

Building Effective SAS® Indexes using Short and Distinct Keys, Covered Queries and Clustered Indexes

Kirk Paul Lafler, Software Intelligence Corporation

Abstract

An index is an important component that helps improve database performance by accessing a specific observation or set of observations when using a WHERE clause. To help achieve optimal database performance, SAS® users should learn a few simple, and basic, rules associated with building efficient indexes. This presentation examines rules to help users achieve high-performing SAS-based applications. Attendees learn techniques related to constructing short and distinct keys, developing covered queries, and constructing all the information that is needed by a query in a clustered index.

Introduction

Given the large number of books and articles on SQL and SQL-related topics, how strange it is not to find more material related to indexes and their impact on WHERE clause processing. Certainly, these topics deserve additional attention in order to assist SQL users' achieve a greater understanding in applying these powerful features in their database applications.

An index is designed to improve the speed in which subsets of data are accessed. Building an effective index permits, as the index in the back of a book does, a faster way to identify a row or subset of data from a database table. Using an index, a database application has a more efficient method of accessing data, opposed to each observation being processed in a sequential (default) manner. The speed of index processing can become even more compelling as the size of a table increases.

As an added benefit, an index is also able to set up a logical data arrangement without the need to physically sort the data, (as performed with the ORDER BY clause or the BY statement in PROC SORT). Setting up a logical data arrangement can reduce CPU and memory requirements, as well as data access time when using WHERE clause processing. This paper presents elements essential to achieving a better understanding of indexes and their effect on data access.

Tables Used in Examples

The data used in all the examples in this paper consists of a selection of movies that I've viewed over the years, along with its actors. The Movies table consists of six columns: title, length, category, year, studio, and rating. Title, category, studio, and rating are defined as character columns with length and year being defined as numeric columns. The data stored in the Movies table is illustrated below.

MOVIES Table

	Title	Length	Category	Year	Studio	Rating
1	Brave Heart	177	Action Adventure	1995	Paramount Pictures	R
2	Casablanca	103	Drama	1942	MGM / UA	PG
3	Christmas Vacation	97	Comedy	1989	Warner Brothers	PG-13
4	Coming to America	116	Comedy	1988	Paramount Pictures	R
5	Dracula	130	Horror	1993	Columbia TriStar	R
6	Dressed to Kill	105	Drama Mysteries	1980	Filmways Pictures	R
7	Forrest Gump	142	Drama	1994	Paramount Pictures	PG-13
8	Ghost	127	Drama Romance	1990	Paramount Pictures	PG-13
9	Jaws	125	Action Adventure	1975	Universal Studios	PG
10	Jurassic Park	127	Action	1993	Universal Pictures	PG-13
11	Lethal Weapon	110	Action Cops & Robber	1987	Warner Brothers	R
12	Michael	106	Drama	1997	Warner Brothers	PG-13
13	National Lampoon's Vacation	98	Comedy	1983	Warner Brothers	PG-13
14	Poltergeist	115	Horror	1982	MGM / UA	PG
15	Rocky	120	Action Adventure	1976	MGM / UA	PG
16	Scarface	170	Action Cops & Robber	1983	Universal Studios	R
17	Silence of the Lambs	118	Drama Suspense	1991	Orion	R
18	Star Wars	124	Action Sci-Fi	1977	Lucas Film Ltd	PG
19	The Hunt for Red October	135	Action Adventure	1989	Paramount Pictures	PG
20	The Terminator	108	Action Sci-Fi	1984	Live Entertainment	R
21	The Wizard of Oz	101	Adventure	1939	MGM / UA	G
22	Titanic	194	Drama Romance	1997	Paramount Pictures	PG-13

The data stored in the ACTORS table consists of three columns: title, actor_leading, and actor_supporting, all of which are defined as character columns. The data stored in the Actors table is illustrated below.

ACTORS Table

	Title	Actor_Leading	Actor_Supporting
1	Brave Heart	Mel Gibson	Sophie Marceau
2	Christmas Vacation	Chevy Chase	Beverly D'Angelo
3	Coming to America	Eddie Murphy	Arsenio Hall
4	Forrest Gump	Tom Hanks	Sally Field
5	Ghost	Patrick Swayze	Demi Moore
6	Lethal Weapon	Mel Gibson	Danny Glover
7	Michael	John Travolta	Andie MacDowell
8	National Lampoon's Vacation	Chevy Chase	Beverly D'Angelo
9	Rocky	Sylvester Stallone	Talia Shire
10	Silence of the Lambs	Anthony Hopkins	Jodie Foster
11	The Hunt for Red October	Sean Connery	Alec Baldwin
12	The Terminator	Arnold Schwarzenegger	Michael Biehn
13	Titanic	Leonardo DiCaprio	Kate Winslet

Understanding Indexes

Database processing can be extremely disk intensive. This is due in part to the size and number of tables, indexes, views, and other objects within the database, and as the size and number of these objects increases, so typically does the number of disk reads. With an increase in the number of disk reads, throughput often adversely impacts processing.

So, what exactly is an index? An index in a relational database is a data structure that is created to improve data access speeds. It typically consists of one or more columns in a table to uniquely identify each row of data within the table. Essentially, an index is a copy of a database table. Operating as a SAS object, an index is comprised of one or more columns and may be defined as numeric, character, or a combination of both. Although no rule mandates that a table must have an index, when present, they are most frequently used by the optimizer to make information retrieval using a WHERE clause more efficient.

To understand the relevance of an index, it is first necessary to analyze the data within the database tables as well as the queries that access the tables. An index should permit flexibility so every column in a table can be accessed and displayed using the most discriminating column(s) to promote the greatest benefit during query processing. Begin by examining the distribution of values in the column(s) being considered for indexing. Armed with this knowledge, database administrators are better able to minimize the creation and support of "poorly" designed indexes that do nothing more than place inefficient burden on the application itself. To assist in the creation of effective indexes, a brief description of short and distinct keys, covered queries and clustered indexes is in order.

Short Keys

Database processing frequently requires a large number of disk reads when accessing the data in the underlying tables. This becomes particularly more costly when the indexes in a database are represented by lengthy character strings, or larger index keys. It is advisable that database developers and administrators create the shortest keys possible to promote more efficient index comparisons. For example, a table column containing an integer, e.g., an employee number, represents the most efficient index for database processing.

Distinct Keys

Indexes with a large percentage of duplicate values can adversely effect database processing, since data selection is made more difficult during query processing. Equate this to an index in the back of a book on SQL, where the word, "SQL" appears. As you might expect, the number of pages associated with the word "SQL" would be many, resulting in the database not being able to reduce the row results to a select few. This scenario may result in greater demands being placed on processing all without the advantage of being able to reduce the size of "matched" results. Armed with this understanding, it is advisable to construct indexes that are highly selective, and less "ambiguous".

Covered Queries

Covered queries refer to an index that contains all the columns required by a query with the minimum number of disk reads. Essentially, the objective for developing covered query indexes is to improve throughput by allowing the database to retrieve all the information needed for a query in a single disk read. Although this concept represents one approach to query performance tuning, it contradicts the objectives prescribed by having shorter keys.

Clustered Indexes

Clustered indexes are designed to contain all the data represented in a row, allowing the database to have all the data values it needs in a single disk read without the need to perform additional references. Database administrators may create a clustered index for a database that contains static type of data, or data that does not change frequently, because in these scenarios the index does not have to be rebuilt. Contrast this to a dynamic database environment where the data changes frequently – the administrator would be ill-advised to construct a clustered index because it very well could result in increased processing burdens being placed on the database application due to the maintenance of the index.

Simple Rules to Remember About Indexes

When an index is specified on one or more tables, a join process may actually be boosted. The PROC SQL processor may use an index, when certain conditions permit its use. Here are a few things to keep in mind before creating an index:

- If the table is small, sequential processing may be just as fast, or faster, than processing with an index
- Avoid creating more indexes than you absolutely need
- If the data subset for the index is not small, sequential access may be more efficient than using the index
- If the page count as displayed in the CONTENTS procedure is less than 3 pages, an index may not be warranted
- If the percentage of matches is 15% or less then an index may be appropriate
- The costs associated with an index can outweigh its performance value – an index is updated each time the rows in a table are replaced in an add, delete, or modify operation.

Sample code will be illustrated next to demonstrate the creation of a simple and composite index using the CREATE INDEX statement in the SQL procedure.

Building Effective Indexes

An effective index is one that helps improve performance in a database application, particularly when you are looking for a specific record or set of records with a WHERE clause. The guidelines used to build effective indexes include the columns selected from a table to the distribution of data values inside them, how many disk reads are warranted for row or subset lookup, and the amount of disk space used to store an index.

Identifying the Most Discriminating Columns

The process of identifying the most discriminating columns to use for indexing requires careful planning and analysis. Although a database application could have an index created for any and all columns in a table, it would most likely produce a less than efficient processing scenario, particularly in dynamic database environments where there are a greater number of data modifications. Static database environments may be better able to support a greater number of indexes since the number of data modifications would be at a minimum, although this would need to be considered on a case by case basis, and only after careful evaluation.

So is there an approach that can be used to identify and evaluate the columns for indexing? An approach that I personally use, and have had some success with, involves the following steps:

1. Identify and select one or more columns that are currently used in the WHERE-clauses of the SQL query
2. Any column being considered for an index should select the smallest subset of rows possible
3. Use the power of the FREQ procedure to evaluate the distribution of values for any column being considered for an index, keeping in mind the 15% rule
4. Composite indexes should be created so the first column in the index is the most discriminating, the second column the next most discriminating, and so on and so forth.

As illustrated in the following FREQ procedure code and output, the TABLES statement is used to display the distribution of unique values for the movie category (CATEGORY) column. As can be seen in the associated output, only one category value, 'Action Adventure', exceeds the 15% rule presented earlier, with the majority of values well below the subset threshold. As a result, the CATEGORY column could comfortably be used in an index.

The FREQ Procedure

```
proc freq data=movies;
  tables category;
run;
```

Category	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Action	1	4.55	1	4.55
Action Adventure	4	18.18	5	22.73
Action Cops & Robber	2	9.09	7	31.82
Action Sci-Fi	2	9.09	9	40.91
Adventure	1	4.55	10	45.45
Comedy	3	13.64	13	59.09
Drama	3	13.64	16	72.73
Drama Mysteries	1	4.55	17	77.27
Drama Romance	2	9.09	19	86.36
Drama Suspense	1	4.55	20	90.91
Horror	2	9.09	22	100.00

As illustrated in the next FREQ procedure code and output, the TABLES statement is used to display the distribution of unique values for the movie rating (RATING) column. As can be seen in the associated output, three rating values, 'PG', 'PG-13' and 'R', exceeds the 15% rule presented earlier, with the 'G' movies well below the subset threshold, see output below. As a result, the RATING column may not produce the desired results if it were used as a simple index.

The FREQ Procedure

```
proc freq data=movies;
  tables rating;
run;
```

Rating	Frequency	Percent	Cumulative Frequency	Cumulative Percent
G	1	4.55	1	4.55
PG	6	27.27	7	31.82
PG-13	7	31.82	14	63.64
R	8	36.36	22	100.00

In the final FREQ procedure code and output, the TABLES statement is used to display the distribution of unique values in a two-way table using the most discriminating column first, movie category (CATEGORY) and the next most discriminating column next, movie rating (RATING). As illustrated in the following output, each combination of values for category and rating appears to be well below the 15% threshold should a WHERE clause be applied. It can be concluded that the combination of these two table columns, as represented by a composite index, may provide an effective index for further testing and analysis.

The FREQ Procedure

Frequency
Percent

```
proc freq data=movies;
  tables category * rating
    / norow nocol;
run;
```

Table of Category by Rating					
Category	Rating				Total
	G	PG	PG-13	R	
Action	0 0.00	0 0.00	1 4.55	0 0.00	1 4.55
Action Adventure	0 0.00	3 13.64	0 0.00	1 4.55	4 18.18
Action Cops & Robber	0 0.00	0 0.00	0 0.00	2 9.09	2 9.09
Action Sci-Fi	0 0.00	1 4.55	0 0.00	1 4.55	2 9.09
Adventure	1 4.55	0 0.00	0 0.00	0 0.00	1 4.55
Comedy	0 0.00	0 0.00	2 9.09	1 4.55	3 13.64
Drama	0 0.00	1 4.55	2 9.09	0 0.00	3 13.64
Drama Mysteries	0 0.00	0 0.00	0 0.00	1 4.55	1 4.55
Drama Romance	0 0.00	0 0.00	2 9.09	0 0.00	2 9.09
Drama Suspense	0 0.00	0 0.00	0 0.00	1 4.55	1 4.55
Horror	0 0.00	1 4.55	0 0.00	1 4.55	2 9.09
Total	1 4.55	6 27.27	7 31.82	8 36.36	22 100.00

Creating a Simple Index

A simple index is specifically defined for one column in a table and must be the same name as the column. Suppose you had to create an index consisting of movie rating (RATING) in the MOVIES table. Once created, the index becomes a separate object located in the SAS library.

SQL Code

```
PROC SQL;
  CREATE INDEX RATING ON MOVIES (RATING);
QUIT;
```

SAS Log Results

```
PROC SQL;
  CREATE INDEX RATING ON MOVIES (RATING);
NOTE: Simple index RATING has been defined.
QUIT;
```

Creating a Composite Index

A composite index is defined for two or more columns in a table and must have a **unique** name that is different than the columns assigned to the index. Suppose you were to create an index consisting of movie category (CATEGORY) and movie rating (RATING) in the MOVIES table. Once the composite index is created, the index consisting of the two table columns become a separate object located in the SAS library.

SQL Code

```
PROC SQL;
  CREATE INDEX CAT_RATING ON MOVIES (CATEGORY, RATING);
QUIT;
```

SAS Log Results

```
PROC SQL;  
    CREATE INDEX CAT_RATING ON MOVIES (CATEGORY, RATING);  
NOTE: Composite index CAT_RATING has been defined.  
QUIT;
```

Index Entries and Pointers

An index file is stored in the same SAS library as its associated data file. Having the same name as its data file, it is represented as a separate entity known as an INDEX member type. An index file contains entries organized hierarchically with entries being connected by pointers and organized in ascending order. Each entry contains a unique value corresponding to the column's frequency distribution and one or more unique observations, referred to as the record identifier (RID). Space that is occupied by deleted values are automatically reclaimed and reused by the index. A sample index containing entries representing the index file for the movie rating (RATING) is illustrated below.

Value	RID
G	21
PG	2, 9, 14, 15, 18, 19
PG-13	3, 7, 8, 10, 12, 13, 22
R	1, 4, 5, 6, 11, 16, 17, 20

Index Limitations

Indexes can be very useful, but they do have limitations. As data in a table is inserted, modified, or deleted, an index must also be updated to address any and all changes. This automatic feature requires additional CPU resources to maintain an index while processing any changes in a table. Also, as a separate structure in its own right, an index can consume considerable storage space, but this depends on the design of the index. As a consequence, care should be exercised not to create too many indexes but assign indexes to only those discriminating variables in a table.

Because of the aforementioned drawbacks, indexes should only be created on tables where query search time needs to be optimized. Any unnecessary indexes may force the SAS System to expend resources needlessly updating and reorganizing after insert, delete, and update operations are performed.

Examining Dictionary.Indexes

The SAS System generates and maintains valuable information at run time about SAS libraries, data sets, catalogs, indexes, macros, system options, titles, and views in a collection of read-only tables called dictionary tables. Although referred to as tables, Dictionary tables are not real tables. Information is automatically generated at runtime and each table's contents are made available once a SAS session is started.

The contents of Dictionary tables permit a SAS session's activities to be easily accessed and monitored. This becomes particularly useful in the design, construction and testing of software applications because the information can be queried and the results acted upon in a specific task such as in the definition of any defined indexes. Using the Dictionary.Indexes "read-only" table, you can quickly inspect the column name(s) used in an index, the type of index defined, and its name for any table, as illustrated in the following example.

```
proc sql;  
    select memname, name,  
           idxusage, indxname  
    from dictionary.indexes  
    where memname="MOVIES";  
quit;
```

		Column	
		Index	
Member Name	Column Name	Type	Index Name
MOVIES	Category	SIMPLE	Category
MOVIES	Rating	SIMPLE	Rating
MOVIES	Category	COMPOSITE	CAT_RATING
	Rating		

Optimizing WHERE Clause Processing with Indexes

A WHERE clause defines the logical conditions that control which rows a SELECT statement will select, a DELETE statement will delete, or an UPDATE statement will update. This powerful, but optional, clause permits SAS users to test and evaluate conditions as true or false. From a programming perspective, the evaluation of a condition determines which of the alternate paths a program will follow. Conditional logic in PROC SQL is frequently implemented in a WHERE clause to reference constants and relationships among columns and values.

To get the best possible performance from programs containing SQL procedure code, an index and WHERE clause can be used together. Using a WHERE clause restricts processing in a table to a subset of selected rows. When an index exists, the SQL processor determines whether to take advantage of it during WHERE clause processing. Although the SQL processor determines whether using an index will ultimately benefit performance, when it does the result can be an improvement in processing speeds.

Conclusion

Indexes can be used to allow rapid access to table rows. Rather than physically sorting a table, an index is designed to set up a logical arrangement for the data without the need to physically sort it. Not only does this have the advantage of reducing CPU and memory requirements, it also reduces data access time when using WHERE clause processing. As was presented, by adhering to a few important rules about creating indexes, SAS users can use an index and a WHERE clause together to improve a query's performance and processing speeds.

References

- Allen, Scott (2004), *“Effective Indexes using Short Keys, Distinct Keys, Covering Indexes and Clustered Indexes,”* <http://www.odetocode.com/Articles/237.aspx>.
- Lafler, Kirk Paul (2010), *“Hard-to-find and Powerful PROC SQL Features,”* DC SAS Users Group (DCSUG) Meeting (June 10th, 2010), Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2008), *“Kirk’s Top Ten Best PROC SQL Tips and Techniques,”* Wisconsin Illinois SAS Users Conference (June 26th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2008), *“Undocumented and Hard-to-find PROC SQL Features,”* Greater Atlanta SAS Users Group (GASUG) Meeting (June 11th, 2008), Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2008), *“Exploring the Undocumented PROC SQL _METHOD Option,”* Proceedings of the SAS Global Forum (SGF) 2008 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2007), *“Simple Rules to Remember When Working with Indexes,”* Proceedings of the 1st Annual SAS Global Forum (SGF) 2007 Conference, Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2006), *“A Hands-on Tour Inside the World of PROC SQL,”* Proceedings of the 31st Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2005), *“Manipulating Data with PROC SQL,”* Proceedings of the 30th Annual SAS Users Group International Conference, Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2004). *PROC SQL: Beyond the Basics Using SAS*, SAS Institute Inc., Cary, NC, USA.
- Lafler, Kirk Paul (2003), *“Undocumented and Hard-to-find PROC SQL Features,”* *Proceedings of the Eleventh Annual Western Users of SAS Software Conference*.
- Lafler, Kirk Paul (1992-2005). *PROC SQL for Beginners*; Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (1998-2005). *Intermediate PROC SQL*; Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2001-2005). *Advanced PROC SQL*; Software Intelligence Corporation, Spring Valley, CA, USA.
- Lafler, Kirk Paul (2002). *PROC SQL Programming Tips*; Software Intelligence Corporation, Spring Valley, CA, USA.
- Raithel, Michael (2005). *The Complete Guide to SAS Indexes*, SAS Institute Inc. Cary, NC, USA.
- SAS® *Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition (1990)*. SAS Institute, Cary, NC, USA.
- SAS® *SQL Procedure User’s Guide, Version 8 (2000)*. SAS Institute Inc., Cary, NC, USA.

Trademark Citations

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. and other countries. ® indicates USA registration.

Acknowledgments

I would like to thank James Van Campen, WUSS 2008 Databases and Warehouses Section Chair for accepting this paper; as well as Patrick Thornton, Conference Program Chair and Mary Federico Katz, Conference Operations Chair for a great conference!

About the Author

Kirk Paul Lafler is consultant and founder of Software Intelligence Corporation and has been programming in SAS since 1979. He is a SAS Certified Professional, SAS Institute Alliance Member (1996 – 2002), and provider of IT consulting services and training to SAS users around the world. As an author of four books including *PROC SQL: Beyond the Basics Using SAS* (SAS Institute, 2004), he has written more than four hundred peer-reviewed papers, been an Invited speaker at more than three hundred SAS International, regional, local, and special-interest user group conferences/meetings, and is the recipient of 16 “Best” contributed paper awards. His popular SAS Tips column, “Kirk’s Korner of Quick and Simple Tips”, appears regularly in several SAS User Group newsletters and Web sites, and his fun-filled SASword Puzzles are featured in SAScommunity.org and many SAS Newsletters.

Comments and suggestions can be sent to:

Kirk Paul Lafler
Software Intelligence Corporation
World Headquarters
P.O. Box 1390
Spring Valley, California 91979-1390
E-mail: KirkLafler@cs.com

