Data Warehouse Efficiency Techniques with the SAS® System

Tony Brown, SAS Institute Inc., Dallas, TX

Abstract

The successful implementation of a data warehouse relies on several key factors, including the ability to deliver clean, consolidated, and accurate information in the most timely and efficient means possible. Previous SUGI data warehouse papers have discussed the data and construction implementation aspects of data warehousing. This paper intends to address techniques for making the performance of the data warehouse as efficient as possible. Most of the concepts presented are not new, but consolidated from several areas as they apply to data warehousing. This paper is intended for a mid-range to advanced audience, and assumes familiarity with general data warehousing principles, and an advanced level of SAS® programming knowledge by the reader. Topics covered will include subject-based modeling, indexing techniques for tables, techniques to prevent table joins, efficient query retrieval techniques, table load strategies, and data summarization considerations.

This paper should be of interest to both data warehouse practitioners, and users. Several SAS® products will be highlighted, but experience in their use is not necessary to derive benefit from this paper.

Introduction

The key challenge to data warehouse implementation involves the ability to turn data into easily and quickly accessible information. This challenge is precedent by the tasks of obtaining, cleaning, and transforming data into usable repositories. Simply spoken, the goal is to engineer acceptably fast response times for information requests. The challenge of efficiency will be the primary focus of this paper as previous SUGI papers have adequately addressed cleansing and transformation topics.

Unfortunately, there is not room in this presentation to cover all the aspects related to making a data warehouse perform efficiently, so several areas have been chosen among those that are deemed important. They are as follows:

1. The Role of Metadata
2. Granularity of Data Structures
3. Data Modeling Efficiencies
4. Indexing techniques and strategies
5. Efficient query retrieval techniques
6. Eliminating table joins
7. Table loading strategies
8. Performance testing
9. Choosing Summary Data Structures

Important topics not addressed due to the space limitations of this paper are LAN, hardware, and physical disk-level storage implementation issues needed to maximize performance. This is a large set of topics in and of itself, and should be explored with your hardware/LAN technical resource team.

The Role of Metadata

Before beginning data warehouse design activities, an extremely thorough understanding of the users' requirements is necessary. This includes a thorough knowledge and assessment of metadata. Aside from the obvious concerns of data cleanliness, integrity, and integration, the understanding of data value cardinality and distribution patterns are of paramount importance in determining sorting and indexing strategies for final data repositories.

The contribution of metadata in decision making and implementation of performance efficiency through data modeling, table indexing, querying and loading, and data summarization cannot be understated. The SAS /Warehouse Administrator™, while not a topic covered in this paper, provides a rapid facility for automatically capturing critical metadata components necessary for the above activities. In addition other needed metadata can be obtained through SAS® procedures. The metadata required for the various efficiency techniques will be pointed where appropriate, with an indicator of where it can be obtained, in either the SAS /Warehouse Administrator™, or elsewhere in the SAS® System.

Granularity of Data Structures

Purposes of Granular vs. Summary Structures

One of the first decisions involving efficiency in building a data warehouse is determining the lowest levels of data granularity (level of detail) that are required for use. Avoiding unnecessary levels of granularity can drastically improve exploitation performance. Smaller, summarized structures are inherently faster to navigate with user written programs and front-end navigation tools such as the SAS/MDDB™ viewer, and other objects provided in SAS/EIS® software.

Unfortunately, utilizing only summarized structures is seldom practical. Most data warehouse users range from those who perform point and click analysis on interactive tools utilizing summarized structures, to those who require the extremely granular data for detailed reporting or to support analysis models.

In addition, a clean repository of transformed, integrated, and validated data is necessary as the foundation from which to build more efficient summarized structures. Without this granular data repository to fall back on, it is difficult to evaluate the integrity of summarized data when questions or doubts about the data arise. Given this, it is incumbent upon data warehouse administrators to determine how to maximize efficiency and balance it within acceptable cost and storage parameters to achieve their user's business goals.

Data Modeling Efficiencies

One of the most important decisions affecting efficiency is the schema with which you choose to model your data. The subject-based Star schema has proven reliable in many data warehousing efforts to support decision support systems.

Subject Based Data Models

Stars and Snowflakes

In the Star schema, relational tables are used to model the subject of interest (for example sales, shipments, and so on). The schema utilizes implicit modeling of relationships instead of the explicit models of the entity-relation schema. It involves a "Fact" table which holds numeric information about a particular subject (for example Sales volume, profit, ROI, and so on). Because it holds large volumes of numeric factual data about the subject, the Fact table is typically the largest table in a subject-based schema. This Fact table is surrounded by accompanying "Dimension" tables that describe attributes of the "Facts" (for example Sales period, region, state, and so on).

Dimension tables are typically much smaller since they provide descriptive data. In order to process a query against a subject-based schema, the keys of the surrounding Dimension tables required to fulfill the query, are acquired and driven against the Fact table. Figure 1.1 shows a Star schema, with the Fact table, and the accompanying Dimension tables surrounding it in a "star" design.

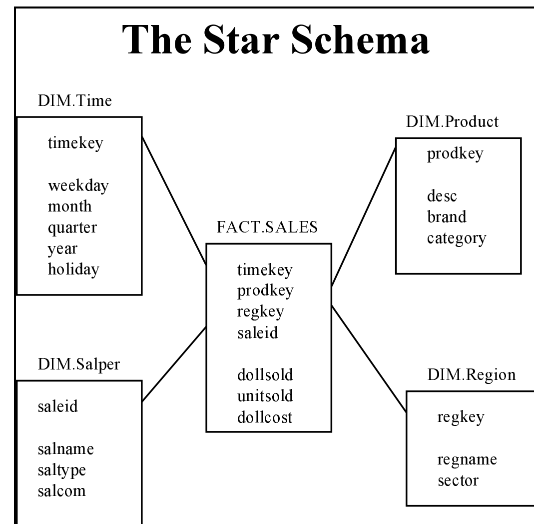

Figure 1.1

While introducing some increased data redundancy in the Dimension tables, the Star schema provides greater flexibility and simplicity in query generation. This is possible because the data redundancy allows the Star schema to utilize fewer tables to model a subject, requiring fewer joins to be made.

Also important in this model, the outer Dimension tables are joined against the central Fact table. SQL views can be created beforehand for efficiency when multiple Dimensions are required to be joined with the Fact table. The entire point here is to make querying fast and easy through simplified models, which aid simplified query construction and reduce join overhead.

Another popular subject-based schema is the Snowflake schema (Figure 1.2). In this example the Member Dimension has been normalized to remove demographic data to the Demog Dimension. This reduces redundancy (cuts down on storage), but at a query construction and performance cost. The Snowflake schema should only be considered viable when data storage requirements are extremely large, and the the lessened query performance and usability are justified by the storage savings. When at all possible, utilize simplified models, like the Star schema, over the decomposed models, like the Snowflake schema to reduce query and table load complexity, and enhance query performance.

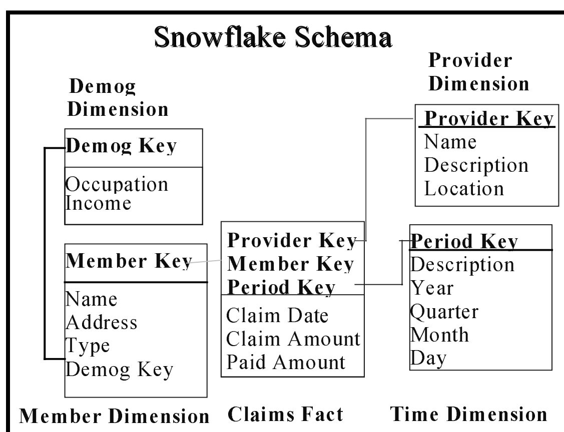

Figure 1.2

Why Not Just Use Single SAS® Data Sets?

Invariably the question pops up, “Why not just use single SAS® data sets with all the data in one observation instead of Fact and Dimension tables?” This situation most often creates far too much redundant data for a data warehouse of any but the smallest size (think about having to store the Provider Name in the figures above for every Claims Fact). The Star schema represents a “happy medium” to allow reasonably good query performance, sharing of Dimension tables across subjects, and elimination of some redundant data.

Keeping the Big Picture In Mind

When designing a subject-based model, careful attention must be paid to how queries will be used across multiple subjects. This requires the initial creation of an enterprise level model first. In the SAS® Rapid Warehouse Methodology, the identification of all identifiable potential subjects is made, the subjects are prioritized, and then a set of subjects is chosen for initial implementation. By having knowledge of the subjects comprising the enterprise, reasonable information can be gathered as to how the subjects will interact, or how queries will potentially cross subject areas. This allows a more holistic subject model design to facilitate sharing of Dimension or Fact data across multiple subjects in queries. This affects table key strategies, and decisions on how changing Dimensions are historically tracked, granularity of subject content, etc.

A Word About Table Keys

An important consideration in designing the Fact and Dimension tables in the Star schema is the selection of table keys to relate the Dimension tables to the Fact table. There several primary considerations when selecting keys. They should be unique and not recyclable (e.g. not reused at a later time to request some other item or record), and they should be as short as possible. Long alphanumeric key values should be considered for replacement by short numeric keys, with formats to provide key descriptions. For keys that could possibly be recycled (e.g. universal product codes (UPC) are utilized differently or duplicated by different manufacturers), a couple of considerations are in order:

1. As just mentioned, key re-mapping should be considered to maintain uniqueness. Mapping to a short numeric value should be considered, using formats for descriptions if necessary. This will save search time on extremely large data sets.
2. For keys that do not change actual key value, but have changing descriptions, the key could be given a suffix to track the values historically. This effectively creates a composite key, allowing the first part of the key to be used for tracking all generic

occurrences of the code. The second part can be used to historically track the changed code description from it's inception. The description here would reside in the Dimension table in multiple records.

The above implementations will help keep key maintenance to a minimum, preventing major re-work of the warehouse. It will also provide flexibility for those hazardous keys that get recycled, or change in descriptive value yet still need to be historically tracked.

Indexing Techniques and Strategies

One of the primary determinants of query performance against granular level data is how effectively indexes are used to aid in data location and retrieval. Indexing is not a simple topic with easy heuristics, but there are some overall things to note. One of the first is to understand how the SAS® System uses indexes. Performance can be degraded if you are not careful. For additional information on indexing with the SAS® System indexes please see the list of manuals and the additional references listed at the end of this paper. The following paragraphs will deal with general knowledge about indexing with the SAS® System, followed with a discussion on more strategic topics for use in data warehousing.

General Indexing Information

The SAS® System utilizes a Btree indexing system. The goal of the index is to reduce physical I/Os in a read. These I/Os are measured as the number of pages that must be read.

A page is the unit of transfer between the supervisor and the host code. On any given host, a page could be mapped into one or more data blocks.

When an index is used to read a file, records are fetched in ascending value order. Unless the file is sorted by the index value, pages are going to be read in physical order, possibly only accessing a sparse number of records on each page (records containing the indexed value searched for).

Some pages may have to be re-read because they may contain subsequent records that have index values that are not directly sort ordered behind the previous index records read on that page. This contributes to page "thrashing", which is very I/O inefficient. This points out that indexes will perform better when the data being selected are clustered together in the data set, or are sorted in the key value order demanded by the index, or the subset retrieved is very small.

Experience shows an extract of 10 to 15 percent or less of the data works best, with performance gains often taking place on extracts up to about 30 percent of the data. These figures are only rough ranges and will be influenced by many factors as will be made apparent shortly. Two of the most important factors in determining whether an index can be useful in WHERE processing are *discrimination* and *distribution* of the data values of the selected index variable.

Discrimination is defined as the number of unique values the indexed variable can have. For example the variable Gender would not be very good discriminator, because it only (hopefully) has two values, male and female, and would return a large subset of records. Last Name would be a better discriminator because it has many different values and a search would return a much smaller subset of records. So it is easy to see how discrimination contributes to index performance.

Even if data values are very discriminatory, they may be not be distributed appropriately for an effective index read. *Distribution* refers to how the index variable is distributed throughout the data set or table, typically either uniformly or normally. Data values for a variable that are uniformly distributed tend to work well because of the algorithm the SAS® System uses to determine the cost of using the index to perform a WHERE expression.

In this process, the SAS® System estimates whether a sequential pass through the data set would be more cost efficient than an index. These cost measurements are made using an algorithm that estimates the number of observations to retrieve the selected index values.

Also, in this estimation process, the SAS® System assumes the data values are uniformly distributed through the data set. If the data values are normally distributed rather than uniformly distributed, the SAS® System may

make the wrong choice of whether to use or not use an index.

How does this happen? If index values are normally distributed, and the selection criteria for the variable values are on either end of a normal distribution (e.g. WHERE AGE >75), then the SAS® System may overestimate how many observations it will take to satisfy the WHERE and choose a sequential read over the cheaper (in this case) indexed read. On the converse, selecting index values within one standard deviation of the mean may cause the SAS® System to underestimate the number of observations involved, and select to use the index over a cheaper sequential read.

To determine how discriminatory a variable is , it is good to use the UNIVARIATE or FREQ procedure for numeric data, and the FREQ procedure for character data. To determine how data are distributed through the table one can use the GPLOT procedure.

In addition, an index on a variable specified in a correlated reference to an outer table usually improves a subquery's (and ultimately, the query's) performance.

Composite indexes can improve the performance of queries that compare the variables named in the composite index with constant values that are linked using the AND operator, the IN operator, and fully bounded range conditions. This is especially useful for Fact table queries, since composite indexes on the table foreign keys can aid query performance. In order for *compound optimization to work*, there are several considerations to keep in mind. They are:

- Starting at the left side of the index description, at least the first two variables must be used in a WHERE condition.

- The variables must be used in a range condition and/or an EQ or IN operator comparison. At least one of the matching WHERE conditions must be the EQ or IN operator; you can't use all range conditions.

- All boolean conditions must use the AND operator; the OR condition will exclude index usage.

- A NOMISS composite index cannot be optimized on a WHERE expression that would qualify all missing values.

- No functions can be optimized in a compound query.

- The use of X>Y comparisons will exclude index usage.

What to Index

Since indexes take up storage space (sometimes as much or more than the data set itself), require machine resources, and affect table build times, we must use them judiciously. The key is to get the most performance for the storage/index build cost trade-off.

The obvious first things to index in the granular level Star schema, are the Dimension and Fact table keys. This will facilitate table loading performance as well as table joins by the key values.

The Dimension table keys often make good candidates for simple indexes. The Fact table keys relating the Fact tables to the Dimension table are an excellent candidate for composite indexes so that compound optimization can be taken advantage of in WHERE processing. Using an example from Figure 1.1 above, suppose the fact.sales table is using the timekey, regkey, prodkey, and saleid related from the Dimension tables. So the resulting keys for the fact.sales table unique record identification (and hence the index itself) are timekey, regkey, prodkey, and salesid. A composite index on all of the keys could be composed of those keys in that particular order in the index description.

Perhaps the most important key value in that index is the Time key element. It is the key element most used in a decision support data warehouse. The Fact table is almost always sorted by the Time key for chronological appending, so it's use as a leftmost, compound index variable is made more efficient, since in a high transaction environment it is not a very *discriminatory* variable.

After the Time key element, the other key elements should be ordered to support the most frequent queries, keeping in mind the rules for

compound optimization. If the other keys tend to fall into a natural hierarchical relationship for the way most user querying is done, this works well. In fact, in this situation, if space allows, other shorter combinations of this index can be built to allow smaller, faster indexes to be chosen by the system for usage, eliminating the rightmost variables that are taking up space and making the compound index larger. A list of this "cascade" of shorter indexes would be:

- timekey, regkey, prodkey
- timekey, regkey
- timekey

If the following are true, simple indexes might prove more efficient than compound indexes:

- If each of these variables has many distinct values and are used in single condition, highly selective lookups

- Too many combinations of the variables are used frequently not in the left to right order they sit in the index description

Set the MSGLEVEL=I option on for confirmation in the log when an index is actually used in an operation.

The indexes need to be tested for value and to see if they are worth the cost. Remember to re-read the rules on indexing and test several scenarios before settling on a final strategy.

Once a usable indexing strategy is devised and implemented, it should be monitored and tested over time for continued viability and storage cost justification.

Efficient Query Retrieval Techniques

One of the many advantages of using the SAS® System as a data warehouse repository is that both SQL and non-SQL users can access and use the data. While SQL provides some advantages in querying (data sets do not have to be sorted in desired order, less code to write joining multiple tables, etc.), SAS® data step and procedure code contain their own advantages. Data step code can be more CPU and I/O efficient than SQL. In addition, data step and procedure code have been mastered by a wider range of SAS® users than SQL, and provide a good alternative for batch and interactive programming. SQL code while frequently more compact, can prove to be challenging to write for novices. The good news here is that both are applicable. Since this paper assumes that the reader is well versed in SAS® data step and procedure coding techniques and efficiencies, the remainder of this topic will be devoted to some general SQL efficiencies. Some ways to make SQL queries more efficient include:

- For those who like to write subqueries, or nested queries, it is usually true that a join is more efficient to process than a subquery. In fact, the PROC SQL query optimizer may change a subquery to a join for you if it deems it will be more efficient and the EXISTS or NOT EXISTS conditions are not present. In addition, in a join, the SQL procedure stores the result values for each unique set of correlation variables temporarily, eliminating the need to calculate the subquery more than once.

- When using set operators like UNION, OUTER, EXCEPT, and INTERSECT to link tables together, use the optional ALL keyword to eliminate significant overhead when you know there are no duplicate rows in your result table.

- Many users create SQL queries and then use them as templates to construct other queries. It is wise to pay attention to what may be in the query creating overhead you may not need. A good example is eliminating the ORDER BY statement on large queries that do not require data to be in a particular sorted order for the table or view's output. Another example may be SQL options left on like STIMER which contribute to overhead.

- The SQL procedure does not optimize between two queries or SQL statements. So if a problem involves several steps requiring temporary tables to be subsequently used to hold intermediate results, combining the queries into one query using in-line views is more efficient once the problem stages are worked out.

Eliminating Table Joins

Since table joins compose the primary expense in many queries, it is wise to avoid them wherever possible. There are some techniques to accomplish this with both simple and complex join scenarios. For some joins a technique of using formats to obtain values needed from other tables can be highly instrumental in avoiding table joins, and their resulting Cartesian products.

The concept is quite simple. By creating a format containing the descriptive values of the desired key variable in the Fact table, an SQL statement can be written to use the format to get the key variable value for the query instead of having to actually join the Dimension table. Let's consider an example. Using Figure 1.1 again, if we wish to query the Sales Fact table and obtain the value for the product description corresponding to the prodkey in the fact.sales table, we could write an SQL query that looks like this:

```
proc sql;
title 'January Product Sales';
select product.desc, sales.dollsold,
        sales.unitsold
from fact.sales, dim.product
where
       /** join criteria **/
       sales.prodkey=dim.prodkey
and
      /** subsetting criteria **/
   '01jan97'd<=fact.time_key<='31dec97'd;
quit;
```

We can use the CNTLIN option of the FORMAT procedure to create a format from the Product Dimension:

```
proc sql;
create view makefmt as
         select desc as label,
                  prodkey as start,
                 'product' as fmtname
         from dim.product;
quit;
proc format cntlin=makefmt lib=dim;
run;
```

Once the Product value format is created, the original SQL query no longer needs to join the Product table to the Fact table to get the Product value. The query code now looks like this:

```
proc sql;
title 'January Product Sales';
select put (prodkey, Product.) as product,
         sales.dollsold,
         sales.unitsold
from fact.sales,
where
       /** subsetting criteria **/
    '01jan97'd<=fact.time_key<='31dec97'd
;
quit;
```

This format conversion technique can also be used in the WHERE statement to subset data. Many simple joins requiring a single descriptive or element value associated with a Dimension table key can be prevented using formats. However there is an associated cost of the format maintenance and storage, and processing overhead of the format usage in the query. This often tends to be much smaller than the associated table join cost on large tables. Another benefit is that format creation and maintenance is usually performed in the waerhouse load/update process. The cardinality and length of formatted values may slightly affect the formats performance. Some other things to watch out for include:

- Formats return only a single descriptive value, so this technique works best when a single value is needed.
- The format must be able to be held in memory.
- If the Dimension tables are sorted and/or indexed by the needed descriptive variable, the use of formats may not be as effective in terms of maintenance and execution cost.
- In some cases the format may be very inefficient if it is large, and is not sorted internally by the unformatted values (requiring longer search time for the unformatted value).

On smaller tables it is recommended to test the most likely join scenarios with and without formatting for simple joins to determine if the query savings warrants their overhead. This technique usually works best when single descriptive values are sought from variables in

the Dimension that indexes are not stored for due to cost.

For more complex joins requiring subsetting, a subset method using compound indexes may prevent table joins. Using Figure 1.1 again, suppose there is a compound index on regkey, prodkey, and salesid_key. An SQL query to select the sales for the Southwest Region, for products with the Brand of 'ACME', and salespersons who are "Franchise" salespersons, a fairly complex query would have to be written joining the Region, Product, and SalesID tables to the fact.sales table. For example:

```
proc sql;
select sales.dollsold, sales.unitsold
from  fact.sales, dim.product, dim.region,
          dim.salper
where
sales.prodkey=product.prodkey and
sales.regkey=region.regkey and
sales.saleid=salper.saleid and
brand='ACME' and
region='Southwest' and
saltype='Franchise' and
'01jan97'd<=fact.time_key<='31dec97'd;
quit;
```

Another alternative is to place the subset criteria into a subquery. If this does not improve the efficiency enough, and the list of subset values is small, try the following technique.

One can submit initial queries to determine the values of the Dimension keys for the requested variable values (e.g. all the Product ID values where the brand is "ACME"):

```
proc sql;
select prodkey
from dim.product
where brand='ACME';
quit;
```

```
Output=     SalesId
            21423
            23421
             43342
             23423
```

By using those collected values an IN expression in the final queries WHERE statement, a join can be avoided.

```
 proc sql;
select sales.dollsold,
        sales.unitsold
from
        fact.sales
where
sales.prodkey in (21423, 23421, 43342,
                        23423)
and  /** Use same technique for Region **/
and  /** Use same technique for Saltyp  **/
and
 '01jan97'd<=fact.time_key<='31dec97'd;
quit;
```

The above technique is quickly adaptable for a limited number of values returned by the initial queries to construct the IN expression for the final query. For voluminous query values resulting from the initial query to support the IN expression, the method becomes more problematic. Also, if the two step query is automated, macro code must be used to pass the values for the IN statement.

Table Loading Strategies

The maintenance loading of the data warehouse tables can be extremely complex and lengthy, depending on the volumes of data utilized, the complexity of the extraction, cleansing, transformation and load code, and subsequent indexing, formatting, and summarization that needs to take place on the data. Batch windows on many decision support systems are very short. This necessitates a very streamlined, and efficient load strategy for the warehouse.

The two obvious choices are full replacement of warehouse tables, and incremental update of only new or changed data.

When data warehouse environments encounter the pristine situation of only having to append new data every update period, incremental updating of the table using the APPEND procedure is usually the best method. Since the Fact data are usually stored in Time key order, having a simple index on the Time key, or using the Time key variable as the leftmost index variable in a compound index for the Fact table lends itself to fairly efficient and easy loading.

Not all situations are that easy. Some complex situations arise where data must be initially loaded into a data warehouse, then altered within a short time to replace missing or incorrect data. (YES, the customer said it had to be that way!) These situations usually result when the Fact data are fairly accurate and need to be entered ASAP to support month-end reporting. Then the records are either deleted, and re-entered, or adjudicated. This creates more complex load patterns and considerations.

The simple append will no longer work here, and total table replacement, single record corrections or table inserts are required depending on the data circumstances. The indexing and table sort strategies to support such loading can often be different than what is desired for query performance.

For tables that require updates of records in existing tables, there are many good examples of applying updates and combining observations into existing tables in the SAS Institute Inc. Publication "Combining and Modifying SAS Data Sets: Examples, Version 6, First Edition", publication number 55219.

Full replacement of tables should only be considered when table sizes are relatively small, and/or when there is a high amount of changed data from the previous update period, and historical data that are not changed or altered can be readily accessed for integration to the changed data.

Performance testing

Once a data model has been chosen, the table key strategies implemented, and an indexing system put into place, extremely thorough query performance and table load testing is required.

Sample loads over several update cycles should be tested to determine accuracy and timeliness of the update process, and it's fit into the available batch window. This testing should be used to determine if the data model and indexing structure is sufficiently designed to allow acceptable load performance.

Query performance testing against the granular level data is also required, using an array of query and data extraction activity from the end-user repertoire. Based upon the user requirements and analysis priorities, the table indexes should provide performance improvement for the most used query variables and table join types. These indexes will need to be individually tested, first running queries without indexes, with simple indexes, and with various combinations of compound indexes. As user needs change, indexing strategies and even model design will have to change to provide efficiency to changing query priorities. It is often the case that altering models or indexes to aid in some operations may hinder others, and a trade-off decision will have to be reached.

Choosing Summary Data Structures

Summary Structure Architectures

Summary structures are an extremely efficient means of providing information. The typical structures are SAS® data sets built with the SUMMARY procedure, and the SAS/MDDB™ Server multi-Dimensional database. Since reader familiarity with the Summary procedure is assumed, a brief discussion will follow concerning the SAS/MDDB™ Server.

The following items are general rules that should be noted in the decisions to store in summary SAS® data sets or the SASMDDB™/ Server:

When it is usually more efficient to use SAS/MDDB™ Servers:

- User exploitation requirements can be met with online EIS viewer objects.
- Need for multiple subtables to yield fast exploitation performance outweighs storage and build cost.
- Summary files are not extremely large, and crossings higher than NWAY are desired.
- Need to incrementally add records to structure.
- Summary table variables and variables do not change often (metadata does not change often), forcing the entire structure to be rebuilt.

When it is usually more efficient to use Summary SAS® data sets:

- User exploitation requirements cannot be met with SAS/EIS® viewer objects alone.
- Table load times are more critical than the benefit having multiple subtables with higher than NWAY crossings.
- Summary files are very large.
- Frequent changes in summary statistics and analysis and class variables to the Summary file.

Summary

Building performance efficiency into the data warehouse is the result of applying a thorough understanding of user requirements in the data modeling, table indexing, query retrieval, and data summarization activities in warehouse construction. The usage of subject based modeling, and new tools such as the SAS/Warehouse Administrator™, and the SAS/MDDB™ Server, along with the skills that are inherent to users of the SAS® System, makes the task of data warehouse efficiency extremely manageable.

References

Kimball, Ralph, *The Data Warehouse Toolkit*, New York: Wiley Computer Publishing, 1996

SAS Institute Inc., “SAS® Guide to the SQL Procedure, Version 6, First Edition”, order number 56070. Available through SAS Institute’s Book Sales

SAS Institute Inc., “SAS® Language, Version 6, First Edition”, order number 56076. Available through SAS Institute’s Book Sales

SAS Institute Inc., “Combining and Modifying SAS Data Sets: Examples, Version 6, First Edition”, order number 55219. Available through SAS Institute’s Book Sales

SAS Institute Inc., “Implementing a Data Warehouse With the SAS® System”, Cary NC: SUGI 22 Proceedings

SAS Institute Inc., “Effective Use of Indexes in the SAS® System”, Cary NC: Steve Beatrous and Karen Armstrong

SAS Institute Inc., “Rapid Data Warehousing with the SAS® System”, Internal Publication

The following SAS® System products discussed in this paper are registered or carry trademarks: will include, base SAS® software, SAS/EIS®, SAS/MDDB Server™, and the SAS/Warehouse Administrator™ product

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration

Acknowledgments

The author wishes to thank the reviewers and editors of this paper; Donna Martinez, Kevin Myers, Jim Craig, and Becky Webb of SAS Institute Inc., and the many SAS Institute Inc. product development and training personnel who provide general dissemination of technical information.

For Additional Information:

Tony Brown
Consulting Services Division
SAS Institute Inc., Dallas, TX
214/977-3916
SASABR@unx.sas.com