Paper 292-2008

Managing large Data with SAS SPD Server®

Daniel Sargent, SAS Institute, Cary, NC

ABSTRACT

This paper provides the concepts behind demonstrations of how you can enhance query performance when you use the SAS SPD Server to manage large data tables. This paper does not cover the main concepts for Dynamic Clusters or Parallel Join. See white papers TW9593 and TW9594 for thorough discussions of the Dynamic Cluster and Parallel Join features. This paper focuses on managing data within standard SAS® Scalable Performance Data Server® (SPD Server) tables, and therefore Dynamic Cluster tables, to enhance data querying. There are three common types of queries for which data within a table can be optimized: ordered processing, subsetting, and a hybrid of ordered processing and subsetting. This paper presents two different ways of managing data at load time for a table that will enhance the performance of queries. One way focuses on ordered processing and one way focuses on subsetting of rows. The third hybrid situation that is a combination of both will be discussed.

INTRODUCTION

The SAS® Scalable Performance Data Server® (SPD Server) is designed to meet the storage and performance needs for processing large amounts of SAS data. As the size of data increases, the demands for processing that data quickly increase, and the storage architecture must fit the business needs. SPD Server is used at hundreds of sites worldwide and hosts some of the largest known SAS data warehouses. SAS SPD Server provides high-performance SAS data storage to customers in banking, credit-card, insurance, telecommunications, healthcare, pharmaceutical, transportation, and brokerage industries; and to government agencies.

SERVER HARDWARE

The SAS SPD Server requires various tasks, processing, and I/O to occur simultaneously. The scalability of the hardware plays a critical role in the overall performance of the server in answering queries. The demonstrations cover the behavior of the SAS SPD Server on a UNIX server that has 8 gigabytes of RAM and four processors. The server has seven independent file systems: one file system is used for indexes and metadata, one for temporary utility files, and five for actual data (rows of the tables). No file system that is attached to the machine shares a disk drive with any other file system. The file systems are set up using a 65536-byte stripe size for writing data.

Data tables

Data tables will be generated to demonstrate how you manage large tables for optimizing query performance using the SAS SPD server. Depending on the purpose of the demonstration, the organization of the rows within the table will change. For all demonstrations, the record layout that is used for the tables used remains the same. The SAS programs used to generate the data tables for this paper are very similar to the sample job shipped with the SPD Server 4.4 or higher media called: Dynamic_cluster_example2_job1.sas. The data tables simulate possible situations that involve data management for subsetting, and they are ordered by processing that SAS SPD Server customers encounter in the field.

Loading of the tables and organization of the data for this paper focus on three columns in the tables. The first two columns, BILLMTH & REGION, are columns that will be used to demonstrate optimizing the data for subsetting. The record key for the table, which has a unique value for every row, is the combination of the columns BILLMTH and CARDHOLDER. Each row in the table can be uniquely identified by BILLMTH and CARDHOLDER. In the tables, every row is considered a summary record for all transactions from a CARDHOLDER during a given month. Although each unique row in the table is regarded as a summary record, there is no additional column that represents the number of transaction or detail rows from which the summary row was created. To be more specific, the column BILLMTH is a date column in which the values contained therein are the first day of each calendar month. A table that represents the calendar year of 2004 will contain 12 dates that all begin on the first day of each month in the year 2004. The value for the month of January will be **`01JAN2004`**, for February the value in the column will be **`01FEB2004`**, continuing through **`01DEC2004`**. The column REGION has four specific character values (**`N`**, **`S`**, **`E`**, **`W`**) representing north, south, east, and west regions.

The column CARDHOLDER contains a set of unique values where each value represents an individual CARDHOLDER. If a data table is generated that contains 12 months of data, then it will contain 12 rows for each unique CARDHOLDER. For example take the CARDHOLDER number 123456, a table with 12 months of data will contain 12 rows for CARDHOLDER 123456, with one row for each month. Because the table is for demonstration, the data will not mimic real customer cases. For example, every month in each table will contain a row for each CARDHOLDER, even though in a real customer situation there might often be months where a CARDHOLDER will not have any transactions and therefore would not have a summary record. Also in the real world, new CARDHOLDERs and old CARDHOLDERs are issued and dropped every month. The two columns BILLMTH and CARDHOLDER will be used to demonstrate the performance of sorting and will also be used for joins or ordered by processing.

The size of the tables that are used for the demonstrations is adjusted for the type of demonstration and the hardware being used. The column REGION within the tables has been converted to character from a numeric format and the number of regions has been reduced to four. A fourth change to the program includes, for each data table, a column that generates a random number that is stored in a column called RANDOM_DIST. This column will be used to randomly order the tables to show the effects that unordered tables can have on performance.

HOW TABLE ROWS ARE WRITTEN TO DISK

An important factor that influences the performance of queries is the organization of the data within the table. The goal of this paper is to enable the reader to better understand how to organize data within a table to optimize the performance for two types of queries. The first type of query is a subsetting query and the second type is one that requires ordered processing of rows. The organization of the rows in the table affects the amount of work that is required to satisfy different types of query requests. You therefore need to understand the structure of data partitions and file systems.

The first item that you need to understand is the table component called a data partition file or dpf. A data partition is a file structure that stores rows for a table. Depending on the size of the table there can be one to many data partitions. Within the data partition file, there are two additional structures. The first additional structure is an I/O block; this structure is used to buffer rows for compression, and it enables direct access to rows in a compressed table. The I/O block is the structure within a data partition that is compressed. The uncompressed size of an I/O Block is around 32k bytes, depending on the width of the rows. For compressed tables, the I/O block contains the second structure (the table rows).

For uncompressed tables, the unit of data that is passed to the OS for writing is controlled by a macro variable called NETPACKSIZE. NETPACKSIZE is mainly used to control the amount of data that gets transferred from the client to the server and vice versa. Like the I/O block, the NETPACKSIZE is around 32k, depending on the width of the row. The I/O block and a data block that is approximately the size of the NETPACKSIZE setting are passed to the OS for writing. When reading rows of a table, these are the units that are read. In this paper we refer to these units of I/O blocks or NETPACKSIZE of data as a READ UNIT. This will be helpful later when we discuss data optimization.

The second item that needs to be better understood is the file system to which the data will be written. Each file system typically consists of a set of disks that are combined to provide a hardware structure that has redundancy in case of a disk failure. This redundancy is called a Redundant Array of Independent Disks (RAID). The disk structure contains a sub-structure that is called a disk stripe, which is a section of disk on which data can be written. For most sites that use the SAS SPD Server, the disk stripe size will be 64k or higher. It is to the disk stripes that the I/O blocks are physically written. The file systems on the test server are set up with a stripe size of 128k, which means that when I/O blocks for a data table are written to disk, each stripe can contain four I/O blocks.

For a more complete discussion about data partitions, please see the white paper “Partitioning of SAS® Scalable Performance Data Server® Tables.”

MANAGING DATA INTRODUCTION

There are two primary data management concepts presented in this section. While these two concepts do not cover every type of possible data management concept, they are the two most important concepts because they can improve performance for a large majority of queries. The first primary concept is optimizing data for ordered processing. Ordered processing includes any type of table-processing step that requires the data to be in a specific order to complete the task.

This can include processing of a table within SAS code, such as a DATA step with BY clause or SQL Joins between two tables.

The second primary concept is optimizing data for subsetting based on values in columns. This type of processing occurs for WHERE clauses that are used to select subsets from a data table. There is a third type of processing situation, which is a cross of ordered and subsetting, or vice versa. These situations are cases where the processing requires both subsetting and ordering of the data to occur to complete the task. A later section will discuss combining the two concepts of optimizing data for both subsetting and ordered processing.

Benefit of Ordering Column Values

Tables are processed in different ways to satisfy the queries from users. Very often a large percentage of the queries will follow a similar pattern for reading and processing a data table that requires identical ordering of the data of the table for all of the queries. One way to enhance the performance of the queries is to load the data into the table in the order that the queries need for ordered processing. This section deals with these cases in which data in tables needs to be ordered and the ways that a table can be managed to enhance the performance of the queries. Three examples of common types of processing in which ordering data is necessary for tables to be processes are joining to another table, processing of a DATA step, and summarizing a table.

It is a common practice for customer sites to pre-order a data table when many queries like the ones above will require ordered reading of the data within the table. This section uses the PROC SORT procedure on a single table to provide a demonstration of the cost/benefit of pre-ordering an example data table for ordered read access of the CARDHOLDER column as described earlier in this paper. The run time and CPU utilization will be collected during the processing of the table to demonstrate the benefits that can be achieved when the CARDHOLDER column is pre-ordered within the table.

Three data tables were generated, each containing an identical set of 134,217,732 million rows and a table width of 256 bytes. To demonstrate the benefit of how per-ordering of a column affects performance, the order of values in either the CARDHOLDER column or the SORTEDBY table attribute, or both, differs for each table. The SORTEDBY table attribute is a set of metadata that is stored within the data table. When the table is being read, this attribute allows the software to determine that the table is already correctly ordered, and to proceed to the phase that directly reads the table, without having to physically order the data.

Sometimes pre-ordering a table cannot be optimally achieved. It might be impossible to ideally optimize a data table because the size of a data table or the need to subset a data table could prohibit the ability to optimally organize the data. If, however, a partial ordering of a table can be done, there can still be significant benefits. This second table provides the benchmark between optimal and worst case.

The first of the three tables provides optimal performance for a query that requires the CARDHOLDER column to be ordered. For this table the CARDHOLDER column is pre-ordered and the SORTEDBY table attributes have been set in the metadata. By pre-ordering the column and having the SORTEDBY attributes set when the table is opened for an ordered read by CARDHOLDER, the software will check the metadata, determine that the data table is already ordered, determine that the order complies with the order requested by the query, and perform a sequential read of the table. The physical process of sorting the data table will be by passed. Sorting a table can be costly. Eliminating this process can greatly improve performance.

The main consideration for loading a table in pre-sorted order is how often that table requires ordered processing compared to other types of processing. Our demonstration table that uses the CARDHOLDER column is a good example of a table column for pre-ordered loading. If this data table is likely to require ordered processing 250 times during a given week, then pre-ordering the CARDHOLDER column and using the SORTEDBY metadata attributes will allow all 250 jobs that require ordered processing by CARDHOLDER to simply read the table sequentially.

When making a choice to manage data in one way or another, you should perform a cost/benefits analysis between possible options. The second table is loaded in a very similar fashion to the first table: the CARDHOLDER column is ordered in the manner that the job will need to read the data table, but the SORTEDBY table attributes are not used. This variation allows us

to measure the cost of not having the metadata set. For this case the data table remains ordered, but the software is not provided that information, which causes the software to behave as if the table was not ordered. Using this second table and capturing the run time and CPU used for ordering the table provides a best-case situation for sorting a table. When a sort occurs on a data table that is already ordered or nearly ordered in the manner it needs to be, much less work is required than if the table had not been ordered. The type of work by passed includes moving data within memory, and the number of compares required to order the data.

The third table allows us to measure the cost of ordering a table when the rows are not already in correct order. This case is less desirable because more work must be performed—moving data and doing compares of the key when ordering the data. This third table contains the same data as the other two tables, but the values in the CARDHOLDER column are randomly distributed throughout the table. The three tables are all created with a column that contains a generated random number. After an initial load of the third table, the column with the random numbers was used to reorder the table randomly and make the CARDHOLDER column randomly distributed.

The test job for measuring the cost of processing the tables is the PROC SORT procedure, which uses the metadata of the table to determine if the table is already ordered. If the table is not ordered, the PROC SORT procedure performs a sort of the table. The general syntax of the code is shown below:

```
PROC SORT  Data=bigdata.table_in Out=bigdata.table_out ;
  By CARDHOLDER ;
Run ;
```

The table below shows the runtime, the average CPU percentage used during the process, and the total CPU utilization factor that was computed using the runtime and average CPU used to process the table. The total CPU utilization factor is a computation that allows the comparison of CPU used between the three jobs. It is computed by taking the average percentage of CPU used and dividing it by 25, which is the percentage of total processing power that each CPU has on the server. The number computed is then multiplied by the runtime in seconds. For example, the average CPU utilization of 66.53 divided by 25 is 2.66, which indicates that an average of 2.66 CPUs was used during the running.

Table processed	Runtime	Average CPU utilization	Total CPU Utilization factor (computed)
First Table (ideally optimized)	N/A	66.53 percent	153,020 units
Second Table	76%	82.20 percent	332,100 units
Third Table (Less desirable)	219%	80.82 percent	596,480 units

Looking at the table above, we can see the importance of managing the order of values in a column. Comparing the performance of reading the first table to the second table shows the importance of using the SORTEDBY table attribute when possible. Comparing the sort performance of the second table to the third table demonstrates how partial ordering of a table can impact performance when sorts are required.

Of the three cases, the best processing time for reading the table was with the first table. This was expected because the PROC SORT procedure detected that the table was already ordered and performed a sequential read of the table. The second best was the second table, which was also expected because the data was organized internally to improve sort performance. This second table shows the best case possible for partial ordering of a table. The sorting of the second table compared to the first table demonstrates the minimum cost of performing a sort on a table. The runtimes and CPU utilization to process the third table shows the worst case for sorting a table. When you compare the run time and CPU with the second table, it becomes clear how even partial ordering of a column such as CARDHOLDER can impact performance.

The numbers presented for runtimes are the percentage increases of time needed to complete the sort compared, to the idealized optimized and are not the actual runtimes used to sort the tables. The numbers used for the average CPU utilization are actual percentages of CPU utilized during the sort of the tables. The numbers can vary depending on many factors that might not be reproducible on other servers. The numbers are dependent on the data used, I/O configuration, and the CPU speed of the server.

MANAGING DATA FOR SUBSETTING QUERIES

We saw in the last section that the data in a table can be optimized for ordered processing. The same can be true for subsetting rows from a table. This section discusses and demonstrates how to manage a data table for optimized subsetting queries, with the goal of reducing runtime and CPU resources used.

The first section will measure four cases for subsetting rows from four data tables with identical sets of rows, but which are organized differently. For this first case the query will select all rows from a table based on a single value in a table column. The column used is the REGION column and the value is **N** for north, which will select approximately 25 percent of the rows. The first query uses a method called a full-table scan to subset the table. This full-table scan will provide a baseline from which to measure different cases. It is a baseline because it is the default method when no other method is possible. For a full-table scan, the subsetting of rows from a table is accomplished by having the software read all rows in the table and manually checking them with the sub-setting criteria. No indexes are used for this type of check, and the organization of the rows does not make any difference in the cost to subset the rows.

A second method that you can use to subset rows from a table is an indexed approach, in which the software reads as little as possible of the table and processes only the rows that meet the selection criteria. In this case, the organization of the data in the table can affect performance of subsetting rows, and two tables are used. One of the indexed tables will not be organized for subsetting. The values for the region column are dispersed throughout data table. For the second data table the values are organized for subsetting. All rows for each value in the REGION column are adjacent to each other. This reduces the number or READ UNITS that the rows to be selected will be on. A fourth case is included where the query is run against a cluster table that uses the MINMAXVARLIST option on the REGION column. The cluster table is a data-management feature that can also be used to organize a data table for optimal subsetting. The cluster table has four members where each member contains all of the rows for a specific region value. For example one cluster member will contain the rows for the N or north region.

Using an index for subsetting is not always better then full-table scans. There are several reasons why subsetting with an index may not be more effective. The first reason is the number of rows. Using an index is not cost free: the index must be read, a list of rows to select must be built, and then the rows must be read from the table. The total cost of building the list of rows to select from the index and then using that list to read the rows from the table increases as the number of rows increases. There is more index work to create a longer list of rows and more rows to read. The second reason is that the data might be poorly organized for using an index to select the rows. Remember that the amount of data that must be read from the disk is approximately the physical size of an I/O block or a NETPACKSIZE. We called this amount of data a READ UNIT. Consider this simplified example: a table has 100 million rows, and it has 1 million READ UNITS. If the subsetting criteria were to select about 1 million rows from the table, and each READ UNIT contained at least one row that matched the subsetting criteria, then every READ UNIT and the entire table would have to be physically read.

That simple example provides an important insight to organizing a data table for optimized subsetting. Our example has 100 million rows and 1 million READ UNITS from which we can calculate that each READ UNIT contains 100 rows. It also means we can calculate the minimum possible number of READ UNITS that are needed to store the 1 million rows that a query subsets. If the rows are organized to fit on the minimum number of possible READ UNITS and an index is used, that index will allow the software direct access to the 10,000 READ UNITS that will contain all of the data rows. Compared to reading the entire table, this reduces the amount of data physically read from disk to one percent, which greatly impacts the performance.

The three most common types of subsetting criteria that are used to subset a table are demographic, geographic, and time-based. Demographic subsetting of tables involves columns that contain information on a person's GENDER, AGE, and INCOME. Geographic subsetting is typically done on columns with information such as COUNTRY, STATE, ZIP CODE, and REGION. Time-based subsetting is typically done on columns containing information such as YEAR, MONTH, and QUARTER. Managing a table for subsetting can be a balancing act between the different queries that will subset data from the table. There could be a set of queries that uses just time criteria to subset rows. Other queries might be based on geographic columns. Queries could subset rows based on demographic columns. Still other queries might subset data from the table using any combination of columns.

While it would be wonderful if a data table could be optimized for every type of subsetting query, which is not really possible. Let's use the last example with the 100 million-row table and now consider three queries that all sub-set 1 million rows. One query subsets rows based on time, the other on age, and the third on state. Only if the three queries were to request the identical set of rows would all three queries be optimized. Some natural optimization does occur where a data table is optimized for subsetting by state and in so doing; queries that select based on zip codes also receive some optimization. Optimizing a data table for one query often prevents it from being ideally optimized for other queries. A good example might be optimizing a data table for time, which would disperse the rows for income class across all time periods.

It is possible to layer optimization of a data table. Consider a column billing month and gender. When optimizing the data table, you can choose to ideally optimize it for billing month. Within billing month, the data table can be optimized for a column such as gender. The layering of optimization can often be used several layers deep. Often, with each additional layer, less optimization occurs because the data at the lower levels of the layers is more dispersed.

On our demonstration table the columns REGION and BILLMTH are used to demonstrate how data can be organized to optimize it for queries that will subset data that is based on just one or both of those columns. The algorithms used to subset data are often dependent on the I/O ability of the hardware because they are able to read the table and perform the subsetting in parallel. Within the table there are 12 distinct values in the column BILLMTH and four distinct values within the column REGION. As I explained in a previous paragraph, if we can minimize the number of reads performed by organizing the rows for subsetting data onto fewer read units, then we can boost performance.

Three tables will be created for the first demonstration, which will select one of four regions. The query that is run against the three tables will select all of the rows for one of the four region values that will subset approximately 25 percent of the rows from the table. The first table will have no indexes, and it will be used to perform a full-table scan. The second table will be ideally optimized for selecting rows based on the region column; and for the last table, the values in the region column will be randomly distributed using the same technique of sorting the table by a column containing a random number. Earlier in this paper, I explained how the data rows of a table are written to disk. We need to revisit this because it plays a huge part when optimizing a data table for subsetting. Consider the two data tables we are using for indexed subsetting: for one table, the number of READ UNITS is minimized; for the second table, every READ UNIT contains rows that will have the region on which the subsetting is being performed.

Below is an example of the query that is used to subset a region from the tables:

```
data bigdata.table_out ;
  set bigdata.sales_history_2004 ;
  where region = "N" ;
run ;
```

The table below shows the runtime, the average CPU percentage that was used during the process, and the total CPU utilization factor that was computed using the runtime and average CPU that was used to run the task to process the table. The total CPU utilization factor is a computation that allows you to compare CPU used between the three jobs. It is computed by taking the average percentage of CPU used dividing it by 25, which is the percentage of total processing power that each CPU has on the server. The computed number is then multiplied by the runtime in seconds. For example, the average CPU utilization of 51 percent divided by 25 is 2.04. This number indicates that, on average, 2.04 CPUs were used during the running of the sub-setting process.

Table processed	Runtime	Average CPU utilization	Total CPU Utilization factor (computed)
Full-table scan	2 Min 30 seconds	51 percent	31,160 units
Optimized for indexed subsetting	2 min 33 seconds	63 percent	35,400 units
Random distribution of subset rows	2 min 40 seconds	79 percent	51,000 units
Clustered table	2 Min 10 seconds	64 percent	33,512 units

We can see from the above table that the three queries, full-table scan, optimized for subsetting with indexes, and the clustered table, were very similar in terms of the CPU used. Out of these three tables, the best CPU time was the full-table scan, and the fastest run time was on the clustered table. Out of the 4 tables used clearly the worst case was the indexed table on which had no optimization for the data. The query to subset rows from this table used more CPU than was needed to subset rows from the other three tables. These demonstrations show that managing a data table as we have described earlier to optimize it for a specific subsetting pattern will affect performance. Please note that the full-table scan was not more costly than the optimized table and cluster table. As was stated earlier, use of an index does have the overhead cost of building the list of rows to select, which in this case, where 25 percent of the rows were selected, balanced the cost of reading the entire table and checking every row.

We now perform the same kind of test with a query that has two where predicates. This second query selects a smaller subset from the table. This query uses the subset based on REGION, but it also uses the BILLMTH column to further subset the table by selecting only rows that are in the first quarter of a year. The demonstration uses the same four tables in the earlier demonstrations, except a second index has been added on BILLMTH for the indexed cases. Also for the ideally organized table, the data table has layered subsetting organization, which includes the first REGION as the primary layer and BILLMTH as the secondary layer. A second cluster table is also used. It has 48 member tables, and each member table matches one unique combination of REGION and BILLMTH. For example, one member contains only rows where REGION is equal to **N** and the BILLMTH is **JANUARY 01, 2004**. The total percent of rows for which subsetting will be performed is approximately 6.25 percent. The query is below:

```
Data junk ;
  set &domain..sales_history_2004_subset_test1 ;
  where region = "N"
     and billmth between '01jan2004'd and '01mar2004'd ;
run ;
```

The table below shows the runtime, the average CPU percentage that was used during the process, and the total CPU utilization factor that was computed using the runtime and average CPU that was used to run the task to process the table.

Table processed	Runtime	Average CPU utilization	Total CPU Utilization factor (computed)
Full-table scan	1 Min 49 seconds	51 percent	23,440 units
Optimized for indexed subsetting	33 seconds	66 percent	9,300 units
Random distribution of subset rows	1 min 55 seconds	62 percent	28,900 units
Clustered by REGION	32 seconds	64 percent	9,000 units
Clustered by REGION & BILLMTH	32 seconds	61 percent	8,540 units

OPTIMIZING A DATA TABLE FOR BOTH SUBSETTING AND ORDERED PROCESSING

The choices for optimizing a data table for subsetting or for ordered processing are often limited by the size of the table. Tables can be so large that they cannot be ideally optimized for ordered processing. From the demonstrations, we know that even less than ideal optimization for ordered processing helps reduce the work that is required for sorting a table. We also saw that a data table can be ideally optimized for subsetting of rows. Optimization for subsetting of data can be achieved across multiple columns, but with each additional column, you can expect to see less benefit.

If the type of optimization possible is not limited by the size of the data table, then it is possible to choose a type of optimization and apply it to the table. The benchmarks from this paper can serve as a guide to determine which type of optimization to apply to a data table. The goal is to minimize the work that is required to satisfy queries. If the types of queries that will read the table are weighted more heavily toward ordered processing than optimizing, you might find a better path to take. In making the choice, it is important to understand by which values the data in a table will be processed. If the columns that will be used for ordered processing are also the record key, then it is very likely that choosing to optimize for ordered processing

will mean no optimization for subsetting will be possible. Often, ordering a data table by the record key will also organize the columns that are used for subsetting the data table as if they were randomly distributed.

Getting the Most Benefit Possible with SAS SPD Server

The SAS SPD Server has a feature called Dynamic Clusters. The feature was designed to allow customers to manage very large tables that are difficult or impossible to manage with standard tables. For example there are customers who are using this feature to manage tables that have billions of rows, and the tables are close to a terabyte in size. The resources and process time required to keep a multi-billion-row, terabyte-sized table ideally optimized for ordered processing make it prohibitive. The issues that customers face with large-table optimizations have been addressed within the Dynamic Cluster feature to allow better management of large tables for the different kinds of processing required. The feature can be used to create very good cross optimization of data for both ordered processing and subsetting of rows from the table. This section explains how.

Earlier in the paper, we used the Dynamic Cluster feature to manage a table that was optimized for subsetting. This was done by creating a table that contained data for a specific value in the REGION column. One table contained all of the rows where the column value was **N** for north. The same was done for the other three values of **S**, **E**, and **W**. These four individual tables can be ordered by the CARDHOLDER column, which is the column that is used for ordered processing of the table. As we know from the numbers presented earlier, this greatly reduces the amount of work that is required by a sort to order the data. But having to sort the table is not the optimal situation for processing. An important feature for a Dynamic Cluster is the ability to recognize that the individual member tables are ordered and to use that order when it matches the requirement of a query. The software has the ability to recognize a set of cases where it is beneficial to interleave the rows across member tables of a cluster and to feed them to a query that is requesting the rows for ordered processing. For the software to do this, the order of the individual members must match the order that is requested by the query.

To demonstrate the benefit of Dynamic Cluster members being interleaved on the fly, three tables with identical rows and ordering of the rows were created and the PROC SORT procedure was used to read the tables. The first table is the same table that was used to demonstrate performance when the full table is ideally organized. The numbers that are used are the same numbers that were used in an earlier section of the paper. This table is presented to provide a baseline for comparisons. The second table is a Dynamic cluster with four members, each member is sorted by CARDHOLDER, and the SORTEDBY table option is used. This causes the software to perform the CLUSTER BY OPTIMAZATION, which will interleave the rows of the four member tables rather than perform a physical sort. The third table is a Dynamic Cluster table that is identical to the second table, but the SORTEDBY table option was not used. This causes a physical sort to be performed on the entire cluster.

The table below shows the benefit of ordering individual members as it compares two jobs, one where the table must be sorted and one where the rows from the members are interleaved. The table below shows the runtime, the average CPU percentage that was used during the process, and the total CPU utilization factor that was computed using the runtime and average CPU that was used to run the task to process the table. The total CPU utilization factor is a computation that allows you to compare the CPU that was used for each of the three jobs. It is computed by taking the average percentage of CPU used and dividing it by 25, which is the percentage of total processing power that each CPU has on the server. The computed number is then multiplied by the runtime in seconds. For example, the average CPU utilization of 66.53 divided by 25 is 2.66. This number indicates that an average of 2.66 CPUs was used during the running.

Table processed	Runtime	Average CPU utilization	Total CPU Utilization factor (computed)
First Table (ideally optimized)	9 Min 30 seconds	66.53 percent	153,020 units
Second Table	10 min 40 seconds	74.57 percent	192,400 units
Third Table (no SORTEDBY)	26 min 19 seconds	57.65 percent	368,980 units

We can see from the above table that although the Dynamic Cluster BY OPTIMAZATION is not ideal, there is a tremendous benefit to using this feature. The run time and cost of interleaving the rows is not dramatically higher than the ideal optimized table when it is compared to the table that is not using this feature. The second table had only four members. If this table contained the same number of rows, but they were divided across more members, the work and therefore the time required to

interleave the rows would increase. This is because the interleave process must perform more comparisons between the member tables as it selects the next row to pass on for ordered processing.

Another helpful aspect with Dynamic Clusters is that you can manually specify a sort order for an entire cluster. For example, a cluster could have two tables that are both ordered by a column; but one table contains values 1 through 10 for the ordered column, and the second table contained values 11 through 20. When the cluster is created; the SORTEDBY table option can be used on the cluster to indicate that the entire cluster has an order. It is important to know that when creating the cluster, the first member that is specified in the PROC SPDO CREATE CLUSTER command must contain the table with rows 1 through 10, and the second table must be the one that contains the rows ordered 11 through 20. Then the SORTEDBY table option can be used to indicate the entire table is ordered.

SUMMARY

What this paper shows is that there are ways to organize data tables to benefit performance. When managing large amounts of data you should consider the types of queries that will be using the data in the tables and understand the different approaches to improving performance. This paper provides the most basic methods of managing large data for the most common types of queries.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Daniel Sargent
SAS Institute
SAS Campus Drive
Cary, NC 27513
Work Phone: 919-531-6477
E-mail: Daniel.Sargent@SAS.com