Microsoft TechNet

Strategies for Partitioning Relational Data Warehouses in Microsoft SQL Server

Published: February 17, 2005

Authors: Gandhi Swaminathan

Contributors: Hao Chen, Rakesh Gujjula, Gang He

Reviewers: Wey Guy, Stuart Ozer, Arun Marathe, John Miller, Mike Ruthruff, Brian Goldstein

Published: January 2005

Summary: Partitioning, in Microsoft SQL Server, facilitates the effective management of highly available relational data warehouses. This white paper discusses several factors that influence partitioning strategies and design, implementation, and management considerations in this environment.

It is recommended that the readers of this white paper have read and understood the following articles:

- **Using Partitions in a Microsoft SQL Server 2000 Data Warehouse** http://msdn.microsoft.com/library/default.asp?URL=/library/techart/PartitionsInDW.htm.
- **SQL Server 2000 Incremental Bulk Load Case Study** http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/incbulkload.mspx
- **SQL Server 2005 Partitioned Tables and Indexes** by Kimberly L. Tripp http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql2k5partition.asp

The focus of this white paper is relational data warehouses and table partitioning. The target audiences for this white paper are:

- Developers and database administrators who have implemented partitioning by using partitioned views in Microsoft SQL Server. This audience will benefit from the sections on the advantages of partitioning in SQL Server 2005 and on sliding window implementation and strategies.
- Developers and database administrators who are planning to implement partitioning in the future will greatly benefit by reading this white paper in its entirety.
- Database and system administrators will benefit from the sections on Storage Area Network management and strategies to optimize I/O use.

On This Page

Partitioning a Relational Data Warehouse
Factors Influencing Partitioning for Relational Data Warehouses
Sliding Window Implementation
Strategies for Partitioning Relational Data Warehouses
Conclusion
For More Information
Appendix A: Performance Numbers
Appendix B: Platform Listing
Appendix C: Server Architecture
Appendix D: EMC CLARiiON Storage
Appendix E: Storage Isolation
Appendix F: Scripts

Partitioning a Relational Data Warehouse

The following sections will briefly explain the concept of a relational data warehouse, the benefits of partitioning relational data warehouses, and the advantages of migrating to partitioning in Microsoft SQL Server 2005.

About Relational Data Warehouses

Relational data warehouses provide a comprehensive source of data and an infrastructure for building Business Intelligence (BI) solutions. In addition, relational data warehouses are used by reporting applications and complex ad-hoc SQL queries.

A typical relational data warehouse consists of dimension tables and fact tables. Dimension tables are usually smaller in size than fact tables and they provide details about the attributes that explain the facts. Examples of dimensions are item, store, and time. Fact tables represent business recordings such as item sales information for all the stores. The fact tables are updated periodically with recently collected data.

A successful implementation of a relational data warehousing solution involves careful and long term planning. Some of the factors to consider while building a relational data warehouse are:

- Data volume
- Data loading window
- Index maintenance window
- Workload characteristics
- Data aging strategy
- Archive and backup strategy
- Hardware characteristics

A detailed discussion of these factors can be found in later sections of this document.

A relational data warehouse can be implemented using a partitioned approach or a monolithic (huge) fact table approach. The design choice of partitioned or nonpartitioned approach largely depends on the factors listed earlier in this paper. Relational data warehouses can benefit from partitioning the data. The following section highlights the benefits of partitioning relational data warehouses.

Benefits of Partitioning

As the databases in an organization scale up and contain large volumes of data, it is critical that high availability be maintained while accommodating the need for a small database maintenance window. This requirement makes partitioning a natural fit for very large databases. Partitioning addresses key issues in supporting very large tables by letting you decompose them into smaller partitions thereby facilitating better management of influencing factors such as data loading, aging, and archival. Microsoft SQL Server supports data partitioning through partitioned views in SQL Server 7.0/2000 and added support for partitioned tables in SQL Server 2005.

Partitioning in SQL Server 7.0/2000

SQL Server 7.0 introduced support for partitioning through partitioned views. In SQL Server 2000, the feature was enhanced to support updatable partitioned views. A partitioned view is best suited for a relational data warehouse when the fact table can be naturally partitioned or divided into separate tables by ranges of data. The underlying tables of the partitioned view are UNIONed to present a unified data set. Partitioned views greatly reduce application complexity because the physical implementation is abstracted from the application data access methods.

In SQL Server 2000, partitioned views can be extended to include distributed partitioned views, enabling database federation across multiple servers/instances. A discussion of distributed partitioned views is outside the scope of this paper. For a detailed discussion, see "Distributed Partitioned Views" on the Microsoft Developer Network (MSDN) at http://www.microsoft.com/sql/prodinfo/features/default.mspx.

Partitioning in SQL Server 2005

Table and index partitioning in SQL Server 2005 mitigate the complexities that are involved in managing very large databases through partitioned views. SQL Server 2005 offers horizontal range partitioning with the data row as the smallest unit of partitioning. Objects that may be partitioned are:

- Base tables
- Indexes (clustered and nonclustered)
- Indexed views

Range partitions are table partitions that are defined by customizable ranges of data. The user defines the partition function with boundary values, a partition scheme with filegroup mappings, and tables mapped to the partition scheme. A partition function determines which partition a particular row of a table or index belongs to. Each partition defined by a partition function is mapped to a storage location (filegroup) through a partition scheme. For a complete discussion on implementing partitioning with SQL Server 2005, see "SQL Server 2005 Partitioned Tables and Indexes" on MSDN.

The following section explains the advantages of partitioning in SQL Server 2005 and provides a strategy for migrating to SQL Server 2005 partitioned tables.

Advantages of Partitioning in SQL Server 2005

Table and index partitioning in SQL Server 2005 facilitates better management of very large databases through the piecemeal management of partitions. This section covers some of the advantages of partitioned tables over partitioned views with respect to relational data warehouses.

Management

One drawback of using partitioned views is that when you do, database operations have to be performed on individual objects as opposed to on the view itself. For example, if the existing index has to be dropped and a new index created, the operation has to be performed on each underlying table.

In SQL Server 2005, database operations such as index maintenance are performed on the partitioned table itself as opposed to the underlying partitions, saving a significant amount of effort managing the indexes.

Better Parallelism Mechanism

In SQL Server 2000, operations are performed on individual tables and the data is aggregated at a partitioned view level. The rows from underlying tables are gathered using the concatenation operator to represent the view. Aggregations are then performed on the resultant data.

In SQL Server 2005, queries against partitioned tables can use a new operator called *demand parallelism*. Demand parallelism is influenced by the system resources and MAXDOP settings.

Queries that use partitioned tables will compile much faster than equivalent queries that use partitioned views. Query compilation time is proportional to the number of partitions when using partitioned views, while compilation is not affected by partition count when using partitioned tables.

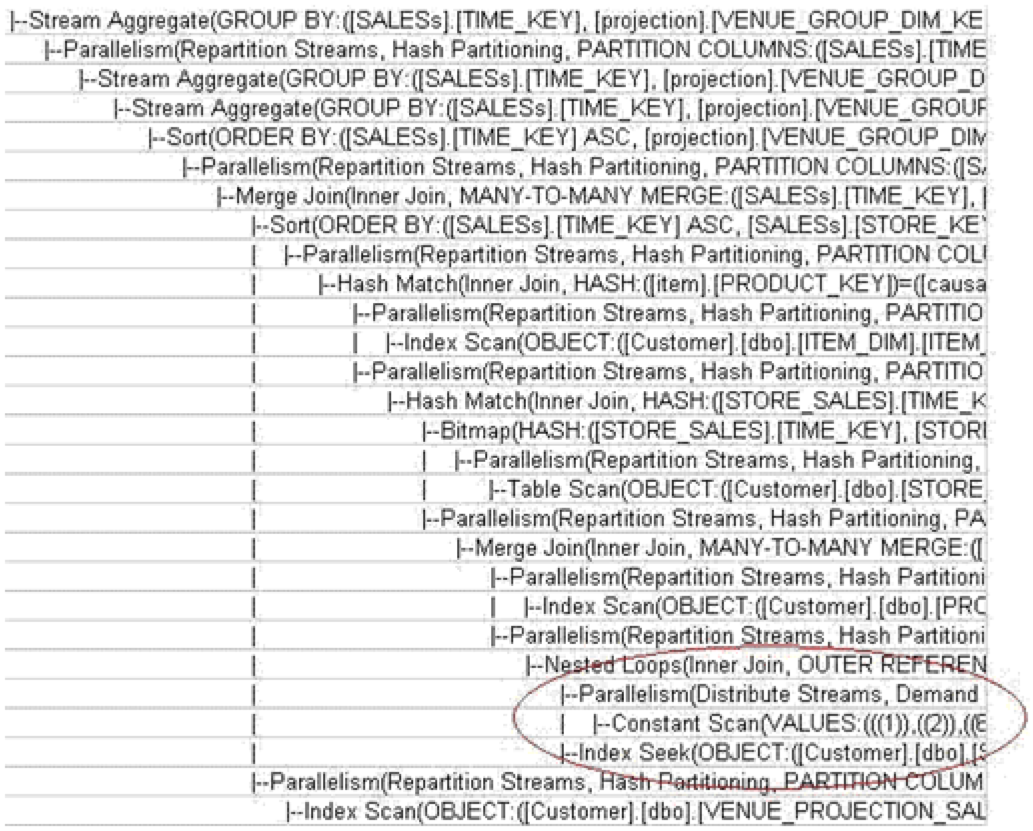
Queries against partitioned views might perform better in some cases. The following are examples of such cases:

- The smallest unit of parallelism, when demand parallelism is chosen by the optimizer, is a partition. The performance of queries against a single partition in a SQL Server 2005 partitioned table might suffer because the degree of parallelism is limited to one. The same query against a partitioned view might perform better because of better parallelism within a partition.
- Partitioned views might perform better when the number of partitions is less than the number of processors because of better use of the available processors through parallelism. When the number of partitions is greater than the number of processors, the performance of queries against partitioned tables can still suffer if the data is not evenly distributed across the partitions.
- Queries against partitioned views might perform better when there is a skew in the distribution of data among the partitions.

Identifying Demand Parallelism in a Query Plan

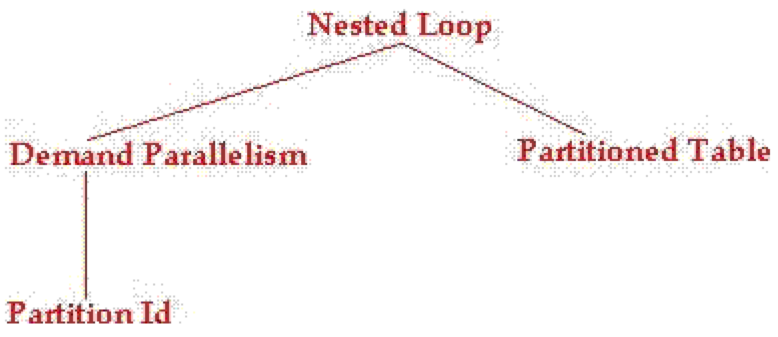
The following is an example of a query plan that was generated from an additive aggregation query.

The section circled in red indicates the presence of demand parallelism in the query plan. The left child of the nested loop operator is represented by the partition IDs. The right child of the nested loop operator is represented by the partitioned table itself. In this illustration, for every partition ID returned by the left child, a parallel index seek operator iterates over the rows from the appropriate partition. All operations above the nested loop operator are also influenced by the number of parallel threads established by demand parallelism. The left child represents only the partition IDs that are affected by the query when partition pruning is effective☐for example, when the query filters the results by partitions.



```
|--Stream Aggregate(GROUP BY:([SALESs].[TIME_KEY], [projection].[VENUE_GROUP_DIM_KE
    |--Parallelism(Repartition Streams, Hash Partitioning, PARTITION COLUMNS:([SALESs].[TIME
        |--Stream Aggregate(GROUP BY:([SALESs].[TIME_KEY], [projection].[VENUE_GROUP_D
            |--Stream Aggregate(GROUP BY:([SALESs].[TIME_KEY], [projection].[VENUE_GROUF
                |--Sort(ORDER BY:([SALESs].[TIME_KEY] ASC, [projection].[VENUE_GROUP_DIM
                    |--Parallelism(Repartition Streams, Hash Partitioning, PARTITION COLUMNS:([S
                        |--Merge Join(Inner Join, MANY-TO-MANY MERGE:([SALESs].[TIME_KEY], |
                            |--Sort(ORDER BY:([SALESs].[TIME_KEY] ASC, [SALESs].[STORE_KE
                            |   |--Parallelism(Repartition Streams, Hash Partitioning, PARTITION COLI
                            |       |--Hash Match(Inner Join, HASH:([item].[PRODUCT_KEY])=([causa
                            |           |--Parallelism(Repartition Streams, Hash Partitioning, PARTITIO
                            |           |   |--Index Scan(OBJECT:([Customer].[dbo].[ITEM_DIM].[ITEM_
                            |           |--Parallelism(Repartition Streams, Hash Partitioning, PARTITIO
                            |               |--Hash Match(Inner Join, HASH:([STORE_SALES].[TIME_K
                            |                   |--Bitmap(HASH:([STORE_SALES].[TIME_KEY], [STORI
                            |                   |   |--Parallelism(Repartition Streams, Hash Partitioning,
                            |                   |       |--Table Scan(OBJECT:([Customer].[dbo].[STORE
                            |                   |--Parallelism(Repartition Streams, Hash Partitioning, PA
                            |                       |--Merge Join(Inner Join, MANY-TO-MANY MERGE:([
                            |                           |--Parallelism(Repartition Streams, Hash Partitioni
                            |                           |   |--Index Scan(OBJECT:([Customer].[dbo].[PRC
                            |                           |--Parallelism(Repartition Streams, Hash Partitioni
                            |                               |--Nested Loops(Inner Join, OUTER REFEREN
                            |                                   |--Parallelism(Distribute Streams, Demand
                            |                                   |   |--Constant Scan(VALUES:(((1)),((2)),((E
                            |                                   |--Index Seek(OBJECT:([Customer].[dbo].[S
                            |--Parallelism(Repartition Streams, Hash Partitioning, PARTITION COLUM
                                |--Index Scan(OBJECT:([Customer].[dbo].[VENUE_PROJECTION_SAL
```

Figure 1. Identifying demand parallelism
See full-sized image



Migrating to SQL Server 2005 Partitioned Tables/Indexes from SQL Server 2000 Partitioned Views

An existing application based on monolithic table or partitioned views can be re-architected or migrated to a SQL Server 2005 solution based on partitioning. The decision to re-architect or migrate the application entails a detailed analysis of performance, manageability, and availability requirements.

A simple path for migrating SQL Server 2000 partitioned views to SQL Server 2005 partitioned tables would include the following steps:

- Create a partition function and scheme to determine the boundary points and physical storage location for each partition. The boundary points should mimic those of the partitioned view's underlying tables.
- Create a partitioned table on the newly created partition scheme. The table should specify the same physical structure, including indexes, as the tables underlying the partitioned view.

- Switch the individual tables underlying the partitioned views into each partition of the newly created partitioned fact table. The filegroups referenced in the partition scheme must match the filegroup of the table that is being switched in. In addition, the tables to be migrated have to comply with the requirements of the switch instruction. For example, the target table cannot be a component of a view with schema binding. For a list of the requirements of the switch instruction, see "Transferring Data Efficiently with Partition Switching" in SQL Server 2005 Books Online.

Top of page

Factors Influencing Partitioning for Relational Data Warehouses

The successful implementation of a partitioned relational data warehouse involves planning for the growth of the database and easy manageability. The next sections explain the factors that influence partitioning for relational data warehouses and include details on sliding window implementations.

Data Volume

A partitioned approach will add more management complexity without adding much value when the fact tables are smaller in size. This size is based on the application characteristics and should be determined for each implementation. As a rule of thumb, several customers have required that fact tables be at least 100 GB before they will implement partitioning.

Data Loading

Data loading is an integral part of a data warehouse. Almost all data warehouses are periodically populated with recently collected data. Managing a successful data warehouse depends on the efficiency of the bulk loading process during which existing data must continue to be available.

There are two options for building your fact table:

- Build a single, monolithic table, or
- Use a partitioned approach

The monolithic table approach results in lower availability compared to a partitioned approach because of the incremental bulk loading that is performed in typical relational data warehouse environments. For example, incremental bulk loading can greatly benefit from taking a table lock on the target table. With a single table, this will block all other users from accessing the table during data loading. The optimal workaround is to have a planned maintenance window for incrementally loading data. For a complete discussion on bulk loading using a monolithic table approach, see "SQL Server 2000 Incremental Bulk Load Case Study" at http://www.microsoft.com/technet/prodtechnol/sql/2000/maintain/incbulkload.mspx.

The partitioned approach entails the bulk loading of data into individual staging tables, each of which represents a determined partition range. The staging table is then added to the partitioned view or switched in to the partitioned table as a new partition. Because each partition is logically represented by an individual staging table, incremental bulk loads do not affect the availability and performance of any queries against the existing data.

A typical data warehousing solution would include the transformation of data along with the bulk loading of data. Transformation includes the cleansing and/or aggregation of source data to populate the destination warehouse.

A transformation is typically accomplished by using tools such as Microsoft System Integration Services. The customers can optionally choose to use SELECT/INTO to do this transformation if the process does not require a complex workflow.

Indexing

After loading the data in a relational data warehouse, it is common to build indexes to support user queries. The building and maintaining of indexes plays a major role in influencing the architecture of relational data warehouses.

The performance of queries against fact tables often suffers without indexes. For a monolithic fact table, an optimal solution could be to drop all indexes, load the data, and rebuild the indexes. This approach results in reduced availability and a growing maintenance window as the monolithic table increases in size making it impractical in some cases.

Partitioned views address this problem effectively in SQL Server 2000 as the indexes are built on underlying tables. SQL Server 2005 supports rebuilding and reorganizing indexes on individual partitions, thereby facilitating better management of partitioned indexes.

Data Aging

Aged data is accessed less frequently than new data. Increasing laws and regulations require businesses to keep aged data online for immediate access. Hence, it is critical that a company manage aged data effectively while maintaining high availability for existing data and facilitating the faster loading of new data. Data aging can be handled effectively by a sliding window. If the data is partitioned, a sliding window implementation is possible. For more details, see "Sliding Window Implementation" later in this paper.

Data Archival

The successful implementation of a multiterabyte data warehouse does not end with building a well-performing and linearly scaling system. It also depends on maintaining a highly available system.

If data is partitioned, piecemeal backup in SQL Server can be implemented. Piecemeal backup and restore operations in SQL Server offer more flexibility for managing partitions. Piecemeal operations imply that individual partition(s), when confined to their own filegroup(s), can be backed up and restored individually without affecting the entire database. The filegroup has to be in read-only mode for piecemeal backup to work in a Simple Recovery model. In the case of either the Bulk-Logged Recovery model or the Full Recovery model, it is necessary to back up the transaction log(s). Doing so is essential to restore the filegroup successfully. For more details on these restrictions, see "Backup (Transact-SQL)" in SQL Server Books Online.

Query Performance

Ad-hoc queries are an integral part of relational data warehousing solutions. The characteristics of the application and the resulting nature of queries greatly influence the partitioning of relational data warehouses. For example, if the queries have a filtering key that corresponds to the partitioning key, response time will be faster compared to the same query against a monolithic table. This is because the use of partitioning encourages the use of parallel operations and the partitioning key in the query predicate makes pruning data easier.

Top of page

Sliding Window Implementation

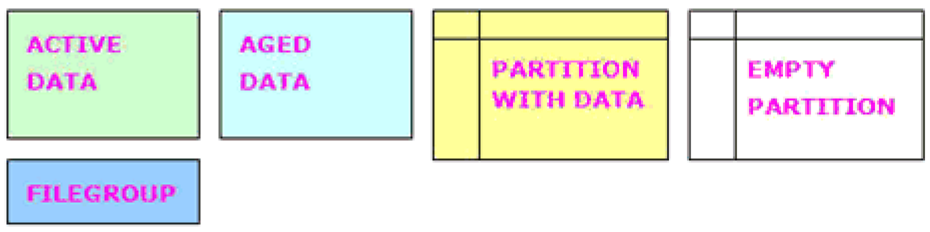
Sliding Window is one of the key factors influencing partitioning for relational data warehouses and deserves a separate section to discuss the implementation details.

The sliding window scenario involves rolling new partitions in and rolling aged partitions out of the partitioned table or view. The new data is available for the users to query while the aged data is archived. The key is to minimize downtime while moving the partitions.

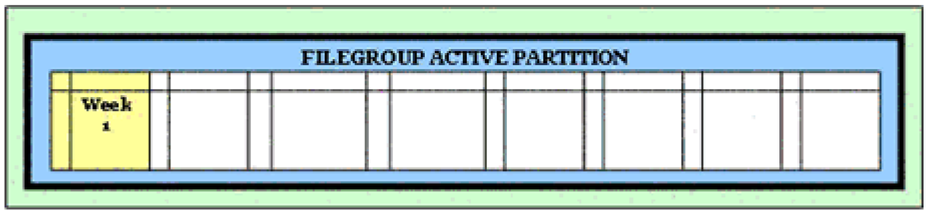
The aged data can be archived and can be retrieved when necessary by restoring the appropriate backups, or it can be moved to a less permanent, more affordable I/O subsystem that is still available to users for queries.

The following illustration represents a sliding window implementation from our test scenario. In our test scenario, the customer gathers retail sales data from stores across the country. The data is loaded, cleansed, and aggregated to support business decisions. In our test scenario, a partition logically represents one week's worth of data. Currently, eight weeks' worth of data is identified as active. Active data is queried much more frequently than aged data. As new data comes in, the aged data slides out. There is a business rule stating that aged data should remain online but be stored on a cost-effective I/O subsystem.

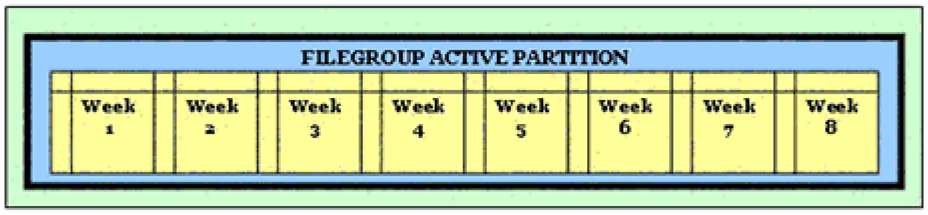
Color Index



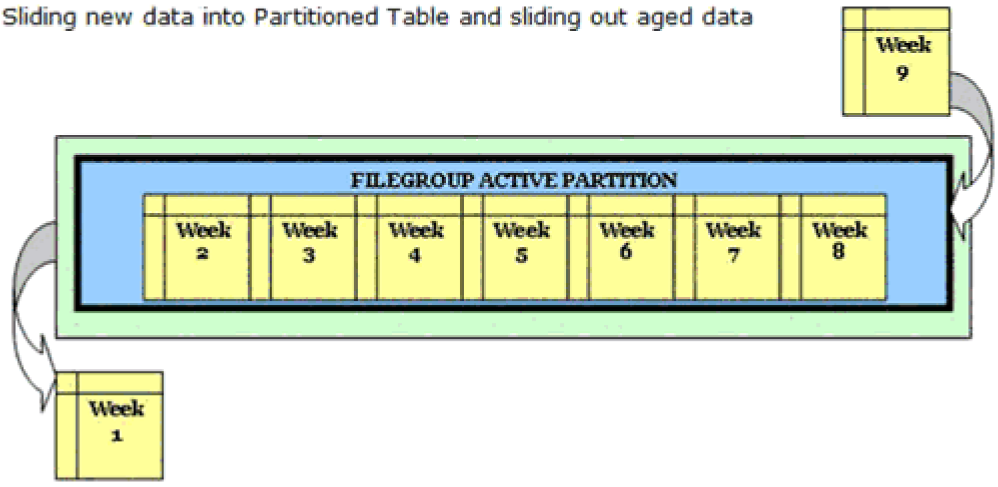
Partitioned Table with 1st week's data



Partitioned Table with 8 weeks of data



Sliding new data into Partitioned Table and sliding out aged data



Active and Aged Data

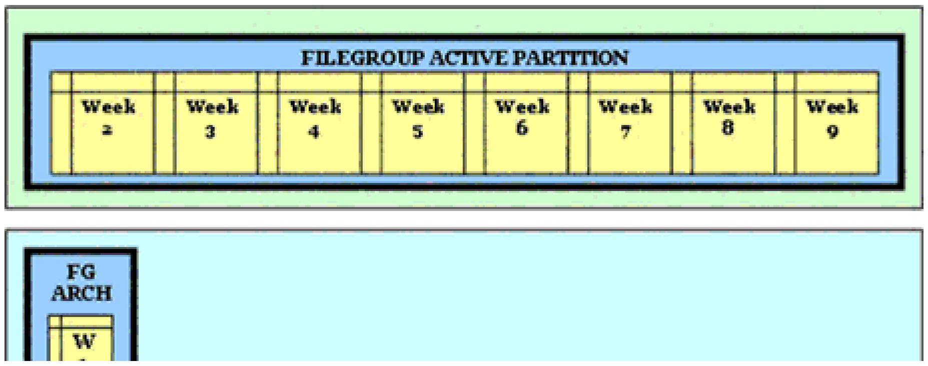


Figure 2. Sliding window scenario
See full-sized image

In SQL Server 2000, sliding windows can be implemented using partitioned views. The drawback is that the partitioned view has to be rebound to include the newly populated data in the UNION view. The rebinding requires a metadata lock and could be blocked by any access to the existing view or underlying tables.

SQL Server 2005 facilitates a better implementation of the sliding window scenario by supporting the switching in and switching out of partitions using Transact-SQL statements. Switching partitions requires a schema lock on the partitioned table. Partitions can be switched in and out when no other process has acquired a table-level lock on the partitioned table. If the partitions are used by other processes or if a table-level lock has been acquired by other processes on the partitioned table, the instruction to switch partitions will wait until the lock has been released by other processes. Partition switching is a metadata operation and is very fast.

The following steps can be used to implement a sliding window scenario in SQL Server 2005 using partitioned tables:

- Create the partition function, scheme, and table with appropriate boundary points and filegroups. Then do the initial load as described in the next four steps.
- Create tables representing individual partitions.
- Populate the tables individually.
- Add check constraints to tables to bind the data value to corresponding ranges and create appropriate indexes. SQL Server 2005 presents the additional option of creating the initial index after creating the partitioned table.
- Switch in the newly populated tables into each partition of the partitioned table.
- After initial loads, any new data is loaded and transformed in a table that is not a part of this partitioned table. When the data is ready, the table is switched in to the partitioned table after manipulating the appropriate boundary points.
- Similarly, aged data can be moved to more cost-effective I/O subsystem but still be kept available online.

The next section covers some best practices for splitting the partitioned table and switching partitions into the partitioned table.

Best Practices for Switching Partitions

The sliding window scenario works only when the target table or partition is empty. For example, if a partition "P" belonging to the partitioned table "PT" has to be switched out to table "T," then target table "T" has to be empty. Similarly, when switching in table "T" into partition "P" of partitioned table "PT," the target partition "P" should be empty.

The sliding window scenario works best when data movement across partitions is minimized. The following code defines the partition function and partition scheme. When a table is created on this partition scheme, the partitioned table will have three partitions. The first partition will contain data with key values <=200401; the second with key values >200401 and <=200403; the third with key values >200403.

```
CREATE PARTITION FUNCTION SALES_MONTHLY_PARTITION_FUNCTION (INT)
AS RANGE LEFT FOR VALUES ( 200401, 200403 )
GO
CREATE PARTITION SCHEME SALES_MONTHLY_PARTITION_SCHEME AS
   PARTITION SALES_MONTHLY_PARTITION_FUNCTION ALL TO ([PRIMARY])
GO
CREATE TABLE t
(
   col1 INT
)ON SALES_MONTHLY_PARTITION_SCHEME(col1)
GO
```

When a new boundary with a value of 200402 is added using the split functionality of the ALTER PARTITION function, the rows are moved between the appropriate partitions.

```
ALTER PARTITION FUNCTION PARTITION_FUNCTION()
SPLIT RANGE (200402)
GO
```

The rows are moved between partitions by deleting the rows at the original position and by inserting the rows at the new position. The partitions involved in this switch are inaccessible during this period. In this example, the new second partition will hold data with key values ranging >200401 and <=200402. Data with appropriate key values are deleted from the second partition and inserted into the new partition. The new partition (>200401 and <=200402) and the third partition (>200402 and <=200403) are inaccessible during this period.

In our customer scenario, the new data is added by splitting the partition function at the active end. Old data is removed by merging the partition function at the aged end. This practice, for implementing sliding window, eliminates the data movement across partitions when switching in or switching out partitions□for example, the new data is bulk loaded into a table and then switched in to the partitioned table by splitting it at the active end as follows:

```
ALTER TABLE NEW_PARTITION SWITCH TO PARTITIONED_TABLE
   PARTITION $partition.WEEK_PARTITION_FUNCTION (200404)
GO
```

For more information, see "Designing Partitions to Manage Subsets of Data" in SQL Server Books Online.

Techniques for Archiving Data to a Cost Effective I/O Subsystem

A sliding window implementation can be extended by sliding the aged data to a cost effective I/O subsystem. For example, in our test scenario, we slide out the aged data from a highly performing I/O subsystem to a less expensive I/O subsystem that does not have the same high performance. This particular sliding window implementation cannot be accomplished using the backup and restore operation available in SQL Server. There are a couple of alternate methods to implement such a strategy:

- If the source files are available, load the data into another table that resides on the cost-effective I/O subsystem. Rebuild the indexes. Delete the old partition and add the newly loaded table to the partitioned table. The downtime will be the time that is required to switch partitions which is very negligible, irrespective of the size of your data set.
- If the loading process involves transformation, it would be efficient to create the new table on the cost-effective I/O subsystem by using a SELECT/INTO query to populate the data from the aged partition and rebuild the indexes. The downtime will be the time required to switch partitions.

Top of page

Strategies for Partitioning Relational Data Warehouses

The next section explains two main strategies for partitioning your relational data warehouse. This section will be followed by a discussion on how the strategies affect the factors that influence partitioning.

Strategy I: Bind a Partition to Its Own Filegroup

A partition can be logically bound to a filegroup using the following steps:

- Create the database with several filegroups. Each filegroup will logically represent a partition.
- Each filegroup has one file. The filegroups can contain one or more physical files that are created from one or more logical volumes/physical disks.
- Create the partition function and map the boundary points to appropriate filegroups using partition schemes to create a one-to-one correlation between the filegroup and the partition.

For example code that shows how to implement partitioning based on this strategy, see Appendix D.

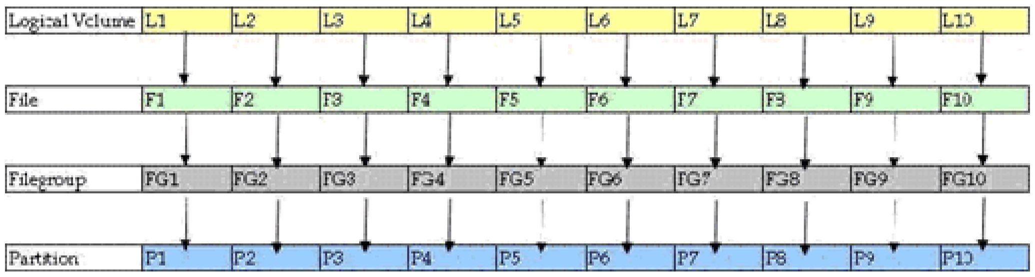


Figure 3. Partition mapped to its own filegroup
See full-sized image

Strategy II: Bind Two or More Partitions to the Same Filegroup

The second strategy is to have one or more partitions mapped to the same filegroup. The filegroup can be comprised of one (or more) physical files spread across one (or more) logical volumes/physical disks. For example code that implements partitioning based on this strategy, see Appendix D.

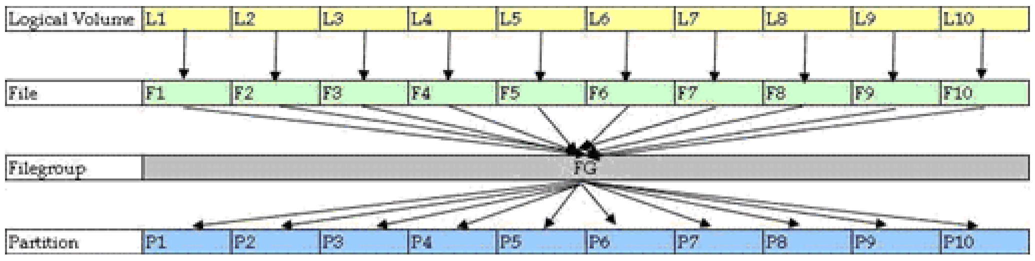


Figure 4. Two or more partitions mapped to the same filegroup
See full-sized image

Which Is Better?

Partitioning can be implemented by using either one of these strategies or else a synergistic combination of both strategies. Strategy I and Strategy II are referenced in the following table that explains their impact on the factors influencing partitioning for relational data warehouses.

	Strategy I	**Strategy II**
Data Loading	• Filegroups cannot be specified explicitly in the SELECT/INTO SQL statement. Because of this limitation, initial transformations cannot be done in parallel when using a SELECT/INTO statement. • The incremental data loads are not affected by the partitioning strategy.	• Transformations can be done in parallel as all the partitions are mapped to the same filegroup. • The incremental data loads are not affected by the partitioning strategy.
Backup/ Restore	• Creating a one-to-one relationship between the filegroup and the partition enables piecemeal backup and restore operations at partition level. • Make sure that the partitions are marked read-only before backing up the database. If not, transaction logs have to be rolled forward when restoring the database.	• If all the partitions are mapped to the same filegroups, the partitioned table as a whole can be backed up and restored using a single command. • This does not provide the flexibility of piecemeal backup at individual partition granularity.
Query		

Performance	• If the filegroups contain only one file and tables are populated one after another in serial, extent allocation for such objects is contiguous. This implies that SQL Server can issue up to 256 KB I/O (4 extents) for a sequential scan. • This benefits workloads favoring many sequential scans because the data is contiguous.	• If filegroups are comprised of several files, SQL Server uses proportional fill, resulting in extent fragmentation. • Similarly, extents allocated for the objects/partitions are not guaranteed to be contiguous during parallel operations such as parallel data loading. • SQL Server can issue only up to 64 KB I/O (1 extent) for sequential scan when the extents allocated for the objects are not contiguous. • This benefits workloads favoring lots of concurrent random I/Os because the data is spread across many physical disks. As an alternative, SQL Server can be started with the -E switch. SQL Server can allocate 4 extents instead of 1 extent when the -E switch is specified during startup. Hence, the -E switch enables SQL Server to issue 256 KB I/O even if extent fragmentation exists because of proportional fill.

Top of page

Conclusion

This white paper discussed the factors that influence partitioning, and compared the pros and cons of two key strategies for designing partitions. The information presented should help you manage your relational data warehouses more effectively through partitioning.

Top of page

For More Information

This document exposes specific features of SQL Server 2005 that are relevant to partitioning relational data warehouses. For more information, see:

- SQL Server 2005 Books Online offers a wealth of information on this subject and would serve as a good starting point for the implementation of data partitioning using SQL Server 2005.
- CLARiiON CX600 Web site: http://www.emc.com/products/systems/clariion_cx246.jsp

Top of page

Appendix A: Performance Numbers

All numbers reported in this section were observed during our tests with SQL Server 2005. The tests were done on the hardware platforms documented in Appendices B, C, and D.

Bulk Insert Performance

In our tests, we were able to load 2.5 billion rows in little more than one hour in SQL Server 2005. The schema of the fact table contains nine integer columns, one datetime column, and one character column.

The performance numbers represented here are captured when all threads were executing in parallel and the partitioning scheme was based on Strategy I.

Number of parallel threads	8
Execution Time (when all threads are executing)	52 minutes
Processor Use (across 8 processors)	88 percent
Number of rows inserted	2550835652
Bulk Copy Throughput/second	44.77 MB/second
Bulk Copy Rows/second	6,53,339
Disk Throughput	41.17 MB/second across 8 LUNs on CLARiiON

The performance numbers represented here are captured when all threads were executing in parallel and the partitioning scheme was based on Strategy II.

Number of parallel threads	8
Execution Time (when all threads are executing)	52 minutes
Processor Use (across 8 processors)	92.625 percent
Number of rows inserted	2,550,835,652
Bulk Copy Throughput/second	46.42 MB/second
Bulk Copy Rows/second	677285
Disk Throughput	44.29 MB/second across 8 LUNs on CLARiiON

The strategy chosen for partitioning the data warehouse did not affect the Bulk Load throughput.

Transformation Performance

In our tests, the bulk loading process was followed by a transformation process. The transformation process involved joining the source data with dimension tables in order to populate the destination warehouse with the extracted dimension key values. The following is a code example that was used in our test scenario.

```
SELECT fact.fact_data_1,
...
sdim.store_key, pdim.product_key, tdim.time_key
INTO sales_transformed
FROM
   sales fact,
   stores sdim,
   products pdim,
   time tdim
WHERE fact.store_id = sdim.store_id
AND convert(bigint, (fact.system + fact.generation +
   fact.vendor + fact.product)) = pdim.productid
AND fact.weekid = tdim.weekid
```

The transformation query was run consecutively, one query immediately following the other, so as to preserve the partitioning scheme set forth in Strategy I.

Number of parallel threads	1
Execution Time	1 hour 13 minutes for each transformation approximately
Processor Use (across 1 processor)	100 percent
Number of rows transformed (each transformation)	300 million rows approximately
Disk Read Throughput	3.5 MB/second across 1 LUN on CLARiiON (256 KB read)
Disk Write Throughput	2.8 MB/second across 1 LUN on CLARiiON (128 KB write)

The following are performance numbers from a partitioning scheme that was based on Strategy II when all threads were executing in parallel.

Number of parallel threads	8
Execution Time	1 hour 9 minutes
Processor Use (across 8 processors)	99 percent
Number of rows transformed (total)	2,478,765,081
Disk Read Throughput	33.10 MB/second across 10 LUNs on CLARiiON
Disk Write Throughput	26.66 MB/second across 10 LUNs on CLARiiON

Index Build Performance

In our tests, we were able to build a clustered index on three integer columns (dimension keys) across eight partitions with 2.5 billion rows in two hours. The SORT_IN_TEMPDB option allows **tempdb** to be used for sorting the data. This option was used during this test to isolate the physical disks used for reads and writes during the index build. When the "sort_in_tempdb" option is on, **tempdb** must have enough free space to hold the size of the entire index during offline index creation. When sorting is done in a user database, each filegroup/partition needs to have enough free disk space to hold the appropriate partition(s).

In our customer scenario and design, each partition logically represents a week's worth of data with a key value that is the same for all rows in any given partition. The maximum degree of parallelism for each individual index build will only be 1 because the number of distinct key values is 1. Hence, we built the initial index after creating the partitioned table in SQL Server 2005 to make better use of available processors for the initial index build.

Number of parallel threads	8
Execution Time	2 hours
Processor Use	86 percent

Number of rows	2,478,765,081

Database Backup Performance

In our tests, we backed up the active partition to four RAID 3 LUNs on EMC CLARiiON in a little more than one hour. The active partition was spread across eight LUNs on a RAID 5 (4+1 10K RPM) EMC CLARiiON array. The target device was a RAID 3 (4+1, 5K RPM ATA) CLARiiON array.

Execution Time	1 hour 20 minutes
Number of pages (8K) backed up	30,518,330
Backup/Restore Throughput/second	50.15 MB/second

Aging Data to ATA Disks

Aging data to ATA disks involves bulk loading data using the SELECT/INTO statement. The following is a code example used to implement the sliding window.

```
ALTER DATABASE [Customer]
   ADD FILEGROUP [ARCHIVE_1]
GO
ALTER DATABASE [Customer]
   ADD FILE (NAME = N'ARCHIVE_1', FILENAME =
      N'F:\ATA\RAID5\2\ARCHIVE_PARTITION_1.NDF',
      SIZE = 100GB, FILEGROWTH = 0)
   TO FILEGROUP [ARCHIVE_1]
GO
ALTER DATABASE Customer MODIFY FILEGROUP [ARCHIVE_1] DEFAULT
GO
SELECT * INTO iri..Sales_129_ARCHIVE
   FROM VISF
   WHERE TIME_KEY = 129
   OPTION (MAXDOP 1)
```

A clustered index identical to the clustered index on the existing partitioned table has to be created on the new table to be able to later switch in the partitioned table. In our tests, we used SELECT/INTO to slide out the first aged partition with 279 million rows to an ATA disk array in less than 20 minutes. The clustered index creation on the same table was completed in less than 40 minutes.

Top of page

Appendix B: Platform Listing

The following hardware and software components were used for the tests described in this paper:

Microsoft Software

Microsoft Windows Server 2003 Datacenter Edition Build 3790

Microsoft SQL Server 2005 Beta 2

Server Platform

64-bit Unisys ES7000 Orion 130

16 Itanium 2 1.30 GHz CPUs with 3 MB Cache

64 GB of RAM

Storage

EMC Symmetrix 5.5

96, 72 GB 10 KB RPM disks 16 GB write-read cache

EMC CLARiiON FC4700

155, 133.680 GB, 10 KB RPM disks 16 GB write-read cache

30, 344 GB, 5 KB RPM disks (ATA)

Host Bus Adapter

8 Emulex LP9002L 2 GB/second fibre channel host bus adapters

All adapters mapped to CLARiiON storage array were load balanced through Powerpath software

Storage Management Software

EMC Powerpath v 3.0.5

Top of page

Appendix C: Server Architecture

We used a Unisys ES7000 Orion 130 server for our testing. It is important to understand your server architecture as it is one of the factors that will determine the theoretical throughput of your system. For example, a 100 MHz PCI bus is capable of 800 Megabit/second throughput. This appendix briefly illustrates the ES7000 Orion architecture.

The Enterprise Server ES7000 Orion 130 is a modular rack mount system based on an Intel chipset and Intel Itanium 2 family of processors.

The Orion server used during the tests was configured with 2 server modules, 16 processors, and 64 GB of memory. Each server module has an IOHM (Input/Output Hub Module). Each IOHM controls one I/O subsystem that consists of four PCIAMs (PCI Adapter Module). Each PCIAM has 2 buses, each serving 2 PCI card slots. The PCI cards are supported up to 100 MHz on each of this bus.

The slots that are highlighted are currently configured. Slot 2 on BUS 0-0 and BUS 0-1 on PCIAM 0-0/IOHM-0 are configured for local SCSI disks and network adapter, leaving us with 16 slots. Eight of these are mapped to CLARiiON Storage Array using Powerpath software.

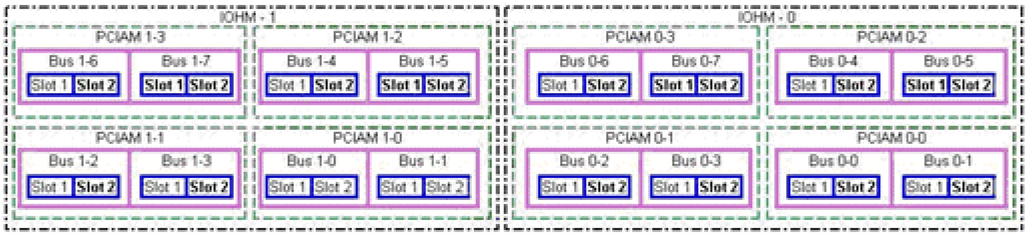


Figure 5. I/O Configuration of CASSIN server used for this scenario
See full-sized image

Top of page

Appendix D: EMC CLARiiON Storage

We used a CLARiiON CX600 storage array for our testing. This appendix outlines the configuration information of the storage array used.

The CLARiiON storage array consists of a series of trays or enclosures. Each tray is capable of holding up to 15 disks. A number of disks across the trays are grouped together to form a raid group. The logical unit number (LUN) created from the raid group are exposed to the Microsoft Windows operating system. If more storage needs to be added in the future, disks can be added to the storage trays and grouped together as raid groups. The newly created raid groups can be joined to any existing raid group, forming a Meta-LUN which is exposed to the Windows operating system. The Meta-LUN will retain the LUN number of the source LUN. The pattern in which these physical disks are grouped as raid groups does not affect performance because the design of CLARiiON storage arrays balances the load across the trays and physical disks.

The EMC CLARiiON storage array used for our tests had a mix of RAID 1+0, RAID 5, and RAID 3 configurations. RAID 1+0 is used for database log files and RAID 5 is used for the data files. RAID 3 was used to back up and restore database partitions. The 10KB RPM physical disks are 133 GB in size and the 5KB RPM disks are 344 GB in size.

Our storage configuration included five RAID 1 + 0 LUNs. The LUNs are carved out of a raid group comprising eight physical disks. Because the disks are mirrored and striped, the capacity of each of these LUNs is approximately 500 GB ((8*133.680 GB)/2) in size. Two of these RAID 1+0 LUNs were used for our database log file.

Twelve RAID 5 LUNs were used for data files. Five RAID 5 LUNs were used for **tempdb**. Each of these LUNs had an approximate capacity of 500 GB. Each LUN is carved from a raid group. The raid group configuration for RAID 5 is comprised of fewer disks than RAID 1 + 0 raid groups. The RAID 5 raid groups are comprised of five 133 GB disks.

The following is an illustration of the raid groups and disks arranged in a CLARiiON storage array. Three physical disks from Tray 2 and two physical disks from Tray 1 are grouped together as a raid group. The next raid group is carved out by alternating the number of disks from each tray. The physical disks are interleaved between the trays, as a best practice, to sustain any failures at the tray level.

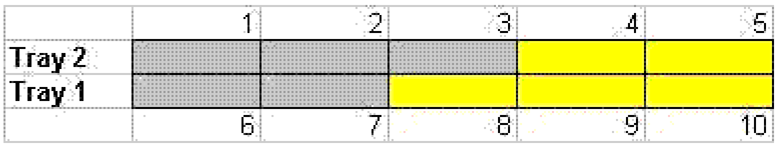


Figure 6. CLARiiON RAID 3/5 and raid group configuration

Topology

For our tests, we used fibre channel technology and implemented a standard switched-fabric topology. The fabric channel was soft zoned using the World Wide Name of Fiber Host Bus Adapters (HBAs) at the switch level. The LUNs were all masked to be visible to the host server at the storage controller for this test scenario.

Figure 7 depicts the mapping between the HBA, the storage port, and the LUNs. Because the logical volumes directly mapped to a single LUN in our test scenario, the illustration can be considered as a mapping to a logical volume.

EMC's Powerpath (multi-path software) was used to load balance I/O across all the LUNs on the CLARiiON storage array. Eight of the Emulex HBAs were zoned to see all the LUNS carved on CLARiiON storage array. For full details on installing Emulex HBAs with EMC storage, visit http://www.emulex.com/ts/docoem/framemc.htm.

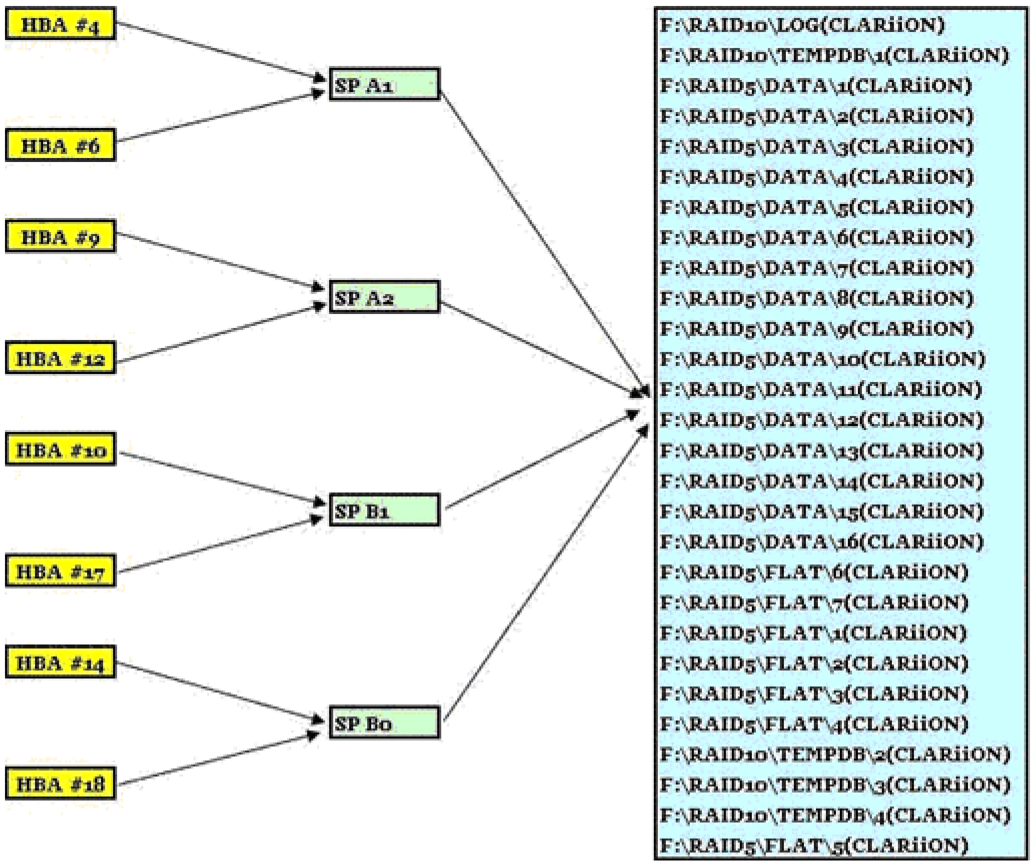

HBA #4
HBA #6
HBA #9
HBA #12
HBA #10
HBA #17
HBA #14
HBA #18
SP A1
SP A2
SP B1
SP B0
F:\RAID10\LOG(CLARiiON)
F:\RAID10\TEMPDB\1(CLARiiON)
F:\RAID5\DATA\1(CLARiiON)
F:\RAID5\DATA\2(CLARiiON)
F:\RAID5\DATA\3(CLARiiON)
F:\RAID5\DATA\4(CLARiiON)
F:\RAID5\DATA\5(CLARiiON)
F:\RAID5\DATA\6(CLARiiON)
F:\RAID5\DATA\7(CLARiiON)
F:\RAID5\DATA\8(CLARiiON)
F:\RAID5\DATA\9(CLARiiON)
F:\RAID5\DATA\10(CLARiiON)
F:\RAID5\DATA\11(CLARiiON)
F:\RAID5\DATA\12(CLARiiON)
F:\RAID5\DATA\13(CLARiiON)
F:\RAID5\DATA\14(CLARiiON)
F:\RAID5\DATA\15(CLARiiON)
F:\RAID5\DATA\16(CLARiiON)
F:\RAID5\FLAT\6(CLARiiON)
F:\RAID5\FLAT\7(CLARiiON)
F:\RAID5\FLAT\1(CLARiiON)
F:\RAID5\FLAT\2(CLARiiON)
F:\RAID5\FLAT\3(CLARiiON)
F:\RAID5\FLAT\4(CLARiiON)
F:\RAID10\TEMPDB\2(CLARiiON)
F:\RAID10\TEMPDB\3(CLARiiON)
F:\RAID10\TEMPDB\4(CLARiiON)
F:\RAID5\FLAT\5(CLARiiON)

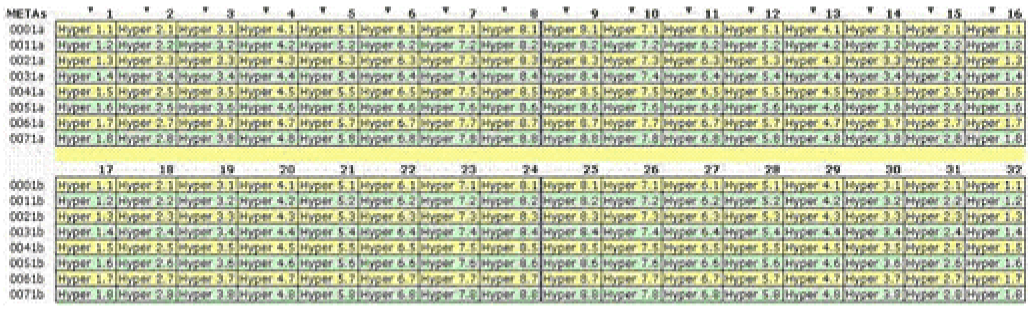
Figure 7. HBA-storage port-volume mapping
See full-sized image

⇧ Top of page

Appendix E: Storage Isolation

Storage isolation plays a significant role in Storage Area Network environments, especially where data is shared between one or more applications. This appendix highlights some of the considerations for storage isolation when partitioning your relational data warehouse. The example in this appendix highlights the EMC Symmetrix storage array which was used in our initial testing.

The logical volume can be created from several physical disks. Similarly, several logical volumes can be created from the same physical disk. And in some cases, several logical volumes can be created from a set of physical disks. This practice is more common in Storage Area Network environments. In EMC Symmetrix, physical disks are logically divided into hyper volumes. Hyper volumes from several physical disks are grouped to create a Meta volume. Meta volumes are grouped to create a logical volume. In this illustration, the numbers 1 to 32 represent individual physical disks. Each physical disk is divided into eight 9-GB hyper volumes. Hyper volumes 1.1 to 8.1 are grouped to create a Meta volume. Meta volumes, in this illustration, are numbers 0001a, 0011a to 0071b. Meta volumes 0001a and 0001b are grouped together to create a logical volume. The physical disks to the right of the thick vertical line represent the mirror pair.



METAs	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0001a	Hyper 1.1	Hyper 2.1	Hyper 3.1	Hyper 4.1	Hyper 5.1	Hyper 6.1	Hyper 7.1	Hyper 8.1	Hyper 8.1	Hyper 7.1	Hyper 6.1	Hyper 5.1	Hyper 4.1	Hyper 3.1	Hyper 2.1	Hyper 1.1
0011a	Hyper 1.2	Hyper 2.2	Hyper 3.2	Hyper 4.2	Hyper 5.2	Hyper 6.2	Hyper 7.2	Hyper 8.2	Hyper 8.2	Hyper 7.2	Hyper 6.2	Hyper 5.2	Hyper 4.2	Hyper 3.2	Hyper 2.2	Hyper 1.2
0021a	Hyper 1.3	Hyper 2.3	Hyper 3.3	Hyper 4.3	Hyper 5.3	Hyper 6.3	Hyper 7.3	Hyper 8.3	Hyper 8.3	Hyper 7.3	Hyper 6.3	Hyper 5.3	Hyper 4.3	Hyper 3.3	Hyper 2.3	Hyper 1.3
0031a	Hyper 1.4	Hyper 2.4	Hyper 3.4	Hyper 4.4	Hyper 5.4	Hyper 6.4	Hyper 7.4	Hyper 8.4	Hyper 8.4	Hyper 7.4	Hyper 6.4	Hyper 5.4	Hyper 4.4	Hyper 3.4	Hyper 2.4	Hyper 1.4
0041a	Hyper 1.5	Hyper 2.5	Hyper 3.5	Hyper 4.5	Hyper 5.5	Hyper 6.5	Hyper 7.5	Hyper 8.5	Hyper 8.5	Hyper 7.5	Hyper 6.5	Hyper 5.5	Hyper 4.5	Hyper 3.5	Hyper 2.5	Hyper 1.5
0051a	Hyper 1.6	Hyper 2.6	Hyper 3.6	Hyper 4.6	Hyper 5.6	Hyper 6.6	Hyper 7.6	Hyper 8.6	Hyper 8.6	Hyper 7.6	Hyper 6.6	Hyper 5.6	Hyper 4.6	Hyper 3.6	Hyper 2.6	Hyper 1.6
0061a	Hyper 1.7	Hyper 2.7	Hyper 3.7	Hyper 4.7	Hyper 5.7	Hyper 6.7	Hyper 7.7	Hyper 8.7	Hyper 8.7	Hyper 7.7	Hyper 6.7	Hyper 5.7	Hyper 4.7	Hyper 3.7	Hyper 2.7	Hyper 1.7
0071a	Hyper 1.8	Hyper 2.8	Hyper 3.8	Hyper 4.8	Hyper 5.8	Hyper 6.8	Hyper 7.8	Hyper 8.8	Hyper 8.8	Hyper 7.8	Hyper 6.8	Hyper 5.8	Hyper 4.8	Hyper 3.8	Hyper 2.8	Hyper 1.8

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0001b	Hyper 1.1	Hyper 2.1	Hyper 3.1	Hyper 4.1	Hyper 5.1	Hyper 6.1	Hyper 7.1	Hyper 8.1	Hyper 8.1	Hyper 7.1	Hyper 6.1	Hyper 5.1	Hyper 4.1	Hyper 3.1	Hyper 2.1	Hyper 1.1
0011b	Hyper 1.2	Hyper 2.2	Hyper 3.2	Hyper 4.2	Hyper 5.2	Hyper 6.2	Hyper 7.2	Hyper 8.2	Hyper 8.2	Hyper 7.2	Hyper 6.2	Hyper 5.2	Hyper 4.2	Hyper 3.2	Hyper 2.2	Hyper 1.2
0021b	Hyper 1.3	Hyper 2.3	Hyper 3.3	Hyper 4.3	Hyper 5.3	Hyper 6.3	Hyper 7.3	Hyper 8.3	Hyper 8.3	Hyper 7.3	Hyper 6.3	Hyper 5.3	Hyper 4.3	Hyper 3.3	Hyper 2.3	Hyper 1.3
0031b	Hyper 1.4	Hyper 2.4	Hyper 3.4	Hyper 4.4	Hyper 5.4	Hyper 6.4	Hyper 7.4	Hyper 8.4	Hyper 8.4	Hyper 7.4	Hyper 6.4	Hyper 5.4	Hyper 4.4	Hyper 3.4	Hyper 2.4	Hyper 1.4
0041b	Hyper 1.5	Hyper 2.5	Hyper 3.5	Hyper 4.5	Hyper 5.5	Hyper 6.5	Hyper 7.5	Hyper 8.5	Hyper 8.5	Hyper 7.5	Hyper 6.5	Hyper 5.5	Hyper 4.5	Hyper 3.5	Hyper 2.5	Hyper 1.5
0051b	Hyper 1.6	Hyper 2.6	Hyper 3.6	Hyper 4.6	Hyper 5.6	Hyper 6.6	Hyper 7.6	Hyper 8.6	Hyper 8.6	Hyper 7.6	Hyper 6.6	Hyper 5.6	Hyper 4.6	Hyper 3.6	Hyper 2.6	Hyper 1.6
0061b	Hyper 1.7	Hyper 2.7	Hyper 3.7	Hyper 4.7	Hyper 5.7	Hyper 6.7	Hyper 7.7	Hyper 8.7	Hyper 8.7	Hyper 7.7	Hyper 6.7	Hyper 5.7	Hyper 4.7	Hyper 3.7	Hyper 2.7	Hyper 1.7
0071b	Hyper 1.8	Hyper 2.8	Hyper 3.8	Hyper 4.8	Hyper 5.8	Hyper 6.8	Hyper 7.8	Hyper 8.8	Hyper 8.8	Hyper 7.8	Hyper 6.8	Hyper 5.8	Hyper 4.8	Hyper 3.8	Hyper 2.8	Hyper 1.8

Figure 8: Logical view of Symmetrix storage array disk layout
See full-sized image

For example, we could create two logical volumes; one, called L1, is created by grouping Meta volumes 0001a, 0001b and the other, L2, created by grouping Meta volumes 0011a, 0011b. If these logical volumes experience high number of concurrent I/O requests, disk thrashing occurs because the hyper volumes are created from the same set of underlying physical disks. To avoid disk thrashing in such cases, care should be taken to create partitions that are likely to be accessed concurrently on isolated logical volumes and physical disks.

As when designing storage for any data warehouse, there exists a balance between isolating data at the physical level and arranging the design such that the hardware is fully used.

The data for the active partition and the archive partitions were placed on common physical spindles but separated on separate logical volumes for monitoring and management. An active partition can be created on logical volume L1 and an archive partition can be created on logical volume L2. Though these two partitions are logically isolated, they share the same set of physical spindles.

The alternative to this approach would have been to isolate the active data from the archive data at the physical spindle level. This would have decreased the number of spindles used for the active data partitions, potentially leading to an underutilization of the spindles used for the archive partitions because that data is accessed less frequently.

With modern Storage Area Network devices equipped with large cache storage and shared unpredictable usage patterns, it is preferable to spread the data across as many spindles as is practical. This maximizes resource use and relies on the hardware to provide acceptable performance.

Configuring Your Storage

The storage array was optimally configured after working closely with EMC engineers based on our scenario requirements. We encourage our customers to work closely with their hardware vendors when configuring storage for a SQL Server implementation.

Top of page

Appendix F: Scripts

Following is the script that was used to create the database, partition function, and partition scheme in the Strategy II implementation described earlier in this white paper in the section called "Strategies for Partitioning Relational Data Warehouses."

```
CREATE DATABASE [SALES]
ON PRIMARY
(
NAME = N'SALES_PRIMARY', FILENAME = N'D:\SALESPrimary\SALES_PRIMARY.MDF',
   SIZE = 100MB, FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition1
(NAME = N'SALES_ACTIVE_PARTITION_1', FILENAME =
   N'F:\RAID5\DATA\1\SALES_ACTIVE_PARTITION_1.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition2
(NAME = N'SALES_ACTIVE_PARTITION_2', FILENAME =
   N'F:\RAID5\DATA\2\SALES_ACTIVE_PARTITION_2.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition3
(NAME = N'SALES_ACTIVE_PARTITION_3', FILENAME =
   N'F:\RAID5\DATA\3\SALES_ACTIVE_PARTITION_3.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition4
(NAME = N'SALES_ACTIVE_PARTITION_4', FILENAME =
   N'F:\RAID5\DATA\4\SALES_ACTIVE_PARTITION_4.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition5
(NAME = N'SALES_ACTIVE_PARTITION_5', FILENAME =
   N'F:\RAID5\DATA\5\SALES_ACTIVE_PARTITION_5.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition6
(NAME = N'SALES_ACTIVE_PARTITION_6', FILENAME =
   N'F:\RAID5\DATA\6\SALES_ACTIVE_PARTITION_6.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition7
   (NAME = N'SALES_ACTIVE_PARTITION_7', FILENAME =
   N'F:\RAID5\DATA\7\SALES_ACTIVE_PARTITION_7.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition8
(NAME = N'SALES_ACTIVE_PARTITION_8', FILENAME =
   N'F:\RAID5\DATA\8\SALES_ACTIVE_PARTITION_8.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition9
(NAME = N'SALES_ACTIVE_PARTITION_9', FILENAME =
   N'F:\RAID5\DATA\9\SALES_ACTIVE_PARTITION_9.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition10
(NAME = N'SALES_ACTIVE_PARTITION_10', FILENAME =
   N'F:\RAID5\DATA\10\SALES_ACTIVE_PARTITION_10.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_DIMENSIONS
(NAME = N'SALES_DIMENSIONS_1', FILENAME =
   N'F:\RAID5\DATA\11\SALES_DIMENSIONS_1.NDF', SIZE = 450GB,
   FILEGROWTH = 0),
(NAME = N'SALES_DIMENSIONS_2', FILENAME =
   N'F:\RAID5\DATA\12\SALES_DIMENSIONS_2.NDF', SIZE = 450GB,
   FILEGROWTH = 0)
LOG ON
(NAME = N'SALES_LOG', FILENAME = N'F:\SQL\Path8\SALES_LOG.LDF',
   SIZE = 120GB, FILEGROWTH = 0)
GO
CREATE PARTITION FUNCTION SALES_WEEK_PARTITION_FUNCTION (INT)
AS RANGE RIGHT FOR VALUES ( 129, 130, 131, 132, 133, 134, 135, 136 )
GO
CREATE PARTITION SCHEME SALES_WEEK_PARTITION_SCHEME AS PARTITION
   SALES_WEEK_PARTITION_FUNCTION
   TO
(
SALES_ActivePartition1, SALES_ActivePartition2, SALES_ActivePartition3,
   SALES_ActivePartition4, SALES_ActivePartition5,
SALES_ActivePartition6, SALES_ActivePartition7, SALES_ActivePartition8,
   SALES_ActivePartition9, SALES_ActivePartition10
)
GO
```

Following is the script that was used to create the database, partition function, and partition scheme in the Strategy II implementation described earlier in this white paper.

```
CREATE DATABASE [SALES]
ON PRIMARY
   (NAME = N'SALES_PRIMARY', FILENAME =
   N'D:\SALESPrimary\SALES_PRIMARY.MDF', SIZE = 100MB, FILEGROWTH = 0),
FILEGROUP SALES_ActivePartition
   (NAME = N'SALES_ACTIVE_PARTITION_1', FILENAME =
   N'F:\SQL\Path2\1\SALES_ACTIVE_PARTITION_1.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_ACTIVE_PARTITION_2', FILENAME =
   N'F:\SQL\Path2\2\SALES_ACTIVE_PARTITION_2.NDF', SIZE = 100GB,
   FILEGROWTH = 0),□□□□
   (NAME = N'SALES_ACTIVE_PARTITION_3', FILENAME =
   N'F:\SQL\Path3\1\SALES_ACTIVE_PARTITION_3.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_ACTIVE_PARTITION_4', FILENAME =
   N'F:\SQL\Path3\2\SALES_ACTIVE_PARTITION_4.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_ACTIVE_PARTITION_5', FILENAME =
   N'F:\SQL\Path4\1\SALES_ACTIVE_PARTITION_5.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_ACTIVE_PARTITION_6', FILENAME =
   N'F:\SQL\Path4\2\SALES_ACTIVE_PARTITION_6.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_ACTIVE_PARTITION_7', FILENAME =
   N'F:\SQL\Path5\1\SALES_ACTIVE_PARTITION_7.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_ACTIVE_PARTITION_8', FILENAME =
   N'F:\SQL\Path6\1\SALES_ACTIVE_PARTITION_8.NDF', SIZE = 100GB,
   FILEGROWTH = 0),
FILEGROUP SALES_DIMENSIONS
   (NAME = N'SALES_DIMENSIONS_1', FILENAME =
   N'F:\RAID5\DATA\11\SALES_DIMENSIONS_1.NDF', SIZE = 450GB,
   FILEGROWTH = 0),
   (NAME = N'SALES_DIMENSIONS_2', FILENAME =
   N'F:\RAID5\DATA\12\SALES_DIMENSIONS_2.NDF', SIZE = 450GB,
   FILEGROWTH = 0),
LOG ON
   (NAME = N'SALES_LOG', FILENAME = N'F:\SQL\Path8\SALES_LOG.LDF',
   SIZE = 120GB, FILEGROWTH = 0)
GO
CREATE PARTITION FUNCTION SALES_WEEK_PARTITION_FUNCTION (INT)
AS RANGE RIGHT FOR VALUES ( 129, 130, 131, 132, 133, 134, 135, 136 )
```

```
GO
CREATE PARTITION SCHEME SALES_WEEK_PARTITION_SCHEME AS PARTITION
   SALES_WEEK_PARTITION_FUNCTION
ALL TO (SALES_ActivePartition)
GO
```

Top of page