HP NonStop SQL/MX Release 3.1 Query Guide

Abstract

This guide describes how to understand query execution plans and write optimal queries for an HP NonStop™ SQL/MX database. It is intended for database administrators and application developers who use NonStop SQL/MX to query an SQL/MX database and who have a particular interest in issues related to query performance.

Product Version

NonStop SQL/MX Releases 3.1

Supported Release Version Updates (RVUs)

This publication supports J06.12 and all subsequent J-series RVUs and H06.23 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
663851-001	October 2011

Document History

Part Number	Product Version	Published
640323-001	NonStop SQL/MX Release 3.0	February 2011
663851-001	NonStop SQL/MX Release 3.1	October 2011

Legal Notices

Printed in the US

HP NonStop SQL/MX Release 3.1 Query Guide

Index | Examples | Figures | Tables

Tables

What’s New in This Manual

Manual Information

Abstract

This guide describes how to understand query execution plans and write optimal queries for an HP NonStop™ SQL/MX database. It is intended for database administrators and application developers who use NonStop SQL/MX to query an SQL/MX database and who have a particular interest in issues related to query performance.

Product Version

NonStop SQL/MX Releases 3.1

Supported Release Version Updates (RVUs)

This publication supports J06.12 and all subsequent J-series RVUs and H06.23 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
663851-001	October 2011

Document History

Part Number	Product Version	Published
640323-001	NonStop SQL/MX Release 3.0	February 2011
663851-001	NonStop SQL/MX Release 3.1	October 2011

New and Changed Information

Changes to the SQL/MX 3.1 manual:

- Added ESP_ACCESS Operator on page 7-11
- Added NEXTVALUEFOR Operator on page 7-50
- Added SEQUENCEGENERATOR Operator on page 7-65

Changes to the H06.22/J06.11 manual:

- Updated a note in the Using Constraint Based Query Pruning on page 1-16.
- Updated SQL/MX Objects in Table 3-1, Histogram Temporary Tables, on page 3-2.
- Replaced schema version 1200 with 3000 in 3, Keeping Statistics Current, on page 3-1.

- Updated Querying the Query Plan Caching Virtual Tables on page 6-13.
- Added PROBE_CACHE Operator on page 7-59.
- Updated Optimization Tips on page 4-9.

Changes to the H06.20/J06.09 manual:

- Updated the DISPLAY_EXPLAIN function with EXPLAIN statement on pages 1-8, 1-9, 1-11, 1-17, 1-18, 4-1, 4-2, 4-5, 4-6, 4-8, 5-2, 5-8, 8-17, 8-18, and 8-22.
- Updated 7, SQL/MX Operators to add new tokens to the existing operators and to provide an example for each operator.
- Added the following operators:
    - BLOCKED_UNION Operator on page 7-5
    - EXPLAIN_CMD Operator on page 7-17
    - FirstN Operator on page 7-22
    - ORDERED_UNION Operator on page 7-54
    - SAMPLE_FILE_SCAN Operator on page 7-62
    - UNARY_UNION Operator on page 7-81

Changes to the H06.19/J06.08 Manual

- Added Order of Evaluation of Predicates on page 1-14.
- Added MultiUnion Support on page 1-16.
- Added Using Constraint Based Query Pruning on page 1-16.
- Updated the value of OPTS_PUSH_DOWN_DAM attribute on page 4-12.
- Added MultiUnion Operator on page 7-46.

Changes to the H06.16/J06.05 Manual

- References to Release Version Updates (RVUs) throughout this manual have been updated to include references to J-series RVUs, where appropriate.
- Updated the description of Sampling and UPDATE STATISTICS on page 3-6.
- Added the token, max_results, under User-Defined Routine (UDR) on page 7-7.

Changes to the H06.05 Manual

Section	New or Changed Information
Section 2, Accessing SQL/MX Data	Example added that shows placing statements for a forced shape into a separate module.
Section 3, Keeping Statistics Current	Added information about HIST_SCRATCH_VOL.
Section 4, Reviewing Query Execution Plans	Clarified information about buffer size settings for PARTITION_ACCESS operators.
Section 7, SQL/MX Operators	Removed examples and corrected token information.

About This Manual

This guide describes the use and formulation of queries, how to understand query execution plans, and how to affect the performance of NonStop SQL/MX databases. It also provides information about:

- How to improve query performance
- How the SQL/MX optimizer chooses a query execution plan
- How you can influence the optimizer's choice of a plan

Audience

This guide is intended for database administrators and application programmers who use NonStop SQL/MX to query an SQL/MX database and who have a particular interest in issues related to query performance. Readers are expected to be data processing professionals who have familiarity with general issues related to the performance of database management systems and who understand relational database theory and terminology.

You should also be familiar with the operating system and one of the host programming languages, such as C/C++ or COBOL.

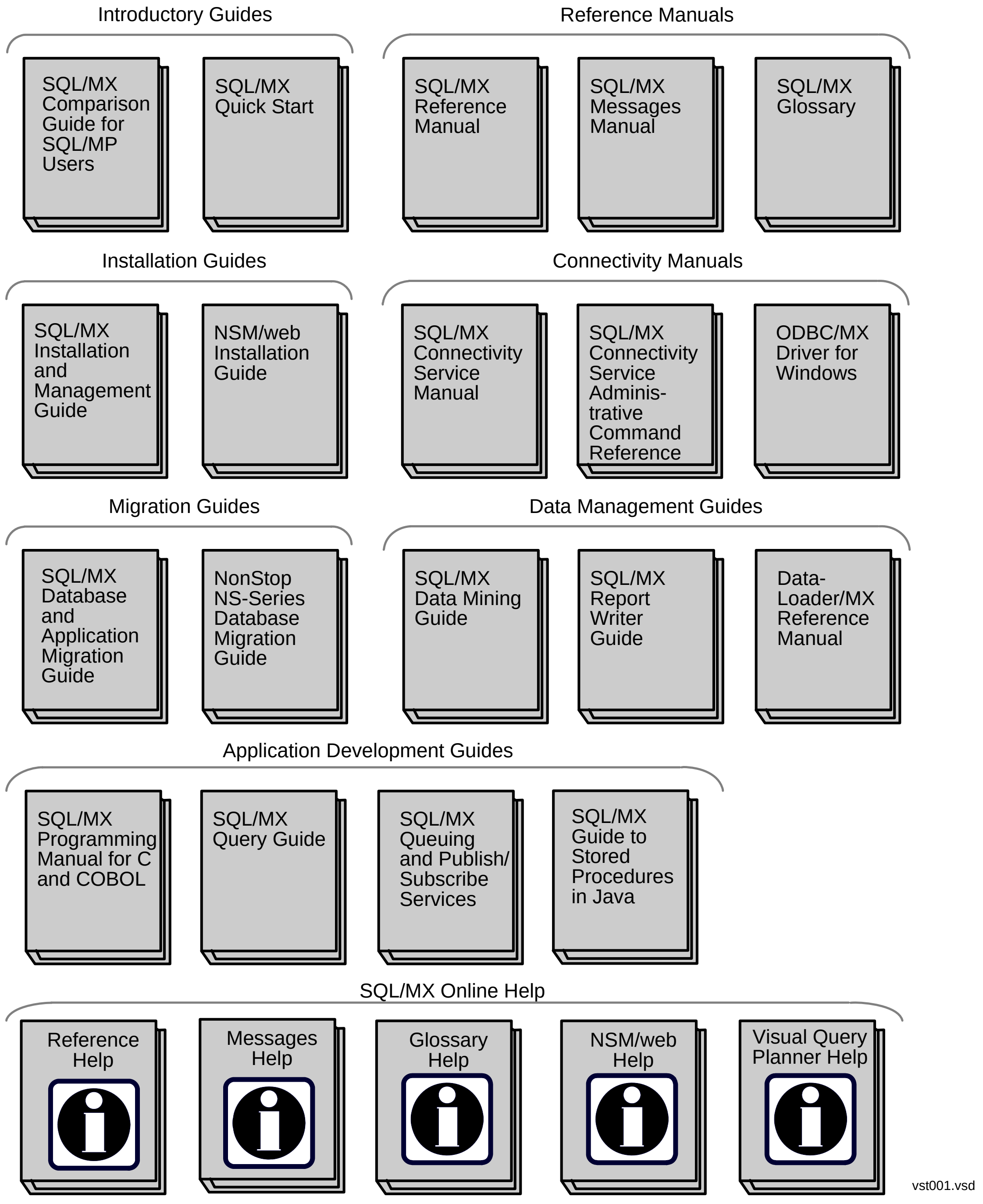
Organization

Section 1, Compiling and Executing a Query	Introduces the NonStop SQL/MX compiler and executor.
Section 2, Accessing SQL/MX Data	Provides information about access methods.
Section 3, Keeping Statistics Current	Provides information about updating statistics and histogram statistics.
Section 4, Reviewing Query Execution Plans	Describes how to read query execution plans.
Section 5, Forcing Execution Plans	Describes how to force query execution plans.
Section 6, Query Plan Caching	Describes how query plan caching works and how to view query plan caching statistics.
Section 7, SQL/MX Operators	Provides information about operators used in query execution plans.
Section 8, Parallelism	Describes how NonStop SQL/MX performs parallelism.

Examples are shown in interactive form, such as that used by the SQL/MX conversational interface (MXCI) and Visual Query Planner (VQP).

Related Documentation

This manual is part of the HP NonStop SQL/MX library of manuals, which includes:


Introductory Guides
SQL/MX Comparison Guide for SQL/MP Users
SQL/MX Quick Start
Reference Manuals
SQL/MX Reference Manual
SQL/MX Messages Manual
SQL/MX Glossary
Installation Guides
SQL/MX Installation and Management Guide
NSM/web Installation Guide
Connectivity Manuals
SQL/MX Connectivity Service Manual
SQL/MX Connectivity Service Adminis-trative Command Reference
ODBC/MX Driver for Windows
Migration Guides
SQL/MX Database and Application Migration Guide
NonStop NS-Series Database Migration Guide
Data Management Guides
SQL/MX Data Mining Guide
SQL/MX Report Writer Guide
Data-Loader/MX Reference Manual
Application Development Guides
SQL/MX Programming Manual for C and COBOL
SQL/MX Query Guide
SQL/MX Queuing and Publish/Subscribe Services
SQL/MX Guide to Stored Procedures in Java
SQL/MX Online Help
Reference Help
Messages Help
Glossary Help
NSM/web Help
Visual Query Planner Help
vst001.vsd

Introductory Guides

SQL/MX Comparison Guide for SQL/MP Users	Describes SQL differences between NonStop SQL/MP and NonStop SQL/MX.
SQL/MX Quick Start	Describes basic techniques for using SQL in the SQL/MX conversational interface (MXCI). Includes information about installing the sample database.

Reference Manuals

SQL/MX Reference Manual	Describes the syntax of SQL/MX statements, MXCI commands, functions, and other SQL/MX language elements.
SQL/MX Messages Manual	Describes SQL/MX messages.
SQL/MX Glossary	Defines SQL/MX terminology.

Installation Guides

SQL/MX Installation and Management Guide	Describes how to plan for, install, create, and manage an SQL/MX database. Explains how to use installation and management commands and utilities.
NSM/web Installation Guide	Describes how to install NSM/web and troubleshoot NSM/web installations.

Connectivity Manuals

SQL/MX Connectivity Service Manual	Describes how to install and manage the HP NonStop SQL/MX Connectivity Service (MXCS), which enables applications developed for the Microsoft Open Database Connectivity (ODBC) application programming interface (API) and other connectivity APIs to use NonStop SQL/MX.
SQL/MX Connectivity Service Administrative Command Reference	Describes the SQL/MX administrative command library (MACL) available with the SQL/MX conversational interface (MXCI).
ODBC/MX Driver for Windows	Describes how to install and configure HP NonStop ODBC/MX for Microsoft Windows, which enables applications developed for the ODBC API to use NonStop SQL/MX.

Migration Guides

SQL/MX Database and Application Migration Guide	Describes how to migrate databases and applications to NonStop SQL/MX and how to manage different versions of NonStop SQL/MX.
NonStop NS-Series Database Migration Guide	Describes how to migrate NonStop SQL/MX, NonStop SQL/MP, and Enscribe databases and applications to HP Integrity NonStop NS-series systems.

Data Management Guides

SQL/MX Data Mining Guide	Describes the SQL/MX data structures and operations to carry out the knowledge-discovery process.
SQL/MX Report Writer Guide	Describes how to produce formatted reports using data from an SQL/MX database.
DataLoader/MX Reference Manual	Describes the features and functions of the DataLoader/MX product, a tool to load SQL/MX databases.

Application Development Guides

SQL/MX Programming Manual for C and COBOL	Describes how to embed SQL/MX statements in ANSI C and COBOL programs.
SQL/MX Query Guide	Describes how to understand query execution plans and write optimal queries for an SQL/MX database.
SQL/MX Queuing and Publish/Subscribe Services	Describes how NonStop SQL/MX integrates transactional queuing and publish/subscribe services into its database infrastructure.
SQL/MX Guide to Stored Procedures in Java	Describes how to use stored procedures that are written in Java within NonStop SQL/MX.

Online Help

Reference Help	Overview and reference entries from the *SQL/MX Reference Manual*.
Messages Help	Individual messages grouped by source from the *SQL/MX Messages Manual.*
Glossary Help	Terms and definitions from the *SQL/MX Glossary*.
NSM/web Help	Context-sensitive help topics that describe how to use the NSM/web management tool.
Visual Query Planner Help	Context-sensitive help topics that describe how to use the Visual Query Planner graphical user interface.

The NSM/web and Visual Query Planner help systems are accessible from their respective applications. You can download the Reference, Messages, and Glossary online help from the $SYSTEM.ZMXHELP subvolume or from the HP NonStop Technical Library (NTL). For more information about downloading online help, see the *SQL/MX Installation and Management Guide*.

These manuals are part of the SQL/MP library of manuals and are essential references for information about SQL/MP Data Definition Language (DDL) and SQL/MP installation and management:

Related SQL/MP Manuals

SQL/MP Reference Manual	Describes the SQL/MP language elements, expressions, predicates, functions, and statements.
SQL/MP Installation and Management Guide	Describes how to plan, install, create, and manage an SQL/MP database. Describes installation and management commands and SQL/MP catalogs and files.

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

The following list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

`file-name`

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

`pathname`

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name
```

```
INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]
```

```
K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }
```

```
ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;
```

```
LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

1 Compiling and Executing a Query

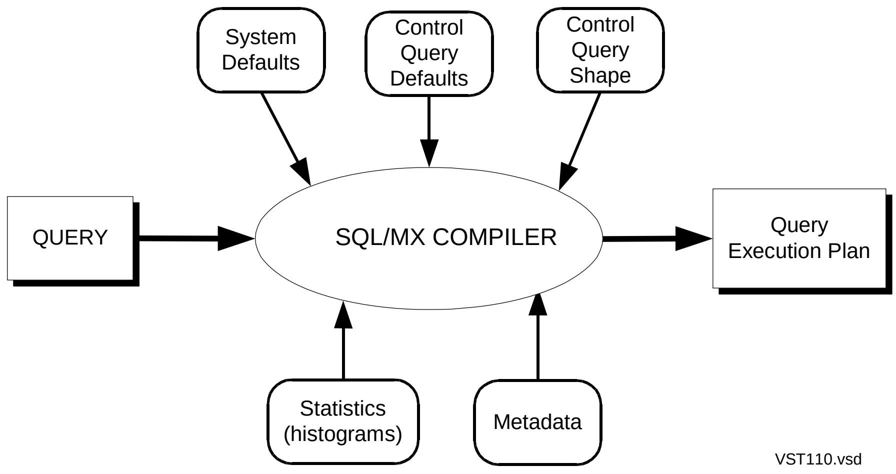
Use the information in this section to understand how your queries are compiled and executed.

- How the Compiler Works on page 1-2
- How the Executor Processes the Plan on page 1-19

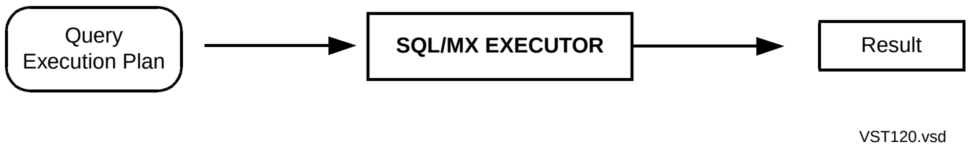
Overview

At the highest level, compiling and executing an SQL query with NonStop SQL/MX consists of two basic steps.

1. The SQL/MX compiler processes the query and produces a query execution plan.



2. The SQL executor processes the execution plan and produces the query result.



These actions can influence query optimization, as shown in Step 1:

- Forcing plans with the CONTROL QUERY SHAPE statement
- Changing default settings with the CONTROL QUERY DEFAULTS statement
- Keeping statistics current with the UPDATE STATISTICS statement
- Changing system default settings in the SYSTEM_DEFAULTS table

- Modifying the database (metadata) by adding columns to a table, splitting or merging partitions, and creating and dropping indexes

For more information, see Improving Query Performance on page 1-6.

How the Compiler Works

To compile a query, the SQL/MX compiler needs information about the tables listed in the query and the query environment. The necessary table information is listed in the schema metadata tables from the SQL/MX catalog. The query environment information can be influenced by the user in the SYSTEM_DEFAULTS table. For more information about the SYSTEM_DEFAULTS table, see the *SQL/MX Reference Manual*.

Compilation Steps

The SQL/MX compiler goes through several steps to compile a query. The corresponding compiler component performs these steps in this way:

Parser	Performs syntax checks and translates the SQL/MX query into a syntactically correct query tree.
Binder	Takes the syntactically correct query tree and translates logical (ANSI) names to physical names and performs many semantic checks. The binder also expands views that are listed in the query, looks up metadata for table information, and produces a semantically correct query tree.
Normalizer	Takes the semantically correct query tree and performs certain unconditional transformations, such as constant folding, subquery elimination, and recognizes equivalent expressions by representing equivalent groups of values. These transformations make the query representation suitable for optimization. The transformations are not based on cost. The normalizer produces the normalized query tree as input to the optimizer (semantically correct query tree in canonical form).
Optimizer	Takes the normalized query tree and generates cost-based, rules-driven alternative plans to choose the best execution plan for the query. The optimizer computes the cost of each alternative and chooses the alternative with the lowest cost as the optimal query execution plan.
Codegen	Takes the optimal query execution plan and translates it to executable code for the executor (see How the Executor Processes the Plan on page 1-19).

Internally, the query is represented as an operator query tree. Each stage adds information and might modify the input tree. The tree produced in the last step is called the query execution plan.

Parsing, Binding, and Normalizing

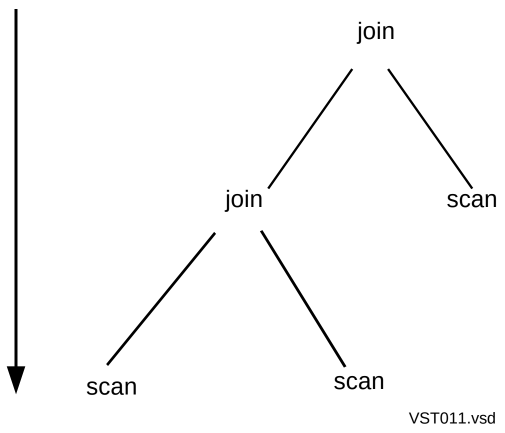
The initial steps in the compile process—parsing, binding, and normalizing—prepare the query for the optimizer. Before query optimization begins, the query tree that is produced by the parser is bound with information from the metadata. Certain query processing instructions in addition to default values needed for optimizing the query are processed. If you have made changes to the default settings, the parser reads the adjusted values from the SYSTEM_DEFAULTS table. All subqueries are eliminated or transformed into joins or semi-joins. In addition, certain predicates are rewritten and pushed down for earliest possible evaluation in the query tree. If the predicates are pushed all the way down to the scan level, they are executed in HP NonStop Data Access Manager (DAM). If the predicates are pushed down to a join, they are executed in the master process or one or more ESP processes. The binding and normalization phases result in a normalized query tree that is provided as input to the optimizer.

Optimizing Queries

NonStop SQL/MX uses a unique combination of optimization strategies to build an optimal query plan:

- Branch and Bound Programming

  This technique consists of a top-down approach that starts with an query tree of logical, relational expressions. Logical expressions contain relational operators that do not denote an implementation. Examples are join, group by, and scan. Using optimization rules, the optimizer makes a decision about the execution method of the top operator in the tree. The optimizer generates optimal solutions for the descendants of the top operator and combines those solutions with the execution method (physical operator) of the top node. Physical operators are relational operators that specify the actual implementation or run-time algorithm. Examples are merge join, hash group by, and file scan.



  The first feasible solution that is found will from then on form an upper cost bound. Any solution that exceeds the cost bound is discarded immediately, and the optimizer remembers not to pursue that solution further. Alternative solutions found by the optimizer go into the search space, where they are stored in search memory.

- Principle of Optimality

  In this strategy, the optimizer starts at the top of the normalized query tree and breaks the optimization tasks into smaller tasks (or subtasks). Depending on the physical properties of the operators, the optimizer combines the optimal solutions of the subtasks for the completed plan.

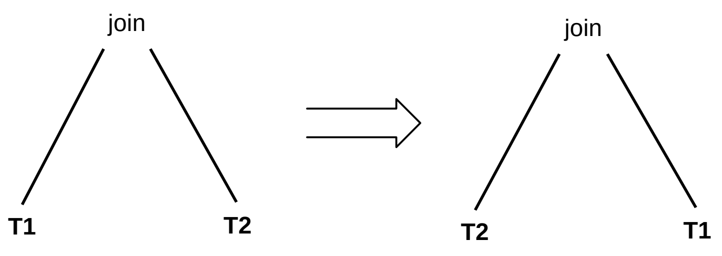
- Avoiding Search Space Overload

  The search space contains all possible plans that the optimizer considers. Because the optimizer tries for as many alternatives as possible per node in the query tree, a risk exists that you can inundate the search space with all the possibilities. Equivalent expressions are organized into groups. Two expressions are in the same group if one can be derived from the other by the application of rules. The organization of the search memory reduces the search space dramatically. To influence the amount of optimization that the optimizer performs in trying to find the best plan, you can set the OPTIMIZATION_LEVEL attribute. For more information about setting the OPTIMIZATION_LEVEL attribute, see Section 4, Reviewing Query Execution Plans.

- Rule-Based Solutions

  In addition to reviewing costs associated with operations, the optimizer also uses rules. Optimization rules define how a query tree or one of its subtrees can be transformed into a semantically equivalent query tree. These rules separate the logical and physical expressions, as follows:

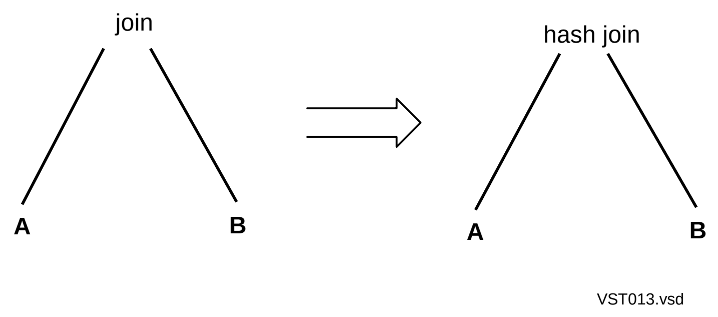
  - **What the optimizer does to find a solution**. Transformation rules transform one logical expression into a different, semantically equivalent logical expression. Logical expressions do not have a cost associated with them and do not have all their physical properties specified; that is, instead of using the query tree suggestions, the optimizer creates another tree. For example, the join of table T1 and table T2 should yield an equivalent logical expression when transformed to a join of table T2 and table T1.



VST012.vsd

- ○ **How the optimizer finds the solution**. Implementation rules transform a logical expression into a semantically equivalent expression whose root node is a physical operator. Recursive application of implementation rules result in a query execution plan that consists only of physical operators. Such plans or subplans can be executed by the executor because they have physical properties, and their cost can be estimated by the optimizer. The implementation rule phase refines the earlier rules that result in physical nodes. For more information about logical and physical operators, see Section 5, Forcing Execution Plans.

  Implementation rules reduce a problem into one or more small subproblems by making a plan decision. This rule follows the Principle of Optimality, which states: “Combine optimal partial solutions to form an optimal solution to the complete problem.” The next figure shows an implementation rule that transforms a join of A and B into a hash join of A and B. The principle of optimality states that the optimal subplans for A and B remain the same in the plan that uses a hash join.



- Multipass Optimization

  The optimizer uses cost-based pruning by supporting a multipass optimization technique. A pilot phase during preoptimization orders tables in ascending order, based on estimated row count; that is, larger tables are scanned last. During the first pass, only those rules that are necessary to generate a feasible plan (with a reasonable cost) are enabled. Subsequent passes can then use the costs generated in previous passes as an upper cost bound, allowing for more cost-based pruning. This strategy has the effect of generating an optimal plan while exploring a smaller search space and, as a result, reducing compile time.

  The multipass optimization also enables error recovery. If an error occurs during the second or subsequent pass, the plan generated from the first pass is returned.

Query Plan Caching

The SQL/MX compiler provides the ability to cache the plans of certain queries. Use query plan caching in an environment where similar queries are generated, compiled, executed, and SQL compilation time is significant compared to SQL execution time.

This feature improves the performance of the compiler when the plan can be produced from cache rather than through a full compilation. The queries that can be considered for plan caching include simple TP-style inserts, updates, deletes, selects, and joins. Two queries are considered equivalent for the purposes of caching if their canonical forms are the same. For query caching, the canonical form of a query is constructed by:

- Removing unmeaningful white space differences
- Removing unmeaningful case differences
- Expanding '*' notation in select lists
- Resolving all object names to fully qualified names
- Replacing most constant literals with parameters
- Encoding all CONTROL QUERY DEFAULT and CONTROL TABLE statements that have been previously executed in the current SQL/MX compiler session

When query caching is enabled, NonStop SQL/MX caches the compiled plans of cacheable statements. When an equivalent query is resubmitted, most of the SQL compilation is skipped, and the query plan is produced from the cache. Use query plan caching statistics to determine important information about the caching process in addition to the current state of stored plans.

Section 6, Query Plan Caching describes the types of cacheable queries, query plan caching statistics, and the default settings that affect query caching.

Improving Query Performance

The SQL/MX compiler performs a series of internal transformations when generating the most suitable plan for a query. Some of these transformations are described next. The actions you can take to influence query performance are described under Influencing Query Performance on page 1-7.

Note. The SQL/MX compiler has been enhanced to improve the quality of query plans for complex queries that include multitable joins of up to 12 tables.

Query Processing by the Compiler

The query tree obtained by parsing the query goes through numerous transformations. While the binder and codegen perform relatively simple changes, more complex transformations are performed in the normalizer and the optimizer stages.

In the normalizer, unconditional transformations are applied to make the query tree more suitable for subsequent optimization. The major transformations are:

- Unconditional predicate transformations
- Subquery transformation to joins
- Predicates pushed down as far as possible

- The computation of the transitive closure of “=” predicates and rewrite of predicate factors based on it
- Canonical reorder of the tables in the query tree
- Constant folding. Constant folding is a compiler optimization technique where an expression, consisting only of constants, is evaluated at compile time. For example:

  ```
   WHERE AGE_IN_DAYS>(2005-1960)*365
  ```

  When constant folded, this expression becomes `WHERE AGE_IN_DAYS>16425.`
- Syntactic and semantic sort elimination

In the optimizer, transformations are conditional and performed based on cost. They are also applied by rules. See Rule-Based Solutions on page 1-4.

A range of options exists for improving query performance. Some involve simple changes to the system defaults and others are more complex and involve investigating the query plan and making appropriate changes to the query or query plan.

Influencing Query Performance

You can influence query performance in online transaction processing environments through one or more of these methods:

- Augment or modify the database by adding indexes.
- Refine table statistics by adding statistics for all columns used in the query. Ensure that statistics have been generated for all columns used in the query. For more information, see Section 3, Keeping Statistics Current.
- Use `FIRST 1` syntax for nonunique `SELECT ... INTO` queries:

  ```
  SELECT [FIRST 1] a INTO :hv ...
  ```
- Use read-only SELECT statements whenever possible:

  ```
  CONTROL QUERY DEFAULT readonly_cursor 'TRUE'
  ```

To ensure that the MXCMP costing model works correctly, first perform a FUP RELOAD and then UPDATE STATISTICS on tables and indexes. For more information about using FUP RELOAD, see the *SQL/MX Installation and Management Guide.* For more information about using the UPDATE STATISTICS statement, see the *SQL/MX Reference Manual.*

- Force plans through the CONTROL QUERY SHAPE utility or through CONTROL QUERY DEFAULT JOIN_ORDER_BY_USER. For more information, see Section 5, Forcing Execution Plans.
- If possible, use host variable types that match the database column type. In all three of these examples, *`col`* is the exact same data type as *`hvar`*:

- `WHERE` predicate match:

  ```
  SELECT a FROM t WHERE col = :hvar
  ```

- Target and source `INSERT` match:

  ```
  INSERT INTO t(col) VALUES (:hvar)
  ```

- `UPDATE` match:

  ```
  UPDATE t SET col = :hvar ...
  ```

- Set different default values. Default values can affect both compile and run time. Performance-related default values include OPTIMIZATION_LEVEL, which indicates the effort the optimizer should use in optimizing queries. Others include default settings related to parallelism. For information about default settings, see the *SQL/MX Reference Manual*.
- Adding or dropping partitions can influence whether you obtain a parallel plan. For more information about parallel plans, see Section 8, Parallelism.
- If you use a key column in an expression like `WHERE A*3=10` or `WHERE SIN(A)=20`, or `UPSHIFT(key col)`, or another similar expression that requires computation or a function like UPSHIFT, a full table scan is used, and key access is not used. As a result, query performance can be degraded. Consider rewriting the expression in the form `col=constant`, for example, `WHERE A=10/3`.
- Whenever possible, formulate your queries to use multiplication instead of division. For example, this query that uses division can be reformulated to use multiplication:

  ```
  --DIVISION
  SELECT A/10 FROM T WHERE B > C/100

  --REFORMULATED TO MULTIPLICATION
  SELECT A*0.1 FROM T WHERE B > C*0.01
  ```

- When questioning “bad plans,” or query plans that do not seem optimal, question the inputs to the compiler before you question the compiler.

Enabling Online Transaction (OLT) Optimization

If your OLTP application is having performance problems, verify that the token olt_optimization is enabled for single-row access type queries. The token is present at three levels: in the ROOT operator, PARTITION_ACCESS operator, and DAM operators (INDEX_SCAN_UNIQUE, INSERT, FILE_SCAN_UNIQUE, UNIQUE_DELETE, and UNIQUE_UPDATE).

The CONTROL QUERY DEFAULT to enable OLT optimization is OLT_QUERY_OPT and is on by default. To check if OLT optimization is actually being used, look for olt_optimization: USED in the EXPLAIN statement output.

You can also use the EXPLAIN statement OPTIONS 'f' command to display operators in the query. Check the OPT column for letter o entries, as the following table shows, to determine whether OLT optimization is enabled.

```
LC  RC  OP   OPERATOR          OPT     DESCRIPTION       CARD
--  --  --   ---------         -----   ------------      -------
2   .   3    root              o       r                 1.00E+0
1   .   2    partition_access  o                         1.00E+0
.   .   1    insert            o       T                 1.00E+0
```

Note. For more information on how to use the EXPLAIN statement OPTIONS 'f' command, see the *SQL/MX Reference Manual.*

To determine if your query uses OLT optimization, check the query type and operators listed in Table 1-1.

Table 1-1. Valid Queries and Operators for OLT Optimization

Query	ROOT Operator	PARTITION_ACCESS Operator	DAM Operators
Embedded unique UPDATE	Yes	Yes	Yes
Embedded unique DELETE	Yes	Yes	Yes
Embedded unique INSERT	Yes	Yes	Yes
Embedded unique SELECT ... into	Yes	Yes	Yes
Embedded nonunique UPDATE	Yes	Yes	No
Embedded nonunique DELETE	Yes	Yes	No
Embedded INSERT with value list	Yes	No	Yes
Embedded nonunique SELECT ... into	Yes	No	No
Dynamic MXCI & ODBC queries	No	Same considerations as Embedded Queries	Same considerations as Embedded Queries
Compound Statements	Yes	Only if forced to use DAM	Yes
Rowsets	No	No	Unique access only

If your query is using OLT optimization and performance is still an issue, review the database or query for design choices that can affect performance. The *SQL/MX*

Programming Manual for C and COBOL, *SQL/MX Programming Manual for Java,* and the *SQL/MX Installation and Management Guide* have tips and guidelines to help identify potential performance problem areas.

Using OR Operators in Predicates

For a narrow subset of queries with OR operations, NonStop SQL/MX uses a feature called OR optimization. OR optimization uses more than one access path to obtain the data and eliminates duplicate predicates to produce the result. In many cases, OR optimization results in the query running significantly faster. While partitioning is not required for OR optimization benefits to take effect, performance benefits might be further enhanced with partitioned tables.

Note. OR optimization in NonStop SQL/MX has a more restrictive set of conditions than its NonStop SQL/MP counterpart.

- OR optimization and application example

  OR optimization might be useful in querying large transaction tables where the expected output is only a few rows.

- MDAM and OR optimization

  MDAM is another feature that works with OR operators. It provides some benefits that the OR optimization feature does not:

  - MDAM eliminates duplicate key values in OR predicates at run time. It does so before NonStop SQL/MX accesses any tables, so there is no performance penalty.
  - MDAM can process multiple tables in a query. It can be used on the inner and outer tables of nested joins and on the outer tables of sort merge, hash, and key-sequenced merge joins.
  - WHERE clauses need not be in disjunctive normal form.

  Note. A search condition in disjunctive normal form has only one level of OR operations; for example, (P1 and P2) or (P4 and P5) or (P6 and P7 and P8). A search condition not in normal disjunctive form can be transformed into one. For example, (A or B) and C can be changed to (A and C) or (B and C). A search condition in conjunctive normal form has one level of AND operations; for example, P1 and P2 and P3.

  MDAM is enabled by default.

- Choosing optimized OR plans

  SQL might use an optimized OR plan when all these conditions are satisfied:

  - Two or more search conditions connected by OR operators.
  - Each search condition contains predicates used as keys on an index. That is, the predicates involve columns that belong to the key prefix of an index.

- ○ The search condition is restricted to simple OR predicates:

  ```
  L_SUPPKEY = 2407 OR L_PARTKEY > 34567
  ```

Note. For execution plans that use OR optimization, the optimizer considers index-only access in addition to index-key lookup through a nested join. Index-only access, however, can be used only if the index contains all the columns referenced in the query.

This query uses the DDL shown in Example 1-1 on page 1-13. The EXPLAIN statement OPTIONS 'f' output is shown in Figure 1-1 below.

```
PREPARE XXZ FROM SELECT * FROM LINEITEM
WHERE L_SUPPKEY = 2407 OR L_PARTKEY = 34567;
```

Figure 1-1. EXPLAIN statement OPTIONS 'f' Output for Nested Join With Indexes LX1 and LX2 From LINEITEM Table

```
LC   RC   OP   OPERATOR         OPT  DESCRIPTION        CARD
---  ---  ---  --------         ---  -----------        ----

15   .    16   root                                     4.30E+1
7    14   15   merge_union                              1.70E+1
10   13   14   nested_join                              1.40E+1
12   .    13   split_top             1:4(logph)         3.49E-1
11   .    12   partition_access                         3.49E-1
.    .    11   file_scan_unique  fr ORCAT.LINEITEM(s) 3.49E-1
9    .    10   split_top             1:4(logph)         4.00E+1
8    .    9    partition_access                         4.00E+1
.    .    8    index_scan     fs fr ORCAT.LX2(s)        4.00E+1
3    6    7    nested_join                              3.00E+0
5    .    6    split_top             1:4(logph)         1.00E+0
4    .    5    partition_access                         1.00E+0
.    .    4    file_scan_unique  fr ORCAT.LINEITEM(s) 1.00E+0
2    .    3    split_top             1:4(logph)         3.00E+0
1    .    2    partition_access                         3.00E+0
.    .    1    index_scan     fs fr ORCAT.LX1(s)        3.00E+0
```

Additional indexes might enable OR optimization for additional columns.

- Plans that do not use OR optimization

  In general, OR optimization is not used when:

  - ○ A query involves more than one table.
  - ○ Columns in the predicate are not part of a key prefix.
  - ○ The search condition includes an AND operator.
  - ○ At least one single predicate—or set of predicates connected by AND operators—contains an executor predicate, which are evaluated for each row.

- Examples of OR optimization

For the DDL to these examples, see the LINEITEM table in Example 1-1 on page 1-13. The primary key is L_ORDERKEY + L_LINENUMBER, and there are indexes on L_PARTKEY (LX1) and L_SUPPKEY (LX2).

If a query on table LINEITEM contains these predicates, OR optimization might be performed:

```
WHERE (L_PARTKEY = 200001 OR L_SUPPKEY = 10)
```

OR optimization would use index access on LX1 for the first predicate and an index access through LX2 for the second predicate. In the absence of an index, SQL reads the table sequentially to search for rows that satisfy the query.

Another option might be between using a single index versus using MDAM. Table LINEITEM could have three indexes on columns (L_PARTKEY, L_SUPPKEY), (L_SUPPKEY), (L_LINESTATUS). The query can either use all three indexes, or it can use the index (L_PARTKEY, L_SUPPKEY) with MDAM for the first two disjuncts and an index L_LINESTATUS for the last disjunct.

```
SELECT * FROM LINEITEM
WHERE L_PARTKEY=10999765 OR L_SUPPKEY=19 OR L_LINESTATUS=30
```

- Examples that do not enable OR optimization
  - Indexes must exist for OR optimization to be considered. If the table, LINEITEM, has secondary indexes on columns L_PARTKEY and L_SUPPKEY, OR optimization is not considered for this query because there is no index on column L_LINESTATUS:

    ```
    SELECT * FROM LINEITEM
    WHERE L_PARTKEY=1099987 OR L_SUPPKEY=20 OR
    L_LINESTATUS='S'
    ```

  - A combination of conjuncts and disjuncts does not enable OR optimization:

    ```
    SELECT * FROM LINEITEM
    WHERE L_PARTKEY=10998765 OR L_SUPPKEY=20
    AND L_LINESTATUS='Y'
    ```

  - Even if the indexes L_PARTKEY, L_SHIPINSTRUCT and L_SUPPKEY, L_SHIPMODE exist on table LINEITEM, NonStop SQL/MX, unlike NonStop SQL/MP, does not consider OR optimization for these types of queries:

    ```
    SELECT * FROM LINEITEM
    WHERE(L_PARTKEY=10997965 AND L_SHIPINSTRUCT='SAIL')
    OR (L_SUPPKEY=20 AND L_SHIPMODE ='ANCH')
    ```

Example 1-1. OR Optimization DDL

```
CREATE TABLE LINEITEM
  ( L_ORDERKEY                INT            NOT NULL
  , L_PARTKEY                 INT            NOT NULL
  , L_SUPPKEY                 INT            NOT NULL
  , L_LINENUMBER              INT            NOT NULL
  , L_QUANTITY                NUMERIC(12,2)  NOT NULL
  , L_EXTENDEDPRICE           NUMERIC(12,2)  NOT NULL
  , L_DISCOUNT                NUMERIC(12,2)  NOT NULL
  , L_TAX                     NUMERIC(12,2)  NOT NULL
  , L_RETURNFLAG              CHAR(1)        NOT NULL
  , L_LINESTATUS              CHAR(1)        NOT NULL
  , L_SHIPDATE                DATE           NOT NULL
  , L_COMMITDATE              DATE           NOT NULL
  , L_RECEIPTDATE             DATE           NOT NULL
  , L_SHIPINSTRUCT            CHAR(25)       NOT NULL
  , L_SHIPMODE                CHAR(10)       NOT NULL
  , L_COMMENT                 VARCHAR(44)    NOT NULL
  , PRIMARY KEY (L_ORDERKEY, L_LINENUMBER))
  STORE BY PRIMARY KEY
  PARTITION (
      ADD FIRST KEY (100001)
      LOCATION $DATA08
      NAME PARTN1
      EXTENT (1024, 1024) MAXEXTENTS 512
     ,ADD FIRST KEY (200001)
      LOCATION $DATA07
      NAME PARTN2
      EXTENT (1024, 1024) MAXEXTENTS 512
     ,ADD FIRST KEY (300001)
      LOCATION $DATA06
      NAME PARTN3
      EXTENT (1024, 1024) MAXEXTENTS 512
    );
CREATE INDEX LX1 ON LINEITEM
  (L_PARTKEY)
  LOCATION $DATA09
  ATTRIBUTE EXTENT (1024, 1024) MAXEXTENTS 512
  PARTITION
    ( ADD FIRST KEY (100001)
      LOCATION $DATA08
     ,ADD FIRST KEY (200001)
      LOCATION $DATA07
     ,ADD FIRST KEY (300001)
      LOCATION $DATA06
     );
CREATE INDEX LX2 ON LINEITEM
  (L_SUPPKEY)
  LOCATION $DATA09
  ATTRIBUTE EXTENT (1024, 1024) MAXEXTENTS 512
  PARTITION
    ( ADD FIRST KEY (2500)
      LOCATION $DATA08
     ,ADD FIRST KEY (5000)
      LOCATION $DATA07
     ,ADD FIRST KEY (7500)
      LOCATION $DATA06
    );
```

Order of Evaluation of Predicates

The SQL/MX compiler can select any order for evaluating the predicates based on the factors such as cost, cardinality, join type, access path, and so on. As a result, SQL/MX might not be able to retain the order in which the predicates appear in a query while executing them. SQL/MX might return different results depending on the order of evaluation. The following example explains how the order of evaluation of predicates can affect the result set.

```
>>CREATE TABLE T_DIVBYZERO (A INT, B INT);
--- SQL operation complete.

>>INSERT INTO T_DIVBYZERO VALUES (1,1),(0,0),(2,2);
--- 3 row(s) inserted.

>>SELECT * FROM T_DIVBYZERO WHERE A <> 0 AND 10/B > 1;
A                B
----------- -----------
1                1
2                2
--- 2 row(s) selected.

>>SELECT * FROM T_DIVBYZERO WHERE 10/B > 1 AND A <> 0;
A                B
----------- -----------
1                1
*** ERROR[8419] An arithmetic expression attempted a
division by zero.
--- 1 row(s) selected.
>>
```

You might notice similar behavior with complex queries, when SQL/MX selects other order for evaluating the predicates than the given order. Another example is given below.

```
>>CREATE TABLE STAFF_UC (
EMPNUM CHAR (3) CHARACTER SET UCS2 NOT NULL UNIQUE,
EMPNAME NCHAR VARYING (20));
--- SQL operation complete.
>>
>>CREATE TABLE WORKS_UC (
EMPNUM CHAR (3) CHARACTER SET UCS2 NOT NULL,
PNUM CHAR (3) CHARACTER SET UCS2 NOT NULL);
--- SQL operation complete.
>>
>>CREATE TABLE PROJ_UC (
PNUM CHAR (3) CHARACTER SET UCS2 NOT NULL UNIQUE,
CITY VARCHAR (15) CHARACTER SET UCS2);
--- SQL operation complete.
>>
```

```
>>INSERT INTO STAFF_UC VALUES (_ucs2'E1', N'Alice');
--- 1 row(s) inserted.

>>INSERT INTO WORKS_UC VALUES (_ucs2'E1', _ucs2'P3');
--- 1 row(s) inserted.

>>INSERT INTO PROJ_UC VALUES (_ucs2'P3', _ucs2'Tampa');
--- 1 row(s) inserted.

>>INSERT INTO PROJ_UC VALUES
(_ucs2 X'4E7A', _ucs2 X'4E2D56FD53174EAC');
--- 1 row(s) inserted.
>>
>>CONTROL QUERY SHAPE HASH_JOIN (CUT, HASH_JOIN (CUT, CUT));
--- SQL operation complete.

>>PREPARE xx FROM
SELECT STAFF_UC.EMPNAME
FROM STAFF_UC
WHERE STAFF_UC.EMPNUM IN
(SELECT WORKS_UC.EMPNUM
FROM WORKS_UC
WHERE WORKS_UC.PNUM IN
(SELECT PROJ_UC.PNUM
FROM PROJ_UC
WHERE
(
PROJ_UC.PNUM < _UCS2 X'00FF'
AND
TRANSLATE (PROJ_UC.CITY USING
UCS2TOISO88591) = 'Tampa'
)
OR
PROJ_UC.CITY = _UCS2 X'5929 6D25'
)
);
--- SQL command prepared.
>>
>>EXECUTE xx;
EMPNAME
-----------------------------------------------------
Alice
--- 1 row(s) selected.
>>
>>CONTROL QUERY SHAPE
MERGE_JOIN (
NESTED_JOIN (CUT, CUT, INDEXJOIN),
NESTED_JOIN (CUT, CUT)
);
--- SQL operation complete.
>>PREPARE yy FROM
SELECT STAFF_UC.EMPNAME
```

```
FROM STAFF_UC
WHERE STAFF_UC.EMPNUM IN
(SELECT WORKS_UC.EMPNUM
FROM WORKS_UC
WHERE WORKS_UC.PNUM IN
(SELECT PROJ_UC.PNUM
FROM PROJ_UC
WHERE
(
PROJ_UC.PNUM < _UCS2 X'00FF'
AND
TRANSLATE (PROJ_UC.CITY USING
UCS2TOISO88591) = 'Tampa'
)
OR
PROJ_UC.CITY = _UCS2 X'5929 6D25'
)
);
--- SQL command prepared.
>>
>>EXECUTE yy;
*** ERROR[8690] An invalid character value encountered in
TRANSLATE function.
--- 0 row(s) selected.
>>
```

Note. You might come across the queries that depend on the data during run time. There is no specific method to avoid this behavior. However, you can try the following workarounds:

1. Rewrite the query to reorder predicates, tables, and so on.
2. Use control query defaults.
3. Use control query shape.

MultiUnion Support

The MultiUnion feature provides enhanced functionality to handle queries that have a large number of table unions. This feature compresses the binary union backbone in such queries into a single n-way “MultiUnion” operator, there by reducing query execution and compilation time. For information on the MultiUnion operator, see MultiUnion Operator on page 7-46.

Note. The MultiUnion feature is supported only on systems running J06.08 and later J-series RVUs and H06.19 and later H-series RVUs.

The MultiUnion feature is controlled by the MULTIUNION CQD. For more information on MULTIUNION CQD, see the *SQL/MX Reference Manual*.

Using Constraint Based Query Pruning

The constraint based query pruning feature enables you to eliminate unwanted table scans when a predicate in that table violates the check constraints defined on the table. This feature enhances query performance by reducing compilation and execution time.

Note. The constraint based query pruning feature is supported only on systems running J06.08 and later J-series RVUs and H06.19 and later H-series RVUs.

During query compilation, NonStop SQL/MX receives the inputs from the constraints that are defined on a table. If the query contains the predicates that fall outside the check constraints on the table, the scan nodes are marked as “not returning any records”. In other words, if the predicates are such that there is no need to scan the table, the scan nodes will be replaced by the VALUES nodes. As a result, the table is not scanned during query execution, thereby improving the query execution.

Example 1-2 shows the output of the EXPLAIN statement for a query when the Control Query Default (CQD), CHECK_CONSTRAINT_PRUNING, is set to ON and OFF.

For information on CHECK_CONSTRAINT_PRUNING CQD, see the *SQL/MX Reference Manual*.

Note. The constraint-based query pruning feature has the following limitations:

1. Only scan node is converted to the VALUES node.
2. The pruning logic is not rolled-up to other nodes.
3. The constraint based query pruning feature includes the following operators that are used in table constraints or selection predicates or both:

   <, >, <=, >=, =, <>, IS NULL, IS NOT NULL
4. The pruning logic does not roll-up for correlated sub queries.
5. The pruning logic does not work for OR predicates. It works only on AND predicates and constraints.

Example 1-2. EXPLAIN statement OPTIONS ‘f’ Output for Query Using CHECK_CONSTRAINT_PRUNING

```
>>showddl t1;
CREATE TABLE CAT.SCH.T1
  (
    COL1                             INT DEFAULT NULL
  , COL2                             INT DEFAULT NULL

  )
  LOCATION \DMR11.$DATA04.ZSDLPGGW.VBNG7V00
  NAME DMR11_DATA04_ZSDLPGGW_VBNG7V00
  ;

ALTER TABLE CAT.SCH.T1
  ADD CONSTRAINT CAT.SCH.CHK1 CHECK (CAT.SCH.T1.COL1 > 100)
DROPPABLE ;

--- SQL operation complete.

>>control query default check_constraint_pruning 'off';

--- SQL operation complete.
>>prepare xx from select * from t1 where col1 = 10;

--- SQL command prepared.
>>explain options 'f' xx;

LC   RC   OP   OPERATOR         OPT      DESCRIPTION  CARD
---  ---  ---  ---------        ------   -----------  -------
2    .    3    root                                   1.00E+0
1    .    2    partition_access                       1.00E+0
.    .    1    file_scan        fs fr    T1 (s)       1.00E+0

--- SQL operation complete.

>>control query default check_constraint_pruning 'on';

--- SQL operation complete.
>>
>>prepare xx from select * from t1 where col1 = 10;

--- SQL command prepared.
>>explain options 'f' xx;

LC   RC   OP   OPERATOR         OPT      DESCRIPTION  CARD
---  ---  ---  ---------        ------   -----------  -------
1    .    2    root                                   1.00E+0
.    .    1    values                                 1.00E+0

--- SQL operation complete.
```

Factors That Can Affect Compile Time

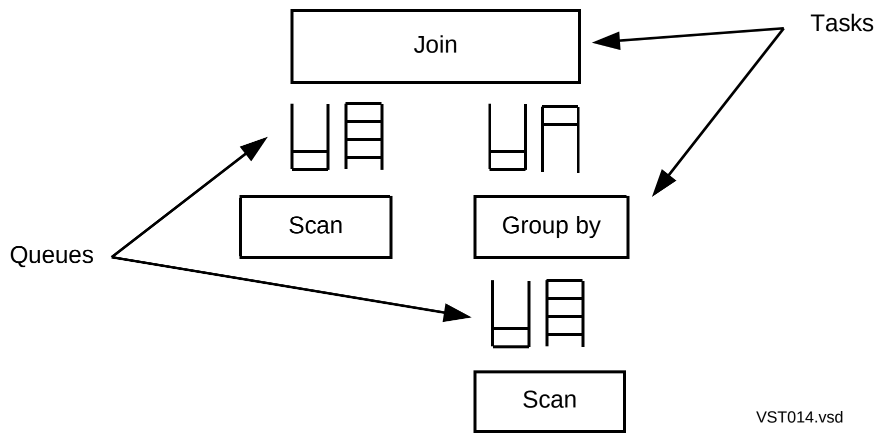
- The number of tables joined, which is the most important factor in determining query complexity
- Other query complexity factors, such as the number of group bys, unions, and subqueries
- The number of predicates on the query and the number of columns in the predicate
- The presence of indexes, which can increase complexity as the new access path must be considered
- Whether statistics have been updated and the number of intervals in the histograms
- Default settings (see Section 4, Reviewing Query Execution Plans), especially the optimization controls such as OPTIMIZATION_LEVEL

Remember that the number of joins is not the only factor that can affect compile time. For example, a 4-way natural join on a 200-column table can take considerably more time than an 8-way join between tables with only a few columns.

How the Executor Processes the Plan

NonStop SQL/MX uses a data-flow and scheduler-driven task model to execute queries. After a query is optimized, the optimizer generates an optimized, executable query plan that goes to the executor. The executor prepares a node for each operator.

Each operator is an independent task and data flows between operators through in-memory queues (up and down) or by interprocess communication. The queue pair operates between two operators. Queues between tasks allow operators to exchange multiple requests or result rows at a time. A scheduler coordinates the execution of tasks and runs whenever it has data on one of its input queues.



The task model makes it easy to perform all internal operations asynchronously so that a single server thread can have multiple I/Os outstanding. This model also provides parallelism for both shared memory and distributed memory architectures. In-memory queues are used for communication, and exchange operators are used for distributed memory.

Concurrent or overlapping work can be performed on rows as they flow through the different stages of the execution plan. For example, while the left child of a nested-loop join is working on producing more rows or waiting for a reply from a DAM process, the right child can be working on obtaining matches on the rows already produced by the left child. For more information about parallelism, see Section 8, Parallelism.

2 Accessing SQL/MX Data

Access methods provide different degrees of efficiency in accessing data contained in key-sequenced tables and indexes. Use the information in this section to understand the various access methods used by NonStop SQL/MX:

- Access Methods on page 2-1
- MultiDimensional Access Method (MDAM) on page 2-13

Access Methods

This subsection describes the access methods used by the SQL/MX compiler.

- Storage-key access
- Index-only access
- Alternate index access

Each access method and its attendant cost are discussed in these subsections. An alternative to using keyed access is a full table scan. For more information, see Full Table Scan on page 2-5.

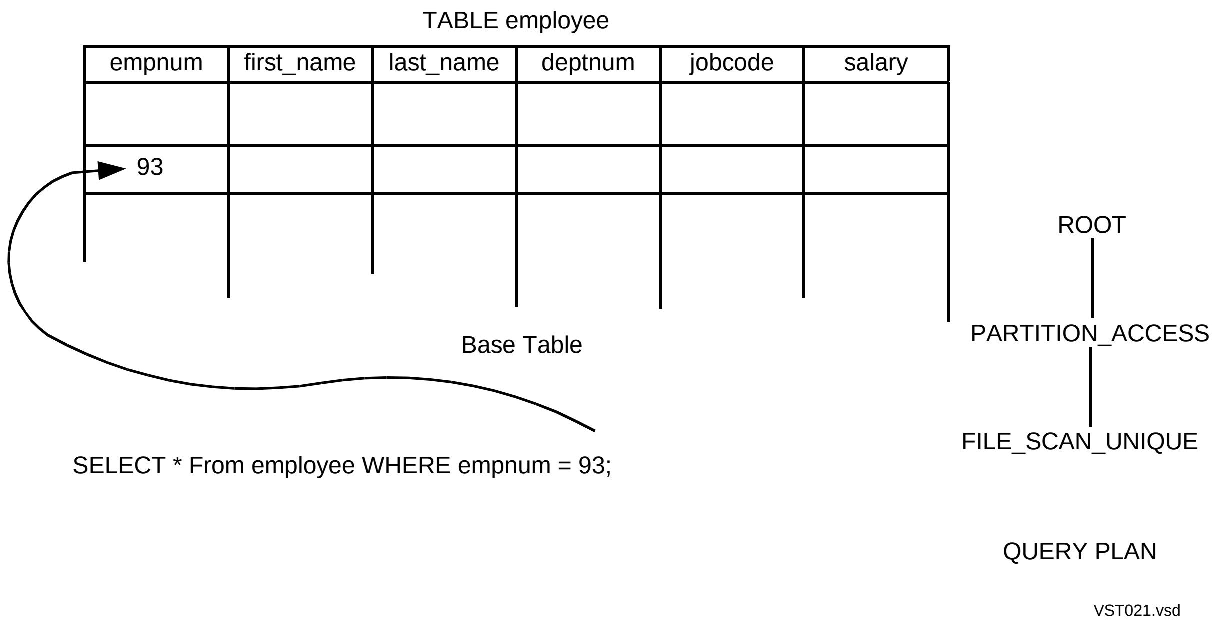
Storage-Key Access

The storage key refers to the physical order in which rows of the base table are stored on disk. This key can be the primary key, clustering key, or syskey, depending on which type of key was defined for the table.

The optimizer might choose storage-key access when:

- The WHERE clause contains a storage-key value.
- The estimated cost of the storage-key access is lower than the estimated cost of an index-only or alternate index access.

The next figure shows a query that contains a storage-key value (empnum) in the WHERE clause. The executor goes directly to the value indicated by the information in the WHERE clause to retrieve the row. The right portion of the figure shows the query execution plan generated by the query.



Storage-Key Approximate Cost

The cost of retrieving information through a storage key depends on how many blocks of data you must access.

Index-Only Access

Index-only access refers to an index that fully satisfies a query without accessing the base table. That is, all columns that the query references can be found in the index. For sequential access, an index-only scan can be superior to storage-key access of the base table. The index row sizes are usually considerably smaller than the base table row sizes (resulting in many more rows being retrieved per physical I/O).

An index contains one or more columns plus the columns that make up the clustering key. An index benefits the query most when all the columns needed by the query are located in the index. Although indexes can improve scan performance, they have a significant cost during updates, deletes, and inserts. Indexes usually use keys other than on the primary key, so you have alternate access paths to the data.

The optimizer is likely to choose access through an index when any of these conditions are true:

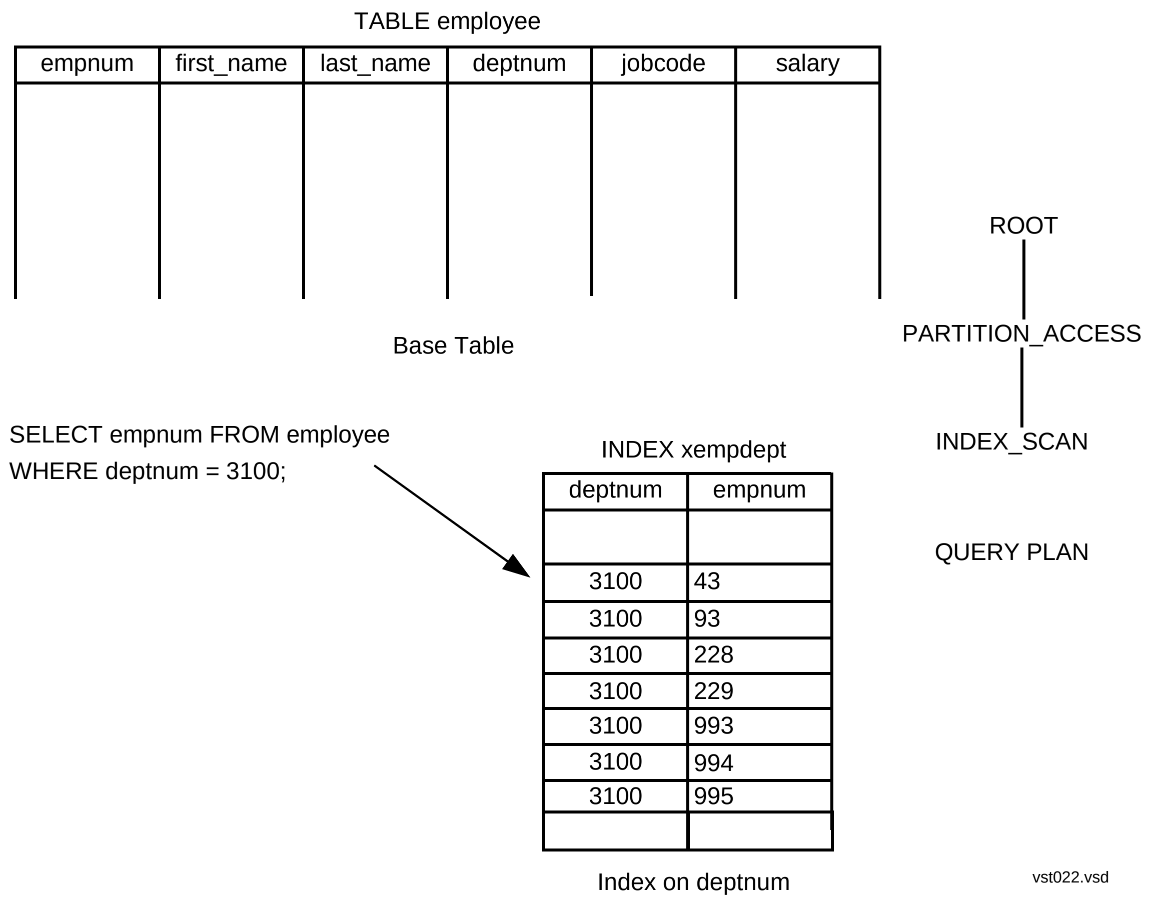
- The SELECT and WHERE clauses reference columns from the index.
- All the information can be retrieved from the index (index-only access) at less cost than accessing the base table.
- ORDER BY, GROUP BY, or DISTINCT is specified and can be satisfied by using the index.

Index-only access is not used if any of these conditions are true:

- The columns required by the query are not all included in the index.
- The query performs an update. Even if the index contains the column, the base table column (and any other index containing the column) also needs to be updated.

The next figure shows a query that contains an indexed item (deptnum) in the WHERE clause. The empnum column is the clustering key for the table and is also a column of the index. (Remember that an index contains one or more columns plus the columns that make up the clustering key of the table.) The compiler goes directly to the index to retrieve the information because:

- The cost is much less than using a full table scan.
- The query requests information that is contained in the index.



The query execution plan shows the index-only access.

Index-Only Approximate Cost

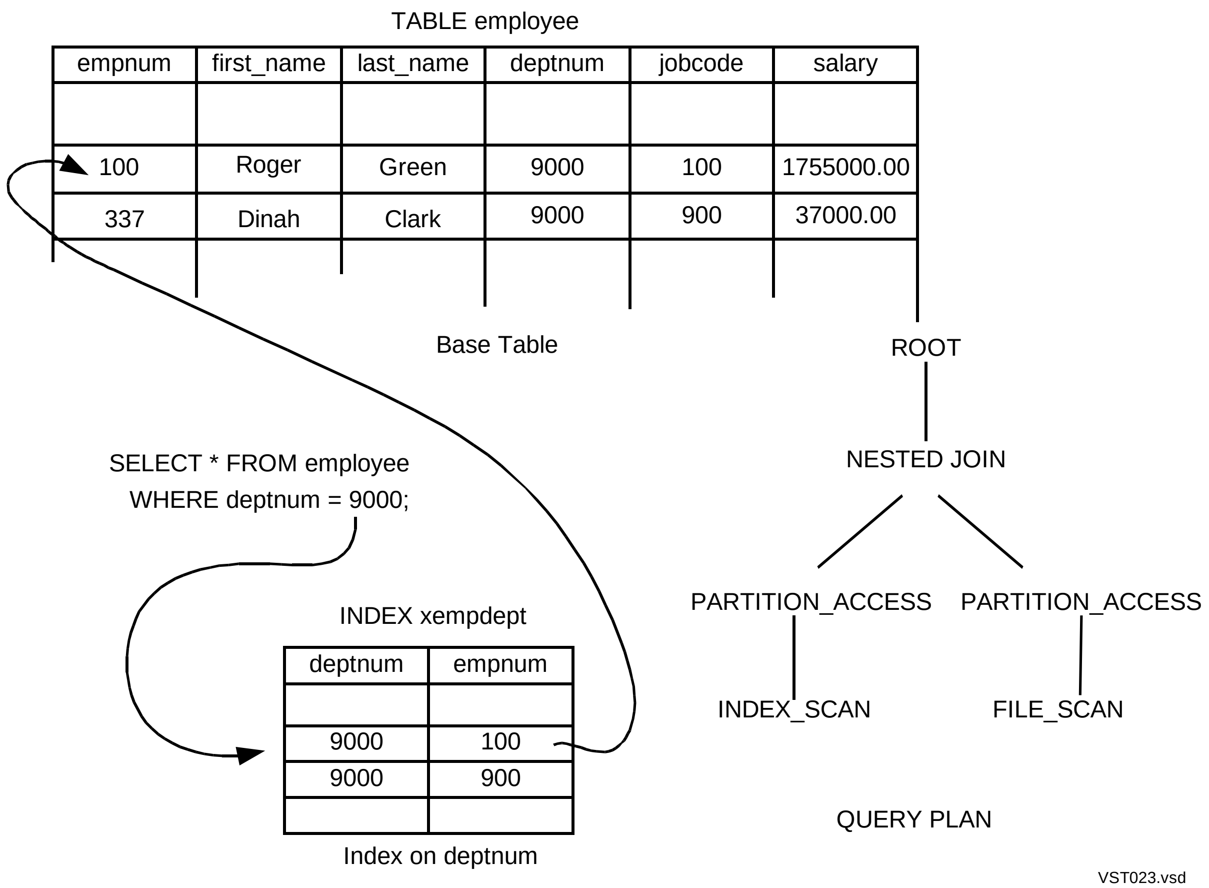
The approximate cost for index-only access is comparable to storage-key access. It might be better because typically index tables are smaller than base tables.

Alternate Index Access

In alternate index access, a join is made between the index and the base table (an index-base table join). Access is not made through the clustering index.

The index row is located by positioning to the requested data. When the index does not contain all the data requested, the index is joined with the base table to provide the requested data. When accessing the base table through an index, rows in the base table are usually read randomly, and some blocks containing those rows might be read more than once.

The next figure shows a query that requests all columns that satisfy the WHERE clause. Because the index contains only a subset of the columns, a join is made between the index and the base table to retrieve the row that satisfies the WHERE clause. The query plan shows the nested join that joins the index (represented by the INDEX_SCAN operator) and the base table (represented by the FILE_SCAN operator).



Alternate Index Access Approximate Cost

Because alternate index access relies on a join between an index and a base table, the cost associated with alternate index access can be high and is chosen only when the cost of a full table scan is even higher.

Full Table Scan

In a full table scan, SQL reads the entire base table from beginning value to end value in storage-key order. (If necessary, SQL can also read the table in reverse order.) Full table scans can be quite costly in terms of performance; response time is directly proportional to the number of rows or blocks processed.

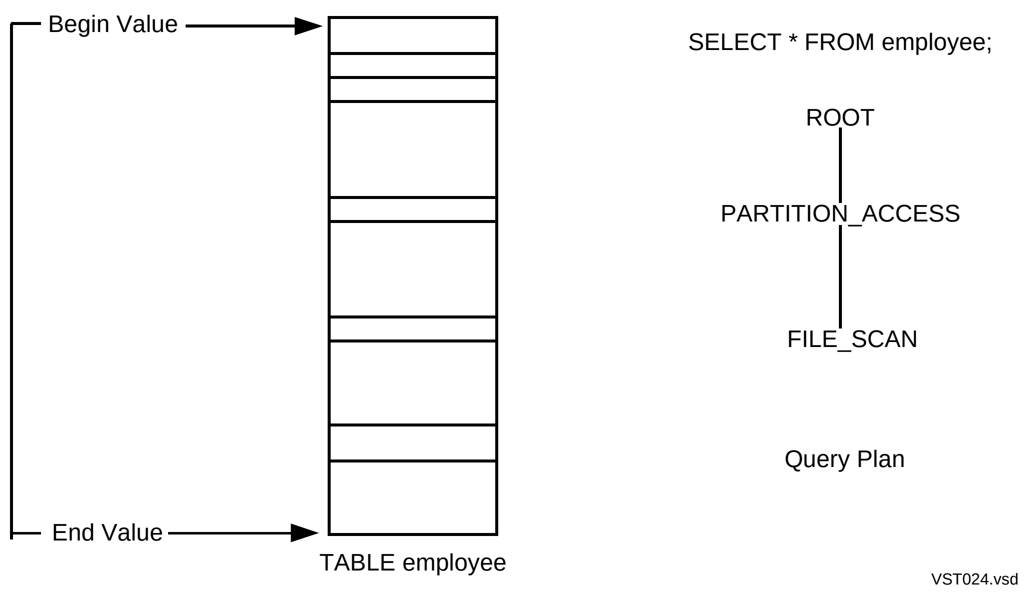
The optimizer might choose to scan the entire table when:

- Small tables are being processed.
- No suitable index is available.
- The estimated cost of reading the index and the corresponding base table rows exceeds the cost of reading the entire table.

To avoid or minimize table scans:

- Define a new index consisting only of required columns on frequently used queries.
- Use the CONTROL QUERY DEFAULT INTERACTIVE_ACCESS. See Minimizing Full Table Scans on page 2-6.
- Do not disable MDAM. For further information, see MultiDimensional Access Method (MDAM) on page 2-13.

To check for full table scans, use the EXPLAIN function. If in your EXPLAIN output for the scan node, the entries for the begin and end key contain the minimum and maximum values for each key column, the scan is reading the entire table. The maximum value of a key column depends on the data type of that column.



Full Table Scan Approximate Cost

A full table scan can be significantly higher in cost than the other methods.

Minimizing Full Table Scans

Use the INTERACTIVE_ACCESS CONTROL QUERY DEFAULT when you want to minimize the number of expensive full table scans. You might be interested only in the first few rows of the query result set, such as when using [`First N`] option. Because the SQL/MX compiler optimizes queries based on the cost of returning a full result set, the plan chosen could be inefficient for applications that require only the first few rows.

By setting INTERACTIVE_ACCESS to ON, the optimizer gives a higher preference to plans that utilize indexes for key lookup over plans that use full table or index scans. An index key lookup could be in the form of a search key or MDAM key on a primary or secondary index. Among plans with minimum number of full table or index scans, the optimizer chooses the plan with the lowest cost.

The following describes the optimizer strategy when INTERACTIVE_ACCESS is set to OFF or ON:

INTERACTIVE_ACCESS 'OFF' (default)

- Optimizer considers various plans for query execution.
- Optimizer chooses the plan with the lowest estimated cost.

INTERACTIVE_ACCESS 'ON'

- Optimizer considers various plans for query execution.

- Among the set of all plans considered, the optimizer chooses the set of plans with the minimum number of full table or index scans.
- Among the set of plans with a minimum number of full table or index scans, the optimizer chooses the plan with the lowest estimated cost.

These examples indicate the optimizer strategy when INTERACTIVE_ACCESS is set to ON:

```
T1(a,b,c,d,e,f) with indexes clustering(a), T1_b(b), T1_c(c)
T2(a,b,c,d,e,f) with indexes clustering(a,b,c), T2_b(b)
T3(a,b,c,d,e,f) with indexes clustering(a), T3_b(b)
T4(a,b,c,d,e,f) with indexes clustering(a,b), T4_c(c)
```

Example 1

```
select * from T1 where c=3;
```

The INTERACTIVE_ACCESS plan uses the index `T1_c` for efficient key lookup on column `T1.c`.

Example 2

```
select * from T1 where a=3;
```

The INTERACTIVE_ACCESS plan uses the `T1` clustering index for efficient key lookup on column `T1.a`.

Example 3

```
select * from T1 where a=3 and b=6;
```

The INTERACTIVE_ACCESS plan could use either `T1` clustering index (for key lookup on column a) or index `T1_b` (for key lookup on column `T1.b`).

Example 4

```
select * from T2 where b=3;
```

When INTERACTIVE_ACCESS is ON, the optimizer considers the index `T2_b`. In addition, if there is cost-efficient MDAM access to the clustering key, the optimizer also considers using the clustering index for the MDAM search. This strategy depends on the table size and UEC of column `T2.a`.

Example 5

```
select T1.c from T1, T3 where T3.b=5 and T1.c=T3.c;
```

The plan with the maximum number of index usages is:

```
          NJ
        /    \
     T3_b     T1_c
```

This plan does not have any full table or index scans because both index T3_b and T1_c are used for key lookup for the predicates T3.b=5 and T1.c=T3.c, respectively. In comparison, this hash join plan has one index used for lookup and one index fully scanned:

```
          HJ
        /    \
     T3_b    T1_c
```

The only index used for lookup in the previous hash join plan is T3_b. The index T1_c might have been chosen because of its smaller size, but it cannot be used for predicate evaluation on its key. As a result, the index T1_c will be fully scanned. In this scenario, when INTERACTIVE_ACCESS is set to ON, the optimizer frequently gives preference to nested join plans because they have a better chance of utilizing indexes for lookups.

Example 6

```
select T1.* from T1, T3 where T3.b=5 and T1.c=T3.c;
```

The plan with the minimum number of full table or index scans is:

```
          NJ
        /    \
     T3_b    NJ
            /   \
         T1_c   T1
```

No full table or index scans are performed in this plan.

Example 7

```
select T4.a, T1.f from T1, T4 where T1.c=T4.b and T4.c=5 and
    T1.a between 12 and 20;
```

In this case, several plans have the minimum number of full table or index scans. For example:

```
          HJ
        /    \
     T4_c    T1
```

This plan uses index T4_c for key lookup on T4.c and the clustering index on T1 for key lookup on T1.a.

Another plan with no full table or index scans would be:

```
               NJ
             /    \
           T4_c    NJ
                  /   \
                T1_c   T1
```

Understanding Unexpected Access Paths

Sometimes the optimizer does not choose the preferred or expected access path. If this happens, check:

- Does a WHERE clause refer to base table columns that do not have corresponding indexes?
- Do too many rows in the base table have to be randomly retrieved to justify using alternate-index access? (See WHERE Clause Indicates Base-Table Access on page 2-9.)
- Was index-only access not possible because of the columns that must be retrieved?
- Does the request UPDATE index columns? (See How the Compiler Avoids the Halloween Update Problem on page 2-9.)
- Did a CONTROL QUERY SHAPE statement instruct the compiler to take a different access path? (See Forcing an Access Path on page 2-11.)
- Did a CONTROL QUERY DEFAULT statement influence the compiler to take a different access path?

WHERE Clause Indicates Base-Table Access

The restriction specified by a WHERE clause might not result in a low enough selectivity to justify alternate-index access. The overall estimated cost might be higher for alternate-index access than for storage-key access.

How the Compiler Avoids the Halloween Update Problem

An alternate index might be ignored if a cursor or stand-alone SET UPDATE is specified for a column that is part of the alternate index. An update of an alternate index column might result in the deletion and reinsertion of the alternate index row after the current row position. This updated row might then be encountered again and updated again and again (the so-called Halloween update problem, named after the holiday on which it was discovered).

For example, suppose that PRICE is the first column of an index and is being incremented by 10 percent. As the column is updated, the row is inserted after its original position. The cursor or set update will once again encounter the row and increment it by 10 percent.

Selecting an index for an UPDATE query could result in a query plan that does not terminate when executed. Consider this query:

```
UPDATE INVNTRY SET RETAIL_PRICE = RETAIL_PRICE * 1.1;
```

The query requests that the price of all items in the INVNTRY table be increased by 10 percent. Suppose that RETAIL_PRICE has a nonunique index and that the index contains these rows before the update:

```
RETAIL_PRICE
------------
     10
     40
```

Suppose that the index on RETAIL_PRICE is the chosen access plan for a query requesting rows that satisfy the predicate:

```
RETAIL_PRICE > 20
```

The SQL/MX compiler finds the row with a retail price of 40 and updates it to 44. When the system looks for the next row that satisfies the predicate, it finds the same row, but with a value of 44 for RETAIL_PRICE. This process goes on forever.

A variation of this update problem occurs when newly inserted index rows satisfy the search condition for the query. In this case, some rows are updated twice, causing the reported “number of rows updated” to exceed the actual number of rows that satisfy the search condition. For example:

```
UPDATE INVNTRY
SET RETAIL_PRICE = 200
WHERE RETAIL_RICE BETWEEN 100 AND 400;
```

The SQL/MX compiler always avoids these types of update situations. One solution is to ignore the index on the column being updated (RETAIL_PRICE in the previous example) and choose another index as the access path, but this can result in an inefficient access plan.

If no other index for the INVNTRY table exists and the index on RETAIL_PRICE is not used, the whole table must be read. If the table is large, using the index is much more efficient.

SQL will likely use an alternate index for the scan operation of the update if any of these conditions are true:

- The alternate index contains no column that will be updated.
- The scan is guaranteed to be a unique access, in which case the compiler will not evaluate the same row more than once.

- All columns to be updated have an equality predicate in the WHERE clause. For example:

  ```
  UPDATE INVNTRY
  SET RETAIL_PRICE = RETAIL_PRICE * 1.1
  WHERE RETAIL_PRICE = 20 AND ITEM = 7;
  ```

  The rows will either get changed, in which case the selection condition does not qualify for the newly updated row, or the rows will get updated to the same exact value as before, in which case the index row does not change.

- No column is referenced on the right-hand side of the SET clause; as follows:

  ```
  UPDATE INVNTRY
  SET RETAIL_PRICE = 20
  WHERE RETAIL_PRICE > 80 AND ITEM > 10;
  ```

  Because the columns are being updated to a constant value, the Halloween update problem is avoided.

A variation of this update problem occurs when newly inserted index rows satisfy the search condition for a query. In this case, some rows are updated twice, causing the reported “number of rows updated” to exceed the actual number of rows that satisfy the search condition.

Forcing an Access Path

You can use the CONTROL QUERY SHAPE statement to force the access path for a query. The CONTROL QUERY SHAPE statement applies to any DML statement. Check the EXPLAIN output to see if the access path used is the path that you want. If the access path is not what you want, you can force your query execution plan with the CONTROL QUERY SHAPE statement. Follow the directions in Section 5, Forcing Execution Plans, to specify a new access path for your query execution plan.

If you think that you might benefit from the use of one of the CONTROL QUERY SHAPE options, check your application with and without forcing the plan by using actual statistics from production data.

If you use one of the options, you might want to change the forced shape later for reasons such as:

- The query might not be able to use a more efficient index that might be created in the future.
- The query might not be able to benefit from future enhancements to SQL.
- Changes to the database structure (such as dropping an index) can require recompilation when the option is in use.

Therefore, make occurrences of CONTROL QUERY SHAPE easy to find and change by using one or more of these alternatives:

- Make sure the forced shape applies only to the statement and table intended. Turn the forced shape off as soon as you are finished (CONTROL QUERY SHAPE OFF).
- Isolate this forced shape in its own section and perform it from the inline application code.
- Place all statements affected by the forced shape in separate modules, called as services by other modules.

  This example shows placing a statement affected by a forced shape into a separate module. Assume that an application uses a set of statically precompiled SQL statements that are in a module named `ansiMXSmd.esql`. In addition, part of the contents of this module include a frequently used query named `primarykeysQ1`:

```
...
/* SQL statement_name=primarykeysQ1 */

exec sql select ... from
:"hv_schemata_table" prototype 'N.SCH.SCHEMATA' sc,
:"hv_objects_table" prototype 'N.DSCH.OBJECTS' ob,
:"hv_cols_table" prototype 'N.DSCH.COLS' co,
:"hv_keycolusage_table" prototype 'N.DSCH.KEY_COL_USAGE' ky,
:"hv_tblconstraints_table" prototype 'N.DSCH.TBL_CONSTRAINTS' tc
where sc.SCHEMA_VERSION = :"hv_schema_version1"
and (sc.SCHEMA_NAME = :"hv_param2" or trim(sc.SCHEMA_NAME)
     LIKE :"hv_param3" ESCAPE '\')
and (ob.OBJECT_NAME = :"hv_param4" or trim(ob.OBJECT_NAME)
     LIKE :"hv_param5" ESCAPE '\')
and sc.SCHEMA_UID = ob.SCHEMA_UID and ob.OBJECT_UID = tc.TABLE_UID
     and tc.CONSTRAINT_TYPE = 'P'
and ob.OBJECT_UID = co.OBJECT_UID and tc.CONSTRAINT_UID =
     ky.CONSTRAINT_UID and ky.COLUMN_NUMBER = co.COLUMN_NUMBER
FOR READ UNCOMMITTED ACCESS order by 1, 2, 3, 5 ;
...
```

  Assume you want to force the `primarykeysQ1` query to use a given plan. You can extract this query from its original `ansiMXSmd.esql` module and place it into its own separate module, for example, `primarykeysQ1.esql`:

```
...
exec sql control query shape sort(
hybrid_hash_join(
nested_join(
hybrid_hash_join(
nested_join(cut,cut),
cut),
cut),
cut));
/* SQL statement_name=primarykeysQ1 */
exec sql select ... from
:"hv_schemata_table" prototype 'N.SCH.SCHEMATA' sc,
:"hv_objects_table" prototype 'N.DSCH.OBJECTS' ob,
:"hv_cols_table" prototype 'N.DSCH.COLS' co,
:"hv_keycolusage_table" prototype 'N.DSCH.KEY_COL_USAGE' ky,
:"hv_tblconstraints_table" prototype 'N.DSCH.TBL_CONSTRAINTS' tc
where sc.SCHEMA_VERSION = :"hv_schema_version1"
and (sc.SCHEMA_NAME = :"hv_param2" or trim(sc.SCHEMA_NAME)
       LIKE :"hv_param3" ESCAPE '\')
and (ob.OBJECT_NAME = :"hv_param4" or trim(ob.OBJECT_NAME)
       LIKE :"hv_param5" ESCAPE '\')
```

```
and sc.SCHEMA_UID = ob.SCHEMA_UID and ob.OBJECT_UID = tc.TABLE_UID
       and tc.CONSTRAINT_TYPE = 'P'
and ob.OBJECT_UID = co.OBJECT_UID and tc.CONSTRAINT_UID =
       ky.CONSTRAINT_UID and ky.COLUMN_NUMBER = co.COLUMN_NUMBER
FOR READ UNCOMMITTED ACCESS order by 1, 2, 3, 5 ;

exec sql control query shape off;
...
```

By extracting the query from its original module and placing it into a separate module, you assure:

- The forced plan is applied only to the SQL compilation of the query.
- You can easily find and maintain the query in case you need to change it again to force a different plan.

MultiDimensional Access Method (MDAM)

The MultiDimensional Access Method (MDAM) provides optimal access to certain types of information when predicates contain key columns. Based on the predicates you specify, MDAM reads the minimal set of records and retrieves rows in key order, allowing the optimizer to avoid sorts when it can and to do merge joins. MDAM is costed by the optimizer whenever a key predicate is contained in the query and statistics exist for the key. The decision to choose MDAM is based on whether the access is less expensive than single subset access.

Use MDAM to:

- Save indexes. With MDAM, you need fewer indexes and so can save the maintenance and space associated with extra indexes.
- Save resources.
- Obtain good performance in situations where previously a full table scan was needed.
- To add rows to a table without suffering performance penalties.

Specifying MDAM

MDAM is enabled by default. Enable means that the SQL/MX compiler can choose to use MDAM if it will provide a better quality plan. Several SQL/MX compiler control settings affect MDAM:

- CONTROL QUERY DEFAULT provides a system-wide “master” switch for the current session. You can switch MDAM off or on and enable MDAM if it is disabled.
- With CONTROL QUERY SHAPE, you can force columns to be used by MDAM, and you can also force enumeration algorithms. For more information about enumeration algorithms, see MDAM’s Use of DENSE and SPARSE Algorithms on page 2-18.

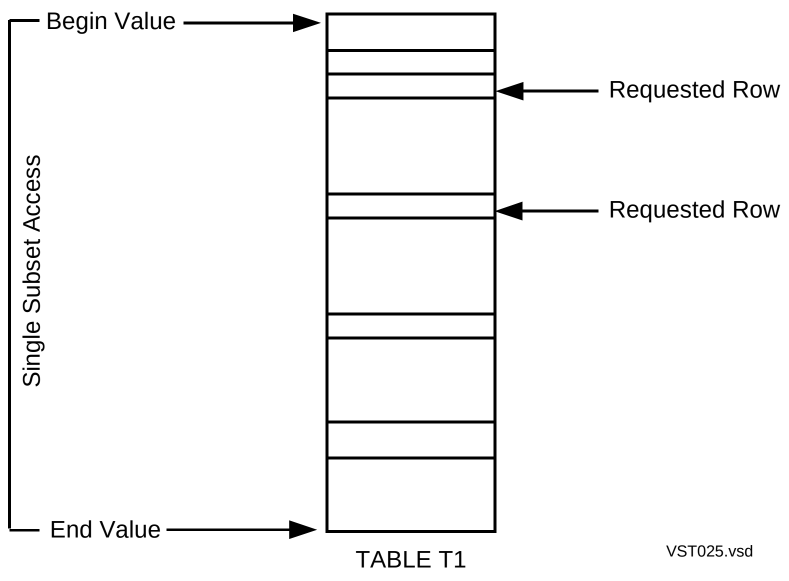
- With CONTROL TABLE, you can control MDAM at the current process level or for a specific table or index. The * (asterisk) option specifies all tables. If you specify CONTROL TABLE * MDAM 'OFF,' MDAM is disabled for all tables and indexes. If you specify CONTROL TABLE * MDAM 'ON', only MDAM access will be tried for all indexes and tables.

Note. If you use CONTROL TABLE *tablename* MDAM 'OFF', the compiler could still use MDAM for an index of the same table.

Turn MDAM off only if you find the query runs better without it.

Comparing MDAM With Single Subset Access

Without MDAM, tables are accessed by the single subset access method.



This figure shows single subset access on table T1. The lines in the table indicate the rows that are being accessed by the predicates on the query. Notice that the entire range from beginning to end is scanned to return the rows requested. Between the requested rows are a number of rows that contain information that is not needed.

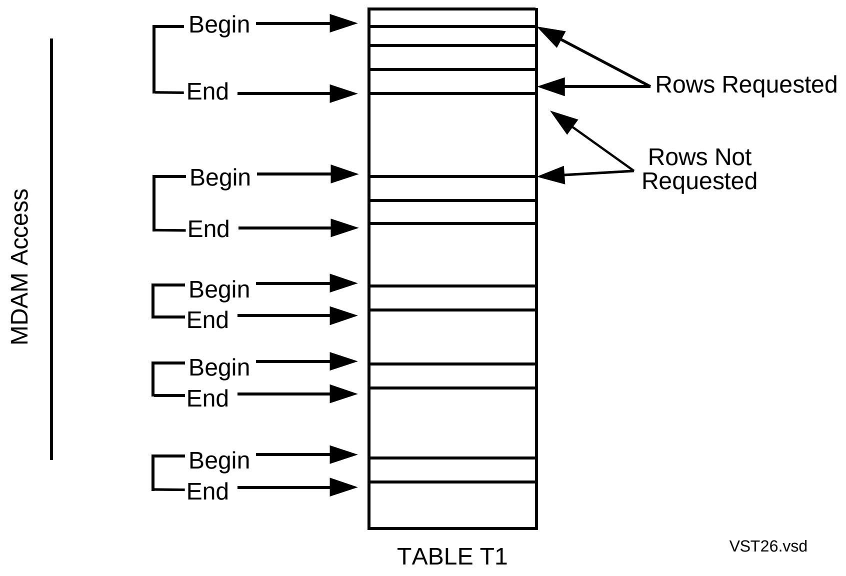
Single subset access is characterized by scanning a range of values, from begin key to end key, that results in reading all the rows between the begin and end keys. The cost associated with single subset access is equal to:

```
Cost of reading N blocks
+ cost of applying M predicates to R records
+ cost of moving the passed records
```

The cost of moving the passed records is a fixed cost, so it cannot be changed. The other costs, however, can be improved.

As shown in the next figure, the scan of the single subset access starts with the begin value and finishes at the end value when the last row is read. To reduce the cost of reading N blocks, you could break the table up into a series of smaller ranges with a high potential for hits. By reducing the number of blocks read, you also reduce the cost of applying the predicates to the records, because fewer records are scanned.

With MDAM, access is based on the key-column predicates, and the table is accessed in a series of smaller begin and end ranges. This scenario requires more key positionings.



With MDAM, the begin and end key range is determined at run time. The executor does the key positionings based on information provided by the optimizer.

The cost associated with MDAM access is determined through this formula:

```
Cost of reading N blocks (covered by ranges)
+ cost of applying M predicates against R records
+ cost of moving the passed rows (again, a fixed cost)
+ number of key positions required
```

This formula works best for low selectivity columns (low unique entry count (UEC) = fewer key positionings).

How MDAM Processes Queries

When processing a query, MDAM:

- Enables range predicates on leading or intervening key columns
- Accommodates missing predicates on leading or intervening key columns
- Performs general OR optimization
- Handles IN lists on multiple key columns
- Eliminates redundant and contradictory predicates

- Does not read the same row twice
- Maintains sort order

Intervening Range Predicates

An intervening range predicate occurs when another key column predicate follows the first predicate:

```
A > 5 AND B = 2
```

MDAM processes range predicates by stepping through the existing values for the column on which the range has been specified. Data outside the bounds is not read from disk or handled in any way.

Missing Key Predicates

When no predicates have been specified for a leading or intervening key column, MDAM can still use the subsequent columns for keyed access. MDAM is most effective when a skipped column (no predicate) has a low UEC. Consider this query where the index contains DEPT, SALES_DATE, ITEM_CLASS, and STORE, in that order:

```
SELECT SALES_DATE, SUM(TOTAL_SALES)
  FROM SALES
  WHERE SALES_DATE BETWEEN DATE'06/01/95' AND DATE'06/30/95'
  AND ITEM_CLASS=20 AND STORE=250 GROUP BY DEPT, SALES_DATE;
```

Notice that no predicate was provided for the first key column, DEPT. Without MDAM, the compiler would have to resort to a full table scan for the query. MDAM treats the missing predicate for DEPT as an implied range of MIN_VALUE to MAX_VALUE (including NULL values). These values are respectively the minimum and maximum permissible values supported for the data type of the missing key column.

OR Predicates

MDAM works with OR operators to:

- Avoid creating duplicates:

  ```
  A IN (1, 2, 3, 2)
  ```

  MDAM reads 1, 2, 3. The last 2 is not read.

- Allow WHERE clauses that are not in disjunctive normal form; that is, WHERE clauses can have more than one level of OR operations.

IN Lists

An IN predicate equivalent is the result when an IN predicate is converted into a series of ORs:

```
COL1 IN (1, 2, 3)
```

This IN predicate is converted into:

```
COL1 = 1 OR COL1 = 2 OR COL1 = 3
```

Consider:

```
SELECT * FROM T WHERE
   ((A = 4 AND B IN (2,5))
    OR (A = 7 AND B IN (6,9)) ;
```

The optimizer transforms this query into these predicate sets:

```
(A = 4 AND B = 2) OR
(A = 4 AND B = 5) OR
(A = 7 AND B = 6) OR
(A = 7 AND B = 9)
```

Redundant and Contradictory Predicates

MDAM eliminates predicates that conflict with other predicates. For example, this query shows conflicting predicates:

```
SELECT * FROM T1
  WHERE DIMENSION_2
    BETWEEN 2 AND 3 AND DIMENSION_2=1;
```

MDAM recognizes that these predicates conflict and removes both of them, resulting in no table accesses (that is, DIMENSION_2 cannot be both 1 and [2 through 3]). In addition, at run time, MDAM resolves conflicting predicates and combines overlapping ranges and IN lists within a single disjunct.

Duplicate Rows

MDAM avoids reading the same data twice so that it does not have to do post-read operations to accomplish duplicate elimination. MDAM combines overlapping ranges among the disjuncts and separates the disjuncts into nonoverlapping accesses.

Sort Order

MDAM orders the retrievals in the order of the index being accessed. The order can be ascending or descending. MDAM maintains the index order even when reading the index backward, which satisfies ordering requirements and prevents a sort of the query.

Influencing the Optimizer to Use MDAM

You can influence the optimizer to choose MDAM as the access path by ordering your key columns in your index design:

- Perform UPDATE STATISTICS on leading columns. As the UECs for leading columns increase (and if the columns do not have predicates on them in the query), the number of seeks to the table increases. The UECs of leading columns are the single most important factor in MDAM access.

- Order your key columns by the columns that you access most frequently. If you frequently perform a query that uses department number, include the DEPTNUM column in your key-column index.

  For example, if the EMPLOYEE table contains the columns STATE, JOBCODE, DEPTNUM, LAST_NAME, FIRST_NAME and you frequently perform queries based on JOBCODE, STATE, and DEPTNUM, place the columns in that order.

Controlling the Number of Key Columns Used by MDAM

In most cases, you should allow MDAM to choose the number of key columns to use for MDAM processing. Occasionally, MDAM chooses more or fewer key columns than you expect. On these occasions, you can force MDAM use with the CONTROL QUERY SHAPE statement to specify the number of key columns for MDAM to use. For information about using CONTROL QUERY SHAPE, see Section 5, Forcing Execution Plans.

Suppose that four columns of the key have predicates. You have determined the Unique Entry Count (UEC) of the third key column used in a predicate. (You have found the count by updating statistics or querying the count for the column.) If you do not want MDAM to step through all the different values for the third column, you can specify the number of columns for MDAM to use. If you specify two, only predicates on the first two key columns are used.

If you specify a number of columns that is less than or equal to zero, SQL returns an error. If the number you specify exceeds the number of key columns available for the index, the optimizer uses the maximum number of key columns usable by MDAM for each predicate set. Specifying SYSTEM for the number of key columns enables MDAM to choose the number of key columns.

MDAM’s Use of DENSE and SPARSE Algorithms

A dense key column has all (or almost all) the possible values for a column. If a column has 100 unique values and the column value ranges from 0 through 99, the column is considered dense. When a dense column is recognized, MDAM has to add only one to the previous value to find the next value in the column.

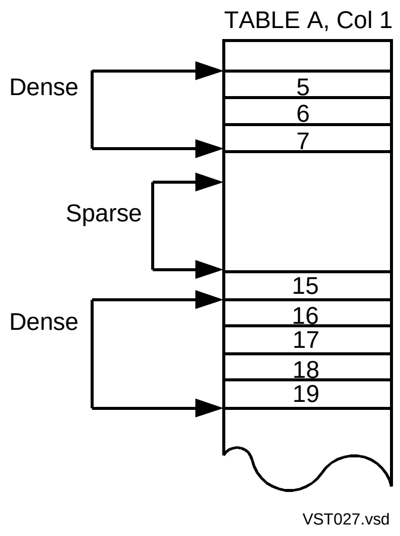
A sparse key column has missing possible values. A column is considered sparse when the number of actual values in the column is small relative to the set of all possible values. Sparse columns contain gaps in the ranges specified for the key column. The optimizer detects these gaps by analyzing the UEC and high and low values of the column.

The compiler and executor use adaptive DENSE and SPARSE algorithms for the leading key column only. The algorithm chosen by the compiler might not be rigidly followed by the executor. For example, if the optimizer chooses a DENSE algorithm and the executor finds the DENSE algorithm is inefficient for accessing the column, the executor adapts by switching to a SPARSE algorithm when it discovers that many values are missing. The executor chooses the appropriate algorithm for the density or sparseness of data.

When the compiler chooses a SPARSE algorithm, the executor executes only the SPARSE algorithm and does not attempt to switch to a DENSE algorithm.

You can force the choice by specifying the SPARSE or DENSE option in the CONTROL QUERY SHAPE statement. If you force a DENSE algorithm, the executor does an adaptive DENSE or SPARSE and switches accordingly when it finds that the chosen algorithm is not efficient for the column it is accessing. If you force a SPARSE algorithm, the executor uses only the SPARSE algorithm.

In the next figure, Table A shows that Col. 1 has values that always increase by one. If the compiler chooses MDAM DENSE, the executor starts with DENSE until it discovers a gap in the useful information requested, and then it switches to SPARSE. With the SPARSE algorithm, the executor must retrieve a row to determine the next actual value in a column. If the compiler chooses SPARSE, the executor does not attempt DENSE.



3 Keeping Statistics Current

When you update statistics, information about a table is updated in the histogram tables so that the information more accurately represents the current content and structure of the database. Use the information in this section to understand when and why you should update statistics:

- Histogram Statistics on page 3-1
- Sampling and UPDATE STATISTICS on page 3-5
- Testing the Results of UPDATE STATISTICS on page 3-8

You must execute the UPDATE STATISTICS statement for each table. NonStop SQL/MX does not automatically update statistics.

Histogram Statistics

Histograms are critical to the optimizer's ability to differentiate between plans. The optimizer uses histograms to estimate the row count that flows out of each operator in a query execution plan and uses these estimates to calculate the total cost of a plan. Different plans are then compared to find the best plan.

The best practice is to update statistics for every table column involved in a query (that is, every column in every predicate, join, group by, and order by expression). However, you might decide that the cost of updating statistics is not justified by the gain in optimizer accuracy. In this case, you can adjust the default setting constant values for histograms. For information about histogram default settings, see the SYSTEM_DEFAULTS table entry of the *SQL/MX Reference Manual*.

When you update statistics, histogram statistics for a group of columns or individual columns are collected, including for each column or set of columns:

- Current number of rows in the table
- Number of unique entries in the table
- Highest value for the table column
- Lowest value for the table column
- Number of intervals in the histogram
- Number of unique entry counts (UEC) in each interval
- Number of rows in each interval
- Interval boundary

Histogram statistics are updated by using the UPDATE STATISTICS statement with the ON clause. Stored in the user catalog, histogram statistics are stored for the entire table and columns in the HISTOGRAM and HISTOGRAM_INTERVALS user metatdata tables. Histogram statistics are discussed under Updating Histogram Statistics on page 3-2.

For the syntax of the UPDATE STATISTICS statement and for more information about histogram tables, see the *SQL/MX Reference Manual*.

Note. The compiler uses default statistics if no updated statistics exist for the table and column. When the compiler uses default statistics, the execution plan provided might not be the optimal plan. If statistics have not been updated, the compiler uses the block count from the file label. However, if the block count is zero, the compiler uses the default value for the HIST_NO_STATS_ROWCOUNT attribute. The compiler issues warnings if statistics have not been generated for a relevant column when the table contains more rows than the value defined by the default HIST_ROWCOUNT_REQUIRING_STATS.

Updating Histogram Statistics

When you update statistics on a column or a group of columns, NonStop SQL/MX generates histogram statistics. Histogram statistics enable the optimizer to create efficient access plans.

Histogram statistics are stored in two tables. The name and location of these tables depends on whether you are using SQL/MX or SQL/MP tables.

Table 3-1 shows the temporary tables information created for histograms.

Table 3-1. Histogram Temporary Tables

Statistics	SQL/MX Objects	SQL/MP Objects
Registration	Registered in the same `catalog.schema` as the table.	Registered in the catalog of the primary partition of the table.
Location	Located in the same `catalog.schema` as the table.	Located in the same `\node.$vol` as the primary partition, in the ZZMXTEMP subvolume.
File names	`catalog.schema.`*`SQLMX_<object_uid_of_tablename>_<seconds_part_from_current_timestamp>_<microseconds_part_from_current_timestamp>`*	`\node.$vol.ZZMXTEMP.`*`tablename`*
Size Limits	Files are always format 2, limited to 1 TB or the amount of available space on the disk volume.	File format is determined by the format of the base table's primary partition. Format 1: The temporary table is limited to 2 GB. Format 2: The temporary table is limited to 1 TB of the space available on the disk volume.

Table 3-2 compares histogram statistics table information.

Table 3-2. Histogram Statistics Tables

Statistics	SQL/MX Objects	SQL/MP Objects
Registration	Registered in the same `catalog.schema` as the table.	Registered in the catalog of the primary partition of the table.
Location	Located in the same `catalog.schema` as the table.	Located in the same `\node.$vol.subvol` as the catalog.
File names	`catalog.schema.HISTOGRAMS` `catalog.schema.HISTOGRAM_INTERVALS`	`\node.$vol.subvol.HISTOGRM` `\node.$vol.subvol.HISTINTS`

The HISTOGRAMS and HISTOGRM tables store histogram-specific table and column information. The HISTOGRAM_INTERVAL and HISTINTS tables store interval information for the data distribution of a column or group of columns. When you run UPDATE STATISTICS again for the same user table, the new data replaces the data previously generated and stored in the histogram tables.

The HISTOGRAMS and HISTOGRM tables show how data is distributed with respect to a column or a group of columns. When generating a histogram for a table, NonStop SQL/MX distributes the values of the specified columns into some number of intervals. An interval represents a range of values for the column. The range of values for each interval is selected by NonStop SQL/MX so that every interval represents approximately the same number of rows in the table.

For example, if a table contains 1 million rows and UPDATE STATISTICS generates 20 intervals for a column in that table, each interval represents 50,000 rows. (This is sometimes known as “equal height” distribution over the histogram intervals.) The optimizer computes statistics associated with each interval and uses the statistics to devise optimized plans.

The ON EVERY COLUMN clause of the UPDATE STATISTICS statement generates separate histogram statistics for every individual column and any multicolumns that make up the primary key and indexes in the table.

Consider a table that contains columns A, B, C, D, and E. The ON EVERY COLUMN option generates a single column histogram for columns A, B, C, D, E. The number of multicolumn histograms generated depends on the primary key definition and the indexes defined. With the primary key defined on A, B, and C and no index defined, NonStop SQL/MX generates two multicolumn histograms (A, B, C) and (A, B). Using the same table and columns, the next example shows when multicolumn histograms are generated based on the primary key and defined indexes:

Table has columns A, B, C, D, E

```
Case 1
KEY: (A, B, C,)  => (A, B, C), (A, B)
INDEX: none   => none
RESULT: (A, B, C), (A, B)
```

In Case 2, the index (E) is defined as nonunique, so the KEY is added to the end of the INDEX, and the index is INDEX+KEY:

```
Case 2
KEY: (A, B, C)  => (A, B, C), (A, B)
INDEX: (E) nonunique => (E, A, B, C), (E, A, B), (E, A)
RESULT: (A, B, C), (A, B), (E, A, B, C), (E, A, B), (E, A)
```

In Case 3, because the index (E) is defined as a unique index, the KEY is not added to the end of the INDEX. The index is INDEX. Because the index is a single column, it is processed as a single-column histogram only (no multicolumn histograms are generated).

```
Case 3
KEY: (A, B, C)  => (A, B, C), (A, B)
INDEX: (E) unique = > no multicolumn histogram
RESULT: (A, B, C), (A, B)
```

Histogram tables are not automatically updated when you update a table for which statistics are stored. To keep the histogram statistics current, execute the UPDATE STATISTICS statement after significantly updating tables.

NonStop SQL/MX reduces compile time for less complex queries by caching histograms. When the histogram is cached, it can be retrieved from the cache rather than from the disk for future queries on the same table. Histogram caching provides faster access to histograms.

Several default settings for histogram tables can be changed. For more information, see the SYSTEM_DEFAULTS table entry in the *SQL/MX Reference Manual*.

Knowing When to Update Statistics

Before you update statistics, consider:

- Using sampling to reduce the amount of time required for updating statistics. See Sampling and UPDATE STATISTICS on page 3-5.
- Updating statistics only after a table has been loaded with data.

Other performance issues to consider when you experience reduced response time are:

- The node containing the table might have heavy disk usage because of long ad hoc queries or reports.
- If the table or index is distributed, the network might be rerouted or might have heavy use.
- NonStop SQL/MX does not automatically recompile your programs when you update statistics.

Analyzing the Possible Impact of Updating Statistics

Depending on the size of the table, updating statistics can take longer than you would like. Consider updating statistics during the hours when peak performance is not required.

If you want to preserve the existing query execution plan, be aware that updating statistics might cause the optimizer to choose a different plan. Usually, you can improve performance by updating the statistics on a table to reflect the current status. The update statistics operations, however, might not improve performance, as discussed next:

- You can update statistics to perform a sampling of rows to determine the statistical information. Depending on your sample size, this procedure could take a long time. (That is, the larger the sample, the longer it might take.) Because this is a statistical sampling method, the statistics gathered are not exact. Inaccurate statistics (wrong by more than 10 percent) can adversely affect the plan. See Sampling and UPDATE STATISTICS.
- Because the UPDATE STATISTICS statement does not automatically recompile programs, the operation does not invalidate the dependent programs. If you want to take advantage of the new statistics, however, you must explicitly recompile the dependent programs.

Sampling and UPDATE STATISTICS

Use sampling to control the amount of time spent calculating statistics. If you do not specify sampling, statistics are collected by scanning the entire table. The optional SAMPLE clause provides several methods of producing a sample set, based on a ratio, to determine the histograms. The *SQL/MX Reference Manual* describes each sampling method in detail.

You might want to use sampling because of the amount of time required to update statistics on the entire table. Sampling techniques that do not perform full large table scans result in significant performance gain. You can use SAMPLE without specifying the number of rows. By default, NonStop SQL/MX samples two percent of the table up to a maximum of two million rows. You might want to explicitly specify a larger sample size to increase statistics accuracy or a smaller sample size to reduce the running time.

When you use the SAMPLE option with UPDATE STATISTICS, a temporary table is created. You can use the HIST_SCRATCH_VOL default attribute for SQL/MX and SQL/MP tables. For SQL/MX tables, NonStop SQL/MX creates a single partition or hash partitions across multiple volumes as specified in the HIST_SCRATCH_VOL default attribute. NonStop SQL/MX determines how many partitions are needed based on the sample set retrieved by the SAMPLE option. It randomly selects the volume or volumes to use from the list of volumes specified in the HIST_SCRATCH_VOL default attribute. NonStop SQL/MX creates only as many partitions as specified with HIST_SCRATCH_VOL. If the default attribute is not used, the temporary table is created in the default volume specified by the _DEFAULTS define. For SQL/MP tables, a single partition is created on the volume specified in the HIST_SCRATCH_VOL default attribute. If the default attribute is not used, the temporary table is created in the same volume as the primary partition of the table. Because the temporary table is used for data gathering and calculation purposes, you must have adequate disk space to accommodate it on your system. For more information about the HIST_SCRATCH_VOL default attribute, see the *SQL/MX Reference Manual*.

Starting with SQL/MX Release 2.3.2, the temporary table can be created as a partitioned table. When you use the USING SAMPLE TABLE WITH PARTITIONS clause, SQL/MX creates a temporary table that is partitioned the same way as the base table for which the statistics are collected. The partitioned temporary table can increase the speed of the UPDATE STATISTICS command significantly because both write and read operations on the table can occur simultaneously.

Use a partitioned temporary table when the UPDATE STATISTICS command takes several tens of minutes to complete—there might not be any gain when the command takes only a few minutes.

The performance with partitioned temporary tables is heavily dependent on the degree of parallelism in the operations that the UPDATE STATISTICS command uses to write to and read from the temporary table. The following steps are recommended for obtaining the maximum parallelism from SQL/MX:

- Ensure that the number of partitions is a multiple of the number of CPUs in the system.
- Distribute the data records evenly across all partitions.
- Distribute the partitions evenly across all available disks.
- Distribute the disks evenly across all available CPUs.

An uneven distribution might degrade the performance with partitioned temporary tables. You can resolve this problem, in some cases, by creating a temporary table and specifying it for the command to use. To do this, use the USING SAMPLE TABLE `table-name` clause. The temporary table must meet these criteria:

- The column attributes must be the same as the base table—the number of columns, order, and data types must match.
- There must not be any indexes or triggers.
- There must be no constraints.
- The table must be empty.

When partitioning the temporary table, avoid using the disks and CPUs that are heavily loaded and try to distribute the partitions evenly across the remaining disks and CPUs.

When the USING SAMPLE TABLE clause is used, the HIST_SCRATCH_VOL default attribute will be ignored.

From SQL/MX 2.3.2 onwards, it is also possible to push the sampling operation down into DP2. As a result, the number of read and discarded records reduces, thereby improving the performance of the command. This technique is effective for low sampling percentages. Use the ALLOW_DP2_ROW_SAMPLING default attribute to control the sampling. The attribute values are SYSTEM, ON, and OFF. The default value is SYSTEM. When the attribute is set to SYSTEM, UPDATE STATISTICS pushes the sampling down into DP2 for sampling percentages up to 5%. When it is set to ON, UPDATE STATISTICS pushes the sampling down into DP2 for sampling percentages up to 50%. When set to OFF, the sampling is not pushed down into DP2.

When you use the sampling option, you can increase the efficiency by specifying a row count for the columns you want to update. The next statement provides a way to obtain the row count for the EMPNUM column of the EMPLOYEE table:

```
SELECT COUNT(*) FROM EMPLOYEE;
```

Use the value returned from running the SELECT COUNT (*) statement (125000) to specify the valid row count in the UPDATE STATISTICS statement:

```
UPDATE STATISTICS FOR TABLE EMPLOYEE ON (EMPNUM)
   SAMPLE 5000 ROWS SET ROWCOUNT 125000
```

Performance Issues and Accuracy in Sampling

Certain sampling options provide more accurate statistics than others, but higher accuracy can mean performance trade-offs. When you need high accuracy or quality in your sample, use random row sampling. In this method, the intermediate table being sampled is scanned, and each row is selected with probability *n*/100, where *n* is the sample percentage.

A faster alternative to random row sampling is cluster sampling. Use this method when performance is a high priority, or when the quality of a sample is determined to be good enough, based on the knowledge of the table for which statistics are being updated.

In cluster sampling, only disk blocks that are actually part of the result set are read and processed. If the sampling percentage is small, which is typically the case, the performance advantage of cluster sampling over other sampling methods can be dramatic. For example, consider two queries, one that selects a 1 PERCENT row sample of a table and another that selects a 1 PERCENT cluster sample of the same table. Because cluster sampling requires reading and processing only approximately 1 percent of the table, and row sampling reads and processes the entire table, cluster sampling can be up to 100 times faster than random row sampling. However, the trade-off for this performance is less accuracy than random row sampling provides.

Other sampling methods, such as PERIODIC, do not provide additional performance or accuracy but provide semantics that are appropriate in certain situations. Use PERIODIC sampling (for example, choosing 1 of every 10 rows) when it is important to compute statistics from all parts of a table. For example, if a table is ordered by timestamp and column values vary widely over time, the use of PERIODIC sampling ensures that statistics are computed from rows spread evenly throughout the entire table, resulting in accurate statistics.

These sampling options perform a full table scan prior to selecting the sample set:

```
SAMPLE RANDOM x PERCENT
SAMPLE PERIODIC x ROWS EVERY y ROWS
```

These sampling options do not perform a full table scan to determine the sample set. In addition, the accuracy of all these examples is equivalent to SAMPLE RANDOM x PERCENT CLUSTER OF y BLOCKS:

```
SAMPLE
SAMPLE r ROWS
SAMPLE SET ROWCOUNT c
SAMPLE r ROWS SET ROWCOUNT c
```

Collecting Statistics for Multiple Columns

Multicolumn unique entry count (UEC) is the UEC for a combination of columns. It is the total number of unique combinations for the set of columns. The multicolumn UEC enables the optimizer to give a better prediction of the number of rows and to provide better plans. Specifically, the multicolumn UEC enables the optimizer to predict the number of rows resulting from a grouping operation on multiple columns and in the case of multicolumn joins.

If you have a set of columns you normally use for grouping (such as JOBCODE and DEPTNUM), without multicolumn statistics, the optimizer multiplies the UECs of the columns together to get the combined UEC. However, this result can be orders of magnitude higher than the real number of combinations. Likewise for joins, if you have two tables joined by two or more columns, a multicolumn histogram for the set of join columns from each table gives a better result.

Testing the Results of UPDATE STATISTICS

Because updating statistics can have a significant impact on plan quality, you can try to determine the benefits of the operation before you decide to update the histogram tables.

Testing the Results for SQL/MP Tables

To test the results of updating statistics:

1. Prepare a sample query from your application. (Consider using a commonly used query from your application.)
2. Execute the query.
3. Use EXPLAIN to obtain the cost information for your query. For information about reviewing plans with the EXPLAIN function, see Displaying Selected Columns of the Execution Plan on page 4-5.

4. Determine the effect of the UPDATE STATISTICS statement and optionally back out the generated histogram if necessary:

   a. In SQLCI, back up current histogram tables, if any:

   ```
   SQLCI> DUP histogrm, myogrm;
   SQLCI> DUP histints, myints;
   ```

   b. In MXCI, issue the UPDATE STATISTICS command for required column groups.

   c. In MXCI, recompile the query.

   d. In MXCI, use EXPLAIN to review the cost information for your query.

   e. In MXCI, use DISPLAY STATISTICS to determine which plan is better.

   f. If necessary, use the UPDATE STATISTICS CLEAR option (in MXCI) to remove histograms for unwanted column groups.

   g. If necessary, in SQLCI, restore backup histogram tables:

   ```
   SQLCI> DROP TABLE histogrm;
   SQLCI> DROP TABLE histints;
   SQLCI> DUP myogrm, histogrm;
   SQLCI> DUP myints, histints;
   ```

Testing the Results for SQL/MX Tables

To test the results of updating statistics:

1. Prepare a sample query from your application. (Consider using a commonly used query from your application.)

2. Execute the query.

3. Use EXPLAIN to obtain the cost information for your query. For more information, see Displaying Selected Columns of the Execution Plan on page 4-5.

4. Determine the effect of the UPDATE STATISTICS statement and optionally back out the generated histogram if necessary:

   a. In MXCI, back up current histogram tables, if any:

   ```
   > CREATE TABLE myhist LIKE HISTOGRAMS;
   > INSERT INTO myhist SELECT * FROM HISTOGRAMS;
   > CREATE TABLE myhistint LIKE HISTOGRAM_INTERVALS;
   > INSERT INTO myhistint SELECT * FROM HISTOGRAM_INTERVALS;
   ```

   b. Issue the UPDATE STATISTICS command for required column groups.

   c. Recompile the query.

d. Use EXPLAIN to review the cost information for your query.

e. Use DISPLAY STATISTICS to determine which plan is better.

f. If necessary, use the UPDATE STATISTICS CLEAR option to remove histograms for unwanted column groups.

g. If necessary, restore backup histogram tables:

```
> DELETE FROM HISTOGRAMS;
```

```
> INSERT INTO HISTOGRAMS SELECT * FROM myhist where
table_uid in (select object_uid from
CAT.DEFINITION_SCHEMA_VERSION_3000.OBJECTS);
```

```
> DELETE FROM HISTOGRAM_INTERVALS;
```

```
> INSERT INTO HISTOGRAM_INTERVALS SELECT * FROM myhistint
where   table_uid in (select object_uid from
CAT.DEFINITION_SCHEMA_VERSION_3000.OBJECTS);
```

4 Reviewing Query Execution Plans

Use the information in this section to display and understand how your query is optimized. At a later time, you might use this information to make decisions about forcing execution plans, described in Section 5, Forcing Execution Plans.

Displaying Execution Plans

Use these methods to display the query execution plan:

- Use the EXPLAIN function to query and display certain columns or all columns of information about execution plans.
- Use the EXPLAIN statement shortcut to display all available information about execution plans.
- Use the Visual Query Planner graphical user interface (GUI) to extract and display execution plans generated by the SQL/MX optimizer for DML statements.

When you view information using the EXPLAIN function, the results are displayed in the machine-readable format. However, when you use the EXPLAIN statement to view the information, the results are displayed in the machine-readable format. Although you can interpret the results of the execution plans, you can view results better using the Visual Query Planner application.

Using the EXPLAIN Function

The EXPLAIN function is a table-valued stored function that returns information about execution plans for SQL DML statements. You can use the information in the EXPLAIN output for these types of tasks:

- Reviewing the chosen execution plan.
- Identifying problems and tuning queries.
- Determining whether the optimizer chose the optimal plan.

You can query and display certain columns or all columns of the information about execution plans by using the EXPLAIN function.

You can selectively display columns from the execution plan by using the EXPLAIN function:

1. In MXCI, prepare the query:

   ```
   PREPARE FINDSAL FROM
     SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEE
     WHERE SALARY > 40000.00;
   ```

2. Display selected rows of the execution plan for the prepared statement FINDSAL by using the EXPLAIN function:

   ```
   SELECT SEQ_NUM, OPERATOR, TOTAL_COST
     FROM TABLE (EXPLAIN (NULL, 'FINDSAL'));
   ```

   Output results from the query are:

   ```
   SEQ_NUM      OPERATOR                      TOTAL_COST
   -----------  ----------------------------  --------------
             1   FILE_SCAN                    2.1645594E-002
             2   PARTITION_ACCESS             2.1645594E-002
             3   ROOT                         1.1964559E-001

   --- 3 row(s) selected.
   ```

For a description of all column fields of the EXPLAIN function, see Description of the EXPLAIN Function Results on page 4-3.

For a detailed description of the EXPLAIN function, see Displaying Selected Columns of the Execution Plan on page 4-5.

Using the EXPLAIN Statement Shortcut

You can display all the columns of the execution plan using the EXPLAIN statement. This command is a shortcut for the EXPLAIN function.

In MXCI, enter the `EXPLAIN` statement followed by your query. For example:

```
EXPLAIN
SELECT FIRST_NAME, LAST_NAME FROM EMPLOYEE
  WHERE SALARY > 40000.00;
```

You can also use the EXPLAIN statement to display the plan for a prepared query.

For more information about the EXPLAIN statement, see Using the EXPLAIN Statement to Review the Execution Plan on page 4-6.

Using the Visual Query Planner

1. To start the Visual Query Planner, do one of the following:
   - Select **Start > Programs > NonStop SQL-MX >Visual Query Planner**.
   - Navigate to the C:\Program Files\Hewlett Packard\NonStop SQL-MX folder and select **Visual Query Planner**.
2. Select **Explain > Connect to ODBC** to connect to your ODBC data source.

For further information about creating ODBC data sources, see the *ODBC Driver Manual for Windows*.

3. In the top pane of the **Visual Query Planner** window, enter your query.
4. Select **Explain > Get Explain Plan** to display your query execution plan.

   The operator tree for the query execution plan appears in the lower left pane of the Visual Query Planner window. Summary detail for the operators displays in the lower right pane. Additional information about each operator is included in the Properties dialog box, which is available from the **Explain** menu.

For a detailed description of the Visual Query Planner, see Using the Visual Query Planner on page 4-14.

The Optimizer and Executor

The optimizer creates an execution plan for each operator in the operator tree for the query. The cost is an estimate of the resources needed to execute a plan. Each operator has a local cost, which is dependent on the algorithm chosen for implementing the operator and the values that the operator receives as input. The local cost is the cost for the leaf node. From then on, the local cost is the cumulative cost of that branch of the node.

The execution plan is chosen based on what is determined to be the optimal performance plan: the total cost to produce the first row of output or last row of output. Total cost is estimated by using a number of formulas to roll up the local costs. The formulas depend on the characteristics of each of the operators in the tree.

Description of the EXPLAIN Function Results

An operator tree is a structure that represents operators used in a query execution plan as nodes, with at most one parent node for each node in the tree, and with only one root node. Each row in the EXPLAIN output corresponds to one node in the tree. A node of an operator tree is a point in the tree that represents an event (involving an operator) in a plan. Each node might have subordinate nodes—that is, each event might generate a subordinate event or events in the plan. For a graphical view of an operator tree, see Figure 4-2 on page 4-16.

The next table lists the columns, data types, and descriptions for each item in the EXPLAIN result table:

Column Name	Data Type	Description
MODULE_NAME	CHAR(60)	Module name as specified in the argument to the EXPLAIN function; NULL for dynamic SQL statements (prepared statements); truncated on the right if longer than 60 characters.
STATEMENT_NAME	CHAR(60)	Statement name after wild-card expansion; truncated on the right if longer than 60 characters.

Column Name	Data Type	Description
PLAN_ID	LARGEINT	Unique system-generated plan ID automatically assigned by SQL; generated at compile time.
SEQ_NUM	INT	Sequence number of the current node in the operator tree; indicates the sequence in which the operator tree is generated.
OPERATOR	CHAR(30)	Current node type. For a full list of valid operator types, see Section 7, SQL/MX Operators.
LEFT_CHILD_ SEQ_NUM	INT	Sequence number for the first child operator of the current node (or operator); null if node has no child operators.
RIGHT_CHILD_ SEQ_NUM	INT	Sequence number for the second child operator of the current node (or operator); null if node does not have a second child.
TNAME	CHAR(60)	For operators in scan group, full name of base table, truncated on the right if too long for column. If correlation name differs from table name, simple correlation name first and then table name in parentheses.
CARDINALITY	REAL	Estimated number of rows that will be returned by the current node.
OPERATOR_COST	REAL	Estimated cost associated with the current node to execute the operator.
TOTAL_COST	REAL	Estimated cost associated with the current node to execute the operator, including the cost of all subtrees in the operator tree.
DETAIL_COST	VARCHAR (200)	Cost vector of five items, which are described in detail in the next table.
DESCRIPTION	VARCHAR (3000)	Additional information about the operation in the form of a stream of token pairs. For a detailed look at the tokens for all operators, see Section 7, SQL/MX Operators.

The DETAIL_COST column of the EXPLAIN function results contains these cost factors:

CPU_TIME	An estimate of the number of seconds of processor time it might take to execute the instructions for this node. A value of 1.0 is 1 second.
IO_TIME	An estimate of the number of seconds of I/O time (seeks plus data transfer) to perform the I/O for this node.
MSG_TIME	An estimate of the number of seconds it takes for the messaging for this node. The estimate includes the time for the number of local and remote messages and the amount of data sent.

IDLETIME	An estimate of the number of seconds to wait for an event to happen. The estimate includes the amount of time to open a table or start an ESP process.
PROBES	The number of times the operator will be executed. Usually, this value is 1, but can be greater when you have, for example, an inner scan of a nested-loop join.

Displaying Selected Columns of the Execution Plan

This query execution plan example is a simple query using the PERSNL schema and querying against the EMPLOYEE table:

```
PREPARE s1 FROM SELECT LAST_NAME, FIRST_NAME, SALARY
  FROM EMPLOYEE ORDER BY SALARY;
```

In this query, the EXPLAIN function extracts these columns: SEQ_NUM, OPERATOR, TOTAL_COST, and DESCRIPTION:

```
SELECT SEQ_NUM, OPERATOR, TOTAL_COST, DESCRIPTION
  FROM TABLE (EXPLAIN (NULL, 'S1'));
```

This output is formatted for readability. The EXPLAIN function and EXPLAIN statement use machine-readable format for application program access.

```
1  FILE_SCAN                       2.0646531E-002
scan_type: file_scan SAMDBCAT.PERSNL.EMPLOYEE
scan_direction: forward
key_type: simple
lock_mode: not specified
access_mode: not specified
columns_retrieved: 6
fast_scan: used
fast_replydata_move: used
key_columns: indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM)
begin_key: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM) = 0)
end_key: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM) = 9999)

2  PARTITION_ACCESS                2.0646531E-002
buffer_size: 31000
record_length: 44

3  SORT                            1.4921140E-001

4  ROOT                            2.4723619E-001
statement_index: 0
statement: select last_name, first_name, salary from
samdbcat.persnl.employee
order by salary;
return select_list:
indexcol(SAMDBCAT.PERSNL.EMPLOYEE.LAST_NAME),
indexcol(SAMDBCAT.PERSNL.EMPLOYEE.FIRST_NAME),
indexcol(SAMDBCAT.PERSNL.EMPLOYEE.SALARY)
order_by: indexcol(SAMDBCAT.PERSNL.EMPLOYEE.SALARY)
```

Four nodes (operators) appear in the plan: FILE_SCAN, PARTITION_ACCESS, SORT, and ROOT. Operators are listed under Section 7, SQL/MX Operators.

This query requires a full table scan (FILE_SCAN), because the query selected all employees, and required a SORT, because employee names are ordered by salary.

Four rows have been selected. The EXPLAIN function returns one row for each operator used to evaluate the query.

Note that the description field contains different information for each operator. The description for the FILE_SCAN node indicates the scan type and file name of the input, the begin and end keys, the read direction, and other unspecified file access information. The description for the SORT node indicates the key for sorting. The description for the ROOT node lists all the columns in the SELECT statement list, the ordering by column for the output, and the SQL statement. For full details of the description column for each operator, see Section 7, SQL/MX Operators.

Extracting EXPLAIN Output From Embedded SQL Programs

You can easily extract the EXPLAIN output from embedded SQL programs. You must supply the module name, which you can supply in the program itself with a MODULE clause, or you can check for the name in the module definition file. For example, suppose that the module name of your embedded program is MYCAT.MYSCH.MYMOD.

If your program contains multiple statements, this query explains all statements in the order that they appear in your module:

```
SELECT * FROM TABLE (EXPLAIN('MYCAT.MYSCH.MYMOD', '%'));
```

You can review the EXPLAIN output any time after you have compiled the program, and you will see the plan that was actually chosen at compile time. Note that the second argument to the EXPLAIN is a LIKE pattern that is used on the statement-name column.

Using the EXPLAIN Statement to Review the Execution Plan

The next example of the EXPLAIN statement shows the information in the execution plan for a query that uses predicates. The EXPLAIN statement displays all the columns for the execution plan.

```
>>EXPLAIN
+>SELECT last_name, first_name, salary
+>FROM employee WHERE
+>salary > 40000.00 AND jobcode=450;
```

When you use the EXPLAIN statement, you must page down to see the entire output:

```
MODULE_NAME
STATEMENT_NAME
PLAN_ID
SEQ_NUM
OPERATOR
```

```
LEFT_CHILD_SEQ_NUM
RIGHT_CHILD_SEQ_NUM
TNAME
CARDINALITY  OPERATOR_COST  TOTAL_COST  DETAIL_COST
DESCRIPTION

?
EXPL_NAME__
211977924124011276           1  FILE_SCAN      ?             ?
SAMDBCAT.PERSNL.EMPLOYEE
2.0000000E+000   2.0646531E-002   2.0646531E-002
CPU_TIME: 0.000287
IO_TIME: 0.020647
MSG_TIME: 0
IDLETIME: 0
PROBES: 1
scan_type: file_scan SAMDBCAT.PERSNL.EMPLOYEE
scan_direction: forward
key_type: simple
lock_mode: not specified
access_mode: not specified
columns_retrieved: 6
fast_scan: used
fast_replydata_move: used
key_columns: indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM)
executor_predicates: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.SALARY)
40000.00) and (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.JOBCODE) = 450)
begin_key: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM) = 0)
end_key:(indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM)=9999)

?
EXPL_NAME__
211977924124011276       2  PARTITION_ACCESS        1
?
2.0000000E+000   2.0422223E-003   2.0646531E-002
CPU_TIME: 0.002329
IO_TIME: 0.020647
MSG_TIME: 0
IDLETIME: 0
PROBES: 1
buffer_size: 31000
record_length: 44

?
EXPL_NAME__
211977924124011276        3  ROOT                  2
?
2.0000000E+000   9.8000802E-002   1.1864653E-001
CPU_TIME: 0.002433
IO_TIME: 0.020647
MSG_TIME: 0
IDLETIME: 0.098
PROBES: 1
statement_index: 0
statement:  SELECT last_name, first_name, salary from
```

```
samdbcat.persnl.employee
where salary > 40000.00 and jobcode=450;
return select_list:
indexcol(SAMDBCAT.PERSNL.EMPLOYEE.LAST_NAME),
 indexcol(SAMDBCAT.PERSNL.EMPLOYEE.FIRST_NAME),
indexcol(SAMDBCAT.PERSNL.EMPLOYEE.SALARY)

--- SQL operation complete.
```

This output for the FILE_SCAN operator is formatted for clarity of illustration:

Column Name	**EXPLAIN statement Output**
MODULE_NAME	`Null`
STATEMENT_NAME	`__EXPL_NAME`
PLAN_ID	`211977924124011276`
SEQ_NUM	`1`
OPERATOR	`FILE_SCAN`
LEFT_CHILD_ SEQ_NUM	`Null`
RIGHT_CHILD_ SEQ_NUM	`Null`
TNAME	`SAMDBCAT.PERSNL.EMPLOYEE`
CARDINALITY	`2.0000000E+000`
OPERATOR_COST	`2.0646531-002`

Column Name	EXPLAIN statement Output
TOTAL_COST	`2.0646531-002`
DETAIL_COST	`CPU_TIME: 0.000287 IO_TIME: 0.020647 MSG_TIME: 0 IDLETIME: 0 PROBES: 1`
DESCRIPTION	`scan_type: file_scan SAMDBCAT.PERSNL.EMPLOYEE scan_direction: forward key_type: simple lock_mode: not specified access_mode: not specified columns_retrieved: 6 fast_scan: used fast_replydata_move: used key_columns: indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM) executor_predicates: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.SALARY) 40000.00) and (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.JOBCODE) = 450) begin_key: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM) = 0) end_key: (indexcol(SAMDBCAT.PERSNL.EMPLOYEE.EMPNUM) = 9999)`

Optimization Tips

NonStop SQL/MX relies on system-defined default settings for hundreds of attributes. Most of these settings are finely tuned already. However, some attributes have values that you can change.

You can change the default settings for externalized attributes in the SYSTEM_DEFAULTS table in the system catalog. The settings are stored in the `NONSTOP_SQLMX_<systemname>.SYSTEM_DEFAULTS_SCHEMA.SYSTEM_DEFAULTS` table. You can also use the CONTROL QUERY DEFAULT command to change the values on a per-process basis. For information about the precedence of the commands and the ways you can specify the default settings, in addition to a listing of all externalized attributes, see the *SQL/MX Reference Manual*.

This subsection provides information about certain optimization-related externalized attributes. For reference information about each attribute, see the *SQL/MX Reference Manual*.

- GEN_EIDR_BUFFER_SIZE and GEN_PA_BUFFER_SIZE

  These default settings determine the buffer size for PARTITION_ACCESS operators. The settings must match for both attributes. These attributes provide suggestions to the compiler, which will adjust the size of the buffer as necessary.

For OLTP queries that use OLT optimization, the settings are not relevant because SQL determines the buffer size depending on the row size.

For non-OLT optimized queries that return or insert multiple rows, the buffer size is set to the default value to maximize the number of rows returned. If the actual number of rows is smaller than the default value, only the actual bytes are shipped. The EXPLAIN function output for the PARTITION_ACCESS operator shows the size of the buffer that SQL choses and any overrides you caused by setting the CQDs. Although the maximum size is 31 kilobytes for remote nodes and 56 kilobytes for local nodes, SQL only uses as many bytes as necessary to send the number of rows in that buffer. If you set a large buffer size, memory consumption is affected; however, messaging is not affected. If you know a query will return only a few rows, you can lower the buffer size. If you reduce the buffer size to a small value and the number of rows returned does not fit, more messages and buffers will be shipped between DP2 and the executor.

You might want to analyze whether the buffer size is appropriate and whether any changes you make to the buffer size cause more messaging to occur. You can use the Measure product and the DISPLAY STATISTICS command to make these decisions. For information about the Measure product, see the *Measure User's Guide*. For information about DISPLAY STATISTICS, see the *SQL/MX Reference Manual*.

- JOIN_ORDER_BY_USER

  This attribute provides an easy alternative to CONTROL QUERY SHAPE that you can use when you want to specify the order of the tables but not the join type. The first table specified in the FROM clause is the outer table in the query.

- OPTIMIZATION_LEVEL

  The settings for this attribute indicate increasing effort in optimizing SQL queries:

Optimization Level	**Description**
0	The compiler optimization effort at this level is minimal (it uses heuristics to perform one-pass optimization for the shortest possible compilation time). This level is most suitable for small tables or when plan quality is not important.

2	The compiler uses a combination of heuristics and limited search space enumeration. In general, this level produces good quality plans, although it might miss a globally optimal plan. The compilation time is significantly shorter in comparison with the higher optimization levels (3 and 5).
3 (default)	This default optimization level is recommended for general use. The compiler performs a thorough two-pass optimization. For more complex queries (above 5-way joins), it uses a combination of heuristics, search space enumeration and compile time controlling algorithms to find the best possible plan within a reasonable compile time. For simple queries (5-way joins and below), the compiler does an exhaustive search. The compiler compiles plans for up to 40-way joins. The generated plans are significantly better than the plans produced with optimization level 0 and, in general, better than the plans generated at level 2. To get faster query compilation time, you might consider optimization 2. To perform a complete exhaustive search, which might lead to a better plan but requires longer compilation time, you might consider optimization 5.
5	The compiler performs a full optimization of the query using exhaustive search algorithms. The resulting plan is a globally optimal one, but the trade off is that it might take a long compilation to find it. For complex queries (11-way joins and higher), the compilation time can be prohibitively long. This compilation time threshold is even longer (9-way joins) for queries that involve semi-joins, outer joins, or subqueries.

Note. Optimization level values must be either 0, 2, 3 or 5. Values outside this range return an error 2055 (invalid value for DEFAULTS attribute OPTIMIZATION_LEVEL). Values 1 and 4 are currently reserved and should not be specified. If you specify these values, the compiler replaces the value with the next lowest optimization level. For example, 4 is replaced by 3, and 1 is replaced by 0.

To maintain compatibility, the compiler accepts the previous optimization settings of MINIMUM, MEDIUM and MAXIMUM. The values are mapped as follows:

SQL/MX Release 1.x	**SQL/MX Release 3.0**
MINIMUM	0
MEDIUM	3
MAXIMUM	5

- OPTS_PUSH_DOWN_DAM

For compound statements, the predicates for each statement must identify the DAM process so that the single DAM process is identified by the compiler. Remember that NonStop SQL/MX requires more than one statement within a compound statement, so the strategy of placing a single SELECT statement between a BEGIN and END statement to push down the SELECT to the DAM process does not work.

The first statement contained in the compound statement:

- Cannot be an INSERT statement with a VALUES clause.
- If the statement is a SELECT or an UPDATE statement, it cannot have rowset host variables in the WHERE clause.

For statements (after the first statement) contained in the compound statement:

- If the statement is an INSERT statement, it cannot have rowset host variables corresponding to partition key columns inside the VALUE clause.
- If the statement is a SELECT or UPDATE statement, it cannot have rowset host variables corresponding to partition key columns in the WHERE clause.

For all statements contained in compound statements:

- All partition key columns should be covered by the same set of scalar (nonrowset) host variables so that the compiler can determine that only one partition can be accessed during run time.
- Aggregates are not allowed.

Nested joins between no more than three tables and indexes can be considered for pushing down to DAM. This option can be used for OLTP type of queries touching a few blocks of participating tables. This option is not recommended for complex DSS type queries for two reasons:

- ○ It could considerably increase compile time because of increasing optimization search space.
- ○ Concurrent scans on the same physical volume could cause lots of extra unexpected seeks and query performance could degrade.

When pushing a plan down to DAM is possible (because you have correctly identified the DAM process in the predicates, and the OPTS_PUSH_DOWN_DAM attribute value is set to ON), NonStop SQL/MX might not push down the plan because of the plan cost. The system-defined default setting (OFF) means that NonStop SQL/MX does not attempt to push down.

You can verify if statements have been pushed down to DAM by reviewing the EXPLAIN output for the plan. If the plan shows the PARTITION_ACCESS operator, DAM access is being used, and the operators below the PARTITION_ACCESS operator have been pushed down to DAM.

For a discussion of compound statements, see the *SQL/MX Programming Manual for C and COBOL.*

- REMOTE_ESP_ALLOCATION

Use the OFF setting to force NonStop SQL/MX to bring up all ESPs on the local system only. Consider this setting in these cases:

- If the network connection is slow and you want to reduce network traffic (depending on the partition distribution of the tables in the query)
- You want to reduce the consumption of memory on remote systems
- If joins increase the number of rows produced

This setting also forces NonStop SQL/MX to try to achieve maximum parallelism without consideration for communication across the network.

Use the SYSTEM setting when you want to use all active nodes. Consider this setting in these cases:

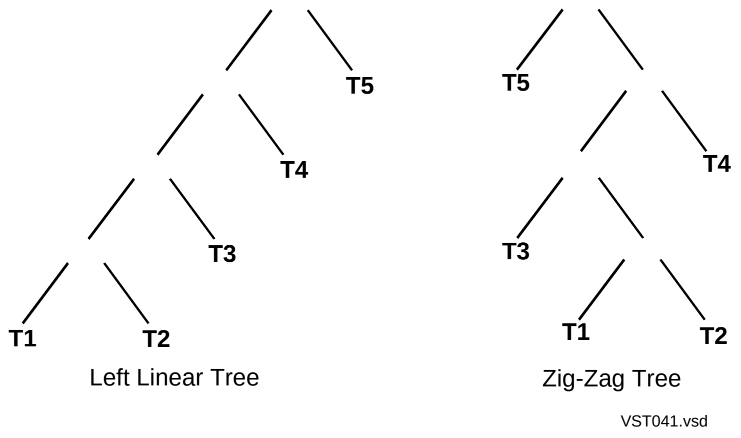
- Nodes where partitions exist on at least two tables
- Nodes where partitions exist on a single table

Use the ON setting to use all nodes where partitions reside.

- ZIG_ZAG_TREES

  In a left linear tree, the right child is always a single table or subquery, and the left child is a subtree of one or more tables. In a zig-zag tree, one child is always a table, and the other child is a subtree of one or more tables, usually formed in zig-zag fashion. When the default value is set to OFF (the default setting), the optimizer examines predominately left linear trees and only a few promising zig-zag trees. When the default is set to ON, the optimizer is directed to enumerate larger numbers of zig-zag trees in the search for the optimal plan. Figure 4-1 on page 4-13 shows left linear and zig-zag trees.

Figure 4-1. Left Linear and Zig-Zag Trees



Verifying DAM Access

You can verify if operators are executing in DAM by reviewing the EXPLAIN output for the plan. If the plan shows the PARTITION_ACCESS operator, DAM access is being used. Operators appearing below the PARTITION_ACCESS operator are executing in DAM.

Using the Visual Query Planner

Visual Query Planner is a Microsoft Windows NT GUI application that enables you to extract and display query execution plans generated by the SQL/MX optimizer for DML statements. Visual Query Planner uses the EXPLAIN function internally to extract the information about the query plan.

You can save plans that you generate for further analysis. Saved plans can be sent to anyone who can look at them without having to access the actual database. You can also provide plans to the Global Customer Support Center (GCSC) in case of problems. Visual Query Planner also provides the ability to force the optimizer to present the plan that you design. For further information, see Section 5, Forcing Execution Plans.

Visual Query Planner Requirements

In addition to NonStop SQL/MX, Visual Query Planner requires that you have data sources defined for ODBC. You must have a PC or workstation with Windows NT 4.0 (or later) to use Visual Query Planner. For information about installing Visual Query Planner, see the *SQL/MX Installation and Management Guide*. For information about defining ODBC data sources with the Microsoft ODBC Data Source Administrator, see the *ODBC Driver for Windows Manual.*

Note. To avoid unexpected results, turn off query caching at the start of your VQP session.

Getting Help for Visual Query Planner

To access the Visual Query Planner online help facility, select **Help** > **Visual Query Planner Help**. Access additional context-sensitive help by pressing F1 or through the **Properties** dialog box. For more information, see Accessing Additional Information About Operators on page 4-17.

Graphically Displaying Execution Plans

1. To start the Visual Query Planner, do one of the following:

   - Select **Start > Programs > NonStop SQL-MX > Visual Query Planner.**
   - Navigate to the C:\Program Files\Hewlett Packard\NonStop SQL-MX folder and select and open **Visual Query Planner**.

     After the program starts, you will notice that a default file name, VQP1, appears on the title bar and that status messages appear along the bottom of the window.

2. Select **Explain > Connect to ODBC** to connect to your ODBC data source.

   Note. MXCS must be running on the server before you can connect Visual Query Planner to MXCS.

   For more information about connecting to MXCS, see the *SQL/MX Connectivity Services Manual.*

   After you are connected to your data source, notice that the status message along the bottom of the window displays the data source name. Also notice that the **Connect to ODBC** icon on the toolbar is dimmed, indicating that the connection has been made.

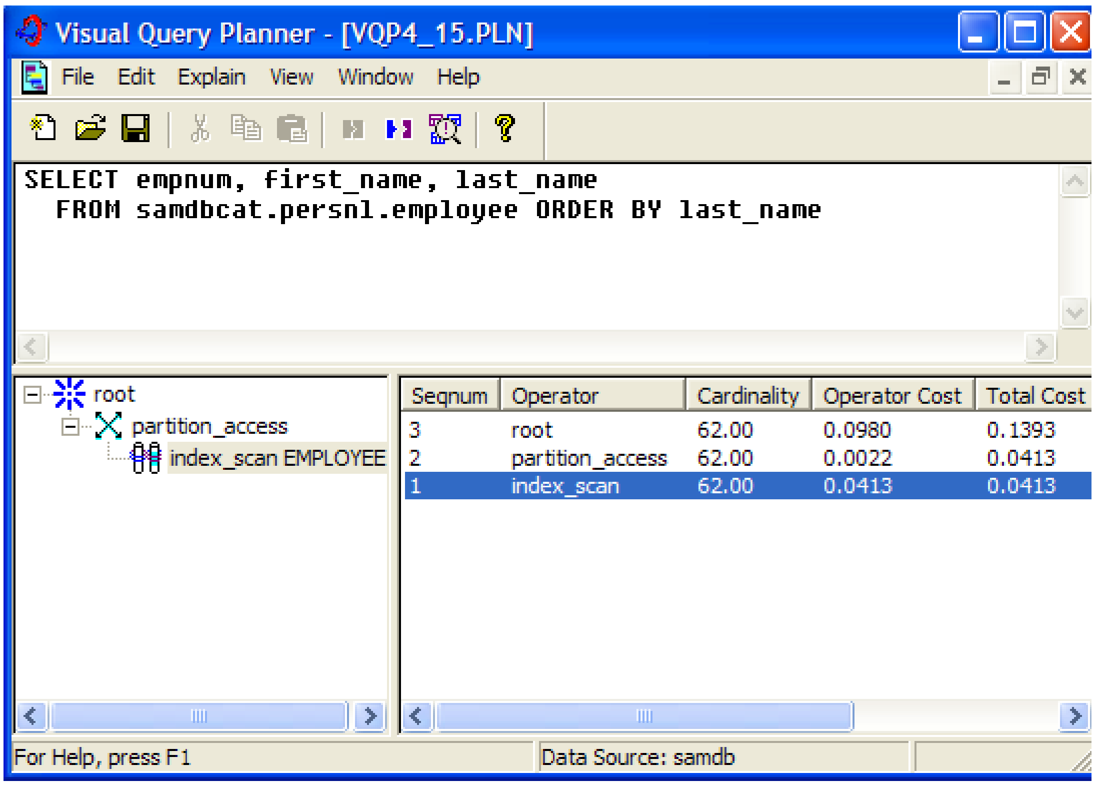
3. In the top pane of the **Visual Query Planner** window, enter your query.

   You must enter (or copy) the query rather than trying to execute a prepared query. Visual Query Planner internally prepares the query and assigns it a statement name.

4. Select **Explain > Get Explain Plan** to direct the EXPLAIN function to prepare and display the query.

   The operator tree for the query execution plan appears in the lower left pane of the window (operator tree pane). Summary details for the operators appear in the right lower pane of the window (summary detail pane). You can sort the summary detail by clicking a column name to sort on that column. Figure 4-2 on page 4-16 shows a query in the top pane and the subsequent query execution plan in the left lower pane of the window.

Figure 4-2. Visual Query Planner



```
SELECT empnum, first_name, last_name
  FROM samdbcat.persnl.employee ORDER BY last_name
```

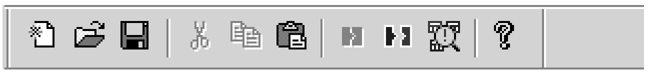
Seqnum	Operator	Cardinality	Operator Cost	Total Cost
3	root	62.00	0.0980	0.1393
2	partition_access	62.00	0.0022	0.0413
1	index_scan	62.00	0.0413	0.0413

vst501.vsd

5. Select **File > Save** to save your query execution plan. The default name for the session appears in the **Save** dialog box. If you choose to provide a different name for the query execution plan, the new name appears on the title bar.

Selecting Tasks on the Toolbar

You can also select tasks on the toolbar positioned immediately below the menu bar.



VST502.vsd

Click the icon buttons to:

	Create a new file
	Open an existing file
	Save a file

Cut text within the edit window

Copy text within the edit window

Paste text within the edit window

Connect to an ODBC source

Disconnect from the ODBC data source

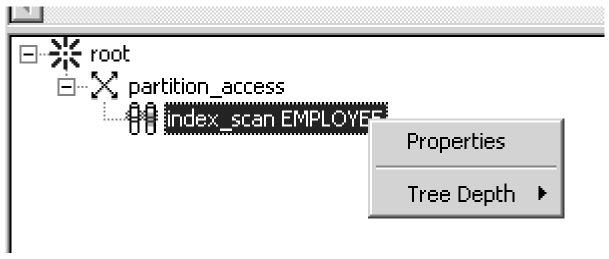
Execute the query

Open help

These tasks are also available from the menus.

Accessing Additional Information About Operators

To access additional information about each operator in the **Properties** dialog box, right-click an operator entry. A dialog box appears that contains **Properties** and **Tree Depth,** shown next. You can also access the **Properties** dialog box by double-clicking an operator or by selecting **Explain > Properties**.

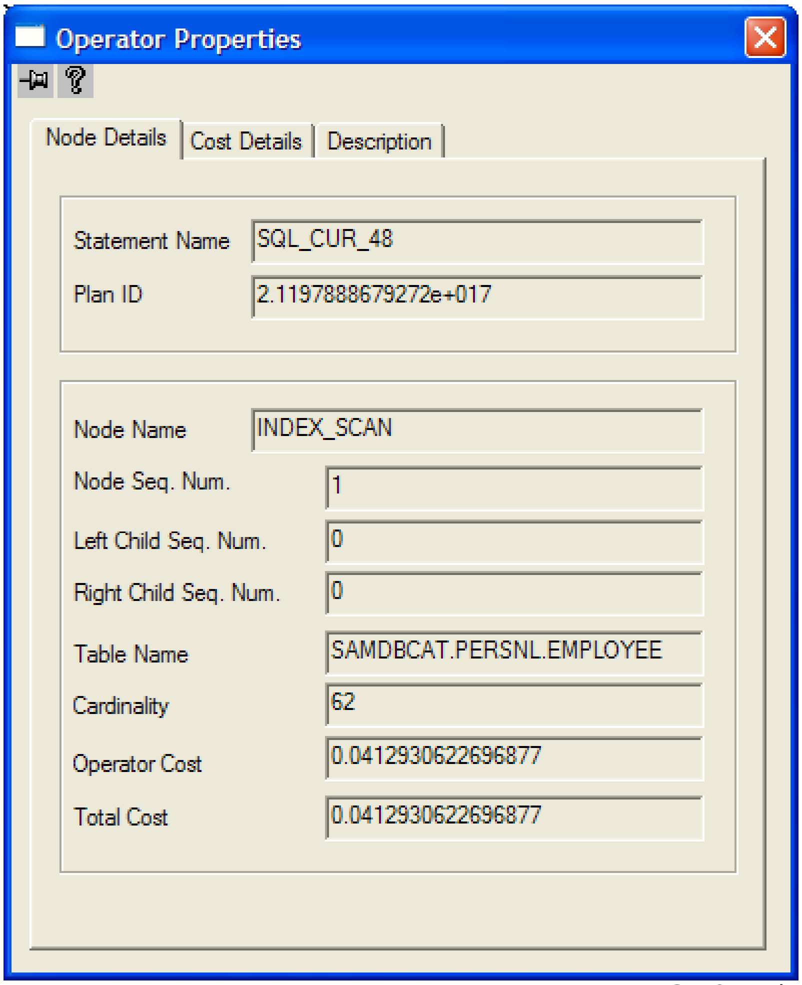


VST503.vsd

Note. *Tree depth* refers to the number of levels you want to display in the operator tree. A fully expanded operator tree appears when you select **All Levels**.

The **Operator Properties** dialog box, shown next, provides three tabs:

- Node details
- Cost details
- Description



Operator Properties

Statement Name	SQL_CUR_48
Plan ID	2.1197888679272e+017
Node Name	INDEX_SCAN
Node Seq. Num.	1
Left Child Seq. Num.	0
Right Child Seq. Num.	0
Table Name	SAMDBCAT.PERSNL.EMPLOYEE
Cardinality	62
Operator Cost	0.0412930622696877
Total Cost	0.0412930622696877

VST504.vsd

Notice the pushpin and ? icon in the upper left corner of the dialog box. You can pin the dialog box open by selecting the pushpin so that you can select on another operator without having to reopen the Properties dialog box. Select the ? icon for context-sensitive help for the operator. The tabs in the **Operator Properties** dialog box are described next.

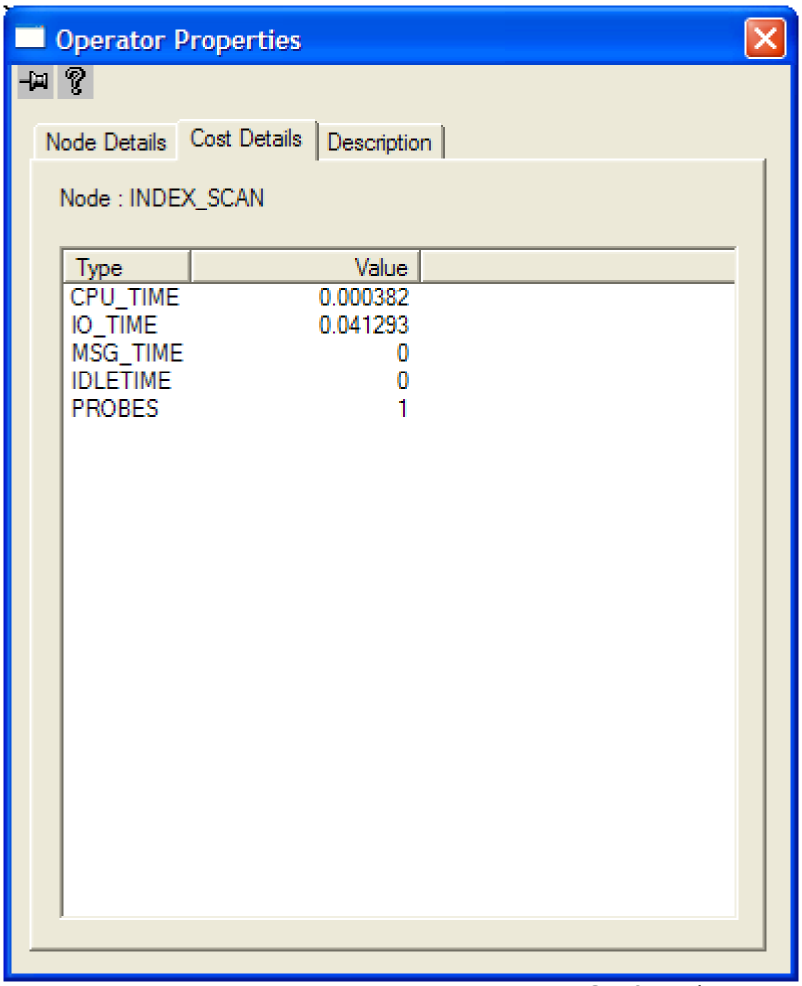
Node Details

- **Statement Name** is the name assigned to the prepared query; in this case, Q2.
- **Plan ID** is a unique identifier generated during compile time.
- **Node Name** describes the operator type; in this case, INDEX_SCAN.

- **Node Seq. Num.** is 1, which indicates that the node was first in sequence during optimization.
- **Left Child Seq. Num.** is 0, which indicates that this is not a join.
- **Right Child Seq. Num.** is 0, which indicates that this is not a join. If both the left and right child sequence numbers contain values, the node is a join.
- **Table Name** provides the *`catalog.schema.tablename`* of the table scanned.
- **Cardinality** provides the estimated rows returned by the INDEX_SCAN.
- **Operator Cost** provides the estimated cost associated with the current node to execute the INDEX_SCAN.
- **Total Cost** provides the estimated cost associated with the current node to execute the INDEX_SCAN, including the cost of all subtrees in the operator tree.

Reviewing the Cost Details

The items listed in the **Cost Details** tab are the same as those items described in the DETAIL_COST column of the EXPLAIN function results. For a detailed description of each of these items, see Description of the EXPLAIN Function Results on page 4-3.



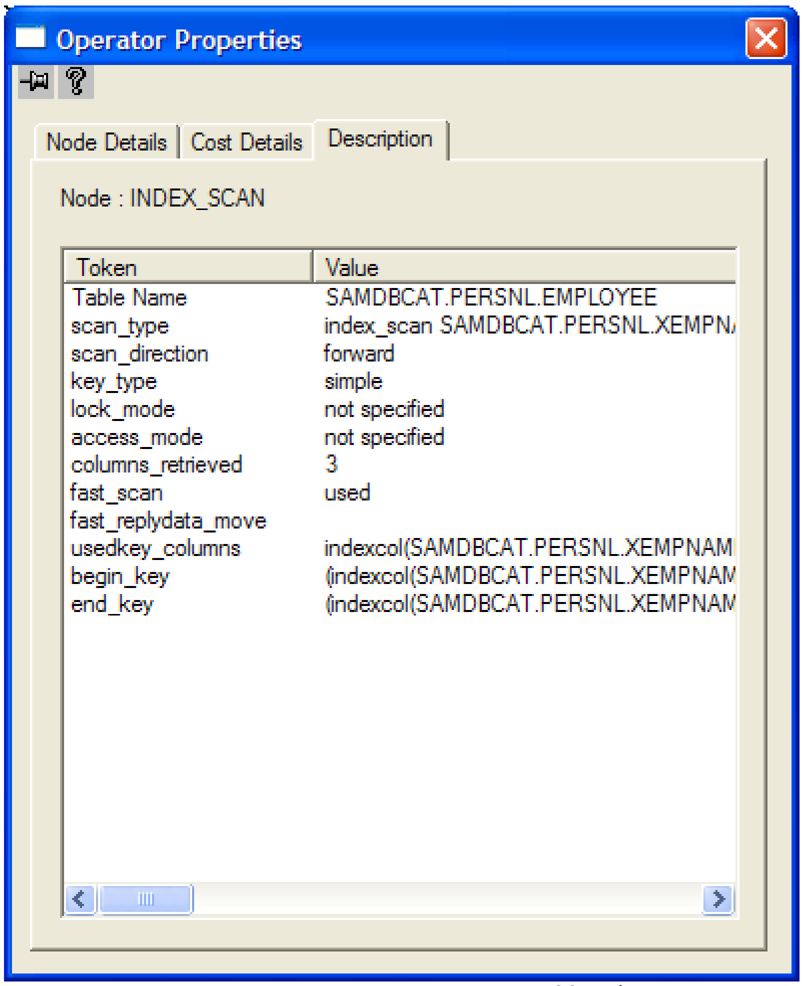
Node : INDEX_SCAN

Type	Value
CPU_TIME	0.000382
IO_TIME	0.041293
MSG_TIME	0
IDLETIME	0
PROBES	1

VST505.vsd

Additional Table Information

The **Description** tab provides additional table information, including key columns, scan type, scan direction, and so on.



Node : INDEX_SCAN

Token	Value
Table Name	SAMDBCAT.PERSNL.EMPLOYEE
scan_type	index_scan SAMDBCAT.PERSNL.XEMPN/
scan_direction	forward
key_type	simple
lock_mode	not specified
access_mode	not specified
columns_retrieved	3
fast_scan	used
fast_replydata_move	
usedkey_columns	indexcol(SAMDBCAT.PERSNL.XEMPNAM
begin_key	(indexcol(SAMDBCAT.PERSNL.XEMPNAM
end_key	(indexcol(SAMDBCAT.PERSNL.XEMPNAM

vst506.vsd

The token descriptions for each operator are described in Section 7, SQL/MX Operators.

Reviewing Run-Time Statistics

NonStop SQL/MX provides statistics for an executed query. Use the DISPLAY STATISTICS command to view statistics.

The DISPLAY STATISTICS command is described in the *SQL/MX Reference Manual*.

Simple Query Example

This query selects all rows and columns from the EMPLOYEE table:

```
>>prepare q1 from
+>select * from employee;

--- SQL command prepared.

>>execute Q1;
```

The query returns this result:

```
EMPNUM  FIRST_NAME   LAST_NAME    DEPTNUM  JOBCODE  SALARY
------  -----------  ----------   -------  -------- ------
     1   ROGER        GREEN          9000      100   175500.00
             .            .             .        .          .
   568   JESSICA      CRINER         3500      300    39500.00


--- 62 row(s) selected.
```

To obtain statistics about the query, after the query executes, enter:

```
>>DISPLAY STATISTICS;
```

The statistics are returned:

```
Start Time      2005/11/04 10:45:53.916
End Time        2005/11/04 10:45:53.975
Elapsed Time               00:00:00.059
Compile Time               00:00:00.000
Execution Time             00:00:00.059


Table Name   Records     Records    Disk   Message      Message  Lock
              Accessed        Used    I/Os     Count        Bytes

SAMDBCAT.PERSNL.EMPLOYEE

                    62          62       2         2        14608     0
```

If you want to display statistics automatically, enter SET STATISTICS ON at any point during the session. As a result, the statistics appear immediately after each command executes.

Using Measure

Use the Measure product to gather statistics on an SQL/MX database and application programs. The Measure product provides statistics on process execution (SQLPROC) and on individual statement execution (SQLSTMT). For information on using Measure with NonStop SQL/MX, see the *SQL/MX Installation and Management Guide*. For information about the Measure product, see the *Measure Reference Manual*.

Process Execution (SQLPROC)

For each process, the Measure product provides these statistics:

- The number of times static SQL statements were recompiled and the elapsed time needed for recompilation
- The number of times the executor server processes were started up and the elapsed time to do this
- The number of open requests issued by SQL and the elapsed time to do this

Statement Execution (SQLSTMT)

The SQLSTMT report provides information for specific statements of modules executed by an SQL process. For each statement, the Measure product provides these statistics:

- The number of times the statement was executed
- The total elapsed time to execute the statement
- The number of rows accessed and returned or altered
- The number of disk reads needed for execution
- The number and length of messages sent to execute the statement
- The number of sorts performed and the elapsed time to do them
- The number of recompilations and the elapsed time to do them
- The number of timeouts, lock escalations, and lock waits

SQLSTMT entities gather statistics for all statements of a process selected for measurement; there is one SQLSTMT entity for each statement. The SQLSTMT report identifies the SQLSTMT section for each statement by the module name and the statement index. A statement index identifies each SQL statement that can be measured. This number appears in the generated SQL module definition file, and you can use it to look up the corresponding Measure SQLSTMT counters.

Note. The run unit reported by Measure is the name of the procedure or COBOL run unit for NonStop SQL/MP. In NonStop SQL/MX, the run unit is the name of the SQL module. The length of the name is 128 bytes.

Evaluating Measure Data

Use the SQLSTMT report to form a baseline performance picture, which you can then use to compare to subsequent versions as you tune your queries.

Note. To translate the statement index back into SQL statements, you can also use the EXPLAIN function. See the statement_index and statement tokens for the PROBE_CACHE Operator on page 7-59.

Optimally, measure each transaction or query in isolation. Otherwise, you do not get a clear view of the transaction of interest. If you do not know which of several transactions is performing poorly, you can execute each transaction separately, measure it, and compare performance among the group of transactions.

When reviewing the SQLSTMT reports for poorly performing queries, examine and isolate queries based on the number of I/O operations, total time consumed relative to other queries, frequency of execution within a single transaction, and other performance-related measurements. Then, generate query plans with the Explain function for the queries to help identify reasons for poor performance. Sometimes a specific type of problem is common to a set of queries.

Stopwatch measurements can also be helpful. When compared to Measure information, they can reveal network problems or other types of delays.

It can be important to establish response-time requirements for specific queries. This strategy permits identification of a specific goal and completion framework for tuning.

When evaluating changes to queries, consider other transactions that might be adversely affected by the change. For example, if you add an index, compare performance before and after for INSERT and UPDATE transactions. Consider the volume of the query being addressed and compare it with the volume of UPDATE and DELETE transactions.

5 Forcing Execution Plans

Use the information in this section to make decisions about forcing query execution plans.

The SQL/MX optimizer attempts to generate the most cost-efficient plan available. In some situations, you might find that you want to vary the plan selected by the optimizer.

△ **Caution.** If you use the CONTROL QUERY SHAPE statement, you can override the optimizer's standard cost estimates and cause negative performance. In addition, if you try to force an invalid plan, no plan will be returned to you. Use CONTROL QUERY SHAPE only if the optimizer does not produce the optimal plan. You might want to contact your service provider for assistance.

When you force an execution plan, you are instructing the executor how you want the plan to execute. You use the CONTROL QUERY SHAPE statement to force a particular plan shape on a query. This statement is presented in SQL syntax as an SQL/MX extension to the ANSI standard. You need to understand the internal query tree structure to force a plan. You can place any allowed syntax within the CONTROL QUERY SHAPE statement, but the syntax does not ensure that a valid plan will be generated.

CONTROL QUERY SHAPE is very dependent on the internal design of the optimizer. Future improvements to the optimizer might make it necessary for users to change their CONTROL QUERY SHAPE statements.

For more information about the CONTROL QUERY SHAPE statement, see the *SQL/MX Reference Manual*.

Why Force a Plan?

Some of the possible reasons for forcing a plan include:

- Testing purposes. You might want to try different execution scenarios than those provided by the optimizer.
- The optimizer might not have found the optimal plan. This situation could occur because of lack of recent statistics and calibration, data skew, or aggressive pruning.

In these situations, forcing a plan gives you the power to control the plan shape. The optimizer chooses the optimal plan that matches the forced shape.

Checklist for Forcing Plans

Before you can force a plan, you need to know the contents of the plan:

1. Display the optimized plan for a prepared statement with the EXPLAIN function.
2. Review the optimized plan and costs associated with the operations.
3. Translate the operator tree into a text format by using the SHOWSHAPE utility or the format rules.
4. Use the CONTROL QUERY SHAPE statement to reshape the operator tree based on the text format that you specify.

Each of these steps is described in greater detail in the next subsections.

Displaying the Optimized Plan

Follow the steps in Section 4, Reviewing Query Execution Plans to display the query execution plan and to understand the operator tree provided by the optimizer. Note that you can view the optimized plan in several formats:

- Use the EXPLAIN function to display portions of the optimized plan.
- Use the EXPLAIN statement to view the entire optimized plan.
- Use the Visual Query Planner (VQP) to graphically display the entire optimized plan.

Note. To avoid unexpected results, turn off query caching at the beginning of your VQP session.

This sample query is used throughout this section to show how you can force an execution plan:

```
SELECT EMPLOYEE.LAST_NAME, EMPLOYEE.FIRST_NAME, DEPT.MANAGER,
EMPLOYEE.DEPTNUM, JOB.JOBCODE
  FROM DEPT, EMPLOYEE, JOB
  WHERE DEPT.DEPTNUM=3100 AND EMPLOYEE.DEPTNUM=3100 AND
     JOB.JOBCODE=300;
```

Reviewing the Optimized Plan

The next example shows the EXPLAIN output for the optimized sample query. While this output simply shows the operators and sequence numbers, you will also want to select the costing columns to review the estimated costs of each operation.

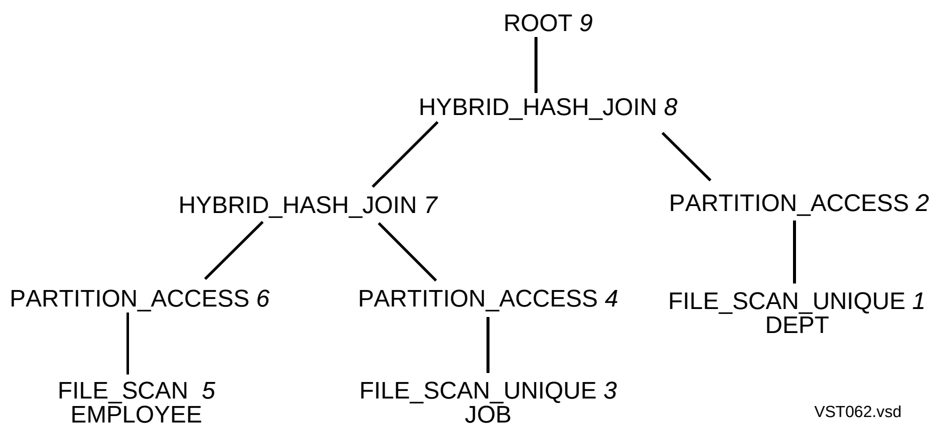
```
>>SET SCHEMA samdbcat.persnl;
>>PREPARE s1 FROM SELECT employee.last_name, employee.first_name,
>+dept.manager, employee.deptnum, job.jobcode
>+FROM dept, employee, job
>+WHERE dept.deptnum=3100 AND employee.deptnum=3100
>+AND job.jobcode=300;

--- SQL command prepared.

>>SELECT seq_num, operator, left_child_seq_num, right_child_seq_num
>+FROM table (EXPLAIN(NULL, 'S1'));

SEQ_NUM  OPERATOR             LEFT_CHILD_SEQ_NUM   RIGHT_CHILD_SEQ_NUM
-------- ------------------   ------------------   -------------------
5        FILE_SCAN                             ?                     ?
6        PARTITION_ACCESS                      5                     ?
3        FILE_SCAN_UNIQUE                      ?                     ?
4        PARTITION_ACCESS                      3                     ?
7        HYBRID_HASH_JOIN                      6                     4
1        FILE_SCAN_UNIQUE                      ?                     ?
2        PARTITION_ACCESS                      1                     ?
8        HYBRID_HASH_JOIN                      7                     2
9        ROOT                                  8                     ?

--- 9 row(s) selected.
```

The output shows that the query plan consists of three scan operators in DAM with exchange (PARTITION_ACCESS) operators that are responsible for the communication between the application process and DAM, two hybrid hash join operators, and a root node.

When you review the output, look at the sequence numbers for the node and the left and right child sequence numbers. Start reading from the root node. The output shows that the ROOT node has one child, a HYBRID_HASH_JOIN. This node has two children, another HYBRID_HASH_JOIN and a PARTITION_ACCESS with a file scan operation. The second HYBRID_HASH_JOIN also has two children nodes, each a PARTITION_ACCESS node with file scan operations. For more information about the operators, see Section 7, SQL/MX Operators.

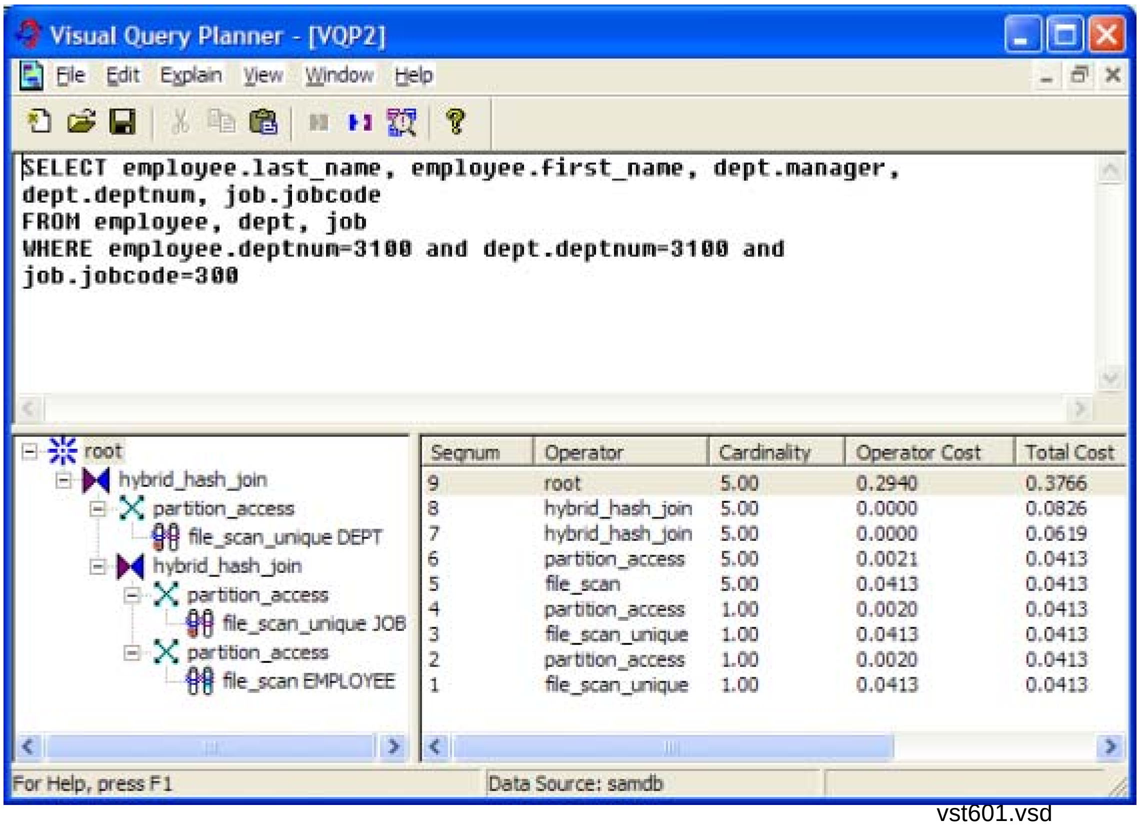
Now, view the output in a more visual tree format that shows the parent and child relationships. The sequence numbers and table names are also shown in Figure 5-1.

Figure 5-1. Query Plan Output in Visual Tree Format



You need to understand the relationships and which operations occur in the left and right hand sides of the operator tree before you can translate the operator tree into a text format that can be used to force the execution plan.

An easier way to see the operator tree for a query is to use the Visual Query Planner application. The Visual Query Planner provides the query execution plan in a graphical form so that you can easily see the operator tree.


Visual Query Planner - [VQP2]
File Edit Explain View Window Help

```
SELECT employee.last_name, employee.first_name, dept.manager,
dept.deptnum, job.jobcode
FROM employee, dept, job
WHERE employee.deptnum=3100 and dept.deptnum=3100 and
job.jobcode=300
```

```
root
  hybrid_hash_join
    partition_access
      file_scan_unique DEPT
    hybrid_hash_join
      partition_access
        file_scan_unique JOB
      partition_access
        file_scan EMPLOYEE
```

Segnum	Operator	Cardinality	Operator Cost	Total Cost
9	root	5.00	0.2940	0.3766
8	hybrid_hash_join	5.00	0.0000	0.0826
7	hybrid_hash_join	5.00	0.0000	0.0619
6	partition_access	5.00	0.0021	0.0413
5	file_scan	5.00	0.0413	0.0413
4	partition_access	1.00	0.0020	0.0413
3	file_scan_unique	1.00	0.0413	0.0413
2	partition_access	1.00	0.0020	0.0413
1	file_scan_unique	1.00	0.0413	0.0413

For Help, press F1 Data Source: samdb

vst601.vsd

Translating the Operator Tree to Text Format

You must translate the operator tree into a text format. The text format used to represent the tree to the CONTROL QUERY SHAPE statement is written in a LISP-like format. (LISP stands for list processor, a high-level programming language.)

Using SHOWSHAPE and SET SHOWSHAPE to View the Text Format

If you are using MXCI to execute your queries, you can use the SHOWSHAPE and SET SHOWSHAPE commands to view the text format for the CONTROL QUERY SHAPE statement. The SHOWSHAPE command simply shows the text format for the shape of the statement you execute; it does not show the Explain output.

```
>>SHOWSHAPE SELECT last_name, first_name, manager,
+>employee.deptnum, job.jobcode FROM dept, employee, job
+>WHERE dept.deptnum=3100 AND employee.deptnum=3100 AND
+>job.jobcode=300;
```

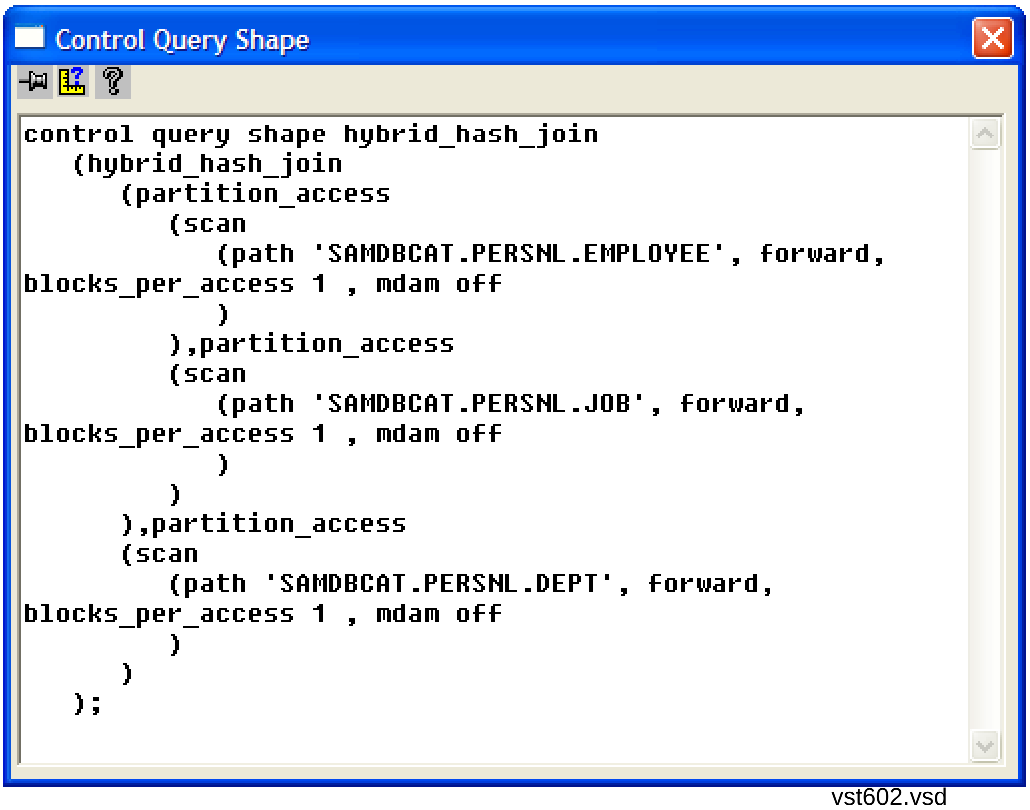
```
control query shape
hybrid_hash_join(hybrid_hash_join(partition_access(
scan(path 'SAMDBCAT.PERSNL.EMPLOYEE',
forward, blocks_per_access 1, mdam off)),partition_access(
scan(path 'SAMDBCAT.PERSNL.JOB', forward, blocks_per_access 1 ,
mdam off))),partition_access(scan(path 'SAMDBCAT.PERSNL.DEPT',
forward, blocks_per_access 1 , mdam off)));

--- SQL operation complete.
```

Use the additional command, SET SHOWSHAPE, to display the execution plans in effect. The text format for the shape is displayed immediately before the query output. If you use the SET SHOWSHAPE command prior to executing the sample query, your output appears as shown:

```
control query shape
hybrid_hash_join(hybrid_hash_join(partition_access(
scan(path 'SAMDBCAT.PERSNL.EMPLOYEE',
forward, blocks_per_access 1, mdam off)),partition_access(
scan(path 'SAMDBCAT.PERSNL.JOB', forward, blocks_per_access 1 ,
mdam off))),partition_access(scan(path 'SAMDBCAT.PERSNL.DEPT',
forward, blocks_per_access 1 , mdam off)));

Last Name             First Name       Mgr    Dept/Num  Job/Code
--------------------  ---------------  -----  --------  --------

WINTER                PAUL                43      3100       300
Farley                Walt                43      3100       300
Buskett               Emmy                43      3100       300
Buskett               Paul                43      3100       300
STRICKER              GEORGE              43      3100       300
WELLINGTON            PETE                43      3100       300
TAYLOR                DONALD              43      3100       300

--- 7 row(s) selected.
```

Using Visual Query Planner to Get the Shape

When you execute a plan with the Visual Query Planner, the text format for shaping the query can be found by choosing the **Show Query Shape** option from the **Explain** menu. The next figure shows the shape from the sample query.



```
control query shape hybrid_hash_join
   (hybrid_hash_join
      (partition_access
         (scan
            (path 'SAMDBCAT.PERSNL.EMPLOYEE', forward,
blocks_per_access 1 , mdam off
            )
         ),partition_access
         (scan
            (path 'SAMDBCAT.PERSNL.JOB', forward,
blocks_per_access 1 , mdam off
            )
         )
      ),partition_access
      (scan
         (path 'SAMDBCAT.PERSNL.DEPT', forward,
blocks_per_access 1 , mdam off
         )
      )
   );
```

vst602.vsd

You can make changes to the shape and then force a new shape by selecting the **Get Shape** icon (the middle icon in the upper left corner). In addition, from the **Explain** menu, you must execute **Get Explain Plan** to see the revised plan.

For more information about the SHOWSHAPE and SET SHOWSHAPE commands, see the *SQL/MX Reference Manual*. For more information about using the Visual Query Planner, see Section 4, Reviewing Query Execution Plans. The Visual Query Planner online help system also provides useful information.

Manually Writing the Shape

You translate the tree by recursively writing the tree with this rule (starting from the root):

```
TEXT (node) = node-identifier + '(' + TEXT(child1) + ',' +
                 TEXT(child2) + ... + ')'
```

node	Operator tree node being transformed
node-identifier	Identifier of the node.
child i	The *ith* child, if any, from left to right

Using the sample query, you can translate the operator tree into this text format:

```
ROOT(HYBRID_HASH_JOIN (HYBRID_HASH_JOIN
   (PARTITION_ACCESS (FILE_SCAN_UNIQUE),
    PARTITION_ACCESS(FILE_SCAN)),
    PARTITION_ACCESS (FILE_SCAN_UNIQUE)))
```

To refine a CONTROL QUERY SHAPE statement, several conditions apply to the text format:

- Leave off the ROOT node.
- Leave off EXPR nodes.
- Write FILE_SCAN and FILE_SCAN_UNIQUE nodes as SCAN.
- Replace SPLIT_TOP(PARTITION_ACCESS(...)) with SPLIT_TOP_PA(...).
- If no syntax exists for an operation in an operator tree (for example, SORT_SCALAR_AGGR or SHORTCUT_SCALAR_AGGR), use something similar, such as SORT_GROUPBY or SHORTCUT_GROUPBY. For the valid operators, check Section 7, SQL/MX Operators.

The refined text format looks like this:

```
HYBRID_HASH_JOIN (HYBRID_HASH_JOIN
    (PARTITION_ACCESS (SCAN),
     PARTITION_ACCESS (SCAN)),
     PARTITION_ACCESS (SCAN))
```

Writing the Forced Shape Statement

You use the CONTROL QUERY SHAPE statement to write the forced shape statement, using the text format that you previously formulated, and to replace the operators that you want to force. The CONTROL QUERY SHAPE statement is described in the *SQL/MX Reference Manual*. To understand why a plan uses certain operators, see Section 7, SQL/MX Operators.

In many cases, you arrive at the decision to force a plan through experimentation. Guidelines that can help you formulate plans follow.

Scope of CONTROL QUERY SHAPE

The result of the execution of a CONTROL QUERY SHAPE statement stays in effect until the end of the current MXCI session or until changed or turned off by another CONTROL QUERY SHAPE statement. Executing the CONTROL QUERY SHAPE statement does not affect the execution of CONTROL statements, the EXPLAIN function and EXPLAIN statement, LOCK and UNLOCK statements, DDL, and transaction statements.

Caution. Always turn off CONTROL QUERY SHAPE after you force a shape for a particular query. Otherwise, when you try to compile another query, the compiler fails to find a plan that matches the persisting forced shape.

Use the CUT, ANYTHING, or OFF option to turn off the shape:

```
CONTROL QUERY SHAPE OFF;
```

Shaping Portions of an Operator Tree

You can use the ANYTHING option to partially shape an operator tree. Use this option when you want a certain operation only and do not care how the rest of the plan is optimized. If you specify a partial tree, ANYTHING marks the point where you want the optimizer to “take over” and choose the best solution. This example shows the partial shape of an operator tree:

```
CONTROL QUERY SHAPE join (anything, union (anything, scan));
```

Using Logical and Physical Specifications

You can use logical or physical operators with the CONTROL QUERY SHAPE statement. Logical operators are relational operators that do not denote an implementation. Examples include join, group by, and scan. Physical operators are relational operators that specify the actual implementation or run-time algorithm, such as merge join, hash group by, or file scan.

Logical Operators	**Physical Operators**
scan	FILE_SCAN, INDEX_SCAN
hash_join	HYBRID_HASH_JOIN, ORDERED_HASH_JOIN
join	NESTED_JOIN, MERGE_JOIN, HYBRID_HASH_JOIN, ORDERED_HASH_JOIN
groupby	SORT_GROUPBY, HASH_GROUPBY, SHORTCUT_GROUPBY
union	UNION
sort	SORT
exchange	PARTITION_ACCESS (file system interface for communicating with DAM) REPARTITION (redistributes data) SPLIT_TOP_PA (reads data from multiple partitions in parallel, parallel version of PARTITION_ACCESS node)
expr	EXPR (internally generated node, not necessary to specify)
tuple	TUPLE

If you want to specify that an operation occurs, but you do not really care which algorithm the optimizer chooses to implement, use the logical specification. For example, you might want to specify that table EMPLOYEE is scanned as the last table of a join, but you want the optimizer to choose the join algorithm. In that case, use the logical JOIN specification, with SCAN(EMPLOYEE) as the second argument of the join. The optimizer is free to choose nested, merge, or hash join as the implementation.

When you force a plan by using the physical operator SHORTCUT_GROUPBY, the SHORTCUT_SCALAR_AGGR operator appears in the EXPLAIN output. If the optimizer cannot produce a plan with SHORTCUT_SCALAR_AGGR, no plan is returned.

Forcing Shapes on Views

When you prepare a query and use the EXPLAIN output to look at compiled statistics, the output shows that the compiler expands the view and that the operation occurs on underlying base tables. To affect a new shape on a SQL view, reference the underlying base tables in your CONTROL QUERY SHAPE statement.

What Happens if No Plan Is Returned?

If you try to shape a plan and the optimizer fails to return a plan, consider these reasons:

- The forced shape might be incompatible with the issued query. That is, the shape has no match in the optimizer search space defined by the optimizer rules. For example, a plan is not returned if you attempt to force a table scan shape on a two-table join query.
- The forced shape might be compatible with the query, but matching plans are pruned by optimizer heuristics. In this case, you might try changing the default value for the DATA_FLOW_OPTIMIZATION and the CROSS_PRODUCT_CONTROL attributes to OFF and try the plan again to see if the results are different.

Migrating Forced Shapes From NonStop SQL/MP

If you forced plans in NonStop SQL/MP by using the CONTROL TABLE statement, you will need to rewrite your plans by using CONTROL QUERY SHAPE, which provides more control. Note that, while CONTROL TABLE enables you to force a single operation, CONTROL QUERY SHAPE requires that you force the entire tree structure.

Forcing Group By Operations to the Data Access Manager

The SQL/MX compiler has two methods for performing grouping and aggregation.

Note. GROUP BY operators are sometimes used even when no GROUP BY clause is specified. The SQL/MX compiler inserts a GROUP BY operator into a SELECT DISTINCT query to remove duplicate rows.

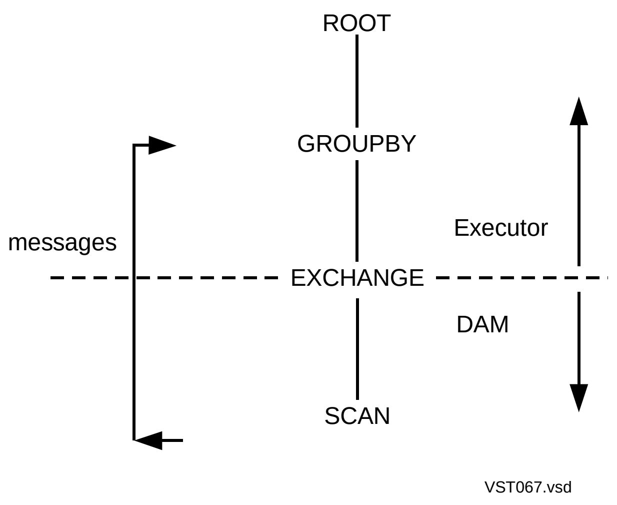
- Sort group by requires the child of the group by to be ordered on the group-by column and thus preserves ordering. This method incurs additional sort operations unless input is already sorted on the grouping columns.

- Hash group by uses hashing operations to perform grouping and has no ordering requirement on children. In general, hash group by is more cost effective than sort group by.

The compiler uses algorithms based on cost to determine which group by to perform, hash or sort. The aggregate functions are AVG, COUNT, MAX, MIN, STDDEV, SUM, and VARIANCE. For further information about aggregate functions, see the individual entries in the *SQL/MX Reference Manual*.

Single table operations can always be pushed down to the DAM level, as shown in Figure 5-2.

Figure 5-2. Group By Operator Not Using DAM



Notice that messages are exchanged from the SCAN operator in DAM to the GROUP BY operator in the executor through the EXCHANGE node. For tables with millions of rows, this amount of messaging can be inefficient. The group by operation can be pushed down to DAM to reduce the message traffic between DAM and the executor, as demonstrated in the next example.

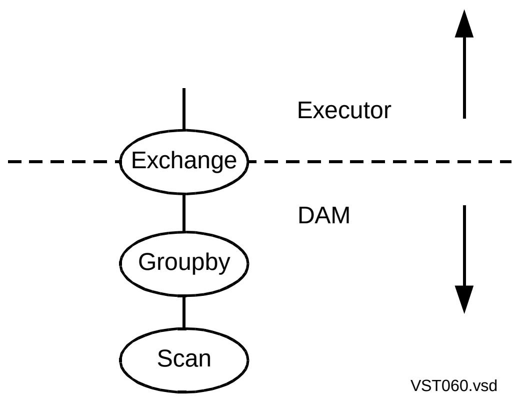
Consider this query against an EMPLOYEE table that contains 50,000 rows (employees) and uses 18 partitions:

```
SELECT COUNT(*) FROM EMPLOYEE;
```

When the compiler presents a query plan like the plan shown in Figure 5-2, the SCAN operator passes messages in blocks to the EXCHANGE node. The EXCHANGE node passes the messages to the GROUP BY operator.

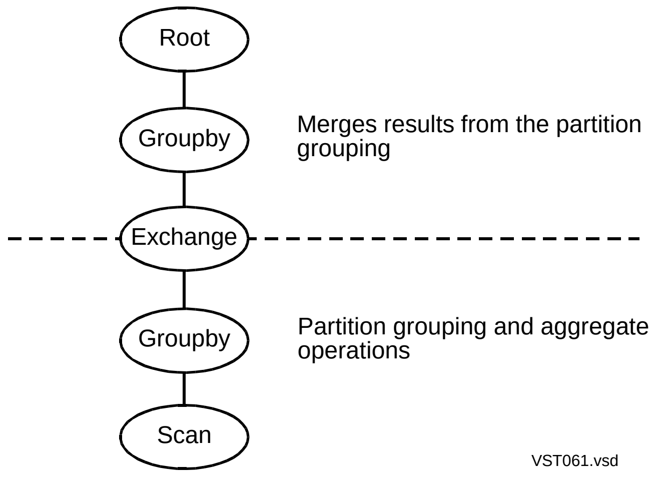
To reduce this message traffic, move the GROUP BY operator down to DAM, as shown in Figure 5-3.

Figure 5-3. GROUP BY Operator at the DAM Level



The message traffic is reduced as follows. The SCAN operation still scans all 50,000 rows. However, the group by operation yields one result for each table partition (18). This result is passed on to the EXCHANGE node. You should include another GROUP BY operator above the EXCHANGE node to combine the results of the lower group by operation, as shown in Figure 5-4.

Figure 5-4. Two GROUP BY Operations


Root
Groupby
Merges results from the partition grouping
Exchange
Groupby
Partition grouping and aggregate operations
Scan
VST061.vsd

General Case for Group By Operations

Pushing down group by operations for partitioned tables (more than one partition) and for any hash group by operations requires an additional group by operation. The lower group by performs the grouping and aggregation per partition, and the upper group by performs the final aggregation across all groups. For hash group by situations, the upper group by is required to handle any overflow that occurs in DAM.

Special Case for Sorted Group By Operations

For single partition sorted group by operations, only one GROUP BY operator is required. Sort group by operations can be performed only if the input to the group by is already ordered on the group by columns. In this case, the compiler can perform an index scan or file scan for the group, and the additional cost of sorting is avoided. The input into the sort group by must already be ordered, because DAM cannot perform the sort.

Forcing Parallel Plans

The optimizer tries to present you with the most cost-efficient plan available. You might find, however, that the optimizer does not always present the parallel plan that you want. If you choose to use CONTROL QUERY SHAPE to force a parallel plan, note that overriding the optimizer's standard cost estimates can cause negative performance.

Forcing ESP Parallelism

To get ESP parallelism, you can put an ESP_EXCHANGE operator in front of the operator you want to run in parallel.

You might also get ESP parallelism by entering this command:

```
CONTROL QUERY SHAPE ESP_EXCHANGE(CUT);
```

Note. To use ESP parallelism, you must have ATTEMPT_ESP_PARALLELISM set to ON or SYSTEM. In addition, you must have more than one CPU in your system.

You can force the number of ESPs used. For joins, specify the number of ESPs when you specify the join. For nonjoin operations, specify the EXCHANGE logical operator with the number of ESPs. Do not use the CUT keyword for the child. Otherwise, you might not get the intended result.

Caution. When you force the number of ESPs, you completely override the internal settings of the optimizer. Use this option carefully and only if the optimizer does not choose the number of ESPs that you think you need.

Forcing DAM Parallelism

To get DAM parallelism, force SPLIT_TOP_PA above the item you want to run in DAM:

```
CONTROL QUERY SHAPE SPLIT_TOP_PA (ANYTHING);
```

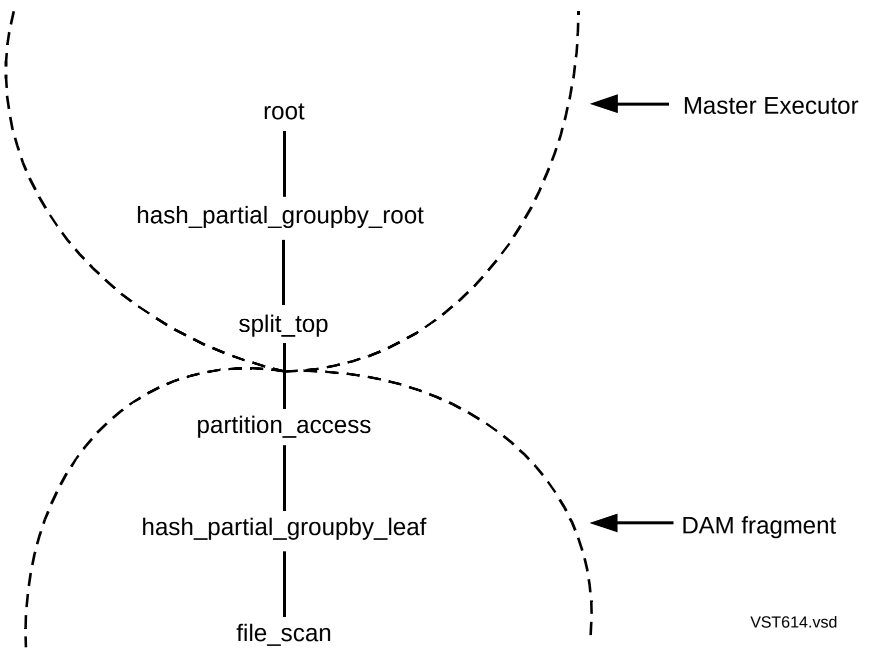
Use this query:

```
SELECT JOBCODE, AVG(SALARY) FROM EMPLOYEE
    WHERE JOBCODE > 55 AND SALARY <= 3000
        GROUP BY JOBCODE
```

This shape forces a partial grouping in DAM with a consolidator grouping in the ESP or master executor, as shown in Figure 5-5.

```
CONTROL QUERY SHAPE
    GROUPBY(
         SPLIT_TOP_PA(
            GROUPBY(SCAN))
                );
```

Figure 5-5. Query Tree for the Forced Plan



To force an operator to execute in the DAM without DAM parallelism, force a PARTITION_ACCESS operator on top of the operator you want to run in the DAM. However, the operator must be capable of the request. For example, a join cannot be used for a direct request to the DAM. The operators that can make a direct request to the DAM include scans and group bys.

Forcing Joins

The syntax for CONTROL QUERY SHAPE provides the ability to force certain types of joins for parallel plans, as described next. For more information on Type1 (matching partition algorithm) and Type2 (parallel access to the inner table) joins, see Section 8, Parallelism.

- Forcing a Type1 join

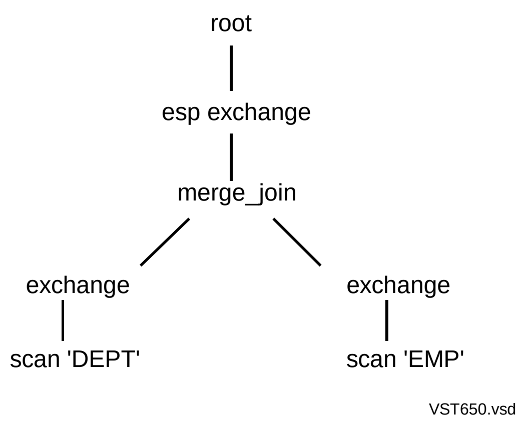
  The matching partitions algorithm can be forced by specifying Type1 as part of the join specification. To force a Type1 join without repartitioning, all criteria must match for both tables. To force a Type1 join when the table criteria does not match, use a CUT for the children, or use an ESP_EXCHANGE above the children for any necessary repartitioning.

This example shows forcing a Type1 join:

```
CONTROL QUERY SHAPE
  ESP_EXCHANGE(
   MERGE_JOIN(
     EXCHANGE(SCAN('DEPT')),
     EXCHANGE(SCAN('EMP')),
     TYPE1)
      );
```

The example uses the logical specification for the lower exchange operators to enable the optimizer to choose parallel access (SPLIT_TOP_PA operator) or serial access (PARTITION_ACCESS operator), as shown in Figure 5-6.

Figure 5-6. Logical Specification and Lower Exchange Operators



For more information about Type1 joins, see Parallelism on page 8-1.

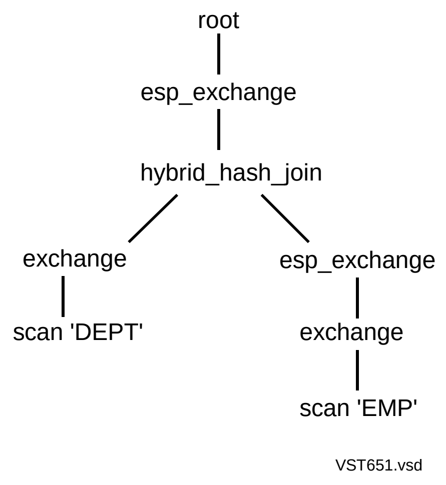
- Forcing a Type2 join

  The join with parallel access to the inner table algorithm can be forced by using Type2 as a part of the join specification. To force a Type2 nested join, the left child must be partitioned. To force a Type2 hash join, the left child must be partitioned, and you need to specify either CUT or ESP_EXCHANGE above the right child to handle the broadcast replication.

  The next statement forces a Type2 hash join, as shown in Figure 5-7 on page 5-16.

```
CONTROL QUERY SHAPE
  ESP_EXCHANGE(
    HYBRID_HASH_JOIN (
      EXCHANGE(SCAN('DEPT')),
      ESP_EXCHANGE(EXCHANGE(SCAN('EMP'))),
      TYPE2)
        );
```

Figure 5-7. Type2 Hash Join



For more information about Type2 joins, see Parallelism on page 8-1.

- Deferring to the optimizer to choose exchange or sort operators

  In the preceding forced join examples, you needed to specify the choice of exchange and sort operators as part of the CONTROL QUERY SHAPE statement. To eliminate this step and simplify the process of writing a valid CONTROL QUERY SHAPE statement, you can use one of the options listed below with the CONTROL QUERY SHAPE statement. Using these options enables you to focus on join orders, join types, and other plan types while deferring the choice of exchange or sort operators to the optimizer:

  - IMPLICIT EXCHANGE enables the optimizer to add exchange nodes at any location necessary to make the CONTROL QUERY STATEMENT valid. The optimizer chooses the optimal placement of the exchange nodes. You can still explicitly add in exchange nodes and the compiler must add them.
  - IMPLICIT SORT enables the optimizer to add sort nodes at any location necessary to make the CONTROL QUERY STATEMENT valid. The optimizer chooses the optimal placement of the sort nodes.You can still explicitly add in sort nodes and the compiler must add them.
  - IMPLICIT EXCHANGE_AND_SORT enables the optimizer to add both exchange and sort nodes at any location necessary to make the CONTROL QUERY STATEMENT valid. The optimizer chooses the optimal placement of the exchange and sort nodes. You can still explicitly add in enforcers and the compiler must add them.

This statement enables the optimizer to add exchange nodes:

```
CONTROL QUERY SHAPE IMPLICIT EXCHANGE
      HYBRID_HASH_JOIN (
      SCAN('DEPT'),
      SCAN('EMP'),
      TYPE2);
```

For syntax and more information, see the *SQL/MX Reference Manual*.

6 Query Plan Caching

Use the information in this section to understand query plan caching:

Overview

Query Plan Caching is a feature of the SQL/MX compiler that provides the ability to cache the plans of certain queries. This feature improves performance when the plan can be produced from the cache rather than through a full compilation. The performance improvement is typically 60 to 80 percent (compile time) for simple TP-style queries.

Certain default settings used with the CONTROL QUERY DEFAULT statement apply to query plan caching. For more information, see SYSTEM_DEFAULTS Table Settings for Query Plan Caching Attributes on page 6-7.

The query plan caching feature has been designed to operate transparently. No SQL/MX application source code changes are required. When given a query, NonStop SQL/MX produces the same plan with or without query caching. This correctness requirement implies that the SQL/MX compiler honors CONTROL QUERY DEFAULT and CONTROL TABLE statements even when query caching is active. To illustrate this behavior, suppose that:

- The table timeout setting for table T is set to infinite (-1).
- The SQL/MX compiler is asked to compile "`SELECT * FROM T`"
- The SQL/MX compiler compiles and caches a plan for "`SELECT * FROM T`"
- A CONTROL TABLE statement changes the timeout setting for table T to five seconds
- The SQL/MX compiler is asked to compile "`SELECT * FROM T`" again

The SQL/MX compiler cannot use the previously cached plan for "`SELECT * FROM T`" because the plan has an infinite timeout setting for Table T. If the compiler were to use the cached plan, the compiler would effectively ignore the CONTROL TABLE statement that changed the timeout for Table T to five seconds. Therefore, the compiler can compile and cache one or more plans for one query with each plan associated with a different set of control settings.

A query that is compiled repeatedly, each time with a new set of control settings, does not result in a cache hit. A query that is compiled subsequently, with a set of control settings that was in effect when this query was previously compiled, results in a cache hit, assuming that all other criteria for a cache hit are met.

For example, consider how the compiler responds to a sequence of control statements and query compilation requests:

```
CONTROL TABLE T TIMEOUT 500;
SELECT * FROM T; -- mxcmp adds "SELECT * FROM T; T TIMEOUT 500" to
cache

CONTROL TABLE T TIMEOUT -1;
SELECT * FROM T; -- mxcmp adds "SELECT * FROM T; T TIMEOUT
-1" to cache

CONTROL TABLE T TIMEOUT 500;
SELECT * FROM T; -- mxcmp hits on "SELECT * FROM T; T TIMEOUT 500"

CONTROL TABLE T TIMEOUT -1;
SELECT * FROM T; -- mxcmp hits on "SELECT * FROM T; T TIMEOUT -1"
```

The SQL/MX compiler responds similarly to CONTROL QUERY DEFAULT statements.

Types of Cacheable Queries

The queries that are considered for query plan caching include simple TP-style inserts, updates, deletes, selects, and joins. Two queries are considered equivalent for the purposes of caching if their canonical forms are the same. For query caching, the canonical form of a query is constructed by:

- Removing unmeaningful white space differences
- Removing unmeaningful case differences
- Expanding ‘*’ notation in select lists
- Resolving all object names to fully qualified names
- Replacing most constant literals with parameters
- Encoding all CONTROL QUERY DEFAULT and CONTROL TABLE statements that have been previously executed in the current SQL/MX compiler session

Query caching is restricted to only those queries whose compiled plans and plan quality are unaffected by the actual values of their literal constants and that have a high probability for reuse.

UPDATE STATISTICS does not affect query caching. Cacheable queries remain cacheable, and noncacheable queries remain noncacheable with or without updating statistics.

The SQL/MX compiler generates the same plan for many TP-style queries that are guaranteed to return or update at most one row. The next examples are all guaranteed to update or return at most one row if table T has K as its primary key column. These are all cacheable queries:

```
DELETE FROM T WHERE K=1;
```

```
UPDATE T SET C=1 WHERE K=2;
```

```
SELECT * FROM T WHERE K=1;
```

```
INSERT INTO T(K,C) VALUES(2,1);
```

Examples of Cacheable Expressions

- Dynamic parameters are cacheable

  ```
  UPDATE T SET C=? WHERE K=?;
  ```

- Arithmetic expressions are cacheable

  ```
  SELECT m + n - p * q / r FROM T WHERE K=?;
  ```

- Aggregate functions (MAX, MIN, SUM, AVG, COUNT are cacheable):

  ```
  SELECT MAX(i), MIN(i), SUM(i), AVG(i), COUNT(DISTINCT i)
  FROM T WHERE K=?;
  ```

- Concatenation is cacheable

  ```
  UPDATE T SET D=D||'1', E=CONCAT(E,'z') WHERE K=?;
  ```

- String functions (CHAR LENGTH, OCTET LENGTH, LCASE, LOWER, UCASE, UPPER, UPSHIFT are cacheable):

  ```
  SELECT CHAR LENGTH(d), OCTET LENGTH(d), LCASE(d), LOWER('A'),
  UCASE(d), UPPER('a'), UPSHIFT('b') FROM T;
  ```

- Datetime functions (CONVERTTIMESTAMP, JULIANTIMESTAMP, CURRENT TIMESTAMP, CURRENT DATE, CURRENT TIME, CURRENT, NOW, DATEFORMAT, DAY, DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, FIRSTDAYOFYEAR, HOUR, MINUTE, MONTH, MONTHNAME, QUARTER, SECOND, WEEK, YEAR FUNCTIONS are cacheable):

  ```
  UPDATE T SET
  TS=CONVERTTIMESTAMP(JULIANTIMESTAMP(CURRENT_TIMESTAMP))
  WHERE K=?;
  ```

  ```
  SELECT CURRENT_DATE, CURRENT_TIME, DATEFORMAT(NOW(),USA),
  DATEFORMAT(NOW(),EUROPEAN), DAY(CURRENT), DAYNAME(NOW(),
  DAYOFMONTH(NOW()), DAYOFWEEK(NOW()), DAYOFYEAR(NOW()),
  FIRSTDAYOFYEAR(NOW()), HOUR(NOW()), MINUTE(NOW)),
  MONTH(NOW()), MONTHNAME(NOW()), QUARTER(NOW()),
  SECOND(NOW()), WEEK(NOW()), YEAR(NOW()) FROM (VALUES(1)) AS
  T;
  ```

- Case expressions are cacheable:

  ```
  SELECT CASE i WHEN 3 THEN 'YES' ELSE 'NO' END FROM T;
  ```

- Math functions (ABS, ATAN, ATAN2, CEILING, COS, COSH, DEGREES, EXP, FLOOR, LOG, LOG10, PI, POWER, RADIANS, SIGN, SIN, SINH, SQRT, TAN, TANH are cacheable):

  ```
  SELECT ABS(i), ATAN(10), ATAN2(x,y), CEILING(r), COS(i),
  COSH(i), DEGREES(i), EXP(i), FLOOR(i), LOG(i), LOG10(i),
  PI(), POWER(b,e), RADIANS(i), SIGN(i), SIN(i), SINH(i),
  SQRT(i), TAN(i), TANH(i) FROM T;
  ```

- Replace functions are cacheable:

  ```
  UPDATE T SET job=replace(job, 'IM', 'IT') WHERE K=?;
  ```

- Two-table single row joins are cacheable. This two-table single row join is cacheable, assuming table T has K as its primary (or partition) key and table U has J as its primary (or partition) key:

  ```
  SELECT * FROM T, U WHERE (K,J)=(?,?) AND K=J;
  ```

All other functions not listed here are not cacheable.

The SQL/MX compiler is guaranteed to generate the same plans for similar TP-style queries that differ only in their literal values. These examples yield the same plan:

```
DELETE FROM T WHERE K=9999;
DELETE FROM T WHERE K=7;
DELETE FROM T WHERE K=?;
```

The next three examples also yield the same plan:

```
UPDATE T SET C=35 WHERE K=14;
UPDATE T SET C=7 WHERE K=31;
UPDATE T SET C=? WHERE K=?;
```

Query plan caching effectively treats most literal values in TP-style queries as wild cards when comparing a query against previously cached queries. However, not all literal values are treated as wild cards. For example, the pattern and escape literals in this example are not treated as wild cards during query comparison:

```
SELECT * FROM T WHERE K=1 AND S LIKE '\_C%' escape '\';
```

The previous query is not considered the same as the next query during query comparison:

```
SELECT * FROM T WHERE K=1 AND S LIKE '\$C%' escape '\';
```

Examples of Queries That Are Not Cacheable

- Queries that have only LIKE predicates are not cacheable:

  ```
  SELECT * FROM T WHERE S LIKE 'c%'; --is not cacheable
  ```

However, a LIKE predicate conjunct of a key equipredicate is cacheable:

`SELECT * FROM T WHERE K=? AND S LIKE 'c%';`

- Queries that have only OR predicates are not cacheable:

  `SELECT * FROM T WHERE a=1 OR b=2;` -- is not cacheable

  However, an OR predicate conjunct of a key equipredicate is cacheable:

  `SELECT * FROM T WHERE K=? and (a=1 OR b=2);`

- Queries that have BETWEEN predicates are not cacheable:

  `SELECT * FROM T WHERE a BETWEEN 1 AND 9;` -- is not cacheable

  `SELECT * FROM T WHERE K=? AND (a BETWEEN 1 AND 9);` -- is not cacheable

- Queries that have only IN predicates are not cacheable:

  `SELECT * FROM T WHERE i IN (1,2);` -- is not cacheable

  However, an IN predicate conjunct of a key equi-predicate is cacheable:

  `SELECT * FROM T WHERE K=? AND i IN (1,2);` -- is cacheable

  Also, a single-value key IN predicate is cacheable:

  `SELECT * FROM T WHERE K IN (1);`

- Queries that have only NOT predicates are not cacheable:

  `SELECT * FROM T WHERE NOT(i IN (1,2));` -- is not cacheable

  `SELECT * FROM T WHERE NOT (K<>1);` -- is not cacheable

  However, a NOT predicate conjunct of a key equipredicate is cacheable:

  `SELECT * FROM T WHERE K=? AND NOT (i IN (1,2));`

- Queries that have function calls other than those listed in Examples of Cacheable Expressions on page 6-3 are not cacheable:

  `SELECT * FROM T WHERE K=? AND SUBSTRING(c,1,1)='z';` --is not cacheable

- Queries that have subqueries are not cacheable:

  `SELECT * FROM T WHERE K=? AND N=(SELECT MAX (b) FROM t);` --is not cacheable

- Queries that have relational unions, intersections, differences, divisions, and table-value stored procedures are not cacheable:

  `SELECT a FROM t UNION SELECT b FROM s;` -- is not cacheable

- Queries that have compound statements or rowsets are not cacheable.

- Queries that have transpose, sample, sequence, offset, or other data mining predicates are not cacheable.

- Data Definition Language (DDL) statements are not cacheable.
- Data Control Language (DCL) statements are not cacheable.

Choosing an Appropriate Size for the Query Cache

To adequately choose an appropriate size for the query cache, examine your applications.

Static applications that precompile their queries once during application development and rarely recompile their queries during application deployment and operation should turn off plan caching by specifying a QUERY_CACHE default setting of 0.

Dynamic applications, such as a book search engine that processes many queries whose text is not known beforehand, can specify a QUERY_CACHE size that can hold most of the frequently processed queries. For example, if an application processes 40 classes of queries and the average plan size of a query is 100 KB, a QUERY_CACHE size of 4000 KB might be optimal. The steps for finding the size of an entry are explained under QUERYCACHE Function on page 6-9.

An application can change the QUERY_CACHE size during operation with the CONTROL QUERY DEFAULT command. For example, a mixed mode application that does both transaction processing (TP) and decision support system (DSS) queries can increase the QUERY_CACHE size just before it switches to TP mode to hold and cache more TP queries. Likewise, the application might reduce the QUERY_CACHE size just before it switches to DSS mode.

Dynamic applications that spend a significant amount of time compiling and executing queries can hold and cache more queries for any given QUERY_CACHE size by turning off the GENERATE_EXPLAIN default setting. Turning off GENERATE_EXPLAIN reduces the average plan size by about 15 percent for TP-style queries. As a result, a query plan cache can hold about 15 percent more queries.

Query Plan Caching Statistics

NonStop SQL/MX provides a convenient way to determine important information about the caching process in addition to the current state of stored plans. This information is provided in two virtual tables. You query these tables at the MXCI prompt by using the SELECT statement as if they were physical tables.

If no query plans have been cached, no rows are returned.

For syntax information, see the QUERYCACHE and QUERYCACHEENTRIES functions in the *SQL/MX Reference Manual.*

SYSTEM_DEFAULTS Table Settings for Query Plan Caching Attributes

This subsection provides additional information about the query plan caching externalized attributes. The SYSTEM_DEFAULTS table entry of the *SQL/MX Reference Manual* provides reference information about these settings:

- QUERY_CACHE

  System-defined default setting: 1024 kilobytes (KB)
  Allowable values: 0 to 4194303

  The value of QUERY_CACHE indicates the KB size to which the cache is allowed to grow. The default setting, 1024, activates a query cache that can grow to 1024 KB in the current session.

  Although the maximum value for QUERY_CACHE is 4194303, you should not set the QUERY_CACHE limit to a value greater than or equal to a fraction of the physical memory of the host machine. Doing so is likely to result in reduced performance as the HP NonStop operating system repeatedly swaps the SQL/MX compiler (bloated by a huge cache) in and out of the host machine's physical memory. A good strategy might be to avoid setting QUERY_CACHE to more than 10 percent of the host machine's physical memory.

  If a new entry causes the size of the query cache to exceed the value of the QUERY_CACHE default, current entries are removed on a “least recently used” basis, taking into account pinned entries and the value of the default QUERY_CACHE_MAX_VICTIMS. See QUERY_CACHE_STATEMENT_PINNING on page 6-9.

  To deactivate the query cache in the current session, set QUERY_CACHE to 0. If a query cache is allocated, this setting frees it.

- QUERY_CACHE_MAX_VICTIMS

  System-defined default setting: 10 cache entries
  Allowable values: 0 to 4194303

  This attribute indicates the maximum number of cache entries that can be displaced to accommodate a new entry and stay within the size limit of the cache. When considering displacement of entries in the cache, the compiler looks for the least recently used unpinned entries of a combined size that is greater than the size of the new entry. If there are not enough least recently used unpinned entries, the compiler looks for any least recently used pinned entries to displace. Setting this attribute to a very large value means that all the cache entries could be displaced to accommodate one very large query.

  Setting this attribute to 0 means that, when the cache becomes full, no cache entries (pinned or unpinned) can be displaced, and no new entries can be entered into the cache. The first *n* queries occupy the cache (where *n* is the number of entries it takes to fill the cache). If the cache is full and a new query comes along, the new entry is not added to the cache, and no resident entries can be displaced. Because the query plan cache feature is transparent, no error messages are issued.

If QUERY_CACHE_MAX_VICTIMS is later set to a nonzero value, replacement resumes as usual. The number of entries that the cache can hold depends on the size of the cache and the size of the cached plans. The system-defined default setting limits the number of cache entries that can be displaced to 10 cache entries.

- QUERY_CACHE_REQUIRED_PREFIX_KEYS

  System-defined default setting: 255
  Allowable values: 0 to 255

  This attribute specifies how many and which columns of a composite primary or partition key are required for an equality predicate to be considered cacheable. If the attribute is set to a value greater than the number of columns in a composite key, all columns of the key are required. The system-defined default setting is 255, which means that only complete primary or partition key equality predicates are cacheable. To avoid compromising query plan quality, it is recommended that you keep the system-defined default setting of 255.

  The value 0 means that the presence of any one column of a composite primary or partition key in an equality key predicate is sufficient to make that predicate cacheable. A value *n* that is greater than zero but less than the number of columns in the key indicates that the first *n* columns of the key are required to be present in a key predicate for that predicate to be considered cacheable.

  Suppose that the QUERY_CACHE_REQUIRED_PREFIX_KEYS setting is 1, and the table T has a composite primary key consisting of columns (a, b, and c). With the setting of 1, provided that the first column of the key (a) is an equality predicate, the query is cacheable. If the QUERY_CACHE_REQUIRED_PREFIX_KEYS setting is 2, the only valid prefixes of (a, b, c) are (a, b) and (a, b, c). That is, these queries are cacheable:

  ```
  SELECT * FROM T WHERE a=1 AND b = 20;
  DELETE FROM T WHERE (a,b,c)=(9,909,10);
  ```

  However, these queries are not cacheable:

  ```
  SELECT * FROM T WHERE a=77;
  DELETE FROM T WHERE (b,c)=(1,23);
  ```

  This attribute can be used to force certain noncacheable queries into cacheable queries. In the previous example, `SELECT * FROM T WHERE A=77` is not cacheable because its equi-predicate specifies only the first of a three-column key. To make it cacheable, specify CONTROL QUERY DEFAULT QUERY_CACHE_REQUIRED_PREFIX_KEYS '1,' and the query becomes cacheable.

- QUERY_CACHE_STATEMENT_PINNING

  System-defined default setting: OFF
  Allowable values: ON, OFF, CLEAR

  This attribute controls whether queries are entered into the cache as pinned or unpinned. You might have important, compile-time critical queries that you want to ensure are in the cache when needed. When a query is pinned in the cache, it usually cannot be displaced from the cache unless the cache becomes full of pinned queries. In this case, the least recently used pinned entries also become displaceable.

  The system-defined default setting, OFF, means that all subsequent query cache entries are unpinned.

  The value CLEAR means that all subsequent query cache entries are unpinned, and all pinned entries in the cache are also unpinned.

  The value ON means that all subsequent query cache entries are pinned.

QUERYCACHE Function

The query plan cache automatically collects statistics regarding its use. When invoked, the QUERYCACHE table-valued stored function collects and returns the current state of these statistics in a single row table. The statistics are reinitialized when an `mxcmp` session is started, and each `mxcmp` session maintains an independent set of statistics.

This table describes the various statistics of the QUERYCACHE table:

Column Name	**Data Type**	**Description**
AVG_PLAN_SIZE	INT UNSIGNED	Total KB size of all cache entries divided by the number of entries.
CURRENT_SIZE	INT UNSIGNED	Current KB size of the query cache.
MAX_CACHE_SIZE	INT UNSIGNED	Maximum cache size in KB.
MAX_NUM_VICTIMS	INT UNSIGNED	Maximum number of plans that can be removed from the cache to make room for a new entry.
NUM_ENTRIES	INT UNSIGNED	Total number of query entries in the cache.
NUM_PINNED	INT UNSIGNED	Total number of pinned entries.
NUM_COMPILES	INT UNSIGNED	Total number of complete compile requests (excludes DESCRIBE and SHOWSHAPE).
NUM_RECOMPILES	INT UNSIGNED	Total number of recompilations. Recompilation of a cached plan occurs when a referenced table has been re-created or altered.

Column Name	Data Type	Description
NUM_RETRIES	INT UNSIGNED	Number of successful compiles that initially fail with caching on (caused by a defect in `mxcmp`) and that succeed with caching off.
NUM_CACHEABLE_PARSING	INT UNSIGNED	Total number of SQL statements that `mxcmp` has processed after parsing and before binding the query that satisfy the conditions for caching.
NUM_CACHEABLE_BINDING	INT UNSIGNED	Total number of SQL statements that `mxcmp` has processed after binding and before transformation of the query that satisfy the conditions for caching.
NUM_CACHE_HITS_PARSING	INT UNSIGNED	Total number of SQL statements that `mxcmp` has processed after parsing and before binding that have produced hits.
NUM_CACHE_HITS_BINDING	INT UNSIGNED	Total number of SQL statements that `mxcmp` has processed after binding and before transformation of the query that have produced hits.
NUM_PIN_HITS_PARSING	INT UNSIGNED	Total number of hits on pinned entries that occurred after parsing and before binding.
NUM_PIN_HITS_BINDING	INT UNSIGNED	Total number of hits on pinned entries that occurred after binding and before transformation.
NUM_CACHEABLE_TOO_LARGE	INT UNSIGNED	Number of SQL statements processed by `mxcmp` that satisfy the conditions for cacheability but with plans too large to fit in the cache.
NUM_DISPLACED	INT UNSIGNED	Number of entries removed from the cache to make room for new entries or as a consequence of a resizing of the cache or recompilation.
OPTIMIZATION_LEVEL	CHAR(10)	Indicates the current desired level of query optimization. Can be 0, 2, 3, or 5.
PINNING_STATE	CHAR(4)	Current state of pinning. Can be ON or OFF.

QUERYCACHEENTRIES Function

The query plan cache automatically collects statistics on each entry of the cache. When invoked, the QUERYCACHEENTRIES table-valued stored function collects and returns these statistics in a table with one row for each entry of the cache. The statistics are reinitialized when an `mxcmp` session is started. Each `mxcmp` session maintains an independent set of statistics.

This table describes the statistics of the QUERYCACHEENTRIES table:

Column Name	**Data Type**	**Description**
ROW_ID	INT UNSIGNED	A zero-based sequential number. Entry number 0 is the most recently used entry. When a new entry is cached or matches the query issued, it occupies zero, and all other cache entries not displaced are increased by one. Entry number 1 is the most recently used entry after the most recent. Entries with the highest row IDs are the ones replaced; they are the least recently used entries.
PLAN_ID	LARGEINT	Primary key. System-generated timestamp stored within each plan that uniquely identifies it. This column appears in the EXPLAIN table and enables joins between the two tables.
TEXT	CHAR(1024)	Text of the original SQL statement.
ENTRY_SIZE	INT UNSIGNED	Size in bytes of this entry.
NUM_HITS	INT UNSIGNED	Total number of hits for this entry.
PHASE	CHAR(10)	Contains the `mxcmp` phase after when the plan associated with this entry was cached (parsing or binding)
OPTIMIZATION_LEVEL	CHAR(10)	Indicates the desired level of code optimization at the time the query was compiled. Can be 0, 2, 3, or 5.
CATALOG_NAME	CHAR(40)	Name of default catalog under which the query was compiled.
SCHEMA_NAME	CHAR(40)	Name of default schema under which the query was compiled.
NUM_PARAMS	INT UNSIGNED	Number of constants in the query that were changed internally into parameters during compilation.

Column Name	Data Type	Description
PARAM_TYPES	CHAR(1024)	Comma-separated list of the types of constants that were changed into parameters. Blank if none.
PLAN_LENGTH	INT UNSIGNED	Size in bytes of the compiled plan associated with this query.
IS_PINNED	CHAR(6)	Indicates whether the entry is pinned. Can be ON or OFF.
COMPILATION_TIME	INT UNSIGNED	Time in milliseconds it took to compile the query associated with this entry.
AVERAGE_HIT_TIME	INT UNSIGNED	Time in milliseconds it took on average to process a query as a cache hit against this entry.
SHAPE	CHAR (1024)	Required CONTROL QUERY SHAPE of the query associated with this entry. Blank if no required shape.
ISOLATION_LEVEL	CHAR(20)	Transaction isolation level associated with the query. Can be READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE, NOT_SPECIFIED.
ACCESS_MODE	CHAR(20)	Transaction access mode value associated with the query. Can be READ_ONLY, READ_WRITE, or NOT_SPECIFIED.
AUTO_COMMIT	CHAR(15)	Transaction auto-commit value associated with the query. Can be ON, OFF, or NOT_SPECIFIED.
ROLLBACK_MODE	CHAR(15)	Transaction rollback mode value associated with the query. Can be WAITED, NOWAITED, or NOT_SPECIFIED.

Querying the Query Plan Caching Virtual Tables

You can query and display certain columns or all columns of the query plan caching virtual tables. You specify the virtual table in a SELECT statement preceded by the keyword *table* and surrounded by parenthesis. In addition, a pair of parentheses must follow the table name. Information is returned in machine-readable format.

If the query plan cache does not contain any stored plans, no rows are returned.

For example, this query selects all columns from the QUERYCACHE virtual table. Only one plan was stored in the cache as reflected by the NUM_ENTRIES value of 1.

```
SELECT * FROM TABLE(QUERYCACHE());
AVG_TEMPLATE_SIZE             31
CURRENT_SIZE                  35
MAX_CACHE_SIZE              1024
MAX_NUM_VICTIMS               10
NUM_ENTRIES                    1
NUM_PLANS                      2
NUM_COMPILES                   2
NUM_RECOMPILES                53
NUM_RETRIES                    0
NUM_CACHEABLE_PARSING          0
NUM_CACHEABLE_BINDING          0
NUM_CACHE_HITS_PARSING         2
NUM_CACHE_HITS_BINDING         0
NUM_CACHEABLE_TOO_LARGE        0
NUM_DISPLACED                  0
OPTIMIZATION_LEVEL             3
TEXT_CACHE_HITS                0
AVG_TEXT_SIZE                500
TEXT_ENTRIES                   0
DISPLACED_TEXTS                0
NUM_LOOKUPS                    2

--- 1 row(s) selected.
```

Although the QUERYCACHE function is a one-row multi-column table, this output has been formatted for readability.

This query selects all columns of the QUERYCACHEENTRIES virtual table (formatted for readability):

```
SELECT * FROM TABLE(QUERYCACHEENTRIES());
```

```
ROW_ID    PLAN_ID             TEXT                          ENTRY_SIZE
------    ------------------  --------------------------    ----------
     0    211894097543468116  select * from employee;            32410
     1    211894097552547493  select * from job;                 24968
     2    211894097548497817  select * from dept;                29730

NUM_HITS    PHASE       OPTIMIZATION_LEVEL       CATALOG_NAME   SCHEMA_NAME
--------    ----------   ----------------------    ------------   -----------
       1     BINDING      2                          SAMDBCAT       PERSNL
       0     BINDING      2                          SAMDBCAT       PERSNL
       0     BINDING      2                          SAMDBCAT       PERSNL

NUM_PARAMS    PARAM_TYPES    PLAN_LENGTH   COMPILATION_TIME
----------    -----------    -----------   -----------------
       0                           31752          334
       0                           24504           54
       0                           29144           96

AVERAGE_HIT_TIME    SHAPE    ISOLATION_LEVEL_    ACCESS_MODE    AUTO_COMMIT
                             FOR_UPDATES
----------------    -------  ---------------     ------------    -----------
41                  Cut (0)   READ COMMITTED     READ/WRITE         ON
0                   Cut (0)   READ COMMITTED     READ/WRITE         ON
0                   Cut (0)   READ COMMITTED     READ/WRITE         ON

ROLLBACK_MODE
--------------      -----------
NOT SPECIFIED
NOT SPECIFIED
NOT SPECIFIED
```

```
--- 3 row(s) selected.
```

Reviewing the Query Plan Caching Statistics With the DISPLAY_QC and DISPLAY_QC_ENTRIES Commands

The DISPLAY_QC and DISPLAY_QC_ENTRIES commands provide a quick look at the most commonly accessed columns of the query plan caching statistics. The commands are entered at the MXCI prompt with no parameters. If no query plans have been cached, no rows are returned.

DISPLAY_QC Command

The DISPLAY_QC command accesses the information in the QUERYCACHE function and displays these columns:

Column Name	Type	Source column in QUERYCACHE Function
AVGSIZE	CHAR(8)	AVG_PLAN_SIZE
CURSIZE	CHAR(8)	CURRENT_SIZE
MAXSIZE	CHAR(8)	MAX_CACHE_SIZE
NPINNED	CHAR(8)	NUM_PINNED
NRECOM	CHAR(8)	NUM_RECOMPILES
NRETR	CHAR(8)	NUM_RETRIES
NCACHE	CHAR(8)	NUM_CACHEABLE_PARSING + NUM_CACHEABLE_BINDING
NHITS	CHAR(8)	NUM_CACHE_HITS_PARSING + NUM_CACHE_HITS_BINDING

```
>>DISPLAY_QC;

AVGSIZE   CURSIZE   MAXSIZE   NPINNED   NRECOM    NRETR     NCACHE    NHITS
-------  --------  --------  --------  --------  --------  --------  --------

31        35        1024      0         0         0         1         0

--- SQL operation complete.
```

DISPLAY_QC_ENTRIES Command

The DISPLAY_QC_ENTRIES command accesses the information in the QUERYCACHEENTRIES function and displays these columns:

Column Name	Type	Source column in QUERYCACHEENTRIES Function
ROWID	CHAR(8)	ROW_ID
TEXT	CHAR(36)	TEXT
NUMHITS	CHAR(8)	NUM_HITS
PH	CHAR(1)	PHASE
COMPTIME	CHAR(8)	COMPILATION_TIME
AVGHITTIME	CHAR(8)	AVERAGE_HIT_TIME

```
DISPLAY_QC_ENTRIES;

ROWID    TEXT                                 NUMHITS   PH COMPTIME  AVGHTIME
-------- ------------------------------------ --------  -  --------  --------

0        select * from job;                   0         B  88        0
1        select * from dept;                  0         B  115       0
2        select * from employee;              0         B  1605      0

--- SQL operation complete.
```

7 SQL/MX Operators

Use the information in this section to understand the DESCRIPTION column when you use the EXPLAIN function and when you view query execution plans with the Visual Query Planner. Operators are frequently called nodes or node types throughout the SQL/MX documentation set. For information about using the EXPLAIN function and Visual Query Planner, see Section 4, Reviewing Query Execution Plans.

This section defines all operators in alphabetic order:

For information about reading the EXPLAIN output, see Description of the EXPLAIN Function Results on page 4-3.

Operator groups are used to conveniently group operators of a similar type. For example, the Join group contains all operators that use joins.

Group	**Operator**
User-Defined Routine	CALL
DAM Subset	FILE_SCAN INDEX_SCAN SUBSET_DELETE SUBSET_UPDATE
DAM Unique	CURSOR_DELETE CURSOR_UPDATE FILE_SCAN_UNIQUE INDEX_SCAN_UNIQUE UNIQUE_DELETE UNIQUE_UPDATE
Data Mining	SAMPLE SEQUENCE TRANSPOSE
Exchange	ESP_EXCHANGE PARTITION_ACCESS SPLIT_TOP
Groupby	HASH_GROUPBY HASH_PARTIAL_GROUPBY_LEAF HASH_PARTIAL_GROUPBY_ROOT SHORTCUT_SCALAR_AGRR SORT_GROUPBY SORT_PARTIAL_AGGR_LEAF SORT_PARTIAL_AGGR_ROOT SORT_PARTIAL_GROUPBY_LEAF SORT_PARTIAL_GROUPBY_ROOT SORT_SCALAR_AGGR
Insert	INSERT INSERT_VSBB

Group	Operator
Join	HYBRID_HASH_ANTI_SEMI_JOIN HYBRID_HASH_JOIN HYBRID_HASH_SEMI_JOIN LEFT_HYBRID_HASH_JOIN LEFT_MERGE_JOIN LEFT_NESTED_JOIN LEFT_ORDERED_HASH_JOIN MERGE_ANTI_SEMI_JOIN MERGE_JOIN MERGE_SEMI_JOIN NESTED_ANTI_SEMI_JOIN NESTED_JOIN NESTED_SEMI_JOIN ORDERED_HASH_ANTI_SEMI_JOIN ORDERED_HASH_JOIN ORDERED_HASH_SEMI_JOIN TUPLE_FLOW
Materialize	MATERIALIZE
Union	BLOCKED_UNION MERGE_UNION ORDERED_UNION UNARY_UNION
Root	ROOT
Rowset	PACK UNPACK
Sort	SORT
Stored Function	EXPLAIN
Tuple	EXPR TUPLELIST VALUES

Operators

BLOCKED_UNION Operator

The BLOCKED_UNION operator always executes the left child element first and then the right. The execution of the right child is always blocked until that of the left child completes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
sort_order	Sort order of the result of the union.	text
merge_expression	Expression used to determine which child operator to read from next—if true, read from left; if false, read from right.	expr(text)
union_type	Merge, physical, or unspecified.	text
condExpr	Expression used for conditional union. Occurs with the IF statement in compound statements.	expr(text)
trigExceptExpr	Expression used for trigger exceptions.	expr(text)

The following is an example of the BLOCKED_UNION operator:

```
create table table_a
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, CONSTRAINT table_a_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
 ASC) NOT DROPPABLE
);

create table table_b
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, owner_count int
, CONSTRAINT table_b_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

ALTER TABLE table_a
ADD CONSTRAINT table_a_KEY FOREIGN KEY
(col1, col2) REFERENCES
```

```
table_b(col1, col2) on update restrict on delete restrict
DROPPABLE;

CREATE TRIGGER table_a_Owner_Count
AFTER INSERT ON table_a
REFERENCING NEW AS newrow
FOR EACH ROW
UPDATE table_b SET owner_count = (
SELECT count(*) FROM table_a
 WHERE (table_b.col1,table_b.col2)=
 (table_a.col1,table_a.col2)
 AND table_a.col3 = 1
 )
 WHERE (table_b.col1,table_b.col2)=
 (newrow.col1,newrow.col2);

insert into table_b values('A', 1, 1, 0);

Prepare TestQuery11 from
insert into table_a values('A', 1, 1);

DESCRIPTION
   fragment_id ............ 0
   parent_frag ............ (none)
   fragment_type .......... master
   union_type ............. merge
```

CALL Operator

User-Defined Routine (UDR)

The CALL operator indicates that a UDR was used.

The CALL operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
input_values	Input values to the CALL statement. One SQL/MX expression is returned for each IN or INOUT parameter. Nothing is returned for OUT parameters.	expr(text)

The max_results_sets token is supported only on systems running J06.05 and later J-series RVUs and H06.16 and later H-series RVUs.

Token	Followed by ...	Data Type
parameter_modes	A sequence of characters that specifies SQL parameter modes for the procedure. *I* is used for an IN parameter, *O* for an OUT parameter, and *N* for an INOUT parameter. Characters are separated by a single space. The value *none* is returned if the procedure has no SQL parameters.	text
routine_name	ANSI name of the procedure	text
routine_label	Guardian name of the stored procedure label	text
sql_access_mode	SQL access mode of the procedure	text
external_name	Java method name	text
external_path	OSS directory or JAR file path that contains the Java class file	text
external_file	Java class name, possibly prefixed by a package name, that contains the SPJ method	text
signature	Java signature of the SPJ method in internal Java Virtual Machine (JVM) format	text
language	Language of the SPJ method, which is always Java	text
runtime_options	UDR_JAVA_OPTIONS setting under which the CALL statement was compiled	text
runtime_option_delimiters	A single-quoted string representing the option delimiter character in the UDR_JAVA_OPTIONS string, which is always a single space character.	text
max_results_sets	The maximum number of result sets the stored procedure can return.	Integer

The max_results_sets token is supported only on systems running J06.05 and later J-series RVUs and H06.16 and later H-series RVUs.

The following is an example of the CALL operator:

```
create procedure u300populateA (
  in table_name char(20)
)
external name 'TEST300.populateA' language java parameter style
java
external path 'W:/regress/udr' modifies sql data;

prepare TestQuery36 from
call u300populateA('customer');

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  parameter_modes ........ I
```

```
routine_name ........... CAT.SCH.U300POPULATEA
routine_label .......... \DMR11.$DATA04.ZSDJG12X.WF32X300
sql_access_mode ........ MODIFIES SQL DATA
external_name .......... populateA
external_path .......... ./
external_file .......... TEST300
signature .............. (Ljava/lang/String;)V
language ............... Java
runtime_options ........ OFF
runtime_option_delimite  ' '
max_result_sets ........ 0
input_values ........... cast('customer' AS CHAR(20) CHARACTER
                           SET ISO88591)
```

CURSOR_DELETE Operator

DAM Unique Group

The CURSOR_DELETE operator describes a portion of an execution plan that works on one row only. The CURSOR_DELETE operation is performed by first retrieving rows from the table and then deleting each row that is required. This operation differs from SUBSET_DELETE, in which the read and delete are performed in a combined operation.

The CURSOR_DELETE operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short and simple operations is used. Its value is *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
iud_type	Type of delete followed by table or index name.	expr(text)
begin_key	Expression of the begin key predicates.	expr(text)

index_begin_key	Expression of the begin key predicates on index.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. Displayed only if partitioning key differs from clustering key.	expr(text)
check_constraint	Check constraints in the delete table.	expr(text)

The following is an example of the CURSOR_DELETE operator:

```
PREPARE TestQuery1 FROM
DELETE FROM customer
WHERE c_nationkey<300;

DESCRIPTION
  olt_optimization ....... used
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  Scan_Direction ......... forward
  olt_optimization ....... used
  olt_opt_lean ........... not used
  iud_type ............... index_ cursor_delete
  DETCAT.DETSCH.CX1
  lock_mode .............. not specified, defaulted to lock
  cursor
  access_mode ............ not specified, defaulted to read
  committed
  columns_retrieved ...... 3
  begin_key .............. (C_NATIONKEY =
                            DETCAT.DETSCH.CUSTOMER.C_NATIONKEY)
                           and (C_CUSTKEY =
                             DETCAT.DETSCH.CUSTOMER.C_CUSTKEY)
```

CURSOR_UPDATE Operator

DAM Unique Group

The CURSOR_UPDATE operator describes a portion of an execution plan that works on one row only. The CURSOR_UPDATE operation is performed by first retrieving rows from the table and then updating each row that is required. This operation differs from SUBSET_UPDATE, in which the read and update are performed in a combined operation.

The CURSOR_UPDATE operator has no child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short and simple operations is used. Its value is *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
iud_type	Type of update followed by table or index name.	expr(text)
new_rec_expr	Computation of the row to be updated.	expr(text)
begin_key	Expression of the begin key predicate.	expr(text)
index_begin_key	Expression of the begin key predicates on index.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
check_constraint	Check constraints in the update table.	expr(text)

The following is an example of the CURSOR_UPDATE operator:

```
UPDATE table_b SET owner_count = (
  SELECT count(*) FROM table_a
    WHERE (table_b.col1,table_b.col2)=
          (table_a.col1,table_a.col2)
       AND table_a.col3 = 1
);

DESCRIPTION
  fragment_id ............ 4
  parent_frag ............ 0
  fragment_type .......... dp2
  Scan_Direction ......... forward
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  iud_type ............... cursor_update DETCAT.DETSCH.TABLE_B
```

```
lock_mode ............. not specified, defaulted to lock
cursor
access_mode ........... not specified, defaulted to read
committed
columns_retrieved ...... 4
new_rec_expr ........... (OWNER_COUNT assign count(1 ))
begin_key ................ (COL1 = COL1) and (COL2 = COL2)
```

ESP_ACCESS Operator

The ESP_ACCESS operator is a form of ESP_EXCHANGE operator. An ESP_ACCESS operator isolates the SEQUENCEGENERATOR operator into its own ESP process. The SEQUENCEGENERATOR operator is isolated because it updates the sequence generator table by using a new transaction, and does not use the user transaction.

The ESP_ACCESS operator appears above the SEQUENCEGENERATOR operator in the query plan. The parent for this operator is the NEXTVALUEFOR operator, which can reside in either the master executor or an ESP.

When the NEXTVALUEFOR operator is in the master executor, the ESP_ACCESS operator defines the process boundary between the master executor and a single ESP that contains the SEQUENCEGENERATOR operator.

When the NEXTVALUEFOR operator is in an ESP, the ESP_ACCESS operator defines the boundary between the ESP layers of NEXTVALUEFOR operator and the SEQUENCEGENERATOR operator.

The description for this operator contains the following:

Token	Followed by ...	Data Type
max_card_set	Cardinality estimate for the operator.	integer
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is (none) for the master executor.	integer
fragment_type	ESP.	text
buffer_size	Size of message buffer.	integer
record_length	Number of bytes in the record sent.	integer
parent_processes	Indicates the number of processes that the ESPs in the ESP_ACCESS operator communicate with either additional ESPs (as indicated by another ESP_ACCESS operator) or the master process.	integer
child_processes	Indicates the number of processes that supply the ESPs of an operator with rows. The number of ROWS_OUT (also called CARDINALITY) from the operator that supplies the ESP_ACCESS operator with data indicates how many rows the ESP_ACCESS operator is expected to receive.	integer
bottom_node_map	Associates each process at the bottom to a processor.	text

ESP_EXCHANGE Operator

Exchange Group

An ESP_EXCHANGE operator describes a portion of an execution plan that redistributes the input data stream. This operator represents an interface between ESPs, between the master executor and one or more ESPs, or between an ESP process and a DAM process. For more information about exchange operators, see Section 8, Parallelism.

The ESP_EXCHANGE operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
buffer_size	Size of message buffer.	integer
record_length	Number of bytes in the record sent.	integer
parent_processes	Indicates the number of processes that the ESPs in the ESP_EXCHANGE operator communicates with either additional ESPs (as indicated by another ESP_EXCHANGE operator) or the master process.	integer
child_processes	Indicates the number of processes that supply the ESPs of an operator with rows. The number of ROWS_OUT (also called CARDINALITY) from the operator that supplies the ESP_EXCHANGE operator with data indicates how many rows the ESP_EXCHANGE operator is expected to receive.	integer
parent_partitioning_function	Type of top partitioning and contains summary information about the parallel plan.	text
child_partitioning_function	Indicates how the input data received by the ESP_EXCHANGE operator is organized.	text
merged_order	Expression describing sort keys used to control the interaction between the parent process (usually the master) and ESPs when the result is ordered.	expr(text)
bottom_partition_input_values	Internal values that identify the part of the data the ESP will work on.	text
partitioning_expression	Expression used to partition data.	expr(text)

The following is an example of the ESP_EXCHANGE operator:

```
create table tabl3 (a int, b int, c int, d int, e int);

create table tabl4 like tabl3;
create view view30 as
select * from tabl3 union all
select * from tabl3 union all
select * from tabl3 union all
select * from tabl3 union all
select * from tabl3;

create view view40 as
select * from tabl4 union all
select * from tabl4 union all
select * from tabl4 union all
select * from tabl4 union all
select * from tabl4;

insert into tabl3 values
(1,2,3,1,1),(4,5,6,1,1),(7,8,9,1,1),(1,2,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,5,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(1,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(1,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(9,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(1,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(1,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(1,2,3,1,1),(4,4,6,1,1),(7,7,9,1,1),(1,9,3,1,1),(4,5,6,1,1),
(7,8,9,1,1),
(4,5,6,1,1),(7,8,9,1,1);

insert into tabl4 select * from tabl3;
insert into tabl3 select * from tabl4;

?section test
CONTROL QUERY DEFAULT CHECK_CONSTRAINT_PRUNING 'ON';
CONTROL QUERY DEFAULT INTERACTIVE_ACCESS 'ON';
CONTROL QUERY DEFAULT UNION_TRANSITIVE_PREDICATES 'ON';
CONTROL QUERY DEFAULT COMP_BOOL_25 'ON';
CONTROL QUERY DEFAULT MERGE_JOINS 'OFF';
CONTROL QUERY DEFAULT HASH_JOINS 'OFF';
CONTROL QUERY DEFAULT OPTIMIZER_PRUNING 'OFF';
CONTROL QUERY DEFAULT DATA_FLOW_OPTIMIZATION 'OFF';
CONTROL QUERY DEFAULT METADATA_CACHE_SIZE '200';
CONTROL QUERY DEFAULT OPTIMIZATION_LEVEL_1_SAFETY_NET '100000';
CONTROL QUERY DEFAULT MULTIUNION 'ON';
CONTROL QUERY DEFAULT ATTEMPT_ESP_PARALLELISM 'ON';
CONTROL QUERY DEFAULT  DEF_NUM_SMP_CPUS '1';
CONTROL QUERY DEFAULT  PARALLEL_NUM_ESPS '2';
CONTROL QUERY DEFAULT DETAILED_STATISTICS 'ALL';

control query shape
nested_join(esp_exchange(cut),esp_exchange(cut));

prepare vv from
select * from view30, view40 where view30.a = view40.b;

DESCRIPTION
  fragment_id ........... 13
  parent_frag ............ 0
  fragment_type .......... esp
  buffer_size ............ 6,250
  record_length ......... 40
  parent_processes ....... 1
  child_processes ........ 2
  child_partitioning_func  hash partitioned 2 ways on
  (([0]ValueIdUnion(\DMR11.$DATA04.ZSDWC1HX.ZVB7V200
  .A, \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.A,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.A,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.A,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.A),
  [0]ValueIdUnion(\DMR11.$DATA04.ZSDWC1HX.ZVB7V200.B
  , \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.B,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.B,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.B,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.B),
  [0]ValueIdUnion(\DMR11.$DATA04.ZSDWC1HX.ZVB7V200.C
  , \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.C,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.C,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.C,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.C),
  [0]ValueIdUnion(\DMR11.$DATA04.ZSDWC1HX.ZVB7V200.D
  , \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.D,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.D,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.D,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.D),
  [0]ValueIdUnion(\DMR11.$DATA04.ZSDWC1HX.ZVB7V200.E
  , \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.E,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.E,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.E,
  \DMR11.$DATA04.ZSDWC1HX.ZVB7V200.E)))
  bottom_partition_input_  \:_sys_HostVarLoHashPart,
  \:_sys_HostVarHiHashPart
```

EXPLAIN Operator

Stored Function Group

The EXPLAIN operator executes a stored function. The operator for an EXPLAIN operator is always EXPLAIN.

EXPLAIN has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
function_parameters	Parameters from the call to the EXPLAIN function.	expr(text)

The following is an example for EXPLAIN operator:

```
prepare TestQuery28 from
SELECT * FROM TABLE (EXPLAIN (NULL,'%'));

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  function_parameters ...... NULL, '%'
```

EXPR Operator

Tuple Group

The EXPR operator calculates an expression for each row it receives from its child node and returns that expression to its parent node.

The EXPR operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
tuple_expr	The tuple produced by this node	expr(text)

EXPLAIN_CMD Operator

The EXPLAIN_CMD operator is generated when the EXPLAIN statement is used and it has no children. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent fragment of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text

The following is an example for EXPLAIN_CMD Operator:

```
prepare TestQuery29 from
EXPLAIN options 'e' TestQuery28;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type ............ master
```

FILE_SCAN Operator

DAM Subset Group

The FILE_SCAN operator contains details about how a certain access path is scanned, such as lock_mode and scan_direction.

The FILE_SCAN operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short, simple operations is used. Its value is *`used`* or *`not used`*.	text

Token	**Followed by ...**	**Data Type**
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
scan_type	FILE_SCAN followed by table name.	text
scan_direction	Direction in which table is scanned: forward or reverse	text
lock_mode	The lock mode specified: shared, exclusive, not specified (defaulted to lock cursor), or unknown.	text
access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
executor_predicate	Any predicate expression that is not a key predicate evaluated by the executor in DAM.	expr(text)
columns_retrieved	Estimate of the number of columns to be returned.	integer
fast_scan	Indicates whether an optimization for a simple scan operation is used. The value *used* is returned if this optimization is used.	text
fast_replydata_move	Indicates whether an optimization for returning data from DAM is used. The value *used* is returned if this optimization is used.	text
key_columns	Columns used as the primary key.	expr(text)
mdam_disjunct	Disjunct key predicates used by MDAM.	expr(text)
begin_key	Expression of the begin key predicate.	expr(text)
end_key	Expression of the end key predicate.	expr(text)
key_type	Simple or MDAM.	text
part_key_predicate	Predicate expression specified on partitioning key. Displayed only if partitioning key differs from clustering key.	expr(text)

The following is an example of the FILE_SCAN operator:

```
PREPARE TestQuery2 FROM
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address,
s_phone, s_comment
FROM part,supplier,partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (SELECT MIN(ps_supplycost)
FROM partsupp ps1,supplier s1, nation n1,region r1
WHERE p_partkey = ps1.ps_partkey
AND s1.s_suppkey = ps1.ps_suppkey
AND s1.s_nationkey = n1.n_nationkey
AND n1.n_regionkey = r1.r_regionkey
AND r1.r_name = 'EUROPE')
ORDER BY s_acctbal desc, n_name, s_name, p_partkey;

DESCRIPTION
  fragment_id ........... 10
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. subset scan of table
                           DETCAT.DETSCH.PARTSUPP PS1
  scan_direction ......... forward
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 5
  fast_scan .............. used
  fast_replydata_move .... used
  key_columns ............ PS_PARTKEY, PS_SUPPKEY
  begin_key .............. (PS_PARTKEY =
                           DETCAT.DETSCH.PSX1.PS_PARTKEY),
                           PS_SUPPKEY = <min>)
  end_key ................ (PS_PARTKEY =
                           DETCAT.DETSCH.PSX1.PS_PARTKEY),
                            (PS_SUPPKEY = <max>)
```

FILE_SCAN_UNIQUE Operator

DAM Unique Group

The FILE_SCAN_UNIQUE operator describes a portion of an execution plan where you are scanning for a unique key value. It selects zero or one row.

The FILE_SCAN_UNIQUE operator has no child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	Integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	Integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short, simple operations is used. Value is used if this optimization is used. Its value is *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
key_columns	Columns used as the primary key.	expr(text)
key	Expression of the key predicate.	expr(text)
scan_type	Unique access of table, followed by table name.	text
key_type	Simple or MDAM.	text
lock_mode	The lock specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
columns_retrieved	Estimate of the number of columns to be returned.	integer
executor_predicates	Any predicate that is not a key predicate evaluated by the executor in DAM.	expr(text)

Token	Followed by ...	Data Type
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
fast_replydata_move	Indicates whether an optimization for returning data from DAM is used. The value *used* is returned if this optimization is used.	text
fast_scan	Indicates whether an optimization for a simple scan operation is used. The value *used* is returned if this optimization is used.	text

The following is an example of the FILE_SCAN_UNIQUE operator:

```
PREPARE TestQuery2 FROM
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address,
s_phone, s_comment
FROM part,supplier,partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (SELECT MIN(ps_supplycost)
FROM partsupp ps1,supplier s1, nation n1,region r1
WHERE p_partkey = ps1.ps_partkey
AND s1.s_suppkey = ps1.ps_suppkey
AND s1.s_nationkey = n1.n_nationkey
AND n1.n_regionkey = r1.r_regionkey
AND r1.r_name = 'EUROPE')
ORDER BY s_acctbal desc, n_name, s_name, p_partkey;

DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. unique access of table
                           DETCAT.DETSCH.PART
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 9
  fast_replydata_move .... used
  key_columns ............ P_PARTKEY
  executor_predicates .... (P_TYPE like '%BRASS') and (P_SIZE =
                           15)
  key .................... (P_PARTKEY =
                             DETCAT.DETSCH.PSX1.PS_PARTKEY)
```

FirstN Operator

The FirstN operator describes a portion of execution plan that selects only first few rows of output. This operator has only one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	Integer
fragment_type	Master, ESP, or DP2.	text

The following is an example for FirstN operator:

```
PREPARE TestQuery6 FROM
SELECT [FIRST 100] s_acctbal, s_name, n_name, p_partkey, p_mfgr,
s_address, s_phone, s_comment
FROM part,supplier,partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (SELECT MIN(ps_supplycost)
FROM partsupp, supplier, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE')
ORDER BY s_acctbal desc, n_name, s_name, p_partkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type ............ master
```

HASH_GROUPBY Operator

Groupby Group

The HASH_GROUPBY operator describes a portion of an execution plan that affects a group. The group values are computed by hashing individual rows into a hash table. When a new row is received, the executor hashes to the hash table and performs the aggregate in the hash table.

The HASH_GROUPBY operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicates	Expression of the HAVING clause.	expr(text)
grouping_columns	Expression of the grouping columns.	expr(text)

The following is an example of the HASH_GROUPBY operator:

```
PREPARE TestQuery4 FROM
SELECT l_orderkey,
CAST(SUM(l_extendedprice*(1-l_discount))AS
NUMERIC(18,2)), o_orderdate, o_shippriority
FROM customer,orders,lineitem
WHERE c_mktsegment = 'BUILDING'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < DATE '1995-03-15'
AND l_shipdate > DATE '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
HAVING sum(l_extendedprice)> 100
ORDER BY 2 DESC,3 ASC;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  grouping_columns ....... DETCAT.DETSCH.OX2.O_ORDERDATE,
                           DETCAT.DETSCH.OX2.O_SHIPPRIORITY,
                           DETCAT.DETSCH.OX2.O_ORDERKEY
  aggregates ............
                    sum(DETCAT.DETSCH.LX3.L_EXTENDEDPRICE),
                    sum((cast(DETCAT.DETSCH.LX3.L_EXTENDEDPRICE
                    AS BIG NUM(12,2) SIGNED) * cast((cast((1 *
                    100) AS NUMERIC(13,2) SIGNED) -
                    DETCAT.DETSCH.LX3.L_DISCOUNT) AS BIG NUM(13,
                    2) SIGNED)))
  selection_predicates ...
```

```
                  (sum(DETCAT.DETSCH.LX3.L_EXTENDEDPRICE) >
                   cast((100   * 100) AS NUMERIC(18,2) SIGNED))
```

Note. The query mentioned in the example provides the HASH_GROUPBY operator in Windows NT. To get this operator in the NonStop operating system, use some more CQS.

HASH_PARTIAL_GROUPBY_LEAF Operator

Groupby Group

The HASH_PARTIAL_GROUPBY_LEAF operator executes a partial group by operation as close to where the data is read as is cost effective. This strategy reduces the amount of data that must be relocated for a query. When executed in DAM, the HASH_PARTIAL_GROUPBY_LEAF is limited to a small amount of memory. Any group that does not fit in memory is passed on ungrouped, with the full grouping occurring at the HASH_PARTIAL_GROUPBY_ROOT. If the HASH_PARTIAL_GROUPBY_LEAF is not executed in DAM, more memory is available, and all rows are grouped. The groups from multiple processes are then rolled up in the HASH_PARTIAL_GROUPBY_ROOT.

The operator must always be accompanied by a HASH_PARTIAL_GROUPBY_ROOT operator above it in the tree, which finalizes the query.

The HASH_PARTIAL_GROUPBY_LEAF operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicates	Expression of the HAVING clause.	expr(text)
grouping_columns	Expression of the grouping columns.	expr(text)

The following is an example of the HASH_PARTIAL_GROUPBY_LEAF operator:

```
control query shape
hash_groupby(partition_access(hash_groupby(scan)));

prepare TestQuery32 from
select d from t016pt1 group by d;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  grouping_columns ......... DETCAT.DETSCH.T016PT1.D
```

HASH_PARTIAL_GROUPBY_ROOT Operator

Groupby Group

The HASH_PARTIAL_GROUPBY_ROOT operator works together as a pair with the HASH_PARTIAL_GROUPBY_LEAF operator. The HASH_PARTIAL_GROUPBY_ROOT operator finalizes the group by at the ESP level. See HASH_PARTIAL_GROUPBY_LEAF Operator on page 7-24.

The HASH_PARTIAL_GROUPBY_ROOT operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is (none) for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicates	Expression of the HAVING clause.	expr(text)
grouping_columns	Expression of the grouping columns.	expr(text)

The following is an example of the HASH_PARTIAL_GROUPBY_ROOT operator:

```
control query shape
hash_groupby(partition_access(hash_groupby(scan)));

prepare TestQuery32 from
select d from t016pt1 group by d;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  grouping_columns ......... DETCAT.DETSCH.T016PT1.D
```

HYBRID_HASH_ANTI_SEMI_JOIN Operator

Join Group

The HYBRID_HASH_ANTI_SEMI_JOIN operator returns rows from the outer table where no match occurs. Also see HYBRID_HASH_JOIN Operator on page 7-27 and HYBRID_HASH_SEMI_JOIN Operator on page 7-28.

The HYBRID_HASH_ANTI_SEMI_JOIN has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, natural, left, inner semi, or inner anti-semi-join	text
join_method	Name of join method: hash	text
join_predicate	Expression of the join predicate, specified in the ON clause. Used for semi and outer joins.	expr(text)
hash_join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause.	expr(text)

The difference between the join_predicate and hash_join_predicate tokens is that the former are the nonequijoin predicates, while the latter are equijoin predicates that you use to help build and probe the hash table.

The following is an example of the HYBRID_HASH_ANTI_SEMI_JOIN operator:

```
PREPARE TestQuery5 FROM
SELECT * FROM partsupp, part
WHERE p_brand <>'Brand#45'
AND ps_suppkey
NOT IN (SELECT s_suppkey FROM supplier
WHERE
s_comment LIKE '%Better BusinessBureauComplaints%');

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type ............. inner anti-semi
  join_method ........... hash
  hash_join_predicate .... (DETCAT.DETSCH.PARTSUPP.PS_SUPPKEY =
                             DETCAT.DETSCH.SUPPLIER.S_SUPPKEY)
```

HYBRID_HASH_JOIN Operator

Join Group

The HYBRID_HASH_JOIN operator joins the data from two child tables. It creates a hash table for the inner table and joins the outer table by hashing each outer row and looking for matches in the hash table. This operator can overflow to disk when the inner table is too large to fit in memory. Equijoins and cross-products are supported by this operator.

The HYBRID_HASH_JOIN has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, natural, left, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
hash_join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the HYBRID_HASH_JOIN operator:

```
PREPARE TestQuery6 FROM
SELECT [FIRST 100] s_acctbal, s_name, n_name, p_partkey, p_mfgr,
s_address, s_phone, s_comment
FROM part,supplier,partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (SELECT MIN(ps_supplycost)
FROM partsupp, supplier, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE')
ORDER BY s_acctbal desc, n_name, s_name, p_partkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.NATION.N_REGIONKEY =
                             DETCAT.DETSCH.REGION.R_REGIONKEY)
```

HYBRID_HASH_SEMI_JOIN Operator

Join Group

The HYBRID_HASH_SEMI_JOIN returns only one row for every outer row, regardless of the number of matches. The HYBRID_HASH_SEMI_JOIN operator differs from the HYBRID_HASH_JOIN operator only when it finds multiple matches in the inner table. In the HYBRID_HASH_JOIN case, a result row is returned for every match in the inner table. See HYBRID_HASH_JOIN Operator on page 7-27.

The HYBRID_HASH_SEMI_JOIN operator has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
hash_join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the HYBRID_HASH_SEMI_JOIN operator:

```
prepare TestQuery31 from
select * from TAB1
where TAB1.col1 in (select col2 from TAB2 where TAB2.col1 < 10);

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner semi
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.TAB1.COL1 =
                             DETCAT.DETSCH.TAB2.COL2)
```

INDEX_SCAN Operator

DAM Subset Group

The INDEX_SCAN operator scans the index built on the key columns. The node description contains details about how a certain access path is scanned, such as lock_mode and scan_direction.

The INDEX_SCAN operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short, simple operations is used. Value is used if this optimization is used. Its value is *used* or *not used*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *used* or *not used*.	text
key_columns	Columns used as the primary key.	expr(text)
begin_key	Expression of the begin key predicates.	expr(text)
end_key	Expression of the end key predicates.	expr(text)
scan_type	INDEX_SCAN followed by table or index name.	text

Token	Followed by ...	Data Type
scan_direction	Direction in which table is scanned: forward or reverse.	text
lock_mode	The lock mode specified: shared, exclusive, not specified (defaulted to lock cursor), or unknown.	text
access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
key_type	Simple or MDAM.	text
executor_predicates	Any predicate expression that is not a key predicate evaluated by the executor in DAM.	expr(text)
columns_retrieved	Estimated number of columns to be returned.	integer
fast_replydata_move	Indicates whether an optimization for returning data from DAM is used. The value *used* is returned if this optimization is used.	text
fast_scan	Indicates whether an optimization for a simple scan operation is used. The value *used* is returned if this optimization is used.	text
part_key_predicate	Predicate expression specified on the partitioning key. Displayed only if partitioning key differs from clustering key.	expr(text)
mdam_disjunct	Disjunct key predicates used by MDAM.	expr(text)

The following is an example of the INDEX_SCAN operator:

```
control query default nested_joins 'off';
control query default hash_joins 'off';
prepare TestQuery37b from
SELECT *
FROM customer LEFT JOIN nation ON c_nationkey = n_nationkey
WHERE c_custkey > 1000 AND c_custkey < 1010
ORDER BY c_custkey;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. subset scan of index
                      DETCAT.DETSCH.CX1(DETCAT.DETSCH.CUSTOMER)
  scan_direction ......... forward
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 3
```

```
fast_scan .............. used
fast_replydata_move .... used
key_columns ............ DETCAT.DETSCH.CX1.C_NATIONKEY,
                         DETCAT.DETSCH.CX1.C_CUSTKEY
executor_predicates .... (DETCAT.DETSCH.CX1.C_CUSTKEY > 1000)
                         and(DETCAT.DETSCH.CX1.C_CUSTKEY <
                         1010) and
                         (DETCAT.DETSCH.CX1.C_CUSTKEY =
                         DETCAT.DETSCH.CX1.C_CUSTKEY)
begin_key .............. (DETCAT.DETSCH.CX1.C_NATIONKEY =
                         <min>),
                         (DETCAT.DETSCH.CX1.C_CUSTKEY = 1000)
end_key ................ (DETCAT.DETSCH.CX1.C_NATIONKEY =
                         <max>),
                           DETCAT.DETSCH.CX1.C_CUSTKEY = 1010)
```

INDEX_SCAN_UNIQUE Operator

DAM Unique Group

The INDEX_SCAN_UNIQUE operator describes a scan on the primary key column with an index on that column.

The INDEX_SCAN_UNIQUE operator has no child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
key_columns	Columns used as the primary key.	expr(text)
key	Expression of the key predicate.	expr(text)
scan_type	INDEX_SCAN_UNIQUE followed by table or index name.	text
lock_mode	The lock mode specified: shared, exclusive, not specified (defaulted to lock cursor), or unknown.	text
key_type	Simple or MDAM.	text

access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
executor_predicates	Any predicate expression that is not a key predicate evaluated by the executor in DAM.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
columns_retrieved	Estimated number of columns to be returned.	integer
fast_replydata_move	Indicates whether an optimization for returning data from DAM is used. The value *used* is returned if this optimization is used.	text
fast_scan	Indicates whether an optimization for a simple scan operation is used. The value *used* is returned if this optimization is used.	text
olt_optimization	Indicates whether an optimization for short, simple operations is used. The value *used* is returned if this optimization is used.	text

The following is an example of the INDEX_SCAN_UNIQUE operator:

```
prepare TestQuery14 from
select * from supplier
where s_nationkey = 12 and
s_suppkey = 14;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type ..............
                           unique access of index
                           CAT.SCH.SX1(CAT.SCH.SUPPLIER)
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 3
  fast_replydata_move .... used
  key_columns ............ CAT.SCH.SX1.S_NATIONKEY,
                           CAT.SCH.SX1.S_SUPPKEY
  executor_predicates .... (CAT.SCH.SX1.S_SUPPKEY = %(14)) and
                           (CAT.SCH.SX1.S_SUPPKEY = %(14))
  key .................... (CAT.SCH.SX1.S_NATIONKEY = %(12)),
                             (CAT.SCH.SX1.S_SUPPKEY = %(14))
```

INSERT Operator

INSERT Group

The INSERT operator describes that part of an execution plan that inserts a new row into a table. The operator for an INSERT operator is always INSERT.

The INSERT operator has no child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
new_rec_expr	Computation of the row to be inserted.	expr(text)
olt_optimization	Indicates whether an optimization for short, simple operations is used. Its value is *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
part_key_predicate	Predicate expression specified on the partitioning key. Displayed only if partitioning key differs from clustering key.	expr(text)
iud_type	Type of insert followed by table or index name.	expr(text)
check_constraint	Check constraints in the insert table.	expr(text)

The following is an example of the INSERT operator:

```
PREPARE TestQuery8 FROM
INSERT INTO custss SELECT *
FROM customer
WHERE c_nationkey IN (1,3,7,8,10,15,18,20,22,44);

DESCRIPTION
  fragment_id ............ 3
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  iud_type ............... insert DETCAT.DETSCH.CUSTSS
  lock_mode .............. not specified, defaulted to lock
```

```
                          cursor
access_mode ............ not specified, defaulted to read
                          committed
columns_retrieved ...... 8
new_rec_expr ........... (C_CUSTKEY assign
                          DETCAT.DETSCH.CUSTOMER.C_CUSTKEY),
                          (C_NAME assign
                          DETCAT.DETSCH.CUSTOMER.C_NAME),
                          (C_ADDRESS assign
                          DETCAT.DETSCH.CUSTOMER.C_ADDRESS),
                          (C_NATIONKEY assign
                          DETCAT.DETSCH.CUSTOMER.C_NATIONKEY),
                          (C_PHONE assign
                          DETCAT.DETSCH.CUSTOMER.C_PHONE),
                          (C_ACCTBAL assign
                          DETCAT.DETSCH.CUSTOMER.C_ACCTBAL),
                          (C_MKTSEGMENT assign
                          DETCAT.DETSCH.CUSTOMER.C_MKTSEGMENT),
                          (C_COMMENT assign
                            DETCAT.DETSCH.CUSTOMER.C_COMMENT)
```

INSERT_VSBB Operator

INSERT Group

The INSERT_VSBB operator describes that part of an execution plan that inserts multiple rows into a table in DAM. The operator for an INSERT_VSBB operator is always INSERT_VSBB.

For information about setting the CONTROL QUERY DEFAULT attribute for INSERT_VSBB, see the *SQL/MX Reference Manual*.

The INSERT_VSBB operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
iud_type	Type of insert followed by table or index name.	expr(text)

new_rec_expr	Computation of the row to be inserted.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
check_constraint	Check constraints in the insert table.	expr(text)

The following is an example of the INSERT_VSBB operator:

```
control query default insert_vsbb 'USER';

prepare TestQuery35 from
Insert NOLOG into Dup_B
Values
(1001,'www','I1'),(1004,'wwt','aa'),(1005,'rrr','B1'),(1007,'qqq
','ff'),(1010,'ppp','gg'),

(1011,'nlmlm','zz'),(1013,'naaa','ff'),(1015,'lll','nn'),(1016,'
iii','uu'),(1017,'hhh','PP'),(1026,'gggggg','y'),
(1035,'fff','ii'),(1037,'eeee','w'),(1040,'ddd','kk');

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  iud_type ............... insert_vsbb DETCAT.DETSCH.DUP_B
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 3
  new_rec_expr ........... (PRIM assign cast(%(1040) AS INTEGER
                           SIGNED)),(PRIM2 assign
                           cast(%('ddd') AS VARCHAR(20)
                           CHARACTER SET ISO88591)),
                           (CH assign cast(cast(cast(%('kk')
                           AS CHAR(2) CHARACTER SET ISO88591)
                           AS VARCHAR(8) CHARACTER
                           SET ISO88591) AS CHAR(2) CHARACTER
                             SET ISO88591))
```

LEFT_HYBRID_HASH_JOIN Operator

Join Group

The LEFT_HYBRID_HASH_JOIN operator returns an unmatched outer row even when it does not find a match in the inner table. Null values are supplied for the missing inner rows. The LEFT_HYBRID_HASH_JOIN operator differs from the HYBRID_HASH_JOIN only when it does not find a match in the inner table. See HYBRID_HASH_JOIN Operator on page 7-27.

The LEFT_HYBRID_HASH_JOIN operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
hash_join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the LEFT_HYBRID_HASH_JOIN operator:

```
control query default nested_joins 'off';

prepare TestQuery37a from
SELECT *
FROM customer LEFT JOIN nation ON c_nationkey = n_nationkey
WHERE c_custkey > 1000 AND c_custkey < 1010
ORDER BY c_custkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. left
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.CUSTOMER.C_NATIONKEY =
                             DETCAT.DETSCH.NATION.N_NATIONKEY)
```

LEFT_MERGE_JOIN Operator

Join Group

The LEFT_MERGE_JOIN operator describes a portion of an execution plan that involves a merge join. The LEFT_MERGE_JOIN differs from MERGE_JOIN only when it does not find a match in the inner table. When no match is found, the left row is still returned, and the data from the right table is set to null. See MERGE_JOIN Operator on page 7-42.

The LEFT_MERGE_JOIN operator has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join	text
join_method	Name of join method: merge	text
merge_join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the LEFT_MERGE_JOIN operator:

```
Control Query Default nested_joins 'off';
Control Query Default hash_joins 'off';

prepare TestQuery37b from
SELECT *
FROM customer LEFT JOIN nation ON c_nationkey = n_nationkey
WHERE c_custkey > 1000 AND c_custkey < 1010
ORDER BY c_custkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. left
  join_method ............ merge
  merge_join_predicate ... (DETCAT.DETSCH.CX1.C_NATIONKEY =
                             DETCAT.DETSCH.NATION.N_NATIONKEY)
```

LEFT_NESTED_JOIN Operator

Join Group

The LEFT_NESTED_JOIN operator describes a portion of an execution plan that involves a nested join. The LEFT_NESTED_JOIN sends each outer (left) row to the inner (right) child. The right child finds all the matches for a row and returns all the matches. If an outer row finds no matches in the inner table, the outer row is returned, and nulls are supplied for inner table values. See NESTED_JOIN Operator on page 7-47.

The LEFT_NESTED_JOIN has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: nested or in-order nested	text
join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the LEFT_NESTED_JOIN operator:

```
prepare TestQuery37 from
SELECT *
FROM customer LEFT JOIN nation ON c_nationkey = n_nationkey
WHERE c_custkey > 1000 AND c_custkey < 1010
ORDER BY c_custkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. left
  join_method .............. nested
```

LEFT_ORDERED_HASH_JOIN Operator

Join Group

The LEFT_ORDERED_HASH_JOIN operator returns an unmatched outer row even when it does not find a match in the inner table. Null values are supplied for the missing inner rows. The LEFT_ORDERED_HASH_JOIN operator differs from the LEFT_HYBRID_HASH_JOIN in that it preserves the order of the outer table and does not overflow to disk. In addition, the reuse feature enables reuse of the hash table for subsequent requests within the same query. Choose this operator when you need to preserve the order of the outer table or if you can benefit from the reuse feature. It should be chosen only if the inner table is small enough to fit in memory.

The LEFT_ORDERED_HASH_JOIN operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
hash_join_predicate	Expression of the join predicate.	expr(text)
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
reuse_comparison_values	List of values that cause the hash table to be rebuilt when they change.	expr(text)
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the LEFT_ORDERED_HASH_JOIN operator:

```
Control Query Default nested_joins 'off';

prepare TestQuery37a from
SELECT *
FROM customer LEFT JOIN nation ON c_nationkey = n_nationkey
WHERE c_custkey > 1000 AND c_custkey < 1010
ORDER BY c_custkey;
```

```
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. left
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.CUSTOMER.C_NATIONKEY =
                             DETCAT.DETSCH.NATION.N_NATIONKEY)
```

MATERIALIZE Operator

Materialize Group

When it first executes, the MATERIALIZE operator evaluates the query beneath it one time and stores the result of that evaluation in a temporary table, in addition to returning the result to the parent. In subsequent requests to the MATERIALIZE operator, it might return the stored temporary table instead of evaluating its child again. Use the MATERIALIZE operator when using correlated subqueries or in place of a HYBRID_HASH_JOIN when the outer order needs to be retained.

The MATERIALIZE node is not used by default starting from SQL/MX Release 2.x. If needed, enable it by setting the MATERIALIZE default to ON. Starting from SQL/MX Release 2.x, SQL/MX uses the ORDERED_HASH_JOIN Operator to replace the functionality of the MATERIALIZE node.

The MATERIALIZE operator has one child node. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
operation_type	Hash table.	text
values_given_to_child	Values whose change causes materialization of the table.	expr(text)
temp_table_key	Key for the temporary table.	expr(text)
begin_key	Begin key predicate.	expr(text)
end_key	End key predicate.	expr(text)
scan_direction	Direction in which table is scanned: forward or reverse.	text
check_input_values	Expression used to check if input values have changed.	expr(text)

The following is an example of the MATERIALIZE operator:

```
control query default materialize 'on';

control query shape
sort_groupby(nested_join(sort(partition_access(
scan(path 'TAB1', forward, mdam off))),materialize(

partition_access(scan(path 'TAB2', forward, mdam off)))));

prepare TestQuery30 from
select TAB1.col1, TAB1.col2, sum(TAB2.col2), count(*)
from TAB1 , TAB2
where TAB1.col1 = TAB2.col1
and TAB2.col2 < 30
group by TAB1.col1, TAB1.col2
order by TAB1.col2;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  operation_type ......... hash
  scan_direction ......... forward
  values_given_to_child    execution_count
  temp_table_key ......... DETCAT.DETSCH.TAB2.COL1
  begin_key .............. (DETCAT.DETSCH.TAB2.COL1 =
                           DETCAT.DETSCH.TAB1.COL1)
  check_input_values ..... (execution_count =
                             convert(execution_count))
```

MERGE_ANTI_SEMI_JOIN Operator

Join Group

The MERGE_ANTI_SEMI_JOIN operator returns rows only when no match occurs in the inner table. The operator discards all rows that have a match. Also see MERGE_JOIN Operator on page 7-42 and MERGE_SEMI_JOIN Operator on page 7-44.

The MERGE_ANTI_SEMI_JOIN operator has two child nodes. The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer

fragment_type	Master, ESP, or DP2.	text
merge_join_predicate	Expression of the join predicate.	expr(text)
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: merge	text
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicate	Expression of the HAVING clause.	expr(text)

The following is an example of MERGE_ANTI_SEMI_JOIN:

```
Control Query Default nested_joins 'off';
Control Query Default hash_joins 'off';

PREPARE TestQuery3a FROM
SELECT s_nationkey, s_suppkey
FROM supplier
WHERE s_suppkey NOT IN
( SELECT  ps_suppkey from partsupp)
GROUP BY s_nationkey, s_suppkey
ORDER BY s_nationkey, s_suppkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner anti-semi
  join_method ............ merge
  merge_join_predicate ... (DETCAT.DETSCH.SUPPLIER.S_SUPPKEY =
                             DETCAT.DETSCH.PSX1.PS_SUPPKEY)
```

MERGE_JOIN Operator

Join Group

The MERGE_JOIN operator describes a portion of an execution plan that involves a merge join. This operator joins the data from its two child nodes. The data streams from both children must be in the same order. The operator joins all matching rows from each data stream. The MERGE_JOIN operator works only with equijoins.

The MERGE_JOIN operator has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: merge	text
merge_join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the HAVING clause.	expr(text)

The following is an example of the MERGE_JOIN operator:

```
PREPARE TestQuery10 FROM
SELECT l_orderkey,
CAST(SUM(l_extendedprice*(1-l_discount))AS
NUMERIC(18,2)), o_orderdate, o_shippriority
FROM customer,orders,lineitem
WHERE c_mktsegment = 'BUILDING'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < DATE '1995-03-15'
AND l_shipdate > DATE '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY 2 DESC,3 ASC;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method ............ merge
  merge_join_predicate ... (DETCAT.DETSCH.SUPPLIER.S_SUPPKEY =
                           DETCAT.DETSCH.PSX1.PS_SUPPKEY)
  selection_predicates ... (DETCAT.DETSCH.PSX1.PS_SUPPLYCOST <
                             DETCAT.DETSCH.SUPPLIER.S_ACCTBAL)
```

MERGE_SEMI_JOIN Operator

Join Group

The MERGE_SEMI_JOIN operator returns one row for the first match it finds in the inner table. Conversely, MERGE_JOIN returns a row for all matches in the inner table. See MERGE_JOIN Operator on page 7-42.

The MERGE_SEMI_JOIN operator has two child nodes. The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
merge_join_predicate	Expression of the join predicate.	expr(text)
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: merge	text
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicates	Expression of the WHERE clause that is not included in the merge_join_predicate or in a selection_predicate of any children.	expr(text)

The following is an example of the MERGE_SEMI_JOIN operator:

```
control query default hash_joins 'off';
control query default nested_joins 'off';

PREPARE TestQuery7 FROM
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_orderdate >= DATE '1993-07-01'
AND o_orderdate < DATE '1993-10-01'
AND EXISTS (SELECT *
FROM lineitem
WHERE l_orderkey = o_orderkey

AND l_commitdate < l_receiptdate)
GROUP BY o_orderpriority
ORDER BY o_orderpriority;

DESCRIPTION
```

```
fragment_id ............ 0
parent_frag ............ (none)
fragment_type .......... master
join_type .............. inner semi
join_method ............ merge
merge_join_predicate ... (DETCAT.DETSCH.ORDERS.O_ORDERKEY =
                           DETCAT.DETSCH.LINEITEM.L_ORDERKEY)
```

MERGE_UNION Operator

MERGE_UNION Group

The MERGE_UNION operator describes that part of an execution plan that merges rows from two child nodes. The operator for a MERGE_UNION operator is always MERGE_UNION.

The MERGE_UNION operator has two child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
condExpr	Expression used for conditional union. Occurs with IF statement in compound statements.	expr(text)
merge_expression	Expression used to determine which child operator to read from next—read from left if true and read from right if false.	expr(text)
union_type	Merge, physical or unspecified.	text
sort_order	Sort order of the result of the union.	text
trigExceptExpr	Expression used for trigger exceptions.	expr(text)

The following is an example of the MERGE_UNION operator:

```
create table table_a
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, CONSTRAINT table_a_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

create table table_c like table_a;
```

```
prepare xx from
select * from table_a
union
select * from table_c;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  union_type ..............  merge
```

MultiUnion Operator

N-ary Group

The MultiUnion operator provides performance enhancement for queries that have a large number of table unions. It denotes a single relational union operator with multiple children. Its children can be any relational operators that are attached to the union backbone.

Note. The MultiUnion operator is supported only on systems running J06.08 and later J-series RVUs and H06.19 and later H-series RVUs.

The description field for this operator contains:

Token	Followed by ...	Data Type
union_type	PhyMultiUnion	text

The following is an example of the MultiUnion operator:

```
create table tabl3 (a int,  b int, c int,d int, e int);

prepare xx from select * from tabl3 union all
select * from tabl3  union all
select * from tabl3  union all
select * from tabl3  union all
select * from tabl3;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  union_type ............. PhyMultiUnion
```

NESTED_ANTI_SEMI_JOIN Operator

Join Group

The NESTED_ANTI_SEMI_JOIN operator describes a portion of an execution plan that involves a nested join. This operator returns all rows from the inner table that do not satisfy the predicates. See NESTED_JOIN Operator on page 7-47.

The NESTED_ANTI_SEMI_JOIN operator has two child nodes. The description field for this operator contains:.

Token	**Followed by...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: nested or in-order nested	text
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
join_predicate	Expression of the ON clause that has not been pushed down to the inner scan, typically empty.	expr(text)
selection_predicate	Expression of the WHERE clause that is not included in the merge_join_predicate or in a selection_predicate of any children.	expr(text)

The following is an example of the NESTED_ANTI_SEMI_JOIN operator:

```
PREPARE TestQuery12 FROM
SELECT *
FROM customer, nation
WHERE c_custkey > 10000 AND c_custkey < 10010
AND c_nationkey NOT IN
(select n_nationkey from nation where n_regionkey = 10)
ORDER BY c_custkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner anti-semi
  join_method .............. nested
```

NESTED_JOIN Operator

Join Group

The NESTED_JOIN operator describes a portion of an execution plan that involves a nested join. This operator sends each outer row to the inner child, where it eventually goes to a scan operation. Normally, the inner scan access is keyed, and the number of outer probes is small, resulting in an efficient join. The actual join is done in the inner scan instead of the NESTED_JOIN operator. Nested joins support range operations (>=, >, <, <=) in addition to equijoins.

The NESTED_JOIN has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: nested or in-order nested.	text
join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
selection_predicate	Expression of the WHERE clause.	expr(text)

The following is an example of the NESTED_JOIN operator:

```
PREPARE TestQuery4 FROM
SELECT l_orderkey,
CAST(SUM(l_extendedprice*(1-l_discount))AS
NUMERIC(18,2)), o_orderdate, o_shippriority
FROM customer,orders,lineitem
WHERE c_mktsegment = 'BUILDING'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < DATE '1995-03-15'
AND l_shipdate > DATE '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
HAVING sum(l_extendedprice)> 100
ORDER BY 2 DESC,3 ASC;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method .............. nested
```

NESTED_SEMI_JOIN Operator

Join Group

The NESTED_SEMI_JOIN operator returns only one matched row from the inner table and ignores duplicate matches. See NESTED_JOIN Operator on page 7-47.

The NESTED_SEMI_JOIN operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: nested or in-order nested	text
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
join_predicate	Expression of the ON clause that has not been pushed down to the inner scan, typically empty.	expr(text)
selection_predicate	Expression of the WHERE clause that is not included in the merge_join_predicate or in a selection_predicate of any children.	expr(text)

The following is an example of the NESTED_SEMI_JOIN operator:

```
Control Query Default merge_joins 'off';
Control Query Default hash_joins 'off';

PREPARE TestQuery7a FROM
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_orderdate >= DATE '1993-07-01'
AND o_orderdate < DATE '1993-10-01'
AND EXISTS (SELECT *
FROM lineitem
WHERE l_orderkey = o_orderkey
AND l_commitdate < l_receiptdate)
GROUP BY o_orderpriority
ORDER BY o_orderpriority;

DESCRIPTION
```

```
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner semi
  join_method ............. nested
```

NEXTVALUEFOR Operator

The NEXTVALUEFOR operator obtains the next values from the SEQUENCEGENERATOR operator, and then assigns the value to the IDENTITY column in the row.

The description field for the operator contains the following:

Token	**Followed by ...**	**Data Type**
max_card_est	Cardinality estimate for the operator.	integer
fragment_id	A sequential number assigned to the fragment. 0 is the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP.	text
generated_next_value	The expression calculates the next value for the IDENTITY column. The expression calculates the next value using `current_value_from_sg`, `sg_cache_size_hv`, and `sg_increment_hv` sent by SEQUENCEGENERATOR operator.	expr(text)
expected_outputs_from_s	The expression specifies the outputs expected from the child SEQUENCEGENERATOR operator.	expr(text)

NESTED_SEMI_JOIN Operator

ORDERED_HASH_ANTI_SEMI_JOIN

Join Group

The ORDERED_HASH_ANTI_SEMI_JOIN operator returns only one row for every outer row when no match occurs. This operator preserves the order of the outer table and does not overflow to disk. The reuse feature allows reuse of the hash table for subsequent requests within the same query. Choose this operator when you need to preserve the order of the outer table or if you can benefit from the reuse feature. It should be chosen only if the inner table is small enough to fit in memory.

The ORDERED_HASH_ANTI_SEMI_JOIN operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
hash_join_predicate	Expression of the join predicate.	expr(text)
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
reuse_comparison_values	List of values that cause the hash table to be rebuilt when values change.	expr(text)
selection_predicates	Expression of the WHERE clause.	expr(text)

The following is an example of the ORDERED_HASH_ANTI_SEMI_JOIN operator:

```
PREPARE TestQuery3 FROM
SELECT s_nationkey, s_suppkey
FROM supplier
WHERE s_suppkey NOT IN
( SELECT  ps_suppkey from partsupp)
GROUP BY s_nationkey, s_suppkey
ORDER BY s_nationkey, s_suppkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner anti-semi
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.SX1.S_SUPPKEY =
                             DETCAT.DETSCH.PSX1.PS_SUPPKEY)
```

ORDERED_HASH_JOIN Operator

Join Group

The ORDERED_HASH_JOIN operator joins the data from two child tables. This operator preserves the order of the outer table and does not overflow to disk. It creates a hash table from the inner table, joins the outer table by hashing each outer row, and looks for matches in the hash table. The reuse feature enables reuse of the hash table for subsequent requests within the same query. Choose this operator when you need to preserve the order of the outer table or if you can benefit from the reuse feature. It should be chosen only if the inner table is small enough to fit in memory.

Equijoins and cross-products are supported by this operator.

The ORDERED_HASH_JOIN operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
hash_join_predicate	Expression of the join predicate.	expr(text)
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
join_predicate	Expression of the join predicate.	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
reuse_comparison_values	List of values that cause the hash table to be rebuilt when values change.	expr(text)
selection_predicates	Expression of the WHERE clause.	expr(text)

The difference between join_predicate and hash_join_predicate tokens is that the former are the non-equijoin predicates, while the latter are equijoin predicates that are used to help build and probe the hash table.

The following is an example of the ORDERED_HASH_JOIN operator:

```
PREPARE TestQuery9 FROM
SELECT s_nationkey, s_suppkey
FROM supplier, partsupp
```

```
WHERE s_suppkey = ps_suppkey
GROUP BY s_nationkey, s_suppkey
ORDER BY s_nationkey, s_suppkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.SX1.S_SUPPKEY =
                             DETCAT.DETSCH.PSX1.PS_SUPPKEY)
```

ORDERED_HASH_SEMI_JOIN Operator

Join Group

The ORDERED_HASH_SEMI_JOIN operator returns the outer rows for all matches. This operator differs from the HYBRID_HASH_SEMI_JOIN operator in that it preserves the order of the outer table and does not overflow to disk. The reuse feature enables reuse of the hash table for subsequent requests within the same query. Choose this operator when you need to preserve the order of the outer table or if you can benefit from the reuse feature. It should be chosen only if the inner table is small enough to fit in memory.

Also see HYBRID_HASH_JOIN Operator on page 7-27.

The ORDERED_HASH_SEMI_JOIN operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
hash_join_predicate	Expression of the join predicate.	expr(text)
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method: hash	text
join_predicate	Expression of the join predicate.	expr(text)

parallel_join_type	Type1 or Type2, depending on parallel join algorithm.	text
reuse_comparison_values	List of values that cause the hash table to be rebuilt when they change.	expr(text)
selection_predicates	Expression of the WHERE clause.	expr(text)

The difference between join_predicate and hash_join_predicate tokens is that the former are the non-equijoin predicates, while the latter are equijoin predicates that you use to help build and probe the hash table.

The following is an example of the ORDERED_HASH_SEMI_JOIN operator:

```
PREPARE TestQuery7 FROM
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_orderdate >= DATE '1993-07-01'
AND o_orderdate < DATE '1993-10-01'
AND EXISTS (SELECT *
FROM lineitem
WHERE l_orderkey = o_orderkey
AND l_commitdate < l_receiptdate)
GROUP BY o_orderpriority
ORDER BY o_orderpriority;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  join_type .............. inner semi
  join_method ............ hash
  hash_join_predicate .... (DETCAT.DETSCH.ORDERS.O_ORDERKEY =
                             DETCAT.DETSCH.LX5.L_ORDERKEY)
```

ORDERED_UNION Operator

The ORDERED_UNION operator ensures that its left and right children work one at a time. It first receives rows from its left child. It then enables the right child to work. In effect, for the same request, ORDERED_UNION operator produces rows from left child followed by rows from right child.

The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer

fragment_type	Master, ESP, or DP2.	text
sort_order	Sort order of the result of the union.	text
merge_expression	Expression used to determine which child operator to read from next—read from left if true and read from right if false.	expr(text)
union_type	Merge, physical or unspecified.	text
condExpr	Expression used for conditional union. Occurs with IF statement in compound statements.	expr(text)
trigExceptExpr	Expression used for trigger exceptions.	expr(text)

The following is an example of the ORDERED_UNION operator:

```
create table table_a
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, CONSTRAINT table_a_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

create table table_b
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, owner_count int
, CONSTRAINT table_b_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

ALTER TABLE table_a
ADD CONSTRAINT table_a_KEY FOREIGN KEY
(col1, col2) REFERENCES
table_b(col1, col2) on update restrict on delete restrict
DROPPABLE ;

CREATE TRIGGER table_a_Owner_Count
 AFTER INSERT ON table_a
 REFERENCING NEW AS newrow
 FOR EACH ROW
 UPDATE table_b SET owner_count = (
  SELECT count(*) FROM table_a
    WHERE (table_b.col1,table_b.col2)=
         (table_a.col1,table_a.col2)
    AND table_a.col3 = 1
 )
    WHERE (table_b.col1,table_b.col2)=
    (newrow.col1,newrow.col2);

insert into table_b values('A', 1, 1, 0);
```

```
Prepare TestQuery11 from
insert into table_a values('A', 1, 1);
```

```
DESCRIPTION
   fragment_id ............ 0
   parent_frag ............ (none)
   fragment_type .......... master
   union_type ............... merge
```

PACK Operator

Rowset Group

Use the PACK operator in a query plan when selecting rows into rowset arrays:

```
SELECT <list> INTO <list of arrays> <body of query>
```

The PACK operator collects all the rows coming from the body of the query and puts them into the arrays in the `<list of arrays>`. For more information about rowsets and arrays, see the *SQL/MX Programming Manual for C and COBOL.*

The PACK operator has one child. The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
pack_expr	Expression used to pack values of a row into a packed row.	expr(text)

Create a module file. For details on creating module file, see the *SQL/MX Programming Manual for C and COBOL*.

Now execute the following command:

```
explain <name of a statement in module file> from <name of
module file>
```

The following is an example of the PACK operator:

```
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  pack_expr .............. (cast(JAVCAT.JAVSCH.T.T1 AS INTEGER
                           SIGNED)RowsetArrayInto 200 ),
                           (cast(JAVCAT.JAVSCH.T.T2 AS
                           INTEGER SIGNED) RowsetArrayInto 200),
(cast(JAVCAT.JAVSCH.T.T3 AS INTEGER
  SIGNED)RowsetArrayInto 200 )
```

PARTITION_ACCESS Operator

Exchange Group

Use the PARTITION_ACCESS operator to describe a portion of an execution plan for a file system interface in which requests are made to DAM. The DAM process runs in parallel to the PARTITION_ACCESS (no waited interface). For more information about exchange operators, see Section 8, Parallelism.

The PARTITION_ACCESS operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
buffer_size	Buffer size for messages between PARTITION_ACCESS operator and DAM.	integer
record_length	Length of the record that is returned by DAM.	integer
begin_key_preds (incl | excl)	Predicates to determine the begin key, which might include or exclude the specified key.	expr(text)
end_key_preds (incl | excl)	Predicates to determine the end key, which might include or exclude the specified key.	expr(text)
begin_key_exclusion_expr	Boolean expression indicating whether the begin key is excluded from the key range (in cases where this is determined dynamically).	expr(text)
end_key_exclusion_expr	Boolean expression indicating whether the end key is excluded from the key range (in cases where this is determined dynamically).	expr(text)

Token	Followed by ...	Data Type
begin_part_no_expr	Expression to calculate the start partition number (appears instead of begin_key_preds and begin_key_exclusion_expr).	expr(text)
end_part_no_expr	Expression to calculate the end partition number (appears instead of end_key_preds and end_key_exclusion_expr).	expr(text)
olt_optimization	Indicates whether an optimization for short, simple operations is used. Its value is `used` or `not used`.	text
space_usage	One of the following: ● Size of packed plan fragment in 1 KB units. ● Size of unpacked plan fragment in 1 KB units. Heap (used for dynamically allocating entities such as queue entries and diagnostic areas) size in 1 KB units. ● Total size in 1 KB units.	integer
parent_partitioning_function	Type of top partitioning and contains summary information about the parallel plan.	text
child_partitioning_function	Type of bottom partitioning and contains summary information about the parallel plan.	text
begin_part_selection_expr	Expression used by SPLIT_TOP or PARTITION_ACCESS to compute the begin_part_no_expr.	expr(text)
end_part_selection_expr	Expression used by SPLIT_TOP or PARTITION_ACCESS to text compute the end_part_no_expr.	expr(text)

The following is an example of the PARTITION_ACCESS operator:

```
prepare TestQuery17 from
select T03.char_100
  from PTAB03 T03
  where T03.char_100 =
    (Select min(T00.char_10)
       from PTAB00 T00
    );

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
```

```
fragment_type .......... dp2
buffer_size ............ 31,000
record_length .......... 11
space_usage ............ 19:8:8:40
                         eid_space_computation on
                         begin_key_preds_(incl)
                         (DETCAT.DETSCH.PTAB00.SINT32_UNIQ =
                         <min>)
end_key_preds_(incl) ... (DETCAT.DETSCH.PTAB00.SINT32_UNIQ =
                         <max>)
begin_part_no_expr ..... \:_sys_hostVarPAPartNo_1606919584
end_part_no_expr ......... \:_sys_hostVarPAPartNo_1606919584
```

PROBE_CACHE Operator

This operator appears above the right child of a nested join and is an optimization to reduce the work done by the right child. The operator has a mechanism to cache results of a probe (request) into the inner child from the outer child. If the probe repeats, the cached result is used to improve the performance.

The DESCRIPTION column of the explain function will contain token-value pairs to name the probe columns sent to the inner table, the sizes of the cache and inner table buffer (called `num_inner_tuples`).

ROOT Operator

Root Group

The ROOT operator is the root, or the top node, of an execution plan and describes the SQL query. The operator for a ROOT is always ROOT.

The ROOT operator has one child. The description field for this operator contains:

Token	Followed by ...	Data Type
statement	Original SQL statement	text
select_list	Expression of the SELECT list columns	expr(text)
input_variables	Expression containing list of input variables (params)	expr(text)
order_by	Expression containing list of sort keys	expr(text)
must_match	When CONTROL QUERY SHAPE is used, textual description of forced query tree ()	expr(text)
update_col	The update column specification of an updatable cursor declaration	expr(text)

olt_optimization	Indicates whether an optimization for short, simple operations is used. The value *used* is returned if this optimization is used.	text
statement_index	Statement index of this statement as reported by the Measure product.	integer
upd_action_on_error	Determines the type of statement atomicity chosen for a query: XN_ROLLBACK: Transaction is rolled back if an error occurs. RETURN: Query stops executing and error is returned without need for statement rollback. SAVEPOINT: DAM savepoints are used to roll back the statement if an error occurs. PARTIAL_UPD: [NonStop SQL/MP style] Partial results are updated and an error is returned.	text

The following is an example of the ROOT operator:

```
PREPARE TestQuery15 FROM
   SELECT CAST(SUM(l_extendedprice*l_discount) AS
   NUMERIC(18,2))
   AS revenue FROM lineitem WHERE l_shipdate >= DATE '1994-0101'
   AND l_shipdate < DATE '1994-01-01' + INTERVAL '1' YEAR AND
 l_discount BETWEEN .06 - 0.01 AND .06 + 0.01
   AND l_quantity < 24;
```

```
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  statement_index ........ 0
  xn_access_mode ......... read_only
  plan_version ........... 2,200
  SCHEMA ................. DETCAT.DETSCH
  HASH_JOINS ............. ON
  NESTED_JOINS ........... ON
  select_list ............
            cast(cast((sum((cast(DETCAT.DETSCH.LX3.L_EXTENDEDPRI
                             CE AS BIG NUM(12,2) SIGNED) *
                             cast(DETCAT.DETSCH.LX3.L_DISCOUNT
                             AS BIG NUM(12,2) SIGNED))) /
                             cast(100 AS BIG NUM(3) SIGNED)) AS
                             NUMERIC(18,2) SIGNED) AS
                               NUMERIC(18,2) SIGNED)
```

SAMPLE Operator

Data Mining Group

The SAMPLE operator occurs as a result of a sample clause in a query.

The SAMPLE operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
sampled_columns	List of column references representing the outputs of the sample operator. Indicates that the column has been sampled.	expr(text)
balance_expression	Expression representing the sampling expression. A simple random selection, but could be more complex if the sample clause contains a balance clause.	expr(text)
sample_type	Indicates the type of sampling being performed. Possible values are RANDOM, PERIODIC, FIRST, and CLUSTER.	text
required_order	Specified order keys for a sample operation.	expr(text)

For more information about data mining, see the *SQL/MX Data Mining Guide*.

The following is an example of the SAMPLE operator:

```
prepare TestQuery34 from
  select [first 180] * from  t064t5
  sample periodic 1 rows every 100 rows
  for read uncommitted access;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  sample_type ............ PERIODIC
  sampled_columns ........ NotCovered(DETCAT.DETSCH.T064T5.A),
                           NotCovered(DETCAT.DETSCH.T064T5.B)
  balance_expression ..... ((0  block (0  assign (cast(0  AS
                           NUMERIC(11) SIGNED) + cast(1  AS
                           NUMERIC(11) SIGNED)))) block
case(if_then_else((0  <= 0), 0,
if_then_else((0  < 1), 1,
if_then_else((1 > 0), ((0  assign
  (1 - 100)) block 1 ), 0)))))
```

SAMPLE_FILE_SCAN Operator

The SAMPLE_FILE_SCAN operator occurs as a result of a sample clause in a query, where it is possible to read randomly after satisfying the conditions mentioned in the query. This operator has no child node. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short, simple operations is used. Its value is *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
scan_type	FILE_SCAN followed by table name.	text
scan_direction	Direction in which table is scanned: forward or reverse.	text
lock_mode	The lock mode specified: shared, exclusive, not specified (defaulted to lock cursor), or unknown.	text
access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
columns_retrieved	Estimation of the number of columns to be returned.	integer
fast_scan	Indication of whether an optimization for a simple scan operation is used. The value *used* is returned if this optimization is used.	text
fast_replydata_move	Indication of whether an optimization for returning data from DAM is used. The value *used* is returned if this optimization is used.	text

key_columns	Columns used as the primary key.	expr(text)
executor_predicates	Any predicate expression that is not a key predicate evaluated by the executor in DAM.	expr(text)
mdam_disjunct	Disjunct key predicates used by MDAM	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
begin_key	Expression of the begin key predicate.	expr(text)
end_key	Expression of the end key predicate.	expr(text)
key_type	Simple or MDAM.	text

The following is an example of the SAMPLE_FILE_SCAN operator:

```
PREPARE TestQuery22 FROM
select * from tt22 sample random 10 percent;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  scan_type .............. sample full scan of table
                           DETCAT.DETSCH.TT22
  scan_direction ......... forward
  key_type ............... simple
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 2
  fast_replydata_move .... used
  key_columns ............ F1
  begin_key .............. (F1 = <min>)
  end_key .................. (F1 = <max>)
```

SEQUENCE Operator

Data Mining Group

The SEQUENCE operator occurs as a result of a SEQUENCE BY clause in the query.

The SEQUENCE operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
required_order	List of column references specifying the ordering required by the SEQUENCE operator. Obtained from a list of columns specified in the SEQUENCE BY clause.	expr(text)
sequence_functions	Represents the list of sequence functions that must be evaluated by this SEQUENCE operator.	ItemExpr tree
num_history_rows	Size of the history buffer (in rows). This number of rows is kept in an integer buffer and is available for access by the sequence functions. Any access to a row outside this buffer results in a NULL value. The default value for this parameter is 1024 rows.	integer
history_row_size	Size of each history row in the history buffer.	integer

For more information about data mining, see the *SQL/MX Data Mining Guide*.

The following is an example of the SEQUENCE operator:

```
prepare TestQuery24 from
SELECT RUNNINGCOUNT(*) FROM customer
SEQUENCE BY c_custkey, c_nationkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  num_history_rows ....... 1,024
  history_row_size ....... 16
  required_order ......... DETCAT.DETSCH.CX1.C_CUSTKEY,
                           DETCAT.DETSCH.CX1.C_NATIONKEY
  sequence_functions ..... (replace null(offset(\:_sys_Result),
                           cast(offset(\:_sys_Result) AS
                           LARGEINT), cast(0  AS LARGEINT)) +
                           replace null(1 , cast(1  AS INTEGER
                           SIGNED), cast(0  AS INTEGER
                             SIGNED)))
```

SEQUENCEGENERATOR Operator

The SEQUENCEGENERATOR operator provides access to the sequence generator table to get the next value or the next block of values. It temporarily suspends the current user transaction, and starts and commits a new transaction to access the sequence generator table.

The SEQUENCEGENERATOR operator is isolated into its own ESP by the ESP_ACCESS operator.

The description field for the operator contains:

Token	**Followed by ...**	**Data Type**
max_card_est	Cardinality estimate for this operator.	integer
fragment_id	A sequential number assigned to the fragment. 0 is the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_flag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	ESP.	text
sg_start_with_option	The sequence generator option that specifies the start value of the cycle.	int64
sg_increment_option	The INCREMENT-BY option of the sequence generator. The next value is obtained by adding this value to current value.	int64
sg_maximum_option	The sequence generator option that specifies the highest value in the cycle.	int64
sg_minimum_option	The sequence generator option that specifies the lowest value in the cycle.	int64
sg_datatype	The sequence generator data type.	int64
sg_cycle_option	The sequence generator option that specifies whether the sequence generator values can be cycled. The option values are CYCLE or NO CYCLE.	text
sg_cache	The user defined sequence generator cache value. The default is 0. If the value is greater than 0, each request gets the user specified number of values for the cache.	int64

sg_cache_initial	The initial cache value that influences the default adaptive cache. The cache value dynamically changes at runtime based on the following CQD settings: SEQUENCE_GENERATOR_CACHE_INITIAL value is multiplied by the SEQUENCE_GENERATOR_CACHE_INCREMENT value, and is compared with SEQUENCE_GENERATOR_CACHE_MAXIMUM.	int64
sg_cache_increment	The multiplier that influences the default adaptive cache. For details, see the explanation for `sg_cache_initial` token.	int64
sg_cache_maximum	The maximum value that influences the default adaptive cache. For details, see the explanation for `sg_cache_initial` token.	int 64
sg_increment_hv	The expression contains the `sg_increment_option` value. It is sent to the NEXTVALUEFOR operator along with `sg_cache_size_hv` and `current_value_from_sg` to calculate the next value.	expr(text)
sg_cache_size_hv	The expression is used to calculate the sequence generator cache size.	expr(text)
current_value_from_sg	The expression holds the current value of the sequence generator. Current value is the next value available from the sequence generator.	expr(text)

SHORTCUT_SCALAR_AGGR Operator

Groupby Group

The SHORTCUT_SCALAR_AGGR operator occurs for aggregates without a GROUP BY clause and returns one row. The description field for this operator contains:

Token	**Followed by...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer

fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicate	Expression of the WHERE clause.	expr(text)

The following is an example of the SHORTCUT_SCALAR_AGGR operator:

```
prepare TestQuery25 from
select min(f2) from tt22;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  aggregates ............... min(DETCAT.DETSCH.II22.F2)
```

SORT Operator

Sort Group

A SORT operator describes a portion of an execution plan that performs a sort. The operator for a SORT operator is always SORT.

The SORT operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
sort_key	Expression describing sort keys.	expr(text)

The following is an example of the SORT operator:

```
PREPARE TestQuery16 FROM
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address,
s_phone, s_comment
FROM part,supplier,partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (SELECT MIN(ps_supplycost)
FROM partsupp ps1,supplier s1, nation n1,region r1
WHERE p_partkey = ps1.ps_partkey
AND s1.s_suppkey = ps1.ps_suppkey
AND s1.s_nationkey = n1.n_nationkey
AND n1.n_regionkey = r1.r_regionkey
AND r1.r_name = 'EUROPE')
ORDER BY s_acctbal desc, n_name, s_name, p_partkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type ........... master self_referencing_update
                           forced_sort
  sort_key ...............
                      inverse(DETCAT.DETSCH.SUPPLIER.S_ACCTBAL),
                      DETCAT.DETSCH.NATION.N_NAME,
                      DETCAT.DETSCH.SUPPLIER.S_NAME,
                        DETCAT.DETSCH.PSX1.PS_PARTKEY
```

SORT_GROUPBY Operator

Groupby Group

The SORT_GROUPBY operator describes a portion of an execution plan that affects a group.

The SORT_GROUPBY operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicate	Expression of the HAVING clause.	expr(text)
grouping_columns	Expression of the grouping columns.	expr(text)

The following is an example of the SORT_GROUPBY operator:

```
prepare TestQuery33a from
select int64_6,count(*)
  from PTAB09
  where        int64_6 IN (0,2,4,5)
  group by int64_6;

DESCRIPTION
```

```
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  grouping_columns ....... DETCAT.DETSCH.PTAB09.INT64_6
  aggregates ............... count(1 )
```

SORT_PARTIAL_AGGR_LEAF Operator

Groupby Group

The SORT_PARTIAL_AGGR_LEAF operator executes a partial group by operation as close to where the data is read as is cost effective. This strategy reduces the amount of data that must be redistributed for a query. The operator must always be accompanied by a SORT_PARTIAL_AGGR_ROOT operator above it in the tree, which finalizes the query.

The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)

The following is an example of the SORT_PARTIAL_AGGR_LEAF operator:

```
prepare TestQuery17 from
select T03.char_100
  from PTAB03 T03
  where T03.char_100 =
    (Select min(T00.char_10)
       from PTAB00 T00);

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  aggregates ............... min(DETCAT.DETSCH.PTAB00.CHAR_10)
```

SORT_PARTIAL_AGGR_ROOT Operator

Groupby Group

The SORT_PARTIAL_AGGR_ROOT operator works together as a pair with the SORT_PARTIAL_AGGR_LEAF operator. The SORT_PARTIAL_AGGR_ROOT operator finalizes the group by at the ESP level. This operator consists of a one-row aggregate without standard aggregate functions (SUM, MIN, MAX, and so on). The root portion occurs in the root. The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)

The following is an example of the SORT_PARTIAL_AGGR_ROOT operator:

```
prepare TestQuery17 from
select T03.char_100
  from PTAB03 T03
  where T03.char_100 =
    (Select min(T00.char_10)
       from PTAB00 T00);

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  aggregates .............
                            min(min(DETCAT.DETSCH.PTAB00.CHAR_10))
```

SORT_PARTIAL_GROUPBY_LEAF Operator

Groupby Group

The SORT_PARTIAL_GROUPBY_LEAF operator executes a partial group by as close to where the data is read as is cost effective. This strategy reduces the amount of data that must be redistributed for a query. The operator must always be accompanied by a SORT_PARTIAL_GROUPBY_ROOT operator above it in the tree, which finalizes the query.

The SORT_PARTIAL_GROUPBY_LEAF operator has one child node. The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
grouping_columns	Expression of the grouping columns.	expr(text)
aggregates	Expression of the aggregate function.	expr(text)
selection_predicates	Expression of the HAVING clause.	expr(text)

The following is an example of the SORT_PARTIAL_GROUPBY_LEAF operator:

```
control query shape
sort_groupby(partition_access(sort_groupby(scan)));

prepare TestQuery33 from
select int64_6,count(*)
  from PTAB09
  where int64_6 IN (0,2,4,5)
  group by int64_6;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  grouping_columns ....... DETCAT.DETSCH.PTAB09.INT64_6
  aggregates .............  count(1 )
```

SORT_PARTIAL_GROUPBY_ROOT Operator

Groupby Group

The SORT_PARTIAL_GROUPBY_ROOT operator works together as a pair with the SORT_PARTIAL_GROUPBY_LEAF operator. The SORT_PARTIAL_GROUPBY_ROOT operator finalizes the group by at the ESP level.

The SORT_PARTIAL_GROUPBY_ROOT operator has one child node. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicate	Expression of the HAVING clause.	expr(text)
grouping_columns	Expression of the grouping columns.	expr(text)

The following is an example of the SORT_PARTIAL_GROUPBY_ROOT operator:

```
control query shape
sort_groupby(partition_access(sort_groupby(scan)));

prepare TestQuery33 from
select int64_6,count(*)
  from PTAB09
  where int64_6 IN (0,2,4,5)
  group by int64_6;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  grouping_columns ....... DETCAT.DETSCH.PTAB09.INT64_6
  aggregates ..............  sum(count(1 ))
```

SORT_SCALAR_AGGR Operator

Groupby Group

The SORT_SCALAR_AGGR operator occurs for aggregates without a GROUP BY clause. It returns one row. The description field for this operator contains:

Token	**Followed by...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
aggregates	Expression of the aggregate function.	expr(text)
selection_predicates	Expression of the HAVING clause.	expr(text)

The following is an example of the SORT_SCALAR_AGGR operator:

```
PREPARE TestQuery18 FROM
SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address,
s_phone, s_comment
FROM part,supplier,partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
AND ps_supplycost = (SELECT MIN(ps_supplycost)
FROM partsupp ps1,supplier s1, nation n1,region r1
WHERE p_partkey = ps1.ps_partkey
AND s1.s_suppkey = ps1.ps_suppkey
AND s1.s_nationkey = n1.n_nationkey
AND n1.n_regionkey = r1.r_regionkey
AND r1.r_name = 'EUROPE')
ORDER BY s_acctbal desc, n_name, s_name, p_partkey;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  aggregates .............
                     min(DETCAT.DETSCH.PARTSUPP.PS_SUPPLYCOST)
  selection_predicates ... (DETCAT.DETSCH.PSX1.PS_SUPPLYCOST =
                      min(DETCAT.DETSCH.PARTSUPP.PS_SUPPLYCOST))
```

SPLIT_TOP Operator

Exchange Group

The SPLIT_TOP operator describes a portion of an execution plan for a file system interface in which requests to DAM occur with some level of parallel processing. The operator for a SPLIT_TOP operator is always SPLIT_TOP. For more information about exchange operators, see Section 8, Parallelism.

The SPLIT_TOP operator has one child node. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
olt_optimization	Indicates whether an optimization for short and simple operations is used. Value is used if this optimization is used. Its value is *`used`* or *`not used`*.	text
parent_processes	Number of ESPs containing this operator. The value will be one if the parent is Master Executor.	integer
child_processes	Number of bottom partitions.	integer
parent_partitioning_function	Type of top partitioning and contains summary information about the parallel plan.	text
child_partitioning_function	Indicates how the input data received by the SPLT_TOP is organized.	text

The following is an example of the SPLIT_TOP operator:

```
PREPARE TestQuery19 FROM
select cust_id, count(*) as order_count from orders
where order_date > (date '2001-01-12') group by cust_id order by
cust_id;

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  olt_optimization ....... not used
  parent_processes ....... 1
 child_processes .......... 3 parent_partitioning_fun  grouped
                            1 to 3,PAPA with 3 PA(s),
                            exactly 1 partition
                            child_partitioning_func range
                            partitioned 3 ways on
                             (T03.SINT32_UNIQ)
```

SUBSET_DELETE Operator

DAM Subset Group

The SUBSET_DELETE operator describes a portion of an execution plan that details how a certain access path is scanned: it deletes more than one row. A subset operation performs the read and delete in a combined operation. This operation differs from CURSOR_DELETE, which performs the read and delete in separate operations. The CURSOR_DELETE operation also involves more messages.

This operator has no child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is (none) for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
iud_type	Type of delete followed by table or index name.	expr(text)
predicate	Expression specified in WHERE clause.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
begin_key	Expression of the begin key predicate.	expr(text)
index_begin_key	Expression of the begin key predicates on index.	expr(text)
end_key	Expression of the end key predicate.	expr(text)
index_end_key	Expression of the end key predicates on index.	expr(text)
check_constraint	Check constraints in the table.	expr(text)
scan_type	Information on indexes.	text
scan_direction	Direction in which table is scanned: forward or reverse.	text

lock_mode	The lock mode specified: shared, exclusive, not specified (defaulted to lock cursor), or unknown.	text
access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
columns_retrieved	Estimate of the number of columns to be returned.	integer

The following is an example of the SUBSET_DELETE operator:

```
PREPARE TestQuery1 FROM
DELETE FROM customer
WHERE c_nationkey<300;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  Scan_Direction ......... forward
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  iud_type ............... subset_delete DETCAT.DETSCH.CUSTOMER
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 8
  predicate .............. (C_NATIONKEY < 300)
  begin_key .............. (C_CUSTKEY = <min>)
  end_key .................. (C_CUSTKEY = <max>)
```

SUBSET_UPDATE Operator

DAM Subset Group

The SUBSET_UPDATE operator describes a portion of an execution plan that details how a certain access path is scanned: it updates more than one row. A subset operation performs the read and update in a combined operation. This operation differs from CURSOR_UPDATE, which performs the read and update in separate operations. The CURSOR_UPDATE operation also involves more messages.

This operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
begin_key	Expression of the begin key predicates on the clustering key columns of the table or index.	expr(text)
index_begin_key	Expression of the begin key predicates on index.	expr(text)
end_key	Expression of the end key predicates on the clustering key columns of the table or index.	expr(text)
index_end_key	Expression of the end key predicates on index.	expr(text)
check_constraint	Check constraints in the update table.	expr(text)
scan_type	Physical scan associated with the scan logical operator.	text
scan_direction	Direction in which table is scanned: forward or reverse.	text
lock_mode	The lock mode specified: shared, exclusive, not specified (defaulted to lock cursor), or unknown.	text
access_mode	The access specified: read uncommitted, skip conflict, read committed, stable, serializable, mx serializable, not specified (defaulted to read committed), or unknown.	text
new_rec_expr	Computation of the row to be updated.	expr(text)
columns_retrieved	Estimate of the number of columns to be returned.	integer
predicate	Predicate used in the UPDATE statement.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
iud_type	Specifies the type of update operator. Could be subset_update, unique_update, or cursor_update.	expr(text)
selection_predicate	Predicate specified in the WHERE clause of a query.	expr(text)

The following is an example of the SUBSET_UPDATE operator:

```
PREPARE TestQuery20 FROM
UPDATE customer SET c_nationkey = c_nationkey + 1
WHERE c_custkey > 300;

DESCRIPTION
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  Scan_Direction ......... forward
  olt_optimization ....... not used
  olt_opt_lean ........... not used
  iud_type ............... subset_update DETCAT.DETSCH.CUSTOMER
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 8
  new_rec_expr ........... (C_NATIONKEY assign (cast(C_NATIONKEY
                           AS NUMERIC(11) SIGNED) + cast(1 AS
                           NUMERIC(11) SIGNED)))
  begin_key .............. (C_CUSTKEY = 300)
  end_key .................. (C_CUSTKEY = <max>)
```

TRANSPOSE Operator

Data Mining Group

The TRANSPOSE operator occurs as a result of a TRANSPOSE clause.

The TRANSPOSE operator has one child. The description field for this operator contains:

Token	Followed by...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
transpose_union_vector	Represents a transpose set of the transpose clause. If multiple transpose sets, then multiple instances of the token.	ItemExpr tree

For more information about data mining, see the *SQL/MX Data Mining Guide*.

The following is an example of the TRANSPOSE operator:

```
prepare TestQuery23 from
insert into T061_T232OR1
(
    select
        cast(c1 || c2 || c3 || c4 || c5 as int),
        cast(c1 || c2 || c3 || c4 || c5 as int),
        cast(c1 || c2 || c3 || c4 || c5 as int),
        cast(c1 || c2 || c3 || c4 || c5 as int)
      from
              (values(1)) t
      transpose '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
      as c1
      transpose '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
      as c2
      transpose '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
      as c3
      transpose '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
      as c4
      transpose '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
      as c5
);

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
                           transpose_union_vector
                           ValueIdUnion('0', '1', '2', '3',
                             '4', '5', '6', '7', '8', '9')
```

TUPLE_FLOW Operator

Join Group

The TUPLE_FLOW operator describes a portion of an execution plan that involves a nested join. This operator enables data to flow from one child to the other.

The TUPLE_FLOW operator has two child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text

Token	Followed by ...	Data Type
join_type	Inner, left, natural, inner semi, or inner anti-semi-join.	text
join_method	Name of join method:nested or in-order nested join.	text
join_predicate	Expression of the join predicate	expr(text)
parallel_join_type	Type1 or Type2, depending on parallel join algorithm	text
selection_predicate	Expression of the WHERE clause	expr(text)

The following is an example of the TUPLE_FLOW operator:

```
prepare TestQuery38 from
UPDATE table_b SET owner_count = (
  SELECT count(*) FROM table_a
    WHERE (table_b.col1,table_b.col2)=
          (table_a.col1,table_a.col2)
    AND table_a.col3 = 1
);

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
                           transpose_union_vector
                           ValueIdUnion('0', '1', '2', '3',
                             '4', '5', '6', '7', '8', '9')
```

TUPLELIST Operator

Tuple Group

The TUPLELIST operator shows the values that you place in the query when the VALUES clause is used. The description field for this operator contains:

Token	Followed by...	Data type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is (none) for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
tuple_expr	The tuple produced by this node.	expr(text)

The following is an example of the TUPLELIST operator:

```
PREPARE TestQuery21 FROM
INSERT INTO nation
VALUES (43, 'botswana', 16, 'african country'),
(44, 'france', 23, 'european country'),
(45, 'nepal', 88, 'asian country');

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  tuple_expr ............. cast(%(45) AS INTEGER SIGNED),
                           cast(%('nepal') AS VARCHAR(25)
                           CHARACTER SET ISO88591), cast(%(88)
                           AS INTEGER SIGNED),
                           cast(%('asian country') AS
                           VARCHAR(152) CHARACTER
                             SET ISO88591)
```

UNARY_UNION Operator

The UNARY_UNION operator first evaluates the request using the condition expression (condExpr) associated to the operator. If the result is true, it passes the request to its child. In this case the UNARY_UNION operator always has one child.

The description field for this operator contains:

Token	Followed by...	Data type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
sort_order	Sort order of the result of the union.	text
merge_expression	Expression used to determine which child operator to read from next—read from left if true and read from right if false.	expr(text)
union_type	Merge, physical or unspecified.	text
condExpr	Expression used for conditional union. Occurs with the IF statement in compound statements.	expr(text)
trigExceptExpr	Expression used for trigger exceptions.	expr(text)

The following is an example of the UNARY_UNION operator:

```
create table table_a
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, CONSTRAINT table_a_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
 ASC) NOT DROPPABLE
);

create table table_b
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, owner_count int
, CONSTRAINT table_b_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

ALTER TABLE table_a
 ADD CONSTRAINT table_a_KEY FOREIGN KEY
 (col1, col2) REFERENCES
 table_b(col1, col2) on update restrict on delete restrict
DROPPABLE ;

CREATE TRIGGER table_a_Owner_Count
 AFTER INSERT ON table_a
 REFERENCING NEW AS newrow
 FOR EACH ROW
 UPDATE table_b SET owner_count = (
  SELECT count(*) FROM table_a
    WHERE (table_b.col1,table_b.col2)=
          (table_a.col1,table_a.col2)
       AND table_a.col3 = 1
    )
    WHERE (table_b.col1,table_b.col2)=
          (newrow.col1,newrow.col2);

insert into table_b values('A', 1, 1, 0);

Prepare TestQuery11 from
insert into table_a values('A', 1, 1);

DESCRIPTION
   fragment_id ............ 0
   parent_frag ............ (none)
   fragment_type .......... master
   union_type ............... merge
```

UNIQUE_DELETE Operator

DAM Unique Group

The UNIQUE_DELETE operator describes a portion of an execution plan that works on one row only. It deletes zero or one row.

The UNIQUE_DELETE operator has no child nodes. The description field for this operator contains:

Token	Followed by ...	Data Type
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
index_key	Expression of the begin key predicates on index.	expr(text)
key	Expression of the key predicate.	expr(text)
part_key_predicate	Predicate expression specified on partitioning key. It is displayed only if partitioning key differs from clustering key.	expr(text)
check_constraint	Check constraints in the delete table.	expr(text)
olt_optimization	Indicates whether an optimization for short, simple operations is used. Its value is *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text
iud_type	Type of delete followed by table or index name.	expr(text)
predicate	Expression specified on WHERE clause that is not reflected in the begin and end predicates.	expr(text)

The following is an example of the UNIQUE_DELETE operator:

```
prepare TestQuery26 from
delete from tt22 where f1=46;

DESCRIPTION
  olt_optimization ....... used
  olt_opt_lean ........... used
  fragment_id ............ 2
```

```
parent_frag ............ 0
fragment_type .......... dp2
Scan_Direction ......... forward
olt_optimization ....... used
olt_opt_lean ........... used
iud_type ............... unique_delete DETCAT.DETSCH.TT22
lock_mode .............. not specified, defaulted to lock
cursor
access_mode ............ not specified, defaulted to read
committed
columns_retrieved ...... 2
key ..................... (F1 = %(46))
```

UNIQUE_UPDATE Operator

DAM Unique Group

The UNIQUE_UPDATE operator describes a portion of an execution plan that works on one row only; it updates zero or one row.

The UNIQUE_UPDATE operator has no child nodes. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
new_rec_expr	Computation of the row to be updated	expr(text)
predicate	Expression specified on WHERE clause that is not reflected in the begin and end predicates.	expr(text)
key	Expression of the key predicate.	expr(text)
index_key	Expression of the key predicates on index.	expr(text)
olt_optimization	Indicates whether an optimization for short, simple operations is used. The value *`used`* or *`not used`*.	text
olt_opt_lean	Indicates for short, simple operations whether a further optimization that reduces the physical size of the plan is used. Its value is *`used`* or *`not used`*.	text

part_key_predicate	Predicate specified on the partitioning key. Displayed only if partitioning key differs from clustering key.	expr(text)
check_constraint	Check constraints in the update table.	expr(text)
iud_type	Type of update followed by table or index name.	expr(text)

The following is an example of the UNIQUE_UPDATE operator:

```
prepare TestQuery27 from
UPDATE tt22
SET f2 = 1
WHERE f1 = 2009;

DESCRIPTION
  olt_optimization ....... used
  fragment_id ............ 2
  parent_frag ............ 0
  fragment_type .......... dp2
  Scan_Direction ......... forward
  olt_optimization ....... used
  olt_opt_lean ........... not used
  iud_type ............... unique_update DETCAT.DETSCH.TT22
  lock_mode .............. not specified, defaulted to lock
                           cursor
  access_mode ............ not specified, defaulted to read
                           committed
  columns_retrieved ...... 2
  new_rec_expr ........... (F2 assign 1)
  key ...................... (F1 = 2009)
```

UNPACK Operator

Rowset Group

Use the UNPACK operator in a query plan when an array is used as input in a query (for example, inserting rows from rowset arrays). The UNPACK operator extracts the elements from the array to use in the query. For more information about rowsets and arrays, see the *SQL/MX Programming Manual for C and COBOL.*

The UNPACK operator has one child. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer
parent_frag	The fragment_id for the parent of the current fragment. The value is `(none)` for the master executor.	integer
fragment_type	Master, ESP, or DP2	text

unpack_expression	Expression used to extract values from a packed row	expr(text)
index_value	System-generated index used when accessing a packed row	expr(text)
packing_factor	Used to extract the packing factor from the packed row. The packing factor is the number of logical rows in the packed row.	integer

Create a module file. For details on creating module file, see the *SQL/MX Programming Manual for C and COBOL*.

Now execute the following command:

```
explain <name of a statement in module file> from <name of
module file>
```

The following is an example of the UNPACK operator:

```
DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
  unpack_expression ...... (:a RowsetArrayScan
                           \:_sys_rowset_index1),
                           (:b
                           RowsetArrayScan\:_sys_rowset_index1),
                           (:c
                           RowsetArrayScan\:_sys_rowset_index1)
  packing_factor ......... 200
  index_value .............. \:_sys_rowset_index1
```

VALUES Operator

Tuple Group

The VALUES operator calculates an expression for each row it receives from its child node and returns that expression to its parent node.

The VALUES operator has one child node. The description field for this operator contains:

Token	**Followed by ...**	**Data Type**
fragment_id	A sequential number assigned to the fragment. 0 is always the master executor and 1 is reserved for the EXPLAIN plan. Numbers 2 to n will be ESP or DAM fragments.	integer

parent_frag	The fragment_id for the parent of the current fragment. The value is (none) for the master executor.	integer
fragment_type	Master, ESP, or DP2.	text
tuple_expr	The tuple produced by this node.	expr(text)

The following is an example of the VALUES operator:

```
create table table_a
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, CONSTRAINT table_a_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

create table table_b
(col1 char(5) NOT NULL NOT DROPPABLE
, col2 int NOT NULL NOT DROPPABLE
, col3 smallint
, owner_count int
, CONSTRAINT table_b_PRIMARY_KEY PRIMARY KEY (col1 ASC, col2
ASC) NOT DROPPABLE
);

ALTER TABLE table_a
 ADD CONSTRAINT table_a_KEY FOREIGN KEY
 (col1, col2) REFERENCES
 table_b(col1, col2) DROPPABLE ;

CREATE TRIGGER table_a_Owner_Count
AFTER INSERT ON table_a
REFERENCING NEW AS newrow
FOR EACH ROW
UPDATE table_b SET owner_count = (
SELECT count(*) FROM table_a
    WHERE (table_b.col1,table_b.col2)=
          (table_a.col1,table_a.col2)
    AND table_a.col3 = 1
    )
    WHERE (table_b.col1,table_b.col2)=
    (newrow.col1,newrow.col2);

insert into table_b values('A', 1, 1, 0);

Prepare TestQuery11 from
insert into table_a values('A', 1, 1);

DESCRIPTION
  fragment_id ............ 0
  parent_frag ............ (none)
  fragment_type .......... master
```

```
tuple_expr ............. NULL
selection_predicates ..... 0.
```

8 Parallelism

Using SQL/MX parallel processing within a large query maximizes the efficiency and performance of such queries. Parallel execution can assume multiple forms. You can partition the data or the query execution (or both) and then process the obtained partitions in parallel. This section describes the types of parallelism supported by NonStop SQL/MX.

The SQL/MX optimizer examines both parallel and nonparallel execution plans and chooses the plan that has the lowest total cost, even if it is a nonparallel plan.

Good candidates for parallelism are operators that return few rows, after processing large amounts of data, against tables that are partitioned. A good candidate for a parallel plan is a query such as `SELECT COUNT(*) FROM T1,T2 WHERE T1.A=T2.B`. In this example, large amounts of data must be retrieved, and an aggregate must be computed. However, only one row is returned at the end of the query.

Types of Parallelism in NonStop SQL/MX

- Partitioned Parallelism
- Pipelined Parallelism
- Independent Parallelism

The SQL/MX architecture provides pipelined and independent parallelism without special processing. An individual query plan produced by NonStop SQL/MX can contain any combination of pipelined, independent, or partitioned parallelism.

Partitioned Parallelism

Partitioned parallelism, the main type of parallelism in NonStop SQL/MX, is the ability to divide the data to be processed into partitions (fractions) and to work on each one in parallel. In a partitioned parallel plan, multiple operators all work on the same plan. Results are merged by using multiple pipelines, enabling NonStop SQL/MX to preserve the sort order of the input partitions. Partitioning is also called *data parallelism* because the data is the unit that gets partitioned into independently executable fractions. Partitioned parallelism can occur in DAM processes (called *DAM parallelism*) and ESP processes (called *ESP parallelism*). See DAM and ESP Parallelism on page 8-5.

Pipelined Parallelism

Pipelined parallelism is an inherent feature of NonStop SQL/MX because of its data flow architecture. This architecture interconnects all operators by pipes, with the output of one operator being piped as input to the next operator, and so on. The result is that each operator works independently of any other operator, producing its output as soon as its input is available. Pipelining is seen in almost all query plans. You cannot force pipelined parallelism. It occurs as a natural by-product of the SQL/MX search engine.

Independent Parallelism

Independent execution is also an inherent feature of NonStop SQL/MX because of the architecture. Independent parallelism means that two or more operators can execute simultaneously. Except for certain synchronization conditions, the operators execute independently; for example, the parts of a UNION query. Usually each of the independent operators is connected by a pipeline to a common ancestor. Independent parallelism could also be called *operator parallelism,* because it is the operators on the data that execute in parallel. Independent parallelism occurs in many plans. Like pipelining, you cannot force independent parallelism.

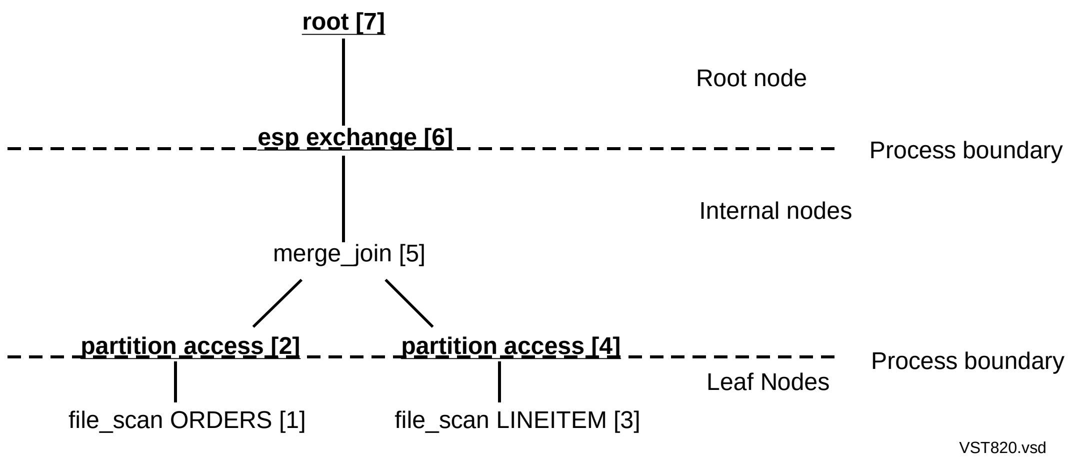
Parallel Execution Principles

Query Plan as a Data Flow Tree

The query plan is represented by a data flow tree with three types of nodes:

Leaf nodes	Each leaf node is a data source. Typically, the leaf node is either a base table scan or an index scan.
Internal nodes	Each internal node corresponds to an operator within the current query. Like the operator itself, an internal node can be unary, binary, or N-ary. The Exchange node is a special internal node that represents a process boundary.
Root node	Each plan has exactly one root node. The result of the query is the output of the root node.

The next figure shows a basic query plan with the data flow elements identified.



Exchange Nodes and Plan Fragments

A plan fragment is a portion of a query plan that executes within a single process. The plan fragment boundaries are identified by the Exchange nodes. These operators are identified as Exchange nodes:

- PARTITION_ACCESS (also identified as PA in EXPLAIN output). This unary node that provides access to a single partition at a time. All disk access within a PARTITION_ACCESS node is serial: that is, one request at a time. This node identifies the DAM process boundary.
- SPLIT_TOP (also identified as PAPA in EXPLAIN output). This node is an N-ary operator that provides concurrent access to more than one PARTITION_ACCESS node. As a result, the SPLIT_TOP node facilitates DAM parallelism. Unlike the other two Exchange nodes, the SPLIT_TOP node does not identify a process boundary.
- ESP_EXCHANGE. This node represents the transfer of data between two sets of processes: bottom processes that produce output rows and top processes that receive the rows produced by the bottom processes.
    - With hash repartitioning, a row produced by a bottom process gets randomly assigned to a top process by applying a hash function to the row’s partitioning key.
    - With range repartitioning, the top processes each have an associated contiguous range of partitioning key values, and a row produced by a bottom process gets assigned to a top process by determining into which range the row’s partitioning key falls.
    - With replication, a copy of each row produced by any bottom process gets sent to all top processes. The ESP_EXCHANGE node also redistributes the input data stream by collecting parallel data streams into one process.

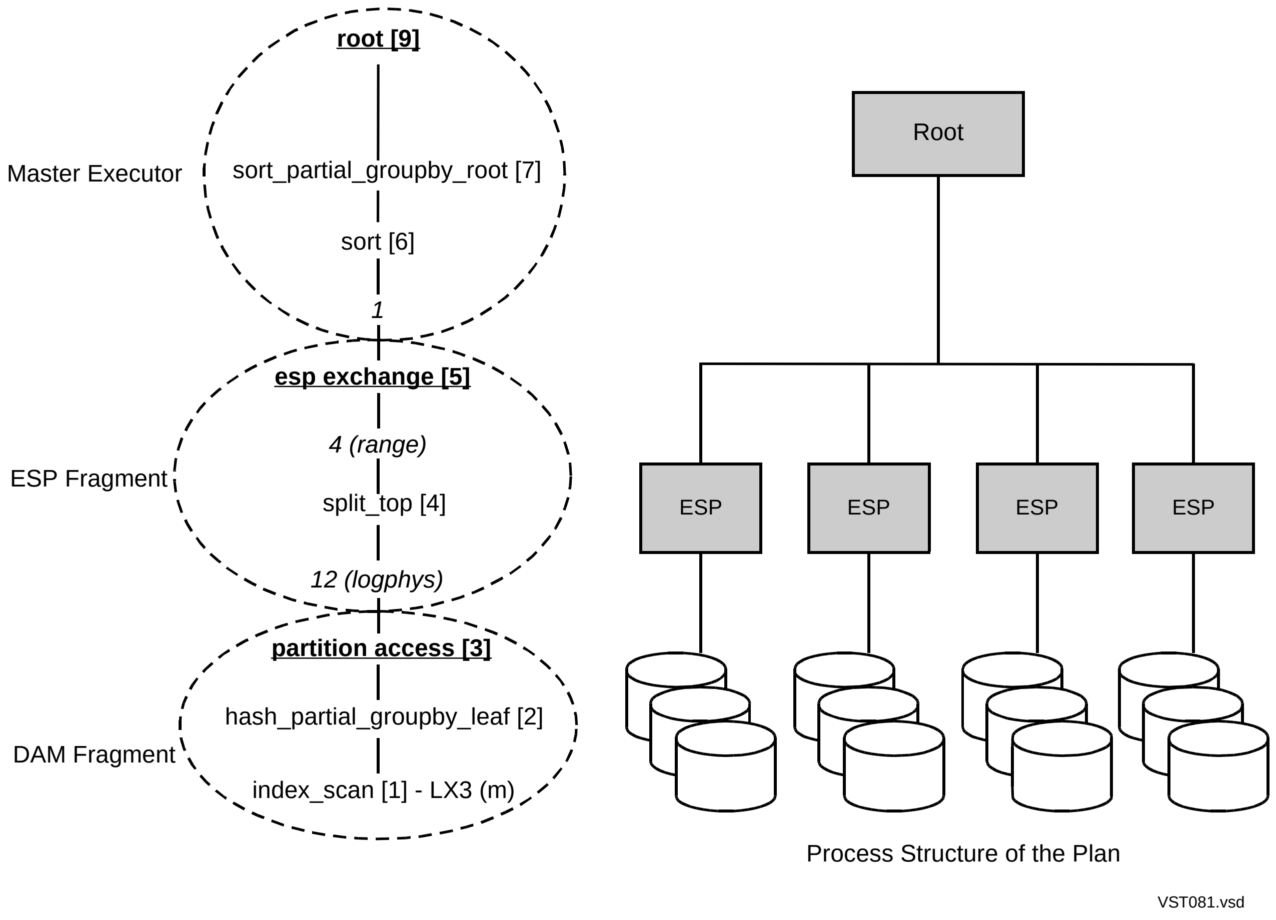
For more information about these types of partitioning functions, see the discussion regarding Join With Matching Partitions on page 8-6.

A plan fragment executes in one of these processes:

- DAM. A plan fragment executes within the DAM process if and only if its root node is a PARTITION_ACCESS node.
- ESP. A plan fragment executes within an ESP process if and only if its root node is an ESP_EXCHANGE node.
- Master executor or root. A plan fragment executes in the master executor process if and only if it contains the ROOT operator.

Multiple instances of a plan fragment can execute in parallel, often on different processors. Each instance differs from the other instances of the same plan fragment only in minor details; for example, the partition boundaries. Each instance usually processes only a part (or partition) of the data.

The next figure shows three plan fragments, DAM (index_scan to partition_access), ESP (split_top to esp_exchange), and the master executor (sort to root). At run time, the plan executes as a collection of processes consisting of 12 DAMs (executing the 12 instances of the DAM plan fragment), 4 ESPs (executing the 4 instances of the ESP fragment), and one master executor process (executing the single instance of the master executor plan fragment).



For more details about understanding plan fragments, plan fragment boundaries, and reading the EXPLAIN output, see Explaining Parallel Plans on page 8-12.

DAM and ESP Parallelism

Partitioned parallelism uses different processes, depending on the type of operators being processed: DAM processes and ESPs.

- DAM parallelism indicates parallel execution in multiple DAM fragment instances. The instances might be accessing different tables, such as in a join or union query, or they might be accessing different partitions of one table under coordination of a SPLIT_TOP node. DAM parallelism is characterized by no-wait communication (asynchronous access). This form of parallelism is inexpensive because it uses existing disk processes; however, it is limited in use. For example, DAM processes cannot repartition, and they might need to service other requests.
- ESP parallelism refers to any parallel plan with at least one ESP plan fragment. ESP parallelism occurs when a plan fragment executes within a special process called the *executor server process* (ESP). ESP parallelism is enabled by the default ATTEMPT_ESP_PARALLELISM.

A cost is associated with starting an ESP process. The optimizer balances this cost against the performance gain resulting from the increased parallelism and chooses ESP parallelism only if the gain exceeds the ESP start-up cost.

Parallel Plan Generation

The SQL/MX optimizer uses different methods to process operators, depending on the general category of operator type, as described next.

Scan, Update, and Delete

DAM parallelism enables multiple PARTITION_ACCESS operators to access data simultaneously by using asynchronous access. Data is consolidated through a SPLIT_TOP node.

Join With Matching Partitions

If the optimizer finds that matching partitions exist in tables that are involved in a join, it tries to join the tables by using the matching partitions algorithm. This type of plan is also known as a Type1 join. Type1 joins can be forced by using the CONTROL QUERY SHAPE statement. For more information, see Influencing Parallel Plans on page 8-24.

In a matching partitions parallel join, the corresponding partitioning key columns of both tables involved in the join must be linked through the join predicate. In addition, for range partitioning, the first key values (that is, the partition boundaries) must match. For hash partitioning, the number of partitions in each table must be identical, and the data types of the corresponding partitioning key columns of both tables must be identical.

Decoupled Keys

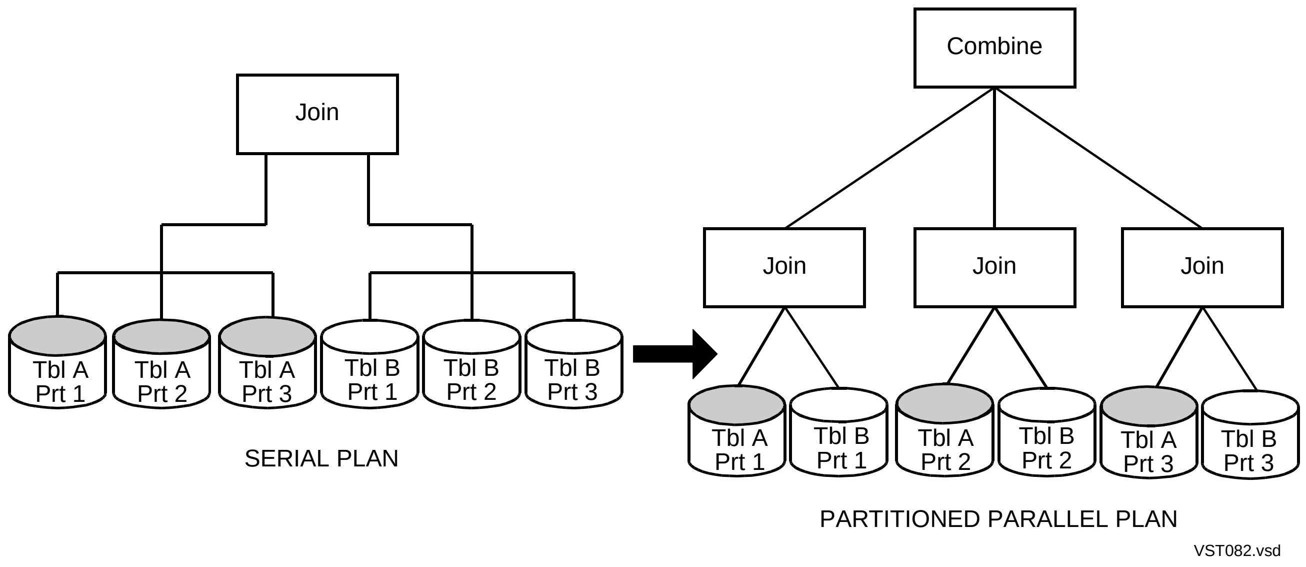
A decoupled table or index is one where the partitioning key is different from the clustering key and it is not a prefix of the clustering key. Decoupled keys follow the same rules as partitioning. If the clustering keys are compatible, the join is more efficient

Range and Hash Partitioning

If the number of partitions match, but not the first key values, the optimizer can still use the matching partitions algorithm while some form of repartitioning or logical subpartitioning occurs to rectify the differing boundary values.

The optimizer has the ability to repartition one input, both inputs, or no inputs (where input means outer and inner tables) before joining if the corresponding partitioning key columns of both tables involved in the join are not linked through the join predicate or if more partitions are needed to produce an optimal degree of parallelism.

In the simple case shown in the next figure, the first partition of Table A is joined with the first partition of Table B, the second partition of Table A is joined with the second partition of Table B, and so on. This method of performing joins works only if, for a given row in Table A, partition 1, all its matching rows are stored in Table B, partition 1. In the case where both Table A and Table B are partitioned on the join attributes with matching key ranges or compatible hashing functions, the optimizer might choose to use this algorithm.



As mentioned previously, the join with matching partitions is called a Type1 join; the case shown in the figure is a generalized Type1 join. Variations of the Type1 join include those discussed next.

- Join With Range Repartitioning

  Range repartitioning allows matching partition joins between two tables where only one table is partitioned on the join column. This type of plan can also be generated if the partition first key boundary values do not match or if the number of partitions of the two tables do not match or if one table is not clustered on the join columns. In this type of plan, only one table is repartitioned.

- Join With Hash Repartitioning

  If both tables are partitioned in a way that does not facilitate parallel execution for the query, the optimizer can request the executor to repartition (reorganize) both tables at run time. The matching partitions join algorithm is used on the reorganized tables.

How the Optimizer Avoids Repartitioning for a Join

Repartitioning involves a lot of extra data movement, so the optimizer tries to avoid it by using a more efficient alternative strategy known as logical partitioning. The optimizer might choose one of these forms:

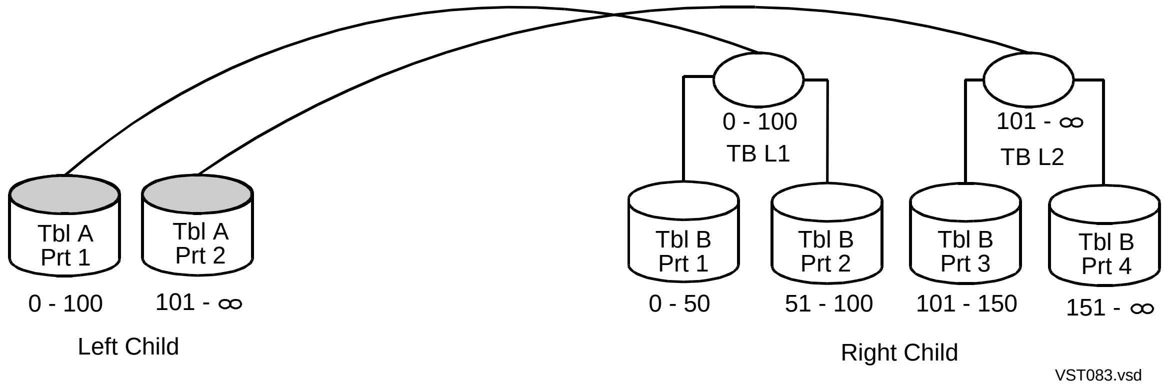
- Logical partition grouping

  Logical partition grouping provides the ability to have fewer ESPs than partitions without repartitioning. The number of ESPs in NonStop SQL/MX is determined more by the number of available CPUs than it is by the number of partitions in the tables. If a table has more partitions than available CPUs, the optimizer can group the partitions so that each ESP processes multiple partitions as if they were a single partition. Each ESP will group adjacent partitions.

  For logical partition grouping in hash partitioned tables, a hash partitioned table can be grouped to have fewer logical partitions, but it can be matched only with another table that has the same number of original partitions. For example, a table with 15 partitions can be grouped to have 4 logical partitions. Three of the logical partitions would have 4 partitions, and one logical partition would have 3 partitions. However, this table can be matched only with another table with 15 partitions and 4 logical partitions. It cannot be matched with a hash partitioned table with 16 partitions and 4 logical partitions.

  The next figure illustrates logical partition grouping (range partition). The left child of the join contains two partitions; the right child contains four partitions. Notice, however, that the partitioning key boundary values of the left child match a subset of the partitioning key boundary values of the right child. (The left child has a partitioning key boundary value of 100, and the right child has partitioning key boundary values of 50, 100, and 150.) The left child is a grouping of the right child, and the right child is a refinement of the left child. In such circumstances, an ESP can combine contiguous physical partitions into a single logical partition.

  This figure shows how physical partitions 1 and 2 of the right child combine to form logical partition 1, and how physical partitions 3 and 4 of the right child combine to form logical partition 2. The logical partitions of the right child match the left child's partitioning scheme.



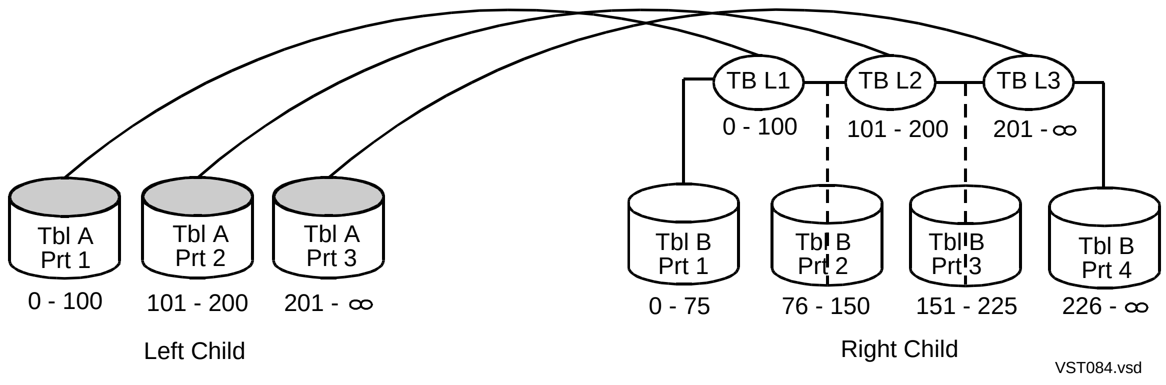
- Join with logical subpartition alignment

  Logical subpartitioning is typically used when the only difference between the join tables is the partition boundaries or the number of partitions. Hash partitioned tables cannot use logical subpartitioning.

  The optimizer can choose this feature if both tables are partitioned on the join columns but the first key values or the number of partitions of the second table do not match the first table and the second table is clustered on the join columns. The partitioning key and the clustering key columns of the second table must be the same.

  For logical subpartitioning, the optimizer behaves as if the other table is partitioned the same way as the first table ("logically" partitions). Consequently, each instance of the join operator (ESP) could be accessing more than one partition or some fraction of a partition (subpartition) of the second table.

  The next figure shows an example of logical subpartitioning. The first table has three partitions. The second table has four partitions. However, because the partitioning key and clustering key columns match the first table, the optimizer can adjust the partitioning key boundary values of the second table into three logical partitions that match the first table.



Join With Parallel Access to the Inner Table

In joins where the inner and outer tables do not match and repartitioning is undesirable or not possible, the optimizer can still achieve parallelism by using parallel access to the inner table. This type of plan is also known as a Type2 join. Type2 joins can be forced by using the CONTROL QUERY SHAPE statement. For more information, see Influencing Parallel Plans on page 8-24.

In a parallel access join, each partition of the outer table is joined with the entire inner table, which might or might not be partitioned. This strategy guarantees the correct result regardless of how tables are partitioned and what the join predicates specify, because any given row in an outer table partition will always be able to find its matching rows in the inner table.

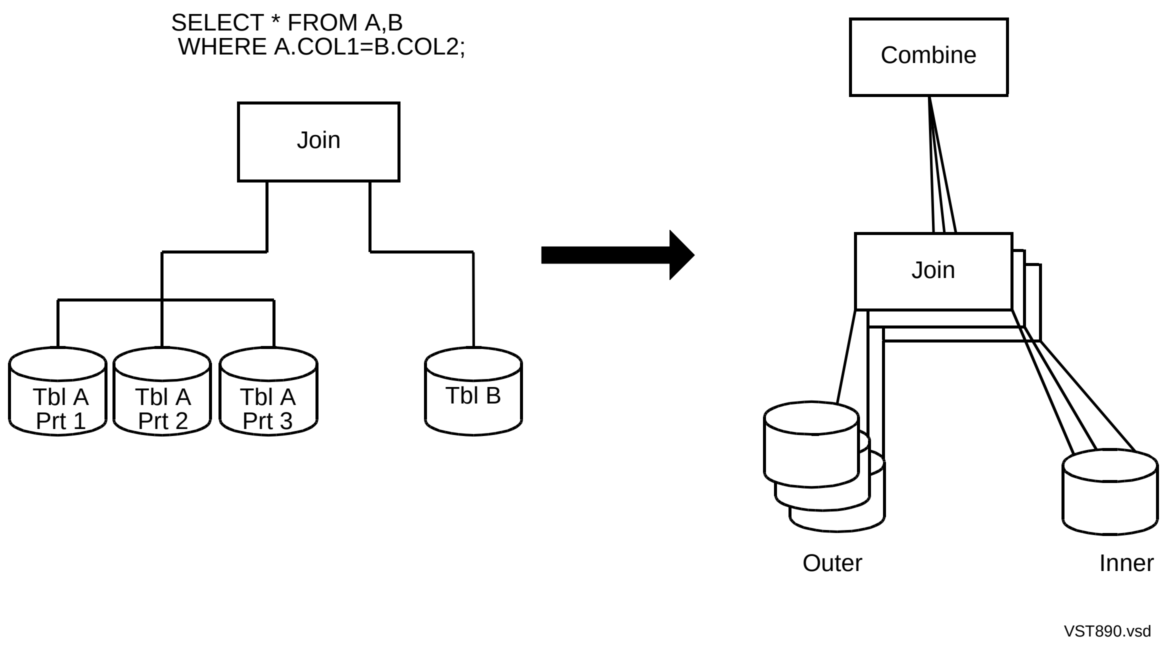
A nested join algorithm might be chosen by the optimizer based on the selectivity of the predicate. If the selectivity is low and the number of probes required to the inner table is small, the nested join algorithm might yield the optimal plan. However, if many probes of the inner table are required (with a lot of concurrent access from the join ESPs to the inner table), the nested join algorithm can cause a performance degradation. This type of join is also called *replicate no broadcast*.

Note. Replicate no broadcast means that no source ESPs exist. The target ESPs independently read the data they need.

You can avoid performance degradation when the parallel access is combined with a hash join, because the inner table is materialized in the memory of the join process. Instead of randomly probing the inner table, each of the join processes receives a complete copy of the inner table from a set of ESPs that broadcast the inner table contents. When the join process receives a complete copy of the inner table, the join is called *replicate by broadcast*.

Note. Replicate by broadcast means that ESPs actually send multiple messages to each recipient.

The next figure shows a nested join with parallel access to the inner table.



Unions

The optimizer can choose to execute the union in parallel when another operator, such as a group by, exists above the MERGE_UNION, which can benefit from receiving parallel streams of data. For parallelism of the MERGE_UNION operator, the optimizer supports a matching partitions union (based on the matching partitions join algorithm). Inputs of the union must be partitioned the same; otherwise, the optimizer must repartition the tables.

A union can be executed in parallel in three ways:

- If you do not need to order the data, the optimizer reads both tables in parallel.
- If you require ordering, the optimizer might use sort or index operations on the individual tables and then merge the streams in parallel.
- In certain cases, the union is run in parallel by using ESPs that read different partitions of each of the tables.

Group Bys

NonStop SQL/MX supports hash and sort group by operations in addition to scalar aggregate operations (group by operation without a group by operator). The grouping can be performed in DAM processes, ESPs, or the master executor.

A full group by can be executed in parallel if the data is partitioned on the group by columns. A full scalar aggregate cannot be executed in parallel.

The partial group by operator can be executed in parallel without regard to the partitioning scheme and might result in duplicate groups. Therefore, the results of a partial group by must be combined in a serial finalization step before the query result is used. A partial scalar aggregate can be executed in parallel without regard to the partitioning scheme, but the partial aggregate must be combined in a serial finalization step.

Sort

NonStop SQL/MX can sort in parallel by using multiple ESPs or serially in the master executor. A parallel sort requires a merge of sorted streams by the master executor or by other ESPs.

Insert and Select

For both parallel and serial inserts and selects, the optimizer tries to insert rows sequentially (rather than randomly) as much as possible. For performance reasons, the optimizer generates a plan so that rows selected are ordered and partitioned the same way as the target table. This feature can involve a sort prior to the insert. You can disable this feature (to ensure that the optimizer does not perform a sort prior to inserting rows) by setting the default setting UPD_ORDERED to OFF. For more information, see System Default Settings That Affect Parallelism on page 8-25 and the SYSTEM_DEFAULTS table entry of the *SQL/MX Reference Manual*.

Combining Different Types of Parallelism

Multilevel parallelism describes the ability of each operator to have a different level of partitioned parallelism, including none. An example of multilevel parallelism is that some of the operators closest to the root are not running in parallel, so they are executing in the master, while lower level operators are running in parallel. It is also possible for different operators to run in parallel but use a different number of ESPs.

NonStop SQL/MX supports the concept of some operators using the natural level of parallelism, while other operators maximize the number of partitions based on the number of available CPUs. At the DAM level, partitioning is fixed.

Explaining Parallel Plans

To review the optimized parallel plan, use the EXPLAIN function. For details on the EXPLAIN function and reading the output, see Section 4, Reviewing Query Execution Plans.

How to Determine if You Have a Parallel Plan

Make sure that the default settings that enable parallelism are on. For more information, see System Default Settings That Affect Parallelism on page 8-25.

Compile the query and use the EXPLAIN function to review the query plan.

Partitioned parallelism in query plans is represented by the operators of the Exchange group: ESP_EXCHANGE and SPLIT_TOP. If your query plan does not contain either of these operators, parallel processing was not used in the plan.

An additional exchange operator, PARTITION_ACCESS, appears in parallel plans but signifies neither ESP nor DAM parallelism. The PARTITION_ACCESS operator is used to describe a portion of an execution plan in which there are requests to DAM without partitioned parallelism. One DAM process is requested at a time.

For information about exchange operators, see Section 7, SQL/MX Operators.

How the Optimizer Chooses the Number of ESPs

The ESP_EXCHANGE operator represents ESP parallelism in a query plan. The number of ESPs chosen for the parallel plan depends on several factors:

- Total number of CPUs in the cluster, which is computed from:
    - The number of CPUs per node in the cluster
    - The number of nodes in the cluster
- The maximum number of ESPs for an operator is the number of CPUs times the default setting value MAX_ESPS_PER_CPU_PER_OP. The default value is 1.

  You cannot have more ESPs than CPUs unless you set the default MAX_ESPS_PER_CPU_PER_OP. You might want to increase the default MAX_ESPS_PER_CPU_PER_OP if your query is I/O bound or if you detect that CPUs are not being completely utilized. For example, a plan that runs on a node (16 CPUs) and accesses 64 partitions would normally use 16 ESPs: one for each CPU. Changing the default value of MAX_ESPS_PER_CPU_PER_OP to 2 might result in 32 ESPs. For more information, see System Default Settings That Affect Parallelism on page 8-25.

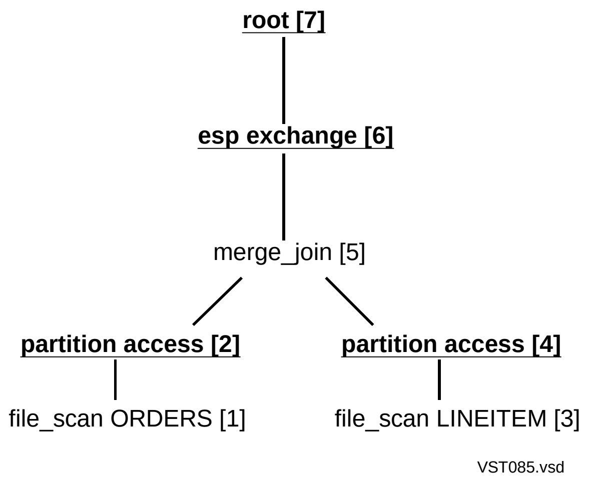
Example

This example shows ESP parallelism in the query tree for the query:

```
SELECT * FROM ORDERS O, LINEITEM L
  WHERE O.O_ORDERKEY=L.L_ORDERKEY
  AND O.O_TOTALPRICE < 25000 AND L.L_QUANTITY < 5;
```

This query looks for small order information where the total price is less than $25,000, and the quantity ordered is less than 5 units. Figure 8-1 on page 8-14 shows the query tree. The EXPLAIN output shows where ESP processes are used for parallelism (the tokens related to parallelism are highlighted). The tables are partititioned on O_ORDERKEY and L_ORDERKEY.

Figure 8-1. Matching Partitions Join Using ESP Parallelism



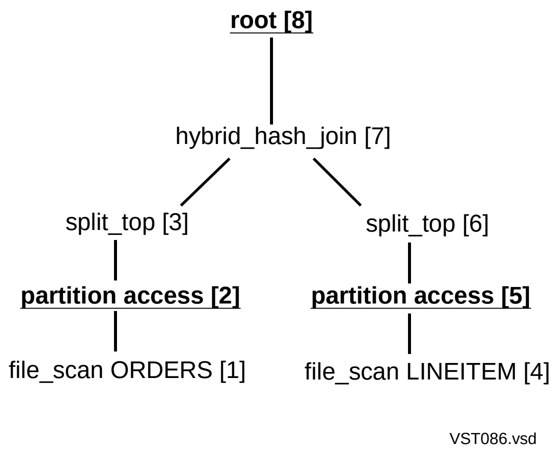
The query plan shows a parallel matching partitions merge join. Two ESPs are started to perform the merge join in parallel (as indicated by the bottom_degree_parallelism token for the ESP_EXCHANGE). Also, the bottom_partitioning_function token indicates that the matching partitions join is range partitioned two ways on the column specified. The tokens relating to parallelism are discussed under Understanding the Parallelism Tokens in the EXPLAIN Output on page 8-16.

A portion of the EXPLAIN output follows:

```
Seq_Num: 6
Operator: ESP_EXCHANGE
Left_Child_Seq_Num: 5
Right_Child_Seq_Num: ?
Cardinality: 1.7563164E+004
Operator Cost: 1.0772745E+000
Total Cost: 6.0278694E+001
Detail Cost: CPU_TIME: 43.9642 IO_TIME: 59.8779
MSG_TIME: 1.30885 IDLETIME: 0.400804 PROBES: 1
bottom_partition_input_values: \:_sys_HostVarLo0,
\:_sys_HostVarHi0, \:_sys_hostVarExclRange
buffer_size: 3951
record_length: 288
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: range partitioned 2 ways on
(([150]equiv(O.O_ORDERKEY)))
```

Figure 8-2 on page 8-15 shows the same query executed by using DAM parallelism to access the partitioned tables. This query plan shows a serial hybrid hash join with no ESPs. However, DAM parallelism is demonstrated by the SPLIT_TOP operator in accessing the underlying partitioned tables. The bottom_partitioning_function token indicates that two PARTITION_ACCESS nodes are started to access the data in parallel for the two-way range partitioned table.

Figure 8-2. Serial Hybrid Hash Join Using DAM Parallelism



A portion of the EXPLAIN output appears next and shows the detail for the two SPLIT_TOP operators:

```
Seq_Num: 3
Operator: SPLIT_TOP
Left_Child_Seq_Num: 2
Right_Child_Seq_Num: ?
Cardinality: 5.9717000E+004
Operator Cost: 3.2347381E+000
Total Cost: 2.1504770E+001
Detail Cost: CPU_TIME: 5.697804  IO_TIME: 9.2084682  MSG_TIME: 0
IDLETIME:  0.7 PROBES:  1
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: logphys partitioned(grouping, PAPA
with 2 PA(s), log=exactly 1 partition, phys=range partitioned 2
ways on (([150]equiv(O.O_ORDERKEY))))

Seq_Num: 6
Operator: SPLIT_TOP
Left_Child_Seq_Num: 5
Right_Child_Seq_Num: ?
Cardinality: 9.3210889E+003
Operator Cost: 5.0158596E-001
Total Cost: 5.4755492E+000
Detail Cost: CPU_TIME: 1.2585902  IO_TIME: 2.1397587 MSG_TIME: 0
IDLETIME: 0.7  PROBES: 1
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: logphys partitioned(grouping, PAPA
with 2 PA(s), log=exactly 1 partition, phys=range partitioned 2
ways on (([150]equiv(O.O_ORDERKEY))))
```

Plan Fragments

To understand the underlying parallelism in your plan, you can divide the query plan into plan fragments. As discussed under Exchange Nodes and Plan Fragments on page 8-3, a plan fragment is a part of the plan that executes in the same process and has one of these nodes as its root:

- ROOT: This fragment is called the root fragment or the master executor and occurs only once per query plan.
- ESP_EXCHANGE: These fragments are called ESP plan fragments and can occur many times in a query plan.
- PARTITION_ACCESS: These fragments are called DAM plan fragments and represent the boundary between the DAM process and the partitions.

A query plan consists of one or more plan fragments. You identify the plan fragments by first compiling the query and reviewing the EXPLAIN output. Fragment boundaries are represented by the exchange nodes ESP_EXCHANGE and PARTITION_ACCESS. Figure 8-6 shows plan fragments identified for a query plan. Each exchange node is part of two fragments.

Draft out the query tree based on the EXPLAIN output. You can identify the plan fragments as shown in Figure 8-6.

At execution time, one or more parallel instances are created for each plan fragment. The root fragment always has one instance and is executed in the master executor.

ESP fragments (ESP_EXCHANGE as the top node) are executed in executor server processes. The EXPLAIN output for the ESP_EXCHANGE node indicates the number of instances created, as described under Understanding the Parallelism Tokens in the EXPLAIN Output next.

DAM fragments (PARTITION_ACCESS as the top node) are executed in the Data Access Manager. Only one active instance of a DAM fragment exists, unless a SPLIT_TOP node is above the PARTITION_ACCESS node. The EXPLAIN output for the SPLIT_TOP node indicates the number of instances created for the DAM fragment, as described next.

Understanding the Parallelism Tokens in the EXPLAIN Output

The exchange operators, ESP_EXCHANGE and SPLIT_TOP, contain tokens in the DESCRIPTION column of the EXPLAIN output that describe the top and bottom degree of parallelism for each plan fragment.

For the ESP_EXCHANGE operator, the top_degree_parallelism token indicates how many instances of the fragment above the node exist. For ESP_EXCHANGE nodes that communicate with the master executor, this value is always one. The bottom_degree_parallelism token indicates how many instances exist of the fragment rooted in this node. The bottom_partitioning_function token provides information about how the plan is parallelized.

For the SPLIT_TOP operator, the top_degree_parallelism token indicates how many fragment instances of the fragment containing the SPLIT_TOP node exist. The bottom_degree_parallelism token indicates how many partitions exist. The bottom_partitioning_function token provides both logical and physical information about the parallel plan. The physical information is the most relevant information and shows you how the fragment is partitioned.

Recognizing the Plan Fragments in a Query Plan

The next series of figures (Figure 8-3 through Figure 8-8) introduces the output that is obtained with `EXPLAIN statement OPTIONS 'f' SQLQUERY` command and the query tree with plan fragments identified for this query:

```
PREPARE SQLQUERY FROM
SELECT l_orderkey, CAST(SUM(l_extendedprice*(1-l_discount))
       AS NUMERIC(18,2)), o_orderdate, o_shippriority
FROM customer,orders,lineitem
WHERE c_mktsegment = 'BUILDING'
      AND c_custkey = o_custkey
      AND l_orderkey = o_orderkey
      AND o_orderdate < DATE '1995-03-15'
      AND l_shipdate > DATE '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY 2 DESC,3 ASC;
```

Note. This query is extracted from the TPCD decision support benchmark and the EXPLAIN statement output is obtained by using the EXPLAIN statement with the OPTIONS 'f' clause. The output portion shown does not include the OPT column. For more information on the OPTIONS 'f' clause, see the `EXPLAIN` statement in the *SQL/MX Reference Manual.*

Figure 8-3. EXPLAIN statement OPTIONS 'f' SQLQUERY Output for Sample Query Using ESP Parallelism

```
LC  RC  OP  OPERATOR                    DESCRIPTION           CARDINALITY
--  --  --  -------------------------   --------------------  ----------
12  .   13  root                                                  1.23E+4
11  .   12  esp_exchange                1:2(range)                1.23E+4
10  .   11  sort                                                  1.23E+4
9   .   10  hash_groupby                                          1.23E+4
2   8   9   hybrid_hash_join                                      3.26E+4
7   .   8   esp_exchange                2(range):2(range)         1.51E+4
4   6   7   merge_join                                            1.51E+4
5   .   6   partition_access                                      7.27E+4
.   .   5   index_scan                  ORDERX1 (s)               7.27E+4
3   .   4   partition_access                                      3.11E+3
.   .   3   file_scan                   CUSTOMER (s)              3.11E+3
1   .   2   partition_access                                      3.24E+5
.   .   1   file_scan                   LINEITEM (s)              3.24E+5
```

This output shows the sequence numbers, the operators, parallel information, and the cardinality for the operators. Using the sequence numbers, you can create a query tree that visually shows the query plan.

Note. The plans shown in Figure 8-4 through Figure 8-8 use a default setting to generate a zig-zag tree shape rather than the default left-leaning tree. For more information on zig-zag trees, see Section 4, Reviewing Query Execution Plans.

Figure 8-4. Query Tree With ESP Parallelism

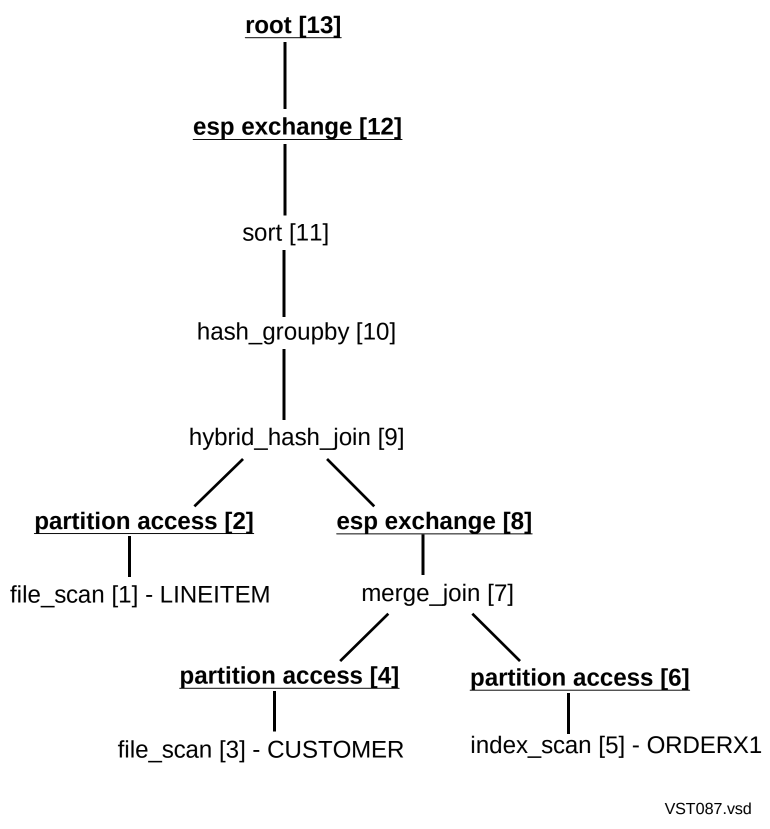
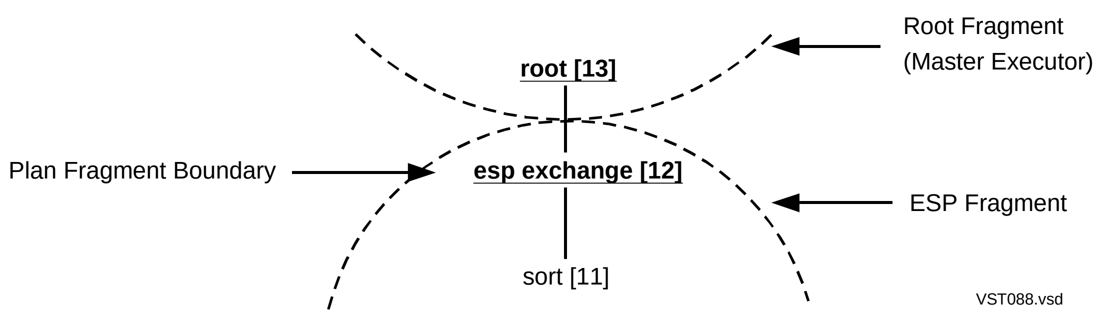


Figure 8-5 shows the ESP_EXCHANGE node that forms the boundary between the root fragment (master executor) and the first ESP fragment.

Figure 8-5. Plan Fragment Boundary

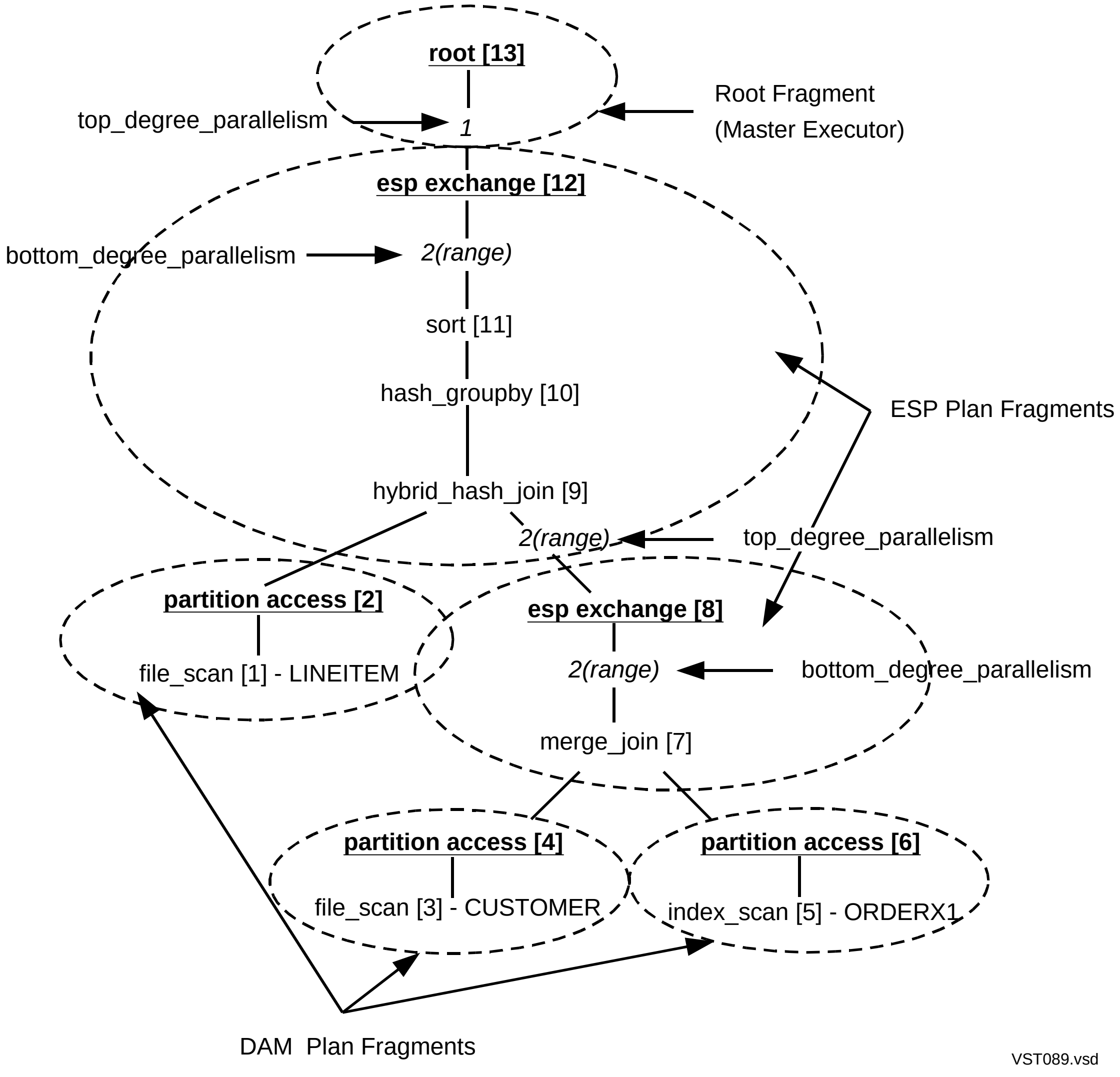


The EXPLAIN output for the ESP_EXCHANGE operator shows tokens in the description that relate to the top degree parallelism (in this case, 1, indicating the parallelism for the root fragment) and the bottom degree parallelism (in this case, 2, indicating the parallelism for the ESP fragment). The bottom_partitioning_function token provides the information about the type of parallel plan.

```
Seq_Num: 12
Operator: ESP_EXCHANGE
Left_Child_Seq_Num: 11
Right_Child_Seq_Num: ?
Cardinality: 1.2347501E+004
Operator Cost: 5.5621022E-001
Total Cost: 2.7717787E+001
Detail Cost: CPU_TIME: 0.0821787  IO_TIME: 0  MSG_TIME: 0
IDLETIME: 0.262715  PROBES: 1
merged_order: inverse(cast(cast((cast(sum((cast(indexcol
(TPCD.XMPS.LINEITEM.L_EXTENDEDPRICE)) * cast((cast((1 * 100)) -
indexcol(TPCD.XMPS.LINEITEM.L_DISCOUNT)))))) / cast(100))))),
indexcol(TPCD.XMPS.ORDERX1.O_ORDERDATE)
bottom_partition_input_values: \:_sys_HostVarLo0,
\:_sys_HostVarHi0, \:_sys_hostVarExclRange
buffer_size: 6250
record_length: 24
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: range partitioned 2 ways on
(([260]equiv(TPCD.XMPS.ORDERS.O_ORDERKEY)))
```

Figure 8-6 shows the entire query plan with plan fragments identified. Sequence numbers for the operators are shown next to the operator name.

Figure 8-6. Sample Query Plan Showing Plan Fragments



Degree of Parallelism

Each plan fragment has its own degree of parallelism, as indicated by the tokens for the exchange operators in the operator description field of the EXPLAIN output. Note that the query plan shown in Figure 8-6 has one root fragment, two ESP fragments, and three DAM fragments. For this query plan, the relevant information for the degree of parallelism is contained in the descriptions of each ESP_EXCHANGE operator. The PARTITION_ACCESS nodes provide information about the beginning and ending key predicates, which instruct each DAM fragment where to start and stop for the partitions it handles.

To further understand the degree of parallelism, this is the same query with no ESPs. Figure 8-7 shows the use of DAM parallelism. The EXPLAIN statement OPTIONS 'f' SQLQUERY shows the output.

```
PREPARE SQLQUERY FROM
SELECT l_orderkey, CAST(SUM(l_extendedprice*(1-l_discount))
       AS NUMERIC(18,2)), o_orderdate, o_shippriority
FROM customer,orders,lineitem
WHERE c_mktsegment = 'BUILDING'
      AND c_custkey = o_custkey
      AND l_orderkey = o_orderkey
      AND o_orderdate < DATE '1995-03-15'
      AND l_shipdate > DATE '1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY 2 DESC,3 ASC;
```

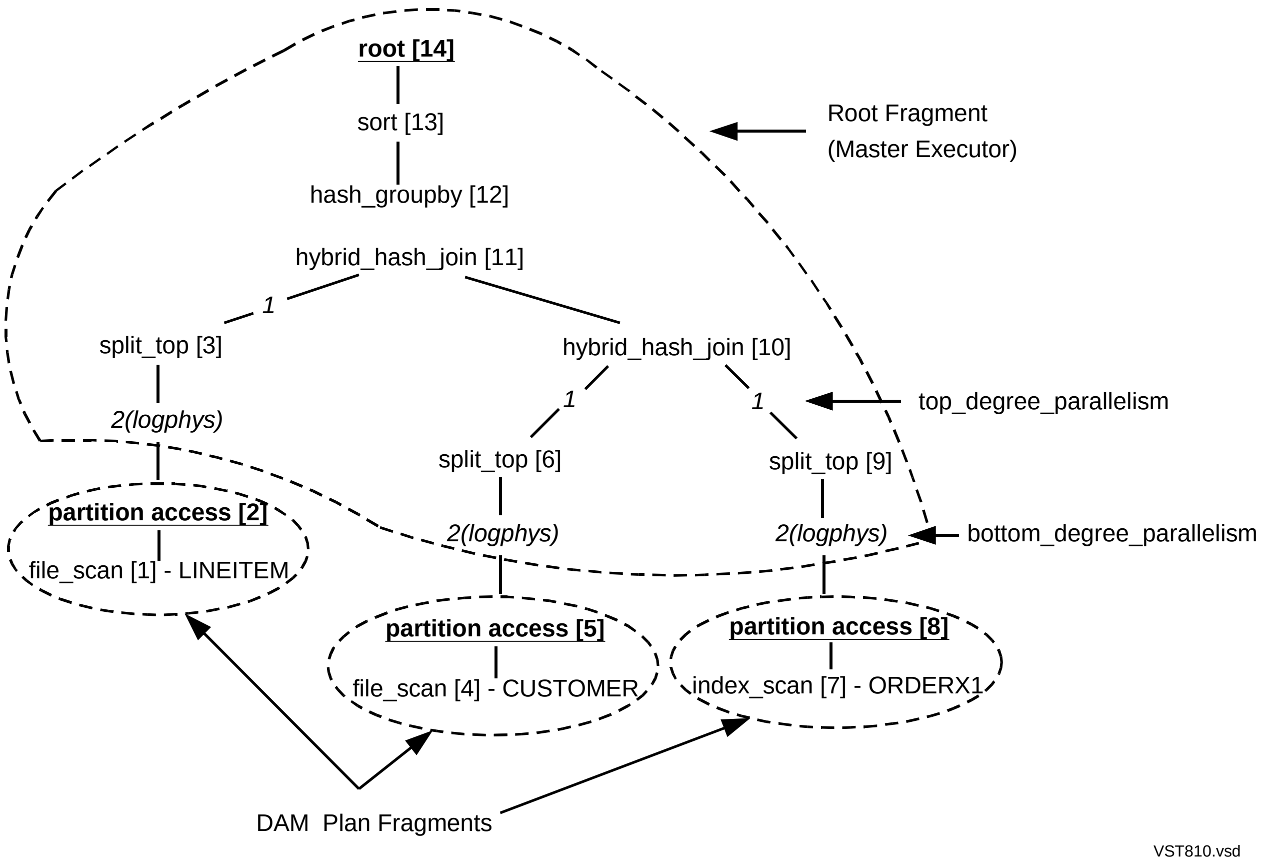
Figure 8-7. EXPLAIN statement OPTIONS 'f' SQLQUERY Output for Sample Query Using DAM Parallelism

```
LC  RC  OP  OPERATOR                  DESCRIPTION        CARDINALITY
--  --  --  ------------------------  -----------------  -----------
13  .   14  root                                             1.23E+4
12  .   13  sort                                             1.23E+4
11  .   12  hash_groupby                                     1.23E+4
3   10  11  hybrid_hash_join                                 3.26E+4
6   9   10  hybrid_hash_join                                 1.51E+4
8   .   9   split_top                 1:2(logph)             3.11E+3
7   .   8   partition_access                                 3.11E+3
.   .   7   file_scan                 CUSTOMER (s)           3.11E+3
5   .   6   split_top                 1:2(logph)             7.27E+4
4   .   5   partition_access                                 7.27E+4
.   .   4   file_scan                 ORDERS (s)             7.27E+4
2   .   3   split_top                 1:2(logph)             3.24E+5
1   .   2   partition_access                                 3.24E+5
.   .   1   file_scan                 LINEITEM (s)           3.24E+5
```

Figure 8-8 on page 8-23 shows the query tree for the plan with the plan fragments identified.

Note. You might wonder why the DAM plan fragment is represented by the PARTITION_ACCESS operator instead of the SPLIT_TOP operator, because the SPLIT_TOP operator represents DAM parallelism. The SPLIT_TOP operator is responsible for dividing the work between the ESP or master executor and the Data Access Manager. The SPLIT_TOP operator assigns the work to the Data Access Manager through the PARTITION_ACCESS nodes, one for each partition.

Figure 8-8. Query Tree for Sample Plan Using DAM Parallelism



Each SPLIT_TOP operator shows the top and bottom degree of parallelism. The EXPLAIN output shows the three SPLIT_TOP operators from the query plan.

```
Seq_Num: 3
Operator: SPLIT_TOP
Left_Child_Seq_Num: 2
Right_Child_Seq_Num: ?
Cardinality: 3.2441872E+005
Operator Cost: 3.4873798E+000
Total Cost: 1.7911737E+001
Detail Cost: CPU_TIME: 3.843434  IO_TIME: 7.6177573495
MSG_TIME: 0 IDLETIME: 0.7 PROBES: 1
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: logphys partitioned(grouping, PAPA
with 2 PA(s), log=exactly 1 partition, phys=range partitioned 2
ways on (([260]equiv(TPCD.XMPS.ORDERS.O_ORDERKEY) )))
```

```
Seq_Num: 6
Operator: SPLIT_TOP
Left_Child_Seq_Num: 5
```

```
Right_Child_Seq_Num: ?
Cardinality: 7.2738547E+004
Operator Cost: 5.6920946E-001
Total Cost: 4.6269255E+000
Detail Cost: CPU_TIME: 0.9101162  IO_TIME: 1.7542597
MSG_TIME: 0  IDLETIME: 0.7 PROBES: 1
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: logphys partitioned(grouping, PAPA
with 2 PA(s), log=exactly 1 partition, phys=range partitioned 2
ways on (([260]equiv(TPCD.XMPS.ORDERS.O_ORDERKEY) )))

Seq_Num: 9
Operator: SPLIT_TOP
Left_Child_Seq_Num: 8
Right_Child_Seq_Num: ?
Cardinality: 3.1110000E+003
Operator Cost: 4.4777110E-002
Total Cost: 1.9319278E+000
Detail Cost: CPU_TIME: 0.1957998  IO_TIME: 0.5657918
MSG_TIME: 0  IDLETIME: 0.7 PROBES: 1
top_degree_parallelism: 1
bottom_degree_parallelism: 2
top_num_data_streams: 1
bottom_num_data_streams: 2
bottom_partitioning_function: logphys partitioned(grouping, PAPA
with 2 PA(s), log=exactly 1 partition, phys=range partitioned 2
ways on (([209]equiv(TPCD.XMPS.CUSTOMER.C_CUSTKEY) )))
```

Influencing Parallel Plans

Several items can affect the plan selected by the optimizer and can determine whether you get parallelism in plans:

- Updated statistics for relevant columns. You should update statistics for any columns appearing in the query. You should also update statistics for subqueries. For more information about updating statistics, see Section 3, Keeping Statistics Current.
- Number of partitions. The number of partitions heavily influences whether the optimizer chooses a parallel plan. You must have more than one partition to obtain a parallel plan. The optimizer uses the partitioning scheme to know how to perform partitioned parallelism.
- Number of CPUs. For ESP parallelism, the number of ESPs possible depends on the number of CPUs per cluster.
- Compiling and executing. You should compile and execute your queries in the same cluster.
- Table partitions. Distribute table partitions evenly across physical disks so that your query can execute in parallel.

- Complexity and quantity of rows returned. The optimizer chooses parallel plans when complex processing on large amounts of data must occur, resulting in a few rows, such as computing an aggregate.
- Forcing parallelism. Use the CONTROL QUERY SHAPE statement to force specific types of parallel plans. See Section 5, Forcing Execution Plans.
- Defaults. Certain defaults might affect the parallel plan, as noted next.

System Default Settings That Affect Parallelism

For more information about default settings and how to override the system-defined default setting, see the *SQL/MX Reference Manual*.

- ATTEMPT_ESP_PARALLELISM

  If ON, the optimizer tries to use the maximum degree of ESP parallelism whenever possible. If the table is too small, the optimizer might not consider ESP parallelism for some operators even with the ON setting. If OFF, the optimizer never generates and costs plans that use ESP parallelism. If SYSTEM (the system-defined default setting), the optimizer considers on an operator-by-operator basis whether it would be worthwhile to generate and cost plans that use ESP parallelism. If the optimizer chooses to do so, the SYSTEM setting also lets the optimizer choose the level of parallelism to use. For each operator, the optimizer uses various heuristics based on the logical and physical operator properties, such as the row selectivity, memory usage, and so on, to determine if generating a parallel plan is worthwhile.

- ATTEMPT_ASYNCHRONOUS_ACCESS

  Enables/disables nowaited access for accessing partitions at the same time (DAM parallelism). The system-defined default setting is ON. This option can greatly improve the performance of SQL/MX queries; it has a lower cost than parallel execution using ESPs. It is recommended that you keep this setting enabled.

- MAX_ESPS_PER_CPU_PER_OP

  Maximum number of ESPs the optimizer considers starting for each CPU for a given operator. The system-defined default setting is one. If set to a value greater than one, the optimizer considers plans where more than one ESP per CPU per operator is considered. For example, if you have four CPUs available and MAX_ESPS_PER_CPU_PER_OP is set to two, the optimizer considers plans where an operator uses eight ESPs (if parallel execution is enabled). This strategy might be advantageous if the CPU load for the query is light but the I/O load is heavy. For example, if the query is I/O bound, and the CPU bandwidth is not needed for other purposes.

- PARALLEL_NUM_ESPS

  Maximum number of ESPs an operator can use (for fewer CPUs per operator). Limits the maximum number of ESPs per operator to a value less than the number of CPUs in the cluster. Limits the number but does not choose which CPUs to use. The system-defined default setting is the number of processors in the cluster.

  If this attribute is not set by the user (SYSTEM setting), the optimizer calculates the degree of parallelism. The optimizer chooses a degree of parallelism equal to the number of CPUs in the cluster times the number of ESPs per CPU. By default, the number of ESPs per CPU is one (1), so typically the optimizer chooses the degree of parallelism equal to the number of CPUs in the cluster.

  If a value is specified for the PARALLEL_NUM_ESPS attribute, the optimizer uses that value for the degree of parallelism unless the value exceeds the number of CPUs in the cluster, in which case it uses the number of CPUs in the cluster as the degree of parallelism. Allowable settings are any unsigned positive integer or SYSTEM.

  The REMOTE_ESP_ALLOCATION attribute also affects parallelism. For more information, see REMOTE_ESP_ALLOCATION on page 4-12.

Index