

# Query processing using HP NonStop SQL/MP software



Introduction .....	2
Query processing components .....	2
Query optimization .....	3
Generating the plan .....	3
Serial execution plans .....	4
Parallel execution plans .....	4
Multiple index-access plans .....	4
Selecting the best plan .....	4
Query processing performance enhancements .....	5
Set-oriented operations in the data access manager .....	5
Horizontal partitioning .....	5
Combination and elimination of sorts .....	5
Hash joins .....	6
Parallel execution .....	6
Parallelism without repartitioning .....	7
Dynamic repartitioning .....	7
Parallel join execution plans .....	7
Grouping and aggregation .....	8
Flexible options for the application developer .....	10
Demonstrated performance improvements .....	10
Scalability .....	10
Efficient algorithms .....	10
Unmatched performance for query processing .....	10
Glossary .....	11
References and additional reading .....	12
For more information .....	12

## Introduction

The HP NonStop SQL/MP relational database management system (RDBMS) is a truly parallel implementation of the structured query language (SQL) that is well suited to meet the demands of critical commercial applications, including online transaction processing (OLTP), batch operations, and query processing.

NonStop SQL/MP software dramatically improves query processing by providing

- Parallelism, or the ability to divide single SQL tasks into multiple subtasks that can be processed in parallel
- Scalability, or the ability to speed up queries for a database of constant size or to maintain the same response time for a growing database by adding processors
- Reliability, the combination of 24-hour-a-day, 365-day-a-year availability and full data integrity
- High performance, with advanced features such as total disk encapsulation, parallel execution, and hash joins

This paper explains how NonStop SQL/MP software optimizes query processing on parallel servers. It describes the software components, the process of query optimization, and the features that allow NonStop SQL/MP to process large amounts of data and complex queries quickly.

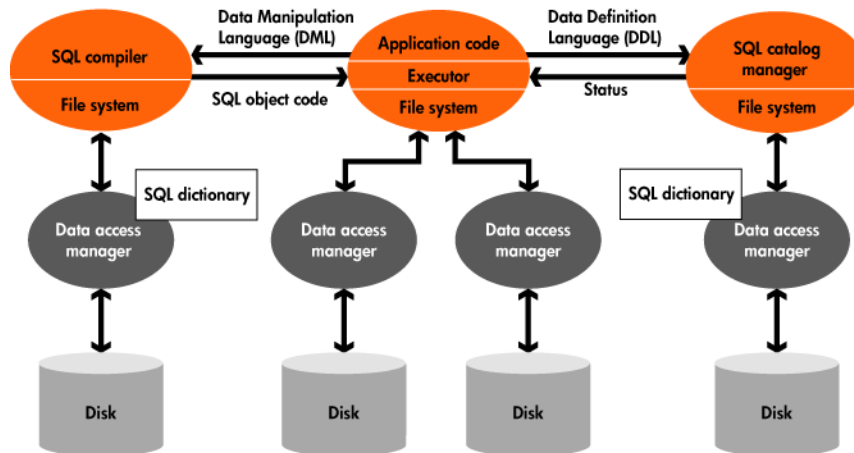
## Query processing components

The NonStop SQL/MP architecture optimizes performance in two ways. First, it takes advantage of the parallel, distributed architecture of the NonStop server. Second, it speeds query execution by placing database operations (for example, scans and joins) at the lowest level of the NonStop server architecture.

The query processing system of NonStop SQL/MP software consists of the following elements (see figure 1):

- The *SQL compiler* compiles dynamic SQL statements and an execution plan (also known as an *access plan*) for each statement, referencing the SQL dictionary for information about table contents. Unlike compilers in other RDBMSs, the NonStop SQL/MP SQL compiler stores the execution plans for embedded SQL statements in the same file as the object code produced by the host-language (3GL) compiler. This reduces the number of files that NonStop SQL/MP needs to read from the database, minimizing bottlenecks.
- The *SQL executor* is a set of system library procedures that application programs invoke to execute an SQL statement.
- The *SQL file system* enables the SQL executor to access a specific table via a specified access path. The file system sends a message to the appropriate data access manager to retrieve the data.
- *Data access managers* control access to objects on a specific disk. To minimize overhead, all disk access is handled exclusively by the data access manager through a technique called *total disk encapsulation*, which eliminates the need for a distributed lock manager. During an SQL operation, the data access manager returns to the SQL executor only the rows and columns needed to complete the SQL request, for more efficient operation.
- The active *SQL dictionary* enables users to add data definitions, such as columns and indexes, without modifying existing applications. If any changes are made to the database, SQL execution plans are automatically recompiled using the up-to-date definition.

**Figure 1.** The NonStop SQL/MP query processing architecture.



## Query optimization

Query optimization is the process of evaluating plans for executing a query and selecting the one with the lowest cost. The NonStop SQL/MP query optimizer enumerates all execution plans for evaluating a query, including serial, parallel, and multiple index-access plans.

When evaluating plans, NonStop SQL/MP software measures cost by the system resources consumed by the operation—that is, the number of processor instructions, input/output (I/O) operations, messages, and so on—expressed as the number of equivalent I/O operations. Cost measurements account for remote data access and the time it takes to perform sorts.

## Generating the plan

For each table, the NonStop SQL/MP optimizer generates one plan using the primary access path, as well as one plan using each alternate access path. Each of the single-table execution plans provides the basis for generating join plans.

The optimizer builds a tree of access paths, called the *search tree*, for enumerating join plans. Each node in the search tree represents an access path for a set of tables joined together. The path is characterized by

- The set of tables it contains
- An access path for reading its rows
- A set of predicates for realizing it
- An order for the rows it contains
- Its cardinality—that is, the number of rows

Order is significant. For example, a scan on a table that retrieves rows through an index yields rows in a particular order. The order determines whether a subsequent sort (for an ORDER BY or a GROUP BY clause) can be eliminated, a predicate can be used in an index key, or a sort-merge join should be performed.

The optimizer may prune the search tree whenever it adds a new composite. When it receives two search trees with the same characterization, the optimizer retains the one with the lower cost.

After building the search tree, the optimizer generates serial or parallel execution plans.

### **Serial execution plans**

Given two tables, the optimizer generates every nested-loops join plan possible. It generates sort-merge join and hash join plans only when the two tables have one or more equi-join predicates relating them. The optimizer generates two sort-merge join plans. The first uses the composite with the lowest cost and adds the cost of sorting, so that the composite and the table yield rows in the same order as the merge phase. The second uses the composite, if one exists, that yields rows in the same order as the table. Additionally, it generates two hash join plans: a simple and a hybrid. The hash join methods are described in the “Hash joins” section.

### **Parallel execution plans**

NonStop SQL/MP software improves the response time of decision support applications by dividing the workload over multiple processors—performing selects, inserts, updates, deletes, joins, grouping, and aggregation in parallel. (For a comprehensive discussion of parallel execution plans, see the “Parallel execution” section.) The optimizer generates parallel execution plans whenever an application selects this option.

To determine the cost of executing a query in parallel, the optimizer

- Computes the number of horizontal partitions of a (composite) table that will be operated on
- Computes the cost of the parallel execution plan
- Assigns a fixed startup cost for starting the executor server processes and the communications costs necessary for initializing the execution environment for each operation
- Amortizes the cost of executing the operation in serial over each partition, resulting in the cost of performing the operation per partition
- Adds the startup costs and the costs for performing one or more operations per partition, determining the cost of the total plan

### **Multiple index–access plans**

The optimizer evaluates a multiple index–access plan for a single-table query when the WHERE clause is represented in the form “A or B.” In a multiple index–access plan, an index that belongs to the table is scanned for each part of the OR predicate, and a logical union is performed on the rows retrieved. This plan competes with other plans based on its cost.

## **Selecting the best plan**

The NonStop SQL/MP optimizer considers the execution plan with the lowest cost the best plan, whether that plan is serial or parallel. However, when the costs of two plans differ by 10 percent or less, the optimizer prefers

- A matching partitions plan over any other parallel execution plan (see “Parallel join execution plans”)
- A parallel execution plan over the hash-repartitioned plan (see “Parallel join execution plans”)
- A local access path over a remote one
- A plan that has an index as an access path when each key column of the index has an equality predicate specified
- A plan with a lower index selectivity
- A plan that employs more predicates on key columns of an index
- A plan that uses a unique index
- A plan that accesses the base table directly

## Query processing performance enhancements

NonStop SQL/MP software achieves unmatched speed for large and complex queries through features such as set-oriented access, horizontal partitioning, elimination of sorts, hash joins, parallel execution, and efficient grouping and aggregation.

### Set-oriented operations in the data access manager

The data access manager supports interfaces that provide single-table, set-oriented access to tables. A set is defined by a restriction expression, such as “WHERE COLUMN = VALUE”, and a list of columns to project. A request (message) from the file system describes the data to be retrieved in terms of a set—that is, the begin and end keys, the restriction expression, and the attributes to be projected. In response, the data access manager can reply with one or more buffers containing the data that corresponds to the set.

NonStop SQL/MP software optimizes set-oriented operations in the following ways:

- *Set-oriented updates.* It sends the update expression—as well as the begin and end keys and the predicates that define the set—to the data access manager.
- *Sequential inserts.* It buffers the inserts in a file system buffer.
- *Set-oriented deletes.* The file system sends the begin and end keys and the predicates (which define the set to be deleted) to the data access manager in a single message.

If the data in certain columns of a table is ordered according to the GROUP BY clause, then grouping and aggregation are subcontracted to the data access manager.

The data access manager also supports prefetch of multiple blocks and postwrites of updated blocks. These features reduce the number of messages between the application (file system) and the data access manager.

### Horizontal partitioning

A table or an index can be horizontally partitioned if you specify the key ranges of each partition. The partitioned table appears as a single table; the file system redirects any SQL SELECT, INSERT, DELETE, or UPDATE requests to the proper partition according to the key range specified in the statement. To improve the performance of queries by reducing the number of file-open operations, users can instruct the system to open partitions only when they are needed.

### Combination and elimination of sorts

A sort may be performed either to eliminate duplicate values or to order the data. The optimizer considers the following as a request to sort data: a SELECT DISTINCT, an aggregate function that contains a DISTINCT, a GROUP BY clause, or an ORDER BY clause.

Since sorting is one of the most time-consuming operations in evaluating a query, the optimizer attempts to reduce redundant sorts by combining or eliminating them. The optimizer combines sorts in the following situations:

- The select list of a SELECT DISTINCT statement is a subset of the ordering columns of the ORDER BY clause or the grouping columns of the GROUP BY clause.
- Grouping columns of the GROUP BY clause form a subset of the select list of a SELECT DISTINCT statement.
- Grouping columns of the GROUP BY clause form a subset of the ordering columns of the ORDER BY clause—that is, the grouping columns form the leading prefix of the ordering columns.

- Ordering columns of the ORDER BY clause form a subset of the select list of a SELECT DISTINCT statement or the grouping columns of a GROUP BY clause.

The optimizer eliminates a sort when it can use an index that meets one of two criteria:

- It provides data in the order required by a GROUP BY or an ORDER BY clause.
- The grouping columns or the entire select list for a SELECT DISTINCT clause are the columns of a unique index.

## Hash joins

Numerous prototype implementations and academic papers have shown that hash join algorithms<sup>1,2</sup> provide the best performance when sufficient main memory is available and there is no index that localizes the number of probes on the inner table. Sufficient memory is usually defined as the square root of the inner-table size, in blocks.

To take advantage of the large amount of main memory common in modern servers, NonStop SQL/MP software supports hash join algorithms in addition to the traditional nested-loops and sort-merge join algorithms. The Adaptive Hash Join algorithm used in NonStop SQL/MP<sup>3</sup> is a variation of the well-known Hybrid Hash Join developed in the GAMMA parallel database machine project of the University of Wisconsin.<sup>4</sup>

The unique benefit of the Adaptive Hash Join algorithm is its robust ability to overcome estimation errors relating to the size of the input relation and the amount of memory available for the join. These values are required to determine the cost of different join algorithms. If the actual values for the input-relation size and memory size differ from the estimates, nested-loops joins and merge joins perform as well as they do ordinarily; their execution plans are not dependent on these parameters. In contrast, other implementations of the hash join may execute more slowly or even fail because parameters such as the number of hash classes and the amount of information maintained in main memory are predetermined. To minimize this risk, the Adaptive Hash Join algorithm uses the following methods, which deliver consistent performance over a large range of memory sizes and prevent failures:

- Variable-length I/O transfers to and from the disks
- Dynamic swapping of parts of the hash table to disk files
- Dynamic splits of “hash buckets” into smaller fractions
- Utilization of a “hash loops” algorithm (an algorithm with a cost of  $n \times m$ , where  $n$  and  $m$  are the table sizes) for cases of extreme overflow

These methods enable the Adaptive Hash Join algorithm to respond to changing system loads by adjusting its memory consumption at various points during its execution. Because of this robust behavior, the optimizer can choose a hash join even if the actual values for required parameters are not known in advance.

## Parallel execution

In NonStop SQL/MP, a query does not have to be formulated in a special way to be executed in parallel. The optimizer considers sequential as well as parallel execution plans, ascertains the cost for each plan, and then selects the plan with the lowest cost. This contrasts with the approach of other RDBMSs, which first generate the optimal sequential execution plan and then render the operations in

<sup>1</sup> D. DeWitt, R. Gerber, “Multiprocessor Hash-based Join Algorithms,” *Proc. VLDB*, 1985.

<sup>2</sup> D. Schneider, D. DeWitt, “A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment,” *Proc. ACM SIGMOD*, 1989.

<sup>3</sup> H. Zeller, J. Gray, “An Adaptive Hash Join Algorithm for Multiuser Environments,” *Proc. 16, VLDB*, 1990.

<sup>4</sup> DeWitt, Gerber, 1985.

parallel. The NonStop SQL/MP strategy yields better performance because the optimal sequential plan does not always yield the best parallel plan.

With NonStop SQL/MP software, the executor component in the application program can start a set of executor server processes (ESPs) to execute queries or parts of queries in parallel. Thus, there is a “master-slave” relationship between the application (the master executor) and the ESPs. Each ESP consists of a small main program, executor, and file system. Master and server executors communicate via messages. Figure 2 shows that the master executes requests using either the file system to which it is bound or the server executors.

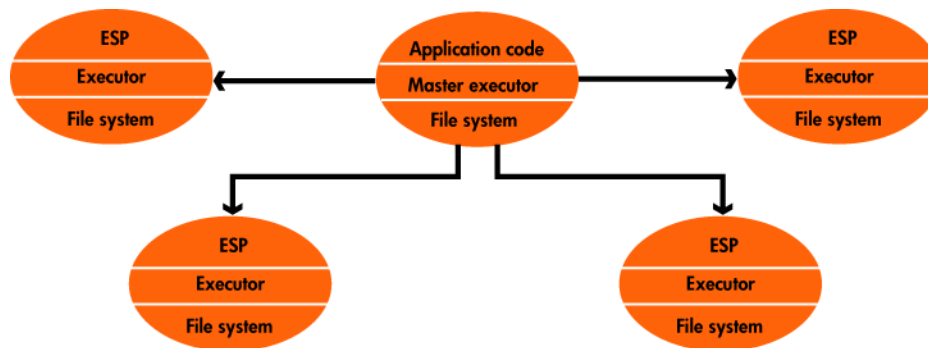
This architecture allows a broad variety of parallel execution plans. ESPs are used for executing operations for SQL SELECT, INSERT, DELETE, and UPDATE statements on partitioned files in parallel. However, if a table’s indices reside on different disks, then updates to a base table record and the corresponding index updates are performed in parallel by the file system. ESPs are shared between SQL statements in a process and are retained beyond transaction boundaries, further improving the efficiency of parallel processing.

### Parallelism without repartitioning

A parallel execution plan is a natural way of operating on a NonStop SQL/MP table that is horizontally partitioned. An index may be horizontally partitioned to avoid a bottleneck during processing of GROUP BY or ORDER BY clauses. For SQL SELECT, INSERT, DELETE, or UPDATE statements, two or more partitions can be processed in parallel.

The number of ESPs used for single-table queries is always equal to the number of partitions of the table. For aggregate queries, each ESP computes a local aggregate and returns its result to the master executor. The master computes the query result in the final step.

Figure 2. Executor server processes (ESPs).



### Dynamic repartitioning

NonStop SQL/MP software is able to perform all operations in parallel—including data-intensive operations such as equi-joins, duplicate elimination, or grouping— even when the tables are not partitioned. To do this, it reads a nonpartitioned table and uses a hash-based partitioning mechanism to build a temporary partitioned table.

### Parallel join execution plans

NonStop SQL/MP offers three parallel execution plans for performing joins: matching partitions, joining a partitioned table with any other table, and hash partitioning (see figure 3).

## Matching partitions

When two tables are partitioned so that the optimizer can join pairs of individual partitions in parallel, the optimizer considers this option. For example, consider a banking application that uses the following tables: *Account* (*Branch*, *AccountNo*, *Owner*, *CreditLimit*) and *Deposits* (*Branch*, *AccountNo*, *Date*, *Amount*).

If both tables are partitioned on the same key ranges in the *Branch* attribute, then the following join will find matching rows stored in corresponding partitions: *Account.Branch = Deposits.Branch AND Account.AccountNo = Deposits.AccountNo*. Therefore, it is possible to join each pair of “corresponding” partitions independently and in parallel. The optimizer recognizes this common case and generates the appropriate parallel execution plan. Designing the database so that it is possible to join matching partitions is a very common and simple means of achieving parallelism.

## Joining a partitioned table with any other table

To achieve parallel execution in this plan, the executor reads each partition of a partitioned table *R* in parallel. Each partition of *R* is subsequently joined with the entire table *S* in parallel. This plan uses either the nested-loops or the hash join method. The optimizer selects the nested-loops join only if an index access path exists for table *S*. For the query to perform best under parallelism, table *S* should be partitioned and have an index on the join columns, or the cost of reading table *S* should be much less than the cost of reading table *R*. If this plan performs a hash join, then the entire table *S* (after local selection predicates are applied) is sent to each ESP that processes a partition of the outer table *R*, a process called *fragment and replicate*. Each ESP then builds a hash table from *S* and performs a hash join. Hash joins work well for this plan when table *S* is relatively small. They are able to avoid contention on the inner table and to achieve a high degree of parallelism.

## Hash partitioning

Unlike the two parallel execution join plans described previously, a hash-partitioned join plan is not confined to a specific partitioning scheme of the base tables. It is applicable, however, for equi-joins only. Figure 3 shows two base tables, with one and two partitions respectively, joined by three parallel join processes. In general, an arbitrary number of join processes can be used. In the current version of NonStop SQL/MP software, the number of join processes is equal to the number of processors in the system.

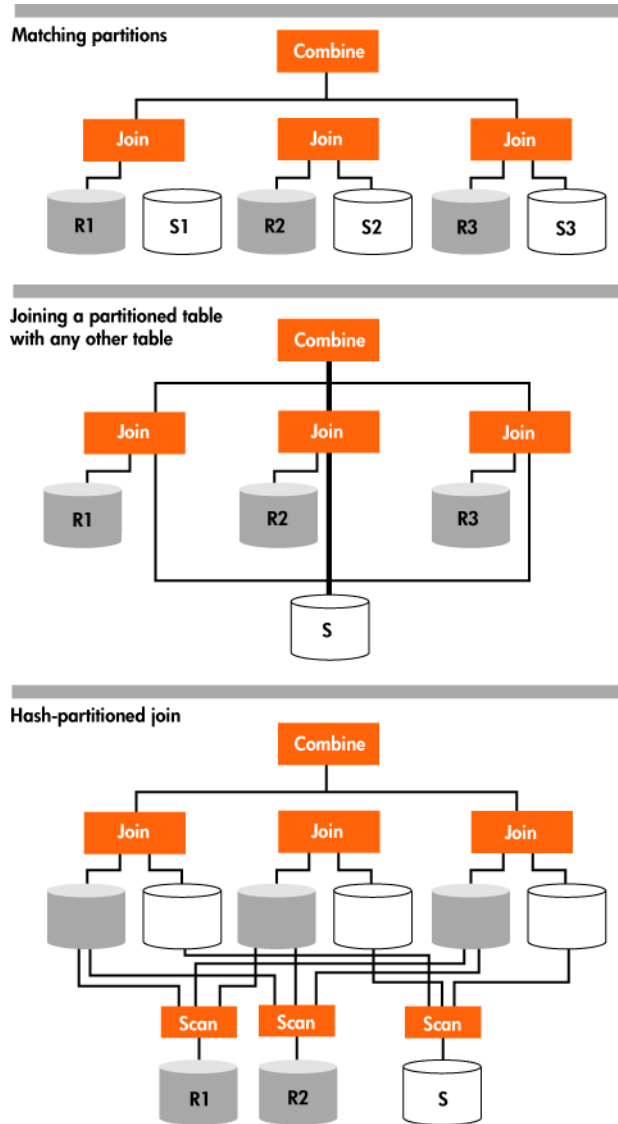
This join plan operates in two phases. In the first phase, each of the two tables to be joined is repartitioned as described in “Dynamic repartitioning.” To repartition the tables, the executor applies the same hash function to the values contained in the join columns from each table. This ensures that rows from the outer and inner tables that have matching join columns are stored in corresponding partitions. In the second phase, a set of ESPs joins the partitions built in the first phase: the same step performed in joining tables with matching partitions. Note that the hash-based partitioning algorithm ensures that matching rows are stored in corresponding partitions.

## Grouping and aggregation

Grouping and aggregation functions are very common in decision support queries, so improving their performance improves overall query processing. NonStop SQL/MP software uses the following techniques for efficient grouping and aggregation:

- *Reduction of network traffic.* The data access manager computes all SQL aggregate functions for a given single-variable query. The optimizer automatically chooses this strategy for single-variable queries and for multiple-variable queries when the aggregate function references the innermost table of a join sequence. This optimization reduces network traffic by reducing the number of messages exchanged between the SQL file system and the data access manager. It is also used for grouping if rows read by the data access manager are in grouping order—that is, whenever the grouping columns form the left prefix of an index key.

**Figure 3.** Parallel join execution plans.



- *Simplified computation of aggregates.* If the operand of the MIN or MAX aggregate function is a column that belongs to an index key and is the only item in the select list of a SELECT statement, NonStop SQL/MP computes the aggregate in a single access by positioning on either the first or the last row of a given key range for an index.
- *Reading of a single row, when possible.* If the operand of the MIN or MAX function is the first key column or if the user provides equality predicates on the key columns that precede the operand, then the scan reads the single row that contains the first or the last row for the user-prescribed key range. This row always contains the MIN or the MAX value (for the first or last row, respectively) for that key column.
- *Parallel grouping and aggregation.* Grouping and aggregation can be performed in parallel, as described in "Parallel execution."
- *Hash-based method.* For greater efficiency and speed, the executor supports new algorithms for performing grouping and aggregation using a hash-based method instead of sorting. If the grouped data can fit in main memory, then grouping is performed in a single pass over the raw data. If the

grouped data does not fit in main memory, then the raw data is written to disk using large transfers according to the hash value of the grouping columns. Subsequently, sections of the raw data are read sequentially according to their serial hash value. Grouping and aggregation are performed after the raw data is reread. This technique, used with both serial and parallel execution plans, ensures that the raw data is read no more than twice. When hash-based grouping is used by ESPs, each slave ESP hashes and groups data belonging to the local partition. After duplicates are eliminated, the resulting groups and partial aggregates are sent to the master executor process, which finalizes the result. This performance enhancement reduces internetwork traffic by applying grouping and aggregation locally.

## Flexible options for the application developer

By default, the optimizer selects the execution plan that produces the fastest total response time for a given SQL query. However, in many decision support applications, it is far more desirable to see a screen of rows rapidly than to wait longer to receive all screens at once. Therefore, NonStop SQL/MP allows application developers to direct the optimizer to choose an execution plan that is biased toward returning the first few rows quickly, usually by reading from an index. This means that the user need not wait for the entire query to finish before viewing the first row.

The application developer can also direct the execution plan to use a specific index, user-prescribed join sequence, or join method.

## Demonstrated performance improvements

As a result of its scalability and its use of more efficient algorithms, NonStop SQL/MP software demonstrates significant performance benefits in massively parallel environments.

### Scalability

For an RDBMS, scalability means speed-up and scale-up ability. Speed-up is a measure of how fast a parallel processing system solves a problem of a fixed size, while scale-up is a measure of how well a parallel processing system handles a growing database. NonStop SQL/MP software demonstrates nearly linear speed-up and scale-up<sup>5</sup>—that is, doubling the number of processors virtually doubles response time or maintains constant response time with a database of twice the size (see figure 4).

### Efficient algorithms

NonStop SQL/MP improves performance significantly by supporting more efficient algorithms for routine query processing, as shown in the following examples:

- The hash join method of query processing can improve performance by 300 to 400 percent over nested-loops or sort-merge joins.
- When grouping and aggregation are performed using a hash-based method instead of a sort-based one, the Wisconsin benchmark queries show a 60 percent performance improvement.
- When aggregation is performed by the data access manager, an aggregate query on a 1,000-row table is processed 75 percent faster; on a 100,000-row table, the same query is processed 230 percent faster.

## Unmatched performance for query processing

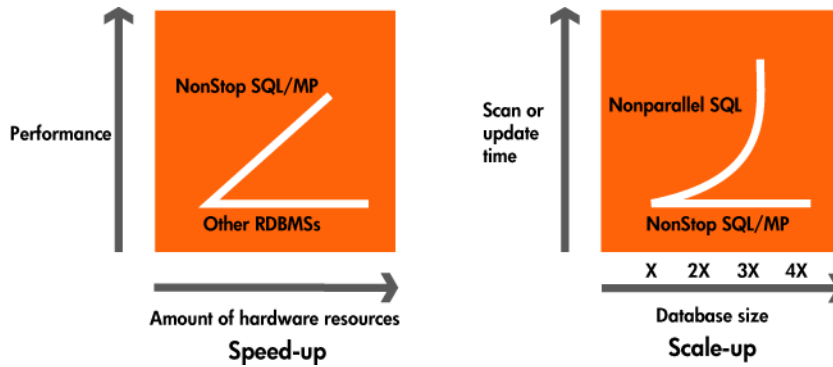
The NonStop SQL/MP relational database management system satisfies all the requirements of critical commercial query-processing operations with high performance resulting from set-oriented operations in the data access manager, horizontal partitioning, combination and elimination of sorts, hash joins,

---

<sup>5</sup> S. Englert, J. Gray, T. Kocher, P. Shah, "NonStop SQL Release 2 Benchmark," *Tandem Systems Review*, Oct. 1990.

parallel execution, and grouping and aggregation. It also provides the NonStop advantages of parallelism, scalability, and reliability.

**Figure 4.** With its scalability, NonStop SQL/MP can deliver faster response times and accommodate increased demand or database growth. These capabilities are known as *speed-up* and *scale-up*.



## Glossary

### equi-join

A join with at least one join predicate that is of the form  $table1.col = table2.col$ .

### hash join

A join algorithm similar to a nested join that first materializes the inner table into a memory-resident hash table.

### join columns

The columns referenced by an equi-join predicate.

### join predicate

A predicate that references columns from more than one table.

### nested join

A join algorithm that performs a sequential read of one table, called the *outer table*, and for each of its rows finds the matching rows in the other (inner) table.

### predicates

Conditions on rows of tables; for example,  $col1 = 5 \text{ AND } col3 > col4$ .

### search tree

A data structure in the query optimizer that helps in enumerating alternative query execution plans.

## sort-merge join

A join algorithm that performs the following two steps: First it sorts the two input tables on its join columns. Then it evaluates the join result by reading both tables sequentially, comparing the join columns and repositioning where necessary.

## References and additional reading

A. Chen, Y-F Kao, Mike Pong, Diana Shak, Sunil Sharma, Jay Vaishnav, Hansjorg Zeller, "Query Processing in NonStop SQL," *IEEE Data Engineering Bulletin*, 16, 4 (December 1993), pp. 29–41.

D. DeWitt, R. Gerber, "Multiprocessor Hash-based Join Algorithms," *Proc. VLDB*, 1985, pp. 151–164.

S. Englert, J. Gray, T. Kocher, P. Shah, "NonStop SQL Release 2 Benchmark," *Tandem Systems Review* 6, 2 (1990), pp. 24–35, Tandem Computers, Part No. 46987.

S. Englert, "Load Balancing Batch and Interactive Queries in a Highly Parallel Environment," *Proc. IEEE Spring COMPCON*, 1991, pp. 110–112.

Harry Leslie, "Optimizing Parallel Query Plans and Execution," *Proc. IEEE Spring COMPCON*, 1991, pp. 105–109.

M. Moore, A. Sodhi, "Parallelism in NonStop SQL," *Tandem Systems Review* 6, 2 (1990), pp. 36–51, Tandem Computers, Part No. 46987.

Mike Pong, "NonStop SQL Optimizer: Query Optimization and User Influence," *Tandem Systems Review* 4, 2 (1988), pp. 22–38, Tandem Computers, Part No. 13693.

D. Schneider, D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-nothing Multiprocessor Environment," *Proc. ACM SIGMOD*, 1989, pp. 110–121.

H. Zeller, J. Gray, "An Adaptive Hash Join Algorithm for Multiuser Environments," *Proc. 16. VLDB*, 1990, pp. 186–197.

## For more information

For more information about NonStop SQL/MP software, visit [www.hp.com/go/nonstop](http://www.hp.com/go/nonstop).