Best Practices for Data Warehousing with SQL Server 2008

SQL Server 2008 SQL Server Technical Article

Writers: Mark Whitehorn, Solid Quality Mentors; Keith Burns, Microsoft

Technical Reviewer: Eric N. Hanson, Microsoft

Published: July 2008

Applies to: SQL Server 2008

Summary: There is considerable evidence that successful data warehousing projects often produce a very high return on investment. Over the years a great deal of information has been collected about the factors that lead to a successful implementation versus an unsuccessful one. These are encapsulated here into a set of best practices, which are presented with particular reference to the features in SQL Server 2008. The application of best practices to a data warehouse project is one of the best investments you can make toward the establishment of a successful Business Intelligence infrastructure.

Introduction

Microsoft SQL Server 2008 represents an excellent choice for the construction and maintenance of data warehouses in enterprises of all sizes.

The term Business Intelligence (BI) describes the process of extracting information from data. The operational data in most enterprises is held in transaction-based systems with specific functions (HR, Sales, Finance, and so on). Frequently the information requested by decision makers within the enterprise requires data from several of the operational systems. Indeed, the more general the question, such as “What is our current profit?” the more operational systems are likely to be involved in providing data.

An integral part of any BI system is the data warehouse—a central repository of data that is regularly refreshed from the source systems. The new data is transferred at regular intervals (often nightly) by extract, transform, and load (ETL) processes.

Typically the data in the data warehouse is structured as a star schema [Kim08] although it may also be structured as normalized relational data [Inmon05] or as a hybrid between the two. No matter which structure is chosen, after the new data has been loaded into the data warehouse, many BI systems copy subsets of the data to function-specific data marts where the data is typically structured as a multi-dimensional OLAP cube as shown in Figure 1.
Figure 1: Overall plan of a data warehouse

Data warehouses have been built in one form or another for over 20 years. Early in their history it became apparent that building a successful data warehouse is not a trivial undertaking. The IDC report from 1996 [IDC96] is a classic study of the state of data warehousing at the time. Paradoxically, it was used by both supporters and detractors of data warehousing.

The supporters claimed that it proved how effective data warehousing is, citing that for the 62 projects studied, the mean return on investment (ROI) over three years was just over 400 percent. Fifteen of those projects (25 percent) showed a ROI of more than 600 percent.

The detractors maintained that the report was a searing indictment of current data warehouse practices because of 45 projects (with outliers discounted) 34 percent failed to return even the cost of investment after five years. A warehouse that has shown no return in that length of time is not a good investment.

Both sets of figures are accurate and, taken together, reflect the overall findings of the paper itself which says "One of the more interesting stories is found in the range of results. While the 45 organizations included in the summary analysis reported ROI results between 3% and 1,838%, the total range varied from as low as ~ 1,857% to as high as 16,000%!"

Worryingly, the trend towards failure for data warehouse projects continues today: some data warehouses show a huge ROI, others clearly fail. In a report some nine years later (2005), Gartner predicted that "More than 50 percent of data warehouse projects will have limited acceptance or will be failures through 2007" (Gartner press release [Gart07]).

Does this mean that we have learned nothing in the intervening time? No, we have learned a great deal about how to create successful data warehouses and a set of best practices has evolved. The problem seems to be that not everyone in the field is aware of those practices.

In this paper we cover some of the most important data warehousing features in SQL Server 2008 and outline best practices for using them effectively. In addition, we cover some of the more general best practices for creating a successful data warehouse project. Following best practices alone cannot, of course, guarantee that your data warehouse project will succeed; but it will improve its chances immeasurably. And it is undeniably true that applying
best practices is the most cost-effective investment you can make in a data warehouse.

A companion paper [Han08] discusses how to scale up your data warehouse with SQL Server 2008. This paper focuses on planning, designing, modeling, and functional development of your data warehouse infrastructure. See the companion paper for more detail on performance and scale issues associated with data warehouse configuration, querying, and management.

**Benefits of Using Microsoft Products for Data Warehousing and Business Intelligence**

BI systems are, of necessity, complex. The source data is held in an array of disparate operational applications and databases. A BI system must turn these nonhomogeneous sets into a cohesive, accurate, and timely set of useful information.

Several different architectures can be successfully employed for data warehouses; however, most involve:

- Extracting data from source systems, transforming it, and then loading it into a data warehouse
- Structuring the data in the warehouse as either third normal form tables or in a star/snowflake schema that is not normalized
- Moving the data into data marts, where it is often managed by a multidimensional engine
- Reporting in its broadest sense, which takes place from data in the warehouse and/or the data marts: reporting can take the form of everything from printed output and Microsoft Office Excel® spreadsheets through rapid multidimensional analysis to data mining.

SQL Server 2008 provides all the tools necessary to perform these tasks [MMD07].

- SQL Server Integration Services (SSIS) allows the creation and maintenance of ETL routines.
- If you use SQL Server as a data source, the Change Data Capture feature simplifies the extraction process enormously.
- The SQL Server database engine holds and manages the tables that make up your data warehouse.
- SQL Server Analysis Services (SSAS) manages an enhanced multidimensional form of the data, optimized for fast reporting and ease of understanding.
- SQL Server Reporting Services (SSRS) has a wide range of reporting abilities, including excellent integration with Excel and Microsoft Office Word. PerformancePoint Server™ makes it easy to visualize multidimensional data.

Moreover, Microsoft Office SharePoint Server® (MOSS) 2007 and Microsoft Office 2007 provide an integrated, easy-to-use, end-user environment, enabling you to distribute analysis and reports derived from your data warehouse data throughout your organization. SharePoint can be used to build BI portal and dashboard solutions. For example, you can build a score card application on top of SharePoint that enables employees to get a custom display of the metrics and Key Performance Indicators (KPIs) depending on their job role.

As you would expect from a complete end-to-end solution, the level of integration between the components is extremely high. Microsoft Visual Studio® provides a consistent UI from which to drive both SQL Server and Analysis Services and, of course, it can be used to develop BI applications in a wide range of languages (Visual C#®, Visual C++®, Visual Basic.Net®, and so on).

SQL Server is legendary for its low total cost of ownership (TCO) in BI solutions. One reason is that the tools you need come in the box at no extra cost—other vendors charge (and significantly) for ETL tools, multidimensional engines, and so on. Another factor is the Microsoft renowned focus on ease of use which, when combined with the integrated development environment that Visual Studio brings, significantly reduces training times and development costs.
**Best Practices: Creating Value for Your Business**

The main focus of this paper is on technical best practices. However, experience has shown that many data warehouse and BI projects fail not for technical reasons, but because of the three Ps—personalities, power struggles, and politics.

**Find a sponsor**

Your sponsor should be someone at the highest level within the organization, someone with the will and the authority to give the project the strong political backing it will need. Why?

Some business people have learned that the control of information means power. They may see the warehouse as an active threat to their power because it makes information freely available. For political reasons, those people may profess full support for the BI system, but in practice be effectively obstructive.

The BI team may not have the necessary influence to overcome such obstacles or the inertia that may come from the higher management echelons: that is the job of the sponsor.

**Get the architecture right at the start**

The overall architecture of the entire BI system must be carefully planned in the early stages. An example of part of this process is deciding whether to align the structure with the design principles of Ralph Kimball or Bill Inmon (see References).

**Develop a Proof of Concept**

Deciding upon a Proof of Concept project is an excellent way to gain support and influence people. This might involve the creation of an OLAP cube and the use of visualization software for one (or several) business units. The project can be used to show business people what they can expect from the new system and also to train the BI team. Choose a project with a small and well-defined scope that is relatively easy to achieve. Performing a Proof of Concept requires an investment of time, effort, and money, but, by limiting the scope, you limit the expenditure and ensure a rapid return on the investment.

**Select Proof of Concept projects on the basis of rapid ROI**

When choosing a Proof of Concept project, it can be a useful exercise to talk to ten or so business units about their needs. Score their requirements for difficulty and for ROI. Experience suggests that there is often little correlation between the cost of developing a cube and the ROI it can generate, so this exercise should identify several low-effort, high-return projects, each of which should be excellent candidates for Proof of Concept projects.

**Incrementally deliver high value projects**

By building the most profitable solutions first, a good ROI can be ensured for the entire data warehouse project after the initial stages. If the first increment costs, say $250,000 and the ROI is $1 million per annum, after the first three months of operation, the initial outlay will have been recouped.

The ease of use and high level of integration between Microsoft BI components are valuable assets that help you deliver results in a timely fashion; this helps you to rapidly build support from your business users and sponsors. The value remains apparent as you build on the Proof of Concept project and start to deliver benefits across the organization.

**Designing Your Data Warehouse/BI solution**

In this section, two areas of best practices are discussed. The first are general design guidelines that should be considered at the start of the project. These are followed by guidance on specifying hardware.

**Best Practices: Initial Design**
Keep the design in line with the analytical requirements of the users

As soon as we start to discuss good data warehouse design we must introduce both Bill Inmon [Inmon05] and Ralph Kimball [Kim08]. Both contributed hugely to our understanding of data warehousing and both write about many aspects of warehouse design. Their views are often characterized in the following ways:

· Inmon favors a normalized data warehouse.
· Kimball favors a dimensional data warehouse.

Warehouses with these structures are often referred to as Inmon warehouses or Kimball warehouses. Microsoft has no bias either way and SQL Server is perfectly capable of supporting either design. We observe that the majority of our customers favor a dimensional warehouse. In response, we introduced star schema optimization to speed up queries against fact and dimension tables.

The computer industry has understood how to design transactional systems since the 1970s. Start with the user's requirements, which we can describe as the User model. Formalize those into a Logical model that users can agree to and approve. Add the technical detail and transform that into a Physical model. After that, it is just a matter of implementation...

Designing a data warehouse should follow the same principles and focus the design process on the requirements of the users. The only difference is that, for a data warehouse, those requirements are not operational but analytical.

In a large enterprise, users tend to split into many groups, each with different analytical requirements. Users in each group often articulate the analysis they need (the User model of analysis) in terms of graphs, grids of data (worksheets), and printed reports. These are visually very different but all essentially present numerical measures filtered and grouped by the members of one or more dimensions.

So we can capture the analytical requirements of a group simply by formalizing the measures and dimensions that they use. As shown in Figure 2 these can be captured together with the relevant hierarchical information in a sun model, which is the Logical model of the analytical requirements.
Figure 2: A sun model used to capture the analytical requirements of users in terms of measures, dimensions, and hierarchical structure. This is the Logical model derived from the User model.

Once these analytical requirements have been successfully captured, we can add the technical detail. This transforms the Logical model into a Physical one, the *star schema*, illustrated in Figure 3.
Figure 3: A representation of a star schema, which is an example of a Physical model of analytical requirements

The star schemas may ultimately be implemented as a set of dimensional and fact tables in the data warehouse and/or as cubes (ROLAP, HOLAP, or MOLAP), which may be housed in multiple data marts.

To design the ETL system to deliver the data for this structure, we must understand the nature of the data in the source systems.

To deliver the analytics that users outlined in their requirements, we must find the appropriate data in the source systems and transform it appropriately. In a perfect world, the source systems come with excellent documentation, which includes meaningful table and column names. In addition, the source applications are perfectly designed and only allow data that falls within the appropriate domain to be entered. Back on planet Earth, source systems rarely meet these exacting standards.

You may know that users need to analyze sales by the gender of the customer. Unfortunately, the source system is completely undocumented but you find a column named Gen in a table called CUS_WeX4. Further investigation shows that the column may store data relating to customer gender. Suppose that you can then determine that it contains 682370 rows and that the distribution of the data is:
You now have more evidence that this is the correct column and you also have an idea of the challenges that lie ahead in designing the ETL process.

In other words, knowledge of the distribution of the data is essential in many cases, not only to identify the appropriate columns but also to begin to understand (and ultimately design) the transformation process.

**Use data profiling to examine the distribution of the data in the source systems**

In the past, gaining an understanding of the distribution of data in the source systems meant running multiple queries (often GROUP BYs) against the source system tables to determine the domain of values in each column. That information could be compared to the information supplied by the domain experts and the defined transformations.

Integration Services has a new Data Profiling task that makes the first part of this job much easier. You can use the information you gather by using the Data Profiler to define appropriate data transformation rules to ensure that your data warehouse contains “clean” data after ETL, which leads to more accurate and trusted analytical results. The Data Profiler is a data flow task in which you can define the profile information you need.

Eight data profiles are available; five of these analyze individual columns:

- Column Null Ratio
- Column Value Distribution
- Column Length Distribution
- Column Statistics
- Column Pattern
Three analyze either multiple columns or relationships between tables and columns:

- Candidate Key
- Functional Dependency
- Value Inclusion

Multiple data profiles for several columns or column combinations can be computed with one Data Profiling task and the output can be directed to an XML file or package variable. The former is the best option for ETL design work. A Data Profile Viewer, shown in Figure 4, is provided as a standalone executable that can be used to examine the XML file and hence the profile of the data.

Figure 4: The Data Profile Viewer

Design from the start to partition large tables, particularly large fact tables

No matter how good the indexing and how fast the hardware, large tables can lead to time-consuming system management operations, such as index creation and removal of old data. This can be particularly apparent for large fact tables. So, partition the tables if they are large. In general, if a table is larger than 50 GB, you should partition it (see Relational Data Warehouse Setup, Query, and Management later in this white paper).

Plan to age out old data right from the start

One of the main functions of a data warehouse is to track the business history of the enterprise—something that the source systems generally do particularly badly. Tracking history means that the data warehouse will acquire vast quantities of data over time. As data ages, it is accessed less frequently and also differently (typically as aggregations rather than the detail). It eventually becomes necessary to treat older data differently, perhaps by using slower, cheaper storage, perhaps storing only the aggregations, removing it altogether, or another plan. It is vital to plan for this right from the start. Partitioning makes this much easier (see Relational Data Warehouse Setup, Query, and Management later in this white paper).
Best Practices: Specifying Hardware

Design for maintenance operation performance, not just query performance

Specifying hardware is never easy but it is common (and good) practice when making hardware decisions to focus most of the attention on query performance for the simple reason that query performance is vital. However, it is also important not to overlook other considerations.

Think about query performance. A very simplistic view is that it scales in a linear fashion with data size. Generations of DBAs have learned the hard way that it often does not scale in this way—double the size of the data and the query may take ten times as long to run—so they may plan hardware requirements accordingly. But this is a somewhat simplistic approach.

For example, consider a query that is run against a 1-terabyte table. Its performance is limited by many factors—indexing, memory, number of rows scanned, number of rows returned and so on. Imagine that the query uses indexes efficiently, scans some fixed number of rows, and returns only a few rows. If we run that same query against 2 terabytes of data in the same table and assume the indexing has been applied efficiently and that the number of scanned and returned rows is not significantly different, the response time is about the same. Certainly it will be nothing like double the time. In other words, query performance can sometimes scale in a nonlinear way to our advantage. However other factors, such as backup time, do scale linearly in the expected way. Backing up twice the data takes twice the resources (such as CPU time and I/O bandwidth), which may well affect the hardware specifications needed to meet your requirements.

In other words, when we design large BI systems we must be careful to consider all the relevant factors. This does not mean that query performance is unimportant; it is still the most important consideration. But it is also a mistake to focus solely on that issue. Other factors, such as maintenance of the BI system (both the relational and multidimensional components) must be considered carefully as well.

Specify enough main memory so most queries never do I/O

According to Microsoft SQL Server Customer Advisory Team engineers, in most data warehouses the data is very unevenly accessed. By far the majority of queries access some proportion of the same 20% of the total; in other words, about 80% of the data is rarely accessed. This means that disk I/O for the warehouse overall reduces dramatically if there is enough main memory to hold about 20% of the total data.

ETL

The field of ETL (extract, transform, and load) is the most complex area of data warehousing [Hath07]. The data itself is often complex and requires a whole range of techniques and processes to be applied before it is loaded into the warehouse. These include merging, time/date stamping, de-duplication and survivorship, data type conversion, normalization and/or denormalization, surrogate key insertion, and general cleansing.

Best Practices: Simplify the ETL Process and Improve Performance

Use SSIS to simplify ETL programming

SSIS was designed from the ground up to simplify the process of developing your ETL code. Its automated tools and predefined transformations and connectors vastly reduce the number of lines of ETL code you have to write compared to programming in a traditional high-level language or scripting language. It is a best practice to use SSIS rather than these other methods if reducing the amount of code for ETL and simplifying ETL job maintenance is important to you.

SSIS provides a comprehensive set of features for building data warehouses, including:

- A scalable pipeline architecture that provides a multithreaded 64-bit platform to transform growing data volumes in narrower batch windows,
- Connectivity to non-SQL Server RDBMs, mainframes, ERP systems and other
heterogeneous data sources.
- A large number of complex transformations to consolidate data from numerous systems.
- Advanced data cleansing transformations to reduce data duplication and dirty data.
- Data warehouse awareness by providing out-of-the-box capability to manage slowly changing dimensions.
- Seamless integration with the SQL Server BI platform to directly build Analysis Services cubes and load into Analysis Services partitions.
- Extensibility with the power of .NET by incorporating custom scripts and pluggable components directly into the data integration flow.

**Simplify the transformation process by using Data Profiling tasks**

The new Data Profiling task in Integration Services can be used to initially understand the nature of the source data for design purposes (see [Designing Your Data Warehouse/BI Solution](msdn.microsoft.com/.../cc719165.aspx)). However, the profiles it produces can also be used to apply business rules to data as part of the transformation process. Suppose, for example, the business rule says that the data from a particular source is acceptable only if the number of nulls does not exceed 1%. The profiles produced by a Data Profiling task can be used to apply this rule.

Note that Data Profiling tasks profile SQL Server tables; data in other locations must be loaded into staging tables before it can be profiled.

**Simplify using MERGE and INSERT INTO**

Whether you are extracting from the source systems into an Operational Data Store (ODS) or from the ODS into the fact or dimension tables, you must manage the movement of the changes (the deltas) that have occurred. During this phase you often need multiple Data Manipulation Language (DML) queries in order to perform one logical movement of the deltas into the relevant table. This is particularly true when you have to deal with slowly changing dimensions (and who doesn’t?).

SQL Server 2008 allows you to combine these multiple queries into one MERGE statement.

**MERGE**

The MERGE statement performs insert, update, or delete operations on a target table based on the results of a join with a source table.

The MERGE statement provides three types of WHEN clauses:

- WHEN MATCHED enables you to UPDATE or DELETE the given row in the target table when the source and target rows match some criteria or criterion.
- WHEN NOT MATCHED [BY TARGET] enables you to INSERT a row into the target when it exists in the source but not in the target.
- WHEN NOT MATCHED BY SOURCE enables you to UPDATE or DELETE the given row in the target table when it exists in the target but not in the source.

You can specify a search condition with each of the WHEN clauses to choose which type of DML operation should be performed on the row.

The OUTPUT clause for the MERGE statement includes a new virtual column called $action that you can use to identify the DML action that was performed on each row.

To illustrate the use of the MERGE statement, imagine that you have a table of Employees:

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Title</th>
<th>IsCurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.08.2010</td>
<td>Best Practices for Data Warehousing w...</td>
<td>msdn.microsoft.com/.../cc719165.aspx</td>
<td>11/32</td>
</tr>
</tbody>
</table>
Type 1 slowly changing dimension

Suppose first that the Employee table is a Type 1 slowly changing dimension, meaning that changes to the Title field are simply allowed to overwrite the existing value and no history is kept of the change nor of the previous value. Further assume that new rows in the source are to be inserted into the Employees table as well. You can manage this simple case with a MERGE statement:

MERGE Employee as TRG
USING EmployeeDelta AS SRC
ON (SRC.EmpID = TRG.EmpID)
WHEN NOT MATCHED THEN
INSERT VALUES (SRC.EmpID, SRC.Name, SRC.Title, 'Yes')
WHEN MATCHED THEN
UPDATE SET TRG.Title = SRC.Title

If we add an OUTPUT clause like this:
OUTPUT $action, SRC.EmpID, SRC.Name, SRC.Title;

we get the following result rowset in addition to the effect on the Employee table:
This can be very helpful with debugging but it also turns out to be very useful in dealing with Type 2 slowly changing dimensions.

**Type 2 slowly changing dimensions**

Recall that in a Type 2 slowly changing dimension a new record is added into the Employee dimension table irrespective of whether an employee record already exists. For example, we want to preserve the existing row for Jones but set the value of IsCurrent in that row to 'No'. Then we want to insert both of the rows from the delta table (the source) into the target.

```
MERGE Employee as TRG
USING EmployeeDelta AS SRC
ON (SRC.EmpID = TRG.EmpID AND TRG.IsCurrent = 'Yes')
WHEN NOT MATCHED THEN
  INSERT VALUES (SRC.EmpID, SRC.Name, SRC.Title, 'Yes')
WHEN MATCHED THEN
  UPDATE SET TRG.IsCurrent = 'No'
OUTPUT $action, SRC.EmpID, SRC.Name, SRC.Title;
```

This statement sets the value of IsCurrent to 'No' in the existing row for Jones and inserts the row for Heron from the delta table into the Employee table. However, it does not insert the new row for Jones. This does not present a problem because we can address that with the new INSERT functionality, described next. In addition, we have the output, which in this case is:

```
$action  EmpID  Name  Title
INSERT  6  Heron  Mr
UPDATE  1  Jones  Prof
```

**New functionality for INSERT**

We can combine the capacity to output the data with a new ability in SQL Server 2008 to have INSERT statements consume the results of DML statements. This ability to consume output can, of course, be used very effectively (and simply) as follows:
INSERT INTO Employee (EmpID, name, Title)

    SELECT EmpID, name, Title from EmployeeDelta

If we combine this new capability with the output above, we have the synergistic ability to extract the row that was updated (Jones) and insert it into the Employees table to achieve the desired effect in the context of Type 2 slowly changing dimensions:

INSERT INTO Employee (EMPID, Name, Title, IsCurrent)

SELECT EMPID, Name, Title, 'Yes'

FROM

(  
    MERGE Employee as TRG
    USING EmployeeDelta AS SRC
    ON (SRC.EmpID = TRG.EmpID AND TRG.IsCurrent = 'Yes')
    WHEN TARGET NOT MATCHED THEN
        INSERT VALUES (SRC.EmpID, SRC.Name, SRC.Title, 'Yes')
    WHEN MATCHED THEN
        UPDATE SET TRG.IsCurrent = 'No'
    OUTPUT $action, SRC.EmpID, SRC.Name, SRC.Title
)

As Changes (action, EmpID, Name, Title)

WHERE action = 'UPDATE';

MERGE in SQL Server 2008 was implemented to comply with the SQL-2006 standard. The main reason for the introduction of MERGE into the SQL standard and into SQL Server 2008 is its usefulness in managing slowly changing dimensions but it is worth remembering that both MERGE and INSERT with output have many other applications both within data warehousing specifically and in databases in general.

Terminate all SQL statements with a semi-colon in SQL Server 2008

Prior to SQL Server 2005, Transact-SQL was relatively relaxed about semi-colons; now some statements (including MERGE) require this terminator. If you terminate all SQL statements with a semi-colon, you avoid problems when the use of a semi-colon is obligatory.

If you cannot tolerate downtime, consider using "ping pong" partitions

Imagine that you have a fact table that is partitioned by month. It is currently August. You need to load today’s data into the August partition, but your users cannot tolerate the performance hit and potential locking conflicts that will be incurred as you load it. Copy the August partition to another table (a working table), load the data into that table, index it as needed, and then simply switch the partitions between the two tables.

Use minimal logging to load data precisely where you want it as fast as possible

Writing data to a database typically involves two separate write-to-disk processes, once writing to the database and once to the log (so transactions can be rolled back or re-done). When inserting data into an existing table it is, in fact, possible to write only once in some cases by using the minimally logged INSERT feature.

Minimal logging enables transactions to be rolled back but does not support point-in-time
recovery. It is also only available with the bulk logged and simple recovery models. In SQL Server 2008, minimal logging can be used with INSERT INTO...SELECT FROM Transact-SQL statements when inserting a large number of rows into an existing table if they are inserted into an empty table with a clustered index and no nonclustered indexes, or a heap, empty or not, with no indexes. (For full details and any late-breaking changes, see SQL Server 2008 Books Online.)

This extends the support for minimal logging, which in SQL Server 2005 included bulk import, SELECT INTO, index creation, and rebuild operations.

One huge benefit of minimal logging is that it speeds up the loading of empty partitions or tables that are on specific filegroups. In SQL Server 2005, you could achieve effectively the same effect by using a work-around that involved changing the default filegroup and performing a SELECT INTO to get minimal logging. Then the default filegroup would be returned to its initial state. Now you can just create a table on the filegroup(s) you want it to be in, define its partitioning scheme and then load it with INSERT INTO tbl WITH(TABLOCK) SELECT FROM and you achieve minimal logging.

Minimal logging makes it much easier to put the data just where you want it and write it to disk only once. As an added bonus, load performance is increased and the amount of log space required is reduced.

**Simplify data extraction by using Change Data Capture in the SQL Server source systems**

SQL Server 2008 has a new data tracking feature that is of particular benefit in data warehousing. The Change Data Capture process tracks changes to user tables and collects them into a relational format. A typical use would be to track changes in an operational database for later inclusion in the warehouse.

The capture process collects change data from the database’s transaction log and inserts it into a change table. Metadata about each transaction is also inserted into a metadata table so that changes can be ordered with regard to time. This enables the identification of, for instance, the type of change made to each row, and which column or columns changed in an updated row. It is also possible to request all rows that changed between two time/dates. Change Data Capture is a big step towards improved extraction performance, and makes programming the change capture portion of your ETL jobs much easier.

**Simplify and speed up ETL with improved Lookup**

The performance of the Lookup component has greatly improved performance in SQL Server 2008 Integration Services and is much easier to program.

A Lookup verifies whether each row in a stream of rows has a matching row in a set of reference data. This is often used within the ETL process to check, for example, the ProductID column in a fact table (acting as the data source) against a dimension table holding a complete set of products (the reference set).

In SQL Server 2008, the Lookup transformation supports two connection types when connecting to the reference dataset: the Cache connection manager and the OLE DB connection manager. The reference dataset can be a cache file, an existing table or view, a new table, or the result of an SQL query.

Reference data is usually cached for efficiency and now a dataflow can be used to populate the cache. Many potential sources can be used as reference data: Excel, XML, text, Web services—anything within reach of an ADO.Net provider. In SQL Server 2005, the cache could only be populated by an SQL query and a Lookup could only take data from specific OLE /DB connections. The new Cache Transform component populates a cache defined by the Cache connection manager.

The cache no longer needs to be reloaded each time it is used: this removes the speed penalty incurred by reloading from a relational source. If a reference dataset is used by two pipelines in a single package, the cache can be saved to permanent file storage as well as to virtual memory so it is available to multiple Lookups within one package. Furthermore the cache file format is optimized for speed and its size is unrestricted.
The miss-cache feature is also new. When running directly against the dataset, a Lookup component can add to the cache any key values from the source where there is no matching value in the reference dataset. So if Lookup has once determined that the reference set does not contain, for example, the value 885, it does not waste time inspecting the reference set for that value if it appears again in the source data. Under certain conditions this feature can produce a performance improvement of 40%.

Finally there is now a 'Lookup no match output’ to which ‘miss-cache’ rows can be directed instead of going to the error output.

Relational Data Warehouse Setup, Query, and Management

The relational database is the heart of any BI system. Best practices here affect not only the performance of the entire system, but also its flexibility and value to the enterprise. For a more in-depth discussion of how to get the best performance from your SQL Server 2008 data warehouse for large-scale data warehouses, see the companion paper [Han08].

Best Practices: General

Use the resource governor to reserve resources for important work such as data loading, and to prevent runaway queries

The workload on a data warehouse can be thought of as a series of requests that compete for the available resources. In SQL Server 2005, there was a single pool of resources and requests competed equally for those. The resource governor in SQL Server 2008 allows the available resources to be allocated into multiple (up to 20) pools. Requests are classified so that they fall into specific groups and those groups are allocated to the resource pools—many requests to each resource pool. Processes that must complete rapidly (such as the data load) can be allocated high resources when they run. In addition, important reports can be allocated enough resources to ensure that they complete rapidly [Bar08].

Many users find unpredictability of performance highly frustrating. If this occurs, it is beneficial to use the resource governor to allocate resources in a way that ensures more predictable performance. Over time, as experience with the resource governor grows, we expect this to evolve into a best practice.

Suppose, as is often the case, that the data warehouse is used for both reporting and ETL processes and is configured for zero or minimal downtime during loads. In some enterprises (despite what the BI team might prefer), the generation of reports on time is seen as more important than the completion of the daily load. In this case the reporting workload group would be allocated a high priority.

Or, by contrast, the load processes may be given high priority in order to guarantee the minimum downtime that the business requires.

Finally, it is worth noting that the resource governor also enables you to monitor the resource consumption of each group, which means that resource usage can be better understood, which in turn allows better management.

Carefully plan when to rebuild statistics and indexes

The query optimizer uses information from the database statistics (number of rows, data distribution, and so on) in order to help determine the optimal query plan. If the statistics are inaccurate, a less optimal plan may be chosen, which degrades query performance.

If you can afford the time during your ETL process, rebuild statistics after every load of the data warehouse. This ensures that the statistics are always accurate. However, rebuilding statistics takes time and resources. In addition, the time between rebuilds is less important than the change to the distribution of the data that has occurred.

When the data warehouse is new and evolving, and also relatively small, it makes sense to update statistics frequently, possibly after every load. As the warehouse matures, you can sometimes reduce the frequency of rebuilding without degrading query performance significantly. If it is important to reduce the cost of updating statistics, you can determine the appropriate statistic refresh frequency by monitoring query performance as you reduce the frequency of
refreshes. Be aware that if the bulk of your queries target only the most recently loaded data, you will not have statistics for that data unless you update statistics after each load. This is a fairly common situation, which is why we recommend updating statistics after each load of the data warehouse by default.

**Best Practices: Date/time**

Use the correct time/date data type

SQL Server 2008 has six date and time data types:

- **date**
- **datetime2**
- **datetime**
- **datetimeoffset**
- **smalldatetime**
- **time**

These store temporal information with varying degrees of precision. Choosing the right one enables you to maintain accurate time and date information in a way that suits your application best, saves storage space, and improves query performance. For example, many older SQL Server applications use **datetime** for dates, but leave the time portion blank. This takes more space than necessary. Now, you can use the new **date** type for these columns, which takes only three bytes compared to eight bytes for **datetime**.

Consider using **datetime2** in some database ports

**datetime2** can be considered to be an extension of the existing **datetime** type—it combines a date with a time based on the 24-hour clock. However, **datetime2** has a larger date range, a larger default fractional precision, and an optional user-specified precision. The precision is such that it can store fractions of a second to seven digits—in other words to within one ten millionth of a second. This new feature can influence the porting of some data warehouse applications.

For example, the DB2 TimeStamp data type has an accuracy of one millionth of a second. In a data warehouse application written on DB2, if the application logic is built to ensure that new records are created at least one microsecond apart (which is not a particularly onerous limitation), time/date can be used as a unique ID. We can debate whether it is a best practice to design a database application in this way but the fact is that it is sometimes done with DB2.

Before the advent of **datetime2**, if such an application were ported to SQL Server, the application logic would have to be rewritten because **datetime** provides an accuracy of only a thousandth of a second. Because **datetime2** is ten times more precise than DB2’s TimeStamp, the application can now be ported with no change to the logic.

**Best Practices: Compression and Encryption**

Use **PAGE compression** to reduce data volume and speed up queries

Full-blown data compression capability has been added to SQL Server 2008; the improvements come in two types—row and page.

Row compression stores all fields in variable width format. If the data is compressible, row compression reduces the number of bytes required to store it.

Page compression does the same but the compression takes place between the rows within each page. A page-level dictionary is used to store common values, which are then referenced from the rows themselves rather than stored redundantly. In addition, common prefixes of column values are stored only once on the page. As an illustration of how prefix compression can help, consider product codes where the prefixes are often similar.
<table>
<thead>
<tr>
<th>Code</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-F234-10-1234</td>
<td>1834</td>
</tr>
<tr>
<td>A-F234-10-1235</td>
<td>1435</td>
</tr>
<tr>
<td>A-F234-10-1236</td>
<td>1237</td>
</tr>
<tr>
<td>A-F234-10-1237</td>
<td>1546</td>
</tr>
<tr>
<td>A-F234-10-1238</td>
<td>1545</td>
</tr>
<tr>
<td>A-F234-10-1239</td>
<td>1543</td>
</tr>
<tr>
<td>A-F234-10-1240</td>
<td>1756</td>
</tr>
<tr>
<td>A-F234-10-1241</td>
<td>1435</td>
</tr>
<tr>
<td>A-F234-10-1242</td>
<td>1544</td>
</tr>
</tbody>
</table>

This can be compressed to:

<table>
<thead>
<tr>
<th>Code</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-F234-10-12</td>
<td>1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>834</td>
</tr>
<tr>
<td>35</td>
<td>435</td>
</tr>
<tr>
<td>36</td>
<td>237</td>
</tr>
<tr>
<td>37</td>
<td>546</td>
</tr>
</tbody>
</table>
Even a column like Quantity can benefit from compression. For more details on how row and page compression work in SQL Server 2008, see SQL Server 2008 Books Online.

Both page and row compression can be applied to tables and indexes.

The obvious benefit is, of course, that you need less disk space. In tests, we have seen compression from two- to seven-fold, with three-fold being typical. This reduces your disk requirements by about two thirds.

A less obvious but potentially more valuable benefit is found in query speed. The gain here comes from two factors. Disk I/O is substantially reduced because fewer reads are required to acquire the same amount of data. Secondly the percentage of the data that can be held in main memory is increased as a function of the compression factor.

The main memory advantage is the performance gains enabled by compression and surging main memory sizes.

Query performance can improve ten times or more if you can get all the data that you ever query into main memory and keep it there. Your results depend on the relative speed of your I/O system, memory, and CPUs. A moderately large data warehouse can fit entirely in main memory on a commodity four-processor server that can accommodate up to 128 GB of RAM—RAM that is increasingly affordable. This much RAM can hold all of a typical 400-GB data warehouse, compressed. Larger servers with up to 2 terabytes of memory are available that can fit an entire 6-terabyte data warehouse in RAM.

There is, of course, CPU cost associated with compression. This is seen mainly during the load process. When page compression is employed we have seen CPU utilization increase by a factor of about 2.5. Some specific figures for page compression, recorded during tests on both a 600-GB and a 6-terabyte data warehouse with a workload of over a hundred different queries, are a 30-40% improvement in query response time with a 10-15% CPU time penalty.

So, in data warehouses that are not currently CPU-bound, you should see significant improvements in query performance at a small CPU cost. Writes have more CPU overhead than reads.

This description of the characteristics of compression should enable you to determine your optimal compression strategy for the tables and indexes in the warehouse. It may not be as simple as applying compression to every table and index.

For example, suppose that you partition your fact table over time, such as by Quarter 1, Quarter 2, and so on. The current partition is Quarter 4. Quarter 4 is updated nightly, Quarter 3 far less frequently, and Quarters 1 and 2 are never updated. However, all are queried extensively.

After testing you might find that the best practice is to apply both row and page compression to Quarters 1 and 2, row compression Quarter 3, and neither to Quarter 4.
Your mileage will, of course, vary and testing is vital to establish best practices for your specific requirements. However, most data warehouses should gain significant benefit from implementing a compression strategy. Start by using page compression on all fact tables and fact table indexes. If this causes performance problems for loading or querying, consider falling back to row compression or no compression on some or all partitions.

If you use both compression and encryption, do so in that order

SQL Server 2008 allows table data to be encrypted. Best practice for using this depends on circumstances (see above) but be aware that much of the compression described in the previous best practice depends on finding repeated patterns in the data. Encryption actively and significantly reduces the patterning in the data. So, if you intend to use both, it is unwise to first encrypt and then compress.

Use backup compression to reduce storage footprint

Backup compression is now available and should be used unless you find good reason not to do so. The benefits are the same as other compression techniques—there are both speed and volume gains. We anticipate that for most data warehouses the primary gain of backup compression is in the reduced storage footprint, and the secondary gain is that backup completes more rapidly. Moreover, a restore runs faster because the backup is smaller.

Best Practices: Partitioning

Partition large fact tables

Partitioning a table means splitting it horizontally into smaller components (called partitions). Partitioning brings several benefits. Essentially, partitioning enables the data to be segregated into sets. This alone has huge advantages in terms of manageability. Partitions can be placed in different filegroups so that they can be backed up independently. This means that we can position the data on different spindles for performance reasons. In data warehouses it also means that we can isolate the rows that are likely to change and perform updates, deletes, and inserts on those rows alone.

Query processing can be improved by partitioning because it sometimes enables query plans to eliminate entire partitions from consideration. For example, fact tables are frequently partitioned by date, such as by month. So when a report is run against the July figures, instead of accessing 1 billion rows, it may only have to access 20 million.

Indexes as well as tables can be (and frequently are) partitioned, which increases the benefits.

Imagine a fact table of 1 billion rows that is not partitioned. Every load (typically nightly) means insertion, deletion, and updating across that huge table. This can incur huge index maintenance costs, to the point where it may not be feasible to do the updates during your ETL window.

If the same table is partitioned by time, generally only the most recent partition must be touched, which means that the majority of the table (and indexes) remain untouched. You can drop all the indexes prior to the load and rebuild them afterwards to avoid index maintenance overhead. This can greatly improve your load time.

Partition-align your indexed views

SQL Server 2008 enables you to create indexed views that are aligned with your partitioned fact tables, and switch partitions of the fact tables in and out. This works if the indexed views and fact table are partitioned using the same partition function. Typically, both the fact tables and the indexed views are partitioned by the surrogate key referencing the Date dimension table. When you switch in a new partition, or switch out an old one, you do not have to drop the indexed views first and then re-create them afterwards. This can save a huge amount of time during your ETL process. In fact, it can make it feasible to use indexed views to accelerate your queries when it was not feasible before because of the impact on your daily load cycle.

Design your partitioning scheme for ease of management first and foremost
In SQL Server 2008 it is not necessary to create partitions to get parallelism, as it is in some competing products. SQL Server 2008 supports multiple threads per partition for query processing, which significantly simplifies development and management. When you design your partitioning strategy, choose your partition width for the convenience of your ETL and data life cycle management. For example, it is not necessary to partition by day to get more partitions (which might be necessary in SQL Server 2005 or other DBMS products) if partitioning by week is more convenient for system management.

For best parallel performance, include an explicit date range predicate on the fact table in queries, rather than a join with the Date dimension.

SQL Server generally does a good job of processing queries like the following one, where a date range predicate is specified by using a join between the fact table and the date dimension:

```sql
select top 10 p.ProductKey, sum(f.SalesAmount) 
from FactInternetSales f, DimProduct p, DimDate d 
where f.ProductKey=p.ProductKey and p.ProductAlternateKey like N'BK-%' 
and f.OrderDateKey = d.DateKey 
and d.MonthNumberOfYear = 1 
and d.CalendarYear = 2008 
and d.DayNumberOfMonth between 1 and 7 
group by p.ProductKey 
order by sum(f.SalesAmount) desc
```

However, a query like this one normally uses a nested loop join between the date dimension and the fact table, which can limit parallelism and overall query performance because, at most, one thread is used for each qualifying date dimension row. Instead, for the best possible performance, put an explicit date range predicate on the date dimension key column of the fact table, and make sure the date dimension keys are in ascending order of date. The following is an example of a query with an explicit date range predicate:

```sql
select top 10 p.ProductKey, d.CalendarYear, d.EnglishMonthName, 
       sum(f.SalesAmount) 
from FactInternetSales f, DimProduct p, DimDate d 
where f.ProductKey=p.ProductKey and p.ProductAlternateKey like N'BK-%' 
and f.OrderDateKey between 20030101 and 20030107 
and f.OrderDateKey=d.DateKey 
group by p.ProductKey, d.CalendarYear, d.EnglishMonthName 
order by sum(f.SalesAmount) desc
```

This type of query typically gets a hash join query plan and will fully parallelize.

**Best Practice: Manage Multiple Servers Uniformly**

Use Policy-Based Management to enforce good practice across multiple servers.
SQL Server 2008 introduces Policy-Based Management, which makes it possible to declare policies (such as "all log files must be stored on a disk other than the data disk") in one location and then apply them to multiple servers. So a (somewhat recursive) best practice is to set up best practices on one server and apply them to all servers. For example, you might build three data marts that draw data from a main data warehouse and use Policy-Based Management to apply the same rules to all the data marts.

**Additional Resources**

You can find additional tips mostly related to getting the best scalability and performance from SQL Server 2008 in the companion white paper [Han08]. These tips cover a range of topics including storage configuration, query formulation, indexing, aggregates, and more.

**Analysis**

There are several excellent white papers on Best Practices for analysis such as OLAP Design Best Practices for Analysis Services 2005, Analysis Services Processing Best Practices, and Analysis Services Query Performance Top 10 Best Practices. Rather than repeat their content, in this paper we focus more specifically on best practices for analysis in SQL Server 2008.

**Best Practices: Analysis**

Seriously consider the best practice advice offered by AMO warnings

Good design is fundamental to robust, high-performance systems, and the dissemination of best practice guidelines encourages good design. A whole new way of indicating where following best practice could help is incorporated into SQL Server 2008 Analysis Services.

SQL Server 2008 Analysis Services shows real-time suggestions and warnings about design and best practices as you work. These are implemented in Analysis Management Objects (AMO) and displayed in the UI as blue wiggly underlines: hovering over an underlined object displays the warning. For instance, a cube name might be underlined and the warning might say:

*The ‘SalesProfit’ and ‘Profit’ measure groups have the same dimensionality and granularity. Consider unifying them to improve performance. Avoid cubes with a single dimension.*

Over 40 different warnings indicate where best practice is not being followed. Warnings can be dismissed individually or turned off globally, but our recommendation is to follow them unless you have an active reason not to do so.

Use MOLAP writeback instead of ROLAP writeback

For certain classes of business applications (forecasting, budgeting, what if, and so on) the ability to write data back to the cube can be highly advantageous.

It has for some time been possible to write back cell values to the cube, both at the leaf and aggregation levels. A special writeback partition is used to store the difference (the deltas) between the original and the new value. This mechanism means that the original value is still present in the cube; if an MDX query requests the new value, it hits both partitions and returns the aggregated value of the original and the delta.

However, in many cases, despite the business need, performance considerations have limited the use of writeback.

In the previous implementation, the writeback partition had to use ROLAP storage and this was frequently a cause of poor performance. To retrieve data it was necessary to query the relational data source and that can be slow. In SQL Server 2008 Analysis Services, writeback partitions can be stored as MOLAP, which removes this bottleneck.

While it is true that this configuration can slow the writeback commit operation fractionally (both the writeback table and the MOLAP partition must be updated), query performance dramatically improves in the majority of cases. One in-house test of a 2 million cell update showed a five-fold improvement in performance.

msdn.microsoft.com/.../cc719165.aspx
Use SQL Server 2008 Analysis Services backup rather than file copy

In SQL Server 2008 Analysis Services, the backup storage subsystem has been rewritten and its performance has improved dramatically as Figure 5 shows.

**Figure 5**: Backup performance in SQL Server 2005 Analysis Services versus SQL Server 2008 Analysis Services. X = Time Y=Data volume

Backup now scales linearly and can handle an Analysis Services database of more than a terabyte. In addition, the limitations on backup size and metadata files have been removed. Systems handling large data volumes can now adopt the backup system and abandon raw file system copying, and enjoy the benefits of integration with the transactional system and of running backup in parallel with other operations.

Although the file extension is unaltered, the format of the backup file has changed. It is fully backwardly compatible with the previous format so it is possible to restore a database backed up in SQL Server 2005 Analysis Services to SQL Server 2005 Analysis Services. It is not, however, possible to save files in the old format.

Write simpler MDX without worrying about performance

A new feature, block computation (also known as *subspace computation*), has been implemented in SQL Server 2008 Analysis Services and one of the main benefits it brings is improved MDX query performance. In SQL Server 2005 complex workarounds were possible in some cases; now the MDX can be written much more simply.

Cubes commonly contain sparse data: there are often relatively few values in a vast sea of nulls. In SQL Server 2005 Analysis Services, queries that touched a large number of nulls could perform poorly for the simple reason that even if it was logically pointless to perform a calculation on a null value, the query would nevertheless process all the null values in the same way as non-nulls.

Block computation addresses this issue and greatly enhances the performance of these queries. The internal structure of a cube is highly complex and the description that follows of how block computation works is a simplified version of the full story. It does, however, give an idea of how
the speed gain is achieved.

This MDX calculation calculates a running total for two consecutive years:

```
CREATE MEMBER CURRENTCUBE.[Measures].RM
AS ([Order Date].[Hierarchy].PrevMember,[Measures].[Order Quantity]) +
Measures.[Order Quantity],
FORMAT_STRING = "Standard",
VISIBLE = 2 ;
```

These tables show orders of a small subset of products and illustrate that there are very few values for most products in 2003 and 2004:

<table>
<thead>
<tr>
<th>Product</th>
<th>2003</th>
<th>2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product 2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Product 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product 4</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Product 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product 7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Product 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This query:

```
SELECT [Order Date].[Hierarchy].[all].[2004] on columns,
[Dim Product].[Product Key].[All].children on rows
From Foo
Where Measures.RM
```

returns the calculated measure RM for all products ordered in 2004.

In SQL Server 2005 Analysis Services, the result is generated by taking the value from a cell for 2004 (the current year), finding the value in the corresponding cell for the previous year and summing the values. Each cell is handled in this way.

This approach contains two processes that are slow to perform. Firstly, for every cell, the query
processor navigates to the previous period to see if a value is present. In most hierarchical structures we know that if data is present for one cell in 2003, it will be there for all cells in 2003. The trip to see if there is data for the previous period only needs to be carried out once, which is what happens in SQL Server 2008 Analysis Services. This facet of block computation speeds up queries regardless of the proportion of null values in the cube.

Secondly, the sum of each pair of values is calculated. With a sparse cube, a high proportion of the calculations result in a null, and time is wasted making these calculations. If we look at the previous tables, it is easy to see that only the calculations for products 2, 4, and 7 will deliver anything that is not null. So in SQL Server 2008 Analysis Services, before any calculations are performed, null rows are removed from the data for both years:

<table>
<thead>
<tr>
<th></th>
<th>2003</th>
<th>2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Product 7</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

The results are compared:

<table>
<thead>
<tr>
<th></th>
<th>2003</th>
<th>2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Product 4</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Product 7</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

and the calculations performed only for the rows that will generate a meaningful result.

The speed gain can be impressive: under testing a query that took two minutes and 16 seconds
in SQL Server 2005 Analysis Services took a mere eight seconds in SQL Server 2008 Analysis Services.

Scale out if you need more hardware capacity and hardware price is important

To improve performance under load, you can either scale up or scale out. Scale up is undeniably simple: put the cube on an extra large high-performance server. This is an excellent solution—it is quick and easy and is the correct decision in many cases. However, it is also expensive because these servers cost more per CPU than multiple smaller servers.

Previous version of Analysis Services offered a scale-out solution that used multiple cost-effective servers. The data was replicated across the servers and a load balancing solution such as Microsoft Network Load Balancing (NLB) was installed between the clients and the servers. This worked but incurred the additional costs of set up and ongoing maintenance.

SQL Server 2008 Analysis Services has a new scale-out solution called Scalable Shared Database (SSD). Its workings are very similar to the SSD feature in the SQL Server 2005 relational database engine. It comprises three components:

- Read-only database – enables a database to be designated ‘read-only’
- Database storage location – enables a database to reside outside the server Data folder
- Attach/detach database - a database can be attached or detached from any UNC path

Used together, these components make it easier to build a scale-out solution for read-only Analysis Services cubes. For example, you can connect four blade servers to a shared, read-only database on a SAN, and direct SSAS queries to any of the four, thereby improving your total throughput by a factor of four with inexpensive hardware. The best possible query response time remains constrained by the capabilities of an individual server, since only one server runs each query.

Reporting

This section offers best practices for different aspects of reporting such as improving performance.

Best Practices: Data Presentation

Allow IT and business users to create both simple and complex reports

Reports have always fallen between two opinions—should IT people or business people design them? The problem is that business users understand the business context in which the data is used (the meaning of the data) while IT people understand the underlying data structure.

To help business users, in SQL Server 2005, Microsoft introduced the concept of a report model. This was built by IT people for the business users who then wrote reports against it by using a tool called Report Builder. Report Builder will ship in the SQL Server 2008 box and will work as before.

An additional tool, Report Builder, aimed squarely at the power user, has been enhanced in SQL Server 2008. This stand-alone product, complete with an Office System 12 interface, boasts the full layout capabilities of Report Designer and is available as a Web download.

IT professionals were well served in SQL Server 2005 by a tool called Report Designer in Visual Studio, which allowed them to create very detailed reports. An enhanced version of Report Designer ships with SQL Server 2008 and is shown in Figure 6.
Present data in the most accessible way possible

Traditionally, grid data has been presented to users as tables, matrices, and lists. Each has its strengths, which is another way of saying that each has its weaknesses. SQL Server 2008 Reporting Services combines all three into a new data region called a Tablix.

In Figure 7, the table on the left shows percentage growth and the matrix on the right shows the actual figures.

<table>
<thead>
<tr>
<th>Customer</th>
<th>Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retail</td>
<td></td>
</tr>
<tr>
<td>Acme</td>
<td>19%</td>
</tr>
<tr>
<td>Nadir, Inc.</td>
<td>322%</td>
</tr>
<tr>
<td>Wholesale</td>
<td></td>
</tr>
<tr>
<td>ABC Corp.</td>
<td>19%</td>
</tr>
<tr>
<td>XYZ, Ltd.</td>
<td>322%</td>
</tr>
<tr>
<td>Grand Total</td>
<td>56%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>2001</th>
<th>2002</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retail</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acme</td>
<td>1,115</td>
<td>1,331</td>
<td>2,446</td>
</tr>
<tr>
<td>Nadir, Inc.</td>
<td>152</td>
<td>642</td>
<td>794</td>
</tr>
<tr>
<td>Wholesale</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABC Corp.</td>
<td>11,156</td>
<td>13,312</td>
<td>24,468</td>
</tr>
<tr>
<td>XYZ, Ltd.</td>
<td>1,523</td>
<td>6,421</td>
<td>7,944</td>
</tr>
<tr>
<td>Grand Total</td>
<td>13,946</td>
<td>21,706</td>
<td>35,653</td>
</tr>
</tbody>
</table>

Figure 7: A table and a matrix

The Tablix shown in Figure 8 combines these and adds some totals.
The Tablix data region gives users far more control over layout than tables, lists, and matrices. It also enables them to add column groups, specify arbitrary nesting on each axis, optionally omit rows and column headers, and have multiple parallel row/column members at each level.

Present data in reports that can be understood easily

The reason for generating reports is to transform data into information for the business user. Many people find graphics easier to understand than numbers. SQL Server 2008 Reporting Services introduces the gauge, a new form of visual output. A gauge displays a single value from the data. As Figure 9 shows, gauges are particularly useful for creating easy-to-read displays that compare several values.

Figure 8: A Tablix data region combining table and matrix features

Figure 9: Examples of the gauges that can be created in SSRS

Charts have also been extended to include polar, range, and shape. Figure 10 shows only a
A large number of charts are available in SQL Server 2008 Reporting Services. Multiple data series can now be displayed on more than one axis, the user has control over scale breaks on the axis, and there is support for run-time formulae.

Present data to users in familiar environments

Reports can now be rendered as Word documents compatible with Word versions back to and including Word 2000. In addition, the existing Excel renderer has been enhanced and now supports features such as nested data regions, merged cells and subreports.

Best Practices: Performance

Structure your query to return only the level of detail displayed in the report

Use report-level aggregation only for subtotals in the report, not for detail rows. Remember that the most commonly used aggregate, Sum, can be summed in any order and still yield the same result. While other commonly used aggregates cannot, they can often be decomposed into simpler, reusable components.

For example, if you are trying to show an average at several grouping levels and also want subtotals, rather than returning detail rows and aggregating everything in the report, you can decompose the calculation into sums and counts. Then you can reconstitute the average in the report by using this kind of division:

\[ \text{Sum(Fields!Sum.Value)} / \text{Sum(Fields!Count.Value)} \]

thereby avoiding the need to pass detail rows to the report. Let the database do the bulk summarization and aggregation, and use SSRS to assemble and format results for display.

Filter by using parameters that are passed to the query.

msdn.microsoft.com/.../cc719165.aspx
As a general rule it is better to filter in the query rather than in the report (SELECT * FROM xyz WHERE field = @param). However, if you do this you cannot create a historical snapshot of the report without locking down the parameter value. If users need to change the parameter value in the snapshot, the best practice is to return all of the data to the report and filter inside the report instead.

Sort within the query

SSRS utilizes sort-stable operations internally, so the order of data returned from the query is not altered within each group instance, thereby allowing sorting to be performed in the query. However, sorting that is based on aggregate values is far more convenient (and usually more efficient) to perform in the report itself.

Avoid using subreports inside a grouping

Each subreport instance is a separate query execution and a separate processing step for SSRS. For master-detail reports, it is far more efficient to simply join the master and detail data in your query and then group by the master key in the report, except in cases where the number of master records is small (in which case subreport usage is not a performance issue).

There are two cases where subreports may be required:

- When the master and detail data are in different data sources. Pulling the data into a single data warehouse is recommended in this case. If that is not possible, SQL Server linked server or open rowset capabilities should be considered.

- When there are multiple independent sets of detail records for each master record. An example might be displaying a detailed list of both sales and returns for each customer. In this case, drillthrough reports are recommended instead of inline subreports unless the number of master records is small.

Limit the data in charts to what a user can see

While it may be tempting to include a large amount of detail data in charts to improve accuracy, it is better to pre-group the data in either the query or in the report, limiting the number of data points. Drawing hundreds of points in a space that only occupies a few pixels degrades performance and does nothing to enhance the visual appeal of the chart.

Pre-sort and pre-group the data in your query

You can normally improve performance by grouping and sorting the data in the query to match the sort order required by the report.

Use drillthrough rather than drilldown when detail data volumes are large

While the SQL Server 2008 on-demand processing engine optimizes away most calculations for items that are not displayed, keep in mind that all of the data is retrieved for every detail row, even if your initial drilldown state has everything collapsed up to the highest level of aggregation. In addition, all grouping, sorting, and filtering must be performed regardless of visibility or drilldown state. So if the user typically is only interested in seeing a small percentage of the detail data, an associated drillthrough report is a better choice.

Avoid complex expressions in the page header and footer

If the page header or footer contains complex expressions (anything other than a simple field or parameter reference), SSRS must assume it may include a reference to the total number of pages. As a result, the entire report must be paginated before the first page can be rendered. Otherwise, the first page can be rendered and returned to the user immediately who will not have to wait for the entire report to be paginated first.

Turn off CanGrow on textboxes and AutoSize on images if possible
Some renderers are more efficient if the sizes of all objects in the report are known to be fixed.

Do not return columns that you are not going to use from your query

Since all of the data in the query must be retrieved (and stored) by the report server, it is more efficient to return only the columns that will be used in the report. If the result set cannot be controlled (such as when the results are returned from a stored procedure), it is sufficient to remove the field definitions from the dataset. While the extra columns will be retrieved, this will at least prevent them from being stored.

**Best Practices: System Architecture and Performance**

**Keep the report server catalog on the same computer as the report server**

Reporting Services generates and uses a database that is essentially a catalog. This is used during report processing for a number of tasks, including managing the data returned from queries. While it is possible to store this database on a server other than the reporting server, doing so requires that the report data be pushed across the network unnecessarily, which slows down report execution. It is a much better practice to hold the catalog database on the reporting server to avoid this network traffic and the associated delays.

**Consider placing Reporting Services on a different server from your data warehouse**

While pulling the data from the queries across the network does slow things down, it is beneficial to not have your data warehouse and SSRS competing for memory and processing time.

**Conclusion**

Designing and building an enterprise data warehouse can be a major effort with a significant cost. Data warehouses are not built as displays of technical excellence; they are investments made by the enterprise and intended to yield a high return. As both the IDC and Gartner papers show, all too often these projects have more of the characteristics of a gamble than a shrewd investment—a potentially high return but a concomitantly high risk of failure and loss.

A great deal has been learned about factors that are associated with successful projects. These have been distilled into a number of best practices, many of which are described here, encompassing business processes, design, and technical implementation using SQL Server 2008. Indeed, many of the new features in SQL Server 2008 were specifically designed to allow these best practices to be implemented easily.

As a proportion of the total effort required to create a data warehouse, applying these best practices is inexpensive, and yet doing so can have a major impact on the success (and therefore the ROI) of the project as a whole.

For more information:

- SQL Server Web site
- SQL Server TechCenter
- SQL Server Developer Center

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screenshots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screenshots, unclear writing?

This feedback will help us improve the quality of the white papers we release. Send feedback.
References


[Han08] Eric N. Hanson et al., *Scaling up Your Data Warehouse with SQL Server 2008*, work in progress, 2008.


