so we add `undo_1.log` to `lg_1` in this `CREATE LOGFILE GROUP` statement:

```
CREATE LOGFILE GROUP lg_1
    ADD UNDOFILE 'undo_1.log'
    INITIAL_SIZE 16M
    UNDO_BUFFER_SIZE 2M
    ENGINE NDBCLUSTER;
```

To add `undo_2.log` to the log file group, use the following `ALTER LOGFILE GROUP` statement:

```
ALTER LOGFILE GROUP lg_1
    ADD UNDOFILE 'undo_2.log'
    INITIAL_SIZE 12M
    ENGINE NDBCLUSTER;
```

Some items of note:

- The `.log` file extension used here is not required. We use it merely to make the log files easily recognisable.
- Every `CREATE LOGFILE GROUP` and `ALTER LOGFILE GROUP` statement must include an `ENGINE` clause. In MySQL 5.1, the permitted values for this clause are `NDBCLUSTER` and `NDB`.

Important

In MySQL 5.1.8 and later, there can exist only one log file group in the same MySQL Cluster at any given time.

- When you add an undo log file to a log file group using `ADD UNDOFILE 'filename'`, a file with the name *`filename`* is created in the `ndb_node_id_fs` directory within the `DataDir` of each data node in the cluster, where *`node_id`* is the node ID of the data node. Each undo log file is of the size specified in the SQL statement. For example, if a MySQL Cluster has 4 data nodes, then the `ALTER LOGFILE GROUP` statement just shown creates 4 undo log files, 1 each on in the data directory of each of the 4 data nodes; each of these files is named `undo_2.log` and each file is 12 MB in size.
- `UNDO_BUFFER_SIZE` is limited by the amount of system memory available.
- For more information about the `CREATE LOGFILE GROUP` statement, see `CREATE LOGFILE GROUP` Syntax. For more information about `ALTER LOGFILE GROUP`, see `ALTER LOGFILE GROUP` Syntax.

2.

Now we can create a tablespace, which contains files to be used by MySQL Cluster Disk Data tables for storing their data. A tablespace is also associated with a particular log file group. When creating a new tablespace, you must specify the log file group which it is to use for undo logging; you must also specify a data file. You can add more data files to the tablespace after the tablespace is created; it is also possible to drop data files from a tablespace (an example of dropping data files is provided later in this section).

Assume that we wish to create a tablespace named `ts_1` which uses `lg_1` as its log file group. This tablespace is to contain two data files named `data_1.dat` and `data_2.dat`, whose initial sizes are 32 MB and 48 MB, respectively. (The default value for `INITIAL_SIZE` is 128 MB.) We can do this using two SQL statements, as shown here:

```
CREATE TABLESPACE ts_1
    ADD DATAFILE 'data_1.dat'
    USE LOGFILE GROUP lg_1
    INITIAL_SIZE 32M
    ENGINE NDBCLUSTER;
ALTER TABLESPACE ts_1
    ADD DATAFILE 'data_2.dat'
    INITIAL_SIZE 48M
    ENGINE NDBCLUSTER;
```

The `CREATE TABLESPACE` statement creates a tablespace `ts_1` with the data file `data_1.dat`, and associates `ts_1` with log file group `lg_1`. The `ALTER TABLESPACE` adds the second data file (`data_2.dat`).

Some items of note:

- As is the case with the `.log` file extension used in this example for undo log files, there is no special significance for the `.dat` file extension; it is used merely for easy recognition of data files.
- When you add a data file to a tablespace using `ADD DATAFILE 'filename'`, a file with the name *`filename`* is created in the `ndb_node_id_fs` directory within the `DataDir` of each data node in the cluster, where *`node_id`* is the node ID of the data node. Each undo log file is of the size specified in the SQL statement. For example, if a MySQL Cluster has 4 data nodes, then the `ALTER TABLESPACE` statement just shown creates 4 undo log files, 1 each on in the

data directory of each of the 4 data nodes; each of these files is named `data_2.dat` and each file is 48 MB in size.

- All `CREATE TABLESPACE` and `ALTER TABLESPACE` statements must contain an `ENGINE` clause; only tables using the same storage engine as the tablespace can be created in the tablespace. In MySQL 5.1, the only permitted values for this clause are `NDBCLUSTER` and `NDB`.
- For more information about the `CREATE TABLESPACE` and `ALTER TABLESPACE` statements, see `CREATE TABLESPACE` Syntax, and `ALTER TABLESPACE` Syntax.

3. Now it is possible to create a table whose non-indexed columns are stored on disk in the tablespace `ts_1`:

```
CREATE TABLE dt_1 (
    member_id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    last_name VARCHAR(50) NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    dob DATE NOT NULL,
    joined DATE NOT NULL,
    INDEX(last_name, first_name)
    )
    TABLESPACE ts_1 STORAGE DISK
    ENGINE NDBCLUSTER;
```

The `TABLESPACE ... STORAGE DISK` option tells the `NDBCLUSTER` storage engine to use tablespace `ts_1` for disk data storage.

> **Note**
>
> Beginning with MySQL Cluster NDB 6.2.5 and MySQL Cluster NDB 6.3.2, it is also possible to specify whether an individual column is stored on disk or in memory by using a `STORAGE` clause as part of the column's definition in a `CREATE TABLE` or `ALTER TABLE` statement. `STORAGE DISK` causes the column to be stored on disk, and `STORAGE MEMORY` causes in-memory storage to be used. See `CREATE TABLE` Syntax, for more information.

Once table `ts_1` has been created as shown, you can perform `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements on it just as you would with any other MySQL table.

For table `dt_1` as it has been defined here, only the `dob` and `joined` columns are stored on disk. This is because there are indexes on the `id`, `last_name`, and `first_name` columns, and so data belonging to these columns is stored in RAM. In MySQL 5.1, only non-indexed columns can be held on disk; indexes and indexed column data continue to be stored in memory. This tradeoff between the use of indexes and conservation of RAM is something you must keep in mind as you design Disk Data tables.

Performance note. The performance of a cluster using Disk Data storage is greatly improved if Disk Data files are kept on a separate physical disk from the data node file system. This must be done for each data node in the cluster to derive any noticeable benefit.

You may use absolute and relative file system paths with `ADD UNDOFILE` and `ADD DATAFILE`. Relative paths are calculated relative to the data node's data directory. You may also use symbolic links; see Section 10.2, “Using Symbolic Links with Disk Data Objects”, for more information and examples.

A log file group, a tablespace, and any Disk Data tables using these must be created in a particular order. The same is true for dropping any of these objects:

- A log file group cannot be dropped as long as any tablespaces are using it.
- A tablespace cannot be dropped as long as it contains any data files.
- You cannot drop any data files from a tablespace as long as there remain any tables which are using the tablespace.
- Beginning with MySQL 5.1.12, it is no longer possible to drop files created in association with a different tablespace than the one with which the files were created. (Bug#20053)

For example, to drop all the objects created so far in this section, you would use the following statements:

```
mysql> DROP TABLE dt_1;
mysql> ALTER TABLESPACE ts_1
    -> DROP DATAFILE 'data_2.dat'
    -> ENGINE NDBCLUSTER;
mysql> ALTER TABLESPACE ts_1
    -> DROP DATAFILE 'data_1.dat'
    -> ENGINE NDBCLUSTER;
```

```
mysql> DROP TABLESPACE ts_1
    -> ENGINE NDBCLUSTER;
mysql> DROP LOGFILE GROUP lg_1
    -> ENGINE NDBCLUSTER;
```

These statements must be performed in the order shown, except that the two `ALTER TABLESPACE ... DROP DATAFILE` statements may be executed in either order.

You can obtain information about data files used by Disk Data tables by querying the `FILES` table in the `INFORMATION_SCHEMA` database. An extra "`NULL` row" was added to this table in MySQL 5.1.14 for providing additional information about undo log files. For more information and examples of use, see The `INFORMATION_SCHEMA FILES` Table.

10.2. Using Symbolic Links with Disk Data Objects

The performance of a MySQL Cluster that uses Disk Data storage can be greatly improved by separating data node file systems from undo log files and tablespace data files and placing these on different disks. While there is currently no direct support for this in MySQL Cluster, it is possible to achieve this separation using symbolic links.

Each data node in the cluster creates a file system in the directory named `ndb_node_id_fs` under the data node's `DataDir` as defined in the `config.ini` file. In this example, we assume that each data node host has 3 disks, aliased as `/data0`, `/data1`, and `/data2`, and that the cluster's `config.ini` includes the following:

```
[ndbd default]
DataDir= /data0
```

Our objective is to place all Disk Data log files in `/data1`, and all Disk Data data files in `/data2`, on each data node host.

> **Note**
>
> In this example, we assume that the cluster's data node hosts are all using Linux operating systems. For other platforms, you may need to substitute you operating system's commands for those shown here.

To accomplish this, perform the following steps:

- Under the data node file system create symbolic links pointing to the other drives:

  ```
  shell> cd /data0/ndb_2_fs
  shell> ls
  D1  D10  D11  D2  D8  D9  LCP
  shell> ln -s /data0 dnlogs
  shell> ln -s /data1 dndata
  ```

  You should now have two symbolic links:

  ```
  shell> ls -l --hide=D*
  lrwxrwxrwx 1 user group   30 2007-03-19 13:58 dndata -> /data1
  lrwxrwxrwx 1 user group   30 2007-03-19 13:59 dnlogs -> /data2
  ```

  We show this only for the data node with node ID 2; however, you must do this for *each* data node.

- Now, in the `mysql` client, create a log file group and tablespace using the symbolic links, as shown here:

  ```
  mysql> CREATE LOGFILE GROUP lg1
      ->    ADD UNDOFILE 'dnlogs/undo1.log'
      ->    INITIAL_SIZE 150M
      ->    UNDO_BUFFER_SIZE = 1M
      ->    ENGINE=NDBCLUSTER;
  mysql> CREATE TABLESPACE ts1
      ->    ADD DATAFILE 'dndata/data1.log'
      ->    USE LOGFILE GROUP lg1
      ->    INITIAL_SIZE 1G
      ->    ENGINE=NDBCLUSTER;
  ```

  Verify that the files were created and placed correctly as shown here:

  ```
  shell> cd /data1
  shell> ls -l
  total 2099304
  -rw-rw-r--  1 user group 157286400 2007-03-19 14:02 undo1.dat
  shell> cd /data2
  shell> ls -l
  total 2099304
  -rw-rw-r--  1 user group 1073741824 2007-03-19 14:02 data1.dat
  ```

- If you are running multiple data nodes on one host, you must take care to avoid having them try to use the same space for Disk Data files. You can make this easier by creating a symbolic link in each data node filesystem. Suppose you are using `/data0` for both data node filesystems, but you wish to have the Disk Data files for both nodes on `/data1`. In this case, you can do

something similar to what is shown here:

```
shell> cd /data0
shell> ln -s ndb_2_fs/dd /data1/dn2
shell> ln -s ndb_3_fs/dd /data1/dn3
shell> ls -l --hide=D* ndb_2_fs
lrwxrwxrwx 1 user group   30 2007-03-19 14:22 dd -> /data1/dn2
shell> ls -l --hide=D* ndb_3_fs
lrwxrwxrwx 1 user group   30 2007-03-19 14:22 dd -> /data1/dn3
```

Now you can create a logfile group and tablespace using the symbolic link, like this:

```
mysql> CREATE LOGFILE GROUP lg1
    ->    ADD UNDOFILE 'dd/undo1.log'
    ->    INITIAL_SIZE 150M
    ->    UNDO_BUFFER_SIZE = 1M
    ->    ENGINE=NDBCLUSTER;
mysql> CREATE TABLESPACE ts1
    ->    ADD DATAFILE 'dd/data1.log'
    ->    USE LOGFILE GROUP lg1
    ->    INITIAL_SIZE 1G
    ->    ENGINE=NDBCLUSTER;
```

Verify that the files were created and placed correctly as shown here:

```
shell> cd /data1
shell> ls
dn2        dn3
shell> ls dn2
undo1.log        data1.log
shell> ls dn3
undo1.log        data1.log
```

10.3. MySQL Cluster Disk Data Storage Requirements

The following items apply to Disk Data storage requirements:

- Variable-length columns of Disk Data tables take up a fixed amount of space. For each row, this is equal to the space required to store the largest possible value for that column.

  For general information about calculating these values, see Data Type Storage Requirements.

  You can obtain an estimate the amount of space available in data files and undo log files by querying the `INFORMATION_SCHEMA.FILES` table. For more information and examples, see The `INFORMATION_SCHEMA FILES` Table.

  > **Note**
  >
  > The `OPTIMIZE TABLE` statement does not have any effect on Disk Data tables.

- In a Disk Data table, the first 256 bytes of a `TEXT` or `BLOB` column are stored in memory; only the remainder is stored on disk.

- Each row in a Disk Data table uses 8 bytes in memory to point to the data stored on disk. This means that, in some cases, converting an in-memory column to the disk-based format can actually result in greater memory usage. For example, convering a `CHAR(4)` column from memory-based to disk-based format increases the amount of `DataMemory` used per row from 4 to 8 bytes.

  > **Important**
  >
  > Starting the cluster with the `--initial` option does *not* remove Disk Data files. You must remove these manually prior to performing an initial restart of the cluster.

Chapter 11. Using High-Speed Interconnects with MySQL Cluster

Even before design of NDBCLUSTER began in 1996, it was evident that one of the major problems to be encountered in building parallel databases would be communication between the nodes in the network. For this reason, NDBCLUSTER was designed from the very beginning to allow for the use of a number of different data transport mechanisms. In this Manual, we use the term *transporter* for these.

The MySQL Cluster codebase includes support for four different transporters:

- *TCP/IP using 100 Mbps or gigabit Ethernet*, as discussed in Section 3.4.8, “MySQL Cluster TCP/IP Connections”.
- *Direct (machine-to-machine) TCP/IP*; although this transporter uses the same TCP/IP protocol as mentioned in the previous item, it requires setting up the hardware differently and is configured differently as well. For this reason, it is considered a separate transport mechanism for MySQL Cluster. See Section 3.4.9, “MySQL Cluster TCP/IP Connections Using Direct Connections”, for details.
- *Shared memory (SHM)*. For more information about SHM, see Section 3.4.10, “MySQL Cluster Shared-Memory Connections”.
- *Scalable Coherent Interface (SCI)*, as described in the next section of this chapter, Section 3.4.11, “SCI Transport Connections in MySQL Cluster”.

Most users today employ TCP/IP over Ethernet because it is ubiquitous. TCP/IP is also by far the best-tested transporter for use with MySQL Cluster.

We are working to make sure that communication with the ndbd process is made in “chunks” that are as large as possible because this benefits all types of data transmission.

For users who desire it, it is also possible to use cluster interconnects to enhance performance even further. There are two ways to achieve this: Either a custom transporter can be designed to handle this case, or you can use socket implementations that bypass the TCP/IP stack to one extent or another. We have experimented with both of these techniques using the SCI (Scalable Coherent Interface) technology developed by Dolphin Interconnect Solutions.

11.1. Configuring MySQL Cluster to use SCI Sockets

It is possible employing Scalable Coherent Interface (SCI) technology to achieve a significant increase in connection speeds and throughput between MySQL Cluster data and SQL nodes. To use SCI, it is necessary to obtain and install Dolphin SCI network cards and to use the drivers and other software supplied by Dolphin. You can get information on obtaining these, from Dolphin Interconnect Solutions. SCI SuperSocket or SCI Transporter support is available for 32-bit and 64-bit Linux, Solaris, Windows, and other platforms. See the Dolphin documentation referenced later in this section for more detailed information regarding platforms supported for SCI.

> **Note**
>
> Prior to MySQL 5.1.20, there were issues with building MySQL Cluster with SCI support (see Bug#25470), but these have been resolved due to work contributed by Dolphin. SCI Sockets are now correctly supported for MySQL Cluster hosts running recent versions of Linux using the -max builds, and versions of MySQL Cluster with SCI Transporter support can be built using either of compile-amd64-max-sci or compile-pentium64-max-sci. Both of these build scripts can be found in the BUILD directory of the MySQL Cluster source trees; it should not be difficult to adapt them for other platforms. Generally, all that is necessary is to compile MySQL Cluster with SCI Transporter support is to configure the MySQL Cluster build using --with-ndb-sci=/opt/DIS.

Once you have acquired the required Dolphin hardware and software, you can obtain detailed information on how to adapt a MySQL Cluster configured for normal TCP/IP communication to use SCI from the *Dolphin Express for MySQL Installation and Reference Guide*, available for download at http://docsrva.mysql.com/public/DIS_install_guide_book.pdf (PDF file, 94 pages, 753 KB). This document provides instructions for installing the SCI hardware and software, as well as information concerning network topology and configuration.

11.2. MySQL Cluster Interconnects and Performance

The ndbd process has a number of simple constructs which are used to access the data in a MySQL Cluster. We have created a very simple benchmark to check the performance of each of these and the effects which various interconnects have on their performance.

There are four access methods:

- **Primary key access.** This is access of a record through its primary key. In the simplest case, only one record is accessed at a time, which means that the full cost of setting up a number of TCP/IP messages and a number of costs for context switching are borne by this single request. In the case where multiple primary key accesses are sent in one batch, those accesses share the cost of setting up the necessary TCP/IP messages and context switches. If the TCP/IP messages are for different destinations, additional TCP/IP messages need to be set up.

- **Unique key access.** Unique key accesses are similar to primary key accesses, except that a unique key access is executed as a read on an index table followed by a primary key access on the table. However, only one request is sent from the MySQL Server, and the read of the index table is handled by `ndbd`. Such requests also benefit from batching.

- **Full table scan.** When no indexes exist for a lookup on a table, a full table scan is performed. This is sent as a single request to the `ndbd` process, which then divides the table scan into a set of parallel scans on all cluster `ndbd` processes. In future versions of MySQL Cluster, an SQL node will be able to filter some of these scans.

- **Range scan using ordered index**

  When an ordered index is used, it performs a scan in the same manner as the full table scan, except that it scans only those records which are in the range used by the query transmitted by the MySQL server (SQL node). All partitions are scanned in parallel when all bound index attributes include all attributes in the partitioning key.

With benchmarks developed internally by MySQL for testing simple and batched primary and unique key accesses, we have found that using SCI sockets improves performance by approximately 100% over TCP/IP, except in rare instances when communication performance is not an issue. This can occur when scan filters make up most of processing time or when very large batches of primary key accesses are achieved. In that case, the CPU processing in the `ndbd` processes becomes a fairly large part of the overhead.

Using the SCI transporter instead of SCI Sockets is only of interest in communicating between `ndbd` processes. Using the SCI transporter is also only of interest if a CPU can be dedicated to the `ndbd` process because the SCI transporter ensures that this process will never go to sleep. It is also important to ensure that the `ndbd` process priority is set in such a way that the process does not lose priority due to running for an extended period of time, as can be done by locking processes to CPUs in Linux 2.6. If such a configuration is possible, the `ndbd` process will benefit by 10–70% as compared with using SCI sockets. (The larger figures will be seen when performing updates and probably on parallel scan operations as well.)

There are several other optimized socket implementations for computer clusters, including Myrinet, Gigabit Ethernet, Infiniband and the VIA interface. However, we have tested MySQL Cluster so far only with SCI sockets. See Section 11.1, “Configuring MySQL Cluster to use SCI Sockets”, for information on how to set up SCI sockets using ordinary TCP/IP for MySQL Cluster.

Chapter 12. Known Limitations of MySQL Cluster

In the sections that follow, we discuss known limitations in current releases of MySQL Cluster as compared with the features available when using the MyISAM and InnoDB storage engines. If you check the "Cluster" category in the MySQL bugs database at http://bugs.mysql.com, you can find known bugs in the following categories under "MySQL Server:" in the MySQL bugs database at http://bugs.mysql.com, which we intend to correct in upcoming releases of MySQL Cluster:

- Cluster
- Cluster Direct API (NDBAPI)
- Cluster Disk Data
- Cluster Replication

This information is intended to be complete with respect to the conditions just set forth. You can report any discrepancies that you encounter to the MySQL bugs database using the instructions given in How to Report Bugs or Problems. If we do not plan to fix the problem in MySQL 5.1, we will add it to the list.

See Section 12.11, "Previous MySQL Cluster Issues Resolved in MySQL 5.1 and MySQL Cluster NDB 6.x" for a list of issues in MySQL Cluster in MySQL 5.0 that have been resolved in the current version.

> **Note**
>
> Limitations and other issues specific to MySQL Cluster Replication are described in Section 9.3, "Known Issues in MySQL Cluster Replication".

12.1. Non-Compliance with SQL Syntax in MySQL Cluster

Some SQL statements relating to certain MySQL features produce errors when used with NDB tables, as described in the following list:

- **Temporary tables.** Temporary tables are not supported. Trying either to create a temporary table that uses the NDB storage engine or to alter an existing temporary table to use NDB fails with the error `TABLE STORAGE ENGINE 'NDBCLUSTER' DOES NOT SUPPORT THE CREATE OPTION 'TEMPORARY'`.
- **Indexes and keys in NDB tables.** Keys and indexes on MySQL Cluster tables are subject to the following limitations:
  - **TEXT and BLOB columns.** You cannot create indexes on NDB table columns that use any of the TEXT or BLOB data types.
  - **FULLTEXT indexes.** The NDB storage engine does not support FULLTEXT indexes, which are possible for MyISAM tables only.

    However, you can create indexes on VARCHAR columns of NDB tables.
  - **BIT columns.** A BIT column cannot be a primary key, unique key, or index, nor can it be part of a composite primary key, unique key, or index.
  - **AUTO_INCREMENT columns.** Like other MySQL storage engines, the NDB storage engine can handle a maximum of one AUTO_INCREMENT column per table. However, in the case of a Cluster table with no explicit primary key, an AUTO_INCREMENT column is automatically defined and used as a "hidden" primary key. For this reason, you cannot define a table that has an explicit AUTO_INCREMENT column unless that column is also declared using the PRIMARY KEY option. Attempting to create a table with an AUTO_INCREMENT column that is not the table's primary key, and using the NDB storage engine, fails with an error.
- **MySQL Cluster and geometry data types.** Geometry datatypes (WKT and WKB) are supported in NDB tables in MySQL 5.1. However, spatial indexes are not supported.
- **Creating NDBCLUSTER tables with user-defined partitioning.** Support for user-defined partitioning for MySQL Cluster in MySQL 5.1 is restricted to [LINEAR] KEY partitioning. Beginning with MySQL 5.1.12, using any other partitioning type with ENGINE=NDB or ENGINE=NDBCLUSTER in a CREATE TABLE statement results in an error.

  Default partitioning scheme. As of MySQL 5.1.6, all MySQL Cluster tables are by default partitioned by KEY using the table's primary key as the partitioning key. If no primary key is explicitly set for the table, the "hidden" primary key automatically created by the NDBCLUSTER storage engine is used instead. For additional discussion of these and related issues, see KEY

Partitioning.

Beginning with MySQL Cluster NDB 6.2.18, MySQL Cluster NDB 6.3.25, and MySQL Cluster NDB 7.0.6, `CREATE TABLE` and `ALTER TABLE` statements that would cause a user-partitioned `NDBCLUSTER` table not to meet either or both of the following two requirements are disallowed, and fail with an error (Bug#40709):

1. The table must have an explicit primary key.

2. All columns listed in the table's partitioning expression must be part of the primary key.

Exception. If a user-partitioned `NDBCLUSTER` table is created using an empty column-list (that is, using `PARTITION BY [LINEAR] KEY()`), then no explicit primary key is required.

Maximum number of partitions for `NDBCLUSTER` tables. The maximum number of partitions that can defined for a `NDBCLUSTER` table when employing user-defined partitioning is 8 per node group. (See Section 1.2, "MySQL Cluster Nodes, Node Groups, Replicas, and Partitions", for more information about MySQL Cluster node groups.

`DROP PARTITION` not supported. It is not possible to drop partitions from `NDB` tables using `ALTER TABLE ... DROP PARTITION`. The other partitioning extensions to `ALTER TABLE` — `ADD PARTITION`, `REORGANIZE PARTITION`, and `COALESCE PARTITION` — are supported for Cluster tables, but use copying and so are not optimised. See Management of `RANGE` and `LIST` Partitions and `ALTER TABLE` Syntax.

- **Row-based replication.** When using row-based replication with MySQL Cluster, binary logging cannot be disabled. That is, the `NDB` storage engine ignores the value of `sql_log_bin`. (Bug#16680)

12.2. Limits and Differences of MySQL Cluster from Standard MySQL Limits

In this section, we list limits found in MySQL Cluster that either differ from limits found in, or that are not found in, standard MySQL.

- **Memory usage and recovery.** Memory consumed when data is inserted into an `NDB` table is not automatically recovered when deleted, as it is with other storage engines. Instead, the following rules hold true:

  - A `DELETE` statement on an `NDB` table makes the memory formerly used by the deleted rows available for re-use by inserts on the same table only. This memory cannot be used by other `NDB` tables.

  - A `DROP TABLE` or `TRUNCATE` operation on an `NDB` table frees the memory that was used by this table for re-use by any `NDB` table, either by the same table or by another `NDB` table.

  Note

  Recall that `TRUNCATE` drops and re-creates the table. See `TRUNCATE` Syntax.

  Memory freed by `DELETE` operations but still allocated to a specific table can also be made available for general re-use by performing a rolling restart of the cluster. See Section 5.1, "Performing a Rolling Restart of a MySQL Cluster".

  Beginning with MySQL Cluster NDB 6.3.7, this limitation can be overcome using `OPTIMIZE TABLE`. See Section 12.11, "Previous MySQL Cluster Issues Resolved in MySQL 5.1 and MySQL Cluster NDB 6.x", for more information.

- **Limits imposed by the cluster's configuration.** A number of hard limits exist which are configurable, but available main memory in the cluster sets limits. See the complete list of configuration parameters in Section 3.4, "MySQL Cluster Configuration Files". Most configuration parameters can be upgraded online. These hard limits include:

  - Database memory size and index memory size (`DataMemory` and `IndexMemory`, respectively).

    `DataMemory` is allocated as 32KB pages. As each `DataMemory` page is used, it is assigned to a specific table; once allocated, this memory cannot be freed except by dropping the table.

    See Section 3.4.6, "Defining MySQL Cluster Data Nodes", for further information about `DataMemory` and `IndexMemory`.

  - The maximum number of operations that can be performed per transaction is set using the configuration parameters `MaxNoOfConcurrentOperations` and `MaxNoOfLocalOperations`.

  Note

Bulk loading, `TRUNCATE TABLE`, and `ALTER TABLE` are handled as special cases by running multiple transactions, and so are not subject to this limitation.

  - Different limits related to tables and indexes. For example, the maximum number of ordered indexes per table is determined by `MaxNoOfOrderedIndexes`.

- **Node and data object maximums.** The following limits apply to numbers of cluster nodes and metadata objects:

  - The maximum number of data nodes is 48.

    A data node must have a node ID in the range of 1 to 48, inclusive. (Previous to MySQL Cluster NDB 6.1.1, management and API nodes were restricted to the range 1 to 63 inclusive as a node ID; starting with MySQL Cluster NDB 6.1.1, management and API nodes may use node IDs in the range 1 to 255, inclusive.)

  - Prior to MySQL Cluster NDB 6.1.1, the total maximum number of nodes in a MySQL Cluster was 63. Beginning with MySQL Cluster NDB 6.1.1, the total maximum number of nodes in a MySQL Cluster is 255. In either case, this number includes all SQL nodes (MySQL Servers), API nodes (applications accessing the cluster other than MySQL servers), data nodes, and management servers.

  - The maximum number of metadata objects in current versions of MySQL Cluster is 20320. This limit is hard-coded.

  See Section 12.11, "Previous MySQL Cluster Issues Resolved in MySQL 5.1 and MySQL Cluster NDB 6.x", for more information.

12.3. Limits Relating to Transaction Handling in MySQL Cluster

A number of limitations exist in MySQL Cluster with regard to the handling of transactions. These include the following:

- **Transaction isolation level.** The `NDBCLUSTER` storage engine supports only the `READ COMMITTED` transaction isolation level. (`InnoDB`, for example, supports `READ COMMITTED`, `READ UNCOMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`.) See Section 7.3.4, "MySQL Cluster Backup Troubleshooting", for information on how this can affect backing up and restoring Cluster databases.)

  Important

  If a `SELECT` from a Cluster table includes a `BLOB` or `TEXT` column, the `READ COMMITTED` transaction isolation level is converted to a read with read lock. This is done to guarantee consistency, due to the fact that parts of the values stored in columns of these types are actually read from a separate table.

- **Transactions and memory usage.** As noted elsewhere in this chapter, MySQL Cluster does not handle large transactions well; it is better to perform a number of small transactions with a few operations each than to attempt a single large transaction containing a great many operations. Among other considerations, large transactions require very large amounts of memory. Because of this, the transactional behaviour of a number of MySQL statements is effected as described in the following list:

  - `TRUNCATE` is not transactional when used on `NDB` tables. If a `TRUNCATE` fails to empty the table, then it must be re-run until it is successful.

  - `DELETE FROM` (even with no `WHERE` clause) *is* transactional. For tables containing a great many rows, you may find that performance is improved by using several `DELETE FROM ... LIMIT ...` statements to "chunk" the delete operation. If your objective is to empty the table, then you may wish to use `TRUNCATE` instead.

  - **`LOAD DATA` statements.** `LOAD DATA INFILE` is not transactional when used on `NDB` tables.

    Important

    When executing a `LOAD DATA INFILE` statement, the `NDB` engine performs commits at irregular intervals that enable better utilization of the communication network. It is not possible to know ahead of time when such commits take place.

    `LOAD DATA FROM MASTER` is not supported in MySQL Cluster.

  - **`ALTER TABLE` and transactions.** When copying an `NDB` table as part of an `ALTER TABLE`, the creation of the copy is non-transactional. (In any case, this operation is rolled back when the copy is deleted.)

- **Transactions and the `COUNT()` function.** When using MySQL Cluster Replication, it is not possible to guarantee the transactional consistency of the `COUNT()` function on the slave. In other words, when performing on the master a series of statements (`INSERT`, `DELETE`, or both) that changes the number of rows in a table within a single transaction, executing `SELECT`

`COUNT(*) FROM table` queries on the slave may yield intermediate results. This is due to the fact that `SELECT COUNT(...)` may perform dirty reads, and is not a bug in the `NDB` storage engine. (See Bug#31321 for more information.)

12.4. MySQL Cluster Error Handling

Starting, stopping, or restarting a node may give rise to temporary errors causing some transactions to fail. These include the following cases:

- **Temporary errors.** When first starting a node, it is possible that you may see Error 1204 `TEMPORARY FAILURE, DISTRIBUTION CHANGED` and similar temporary errors.
- **Errors due to node failure.** The stopping or failure of any data node can result in a number of different node failure errors. (However, there should be no aborted transactions when performing a planned shutdown of the cluster.)

In either of these cases, any errors that are generated must be handled within the application. This should be done by retrying the transaction.

See also Section 12.2, “Limits and Differences of MySQL Cluster from Standard MySQL Limits”.

12.5. Limits Associated with Database Objects in MySQL Cluster

Some database objects such as tables and indexes have different limitations when using the `NDBCLUSTER` storage engine:

- **Identifiers.** Database names, table names and attribute names cannot be as long in `NDB` tables as when using other table handlers. Attribute names are truncated to 31 characters, and if not unique after truncation give rise to errors. Database names and table names can total a maximum of 122 characters. In other words, the maximum length for an `NDB` table name is 122 characters, less the number of characters in the name of the database of which that table is a part.
- **Table names containing special characters.** `NDB` tables whose names contain characters other than letters, numbers, dashes, and underscores and which are created on one SQL node were not always discovered correctly by other SQL nodes. (Bug#31470)

  > **Note**
  >
  > This issue was fixed in MySQL 5.1.23, MySQL Cluster NDB 6.2.7, and MySQL Cluster NDB 6.3.4.

- **Number of database objects.** The maximum number of *all* `NDB` database objects in a single MySQL Cluster — including databases, tables, and indexes — is limited to 20320.
- **Attributes per table.** The maximum number of attributes (that is, columns and indexes) per table is limited to 128.
- **Attributes per key.** The maximum number of attributes per key is 32.
- **Row size.** The maximum permitted size of any one row is 8KB. Note that each `BLOB` or `TEXT` column contributes 256 + 8 = 264 bytes towards this total.

12.6. Unsupported or Missing Features in MySQL Cluster

A number of features supported by other storage engines are not supported for `NDB` tables. Trying to use any of these features in MySQL Cluster does not cause errors in or of itself; however, errors may occur in applications that expects the features to be supported or enforced:

- **Foreign key constraints.** The foreign key construct is ignored, just as it is in `MyISAM` tables.
- **`OPTIMIZE` operations.** `OPTIMIZE` operations are not supported.

  Beginning with MySQL Cluster NDB 6.3.7, this limitation has been lifted. See Section 12.11, “Previous MySQL Cluster Issues Resolved in MySQL 5.1 and MySQL Cluster NDB 6.x”, for more information.
- **`LOAD TABLE ... FROM MASTER`.** `LOAD TABLE ... FROM MASTER` is not supported.
- **Savepoints and rollbacks.** Savepoints and rollbacks to savepoints are ignored as in `MyISAM`.

- **Durability of commits.** There are no durable commits on disk. Commits are replicated, but there is no guarantee that logs are flushed to disk on commit.
- **Replication.** Statement-based replication is not supported. Use --binlog-format=ROW (or --binlog-format=MIXED) when setting up cluster replication. See Chapter 9, *MySQL Cluster Replication*, for more information.

> **Note**
>
> See Section 12.3, "Limits Relating to Transaction Handling in MySQL Cluster", for more information relating to limitations on transaction handling in `NDB`.

12.7. Limitations Relating to Performance in MySQL Cluster

The following performance issues are specific to or especially pronounced in MySQL Cluster:

- **Range scans.** There are query performance issues due to sequential access to the `NDB` storage engine; it is also relatively more expensive to do many range scans than it is with either `MyISAM` or `InnoDB`.
- **Reliability of `Records in range`.** The `Records in range` statistic is available but is not completely tested or officially supported. This may result in non-optimal query plans in some cases. If necessary, you can employ `USE INDEX` or `FORCE INDEX` to alter the execution plan. See Index Hint Syntax, for more information on how to do this.
- **Unique hash indexes.** Unique hash indexes created with `USING HASH` cannot be used for accessing a table if `NULL` is given as part of the key.

12.8. Issues Exclusive to MySQL Cluster

The following are limitations specific to the `NDBCLUSTER` storage engine:

- **Machine architecture.** All machines used in the cluster must have the same architecture. That is, all machines hosting nodes must be either big-endian or little-endian, and you cannot use a mixture of both. For example, you cannot have a management node running on a PowerPC which directs a data node that is running on an x86 machine. This restriction does not apply to machines simply running `mysql` or other clients that may be accessing the cluster's SQL nodes.
- **Adding and dropping of data nodes.** Online adding or dropping of data nodes is not currently possible. In such cases, the entire cluster must be restarted.
- **Binary logging.** MySQL Cluster has the following limitations or restrictions with regard to binary logging:
  - `sql_log_bin` has no effect on data operations; however, it is supported for schema operations.
  - MySQL Cluster cannot produce a binlog for tables having `BLOB` columns but no primary key.
  - Only the following schema operations are logged in a cluster binlog which is *not* on the `mysqld` executing the statement:
    - `CREATE TABLE`
    - `ALTER TABLE`
    - `DROP TABLE`
    - `CREATE DATABASE` / `CREATE SCHEMA`
    - `DROP DATABASE` / `DROP SCHEMA`
    - `CREATE TABLESPACE`
    - `ALTER TABLESPACE`
    - `DROP TABLESPACE`
    - `CREATE LOGFILE GROUP`
    - `ALTER LOGFILE GROUP`
    - `DROP LOGFILE GROUP`

See also Section 12.10, “Limitations Relating to Multiple MySQL Cluster Nodes”.

12.9. Limitations Relating to MySQL Cluster Disk Data Storage

- Disk data objects are subject to the following maximums:
  - Maximum number of tablespaces: 2^32 (4294967296)
  - Maximum number of data files per tablespace: 2^16 (65535)
  - The theoretical maximum number of extents per tablespace data file is 2^16 (65525); however, for practical purposes, the recommended maximum number of extents per data file is 2^8 (32768).
  - Maximum data file size: The theoretical limit is 64G; however, in MySQL 5.1, the practical upper limit is 32G. This is equivalent to 32768 extents of 1M each.

    The minimum and maximum possible sizes of extents for tablespace data files are 32K and 2G, respectively. See `CREATE TABLESPACE` Syntax, for more information.
- Use of Disk Data tables is not supported when running the cluster in diskless mode. Beginning with MySQL 5.1.12, it is disallowed altogether. (Bug#20008)

12.10. Limitations Relating to Multiple MySQL Cluster Nodes

Multiple SQL nodes. The following are issues relating to the use of multiple MySQL servers as MySQL Cluster SQL nodes, and are specific to the `NDBCLUSTER` storage engine:

- **No distributed table locks.** A `LOCK TABLES` works only for the SQL node on which the lock is issued; no other SQL node in the cluster “sees” this lock. This is also true for a lock issued by any statement that locks tables as part of its operations. (See next item for an example.)
- **`ALTER TABLE` operations.** `ALTER TABLE` is not fully locking when running multiple MySQL servers (SQL nodes). (As discussed in the previous item, MySQL Cluster does not support distributed table locks.)

Multiple management nodes. When using multiple management servers:

- You must give nodes explicit IDs in connectstrings because automatic allocation of node IDs does not work across multiple management servers.
- You must take extreme care to have the same configurations for all management servers. No special checks for this are performed by the cluster.

Multiple network addresses. Multiple network addresses per data node are not supported. Use of these is liable to cause problems: In the event of a data node failure, an SQL node waits for confirmation that the data node went down but never receives it because another route to that data node remains open. This can effectively make the cluster inoperable.

> **Note**
>
> It is possible to use multiple network hardware *interfaces* (such as Ethernet cards) for a single data node, but these must be bound to the same address. This also means that it not possible to use more than one `[tcp]` section per connection in the `config.ini` file. See Section 3.4.8, “MySQL Cluster TCP/IP Connections”, for more information.

12.11. Previous MySQL Cluster Issues Resolved in MySQL 5.1 and MySQL Cluster NDB 6.x

A number of limitations and related issues existing in earlier versions of MySQL Cluster have been resolved:

- **Variable-length column support.** The `NDBCLUSTER` storage engine now supports variable-length column types for in-memory tables.

Previously, for example, any Cluster table having one or more VARCHAR fields which contained only relatively small values, much more memory and disk space were required when using the NDBCLUSTER storage engine than would have been the case for the same table and data using the MyISAM engine. In other words, in the case of a VARCHAR column, such a column required the same amount of storage as a CHAR column of the same size. In MySQL 5.1, this is no longer the case for in-memory tables, where storage requirements for variable-length column types such as VARCHAR and BINARY are comparable to those for these column types when used in MyISAM tables (see Data Type Storage Requirements).

> **Important**
>
> For MySQL Cluster Disk Data tables, the fixed-width limitation continues to apply. See Chapter 10, *MySQL Cluster Disk Data Tables*.

- **Replication with MySQL Cluster.** It is now possible to use MySQL replication with Cluster databases. For details, see Chapter 9, *MySQL Cluster Replication*.

  Circular Replication. Circular replication is also supported with MySQL Cluster, beginning with MySQL 5.1.18. See Section 9.10, "MySQL Cluster Replication — Multi-Master and Circular Replication".

- **auto_increment_increment and auto_increment_offset.** The auto_increment_increment and auto_increment_offset server system variables are supported for Cluster replication beginning with MySQL 5.1.20, MySQL Cluster NDB 6.2.5, and MySQL Cluster 6.3.2.

- **Database autodiscovery and online schema changes.** Autodiscovery of databases is now supported for multiple MySQL servers accessing the same MySQL Cluster. Formerly, autodiscovery in MySQL Cluster 5.1 and MySQL Cluster NDB 6.x releases required that a given mysqld was already running and connected to the cluster at the time that the database was created on a different mysqld — in other words, when a mysqld process connected to the cluster after a database named *db_name* was created, it was necessary to issue a CREATE DATABASE *db_name* or CREATE SCHEMA *db_name* statement on the "new" MySQL server when it first accesseed that MySQL Cluster. Beginning with MySQL Cluster NDB 6.2.16 and MySQL Cluster NDB 6.3.18, such a CREATE statement is no longer required. (Bug#39612)

  This also means that online schema changes in NDB tables are now possible. That is, the result of operations such as ALTER TABLE and CREATE INDEX performed on one SQL node in the cluster are now visible to the cluster's other SQL nodes without any additional action being taken.

- **Backup and restore between architectures.** Beginning with MySQL 5.1.10, it is possible to perform a Cluster backup and restore between different architectures. Previously — for example — you could not back up a cluster running on a big-endian platform and then restore from that backup to a cluster running on a little-endian system. (Bug#19255)

- **Character set directory.** Beginning with MySQL 5.1.10, it is possible to install MySQL with Cluster support to a non-default location and change the search path for font description files using either the --basedir or --character-sets-dir options. (Previously, ndbd in MySQL 5.1 searched only the default path — typically /usr/local/mysql/share/mysql/charsets — for character sets.)

- **Multiple management servers.** In MySQL 5.1 (including all MySQL Cluster NDB 6.x versions), it is no longer necessary, when running multiple management servers, to restart all the cluster's data nodes to enable the management nodes to see one another.

  Also, when using multiple management servers and starting concurrently several API nodes (possibly including one or more SQL nodes) whose connectstrings listed the management servers in different order, it was possible for 2 API nodes to be assigned the same node ID. This issue is resolved in MySQL Cluster NDB 6.2.17, 6.3.23, and 6.4.3. (Bug#42973)

- **Multiple data node processes per host.** Beginning with MySQL Cluster NDB 6.2.0, you can use multiple data node processes on a single host. (In MySQL Cluster NDB 6.1, MySQL 5.1, and earlier release series, we did not support production MySQL Cluster deployments in which more than one ndbd process was run on a single physical machine.)

- **Length of CREATE TABLE statements.** CREATE TABLE statements may be no more than 4096 characters in length. *This limitation affects MySQL 5.1.6, 5.1.7, and 5.1.8 only*. (See Bug#17813)

- **IGNORE and REPLACE functionality.** In MySQL 5.1.7 and earlier, INSERT IGNORE, UPDATE IGNORE, and REPLACE were supported only for primary keys, but not for unique keys. It was possible to work around this issue by removing the constraint, then dropping the unique index, performing any inserts, and then adding the unique index again.

  This limitation was removed for INSERT IGNORE and REPLACE in MySQL 5.1.8. (See Bug#17431.)

- **AUTO_INCREMENT columns.** In MySQL 5.1.10 and earlier versions, the maximum number of tables having AUTO_INCREMENT columns — including those belonging to hidden primary keys — was 2048.

  This limitation was lifted in MySQL 5.1.11.

- **Maximum number of cluster nodes.** Prior to MySQL Cluster NDB 6.1.1, the total maximum number of nodes in a MySQL Cluster was 63, including all SQL nodes (MySQL Servers), API nodes (applications accessing the cluster other than MySQL servers), data nodes, and management servers.

  Starting with MySQL Cluster NDB 6.1.1, the total maximum number of nodes in a MySQL Cluster is 255, including all SQL nodes (MySQL Servers), API nodes (applications accessing the cluster other than MySQL servers), data nodes, and management servers. The total number of data nodes and management nodes beginning with this version is 63, of which up to 48 can be data nodes.

  > **Note**
  >
  > The limitation that a data node cannot have a node ID greater than 49 continues to apply.

- **Recovery of memory from deleted rows.** Beginning with MySQL Cluster NDB 6.3.7, memory can be reclaimed from an `NDB` table for reuse with any `NDB` table by employing `OPTIMIZE TABLE`, subject to the following limitations:

  - Only in-memory tables are supported; the `OPTIMIZE TABLE` statement still has no effect on MySQL Cluster Disk Data tables.

  - Only variable-length columns (such as those declared as `VARCHAR`, `TEXT`, or `BLOB`) are supported.

    However, you can force columns defined using fixed-length data types (such as `CHAR`) to be dynamic using the `ROW_FORMAT` or `COLUMN_FORMAT` option with a `CREATE TABLE` or `ALTER TABLE` statement.

    See `CREATE TABLE` Syntax, and `ALTER TABLE` Syntax, for information on these options.

  You can regulate the effects of `OPTIMIZE` on performance by adjusting the value of the global system variable `ndb_optimization_delay`, which sets the number of milliseconds to wait between batches of rows being processed by `OPTIMIZE`. The default value is 10 milliseconds. It is possible to set a lower value (to a minimum of `0`), but not recommended. The maximum is 100000 milliseconds (that is, 100 seconds).

- **Rollbacks.** Prior to MySQL Cluster NDB 6.3.19, the `NDBCLUSTER` storage engine did not support partial transactions or partial rollbacks of transactions. A duplicate key or similar error aborted the entire transaction, and subsequent statements raised `ERROR 1296 (HY000): Got error 4350 'Transaction already aborted' from NDBCLUSTER`. In such cases, it was necessary to issue an explicit `ROLLBACK` and retry the entire transaction.

  Beginning with MySQL Cluster NDB 6.3.19, this limitation has been removed, and the behavior of `NDBCLUSTER` is now in line with that of other transactional storage engines such as `InnoDB` which can roll back individual statements. (Bug#32656)

- **Number of tables.** Previously, the maximum number of `NDBCLUSTER` tables in a single MySQL Cluster was 1792, but this is no longer the case in MySQL 5.1 (including MySQL Cluster NDB 6.x). However, the number of tables is still included in the total maximum number of `NDBCLUSTER` database objects (20320). (See Section 12.5, “Limits Associated with Database Objects in MySQL Cluster”.)

- **DDL operations.** Beginning with MySQL Cluster NDB 6.4.0, DDL operations (such as `CREATE TABLE` or `ALTER TABLE`) are protected from data node failures. Previously, if a data node failed while trying to perform one of these, the data dictionary became locked and no further DDL statements could be executed without restarting the cluster (Bug#36718).

Chapter 13. MySQL Cluster Development Roadmap

In this section, we discuss changes in the implementation of MySQL Cluster in MySQL 5.1 and MySQL Cluster NDB 6.x as compared to MySQL 5.0.

We also discuss our roadmap for further improvements to MySQL Cluster planned for MySQL Cluster NDB 7.0 and later.

There are a number of significant changes in the implementation of the `NDBCLUSTER` storage engine in mainline MySQL 5.1 releases up to and including MySQL 5.1.23 as compared to that in MySQL 5.0; MySQL Cluster NDB makes further changes and improvements in MySQL Cluster in addition to these. The changes and features most likely to be of interest are shown in the following table:

MySQL 5.1 (through 5.1.23)
MySQL Cluster Replication
Disk Data storage
Variable-size columns
User-defined partitioning
Autodiscovery of table schema changes
Online adding and dropping of indexes

MySQL Cluster NDB 6.1
Greater number of cluster nodes
Disabling of arbitration
Additional `DUMP` commands
Faster Disk Data backups
Batched slave updates

MySQL Cluster NDB 6.2
Improved backup status reporting (`BackupReportFrequency`, `REPORT BackupStatus`)
Multiple connections per SQL node
Data access with `NdbRecord` (NDB API)
`REPORT MemoryUsage` command
Memory allocation improvements
Management client connection control
Micro-GCPs
Online `ADD COLUMN`; improved online index creation

MySQL Cluster NDB 6.3
Conflict detection and resolution for multi-master replication
Compressed backups and local checkpoints
Support for `OPTIMIZE TABLE`
Parallel data node recovery
Enhanced transaction coordinator selection
Improved SQL statement performance metrics
Transaction batching
`ndb_restore` attribute promotion
Support for `epoll` (Linux only)
Distribution awareness
`NDB` thread locks; realtime extensions for multiple CPUs

MySQL Cluster NDB 7.0
Multi-threaded data nodes (`ndbmtd` data node daemon)
Online addition of data nodes; online data redistribution
MySQL on Windows (alpha; source releases only)
Configuration cache
Backup snapshots (`START BACKUP ... SNAPSHOTSTART`, `START BACKUP ... SNAPSHOTEND` commands)
IPv6 support for geo-replication
Protected DDL operations
Dynamic buffering for NDB transporters
`NDB$INFO` meta-information database (in development)

13.1. Features Added in MySQL 5.1 Cluster

A number of new features for MySQL Cluster have been implemented in MySQL 5.1 through MySQL 5.1.23, when support for MySQL Cluster was moved to MySQL Cluster NDB. All of the features in the following list are also available in all MySQL Cluster NDB (6.1 and later) releases.

- **Integration of MySQL Cluster into MySQL Replication.** MySQL Cluster Replication makes it possible to replicate from one MySQL Cluster to another. Updates on any SQL node (MySQL server) in the cluster acting as the master are replicated to the slave cluster; the state of the slave side remains consistent with the cluster acting as the master. This is sometimes referred to as *asynchronous replication* between clusters, providing *geographic redundancy*. It is also possible to replicate from a MySQL Cluster acting as the master to a standalone MySQL server acting as the slave, or from a standalone MySQL master server to to a slave cluster; in either of these cases, the standalone MySQL server uses a storage engine other than `NDB-CLUSTER`. Multi-master replication setups such as circular replication are also supported.

  See Chapter 9, *MySQL Cluster Replication*.

- **Support for storage of rows on disk.** Storage of `NDBCLUSTER` table data on disk is now supported. Indexed columns, including the primary key hash index, must still be stored in RAM; however, all other columns can be stored on disk.

  See Chapter 10, *MySQL Cluster Disk Data Tables*.

- **Variable-size columns.** In MySQL 5.0, an `NDBCLUSTER` table column defined as `VARCHAR(255)` used 260 bytes of storage independent of what was stored in any particular record. In MySQL 5.1 Cluster tables, only the portion of the column actually taken up by the record is stored. This makes possible a significant reduction in space requirements for such columns as compared to previous release series — by a factor of up to 5 in many cases.

- **User-defined partitioning.** Users can define partitions based on columns that are part of the primary key. It is possible to partition `NDB` tables based on `KEY` and `LINEAR KEY` schemes. This feature is also available for many other MySQL storage engines, which support additional partitioning types that are not available with `NDBCLUSTER` tables.

  For additional general information about user-defined partitioning in MySQL 5.1, see Partitioning. Specifics of partitioning types are discussed in Partition Types.

  The MySQL Server can also determine whether it is possible to “prune away” some of the partitions from the `WHERE` clause, which can greatly speed up some queries. See Partition Pruning, for information about designing tables and queries to take advantage of partition pruning.

- **Autodiscovery of table schema changes.** In MySQL 5.0, it was necessary to issue a `FLUSH TABLES` statement or a “dummy” `SELECT` in order for new `NDBCLUSTER` tables or changes made to schemas of existing `NDBCLUSTER` tables on one SQL node to be visible on the cluster's other SQL nodes. In MySQL 5.1, this is no longer necessary; new Cluster tables and changes in the definitions of existing `NDBCLUSTER` tables made on one SQL node are immediately visible to all SQL nodes connected to the cluster.

  > **Note**
  >
  > When creating a new database, it is still necessary in MySQL 5.1 to issue a `CREATE DATABASE` or `CREATE SCHEMA` statement on each SQL node in the cluster.

- **Distribution awareness (NDB API).** *Distribution awareness* is a mechanism by which the best data node is automatically selected to be queried for information. (Conceptually, it is similar in some ways to partition pruning (see Partition Pruning). To take advantage of distribution awareness, you should do the following:

1. Determine which table column is most likely to be used for finding matching records.

2. Make this column part of the table's primary key.

3. Explicitly partition the table by KEY, using this column as the table' partitioning key.

Following these steps causes records with the same value for the partitioning column to be stored on the same partition (that is, in the same node group). When reading data, transactions are begun on the data node actually having the desired rows instead of this node being determined by the usual round-robin mechanism.

> **Important**
>
> In order to see a measureable impact on performance, the cluster must have at least four data nodes, since, with only two data nodes, both data nodes have exactly the same data.

Using distribution awareness can yield performance increase of as great as 45% when using four data nodes, and possibly more when using a greater number of data nodes.

> **Note**
>
> In mainline MySQL 5.1 releases, distribution awareness was supported only when using the NDB API; support was added for SQL and API nodes in MySQL Cluster NDB 6.3 (see Section 13.4, "Features Added in MySQL Cluster NDB 6.3", which includes an example showing how to create a table in order to take advantage of distribution awareness).

See Section 12.11, "Previous MySQL Cluster Issues Resolved in MySQL 5.1 and MySQL Cluster NDB 6.x", for more information.

13.2. Features Added in MySQL Cluster NDB 6.1

The following list provides an overview of significant feature additions and changes made in MySQL Cluster NDB 6.1. All of the changes in this list are also available in MySQL Cluster NDB 6.2 and 6.3 releases. For detailed information about all changes made in MySQL Cluster NDB 6.1, see Changes in MySQL Cluster NDB 6.1.

- **Increased number of cluster nodes.** The maximum number of all nodes in a MySQL Cluster has been increased to 255. For more information, see Section 12.10, "Limitations Relating to Multiple MySQL Cluster Nodes".
- **Disabling arbitration.** It is now possible to disable arbitration by setting `ArbitrationRank=0` on all cluster management and SQL nodes. For more information, see *Defining the Management Server: `ArbitrationRank`* and *Defining SQL and Other API Nodes: `ArbitrationRank`*.
- **Additional `DUMP` commands.** New management client `DUMP` commands provide help with tracking transactions, scan operations, and locks. See Section 7.2, "Commands in the MySQL Cluster Management Client", and `DUMP` Commands, for more information.
- **Faster Disk Data backups.** Improvements in backups of Disk Data tables can yield a 10 to 15% increase in backup speed of Disk Data tables.
- **Batched slave updates.** Batching of updates on cluster replication slaves, enabled using the `--slave-allow-batching` option for `mysqld`, increases replication efficiency. For more information, see Section 9.6, "Starting MySQL Cluster Replication (Single Replication Channel)".

13.3. Features Added in MySQL Cluster NDB 6.2

The following list provides an overview of significant feature additions and changes made in MySQL Cluster NDB 6.2. All of the changes in this list are also available in MySQL Cluster NDB 6.3 . For more detailed information about all feature changes and bugfixes made in MySQL Cluster NDB 6.2, see Changes in MySQL Cluster NDB 6.2.

- **Enhanced backup status reporting.** Backup status reporting has been improved, aided in part by the introduction of a `BackupReportFrequency` configuration parameter; see *Defining Data Nodes: `BackupReportFrequency`*, for more information.
- **Multiple cluster connections per SQL node.** A single MySQL server acting as a MySQL Cluster SQL node can employ multiple connections to the cluster using the `--ndb-cluster-connection-pool` startup option for `mysqld`. This option is described in *MySQL Cluster-Related Command Options for `mysqld`: `--ndb-cluster-connection-pool` option*.

- **New data access interface.** The `NdbRecord` interface provides a new and simplified data handler for use in NDB API applications. See The `NdbRecord` Interface, for more information.

- **New reporting commands.** The new management client `REPORT BackupStatus` and `REPORT MemoryUsage` commands provide better access to information about the status of MySQL Cluster backups and how much memory is being used by MySQL Cluster for data and index storage. See Section 7.2, "Commands in the MySQL Cluster Management Client", for more information about the `REPORT` commands. In addition, in-progress status reporting is provided by the `ndb_restore` utility; see Section 6.17, "`ndb_restore` — Restore a MySQL Cluster Backup".

- **Improved memory allocation and configuration.** Memory is now allocated by the `NDB` kernel to tables on a page-by-page basis, which significantly reduces the memory overhead required for maintaining `NDBCLUSTER` tables. In addition, the `MaxAllocate` configuration parameter now makes it possible to set the maximum size of the allocation unit used for table memory; for more information about this configuration parameter, see *Defining Data Nodes: `MaxAllocate`*.

- **Choice of fixed-width or variable-width columns.** You can control whether fixed-width or variable-width storage is used for a given column of an `NDB` table by employing of the `COLUMN_FORMAT` specifier as part of the column's definition in a `CREATE TABLE` or `ALTER TABLE` statement. In addition, the ability to control whether a given column of an `NDB` table is stored in memory or on disk, using the `STORAGE` specifier as part of the column's definition in a `CREATE TABLE` or `ALTER TABLE` statement. For more information, see `CREATE TABLE` Syntax, and `ALTER TABLE` Syntax.

- **Controlling management client connections.** The `--bind-address` cluster management server startup option makes it possible to restrict management client connections to `ndb_mgmd` to a single host (IP address or host name) and port, which can make MySQL Cluster management operations more secure. For more information about this option, see Section 6.24.3, "Program Options for `ndb_mgmd`".

- **Micro-GCPs.** Due to a change in the protocol for handling of global checkpoints (GCPs handled in this manner sometimes being referred to as "micro-GCPs"), it is now possible to control how often the GCI number is updated, and how often global checkpoints are written to disk, using the `TimeBetweenEpochs` configuration parameter. This improves the reliability and performance of MySQL Cluster Replication. For more information, see *Defining Data Nodes: `TimeBetweenEpochs`* and *Defining Data Nodes: `TimeBetweenEpochsTimeout`*.

- **Core online schema change support.** Support for the online `ALTER TABLE` operations `ADD COLUMN`, `ADD INDEX`, and `DROP INDEX` is available. When the `ONLINE` keyword is used, the `ALTER TABLE` is non-copying, which means that indexes do not have to be re-created, which provides these benefits:

  - Single user mode is no longer required for `ALTER TABLE` operations that can be performed online.

  - Transactions can continue during `ALTER TABLE` operations that can be performed online.

  - Tables being altered online are not locked against access by other SQL nodes.

    However, such tables are locked against other operations on the *same* SQL node for the duration of the `ALTER TABLE`. We are working to overcome this limitation in a future MySQL Cluster release.

  Online `CREATE INDEX` and `DROP INDEX` statements are also supported. Online changes can be suppressed using the `OFFLINE` key word. See `ALTER TABLE` Syntax, `CREATE INDEX` Syntax, and `DROP INDEX` Syntax, for more detailed information.

- **`mysql.ndb_binlog_index` improvements.** More information has been added to the `mysql.ndb_binlog_index` table so that it is possible to determine which originating epochs have been applied inside an epoch. This is particularly useful for 3-way replication. See Section 9.4, "MySQL Cluster Replication Schema and Tables", for more information.

- **Epoch lag control.** The `MaxBufferedEpochs` data node configuration parameter provides a means to control the maximum number of unprocessed epochs by which a subscribing node can lag. Subscribers which exceed this number are disconnected and forced to reconnect. For a discussion of this configuration parameter, see *Defining Data Nodes: `MaxBufferedEpochs`*.

- **Fully automatic database discovery.** It is no longer a requirement for database autodiscovery that an SQL node already be connected to the cluster at the time that a database is created on another SQL node, or for a `CREATE DATABASE` or `CREATE SCHEMA` statement to be issued on the new SQL node after it joins the cluster.

- **Multiple data node processes per host.** In earlier MySQL Cluster release series, we did not support MySQL Cluster deployments in production where more than one `ndbd` process was run on a single physical machine. However, beginning with MySQL Cluster NDB 6.2.0, you can use multiple data node processes on a single host.

  > **Note**
  >
  > A multi-threaded version of `ndbd` tailored for use on hosts with multiple CPUs or cores was introduced in MySQL Cluster NDB 7.0. See Section 13.5, "Features Added in MySQL Cluster NDB 7.0", and Section 6.3, "`ndbmtd` — The MySQL Cluster Data Node Daemon (Multi-Threaded)", for more information.

- **Improved Disk Data filesystem configuration.** As of MySQL Cluster NDB 6.2.17, you can specify default locations for MySQL Cluster Disk Data data files and undo log files using the data node configuration parameters `FileSystemPathDD`, `FileSystemPathDataFiles`, and `FileSystemPathUndoFiles`. For more information, see *Disk Data filesystem parameters*.

- **Automatic creation of Disk Data log file groups and tablespaces.** Beginning with MySQL Cluster NDB 6.2.17, using the data node configuration parameters `InitialLogFileGroup` and `InitialTablespace`, you can cause the creation of a MySQL Cluster Disk Data log file group, tablespace, or both, when the cluster is first started. When using these parameters, no SQL statements are required to create these Disk Data objects. For more information, see *Disk Data object creation parameters*.

13.4. Features Added in MySQL Cluster NDB 6.3

The following list provides an overview of significant feature additions and changes first made in MySQL Cluster NDB 6.3. For more detailed information about all feature changes and bugfixes made in MySQL Cluster NDB 6.3, see Changes in MySQL Cluster NDB 6.3.

- **Conflict detection and resolution.** It is now possible to detect and resolve conflicts that arise in multi-master replication scenarios, such as circular replication, when different masters may try to update the same row on the slave with different data. Both "greatest timestamp wins" and "same timestamp wins" scenarios are supported. For more information, see Section 9.11, "MySQL Cluster Replication Conflict Resolution".

- **Recovery of "one master, many slaves" replication setups.** Recovery of multi-way replication setups ("one master, many slaves") is now supported via the `--ndb-log-orig` server option and changes in the `mysql.ndb_binlog_index` table. See Section 9.4, "MySQL Cluster Replication Schema and Tables", for more information.

- **Enhanced selection options for transaction coordinator.** New values and behaviors are introduced for `--ndb_optimized_node_selection` allowing for greater flexibility when an SQL node chooses a transaction coordinator. For more information, see the description of `ndb_optimized_node_selection` in Section 4.3, "MySQL Cluster System Variables".

- **Replication heartbeats.** Replication heartbeats facilitate the task of monitoring and detecting failures in master-slave connections in real time. This feature is implemented via a new `MASTER_HEARTBEAT_PERIOD = value` clause for the `CHANGE MASTER TO` statement and the addition of two status variables `Slave_heartbeat_period` and `Slave_received_heartbeats`. For more information, see `CHANGE MASTER TO` Syntax.

- **`NDB` thread locks.** It is possible to lock `NDB` execution threads and maintenance threads (such as file system and other operating system threads) to specific CPUs on multiprocessor data node hosts, and to leverage real-time scheduling.

- **Improved performance of updates using primary keys or unique keys.** The number of unnecessary reads when performing a primary key or unique key update has been greatly reduced. Since it is seldom necessary to read a record prior to an update, this can yield a considerable improvement in performance. In addition, primary key columns are no longer written to when not needed during update operations.

- **Batching improvements.** Support of batched `DELETE` and `UPDATE` operations has been significantly improved. Batching of `UPDATE WHERE...` and multiple `DELETE` operations is also now implemented.

- **Improved SQL statement performance metrics.** The `Ndb_execute_count` system status variable measures the number of round trips made by SQL statements to the `NDB` kernel, providing an improved metric for determining efficiency with which statements are excuted. For more information, see *MySQL Cluster Status Variables: `Ndb_execute_count`*.

- **Compressed LCPs and backups.** Compressed local checkpoints and backups can save 50% or more of the disk space used by uncompressed LCPs and backups. These can be enabled using the two new data node configuration parameters `CompressedLCP` and `CompressedBackup`, respectively. See *MySQL Cluster Status Variables: `CompressedBackup`*, and *MySQL Cluster Status Variables: `CompressedLCP`*, for more information about these parameters.

- **`OPTIMIZE TABLE` support with `NDBCLUSTER` tables.** `OPTIMIZE TABLE` is supported for dynamic columns of in-memory `NDB` tables. In such cases, it is no longer necessary to drop (and possibly to re-create) a table, or to perform a rolling restart, in order to recover memory from deleted rows for general re-use by Cluster. The performance of `OPTIMIZE` on Cluster tables can be tuned by adjusting the value of the `ndb_optimization_delay` system variable, which controls the number of milliseconds to wait between processing batches of rows by `OPTIMIZE TABLE`. In addition, `OPTIMIZE TABLE` on an `NDBCLUSTER` table can be interrupted by, for example, killing the SQL thread performing the `OPTIMIZE` operation.

- **Batching of transactions.** It is possible to cause statements occurring within the same transaction to be run as a batch by setting the session variable `transaction_allow_batching` to `1` or `ON`. To use this feature, `autocommit` must be set to `0` or `OFF`. Batch sizes can be controlled using the `--ndb-batch-size` option for `mysqld`. For more information, see Section 4.2, "`mysqld` Command Options for MySQL Cluster", and Section 4.3, "MySQL Cluster System Variables".

- **Attribute promotion with `ndb_restore`.** It is possible using `ndb_restore` to restore data reliably from a column of a given type to a column that uses a "larger" type. This is sometimes referred to as *attribute promotion*. For example, MySQL Cluster backup data that originated in a `SMALLINT` column can be restored to a `MEDIUMINT`, `INT`, or `BIGINT` column. See Section 6.17, "`ndb_restore` — Restore a MySQL Cluster Backup", for more information.

- **Parallel data node recovery.** Recovery of multiple data nodes can now be done in parallel, rather than sequentially. In other words, several data nodes can be restored concurrently, which can often result in much faster recovery times than when they are restored one at a time.

- **Increased local checkpoint efficiency.** Only 2 local checkpoints are stored, rather than 3, lowering disk space requirements and the size and number of redo log files.

- **`NDBCLUSTER` table persistence control.** Persistence of `NDB` tables can be controlled using the session variables `ndb_table_temporary` and `ndb_table_no_logging`. `ndb_table_no_logging` causes `NDB` tables not to be checkpointed to disk; `ndb_table_temporary` does the same, and in addition, no schema files are created. See Section 4.1, "MySQL Cluster Server Option and Variable Reference".

- **Epoll support (*Linux only*).** *Epoll* is an improved method for handling file descriptors, which is more efficient than scanning to determine whether a file descriptor has data to be read. (The term *epoll* is specific to Linux and equivalent functionality is known by other names on other platforms such as Solaris and FreeBSD.) Currently, MySQL Cluster supports this functionality on Linux only.

- **Distribution awareness (SQL nodes).** In MySQL Cluster NDB 6.3, SQL nodes can take advantage of distribution awareness. Here we provide a brief example showing how to design a table to make a given class of queries distrubtion-aware. Suppose an `NDBCLUSTER` table `t1` has the following schema:

  ```
  CREATE TABLE t1 (
      userid INT NOT NULL,
      serviceid INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      data VARCHAR(255)
  )   ENGINE=NDBCLUSTER;
  ```

  Suppose further that most of the queries to be used in our application test values of the `userid` column of this table. The form of such a query looks something like this:

  ```
  SELECT columns FROM t1
      WHERE userid relation value;
  ```

  In this query, *`relation`* represents some relational operator, such as `=`, `<`, `>`, and so on. Queries using `IN` and a list of values can also be used:

  ```
  SELECT columns FROM t1
      WHERE userid IN value_list;
  ```

  In order to make use of distribution awareness, we need to make the `userid` column part of the table's primary key, then explicitly partition the table with this column being used as the partitioning key. (Recall that for a partitioned table having one or more unique keys, all columns of the table's partitioning key must also be part of all of the unique keys — for more information and examples, see Partitioning Keys, Primary Keys, and Unique Keys.) In other words, the table schema should be equivalent to the following `CREATE TABLE` statement:

  ```
  CREATE TABLE t1 (
      userid INT NOT NULL,
      serviceid INT NOT NULL AUTO_INCREMENT,
      data VARCHAR(255),
      PRIMARY KEY p (userid,serviceid)
  )   ENGINE=NDBCLUSTER
      PARTITION BY KEY(userid);
  ```

  When the table is partitioned in this way, all rows having the same `userid` value are found on the same node group, and the MySQL Server can immediately select the optimal node to use as the transaction coordinator.

- **Realtime extensions for multiple CPUs.** When running MySQL Cluster data nodes on hosts with multiple processors, the realtime extensions make it possible to give priority to the data node process and control on which CPU cores it should operate. This can be done using the data node configuration parameters `RealtimeScheduler`, `SchedulerExecutionTimer` and `SchedulerSpinTimer`. Doing so properly can significantly lower response times and make them much more predictable response. For more information about using these parameters, see Defining Data Nodes: Realtime Performance Parameters

- **Fully automatic database discovery.** It is no longer a requirement for database autodiscovery that an SQL node already be connected to the cluster at the time that a database is created on another SQL node, or for a `CREATE DATABASE` or `CREATE SCHEMA` statement to be issued on the new SQL node after it joins the cluster.

- **Restoring specific tables and databases from a MySQL Cluster backup.** It is now possible exercise more fine-grained control when restoring a MySQL Cluster from backup using `ndb_restore`. You can restore only specified tables or databases, or exclude specific tables or databases from being restored, using the new `ndb_restore` options `--include-tables`, `--include-databases`, `--exclude-tables`, and `--exclude-databases`. For more information about these options, see Section 6.17, "`ndb_restore` — Restore a MySQL Cluster Backup".

- **Improved Disk Data filesystem configuration.** As of MySQL Cluster NDB 6.3.22, you can specify default locations for MySQL Cluster Disk Data data files and undo log files using the data node configuration parameters `FileSystemPathDD`, `FileSystemPathDataFiles`, and `FileSystemPathUndoFiles`. For more information, see *Disk Data filesystem parameters*.

- **Automatic creation of Disk Data log file groups and tablespaces.** Beginning with MySQL Cluster NDB 6.3.22, using the data node configuration parameters `InitialLogFileGroup` and `InitialTablespace`, you can cause the creation of a MySQL Cluster Disk Data log file group, tablespace, or both, when the cluster is first started. When using these parameters, no SQL statements are required to create these Disk Data objects. For more information, see *Disk Data object creation parameters*.

- **Configuration parameter data dumps.** Starting with MySQL Cluster NDB 6.3.25, the `ndb_config` utility supports a `--configinfo` option that causes it to dump a list of all configuration parameters supported by the cluster, along with brief descriptions, information about the parameters' default and allowed values, and the sections of the `config.ini` file in which the parameters apply. An additional `--xml` switch causes `ndb_config` to use XML rather than plaintext output. Using `ndb_config --configinfo` or `ndb_config --configinfo --xml` requires no access to a running MySQL Cluster, any other programs, or any files. For more information and examples, see Section 6.6, "`ndb_config` — Extract MySQL Cluster Configuration Information".

13.5. Features Added in MySQL Cluster NDB 7.0

The following list provides an overview of significant feature additions and changes made in or planned for MySQL Cluster NDB 7.0. For more detailed information about all feature changes and bugfixes made in MySQL Cluster NDB 7.0, see Changes in MySQL Cluster NDB 7.0.

> **Important**
>
> Early development versions of MySQL Cluster NDB 7.0 were known as "MySQL Cluster NDB 6.4", and the first four releases in this series were identified as MySQL Cluster NDB 6.4.0 through 6.4.3. Any information relating to these MySQL Cluster NDB 6.4.x releases appearing in this documentation apply to MySQL Cluster NDB 7.0.
>
> MySQL Cluster NDB 7.0.4 is the fifth MySQL Cluster NDB 7.0 release; it is the successor to MySQL Cluster NDB 6.4.3.

- **MySQL Cluster on Windows (*alpha*).** MySQL Cluster is now available on an experimental basis for Windows operating systems. Features and behavior comparable to those found on platforms that are already supported — such as Linux and Solaris — are planned for MySQL Cluster on Windows. Currently, you must build from source, although we intend to start making Windows binaries available in the near future. To enable MySQL Cluster support on Windows, you must configure the build using the `WITH_NDBCLUSTER_STORAGE_ENGINE` option. For more information, see Installing MySQL from Source on Windows.

- **Ability to add nodes and node groups online.** Beginning with MySQL Cluster NDB 6.4.0, it is possible to add new node groups (and thus new data nodes) to a running MySQL Cluster without shutting down and reloading the cluster. As part of enabling this feature, a new command `CREATE NODEGROUP` has been added to the cluster management client and the functionality of the `ALTER ONLINE TABLE ... REORGANIZE PARTITION` SQL statement has been extended. For more information, see Section 7.8, "Adding MySQL Cluster Data Nodes Online".

- **Data node multithreading support.** Beginning with MySQL Cluster NDB 6.4.0, a multithreaded version of the data node daemon, named `ndbmtd`, is available for use on data node hosts with multiple CPU cores. This binary is built automatically when compiling with MySQL Cluster support; no additional options other than those needed to provide MySQL Cluster support are needed when configuring the build. In most respects, `ndbmtd` functions in the same way as `ndbd`, and can use the same command-line options and configuration parameters. In addition, the new `MaxNoOfExecutionThreads` configuration parameter can be used to determine the number of data node process threads for `ndbmtd`. For more information, see Section 6.3, "`ndbmtd` — The MySQL Cluster Data Node Daemon (Multi-Threaded)".

  > **Note**
  >
  > Disk Data tables are not yet supported for use with `ndbmtd`.

- **Configuration cache.** Formerly, MySQL Cluster configuration was stateless — that is, configuration information was reloaded from the cluster's global configuration file (usually `config.ini`) each time `ndb_mgmd` was started. Beginning with MySQL Cluster NDB 6.4.0, the cluster's configuration is cached internally, and the global configuration file is no longer automatically re-read when the management server is restarted. This behavior can be controlled via the three new management server options `--configdir`, `--initial`, and `--reload`. For more information about this change, see Section 3.4, "MySQL Cluster Configuration Files". For more information about the new management server options, see Section 6.24.3, "Program Options for `ndb_mgmd`".

- **Snapshot options for backups.** Beginning with MySQL Cluster NDB 6.4.0, you can determine when performing a cluster backup whether the backup matches the state of the data when the backup was started or when it was completed, using the new options `SNAPSHOTSTART` and `SNAPSHOTEND` for the management client's `START BACKUP` command. See Section 7.3.2, "Using The MySQL Cluster Management Client to Create a Backup", for more information.

- **Dynamic NDB transporter send buffer memory allocation.** Previously, the NDB kernel used a fixed-size send buffer for every data node in the cluster, which was allocated when the node started. Because the size of this buffer could not be changed after the cluster was started, it was necessary to make it large enough in advance to accomodate the maximum possible load on any transporter socket. However, this was an inefficient use of memory, since much of it often went unused. Beginning with MySQL Cluster NDB 6.4.0, send buffer memory is allocated dynamically from a memory pool shared between all transporters, which means that the size of the send buffer can be adjusted as necessary. This change is reflected by the addition of the configuration parameters `TotalSendBufferMemory`, `ReservedSendBufferMemory`, and `OverLoadLimit`, as well as a change in how the existing `SendBufferMemory` configuration parameter is used. For more information, see Section 3.7, "Configuring MySQL Cluster Send Buffer Parameters".

- **Robust DDL operations.** Beginning with MySQL Cluster NDB 6.4.0, DDL operations (such as `CREATE TABLE` or `ALTER TABLE`) are protected from data node failures; in the event of a data node failure, such operations are now rolled back gracefully. Previously, if a data node failed while trying to perform a DDL operation, the MySQL Cluster data dictionary became locked and no further DDL statements could be executed without restarting the cluster.

- **IPv6 support in MySQL Cluster Replication.** Beginning with MySQL Cluster NDB 6.4.1, IPv6 networking is supported between MySQL Cluster SQL nodes, which makes it possible to replicate between instances of MySQL Cluster using IPv6 addresses. However, IPv6 is supported only for direct connections between MySQL servers; all connections *within* an individual MySQL Cluster must use IPv4. For more information, see Section 9.3, "Known Issues in MySQL Cluster Replication".

- **Restoring specific tables and databases from a MySQL Cluster backup.** It is now possible exercise more fine-grained control when restoring a MySQL Cluster from backup using `ndb_restore`. You can restore only specified tables or databases, or exclude specific tables or databases from being restored, using the new `ndb_restore` options `--include-tables`, `--include-databases`, `--exclude-tables`, and `--exclude-databases`. For more information about these options, see Section 6.17, "`ndb_restore` — Restore a MySQL Cluster Backup".

- **Improved Disk Data filesystem configuration.** As of MySQL Cluster NDB 6.4.3, you can specify default locations for MySQL Cluster Disk Data data files and undo log files using the data node configuration parameters `FileSystemPathDD`, `FileSystemPathDataFiles`, and `FileSystemPathUndoFiles`. For more information, see *Disk Data filesystem parameters*.

- **Automatic creation of Disk Data log file groups and tablespaces.** Beginning with MySQL Cluster NDB 6.4.3, using the data node configuration parameters `InitialLogFileGroup` and `InitialTablespace`, you can cause the creation of a MySQL Cluster Disk Data log file group, tablespace, or both, when the cluster is first started. When using these parameters, no SQL statements are required to create these Disk Data objects. For more information, see *Disk Data object creation parameters*.

- **Improved internal message passing and record handling.** MySQL Cluster NDB 7.0 contains 2 changes that optimize the use of network connections by addressing the size and number of messages passed between data nodes, and between data nodes and API nodes, which can increase MySQL Cluster and application performance:

  - **Packed reads.** Formerly, each read request signal contained a list of columns to be retrieved, each of these column identifiers using 4 bytes within the message. This meant that the message size increased as the number of columns being fetched increased. In addition, in the response from the data node, each column result was packed to a 4-byte boundary, which resulted in wasted space. In MySQL Cluster NDB 7.0, messaging for read operations is optimized in both directions, using a bitmap in the read request to specify the columns to be fetched. Where many fields are requested, this can result in a significant message size reduction as compared with the old method. In addition, the 4-byte packing in responses is no longer used, which means that smaller fields consume less space.

  - **Long signal transactions.** This enhancement reduces the number of messages and signals that are sent to data nodes for complex requests. Prior to MySQL Cluster NDB 7.0, there was a 100 byte limit on the size of the request signal, which meant that complex requests had to be split up between multiple messages prior to transmission, then reassembled on the receiving end. In addition to actual payload data, each message required its own operating system and protocol overhead such as header information. This often wasted network bandwidth and data node CPU. The maximum size of the message is now 32 KB, which is sufficient to accommodate most queries.

  Both of these optimizations are internal to the NDB API, and so is transparent to applications; this is true whether an application uses the NDB API directly or does so indirectly through an SQL node.

- **Configuration parameter data dumps.** Starting with MySQL Cluster NDB 7.0.6, the `ndb_config` utility supports a `--configinfo` option that causes it to dump a list of all configuration parameters supported by the cluster, along with brief descriptions, information about the parameters' default and allowed values, and the sections of the `config.ini` file in which the parameters apply. An additional `--xml` switch causes `ndb_config` to use XML rather than plaintext output. Using `ndb_config --configinfo` or `ndb_config --configinfo --xml` requires no access to a running MySQL Cluster, any other programs, or any files. For more information and examples, see Section 6.6, "`ndb_config` — Extract

MySQL Cluster Configuration Information”.

13.6. Features Planned for MySQL Cluster NDB 7.1 and Later

The following improvements to MySQL Cluster are planned for MySQL Cluster NDB 7.1 and later release series.

> **Important**
>
> These features are in planning or early development phase. Timing, availability, and implementation details are not guaranteed, and are subject to change at any time without notice.

- **`NDB$INFO` meta-information database.** This feature is intended to make it possible to obtain real-time characteristics of a MySQL Cluster by issuing queries from the `mysql` client or other MySQL client applications. `NDB$INFO` is a database which is planned for MySQL Cluster NDB 7.1 to provide metadata specific to MySQL Cluster similar to that provided by the `INFORMATION_SCHEMA` database for the standard MySQL Server. This would eliminate much of the need to read log files, issue `DUMP` commands, or parse the output of `ndb_config` in order to get configuration and status information from a running MySQL Cluster.

Chapter 14. MySQL Cluster Glossary

The following terms are useful to an understanding of MySQL Cluster or have specialized meanings when used in relation to it.

- **Cluster.** In its generic sense, a cluster is a set of computers functioning as a unit and working together to accomplish a single task.

  NDBCLUSTER. This is the storage engine used in MySQL to implement data storage, retrieval, and management distributed among several computers.

  MySQL Cluster. This refers to a group of computers working together using the NDB storage engine to support a distributed MySQL database in a *shared-nothing architecture* using *in-memory storage*.

- **Configuration files.** Text files containing directives and information regarding the cluster, its hosts, and its nodes. These are read by the cluster's management nodes when the cluster is started. See Section 3.4, “MySQL Cluster Configuration Files”, for details.

- **Backup.** A complete copy of all cluster data, transactions and logs, saved to disk or other long-term storage.

- **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. More specific to Cluster, it is a point in time where all committed transactions are stored on disk. With regard to the NDB storage engine, there are two types of checkpoints which work together 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, LCP's 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. The precise interval varies, and depends upon the amount of data stored by the node, the level of cluster activity, and other factors.

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

- **Cluster host.** A computer making up part of a MySQL Cluster. A cluster has both a *physical* structure and a *logical* structure. Physically, the cluster consists of a number of computers, known as *cluster hosts* (or more simply as *hosts*. See also **Node** and **Node group** below.

- **Node.** This refers to a logical or functional unit of MySQL Cluster, and is sometimes also referred to as a *cluster node*. In the context of MySQL Cluster, we use the term “node” to indicate a *process* rather than a physical component of the cluster. There are three node types required to implement a working MySQL Cluster:

  - **Management nodes.** Manages the other nodes within the MySQL Cluster. It provides configuration data to the other nodes; starts and stops nodes; handles network partitioning; creates backups and restores from them, and so forth.

  - **SQL nodes.** Instances of MySQL Server which serve as front ends to data kept in the cluster's **data nodes**. Clients desiring to store, retrieve, or update data can access an SQL node just as they would any other MySQL Server, employing the usual MySQL authentication methods and APIs; the underlying distribution of data between node groups is transparent to users and applications. SQL nodes access the cluster's databases as a whole without regard to the data's distribution across different data nodes or cluster hosts.

  - **Data nodes.** These nodes store the actual data. Table data fragments are stored in a set of node groups; each node group stores a different subset of the table data. Each of the nodes making up a node group stores a replica of the fragment for which that node group is responsible. Currently, a single cluster can support up to 48 data nodes total.

  It is possible for more than one node to co-exist on a single machine. (In fact, it is even possible to set up a complete cluster on one machine, although one would almost certainly *not* want to do this in a production environment.) It may be helpful to remember that, when working with MySQL Cluster, the term *host* refers to a physical component of the cluster whereas a *node* is a logical or functional component (that is, a process).

  Note Regarding Terms. In older versions of the MySQL Cluster documentation, data nodes were sometimes referred to as “database nodes”. The term “storage nodes” has also been used. In addition, SQL nodes were sometimes known as “client nodes”. This older terminology has been deprecated to minimize confusion, and for this reason should be avoided. They are also often referred to as “API nodes” — an SQL node is actually an API node that provides an SQL interface to the cluster.

- **Node group.** A set of data nodes. All data nodes in a node group contain the same data (fragments), and all nodes in a single group should reside on different hosts. It is possible to control which nodes belong to which node groups.

  For more information, see Section 1.2, “MySQL Cluster Nodes, Node Groups, Replicas, and Partitions”.

- **Node failure.** MySQL Cluster is not solely dependent upon the functioning of any single node making up the cluster; the cluster can continue to run if one or more nodes fail. The precise number of node failures that a given cluster can tolerate depends upon the number of nodes and the cluster's configuration.
- **Node restart.** The process of restarting a failed cluster node.
- **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 reinitializing its state from disk logs and checkpoints. This is required after either a planned or an unplanned shutdown of the cluster.
- **Fragment.** A portion of a database table; in the `NDB` storage engine, a table is broken up into and stored as a number of fragments. A fragment is sometimes also called a “partition”; however, “fragment” is the preferred term. Tables are fragmented in MySQL Cluster in order to facilitate load balancing between machines and nodes.
- **Replica.** Under the `NDB` storage engine, each table fragment has number of replicas stored on other data nodes in order to provide redundancy. Currently, there may be up four replicas per fragment.
- **Transporter.** A protocol providing data transfer between nodes. MySQL Cluster currently supports four different types of transporter connections:
  - **TCP/IP.** This is, of course, the familiar network protocol that underlies HTTP, FTP (and so on) on the Internet. TCP/IP can be used for both local and remote connections.
  - **SCI.** **S**calable **C**oherent **I**nterface is a high-speed protocol used in building multiprocessor systems and parallel-processing applications. Use of SCI with MySQL Cluster requires specialized hardware, as discussed in Section 11.1, “Configuring MySQL Cluster to use SCI Sockets”. For a basic introduction to SCI, see this essay at dolphinics.com.
  - **SHM.** Unix-style **sh**ared **m**emory segments. Where supported, SHM is used automatically to connect nodes running on the same host. The Unix man page for `shmop(2)` is a good place to begin obtaining additional information about this topic.

  > **Note**
  >
  > The cluster transporter is internal to the cluster. Applications using MySQL Cluster communicate with SQL nodes just as they do with any other version of MySQL Server (via TCP/IP, or through the use of Unix socket files or Windows named pipes). Queries can be sent and results retrieved using the standard MySQL client APIs.
- **`NDB`.** This stands for **N**etwork **D**ata**b**ase, and refers to the storage engine used to enable MySQL Cluster. The `NDB` storage engine supports all the usual MySQL data types and SQL statements, and is ACID-compliant. This engine also provides full support for transactions (commits and rollbacks).
- **Shared-nothing architecture.** The ideal architecture for a MySQL Cluster. In a true shared-nothing setup, each node runs on a separate host. The advantage such an arrangement is that there no single host or node can act as single point of failure or as a performance bottle neck for the system as a whole.
- **In-memory storage.** All data stored in each data node is kept in memory on the node's host computer. For each data node in the cluster, you must have available an amount of RAM equal to the size of the database times the number of replicas, divided by the number of data nodes. Thus, if the database takes up 1GB of memory, and you want to set up the cluster with four replicas and eight data nodes, a minimum of 500MB memory will be required per node. Note that this is in addition to any requirements for the operating system and any other applications that might be running on the host.

  In MySQL 5.1 and MySQL Cluster NDB 6.x, it is also possible to create *Disk Data* tables where non-indexed columns are stored on disk, thus reducing the memory footprint required by the cluster. Note that indexes and indexed column data are still stored in RAM. See Chapter 10, *MySQL Cluster Disk Data Tables*.
- **Table.** As is usual in the context of a relational database, the term “table” denotes a set of identically structured records. In MySQL Cluster, a database table is stored in a data node as a set of fragments, each of which is replicated on additional data nodes. The set of data nodes replicating the same fragment or set of fragments is referred to as a *node group*.
- **Cluster programs.** These are command-line programs used in running, configuring, and administering MySQL Cluster. They include both server daemons:
  - `ndbd`:

    The data node daemon (runs a data node process)
  - `ndb_mgmd`:

The management server daemon (runs a management server process)

and client programs:

- `ndb_mgm`:

  The management client (provides an interface for executing management commands)

- `ndb_waiter`:

  Used to verify status of all nodes in a cluster

- `ndb_restore`:

  Restores cluster data from backup

For more about these programs and their uses, see Chapter 6, *MySQL Cluster Programs*.

- **Event log.** MySQL Cluster logs events by category (startup, shutdown, errors, checkpoints, and so on), priority, and severity. A complete listing of all reportable events may be found in Section 7.4, "Event Reports Generated in MySQL Cluster". Event logs are of two types:

  - **Cluster log.** Keeps a record of all desired reportable events for the cluster as a whole.

  - **Node log.** A separate log which is also kept for each individual node.

  Under normal circumstances, it is necessary and sufficient to keep and examine only the cluster log. The node logs need be consulted only for application development and debugging purposes.

- **Angel process.** When a data node is started, `ndbd` actually starts two processes. One of these is known as the "angel" process; its purpose is to check to make sure that the main `ndbd` process continues to run, and to restart the main process if it should stop for any reason.

- **Watchdog thread.** Each `ndbd` process has an internal *watchdog thread* which monitors the main worker thread, ensuring forward progress and a timely response to cluster protocols such as the cluster heartbeat. If the `ndbd` process is not being woken up promptly by the operating system when its sleep time expires, `INFO` and `WARNING` events, which are identifiable because they contain "Watchdog:...", are written to the cluster log. Such messages are usually a symptom of an overloaded system; you should see what else is running on the system, and whether the ndbd `process` is being swapped out to disk. If `ndbd` cannot wake up regularly then it cannot respond to heartbeat messages on time, and other nodes eventually consider it "dead" due to the missed heartbeats, causing it to be excluded from the cluster.

Chapter 15. MySQL 5.1 FAQ — MySQL Cluster

In the following section, we answer questions that are frequently asked about MySQL Cluster and the NDBCLUSTER storage engine.

Questions

Questions and Answers

15.1: Which versions of the MySQL software support Cluster? Do I have to compile from source?

Beginning with MySQL 4.1.3, MySQL Cluster is supported in all `MySQL-Max` server binaries in the 5.1 release series for operating systems on which MySQL Cluster is available. See `mysqld`. You can determine whether your server has `NDB` support using either either of the statements `SHOW VARIABLES LIKE 'have_%'` or `SHOW ENGINES`.

You can also obtain `NDB` support by compiling MySQL from source, but it is not necessary to do so simply to use MySQL Cluster. To download the latest binary, RPM, or source distribution in the MySQL 5.1 series, visit http://dev.mysql.com/downloads/mysql/5.1.html.

However, you should use MySQL NDB Cluster NDB 6.2 or 6.3 for new deployments, and if you are already using MySQL 5.1 with clustering support, to upgrade to one of these MySQL Cluster NDB 6.x release series. For an overview of improvements made in MySQL Cluster NDB 6.2 and 6.3, see *MySQL 5.1 Manual: Features Added in MySQL Cluster NDB 6.2*, and *MySQL 5.1 Manual: Features Added in MySQL Cluster NDB 6.3*.

15.2: What does “NDB” mean?

This stands for “**N**etwork **D**ata**b**ase”. `NDB` (also known as `NDBCLUSTER`) is the storage engine that enables clustering in MySQL.

15.3: What is the difference between using MySQL Cluster *vs* using MySQL replication?

In traditional MySQL replication, a master MySQL server updates one or more slaves. Transactions are committed sequentially, and a slow transaction can cause the slave to lag behind the master. This means that if the master fails, it is possible that the slave might not have recorded the last few transactions. If a transaction-safe engine such as `InnoDB` is being used, a transaction will either be complete on the slave or not applied at all, but replication does not guarantee that all data on the master and the slave will be consistent at all times. In MySQL Cluster, all data nodes are kept in synchrony, and a transaction committed by any one data node is committed for all data nodes. In the event of a data node failure, all remaining data nodes remain in a consistent state.

In short, whereas standard MySQL replication is *asynchronous*, MySQL Cluster is *synchronous*.

We have implemented (asynchronous) replication for Cluster in MySQL 5.1 and MySQL Cluster NDB 6.x. This includes the capability to replicate both between two clusters, and from a MySQL cluster to a non-Cluster MySQL server. However, we do not plan to backport this functionality to MySQL 5.1. See *MySQL 5.1 Manual: MySQL Cluster Replication*, for more information.

15.4: Do I need to do any special networking to run MySQL Cluster? How do computers in a cluster communicate?

MySQL Cluster is intended to be used in a high-bandwidth environment, with computers connecting via TCP/IP. Its performance depends directly upon the connection speed between the cluster's computers. The minimum connectivity requirements for MySQL Cluster include a typical 100-megabit Ethernet network or the equivalent. We recommend you use gigabit Ethernet whenever available.

The faster SCI protocol is also supported, but requires special hardware. See Chapter 11, *Using High-Speed Interconnects with MySQL Cluster*, for more information about SCI.

15.5: How many computers do I need to run a MySQL Cluster, and why?

A minimum of three computers is required to run a viable cluster. However, the minimum **recommended** number of computers in a MySQL Cluster is four: one each to run the management and SQL nodes, and two computers to serve as data nodes. The purpose of the two data nodes is to provide redundancy; the management node must run on a separate machine to guarantee continued arbitration services in the event that one of the data nodes fails.

To provide increased throughput and high availability, you should use multiple SQL nodes (MySQL Servers connected to the cluster). It is also possible (although not strictly necessary) to run multiple management servers.

15.6: What do the different computers do in a MySQL Cluster?

A MySQL Cluster has both a physical and logical organization, with computers being the physical elements. The logical or functional elements of a cluster are referred to as *nodes*, and a computer housing a cluster node is sometimes referred to as a *cluster host*. There are three types of nodes, each corresponding to a specific role within the cluster. These are:

- **Management node.** This node provides management services for the cluster as a whole, including startup, shutdown, backups, and configuration data for the other nodes. The management node server is implemented as the application `ndb_mgmd`; the management client used to control MySQL Cluster is `ndb_mgm`.
- **Data node.** This type of node stores and replicates data. Data node functionality is handled by instances of the `NDB` data node process `ndbd`.
- **SQL node.** This is simply an instance of MySQL Server (`mysqld`) that is built with support for the `NDBCLUSTER` storage engine and started with the `--ndb-cluster` option to enable the engine and the `--ndb-connectstring` option to enable it to connect to a MySQL Cluster management server. For more about these options, see Section 4.2, "`mysqld` Command Options for MySQL Cluster".

> **Note**
>
> An *API node* is any application that makes direct use of Cluster data nodes for data storage and retrieval. An SQL node can thus be considered a type of API node that uses a MySQL Server to provide an SQL interface to the Cluster. You can write such applications (that do not depend on a MySQL Server) using the NDB API, which supplies a direct, object-oriented transaction and scanning interface to Cluster data; see The NDB API, for more information.

15.7: When I run the `SHOW` command in the MySQL Cluster management client, I see a line of output that looks like this:

```
id=2    @10.100.10.32  (Version: 5.1.34-ndb-6.3.26, Nodegroup: 0, Master)
```

What is a "master node", and what does it do? How do I configure a node so that it is the master?

The simplest answer is, "It's not something you can control, and it's nothing that you need to worry about in any case, unless you're a software engineer writing or analyzing the MySQL Cluster source code".

If you don't find that answer satisfactory, here's a longer and more technical version:

A number of mechanisms in MySQL Cluster require distributed coordination among the data nodes. These distributed algorithms and protocols include global checkpointing, DDL (schema) changes, and node restart handling. To make this coordination simpler, the data nodes "elect" one of their number to be a "master". There is no user-facing mechanism for influencing this selection, which is is completely automatic; the fact that it *is* automatic is a key part of MySQL Cluster's internal architecture.

When a node acts as a master for any of these mechanisms, it is usually the point of coordination for the activity, and the other nodes act as "servants", carrying out their parts of the activity as directed by the master. If the node acting as master fails, then the remaining nodes elect a new master. Tasks in progress that were being coordinated by the old master may either fail or be continued by the new master, depending on the actual mechanism involved.

It is possible for some of these different mechanisms and protocols to have different master nodes, but in general the same master is chosen for all of them. The node indicated as the master in the output of `SHOW` in the management client is actually the `DICT` master (see The `DBDICT` Block, in the *MySQL Cluster API Developer Guide*, for more information), responsible for coordinating DDL and metadata activity.

MySQL Cluster is designed in such a way that the choice of master has no discernable effect outside the cluster itself. For example, the current master does not have significantly higher CPU or resource usage than the other data nodes, and failure of the master should not have a significantly different impact on the cluster than the failure of any other data node.

15.8: With which operating systems can I use Cluster?

MySQL Cluster is supported on most Unix-like operating systems, including Linux, Mac OS X, Solaris, and HP-UX. MySQL Cluster is *not* supported on Windows at this time. We are working to add MySQL Cluster support for other platforms, including Windows; eventually we intend to offer MySQL Cluster on all platforms for which MySQL itself is supported.

For more detailed information concerning the level of support which is offered for MySQL Cluster on various operating system versions, OS distributions, and hardware platforms, please refer to http://www.mysql.com/support/supportedplatforms/cluster.html.

15.9: What are the hardware requirements for running MySQL Cluster?

MySQL Cluster should run on any platform for which `NDB`-enabled binaries are available. For data nodes, faster CPUs and more memory are likely to improve performance, and 64-bit CPUs are likely to be more effective than 32-bit processors. There must be sufficient memory on machines used for data nodes to hold each node's share of the database (see *How much RAM do I Need?* for

more information). Nodes can communicate via a standard TCP/IP network and hardware. For SCI support, special networking hardware is required (see Chapter 11, *Using High-Speed Interconnects with MySQL Cluster*).

15.10: How much RAM do I need to use MySQL Cluster? Is it possible to use disk memory at all?

In MySQL 5.1, Cluster is in-memory only. This means that all table data (including indexes) is stored in RAM. Therefore, if your data takes up 1 GB of space and you want to replicate it once in the cluster, you need 2 GB of memory to do so (1 GB per replica). This is in addition to the memory required by the operating system and any applications running on the cluster computers.

If a data node's memory usage exceeds what is available in RAM, then the system will attempt to use swap space up to the limit set for `DataMemory`. However, this will at best result in severely degraded performance, and may cause the node to be dropped due to slow response time (missed heartbeats). We do not recommend on relying on disk swapping in a production environment for this reason. In any case, once the `DataMemory` limit is reached, any operations requiring additional memory (such as inserts) will fail.

(We have implemented disk data storage for MySQL Cluster in MySQL 5.1, including MySQL Cluster NDB 6.2 and 6.3, but we have no plans to add this capability in MySQL 5.1. See *MySQL 5.1 Manual: MySQL Cluster Disk Data Tables*, for more information.)

You can use the following formula for obtaining a rough estimate of how much RAM is needed for each data node in the cluster:

```
(SizeofDatabase × NumberOfReplicas × 1.1 ) / NumberOfDataNodes
```

To calculate the memory requirements more exactly requires determining, for each table in the cluster database, the storage space required per row (see Data Type Storage Requirements, for details), and multiplying this by the number of rows. You must also remember to account for any column indexes as follows:

- Each primary key or hash index created for an `NDBCLUSTER` table requires 21–25 bytes per record. These indexes use `IndexMemory`.
- Each ordered index requires 10 bytes storage per record, using `DataMemory`.
- Creating a primary key or unique index also creates an ordered index, unless this index is created with `USING HASH`. In other words:
  - A primary key or unique index on a Cluster table normally takes up 31 to 35 bytes per record.
  - However, if the primary key or unique index is created with `USING HASH`, then it requires only 21 to 25 bytes per record.

Note that creating MySQL Cluster tables with `USING HASH` for all primary keys and unique indexes will generally cause table updates to run more quickly — in some cases by a much as 20 to 30 percent faster than updates on tables where `USING HASH` was not used in creating primary and unique keys. This is due to the fact that less memory is required (because no ordered indexes are created), and that less CPU must be utilized (because fewer indexes must be read and possibly updated). However, it also means that queries that could otherwise use range scans must be satisfied by other means, which can result in slower selects.

When calculating Cluster memory requirements, you may find useful the `ndb_size.pl` utility which is available in recent MySQL 5.1 releases. This Perl script connects to a current (non-Cluster) MySQL database and creates a report on how much space that database would require if it used the `NDBCLUSTER` storage engine. For more information, see Section 6.21, “`ndb_size.pl` — NDBCLUSTER Size Requirement Estimator”.

It is especially important to keep in mind that *every MySQL Cluster table must have a primary key*. The `NDB` storage engine creates a primary key automatically if none is defined, and this primary key is created without `USING HASH`.

There is no easy way to determine exactly how much memory is being used for storage of Cluster indexes at any given time; however, warnings are written to the Cluster log when 80% of available `DataMemory` or `IndexMemory` is in use, and again when use reaches 85%, 90%, and so on.

15.11: What file systems can I use with MySQL Cluster? What about network file systems or network shares?

Generally, any file system that is native to the host operating system should work well with MySQL Cluster. If you find that a given file system works particularly well (or not so especially well) with MySQL Cluster, we invite you to discuss your findings in the MySQL Cluster Forums.

We do not test MySQL Cluster with `FAT` or `VFAT` file systems on Linux. Because of this, and due to the fact that these are not very useful for any purpose other than sharing disk partitions between Linux and Windows operating systems on multi-boot computers, we do not recommend their use with MySQL Cluster.

MySQL Cluster is implemented as a shared-nothing solution; the idea behind this is that the failure of a single piece of hardware should not cause the failure of multiple cluster nodes, or possibly even the failure of the cluster as a whole. For this reason, the use of network shares or network file systems is not supported for MySQL Cluster. This also applies to shared storage devices such as

SANs.

15.12: Can I run MySQL Cluster nodes inside virtual machines (such as those created by VMWare, Parallels, or Xen)?

This is possible but not recommended for a production environment.

We have found that running MySQL Cluster processes inside a virtual machine can give rise to issues with timing and disk subsystems that have a strong negative impact on the operation of the cluster. The behavior of the cluster is often unpredictable in these cases.

If an issue can be reproduced outside the virtual environment, then we may be able to provide assistance. Otherwise, we cannot support it at this time.

15.13: I am trying to populate a MySQL Cluster database. The loading process terminates prematurely and I get an error message like this one: `ERROR 1114: THE TABLE 'MY_CLUSTER_TABLE' IS FULL` Why is this happening?

The cause is very likely to be that your setup does not provide sufficient RAM for all table data and all indexes, *including the primary key required by the `NDB` storage engine and automatically created in the event that the table definition does not include the definition of a primary key*.

It is also worth noting that all data nodes should have the same amount of RAM, since no data node in a cluster can use more memory than the least amount available to any individual data node. For example, if there are four computers hosting Cluster data nodes, and three of these have 3GB of RAM available to store Cluster data while the remaining data node has only 1GB RAM, then each data node can devote at most 1GB to MySQL Cluster data and indexes.

15.14: MySQL Cluster uses TCP/IP. Does this mean that I can run it over the Internet, with one or more nodes in remote locations?

It is *very* unlikely that a cluster would perform reliably under such conditions, as MySQL Cluster was designed and implemented with the assumption that it would be run under conditions guaranteeing dedicated high-speed connectivity such as that found in a LAN setting using 100 Mbps or gigabit Ethernet — preferably the latter. We neither test nor warrant its performance using anything slower than this.

Also, it is extremely important to keep in mind that communications between the nodes in a MySQL Cluster are not secure; they are neither encrypted nor safeguarded by any other protective mechanism. The most secure configuration for a cluster is in a private network behind a firewall, with no direct access to any Cluster data or management nodes from outside. (For SQL nodes, you should take the same precautions as you would with any other instance of the MySQL server.) For more information, see Chapter 8, *MySQL Cluster Security Issues*.

15.15: Do I have to learn a new programming or query language to use MySQL Cluster?

No. Although some specialized commands are used to manage and configure the cluster itself, only standard (My)SQL queries and commands are required for the following operations:

- Creating, altering, and dropping tables
- Inserting, updating, and deleting table data
- Creating, changing, and dropping primary and unique indexes

Some specialized configuration parameters and files are required to set up a MySQL Cluster — see Section 3.4, “MySQL Cluster Configuration Files”, for information about these.

A few simple commands are used in the MySQL Cluster management client (`ndb_mgm`) for tasks such as starting and stopping cluster nodes. See Section 7.2, “Commands in the MySQL Cluster Management Client”.

15.16: How do I find out what an error or warning message means when using MySQL Cluster?

There are two ways in which this can be done:

- From within the `mysql` client, use `SHOW ERRORS` or `SHOW WARNINGS` immediately upon being notified of the error or warning condition. Errors and warnings also be displayed in MySQL Query Browser.
- From a system shell prompt, use `perror --ndb error_code`.

15.17: Is MySQL Cluster transaction-safe? What isolation levels are supported?

Yes: For tables created with the NDB storage engine, transactions are supported. Currently, MySQL Cluster supports only the READ COMMITTED transaction isolation level.

15.18: What storage engines are supported by MySQL Cluster?

Clustering with MySQL is supported only by the NDB storage engine. That is, in order for a table to be shared between nodes in a MySQL Cluster, the table must be created using ENGINE=NDB (or the equivalent option ENGINE=NDBCLUSTER).

It is possible to create tables using other storage engines (such as MyISAM or InnoDB) on a MySQL server being used with a MySQL Cluster, but these non-NDB tables do *not* participate in clustering; they are strictly local to the individual MySQL server instance on which they are created.

15.19: In the event of a catastrophic failure — say, for instance, the whole city loses power and my UPS fails — would I lose all my data?

All committed transactions are logged. Therefore, although it is possible that some data could be lost in the event of a catastrophe, this should be quite limited. Data loss can be further reduced by minimizing the number of operations per transaction. (It is not a good idea to perform large numbers of operations per transaction in any case.)

15.20: Is it possible to use FULLTEXT indexes with MySQL Cluster?

FULLTEXT indexing is not supported by any storage engine other than MyISAM. We are working to add this capability to MySQL Cluster tables in a future release.

15.21: Can I run multiple nodes on a single computer?

It is possible but not advisable. One of the chief reasons to run a cluster is to provide redundancy. To obtain the full benefits of this redundancy, each node should reside on a separate machine. If you place multiple nodes on a single machine and that machine fails, you lose all of those nodes. Given that MySQL Cluster can be run on commodity hardware loaded with a low-cost (or even no-cost) operating system, the expense of an extra machine or two is well worth it to safeguard mission-critical data. It also worth noting that the requirements for a cluster host running a management node are minimal. This task can be accomplished with a 200 MHz Pentium CPU and sufficient RAM for the operating system plus a small amount of overhead for the ndb_mgmd and ndb_mgm processes.

It is acceptable to run multiple cluster data nodes on a single host for learning about MySQL Cluster, or for testing purposes; however, this is not generally supported for production use.

15.22: Can I add data nodes to a MySQL Cluster without restarting it?

Not at present. A rolling restart is all that is required for adding new management or SQL nodes to a MySQL Cluster (see Section 5.1, “Performing a Rolling Restart of a MySQL Cluster”). Adding data nodes is more complex, and requires the following steps:

1. Make a complete backup of all Cluster data.
2. Completely shut down the cluster and all cluster node processes.
3. Restart the cluster, using the --initial startup option for all instances of ndbd.

   > **Warning**
   >
   > Never use the --initial when starting ndbd except when necessary to clear the data node file system. See Section 6.24.2, “Program Options for ndbd and ndbmtd”, for information about when this is required.

4. Restore all cluster data from the backup.

In a future MySQL Cluster release series, we hope to implement a “hot” reconfiguration capability for MySQL Cluster to minimize (if not eliminate) the requirement for restarting the cluster when adding new nodes. However, this is not planned for MySQL 5.1.

15.23: Are there any limitations that I should be aware of when using MySQL Cluster?

Limitations on NDB tables in MySQL 5.1 include the following:

- Temporary tables are not supported; a CREATE TEMPORARY TABLE statement using ENGINE=NDB or ENGINE=NDBCLUSTER fails with an error.
- FULLTEXT indexes and index prefixes are not supported. Only complete columns may be indexed.

- In MySQL 5.1, MySQL Cluster does not support spatial data types or spatial indexes. See Spatial Extensions.
- Only complete rollbacks for transactions are supported. Partial rollbacks and rollbacks to savepoints are not supported. A failed insert due to a duplicate key or similar error causes a transaction to abort; when this occurs, you must issue an explicit `ROLLBACK` and retry the transaction.
- The maximum number of attributes allowed per table is 128, and attribute names cannot be any longer than 31 characters. For each table, the maximum combined length of the table and database names is 122 characters.
- The maximum size for a table row is 8 kilobytes, not counting `BLOB` values. There is no set limit for the number of rows per table. Table size limits depend on a number of factors, in particular on the amount of RAM available to each data node.
- The `NDB` engine does not support foreign key constraints. As with `MyISAM` tables, if these are specified in a `CREATE TABLE` or `ALTER TABLE` statement, they are ignored.

For a complete listing of limitations in MySQL Cluster, see Chapter 12, *Known Limitations of MySQL Cluster*.

15.24: How do I import an existing MySQL database into a MySQL Cluster?

You can import databases into MySQL Cluster much as you would with any other version of MySQL. Other than the limitations mentioned elsewhere in this FAQ, the only other special requirement is that any tables to be included in the cluster must use the `NDB` storage engine. This means that the tables must be created with `ENGINE=NDB` or `ENGINE=NDBCLUSTER`.

It is also possible to convert existing tables using other storage engines to `NDBCLUSTER` using one or more `ALTER TABLE` statement. However, the definition of the table must be compatible with the `NDBCLUSTER` storage engine prior to making the conversion. In MySQL 5.1, an additional workaround is also required.

See Chapter 12, *Known Limitations of MySQL Cluster*, for details.

15.25: How do cluster nodes communicate with one another?

Cluster nodes can communicate via any of three different transport mechanisms: TCP/IP, SHM (shared memory), and SCI (Scalable Coherent Interface). Where available, SHM is used by default between nodes residing on the same cluster host; however, this is considered experimental. SCI is a high-speed (1 gigabit per second and higher), high-availability protocol used in building scalable multi-processor systems; it requires special hardware and drivers. See Chapter 11, *Using High-Speed Interconnects with MySQL Cluster*, for more about using SCI as a transport mechanism for MySQL Cluster.

15.26: What is an *arbitrator*?

If one or more nodes in a cluster fail, it is possible that not all cluster nodes will be able to “see” one another. In fact, it is possible that two sets of nodes might become isolated from one another in a network partitioning, also known as a “split brain” scenario. This type of situation is undesirable because each set of nodes tries to behave as though it is the entire cluster.

When cluster nodes go down, there are two possibilities. If more than 50% of the remaining nodes can communicate with each other, we have what is sometimes called a “majority rules” situation, and this set of nodes is considered to be the cluster. The arbitrator comes into play when there is an even number of nodes: in such cases, the set of nodes to which the arbitrator belongs is considered to be the cluster, and nodes not belonging to this set are shut down.

The preceding information is somewhat simplified. A more complete explanation taking into account node groups follows:

When all nodes in at least one node group are alive, network partitioning is not an issue, because no one portion of the cluster can form a functional cluster. The real problem arises when no single node group has all its nodes alive, in which case network partitioning (the “split-brain” scenario) becomes possible. Then an arbitrator is required. All cluster nodes recognize the same node as the arbitrator, which is normally the management server; however, it is possible to configure any of the MySQL Servers in the cluster to act as the arbitrator instead. The arbitrator accepts the first set of cluster nodes to contact it, and tells the remaining set to shut down. Arbitrator selection is controlled by the `ArbitrationRank` configuration parameter for MySQL Server and management server nodes. (See Section 3.4.5, “Defining a MySQL Cluster Management Server”, for details.)

The role of arbitrator does not in and of itself impose any heavy demands upon the host so designated, and thus the arbitrator host does not need to be particularly fast or to have extra memory especially for this purpose.

15.27: What data types are supported by MySQL Cluster?

In MySQL 5.1, MySQL Cluster supports all of the usual MySQL data types, except for those associated with MySQL's spatial extensions. (Spatial data types and spatial indexes are supported only by `MyISAM`; see Spatial Extensions, for more information.) In addition, there are some differences with regard to indexes when used with `NDB` tables.

> **Note**
>
> MySQL Cluster tables (that is, tables created with `ENGINE=NDBCLUSTER`) have only fixed-width rows. This means

that (for example) each record containing a `VARCHAR(255)` column will require space for 255 characters (as required for the character set and collation being used for the table), regardless of the actual number of characters stored therein. This issue is expected to be fixed in a future MySQL release series.

See Chapter 12, *Known Limitations of MySQL Cluster*, for more information about these issues.

15.28: How do I start and stop MySQL Cluster?

It is necessary to start each node in the cluster separately, in the following order:

1. Start the management node, using the `ndb_mgmd` command.

   You must include the `-f` or `--config-file` option to tell the management node where its configuration file can be found.

2. Start each data node with the `ndbd` command.

   Each data node must be started with the `-c` or `--connect-string` option so that the data node knows how to connect to the management server.

3. Start each MySQL Server (SQL node) using your preferred startup script, such as `mysqld_safe`.

   Each MySQL Server must be started with the `--ndbcluster` and `--ndb-connectstring` options. These options cause mysqld to enable `NDBCLUSTER` storage engine support and how to connect to the management server.

Each of these commands must be run from a system shell on the machine housing the affected node. (You do not have to be physically present at the machine — a remote login shell can be used for this purpose.) You can verify that the cluster is running by starting the `NDB` management client `ndb_mgm` on the machine housing the management node and issuing the `SHOW` or `ALL STATUS` command.

To shut down a running cluster, issue the command `SHUTDOWN` in the management client. Alternatively, you may enter the following command in a system shell:

```
shell> ndb_mgm -e "SHUTDOWN"
```

(The quotation marks are optional; in addition, the `SHUTDOWN` command is not case-sensitive.)

Either of these commands causes the `ndb_mgm`, `ndb_mgm`, and any `ndbd` processes to terminate gracefully. MySQL servers running as Cluster SQL nodes can be stopped using `mysqladmin shutdown`.

For more information, see Section 7.2, “Commands in the MySQL Cluster Management Client”, and Section 2.6, “Safe Shutdown and Restart of MySQL Cluster”.

15.29: What happens to MySQL Cluster data when the cluster is shut down?

The data that was held in memory by the cluster's data nodes is written to disk, and is reloaded into memory the next time that the cluster is started.

15.30: Is it a good idea to have more than one management node for a MySQL Cluster?

It can be helpful as a fail-safe. Only one management node controls the cluster at any given time, but it is possible to configure one management node as primary, and one or more additional management nodes to take over in the event that the primary management node fails.

See Section 3.4, “MySQL Cluster Configuration Files”, for information on how to configure MySQL Cluster management nodes.

15.31: Can I mix different kinds of hardware and operating systems in one MySQL Cluster?

Yes, as long as all machines and operating systems have the same “endianness” (all big-endian or all little-endian). We are working to overcome this limitation in a future MySQL Cluster release.

It is also possible to use software different MySQL Cluster releases on different nodes. However, we support this only as part of a rolling upgrade procedure (see Section 5.1, “Performing a Rolling Restart of a MySQL Cluster”).

15.32: Can I run two data nodes on a single host? Two SQL nodes?

Yes, it is possible to do this. In the case of multiple data nodes, it is advisable (but not required) for each node to use a different data directory. If you want to run multiple SQL nodes on one machine, each instance of `mysqld` must use a different TCP/IP port. However, running more than one cluster node of a given type per machine is generally not encouraged or supported for production use.

We also advise against running data nodes and SQL nodes together on the same host, since the `ndbd` and `mysqld` processes may compete for memory.

15.33: Can I use host names with MySQL Cluster?

Yes, it is possible to use DNS and DHCP for cluster hosts. However, if your application requires “five nines” availability, we recommend using fixed (numeric) IP addresses. Making communication between Cluster hosts dependent on services such as DNS and DHCP introduces additional potential points of failure.

15.34: How do I handle MySQL users in a MySQL Cluster having multiple MySQL servers?

MySQL user accounts and privileges are not automatically propagated between different MySQL servers accessing the same MySQL Cluster. Therefore, you must make sure that these are copied between the SQL nodes yourself. You can do this manually, or automate the task with scripts.

> **Warning**
>
> Do not attempt to work around this issue by converting the MySQL system tables to use the `NDBCLUSTER` storage engine. Only the `MyISAM` storage engine is supported for these tables.

15.35: How do I continue to send queries in the event that one of the SQL nodes fails?

MySQL Cluster does not provide any sort of automatic failover between SQL nodes. Your application must be prepared to handlethe loss of SQL nodes and to fail over between them.