SQL/MX

Best Practices white paper

Part number: 540372-003

Second edition: 01/2007

Printed in the US

Part number: 540372-003

Second edition: 01/2007

Contents

Objectives of This Document

This document describes the leading best practices for designing, implementing, and supporting databases using NonStop SQL/MX. These best practices were gained from time-tested customer experiences.

Many of these techniques apply to any type of large database environment, including decision-support systems (DSS), data warehouses, operational data stores (ODS), and online transaction processing (OLTP) applications, as well as applying to smaller data warehouses.

This document does not replace the more in-depth knowledge that can be gained through the NonStop Technical Library (NTL), class education, or HP-approved consulting.

Project Planning

HP Roles and Responsibilities

This list of suggested roles and responsibilities is intended to move a typical database project toward a successful conclusion. One person can assume more than one role, although where a conflict exists between roles and the project timeline, separate individuals should take the separate roles.

For example, the roles of Database Consultant and Extract/Load/Transform (ETL) Specialist require complementary skills and could be interchangeable, but the nature and sequence of these tasks might cause difficulties if the same person attempts to perform both functions. The Technical Lead also could act as the Performance Specialist, since the major responsibility for most performance aspects of the system is on the Database Consultant with input from the Performance and Communication consultants.

Each person who fills these roles should be experienced with the NonStop operating system and OSS (Open System Services), and should have prior database experience. While third-party consultants might assist in these roles, it is strongly advised that the Technical Lead be an HP employee or approved person.

Roles

Technical Lead

- Acts as lead architect and technical specialist
- Participates in all activities defined for other consultants and specialists
- Thoroughly understands the NonStop operating system environment, and has a working knowledge of all selected products
- Acts as liaison within the project team and as primary point of contact with project management

Database Consultant

Database-management tasks

Managing an SQL/MX database involves the tasks required to create the database, ensure its availability to users, and perform required changes. Because the database is an integral part of the application, measuring application performance and tuning the database configuration to enhance performance are also database-management tasks.

The database consultant does these tasks:

- Migrates to newer releases of SQL/MX software or falls back to earlier releases
- Determines database layouts and data dictionary plans
- Plans for database security, integrity, and recovery
- Creates and loads databases
- Queries catalog tables for information about databases
- Alters databases
- Manages databases and programs
- Reorganizes and moves databases
- Manages database applications
- Performs recovery operations
- Manages distributed databases
- Assembles and optimizes queries
- Measures and enhances performance
- Does TACL/OSS scripting for job management
- Sets up and installs any database-management tools

- Reviews queries
- Explains plans
- Is responsible for the physical design of the database based on accepted database design principles
- Facilitates a transfer of information of the physical design of the database to customer personnel
- Thoroughly understands SQL/MX software technology and its application to a database that must support database activities
- Understands the implications of the design on load, maintenance, and archive activities
- Works closely with the designer of the logical database to translate the design into an implementation that meets the project objectives

Extract, Transform, and Load (ETL) Specialist

- Acts as the architect for the design of all NonStop operating system–based data loading, data extraction, and transformation functions
- Develops functional specifications to be used by programmers to implement the architecture
- Oversees the implementation of the ETL architecture
- Thoroughly understands parallel processing principles and the NonStop operating system application-development environment
- Thoroughly understands SQL/MX technology and its uses
- Understands various messaging formats (for example, XML)

Performance Consultant

- Provides in-depth knowledge of system-level performance and tuning
- Works closely with other HP consultants and architects to identify potential performance problems and suggests corrective action
- Aids in the sizing of the system prior to implementation
- Has a thorough knowledge of NonStop operating system tools, such as Measure and HP Asset BMC

Database Communication Consultant (ODBC and/or JDBC)

- Has a specialized knowledge of LAN technology and of the ODBC/MX product or JDBC access
- Is responsible for configuration of the MXCS environment, physical placement of the MXCS Server installation, and configuration on the workstations to be used by ODBC Server tools and utilities.
- Is responsible for the installation of the Java Runtime for NonStop SQL/MX

Operations and Systems Specialist

- Has general knowledge of operations management best practices
- Has a knowledge of NonStop operating system tools and their applicability to the project
- Has a knowledge of OSS tools and their applicability to the project
- Devises and oversees application and system backup functions
- Devises and tests data-recovery strategy
- Performs system software installation and configuration

Project Activities Guideline for Database Activities

This section lists activities that have been successfully employed at HP accounts. The list is intended as a guide to acquiring the information and data necessary to implement a database solution successfully. Assumptions include the use of a phased approach to limit the scope of the project and to implement additional functionality in subsequent phases. The completion time for each task will vary based on the customer's experience with successfully implementing database solutions. The actual time will be a function of the overall project's complexity and scope.

Requirements Definition

- User interviews. Conduct interviews with representatives of user groups and business groups. The customer's project-team member who represents the interests of the group that will regularly use the system should be present for all interviews. Allow 60 to 90 minutes for each interview. Schedule no more than four interviews each day, leaving 30 minutes between interviews in which the interviewers can compare notes.
- Document requirements. Encapsulate, combine, and consolidate the interview notes to produce a draft of the requirements definition.
- User review. Review the draft requirements with the user groups. Take no more than 30 minutes with each user group.
- Requirements completion. Produce completed requirements-definition documentation.

High-level Architecture and Design

Architecture

Obtain these information:

- Type of database
- Number of levels
- Source identification
- Data flow
- Change data capture
- Transformation and cleansing
- Interaction among components
- Network requirements
- Service-level agreements
- Volume of data

Logical Design

- Data analysis. Identify all data sources. Determine source data types, formats, ranges, and validation. Obtain row and record counts, unique entry counts, data distribution, and second high and low values.
- Logical database design. Define all entities and attributes. Identify all candidate keys.
- Data cleansing. Determine the correct format, ranges, and validation for data from disparate sources. Determine the system of record for the data.
- Logical schema. Determine the data model that will represent the logical database.
- Data transformation. Determine the desired format, ranges, and validation for data to be loaded.

Query Analysis

- Query types. Determine the query types and rank by importance. Group them by complexity. Determine frequency of execution. Limit your analysis to approximately 10 to 20 queries. More would be too complex; fewer would not represent business requirements.
- Query structure. Are the queries produced by individuals or intelligent clients? Are there multistep or multipass queries?
- Response time requirements. Determine the response-time requirements for each type of query. Determine which queries require online response times and which queries can use deferred response times.

Logical Access

- Translate queries. Translate business queries to SQL queries. Identify tables, joins, predicates, groupings, aggregations, and so on.
- I/O analysis. Estimate the number of rows accessed and the number of rows used. Evaluate these numbers against response-time requirements.
- Logical query analysis. Perform logical analysis using the Asset BMC product.

Client Tools

- Tool categories. Determine the categories of tools that must be supported for each user group.
- Tool selection. In each category, choose two or three tools that must be supported for evaluation (for example, relational online analytical processing [ROLAP], statistical, ad hoc, and so on). Limit the number of tools to be evaluated. Note that this number may be driven by user preferences or installed tools, or both. Is the tool "database aware"? Does the tool aggregate tables, insulate the user from physical changes, and so on?
- Tool database requirements. Determine the space, metadata, and platform requirements; the connectivity requirements; data-type compatibility; naming-convention requirements, and so on.
- Data mining. Will data-mining tools be used against data extracts? Can data-mining tools that access NonStop SQL/MX be used directly? (This determination might require a common definition for the term "data mining." Many users misuse this term, thinking that ROLAP and data mining are the same.)

Physical Design

Denormalization Analysis

- Join elimination. Eliminate joins, define aggregate tables, and resolve data-sparsity issues.
- Aggregate definitions. Identify which aggregates are frequently used. Estimate aggregate disk space. Identify the impact on load and update processing. Does the potential exist for significant positive-performance impact?
- Data sparsity. Will combining tables create data sparsity? (For example, combining transaction tables and account history tables would improve account history queries, but not every account has activity every day.)

Partition and Primary Key Analysis

- Analysis of time-ordered data. Partitions by date allow for fast loads and archiving, but create hot spots. Partitions by another dimension key can create fragmentation, reducing sequential I/O and creating need for frequent reorganization.
- Hash partition analysis. What is the impact on load and update processing and on the archive strategy? Should the sequential I/O be increased or decreased?
- Skewed data analysis. Do candidate columns for the primary key contain skewed data?
- Fragmentation analysis. What degree of fragmentation is introduced by the data-load cycle? How much of the database must be reorganized and how often must reorganization occur to eliminate the fragmentation?

- Primary key analysis of fact tables. Are multicolumn keys needed or desired? Determine key distributions by partition. Consider the effect on MDAM query technology processing.
- Primary key analysis of dimension tables. For a star schema, typically choose single-column keys. To use multicolumn keys, you must revisit the data model. Is the snowflake design technique a better choice?
- Map partitions to configuration. Determine which partitions of which tables to collocate. How would partitions affect fact and dimension tables and query-plan generation and execution? Also, consider the effect of partitioned and non-partitioned dimension tables.
- Analyze artificial values against intelligent values and derived key values. Is key mapping necessary? What is the effect of key mapping on query performance? Can MDAM query technology be used for intelligent keys? Derived keys or meaningful keys can be exploited to avoid joins. Will dimension-key changes cascade to the fact tables?

Secondary Index Analysis

- Design for MDAM query technology. Avoid the use of secondary indexes on fact tables. Small dimension tables might not benefit from secondary indexes.
- Design for index-only access. If secondary indexes are used, can index-only access be designed in?

Create Catalog and Schema

- Create the catalog and schema. Create the system and database catalog and schema.
- Create the Data Definition Language (DDL). Build the DDL to reflect the specifications listed in this section.
- Create scripts to manipulate and manage the schema. Scripting may be done using utilities, editors, and so on, as defined by the project.

Space Analysis

Temporary space analysis. How much space is needed for hashing of temp tables, for client tools, for sort space, for refresh and staging areas, and for anticipated growth?

Physical Access

- Location of catalog and schema, application, and audit. Determine the impact of metadata placement on query performance. Determine the correct audit configuration.
- View analysis. Analyze the views to hide partitions and force joins. Hide some physical constructs from client tools.
- Analysis of parallelism. Will parallel query plans be generated? Will parallel plans degenerate into serial access? How many executor server processes (ESPs) will be spawned?

Insert and Update

- Change data capture. Where does data capture occur? How is data migrated to the data-load process? Are updates applied to history?
- Data transformation. Translate, encode, decode, and recode.
- Cleanse, correct, and discard. Reconcile, integrate, combine, and de-dup.
- Dimension table updates. Are dimension-table updates changing slowly? Update, insert, and reload.
- Audit implications. Audit is always on. Inserts generate audit but immediate dump is necessary if the data is high volume.
- Keys for aggregates. What is the use of dimension keys compared to artificial keys?
- Management of transactions. How and when are transactions bracketed?
- Recoverability and restart ability. Build robust code. Do tables need to carry a “last update” column?

- Trickle compared to batch updates. Compare trickle against batch. What is the impact on rerunning queries and on query performance? Consider the size of the load and update window and the impact on aggregates.
- Custom versus DataLoader/MX versus tool. Consider the need for parallelism and for robust architecture. Compare flexibility against a "canned" approach, partition management needs, and so on.
- Coded data insert and update processes. Coded data insert and update processes are resource intensive and require a thorough knowledge of the database management system (DBMS), regardless of the tool or approach used.
- Initial data load. Consider the load dimension tables, the initial load of fact tables, and the initial build of aggregate tables.

Manageability

- Catalog and schema statistics. Update statistics for all tables. Modify statistics as needed for skewed data.
- ESP management. Control the number and placement of ESPs through the use of a partitioning scheme for dimension tables.
- Partition management. Define the schedule and create the scripts for partition reorganization and growth and splits.
- Memory management. Configure the cache while keeping in mind the use or non-use of cache for sequential I/O. Consider the hash memory requirements.
- Resource accounting. Evaluate and implement resource accounting within ODBC.

Design Guidelines and Considerations

Physical System Configuration Guidelines

Distribute the disk subsystem across all processors to balance more evenly the I/O processing activity and thereby minimize processor and disk bottlenecks.

Construct a symmetric hardware configuration where all hardware (especially disk devices) is distributed over all processors as evenly as possible. Some customers might need to obtain additional I/O enclosures to achieve a balance. The new XP storage system provides better performance than using internal drives.

Balance resource use and queues. It is extremely difficult to achieve the type of system balance normally seen on OLTP systems, since ad-hoc queries are much more dynamic. The goal is to eliminate the resource bottlenecks—usually processor and disk—and worry less about overall resource balance. The resource queues impact response time, and the resource queues result from excessive workloads concentrated on a few devices.

Balance database activity across the disk and processor resources. For overly busy disk volumes, move or split the active partitions and distribute them to less-busy devices.

Balancing and tuning complex database applications is a challenging task, made more difficult when the underlying hardware is unbalanced. Systems that have an uneven number of disk volumes for each processor, or an uneven number of I/O enclosures, create a natural imbalance in resource use that is difficult to tune around. Having only a few CPUs to process the workload results in extremely long queue lengths, degrading performance for any process running in those processors. Bottlenecks such as these mentioned here degrade any application. However, severely unbalanced processors and disks are especially bad for parallel queries.

When a parallel query runs, groups of processes known as executor server processes (ESP) each run a subset of the query. The results are later combined and returned to the user. When one processor is extremely busy, the ESPs running in it cannot make adequate progress servicing the query plan. Although all other ESPs may have completed their tasks, the degraded ESP delays the overall completion of the query. Therefore, a single system bottleneck is extremely important to resolve; use a graphical interface to monitor system performance, in addition to monitoring EMS events.

When the I/O processing activity is balanced, disk memory cache is used more effectively, which improves performance.

Logical Database-design Considerations

For a detailed discussion about designing and implementing a database application on the NonStop server platform, refer to “Database Design Guidelines for Improving OLTP Performance” in the SQL/MX Installation and Management Guide.

Physical Database-design Considerations

Primary, Partitioning, and Clustering Keys

Definitions:

- The Primary Key is used to uniquely identify rows in a table.
- The Partitioning Key is used to identify the disk volume locations of the rows.
- The Clustering Key is used to physically order rows

Primary and clustering keys have often been thought of as the same key. By default, the clustering key is the primary key. However, different columns can be used as the primary and clustering keys. Therefore, these keys should be thought of as separate key items.

SQL/MX table data is physically maintained in a b-tree format. The organization of the b-tree is based on the clustering key. The clustering key must be unique, in order to support the b-tree architecture. By

default, the clustering key is the primary key, but a separate clustering key can be specified with the STORE BY clause.

When a primary key is specified for a table, the clustering key can comprise all or some of the columns of the primary key. If only some of the primary key columns are used for clustering, you must use the leading columns of the primary key in the STORE BY clause. When all or some of the primary key columns are used in the STORE BY clause, the data is clustered by the primary key. In this situation, the primary key becomes the clustering key.

When a primary key contains the DROPPABLE clause, the columns of the primary key can be updatable. When the primary key is droppable, the columns in the STORE BY clause can be different from the primary key columns. Data will be clustered by the STORE BY columns. In this situation, the primary key is not the clustering key. To enforce uniqueness of the clustering key, a SYSKEY is appended to the end of the clustering key. The b-tree organization of the table is based on the clustering key. The primary key is maintained in a separate index, generated by NonStop SQL/MX. This generated index will be a single partitioned index and will physically reside on the volume of the first partition for the table. Using the MODIFY utility, you can alter the primary key index to move the location of this partition and add additional partitions based on range partitioning. Use the SHOWDDL command to determine the name of the SQL/MX generated primary key index.

NonStop SQL/MX supports two types of partitioning types: range partitioning and hash partitioning. Range partitioning places rows into different disk volumes based on the range partitioning key. Hash partitioning places rows into different disk volumes based on an algorithm of the hash key. The partitioning key must be the clustering key or part of the clustering key. If the clustering key has a SYSKEY appended to it to provide uniqueness when the clustering key is not the primary key, the SYSKEY portion is not used for partitioning. In other words, the SYSKEY is dropped when partitioning occurs, since the partitioning key does not need to be unique.

Rows are stored in each partition based on the order of the clustering key. Using the clustering key to access data will provide the most efficient access method. Therefore, it is recommended that the columns used as the clustering key should also be used as the primary key.

Primary-key Selection

The selection of a primary key is one of the most important design considerations. The proper key structure gives the query an efficient access path, which enables good performance.

Primary keys for fact tables normally are multicolumn keys composed of foreign key relationships with dimension tables. The combination of dimension column values makes each fact table row unique. Since the primary key is multicolumn, you can select the column order, which can have a substantial impact on query performance.

Generally, the primary-key order is determined by the data-access requirements. That is, query requirements usually are given primary consideration, while load requirements are secondary. Columns that restrict the search space the most, and are used often by queries, are good candidates for the high-order column. When different queries access the data through different columns, you must determine column placement based on additional considerations. You also might need to create a secondary index on certain columns to avoid full-table scans, although, for many cases, MDAM query technology eliminates this need. (Refer to the “MDAM query technology design considerations” section of this document for more details.)

The principles of primary-key selection also apply to dimension tables. However, dimension tables usually have fewer columns and require fewer considerations for their design. Primary keys for dimension tables are usually composed of a single column, although sometimes multicolumn keys are used for tables that have meaningful keys.

Listed below are the most important considerations for primary-key selection. Determine the priority of these items based on performance needs:

- Data-access efficiency
- Cardinality
- Partitioning

- Join efficiency
- Insert efficiency
- Data clustering

You must determine whether to create the index before or after the data is loaded to the table. If you create the index first, the index will be a constraint, limiting the ability to insert duplicate data. If you create the index after the data is loaded to the table, you may need to eliminate any duplicates prior to creating the unique index.

The primary key (when different from the store-by value) created during the CREATE TABLE statement, creates a single partition index on the same drive as the first store-by partition. This single partition index is a physical index that can be modified, but you must use the index name that the system has generated for the physical index. To get the index name, use SHOWDDL on the table. The primary key can only be modified by using range partitioning. You cannot use hash partitioning to modify a primary key.

You cannot update a column that is part of the non-droppable primary key or the clustering key. To update a column, you should plan to have the data to be modified defined as an index instead of as a primary key or clustering key. Otherwise, to update the data in the column, you must follow these steps:

1. Select the rows containing the columns to be modified from the table to program storage variables or a temporary table.
2. Delete the rows from the original table.
3. Update the columns in the program or temporary table.
4. Insert the rows back into the original table from the program or temporary table.

At this time, you cannot use NOT DROPPABLE on a primary key when the primary key differs from the store-by key. Usually, it is recommended that you use NOT DROPPABLE because it aids performance. A column that potentially contains a null value is not checked for an additional constraint. This suggestion also applies to making columns NOT NULL NOT DROPPABLE.

For better design and functionality, you should create a unique index to function as the primary key (however, only if the store-by key value differs from the primary-key value). A unique index allows you to use hash partitioning if desired, and to easily modify the locations of the partitions.

If the primary-key value and the store-by key value are the same, the data is partition-distributed in the format requested by the STORE BY syntax.

Data Access Efficiency

As described in this document, data access efficiency is the degree to which query predicates restrict the search space of the data selected. You must have prior knowledge of the most common predicates that queries will use. Look for predicates that are common to all or most queries. These predicates restrict the search space for the majority of database queries. These columns become primary candidates for determining the key order.

Cardinality

Cardinality is the number of unique values that can appear for a particular column. For example, a demographic column might contain only 10 distinct values, although that column is included in several million fact table rows. Hence, the column's cardinality is 10.

Cardinality is an important consideration for MDAM query technology processing. Low-cardinality columns tend to enable efficient MDAM usage. Cardinality is most useful for avoiding full table scans when the query omits predicates on leading key columns.

When you have a choice, place columns that have low cardinality as high in the key order as possible, and in increasing order of cardinality. This arrangement ensures the greatest degree of MDAM query technology processing when high-order key columns are omitted from the query. You can place columns that have low cardinality after columns that have higher cardinality, providing that predicates are always supplied on the leading high-predicate columns. MDAM query technology is used for the low-cardinality columns when those columns are omitted from the query predicates. However, generally you should give preference to data-access efficiency considerations over cardinality. That is, if you are certain that a

particular high-cardinality column will always be included in a query predicate, use that column as part of the primary key because that column will immediately restrict the query search space.

Be aware of the following situation associated with low-cardinality columns. Consider a case where the primary key is composed of a very low cardinality column (for example, 1 or 2) that is always specified in the query, followed by a column that has very high cardinality that is rarely included in a query predicate. Even though the query always includes a predicate on the leading column, the effect is similar to a primary key that omits the leading low-cardinality column. This is because the predicate on the low-cardinality column does little to restrict the search space. A cardinality of 1 or 2 has a corresponding selectivity of 100 percent or 50 percent, respectively.

Partitioning

Partitioning is a method of distributing a large table over multiple volumes (and systems) that remain transparent to application programs and end users. The partitioning key corresponds to the leading primary key columns and groups rows beginning with the named partition key value into the same disk partition. This arrangement is also true for hash partitioning, which is derived from the value of the key.

Partitioning is an important consideration for achieving effective parallel-query and load processing. Since partitioning depends on the chosen store-by key, partitioning and primary key selection decisions must be coordinated for proper operation. See the “Fact table partitioning techniques” and “Dimension table partitioning techniques” sections of this document for more details.

Design Techniques to Avoid

Joining Tables on Calculated Columns

Joins are usually made between the existing columns of two tables. A calculated column is one in which an arithmetic calculation must be made to generate the column value prior to the join. Usually, this calculation requires a full scan of the table, and either a sort or a repartition hash on the calculated column value, even if a small row-set results from the join. (A repartition hash hashes the calculated column value, writes the results to temporary workspace, then joins to the other table.) This operation is very expensive and should be avoided, if possible.

Over-normalizing Tables

Some degree of normalization is acceptable in a database. However, avoid designs that rigidly enforce normalization or are “over-normalized.” These structures are unlikely to perform well for queries, especially when the tables involved are large and the expected number of rows accessed also is large.

In some cases you can implement the original normalized structures—which maintain complex relationships—but extend the design by replicating data from selected tables into a single structure to meet performance requirements. This arrangement can work well when the original structures are characterized by low volatility and when the database supports mixed application requirements, such as in ODS and DSS applications. Over-normalization can be very useful if a logical table has an “active” part that is used in many queries, and a “non-active” part that is used in only a few queries. By splitting the table in two, you get better query performance for those queries that select only the active set of columns.

Load-only Optimized Design

Database projects often are characterized by tight schedules; implementation begins even though query requirements are barely understood by the designers. This situation might be acceptable for proof-of-concept efforts or for pilot projects, which often are experimental in nature. But it is a mistake to proceed with a production implementation without understanding how the queries will access the data.

To construct the database, the designers must make choices about primary keys, key-column placement, partitioning, and indexing. Invariably, tight schedules result in suboptimal design decisions, causing either the users to have to accept slower query-response times or the database to be rebuilt or significantly modified.

In the absence of query requirements, designers often optimize for the requirements they understand, usually database loading. Although such a design is not necessarily bad, it often produces good performance for database loading, but poor performance for query processing.

Whenever possible, try to anticipate the query requirements in your design decisions. At the very least, set the expectation that portions of the database might need to be modified (and programs changed) once query requirements are known. Consider implementing a full-scale (or near-full-scale) pilot of the proposed design, and have alternative designs ready in case they are needed.

Built-in Application Bottlenecks: Inadequate Design Analysis and Testing

During project planning, provide enough time for adequate testing, planning, and investigation of alternative database designs. Take time to create a large or full scale test database, partitioned and organized as it would be in production. Construct typical queries and analyze their DISPLAY_EXPLAIN output to determine whether the database is properly structured. Load the tables with test data and use the Measure product to analyze both load and query performance. Load-only design is another potential issue; see the “Load-only optimized design” section above.

Database Sizing Considerations

General Rules for Sizing

This section includes some general rules for sizing a database system when very few details are known about the application.

- A processor supports about 30 to 50 gigabytes of data (logical data, based on common customer implementations); the number varies based on the processors and types of storage chosen. Please consult your hardware representative for your processor and storage configuration abilities.
- The total table space is about 1.5 times that of the base table data. Indexes generally are used only on the dimension tables.
- Specify at least one empty disk volume for sort and workspace.
- Specify at least one empty disk volume for OSS for standard I/O.
- Purchase the maximum memory that the processors can support.
- Plan to mirror all volumes. There is an advantage to parallel reads from a disk that is mirrored.

Sizing Questionnaire

These questions arise from the need to properly size the NonStop system. Much of the information needed probably already exists from the initial project-planning activities.

Quantifiable Metrics

For each legacy system involved:

- How many records will be extracted:
  - For the initial load?
  - For cyclical updates?
- What are the record sizes?
- What is the frequency of extraction?

For the Database System

- What is the required transaction each second?
- What are the service-level agreements?
- How many rows will be loaded initially?
- What is the anticipated row size?
- What is the frequency of updates:
  - For the operational data store (ODS)?
  - For the warehouse?
- How many rows will be inserted cyclically? Updated? Deleted?
- What are the response-time requirements?
- What is the number of concurrent users? (For sizing, concurrent is defined as the number of users actively running a query and not the number of sessions established.)
- What is the total number of users?
- What is the total number of sessions?
- How long will a history be maintained:
  - In the ODS?
  - In the warehouse?

Network Data

For each legacy system involved:

- What is the network connection?
- Are multiple links to the legacy systems needed or desired?
- What is the preferred transfer mechanism—for example, file transfer protocol (FTP) or network data management protocol (NDMP)?
- Does the transfer mechanism support the anticipated volumes and cycles of updates? (This consideration is not really a sizing parameter, but rather a “sanity check” regarding the preferred mechanism.)

Operational Data

For each legacy system involved:

- What is the system platform, operating system, and database or file system?
- What are the system-maintenance windows?

For the Database System

- What is the system availability (for example, 9 x 5, 15 x 7, or something else)?
- How much data will be archived, and what is the archive cycle?

Work and Swap Space Considerations

- Establish effective, consistent environmental settings for users of both MXCS clients and ODBC clients, including the clients using third-party tools.
- Designate specific disk volumes for workspace requirements—specifically, the sort workspace and temporary workspace for query-process components, when possible.
- Distribute this information to interactive users of MXCS clients, or have the information loaded into their HP Open System Services (OSS) segments at logon.
- Add this information to the ODBC/MX configuration.
- Monitor workspace usage over time to determine if and when additional space is required. To get helpful information about space usage, you can use the HP Event Management Service (EMS) logs, Availability Stats and Performance (ASAP), Disk Space Analysis Program (DSAP), and the HP Measure product.

About SQL/MX Scratch Disks

NonStop SQL/MX selects a disk to be used for a scratch file from the pool of available disks. The pool initially consists of the set of all suitable disks. Disks such as optical disks, phantom disks, and SMS virtual disks are not considered suitable. The disks specified by the SCRATCH_DISKS_EXCLUDED control are removed. If the SCRATCH_DISKS control is specified, the disks that are not specified in the SCRATCH_DISKS control are removed from the pool. From this disk pool, a disk is selected based on this criteria:

- The amount of used space on the disk. (rank * 30)
- The number of scratch files on the disk. (rank * 70)
- The number of fragments on the disk. (rank * 20)
- The biggest available fragment on the disk. (inverted rank * 80)
- Is the disk a preferred disk? (10000)
- Is the disk the primary disk of the CPU of this process? (100000)

The value in parentheses indicates the weighting of that criterion. The rank is the ordinal rank of that disk among all the disks in the pool based on the criterion. The inverted rank is the inverted ordinal rank. In the case of the biggest available fragment criterion, if the pool contains 20 disks, the disk with the biggest available fragment would have an inverted rank of 20. The weights are summed for all the disks in the pool, and the disk with the biggest weight is selected. As can be seen, the primary disk of the current CPU is given a large weight.

In NonStop SQL/MX, a scratch file can overflow to another disk. So, if a scratch file becomes full or if the disk becomes full, the operation does not necessarily fail. An additional scratch file on another disk is selected (using the criterion procedure). As a result, there is no 2 GB limit on scratch space.

In NonStop SQL/MX, the operations that can create scratch files are sort, hash join, and hash groupby. They all use the criterion procedure to determine which scratch disk to use.

- NonStop SQL/MX does not manage swap file space directly. Instead, SQL/MX processes rely on the Kernel-Managed Swap Facility (KMSF), which is setup in the NonStop operating system with the NSKCOM tool. Each CPU has an associated swap file.

Scratch Disk Management

These attributes determine how NonStop SQL/MX manages scratch disks for the SORT operation:

- SCRATCH_DISKS_EXCLUDED
  - Set to a list of scratch disk volumes, where each item in the list has the form [\node.]$volume, and the items in the list are separated by a comma (,). Use this default to exclude certain volumes from being used for scratch disks.

    If none of the three scratch disk defaults are set, the system determines the scratch disk volumes to be used.
- SCRATCH_DISKS_PREFERRED
  - Set to a list of scratch disk volumes, where each item in the list has the form [\node.]$volume, and the items in the list are separated by a comma (,). Use this default to indicate preference for volumes to be used for scratch disks.

    If none of the three scratch disk defaults are set, the system determines the scratch disk volumes to be used.
- SCRATCH_FREESPACE_THRESHOLD_PERCENT
  - Indicates how much free space, as a percentage, is left on a disk as a threshold. When that threshold is reached, hash or sort operations will use a different disk. If all disks reach their threshold, NonStop SQL/MX displays an error.

    The default value is 10. When disk usage reaches the point where only 10% of the space remains, hash or sort operations stop using that disk.

Data-type Considerations

Use either standard DATE or DATETIME data types for date columns. Do not use YEAR TO MONTH, because YEAR TO MONTH is not standard ANSI and can lead to problems when accessing data through tools that rely on the ODBC or ANSI standards.

No auto-incrementing column type is available. This function exists in some other databases, but is not currently available in NonStop SQL/MX. It is recommended that you handle this action programmatically by using a cache of numbers from a source table. This arrangement eliminates any bottlenecks that arise from returning to the source table for each number needed.

Consideration for Summary Tables

Some queries might benefit from the creation of summary tables, where pre-aggregated data is created along one or more dimension attributes such as claim type, region, month, and so on. Although these tables require more effort to build and maintain, they can produce profound performance improvements for suitable queries that are currently long running, especially because these tables are expected to reduce row access by a factor of 100 or more. Queries must be analyzed individually to determine whether summary tables would be useful.

Summary tables are also useful for materializing calculated columns, based on existing table columns. Applications that perform frequent extensive calculations can benefit from accessing precalculated columns.

Carefully determine the space requirements of summary tables. Often, summary tables are larger than expected due to the density of the dimensional values when aggregated compared with the sparsity of the dimensional values in the fact table.

If fact table updates are permitted, you must determine whether the summary tables should be updated or rebuilt. Correctly updating the summary tables is challenging, while rebuilding the summary tables requires many processing cycles.

Third-party productivity tools can be useful for creating and maintaining summary tables. These tools can generate programs that create summary tables for initial and ongoing loads. These tools remove much of the labor associated with creating a summary table.

Tables also can be maintained using publish and subscribe features. Analyze this aspect thoroughly to determine if this solution fits your situation.

Considerations for Secondary Indexes

It is desirable to avoid indexing fact tables for a number of reasons. Secondary indexes on large tables can require significant database space, and indexes are usually useful only when relatively few rows match the predicate. Usually it is more efficient to perform a table scan.

When the table design is suitable, MDAM query technology often provides better performance than a secondary index. However, secondary indexes do have their place. They are commonly used for large dimension tables, to provide efficient, direct paths along predicate columns. Secondary indexes offer several advantages: they are easy to build (no code development); they are maintained automatically by the system (and in parallel when more than one exist on the same table); they are transparent to queries; and they can be dropped quickly and easily.

These characteristics generally indicate that a secondary index is useful:

- Tables are large.
- Rows are added using the SQL/MX database INSERT statement.
- Access is by key-column, other than the leading primary key column or columns, or by a non-indexed, data-column.
- Predicates produce a fairly low selectivity (where only a small portion of the rows would qualify).
- MDAM query technology is not used.

Performance also can be improved by ensuring index-only access, where columns are retrieved from only the secondary index without needing to access the base table, saving I/Os. Include in the secondary index additional non-key columns that are likely to be used with the indexed column. (Key columns are always included in the secondary index.)

Partition secondary indexes to distribute the workload and data across multiple resources.

You cannot run SHOWLABEL during an index build; SHOWLABEL cannot locate the information from the catalog while the index is being built. The error message does not explain that the table is unavailable; the error message states only that the table entry cannot be found.

HP recommends that you create indexes after loading the data.

Data Locality Objectives

Data locality refers to the objective of keeping data and processing together, such as on the same system. Very Large Database (VLDB) applications often require multiple interconnected systems with the database distributed across all systems. When possible, run programs on the system where the data resides. This practice is usually feasible when loading data or performing other batch activities, but more difficult during query processing.

Collocation of the data in the keys and indexes is best. Collocation means keeping similar information to be joined on the same drives. Collocation works both for range and hash partitioning. By using collocation of the data, you decrease the amount of message traffic between disks for joins. If all the same linked data is located on the same drive, the join can be isolated to that drive, without having to message other drives for the join data.

When designing a database distributed across multiple systems, you have several alternatives for locating tables. You must consider the location of the fact, dimension, and summary tables, as well as secondary indexes.

You might choose to keep each table entirely on one system, but distribute the collection of tables over all systems, which is feasible if the division of tables coincides with the workload division. This approach is the easiest to manage and makes efficient use of resources, including disk space.

Alternatively, you might choose to distribute the fact tables across systems and replicate all dimension tables, summary tables, and secondary indexes, which is feasible if the division of fact tables coincides with the division of their use; users on a specific system access data only on that system. This approach provides good performance, but requires replicating large amounts of data.

Most customers cannot effectively maintain data locality for query access because of the ad-hoc nature of query processing. Consequently, customers usually distribute the fact tables across systems, and may or may not replicate dimension tables, summary tables, and secondary indexes. Many customers avoid dealing with this issue by migrating to the latest hardware technology that allows them to use a single system.

Additional Information

Renaming a Table

NonStop SQL/MX does not provide a mechanism for renaming tables. However, you can use these workaround:

1. Create temporary or additional tables.
2. Load the data to the new table or tables.
3. Recreate the original table.
4. Load the data back to the recreated table using the original table name.

Limits

This list describes limits for various parts of NonStop SQL/MX Release 2.0:

- Constraints

  The maximum combined length of the columns for a REFERENCE, PRIMARY KEY, or UNIQUE constraint is 255 bytes.

- DROP SCHEMA CASCADE transaction limits

  You might need to increase the number of locks for each volume; otherwise, DROP SCHEMA CASCADE might fail.

- EXTENTS

  Limited only by the size of the disk.

- FROM clause of the SELECT statement

  16 SQL/MP tables are allowed. For SQL/MX Release 2.0 tables, NonStop SQL/MX generates good plans up to approximately 40 tables. Beyond that number, executor performance is adversely affected.

- IN predicate

  1900 expressions are allowed.

- Indexes

  The maximum combined length of the columns for an index is 255 bytes. A nonunique index consists of columns and a clustering key. A unique index consists only of columns.

  No restriction exists on the number of indexes for each table; however, creating many indexes on a table affects performance.

No restrictions exist on the number of partitions that an index supports; however, beyond 512 partitions, performance and memory issues can occur.

- INSERT operations

  150 records can be inserted into SQL/MX tables in a single INSERT operation.

  250 records can be inserted into SQL/MP tables in a single INSERT operation.

- Joins

  40 tables can be joined, including base tables of views, but joining more tables affects performance.

- MAXEXTENTS size

  NonStop SQL/MX supports 768 (compared with 919 for NonStop SQL/MP).

- Partitions

  No restrictions exist on the number of partitions that a table can support; however, beyond 512 partitions, performance and memory issues can occur.

  Partitions can be on the same physical disk as the main file.

- PFS (Process File Segment) usage is decreased in the file system

  This issue affects the number of opens.

- Referential constraints

  A table can have an unlimited number of referential constraints, and you can specify the same foreign key in more than one referential constraint, but you must define each referential constraint separately.

- Tables

  The maximum length of a row is 4040 bytes, but not all of that space is available. Depending on the data types that you use, NonStop SQL/MX will always use some bytes for internal use.

  The clustering key length for a table cannot be longer than 255 bytes. If the table has triggers, the clustering key length cannot be longer than 247 bytes; if the key length is greater than 247, trigger creation will fail.

Practical Limitations

If you do not need parallel execution, or you can use an outer table with fewer partitions and join to the larger table, it can be futile to create such a large table, even though it is possible. In this case, you must reduce the number of partitions by omitting either columns or rows from the table.

Techniques for Creating Large Tables

Using a Single CREATE TABLE Statement

Writing a CREATE TABLE DDL statement for a large table in MXCI is tedious because of the large number of partitions involved. Instead, you should use an OSS script or other means to generate the DDL statement. Tools such as Perl can be used to automate this process.

Additional Considerations Tables

The LOCATION clause in the CREATE TABLE statement provides the ability to assign a physical location for a table and its partitions. It also provides the capability to assign a more meaningful name for each partition. A good practice is to use the LOCATION clause to assign physical locations and meaningful names. Otherwise, NonStop SQL/MX could generate a random name for each partition.

The LOCATION clause is also used in the MODIFY utility. If you move a partition, the initial name used for the partition in the LOCATION clause of the CREATE TABLE is not maintained. If the LOCATION clause is not used when a partition is moved, a new SQL/MX-generated name will be created. In order to maintain a consistent naming conversion, you must use the LOCATION clause.

This practice enables RDF to properly map files between primary and backup systems. If you allow the system to generate the names, there will be incompatible names between your primary and secondary systems.

Using SQLMXBUFFER to Improve Database Performance

SQL/MX Session Data Space and Data Cache Guidelines

When a SQL statement is compiled, a query plan is created. Depending on the SQL statement, a query plan can execute against a single non-partitioned table, several partitions of a table, or several tables in the case of a multi-table join. The part of the query plan that can be executed by the Disk Process (DP2) is stored in the SQL/MX buffer. The piece of the query plan executed by a specific DP2 is called a query fragment. For a partitioned or multi-table query, query fragments are sent to the DP2 where the data can reside. In addition to the query fragment, DP2 can store state and temporary working data in the SQL/MX buffer. Therefore, it is important to have reserved enough SQL/MX buffer space to execute the query.

SQL/MX buffer space competes for the same virtual memory space with DP2 data cache and lock tables. DP2 allocates all segment spaces based upon the user configuration, except for DP2 data cache. The default size of SQL/MX buffer space is currently 128 MB. This leaves about 850 MB of virtual space for a maximum data cache with a default configuration.

If the cache configuration is larger than the available space, DP2 automatically reduces the cache configuration to match the remaining virtual space. The default size of the SQLMX buffer space may be insufficient, and you can increase the size by using SCF. If you increase the size, the data cache may be reduced, if needed.

SQL/MX buffers can be reused when a query is repeated. This avoids the need to resend the query plan when the application re-executes the query. If the reuse fails, it indicates that the query plan was stolen due to insufficient SQLMX buffer configuration. This results in an increase in messages between the applications and DP2. As a result, SQL/MX performance will be sub-optimal.

SQL/MX buffers can be monitored through SCF. Below is an example:

```
 SCF - T9082G02 - (04MAY04) (07APR04) - 05/12/2005 22:26:46 System \DRIS1

(C) 1986 Tandem (C) 2004 Hewlett Packard Development Company, L.P.

(Invoking \DRIS1.$SYSTEM.STARTUP.SCFCSTM)

1-> stats disk $D*,sqlmx

STORAGE - Stats DISK \DRIS1.$D01001


SQL/MX Statistics:
  Session Data bytes.......    786432 KB  Max Data bytes...    786432 KB
  Total Sessions...........      1708     Active Sessions..         0
  - 4KB Blocks  - - - - - - - - - - -     - Reuse - - - - - - - - - -
  Max......................    195840     Attempts.........     60155
  Number...................    195840     OK...............     58214
  In Use...................     53621     Failed FST.......        12
                                         Failed ID........      1929
```

Total Sessions	The number of query fragments residing in SQL/MX buffer space. A Session is a query fragment.
Session Data bytes	The amount of SQL/MX buffer space in use by the Total Sessions.
Max Data bytes	The amount of SQL/MX buffers configured for the disk. In this example, this disk was configured for the maximum amount.

MAX under 4KB Blocks	The amount of 4 KB blocks that can be used within the SQL/MX buffer space for query fragments.
Attempts under Reuse	The number of times an attempt was made to reuse information in the SQL/MX buffer.
OK under Reuse	The number of times an attempt to reuse information in the SQL/MX buffer was successful.
Failed FST and Failed ID under Reuse	The number of times an attempt to reuse information in the SQL/MX buffer was unsuccessful. An unsuccessful attempt could occur because one or more 4KB blocks of the query fragment space were stolen by another query. Therefore, the query plan would need to be resent from the application to the DP2.

If a disk volume contains many different SQL objects (tables and/or indexes), query fragments from each SQL/MX executing process will be stored in the SQL/MX buffer for the DP2. Therefore, placing many SQL objects on a volume can potentially fill SQL/MX buffers. Monitoring SQL/MX buffer STATS through SCF is important. When FST or ID Failures occur, SQL/MX buffer thrashing is taking place. To alleviate SQL/MX buffer thrashing, avoid placing too many SQL objects on the same volume. Also, avoid placing frequently accessed SQL objects on the same volume.

Before monitoring SQL/MX buffer statistics for an application, the statistics should be reset. Use `SCF DISK STATS $vol, SQLMX, RESET`.

If SQL/MX buffer space is under-configured, these errors could occur:

```
8570 SQL/MX could not allocate sufficient memory to build query.
```

```
8571 SQL/MX could not allocate sufficient memory to execute query.
```

Increasing SQL/MX buffer space can help to eliminate this error condition.

Setting the SQLMXBUFFER Attribute by Using SCF

For data volumes with SQL/MX, you can change the size of the segment data area by changing the SQLMXBUFFER attribute. The volume must be in the STOPPED state to allow this attribute to be altered.

From SCF, set SQLMXBUFFER as:

```
ALTER DISK $volume, SQLMXBUFFER n
```

(where n is the unit in megabytes)

The minimum SQLMXBUFFER size is 1 MB, and the maximum size is 768 MB. The default value for SQLMXBUFFER is zero, which causes DP2 to automatically establish the appropriate size for SQL/MX data space. For volumes accessed by NonStop SQL/MX, use the default value (0) or values of at least 128 MB. Increasing SQLMXBUFFER beyond 128 MB effectively reduces the maximum space available for cache. You do not need to alter SQLMXBUFFER unless more virtual memory is needed for cache.

HP recommends that the SQLMXBUFFER size be divisible by 16.

For detailed information about the SQLMXBUFFER attribute, see the SCF Reference Manual for the Storage Subsystem.

Optimizing SQL/MX Memory Management

Certain embedded SQL/MX programs can use very large portions of the process' flat segment address range for:

- Application-addressable memory
- Memory space for the master executor to run SQL/MX statements

The memory space requirement for the master executor can be minimized by using parallelism to distribute the work to other processes, as explained in the discussion of parallelism in the SQL/MX Query Guide.

SQL/MX processes can consume large amounts of addressable memory space as a result of:

- Parallelism among a large number of ESPs
- Running plans that use sort and grouping operators
- The optimization of complex plans

The heavy consumption of addressable memory space by SQL/MX processes can lead to insufficient swap-file space. Therefore, you should provide more kernel-managed swapped space on each CPU by increasing the size of existing swap files or by adding new swap files.

You should periodically monitor your kernel-managed swap files while SQL/MX programs are being compiled and run to ensure that adequate swap-file space is available. If the required swap-file space is not available, an SQL/MX compilation might fail, or a running statement might return an “insufficient memory” error.

Use NSKCOM to do any or all these tasks:

- Monitor kernel-managed swap-file use
- Monitor swap-file use by process
- Change your KMSF configuration

For more information, see the Kernel-Managed Swap Facility (KMSF) Manual.

To identify and avoid possible memory contention between the application, the master executor, and other system components in process space such as QIO, see the discussion on configuring the QIO subsystem in the QIO Configuration and Management Manual.

Compiling Large Queries for Tables

The compiler can run out of memory if the plan search space is large. One workaround is to reduce search space by forcing a change in the join order, join strategy, and so on. To limit the frequency with which the optimizer searches for available plans, set the CONTROL QUERY DEFAULT OPTIMIZATION LEVEL. The lower the setting (0 is the lowest and 5 is the highest), the fewer plans are built and evaluated. For more information, see the SQL/MX Query Guide.

Using Multiple SQL/MX Database Catalogs and Schemas

You can create multiple SQL/MX catalogs and schemas for databases that are distinct from one another, such as those used for different applications.

Try to limit the number of separate schemas you create because too many separate schemas make managing and programming linking tables across multiple environments difficult.

Schema Ownership

ANSI schema-ownership specifications prohibit tables from being created in a schema by a user ID other than the owner of the schema. This requirement creates problems for ODBC tools, which depend on temporary table creation for intermediated query results. Here are two workarounds:

- Give users a SAFEGUARD alias that maps to the owner's ID. This arrangement allows for table creation by other users using their own alias, but the tables are actually created using the owner's ID. This workaround may not be the best solution depending on the security requirements of the customer.
- Use the undocumented PUBLIC_ACCESS_SCHEMA feature. The PUBLIC_ACCESS_SCHEMA must be created by the super ID (255,255) in the catalog that also contains the owner's schema. Any user ID can create tables in the PUBLIC_ACCESS_SCHEMA. Any references to those newly created tables must contain the schema name (three-part ANSI name) as well.

Table Creation in Third-party Tools, Where LOCATION is an Unknown

The LOCATION clause of the CREATE TABLE statement is not ANSI standard, so third-party ANSI-standard ODBC tools cannot specify the Guardian location of the table in the CREATE statement. If the location

clause is omitted from the CREATE TABLE statement, table creation defaults to the volume from which NonStop SQL/MX was started, which could be an unaudited volume. Some tools, like MicroStrategy, allow you to set a preference to append information to an SQL statement. You could use this capability to add the location clause. Not all tools allow this use. However, you can use this workaround:

Add a default location through MXCS:

```
Name: _DEFAULTS

Value: class defaults, volume $<vol>.<subvol>
```

This action forces all tables created through ODBC to be created in the defined volume, with the subvolume dependent on the schema subvolume.

Starting with SQL/MX Release 2.1.0 (SPR ABU), a new CONTROL QUERY DEFAULT called DDL_DEFAULT_LOCATIONS is available. This default permits defining a default location for tables and indexes created without a LOCATION clause.

MDAM

MDAM Query Technology Design Considerations

To exploit the capabilities of MDAM query technology, first you must know the data distributions and unique values for each key column in the table. Generally, you should order key columns to provide the most efficient and direct access into the table, based on the columns most frequently used by the queries. If you have a choice in determining the order of the key columns, order them by their unique value count, from least to greatest. For example, assume that the columns and unique entry (values) counts of the table are:

Column	Unique Entry Count
Customer key	1,000,000
Store key	500
Date key	1,825
Product key	50,000

In this table, the existing order of key columns might be appropriate if queries always provide the customer key when accessing this table, perhaps through a dimension table lookup. If the customer key and date key were supplied, but not the store key, MDAM query technology might probe through the store key values to access the qualifying data. However, if the customer key was not supplied, a full table scan might be used.

For a detailed description of the functionality of the SQL/MX MultiDimensional Access Method (MDAM) facility, refer to the document Efficient Search of Multidimensional B-Trees, written by the SQL/MP development group.

Forcing MDAM Query Technology

MDAM query technology can provide substantial performance benefits when used properly. However, MDAM cannot resolve all performance issues. Do not force MDAM query technology on all queries. The optimizer normally chooses the correct optimizations. Forcing MDAM query technology when the optimizer chooses otherwise may degrade query performance. Sometimes forcing MDAM query technology improves queries; however, before you take that action, you should have a thorough understanding of the data relationships and queries, and always perform extensive testing.

Possible Implementations

Join Efficiency

Primary-key order can have a large impact on join performance. A typical example involves a frequently used dimension table that is joined to a fact table.

Assume that a predicate is supplied on a column of the dimension table. If the join column in the fact table is the first key column, the SQL/MX optimizer may read the dimension table, select the rows that match the predicate, and then join directly to the fact table. However, if the join column is very low in the fact table key order and MDAM query technology cannot be used, the optimizer may scan the entire fact table and perform a hash join to the dimension table. This can be an intensive operation.

When you can identify dimension tables that are likely candidates for all or most query access, consider promoting the join column to a higher position in the primary key order of the fact table. It might be sufficient to place this column immediately after a column that has low cardinality, where MDAM query technology can be used.

When multiple tables (especially large tables) share similar key columns, consider using the same ordering and partitioning schemes for all tables. Joins between tables that have the same key and partitioning structures are very efficient because the query might benefit from partition-by-partition key-sequence merge joins rather than less-efficient repartition joins.

Insert Efficiency

Although insert efficiency is an important consideration for performance, usually insert efficiency should be secondary to other considerations unless specific circumstances indicate otherwise.

The choice of primary key determines the degree of parallelism available to a data-load application. For example, consider what occurs when SALES_DATE is the high-order column of a fact table's primary key. If data is always loaded in the order of SALES_DATE, the parallelism of the load application is limited to the number of volumes over which an individual date can be partitioned. Conversely, if the high-order column is ITEM_NUMBER, parallelism is inherently greater because the data are partitioned over many more volumes. One way to balance this situation is to use hash partitioning.

Usually load processing is a batch operation that occurs regularly (each day, week, or month) and is bounded by the time window available for the load. Query processing occurs at all other times.

Generally you should optimize query processing, which occurs frequently, rather than optimizing less-frequent batch load processing. Your circumstances may differ, however, so you must determine the tradeoffs that are appropriate for your application.

For example, for some applications, the batch load processing must complete within a certain time. In this case, you should optimize the batch load processing design for best performance, even if that means some tradeoff in query performance. See the "Fact table partitioning techniques" and "Dimension table partitioning techniques" sections of this document for more details.

Data Clustering

In addition to providing a fast and efficient access path, the primary key should be used to physically cluster the data. This arrangement is advantageous for sequential access methods in primary key order, which are often used in DSS query processing, because each I/O operation reads a large block of data from disk and transfers that data to cached memory. This enhances query performance.

When a primary key involves data that has a skewed distribution, system performance can degrade. To ensure a satisfactory distribution of data and provide optimal query performance, you must understand the distribution of all the column values that comprise the key. Specifying more precise partitioning values might be sufficient for dealing with skewed values. However, if the key contains values that are severely skewed, creating a hash partition for more uniform distribution may be easier.

Partitioning of tables and indexes is required if you want to take advantage of parallelism. HP recommends that you limit the number of partitions used, because processes perform better when they access fewer drives. This arrangement is different from the traditional SQL/MP method of database design. Normally, distributing partitions across many drives increases performance.

Fact Table Partitioning Techniques

Fact tables are often very large and, therefore, must be partitioned across multiple disk volumes.

To obtain good parallel scan performance, the partitions must be distributed over all processors. Ideally, the number of table partitions should be a multiple of the number of processors, and the partitions should be equal in size. Otherwise, scan performance is uneven and is dominated by the largest partition.

To avoid controller I/O contention, place multiple partitions within the same processor on disk volumes attached to different controllers, if possible.

You can partition tables in three ways. Each method relies on the standard range partitioning or hash partitioning capabilities of SQL/MX software. You must fully understand the advantages and disadvantages of each method at design time. The specific method used affects the load architecture, query performance, and degree of database maintenance.

To choose an effective partitioning method, you must first understand the nature of the data and the type of processing that will occur, both in batch and query.

Consider these questions when you decide which column or columns to use for partitioning and the order of the key columns:

- What is the expected rate and frequency of loads?
- What is the time window for the loads?
- Are data loads done in batch or online?
- Will new rows be added in sequential order or in random order?
- After the data is loaded, will it be processed in batch or by queries?
- Does the partitioning method aid the order in which the data is accessed?
- How often will queries supply predicates for the key columns, including the partitioning column?
- What is the data distribution and unique value count of each key column?
- Which key columns are likely to enable MDAM query technology processing?

A fact table is the central part of the star schema, often the schema of choice for databases. The granularity of the fact table is determined by the level of detail in the dimension tables. Each dimension is represented in the fact table by a dimensional value, which becomes one of the fact table key columns. The number of dimensions usually equals the number of fact table key columns. The primary key of the fact table is composed of the collection of dimension columns and uniquely identifies each row. Each fact table key column is a foreign key to a corresponding dimension table.

For example, a simple retail fact table may consist of three dimensions and has a grain of time, location, and product. Each dimension is represented in the fact table as an individual key column, and each row of the fact table includes all three dimensions.

If your fact table does not possess these characteristics—that is, if there are key columns in the fact table that do not reference dimension tables, or if there are data columns in the fact table that are foreign keys to dimension tables—review your design's dimensional modeling techniques.

In standard range partitioning, the leading column or columns partition the table horizontally. For example, if the chosen key order is location, time, and product, you could use location for the partitioning column. You would order the table data by location and distribute it across all disk volumes based on specific partitioning values.

Sequential Range Partitioning

Sequential partitioning means that the partitioning and clustering key values of all subsequent data added to a fact table are always greater than the partitioning and clustering key values in the existing data. That is, the partition range is continually increasing.

For example, sequential partitioning occurs in a table that uses a date for the partitioning column. In the retail fact table example, time (expressed by date) is the leading clustering key column and is also the partitioning key. All new rows added to the table have date values greater than the existing dates in the table. The partitioning range of this table increases continually as the dates increase.

Fact tables normally undergo an initial load, where the data collected to date is added to the table, and subsequent loads periodically add additional data.

Consider the impact of adding new data, deleting old data, updating data, and querying existing data. Also, consider the impact of these actions on database-management activities and backup-and-restore operations.

In this example, all new rows added to the table always fall logically at the end of the table, since new rows have dates later than dates in the existing rows. To achieve more even distribution, you should use hash partitioning, hashing on the date, to balance the load.

Note that since the newly added data is clustered around the partitioning value, parallelism during a load is limited or nonexistent. Also, a large data load may not complete within the time allotted.

Deleting old data may be as simple as dropping the oldest partitions, or using the SQL/MX software reuse-partition facility, which allows an existing partition to be reused by assigning a new partition value, enabling data to be rotated over a constant set of disk partitions. The reuse-partition facility is faster and more efficient than dropping old partitions and adding new partitions to a table. However, neither method requires queries to delete rows from the table to purge old data.

Usually, it is not necessary to update existing fact table data, because this data is historical in nature. However, when updating is necessary, its processing will be pseudo random at best, although it can be performed in parallel streams, processing each disk partition concurrently.

Since date is the leading key column in this example, and most DSS queries use a date predicate on historical data, some queries seldom benefit from parallelism. The requested data is often distributed over a few partitions or contained entirely within a single partition. This arrangement is especially common if many dates are contained within one partition. If many users query across the same date range concurrently, performance bottlenecks may occur on the processors and disk volumes being accessed, and hardware resources may not be used effectively. However, if the queries specify data ranges that are non-overlapping and diverse, this partitioning technique may be adequate.

Database-management activities are minimized when using this partitioning method. Partitions must be sized for the expected data volume for a specific date range. Once the data is loaded and FUP RELOAD has been run, no further management is usually needed other than purging the old data at the appropriate time.

Secondary indexes are seldom used on fact tables because MDAM query technology often provides efficient access to data even though the leading key columns may be excluded from the query. However, more complex data models that require many dimension columns in fact tables use secondary indexes. Note that LOAD and LOAD APPEND are not available in SQL/MX Release 2.0.

If you must drop and create new partitions on a regular basis, you should use the MODIFY REUSE command to modify the index.

Tape backups must be explicitly managed with the Backup utility. However, if previously loaded data is not updated, only the newly loaded partitions must be backed up to tape. Previous data partitions are unaffected.

Hash Partitioning

Hash partitioning is a new function available with NonStop SQL/MX Release 2.0. The partitioning column is selected from any of the clustering key columns of the table.

Hash partitioning uses a hashing key to randomly distribute the data. New rows are inserted within existing cluster key order within the volume, causing table fragmentation as existing blocks are split to make room for newly added rows. Online reloads are required periodically to defragment all partitions and maintain adequate scan performance. Note that all partitions become fragmented as new data is inserted into the table across the entire partitioning range.

This type of partitioning is not difficult to plan. The partition sizes are balanced, and individual partitions are not unexpectedly filled, provided that the unique entry count of the hash partitioning key is greater than 50 times the number of partitioning volumes.

However, delete processing of old data is complex, and must be accomplished through the use of queries. Scans of each partition can occur in parallel; however, each partition must be entirely scanned to locate old rows for purging.

Depending on the number of unique values for the clustering key columns that precede the search predicate (i.e. date column), you could use MDAM query technology processing to skip ranges of unaffected rows. Be aware that if delete processing is done through the MXCI utility (or other query tools), a transaction based on NonStop TM/MP software will exist for the duration of the query, and may be aborted due to the NonStop TM/MP software transaction timer. However, you can alter this timer before and after running the query.

Delete processing will acquire row locks that might escalate into partition or table locks, preventing other transactional access to the data. Managing delete processing programmatically gives you more control over the granularity of locks placed on the data. Make sure you understand the consequences of either technique.

Update processing requirements are similar to those of delete processing.

Query processing is more likely to use parallel execution, especially if the high-order clustering column is omitted from the list of predicates. In this case, you might use MDAM query technology to avoid full partition scans. Hash partitioning structures the data in each partition based on the clustering key which is suitable for query processing. It can take advantage of MDAM query technology processing capabilities. The impact of deleting and updating with hash partitioning is greater than that of sequential partitioning on database management.

You must schedule online reloads to run periodically to eliminate fragmentation and maintain good scan performance. All partitions may be online reloads because data will be inserted to every partition of the table. Hash data partitions must be managed carefully to avoid partition-full conditions, and to maintain the balance of data across partitions. You might need to run the online partition-management facilities periodically to rebalance the partitions.

Hash partitioning supports secondary indexes, if they are required on the fact table, because the data is added using insert processing.

Backup requirements also are complex. Since a load adds rows to all partitions, the entire table must be backed up. For small to medium-size tables, backing up the entire table or taking NonStop TM/MP online dumps, and completing within the availability window, may be adequate. For larger tables or applications that load data frequently or continuously, keeping copies of the original source data and reloading it when necessary may be more effective.

Sequential range partitioning and hash partitioning each have advantages. Sequential partitioning preserves table compactness and minimizes database maintenance—there is no fragmentation—while hash partitioning can evenly distribute rows across all partitions, reducing bottlenecks by increasing parallelism.

When queries access the data from the fact table, MDAM query technology probes for all values of the hash key, one for each partition, which is efficient for query processing. Queries are more likely to exploit the capabilities of the inherent parallelism. Additionally, a view can be constructed in place of the fact table, which forces a join between the partition table and the fact table on the hash key. This method incurs the cost of an additional join, but eliminates MDAM query technology processing, because the hash-key values can be obtained from the dimension table. This method has been shown to produce better results when multiple ESPs all begin MDAM query technology probing of the fact table at the same

time, starting with the same partition-number value. This occurs only when the fact table is not the first outer table of the query.

Other Benefits of Partitioning

The partition table has other benefits. If the partition table is selected as the outer table of the join, the partition table determines the number of ESPs the query uses. By creating the partition table using one partition for each processor, and by structuring the partition table so that the rows within each partition align with the corresponding fact table rows, the join performed by the ESPs accesses corresponding partitions within the same processor. This method provides efficient join-performance management of the ESPs.

New data loaded into the fact table can occur in parallel. Purge processing is similar to hash partitioning, but more efficient. Update processing is similar to sequential partitioning. However, MDAM query technology is used to locate the correct partition because this value is unknown to the application. Database maintenance is simplified because the amount of data directed to a set of partitions is controlled through the loading application, and each partition fills at the same rate. Online reloads are not needed as often since partitions do not become easily fragmented. Secondary index creation of hash partitioning is similar to that of sequential partitioning. The backup requirements are similar to those of hash partitioning, because all partitions are affected by data loading.

Hash partitioning offers very flexible implementation alternatives. You could distribute data across all partitions, or—when the data volume is relatively small—only a subset of them. This method is also adaptable to growth in the database. If the existing database is evenly divided across the processors, you can provide additional capacity by adding another set of disk drives (also balanced across processors), creating another set of partitions, and distributing new data over these new partitions. This method eliminates the need to rebalance the entire database.

Fact Table Partitioning Summary

Each partitioning method has advantages and disadvantages. You must have clearly defined objectives and requirements before finalizing your choice. You should have well-defined queries that accurately represent the business concerns of the users. Engage your users in this decision process; no IT department understands the business requirements better than the users do. If possible, construct a full-scale prototype and study the resulting query plans and performance characteristics to validate that the design will work as intended. The complexity of some database designs makes predicting query response time very difficult.

If a full-scale environment is not available, there are several alternatives. You could implant the SQL catalog and schema with production statistics so that the subsequent analysis of query plans is more accurately represented. This action can be done without loading any data at all. Then you can make an analysis of the query plans.

The goal is to validate that the physical design is used as intended. Once the query-plan analysis confirms the design, load some data into the tables to get an indication of what the response time will be, and how well the system will be used. Changing the physical design at this stage is much easier than changing the physical design after the application has been implemented.

You may need to do several iterations before you find an effective design, but the effort can be worthwhile, especially when done early in the project. Be aware, however, that no amount of effort can substitute for well-defined objectives and requirements, and for including the end users in decision making.

Testing the Results for SQL/MX Tables or Creating Artificial Statistics

To test the results of updating statistics:

1. Prepare a sample query from your application. Consider using a commonly used query from your application.
2. Use EXPLAIN to obtain the cost information for your query.

3. Determine the effect of the UPDATE STATISTICS statement, and, optionally, back out the generated histogram, if necessary.
   a. In MXCI, back up current histogram tables, if any:
      - CREATE TABLE myhist LIKE HISTOGRAMS;
      - INSERT INTO myhist SELECT * FROM HISTOGRAMS;
      - CREATE TABLE myhistint LIKE HISTOGRAM_INTERVALS;
      - INSERT INTO myhistint SELECT * FROM HISTOGRAM_INTERVALS;
   b. Issue the UPDATE STATISTICS command for the required column groups.
   c. Recompile the query.
   d. Use EXPLAIN to review the cost information and determine if MDAM (if desired) is used for your query. To determine if MDAM will be used, look at the results of EXPLAIN. MDAM occurs for the FILE_SCAN operator. When key_type indicates MDAM for the FILE_SCAN operator, MDAM query technology is used.
   e. If necessary, use the UPDATE STATISTICS CLEAR option to remove histograms for unwanted column groups.
   f. If necessary, restore the backup histogram tables.
      - `DELETE FROM HISTOGRAMS;`
      - `INSERT INTO HISTOGRAMS`
        ```
        SELECT * FROM myhist where table_uid in (select object_uid
        from CAT.DEFINITION_SCHEMA_VERSION_1200.OBJECTS);
        ```
      - `DELETE FROM HISTOGRAM_INTERVALS;`
      - `INSERT INTO HISTOGRAM_INTERVALS`
        ```
        SELECT * FROM myhistint where table_uid in (select object_uid
        from CAT.DEFINITION_SCHEMA_VERSION_1200.OBJECTS);
        ```

Dimension Table Partitioning Techniques

If you plan to use parallel execution (note that some applications are more suited to serial execution), plan to partition all dimension tables, even if they are small. This partitioning provides the optimizer with more opportunities to select effective parallel plans. Small tables can easily be changed to non-partitioned tables at a later time.

To take maximum advantage of parallel index updates, put a table's indexes on separate volumes and on separate CPUs to eliminate contention of parallel operations on indexes serviced by the same disk process.

If possible, take advantage of any opportunity to partition large tables on the same columns. Joins between these tables are more efficient than if the tables are unordered relative to one another.

Some customers have found it effective to use hash-partitioning techniques with dimension tables to eliminate data-clustering patterns that reduce the effectiveness of parallel queries. For example, if all (or many) of the predicate values for a particular column fall within one or two partitions, and this table is the outer table, only one ESP (or just a few) will find qualifying rows. Only this ESP will continue running the query, with diminished parallelism.

Managing Cache Memory Size

Cache is the buffer in memory that the disk process uses to keep copies of the disk blocks that have been accessed most recently. If the disk process finds a requested block in cache, the disk process can satisfy the request immediately without requesting a physical I/O operation.

Cache size has an important effect on performance. The larger the cache, the more likely that a block needs to be read only once.

To see if cache is operating efficiently, use the STAT option of the PUP LISTCACHE command. If CACHE READ HITS are less than 90 percent, consider increasing the cache size. If the ratio of CACHE FAULTS to CACHE CALLS is greater than one percent, consider reducing the cache size, adding more physical memory to the CPU, or processing to other CPUs.

Cache size is controlled through the Subsystem Control Facility (SCF). For more information about using SCF to set cache size, see the SCF documentation.

Maximizing Disk Process Prefetch Capabilities

NonStop SQL/MX can enhance performance by reading blocks of data into cache asynchronously before data is needed. This disk-process prefetch operation works best when you request long sequential scans through data or when your access plan has a low selectivity value (as described in the SQL/MX Query Guide).

The optimizer requests sequential prefetch for all scan operations expected to read sequentially for more than a few blocks.

When sequential prefetch is used, the disk process attempts to use a single I/O operation to read a group of several consecutive blocks. The successive read operations do not have to wait for physical I/O and can be satisfied from cache, in parallel, while the disk process performs other I/O operations. To determine if your query uses sequential prefetch, look for the words sequential cache in the EXPLAIN output for the query.

You can perform a prefetch operation for forward processing for certain types of operations, such as scans, updates, and deletes of subsets, and for disk operations using virtual sequential block buffering (described in the SQL/MX Reference Manual).

To maximize disk process prefetch operations, use:

- Large cache.
- Mirrored disks.
- Well-organized, key-sequenced tables (whose physical sequence closely maps to their logical sequence). The FUP RELOAD operation can help reorganize an existing table.
- Multiple PINs. DP2 automatically increases the number of PINS to six when an SQL/MX query is started and prefetch is used.

To check whether the disk process uses prefetch capabilities for your queries, set statistics on, and use the SCF STATS disk command and the Measure DISK and DISKOPEN entities.

Column Alignment

It is a good practice to align each column on a word boundary whenever possible. Data types vary in length and may not always be word-aligned. Numeric values are always word-aligned, and vary in size from 2 bytes to 4 bytes and 8 bytes. Character columns occupy the number of bytes specified in the declaration. For example, CHAR (3) occupies 3 bytes. If CHAR (3) is placed between two numeric columns, 1 byte is wasted to align the second numeric column in the program's memory structure. If an even number of odd-length columns appears in the row, place them together because they will terminate on a word-boundary, wasting no space.

However, always select the partition and key-column order based on the access path required for the table. Column alignment is secondary.

These are other good practices:

- If a numeric column will contain only positive values, use an unsigned numeric.
- Avoid the use of DECIMAL data types. This is an ASCII character representation of a number that will need to be converted to binary for arithmetic operations.
- Align VARCHAR columns at the end of the row.
- Avoid using VARCHAR for small length columns.

- Avoid the excessive use of CASE statements, data type casting, and function calls within a SQL statement if these features can be performed within the logic of a program. Often compiled application logic can perform these features more efficiently.

Managing and Monitoring

Catalog and Schema Placement and Maintenance

Maintain separate catalog and schema for test and production purposes. Drop catalog and schema entries when they are no longer required. Keep production catalog and schema free of unnecessary entries.

Keep catalog and schema fragmentation to a minimum by regularly performing online reloads. Do an online reload after the initial creation of the database and after tables or partitions have been added and dropped.

Statistics for catalog and schema cannot be maintained.

These measures will ensure the most efficient access to the catalog and schema tables.

Data-loading Techniques and Considerations

This section suggests procedures to load data from a flat file to SQL/MX tables. Your procedures may vary depending on your circumstances.

Create the Table for the Final Destination of the SQL/MX Table

SQL/MX tables cannot be renamed; if you want to partition SQL/MX tables, partition them initially. Do not partition smaller tables that may be best served by a single drive's cache.

Do not create the table along with its indexes. Load the data first, and create the indexes after you load the data. The exception is when you need a unique index to create a data constraint prior to the data load. However, note that load performance will be affected. Review how to create a primary key if the primary key differs from the store-by value. To partition the primary key when it differs from the store-by value, create the table with a unique index that has the values of the primary key, then issue the command for the primary key. This method will reuse the value for the index and make that value the primary key.

To use IMPORT, create or obtain a delimited file. Create the file with columns ordered in the same order as the table that you will be populating. Choose a delimiter that is not in the data content of the file you wish to import. Currently, a problem occurs in IMPORT when IMPORT encounters special data, even when the value is flagged as a special character.

Use FTP to Send the File to the Desired Location

Verify that FTP is completed before starting the IMPORT. IMPORT does not issue a warning or error if the file being sent is still open when IMPORT is issued or running. Neither FTP nor IMPORT fails if they both have the file open. This situation can result in not loading all the data from the input file.

Import the File to the Table

Create indexes on the table, and partition the indexes if desired.

Select Guardian names from the SQL/MX catalog for your table or the tables you wish to reload.

FUP RELOAD the Guardian names of the table (include all partitions of a table and any indexes or index partitions). You can run these commands concurrently; however, you should distribute the FUP RELOADs across CPUs at a low rate to ensure that the reloads do not monopolize the system resources.

Select the count(*) Table for Read Uncommitted Access

For the UPDATE STATISTICS command, you need to know how many rows were loaded during the IMPORT or the data insert. However, you do not need to select the count(*) table if you note how many rows were loaded during the IMPORT or the data insert. UPDATE STATISTICS can obtain its own row count of the table with its algorithms, although doing so is inefficient. Your statistics will run faster if you provide the row count manually as part of the UPDATE STATISTICS command.

The suggested values for the syntax:

```
update statistics for table on every key sample random 10 percent, set rowcount
'count from table'
```

If the table is medium to small in size, you may run statistics on all columns without sampling; however, using the row count will increase performance.

No bulk load command is currently available; you must use an INSERT…SELECT statement. For parallelism, you can use an INSERT with a SELECT FROM and describe the range and the data requested. From NonStop SQL/MX SPR ABA and superseding SPRs, the INSERT…SELECT option works well. The other options

for data load are IMPORT, the DataLoader/MX programming API, custom programming, or third-party ETL tools.

Getting Guardian Names from the System Catalog for FUP RELOAD

Note that definition_schema_version_1200 is correct for now; however, in the future, there will be version numbers higher than 1200.

```
set schema definition_schema_version_1200;


select 'fup reload ', substring(p.system_name, 1, 15) ,
    '.' , rtrim(p.data_source) ,
    '.' , rtrim(p.file_suffix) ,
    ', rate 40'
from    partitions p,  objects o
where   o.object_uid  = p.object_uid
and    o.schema_uid   =
    (select schema_uid
    From nonstop_sqlmx_SYSTEMNAME.system_schema.schemata
    Where schema_name = 'SCHEMANAME'
    And cat_uid =
        (select cat_uid
         From nonstop_sqlmx_SYSTEMNAME.system_schema.catsys
         Where cat_name = 'CATALOGNAME'
         For read uncommitted access
         )
     For read uncommitted access
     )
and    o.object_name not like 'HISTOGRAM%'
and    o.object_name not like 'MVS%'
and    o.object_name_space in ('IX','TA')
order by p.system_name, p.data_source, p.file_suffix
       for read uncommitted access;
```

Note: schemaname and catalogname in the query are internal format ANSI identifiers. Regular identifiers must be specified in all uppercase. Delimited identifiers must be specified without surrounding double-quotes.

Block Slack Space

Scan performance for database queries depends largely on data compaction—that is, how much data a single I/O request will return. This, in turn, depends on the degree of table fragmentation, internal block free space, and row size. The more compact and organized the data blocks are, the better the scan performance will be.

One way to ensure that data blocks are as compact as possible is to reload the table data, using Slack to 0. SQL/MX software uses a default Slack parameter of 15 percent, meaning that 15 percent of the

block is reserved for future inserts. Because most fact tables are loaded once and seldom updated, this free space is never used. Furthermore, each I/O reads less data. Setting the Slack to 0 ensures that all available space is used for data. This parameter is specified in the FUP RELOAD utility.

Reorganizing Tables

For a database table to deliver good query scan performance, it must be reorganized periodically. Table reorganization arranges the physical data and index blocks efficiently, and compacts each block so the block is completely full. As blocks are scanned for a subsequent query, their physical ordering on disk matches their logical access order, and each block contains the maximum number of rows. This arrangement ensures the highest data-scan rate possible, because the disk subsystem can use optimized data-access facilities to retrieve the data.

Tables usually become fragmented when rows are randomly inserted into a table, causing blocks to split and decreasing the effective use of space. Additionally, the physical ordering no longer matches the logical ordering.

Tables that experience random or pseudo-random inserts, random deletes, and updates that change row lengths need to be reorganized periodically to ensure maximum scan performance. (Tables used in database applications are often reloaded with zero slack space to provide maximum scan performance.)

This FUP command is useful for determining which partitions of a table to reorganize:

```
FUP INFO tablename, STAT
```

Only those partitions that show fragmentation should be reorganized. The Tandem Reload Analyzer (Reload Analyzer) can also help determine what files need to be reloaded.

Table reorganization in NonStop SQL/MX is an online maintenance function that provides full read and write access during the process, using the FUP RELOAD utility. Many customers schedule batch jobs to perform regular online reorganizations. The database administrator should maintain a list of tables that do or do not require periodic reorganization, and a schedule of when reorganization should be performed. An exhaustive verification of which tables are fragmented should be initiated to ensure that the fragmentation does not contribute to poor performance.

Run FUP RELOADs before updating statistics. Updating statistics can be influenced by the index level of the file. You can reduce the index levels by using FUP RELOAD. FUP currently recognizes only Guardian names; it does not recognize ANSI names. ANSI names are the only names recognized in the SQL/MX environment. You can obtain the Guardian names for FUP RELOAD by using a SHOWDDL or SHOWLABEL for the table. To automate the process, or to obtain multiple entries to process, you can obtain the information about the Guardian names from the SQL/MX catalog.

All SQL/MX database and ODBC Server catalog tables should be reorganized, especially after many tables have been created or made available to ODBC. This action can improve SQL/MX database compile performance and ODBC Server access.

Some database functionality cannot be done online while data is being accessed by queries or applications. For example, you cannot create indexes online or move a partition while the partition is in use.

The built-in delay within a TMF transaction (for boxcarring) allows TMF to wait for additional information that can be combined within the transaction coming, and can reduce messaging within the system.

When collapsing a partition, you should reload data without slack both in the source partition and the destination partitions. This action aids in accommodating the data into the table. Collapsing from two partitions to one can result in a failure if you have a version of NonStop SQL/MX earlier than the ABJ SPR.

When migrating from one environment to another, use caution when recompiling prototypes-code from source code. Recompiling only the executable code that contains prototypes sometimes causes the code to attempt to access the previously compiled environment linked for the code.

Make sure that you do not have a “Halloween” problem when you use cursors. This problem occurs when you update information that could place the row forward into the selection path, thus making the row

appear again for processing. You may want to place the row forward into the selection path, but be aware that doing so can cause erratic cursor behavior and is hard to trace.

nohup (no hangup) is the OSS option that bests represents the Guardian NOWAIT option. For more information about this option, seethe Open System Services Shell and Utilities Reference Manual.

To find out if the file is full, issue the SHOWLABEL tablename command to obtain file extent information and the table-percentage full.

To move a partition location, enter the key word LOCATION in both the source partition and in front of the destination location.

Updating Statistics

Statistics are the single most important topic in database management. All tables, even if they are empty, need to have their statistics generated. Statistics influence almost every aspect of the query plans to be generated by the optimizer.

You cannot update statistics on system metadata tables, including tables residing in the DEFINITION_SCHEMA, MXCS_SCHEMA, SYSTEM_DEFAULTS_SCHEMA, and SYSTEM_SCHEMA.

Use the SELECT statement to retrieve the statistics in the SQL/MX HISTOGRAMS table generated by the UPDATE STATISTICS statement, based on table and column names:

```
SELECT O.OBJECT_NAME
     TABLE_UID,
     C.COLUMN_NAME,
     H.HISTOGRAM_ID,
     H.INTERVAL_COUNT,
     H.ROWCOUNT,
     H.TOTAL_UEC,
     LOW_VALUE,
     HIGH_VALUE
     FROM
     cat.DEFINITION_SCHEMA_VERSION_1200.OBJECTS O,
     cat.DEFINITION_SCHEMA_VERSION_1200.COLS C,
     cat.sch.HISTOGRAMS H
     WHERE O.OBJECT_UID = H.TABLE_UID
     AND O.OBJECT_UID = C.OBJECT_UID
     AND C.COLUMN_NUMBER = H.COLUMN_NUMBER;
```

Various statistics about table-data content and size are stored within a SQL/MX database catalog. This information is used by the SQL/MX database optimizer at query compile time to develop the most efficient access path to the data. The statistics are produced as a result of running the UPDATE STATISTICS utility against each table. Normally, statistics are updated after the database has been initially loaded with data, and thereafter when the volume or range of data changes significantly or after the addition of secondary indexes.

The SQL/MX database optimizer cannot produce an effective access plan unless the statistics adequately represent the data within the tables. When no statistics are present, the optimizer assumes default values, but these values might not produce plans that provide adequate performance. Indexes might not be used properly without updating statistics on the columns involved in indexes. This can be done by naming these columns and columns sets (for multi column statistics) explicitly in the update statistics statement or by using the ON EVERY KEY option of update statistics that would update statistics on all columns involved in any table indexes including the clustering index of the base table.

You should also generate statistics for all SQL/MX database and ODBC Server catalog tables (and indexes), because generating statistics improves SQL/MX database query compile time and ODBC Server performance.

This example runs the UPDATE STATISTICS statement on a large table:

```
UPDATE STATISTICS FOR TABLE tablename

SAMPLE RANDOM 10 PERCENT SET ROWCOUNT rowcount;
```

Using the UPDATE STATISTICS statement is a processor-intensive operation. To limit the impact on the system, execute this statement with a low priority so the system can manage the mixed workloads.

By default, UPDATE STATISTICS ON EVERY KEY gathers statistics for all key columns, including indexed columns. To gather statistics for all columns, use the ON EVERY COLUMN option.

Information to Aid "Implanting" SQL Statistics

1. Set catalog statistics.
2. Update the row count for each partition of the table.
3. If the root partition is empty, add an additional statement that sets the rowcount to 0 for that partition only.
4. Assume that no blocks are empty. Estimate the NONEMPTYBLOCKCOUNT by dividing the EOF of the partition by the block size. Note that EOF refers to the partition EOF.

Caution: The unique entry count (UEC) needs to fall within the range indicated by second low/high values. Otherwise, the selectivity is based on the range and not on the UEC.

For large tables, INDEXLEVELS is usually not greater than 3 (or 4).

Indexes are not recognized by the compiler until UPDATE STATISTICS has been run for the corresponding columns in the index. This action can be done by using the ON EVERY KEY statement.

Statistics are cumulative. For example, you can run statistics ON EVERY KEY for a table. Then, if you have a query that would perform better with statistics on other additional columns, you can update statistics on those column combinations. In a larger table, you should use sampling, statistics on keys, and rowcounts. An average of 10 percent is a good average to use when sampling. However, note that when sampling more than 50 million rows, using 1 percent also gives you a good sampling.

Use of the sample and random parameter aids in the performance of the UPDATE STATISTICS operation. This use allows general information to be generated about the data content, without having to read every row.

You must know and use the rowcount. If you are unaware of the rowcount of the table, run SELECT COUNT(*) on the table prior to generating statistics. UPDATE STATISTICS uses certain algorithms to determine sampling for the table, and one of those parameters is rowcount. If the UPDATE STATISTICS operation does not have the specified rowcount, it makes default assumptions that might not perform well for the amount of data.

Publish and Subscribe

This section compares the use of publish and subscribe against the use of queue files.

- SKIP CONFLICT ACCESS is available through publish and subscribe, but not through queue files, which is a performance advantage that supports scalability and avoids hotspots.
- Queue files must be accessed through Enscribe, but publish and subscribe can be accessed through ODBC, JDBC, and embedded SQL, and is also supported through JMS.
- Queue files do not support partitioned tables, but publish and subscribe does (note that for good performance, most publish and subscribe applications should access only one partition).
- Publish and subscribe supports nondestructive streams. You need not delete the row that is being read, as is required for queue files.

Implementing an ODBC or JDBC Architecture

For a detailed discussion of various considerations for managing the ODBC and JDBC environment, refer to the SQL/MX Installation and Management Guide.

Hardware and software requirements for the JDBC Driver for NonStop SQL/MX are described in the softdoc file on the NonStop Server for Java product CD, with which JDBC/MX is delivered. Read that document before installing the product.

Performance Tuning

Improving Performance

This section discusses some of the common performance problems associated with large databases. Additional information exists in other manuals and is not repeated here. Specifically, the SQL/MX Query Guide contains several chapters dedicated to analyzing and improving query and database performance, and the SQL/MX Installation and Management Guide contains chapters on measuring and enhancing performance. Be sure to read these manuals for a thorough understanding of the material, and to gain an understanding of the tools and techniques available for performance analysis.

When you must tune a database, you must consider several tradeoffs. For example, it is impossible to optimize all queries or queries and batch-loading workloads. One of the first steps involved in tuning a database is to determine the most important objectives. You also must be prepared to replicate certain portions of the database through the use of aggregate tables, or to create secondary indexes.

Addressing Problems with Numerous ESP Processes

Large databases with tables that have many partitions can sometimes experience problems related to the large number of active ESP processes. Relatively few active ESP processes can saturate a processor, though how many varies with the processor technology and query workload.

Minimize Table Partitions

Although a large database almost always uses partitioned tables, try to minimize the number of table partitions, and keep the partitions to a multiple of the number of processors. When possible, use one or two partitions for each processor. To minimize successfully, you need to understand the growth requirements and volatility of the table. For example, a table that is partitioned, organized, and loaded by date would be a good candidate. Once loaded, the partition sizes would not change. A table like this could be sized with the minimum number of partitions required to hold the data during its lifetime. In addition, it would be advantageous to round the number of partitions to a multiple of the available processors, and balance the data across the partitions. This would provide the most balanced and efficient scan performance while minimizing the number of ESP processes.

Tables that are partitioned and organized differently (such as in hash partitioning) are volatile because blocks split during random inserts. You must establish a schedule for reloading these tables and understand the variability of the space usage. The same approach can be taken to minimize the number of partitions.

Minimizing table partitions also minimizes the number of active ESPs used for queries, and helps eliminate resource contention. Note that if overly busy processors are a pervasive problem, you may need additional hardware resources. Attempting to tune an undersized system provides little gain.

Recommended Maximum Number of Partitions

HP recommends that you limit the number of partitions in an SQL/MX table or index to no more than 512. If you exceed this recommended limit, you may get an MXCMP internal error because of a shortage of virtual memory space.

Addressing Problems with Skewed Data Distributions

Tables sometimes contain columns that have significantly skewed values. For example, a customer had a date column with a range that exceeded 100 years, although the overwhelming majority of dates occurred within the last several years, and the current time period was the most often queried. Since the optimizer assumes an even distribution, a range predicate of the form “DATE between “1999-01-01” and “1999-12-31” caused the optimizer to expect that fewer than 1 percent of the table’s rows would qualify for the predicate. In fact, closer to 20 percent of the table’s rows qualified for the predicate. In many queries, the optimizer often expected only about 100,000 rows from a table when several tens of millions actually qualified.

The optimizer chooses specific access plans in subsequent steps of the query based upon the number of rows expected from earlier steps. When the difference between the predicted and actual values is as great as several million, the subsequent query steps often cannot deliver adequate performance. In cases of severe data skewing, the optimizer has little chance of producing a fair plan. The optimizer might choose a plan that is fast and efficient for small quantities of data but very expensive when large results are returned unexpectedly.

This problem was relatively simple for the customer to correct by explicitly setting the range of values in the catalog to more accurately represent the volume of data that would be selected.

Skewed data also can occur as data is loaded into the fact tables. Another customer had a table that used a partition number column as the partitioning key, over which data rows were distributed. The original intent was to round-robin rows over the range of partition numbers during the load processing each month. However, the actual distribution of values was less balanced, with some partitions holding several times more values than other partitions. This situation meant that ESPs accessing a large range within one partition delayed the processing for the entire query. This situation also caused an imbalance in the system use because only a few processors and disks worked on the query while others became idle within a short time.

The original design was based on distributing rows over a subset of data volumes, for example, one for each processor. Other customers have used the same approach with fairly good results. The ideal way to load the data is to distribute the data to be queried as evenly as possible across all processors, and possibly across all partitions.

Opinions differ regarding whether the data should be distributed in striped sets (where each set consists of one disk partition for each processor), or across all partitions for the table. One benefit of using all partitions is that increasing the number of processors does not invalidate the original partitioning strategy. Another advantage of using all partitions is that each partition is smaller than if the data were clustered over a few partitions, and more of the partitions are likely to contain the requested data. If some partitions are skipped because of skewed data, it is less likely that only a few ESPs will contribute to the query.

Another common problem involving skewed data arises when the dimension data distribution is skewed over the predicate column or columns. This problem is most apparent when the dimension is selected as the outer table.

As the outer table, each ESP reads its partition to select qualifying rows. If most rows are clustered within a single partition, one ESP has the majority of rows to process. This plan effectively degrades into a serial plan.

The solution involves either indexing the dimension table (and the optimizer selecting a different plan) or distributing the dimension rows to eliminate the clustering effect.

Addressing Problems in Large Tables

Large tables present unique challenges for performance optimization. These problems arise in several ways:

- Access through non-key columns
- No use of access through a subset of the primary key and MDAM query technology
- Frequent access to detail data that is aggregated
- Joins between large tables

The first two problems can be addressed by creating secondary indexes on the tables. Refer to the “Design guidelines and considerations” section of this document for additional considerations for secondary indexes. Use the SQL/MX database runtime statistics (the DISPLAY STATISTICS command) and DISPLAY_EXPLAIN output to determine whether secondary indexes would benefit query performance. Consider creating a secondary index when the DISPLAY_EXPLAIN output indicates that a column predicate having a low selectivity will result in a table scan and the runtime statistics indicate a large difference between rows accessed and rows used.

Queries that frequently read large tables and aggregate the results to a high level will benefit from the creation of summary table or tables. The “Design guidelines and considerations” section of this document includes additional considerations for summary tables.

Joins between large tables can be quite expensive, in terms of both processing time and resources. This issue is often one of the most significant issues faced by customers in the database environment. When possible, use similar partitioning and primary-key structures, which enable efficient partition-to-partition joins.

Database applications usually have at least one large dimension table that must be joined to the fact table. In this situation:

- Ensure that the dimension table is indexed as needed to avoid full table scans.
- Ensure that the statistics are updated and correct.
- Deal with skewed data distributions as described in the “Addressing problems with skewed data distributions” section of this document.
- Note whether the join column or columns between the dimension table and the fact table represent an efficient path.
- Review DISPLAY_EXPLAIN output to ensure that an efficient join method exists to the fact table. If the DISPLAY_EXPLAIN indicates that the type of join is a repartitioned hybrid hash join, the join will be a costly operation and response time may be excessive. Repartitioned hash joins are used when the join columns are not in the same order. It is preferable to use a sort-merge join, but it is still a costly operation, especially when run concurrently by multiple queries.

If none of these methods improves the access plan, you may need to restructure one or both tables, create an index on the fact table, or create an aggregate table, if reasonable, to improve performance. Of course, review the query objectives to determine whether a more efficient query can be developed prior to making any database changes.

Another common performance problem involving two large tables is one where a join is made between a large dimension table and a fact table through a higher-level dimension table. Consider this situation: A customer dimension table has a column called “customer_demographics,” which is further described in a small dimension table called “demographics.” The customer table has a normal foreign-key relationship to the fact table. A user submits a query that sums a fact column but reports the results by the demographics description. In this case, a large join is required between the customer and fact tables, in order to relate the demographics information to the fact table data.

Query performance could be improved greatly by treating customer_demographics as a separate dimension on which fact data can be analyzed. The join involves a small table, demographics, and the fact table, and provides much better performance than in the preceding case.

Look for opportunities that limit the joins between large tables.

Addressing Problems in SQL Catalog Performance

These measures will ensure the most efficient access to the catalog and schema tables.

- To avoid I/O contention during query compilation, customers who expect a high degree of concurrent query execution should plan to keep the SQL/MX database catalog and schema on a disk volume separate from the ODBC/MP Server catalog. (Note that NonStop ODBC/MX does not have a separate catalog; it uses the SQL/MX catalog.) Do not share a catalog volume with active data tables or with any files that are likely to experience high levels of I/O activity.
- Maintain separate catalog and schema for test and production purposes. Delete catalog and schema entries when they are no longer required. Keep production catalog and schema free of unnecessary entries.
- Keep catalog and schema fragmentation to a minimum by regularly performing online reloads. Do online reloads after the initial creation of the database, and after tables or partitions have been added and dropped.
- Keep catalog and schema statistics current for all catalog and schema tables. Update statistics regularly, both after the initial creation of the database and after tables or partitions have been added and dropped.

Analysis Methodology

Methodically follow these steps as a guide for improving query performance:

- Identify several key queries suffering poor performance, and use them as a test suite to evaluate the effectiveness as the changes discussed later in this document are made. Obtain current DISPLAY_EXPLAIN output for each query.
- For ODBC Server clients, when predicates for date columns are present in the query, replace them with parameters ("?p"), to mimic the actions of the ODBC Server subsystem. This action can affect the resulting plan.
- Obtain runtime statistics for each query, using the MXCI SET STATISTICS ON option. If a query returns a large amount of data but you are only interested in obtaining the runtime statistics, use the [LAST 0] clause in the SELECT statements. For example, SELECT [LAST 0] * FROM MYTABLE; will return zero rows, but still show the runtime statistics.
- Identify all predicate columns and join columns for each query. These predicate columns will become the entry points into each data table.
- Ensure that all tables are loaded with zero slack space, or are regularly reorganized online to eliminate fragmentation and to ensure optimal scan performance.
- Update statistics for all tables after they have been loaded, after altering partitions, and after adding secondary indexes.
- Obtain data-value distributions for critical key columns to determine the existence and pervasiveness of skewed values. Critical key columns are those used for partitioning, MDAM query technology, query predicates, and joins.
- Work with HP support representatives to establish queries that change statistics, where needed. Re-run these queries after using the UPDATE STATISTICS statement.
- Regenerate Explain plans for the query suite and determine whether any of the plans have changed.
- For queries that have new plans, re-run the queries and obtain runtime statistics. Determine whether the performance improvement efforts have been effective.
- For the remaining poorly performing queries, analyze the data-access points (predicate and join columns) for each table.
- Consider adding a secondary index.
- Consider adding a summary table.
- Consider reordering key column positions.
- Consider restructuring tables, either combining tables to avoid joins, or dividing a single table to reduce the number of rows.
- Always regenerate Explain plans, and re-run the queries to determine the effectiveness of each change.
- Modify the database design to accommodate the query requirements.

The main objective of these analysis steps is to provide the most efficient access path to the requested data. Because of the skewed data distributions, the SQL/MX database optimizer cannot select an adequate access plan. Consequently, evenly distributing data is the first change to make. After balancing the data, you must identify existing entry points into the tables and determine whether a secondary index, a summary table, or key-column reordering would improve response time. The most extreme action involves physically altering the table structure in some way. Although physically altering the table structure is normally the last course of action, it can produce the most profound query improvements.

CONTROL QUERY SHAPE

Controlling the query shape is a last resort for changing access to the data. This method forces the query to run exactly the plan you determine is best. If the table ever changes, (for example, the table acquires a new index, partition, and so on) the plan you forced may no longer be efficient. Any forced queries you make must be tracked manually. No automated way exists to identify queries that have been

forced. You can use the DISPLAY USE OF command for programs, but that command does not list MXCI, ODBC, or other dynamic queries that may be forced. Be careful if you use CONTROL QUERY SHAPE because doing so could cause more problems in the long run for database maintenance and performance. Using CONTROL QUERY SHAPE forces the same query even if the database changes. If the database changes, a more efficient query may be available through normal use of the compiler without using CONTROL QUERY SHAPE.

Contributors to this Document

- Doug Tully. Provided general information about SQL/MX environments. Acting coordinator of the Best Practices content providers for NonStop SQL/MX.
- Paul Denzinger. Provided information about ways to partition tables and general information about SQL data warehouses. Acting coordinator of the Best Practices content providers for NonStop SQL/MP.
- Gary Grosch. Author of the "Solution Factory Data Specification" document referenced here.[1]
- Todd Gunn. Provided information about managing the Open Database Connectivity (ODBC) and SQL database environments.
- Rohit Jain. Author of the "Physical Database Design for DSS" document referenced here.
- Marsha Lee. Provided information about creating, partitioning, and managing SQL tables in a decision support system (DSS) environment, and provided examples.
- Tom McGrath. Provided information about managing a DSS project, including roles and responsibilities.
- Jim Mills. Author of much of the "SQL ODBC" document referenced here.
- SQL/MP and SQL/MX Development. Authors of the "Efficient Search of Multidimensional B-Trees" document.
- Many thanks to all the additional reviewers and contributors that have been valuable in compiling the information for this document.

For more information

Website: www.HP.com.

[1] This and certain other documents referenced in the current document are not yet posted on the Web. Some are available through the HP NonStop Technical Library (NTL) product. If the author's name is listed, please contact the author at First_name.Last_Name@HP.com for more information.