Paper 008-30

Frequently Asked Questions about SAS® Indexes

Billy Clifford, SAS Institute, Inc., Cary, NC

ABSTRACT

Indexes in the BASE engine have been part of Base SAS since SAS 6. Over the years the use and mis-use of indexes has generated a number of questions from users. The answers to these questions might help you understand more about indexes and how they interact with other SAS components. This paper begins with an overview of index technology used by the BASE engine and how SAS uses these indexes. Then, a set of Frequently Asked Questions (FAQs) collected from customers, in-house users, and Tech Support is answered, along with the rationale for the answer.

WHAT IS AN INDEX?

An index is an auxiliary data structure that provides fast access to objects by value. All the data in the index is redundant and is derived from the data set. It's the organization of this redundant data that provides the performance benefit.

Without an index, any operation that subsets the data, such as a WHERE expression, must search every observation in the data set sequentially, looking for matching criteria. The use of an index can significantly reduce the time to locate the desired subset of observations, especially if the subset is small compared to the entire data set.

You can create an index for one or more variables of the data set. If multiple variables are specified, the values of each variable are concatenated in the order specified, to form the indexed value. An index that has one variable is called a simple index. An index that has multiple variables is called a composite index. Unless specifically mentioned, this paper does not distinguish between a simple index and a composite index.

You can create multiple indexes for the same data set. All indexes for a given data set are stored in a single file that is associated with the data set. When the data set is updated by adding observations, deleting observations, or changing variable values, the indexes are automatically updated.

HOW IS AN INDEX STRUCTURED?

The values for an indexed variable are stored in an inverted tree structure, which is similar to the diagram shown in Figure 1.

Each leaf node contains a set of value/RID pairs that are ordered by value. The RID, or Record Identifier, is used to locate the observation on disk that contains the value. All leaf nodes, as a unit, form an ordered set of value/RID pairs for all values of the indexed variable. The first value in the left-most leaf node is the lowest value. The last value in the right-most leaf node is the highest value.

Each non-leaf node contains a set of value/NID pairs that are ordered by value. The NID, or Node Identifier, is used to locate the child node at the next lower level. The value is the highest value that's contained in the child node that is pointed to by NID. All non-leaf nodes, as a unit, form a multilevel directory that is used to locate the value/RID pair equal to or greater than the requested value.

The search for a requested value begins at the root node and moves down through one node at each level until the leaf node that should contain the value is found. If the requested value is not found, then the value is not in the index. If the requested value is found, its RID can be used to read the observation that contains the requested value. Subsequent value/RID pairs that are greater than the requested value are found by reading the remaining value/RID pairs in the node, and following the matching NID to the leaf nodes that contain the higher values.

An index tree is balanced when all leaf nodes are the same distance from the root. This attribute of an index is important because it provides a uniform cost for accessing any node. Updating the data set can cause the tree to become unbalanced. Adding values can cause parts of the tree to expand. Deleting values can cause parts of the tree to shrink. The index code dynamically prunes undesirable growth as updates are made, so the index tree is always maintained in a balanced state.

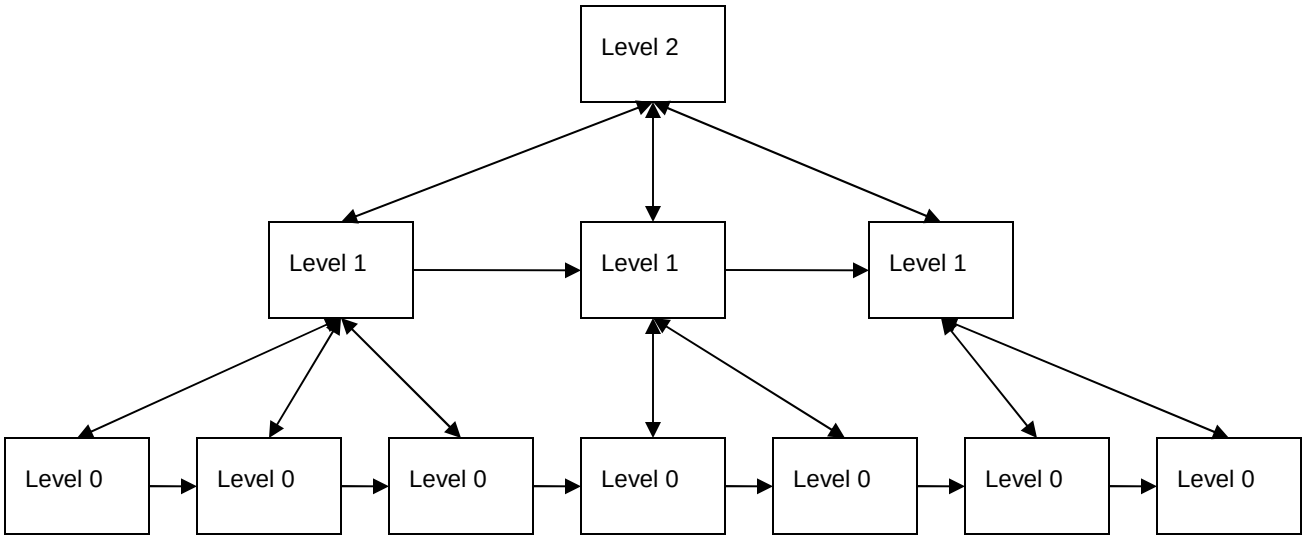

Figure 1. Example of an Inverted Tree Structure for Index Values

HOW DOES THE BASE ENGINE USE AN INDEX?

The primary use of an index is to optimize a WHERE expression. A sequential scan of the data set can be very expensive. Using an index to locate a small subset of the observations can provide substantial performance benefits.

When it receives a WHERE expression, the BASE engine automatically looks for the best index to use. There are some data set options that you can use to override the selection that's made by the BASE engine. Details are discussed in the FAQ section of this paper.

Because the observations that are retrieved by an index are returned in sorted order, any statement that uses BY processing can use an index to obtain the necessary order when the data set is not sorted by the BY variable.

Referential integrity constraints use indexes. When a Primary Key is defined, the BASE engine creates a unique index to ensure that there are no duplicates. When a Foreign Key is defined, a non-unique index is created to enable fast lookup of the Foreign Key values. When you define a UNIQUE integrity constraint, a unique index is defined for that variable.

UNDERSTANDING INDEXING AND SORTING

Q1: Why is the data sorted before the index is built?

A1**:** The index is stored on disk in fixed-size pages. Each node in Figure 1 is a page. If the data is sorted, the code that creates the index does not have to search for the correct place to store the new value, which consumes extra I/O and CPU resources. The new value will always be stored physically adjacent to the last value that was stored. When an index is built with sorted data, each page can be completely filled before going to the next page. Think of this as building the tree sequentially, from lowest value to highest value.

If an index is built with unsorted data, it might be larger than an index that is built with sorted data because all the pages might not be full. Sorting the data takes extra time, but generally results in a more compact index that builds faster.

Q2: How are sort assertion and indexes related?

A2: There are two kinds of sort assertions: weak and strong. A strong assertion is set (that is, stored in the data set) by the SORT procedure. A weak assertion can be set via a data set option if the user knows that the data is sorted. SAS software can query the sort assertion and avoid sorting if the data is already sorted as desired.

PROC SORT will skip the sort if the data set has a strong assertion that the data is correctly sorted.

Index creation and index updating for PROC APPEND will skip the sort if there is a weak or strong assertion. If the assertion is weak, the index code checks the data values to make sure that they are in the correct order.

Updating a data set that has a sort assertion will cancel the assertion. However, when using PROC APPEND, if the base data set and the input data set have the same sort assertion and the new values are larger than the old values, the sort assertion on the base data set is preserved.

Q3: Is there any benefit to indexing a data set that is already sorted?

A3: Yes and no. The benefit depends on your queries. When you use a WHERE expression, the answer is probably.

Without an index, a sequential scan of the entire data set must be performed. Beginning with SAS 8, a strong sort assertion might terminate the full sequential scan early if there is no suitable index. For example, if the WHERE expression is

```
where x = 1;
```

and the data set is sorted by x, when the engine encounters the first observation that contains x > 1, it terminates the scan.

If you are only performing BY processing, then sorting the data set on the BY variable will perform better than creating an index on the BY variable. Note that index creation is faster if the data set is sorted on the to-be-indexed variable.

Q4: Why does sorting in place destroy my index?

A4: Using PROC SORT to sort in-place will physically re-arrange the observations in the data set. Remember that the index contains RIDs that point to the location of observations in the data set. When the observations are physically moved, the RIDs in the index become invalid, and all deleted observations are eliminated from the newly sorted data set. Any indexes that remain after sorting in-place would be unusable because the data set would no longer match the index. Indexes are not preserved by PROC SORT.

Additionally, the structure of a sorted data set can be changed by using a WHERE expression, the OBS= option, the DROP= option, or the KEEP= option. Any of these elements can modify the contents of the sorted data set and, therefore, invalidate any indexes. Similar reasons require that indexes be rebuilt after using PROC COPY.

CREATING AND MAINTAINING INDEXES AND INDEXED DATA SETS

Q5: How much extra disk space will an index take up?

A5: SAS Technical Support has a DATA step program that will provide an estimate of the amount of disk space that is used by an index. A copy of the program is included in the Appendix. This program was written when the only 64-bit platform was Tru64 under UNIX. In SAS releases after Version 6, if you are using a 64-bit platform, specify ALPHA as the host.

You can also get a copy of this program from the Technical Support Web site:
http://support.sas.com/techsup/sample/utilities.html

Q6: When is it necessary to re-create an index for a data set that has observations added, deleted, and modified?

A6: There is no need to re-create the index to restore the tree to an optimum configuration after heavy updating. The index code automatically balances the tree as updates are made.

Q7: Can I create an index in descending order?

A7: No. Remember that the primary purpose of an index is fast lookup of a given value or range of values. SAS indexes are stored in ascending order by default. Returning the indexed values in ascending sort order is a side effect of using an index. There has not been sufficient user feedback to indicate that an option is needed to enable storing the values in either ascending or descending order.

Q8: Is the index compressed when I specify COMPRESS=YES?

A8: No. There is substantial overhead associated with compressing a file. Given that the data set is generally much larger than the index, compressing the index would not provide enough disk savings to justify the cost.

Q9: Is the index encrypted when I specify ENCRYPT=YES?

A9: Yes. Encrypting a data file is a security measure. If the data in the data set needs security, then the data in the index does too. For example, a variable that contains salary information might be indexed. If an index is encrypted, you cannot determine the name of the person who has a specific salary because the observations that are referenced by the index are encrypted. If the index is not encrypted, you can see all the values for SALARY, which might enable you to infer connections between specific salary values and specific employees.

Q10: How do I index a view?

A10: You cannot create an index for a view. An index can only be created for a physical data file. A view represents a virtual data file. There are no observations in a view. The view contains instructions for materializing the observations from physical files or programs. Indexes can be created for the physical files that are referenced by the view, but you cannot create an index for the view itself.

Q11: How can I store the index in a separate folder from the data?

A11: SAS provides no statements that enable you to separate the data set from the index file. On some hosts such as UNIX, you can create a link in the same directory as the data set to the index file that is in another directory. You would have to move the index file and make the link manually after creating the data set and the index.

Q12: When I try to open a data set, I get a message that the data set is corrupted. I can see that SAS is trying to rebuild my index. Unfortunately, I do not have enough disk space to rebuild the index. How can I delete the index that is associated with this data set without first rebuilding the index?

A12: Unfortunately, at this time, you cannot delete the index without rebuilding it. The repair code first deletes the index file, then repairs the data set. If there were indexes, the indexes are then rebuilt as the last step.

Actually, there is an undocumented debug option that will allow a damaged data set to be opened for normal read-only processing. With this option, you can use PROC COPY or the DATA step to make a copy of the data set without the index file. SAS Technical Support can provide this option if you really need it.

A better solution to this situation is being developed. A new value for the DLDMGACTION option will allow the data set to be repaired without the indexes. The metadata that describes the indexes will remain in the data set, and you will be able to reconstruct the indexes at a more appropriate time. Index metadata can also be deleted if you do not want to reconstruct the indexes.

CONSIDERATIONS WHEN USING PROC APPEND WITH INDEXED DATA SETS

Q13: Why do I have to drop and re-create indexes when I use PROC APPEND?

A13: We often recommended dropping and re-creating the indexes in SAS 6, however, many improvements to index creation, especially when using PROC APPEND, were implemented in later releases of SAS. If dropping and re-creating the indexes was inherited from a version 6 SAS application, you can probably remove that part of the code in SAS programs that run later releases of SAS.

In SAS 6, as each observation is added to the data set, all relevant indexes are updated. If the values that are being inserted into the index are not sorted, the index updates take longer.

In SAS releases after SAS 6, by default, observations are appended to the data set as they are sent to the engine from PROC APPEND (as they are in SAS 6), but the index updates are delayed. The indexes are not updated until all the observations are appended. During index creation, the new data to be added to the indexes is sorted before being added to the indexes. If all the new index values are higher than the values that are currently in the index, then the index creation code is invoked to append the new values to the index.

Even if the new values are not higher than the old values, it is more efficient to sort the new values because all the new values are applied sequentially, which prevents the need to re-read the same index page.

In newer versions of SAS, you can use the data set option APPENDVER=V6 to revert to the earlier behavior.

Q14: Sometimes when I run PROC APPEND, adding one observation takes 1 second. I do the same thing again, add just one observation, and it takes 100 seconds. Why?

A14: If your data set has indexes, the most likely answer is a centiles refresh. Centiles are refreshed at given intervals when the data set is closed. When centiles are refreshed, the entire index is read to update the centile values. In this example, the second PROC APPEND might have crossed the threshold for refreshing centiles. The UPDATECENTILES= statement in PROC DATASETS can be used to change the default threshold for refreshing the index centiles.

Centiles (also called cumulative percentiles) are 21 values that are stored in the index metadata that contain the distribution information for the indexed values. Centiles are markers for the 5 percent points. For example, centile 0 is the minimum value, centile 5 indicates that 5% of the values are below this value, centile 10 indicates that 10% of the values are below this value, and so forth up to centile 100, which is the maximum value. Centiles are used to estimate the number of observations that will be selected by the WHERE expression.

Q15: I have an application that uses PROC APPEND to add data to an indexed base data set that has the UNIQUE attribute on some indexes, and it routinely rejects 40% or more of the incoming observations as duplicates. That's OK because it's part of the design. I noticed that, in SAS releases after SAS 6, my base data set is growing a lot faster than it used to. PROC CONTENTS shows that the data set has a lot of deleted records. Can I prevent this?

A15: As part of the PROC APPEND performance enhancement in SAS releases after SAS 6, all observations are unconditionally added to the data set. Then the indexes are updated. A newly added observation that has a duplicate value for a unique index is rejected by the index update process, and the observation is deleted. But the observation (like all deleted observations) remains in the data set and continues to take up disk space.

The easiest way to prevent this duplication is to use the APPENDVER=V6 option in PROC APPEND. This option causes SAS to use the SAS 6 algorithm to update the index at the same time that the observation is added to the data set, and prevents duplicate observations from being added to the data set. However, there might be a performance penalty for using the SAS 6 method of appending.

MORE ABOUT HOW SAS USES INDEXES

Q16: When I access a data set that’s stored on a UNIX disk from my PC, WHERE expressions do not seem to take advantage of the indexes.

A16: The Cross Environment Data Access (CEDA) feature enables you to transparently access data across heterogeneous platforms for reading. The code for CEDA does not support updates and does not support indexes. Adding CEDA support for read-only solved the heterogeneous data access problem for most users. User feedback will determine if support for update and indexes will be added in the future.

Q17: Do integrity constraints use indexes?

A17: Yes, some constraints use indexes. A unique index is needed for a UNIQUE constraint or for a PRIMARY KEY constraint. A non-unique index is needed for a FOREIGN KEY constraint. If no suitable index exists when the integrity constraint is defined, the BASE engine will create an index for you. If you have already created a suitable index, then the integrity constraint creation code will not create another one.

CHECK constraints such as x < 1000 do not use or require an index.

Q18: Is it possible to tell the engine which index should be used?

A18: Yes. The engine attempts to make the right decision about whether it is more efficient to perform a sequential scan of the data set or to use an index for processing a WHERE expression. The engine makes the right decision most of the time. When it does not, the data set options IDXNAME= and IDXWHERE=, which are provided in SAS releases after SAS 6, give you control over which index will be used.

The option IDXWHERE= tells SAS to use the best index (IDXWHERE=YES) or not to use an index (IDXWHERE=NO) when processing a WHERE expression. Note that this option cannot be used to control the use of an index for BY processing. IDXWHERE=YES is the default.

Use the option IDXNAME=*index-name* to tell SAS to use a particular index. Note that SAS disregards your request if use of the index will produce incorrect results. An example would be trying to use a NOMISS index for a WHERE expression that can select missing values.

Q19: The WHERE expression selects all the observations but the index is still used, even though you'd expect sequential processing to be used. Notice that the data set is just one page.

A19: For very small data sets such as this, the WHERE expression costing algorithm is not especially accurate. Using the index results in almost the same speed as a sequential scan. For such a small data set, creating an index is not likely to provide a performance benefit.

Q20: Why didn't SAS use my index?

A20: This is one of the most difficult questions to answer about indexes because the answer depends so heavily on the data. The engine knows how many pages of data are in the data set; this is the amount of data to be read during a sequential pass. That's the easy part. Then the engine must estimate how many pages of data (really, how many I/Os) will be needed to use the index. The index cost is compared to the sequential cost, and the engine chooses the cheapest method.

The engine uses several calculations to estimate the index cost. First, it estimates the number of observations qualified by the index by using the 21 numbers in the Centiles array. The resulting estimate is accurate to within 5%.

Next, the index is used to estimate the "sortedness" of the observations. The engine examines the RIDs that would be encountered on the first index page during the index scan and calculates the number of different data pages that are represented. This number can range from 1 (if the data set is sorted on the indexed value) to the number of RIDs (if the indexed values are widely scattered throughout the data set). An I/O cost per RID is computed from this number and the number of RIDs on an index page. The result is a decimal number that is less than or equal to 1. Smaller is better (less costly).

Finally, the I/O cost per RID is multiplied by the estimated number of qualified observations to get the number of data pages that will be read if the index is used. Comparing this number with the sequential cost determines the access method. Because these calculations rely on estimates rather than exact numbers, and because the cost of sequential versus random I/O can vary from host-to-host, this algorithm might not always pick the fastest access method.

Using the option MSGLEVEL=I can provide helpful hints as to why an index was not used for optimization of your WHERE expression. Here are examples of hints that you might see in the log.

INFO: Index I not used. Sorting into index order may help.

INFO: Index I not used. Increasing bufno to 3 may help.

Q21: Why do observations print in a different order when an index is used?

A21: Without an index, the order of observations is the physical order in which they are stored on the disk. However, with an index, the observations are returned in the order that the observations were found in the index, that is, sorted on the indexed variable. The sorted order of indexed variables is why you can use an index instead of PROC SORT to provide the necessary order for BY processing.

If your application is expecting to receive the observations in physical order, you can use the option IDXWHERE=NO to tell the BASE engine not to use an index.

Q22: Why can't an index be used if there is an OR in the WHERE expression?

A22: The BASE engine is limited to using only one index to optimize a WHERE expression. This limitation means that the selected index must supply every observation that satisfies the WHERE expression. If the index supplies additional observations that do not satisfy the WHERE expression, they will be filtered out later by code that applies the full WHERE expression to each observation.

An OR operation constructs the selected observations by including observations from two different WHERE conditions. To optimize a WHERE expression with an OR operation would require the use of two indexes, one index for each of the conditions.

Some WHERE expressions that contain an OR operator can be transformed by SAS into an IN operator before sending it to the engine. For example,

```
where firstname = 'John' and

   (lastname = 'Smith' or lastname = 'Jones');
```

is transformed by SAS into

```
where firstname = 'John' and

   lastname in ('Smith' , 'Jones');
```

In this case, the engine never sees the OR and can use an index on either FIRSTNAME or LASTNAME for optimization.

Q23: My index was created with the NOMISS attribute. Sometimes it is used for optimization, and sometimes it isn't.

A23: This behavior might be related to the specifics of your WHERE expression. A NOMISS index does not store any information about observations that contain a missing value for the indexed variable. If your WHERE expression can select missing values, then the index will not be used.

Q24: Why is using an index for BY processing slower than PROC SORT?

A24: Using an index for BY processing isn't necessarily slower than using PROC SORT. Performance depends on your data and how you are using it.

If you are applying a WHERE expression and a BY statement that select a small subset of your data, then using an index might be much faster.

However, if your application runs repeatedly, you will incur the overhead of using the index each time the application runs. If you sort the data first, then you have this overhead only once.

Q25: A message was issued that a data set was not sorted properly for BY processing although a composite index should have been selected. How can the index not be in sorted order? The BY clause was "by C", and there was a composite index on the variables I and C.

A25: If a composite index was available in which the first variable that was defined was the variable that was used in the BY statement, then that index is selected to assist in processing. However, if the BY variable is the second variable in a composite index, then that index would not be ordered by the BY variable and cannot help.

You can use the option MSGLEVEL=I; to give you a message in the log to find out which index SAS uses. The message will be similar to the following:

INFO: Index X selected for BY clause processing.

CONCLUSION

An index can be a powerful performance enhancement to your application. However, there are a variety of situations that need to be considered before you make indexes a permanent part of your application. As with most performance-tuning choices, it is best to drive them around the block a few times before you make the final decision.

APPENDIX—SAS JOB TO ESTIMATE THE SIZE OF AN INDEX

```
/*-----------------------------------------------------------------*/
/* Copyright (c) 1996 by SAS Institute, Inc. Austin, Texas.        */
/* NAME:     index_est                                             */
/* AUTHOR:   Billy Clifford                                        */
/* DATE:     15Jun99                                               */
/* SUPPORT:  saswdc - Billy Clifford                               */
/* PURPOSE:  estimate the number of pages required for an index    */
/* UPDATES:                                                        */
/* 27Jul99 wdc correct VAX ALPHA size. Should be 44, not 40.       */
/*                                                                 */
/* NOTES:                                                          */
/* 1. This program is for V7/V8 SAS index files.                   */
/* 2. The algorithm expects all values to be integers. Thus the need */
/*    to use the round function.                                   */
/* 3. This program is based upon the algorithm published in a paper */
/*    written by Clifford, et al, for SUGI 14 - Using new SAS      */
/*    Database Features and Options.                               */
/* 4. This program estimates the size of one index. If your        */
/*    application has multiple indexes, you must run the program   */
/*    once (specifying the characteristics of each index) for each */
/*    index.                                                       */
/* 5. To estimate the size of your particular index, change the    */
/*    values below in the "User-supplied values" section and run   */
/*    the program.                                                 */
/*-----------------------------------------------------------------*/

data _null_;
length host $5;

/*-----------------------------------------------------------------*/
/* User-supplied values                                            */
/*-----------------------------------------------------------------*/
psize = 6000;  /* page size of index file                          */
vsize = 32;    /* total number of bytes in the value to be indexed. */
               /* For a composite index, this is the sum of the    */
               /* constituent variable sizes.                      */
uval  =  3000; /* number of unique values                          */
nrec  = 20000; /* total number of records (ie, observations)       */
               /* NOTE: if uval and nrec are equal, this is assumed */
               /* to be a unique index.                            */
host  = "HPUX";  /* name of host: MVS, CMS, HPUX, SUN, PC, AIX,    */
                 /* ALPHA, VMS, MIPS, MAC                          */
/*-----------------------------------------------------------------*/

/*-----------------------------------------------------------------*/
/* Program-generated values                                        */
/*-----------------------------------------------------------------*/
/*   lpages  -  number of leaf (bottom) index pages                */
/*   upages  -  number of upper level (non-leaf) index pages       */
/*   maxent  -  maximum number of entries on an upper-level page   */
/*   totpgs  -  total number of index pages                        */
/*   bytes   -  total number of pages converted to bytes           */
/*   offset  -  size of offset                                     */
/*   levels  -  number of levels in the index tree                 */
/*   entry   -  the space needed to store a single indexed value and */
/*              all of its RIDs (Record IDs). For a unique index   */
/*              there is one RID per indexed value. For a non-unique */
/*              index there will be unval/nrec RIDs.               */
/*   noperpg -  number of entries per leaf page                    */
/*   cont    -  continuation leaf pages (for non-unique index)     */
/*-----------------------------------------------------------------*/

/*-----------------------------------------------------------------*/
/* Host-specific values                                            */
/*-----------------------------------------------------------------*/
```

```
/* shrtsize - sizeof(short)                                          */
/* header   - sizeof(struct IDXPAGES)                                */
/*            struct IDXPAGES                                        */
/*              {                                                    */
/*              uint32_t pageid;                                     */
/*              long  spare1;                                        */
/*              long  nextpage;                                      */
/*              long  idxid;                                         */
/*              short flags;                                         */
/*              short ctr;                                           */
/*              short nbrentry;                                      */
/*              short free;                                          */
/*              char  level;                                         */
/*              char  spare2[3];                                     */
/*              };                                                   */
/* longsize - sizeof(long)                                           */
/* rsize    - sizeof(struct BASE_RID). This is the size of the       */
/*            pointer (RID) to the record in the data set stored in  */
/*            the index.                                             */
/*-------------------------------------------------------------------*/

if (host eq "ALPHA") then do;
  /* VAX ALPHA host  */
  shrtsize =  2;
  header   = 44;
  longsize =  8;
  rsize    = 12;
end;
else do;
  /* all other hosts */
  shrtsize =  2;
  header   = 28;
  longsize =  4;
  rsize    =  8;
end;

if (uval eq nrec) then do;
  /* Unique index. No offset for a unique index.                    */
  offset = 0;
end;
else do;
  /* Non-unique index. Assume each value occurs nrec/uval times.    */
  /* So there will be nrec/uval RIDs and one value.                 */
  offset = shrtsize;
  rsize  = round( ((nrec / uval) * rsize), 1 );
end;

entry   = vsize + (offset * 2) + rsize;
noperpg = round( ((psize - header) / entry ) , 1 );

if (noperpg lt 1) then do;
   /*---------------------------------------------------------------*/
   /* An entire entry will not fit on a page. This is possible only */
   /* for non-unique indexes. Remember that 'entry' is the size of  */
   /* the indexed value plus all of its RIDs. Here we calculate cont,*/
   /* the number of continuation pages needed for the extra RIDs.   */
   /*---------------------------------------------------------------*/
   noperpg = 1;
   cont = round( (entry / (psize - header)), 1 );
end;
else cont = 0;

lpages = round( uval / noperpg, 1 );   /* number of leaf pages */
if ((lpages * noperpg) ne uval) then lpages = lpages + 1;

lolev  = lpages;
lpages = lpages + cont;
```

```
maxent = round ( ((psize - header) / (offset + vsize + longsize)), 1 );

/*------------------------------------------------------------*/
/* Simulate index creation and count the number of leaf and  */
/* non-leaf pages.                                            */
/*------------------------------------------------------------*/
upages = 0;
levels = 1;
do while (lolev > 1);
   curlev = round( ((lolev + maxent - 1) / maxent) , 1);
   upages = upages + curlev;
   lolev  = curlev;
   levels = levels + 1;
end;

totpgs = lpages + upages;
bytes  = totpgs * psize;

/*------------------------------------------------------------*/
/* All values have been calculated. Create the report.       */
/*------------------------------------------------------------*/

put "=============================================================";
put "  ";
put "Index characteristics:";
put "   Host Platform            = " host;
put "   Page Size (bytes)        = " psize;
put "   Index Value Size (bytes) = " vsize;
put "   Unique Values            = " uval;
put "   Total Number of Values   = " nrec;
put "   Number of Index Levels   = " levels;
put " ";
put "Estimated storage requirements for a V7 or V8 index:";
put "   Number of Upper Level Pages = " upages 8.;
put "   Number of Leaf Pages        = " lpages 8.;
put "   Total Number of Index Pages = " totpgs 8." or " bytes comma14." bytes";
put " ";
put "Note: the above estimate does not include storage for the index";
put "      directory (usually one page) or the host header page.";
put "  ";
put "Estimation of index size complete.";
put "  ";
put "=============================================================";

run;
```

REFERENCES

SAS Institute Inc. 2004. "Understanding SAS Indexes," *SAS Language Reference: Concepts*, SAS OnlineDoc® 9.1.3. CD-ROM. Cary, NC: SAS Institute. Available http://support.sas.com/onlinedoc/913/getDoc/en/lrcon.hlp/a000440261.htm.

ACKNOWLEDGMENTS

Art Jensen
Alec Fernandez
Gary Franklin
Jim Craig
Deanna Warner
Jason Epstein
Charley Mullin

CONTACT INFORMATION

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

Billy Clifford
SAS Institute Inc.
11920 Wilson Parke Avenue
Austin, Texas 78720
512-258-5171 x3254
Billy.Clifford@sas.com