SYBASE®

Query Processor

Adaptive Server® Enterprise

Version 15.0

DOCUMENT ID: DC00385-01-1500-03

LAST REVISED: October 2005

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book ..... **ix**

CHAPTER 1 **Understanding Query Processing in Adaptive Server** **1**
- Query optimizer 3
  - Factors analyzed in optimizing queries 6
  - Transformations for query optimization 8
  - Handling search arguments and useful indexes 12
  - Handling joins 14
- Optimization goals 16
  - Exceptions 17
  - Limiting the time spent optimizing a query 17
- Parallelism 18
- Optimization issues 18
- Lava query execution engine 21
  - Lava query plans 22

CHAPTER 2 **Parallel Query Processing** **31**
- Vertical, horizontal, and pipelined parallelism 31
- Queries that benefit from parallel processing 32
- Enabling parallelism 33
  - Setting the number of worker processes 33
  - Setting max parallel degree 34
  - Setting max resource granularity 34
  - Setting max repartition degree 35
  - Setting max scan parallel degree 35
- Controlling parallelism at the session level 36
  - set command examples 36
- Controlling parallelism for a query 37
  - Query level parallel clause examples 38
- When parallel query results differ 38
  - Queries that use set rowcount 39
  - Queries that set local variables 39
- Understanding Parallel Query Plans 40

Adaptive Server's parallel query execution model 42
exchange operator 42
Using parallelism in SQL operations 47
Partition elimination 90
Partition skew 91
Why queries do not run in parallel 92
Run time adjustment 92
Recognizing and managing run time adjustments 93

CHAPTER 3 Using showplan 95
Displaying the query plan 95
Query Plans in ASE 15.0 96
Statement level output 96
Lava Query Plan shape 100
Lava operators 103
Emit operator 104
Scan operator 104
From cache 104
From or list 104
from table 106
Union Operators 140
hash union 140
merge union 141
union all operator 141
scalaragg operator 143
restrict Operator 144
sort operator 144
store operator 146
sequencer operator 148
remscan operator 151
scroll operator 151
ridjoin operator 152
sqfilter operator 152
exchange operator 154

CHAPTER 4 Displaying Query Optimization Strategies And Estimates 157
Set commands for text format messages 157
Set commands for XML format messages 158
Usage scenarios 160
Permissions for Set commands 163
Discontinued tracing commands 163

CHAPTER 5 Query Processing Metrics 165
What are query processing metrics? 165
Executing QP metrics 166
Accessing metrics 166
Using metrics 166
Should I use QP metrics or monitoring tables? 167
sysquerymetrics view 167
Examples 168
Clearing the metrics 170

CHAPTER 6 Abstract Plans 171
New operators and syntax 172
New directives and syntax 175
Optimization goal 175
Optimization timeout limit 175
Support for pre-15.0 operators 176
A complex query example 176
Semantics 177
Worktables and steps 177
Syntactic qualification 178
Legacy partial plans 179

CHAPTER 7 Using Statistics To Improve Performance 181
Statistics maintained in Adaptive Server 181
Definitions 182
Importance of statistics 182
Updating statistics 183
Adding statistics for unindexed columns 183
update statistics commands 184
Using sampling for update statistics 185
Automatically updating statistics 187
What is the datachange function? 188
Configuring automatic update statistics 190
Using Job Scheduler to update statistics 190
Examples of updating statistics with datachange 192
Column statistics and statistics maintenance 193
Creating and updating column statistics 194
When additional statistics may be useful 195
Adding statistics for a column with update statistics 195
Adding statistics for minor columns with update index statistics .. 196
Adding statistics for all columns with update all statistics 196
Choosing step numbers for histograms 196

Disadvantages of too many steps 197
Choosing a step number 197
Scan types, sort requirements, and locking 198
Sorts for unindexed or non leading columns 198
Locking, scans, and sorts during update index statistics 199
Locking, scans and sorts during update all statistics 199
Using the with consumers clause 199
Reducing update statistics impact on concurrent processes 199
Using the delete statistics command 200
When row counts may be inaccurate 201

APPENDIX A **Abstract Plan Specifications 203**
delete 204
distinct 205
distinct_hashing 208
distinct_sorted 210
distinct_sorting 212
enforce 214
group 215
group_hashing 217
group_sorted 218
h_join 220
h_union_distinct 221
hints 223
insert 224
join 225
m_join 227
m_union_all 229
m_union_distinct 231
nl_join 233
rep_xchg 235
scalar_agg 236
sequence 238
sort 240
store 242
store_index 243
union 245
union_all 247
update 249
use optgoal 250
use opttimeoutlimit 251
values 252
xchg 253

Glossary **255**

Index **265**

About This Book

Audience

This book is for use by System Administrators and Database Administrators.

How to use this book

This book describes the Query Processor in Adaptive Server Enterprise and how it is used to optimize query processing in Adaptive Server.

- Chapter 1, “Understanding Query Processing in Adaptive Server” describes enhancements to the query processor for Adaptive Server Enterprise.
- Chapter 2, “Parallel Query Processing” describes parallel query processing in Adaptive Server Enterprise.
- Chapter 3, “Using showplan” describes the messages printed by the showplan utility.
- Chapter 4, “Displaying Query Optimization Strategies And Estimates” describes the set option and set plan query optimization strategies and estimates display commands for diagnostics.
- Chapter 5, “Query Processing Metrics” describes the query processing metrics feature, which identifies and compares empirical metric values during query execution.
- Chapter 6, “Abstract Plans” describes changes and additions to abstracts plans for Adaptive Server Enterprise.
- Chapter 7, “Using Statistics To Improve Performance” describes the use of statistics to help improve query execution performance, it also includes adescription for the automatic update statistics feature in Adaptive Server Enterprise.
- Chapter A, “Abstract Plan Specifications”describes details of various Abstract Plan Specifications.

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

❖ **Finding the latest information on component certifications**

1. Point your Web browser to Availability and Certification Reports at http://certification.sybase.com/.

2 Either select the product family and product under Search by Product; or select the platform and product under Search by Platform.

3 Select Search to display the availability and certification report for the selection.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1 Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.

2 Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.

3 Select a product.

4 Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the "Technical Support Contact" role to your MySybase profile.

5 Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

The following sections describe conventions used in this manual.

SQL is a free-form language. There are no rules about the number of words you can put on a line or where you must break a line. However, for readability, all examples and most syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented. Complex commands are formatted using modified Backus Naur Form (BNF) notation.

Table 1 shows the conventions for syntax statements that appear in this manual:

Table 1: Font and syntax conventions for this manual

Element	Example
Command names,procedure names, utility names, and other keywords display in sans serif font.	select sp_configure
Database names and datatypes are in sans serif font.	master database
Book names, file names, variables, and path names are in italics.	*System Administration Guide* *sql.ini* file *column_name* *$SYBASE/ASE* directory
Variables—or words that stand for values that you fill in—when they are part of a query or statement, are in italics in Courier font.	`select` *`column_name`* `from` *`table_name`* `where` *`search_conditions`*
Type parentheses as part of the command.	compute *row_aggregate* (*column_name*)
Double colon, equals sign indicates that the syntax is written in BNF notation. Do not type this symbol. Indicates “is defined as”.	`::=`
Curly braces mean that you must choose at least one of the enclosed options. Do not type the braces.	`{cash, check, credit}`
Brackets mean that to choose one or more of the enclosed options is optional. Do not type the brackets.	`[cash | check | credit]`
The comma means you may choose as many of the options shown as you want. Separate your choices with commas as part of the command.	`cash, check, credit`
The pipe or vertical bar(|) means you may select only one of the options shown.	`cash | check | credit`
An ellipsis (...) means that you can *repeat* the last unit as many times as you like.	`buy thing = price [cash | check | credit] [, thing = price [cash | check | credit]]...` You must buy at least one thing and give its price. You may choose a method of payment: one of the items enclosed in square brackets. You may also choose to buy additional things: as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment.

- Syntax statements (displaying the syntax and all options for a command) appear as follows:

  sp_dropdevice [*device_name*]

  For a command with more options:

  select *column_name*
  from *table_name*
  where *search_conditions*

  In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase. Italic font shows user-supplied words.

- Examples showing the use of Transact-SQL commands are printed like this:

  ```
  select * from publishers
  ```

- Examples of output from the computer appear as follows:

```
pub_id    pub_name                city          state
-------   ---------------------   -----------   -----
0736      New Age Books           Boston        MA
0877      Binnet & Hardley        Washington    DC
1389      Algodata Infosystems    Berkeley      CA

(3 rows affected)
```

In this manual, most of the examples are in lowercase. However, you can disregard case when typing Transact-SQL keywords. For example, SELECT, Select, and select are the same.

Adaptive Server's sensitivity to the case of database objects, such as table names, depends on the sort order installed on Adaptive Server. You can change case sensitivity for single-byte character sets by reconfiguring the Adaptive Server sort order. For more information, see the *System Administration Guide*.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

Adaptive Server 15.0 and the HTML documentation have been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

The online help for this product is also provided in HTML, which you can navigate using a screen reader.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Understanding Query Processing in Adaptive Server

This chapter provides an overview of the query processor in Adaptive Server Enterprise.

Topic	Page
Query optimizer	3
Optimization goals	16
Parallelism	18
Optimization issues	18
Lava query execution engine	21

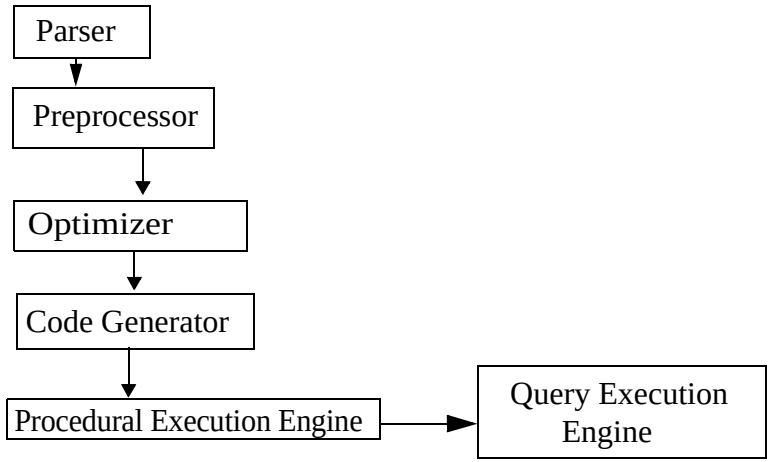
The query processor is designed to process queries you specify. The processor yields highly efficient query plans that execute using minimal resources and ensure that results are consistent and correct.

The query processor uses this information to process a query efficiently:

- the specified query
- statistics about the tables, indexes, and columns named in the query
- configurable variables

The query processor has to execute several steps to successfully process a query. Figure 1-1 shows the query processor modules:

Figure 1-1: : Query Processor modules



- The parser converts the text of the SQL statement to an internal representation called a query tree.
- The preprocessor transforms the query tree for some types of SQL statements, such as SQL statements with sub queries and views, to a more efficient query tree.
- The optimizer analyzes the possible combinations of operations (join ordering, access and join methods, parallelism) to execute the SQL statement, and selects an efficient one based on the cost estimates of the alternatives.
- The code generator converts the query plan generated by the optimizer into a format more suitable for the query execution engine.
- The procedural engine executes command statements such as create table, execute procedure, and declare cursor directly. For Data Manipulation Language (DML) statements, such as select, insert, delete, and update, the engine sets up the execution environment for all query plans and calls the query execution engine.
- The query execution engine executes the ordered steps specified in the query plan provided by the code generator.

Query optimizer

The query optimizer provides speed and efficiency for online transaction processing (OLTP) and operational decision-support systems (DSS) environments. You can choose an optimization strategy that best suits your query environment.

The query optimizer is self-tuning, and requires fewer interventions than earlier versions of Adaptive Server Enterprise. It relies infrequently on worktables for materialization between steps of operations; however, more worktables could be used in cases where it is determined that hash and merge operations are more effective.

Some of the key features in the query optimizer include support for:

- New optimization techniques and query execution operator supports that enhance query performance, such as:
  - On-the-fly grouping and ordering operator support using in-memory sorting and hashing for queries with group by and order by clauses
  - hash and merge join operator support for efficient join operations
  - index union and index intersection strategies for queries with predicates on different indexes

  The complete list of optimization techniques and operator support provided in Adaptive Server Enterprise is listed in Table 1-1. Many of these techniques map directly to the operators supported in the query execution. See “Lava query execution engine” on page 21.
- Improved index selection, especially for joins with or clauses, and joins with and search arguments (SARGs) with mismatched but compatible data types.
- Improved costing that employs join histograms to prevent inaccuracies that might otherwise arise due to data skews in joining columns.
- New cost-based pruning and timeout mechanisms in join ordering and plan strategies for large, multi-way joins, and for star and snowflake schema joins.
- New optimization techniques to support data and index partitioning (building blocks for parallelism) that are especially beneficial for very large data sets.
- Improved query optimization techniques for vertical and horizontal parallelism. See Chapter 2, “Parallel Query Processing,” for more details.

- Improved problem diagnosis and resolution through:
  - Searchable XML format trace outputs
  - Detailed diagnostic output from new set commands. See Chapter 5, “Query Processing Metrics,” for more details.

Table 1-1: Optimization techniques and operator support

Operator	Description
hash join	Determines whether the query optimizer may use the hash join algorithm. hash join may consume more runtime resources, but is valuable when the joining columns do not have useful indexes or when a relatively large number of rows satisfy the join condition, compared to the product of the number of rows in the joined tables.
hash union distinct	Determines whether the query optimizer may use the hash union distinct algorithm, which is inefficient if most rows are distinct.
merge join	Determines whether the query optimizer may use the merge join algorithm, which relies on ordered input. merge join is most valuable when input is ordered on the merge key, for example, from an index scan. merge join is less valuable if sort operators are required to order input.
merge union all	Determines whether the query optimizer may use the merge algorithm for union all. merge union all maintains the ordering of the result rows from the union input. merge union all is particularly valuable if the input is ordered and a parent operator (such as merge join) benefits from that ordering. Otherwise, merge union all may require sort operators that reduce efficiency.
merge union distinct	Determines whether the query optimizer may use the merge algorithm for union. merge union distinct is similar to merge union all, except that duplicate rows are not retained. merge union distinct requires ordered input and provides ordered output.
nested-loop-join	Determines whether the query optimizer may use the nested-loop-join algorithm. It is the most common type of join method and is most useful in simple OLTP queries that do not require ordering.
append union all	Determines whether the query optimizer may use the append algorithm for union all.
distinct hashing	Determines whether the query optimizer may use a hashing algorithm to eliminate duplicates, which is very efficient when there are few distinct values compared to the number of rows.

Operator	Description
distinct sorted	Determines whether the query optimizer may use a single-pass algorithm to eliminate duplicates. distinct sorted relies on an ordered input stream, and may increase the number of sort operators if its input is not ordered.
group-sorted	Determines whether the query optimizer may use an on-the-fly grouping algorithm. group-sorted relies on an input stream sorted on the grouping columns, and it preserves this ordering in its output.
distinct sorting	Determines whether the query optimizer may use the sorting algorithm to eliminate duplicates. distinct sorting is useful when the input is not ordered (for example, if there is no index) and the output ordering generated by the sorting algorithm could benefit; for example, in a merge join.
group hashing	Determines whether the query optimizer may use a group hashing algorithm to process aggregates.

Technique	Description
multi table store ind	Determines whether the query optimizer may use reformatting on the result of a multiple table join. Using multi table store ind may increase the use of worktables.
opportunistic distinct view	Determines whether the query optimizer may use a more flexible algorithm when enforcing distinctness.
index intersection	Determines whether the query optimizer may use the intersection of multiple index scans as part of the query plan in the search space.

Factors analyzed in optimizing queries

Query plans consist of retrieval tactics and an ordered set of execution steps, which retrieve the data needed by the query. In developing query plans, the query optimizer examines:

- The size of each table in the query, both in rows and data pages, and the number of OAM and allocation pages to be read.
- The indexes that exist on the tables and columns used in the query, the type of index, and the height, number of leaf pages, and cluster ratios for each index.

- The index coverage of the query; that is, whether the query can be satisfied by retrieving data from the index leaf pages without accessing the data pages. Adaptive Server can use indexes that cover queries, even if no where clauses are included in the query.
- The density and distribution of keys in the indexes.
- The size of the available data cache or caches, the size of I/O supported by the caches, and the cache strategy to be used.
- The cost of physical and logical reads; that is, reads of physical I/O pages from the disk, and of logical I/O reads from main memory.
- join clauses, with the best join order and join type, considering the costs and number of scans required for each join and the usefulness of indexes in limiting the I/O.
- Whether building a worktable (an internal, temporary table) with an index on the join columns is faster than repeated table scans if there are no useful indexes for the inner table in a join.
- Whether the query contains a max or min aggregate that can use an index to find the value without scanning the table.
- Whether data or index pages must be used repeatedly, to satisfy a query such as a join, or whether a fetch-and-discard strategy can be employed because the pages need to be scanned only once.

For each plan, the query optimizer determines the total cost by computing the costs of logical and physical I/Os, and CPU processing. If there are proxy tables, additional network related costs are evaluated as well. The query optimizer then selects the cheapest plan.

Stored procedures and triggers are optimized when the object is first executed, and the query plan is stored in the procedure cache. If other users execute the same procedure while an unused copy of the plan resides in cache, the compiled query plan is copied in cache, rather than being recompiled.

Transformations for query optimization

After a query is parsed and preprocessed, but before the query optimizer begins its plan analysis, the query is transformed to increase the number of clauses that can be optimized. The transformation changes made by the optimizer are transparent unless the output of such query tuning tools as showplan, dbcc(200), statistics io, or the set commands is examined. If you run queries that benefit from the addition of optimized search arguments, the added clauses are visible. In showplan output, it appears as "Keys are" messages for tables for which you specify no search argument or join.

Search arguments converted to equivalent arguments

The optimizer looks for query clauses to convert to the form used for search arguments. These are listed in Table 1-2.

Table 1-2: Search argument equivalents

Clause	Conversion
between	Converted to >= and <= clauses. For example, between 10 and 20 is converted to >= 10 and <= 20.
like	If the first character in the pattern is a constant, like clauses can be converted to greater than or less than queries. For example, like "sm%" becomes >= "sm" and < "sn". If the first character is a wildcard, a clause such as like "%x" cannot use an index for access, but histogram values can be used to estimate the number of matching rows.
in(values_list)	Converted to a list of or queries, that is, int_col in (1, 2, 3) becomes int_col = 1 or int_col = 2 or int_col = 3. The maximum number of elements in an in-list is 1025

Search argument transitive closure applied where applicable

The optimizer applies transitive closure to search arguments. For example, the following query joins titles and titleauthor on title_id and includes a search argument on titles.title_id:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
        and a.au_id = ta.au_id
        and t.title_id = "T81002"
```

This query is optimized as if it also included the search argument on titleauthor.title_id:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
        and a.au_id = ta.au_id
        and t.title_id = "T81002"
        and ta.title_id = "T81002"
```

With this additional clause, the query optimizer can use index statistics on titles.title_id to estimate the number of matching rows in the titleauthor table. The more accurate cost estimates improve index and join order selection.

equi-join predicate transitive closure applied where applicable

The optimizer applies transitive closure to join columns for a normal equi-join. The following query specifies the equi-join of t1.c11 and t2.c21, and the equi-join of t2.c21 and t3.c31:

```
select *
from t1, t2, t3
where t1.c11 = t2.c21
      and t2.c21 = t3.c31
      and t3.c31 = 1
```

Without join transitive closure, the only join orders considered are (t1, t2, t3), (t2, t1, t3), (t2, t3, t1),and (t3, t2, t1). By adding the join on t1.c11 = t3.31, the query processor expands the list of join orders with these possibilities: (t1, t3, t2) and (t3, t1, t2). Search argument transitive closure applies the condition specified by t3.c31 = 1 to the join columns of t1 and t2.

Similarly, equi-join transitive closure is also applied to equi-joins with or predicates as follows:

```
select *
from R,S
where R.a = S.a
and (R.a = 5 OR S.b = 6)
```

The query optimizer infers that the following query would be equivalent to:

```
select *
from R,S
where R.a = S.a
and (S.a = 5 or S.b = 6)
```

The or predicate could be evaluated on the scan of S and possibly be used for an or optimization, thereby using the indexes of S very effectively.

Another example of join transitive closure is its application to non-simple SARGs, so that a query such as:

```
select *
from R,S
where R.a = S.a and (R.a + S.b = 6)
```

is transformed and inferred as:

```
select *
from R,S
where R.a = S.a
and (S.a + S.b = 6)
```

The complex predicate could be evaluated on the scan of S, resulting in significant performance improvements due to early result set filtering.

Transitive closure is used only for normal equi-joins, as shown. join transitive closure is not performed for:

- Non-equi-joins; for example, t1.c1 > t2.c2
- Outer joins; for example t1.c11 *= t2.c2, or left join or right join
- Joins across sub query boundaries
- Joins used to check referential integrity or the with check option on views

Note In Adaptive Server Enterprise 15.0, the sp_configure option to turn on or off join transitive closure and sort merge join is discontinued. This means that whenever applicable, join transitive closure is always applied in Adaptive Server Enterprise 15.0.

Predicate transformation and factoring done to provide additional optimization paths

Predicate transformation and factoring increases the number of choices available to the query processor. It adds clauses that can be optimized to a query by extracting clauses from blocks of predicates linked with or into clauses linked by and. The additional optimized clauses mean that there are more access paths available for query execution. The original or predicates are retained to ensure query correctness.

During predicate transformation:

1 Simple predicates (joins, search arguments, and in lists) that are an exact match in each or clause are extracted. In the sample query, this clause matches exactly in each block, so it is extracted:

```
t.pub_id = p.pub_id
```

between clauses are converted to greater-than-or-equal and less-than-or-equal clauses before predicate transformation. The sample query uses between 15 in both query blocks (though the end ranges are different). The equivalent clause is extracted by step 1:

```
price >=15
```

2 Search arguments on the same table are extracted; all terms that reference the same table are treated as a single predicate during expansion. Both type and price are columns in the titles table, so the extracted clauses are:

```
(type = "travel" and price >=15 and price <= 30)
or
(type = "business" and price >= 15 and price <= 50)
```

3 in lists and or clauses are extracted. If there are multiple in lists for a table within a blocks, only the first is extracted. The extracted lists for the sample query are:

```
p.pub_id in (“P220”, “P583”, “P780”)
or
p.pub_id in (“P651", “P066", “P629”)
```

Since these steps can overlap and extract the same clause, duplicates are eliminated.

Each generated term is examined to determine whether it can be used as an optimized search argument or a join clause. Only those terms that are useful in query optimization are retained.

The additional clauses are added to the query clauses specified by the user.

For example, all clauses optimized in this query are enclosed in the or clauses:

```
       select p.pub_id, price
from publishers p, titles t
where (
     t.pub_id = p.pub_id
     and type = “travel"
     and price between 15 and 30
     and p.pub_id in (“P220", “P583", “P780")
     )
or  (
```

```
t.pub_id = p.pub_id
and type = "business"
and price between 15 and 50
and p.pub_id in ("P651", "P066", "P629")
)
```

Predicate transformation pulls clauses linked with and from blocks of clauses linked with or, such as those shown above. It extracts only clauses that occur in all parenthesized blocks. If the example above had a clause in one of the blocks linked with or that did not appear in the other clause, that clause would not be extracted.

Handling search arguments and useful indexes

It is important to distinguish between where and having clause predicates that can be used to optimize the query and those that are used later during query processing to filter the returned rows.

You can use search arguments to determine the access path to the data rows when a column in the where clause matches an index key. The index can be used to locate and retrieve the matching data rows. Once the row has been located in the data cache or has been read into the data cache from disk, any remaining clauses are applied.

For example, if the authors table has on an index on au_lname and another on city, either index can be used to locate the matching rows for this query:

```
select au_lname, city, state
from authors
where city = "Washington"
and au_lname = "Catmull"
```

The query optimizer uses statistics, including histograms, the number of rows in the table, the index heights, and the cluster ratios for the index and data pages to determine which index provides the cheapest access. The index that provides the cheapest access to the data pages is chosen and used to execute the query, and the other clause is applied to the data rows once they have been accessed.

Non-equality operators

The non-equality operators, < > and !=, are special cases. The query optimizer checks whether it should cover non-clustered indexes if the column is indexed, and uses a non-matching index scan if an index covers the query. However, if the index does not cover the query, the table is accessed through a Row ID lookup of the data pages during the index scan.

Examples of search argument optimization

Shown below are examples of clauses that can be fully optimized. If there are statistics on these columns, they can be used to help estimate the number of rows the query will return. If there are indexes on the columns, the indexes can be used to access the data.

```
au_lname = "Bennett"
price >= $12.00
advance > $10000 and advance < $20000
au_lname like "Ben%" and price > $12.00
```

These search arguments cannot be optimized unless a functional index is built on them:

```
advance * 2 = 5000  /*expression on column side
                      not permitted */
substring(au_lname,1,3) = "Ben" /* function on
                      column name */
```

These two clauses can be optimized if written in this form:

```
advance = 5000/2
au_lname like "Ben%"
```

Consider this query, with the only index on au_lname:

```
select au_lname, au_fname, phone
   from authors
   where au_lname = "Gerland"
     and city = "San Francisco"
```

The clause qualifies as a SARG (Search Argument):

```
au_lname = "Gerland"
```

- There is an index on au_lname
- There are no functions or other operations on the column name.
- The operator is a valid SARG operator.

This clause matches all the criteria above except the first; there is no index on the city column. In this case, the index on au_lname is used for the query. All data pages with a matching last name are brought into cache, and each matching row is examined to see if the city matches the search criteria.

Handling *joins*

The query optimizer deals with join predicates the same way it deals with search arguments, in that it uses statistics, number of rows in the table, index heights, and the cluster ratios for the index and data pages to determine which index and join method provides the cheapest access. In addition, the query optimizer also uses join density estimates derived from join histograms that give accurate estimates of qualifying joining rows and the rows to be scanned in the outer and inner tables. The query optimizer also must decide on the optimal join ordering that will yield the most efficient query plan. The next sections describe the key techniques used in processing joins.

Join density and join histograms

The query optimizer uses a cost model for joins that uses table-normalized histograms of the joining attributes. This technique gives an exact value for the skewed values (that is, frequency count) and uses the range cell densities from each histogram to estimate the cell counts of corresponding range cells.

The join density is dynamically computed from the “join histogram,” which considers the joining of histograms from both sides of the join operator. The first histogram join occurs typically between two base tables when both attributes have histograms. Every histogram join creates a new histogram on the corresponding attribute of the parent join's projection.

The outcome of the join histogram technique is accurate join selectivity estimates, even if data distributions of the joining columns are skewed, resulting in superior join orders and performance.

Joins with mixed data types

A basic requirement is the ability to build keys for index lookups whenever possible, without regard to mixed data types of any of the join predicates versus the index key. Consider the following query

```
create table T1 (c1 int, c2 int)
create table T2 (c1 int, c2 float)
```

```
create index i1 on T1(c2)
create index i1 on T2(c2)

select * from T1, T2 where T1.c2=T2.c2
```

Assume that T1.c2 is of type int and has an index on it, and that T2.c2 is of type float with an index.

As long as data types are implicitly convertible, index scans can be gainfully used to process the join. In other words, the query optimizer will use the column value from the outer table to position the index scan on the inner table, even when the lookup value from the outer table has a different data type than the respective index attribute of the inner table.

Joins with expressions and *or* predicates

See “Predicate transformation and factoring done to provide additional optimization paths” on page 10 for description of how the query optimizer handles joins with expressions and or predicates

join Ordering

One of the key tasks of the query optimizer is to generate a query plan for join queries so that the order of the relations in the joins processed during query execution is optimal. This involves elaborate plan search strategies that can consume significant time and memory. The query optimizer uses several effective techniques to obtain the optimal join ordering. The key techniques are:

- Use of a greedy strategy to obtain an initial good ordering that can be used as an upper boundary to prune out other, subsequent join orderings. The greedy strategy employs join row estimates and the nested loop join method to arrive at the initial ordering.
- An exhaustive ordering strategy follows the greedy strategy. In this strategy, a potentially better join ordering replaces the join ordering obtained in the greedy strategy. This ordering may employ any join method.
- Use of extensive cost-based and rule-based pruning techniques eliminates undesirable join orders from consideration. The key aspect of the pruning technique is that it always compares partial join orders (the prefix of a potential join ordering) against the best complete join ordering to decide whether to proceed with the given prefix. This significantly improves the time required determine an optimal join order.

- The query optimizer can recognize and process star or snowflake schema joins and process their join ordering in the most efficient way. A typical star schema join involves a large Fact table that has equi-join predicates that join it with several Dimension tables. The Dimension tables have no join predicates connecting each other; that is, there are no joins between the Dimension tables themselves, but there are join predicates between the Dimension tables and the Fact table. The query optimizer employs special join ordering techniques during which the large Fact table is pushed to the end of the join order and the Dimension tables are pulled up front, yielding highly efficient query plans. The query optimizer will not, however, use this technique if the star schema joins contain sub queries, outer joins or or predicates.

Optimization goals

Optimization goals are a convenient way of matching query demands with the best optimization techniques, thus ensuring optimal use of the optimizer's time and resources. The query optimizer allows you to configure two types of optimization goals, which you can specify at three tiers: server level, session level, and query level.

Set the optimization goal at the desired level. The server-level optimization goal is overridden at the session level, which is overridden at the query level.

These optimization goals allow you to choose an optimization strategy that best fits your query environment:

- allrows_mix – the default goal, and the most useful goal in a mixed-query environment. It balances the needs of OLTP and DSS query environments.
- allrows_dss – the most useful goal for operational DSS queries of medium to high complexity. Currently, this goal is provided on an experimental basis.

At the server level, use sp_configure. For example:

```
sp_configure "optimization goal", 0, "allrows_mix"
```

At the session level, use set plan optgoal. For example:

```
set plan optgoal allrows_dss
```

At the query level, use the select or other DML command. For example:

```
select * from A order by A.a plan
```

```
"(use optgoal allrows_dss)"
```

Exceptions

In general, you can set query-level optimization goals using select, update, and delete statements. However, you cannot set query-level optimization goals in pure insert statements, although you can set optimization goals in insert...select statements.

Limiting the time spent optimizing a query

Long-running and complex queries can be time-consuming and costly to optimize. The timeout mechanism helps to limit that time while supplying a satisfactory query plan. The query optimizer provides a mechanism by which the optimizer can limit the time taken by long-running and complex queries; timing out allows the query processor to stop optimizing when it is reasonable to do so.

The optimizer triggers timeout during optimization when both these circumstances are met:

- At least one complete plan has been retained as the best plan.
- The user configured timeout percentage limit has been exceeded.

You can limit the amount of time Adaptive Server spends optimizing a query at every level, using the optimization timeout limit parameter. Its value can be any value between 0 and 1000. The optimization timeout limit parameter represents the percentage of estimated query execution time that Adaptive Server must spend to optimize the query. For example, specifying a value of 10 tells Adaptive Server to spend 10% of the estimated query execution time in optimizing the query. Similarly, a value of 1000 tells Adaptive Server to spend 1000% of the estimated query execution time, or 10 times the estimated query execution time, in optimizing the query.

A large timeout value may be useful for optimization of stored procedures with complex queries. It is expected that the longer optimization time of the stored procedures will yield better plans; the longer optimization time can be amortized over several executions of the stored procedure.

A small timeout value may be used when a faster compilation time is wanted from complex ad-hoc queries that normally take a long time to compile. However, for most queries, the default timeout value of 10 should suffice.

Use sp_configure to set the optimization timeout limit configuration parameter at the server level. For example, to limit optimization time to 10% of total query processing time, enter:

```
sp_configure "optimization timeout limit", 10
```

Use set to set timeout at the session level:

```
set plan opttimeoutlimit <n>
```

where *n* is any integer between 0 and 1000.

Use select to limit optimization time at the query level:

```
select * from <table> plan "(use opttimeoutlimit <n>)"
```

where *n* is any integer between 0 and 1000.

Table 1-3: Optimization time out limit

Summary information	
Default value	10
Range of values	1 - 1000
Status	Dynamic
Display level	Comprehensive
Required role	System Administrator

Parallelism

Adaptive Server supports horizontal and vertical parallelism for query execution. Vertical parallelism is the ability to run multiple operators at the same time by employing different system resources such as CPUs, disks, and so on. Horizontal parallelism is the ability to run multiple instances of an operator on the specified portion of the data.

See Chapter 2, “Parallel Query Processing,” for a more detailed discussion of parallel query optimization in Adaptive Server.

Optimization issues

Although the query optimizer can optimize most queries efficiently, there are some optimization issues that should be noted:

- If statistics have not been updated recently, the actual data distribution may not match the values used to optimize queries
- The rows referenced by a specified transaction may not fit the pattern reflected by the index statistics
- An index may access a large portion of the table
- where clauses (SARGS) are written in a form that cannot be optimized
- No appropriate index exists for a critical query
- A stored procedure was compiled before significant changes to the underlying tables were performed
- No statistics exists for the SARG or joining columns

These situations highlight the need to follow some best practices that will allow the query optimizer to perform at its full potential. Some of the practices that the you may employ could include:

Creating search arguments

Follow these guidelines when you write search arguments for your queries:

- Avoid functions, arithmetic operations, and other expressions on the column side of search clauses. When possible, move functions and other operations to the expression side of the clause.
- Use all the search arguments you can to give the query processor as much as possible to work with.
- If a query has more than 400 predicates for a table, put the most potentially useful clauses near the beginning of the query, since only the first 102 SARGs on each table are used during optimization. (All of the search conditions are used to qualify the rows.)
- Queries using > (greater than) may perform better if you can rewrite them to use >= (greater than or equal to). For example, this query, with an index on int_col, uses the index to find the first value where int_col equals 3, and then scans forward to find the first value that is greater than 3. If there are many rows where int_col equals 3, the server has to scan many pages to find the first row where int_col is greater than 3:

  ```
  select * from table1 where int_col > 3
  ```

  It is more efficient to write the query this way:

  ```
  select * from table1 where int_col >= 4
  ```

  This optimization is more difficult with character strings and floating-point data.

- Check the showplan output to see which keys and indexes are used.
- If an index is not being used when you expect it to be, use output from the set commands in Table 1-1 on page 5 to see whether the query processor is considering the index.

Use of SQL derived tables

Queries expressed as a single SQL statement exploit the query processor better than queries expressed in two or more SQL statements. SQL-derived tables enable you to express, in a single step, what might otherwise require several SQL statements and temporary tables, especially where intermediate aggregate results must be stored. For example:

```
select dt_1.* from
   (select sum(total_sales)
        from titles_west group by total_sales)
               dt_1(sales_sum),
   (select sum(total_sales)
         from titles_east group by total_sales)
               dt_2(sales_sum)
where dt_1.sales_sum = dt_2.sales_sum
```

Here, aggregate results are obtained from the SQL derived tables dt_1 and dt_2, and a join is computed between the two SQL derived tables. Everything is accomplished in a single SQL statement.

For more information on SQL derived tables, see the *Transact-SQL User's Guide*.

Tuning according to object sizes

To understand query and system behavior, know the sizes of your tables and indexes. At several stages of tuning work, you need size data to:

- Understand statistics i/o reports for a specific query plan.
- Understand the query processor's choice of query plan. The Adaptive Server cost-based query processor estimates the physical and logical I/O required for each possible access method and selects the cheapest method.
- Determine object placement, based on the sizes of database objects and on the expected I/O patterns on the objects.

  To improve performance, distribute database objects across physical devices, so that reads and writes to disk are evenly distributed.

  Object placement is described in “Controlling Physical Data Placement,” in *Performance and Tuning: Basics.*

- Understand changes in performance. If objects grow, their performance characteristics can change. For example, consider a table that is heavily used and is usually 100 percent cached. If the table grows too large for its cache, queries that access the table can suffer poor performance. This is particularly true of joins that require multiple scans.
- Do capacity planning. Whether you are designing a new system or planning for the growth of an existing system, you must know the space requirements in order to plan for physical disks and memory needs.
- Understand output from Adaptive Server Monitor Server and from sp_sysmon reports on physical I/O.

See the *System Administration Guide* for more information on sizing.

Lava query execution engine

In Adaptive Server, all query plans are submitted to the Procedural Execution Engine for execution. The Procedural Execution Engine drives execution of the query plan by:

- Executing simple SQL statements such as set, while and goto directly.
- Calling out to the Utility modules to execute create table, create index and other utility commands.
- Setting up the context for and driving the execution of stored procedures and triggers.
- Setting up the execution context and calling the Query Execution Engine to execute query plans for select, insert, delete and update statements.
- Setting up the cursor execution context for cursor open, fetch and close statements and calling the Query Execution Engine to execute these statements.
- Doing transaction processing and post execution cleanup

The Procedural Execution Engine is largely unchanged in Adaptive Server 15.0. However, to support the demands of today's applications, a new generation of query execution techniques is required. To meet that demand, the query execution engine has been completely rewritten. With a new query execution engine and query optimizer in place, the Procedural Execution Engine in Adaptive Server 15.0 passes all query plans generated by the new query optimizer to the Lava Query Execution Engine.

The Lava Query Execution Engine executes Lava Query Plans. All query plans chosen by the optimizer are compiled into Lava Query Plans. However, SQL statements that are not optimized, such as set or create, are compiled into query plans like those in prior versions of Adaptive Server and are not executed by the Lava Query Execution Engine. Non-Lava Query Plans are either executed by the Procedural Execution Engine or by Utility modules called by the Procedural Engine. Adaptive Server version 15.0 has two distinct kinds of query plans and this is clearly seen in the showplan output (see Chapter 4, “Displaying Query Optimization Strategies And Estimates”).

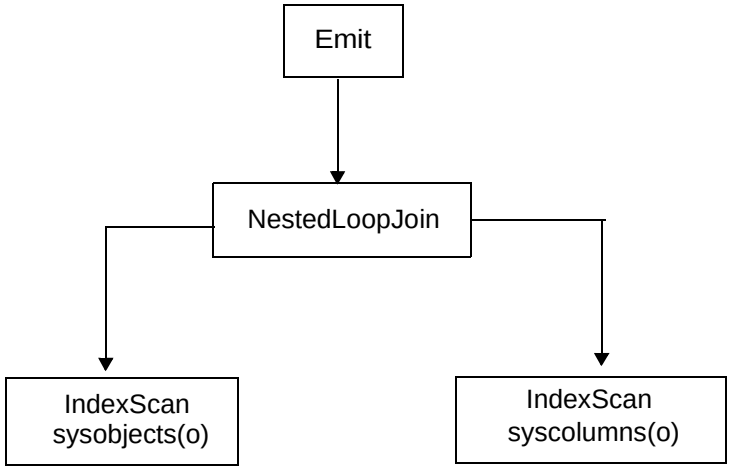
Lava query plans

A Lava Query Plan is built as an upside down tree of Lava Operators: The top Lava Operator can have one or more child operators, which in turn can have one or more child operators, and so on, thus building a bottom-up tree of operators. The exact shape of the tree and the operators in it are chosen by the optimizer.

An example of a Lava Query Plan for the following query is shown in Figure 1-2 below:

```
Select o.id from sysobjects o, syscolumns c
where o.id = c.id and o.id < 2
```

Figure 1-2: Lava Query Plan



The Lava Query Plan for this query consists of four Lava Operators. The top operator is an Emit (also called Root) operator that dispatches the results of query execution either by sending the rows to the client or by assigning values to local variables.

The only child operator of the Emit is a NestedLoopJoin (NLJoin)that uses the nested loop join algorithm to join the rows coming from its two child operators, (1) the Scan of sysobjects and (2) the scan of syscolumns.

Since the optimizer optimizes all select, insert, delete and update statements, these are always compiled into Lava Query Plans and executed by the Lava Query Engine.

Some SQL statements are compiled into hybrid query plans. Such plans have multiple steps, some of which are executed by the Utility modules and a final step that is a Lava Query Plan. An example is the select into statement; select into is compiled into a two-step query plan. The first step is a create table step to create the target table of the statement. The second step is a Lava Query Plan to insert the rows into the target table. To execute this query plan, the Procedural Execution Engine calls the create table utility to execute the first step to create the table. Then the Procedural Engine calls the Lava Query Execution Engine to execute the Lava Query Plan to select and insert the rows into the target table. The two other SQL statements that generate hybrid query plans are alter table (but only when data copying is required) and reorg rebuild.

A Lava Query Plan is also generated and executed to support BCP. The support for BCP in Adaptive Server has always been in the BCP Utility. Now, in 15.0, the BCP Utility generates a Lava Query Plan and calls the Lava Query Execution Engine to execute the plan.

More examples of Lava Query Plans can be found in Chapter 3, "Using showplan."

Lava operators

The Lava Query Plans are built up of Lava Operators. Each Lava Operator is a self-contained software object that implements one of the basic physical operations that the optimizer uses to build query plans. Each Lava Operator has five methods that can be called by its parent operator. These five methods correspond to the five phases of query execution and are called Acquire, Open, Next, Close, and Release. Because the Lava operators all provide the same methods (that is, the same API), they can be interchanged like building blocks in a Lava Query Plan. The NLJoin operator in Figure 1 could be replaced by a MergeJoin operator or a HashJoin operator without impacting any of the other three operators in the query plan.

The Lava Operators that can be chosen by the optimizer to build Lava Query Plans are listed in Table 1-4:

Table 1-4: Lava operators

Operator	Description
BulkOp	Executes the part of BCP processing that is done in the Lava Query Engine. Only found in query plans that are created by the BCP utility, not those created by the optimizer.
CacheScanOp	Reads rows from an in-memory table.
DelTextOp	Deletes text page chains as part of the alter table drop column processing.
DeleteOp	Deletes rows from a local table. Deletes rows from a proxy table when the entire SQL statement cannot be shipped to the remote server. See also RemoteScanOp.
EmitOp (RootOp)	Routes query execution result rows. Can send results to the client or assign result values to local variables or fetch into variables. An EmitOp is always the top operator in a Lava Query Plan.
EmitExchangeOp	Routes result rows from a sub-plan that is executed in parallel to the ExchangeOp in the parent plan fragment. EmitExchangeOp always appears directly under an ExchangeOp. See Chapter 2, "Parallel Query Processing."
GroupSortedOp (Aggregation)	Performs vector aggregation (group by) when the input rows are already sorted on the group-by columns. See also HashVectorAggOp.
GroupSorted (Distinct)	Eliminates duplicate rows. Requires the input rows to be sorted on all columns. See also HashDistinctOp and SortOp (Distinct).
HashVectorAggOp	Performs vector aggregation (group by). Uses a Hash algorithm to group the input rows, so no requirements on ordering of the input rows. See also GroupSortedOp (Aggregation).
HashDistinctOp	Eliminates duplicate rows using a hashing algorithm to find duplicate rows. See also GroupSortedOp (Distinct) and SortOp (Distinct).
HashJoinOp	Performs a join of two input row streams using the HashJoin algorithm.
HashUnionOp	Performs a union operation of two or more input row streams using a hashing algorithm to find and eliminate duplicate rows. See also MergeUnionOp and UnionAllOp.
InsScrollOp	Implements extra processing needed to support insensitive scrollable cursors. See also SemiInsScrollOp.

Operator	Description
InsertOp	Inserts rows to a local table. Inserts rows to a proxy table when the entire SQL statement cannot be shipped to the remote server. See also RemoteScanOp.
MergeJoinOp	Performs a join of two streams of rows that are sorted on the joining columns using the merge join algorithm.
MergeUnionOp	Performs a union or union all operation on two or more sorted input streams. Guarantees that the output stream retains the ordering of the input streams. See also HashUnionOp and UnionAllOp.
NestedLoopJoinOp	Performs a join of two input streams using the NestedLoopJoin algorithm.
NaryNestedLoopJoinOp	Performs a join of three or more input streams using an enhanced NestedLoopJoin algorithm. This operator replaces a left-deep tree of NestedLoopJoin operators and can lead to significant performance improvements when rows of some of the input streams can be skipped.
OrScanOp	Inserts the in or or values into an in-memory table, sorts the values and removes the duplicates.Then returns the values, one at a time. Only used for SQL statements with in clauses or multiple or clauses on the same column.
PtnScanOp	Reads rows from a local table (partitioned or not) using either a table scan or an index scan to access the rows.
RIDJoinOp	Receives one or more Row Identifiers (RIDs) from its left child operator and calls on its right child operator (PtnScanOp) to find the corresponding rows. Used only on SQL statements with or clauses on different columns of the same table.
RIFilterOp (Direct)	Drives the execution of a sub-plan to enforce referential integrity constraints that can be checked on a row-by-row basis. Appears only in insert, delete, or update queries on tables with referential integrity constraints.
RIFilterOp (Deferred)	Drives the execution of a sub-plan to enforce referential integrity constraints that can only be checked after all rows that will be affected by the query have been processed.

Operator	Description
RemoteScanOp	Accesses proxy tables. The RemoteScanOp can: • Read rows from a single proxy table for further processing in a Lava query plan on the local host. • Pass complete SQL statements to a remote host for execution: insert, delete, update, and select statements. In this case, the Lava query plan will consist of an EmitOp with a RemoteScanOp as its only child operator. • Pass an arbitrarily complex query plan fragment to a remote host for execution and read in the result rows (function shipping).
RestrictOp	Evaluates expressions.
SQFilterOp	Drives the execution of a sub plan to execute one or more subqueries.
ScalarAggOp	Performs scalar aggregation, such as aggregates without group by.
SemiInsScrollOp	Performs extra processing to support semi-insensitive scrollable cursors. See also InsScrollOp.
SequencerOp	Enforces sequential execution of different sub-plans in the query plan.
SortOp	Sorts its input rows based upon specified keys.
SortOp (Distinct)	Sorts its input and removes duplicate rows. See also HashDisitnctOp and GroupSortedOp (Distinct).
StoreOp	Creates and coordinates the filling of a worktable, and creates a clustered index on the worktable if required. This can only have an InsertOp as a child; the InsertOp populates the worktable.
UnionAllOp	Performs a union all operation on two or more input streams. See also HashUnionOp and MergeUnionOp.
UpdateOp	Changes the value of columns in rows of a local table or of a proxy table when the entire update statement cannot be sent to the remote server. See also RemoteScanOp.
ExchangeOp	Enables and coordinates parallel execution of Lava Query Plans. The ExchangeOp can be inserted between almost any two Lava Operators in a query plan to divide the plan into sub-plans that can be executed in parallel. See Chapter 2, “Parallel Query Processing.”.

Lava query execution

Execution of a Lava Query Plan involves five phases:

1 Acquire – acquires resources needed for execution, such as memory buffers and creating worktables.

2 Open – prepares to return result rows.

3 Next – generates the next result row.

4 Close – cleans up; for example, notifies the access layer that scanning is complete or truncates worktables

5 Release – releases resources acquired during Acquire, such as memory buffers, drops worktables.

Each Lava Operator has a method with the same name as the phase, which is invoked for each of these phases.

The query plan in Figure 1-2 can be used to demonstrate query plan execution:

- Acquire phase

  The Acquire method of the Emit Operator is invoked. The Emit Operator calls Acquire of its child, the NLJoin Operator, which in turn calls Acquire on its left child operator (the Index Scan of *sysobjects*) and then on its right child operator (the Index Scan of *syscolumns*).

- Open phase

  The Open method of the Emit Operator is invoked. The Emit Operator calls Open on the NLJoin Operator, which calls Open only on its left child operator.

- Next phase

  The Next method of the Emit Operator is invoked. Emit calls Next on the NLJoin Operator, which calls Next on its left child, the Index Scan of *sysobjects*. The Index Scan Operator reads the first row from *sysobjects* and returns it to the NLJoin Operator. The NLJoin Operator then calls the Open method of its right child operator, the Index Scan of *syscolumns*. Then the NLJoin Operator calls the Next method of the Index Scan of *syscolumns* to get a row that matches the joining key of the row from *sysobjects*. When a matching row has been found, it is returned to the Emit Operator, which sends it back to the client. Repeated invocations of the Next method of the Emit Operator generate more result rows.

- Close phase

  After all rows have been returned, the Close method of the Emit Operator is invoked, which in turn calls Close of the NLJoin Operator, which in turn calls Close on both of its child operators.

- Release phase

  The Release method of the Emit Operator is invoked and the calls to the Release method of the other operators is propagated down the query plan

After successfully completing the Release phase of execution, the Lava Query Engine returns control to the Procedural Execution Engine for final statement processing.

CHAPTER 2

Parallel Query Processing

This chapter provides an in-depth description of parallel query processing.

Topic	Page
Vertical, horizontal, and pipelined parallelism	31
Queries that benefit from parallel processing	32
Enabling parallelism	33
Controlling parallelism at the session level	36
Controlling parallelism for a query	37
When parallel query results differ	38
Understanding Parallel Query Plans	40
Adaptive Server's parallel query execution model	42

Vertical, horizontal, and pipelined parallelism

Adaptive Server supports horizontal and vertical parallelism for query execution. Vertical parallelism is the ability to run multiple operators at the same time by employing different system resources such as CPUs, disks, and so on. Horizontal parallelism is the ability to run multiple instances of an operator on the specified portion of the data.

The way you partition your data greatly affects how well horizontal parallelism works. The logical partitioning of data is useful in operational decision-support systems (DSS) queries where large volumes of data are being processed. See Partitioning in the *System Administration Guide* for a more detailed discussion of partitioning on Adaptive Server. Understanding different types of partitioning is a prerequisite to understanding this chapter.

Adaptive Server 15.0 also supports pipelined parallelism. Pipelining is a form of vertical parallelism in which intermediate results are piped to higher operators in a query tree. The output of one operator is used as input for another operator. The operator used as input can run at the same time as the operator feeding the data, which is an essential element in pipelined parallelism. Only use parallelism when multiple resources like disks and CPUs are available; using parallelism can be detrimental if your system is not configured for resources that can work in tandem. In addition, data must be spread across disk resources in a way that closely ties the logical partitioning of the data with the physical partitioning on parallel devices. The biggest challenge for a parallel system is to control the correct granularity of parallelism. If parallelism is too finely grained, the communication and synchronization overhead can offset any benefit that can be obtained through parallel operations. Making parallelism too coarse does not permit proper scaling.

Queries that benefit from parallel processing

When Adaptive Server is configured for parallel query processing, the query optimizer evaluates each query to determine whether it is eligible for parallel execution. If it is eligible, and if the optimizer determines that a parallel query plan can deliver results faster than a serial plan, the query is divided into plan fragments that are processed simultaneously. The results are combined and delivered to the client in a shorter period of time than it would take to process the query serially as a single fragment.

Parallel query processing can improve the performance of these types of queries:

- select statements that scan large numbers of pages but return relatively few rows, such as table scans or clustered index scans with grouped or ungrouped aggregates.
- Table scans or clustered index scans that scan a large number of pages, but have where clauses that return only a small percentage of rows.
- select statements that include union, order by, or distinct, since these query operations can make use of parallel sorting or parallel hashing.
- select statements where a reformatting strategy is chosen by the optimizer, since these can populate worktables in parallel and can make use of parallel sorting.
- join queries also benefit from parallel access.

Commands that return large, unsorted result sets are unlikely to benefit from parallel processing due to network constraints. In most cases, results can be returned from the database faster than they can be merged and returned to the client over the network.

Parallel DMLs like insert, delete, and update are not supported and so do not benefit from parallelism.

Enabling parallelism

To configure Adaptive Server for parallelism, you must enable the number of worker processes and max parallel degree parameters.

To get optimal performance, you must be aware of other configuration parameters that affect the quality of plans generated by Adaptive Server.

Setting the number of worker processes

Before you enable parallelism, you must first configure the number of worker processes (also referred to as threads) available for Adaptive Server by setting the configuration parameter number of worker processes. Make sure you configure a sufficient number of worker processes. Sybase recommends that you set the value for number of worker processes to one and a half times the total number required at peak load. You can calculate an approximate number using the max parallel degree configuration parameter, which indicates the total number of worker processes that can be used for any query. Depending on the number of connections to the Adaptive Server and the approximate number of queries that are run simultaneously, you can roughly estimate the value for the number of worker processes that may be needed at any time using this rule:

Value for number of worker processes = [max parallel degree] times [the number of concurrent connections wanting to run queries in parallel] times [1.5]

If the query processor has insufficient worker processes, it tries to adjust the query plan during run time. If a minimal number of worker processes are required but unavailable, the query aborts with this error message:

```
Insufficient number of worker processes to execute the
parallel query. Increase the value of the configuration
parameter 'number of worker processes
```

To set the number of worker process to 40:

```
sp_configure "number of worker processes", 40
```

Any run time adjustment for the number of threads may have a negative effect on the performance of the query. Adaptive Server tries to optimize the usage of threads in all cases, but when trying to adjust for threads it may have already committed to a plan that needs increased resources and hence does not guarantee a linear scaledown when made to run with fewer threads.

Setting max parallel degree

Configure the maximum amount of parallelism for a query using the max parallel degree configuration parameter, which determines the maximum number of threads Adaptive Server uses when processing a given query. To set the value of max parallel degree to 10:

```
sp_configure "max parallel degree", 10
```

Unlike earlier versions of Adaptive Server, this is not entirely enforced by the query optimizer. A complete enforcement process is very expensive in terms of optimization time. Adaptive Server comes very close to the desired setting of max parallel degree and only exceeds it for semantic reasons

Setting max resource granularity

The value of max resource granularity indicates the maximum percentage of the system resources a query can use. At this time, only procedure cache is considered in this option. It is set to 10% by default. However, this parameter is not enforced at execution time; is only a guide for the query optimizer. The query engine can avoid memory intensive strategies, such as hash-based algorithms, when max resource granularity is set to a low value.

To set max resource granularity to 5%:

```
sp_configure "max resource granularity", 5
```

Setting max repartition degree

Adaptive Server needs to dynamically repartition intermediate data to match the partitioning scheme of another operand or to do an efficient partition elimination. The configuration parameter max repartition degree controls the amount of dynamic repartitioning Adaptive Server can do. If the value of max repartition degree is too high, the number of intermediate partitions becomes too large and the system becomes flooded with worker processes that compete for resources, which eventually degrades performance. The value for max repartition degree enforces the maximum number of partitions created for any intermediate data. Repartitioning is a CPU intensive operation. Hence, the value of max repartition degree should not exceed the total number of Adaptive Server engines.

If all of the tables and indices are unpartitioned, Adaptive Server uses the value for max repartition degree to provide the number of partitions to create as a result of repartitioning the data. When the value is set to 1, which is the default case, the value of max repartition degree is set to the number of online engines.

max repartition degree is also used when force option is used to do parallel scan on a table or an index.

```
select * from customers (parallel)
```

If the customers table is unpartitioned and the force option is used, Adaptive Serve tries to find the inherent partitioning degree of that table or index, which in this case is 1. So, it will use a degree that is a function of two things: the number of engines configured for the server; or, whatever degree is best based on the number of pages in the table or index, but not exceeding the value of max repartition degree.

To set max repartition degree to 5:

```
sp_configure "max repartition degree", 5
```

Setting max scan parallel degree

The configuration parameter max scan parallel degree is used only for backward compatibility, when the data in a partitioned table or index is highly skewed. If the value of this parameter is greater than 1, Adaptive Server uses this value to do a hash based scan. The value of max scan parallel degree cannot exceed the value of max parallel degree.

Controlling parallelism at the session level

Set options let you restrict the degree of parallelism on a session basis or in stored procedures or triggers. These options are useful for tuning experiments with parallel queries and can also be used to restrict non-critical queries to run in serial, so that worker processes remain available for other tasks. The set options are summarized in Table 2-1.

Table 2-1: Session level parallelism control

Parameter	Function
parallel_degree	Sets the maximum number of worker processes for a query in a session, stored procedure, or trigger. Overrides the max parallel degree configuration parameter, but must be less than or equal to the value of max parallel degree.
scan_parallel_degree	Sets the maximum number of worker processes for a hash-based scan during a specific session, stored procedure, or trigger. Overrides the max scan parallel degree configuration parameter but must be less than or equal to the value of max scan parallel degree.
resource_granularity	Overrides the global value max resource granularity and sets it to a session specific value, which influences whether Adaptive Server uses memory-intensive operation or not.
repartition_degree	Sets the value of max repartition degree for a session. This is the maximum degree to which any intermediate data stream will be re-partitioned for semantic purposes.

If you specify a value that is too large for any of the set options, the value of the corresponding configuration parameter is used, and a message reports the value in effect. While set parallel_degree or set scan_parallel_degree or set repartition_degree or set resource_granularity is in effect during a session, the plans for any stored procedures that you execute are not placed in the procedure cache. Procedures executed with these options in effect may produce less than optimal plans.

set command examples

This example restricts all queries started in the current session to 5 worker processes:

```
set parallel_degree 5
```

While this command is in effect, any query on a table with more than 5 partitions cannot use a partition-based scan.

To remove the session limit, use:

```
set parallel_degree 0
```

or

```
set scan_parallel_degree 0
```

To run subsequent queries in serial mode, use:

```
set parallel_degree 1
```

or

```
set scan_parallel_degree 1
```

To set resource granularity to 25% of the total resources available in the system, use:

```
set resource_granularity 25
```

The same is true for repartition degree as well; you can set it to a value of 5. It cannot, however, exceed the value of max parallel degree.

```
set repartition_degree 5
```

Controlling parallelism for a query

The parallel extension to the from clause of a select command allows users to suggest the number of worker processes used in a select statement. The degree of parallelism that you specify cannot be more than the value set with sp_configure or the session limit controlled by a set command. If you specify a higher value, the specification is ignored, and the optimizer uses the set or sp_configure limit.

The syntax for the select statement is:

select ...

from *tablename* [([index *index_name*]
[parallel [*degree_of_parallelism* | 1]]
[prefetch *size*] [lru|mru])] ,
tablename [([index *index_name*]
[parallel [*degree_of_parallelism* | 1]
[prefetch *size*] [lru|mru])] ...

Query level parallel clause examples

To specify the degree of parallelism for a single query, include parallel after the table name. This example executes in serial:

```
select * from huge_table (parallel 1)
```

This example specifies the index to use in the query, and sets the degree of parallelism to 2:

```
select * from huge_table (index ncix parallel 2)
```

When parallel query results differ

When a query does not include scalar aggregates or does not require a final sorting step, a parallel query might return results in a different order from the same query run in serial, and subsequent executions of the same query in parallel might return results in different order. The relative speed of the different worker processes leads to differences in result set ordering. Each parallel scan behaves differently, due to pages already in cache, lock contention, and so forth. Parallel queries always return the same set of results, just not in the same order. If you need a dependable ordering of results, use order by or run the query in serial mode.

In addition, due to the pacing effects of multiple worker processes reading data pages, two types of queries accessing the same data may return different results when an aggregate or a final sort is not done. They are:

- Queries that use set rowcount
- Queries that select a column into a local variable without sufficiently restrictive query clauses

Queries that use *set rowcount*

The set rowcount option stops processing after a certain number of rows are returned to the client. With serial processing, the results are consistent in repeated executions as long as the plans are the same. In serial mode, given the same plan, the same rows are returned in the same order for a given rowcount value, because a single process reads the data pages in the same order every time. With parallel queries, the order of the results and the set of rows returned can differ, because worker processes may access pages sooner or later than other processes. To get consistent results, you must either use a clause that performs a final sort step or run the query in serial.

Queries that set local variables

This query sets the value of a local variable in a select statement:

```
select @tid = title_id from titles
where type = "business"
```

The where clause matches multiple rows in the titles table, so the local variable is always set to the value from the last matching row returned by the query. The value is always the same in serial processing, but for parallel query processing, the results depend on which worker process finishes last. To achieve a consistent result, use a clause that performs a final sort step, execute the query in serial mode, or add clauses so that the query arguments select only single rows.

Understanding Parallel Query Plans

The key to understanding parallel query processing in Adaptive Server 15.0 is to know what the basic building blocks in a parallel query plans are see Chapter 3, “Using showplan.” A compiled query plan consists of a tree of execution operators that closely resemble the relational semantics of the query. Each of the query operators implement a relational operation using a specific algorithm. For example, a query operator called nested loop join will implement the relational join operation. In Adaptive Server15.0, the primary operator for parallelism is the xchg operator (pronounced "exchange"). It is a control operator and does not implement any relational operation. The purpose of an xchg operator is to create new worker processes that can handle a fragment of the data. During optimization, Adaptive Server strategically places the xchg operator to create operator tree fragments that can be run in parallel. All operators found below the exchange operator (down to the next exchange operator) are executed by worker threads that clone the fragment of the operator tree to produce data in parallel. The exchange operator can then redistribute this data to the parent operator above it in the query plan. The exchange operator handles the pipelining and rerouting of data.

In the following sections, the word *degree* is used in different context. When degree N of a table or index is referred to, it references the number of partitions that the table or index has. When the degree of an operation or a configuration parameter is referred to, it references the number of partitions generated in the intermediate data stream.

The following example shows how operators in the query processor work in serial with the following query run in the pubs2 database. The table titles is hash partitioned three ways on the column pub_id.

```
select * from titles
QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

    |SCAN Operator
    |  FROM TABLE
    |  titles
    |  Table Scan.
    |  Forward Scan.
```

```
|  Positioning at start of table.
|  Using I/O Size 2 Kbytes for data pages.
|  With LRU Buffer Replacement Strategy for data
   pages.
```

As can be seen from this example, the table titles is being scanned by the Scan operator, the details of which can be seen in the output of "showplan". The Emit operator reads the data from the Scan operator and sends it out to the client application. A given query can create an arbitrarily complex tree of such operators.

Now, with parallelism turned on, Adaptive Server can perform a simple scan in parallel using the xchg operator above the scan operator. xchg produces three worker processes (based on the three partitions), each of which scans the three disjointed parts of the table and sends its output to the consumer process. The Emit operator at the top of the tree does not know that the scans are done in parallel.

Example A:

```
   select * from titles
Executed in parallel by coordinating process and 3 worker processes.

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 3 Producer and 1 Consumer processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |RESTRICT Operator
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  titles
    |   |   |   |  Table Scan.
    |   |   |   |  Forward Scan.
    |   |   |   |  Positioning at start of table.
    |   |   |   |  Executed in parallel with a 3-way     partition scan.
    |   |   |   |  Using I/O Size 2 Kbytes for data pages.
    |   |   |   |  With LRU Buffer Replacement Strategy for data pages.
```

Note the presence of an operator called Exchange:Emit. This is an operator that is placed under an Exchange operator to funnel data. The exchange operator is described in detail in “exchange operator” on page 42.

Adaptive Server's parallel query execution model

One of the key components of the parallel query execution model is the exchange operator. You can see it in the showplan output of a query.

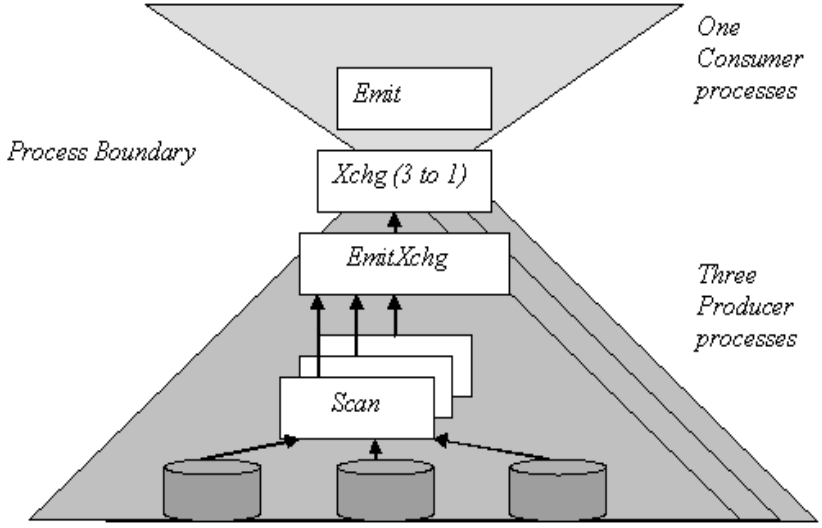
exchange operator

The exchange operator marks the boundary between a producer and a consumer operator (the operators below the exchange operator produce data and those above it consume data). In an earlier example (Example A) that showed parallel scan of the titles table (select * from titles), the exchange:emit and the scan operator produce data. This is shown briefly.

```
select * from titles

The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 3 Producer and 1 Consumer
     processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |RESTRICT Operator
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  titles
    |   |   |   |  Table Scan.
```

In this example, one consumer process reads data from a pipe (which is used as a medium to transfer data across process boundaries) and hands it off to the emit operator, which in turn routes the result to the client. The exchange operator also spawns worker processes, which are called producer threads. The exchange:emit operator is responsible for writing the data into a pipe managed by the exchange operator.

Figure 2-1: Binding of thread to plan fragments in query plan



The figure also shows the process boundary between a producer and a consumer task. There are indeed two plan fragments in this query plan. The plan fragment with the scan and the emitxchg operators are being cloned three ways and then a three-to-one xchg operator writes it into a pipe. The Emit operator and the xchg operator are run by a single process, which means there is a single clone of that plan fragment.

Pipe Management

The four types of pipes managed by the exchange operator are distinguished by how they split and merge data streams. You can determine which type of pipe is being managed by the exchange operator by looking at its description in the showplan output, where the number of producers and consumers are shown. The four pipe types are described below.

Many-to-one

In this case, the exchange operator spawns multiple producer threads and has one consumer task that reads the data from a pipe, to which multiple producer threads write. The exchange operator in the previous example implements a many-to-one exchange. A many-to-one exchange operator can be order preserving and this technique is employed particularly when doing a parallel sort for an orderby clause and the resultant data stream merged to generate the final ordering. The showplan output will show more than one producer process and one consumer process.

```
|EXCHANGE Operator
        |Executed in parallel by 3 Producer and 1
         Consumer processes
```

One-to-many

In this case, there is one producer and multiple consumer threads. The producer thread writes data to multiple pipes according to a partitioning scheme devised at query optimization and then routes data to each of these pipes. Each of the consumer threads read data from one of the assigned pipes. This kind of data split can preserve the ordering of the data. The showplan output will say one producer process and more than one consumer processes.

Many-to-many

"Many-to-many" means that there are multiple producers and multiple consumers. Each producer writes to multiple pipes, and each pipe has multiple consumers. Each stream is written to a pipe. Each of the consumer threads read data from one of the assigned pipes.

```
|EXCHANGE Operator
   |Executed in parallel by 3 Producer and 4
    Consumer processes
```

Replicated exchange operators

In this case, the producer thread writes all of its data to each of the pipes that the exchange operator configures. The producer thread makes a number of copies of the source data (the number is specified by the query optimizer) equal to the number of pipes in the xchg operator. Each of the consumer threads read data from one of the assigned pipes.

Worker process model

A parallel query plan is composed of different operators, at least one of which is an xchg operator. At run time, a parallel query plan is bound to a set of server processes that will, together, execute the query plan in a parallel fashion.

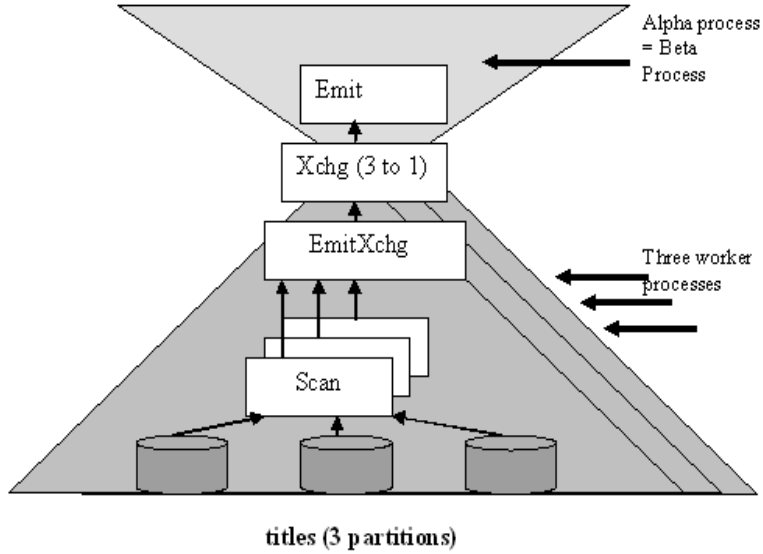
The server process associated with the user connection is called the *alpha process* because it is the source process from which parallel execution is initiated. In particular, each worker process involved in the execution of the parallel query plan is spawned by the alpha process.

In addition to spawning worker processes, the alpha process initializes all the worker processes involved in the execution of the plan, and creates and destroys the pipes necessary for worker processes to exchange data. The alpha process is, in effect, the global coordinator for the execution of a parallel query plan.

At run time, Adaptive Server 15.0 associates each xchg operator in the plan with a set of worker processes. The worker processes will execute the query plan fragment located immediately below the xchg operator.

For the query in Example A, represented in "exchange operator" on page 42, the xchg operator is associated with 3 worker processes. Each of the three worker processes will execute the plan fragment made of the EmitXchg operator and of the Scan operator.

Figure 2-2: Query execution plan with one xchg operator

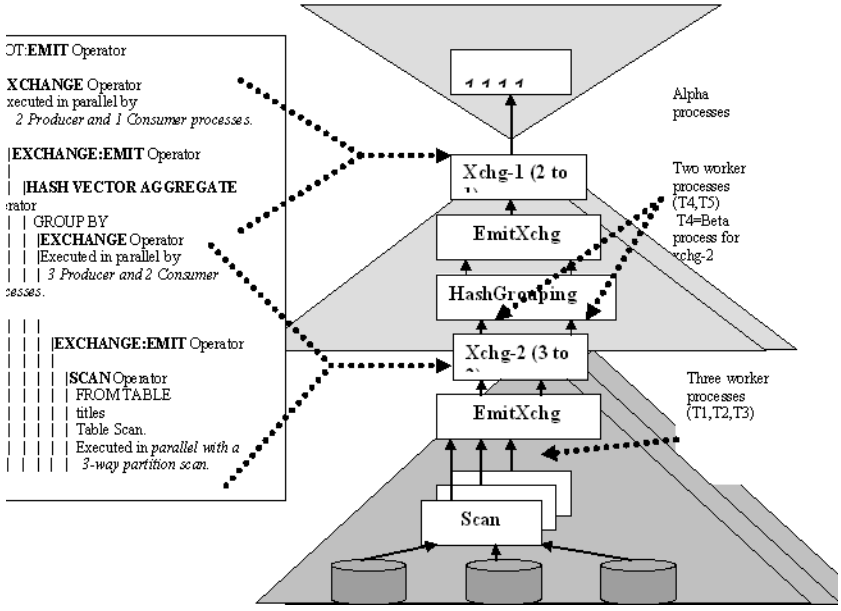


Each xchg operator is also associated with a server process named the *beta process*, which can be either the alpha process or a worker process. The beta process associated with a given xchg operator is the local coordinator for the execution of the plan fragment below the xchg operator. In the example above, the beta process is the same process as the alpha process, because the plan to be executed has only one level of xchg operators.

Next, we'll use this query to illustrate what happens when the query plan contains multiple xchg operators.

```
select count(*),pub_id, pub_date
from titles
group by pub_id, pub_date
```

Figure 2-3: Query execution plan with two xchg operators



There are two levels of xchg operators marked as Xchg-1 and Xchg-2 in Figure 2-3. Worker process T4 is the beta process associated with xchg operator Xchg-2.

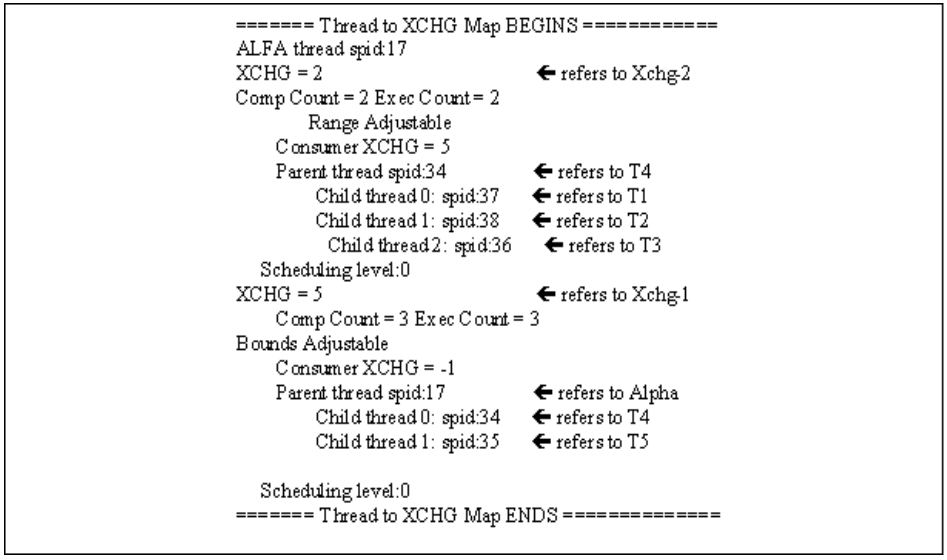
The function of the beta process is to locally orchestrate the execution of the plan fragment below the xchg; it dispatches query plan information that is needed by the worker processes and synchronizes the execution of the plan fragment.

A process involved in the execution of a parallel query plan that is neither the alpha process nor a beta process is called a *gamma process*.

A given parallel query plan is bound at run time to a unique alpha process, to one or more beta processes, and to at least one gamma process. It follows that any ASE 15.0 parallel plan will need at least two different processes (alpha and gamma) to be executed in parallel.

To find out the mapping between xchg operators and worker processes, as well as to figure out which process is the alpha process, and which processes are the beta processes, use dbcc traceon(516).

Figure 2-4: Mapping between operators and processes



Using parallelism in SQL operations

You can partition tables or indexes in any way that best reflects the needs of your application. Sybase recommends that you put partitions on segments that use different physical disks so that enough I/O parallelism is present. For example, you can have a well-defined partition based on hashing of certain columns of a table or certain ranges or a list of values ascribed to a partition. Hash, range, and list partitions belong to the category of "semantic-based" partitioning, so called because, given a row, you can determine which partition the row belongs to.

On the other hand, round-robin partitioning has no semantics associated with its partitioning. A row can occur in any of its partitions. The choice of columns to partition and the type of partitioning used can have a significant impact on the performance of the application. Partitions can be thought of as a low cardinality index; hence the columns on which partitioning needs to be defined are based on the queries in the application.

The query processing engine and its operators take advantage of Adaptive Server's partitioning strategy. Partitioning defined on table and indices is called static partitioning. In addition, Adaptive Server *dynamically* repartitions data to match the needs for relational operations like joins, vector aggregation, distinct, union, and so on. Repartitioning is done in streaming mode and no storage is associated with it. Note that repartitioning is different from the alter table repartition command, where static repartitioning is done.

As mentioned before, a query plan consists of query execution operators. In Adaptive Server 15.0, operators belong to one of two categories:

- Attribute-insensitive operators include scans, union alls, and scalar aggregation. They are not concerned about the underlying partitions.
- Attribute-sensitive operators (for example, join, distinct, union, and vector aggregation operators) allow for an operation on a given amount of data to be broken into a smaller number of operations on smaller fragments of the data using semantics-based partitioning. Afterwards, a simple union all provides the final result set. The union all is implemented using a many-to-one exchange operator.

The following sections discuss these two classes of operators. The examples in these sections use the following table with enough data to trigger parallel processing.

```
create table RA2(a1 int, a2 int, a3 int)
```

Parallelism of attribute-insensitive operation

This section discusses the attribute-insensitive operations, which include scans (serial and parallel), scalar aggregations, and union alls.

Table scan

For horizontal parallelism, either at least one of the tables in the query must be partitioned or the configuration parameter max repartition degree must be greater than 1. If max repartition degree is set to 1, Adaptive Server uses the number of online engines as a hint. When Adaptive Server runs horizontal parallelism, it runs multiple versions of one or more operators in parallel. Each clone of an operator works on its partition, which can be statically created or dynamically built at execution.

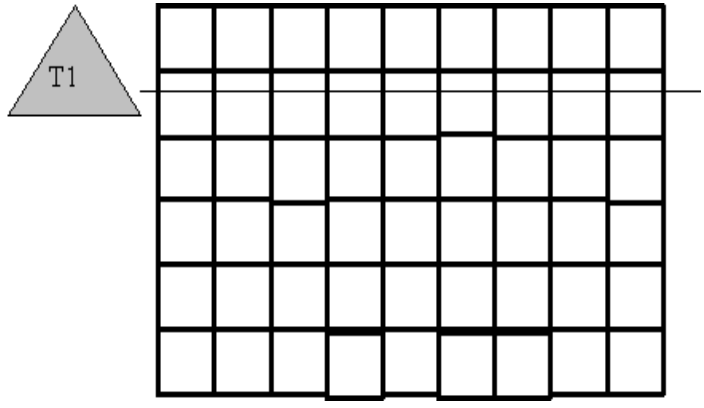
Serial table scan

The example below shows the serial execution of a query. In this example, the table RA2 is scanned using the Table Scan operator. The result of this operation is routed to the Emit operator, which forwards the result to the client.

```
select * from RA2
QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

     |SCAN Operator
     |  FROM TABLE
     |  RA2
     |  Table Scan.
     |  Forward Scan.
     |  Positioning at start of table.
     |  Using I/O Size 2 Kbytes for data pages.
     |  With LRU Buffer Replacement Strategy for data
        pages.
```

In earlier releases, Adaptive Server does not try to scan an unpartitioned table in parallel using a hash-based scan unless a force option is used. Figure 2-5 shows a scan of an allpages-locked table executed in serial mode by a single task T1. The task follows the page chain of the table to read each page, while doing physical I/O if the needed pages are not in the cache.

Figure 2-5: Serial task scans data pages



Parallel table scan

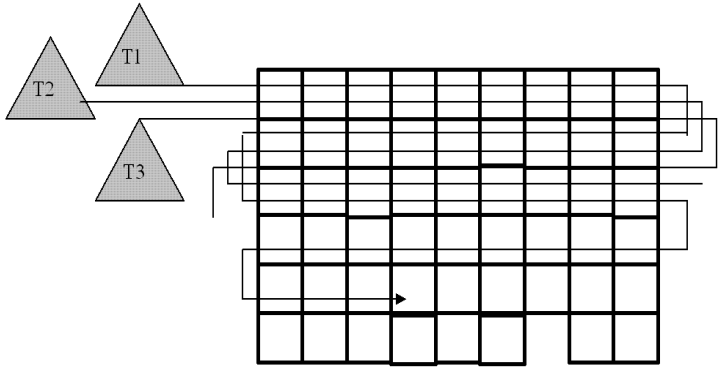
You can force a parallel table scan of an unpartitioned table using Adaptive Server's force option as in earlier releases. In this case, Adaptive Server uses a hash-based scan.

Hash based table scans

Figure 2-6 shows how three worker processes divide the work of accessing data pages from an allpages-locked table during hash-based table scan. Each worker process performs a logical I/O on every page, but each process examines rows on one third of the pages, as indicated by the differently shaded pages. Hash-based table scans are used only if the user forces a parallel degree. See “Partition skew” on page 91 for more information.

With one engine, the query still benefits from parallel access because one work process can execute while others wait for I/O. If there are multiple engines, some of the worker processes could be running simultaneously.

Figure 2-6: Multiple worker processes scans un-partitioned table



Hash based scans increase the logical I/O for the scan, since each worker process must access each page to hash on the page ID. For data-only-locked table, hash-based scans hash either on the extent ID or the allocation page ID, so that only a single worker process scans a page and logical I/O does not increase.

Partitioned based table scans

However, if you partition this table as follows:

```
alter table RA2 partition by range(a1, a2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
```

When the same query is run again, Adaptive Server may choose a parallel scan of the table. Parallel scan is chosen only if there are sufficient pages to scan and the partition sizes are similar enough that the query will benefit from parallelism.

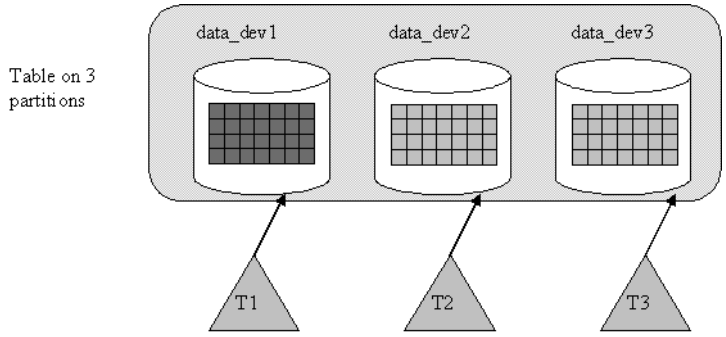
```
select * from RA2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.
```

```
3 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

     |EXCHANGE Operator
     |Executed in parallel by 2 Producer and 1 Consumer
processes.

      |
     |    |EXCHANGE:EMIT Operator
     |    |
     |    |    |SCAN Operator
     |    |    |  FROM TABLE
     |    |    |  RA2
     |    |    |  Table Scan.
     |    |    |  Forward Scan.
     |    |    |  Positioning at start of table.
     |    |    |  Executed in parallel with a 2-way
                   partition scan.
     |    |    |  Using I/O Size 2 Kbytes for data pages.
     |    |    |  With LRU Buffer Replacement Strategy for
data pages.
```

After partitioning the table, the showplan output includes two additional operators, exchange and exchange:emit. This query includes two worker processes, each of which scans a given partition and hands off the data to the exchange:emit operator, as explained in Example A.

Figure 2-7 shows how a query scans a table that has three partitions on three physical disks. With a single engine, this query can benefit from parallel processing because one worker process can execute while others sleep, waiting for I/O or waiting for locks held by other processes to be released. If multiple engines are available, the worker processes can run simultaneously on multiple engines. Such a configuration can perform extremely well.

Figure 2-7: Multiple worker processes access multiple partitions



Index scan

Indexes, like tables, can be partitioned or unpartitioned. Local indexes inherit the partitioning strategy of the table. Each local index partition scans data in one partition only. Global indexes have a different partitioning strategy from the base table; they reference one or more partitions. The following sections describe the index configurations supported by Adaptive Server.

Global non-clustered indexes

Adaptive Server supports global indexes that are non-clustered and unpartitioned for all table partitioning strategies. Global indexes are supported for compatibility with earlier versions of Adaptive Server; they are also useful in OLTP environments. The index and the data partitions can reside on the same or different storage areas.

Non-covered scan of global non-clustered index using hashing

To create an unpartitioned global non-clustered index on table RA2, which is partitioned by range, enter:

```
create index RA2_NC1 on RA2(a3)
```

The next query has a predicate that uses the index key of a3 as follows:

```
select * from RA2 where a3 > 300
QUERY PLAN FOR STATEMENT 1 (at line 1).
. . . . . .
The type of query is SELECT.

ROOT:EMIT Operator

        |EXCHANGE Operator
        |Executed in parallel by 3 Producer and 1
         Consumer processes.
```

```
|
 |   |EXCHANGE:EMIT Operator
 |   |
 |   |   |SCAN Operator
 |   |   |  FROM TABLE
 |   |   |  RA2
 |   |   |  Index : RA2_NC1
 |   |   |  Forward Scan.
 |   |   |  Positioning by key.
 |   |   |  Keys are:
 |   |   |    a3 ASC
 |   |   |  Executed in parallel with a 3-way hash
             scan.
 |   |   |  Using I/O Size 2 Kbytes for index leaf
             pages.
 |   |   |  With LRU Buffer Replacement Strategy
              for index leaf pages.
 |   |   |  Using I/O Size 2 Kbytes for data pages.
 |   |   |  With LRU Buffer Replacement Strategy
              for data pages.
```

What is notable in the above example is that Adaptive Server uses an index scan using the index RA2_NC1 using three producer threads spawned by the exchange operator. Each of the producer threads scans all of the qualifying leaf pages and uses a hashing algorithm on the row id of the qualifying data and accesses the data pages that belong to it. The parallelism in this case is exhibited at the data page level.

Figure 2-8: Hash based parallel scan of global non-clustered index

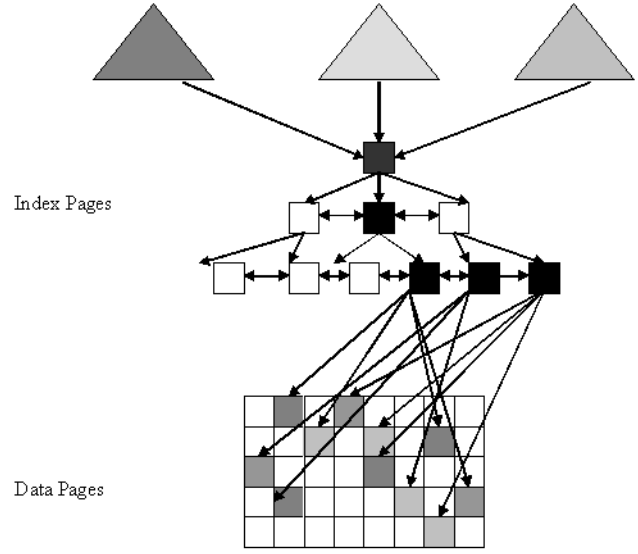
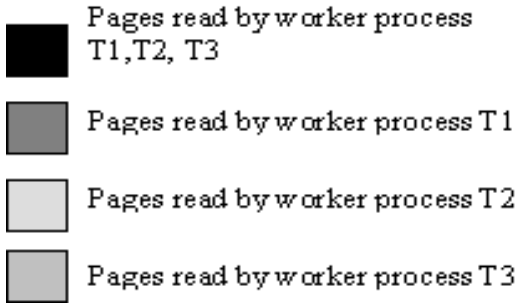


Figure 2-9: Legend for figure 2-8



If the query does not need to access the data page, then it will not be executed in parallel. However, in the current scheme we do have to add the partitioning columns to the query; hence, it becomes a non-covered scan as illustrated in the next example.

```
select a3 from RA2 where a3 > 300

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.
```

```
3 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

     |EXCHANGE Operator
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  RA2
    |   |   |  Index : RA2_NC1
    |   |   |  Forward Scan.
    |   |   |  Positioning by key.
    |   |   |  Keys are:
    |   |   |    a3 ASC
    |   |   |  Executed in parallel with a 2-way hash
                 scan.
    |   |   |  Using I/O Size 2 Kbytes for index leaf
                 pages.
    |   |   |  With LRU Buffer Replacement Strategy for
                 index leaf pages.
    |   |   |  Using I/O Size 2 Kbytes for data pages.
    |   |   |  With LRU Buffer Replacement Strategy for
                 data pages.
```

Covered scan using non-clustered global index

If there is a non-clustered index that includes the partitioning column, then there is no reason for Adaptive Server to access the data pages and the query will be executed in serial. This is illustrated in the next example.

```
create index RA2_NC2 on RA2(a3,a1,a2)

select a3 from RA2 where a3 > 300

QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator
```

```
|SCAN Operator
|  FROM TABLE
|  RA2
|  Index : RA2_NC2
|   Forward Scan.
|   Positioning by key.
|   Index contains all needed columns. Base table
     will not be read.
|   Keys are:
|     a3 ASC
|   Using I/O Size 2 Kbytes for index leaf pages.
|   With LRU Buffer Replacement Strategy for index
    leaf pages.
```

Clustered index scans

With clustered index on APL (all pages) table, no hash based scan strategy is permitted. The only allowable strategy is a partitioned scan. Adaptive Server will use a partitioned scan if that is the right thing to do. For a DOL (data only locked) table, clustered index is usually a placement index, which behaves as a non-clustered index. Hence, all discussions pertaining to a non-clustered index on an APL table apply to a clustered index on a DOL table as well.

Local indexes

Adaptive Server supports clustered and non-clustered local indexes.

Clustered indexes on partitioned tables

Local clustered indexes allow multiple threads to scan each data partition in parallel, which can greatly improve performance. To take advantage of this parallelism, use a partitioned clustered index. Because this is a local index, data is sorted separately within each partition. The information in each data partition conforms to the boundaries established when the partitions were created, which makes it possible to enforce unique index keys across the entire table.

Unique, clustered local indexes have the following restrictions:

- Index columns must include all partition columns.
- Partition columns must have the same order as the index definition's partition key.
- Unique, clustered local indexes cannot be included on a round-robin table with more than one partition.

Non-clustered indexes on partitioned tables

Adaptive Server supports local, non-clustered indexes on partitioned tables.

There is, however, a slight difference when using local indices. When doing a covered index scan of a local non-clustered index, Adaptive Server can still use a parallel scan because the index pages are partitioned as well.

To illustrate the difference, a local non-clustered index is created in the following example.

```
create index RA2_NC2L on RA2(a3,a1,a2) local index

select a3 from RA2 where a3 > 300

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

3 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

     |EXCHANGE Operator
     |Executed in parallel by 2 Producer and 1 Consumer
processes.

     |
     |    |EXCHANGE:EMIT Operator
     |    |
     |    |    |SCAN Operator
     |    |    |  FROM TABLE
     |    |    |  RA2
     |    |    |  Index : RA2_NC2L
     |    |    |  Forward Scan.
     |    |    |  Positioning by key.
     |    |    |  Index contains all needed columns. Base
                  table will not be read.
     |    |    |  Keys are:
     |    |    |    a3 ASC
     |    |    |  Executed in parallel with a 2-way
                   partition scan.
     |    |    |  Using I/O Size 2 Kbytes for index leaf
                   pages.
     |    |    |  With LRU Buffer Replacement Strategy for
                  index leaf pages.
```

Sometimes Adaptive Server will choose a hash-based scan on a local index. This occurs when a different parallel degree is needed or when the data in the partition is skewed such that a hash-based parallel scan is preferred.

Scalar aggregation

The T-SQL scalar aggregation operation can be done in serial or in parallel.

Two phased scalar aggregation

In a parallel scalar aggregation, the aggregation operation is performed in two phases, using two scalar aggregate operators. In the first phase, the lower scalar aggregation operator performs aggregation on the data stream. The result of scalar aggregation from the first phase is merged using a many-to-one exchange operator, and this stream is aggregated a second time.

In case of a count(*) aggregation, the second phase aggregation performs a scalar sum. This is highlighted in the showplan output of the next example.

```
select count(*) from RA2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

5 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

   |SCALAR AGGREGATE Operator
   |  Evaluate Ungrouped SUM OR AVERAGE AGGREGATE.
   |
   |    |EXCHANGE Operator
   |   |Executed in parallel by 2 Producer and 1 Consumer
   processes.

   |    |
   |    |    |EXCHANGE:EMIT Operator
   |    |    |
   |    |    |    |SCALAR AGGREGATE Operator
   |    |    |    |  Evaluate Ungrouped COUNT AGGREGATE.
   |    |    |    |
   |    |    |    |    |SCAN Operator
   |    |    |    |    |  FROM TABLE
   |    |    |    |    |  RA2
   |    |    |    |    |  Table Scan.
   |    |    |    |    |  Forward Scan.
   |    |    |    |    |  Positioning at start of table.
   |    |    |    |    |  Executed in parallel with a 2-way
```

```
                        partition scan.
    |   |   |   |   |  Using I/O Size 2 Kbytes for data
                        pages.
    |   |   |   |   | With LRU Buffer Replacement Strategy
                       for data pages.
```

Serial aggregation

Adaptive Server may also choose to do the aggregation in serial. If the amount of data to be aggregated is not enough to guarantee a performance advantage, a serial aggregation may be the preferred technique. In case of a serial aggregation, the result of the scan is merged using a many-to-one exchange operator. This is shown in the example below, where a very selective predicate has been added to minimize the amount of data flowing into the scalar aggregate operator. In such a case, it probably does not make sense to do the aggregation in parallel.

```
select count(*) from RA2 where a2 = 10

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |SCALAR AGGREGATE Operator
    |  Evaluate Ungrouped COUNT AGGREGATE.
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 2 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  RA2
    |   |   |   |  Table Scan.
    |   |   |   |  Forward Scan.
    |   |   |   |  Positioning at start of table.
    |   |   |   |  Executed in parallel with a 2-way
                   partition scan.
```

```
|   |   |   | Using I/O Size 2 Kbytes for data pages.
|   |   |   |  With LRU Buffer Replacement Strategy
            for data pages.
```

Union all

Union All operators are implemented using a physical operator by the same name. Union All is a fairly simple operation and it pays to parallelize it only when there is a lot of data being moved through it.

Parallel union all

The only pre-condition to generating a parallel union all is that each of its operands must be of the same degree, irrespective of the type of partitioning they have. The following example shows a union all operator being processed in parallel. The position of the exchange operator above the union all operator signifies that it is being processed by multiple threads.

A new table, HA2, is taken to illustrate this next example.

```
create table HA2(a1 int, a2 int, a3 int)
partition by hash(a1, a2) (p1, p2)

select * from RA2
union all
select * from HA2
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 2 Producer and 1 Consumer
     processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |UNION ALL Operator has 2 children.
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
```

```
    |   |   |   |  RA2
    |   |   |   |  Table Scan.
. . . . . . . . . . . . . . . . . .
    |   |   |   |  Executed in parallel with a 2-way
                   partition scan.
. . . . . . . . . . . . . . . . . . .
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  HA2
    |   |   |   |  Table Scan.
    . . . . . . . . . . . . . . . . .
    |   |   |   |  Executed in parallel with a 2-way
                   partition scan.
```

Serial union all

In the next example, the data coming from each side of the union operator is restricted by using selective predicates on either sides. Thus, the amount of data being sent through the union all operator is small enough that Adaptive Server decides not to run them in parallel. Instead, each scan of the tables RA2 and HA2 are serialized by putting 2-to-1 exchange operators on each side of the union. The resultant operands are then processed in serial by the union all operator. This is illustrated in the next query.

```
select * from RA2
where a2 > 2400
union all
select * from HA2
where a3 in (10,20)



Executed in parallel by coordinating process and 4
worker processes.

7 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

     |UNION ALL Operator has 2 children.
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 2 Producer and 1
Consumer processes.
```

```
     |    |
     |    |    |EXCHANGE:EMIT Operator
     |    |    |
     |    |    |    |SCAN Operator
     |    |    |    |  FROM TABLE
     |    |    |    |  RA2
     |    |    |    |  Table Scan.
     |    |    |    |  Executed in parallel with a 2-way
partition scan.
     |
     |    |EXCHANGE Operator
     |    |Executed in parallel by 2 Producer and 1
Consumer processes.

     |    |
     |    |    |EXCHANGE:EMIT Operator
     |    |    |
     |    |    |    |SCAN Operator
     |    |    |    |  FROM TABLE
     |    |    |    |  HA2
    |    |    |    |  Table Scan.
     |    |    |    |  Executed in parallel with a 2-way
partition scan.
```

Parallelism of attribute-sensitive operation

This section discusses issues involving the attribute-sensitive operations, which includes such operations as joins, vector aggregations and unions.

join

If two tables are being joined in parallel, Adaptive Server will try to use semantics-based partitioning to make the join more efficient, depending on the amount of data being joined and the type of partitioning that each of the operands have. If the amount of data to be joined is small, but the number of pages to scan for each of the tables is quite significant, Adaptive Server will serialize the parallel streams from each side and the join will be done in serial mode. In this case, the query optimizer determines that it is probably suboptimal to run a join operation in parallel. In general, one or both of the operands used for the join operators may be any intermediate operator, like another join or a grouping operator, but the examples used show only scans as operands.

Tables with same useful partitioning

The partitioning of each operand of a join is useful only with respect to the join predicate. If two tables have same partitioning, and the partitioning columns are a subset of the join predicate, then the tables are said to be equi-partitioned. For example, if you create another table, RB2, which is partitioned similarly to that of RA2, using the following DDL command:

```
create table RB2(b1 int, b2 int, b3 int)
partition by range(b1,b2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
```

and then join RB2 with RA2, the scans and the join can be done in parallel without additional repartitioning. This is possible because Adaptive Server can join the first partition of RA2 with the first partition of RB2 and then the second partition of RA2 with the second partition of RB2. This is called an equi-partitioned join and is possible only if the two tables join on columns a1, b1 and a2, b2 as shown below:

```
select * from RA2, RB2
where a1 = b1 and a2 = b2 and a3 < 0

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

7 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

      |EXCHANGE Operator
     |Executed in parallel by 2 Producer and 1 Consumer
processes.

      |
      |   |EXCHANGE:EMIT Operator
      |   |
      |   |   |NESTED LOOP JOIN Operator (Join Type:
Inner Join)
      |   |   |
      |   |   |   |RESTRICT Operator
      |   |   |   |
     |   |   |   |   |SCAN Operator
     |   |   |   |   |  FROM TABLE
     |   |   |   |   |  RB2
     |   |   |   |   |  Table Scan.
     |   |   |   |   |  Forward Scan.
```

```
     |   |   |   |   |  Positioning at start of table.
     |   |   |   |   |  Executed in parallel with a 2-
way partition scan.
     |   |   |
     |   |   |   |RESTRICT Operator
     |   |   |   |
     |   |   |   |   |SCAN Operator
     |   |   |   |   |  FROM TABLE
    |   |   |   |   |  RA2
     |   |   |   |   |  Table Scan.
     |   |   |   |   |  Forward Scan.
     |   |   |   |   |  Positioning at start of table.
     |   |   |   |   |  Executed in parallel with a 2-
way partition scan.
```

The exchange operator is shown above the nested loop join. This implies that it spawns two producer threads: the first scans the first partition of RA2 and RB2 and performs the nested loop join; the second scans the second partition of RA2 and RB2 to do the nested loop join. The two threads then merge the results using a many-to-one (in this case, two-to-one) exchange operator.

One of the tables with useful partitioning

In this example, the table RB2 is repartitioned to a three-way hash partitioning on column b1 using the alter table command.

```
alter table RB2 partition by hash(b1) (p1, p2, p3)
```

Now, take a slightly modified join query as shown below:

```
select * from RA2, RB2 where a1 = b1
```

The partitioning on table RA2 is not useful because the partitioned columns are not a subset of the joining columns (that is, given a value for the joining column `a1`, you cannot say which partition it belongs to). However, the partitioning on RB2 is helpful because it matches the joining column `b1` of RB2. In this case, the query optimizer repartitions table RA2 to match the partitioning of RB2 by using hash partitioning on column `a1` of RA2 (the joining column, which is followed by a three-way merge join). The many to many (2 to 3) exchange operator above the scan of RA2 does this dynamic re-partitioning. The exchange operator above the merge join operator merges the result using a many to one (3 to 1 in this case) exchange operator. Here is the `showplan` output for this query as shown in the following example:

```
select * from RA2, RB2 where a1 = b1

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 5
worker processes.
```

```
10 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator
   |EXCHANGE Operator
   |Executed in parallel by 3 Producer and 1 Consumer
      processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |MERGE JOIN Operator (Join Type: Inner Join)
   |   |   | Using Worktable3 for internal storage.
   |   |   |  Key Count: 1
   |   |   |  Key Ordering: ASC
   |   |   |
   |   |   |   |SORT Operator
   |   |   |   | Using Worktable1 for internal storage.
   |   |   |   |
   |   |   |   |   |EXCHANGE Operator
   |   |   |   |   |Executed in parallel by 2 Producer
                      and 3 Consumer processes.

   |   |   |   |   |
   |   |   |   |   |   |EXCHANGE:EMIT Operator
   |   |   |   |   |   |
   |   |   |   |   |   |   |RESTRICT Operator
   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |SCAN Operator
   |   |   |   |   |   |   |   |  FROM TABLE
   |   |   |   |   |   |   |   |  RA2
   |   |   |   |   |   |   |   |  Table Scan.
   |   |   |   |   |   |   |   |  Forward Scan.
   |   |   |   |   |   |   |   |  Positioning at start
                                    of table.
   |   |   |   |   |   |   |   |  Executed in parallel
                                    with a 2-way
                                    partition scan.
   |   |   |
   |   |   |   |SORT Operator
   |   |   |   | Using Worktable2 for internal storage.
   |   |   |   |
   |   |   |   |   |SCAN Operator
   |   |   |   |   |  FROM TABLE
   |   |   |   |   |  RB2
```

```
    |   |   |   |   |  Table Scan.
    |   |   |   |   |  Forward Scan.
    |   |   |   |   |  Positioning at start of table.
    |   |   |   |   |  Executed in parallel with a 3-way
                       partition scan.
```

Both tables with useless partitioning

In the next example, we have a join where the native partitioning of the tables on both sides is useless. The partitioning on table RA2 is on columns (a1,a2) and that of RB2 is on (b1). The join predicate is on entirely different sets of columns, and the partitioning for both tables does not help at all. One option is to dynamically repartition both sides of the join. This is done by repartitioning table RA2 using a M to N (2 to 3) exchange operator. Adaptive Server chooses column a3 of table RA2 for repartitioning, as it is involved in the join with table RB2. For identical reasons, table RB2 is also repartitioned 3 ways on column b3. The repartitioned operands of the join are equi-partitioned with respect to the join predicate, which means that the corresponding partitions from each side will join. In general, when repartitioning needs to be done on both sides of the join operator, Adaptive Server employs a hash-based partitioning scheme. In the previous example, Adaptive Server will use the same range partitioning as that of table RB2.

```
select * from RA2, RB2 where a3 = b3

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 8
worker processes.

12 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 3 Producer and 1 Consumer
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |MERGE JOIN Operator (Join Type: Inner Join)
    |   |   | Using Worktable3 for internal storage.
    |   |   |  Key Count: 1
    |   |   |  Key Ordering: ASC
    |   |   |
    |   |   |   |SORT Operator
```

```
    |   |   |   | Using Worktable1 for internal storage.
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |Executed in parallel by 2 Producer
and 3 Consumer processes.

    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |RESTRICT Operator
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |  RA2
    |   |   |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |   |   |  Forward Scan.
    |   |   |   |   |   |   |   |  Positioning at start
of table.
    |   |   |   |   |   |   |   |  Executed in parallel
with a 2-way partition scan.
    |   |   |
    |   |   |   |SORT Operator
    |   |   |   | Using Worktable2 for internal storage.
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |Executed in parallel by 3 Producer
and 3 Consumer processes.

    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |  RB2
    |   |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |   |  Forward Scan.
    |   |   |   |   |   |   |  Positioning at start of
table.
    |   |   |   |   |   |   |  Executed in parallel with
a 3-way partition scan.
```

In general, all joins, including nested loop, merge, and hash joins, behave in a similar way. Nested loop joins display one exception, and that is that the inner side of a nested loop join cannot be repartitioned. This limitation occurs because, in the case of a nested loop join, a column value for the joining predicate is pushed from the outer side to the inner side.

Replicated Join

A replicated join is very useful when an index nested loop join needs to be used. Consider a case where a large table with a very useful index on the joining column but useless partitioning, joining to a small table that is un-partitioned or partitioned. In this case, the small table can be replicated N ways to that of the inner table, where N is the number of partitions of the large table. Each partition of the large table is then joined with the small table and, because no xchg operator is needed on the inner side of the join, index nested loop join is permissible. This is illustrated in the next example.

```
create table big_table(b1 int, b2 int, b3 int)
partition by hash(b3) (p1, p2)

create index big_table_nc1 on big_table(b1)

create table small_table(s1 int, a2 int, s3 int)

select * from small_table, big_table
where small_table.s1 = big_table.b1

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3
worker processes.

7 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |NESTED LOOP JOIN Operator (Join Type: Inner
Join)
    |   |   |
    |   |   |   |EXCHANGE Operator
    |   |   |   |Executed in parallel by 1 Producer and
2 Consumer processes.

    |   |   |   |
    |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
```

```
        |   |   |   |   |   |  FROM TABLE
        |   |   |   |   |   |  small_table
        |   |   |   |   |   |  Table Scan.
        |   |   |
        |   |   |   |SCAN Operator
        |   |   |   |  FROM TABLE
        |   |   |   |  big_table
        |   |   |   |  Index : big_table_nc1
        |   |   |   |  Forward Scan.
        |   |   |   |  Positioning by key.
        |   |   |   |  Keys are:
        |   |   |   |    b1 ASC
        |   |   |   |  Executed in parallel with a 2-way
hash scan.
```

Parallel Reformatting

This is especially useful when dealing with a nested loop join. Usually, reformatting refers to materializing the inner side of a nested join into a work table and then creating an index on the joining predicate. With parallel queries and nested loop join, there is another reason to do reformatting. When there is no useful index on the joining column or nested loop, join is the only viable option for a query because of the server/session/query level settings. This becomes an important option for Adaptive Server. As explained before, the outer side may have useful partitioning and, if not, it can be repartitioned to create that useful partitioning. But for the inner side of a nested loop join, any repartitioning means that the table must be reformatted into a work table with the new partitioning strategy. The inner scan of a nested loop join must then access the work table.

In this next example, partitioning for tables RA2 and RB2 is on columns (a1, a2) and (b1, b2) respectively. The query is run with merge and hash join turned off for the session.

```
select * from RA2, RB2 where a1 = b1 and a2 = b3

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 12
worker processes.

17 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

     |SEQUENCER Operator has 2 children.
     |
```

```
    |   |EXCHANGE Operator
    |   |Executed in parallel by 4 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |STORE Operator
    |  |  |  |  Worktable1 created, in allpages locking
mode, for REFORMATTING.
    |   |   |   |  Creating clustered index.
    |   |   |   |
    |   |   |   |   |INSERT Operator
    |   |   |   |   |  The update mode is direct.
    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |   |Executed in parallel by 2
Producer and 4 Consumer processes.
    |   |   |   |   |   |
    |   |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |RESTRICT Operator
    |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |   |  RB2
    |   |   |   |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |   |   |   |  Executed in
parallel with a 2-way partition scan.
    |   |   |   |   |
    |   |   |   |   |  TO TABLE
    |   |   |   |   |  Worktable1.
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 4 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |NESTED LOOP JOIN Operator (Join Type:
Inner Join)
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |Executed in parallel by 2 Producer
and 4 Consumer processes.
```

```
    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |RESTRICT Operator
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |  RA2
    |   |   |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |   |   |  Executed in parallel
with a 2-way partition scan.
    |   |   |   |   |SCAN Operator
    |   |   |   |   |  FROM TABLE
    |   |   |   |   |  Worktable1.
    |   |   |   |   |  Using Clustered Index.
    |   |   |   |   |  Forward Scan.
    |   |   |   |   |  Positioning by key.
```

Note the presence of a sequence operator. In essence, this operator will execute all of its child operators but the last, before executing the last child operator. In this case, it executes the first child operator, which reformats table RB2 into a work table using a four-way hash partitioning on columns `b1` and `b3`. The table RA2 is also repartitioned four ways to match the stored partitioning of the work table.

Serial join

Sometimes, it may not make sense to run a join in parallel because of the amount of data that needs to be joined. If you run a query similar to that of the earlier join queries, but now have predicates on each of the tables (RA2 and RB2) such that the amount of data to be joined is not enough, the join may be done in serial mode. In such a case, it does not matter how these tables are partitioned. The query still benefits from scanning the tables in parallel, as shown in this next example.

```
select * from RA2, RB2 where a1=b1 and a2 = b2
and a3 =  0 and  b2 = 20


QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 4
worker processes.

11 operator(s) under root

The type of query is SELECT.
```

```
ROOT:EMIT Operator

    |MERGE JOIN Operator (Join Type: Inner Join)
    | Using Worktable3 for internal storage.
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |SORT Operator
    |   | Using Worktable1 for internal storage.
    |   |
    |   |   |EXCHANGE Operator
    |   |   |Executed in parallel by 2 Producer and 1
Consumer processes.

    |   |   |
    |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |
    |   |   |   |   |RESTRICT Operator
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |  RA2
    |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |  Executed in parallel with a
2-way partition scan.
    |   |SORT Operator
    |   | Using Worktable2 for internal storage.
    |   |
    |   |   |EXCHANGE Operator
    |   |   |Executed in parallel by 2 Producer and 1
Consumer processes.

    |   |   |
    |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |
    |   |   |   |   |RESTRICT Operator
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |  RB2
    |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |  Executed in parallel with a
2-way partition scan.
```

Semi Joins

Semi-joins, which result from flattening of IN/EXIST subqueries, behave the same way as regular inner joins. The only caveat is that replicated joins are not used for semi-joins, because an outer row can match more than one time in such a situation.

Outer joins

In terms of parallel processing for outer joins, replicated joins are not considered. Everything else behaves in a similar way as regular inner joins. One other point of difference is that no partition elimination is done for any table in an outer join that belongs to the outer group.

Vector aggregation

Vector aggregation refers to queries with group-bys. There are different ways Adaptive Server can perform vector aggregation. The actual algorithms are not described here; only the technique for parallel evaluation is shown in the following sections.

In-partitioned vector aggregation

If any base or intermediate relation requires a grouping and is partitioned on a subset, or the same columns as that of the columns in the group by clause, the grouping operation can be done in parallel on each of the partition and the resultant grouped streams merged using a simple N to 1 exchange. This is because a given group cannot appear in more than one stream. The same goes for grouping over any SQL query as long as you use semantics-based partitioning on the grouping columns or a subset of them. This method of parallel vector aggregation is called in-partitioned aggregation.

The following query uses a parallel in-partitioned vector aggregation since range partitioning is defined on the columns a1 and a2, which also happens to be the column on which the aggregation is needed.

```
select count(*), a1, a2 from RA2 group by a1,a2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
```

```
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |HASH VECTOR AGGREGATE Operator
    |   |   |  GROUP BY
    |   |   |  Evaluate Grouped COUNT AGGREGATE.
    |   |   | Using Worktable1 for internal storage.
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  RA2
    |   |   |   |  Table Scan.
    |   |   |   |  Forward Scan.
    |   |   |   |  Positioning at start of table.
    |   |   |   |  Executed in parallel with a 2-way
partition scan.
    |   |   |   | Using I/O Size 2 Kbytes for data pages.
    |   |   |   |  With LRU Buffer Replacement Strategy
for data pages.
```

Re-partitioned vector aggregation

Sometimes, the partitioning of the table or the intermediate results may not be useful for the grouping operation. It may still be worthwhile to do the grouping operation in parallel by repartitioning the source data to match the grouping columns and then applying the parallel vector aggregation. Such a scenario is shown below, where the partitioning is on columns (a1, a2), but the query requires a vector aggregation on column a1.

```
select count(*), a1 from RA2 group by a1

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 4
worker processes.

6 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |HASH VECTOR AGGREGATE Operator
```

```
    |   |   |  GROUP BY
    |   |   |  Evaluate Grouped COUNT AGGREGATE.
    |   |   | Using Worktable1 for internal storage.
    |   |   |
    |   |   |   |EXCHANGE Operator
    |   |   |   |Executed in parallel by 2 Producer and
2 Consumer processes.

    |   |   |   |
    |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |  RA2
    |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |  Forward Scan.
    |   |   |   |   |   |  Positioning at start of table.
    |   |   |   |   |   |  Executed in parallel with a
2-way partition scan.
```

Two phased vector aggregation

For the query in the previous example, re-partitioning could be expensive. One other possibility would be to do a first level of grouping then merge the data using a N to 1 exchange operator and then do another level of grouping. This is called a two phased vector aggregation. Depending on the number of duplicates for the grouping column, Adaptive Server could reduce the cardinality of the data streaming through the N to 1 exchange, then the second level of grouping will become relatively inexpensive. This is shown in the example below.

```
select count(*), a1 from RA2 group by a1

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

5 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

    |HASH VECTOR AGGREGATE Operator
    |  GROUP BY
    |  Evaluate Grouped SUM OR AVERAGE AGGREGATE.
    | Using Worktable2 for internal storage.
    |
```

```
    |   |EXCHANGE Operator
    |   |Executed in parallel by 2 Producer and 1
Consumer processes.
    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |HASH VECTOR AGGREGATE Operator
    |   |   |   |  GROUP BY
    |   |   |   |  Evaluate Grouped COUNT AGGREGATE.
    |   |   |   | Using Worktable1 for internal storage.
    |   |   |   |
    |   |   |   |   |SCAN Operator
    |   |   |   |   |  FROM TABLE
    |   |   |   |   |  RA2
    |   |   |   |   |  Table Scan.
    |   |   |   |   |  Executed in parallel with a 2-
way partition scan.
```

Note the presence of two vector aggregate operators; hence the name two phase vector aggregation

Serial vector aggregation

As with some of the earlier examples, if the amount of data flowing into the grouping operator is restricted by using a predicate, then executing that in parallel may not make much sense. In such a case, the partitions will be scanned in parallel and a N to 1 exchange is used to serialize the stream followed by a serial vector aggregation. This is shown in the next example.

```
select count(*), a1, a2 from RA2
where a1 between 100 and 200
group by a1, a2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

4 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

   |HASH VECTOR AGGREGATE Operator
   |  GROUP BY
   |  Evaluate Grouped COUNT AGGREGATE.
   | Using Worktable1 for internal storage.
   |
```

```
    |    |EXCHANGE Operator
    |   |Executed in parallel by 2 Producer and 1 Consumer
processes.
    |    |
    |    |    |EXCHANGE:EMIT Operator
    |    |    |
    |    |    |    |SCAN Operator
    |    |    |    |  FROM TABLE
    |    |    |    |  RA2
    |    |    |    |  Positioning at start of table.
    |    |    |    |  Executed in parallel with a 2-way
partition scan.
```

The bottom line is that you cannot always group on the partitioning columns, or take advantage of a table that is already partitioned on the grouping columns. It is up to the query optimizer to determine if it is better to repartition and perform the grouping in parallel, or merge the data stream in a partitioned table and do the grouping in serial or a two phased aggregation.

Distinct

Queries with distinct operations can be thought to be grouped vector aggregation without the aggregation part. For example:

```
select distinct a1, a2 from RA2
```

is same as saying:

```
select a1, a2 from RA2 group by a1, a2
```

All of the methodologies that are applicable to vector aggregates are applicable here as well.

Queries with IN list

Adapative Server uses a very optimized technique to handle IN list. This is a common SQL construct. So, a construct like:

```
col in (value1, value2,..valuek)
```

is same as saying:

```
col = value1 OR col = value2 OR .... col = valuek
```

The values in the IN list is put into a special in-memory table and sorted for duplicates removal. Then, it is joined back with the base table using an index nested loop join. The following example illustrates this phenomenon with two values in the IN list that corresponds to two values in the OR list as shown in the lines:

```
SCAN Operator
FROM OR List
OR List has up to 2 rows of OR/IN values.
```

```
select * from RA2 where a3 in (1425, 2940)

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

6 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

  |EXCHANGE Operator
  |Executed in parallel by 2 Producer and 1 Consumer
processes.

  |
  |   |EXCHANGE:EMIT Operator
  |   |
  |   |   |NESTED LOOP JOIN Operator (Join Type: Inner
Join)
  |   |   |
  |   |   |   |SCAN Operator
  |   |   |   |  FROM OR List
  |   |   |   |  OR List has up to 2 rows of OR/IN values.
  |   |   |
  |   |   |   |RESTRICT Operator
  |   |   |   |
  |   |   |   |   |SCAN Operator
  |   |   |   |   |  FROM TABLE
  |   |   |   |   |  RA2
  |   |   |   |   |  Index : RA2_NC1
  |   |   |   |   |  Forward Scan.
  |   |   |   |   |  Positioning by key.
  |   |   |   |   |  Keys are:
  |   |   |   |   |    a3 ASC
  |   |   |   |   |  Executed in parallel with a 2-way
hash scan.
```

Queries with OR clauses

Adaptive Server can take a disjunctive predicate like an OR clause and apply each side of the disjunction separately to qualify a set of row ids (RIDs). The important point to note is that the set of conjunctive predicates on each side of the disjunction must be indexable. Also, the conjunctive predicates on each side of the disjunction cannot have further disjunction within them; that is, it makes little sense to use an arbitrarily deep nesting of disjunctive and conjunctive clauses. In the next example, a disjunctive predicate is taken on the same column (you can have predicates on different columns as long as you have indices that can do inexpensive scans), but the predicates may qualify an overlapping set of data rows. Adaptive Server uses the predicates on each side of the disjunction separately and qualifies a set of row ids. These row ids are then subjected to duplicate elimination.

```
select a3 from RA2 where a3 = 2955 or a3 > 2990

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

8 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

  |EXCHANGE Operator
  |Executed in parallel by 2 Producer and 1 Consumer
processes.

  |
  |   |EXCHANGE:EMIT Operator
  |   |
  |   |   |RID JOIN Operator
  |   |   | Using Worktable2 for internal storage.
  |   |   |
  |   |   |   |HASH UNION Operator has 2 children.
  |   |   |   | Using Worktable1 for internal storage.
  |   |   |   |
  |   |   |   |   |SCAN Operator
  |   |   |   |   |  FROM TABLE
  |   |   |   |   |  RA2
  |   |   |   |   |  Index : RA2_NC1
  |   |   |   |   |  Forward Scan.
  |   |   |   |   |  Positioning by key.
  |   |   |   |   |  Index contains all needed
```

```
columns.Base table will not be read.
   |   |   |   |   |  Keys are:
   |   |   |   |   |    a3 ASC
   |   |   |   |   |  Executed in parallel with a 2-way
hash scan.
   |   |   |   |
   |   |   |   |   |SCAN Operator
   |   |   |   |   |  FROM TABLE
   |   |   |   |   |  RA2
   |   |   |   |   |  Index : RA2_NC1
   |   |   |   |   |  Forward Scan.
   |   |   |   |   |  Positioning by key.
   |   |   |   |   |  Index contains all needed columns.
Base table will not be read.
   |   |   |   |   |  Keys are:
   |   |   |   |   |    a3 ASC
   |   |   |   |   |  Executed in parallel with a 2-way
hash scan.
   |   |   |   |RESTRICT Operator
   |   |   |   |
   |   |   |   |   |SCAN Operator
   |   |   |   |   |  FROM TABLE
   |   |   |   |   |  RA2
   |   |   |   |   |  Using Dynamic Index.
   |   |   |   |   |  Forward Scan.
   |   |   |   |   |  Positioning by Row IDentifier (RID.)
   |   |   |   |   |  Using I/O Size 2 Kbytes for data
pages.
   |   |   |   |   |  With LRU Buffer Replacement Strategy
for data pages.
```

As can be seen, two separate index scans are employed using the index RA2_NC1, which is defined on the column a3. The qualified set of row ids are then checked for duplicate row ids and finally joined back to the base table. Note the line "Positioning by Row Identifier (RID)". Different indices for each side of the disjunction can be used, depending on what the predicates are, as long as they are indexable. One easy way to identify this is to run the query separately with each side of the disjunction to make sure that they are indexable. Adaptive Server may not choose an index intersection if it seems more expensive than a single scan of the table.

Queries with order by clause

If a query requires sorted output because of the presence of an order by clause, Adaptive Server can apply the sort in parallel. First it will try to avoid the sort if there is some inherent ordering available. If it is forced to do the sort, it will see if the sort can be done in parallel. To do that, it may repartition an existing data stream or it may use the existing partitioning scheme and then apply the sort to each of the constituent streams. The resultant data is merged using an N to 1 order, preserving xchg operator.

```
select * from RA2 order by a1, a2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

4 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

   |EXCHANGE Operator
   |Executed in parallel by 2 Producer and 1 Consumer
processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |SORT Operator
   |   |   | Using Worktable1 for internal storage.
   |   |   |
   |   |   |   |SCAN Operator
   |   |   |   |  FROM TABLE
   |   |   |   |  RA2
   |   |   |   |  Index : RA2_NC2L
   |   |   |   |  Forward Scan.
   |   |   |   |  Positioning at index start.
   |   |   |   |  Executed in parallel with a 2-way
partition scan.
```

Depending upon the volume of data to be sorted and the available resources, Adaptive Server may repartition the data stream to a higher degree than the current degree of the stream, so that the sort operation will be even faster. This depends on whether the benefit obtained from doing the sort in parallel far outweighs the overheads of re-partitioning.

Subqueries

When a query contains a subquery, Adaptive Server uses different methods to reduce the cost of processing the subquery. Parallel optimization depends on the type of subquery:

- Materialized subqueries: Parallel query methods are not considered for the materialization step.
- Flattened subqueries: Parallel query optimization is considered only when the subquery is flattened to a regular inner join or a semi join.
- Nested subqueries: Parallel operations are considered for the outermost query block in a query containing a subquery; the inner, nested queries always execute serially. This means that all of the tables in nested subqueries are accessed serially. In the following example, the table RA2 is accessed in parallel, but the result of it is serialized using a 2-to-1 xchg operator before accessing the subquery. The table RB2 inside the subquery is accessed in parallel.

```
select count(*) from RA2 where not exists
(select * from RB2 where RA2.a1 = b1)

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

8 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

   |SCALAR AGGREGATE Operator
   |  Evaluate Ungrouped COUNT AGGREGATE.
   |
   |   |SQFILTER Operator has 2 children.
   |   |
   |   |   |EXCHANGE Operator
   |   |   |Executed in parallel by 2 Producer and 1
Consumer processes.

   |   |   |
   |   |   |   |EXCHANGE:EMIT Operator
   |   |   |   |
   |   |   |   |   |RESTRICT Operator
```

```
    |    |    |    |    |
    |    |    |    |    |    |SCAN Operator
    |    |    |    |    |    |  FROM TABLE
    |    |    |    |    |    |  RA2
    |    |    |    |    |    |  Index : RA2_NC2L
    |    |    |    |    |    |  Forward Scan.
    |    |    |    |    |    |  Executed in parallel with a
2-way partition scan.
    |    |
    |    |  Run subquery 1 (at nesting level 1).
    |    |
   |    |  QUERY PLAN FOR SUBQUERY 1 (at nesting level 1
and at line 2).
    |    |
    |    |   Correlated Subquery.
    |    |   Subquery under an EXISTS predicate.
    |    |
    |    |   |SCALAR AGGREGATE Operator
    |    |   |  Evaluate Ungrouped ANY AGGREGATE.
    |    |   |  Scanning only up to the first qualifying
row.
    |    |   |
    |    |   |    |SCAN Operator
    |    |   |    |  FROM TABLE
    |    |   |    |  RB2
    |    |   |    |  Table Scan.
    |    |   |    |  Forward Scan.
    |    |
    |    |  END OF QUERY PLAN FOR SUBQUERY 1.
```

The following example shows an IN subquery flattened into a semi-join. Actually, Adaptive Server does even better; it converts this into an inner join to provide greater flexibility in shuffling the tables in the join order. As can be seen below, the table RB2, which was originally in the subquery, is now being accessed in parallel.

```
select * from RA2 where a1 in (select b1 from RB2)

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 5
worker processes.

10 operator(s) under root


The type of query is SELECT.
```

```
ROOT:EMIT Operator

   |EXCHANGE Operator
   |Executed in parallel by 3 Producer and 1 Consumer
processes.

   |
   |   |EXCHANGE:EMIT Operator
   |   |
   |   |   |MERGE JOIN Operator (Join Type: Inner Join)
   |   |   | Using Worktable3 for internal storage.
   |   |   |  Key Count: 1
   |   |   |  Key Ordering: ASC
   |   |   |
   |   |   |   |SORT Operator
   |   |   |   | Using Worktable1 for internal storage.
   |   |   |   |
   |   |   |   |   |SCAN Operator
   |   |   |   |   |  FROM TABLE
   |   |   |   |   |  RB2
   |   |   |   |   |  Table Scan.
   |   |   |   |   |  Executed in parallel with a 3-way
partition scan.
   |   |   |
   |   |   |   |SORT Operator
   |   |   |   | Using Worktable2 for internal storage.
   |   |   |   |
   |   |   |   |   |EXCHANGE Operator
   |   |   |   |   |Executed in parallel by 2 Producer
and 3 Consumer processes.

   |   |   |   |   |
   |   |   |   |   |   |EXCHANGE:EMIT Operator
   |   |   |   |   |   |
   |   |   |   |   |   |   |RESTRICT Operator
   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |SCAN Operator
   |   |   |   |   |   |   |   |  FROM TABLE
   |   |   |   |   |   |   |   |  RA2
   |   |   |   |   |   |   |   |  Index : RA2_NC2L
   |   |   |   |   |   |   |   |  Forward Scan.
   |   |   |   |   |   |   |   |  Positioning at index
start.
   |   |   |   |   |   |   |   |  Executed in parallel
with a 2-way partition scan.
```

select-intos

Queries with select into clauses create a new table to store the query's result set. Adaptive Server optimizes the base query portion of a select into command in the same way it does a standard query, considering both parallel and serial access methods. A select into statement that is executed in parallel:

- Creates the new table using columns specified in the select into statement.
- "Creates N partitions in the new table, where N is the degree of parallelism that the optimizer chooses for the insert operation in the query.
- "Populates the new table with query results, using N worker processes.
- "Unpartitions the new table, if no specific destination partitioning is required.

Performing a select into statement in parallel requires more steps than an equivalent serial query plan. The next example shows a simple select into done in parallel:

```
select * into RAT2 from RA2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

4 operator(s) under root


The type of query is INSERT.

ROOT:EMIT Operator

    |EXCHANGE Operator
    |Executed in parallel by 2 Producer and 1 Consumer
processes.

    |
    |    |EXCHANGE:EMIT Operator
    |    |
    |    |    |INSERT Operator
    |    |    |  The update mode is direct.
    |    |    |
    |    |    |    |SCAN Operator
    |    |    |    |  FROM TABLE
    |    |    |    |  RA2
    |    |    |    |  Table Scan.
```

```
    |   |   |   |  Forward Scan.
    |   |   |   |  Positioning at start of table.
    |   |   |   |  Executed in parallel with a 2-way
partition scan.
    |   |   |
    |   |   |  TO TABLE
    |   |   |  RAT2
    |   |   |  Using I/O Size 2 Kbytes for data pages.
```

In this case, Adaptive Server does not try to increase the degree of the stream coming from the scan of table RA2 and uses it to do a parallel insert into the destination table. The destination table is initially created using round robin partitioning of degree two. After the insert is over, the table is unpartitioned.

If the data set to be inserted is not big enough, Adaptive Server may choose to insert this data in serial. The scan of the source table can still be done in parallel. The destination table is then created as an unpartitioned table.

In Adaptive Server 15.0, the select into clause has been enhanced to allow destination partitioning to be specified. In such a case, the destination table is created using that partitioning, and Adaptive Server finds out the most optimal way to insert data. If the destination table needs to be partitioned the same way as the source data, and there is enough data to insert, the insert operator will be executed in parallel.

The next example shows the same partitioning for source and destination table, and demonstrates that Adaptive Server recognizes this scenario and chooses not to repartition the source data.

```
select * into new_table
partition by range(a1, a2)
(p1 values <= (500,100), p2 values <= (1000, 2000))
from RA2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 2
worker processes.

4 operator(s) under root


The type of query is INSERT.

ROOT:EMIT Operator

   |EXCHANGE Operator
   |Executed in parallel by 2 Producer and 1 Consumer
```

```
processes.

    |
    |   |EXCHANGE:EMIT Operator
    |   |
    |   |   |INSERT Operator
    |   |   |  The update mode is direct.
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  RA2
    |   |   |   |  Table Scan.
    |   |   |   |  Forward Scan.
    |   |   |   |  Positioning at start of table.
    |   |   |   |  Executed in parallel with a 2-way
partition scan.
    |   |   |
    |   |   |  TO TABLE
    |   |   |  RRA2
    |   |   |  Using I/O Size 16 Kbytes for data pages.
```

If the source partitioning does not match that of the destination table's, the source data must be repartitioned. This is illustrated in the next example, where the insert is done in parallel using two worker processes after the data is repartitioned using a 2 to 2 exchange operator that converts the data from range partitioning to hash partitioning.

```
select * into HHA2
partition by hash(a1, a2)
(p1, p2)
from RA2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 4
worker processes.

6 operator(s) under root


The type of query is INSERT.

ROOT:EMIT Operator

|EXCHANGE Operator
|Executed in parallel by 2 Producer and 1 Consumer
processes.
```

```
|
|   |EXCHANGE:EMIT Operator
|   |
|   |   |INSERT Operator
|   |   |  The update mode is direct.
|   |   |
|   |   |   |EXCHANGE Operator
|   |   |   |Executed in parallel by 2 Producer and 2
Consumer processes.

|   |   |   |
|   |   |   |   |EXCHANGE:EMIT Operator
|   |   |   |   |
|   |   |   |   |   |SCAN Operator
|   |   |   |   |   |  FROM TABLE
|   |   |   |   |   |  RA2
|   |   |   |   |   |  Table Scan.
|   |   |   |   |   |  Forward Scan.
|   |   |   |   |   |  Positioning at start of table.
|   |   |   |   |   |  Executed in parallel with a 2-
way partition scan.
|   |   |   |
|   |   |   |  TO TABLE
|   |   |   |  HHA2
|   |   |   |  Using I/O Size 16 Kbytes for data pages.
```

insert/delete/update

Insert, delete, and update operations are done in serial in Adaptive Server 15.0. However, tables other than the destination table used in the query to qualify rows to be deleted or updated can be accessed in parallel.

```
delete from RA2
where exists
(select * from RB2
where RA2.a1 = b1 and RA2.a2 = b2)

QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 3
worker processes.

9 operator(s) under root


The type of query is DELETE.
```

```
ROOT:EMIT Operator

|DELETE Operator
|  The update mode is deferred.
|
|    |NESTED LOOP JOIN Operator (Join Type: Inner Join)
|    |
|    |    |SORT Operator
|    |    | Using Worktable1 for internal storage.
|    |    |
|    |    |    |EXCHANGE Operator
|    |    |    |Executed in parallel by 3 Producer and 1
Consumer processes.

|    |    |    |
|    |    |    |    |EXCHANGE:EMIT Operator
|    |    |    |    |
|    |    |    |    |    |RESTRICT Operator
|    |    |    |    |    |
|    |    |    |    |    |    |SCAN Operator
|    |    |    |    |    |    |  FROM TABLE
|    |    |    |    |    |    |  RB2
|    |    |    |    |    |    |  Table Scan.
|    |    |    |    |    |    |  Forward Scan.
|    |    |    |    |    |    |  Positioning at start of table.
|    |    |    |    |    |    |  Executed in parallel with a
3-way partition scan.
|    |    |    |    |    |    |  Using I/O Size 2 Kbytes for
data pages.
|    |    |    |    |    |    |  With LRU Buffer Replacement
Strategy for data pages.
|    |
|    |    |RESTRICT Operator
|    |    |
|    |    |    |SCAN Operator
|    |    |    |  FROM TABLE
|    |    |    |  RA2
|    |    |    |  Index : RA2_NC1
|    |    |    |  Forward Scan.
|    |    |    |  Positioning by key.
|    |    |    |  Keys are:
|    |    |    |    a3 ASC
|
|  TO TABLE
|  RA2
|  Using I/O Size 2 Kbytes for data pages.
```

As can be seen in this case, the table RB2, which is being deleted, is scanned and deleted in serial. However, table RA2 was scanned in parallel. The same scenario is true for update or insert statements.

Partition elimination

One of the advantages of semantic partitioning is that the query processor may be able to take advantage of it and be able to disqualify partitions at compile time. This is possible for range, hash, and list partitions. With hash partitions, only equality predicates can be used, whereas for range and list partitions equality and in-equality predicates can be used to eliminate partitions. For example, consider table RA2 with its semantic partitioning defined on columns a1, a2 where (p1 values <= (500,100) and p2 values <= (1000, 2000)). If there are predicates on columns a1 or columns a1, a2, then it would be possible to do some partition elimination. For example:

```
select * from RA2 where a1 > 1500
```

does not qualify any data. This can be seen in the showplan output.

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
..............................
|   |   |SCAN Operator
|   |   |  FROM TABLE
|   |   |  RA2
|   |   |  [ Eliminated Partitions : 1 2 ]
|   |   |  Index : RA2_NC2L
```

The phrase "Eliminated Partitions" identifies the partition in accordance with how it was created and assigns an ordinal number for identification. For table RA2, the partition represented by p1 where (a1, a2) <= (500, 100) is considered to be partition number one and p2 where (a1, a2) > (500, 100) and <= (1000, 2000) is identified as partition number two.

Consider an equality query on a hash-partitioned table where all keys in the hash partitioning have an equality clause. This can be shown by taking table HA2, which is hash-partitioned two ways on columns (a1, a2). The ordinal numbers refer to the order in which partitions are listed in the output of sp_help.

```
select * from HA2 where a1 = 10 and a2 = 20

QUERY PLAN FOR STATEMENT 1 (at line 1).
..............................

|SCAN Operator
```

```
|  FROM TABLE
|  HA2
|  [ Eliminated Partitions : 1 ]
|  Table Scan.
```

Partition skew

Partition skew plays a very important part in determining whether a parallel partitioned scan can be employed. Partition skew in Adaptive Server is defined as the ratio of the size of the largest partition to the average size of a partition. Consider a table with 4 partitions of sizes 10, 20, 35, and 80 pages. In this case, the size of the average partition is (20 + 20 + 35 + 85)/4 = 40 pages. The biggest partition has 85 pages so partition skew is calculated as 85/40 = 2.125. In case of partitioned scans, the cost of doing a parallel scan is really as expensive as doing the scan on the largest partition. Instead, a hash-based partition may turn out to be quite fast, as each worker process may hash on a page number or an allocation unit and scan its portion of the data. The penalty paid in terms of loss of performance by skewed partitions is not always at the scan level, but rather as more complex operators like several join operations are built over the data. The margin of error increases exponentially in such cases.

Partition skew can be easily found by running sp_help on a table.

```
sp_help HA2

........
name   type partition_type partitions  partition_keys
------ -------------------- -------------- -----------
HA2   base table        hash               2 a1, a2

partition_name partition_id pages       segment
create_date
-------------------------- ------------ ----------- --
------------
---------------------------
HA2_752002679             752002679       324 default
Aug 10 2005  2:05PM
HA2_768002736             768002736       343 default
Aug 10 2005  2:05PM

Partition_Conditions
--------------------
NULL
```

```
Avg_pages   Max_pages   Min_pages   Ratio(Max/Avg)

Ratio(Min/Avg)
----------- ----------- ----------- ------------------
---------
----------------------------
333         343         324                    1.030030

                                0.972973
```

Alternatively, skew can be calculated by querying the systabstats system catalog, where the number of pages in each partition is listed.

Why queries do not run in parallel

Adaptive Server runs a query in serial when:

- there is not enough data to benefit from parallel access.
- "the query contains no equi-join predicates like:

  ```
  select * from RA2, RB2
  where a1 > b1
  ```

- "there are not enough resources like thread or memory to run a query in parallel.
- "using covered scan of a global non-clustered index.
- "tables/indices are accessed inside a nested sub-query that cannot be flattened.

Run time adjustment

If there are not enough worker processes available at runtime, the execution engine attempts to reduce the number of worker processes used by the xchg operators present in the plan.

It does so in two ways:

- "First, by attempting to reduce the worker process usage of certain xchg operators in the query plan without resorting to serial recompilation of the query. Depending on the semantics of the query plan, certain xchg operators are adjustable and some are not. Some are limited in the way they can be adjusted.

- "Parallel query plans need a minimum number of worker processes to be able to run. When enough worker processes are not available, the query is recompiled serially. When recompilation is not possible, the query is aborted and the appropriate error message is generated.

Adaptive Server 15.0 supports serial recompilation for these type of queries:

- "All ad-hoc select queries, except for select into, alter table and execute immediate queries.
- "All stored procedures except for select into and alter table queries.

Support for select into for ad-hoc and stored procedures will be available in a future release.

Recognizing and managing run time adjustments

Adaptive Server provides two mechanisms to help you observe runtime adjustments of query plans:

- set process_limit_action allows you to abort batches or procedures when runtime adjustments take place or print warnings.
- showplan prints an adjusted query plan when runtime adjustments occur, and showplan is effect.

Using set process_limit_action

The process_limit_action option to the set command lets you monitor the use of adjusted query plans at a session or stored procedure level. When you set process_limit_action to "abort," Adaptive Server records Error 11015 and aborts the query, if an adjusted query plan is required. When you set process_limit_action to "warning," Adaptive Server records Error 11014 but still executes the query. For example, this command aborts the batch when a query is adjusted at runtime:

```
set process_limit_action abort
```

By examining the occurrences of Errors 11014 and 11015 in the error log, you can determine the degree to which Adaptive Server uses adjusted query plans instead of optimized query plans. To remove the restriction and allow runtime adjustments, use:

```
set process_limit_action quiet
```

See set in the Adaptive Server Reference Manual for more information about process_limit_action.

Using showplan

When you use showplan, Adaptive Server displays the optimized plan for a given query before it runs the query. When the query plan involves parallel processing, and a runtime adjustment is made, showplan displays this message, followed by the adjusted query plan:

```
AN ADJUSTED QUERY PLAN IS BEING USED FOR STATEMENT 1
BECAUSE NOT ENOUGH WORKER PROCESSES ARE CURRENTLY
AVAILABLE.

ADJUSTED QUERY PLAN:
```

Adaptive Server does not attempt to execute a query when the set noexec is in effect, so runtime plans are never displayed while using this option.

Reducing the likelihood of runtime adjustments

To reduce the number of runtime adjustments, you must increase the number of worker processes that are available to parallel queries. You can do this either by adding more total worker processes to the system or by restricting or eliminating parallel execution for noncritical queries, as follows:

- Use set parallel_degree to set session-level limits on the degree of parallelism, or
- Use the query-level parallel 1 and parallel N clauses to limit the worker process usage of individual statements.

To reduce the number of runtime adjustments for system procedures, recompile the procedures after changing the degree of parallelism at the server or session level. See sp_recompile in the *Adaptive Server Reference Manual* for more information.

CHAPTER 3

Using *showplan*

This chapter describes the messages printed by the showplan utility. Showplan displays the query plan in a text-based format for each SQL statement in a batch or stored procedure.

Topic	Page
Displaying the query plan	95
Statement level output	96
Lava Query Plan shape	100
Union Operators	140

Displaying the query plan

To see the query plan for a query, use:

```
set showplan on
```

To stop displaying query plans, use:

```
set showplan off
```

You can use showplan in conjunction with other set commands.

To display showplans for a stored procedure, but not execute them, use the set fmtonly command.

See Chapter 32, Query Tuning Tools in *Performance and Tuning: Optimizer and Abstract Plans* for information on how options affect each other's operation.

Note Do not use set noexec with stored procedures—compilation and execution does not occur and you do not receive the necessary output.

Query Plans in ASE 15.0

In Adaptive Server 15.0 there are two kinds of query plans:

- The legacy query plans from pre ASE 15.0 are still used for SQL statements that are not executed by the Lava Query Engine, such as set or create table, etc.
- The query plans chosen by the new optimizer are executed by the Lava Query Execution Engine.

The legacy query plans are unchanged in Adaptive Server 15.0, and their showplan output is also unchanged. The following showplan output is an example of a legacy query plan.

```
1> set showplan off
2> go

QUERY PLAN FOR STATEMENT 1 (at line 1).

     STEP 1
            The type of query is SET OPTION OFF.
```

The query plans that are executed by the Lava Query Engine are very different from those executed by the query engine in earlier versions. Accordingly, the corresponding showplan output has changed significantly. Some of the new features of the Lava query plans that showplan must display are:

- Plan elements – Lava query plans can be composed from over thirty different Lava operators.
- Plan shape – Lava query plans are upside down trees of Lava Operators. In general, more operators in a query plan results in more combinations of possible tree shapes.
- Sub-plans that are executed in parallel.

The rest of this chapter describes the showplan output for Lava Query Plans.

Statement level output

The first section of showplan output for each query plan presents some statement level information. There is always a message giving the statement and line number in the batch or stored procedure of the query for which the query plan was generated:

```
QUERY PLAN FOR STATEMENT N (at line N).
```

A message about abstract plan usage appears next if the query plan was generated using an abstract plan. The message indicates how the abstract plan was forced.

- If an explicit abstract plan was given by a plan clause in the SQL statement, the message is:

  ```
  Optimized using the Abstract Plan in the PLAN clause.
  ```

- If an abstract plan has been internally generated (that is, for alter table and reorg commands that are executed in parallel) the message is:

  ```
  Optimized using the forced options (internally
  generated Abstract Plan).
  ```

- If an abstract plan has been retrieved from *sysqueryplans* because automatic abstract plan usage is enabled, the message is:

  ```
  Optimized using an Abstract Plan (ID : N).
  ```

- If the query plan is a parallel query plan, the following message shows the number of processes (coordinator plus worker) that are required to execute the query plan.

  ```
  Executed in parallel by coordinating process and N
  worker processes.
  ```

- If the query plan was optimized using simulated statistics, this message appears next:

  ```
  Optimized using simulated statistics.
  ```

- ASE uses a scan descriptor for each database object that is accessed during query execution. Each connection (or each worker process for parallel query plans) has 28 scan descriptors by default. If the query plan requires access to more than 28 database objects, auxiliary scan descriptors are allocated from a global pool. If the query plan uses auxiliary scan descriptors, this message is printed, showing the total number required:

  ```
  Auxiliary scan descriptors required: N
  ```

- This message shows the total number of Lava Operators appearing in the query plan:

  ```
  N operator(s) under root
  ```

- The next message shows the type of query for the query plan. For Lava Query Plans, the query type is select, insert, delete, or update:

  ```
  The type of query is SELECT.
  ```

- A final statement level message is printed at the end of showplan output if Adaptive Server has been configured to enable resource limits. The message displays the optimizer's total estimated cost of logical and physical I/O:

  ```
  Total estimated I/O cost for statement N (at line M):
  X.
  ```

The following query, with showplan output, shows some of these messages:

```
1> use pubs2

1> set showplan on

1> select stores.stor_name, sales.ord_num
2> from stores, sales, salesdetail
3> where salesdetail.stor_id = sales.stor_id
4> and stores.stor_id = sales.stor_id
5> plan " ( m_join ( i_scan salesdetailind salesdetail)
6> ( m_join ( i_scan salesind sales ) ( sort ( t_scan
stores ) ) ))"


QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |MERGE JOIN Operator (Join Type: Inner Join)
    | Using Worktable3 for internal storage.
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  salesdetail
    |   |  Index : salesdetailind
    |   |  Forward Scan.
    |   |  Positioning at index start.
    |   |  Index contains all needed columns. Base table
          will not be read.
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for
```

```
          index leaf pages.
|
|    |MERGE JOIN Operator (Join Type: Inner Join)
|    | Using Worktable2 for internal storage.
|    |  Key Count: 1
|    |  Key Ordering: ASC
|    |
|    |    |SCAN Operator
|    |    |  FROM TABLE
|    |    |  sales
|    |    |  Table Scan.
|    |    |  Forward Scan.
|    |    |  Positioning at start of table.
|    |    |  Using I/O Size 2 Kbytes for data pages.
|    |   |  With LRU Buffer Replacement Strategy for
          data pages.
|    |
|    |    |SORT Operator
|    |    | Using Worktable1 for internal storage.
|    |    |
|    |    |    |SCAN Operator
|    |    |    |  FROM TABLE
|    |    |    |  stores
|    |    |    |  Table Scan.
|    |    |    |  Forward Scan.
|    |    |    |  Positioning at start of table.
|    |   |   |   | Using I/O Size 2 Kbytes for data pages.
|    |   |   |    |  With LRU Buffer Replacement Strategy
                    for data pages.
```

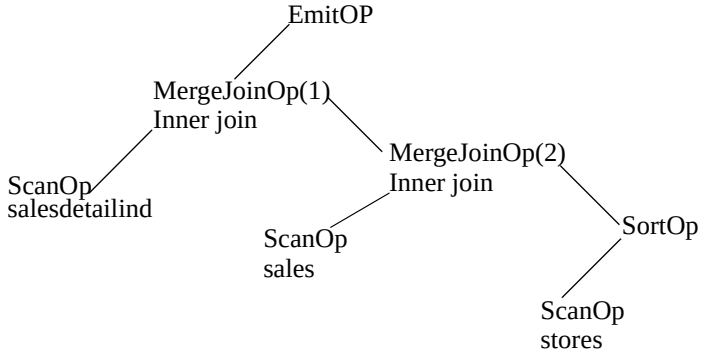
After the statement level output, the query plan is displayed. The showplan output of the query plan consists of two components:

- The names of the Lava Operators (some provide additional information) to show which operations are being executed in the query plan.
- Vertical bars (the " | " symbol) with indentation to show the shape of the query plan operator tree.

Lava Query Plan shape

A Lava Query Plan is an upside down tree of Lava Operators. The position of each operator in the tree determines its order of execution. Execution starts down the left-most branch of the tree and proceeds to the right. To illustrate execution, this section steps through the execution of the query plan for the example, above. Figure 3-1 shows a graphical representation of the query plan.

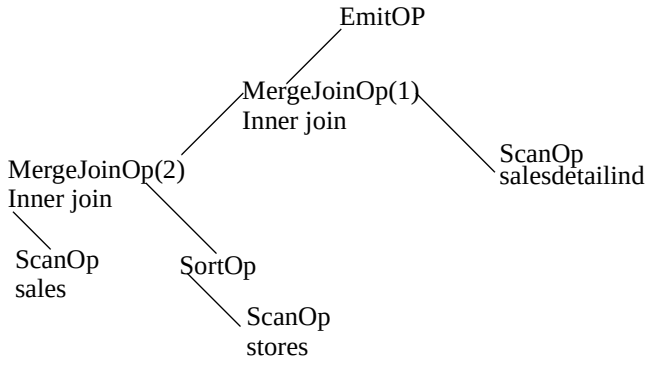
Figure 3-1: Query plan



To generate a result row, the EmitOp calls for a row from its child, the MergeJoinOp(1). MergeJoinOp(1) calls for a row from its left child, the ScanOp for salesdetailind. When it receives a row from its left child, MergeJoinOp(1) calls for a row from its right child, MergeJoinOp(2). MergeJoinOp(2) calls for a row from its left child, the ScanOp for sales. When it receives a row from its left child, MergeJoinOp(2) calls for a row from its right child, the SortOp. The SortOp is a data blocking operator. That is, it needs all of its input rows before it can sort them, so the SortOp keeps calling for rows from its child, the ScanOp for stores, until all rows have been returned. Then the SortOp sorts the rows and passes the first one up to the MergeJoinOp(2). The MergeJoinOp(2) keeps calling for rows from either the left or right child operators until it gets two rows that match on the joining keys. The matching row is then passed up to MergeJoinOp(1). MergeJoinOp(1) also calls for rows from its child operators until a match is found, which is then passed up to the EmitOp to be returned to the client.

Figure 3-2 shows a graphical representation of an alternate query plan for the same example query. This query plan contains all of the same operators, but the shape of the tree is different.

Figure 3-2: Alternate query plan



The showplan output corresponding to the query plan in Figure 3-2 is:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |MERGE JOIN Operator (Join Type: Inner Join)
    | Using Worktable3 for internal storage.
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |MERGE JOIN Operator (Join Type: Inner Join)
    |   | Using Worktable2 for internal storage.
    |   |  Key Count: 1
    |   |  Key Ordering: ASC
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  sales
    |   |   |  Table Scan.
    |   |   |  Forward Scan.
    |   |   |  Positioning at start of table.
    |   |   |  Using I/O Size 2 Kbytes for data pages.
    |   |   |  With LRU Buffer Replacement Strategy for
                data pages.
    |   |
    |   |   |SORT Operator
    |   |   | Using Worktable1 for internal storage.
```

```
|   |   |
|   |   |   |SCAN Operator
|   |   |   |  FROM TABLE
|   |   |   |  stores
|   |   |   |  Table Scan.
|   |   |   |  Forward Scan.
|   |   |   |  Positioning at start of table.
|   | |   | Using I/O Size 2 Kbytes for data
              pages.
|   |   |   |  With LRU Buffer Replacement Strategy
              for data pages.
|
|   |SCAN Operator
|   |  FROM TABLE
|   |  salesdetail
|   |  Index : salesdetailind
|   |  Forward Scan.
|   |  Positioning at index start.
|   | Index contains all needed columns. Base table
       will not be read.
|   | Using I/O Size 2 Kbytes for index leaf pages.
|   |  With LRU Buffer Replacement Strategy for
      index leaf pages.
```

The showplan output conveys the shape of the query plan by using indentation and the " | " symbol to indicate which operators are under which and which ones are on the same or different branches of the tree. There are two rules to interpreting the tree shape:

- The first rule is that the " | " symbols form a vertical line that starts at the operator's name and continue down past all of the operators that are under it on the same branch.
- The second rule is that child operators are printed left to right.

Using these rules, the shape of the query plan in Figure 3-2 can be derived from the previous showplan output with the following steps:

1 The root or emit operator is at the top of the query plan tree: There is no vertical line traveling down from the root, since it would always run from the top to the bottom of the showplan output because the root is always the single topmost operator.

2 The merge join operator (MergeJoinOp(1)) is the left child of the root. The vertical line that starts at MergeJoinOp(1) travels down the length of the entire output, so all of the other operators are below MergeJoinOp(1) and on the same branch.

3 The left child operator of the MergeJoinOp(1) is another merge join operator, (MergeJoinOp(2)).

4 The vertical line that starts at MergeJoinOp(2) travels down past a scan, a sort, and another scan operator before it ends. These operators are all below (or further down the tree) than MergeJoinOp(2).

5 The first SCAN under MergeJoinOp(2) is its left child, the scan of the sales table.

6 The SORT Operator is the right child of MergeJoinOp(2) and the SCAN of the stores table is the only child of the SORT.

7 Below the output for the SCAN of the stores table, several vertical lines end. This indicates that a branch of the tree has ended.

8 The next output is for the SCAN of the salesdetail table. It has the same indentation as MergeJoinOp(2), indicating that it is on the same level. In fact, this SCAN is the right child of MergeJoinOp(1).

Note Most operators are either unary or binary. That is, they have either a single child operator or two child operators directly beneath. Operators that have more than two child operators are called nary. Operators that have no children are leaf operators in the tree and are termed nullary.

Another way to get a graphical representation of the query plan is to use the command set statistics plancost on. See *Adaptive Server Reference Manual: Commands* for more information. This command is used to compare the estimated and actual costs in a query plan. It prints its output as a semi-graphical tree representing the query plan tree.

Lava operators

The Lava Operators were introduced in Chapter 2, “Parallel Query Processing,” and are listed in Table 2-1 of that chapter. In this section, additional messages that give more detailed information about each operator are presented.

Emit operator

The emit operator appears at the top of every Lava Query Plan. It is the root of the query plan tree and always has exactly one child operator. The emit operator routes the result rows of the query by sending them to the client (an application or another Adaptive Server instance) or by assigning values from the result row to local variables or to fetch into variables.

Scan operator

The scan operator reads rows into the Lava Query Plan and makes them available for further processing by the other operators in the query plan. The scan operator is a leaf operator; that is, it never has any child operators. The scan operator can read rows from multiple sources, so the showplan message identifying it is always followed by a from message to identify what kind of scan is being performed. The three from messages are: from cache, from or list, and from table.

From cache

This message shows that a CacheScanOp is reading a single-row in-memory table.

From or list

An or list has up to *N* rows of OR/IN values.

The first message shows that an OrScanOp is reading rows from an in-memory table that contain values from an in-list or multiple or clauses on the same column. The OrScanOp only appears in query plans that use the Special OR strategy for in-lists. The second message shows the maximum number of rows (*N*) that the in-memory table can have. Since the OrScanOp eliminates duplicate values when filling the in-memory table, *N* may be less than the number of values appearing in the SQL statement. As an example, the following query generates a query plan with the Special Or strategy and an OrScanOp:

```
1> select s.id from sysobjects s where s.id in (1, 0,
1, 2, 3)
2> go
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |NESTED LOOP JOIN Operator (Join Type: Inner Join)
    |
    |   |SCAN Operator
    |   |  FROM OR List
    |   |  OR List has up to 5 rows of OR/IN values.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  sysobjects
    |   |  s
    |   |  Using Clustered Index.
    |   |  Index : csysobjects
    |   |  Forward Scan.
    |   |  Positioning by key.
    |   |  Index contains all needed columns. Base table
          will not be read.
    |   |  Keys are:
    |   |    id ASC
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for
          index leaf pages.
```

In this example there are five values in the in-list, but only four are distinct, so the OrScanOp puts only the four distinct values in its in-memory table. In the example query plan, the OrScanOp is the left child operator of the NLJoinOp and a ScanOp is the right child of the NLJoinOp. When this plan is executed, the NLJoinOp calls the OrScanOp to return a row from its in-memory table, then the NLJoinOp calls on the ScanOp to find all matching rows (one at a time), using the clustered index for lookup. This example query plan is much more efficient than reading all of the rows of sysobjects and comparing the value of sysobjects.id in each row to the five values in the in-List.

from table

tablename

correlation name

from table shows that a PtnScanOp is reading a database table. The second message gives the table name, and, if there is a correlation name, that is printed on the next line. Under the from table message in the previous example output, *sysobjects* is the table name and *s* is the correlation name. The previous example also shows several additional messages under the from table message. These messages give more information about how the PtnScanOp is directing the access layer of Adaptive Server to get the rows from the table being scanned.

These messages indicate whether the scan is a table scan or an index scan:

- table scan – indicates that the rows will be fetched by reading the pages of the table.
- Using clustered index – indicates that a clustered index will be used to fetch the rows of the table.
- Index : Indexname – indicates that an index will be used to fetch the rows of the table. If this message is not preceded by the Using Clustered Index message, a non-clustered index is used. *indexname* is the name of the index that will be used.

These messages indicate the direction of a table or index scan. The scan direction depends on the ordering specified when the indexes were created and the order specified for columns in the order by clause.

Backward scans can be used when the order by clause contains the ASC or DESC qualifiers on index keys, in the exact opposite of those in the create index clause.

```
Forward scan

Backward scan
```

The scan-direction messages are followed by positioning messages which describe how access to a table or to the leaf level of an index takes place:

- `Positioning at start of table` – Indicates a table scan that starts at the first row of the table and goes forward.
- `Positioning at end of table` – Indicates a table scan that starts at the last row of the table and goes backward.

- `Positioning by key` – Indicates that the index is used to position the scan at the first qualifying row.
- `Positioning at index start`

  `Positioning at index end` – These messages are similar to the corresponding messages for table scans, except that an index is being scanned instead of a table.

If the scan can be limited due to the nature of the query, the following messages describe how:

- `Scanning only the last page of the table` – This message appears when the scan uses an index and is searching for the MAX value for scalar aggregation. If the index is on the column whose maximum is sought, and the index values are in ascending order, the maximum value will be on the last page.
- `Scanning only up to the first qualifying row` – This message appears when the scan uses an index and is searching for the MIN value for scalar aggregation.

Note If the index key is ordered in descending order, the above messages for min and max aggregates are reversed.

In some cases, the index being scanned contains all of the columns of the table that are needed in the query. In such a case, this message is printed:

```
Index contains all needed columns. Base table will
not be read.
```

The optimizer may choose an index scan over a table scan even though there are no useful keys on the index columns, if the index contains all of the columns needed in the query. The amount of I/O required to read the index can be significantly less than that required to read the base table. Index scans that do not require base table pages to be read are call *covered index scans*.

If an index scan is using keys to position the scan, the following message is printed:

```
Keys are:
     Key [ASC] [DESC]
```

This message shows the names of the columns used as keys (each key on its own output line) and shows the index ordering on that key: ASC for ascending and DESC for descending.

After the messages that describe the type of access being used by the scan operator, messages about the I/O sizes and buffer cache strategy are printed. The I/O messages are:

```
Using I/O size N Kbytes for data pages.
Using I/O size N Kbytes for index leaf pages.
```

I/O size messages

```
Using I/O size N Kbtyes for data pages.

Using I/O size N Kbtyes for index leaf pages.
```

These messages report the I/O sizes used in the query. The possible sizes are 2K, 4K, 8K, and 16K.

If the table, index, LOB object, or database used in the query uses a data cache with large I/O pools, the optimizer can choose large I/O. It can choose to use one I/O size for reading index leaf pages, and a different size for data pages. The choice depends on the pool size available in the cache, the number of pages to be read, the cache bindings for the objects, and the cluster ratio for the table or index pages.

Either or both of these messages can appear in the showplan output for a scan operator. For a table scan, only the first message is printed; for a covered index scan, only the second message is printed. For an index scan that requires base table access, both messages are printed.

After each I/O size message, a cache strategy message is printed:

```
With <LRU/MRU> Buffer Replacement Strategy for data
pages.

With <LRU/MRU> Buffer Replacement Strategy for index
leaf pages.
```

Sample I/O and cache messages are shown in the following query:

```
1> use pubs2
1> set showplan on
1> select au_fname, au_lname, au_id from authors
2> where au_lname = "Williams"

QUERY PLAN FOR STATEMENT 1 (at line 1).

1 operator(s) under root


The type of query is SELECT.
```

```
ROOT:EMIT Operator

    |SCAN Operator
    |  FROM TABLE
    |  authors
    |  Index : aunmind
    |  Forward Scan.
    |  Positioning by key.
    |  Keys are:
    |    au_lname ASC
    |  Using I/O Size 2 Kbytes for index leaf pages.
    |  With LRU Buffer Replacement Strategy for index
       leaf pages.
    |  Using I/O Size 2 Kbytes for data pages.
    |  With LRU Buffer Replacement Strategy for data
       pages.
```

The scan of the *authors* table uses the index aunmind, but must also read the base table pages to get all of the required columns from *authors*. In this example, there are two I/O size messages, each followed by the corresponding buffer replacement message.

Finally, there are two special kinds of table scan operators that have their own special messages: The rid scan and the log scan.

RID Scan

The RID scan is only found in query plans that use the second or strategy that the optimizer can choose, the general or strategy. The General or strategy may be chosen used when multiple or clauses are present on different columns. An example of a query for which the optimizer can choose a general or strategy and its showplan output is:

```
1> use pubs2
1> set showplan on
1> select id from sysobjects where id = 4 or name = 'foo'

QUERY PLAN FOR STATEMENT 1 (at line 1).

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator
```

```
|RID JOIN Operator
| Using Worktable2 for internal storage.
|
|   |HASH UNION Operator has 2 children.
|   | Using Worktable1 for internal storage.
|   |
|   |   |SCAN Operator
|   |   |  FROM TABLE
|   |   |  sysobjects
|   |   |  Using Clustered Index.
|   |   |  Index : csysobjects
|   |   |  Forward Scan.
|   |   |  Positioning by key.
|   |   |  Index contains all needed columns. Base
           table will not be read.
|   |   |  Keys are:
|   |   |    id ASC
|   |   |  Using I/O Size 2 Kbytes for index leaf
           pages.
|   |   |  With LRU Buffer Replacement Strategy for
           index leaf pages.
|   |
|   |   |SCAN Operator
|   |   |  FROM TABLE
|   |   |  sysobjects
|   |   |  Index : ncsysobjects
|   |   |  Forward Scan.
|   |   |  Positioning by key.
|   |   |  Index contains all needed columns. Base
           table will not be read.
|   |   |  Keys are:
|   |   |    name ASC
|   |   |  Using I/O Size 2 Kbytes for index leaf
           pages.
|   |   |  With LRU Buffer Replacement Strategy for
           index leaf pages.
|
|   |RESTRICT Operator
|   |
|   |   |SCAN Operator
|   |   |  FROM TABLE
|   |   |  sysobjects
|   |   |  Using Dynamic Index.
|   |   |  Forward Scan.
|   |   |  Positioning by Row IDentifier (RID).
|   |   |  Using I/O Size 2 Kbytes for data pages.
```

```
    |   |   | With LRU Buffer Replacement Strategy for
              data pages.
```

In this example, the where clause contains two disjuncts, each on a different column (id and name) . There are indexes on each of these columns (csysobjects and ncsysobjects), so the optimizer chose a query plan that uses an index scan to find all rows whose id-column is 4 and another index scan to find all rows whose name is “foo”. Since it is possible that a single row has both an id of 4 and a name of “foo,” that row would appear twice in the result set. To eliminate these duplicate rows, the index scans only return the Row Identifiers (RIDs) of the qualifying rows. The two streams of RIDs are concatenated by the hash union operator, which also removes any duplicate RIDs. The stream of unique RIDs is passed to the rid join operator. The rid join operator creates a worktable and fills it with a single-column row with each RID. The rid join operator then passes its worktable of RIDs to the rid scan operator. The rid scan operator passes the worktable to the access layer, where it is treated as a keyless non-clustered index and the rows corresponding to the RIDs are fetched and returned. The last scan in the showplan output is the rid scan. As can be seen from the example output, the rid scan output contains many of the messages already discussed above, but it also contains two messages that are only printed for the rid scan:

- `Using Dynamic Index` – This message indicates that the scan is using the worktable with RIDs that was built during execution by the rid join operator as an index to locate the matching rows.
- `Positioning by Row Identifier (RID)` – This message indicates that the rows are being located directly by the RID.

Log scan

log scan appear only in triggers that access inserted or deleted tables. These tables are dynamically built by scanning the transaction log when the trigger is executed. Triggers are only be executed after insert, delete, or update queries modify a table with a trigger defined on it for the specific query type. The following example is a delete query on the *titles* table, which has a delete trigger called deltitle defined on it:

```
1> use pubs2
1> set showplan on
1> delete from titles where title_id = 'xxxx'

QUERY PLAN FOR STATEMENT 1 (at line 1).

2 operator(s) under root
```

```
The type of query is DELETE.

ROOT:EMIT Operator

    |DELETE Operator
    |  The update mode is direct.
    |
    |    |SCAN Operator
    |    |  FROM TABLE
    |    |  titles
    |    |  Using Clustered Index.
    |    |  Index : titleidind
    |    |  Forward Scan.
    |    |  Positioning by key.
    |    |  Keys are:
    |    |    title_id ASC
    |    |  Using I/O Size 2 Kbytes for data pages.
    |    | With LRU Buffer Replacement Strategy for data
           pages.
    |
    |  TO TABLE
    |  titles
    |  Using I/O Size 2 Kbytes for data pages.
```

The showplan output up to this point is for the actual delete query. The output below is for the trigger, deltitle.

```
QUERY PLAN FOR STATEMENT 1 (at line 5).

6 operator(s) under root

The type of query is COND.

ROOT:EMIT Operator

    |RESTRICT Operator
    |
    |    |SCALAR AGGREGATE Operator
    |    |  Evaluate Ungrouped COUNT AGGREGATE.
    |    |
    |    |   |MERGE JOIN Operator (Join Type: Inner Join)
    |    |    | Using Worktable2 for internal storage.
    |    |    |  Key Count: 1
    |    |    |  Key Ordering: ASC
    |    |    |
    |    |    |   |SORT Operator
```

```
    |   |   |   | Using Worktable1 for internal storage.
    |   |   |   |
    |   |   |   |   |SCAN Operator
    |   |   |   |   |  FROM TABLE
    |   |   |   |   |  titles
    |   |   |   |   |  Log Scan.
    |   |   |   |   |  Forward Scan.
    |   |   |   |   |  Positioning at start of table.
    |   |   |   |   |  Using I/O Size 2 Kbytes for data
                        pages.
    |   |   |   |   |  With MRU Buffer Replacement
                        Strategy for data pages.
    |   |   |
    |   |   |   |SCAN Operator
    |   |   |   |  FROM TABLE
    |   |   |   |  salesdetail
    |   |   |   |  Index : titleidind
    |   |   |   |  Forward Scan.
    |   |   |   |  Positioning at index start.
    |   |   |   |  Index contains all needed columns.
                    Base table will not be read.
    |   |   |   |  Using I/O Size 2 Kbytes for index
                    leaf pages.
    |   |   |   |  With LRU Buffer Replacement Strategy
                    for index leaf pages.

QUERY PLAN FOR STATEMENT 2 (at line 8).

     STEP 1
          The type of query is ROLLBACK TRANSACTION.

QUERY PLAN FOR STATEMENT 3 (at line 9).

     STEP 1
          The type of query is PRINT.

QUERY PLAN FOR STATEMENT 4 (at line 0).

    STEP 1
         The type of query is GOTO.
```

The procedure that defines the trigger, deltitle, consists of four SQL statements (The SQL text of the trigger definition can be displayed by the command: sp_helptext deltitle). The first statement in deltitle has been compiled into a Lava Query Plan, the other three statements are compiled into legacy query plans and are executed by the Procedural Query Execution Engine, not the Lava Query Execution Engine.

The showplan output for the scan operator for the titles table indicates that it is doing a scan of the log by printing the message: `Log Scan`.

delete, insert, update operators

The DML operators usually have only one child operator. However, they can have up to two additional child operators enforce referential integrity constraints and to deallocate text data in the case of alter table drop of a text column.

The DML operators modify data by inserting, deleting, or updating rows belonging to a target table.

Child operators of DML operators can be scan operators, join operators, or any data streaming operator.

The data modification can be done using different update modes, as specified by this message:

```
The Update Mode is < Update Mode>.
```

The table update mode may be direct, deferred, deferred for an index, or differed for a variable column. The update mode for a worktable is always direct. See the *Performance and Tuning Guide* for more information.

The target table for the data modification is displayed in this message:

```
TO TABLE
<Table Name>
```

Also displayed is the I/O size used for the data modification:

```
Using I/O Size <N> Kbytes for data pages.
```

The next example uses the delete DML operator:

```
1> use pubs2
2> go
1> set showplan on
2> go
1> delete from authors where postalcode = '90210'
2> go
```

```
QUERY PLAN FOR STATEMENT 1 (at line 1).

2 operator(s) under root

The type of query is DELETE.

ROOT:EMIT Operator

    |DELETE Operator
    |  The update mode is direct.
    |
    |    |SCAN Operator
    |    |  FROM TABLE
    |    |  authors
    |    |  Table Scan.
    |    |  Forward Scan.
    |    |  Positioning at start of table.
    |    |  Using I/O Size 4 Kbytes for data pages.
    |    |  With LRU Buffer Replacement Strategy for data
pages.
    |
    |  TO TABLE
    |  authors
    |  Using I/O Size 4 Kbytes for data pages.
```

text delete Operator

Another type of query plan where a DML operator can have more than one child operator is for the alter table drop textcol command, where textcol is the name of a column whose datatype is text, image, or unitext. The following queries and query plan are an example of the use of the text delete operator:

```
1> use tempdb
1> create table t1 (c1 int, c2 text, c3 text)
1> set showplan on
1> alter table t1 drop c2

QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.

5 operator(s) under root

The type of query is ALTER TABLE.

ROOT:EMIT Operator
```

```
    |INSERT Operator
    |  The update mode is direct.
    |
    |    |RESTRICT Operator
    |    |
    |    |    |SCAN Operator
    |    |    |  FROM TABLE
    |    |    |  t1
    |    |    |  Table Scan.
    |    |    |  Forward Scan.
    |    |    |  Positioning at start of table.
    |    |    |  Using I/O Size 2 Kbytes for data pages.
    |    |    |  With LRU Buffer Replacement Strategy for
data pages.
    |
    |    |TEXT DELETE Operator
    |    |  The update mode is direct.
    |    |
    |    |    |SCAN Operator
    |    |    |  FROM TABLE
    |    |    |  t1
    |    |    |  Table Scan.
    |    |    |  Forward Scan.
    |    |    |  Positioning at start of table.
    |    |    |  Using I/O Size 2 Kbytes for data pages.
    |    |    |  With LRU Buffer Replacement Strategy for
data pages.
    |
    |  TO TABLE
    |  #syb__altab
    |  Using I/O Size 2 Kbytes for data pages.
```

In the example, one of the two text columns in t1 is dropped, using the alter table command. The showplan output looks like a select into query plan because alter table internally generated a select into query plan. The insert operator calls on its left child operator, the scan of t1, to read the rows of t1 and builds new rows with only the c1 and c3 columns inserted into #syb_altab. When all of the new rows have been inserted into #syb_altab, the insert Operator calls on its right child, the text delete operator, to delete the text page chains for the c2 columns that have been dropped from t1. Post processing replaces the original pages of t1 with those of #syb_altab to complete the alter table command.

- The text delete operator only appears in alter table commands that drop some, but not all text columns of a table, and it always appears as the right child of an insert operator.

- The deltext operator displays the update mode message, exactly like the update, delete, and insert operators.

Query plans for referential integrity enforcement

When insert, delete, or update operators operate on a table that has one or more referential integrity constraints, the showplan output shows one or two additional child operators of the DML operator. The two additional operators are the direct ri filter operator and the deferred ri filter operator. The kind of referential integrity constraint determines whether one or both of these operators are present.

The following example is for an insert into the *titles* table of the pubs3 database. This table has a column called pub_id that references the pub_id column of the *publishers* table. The referential integrity constraint on titles.pub_id requires that every value that is inserted into titles.pub_id must have a corresponding value in publishers.pub_id.

The query and its query plan are:

```
1> use pubs3
1> set showplan on
1> insert into titles values ("AB1234", "Abcdefg",
"test", "9999", 9.95, 1000.00, 10, null, getdate(),1)

QUERY PLAN FOR STATEMENT 1 (at line 1).

4 operator(s) under root

The type of query is INSERT.

ROOT:EMIT Operator

    |INSERT Operator
    |  The update mode is direct.
    |
    |    |SCAN Operator
    |    |  FROM CACHE
    |
    |    |DIRECT RI FILTER Operator has 1 children.
    |    |
    |    |    |SCAN Operator
    |    |    |  FROM TABLE
    |    |    |  publishers
    |    |    |  Index : publishers_6240022232
    |    |    |  Forward Scan.
```

```
    |   |   |  Positioning by key.
    |   |   |  Index contains all needed columns. Base
table will not be read.
    |   |   |  Keys are:
    |   |   |    pub_id ASC
    |   |   |  Using I/O Size 2 Kbytes for index leaf
pages.
    |   |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
    |
    |  TO TABLE
    |  titles
    |  Using I/O Size 2 Kbytes for data pages.
```

In the query plan, the insert operator's left child operator is a cache scan, which returns the row of values to be inserted into titles. The insert operator's right child is a direct ri filter operator. The direct ri filter operator executes a scan of the *publishers* table to find a row with a value of pub_id that matches the value of pub_id to be inserted into *titles*. If a matching row is found, the direct ri filter operator allows the insert to proceed, but if a matching value of pub_id is not found in *publishers*, the direct ri filter operator aborts the command. In this example, the direct ri filter can check and enforce the referential integrity constraint on *titles* for each row that is inserted, as it is inserted.

The next example shows a direct ri filter operating in a different mode, together with a deferred ri filter operator:

```
1> use pubs3
1> set showplan on
1> update publishers set pub_id = '0001'

QUERY PLAN FOR STATEMENT 1 (at line 1).

13 operator(s) under root

The type of query is UPDATE.

ROOT:EMIT Operator

    |UPDATE Operator
    |  The update mode is deferred_index.
    |
    |    |SCAN Operator
    |    |  FROM TABLE
    |    |  publishers
    |    |  Table Scan.
    |    |  Forward Scan.
```

```
    |   |  Positioning at start of table.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
    |
    |   |DIRECT RI FILTER Operator has 1 children.
    |   |
    |   |   |INSERT Operator
    |   |   |  The update mode is direct.
    |   |   |
    |   |   |   |SQFILTER Operator has 2 children.
    |   |   |   |
    |   |   |   |   |SCAN Operator
    |   |   |   |   |  FROM CACHE
    |   |   |   |
    |   |   |   |  Run subquery 1 (at nesting level 0).
    |   |   |   |
    |   |   |   |  QUERY PLAN FOR SUBQUERY 1 (at nesting
level 0 and at line 0).
    |   |   |   |
    |   |   |   |   Non-correlated Subquery.
    |   |   |   |   Subquery under an EXISTS predicate.
    |   |   |   |
    |   |   |   |   |SCALAR AGGREGATE Operator
    |   |   |   |   |  Evaluate Ungrouped ANY AGGREGATE.
    |   |   |   |   |  Scanning only up to the first
qualifying row.
    |   |   |   |   |
    |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |  titles
    |   |   |   |   |   |  Table Scan.
    |   |   |   |   |   |  Forward Scan.
    |   |   |   |   |   |  Positioning at start of table.
    |   |   |   |   |   |  Using I/O Size 2 Kbytes for
data pages.
    |   |   |   |   |   |  With LRU Buffer Replacement
Strategy for data pages.
    |   |   |   |
    |   |   |   |  END OF QUERY PLAN FOR SUBQUERY 1.
    |   |   |
    |   |   |  TO TABLE
    |   |   |  Worktable1.
    |
    |   |DEFERRED RI FILTER Operator has 1 children.
    |   |
```

```
    |    |    |SQFILTER Operator has 2 children.
    |    |    |
    |    |    |    |SCAN Operator
    |    |    |    |  FROM TABLE
    |    |    |    |  Worktable1.
    |    |    |    |  Table Scan.
    |    |    |    |  Forward Scan.
    |    |    |    |  Positioning at start of table.
    |    |    |    |  Using I/O Size 2 Kbytes for data pages.
    |    |    |    |  With LRU Buffer Replacement Strategy
for data pages.
    |    |    |
    |    |    |  Run subquery 1 (at nesting level 0).
    |    |    |
    |    |    |  QUERY PLAN FOR SUBQUERY 1 (at nesting
level 0 and at line 0).
    |    |    |
    |    |    |   Non-correlated Subquery.
    |    |    |   Subquery under an EXISTS predicate.
    |    |    |
    |    |    |    |SCALAR AGGREGATE Operator
    |    |    |    |  Evaluate Ungrouped ANY AGGREGATE.
    |    |    |    |  Scanning only up to the first qualifying
row.
    |    |    |    |
    |    |    |    |    |SCAN Operator
    |    |    |    |    |  FROM TABLE
    |    |    |    |    |  publishers
    |    |    |    |    |  Index : publishers_6240022232
    |    |    |    |    |  Forward Scan.
    |    |    |    |    |  Positioning by key.
    |    |    |    |    |  Index contains all needed columns.
Base table will not be read.
    |    |    |    |    |  Keys are:
    |    |    |    |    |    pub_id ASC
    |    |    |    |    |  Using I/O Size 2 Kbytes for index
leaf pages.
    |    |    |    |    |  With LRU Buffer Replacement Strategy
for index leaf pages.
    |    |    |
    |    |    |  END OF QUERY PLAN FOR SUBQUERY 1.
    |
    |  TO TABLE
    |  publishers
    |  Using I/O Size 2 Kbytes for data pages.
```

As in the first example, the referential integrity constraint on *titles* requires that for every value of titles.pub_id there must exist a value of publishers.pub_id. However, this example query is changing the values of publisher.pub_id, so a check must be made to maintain the referential integrity constraint. The example query can change the value of publishers.pub_id for several rows in *publishers*, so a check to make sure that all of the values of titles.pub_id still exist in publisher.pub_id cannot be done until all rows of *publishers* have been processed. This example calls for deferred referential integrity checking: As each row of *publishers* is read, the update operator calls upon the direct ri filter operator to search *titles* for a row with the same value of pub_id as the value that is about to be changed. If a row is found, it indicates that this value of pub_id must still exist in *publishers* to maintain the referential integrity constraint on *titles*, so the value of pub_id is inserted into WorkTable1.

After all of the rows of *publishers* have been updated, the update operator calls upon the deferred ri filter operator to execute its subquery to verify that all of the values in Worktable1 still exist in *publishers*: The left child operator of the deferred ri filter is a scan which reads the rows from Worktable1 and the right child is a sq filter operator that executes an existence subquery to check for a matching value in *publishers*. If a matching value is not found, the command is aborted.

The above examples used simple referential integrity constraints, between only two tables. Adaptive Server allows up to 192 constraints per table, so it is possible to generate much more complex query plans. When multiple constraints need to be enforced, there is still only a single direct ri filter or deferred ri filter operator in the query plan, but these operators can have multiple sub-plans, one for each constraint that must be enforced.

join operators

Adaptive Server Enterprise 15.0 provides three primary join strategies. They are the NestedLoopJoin, MergeJoin, and HashJoin. NestedLoopJoin was the primary join strategy in earlier versions. MergeJoin was also available, but was not enabled by default. Adaptive Server Enterprise 15.0 provides a fourth join strategy NaryNestedJoin, which is a variant of NestedLoopJoin.

Each join operator is described in further detail below. A general description of the each algorithm is provided. These descriptions give a high-level overview of the processing required for each join strategy. However, they do not discuss the detailed performance enhancements that have been applied to these strategies.

NestedLoopJoin

NestedLoopJoin is the simplest join strategy. It is a binary operator with the left child forming the outer data stream and the right child forming the inner data stream. For every row from the outer data stream, the inner data stream is opened. Often, the right child is a scan operator. Opening the inner data stream effectively positions the scan on the first row that qualifies all of the searchable arguments (SARGs). The qualifying row is returned to the NestedLoopJoin's parent operator. Subsequent calls to the join operator continue to return qualifying rows from the inner stream. After the last qualifying row from the inner stream is returned for the current outer row, the inner stream is closed. A call is made to get the next qualifying row from the outer stream. The values from this row provide the SARGs used to open and position the scan on the inner stream. This process continues until the NestedLoopJoin's right child returns `End Of Scan` (EOS).

```
1> -- Collect all of the title ids for books written by
"Bloom".
2> select ta.title_id
3>       from titleauthor ta, authors a
4> where a.au_id = ta.au_id
5>       and au_lname = "Bloom"
6> go

QUERY PLAN FOR STATEMENT 1 (at line 2).

3 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |NESTED LOOP JOIN Operator (Join Type: Inner Join)
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  a
    |   |  Index : aunmind
    |   |  Forward Scan.
    |   |  Positioning by key.
    |   |  Keys are:
    |   |    au_lname ASC
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
```

```
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  titleauthor
    |   |  ta
    |   |  Using Clustered Index.
    |   |  Index : taind
    |   |  Forward Scan.
    |   |  Positioning by key.
    |   |  Keys are:
    |   |    au_id ASC
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this example, the *authors* table is being joined with the *titleauthor* table. A NestedLoopJoin strategy has been chosen. Note that the NestedLoopJoin operator's type is “Inner Join.” First, the *authors* table is opened and positioned on the first row (using the aunmind index) containing an l_name value of “Bloom.” Then, the *titleauthor* table is opened and positioned on the first row with an au_id equal to the au_id value of the current *authors*' row using the clustered index “taind”. If there is no useful index for lookups on the inner stream, then the optimizer may generate a reformatting strategy.

Generally, a nested loop join strategy is effective when there are relatively few rows in the outer stream and there is an effective index available for probing into the inner stream.

MergeJoin

The MergeJoin operator is a binary operator. The left and right children are the outer and inner data streams respectively. Both data streams must be sorted on the merge-join's key values. First, a row from the outer stream is fetched. This initializes the merge-join's join key values. Then, rows from the inner stream are fetched until a row with key values that match or are greater than (less than if key column is descending) is encountered. If the join key matches, then the qualifying row is passed on for additional processing, and a subsequent next call to the merge-join operator continues fetching from the currently active stream. If the new values are greater than the current comparison key, then these values are used as the new comparison join key while fetching rows from the other stream. This process continues until one of the data streams is exhausted.

Generally, the MergeJoin strategy is effective when a scan of the data streams requires that most of the rows must be processed and one or both of the streams are already sorted on the join keys.

```
1> -- Collect all of the title ids for books written by
"Bloom".
2> select ta.title_id
3>       from titleauthor ta, authors a
4> where a.au_id = ta.au_id
5>       and au_lname = "Bloom"
6> go

QUERY PLAN FOR STATEMENT 1 (at line 2).

    STEP 1
        The type of query is EXECUTE.
        Executing a newly cached statement.

QUERY PLAN FOR STATEMENT 1 (at line 2).

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |MERGE JOIN Operator (Join Type: Inner Join)
    | Using Worktable2 for internal storage.
    |  Key Count: 1
    |  Key Ordering: ASC
    |
    |   |SORT Operator
    |   | Using Worktable1 for internal storage.
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  authors
    |   |   |  a
    |   |   |  Index : aunmind
    |   |   |  Forward Scan.
    |   |   |  Positioning by key.
    |   |   |  Keys are:
    |   |   |    au_lname ASC
    |   |   |  Using I/O Size 2 Kbytes for index leaf
pages.
    |   |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
```

```
    |   |   |  Using I/O Size 2 Kbytes for data pages.
    |   |   |  With LRU Buffer Replacement Strategy for
data pages.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  titleauthor
    |   |  ta
    |   |  Index : auidind
    |   |  Forward Scan.
    |   |  Positioning at index start.
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for index
leaf pages.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this example, a sort operator is the left child or outer stream. The data source for the sort operator is the *authors* table. The sort operator is required because the *authors* table has no index on au_id that would otherwise provide the necessary sorted order. A scan of the *titleauthor* table is the right child/inner stream. The scan uses the auidind index which provides the necessary ordering for the MergeJoin strategy.

A row is fetched from the outer stream (the *authors* table is the original source) to establish an initial join key comparison value. Then rows are fetched from the *titleauthor* table until a row with a join key equal to or greater than the comparison key is found.

Inner stream rows with matching keys are stored in a cache in case they need to be refetched. These rows are refetched when the outer stream contains duplicate keys. When a titleauthor.au_id value that is greater than the current join key comparison value is fetched, then the MergeJoin operator starts fetching from the outer stream until a join key value equal to or greater than the current titleauthor.au_id value is found. The scan of the inner stream resumes at that point.

The MergeJoin operator's showplan output contains a message indicating what worktable will be used for the inner stream's backing store. The worktable is written to if the inner rows with duplicate join keys no longer fits in cached memory. The width of a cached row is limited to 64KB.

HashJoin

The Hash Join operator is a binary operator. The left child generates the build input stream. The right child generates the probe input stream. The build set is generated by completely draining the build input stream when the first row is requested from the Hash Join operator. Every row is read from the input stream and hashed into an appropriate bucket using the hash key. If there is not enough memory to hold the entire build set, then a portion of it spilled to disk. This portion is referred to as a *hash partition* and should not be confused with table partitions. A hash partition consists of a collection of hash buckets. After the entire left child's stream has been drained, the probe input is read.

Each row from the probe set is hashed. A lookup is done in the corresponding build bucket to check for rows with matching hash keys. This occurs if the build set's bucket is memory resident. If it has been spilled, then the probe row is written to the corresponding spilled probe partition. When a probe row's key matches a build row's key, then the necessary projection of the two row's columns is passed up for additional processing.

Spilled partitions are processed in subsequent recursive passes of the hash join algorithm. New hash seeds are used in each pass so that the data will be redistributed across different hash buckets. This recursive processing continues until the last spilled partition is completely memory resident. When a hash partition from the build set contains a lot of duplicates, then the hash join operator reverts back to nested loop join processing.

Generally, the hash join strategy is good in cases where most of the rows from the source sets must be processed and there are no inherent useful orderings on the join keys or there are no interesting orderings that can be promoted to calling operators (for example, an order by clause on the join key). Hash joins perform particularly well if one of the datasets is small enough to be memory resident. In this case, no spilling occurs and no I/O is needed to perform that hash join algorithm.

```
1> -- Collect all of the title ids for books written by
"Bloom".
2> select ta.title_id
3>       from titleauthor ta, authors a
4> where a.au_id = ta.au_id
5>       and au_lname = "Bloom"

QUERY PLAN FOR STATEMENT 1 (at line 2).

3 operator(s) under root

The type of query is SELECT.
```

```
ROOT:EMIT Operator

    |HASH JOIN Operator (Join Type: Inner Join)
    | Using Worktable1 for internal storage.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  a
    |   |  Index : aunmind
    |   |  Forward Scan.
    |   |  Positioning by key.
    |   |  Keys are:
    |   |    au_lname ASC
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  titleauthor
    |   |  ta
    |   |  Index : auidind
    |   |  Forward Scan.
    |   |  Positioning at index start.
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this example, the source of the build input stream is an index scan of author.aunmind.

Only rows with an au_lname value of “Bloom” are returned from this scan. These rows are then hashed on their au_id value and placed into their corresponding hash bucket. After the initial build phase is completed, the probe stream is opened and scanned. Each row from the source index, titleauthor.auidind, is hashed on the au_id column. The resulting hash value is used to determine which bucket in the build set should be searched for matching hash keys. Each row from the build set's hash bucket is compared to the probe row's hash key for equality. If the row matches, then the titleauthor.au_id column is returned to the Emit Operator.

The Hash Join Operator's showplan output contains a message indicating what worktable will be used for the spilled partition's backing store. The input row width is limited to 64KB.

NaryNestedLoopJoin operator

The Nary Nested Loop Join strategy is never evaluated or chosen by the optimizer. It is an operator that is constructed during code generation. If the compiler finds series of two or more left-deep nested looped joins, then it attempts to transform them into an Nary Nested Loop Join Operator. Two additional requirements allow for transformation scan; each Nested Loop Join Operator has an "inner join" type and the right child of each nested loop join is a Scan Operator. A Restrict Operator is permitted above the Scan Operator.

Nary Nested Loop Join execution has a performance benefit over the execution of a series of Nested Loop Join Operators. The example below demonstrates this. There is one fundamental difference between the two methods of execution. With a series of nested loop joins, a scan may eliminate rows based on SARG values initialized by an earlier scan. That scan may not be the one that immediately preceded the failing scan. With a series of nested looped joins, the previous scan would be completely drained although it has no effect on the failing scan. This could result in a significant amount of needless I/0. With Nary Nested Loop Joins, the next row fetched comes from the scan that produced the failing SARG value. This is far more efficient.

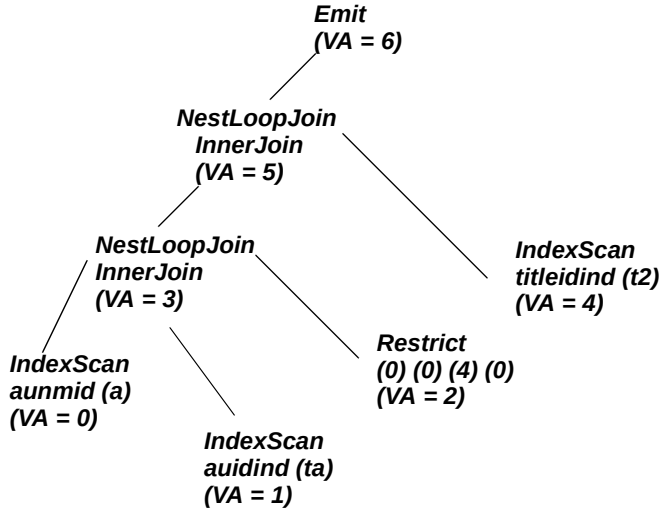
```
1> -- Collect the author id and name for all authors
with the
2> -- last name "Bloom" and who have a listed title and
the
3> -- author id is the same as the title_id.
4> select a.au_id, au_fname, au_lname
5>     from titles t, titleauthor ta, authors a
6> where a.au_id = ta.au_id
7>     and ta.title_id = t.title_id
8>     and a.au_id = t.title_id
```

```
9>     and au_lname = "Bloom"

5 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |N-ARY NESTED LOOP JOIN Operator has 3 children.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  a
    |   |  Index : aunmind
    |   |  Forward Scan.
    |   |  Positioning by key.
    |   |  Keys are:
    |   |    au_lname ASC
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
    |
    |   |RESTRICT Operator
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  titleauthor
    |   |   |  ta
    |   |   |  Index : auidind
    |   |   |  Forward Scan.
    |   |   |  Positioning by key.
    |   |   |  Keys are:
    |   |   |    au_id ASC
    |   |   |  Using I/O Size 2 Kbytes for index leaf
pages.
    |   |   |  With LRU Buffer Replacement Strategy for
index leaf pages.
    |   |   |  Using I/O Size 2 Kbytes for data pages.
    |   |   |  With LRU Buffer Replacement Strategy for
data pages.
    |
    |   |SCAN Operator
```

```
        |    |  FROM TABLE
        |    |  titles
        |    |  t
        |    |  Using Clustered Index.
        |    |  Index : titleidind
        |    |  Forward Scan.
        |    |  Positioning by key.
        |    |  Keys are:
        |    |    title_id ASC
        |    |  Using I/O Size 2 Kbytes for data pages.
        |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this example, there are a series of nested loop joins as depicted by the tree below:

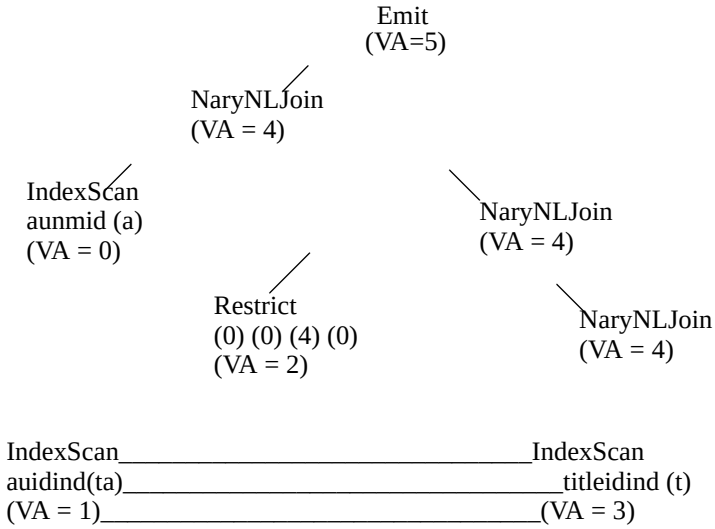
Figure 3-3: Lava operator tree



All Lava operators are assigned a Virtual Address. The lines printed above with VA = in them report the Virtual Address for a given operator.

The effective join order is authors, titleauthor, titles. A Restrict Operator is the parent operator of the scan on titleauthors. This plan is transformed into the Nary Nested Loop Join plan below:

Figure 3-4: Lava operator NaryNestedLoop



Note that the transformation retains the original join order of authors, titleauthor, and titles. In this example, the scan of titles has two SARGs on it. They are ta.title_id = t.title_id and a.au_id = t.title_id. So, the scan of titles could fail because of the SARG value established by the scan of titleauthor or it could fail because of the SARG value established by the scan of authors. If no rows are returned from a scan of titles because of the SARG value set by the scan of authors, then there is no point in continuing the scan of titleauthor. For every row fetched from titleauthor, the scan of titles will fail. It is only when a new row is fetched from authors that the scan of titles might succeed. This is exactly why Nary NLJoins were implemented. They eliminate the useless draining of tables which have no impact on the rows returned by successive scans. In this example, the Nary Nested Loop Join Operator closes the scan of titleauthor, fetches a new row from authors, and repositions the scan of titleauthor based on the au_id fetched from authors. Again, this can be a significant performance improvement as it eliminates the needless draining of the titleauthor table and the associated I/O that could occur.

Distinct operators

There are three Lava Operators that can be used to enforce distinctness. They are the Group Sorted Distinct, Sort Distinct, and Hash Distinct Operators. They are all unary operators. Each has advantages and disadvantages. The optimizer chooses an efficient distinct operator with respect to its use within the entire query plan's context.

Group sorted operator

The Group Sorted Operator is a unary operator. It can be used to apply distinctness. It requires that the input stream is already sorted on the distinct columns. It reads a row from its child operator and initializes the current distinct columns' values to be filtered. The row is returned to the parent operator. When the Group Sorted operator is called again to fetch another row, it fetches another row from its child and compares the values to the current cached values. If the value is a duplicate, then the row is discarded and the child is called again to fetch a new row. This process continues until a new distinct row is found. The distinct columns' values for this row are cached and will be used later to eliminate nondistinct rows. The current row is returned to the parent operator for further processing.

The Group Sorted Operator returns a sorted stream. The fact that it returns a sorted and distinct data stream are properties that the optimizer can exploit to improve performance in additional upstream processing. The Group Sorted Operator is a non-blocking operator. It returns a distinct row to its parent as soon as it is fetched. It does not require that the entire input stream is processed before it can start returning rows.

```
1> -- Collect distinct last and first author names.
2> select distinct au_lname, au_fname
3> from authors
4> where au_lname = "Bloom"

QUERY PLAN FOR STATEMENT 1 (at line 2).

2 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |GROUP SORTED Operator
    |Distinct
    |
```

```
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  Index : aunmind
    |   |  Forward Scan.
    |   |  Positioning by key.
    |   |  Index contains all needed columns. Base table
will not be read.
    |   |  Keys are:
    |   |    au_lname ASC
    |   |  Using I/O Size 2 Kbytes for index leaf pages.
    |   |  With LRU Buffer Replacement Strategy for index
leaf pages.
```

The Distinct Sorted Operator is chosen in this query plan to apply the distinct property because the scan operator is returning rows in sorted order for the distinct columns au_lname and au_fname. By using the Group Sorted Operator here, there is no I/O and minimal CPU overhead.

The Group Sorted Operator can also be used to implement vector aggregation. See “Vector Aggregate Operators” on page 135 for more information. The showplan output prints the line `Distinct` to indicate that this Group Sorted Operator is implementing the distinct property.

Sort Distinct Operator

The Sort Distinct Operator is a unary operator. It does not require that its input stream be already sorted on the distinct key columns. It is a blocking operator that drains its child operator's stream and sorts the rows as they are read. A distinct row is returned to the parent operator after all of the rows have been sorted. Rows are returned sorted on the distinct key columns. An internal worktable is used as a backing store in case the input set will not fit entirely in memory.

```
1> select distinct au_lname, au_fname
2> from authors
3> where city = "Oakland"

2 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |SORT Operator
    | Using Worktable1 for internal storage.
```

```
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  Table Scan.
    |   |  Forward Scan.
    |   |  Positioning at start of table.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

The scan of the authors table does not return rows sorted on the distinct key columns. This requires that a Sort Distinct Operator be used rather than a Group Sorted Operator. The sort operator's distinct key columns are au_lname and au_fname. The showplan output indicates that Worktable1 is used for disk storage in case the input set will not fit entirely in memory.

Hash Distinct Operator

The Hash Distinct Operator does not require that its input set be sorted on the distinct key columns. It is a non-blocking operator. Rows are read from the child operator and are hashed on the distinct key columns. This determines the bucket in which the row should reside. The corresponding bucket is searched to see if the key already exists. The row is discarded if it contains a duplicate key and another row is fetched from the child operator. The row is added to the bucket if no duplicate distinct key already exists and the row is passed up to the parent operator for further processing. Rows are not returned sorted on the distinct key columns.

The Hash Distinct Operator is generally used when the input set is not already sorted on the distinct key columns or when the optimizer is not able to exploit the ordering coming out of the distinct processing later in the plan.

```
1> select distinct au_lname, au_fname
2> from authors a
3> where city = "Oakland"
4> go

QUERY PLAN FOR STATEMENT 1 (at line 1).

2 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator
```

```
        |HASH DISTINCT Operator
        | Using Worktable1 for internal storage.
        |
        |   |SCAN Operator
        |   |  FROM TABLE
        |   |  authors
        |   |  a
        |   |  Table Scan.
        |   |  Forward Scan.
        |   |  Positioning at start of table.
        |   |  Using I/O Size 2 Kbytes for data pages.
        |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this example, the output of the authors table scan is not sorted. The optimizer can choose either a sort distinct or hash distinct strategy. The ordering provided by a sort distinct strategy is not useful anywhere else in the plan, so the optimizer will probably choose a hash distinct strategy. The optimizer's decision is ultimately based on cost estimates. The Hash Distinct Operator is typically less expensive because of its ability to eliminate rows as they are processed by aggregating the current row's values. The Sort Distinct Operator cannot eliminate any rows until the entire data set has been sorted.

The showplan output for the Hash Distinct Operator reports that Worktable1 will be used. A worktable is needed in case the distinct row result set cannot fit in memory. In that case, partially processed groups will be spilled to disk.

Vector Aggregate Operators

There are two unary operators used for vector aggregation. They are the Group Sorted Operator and Hash Vector Aggregate Operator.

Grouped Aggregate Message

```
Evaluate Grouped type AGGREGATE.
```

This message is printed by queries that contain group by aggregates.

The type variable indicates the aggregate function being applied; count, sum, average, minimum, or maximum.

Group Sorted Aggregate Operator

The group sorted agg operator is a simple variant of the Group Sorted Distinct Operator described above. It requires that the input set is sorted on the group by columns. The algorithm is very similar. A row is read from the child operator. If it is the start of a new vector, then its grouping columns are cached and the aggregation results are initialized. If the row belongs to the current group being processed, then the aggregate functions are applied to the aggregate results. When the child operator returns a row that starts a new group or End Of Scan, the current vector and its aggregated values are returned to the parent operator.

This is a non-blocking operator similar to the Group Sorted Operator with one difference. The first row in the Group Sorted Aggregate Operator is returned after an entire group is processed, where the first row in the Group Sorted Distinct Operator is returned at the start of a new group.

```
1> -- Collect a list of all cities with the number of
authors that
2> -- live in each city.
3> select city, total_authors = count(*)
4>     from authors
5>     group by city
6> plan
7> "(group_sorted
8>      (sort (scan authors))
9> )"
10> go

QUERY PLAN FOR STATEMENT 1 (at line 3).
Optimized using the Abstract Plan in the PLAN clause.

3 operator(s) under root


The type of query is SELECT.

ROOT:EMIT Operator

    |GROUP SORTED Operator
    |  Evaluate Grouped COUNT AGGREGATE.
    |
    |   |SORT Operator
    |   | Using Worktable1 for internal storage.
    |   |
    |   |   |SCAN Operator
    |   |   |  FROM TABLE
    |   |   |  authors
```

```
        |   |   |  Table Scan.
        |   |   |  Forward Scan.
        |   |   |  Positioning at start of table.
        |   |   |  Using I/O Size 2 Kbytes for data pages.
        |   |   |  With LRU Buffer Replacement Strategy for
data pages.
```

In this query plan, the scan of authors does not return rows in grouping order. A Sort Operator is applied to order the stream based on the grouping column city. At this point, a Group Sorted Vector Aggregate Operator can be applied to evaluate the count() aggregate.

The Group Sorted Vector Aggregate Operator showplan output reports the aggregate functions being applied as:

```
    |  Evaluate Grouped COUNT AGGREGATE.
```

Hash vector aggregate operator

The hash vector aggregate operator is a blocking operator. All rows from the child operator must be processed before the first row from the Hash Vector Aggregate Operator can be returned to its parent operator. Other than this, the algorithm is similar to the Hash Distinct Operator's algorithm.

Rows are fetched from the child operator. Each row is hashed on the query's grouping columns. The bucket that is hashed to is searched to see if the vector already exists.

If the group by values do not exist, then the vector is added and the aggregate values are initialized using this first row. If the group by values do exist, then the current row is aggregated to the existing values.

```
1> -- Collect a list of all cities with the number of
authors that
2> -- live in each city.
3> select city, total_authors = count(*)
4>     from authors
5>     group by city
6> go

QUERY PLAN FOR STATEMENT 1 (at line 3).

2 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator
```

```
    |HASH VECTOR AGGREGATE Operator
    |  GROUP BY
    |  Evaluate Grouped COUNT AGGREGATE.
    | Using Worktable1 for internal storage.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  Table Scan.
    |   |  Forward Scan.
    |   |  Positioning at start of table.
    |   |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this query plan, the Hash Vector Aggregate Operator reads all of the rows from its child operator, which is scanning the authors table. Each row is checked to see if there is already an entry bucket entry for the current city value. If not, a hash entry row is added with the new city grouping value and the count result is initialized to 1. If there is already a hash entry for the new row's city value, then the aggregation function is applied. In this case, the count result is incremented.

The showplan output prints a "GROUP BY" message specifically for the Hash Vector Aggregate Operator, and then prints the grouped aggregate messages:

```
|  Evaluate Grouped COUNT AGGREGATE.
```

The showplan output then reports what worktable will be used to store spilled groups and unprocessed rows:

```
| Using Worktable1 for internal storage.
```

compute by message

“compute by” processing is done in the Emit Operator. It requires that the Emit Operator's input stream be sorted according to any order by requirements in the query. The processing is similar to what is done in the Group Sorted Aggregate Operator. Each row read from the child is checked to see if it starts a new group. If not, the aggregate functions are applied as appropriate to the query's requested groups. If so, then the new group or group's aggregate values are reinitialized from the new row's values.

```
1> -- Collect an ordered list of all cities and report
a count of the
2> -- number of entries for each city after the city's
```

```
list is finished.
3> select city
4>     from authors
5>     order by city
6>     compute count(city) by city
7> go

QUERY PLAN FOR STATEMENT 1 (at line 3).

2 operator(s) under root

The type of query is SELECT.
Emit with Compute semantics

ROOT:EMIT Operator

    |SORT Operator
    | Using Worktable1 for internal storage.
    |
    |    |SCAN Operator
    |    |  FROM TABLE
    |    |  authors
    |    |  Table Scan.
    |    |  Forward Scan.
    |    |  Positioning at start of table.
    |    |  Using I/O Size 2 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

In this example, the Emit Operator's input stream is sorted on the city attribute. For each row, the compute by's count value is incremented. When a new city value is fetched, the count for the previous city's value is returned to the user and the new city's count is reinitialized to one and the new city's value is cached as the new compute by's grouping value.

Union Operators

hash union

The hash union operator uses Adaptive Server 15.0 hashing algorithms to simultaneously perform a union all operation on several data streams and hash-based duplicate elimination.

The hash union operator is a n-ary operator. It displays the following message:

```
HASH UNION OPERATOR has <N> children.
```

N is the number of input streams into the operator.

It also displays the name of the work table it uses, in this format:

```
HASH UNION OPERATOR Using Worktable  <X>  for internal
storage.
```

Example

This example demonstrates the use of hash union.

```
select * from sysindexes
union
select * from sysindexes

QUERY PLAN FOR STATEMENT 1 (at line 8).
Executed in parallel by coordinating process and 2 worker processes.

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |SORT Operator
    | Using Worktable2 for internal storage.
    |
    |    |EXCHANGE Operator
    |    | Executed in parallel by 2 Producer and 1 Consumer processes.

    |    |
    |    |    |EXCHANGE:EMIT Operator
    |    |    |
    |    |    |    |HASH UNION Operator has 2 children.
    |    |    |    | Using Worktable1 for internal storage.
    |    |    |    |    |SCAN Operator
    |    |    |    |    |  FROM TABLE
```

```
|     |     |     |     |  sysindexes
|     |     |     |     |  Table Scan.
|     |     |     |     |  Forward Scan.
|     |     |     |     |  Positioning at start of table.
|     |     |     |     |  Using I/O Size 2 Kbytes for data pages.
|     |     |     |     |  With LRU Buffer Replacement Strategy for data
|     |     |     |     |    pages.
|     |     |
|     |     |     |     |SCAN Operator
|     |     |     |     |  FROM TABLE
|     |     |     |     |  sysindexes
|     |     |     |     |  Table Scan.
|     |     |     |     |  Forward Scan.
|     |     |     |     |  Positioning at start of table.
|     |     |     |     |  Using I/O size 2 Kbytes for data pages.
|     |     |     |     |  With LRU Buffer Replacement Strategy for data
|     |     |     |     |    pages.
```

merge union

The merge union operator performs an union all operation on several sorted compatible data streams and eliminates duplicates within these streams.

The merge union operator is a n-ary operator. It displays this message:

```
MERGE UNION OPERATOR has <N> children.
```

N is the number of input streams into the operator.

union all operator

The union all Operator merges several compatible input streams without performing any duplicate elimination. Every data row that enters the union all opertor will be included in the operator output stream.

The union all operator is a n-ary operator. It will display the following message:

```
UNION ALL OPERATOR  has N children.
```

N is the number of input streams into the operator.

Example

This example demonstrates the use of union all.

```
select * from sysindexes
union all
select * from sysindexes
```

```
QUERY PLAN FOR STATEMENT 1 (at line 4).
Executed in parallel by coordinating process and 3 worker processes.

6 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |SORT Operator
    |  Using Worktable1 for internal storage.
    |
    |    |UNION ALL Operator has 2 children.
    |    |
    |    |    |EXCHANGE Operator
    |    |    |Executed in parallel by 3 Producer and 1 Consumer processes.

    |    |    |
    |    |    |    |EXCHANGE:Emit Operator
    |    |    |    |
    |    |    |    |    |SCAN Operator
    |    |    |    |    |  FROM TABLE
    |    |    |    |    |  sysindexes
    |    |    |    |    |  Table Scan.
    |    |    |    |    |  Forward Scan.
    |    |    |    |    |  Positioning at start of table.
    |    |    |    |    |  Using I/O size 2 Kbytes for data pages.
    |    |    |    |    |  With LRU Buffer Replacement Strategy for data
    |    |    |    |    |    pages.
    |    |    |
    |    |    |SCAN Operator
    |    |    |  FROM TABLE
    |    |    |  sysindexes
    |    |    |  Table Scan.
    |    |    |  Forward Scan.
    |    |    |  Positioning at start of table.
    |    |    |  Using I/O size 2 Kbytes for data pages.
    |    |    |  With LRU Buffer Replacement Strategy for data pages.
```

scalaragg operator

The scalar aggregate operator keeps track of running information about an input data stream, such as for example the number of rows in the stream, or the maximum value of a given column in the stream.

The scalar aggregate operator will print a list of up to 10 messages describing the scalar aggregation operations it executes. The message has the following format:

```
Evaluate Ungrouped  <Type of the Aggragate> Aggregate
```

type of aggregate can be any of the following: count, sum, average,min, max, any, once-unique, count-unique, sum-unique, average-unique, or once.

The following query performs a scalar (i.e. Ungrouped) aggregate on the table authors in database pubs2:

```
1> use pubs2
2> go
1> set showplan on
2> go
1> select count(*) from authors
2> go

QUERY PLAN FOR STATEMENT 1 (at line 1).

2 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |SCALAR AGGREGATE Operator
    |  Evaluate Ungrouped COUNT AGGREGATE.
    |
    |    |SCAN Operator
    |    |  FROM TABLE
    |    |  authors
    |    |  Index : aunmind
    |    |  Forward Scan.
    |    |  Positioning at index start.
    |    |  Index contains all needed columns. Base table
will not be read.
    |    |  Using I/O Size 4 Kbytes for index leaf pages.
    |    |  With LRU Buffer Replacement Strategy for index
leaf pages.
```

```
-----------
    23

(1 row affected)
```

The message displayed in regard of the scalar aggregate operator indicates that the query to be executed is an ungrouped count aggregate.

restrict Operator

The restrict operator evaluates expressions based on column values. The restrict operator is associated with multiple column evaluations lists that can be processed before fetching a row from the child operator, after fetching a row from the child operator, or to compute the value of virtual columns after fetching a row from the child operator.

sort operator

The sort operator has only one child operator within the query plan. Its role is to generate an output data stream from the input stream, using a specified sorting key.

The sort operator may execute a streaming sort when possible, but may also have to store results temporarily into a work table. If it uses a work table, the sort operator will display its name in the following format:

```
Using Worktable <N>  for internal storage.
```

Where *N* is a numeric identifier for the worktable within the SHOWPLAN output.

Here is an example of a simple query plan using a sort operator and a work table:

```
1> use pubs2
2> go
1> set showplan on
2> go
1> select au_id from authors order by postalcode
2> go

QUERY PLAN FOR STATEMENT 1 (at line 1).
```

```
2 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

|SORT Operator
| Using Worktable1 for internal storage.
|
|   |SCAN Operator
|   |  FROM TABLE
|   |  authors
|   |  Table Scan.
|   |  Forward Scan.
|   |  Positioning at start of table.
|   |  Using I/O Size 4 Kbytes for data pages.
|   |  With LRU Buffer Replacement Strategy for data
pages.

au_id
----------------------
807-91-6654
527-72-3246
722-51-5454
712-45-1867
341-22-1782
899-46-2035
998-72-3567
172-32-1176
486-29-1786
427-17-2319
846-92-7186
672-71-3249
274-80-9391
724-08-9931
756-30-7391
724-80-9391
213-46-8915
238-95-7766
409-56-7008
267-41-2394
472-27-2349
893-72-1158
648-92-1872

(23 rows affected)
```

store operator

The store operator is used to create a work table, fill it and possibly create an index on it, as part of the execution of a query plan. The worktable will be used by other operators in the plan. A SEQUENCER operator will guarantee that the plan fragment corresponding to the work table and potential index creation will be executed prior to other plan fragments that make use of the work table. This is especially important when a plan is executed in parallel, as execution processes operate asynchronously.

In particular reformatting plans use this operator to create a work table and create an index on it.

If the store operator is used for a reformatting operation, it will print the following message:

```
Worktable <X> created, in <L> locking mode for
reformatting.
```

The locking mode *L* has to be one of “allpages”, “datapages,” “datarows.”

It will also print the following message:

```
Creating clustered index.
```

If the store operator is not used for a reformatting operation, it will print the following message:

```
Worktable <X> created, in <L> locking mode.
```

The locking mode *L* has to be one of "allpages", "datapages", "datarows."

The following example will be used for the store operator as well as for the sequencer operator in the next section of this document:

```
1> use master
2> go
1> set showplan on
2> go
1> select * from sysindexes S1, sysobjects S2
2> where S1.id = S2.id
QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the Abstract Plan in the PLAN clause.
Executed in parallel by coordinating process and 42
worker processes.

15 operator(s) under root

The type of query is SELECT.
```

```
ROOT:EMIT Operator

    |SEQUENCER Operator has 2 children.
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 20 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |STORE Operator
    |  |  |  |  Worktable1 created, in allpages locking
mode, for REFORMATTING.
    |   |   |   |  Creating clustered index.
    |   |   |   |
    |   |   |   |   |INSERT Operator
    |   |   |   |   |  The update mode is direct.
    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE Operator
    |  |  |  |  |  |Executed in parallel by 1 Producer
and 20 Consumer processes.

    |   |   |   |   |   |
    |   |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |  sysobjects
    |   |   |   |   |   |   |   |  S2
    |   |   |   |   |   |   |   |  Using Clustered Index.
    |   |   |   |   |   |   |   |  Index : csysobjects
    |   |   |   |   |   |   |   |  Forward Scan.
    |   |   |   |   |   |   |   |  Positioning at index
start.
    |   |   |   |   |   |   |   |  Using I/O Size 4 Kbytes
for index leaf pages.
    |   |   |   |   |   |   |   |  With LRU Buffer
Replacement Strategy for index leaf pages.
    |   |   |   |   |   |   |   |  Using I/O Size 4 Kbytes
for data pages.
    |   |   |   |   |   |   |   |  With LRU Buffer
Replacement Strategy for data pages.
    |   |   |   |   |
    |   |   |   |   |  TO TABLE
    |   |   |   |   |  Worktable1.
```

```
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 20 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |NESTED LOOP JOIN Operator (Join Type:
Inner Join)
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |Executed in parallel by 1 Producer
and 20 Consumer processes.

    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |  sysindexes
    |   |   |   |   |   |   |  S1
    |   |   |   |   |   |   |  Using Clustered Index.
    |   |   |   |   |   |   |  Index : csysindexes
    |   |   |   |   |   |   |  Forward Scan.
```

The store operator is highlighted for clarity in the above plan. In this plan, the store operator is located below the sequencer node, in the left child plan of the sequencer node. Its parent operator is an emit:exchange operator, and its child operator is an insert operator. It is located in a plan fragment below an exchange operator and will be executed in parallel by 20 worker processes, as indicated in the exchange operator.

The store operator will create a work table, that will be filled by the insert operator below it. The store operator wil then create a clustered index on the work table. The index will be built on the nested-loop join keys. The name of the worktable created by the store operator is Worktable1 in this case.

sequencer operator

The sequencer operator is a n-ary operator used to execute sequentially each of the child plans below it. It is used in particular in reformatting plans, and certain aggregate processing plans.

The sequencer operator will execute each of its child sub-plans except for the rightmost one. Once all the left child sub-plans are executed, it will execute the rightmost sub-plan.

The sequencer operator will display the following message:

```
SEQUENCER operator has N children.
```

Let's again take a look at the query plan from the section immediately above store operator:

```
ROOT:EMIT Operator

    |SEQUENCER Operator has 2 children.
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 20 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |STORE Operator
    |  |  |  |  Worktable1 created, in allpages locking
mode, for REFORMATTING.
    |   |   |   |  Creating clustered index.
    |   |   |   |
    |   |   |   |   |INSERT Operator
    |   |   |   |   |  The update mode is direct.
    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |   |Executed in parallel by 1
Producer and 20 Consumer processes.

    |   |   |   |   |   |
    |   |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |   |  sysobjects
    |   |   |   |   |   |   |   |  S2
    |   |   |   |   |   |   |   |  Using Clustered
Index.
    |   |   |   |   |   |   |   |  Index : csysobjects
    |   |   |   |   |   |   |   |  Forward Scan.
    |   |   |   |   |   |   |   |  Positioning at index
start.
    |  |  |  |  |  |  |  |  Using I/O Size 4 Kbytes
```

```
for index leaf pages.
    |   |   |   |   |   |   |   |  With LRU Buffer
Replacement Strategy for index leaf pages.
    |   |   |   |   |   |   |   |  Using I/O Size 4 Kbytes
for data pages.
    |   |   |   |   |   |   |   |  With LRU Buffer
Replacement Strategy for data pages.
    |   |   |   |   |
    |   |   |   |   |  TO TABLE
    |   |   |   |   |  Worktable1.
    |
    |   |EXCHANGE Operator
    |   |Executed in parallel by 20 Producer and 1
Consumer processes.

    |   |
    |   |   |EXCHANGE:EMIT Operator
    |   |   |
    |   |   |   |NESTED LOOP JOIN Operator (Join Type:
Inner Join)
    |   |   |   |
    |   |   |   |   |EXCHANGE Operator
    |   |   |   |   |Executed in parallel by 1 Producer
and 20 Consumer processes.

    |   |   |   |   |
    |   |   |   |   |   |EXCHANGE:EMIT Operator
    |   |   |   |   |   |
    |   |   |   |   |   |   |SCAN Operator
    |   |   |   |   |   |   |  FROM TABLE
    |   |   |   |   |   |   |  sysindexes
    |   |   |   |   |   |   |  S1
    |   |   |   |   |   |   |  Using Clustered Index.
    |   |   |   |   |   |   |  Index : csysindexes
    |   |   |   |   |   |   |  Forward Scan.
```

This time, we will highlight the sequencer operator within the plan. We can see that the sequencer operator has 2 child operators. The leftmost sub-plan will create the worktable used in reformatting, and the rightmost sub-plan will use this worktable to effect a nested-loop join with the system table sysindexes. Note that, in this example, both the table creation/index creation and the nested-loop join operations are done in parallel.

remscan operator

The remote scan operator ships a SQL query to a remote server for execution. It will then process the results returned by the remote server, if any. The remote scan operator will display the formatted text of the SQL query it handles.

The remote scan operator has 0 or 1 child operators.

scroll operator

The scroll operator encapsulates the functionality of scrollable cursors in ASE. Scrollable cursors may be insensitive, meaning that they will display a snapshot of their associated data, taken at open cursor time, or semi-sensitive, meaning that the next row(s) to be fetched will not be retrieved from a snapshot but from the live data.

The scroll operator is a unary operator.

The scroll operator will display the following message:

```
SCROLL OPERATOR ( Sensitive Type: <T>)
```

The type may be "Insensitive" or "Semi-Sensitive."

Following is an example of a plan featuring an insenistive scrollable cursor:

```
1> use pubs2
2> go
1> declare CI insensitive scroll cursor for
2> select au_lname, au_id from authors
3> go
1> set showplan on
2> go
1> open CI
2> go

QUERY PLAN FOR STATEMENT 1 (at line 1).

    STEP 1
       The type of query is OPEN CURSOR CI.


QUERY PLAN FOR STATEMENT 1 (at line 2).

2 operator(s) under root

The type of query is DECLARE CURSOR.
```

```
ROOT:EMIT Operator

    |SCROLL Operator (Sensitive Type: Insensitive)
    |     Using Worktable1 for internal storage.
    |
    |   |SCAN Operator
    |   |  FROM TABLE
    |   |  authors
    |   |  Table Scan.
    |   |  Forward Scan.
    |   |  Positioning at start of table.
    |   |  Using I/O Size 4 Kbytes for data pages.
    |   |  With LRU Buffer Replacement Strategy for data
pages.
```

We can see that the scroll operator is the child operator of the root emit operator, and its only child is the scan operator on the authors table. The scroll operator message specifies that cursor CI is insensitive.

ridjoin operator

The rid join operator effects a row-id based join of two data streams from the same source table. The rid join operator is a binary operator.

Each data row in a SQL table is associated with a unique row id or rid. A rid join will be used for a self-join query. The left child will fill a work table with the qualifying RIDs resulting from an index scan of the source table. Then this work table will be joined with the RIDs returned by the right child scanning another index on the same source table. The last step will be to retrieve the data rows associated with the resulting RIDs.

The RID JOIN operator will display the following message:

```
Using Worktable  <X>  for internal storage.
```

sqfilter operator

The sqfilter is used to execute subqueries. Its leftmost child represents the outer query, and the other children represent query plan fragments associated with one or more subqueries. The sqfilterop operator is a n-ary operator.

The leftmost child generates correlation values that will be substituted into the other child plans.

The SQFILTER operator will display the follwing message:

```
SQFILTER Operator has <N> children.
```

Example

This example illustrates the use of sqfilter.

```
select pub_name from publishers
where pub_id =
(select distinct titles.pub_id from titles
   where publishers.pub_id = titles.pub_id
   and price > $1000)
QUERY PLAN FOR STATEMENT 1 (at line 1).
4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

  |SQFILTER Operator has 2 children.
  |
  |  |SCAN Operator
  |  |  FROM TABLE
  |  |  publishers
  |  |  Table Scan.
  |  |  Forward Scan.
  |  |  Positioning at start of table.
  |  |  Using I/O Size 8 Kbytes for data pages.
  |  |  With LRU Buffer Replacement Strategy for data pages.
  |
  | Run subquery 1 (at nesting level 1)
  |
  |  QUERY PLAN FOR SUBQUERY 1 (at nesting level 1 and at line 3)
  |
  |  Correlated Subquery
  |  Subquery under an EXPRESSION predicate.
  |
  |  |SCALAR AGGREGATE Operator
  |  |  Evaluate Ungrouped ONCE-UNIQUE AGGREGATE
  |  |
  |  |  |SCAN Operator
  |  |  |  FROM TABLE
  |  |  |  titles
  |  |  |  Table Scan.
  |  |  |  Forward Scan.
  |  |  |  Postitioning at start of table.
  |  |  |  Using I/O Size 8 Kbytes for data pages.
  |  |  |  With LRU Buffer Replacement Strategy for data pages.
  |
```

```
| END OF QUERY PLAN FOR SUBQUERY 1
```

exchange operator

The exchange operator encapsulates parallel processing of SQL queries. It can be located almost anywhere in a query plan. It divides the plan into plan fragments, that is maximum subplans delimited by the root operator of the plan, exchange operators and leaf nodes, typically scan nodes. The exchange operator is a unary operator. The child operator of an exchange operator is an exchange:emit operator. The exchange:emit operator immediately below an exchange operator is the root operator for the plan fragment below the exchange operator. This plan fragment will be executed by the worker processes associated with the exchange operator.

The exchange operator manages the worker processes that execute in parallel the plan fragment located beneath the exchange operator. It also manages the exchange of data between processes.

The exchange operator is associated with a server process that acts as a local execution coordinator. This process is called the Beta process associated with the exchange operator. It can be a worker process as well as a process associated with a user connection.

The exchange operator will display the following message:

```
Executed in parallel by N producer and P consumer
processes.
```

The number of producer processes refers to the number of worker processes that execute the plan fragment located beneath the exchange operator, and the number of consumer processes refers to the number of worker processes that execute the plan fragment in which the exchange operator is included.

Let's look at the following simple example; a parallel query in database master against the system table sysmessages:

```
1> use master
2> go
1> set showplan on
2> go
1> select count(*) from sysmessages (parallel 4)
2> go

QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using the forced options (internally
```

```
generated Abstract Plan).
Executed in parallel by coordinating process and 4
worker processes.

4 operator(s) under root

The type of query is SELECT.

ROOT:EMIT Operator

    |SCALAR AGGREGATE Operator
    |  Evaluate Ungrouped COUNT AGGREGATE.
    |
    |    |EXCHANGE Operator
    |    |Executed in parallel by 4 Producer and 1
Consumer processes.

    |    |
    |    |    |EXCHANGE:EMIT Operator
    |    |    |
    |    |    |    |SCAN Operator
    |    |    |    |  FROM TABLE
    |    |    |    |  sysmessages
    |    |    |    |  Table Scan.
    |    |    |    |  Forward Scan.
    |    |    |    |  Positioning at start of table.
    |   |   |   |  Executed in parallel with a 4-way hash
scan.
    |    |   |   |  Using I/O Size 4 Kbytes for data pages.
    |    |   |   |  With LRU Buffer Replacement Strategy
for data pages.

-----------
7597


(1  row affected)
```

We can see that the root emit operator only child is the scalar aggregate operator that i sused to compute the count of the number of rows in the base table.

The only child of the scalar aggregate operator is the exchange operator.

This exchange operator has one consumer process, which is the process associated with the use connection, and four worker processes. Each of these worker processes will execute the same plan fragment in parallel. This plan fragment is made up of the exchange:emit operator and of the scan operator below it.

Data rows are propagated from the worker processes to the user process, also called the Beta process.

CHAPTER 4

Displaying Query Optimization Strategies And Estimates

This chapter describes the messages printed by the set commands designed for query optimization.

Topic	Page
Set commands for text format messages	157
Set commands for XML format messages	158
Usage scenarios	160
Permissions for Set commands	163
Discontinued tracing commands	163

Set commands for text format messages

These set commands generate diagnostics output in text format. They provide a convenient syntax that uses a single command, set option show, to show the level of each specific module

Table 4-1: Optimizer set command for text format messages

Command	Module
set option show <normal/brief/long/on/off>	Basic syntax common to all modules
set option show_lop <normal/brief/long/on/off>	Shows the logical operators (scans, joins, etc.) used
set option show_managers <normal/brief/long/on/off>	Shows data structure managers used during optimization.
set option show_log_props <normal/brief/long/on/off>	Shows the logical properties (row count, selectivity, etc.) evaluated.
set option show_parallel <normal/brief/long/on/off>	Shows details of parallel query optimization
set option show_histograms <normal/brief/long/on/off>	Shows the processing of histograms associated with SARG/Join columns
set option show_abstract_plan <normal/brief/long/on/off>	Shows the details of an abstract plan
set option show_search_engine <normal/brief/long/on/off>	Shows the details of the join ordering algorithm

Command	Module
set option show_counters <normal/brief/long/on/off>	Shows the optimization counters
set option show_best_plan <normal/brief/long/on/off>	Shows the details of the best query plan selected by the optimizer
set option show_pio_costing <normal/brief/long/on/off>	Shows estimates of physical input/output (reads/writes from/to the disk)
set option show_lio_costing <normal/brief/long/on/off>	Shows estimates of logical input/output (reads/writes from/to memory)
set option show_elimination <normal/brief/long/on/off>	Shows partition elimination
set option show_missing_stats <normal/brief/long/on/off>	Shows details of useful statistics missing from SARG/Join columns

Set commands for XML format messages

In Adaptive Server 15.0, diagnostics have been enhanced so that they can be sent out as an XML document. This makes it easier for front end tools to interpret the document. In some cases, users using the native XPath query processor inside Adaptive Server can query this output.

The diagnostics output can come from either the query optimizer or from the query execution layer. To generate an XML document for the diagnostic output, use this set plan command.

```
set plan for
    {show_exec_xml, show_opt_xml, show_execio_xml,
    show_lop_xml, show_managers_xml, show_log_props_xml,
    show_parallel_xml, show_histograms_xml,
    show_abstract_plan_xml, show_search_engine_xml,
    show_counters_xml, show_best_plan_xml, show_pio_costing_xml,
    show_lio_costing_xml, show_elimination_xml}
    to {client | message} on
```

The more interesting are the first three commands, though there are other low level options.

Command	Definition
show_exec_xml	Gets the compiled plan output in XML, showing each of the query plan operators.
show_execio_xml	Gets the plan output along with estimated and actual IOs. This also includes the query text.

Command	Definition
show_opt_xml	Gets optimizer diagnostic output, which shows all of the different components like logical operators, output from the managers, some of the search engine diagnostics, and the best query plan.
show_lop_xml	Gets the output logical operator tree in XML.
show_managers_xml	Shows the output of the different component managers during the preparation phase of the query optimizer.
show_log_props_xml	Shows the logical properties for a given equivalence class (one or more group of relations in the query).
show_parallel_xml	Shows the diagnostics related to the optimizer while generating parallel query plans.
show_histograms_xml	Shows diagnostics related to histograms and the merging of histograms.
show_abstract_plan_xml	Shows the AP generation/application.
show_search_engine_xml	Shows the search engine related diagnostics.
show_counters_xml	Shows plan object construction/destruction counters.
show_best_plan_xml	Shows the best plan in XML.
show_pio_costing_xml	Shows actual PIO costing in XML.
show_lio_costing_xml	Shows actual LIO costing in XML.
show_elimination_xml:	Shows partition elimination in XML.
client	When specified, output goes to the client.
message	When specified, output goes to an internal message buffer.

To turn an option off, specify:

```
set plan for
    {show_exec_xml, show_opt_xml, show_execio_xml, show_lop_xml,
    show_managers_xml, show_log_props_xml, show_parallel_xml,
    show_histograms_xml, show_abstract_plan_xml,
    show_search_engine_xml,
    show_counters_xml, show_best_plan_xml, show_pio_costing_xml,
    show_lio_costing_xml, show_elimination_xml} off
```

Note that you do not need to specify the destination stream when turning the option off.

When message is specified, the client application must get the diagnostics from the buffer using a built-in function called showplan_in_xml([query_num]).

Currently, no more than 20 queries are cached in the buffer; hence, the legal values to identify the query number are from 0 to 19. The cache stops collecting query plans when it reaches 20 queries; it ignores the rest of the query plans. However, the message buffer keeps collecting query plans. After 20, you can only get the whole of the message buffer, by using a value of 0.

A value of -1 refers to the last XML doc in the cache.

A value of 0 refers to the entire message buffer.

The message buffer may overflow. If this occurs, there is no way to log all of the XML doc, which could result in a partial and thereby invalid XML doc.

When accessed using showplan_in_xml, the message buffer is destroyed after execution.

You may want to set the maximum text size, as the XML document is printed as a text column and the document will be truncated if it is not large enough. Set the textsize to 100000 bytes using this command:

```
set textsize 100000
```

When set plan is issued with off, all XML tracing is turned off if all of the trace options have been turned off. Otherwise, only a given option or options are turned off. The rest is still valid and tracing continues on the specified destination stream. When you issue another set plan option, the previous option is unioned with the current option, but the destination stream will be switched unconditionally to a new one.

Usage scenarios

Scenario A

To get the XML plan for the execution plan to the client as trace output, use:

```
set plan for show_exec_xml  to client on
go
```

Then run the queries for which the plan is wanted:

```
select id from sysindexes where id < 0
```

You should see the XML doc here:

```
set plan for show_exec_xml off
```

Scenario B

To get the execution plan, use the showplan_in_xml built-in. You can get the output from the last query, or from any of the first 20 queries in a batch or stored procedure and nothing more.

```
set plan for show_opt_xml to  message on
```

Run the query as:

```
select id from sysindexes where id < 0
    select name from sysobjects where id > 0
go

select showplan_in_xml(0)
go
```

The example gets you two XML docs as text streams. You can run a Xpath query over this built-in as long as the XML option is enabled in Adaptive Server.

```
select xmlextract("/", showplan_in_xml(-1))
    go
```

This allows the xpath query "/" to be run over the XML doc produced by the last query.

Scenario C

To set multiple options on:

```
set plan for show_exec_xml, show_opt_xml to client on
    go

    select name from sysobjects where id > 0
    go
```

This sets up the output from the optimizer and the query execution engine to send the result to the client, as is done in normal tracing.

```
set plan for show_exec_xml off
    go

    select name from sysobjects where id > 0
    go
```

The optimizer's diagnostics are still available, as show_opt_xml is left on.

Scenario D

When running a set of queries in a batch, you can ask for the optimizer plan for the last query. This has been an issue in the past and is solved in the current paradigm.

```
    set plan for show_opt_xml to message on
    go
    declare @v int
```

```
select @v = 1
select name from sysobjects where id = @v
go

select showplan_in_xml(-1)
go
```

showplan_in_xml() can also be part of the same batch; it works the same way. Special care is taken to ignore logging any message for the showplan_in_xml() built-in.

It behaves in a very similar way for stored procedure. To create a procedure:

```
create proc PP as
declare @v int
select @v = 1
select name from sysobjects where id = @v
go

exec P
go

select showplan_in_xml(-1)
go
```

If the stored procedure calls another stored procedure, and the called stored procedure compiles, and optimizer diagnostics are turned on, you get the optimizer diagnostics for the new set of statements as well. The same is true if show_execio_xml is turned on and only the called stored procedure is executed.

Scenario E

To query the output of the showplan_in_xml() for the query execution plan, which is an XML doc:

```
set plan for show_exec_xml to message on
go

select name from sysobjects
go

select  case when
'/Emit/Scan[@Label="Scan:myobjectss"]' xmltest
showplan_in_xml(-1)
then "PASSED" else "FAILED"
go

set plan for show_exec_xml off
go
```

Permissions for Set commands

The System Administrator (SA) who has sa role has full access to the set commands described above.

For other users, however, a new set tracing permission must be granted and revoked by the System Administrator to allow set option and set plan for XML, as well as dbcc traceon/off (3604,3605), to work.

For more information, see the grant command description in the *Adaptive Server Reference Series: Commands*.

Discontinued tracing commands

Earlier versions of optimization tracing options (dbcc traceon/off(302,310,317)) are not supported anymore.

CHAPTER 5

Query Processing Metrics

Topic	Page
What are query processing metrics?	165
Executing QP metrics	166
Accessing metrics	166
Using metrics	166
Clearing the metrics	170

What are query processing metrics?

Query processing (QP) metrics identify and compare empirical metric values in query execution. When a query is executed, it is associated with a set of defined metrics that are the basis for comparison in QP metrics.

The metrics captured include:

- CPU execution time – the time, in milliseconds, it takes to execute the query.
- Elapsed time – the CPU time, in milliseconds, and the time it takes to parse, compile, and optimize the query. Elapsed time is recorded after the query plan is compiled.
- Logical IO – the number of logical IO reads.
- Physical IO – the number of physical IO reads.
- Count – the number of times a query is executed.
- Abort count – the number of times a query is aborted by the resource governor due to a resource limit being exceeded.

Each metric has three values: minimum, maximum, and average. Count and abort count are not included.

Executing QP metrics

You can activate and use QP metrics at the server level or at the session level.

At the server level, use sp_configure with the enable metrics capture option. The qpmetrics for ad hoc statements are captured directly into a system catalog, while the qpmetrics for statements in a stored procedure are saved in a procedure cache. When the stored procedure or query in the statement cache is flushed, the respective captured metrics are written to the system catalog.

```
sp_configure "enable metrics capture", 1
```

At a session level, use set metrics_capture on/off.

```
set metrics_capture on/off
```

Accessing metrics

QP metrics are always captured in the default running group, which is group 1 in each respective database. Use sp_metrics ‘backup’ to move saved QP metrics from the default running group to a backup group. Access metric information using a select statement with order by against the sysquerymetrics view.

You can also use a Data Manipulation Language (DML) statement to sort the metric information and identify the specific queries for evaluation.

Using metrics

Use the information produced by QP metrics to identify:

- Query performance regression
- Most “expensive” query from a batch of running queries
- Most frequently run queries

When you have information on the queries that may be causing problems, you can tune the queries to increase efficiency.

For example, identifying and fine-tuning an “expensive” query may be more effective than tuning the “cheaper” ones in the same batch.

You can also identify the queries that are run most frequently, and fine-tune them to increase efficiency.

Turning on query metrics may involve extra I/O for every query being run, so there may be perforrmance impact. However, the benefits mentioned above should be considered. Also, when the use of query metrics is contrasted with the information available in MDA tables, it is worth noting that with query metrics, aggregated historical data about a query can be gathered and stored in a system catalog. Information in MDA tables is transient.

Should I use QP metrics or monitoring tables?

Both QP metrics and monitoring tables have their place for gathering statistical information. However, you can use QP metrics instead of the monitoring tables to gather aggregated historical query information in a persistent catalog, rather than have transient information from the monitor tables.

sysquerymetrics view

Field	Definition
uid	User ID
gid	Group ID
id	Unique ID
hashkey	Hashkey over the SQL query text
sequence	Sequence number for a row when multiple rows are required for the text of the SQL
exec_min	Minimum execution time
exec_max	Maximum execution time
exec_avg	Average execution time
elap_min	Minimum elapsed time
elap_max	Maximum elapsed time
elap_avg	Average elapsed time
lio_min	Minimum logical IO
lio_max	Maximum logical IO
lio_avg	Average logical IO
pio_min	Minimum physical IO
pio_max	Maximum physical IO
pio_avg	Average physical IO

Field	Definition
cnt	Number of times the query has been executed.
abort_cnt	Number of times a query is aborted by the Resource Governor when a resource limit is exceeded
qtext	Query text

Average values in this view are calculated using this formula:

```
new_avg = (old_avg * old_count + new_value )/ (old_count + 1) = old_avg +
round((new_value - old_avg)/(old_count + 1))
```

This is an example of the sysquerymetrics view:

select * from sysquerymetrics

```
uid   gid   hashkey   id   sequence   exec_min
exec_max   exec_avg   elap_min   elap_max   elap_avg   lio_min
lio_max   lio_avg   pio_min   pio_max   pio_avg   cnt   abort_cnt
qtext
----------- ----------- ----------- ----------- ----------- -----------
----------- ----------- ----------- ----------- ----------- -----------
----------- ----------- ----------- ----------- ----------- -----------
------------------------------------------------------------------------
1   1   106588469   480001710   0   0
0   0   16   33   25   4
4   4   0   4   2   2   0
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
```

The above example displays a record for a SQL statement. The query text of the statement is select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2). This statement has been executed twice so far (cnt = 2). The minimum elapsed time is 16 milliseconds, the maximum elapsed time is 33 milliseconds, and the average elapsed time is 25 milliseconds. All the execution times are 0, and this may be due to the CPU execution time being less than 1 millisecond. The maximum physical I/O is 4, which is consistent with the maximum logical I/O. However, the minimum physical I/O is 0 because data is already in cache in the second run. The logical I/O equals 4, as LIO should be static whether or not the data is in memory.

Examples

You can use QP metrics to identify specific queries for tuning and possible regression on performance.

Identify the most expensive statement

Typically, to find the most expensive statement as the candidate for tuning, sysquerymetrics provides CPU execution time, elapsed time, logical IO, and physical IO as options for measure. For example, a typical measure is based on logical IO. Use the following query to find the statements that incur too many IOs as the candidates for tuning:

select lio_avg, qtext from sysquerymetrics order by lio_avg

```
lio_avg qtext
-----------
------------------------------------------------------
2
select c1, c2 from t_metrics1 where c1 = 333
4
select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
6
select count(t_metrics1.c1) from t_metrics1, t_metrics2,
t_metrics3 where (t_metrics1.c2 = t_metrics2.c2 and
t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3 = 0)
164
select min(c1) from t_metrics1 where c2 in (select t_metrics2.c2 from
t_metrics2, t_metrics3 where (t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3
= 1))

(4 rows affected)
```

The best candidate for tuning can be seen in the last statement in the above result which has the biggest value for average logical IO.

Identify the most frequently used statement for tuning

If a query is used frequently, fine-tuning may improve its performance. Identify the most frequently used query using the select statement with order by:

select elap_avg, cnt, qtext from sysquerymetrics order by cnt

```
elap_avg cnt
qtext
----------- -----------
--------------------------------------------------
0         1
select c1, c2 from t_metrics1 where c1 = 333
16        2

select distinct c1 from t_metrics1 where c2 in (select c2 from t_metrics2)
24        3
```

```
select min(c1) from t_metrics1 where c2 in (select t_metrics2.c2 from
t_metrics2, t_metrics3 where (t_metrics2.c2 = t_metrics3.c2 and t_metrics3.c3
= 1))

78        4

select count(t_metrics1.c1) from t_metrics1, t_metrics2, t_metrics3 where
(t_metrics1.c2 = t_metrics2.c2 and t_metrics2.c2 = t_metrics3.c2 and
t_metrics3.c3 = 0)

(4 rows affected)
```

Identify possible performance regression

In some cases, when a server is upgraded with a newer version, QP metrics may be useful for comparing performance. To identify queries that may have some degradation, use the following process:

1 Back up the QP metrics from the old server into a backup group:

    sp_metrics ‘backup’ <backup group ID>

2 Change to the new server and enable QP metrics:

    sp_configure “enable metrics capture”, 1

3 Compare QP metrics between the reports from the old and new servers to identify any queries that may have regression problems.

Clearing the metrics

Use sp_metrics ‘flush’ to flush all aggregated metrics in memory to the system catalog. The aggregated metrics for all statements in memory are zeroed out.

To remove QP metrics from the system catalog, use:

    sp_metrics ‘drop’, <*gid*>, <*id*>

CHAPTER 6

Abstract Plans

Topic	Page
New operators and syntax	172
New directives and syntax	175
Support for pre-15.0 operators	176
A complex query example	176
Semantics	177
Worktables and steps	177
Syntactic qualification	178
Legacy partial plans	179

Abstract plans are editable representations of a query plan created by the query processor. They can be captured, associated with the originating query, and reused whenever the originating query is run. They can also be written into a query using the plan clause in a select or other SQL statements.

Although the optimizer normally provides the most efficient query plans, sometimes a particular query may require, for example, a different join order or a different evaluation order of subqueries.

Abstract plans can be used to:

- Provide certain queries with a execution plan other than that provided by the optimizer
- Capture query plans before an upgrade to protect against any possible performance degradation caused by the upgrade

Abstract plans also provide a means to capture query plans before and after major system changes. The sets of before-and-after query plans can be compared to determine the effects of changes on your queries. Other uses include:

- Searching for specific types of plans, such as table scans or reformatting
- Searching for plans that use particular indexes

- Saving plans for queries with long optimization times

Abstract plans provide an alternative to options that must be specified in the batch or query in order to influence optimizer decisions. Using abstract plans, you can influence the optimization of a SQL statement without having to modify the statement syntax. Matching query text to stored text requires some processing overhead, but using a saved plan reduces query optimization overhead.

Adaptive Server 15.0 supports an improved approach to abstract plans. The structure of the abstract plan language has not changed; however, Adaptive Server 15.0 supports many new operators, and each operator now corresponds directly to a processing algorithm.

New operators and syntax

Table 6-1 describes the new abstract plan operators and their syntax. Some pre-15.0 operators are also still supported and are listed in "Support for pre-15.0 operators" on page 176. Each abstract plan operator corresponds to an operator used by the query engine. For example, h_join corresponds to the hash join operator, and hash_union_distinct corresponds to the hash-based N-ary union with duplicates elimination (hash distinct).

Note The nonderived table operators are largely unchanged.

For a complete description of the new operators, see Appendix A, "Abstract Plan Specifications."

Table 6-1: Derived table operators for abstract plans

Number of operands	Type of operator	Syntax	Description
Nullary – 0	Stored table scans	(scan *stored_table*)	Specifies a scan of a table or index.
		(t_scan *stored_table*)	Specifies a scan of a table.
		(i_scan *stored_index stored_table*)	Specifies an index scan of a table *stored_table* using index *stored_index*.
	Table literal scans	(*scan_values*)	Specifies the scan of literal values, such as select 1.

Number of operands	Type of operator	Syntax	Description
Unary – 1	Enforcers	(sort *derived_table*)	Sorts a *derived_table*. The sorting columns are automatically determined.
		(xchg degree *derived_table*)	Repartitions a *derived_table* so the number of streams is equal to the value degree.
		(store_index *derived_table*)	Indicates reformatting of a *derived_table*.
	Distinctness	(distinct *derived_table*)	Indicates distinctness enforcement on a *derived_table*.
		(distinct_sorted *derived_table*)	Enforces distinctness by removing duplicates without actually having to sort the data in *derived_table*. Constraint: The *derived_table* must be sorted on the columns where distinctness is required.
		(distinct_sorting *derived_table*)	Enforces distinctness by sorting on the columns where distinctness is required.
		(distinct_hashing *derived_table*)	Enforces distinctness by hashing on columns where distinctness is required.
	Grouping	(group *derived_table*)	Indicates a SQL grouping operation on a *derived_table*.
		(group_sorted *derived_table*)	Indicates grouping by performing aggregation without actually having to sort the data in *derived_table*. Assumes that data from the *derived_table* is sorted on the grouping columns.
		(group_hashing *derived_table*)	Indicates grouping by hashing on grouping columns and simultaneously performing aggregation.
Binary – 2	Joins	(join *derived_table1 derived_table2*)	Indicates SQL joining of data from *derived_table1* to data in *derived_table2*. The joining columns are automatically determined by the SQL query.
		(nl_join *derived_table1 derived_table2*)	Indicates SQL joining of data from *derived_table1* to data in *derived_table2*, using the algorithm for nested loop joins.

Number of operands	Type of operator	Syntax	Description
		(m_join *derived_table1 derived_table2*)	Indicates SQL joining of data from *derived_table1* to data in *derived_table2*, using the algorithm for merge joins. Constraint: *derived_table1* and *derived_table2* must be sorted on columns in equijoin predicates. Constraint: A merge join can be performed only when the two derived tables are connected by one or more equijoin predicates.
		(h_join *derived_table1 derived_table2*)	Indicates SQL joining of data from *derived_table1* to data in *derived_table2*, using the algorithm for hash joins. Constraint: A hash join can be performed only when the two derived tables are connected by one or more equijoin predicates.
	Nest subquery	(nested *derived_table subquery*)	Indicates that a subquery specified by its own abstract plan is evaluated during the derived table scan specified by *derived_table*.
N-ary – N	Unions	(union *derived_table1 derived_table2* ...)	Specifies the SQL union operation to be performed for derived tables specified as *derived_table1*, *derived_table2*, and so forth.
		(_union_distinct *derived_table1 derived_table2* ...)	Specifies the SQL union operation to be performed for derived tables specified as *derived_table1*, *derived_table2*, using the algorithm for merge. Constraint: Each derived table must be sorted on the columns in the select list of its underlying queries.
		(h_union_distinct *derived_table1 derived_table2* ...)	Specifies the SQL union operation to be performed for derived tables specified as *derived_table1*, *derived_table2*, using a hash-based algorithm.
		(m_union_all *derived_table1 derived_table2* ...)	Specifies the SQL union all operation to be performed for derived tables specified as *derived_table1*, *derived_table2*, using a merge algorithm that allows the ordering on at least one column in the select list of each underlying query to be preserved. Constraint: Each derived table must be sorted on the columns in the select list of its underlying queries.

Number of operands	Type of operator	Syntax	Description
		union_all *derived_table1 derived_table2* ...)	Specifies the SQL union all operation to be performed for derived tables specified as *derived_table1*, *derived_table2*, and so forth, by appending derived tables, one after another.
	Sequence	(sequence *derived_table1 derived_table2* ...derived_tableN)	Specifies that *derived_table1*, *derived_table2*, and so forth must all be processed before the last derived table, represented by *derived_tableN*, is accessed. Usually *derived_tableN* depends on a result set created by one of the derived tables *derived_table1*, *derived_table2*, and so forth.

New directives and syntax

Adaptive Server 15.0 supports query level directives for optimization goals and optimization timeout limit through the AbstractPlan mechanism. Such directives can be specified through the use keyword (see the Appendix, "Abstract Plan Specifications," for more information). The following examples highlight the use directive specification through Abstract Plans.

Optimization goal

This example highlights query level specification for the allrows_dss optimization goal.

```
select * from publishers p, titles t
where t.pub_id = p.pub_id
plan
"(use optgoals allrows_dss)"
```

Optimization timeout limit

This example highlights query level specification for optimization timeout limit setting.

```
select * from publishers p, titles t
where t.pub_id = p.pub_id
```

```
plan
"(use opttimeoutlimit 100)"
```

Support for pre-15.0 operators

Adaptive Server supports the following pre-15.0 operators as synonyms, but the optimizer no longer generates them:

- g_join and nl_g_join – replaced by join and nl_join.

  With version 15.0, the optimizer decides which join semantics to use. Adaptive Server does not provide abstract plan syntax that allows you to force inner, outer, or semijoins.

- plan – indicates a sequence of steps, usually connected through worktables or scalar results.

  With version 15.0, Adaptive Server uses the actual relational operators and hides the worktable. The server parses plan into the corresponding version 15.0 operators. When processing steps are required—for shared worktables and uncorrelated subqueries, for example—use sequence to avoid confusion.

A complex query example

```
select r1, sum(s1) from r, s
    where r2=s2
    group by r1
  union
  select t1, u2
    from t, u
    where t1=u1
  order by 1
plan
"(merge_union_all
    (group_sorted
        (nl_join
            (i_scan ir1 r)
            (i_scan is2 s)
        )
```

```
        )
        (m_join
            (i_scan it1 t)
            (i_scan iu1 u)
        )
    )"
```

This example forces a query plan that delivers a result set sorted on the first column by merge_union_all.

It relies on ordered operands. The union operand uses:

- group_sorted, which does on-the-fly vector aggregation based on the ordering delivered by its child, nl_join, which needs no ordering but preserves the ordering of its outer child, i_scan, which produces ordering based on the indexed columns.
- m_join, the merge join, which relies on its children being ordered on the equijoin clause columns. Both of its operands are i_scan on columns that produce the necessary ordering.

Semantics

Adaptive Server 15.0 checks the validity of each abstract plan, and rejects those that use algorithms incorrectly.

For example, Adaptive Server rejects this abstract plan because t_scan does not produce the ordering required by group_sorted:

```
select r1, sum(r2)
     from r
     group by r1
plan
"(group_sorted (t_scan r))"
```

Worktables and steps

Adaptive Server 15.0 does not expose worktables—worktables are an implementation detail of an operator, except in special circumstances.

In this example, a sorter worktable is unnecessary because the sort is implemented by distinct_sorting:

```
select *
     from
          r,
          (select distinct s1 from s) as d(d1)
          where r1=d1
plan
"(m_join
     (distinct_sorting
               (t_scan s)
          )
     (i_scan ir1 r)
)"
```

In this second example, worktables are necessary. The query plan uses two steps. The first step materializes the result of the distinct view as a worktable. The second step performs the self-join. Because the worktable is scanned twice and joined, it is not a detail of the algorithm.

```
create view v(v1, v2)
as select distinct s1, s2 from s

select * from v a, v b
where a.v1 = b.v2
plan
"(sequence
     (store
          (distinct_hashing
               (t_scan s)
     )
)
     nl_join
          (t_scan (table (w_table 1 a v)))
          (t_scan (table (w_table1 b v))))

)"
```

Syntactic qualification

In versions earlier than 15.0, syntactic qualification of tables names was required. In Adaptive Server 15.0, syntactic qualification of table names is necessary only when ambiguity would result otherwise.

In this example, qualification is unnecessary because there is no ambiguity between the s and t tables (qualification would have been required in earlier versions of Adaptive Server):

```
select * from t where t1 in (select s2 from s)
plan
"(nl_join
    (t_scan t)
    (t_scan s)
)"
```

In this example, qualification is necessary to distinguish between the two occurences of the t table that would otherwise be ambiguous.

```
select * from t where t1 in (select t2 from t)
plan
"(nl_join
    (t_scan t)
    (t_scan (table t (in (subq 1))))
)"
```

Legacy partial plans

Adaptive Server 15.0 accepts the syntax for empty hints, but no longer applies them.

In this example, m_join is applied as it specifies the abstract plan down to the leaves of the query tree. However, Adaptive Server ignores group and union as they are over the () empty hint that is ignored.

```
select r1, sum(s1)
     from r, s
     where r2 = s2
     group by r1
union
select t1, u2
     rom t, u
     where t1 = u1
order by 1
plan
"(merge_union_all
     (group_sorted
          ( )
     )
```

```
        (m_join
             (i_scan it1 t)
             (i_scan iu1 u)
        )
)"
```

CHAPTER 7

Using Statistics To Improve Performance

Accurate statistics are essential to query optimization. In some cases, adding statistics for columns that are not leading index keys also improves query performance. This chapter explains how and when to use the commands that manage statistics.

Topic	Page
Statistics maintained in Adaptive Server	181
Importance of statistics	182
Updating statistics	183
update statistics commands	184
Automatically updating statistics	187
Configuring automatic update statistics	190
Column statistics and statistics maintenance	193
Creating and updating column statistics	194
Choosing step numbers for histograms	196
Scan types, sort requirements, and locking	198
Using the delete statistics command	200
When row counts may be inaccurate	201

Statistics maintained in Adaptive Server

These key optimizer statistics are maintained in Adaptive Server Enterprise:

- Statistics per table: table row count; table page count. Can be found in systabstats.
- Statistics per index: index row count; index height; index leaf page count. Can be found in systabstats.
- Statistics per column: data distribution. Can be found in sysstatistics.

- Statistics per group of columns: density information. Can be found in sysstatistics.
- Statistics per partition
  - Partition data: partition data row count; partition data page count. Can be found in systabstats.
  - Partition index: partition index row count; partition index page count. Can be found in systabstats.
  - Column statistics: data distribution per column; density per group of columns. Can be found in sysstatistics.

Definitions

These definitions will help you to understand the material in this chapter.

density — Density is a statistical measurement of the uniqueness of a given column's values.

histogram — A histogram is a statistical representation of the distribution of values of a given column of the relation.

Importance of statistics

The Adaptive Server cost-based optimizer uses statistics about the tables, indexes, partitions, and columns named in a query to estimate query costs. It chooses the access method that the optimizer determines has the least cost. But this cost estimate cannot be accurate if statistics are not accurate.

Some statistics, such as the number of pages or rows in a table, are updated during query processing. Other statistics, such as the histograms on columns, are updated only when update statistics runs or when indexes are created.

If your query is performing slowly and you seek help from Technical Support or a Sybase newsgroup on the Internet, one of the first questions you are likely be asked is "Did you run update statistics?" You can use the optdiag command to see when update statistics was last run for each column on which statistics exist:

```
Last update of column statistics: Aug 31 2004
4:14:17:180PM
```

Another command you may need for statistics maintenance is delete statistics. Dropping an index does not drop the statistics for that index. If the distribution of keys in the columns changes after the index is dropped, but the statistics are still used for some queries, the outdated statistics can affect query plans.

Updating statistics

The update statistics command updates column-related statistics such as histograms and densities. Statistics must be updated on those columns where the distribution of keys in the index changes in ways that affect the use of indexes for your queries.

Running update statistics requires system resources. Like other maintenance tasks, it should be scheduled at times when the load on the server is light. In particular, update statistics requires table scans or leaf-level scans of indexes, may increase I/O contention, may use the CPU to perform sorts, and uses the data and procedure caches. Use of these resources can adversely affect queries running on the server if you run update statistics when usage is high. In addition, some update statistics commands require shared locks, which can block updates. See ""“Scan types, sort requirements, and locking” on page 198" for more information.

You can also configure Adaptive Server to automatically run update statistics at times that have minimal impact on the system resources. For more information, see “Automatically updating statistics” on page 187.

Adding statistics for unindexed columns

When you create an index, a histogram is generated for the leading column in the index. Examples in earlier chapters have shown how statistics for other columns can increase the accuracy of optimizer statistics.

You should consider adding statistics for virtually all columns that are frequently used as search arguments, as long as your maintenance schedule allows time to keep these statistics up to date.

In particular, adding statistics for minor columns of composite indexes can greatly improve cost estimates when those columns are used in search arguments or joins along with the leading index key.

update statistics commands

The update statistics commands create statistics if there are no statistics for a particular column, or replaces existing statistics if they already exist. The statistics are stored in the system tables systabstats and sysstatistics. The syntax is:

```
update statistics table_name
[[ partition data_partition_name ] [ (column_list ) ] |
index_name [ partition index_partition_name ] ]
[ using step values ]
[ with consumers = consumers] [, sampling=percent]

    update index statistics
table_name [[ partition data_partition_name ] |
[ index_name [ partition index_partition_name ] ] ]
[ using step values ]
[ with consumers = consumers] [, sampling=percent]

    update all statistics table_name
[ partition data_partition_name ]

    update table statistics
table_name [partition data_partition_name ]

    delete [ shared ] statistics table_name
[ partition data_partition_name ]
[( column_name[, column_name ] ...)]
```

The effects of the commands and their parameters are:

- For update statistics:
  - *table_name* – generates statistics for the leading column in each index on the table.
  - *table_name index_name* – generates statistics for all columns of the index.
  - *partition_name* – generates statistics for only this partition.
  - *partition_name table_name* (*column_name*) – generates statistics for this column of this table on this partition.
  - *table_name* (*column_name*) – generates statistics for only this column.

    - *table_name* (*column_name*, *column_name*...) – generates a histogram for the leading column in the set, and multi-column density values for the prefix subsets.
    - *using step values* – identifies the number of steps used. The default is 20 steps. If you need to change the default number of steps, use sp_configure.
    - sampling = *percent* – the numeric value of the sampling percentage, such as 05 for 5%, 10 for 10%, and so on. The sampling integer is between zero (0) and one hundred (100).
- For update index statistics:
    - *table_name* – Generates statistics for all columns in all indexes on the table.
    - *partition_name table_name* – Generates statistics for all columns in all indexes for the table on this partition.
    - *table_name index_name* – Generates statistics for all columns in this index.
- For update all statistics:
    - *table_name* – Generates statistics for all columns of a table.
    - *table_name partition_name* – Generates statistics for all columns of a table on a partition.
    - *using step values* – Identifies the number of steps used. The default is 20 steps. To change the default number of steps, use sp_configure. A new option in sp_configure is histogram tuning factor, which allows superior selection of the number of histogram steps. See the *System Administration Guide* for information about sp_configure.

Using sampling for *update statistics*

The optimizer for Adaptive Server uses the statistics on a database to set up and optimize queries. The statistics must be as current as possible to generate optimal results.

Run the update statistics commands against data sets, such as tables, to update information about the distribution of key values in specified indexes or columns, for all columns in an index, or for all columns in a table. The commands revise histograms and density values for column-level statistics. The results are then used by the optimizer to calculate the best way to set up a query plan.

update statistics requires table scans or leaf-level scans of indexes, may increase I/O contention, may use the CPU to perform sorts, and uses data and procedure caches. Use of these resources can adversely affect queries running on the server if you run update statistics when usage is high. In addition, some update statistics commands require shared locks, which can block updates.

To reduce I/O contention and resources, run update statistics using a sampling method, which can reduce the I/O and time when your maintenance window is small and the data set is large. If you are updating a large data set or table that is in constant use, being truncated and repopulated, you may want to do a statistical sampling to reduce the time and the size of the I/O. Because sampling does not update the density values, you should run a full update statistics prior to using sampling for an accurate density value.

You must use caution with sampling since the results are not fully accurate. Balance changes to histogram values against the savings in I/O.

Although a sampling of the data set may not be completely accurate, usually the histograms and density values are reasonable within an acceptable range.

When you are deciding whether or not to use sampling, consider the size of the data set, the time constraints you are working with, and if the histogram produced is as accurate as needed.

The percentage to use when sampling depends on your needs. Test various percentages until you receive a result that reflects the most accurate information on a particular data set.

Example:

```
update statistics authors(auth_id) with sampling = 5 percent
```

The server-wide sampling percent can be set using:

```
sp_configure 'sampling percent', 5
```

This command sets a server-wide sampling of 5% for update statistics that allows you to do the update statistics without the *sampling* syntax. The percentage can be between zero (0) and one hundred (100) percent.

Automatically updating statistics

The Adaptive Server cost-based query processor uses statistics for the tables, indexes, and columns named in a query to estimate query costs. Based on these statistics, the query processor chooses the access method it determines has the least cost. However, this cost estimate cannot be accurate if the statistics are not accurate. You can run update statistics to ensure that the statistics are current. However, running update statistics has an associated cost because it consumes system resources such as CPU, buffer pools, sort buffers, and procedure cache.

Instead of manually running update statistics at a certain time, you can set update statistics to run automatically at the time that best suits your site and avoid running it at times that hamper your system. The best time for you to run update statistics is based on the feedback from the datachange function. datachange also helps to ensure that you do not unnecessarily run update statistics. You can use these templates to determine the objects, schedules, priority, and datachange thresholds that trigger update statistics, which ensures that critical resources are used only when the query processor generates more efficient plans.

Because it is a resource intensive task, the decision to run update statistics should be based on a specific set of criteria. Some of the key parameters that can help you determine a good time to run update statistics are:

- How much have the data characteristics changed since you last ran update statistics? This is known as the datachange parameter.
- Are there sufficient resources available to run update statistics? These include resources such as the number of idle CPU cycles and making sure that critical online activity does not occur during update statistics.

Datachange is a key metric that helps you measure the amount of altered data since you last ran update statistics, and is tracked by the datachange function. Using this metric and the criteria for resource availability, you can automate the process of running update statistics. The Job Scheduler provides the mechanism to automatically run update statistics. Job Scheduler includes a set of customizable templates that determine when update statistics should be run. These inputs include all parameters to update statistics, the datachange threshold values, and the time to run update statistics. The Job Scheduler runs update statistics at a low priority so it does not affect critical jobs that are running concurrently.

What is the *datachange* function?

The datachange function measures the amount of change in the data distribution since update statistics last ran. Specifically, it measures the number of inserts, updates, and deletes that have occurred on the given object, partition, or column, and helps you determine if running update statistics would benefit the query plan.

The syntax for datachange is:

select datachange(*object_name*, *partition_name*, *colname*)

Where:

- *object_name* – is the object name. This object is assumed to be in the current database. This is a required parameter. It cannot be null.
- *partition_name* – is the data partition name. This can also be a null value.
- *colname* – is the column name for which the datachange is requested. This can also be a null value.

The datachange function requires all three parameters.

datachange is expressed as a percentage of the total number of rows in the table or partition (if the partition is specified). The percentage value can be greater than 100 percent because the number of changes to an object can be much greater than the number of rows in the table, particularly when the number of deletes and updates happening to a table is very high.

The following set of examples illustrate the various uses for the datachange function. The examples use the following:

- Object name is "O."
- Partition name is "P."
- Column name is "C."

Passing a valid object, partition, and column name

The value reported when you include the object, partition, and column name is determined by this equation: the datachange value for the specified column in the specified partition divided by the partitions's rowcount. The result is expressed as a percentage:

```
datachange = 100 * (data change value for column C/ rowcount (P))
```

Using null partition names

If you include a null partition name, the datachange value is determined by this equation: the sum of the datachange value for the column across all partitions divided by the rowcount of the table. The result is expressed as a percentage:

```
datachange = 100 * (Sum(data change value for (O, P(1-N) , C))/rowcount(O)
```

Where P(1-*N*) indicates that the value is summed over all partitions.

Using null column names

If you include null column names, the value reported by datachange is determined by this equation: the maximum value of the datachanges for all columns that have histograms for the specified partition divided by the number of rows in the partition. The result is expressed as a percentage:

```
datachange = 100 * (Max(data change value for (O, P, Ci))/rowcount(P)
```

Where the value of *i* varies through the columns with histograms (for example, formatid 102 in sysstatistics).

Null partition and column names

If you include null partition and column names, the value of datachange is determined by this equation: the maximum value of the datachange for all columns that have histograms summed across all partitions divided by the number of rows in the table. The result is expressed as a percentage:

```
datachange = 100 * ( Max(data change value for (O, NULL, Ci))/rowcount(O)
```

Where *i* is 1 through the total number of columns with histograms (for example, formatid 102 in sysstatistics)

The following session illustrates datachange gathering statistics:

```
create table matrix(col1 int, col2 int)
go
insert into matrix values (234, 560)
go
update statistics matrix(col1)
go
insert into matrix values(34,56)
go
select datachange ("matrix", NULL, NULL)
go

------
50.000000
```

The number of rows in matrix is two. The amount of data that has changed since the last update statistics command is 1, so the datachange percentage is 100 * 1/2 = 50 percent

datachange counters are all maintained “in-memory.” These counters periodically get flushed to disk by the housekeeper or when you run sp_flushstats.

Configuring automatic *update statistics*

There are three methods for automatically updating statistics:

- Defining update statistics jobs with the Job Scheduler.
- Defining update statistics jobs as part of the self-management installation.
- Creating user-defined scripts.

The creation of user-defined scripts is not discussed in this document.

Using Job Scheduler to update statistics

The Job Scheduler includes the Update Statistics template, which you can use to create a job that runs update statistics on a table, index, column, or partition. The datachange function determines when the amount of change in a table or partition has reached the predefined threshold. You determine the value for this threshold when you configure the template.

Templates perform the following operations:

- Run update statistics on specific tables, partitions, indexes, or columns. The templates allow you to define the value for datachange that you want update statistics to run.
- Run update statistics at the server level, which configures Adaptive Server to sweep through the available tables in all databases on the server and update statistics on all the tables, based on the threshold you determined when creating your job.

Use the following steps to configure the Job Scheduler to automate the process of running update statistics (the chapters listed below are from the *Job Scheduler User's Guide*:

1 Install and set up the Job Scheduler (described in Chapter 2 "Configuring and Running Job Scheduler."

2 Install the stored procedures required for the templates (described in Chapter 4, "Using Templates to Schedule Jobs.")

3 Install the templates. Job Scheduler provides the templates specifically for automating update statistics (described in Chapter 4, "Using Templates to Schedule Jobs").

4 Configure the templates. The templates for automating update statistics are listed under the Statistics Management folder.

5 Schedule the job. After you have defined which index, column, or partition you want tracked, you can also create a schedule that determines when Adaptive Server runs the job, making sure that update statistics is run only when it does not impact performance.

6 Identify success or failure. The Job Scheduler infrastructure allows you to identify success or failure for the automated update statistic.

The template allows you to supply values for the various options of the update statistics command such as sampling percent, number of consumers, steps, and so on. Optionally, you can also provide threshold values for the datachange function, page count, and row count. If you include these optional values, they are used to determine when and if Adaptive Server should run update statistics. If the current values for any of the tables, columns, indexes, or partitions exceed the threshold values, Adaptive Server issues update statistics. After Adaptive Server runs update statistics, it runs sp_recompile for the table specified in the template.

When does Adaptive Server run *update statistics*?

There are many forms of the update statistics command (update statistics, update index statistics, and so on) and you can form the command in many ways depending on your needs.

You must specify three thresholds: rowcount, pagecount, and datachange. All the thresholds must be satisfied for update statistics to run. Although values of NULL or 0 are ignored, these values do not prevent the command from running.

Table 7-1 describes the circumstances under which Adaptive Server automatically runs update statistics, based on the parameter values you provide.

Table 7-1: When does Adaptive Server automatically run update statistics?

If the user	Action taken by Job Scheduler
Specifies a datachange threshold of zero or NULL	Runs update statistics at the scheduled time.
Specifies a datachange threshold greater than zero for a table only, and does not request the update index statistics form	Gets all the indexes for the table and gets the leading column for each index. If the datachange for any leading column is greater than or equal to the threshold, run update statistics.
Specifies threshold values for the table and index but does not request the update index statistics form	Gets the datachange value for the leading column of the index. If the datachange value is greater than or equal to the threshold, runs update statistics.
Specifies a threshold value for a table only, and requests the update index statistics form	Gets all the indexes for the table and gets the leading column for each index. If the datachange for any leading column exceeds the threshold, runs update statistics.
Specifies threshold values for table and index and requests the update index statistics form	Gets the datachange value for the leading column of the index. If the datachange value is greater than or equal to the threshold, runs update statistics.

If the user	Action taken by Job Scheduler
Specifies threshold values for a table and one or more columns (ignores any indexes or requests for the update index statistics form)	Gets the datachange value for each column. If the datachange for any column is greater than or equal to the threshold, runs update statistics.

The datachange function compiles the number of changes in a table and displays this as a percentage of the total number of rows in the table. You can use this compiled information to create rules that determine when Adaptive Server runs update statistics. The best time for this to happen can be based on any number of objectives:

- The percentage of change in a table.
- Number of CPU cycles available.
- During a maintenance window.

After update statistics runs, the datachange counter is reset to zero. The count for datachange is tracked at the partition level (not the object level) for inserts and deletes and at the column level for updates.

Examples of updating statistics with *datachange*

You can write scripts that check for the specified amount of changed data at the column, table, or partition level. When you decide to run update statistics can be based on a number of variables collected by the datachange function; CPU usage, percent change in a table, percent change in a partition, and so on.

Running update statistics based on datachange in a partition

In this example, the authors table is partitioned, and the user wants to run update statistics when the data changes to the city column in the author_ptn2 partition are greater than or equal to 50 percent:

```
select  @datachange = datachange("authors","author_ptn2", "city")
if @datachange >= 50
begin
         update statistics authors partition author_ptn2(city)
end
go
```

The user can also specify that the script is executed when the system is idle or any other parameters they see fit.

Running update statistics based on datachange in a column

In this example, the user triggers update statistics when the data changes to the city column of the authors table are greater than or equal to 100 percent (the table in this example is not partitioned):

```
select  @datachange = datachange("authors",NULL, "city")
if @datachange > 100
begin
         update statistics authors (city)
end
go
```

Column statistics and statistics maintenance

Histograms are kept on a per-column basis, rather than on a per-index basis. This has certain implications for managing statistics:

- If a column appears in more than one index, update statistics, update index statistics, or create index updates the histogram for the column and the density statistics for all prefix subsets.

  update all statistics updates histograms for all columns in a table.

- Dropping an index does not drop the statistics for the index, since the optimizer can use column-level statistics to estimate costs, even when no index exists.

  To remove the statistics after dropping an index, you must explicitly delete them using delete statistics.

  If the statistics are useful to the query processor and to keep the statistics without having an index, use update statistics, specifying the column name, for indexes where the distribution of key values changes over time.

- Truncating a table does not delete the column-level statistics in sysstatistics. In many cases, tables are truncated and the same data is reloaded.

  Since truncate table does not delete the column-level statistics, you need not run update statistics after the table is reloaded, if the data is the same.

  If you reload the table with data that has a different distribution of key values, run update statistics.

- You can drop and re-create indexes without affecting the index statistics, by specifying “0” for the number of steps in the with statistics clause to create index. This create index command does not affect the statistics in sysstatistics:

  ```
  create index title_id_ix on titles(title_id)
  ```

```
with statistics using 0 values
```

This allows you to re-create an index without overwriting statistics that have been edited with optdiag.

- If two users attempt to create an index on the same table, with the same columns, at the same time, one of the commands may fail due to an attempt to enter a duplicate key value in sysstatistics.

Creating and updating column statistics

Creating statistics on unindexed columns can improve the performance of many queries. The optimizer can use statistics on any column in a where or having clause to help estimate the number of rows from a table that match the complete set of query clauses on that table.

Adding statistics for the minor columns of indexes and for unindexed columns that are frequently used in search arguments can greatly improve the optimizer's estimates.

Maintaining a large number of indexes during data modification can be expensive. Every index for a table must be updated for each insert and delete to the table, and updates can affect one or more indexes.

Generating statistics for a column without creating an index gives the optimizer more information to use for estimating the number of pages to be read by a query, without the processing expense of index updates during data modification.

The optimizer can apply statistics for any columns used in a search argument of a where or having clause and for any column named in a join clause.

Use these commands to create and maintain statistics:

- update statistics, when used with the name of a column, generates statistics for that column without creating an index on it.

  The optimizer can use these column statistics to more precisely estimate the cost of queries that reference the column.

- update index statistics, when used with an index name, creates or updates statistics for all columns in an index.

  If used with a table name, it updates statistics for all indexed columns.

- update all statistics creates or updates statistics for all columns in a table.

Good candidates for column statistics are:

- Columns frequently used as search arguments in where and having clauses
- Columns included in a composite index, and which are not the leading columns in the index, but which can help estimate the number of data rows that need to be returned by a query.

When additional statistics may be useful

To determine when additional statistics are useful, run queries using set options commands and set statistics io on. If there are significant discrepancies between the "rows to be returned" and I/O estimates displayed by set options commands and the actual I/O displayed by statistics io, examine these queries for places where additional statistics can improve the estimates. Look especially for the use of default density values for search arguments and join columns.

Also, note that the set option show_missing_stats command prints the names of columns that could have used histograms, and groups of columns that could have used multi-attribute densities. This is particularly useful in pointing out where additional statistics can be useful.

Adding statistics for a column with *update statistics*

This command adds statistics for the price column in the titles table:

```
update statistics titles (price)
```

This command specifies the number of histogram steps for a column:

```
update statistics titles (price)
   using 50 values
```

This command adds a histogram for the titles.pub_id column and generates density values for the prefix subsets pub_id; pub_id, pubdate; and pub_id, pubdate, title_id:

```
update statistics titles(pub_id, pubdate, title_id)
```

Note Running update statistics with a table name updates histograms and densities for leading columns for indexes only; it does not update the statistics for unindexed columns. To maintain these statistics, run update statistics and specify the column name, or run update all statistics

Adding statistics for minor columns with *update index statistics*

To create or update statistics on all columns in an index, use update index statistics. The syntax is:

update index statistics
table_name [[partition *data_partition_name*] |
[*index_name* [partition *index_partition_name*]]]
[using step values]
[with consumers = consumers] [, sampling = *percent*]

Adding statistics for all columns with *update all statistics*

To create or update statistics on all columns in a table, use update all statistics. The syntax is:

update all statistics *table_name*
[partition *data_partition_name*]

Choosing step numbers for histograms

By default, each histogram has 20 steps, which provides good performance and modeling for columns that have an even distribution of values. A higher number of steps can increase the accuracy of I/O estimates for:

- Columns with a large number of highly duplicated values
- Columns with unequal or skewed distribution of values

- Columns that are queried using leading wildcards in like queries

Note If your database was updated from a pre-11.9 version of the server, the number of steps defaults to the number of steps that were used on the distribution page.

Disadvantages of too many steps

Increasing the number of steps beyond what is needed for good query optimization can hurt Adaptive Server performance, largely due to the amount of space that is required to store and use the statistics. Increasing the number of steps:

- Increases the disk storage space required for sysstatistics
- Increases the cache space needed to read statistics during query optimization
- Requires more I/O, if the number of steps is very large

During query optimization, histograms use space borrowed from the procedure cache. This space is released as soon as the query is optimized.

Choosing a step number

For example, if your table has 5000 rows, and one value in the column that has only one matching row, you may need to request 5000 steps to get a histogram that includes a frequency cell for every distinct value. The actual number of steps is not 5000; it is either the number of distinct values plus one (for dense frequency cells) or twice the number of values plus one (for sparse frequency cells).

Another point to note is that the sp_configure option histogram tuning factor automatically chooses a larger number of steps, within parameters, when there are a large number of highly duplicated values.

Scan types, sort requirements, and locking

Table 7-2 shows the types of scans performed during update statistics, the types of locks acquired, and when sorts are needed.

Table 7-2: Scans, sorts, and locking during update statistics

update statistics specifying	Scans and sorts performed	Locking
Table name		
Allpages-locked table	Table scan, plus a leaf-level scan of each nonclustered index	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Table scan, plus a leaf-level scan of each nonclustered index and the clustered index, if one exists	Level 0; dirty reads
Table name and clustered index name		
Allpages-locked table	Table scan	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Leaf level index scan	Level 0; dirty reads
Table name and nonclustered index name		
Allpages-locked table	Leaf level index scan	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Leaf level index scan	Level 0; dirty reads
Table name and column name		
Allpages-locked table	Table scan; creates a worktable and sorts the worktable	Level 1; shared intent table lock, shared lock on current page
Data-only-locked table	Table scan; creates a worktable and sorts the worktable	Level 0; dirty reads

Sorts for unindexed or non leading columns

For unindexed columns and columns that are not the leading columns in indexes, Adaptive Server performs a serial table scan, copying the column values into a worktable, and then sorts the worktable to build the histogram. The sort is performed in serial, unless the with consumers clause is specified.

See Chapter 9, “Parallel Sorting” in *Performance and Tuning: Optimizer and Abstract Plans* for information on parallel sort configuration requirements.

Locking, scans, and sorts during *update index statistics*

The update index statistics command generates a series of update statistics operations that use the same locking, scanning, and sorting as the equivalent index-level and column-level command. For example, if the salesdetail table has a nonclustered index named sales_det_ix on salesdetail(stor_id, ord_num, title_id), this command:

```
update index statistics salesdetail
```

performs these update statistics operations:

```
update statistics salesdetail sales_det_ix
update statistics salesdetail (ord_num)
update statistics salesdetail (title_id)
```

Locking, scans and sorts during *update all statistics*

The update all statistics commands generate a series of update statistics operations for each index on the table, followed by a series of update statistics operations for all unindexed columns.

Using the *with consumers* clause

The with consumers clause for update statistics is designed for use on partitioned tables on RAID devices, which appear to Adaptive Server as a single I/O device, but are able to produce the high throughput required for parallel sorting. See Chapter 9, “Parallel Sorting” in *Performance and Tuning: Optimizer and Abstract Plans* for more information.

Reducing *update statistics* impact on concurrent processes

Since update statistics uses dirty reads (transaction isolation level 0) for data-only-locked tables, you can execute it while other tasks are active on the server; it does not block access to tables and indexes. Updating statistics for leading columns in indexes requires only a leaf-level scan of the index, and does not require a sort, so updating statistics for these columns does not affect concurrent performance very much.

However, updating statistics for unindexed and non-leading columns, which require a table scan, worktable, and sort can affect concurrent processing.

- Sorts are CPU-intensive. Use a serial sort, or a small number of worker processes to minimize CPU utilization. Alternatively, you can use execution classes to set the priority for update statistics.

  See "Using Engines and CPUs" in *Performance and Tuning: Basics*.

- The cache space required for merging sort runs is taken from the data cache, and some procedure cache space is also required. Setting the number of sort buffers to a low value reduces the space used in the buffer cache.

  If number of sort buffers is set to a large value, it takes more space from the data cache, and may also cause stored procedures to be flushed from the procedure cache, since procedure cache space is used while merging sorted values.

Creating the worktables for sorts also uses space in tempdb.

Using the *delete statistics* command

In pre-11.9 versions of SQL Server and Adaptive Server, dropping an index removed the distribution page for the index. Since version 11.9.2, maintaining column-level statistics is under explicit user control, and the optimizer can use column-level statistics even when an index does not exist. The delete statistics command allows you to drop statistics for specific columns.

If you create an index and then decide to drop it because it is not useful for data access, or because of the cost of index maintenance during data modifications, you must determine:

- Whether the statistics on the index are useful to the optimizer.
- Whether the distribution of key values in the columns for this index are subject to change over time as rows are inserted and deleted.

  If the distribution of key values changes, run update statistics periodically to maintain useful statistics.

This example deletes the statistics for the price column in the titles table:

```
delete statistics titles(price)
```

Note delete statistics, when used with a table name, removes all statistics for a table, even where indexes exist.

You must run update statistics on the table to restore the statistics for the index.

When row counts may be inaccurate

Row count values for the number of rows, number of forwarded rows, and number of deleted rows may be inaccurate, especially if query processing includes many rollback commands. If workloads are extremely heavy, and the housekeeper wash task does not run often, these statistics are more likely to be inaccurate.

Running update statistics corrects these counts in systabstats.

Running dbcc checktable or dbcc checkdb updates these values in memory.

When the housekeeper wash task runs, or when you execute sp_flushstats, these values are saved in systabstats.

Note The configuration parameter housekeeper free write percent must be set to 1 or greater to enable housekeeper statistics flushing.

APPENDIX A

Abstract Plan Specifications

These operators have been added for Abstract Plans in Adaptive Server 15.0.

delete

Description — Specifies the placement of the delete operator over the child *derived_table*.

Syntax — (delete *derived_table*)

Parameters — *derived_table*

The child derived table that qualifies the rows to be deleted.

Examples — Qualifies the row for deletion by an index scan.

```
delete t where t1>0
plan
     "( delete
          ( i_scan it1 t )
     )"
```

Usage

- Returns the derived table corresponding to the delete result.
- The query must be part of a delete statement.
- delete is useful only for plans that do not have the delete Lava operator at the root. For example, you can get the same result in Example 1 with the partial abstract plan `"(i_scan it1 t)"`.
- In general, a delete SQL statement does not return a useful derived table. Its outcome is instead the changes made to the result table. However, because the abstract plan operators tree must match the query execution plan operators tree, use delete whenever you must use an abstract plan that also describes the parent of the delete operation in the query execution plan.

See also

- **Commands** insert, update

distinct

Description

Specifies the placement of one of the distinct operators over the child *derived_table*. distinct enforces duplication elimination according to the semantics of the query.

Syntax

(distinct *derived_table*)

Parameters

derived_table

The child derived table that gets duplicate elimination.

Examples

Example 1 Forces the existence subquery, duplicates semantics by obtaining distinct correlation values, and lets the query processor select the best physical operators for the distinct, the scans, and the joins.

```
select * from t
     where t1 in
          (select r1 from r, s where r2=s2)
plan
     "( join
          (distinct
               (join
                    (scan r)
                    (scan s)
               )
          )
          (scan t)
)"
```

Example 2 Performs the join of tables r, s, and t before evaluating for distinctness. This is an example of "late evaluation" for distinctness, which is beneficial in some circumstances. This query lets the query processor select the best cost-based physical operators for the distinct, the scans, and the join.

```
create view dv(dv1, dv2)
as
     select distinct r1, s1
          from r, s
          where r2=s2

select * from t, dv where t1=dv1
plan

     "(distinct
          (join
               (join
                    (scan t)
                    (scan r)
               )
```

```
                    (scan s)
                )
)"
```

Usage

- The value returned indicates the distinct enforced derived table.
- If a query enforces the elimination of duplicate operators and also includes joins, the query processor may be able to evaluate the distinctness early enough so that it can reduce the size of the result set that is joined. However, if a join reduces the result set, it is probably better to perform the distinct evaluation late in the query evaluation. Whether the query is evaluated early or late for distinctness can be particularly significant for queries involving views with distincts, in, or exists type subqueries.
- Distinctness enforcement must be both needed and possible over the child *derived_table*.
- The query must require that you eliminate duplicates through a `select distinct` or an `exists/in` subquery.
- The distinct operator removes duplicates according to a distinct key. The query processor computes the distinct key according to the position of the distinct operator and the semantics of the query.
- The query processor checks whether distinctness is possible and needed. It applies the abstract plan distinct operator only when it is legal. Otherwise, this operator and all parents up to the root are ineffective.
- There is no distinct query execution plan operator. The distinct abstract plan operator lets the query processor evaluate the cost among the available distinct implementations: DistinctHashing, DistinctSorted, and DistinctSorting.
- When distinct enforcement is needed, a distinct operator is not always mandatory. In some cases, the query processor can enforce distinctness using a semijoin.
- The distinct abstract plan operator does not imply the use of a worktable. Some distinct algorithms (for example, DistinctSorted) do not use one. For algorithms that do, the work table is hidden in the operator. This is a change from earlier releases, when a work table and two processing steps were always used by the query execution plan.
- In earlier versions of Adaptive Server, the worktable and the two processing steps were always exposed by the abstract plan. In earlier versions of Adaptive Server, the abstract plan in Example 2 would look like this:

```
(plan
     (store (work_t Worktable1)
          (nl_join
               (t_scan r)
               (i_scan is2 s)
          )
          (nl_join
               (t_scan (work_t Worktable1))
               (i_scan it1 t)
          )
)
```

- The equivalent Adaptive Server 15.0 abstract plan is obtained by replacing the scan of the worktable with the distinct abstract plan operator over the child of the store.

See also

- **Commands** distinct_hashing, distinct_sorted, distinct_sorting

distinct_hashing

Description

Specifies the placement of the DistinctHashing operator over the child *derived_table*. Enforces duplicates elimination according to the semantics of the query.

Syntax

(distinct_hashing *derived_table*)

Parameters

derived_table

The child derived table that gets duplicate elimination.

Examples

Example 1 Forces the existence subquery, duplicates, and semantics to be enforced by obtaining hashing distinct correlation values. It lets the query processor select the best cost-based physical operators for the scans and the joins.

```
select * from t
where t1 in
     (select r1 from r, s where r2=s2)
plan
     "( join
          (distinct_hashing
               (join
                    (scan r)
                    (scan s)
               )
     )
          (scan t)
)"
```

Example 2 Performs a join on tables r, s and t, then enforces distinctness using the hash-based distinct operation and lets the query processor select the best cost-based physical operators for the scan and the join. This plan generalizes the strategy in earlier Adaptive Server releases where existence joins were converted into regular inner joins followed by duplicate elimination.

```
create view dv(dv1, dv2)
as
     select distinct r1, s1
          from r, s
     where r2=s2

select * from t, dv where t1=dv1
plan
     "(distinct_hashing
          (join
               (join
                    (scan t)
```

```
                                (scan r)
                            )
                            (scan s)
                    )
    )"
```

Usage

- The value returned represents the distinctness enforced derived table.
- Distinctness enforcement must be both required and possible over the child *derived_table*.
- The hash-based enforcement of distinctness is less expensive in terms of CPU and I/O cost than a sort-based distinct evaluation.
- distinct_hashing has the advantage of having no order-related precondition.
- The abstract plan distinctness enforcement is described in more detail under the distinct abstract plan operator.

See also

- **Commands** distinct_sorted, distinct_sorting

distinct_sorted

Description

Specifies the placement of the DistinctSorted operator over the child *derived_table*. Enforces duplicates elimination according to the semantics of the query.

Syntax

(distinct_sorted derived_table)

Parameters

derived_table

The child derived table whose duplicates are eliminated.

Examples

Example 1 Enforces the existence subquery duplicates semantics by obtaining distinct correlation values; duplicates are dropped on the fly. This example lets the query processor select, according to a cost-based evaluation, the best physical operators for some of the scans and joins. However, distinct key ordering is enforced by forcing the index scan and the nested-loop join. This is a generalization of earlier versions of Adaptive Server row-filtering strategy.

```
select * from t
where t1 in
     (select r1 from r, s where r2=s2)
plan
     "( join
          (distinct_sorted
               (nl_join
                    (i_scan ir1 r)
                    (scan s)
               )
          )
          (scan t)
)"
```

Example 2 This abstract plan eliminates duplicates in the distinct view projection on the fly. Its application succeeds if there is at least an r-s join subplan that provides an ordering on (r1,r2).

```
create view dv(dv1, dv2)
as
     select distinct r1, r2
          from r, s
     where r2=s2

select * from t, dv where t1=dv1
plan
     "( join
          (distinct_sorted
               (join
                    (scan r)
```

```
                        (scan s)
                    )
                )
                (scan t)
)"
```

Usage

- The returned value is a derived table with distinct values.
- Distinctness enforcement must be both needed and possible over the child *derived_table*.
- The child *derived_table* is ordered by the distinct key.
- The available order-based, on the fly distinctness enforcer is the cheapest, zero-cost solution.
- distinct_sorted is similar to earlier versions of Adaptive Server row filtering.
- distinct_sorted needs the distinct key to be ordered. Under the various distinctness enforcement scenarios, it is sometimes hard to identify the distinct key for the query processor. In some cases, the row IDs (RIDs) of the base table are used, and such an ordering is available only for the order-based index union.
- When this operator is not legal, it and all of its parents up to the abstract plan root are ineffective.
- The abstract plan distinctness enforcement is described in more detail under the distinct abstract plan operator.

See also

- **Commands** distinct, distinct_hashing, distinct_sorting

distinct_sorting

Description

Specifies the placement of the DistinctSorting operator over the child *derived_table*. Eliminates duplicates according to the semantics of the query.

Syntax

(distinct_sorting derived_table)

Parameters

derived_table

The child *derived_table* for which duplicates are eliminated.

Examples

Example 1 Enforces the existence subquery duplicates semantics by obtaining sort-based distinct correlation values. This example lets the query processor select the best cost-based physical operators for the scans and the joins. The ordering obtained when enforcing the distinctness is useful for the order by clause, provided the query processor selects as its best plan a nested-loop join or a merge join that preserves the ordering. Otherwise, distinct_hashing gives a better plan.

```
select * from t
where t1 in
     (select r1 from r, s where r2=s2)
          order by t1
plan
     "( join
          (distinct_sorting
               (join
                    (scan r)
                    (scan s)
               )
          )
          (scan t)
)"
```

Example 2 Joins tables r, s, and t and evaluates whether the values are distinct. This is an example of “late” evaluation for distinctness. It lets the query processor select the best cost-based physical operators for the scans and the join. The distinct key is t.RID, which is a hidden column that computes the row ID of table t. It is very likely that a distinct_hashing based plan is faster.

```
create view dv(dv1, dv2)
as
     select distinct r1, s1
          from r, s
     where r2=s2

select * from t, dv where t1=dv1
plan
     "(distinct_sorting
```

```
                join
                     (join
                          (scan t)
                          (scan r)
                     )
                     (scan s)
                )
)"
```

Usage

- The returned value is a derived table with distinct values.
- The ability to enforce distinct values must be required and possible over the child derived_table.
- Sort-based distinct enforcement is the most expensive solution.
- distinct_sorting creates an outcome that is ordered on the distinct key. The extra cost of such a plan is zero if the best plan needs this ordering anyway (for example, to put a merge join on top without paying the cost of an extra sort).
- When the sort key is not useful for the parent, distinct_hashing produces a better plan.

See also

- **Commands** distinct, distinct_hashing, distinct_sorting

enforce

Description — Enforces all of the needed properties.

Syntax — (enforce *derived_table*)

Parameters

derived_table
The child-derived table to have its properties enforced.

Examples — This plan forces a join order of tables s and r using merge join, and guarantees that the ordering and partitioning of the children are legal.

```
select r1, s1 from r, s where r2=s2
plan
     "(m_join
          (enforce
               (scan s)
          )
          (enforce
               (scan r)
          )
)"
```

Usage

- The returned value is the enforced derived table that is guaranteed to be legal under any operator.
- You can satisfy the enforceable precondition for any operator with an `(enforce …)` child.
- If the operator is semantically illegal within the query, enforce does not work.
- When the child already provides all needed enforceable properties, the enforce abstract plan operator returns the child-derived table.
- The enforce abstract plan operator is, in general, an expensive operator. It results in the addition of a sort or xchg operator over the child. Use enforce to check whether the query processor rejects an abstract plan based on missing properties. It is better to explicitly enforce the needed properties by using sort or xchg.

See also

- **Commands** sort, xchg, rep_xchg

group

Description — Specifies the placement of a group operator over the child-derived table.

Syntax — (group derived_table)

Parameters — *derived_table*
The child-derived table to be grouped.

Examples — **Example 1** Groups over the index scan of *t*.

```
select t1, sum(t2) from t group by t1
plan
"(group
     (i_scan it1 t)
)"
```

Example 2 The group result is outer to the join, the inner side being an index scan on the join attribute.

```
create view gv(gv1, gv2)
as
select t1, sum(t2) from t group by t1

select * from s, gv where gv1=s1
plan
     "(join
          (group
               (scan t)
          )
          (i_scan is1 s)
)"
```

Usage

- The returned value is the grouped-derived table.
- The query semantics requires grouping.
- The query must contain a group by clause.
- Grouping is not covered by opportunistic enforcement. The group abstract plan operator is only legal over the full set of tables grouped in the query, which is in the from clause of the relational expression that has a group by.
- You must use group when grouping is required.
- There is no query execution plan operator as group, only the abstract plan group operator. The group abstract plan operator lets the query processor make a cost and property-based choice among the available grouping implementations: GroupHashing and GroupSorted.

- The earlier version of the Adaptive Server grouping algorithm, called GroupInserting, is not supported in Adaptive Server 15.0. Hash-based grouping is always faster.
- The group abstract plan operator does not imply the use of a worktable.
- During processing in earlier versions, the worktable and the two processing steps were always exposed by the abstract plan. In Example 1, the abstract plan in earlier versions was:

```
(plan
     (store (work_t Worktable1)
          (i_scan it1 t)
     )
     (t_scan (work_t Worktable1)
)
```

- The 15.0 grouping abstract plan is obtained by replacing the scan of the worktable with the group abstract plan operator over the child of the store. If `(plan …)` has only one child, it is dropped altogether.

See also

- **Commands** group_sorted, group_hashing

group_hashing

Description

Specifies the placement of a GroupHashing operator over the child-derived table.

Syntax

(group_hashing derived_table)

Parameters

derived_table

The child derived table to be grouped

Examples

Example 1 The grouping is performed using the hashing algorithm over the table scan of *t*.

```
select t1, sum(t2) from t group by t1
plan
     "(group_hashing
          (t_scan t)
)"
```

Example 2 Grouping is performed using the hashing algorithm. The group result is outer to the join, the inner side being an index scan on the join attribute.

```
create view gv(gv1, gv2)
as
select t1, sum(t2) from t group by t1

select * from s, gv where gv1=s1
plan
     "(join
          group_hashing
               (scan t)
          )
          (i_scan is1 s)
)"
```

Usage

- The returned value is the grouped derived table.
- The query semantics require grouping.
- group_hashing requires no ordering on the grouping columns.

See also

- **Commands** group, group_sorted

group_sorted

Description

Specifies the placement of the GroupSorted operator over the child-derived table.

Syntax

(group_sorted derived_table)

Parameters

derived_table

The child-derived table to be grouped.

Examples

Example 1 The grouping is performed using the on the fly algorithm over the index scan of *t*, which provides the required ordering on the group by column t1.

```
     select t1, sum(t2) from t group by t1
plan
"(group_sorted
     (i_scan it1 t)
)"
```

Example 2 Performs on the fly grouping over the ordered result of the index scan of *t*. The group result is outer to the merge join, and the grouping column is the equijoin attribute. The inner side of the join is an index scan that provides an ordering on the *s1* equijoin attribute. The grouping preserves its child ordering on the grouping column t1 and ensures the merge join is legal:

```
create view gv(gv1, gv2)
as
select t1, sum(t2) from t group by t1

select * from s, gv where gv1=s1
plan
"(m_join
     (group_sorted
          (i_scan it1 t)
     )
     (i_scan is1 s)
)"
```

Example 3 This abstract plan is very similar to the one in Example 2, but it lets the query processor choose the scan and join operators. The abstract plan is legal only when using indexes of *t* that start with the grouping column t1. If no such index is available, group_sorted and join above it are ineffective.

```
select * from s, gv where gv1=s1
plan
"(join
     (group_sorted
          (scan t)
     )
```

```
        (scan s)
)"
```

Usage

- The returned value is the grouped-derived table.
- The query semantics requires grouping.
- The child derived table is ordered on the grouping columns.
- When the needed ordering is available, the on the fly grouping operator is the cheapest solution.
- group_sorted is similar to the compute clause in earlier versions, but does not require an order by clause: the query processor checks for the availability of the needed ordering.
- The ordering precondition is simpler than the distinct_sorted one: the child must be ordered on all columns in the group by clause, in any order.
- When group_sorted is not legal, it and all of its parents up to the abstract plan root are ineffective.

See also

- **Commands** group, group_hashing

h_join

Description

Specifies that the join is performed using a hash join algorithm.

Syntax

(h_join derived_table1 derived_table2)

Parameters

derived_table_1, *derived_table_2*

Child derived tables. *derived_table1* is the outer table; *derived_table2* is the inner table.

Examples

Example 1 Joins two tables, r and s, using a hash join and performs a hash join over the scans of tables r and s. The query processor chooses the best methods for scanning tables r and s:

```
select * f
rom r, s where r1 = s1
plan
" (h_join (scan r) (scan s))"
```

Example 2 Performs a join between two tables that contain grouped aggregation using a hash join:

```
create view v(v1, v2) as select s1, sum(s2) from s group
by s1
select * from r,v, where r1 = v2
plan
"(h_join)
     (scan r)
     (group_hashing
          (scan s)
     )
)"
```

Usage

- The returned value is the joined derived table.
- The results of a hash join do not provide any ordering to data rows.
- The query must contain an equijoin.
- You can also use hash joins in outer joins and semijoins.
- h_join does not require any ordering condition on its input-derived tables.

See also

nl_join, m_join, peer1, peer2

h_union_distinct

Description

Specifies the placement of the HashUnionDistinct operator, which performs the duplicates-removing union using a hash-based strategy.

Syntax

(h_union_distinct *derived_table...*)

Parameters

derived_table

Specifies a derived table corresponding to each side of the union query.

Examples

Example 1 This plan eliminates the union duplicates through hashing over the table scans:

```
select t1 from t
union
select s1 from s
plan
"(h_union_distinct
     (t_scan t)
     (t_scan s)
)"
```

Example 2 This plan eliminates the union duplicates through hashing over the table scans. The hash-based union does not need any ordering:

```
create view uv(uv1, uv2)
as
select r1, r2 from r
union
select s1, s2 from s

select * from t, uv where t1=uv1
plan
"(h_join
     (t_scan t)
          (h_union_distinct
               (t_scan r)
               (t_scan s)
          )
     )
)"
```

Usage

- The returned value is the union derived table.
- The query semantics require a duplicates-removing union.
- The query must contain union [distinct].
- h_union_distinct is an alternative union [distinct] operator. It eliminates duplicates through hashing.

- h_union_distinct is the cheapest union [distinct] operator when the children have no matching ordering on their entire projection.
- For more information about matching orderings, see m_union_distinct.
- union [distinct] handling based on AppendUnionAll followed by DistinctSorting no longer works. When the children are ordered, MergeUnionDistinct is the best algorithm. Otherwise, it is cheaper to sort each child, before the union, and perform a MergeUnionDistinct, which also preserves the ordering for the parents. If the parents do not need ordering, HashUnionDistinct is the best algorithm.
- h_union_distinct is legal only for union [distinct]. The query processor rejects it for union all, where duplicate rows must be reserved.

See also union, h_union_distinct, m_union_all

hints

Description — Binds together a set of unrelated abstract plan fragments.

Syntax — (insert *derived_table...*)

Parameters

derived_table
Is the child derived table.

Examples — This is a counter-example. Do not use the hints abstract plan operator this way. This plan appears to say: use these indexes for r, s, and t; place r and s anywhere with regard to u, but on the outer side of a nested loop join that has t on the inner side. However, it succeeds only in forcing the three index scans. The nl_join parent of hints is ineffective.

```
        select * from r, s, t, u where r1=s1 and s2=t2 and
        s3=u3
    plan
    "(nl_join
        (hints
            (I_scan ir1 r)
            (I_scan is1 s)
        )
        (I_scan it2 t)
    )"
```

Usage — The parents of hints and all abstract plan operators for the root of the abstract plan expression are ineffective.

insert

Description — Specifies the placement of the insert statement over the child-derived table.

Syntax — (insert derived_table)

Parameters — *derived_table*

The child derived table that provides the new rows to be inserted.

Examples — An index scan provides the rows to be inserted.

```
insert s
select * from t where t1>0
plan
"( insert
     ( i_scan it1 t )
)"
```

Usage

- The returned value is the derived table corresponding to the insert result.
- The query must be part of an insert statement.
- The insert abstract plan operator is useful only for plans that do not have the insert Lava operator at the root.
- In general, a insert SQL statement does not return any useful derived tables. Its outcome is instead the changes made to the result table. However, as the abstract plan operator's tree matches the query execution plan operators tree, you must use the abstract plan insert operator whenever you need an abstract plan that also describes the parent of the insert in the query execution plan.

See also

- **Commands** delete, update

join

Description

Specifies the join of two or more abstract plan-derived tables without specifying the join algorithm (for example, nested loop, merge, or hash join).

Syntax

(join derived_table1 derived_table2)

Parameters

derived_table

The abstract plan-derived tables to be joined.

Examples

t1 is an outer table, and t2 is an inner table. The abstract plan uses a table scan on t1, but chooses the best way to scan table t2. The query processor chooses the join operation:

```
select * from t1, t2
where c21 = 0 and c11 = c22
plan
"(join (t_scan t1) (scan t2))"
```

Usage

- join is a generic logical operator that describes all binary joins (inner join, outer join, or semi-join).
- In an Adaptive Server generated abstract plan, the nl_join, m_join, or h_join operators are used instead of join, to indicate the actual join algorithm.
- The join syntax provides a shorthand method of describing a join involving multiple tables. This syntax:

```
(join
        (scan t1)
        (scan t2)
        (scan t3)
.............
        (scan tN-1)
        (scan tN)
)
```

Is shorthand for:

```
(join
    (join
        ......
           (join
                 (join
                      (scan t1)
                      (scan t2)
                 )
                 (scan t3)
            )
```

```
                    .......
                (scan tN-1)
            )
            (scan tN)
        )
```

- The tables are joined using the tree structure specified in the abstract plan.

See also

- **Commands** peer1, peer2

m_join

Description

An abstract plan derived table that is the result of a merge join between the specified abstract plan derived tables.

Syntax

(m_join derived_table_1 derived_table_2)

Parameters

derived_table

The abstract plan derived tables to be joined. *derivied_table_1* is the outer table and *derived_table_2* is the inner table.

Examples

Example 1 Specifies a merge join of tables t1 and t3, followed by the nested-loop join with table t2:

```
select t1.c11, t2, c21
      from t1, t2, t3
         where t1.c11 = t2.c21
            and t1.c11 = t3.c31
plan
"(nl_join
      (m_join
            (i_scan i_c31 t3)
            (i_scan i_c11 t1)
      )
   (i_scan i_c21 t2)
)"
```

Example 2 Specifies a merge join, during which the table scan of t1 must be sorted so the query processor can perform the merge join:

```
select * from t1, t2, t3
where t1.c11 = t2.c21 and t1.c11 = t3.c31
and t2.c22 = 7
plan
"(nl_join
    (m_join
        (i_scan i_c21 t2)
        (sort
            (t_scan t1)
        )
    )
    (i_scan i_c31 t3)
)"
```

Example 3 Scans of tables t2 and t3 are sorted to get the right ordering for a merge join. Once the merge join is performed, it has the right ordering for a merge join with table t1, which is sorted for this merge join:

```
select * from t1, t2, t3
```

```
where c11 = c23 and c13 = c23
plan
"(m_join
      (sort
         (t_scan t1)
      (m_join
         (sort
             (t_scan t2)
         )
         (sort
             (t_scan t3)
         )
      )
)"
```

Usage

- The tables in the m_join clause are joined using the specified tree structure.
- If the ordering needed by the merge join is not available from the children, you must explicitly specify the sort operator to enforce it (shown in Examples 2 and 3).
- Any m_join operator used to specify a join that cannot be performed as a merge join is ignored.

See also

- **Commands** join, nl_join, h_join

m_union_all

Description

Specifies the placement of the MergeUnionAll operator that issues union all using the merge algorithm. The m_union_all operator preserves the ordering of the child derived table.

Syntax

(m_union_all derived_table …)

Parameters

derived_table

The abstract plan derived tables to be joined in the union.

Examples

Example 1 This plan supplies the required ordering for the order by clause through the ordering that preserves the MergeUnionAll operator over the ordering produced using index scans.

```
select t1 from t where t1>0
union all
select s1 from s where s1>0
order by 1
plan
"(m_union_all
     (i_scan it1 t)
     (i_scan is1 s)
)"
```

Example 2 Provides the ordering needed by the merge join through the ordering preserving MergeUnionAll operator over the ordering producing index scans.

```
create view uv(uv1, uv2)
as
select r1, r2 from r
union all
select s1, s2 from s

select * from t, uv where t1=uv1
plan
"(m_join
          (i_scan it1 t)
          (m_union_all
               (i_scan ir1 r)
               (i_scan is1 s)
          )
)"
```

Usage

- The returned value is the union-derived table.
- The query semantics require a duplicates-preserving union.

- The query must contain a union all.
- m_union_all is an alternative union all operator; it merges its ordered children and produces an ordered output.
- m_union_all does not need the ordering for union all processing. The advantage of m_union_all is that it propagates available ordering, whenever it is needed by the union parents.
- m_union_all is legal only for union all. The query processor rejects it for union [distinct], where the duplicate rows must be rejected.

See also

- **Commands** union, union_all, m_union_distinct

m_union_distinct

Description

Specifies the placement of the MergeUnionDistinct operator. The m_union_distinct operator performs the SQL union operation by eliminating duplicates in the result set using a merge-based algorithm.

Syntax

(m_union_distinct derived_table …)

Parameters

derived_table

The derived tables that occupies each side of a union all query.

Examples

Example 1 This plan supplies the ordering for order by through the ordering preserving m_union_distinct operator over the ordering-producing index scans.

```
select t1 from t where t1>0
union
select s1 from s where s1>0
order by 1
plan
"(m_union_distinct
     (i_scan it1 t)
     (i_scan is1 s)
)"
```

Example 2 This plan provides the ordering needed by the merge join through the ordering preserved by the m_union_distinct operator over the ordering produced by the index scans on `r(r1, r2)` and `s(s1, s2)`.

```
create view uv(uv1, uv2)
as
select r1, r2 from r
union
select s1, s2 from s
select * from t, uv where t1=uv1
plan
"(m_join
          (i_scan it1 t)
          (m_union_distinct
               (i_scan ir12 r)
               (i_scan is12 s)
          )
)"
```

Usage

- The returned value is the union-derived table.
- The query semantics require a duplicates-removing union.
- All children have a compatible ordering.
- The query must contain a union [distinct].

- m_union_distinct is an alternative to the union [distinct] operator; it merges its ordered children, dynamically eliminates the duplicates, and produces an ordered output.
- m_union_distinct is the cheapest union [distinct] operator when all children have a matching ordering on their entire projection.
- Matching means that positionally corresponding columns of each union side select lists are on the same position of a major-to-minor composite ordering.

  For instance, Example 2 is correct, as both indexes `r(r1, r2)` and `s(s1, s2)` provide a composite ordering where the first union columns on all sides, r1 and s1, are both the major attribute and the second one, r2 and s2, are both the minor attribute.
- The earlier versions of Adaptive Server provided a union [distinct] that was based on AppendUnionAll followed by a DistinctSorting; this no longer works. When the children are ordered, m_union_distinct is the best algorithm. Otherwise, it is cheaper to sort each child, before the union, then perform m_union_distinct, which also preserves the ordering for the parents. If the parents do not need an ordering, HashUnionDistinct is the best algorithm.
- This operator is legal only for union [distinct]. The query processor rejects it for union all, where the duplicate rows must be preserved.

See also

- **Commands** union, h_union_distinct, m_union_all

nl_join

Description

Specifies a nested loop join of two or more abstract plan derived tables.

Syntax

(nl_join *derived_table_1, derived_table_2*)

Parameters

derived_table

The abstract plan derived tables to be joined.

Examples

Example 1 Uses a join order of table t2 as the outer table and table t1 as the inner table:

```
select * from t1, t2
where c21 = 0 and c22 = c12
plan
"(nl_join
     (i_scan i_c21 t2)
     (i_scan i_c12 t1)
)"
```

Example 2 Joins table t2 with table t1, and the abstract-plan derived table is joined with table t3:

```
select * from t1, t2
where c21 = 0
and c22 = c12
and c11 = c31
plan
"(nl_join
     (i_scan i_c21 t2)
     (i_scan i_c12 t1)
     (i_scan i_c31 t3)
)"
```

Usage

- The nl_join operator is a join operator that describes all binary joins (inner join, outer join, or semijoin). The joins are performed using the nested-loop, query-execution method.
- The tables are joined in the order specified by the nl_join clause.
- The nl_join syntax provides a shorthand method of describing a join involving multiple tables. This syntax:

```
(nl_join
          (scan t1)
          (scan t2)
          (scan t3)
..............
          (scan tN-1)
          (scan tN)
```

```
)
```

Is shorthand for:

```
(nl_join
     (nl_join
         .........
              (nl_join
                   (nl_join
                       (scan t1)
                       (scan t2)
                   )
                   (scan t3)
              )
              .......
      )
      (scan tN)
)
```

- Returns an abstract plan derived table

that is the result of a join of the specific abstract plan derived tables.

See also m_join, h_join, join

rep_xchg

Description

Variation of the xchg operator where the child-derived table is replicated *n* ways, where *n* is specified in the abstract plan syntax.

Syntax

(rep_xchg derived_table *n*)

Parameters

derived_table

The child-derived table that gets replicated.

n

The number of ways that the child derived table must be replicated. It is also called the degree of replication.

Examples

Replicates the table r three ways and joins with table s, which is partitioned three ways. This example also uses the xchg operator, which allows you to merge the result of the three streams that are joined:

```
select * from r, s where r1 = s1
  plan
"   (xchg 1
      (nl_join
         (rep_xchg 3
             (scan r)
        )
        (scan s)
         )
       )"
```

Usage

- The return value is the replicated derived table.
- You can use this operator only when you use it in parallel mode and in respect to join operators only.
- Useful for large inner tables on which an index is defined on the column in the join predicate, but partitioning of the table is deemed useless with respect to the join predicate. If you repartition this table, you lose the advantage of using an index. In this case, it is often worthwhile to replicate the outer table to have the same number of partitions as that of the large inner table.
- When rep_xchg is placed in an arbitrarily place and the parent operators cannot evaluate a relational operator correctly, the query processor deems it ineffective.

See also

- **Commands** xchg

scalar_agg

Description

Specifies the placement of the ScalarAgg operator, which is used to perform scalar aggregation.

Syntax

(scalar_agg derived table)

Parameters

derived_table

derived_table is the child derived table to be aggregated.

Examples

Example 1 Forces the scalar_agg to run on the index scan, which allows the min and max optimizations.

```
select max(t1) from t where t2=0
plan
"(scalar_agg
     (i_scan it1 t)
)"
```

Example 2 Forces the table scans and the hash-based union operator.

```
create view av(av1, av2)
as
select max(t1), min(t1) from t

select * from av
union
select s1, s2 from s
plan
"(h_union_distinct
     (scalar_agg
          (t_scan t)
     )
     (t_scan s)
)"
```

Example 3 Forces the outer table scan, and, within the subquery, the index scan on the correlation column.

```
select * from r
where r1 > (select max(s1) from s where s2=t.t2)
plan
"(nested
     (t_scan r)
          (subq
          (scalar_agg
               (i_scan is2 s)
          )
     )
```

```
) "
```

Usage

- The returned value is the one-line aggregated derived table.
- The query must contain scalar aggregation (for example, any of the min, max, count, or avg aggregate functions, but no group by clause).
- The scalar_agg abstract plan operator is useful only for plans that do not have aggregation at the root, as an abstract plan is effective even if it does not cover the plan up to its root.
- The scalar_agg abstract plan operator is legal only over the full set of tables aggregated in the query in the from clause of the relational expression that has aggregate functions and has no group by.
- When aggregation is needed, the scalar_agg abstract plan operator is mandatory.
- Pre-processing sometimes requires special handling for scalar aggregates. For example, setting up the avg computation as sum/count, pulling scalar aggregations out of a join as a separate processing step, materializing the scalar result of uncorrelated subqueries using a separate aggregation processing step, all require pre-processing. These are rule-based rather than cost-based operations.
- Abstract plans do not influence pre-processing. The guide only the query processor; the query processor input is the pre-processed query.

See also

join, nested, subq

sequence

Description

Specifies the evaluation of child derived tables from the first to the next-to-last. The last derived table is evaluated after the evaluation of these is done.

Syntax

(sequence *derived_table_1 derived_table_2...derived_table_N*)

Parameters

derived_table

The name of the table being evaluated.

Examples

Example 1 The scalar aggregates, which cannot be combined, requiring two processing steps.

```
select sum(distinct t1), max(t2) from t
plan
*( sequence
     ( scalar_agg
          ( i_scan it2 t)
     )
          ( scalar_agg
               ( distinct_sorted
                    ( i_scan it1 t)
               )
           )
)"
```

Example 2 The self-join view is materialized in a worktable as the first step; the worktable is scanned twice in the second step.

```
create view dv(dv1, dv2)
as
select distinct t1, t2 from t

select * from dv a, dv b where a.dv1=b.dv2
plan
"( sequence
     ( store
          ( distinct_hashing
               ( t_scan t )
          )
     )
     (m_join
          ( sort
               ( t_scan ( work_t (a dv)))
               )
          ( sort
               ( t_scan ( work_t (b dv)))
          )
      )
```

```
) "
```

Usage

- The returned value is the derived table returned by the last child.
- The pre-processed query must have a sequence of multiple processing steps.
- Adding the sequence abstract plan operator is useful only when you must create full abstract plans or to force the parent operators of the sequence. Otherwise, if all that is needed is abstract plan fragments for each processing step, (hints...) can also be used.
- In earlier versions of Adaptive Server, the query processor was worktable-oriented and many queries used a sequence of steps. In Adaptive Server 15.0, worktables are usually hidden because implementation details of some query execution plan operators and the processing have only one step. You need not describe the sequence of steps in most abstract plans. However, there are still cases where a sequence of steps is required. For example, scalar aggregation requires two steps.
- A worktable is still used in self-joined materialized views, as seen in Example 2.

See also

scalar_agg, store

sort

Description

Sorts the child derived table.

Syntax

(sort derived_table)

Parameters

derived_table

The child derived table to be sorted.

Examples

Example 1 This plan forces a merge join order of tables s and r, making sure that the ordering of the children is legal. The t_scan outer child has no ordering and is sorted on s2. If the cheapest access method to r already has the needed ordering, the sort over `scan r` is ineffective.

```
select r1, s1 from r, s where r2=s2
plan
"(m_join
     (sort
          (t_scan s)
     )
     (sort
          (scan r)
     )
)"
```

Example 2 This plan enables the cheap merge_union_distinct by sorting `t_scan r` when r has no index on r1 or r2. The indexes on `s(s1, s2)` and `t(t1, t2)` provide the cheap ordering that makes this plan more attractive than a hash_union_distinct-based plan.

```
select r1, r2 from r
union
select s1, s2 from s
union
select t1, t2 from t
plan
     "(merge_union_distinct
          (sort
               (t_scan r)
          )
          (i_scan is12 s)
          (i_scan it12 t)
     )"
```

Example 3 Forces an early sorting of table r before the nested loop join is performed. The ordering on the outer table r is preserved beyond the join such that it satisfies the ordering required by the order by clause. This plan would be more expensive if the sort for the order by clause was performed after the join, because the join increases the result cardinality.

```
select r1, s1 from r, s where r2<s2
order by r1
plan
"(nl_join
     (sort
          (t_scan r)
     )
     (i_scan is2 s)
)"
```

Usage

- The returned value is the ordered derived table.
- Ordering is a physical property because it depends on the actual physical operators that form a plan fragment and is needed by the algorithms in some parent physical operators.
- Ordering is a useful physical property that enables many cheap operators, such as MergeJoin, MergeUnionDistinct, DistinctSorted, and GroupSorted.
- Ordering is available when it is provided by the child operator.
- Some operators produce an ordering, as IndScan and sort.
- Other operators preserve the ordering of their child: nested loop joins and merge joins preserve the ordering of their outer child.
- sort is an expensive operator that does not modify the result relation contents, but only the ordering of the rows.
- It is always cheaper to use ordering obtained from an existing child. If no child provides the needed ordering, a sort is useful.
- The query processor computes the attributes that need ordering, based on the semantics of the query. The sort abstract plan operator needs not specify the sort key: the query processor sorts all needed columns.
- If a sort abstract plan operator is placed over a child that has all needed ordering already available, then no sort is generated and the child-derived table is directly returned.

See also

- **Commands** enforce, rep_xchg, xchg

store

Description — Stores the result of the child derived table evaluation.

Syntax — (store *derived_table*)

Parameters — *derived_table*
The name of the child derived table being materialized.

Examples — This example self-joins a view containing the distinct operator. It first materializes the view, then performs the self join. The view is evaluated in the first section of the sequence operator and the result is materialized. The worktable is joined to itself using a merge join during the second section of sequence. These steps are more efficient than evaluating the same view twice:

```
create view vg(v1, v2)
as
select distinct t1, s2 from s, t
where s1 = t1 and t3 = 1

select * from vg a, vg b
where a.v1 = b.v1 and a.v2 b.v2
plan
("sequence
     (store
          (distinct hashing
               (n1_join
                    (t_scan s)
                    (scan t)
               )
          )
     )
     (m_join
        (sort
          (t_scan (work_t (b Worktable)))
        )
        (sort
          (t_scan (work_t (a Worktable)))
        )
      )
)"
```

Usage

- The derived table you specify with the store operator is scanned and materialized into a worktable.
- Worktables created in the store operator can be referred to as Worktab1, Worktab2, and so on.

See also — store_index, sequence

store_index

Description

Specifies the placement of a StoreIndex operator, which is also the general reformatting strategy. In this strategy, the child derived table is materialized and a clustered index is built on the joining columns.

Syntax

(store_index *derived_table*)

Parameters

derived_table

The child derived table to be reformatted.

Examples

Example 1 This abstract plan uses the scan limiting index for each of the search clauses. Inside the nested loop join, it reformats the index scan result to a worktable indexed on the join column, r1:

```
select * from r, s
where
     r1=s1
     and r2=0
     and s2>0
plan
"(n1_join
     (i_scan is2 s)
     (store_index
          (i_scan ir2 r)
     )
)"
```

Example 2 This abstract plan uses the scan limiting index with the search clauses. It reformats the index scan result to a work table indexed on the join column, r1. The ordering needed by order by is provided by the r(r3) index scan and is preserved by the nested loop join:

```
select r3, s3 from r, s
where
     r1=s1
     and s2=0
order by r3
plan
"(nl_join
     (i_scan ir3 r)
     (store_index
          (i_scan is2 s)
     )
)*
```

Example 3 This abstract plan places the lower merge join and its order-providing index scan inside a nested loop join. It avoids the repeated evaluation of the merge join by placing a store_index over it. The ordering the merge join reuires on column r(r2) is provided by the index scan and is preserved by the nested loop join:

```
select r4, s4, t4 from r, s, t, u
where
     r1=s1
     and s2=t2
     and s3 like "%abcdef%"
     and t3 like "%123456%"
     and r2=u2
plan
"(m_join
     (nl_join
          (i_scan ir2 r)
          (store_index
               (m_join
                    (i_scan is2 s)
                    (i_scan it2 t)
               )
          )
     )
     (i_scan iu2 u)
)"
```

Usage

- The returned value is the derived table produced by the index scan of the reformatted worktable.
- As in earlier versions of Adaptive Server, reformatting refers to storing a derived table in a worktable and creating a clustered index on the attributes used by the parent joins. This allows the placement of the index scan of the worktable inside a nested loop join.
- In earlier versions, only single table scans were reformatted. In Adaptive Server 15.0, reformatting has been generalized to any derived table. As shown in Example 2, the result of a two-table join is stored in a worktable.
- store_index is an expensive operation because it involves a sort.
- In Adaptive Server 15.0, the merge join and hash reduct the necessity for store_index. It is useful when a nested loop join and store_index on its inner side is cheaper than a merge join and sorting both sides. However, in such cases, the derived table under store_index is small and the outer table is large.

See also

nl_join, h_join, m_join, sort

union

Description — Specifies the placement of one of the union operators.

Syntax — (union *derived_table*...)

Parameters

derived_table
A derived table used for each side of the union operator.

Examples

Example 1 This plan forces the table scan on each side of the union.

```
select * from t where t1>0
union
select * from s where s1>0
plan
"(union
     (t_scan t)
     (t_scan s)
)"
```

Example 2 This plan lets the query processor choose the operators for the union plan fragment and forces the union to be inside a hash join that has a table scan on the outer side.

```
create view uv(uv1, uv2)
as
select * from r
union
select * from s

select * from t, uv where t1=uv1
plan
"(h_join
     (t_scan t)
     (union
           (scan r)
           (scan s)
     )
)"
```

Usage

- The returned value is the union-derived table.
- The query semantics require a union.
- Each child abstract plan of a union must correspond to the query fragment on the corresponding union side.
- During pre-processing, unions and joins are sometimes permuted. This is a rule-based rather than cost-based operation.

- Abstract plans do not influence pre-processing. They guide only the query processor, and the query processor input is the pre-processed query.
- The query processor knows from the query whether this is a union all or a union distinct. The abstract plan does not need to specify.
- For each of the union all/distinct plans, there are several physical operators. The query processor chooses the best cost-based physical operator.

See also h_union_distinct, m_union_distinct, union_all, m_union_all

union_all

Description

Specifies the placement of the AppendUnionAll operator, which performs the union all operation.

Syntax

(union_all derived_table …)

Parameters

derived_table

Indicates a derived table for each side of the union all operator.

Examples

Example 1 This plan forces the union_all operator and the table scan on each side of the union.

```
select * from t where t1>0
union all
select * from s where s1>0
plan
"(union_all
     (t_scan t)
     (t_scan s)
)"
```

Example 2 This plan lets the query processor chose the operators for the union_all children and forces the union to be inside a hash join that has a table scan on the outer side.

```
create view uv(uv1, uv2)
as
select * from r
union all
select * from s

select * from t, uv where t1=uv1
plan
"(h_join
     (t_scan t)
          (union_all
               (scan r)
               (scan s)
          )
)"
```

Usage

- The returned value is the union-derived table.
- The query semantics require a duplicates-preserving union.
- The query must contain a union all.
- union_all is the basic union all operator; it drains each child.

- This operator is legal only for union all. The query processor rejects it for union [distinct], where the duplicate rows must be rejected.

See also

- **Commands** union, m_union_all

update

Description — Specifies the placement of the update statement over the child-derived table.

Syntax — (update derived_table)

Parameters

derived_table
The child derived table that qualifies the rows to be updated and provide the new values.

Examples — The rows are qualified for updating by an index scan.

```
update t set t1=t2+1 where t1<0
plan
"( update
     ( i_scan it1 t )
)"
```

Usage

- The query must be part of an update statement.
- The update abstract plan operator is useful only for plans that do not include the update Lava operator at the root.
- A update SQL statement does not return any useful derived tables. Instead, its outcome is the changes made to the result table. However, because the abstract plan operator's tree must match the query execution plan operator's tree, you must use the abstract plan update operator when you need to use an abstract plan that also describes the parent of the update in the query execution plan.

See also

- **Commands** delete, insert

use optgoal

Description — Specifies a directive for optimization goal to be used for the query.

Syntax — (use optgoal *optgoal_name*)

Parameters — *optgoal_name*
The name of the optimization goal to be used for the query.

Examples — This directive specifies use of the allrows_dss optimization goal for a query.

```
select * from publishers p, titles t
where t.pub_id = p.pub_id
plan
"(use optgoal allrows_dss)"
```

Usage

- The optimization goal is applied to the specified query only.
- The query level specification overrides any session level or server level specification of an optimization goal.
- The default optimization goal is allrows_mix. allrows_dss is available on an experimental basis.
- No other Abstract Plan operator may be specified with the use optgoal directive.

See also — use opttimeoutlimit

use opttimeoutlimit

Description

Specifies a directive for optimization timeout limits to be used for the query. Optimization timeout describes the percentage of the estimated query execution time that Adaptive Server must spend in optimizing the query.

Syntax

(use optgoal *opttimeoutlimit_value*)

Parameters

opttimeoutlimit_value
The value for the timeout limit.

Examples

This example shows the directive to used an optimization timeout limit of 100% for a query.

```
select * from publishers p, titles t
where t.pub_id = p.pub_id
plan
"(use opttimeoutlimit 100)"
```

Usage

- The optimization timeout limit is applied to the specified query only.
- The query level specification overrides any session level or server level specification of an optimization timeout limit.
- The valid values for opttimeoutlimit are 1 to 1000. The default value is 10.
- No other Abstract Plan operator may be specified with the use opttimeoutlimit directive.

See also

use optgoal

values

Description — Specifies the placement of a table literal.

Syntax — (values)

Examples — This plan forces the union operator over a tables scan and two table literals.

```
select t1 from t
union
select 1
union select 2
plan
"(h_union_distinct
     (t_scan t)
          (values)
          (values)
)"
```

Usage

- The returned value is the table literal as a derived table.
- A table literal is given in SQL through the table value constructor, the select relational expression that has no from clause, or the values clause of the insert statement.
- The values abstract plan operator is useful only when a larger abstract plan is needed.
- When the query contains several table value constructors, the values abstract plan operators match them positionally.

See also

- **Commands** insert, scalar_aggh_join

xchg

Description

xchg (pronounced “exchange”) is the key operator for building parallel query plans. xchg forces a repartitioning of a table with the specified “degree” on the child-derived table, where the value of degree is the number of partitions created. Repartitioning is a dynamic operation, so the results of repartitioning are never materialized.

Syntax

(xchg derived_table *n*)

Parameters

derived_table

The child derived table to which you apply repartitioning.

n

The number of repartitions for the child-derived table

Examples

Example 1 The examples in this section uses this table:

```
create table r(r1 int, r2 int, r3 int)
partition by range(r1, r2)
(
     p1 values <= (100,100),
     p2 values <= (200,200),
     p3 values <= (300,300)
)

create table s(s1 int, s2 int, s3 int)
partition by hash (s2)
     (p1,p2,p3,p4)
```

In this parallel scan (with parallelism enabled), xchg takes the three streams coming from the scan of tables r and merges these streams into one, using a “many-to-one” xchg operator:

```
select * from r
  plan
    "(xchg 1
    (scan r)
       )"
```

Example 2 Joins two partitioned tables in parallel. The partition on table r is not helpful for the join predicate and must be repartitioned with the same scheme as table s. You need not specify the repartitioning columns, the type of partitioning, or even the boundary values of the partition, just the degree of repartitioning:

```
select * from r, s
     where r2 = s2
plan
```

```
"(xchg 1
    (join
         (xchg 4
                (scan r)
                       )
          ( scan s)
     )
 )"
```

In this example, the first xchg operator (`xchg 1`) uses a many-to-one mode and merges the input streams to form a single stream. The query processor uses the next xchg operator, (`xchg 4`), to perform a many-to-many repartitioning. In this case, the range-partitioned based table is repartitioned four ways using hash partitioning, then joined to table s.

Example 3 This example includes a grouped aggregation on table r. The `xchg 2` operator repartitions the derived table, which is created by the scan's output. This repartitioning occurs on the grouping columns, and the group-hashing operator preforms the grouping in parallel. The result of this grouping creates two streams that are merged using the `xchg 1` operator at the top of the QEP:

```
select count(*), r1 from r group by r1
"plan
  "(xchg 1
      (group_hashing
          (xchg 2
              (t_scan r)
           )
       )
)"
```

Usage

- The result of xchg applied to a child-derived table is a re-partitioned stream.
- An xchg value of 1 informs the query processor to merge the data stream.
- xchg values greater than 1 are either "many-to-many" or "one-to-many" operators, depending on the partitioning of the child-derived table.
- If a query does not need an xchg operator because the child-derived table already has the partitioning property, the query processor does not apply the xchg operator.
- The query processor evaluates the appropriate partitioning and the columns that require this partitioning.

See also

rep_xchg, enforce

Glossary

Abstract plan — Instructions to Adaptive Server Enterprise query processing on the access path to use to manipulate the data for a query; for example, specifying join order, join algorithms, index usage, etc.

Arithmetic operator — Symbols that allow you to create an arithmetic expression in SQL statements. Addition (+), subtraction (-), division (/) and multiplication (*) can be used with numeric columns. Module (%) can only be used with the integer datatypes. See also **comparison operator**.

Bushy parallelism — Occurs when several CPUs execute different sub-plans of a complex query plan in parallel.

Bushy tree plan — A query plan in which some join operations have two or more direct children that are also join operations. Non-bushy plans are typically referred to as left deep trees (in which the right child is a scan operator) or a join list in which tables are joined in linear order.

Buffer replacement strategy — See **buffer reuse strategy**.

Cache strategy — The optimizer specification for the characteristics of the buffer cache to be used by the execution engine to process pages of explicit tables referenced in the query or transient information used during internal processing of the query.

Code generation — The representation of the plan used by the optimizer is designed for efficient comparisons between competing best plans, whereas the representation of the plan used by the execution engine is designed for efficient execution. Code generation is the mechanism in query compilation that converts the optimizer best plan representation into the execution engine plan representation.

Compilation — Phase of query processing which analyzes the query text and creates a query plan to be provided to the execution engine.

Cost based pruning — Query optimization technique to use estimated costs of sub-plans to avoid analyzing more expensive sub-plan alternatives.

CPU cost — Optimizer's estimated amount of CPU needed to process a query.

Data flow engine	A descriptive term for the execution engine that implies one large plan with fewer materialization steps.
Derived statistics	Statistics computed and used during search space operations, and whose lifetime is for the duration of the query optimization. These statistics include modified histograms, densities, column widths, and table statistics that exist after all predicates (filtering or otherwise) are applied.
Dimension table	A table in a "star" schema which, as a primary key, can be combined with other dimension tables in the same "star" schema to form a composite key to access or join information in the central fact table.
Decision support system (DSS)	Applications characterized by queries that process large volumes of data.
Dynamic partition elimination	If the value of unbound predicates becomes known at runtime, one or more partitions can be eliminated from the list of partitions to be scanned. This is also true for a column whose value becomes known for the inner scan of a nested loop join.
Equi-partitioned	Two tables having compatible partitioning keys and partitioning criteria. If two tables have same number of partition keys with compatible data types, and the partition criteria, such as the intervals, are the same for the range partitions, the two tables are considered equi-partitioned.
Exchange operator	This is an operator which, when applied to a data stream, can change the degree or data semantics of the stream. It takes part in repartitioning. The exchange operator has a producer side and a consumer side. The producer tasks run the relational operator clone below the exchange. The consumer side runs in as many clones as the consumer operator needs to run.
Existence scan	A scan algorithm based on stopping the scan of the table as soon as the first row is fully qualified. Typically introduced by tables from a flattened exists subquery.
Fact table	The main table of a "star" database schema that has a composite key composed of attributes that are foreign keys for several dimension tables.
Fetch	A fetch retrieves one or more rows and changes the current cursor position in the cursor result set. Also called a cursor fetch.
Generic Column	Normally a column in a table referenced in a query, but also an abstraction that includes interesting expressions; for example, those that can be used in an expression join.

Generic Table — A table referenced in a query, but also a convenient abstraction to represent any object that is permutated in the join order by the optimizer; for example, a subquery modeled as a generic table.

Global index — Global indexes refer to indexes on partitioned tables. A global index results when an index and the table have different partitioning strategies, such that index leaf pages in global indexes point to more than one partition.

Global statistics — Statistics that apply to all data values of a table.

Greedy search strategy — Any optimizer permutation strategy whose goal is to obtain a query plan very quickly. The result is likely to be a sub-optimal plan because very coarse criteria is used to avoid looking exhaustively at all query plans.

Hash based aggregation — Strategy for evaluating group by aggregates in which the group is looked up by a hash key on the grouping columns.

Heuristics based pruning — Optimization techniques where portions of search space (tree shapes, permutations) are skipped based on a set of predetermined rules applicable to a query.

Histogram tuning factor — A factor used to increase the number of steps in a histogram over the default or specified step count, used only in cases in which frequency cells exist. For example, a factor of 3 could potentially increase the default step count of 20 to 60 if frequency cells exist in the distribution.

Histogram weight array — An array of float values associated with a histogram which gives either the percentage of the table selected by that cell (for table-normalized histograms), or the percentage of a cell selected by a particular predicate (for cell-normalized histograms).

Horizontal parallelism — Partitioned parallelism and independent parallelism are classified as horizontal parallelism. The ability to run multiple instances of operators on different data sets located across different storage units is also called horizontal, or partitioned, parallelism.

In-order Join — A join operation where some (or all) of the joining attributes from the outer join are ordered, as occurs from a sort or index scan.

Independent parallelism — Also known as bushy parallelism. See **bushy parallelism**.

Index intersection — An access path in which several RIDs from two or more indices of a table are joined to obtain the set of RIDs that qualify the result set for the scan on the table with several SARGs that are anded.

Index union

An access path in which several RIDs from two or more indices of a table are unioned, with duplicate removal, in order to obtain a set of RIDs that qualify the result set for the scan on the table with several SARGs that are ored.

Iterator

An execution engine operator. Query results are encapsulated using iterators that are self-contained software objects that accept a stream of rows from null or n-ary data sets. The role of an iterator is to process many iterations of a data set across many nodes in serial or parallel. Iterators do not know what the source of the data stream is, if the source is external, such as another iterator, or do know the source if source is internal, such as when the source is produced by the iterator itself. For each iteration of a data set, the iterator applies a predefined behavior to the data set being processed, manipulating the data according to the specification of that iterator. For example, scanning rows from a table on disk can be the behavior of one type of iterator. A key feature of iterators is that regardless of what type the iterator is and what behavior is associated with it, all iterators follow the same mode and have the same external interface. They all open data streams, iteratively read the streams, then process and close the streams.

Join density

See **total density**.

Join histogram

An intermediate histogram created during optimization only and then discarded. It is the result of taking the histograms of two columns that are equi-joined and producing a histogram which models the data distribution after the join has occurred.

Lava query plan

An “upside down” tree of Lava operators. The top operator can have one or more child operators, which in turn can have one or more child operators, and so on, thus building the upside down tree of operators. The exact shape of the tree and the operators in a lava query plan are chosen by the optimizer and the plan is executed by the Lava Query Execution Engine.

Lava operator

A self-contained software object that implements a basic operation; it may be chosen by the optimizer as part of a Lava query plan. Some examples of Lava operators are: the ScanOp that reads rows from database tables, the MergeJoinOp that implements the merge join, and the InsertOp that inserts rows into tables. There are 32 Lava operators.

Lava Query Execution Engine

The module in Adaptive Server that executes the Lava query plan chosen by the optimizer. Query plans are executed by calling methods of the top operator in the plan (the RootOp), which calls methods on its child operator(s), which in turn, call methods on their child operators, down to the leaf operators if necessary, to generate a result row. Result rows are generated at the leaf operator nodes and are passed up the operator tree to the RootOp, which consumes them (that is, sends them to the client) or assigns values to variables.

LIO

See **logical IO cost**.

Left deep tree plan

This is an alternative tree-based description of a join order on a set of tables. It is a tree shape of query plan structure where right nodes are always leaf nodes. This order of tables in tree shape can be influenced by the set forceplan option.

Local indexes

A table index that is partitioned the same way as its data.

Local server

The server or node where a query originates.

Local statistics

Statistics that apply to data values for a specific partition on a partitioned table.

Logical IO cost

The optimizer's estimate of the number of logical reads.

Logical operator

In the context of the where or on clause, the keywords and, or, and not are part of the predicate that filters rows.

In the context of optimization, this describes an operation in query processing without specifying a specific algorithm, such as join, scan, sort.

Logical partitioning

A way to partition data into *n* units such that when function *f* is applied to the keys of a tuple *t*, it generates an ordinal number that maps to one and only one partition. In other words it is 0 <= f(t,n) <= n - 1. An exception to this is round-robin partitioning, where such mapping does not hold.

Logical property

A property that is common to a set of sub-plans associated with a set of tables (equivalence class). An example is row count, since no matter how the set of tables are joined the same row should exist after the same predicates are applied.

Mixed workload

Relational queries are broadly classified into the simple transactional queries found in OLTP environments and the complex queries found in DSS environments. In a production environment, database systems are configured to run transactional or complex queries at the same or different times. Installations that support both are referred to as “mixed workload” systems. Since it is not always possible to predict the type of workload, it is important to support both OLTP and DSS queries in the same configuration of the data processing system to efficiently support workloads of all types.

Multi-way joins	join queries in which some tables join to two or more tables, resulting in star join and snowflake join configurations.
OLTP	Online transaction processing, an application characterized by many short transactions containing queries that use minimal resources.
Optimization goals	A set of user defined goals that can be specified to influence which optimization techniques are considered, so as to generate plans suitable for a specific query or application.
Optimization rules	When determining the best plan, most decisions made by the optimizer are based on estimated costs. Some decisions are based on specific characteristics of the query predicate and the tables involved in the query. For example, it is best to have a join predicate between two tables, except in the case of a star join, so that other join order permutations are not evaluated.
Optimization timeout	The mechanism by which the optimizer stops searching for a better plan than the current best plan because the compilation time has exceeded the specified criteria. The then current best plan is used for processing the query.
Ordering	A specific sequence (ascending or descending) of attributes in a result set as would occur from an index scan or a sort.
Parallelizer	A component of the optimizer that adds scheduling information to a plan, re-evaluates the plan, and then generates the best parallel plan. The parallel optimizer is really a scheduler and parallelizer that looks at a set of parallel plans annotated with resource usage. Based on the total resources available for a query, it finds the best plan based on response time.
Partition elimination	Given a query that has predicates on the partitioning keys, it is possible to find out which partitions qualify a given predicate. However, predicates that are currently useful for partition elimination must qualify as conjunctive or disjunctive predicates on a single table of the form: column <relop> <literal>.
Partitioning key	A search condition that evaluates a partition specification. The set of columns participating in the key specification is known as the partitioning key.
Partitioned parallelism	The data is divided into more than one physical partition so that each partition can be accessed in parallel and can be managed by a worker thread. The IO and CPU parallelism resulting from such a configuration speeds up the SQL queries in proportion to the number of partitions.
Physical operator	An algorithm implementing a logical operator, such as index scan, sort-merge join, nested loop join, and so on.

Physical property A property associated with a physical operator and dependent on the actual algorithm implemented by that operator and on the physical properties of its children (thus, recursively, on the physical operators in the sub-plan). For example, the ordering (from an index scan or sort) of the outer child is usually inherited after subsequent join operators are evaluated, but each plan in an equivalence class has potentially different ordering depending on the underlying operators used in the sub-plan.

PIO See **physical IO cost**.

Physical IO cost The optimizer's estimated number of physical reads.

Pipelined parallelism In a multi-step SQL operation, each independent step can begin execution before the preceding step is completed. More than one processor can work on single query, resulting in shorter response times.

Pipelining Two sets of threads serve as producers and consumers. While producers put data in a shared buffer, consumers can process the data in the shared buffer concurrently.

Plan cache In the context of the optimizer, a usage of the procedure cache in which it stores useful partial plans (plan fragments) that may be necessary in future construction of complete plans. The plan cache only exists during query compilation and is released before query execution.

Pruning An optimizer technique of searching for the best execution plan. Only promising sub-plans are retained; that is, the ones that could be part of the best total plan. The optimizer uses cost-based and heuristics-based pruning.

Projection The set of attributes available on the output of an operator. This implies a minimal set of attributes in which each attribute is needed by some parent of the respective operator.

Range partitioning In this table-partitioning scheme, a data set for one or more attributes is partitioned on the value range. Thus, every row can be pinpointed to a given partition.

Round-robin partitioning A scheme that is best suited for load balancing. The data set is distributed in round-robin fashion, and no attention is given to where a data value ends up.

Scalar A term for an SQL expression that produces a single value, not a set of values.

Search criteria A user-specified or system-determined criteria used to influence optimization techniques used to generate plans.

Search engine — A component of the query optimizer that generates and evaluates alternative execution plans and selects the most optimal one. The search engine comprises three key components; search criteria, search space, and search strategy.

Search space — The exhaustive set of plans considered for selection by the search engine.

Search strategy — A module of the search engine that generates a specific search space and selects among the alternatives available in that search space.

Semi-join — A join algorithm which terminates the inner scan for each outer row as soon as the first inner row qualifies.

Snowflake schema joins — Queries where several dimension tables are joined to local fact tables that in turn are joined to a central fact table. There are no join clauses between the dimension tables (cross products). The fact tables are large compared to their respective dimension tables.

Star schema joins — Queries where several dimension tables are joined to a central fact table. The dimension tables do not have join clauses between them (cross products) and the fact table is large compared to its dimension table.

Store operator — Operator that creates a fully materialized table, usually in support of the reformatting strategy.

Surrogate Pairs — A coded character representation for a single abstract character that consists of a sequence of two code values. Surrogate pairs are designed to allow additional 220 code values to be represented in the Unicode Standard. The concept only applies to UTF-16 encoding.

Table normalized histogram — Normally called a histogram; that is, a histogram computed by update statistics in which the weight array values (fractions of table rows) always sum to 1.0.

Transitive closure — A set of attributes connected by equi-joins.

Tuple filtering — An execution operator with a single input stream. It assumes that all referenced attributes are ordered and eliminates duplicate tuples based on that assumption.

UTF-16 — Universal Character Set (UCS) Transformation Format, 16-bit form. In UTF-16, each UCS-2 code value represents itself. Code values beyond the BMP (Basic Multilingual Plane: 0..0xFFFF) are represented using pairs of special 16-bit codes called surrogate pairs. This allows an additional 2^{20} (1 MB) code values to be represented, using 4 bytes to do so.

UTF-8 — UCS Transformation Format, 8-bit form. UTF-8 is a variable length encoding of the Unicode Standard using 8-bit sequences, where the high bits indicate which part of the sequence a byte belongs to.

Vertical parallelism The ability to use multiple CPUs simultaneously on more than one operator in a single plan fragment. Also called pipelined parallelism.

Virtual column Any column in the output of an execution engine operator that does not map directly back to a persistent column of a table. Mostly, but not always, an expression involved on one side of a join.

Index

Symbols

::= (BNF notation)
    in SQL statements xii
, (comma)
    in SQL statements xii
{} (curly braces)
    in SQL statements xii
() (parentheses)
    in SQL statements xii
[ ] (square brackets)
    in SQL statements xii

A

abstract plans 171
    legacy partial plans 179
    new directives 175
    operators 172
    optimization goal 175
    optimization timeout limit 175
    semantics 177
    specifications for operators 203
    support for pre 15.0 operators 176
    syntactic qualification 178
    syntax 172
    syntax, new 175
    worktables and steps 177
accessing
    query processing metrics 166
adding
    statistics for unindexed columns 183
adding statistics 183
adjustment
    managing run time 93
    recognizing run time 93
    reducing run time 94
    run time 92
attribute-insensitive operation
    parallelism 48
attribute-sensitive operation
    parallelism 62
automatically
    update statistics 190
automatically updating
    statistics 187

B

Backus Naur Form (BNF) notation xii
BNF notation in SQL statements xii
brackets. *See* square brackets [ ]

C

case sensitivity
    in SQL xiii
clearing
    query processing metrics 170
column-level
    statistics 193
column-level statistics
    generating the **update statistics** 195
    truncate table and 193
    update statistics and 193
comma (,)
    in SQL statements xii
composite indexes
    update index statistics and 196
compute by processing 138
control parallelism at session level 36
controlling parallelism for a query 37
conventions
    See also syntax
    Transact-SQL syntax xii
    used in the Reference Manual xi
converted

search arguments 8
creating
column statistics 194
search arguments 19
curly braces ({}) in SQL statements xii

D

data types
join 14
datachange function
statistics 188
degree
setting max parallel 34
delete 114
delete 88
delete statistic 201
delete statistics command
managing statistics and 200
density
join 14
derived
SQL tables 20
differing parallel query results 38
directives, new
abstract plans 175
discontinued trace commands
XML 163
drop index command
statistics and 200

E

elimination
partition 90
emit
operator 104
enable
parallelism 33
engine
query execution 21
equi-join
transitive closure 9
exceptions
optimization goals 17
exchange
operator 154
exchange
operator 42
pipemanagement 43
worker process mode 44
executing
query processing metrics 166
execution
preventing with **set noexec on** 95
expressions
join 15

F

factors
analyzed for optimization 6
from table 106
function
datachange, statistics 188

G

goals
optimization 16
optimization exceptions 17
group sorted
operator 132
group sorted agg
operator 136
grouped by aggregate message 135

H

hash based table scan 50
hash distinct
operator 134
hash join
operator 126
hash union
operator 140
hash vector aggregate

operator 137
histograms
join 14
steps, number of 196

I

index scan 52
clusteed, partitioned table 56
clustered 56
covered using non-clustered global 55
global non-clustered 52
non-clustered, partitioned table 56
non-covered, global non-clustered 52
indexes
search arguments 12
update index statistics on 196
update statistics on 196
insert 114
insert 88
introduction
query processing metrics 165

J

job scheduler
update statistics 190
join
both tables with useless partitioning 66
outer 73
parallelism 62
parallelism, one table with useful partitioning 64
parallelism, replicated 68
parallelism, tables with same useful partitioning 63
semi 73
serisl 71
join
density 14
expressions 15
histograms 14
mixed data types 14
or predicates 15
ordering 15
join operator 121
joins 14

L

lava
operator 103
operators 24
query execution 27
query plan 100
query plans 22
lava query engine 22
legacy partial plans
abstract plans 179
locking
statistics 198
log scan 111

M

maintenance
statistics 193
max repartition degree
setting 35
max resource granularity
setting 34
merge join
operator 123
merge union
operator 141
minor columns
update index statistics and 196

N

nary nested loop join
operator 128
nested loop join 122
non leading columns
sort statistics 198
non-equality
operators 13

O

object sizes
tuning 20
operations
insert, **delete**, **update** 88
operator
delete 114
emit 104
exchange 154
exchange 42
group sorted 132
group sorted agg 136
hash distinct 134
hash join 126
hash union 140
hash vector aggregate 137
insert 114
lava 103
merge join 123
merge union 141
nary nested loop join 128
remote scan 151
restrict 144
rid join 152
scalar aggregate 143
scan 104
scroll 151
sequencer 148
sort 144
sort distinct 133
sqfilter 152
store 146
text delete 115
union all 141
update 114
vector aggregate 135
operators
abstract plans 172
lava 24
non-equality 13
optimization 5
optimization
additional paths 10
example search arguments 13
factors analyzed 6
goals 16
goals, exceptions 17
limit time optimizing query 17
operators 5
predicate transformation 10
problems 18
query transformation 8
techniques 5
timeout limit, abstract plans 175
optimization goal
abstract plans 175
optimizer
query 3
option
set rowcount 39
or list 104
or predicates
join 15
ordering
join 15
output
statement 96
XML diagnostic 158
overview
query processing 1

P

parallel
query execution model 42
query plans 40
query processing 31
setting max degree 34
setting max resource granularity 34
table scan 49
union all 60
parallel degree
setting max scan 35
parallel processing
query 32
parallelism 18
attribute-insensitive operation 48
attribute-sensitive operation 62
controlling at session level 36
controlling for a query 37
distinct vector aggregation 77

enable 33
in-partitioned vector aggregation 73
join 62
join, both tables with useless partitioning 66
join, one table with useful partitioning 64
join, replicated 68
join, tables with same useful partitioning 63
outer joins 73
query with IN list 77
query with OR clause 79
query with order by clause 81
reformatting 69
re-partitioned vector aggregation 74
semi joins 73
serial join 71
serial vector aggregation 76
setting number of worker processes 33
SQL operatoions 47
table scan 48
two phased vector aggregation 75
vector aggregation 73
parentheses ()
in SQL statements xii
partition
skew 91
table scan 50
partition elimination 90
permissions
XML 163
pipe management
exchange 43
plans
legacy partial, abstract plans 179
query 96
predicate
transformation 10
problems
optimizing queries 18
process_limit_action 93

Q

QP metrics *See* query processing metrics
queries
execution settings 95
problems optimizing 18
query
execution engine 21
lava execution 27
limit optimizing time 17
not run in parallel 92
optimizer 3
OR clause 79
parallel execution model 42
parallel processing 32
plans 96
select-into clause 85
set local variables 39
with IN list 77
with order by clause 81
query analysis
showplan and 95
query optimization 157
query plan
lava 100
query plans
lava 22
parallel 40
query processing
overview 1
parallel 31
query processing metrics
accessing 166
clearing 170
executing 166
introduction 165
sysquerymetrics view 167
using 166

R

reduce
impact 199
referential integrity constraints 117
reformatting
parallelism 69
remote scan
operator 151
restrict
operator 144

results
differing parallel query 38
rid join
operator 152
rid scan 109
row counts
statistics, inaccurate 201
run time
adjustment 92
managing adjustment 93
recognizing adjustment 93
reducing adjustments 94

S

samplicing
use for updating statistics 186
sampling
statistics 185
scalar aggregate
operator 143
scalar aggregation
serial 59
two phased 58
scan
clustered index 56
clustered index on partitioned tables 56
index 52
index global non-clustered 52
index non-covered of global non-clustered 52
index, covered use non-clustered global 55
local indexes 56
non-clustered, partitioned table 56
operator 104
scan types
statistics 198
scroll
operator 151
search arguments
converted 8
creating 19
example of optimization 13
indexes 12
transitive closure 8
select-into
query 85
semantics
abstract plans 177
sequencer
operator 148
serial
scalar aggregation 59
union all 61
serial table scan 48
set
local variables 39
XML command 158
set
examples 36
set rowcount option 39
setting
max scan parallel degree 35
number of worker processes 33
setting mac parallel degree 34
setting max repartition degree 35
setting max resource granularity 34
showplan
query plans ASE 15.0 96
statement level output 96
using 94, 95
skew
partition 91
sort
operator 144
statistics, unindexed columns 198
sort distinct
operator 133
sort requirements
statistics 198
sqfilter
operator 152
SQL
parallelism 47
SQL tables
derived 20
square brackets [ ]
in SQL statements xii
statement level output 96
statistics
adding for unindexed columns 183
automatically updating 187

column-level 193, 194, 195
creating column statistics 194
datachange function 188
deleting table and column with **delete statistics** 200
drop index and 193
getting additional 195
locking 198
sampling 185
scan types 198
sort requirements 198
sorts for unindexed columns 198
truncate table and 193
update statistics 184
update statistics automatically 190
updating 183, 194
using 181
using job scheduler 190
statistics clause, **create index** command 193
statisticsmaintenance 193
statisticssorts, non leading columns 198
store
operator 146
subqueries 82
symbols
in SQL statements xii
syntactic qualification
abstract plans 178
syntax
abstract plans 172
syntax conventions, Transact-SQL xii
syntax, new
abstract plans 175
sysquerymetrics view
query processing metrics 167

T

table scan
hash based 50
parallel 49
parallelism 48
partition based 50
serial 48
techniques
optimization 5
text delete
operator 115
timeout
limit, abstract plans 175
transformation
predicate 10
transformations
query optimization 8
transitive closure
equi-join 9
search arguments 8
truncate table command
column-level statistics and 193
tuning
according to object size 20
two phased scalar aggregation 58

U

unindexed columns 183
union all
operator 141
parallel 60
serial 61
update 114
update 88
update all statistics 194
update all statistics command 196
update index statistics 194, 196, 199
update statistics 184
update statistics command
column-level 195
column-level statistics 195
managing statistics and 193
with consumers clause 199
updating
statistics 183, 186, 194
updating statistics
use sampling 186
using
query processing metrics 166
showplan 94

V

variables
set local 39
vector aggregate operator 135
vector aggregation 73
distinct 77
in-partitioned 73
re-partitioned 74
serial 76
two phased 75
view
sysquerymetrics, query processing metrics 167

W

with statistics clause, **create index** command 193
worker process mode
exchange 44
worker processes
setting number 33
worktables
abstract plans 177

X

XML
diagnostic output 158
discontinued trace commands 163
permissions 163
set 158