

SPD Server 4.1: Scalability Solution for SAS ®

Turning Big Data into Business and Analytic Intelligence

Daniel M. Sargent
Advanced Server Development
SAS Institute Inc., Cary, NC 27513



Abstract

This paper will provide an overview to the new features and enhancements for SPD Server 4.0. The features and enhancements added address performance for the SAS Enterprise Marketing Automation Solution. They are SQL planner optimization, Where Planner Costing, Enhancements to Parallel Group By, Cluster Tables, Random DPF start placement, Time Based Partitioning, Partition By Value, a significant increase in the maximum number of rows and columns in a table, and SPDSSNET Consolidation.

Introduction

The SAS Scalable Performance Data Server™ (SPD Server) was first released in 1995 to meet the storage and performance needs for processing large amounts of data using SAS. Over the years SAS SPD Server has developed the nickname, Speedy Server. The very first SAS SPD Server customer has upgraded and expanded their use of the software. The largest physical disk footprint of a SAS SPD Server site is 8 terabytes. The largest tables stored in SAS SPD Server are over 500 Gigabytes, with one table planned for 2003 which is estimated to be over 1 terabyte. We have a customer with a table that approaches 1 billion rows. One site has recorded over 1 terabyte of data being pulled in a single day. On an average day at this site over 300 users extract in excess of half terabyte of data. To summarize SAS SPD server is used to turn big data into business and analytic intelligence.

Speedy Server is installed at over 400 sites world wide. It provides customers from industries such as banking, credit card, insurance, telecommunications, healthcare, pharmaceutical, transportation, and brokerage, plus a wide variety of government agencies high performance data storage.

For this paper a basic understanding of SAS SPD Server is required. The concepts of using data partitioning with the libname.parm parameter file, SMP enabled servers, and disk I/O setup is helpful.

SQL Planner Optimizations

Optimizations have been incorporated in the SAS SPD Server 4.0 SQL planner to improve the performance of many common queries used for Enterprise Marketing Automation. This section will take a high level view of the changes to the planner. The next four sections on Optimized Correlated Queries, Where Planner Costing, Parallel Group By, and Index Scans provide more details on enhancements that are complete.

SAS offers solutions which rely upon SQL query performance. Enhancements to SPD Server 4.0 optimize correlated queries, which are common when business and analytic intelligence are put to use driving Enterprise Marketing Campaigns. The work includes tighter integration of the Parallel Group By technology in the planner so nested Group By syntax will gain performance benefits. The server also optimizes where clause planning if a query makes use of views. Included are optimizations to speed up performance of correlated queries using query rewrite techniques. From Planning, to ETL, to storage, to business intelligence, to analytic intelligence, SAS offers robust end to end solutions which simplify IT shopping so one package meets the needs for cost, creation of intelligence, performance, and ROI.

Enterprise Marketing Automation queries are ideal to optimize for they represent a frequent use of business intelligence. Intelligent businesses know how to target profitable new customers, retain current customers, and cross market existing customers, all in a cost effective way. Enterprise Marketing is the business and analytic intelligence put into action. The purpose of the marketing

is to grow revenues of the company in a profitable manner.

Today, companies have data that is rich with demographic information, predictive measures of profitability, behavioral information as well as the success and mistakes of the past. A company must have tools to help them to climb to the top of these enormous mountains of data and reach the highest peak, view the entire landscape of business and analytic truth or simply put intelligence. A company with rich data and industry- leading business intelligence software can put it to use for data mining and predictive modeling creating competitive advantages. Making good use of this data secures the future of the business.

The SQL generated by these intelligence tools involve highly complex business knowledge. These SQL statements have a multitude of sub setting where clauses, joins to include or exclude populations, and criteria to focus on groups who match the business model for profitable customers. Our effort is to make sure the SAS SPD Server 4.0 SQL planner will execute the complex queries in a timely manner.

Optimizing Correlated Queries

The enterprise of today uses a number of front end tools that match different goals of the company: financial aspects of a business, marketing, sales prospecting, operations, and supply chain are a few. With all the different front end tools managing the business and analytic intelligence it is necessary to have a single back end storage that will handle their requests.

Turning complex rules into action is where intelligent storage steps up. Intelligent storage must have the ability to interpret and process complex requests. SQL engine is such a tool that takes complex rules and breaks them into a series of simple steps to accomplish the task.

SAS SPD Server 4.0 includes new techniques that optimize correlated queries. The complexity of business and analytic intelligence being put into action by front end tools generate SQL queries that are often nested 3 or 4 layers deep.

A correlated query is one where a nested section has a relationship to a column in a un-nested or higher nested section of the query. The planner has been rewritten to detect these relationships between the nested and un-nested sections of the SQL and restructure or recode the SQL into a simpler form. The recoding of the SQL will execute steps to create temporary SPD Server tables on the fly which replace entire sections of code. This intelligence has allowed the new planner to speed up the types of queries our customers use and proving once again why SAS SPD Server gets the nickname Speedy Server.

Where Plan Costing

The where plan costing implemented in SAS SPD Server 4.0 is designed to prevent taking resource-expensive approaches. By eliminating resource-expensive approaches, the where planner provides assurance of making choices that will provide good performance.

With SPD Server 3.x and earlier at times the user had to manually tune queries to achieve good performance by setting macro variables or turning off indexes. SPD Server 4.0 does the work for the user by costing the different approaches to index evaluation. This section will provide overviews for the segmented hybrid index design, an understanding of the information used in costing, how a where clause is evaluated, and how the costing is performed.

A segmented index is an index scheme that divides the index of a table into small parts. It is like having a series of mini or sub-indexes for blocks of rows in the table. When an index is built on a SAS SPD Server table by default every 8192 rows in the table has its own index segment.

A hybrid index is simply an index scheme that uses both b-tree and bitmap technology under the covers. It checks the data to be indexed and calculates the estimated storage required for the value using both b-tree and bitmap. After this estimate is done it will choose one or the other type of index technology for that value in that segment. That means for one segment a value might be stored as a b-tree index and for a different segment it might use a bitmap. This technology simplifies administration of data storage which in turn keeps costs down.

There are two key factors used in the costing of indexes density and distribution. Density is the number of unique values divided by the number of rows. When there are a lot of rows for a given value in a column that value has a high density. When there is one row or few rows for a given value in a column that value has low density. The density value for an index ranges from 0 to 1. The closer to 1 the index density is the lower the density. Indexes with a density of 1 are unique, an index with a density of .9 or higher are considered low density. Distribution refers to how close physically all rows with the same value are stored to each other in a table. When a column value is in many rows but they are scattered throughout the table they have a wide distribution. If a column value is in many rows and the rows are adjacent or nearly adjacent the distribution is clustered.

The SPD Server indexing keeps track of the values and their density and distribution. This information is used for costing a where clause.

The where planner has four evaluation strategies used to select the rows required of a where clause. We call these four methods EVAL 1, EVAL 2, EVAL 3, and EVAL 4.

EVAL 1, EVAL 3, and EVAL 4 are used to evaluate true rows in the table using indexes. EVAL 2 is a strategy used to evaluate true rows of a table without using indexes.

EVAL 1 evaluates true rows using an index for each segment of the table. The index evaluation process results in a list of true row id's per-segment. This strategy can handle all where clause syntax such as EQ, =, LE, <=, LT, <, GE, >=, GT, >, IN, BETWEEN are some examples. This method uses threaded parallel processing across the index segments to allow the index to be evaluated concurrently. The method includes combining multiple per-segment bitmaps from queries that utilize multiple indexes to determine per-segment row id's to return.

EVAL 2 is a strategy that is handed true rows from and EVAL 1, EVAL 3, EVAL 4, or directly from the table and uses brute force to eliminate rows which are false. EVAL 2 is used to process all rows of a table when no index evaluation is possible. This can happen when rows cannot be selected using an index because an index is not present or a function has made the use of an index invalid.

EVAL 3 is a single index sequential process strategy. It is chosen when the number of rows being returned by an index is unique or is near unique. EVAL 3 will return a list of true rows for the entire table. EVAL 3 has support for syntax that handles EQ and = only.

EVAL 4 is similar to EVAL 3 but is designed to handle complex verbs like IN, GT, GE, LT, LE, and BETWEEN.

With SPD Server 3.x and earlier the planner relied heavily on threaded strategies EVAL 1 and EVAL 2 for most where clauses. For some where clauses these strategies would over-thread and over-manipulate indexes during the multi-index evaluations which reduced performance or consume too many resources. With the costing, EVAL 3 and EVAL 4 become more natural evaluation choices benefiting performance and resource utilization.

The following is a step by step explanation of how the where planner processes a where clause.

When the planner encounters a where clause, an expression tree is built to represent all the predicate criteria. The purpose of the where planner is to split the predicates into two trees. One tree will contain predicates that use an EVAL 2 approach because they lack indexes that can be used or functions invalidate index use. The remaining tree will be assigned to an evaluation strategy of EVAL 1, EVAL 3, or EVAL 4. This last assignment is based on costing and syntax.

The tree which was not assigned to the EVAL 2 strategy is first checked for low density predicates. When one or more of the predicates are determined to be low density the predicate with the lowest density value is chosen to be

evaluated by using an index approach and all remaining predicates are assigned to the EVAL 2 tree strategy. Based on the syntax explained in the sections above the one remaining predicate on the tree is assigned EVAL 3 or EVAL 4. The single index EVAL 3 or EVAL 4 is chosen because the combination of the index density and distribution two characteristics can be determined: low work with high yield. With a strong work/yield ratio using just one index there is no reason to include other indexes which may cause more threading or index manipulation then is needed.

When the planner has determined there are no predicates that meet the low density criteria an EVAL 1 strategy is chosen. Before EVAL 1 one is performed costing is performed on the remaining predicates to prune high yield and high work predicates to the EVAL 2 tree.

A high yield predicate is one where a high percentage of rows of the true segments will be selected. The default for high yield is set to 25% of the rows being returned by the predicate. At this point the cost of using the index begins to return little to the evaluation.

High work predicates are predicates that will require a great deal of index manipulation to complete. When the amount of index work required is greater then the work required by using an EVAL 2 strategy the predicate is best in the EVAL 2 tree. Many open ended predicates that contain many values such as GT, LT, or between types of syntax are good high work candidates for EVAL 2. We determine high work predicates by checking the number of unique values in the predicate against the number of unique values in the index.

Segmented costing is a strategy used when all predicates in EVAL 1 are high yield or high work. For this costing true segments are passed to EVAL 2 for processing. This limits EVAL 2 to only segments of a table which may provide true rows for the where clause.

As discussed in the second paragraph of this section Where Costing was done to prevent users from having to manually tune where clauses by selecting the where evaluation strategy. The sample spdsserv.parm shipped with the media has the parm WHERECOSTING; set to turn costing on. When WHERECOSTING is on a job or step can use macros to turn it off. The macro SPDSWCST=NO will turn off costing. The macro SPDSWSEQ=YES over rides costing by allowing the user to force a global EVAL3 or EVAL4 Strategy. By removing the WHERECOSTING parm or setting it to NOWHERECOSTING; costing will be turned off for the entire server and cannot be used. Turning costing off by either the parm or setting SPDSWCST=NO will cause the where planner to revert to rules based planning equivalent to previous releases of SPDS. The appendix provides a matrix that shows the where evaluation policy selected for a where clause with an index for the WHERECOSTING

spddsserv.parm, SPDSWCST macro, and SPDSWSEQ macro.

The new costing strategies in SPD Server 4.0 assure better performance without any human intervention. Costing is a win all around for it reduces human involvement, simplifies coding, and increases performance while decreasing resource utilization.

Parallel Group By

Parallel Group By was introduced in SAS SPD Server release 2.1. The Parallel Group By (PGB) is a high performance parallel summarization of data executed using SQL. PGB works against single tables that perform aggregation of data. Summarization is common in warehousing applications and PGB was developed to speed up performance in these areas. It is often used in SQL queries through the use of sub queries to apply selection lists for inclusion or exclusion. Many of these lists use summarization functions and other clauses to select candidates for inclusion or exclusion.

PGB looks for specific patterns in a query that can be performed using parallel processing summarization. When a query pattern is found the SQL planner will perform a summarization using our PGB methods. These methods have limitations for the types of functions they can handle.

In release SAS SPD Server 4.0 the PGB support has been expanded in many areas. The three most important areas are new functions, support of table aliases, and nested queries that meet the group by syntax. PGB has been hooked into the main body of the planner to bring the benefits of PGB to more areas of our SQL engine. Any section of code that matches the PGB trigger pattern will use it. The following sections highlight some examples of SQL syntax that will now employ PGB technology.

The functions supported in Parallel Group By are now: COUNT, FREQ, N, AVG, MEAN, MAX, MIN, NMISS, STD, SUM, VAR. These functions are also able to handle use of the distinct verb inside the function. These functions are the minimum summary functions required to support the SAS Enterprise Marketing Automation tool.

Table alias support was added to better support front end tools, such as SAS Enterprise Marketing Automation, which generate SQL queries with table aliases. A table alias allows two things: shorter coding syntax and a way to select a specific column when two tables in a query have a common column to choose from.

With PGB integrated into the fabric of the planner many sections of queries will now take advantage of the performance and parallel processing. Some common areas include sub queries used to generate value lists in an

IN clause, views that meet the PGB syntax, views with nested Group By syntax.

With the expansion of PGB many common user query patterns will benefit. The appendix has several examples of queries with highlighted code to provide a clearer idea of where the enhancements have been made to both PGB and queries that fit the correlated patterns.

Index Scans

Index Scan technology is being introduced into the SAS SPD Server SQL Planner with release 4.1. An index scan is a SQL query optimization that uses the index(es) to resolve a SQL query rather than the physical table. SPD Server indexes contain a great deal of information about the column that is indexes. Some of the basic information contained about the column data is minimum value, maximum value, and counts for each value, and more. Many queries can be resolved using this data alone. Rather than recreate it from the physical table SPD Server SQL Planner uses this data in the index to resolve the query. The appendix section on SQL enhancements covers the SQL syntax supported by the index scan facility.

Cluster Tables

This section is an overview of the new virtual table structure called a cluster table. The following three sections will cover the Random DPF start placement is used by cluster tables and available to standard SPD Server tables, a new feature in SPD Server 4.0 called Time Based partitioning, and the experimental Partition By Value feature.

A cluster is a structure which can store multiple SAS SPD Server tables. The cluster table uses a layer of metadata to manage the slots or members. In a cluster table each slot or member can store a single SAS SPD Server table.

For cluster tables the sortedby flag, sortedby column list, unique indexes, rows inserts, and row deletions has been disabled. Future development work will look to address these areas.

A clustered data table provides a storage architecture that opens new areas of development for parallel processing and data management capabilities for SAS. Future areas that could be explored are racking for HOLAP cubes, range based parallel joins, range based processing, faster access to data, parallel loading, and faster refreshes of tables.

Where costing as described earlier in this paper for cluster table is applied on a member by member basis. For a cluster table with 48 members the best Eval strategy for each member will be used. Using a multiple Eval strategies provides better granularity for the process which can improve performance.

Random DPF Start Placement

Previous releases of SPD Server the first partition for all table in the same domain started in the first datapath defined in the libnames.parm domain definition. Time Based Partitioning and Partition By Value are tools that provide benefits and ways to manage large tables. By using these new data management tools large tables will be divided into smaller tables.

For large systems that require intensive I/O many files systems are used to distribute data. By having the first data partition begin in the first datapath of a domain created new design considerations. The following illustration is an example. Consider SPD Server running on a server with multi-terabytes of disk. Often for this size warehouse the domains could have 24 datapaths on separate file systems. If for example there was a table with 4 years of data that required 48 gigabytes each data partition would have to be 2 gigabytes to distribute the data evenly across all the file systems. If that same table was loaded into an Time Based Partitioning cluster with 48 monthly slots the entire table would require 1152 data partitions of about 42 megabytes. Having large numbers of files would require the software to issue a file open when a user accessed a table. Using Random DPF start placement allows for larger partitions and achieves good distribution.

With SPD Server being used for larger terabyte sites we have learned a great deal more about large warehousing. Domains both with larger number of tables and a wider variety of table sizes have been encountered. Often many small to medium files would all reside in the first few data paths of the domain. This can cause the I/O to become unbalanced. With SAS SPD Server 4.0 a parameter can be set to allow Random DPF Start Placement for all tables stored in the server.

Time Based Partitioning

This section introduces the new SAS SPD Server 4.0 Time Based Partitioning feature for loading and removing time based units of data from a table. It was developed to facilitate loading and removal of data from very large tables.

How quickly a company reacts to business and analytic intelligence is a key competitive advantage. As the sizes of data warehouses grow, their management becomes more difficult, driving up costs, stretching limited resources and budgets, complicating application management, and wasting valuable time needed to accomplish new development. Rather than having the organization's attention on the hassles of managing large data, Time Based Partitioning let them focus their attention on creating and using vital business and analytic intelligence.

The Time Based Partitioning feature uses the virtual table structure called a cluster. For Time Based Partitioning a cluster is created with a specific number of slots. When all of the slots are full with SAS SPD Server tables, and then a new one is added, the oldest one will be automatically deleted from its slot to accommodate the newest.

This section will refer to an example, ABC Company, which requires the warehouse table to hold the most recent 48 months of data. Each month of data will take over 10 gig of disk space for data and indexes with a full table size over 500 gigabytes. This data is vital to generate business and analytic intelligence for the organization. Suppose the most recently loaded month is December 2002. The oldest data required by the warehouse is January 1999. January 1999 is loaded first into slot 1, and then February 1999 is loaded into slot 2, and so on until all 48 slots have been loaded. The slots in the cluster are continuous by month or year making any unfilled slot valid.

Defining a cluster table requires syntax to specify the number of columns, and each column's name, type (numeric or character), width, and indexes. By default, the cluster will be set up with month-based time units, but can be defined with a table option to use annual based time units. The cluster will have generation dates associated with each time unit. The default generation date for monthly time units will be the first day of the month in which the cluster was created. For yearly clusters the default generation date will be the first day of the year in which the cluster was created. See examples for specifying a cluster in the Appendix section.

Once a cluster is specified, the table name, formats, informats, labels, and names of the columns can be changed using Proc Datasets. Indexes can be set using standard SAS language syntax. Indexes can be added or dropped as business needs change.

The load process for a clustered table requires up to three steps to complete. The first step is an optional step to archive any member that will be aged out during the load process. This is very important, because once a member has been aged out from the cluster; the data can only be recovered if there is a system backup or another source. In our example, after a new month, January 2003 is loaded the oldest month January 1999 will be deleted keeping the cluster at 48 slots.

The second step in the load process is adding a new member to the cluster. This is done through the SAS procedure Proc Append. Syntax to load the new member can be specified using table option for relative or absolute addressing. The macro variable spdsgend can also be set to direct loading of data to a new member. See examples for loading in the appendix section.

The last step is to run the system backup utilities to backup any new or changed files in the cluster. In our ABC Company example, when a new member is added and an old one is deleted, approximately 10 gigabytes of data is added. The system utilities will detect the new and any changed files and back them up. The amount of data to be backed up will approximate the size of the data added, any indexes for the new data, and some metadata that is modified by the cluster. Time Based Partitioning can reduce load times, system resources, and human intervention needed by transparently retiring obsolete data.

In conclusion the new Time Based Partitioning feature provides a rich benefit to the administration of data warehousing. It is designed to streamline loading and deleting of data from large tables. It requires one time up front administration to set up a cluster table and little else which keeps costs down by streamlining warehouse operations.

Partition By Value

Using the virtual table design for cluster tables SAS SPD Server 4.0 will introduce an experimental version of a cluster table that allows Partition By Value (PBV). For example a virtual table is created with 51 slots. Each slot has a where clause that specifies a state value and the 51st slot catches any non valid state values. Each slot will have virtual index(es) based on the where clause used to restrict the rows loaded in it. For the PBV example no physical index would be required to get fast access to the state values. No creation and no storage space on disk. This is beneficial to warehouses that need to access to both an entire table and one where users often access specific sets of data in a table.

Lines of business and geographic boundaries are two common examples of criteria well suited for PBV. Physical indexes can be built for any columns and like Time Based Partitioning are defined at the PBV cluster level. The where planner will view the virtual index like it was any other index. Using the 51 state cluster example a user could access all data for the state of CT with a where clause. No physical reading of an index will be necessary.

For the experimental cluster feature, Partition By Value, we will support single column partitioning. A example of partitioning by a column on state is a simple case. An example for use of a function would be year

(processed_date). The only functions supported are functions that have only the column as input.

Increased Row Capacity

With mountains of data to store the ability to address updating tables with billions of rows to enable generating business intelligence quickly, and thus providing immediate ROI is more important then ever. The Clustered Data Table with SAS SPD Server 4.0 provides a new foundation for the next generation of data storage with the SAS system. Previous versions of SPD Server were based on 32 bit architecture that supported just over 2 billion rows. The number of columns a table could have was limited to 32,768. SPD Server 4.0 is based on 64 bit architecture which will support tables over 9 quintillion rows with over 2 billion columns.

SPDSSNET Consolidation

SPDSSNET has been the path for a windows desktop application to connect to the power of SPD Server backend. To use SPDSSNET a separate port and process had to be running on the SPD Server box. With SPD Server 4.0 ODBC connectivity can go through the standard SPD Server ports defined on the server box.

With SPD Server 3.x and earlier multiple library connections required manually editing two parameter files. With 4.0 those requirements are outdated. SPD Server can now use the SAS ODBC Driver GUI to define everything required to make connections to the server.

Removing SPDSSNET from the ODBC client - SPD Server channel improves performance and simplifies SPD Server administration

Conclusion

The new features in SAS SPD Server 4.0 provide a new direction for Storage at SAS and for SPD Server. Supporting SAS tools used to Plan, extraction, transform, loading, and generates business & analytic intelligence; SPD Server seamlessly fills high performance storage needs for the SAS in the Intelligence Value Chain.

References

SAS® and SAS/SPD Server® are registered trademarks of SAS Institute, Inc., Cary, NC

Appendix

Time Based Partitioning

Defining and loading:

Example 1 – define and load first slot, add addition rows to first slot, load data into second slot.

```
%let spdsgend=1/1/1999 ;
/* Monthly slots by default */
data spdslib.aging_cluster
    (maxgen=48
     index=(clientid state trans_dt)) ;
    set temporary_sas_table ;
    /* more lines of code */
run ;

/* append new rows to first slot */
proc append base=spdslib.aging_cluster data=temp ;
run ;

/* load data into second slot */
proc append base=spdslib.aging_cluster (numgen=+1)
    data=temp ;
run ;
```

Example 2 - Define cluster based existing SAS table. No slots are loaded.

```
%let spdsgend=1/1/1998 ;

data spdslib.aging_cluster
    (maxgen=7
     index=(clientid state trans_dt)
     typegen=year) ;

    set temporary_sas_table (obs=0) ;
    /* more lines of code */
run ;
```

Example 3 – loading member from flat file.

```
libname qabig1 sasspds 'qabig1' host='zztop.unx.sas.com'
serv='5160' user='anonymous' ;
```

```
%let spdsgend=1/1/1999 ;
data qabig1.table (maxgen=48) ;
    /* define name, length of column */
    length a 8.
           b $4. ;
    /* define format for column */
    format a dollar12.2
           b $4. ;
    /* prevent writing record */
    stop ;
```

```
run ;

/* read flat file into slot */
/* cluster must be defined */

data temp_view / view=temp_view ;
    infile 'C:\mydata\flat_file.txt';
    input b $4.
           a 8. ;
run ;

proc append
    base=qabig1.table
    data=temp_view ;
run ;

proc append
    base=qabig1.table(numgen=+1)
    data=temp_view ;
run ;
```

Access entire table:

```
/* get all records for states of NC, CT, and WA */  
Data temp ;  
  Set spdslib.aging_rollout (keep=clientid state sales) ;  
  Where state in ('NC','CT','WA') ;  
Run ;
```

Relative access:

Use relative numbering to access December 2002, January 2002, and then January 2001 from ABC Company table.

```
/* Read most current slot Dec 2002 */  
data temp ;  
  set spdslib.aging_rollout (numgen=0) ;  
run ;  
  
/* read January 2002 */  
Data temp ;  
  Set spdslib.aging_rollout (numgen=-11) ;  
Run ;  
  
/* read January 2001 */  
Data temp ;  
  Set spdslib.aging_rollout (numgen=23) ;  
Run ;
```

Absolute access:

Use absolute numbering to access December 2002, January 2002, and then January 2001 from ABC Company table.

```
/* Read most current slot Dec 2002 */  
data temp ;  
  set spdslib.aging_rollout (numgen=48) ;  
run ;  
  
/* read January 2002 */  
Data temp ;  
  Set spdslib.aging_rollout (numgen=37) ;  
Run ;  
  
/* read January 2001 */  
Data temp ;  
  Set spdslib.aging_rollout (numgen=25) ;  
Run ;
```

SQL ENHANCEMENTS:

Includes new parallel Group by Support and correlated query examples.

All new syntax support in **BOLD CAPITAL**

Example 1 Correlated Query:

```
Select a.clientid
  from VCLIENT A
 where a.gender = 'F'
       and a.age between 30 and 40
       and (a.status in ( 'C', 'E', 'X') or a.employmentstatus = 'E')
       and a.clientid in (select hto.clientid
                          from hld51fl hto)
       and a.clientid in (select w.clientid
                          from vmotor w
                          where w.grossannualpremium > 50
                                and w.effectivedate = (select min (w.effectivedate)
                                                         from VMOTOR W
                                                         where W.CLIENTID = A.CLIENTID))
```

Example 2 table alias support in Parallel Group By:

```
/* With alias */
proc sql _method ;
  select a.clientid,
         sum (A.savings) as client_savings
  from spds.client_table A
  group by a.clientid ;
quit ;

/* without alias */
proc sql _method ;
  select clientid,
         sum (savings) as client_savings
  from spds.client_table
  group by clientid ;
quit ;
```

Example 3 nested Parallel Group By:

```
Proc sql ;
  Select a.customer
  From spdsdat.customer_sales a
  Where a.customer in
        (SELECT B.CUSTOMER,
         SUM (B.TOTAL_SALES)
         FROM SPDS.CUSTOMER_SALES B
         GROUP BY B.TOTAL_SALES
         HAVING B.TOTAL_SALES > 100000) ;
Quit ;
```

Example 4 group by in view:

```
libname qabig1 sasspds 'qabig1' host='zztop.unx.sas.com' serv='5160' user='anonymous'  
  passthru=dbq="qabig1" user="anonymous" host="zztop.unx.sas.com" serv="5160" ;
```

```
proc sql ;  
  create view qabig1.sugi28_view as  
  select agent,  
         sum(amount) as amount,  
         sum(units) as units,  
         count (*) as tranacts  
  from qabig1.sortedsugi28  
  where trandate between '01dec1996'd and '31dec1996'd  
         and trantype in (1,4,5)  
  group by agent ;  
quit ;
```

```
proc sql ;  
  create table temp as  
  select *  
  from qabig1.sugi28_view  
  where units > 50  
         and amount < 50 ;  
quit ;
```

Index Scan facility

SPD Server 4.2 SQL provides enhanced index scan support for the following functions:

min, max, count, count distinct, nmiss, and range functions.

All index scan capabilities listed above are available for both standard SPD Server tables as well as clustered tables, with the exception of count distinct. The count distinct index scan function is not available in clustered tables.

The count(*) function is the only function included with the index scan support enhancement that does *not* require an index on the table. For example,

```
select count(*) from tablename;
```

will return the number of rows in the large table *tablename* without performing a row scan of the table. Table metadata is able to return the correct number of rows. As a result, the response is as fast as an index scan, even on an unindexed table in this case.

Count(*) functions with WHERE-clauses require an index for the index scan feature to provide the performance enhancement. For example, suppose SPD Server table Foo has indexes on numeric columns a and b. The following count(*) functions benefit from SPD Server index scan support:

```
select count(*) from Foo where a = 1;  
select count(*) from Foo where a LT 4 and b EQ 5;  
select count(*) from Foo where a in (2,4,5) or b in (10,20,30);
```

The **min, max, count, count distinct, nmiss, and range** functions all require indexes on function columns to exploit the index scan performance savings. Unlike the count(*) function, index scan support for the **min, max, count, count distinct, nmiss, and range** functions does *not* support WHERE-clauses. For example, suppose SPD Server table Bar has indexes on numeric columns x and y. The following SQL submissions will be able to exploit the performance gains of index scans:

```
select min(x), max(x), count(x), nmiss(x), range(x), count(distinct x) from Bar;  
select min(x), min(y), count(x), count(y) from Bar;
```

If any one function in a statement does not meet the index scan criteria, all functions in that statement will revert to being resolved by table scan instead of index scan. Suppose the SPD Server table Oops has indexes on numeric columns x and y. Column z is not indexed. Then, the SPD Server SQL statement below

```
select min(x), min(y), count(x), count(y), count(z) from Oops;
```

will be entirely evaluated by table scan; index scanning will not be performed on any of the functions. To take advantage of index scans, the statement above could be resubmitted as

```
select min(x), min(y), count(x), count(y) from Oops;  
select count(y) from Bar;
```

The functions min(x), min(y), count(x), and count(y) will be evaluated using index scan metadata and will exploit the performance gains. The function count(y) will continue to be evaluated by table scan. The count(*) function can be combined with other functions and benefit from index scan performance gains. Continuing to use the SPD Server table Oops with indexes on numeric columns x and y, the following SPD Server SQL statement will benefit from index scan performance:

```
select min(x), range(y), count(x), count(*) from Oops;
```

Where Planner Costing

Create table:

NOTE: Libref F00 was successfully assigned as follows:

```
Engine:          SASSPDS  
Physical Name:  :29762/IDX1/srcmgr/qabig1/
```

```
247  
248 data foo.x(index=(x y z));  
249 do x = 1 to 100000;  
250 y = mod(x,2);  
251 z = mod(x,1000);  
252 output;  
253 end;  
254 run;
```

NOTE: The data set F00.X has 100000 observations and 3 variables.

```
NOTE: DATA statement used:  
real time          2.61 seconds  
cpu time           2.46 seconds
```

Example 1:

```
255  
256 /*  
257 * unique costing: Do EVAL3 on the unique index "x" and EVAL2 on  
258 * the resulting rows (index y is not used)  
259 */  
260 %let SPDSWDEB=YES;  
261 data foo._null_;  
262 set foo.x;  
263 where x=1 and y=1;
```

```
whinit: WHERE ((x=1) and (y=1))  
whinit: wh-tree presented  
                /-NAME = [x]  
                /-CEQ----|  
                |         \-LITN = [1]  
--LAND---|
```

```

        |           /-NAME = [y]
        \-CEQ----|
                \-LITN = [1]
whinit: wh-tree after split
--<empty>
whinit costing: wh-tree after near unique xtree pruning
        /-NAME = [y] INDEX Y (y)
--CEQ----|
        \-LITN = [1]
whinit: INDEX tree after split
        /-NAME = [x] <1>INDEX X (x)
--CEQ----|
        \-LITN = [1]
whinit returns: ALL EVAL2 EVAL3

264 run;

```

NOTE: There were 1 observations read from the data set F00.X.
WHERE (x=1) and (y=1);

Example 2:

```

266 /*
267 * segmented costing: Detection of a trivially FALSE predicate
268 * due to no TRUE segments results in no further segmenting logic,
269 * no rows are returned.
270 */
271 data foo._null_;
272 set foo.x;
273 where y eq .5;

whinit: WHERE (y=0.5)
whinit: wh-tree presented
        /-NAME = [y]
--CEQ----|
        \-LITN = [0.5]
whinit: wh-tree after split
--<empty>
whinit: INDEX Y uses 0% of segs (WITHIN maxsegratio 75%)
whinit: INDEX tree after split
        /-NAME = [y] <1>INDEX Y (y)
--CEQ----|
        \-LITN = [0.5]
whinit costing: detects no TRUE segments (TRIVIALY FALSE)
whinit returns: FALSE

274 run;

```

NOTE: There were 0 observations read from the data set F00.X.
WHERE y=0.5;

Example 3:

```
276 /*
277  * segmented costing: High yield Pruning. Do EVAL1 on the z predicate,
278  * detect that the y predicate returns lots of rows and therefore
279  * adds little to the evaluation. Push the y predicate to EVAL2.
280  *
281  * minimize where threads: This example also reduces the number of
282  * EVAL1 where threads to assure sufficient work for the threads.
283  */
284 data foo._null_;
285 set foo.x;
286 where y eq 0 and z eq 500;

whinit: WHERE ((y=0) and (z=500))
whinit: wh-tree presented
      /-NAME = [y]
      /-CEQ----|
      |          \-LITN = [0]
--LAND---|
      |          /-NAME = [z]
      \-CEQ----|
              \-LITN = [500]
whinit: wh-tree after split
--<empty>
whinit: INDEX Y uses at least 76% of segs (EXCEEDS maxsegratio 75%)
whinit: INDEX Z uses at least 76% of segs (EXCEEDS maxsegratio 75%)
whinit: INDEX tree after split
      /-NAME = [y] <1>INDEX Y (y)
      /-CEQ----|
      |          \-LITN = [0]
--LAND---|
      |          /-NAME = [z] <2>INDEX Z (z)
      \-CEQ----|
              \-LITN = [500]
whinit costing: segment yield of 3 reduces whthrds from 12 to 4
whinit costing: prunes high yield node 0
whinit costing: wh-tree after costing
      /-NAME = [y] <1>INDEX Y (y)
--CEQ----|
      \-LITN = [0]
whinit costing: INDEX-tree after costing
      /-NAME = [z] <2>INDEX Z (z)
      /-CEQ----|
      |          \-LITN = [500]
--LAND---|
whinit: checking all index segments
whinit returns: ALL EVAL1 EVAL2

287 run;

NOTE: There were 100 observations read from the data set F00.X.
      WHERE (y=0) and (z=500);
```

Example 4:

```

289 /*
290 * segmented costing: High work Pruning. Costing detects it's a lot of
291 * work to evaluate "x>100" for each segment. Therefore that predicate
292 * is pruned to EVAL2.
293 */
294 data foo._null_;
295 set foo.x;
296 where z = 5 and x > 100;

```

```

whinit: WHERE ((z=5) and (x>100))
whinit: wh-tree presented
      /-NAME = [z]
    /-CEQ----|
  |           \-LITN = [5]
--LAND---|
  |           /-NAME = [x]
    \-CGT----|
      \-LITN = [100]
whinit: wh-tree after split
--<empty>
whinit: INDEX Z uses at least 76% of segs (EXCEEDS maxsegratio 75%)
whinit: INDEX X uses at least 76% of segs (EXCEEDS maxsegratio 75%)
whinit: INDEX tree after split
      /-NAME = [z] <1>INDEX Z (z)
    /-CEQ----|
  |           \-LITN = [5]
--LAND---|
  |           /-NAME = [x] <2>INDEX X (x)
    \-CGT----|
      \-LITN = [100]
whinit costing: segment yield of 3 reduces whthrds from 12 to 4
whinit costing: prunes high cost node 1
whinit costing: wh-tree after costing
      /-NAME = [x] <2>INDEX X (x)
    --CGT----|
      \-LITN = [100]
whinit costing: INDEX-tree after costing
      /-NAME = [z] <1>INDEX Z (z)
    /-CEQ----|
  |           \-LITN = [5]
--LAND---|
whinit: checking all index segments
whinit returns: ALL EVAL1 EVAL2

```

```
297 run;
```

NOTE: There were 99 observations read from the data set F00.X.
 WHERE (z=5) and (x>100);

WHERE COSTING MATRIX:

WHERECOSTING	SPDSWCST	SPDSWSEQ	result
<don't care> NO	<don't care> <don't care>	YES NO	EVAL3 or EVAL4 EVAL1

not defined
<don't care>
YES

<don't care>
NO
YES

NO
NO
NO

EVAL1
EVAL1
where costing