

HP NonStop SQL Software best practices



Objectives of this document	3
Design guidelines and considerations	3
Physical system configuration	3
Logical database design considerations	3
Physical database design considerations	4
Primary key selection	4
Fact table partitioning techniques	6
Dimension table partitioning techniques	11
Column alignment	11
Data compression	11
Catalog placement and maintenance	11
Database sizing considerations	12
Database sizing spreadsheet	12
General rules for sizing	12
Work and swap space considerations	12
MDAM query technology design considerations	13
Data-type considerations	14
Consideration for summary tables	14
Considerations for secondary indexes	14
Using NonStop SMF Software to distribute catalog tables	15
Design techniques to avoid	15
Joining tables on calculated columns	15
“Over-normalizing” tables	16
Forcing MDAM query technology	16
Load-only optimized design	16
Built-in application bottlenecks: inadequate design analysis and testing	16
Data loading techniques and considerations	17
Block slack space	17
Database management activities	17
Reorganizing tables	17
Updating statistics	18
Issues involving very large databases (VLDBs)	18

Data locality objectives	18
Maximum table size	19
Number of table partitions	19
Formula for calculating the maximum number of partitions	19
Practical limitations	19
Techniques for creating large tables	19
Using a single Create Table statement	19
Increasing heap memory space	22
Increasing process file segment size	22
Compiling queries for large tables	22
Special NonStop SQL/MP defines	22
Using multiple NonStop SQL/MP Database catalogs	22
Maximum catalog size	22
Improving performance	23
Managing numerous ESP processes	23
Reduce table partitions	23
Increase the ESP startup timeout	23
Use early cross-products	23
Addressing skewed data distributions	24
Addressing large table issues	24
Addressing NonStop SQL/MP catalog performance	25
NonStop SQL/MP defines for special circumstances	26
Query performance analysis methodology	28
Implementing NonStop ODBC/MP architecture	29
For more information	31

Objectives of this document

The purpose of this document is to describe the leading best practices, gained from time-tested customer experiences, for designing, implementing, and supporting large decision support solution (DSS) databases with HP NonStop SQL/MP Software. Many of these techniques are useful for any type of large database environment, including data warehouses, operational data stores (ODS), and online transaction processing (OLTP) applications, as well as for smaller data warehouses.

The information described in this paper should be used in conjunction with other information resources, such as the NonStop Technical Library, specifically the NonStop SQL/MP and NonStop ODBC/MP server manuals.

The techniques and practices described in this document apply specifically to the NonStop SQL/MP and NonStop ODBC/MP products. Although the general techniques are applicable to the NonStop SQL/MX and NonStop ODBC/MX products, the specific implementation details differ to such a degree that they must be described separately.

Design guidelines and considerations

Physical system configuration

Balancing and tuning complex DSS applications is a very challenging task, which is made more difficult when the underlying hardware is unbalanced. Systems with an uneven number of disk volumes per processor create a natural imbalance in resource utilization that is difficult to tune. Having only a few active processors for the workload may result in long queue lengths, which degrades performance for any process running in those processors.

When a parallel query begins, groups of processes known as executor server processes (ESPs) each execute a subset of the query. The results are later combined and returned to the user. If a processor is overly busy, the ESPs running in it cannot make adequate progress servicing the query plan. Even though all other ESPs may have completed their work, the ESPs in the overly busy processor delay the overall completion of the query.

To fully realize the potential of the NonStop system, it is important to take full advantage of its parallel and scalable architecture, as follows:

- Distribute the disk subsystem across all processors to balance more evenly the I/O processing activity and thereby reduce 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 system may require additional I/O cabinets to achieve this balance.

Balance resource utilization and queues. The goal is to eliminate the resource bottlenecks—usually at the processor and disk levels. These queues impact response time, and result from excessive workloads concentrated on a few hardware 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.

Logical database design considerations

For a detailed discussion of designing and implementing a DSS application on NonStop servers, refer to the document *Physical Database Design for DSS*.

Physical database design considerations

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. Because 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 will have to decide 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, there are usually fewer columns, and 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.

The most important considerations for primary key selection are as follows:

- Data access efficiency
- Cardinality
- Partitioning
- Join efficiency
- Load/insert efficiency
- Data clustering

Data access efficiency

As described earlier in this document, *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, because these will restrict the search space for the majority of DSS 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 it is included in several million fact table rows. Hence, its cardinality is 10.

Cardinality is an important consideration for MDAM query technology processing. Low-cardinality columns tend to enable its use. It is most useful for avoiding full-table scans when the query omits predicates on leading key columns, and for processing “in-lists.”

When you have a choice, place low-cardinality columns as high in the key order (left most) as possible, and in increasing order of cardinality. This provides the greatest degree of MDAM query technology processing when high-order key columns are omitted from the query. You can place columns with low cardinality after those with higher cardinality, provided that predicates are always

supplied on the leading high-cardinality columns. MDAM query technology will be used for the low-cardinality columns when they are omitted from the query predicates. However, generally it is better to 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 instead, as it will immediately restrict the query search space.

There is one subtlety associated with low-cardinality columns that is important to note. Consider a case where the primary key is composed of a very low-cardinality column (say, 1 or 2) that is always specified in the query, followed by a column with 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 effectiveness

Partitioning is a method of distributing a large table over multiple volumes (and systems) that remains 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.

Partitioning is an important consideration for achieving effective parallel query and load processing. Because it is dependent on the chosen primary 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.

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. For example, 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, then the NonStop SQL/MP optimizer is likely to read the dimension table, select the rows that match the predicate, 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, then the optimizer is likely to scan the fact table entirely and perform a hash join to the dimension table. The latter operation is much more intensive than the former. 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 key order of the fact table. It might be sufficient to place this column immediately after one with 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 with the same key and partitioning structures are very efficient, as the query might benefit from partition-by-partition key-sequence merge joins, rather than less-efficient repartition joins.

Load/insert efficiency

Although load/insert efficiency is an important consideration for performance, generally it should be secondary to the other considerations, unless the specific circumstances indicate otherwise.

The choice of primary key determines the degree of parallelism available to the load application. Consider an example where SALES_DATE is the high-order column of a fact table’s primary key. If data is always loaded in order of SALES_DATE, then 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 as the data will be partitioned over many more volumes.

Load processing usually is a batch operation that occurs regularly (for example, each day, week, or month), and is bounded by the time window available for the load. Query processing occurs at all other times. It is usually preferable to optimize query processing, which occurs frequently, rather than the less-frequent batch processing. Your circumstances may differ, however, so you will need to determine the tradeoffs that are appropriate for your application. For example, for some applications it is imperative that the batch load processing complete within a certain time period. In this case, the batch processing design should be optimized 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 physically clusters the data. This is advantageous for sequential access methods, which are often used in DSS query processing, as each I/O operation reads a large block of data from disk and transfers it to cached memory, enhancing query performance.

When a primary key involves data that has a skewed distribution, system performance can degrade. To promote a satisfactory distribution of data and provide superior query performance, you must understand the distribution of all column values that comprise the key. Specifying more precise partitioning values might be sufficient for dealing with skewed values.

Fact table partitioning techniques

Fact tables are usually quite large and therefore need to 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 partitions should be equal in size. Otherwise, scan performance is uneven and is dominated by the largest partition. If possible, place multiple partitions within the same processor on disk volumes attached to different SCSI controllers to avoid controller I/O contention.

There are three primary ways to partition tables. Each relies on the standard range partitioning capabilities of NonStop SQL/MP Software. Each method also carries both advantages and disadvantages that must be fully understood at the design stage. The specific method of partitioning affects the load architecture, query performance, and the 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. The partitioning method is closely related to the key structure, because the partitioning column(s) is the leading part of the primary key and affects the order in which the data is stored. Several questions must be addressed when considering which column(s) 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 or 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 DSS 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 usually 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 is said to have a grain of time, location, and product. Each dimension would be represented in the fact table as an individual key column, and each row of the fact table would include all three dimensions.

If your fact table does not possess these characteristics—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 for proper dimensional modeling techniques.

In standard range partitioning, the leading column(s) are used to partition the table horizontally. For example, if the chosen key order was location, time, and product, location could be used for the partitioning column. The table data would be ordered by location and be distributed across all disk volumes based on specific partitioning values—sequential groups of locations.

Partitioning for sequential access

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

Partitioning for sequential access is characterized by choosing the partitioning keys in a way that causes inserts and loads to add data sequentially to partitions, from one partition to another.

A good example is a table that uses a date for the partitioning column. In our retail fact table example, the date column would be the leading key column and also would be used as the partitioning column. New rows added to the table always would have date values greater than the existing dates in the table. The partitioning range of this table would increase continually as the dates increase.

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

All new rows added to the table would always fall logically at the end of the table, because new rows have dates greater than existing rows. This makes sequential partitioning suitable for load or load-append processing, which is faster and more efficient than randomly inserting data and avoids table fragmentation.

Because the newly added data is clustered around the partitioning value, parallelism during a load is limited or nonexistent. Because of this, a large data load may not complete within the allotted time window.

Deleting old data may be as simple as dropping the oldest partitions, or using the NonStop SQL/MP 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 and adding new partitions to a table. Neither method requires queries to delete rows from the table to purge old data.

Updating existing fact table data is usually not required, as this data is historical in nature. However, when it is necessary, update processing will be pseudo random, but can be performed in parallel streams, processing each disk partition concurrently.

Because 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 often is distributed over at most a few partitions, or perhaps contained entirely within a single partition. This is especially true if many dates are contained within one partition. If many users query across the same

date range concurrently, performance bottlenecks probably will occur on the processors and disk volumes being accessed, and hardware resources will not be effectively used. However, if the queries specify data ranges that are nonoverlapping and diverse, this partitioning technique may prove adequate.

Database management activities are reduced when using this partitioning method. Partitions must be sized for the expected data volume for a specific date range. Once the data is loaded, it normally requires no further management 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, some of the more complex data models that require many dimension columns in fact tables will use secondary indexes.

Keep in mind that load and load-append cannot be used when a secondary index is present. Either the index must be dropped and recreated after the load completes, or Insert processing must be used instead.

If data is added using the load or load append utilities, it will bypass HP NonStop Transaction Management Facility (NonStop TMF) Software processing. Tape backups must be explicitly managed, whether using NonStop TMF Software or 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.

Partitioning for random access

Partitioning for random access is characterized by choosing a partitioning column that causes inserts to add data randomly across all partitions of the table.

In the retail example, using the location or product dimensions as leading key columns are examples of partitioning for random access. Loading a batch of new data into the table requires the use of random inserts. Consequently, new rows cannot be added using load or load-append.

New rows are inserted within existing ranges, causing table fragmentation as existing blocks are split to make room for newly added rows. Online reloads are required periodically to defragment all partitions for maintaining adequate scan performance.

More effort is required to plan for this type of partitioning. You must know the current and future data distributions very well; otherwise, the partition sizes will be unbalanced and susceptible to filling up unexpectedly. You also need to obtain row counts for each range of data for the partitioning column, and a forecast of how future data will affect that distribution. Delete processing of old data is more 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 the old rows for purging, if purging data by a date range.

Depending on the number of unique values for the key columns that precede the date column, you could use MDAM processing to skip ranges of unaffected rows. Be aware that if delete processing is done through the SQLCI utility (or other query tools), a transaction will exist for the duration of the query, and may be aborted due to the NonStop TMF Software transaction timer. You might need to 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. As an alternative, programmatically managing delete processing 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 column is omitted from the list of predicates. In this case, you might use MDAM to avoid full partition scans.

Online reloads must be scheduled to run periodically to eliminate fragmentation and maintain good scan performance. All partitions are likely to need this, as data will be inserted to every partition of the table. Data partitions must be managed more 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.

Secondary indexes are supported, because the data is added using insert processing.

Database backups can be lengthy. Because loading new data 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 TMF online dumps might 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 might be more effective.

Hash partitioning

The previous partitioning that results in sequential and random access each have advantages that ideally could be combined for the same table. Partitioning for sequential access preserves table compactness and reduces database maintenance—there is no fragmentation—while partitioning for random access has the advantage of distributing rows across all partitions, reducing bottlenecks by increasing parallelism.

Hash partitioning seeks to combine the advantages of the two methods into one.

Hash partitioning is characterized by the introduction of an additional column used specifically for partitioning purposes. The partitioning column is the first key column of the table, and it is used to specify the range of values over which the table data is distributed. There are several ways to implement the hash partitioning method, but the simplest form is called *round-robin partitioning*.

Round-robin partitioning introduces a partition column as the first key column of the table. The values assigned to this column correspond exactly to each table partition, but are otherwise arbitrary. Most implementations use the range “1” through “n,” with each value being assigned to exactly one data partition. A partition dimension table is usually created that holds the range of partition values and is treated like any other dimension, except that it is transparent to the users.

When new rows are added to the table, the natural key is subordinate to the partition column. This characteristic can be exploited in certain situations. For example, if the date column is placed immediately after the partition column (and assuming all new dates are added to the logical end of the table), then the entire set of rows for a specific date can be distributed over all partitions. The advantage that this offers over sequential partitioning is that a specific date value is no longer clustered around one or two partitions, but instead is distributed across all partitions (as in partitioning for random access), while preserving table compactness (as in partitioning for sequential access). Because the partition column contains a constant value for each partition, all new dates are guaranteed to fall after existing data within each partition, eliminating fragmentation.

When queries access the data from the fact table, MDAM query technology probes for all values of the partition column, one per 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 partition column. This method incurs the cost of an additional join, but eliminates MDAM query technology processing, as the partition column values now 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 (which only occurs when the fact table is not the first outer table

of the query.) Otherwise, the ESPs collide on each partition of the fact table, creating a “wave” effect as ESP processing moves across each partition.

The partition table can bring other benefits, too. If the partition table is selected as the outer table of the join, then it determines the number of ESPs the query uses. By creating the partition table with one partition per processor, and by structuring it so that the rows within each partition align with the corresponding fact table rows, the join performed by the ESPs access corresponding partitions within the same processor. This provides efficient join performance and manages the ESPs.

New data loaded into the fact table can make use of load-append processing, and can occur in parallel. Purge processing is similar to random partitioning, but more efficient. Update processing is similar to sequential partitioning. However, MDAM query technology is used to locate the correct partition, as this value is unknown to the application. Database maintenance is simplified, as 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 because partitions do not become fragmented. Secondary index creation for round-robin partitioning is similar to that of sequential partitioning. The backup requirements are similar to those of random partitioning, because all partitions are affected by data loading.

Round-robin 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, extending the partition number range, and distributing new data over these new partitions. This method eliminates the need to rebalance the entire database.

Another way to implement hash partitioning is by selecting an appropriate dimension column and hashing the value for use as the partitioning column in the fact table. Additionally, the dimension table could use the same hashed value as its partitioning column. Joins between this dimension table and the fact table would specify both the original column value and the hash value, and would benefit from being similarly partitioned tables. This method can be useful for frequent joins between two large tables that otherwise might use repartitioned hybrid hash joins.

The column selected for the hash key needs to have a fairly even distribution so the table partitions are reasonably unequal in size. To be of value, it should also be a dimension that is used very frequently by queries.

Fact table partitioning summary

Each partitioning method described here 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 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 DSS designs makes predicting query response time very difficult.

If a full-scale environment is not available, several alternatives are possible. For example, if HP NonStop Storage Management Foundation (NonStop SMF) Software is available, you could create a prototype database with the same number of partitions as are planned for production. You could implant the SQL catalog with production statistics so that the subsequent analysis of query plans is more accurately represented. This can be done without loading any data at all. Alternatively, if NonStop SMF Software is unavailable, the Build Model Catalog facility of the Asset internal sizing tool could be used to construct a very small database that appears to the NonStop SQL/MP Database to be production scale. Then an analysis of query plans can be made.

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 response time will be, and how well the system will be utilized. It is much easier to change the physical design at this stage than after the application has been implemented.

While it may take several iterations to find an effective design, it is well worth the effort to do this 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 the process.

Dimension table partitioning techniques

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

If possible, take advantage of opportunities 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 executing the query, with diminished parallelism.

Column alignment

Although often overlooked, it is 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 will be wasted to align the second numeric column in the program's memory structure. If an even number of odd-length columns appear in the row, place them together, as 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.

Data compression

Although the two data compression techniques (DCOMPRESS 1 and 2) have limited data type support, they are particularly well suited to DSS tables due to the large number of repeating key values. In tests where compression has been used, the results show similar response time to queries not using compression, but still save disk space. The increase in processor time for decompression is offset by the improved I/O response, as more data is read in the same number of I/Os.

Catalog placement and maintenance

Always place NonStop SQL/MP Database and NonStop ODBC Server catalogs on mirror volumes. Separate catalogs are required for each system in a database distributed across multiple nodes. If the application expects a high degree of concurrent query execution, plan to keep the NonStop SQL/MP Database catalog on a disk volume separate from the NonStop ODBC Server catalog, to avoid I/O contention during query compilation. 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.

If I/O activity to NonStop SQL/MP Database or NonStop ODBC Server catalogs becomes excessive, the individual catalog tables can be moved to less-busy disk volumes, provided NonStop SMF Software is installed. (For details, see the section "Using NonStop SMF Software to distribute catalog tables" in this document.) However, you also should mirror these disk volumes.

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

Keep catalog fragmentation to a minimum by regularly performing online reloads. This should be done after the initial creation of the database, and after tables or partitions have been added and dropped.

Keep catalog statistics current for all catalog tables. Update statistics regularly after the initial creation of the database, and after tables or partitions have been added and dropped.

These measures will provide the most efficient access to the catalog tables.

Database sizing considerations

Database sizing spreadsheet

For a spreadsheet for sizing NonStop SQL/MP Database tables, useful for any database sizing effort, including DSS, refer to the "Table Sizing" document on DMF.

General rules for sizing

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

- Each processor supports about 30 to 50 GB of data (logical data, based on common customer implementations), although there are exceptions. For example, the HP Zero Latency Enterprise (ZLE) solutions support about 400 GB of storage per processor.
- The amount of 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 per processor for sort and workspace.
- Purchase the maximum amount of memory that the processors can support.
- Plan to mirror the system and audit volumes, program development volume(s), the volume(s) that holds the NonStop SQL/MP Database and NonStop ODBC Server catalogs, and at least the volume(s) that hold(s) the (empty) root partitions of all the data tables. If NonStop SMF Software is used to relocate NonStop SQL/MP Database or NonStop ODBC Server catalog tables to multiple disk volumes, those volumes also should be mirrored.

Work and swap space considerations

- Establish effective, consistent environmental settings for users of both SQLCI clients and NonStop ODBC clients, including the one produced by MicroStrategy Incorporated.
- Designate specific disk volumes for workspace requirements, specifically, sort workspace and temporary workspace for query process components, including ESPs.
- Construct the proper DEFINES that direct the SQL and SORT components to use the designated devices.
- Distribute this information to interactive users of SQLCI clients, or have the information loaded into their HP Advanced Command Language (TACL) segments at logon.
- Add this information to the ODBC configuration.
- Monitor workspace usage over time to determine if and when additional space is required. HP Event Management Service (EMS) logs (Prognosis messages), Disk Space Analysis Program (DSAP), and the HP Measure product all provide helpful information for determining space usage.

The NonStop SQL/MP Database creates temporary tables as needed, according to the following rules:

- If a query involves a join operation, the NonStop SQL/MP Database creates a temporary table on the volume on which the outermost table in the join resides. If the table is partitioned, the NonStop SQL/MP Database creates a temporary table on the volume on which the specified partition resides.
- If a query involves a hash join or a hash grouping, the NonStop SQL/MP Database creates temporary files on the program swap volume.
- If a query requires a sort operation but does not use the FastSort technique, the NonStop SQL/MP Database creates any temporary tables used for the sort on the volume on which the table being sorted resides. If the table is partitioned, the NonStop SQL/MP Database creates any temporary table on the same partition as the partition being sorted.
- If the NonStop SQL/MP Database uses the FastSort technique to sort a table, the FastSort SORTPROG process creates a temporary file on the SORTPROG scratch volume.

Specify the volumes to use for sorting with the following DEFINE:

```
ADD DEFINE =_SORT_DEFAULTS,

CLASS SORT,

SCRATCHON ($SORT01,$SORT02,$SORT03,...) ,

SWAP $SWAP00;
```

When using parallel queries, disable parallel FastSort with the following DEFINE:

```
ADD DEFINE =_SQL_CMP_NO_MULTISORT,

CLASS MAP,

FILE X;
```

Use the =_SQL_TM_node_volume DEFINE to create temporary tables that would normally go to the specified volume on another specified volume instead, as follows:

```
ADD DEFINE =_SQL_TM_NODE1_DATA01,

CLASS MAP,

FILE \NODE1.$TEMP01.X;
```

MDAM query technology design considerations

To exploit the capabilities of MDAM query technology¹, you first must know the data distributions and unique values for each key column in the table. Generally, the key columns are ordered 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 in sequence of their unique value count, from least to greatest. For example, assume that the columns and unique entry (values) counts of the table are shown in the following table.

¹ For a detailed description of the functionality of the NonStop SQL/MP MDAM facility, refer to the document "Efficient Search of Multidimensional B-Trees."

Key columns and value counts in MDAM query technology

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, but not the store key, were supplied, MDAM query technology likely would probe through the store-key values to access the qualifying data. However, if the customer key was not supplied, a full table scan would more likely be used.

Data-type considerations

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

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 presently long running, especially when they 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 size 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 a lot of processing cycles.

Third-party productivity tools exist that can be quite useful for creating and maintaining summary tables. They 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.

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. Indexes are usually only useful when relatively few rows match the predicate. Otherwise, it is more efficient to perform a table scan. Fact tables are often loaded using a bulk-load method, such as load or load-append, which only works when no secondary indexes exist.

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 exists on the same table); and they are transparent to queries and can be dropped quickly and easily.

The following characteristics generally indicate that a secondary index is useful:

- Large tables
- Rows are added using the NonStop SQL/MP Database Insert statement, not load or load-append utility
- Key column access, other than the leading primary key column(s), or nonindexed data column access
- 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 providing 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 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.

Using NonStop SMF Software to distribute catalog tables

NonStop SQL/MP Software catalogs consist of several tables that reside together on a single volume and subvolume. In large DSS environments, the volume on which these tables reside can experience queuing delays when many queries are dynamically compiled at the same time, decreasing query response time. NonStop SQL/MP Software does not permit these tables to be individually moved or partitioned across disk volumes.

However, by placing the NonStop SQL/MP Database or NonStop ODBC Server catalog on a logical volume, NonStop SMF Software permits individual files and tables—including NonStop SQL/MP Software catalog tables—to be relocated to other physical volumes, using the FUP RELOCATE command. Even though the physical table resides elsewhere, the NonStop SQL/MP Database and NonStop ODBC Server view it as still being part of the original disk volume, now the logical volume. Use the Measure product to determine which tables are most active, as these are good candidates for relocation. You only need to relocate those NonStop SQL/MP Database catalog tables whose relocation would provide the greatest performance improvement. Consider placing relocated catalog components on volumes whose primary processor is different from that of the original catalog. Use the Measure product to determine disk and processor candidates for relocated components.

Always place NonStop SQL/MP Database and NonStop ODBC Server catalogs on mirrored volumes. Additionally, make sure the physical volume to which any catalog components are moved is mirrored. If any catalog components are relocated to an unmirrored volume that fails, the entire catalog will be unusable. Always seek guidance from HP field engineers before relocating catalog components.

Design techniques to avoid

Joining tables on calculated columns

Joins are usually made between 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 requires a full scan of the table, and either a sort or repartition hash (hash on the calculated column

value, write the results to temporary workspace, and then join to the other table) on the calculated column value, even if a small row-set results from the join. This is a very expensive operation and should be avoided, if at all possible.

“Over-normalizing” tables

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

In some cases, it is acceptable to 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 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.

Forcing MDAM query technology

MDAM query technology can provide substantial performance benefits when used properly. However, it is not a panacea for all performance issues. Resist the temptation to simply force MDAM query technology on all queries. The optimizer normally chooses the correct optimizations. Forcing on MDAM query technology when the optimizer chooses otherwise has been shown to degrade query performance in several cases. There may be times when forcing on MDAM query technology improves queries. Before doing so, have a thorough understanding of the data relationships and queries, and always perform extensive testing.

Load-only optimized design

Data warehouse projects often are initiated with tight time schedules. Implementation begins, even though query requirements are barely understood by the designers. This might be acceptable for proof-of-concept efforts or 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, suboptimal design decisions are made, and either the users have to accept slower query response times or the database will need to be rebuilt or significantly modified.

In the absence of query requirements, designers often optimize for the requirements they understand, usually database loading. While this is not necessarily bad, it often results in a design that performs very well for database loading, but poorly for query processing.

Whenever possible, avoid this approach to design. 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 investigating alternative database designs. Take time to create a large or full-scale test database (consider using Build Model Catalog if a large-system environment is unavailable), partitioned and organized as it would be in production. Construct typical queries and analyze their Explain plan 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 of this document.

Data loading techniques and considerations

For detailed discussion of this issue, refer to the “Solution Factory Data Specification” document.

Block slack space

Scan performance for DSS queries is largely dependent upon data compaction—that is, how much data a single I/O request will return, which, 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 scan performance will be.

One way to verify that data blocks are as compact as possible is to set the table’s Slack parameter to 0 when loading the table. NonStop SQL/MP 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 will never be used. Furthermore, each I/O will read less data. Setting the Slack parameter to 0 maintains that all available space will be used for data. (Note that this parameter is specified in the Load utility.)

Database management activities

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 it 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 greatest number of rows. This provides the highest data scan rate possible, as 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 are bulk loaded (using either the Load or Append utility with the Slack parameter set to 0) do not require reorganizations. However, tables that experience random or pseudo-random inserts, random deletes, and updates that change row lengths need to be reorganized periodically to provide maximum scan performance. (Tables used in a DSS application are often loaded or reloaded with zero slack space to provide maximum scan performance.)

The FUP INFO utility with the “Stat” option or the Reload Analysis Program (RAP) are both effective for determining which partitions of a table to reorganize. Only those partitions that show fragmentation should be reorganized.

Table reorganization in NonStop SQL/MP Software 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 maintain that the fragmentation does not contribute to poor performance.

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

Updating statistics

Various statistics about table data content and size are stored within a NonStop SQL/MP Database catalog. This information is used by the database's 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 NonStop SQL/MP Database optimizer cannot produce an effective access plan unless the statistics adequately represent the data within the tables. When no statistics are present, the NonStop SQL/MP Database optimizer assumes default values, but these might not produce plans that provide adequate performance.

If statistics are not updated after the creation of a secondary index, NonStop SQL/MP Software estimates the index statistics based on the statistics for the base table. However, it is best to update statistics on the table after creating an index, as the file table maintains two column values for the index, the End of File pointer (EOF) and NONEMPTYBLOCKCOUNT.

Statistics should also be generated for all NonStop SQL/MP Database and NonStop ODBC Server catalog tables (and indexes), as this improves NonStop SQL/MP Database query compile time and NonStop ODBC Server performance.

For large tables, the probabilistic option should be used on the Update Statistics command, as follows:

```
UPDATE ALL STATISTICS FOR TABLE <tablename> NO RECOMPILE
PROBABILISTIC;
```

The Update Statistics utility launches two statsrv processes per processor, and each reads and processes partitions of the table until complete. As this is a very processor-intensive operation, the processors become very busy. To limit the impact on the system, run the Update Statistics utility using a low priority. The processors are still busy, but the system manages the mixed workloads.

By default, the Update Statistics utility gathers statistics for all key columns, including indexed columns. Use the All option to gather statistics for all columns.

Issues involving very large databases (VLDBs)

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 systems interconnected with the database distributed across all systems. When possible, execute programs on the system where the data resides.

When designing a database distributed across multiple systems, you have several alternatives for locating tables. You have to 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. This 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. This is feasible if the division of fact tables coincides

with the division of their use; users on a specific system only access data on that system. This approach provides good performance, but by replicating large amounts of data.

Maximum table size

Number of table partitions

Partitions for tables created with the Format-1 attribute are limited to 2 GB, while tables using the Format-2 attribute can have partitions as large as the underlying disk volume.

The main factor dominating the maximum table size is the number of partitions and is determined by the formulas in the next section.

Formula for calculating the maximum number of partitions

The formula for determining the number of partitions is shown below. This formula provides a very close approximation, but it is not exact.

The limit on the number of table partitions is approximated by:

- Standard (8000) / (24 + key size)
- Extended (31000) / (24 + key size)
- Format-2 enabled (31000) / (28 + key size)

The limit on the number of index partitions is approximated by:

- Standard (8000) / (26 + primary key size + index key size)
- Extended (31000) / (26 + primary key size + index key size)
- Format-2 enabled (31000) / (30 + primary key size + index key size)

Practical limitations

Although you may be able to create a table with more than 650 partitions, the NonStop SQL/MP Software ESP manager can only manage about 580 ESPs.

Techniques for creating large tables

Using a single Create Table statement

Large tables are large because they contain many partitions. Writing a Create Table DDL statement for a large table is tedious due to the large number of partitions involved. Instead, you should use a TACL script or other means to generate the DDL statement.

When you generate the Create Table statement, use define names for all partition names and catalog names. By thinking small, you might be able to get the entire statement to fit within the NonStop SQL/MP Database's 32 KB buffer limit. Use the fewest characters possible for the DEFINE names.

In the example below, first a small "skeleton" table is created that has the basic table structure. Then a large table is created across multiple systems, using the skeleton table as the basis for its structure.

```
== Catalog defines  
  
ADD DEFINE =C1,CLASS CATALOG,SUBVOL \SYS1.$D11011.CAT  
  
ADD DEFINE =C2,CLASS CATALOG,SUBVOL \SYS2.$D11011.CAT  
  
. . .  
  
ADD DEFINE =C8,CLASS CATALOG,SUBVOL \SYS8.$D11011.CAT
```

== Define for skeleton table

ADD DEFINE =T,CLASS MAP,FILE \SYS1.\$D03011.DB.ODS

== Defines for 496 partition table

ADD DEFINE =P000, CLASS MAP, FILE \SYS1.\$D03011.DB.ODS000

ADD DEFINE =P001, CLASS MAP, FILE \SYS1.\$D00003.DB.ODS000

. . . .

ADD DEFINE =P495, CLASS MAP, FILE \SYS8.\$D15010.DB.ODS000

== Create the skeleton table

```
CREATE TABLE =T (  
    AREACODE    SMALLINT    DEFAULT SYSTEM NOT  
NULL,  
    PHONE_NO    INTEGER     DEFAULT SYSTEM NOT  
NULL,  
    LOAD_TIME   DATETIME    YEAR TO SECOND  
    DEFAULT SYSTEM NOT NULL,  
    CALLED_NO   LARGEINT    DEFAULT SYSTEM NOT  
NULL,  
    PLAN        CHAR(1)     DEFAULT SYSTEM NOT  
NULL,  
    PRIMARY KEY (  
        AREACODE  
        , PHONE_NO  
        , LOAD_TIME  
        , CALLED_NO  
    )  
)  
CATALOG =C1
```

ORGANIZATION KEY SEQUENCED
AUDIT
BLOCKSIZE 2048
BUFFERED
AUDITCOMPRESS
EXTENT (1 PAGES, 1 PAGES)
MAXEXTENTS 16
PARTITION ARRAY EXTENDED
SIMILARITY CHECK ENABLE
SECURE "NNNN";

== Create the skeleton table

```
CREATE TABLE =P000    LIKE =T
      PARTITION (
                =P001 EXTENT(65535,50000) FIRST KEY 100
CATALOG =C1
                ,=P002 EXTENT(65535,50000) FIRST KEY 150  CATALOG =C1
                . . .
                ,=P495 EXTENT(65535,50000) FIRST KEY 900
CATALOG =C8
      )
      AUDIT BUFFERED AUDITCOMPRESS
      BLOCKSIZE 4096
      CATALOG =C1
      EXTENT(65535,50000)
      MAXEXTENTS 16
      PARTITION ARRAY EXTENDED
      SECURE "NNNN";
```

Increasing heap memory space

The default heap size for SQLCAT is 4 MB and it can expand up to 16 MB. You will likely need to increase the heap space used by SQLCAT when creating a table with many partitions. Use this DEFINE:

```
=_SQL_CAT_HEAP_LIMIT,  
  
CLASS MAP,  
  
FILE MXXXX
```

where “x” can be as high as 2,047 (in MB). Memory consumption can be monitored with the NSKCOM command. You need to use this DEFINE if you receive an error 1,120 during the Create Table statement.

Increasing process file segment size

For processes that must open tables with many partitions, the process file segment (PFS) size will need to be increased. The PFS size can be increased with the BIND program (Change PFS command), up to 128 MB per process. Increase the PFS for SQLCAT, SQLCOMP, AUDSERV, and SQLUTIL. Application programs that need to open every part of the fact table also might need to increase the PFS size.

Compiling queries for large tables

Special NonStop SQL/MP defines

For tables with many partitions (for example, 600 or more), it may be undesirable to incur the cost of executing repartition plans. Therefore, disable the optimizer from considering repartitioned type joins (“Type 3” joins). A repartitioning plan could result in having an ESP per fact table partition (even when the fact is not the outer table). You can disable this using the following DEFINE

```
=_SQL_CMP_NO_REPARTITION,  
  
CLASS MAP,  
  
FILE X
```

Using multiple NonStop SQL/MP Database catalogs

DSS databases often involve the use of many table partitions. To avoid filling certain catalog tables, you should consider creating the DSS database with multiple catalogs. Partitions of the same table can be cataloged in different SQL catalogs.

Maximum catalog size

NonStop SQL/MP Database catalogs cannot be extended beyond their default settings. In a very large DSS environment certain NonStop SQL/MP Database catalog tables might become full, preventing tables or partitions from being created. If this happens, you must create the new objects in one or more additional NonStop SQL/MP Database catalogs. Multiple partitions of the same table can be defined in different NonStop SQL/MP Database catalogs.

The most likely catalog components to fill are the PARTNS table and its index, INPART01. The number of PARTNS rows is equal to the square of the number of partitions (P^2). The maximum size of the PARTNS table is approximately 234 MB, and half that value for its index. The precise number of partitions that the table and its index can hold varies, depending on the partition value length, but typically is about 1,400 partitions (of all partitioned tables). For example, in a large DSS environment

that might have several tables with 300 partitions each, it is not uncommon to encounter this limitation.

If a table- or index-full condition is encountered during table creation, create the table in a new NonStop SQL/MP Database catalog. For a large DSS environment, assume that you will need multiple NonStop SQL/MP Database catalogs and plan accordingly.

Improving performance

This section discusses some of the common performance problems associated with large DSS databases. Additional information exists in other manuals and is not repeated here. Specifically, the *NonStop Query Design Guide* contains several chapters dedicated to analyzing and improving query and database performance, and the *NonStop 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 faced with the challenge of tuning a DSS system or database, several tradeoffs often need to be considered. For example, it is impossible to optimize all queries or queries and batch-loading workloads. One of the first steps involved in tuning a DSS system 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 create secondary indexes.

Managing numerous ESP processes

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

Following are a few suggestions for handling some of the problems created by many ESP processes.

Reduce table partitions

While a DSS database almost always uses partitioned tables, try to reduce the number of table partitions and keep the partitions a multiple of the number of processors, with one or two partitions per processor when possible. Using format-2 table allows each partition to be as large as the underlying disk drive.

Reducing table partitions also will reduce the number of active ESPs used for queries, and help to eliminate resource contention. Note that if overly busy processors are a pervasive problem, then it is likely you need additional hardware resources. Attempting to tune an undersized system will provide little gain.

Increase the ESP startup timeout

In some heavily loaded systems, ESPs timeout before receiving their startup message. In this case, you can use the `DEFINE =_SQL_EXE_ESP_CRE_TIMER` to increase the timeout value.

While this may prevent ESP timeouts, their existence indicates an overly busy system. Review the system sizing, database design, and query design to determine whether the execution workload can be decreased or whether additional resources are needed.

Use early cross-products

The number of ESPs used for a query is based upon the outer table selected in the execution plan. ESP overload is usually only a problem when the fact table (or another very large table) is chosen as the outer table. When smaller tables are selected first, ESP overload is less likely due to the fewer partitions.

A new DEFINE =_SQL_CMP_DO_EARLY_XPROD has been created that causes the optimizer to consider early cross-product joins.

This DEFINE may prevent the fact table from being the first table accessed in the query. The number of ESPs is determined by the outermost table, which is likely to be a smaller dimension table. It is usually good practice to partition the dimension tables to no more than one per processor.

Restrict the maximum number of ESPs

If the fact table is the only table with an excessive number of partitions, it might help to restrict the maximum number of ESPs to a value that is less than the number of fact table partitions (but larger than the dimension table partitions).

Using the DEFINE =_SQL_CMP_MAX_NUM_ESPS prevents the fact table from being selected as the outer table in a parallel plan, controlling the number of ESPs used in the query. Partitioned dimension tables still can be used as outer tables for parallel plans. In some cases, the optimizer might also select the fact table as the outer table for a serial plan, which is usually undesirable. This technique is best used for individual queries, as it may have undesired consequences when applied overall.

Use a partitioning table

Some DSS applications have successfully used a small “partitioning” table to control the number of ESPs. This technique is often used when the fact table uses round-robin partitioning, and uses a view to join the fact table to the partitioning table. See the “Fact table partitioning techniques” and “Dimension table partitioning techniques” sections of this document for more details.

Addressing skewed data distributions

Tables sometimes contain columns with significantly skewed values. For example, assume a table has a date column with a range of values exceeding 100 years, but nearly all rows having dates within the last several years, the time period most often queried.

The optimizer assumes an even distribution, so a range predicate of the form—DATE between “1999-01-01” and “1999-12-31”—can cause the optimizer to select a small percentage of rows, when, in fact, it may be much higher. The optimizer may expect only a few thousand rows to be selected from the table, when the query may actually return several million.

The problem can be corrected easily by explicitly settling the range of values in the catalog to more accurately represent the volume of data that would be selected.

Addressing large table issues

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

- Access through non-key columns
- Access through a subset of the primary key and MDAM query technology is not used
- 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 NonStop SQL/MP Database runtime statistics (that is, the Display Statistics command) and Explain Plan output to determine whether secondary indexes would benefit query performance. Consider creating a secondary index when the Explain Plan output indicates that

a column predicate having a low selectivity will result in a table scan and that 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(s). 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 is often one of the most significant issues faced by customers in the DSS environment. When possible, use similar partitioning and primary-key structures, which enables efficient partition-to-partition joins.

DSS applications usually have at least one large dimension table that must be joined to the fact table. Confirm that the dimension table is indexed as needed to avoid full-table scans. Verify that the statistics are updated and correct. Deal with skewed data distributions as described in the “Addressing skewed data distributions” section of this document. Note whether the join column(s) between the dimension table and the fact table represent an efficient path. Review Explain Plan output to verify that an efficient join method exists to the fact table. If the Explain Plan indicates that the type of join is a repartitioned hybrid hash join, then 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 a sort-merge join, but it is still a costly operation, especially when executed concurrently by multiple queries.

Consider using the DEFINE, `=_SQL_CMP_DO_EARLY_XPROD`, to influence the optimizer to perform an early cross-product with other dimensions. This may be sufficient to obtain the primary keys of the fact table so that the join is more efficient. Also consider using the DEFINE, `=_SQL_CMP_OPTMZ_LEVEL` (with the FILE X2 optimizer level). The optimizer will consider additional access plans.

If none of these methods improve the access plan, you will likely need to restructure one or both tables, create an index on the fact table, or create an aggregate table (if reasonable) to improve performance.

Another common performance problem involving two large tables is one where a join is made between a large dimension table and a fact table, but “through” a higher-level dimension table.

Consider the following 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 to sum a fact column but report the results by the demographics description. In this case, a large join is required between the customer and fact tables, 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 the preceding case.

Look for opportunities that limit the joins between large tables.

Addressing NonStop SQL/MP catalog performance

Always place NonStop SQL/MP Database catalogs and NonStop ODBC Server catalogs on mirrored volumes. Separate catalogs are required for each system, in the case of a database distributed across multiple nodes.

Customers that expect a high degree of concurrent query execution should plan to keep the NonStop SQL/MP Database catalog on a disk volume separate from the NonStop ODBC Server catalog, to

avoid I/O contention during query compilation. 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.

If I/O activity to NonStop SQL/MP Database or NonStop ODBC Server catalogs becomes excessive, the individual catalog tables can be moved to less busy disk volumes, provided NonStop SMF Software is installed. (See the “Using NonStop SMF Software to distribute catalog tables” section of this document for details.) However, these disk volumes also should be mirrored.

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

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

Keep catalog statistics current for all catalog tables. Update statistics regularly, after the initial creation of the database, and after tables or partitions have been added and dropped.

These measures will provide the most efficient access to the catalog tables.

NonStop SQL/MP defines for special circumstances

The *NonStop SQL Reference Manual* contains more information about some of the DEFINES in this section.

Also be aware that occasionally new DEFINES are documented only in the NonStop SQL/MP Database softdocs. Refer to these documents for the most current information and for details of the parameter values.

It is not possible to explain how and when each DEFINE should be used, but the information here is intended to identify some of the more commonly used DEFINES, and how they are used. DEFINES are not a panacea for all performance or query issues, so use them with care, and fully understand their consequences.

- =_SQL_CAT_HEAP_LIMIT

When creating a large table, the SQLCAT process might need a larger heap size, which can be set using this DEFINE. Otherwise, the Create statement will fail. The valid range is from 8 to 2,047, and is specified as FILE Mnnnn.

- =_SQL_TM_SYSTEM_VOLUME

This is designed to give users control over placement and use of temporary files and to minimize the possibility of encountering a 122 error. Use this DEFINE to move temporary files from data volumes to designated work volumes.

- =_SQL_CMP_EXPLAIN_CAT

This causes the Explain Plan to display the table and column statistics for the tables used in the query, which is very useful when analyzing query performance.

- =_SQL_CMP_EXPMDAM_ACCESS

This causes the Explain Plan to display whether a sparse or adaptive dense algorithm will be used by MDAM query technology for specific columns, which is useful when analyzing query performance.

- =_SQL_CMP_MAX_NUM_ESPS

This is used for large tables. Without this DEFINE, NonStop SQL/MP Software will never choose a parallel query on a table with more than 256 partitions.

Overrides the default limit for the maximum number of ESPs a parallel plan can have. Any parallel plans beyond this limit will not be considered.

The filename should be of the form *Mnnn*, where the numeric portion specifies the maximum number of ESPs. The default limit is 256.

In certain circumstances, this DEFINE can be used to ensure that the NonStop SQL/MP Software never chooses a large, partitioned table as the outer table (first table) of a query plan, by setting the DEFINE to a value that is less than the number of partitions on the large table.

- `=_SQL_CMP_NUMRECS_SCALE`

This is used when the optimizer inaccurately estimates the number of rows. Lets the user specify a percentage that is used to scale the optimizer number-of-records estimate. The format for the file name is *Xnnnnnnn*, where the first character, *X*, can be any alpha character. Valid values for the numeric portion are 0 (signifying 0 percent) to .9999999 (99.99999 percent). The optimizer limits the resulting number of rows to one on the low end and the largest real number on the high end. This DEFINE is only used for correcting estimates of the number of records for temporary file creation and for sorting. It is not used for costing.

- `=_SQL_CMP_DO_EARLY_XPROD`

This allows the optimizer to consider early cross-product plans when selecting an access plan. This can be very effective for plans that have many small dimension tables joining to a large fact table. Instead of scanning one dimension table, then scanning the fact table, then joining the other dimensions, cross-product does a Cartesian product on the dimension tables, then joins the fact table. This has been very effective for some types of very large queries.

This DEFINE forces the optimizer to consider a cross-product between a composite and an inner table if no join predicate is found that relates them. The use of this DEFINE enlarges the search space considerably and may increase compile times significantly, especially for queries with a large number of tables. However, using this DEFINE might result in better plans.

- `=_SQL_CMP_OPTMZ_LEVEL`

This allows the optimizer to consider more plans for hash joins and hybrid hash operations before making a selection. Useful for complicated queries. This has been very effective for some types of very large queries, although it usually increases compile time—not normally a problem for long-running queries.

With this DEFINE, the user can specify different levels of extended search space so that more plans are considered and evaluated, thus increasing the probability that the optimizer selects the best plan.

The DEFINE can be involved in two different forms for different levels of optimization extension.

- The FILE X1 instructs the optimizer to extend the search space to include as many hash join plans as possible.
- The FILE X2 instructs the optimizer to extend the search space to include as many hash join and hybrid hash plans as possible.

Note that the parameter to the keyword FILE decides the level of extension. If this DEFINE is not found during the compilation, or the following form of this DEFINE is added, the default search space is used.

If the extended optimization level is greater than the default (0), an indication can be found in the Explain plan as follows:

- Query Plan 1 (with extended optimization level 1)
- SQL request: Select

With expanded search space, NonStop SQL/MP Software's compiler is expected to spend more time carrying out the optimization. As an example, a set of tests were done with the query that included a nine-way join and a large number of equality join predicates. The test showed an increase of approximately 8 percent for Level X1, and 30 percent for Level X2. For most queries, the penalty should be less than the numbers shown here. The penalty increases as the number of equality join predicates increases.

- =_SQL_EXE_ESP_CRE_TIMER

SQL ESP used a default time limit for receiving a startup message from its creator. In certain heavily loaded systems, the timer might expire before the startup message is received, and the process fails. A new DEFINE is introduced to allow user to specify the time limit for the SQL ESP process creation. The time limit is embedded in the simple file name (the last part of the file name). The file name starts with an alpha character and the remaining seven characters specify the amount of the time limit in seconds. The default time limit is 60 seconds. The timer could range from 60 seconds to 3,600 seconds. FILE T0120 sets the time limit to 120 seconds.

- =_SQL_EXE_SRV_STEAL

Previously, the NonStop SQL/MP Software ESPs that executed a parallel query were not released until the statement that the query was prepared under was prepared again. For processes such as those of NonStop ODBC Server Software, which maintain multiple prepared statements, this meant that a large number of ESP processes could be associated with each server process. For this reason, NonStop ODBC Server Software does not keep more than one parallel query statement active at any time (that is, queries that generate parallel plans are not cached by NonStop ODBC Server Software).

Now the NonStop SQL/MP Software Executor attempts to reuse ESPs when the number of active queries exceeds a user-defined threshold. In this context, an active query is one that has been prepared and executed, and whose cursor is no longer open. For example, if the threshold is set to five, and five parallel queries are currently active, when an attempt is made to execute a new parallel query, the SQL Executor attempts to reuse the ESP processes from the oldest active parallel query.

With NonStop ODBC Server Software, this means that it is now possible to cache parallel queries and control the number of ESP processes. This threshold applies to each process, and is specified via a DEFINE of the following form, FILE Xn, where X is any alphabetic character, and n is the maximum number of concurrent ODBC queries beyond which the executor attempts to reuse ESPs. For example, the filename S6 would specify that the server reuse threshold should be set to six queries.

If this DEFINE is absent, or if zero is specified as the threshold, ESP reuse is turned off.

- SET DEFINE CLAS SORT, [NO]SCRATCHON (scratch-vol,...)

This define allows you to specify multiple scratch volumes for sort operations. The parenthesized list of scratch volumes could have up to 32 scratch volume specifications and include "?" and "*" as wildcard characters.

Query performance analysis methodology

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

- Select a representative set of key queries to use for performance evaluation as changes are made.
- Obtain current Explain Plans for each query, including the DEFINE,

```
=_SQL_CMP_EXPLAIN_CAT
```

- For NonStop ODBC/MP Server clients, when predicates for date columns are present in the query, replace them with parameters (“?p”) to mimic the actions of the NonStop ODBC Server subsystem. This can affect the resulting plan.
- Obtain runtime statistics for each query, using the SQLCI SET STATISTICS ON option.
- Identify all predicate columns and join columns for each query. These will become the “entry points” into each data table.
- Verify that all tables are loaded with zero slack space, or are regularly reorganized online to eliminate fragmentation and provide optimal scan performance.
- Update statistics for all tables after they have been loaded, and after adding secondary indexes.
- Obtain data value distributions for critical key columns—those used for partitioning, MDAM query technology, query predicates, and joins—to determine the existence and pervasiveness of skewed values.
- Work with HP support representatives to establish queries that change catalog statistics, where needed. Re-run these queries after using the Update Statistics utility.
- Regenerate Explain Plans for the query suite and determine whether any of the plans have changed.
- For queries with new plans, re-run the queries and obtain runtime statistics. Determine whether the performance improvement efforts have been effective.

For queries that have not improved from the above steps, analyze the data access points (predicate and join columns) for each table, and consider the following:

- 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 volume of rows.
- Always regenerate explain plans and re-run the queries to determine the effectiveness of each change.

The main objective of these analysis steps is to provide the most efficient access path to the requested data. Make sure the catalog statistics account for data skew, because skewed data distributions can cause the NonStop SQL/MP Database optimizer to select inefficient access plans.

Following that, 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 this is normally the last course of action, it can produce the most profound query improvements.

Implementing NonStop ODBC/MP architecture

For a detailed discussion of various considerations for managing the ODBC environment, refer to the “NonStop SQL/MP Database and NonStop ODBC Server tips and hints” document.

For more information

www.hp.com/go/nonstop

© 2005 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

06/2005

