HP NonStop SQL/MP Query Guide

Abstract

This manual describes how to write queries for an HP NonStop™ SQL/MP database. Users who want information on how to use the SELECT statement, as well as those who program or manage a NonStop SQL/MP database, will find this manual helpful.

Product Version

NonStop SQL/MP G07

Supported Release Version Updates (RVUs)

This publication supports D31.00 and all subsequent D-series RVUs and G06.13 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
524488-003	November 2004

Document History

Part Number	Product Version	Published
093964-000	NonStop SQL/MP D30	December 1994
118375-001	NonStop SQL/MP D30	February 1996
524488-001	NonStop SQL/MP D30	August 2002
524488-002	NonStop SQL/MP G07	February 2004
524488-003	NonStop SQL/MP G07	November 2004

HP NonStop SQL/MP Query Guide

Index Examples Figures Tables

1. Retrieving Data: How to Write Queries (continued)

3. Improving Query Performance Through Query Design (continued)

4. Improving Query Performance With Environmental Options (continued)

6. Analyzing Query Performance

6. Analyzing Query Performance (continued)

Examples

Examples (continued)

Examples (continued)

Figures

Figures (continued)

Tables

What's New in This Manual

Manual Information

Abstract

This manual describes how to write queries for an HP NonStop™ SQL/MP database. Users who want information on how to use the SELECT statement, as well as those who program or manage a NonStop SQL/MP database, will find this manual helpful.

Product Version

NonStop SQL/MP G07

Supported Release Version Updates (RVUs)

This publication supports D31.00 and all subsequent D-series RVUs and G06.13 and all subsequent G-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
524488-003	November 2004

Document History

Part Number	Product Version	Published
093964-000	NonStop SQL/MP D30	December 1994
118375-001	NonStop SQL/MP D30	February 1996
524488-001	NonStop SQL/MP D30	August 2002
524488-002	NonStop SQL/MP G07	February 2004
524488-003	NonStop SQL/MP G07	November 2004

New and Changed Information

Added a new subsection, Processor Assignment by the SQL/MP Optimizer and Executor for Executor Server Processes (ESPs) on page 2-5.

About This Manual

NonStop SQL/MP is an HP implementation of a relational database management system that uses the industry-standard Structured Query Language (SQL) to define and manipulate data.

Who Should Use This Manual?

This manual is for any NonStop SQL/MP user, although there are two primary audiences:

- Users who want information on how to write SELECT statements
- Users who want to know how query design affects system performance

Prerequisites

This manual discusses the use and formulation of queries. Examples are in interactive form, such as that used by the NonStop SQL/MP Conversational Interface (SQLCI). Before reading this manual, you should understand these concepts:

- Tables, including key-sequenced, relative, and entry-sequenced table structures
- Primary keys
- Indexes
- Views
- Data Partitioning

For more information about these concepts, see the *Introduction to NonStop SQL/MP*.

You should also be familiar with the operating system and one of the host programming languages: C, COBOL85, Pascal, or TAL. You should also understand relational database theory and terminology.

If you are not yet familiar with NonStop SQL/MP, you should read the *Introduction to NonStop SQL/MP* and the *SQL/MP Quick Start* before reading this manual.

Organization

Table i. Summary of Contents (page 1 of 2)

Section 1, Retrieving Data: How to Write Queries	Describes how to write queries for a NonStop SQL/MP database and provides information on how to retrieve data.
Section 2, The Optimizer	Explains how the NonStop SQL/MP optimizer chooses a query execution plan and how you can influence its choice of a plan
Section 3, Improving Query Performance Through Query Design	Describes how to write queries so that they capitalize on SQL performance features and how to improve query performances.

Table i. Summary of Contents (page 2 of 2)

Section 4, Improving Query Performance With Environmental Options	Discusses several environmental factors that can influence query performance
Section 5, Selectivity and Cost Estimates	Discusses how the optimizer estimates selectivity, assigning cost to a query, evaluating cost estimates, how the optimizer chooses an execution plan, and forcing execution plans.
Section 6, Analyzing Query Performance	Discusses how to analyze query performance by using DISPLAY STATISTICS, the Measure product, and the EXPLAIN utility.

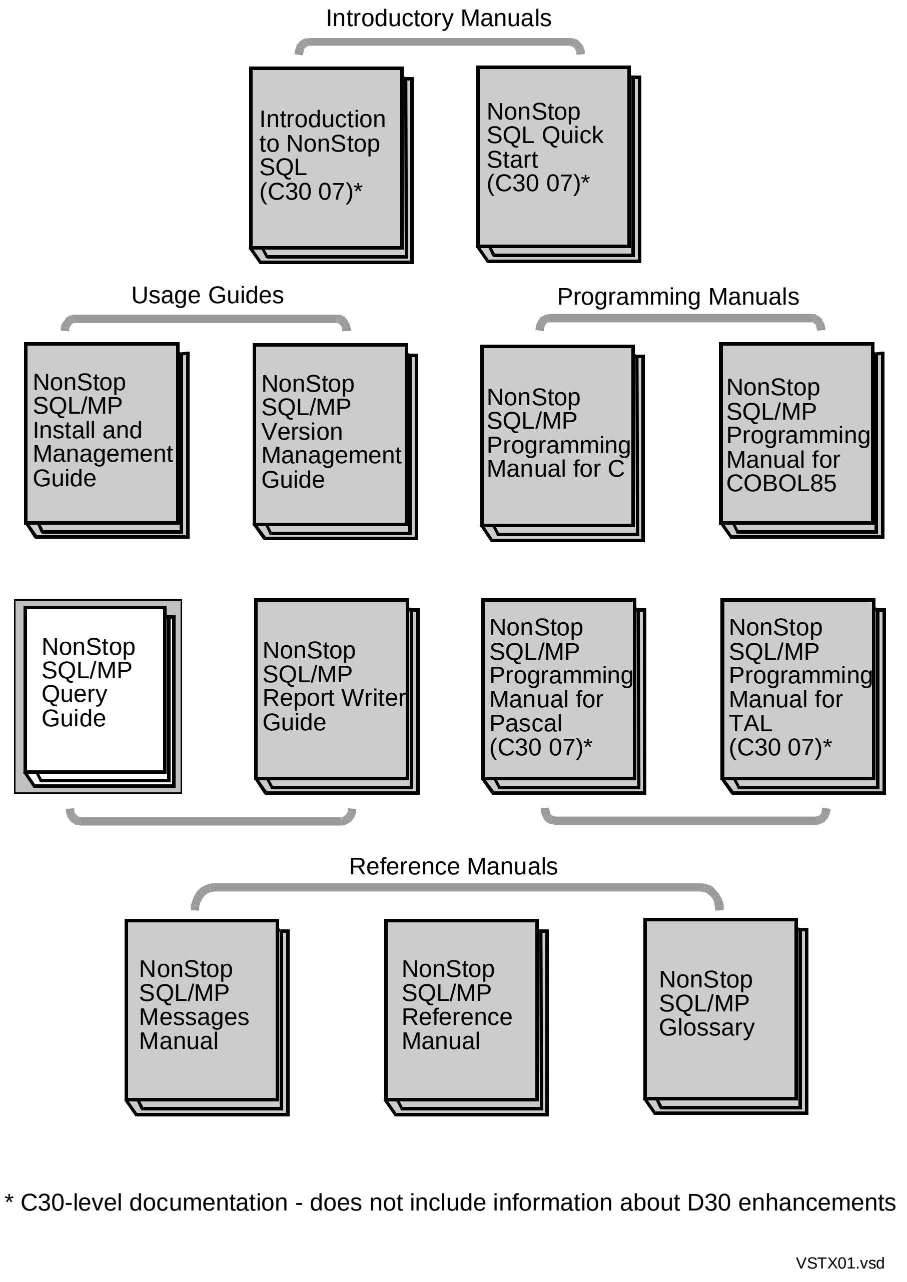
Related Manuals

This manual is part of the NonStop SQL/MP library of manuals, as shown in Figure i on page -xv.

In addition to this manual, the library includes these manuals:

- *Introduction to NonStop SQL/MP* provides an overview of the NonStop SQL/MP relational database management system.
- *SQL Quick Start* describes how to use basic features of the NonStop SQL/MP conversational interface (SQLCI), how to execute simple queries, and how to use the SQLCI report writer to produce a simple formatted report.
- *SQL/MP Reference Manual* describes the language elements and statement and command syntax for all NonStop SQL/MP statements and SQLCI commands.
- *SQL/MP Report Writer Guide* describes use of the NonStop SQL/MP report writer commands and SQLCI options that relate to reports.
- *SQL/MP Programming Manual* (available for C and COBOL) and the *SQL Programming Manual* (available for Pascal and TAL) describe the programmatic interface for the particular host language.
- *SQL/MP Installation and Management Guide* describes how to perform the tasks of planning, installing, creating, and managing a NonStop SQL/MP database.
- *SQL/MP Version Management Guide* provides guidelines for managing NonStop SQL/MP installations using the NonStop SQL/MP version management system.
- *SQL/MP Messages Manual* describes NonStop SQL/MP messages for the NonStop SQL/MP conversational interface, the application programming interface, and NonStop SQL/MP utilities, as well as file-system and FastSort messages returned by NonStop SQL/MP.
- *SQL/MP Glossary* describes the SQL database terminology used in the NonStop SQL/MP documentation library.

Figure i. NonStop SQL/MP Library Map


Introductory Manuals
Introduction to NonStop SQL (C30 07)*
NonStop SQL Quick Start (C30 07)*
Usage Guides
NonStop SQL/MP Install and Management Guide
NonStop SQL/MP Version Management Guide
Programming Manuals
NonStop SQL/MP Programming Manual for C
NonStop SQL/MP Programming Manual for COBOL85
NonStop SQL/MP Query Guide
NonStop SQL/MP Report Writer Guide
NonStop SQL/MP Programming Manual for Pascal (C30 07)*
NonStop SQL/MP Programming Manual for TAL (C30 07)*
Reference Manuals
NonStop SQL/MP Messages Manual
NonStop SQL/MP Reference Manual
NonStop SQL/MP Glossary
* C30-level documentation - does not include information about D30 enhancements
VSTX01.vsd

Notation Conventions

General Syntax Notation

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

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type 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. Type 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 can 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 can 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 typed 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 type 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 this example, no spaces are 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 ]…
```

1 Retrieving Data: How to Write Queries

A query is a statement that requests data from a database. This section describes how to write queries for a NonStop SQL/MP database. You can specify a query explicitly by using interactive SELECT statements, application-embedded SELECT and CURSOR statements, and report writer selections. You can specify a query implicitly in UPDATE, INSERT, and DELETE statements.

A query can use either dynamic or static SQL. Ad hoc queries submitted through an interface such as the NonStop ODBC Server and queries submitted directly through the conversational interface (SQLCI) are likely to be dynamic. NonStop SQL/MP prepares these queries for execution, compiles them, and executes them as soon as they are submitted.

Host-language programs containing embedded SQL statements are likely to use static SQL. NonStop SQL/MP compiles static SQL statements when the program is developed, after the language compiler compiles the host-language source code. For further information, see the *Introduction to NonStop SQL/MP* manual, the *SQL/MP Programming Manual for C,* or the *SQL/MP Programming Manual for COBOL85*.

This section discusses these topics:

- Using the SELECT Statement on page 1-2
- Using Null Values on page 1-14
- Using String Functions on page 1-15
- Using the Concatenation Operator on page 1-20
- Using Date-Time Columns on page 1-21
- Defining Subqueries on page 1-28
- Defining Predicates on page 1-30
- Using CASE Expressions on page 1-43
- Combining Data From More Than One Table on page 1-51
- Using the UNION Operator on page 1-61
- Developing Interactive Multistep Queries on page 1-65

If you are already familiar with SQL and you know how to write queries, you might want to read only parts of this section; for example, you might want to read the subsection that describes how to use join queries.

In general, the examples in this section are written for an interactive interface such as SQLCI, but the same concepts can be used in programmatic queries (embedded

SQL). For more information, see Using the SELECT Statement in Programs on page 1-10.

These related topics, discussed in other manuals, might also be of interest:

- To modify data with an UPDATE, INSERT, or DELETE statement, use query components. For more information about UPDATE, INSERT, and DELETE statements, see the *SQL/MP Reference Manual* and the *SQL/MP Programming Manual* for your host language.
- To modify data in a database being updated concurrently by other users or programs, use transactions to preserve database consistency. The HP NonStop Transaction Management Facility (TMF) simplifies the task of maintaining data consistency.

  For more information, see the *TMF Reference Manual* and the descriptions of the BEGIN WORK, COMMIT WORK, and ROLLBACK WORK statements in the *SQL/MP Reference Manual.*
- To customize query reports, use the SQLCI report writer. The report writer includes report formatting commands, layout and style options, and report functions. For more information, see the *SQL/MP Report Writer Guide*.
- To change the default message file (and thus change the language used to display messages) during an SQL session, use the =_SQL_MSG_*system* DEFINE, described in the *SQL/MP Reference Manual* or in SQLCI online help.

Using the SELECT Statement

To retrieve data from an SQL database, use the SELECT statement. A SELECT statement must contain a select list and a FROM clause:

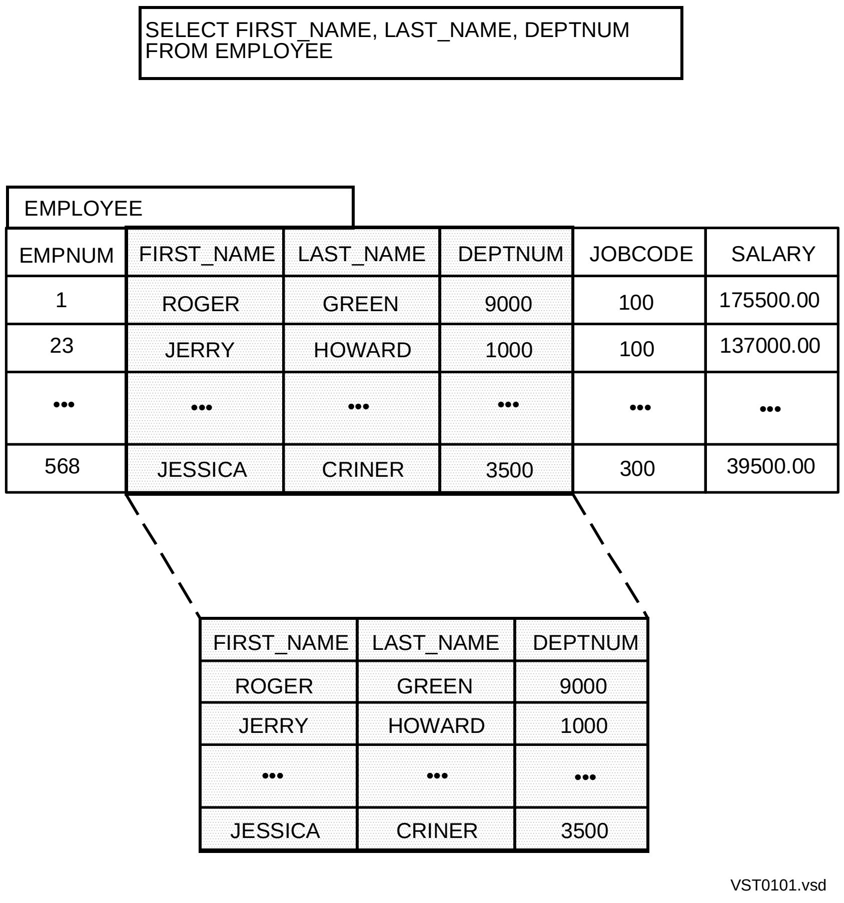
- The select list names the columns to be retrieved.
- The FROM clause identifies the table or tables that contain the columns.

Note. If you are using multiple character sets, a SELECT statement might return column contents that are not supported by some display and print devices.

Selecting Columns

Selecting columns from a table is known as projection. The query in Figure 1-1 selects three columns: FIRST_NAME, LAST_NAME, and DEPTNUM.

Figure 1-1. Selecting Columns From a Table (Projection)



```
SELECT FIRST_NAME, LAST_NAME, DEPTNUM
FROM EMPLOYEE
```

EMPLOYEE

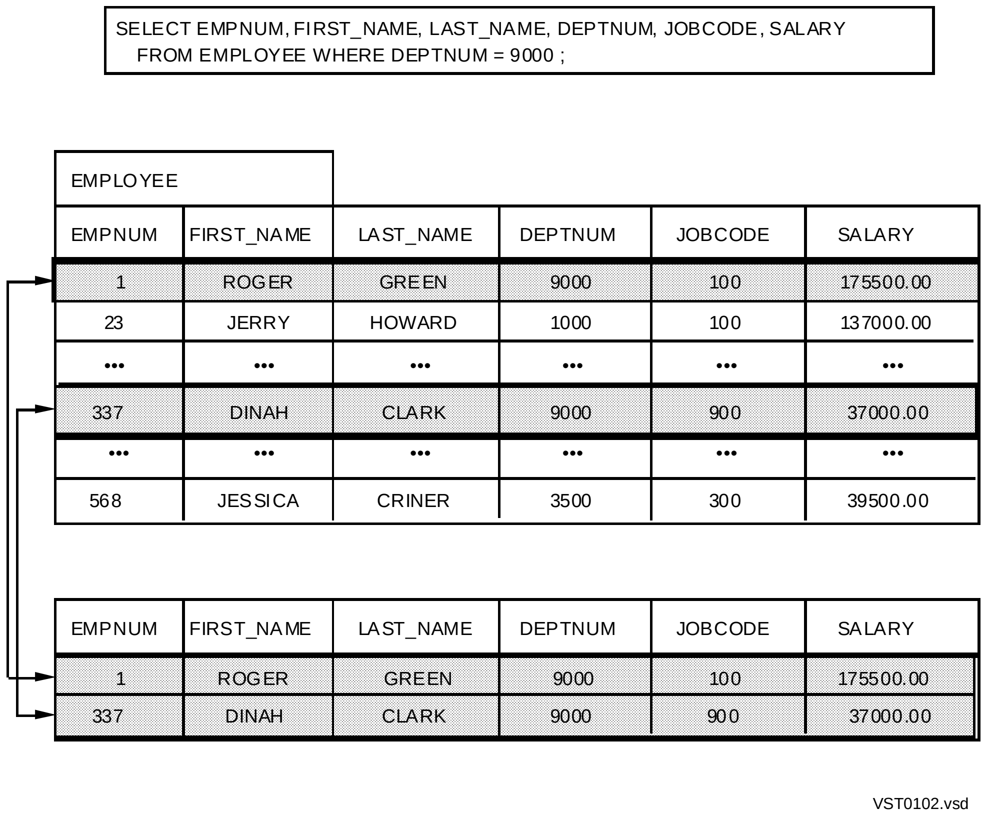
EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE	SALARY
1	ROGER	GREEN	9000	100	175500.00
23	JERRY	HOWARD	1000	100	137000.00
•••	•••	•••	•••	•••	•••
568	JESSICA	CRINER	3500	300	39500.00

FIRST_NAME	LAST_NAME	DEPTNUM
ROGER	GREEN	9000
JERRY	HOWARD	1000
•••	•••	•••
JESSICA	CRINER	3500

Selecting Rows

Selecting rows in a table is called restriction. Figure 1-2 shows a SELECT statement that selects specific rows from the EMPLOYEE table and shows the result table. The SELECT statement uses a WHERE clause and predicates to restrict the number of rows returned: return only those employees who are in department number 9000.

Figure 1-2. Selecting Rows From a Table (Restriction)



```
SELECT EMPNUM, FIRST_NAME, LAST_NAME, DEPTNUM, JOBCODE, SALARY
   FROM EMPLOYEE WHERE DEPTNUM = 9000 ;
```

EMPLOYEE

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE	SALARY
1	ROGER	GREEN	9000	100	175500.00
23	JERRY	HOWARD	1000	100	137000.00
•••	•••	•••	•••	•••	•••
337	DINAH	CLARK	9000	900	37000.00
•••	•••	•••	•••	•••	•••
568	JESSICA	CRINER	3500	300	39500.00

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE	SALARY
1	ROGER	GREEN	9000	100	175500.00
337	DINAH	CLARK	9000	900	37000.00

Organizing Results

A SELECT statement can include one or more of these optional clauses that organize results:

- An ORDER BY clause to list the retrieved rows in a specified order
- A DISTINCT clause to eliminate duplicate rows from the result
- A GROUP BY clause to identify columns used for grouping

Note. When you specify certain options such as ORDER BY, GROUP BY, and DISTINCT, SQL usually performs a sort or hashing operation. Such queries can require significant resources. For more information on using these clauses efficiently, see Minimizing Sort Costs for Ordering and Grouping Operations on page 3-54.

The WHERE clause, used in these examples, is described in Specifying Search Conditions on page 1-8.

The ORDER BY Clause

If you want your report to list employees from highest paid to lowest paid, you can add an ORDER BY clause, as shown in Example 1-1. The DESC keyword tells SQL to sort in descending order; the report lists Ben Henderson, who makes $65,000, before Mary Miller, who makes $56,000. (If you omit the DESC keyword, the ORDER BY clause automatically sorts in ascending order.)

Example 1-1. SELECT Statement With ORDER BY Clause

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY > 50000
   ORDER BY SALARY DESC ;
LAST_NAME        FIRST_NAME      SALARY
-------------    -------------   ----------
GREEN            ROGER           175500.00
  .                .               .
  .                .               .
  .                .               .
HENDERSON        BEN              65000.00
MILLER           MARY             56000.00

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

When evaluating the ORDER BY clause, SQL considers all null values to be equal. Null values are considered greater than nonnull values.

If a collation is specified as part of the ORDER BY clause in a SELECT statement, the character set associated with the collation must be the same as the character set associated with the column in the SELECT statement.

The DISTINCT Clause

The DISTINCT clause eliminates duplicate rows from the result table. Consider the query in Example 1-2. Part numbers 212 and 244 appear several times in the result.

Example 1-2. SELECT Statement With Duplicate Rows

```
SELECT PARTNUM FROM ODETAIL ;
PARTNUM
-------
    244
   2001
    .
    .
    244
   5103
    .
    .
    244
    .
    .
    212
    .
    .
    212
   7301

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

To eliminate the duplicate rows, you can add a DISTINCT clause, as shown in Example 1-3.

Example 1-3. SELECT Statement With DISTINCT Clause

```
SELECT DISTINCT PARTNUM FROM ODETAIL ;
PARTNUM
-------
    244
      .
    212
      .
      .
   7301

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

When evaluating the DISTINCT clause, SQL considers all null values to be duplicates and leaves a single null value.

The DISTINCT clause does not imply ordering; to request a specific order, use the ORDER BY clause.

The GROUP BY Clause

The GROUP BY clause groups rows with the same value and returns one row per group. The GROUP BY clause, like the DISTINCT clause, removes duplicate rows from the result, as well as performing other functions.

To show how the GROUP BY clause works, consider the query from the previous subsection:

```
SELECT PARTNUM FROM ODETAIL ;
```

This query returns 72 rows, with part numbers 212 and 244 appearing several times in the result.

If you specify a GROUP BY clause on the PARTNUM column, as shown in Example 1-4, the query returns the same result as if you had specified a DISTINCT clause.

Example 1-4. SELECT Statement With GROUP BY Clause

```
SELECT PARTNUM FROM ODETAIL
   GROUP BY PARTNUM ;
PARTNUM
-------
    244
      .
    212
      .
      .
   7301

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

The GROUP BY clause is powerful because you can use it to combine information from groups of rows for processing by aggregate functions. For example, you might want to calculate values such as sums and averages.

The GROUP BY clause does not imply ordering; to request a specific order, use the ORDER BY clause.

In Example 1-5, the GROUP BY clause determines the rows to which the SUM function is applied. Each row with the same part number has been grouped and the SUM function applied to the values in the QTY_ORDERED column.

Example 1-5. SELECT Statement With GROUP BY Clause and SUM Function

```
SELECT PARTNUM, SUM (QTY_ORDERED) FROM ODETAIL
   GROUP BY PARTNUM ;
PARTNUM   (EXPR)
-------   ----------
    212           20
    244           47
      .            .
      .            .
   7301           96

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

When evaluating the GROUP BY clause, SQL considers all null values to be equal. The result can have at most one null group.

If a collation is specified as part of the GROUP BY clause in a SELECT statement, the character set associated with the collation must be the same as the character set associated with the column in the SELECT statement.

For more information about aggregate functions such as SUM, see Aggregate Functions in Predicates on page 1-41. For more information about GROUP BY operations, see How the Optimizer Processes Aggregates and Group-By Operations on page 3-46.

Specifying Search Conditions

When you write a query, you can specify a set of conditions called search conditions that restrict the amount of data retrieved from the database. Search conditions determine which rows are returned in the result table.

A search condition consists of one or more subqueries and predicates, used together as a single test for the data:

- A subquery is a form of SELECT statement specified as part of a search condition. Defining Subqueries on page 1-28 describes how to write queries that contain subqueries.
- A predicate is a condition that always evaluates to one of three values: true, false, or unknown (if not enough information is known by SQL to return true or false). SQL returns rows only if the predicate evaluates to true. Example 1-1 used a predicate with the greater-than operator (>) to compare values in the table. For more information, see Defining Predicates on page 1-30.

You can specify search conditions within these clauses of a SELECT statement:

- WHERE clause
- HAVING clause
- ON clause in a SELECT statement involving a join operation

The WHERE clause and HAVING clause are described next. The ON clause is described in Combining Data From More Than One Table on page 1-51.

The WHERE Clause

Suppose that you want a report of all employees whose salaries are greater than $50,000. You can add a WHERE clause to restrict the number of rows returned. Only those employees who earn more than $50,000 are included in the report.

Example 1-6 shows the query and its results.

Example 1-6. SELECT Statement With WHERE Clause

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY > 50000 ;

LAST_NAME        FIRST_NAME       SALARY
-------------    -------------    ----------
GREEN            ROGER            175500.00
  .                  .                .
  .                  .                .
MILLER           MARY              56000.00
HENDERSON        BEN               65000.00

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

The HAVING Clause

You can use the HAVING clause to restrict groups selected by a prior GROUP BY clause; you should use it only in conjunction with the GROUP BY clause.

Example 1-7 shows a SELECT statement with a HAVING clause. Note that the part numbers with a total quantity ordered of 20 or less (such as 212) do not appear in the result:

Example 1-7. SELECT Statement With HAVING Clause

```
SELECT PARTNUM, SUM (QTY_ORDERED)
   FROM ODETAIL
   GROUP BY PARTNUM
   HAVING SUM (QTY_ORDERED) > 20 ;

PARTNUM   (EXPR)
-------   ----------
    244           47
      .            .
      .            .
   7301           96

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

Using the SELECT Statement in Programs

A SELECT statement in a program typically retrieves data into a host variable. You can use two types of SELECT statements in a program:

- A single-row SELECT (also called a singleton or standalone SELECT) that returns a single row or value.
- A multiple-row SELECT (also called a cursor SELECT) that returns multiple rows one row at a time. Use of a cursor handles the uncertainty involved in retrieving a variable number of rows.

These paragraphs describe each type of statement; for performance information, see Section 3, Improving Query Performance Through Query Design.

Single-Row SELECT

A single-row SELECT statement is a request to return a single row to the host program. This method is preferable to a cursor SELECT when only one row needs to be retrieved.

The single row is typically one of these:

- An aggregate without a GROUP BY clause
- A row identified by a unique key value
- A row identified by a unique value of a column within a row

This example shows an aggregate without a GROUP BY clause:

```
SELECT SUM (SALARY) FROM EMPLOYEE ;
```

For more information about aggregate functions, see Aggregate Functions in Predicates on page 1-41.

You can write a single SELECT statement to return the desired row, whether the identifying value is a key value or a nonkey value. Such a SELECT statement contains a WHERE clause that should uniquely identify one row.

You can also use a unique alternate-key value; to optimize efficiency in such a case, the index should be defined with the UNIQUE attribute.

The INTO clause of the SELECT statement is used to return a single-row result of a query to a host variable. Here is an example of a single-row SELECT statement that selects by a primary-key column, `col1`:

```
EXEC SQL
   SELECT col2,
          col3,
          col4
    INTO :hv2,
         :hv3,
         :hv4
    FROM MYTABLE
    WHERE col1 = :hvkey
END-EXEC.
```

In this example, the WHERE clause specifies that the selected row contains a primary key, `col1,` whose value is equal to the value of a specified host variable. Only one row is retrieved from the table because a unique, primary-key value is used for the selection.

Here is an example of a single-row SELECT statement that selects by a nonkey column, `price:`

```
EXEC SQL
   SELECT partnum,
          partdesc,
          price,
          qty_available
    INTO :part-no of parts,
         :part-desc of parts,
         :price of parts,
         :qty_available of parts
    FROM MYTABLE
    WHERE price = :hvdata
END-EXEC.
```

The WHERE clause specifies that the column price contains a value equal to the value of a host variable, *`:hvdata.`*

When the SELECT statement is executed, the system scans the database to find the first row with the specified value in price. When found, this specific row is returned to the program. Because price is not a primary key, and assuming that UNIQUE was not specified for price in any alternate index, the system then reads the rest of the table to make sure the row it found is the only qualifying row; if it is not, the system returns an error.

Multiple-Row (Cursor) SELECT

A multiple-row SELECT statement returns multiple rows one row at a time. This technique is usually preferred over a single-row SELECT when retrieving multiple rows.

A host variable cannot hold data from more than one row, so you must declare a cursor for this type of SELECT statement. A cursor is the mechanism for dealing with a set of rows returned in sequence to an application program. To use cursors, use these SQL statements in your programs:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

Use these statements as indicated in these steps:

1. Name and define a cursor in a DECLARE CURSOR statement. The DECLARE CURSOR statement includes a SELECT statement to describe the rows to be returned.
2. Initialize any host variables used in the SELECT statement. After the cursor is declared and the values initialized, you can open the cursor and fetch each selected row sequentially.
3. Open the cursor using the OPEN statement.
4. Fetch each selected row into the program with the FETCH statement.
5. Close the cursor with the CLOSE statement.

These steps are required even when only the next single row is needed and only one FETCH is done.

Here is a pseudocode example of a multiple-row SELECT statement using a cursor:

```
EXEC SQL
   DECLARE listnext CURSOR FOR
      SELECT partnum,
             partdesc,
             price,
             qty_available
         FROM parts
         WHERE partnum > :hvkey
END-EXEC.
           ...

Move initial value to :hvkey

EXEC SQL
   OPEN listnext
END-EXEC.

EXEC SQL
```

```
   FETCH listnext
     INTO :part-no,
          :part-desc,
          :price,
          :qty_available
END-EXEC.

EXEC SQL
   CLOSE listnext
END-EXEC.
```

A row is returned each time the FETCH statement is executed. This example retrieves all the rows with partnum values greater than the *:hvkey* value.

Note. If a single-row SELECT statement is sufficient, do not use a cursor because it requires three calls (OPEN, FETCH, and CLOSE) instead of one and therefore performs less efficiently for the same result.

Initializing a Cursor

Opening a cursor causes the set of rows in the query result to be defined and ordered. If a cursor SELECT statement contains host variables, you must initialize the values of the host variables before you open the cursor with an OPEN statement.

The SQL executor copies input variables into its buffers when it opens the cursor. If you do not initialize the variables before the OPEN statement, several things can happen:

- If the variables contain values that do not conform to the data types expected, overflow or truncation errors can result when the cursor is opened.
- If the variables are of the expected types but they contain values left from a previous use of the program, these bad values are used as a starting point for subsequent FETCH operations. As a result, the returned rows do not begin at the expected location.

Closing a Cursor

A FREE RESOURCES, COMMIT WORK, ROLLBACK WORK, or an explicit CLOSE statement closes an open cursor. As a general rule, you can leave cursors open to save the overhead of reopening a cursor you plan to use again. In some cases, however, you should explicitly close open cursors. In particular, when you use cursors in Pathway applications, you should follow these rules:

- Close any open cursor before returning control to a requester.
- If your program is a server and a TMF transaction was started by a requester, direct your program to close cursors, to release space used by the cursors and to free locks before returning control to the requester.

The FREE RESOURCES statement is usually more efficient than CLOSE CURSOR unless only a small percentage of defined cursors are active.

Note. A FREE RESOURCES statement is required for nonaudited tables to release locks.

Cursors and Performance

When you use a cursor, the sequential block buffer may be invalidated frequently if these two conditions are true:

- The execution plan for the cursor uses RSBB or VSBB to access a table or view from within a process.
- The process also performs insertions, updates, or deletions against the same table or view.

The result can be poor performance.

To improve performance, use the same cursor to do the updates or deletions. If you must mix cursor retrievals (using sequential block buffering) with insertions into the same table or view, disable sequential block buffering by using a CONTROL TABLE . . . SEQUENTIAL READ OFF directive. For more information on this directive, see the *SQL/MP Reference Manual.*

Using Null Values

A null value is a special symbol, independent of data type, that represents an unknown or inapplicable value. A null value indicates that an item has no value.

A column that allows null values can be empty at any row position. In SQL, such a column has two extra bytes associated with it in each row. A minus one (-)1 stored in those two bytes indicates that the column is empty for that row. Unless a column definition includes the NOT NULL clause to prohibit nulls or the column is part of the primary key of the table, nulls are allowed.

Unless a column definition includes either the DEFAULT clause to specify some value or the NO DEFAULT clause to prohibit any default value, a null value is used as the default value. The default value for a column is the value the system inserts in a row when an INSERT statement omits a value for a particular column or when a column is added to a table.

Various scenarios exist in which a row in a table might contain no value for a specific column, as for example, the following:

- A database of telemarketing contacts might have AGE columns empty if contacts did not give their age.
- An order record might have a DATE_SHIPPED column empty until the order is actually shipped.
- An employee record for an international employee might not have a social security number.

Null values are not the same as blanks. Two blanks can be compared and found equal, while the equivalence of two null values is indeterminate. Similarly, null values are not the same as zeros. Zeros can participate in arithmetic operations, while null values are excluded from arithmetic.

To determine whether a column accepts null values, you can query the COLUMNS catalog table, or you can invoke a table description in the SQL format. The COLUMNS table contains descriptions of all columns of all tables registered in a catalog (as recorded in the TABLES catalog table). The one-character NULLALLOWED column in the COLUMNS table contains a Y if a null value is allowed and an N if a null value is prohibited. For more information, see the *SQL/MP Reference Manual*.

Using String Functions

A function is a specialized routine that can be applied to data to return a result. You can use functions in SQL statements to manipulate characters. For example, using a string function within SQL, you can:

- Extract part of a string
- Search for a string within a string
- Search for a string, disregarding its case
- Determine the length of a string in either characters or bytes
- Remove leading and trailing characters from a string

You can apply string functions to all character data types, including VARCHAR. You can also apply them to CHAR and VARCHAR data types that have the UPSHIFT function applied.

Character operands must all have comparable collations if they will be compared to each other. The result of a string function contains the same character set as the operands.

You can also concatenate the results of string functions. For more information, see Using the Concatenation Operator on page 1-20.

Extracting Part of a String

You can use the SUBSTRING function to extract any part of a string. You do this by providing the string, the starting position for the extraction, and an optional length for the result. The starting position is represented by a count of the number of characters from the beginning of the string. The result of the SUBSTRING function is a character expression called a substring.

Consider these examples:

```
SUBSTRING ("ROBERT JOHN SMITH" FROM 8 FOR 4)
SUBSTRING ("ROBERT JOHN SMITH" FROM 8)
SUBSTRING ("ROBERT JOHN SMITH" FROM 1 FOR 17)
```

In the first example, the extracted string starts from the eighth position of the original string, “ROBERT JOHN SMITH”, and extends for four characters. “JOHN” is the result.

In the second example, the extracted string starts from the eighth position of the original string and extends until the end. “JOHN SMITH” is the result.

The substring in the third example is the whole string. “ROBERT JOHN SMITH” is the result.

If the sum of the starting position and the substring length is greater than the length of the original character string, the substring from the start position to the end of the string is returned. In this example, the sum of 8 and 15 is 23, which is longer than the 17-character string:

```
SUBSTRING ("ROBERT JOHN SMITH" FROM 8 FOR 15)
```

Therefore, the string “JOHN SMITH” is returned.

If you do not specify a substring length, the result is the original string, beginning with the specified start position and continuing through the end of the original string. “RT JOHN SMITH” is the result in this example:

```
SUBSTRING ("ROBERT JOHN SMITH" FROM 5)
```

If the starting position is negative, the negative positions and position 0 are counted as one character each but do not show up in the resulting string:

```
SUBSTRING ("ROBERT JOHN SMITH" FROM -1 FOR 5)
```

Positions -1 and 0 are counted as the first two positions. “ROB” is the resulting string.

Using a Substring in a Query

This example shows a SUBSTRING function used in a query:

```
SELECT LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE SUBSTRING (FIRST_NAME FROM 1 FOR 4) = "MARY" ;
```

A list of all employees whose first names start with “MARY” is the result.

Data Types for Substring Results

A FIXED CHAR or VARCHAR data type returns a VARCHAR data type, and an UPSHIFT CHAR or VARCHAR data type returns an UPSHIFT VARCHAR data type with the same collating attributes as those of the source character string.

This example returns “ROBERT” with a collating sequence of FRENCH:

```
SUBSTRING ("ROBERT JOHN SMITH" COLLATE FRENCH FROM 1 FOR 6)
```

Substring Results That Are Null

If the character string, the starting position, or the substring length is a null value, the result is null.

Substring Results That Are Empty Strings

Sometimes a SUBSTRING function returns a result that is an empty string. An empty string is a string with a length of 0 (“”), which is not the same as a null value.

These examples of SUBSTRING functions return empty strings:

- The sum of the starting position and the substring length is less than 1:

  ```
  SUBSTRING ("ROBERT JOHN SMITH" FROM -5 FOR 3)
  ```

  The sum of -5 and 3 is -2. The resulting substring length is 0.

- The starting position is greater than the length of the character string:

  ```
  SUBSTRING ("ROBERT JOHN SMITH" FROM 19 FOR 3)
  ```

  The starting position is 19, but the string length is only 17.

- The length for the extracted substring is 0:

  ```
  SUBSTRING ("ROBERT JOHN SMITH" FROM 8 FOR 0)
  ```

Substring Errors

An error occurs if either of these conditions is violated:

- If the substring is not part of a dynamically prepared statement, the data types of the starting position and the substring length each must be an exact numeric value with a scale of 0. (If the substring is part of a dynamically prepared statement, the data type is processed as if it were numeric (x,0)).

  In this example, if `colb` is a numeric column with the value 5.3, a compilation error occurs:

  ```
  SUBSTRING ("ROBERT JOHN SMITH" FROM 2 FOR colb)
  ```

- The sum of the starting position and the substring length can be negative if the substring length is not explicitly negative. Because the substring length in this example is an explicit -3, the query returns an error:

  ```
  SELECT SUBSTRING (A FROM 2 FOR -3) FROM TABLE ;
  ```

Searching for a String Within a String

The POSITION function searches for a given substring in a character string and returns the starting character position of that substring. In this example, the result is 6:

```
POSITION ("JANE" IN "MARY JANE MASTERS")
```

You can optionally specify which occurrence of the substring you are seeking; for example, you can specify the first occurrence or the third. The data type of the occurrence is unsigned numeric with a scale of 0. The result for this search is 5:

```
POSITION ("IS" IN "MISSISSIPPI", 2)
```

If no substring is found, the function returns 0. If you omit occurrence, then the function returns the first occurrence of the substring.

Searching for a String Without Regard for its Case

Using the UPSHIFT function, you can ignore the case of a string when searching for a value:

```
SELECT * FROM EMPLOYEE
   WHERE UPSHIFT(LAST_NAME) = "SMITH" ;
```

The example results in a list of all employees whose last names are Smith. The value “SMITH” in the LAST_NAME column can be uppercase, lowercase, or a combination of cases, and it will be found.

You can also use a collation to ignore case, but performance could be better when you use the UPSHIFT function.

Determining the Length of a String

You can use the OCTET_LENGTH function to obtain the number of bytes in a character string. You can use the CHARACTER_LENGTH function, abbreviated CHAR_LENGTH, to obtain the number of characters in a character string.

For multibyte characters, such as Kanji, the OCTET_LENGTH and CHAR_LENGTH functions return results that differ from each other.

```
OCTET_LENGTH (_KANJI "abcdef")
```

returns the value 6, but this example returns the value 3:

```
CHAR_LENGTH (_KANJI "abcdef")
```

For single-byte characters, the results returned by OCTET_LENGTH and CHAR_LENGTH are the same. The data type of the result for single-byte and for multibyte characters for both functions is a 2-byte signed integer with a scale of 0.

When the OCTET_LENGTH or the CHAR_LENGTH functions are applied to a string literal, such as “ROBERT”, the result is the length of the string. But when these functions are applied to a column, the value depends on the definition of the column.

For fixed-length CHAR columns, the result is the length of the column. For columns defined as VARCHAR, the result is the length of the string in the column. Consider this example:

```
CREATE TABLE EMPLOYEE (LAST_NAME CHAR(20),
                       ADDRESS VARCHAR (100);

INSERT INTO EMPLOYEE VALUES ("ROBERT SMITH",
                            ("19333 LEXINGTON PARKWAY") ;

CHAR_LENGTH (LAST_NAME)

CHAR_LENGTH (ADDRESS)
```

Because LAST_NAME is a fixed-length column of 20 characters, the result for the CHAR_LENGTH function on LAST_NAME is 20, if LAST_NAME is not null. If LAST_NAME is null, then CHAR_LENGTH is null.

Because ADDRESS is a variable-length column, the result for the CHAR_LENGTH function on ADDRESS is the current length of the string. The length of the string in the example is 23. If the address is updated to “12 BENTON PARK”, then the function returns the updated string length of 14.

Any argument that is null returns a null value. If the argument is a host variable with a null value or a null result of another function, the result is null. In the previous example, if ADDRESS is null, a null value is returned.

Do not confuse a null value with a string that has a length of 0. A string with 0 length returns a value of 0:

```
CHAR_LENGTH ("")
```

Removing Leading or Trailing Characters From a String

You can use the TRIM function to remove any of these from a character string:

- Leading characters
- Trailing characters
- Both leading and trailing characters

The trim option you specify describes which of the three options you want. If you specify no trim option, the default is both. You can provide the TRIM character you want removed or use the default, which is a blank character.

```
TRIM (ADDRESS)
```

uses the default blank TRIM character, removes leading and trailing blank characters, and implies this:

```
TRIM (BOTH " " FROM ADDRESS)
```

The next example uses an asterisk as a TRIM character and removes leading asterisks from the value in the ADDRESS column:

```
TRIM (LEADING "*" FROM ADDRESS)
```

This example removes trailing blank characters from the value in the LAST_NAME column:

```
TRIM (TRAILING " " FROM LAST_NAME)
```

The resulting string is always VARCHAR. For example, a CHAR or VARCHAR returns a VARCHAR. An UPSHIFT CHAR or VARCHAR returns an UPSHIFT VARCHAR with the same collating and character set attributes as those of the source character string.

The TRIM character and the character string to be trimmed must have comparable collations and identical character sets.

TRIM can be useful with the concatenation operator. For an example, see Using the Concatenation Operator next. You can also use TRIM to do LIKE comparisons with fixed-length host variables. See Using LIKE With TRIM on page 1-35.

Using the Concatenation Operator

Using the concatenation operator (||), you can concatenate two strings to generate a single string result, as in this example:

```
"ROBERT " || "SMITH"
```

which results in:

```
"ROBERT SMITH"
```

If either of the character strings is VARCHAR, then the concatenated result is VARCHAR. If both character strings are CHAR, then the concatenated result is CHAR.

The collating sequence attribute of the concatenated string is determined by the rules you use to determine the collating sequence of a comparison predicate. For further information, see “Comparison Predicate” in the *SQL/MP Reference Manual.*

The concatenation operator is useful in combination with the TRIM function. Consider this table:

```
CREATE TABLE NAMES (FIRST_NAME CHAR(15), LAST_NAME CHAR (15));
INSERT INTO NAMES VALUES ("ROBERT", "SMITH");
```

You can use the concatenation operator to retrieve the full name:

```
FIRST_NAME || LAST_NAME
```

This is the result:

```
"ROBERT         SMITH          "
```

This example removes the trailing blanks after FIRST_NAME and LAST_NAME and inserts one blank between the names:

```
TRIM (TRAILING " " FROM FIRST_NAME || " " ||
  TRIM (TRAILING " " FROM LAST_NAME)
```

The result is:

```
"ROBERT SMITH"
```

For more information on the TRIM function, see Removing Leading or Trailing Characters From a String on page 1-19.

Using Date-Time Columns

SQL supports a set of date-time data types so that you can define, modify, and access data that specifies date and time values. SQL also provides a set of date-time functions you can use in expressions that involve columns defined with the date-time data types.

Table 1-1. Date-Time Data Types

Data Type	Description
DATETIME	Contains a range of logically contiguous date and time fields, called DATETIME columns, in this implied order: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION.
DATE	Designates a date according to the Gregorian calendar and is a synonym for DATETIME YEAR TO DAY. A column of type DATE can contain values that have this contiguous DATETIME fields: YEAR, MONTH, and DAY.
TIME	Designates a time of day according to a 24-hour clock and is the same as DATETIME HOUR TO SECOND. A column of type TIME can contain values that have this contiguous DATETIME fields: HOUR, MINUTE, and SECOND.
TIMESTAMP	Designates a date according to the Gregorian calendar and a time of day according to a 24-hour clock. TIMESTAMP is a synonym for DATETIME YEAR TO FRACTION(6). A column of type TIMESTAMP can contain values that have all the DATETIME fields, which are contiguous: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION.
INTERVAL	Contains values that designate durations of time in either year-month or day-time intervals. The YEAR and MONTH fields of INTERVAL designate a number of years and months. The DAY, HOUR, MINUTE, SECOND, and FRACTION fields of INTERVAL designate a number of days, hours, minutes, seconds, and fractions of a second.

For a complete description of date-time data types, see the *SQL/MP Reference Manual*. Accessing Date-Time Values on page 1-22 describes how to access date-time values in SQL.

Accessing Date-Time Values

For these examples, suppose that the PROJECTS table contains the data shown in Table 1-2. The WAIT_TIME column specifies a number of days.

Table 1-2. Sample Table for Date-Time and INTERVAL Arithmetic Examples

project_name	start_date	end_date	wait_time
920	1988-02-21:20:30	1989-03-21:20:30	20
134	1970-01-01:00:00	1992-01-29:20:30	30
922	1940-02-21:12:30	1991-01-20:12:30	13
955	1990-10-14:14:30	1991-01-20:12:30	14
945	1989-10-20:00:00	1990-10-21:00:00	30

This query accesses two rows of the PROJECTS table:

```
SELECT * FROM PROJECTS
   WHERE PROJECT_NAME = "134"
   OR PROJECT_NAME = "920" ;
```

The query returns this result:

```
PROJECT_NAME   START_DATE         END_DATE           WAIT_TIME
------------   ----------------   ----------------   ---------
920            1988-02-21:20:30   1989-03-21:20:30          20
134            1970-01-01:00:00   1992-01-29:20:30          30

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

Adding an INTERVAL Value to a DATETIME Value

This example adds an INTERVAL value with a YEAR value to a DATETIME value:

```
SELECT end_date + INTERVAL "1" YEAR
   FROM projects
   WHERE project_name = "922" ;
```

The query returns this result:

```
(EXPR)
--------------
1992-01-20:12:30
```

The next example adds an INTERVAL value (WAIT_TIME) to a DATETIME value (START_DATE). The system handles 1988 as a leap year.

```
SELECT start_date + wait_time
   FROM projects
   WHERE project_name = "920" ;
```

The query returns this result:

```
(EXPR)
--------------
1988-03-12:20:30
```

Subtracting an INTERVAL Value From a DATETIME Value

This example subtracts an INTERVAL value with a MONTH value from a DATETIME value:

```
SELECT end_date - INTERVAL "1" MONTH
   FROM projects
   WHERE project_name = "955" ;
```

The query returns this result:

```
(EXPR)
--------------
1990-12-20:12:30
```

In this case, the YEAR value was decremented by 1 because subtracting a month from January 20 caused the date to be in the previous year.

The next example subtracts an INTERVAL value from a DATETIME value and adjusts the adjacent column:

```
SELECT start_date - INTERVAL "15:30" HOUR TO MINUTE
   FROM projects
   WHERE project_name = "922" ;
```

The query returns this result:

```
(EXPR)
--------------
1940-02-20:21:00
```

Adding Two INTERVAL Values

This expression adds two INTERVAL values:

```
INTERVAL "30" DAY + INTERVAL "3" HOUR
```

Because the receiving column (on the left) has a DAY range, the result of adding 3 hours is 30 days. To retain the HOUR value, the receiving column must be defined with the range DAY TO HOUR; for example:

```
WAIT_TIME  INTERVAL DAY(2) TO HOUR  NO DEFAULT  NOT NULL
```

Multiplying an INTERVAL Value

This expression doubles an INTERVAL value:

```
INTERVAL "2-7" YEAR TO MONTH * 2
```

The result is 5 years 2 months.

For example, suppose that you specify this query:

```
SELECT END_DATE + INTERVAL "2-7" YEAR TO MONTH * 2
   FROM PROJECTS
   WHERE PROJECT_NAME = "922" ;
```

The value of the END_DATE column increases by 5 years and 2 months. The query returns this result:

```
(EXPR)
----------------
1996-03-20:12:30
```

Dividing an INTERVAL Value

This expression divides an INTERVAL value by another INTERVAL value:

```
INTERVAL "3" DAY / INTERVAL "2" HOUR
```

The result is 36. Using Date-Time Functions

Date-time functions, as well as the MAX and MIN functions, are used in expressions that involve columns defined with the date-time data types. You can use the date-time functions anywhere a DATETIME expression is allowed.

SQL provides the date-time functions listed in Table 1-3.

Table 1-3. Date-Time Functions

Function	Description
CONVERTTIMESTAMP	Converts a Julian timestamp to a DATETIME value.
CURRENT	Returns the current date, current time, or current date and time.
DATEFORMAT	Formats a DATETIME value.
DAYOFWEEK	Returns an integer representing a day of the week.
EXTEND	Adjusts the range of fields for a DATETIME value.
JULIANTIMESTAMP	Returns the Julian timestamp (an operating system timestamp) representation of a DATETIME value.

Note. SQL does not recognize an operating system timestamp as a data type. An operating system timestamp is a LARGEINT data type that contains valid values for the JULIANTIMESTAMP function.

The date-time qualifiers on both sides of a comparison operator must have the same precision. If, for example, one of your columns contains a fraction value, you might change the other literal to include the fraction column. Alternatively, you could use the EXTEND function to adjust the range.

CONVERTTIMESTAMP Function

Suppose that the BDAY2 table contains birth dates in Julian timestamp form. This example converts the Julian timestamp to a DATETIME value:

```
SELECT NAME, CONVERTTIMESTAMP (JULIAN_BDAY), HOBBIES
   FROM BDAY2
   WHERE NAME = "CAROLYN" ;
```

The query returns this result:

```
NAME      (EXPR)                        HOBBIES
-------   --------------------------    ---------------------
CAROLYN   1957-04-23:00:00:00.000000    GARDENING, DACHSHUNDS

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

CURRENT Function

This example calls the CURRENT function and returns all rows from the PID table with the current date:

```
SELECT PROJ_ID, START_DATE
   FROM PID
   WHERE START_DATE = CURRENT YEAR TO DAY ;
```

If CURRENT is called on January 29, 1992, at 11:30 PM, CURRENT YEAR TO DAY returns all rows with a start date of 1992-01-29. The query returns this result:

```
PROJ_ID           START_DATE
-----------       ------------
5551              1992-01-29

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

The date is returned in the default date format. You can specify a different format by using the DATEFORMAT function.

DATEFORMAT Function

For an example of the DATEFORMAT function, consider modifying the query used previously for the CURRENT function:

```
SELECT PROJ_ID, DATEFORMAT (START_DATE, USA)
   FROM PID
   WHERE START_DATE = CURRENT YEAR TO DAY ;
```

The query returns this result. The date is now in USA format:

```
PROJ_ID            (EXPR)
-----------        ------------
5551               1992/01/29

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

DAYOFWEEK Function

This example retrieves the day of the week from the date-time value in the START_DATE column of the PROJECTS table:

```
SELECT project_name, start_date, DAYOFWEEK(start_date)
   FROM projects
   WHERE project_name = "920";
```

The query returns this result:

```
PROJECT_NAME  START_DATE        (EXPR)
------------  ----------------  ------
920           1988-02-21:20:30       1

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

The value selected is 1, representing Sunday.

The next example retrieves the day of the week from the date-time value in the BIRTHDATE column of the BDAY2 table:

```
SELECT NAME, BIRTHDATE, DAYOFWEEK(BIRTHDATE)
   FROM BDAY2
   WHERE NAME = "CAROLYN" ;
```

The query returns this result:

```
NAME           BIRTHDATE        (EXPR)
-----------    -----------      ------
CAROLYN        1957-04-23            3

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

The value selected is 3, representing Tuesday.

EXTEND Function

In this example, the DAY, HOUR, MINUTE, SECOND, and FRACTION fields to the right of MONTH are initialized to 01 (for DAY), 00 (for HOUR, MINUTE, and SECOND) and 000000 (for FRACTION):

```
EXTEND ( DATETIME "1989-11" YEAR TO MONTH, YEAR TO FRACTION )
```

The function returns this value:

```
1989-11-01:00:00:00.000000
```

In the next example, the YEAR field to the left of MONTH is initialized to the current year. The HOUR and MINUTE fields to the right of MONTH are initialized to 00:

```
EXTEND ( DATETIME "11-24" MONTH TO DAY , YEAR TO MINUTE )
```

In 1989, the function returns this value:

```
1989-11-24:00:00
```

The EXTEND function is especially useful when comparing DATETIME columns that contain different fields.

You can also use the EXTEND function to shorten a DATETIME column. For example, suppose column ATIME is defined as YEAR TO FRACTION (6) and you want to specify grouping on YEAR TO DAY. You can specify this query:

```
SELECT EXTEND ( ATIME, YEAR TO DAY ) FROM TIMER GROUP BY 1 ;
```

JULIANTIMESTAMP Function

This example converts a DATETIME value into a Julian timestamp representation of the value. The query selects the START_DATE column from this row in the PROJECTS table:

```
PROJECT_NAME  START_DATE        END_DATE          WAIT_TIME
------------  ----------------  ----------------  ---------
920           1988-02-21:20:30  1989-03-21:20:30         20
```

```
SELECT JULIANTIMESTAMP( START_DATE )
   FROM PROJECTS
   WHERE PROJECT_NAME = "920" ;
```

The query returns this result:

```
(EXPR)
------------------
211439233800000000
```

Specifying Date-Time Values in Programs

In SELECT, UPDATE, DELETE, and SELECT with INSERT statements, date values can be specified in different ways, depending on whether the date value is a date literal or a parameter or host variable in a program.

For a literal, a date-time value is specified as follows:

```
DATETIME "09-15-1994" YEAR TO DAY
```

For a parameter, a date-time data type is specified as follows:

```
?date TYPE AS DATE
```

Defining Subqueries

SQL supports the construct of nested queries or subqueries. A subquery is a SELECT statement that appears within the body of another expression, such as a SELECT, UPDATE, or DELETE statement, and selects an element for comparison purposes. For example, a subquery can appear in the ON, WHERE, or HAVING clause of a SELECT statement (called the outer query).

A subquery selects an element for the purpose of comparison. A subquery is one way to relate data in more than one table. For example, suppose that you want to find all employees whose salary is greater than the average salary of all employees. Logically, this information can be derived from two separate queries:

```
SELECT AVG(SALARY)
   FROM EMPLOYEE ;
```

The first query returns this result:

```
(EXPR)
--------
48784.65

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

Now enter the second query, substituting the value retrieved from the first query:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY > 48784.65 ;
```

The query returns this result:

```
LAST_NAME        FIRST_NAME       SALARY
-------------    -------------    ----------
GREEN            ROGER            175500.00
  .                   .                .
  .                   .                .
HENDERSON        BEN               65000.00

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

You can combine these two queries into a single query that contains a subquery, as follows:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY > (SELECT AVG(SALARY) FROM EMPLOYEE) ;
```

The SELECT that appears on the right-hand side of the predicate is the subquery, sometimes called an inner query. The other SELECT, sometimes called the outer SELECT, uses the value or set of values computed by the subquery.

The select list of a subquery must contain only one element unless the subquery is part of an EXISTS predicate or a quantified predicate (ANY or ALL), described under Defining Predicates on page 1-30.

The two types of subqueries, correlated and noncorrelated, are described next. A subquery can also be quantified or nonquantified, depending on the type of predicate specified; for more information about the latter, see Quantified Predicates on page 1-38.

The use of subqueries can have an impact on performance; for more information, see Section 3, Improving Query Performance Through Query Design.

Correlated Subqueries

A correlated subquery references values from the outer query. For example, the subquery uses values returned by the outer query and is, therefore, a correlated subquery.

```
SELECT item_name, retail_price
  FROM INVNTRY outer
  WHERE retail_price >
    (SELECT AVG(retail_price)
      FROM INVNTRY
      WHERE producer = outer.producer) ;
```

Note. The preceding example associates the correlation name OUTER with the first occurrence of the INVNTRY table. This technique is also used in subsequent examples. A correlation name is an SQL identifier that you associate with a table or view. You can define correlation names in the FROM clause of the SELECT statement. For more information on correlation names, see the *SQL/MP Reference Manual*.

Noncorrelated Subqueries

A noncorrelated subquery does not reference or depend on the result of the outer query. The subquery is performed only once for the query, instead of being performed once for each qualifying outer table row. This subquery uses two instances of the EMPLOYEE table, EMP1 and EMP2, and is a noncorrelated subquery:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
  FROM EMPLOYEE EMP1
```

```
    WHERE SALARY > (SELECT AVG(SALARY)
      FROM EMPLOYEE EMP2) ;
```

Defining Predicates

A predicate is a condition that a row must satisfy to be returned to the application. For example, “ITEM_NO > 10” is the predicate in this query:

```
SELECT ITEM_NAME, RETAIL_PRICE
FROM INVNTRY
WHERE ITEM_NO > 10 ;
```

You can use these predicates to specify a search condition:

- Comparison predicates (also known as relational predicates), which allow you to compare values:

  `<>   <   <=   =   >=   >`

- BETWEEN predicates, used to perform range comparisons
- LIKE predicates, used to match patterns in character strings
- IN predicates, used to compare one value with a list of values
- EXISTS predicates, used to check whether at least one value satisfies a given condition
- SOME, ANY, and ALL predicates, known as quantified predicates, which enable you to quantify the number of values to be compared
- IS NULL predicates, used to check whether a given value is null

You can combine different kinds of predicates with the operators AND and OR, or use the NOT operator to reverse the truth value of a predicate. In addition, you can use aggregate functions such as MAX and MIN as part of a predicate.

The examples under Comparison Predicate refer to a database that consists of two tables, EMPLOYEE and DEPT, as shown in Example 1-8.

Example 1-8. Sample Tables for Predicate Examples

```
EMPLOYEE Table

EMP_ID  LAST_NAME  FIRST_NAME   DEPT_NUM  MGR_ID  SALARY
------  ---------  ----------   --------  ------  --------
  2703  Smith      James            7620    2705  47500.00
  2705  Simpson    Travis           7600    6554  68000.00
  2906  Nakagawa   Etsuro           6400    6554  72000.00
  3598  Nakamura   Eichiro          6480    2906  50000.00
  4096  Murakami   Kazuo            6410    3598  36000.00
  5361  Smythe     Roger            7690    9069  42650.00
  9069  Smith      John             7690    2705  38760.00
  9502  Smithson   Richard          6400    6554  58300.00


DEPT Table

DEPT_NUM  DEPT_NAME                  DEPT_LOC
--------  ------------------------  --------
    6400  Marketing - Far East           900
    6410  Marketing - Korea              910
    6420  Marketing - Hong Kong          920
    6440  Marketing - Singapore          940
    6470  Marketing - Taiwan             970
    6480  Marketing - Australia          980
    7600  Marketing - USA                100
    7620  Marketing - USA West           120
    7690  Marketing - USA East           200
```

Comparison Predicate

You can use this comparison operators to compare the value of one expression with the value of another expression:

<>   <   <=   =   >=   >

The data types of the two expressions must be compatible for the comparison to take place. For example, an expression of data type CHAR cannot be compared with another expression of data type NUMERIC.

This SELECT statement illustrates the use of the >= comparison operator:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY >= 50000 ;
```

The query returns this result:

```
LAST_NAME  FIRST_NAME  SALARY
---------  ----------  --------
Simpson    Travis      68000.00
Nakagawa   Etsuro      72000.00
Nakamura   Eichiro     50000.00
```

```
Smithson   Richard     58300.00

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

You can also use comparison operators with a subquery that returns a single value (sometimes called a scalar subquery). Suppose that you want to know the name and location of the department that James Smith (employee ID 2703) is in. This example illustrates the use of the = operator with a subquery:

```
SELECT DEPT_NAME, DEPT_LOC
   FROM DEPT
   WHERE DEPT_NUM =
   (SELECT DEPT_NUM FROM EMPLOYEE
   WHERE EMP_ID = 2703) ;
```

The query returns this result:

```
DEPT_NAME                 DEPT_LOC
------------------------  --------
Marketing - USA West           120

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

If a key column has a collation, you can use a comparison predicate as a begin or end key only if you compare the column to a value that has the same collation and the same length. A begin key establishes an initial row position within a table or index; An end key establishes a stopping point.

If the comparison predicate compares two character expressions, these guidelines apply to the use of character sets:

- The same character set must be associated with each of the two character expressions. Any character data type is compatible with other character data types as long as both have the same associated character set.
- If neither character expression references a column with an associated collation, a binary comparison is used for all comparisons.
- If the character expressions have two different lengths, the shorter string is filled on the right with spaces to match the length of the longer string. Spaces are used whether or not a single-byte or double-byte character set is associated with the expression.

BETWEEN Predicate

You can use a BETWEEN predicate to perform a bounded search. A bounded search is a search within a specific range of values. Predicates that specify bounds on a search are also called range predicates.

Consider this statement:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY BETWEEN 50000 AND 72000 ;
```

SQL transforms this kind of predicate into a range predicate, as follows:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY >= 50000 AND SALARY <= 72000 ;
```

The query returns this result:

```
LAST_NAME  FIRST_NAME  SALARY
---------  ----------  --------
Simpson    Travis      68000.00
Nakagawa   Etsuro      72000.00
Nakamura   Eichiro     50000.00
Smithson   Richard     58300.00

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

If the BETWEEN predicate compares two character expressions, these guidelines apply to the use of character sets:

- The same character set must be associated with each of the two character expressions. Any character data type is compatible with other character data types as long as both have the same associated character set.
- If neither character expression references a column with an associated collation, a binary comparison is used for all comparisons.
- If the character expressions have two different lengths, the shorter string is filled on the right with spaces to match the length of the longer string. Spaces are used whether or not a single-byte or double-byte character set is associated with the expression.

LIKE Predicate

A LIKE predicate searches for character strings that match a pattern. You can use LIKE to find a character string within a column. For example, you might want to find the string “Ann” in the VARCHAR column FIRST_NAME.

```
SELECT FIRST_NAME FROM NAMES
   WHERE FIRST_NAME LIKE "Ann" ;
```

SQL retrieves all values in the column FIRST_NAME that contain the string “Ann” and contain no trailing blanks.

LIKE performs differently on CHARACTER and VARCHAR columns.

Using LIKE With CHARACTER Columns

Columns of data type CHARACTER are fixed length. If the string stored in CHARACTER data columns is shorter than the length of the column, the string is padded with blanks for the length of the column. For example, if you insert “Joe” into a CHAR(6) column, the stored value becomes “Joe   ”, which is padded with three blank characters at the end of the string. In this example, the column FIRST_NAME contains the value “Joe   ”.

```
SELECT FIRST_NAME FROM NAMES
   WHERE FIRST_NAME LIKE "Joe" ;
```

The query does not result in a match because of the column value’s blank padding. For information on how to force a match in similar instances, see Using LIKE With TRIM on page 1-35.

Using LIKE With Varying-Length Character Columns

Columns of varying-length data types do not include trailing blanks unless you specify trailing blanks when you enter the values. For example, if you enter “Joe” into a VARCHAR(4) column, the column contains “Joe”, with no padded blanks. In the previous example, if FIRST_NAME were a VARCHAR column containing “Joe”, the result would be a match.

Using LIKE With Wild-Card Characters

The LIKE predicate becomes more useful when you use wild-card characters. Wild-card characters allow SQL to select values that include any characters, instead of specific characters. For example, “Joe%” contains the wild-card character %. The % character represents 0 or more characters of any value. By specifying “Joe%” as the pattern for SQL to match, you are directing SQL to select all values that start with “Joe” and end with any or no characters. In this example, “Joey”, “Joellen”, and “Joe” are matches for “Joe%”:

```
SELECT FIRST_NAME FROM NAMES
   WHERE FIRST_NAME LIKE "Joe%" ;
```

The underscore (_) wild-card character signifies that any single character is acceptable. For example, “Joe_” matches “Joey” but not “Joellen” or “Joe”.

You can use more than one wild-card character. For example, “%Joe%” indicates a string that contains the string “Joe”, regardless of the characters or number of characters that precede or follow “Joe”.

Wild-card characters work on columns of fixed or varying length.

Be sure to specify wild-card characters (percent and underscore) in the character set associated with the column or you might receive unexpected results. The *SQL/MP Reference Manual* specifies the values for the character sets SQL supports.

Using LIKE With TRIM

You can use LIKE in combination with TRIM to do comparisons. This combination is useful when the host variable is fixed length and the pattern to match is shorter than the length of the host variable.

In this example, *`:hostvar`* contains the value “ROB%   ” (padded with three blanks):

```
CREATE TABLE NAMES (FIRST_NAME VARCHAR (10)) ;
INSERT INTO NAMES VALUES ("ROBERT") ;
INSERT INTO NAMES VALUES ("ROB") ;
INSERT INTO NAMES VALUES ("ROBBIE") ;
INSERT INTO NAMES VALUES ("PEDRO") ;
```

```
SELECT FIRST_NAME FROM NAMES
   WHERE (FIRST_NAME) LIKE TRIM (:hostvar) ;
```

SQL selects the first three rows because *`:hostvar`* is trimmed to “ROB%” and the % wild card directs SQL to select all NAMEs whose values begin with “ROB” and have any or no trailing characters.

For information on TRIM, see Removing Leading or Trailing Characters From a String on page 1-19.

Using LIKE Efficiently

Some formulations of LIKE predicates are more efficient than others. For more information, see Using LIKE Predicates on page 3-24.

For more information on LIKE, see the *SQL/MP Reference Manual.*

Predicates Connected by OR Operators

You can use the OR operator to connect predicates. You might want to use the OR operator, for example, to select a set of rows that have different values in the same column or to compare values in different columns.

In this example, the OR operator is used to select information about two employees from the database:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, DEPT_NUM, MGR_ID
   FROM EMPLOYEE
   WHERE EMP_ID = 2705 OR EMP_ID = 2906 ;
```

The query returns this result:

```
EMP_ID       LAST_NAME  FIRST_NAME  DEPT_NUM     MGR_ID
------       ---------  ----------  --------     ------
  2705       Simpson    Travis          7600       6554
  2906       Nakagawa   Etsuro          6400       6554

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

IN Predicate

You can use an IN predicate to compare the value of an expression with one or more values of another expression. There are two types of IN predicates, classified according to the comparison expression:

- An IN predicate with a list of values as its comparison expression
- An IN subquery predicate

If the expression is a list of values, the predicate is specified as shown by this example:

```
SELECT DEPT_NUM, LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE DEPT_NUM IN (6400, 6410, 6470, 6480) ;
```

The query returns this result:

```
DEPT_NUM     LAST_NAME  FIRST_NAME
--------     ---------  ----------
    6400     Nakagawa   Etsuro
    6480     Nakamura   Eichiro
    6410     Murakami   Kazuo
    6400     Smithson   Richard

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

SQL transforms the list of values associated with an IN predicate into a search condition involving predicates connected by one or more OR operators. The IN predicate in the previous example is transformed into this search condition:

```
WHERE DEPT_NUM = 6400
   OR DEPT_NUM = 6410
   OR DEPT_NUM = 6470
   OR DEPT_NUM = 6480
```

Therefore, the IN predicate that uses a list of values provides a convenient way for formulating OR predicates.

The second type of IN predicate is provided by using a subquery, instead of a list of values, as follows:

```
SELECT DEPT_NUM, LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE DEPT_NUM IN (SELECT DEPT_NUM
      FROM DEPT
      WHERE DEPT_LOC BETWEEN 900 AND 999) ;
```

The query returns this result:

```
DEPT_NUM     LAST_NAME  FIRST_NAME
--------     ---------  ----------
    6400     Nakagawa   Etsuro
    6480     Nakamura   Eichiro
    6410     Murakami   Kazuo
    6400     Smithson   Richard

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

The list provided by the subquery is a result of the location of the department. (If the database contained employees in departments 6420, 6440, or 6470, SQL would have also selected these departments and their employees.)

If the IN predicate compares two character strings, these guidelines apply to the use of character sets:

- The same character set must be associated with each of the two character expressions. Any character data type is compatible with other character data types as long as both have the same associated character set.
- If neither character expression is a column with an associated collation, a binary comparison is used for all comparisons.
- If the character expressions have two different lengths, the shorter string is filled on the right with spaces to match the length of the longer string. Spaces are used whether or not a single-byte or double-byte character set is associated with the expression.

EXISTS Predicate

An EXISTS predicate always involves a subquery. The EXISTS predicate evaluates to true if the subquery selects a row that satisfies the specified condition or conditions. This statement queries the database to select information about all those employees who are identified as managers:

```
SELECT DISTINCT EMP_ID, LAST_NAME, FIRST_NAME
   FROM EMPLOYEE EMP1
   WHERE EXISTS (SELECT MGR_ID
     FROM EMPLOYEE EMP2
     WHERE EMP2.MGR_ID = EMP1.EMP_ID) ;
```

The query returns this result:

```
EMP_ID       LAST_NAME  FIRST_NAME
------       ---------  ----------
  2705       Simpson    Travis
  2906       Nakagawa   Etsuro
  3598       Nakamura   Eichiro
  9069       Smith      John

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

Quantified Predicates

A quantified predicate always involves a subquery. You can use a quantified predicate to compare an expression that applies to the outer query with all, any, or some of the values returned by a subquery predicate.

If you specify ALL, a row selected by the outer query appears in the result if the quantified predicate is true for each and every value selected by the subquery.

This example queries the database to select information about employees whose salary is greater than or equal to the salary of all other employees:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY >= ALL (SELECT SALARY
         FROM EMPLOYEE) ;
```

The query returns this result:

```
LAST_NAME  FIRST_NAME  SALARY
---------  ----------  --------
Nakagawa   Etsuro      72000.00

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

If you specify SOME or ANY, a row selected by the outer query appears in the result if the quantified predicate is true for at least one value selected by the subquery, as follows:

```
SELECT LAST_NAME, FIRST_NAME, DEPT_NUM
   FROM EMPLOYEE
   WHERE DEPT_NUM = ANY (SELECT DEPT_NUM
         FROM DEPT
         WHERE DEPT_LOC BETWEEN 900 AND 999) ;
```

The query returns this result:

```
LAST_NAME  FIRST_NAME  DEPT_NUM
---------  ----------  --------
Nakagawa   Etsuro          6400
Nakamura   Eichiro         6480
Murakami   Kazuo           6410
Smithson   Richard         6400

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

SOME and ANY are synonyms. In this example, a SOME query is formulated to select information about employees who are also managers. Note that this example is logically equivalent (that is, it retrieves the same data) to the EXISTS example previously discussed.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE EMP_ID = SOME (SELECT MGR_ID
         FROM EMPLOYEE) ;
```

The query returns this result:

```
EMP_ID       LAST_NAME  FIRST_NAME
------       ---------  ----------
  2705       Simpson    Travis
  2906       Nakagawa   Etsuro
  3598       Nakamura   Eichiro
  9069       Smith      John

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

If the quantified predicate compares two character strings, these guidelines apply to the use of character sets:

- The same character set must be associated with each of the two character expressions. Any character data type is compatible with other character data types as long as both have the same associated character set.
- If neither character expression is a column with an associated collation, a binary comparison is used for all comparisons.
- If the character expressions have two different lengths, the shorter string is filled on the right with spaces to match the length of the longer string. Spaces are used whether or not a single-byte or double-byte character set is associated with the expression.

IS NULL Predicate

You can use the IS NULL and IS NOT NULL predicates to determine whether a column contains an unknown (null) value.

The existence of null values produces logic with three possible values: true, false, and unknown.

Table 1-4 on page 1-40 summarizes expression evaluation with null values. For complete syntax and details on each type of expression, see the *SQL/MP Reference Manual*.

Table 1-4. Evaluation of Expressions That Contain Null Values

Expression Type	Condition	Result
Boolean (AND, OR, NOT)	Either value is null or both values are null.	Null
Arithmetic	Either value is null or both values are null.	Null
Aggregate (except COUNT)	Expression is evaluated after eliminating nulls.	Null if set is empty
COUNT DISTINCT	Expression is evaluated after eliminating nulls.	Zero if set is empty
COUNT	Expression is evaluated without eliminating nulls.	Zero if set is empty
> < = >=<= <> LIKE	Either value is null or both values are null.	Null
IN predicate	Expression is null.	Null
Subquery	No values are returned.	Null

If the operand of the IS NULL predicate evaluates to null, then the IS NULL predicate evaluates to true; otherwise, IS NULL evaluates to false. For example, this predicate finds all rows with null values in the FIRST_NAME column:

```
FIRST_NAME IS NULL
```

Similarly, if the operand of the IS NULL predicate evaluates to a value other than null, then the IS NOT NULL predicate evaluates to true; otherwise, IS NOT NULL evaluates to false. This predicate evaluates to true if the value in *:jobcode* is not null:

```
:jobcode IS NOT NULL
```

Multivalued Comparison Predicate

A multivalued comparison predicate specifies more than one value in the same predicate.

These examples show two formulations of the same query. The first example uses the AND and OR operators to retrieve the desired result. The second example shows how you can simplify the query using a multivalued predicate.

This query, written with the AND and OR operators, lists all employees whose names follow JAMES SMITH:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE LAST_NAME > "Smith"
     OR ( LAST_NAME = "Smith"
     AND FIRST_NAME > "James" ) ;
```

Note that there are two conditions for the name Smith, so that the query retrieves information about other employees with the last name of Smith as well as employees whose last name follows Smith in the alphabet.

You can reformulate the preceding query using a multivalued comparison predicate, as follows:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE LAST_NAME, FIRST_NAME > "Smith", "James" ;
```

Both formulations return this result:

```
EMP_ID       LAST_NAME  FIRST_NAME
------       ---------  ----------
  5361       Smythe     Roger
  9069       Smith      John
  9502       Smithson   Richard

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

The second formulation can result in a more efficient plan. For more information about efficiency, see Writing Efficient Predicates on page 3-15.

Using Multivalued Comparison Predicates in Context-Free Servers

Multivalued comparison predicates can be useful in context-free servers. For example, suppose that a server processes a batch of employees for each request from the requester. The server then positions to a row following the one that has a key value supplied by the requester.

A multivalued predicate can be matched with a group of key columns for use as a begin or end key predicate only if all of the key columns have the same ordering attribute: either all must be ascending, or all must be descending, as defined in the KEY specifications of CREATE TABLE and the CREATE INDEX statements.

Aggregate Functions in Predicates

Aggregate functions compute a value. They take a set of rows as their arguments and return a single row as their result.

Table 1-5. Aggregate Functions

Function	Description
AVG	Computes the average of a set of numbers
MAX	Determines a maximum value
MIN	Determines a minimum value
SUM	Computes the sum of a set of numbers
COUNT	Counts the number of rows that result from the query

Suppose that you want to find the sum of all salaries in the EMPLOYEE table. You can specify this query:

```
SELECT SUM(SALARY)
   FROM EMPLOYEE ;
```

The query returns this result:

```
(EXPR)
-----------
413210.00

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

The query returns the sum of salaries of all employees in EMPLOYEE.

Now suppose that you want to find the name of the employee who makes the maximum salary. You specify this query:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY =
      (SELECT MAX(SALARY) FROM EMPLOYEE) ;
```

The query returns this result:

```
LAST_NAME  FIRST_NAME  SALARY
---------  ----------  ------------
Nakagawa   Etsuro          72000.00

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

You can use the GROUP BY clause to group data for calculations. For example, suppose that you want to find the sum of the salaries for employees in departments 6400 and 7690:

```
SELECT DEPT_NUM, SUM (SALARY)
   FROM EMPLOYEE
   WHERE DEPT_NUM IN (6400, 7690)
   GROUP BY DEPT_NUM ;
```

The query returns this result:

```
DEPT_NUM     (EXPR)
--------     ----------------
6400                130300.00
7690                 81410.00

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

In a query that selects both aggregate and nonaggregate columns, you must include the GROUP BY clause on the nonaggregate columns. For more information about how to use aggregate functions, see the *SQL/MP Reference Manual*.

The MAX and MIN functions support collations for character string arguments that involve comparisons. For more information about collations, see the *SQL/MP Reference Manual*.

Using CASE Expressions

A CASE expression evaluates a set of conditions and returns a result that depends on which condition is true. These are some of the ways you can use CASE:

- Decoding values
- Evaluating multiple conditions
- Computing aggregates based on specific conditions
- Finding the highest value in a row
- Converting long, narrow tables into short, wide ones
- Ignoring the largest and smallest values in a set

Using CASE expressions, you can reduce the number of table scans, often to a single scan.

CASE is a conditional expression, not a statement. You can use a CASE expression in a subquery, but only in the predicate portion, not in the SELECT portion of the statement.

The following subsections provide examples and more information on some of the possible uses for CASE.

Decoding Values

You can use CASE expressions when you need to change the representation of data. Typically, a CASE expression is used to decode values so that the results are more meaningful. For example, you might want to store the value “M” but present the value “MALE”. Using CASE, you can do this without involving the host program. CASE expressions help you to avoid using a join against a lookup table and they can eliminate your need to write an application program or client tool code.

This an example of a CASE expression that decodes values:

```
SELECT LAST_NAME, FIRST_NAME,
   CASE MARITAL_STATUS
      WHEN 1 THEN "SINGLE"
      WHEN 2 THEN "MARRIED"
      WHEN 3 THEN "DIVORCED"
      ELSE NULL
   END
FROM EMPLOYEE ;
```

This is the result:

```
LAST_NAME       FIRST_NAME      (EXPR)
-------------   -------------   --------
CHENG           TINA            MARRIED
GONZALES        LINDA           SINGLE
LEBLANC         PIERRE          DIVORCED
PETSKI          STEVE           SINGLE

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

Although the marital statuses of the employees were encoded as 1, 2, and 3, the result contains marital statuses that are decoded.

Evaluating Multiple Conditions

When you use CASE with a search condition, SQL can evaluate multiple conditions in a single query, eliminating your need to write several queries that scan tables multiple times. Processing takes place within the SQL statement, reducing the coding required for the application program.

Suppose that you want to know the salaries of employees after you give a 10 percent raise to those in Department 9000 and a 12 percent raise to those in Department 1000. You can determine these salaries by using CASE with a search condition.

This example shows your employees with their current salaries:

```
SELECT LAST_NAME, FIRST_NAME, DEPTNUM, SALARY FROM EMPLOYEE ;
```

```
LAST_NAME       FIRST_NAME      DEPTNUM     SALARY
-------------   -------------   ------      ----------
CHENG           TINA             1000         65000.00
GONZALES        LINDA            9000         75500.00
LEBLANC         PIERRE           9000         37000.00
PETSKI          STEVE            3500         50000.00

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

When you perform a query using a CASE expression with a search condition, you can see each employee’s new salary:

```
SELECT LAST_NAME, FIRST_NAME, DEPTNUM,
  CASE
     WHEN DEPTNUM  = "9000"
        THEN SALARY * 1.10
     WHEN DEPTNUM = "1000"
        THEN SALARY * 1.12
     ELSE SALARY
   END
 FROM EMPLOYEE;

LAST_NAME       FIRST_NAME      DEPTNUM     (EXPR)
-------------   -------------   -------     ----------
CHENG           TINA            1000          72800.00
GONZALES        LINDA           9000          83050.00
LEBLANC         PIERRE          9000          40700.00
PETSKI           STEVE            3500           50000.00

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

Linda Gonzales and Pierre LeBlanc, both in Department 9000, received 10 percent raises, and Tina Cheng, in Department 1000, received a 12 percent raise. Steve Petski, who is in neither Department 9000 nor Department 1000, received no raise.

Computing Aggregates Based on Specific Conditions

You can use CASE to produce a report that contains aggregate values calculated for groups, which are each selected using different criteria. By using CASE, you eliminate the need to create a temporary table and insert data into it.

For example, suppose that a report counts the number of employees, by department, whose salaries are in these ranges:

Range 1	< 20000
Range 2	< 50000
Range 3	< 200000

Using CASE, you aggregate the necessary information with a single scan of the EMPLOYEE table:

```
SET LIST_COUNT 0 ;
SELECT DEPTNUM,
   SUM (CASE
      WHEN SALARY < 20000 THEN 1
      ELSE 0
      END),
   SUM (CASE
      WHEN SALARY < 50000 THEN 1
      ELSE 0
      END),
   SUM (CASE
      WHEN SALARY < 200000 THEN 1
      ELSE 0
      END)
FROM PERSNL.EMPLOYEE
GROUP BY DEPTNUM ;

DETAIL DEPTNUM,
        COL 2 AS I12 HEADING "SAL. < 20000",
        COL 3 AS I12 HEADING "SAL. < 50000",
        COL 4 AS I12 HEADING "SAL. < 200000" ;
LIST ALL ;

DEPTNUM  SAL. < 20000  SAL. < 50000  SAL. < 200000
-------  ------------  ------------  -------------

   1000             1             3              5
   1500             1             3              4
   2000             0             4              5
    .               .             .              .
    .               .             .              .

   4000             1             9             15
   9000             0             1              2

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

For information on SUM, see Aggregate Functions in Predicates on page 1-41. For information on GROUP BY, see The GROUP BY Clause on page 1-7.

Another example that uses CASE with aggregates: Suppose that you have a table with rows that contain each employee’s name, age, department, and number of cars. The table contains no nulls and all numeric values for the number of cars. The primary key is NAME.

These are the values in the table:

```
NAME      AGE     DEPT         CARS
--------  ------  -----------  ------

BROWN         30           50       1
CHANG         38           50       2
GONZALES      22           50       1
HO
38           50       2
KAPOOR        28           50       1
LEBLANC       25           50       0
PETSKI        23           40       4
YAMASAKI      24           40       2

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

You need to compute the number of employees who have one car, two cars, and so on. Using CASE and SUM, you can get the result:

```
set list_count 0 ;
select SUM(CASE when cars = 0 then 1 else 0 END),
       SUM(CASE when cars = 1 then 1 else 0 END),
       SUM(CASE when cars between 2 and 3 then 1 else 0 END),
       SUM(CASE when cars > 3 then 1 else 0 END)
  from emp;
detail col 1 as I1 heading “None”,
       col 2 as I1 heading “One”,
       col 3 as I1 heading “Two or Three”,
       col 4 as I1 heading “More Than Three”;
list all;

None  One  Two or Three  More Than Three
----  ---  ------------  ---------------

   1    3             3                1
```

See CASE With Aggregates on page 6-34 for the EXPLAIN plan. For information on SUM, see Aggregate Functions in Predicates on page 1-41.

Finding the Highest Value in a Row

Suppose that you need to select the largest (or smallest) value from multiple columns in each row of a table. For example, each row in a table of students contains the student’s name and scholastic aptitude test scores (SATs) for the past two years. All SAT scores are numeric, and none are null. The primary key is NAME.

The values in the table are as follows:

```
NAME      SAT1         SAT2
--------  -----------  -----------

BROWN             480          520
BYSTROM           510          715
CHUNG             725          650
GOMEZ             780          610
HO                715          680
MCLAIN            600          520
MINSKY            400          510
PONG              790          720
SCHMIDT           580          590
SMITH             550          630

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

You want to list the name and the highest of the two scores for each student. You can use CASE to produce the result:

```
set list_count 0 ;
select name, CASE
               when sat1 >= sat2 then sat1
               else sat2
             END
  from scores;
detail name,
       col 2 as I3 heading "Highest Score";
list all;

NAME      Highest Score
--------  -------------

BROWN               520
BYSTROM             715
CHUNG               725
GOMEZ               780
HO                  715
MCLAIN              600
MINSKY              510
PONG                790
SCHMIDT             590
SMITH               630

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

For the EXPLAIN plan, see CASE for Finding the Highest Value in a Row on page 6-35.

Converting Long, Narrow Tables Into Short, Wide Ones

You might find that you can manipulate data easily when you use long, narrow tables, but for reports, you may prefer short, wide tables.

Suppose that you have a table that contains each salesperson’s name, a month number, and a bonus amount for that month. No amounts are null and all are numeric. The primary key is NAME and MONTH. Twelve rows represent the amounts for a year for one salesperson, as in this example:

```
NAME      MONTH   AMOUNT
--------  ------  --------------------

CHIN           1                   200
CHIN           2                     0
CHIN           3                     0
CHIN           4                     0
CHIN           5                   400
CHIN           6                   120
CHIN           7                    80
CHIN           8                   220
CHIN           9                   115
CHIN          10                   130
CHIN          11                    75
CHIN          12                   105
KAPOOR         1                   200
KAPOOR         2                     0
KAPOOR         3                   150
KAPOOR         4                   300
KAPOOR         5                     0
KAPOOR         6                   110
KAPOOR         7                    20
KAPOOR         8                   130
KAPOOR         9                     0
KAPOOR        10                    50
KAPOOR        11                   200
KAPOOR        12                   110
KLEIN          1                   100
KLEIN          2                    60
KLEIN          3                   220
KLEIN          4                   230
KLEIN          5                   210
KLEIN          6                    40
KLEIN          7                   120
KLEIN          8                   140
KLEIN          9                    20
KLEIN         10                   150
KLEIN         11                     0
KLEIN         12                    60
--- 36 row(s) selected.
```

To produce a report with a row for each salesperson's name and corresponding bonus amounts for months 1 through 12, you can use CASE with GROUP BY:

```
set list_count 0 ;
select name,
       SUM(CASE when month =  1 then amount else 0 END),
       SUM(CASE when month =  2 then amount else 0 END),
       SUM(CASE when month =  3 then amount else 0 END),
       SUM(CASE when month =  4 then amount else 0 END),
       SUM(CASE when month =  5 then amount else 0 END),
       SUM(CASE when month =  6 then amount else 0 END),
       SUM(CASE when month =  7 then amount else 0 END),
       SUM(CASE when month =  8 then amount else 0 END),
       SUM(CASE when month =  9 then amount else 0 END),
       SUM(CASE when month = 10 then amount else 0 END),
       SUM(CASE when month = 11 then amount else 0 END),
       SUM(CASE when month = 12 then amount else 0 END)
  from bonus
 group by name;
detail name,
       col 2 as I3 heading "JAN",
       col 3 as I3 heading "FEB",
       col 4 as I3 heading "MAR",
       col 5 as I3 heading "APR",
       col 6 as I3 heading "MAY",
       col 7 as I3 heading "JUN",
       col 8 as I3 heading "JUL",
       col 9 as I3 heading "AUG",
       col 10 as I3 heading "SEP",
       col 11 as I3 heading "OCT",
       col 12 as I3 heading "NOV",
       col 13 as I3 heading "DEC";
list all;
NAME      JAN  FEB  MAR  APR  MAY  JUN  JUL  AUG  SEP  OCT  NOV  DEC
--------  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---  ---

CHIN      200    0    0    0  400  120   80  220  115  130   75  105
KAPOOR    200    0  150  300    0  110   20  130    0   50  200  110
KLEIN     100   60  220  230  210   40  120  140   20  150    0   60

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

If required, you could also calculate totals for each month and a yearly total for each salesperson.

For the EXPLAIN plan for the query in the example, see CASE for Converting Long, Narrow Tables Into Short, Wide Ones on page 6-36. For information on SUM, see Aggregate Functions in Predicates on page 1-41. For information on GROUP BY, see The GROUP BY Clause on page 1-7.

Ignoring the Largest and Smallest Values in a Set

You can use CASE with SUM to retrieve the values in the table used in the previous example and ignore the largest and smallest values in the table:

```
set list_count 0 ;
select x.value
  from data x, data y
 group by x.value
 having SUM (CASE when y.value <= x.value then 1 else 0 END) > 1
   AND SUM (CASE when y.value >= x.value then 1 else 0 END) > 1;
detail col 1 as I4 heading "Value";
list all;

Value
-----

   10
   12
   20
   24
   38
   40
   48
   50
   52
   66
   67
   80

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

For the EXPLAIN plan, see CASE for Ignoring the Largest and Smallest Values in a Set on page 6-38. For information on SUM, see Aggregate Functions in Predicates on page 1-41.

Combining Data From More Than One Table

You can join two tables to form a new table. When tables are joined, each new row is formed by concatenating two rows, one from each of the original tables. This type of query is called a join query.

To join tables, name the tables (or views) in the FROM clause of a SELECT statement. When tables are joined, each new row is formed by concatenating the rows from each of the original tables. The values in the paired rows must satisfy the join condition. For example, this query joins two tables together on the column DEPT_NO:

```
SELECT EMPLOYEE_NAME, DEPT_NAME
  FROM EMPLOYEE, DEPT
  WHERE EMPLOYEE.DEPT_NO = DEPT.DEPT_NO ;
```

The predicate EMPLOYEE.DEPT_NO = DEPT.DEPT_NO is called a join predicate. A join query contains zero or more join predicates that identify and compare columns from each table. The join predicate determines whether or not the columns satisfy a given search condition. Defining Predicates on page 1-30 describes how to use predicates to narrow the range of searching. The same principles apply to join queries.

If columns satisfy the condition, the join operation selects the desired columns, concatenates the rows, and returns them to the result table.

If you specify multiple tables in the FROM clause but do not specify search conditions, SQL forms a Cartesian product (or cross product) by concatenating, in turn, each row of each table with every other row of every other table.

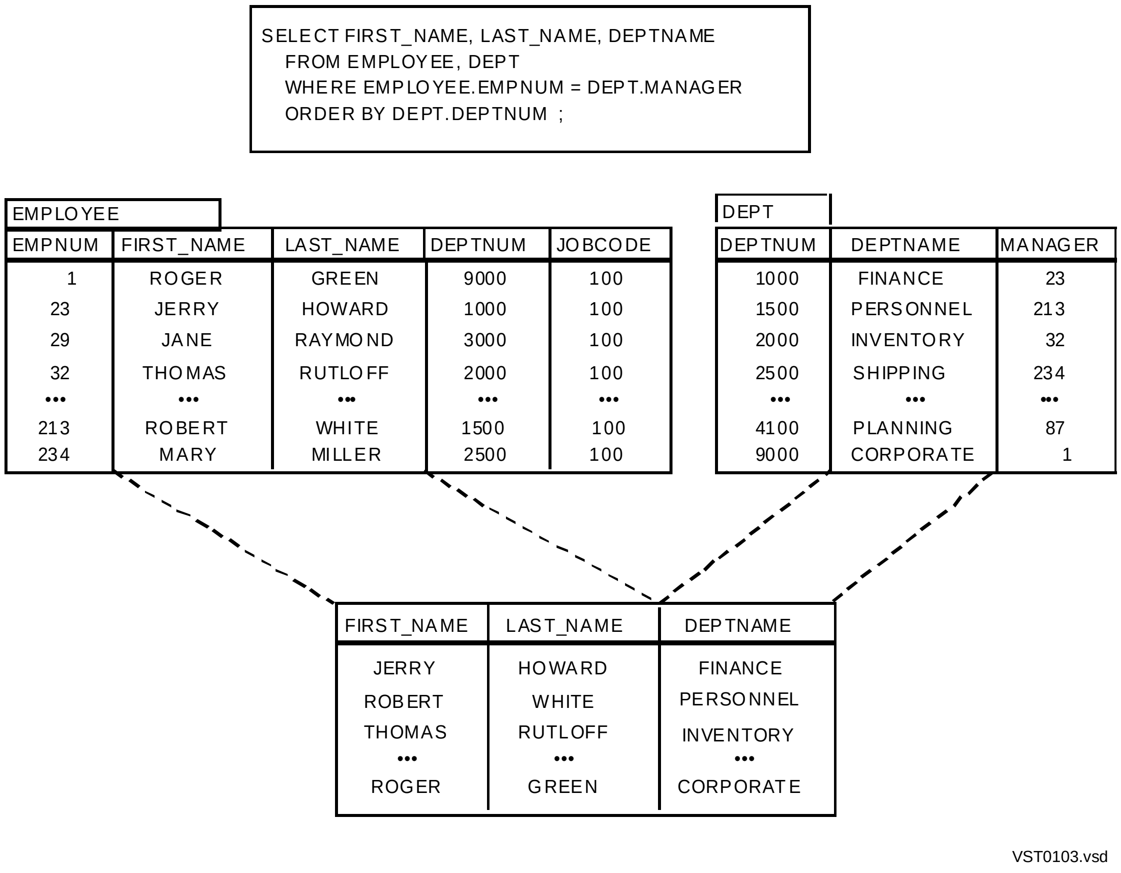
A Cartesian product involving two tables, one with M rows and the other with N rows, is of size M x N. Therefore, the use of join predicates is likely to reduce the size of the result. Furthermore, use of join predicates can reduce the amount of work done by SQL to produce the result. For more information, see Section 3, Improving Query Performance Through Query Design.

In the query in Figure 1-3 on page 1-53, the joining columns are EMPLOYEE.EMPNUM and DEPT.MANAGER. The predicate is:

```
WHERE EMPLOYEE.EMPNUM = DEPT.MANAGER
```

SQL concatenates those rows from the two tables, EMPLOYEE and DEPT, that have the same values in EMPNUM and MANAGER, respectively.

Figure 1-3. Selecting From Two Tables



```
SELECT FIRST_NAME, LAST_NAME, DEPTNAME
   FROM EMPLOYEE, DEPT
   WHERE EMPLOYEE.EMPNUM = DEPT.MANAGER
   ORDER BY DEPT.DEPTNUM ;
```

EMPLOYEE

EMPNUM	FIRST_NAME	LAST_NAME	DEPTNUM	JOBCODE
1	ROGER	GREEN	9000	100
23	JERRY	HOWARD	1000	100
29	JANE	RAYMOND	3000	100
32	THOMAS	RUTLOFF	2000	100
•••	•••	•••	•••	•••
213	ROBERT	WHITE	1500	100
234	MARY	MILLER	2500	100

DEPT

DEPTNUM	DEPTNAME	MANAGER
1000	FINANCE	23
1500	PERSONNEL	213
2000	INVENTORY	32
2500	SHIPPING	234
•••	•••	•••
4100	PLANNING	87
9000	CORPORATE	1

FIRST_NAME	LAST_NAME	DEPTNAME
JERRY	HOWARD	FINANCE
ROBERT	WHITE	PERSONNEL
THOMAS	RUTLOFF	INVENTORY
•••	•••	•••
ROGER	GREEN	CORPORATE

Types of Join Queries

You can perform two types of join operations using SQL:

- An inner join operation returns all rows that satisfy a given search condition. The inner join is useful for reporting information that satisfies a particular set of requirements—for example, salespersons who have booked orders.
- A left join operation, or left outer join, returns all rows that satisfy a given search condition plus those rows in the left table (listed left of the JOIN keyword) that fail to satisfy the condition—for example, salespersons who have not booked any orders. These rows, the exceptions to the rule represented by the join condition, can provide useful information for exception reports.

Do not confuse inner and left join operations with join strategies, such as sort merge, key-sequenced merge, nested, and hash. Join strategies refer to the mechanisms SQL uses to perform joins. These strategies are discussed in Section 3, Improving Query Performance Through Query Design.

Note. Certain combinations of left joins, inner joins, and the UNION operator are not valid in a single SELECT statement. For more information, see the *SQL/MP Reference Manual*.

Example 1-9 shows the sample tables for several of the join examples that follow.

Example 1-9. Sample Tables for Join Examples

```
SALESEMP Table

EMP_NUM    EMP_NAME       REG_NUM     MGR_NUM
-------    --------       -------     -------
   2703    MORRISON, J.      7600        2705
   2705    HENNESSY, A.      7600        6554
   2906    NAKAGAWA, E.      6400        6554
   3598    CHU, F.           6470        2906
   4096     CHOW, J.           6420        3598
```

```
REGION Table

REG_NUM     REG_NAME       REG_HQLOC
--------    --------       ---------
    6400    JAPAN                900
    6420    HONG KONG            920
    6470    TAIWAN               970
    7600     USA                    100
```

```
ORDERS Table

ORD_NUM    CUST_NUM       BOOKED_BY
-------    ---------      ---------
     12    729                 3598
     57    283                 2705
     77    1064                  2906
```

Inner Join

This query formulates an inner join of the tables SALESEMP and ORDERS:

```
SELECT S.EMP_NUM, S.EMP_NAME, O.ORD_NUM
   FROM SALESEMP S, ORDERS O
   WHERE S.EMP_NUM = O.BOOKED_BY ;
```

You can also write this query using the INNER JOIN keywords as follows:

```
SELECT S.EMP_NUM, S.EMP_NAME, O.ORD_NUM
   FROM SALESEMP S INNER JOIN ORDERS O
   ON S.EMP_NUM = O.BOOKED_BY ;
```

Both queries produce the same result. The result contains only those rows that satisfy the join predicate given in the WHERE clause in the first query and the ON clause in the second query, as follows:

```
EMP_NUM     EMP_NAME     ORD_NUM
-------     --------     -------
   2705     HENNESSY, A.      57
   2906     NAKAGAWA, E.      77
   3598     CHU, F.           12

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

The order of the table names in the queries does not influence join order (which table is the outer table and which is the inner table).

An inner join discards all rows that do not satisfy the given search condition, so information about salespersons who have not booked any orders is missing from the result of this query. If you want such information, you can specify a left join operation.

Left Join

The left join, or left outer join, returns all rows that satisfy the given predicates. In addition, it returns all rows in the left table that fail to satisfy the join predicates; these rows, however, have information missing in all of the columns that correspond to the right table. In other words, the result contains information from the left table, regardless of whether matching right table rows exist.

This query formulates a left join operation of the tables SALESEMP and ORDERS:

```
SELECT S.EMP_NUM, S.EMP_NAME, O.ORD_NUM
   FROM SALESEMP S LEFT JOIN ORDERS O
   ON S.EMP_NUM = O.BOOKED_BY ;
```

The query returns this result:

```
EMP_NUM     EMP_NAME     ORD_NUM
-------     --------     -------
   2703     MORRISON, J.       ?
   2705     HENNE0SSY, A.      57
   2906     NAKAGAWA, E.      77
   3598     CHU, F.           12
   4096     CHOW, J.           ?

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

The left join shows three employees who booked orders and also shows the employees who did not book any orders, indicated by a question mark (?) in the ORD_NUM column: J. Morrison (employee number 2703) and J. Chow (employee number 4096). The question mark indicates a null value, denoting unknown or null information. These are rows that failed to satisfy the search condition.

If you use host variables to store results in a program, the program must handle the null values that can result from left join operations.

Notice also that the ON clause specifies the join conditions. In a left outer join query, the WHERE clause applies restrictions on the result of the join operation. To list employees who do not have departments, you could check for the occurrence of a null value in the DEPT_NUM column of the DEPT table as follows:

```
SELECT S.EMP_NAME
   FROM SALESEMP S LEFT JOIN
    DEPT DT
    ON S.DEPT_NUM = DT.DEPT_NUM
   WHERE DT.DEPT_NUM IS NULL;
```

The IS NULL predicate applies to the DEPTNUM column of the DEPT table because it appears in the join predicate and belongs to the table that is not to be preserved. Because of these two reasons, a null value is guaranteed to appear in the DEPTNUM columns of the result whenever SQL finds that, for a given EMPLOYEE.DEPTNUM value, a matching DEPT.DEPTNUM value does not exist.

How a Left Join Preserves Data

SQL preserves data from the table on the left of the keywords LEFT JOIN. So, in the preceding example, data from the SALESEMP table is preserved.

To preserve rows from the SALESEMP table, SQL extends each row that does not satisfy the search condition with as many null values as there are columns from the table on the right in the result. Such a row is called a null-augmented row. The row is added to the result after extending or augmenting it with null values.

Note that the null values only appear in those columns that belong to the table that appears on the right of the keywords LEFT JOIN (in the preceding example, the ORDERS table).

Predicates that involve columns from a table that appears on the right of the keywords LEFT JOIN are evaluated after the left join is performed: that is, only after null augmentation is performed.

Using a Left Join to Show Hierarchical Relationships

You can use the left join operation to display hierarchical relationships among data. These examples use the sample database shown in Example 1-10.

Example 1-10. Sample Tables For Hierarchical Relationship Examples

```
DEPT Table

DEPTNO    DEPTNAME         MANAGER    RPTDEPT     LOCATION
------    -------------    -------    -------     ------------
   101    Accounting            23       9000     CHICAGO
   102    Maintenance          213       1000     CHICAGO
   103    Personnel             32       9000     LOS ANGELES

EMPLOYEE Table

EMPNO  EMP_FNAME     EMP_LNAME      DEPTNO  JOBCODE  SALARY
-----  ------------- ------------- ------  -------  -----------
   10  John          Smith            101      100    175500.00
   20  Tina          Gray             102      100    137000.10

DEPD Table

DEPDNO      DEP_FNAME      DEP_LNAME           EMPNO
------      -------------  ------------------  -----
   521      William        Gray                   20
   522      Gwendolyn      Gray                   20
```

Note. The examples in this subsection associate correlation names—DT, E, and DD—with the table names DEPT, EMPLOYEE, and DEPD.

This query requests information about employees and their dependents from three tables: DEPT, EMPLOYEE, and DEPD:

```
SELECT DT.DEPTNAME, E.EMP_LNAME, E.EMP_FNAME, DD.DEP_FNAME
  FROM DEPT DT, EMPLOYEE E, DEPD DD
  WHERE DT.DEPTNO = E.DEPTNO AND E.EMPNO = DD.EMPNO ;
```

Because this query specifies an inner join operation, the result does not include departments that have no employees or employees who have no dependents:

```
DEPTNAME      EMP_LNAME        EMP_FNAME      DEP_FNAME
------------  ---------------  -------------  ----------
Maintenance   Gray             Tina           William
Maintenance   Gray             Tina           Gwendolyn

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

The next query specifies a left join operation. All data is preserved from the DEPT table:

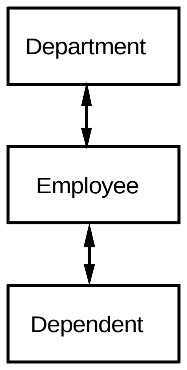
```
SELECT DT.DEPTNAME, E.EMP_LNAME, E.EMP_FNAME, DD.DEP_FNAME
  FROM DEPT DT
    LEFT JOIN EMPLOYEE E ON DT.DEPTNO = E.DEPTNO
    LEFT JOIN DEPD DD ON E.EMPNO = DD.EMPNO ;
```

Now, the Personnel department, which has no employees, and the Accounting department, which has one employee with no dependents, appear in the result table:

```
DEPTNAME        EMP_LNAME       EMP_FNAME        DEP_FNAME
------------    --------------  ---------------  -----------
Accounting      Smith           John             ?
Maintenance     Gray            Tina             William
Maintenance     Gray            Tina             Gwendolyn
Personnel       ?               ?                ?

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

This left join operation displays this hierarchical relationship:



010

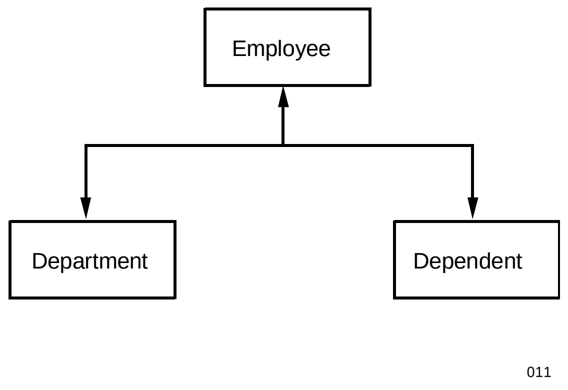
For another example of a left join operation, consider this query and its result table. All data is preserved from the EMPLOYEE table. Note that the department without an employee is not shown as this is not the relation requested:

```
SELECT DT.DEPTNAME, E.EMP_LNAME, E.EMP_FNAME, DD.DEP_FNAME
   FROM EMPLOYEE E
     LEFT JOIN DEPT DT ON E.DEPTNO = DT.DEPTNO
     LEFT JOIN DEPD DD ON E.EMPNO = DD.EMPNO ;


DEPTNAME      EMP_LNAME     EMP_FNAME      DEP_FNAME
------------  ------------  -------------  ------------
Accounting    Smith        John           ?
Maintenance   Gray         Tina           William
Maintenance   Gray         Tina           Gwendolyn

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

In this case, the left join operation displays this hierarchical relationship:



Restrictions on Join Queries

There are a few restrictions on the use of the join operation, as follows:

- A shorthand view based on any join cannot be specified as the right table of a left join.
- A shorthand view whose definition is based on a union of SELECT commands cannot participate in another join operation.

For additional information about join operations, see the *SQL/MP Reference Manual.*

Using Views With Joins

For frequently performed join operations, the definition of a view can hide the join operation from the user and make the columns simpler to access.

The ON Clause and the WHERE Clause in Join Queries

A query can combine inner and left join operations in the same FROM clause. Therefore, you must indicate which join conditions apply to which join operations, as follows:

- Specify join conditions for a left join in the ON clause.
- Specify join conditions for an inner join or between tables participating in different left joins in the WHERE clause.

Consider this query:

```
SELECT S.EMP_NUM, S.EMP_NAME, O.ORD_NUM, R.REG_NAME
FROM REGION R,
  SALESEMP S LEFT JOIN ORDERS O
     ON S.EMP_NUM = O.BOOKED_BY
     AND S.EMP_NUM < 2800
  WHERE S.REG_NUM = R.REG_NUM
     AND S.REG_NUM IN (6400, 7600) ;
```

SQL first performs an inner join of the REGION table with the SALESEMP table. The result table of this inner join operation is then left joined with the ORDERS table.

Because SALESEMP appears on the left of the keywords LEFT JOIN, SALESEMP is the table that is preserved from the left join operation.

The query returns this result:

```
EMP_NUM     EMP_NAME      ORD_NUM    REG_NAME
-------     --------      -------    ------------
   2703     MORRISON, J.        ?    USA
   2705     HENNESSY, A.       57    USA

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

The ON Clause

For each occurrence of the LEFT JOIN keywords in the FROM clause, there is a corresponding ON clause.

The predicates in the ON clause specify the conditions for the left join evaluation. These conditions determine whether rows can be joined together or whether the rows from the preserved table are preserved through null augmentation.

In the former example, there was one left join operation:

```
SALESEMP S LEFT JOIN ORDERS O
  ON S.EMP_NUM = O.BOOKED_BY
  AND S.EMP_NUM < 2800
```

The table preserved in each left join can, in turn, be inner-joined or left-joined with another table.

The WHERE Clause

Predicates in the WHERE clause specify join conditions and define restrictions on individual tables named in the FROM clause.

In the previous example, the WHERE clause joins the REGION table with the SALESEMP table and specifies a restriction on the SALESEMP table:

```
WHERE S.REG_NUM = R.REG_NUM
   AND S.REG_NUM IN (6400, 7600)
```

In a left join query, predicates from the WHERE clause that apply to a table participating in a left join operation are evaluated as follows:

- If a predicate contains columns from the right table, the predicate is evaluated after the left join operation: that is, only after null augmentation is performed.
- Otherwise, the predicate can be evaluated before or after the left join operation.

For example, consider this query:

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.DEPTNUM
   FROM EMPLOYEE E LEFT JOIN DEPT DT
   ON E.DEPTNO = DT.DEPTNO
   WHERE E.LAST_NAME = "SPENCER"
   AND DT.MANAGER = 23;
```

Table E is preserved, table DT is not. The predicates, therefore, can be evaluated as follows:

- The predicate E.LAST_NAME = SPENCER can be evaluated anytime after SQL has examined a row from E.
- The predicate DT.MANAGER = 23, however, must be evaluated only after null augmentation is performed. This restriction exists because columns like DT.MANAGER might contain a null value whenever a row from E is preserved.

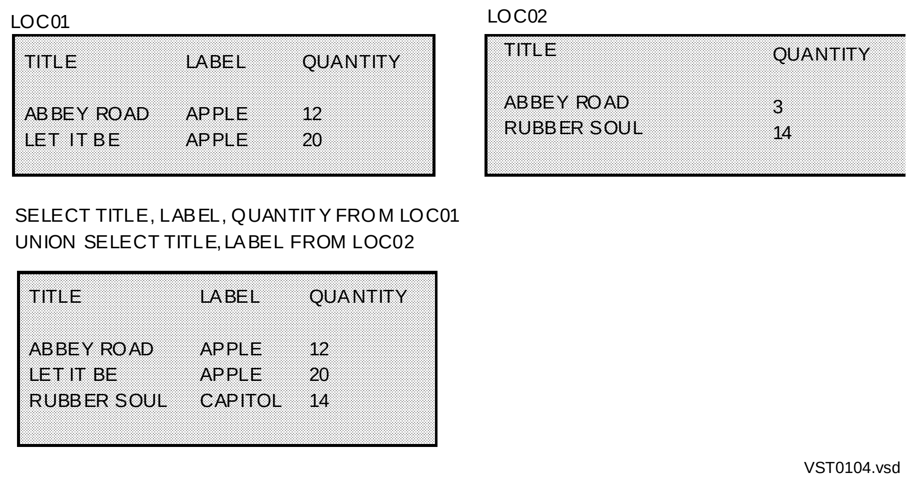
Using the UNION Operator

SQL supports two types of union operations:

- A UNION operation combines two tables whose respective column data types are comparable and automatically deletes duplicate rows from the result. Thus, a UNION of two select operations, one on TABLE1 and one on TABLE2, is the set of all distinct rows returned from the first select (TABLE1) and the second select (TABLE2).
- A UNION ALL operation works the same as the UNION operation, except that UNION ALL preserves duplicate rows in the result. If you know that no duplicates exist in your data—or if you are willing to handle duplicate rows within your application—the ALL option can avoid a sort operation.

Figure 1-4 shows how a union of two selects, one on LOC01 and one on LOC02, could be useful for determining all record titles available at two different store locations.

Figure 1-4. UNION of Two Tables



LOC01

TITLE	LABEL	QUANTITY
ABBEY ROAD	APPLE	12
LET IT BE	APPLE	20

LOC02

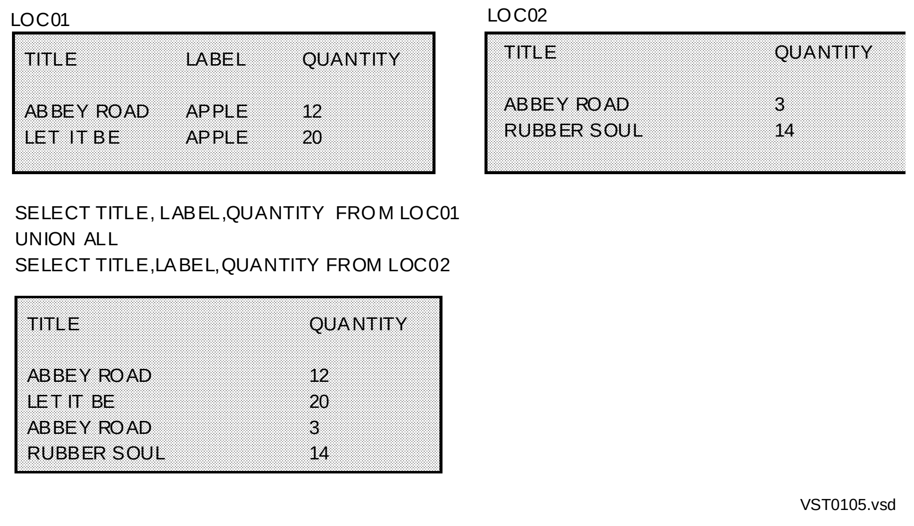
TITLE	QUANTITY
ABBEY ROAD	3
RUBBER SOUL	14

```
SELECT TITLE, LABEL, QUANTITY FROM LOC01
UNION SELECT TITLE, LABEL FROM LOC02
```

TITLE	LABEL	QUANTITY
ABBEY ROAD	APPLE	12
LET IT BE	APPLE	20
RUBBER SOUL	CAPITOL	14

Figure 1-5 shows how a UNION ALL of two selects, one on LOC01 and one on LOC02, could be useful for evaluating inventory at both stores.

Figure 1-5. UNION ALL of Two Tables



LOC01

TITLE	LABEL	QUANTITY
ABBEY ROAD	APPLE	12
LET IT BE	APPLE	20

LOC02

TITLE	QUANTITY
ABBEY ROAD	3
RUBBER SOUL	14

```
SELECT TITLE, LABEL,QUANTITY  FROM LOC01
UNION ALL
SELECT TITLE,LABEL,QUANTITY FROM LOC02
```

TITLE	QUANTITY
ABBEY ROAD	12
LET IT BE	20
ABBEY ROAD	3
RUBBER SOUL	14

Note that the UNION ALL operation does not eliminate duplicate rows. Neither UNION operation orders the results. To order results, use the ORDER BY clause, as described in this subsection.

To specify a UNION operation, these rules apply:

- Both select lists must specify the same number of columns.
- Columns in corresponding positions must have compatible data types. (For information on compatible data types, see the *SQL/MP Reference Manual*.)

You can specify a UNION operator in these instances where a single SELECT statement is allowed:

- A shorthand view definition
- A subquery
- An INSERT operation through a query
- A nonupdatable cursor statement

Note. A shorthand view whose definition involves a UNION cannot participate in a join operation. A SELECT on the view cannot specify a GROUP BY clause, a HAVING clause, or aggregate functions on any view column. A shorthand view cannot be updated, regardless of whether the UNION operator is used.

Additional guidelines for using the UNION operator follow. In this discussion, catalog tables are used to demonstrate the UNION operator. Catalog tables are candidates for a UNION operation because different instances of the catalog tables in different catalogs have the same attributes for their columns.

To get a list of all programs that are described in the application catalogs from $VOL1 and $VOL2 on your system, you can specify this query:

```
  SELECT A.PROGRAMNAME
    FROM $VOL1.PROGCAT.PROGRAMS A
UNION
  SELECT B.PROGRAMNAME
    FROM $VOL2.PROGCAT.PROGRAMS B ;
```

For complete information about UNION and UNION ALL, see the *SQL/MP Reference Manual*.

ORDER BY Clause With UNION Operator

You can use the ORDER BY clause to order the result of a UNION operation. These restrictions apply, however:

- The ORDER BY clause must follow the last SELECT statement and any options associated with that individual SELECT statement. You cannot use parentheses to associate an ORDER BY clause with either SELECT statement.
- The ORDER BY clause must contain one of these:
    - The names of the columns explicitly referenced outside a function or an expression in the select list

  - An integer that indicates the ordinal position of a column, a function, or an expression in the select list

This statement shows incorrect use of the ORDER BY clause and UNION operator:

```
  SELECT A.PROGRAMNAME
    FROM $VOL1.PROGCAT.PROGRAMS A
    ORDER BY A.PROGRAMNAME
UNION
  SELECT B.PROGRAMNAME
    FROM $VOL2.PROGCAT.PROGRAMS B ;
```

Instead, you should formulate the preceding query as:

```
  SELECT A.PROGRAMNAME
    FROM $VOL1.PROGCAT.PROGRAMS A
UNION
  SELECT B.PROGRAMNAME
    FROM $VOL2.PROGCAT.PROGRAMS B
ORDER BY A.PROGRAMNAME ;
```

Note. When requesting UNION operations, the choice of access option can affect performance. For more information, see Specifying Access Option and Lock Characteristics on page 4-16.

GROUP BY and HAVING Clauses With UNION Operator

You cannot use the GROUP BY and HAVING clauses to form groups in the result of a UNION operation.

Unlike the ORDER BY clause, which can define an ordering on the result of the UNION operation, the GROUP BY and HAVING clauses are associated only with the SELECT statement in which the clauses appear. Consequently, the groups are visible only in the result table of the SELECT statement that contains the clauses and not in the result of the UNION operation.

Using Collations With the UNION Operator

The UNION operator is affected by collations in associated tables as follows:

- Comparisons might occur during the computation of the result
- Because the output of the UNION operation is a table, the default collation for its columns must be specified

For example, if column A in table X has collation FINNISH and column B in table Y has collation ITALIAN and you attempt to SELECT A FROM X UNION Y, then column A has no collation. Instead, you could use this statement:

```
SELECT A COLLATE FRENCH FROM X
  UNION
SELECT A COLLATE FRENCH FROM Y;
```

Developing Interactive Multistep Queries

This subsection describes techniques for developing multistep queries to select data for reports. You can use multistep queries as follows:

- To apply aggregate functions to multiple levels of groups
- To compute what percent the current row value is of all rows

These techniques use temporary tables to select data based on more than one query. To create a table, you use the CREATE TABLE statement. You must have access to a catalog to define a table, or you must have the authority to create your own catalog. For information about the requirements for creating tables, see the CREATE TABLE statement in the *SQL/MP Reference Manual*.

For information about the report writer commands used in these examples (SET LIST_COUNT, NAME, DETAIL, BREAK ON, LIST), see the *SQL/MP Reference Manual* and the *SQL/MP Report Writer Guide*.

Multilevel Group Aggregates

The GROUP BY Clause on page 1-7 describes how to apply aggregate functions to groups of rows. To apply aggregate functions to multiple levels of groups, you must specify more than one query and use temporary tables.

For example, suppose that you want to report the average salary for each department and, within each department, for each job classification. You must follow these steps:

1. Create a temporary table to contain the department number and average salary for each department. Use the INVOKE statement to determine the data type for the DEPTNUM column of the DEPT table. Then use CREATE TABLE to create the temporary table:

```
>> CREATE TABLE DEPTAVG (
+>    DEPTNUM      NUMERIC (4) UNSIGNED    NO DEFAULT,
+>    AVGSAL       NUMERIC (6) UNSIGNED    NO DEFAULT )
+>  CATALOG TEMPTABS ;
```

2. Insert the department number and average salary for each department in the DEPTAVG table:

```
>> INSERT INTO DEPTAVG
+>     (SELECT DEPTNUM, AVG(SALARY)
+>      FROM PERSNL.EMPLOYEE
+>      GROUP BY DEPTNUM) ;
```

3. Create another temporary table to contain the job code and average salary for each type of job within a department:

```
>> CREATE TABLE JOBAVG (
+>    DEPTNUM      NUMERIC (4) UNSIGNED    NO DEFAULT,
+>    JOBCODE      NUMERIC (4) UNSIGNED    NO DEFAULT,
+>    AVGSAL       NUMERIC (6) UNSIGNED    NO DEFAULT )
+>  CATALOG TEMPTABS ;
```

4. Insert the department number, job code, and average salary for each job in each department in the JOBAVG table:

```
>> INSERT INTO JOBAVG
+>     (SELECT DEPTNUM, JOBCODE, AVG(SALARY)
+>      FROM PERSNL.EMPLOYEE
+>      GROUP BY DEPTNUM, JOBCODE) ;
```

5. Join the temporary tables and select the report information:

```
>> SET LIST_COUNT 0 ;
>> SELECT D.DEPTNUM, JOBCODE, D.AVGSAL, J.AVGSAL
+>     FROM DEPTAVG D, JOBAVG J
+>     WHERE D.DEPTNUM = J.DEPTNUM
+>     ORDER BY D.DEPTNUM;
S> NAME COL 3 DEPT_AVGSAL ;
S> NAME COL 4 JOB_AVGSAL ;
S> DETAIL DEPTNUM, DEPT_AVGSAL, JOBCODE, JOB_AVGSAL ;
S> BREAK ON DEPTNUM, DEPT_AVGSAL ;
S> LIST N 8 ;

DEPTNUM  DEPT_AVGSAL  JOBCODE  JOB_AVGSAL
-------  -----------  -------  ----------
   1000        52000      100      137000
                          500       34666
                          900       19000
   1500        41250      100       90000
                          600       29000
                          900       17000
   2000        50000      100       13800
                          200       24000
S>
```

The BREAK ON command suppresses printing of the same department number and salary average in multiple lines.

You can drop the tables or purge the data and reuse the tables in future reports.

Computing Row Value as a Percent of All Row Values

You can also use multistep queries to compute what percent the current row value is of all row values. For example, these report displays the percent that an individual’s salary is of all salaries in a department. The report is produced with these steps:

1. Create a temporary table to contain the average salary for each department:

```
>> CREATE TABLE TEMPTABS.AVGTEMP (
+>    DEPTNUM      NUMERIC (4) UNSIGNED  NOT NULL,
+>    AVGSAL       NUMERIC (6) UNSIGNED  NOT NULL)
+>  CATALOG TEMPTABS ;
```

2. Insert the department number and average salary into the temporary table:

```
>> INSERT INTO TEMPTABS.AVGTEMP
+>     (SELECT DEPTNUM, AVG(SALARY) FROM PERSNL.EMPLOYEE
+>      GROUP BY DEPTNUM) ;
```

3. Select the information for the report. Include an expression in the select list to compute the percent of the department average:

```
>> SELECT E.DEPTNUM, EMPNUM, LAST_NAME, SALARY,
+>         SALARY/AVGSAL*100.00
+>  FROM PERSNL.EMPLOYEE E, TEMPTABS.AVGTEMP A
+>  WHERE E.DEPTNUM = A.DEPTNUM;
S>  DETAIL DEPTNUM, EMPNUM, LAST_NAME,
+>         SALARY AS F10.2,
+>         COL 5 AS F10.2 HEADING "PCT OF AVG" ;
S>  LIST ALL ;

DEPTNUM  EMPNUM  LAST_NAME         SALARY      PCT OF AVG
-------  ------  ----------------  ----------  ----------

   1000      23  HOWARD             137000.10      263.46
   1000     202  CLARK               25000.75       48.08
   1000     208  CRAMER              19000.00       36.54
    .       .      .                    .             .
    .       .      .                    .             .
    .       .      .                    .             .
   9000       1  GREEN              175500.00      165.00
   9000     337  CLARK               37000.00       34.82

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

You can drop the temporary table or purge the data and keep the table for use in producing future reports.

2 The Optimizer

The NonStop SQL/MP optimizer is a component of the SQL compiler. The optimizer plays an important role in the high-performance operation of SQL by selecting the most efficient access plan for a query. During compilation, the optimizer examines each data manipulation statement and generates query execution plans (also called access plans) to retrieve the requested data from the database. The optimizer also examines statistics in the catalog for each column referenced by the statement; if statistics are current, the optimizer can choose an efficient query execution plan to retrieve the required data.

For example, a plan for a query that references a single table consists of a strategy for accessing the table using a specified index and begin and end keys. For a query that references multiple tables, an execution plan also specifies the order in which the tables should be accessed.

This section contains these topics:

- SQL Components and the Optimizer on page 2-1
- How the Optimizer Chooses an Execution Plan on page 2-3

To evaluate the performance of an existing query, see Section 6, Analyzing Query Performance.

Issues of database design, tuning, and ongoing maintenance are not discussed in this manual. For more information on these topics, see the *SQL/MP Installation and Management Guide*.

SQL Components and the Optimizer

These components play a role in executing an execution plan provided by the optimizer:

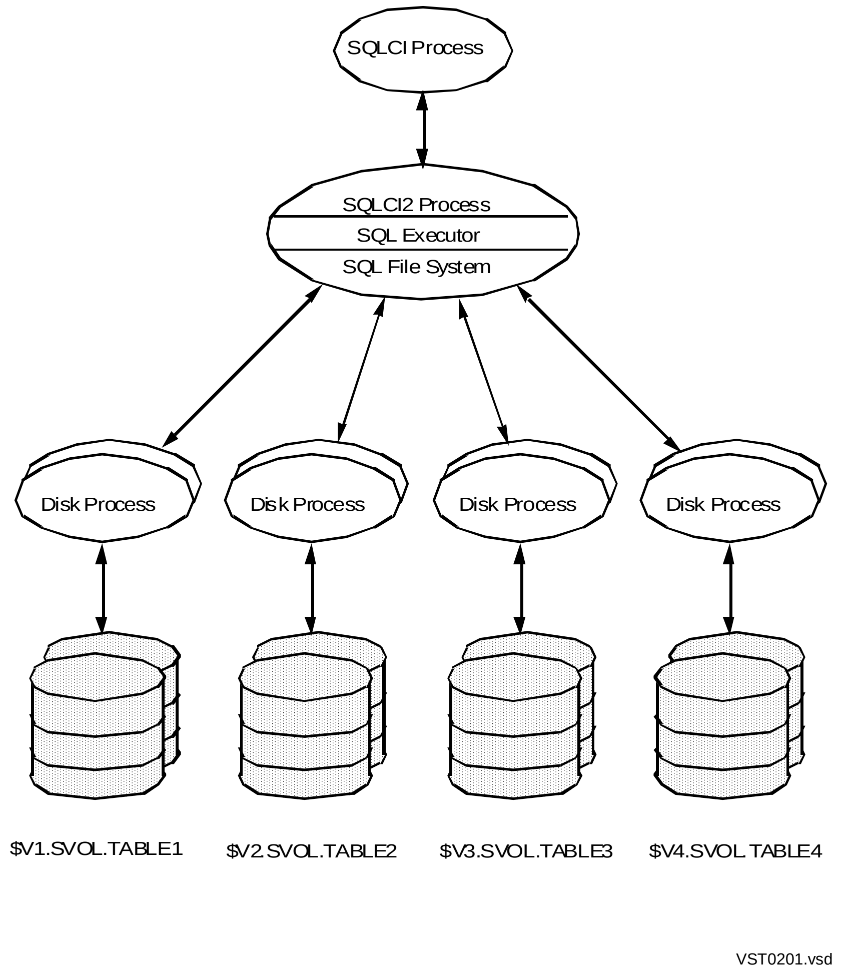
- SQL executor
- File system
- Disk process

SQL statements invoke the SQL executor, a set of library routines that run in the application’s process environment. The executor invokes the file system. The file system accesses each table required by the query execution plan by sending messages to the appropriate disk processes.

Message traffic is reduced by filtering data at its source. Data can be returned to the file system one row at a time or can be buffered and returned as a block by the disk process.

Figure 2-1 on page 2-2 shows the components involved in the execution of an interactive SQL query using SQLCI.

Figure 2-1. SQL Components That Execute a Query



For embedded SQL, the model in Figure 2-1 would not include the SQLCI process, and the SQLCI2 process would be replaced by the user process.

In Figure 2-1, each disk process communicates with a different table. Multiple disk processes can also work on separate partitions of a table or index, thus providing parallel access to a table. To facilitate faster execution, SQL can also process queries in parallel by dividing the query into smaller tasks assigned to separate processors. For more information, see Requesting Parallel Processing on page 4-13 and How Parallel Processing Is Implemented on page 4-14.

These paragraphs explain, in more detail, the tasks performed by the executor, the file system, and the disk process in executing a query.

SQL Executor

The executor is a set of system library procedures that executes compiled DML statements against database tables, views, or the database catalogs. To execute a DML statement, the executor uses the query execution plan generated and optimized by the SQL compiler.

The executor maps logical names to physical names, obtains rows from various tables by using the file system, joins tables, sorts where required, and returns the result to the SQLCI process or to the host variables in a user program.

File System

The file system, which also resides in the system library, handles the opening of tables and indexes, partitioning, sending requests to appropriate disk processes, and buffering of replies, inserts, and updates. When a table is updated, the file system manages the updates to a table and all its alternate indexes.

Disk Processes

Disk processes manage disk space, access paths, locks, log records, and a main memory buffer pool of recently used blocks called the cache. Disk processes can also evaluate predicates, aggregates, groupings, and table constraints. Each disk process authorizes the application process to access the table when the file system sends an OPEN request.

Each disk volume is managed by a set of disk processes, which have a common request queue and a shared cache.

How the Optimizer Chooses an Execution Plan

Before determining the most efficient access plan, the optimizer performs query transformations, if possible, to make the query more efficient and then evaluates these:

- An access path (primary key, index, index-only) to each table:
    - In primary access, rows are read directly from the base table, without using an alternate index.
    - In index access, an alternate index is used to access the base table. An alternate index is a key-sequenced file containing a copy of selected columns of every row of the base table. The columns of an index consist of the index key columns plus the primary key columns. After reading a row from an index, the file system can use the primary key within the index to read the corresponding row of the base table.
    - Index-only access occurs if all of the needed columns are already present in the index row. In such a case, there is no need to access the base table. SQL can retrieve the data directly from the index.

- Execution step at which a predicate or subquery should be evaluated
- The sequence (join order) in which tables will be scanned
- The join strategy—hash, sort merge, key-sequenced merge, or nested
- Optimal sort strategy, by the following:
    - Examining the sort keys for any ORDER BY, GROUP BY, or DISTINCT requests, and combining the sorts whenever possible
    - Eliminating sorts for ORDER BY, GROUP BY, or DISTINCT requests if the chosen access path returns rows in the desired order
- Optimal sort type: sorting in memory (UPS), using a sort process (SORTPROG) or insertion sort
- Whether sequential block buffering (SBB) will be used
- If parallel execution is enabled, whether parallel or nonparallel execution is more efficient
- Whether table locking would be appropriate

These topics are described in Section 3, Improving Query Performance Through Query Designand Section 4, Improving Query Performance With Environmental Options.

Two important aspects of query evaluation are selectivity and cost, both described in Section 5, Selectivity and Cost Estimates

- Selectivity is an estimate of the percentage of rows in a table or an index that satisfy a search condition. Selectivity is represented as a percentage from 0 to 100.
- Cost is an estimate of the amount of time the system takes to complete evaluation of a specific query. Cost includes an estimate of consumption of these resources:
    - Number of I/Os
    - Number of sorts to be performed
    - Amount of data to be processed
    - Number of interprocess messages
    - CPU processing for searches, fetches, processing (joins, grouping), and moving data between buffers

Cost is expressed in terms of the equivalent number of I/O operations that could be performed in the same time.

SQL chooses the execution plan with the lowest estimated cost.

Processor Assignment by the SQL/MP Optimizer and Executor for Executor Server Processes (ESPs)

Apply these rules:

1. The SQL/MP optimizer tries to assign the ESP the same processor as that assigned for the primary disk process of the table partition that the ESP will access.
2. If the disk processor has already been assigned to another ESP, postpone the assignment until other ESPs are assigned to the processors.
3. For all the unassigned ESPs, use the round-robin approach to distribute the rest of the processors:
    a. For the system in which the partition resides, use a processor that is not yet assigned in the first round and assign it to the unassigned ESP.
    b. If there are no free processors, the optimizer marks it as a case where there are more partitions than processors. It then tries to assign processors by repeating Step 1 through Step 3. This strategy involves reusing all the processors in the system (that is treating all processors as not-in-use) to increase the likelihood that the second round will succeed in assigning a processor to a partition.
    c. If no processors are available to the system (that is, if it is a remote partition access or an error occurs in accessing disk information), the current processor status is set to -1, indicating that the SQL/MP executor will assign the processor at run time.
4. During run time, the SQL/MP executor runs a check to see if the processor mentioned in the plan is available. If the planned processor is not available or if the status is set to -1, any processor on the system is assigned.
5. For a parallel repartitioning plan, the temporary tables and ESPs that access them are distributed evenly across available processors. Each ESP is assigned to the processor that is the primary disk process processor for the temporary table that a specific ESP will access. If the amount of data to be repartitioned is large, more than one temporary table (and related ESPs) might be assigned to the same processor.

3 Improving Query Performance Through Query Design

You can formulate the same NonStop SQL/MP query in a number of different ways. Some formulations perform better than others because SQL requires less work to return the same result.

This section describes how to write queries so that they capitalize on SQL performance features.

The guidelines in this section focus on individual SQL statements. Topics include:

Although the focus of this section is on practical, useful guidelines, the predicate-optimization information describes how the optimizer performs its underlying operations. Therefore, portions of this section require a basic understanding of the optimizer, executor, file system, and disk processes as presented in Section 2, The Optimizer.

The premise of this section is that the physical database design process produced an efficient access path to the data; if such a path does not exist, the design cycle should be revisited. For more information, see the *SQL/MP Installation and Management Guide*.

For information about database management options and directives that can influence query performance, see Section 4, Improving Query Performance With Environmental Options. Several management options can be used interactively from SQLCI as well as from a program.

Note. This manual supports NonStop SQL/MP D30.02 and D30.03. Information that describes how the NonStop SQL/MP optimizer chooses a query execution plan can change from release to release.

Selecting Columns for Faster Data Access

When requesting specific columns within a row, these guidelines can help you write queries that can be processed efficiently with minimal message overhead:

- Specify only columns that you really need. Avoid the use of SELECT *, which can disable efficient access mechanisms such as virtual sequential block buffering (VSBB, described in Section 4, Improving Query Performance With Environmental Options) and index-only access.
- Avoid selecting columns you already have values for (such as those having equal predicates to host variables). Column selection can influence message size; it can also influence whether the optimizer chooses sequential block buffering.
- If you specify most of a leftmost group of contiguous columns in your query, consider adding remaining columns so that the SELECT specifies contiguous columns. This can allow efficient bulk move operations, but might, however, disable index-only access; check EXPLAIN output to compare resulting plans.
- If you retrieve a set of column values to accumulate a total, and could perform the operation with aggregate functions, change the queries so that aggregates perform the operation. As of SQL/MP 2.0, this change reduces message cost because the evaluation is performed at the disk process level when possible. For more information, see How the Optimizer Processes Aggregates and Group-By Operations on page 3-46.
- For complex queries or difficult problems, consider using multiple step queries, left outer join, and UNION as possible alternatives. Try to break up the query into smaller steps that use keys and combine smaller result sets.
- To avoid sorts, specify DISTINCT only for matching leftmost columns of an index.

Column definition is also important for good performance. To learn how to define columns for maximum performance, see the *SQL/MP Installation and Management Guide*.

When writing queries, a smaller number of selected columns generally results in a smaller message size—or fewer blocks moved—between the file system and the disk process. For example, this query retrieves a list of employees from the EMPLOYEE table in the sample database:

```
SELECT * FROM EMPLOYEE;
```

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

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

To improve the performance of the query, you can refine the query by specifying only the columns you really need, reducing the amount of data SQL must return.

Suppose that the only items you are really interested in are the employee's identification number, last name, first name, and job category for employees in department 3000. You can request this information by entering

```
SELECT empnum, last_name, first_name, jobcode
   FROM employee
   WHERE deptnum = 3000 ;
```

The query returns this result:

```
EMPNUM   LAST_NAME      FIRST_NAME      JOBCODE
------   -------------  -------------   -------
    29   RAYMOND        JANE                100
    75   WALKER         TIM                 300
   201   HERMAN         JIM                 300
   343   TERRY          ALAN                900

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

As you refine the query to return only the data you require, the number of rows that satisfy the query is reduced, which makes the query more efficient and diminishes the processing overhead on the system.

Preparing Queries

For SQLCI and dynamic SQL users, if you plan to use the same query repeatedly, you can prepare the statement and cause SQL to save a compiled version of it for later use. By doing so, you increase the efficiency of subsequent use of the query. You can do this from within an SQLCI session or from a program. For example, this statement prepares a query and associates the name Q1 with it:

```
>> PREPARE Q1 FROM
+>  SELECT * FROM EMPLOYEE ;
```

To execute the query, enter this statement:

```
>> EXECUTE Q1 ;
```

To explain the query, enter this statement:

```
EXPLAIN Q1 ;
```

For more information about EXPLAIN, see Section 6, Analyzing Query Performance.

How the Optimizer Processes Predicates

A predicate is an expression that makes an assertion about data. This subsection describes how predicates are classified by the optimizer, how they can be transformed by SQL prior to evaluation, and how SQL evaluates predicates.

Classification of Predicates

This terminology is used to classify predicates by syntactic structure:

- A join predicate is any predicate that refers to columns in two or more tables (including the same table referenced more than once); for example:

  ```
  t.col = u.col
  ```

- An equijoin predicate relates columns using an equal (=) comparison operator.
- A range predicate establishes an upper or lower limit on the value of a column. A range predicate uses one of the comparison operations (<, <=, >=, or >) to compare a column with the value of an expression; for example:

  ```
  EMP.EMPNUM > :hv1
  ```

- A multivalued (compound) predicate specifies more than one column or value on each side of a predicate. It compares multiple columns with corresponding values.
- A predicate set is a series of predicates that comprises only ANDs, BETWEENs, and IN predicates. This is an example of a predicate set:

  ```
  B BETWEEN 5 AND 10 AND B = 5 AND C = 10 AND C IN (5, 10)
  ```

Transformation of Predicates

SQL transforms some types of predicates into other forms that are more efficient but logically equivalent to the original predicate. The modified query either reduces the complexity of the query or improves the performance of the query. Transformation is typically done with LIKE, BETWEEN, NOT, IN, and join predicates.

Transformation of Key Column Predicates and Predicate Sets

For predicates that contain key columns, SQL uses an optimization method called the MultiDimensional Access Method (MDAM). Based on the key predicates you specify, MDAM considers all possible key values and attempts to read only those rows. Duplicate key predicates are eliminated at run time. Whenever the cost of a direct access method is less than the cost of a table scan, SQL could choose MDAM.

Some WHERE clauses cannot be processed by MDAM as single predicate sets. Usually these clauses contain one or more ORs and must be processed as multiple predicate sets.

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

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

This IN predicate is converted into this:

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

Consider this query:

```
SELECT * FROM T WHERE
   ((C = 10 AND B BETWEEN 5 AND 10) OR A IN (2, 4, 5)) AND
   ((A = 4 AND C = 5) OR (C IN (5,10) AND (B = 5 OR A = 2))) ;
```

Using MDAM, the optimizer transforms this query into these six predicate sets:

```
(A = 4 AND B BETWEEN 5 AND 10 AND C = 10 AND C = 5)
OR (B BETWEEN 5 AND 10 AND B = 5 AND C = 10 AND C IN (5, 10))
OR (A = 2 AND B BETWEEN 5 AND 10 AND C = 10 AND C IN (5, 10))
OR (A IN (2, 4, 5) AND A = 4 AND C = 5)
OR (A IN (2, 4, 5) AND B = 5 AND C IN (5, 10))
OR (A IN (2, 4, 5) AND A = 2 AND C IN 5, 10))
```

SQL might choose MDAM based on the key predicates you specify in the query, but you can force MDAM by using a CONTROL directive. The EXPLAIN plan for the query shows when SQL uses MDAM.

Plans That Do Not Use MDAM

MDAM is not used in these cases:

- The predicate contains no key columns.
- A join predicate is used for a key-sequenced merge join. For more information on key-sequenced merge joins, see Key-Sequenced Merge Join on page 3-27 and be aware that MDAM can be used for reading the outer table.
- An OR is used to connect a key predicate with a nonkey predicate.

  In this example, the query would not use MDAM because an `OR` connects the key predicate `UNIQUE2 = 5` with the nonkey predicate `FOUR = 2`:

  ```
  SELECT * FROM
    FROM TENKTUP1
    WHERE UNIQUE2 = 4
    OR (UNIQUE2 = 5 OR FOUR = 2);
  ```

  However, MDAM would be considered for this query because an `AND` is used to connect the key predicate `UNIQUE2 = 5` with the nonkey predicate `FOUR = 2`:

  ```
  SELECT * FROM
    FROM TENKTUP1
  ```

```
     WHERE UNIQUE2 = 4
     OR (UNIQUE2 = 5 AND FOUR = 2);
```

Transformation of LIKE Predicates

A LIKE predicate searches for rows that match a pattern. When using LIKE against a positioning column of an index, if the match is on a leftmost matching string (a literal beginning with anything other than the pattern match symbol, % or _), SQL transforms the statement into an equivalent range predicate.

For example, this request contains a leftmost matching string (the leftmost characters are stated explicitly). Suppose that the column C contains only uppercase letters:

```
C LIKE "ABC%"
```

SQL transforms this clause to this:

```
C >= "ABC     " AND C <= "ABCD    "
```

This query retrieves all names that start with the string “CH” (CHARLES, CHRIS, and so on):

```
SELECT NAME, PHONE_NUMBER
  FROM PHONE_BOOK
  WHERE NAME LIKE "CH%"
```

SQL transforms the preceding query into this:

```
SELECT NAME, PHONE_NUMBER FROM PHONE_BOOK
  WHERE NAME LIKE "CH%"
  AND NAME >= "CH"
  AND NAME < "CI"
```

With this modification, SQL can take advantage of an index on NAME (if one exists) and use values in the predicates as the begin and end keys to the index:

```
NAME >= "CH" and NAME < "CI"
```

Thus, SQL retrieves only those rows that are alphabetically equal to or greater than CH but before CI.

SQL does not transform a LIKE predicate if there is a wild-card character (% or _) as the leftmost character of a column value in the predicate.

SQL does not transform a LIKE predicate if there is a collation involved.

When a LIKE predicate is present on a key column and the column value does not start with a wild card, the predicate becomes a candidate for MDAM optimization. When MDAM is chosen, the transformed predicate shows as an MDAM predicate set in the EXPLAIN plan.

Transformation of BETWEEN Predicates

SQL transforms a BETWEEN predicate into the equivalent range predicate; for example:

```
X BETWEEN Y AND Z
```

is transformed to

```
X >= Y AND X <= Z
```

Transformation of Predicates With the NOT Operator

A predicate with one or more NOT operators is transformed to simplify and reduce the number of NOT operations; these series of transformations illustrates the process:

```
NOT ((a > b) OR (x < y))
```
becomes:
```
NOT (a > b) OR NOT (x < y)
```
and then becomes:
```
a <= b OR x >= y
```

When used with EXISTS, LIKE, or IS NULL, the NOT operator is not transformed; it remains the same.

MDAM processes NOT predicates on key columns as key predicates and does key accesses on them.

Transformation of IN Predicates

SQL always transforms an IN predicate into another form. The final form depends on whether the expression of the IN predicate is a value list or a subquery.

IN Predicates With Value Lists

If the expression of an IN predicate contains a value list, SQL transforms the list into a search condition with the predicates connected by one or more OR operators; for example, this predicate:

```
DEPT_NUM IN (:hv1, :hv2, :hv3)
```

is transformed into:

```
DEPT_NUM = :hv1 OR DEPT_NUM = :hv2 OR DEPT_NUM = :hv3
```

If DEPT_NUM is the key prefix—the leading (leftmost) contiguous set of columns in the key—then OR optimization might be performed on the query predicate. DEPT_NUM need not be a key prefix for MDAM processing to take place.

SQL transforms the list of values into a search condition that has equality predicates connected by one or more OR operators:

```
WHERE DEPT_NUM = :hv1
  OR DEPT_NUM = :hv2
  OR DEPT_NUM = :hv3
```

With MDAM, only rows that match the values in the value list are read. Unlike OR optimization, when a predicate has more than one element in the value list, MDAM eliminates any duplicate values at run time, not at compile time. Because this is done before any tables are accessed, there is no performance penalty. MDAM processing appears as an MDAM predicate set in the EXPLAIN plan.

For more information, see Using OR Operators in Predicates on page 3-22.

IN Predicates With Subqueries

If the expression of an IN predicate is a subquery, then SQL transforms the IN predicate into an “= ANY” subquery. For example, this predicate

```
DEPT_NUM IN ( SELECT DNUM ... )
```

is transformed into:

```
DEPT_NUM = ANY ( SELECT DNUM ... )
```

SQL optimizes the execution of noncorrelated ANY, ALL, and SOME subqueries. The SQL executor builds a table in memory that contains the result of the subquery. In the preceding example, SQL uses the key DNUM to search the in-memory table for values that match the DEPT_NUM value retrieved by the outer query.

SQL does not generate parallel plans for IN subqueries. The use of EXISTS or a join operation, which can both support parallel plans, might be more efficient than using an IN predicate in a subquery.

Transformations Related to Joins

The optimizer uses a feature called Query Rewrite to transform user-specified search conditions for faster execution of join queries. These automatic transformations are especially useful in decision support applications—applications that allow you to invoke ad hoc queries and often can consume large amounts of time and disk space.

The transformation of search conditions can provide these benefits:

- Reduce the number of rows involved in a join operation
- Present a broader range of alternative plans during join optimization
- Remove redundant or unnecessary operations
- Make better use of existing access paths

Applications that benefit most include those that have one or more of these characteristics:

- SQL DML generated by software
- Extensive use of shorthand views
- Applications ported to NonStop SQL/MP from another database management system

- Queries written by users who are not NonStop SQL/MP experts

Optimization time can increase when these transformations take place. Although the transformations are automatic, you can use a DEFINE to control compilation time and still receive optimization benefits. For information on the DEFINE, see Controlling the Expansion of Predicates on page 3-20.

How Query Rewrite Works

When the optimizer determines that a query can benefit from Query Rewrite, it repeatedly performs these tasks until the query reaches a final state:

- Transforms unnecessary left joins to inner joins
- Propagates constants
- Expands equality predicates
- Eliminates unnecessary predicates
- Simplifies predicates

The EXPLAIN plan for the query shows the results of these tasks.

Transforming Left Joins to Inner Joins

In a left join, if a row from the table specified on the right side of the left join operator does not satisfy a search condition, SQL preserves the row from the table specified on the left by extending it with as many null values as there are columns in the table on the right. These rows are called null-augmented rows.

If a WHERE or INNER JOIN predicate is certain to eliminate all of the null-augmented rows generated by a left join, then the optimizer transforms the left join into a more efficient inner join.

Consider this view and query:

```
CREATE VIEW V AS SELECT T.C, U.D
   FROM T LEFT JOIN U ON T.C = U.C ;
```

```
SELECT * FROM V WHERE D = 1 ;
```

T is the table on the left side of the JOIN keyword. T contains no column D. To preserve rows from the T table, SQL extends each row that does not satisfy the condition D = 1. The result is the generation of null-augmented rows.

If an inner join were performed, only the rows that satisfy the condition D = 1 would be returned, and the null-augmented rows would not be generated. Therefore, for the query in the example, the optimizer converts the left join to an inner join.

The same is true in this query:

```
SELECT * FROM V WHERE D = 1 or D = 2 ;
```

In this query, the left join cannot be converted to an inner join:

```
SELECT * FROM V WHERE C = 1 OR D = 2 ;
```

The search condition C = 1 OR D = 2 selects rows in which C = 1, regardless of the value of D.

Suppose that the left-join operation generates a null-augmented row in which C = 1 and D is null. A logical OR is true if either of its operands is true. For this row, the condition C = 1 is true, so the row is included in the query result, regardless of the value of D. The optimizer cannot convert this left join to an inner join because the conversion would eliminate the null-augmented row and change the result of the query.

Propagating Constants

Propagation of constants deals with constant expressions and equivalence classes. A constant expression is an expression that contains no subqueries and no column references other than outer references, such as in this example:

```
:hva + :hvb * 2
```

An equivalence class is a set of expressions equal to each other. For example, in this statement, 1 is a constant expression, and A, B, C, and 1 are all expressions in the same equivalence class:

```
SELECT * FROM T WHERE
   A = 1
   AND B = 1
   AND C = A ;
```

Some queries execute faster if members of the same equivalence class are replaced by constant expressions. This replacement is called constant propagation. If a member of an equivalence class is a constant expression, then most occurrences of the other members of the class can be replaced by the constant expression; for example:

```
A = B + C AND C = 1
```

can be transformed into:

```
A = B + 1 AND C = 1
```

Because the LIKE predicate can distinguish character strings with different numbers of trailing spaces, LIKE predicates are exempted from constant propagation. For example, if (C = D) and (C LIKE X), the optimizer cannot infer that D LIKE X.

Expanding Equality Predicates

You can code the same query in several different ways. Each way generates the same result, but if Query Rewrite did not exist, the queries could differ widely in performance, as in these examples:

```
SELECT * FROM T, U WHERE T.C = :hv AND U.D = :hv ;

SELECT * FROM T, U WHERE T.C = U.D AND T.C = :hv ;

SELECT * FROM T, U WHERE T.C = U.D AND U.D = :hv ;

SELECT * FROM T, U WHERE T.C = U.D AND T.C = :hv
   AND U.D = :hv ;
```

All these alternatives produce the same result. For each alternative, the optimizer would consider a different subset of the possible access plans, resulting in a variance in performance.

For instance, no hash joins would be considered for the first query because T.C = U.D is not explicitly stated. The nested loop join method is the only method that can be used when no equijoin predicate directly connects a pair of join columns.

The second query would require a 100 percent scan of table U because U.D = *:hv* is not explicitly stated. The third alternative would perform a 100 percent scan of table T because T.C = *:hv* is not explicitly stated.

The fourth alternative states all equality relationships explicitly. Therefore, all appropriate join methods would be considered. The optimizer would apply the T.C = *:hv* and U.D = *:hv* predicates before the join.

Using Query Rewrite, the optimizer adds any missing predicates so that it has more plans to consider and a better chance of executing the best plan. In the example, the optimizer converts the first three queries into the fourth.

Before adding missing predicates, the optimizer considers whether additional equality predicates are potentially useful for optimization and then takes the appropriate action.

When Additional Equality Predicates Are Useful

The optimizer might consider additional equality predicates useful if their positions within the query allow them to execute independently from other predicates. It expands predicates that are not within the scope of a logical OR operator and queries that are eligible for OR optimization. For details about OR optimization, see Using OR Operators in Predicates on page 3-22. For information on how you can affect the expansion of predicates, see Controlling the Expansion of Predicates on page 3-20.

Eliminating Unnecessary Predicates

The optimizer eliminates predicates not necessary for processing, such as redundant predicates.

Simplifying Predicates

Sometimes the optimizer can determine the results of predicates at SQL compilation time and then simplify or eliminate the predicates.

When the optimizer compares values known to be equal, it substitutes a NOT NULL predicate if both these conditions occur:

- The expression can be null.
- The operator is one of the following: <=, =, or >=.

Otherwise, the comparison reduces to either true or false, depending on the operator.

Evaluation of Predicates

These three NonStop SQL/MP components participate in data retrieval and can also evaluate predicates:

- SQL executor, which usually evaluates predicates for a serial search against nonkey columns
- File system, which evaluates predicates for a serial search of a memory-resident block of rows
- Disk process, which evaluates predicates for searches with leftmost column matches on defined ranges

The disk process obtains data first, then passes it to the file system, which in turn passes the data to the SQL executor process. Section 2, The Optimizer, describes these components in more detail.

To minimize resource use, SQL evaluates predicates as early as possible in this processing chain. By evaluating predicates in the disk process or file process, SQL can minimize the number of rows searched and the data movement between the application and the disk process. Unwanted rows contribute nothing to the output but add to the expense of the query. If unwanted rows can be eliminated early, by the disk process for example, the query uses fewer resources to transmit the data to the user.

To determine where a predicate is evaluated in an existing query, use the EXPLAIN utility, described in Section 6, Analyzing Query Performance.

The types of predicates you use determine where they are evaluated. The optimizer analyzes predicates and assigns each one to the most appropriate role within the execution plan, using these four predicate categories (in order of decreasing efficiency):

- Key predicate
- Index predicate
- Base-table predicate
- Executor predicate

If a query execution plan changes (because of new statistics or a new index, for example) the category of a predicate might change.

These paragraphs describe the four evaluation categories of predicates.

Key Predicates

A key predicate is a begin key or an end key that defines a lower or upper bound on key columns for sequential retrieval. A begin key establishes an initial row position within a table or index; the end key establishes a stopping point. Rows are read sequentially (in ascending or descending order) as long as the end-key predicate remains true.

In this query, the WHERE...AND... clause defines lower and upper bounds for the search, assuming there is an index on the LAST_NAME column:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE LAST_NAME >= JONES
     AND LAST_NAME <= SMITH ;
```

Key predicates can greatly reduce the resources needed for a query. Specify key predicates to avoid the expense of reading and examining an entire table or index. Key predicates are evaluated by the disk process before returning data to the file system.

If a query specifies a range predicate or an equality predicate for any key column, SQL considers MDAM. When this happens, data outside the bounds need not be read from disk or handled in any way.

For more information on key predicates, see Positioning With Key Predicates on page 3-16.

Index Predicates

An index predicate is any predicate, other than a begin-key or end-key predicate, that is applied to the rows of an alternate index. Whenever a scan operation uses alternate index access, all possible predicates are applied to index rows before accessing the base table. If an index predicate evaluates to false or unknown, SQL does not access the corresponding base table row.

Index predicates are evaluated for every index row within the bounds defined by the begin-key and end-key predicates. Index predicates can reduce the number of rows read from the base table, thus avoiding physical I/O operations.

Index predicates are evaluated by the file system or the disk process, depending on the type of I/O buffering chosen:

- If single-row access (no sequential block buffering) or virtual sequential block buffering (VSBB) is chosen, the disk process evaluates index predicates, returning only those rows for which the predicates are true.

- If real sequential block buffering (RSBB) is chosen, the file system evaluates index predicates.

For more information about RSBB, VSBB, and other buffering options, see Section 4, Improving Query Performance With Environmental Options.

To review the index predicates chosen for a scan, see the EXPLAIN listing for the query. Index predicates are noted with the title “Index pred.”

This example illustrates the use of key predicates with a table that has 100 rows:

```
CREATE TABLE t (a INT, b INT, c INT) ;
CREATE INDEX ti ON t (a,b) ;

INSERT INTO t VALUES (0,0,0) ;
INSERT INTO t VALUES (1,1,1) ;
INSERT INTO t VALUES (2,2,2) ;
INSERT INTO t VALUES (3,3,3) ;
 ...
INSERT INTO t VALUES (99,99,99) ;

SELECT * FROM t WHERE a < 4 AND b < 2 ;
```

SQL requests this, assuming an index access path is chosen:

- Access to the data through index ti. The end key would specify a<4 and an index predicate would specify b<2. Two rows of the index would qualify.
- Access to the base table to retrieve the corresponding rows (b = 0 and b = 1). The file system requires base-table access because of the * (all) in the SELECT statement.

Base-Table Predicates

A base-table predicate is any predicate applied to rows in the base table.

SQL evaluates base-table predicates for every base table row within the bounds defined by the primary begin-key and end-key predicates or for every base table row accessed.

Base-table predicates are evaluated by the file system or the disk process, depending on the type of I/O buffering chosen:

- If the primary access path is chosen and real sequential block buffering (RSBB) is used, the file system evaluates the base-table predicates.
- If index-only access is chosen, there were no base-table predicates.
- In all other cases, the disk process evaluates base-table predicates before returning data to the file system.

Base-table predicates do not reduce the amount of physical I/O to the base table. If, however, base-table predicates are evaluated by the disk process, they can reduce the number and size of messages returned from the disk process to the file system, and can reduce the amount of data to be sorted or hashed for sort merge joins, hash joins, or aggregate functions.

Executor Predicates

An executor predicate is a predicate that must be evaluated by the SQL executor instead of by the disk process or file system.

Executor predicates are the least efficient type of predicate because they reject rows only after the rows have already been handled by the disk process and the file system. These predicates do not reduce the amount of physical I/O to the base table, but like all types of predicates, executor predicates might reduce the number of rows processed for a sort, correlated subquery, merge join, and so on, and therefore reduce the total cost of the query.

These predicates are always evaluated by the SQL executor:

- Correlated subquery predicates
- Quantified (ANY, ALL, SOME) subquery predicates
- IN and EXISTS subquery predicates
- Merge-join and hash-join predicates
- Predicates contained in these clauses:
  - HAVING clause
  - WHERE clause of a left join query that references columns from the table that appears on the right of the keywords LEFT JOIN
- Executor predicates connected by an OR operator:

```
(EMP.EMPNUM = :hv1) OR
  (EXISTS (SELECT col1,col2 FROM TABLE1 WHERE col1 = 50) )
```

Writing Efficient Predicates

There are several guidelines that can help improve the performance of predicates. These guidelines are summarized here and are described in the following subsections:

- Use key predicates for positioning whenever possible.
- Use join predicates to specify search conditions when joining multiple tables.
- Use multivalued predicates when possible.
- Understand when OR operations are optimized and when they are not.
- Understand the performance implications of using the LIKE predicate.

Positioning With Key Predicates

Key predicates can greatly reduce the resources required for a query. To specify a key predicate, use a WHERE clause that restricts the search based on the primary key or an index. You can do this, for example, using these predicates:

- An equality predicate (=) on a key or index column
- Range (begin-key and end-key) predicates on key or index columns
- A BETWEEN predicate on a key column
- An IS NULL predicate on a key column
- A LIKE predicate on a key column that uses a literal beginning with anything but the pattern match symbol (% or _)
- IN predicate with a value-list on key columns
- AND predicates
- OR predicates that match leftmost key or index columns

If the leading key column or columns are missing from the predicate, MDAM allows tables and indexes to be accessed through an index:

```
WHERE KEY3 < 10
  AND KEY4 = 6
```

The EXPLAIN utility lists begin-key and end-key predicates or MDAM key predicates. If your EXPLAIN output says NONE for either the begin-key or end-key predicate or for both, then consider adding bounds to the query to improve performance.

Use equality predicates for keys or partial keys where possible. SQL uses key positioning for all leftmost key columns that have equality predicates.

These predicates cannot be used to specify begin-key and end-key conditions:

- EXISTS predicates
- IN predicates with a subquery providing the list
- LIKE predicates that use a literal beginning with a pattern match symbol (% or _) or that use a host variable
- Quantified predicates (SOME, ANY, ALL)
- Predicates with arithmetic expressions

If both the left and right arguments of a comparison predicate reference the same table, the compiler does not use the predicate in a begin-key or end-key for index access.

Examples of Key Predicates

This query specifies a begin key and uses a host variable. If an index exists on the column LAST_NAME and if SQL uses the index to perform the search, then this query performs better than if SQL sequentially reads every row in the table, starting with the first row.

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE LAST_NAME >= :hvar1 ;
```

The query returns this result:

```
LAST_NAME    FIRST_NAME   SALARY
---------    ----------   ---------
GREEN        ROGER        175500.00
HENDERSON    BEN           65000.00
HOWARD       JERRY        137000.10
     .          .              .
     .          .              .
WINTER       PAUL          90000.10

--- 17 row(s) returned.
```

The next query specifies a begin and end key, and probably performs better. Specification of both a begin and end key increases the likelihood that SQL will choose index access instead of a table:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE LAST_NAME >= :hvar1
     AND LAST_NAME <= :hvar2 ;
```

The likelihood of index access is higher for the second example because the selectivity is lower for the second example. Selectivity is estimated as .33 for the first example and .33 * .33 = .11 for the second example. For more information about selectivity, see Section 5, Selectivity and Cost Estimates.

The query returns the same result:

```
LAST_NAME    FIRST_NAME   SALARY
---------    ----------   ---------
GREEN        ROGER        175500.00
HENDERSON    BEN           65000.00
HOWARD       JERRY        137000.10
     .          .              .
     .          .              .
WINTER       PAUL          90000.10

--- 17 row(s) returned.
```

Using Range Predicates for Positioning

A range predicate on a column of the key can be used for key positioning, but subsequent key columns cannot be used for key positioning and instead are evaluated as index or base table predicates.

In this example, PKEY1, PKEY2, PKEY3, and PKEY4 are the first four columns of the primary key:

```
SELECT PKEY1, PKEY2, PKEY3, PKEY4, COL1 FROM TABLE1
  WHERE  PKEY1 =  :hv1
    AND  PKEY2 =  :hv2
    AND  PKEY3 >= :hv3 AND PKEY3 <= :hv4
    AND  PKEY4 >  :hv5
```

SQL uses PKEY1, PKEY2, and PKEY3 as key predicates. PKEY4 > *:hv5* cannot be a key predicate, because it follows an expression with a range predicate. Instead, it becomes a base table predicate.

For an alternative to specifying key columns in separate clauses, see Specifying Multivalued Predicates on page 3-19. For information on predicates with missing key columns, see Index Predicates on page 3-13.

Defining Key Predicates With Multiple Columns

For tables and alternate indexes with keys that consist of multiple columns, specify predicates with a key prefix (either a full or partial key, starting with the leftmost key column). In this example, the PARTLOC table has a primary key consisting of LOC_CODE and PARTNUM. There is no index on PARTLOC. This query causes a scan of the entire table:

```
SELECT PARTNUM, QTY_ON_HAND
   FROM PARTLOC
   WHERE PARTNUM > 10 ;
```

If you add a value for the leftmost part of the key (defined as LOC_CODE), or provide an index on PARTNUM, SQL can retrieve the data more efficiently.

Include leftmost key column values, if known, in every predicate.

Specifying Join Predicates

In general, specify a join predicate when you request a join operation. (For a description of join predicates, see Combining Data From More Than One Table on page 1-51.) To broaden the types of join strategies SQL can use, include an equal operator in the join predicate. For more information, see Writing Efficient Joins on page 3-39.

Specifying Multivalued Predicates

You can influence the selection of an access path—and eliminate extra scanning—by specifying more than one column or value on each side of a predicate. Such a predicate, called a multivalued predicate, compares multiple columns with corresponding values.

This strategy can be especially useful for key prefixes. If key columns are supplied sequentially (as in `col1 = x AND col2 = y`), SQL might read the entire table, possibly scanning unnecessary rows several times. Instead, if the keys are a prefix for an index, you can use the multivalued predicate construct to specify the set of keys as a group, eliminating extra scanning.

The columns in a multivalue predicate can be used as keys if a corresponding multicolumn index exists.

For example, suppose that you want to list all employee rows with names that come after “JONES, JOHN”; you might write this query, assuming that LAST_NAME and FIRST_NAME are key columns:

```
SELECT *
   FROM employee
   WHERE last_name > "JONES"
     OR (last_name = "JONES"
        AND first_name > "JOHN") ;
```

This query essentially searches for all JONES entries after JOHN JONES and then searches for all rows past LAST_NAME = JONES. Because an OR operator is used, this query might cause SQL to read the entire table, with possible rescanning of unneeded rows.

You can write a more efficient query by combining the two key columns in a single, multivalued predicate that represents the same conditions, as follows:

```
SELECT *
   FROM EMPLOYEE
   WHERE LAST_NAME, FIRST_NAME > "JONES", "JOHN" ;
```

The columns in the multivalued predicate are used as keys if a multicolumn index exists on LAST_NAME, FIRST_NAME. The statement is equivalent to the earlier one, but is more efficient and more compact.

Column Order Considerations

A multivalue predicate can be matched with a group of key columns for use as a begin-key or end-key predicate only if all of the key columns have the same ordering attribute: either all must be ascending, or all must be descending, as defined in the KEY specifications of CREATE TABLE and the CREATE INDEX statements.

To maximize the performance of multivalue key predicates, avoid using the DESC option in KEY specifications in the CREATE TABLE and CREATE INDEX statements, or define all key columns as DESC.

If the application does not permit a uniform ordering of key columns, an alternative is to truncate the multivalue predicate so that it excludes the nonconforming column.

Controlling the Expansion of Predicates

If the optimizer determines additional equality predicates are useful, then it considers each equivalence class separately for expansion. You can affect the expansion by using a DEFINE for =_SQL_CMP_EQ_LIMIT. This system DEFINE allows you to specify the number of equivalent predicates that SQL can use to optimize the join order of the tables and the index selection.

If a join involves numerous tables, the time the optimizer uses to consider the various combinations of tables can be long. Set the value for the DEFINE low enough to obtain a reasonable SQL compilation time but high enough to obtain the benefits of the optimization process.

Table 3-1 shows typical values and their effects on optimization.

Table 3-1. The Effect of =_SQL_CMP_EQ_LIMIT Values on Compilation Time

=_SQL_CMP_EQ_LIMIT Value	Effect on Optimization
0 or 1	SQL does not generate any extra join predicates.
2 or 3	Increased compile time is negligible.
4 to 6	Compile time is increased but a wider range of table combinations can occur, allowing a more efficient possibly query plan.

The default value for =_SQL_CMP_EQ_LIMIT is 5.

For more information on this DEFINE, see the *SQL/MP Reference Manual.*

Table 3-2 on page 3-21 describes the rules for expansion of predicates for various DEFINE values. The optimizer considers the conditions in the table in sequence and applies the first condition satisfied.

Table 3-2. Rules for Expansion of Useful Equality Predicates

Condition	Expansion	Example

The equivalence class contains a constant expression.	Equality predicates are generated between all pairs of members that are single columns of different tables so that a hash join, sort merge join, or key-sequenced merge join can occur. Also, an equality predicate is generated between the constant expression and each of the other members.	Before:
		T.C = 1 AND
		U.D = 1
		After:
		T.C = 1 AND
		U.D = 1 AND
		T.C = U.D

The number of members of the equivalence class is less than or equal to the value for the DEFINE for =_SQL_CMP_EQ_LIMIT.	Equality predicates are generated between all pairs of members.	Suppose that the DEFINE =_SQL_CMP_EQ_LIMIT is set to 3 or greater.
		
		Before:
		T.C = T.D AND
		T.D = V.E
		After:
		T.C = T.D AND
		T.D = V.E AND
		T.C = V.E

The number of members of the equivalence class that are single columns is less than or equal to the value for the DEFINE for =_SQL_CMP_EQ_LIMIT.	Equality predicates are generated between all pairs of members that are single columns of different tables.	Suppose that the DEFINE =_SQL_CMP_EQ_LIMIT is set to 2.
		
		Before:
		T.C = T.D + :hv1 AND
		U.C = T.D + :hv1
		After:
		T.C = T.D + :hv1 AND
		U.C = T.D + :hv1 AND
		T.C = U.C

The number of members in the equivalence class that are single columns exceeds the value for the DEFINE for =_SQL_CMP_EQ_LIMIT.	No extra equality predicates are added for this equivalence class.	Suppose that the DEFINE =_SQL_CMP_EQ_LIMIT is set to 2.
		
		Before:
		T.C = T.D AND
		U.C = T.D
		After:
		T.C = T.D AND
		U.C = T.D

Using OR Operators in Predicates

SQL uses a feature called OR optimization for some queries with OR operations. OR optimization uses more than one access path to obtain the data and eliminates duplicate predicates at compile time to produce the result.

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 SQL/MP accesses any tables so there is no performance penalty.
- MDAM can process multiple tables in a query and 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. That is, WHERE clauses can have more than one level of OR operations.

MDAM is enabled by default.

Choosing Optimized OR Plans

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

- There are 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 in disjunctive normal form. That is, there is only one level of OR operations; for example:

  ```
  (P1 AND P2 AND P3) OR (P4 AND P5) OR (P6 AND P7 AND P8)
  ```

  Note that the parentheses are not required; the AND operator has precedence over the OR operator.

For execution plans that use OR optimization, the optimizer considers index-only access for each index scan independently. Index-only access can, however, be used only if the index contains all of the columns referenced in the entire query.

Additional indexes might enable OR optimization for additional columns.

Plans That Do Not Use OR Optimization

In general, OR optimization is not used in these cases:

- A query involves more than one table.
- Columns in the predicate are not part of a key prefix.
- At least one single predicate—or set of predicates connected by AND operators—contains only nonkey predicates.

- At least one single predicate—or set of predicates connected by AND operators—contains an executor predicate.
- A group of predicates connected by AND operators has a high selectivity, which is likely to occur when the operator is not an equal operator (=); instead, SQL might choose to read the table sequentially to search for rows that satisfy the query. (For more information about selectivity, see Section 5, Selectivity and Cost Estimates.)

If you do not see entries like Access Path 1 and Access Path 2 in your EXPLAIN output, you are not getting OR optimization. One alternative might be to use the UNION operator.

Examples of OR Optimization

Suppose that columns B and C are key columns of an index I on table T (A, B, C, D, E, primary key A). If a query on T contains these predicates, then OR optimization might be performed:

```
WHERE ( B = :hv1 AND C BETWEEN :hv2 AND :hv3 )
  OR ( B = :hv1 AND C > :hv1 AND E < :hv1 )
  OR A = :hv1
```

OR optimization would use index access on I for the first predicate, an index access through I for the second predicate, and a primary-keyed access for the third predicate. In the absence of an index, SQL reads the table sequentially to search for rows that satisfy the query.

This example would not enable OR optimization:

```
WHERE (B = :hv1)
  OR  (B = :hv2)
  AND (D = :hv3)
```

but this would enable OR optimization:

```
WHERE (B = :hv1 AND D = :hv3)
  OR  (B = :hv2 AND D = :hv3)
```

This example, with primary key A,B and index C,D, would also enable OR optimization:

```
WHERE ((C=10) AND ...    )
  OR  ((A=10) AND ...    )
  OR  (((A,B)>(10,10)) AND ...)
  OR  ((C>10) AND ...    )
```

Note that this example would enable OR optimization because each set of predicates separated by OR operators contains possible key predicates (either for the alternate index or for the primary key) possibly connected by AND operators to additional index or base table predicates. If you connected this list with an AND operator and another predicate (either base table or index), then OR optimization would not be considered.

Using LIKE Predicates

LIKE constructs can cause full table scans and can therefore result in inefficient queries. Avoid using the LIKE predicate when another operator might be more efficient. Instead, use the equal operator (=) whenever possible.

If you must use LIKE, consider these guidelines:

- Avoid using the LIKE predicate beginning with a pattern match symbol (% or _), as in this example:

  ```
  WHERE LAST_NAME LIKE "%SON"
  ```

  If the wild-card character (% or _) is used as the leftmost character of a column value, SQL cannot transform the LIKE predicate into a range predicate for a bounded search.

  Consequently, specifying LIKE, beginning with a pattern match symbol, causes a full table scan.

- Avoid constructs such as this:

  ```
  WHERE LAST_NAME LIKE ?P1
  ```

  If the LIKE predicate compares a column with a parameter in SQLCI, SQL cannot transform the LIKE predicate into a range predicate for a bounded search, but must scan the table instead.

- Avoid LIKE *`hostvar`* when *`hostvar`* can represent anything.

  SQL assumes the worst case, "%...", and a full table scan is performed.

How the Optimizer Processes Join Operations

Choosing an execution plan for queries involving a join of two or more tables is an extension of the process of choosing plans for single-table queries. In addition to determining how to access a table before the join, the optimizer evaluates different ways to join the tables.

The optimizer evaluates how each of these four join strategies would perform for the query:

- Nested join (also called nested-loop join)
- Sort merge join
- Key-sequenced merge join
- Hash join

Except for the nested join, all other join strategies require the existence of an equijoin predicate, which relates columns using an equal (=) comparison operator.

The following subsections describe the four join methods, followed by a description of how the optimizer compares and evaluates the different types of join strategies for a query. In this discussion, an outer table is a table that is examined before another table. An inner table is a table examined after the outer table.

Two tables can be joined even if there are no joining predicates. In this case, SQL creates the new table by concatenating every row in one table with every row in the other table, using a nested join strategy.

Nested Join

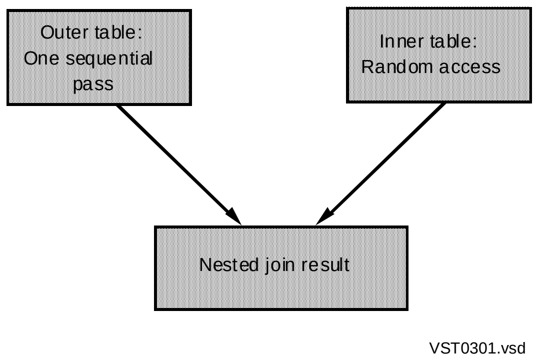
The nested join method retrieves rows one-at-a-time from the outer table and compares them with the rows in the inner table. If index predicates are specified for the inner table, they are applied before accessing the inner table.

This method retrieves the rows from the inner table that satisfy the join predicate and concatenates them with the corresponding rows from the outer table.

The nested join method uses VSBB. SQL sets the begin and end keys to the same value. Therefore, the buffer contains only the rows that match the current input. The key-sequenced merge join uses VSBB differently. For more information on the key-sequenced merge join, see Key-Sequenced Merge Join on page 3-27. For information on VSBB, see Section 4, Improving Query Performance With Environmental Options.

Figure 3-1 shows a nested join.

Figure 3-1. Nested Join



For information on forcing a nested join, see the *SQL/MP Reference Manual.* Also, see Specifying a Join Method on page 3-43.

Sort Merge Join

The sort merge join method requires that the joining columns of the outer and inner tables exist in ascending or descending order. The sort merge join method is used only for equijoin queries (queries in which the operator in the join predicate is an equal (=) comparison operator).

The inner table is always sorted into a temporary entry-sequenced table prior to performing the join. The optimizer uses one column for the sort. From potential equijoin predicates, it chooses the column with the lowest selectivity. That is, it picks the column that it expects to have the fewest matches between values in the inner and outer table rows.

If the outer table is not already sorted on the joining column, the table is sorted into a temporary entry-sequenced table. The join is done between the temporary sorted inner table and the outer table.

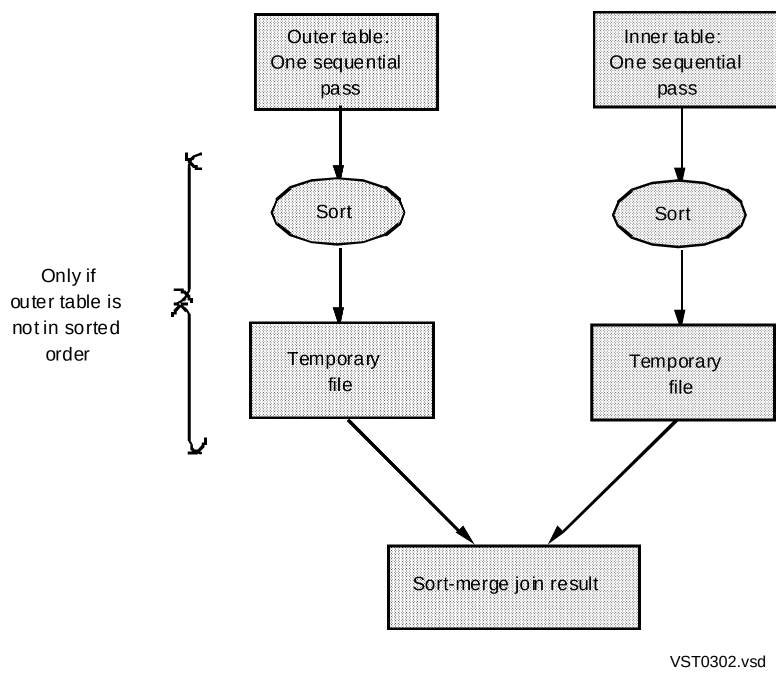
A row is retrieved from the outer table, another row is retrieved from the inner table, and the values of the join columns for the two rows are compared as follows:

1. If the values are the same, the rows are concatenated, projected, and returned to the user, and the position of this inner row is stored in memory.
2. The next inner row is retrieved, and the process is repeated until the join-column values of the inner and outer table rows are different.
3. The next row is then retrieved from the outer table:
    a. If the join-column value is the same as that in the previous outer row, the inner table is restored to its original position, and the process is repeated from Step 1.
    b. If the join-column value of the inner row is less than that of the outer row, the next inner row is retrieved until the value of the inner row is greater than or equal to that of the outer row.
    c. If the join-column value of the inner row is greater than that of the outer row, the next outer row is retrieved until the outer row has a value greater than or equal to that of the inner row.
    d. If the join column values of the outer row and inner row are equal, then the process is repeated from Step 1.

SQL makes one pass through the inner table, with possible limited looping within a value range. The performance difference between a nested join and a sort merge join is the difference between random access with possible repeated access to the same pages needed for a nested join and the cost of sort operations needed for the sort merge join.

Steps 1 through 3 are repeated until all rows from the outer table have been examined.

Figure 3-2. Sort Merge Join


Outer table: One sequential pass
Inner table: One sequential pass
Sort
Sort
Only if outer table is not in sorted order
Temporary file
Temporary file
Sort-merge join result
VST0302.vsd

For information on forcing a merge join, see the *SQL/MP Reference Manual.* Also, see Specifying a Join Method on page 3-43.

Key-Sequenced Merge Join

The key-sequenced merge join method can apply a merge-join method to tables without doing a sort or using a temporary file. A key-sequenced merge join involves an outer composite and an inner table. An outer composite is either a single outer table or the result of any joins that have occurred so far. The operator in the join predicate must be an equal (=) operator, and one of these conditions must be met:

- The tables must be in the same order.
- If rows from the inner table must be accessed by index, then the outer table of the join must be in the same sequence as one of the indexes of the inner table.

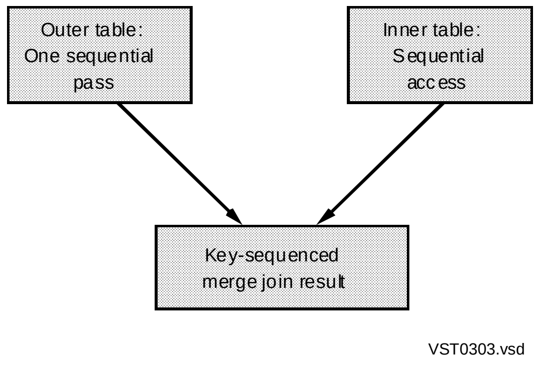
The key-sequenced merge join has two advantages over a sort merge join: it does not require a sort on one or both tables and it is not limited to a single column.

The read process for the key-sequenced merge join is the same as that for the sort merge join, except for the third step. A row is retrieved from the outer table, another row is retrieved from the inner table, and the values of the join columns for the two rows are compared as follows:

1. If the values are the same, the rows are concatenated, projected, and returned to the user, and the position of this inner row is stored in memory.
2. The next inner row is retrieved and the process is repeated until the join-column values of the inner and outer table rows are different.
3. The next row is then retrieved from the outer table:
    a. If the join-column value is the same as that in the previous outer row, the inner table is restored to its original position, and the process is repeated from Step 1.
    b. If the join-column value of the inner row is less than that of the outer row, then the following occurs:
        1. A limited number of rows are read until the value of the inner row is greater than or equal to that of the outer row. (The optimizer chooses the limit.)
        2. If the limit is reached before SQL finds such an inner row, the inner table rows are skipped and the outer row value is then used as an index into the inner table.
    c. If the join-column value of the inner row is greater than that of the outer row, the next outer row is retrieved until the outer row has a value greater than or equal to that of the inner row.
    d. If the join column values of the outer row and inner row are equal, then the process is repeated from Step 1.

When a left join uses a key-sequenced merge join, a null-augmented row is created for the inner table when no match occurs. Then a random position takes place on the inner table, using the next new outer row key.

Figure 3-3. Key-Sequenced Merge Join


Outer table: One sequential pass
Inner table: Sequential access
Key-sequenced merge join result
VST0303.vsd

The key-sequenced merge join method differs from a nested join in the way that records are retrieved from both the outer and inner tables. The key-sequenced merge join method may read several rows on the outer table before retrieving a row from the inner table. The nested join method reads only one row from the outer table before accessing the inner table.

A key-sequenced merge join uses virtual sequential block buffering (VSBB) when it sequentially retrieves rows from the inner table. A nested join randomly accesses rows on the inner table, using separate retrievals for each outer row.

Even though the key-sequenced merge join method can reposition and reread rows on the inner table, this method can still be more efficient than the nested join. When the key-sequenced merge join uses VSBB, it sets the end key to the maximum possible value. As a result, more data can be in the buffer than is actually needed. On subsequent retrievals, the data will already be in the buffer. This result is not true for the VSBB used in the nested join. For more information on VSBB and the nested join, see Nested Join on page 3-25. For information on VSBB, see Section 4, Improving Query Performance With Environmental Options.

The DISPLAY STATISTICS for a key-sequenced merge join can be misleading because the counts for records accessed and for records used can be greater than those in a nested join.

For information on forcing a key-sequenced merge join, see the *SQL/MP Reference Manual.* Also, see Specifying a Join Method on page 3-43.

Hash Join

Hash methods eliminate sort operations for a join operation. The hash-join strategy is considered an asymmetric algorithm because only the inner table is stored in memory.

Hashing is most efficient when the inner table can fit entirely into memory and is ideal for joining a large table with a small table when the joining columns are not key columns. The size of a large outer table does not influence the amount of memory needed—an advantage over sort merge joins, which sometimes must sort the large table.

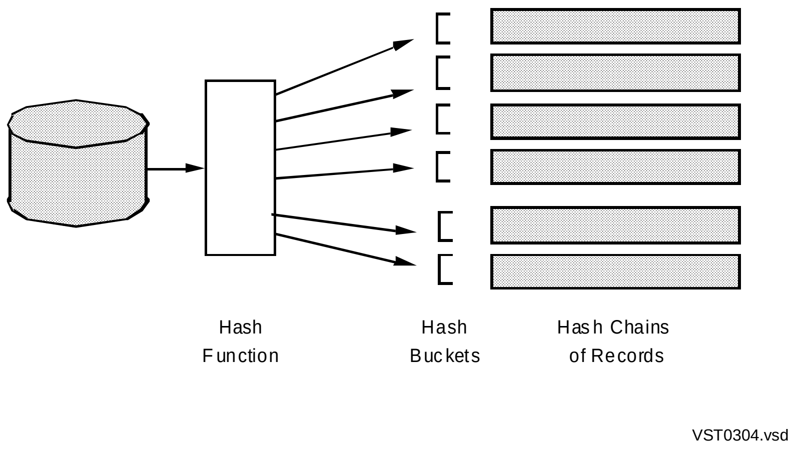
A hash join uses a hashing function, rather than indexes or sequential reads, to access specific rows in a file. In general, the process has two phases:

1. The build phase reads the inner table of the join into virtual memory and builds an in-memory hash table for it, using a hashing function based on join attributes.

   The hashing function calculates indexes into an array of values in main memory. All equal values go to the same table entry and are chained by linked list; other values can also hash to the same array entry. The array is divided into buckets, each of which is small enough to fit into main memory. If the join is partitioned, the arrays are distributed across partitions.

   Figure 3-4 shows the in-memory hash chain structure for an inner table.

Figure 3-4. Hash Function Example



2. The probe phase reads the outer table sequentially. For each outer (probing) row, it accesses matching rows in the hashed inner table and generate result rows.

SQL supports two types of hash joins, simple and hybrid:

- A simple hash join is fastest when the inner table of the join operation fits entirely within memory.
- A hybrid hash join increases performance when the inner table is much larger than available memory.

SQL supports hashing for multiple join columns. This strategy allows efficient hashing when more than one column is specified in the join predicate.

SQL does not support hash joins for columns that use collations.

For information on forcing a hash join, see the *SQL/MP Reference Manual.* Also, see Specifying a Join Method on page 3-43.

The following subsections describe sequential and parallel operation of each type of hash join.

Simple Sequential Hash Join

To execute a simple sequential hash join, the executor performs two steps.

1. The executor makes a single pass over the inner table of the join, applying selection and projection operations (if possible) to eliminate unnecessary rows and columns of the inner table prior to the actual join operation. The executor stores the remaining portion of the inner table in memory, building an in-memory hash table using the smaller table hashed on the join attribute. (The join attributes are used as a hash key.)
2. The executor then makes one pass over the outer table of the join, applying selection and projection operations if possible. For each row of the outer table that remains after selection and projection, the executor obtains the corresponding row or rows of the inner table (as in a nested join). The executor joins matching rows and delivers them to the next stage of the query execution process.

Instead of sorting, the inner table is read into a memory-resident hash table. The outer table is processed like the outer table of a nested join. Building a hash table is faster than sorting, so hash joins have better performance as long as enough memory is available.

The order of the outer table is preserved during a simple sequential hash-join operation.

Hybrid Hash Join

A hybrid hash join handles overflow situations when the smaller table does not fit entirely in physical memory. The hybrid join retains as much as possible of the inner table in memory, but divides rows of both inner and outer tables into buckets, or clusters, that reside on disk and can be processed in memory.

If the outer table is actually a composite of more than one table, SQL must write it entirely to disk and complete the previous join before it can begin the current hybrid hash join. This is an expensive operation.

The hybrid hash join does not preserve ordering, because clusters of data might be written to disk. Each cluster is in order, but the concatenation of them is not in order. This situation affects performance when the query requests subsequent orderings such as an ORDER BY request.

Parallel Hash Join

The optimizer considers a parallel hash-join strategy if CONTROL EXECUTOR PARALLEL EXECUTION ON is specified for the query. SQL supports three types of parallel hash joins:

- Plain parallel
- Repartitioned parallel hash
- Matching partitions

Plain Parallel

SQL can execute hash joins across processors and across disk partitions. A parallel hash join uses existing partitions and follows these steps:

1. The executor starts an ESP for each partition of the inner table.
2. The executor starts an ESP for each partition of the outer table. (The outer table must have more than one partition.)
3. Each inner ESP reads its partition, applies selection and projection criteria, and sends the results to the outer ESPs.
4. Each outer ESP receives a copy of the inner ESP results, stores them in a hash structure in memory, and reads its own partition in a single pass to perform the join. For each row, the matching rows are searched in the hash table and result rows are generated.

Thus, a large join can be split into smaller joins and executed in parallel, and, perhaps more importantly, the multiple processes can take advantage of a greater amount of main memory. An inner table that might not fit into memory in a single processor might fit into memory when divided into smaller pieces and directed to use multiple processors

Note. If your system has limited process resources, the number of ESP processes might cause a performance reduction. In such a case, if you plan to use parallel hash joins, you might consider defining fewer partitions for the outer table so that fewer ESPs are started up.

For a two-table parallel hash join, a hybrid parallel hash join is used for the parallel hash join. For a join with three or more tables, where parallel hash join is selected for the third table, a simple hash join is selected.

As with all parallel plans, order is not preserved after the join operation.

Repartitioned Parallel Hash Join

A repartitioned parallel hash join reads from existing partitions and repartitions the data across all processors, using these steps:

1. The executor starts an ESP for each partition of the inner table.
2. The executor starts an ESP for each partition of the outer table. (The outer table must have more than one partition.)
3. The executor starts an ESP in each processor on the local system to join repartitioned inner and outer rows.
4. Data from the inner table is read and repartitioned into ESP processes for each processor in the system. The inner table (usually the smaller table) is divided into partitions by applying a hash function to the join attributes and computing the partition number. Each partition resides on its own processor.
5. Data from the outer table is read and repartitioned into ESP processes for each processor in the system. It is repartitioned in the same way the inner table was. After the outer row is repartitioned and sent to an ESP, a hash join is done.

Thus, the parallel repartitioned hybrid hash join uses three levels of hashing:

- Dividing tables into partitions (for parallelism)
- Dividing tables into buckets, or clusters (for overflow)
- Optimizing access and comparison overhead (base level of hashing)

The repartitioned parallel hash join always uses a hybrid hash join strategy (as opposed to a simple hash join). In general, the repartitioned parallel hash join is more efficient than a parallel sort merge join because sorting is avoided.

As with all parallel plans, order is not preserved after the join operation.

Matching Partitions

A matching partitions hash join has inner and outer tables that are keyed and partitioned the same way. This join method uses only one set of ESPs. The method works best when most of the rows need to be scanned to perform the join. These are the steps:

1. The executor starts an ESP for each partition of the outer table.
2. Each ESP reads the corresponding inner table partition and hashes the inner rows into memory.
3. Each ESP reads its outer table partition sequentially, applies the selection and projection criteria, hashes the key, and checks the inner hash table for a match.
4. Matches are passed to the next step of the query.

As with all parallel plans, order is not preserved after the join operation.

Determining a Join Strategy

When evaluating join methods, the optimizer looks at ways to join tables, both with structure and ordering of combinations of tables and with various join strategies. The goal is to minimize processor time, disk access, sorts, and other resource use so that resource consumption is minimized, and performance is as fast as possible.

The optimizer evaluates performance for each access path to each table.

The optimizer uses a cost model to evaluate join strategies and chooses the strategy with the lowest estimated cost. (For information about cost, see Section 5, Selectivity and Cost Estimates. To list the chosen strategy, see EXPLAIN output, described in Section 6, Analyzing Query Performance.)

Combining Tables

When two tables are joined, SQL forms a composite table. For example, if two tables, T1 and T2, are joined, SQL forms a composite table, T1 JOIN T2, in the course of evaluating the query.

The number of different ways to join a set of tables increases exponentially as the number of tables increases, so SQL reduces the number of possibilities when possible.

When joining more than two tables, SQL follows these steps:

1. Joins tables a pair at a time. SQL considers only two-way joins that involve either two tables or a composite table and a table that does not already belong to the composite.
2. Evaluates joins between the composite table and all single tables that have not yet been added to the composite. For each single table, associate all join predicates that relate the single table with tables that are already in the composite.
3. Generates plans for types of join strategies (nested, sort merge, key-sequenced merge, and hashed).
4. When the composite table contains all tables that need to be joined, chooses the plan with the least cost. (For information about cost, see Section 5, Selectivity and Cost Estimates.)

This pairwise strategy reduces the number of combinations that must be examined and simplifies evaluation. For example, if tables T1, T2, T3, and T4 are to be joined, this combination is considered:

```
(((T1 JOIN T2) JOIN T3) JOIN T4)
```

This combination is not considered, because it joins two composite tables:

```
((T1 JOIN T2) JOIN (T3 JOIN T4))
```

The number of combinations is also reduced by discarding more expensive combinations that give the same order to result rows. For example, suppose that the EMPLOYEE table has EMP_NAME as the primary key and an index exists on the

DEPT_NUM column. The DEPT table has DEPT_NUM as the primary key. this query asks for employee and department information:

```
SELECT EMP_NAME, DEPT_NAME, SALARY
   FROM EMPLOYEE, DEPT
   WHERE EMPLOYEE.DEPT_NUM = DEPT.DEPT_NUM ;
```

The information is retrieved by joining the EMPLOYEE and DEPT tables. There are several ways to join the tables; for example:

- (EMPLOYEE with primary key JOIN DEPT)
- (EMPLOYEE with index JOIN DEPT)
- (DEPT JOIN EMPLOYEE with primary key)
- (DEPT JOIN EMPLOYEE with index)

The order in which the rows are presented depends on the access path used for the outer table. For choice 1, the rows are presented in EMP_NAME order. For choices 2, 3, and 4, the rows are presented in DEPT order.

If another table, JOB, is to be joined with the preceding result table, SQL considers only the composite from choice 1 or the composite from the least expensive of choices 2, 3, and 4. This strategy reduces the number of combinations to be joined with JOB from four to two. In general, SQL discards all but the least expensive of the combinations for a given order.

Combining Tables for Hash Join Operations

When evaluating a hash join operation, SQL attempts to choose the smaller of two tables as the inner table so the table is more likely to fit into memory.

Forming Cross Products

If you specify multiple tables in the FROM clause without a join predicate or with a join predicate that is always true, SQL forms a Cartesian product (or cross product) by concatenating each row of each table with every row of every other table. This strategy produces a set of composite rows that contains all possible concatenations of a row from the first table with a row from the second table.

SQL performs this operation whenever there is a join without an equality predicate between a table or composite table and another table. For queries with several tables, the process might be repeated several times.

A Cartesian product involving two tables, one of size M and the other of size N, is of size M x N. If the tables are large, the performance overhead can be quite costly. Therefore, in most situations, SQL does not form a Cartesian product unless a single table does not have a join predicate connecting it to the composite table. If none of the remaining single tables has a join predicate connecting it to the composite, then the optimizer considers a Cartesian product between the composite and each of the remaining tables.

In some situations, SQL forms a single Cartesian product that significantly reduces the number of rows to be joined.

Consider this example that creates three tables and then selects data from the tables:

```
CREATE TABLE buildng (a INT, m INT, n INT, KEY a) ;
CREATE TABLE room (x INT, b INT, KEY x) ;
CREATE TABLE locatn (c INT, d INT, e INT, f INT, KEY (c,d)) ;

SELECT * from buildng, room, locatn
  WHERE buildng.a, room.b = locatn.c,locatn.d
  AND   room.b = 5
  AND   buildng.m > 20 ;
```

Suppose that the predicates produce these results:

- `buildng.m > 20` produces 100 rows
- `room.b = 5` produces 10 rows
- The first key of locatn produces 1,000 rows for each equal relation, but both keys (c and d) yield exactly one row

Without an intermediate cross product, SQL would join buildng to locatn (as in `buildng.a = locatn.c`) and then join the composite with room. The join between the composite and room would involve 100,000 rows.

With a cross product between buildng and room, however, before the join with locatn, only 1,000 rows remain after the cross product and only 1,000 rows are read from table locatn, resulting in a substantial reduction in query time.

SQL forms Cartesian products for both serial and parallel execution plans; for parallel plans, browse access or TABLE LOCK ON must be specified. For more information, see Section 4, Improving Query Performance With Environmental Options and the *SQL/MP Reference Manual*.

Relative Performance of Join Strategies

For each two-way join, SQL considers four join methods: nested, sort merge, key-sequenced merge, or hash.

If M and N are the number of rows in two joined tables, then the increasing resource cost of joins is roughly calculated as shown in Table 3-3.

Table 3-3. Calculation of Resource Costs for Joins

Join Strategy	Order of Cost
Nested Join	M * N, with no indexes
Sort Merge Join	N log N
Key-Sequenced Merge Join	N
Hash Join	N, with a minimum amount of memory

For each join performed within the join strategy listed in the “Join Strategy” column, the cost increases by the algorithm shown in the “Order of Cost” column. Various factors influence these general rules.

Table 3-4 compares join strategies.

Table 3-4. Comparison of Join Strategies (page 1 of 2)

Element of Comparison	**Nested Join**	**Sort Merge Join**	**Key-Sequenced Merge Join**	**Hash Join**
Requirements for join strategy	None	Query must include an equality operation as part of the join predicate.	Query must include an equality operation as part of the join predicate. Join predicate must include the leading key columns of the inner table. Inner and outer tables must be in the same order.	Query must include an equality operation as part of the join predicate.
Processor overhead	If no indexes are available, cost increases with the number of rows.	Cost increases by approximately *n* log *n* with the number of rows in the smaller table.	Cost increases by approximately *n,* with the number of rows in the larger table.	Cost increases by approximately *n,* with the number of rows in the smaller table.
Sort requirements	None	Always requires one sort for the inner table. If values in the joining columns are not in the same ascending or descending sequence, the optimizer might choose to sort the outer table as well and then perform the sort merge join.	Does not require a sorting operation. This can be especially helpful if a sort is not needed for an ORDER BY or GROUP BY operation in the query.	Does not require a sorting operation. This can be especially helpful if a sort is not needed for an ORDER BY or GROUP BY operation in the query.

Table 3-4. Comparison of Join Strategies (page 2 of 2)

Element of Comparison	**Nested Join**	**Sort Merge Join**	**Key-Sequenced Merge Join**	**Hash Join**
Hash requirements	None	None	None	All hash joins hash the inner table. Hybrid hash joins also hash the outer table.
Use of MDAM	Can be used on inner and outer tables.	Can be used on outer table.	Can be used on outer table.	Can be used on outer table.
Scan requirements	Typically scans entire inner table for each row of the outer table if join columns are not primary key columns or alternate key columns of the inner table	Inner table typically scanned once, but might be scanned once per row of the outer table, in the worst case scenario.	Inner table scanned once.	Inner table scanned once.
Use of join predicates as begin and end keys	Can use join predicates as begin and end key predicates if join columns are primary or alternate key columns of the inner table. In this manner, each row of the outer table provides starting and stopping values for keyed retrieval from the inner table.		Can use join predicates as begin and end key predicates if join columns are primary or alternate key columns of the inner table. Balances keyed and sequential retrieval. Rows from the outer table provide starting values for keyed retrieval from the inner table.	Cannot use join predicates as begin and end keys. Must read the entire inner table unless there are additional nonjoining key predicates to limit the scan. Single pass and sequential I/O might, however, still make the hash join more efficient.
Order of results	Preserves ordering of outer table.	Produces the join result in order of the join attribute, which can eliminate a later sort operation.	Preserves ordering of outer table.	Simple join preserves ordering of the outer table.

In general, the optimizer chooses a hash join in preference to a sort merge join for situations where an equality predicate exists for the join operation. The performance advantage of the hash join increases with increasing size difference between input tables. This advantage decreases when data is not distributed uniformly.

The choice between the hash join and the key-sequenced merge join is not so simple. Both joins scan the inner table at almost the same speed. The major difference is that the hash join spends time building or probing the hash table, while the key-sequenced merge join spends time evaluating the join predicates against all the rows qualified by the scan. However, the key-sequenced merge join has two main advantages:

- The key-sequenced merge join has no space requirement for preprocessing the inner table, while the hash join needs to allocate space for it.
- Unlike the hash join, the key-sequenced merge join does not have to send the hashed tables to all of the outer executor server processes. Therefore, a key-sequenced merge join does not incur this extra message cost.

Also, uneven data distribution can lessen performance by causing differential overflow of hash clusters; this effect can be especially influential when selectivity is low. (For information about selectivity, see Section 5, Selectivity and Cost Estimates.)

If tables are partitioned similarly on joining columns, a parallel nested join is used in preference to a parallel hash join.

The existence of an index on each column of a join predicate also influences the selection of join technique.

Writing Efficient Joins

The optimizer determines efficient join strategies for various combinations of tables and join methods (described in How the Optimizer Processes Join Operations on page 3-24). There are five ways to assist this process:

- Specifying predicates for smaller tables of the join
- Using indexes for the join
- Eliminating implicit joins
- Adding join predicates
- Using joins instead of subqueries

The first option minimizes the amount of data returned by the join; the other options extend the methods available to SQL for optimizing the operation.

This subsection describes two CONTROL directives that force the optimizer to evaluate joins in a specific way. These options should only be used with extreme caution and thorough knowledge of join operations.

If your query requires a join, first make sure there is nothing missing from the query, such as predicates on key columns.

Using Indexes

The use of an index improves join performance by eliminating sort operations. In situations where the fastest possible response time is required, do not specify joins where no index exists. If you are not sure which columns are keys or indexes, check with your database administrator.

Eliminating Implicit Joins

When tables are joined, each new row is formed by concatenating two rows, one from each of the original tables. The paired rows must have the same value in the joining column. You do not need to specify a join predicate for a join, but the use of a predicate can improve the performance of the join operation.

If you specify multiple tables in the FROM clause without a search predicate, SQL forms a Cartesian product (or cross product) by concatenating, in turn, each row of each table with every other row of every other table. This strategy is known as an implicit join.

A Cartesian product involving two tables, one of size M and the other of size N, is of size M x N. If the tables are large, the performance overhead can be quite costly.

By specifying a join predicate, you can

- Eliminate costly scans of multiple tables
- Provide a greater choice of plans for SQL in these ways:
  - A greater number of predicates provides a greater number of ways that SQL can evaluate possible join combinations.
  - SQL does not consider the efficient hash or merge joins unless an equality predicate is specified for the join operations.
- Reduce the size of the result and, consequently, the amount of work done by SQL to produce the result

In this example, a join is created between the PARTS and ODETAIL tables. Specifying ODETAIL O, PARTS P in the FROM clause creates an implicit join; however, because no join column is specified, a Cartesian product is produced:

```
SELECT ORDERNUM
  FROM ODETAIL O, PARTS P
  WHERE O.PARTNUM = 5100
  AND QTY_ORDERED <
    (SELECT AVG(QTY_AVAILABLE)
      FROM PARTS
      WHERE P.PARTNUM = 5100) ;
```

The query returns this result:

```
ORDERNUM
----------
```

```
    100210
    300350
    600480
    800660

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

To eliminate the unnecessary join and include a join predicate, therefore eliminating the Cartesian result, rewrite the query as follows:

```
SELECT ORDERNUM
  FROM ODETAIL O
  WHERE O.PARTNUM = 5100
  AND QTY_ORDERED <
    (SELECT AVG(QTY_AVAILABLE)
      FROM PARTS P
      WHERE P.PARTNUM = 5100) ;
```

For a cost analysis of both formulations of the query, see Section 5, Selectivity and Cost Estimates.

Adding Join Predicates

Beyond the advantage of a single join predicate, additional predicates can increase the choice of execution plans available to SQL without changing the meaning of the query. Consider this query:

```
SELECT *
   FROM T1,T2,T3
   WHERE T1.A = T2.B
      AND T2.B = T3.C
```

Adding this predicate can increase the combinations available to SQL:

```
      AND T1.A = T3.C
```

Using Joins Instead of Subqueries

In general, look for ways to formulate your query with join operations instead of subqueries. The use of join operations can reduce I/O operations, reduce message traffic, and increase the flexibility with which SQL can choose an execution plan.

When you use a subquery, you direct SQL to perform the subquery first and then perform the main query. SQL must process two SELECT statements to obtain the result.

If the select statement in the subquery produces unique results (no duplicate rows), then the subquery can be transformed into a join. This restriction applies because even though a subquery can have duplicate rows, each row of the outer table can produce at most one result row. If, however, a subquery with duplicate rows is converted into a join, then each row of the outer table can produce multiple result rows, which could change the result set.

These examples show how to reformulate subqueries into join queries. Consider this query, which contains a subquery:

```
SELECT employee.name
   FROM employee
   WHERE employee.dept_no IN
     (SELECT dept.dept_no FROM dept
      WHERE dept.name = "development")
   ORDER BY employee.name ;
```

You can change this query into a join query as follows:

```
SELECT employee.name
  FROM employee, dept
  WHERE employee.dept_no = dept.dept_no
  AND dept.name = "development"
  ORDER BY employee.name ;
```

Although both formulations produce the same result, their performances are likely to be very different: the second formulation always matches or outperforms the first one. The use of the join minimizes I/O operations for separate SELECT statements. In addition, in the first formulation, the user has dictated how the query is to be performed (perform the subquery first and then perform the main query). In the second formulation, SQL can determine the order of the join and is therefore able to choose the most efficient way to execute the query.

For another example, this noncorrelated subquery:

```
SELECT emp_id, first_name, last_name, mgr_id
   FROM employee emp1
   WHERE emp_id IN (SELECT mgr_id
   FROM employee emp2) ;
```

can be reformulated as a self join in this way:

```
SELECT emp1.emp_id, emp1.first_name,
   emp1.last_name, emp1.mgr_id
   FROM employee emp1, employee emp2
   WHERE emp2.mgr_id = emp1.emp_id ;
```

The second query performs better because SQL obtains the entire result with a single SELECT operation rather than two SELECT operations.

A correlated subquery allows you to take advantage of the EXISTS predicate, however. If you just want to know whether some condition actually exists in the database or not, a join can be more costly. Do not discard correlated subqueries unconditionally.

You can tell the type of subquery by looking at EXPLAIN output. For a correlated subquery, the EXPLAIN utility lists “Executes once for each row retrieved.”

For more information about correlated and noncorrelated subqueries, see Section 1, Retrieving Data: How to Write Queries.

Specifying a Join Method

Two directives control the selection of join method:

- The CONTROL QUERY HASH JOIN option specifies whether SQL can use hash joins if the optimizer expects hash joins to improve query performance.
- The CONTROL TABLE JOIN METHOD option specifies the join method SQL uses when the specified table is the inner table of a join operation. Options include NESTED, MERGE, KEY SEQUENCED MERGE, and HASH. If you choose not to specify a join method, SQL selects an appropriate method for each join of the table or tables that you specify in the CONTROL TABLE directive.

For more information on these directives, see the *SQL/MP Reference Manual*.

CONTROL QUERY HASH JOIN Option

A sample CONTROL QUERY HASH JOIN directive follows:

```
CONTROL QUERY HASH JOIN ENABLE ;
```

This directive ensures that SQL considers hash joins for subsequent queries.

You should usually leave the HASH JOIN option set to SYSTEM or ENABLE, because the optimizer is designed to select a hash join only if the resulting plan improves the performance of your query.

You might, however, choose HASH JOIN OFF if you know that memory contention is severe in the processor or processors that run your query.

For more information, see the *SQL/MP Reference Manual*.

CONTROL TABLE JOIN METHOD Option

The JOIN METHOD option applies only to SELECT and INSERT-SELECT statements, and can be specified with other CONTROL options. Sample directives are:

```
CONTROL TABLE EMPLOYEE JOIN METHOD NESTED ;
CONTROL TABLE * JOIN METHOD SYSTEM ;
```

When specifying a merge join or hash join, the query must contain an equijoin predicate between the two tables.

If you specify JOIN METHOD HASH and the join involves columns with collations, SQL returns an error.

Join method does not affect the first table in a join sequence (the table that ends up as Step 1 of the query plan). Step 1 is always a scan operation; the remaining steps are join operations. (For more information see, Specifying a Join Sequence on page 3-45.)

△ **Caution.** The JOIN METHOD option overrides the optimizer's standard cost estimates (described in Section 5, Selectivity and Cost Estimates), and therefore might cause performance degradation instead of enhancement. To use this option, you must have a thorough understanding of the optimizer. Use it *only* if the optimizer does not produce the optimal plan.

If you suspect that you might benefit from the use of one of these options, check your application with and without the CONTROL option. To check your application, run Measure, a performance measurement tool for NonStop systems, on your production data. This tool enables you to collect and examine performance statistics.

If you use one of the CONTROL options, you might want to change this directive later for reasons such as these:

- 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 any occurrences of it easy to find and change, using one or more of these alternatives:

- Make sure the directive only applies to the statement and table intended. Return the specified table to SYSTEM method directly after the statement; for example:

  ```
  CONTROL TABLE * JOIN METHOD SYSTEM
  ```

- Isolate this directive in its own section and perform it from the inline application code.
- Place all statements affected by this directive in separate modules, called as services by other modules.

Confirm all use of this option with data from the Measure product and verify its use periodically to account for changes in data distributions and volumes. Reevaluate its effectiveness with each new version of SQL.

Combining HASH JOIN and JOIN METHOD Options

If JOIN METHOD SYSTEM and CONTROL QUERY HASH JOIN OFF are both specified, SQL never selects a hash join.

If JOIN METHOD HASH is specified, it overrides any setting of the CONTROL QUERY HASH JOIN option for the specified table.

Specifying a Join Sequence

The CONTROL TABLE JOIN SEQUENCE option controls the order in which tables are combined in a join operation. You specify access by defining an ordinal position for a table. The statement accepts integers; the table associated with the number one is the first table processed. For example, JOIN SEQUENCE 1 forces SQL to assign the specified table to the first step of the join operation (the outermost loop).

The JOIN SEQUENCE option applies only to SELECT and INSERT-SELECT statements and can be specified with other CONTROL options. A sample directive follows:

```
CONTROL TABLE EMPLOYEE JOIN SEQUENCE SYSTEM ;
```

If you specify the SYSTEM option, SQL chooses the join sequence SYSTEM (default) option. If, for example, you specified CONTROL TABLE EMP1 JOIN SEQUENCE 1 in an SQLCI session, use CONTROL TABLE EMP1 JOIN SEQUENCE SYSTEM (for table EMP1) or CONTROL TABLE * JOIN SEQUENCE SYSTEM (for all tables in subsequent queries) to restore the default join sequence mechanism.

△ **Caution.** The JOIN SEQUENCE option overrides the optimizer's standard cost estimates (described in Section 5, Selectivity and Cost Estimates) and therefore might cause performance degradation instead of enhancement. To use this option, you must have a thorough understanding of the optimizer. Use it *only* if the optimizer does not produce the optimal plan.

If you suspect that you might benefit from the use of one of these options, check your application with and without the CONTROL option, using actual Measure statistics from production data.

You might want to change this directive later for reasons such as these:

- 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 SQL enhancements
- Changes to the database structure (such as dropping an index) can require recompilation when the option is in use

Therefore, make any occurrences of it easy to find and change, using one or more of these alternatives:

- Make sure the directive only applies to the statement and table intended. Return the specified table to SYSTEM sequence directly after the statement.
- Isolate this directive in its own section and perform it from the inline application code.
- Place all statements affected by this directive in separate modules, called as services by other modules.

Confirm all use of this option with data from the Measure product and verify its use periodically to account for changes in data distributions and volumes. Reevaluate its effectiveness with each new version of SQL.

How the Optimizer Processes Aggregates and Group-By Operations

SQL can use these strategies to evaluate aggregates and GROUP BY operations, listed from most to least optimal:

- MIN/MAX optimization
- Evaluation by the disk process
- Evaluation by the executor, without sorting
- Hashed aggregation and grouping
- Sorted GROUP BY operation

In addition, for partitioned tables, SQL can process aggregations in parallel. To do this, SQL starts an ESP for each partition. Each server process reads its corresponding partition, processes the partial aggregate, and sends it to the master ESP. If partitions are distributed across a network, this strategy can reduce the amount of network traffic as well as interprocessor traffic.

EXPLAIN output lists the location of aggregate and GROUP BY evaluation (disk process or executor).

These paragraphs describe each type of evaluation strategy.

MIN and MAX Optimization

The processing of MAX and MIN functions usually requires reading the entire table. If, however, all of these conditions are met for a query, SQL reads only one row when evaluating a MAX or MIN function:

- The FROM clause names only one table.
- The select list contains only one function.
- The column operand of the MIN or MAX function is in the begin key or the end key.

For example, consider this query. Suppose that there is an index on (A, B, C):

```
SELECT MAX(C) FROM T
   WHERE A = 10 AND B = 20 ;
```

This query can be evaluated by reading a single row from the index, even though there is no predicate on C.

If the begin-key and end-key predicates contain only equal comparisons (=), then the MIN or MAX processing of the next key column in sequence, which is not part of the begin or end key, is also evaluated by reading only one row from the index.

MAX and MIN functions are not optimized when

- The operand of the MAX or MIN function is an expression; for example:

  ```
  MAX(-C)
  ```

- The query specifies a GROUP BY or a HAVING clause.
- The WHERE clause contains any of these subqueries:
  - A correlated subquery
  - An ANY, SOME, or ALL subquery
  - An EXISTS subquery
- The WHERE clause contains any predicate connected by an OR operator that might use a keyed access.
- Both MAX and MIN are requested in one SELECT statement; for example:

  ```
  SELECT MIN(A), MAX(B) FROM T ;
  ```

  In such cases, you should write two SELECT statements.

If an index exists on the column that is an argument of the MIN or MAX function and the column is the prefix of the index, the executor component can read the first or last row to retrieve the MIN or MAX value; for example:

```
SELECT MIN (RETAIL_PRICE)
 FROM INVNTRY
```

Assuming that RETAIL_PRICE is the first key column of an index, SQL reads only the first row of the index to find the minimum value.

Evaluation by the Disk Process

The disk process can evaluate these:

- Aggregate functions AVG, COUNT, MAX, MIN, and SUM
- The GROUP BY clause

When an aggregate or GROUP BY operation can be evaluated by the disk process, the disk process scans the data and returns only the aggregated or grouped values to the file system. This can reduce the number of messages sent between the file system and the disk process.

The requirements for disk process aggregation depend on whether there is a GROUP BY clause in the query.

If there is no GROUP BY clause, disk process aggregation is selected if these conditions are true:

- The query does not contain any executor predicates (except HAVING); for example:

  ```
  SELECT SUM(SALARY)
    FROM EMPDATA
    WHERE SALARY >= 50000;
  ```

- The query uses primary index access or index-only access; that is, all columns referenced by the query can be found in the index.

- If the query includes a join operation, aggregation must be on the innermost table of the join, as follows:

  ```
  SELECT SUM(EMPDATA.SALARY)
    FROM DEPT, EMPDATA
    WHERE DEPT.DEPTNO = EMPDATA.DEPTNO ;
  ```

  Note: If a left join operator is present with a COUNT (*) operation, the executor process must perform null augmentation prior to aggregation.

If there is no GROUP BY clause, then either serial or parallel processing might occur.

If there is a GROUP BY clause, then grouping and aggregation are done by the disk process only if certain additional conditions are satisfied. These additional conditions vary, depending on whether the user has requested parallel processing. You can use the =_SQL_CMP_PARALLEL DEFINE or a CONTROL EXECUTOR directive to request the optimizer to consider a parallel plan. (For information on DEFINES, see the *SQL/MP Reference Manual.* For information on the CONTROL EXECUTOR directive, see Section 4, Improving Query Performance With Environmental Options and also the *SQL/MP Reference Manual.*) If neither the CONTROL EXECUTOR directive nor the DEFINE is used, the default is a serial plan.

VSBB is disabled for disk-process aggregation.

GROUP BY Using a Serial Plan

In a serial plan, grouping and aggregation are done by the disk process if these conditions are satisfied:

- All aggregate columns and grouping columns are on the same table, or the aggregate columns are on different tables but are in sorted order before the GROUP BY operation is performed.

- The access path satisfies the order of the group-by columns, as follows:

  ```
  CREATE INDEX ix1 ON empdata (manager, salary);
  SELECT SUM(salary)
    FROM empdata
    GROUP BY manager ;
  ```

- The query only references one table in the FROM clause.

- The optimizer determines that there is a saving in messages when using disk process aggregation and grouping. This determination is based on costing.
- The query does not use OR optimization.

GROUP BY Using a Parallel Plan

In a parallel plan, grouping and aggregation are done by the disk process if this conditions are satisfied:

- The query accesses a single table.
- The query uses primary index access against a partitioned table that is in grouping column order, or the query uses index-only access against a partitioned index that is in grouping column order. Following is an example of the second condition:

  ```
  CREATE INDEX ix1 ON empdata (manager, salary)
  PARTITION (
    $vol1.subvol.ix1p
    FIRST KEY ‘I’,
    $vol2.subvol.ix1p
    FIRST KEY ‘R’);
  SELECT manager SUM(salary)
    FROM empdata
    GROUP BY manager ;
  ```

  Although the index must be partitioned, the table need not be partitioned. The example creates an index with three partitions for manager. (Manager ranges from A to H, I to Q, and R to Z.)
- The optimizer determines that there is a lower execution cost when using disk process aggregation and grouping. This determination is based on the ratio between the number of groups selected and the total number of rows read.
- The query does not use OR optimization.

GROUP BY and MDAM

When the optimizer chooses MDAM processing, the decision to process GROUP BY predicates and aggregates in the disk process or in the executor is based, in part, on the number of columns the optimizer selects for processing by MDAM. (The selected columns are listed in the EXPLAIN plan under “MDAM predicate set” and “next set.”)

For example, if the EXPLAIN plan shows that the first four columns of a six-column key are used for MDAM, then a GROUP BY on the first four, five, or all six columns can be done by the disk process. A GROUP BY on the first, first two, or first three columns must be done in the executor.

If you do not like the number of columns the optimizer chooses for MDAM processing, you can use the CONTROL TABLE directive for MDAM and specify the number of key columns you want MDAM to use. For more information, see Controlling the Number of Key Columns Used by MDAM on page 4-30 and “CONTROL TABLE Directive” in the *SQL/MP Reference Manual.*

MDAM can process GROUP BY predicates in queries that use serial or parallel plans.

Evaluation by the Executor Component

If an aggregate or GROUP BY operation does not fit the preceding requirements, it cannot be evaluated by the disk process. All participating rows are transferred from the disk process to the executor, where groups are formed and aggregation is done.

These situations also require evaluation by the executor:

- A query that includes an executor predicate, because a row does not qualify until the executor predicate is evaluated, as shown

  ```
  SELECT SUM(A) FROM T WHERE A IN (SELECT * FROM T1) ;
  ```
- An index access with the aggregate column on the base table.
- A WHERE clause in a LEFT JOIN query that is evaluated on a column from the inner table after the outer join is done.
- A query that uses real sequential block buffering (RSBB); for more information, see Section 4, Improving Query Performance With Environmental Options.
- Aggregation if any partition of the table is on a node that runs a D10 or earlier version of SQL, or if information about a partition is not available at compile time.

Evaluation by Both the Disk Process and the Executor

In these situations, aggregation is done by the disk process but is finalized by the executor process:

- When OR optimization or MDAM is performed, as shown:

  ```
  SELECT SUM(A) FROM T WHERE A = 5 OR (A >= 10 AND B = 6) ;
  ```
- If there is a nested join in the query
- If the user has requested that SQL skip unavailable partitions, by using the CONTROL TABLE...SKIP UNAVAILABLE PARTITIONS directive

Hashed Aggregation and Grouping

If the table is not already ordered on the GROUP BY columns, SQL can perform aggregation and GROUP BY processing with either a sort operation or a hash operation, depending on which one has a lower cost.

Hashing reduces the cost of sorting data and therefore performs better than other methods when rows are not already ordered on the grouping columns. (For more information about cost, see Section 5, Selectivity and Cost Estimates.) SQL hashes the rows on its grouping columns and then computes the aggregate on the result of the hashed rows.

For parallel plans, SQL always uses hashed aggregation unless a previous version of software is running that does not support hashed aggregations. Each ESP process does its own aggregation and sends its results to the master ESP. This strategy reduces network traffic by applying grouping and aggregation locally.

SQL does not use hashing for columns that use collations.

Sorted GROUP BY Operation

A sort is usually the least efficient method for processing aggregates and GROUP BY operations; when using a sort, SQL sorts the entire table on the grouping columns, and then evaluates the aggregate on the result of the sorted rows.

Optimizing Subqueries

In general, use joins instead of subqueries, as discussed in the preceding subsection. When you do write subqueries, however, you should understand how the optimizer evaluates subqueries, and use noncorrelated subqueries instead of correlated subqueries whenever possible.

Correlated Subquery

Subqueries are usually evaluated before an outer query is executed. If, however, a subquery cannot be executed independently of the outer query, as when the subquery references values from the outer query, the subquery is executed for every qualifying row of the outer query. This type of subquery is called a correlated subquery.

Consider this query, which attempts to select the names of all employees who are designated as managers:

```
SELECT EMP_ID, FIRST_NAME, LAST_NAME, MGR_ID
   FROM EMPLOYEE EMP1
   WHERE EXISTS (SELECT MGR_ID
     FROM EMPLOYEE EMP2
     WHERE EMP2.MGR_ID = EMP1.EMP_ID) ;
```

The query returns this result:

```
EMP_ID   FIRST_NAME  LAST_NAME  MGR_ID
------   ----------  ---------  ------
  2705   Travis      Simpson      6554
  2906   Etsuro      Nakagawa     6554
  3598   Eichiro     Nakamura     2906
  9069   John        Smith        2705

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

Note. The preceding example associates the correlation names EMP1 and EMP2 to different instances of the EMPLOYEE table. A correlation name is an SQL identifier that you associate with a table or view. You can define correlation names in the FROM clause of the SELECT statement. For more information on correlation names, see the *SQL/MP Reference Manual*.

The EMP1 table appears in the FROM clause of the outer query, but its column, EMP_ID, is referenced in the search condition of the subquery. This type of reference is called a correlated reference. A correlated reference acts as a placeholder for a value belonging to the correlated column, in this case, EMP_ID.

When SQL executes the query for each row selected by the outer query, the correlated reference in the search condition of the subquery is replaced by its corresponding value. Therefore, if the first row selected by the outer query contains the value 2705 in the column EMP_ID, the correlated column reference is replaced by that value.

The subquery is then executed, and the row selected by the outer query is returned if its search condition evaluates to true. The placeholder is refreshed upon the selection of the next row by the outer query. The subquery is executed again to select rows satisfying the new search condition.

You can see that a correlated subquery impacts performance adversely because the subquery is executed every time its search condition changes; that is, it is executed once for every row selected by the outer query. If the outer query retrieves 10,000 rows, for example, then the correlated subquery executes 10,000 times.

In some cases, however, a correlated subquery might be more efficient than a noncorrelated subquery. A correlated subquery might be more efficient, for example, if there are very few rows returned from the outer query and the subquery queries a very small table.

Suppose that the outer query in the former example retrieves only 5 rows, and an index exists on the column specified in the inner query (an index exists on MGR_ID). In this case, evaluation of the correlated subquery can be satisfied with an index-only access; that is, all columns that the query references can be found in the index. To retrieve five rows, the index is accessed five times.

A subquery is dependent on the outer query if the subquery is correlated to the outer query, as shown:

```
SELECT ITEM_NAME, RETAIL_PRICE
  FROM INVNTRY OUTER
  WHERE RETAIL_PRICE > SELECT AVG(RETAIL_PRICE)
    FROM INVNTRY
    WHERE  PRODUCER = OUTER.PRODUCER
```

This example selects information on items that cost more than the average price of the items produced by the same producer. The subquery is dependent on the outer SELECT because it references the PRODUCER column of a row retrieved for the outer SELECT. This correlation forces the evaluation of the subquery for every row retrieved from the outer SELECT. The overall query is more expensive to evaluate because of the repeated evaluation of the subquery.

Noncorrelated Subquery

A noncorrelated subquery does not reference or depend on the result of the outer query. Consequently, a noncorrelated subquery can be evaluated once and the results used repeatedly for the outer query.

This example contrasts the performance of the correlated subquery with that of a noncorrelated subquery. The subquery in this example executes only once to select rows that satisfy the specified search condition:

```
SELECT emp_id, first_name, last_name, mgr_id
   FROM employee emp1
   WHERE emp_id IN (SELECT mgr_id
      FROM employee emp2) ;
```

The subquery is evaluated to determine MGR_ID, and MGR_ID is then substituted in the predicate.

The noncorrelated subquery returns the same result as the correlated subquery in the preceding example, but with fewer system resources:

```
EMP_ID   FIRST_NAME  LAST_NAME  MGR_ID
------   ----------  ---------  ------
  2705   Travis      Simpson      6554
  2906   Etsuro      Nakagawa     6554
  3598   Eichiro     Nakamura     2906
  9069   John        Smith        2705
```

--- 4 row(s) selected.

The subquery is independent of the outer SELECT because it can be evaluated without any knowledge of the result of the outer SELECT. This independence allows the subquery to be evaluated only once.

For a cost analysis of correlated and noncorrelated subqueries, see Section 5, Selectivity and Cost Estimates.

Avoiding Full Table Scans

Scans of an entire table can be quite costly in terms of performance. Response time is directly proportional to the number of rows or blocks processed. In general, you should avoid online transaction processing queries that invoke full table scans unless the table is quite small, consisting of only a few blocks.

To avoid table scans, do the following:

- Provide a starting position by using a >= predicate on a key column, as described in Positioning With Key Predicates on page 3-16.
- Include in the select list only those columns that appear in an index and primary key.

An alternate option that requires system management knowledge is to define a new index that includes necessary columns.

You can check for full table scans using DISPLAY STATISTICS or the EXPLAIN utility. For details, see Section 6, Analyzing Query Performance.

Minimizing Sort Costs for Ordering and Grouping Operations

A sort might be needed when a query specifies ordering or grouping options. Because sorting can require significant overhead, the optimizer attempts to select strategies that minimize the number of sorts.

You cannot control the initiation of the sort process; however, by understanding how to formulate queries that can minimize sorts, you can improve the performance of the queries. To eliminate sorts or minimize sort requests, use these guidelines:

- Specify these clauses in your queries only when you really need them:
  - The ORDER BY clause (to present the result in a certain order)
  - The GROUP BY clause (to group rows)
  - The DISTINCT clause (to eliminate duplicate rows)
- Use keys or index columns in your queries where possible.

  Use them in the order defined in the database. If you are not sure which columns are keys or indexes, check with your database administrator. SQL does not use keys or indexes if the columns are not in the same order as defined in the database.

  If you are not sure which columns are key or index columns, you can ask your database administrator. Alternatively, you can get this information by using the FUP INFO *tablename*, DETAIL command, which returns the column numbers used in the index (the first column is 0). You can also retrieve this information from the indexes table in the catalog.

- Use predicates for the leading key columns where possible, because SQL uses predicates for key positioning. As the number of leading key columns with predicates increases, the key positioning becomes more precise.
- When you use an ORDER BY clause with an index, include all leading columns of the index being used. This strategy avoids a sort operation. When reading in reverse, use all leading columns and specify DESCENDING (assuming that the columns are in ascending order), to avoid the sort.
- When you use a UNION operator, specify UNION ALL. Specifying UNION without ALL causes a sort operation to eliminate duplicate rows.
- Include DISTINCT columns at the beginning of an ORDER BY list when both clauses are included in a query.

An alternate option that requires system management knowledge is to maintain indexes that provide SQL with a naturally sorted order for the sequencing of rows most often requested.

The next subsection describes sort operations. The following subsections describe how to use ORDER BY, GROUP BY, and DISTINCT in combination and how an index can improve performance of a query.

Sort Operations

A logical sort returns values in a specified order. The ORDER BY clause requires a logical sort operation. The GROUP BY clause, DISTINCT clause, UNION clause without the ALL option, and noncorrelated subqueries can also require a sort operation, but these operations can sometimes use a hashing operation, which is more efficient than a sort. A noncorrelated subquery causes a sort only if it would perform a sort if it were performed alone.

A sort is logically required when a query specifies that:

- The result is presented in a certain order (using the ORDER BY clause).
- Duplicates are removed (using the DISTINCT key word).
- The result is grouped (using the GROUP BY clause) on certain columns.

Avoiding Sorts

Because sorting is an expensive operation, the optimizer attempts to minimize the number of sorts that must be performed for a query. The optimizer avoids a sort for an ORDER BY, GROUP BY, DISTINCT, or UNION operation if the sort specification (such as the ORDER BY columns) matches a prefix of the index columns.

SQL also avoids unnecessary sorts by removing an ORDER BY clause if no column is present in the SELECT list.

Sort Implementations

Depending upon the query request, SQL determines if a sort of the data rows is required and automatically initiates the sort process. A logical sort can be implemented in one of these ways:

- An in-memory user process sort (UPS). If the optimizer estimates that the data meets all of these requirements, the sort can be performed within the executor’s extended segment:
    - The data to be sorted is less than 4 megabytes (MB)
    - The number of rows is less than 32,767
    - The number of sort keys is less than 63

  At runtime, if the segment is discovered to be too small to hold all of the rows, SQL invokes an external physical sort. When performed in memory, the sort does not require startup of a sort process, nor does it use scratch files, thus reducing elapsed time for queries.

  SQL does not choose an in-memory sort if the table to be sorted is the inner table of a sort merge join operation.

  SQL can choose an in-memory sort for the serial portion of a parallel plan if all of these conditions apply:
    - UPS is used for an ORDER BY clause executed by the master ESP
    - The ORDER BY clause is preceded by a GROUP BY clause
    - The GROUP BY order is achieved by hash grouping
- An external physical sort. SQL can invoke these:
    - An external FastSort process if the optimizer estimates that the data to be sorted might exceed 4 MB, the number of rows is less than 32,767, and the number of sort keys is less than 63.
    - An external physical sort done by a series of inserts into a temporary key-sequenced table if the number of rows is less than 500, the number of columns is greater than 63, and the total key length is less than 255 bytes. (If the total key length is greater than 255 bytes, the sort process returns an error.)
- A primary-key scan
- An alternate-index scan. For example, if an index is available and is chosen by SQL to satisfy an ORDER BY clause, a physical sort is not necessary.

If a sort process is initiated, the =_SORT_DEFAULTS DEFINE can define the location of swap and scratch files. (For more information on using this and other DEFINEs, see the *SQL/MP Reference Manual*.)

You can use the EXPLAIN utility to check the type of sort operation performed and the size of the workspace allocated by the executor (UPS workspace) for in-memory sorts. For more information on EXPLAIN, see Section 6, Analyzing Query Performance.

Optimizing Combinations of Clauses

The SQL optimizer attempts to minimize the number of sorts required for a query. You can assist this process by using the guidelines in this subsection.

Specifying ORDER BY With GROUP BY

You can specify order and grouping and still eliminate extra sorts. A single sort orders and groups results when this occurs:

- The ORDER BY list is a subset of the GROUP BY list.
- For example, consider this query:

  ```
  SELECT A, B, C, D FROM T
     GROUP BY A, B, C, D
     ORDER BY B, C DESC ;
  ```

  A single sort (on B, C DESC, A, D) performs both the grouping and the ordering.

- The GROUP BY list contains *n* items, and those items are also the first *n* items of the ORDER BY list; for example:

  ```
  SELECT A, B, C, COUNT(*), SUM(A) FROM T
     GROUP BY A, B, C
     ORDER BY 1, 2 DESC, 3, 5, 4 ;
  ```

  SQL need only perform a single sort (on A, B DESC, C) to perform both the grouping and the ordering.

The formation of groups requires the groups to be hashed in memory or to ordered by the grouping columns. If they are already in order, then no sorting or hashing is needed. If they are not in order, the optimizer compares a plan that sorts columns first with a plan that hashes the columns.

When both an ORDER BY clause and a GROUP BY (or DISTINCT) clause are used in a query, SQL can choose to combine the lists into a single sort. This is possible only if the GROUP BY list is a prefix of the ORDER BY list. In such a case, a single sort can fulfill both the GROUP BY and ORDER BY requests. This can save a sort and make the plan more efficient, but might still be less efficient than hashing the groupings and then sorting a small number of resulting rows.

Specifying GROUP BY With DISTINCT

You can specify grouping and the elimination of duplicate rows and still avoid extra sorts. A single sort satisfies both grouping and the elimination of duplicate rows when this occurs:

- The GROUP BY list is a subset of the SELECT DISTINCT list, as follows:

  ```
  SELECT DISTINCT COUNT(*), B, B-D, D FROM T
     GROUP BY B, D ;
  ```

  SQL performs a single sort (on B, D) to perform the grouping. Because each (B, D) value is unique after grouping, so is each (B, D, B-D, COUNT (*)) value.

- The SELECT DISTINCT list is a subset of the GROUP BY list, there are no expressions in the select list, and no aggregates in a HAVING clause, as follows:

  ```
  SELECT DISTINCT A, C FROM T
     GROUP BY A, B, C ;
  ```

  In this example, the GROUP BY clause is unnecessary. A single sort eliminates duplicates and performs the grouping (A,C). Because B does not appear in the select list and there are no aggregates or HAVING clause that depend on the full grouping, it is not necessary to group by B.

If the DISTINCT column is not already in the GROUP BY list, you can avoid the sort for the DISTINCT column by adding the DISTINCT column to the list of grouping columns.

Specifying ORDER BY With DISTINCT

You can specify ordering and the elimination of duplicate rows and still avoid extra sorts, if the ORDER BY list is a subset of the DISTINCT list. Consider this query:

```
SELECT DISTINCT A, B, C, D FROM T
   ORDER BY A, B DESC ;
```

SQL can evaluate the query with a single sort on (A, B DESC, C, D) to satisfy both the ordering and the elimination of duplicate rows. The position and sorting order (ascending or descending) of A and B must match the index chosen to perform the sort, but C and D can appear in any position after A and B and in either ascending or descending order.

Using Indexes

Indexes can improve the performance of queries that would otherwise require a sort operation.

If an index exists that has the same key columns as the ORDER BY columns, you can avoid a sort if the sequence of columns in the ORDER BY clause matches the sequence of columns in the index. State your ordering requirements explicitly; do not assume that rows will be returned in a specific order because of the primary-key sequence or because there are equality predicates on index columns. When you specify complete column order, performance is best, but when you cannot, MDAM preserves the order of the key.

For information about creating indexes, see the *SQL/MP Installation and Management Guide*.

Examples

Consider this query:

```
SELECT * FROM INVNTRY
  ORDER BY ITEM, RETAIL_PRICE ;
```

If the INVNTRY table is large, the cost of sorting the table might be very high. An index on the columns ITEM and RETAIL_PRICE would mean that no sort is required to satisfy the ORDER BY clause:

```
CREATE INDEX RPRICE
   ON INVNTRY (ITEM, RETAIL_PRICE) ;
```

It is still possible, however, that a scan and sort might be less expensive than an index access. If so, SQL ignores the index and scans the base table instead.

For another example, consider a query that specifies the elimination of duplicate rows:

```
SELECT DISTINCT A, B, C FROM T ;
```

A unique index on any subset of columns A, B, and C would guarantee that all A, B, and C values are unique. So, if SQL chooses the unique index as the access path, there is no need to eliminate duplicates during query execution.

```
CREATE UNIQUE INDEX UI
   ON T (A, B) ;
```

As a final example, consider this query:

```
SELECT COUNT(*) FROM T1
   GROUP BY T1.A, T1.B, T1.C ;
```

All items in the GROUP BY list are from a single table, so a unique index on any subset of the GROUP BY list can be used to perform the grouping, as follows:

```
CREATE UNIQUE INDEX UI
   ON T1 (T1.A, T1.C) ;
```

Writing Efficient Programmatic Statements

When writing programmatic queries, you might have a choice between several strategies for combinations of SELECT statements, cursor use, and update, delete, and insert operations.

Single-Row and Multiple-Row SELECT Statements

A single-row SELECT statement is a request to return a single row to the host program. This method is preferable to a cursor SELECT when only one row needs to be retrieved, because

- There are fewer executor calls
- The disk process returns a single message instead of two messages when an equality predicate is specified against columns of a unique index
- The operation disables SBB so that unnecessary scanning and transfer to buffers is avoided for the single row

Multiple-Row (Cursor) SELECT Statements

A multiple-row (or cursor) SELECT statement returns multiple rows one row at a time. This technique is useful when retrieving multiple rows because

- The cost of additional executor calls (for OPEN and CLOSE CURSOR) is spread out over multiple fetches
- SBB allows efficient scanning and transfer to the buffer for multiple-row access

Update and Insert Operations

To minimize executor calls and message traffic, use these guidelines when deciding how to handle SELECT-with-update or SELECT-with-insert requests:

- Choose a set update or delete instead of either
    - A single-row SELECT followed by an exact update or delete.
    - Multiple single-row SELECTS and subsequent exact updates or deletes. (Note, however, that if you update large numbers of rows, lock escalations or exceeded lock limits might cause aborts of transactions. In such a case, consider committing a transaction after a certain number of updates or specify ranges of rows in multiple set updates.)
    - A cursor SELECT and one or more UPDATE or DELETE WHERE CURRENT operations (unless information must be examined prior to update).
- In general, when selecting and updating a single row, choose a single-row SELECT and exact update or delete over a single updatable cursor UPDATE or DELETE WHERE CURRENT.

- Conversely, when selecting and updating multiple rows, choose multiple updatable cursor UPDATES and DELETES WHERE CURRENT over multiple single-row selects followed by exact updates or deletes. The EXCLUSIVE and REPEATABLE lock mode and access options are recommended for these types of operations. For more information, see Section 4, Improving Query Performance With Environmental Options.
- Choose an INSERT from a SELECT instead of a SELECT followed by one or more INSERTS; for example:

  ```
  INSERT into INVNTRY
    (SELECT part_no, qty FROM WRKNPROC
      WHERE part_status = "FINISHED") ;
  ```

  (Note, however, that if you update large numbers of rows, lock escalations or exceeded lock limits might cause aborts of transactions. In such a case, consider committing a transaction after a certain number of updates or specify ranges of rows in multiple set updates.)

Decision Support Considerations

Decision Support Systems (DSS) are systems that provide users a means to retrieve information from a large database and perform information analysis. These systems require processing, summarizing, and aggregation of data for quick business decisions. Requirements include fast processing of large amounts of data.

A typical DSS database consists of a very large table, with the actual temporal data of the database, and several small tables that contain static information about the data. Typical queries are joins of these small, static tables to the large database table.

For join operations, these guidelines apply:

- If the joining columns are part of the primary key of the joining tables, a nested join is typically the most efficient join method.
- If the joining columns are not part of the primary key and alternate indexes are not available, a hash join is the preferable method.
- If the joining columns are part of the primary key of the tables to be joined and the tables are joined in the same order as the columns in the primary key, a key-sequenced merge join is the preferable method.

Features that support DSS include:

- Hash joins. See Hash Join on page 3-29.
- Key-sequenced merge joins. See Key-Sequenced Merge Join on page 3-27.
- MDAM. See Transformation of Predicates on page 3-4.
- Aggregate processing. See How the Optimizer Processes Aggregates and Group-By Operations on page 3-46.

- Column selectivity. See Combinations of Predicates on page 5-4.
- Query Rewrite. See Transformations Related to Joins on page 3-8.
- Reduction of sorts. See Sort Operations on page 3-55.
- In-memory sorts. See Sort Operations on page 3-55.
- Hashed groupings. See Hashed Aggregation and Grouping on page 3-50.
- CASE expressions. See Using String Functions on page 1-15.
- String functions. See Using String Functions on page 1-15.
- Effective optimizer plan selection. See How the Optimizer Chooses an Execution Plan on page 2-3.

Online Transaction Processing Considerations

These guidelines apply to interactive queries:

- Avoid processing large numbers of rows.
- For small result sets, use CONTROL QUERY INTERACTIVE ACCESS ON, described in Section 4, Improving Query Performance With Environmental Options. This directive requests the optimizer to use an optimal index whenever possible because the query requests only a few rows.
- Specify SEQUENTIAL INSERT/UPDATE OFF. Sequential operations are usually reserved for batch processing.
- Avoid the use of REPEATABLE ACCESS because of potential contention and delays. For more information, see Section 4, Improving Query Performance With Environmental Options.
- Consider specifying SEQUENTIAL READ OFF. For more information, see Section 4, Improving Query Performance With Environmental Options.
- Do not use sequential cache unless tables are relatively small.
- Depending on your concurrency requirements, consider using a CONTROL TABLE LOCK directive. For more information, see Section 4, Improving Query Performance With Environmental Options.

Batch Considerations

Batch operations imply queries that process large amounts of data in a sequential order.

Use of these can improve batch performance:

- Block buffering, described in Reducing Messages With Buffering Options on page 4-21.
- Parallel sorts to increase speed and balance processor and disk use.
- Parallel processes to even out workload and make the system easier to balance.
- Multiple spooler processes so that parallel batch processes can write to multiple spoolers.
- Key-sequenced tables with indexes to avoid sorts.

These guidelines apply to update operations:

- Whenever an insertion, update, or deletion occurs within a process that has an open cursor (whose execution plan uses sequential block buffering (SBB) to access the same table), perform the update or delete with the same cursor (use the WHERE CURRENT clause). If you do not follow this guideline, virtual sequential block buffering (VSBB) is used for the SELECT with the UPDATE, invalidating that buffer, which can be very expensive. CONTROL TABLE *`tablename`* SEQUENTIAL READ OFF cannot be used to avoid this situation.

  For more information about SBB and VSBB, see Section 4, Improving Query Performance With Environmental Options.

- Use cursors for UPDATE with EXCLUSIVE when you know you will update the row and then UPDATE WHERE CURRENT. Otherwise, you can use a single SELECT.
- Use BEGIN WORK and COMMIT WORK only for update transactions; otherwise, you incur unnecessary overhead in the audit trails. A start and stop transaction audit record is written unnecessarily to the audit trail, and TMF maintains a transaction reference throughout the execution of the transaction.
- Commit as soon as possible when finished. Note that you might need extra commits to avoid a lock limit or lock escalation.
- If batch updates occur during online processing, define short transaction intervals so that rows are not locked for long periods of time. If updates are large, consider dividing them into smaller updates and committing transactions at a frequency based on concurrency issues and transaction limits. For example, you can commit transactions based on number of updates or after a fixed length of time.

4 Improving Query Performance With Environmental Options

The best way to control your NonStop SQL/MP processing environment is to design and maintain your database so that the mix of queries executes efficiently. Beyond structural design and ongoing maintenance, however, there are several environmental factors, discussed in this section, that can influence query performance:

For information about how to optimize specific queries, see Section 3, Improving Query Performance Through Query Design.

The *SQL/MP Installation and Management Guide* contains information about topics related to the performance of the database as a whole. Topics include:

- Defining columns for optimal access
- Creating alternate indexes
- Maximizing parallel index maintenance
- Partitioning data
- Specifying cache buffer size
- Maximizing disk prefetch capabilities

Note. This manual supports NonStop SQL/MP D30.02 and D30.03. Information that describes how the optimizer chooses a query execution plan can change from release to release.

Keeping Statistics Current

SQL provides an UPDATE STATISTICS utility to collect and save statistics on columns and tables. The SQL compiler uses these statistics to determine the selectivity of predicates, indexes, and tables.

Because selectivity directly influences the cost of access plans, it is important to have current statistics for a table, to increase the likelihood that the optimizer will choose an efficient plan. (For more information about selectivity, see Section 5, Selectivity and Cost Estimates.)

You might want to run UPDATE STATISTICS after loading or re-creating a table, after structural changes such as creation of an index, or after significant update activity (growth in database size). Before running UPDATE STATISTICS, however, you should consider the following:

- If you experience performance degradation, check for fragmentation of blocks. Use the FILEINFO command with the STATISTICS option set on. If blocks are fragmented, running UPDATE STATISTICS and recompiling the queries does not help; first reload the table online by using the FUP RELOAD command.
- Run UPDATE STATISTICS only after a table has been loaded with data. Do not run UPDATE STATISTICS when a table is empty.
- Depending on the size of the table, updating statistics can take longer than you would like; therefore, run UPDATE STATISTICS during off hours when peak performance is not required. You can determine the effect of UPDATE STATISTICS on a production query by bracketing UPDATE STATISTICS and EXPLAIN on the queries in a transaction.
- First determine the effect of the UPDATE STATISTICS statement by issuing the statement within a TMF transaction. You can then back out the operation if necessary. In an SQLCI session, do the following:
    - Issue a BEGIN WORK statement; then issue UPDATE STATISTICS with the NO RECOMPILE option.
    - Use EXPLAIN to see if the new statistics would give you the better query execution plan.
    - Depending on the EXPLAIN output, you can decide whether to commit the transaction (COMMIT WORK) or back out the transaction (ROLLBACK WORK).

- Always specify the NO RECOMPILE option when using UPDATE STATISTICS, for this reasons:
    - By default, an UPDATE STATISTICS operation invalidates dependent programs, even if UPDATE STATISTICS is executed within a transaction that is backed out.

      Catalogs are audited; program file labels are not. Because program file labels are not audited, updates to program file labels are not backed out. Consequently, if a transaction is backed out, the program file labels are left in an invalid state, while the catalog specifies a valid state.
    - To avoid invalidating dependent programs and therefore avoid inconsistencies between the program file label and the catalog. Until you explicitly compile the affected programs, however, they will not use the new statistics.
- If you want to preserve the existing query execution plan, please be aware that running UPDATE STATISTICS might cause the optimizer to choose a different plan.
- Run UPDATE STATISTICS after creating a new index for a table; otherwise, SQL returns a warning for subsequent operations on the table.

For a thorough evaluation of access options, include key columns, index columns, and those nonindex columns that participate in predicates. To update statistics for all columns, you must specify UPDATE ALL STATISTICS.

This example updates statistics for primary key columns of the EMPLOYEE table and columns that have been specified in any alternate index on the table:

```
UPDATE STATISTICS FOR TABLE EMPLOYEE NO RECOMPILE;
```

This example requests statistics by reading all rows in the first 50 blocks of each partition of the EMPLOYEE file:

```
UPDATE STATISTICS FOR TABLE EMPLOYEE SAMPLE 50 BLOCKS;
```

You can choose to read the entire table (EXACT option) or a specified number of blocks of each partition (SAMPLE n BLOCKS option) for computing statistics. These options help control the amount of time spent calculating statistics. If neither of these options is specified, statistics are collected by reading all rows in partitions smaller than 1,000 blocks, and approximately 500 blocks from each partition larger than 1,000 blocks.

Statistics are collected at the table level, except for row count and nonempty block count, which are stored on a partition-by-partition basis. Unique entry count is divided equally among the partitions of a table, with any remainder added to the primary partition.

For more information about the UPDATE STATISTICS statement, see the *SQL/MP Reference Manual* and the *SQL/MP Installation and Management Guide*. For information on using FILEINFO and FUP RELOAD, see the *SQL/MP Installation and Management Guide.*

Optimizing the Access Path

An access path is the method by which data is accessed. Access can be one of these:

- Primary access (table scan or primary key)
- Alternate-index access

Different access paths provide different degrees of efficiency in accessing a table.

Primary Access

There are two types of primary access: table scan and primary key.

Table Scan

In a table scan, SQL reads the entire base table from beginning to end in primary-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

- Processing small tables
- Processing a large percentage of rows in a table
- Using parallel execution
- There is no suitable index available
- The estimated cost of reading the index and the corresponding base table rows exceeds the cost of reading the entire table

In general, OLTP queries should not invoke full table scans unless the table is quite small, consisting of a few blocks only.

To avoid table scans, do the following:

- Specify a starting position by using a >= predicate on a key column.
- Include in the select list only those columns that appear in an index and primary key.
- Define a new index.
- Do not disable MDAM.

To check for full table scans, use DISPLAY STATISTICS (if row count is available) or the EXPLAIN utility. If your EXPLAIN SCAN output says that 100 percent of the table is being accessed, or if there is no entry for begin or end key in the EXPLAIN plan, the scan is reading the entire table. For details, see Section 6, Analyzing Query Performance.

Primary Key

Access through a primary key means reading a portion of the base table derived from the primary-key value.

Using a primary-key access to retrieve a row is usually cheaper than these other methods:

- A sequential scan
- Base-table access through an index (described under “Alternate-Index Access,” following)

The latter is usually less efficient because scanning a range of values in primary-key order is faster than the random I/O required to read base table rows when scanned through an index.

The optimizer might choose access through a primary key when

- An index-only access is not possible.
- The estimated cost of the primary-key access is less than the estimated cost of the alternate-index access.

  For example, suppose that the following:

  - The primary access reads 100 rows from 5 blocks.
  - The index access reads 10 rows from 10 blocks plus the index.

  In this case, it is cheaper to read 5 blocks (through the primary key) than 10 blocks plus the index (through the alternate index).

In general, a scan is more efficient if you request more than about five percent of the rows in the table. If sequential cache is chosen by the optimizer, the percentage drops to approximately one percent.

Alternate-Index Access

There are two types of alternate-index access:

- Index-only access
- Base-table access through an index

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

- All of 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 (and reading the index is less expensive than reading the base table and performing the sort).

- A table scan can be avoided and the percentage of rows to be read is small enough to make index access cheaper.

An index contains one or more columns defined as the index, plus the columns that make up the primary key. An index benefits the query most when all the columns needed by the query are located in the index.

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.

Index-only access compares to base-table access through the primary key as follows:

- For random access, an index-only access usually costs about the same as a primary-key access of the base table.
- For sequential access, an index-only scan is superior to a primary-key scan of the base table, because 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).

Base-Table Access Through an Index

Base-table access through an index means that the index row is located first, then the row in the base table is accessed through its primary key.

If many rows must be accessed to satisfy a query, access to the base table through an index can be more expensive than a full table scan. When accessing the base table through an index, rows in the base table are read randomly, and some blocks containing those rows might be read more than once.

Using the CONTROL QUERY Directive

To inform the SQL compiler that a particular query is interactive or batch, use the CONTROL QUERY directive; for example:

```
CONTROL QUERY INTERACTIVE ACCESS ON
```

This directive influences the optimizer’s choice of access to a table, as follows:

- If you specify INTERACTIVE ACCESS ON, you instruct the optimizer to return rows by choosing an alternate index over sequential access. This strategy optimizes the response time for returning the first few rows requested.

  You might want to specify INTERACTIVE ACCESS ON, for example, when you want only the first few rows in the result or if you know that very few rows will be examined through an index.

- If you specify INTERACTIVE ACCESS OFF, the optimizer optimizes the response time for returning all rows requested.

You should specify INTERACTIVE ACCESS OFF when query or batch processing a large number of rows.

The default value is OFF.

The CONTROL QUERY directive is most valuable in application use when host-variable values are not available at compilation time. Because the values are not available, the optimizer chooses a query execution plan that assumes a large number of rows will be returned and so might overestimate the resources required for a smaller interactive query.

The optimizer sometimes chooses sequential access of the base table over an alternate-index access. This is because an alternate-index access causes a row to be retrieved from the underlying base table through an additional random I/O request.

Use the CONTROL QUERY directive to turn MDAM off only if you find the query runs better without it. Using the CONTROL QUERY directive to enable MDAM does not force MDAM and has no effect because MDAM is enabled by default. For more information on MDAM, see Transformation of Predicates on page 3-4.

You can also use the CONTROL QUERY directive to enable or disable hash joins. For more information, see Specifying a Join Method on page 3-43. For a complete description of the CONTROL QUERY directive, see the *SQL/MP Reference Manual*.

The CONTROL QUERY directive stays in effect until the end of an SQLCI session, the end of a program, or until the directive is reentered.

In a host language program, specific placement rules might apply to the CONTROL QUERY directive. For more information, see the *SQL/MP Programming Manual* for your host language.

Selecting an Access Path When an Index Is Not Available

SQL can access required data even if some system resource—an index, for example—is not available.

For example, suppose that the table TPHONE is on the volume $PHONE; an index on PHONE_NUMBER is stored on the volume $NUMBER. Consider this query:

```
SELECT LAST_NAME, FIRST_NAME, PHONE_NUMBER FROM TPHONE
  WHERE PHONE_NUMBER = "725-6000" ;
```

Suppose further that SQL has chosen to use the index on $NUMBER to retrieve the requested data.

If the volume $NUMBER is not available at run time, SQL attempts to locate an alternate path to the data by performing an automatic recompilation—unless you have specified the NORECOMPILE option. (If the NORECOMPILE option is in effect, SQL cannot automatically recompile the program, and an error is returned for this SQL statement.)

At compile time, if any information is unavailable, the SQL compiler sets a flag indicating that the query must be recompiled at run time. A valid SQL object is still produced because information might be available for other queries in the same program.

At run time, the SQL executor recompiles the query when it encounters the flag that indicates a recompile is necessary. The SQL executor instructs the SQL compiler to ignore unavailable information during this compile. If all information is now available, the most efficient access plan can be selected. If some information is still not available, the SQL compiler attempts to identify the most efficient access plan based on the information that is available.

If the chosen index is still not available (for example, the communications line to a node is down, or the volume containing the chosen index is down after the recompilation), the executor instructs the optimizer to choose the primary key of the table as the access path. This instruction enables the application to access the data whenever the data is available, even if an optimal path is unavailable.

For more information about compiling programs and the RECOMPILE and NORECOMPILE options, see the *SQL/MP Programming Manual* for your host language.

Note. Compilation rules differ for dynamic queries. For details, see the *SQL/MP Programming Manual* for your host language.

Understanding Unexpected Access Paths

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

- Did a CONTROL TABLE or CONTROL QUERY directive instruct the compiler to take a different access path?
- Was index-only access not possible because of the columns that must be retrieved?
- Does a WHERE clause indicate base-table access?
- Does the data have an uneven distribution not recognized by the compiler?
- Would the request cause the Halloween problem?

Control Directive Is Specified

Either a CONTROL QUERY INTERACTIVE ACCESS ON directive instructed the compiler to take a path that did not require sorting, or a CONTROL TABLE directive was used to force a particular path.

Index-Only Access Is Not Possible

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

- The columns required by the query are not all included in the index.
- An exclusive lock mode is selected.
- OR optimization is done. (For more information, see Writing Efficient Predicates on page 3-15.)
- The access is part of a DECLARE CURSOR statement with a FOR UPDATE clause specifying a column that belongs to the index key, or part of an UPDATE statement that updates a column of the index key.

  SQL does not use index-only access if it estimates that more than a third of the table is to be updated or deleted.
- Data is accessed from a protection view defined with a WHERE clause—unless real sequential block buffering (RSBB) can be used.

  For more information on RSBB, see Reducing Messages With Buffering Options on page 4-21.

When the SELECT statement specifies more columns than can be satisfied by the index alone (ineffective projection), base-table access is required. The index might be used to access the base table, but index-only access is not possible.

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. For example, the overall estimated cost might be higher for alternate-index access than for primary key access. In addition, if the base table is small the optimizer might choose to read the base table without accessing the alternate index.

Data Is Not Evenly Distribution

The optimizer operates as if a column contains a uniform distribution of values. Consequently, the optimizer might not operate appropriately if the values are, in fact, unevenly distributed.

For an example, consider table T, which has these characteristics:

- The table contains 2000 rows.
- There are two indexes on the table: one on column A and one on column B.
- The values in column A range from 1 to 10.

  There are, however, 1,000 rows with A=10. The other values of column A are evenly distributed from 1 to 9.
- The values in column B range from 21 to 30 and are evenly distributed.

If you specify this query:

```
SELECT * FROM T
   WHERE A > 9 AND B > 28 ;
```

The optimizer operates as if these were true:

- 10 percent of the table will be selected through index A.
- 20 percent of the table will be selected through index B.

Based on this assumption, the optimizer will choose index A. However, because there are 1,000 rows with A = 10, index A will actually access 50 percent of the table.

The Halloween Problem Could Occur

An alternate index is ignored if a cursor or standalone set update was 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 following the current row position. This updated row might then be encountered again and updated again and again (the so-called Halloween problem, named after the holiday on which it was supposedly discovered).

For example, suppose PRICE is the first column of an index and is being incremented by 10 percent. As the column is updated, the row is inserted following 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 there is a nonunique index on RETAIL_PRICE 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 system 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.

The SQL compiler avoids Halloween situations whenever possible. If the use of a given index causes a Halloween problem and a CONTROL directive instructs the compiler to use this index, SQL issues an error message.

One way to avoid the Halloween problem 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. There are, however, instances when it is appropriate to use the index; for example, the index on RETAIL_PRICE is appropriate for this query, even though RETAIL_PRICE is being updated:

```
UPDATE INVNTRY
  SET RETAIL_PRICE = 200
  WHERE RETAIL_PRICE BETWEEN 300 AND 400
```

If there is no other index for the INVNTRY table 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 considers using the index on a column being updated if any of these conditions are satisfied:

- No key column in the index is being updated
- All key columns in the index are specified with equal (=) predicates, as follows, and OR optimization is not being considered:

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

- No column is referenced on the right-hand side of the SET clause, and the index selectivity of the index is less than 20 percent, as follows:

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

  The less-than-20-percent restriction for index selectivity ensures that not too many rows are updated more than once. (For more information about selectivity, see Section 5, Selectivity and Cost Estimates.)

A variation of the Halloween 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.

Specifying an Access Path

You can use the CONTROL TABLE ACCESS PATH option to specify the primary access path or a specific alternate index. The ACCESS PATH option applies only to DML (DELETE, SELECT, UPDATE, and the SELECT portion of an INSERT-SELECT) statements. Sample directives are

```
CONTROL TABLE EMPLOYEE ACCESS PATH INDEX DEPTNUM ;
CONTROL TABLE EMPLOYEE ACCESS PATH SYSTEM ;
```

If you specify the SYSTEM option, SQL chooses the access path; this option is the default. If, for example, you specified CONTROL TABLE * ACCESS PATH PRIMARY to force primary key access during an SQLCI session, use CONTROL TABLE * ACCESS PATH SYSTEM to restore the default access path selection mechanism.

Whenever you force MDAM by using the CONTROL TABLE directive, you must specify an access path. You should not use ACCESS PATH SYSTEM with MDAM ON or in conjunction with CONTROL TABLE *. Only ACCESS PATH options PRIMARY and INDEX are valid for MDAM. For more information on MDAM, see Transformation of Predicates on page 3-4.

Selecting an index for an UPDATE query could result in a Halloween situation. For more information, see The Halloween Problem Could Occur on page 4-10.

If a forced path is not available, the query does not run. If you want a program to choose an alternate path, check for error codes.

△ **Caution.** If you use the CONTROL TABLE ACCESS PATH option to specify a primary or index path, you override the optimizer’s standard cost estimates (described in Section 5, Selectivity and Cost Estimates) and therefore, might cause performance degradation instead of enhancement. If you use this option, you must have a thorough understanding of the SQL optimizer. Use it *only* if the optimizer does not produce the optimal plan.

If you suspect that you might benefit from the use of one of these options, check your application with and without the CONTROL option, using actual Measure statistics from production data.

If you use one of the options, you might want to change this directive 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 any occurrences of it easy to find and change, using one or more of these alternatives:

- Make sure the directive only applies to the statement and table intended. Return the specified table to SYSTEM method directly after the statement.
- Isolate this directive in its own section and perform it from the inline application code.
- Place all statements affected by this directive in separate modules, called as services by other modules.

Confirm all use of this option with data from the Measure product and verify its use periodically to account for changes in data distributions and volumes. Reevaluate its effectiveness with each new version of NonStop SQL/MP.

Requesting Parallel Processing

SQL can take advantage of multiprocessor architecture by dividing an SQL query into smaller tasks and assigning the tasks to separate processors. During parallel processing, each part or partition of data is processed in parallel. After all partitions are processed, they are merged to produce the final result.

Because databases can span multiple disks, using separate processes to access these disks can dramatically improve the performance of a query. This approach can improve the response time of all the basic SQL operations, including selects, inserts, updates, deletes, joins, and aggregate functions. Parallel execution is especially helpful when a large number of rows need to be processed by the executor, but only a small number of rows need to be returned to satisfy the query.

Using the CONTROL EXECUTOR Directive

To request parallel processing, use the CONTROL EXECUTOR directive. The optimizer then evaluates whether the system can process the entire statement or parts of the statement in parallel. Specifying parallel execution, however, does not guarantee that queries are executed in parallel. The optimizer selects the parallel execution plan only if it is the best plan. In particular, parallel execution is not chosen in these cases:

- The statement includes a UNION operation.
- The SELECT statement references more than one table and contains a noncorrelated, nonquantified subquery.
- The statement includes a FOR UPDATE OF clause.
- The CONTROL EXECUTOR PARALLEL EXECUTION OFF statement precedes the query in the source program file text or in the SQLCI session. (OFF is the default.)
- The local system has only one processor or none of the tables in the FROM clause are partitioned.
- The system includes fewer than two logical volumes (excluding $SYSTEM).
- The SELECT statement references a single table that is not partitioned.
- The SELECT statement references a nonaudited table and the FOR BROWSE ACCESS option is not specified.
- A CREATE INDEX statement includes the WITH SHARED ACCESS option.
- The SQL compiler estimates that the cost is higher to process the query in parallel.

Queries against small tables are usually not candidates for parallel processing. Neither are queries where relatively few rows are processed by the executor. While these queries might not be precluded from parallel processing, the cost factor might direct the optimizer to choose a plan that does not include parallel processing.

The optimizer does not choose a nested parallel join when stable access is specified and the tables are not partitioned exactly the same and on a single column.

The CONTROL EXECUTOR PARALLEL EXECUTION ON directive stays in effect until the end of an SQLCI session or until the directive is specified again.

You can also specify CONTROL EXECUTOR PARALLEL EXECUTION OFF to prevent the SQL compiler from generating an execution plan that uses multiple executors. This option is the default.

In a host language program, specific placement rules might apply to the CONTROL EXECUTOR directive. For more information, see the *SQL/MP Programming Manual* for your host language.

For complete syntax of the CONTROL EXECUTOR directive, see the *SQL/MP Reference Manual*.

How Parallel Processing Is Implemented

To implement parallel processing of queries, SQL uses an executor server process (ESP), which uses parallel processing when executing SQL statements.

A master executor invokes several ESPs in parallel to process a statement or part of a statement. The ESPs perform the work and return either data or status information to the master executor. The master executor then processes that information and returns the end result to the user.

Tables and indexes need not be physically partitioned to achieve parallel execution if they are participating in a join query. If a table or index is not partitioned then, for the purposes of processing the query, the data is physically broken into multiple parts (repartitioned), and each part is processed in parallel. Parallel processing, however, operates best when a table or index is partitioned.

Consider this query, which requests the average salary of all employees:

```
CONTROL EXECUTOR PARALLEL EXECUTION ON ;
SELECT AVG(SALARY) FROM EMPLOYEE ;
```

Because the query is processed in parallel, each partition of the EMPLOYEE table is processed by an executor server process (ESP). Each ESP computes a partial sum and count of the SALARY values in the rows in that partition. (The sum and count are used by the master executor to calculate the average.) During the final processing stage of the query, the partial results of each ESP are combined to determine the average for the entire query.

ESPs can be reused by the same SQLCI session.

Requesting Parallel Operations on Partitioned Data

Partitioning a table and its indexes increases the likelihood that the optimizer will choose a parallel execution plan. Partitions might reside on one system or across many systems.

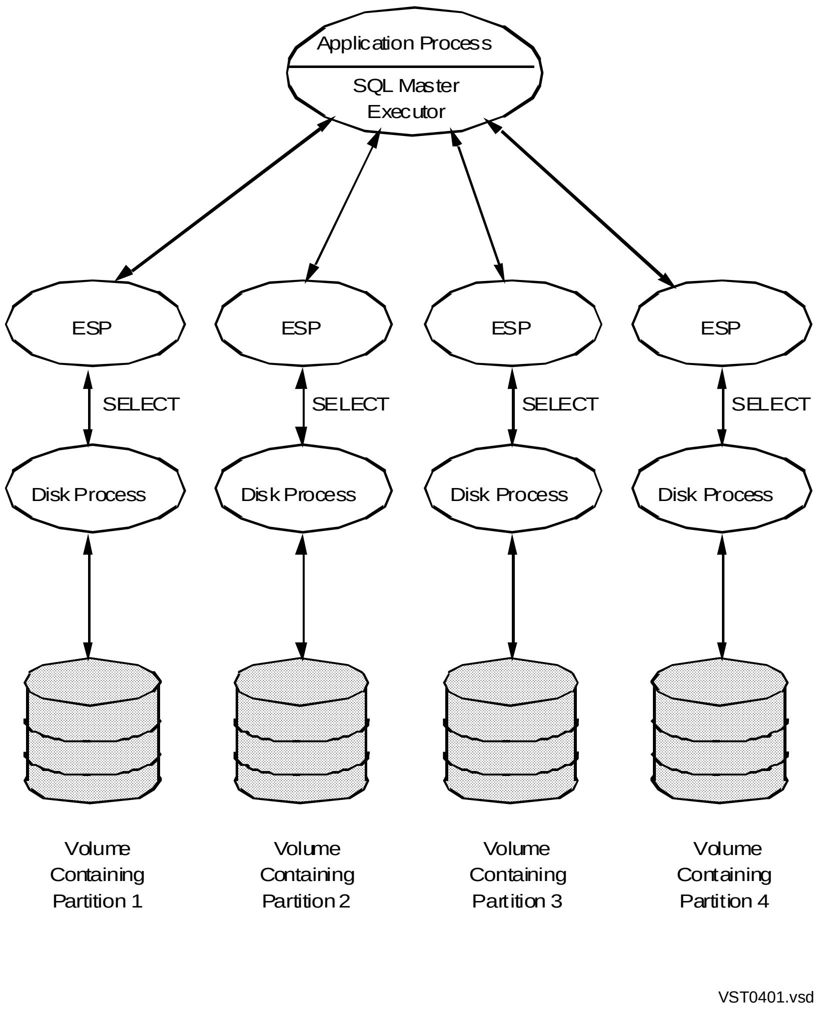
The master executor assigns one ESP process to each partition that must be accessed. At run time, the master executor starts an ESP process in the current primary processor of each partition's disk volume (unless an existing ESP process can be used).

Each ESP works only on the partition to which it is assigned. The master executor simultaneously employs a number of ESPs to work in parallel on the chosen part of the statement. If the table is partitioned across multiple nodes, the ESPs are started on the nodes where the data resides.

For more information on ESPs, see Processor Assignment by the SQL/MP Optimizer and Executor for Executor Server Processes (ESPs) on page 2-5.

Figure 4-1 shows how the master executor assigns four ESPs to perform a SELECT operation on a table with four partitions. Each ESP selects data from one partition and returns it to the master executor.

Figure 4-1. Parallel Execution of a SELECT Statement



Prior to parallel execution of a SELECT statement, some table might not be partitioned or might be partitioned in a way that does not facilitate parallel processing. The optimizer can request that the executor repartition (reorganize) a copy of the data at run time. During repartitioning, SQL distributes the data over a set of temporary partitions. Each partition contains data that can then be processed in parallel by a separate ESP.

The optimizer considers the cost of repartitioning and sorting the data when it selects the best execution plan for the query. For more information, see Section 5, Selectivity and Cost Estimates. For more information about adding partitions to tables, see the *SQL/MP Installation and Management Guide*.

Specifying Access Option and Lock Characteristics

SQL supplies locks and access options to protect the integrity and concurrency of a database:

- Integrity refers to data that is accurate, valid, and consistent according to rules established for changing the database.
- Concurrency refers to the ability of two or more processes to gain access to the same data at the same time. With more concurrency, a greater number of transactions can complete in a given timeframe.

Access option and lock mode can also influence the use of sequential block buffering and the use of certain parallel access plans.

This subsection provides an introduction to access options, lock mode, and lock granularity. For more information, see the *SQL/MP Reference Manual*.

Access Option

SQL provides these access options that affect the characteristics of locks:

- Browse access

  Browse access ignores existing locks and does not acquire any locks. A DML statement using this option reads data locked by other users. This option is allowed only for querying data (not for updating data). Browse access is also known as read through locks or dirty reads. For a UNION operation, browse access applies only to the portion of the query for which it is specified, and does not necessarily apply to the entire query.

- Repeatable access

  Repeatable access locks all data accessed through the DML statement. If a transaction reads the same row multiple times, the row will contain the same data. Range locks prohibit the insertion of rows with keys between the first key and the last key of the range.

Each row locked with repeatable access stays locked until the corresponding transaction is committed.

- Stable access

  Stable access requires that when a row is retrieved, it must not be locked in an exclusive manner by another transaction. Stable access locks all data accessed through the DML statement. This access is useful for applications that want to read only committed data. Stable access is also known as cursor stability or test lock.

  In general, each row locked with stable access is released when the next row is read, if the row was accessed but not updated. For standalone SELECT statements with stable access, SQL releases the lock as soon as the row is returned to the application.

Stable access is the default.

I/O buffering options influence access option behavior. For example, stable access with VSBB locks all rows and does not release them until SQL is finished with the entire block. For more information, see Reducing Messages With Buffering Options on page 4-21.

The option you specify can influence the resources required for a query. In general, these rules apply:

- Use browse access if possible, but use it with caution—only if inconsistent data is acceptable.

  The BROWSE ACCESS option enables you to read data that is currently being updated or deleted. This option allows you to read data for which an associated transaction might not have been committed. This option is especially effective for a read-only database.

  BROWSE ACCESS requires no lock management and allows other techniques such as RSBB to be used. Many queries (such as those used for application navigation through a database) can use BROWSE ACCESS successfully.

  If potentially inconsistent data is unacceptable, do not specify browse access, but make sure repeatable access is used only where necessary. (To determine current access path, review the EXPLAIN plan as shown in Section 6, Analyzing Query Performance.) If you do use data accessed with BROWSE ACCESS, supply update protection by comparing the BROWSE data with actual database contents during update and delete operations to make sure that you do not lose interim updates.

  If you omit BROWSE ACCESS, the default locking mode is STABLE. The executor obtains a shared lock on each row to ensure that the rows are not changed while they are being read. If the database is not audited by TMF, stable access is especially expensive because it prevents the executor from using some of its more efficient scanning mechanisms.

When using BROWSE ACCESS in SQLCI for a query that might run for a long time, disable transaction generation before starting the query. By doing so, you prevent the lengthy execution of an automatically generated transaction.

To disable transaction generation for user-defined DDL or DML statements, execute this statement:

```
SET AUTOWORK OFF ;
```

See the *SQL/MP Reference Manual* for information on this statement.

- To encourage parallel execution of queries:
    - Use the BROWSE ACCESS or REPEATABLE ACCESS option instead of STABLE ACCESS. The BROWSE and REPEATABLE options typically allow parallel execution when the STABLE option does not (unless you specify a table lock).
    - For nonaudited tables, use the BROWSE ACCESS option whenever possible.
    - For audited tables, use the BROWSE ACCESS, STABLE ACCESS with table locking or REPEATABLE ACCESS option whenever possible.
- For interactive queries that access large result sets, avoid the use of REPEATABLE ACCESS because of potential contention and delays.
- Avoid the use of REPEATABLE ACCESS when locking a range of rows. SQL locks all rows in the entire range, and no other transaction can access, delete, or insert rows within that range. Instead, specify shared access or specify only small ranges of rows and use multiple open cursors.
- Consider specifying REPEATABLE ACCESS for multistep updates or deletes.
- For single SELECT statements followed by an update operation, request REPEATABLE ACCESS IN EXCLUSIVE MODE. This strategy avoids lock escalation from shared to exclusive and avoids possible deadlocks in those situations.

This example retrieves information about employees from the EMPLOYEE table. Because browse access is specified, no locks are held while the query is processed.

```
SELECT LAST_NAME, FIRST_NAME, SALARY
  FROM EMPLOYEE
  WHERE SALARY > 50000
  BROWSE ACCESS ;
```

Lock Mode

Lock mode controls access to locked data. The two modes are these:

- Exclusive. The lock owner can access and modify the data. Users who have specified browse access can read the data. This is typically specified for multistep updates or deletes, where data is selected and then updated or deleted, to eliminate the need to convert a shared lock (for the select) to an exclusive lock (for the update or delete).

  Exclusive locking mode disables index-only access, because it indicates that the base table row will be deleted or updated.

- Shared. Multiple users can lock and read the same data. Inserts, updates, and deletes, however, can only be done by the lock owner.

In the SELECT statement, you can specify IN EXCLUSIVE MODE or IN SHARE MODE.

Exclusive and shared locks can be used at either the table or row level of granularity.

Lock modes are the same whether you choose stable access or repeatable access. Lock mode is sometimes determined by the system. SQL ensures that an exclusive lock is in effect for modified data. For data accessed but not modified, SQL usually provides a shared lock.

Lock Granularity

Granularity controls the number of rows affected by a single lock. Granularity can refer to a table, a partition, a subset of rows, or a single row.

The level of concurrency decreases as the size of the lock unit increases.

The LOCKLENGTH attribute for a base table controls the granularity of row locks for the table. You can control table locks with the LOCK TABLE and CONTROL TABLE statements; otherwise, the system determines the granularity by considering the access option you specify, the table size and definition, and the estimated percentage of rows that the query would access.

SQL might choose a different access path if a table lock can be used. In SQL, you can specify the use of table locks with these commands:

- In the LOCK TABLE statement, you can choose either the EXCLUSIVE or SHARED option:

  ```
  LOCK TABLE table-name
  ```

- You can use the TABLELOCK option to control system-chosen table locks from an SQL compile-time directive. For example, this statement instructs the SQL compiler to use a table lock for any subsequently compiled DML statements that specify the CUSTOMER table:

  ```
  CONTROL TABLE table-name TABLELOCK ON
  ```

LOCK TABLE is an executable SQL statement. If you use LOCK TABLE, the SQL compiler is not aware of the table lock before compiling subsequent statements because each SQL statement is compiled independently. The LOCK TABLE statement might even be in a separate program from subsequent statements.

CONTROL TABLE is a compiler directive. If you use CONTROL TABLE, the SQL compiler is aware of the table lock before compiling queries that reference the specified table.

Requesting a table lock eliminates the overhead of row locking and therefore reduces the overhead on queries that access many rows. By requesting a table lock (or by specifying browse access), you make it more likely that VSBB and parallel execution will be selected. Table locking, however, inhibits concurrency; other users cannot access the table while it is locked. For batch-type queries without OLTP activity, an application should request table locks with the CONTROL TABLE directive.

Waiting For Locks

You can use the RETURN/WAIT IF LOCKED option to enable or disable the wait mechanism for lock requests on data that is already locked by other users. This example enables the wait mechanism so that a lock request on locked data is held for 60 seconds (the default) before a timeout occurs:

```
CONTROL TABLE SALES.CUSTOMER WAIT IF LOCKED TIMEOUT DEFAULT
```

Performance Implications

By changing the characteristics of locks, you might increase the number of transactions that can be handled in a given time frame. The more concurrency, the greater number of transactions that can complete.

Access option (browse, stable, repeatable) and lock mode (exclusive or shared) can affect the query execution plan chosen by the optimizer. Consider these:

- Exclusive mode disables index-only access because exclusive mode usually indicates that the base table row will be deleted or updated.
- Access option can affect whether sequential block buffering is chosen. For example, real sequential block buffering (RSBB) is used only when browse access is specified, or the entire table is locked with a CONTROL TABLE directive.
- Stable access often prohibits parallel execution of queries.

Reducing Messages With Buffering Options

This subsection describes how SQL retrieves data from the disk process. Retrieval can be by row or by block. The size of a block is specified for a table when the table is created.

Types of Buffering

The optimizer can choose between these types of access:

- Single row—SQL returns data from the disk process one row at a time.
- Sequential block buffering (SBB):
  - Real Sequential Block Buffering (RSBB)—SQL obtains data from the disk process one block at time. Data blocks sent to the requesting process are exact copies of what is on the disk.
  - Virtual Sequential Block Buffering (VSBB)—the disk process builds a block in the disk process that has only the data the requester is interested in. Thus, the disk process performs column selection (projection) and row selection (restriction). The data can come from one or more table blocks. The data area used for this is called a virtual block because it is not an exact copy of data on disk.

The optimizer attempts to minimize the number of messages and the amount of data transferred between the file system and the disk process. Using SBB is one way to achieve this goal.

When an application uses SBB (either RSBB or VSBB), SQL retrieves a whole block of rows at a time. All rows in the block are locked. When the application process requests the next row, the file system returns the next row from the copy of the physical block of rows. Therefore, SBB reduces the number of messages between the file system and the disk process by the file's physical blocking factor (the number of rows per block).

If SBB is used, SQL chooses the type of SBB (virtual or real) to use. The optimizer does not, however, always choose SBB, because SBB involves a certain amount of overhead.

The following subsections describe each buffering mode in more detail, using an INVNTRY table defined with these columns:

```
ITEM_NAME (20 bytes)
RETAIL_PRICE (4 bytes)
ITEM_ON_HAND (4 bytes)
COMMENTS (400 bytes)
```

The INVNTRY table has these characteristics:

- The total size of each row is 428 bytes (20 + 4 + 4 + 400).
- The table contains 100 rows.
- There are 90 items with a RETAIL_PRICE value greater than 10.
- There is no alternate index.
- The block size is 4096 bytes.
- There is no slack space in the table.

This query requests the name and retail price of all items whose retail price is greater than 10:

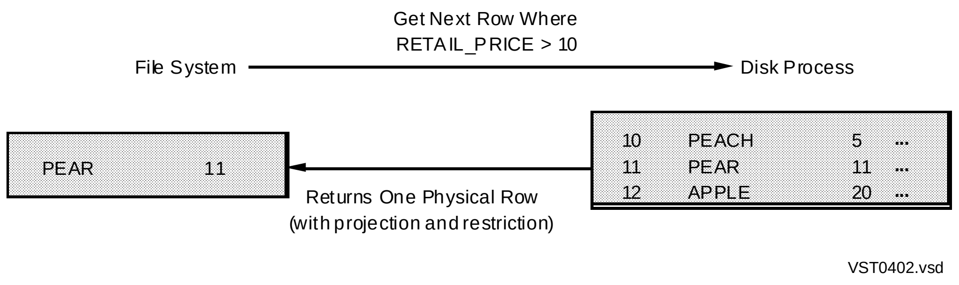
```
SELECT ITEM_NAME, RETAIL_PRICE
   FROM INVNTRY
   WHERE RETAIL_PRICE > 10
   FOR BROWSE ACCESS ;
```

SQL evaluates the query by reading the table sequentially.

Single-Row Access

With single-row access, each request for a row from the file system causes a row to be returned from the disk process, as shown in Figure 4-2.

Figure 4-2. Single-Row Access



The disk process returns each row that satisfies the predicate (RETAIL_PRICE > 10) to the file system. To evaluate this query, the file system sends 90 messages to the disk process. The disk process transfers 2,160 bytes of data (90 rows at 24 bytes per row) to the file system.

Guidelines for Choosing Single-Row Access

Single-row access is chosen when either a single row or a small number of rows satisfy the query—or when SBB is explicitly disabled by the user (by using the CONTROL TABLE directive, described in this subsection).

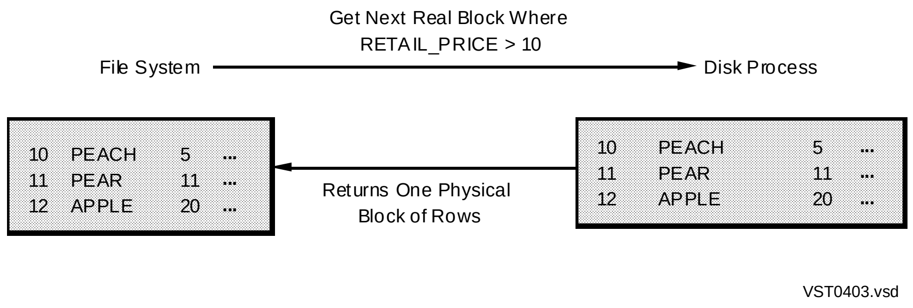
If a WHERE clause matches more than one row but you are interested in only the first row returned, set SBB off (and request single-row access) using the CONTROL TABLE SEQUENTIAL READ OFF directive. Similarly, if you are inserting only one row, use CONTROL TABLE SEQUENTIAL INSERT OFF to disable SBB.

Real Sequential Block Buffering (RSBB)

If a query accesses most of the rows in a table and most of the columns in each row, it can be beneficial for the disk process to return a complete block of rows to the file system rather than returning one row at a time. This process is called real sequential block buffering (RSBB).

RSBB reduces the number of messages it would otherwise take to return all qualifying rows, one at a time, to the file system. Figure 4-3 shows the disk process returning a copy of a physical block of rows to the file system. When the executor requests the next row, the file system does the projection and restriction of data and returns the next row to the executor. Therefore, the number of requests (messages) between the file system and the disk process is reduced by the file's physical blocking factor (that is, the number of rows per block).

Figure 4-3. Real Sequential Block Buffering (RSBB)



The disk process returns a physical block of rows to the file system. The file system examines each row in the returned block and tests the rows against the predicate. After all the rows in the block have been processed, the file system requests another block.

The efficiency of RSBB access can be estimated by dividing the number of useful bytes per physical block by the block size.

Slack Space

RSBB efficiency is diminished if there is slack (unused) space within the data blocks.

In the preceding example, 100 rows are transferred from the disk process to the file system; however, the file system sends only 12 messages to the disk process (assuming four-kilobyte data blocks with no slack).

If there is fifteen percent slack space within data blocks, then each physical block contains eight actual data rows instead of nine and the number of messages increases from 12 to 13.

For more information about slack space and how you can specify slack space, see the *SQL/MP Reference Manual*. The FUP INFO `filename`, STAT command displays the average number of rows per block and average percent slack space per block. Note that it processes the whole table, so use with caution if the file is large. For more information, see the *File Utility Program (FUP) Reference Manual*.

Guidelines for Choosing RSBB

RSBB is used when the disk process can do only a minimal amount of filtering (selection and projection). For example, this query requests the whole INVNTRY table:

```
SELECT * FROM INVNTRY ;
```

In general, SQL uses RSBB when all of these are true:

- Most rows examined satisfy all the predicates.
- More than two-thirds of a row is to be returned.
- More than two-thirds of the table or index is on the local node.
- Browse access is specified or the entire table is locked with a CONTROL TABLE directive.

Depending on the amount of slack, the optimizer might choose VSBB to reduce the number of messages. If there is little or no slack, then VSBB offers no advantage, so the optimizer choose RSBB if possible.

RSBB is used for SELECT operations if the table is accessed through an index from a protection view that has a selection expression and if at least one of these is true:

- BROWSE ACCESS is specified for the request
- The TABLE LOCK ON option of CONTROL TABLE is specified
- The TABLE LOCK OFF option of CONTROL TABLE is not specified and a table lock is chosen by the optimizer

RSBB is used for UPDATE and DELETE operations if a table lock is requested and less than 1 row per block is to be updated.

If a query accesses a protection view, index-only access implies use of RSBB if the view has a WHERE clause. The optimizer does not use index-only access with a protection view unless SQL can also use RSBB.

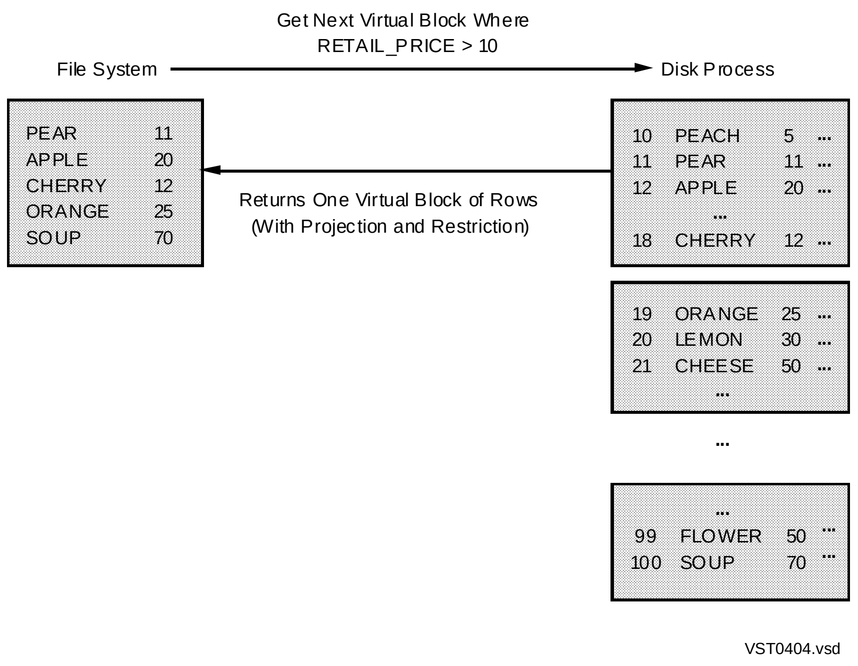
Virtual Sequential Block Buffering (VSBB)

The disk process can construct a virtual block that contains the qualified rows and selected columns to be returned to the file system. This process is called virtual sequential block buffering (VSBB).

Figure 4-4 on page 4-25 illustrates how the disk process does the projection and restriction of data, which comes from several physical blocks.

VSBB reduces the number of messages between the file system and the disk process. Further, it reduces the amount of data transferred between the file system and the disk process.

Figure 4-4. Virtual Sequential Block Buffering (VSBB)



The disk process returns a block of rows to the file system, but the block contains only the requested columns from rows that satisfy the predicate (because the disk process does the projection and restriction of data). So, only the columns ITEM_NAME and RETAIL_PRICE are returned to the file system.

Therefore, the disk process returns 2160 bytes of data (90 rows at 24 bytes per row) to the file system. Because the answer to the query can be contained in one 4 kilobyte page, the file system sends only one message to the disk process.

VSBB is also used for insert and update operations. It significantly reduces the number of messages passed between the file system and the disk process.

Guidelines for Choosing VSBB

For a sequential scan, VSBB is chosen when one or more of these is true:

- Less than two-thirds of a row is retrieved or examined.
- Most records examined do not satisfy all of the predicates. (The difference between the table and indexes selectivities is large.)
- More than one-third of the table is on a remote node.

VSBB is used for audited tables and cursor stability if less than 1 out of 16 rows examined is to be returned and if any of the preceding conditions are met.

VSBB is used for UPDATE and DELETE operations if no more than one row is to be updated out of 32 rows examined.

VSBB is not used for nonaudited tables if any of these is true:

- The syncdepth value is greater than 0.
- A table lock is not used.
- The table contains alternate indexes, and a varying-length column is being updated.

When an operation is buffered, data is transferred between the file system and the disk process a block at a time instead of a row at a time. Consequently, virtual sequential block buffering (VSBB) improves the performance of queries by reducing the number of messages exchanged and the amount of data transferred between the file system and the disk process.

To find out if the optimizer is choosing VSBB, use the EXPLAIN utility, described in Section 6, Analyzing Query Performance.

Effects of VSBB on Concurrency

The use of VSBB can cause concurrency problems because it requires locks. Increased lock waits and timeouts can occur for reasons described in these subsections. If you are experiencing concurrency problems, consider disabling VSBB (by using the CONTROL TABLE directive).

For sequential read operations, the disk process locks all rows scanned (rather than a row at a time). Consequently, SQL operations that use VSBB, even with stable access, can acquire more locks that remain in place longer than operations that do not use VSBB. (The EXPLAIN utility, described in Section 6, Analyzing Query Performance, lists VSBB and STABLE ACCESS when both are in use.) Stable locks on rows are released only when the program is done with the block. You can disable VSBB for read operations by specifying this directive:

```
CONTROL TABLE * SEQUENTIAL READ OFF
```

For sequential update operations, the compiler performs a sequential read before performing the update. You can disable VSBB for update operations by specifying this directive:

```
CONTROL TABLE * SEQUENTIAL UPDATE OFF
```

When requesting an insert into a key-sequenced table that uses a SYSKEY column or a timestamp as the primary key, VSBB is usually chosen for the operation, because inserts into such a table will very likely be sequential. If concurrent servers are inserting into the table, a high percentage of lock waits and timeouts might occur. You can disable VSBB for insert operations by specifying this directive:

```
CONTROL TABLE * SEQUENTIAL INSERT OFF
```

Note. Disabling sequential insert or update operations does not automatically disable sequential read operations. You must specify CONTROL TABLE SEQUENTIAL READ OFF to disable sequential read operations.

Comparison of Buffering Types

This discussion compares the three types of buffering techniques, using the example described earlier in this subsection.

Table 4-1 shows the number of messages transferred between the disk process and the file system, the number of rows and the number of bytes transferred from the disk process to the file system, and the number of bytes read from disk by the disk process for each type of buffering mode: single row, RSBB, and VSBB.

Table 4-1. A Comparison of Buffering Modes

Buffering Mode	Number of Messages Between File System and Disk Process	Number of Bytes Read From Disk by Disk Process	Number of Bytes Transferred to File System for Each Message	Number of Rows Transferred to File System
Single Row	90	48K	2.2K	90
RSBB	12	48K	48.0K	100
VSBB	1	48K	2.2K	90

Note. If stable access is used, VSBB requires 3 messages, because a maximum of 32 rows can be kept locked within any virtual sequential block.

As shown in this table, the choice of buffering mode can have a dramatic influence on the number of messages required for a query and can, therefore, influence performance in a major way.

For online transaction processing, you might find that single-row access gives the fastest response time, because the disk process responds immediately upon finding the first qualifying row. Alternately, with VSBB, the response might be delayed while the disk process continues scanning to fill the virtual block. To disable SBB, use the SEQUENTIAL READ OFF option of the CONTROL TABLE statement. (For more information, see the CONTROL TABLE SEQUENTIAL READ statement in the *SQL/MP Reference Manual*).

Requesting Buffering

You can use the CONTROL TABLE directive to set buffering to ON, OFF, or ENABLE for sequential insert, read, or update operations. The ENABLE setting lets the system choose whether to use SBB or not for a specific query; this is the default for read operations.

This example requests buffering for sequential UPDATE operations:

```
CONTROL TABLE SALES.CUSTOMER SEQUENTIAL UPDATE ON
```

The preceding directive requests buffering of update operations by the file system for the disk process.

In a host language program, specific placement rules might apply to the CONTROL TABLE directive. For more information, see the *SQL/MP Programming Manual* for your host language.

Optimizing Sequential Access With Block Buffering

When a row is inserted into an SQL table or view, each INSERT statement results in a single message to the disk process. For files in which inserts are random throughout the file, using the INSERT statement in this way is the best method. When inserts are performed in key groups of sequential nature, however, as in batch operations, it is more efficient to block the inserts, grouping sequential keys in one message to the disk process.

You can use the CONTROL TABLE statement to control insert operations to allow for sequential block buffering or to force single writes. For audited tables, the default option for CONTROL TABLE SEQUENTIAL is ENABLE, which enables the system to decide which method (sequential block buffering or single messages) is the most efficient for these operations. For nonaudited tables, the default option for CONTROL TABLE SEQUENTIAL is OFF, indicating the single messages method.

You can control the option programmatically so that batch throughput performance can be increased. For more information, see the *SQL/MP Programming Manual* for your host language.

The CONTROL TABLE SEQUENTIAL ENABLE option is available for update operations as well as for the insert function.

Effects of Cursor Operations on Performance

The optimizer often chooses VSBB when compiling a cursor definition.

You should be aware, however, that certain operations invalidate buffering for cursor operations, and performance can be degraded. For more information, see the *SQL/MP Programming Manual* for your programming language.

Controlling the Opening of Tables, Views, and Indexes

The NonStop SQL/MP file system opens (grants access to) objects when directed to do so by the NonStop SQL/MP executor. Tables, views, and indexes, and partitions of these objects, are usually opened on demand. SQLCI users or application programs do not influence when tables, views, and indexes are opened or closed.

With the open-on-demand feature, each object or partition is opened during the processing of individual SQL statements. When many statements require the opening of many different tables, views, and indexes used as access paths, or partitions of these objects, this feature can require additional time at the first execution of individual statements.

You can, however, control this overhead by using the CONTROL TABLE *object-name* OPEN directive to override open-on-demand. When the CONTROL TABLE directive is processed, one or more objects can be opened immediately. You can use this directive to open all tables, views, indexes, and any partitions of these objects as soon as a program is started.

This approach shifts the processing time for opening objects to the beginning of program execution and away from the times when the individual statements that require the objects are processed. After an object is opened, it stays open until the program is stopped.

When you enter CONTROL TABLE *table-name* OPEN ALL PARTITIONS, the table and all of its partitions and indexes are all opened.

When you enter CONTROL TABLE *view-name* OPEN ALL, the view and the underlying tables and indexes are opened.

△ **Caution.** Use the CONTROL TABLE statement with the OPEN ALL option only if all these are true:

- All open activities must occur when the program starts (add a “dummy” call to the cursor during initialization).
- The object containing the cursor eventually accesses all partitions.
- The plan for the cursor is not a parallel plan.

In addition to the open-on-demand feature, NonStop SQL/MP also ensures that tables are automatically created to provide local autonomy. Local autonomy ensures that you can access local data regardless of the availability of other local dependent objects or remote dependent objects, if the local data that is available can fully satisfy your request.

With local autonomy, the compiler stores information about partitions that underlie tables and indexes, enabling the file system to open any available partitions as needed.

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 key columns than you want. On these occasions, you can use the ACCESS PATH . . . MDAM ON option in a CONTROL TABLE directive, and apply the USE option to specify the number of key columns for MDAM to use.

Suppose four columns of the key have MDAM predicates. You have determined the unique entry count of the fourth key column used in a predicate. (You have found the count by updating statistics and querying the catalog table for the column). If you do not want MDAM to step through all the different values for the fourth column, you can specify the number of columns for MDAM to use. If you specify three, then only predicates on the first three 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 DEFAULT for the number of key columns allows MDAM to choose the number of key columns.

For more information on MDAM, see Transformation of Predicates on page 3-4.

Controlling MDAM’s Use of DENSE or SPARSE Algorithms

When you use a CONTROL TABLE directive with MDAM ON, you can specify whether the optimizer should use an adaptive DENSE or SPARSE algorithm for all columns during row access. An adaptive algorithm is an algorithm chosen by the optimizer but might not be chosen by the executor. For example, if the optimizer chooses a DENSE algorithm and the executor finds a DENSE algorithm is inefficient for accessing a certain column, the executor adapts by switching to a SPARSE algorithm. It switches back to the DENSE algorithm as soon as it finds the value it is seeking.

You can determine the density or sparsity of data by the values in a column. For example, if column A contains the values 1, 2, 3, and 4, it has dense data distribution.

If you choose a DENSE algorithm, the executor starts with 1 and increments the value sequentially to obtain the next values for column A.

However, if the values for column A are 25, 135, 400, and 525, then DENSE would not be a good algorithm. Too many accesses would be made if each value were sequentially incremented until the next value were found. If you specified a SPARSE algorithm, the executor would process all qualifying rows for A = 25 and then do an extra position to find the next value of A that is greater than 25.

If statistics cause the optimizer to make a mistake and choose DENSE instead of SPARSE, or SPARSE instead of DENSE, you can change the choice by specifying the correct option in the CONTROL TABLE directive. However, because the executor does an adaptive DENSE or SPARSE, it switches accordingly when it finds that the chosen algorithm is not efficient for the column it is accessing.

If you use a CONTROL TABLE directive to force the use of DENSE, the optimizer still uses SPARSE for character and float data types. If you specify the SYSTEM option, the system chooses the algorithm for each column.

When the optimizer chooses a SPARSE algorithm, the executor executes only the SPARSE algorithm. When the optimizer chooses a DENSE algorithm, the executor uses a SPARSE algorithm to retrieve the first existing key value but then switches to DENSE. It continues to use DENSE until it makes several misses and then switches back to SPARSE to get the next existing key value.

For more information on MDAM, see Transformation of Predicates on page 3-4.

Controlling the Creation of NonStop SQL/MP Processes

The SQL catalog manager and SQL compiler are always started upon demand for their services. If these processes are not already running when individual SQL statements are submitted, a longer response time for statements results. Furthermore, after the processes are started, they normally remain active for about five minutes. Therefore, you can improve the performance of certain operations if you group the operations within your program to take advantage of this timing. For example, if a program needs to create three temporary tables, you could group the three CREATE TABLE statements together rather than including them at the separate points where the program might naturally need them. Beyond these techniques, you have no other control over the life cycle of the SQL processes.

Enhancing Sort Performance

You can enhance the performance of sorts within SQL queries in several ways:

- Use subsorts, configured by specifying appropriate SUBSORT attributes in =_SORT_DEFAULTS DEFINEs.
- Direct FastSort to use additional memory when sorting data by setting the VLM attribute ON in a user-specified SORT DEFINE or a =_SORT_DEFAULTS DEFINE. When ON, the VLM attribute allows FastSort to use up to 127.5 Megabytes of extended memory (if available on the system), enhancing performance significantly. FastSort uses the additional memory in either of two ways:
    - To extend the size of the sort tree beyond its default limit of 32 kilobyte nodes if the extended size would permit the sort to be done in a single pass
    - To store some intermediate sorting runs and thereby reduce disk input-output to a minimum if the sort is still too large for a single pass
- Avoid using a busy processor for your sort operations. Instead, specify the processor attribute in a =_SORT_DEFAULTS DEFINE to direct all main sort and subsort operations to another processor.
- Adjust the execution priority of a sort operation by defining the PRI attribute in a =_SORT_DEFAULTS DEFINE to alter the priority.
- Avoid excessive contention for your sort scratch file by using the SCRATCH attribute in a =_SORT_DEFAULTS DEFINE to redirect scratch operations to another volume. In particular, the primary partition's volume, which is the default volume for the scratch file, often becomes very busy; if your scratch file resides on this volume, you might want to move it to another. For cases in which performance is vitally important, you might dedicate an entire volume to sort scratch operations.
- Use SCRATCH, SCRATCHON, AND NOSCRATCHON in a SORT DEFINE so that FastSort can build a pool of scratch files on appropriate volumes. For more information, see the *FastSort Manual.*

For more information about invoking FastSort from SQL, see the *FastSort Manual.*

Understanding Concurrency

Concurrency is access to the same data by two or more processes at the same time. The degree of concurrency available depends on the purpose of the access, the access mode, and whether virtual sequential block buffering (VSBB) is used for the access.

NonStop SQL/MP provides concurrent database access for most operations, but some longer-running DDL and utility operations can reduce concurrent access.

For information about limits on concurrency, see the *SQL/MP Reference Manual*. For information about maximizing concurrency during DDL operations, see the *SQL/MP Installation and Management Guide.*

Minimizing Overhead of Query Programs

NonStop SQL/MP supports several features that can help you minimize the overhead of query programs:

- For query programs that use static SQL statements, name resolution can occur at execution time. Thus, if you want a transaction to execute against one of several different tables, this feature makes it possible to do so without using dynamic SQL, thus avoiding the compilation overhead of dynamic SQL. To specify execution-time name resolution, use the CONTROL QUERY BIND NAMES directive.
- After DDL operations or other activity that might invalidate a query plan, you can minimize recompilation of statements within a program by using the similarity check to avoid recompilation if the operation did not affect the query plan of the specific statement.
- You can install programs in a new system without recompiling them by using the REGISTERONLY option.
- You can install programs in a new system without recompiling or registering them by using the NOREGISTER option.

For more information about these features, see the *SQL/MP Programming Manual* for your host language, the *SQL/MP Reference Manual*, and the *SQL/MP Installation and Management Guide*.

5 Selectivity and Cost Estimates

NonStop SQL/MP uses selectivity and cost when choosing an execution plan. This section describes these topics:

How the Optimizer Estimates Selectivity

Selectivity is an estimate of the number of rows in a table or an index that satisfy a given search condition and is represented as a percentage of rows, from 0 to 100. It is central to the selection of an access plan by the optimizer. It depends on column statistics maintained in the SQL catalog and on the predicates specified for a given query.

The efficiency of a given index is determined based on its selectivity. If the control statement INTERACTIVE ACCESS ON is specified, however, the optimizer attempts to use an index, if feasible, without considering selectivity. The control statement indicates that only the first few rows of the result set are pertinent to the request.

The number of rows that are examined affects the number of messages that are exchanged between the file system and disk processes to retrieve the data, the number of input/output operations that are performed, and so on.

There are three levels of selectivity:

- Predicate selectivity is the fraction of rows in a table that satisfy the predicate.
- Table selectivity is the fraction of rows that satisfy all the predicates of a query.
- Index selectivity is the fraction of index rows that must be examined in evaluating a query.

Selectivity influences the optimizer's choice of these:

- Access path (base table, alternate index, or index only)

  For example, if the restriction specified by a WHERE predicate does not result in a low enough selectivity to justify alternate-index access, base-table access is chosen instead. (For more information about access paths, see Optimizing the Access Path on page 4-4.)

- Join order

  The selectivity of each table helps determine the optimizer's choice of the outer and inner table, because it helps determine cost. Cost and available access paths are the determining criteria for join order.

- Types of sorts performed (in-memory user process sort or external physical sort)

The following subsections describe general selectivity computations; predicate, index, and table selectivity; default selectivity; and join and grouping selectivity.

Computing Selectivity

The optimizer estimates selectivities based on statistics obtained prior to the compilation of queries; therefore, SQL must have current statistics to work from. When you specify an UPDATE [ALL] STATISTICS command, SQL inserts these statistics into the COLUMNS, FILES, BASETABS, AND INDEXES catalog table.

Furthermore, the optimizer operates as if the data were uniformly distributed in each column within the range specified by the statistics and the unique values were uniformly distributed between the lowest and highest values.

Selectivities are cumulative; that is, the combination of predicate selectivities determines the selectivity for a table or an index.

The optimizer uses these statistics to compute selectivities:

- The second-high and second-low values of a column (SECONDHIGHVALUE and SECONDLOWVALUE in the COLUMNS table). To avoid extreme values that might be very different from the rest of the values, SQL does not use the first-high and first-low values of a column.
- The unique entry count (UEC), which is the number of unique values of a column (UNIQUEENTRYCOUNT in the COLUMNS table), used to calculate the selectivity for equal (=) and not equal (<>) comparisons, IS [NOT] NULL clauses, and join selectivity.

SQL estimates the overall UEC by linear interpolation. For partitioned tables, the UEC equals the total number of unique values divided by the number of partitions. Any remainder is added to the primary partition. The quotient H2/L2 is the same for all partitions.

By default, UPDATE STATISTICS reads partitions smaller than 1,000 blocks in their entirety and uses sampling for partitions of 1,000 blocks or larger. You can influence the amount of data to be examined by specifying the exact number of blocks:

```
SAMPLE n BLOCKS
```

For more information about UPDATE STATISTICS, see Section 6, Analyzing Query Performance. For a complete description of the UPDATE STATISTICS command and the COLUMNS catalog table, see the *SQL/MP Reference Manual*.

Predicate Selectivity

Predicate selectivity is the estimated percentage of rows in a table or an index that satisfy a given predicate. The optimizer uses the selectivity of key predicates to estimate the number of rows to examine; it uses the selectivity of the rest of the predicates to estimate the number of rows that qualify. Those rows that actually satisfy the predicate are selected for processing at the next stage.

Predicate selectivity is a number that expresses the effectiveness of a predicate as a filter. The number is a probabilistic estimate.

For example, suppose that there are 100 items, numbered from 1 to 100, in the INVNTRY table. The selectivity of this predicate is 0.9 or 90 percent, because 90 out of 100 rows satisfy the condition specified by the predicate:

```
ITEM_NO > 10
```

Alternately, if the selectivity of the predicate (SALARY < 50000) equals .6667, then the optimizer has estimated that 66.67 percent of the rows in the table contain a value less than 50,000 in the column SALARY.

In general, these rules apply:

- Transformations are applied according to the rules listed in How the Optimizer Processes Predicates on page 3-4.
- For predicates of the form column = constant, the selectivity of the predicate is:

  ```
  1 / (UNIQUEENTRYCOUNT)
  ```

- Reversing a predicate does not affect its selectivity. For example, these predicates have identical selectivities:

  ```
  X >= Y
  Y <= X
  ```

Numeric Range Predicates

Predicates with one of these forms are called exact numeric range predicates:

```
column  <  constant
column  <= constant
column  >  constant
column  >= constant
```

The optimizer distinguishes between < and <=, and > and >=, as follows:

- *column* < is treated as *column < constant*
- *column* <= *constant* is treated as *column < constant + LSB*; the least significant bit of the constant is incremented by 1.
- *column* > *constant* is treated as NOT *column* < *constant + LSB*; the least significant bit of the constant is incremented by 1.

- column >= is treated as NOT *`column < constant`*

For numeric ranges, the optimizer uses the constant to interpolate between the SECONDLOWVALUE and SECONDHIGHVALUE obtained by UPDATE STATISTICS. For example, if the RETAIL_PRICE column of the INVNTRY table has a SECONDHIGHVALUE of 99 and a SECONDLOWVALUE of 2, the selectivity of the predicate:

```
RETAIL_PRICE > 10
```

is the second-high value minus the supplied value, divided by the quantity (SECONDHIGHVALUE minus the SECONDLOWVALUE):

```
99 - 10 / 99 - 2, or approximately 0.92
```

Ideally, this process would allow exact computation of the number of values selected from a column that has a uniform distribution of data values. In practice, if a predicate selects a larger proportion of records between the SECONDLOWVALUE and the SECONDHIGHVALUE, then that predicate has a higher selectivity than one that selects a smaller proportion of records. The optimizer cannot determine the distribution of data. Therefore, when data is unevenly distributed, a higher selectivity does not imply a greater number of selected rows.

Combinations of Predicates

When predicates are connected by the AND or OR operator, SQL calculates selectivity in one of two ways. If the query contains no range predicate, then selectivity is calculated according to rules of probability theory. For queries with range predicates, a different calculation is used.

Selectivity When No Range Is Used

When no range is used, SQL calculates selectivity as follows:

- If predicates are connected by the AND operator, their combined selectivity is estimated to be the product of their individual selectivities. Thus, a combined selectivity for these query equals .1 * 0.6667 or 0.06667:

```
WHERE empno = :hv        (selectivity = 0.1)
AND SALARY < 50000       (selectivity = 0.6667)
```

- If predicates are connected by the OR operator, their combined selectivity is estimated as the sum of the individual selectivities minus the product of the individual selectivities. Thus, a combined selectivity for these query equals (0.1 + 0.1 - (0.1 * 0.1)), or 0.19:

```
WHERE EMPNO = :hv        (selectivity = 0.1)
OR SALARY < 5000         (selectivity = 0.1)
```

These rules rely on the assumption that the truth values of the individual predicates are independent of one another. This estimate can become inaccurate if the predicates are highly correlated or redundant.

Selectivity for Range Predicates

When a range is used in a predicate, selectivity is calculated as follows:

1. The SECONDLOWVALUE is subtracted from the SECONDHIGHVALUE. The result is the total number of values in the range. For example, if the SECONDLOWVALUE is 0 and the SECONDHIGHVALUE is 1000, the result is 1000 minus 0, or 1000.
2. The number of values between the begin-range value and the total number of values in the range is divided by the total number of values in the range. Consider this example:

   ```
   WHERE A >=  100
   AND A <= 200
   ```

   The result is (1000-100)/1000 or .90.
3. Then the number of values between the end-range value and the total number of values in the range is divided by the total number of values in the range. The result for the same example is (1000-200)/1000 or .20.
4. Instead of multiplying the two results, as is done when no range is used, the two results are added together. In the example, the result is .90 + .20 or 1.10.
5. Selectivity is calculated by subtracting 1.00 from the result. In this case, the selectivity is 1.10 - 1.00 or 10 percent selectivity.

Selectivity of Multivalued Predicates

For a multivalued predicate, SQL uses the formulas in Table 5-1 to determine selectivity. These formulas are applied recursively so that all comparisons participate in the result.

Table 5-1. Selectivity Formulas for Multivalued Predicates

Predicate Form	Formula
a,b < c,d	The selectivity of (a < c) plus the selectivity of (a = c AND b < d)
a,b <= c,d	The selectivity of (a < c) plus the selectivity of (a = c AND b <= d)
a,b >= c,d	The selectivity of (a > c) plus the selectivity of (a = c AND b >= d)
a,b > c,d	The selectivity of (a > c) plus the selectivity of (a = c AND b > d)

SQL calculates selectivity using all columns in the primary key and indexes according to the rules for AND and OR noted in the preceding subsection.

Beginning with version 3.0, if the operator is =, then all columns of the multivalued predicate are taken into account for selectivity. Otherwise, only the first pair of comparison values is used for the calculation.

NULL Values

The presence of NULL values in a column is considered when determining selectivity of IS NULL predicates and range predicates and in determining index selectivity for pairs of range predicates. For example, the selectivity of C IS NULL is 0 percent if column C has the NOT NULL attribute

If null values are present and the UEC for a column is two or less, SQL sets SECONDHIGHVALUE to the highest nonnull value in the table. If UEC is three or less, UPDATE STATISTICS sets the SECONDLOWVALUE to the lowest value in the table. The optimizer uses these statistics to determine whether SQL detected any null values in the column.

If UEC is 3 or more, Non Stop SQL operates as if nulls are present unless the column is defined with the NOT NULL attribute. If nulls are present, SQL operates as if the nulls occur in equal proportion with other values, and their frequency is estimated as 1 / UEC.

Index Selectivity

Index selectivity refers to the combined selectivity of the begin-key and end-key predicates for an access path:

- For primary-key access, index selectivity is the estimated percentage of rows in the base table that will be examined in evaluating the query.
- For alternate-index access, index selectivity is the estimated percentage of index entries that will be examined.
- For a table scan, begin keys and end keys are not applicable. All rows must be examined, so index selectivity is always 100 percent.

MDAM predicates also affect selectivity. For related information on MDAM, see The Effect of the MultiDimensional Access Method (MDAM) on Costs on page 5-17.

Another way to look at index selectivity is the number of rows actually accessed by the disk process.

To examine a row, the row must be read from disk. Rows that are outside the bounds set by the begin keys and end keys need not be read, however, so index selectivity strongly affects the cost of a scan operation. A low index selectivity implies a low cost estimate for the scan.

To determine index selectivity, the optimizer identifies index predicates for a specific access path. The optimizer then combines the selectivities as if they are independent of each other. Thus, the index selectivity is the combined selectivity of begin-key and end-key predicates for the access path. (Note that this strategy differs from index predicates, which are used in the calculation of table selectivity.)

The presence of NULL values in a column is considered when determining index selectivity for pairs of range predicates; for more information, see the preceding subsection “NULL Values on page 5-6.”

If index predicates are present, their selectivity is combined with the selectivity of the begin-key and end-key predicates. You can access this information using the EXPLAIN utility, under the titles "Pred. selectivity" and "Index selectivity."

Consider this query:

```
SELECT ITEM_NAME, RETAIL_PRICE
   FROM INVNTRY
   WHERE ITEM_NO = 20 ;
```

Suppose that there is a unique index on ITEM_NO in the INVNTRY table. If the index is chosen to evaluate the query, then the predicate (ITEM_NO = 20) is used as both the begin-key and the end-key predicate. Only one row must be examined because the index is unique. If there are 100 items, the index selectivity is 1 percent, or .01.

In contrast, if there is no index on ITEM_NO, then the predicate is classified as a base-table predicate. Because there is no begin-key or end-key predicate, it is necessary to examine the entire table, so the index selectivity is 100 percent.

Note. If your selectivity estimate is high but you know that the number of rows to be processed is small, consider using the CONTROL TABLE ACCESS PATH feature, described in Section 4, Improving Query Performance With Environmental Options. For example, this feature might be useful if you have a large table with an index on a column that has many duplicate values.

Table Selectivity

Table selectivity is the estimated percentage of rows in the table that satisfy all of the predicates in a query. The difference between index and table selectivity can be viewed as the difference between the selectivity of positioning predicates and the selectivity of all (positioning and nonpositioning) predicates. You can also view this difference as the difference between rows accessed by the disk process and rows actually returned by the disk process to the file system.

Consider this query:

```
SELECT ITEM_NAME, RETAIL_PRICE
   FROM INVNTRY
   WHERE ITEM_NO > 10 ;
```

Because the query has only one predicate (WHERE ITEM_NO > 10), and 90 out of 100 items in the table satisfy the search condition, the table selectivity is also 90 percent. More than one predicate can be specified in a query, however, so table selectivity is not always equal to predicate selectivity.

Table selectivity is the combined selectivity of all the predicates that participate in a scan operation. The combined selectivity is estimated as follows:

1. First, the optimizer uses the value estimated for the index selectivity—the combined selectivity of begin-key and end-key predicates for the access path.
2. Then, if there are other kinds of predicates involved, the optimizer multiplies the value estimated for the index selectivity with the values estimated for the other

predicates in this order: index predicates, base-table predicates, executor predicates.

The combined value is the estimated value for the table selectivity.

Table selectivity strongly influences the estimated cost for subsequent steps of the query execution plan. For example, suppose that the estimated selectivity for a table with 1,000 rows is 20 percent. Consequently, it is expected that 200 rows will be retrieved. This number is used to calculate subsequent cost. For example, if the rows need to be sorted after the scan, the expected number of rows strongly influences the sort cost estimate.

For an example of table selectivity with join operations, see Cost of Join Operations on page 5-15.

If there are no predicates, selectivity is 100 percent because all rows are selected for a full table scan.

Example Combining Predicate, Index, and Table Selectivity

This example selects data from EMP_TABLE. An index is defined on EMP_TABLE that consists of the columns EMP_DEPT, EMP_MGR, and EMP_START. The SELECT statement has six predicates, four of which are used as index predicates:

```
SELECT ADDRESS, PHONE, SPOUSE_NAME
  FROM  EMP_TABLE
  WHERE EMP_DEPT   =  :dept-num   AND
        EMP_MGR    =  :mgr-num    AND
        EMP_START  >  :min-start  AND
        EMP_START  <  :max-start  AND
        EMP_SALARY <> :min-salary AND
        EMP_SALARY <  :max-salary ;
```

The optimizer assigns a predicate selectivity to each combination of column, operator, and value, such as EMP_DEPT, =, and *:dept-num:*

```
                                                Predicate
                                               Selectivity
SELECT ADDRESS, PHONE, SPOUSE_NAME
  FROM  EMP_TABLE
  WHERE EMP_DEPT   =  :dept-num   AND          .0100
        EMP_MGR    =  :mgr-num    AND          .0100
        EMP_START  >  :min-start  AND          .3333
        EMP_START  <  :max-start  AND          .3333
        EMP_SALARY <> :min-salary AND          .9900
        EMP_SALARY <  :max-salary ;            .3333
```

Note that the optimizer combines both EMP_START predicates to obtain a selectivity of .111. Finally, the optimizer uses the selectivity in steps:

```
   .0100
 * .0100
 * .3333
 * .3333
 ---------
   .0000111    = index selectivity
 * .9900
 * .3333
 ---------
   .00000367  = table selectivity
```

Use of Default Selectivity Values

The optimizer uses default selectivity values when

- Statistics are not available because an UPDATE STATISTICS has not been performed.
- Statistics are not available because the table was empty when UPDATE STATISTICS was last done.
- A predicate involving a >, >=, <, or <= either
    - Cannot be used as a key; for example, a + 1 < 5 is not the same as a < 4
    - Compares a column with a host variable or parameter

If, for example, the value specified in a predicate is a host variable (as in a COBOL program), SQL cannot compute the selectivity because the value of the host variable is not known until run time.

In such cases, SQL operates as if an arbitrarily chosen default selectivity is in effect.

The default values depend on the type of predicate. Table 5-2 lists both computed values and default values for different types of predicates.

Table 5-2. Computed and Default Selectivity Values for Predicates (page 1 of 2)

Type of Predicate	Computed Value	Default Selectivity
Equals (=)	1 / UNIQUEENTRYCOUNT	1 percent
Not equals (<>)	1 – (1 / UNIQUEENTRYCOUNT)	99 percent
Greater than (>)	range / (SECONDHIGHVALUE – SECONDLOWVALUE)	33 percent
Greater than or equal to (>=)	range / (SECONDHIGHVALUE – SECONDLOWVALUE)	33 percent
Less than (<)	range / (SECONDHIGHVALUE – SECONDLOWVALUE)	33 percent
Less than or equal to (<=)	range / (SECONDHIGHVALUE – SECONDLOWVALUE)	33 percent

Table 5-2. Computed and Default Selectivity Values for Predicates (page 2 of 2)

Type of Predicate	Computed Value	Default Selectivity
LIKE	N. A	10 percent
NOT LIKE	N. A	30 percent
EXISTS	N. A	40 percent
NOT EXISTS	N. A	60 percent
IS NULL	0 (zero) percent for a column with the NOT NULL attribute; otherwise, 1 / UNIQUEENTRYCOUNT	0 percent for a column with the NOT NULL attribute; otherwise, 1 percent

For example, the selectivity of this predicate is 0.33:

```
RETAIL_PRICE > :host_variable
```

The selectivity of this predicate can be reasonably computed because the computation does not depend on the supplied value:

```
RETAIL_PRICE = :host_variable
```

If the default selectivity differs very much from the actual selectivity, SQL might choose an inefficient execution plan for the query. Therefore, you should periodically collect statistics with the UPDATE STATISTICS command.

Join Selectivity

For joins, the effective UEC for join columns equals the UEC of each column times the selectivity of predicates applied prior to joining. This calculation affects the choice of join method; for example, if UEC equals 1, hash join is not chosen.

Grouping Selectivity

A grouping operation is performed whenever a GROUP BY is specified. The elements of a GROUP BY list are either named columns or expressions. For each named grouping column with statistics, the optimizer uses the UEC as follows:

- Estimates the UEC prior to grouping.
- If there are single-table predicates in a WHERE clause on the column, the optimizer estimates the UEC as the product of the predicate selectivity times the lowest selectivity value on the grouping column times the UEC computed by UPDATE STATISTICS.
- If there is a join predicate on the grouping columns, the optimizer uses the UEC computed during plan generation.

The number of groups is then the product of the UECs of each grouping column, bounded by the number of rows being grouped.

Default selectivity is used when statistics are not available. If the grouping element is an expression, the element’s selectivity is based on the underlying columns in the expression.

Assigning Cost to a Query

Cost is an estimate of the amount of time the system takes to complete evaluation of a specific query. The optimizer uses cost as a factor in selecting an optimal access plan. Even though cost is an estimate and is not an exact measure, it is very useful for comparing the relative efficiency of different execution plans for a given query.

Cost has many components, including the number of physical I/O operations to perform, the number of instructions to execute, the number of sorts to perform, and the number of messages (local and remote), and the amount of data to be transferred between processes. The optimizer does not take into account the types of processors, disks, or communication lines when it calculates cost. So the cost of a query will not change if you use faster processors, disks, or communication lines.

To facilitate the computation of cost, all components of cost are expressed in the number of physical I/Os. The optimizer expresses cost in the equivalent number of physical I/Os that must be issued to complete the query.

Cost is only an estimate because there are many variables the optimizer does not have access to at compile time—or that SQL cannot control—such as:

- The type of processor can change at run time.
- The load of the system cannot be predicted for the time the query is executed.
- The elapsed time for the completion of a query varies depending a variety of runtime factors.
- The values of host variables cannot be predicted.
- The size and availability of cache can change. (For information about the effect of cache on query performance, see Section 4, Improving Query Performance With Environmental Options.)

In the final phase of query plan selection, the optimizer chooses the plan with the smallest numeric cost. To display this cost, use DISPLAY STATISTICS or EXPLAIN, described in Section 6, Analyzing Query Performance.

The following subsections describe how SQL calculates cost for various types of query operations.

Cost of Accessing Tables

The cost of using an index to access a table in a query that references only one table is as follows:

```
Cost(index) = Cost(physical I/O)
+ Cost(record overhead)
+ Cost(data transfer)
+ Cost(message)
+ Cost(sub-query)
+ Cost(sort)
```

If the cost of a component is less than one physical I/O, the cost of the component is truncated to zero.

If a query references multiple tables, the optimizer also considers the different combinations in which the tables can be joined. Each of these combinations is also assigned a numeric cost.

These paragraphs describe the elements used in the calculation of index cost and join cost.

Cost of Physical I/Os

The cost of physical I/O is the estimated number of physical I/Os that must be performed to retrieve all the rows that satisfy the predicates of the query. This estimate includes all physical I/Os to retrieve the requested data.

Consider the query:

```
SELECT ITEM_NUMBER, ITEM_NAME, RETAIL_PRICE
  FROM INVNTRY
  WHERE RETAIL_PRICE > 100
```

Suppose that there is an index on RETAIL_PRICE of the INVNTRY table. INVNTRY contains 10,000 rows, and each row is 100 bytes. Assuming a block size of 4 KB, INVNTRY has approximately 250 blocks. ITEM_NUMBER is the primary-key column and is 4 bytes, and RETAIL_PRICE is also 4 bytes. Therefore, the index row has a size of 10 bytes (key tag plus four plus the primary key size), and the index has about 25 blocks. Finally, suppose that 100 rows or 1 percent of the rows will satisfy the predicate.

If the query is to be evaluated using the primary key, 250 pages must be read from disk, and the cost of physical I/O would be 250. If the query is to be evaluated with the index, 102 pages must be read (two index pages + one data page for each qualifying index row), and the cost of physical I/O would be 102.

Cost of Record Overhead

The cost of record overhead is the processor time, expressed in terms of physical I/Os, associated with handling rows. This estimate includes the cost of setting up various control blocks and is dependent on the number of rows examined. The cost equals the following:

overhead per row × the number of rows to examine

For example, suppose that a processor can perform 2,000,000 instructions per second, and its disks can perform 30 I/Os per second. If 2,000 instructions are required before a row can be examined, the overhead per row would be approximately 0.03 I/Os (2,000 instructions would take one millisecond and is approximately the amount of time needed to perform 0.03 physical I/Os). If 10,000 rows must be examined, the cost of row overhead would be 300.

Cost of Messages

The cost of messages is the processor time, expressed in terms of physical I/Os, spent in sending messages between the file system and the disk process. This measurement is dependent on the type of SBB being used.

SQL computes the cost of messages as the following:

cost per message × number of messages

The cost per message is a weighing factor computed similarly to overhead per record.

The cost of messages is influenced by the number of columns projected and the number of rows selected for VSBB. If grouping or aggregation takes place in the disk process, a considerable reduction in messages occurs over aggregation that takes place in the executor.

Cost of Data Transfer

The cost of data transfer is the estimated elapsed time, expressed in terms of physical I/Os, for transferring data from the disk process (possibly remote) to the file system. In general, transfer cost is negligible for local transfers; it becomes substantial with remote transfers. This cost is computed as the following:

transfer rate × amount of data to be transferred

For example, suppose that 4,000 bytes are to be transferred from a remote node to the local node and that the two nodes are connected by one 4 kilobits per second communication line. Using a disk transfer rate of 30 I/Os per second, the cost of data transfer is 240.

Cost of Subqueries

The cost of a subquery is estimated as if it were a query in its own right. The cost equals the cost of the subquery, multiplied by the number of times it must be executed. It is computed as the cost of index’, where index’ is the index chosen to execute the subquery.

Cost also depends on whether a subquery is correlated or noncorrelated, as described in Section 1, Retrieving Data: How to Write Queries. These paragraphs discuss costs for both types of subqueries.

Correlated Subqueries

If a subquery cannot be executed independent from the outer query, the subquery is executed for every qualifying row of the outer query. This query selects information on items that cost more than the average price of the items produced by the same producer:

```
SELECT ITEM_NAME, RETAIL_PRICE
  FROM INVNTRY OUTER
  WHERE RETAIL_PRICE > SELECT AVG(RETAIL_PRICE)
    FROM INVNTRY
    WHERE  PRODUCER = OUTER.PRODUCER
```

The subquery in this example is dependent on the outer SELECT because it references the PRODUCER column of a row retrieved for the outer SELECT.

This correlation forces an evaluation of the subquery for every row retrieved from the outer SELECT. The overall query is more expensive to evaluate because of the repeated evaluation of the subquery. If the INVNTRY table contains 100 rows, the cost of evaluating the query is

```
COST(outer SELECT) + (100 x COST(inner SELECT) )
```

Noncorrelated Subqueries

A noncorrelated subquery can be evaluated before the outer query is executed. Because a noncorrelated subquery is only evaluated once, the cost of evaluating the original query is the sum of the cost of evaluating the individual SELECT statements. The practical limit for nesting is the amount of compile-time and run time resources (stack space, extended segment space).

The cost of a noncorrelated query is simply added to the cost of the overall plan.

Cost of Sorts

The cost of a sort is the estimated cost of sorting specified rows in a particular order. (The sort might be initiated by an ORDER BY, DISTINCT, UNION, or GROUP BY request or by the use of a sort merge join.) SQL supports three types of sorts: in-memory sort, a sort to a key-sequenced file, and the FastSort process.

The cost of a sort depends on the type of sort required. In general, cost items include:

- I/O cost to insert data into entry-sequenced file
- Scratch file cost
- Compare cost
- Temporary file creation
- Miscellaneous message and setup costs

For an in-memory user process sort, only the compare cost is necessary.

Sort costs for an ORDER BY are based on the number of rows going into the sort. If the ORDER BY is preceded by a hashed grouping operation, the number of rows for the sort is based on the data reduction expected by the grouping operation.

Parallel sort estimates calculate cost based on the number of rows, divided by the number of repartitions performing the sorting, including start costs for sort and ESP processes as well as interprocess messages and gains from parallel execution.

Cost of Join Operations

When estimating the cost of performing a join, SQL computes the cost of accessing each of the tables involved in the join. The cost of accessing each table is computed in the same way as in single-table queries, except that predicates must be identified as associated with a particular table. This approach is necessary because some join predicates cannot be evaluated until a qualifying row for an outer table has been retrieved.

Consider this query:

```
SELECT EMP_NAME, DEPT_NAME, SALARY
  FROM EMPLOYEE, DEPT
  WHERE EMPLOYEE.DEPT_NUM = DEPT.DEPT_NUM
    AND DEPT_NUM < 100
```

If the DEPT table is the outer table of the join, this predicate can be used to scan DEPT:

```
DEPT_NUM < 100
```

Thus, the predicate is involved in the computation of the cost of accessing DEPT.

The next predicate cannot be used in the cost computation for DEPT because no row has been retrieved for EMPLOYEE:

```
EMPLOYEE.DEPT_NUM = DEPT.DEPT_NUM
```

This predicate, however, could be used in the computation of the cost of accessing EMPLOYEE.

After the cost of accessing each table in the join has been determined, the cost of the join can be determined. For a nested join of two tables, the cost is

```
cost (a join b) = cost (a) + n x cost(b)
```

N is the number of rows that satisfy the nonjoin predicates on table A. (N is the number of times the inner loop must be performed.) For example, suppose that DEPT is the outer table of the join, EMPLOYEE is the inner table of the join, and 1,000 employees are in departments with department number less than 100. If the cost of accessing DEPT is 10 and the cost of accessing EMPLOYEE is 20, the cost of (DEPT join EMPLOYEE) is (10 + 1000 * 20), or 20,010.

The optimizer calculates separate plans for different access paths and join orders, evaluates whether to use index-only access, chooses a specific buffering option and method, if applicable, and takes partitioning into account.

The Effects of Indexes and Predicates on Costs

Because the complete row is not stored in an index, the cost of using an index is different than the cost of scanning the table. Predicates also play an important role in determining the cost associated with an index, because some predicates can be used as a begin-key or end-key for one index but not for other indexes.

This subsection describes the cost formulae when different indexes and predicates are available. SQL considers six different situations when computing the cost of using an index; these situations are listed in Table 5-3.

Table 5-3. Costs for Indexes With Predicates (page 1 of 2)

Type of Access	Predicates	Physical I/O Cost	Approximate Index Cost
Primary key	Equality predicates (column = value) specify all key columns; can use key positioning.	Number of index levels minus 1	The cost of physical I/O (assumes root of file is in cache)
Index	Equality predicates specify all key columns; can use keyed read on index followed by keyed read on base table.	Number of index levels of index minus 1, plus the number of index levels in the primary file minus 1	The cost of physical I/O
	-or- Equality predicates specify all key columns; can use keyed read on index followed by keyed read on base table, and all requested columns are in the index.	Same as above	Same as above

Table 5-3. Costs for Indexes With Predicates (page 2 of 2)

Type of Access	Predicates	Physical I/O Cost	Approximate Index Cost
Primary key	Equality predicates do not specify all key columns.	I/Os for blocks, (index selectivity × the number of nonempty blocks in the primary key file) × (the number of rows in the primary key file)	The cost of physical I/O plus the cost of record overhead
Index	Equality predicates do not specify all key columns, but all requested columns are in the index.	Same as above	Same as above
Index	Equality predicates do not specify all key columns, and a physical I/O is required for each row in the index.	I/Os for blocks, (index selectivity × number of nonempty blocks in the primary key file) × (the number of rows in the primary key file), + a physical I/O required for each qualifying row in the index.	The cost of physical I/O plus the cost of record overhead

The Effect of the MultiDimensional Access Method (MDAM) on Costs

SQL estimates costs for MDAM according to Table 5-3, but the cost estimation is divided into two parts: reading the data for a single access and counting the total number of accesses. The cost for reading the data is multiplied by the number of accesses. The result is the cost for a single predicate set. For information about predicate sets, see Transformation of Key Column Predicates and Predicate Sets on page 3-4.

Estimating MDAM Costs for a Single Predicate Set

These factors are considered when estimating costs for a single predicate set:

- The number of unique values within a predicate set
- Whether the data is sparse or dense
- Whether a column is the last-used column in a key

The Number of Unique Values Within a Predicate Set

A positioning is done for each unique value within the predicate set. For example, in a range, SQL does a positioning for each unique value within the range. For the range B

`BETWEEN 5 AND 10,` a positioning takes place for each value of B between 5 and 10. The number of positionings that take place for a predicate set are totaled as part of the cost.

A positioning does not take place if a range is for the last-used key column.

To estimate the cost of an IN predicate within a predicate set, the optimizer converts it to an IN predicate equivalent. A positioning is done on each of the equal predicates and the number of positionings is added to the cost.

The Sparsity or Density of the Data

The optimizer considers the sparsity or density of data whenever a range is present. If the data in the predicate set is sparse, two positionings are done. The first positioning finds the next value for a column. The second one accesses the data using the values for all of the other columns. Based on the statistics, the optimizer estimates the number of positionings that the executor will do.

If the data is dense, SQL positions only once. The number of key columns and the predicate selectivity are also considered.

A dense range has the same cost as an IN predicate with the same values, as in:

```
B BETWEEN 5 and 10
```

would be costed the same as

```
IN (5, 6, 7, 8, 9, 10)
```

The Last-Used Column in a Key

When the last-used key column is a range, the executor reads the entire range as a unit, and no key positionings are required for each possible value. The cost of reading a range on the last key column is included in the cost of an access.

Estimating MDAM Costs for Multiple Predicate Sets

For multiple predicate sets, the cost for each predicate set is completed separately, and the costs are accumulated. For joins, the cost of each predicate set is multiplied by the number of rows used from the outer table.

Evaluating Cost Estimates

When examining cost, these guidelines apply:

- High cost indicates that the given query appears to be (and probably is) expensive.

  Always review high cost statements. Try to estimate how much I/O such a query should take and if it is consistent with that reported by EXPLAIN. If your estimate and that of EXPLAIN vary considerably, carefully review the EXPLAIN plan to determine why the optimizer estimated that the query is so expensive. For example, the data might not be distributed uniformly.

  When up-to-date statistics are available, a high Total Cost might indicate that you are reading too many rows. For a given database with good production level statistics, you can plot the total cost of a statement with results from SQLSTMT (see Using Measure on page 6-7) and look at the relationship between Total Cost, rows accessed, and perceived response time (as experienced by the end user).
- Although low cost is generally desirable, low cost does not necessarily indicate low overhead. Instead, a low cost might indicate that the optimizer does not have enough information to estimate the cost accurately. Catalog statistics might be nonexistent or might not represent the production environment accurately.
- The cost estimate might be inaccurate if data has an uneven distribution instead of being distributed uniformly across blocks and partitions.
- Beware of plans built for small tables. If your development or test tables are small, the cost might be very different when the same plan is used on large production tables. Whenever possible, evaluate EXPLAIN output as it would appear in production during the query development and program test phases of development.

For information about how to access cost estimates, see °Section 6, Analyzing Query Performance.

How the Optimizer Chooses an Execution Plan

The optimizer selects the most efficient execution plan, defined as the one that takes the least time to complete the evaluation of a query.

The estimated costs associated with several query plans might, however, be very close; that is, within 10 percent of one another. To select between two plans whose costs are close, the optimizer chooses a plan based on these priorities, listed in order of preference:

- A local table or index (as opposed to a remote table or index)
- A table or index in which predicates of the form “column=value” have specified all the key columns forming a unique access
- A table or index with a lower selectivity

- A table or index with a lower estimated cost

In general, the optimizer attempts to choose a local table or index that has the least number of qualifying rows that must be examined.

Forcing Execution Plans

The goal of the optimizer is to generate a plan that works well on the average. Because of variations in applications and data, however, SQL sometimes chooses a plan that is not optimal. In such cases, you can specify a CONTROL TABLE option that forces the optimizer to choose these (listed with examples showing when you might use each option):

- Access path for a table (ACCESS PATH), if CONTROL TABLE INTERACTIVE ACCESS ON did not cause the optimizer to choose a specific index
- Join sequence when processing a query (JOIN SEQUENCE), if ORDER BY or other specifications did not influence the join sequence
- Join methods when processing a query (JOIN METHOD), if the Measure product has been run against different executions of the query and consistently indicates that a certain join method would perform better than the one chosen by the optimizer

Note that while CONTROL QUERY and CONTROL EXECUTOR directives recommend actions, the CONTROL TABLE directive is treated as a specific request.

△ **Caution.** These CONTROL TABLE options override the optimizer's standard cost estimates and therefore might cause performance degradation instead of enhancement. Use of these options requires a thorough understanding of the optimizer. Use these options *only* if the optimizer does not produce the optimal plan.

Situations that might benefit from these CONTROL TABLE options include:

- Nonuniform data distribution; the optimizer operates as if distinct values are uniformly distributed over ranges of values
- Predicates that are not independent of one another; the optimizer operates as if predicates are independent.
- Selectivity estimates, being probabilistic, do not reflect the actual runtime environment.

If you suspect that you might benefit from the use of one of these options, check your application with and without the CONTROL option, using actual Measure statistics from production data.

If you use one of the options, you might want to change this directive 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 any occurrences of it easy to find and change, using one or more of these alternatives:

- Make sure the directive only applies to the statement and table intended. Return the specified table to SYSTEM method directly after the statement.
- Isolate this directive in its own section and perform it from the inline application code.
- Place all statements affected by this directive in separate modules, called as services by other modules.

Verify the use of any of these options periodically to account for changes in data distributions and volumes. Reevaluate their effectiveness with each new version of SQL.

6 Analyzing Query Performance

Different queries have varying levels of impact on your system. One way to estimate query use is with the 90-10 rule, which estimates that 10 percent of the queries use up 90 percent of critical resources. The 90-10 rule can help you determine which queries are most important from a performance viewpoint.

Note, however, that performance evaluation at the statement level should be done on a system that is as well-tuned as possible. For more information on system tuning and performance, see the *SQL/MP Installation and Management Guide*.

This section discusses these topics:

Guidelines for Tuning Queries

When examining and tuning queries, use available tools such as these:

- The DISPLAY STATISTICS command, which displays run time statistics about the last DML command you executed.
- The Measure product, which collects statistical information about SQL database objects and processes, and generates reports about them.
- The EXPLAIN utility, which provides detailed information about the optimizer-generated query execution plan for a compiled query.

The rules for tuning SQL statements can be summarized as follows:

- For new queries, prototype, prepare, and test the statements. Optimally, use EXPLAIN and run the statements against production data before incorporating the query into a program.
- For queries already in use, obtain Measure information and categorize your programs based on:
  - High consumption of system resources
  - Poor performance and critical priority
  - High volume queries

Focus ongoing performance work on these areas:

- High-impact queries (these might be most-used, highest system resources, longest response times, or most critical queries)
- Queries being migrated to production
- Review plans after an UPDATE STATISTICS operation

Preparing Your Queries

Before you test your queries, you should prepare them using the SQLCI PREPARE command.

The PREPARE command compiles an SQL statement and assigns a name to the statement. You can then reference the statement name to execute the statement multiple times without recompiling, and you can obtain an EXPLAIN plan for the compiled statement. These steps outline this procedure:

1. Prepare the query:

   ```
   >> PREPARE QUERY FROM
   +>   SELECT * FROM EMPLOYEE ;
   ```

2. Obtain EXPLAIN plan for compiled query:

   ```
   >> EXPLAIN PLAN FOR QUERY ;
   ```

3. Execute the query:

   ```
   >> EXECUTE QUERY ;
   ```

4. Display execution statistics for the query:

   ```
   >> DISPLAY STATISTICS ;
   ```

Details on analyzing the EXPLAIN plan and execution statistics follow.

Using DISPLAY STATISTICS

The DISPLAY STATISTICS command provides statistics for an executed query. You do not need to prepare a statement to use DISPLAY STATISTICS. However, if you want to execute the statement more than once by referring to the statement name, you should first prepare the statement. You can also refer to the statement name to generate an EXPLAIN plan.

These queries retrieve data from the EMPLOYEE table of the sample database provided with the SQL software. The primary key of the EMPLOYEE table is EMPNUM.

Simple Query Example

Example 6-1 shows a simple query that selects all rows and columns from the EMPLOYEE table:

Example 6-1. Simple Query

```
23> SQLCI
>> PREPARE Q1 FROM
+>  SELECT * FROM EMPLOYEE ;
--- SQL command prepared.
>> EXECUTE Q1 ;
The query returns the following result:
EMPNUM FIRST_NAME      LAST_NAME      DEPTNUM JOBCODE SALARY
------ --------------- -------------- ------- ------- ----------
     1 ROGER           GREEN             9000     100  175500.00
         .                .                .        .       .
   568 JESSICA         CRINER            3500     300   39500.00
--- 57 row(s) selected.
```

To obtain statistics about the query, simply enter DISPLAY STATISTICS after the query executes:

```
>>DISPLAY STATISTICS ;
```

If you want SQL to display the statistics automatically, you can enter SET STATISTICS ON at any point during the session. From this point on, the statistics will appear immediately after each command executes.

Example 6-2. DISPLAY STATISTICS for Simple Query

```
Estimated Cost            2

Start Time            95/09/11 09:06:07.864928
End Time              95/09/11 09:06:08.196387
Elapsed Time                   00:00:00.331459
SQL Execution Time             00:00:00.112590

                        Records     Records        Disk     Message    Message
Lock
Table Name             Accessed        Used       Reads       Count      Bytes
\SQL1.$DATA8.PERSNL.EMPLOYEE
                             57          57           0           3       2916
```

The statistics in Example 6-2 indicate:

- The estimated cost of the query

  The number 2 represents a relative measure derived using the same cost functions that the optimizer uses to choose an execution plan. The estimate includes processor, disk I/O, and message costs.

- The start time and the end time
- The SQL elapsed time and execution time:

    - Elapsed time includes the execution time, I/O time, and the time to display the result.
    - Execution time is the amount of processor time used by the executor.

- The number of records accessed and the number of records used
    - Records accessed is the number of rows read (including rows that do not satisfy the selection criteria).
    - The rows are counted for each table, underlying table of a protection view, and temporary table. If you join a table to itself, separate statistics are reported for each instance of the table. The number of rows accessed in an index is not reported.
    - This number does not indicate the specific number of physical disk reads or writes, because the system uses disk caching to reduce the number of physical read and write operations.
    - Records used is the number of rows returned to the executor by the disk process. If the query includes a join operation, the number of rows returned might be smaller than the actual number of rows retrieved. The row count for a delete operation depends on the type of delete operation; a cursor delete returns a different count than a set delete because of differences in the ways the types of delete operations are performed.
    - In Example 6-1 on page 6-4, all rows must be returned to satisfy the query (SELECT * with no WHERE clause), so the number of rows accessed and used is the same.
- The number of disk reads. In Example 6-1 on page 6-4, the data is found in the disk cache, so there is no need to access the disk.
- The message count. Message count is usually equal to the number of blocks passed from the disk process to the file system. Sometimes an additional message is necessary to ensure that the last row was processed. In this example, there were two disk blocks passed, plus one final message.
- The message lock bytes.

Simple Query With ORDER BY Example

Example 6-3 selects specific columns and orders them by salary:

Example 6-3. Simple Query With ORDER BY

```
>>PREPARE Q2 FROM
+> SELECT LAST_NAME, FIRST_NAME, SALARY
+>   FROM EMPLOYEE
+>  ORDER BY SALARY ;
--- SQL command prepared.
>>
>>EXECUTE Q2 ;

LAST_NAME             FIRST_NAME       SALARY
--------------------  ---------------  -----------

DAY                   KATHRYN             12000.00
CHAPMAN               SUSAN               17000.00
   .                     .                   .
GREEN                 ROGER              175500.00

--- 57 row(s) selected.
>>
>>DISPLAY STATISTICS ;
```

Example 6-4. DISPLAY STATISTICS for Simple Query With ORDER BY

```
Estimated Cost              3

Start Time            95/09/11 09:06:11.863448
End Time              95/09/11 09:06:12.175160
Elapsed Time                   00:00:00.311712
SQL Execution Time             00:00:00.124730

                         Records     Records       Disk    Message   Message Lock
Table Name              Accessed        Used      Reads      Count          Bytes
\SQL1.$DATA8.PERSNL.EMPLOYEE
                              57          57          0          4           2962
```

Notice that the estimated cost of the simple query in Example 6-1 on page 6-4 is 2, and that of the simple query with ORDER BY in Example 6-3 on page 6-6 is 3. This is because the query in Example 6-3 on page 6-6 requires a sort operation for the ORDER BY SALARY request.

Using Measure

You can use the Measure product to gather statistics on an SQL database and application programs. This subsection briefly describes the kinds of statistics that Measure provides. For more information on using Measure, see the *SQL/MP Installation and Management Guide*.

The Measure product provides statistics on process execution and on individual statement execution.

Process Execution

For each process, the Measure product provides these statistics:

- The number of times the entire SQL program was recompiled and the elapsed time needed for the recompilation
- The number of times static SQL statements were recompiled and the elapsed time needed for recompilation
- The number of times the SQL compiler and SQL catalog manager 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

The SQLSTMT report provides information for specific statements of 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 altered
- The number of I/O operations
- 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 name for each statement.

In the report, a section name is identified by the procedure name and index #*nn*, which relates to the SQL Section Paragraph number generated during the host language compilation. An SQL section is generated for each SQL statement and is listed in the compilation output following the program code. The exception is for the statements on cursors: OPEN, FETCH, and CLOSE cursor statements. The counters of the OPEN, FETCH, and CLOSE cursor statements all contribute to the counter of the DECLARE CURSOR section number.

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.

Optimally, measure each transaction or query in isolation; otherwise, you will 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 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 EXPLAIN plans for the queries; the plans should 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, then 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.

Using EXPLAIN

An EXPLAIN report describes the execution plan for a DML statement. Each plan is divided into one or more steps: one for a scan of each table in a FROM clause and one for each union operator in a query. Each step can involve one or more of these:

- Scan of a table
- Join of two or more tables
- UNION operation
- Insert into a table
- Sort operations
- Parallel execution
- Sequential buffering

If a query has subqueries, the report shows additional steps in the same way for each subquery.

You can use the information in the EXPLAIN plan for these types of tasks:

- Assisting application program design; for example, the plan can help you
  - Determine the access path to be chosen.
  - Identify problems causing long response times, such as large sorts, full table scans, and correlated subqueries.
- Assisting database design; for example, you can use it to tune queries and to help determine whether to add or drop indexes for a database.
- Determining whether the optimizer chooses the expected plan and whether there are ways to improve query performance. For example, you might decide that creating an additional index can improve the performance of the query.

The EXPLAIN report is based on current information at the time you generate the report. If access paths or statistics change before you execute a query for which you obtained an EXPLAIN report, SQL might use a different execution plan. If, for example, you drop an index, the execution plan will be different than the plan that existed before you dropped the index.

Reports generated through the SQLCOMP compiler option include a section that lists each DEFINE used in an SQL statement. For more information on the compiler option, see the *SQL/MP Programming Manual* for your host language. For detailed information about each element in the report, see the *SQL/MP Reference Manual*.

Note. This manual supports NonStop SQL/MP D30.02 and D30.03. Cost estimates reported by the EXPLAIN utility are not considered to be comparable with cost estimates from previous versions; the cost for the same query might differ from release to release.

Generating an EXPLAIN Plan

You can invoke EXPLAIN in one of two ways:

- As an SQLCI command
- As an option of the SQL compiler

This subsection describes how to invoke EXPLAIN as an SQLCI command. The general syntax is:

```
EXPLAIN [ PLAN FOR ] { statement      }
                     { statement-name }
```

PLAN FOR is an optional clause that does not affect output. `statement` is an SQL DML statement, by itself or enclosed in single or double quotation marks. `statement-name` is the name of a prepared SQL statement.

This example prepares the statement first and then specifies EXPLAIN:

```
>> PREPARE Q2 FROM
+> SELECT LAST_NAME, FIRST_NAME, SALARY
+>  FROM EMPLOYEE
+>  ORDER BY SALARY ;
--- SQL command prepared.

>> EXPLAIN PLAN FOR Q2 ;
```

You can also specify EXPLAIN as part of the statement syntax:

```
>> EXPLAIN
+>   SELECT LAST_NAME, FIRST_NAME, SALARY
+>   FROM EMPLOYEE
+>   ORDER BY SALARY ;
```

A plan provides different types of information for different types of operations. For instance, Example 6-5 lists the EXPLAIN plan for the preceding query, and shows information about one scan operation and one sort operation.

Example 6-5. EXPLAIN Plan for Simple Query With ORDER BY

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 57 rows after the scan
  Operation cost    : 2

Operation 1.1  : Sort
  Requested         : Explicitly in the query
  Sort rows in the  : Result of a Select
  Purpose           : To order rows for an Order By
  Sort technique    : FASTSORT
  Sort type         : Plan to use User Process Sort
  UPS workspace     : 24 Kbytes
  Sort key columns  : EMPLOYEE.SALARY asc
  Sort cost         : 1

Total cost        : 3
```

The total cost represents the cost of doing all the operations to complete the statement. In Example 6-4 on page 6-6, the DISPLAY STATISTICS command shows the estimated cost for this query is 3. By using EXPLAIN, you can see that the sort cost for this query is 1. By removing the ORDER BY clause (which causes the sort request), you can reduce the total cost of the query to 2, as shown in Example 6-6.

```
>>PREPARE Q3 FROM
+> SELECT LAST_NAME, FIRST_NAME, SALARY
+> FROM EMPLOYEE  ;
--- SQL command prepared.
>> EXPLAIN PLAN FOR Q3 ;
```

Example 6-6. EXPLAIN Plan for Simple Query Without ORDER BY

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

--------------------------------------------------------------------
Plan step 1
--------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 57 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

These examples show the cost determined for various operations, such as sorts. (If you need rows returned in a specific order, however, and the sort cost is a problem, you can create an index on the sorted column.)

Detailed analyses of several EXPLAIN plans appear later in this section.

Interpreting an EXPLAIN Plan

When you examine the EXPLAIN plan, look at all factors and information about the query. Use this information to assist you in tailoring queries so that they are efficient and return the information you require.

You should examine the EXPLAIN plan for the following:

- Total cost
- Table scans
- Access path
- Sort operations
- Correlated subqueries
- Key predicates and MDAM predicate sets
- Executor predicates
- Sequential block buffering
- Sequential cache
- Selectivity
- Locking strategy
- Parallel execution

Total Cost

EXPLAIN assigns a total cost to scans and sorts, which represents the cost of doing all the operations to complete the statement.

Cost is an estimate and not an exact measure. Many variables that the optimizer cannot control can affect the actual run time execution. Cost, however, can be useful for comparing the different execution plans for a given query.

Note that the total cost is not the sum of the costs of each step. For example, nested join steps usually have costs that are multiplied instead of added. Also, sort costs are not linear with the number of input rows.

For OLTP requests, watch for costs that are high. In some production application programs, costs in the millions are possible. The maximum displayable value is equal to the size of an SQL LARGEINT value—a maximum value of (2**63) –1. Any amount greater than this cannot be displayed. In some cases, reformulating queries with very high costs can reduce execution time considerably. For fast OLTP response (seconds), the total cost should be low.

For more information about cost, see Evaluating Cost Estimates on page 5-19.

The following subsections describe factors that influence cost.

Table Scans

Full table scans can affect performance adversely—especially if the table is quite large. To check for costly table scans, look for:

- Index selectivity of 100 percent
- No begin-key predicates, end-key predicates, or MDAM predicates

Access Path

Access can be by primary key or alternate index. Remember that an alternate index consists of all the columns defined for the index plus the column (or columns) that make up the primary key.

For example, the CONTROL QUERY INTERACTIVE ACCESS ON directive forces the optimizer to consider index access. The directive indicates to the optimizer that only a few rows are needed by the query and not the complete qualifying set. If the WHERE clause contains columns that are part of an index prefix, the index will be used despite selectivity and other considerations. If the EXPLAIN listing seems to indicate an otherwise illogical choice, you should look for this directive.

For more information about access path, see Optimizing the Access Path on page 4-4.

Sort Operations

Always check for sorts and high sort costs.

Sorts can occur as a result of ordering requests (specifying an ORDER BY, GROUP BY, or DISTINCT clause) or as a result of unanticipated join ordering requirements. For more information, see Minimizing Sort Costs for Ordering and Grouping Operations on page 3-54.

Correlated Subqueries

Correlated subqueries often impact performance adversely.

For queries containing subqueries, review the EXPLAIN plan for correlated subquery characteristics. If the plan gives this message about the subquery, a correlated subquery is present:

```
Executes once per row retrieved by operation
```

For more information, see Optimizing Subqueries on page 3-51.

Key Predicates and MDAM Predicate Sets

The EXPLAIN plan lists all begin-key and end-key predicates. Key predicates narrow the range of searching. Such conditions reduce the number of rows processed by the disk process; therefore, the query executes faster.

If the EXPLAIN plan indicates there are no key predicates, you might want to review the query and consider adding search conditions, based on leftmost key columns, that restrict the number of rows accessed, if feasible. For more information, see Writing Efficient Predicates on page 3-15.

A lack of key predicates can also be a factor in causing a full table scan.

When the optimizer uses MDAM, the EXPLAIN plan shows the result of converting the key-column predicates to MDAM predicate sets. These predicate sets show how the optimizer processes the predicates.

Executor Predicates

Check the EXPLAIN plan for evaluation of predicates at the executor level.

The most efficient predicate evaluation is at the disk process level. Executor evaluation indicates a bigger impact on performance. For more information on which predicates cause evaluation at the executor level, see How the Optimizer Processes Predicates on page 3-4.

Sequential Block Buffering (SBB)

If you specify sequential access by using the CONTROL TABLE SEQUENTIAL directive, the optimizer uses virtual sequential block buffering (VSBB) and prefetch techniques, if feasible.

In some cases, the preferred SBB might not be selected if the criteria required for SBB are not met or if you have disabled sequential buffering (by using the CONTROL TABLE directive). For more information, see Reducing Messages With Buffering Options on page 4-21.

When you expect SQL to return only a few rows, you may want to turn off SBB to eliminate unnecessary processing of additional rows. When opening a cursor, fetching one row, and closing the cursor, use a CONTROL TABLE SEQUENTIAL READS OFF statement. This statement will keep SQL from processing additional rows to fill a VSBB buffer.

Sequential Cache

In the description of the access path, the EXPLAIN plan states whether access is sequential and whether data is kept in cache memory, or sequential cache, for only a short time.

With sequential cache, as soon as all the rows in a block have been used, the block is discarded twice as quickly as it would have been if sequential cache were not used.

Sequential cache is ignored if the threshold for sequential prefetch is not reached.

For more information about the cache buffer, see the *SQL/MP Installation and Management Guide*.

Selectivity

The EXPLAIN plan lists selectivities for tables and indexes. Selectivity values influence the optimizer’s choice of the following:

- Access path (base table, alternate index, or index only)

  For example, if the restriction specified by a WHERE predicate does not result in a low enough selectivity to justify alternate-index access, base-table access is chosen instead. In such a case, the index plus base table access could be worse than a base table scan with sequential prefetch.

- Join order

  The selectivity of each table determines the optimizer’s choice of the outer and inner table.

- Type of sorts performed

Check for table or index selectivities of 100 percent, which indicate costly table scans.

For more information about selectivity, see Section 5, Selectivity and Cost Estimates.

Locking Strategy

The EXPLAIN plan indicates the following:

- Granularity of lock (row, partition, table).
- Access option (browse, stable, repeatable).
- Whether the lock mode (exclusive or shared) is chosen by the system or by the user.
  - If the lock mode is chosen by the system, the plan indicates “chosen by the system,” but does not show which mode was chosen.
  - If the lock mode is specified by the user, the plan simply states “Share” or “Exclusive.”

Check for unexpected lock escalation and check the access option. Is the system choosing stable access (the default) when browse access is sufficient? Browse access enables you to read data currently being updated or deleted. If potentially inconsistent data is unacceptable, do not specify browse access.

Parallel Execution

EXPLAIN indicates at the beginning of the plan whether parallel execution is used and for which operations.

Parallel execution is especially useful when a large number of rows needs to be processed by the executor, but only a small number of rows needs to be returned to satisfy the query. (Section 4, Improving Query Performance With Environmental

Options, provides a more thorough description of when parallel execution is and is not chosen.)

If you are expecting parallel execution and it is not chosen, you might not have enabled parallel execution. You must enable parallel execution by specifying CONTROL EXECUTOR PARALLEL EXECUTION ON before SQL attempts to process a query or parts of a query in parallel.

Reviewing Sample EXPLAIN Plans

The following subsections show several examples of EXPLAIN plans. The descriptions of the first two examples include a detailed analysis of the entire EXPLAIN plan.

To avoid needless repetition, the remaining examples, describe only specific aspects of each EXPLAIN plan—details that were not seen in previous plans. For example, the descriptions of the EXPLAIN plans for subqueries emphasize only those details related to subquery evaluation.

Scan through all the examples to get a thorough knowledge of the type of information you can discern from an EXPLAIN plan. Also, see the *SQL/MP Reference Manual*, which provides an alphabetic description of each element of an EXPLAIN plan.

When scanning the following examples, keep in mind that an EXPLAIN plan can consist of several plan steps. The steps describe such things as scans of tables, UNION operations, and so on. Plan steps are not necessarily executed according to how they are numbered. For example, if the query is a noncorrelated subquery, then plan step 2, the evaluation of the subquery, evaluates once before plan step 1, the evaluation of the outer query.

Each plan step might consist of one or more operations, such as scans or sorts. Each operation is assigned a separate cost. Each query is assigned a total cost.

EXPLAIN Plan for Simple SELECT

This example shows an EXPLAIN plan for a simple SELECT statement on a single table.

The query retrieves all rows from the EMPLOYEE table. The primary key is EMPNUM. The query follows:

```
EXPLAIN SELECT * FROM EMPLOYEE ;
```

The total cost of the query is 2.

Example 6-7 on page 6-17 shows the EXPLAIN plan for the query. The plan consists of one step with one operation. A detailed analysis follows.

Example 6-7. EXPLAIN Plan for Simple SELECT

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 57 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

Plan step 1 involves a scan of the EMPLOYEE table. The SELECT `*` operation requires that all columns (6 out of 6) be retrieved from the table. The EXPLAIN plan includes this information:

- The lock granularity, shown in the access type, is row (record). The access option is stable (the default).
- The lock mode is chosen by the system. (If the system chooses the lock mode, the EXPLAIN plan does not specify which type. The plan simply states "Chosen by the system.")
- The access path is by primary key.
- Virtual sequential block buffering (VSBB) is used to read the table.
- Index selectivity is 100 percent, which indicates that the optimizer estimates that the entire table will be read.
- Table selectivity is 100 percent, which indicates that all rows are selected from the table.

Because the index and table selectivity are 100 percent, SQL performs a full table scan with all rows returned. The table is small, however, so the total cost of the query is small.

EXPLAIN Plan for Primary Access

Example 6-8 and Example 6-9 show different EXPLAIN plans for the same query: one plan uses primary access; the other uses index-only access.

The query selects all rows from the EMPLOYEE table (primary key EMPNUM) and orders the rows by LAST_NAME, FIRST_NAME:

```
EXPLAIN PLAN FOR
  SELECT EMPNUM, FIRST_NAME, LAST_NAME
  FROM EMPLOYEE
  ORDER BY LAST_NAME, FIRST_NAME ;
```

Example 6-8 shows the EXPLAIN plan for the query before creating an index on the LAST_NAME, FIRST_NAME columns.

SQL must sort the rows to satisfy the ORDER BY clause. A detailed analysis of the plan follows.

Example 6-8. EXPLAIN Plan Choosing Primary Access

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 57 rows after the scan
  Operation cost    : 2

Operation 1.1  : Sort
  Requested         : Explicitly in the query
  Sort rows in the  : Result of a Select
  Purpose           : To order rows for an Order By
  Sort technique    : FASTSORT
  Sort type         : Plan to use User Process Sort
  UPS workspace     : 24 Kbytes
  Sort key columns  : EMPLOYEE.LAST_NAME asc, EMPLOYEE.FIRST_NAME asc
  Sort cost         : 1

Total cost        : 3
```

Plan step 1 involves two operations: a scan of the EMPLOYEE table and a sort operation.

Operation 1.0 is a scan of the EMPLOYEE table. The query requires that 3 out of 6 columns be retrieved from the table. The EXPLAIN plan for this operation includes the following:

- The lock granularity is row (record). The access type is stable (the default).
- The lock mode is chosen by the system.
- The access path is the primary key.
- Virtual sequential block buffering (VSBB) is used to read the table.
- Index selectivity is 100 percent. This indicates that the optimizer expects that the entire table will be read.
- Table selectivity is 100 percent. All rows are selected from the table.
- The cost of operation 1.0 is 2.

Operation 1.1 is a sort operation requested explicitly in the query as a result of the ORDER BY clause. The purpose of the sort is to sort rows in a specific order (LAST_NAME, FIRST_NAME). The EXPLAIN plan for this operation includes the following:

- The sort technique is FASTSORT.
- The columns are sorted in ascending order.
- The sort cost is 1.

The total cost of the query is 3.

EXPLAIN Plan for Index-Only Access

This query selects all rows from the EMPLOYEE table (primary key EMPNUM) and orders the rows by LAST_NAME, FIRST_NAME:

```
EXPLAIN PLAN FOR
  SELECT EMPNUM, FIRST_NAME, LAST_NAME
  FROM EMPLOYEE
  ORDER BY LAST_NAME, FIRST_NAME ;
```

Example 6-9 on page 6-20 shows the EXPLAIN plan for the query after creating an index that will satisfy the ORDER BY clause:

```
CREATE INDEX XEMPNAME
  ON EMPLOYEE
  (LAST_NAME, FIRST_NAME) ;
```

Example 6-9. EXPLAIN Plan Choosing Index-Only Access

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XEMPNAME, index only
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 57 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

The optimizer chooses index-only access rather than access to the base table through the primary key:

```
Access path 1     : Alternate\SQL1.$DATA8.PERSNL.XEMPNAME,
                    index only
```

Because the index is in the same key sequence as the ORDER BY request (LAST_NAME, FIRST_NAME), a sort is not required before returning the rows in the requested order. Note that the total cost of the query is now 2 (compared to 3 for the plan choosing primary access to the table).

EXPLAIN Plans for Bounded Predicates

Example 6-10 on page 6-21 and Example 6-11 on page 6-22 show EXPLAIN plans for two queries that specify bounds in the search condition. Both queries retrieve data from the EMPLOYEE table (primary key EMPNUM).

Query With Lower Bound

The query for the EXPLAIN plan in Example 6-10 on page 6-21 specifies a predicate with a lower bound (WHERE SALARY >= 50000):

```
EXPLAIN
  SELECT LAST_NAME, FIRST_NAME, SALARY
  FROM EMPLOYEE
  WHERE SALARY >= 50000 ;
```

The total cost of the query is 2.

Example 6-10. EXPLAIN Plan for Lower-Bound Predicate

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        SALARY >= 50000
    Pred. selectivity : Expect to select 72.3141% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 72.3141% of rows from table
  Expected row count: 41 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

The base table predicate is evaluated by the disk process:

```
Base table pred.  : Will be evaluated by the disk process
                    SALARY >= 50000
```

Predicate and table selectivity are both approximately 72 percent, which indicates that approximately 72 percent of the rows in the table satisfy the predicate. Specifically, for a range predicate:

```
                   SECONDHIGHVALUE - 50000
selectivity = --------------------------------
              SECONDHIGHVALUE - SECONDLOWVALUE
```

(Because there is only one predicate, predicate and table selectivity are the same.)

Query With Lower and Upper Bound

The query for the EXPLAIN plan in Example 6-11 specifies both a lower bound (>=50,000) and an upper bound (<=100,000):

```
EXPLAIN
  SELECT LAST_NAME, FIRST_NAME, SALARY
  FROM EMPLOYEE
  WHERE SALARY >= 50000
  AND SALARY <= 100000 ;
```

The total cost of the query is 2.

Example 6-11. EXPLAIN Plan for Lower and Upper Bounded Predicates

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        ( SALARY >= 50000 ) AND ( SALARY <= 100000 )
    Pred. selectivity : Expect to select 38.5048% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 38.5048% of rows from table
  Expected row count: 22 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

Note that predicate and table selectivity are both approximately 38.5 percent. For the range predicates, selectivity is calculated as follows:

```
                     100000 - 50000
selectivity = --------------------------------
                SECONDHIGHVALUE - SECONDLOWVALUE
```

EXPLAIN Plan for Key Predicates

The EXPLAIN plan shows a query that specifies both a begin-key and an end-key predicate.

The query retrieves data from the PARTS table. PARTNUM is the primary key. The query follows:

```
EXPLAIN
  SELECT PARTNUM, PRICE, QTY_AVAILABLE
  FROM PARTS
  WHERE PARTNUM BETWEEN 5000 AND 7000 ;
```

The total cost of the query is 1.

Example 6-12. EXPLAIN Plan for Key Predicates

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.SALES.PARTS
                      with correlation name PARTS
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 4 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : PARTNUM >= 5000
    End key pred.     : PARTNUM <= 7000
    Index selectivity : Expect to examine 26.0514% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 26.0514% of rows from table
  Expected row count: 7 rows after the scan
  Operation cost    : 1

Total cost        : 1
```

The plan includes this information:

- The BETWEEN operator is converted to a range predicate, “PARTNUM >= 5000 AND PARTNUM <= 7000.”
- The WHERE predicate is specified against PARTNUM, the primary key. The predicate specifies a range of values, so there is both a begin-key and an end-key predicate:

```
Begin key pred.   : PARTNUM >= 5000
End key pred.     : PARTNUM <= 7000
```

- Index and table selectivity are both approximately 26 percent. For the range predicates, selectivity is calculated as follows:

```
                     7000 - 5000
selectivity = --------------------------------
                SECONDHIGHVALUE - SECONDLOWVALUE
```

- The selectivity is significantly lowered because of the addition of the upper bound; that is, the optimizer expects the number of rows returned will be smaller.

EXPLAIN Plan for DISTINCT

This query uses DISTINCT to remove duplicate rows from the result:

```
SELECT DISTINCT PARTNUM, PRICE
   FROM PARTS
   WHERE PARTNUM < 7000 AND PARTNUM > 5000 ;
```

The total cost of the query is 1.

Example 6-13. EXPLAIN Plan for SELECT DISTINCT

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.SALES.PARTS
                      with correlation name PARTS
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 4 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : PARTNUM > 5000
    End key pred.     : PARTNUM < 7000
    Index selectivity : Expect to examine 26.0254% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 26.0254% of rows from table
  Expected row count: 7 rows after the scan
  Operation cost    : 1

Total cost        : 1
```

The plan contains one step, which scans the PARTS table. Access is by primary key.

EXPLAIN Plan for ORDER BY

This query requires a sort operation to present columns in a specific order:

```
EXPLAIN
 SELECT PARTNUM,PRICE,QTY_AVAILABLE
   FROM PARTS
   ORDER BY 2 DESC, 3 ASC, 1 ;
```

The total cost of the query is 3.

Example 6-14. EXPLAIN Plan for SELECT With ORDER BY

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.SALES.PARTS
                      with correlation name PARTS
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 4 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 28 rows after the scan
  Operation cost    : 2

Operation 1.1  : Sort
  Requested         : Explicitly in the query
  Sort rows in the  : Result of a Select
  Purpose           : To order rows for an Order By
  Sort technique    : FASTSORT
  Sort type         : Plan to use User Process Sort
  UPS workspace     : 24 Kbytes
  Sort key columns  : PARTS.PRICE desc, PARTS.QTY_AVAILABLE asc,
                      PARTS.PARTNUM asc
  Sort cost         : 1

Total cost        : 3
```

The plan consists of one step involving two operations: a scan of the PARTS table and a sort operation.

The sort operation is requested explicitly in the query as a result of the ORDER BY clause. The purpose of the sort is to order columns in a specific order. The sort technique is FastSort.

The columns are sorted in the following order:

- The PRICE column in descending order
- The QTY_AVAILABLE column in ascending order
- The PARTNUM column in ascending order

EXPLAIN Plans for GROUP BY

These examples show EXPLAIN plans that use GROUP BY to group data.

SELECT With GROUP BY Using a Serial Plan

This query requires a hash operation to group rows for an aggregate function:

```
CONTROL EXECUTOR PARALLEL EXECUTION OFF;

EXPLAIN
 SELECT DEPTNUM, JOBCODE, COUNT(*) FROM EMPLOYEE
  GROUP BY 1,2;
```

The plan, which consists of one step, is performed serially. The total cost of the query is 3.

Example 6-15. EXPLAIN Plan for SELECT With GROUP BY Using a Serial Plan (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 6 columns
```

Example 6-15. EXPLAIN Plan for SELECT With GROUP BY Using a Serial Plan (page 2 of 2)

```
      Access path 1     : Primary
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.       : None
      Base table pred.  : None

    Executor pred.    : None
    Executor aggr.    : Computed for each group
                        COUNT ( * )
    Table selectivity : Expect to select 100% of rows from table
    Expected row count: 57 rows after the scan
    Operation cost    : 2


  Operation 1.1   : Hash
    Requested         : By the optimizer
    Hash rows in the  : Result of a Select
    Purpose           : To form groups of rows for a Group By
    Hash key columns  : EMPLOYEE.DEPTNUM , EMPLOYEE.JOBCODE
    Expected row count: 57 rows after the group by
    Hash cost         : 1

  Total cost        : 4
```

The plan consists of one step involving two operations: a scan of the EMPLOYEE table and a hash operation. The plan contains this information:

- The aggregate function COUNT, which is computed for each group, is evaluated at the executor level:

  ```
  Executor aggr.    : Computed for each group
  ```

  GROUP BY Using a Serial Plan on page 3-48 lists the conditions that must be met if you want the disk process to do the aggregation. If all the conditions had been satisfied, DP2 aggregation would have taken place in this step. The entry in the EXPLAIN plan would look like this:

  ```
  DP2 aggregate     : Computed for each group
                            COUNT ( * )
  ```

- The hash operation is requested explicitly in the query as a result of the GROUP BY clause; its purpose is to group rows for the aggregate function.

SELECT With GROUP BY Using a Parallel Plan

This query creates an index on DEPTNUM and requests parallel execution:

```
CONTROL EXECUTOR PARALLEL EXECUTION ON ;

CREATE INDEX XDEPT ON EMPLOYEE (DEPTNUM,JOBCODE)
  PARTITION ( $DATA2.PERSNL.XDEPT FIRST KEY 4,
              $DATA5.PERSNL.XDEPT FIRST KEY 7);

CONTROL TABLE EMPLOYEE ACCESS PATH INDEX XDEPT ;
EXPLAIN
 SELECT DEPTNUM, JOBCODE, COUNT(*) FROM EMPLOYEE
  GROUP BY 1,2;
```

Example 6-16. EXPLAIN Plan for SELECT With GROUP BY Using a Parallel Plan (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1   : Will utilize parallel execution
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1

  Each operation is performed in parallel for this step

  Each ESP will read one of the following partitions:
    \SQL1.$DATA2  \SQL1.$DATA5  \SQL1.$DATA8
  The ESP's will be started in the cpu's numbered
    0  2  1

  Group By will be performed by 3 ESP's in parallel

---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XDEPT, index only,
                        partitioned, path forced
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from index
    Index pred.       : None
    Base table pred.  : None
```

Example 6-16. EXPLAIN Plan for SELECT With GROUP BY Using a Parallel Plan (page 2 of 2)

```
    Executor pred.    : None
    Executor aggr.    : Computed for each group
                        COUNT ( * )
    Table selectivity : Expect to select 100% of rows from table
    Expected row count: 57 rows after the scan
    Operation cost    : 151

  Operation 1.1  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Result of a Select
    Purpose           : To form groups of rows for a Group By
    Hash key columns  : EMPLOYEE.DEPTNUM , EMPLOYEE.JOBCODE
    Expected row count: 57 rows after the group by
    Hash cost         : 1

  Total cost        : 153
```

The plan has one step with two operations:

- In Operation 1.0, SQL scans the EMPLOYEE table. The executor computes aggregate values for the local partitions:

  ```
  Executor aggr.    : Computed for each group
                      COUNT ( * )
  ```

  GROUP BY Using a Parallel Plan on page 3-49 lists the conditions that must be met if you want the disk process to do the aggregation. If all the conditions had been satisfied, DP2 aggregation would have taken place in this step. The entry in the EXPLAIN plan would look like this:

  ```
  DP2 aggregate     : Computed for each group
                      COUNT ( * )
  ```

- In Operation 1.1, the master executor uses a hash table to compute the global aggregate results.

EXPLAIN Plans for Subqueries

These examples show EXPLAIN plans for two types of subqueries: noncorrelated and correlated.

The queries retrieve data from the ordone and ordtwo tables. Both tables have system-defined primary keys (SYSKEY).

Noncorrelated Subquery

This query and EXPLAIN plan are for a noncorrelated subquery.

```
EXPLAIN
 SELECT *
   FROM ordone
  WHERE A = ( SELECT B FROM ordtwo ) ;
```

The total cost for the query is 2.

Example 6-17. EXPLAIN Plan for Noncorrelated Subquery

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------
Operation 1.0  : Scan
  Table             : \SQL1.$DATA7.REG1.ORD1
                      with correlation name ORDONE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 2 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        A = B .. result of plan step 2
    Pred. selectivity : Expect to select 50% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 50% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 1

--------------------------------------------------------------------------
Plan step 2
Characteristic : Executes once before plan step 1
---------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA7.REG1.ORD2
                      with correlation name ORDTWO
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 2 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 1

Total cost        : 2
```

The plan consists of two steps:

- Plan step 1 includes a scan of the ordone table.
- Plan step 2 includes a scan of the ordtwo table and is an evaluation of the subquery:

  ```
  SELECT B FROM ordtwo
  ```

Plan step 2 executes once before plan step 1, which indicates that the subquery is evaluated once before the outer query and that this is a noncorrelated subquery.

Correlated Subquery

This query and EXPLAIN plan are for a correlated subquery:

```
EXPLAIN
 SELECT *
   FROM ordone o1
  WHERE NOT EXISTS ( SELECT B
                       FROM ordtwo o2
                      WHERE o1.A = o2.B ) ;
```

The total cost for this query is 3.

Example 6-18. EXPLAIN Plan for Correlated Subquery (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA7.REG1.ORD1
                      with correlation name O1
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 2 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None
```

Example 6-18. EXPLAIN Plan for Correlated Subquery (page 2 of 2)

```
    Executor pred.    : On rows retrieved by the scan
                        NOT ( EXISTS ( .. result of plan step 2 ) )
    Pred. selectivity : Expect to select 60% of rows from table
    Table selectivity : Expect to select 60% of rows from table
    Expected row count: 1 row after the scan
    Operation cost    : 1

  ---------------------------------------------------------------------------
  Plan step 2
  Characteristic : Executes once per row retrieved in plan step 1
  ---------------------------------------------------------------------------
    Operation 2.0  : Scan
    Table             : \SQL1.$DATA7.REG1.ORD2
                        with correlation name O2
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 0 out of 2 columns

      Access path 1     : Primary
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.       : None
      Base table pred.  : Will be evaluated by the disk process
                          O2.B = O1.A
      Pred. selectivity : Expect to select 100% of rows from table

    Executor pred.    : None
    Table selectivity : Expect to select 100% of rows from table
    Expected row count: 1 row after the scan
    Operation cost    : 1

  Total cost        : 3
```

The plan consists of two steps:

- Plan step 1 is a scan of the ordone table.
- Plan step 2 includes a scan of the ordtwo table and is an evaluation of the subquery:

```
SELECT B FROM ordtwo
WHERE ordone.A = ordtwo.B
```

Plan step 2 executes once per row retrieved in plan step 1, which indicates that the subquery is evaluated for every row retrieved from plan step 1 and is therefore a correlated subquery.

EXPLAIN Plans for CASE

This subsection contains EXPLAIN plans that show how the optimizer processes CASE.

CASE With Multiple Conditions

This query selects the last names, first names, and department numbers of employees with job codes of 100 whose salaries are less than 100,000 and also those employees with job codes of 600 whose salaries are less than 30,000.

```
EXPLAIN
  SELECT LAST_NAME,FIRST_NAME,DEPTNUM FROM EMPLOYEE
  WHERE SALARY < CASE JOBCODE
      WHEN 100 THEN  100000
      WHEN 600 THEN  30000
      END ;
```

Example 6-19. EXPLAIN Plan for CASE With Multiple Conditions

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        SALARY < CASE WHEN JOBCODE = 100 THEN 100000 WHEN
                        JOBCODE = 600 THEN 30000 END
    Pred. selectivity : Expect to select 33.33% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 33.33% of rows from table
  Expected row count: 19 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

The plan consists of one step with one operation. The CASE expression shows as a base table predicate that is evaluated by the disk process. The total cost is 2.

CASE With Aggregates

This create statement and query appear under Computing Aggregates Based on Specific Conditions on page 1-45.

```
create table emp
      (name         char(8),
       age          smallint NOT NULL,
       dept         int,
       cars         smallint NOT NULL,
       primary key name);

select SUM(CASE when cars = 0 then 1 else 0 END),
       SUM(CASE when cars = 1 then 1 else 0 END),
       SUM(CASE when cars between 2 and 3 then 1 else 0 END),
       SUM(CASE when cars > 3 then 1 else 0 END)
  from emp;
```

Example 6-20. EXPLAIN Plan for CASE With Aggregates

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PUBS.EMP
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 4 columns

    Access path 1      : Primary
    SBB for reads      : Not used
    Begin key pred.    : None
    End key pred.      : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.        : None
    Base table pred.   : None

  Executor pred.    : None
  DP2 aggregate     : Computed for each group
                      SUM ( CASE WHEN CARS = 0 THEN 1 ELSE 0 END )
                      SUM ( CASE WHEN CARS = 1 THEN 1 ELSE 0 END )
                      SUM ( CASE WHEN ( CARS >= 2 ) AND ( CARS <= 3 )
                      THEN 1
                      ELSE 0 END )
                      SUM ( CASE WHEN CARS > 3 THEN 1 ELSE 0 END )
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 81 rows after the scan
  Operation cost    : 3

Total cost        : 3
```

The plan is one step with one operation. Access is by primary key. The optimizer uses DP2 aggregates to compute each group. The total cost is 3.

CASE for Finding the Highest Value in a Row

Following are the create statement and the query in Finding the Highest Value in a Row on page 1-47. The query retrieves the highest SAT scores from each row in the scores table.

```
create table scores
      (name         char(30),
       sat1         int NOT NULL,
       sat2         int NOT NULL,
       primary key name);

select name, CASE
               when sat1 >= sat2 then sat1
               else sat2
             END
  from scores;
```

Example 6-21. EXPLAIN Plan for CASE for Finding the Highest Value in a Row

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PUBS.SCORES
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 3 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 90 rows after the scan
  Operation cost    : 3

Total cost        : 3
```

The EXPLAIN plan shows that the optimizer uses the primary key for access and expects to examine all the rows. CASE is not mentioned in the plan. The total cost is 3.

CASE for Converting Long, Narrow Tables Into Short, Wide Ones

Following are the create statement and the query in Converting Long, Narrow Tables Into Short, Wide Ones on page 1-49.

```
create table bonus
      (name         char(30),
       month        smallint NOT NULL,
       amount       largeint NOT NULL,
       primary key (name, month));
select name,
       SUM(CASE when month =  1 then amount else 0 END),
       SUM(CASE when month =  2 then amount else 0 END),
       SUM(CASE when month =  3 then amount else 0 END),
       SUM(CASE when month =  4 then amount else 0 END),
       SUM(CASE when month =  5 then amount else 0 END),
       SUM(CASE when month =  6 then amount else 0 END),
       SUM(CASE when month =  7 then amount else 0 END),
       SUM(CASE when month =  8 then amount else 0 END),
       SUM(CASE when month =  9 then amount else 0 END),
       SUM(CASE when month = 10 then amount else 0 END),
       SUM(CASE when month = 11 then amount else 0 END),
       SUM(CASE when month = 12 then amount else 0 END)
  from bonus
 group by name;
```

Example 6-22 on page 6-37 shows the EXPLAIN plan.

Example 6-22. EXPLAIN Plan for Converting Long, Narrow Tables Into Short, Wide Ones

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PUBS.BONUS
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 3 out of 3 columns

    Access path 1     : Primary
    SBB for reads     : Not used
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  DP2 aggregate     : Computed for each group
                      SUM ( CASE WHEN MONTH = 1 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 2 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 3 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 4 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 5 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 6 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 7 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 8 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 9 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 10 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 11 THEN AMOUNT ELSE 0 END )
                      SUM ( CASE WHEN MONTH = 12 THEN AMOUNT ELSE 0 END )
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 81 rows after the scan
  Operation cost    : 3

Total cost        : 3
```

The plan contains one step with one operation. The optimizer uses the primary key to access the table and uses DP2 aggregates to compute the name groups. The total cost is 3.

CASE for Ignoring the Largest and Smallest Values in a Set

Following are the create statement and the query in Ignoring the Largest and Smallest Values in a Set on page 1-51.

```
create table data
      (value       int NOT NULL,
       primary key  value);
```

```
select x.value
  from data x, data y
 group by x.value
 having SUM (CASE when y.value <= x.value then 1 else 0 END) > 1
   AND SUM (CASE when y.value >= x.value then 1 else 0 END) > 1;
```

Example 6-23. EXPLAIN Plan for Ignoring the Largest and Smallest Values in a Set (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PUBS.DATA
                      with correlation name X
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 1 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : From the Having clause
                      ( SUM ( CASE WHEN Y.VALUE <= X.VALUE THEN 1 ELSE 0
                      END
                      ) > 1 ) AND ( SUM ( CASE WHEN Y.VALUE >= X.VALUE THEN
                      1
                      ELSE 0 END ) > 1 )
  Executor aggr.    : Computed for each group
                      SUM ( CASE WHEN Y.VALUE <= X.VALUE THEN 1 ELSE 0
                      END )
                      SUM ( CASE WHEN Y.VALUE >= X.VALUE THEN 1 ELS 0
                      END )

  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 204 rows after the scan
  Expected row count: 100 rows after the having
  Operation cost    : 6
```

Example 6-23. EXPLAIN Plan for Ignoring the Largest and Smallest Values in a Set (page 2 of 2)

```
   --------------------------------------------------------------------------
  Plan step 2    : Perform an Inner Join
  Join strategy  : Nested Join
  Characteristic : Joins a row resulting from plan step 1
 ---------------------------------------------------------------------------

  Operation 2.0  : Scan
    Table             : \SQL1.$DATA8.PUBS.DATA
                        with correlation name Y
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 1 out of 1 columns

      Access path 1     : Primary
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.       : None
      Base table pred.  : None

    Executor pred.    : None
    Table selectivity : Expect to select 100% of rows from table
    Expected row count: 41616 rows after the join
    Operation cost    : 6

  Total cost        : 1162
```

The plan contains two steps. Plan step 1 is a scan of the table. The plan shows that the HAVING clause is an executor predicate and the GROUP BY clause is an executor aggregate. The expected row count is 204 after the scan and 100 after the HAVING clause. The operation cost is 6.

Plan step 2 is a nested inner join that scans the same table and joins the rows that resulted from plan step 1. Its expected row count is 41,616 after the join.

The total cost is 1162.

EXPLAIN Plans for String Functions

This subsection contains EXPLAIN examples for queries that contain the string functions SUBSTRING, TRIM, and CHAR_LENGTH.

SUBSTRING

This query uses a substring to search for the letter “C” in the first position of each employee’s last name.

```
EXPLAIN
  SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEE
  WHERE SUBSTRING(LAST_NAME FROM 1 FOR 1) = 'C' ;
```

Example 6-24. EXPLAIN Plan for SUBSTRING

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        SUBSTRING ( LAST_NAME FROM 1 FOR 1 ) = "C"
    Pred. selectivity : Expect to select 1.7544% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 1.7544% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 2

Total cost        : 2
```

The plan consists of one step with one operation. The substring shows as a base table predicate that is evaluated by the disk process. The total cost is 2.

TRIM and CHAR_LENGTH

This query concatenates the first and last names of employees whose last names have more than five characters:

```
EXPLAIN
  SELECT LAST_NAME, FIRST_NAME || LAST_NAME FROM EMPLOYEE
  WHERE CHAR_LENGTH(TRIM(LAST_NAME)) > 5 ;
```

Example 6-25. EXPLAIN Plan for TRIM and CHAR_LENGTH

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        CHAR_LENGTH ( TRIM ( BOTH " " FROM LAST_NAME ) )
                        > 5
    Pred. selectivity : Expect to select 33.33% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 33.33% of rows from table
  Expected row count: 19 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

The plan consists of one step with one operation. The CHAR_LENGTH and TRIM functions appear in the base table predicate evaluated by the disk process. The total cost is 2.

EXPLAIN Plans for MDAM

These examples show how the optimizer uses MDAM.

MDAM With OR and Equality Predicate on Second Key Column

In this example, an index is created on LAST_NAME, FIRST_NAME. The query specifies an OR on the first key column (LAST_NAME) and an equality predicate on the second key column (FIRST_NAME).

```
CREATE INDEX XEMPNAME
 ON EMPLOYEE
 (LAST_NAME, FIRST_NAME);
```

```
EXPLAIN SELECT * FROM EMPLOYEE
        WHERE (LAST_NAME = "Marks" OR LAST_NAME = "Jones") AND
              FIRST_NAME = "Mary";
```

Example 6-26. EXPLAIN Plan for MDAM With OR and Equality Predicate on Second Key Column

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

----------------------------------------------------------------------------
Plan step 1
----------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XEMPNAME
    SBB for reads     : Virtual
    MDAM predicate set: ( LAST_NAME = "Jones" OR "Marks" ) AND FIRST_NAME =
                        "Mary"
    Index selectivity : Expect to examine 3.5088% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 3.5088% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 4

Total cost        : 4
```

The OR and the equality predicates show in the MDAM predicate set. The total cost is 4.

MDAM with Missing First Key Column

This creates an index on LAST_NAME, FIRST_NAME. The first key column is missing from the query predicate.

```
CREATE INDEX XEMPNAME
 ON EMPLOYEE
 (LAST_NAME, FIRST_NAME);
```

```
EXPLAIN SELECT * FROM EMPLOYEE WHERE FIRST_NAME = "Mary";
```

Example 6-27. EXPLAIN Plan for MDAM With Missing First Key Column

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XEMPNAME
    SBB for reads     : Virtual
    MDAM predicate set: ( ALL LAST_NAME VALUES ) AND FIRST_NAME = "Mary"
    Index selectivity : Expect to examine 96.4912% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 96.4912% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 169

Total cost        : 169
```

Although the LAST_NAME column is missing from the query, the MDAM predicate set shows that SQL considers all LAST_NAME values. It finds all the values for LAST_NAME and uses them with the specified value for FIRST_NAME to retrieve only the qualifying rows. The total cost is 169.

MDAM With IN List on Key Column

This creates an index on LAST_NAME, FIRST_NAME. The query uses MDAM to process an IN list for the LAST_NAME key column.

```
CREATE INDEX XEMPNAME
 ON EMPLOYEE
 (LAST_NAME, FIRST_NAME);
```

```
EXPLAIN SELECT * FROM EMPLOYEE
   WHERE LAST_NAME IN ("Marks","Jones") AND FIRST_NAME = "Mary";
```

Example 6-28. EXPLAIN Plan for MDAM With IN List on Key Column

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XEMPNAME
    SBB for reads     : Virtual
    MDAM predicate set: ( LAST_NAME = "Jones" OR "Marks" ) AND FIRST_NAME =
                        "Mary"
    Index selectivity : Expect to examine 3.5088% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 3.5088% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 4

Total cost        : 4
```

The plan converts the IN list for the key column LAST_NAME to a list of OR predicates. The total cost of the query is 4.

MDAM With Multiple Predicate Sets, LIKE, and Missing Key Column

This query uses MDAM to process the following:

- A predicate that specifies both key columns
- A predicate with a missing key column
- LIKE

An index is created on LAST_NAME, FIRST_NAME.

```
CREATE INDEX XEMPNAME
 ON EMPLOYEE
 (LAST_NAME, FIRST_NAME);
```

```
EXPLAIN SELECT * FROM EMPLOYEE
        WHERE ((LAST_NAME = "Marks" OR LAST_NAME = "Jones") AND
               FIRST_NAME = "Mary")
        OR
              (FIRST_NAME LIKE "Ha%");
```

Example 6-29. EXPLAIN Plan for MDAM With Multiple Predicate Sets, LIKE, and Missing Leading Key Column

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

----------------------------------------------------------------------------
Plan step 1
----------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XEMPNAME
    SBB for reads     : Virtual
    MDAM predicate set: ( LAST_NAME = "Jones" OR "Marks" ) AND FIRST_NAME =
                        "Mary"
              next set: ( ALL LAST_NAME VALUES ) AND FIRST_NAME < "Hb" AND
                        FIRST_NAME >= "Ha"
    Index selectivity : Expect to examine 3.5088% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 3.5088% of rows from table
  Expected row count: 13 rows after the scan
  Operation cost    : 173

Total cost        : 173
```

SQL converts the predicates to two MDAM predicate sets.

In the first predicate set, both key columns are specified. In the second predicate set, the leading key column (LAST_NAME) is missing. SQL finds all values for this column.

Duplicate key predicates and contradictory key predicates are eliminated at run time.

SQL converts the LIKE to a range.

EXPLAIN Plan for Determining the Cost of Multiple Predicate Sets

This query contains a predicate that SQL converts into multiple predicate sets. An index is created for DEPTNUM and JOBCODE.

```
CREATE INDEX HLX ON EMPLOYEE (DEPTNUM, JOBCODE) CATALOG PERSNL ;
EXPLAIN
  SELECT * FROM EMPLOYEE
    WHERE ((DEPTNUM = 1500 AND JOBCODE BETWEEN 100 AND 300)
      OR JOBCODE IN (400, 450))
    AND ((SALARY = 50000 AND JOBCODE = 450) OR (JOBCODE IN
        (500,600)
     AND (DEPTNUM = 4000 OR DEPTNUM = 7000))) ;
```

Example 6-30. EXPLAIN Plan for Determining the Cost of Multiple Predicate Sets (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PUBS.HLX
    SBB for reads     : Not used
    MDAM predicate set: DEPTNUM = 1500 AND JOBCODE <= 300 AND JOBCODE >=
                        100 AND JOBCODE = 450
              next set: DEPTNUM = 1500 AND ( DEPTNUM = 7000 OR 4000 ) AND
                        JOBCODE <= 300 AND JOBCODE >= 100 AND ( JOBCODE =
                        600 OR 500 )
              next set: ( ALL DEPTNUM VALUES ) AND ( JOBCODE = 450 OR 400 )
                        AND JOBCODE = 450
              next set: ( DEPTNUM = 7000 OR 4000 ) AND ( JOBCODE = 450 OR
                        400 AND ( JOBCODE = 600 OR 500 )
    Index selectivity : Expect to examine 30.5861% of rows from index
    Index pred.       : Will be evaluated by the disk process
                        ( ( DEPTNUM = 1500 ) AND ( JOBCODE >= 100 ) AND (
                        JOBCODE <= 300 ) ) OR ( JOBCODE = 400 ) OR (
                        JOBCODE = 450 )
```

Example 6-30. EXPLAIN Plan for Determining the Cost of Multiple Predicate Sets (page 2 of 2)

```
      Pred. selectivity : Expect to select 30.5861% of rows from index
      Base table pred.  : Will be evaluated by the disk process
                          ( ( SALARY = 50000 ) AND ( JOBCODE = 450 ) ) OR ( (
                          ( JOBCODE = 500 ) OR ( JOBCODE = 600 ) ) AND ( (
                          DEPTNUM = 4000 ) OR ( DEPTNUM = 7000 ) ) )
      Pred. selectivity : Expect to select 30.5861% of rows from table

    Executor pred.    : None
    Table selectivity : Expect to select 31.3476% of rows from table
    Expected row count: 1 row after the scan
    Operation cost    : 48

  Total cost        : 48
```

SQL converts the predicates into four MDAM predicate sets. In the first set, a positioning takes place for each unique JOBCODE value between 100 and 300.

In following predicate sets, SQL converts IN lists for JOBCODE values into OR predicates. A positioning is done on each of the equal predicates for JOBCODE. The total cost for the query is 48.

EXPLAIN Plan for Selectivity for Range Predicates

This query contains range predicates:

```
UPDATE STATISTICS FOR TABLE EMPLOYEE;
EXPLAIN
 SELECT * FROM EMPLOYEE
  WHERE JOBCODE >=  150
    AND JOBCODE <= 500 ;
```

Example 6-31 on page 6-48 shows the selectivity.

Example 6-31. EXPLAIN Plan for Selectivity for Range Predicates

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 6 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        ( JOBCODE >= 150 ) AND ( JOBCODE <= 500 )
    Pred. selectivity : Expect to select 61.425% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 61.425% of rows from table
  Expected row count: 35 rows after the scan
  Operation cost    : 2

Total cost        : 2
```

The EXPLAIN plan has one step. SQL expects that approximately 61 percent of the rows have JOBCODE values between 150 and 500. SQL computed this value by calculating selectivities separately for the >= 150 predicate and the <=500 predicate and applying the rules provided in Selectivity for Range Predicates on page 5-5.

EXPLAIN Plans for Join Queries

This subsection shows EXPLAIN plans for join queries.

Join queries involve two or more tables. The optimizer determines the outer (left) and inner tables based on the number of rows selected. Usually, the table with the most rows selected is chosen as the outer table. This table is listed first in the EXPLAIN plan.

The plans chosen show both types of join queries (left and inner), the four types of join methods (hash, nested, sort merge, and key-sequenced merge), and parallel execution of queries.

For a plan involving parallel execution, each partition is read by an ESP in a separate processor. The EXPLAIN plan indicates which partitions will be read by each of the ESPs and in which processors.

For more information on ESPs, see Processor Assignment by the SQL/MP Optimizer and Executor for Executor Server Processes (ESPs) on page 2-5.

Parallel Execution of Hash Join

The EXPLAIN plan in Example 6-32 chooses parallel execution for a join operation. SQL chooses the hash join method to join the tables. The hash join method is often chosen when joining a large table and a much smaller table. The query follows:

```
EXPLAIN
  SELECT *
  FROM TENKTUP1 A, TENKTUP2 B
  WHERE A.TENPCT = B.TENPCT ;
```

The query involves a scan of two tables, TENKTUP1 (correlation name A) and TENKTUP2 (correlation name B), which are joined according to the following search condition: WHERE A.TENPCT = B.TENPCT.

Example 6-32. EXPLAIN Plan for Hash Join (page 1 of 3)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1   : Will utilize parallel execution
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1

  Each operation is performed in parallel for this step

  Each ESP will read one of the following partitions:
    \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
  The ESP's will be started in the cpu's numbered
    0  2  3

 ---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP1
                      with correlation name A
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns

    Access path 1     : Primary, partitioned, sequential cache
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 74999 rows after the scan
  Operation cost    : 2279
```

Example 6-32. EXPLAIN Plan for Hash Join (page 2 of 3)

```
---------------------------------------------------------------------------
 Plan step 2    : Perform an Inner Join
 Join strategy  : Hybrid Hash Join

   Each operation is performed in parallel for this step

   Each ESP will read one of the following partitions:
     \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
   The ESP's will be started in the cpu's numbered
     0  1  2

   Current table and join composite (excluding current table)
    will each be repartitioned 4 ways on the join column to:
     \SQL1.$DATA7  \SQL1.$DATA4  \SQL1.$DATA6  \SQL1.$DATA3

   4 ESP's will be started to read the repartitioned data
   The ESP's will be started in the cpu's numbered
     2  0  3  1

   Each ESP will:
     Hash one repartition of the join composite excluding the current table
     Hash one repartition of the current table

   Each ESP will perform a Hybrid Hash Join (repartitioned)
 Characteristic : Joins a row resulting from plan step 1

 ---------------------------------------------------------------------------

 Operation 2.0  : Scan
   Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                       with correlation name B
   Access type       : Record locks, stable access
   Lock mode         : Chosen by the system
   Column processing : Requires retrieval of 16 out of 16 columns

     Access path 1     : Primary, partitioned, sequential cache
     SBB for reads     : Virtual
     Begin key pred.   : None
     End key pred.     : None
     Index selectivity : Expect to examine 100% of rows from table
     Index pred.       : None
     Base table pred.  : None

   Executor pred.    : On hashed rows
                       A.TENPCT = B.TENPCT
   Pred. selectivity : Expect to select 0.0131% of rows from table
   Table selectivity : Expect to select 0.0131% of rows from table
   Expected row count: 733929 rows after the join
   Operation cost    : 2276
```

Example 6-32. EXPLAIN Plan for Hash Join (page 3 of 3)

```
  Operation 2.1  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Join composite excluding current table
    Purpose           : To hash its rows before the Join
    Hash key columns  : A.TENPCT
    Hash cost         : 126

  Operation 2.2  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Current table
    Purpose           : To hash its rows before the Join
    Hash key columns  : B.TENPCT
    Hash cost         : 631

  Total cost        : 7959
```

The plan consists of two steps:

- Plan step 1 includes a scan of the TENKTUP1 table, which is partitioned across three disk volumes.
- Plan step 2 includes a scan of the TENKTUP2 table, which is then inner joined with the TENKTUP1 table.

Plan step 2 involves the following:

- A hashing operation on the predicate A.TENPCT = B.TENPCT (instead of a sort operation).
- Performance of each operation in parallel:
  - Each ESP joins one of the partitions and performs a hash join with parallel access
  - The current table is repartitioned three ways on the join column
  - An ESP is started to read the repartitioned data

Nested Inner Join

Example 6-33 on page 6-52 lists EXPLAIN output for a query that involves a scan of two tables, TENKTUP1 (correlation name A) and TENKTUP2 (correlation name B). The tables are joined according to this search condition: WHERE A.UNIQUE2 <10 AND B.UNIQUE1 = 3:

```
EXPLAIN SELECT * FROM TENKTUP1 A, TENKTUP2 B
  WHERE A.UNIQUE2 < 10 AND B.UNIQUE1 = 3 ;
```

This EXPLAIN plan chooses parallel execution for an inner join query. The nested-join method is chosen instead of a hashed, sort merge, or key-sequenced merge join because the join predicate is “less than” (<) instead of equal.

Example 6-33. EXPLAIN Plan for Nested Inner Join (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1   : Will utilize parallel execution
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1

  Each operation is performed in parallel for this step

  Each ESP will read one of the following partitions:
    \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
  The ESP's will be started in the cpu's numbered
    0  2  3

---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP1
                      with correlation name A
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns

    Access path 1     : Primary, partitioned
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : A.UNIQUE2 < 10
    Index selectivity : Expect to examine 0.0133% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 0.0133% of rows from table
  Expected row count: 10 rows after the scan
  Operation cost    : 152
---------------------------------------------------------------------------
Plan step 2    : Perform an Inner Join
Join strategy  : Nested Join

  Each operation is performed in parallel for this step

  Each ESP from previous step will join one of the following partitions:
    \SQL1.$DATA2
  Each ESP will perform a Nested Join (parallel access)
Characteristic : Joins a row resulting from plan step 1
```

Example 6-33. EXPLAIN Plan for Nested Inner Join (page 2 of 2)

```
--------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                      with correlation name B
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns

    Access path 1     : Alternate \SQL1.$DATA2.WISCREG.TKTUP2I, unique
    SBB for reads     : Not used
    Begin key pred.   : B.UNIQUE1 = 3
    End key pred.     : B.UNIQUE1 = 3
    Index selectivity : Expect to examine 0.0013% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 0.0013% of rows from table
  Expected row count: 10 rows after the join
  Operation cost    : 3

Total cost        : 174
```

This plan consists of two steps:

- Plan step 1 includes a scan of the TENKTUP1 table.
- Plan step 2 includes a scan of the TENKTUP2 table, which is then inner joined with the TENKTUP1 table.

Cross Product Join

A cross product join is a nested join without any join predicates. Example 6-34 on page 6-54 shows the EXPLAIN plan for the following join.

```
EXPLAIN SELECT * FROM TENKTUP1 A,TENKTUP2 B,TENKTUP2 C
  WHERE A.UNIQUE2 <10 AND B.UNIQUE1 = 3
  AND A.TENPCT=C.TENPCT
  AND B.TWENTY=C.TWENTY;
```

The plan includes a cross product as an intermediate step.

Example 6-34. EXPLAIN Plan for Cross Product Join (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                      with correlation name B
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns

    Access path 1     : Alternate \SQL1.$DATA2.WISCREG.TKTUP2I, unique
    SBB for reads     : Not used
    Begin key pred.   : B.UNIQUE1 = 3
    End key pred.     : B.UNIQUE1 = 3
    Index selectivity : Expect to examine 0.0013% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 0.0013% of rows from table
  Expected row count: 10 rows after the join
  Operation cost    : 3

---------------------------------------------------------------------------
Plan step 2    : Perform an Inner Join
Join strategy  : Nested Join
Characteristic : Joins a row resulting from plan step 1
---------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                      with correlation name C
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns
```

Example 6-34. EXPLAIN Plan for Cross Product Join (page 2 of 2)

```
Access path 1     : Primary, partitioned, sequential cache
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.       : None
      Base table pred.  : Will be evaluated by the disk process
                          B.TWENTY = C.TWENTY
      Pred. selectivity : Expect to select 5% of rows from table

    Executor pred.    : None
    Table selectivity : Expect to select 5% of rows from table
    Expected row count: 3734 rows after the join
    Operation cost    : 6322

  ---------------------------------------------------------------------------
  Plan step 3    : Perform an Inner Join
  Join strategy  : Hash Join
  Characteristic : Joins a row resulting from plan step 2
  ---------------------------------------------------------------------------

  Operation 3.0  : Scan
    Table             : \SQL1.$DATA2.WISCREG.TENKTUP1
                        with correlation name A
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 16 out of 16 columns

      Access path 1     : Primary, partitioned
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : A.UNIQUE2 < 10
      Index selectivity : Expect to examine 0.0133% of rows from table
      Index pred.       : None
      Base table pred.  : None

    Executor pred.    : On hashed rows
                        A.TENPCT = C.TENPCT
    Pred. selectivity : Expect to select 0% of rows from table
    Table selectivity : Expect to select 0.0013% of rows from table
    Expected row count: 5 rows after the join
    Operation cost    : 2

  Operation 3.1  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Current table
    Purpose           : To hash its rows before the Join
    Hash key columns  : A.TENPCT
    Hash cost         : 1

  Total cost        : 6227
```

This plan consists of three steps:

- Plan step 1 includes a scan of the TENKTUP2 table.
- The TENKTUP2 table is also scanned in step 2 and is then inner joined with the TENKTUP1 table (cross product step).
- Plan step 3 includes a scan of the TENKTUP1 table, which is then inner joined with the result of step 2.

Parallel Execution of Nested Inner Join

This example and the following merged join example both use the query from Example 6-32 on page 6-49, but each uses a different CONTROL TABLE directive:

```
CONTROL TABLE * JOIN METHOD NESTED;

EXPLAIN
  SELECT *
    FROM TENKTUP1 A, TENKTUP2 B
    WHERE A.TENPCT = B.TENPCT ;
```

Example 6-35. EXPLAIN Plan for Nested Inner Join (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1   : Will utilize parallel execution
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1

  Each operation is performed in parallel for this step

  Each ESP will read one of the following partitions:
    \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
  The ESP's will be started in the cpu's numbered
    0  2  3

---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                      with correlation name B
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns

    Access path 1     : Primary, partitioned, sequential cache
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 74666 rows after the scan
  Operation cost    : 2276
```

Example 6-35. EXPLAIN Plan for Nested Inner Join (page 2 of 2)

```
---------------------------------------------------------------------------
Plan step 2    : Perform an Inner Join
  Join strategy  : Nested Join
  Plan Forced    : Join Method forced by user directive

    Each operation is performed in parallel for this step

    Each ESP from previous step will join one of the following partitions:
      \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
    Each ESP will perform a Nested Join (parallel access)
  Characteristic : Joins a row resulting from plan step 1
  ---------------------------------------------------------------------------

   Operation 2.0  : Scan
    Table              : \SQL1.$DATA2.WISCREG.TENKTUP1
                         with correlation name A
    Access type        : Record locks, stable access
    Lock mode          : Chosen by the system
    Column processing : Requires retrieval of 16 out of 16 columns

      Access path 1      : Primary, partitioned, sequential cache
      SBB for reads      : Not used
      Begin key pred.    : None
      End key pred.      : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.        : None
      Base table pred.   : Will be evaluated by the disk process
                           A.TENPCT = B.TENPCT
      Pred. selectivity : Expect to select 0.0131% of rows from table

    Executor pred.     : None
    Table selectivity : Expect to select 0.0131% of rows from table
    Expected row count: 733929 rows after the join
    Operation cost     : 6297

  Total cost         : 56658817
```

The plan contains the following information:

- Both steps are performed in parallel. For a plan involving parallel execution, each partition is read by an ESP in a separate processor. The EXPLAIN plan states which partitions will be read by each of the ESPs and in which processors.
- Step 2 indicates that the join was forced by user directive. In practice, this directive should be used with caution. To force a join method, use the CONTROL TABLE directive. For more information, see Specifying a Join Method on page 3-43. Without this directive, SQL would choose the hash join method for this query; its cost, listed in Example 6-32 on page 6-49, is lower than that of the nested join.

If the total cost for the nested inner join seems large in comparison to the operation costs for operations 1.0 and 2.0, the reason is that SQL calculates the total cost by first multiplying the cost of the scan by the number of probes (outer composite expected row counts). Note that this amount is then reduced to reflect the benefit of using cache.

Parallel Execution of Forced Merged Inner Join

This EXPLAIN plan chooses parallel execution for an inner join query. The merge join method was chosen to join the tables. The query for this example is the same as the one for Example 6-32 on page 6-49, but the CONTROL TABLE directive differs:

```
CONTROL TABLE * JOIN METHOD MERGE;

EXPLAIN
  SELECT *
    FROM TENKTUP1 A, TENKTUP2 B
    WHERE A.TENPCT = B.TENPCT ;
```

In Example 6-32, the query is run by default and the optimizer chooses a hybrid hash join. But in Example 6-36, a join method is forced, so the optimizer chooses a parallel sort merge join.

Example 6-36. EXPLAIN Plan for Parallel Execution of Sort Merge Inner Join (page 1 of 3)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1   : Will utilize parallel execution
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

--------------------------------------------------------------------------
Plan step 1

  Each operation is performed in parallel for this step

  Each ESP will read one of the following partitions:
    \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
  The ESP's will be started in the cpu's numbered
    0  3  1

--------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                      with correlation name B
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 16 out of 16 columns

    Access path 1     : Primary, partitioned, sequential cache
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 74666 rows after the scan
  Operation cost    : 2276
```

Example 6-36. EXPLAIN Plan for Parallel Execution of Sort Merge Inner Join (page 2 of 3)

```
  ---------------------------------------------------------------------------
  Plan step 2    : Perform an Inner Join
  Join strategy  : Merge Join
  Plan Forced    : Join Method forced by user directive

    Each operation is performed in parallel for this step

    Each ESP will read one of the following partitions:
      \SQL1.$DATA3  \SQL1.$DATA4  \SQL1.$DATA2
    The ESP's will be started in the cpu's numbered
      0  1  2

    Current table and join composite (excluding current table)
     will each be repartitioned 4 ways on the join column to:
      \SQL1.$D30SYS  \SQL1.$DATA10  \SQL1.$DSSQA1  \SQL1.$DATA9

    4 ESP's will be started to read the repartitioned data
    The ESP's will be started in the cpu's numbered
      0  2  3  1

    Each ESP will start 2 sorts to:
      Sort one repartition of the join composite excluding the current table
      Sort one repartition of the current table
    Each ESP will perform a Merge Join (repartitioned)
    Characteristic : Joins a row resulting from plan step 1
  ---------------------------------------------------------------------------
  Operation 2.0  : Scan
    Table             : \SQL1.$DATA2.WISCREG.TENKTUP1
                        with correlation name A
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 16 out of 16 columns

      Access path 1     : Primary, partitioned, sequential cache
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.       : None
      Base table pred.  : None

    Executor pred.    : On sorted rows
                        A.TENPCT = B.TENPCT
    Pred. selectivity : Expect to select 0.0131% of rows from table
    Table selectivity : Expect to select 0.0131% of rows from table
    Expected row count: 733929 rows after the join
    Operation cost    : 2276

Operation 2.1  : Sort
    Requested         : By the optimizer
    Sort rows in the  : Join composite excluding current table
    Purpose           : To order its rows before the Join
    Sort technique    : FASTSORT
    Sort type         : Insertion into an entry-sequenced disk file
    Sort key columns  : B.TENPCT asc
    Sort cost         : 7049
```

Example 6-36. EXPLAIN Plan for Parallel Execution of Sort Merge Inner Join (page 3 of 3)

```
  Operation 2.2  : Sort
    Requested         : By the optimizer
    Sort rows in the  : Current table
    Purpose           : To order its rows before the Join
    Sort technique    : FASTSORT
    Sort type         : Insertion into an entry-sequenced disk file
    Sort key columns  : A.TENPCT asc
    Sort cost         : 7049

  Total cost        : 32307
```

The plan consists of two steps:

- Plan step 1 includes a scan of the TENKTUP2 table, which is partitioned across three disk volumes.
- Plan step 2 includes a scan of the TENKTUP1 table, which is then inner joined with the TENKTUP2 table.

Plan step 2 involves the following operations:

- The scan.
- Two sort operations, each using FastSort, to satisfy the merge join method. Both sort operations are requested by the optimizer for the purpose of ordering rows before the join. The key columns for each sort as well as the cost of each sort are described.
- Each operation is performed in parallel. Each ESP reads one of the partitions. The current table is repartitioned four ways on the join column. An ESP is started to read the repartitioned data. Each ESP starts two sorts:
    - Operation 2.1 is the first sort required to satisfy the merge join method. The sort is requested by the optimizer to order rows in the composite table before completing the join operation.
    - Operation 2.2 is the second sort required to satisfy the merge join method. The sort is requested by the optimizer to sort rows in the current table (TENKTUP1) before completing the join operation.
- Each ESP performs a merge join of the repartitioned data to join a row resulting from plan step 1.

Key-Sequenced Merge Join

This query chooses the key-sequenced merge join to list orders by date and provide a count of the parts per order. The join method is forced for the purpose of this example.

```
CONTROL TABLE * JOIN METHOD KEY SEQUENCED MERGE;

EXPLAIN
  SELECT X.ORDER_DATE, X.ORDERNUM, COUNT(*)
    FROM ORDERS X,ODETAIL Y
    WHERE X.ORDERNUM = Y.ORDERNUM
    GROUP BY X.ORDERNUM,X.ORDER_DATE
    ORDER BY X.ORDER_DATE,X.ORDERNUM ;
```

Example 6-37. EXPLAIN Plan for Key-Sequenced Merge Join (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.SALES.ORDERS
                      with correlation name X
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 5 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Executor aggr.    : Computed for each group
                      COUNT ( * )
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 13 rows after the scan
    Operation cost    : 1

---------------------------------------------------------------------------
Plan step 2    : Perform an Inner Join
Join strategy  : Key-Sequenced Merge Join
Plan Forced    : Join Method forced by user directive
Characteristic : Joins a row resulting from plan step 1
---------------------------------------------------------------------------
Operation 2.0  : Scan
  Table             : \SQL1.$DATA8.SALES.ODETAIL
                      with correlation name Y
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 4 columns
    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : X.ORDERNUM = Y.ORDERNUM
    End key pred.     : None
    Index selectivity : Expect to examine 7.1429% of rows from table
    Index pred.       : None
    Base table pred.  : None
```

Example 6-37. EXPLAIN Plan for Key-Sequenced Merge Join (page 2 of 2)

```
    Executor pred.    : None
    Table selectivity : Expect to select 7.1429% of rows from table
    Expected row count: 67 rows after the join
    Operation cost    : 1

  Operation 2.1  : Sort
    Requested         : Explicitly in the query
    Sort rows in the  : Result of a Select
    Purpose           : To form groups of rows for a Group By
    Sort technique    : FASTSORT
    Sort type         : Plan to use User Process Sort
    UPS workspace     : 24 Kbytes
    Sort key columns  : X.ORDER_DATE asc, X.ORDERNUM asc
    Expected row count: 67 rows after the group by
    Sort cost         : 1

  Total cost        : 5
```

The plan contains two steps. Step two shows that the key-sequenced merge join was chosen.

In plan step 1, the access path is primary and the operation cost is 1.

In plan step 2, the access path is primary and the operation cost for the first operation is 1. The second operation in step 2 includes a sort by X.ORDER_DATE and X.ORDERNUM, both ascending. The cost of the sort is 1.

The total cost for the query is 5.

Key-Sequenced Merge Join With Executor Aggregates

This query also chooses the key-sequenced merge join but contains executor aggregates.

```
EXPLAIN
  SELECT COUNT(*)
     FROM TENKTUP1,TENKTUP2
     WHERE TENKTUP1.UNIQUE1 = TENKTUP2.UNIQUE1 ;
```

Example 6-38. EXPLAIN Plan for Key-Sequenced Merge Join With Executor Aggregates

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

----------------------------------------------------------------------------
Plan step 1
----------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP2
                      with correlation name TENKTUP2
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 16 columns

    Access path 1     : Alternate \SQL1.$DATA2.WISCREG.TKTUP2I, index
                        only, sequential cache
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Executor aggr.    : Computed for each group
                      COUNT ( * )
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 74666 rows after the scan
  Operation cost    : 2187

----------------------------------------------------------------------------
Plan step 2    : Perform an Inner Join
Join strategy  : Key-Sequenced Merge Join
Plan Forced    : Join Method forced by user directive
Characteristic : Joins a row resulting from plan step 1
----------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA2.WISCREG.TENKTUP1
                      with correlation name TENKTUP1
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 16 columns

    Access path 1     : Alternate \SQL1.$DATA2.WISCREG.TKTUP1I, unique,
                        index only, sequential cache
    SBB for reads     : Virtual
    Begin key pred.   : TENKTUP1.UNIQUE1 = TENKTUP2.UNIQUE1
    End key pred.     : None
    Index selectivity : Expect to examine 0.0013% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 0.0013% of rows from table
  Expected row count: 74704 rows after the join
  Operation cost    : 1

Total cost        : 4389
```

This plan contains two steps. Step 2 shows that the key-sequenced merge join method was chosen. Executor aggregates were chosen because aggregates cannot be done in DP2 on the inner table of a key-sequenced merge join.

In plan step 1, the access path is alternate, index only, sequential cache. The operation cost is 2187.

In plan step 2, the access path is alternate, unique, index only, sequential cache. No sort takes place. The operation cost is 1.

The total cost for the query is 4389.

Left Join Not Transformed Into an Inner Join

The EXPLAIN plans in Example 6-39 on page 6-65 and Example 6-40 on page 6-67 are for two queries that use the same view:

```
CREATE VIEW EMPORD AS
SELECT *
    FROM EMPLOYEE E LEFT JOIN ORDERS O
      ON E.EMPNUM = O.SALESREP ;
```

This query executes a left join that is not transformed into an inner join. It does this with an IS NULL predicate.

```
EXPLAIN
  SELECT DISTINCT LAST_NAME
    FROM EMPORD
   WHERE SALESREP IS NULL ;
```

The query for Example 6-40 on page 6-67 executes a left join that *is* transformed into an inner join. It does this with an IS NOT NULL predicate.

In both examples, if the right table does not satisfy the search condition, SQL creates null-augmented rows on the left table. If a WHERE or inner join predicate is certain to eliminate all of the null-augmented rows generated by the left join, then the optimizer transforms the left join into a more efficient inner join.

This example shows the EXPLAIN plan for the left join not transformed into an inner join.

Example 6-39. EXPLAIN Plan for Left Join Not Transformed Into an Inner Join (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

--------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table            : \SQL1.$DATA8.PERSNL.EMPLOYEE
                     with correlation name E
  Access type      : Record locks, stable access
  Lock mode        : Chosen by the system
  Column processing : Requires retrieval of 2 out of 6 columns
    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 57 rows after the scan
  Operation cost    : 2

---------------------------------------------------------------------------
Plan step 2    : Perform a Left Join
Join strategy  : Hash Join
Characteristic : Joins a row resulting from plan step 1
---------------------------------------------------------------------------

Operation 2.0  : Scan
  Table            : \SQL1.$DATA8.SALES.ORDERS
                     with correlation name O
  Access type      : Record locks, stable access
  Lock mode        : Chosen by the system
  Column processing : Requires retrieval of 1 out of 5 columns

    Access path 1     : Alternate \SQL1.$DATA8.SALES.XORDREP, index only
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : On hashed rows
                     O.SALESREP = E.EMPNUM
  Pred. selectivity : Expect to select 11.1111% of rows from table
  Executor pred.    : On null augmented rows
                     O.SALESREP IS NULL
  Pred. selectivity : Expect to select 1.2346% of rows from table
  Table selectivity : Expect to select 11.1111% of rows from table
  Expected row count: 1 row after the join
  Operation cost    : 1
```

Example 6-39. EXPLAIN Plan for Left Join Not Transformed Into an Inner Join (page 2 of 2)

```
  Operation 2.1  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Current table
    Purpose           : To hash its rows before the Join
    Hash key columns  : O.SALESREP
    Hash cost         : 1

  Operation 2.2  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Result of a Select
    Purpose           : To form groups of rows for a Group By
    Hash key columns  : E.LAST_NAME
    Expected row count: 1 after the group by
    Hash cost         : 1

  Total cost        : 6
```

The IS NULL predicate selects only the special null-augmented rows generated by the left join operator. In this query, SQL must keep the null-augmented rows, so the left join is necessary, and Query Rewrite does not change the left join to an inner join.

Plan step 2 shows the left join. SQL uses a hash join in this step.

The expected row count for the scan on the EMPLOYEE table is 57. For the scan on the ORDERS table it is 1. The total cost is 6. Compare this EXPLAIN plan to the one in Example 6-40 on page 6-67.

Left Join Transformed Into an Inner Join

Example 6-40 on page 6-67 uses the view created for Example 6-39 on page 6-65:

```
CREATE VIEW EMPORD AS
SELECT *
    FROM EMPLOYEE E LEFT JOIN ORDERS O
      ON E.EMPNUM = O.SALESREP ;
```

Instead of the IS NULL predicate specified in the query for Example 6-39 on page 6-65, this query specifies an IS NOT NULL predicate:

```
EXPLAIN
  SELECT DISTINCT LAST_NAME
    FROM EMPORD
   WHERE SALESREP IS NOT NULL ;
```

Example 6-40 on page 6-67 shows the EXPLAIN plan.

Example 6-40. EXPLAIN Plan for Left Join Transformed Into an Inner Join

```
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  Query plan 1
  SQL request    : Select
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

 ---------------------------------------------------------------------------
Plan step 1
  ---------------------------------------------------------------------------

  Operation 1.0  : Scan
    Table             : \SQL1.$DATA8.SALES.ORDERS
                        with correlation name O
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 1 out of 5 columns

      Access path 1     : Alternate \SQL1.$DATA8.SALES.XORDREP, index only
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from index
      Index pred.       : None
      Base table pred.  : None
    Executor pred.    : None
    Table selectivity : Expect to select 100% of rows from table
    Expected row count: 13 rows after the scan
    Operation cost    : 1
 ----------------------------------------------------------------------------
  Plan step 2    : Perform an Inner Join
  Join strategy  : Key-Sequenced Merge Join
  Characteristic : Joins a row resulting from plan step 1
  ---------------------------------------------------------------------------

  Operation 2.0  : Scan
    Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                        with correlation name E
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 2 out of 6 columns

      Access path 1     : Primary, unique
      SBB for reads     : Virtual
      Begin key pred.   : O.SALESREP = E.EMPNUM
      End key pred.     : None
      Index selectivity : Expect to examine 1.7544% of rows from table
      Index pred.       : None
      Base table pred.  : None

    Executor pred.    : None
    Table selectivity : Expect to select 1.7544% of rows from table
    Expected row count: 13 rows after the join
    Operation cost    : 1
   Operation 2.1  : Hash
    Requested         : By the optimizer
    Hash rows in the  : Result of a Select
    Purpose           : To form groups of rows for a Group By
    Hash key columns  : E.LAST_NAME
    Expected row count: 13 rows after the group by
    Hash cost         : 1

  Total cost        : 4
```

The IS NOT NULL predicate in the query eliminates the special null-augmented rows produced by the left join operator. The search condition retains only the joined rows that are the same as those that result for an inner join. No performance gain results if SQL generates null-augmented rows that will be discarded, so Query Rewrite changes the left join to an inner join.

The EXPLAIN plan shows that plan step 2 is an inner join. The left join allows the SALESREP column to contain nulls, so the NOT NULL predicate is unnecessary. Therefore, Query Rewrite eliminates it. Without the left join, SQL can consider using access plans that use ORDERS as the outer table (processed in plan step 1). As a result, SQL uses a key-sequenced merge join.

The expected row count for the scan on the ORDERS table is 13. The expected row counts for the scan on the EMPLOYEE table and for the hash operation are both 13.

The total expected row counts for the EXPLAIN plan in Example 6-39 on page 6-65 is 58 (57 plus 1) and the total cost is 6. In Example 6-40 on page 6-67, the total expected row count is only 39 (13 times 3) and the total cost is 4.

EXPLAIN Plan for UNION Operation

This example shows an EXPLAIN plan for a query that includes a UNION of two SELECT statements.

The statements retrieve data from the ordone and ordtwo tables. Both tables have a system-defined primary key (SYSKEY).

The query follows:

```
EXPLAIN
 SELECT * FROM ordone
  UNION
 SELECT * FROM ordtwo ;
```

The total cost of the query is 3.

Example 6-41. EXPLAIN Plan for UNION Operation (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Union of Selects
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1    : perform a Union
---------------------------------------------------------------------------

Operation 1.0  : Union of plan step 2 and plan step 3

Operation 1.1  : Sort
  Requested         : Explicitly in the query
  Sort rows in the  : Result of a Union
  Purpose           : To discard duplicate rows for a Distinct
  Sort technique    : FASTSORT
  Sort type         : Plan to use User Process Sort
  UPS workspace     : 24 Kbytes
  Sort key columns  : ORDONE.A asc
  Sort cost         : 1
---------------------------------------------------------------------------
Plan step 2
Characteristic : Executes once before plan step 1
---------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA7.REG1.ORD1
                      with correlation name ORDONE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 2 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.  : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 3 rows after the scan
  Operation cost    : 1
---------------------------------------------------------------------------
Plan step 3
Characteristic : Executes once before plan step 1
---------------------------------------------------------------------------

Operation 3.0  : Scan
  Table             : \SQL1.$DATA7.REG1.ORD2
                      with correlation name ORDTWO
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 2 columns
    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None
```

Example 6-41. EXPLAIN Plan for UNION Operation (page 2 of 2)

```
    Executor pred.    : None
    Table selectivity : Expect to select 100% of rows from table
    Expected row count: 1 row after the scan
    Operation cost    : 1

  Total cost        : 3
```

Plan step 1 is a union of the result of plan steps 2 and 3. Plan step 1 executes after plan steps 2 and 3.

Plan step 1 involves a sort operation to eliminate duplicate rows in the union result. (Unless you specify UNION ALL, duplicate rows are automatically discarded from a union result.)

Plan step 2 is a scan of the ordone table.

Plan step 3 is a scan of the ordtwo table.

EXPLAIN Plan for MAX Optimization

The EXPLAIN plan chooses MAX optimization for a query containing the MAX aggregate function.

An index is created on the EMPLOYEE table:

```
CREATE INDEX SAL ON EMPLOYEE (SALARY,LAST_NAME,FIRST_NAME);

EXPLAIN
 SELECT MAX(SALARY)
   FROM EMPLOYEE
  WHERE LAST_NAME > "JONES"
    AND FIRST_NAME BETWEEN ?P1 AND ?P2;
```

The total cost of the query is 1.

Example 6-42. EXPLAIN Plan for MAX Optimization

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  Query plan 1
  SQL request    : Select
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

  ---------------------------------------------------------------------------
  Plan step 1
  ---------------------------------------------------------------------------

  Operation 1.0  : Scan
    Table              : \SQL1.$DATA8.PERSNL.EMPLOYEE
                         with correlation name EMPLOYEE
    Access type        : Record locks, stable access
    Lock mode          : Chosen by the system
    Column processing : Requires retrieval of 1 out of 6 columns

      Access path 1      : Alternate \SQL1.$DATA8.PUBS.SAL, index only
      SBB for reads      : Not used
      Begin key pred.    : None
      End key pred.      : None
      Index selectivity : Expect to examine 26.7686% of rows from index
      Index pred.        : Will be evaluated by the disk process
                           ( LAST_NAME > "JONES" ) AND ( FIRST_NAME <= ?P2 )
                           AND
                           ( FIRST_NAME >= ?P1 )
      Pred. selectivity : Expect to select 1.7544% of rows from index
      Base table pred.   : None

    Executor pred.     : None
    Executor aggr.     : Derived from the 1st row returned by the scan
                         MAX ( SALARY )
    Table selectivity : Expect to select 1.7544% of rows from table
    Expected row count: 4 rows after the scan
    Operation cost     : 1

  Total cost        : 1
```

The plan contains the following information:

- It consists of one step.
- Access is by alternate index:

```
Access path 1      : Alternate \SQL1.$DATA8.PUBS.SAL, index only
```

- The optimizer reads only one row from the table;

```
Executor aggr.    : Derived from the 1st row returned by the scan
                    MAX ( SALARY )
```

EXPLAIN Plan for Cursor UPDATE

This EXPLAIN plan shows an UPDATE operation using a cursor.

The query updates the EMPLOYEE table (primary key EMPNUM) according to data in the TABLES catalog table (primary key TABLENAME). The query follows:

```
EXPLAIN
 UPDATE EMPLOYEE
    SET JOBCODE = 105
  WHERE EXISTS ( SELECT TABLENAME
                   FROM TABLES
                  WHERE TABLETYPE = 'VI' );
```

The total cost of the query is 54.

Example 6-43. EXPLAIN Plan for Cursor UPDATE (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1   : Will utilize parallel execution
SQL request    : Update
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1

  Each operation is performed in parallel for this step

  Each ESP will read one of the following partitions:
    \SQL1.$DATA8
  The ESP's will be started in the cpu's numbered
    2

---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 0 out of 6 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None
    Type of Update    : Cursor
    SBB for Update    : Requested by the optimizer
```

Example 6-43. EXPLAIN Plan for Cursor UPDATE (page 2 of 2)

```
    Executor pred.    : On rows retrieved by the scan
                        EXISTS ( .. result of plan step 2 )
    Pred. selectivity : Expect to select 40% of rows from table
    Table selectivity : Expect to select 40% of rows from table
    Expected row count: 23 rows after the scan
    Operation cost    : 52

  ---------------------------------------------------------------------------
  Plan step 2
  Characteristic : Executes once before plan step 1
  ---------------------------------------------------------------------------

  Operation 2.0  : Scan
    Table             : \SQL1.$DATA8.PERSNL.TABLES
                        with correlation name TABLES
    Access type       : Record locks, stable access
    Lock mode         : Chosen by the system
    Column processing : Requires retrieval of 0 out of 12 columns

      Access path 1     : Primary
      SBB for reads     : Virtual
      Begin key pred.   : None
      End key pred.     : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.       : None
      Base table pred.  : Will be evaluated by the disk process
                          TABLETYPE = "VI"
      Pred. selectivity : Expect to select 50% of rows from table
    Executor pred.    : None
    Table selectivity : Expect to select 50% of rows from table
    Expected row count: 11 rows after the scan
    Operation cost    : 2

  Total cost        : 54
```

The plan consists of 2 steps:

- Plan step 1 is a scan of the EMPLOYEE table.
- Plan step 2 is a scan of the TABLES table. Plan step 2 executes once before plan step 1.

The type of update is a cursor update, as shown in operation 1.0. Virtual sequential block buffering is requested by the optimizer for the update operation.

EXPLAIN Plan for Cursor DELETE

The EXPLAIN plan shows a cursor DELETE operation.

The query deletes data from the EMPLOYEE table (primary key EMPNUM). The query follows:

```
EXPLAIN
 DELETE
   FROM EMPLOYEE
  WHERE DEPTNUM = 1500;
```

The total cost of the query is 1.

Example 6-44. EXPLAIN Plan for Cursor DELETE

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Delete
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 0 out of 6 columns

    Access path 1     : Alternate \SQL1.$DATA8.PERSNL.XEMPDEPT, index
                        only
    SBB for reads     : Virtual
    Begin key pred.   : DEPTNUM = 1500
    End key pred.     : DEPTNUM = 1500
    Index selectivity : Expect to examine 9.0909% of rows from index
    Index pred.       : None
    Base table pred.  : None
    Type of Delete    : Cursor

  Executor pred.    : None
  Table selectivity : Expect to select 9.0909% of rows from table
  Expected row count: 11 rows after the scan
  Operation cost    : 1

Total cost        : 1
```

The plan contains the following information:

- The lock granularity is record (row):

  ```
  Access type       : Record locks, stable access
  ```

- The type of delete is a cursor delete.

EXPLAIN Plan for INSERT

The EXPLAIN plan shows an INSERT operation.

The query inserts data into the ordtwo table, as follows:

```
EXPLAIN
 INSERT INTO ordtwo VALUES (100) ;
```

The total cost of the query is 1.

Example 6-45 on page 6-75 shows the EXPLAIN plan for the query.

Example 6-45. EXPLAIN Plan for INSERT Statement

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Insert
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Insert
  Table              : \SQL1.$DATA7.REG1.ORD2
  Access path        : Primary
  SBB for Insert     : Requested by the optimizer
  Insert Cost        : 1
  Operation cost     : 1

Total cost        : 1
```

EXPLAIN Plan for INSERT-SELECT

This example shows the EXPLAIN plan for an INSERT-SELECT operation.

The query selects data from the ordone table and inserts it into the ordtwo table, as follows:

```
EXPLAIN
 INSERT INTO ordone SELECT * FROM ordtwo ;
```

The total cost of the query is 2.

Example 6-46. EXPLAIN Plan for INSERT With SELECT

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Insert-Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA7.REG1.ORD2
                      with correlation name ORDTWO
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 2 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 1

Operation 1.1  : Insert
  Table             : \SQL1.$DATA7.REG1.ORD1
  Access path       : Primary
  SBB for Insert    : Requested by the optimizer
  Insert Cost       : 1

Total cost        : 2
```

If you are using INSERT-SELECT to populate a table from another table, be aware that the LOAD command performs this task more efficiently. The next most efficient way is to use an INSERT-SELECT statement on tables that are audited with sequential block buffering on. If the output table is unaudited and uses sequential block buffering, the performance decreases measurably.

The LOAD command and the INSERT-SELECT statement differ in the ways they write to the target table. The LOAD command writes in blocks and the INSERT-SELECT statement writes one row at a time.

For more information on the LOAD command and the INSERT-SELECT statement, see the *SQL/MP Reference Manual.*

EXPLAIN Plan for UPDATE

This example shows the EXPLAIN plan for an UPDATE operation. The query updates the EMPLOYEE table as follows:

```
EXPLAIN
 UPDATE employee SET deptnum = 5
  WHERE empnum = ?parm3;
```

The total cost of the query is 1.

Example 6-47. EXPLAIN Plan for Unique UPDATE

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Update
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.PERSNL.EMPLOYEE
                      with correlation name EMPLOYEE
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 0 out of 6 columns

    Access path 1     : Primary, unique
    SBB for reads     : Not used
    Begin key pred.   : EMPNUM = ?PARM3
    End key pred.     : EMPNUM = ?PARM3
    Index selectivity : Expect to examine 1.7544% of rows from table
    Index pred.       : None
    Base table pred.  : None
    Type of Update    : Unique
    SBB for Update    : Not used

  Executor pred.    : None
  Table selectivity : Expect to select 1.7544% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 1

Total cost        : 1
```

EXPLAIN Plan With Date-Time Values

These examples show EXPLAIN plans for an UPDATE operation.

If no end-date-time is provided for an INTERVAL data type and is implied for a start-date-time, SQL expands the original syntax of the query to show the implied end-date-time.

HOUR Date-Time Values

Example 6-48 on page 6-79 shows HOUR(2) expanded to HOUR. The query used in this example is as follows.

```
INVOKE B2UNS01 ;
 Definition of table \SQL1.$DATA5.SQLDOPTS.B2UNS01
 Definition current at 09:08:07 - 09/11/95

  (
    CHAR0_N10                      CHAR(2) DEFAULT "AD"
                                   HEADING 'char0_n10 with default ''AD'''
  , SBIN0_UNIQ                     SMALLINT DEFAULT SYSTEM NOT NULL
  , SDEC0_N500                     DECIMAL( 18, 0 ) DEFAULT SYSTEM
  , DATE0_UNIQ                     DATETIME YEAR TO DAY NO DEFAULT NOT NULL
  , INT0_YTOM_NUNIQ                INTERVAL YEAR(5) TO MONTH NO DEFAULT
  , INT1_HTOS_1000                 INTERVAL HOUR(2) TO SECOND DEFAULT SYSTEM
                                   NOT NULL
  , DATE1_N4                       DATETIME YEAR TO DAY DEFAULT SYSTEM
  , REAL1_UNIQ                     FLOAT(22) NO DEFAULT NOT NULL
  , UBIN1_N2                       NUMERIC( 4, 0) UNSIGNED NO DEFAULT
  , UDEC1_100                      DECIMAL( 2, 0 ) UNSIGNED DEFAULT SYSTEM
                                   NOT NULL
  )
EXPLAIN
 SELECT INT1_HTOS_1000 FROM B2UNS01
   WHERE INT1_HTOS_1000 = INTERVAL '0' HOUR(2) ;
```

Example 6-48. EXPLAIN Plan With HOUR Date-Time Values

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA5.SQLDOPTS.B2UNS01
                      with correlation name B2UNS01
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 11 columns

    Access path 1     : Primary, sequential cache
    SBB for reads     : Not used
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        INT1_HTOS_1000 = INTERVAL ' 00' HOUR ( 2 ) TO HOUR
    Pred. selectivity : Expect to select 0.1% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 0.1% of rows from table
  Expected row count: 2 rows after the scan
  Operation cost    : 272

Total cost        : 272
```

DAY Date-Time Values

Example 6-49 on page 6-80 shows DAY(3) expanded to DAY. The query used in this example is as follows:

```
INVOKE B2UNL13 ;
 Definition of table \SQL1.$DATA5.SQLDOPTS.B2UNL13
 Definition current at 09:08:10 - 09/11/95

  (
    DATE0_N100                     DATETIME YEAR TO DAY DEFAULT NULL
  , SBIN0_4                        SMALLINT DEFAULT SYSTEM NOT NULL
  , SDEC0_N100                     DECIMAL( 2, 0 ) DEFAULT SYSTEM
  , INT0_DTOF6_UNIQ                INTERVAL DAY(2) TO FRACTION(6) NO DEFAULT
                                   NOT NULL
  , VARCHAR0_N1000                 VARCHAR(8) NO DEFAULT
  , UDEC1_10P                      DECIMAL( 9, 0 ) UNSIGNED DEFAULT SYSTEM
                                   NOT NULL
  , REAL1_N100                     FLOAT(22) DEFAULT SYSTEM
  , UBIN1_UNIQ                     NUMERIC( 9, 0) UNSIGNED NO DEFAULT NOT
NULL
  , TS1_NUNIQ                      DATETIME YEAR TO FRACTION(6) NO DEFAULT
  , INT1_YTOM_100                  INTERVAL YEAR(2) TO MONTH DEFAULT SYSTEM
                                   NOT NULL
  )
EXPLAIN
 SELECT
    INT0_DTOF6_UNIQ FROM B2UNL13
```

```
   WHERE INT0_DTOF6_UNIQ >= INTERVAL '0' DAY(3) AND
   INT0_DTOF6_UNIQ <= INTERVAL '2' DAY(3) ;
```

Example 6-49. EXPLAIN Plan With DAY Date-Time Values

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA5.SQLDOPTS.B2UNL13
                      with correlation name B2UNL13
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 10 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : INT0_DTOF6_UNIQ >= INTERVAL '  00' DAY ( 3 ) TO DAY
    End key pred.     : INT0_DTOF6_UNIQ <= INTERVAL '  02' DAY ( 3 ) TO DAY
    Index selectivity : Expect to examine 0.0012% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Table selectivity : Expect to select 0.0012% of rows from table
  Expected row count: 2 rows after the scan
  Operation cost    : 3

Total cost        : 3
```

Comparing Cost: A Scenario

The next two query examples show how you can reformulate a query and produce the same result but with much improved performance. The DISPLAY STATISTICS and EXPLAIN plans show you the results.

The first formulation has an estimated cost of 50. The second formulation has an estimated cost of 4.

First Formulation

This statement prepares the first formulation of the query:

```
PREPARE QUERY1 FROM
 SELECT DISTINCT ORDERNUM
 FROM ODETAIL O, PARTS P
 WHERE O.PARTNUM = 5100
 AND QTY_ORDERED <
 (SELECT AVG(QTY_AVAILABLE)
 FROM PARTS
 WHERE P.PARTNUM = 5100) ;
```

After executing the query, use the DISPLAY STATISTICS command to display the statistics as shown in Example 6-50 The estimated cost is 50. The ODETAIL table is scanned once. The PARTS table is scanned twice.

Example 6-50. DISPLAY STATISTICS Output for QUERY1

```
Estimated Cost          50

Start Time          95/09/11 09:08:13.871517
End Time            95/09/11 09:08:14.546697
Elapsed Time                 00:00:00.675180
SQL Execution Time           00:00:00.229874

                        Records     Records        Disk     Message   Message Lock
Table Name             Accessed        Used       Reads       Count       Bytes
\SQL1.$DATA8.SALES.ODETAIL
                             72           4           2           3         490
\SQL1.$DATA8.SALES.PARTS
                            112         112           2           8        3792
\SQL1.$DATA8.SALES.PARTS
                           3136         112           2         336       90048
```

By examining the EXPLAIN plan shown in Example 6-51 on page 6-82, you can better understand how the cost is determined.

Example 6-51. EXPLAIN Plan for QUERY1 (page 1 of 2)

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Query plan 1
SQL request    : Select
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

---------------------------------------------------------------------------
Plan step 1
---------------------------------------------------------------------------

Operation 1.0  : Scan
  Table             : \SQL1.$DATA8.SALES.ODETAIL
                      with correlation name O
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 2 out of 4 columns

    Access path 1     : Primary
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        O.PARTNUM = 5100
    Pred. selectivity : Expect to select 3.7037% of rows from table

  Executor pred.    : None
  Table selectivity : Expect to select 3.7037% of rows from table
  Expected row count: 3 rows after the scan
  Operation cost    : 3

---------------------------------------------------------------------------
Plan step 2    : Perform an Inner Join
Join strategy  : Nested Join
Characteristic : Joins a row resulting from plan step 1
---------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA8.SALES.PARTS
                      with correlation name P
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 4 columns

    Access path 1     : Alternate \SQL1.$DATA8.SALES.XPARTDES, index only
    SBB for reads     : Virtual
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from index
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : On rows retrieved by the scan
                      QTY_ORDERED < AVG ( QTY_AVAILABLE ) .. result of plan
                      step 3
  Pred. selectivity : Expect to select 33.33% of rows from table
  Table selectivity : Expect to select 33.33% of rows from table
  Expected row count: 25 rows after the join
  Operation cost    : 2
```

Example 6-51. EXPLAIN Plan for QUERY1 (page 2 of 2)

```
---------------------------------------------------------------------------
Plan step 3
Characteristic : Executes once per row retrieved in plan step 2
---------------------------------------------------------------------------

Operation 3.0  : Scan
  Table             : \SQL1.$DATA8.SALES.PARTS
                      with correlation name PARTS
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 4 columns

    Access path 1     : Primary
    SBB for reads     : Not used
    Begin key pred.   : None
    End key pred.     : None
    Index selectivity : Expect to examine 100% of rows from table
    Index pred.       : None
    Base table pred.  : Will be evaluated by the disk process
                        P.PARTNUM = 5100
    Pred. selectivity : Expect to select 100% of rows from table

  Executor pred.    : None
  DP2 aggregate     : Computed for each group
                      AVG ( QTY_AVAILABLE )
  Table selectivity : Expect to select 100% of rows from table
  Expected row count: 28 rows after the scan
  Operation cost    : 2

Total cost        : 50
```

The plan consists of three steps: a scan of the ODETAIL table, a join of the PARTS and ODETAIL tables, and a scan of the PARTS table.

The PARTS table is scanned twice, because the FROM clause of QUERY 1 specifies an implicit join operation:

```
FROM ODETAIL O, PARTS P
```

In step 3, the optimizer chooses aggregate evaluation by the disk process (DP2), which is the most efficient aggregate evaluation method.

Second Formulation

Now suppose that you reformulate the query to eliminate the unnecessary join as follows:

```
PREPARE QUERY2 FROM
 SELECT ORDERNUM
 FROM ODETAIL O
 WHERE O.PARTNUM = 5100
 AND QTY_ORDERED <
 (SELECT AVG(QTY_AVAILABLE)
 FROM PARTS P
 WHERE P.PARTNUM = 5100) ;
```

After executing the query, use the DISPLAY STATISTICS command to display the statistics shown in Example 6-52 on page 6-84.

Example 6-52. DISPLAY STATISTICS Output for QUERY2

```
Estimated Cost             4

Start Time            95/09/11 09:08:17.871610
End Time              95/09/11 09:08:18.010648
Elapsed Time                   00:00:00.139038
SQL Execution Time             00:00:00.016799

                        Records     Records      Disk    Message   Message Lock
Table Name             Accessed        Used     Reads      Count      Bytes
\SQL1.$DATA8.SALES.PARTS
                              1           1         0          2        188
\SQL1.$DATA8.SALES.ODETAIL
                             72           4         0          5        744
```

By reformulating the query to remove the unnecessary join operation, you have reduced the estimated cost of the query from 50 to 4.

The EXPLAIN plan in Example 6-53, shows that the plan now consists of two steps: a scan of the ODETAIL table and a scan of the PARTS table to satisfy the subquery.

Example 6-53. EXPLAIN Plan for QUERY2 (page 1 of 2)

```
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  Query plan 1
  SQL request    : Select
  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

  ---------------------------------------------------------------------------
  Plan step 1
  ---------------------------------------------------------------------------

  Operation 1.0  : Scan
    Table              : \SQL1.$DATA8.SALES.ODETAIL
                         with correlation name O
    Access type        : Record locks, stable access
    Lock mode          : Chosen by the system
    Column processing : Requires retrieval of 1 out of 4 columns

      Access path 1      : Primary
      SBB for reads      : Not used
      Begin key pred.    : None
      End key pred.      : None
      Index selectivity : Expect to examine 100% of rows from table
      Index pred.        : None
      Base table pred.   : Will be evaluated by the disk process
                           ( O.PARTNUM = 5100 ) AND ( QTY_ORDERED < AVG (
                           QTY_AVAILABLE ) .. result of plan step 2 )
      Pred. selectivity : Expect to select 1.3889% of rows from table

    Executor pred.     : None
    Table selectivity : Expect to select 1.3889% of rows from table
    Expected row count: 1 row after the scan
    Operation cost     : 3
```

Example 6-53. EXPLAIN Plan for QUERY2 (page 2 of 2)

```
---------------------------------------------------------------------------
Plan step 2
Characteristic : Executes once before plan step 1
---------------------------------------------------------------------------

Operation 2.0  : Scan
  Table             : \SQL1.$DATA8.SALES.PARTS
                      with correlation name P
  Access type       : Record locks, stable access
  Lock mode         : Chosen by the system
  Column processing : Requires retrieval of 1 out of 4 columns

    Access path 1     : Primary, unique
    SBB for reads     : Not used
    Begin key pred.   : P.PARTNUM = 5100
    End key pred.     : P.PARTNUM = 5100
    Index selectivity : Expect to examine 3.5714% of rows from table
    Index pred.       : None
    Base table pred.  : None

  Executor pred.    : None
  Executor aggr.    : Computed for each group
                      AVG ( QTY_AVAILABLE )
  Table selectivity : Expect to select 3.5714% of rows from table
  Expected row count: 1 row after the scan
  Operation cost    : 1

Total cost        : 4
```

Index