

Paper 270-29

Improve Your Queries; Hints and Tips for Using SQL

Marje Fecht, Prowerk Consulting, Mississauga, Ontario, Canada

Linda Mitterling, SAS, Cary, NC

ABSTRACT

Are you using PROC SQL but never quite sure your queries are written as well as they could be?

Do you get results different from what you expected?

Do you use SQL options/clauses "because it works" even if you are not sure why?

If you answered yes, this presentation is for you!

This tutorial focuses on improving your SQL queries including:

- understanding the order of execution in SQL
- incorporating functions into queries
- specifying correct joining criteria
- joining tables to produce the desired results.

The target audience is PROC SQL users, on any platform, who want to improve their queries.

INTRODUCTION

This paper is a combination of several miscellaneous tips and techniques, and sometimes common misconceptions that we have come across when teaching or working with SQL programmers. The topics will fall into these general areas:

- performing searches
- joining tables
- using summary functions
- using indexes wisely.

The words "in general" will be used a lot in this paper. The tips and techniques that are discussed generally improve query performance. However, there are many factors that can affect how much or how little performance is actually improved, or if it will be improved at all. For example, variations in data or in the criteria being specified can change the results of a query.

Also keep in mind that different implementations of SQL like SAS, DB2, Oracle, etc. all have their own optimizers. When looking at optimizing or improving the performance of a query, it is best to know how the optimizer in your SQL implementation works, and not to try to generalize about a technique or tip or assume that one technique works efficiently with all optimizers. In fact, a technique that works well with one optimizer might not have any effect with another. Or, it is possible that the technique could cause performance to be even worse in a different SQL implementation.

PERFORMING SEARCHES

A common use of SQL is to perform searches against large volumes of data. There are many operators available in SQL to make searching easy. And, often times, there are several ways to conduct a search. Then the question becomes, which technique is the best to use in a given situation.

The following are tips that are pretty obvious, if you stop and think about it. But, when we are in a hurry writing queries, we do not always stop and think about the best way of doing things – do we? In general, these tips should become common practice when you are writing queries in the future.

SEARCHING FOR SPECIFIC VALUES

In our first scenario, we would like to search a table to find all employees who are pilots. The variable JOBCODE holds this information. Each JOBCODE value starts with a two-character designator like PT for pilots, ME for mechanics, FA for flight attendants, etc. The last part of the JOBCODE designator contains the job level as 1, 2, 3, 4, etc. Note that there are only 1-digit levels in the data. To find all of the pilots we will set up the WHERE criteria using a LIKE operator. We are looking for those rows that have a JOBCODE starting with a 'PT' followed by any one digit.

Possible Solutions:

```
where jobcode like 'PT_'
where jobcode like 'PT%'
```

In this scenario, either an underscore or % sign can be used as a placeholder. The underscore is a placeholder for exactly one character value. 'PT_' says a 'P', followed by a 'T' followed by exactly one character value. If we were looking for two values, we would use two underscores.

The % is a wildcard character. The % character says that anything goes – one, two, three, etc. characters of any kind. So we can have a string that starts with PT and is followed by anything. That way, we do not have to worry at all about how many digits a job level might contain.

Which is the better way to go, or is there even a “better way to go”? Is one type of placeholder or wildcard more efficient than another? In general it is better to use the underscore character rather than using the percent character. When the parser is looking at the first WHERE clause in the example above which uses the underscore, it parses from left to right. It starts by looking for values that begin with the letter P, once it has those, it looks within that result set for those values whose next letter is a T. Then, it looks for exact matches on these characters. Next it matches the underscore with any character in those values that come third. It can complete the search at this point, because it only needs to look for one value as denoted by the underscore.

With the percent comparison, the parser again works from left to right. It would perform an exact match on the letter "P" and then a "T". Then it looks for any third character, and then it would look for any fourth character value, etc. It would have to parse through every digit/character and put them in the result set.

Again, this example assumes that there is only a one-digit level. If you did NOT know this, then you would not even be able to use the underscore and your only choice would be to use the percent sign.

What would happen if the search criteria were reversed? In this case, we would be looking for a set of characters that begin with an unknown string value and end in an exact match of characters. We want to search our data to find any level 1 employees.

Possible Solutions:

```
where jobcode like '__1'
where jobcode like '%1'
```

We can solve this by specifying a LIKE operator with 2 underscores or a LIKE operator with a percent sign. Which is the better way to go? Which would be faster....? As in the last example, the underscore represents greater specificity, so it should, in general be faster. But, again, keep in mind that different data values can cause different results. Take the case with the string values above. They are so short in length that you might not realize as much of a difference between the two techniques then if you were comparing really long string values to one another.

What if in the last example, we wanted to find all level 1 employees but we are dealing with the possibility that there are 1, 2 or 3 characters that come before the job level for each value?

Possible Solutions:

```
where jobcode like '_1'
   or jobcode like '__1'
   or jobcode like '___1'

where jobcode like '%1'
```

How do these two techniques compare? Just as before, in general, it is still better to use specificity with the underscore placeholder. However, keep in mind that there is a point of diminishing returns. The more ORs that you have to connect to specify all possible combinations, the less of a savings you might notice. And at some point, the underscore could actually be the worse of the two techniques to use.

SEARCHING FOR A LIST OF VALUES

In our next search, we would like to search our table for these JOBCODE values: BCK3, ME2, PT2 and PT3. We will use the IN operator to specify the list. Note that the job designators (i.e. PT, ME, etc.) can be 2 or 3 characters long. We can set up the values within the IN list in several different ways.

Many SQL packages will tell you that if you list the values in the order that they most frequently occur, then the lookup process will perform better. Let's try this premise with SAS SQL. In the next example, we know that we have more level 3 pilots than any other values in our list, followed by level 3 baggage check employees, followed by mechanic 2s and then pilot 2s.

Possible Solutions:

```
Specify the values in frequency order:
  where jobcode in ('PT3', 'BCK3', 'ME2', 'PT2')

Specify the values in sorted order:
  where jobcode in ('BCK3', 'ME2', 'PT2', 'PT3')
```

In the first solution, we list the values in order of frequency. In the second solution, we specify the values in sorted order. Which technique is faster?

Surprisingly, in the SAS implementation of SQL, using frequency order should not save any resources, because the WHERE clause processor always sorts the items in the IN list before it processes them. Therefore, the best way to specify your list is in sorted order. Keep in mind that in this example, the list of values is so short, that you probably would not realize any efficiency differences. But if your lists became long, you could notice more dramatic results.

JOINING TABLES

Joining tables is one area where SQL users seem to feel the least "in control" of what is happening. If you are joining tables, chances are good that you have wondered, "Am I really getting the results I want ". Or if you are working with large database tables "Is 5 hours really the best performance I can get with this join".

In the next section, we are going to look at ways that you might be able speed up the processing of your joins. Again, as we go through this section, please keep in mind that we are making general comments and observations and are using examples that lend themselves toward the points that we are trying to illustrate. There are many different factors that can add to the equation of the best way to perform a join operation. But, this section will give you some ideas to experiment with and benchmark. Hopefully they will help you find the best way to approach similar join scenarios that you might encounter along the way.

SPLITTING UP JOINS

First, we are going to show how splitting up a large, complex join into a couple of more simple joins helps improve join performance. To make our discussion of splitting up joins a little easier, we're going to start with a very simple example where we want to join three tables that deal with staff information. Then we will move on to a more complex real life scenario.

JOIN EXAMPLE 1: SIMPLE JOIN

In this first example, we want to join all staff information with salary information into one resulting table. There are two tables for employees: one for existing employees that we call STAFF, and one for any new hires that started within the last month that we call NEWSTAFF. We want to combine the information in the SALARY table with our staff tables. So, we will join all three tables together at the same time using an inner join.

Before we do this, note that the STAFF and NEWSTAFF tables do contain the same variable names and types. And they both have a common key variable named IDNUM. It is a unique identifier between all three tables, including the SALARY table.

Solution #1:

```
proc sql;
  select *
    from airline.staff a, airline.newstaff b, airline.salary c
   where a.idnum = c.idnum OR b.idnum = c.idnum;
quit;
```

Our first solution for this scenario is to place all three tables on the FROM clause in an inner join. Then we will set up our WHERE criteria so that the IDNUMS between the three files match – the IDNUM from the STAFF table matches

the IDNUM in the SALARY table and the IDNUM in the NEWSTAFF table matches the IDNUM in the SALARY table. This gives us the results that we are looking for, but is there maybe a faster or more efficient way that this join can be handled?

In solution #2, we approach the scenario a little differently, by reducing the joins into smaller dimensions. In other words, we perform the join between the STAFF and SALARY tables first. Then we perform a second join between the NEWSTAFF and SALARY tables.

Solution #2:

```
proc sql;
  select *
    from airline.staff a, airline.salary b
   where a.idnum = b.idnum
 outer union corr
  select *
    from airline.newstaff c, airline.salary d
   where c.idnum = d.idnum;
quit;
```

First, the results of the joins are now sitting in two tables. Remember that all of the variables in the STAFF and NEWSTAFF tables are the same. So, the resulting two tables look the same except that one table contains information about existing employees and the other contains information about new employees.

Next, we need to stack these two tables together into one table. To do this, we will use an OUTER UNION CORR. And, if we wanted the results to be in a particular order, we could add an ORDER BY clause at the bottom of the query to ensure that the data is sorted a certain way.

Which technique is faster? Solution #2 can save computer resources, speed up processing and make queries more readable. But again, this is not a hard and fast rule. It is just another technique that you can try to see if it helps. On the flip side, splitting up joins usually means writing a more complex program, which means that there is greater investment of programmer time. Benchmarking several techniques is your best bet to achieve the best join performance. And, if the method that is chosen to split up the query and put it back together is not compatible with the intent of the original query, then you could get incorrect results. For instance, in the last example, if we had used a UNION operator rather than an OUTER UNION CORR, we would have wound up with VERY different results. So, choose your alternate methods carefully.

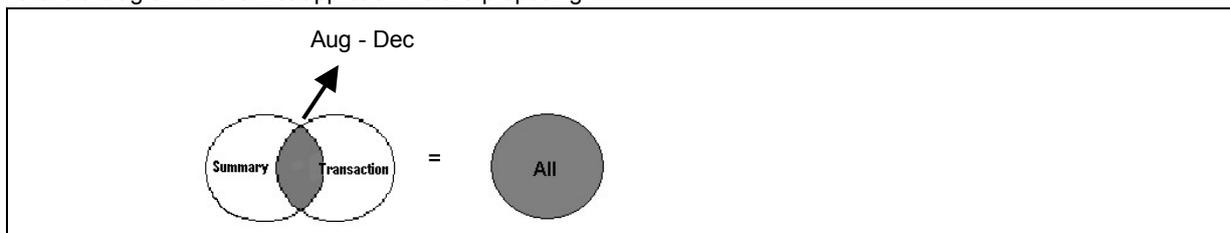
JOIN EXAMPLE 2: COMPLEX JOIN

Let's take a look at a real scenario that we have run across where splitting up a query did in fact speed up our join performance. Keep in mind as we go through this example, that the tables that we were working with were extremely large, on the order of 100 million records.

In Example #2, we have financial summary information on cardholders stored in a monthly summary table. The summary table contains all kinds of information about our customers including balances, credit scores, fees, etc. Another table represents the cardholder's detailed transactions. There are several years of information in both files, and each file has indexes on the variables ME_DT and ACCT_NO. We are interested in getting information for only the months of August-December 2003.

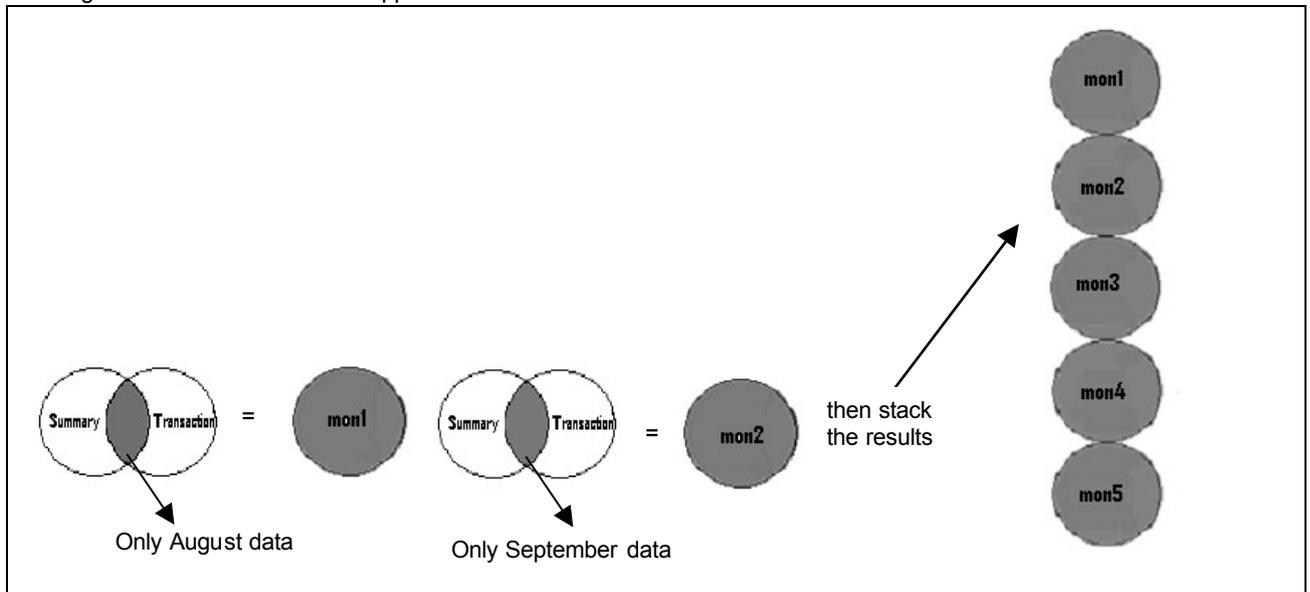
We are going to tackle this scenario in two different ways. One approach is to simply inner join the two tables together. We will extract five months of data from the summary table and join it with 5 months of detailed transactional information for those cardholders. The two tables we are joining are huge.

Here is a diagram of the first approach we are proposing:



A different approach is to join the financial summary and detailed transaction tables together one month at a time, so that we are performing smaller joins. Then we will outer union all of the resulting tables together.

This diagram illustrates the second approach:



The first approach is a good solution for our scenario, and it is the one that would probably come to mind the fastest. In the second approach, however, our aim is to pare down the size of the joins. Instead of one huge join, we are going to ask PROC SQL to perform 5 smaller joins and then OUTER UNION CORR the 5 files together that we saw at the beginning of this section. It turns out that in our real life scenario, approach #2 is more efficient.

Here is the query used for the first approach:

Approach #1:

```
proc sql;
  connect to db2(database = crd &password);
  create table all as select * from connection to db2
    (select sum(a3.bal_current) as SpotBal,
      sum(case when a1.status = 'VOL' then 1 else 0 end) as VolAttrn,
      count(*) as NumAcct,
      a1.me_dt as me_dt
    from crd.cards_fin_sum a1, crd.cards_acct_mth_ a3
    where a1.acct_no = a3.acct_no
      and a1.me_dt = a3.me_dt
      and a1.me_dt in ( &medt )
    group by a1.me_dt
    order by a1.me_dt);
quit;
```

The query for the second approach is shown below. The program only includes the table being created for the month of August. The same program would have to be duplicated for the remaining months.

Approach #2:

```

/*Step #1 - Join one month at a time - August program only shown*/

proc sql;
  connect to db2(database = crd &password);
  create table mon1 as select * from connection to db2
    (select sum(a3.bal_current) as SpotBal,
      sum(case when a1.status = 'VOL' then 1 else 0 end) as VolAttrn,
      count(*) as NumAcct,
      a1.me_dt as me_dt
    from crd.cards_fin_sum a1,
      crd.cards_acct_mth_ a3
    where a1.acct_no = a3.acct_no
      and a1.me_dt = a3.me_dt
      and a1.me_dt in ( '08/31/2001' )
    group by a1.me_dt);
quit;

```

The FROM clause is the same as before since we are still joining the same two tables. But, the big difference is what we are now joining from each of those tables. The WHERE criteria has changed. Instead of asking for all 5 months of data, we are now asking for just the month of August. And, if you look at the top of the CREATE TABLE clause, we are now asking to create a table named MON1— not the ALL table with all of the 5 months of information in it. This resulting table will contain only one month – AUGUST – of information.

We then repeat the same program for the remaining 4 months – SEPTEMBER through DECEMBER. (Just a quick aside... One way that we could get around repeating the same program 5 times is to MACROize our program. But, that should be saved for another discussion.)

After all 5 months of data have been extracted into 5 tables named MON1, MON2, MON3, MON4, MON5, we are ready to stack them together into one resulting table named ALL. To do that, we will perform an OUTER UNION CORR of two tables at a time. You can only OUTER UNION CORR two tables at a time, but you can stack as many steps as you want into one program. So we will OUTER UNION CORR table MON1 with table MON2, and then that with table MON3, etc. Again, the end result will be one table named ALL. Here is the query:

```

/*Step 2: Concatenate after all months are created. */;

proc sql;
  create table all as
    select *
      from mon1
    outer union corr
    select *
      from mon2
      .
      .
      .
    outer union corr
    select *
      from mon5;
quit;

```

Which technique is more efficient for this scenario? In this example, the transaction table has over 150,000,000 records, and on a typical workday is very busy with other queries. Splitting up the joins enables the query to return results in a reasonable timeframe. Staying with one large query put this job behind most others, and results often would return only "after hours" when we would all prefer to be someplace **other than work!**

To summarize what we have learned in this section... Splitting up the processing of a WHERE clause which involves very large tables can potentially help improve performance. You'll just have to benchmark to see if it helps in your particular situation.

SUMMARY FUNCTIONS

In the previous example, notice that the SUM and COUNT functions are used to aggregate values for an entire month within the SQL pass thru portion of the program. This means that the summarization is handled by the database and fewer records are passed to SAS for further processing. An alternative would be to extract all the detail records and handle the summarization on the SAS side.

Which approach is more efficient? In most cases, it is more efficient to use a summary function at the same time as the data extract, which minimizes the records that result in being passed to SAS for processing. In this example, with the high data volume, minimizing the size of the query result is extremely important!

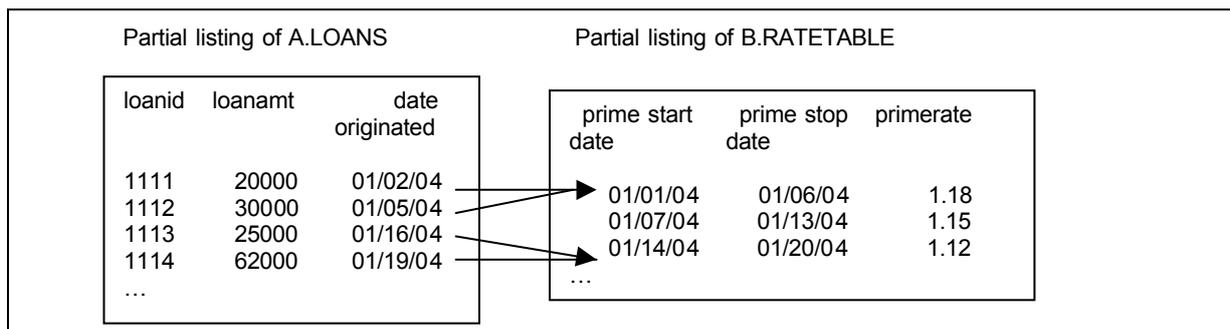
EQUIJOINS VERSUS FUZZY JOINS

An EQUIJOIN is an equality join where the values that you are specifying as joining criteria are an exact match. An example is the following: `where staff.idnum = changestaff.idnum`. The names of the variables do not have to be the same as shown in this example: `where loan.createdate = primecalendar.date`. Again we are looking at a date value in the loan table matching exactly with a date value in the PRIMECALENDAR table.

With a fuzzy join, the matching criterion does not bring back exact matches; but rather, the values that match are approximately the same. An example would be where the value of the variable CREATEDATE in the LOAN table is between some start and stop date in the PRIME table. The resulting values are not exact matches; they do however fall between specified ranges.

The point here is that sometimes changing fuzzy joins to equijoins can speed up query performance because equijoins are able to make use of indexes that have been created on a table. Equijoins provide the SQL query optimizer more options in joining. Therefore, it is often worth the effort to create them.

Here is the scenario. We have a financial institution that would like to determine what the prime rate was at the time that a list of loans was originated. A table called A.LOANS has loan id, the amount of the loan and the date that the loan originated. There are other variables in this table, but we will just be dealing with these in this example. The second table is B.RATETABLE. It contains the dates that certain rates applied. So, it has the prime start date, prime stop date and the prime rates for those date values in this table. For instance, the prime rate between January 1, 2004 and January 6, 2004 was 1.18. Between January 7 and January 13 it was 1.15. Again, what we want to do is to use this table to match up the rates with the dates that particular loans were originated.



For a LOANID value of 1111, the origination date is January 2, 2004. If we look up that date value in the lookup table RATETABLE, we see that January 2, 2004 falls between January 1, 2004 and January 6, 2004 in the lookup table, and the rate at that time according to our lookup table was 1.18. Likewise LOANID value 1112 fell under the same rate. And LOANID value 1114 falls between Jan 14 and January 20, so it will have a rate of 1.12. We want all of this information together in a table. Here is the query that will accomplish our task. This solution left joins the two tables using a fuzzy join.

Solution #1:

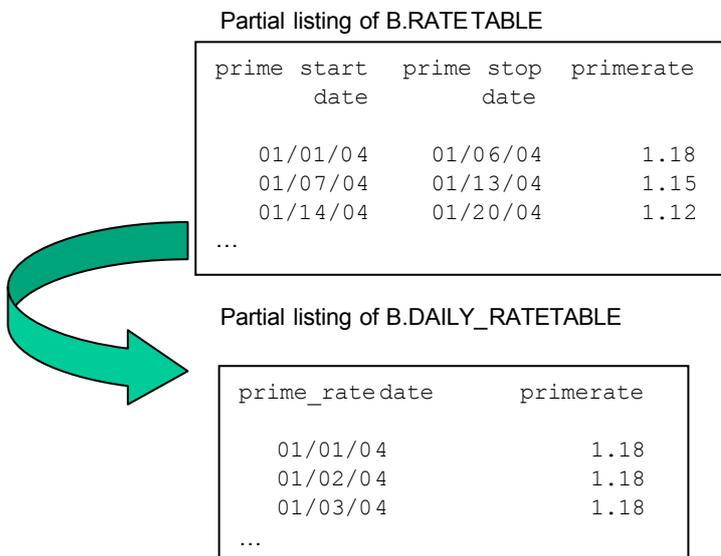
```
proc sql;
  create table combined as
    select loanid, loanamt, loan.date_originated, ratetable.primerate
    from a.loan left join b.ratetable
    where loan.date_originated BETWEEN
      ratetable.primestart AND ratetable.primestop
    order by loan.date_originated;
quit;
```

This is a good solution and one that probably would come to mind pretty fast. But, the way that this query is written, there are limited optimizing routines that PROC SQL can use to process this query.

On the other hand, equijoins provide the SQL query optimizer more options for joining tables. In solution #2, we are going to change our fuzzy join into an equijoin.

Solution #2:

Step 1: Unroll the date ranges in the B.RATETABLE table into a daily rate table.



The way that we will accomplish this is to create a third table to translate the ranges of dates in the prime rate table into daily values. That way, when we use this as a lookup table later for the prime rate values, we can set up our join criteria as an equijoin. For example, remember that the first loan that we had in the LOAN table originated on January 2. With our third daily rate table built, we can ask PROC SQL to find the date January 2, 2004 exactly in the lookup table, instead of having it look at a range of values. We have changed our fuzzy solution into an equality solution.

Once this table is built, we can now change our BETWEEN AND operator in our WHERE clause to an EQUAL sign. Step 2 inner joins the LOAN and DAILY_RATETABLE tables using an equijoin condition or exact match situation.

Step 2: Left join the A.LOAN and B.DAILY_RATETABLE tables using an equijoin condition (an exact match)

```
proc sql;
  create table combined as
    select loanid, loanamt, loan.date_originated,
           primerate
    from a.loan left join a.daily_ratetable
    where loan.date_originated = daily_ratetable.prime_ratedate
    order by date_originated;
quit;
```

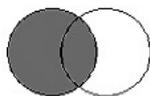
To summarize what we have learned here...Equijoins allow PROC SQL to have more options when it comes to which optimizer it picks when processing a query. Therefore, it is often worth the extra time that it takes to change a fuzzy join into an equijoin situation.

Keep in mind that equijoins can make very good use out of indexes. So, if a table is indexed and the join condition is one that can utilize the index – like an equijoin situation, that can help speed up the processing of your query.

INDEXING WISELY

Almost any SQL book that you read contains a section on using indexes wisely. They talk about uniformly distributed values, physical size of tables, retrieval amounts, etc. But one topic that we came across in our experiences and found to be very interesting, and not well documented, involves creating indexes to be used in join situations where the joins cannot use indexes for processing. Therefore the effort to create these indexes is wasted. Let's look at some examples.

In a LEFT JOIN situation, PROC SQL is going to select all rows that match between the tables. Then it is also going to go back and process all of the remaining rows in the left table, or first table listed on the FROM clause.



In a LEFT join PROC SQL must pick every row in the LEFT table. If there are any indexes built on the LEFT table, SQL cannot use them for a LEFT join. It can make use of indexes on the right table, because only certain rows are selected from the RIGHT table – those that match what are in the LEFT table. But, there is no way for PROC SQL to use an index built on the LEFT table. So, do not waste your time building and maintaining an index on the LEFT table for the purpose of doing a LEFT join.

You would do the reverse for a RIGHT join. There is no way for SAS to use any indexes built on the RIGHT table in a RIGHT join because it has to select all of the rows from the RIGHT table. And, with a FULL join, all of the rows are selected from both tables. Therefore, no indexes can be used.

Inner joins are a different story. Inner joins select only those rows that match between the tables that are being joined. So, under the right circumstance, which is a whole different conversation, indexes may be chosen from either table to speed up an inner join scenario.

To summarize... do not waste time creating indexes on tables that are used in a way that PROC SQL cannot ever select the index for processing – as in indexing a LEFT table in a LEFT join, etc.

CONCLUSIONS

We hope you have enjoyed this compilation of examples. Hopefully we have shown you some concepts in SQL that you might not have been familiar with, or did not quite feel comfortable using. And hopefully, we have given you some tips and techniques that you can experiment with to help improve your queries.

REFERENCES

- SAS SQL Procedure User's Guide, Version 8*. 2000. Cary, NC: SAS Institute, Inc.
SAS Procedures Guide, Version 8, Volume 2. 1999. Cary, NC: SAS Institute, Inc.
Bowman, Judith S. 1996 *The Practical SQL Handbook: Using Structured Query Language*. Reading, Mass: Addison-Wesley Developers Press.
Celko, Joe. 1999. *SQL for Smarties: Advanced SQL Programming*, Second Edition. San Francisco, CA: Morgan Kaufmann Publishers.
Feuerstein, Steven. 1996. *Advanced Oracle PL/SQL Programming with Packages*. Sebastopol, Ca.: O'Reilly & Associates.
Urman, Scott. 1996 *Oracle PL/SQL Programming*. Berkeley, Ca.: Osborne McGraw-Hill.

ACKNOWLEDGEMENTS

The authors appreciate the feedback and advice from

- Lewis Church
- Grant Cooper
- Kent Reeve.

CONTACT INFORMATION

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

Marje Fecht
Prowerk Consulting
Email: Marje.Fecht@prowerk.com

Linda Mitterling
SAS
E-Mail: Linda.Mitterling@sas.com

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

OTHER BRAND AND PRODUCT NAMES ARE TRADEMARKS OF THEIR RESPECTIVE COMPANIES.