Microsoft SQL Server 2005

High Performance Data Warehouse with SQL Server 2005

SQL Server Technical Article

Writer: Erin Welker, Scalability Experts
Technical Reviewers: Eric Hanson, Microsoft Corp.
Lubor Kollar, Microsoft Corp.
Torsten Grabs, Microsoft Corp.

Published: October 2006
Applies To: SQL Server 2005

Summary: This document discusses things to consider when architecting a large, high-performance relational data warehouse, especially one that is host to unpredictable ad hoc queries. The discussion includes some of the new features of SQL Server 2005 and considerations to take into account when using these features. It also includes methodologies for creating and storing pre-aggregated result sets to facilitate mainstream queries and reports.

Copyright

Table of Contents

Introduction

We have visited a number of customer environments where a data mart has been configured on Microsoft® SQL Server™ to allow unfettered access to the data. Some of the "super users" that have access to this data have learned to write expert Transact-SQL. The availability of Report Builder in SQL Server 2005 Reporting Services (SSRS) extends powerful Transact-SQL query creation capability to even more users. Their ability to consume server resources is unsurpassed, making it a challenge for database administrators (DBAs) to guarantee consistent performance. Yet, when SQL Server Analysis Services (SSAS) and cube building is mentioned, users who need to access the data in unpredictable ways may perceive their query flexibility to be thwarted. So, how do you give your business users the access they demand, as well as give them well-performing queries and still meet availability Service Level Agreements (SLAs)?

A great deal of information surrounding data warehousing and Business Intelligence (BI) has been published over the past several years. Most of this information focuses on the process of building a data warehouse or data mart to meet and exceed business goals related to making strategic decisions. While this aspect of building the data warehouse is critical to the success of any BI undertaking, far less has been written about improving the performance and management of a SQL Server relational data warehouse. With the release of SQL Server 2005, additional features support the implementation of relational data warehouses in the multi-terabyte range. This white paper addresses some of the decisions involved and recommended best practices when architecting a relational data warehouse or data mart on SQL Server 2005 from a performance perspective.

Though this white paper discusses the obvious topics around hardware selection, SQL Server instance and database configuration, and table/index design, it also introduces overall strategies for delivering a data mart or data warehouse that balances query performance with the time investment for providing such an environment in terms of index and aggregation maintenance. We incorporate many of the features of SQL Server 2005 in this discussion, while challenging some frequently assumed approaches regarding the relational design of the tables, indexes, and constraints.

Performance of a relational data warehouse is usually perceived in two ways: updating the database and querying. Updating requires more than the extraction, transformation, and loading (ETL) of new data. Database administrators are responsible for designing and maintaining databases as efficiently as possible and those processes need to be worked into the batch update window. Querying can also come in various forms, such as end-user ad hoc queries, Reporting Services and other query tools, and loading into Analysis Services or other OLAP cubes. Since Analysis Services is very predictable in the types of queries it issues, this paper focuses on direct queries against the relational data warehouse. These could be ad hoc in nature via Report Builder or other direct query tools, or more predictable queries that result from pre-coded reports.

Before continuing, we will establish some assumptions on the terms already mentioned. The *relational data warehouse* refers to a store of information, usually very large, that is housed in a relational database management system for purposes of reporting, analysis and data mining. The terms *data warehouse* and *data mart* are often used interchangeably. The data warehouse, however, more often refers to the storage of information across all of the enterprise. This data is cleansed and often stored in a

normalized manner and seldom queried directly. Instead, it is used as a consistent source for one or more subject-oriented data marts that service users directly or feed OLAP cubes. The information in a data mart could be designed in any manner but usually is a dimensional design following the Kimball star or snowflake design methods[1]. "Data marts" are often lumped into the term "data warehouse" in the industry, however. For consistency sake, we use the term "data warehouse" as a generic term to include data marts throughout this white paper.

This paper assumes basic knowledge of data warehousing concepts, dimensional modeling, SQL Server 2005 table partitioning, and indexed views. Many of the comparative query tests were performed on the Project REAL database. Project REAL is a reference implementation of an at-scale, real-world data warehouse on SQL Server 2005. For supplemental information on these topics, see the References section at the end of this white paper.

Creating a Solid Foundation

The key to creating a well-performing relational data warehouse is to understand the data and how users query the database. The latter can change substantially over time, so it is necessary to continually make adjustments based on changes in query patterns. Prior to this, there are several infrastructure decisions to be made that can greatly affect the performance of the system after it is implemented.

Hardware considerations

Hardware decisions often are made prematurely, before any significant knowledge of the system is known. This is particularly tricky when choosing hardware for a relational data warehouse that allows unbarred ad hoc access to its end users. It has become increasingly common for "super users" to have access to the warehouse to satisfy their reporting needs, as Information Technology departments are hard-pressed to keep up with the demand for new reporting and analysis requirements. The hardware platform needs to allow for this, to the extent possible.

The 64-bit platform provides many advantages to a data warehouses due to the extended memory architecture. A data warehouse that supports ad hoc querying requires sufficient memory to read gigabytes (GBs) and terabytes of data. The cost-effectiveness of the x64 platform has all but eliminated the consideration of 32-bit in server purchasing decisions. This is particularly true for a data warehouse because it stores and queries very large databases.

Even a 64-bit decision is complicated by the two high-level chip architectures—x64 and IA64 (Itanium). There are a few things to take into consideration when making this decision. x64 is fully supported in SQL Server 2005 and is an excellent option for data warehouses due to its cost efficiency. Currently, Itanium-based hardware provides more scalability but the x64 platform is quickly catching up. It is important to remember that Itanium is a completely different chip architecture that favors parallelism and improved computational logic, often found in data warehouses. Since both chip architectures are quickly evolving, it is best to study the current vendor offerings. A good indicator can be the TPC-H results posted on the TPC Web site

[1] For more information on dimensional modeling, see the books referenced in the References section of this whitepaper.

(http://www.tpc.org/tpch/results/tpch_perf_results.asp). This is particularly informative when looking at the underlying database size, though that is not a pure indication of system size.

In addition to chip architecture, the number and speed of processors is important. Data warehouses require much number crunching and the need for parallelism, which is facilitated by both the number of processors and their clock speed. It can be difficult to estimate the actual requirements prior to implementing the data warehouse in production. A good approach is to not only establish a baseline to start with, but also to identify hardware and an architecture that can be scaled to more, faster processors in the future.

Data warehousing requires a great amount of memory. For medium to large data warehouses, make sure that the host SQL Server instance does not need to compete with other processes for memory. All queried data must go through memory. For instance, if a user designs a query that reads through all of a 500-GB fact table, every data page in that fact table or index will ultimately be read into memory. The use of summary tables, as mentioned later in this paper, greatly minimizes the amount of data that flows through memory. However, it is almost impossible to design summary tables to satisfy every query that the data warehouse will see, even in a given day. Specify as much memory as possible for the data warehouse server—it will never be enough.

Disk configuration

The disk configuration for a data warehouse is arguably the most critical factor. The very large memory configurations currently available can result in far fewer I/O operations. However, data warehouse databases can easily fall into the range of many terabytes. If all of this information is even infrequently queried, I/O will be a big factor in your environment as even very large memory will not prevent I/O. I/O still remains the slowest aspect of the hardware system and careful planning is required to purchase and configure the disk subsystem appropriately. Storage Area Network (SAN) continues to be the ideal choice for large databases, such as data warehouses. The details regarding SAN setup is best left to the SAN vendor and is outside of the scope of this discussion. Be sure to relay the characteristics of the different types of files (**tempdb**, data files, and log files) so that the disk subsystem will be designed appropriately.

Some basic best practices for disk configuration include:

- Create more SAN disk groups to support multiple and parallel I/O from SQL Server. Each disk group is made up of distinctly different disk arrays that are configured based on the type of data that is hosted (active data, historical data, logs, **tempdb**).
- Configure host bus adapter (HBA) to disk ratios to avoid HBA bottlenecks. Consult the SAN vendor for recommendations based on I/O ratings for each.
- Place the data and log files on separate disk groups for isolation to ensure recovery in the event of failure on either part. This also enables you to configure disk groups to be customized to the read/write characteristics of each.
- Place the **tempdb** data files and log file on separate disk groups.
- Ensure that arrays are built from a large number of physical disks, while not saturating the controllers.

- Stripe large tables that typically experience large range reads, such as fact tables, across a large number of disks to evenly distribute I/O (see Partitioning for Fast Loads and Query Performance later in this paper).

When defining RAID requirements, many of the best practices of OLTP follow into the data warehousing environment. **Tempdb** can be a critical database that should be segregated from all other database files on its own RAID array. **Tempdb** I/O characteristics are random read/write I/O, which benefits from RAID striping, preferably RAID 1 or 10. The transaction log I/O characteristics are sequential write I/O and should be placed on either RAID 1 or RAID 10 arrays. Data files should usually be configured to favor read I/O unless there are frequent updates throughout the day. If that is the case, the I/O priority should be weighed based on business requirements. Though RAID 5 is generally discouraged on OLTP, it is a relatively good option for data warehouse data files where write operations are infrequent (once a day or less). The huge disk requirements of a data warehouse often minimize or eliminate the option of RAID 1 or 10 for data files due to the high cost of such redundancy.

When architecting your data warehouse, think creatively about your data, especially if you come from an OLTP background. Remember that, even though the database is huge, it is extremely rare that any but a small portion is regularly updated. Use this to your advantage in database maintenance.

Look at the user's query requirements closely to determine innovative architectures. For instance, a point in time can usually be identified to segregate active data from inactive data. In this case, "active" refers not only to data that is regularly updated but is also frequently queried. Business requirements often require that data be stored for years, but you can usually use an 80/20 rule to determine a cutoff where only 20% of the data (or less) is read in 80% (or more) of the queries. Use this to isolate older, less active data, to less expensive disks and maybe even a separate server. The latter will isolate queries that reference all data since the beginning of time and wipe out active data in cache.

Networking requirements can vary. Though queries can filter through a very large amount of data, the ultimate result set returned to the client is generally small. An exception is when a separate application is inserted between the relational data warehouse and the end user of the data. This application may be an OLAP server, such as Analysis Services, or a front-end analysis tool, such as Proclarity. Such applications tend to request large amounts of data from SQL Server, requiring a higher speed network connection between the two servers.

SQL Server configuration

Once appropriate hardware has been purchased, it is important to configure SQL Server 2005 for its fullest potential. SQL Server was designed to be self-tuning so, in many cases, the very best thing to do is to leave the default configuration values as they are out of the box. Some exceptions to that rule, as well as some informative items and suggestions on database layout, are discussed in this section.

Sharing a server that houses a data warehouse with any other application or database is typically unwise. The inconsistent resource utilization can make it difficult or impossible to provide consistent performance to application databases whose characteristics can and should be consistent (OLTP).

The first topic is in regard to memory configuration. In many cases, a 64-bit platform is the platform of choice due to the high demands on memory made by a relational data

warehouse. Remember that queries against a relational data warehouse are varied and usually span numerous rows in the database. All database pages on which these rows reside must be read into memory in order to satisfy the query. AWE (Address Windowing Extensions) can logically raise the memory bar on a 32-bit platform, but it introduces memory mapping overhead that can be avoided on a 64-bit platform. Also, only the data buffer benefits from AWE—other memory resident objects such as procedure cache, locking memory and workspace memory are still restricted by the 4-GB limitation of a 32-bit platform.

AWE is still relevant in a 64-bit scenario and is often recommended as a means of locking SQL Server memory to prevent Microsoft Windows® from swapping database pages to disk. This is only a consideration when SQL Server shares the server resources with another memory-heavy application, such as another SQL Server instance or other SQL Server components (SQL Server 2005 Integration Services (SSIS), SSAS, or SSRS). It is recommended that a SQL Server relational data warehouse of any significant size (300 GB or greater) reside on its own server. It is not necessary for SQL Server to lock pages in memory in this scenario.

The **Max Degree of Parallelism** option tells SQL Server the maximum degree of parallelism that will be considered for a single query execution. The default of 0 tells SQL Server to determine this at run time. When a parallel plan is generated for a query, the query optimizer bases this decision on the current processor availability at run time. If there are several users on the server at a given time, it may be a good idea to throttle this number back so that a single query does not monopolize all of the processor resources, making them unavailable for queries that are subsequently executed. This can also be overridden for an individual query with the MAXDOP query hint. In SQL Server 2005, MAXDOP can also be used on index creation and rebuilds.

Cost threshold for parallelism is a server instance option that tells the optimizer when to begin consideration of a parallel plan. The optimizer has to balance the cost of plan generation with execution cost. If a query will run in less than a second with a good plan, it doesn't make sense to spend 4 seconds finding the very best plan. By default, parallel plans are not considered by the optimizer unless the best serial plan exceeds 5 seconds. If you consistently find that query plans are not parallel plans and you think they should be, this may be the reason. Remember that, though parallel plans may result in a faster query, they are usually more resource-intensive due to the overhead cost of bringing parallel thread results back together. Reducing the value for this option may benefit individual queries slightly, but at a cost at the system level.

An often overlooked SQL Server option is the query governor cost limit, which denies execution of queries that have a cost greater than the current cost limit. The default is 0, which means that all queries will run, regardless of their estimated execution time. Resetting this option is usually not valid in a data warehouse environment where large queries are assumed. It can be used to govern queries during specified times, or for "express lane" SQL Server instances that are reserved for less-resource intensive queries. For instance, you may have a SQL Server instance that is used for querying more active data and a separate instance for queries that may span all data. The query governor is an advanced option but does not require a SQL Server restart.

CPU and I/O affinity can also be specified for a SQL Server instance. Unless the instance shares server resources with other applications on the server, they should not be used. This configuration is strongly discouraged in all but the smallest data warehouses.

Tempdb

The **tempdb** database can be an extremely critical component in the relational data warehouse. Data warehouse queries tend to perform the grouping, ordering, and aggregating of huge numbers of rows. Depending on memory resources, indexing, and query structure, great demands can be placed on **tempdb**. **Tempdb** should be tuned to perform as efficiently as possible. Formalized reporting should be scheduled outside of the ad hoc query window to the greatest extent possible to avoid contention for this shared resource.

The first step for configuring **tempdb** is to place it on the most efficient disk available. This should be a striped RAID configuration with no parity (no RAID 5, which introduces a write overhead). Note that disk recovery is not an issue with **tempdb** since it is reinitialized every time SQL Server starts. The underlying disk array should have numerous spindles and, ideally, multiple I/O paths. Also be aware of other operations that place a high demand on **tempdb**, such as online indexing and row versioning. In general, these features are not used in a relational data warehouse but it is good to be aware of their effects on resources if they are being considered.

- Make as many **tempdb** files as you have physical CPUs[2], accounting for any affinity mask settings. Do not factor in hyperthreading or dual cored CPUs into the count.
- Make the file sizes of equal amounts (be sure they are 'in total,' big enough to handle anything they may encounter) and that they are on your best and fastest drives.
- The **tempdb** files should be isolated to their own disk groups and can, therefore, be sized to the capacity of that disk group. Auto grow is not recommended except as a safety net. Even if the conditions exist for instant file initialization, operations that use **tempdb** must pause while the file auto grows.

Determining an adequate disk capacity for **tempdb** can be particularly challenging in an ad hoc query environment. If business users directly query the database, training might be useful in order to promote query best practices that have less impact on **tempdb**. One method is to collect **tempdb** usage information from a QA or test environment, then extrapolate based on the expected usage growth in the production environment. For great information on how SQL Server 2005 uses **tempdb**, as well as scripts and DMVs (dynamic management views) for monitoring **tempdb** usage, see the Working with tempdb in SQL Server 2005 white paper (http://download.microsoft.com/download/4/f/8/4f8f2dc9-a9a7-4b68-98cb-163482c95e0b/WorkingWithTempDB.doc).

Database configuration

Locking

Database locking is performed entirely for the sake of data consistency, ensuring that data updates are atomic, consistent, isolated, and durable. The concept of data consistency only applies in an environment where updates occur, but SQL Server has no knowledge of a read-only environment unless that is explicitly specified. As such, all queries result in locking behavior. In a data warehouse environment where thousands,

[2] These files do not need to spread across multiple disk arrays unless you are experiencing an I/O bottleneck in **tempdb**.

millions, billions, or more rows are read in a single query, locking overhead can be tremendous. Toggling the database read-only option off and on during the batch process can provide performance improvements across the entire query environment. Note that changing this database option requires exclusive access to the database. The database can be made to be read-only with the following code:

```
USE master;
-- Change the database to single user mode - rollback any
-- transactions and disconnect all users in the database
ALTER DATABASE [DataWarehouse]
SET SINGLE_USER
WITH ROLLBACK IMMEDIATE;

-- Change the database to read-only
ALTER DATABASE [DataWarehouse]
SET READ_ONLY;

-- Change the database back to multi-user mode
ALTER DATABASE [DataWarehouse]
SET MULTI_USER;
```

This has the same result as specifying the NOLOCK query hint on a query by query basis. NOLOCK usually cannot be specified on queries that are issued from a reporting or analysis tool and requires knowledge of this hint by users that directly access the database. Therefore, if possible, it is recommended that the data warehouse database be made read-only outside of the batch update cycle.

To realize the impact of a read-only database on performance, review the following query times for this sample query against the Project REAL full-sized database:

```
SELECT d.Calendar_Month_Desc, i.Dept, s.District, i.Category,
    SUM(On_Hand_Qty)
FROM dbo.Tbl_Fact_Store_Inventory inv
FULL OUTER JOIN Tbl_Fact_Store_Sales sales
    ON Sales.SK_Date_ID = inv.SK_Date_ID
    AND Sales.SK_Item_ID = inv.SK_Item_ID
    AND Sales.SK_Store_ID = inv.SK_Store_ID
JOIN Tbl_Dim_Store s on
    inv.SK_Store_ID = s.SK_Store_ID
JOIN Tbl_Dim_Item i on
    inv.SK_Item_ID = i.SK_Item_ID
```

```
JOIN Tbl_Dim_Date d on
    inv.SK_Date_ID = d.SK_Date_ID
WHERE d.Calendar_Year_ID = 2004
AND I.Dept = 'Hardcover'
AND S.District = 'Washington'
AND I.Category = 'Art'
AND On_Hand_Qty <> 0
GROUP BY d.Calendar_Month_Desc, i.Dept, s.District, i.Category
```

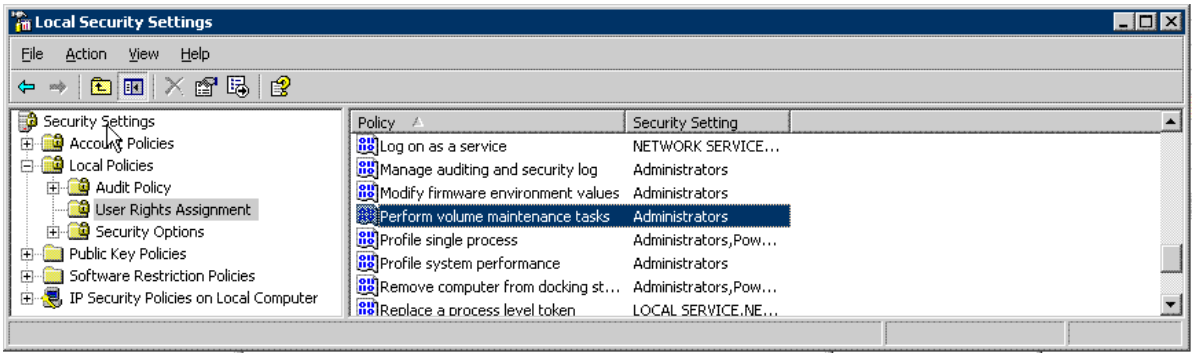
Database READ_ONLY	Avg. Response Time (ms)
False	1042
True	776

That's an improvement of over 25%. Both query runs were preceded by DBCC FREEPROCCACHE and DBCC DROPCLEANBUFFERS. Note that this minimal locking behavior is not observed simply by placing objects on read-only filegroups. The read-only filegroup option is simply to facilitate the recovery of an unchanged filegroup so as not to require the transaction log.

Though the above performance enhancements can also be observed by using the NOLOCK hint in the query, setting this at the database level will benefit queries that are issued from other applications, such as Report Builder. It also guarantees a stable database state during the course of the day in environments that perform batch updates on a daily or less frequent schedule.

Instant file initialization

Again, we will not review all database options, only those that are known to significantly affect performance in a relational data warehouse. SQL Server 2005, in conjunction with Windows 2003, is able to rapidly create or increase database sizes through instant file initialization. To take advantage of this feature, verify that the SQL Server service account is added to the Perform Volume Maintenance Tasks security policy. If the service account is not already a member of a group with this privilege, use the Local Security Settings option under Administrative tools to add it. Select **Security Settings**, **Local Policies**, and then **User Rights Assignment**. Add the SQL Server account or group to **Perform volume maintenance tasks**, as shown in the following figure.



Assigning Perform Volume Maintenance Tasks rights

This will greatly reduce the time it takes to create the very large databases associated with data warehouses. Note that instant file initialization only applies to data files, not log files; the database creation time will be constrained to the time it takes to create the associated transaction log files.

Instant file initialization applies to the auto growing of database files, as well. Yet, it still remains a best practice to proactively size database and log files to their anticipated size in the foreseeable future. Even though auto grow is much faster with this new capability, database activity is at a stand still until the growth operation completes. Though sizing should be proactive, leave the auto grow capability on the files as a safety net, but increase the default growth rate from the default 1 MB to prevent repetitively growing the database in very small increments. This also minimizes fragmentation.

As mentioned, transaction log files cannot take advantage of instant file initialization. It is recommended that the transaction log file be pre-allocated to avoid auto grow. An update that has to wait for the transaction log file to grow can greatly slow down an ETL process. Also, make sure that auto grow increments (which should be left on as a safety net) are large enough to prevent many small log segments which can result in disk fragmentation.

Auto Shrink

It is generally a best practice to leave Auto Shrink turned off. If new business requirements reduce the volume of data that needs to be kept (this is rare), the database can be shrunk manually. Turning off Auto Shrink prevents databases from shrinking after data is archived only to grow again once new data is added.

Auto update statistics

Auto update statistics presented an issue in previous versions of SQL Server because the first query to compile with outdated statistics waited for new statistics to be created before it could be compiled and run. The batch updates in a data warehouse seldom trigger out-of-date statistics. However, when they do, users who were the first to run queries after the nightly update were burdened with a query lag as statistics were updated. In SQL Server 2005, statistics can be updated asynchronously and the query that triggered them compiles and runs with the old statistics in the meantime. This means the query compiles with old statistics, which are usually good enough. It is recommended that Auto Update Statistics be left on, as is the default. You may wish to enable asynchronous statistics update using the following command if statistics update overhead as part of a user query causes an undesired delay.

```
ALTER DATABASE database_name SET AUTO_UPDATE_STATISTICS_ASYNC ON
```

Disk layout

A generally recognized best practice for laying out tables on disk is to use the primary filegroup only for system objects and use explicitly defined filegroups for remaining items. Dimension tables can go on their own filegroup, which resides on its own disk array. This facilitates joins with fact and summary tables by distributing I/O. Fact tables should go on their own filegroup, which is striped across multiple files, slightly fewer than the number of available physical processors (accounting for CPU affinity). If

partitioning is implemented, see the section on Partition disk allocation later in this document.

It is important to understand your SAN environment to know how LUNs are laid out. Work with your SAN vendor and your storage administration group to design the best disk subsystem for your data warehouse. Make sure they are aware of the emphasis toward large, sequential read I/O that characterize data warehouse I/O.

Table Design

There are generally two approaches to the physical design of a relational data warehouse where data is consumed directly by end users. The first is to retain the third normal form design of the source data. This is usually the approach taken when trying to minimize the time it takes to deliver data to the reporting user (near real-time). Such a design is good for operational reports that are more detailed in nature. In the case of a third-party source system, such a database can be used to feed application reports that are segregated from high transaction-rate operational systems.

The second approach is a dimensional design, usually referred to as a *star* or *snowflake* schema. The primary benefits of this approach are simplicity and performance. The simplicity of the model makes it easy and quick for end users to grasp and browse. The performance of a dimensional design over a relational design can best be observed in actual tests that you can even perform on your own (**AdventureWorks** and **AdventureWorksDW** can be used for this purpose). Knowledge of what constitutes a star or snowflake design is assumed and, therefore, out of the scope of this white paper. Due to the overwhelming arguments for using a dimensional design for the relational data warehouse, this table design approach is assumed in the remainder of this document.

The next design question has to do with whether to implement a star schema or a snowflake schema design. This has been a controversial debate in the past, but snowflake is getting more and more favor. Traditional arguments against snowflaking are data model complexity and performance drawbacks.

In a SQL Server environment, performance of a star over a snowflake design is rarely a big issue. Even on larger dimensions, such as a customer dimension, the narrower tables that result from a snowflake design usually compensate for the additional JOINS. This can, once again, be observed in testing scenarios in your own environment. The complexity that is introduced with multiple tables in a snowflaked dimension can be hidden from end users by implementing views that will make the design appear to be a star schema. Ultimately, the argument between the two designs is not so compelling as to rule out one over the other in all cases. The primary message is that performance is not a reason for preferring one over the other.

Declarative referential integrity and constraints

In the data warehouse, a fine balance between data integrity and performance needs to be maintained. Though data integrity is of the utmost importance, updates to the data warehouse are typically finely controlled through the periodic batch ETL process, often performed daily or hourly. The star or snowflake schema model strictly implies that the low-level dimension tables contain a primary key that is a surrogate key. The source system does not have this surrogate key information, so it must be obtained in the ETL by looking it up based on the business key. Defining declarative referential integrity greatly slows down the performance of the ETL code and is redundant in this case.

Again, data integrity takes priority, but if updates are as finely controlled as they should be, it is recommended that referential integrity in the form of a declared foreign key/primary key relationship between fact tables and dimension tables be omitted.

The same is true of constraints on the fact tables. It may be reasonable to define constraints on dimension tables since the rate of update is usually far less than on fact tables. As with PK/FK defined integrity, it should be the job of the ETL process to insure that all data that is loaded into the data warehouse is clean.

Note that referential integrity can be defined in data source views (DSVs), allowing Analysis Services and Report Builder to leverage these inherent relationships and formulate appropriate queries back to SQL Server.

Indexing strategies

Appropriate indexing is extremely important in a SQL Server relational data warehouse. The first tendency is to create as many indexes as possible to facilitate the dynamic queries that may be directed at the fact tables. It is important to carefully plan the indexing strategy to use, to fully understand the data that is represented in the warehouse, how SQL Server chooses helpful indexes, and the nature of the queries that will be issued against the warehouse. The last item is very difficult to predict. The business user interviews that typically take place at the beginning of a data warehouse venture will usually help to distinguish which items (dimension and measures) will be in most high-demand when the data is made available. The delivery of even a subset of the data to key business users early in the development phase is helpful in both validating the design and inclusion of information, and illustrating the nature of the queries.

Dimension table indexing

Dimension table indexing is relatively straightforward. Even if the strategy is off the mark, inefficient indexes don't have a high overhead since dimension tables are usually small and relatively stable. The primary exception is very large dimensions such a customer dimension and Type 2 slowly changing dimensions. A general best practice is to create a clustered, primary key on the surrogate key of each dimension table. The surrogate key is usually an IDENTITY column, which also facilitates INSERTs. A nonclustered index on the business key should be considered for query purposes, or if surrogate key lookups are performed through a Transact-SQL statement during ETL. When using SSIS to perform the lookups, the table or a subset of the table will be loaded into memory on the SSIS server and an index on the business key is usually not helpful. Note that a non-clustered index on the business key for Transact-SQL lookups will not require a lookup of the actual data page since the clustered index key, the surrogate key in this case, is replicated in all non-clustered indexes.

Fact table indexing

Just because an index exists, doesn't mean SQL Server will use it. The creation of indexes that SQL Server will seldom use is just overhead for the population and management processes, not to mention disk resources. One commonly prescribed strategy for indexing fact tables is to create a primary key on a fabricated IDENTITY column and create a nonclustered index on each of the foreign keys to the dimension tables. Another strategy is to create a clustered, composite index composed of each of the foreign keys to the fact tables. The first caution is to consider all aspects of your

environment and not blindly follow prescriptive guidance. Know your data, your users, and the SQL Server optimizer. This information will greatly facilitate designing the most efficient indexing strategy. It is often the case that neither of these strategies, alone, are fully effective in delivering improved performance.

One consideration on index creation is to keep them as tight as possible. Compact indexes require fewer pages, which helps performance, especially on enormous fact tables. Remember that the clustering key is replicated on each nonclustered index leaf page, so a smaller clustered key results in smaller nonclustered indexes.

A clustered index with the most commonly queried date column as the leftmost column in the index is almost always a good idea in a data warehouse. If the fact table is partitioned, this will usually be the same column as the partitioning key. Date is the most commonly queried column in most data warehouses. Note that there can be more than one date in the fact table (sale date, ship date, etc.) but there is usually one date that is of the most interest to business users. A clustered index on this column has the effect of quickly segmenting the amount of data that must be evaluated for a given query. You might also add additional dimension foreign keys to this index to create a composite index. Making the clustered index a composite index facilitates a star join plan called a cross product plan. In a cross product plan, a cross product of the row sets from different dimensions is formed, and the resulting keys consisting of two or more dimension keys composed together are used to probe the composite clustered index in the fact table. It is a good idea to order the foreign key columns left to right by the dimensions that will be used most often. If there will be no nonclustered indexes, the increased cluster key size of this composite index is much less of an issue.

Other nonclustered indexes should be evaluated very carefully before you create them. Even dimension keys with a high cardinality (large number of distinct values) often correlate to many rows in a fact table. The optimizer will often choose to use a clustered index scan approach instead of using a nonclustered index to avoid numerous lookups back to the data pages. When in doubt, create the nonclustered indexes on a full data set, then check the query execution plans. You may be surprised by the seemingly useful indexes that the optimizer sees little value in.

An exception to this is when there are frequent queries that do not filter on the first column of the clustered key in the fact table. An example of this might be when multiple dates are used for analysis (order date, shipping date, received date, etc.). If this is the case, it can be beneficial to create separate non-clustered indexes on each of the most commonly used dimension keys of the fact table. These can be used by the SQL Server query optimizer for index intersection plans. In an index intersection plan, sets of row IDs from two or more dimension key indexes on the fact table are intersected. The resulting row IDs are used to seek into the fact table.

Using a Smart Date key

A useful technique is to use the **smalldatetime**[3] data type as the surrogate keys for the Date dimension. This allows you to specify date range filters directly against the

[3] **smalldatetime** is recommended over **datetime**, where possible, since it is more compact (4 bytes versus 8 bytes). The date range that **smalldatetime** covers a smaller date range, from January 1, 1900 through June 6, 2079, which may prevent its use. It is also less granular, as it is rounded to the minute, which is usually not an issue in a data warehouse.

fact table in a readable way. Specifying range predicates directly on the partitioning key of the fact table allows SQL Server 2005 to eliminate partitions for queries that must scan the fact table. Specifying date range filters on the fact table via a join with the Date dimension can cause the entire fact table to be scanned in some cases, even when only a small number of partitions lie in the date range. For example, a query roughly based on the Project REAL schema with a date range on the Fact table might look like this (note the date criteria in bold):

```
SELECT Subject, SUM(Sales.Sales_Qty) AS Sales_Qty
FROM Tbl_Fact_Store_Sales as Sales
JOIN Tbl_Dim_Store Store
  ON Sales.SK_Store_ID = Store.SK_Store_ID
JOIN Tbl_Dim_Item Item
  ON Sales.SK_Item_ID = Item.SK_Item_ID
WHERE Sales.Transact_Date BETWEEN '01/01/2004' AND '03/31/2004'
AND Store.Region = 'West'
GROUP BY Subject
ORDER BY Sales_Qty DESC
```

For selective queries that seek into the fact table's clustered index, with Transact_Date as the leading key, partition elimination is performed naturally even via joins between the fact table and the Date dimension.

Evaluating index utilization

A useful DMV in SQL Server 2005 is **sys.dm_db_index_usage_stats**, which records the number of times indexes were used in the current SQL Server run (since last restart). Any indexes that are not recorded in this DMV have never been used since SQL Server last started[4].

For some useful scripts that can help you identify frequently used indexes, indexes that have not been used, and comparison of index usage to cost of management, see the Customer Advisory Team blog on "How can SQL Server 2005 help me evaluate and manage indexes?" (http://blogs.msdn.com/sqlcat/archive/2006/02/13/531339.aspx). These scripts are good for evaluating the usefulness of your indexing strategy once it has been implemented.

Index fragmentation

One performance aspect of indexes that can be overlooked in a data warehouse environment is index fragmentation. We've found customers who didn't perform index maintenance because it was perceived that there just wasn't enough of a batch window to support it! A fragmented index can have an enormous negative impact on performance. The trick is to minimize the index maintenance by only creating useful

[4] Note that there is a limit of 500 indexes that will be captured in the **sys.dm_db_index_usage_stats** DMV. If there are 500 rows in this DMV, results could be inaccurate since some relevant rows/indexes could have been evicted.

indexes, and reorganizing only the indexes that are fragmented. The first point has already been addressed.

The second point can be addressed by not assuming that all indexes must be reorganized all the time. Small- to mid-size dimension clustered indexes are slow to become fragmented because they are usually inserted in order by a sequential identity column. Nonclustered dimension indexes are somewhat more problematic but also less expensive to reorganize. Large fact tables are frequently partitioned. All index operations are partition-aware, including those that detect fragmentation and re-organize the index (ALTER INDEX with REBUILD or REORGANIZE). ETL programs can even monitor updated tables/partitions to identify those that need to be evaluated for fragmentation. A future white paper will discuss more tips on building index maintenance into the data warehouse environment so that it is minimally intrusive.

Partitioning for fast loads and query performance

Table partitioning in a data warehouse is a common practice, primarily to facilitate the management of very large fact tables. Knowledge of the general concepts of partitioning is assumed in this white paper. Papers that address these concepts can be found elsewhere (note the references at the end of this paper). Likewise, we will focus on the new table and index partitioning feature in SQL Server 2005, though some of the concepts can be implemented through local partitioned views which are available in SQL Server 7.0 and 2000.

For this discussion, we assume that horizontal partitioning based on a date that is roughly parallel to the actual progression of time is generally a good selection for a data warehouse. For instance, the date of sale would be a good selection in a retail data warehouse that is focused on sales. The idea is to select a partitioning key that is the basis of the majority of queries against the data warehouse and would also be used when determining the age of the fact records. Though dimension tables can be partitioned, they are usually too small to consider. This discussion focuses on the partitioning of fact tables and the impact this has on loading the data warehouse and the queries that reference them.

We'll readdress the most obvious benefits of partitioning so we can focus the remaining discussion on how to best leverage those benefits:

- Database Maintenance—the time needed for common database maintenance operations (such as backups, restores, index maintenance) can be greatly minimized when fact tables are divided on a column that segregates frequently updated data from read-only data.
- Efficient loading—partitions can be loaded outside of the partitioned table, thereby minimizing the impact on active queries.
- Partition elimination—the query optimizer can eliminate large portions of the fact table if a query filters on the partitioning key.
- Data archival—the removal of old data from a partitioned table is greatly facilitated.

Partitioning considerations

There are a few factors to take into consideration prior to establishing a partitioning strategy. First, know the business requirements around data archival. Don't partition at a boundary beyond that where you will be removing data. For instance, if requirements are to remove data a month at a time, it is not a good idea to partition by year since a

DELETE statement will be required to remove the data. Partitioning by month, in this case, means that you can simply SWITCH out the month(s) to remove from the partitioned table.

Query parallelism on a partitioned table is at the partition level. The only exception is if the query optimizer is able to eliminate all but one partition to satisfy the query. In this case, parallelism can be implemented up to the specified maximum degree of parallelism. This can influence the partitioning strategy in an environment where many of the queries are focused on very recent data.

To elaborate, consider a Sales data warehouse scenario where a large number of queries analyze this month's sales compared to last month's and the table is partitioned by month. Queries of this nature result in a maximum of two worker threads—one for this month and one for last month. On a high-end server with numerous processors, partitioning by month in this query environment may not be a good idea. One option is to rewrite queries that only reference two or three partitions to use a UNION ALL statement:

```
SELECT columns FROM partitioned_table … WHERE partition1
UNION ALL
SELECT columns FROM partitioned_table … WHERE partition2
```

It is also possible to change the partitioning unit throughout the data life cycle through the MERGE and SPLIT verbs. In the aforementioned scenario, a table that is typically partitioned by month could be modified for the current and previous month to be partitioned by week until each partition ages past two months. Queries against the current and previous month would then result in a maximum degree of parallelism of eight. This option should be cautiously considered before implementing, due to the expense associated with ultimately merging the weekly partitions into monthly partitions. These merge operations may need to be performed during an extended batch window, such as on a weekend. Furthermore, in a multi-user environment where multiple parallelizable queries are frequently utilizing available resources anyway, queries might not be impacted by changing the partitioning strategy. A best practice in this case would be to implement a change in the partitioning unit after this is noted to be an issue in your environment.

With the information addressed thus far, let's consider some options for a partitioning strategy. Query patterns can vary tremendously from one environment to another so the following options should be only be used as input to identify a suitable strategy for your environment.

Partition loading

You will usually find it more efficient to load fact data outside of the partitioned table. If the fact table is large enough to warrant partitioning, this usually means that incremental updates are faster if performed external to the partitioned table. Since performance varies across environments, be sure to validate this before implementing. This can be done by switching the current partition out to an external table or by utilizing a partitioning strategy where new data results in a new partition (daily load = daily partition). Late-arriving facts can usually be handled through the partitioned table but, again, performance will vary based on your environment and the average number of late-arriving facts.

Co-aligned partitions

It is generally a good idea to come up with a consistent partitioning strategy across all partitioned tables in your data warehouse. If there is any potential to join these tables, the SQL Server optimizer can consider a plan whereby parallel threads join the tables within each partition and then combine the results. This is mostly relevant with joined fact tables, such as a Sales and Inventory data warehouse where reporting often compares sales to in-stock inventory.

Again, there is not prescriptive guidance in this document since query patterns, resource availability, and business requirements are so varied across customer environments. The goal of this discussion is to give information on partitioned table behavior and to provide options for consideration.

Partition disk allocation

Once a partitioning strategy has been established, you need to determine how the partitions will lay out on disk. There are two high-level approaches—map multiple partitions to one filegroup, or map individual partitions to their own filegroup. Variations to each approach are also discussed.

The biggest issue with the one-filegroup strategy is that all partition data will be spread out across the same files on disk. If there are multiple files in the filegroup, SQL Server uses a proportional fill strategy to insert data. This noncontiguous data means that SQL Server sequential scans may be less efficient. This strategy also limits the flexibility in partition-based backups and piecemeal restores.

Mapping partitions to their own filegroup is usually the better strategy. Each filegroup should have a single file so there is the potential for contiguous data. Whether the data is actually contiguous depends on how it is loaded or if the clustered index has been recently defragmented. Using the single-file per filegroup per partition strategy, be sure that each file is striped across a large number of disks. Preferably, each file should be striped across all disks, or all disks in the storage tier if you have multiple tiers of disks of different speeds.

Optimizing the Data Warehouse Environment

Aggregation strategies

A common observation with customers that allow direct, ad hoc querying of SQL Server is the frequent use of temporary summary tables. This should not be a surprise—this is how Analysis Services gets its performance. The issue is that the users (usually technically advanced "super users") are often the ones doing the summarization and frequently they are summarizing the same information over and over again since they are unaware that other users are doing the same. This has a heavy impact on system resources. As previously mentioned, **tempdb** takes a very heavy hit on this type of query. Also, the pre-aggregations have to be stored somewhere, either in the source database or in **tempdb**. If this information is stored repeatedly, the disk consumption can be enormous. When these are built during peak hours, memory demands can be huge as all detailed data to support the summarization must be read into memory. This usually best reveals itself with a low page life expectancy (performance counter SQLServer:Buffer Manager\Page life expectancy).

Designing summary tables

A strategic and highly effective goal is to design and implement a handful of permanent summary tables to include pre-aggregated information. What often kills this type of initiative is the desire to create summary tables that satisfy 100% of user queries. This goal is almost certainly unobtainable and can result in very large summary tables that provide little benefit. The target should be closer to satisfying 75% or 80% of queries. This initiative cannot be started until a good representation of the queries is available. This is another reason for pushing data to the users as quickly as possible, as query patterns can be reviewed even in a testing or QA environment.

The following is an example process for designing summary tables:

1. Collect a good sampling of queries. These may come from user interviews, testing/QA queries, production queries, reports, or any other means that provide a good representation of expected production queries.
2. Analyze the dimension hierarchy levels, dimension attributes, and fact table measures that are required by each query or report.
3. Identify the row counts associated with each dimension level represented.
4. Balance the most commonly queried dimension levels against the number of rows in the resulting summary tables. A goal should be to design summary tables that are roughly 1/100th the size of the source fact tables in terms of rows (or less). Also, minimize the columns that are carried in the summary table in favor of joining back to the dimension table. The larger the summary table, the less performance advantages it provides.

Analysis Services

Those who are familiar with the workings of SQL Server Analysis Services (SSAS) may have already noticed that many of the recommendations in this paper come with SSAS by default. The performance advantages in SSAS come directly by its inherent ability to determine the best pre-aggregation algorithms and dynamically navigate through dimensional data. In most cases, it outperforms the relational data warehouse simply because this type of querying is what it is designed to do.

There may be inhibitors to moving to Analysis Services, perhaps because of the added expense of a separate server and the perceived time needed to process cubes. The latter, however, is very similar to the concept of updating summary tables with new information during or after ETL processing. Analysis Services is the preferred platform for reporting on aggregated data and trend analysis of data warehouse information. Following are a set of advantages and disadvantages for Analysis Services to help you decide whether it is a good option for your environment.

Advantages

- Many of the manual processes mentioned in this white paper, particularly regarding pre-aggregation, are built into Analysis Services.
- There is a semi-automated process that analyzes queries submitted to the server over time and allows the administrator to specify a modification to the aggregation design based on those queries (usage-based optimization).
- Several third-party applications are designed to read SSAS metadata and present a user-friendly ad hoc query interface.

- No explicit report model has to be designed; Report Builder can generate a model directly from SSAS cubes.
- Aggregation design is more flexible.
- Query performance for queries that aggregate along predefined dimension hierarchies tends to be much faster.

Disadvantages

- Programmatic access of SSAS data must utilize MDX (Multidimensional Extensions), a language that is far less known and less intuitive than Transact-SQL.
- There is little overlap in skill sets between SQL Server and Analysis Services, requiring existing personnel to develop the new skill set or the direct hiring of this skill set.
- Cube (measure group) and dimension processing can be less flexible than explicitly maintaining summary tables. SSAS may trigger a full measure group or dimension process even when it is not really required.
- Not all data may have a home in the cube.
- Queries that must touch each fact in a large fact table (because no aggregates are available to make them go faster) may run much slower on SSAS than the equivalent Transact-SQL query on SQL Server.

A process for designing summary tables

Designing summary tables is not an easy task and requires a fair amount of analysis of the expected or observed query load from reporting and ad hoc queries. If you don't have any of this information, you can go back to operational reports that are of the greatest value to the business users. Focus on those that they rely upon heavily since there will be many stale reports that are not useful input to this process. Keep in mind that you want to design as few summary tables as possible to satisfy the largest number of queries.

The following is an abbreviated process that was used for Project REAL. Using a handful of reports that were provided by the customer, we created a Microsoft Excel spreadsheet (a simplistic representation is in the table below) to map out the primary dimensions (Date, Store, and Item) and record the lowest dimension hierarchy levels that were queried by each report and the fact table measures that were referenced. The number of members at a given level was used to determine whether it was included or not. The first summary table was designed to be a multi-purpose summary table with no focus on a particular dimension. The bolded reports are those that would ultimately benefit from this summary table using this methodology.

Report	**Dimension Level**			**Measures**	
	Store	**Item**	**Date**	**Sales**	**Inventory**
Report 1	District		Calendar Year	Sale_Amt Sales_Qty	
Report 2	District	Category	Calendar Year Calendar Month	Sale_Amt Sales_Qty	
Report 3	District		Calendar Year	Sale_Amt	

			Calendar Month	Sales_Qty	
Report 4	**District**		**Fiscal Period**	**Sale_Amt**	
Report 5	Store	Dept	Fiscal Week	Sales_Qty	Model_Qty
Report 6		**Dept**	**Fiscal Period**	**Sale_Amt**	
Report 7	**District**		**Fiscal Week**	**Sale_Amt** **Sales_Qty**	
Report 8	**District**		**Fiscal Week**	**Sale_Amt** **Sales_Qty**	
Report 9	**District**	**Dept**	**Fiscal Quarter**	**Sale_Amt**	
Report 10	**District**		**Fiscal Period**	**Sales_Qty**	
Report 11	**Region**	**Category**	**Fiscal Week**		**Model_Qty**
Report 12	**District**		**Fiscal Week**	**Sales_Qty**	**On_Hand_Qty** **Days_In_Stock** **Model_Qty** **Return_Qty**
Report 13	**Region**		**Fiscal Week**	**Sales_Qty**	**On_Hand_Qty** **Model_Qty** **On_Order_Qty**
Report 14	**Region**		**Fiscal_Period**	**Sales_Qty**	**On_Hand_Qty**
Report 15		**Dept**	**Fiscal Week**	**Sales_Qty**	**On_Hand_Qty**
Report 16			**Fiscal Period**	**Sale_Amt** **Sales_Qty**	

Next, we reviewed the member row counts at specific dimension levels to understand the impact, based on potential rows in the summary table, of our summary table decisions

Dimension	**Level**	**# Populated of Members**
Store Geography	Division	1
	Region	3
	District	50
	Store	3980
Item Category	Subject	279
	Category	1987
	Department	4145
Date	Fiscal Year	3

	Fiscal Quarter	12
	Fiscal Period	36
	Fiscal Week	156

Remember that our target is to design a summary table that is roughly 1/100th of the base fact table or less. The Store Inventory fact table contains 8.5 billion rows and the Store Sales fact table about 1.5 billion. A summary table with 85 million rows or less is our goal. Realize that there will likely not be a row for every combination of the levels, but we prefer a conservative estimate and will assume a high density. We decided to summarize at the Store District (50), Item Category/Department (4145), and Fiscal Week (156) level. The Fiscal Week ending date is stored in the same integer format (CCYYMMDD) as the date surrogate key and is named "Date_Summary_Key" to provide the capability to include additional date rollups, if desired. The current maximum estimated rows in our summary table will be roughly equivalent to 50 x 4145 x 156, or 32.3 million. The following is the Transact-SQL code used to create the summary table:

```
CREATE TABLE [dbo].[Tbl_Fact_Summary](
    [Date_Summary_Key] [int] NOT NULL,
    [Store_District_Num] [int] NULL,
    [Item_Category_Code] [char](4) NULL,
    [Item_Dept_Num] [int] NULL,
    [Sales_Qty] [int] NULL,
    [Sales_Amt] [decimal](11, 2) NULL,
    [Inv_Model_Qty] [int] NULL,
    [Inv_On_Hand_Qty] [int] NULL,
    [Inv_On_Order_Qty] [int] NULL,
    [Inv_Return_Qty] [int] NULL,
    [Inv_Days_In_Stock] [int] NULL
)
```

Based on the report spreadsheet, two more focused summary tables are warranted, one with a focus on Store and one with a focus on Calendar Month. The goal is to aggregate at the highest level possible for the dimensions that are not in the focus. Remember that the higher the number of rows in the summary table, the slower it will be and it may not be worth the maintenance versus the cost of querying the fact table directly.

A few more specialized summary tables can be added. Look at possibly 5-10 summary tables per subject area (data mart). Summary tables are a lot like indexes in that it is easy to create so many that several will not be used, and they add to the cost of ETL. If users are directly querying summary tables, they will usually focus on just two or three that serve them well.

Indexed views vs. summary tables

It is very beneficial to automate the maintenance of all summary tables. This would be greatly facilitated by using indexed views, a SQL Server feature, as the means for pre-aggregating data. This seems to be a logical step due to the nature of indexed views. There are several requirements that an indexed view must adhere to, however, which can make it difficult to implement as a means of summarizing fact tables. Here are some of the requirements that are often relevant in this scenario:

- OUTER JOINS are not allowed. If more than one fact table is being summarized and row intersection is not guaranteed, an OUTER JOIN may be required. A good example is in a data warehouse to analyze sales versus inventory. Sales and the inventory fact tables may both contain rows that satisfy key combinations that are not in the other, necessitating a FULL OUTER JOIN.
- Indexed views are schema-bound to their source tables. A SWITCH operation on a partitioned fact table thereby requires that any indexed views based on that table be dropped and recreated.
- Indexed views require a unique, clustered index as the first index. This may require the addition of columns into the index that don't really need to be there. This is primarily an issue if there are additional indexes on the indexed view, since a very long clustered index key will be perpetuated in all nonclustered indexes.
- There are conditions that must exist in order for a query to automatically use an indexed view over the table specified in the query. This could require the creation of additional indexed views that are expensive to build and maintain (or that queries explicitly name the relevant indexed views combined with the NOEXPAND hint). One example of this is that all tables that are referenced in the indexed view must be referenced in the query. If the indexed view references Tbl_Fact_Store_Sales, Tbl_Dim_Store, Tbl_Dim_Date and Tbl_Dim_Item, but the query only references the first three tables, the optimizer will not consider using the indexed view to satisfy the query.

An obvious benefit of indexed views is that they are automatically maintained by SQL Server. This benefit is tremendous, so indexed views should definitely be considered if the aforementioned issues are not relevant to your environment. You can query the indexed views directly, using the NOEXPAND hint, to get the benefit of summary tables for query processing performance improvement, combined with the benefit of automatic summary maintenance.

As previously described, an alternative to indexed views are explicit summary tables. These tables must be manually maintained and referenced directly by queries. The former issue is a big one, but is no less of an issue than the fact that they are already manually maintained by users currently. Queries against summary tables can be orders of magnitude faster than queries against the detail data. This should encourage users in SQL Server to favor these tables over the enormous detailed tables. The summary tables can be incorporated into report models to satisfy users of Report Builder. This topic is addressed later in this document.

To quickly demonstrate the value of summary tables compared to detail, let's look at a relatively simple, yet realistic, query against the Project REAL full-sized database.

```
SELECT Subject, SUM(Sales.Sales_Qty) AS Sales_Qty
FROM Tbl_Fact_Store_Sales as Sales
```

```
JOIN Tbl_Dim_Store Store
  ON Sales.SK_Store_ID = Store.SK_Store_ID
JOIN Tbl_Dim_Date Date
  ON Sales.SK_Date_ID = Date.SK_Date_ID
JOIN Tbl_Dim_Item Item
  ON Sales.SK_Item_ID = Item.SK_Item_ID
WHERE Date.Fiscal_Period_Desc IN ('F04 P5 (JUN)')
  AND Store.Region = 'West'
GROUP BY Subject
ORDER BY Sales_Qty DESC
```

Avg response time: 1 minute 43 seconds

This query references the detailed fact table directly. Not only does it take a while to run, but each relevant detail record is read into buffer cache, lowering the page life expectancy and increasing I/O for all queries on the SQL Server instance. The query was rewritten, below, to reference our summary table instead of the detail. This improved the query time by over 87%.

```
SELECT Subject, SUM(Sales_Qty) AS Sales_Qty
FROM Tbl_Fact_Summary as summary
JOIN (SELECT DISTINCT Region, District_Num
FROM Tbl_Dim_Store) Tbl_Dim_Store
  ON summary.Store_District_Num = Tbl_Dim_Store.District_Num
JOIN (SELECT DISTINCT Subject, Category_Code
FROM Tbl_Dim_Item) Tbl_Dim_Item
  ON summary.Item_Category_Code = Tbl_Dim_Item.Category_Code
JOIN Tbl_Dim_Date
  ON summary.Date_Summary_Key = Tbl_Dim_Date.SK_Date_ID
WHERE Tbl_Dim_Date.Fiscal_Period_Desc IN ('F04 P5 (JUN)')
  AND Tbl_Dim_Store.Region = 'West'
GROUP BY Subject
ORDER BY Sales_Qty DESC
```

Avg response time: 13 seconds

The expense of maintaining summary tables should be carefully controlled. The ETL process can be modified to keep track of updated dates/partitions. This information can be used to only update (re-calculate) aggregations on the rows of the summary table that were affected. Assure that the update environment is tightly controlled to make sure the numbers in the summary table are always accurate. How this is controlled depends on how the ETL process was implemented.

Improving summary table performance

The summary tables that are created should be at a higher grain for most of the joined dimension tables. For instance, if the detailed fact data is at the day level of the Date dimension, the summary Date grain may be at the week, month, or quarter level. At that point, you either need to carry the associated properties for that level and above in the summary table, or join back to the dimension to get that information. You will usually choose the latter option, especially if there are several properties that would be required. This results in more flexibility if the dimension tables change and also improves the performance of the summary table build and maintenance.

If the dimensional table design that is the basis for the summary tables is in a snowflake schema, this is very easy. The table that represents the higher level in the dimension table is simply joined to the summary table. If the dimension tables are flattened, per the star schema design, it is easy to introduce a Cartesian product into the resulting queries that join for the dimension property information since there are several rows that represent data at the higher level. To solve this, we make sure we select only distinct values for the higher level. Revisiting the query from above, the bold lines represent subqueries to return these distinct values.

```
SELECT Subject, SUM(Sales_Qty) AS Sales_Qty
FROM Tbl_Fact_Summary as summary
JOIN (SELECT DISTINCT Region, District_Num
FROM Tbl_Dim_Store) Tbl_Dim_Store
  ON summary.Store_District_Num = Tbl_Dim_Store.District_Num
JOIN (SELECT DISTINCT Subject, Category_Code
FROM Tbl_Dim_Item) Tbl_Dim_Item
  ON summary.Item_Category_Code = Tbl_Dim_Item.Category_Code
JOIN Tbl_Dim_Date
  ON summary.Date_Summary_Key = Tbl_Dim_Date.SK_Date_ID
WHERE Tbl_Dim_Date.Fiscal_Period_Desc IN ('F04 P5 (JUN)')
  AND Tbl_Dim_Store.Region = 'West'
GROUP BY Subject
ORDER BY Sales_Qty DESC
```

Note that we have to query far more rows than we need to in the dimension tables. In large dimension tables, such as the Item dimension, this can add significantly to the query's response time. A simple measure is to implement a modified snowflake schema behind the scenes by using indexed views. In this case, we created one for the item category and the store district, per the subqueries highlighted above. An example of the item category indexed view definition follows:

```
CREATE VIEW [dbo].[vTbl_Dim_Item_Category] WITH SCHEMABINDING
AS
SELECT Subject_Code, Subject, Category_Code, Category,
```

```
COUNT_BIG(*) AS CountBig
FROM     dbo.Tbl_Dim_Item
GROUP BY Subject_Code, Subject, Category_Code, Category
GO
CREATE UNIQUE CLUSTERED INDEX [IV_Dim_Item_Category] ON
[dbo].[vTbl_Dim_Item_Category]
(
      [Category_Code] ASC,
      [Subject] ASC,
      [Subject_Code] ASC
)
GO
```

We added properties from the parent dimension levels (Subject, in this case) to facilitate the automated use of the indexed views. Now we can rerun the query with no changes and it will utilize the new indexed views for the item department and store district. The query response time was reduced from 13 seconds to 1 second in the Project REAL full-sized database, with no changes to the query. It is a best practice to reference the indexed view name explicitly in queries and use the NOEXPAND hint. This ensures that the indexed views will be used instead of the base tables.

The general recommendation is to create an indexed view for each level represented at or above the grain in the summary tables. Include all columns related to that grain and above. Create the unique, clustered index based on the grain represented. For the store city level, for instance, index by city and state (you cannot index by city alone as it is not the unique index that is required for an indexed view). Nonclustered indexes can be added to the views as necessary to facilitate the queries.

Using the a combination of a well-designed summary table and indexed views on dimension tables, we were able to reduce the response time for a query that ran an average of 1 minute 43 seconds to 1 second! That's an improvement of 99%, or over 100 times faster. Additionally, several queries should be able to benefit from using this single summary table.

Summary tables and Report Builder

We've shown how summary tables can provide a tremendous benefit to a majority of queries. "Super users" who write Transact-SQL queries and can understand the contents of the summary tables and how to use them can incorporate them into their daily queries to greatly enhance performance. Typical business users may have a lesser skill set around building query code and no time or desire to learn this skill.
SQL Server 2005 Reporting Services released a component to facilitate ad hoc queries called Report Builder specifically for this type of user. One or more report models must be defined in order for Report Builder to report against a SQL Server relational database. A report model is built on top of a DSV (data source view) and these two concepts, together, can provide the tools to insert summary tables into the query in place of fact tables as appropriate.

There are two general approaches to incorporating summary tables into the report model. The first is to create a separate report model for each or most of the summary tables, and then one more for the detailed view. The key is to name the report model appropriately so that it is obvious to the user which one is desirable to select when creating a new Report Builder report. The biggest issue with this approach is that if the user selects an incorrect report model, they will have to start over with a new report.

Another approach is to incorporate the summary tables into a single report model that also includes the detailed fact information. This will present multiple views of the data to the end user. When using a summary-based hierarchy in the report model, the user will automatically gain access only to the levels of information in the remaining dimensions in the summary table. The biggest consideration with this strategy is to configure the report model and summary tables in such a way that users are not confused by the hierarchies and will not repeatedly select inappropriate hierarchies.

In both approaches, the dimension tables that are related to the summary level information can be sourced from the indexed views that were previously mentioned, or to named queries that are created within the source DSV.

Conclusion

A scalable, high-performance data warehouse can be developed irrespective of the size of the underlying database(s). Both query performance and the batch update window can be managed by building a solid foundation as follows:

- Create a foundation based on a robust hardware platform and a properly configured SQL Server environment. Consider an architecture where the less frequently accessed data is isolated completely from data which is in high demand.
- Rethink some of the perceived table design best practices. Carefully plan your indexing strategy, and use table and index partitioning to make your data warehouse more manageable.
- Study the query load on your server and build strategic pre-aggregations to facilitate good query performance.

By developing a well-thought-out data warehousing strategy and continuously monitoring your batch maintenance and query environment, you can deliver a data mart that provides high performance and high availability so that your users can focus on Business Intelligence.

References

Data Warehousing concepts and dimensional modeling

- "The Data Warehouse Lifecycle Toolkit" by Ralph Kimball, Laura Reeves, Margy Ross and Warren Thornthwaite
- "The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling" by Ralph Kimball and Margy Ross
- "The Microsoft Data Warehouse Toolkit" by Joy Mundy and Warren Thornthwaite with Ralph Kimball

SQL Server 2005 table partitioning

- SQL Server 2005 Books Online
- Project REAL: Data Lifecycle – Partitioning by Erin Welker (http://www.microsoft.com/technet/prodtechnol/sql/2005/realpart.mspx)

- Partitioned Tables and Indexes in SQL Server 2005 by Kimberly Tripp (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql2k5partition.asp)

SQL Server indexed views

- SQL Server 2005 Books Online
- Improving Performance with SQL Server 2005 Indexed Views by Eric Hanson (http://www.microsoft.com/technet/prodtechnol/sql/2005/ipsql05iv.mspx)

SQL Server 2005 and tempdb

- Working with tempdb in SQL Server 2005 by Wei Xiao, Matt Hink, Mirek Sztajno, and Sunil Agarwal (http://download.microsoft.com/download/4/f/8/4f8f2dc9-a9a7-4b68-98cb-163482c95e0b/WorkingWithTempDB.doc)

Project REAL (http://www.microsoft.com/sql/solutions/bi/projectreal.mspx)

For more information:

http://www.microsoft.com/technet/prodtechnol/sql/default.mspx

Did this paper help you? Please give us your feedback. On a scale of 1 (poor) to 5 (excellent), how would you rate this paper?