Some Strategies for Improving Queries using SQL-Pass Through

Rena Manning, Texas Guaranteed Student Loan Corporation

Abstract

Why is my query taking so long to run? This paper suggests a few answers to that question and then shows ways to improve query performance by taking a journey through Oz. The techniques presented were developed using SAS against (it is a battle, after all) a DB2 database on an IBM Mainframe (z/OS). It is hoped that most of the principles are general enough in nature to be useful in other environments, with appropriate yet minor modifications.

Background Information

Processing Engines

SAS Engine

Data Steps and PROC SQL are frequently used together in SAS programs. PROC SQL is great way to get a quick count, or create a macro variable, for example. When submitting code similar to the example below, even though the code is in Structured Query Language or 'SQL', the processing is still occurring within the confines of the SAS System. Regular SAS conventions, such as SAS date values can be used . In this example, the code is running on data that has already been brought into SAS, regardless of that data's original file format.

```
PROC SQL;
CREATE TABLE ONE AS
 (SELECT          A.ID, A.NAME, B.AMOUNT, B.POSTDATE
FROM              IDS      A
INNER JOIN        BANK  B
ON                A.ID = B.ID
WHERE             '01JAN2007'D LE POSTDATE LE '30SEP2007'D);
```

DB2 Engine

When needing to access data in a DB2 database, another product- SAS ACCESS to DB2 must be used. This software goes beyond BASE SAS and provides a way for BASE SAS to 'communicate' with the DB2 database. Usually a DBA or other Mainframe specialist must set up a connection to the DB2 database before SAS programmers can access the data. Here is the sample code above modified read directly from a DB2 database:

```
PROC SQL;
CREATE TABLE ONE AS SELECT * FROM CONNECTION TO DB2
 (SELECT          A.ID, A.NAME, B.AMOUNT, B.POSTDATE
FROM              IDS      A
INNER JOIN        BANK  B
ON                A.ID = B.ID
WHERE             POSTDATE  BETWEEN '2007-01-01-00.00.00.000000'
                  AND '2007-09-30-23.59.59.999999'
```

Although the code does not look radically different from the first PROC SQL example, an important change has taken place. The SAS Engine is no longer processing the data. Furthermore, the data being processed is not SAS data- data that has been brought INTO SAS and transformed into SAS format. This data is still in DB2 format, and the DB2 Engine is being used to process it. The code that controls this is **SELECT * FROM CONNECTION to DB2**. This code passes control from the SAS Engine to the DB2 Engine, and the DB2 Engine has a component called a Query Optimizer, that is invoked at this point. More on that in a minute.

Because the DB2 engine is now in control of parsing the code, code with SAS date formats needs to be rewritten to code with DB2 date formats, it this case a timestamp. Also, some conventions such as NE or EQ are not acceptable, and need to be replaced with '<>' or '= ' or the word 'BETWEEN'. In other words, 'You're not in Kansas anymore'.

Most programmers have learned to find and modify existing code, rather than programming from scratch. One advantage of SQL is that it works in conjunction with many other languages, in both mainframe and server environments. So it is likely that SQL code exists that can be studied, modified and used. One disadvantage to this situation is that often SQL is written so that all the needed fields are selected in one complex query. Perhaps this is so because it is not as easy to create and discard temporary working data sets in other programming environments as it is in the SAS environment. In any case, the longer and more complex the query, the greater the potential for poor performance.

The Query Optimizer

Pay no Attention... To that query optimizer behind the curtain. Just as the Wizard of Oz wanted everyone to believe in the illusion of the all-powerful OZ, some DB2 manuals would like programmers to trust in the All Powerful Query Optimizer. SAS recommends that programmers pass control to the DB2 Engine as well, to take advantage of the query optimizer feature. So what exactly is a query optimizer and why should it be trusted?

A Query Optimizer is a series of algorithms built into the software that should, in theory, determine the best, most efficient way for the DB2 Engine to retrieve and return all the data that you have asked for. The algorithms rest on suppositions about the structure of the tables in database. The algorithms are written to take advantage of features, such as indices. Sadly, just because a query takes a long time to run, that doesn't mean that the query optimizer is to blame. It could be that the query just doesn't lend itself to any of those key features addressed by the algorithms. Sometimes programmers have to have the courage to look behind the curtain.

How the Query Optimizer Works, sort of

Each table in the DB2 database has one or more **key fields**. Key fields typically hold pieces of identifying information, such as SSN, or something else that answers the question ' Who does this information belong to?' Other fields further categorize information. Imagine a database that describes your wardrobe. Tables would be your closet racks, your chest of drawers, your dresser, your shoe rack. Fields in those tables would list your items, and answer the questions 'what is it – shoe, shirt, sock? Or, 'What style is it- sandal, sneaker, short sleeve, crew sock?' Or 'What color is it, white, black, red etc?'

Some of these properties- item type, style and color are used frequently when we decide what to wear in the morning. So those properties would be useful to consider easily and quickly, and in database terms, they would become part of an **index.**

Think of a big disorganized sock drawer. It's full of both you *and* your significant others white, green, blue, red, and black socks. The socks may either be crew, ankle or calf high. You need the white ankle high pair (to complement the ruby slippers).

```
SELECT * FROM CONNECTION TO DB2
(SELECT      SOCKS
   FROM      DRAWER.SOCKS
        WHERE
        ID              =       'F'
        AND
        COLOR           =       'RED'
        AND
        STYLE           =       'ANKLE' );
```

If you have to rummage through the whole drawer, it may take a while to find that pair. If, however, you had your drawer divided with a grid-like partition, and you had an organized scheme for classifying those spaces by color and style, then you could find your socks quickly by going straight to the designated spot. So the properties **color** and **style** are part of the index, along with the **key field**, ownership.

	white	blue	green	red	black
crew					
ankle					
calf					
His / Hers					
crew					
ankle	X				
calf					
	white	blue	green	red	black

The analogous term for the query optimizer would be a full index scan. In the 'socks' database you might have fields (columns) for color and size. The key would be his or hers. The index would be a combination of the two fields, and the key. There would be 2 X 5 X 3, or 30 boxes, with 30 individual box pointers.

In the next example, the divider between His/Hers is gone. So now the optimizer has to dig through fewer boxes, but each box has more socks. So the query probably takes longer to run. That terminology is similar to a partial index scan.

Everyone's					
	white	blue	green	red	black
crew					
ankle	X X				
calf					

Another example that relies even less on the index:

Just Colors					
	white	blue	green	red	black
all types	XXXXXX				

Working with the Query Optimizer – Some General Tips

Know and use the fields in the INDEX

Knowing the fields that make up the index or indices on the tables is paramount. Often queries run more quickly when all fields in the index are coded for, even if the value of that field is irrelevant to the data criteria. For example – you just want a pair of white socks.

```
SELECT * FROM CONNECTION TO DB2
(SELECT       SOCKS
   FROM       DRAWER.SOCKS
        WHERE
        ID              IN       ('F','M')
        AND
        COLOR           =        'WHITE'
        AND
        STYLE           IN       ('ANKLE, CALF, CREW) );
```

How to find which fields comprise the index depends on tools particular to your DB2 environment, and on proprietary documentation.

Positive Logic

In many DB2 environments, queries run more quickly when **positive** logic is used in the WHERE clause:

```
AND STYLE  IN  ('ANKLE, CALF, CREW) );
```

runs more efficiently than

```
AND STYLE <>  '    ' );
```

No Functions in the WHERE clause

Sometimes using a function in the WHERE clause will cause the query optimizer to be unable to use the index. So using the sock example, if you wanted either crew or calf socks, you should **NOT** code

AND SUBSTR(STYLE,1,1) = 'C');

Know Your Table Sizes

A field may be stored in more than one table. Always read from the smallest table (number of rows) that has all the data that you need, unless reading from the smaller table means forgoing use of an index.

Working with the Database – Some Specific Considerations

The primary purpose of the database I use is to enable collectors to look up and edit information about ***individuals*** quickly. Most of the time, the business and research analysts I work with are asked to extract information about ***groups*** of individuals, and to create reports that summarize data for those groups.

Groups are often defined by database fields that are not part of indices. In that case, the query optimizer has no choice but to read through each row of each table. That is called a **full table scan** and is to be avoided at all costs, as it comes with a high processing cost! Sometimes really creative measures can be taken to avoid full table scans. Corrolary: the slower they query, the more creative the programmer.

The example presented below shows a fairly simple query modified in three main steps. This query has been edited for presentation purposes (the names of the fields and tables have been abbreviated to fit your screen). The modifications dramatically increased the performance time of the query.

1) Use your Brain- Pre-process data to exploit index usage

The database I work with contains a large history table that holds information for phone contact with *individuals.* That table also tracks which employee has contacted the individual. The database also contains a small table that shows which employees (USER_ID) report to which supervisors (SUPERVISOR_ID). The original query used this table as well the large history table to retrieve contact outcome data for a *group* of individuals based on certain values of SUPERVISOR_ID. However, USER_ID is part of the **index** on the large outcome table, whereas SUPERVISOR_ID is not. In this case, it pays to be clever, and to be clever you need ... what the scarecrow needed.

Original Query

```
SELECT
        H.OUTCOME,
        H.USER_ID,
        I.SUPERVISOR_ID,
        A.ADDRESS
FROM
        IDTABLE I
INNER JOIN
        HISTORY  H
ON
        I.USER_ID = H.USER_ID
INNER JOIN
        ADDRESS A
ON
        H.CASE_NUM = A.CASE_NUM
WHERE
        I.SUPERVISOR_ID IN (‘XXJZX’,’XXREM’,’XXJDE’,’XXBHO’,’XXHRC’)
```

One solution would be to list the 10 employee ids associated with each of the 5 supervisor ids. Although that would involve a lot of typing, the actual reason that isn’t the best solution is employee reassignment and turnover rate. A better solution is to use pre-processing steps to create a macro variable containing the 50 employee ids associated with the 5 supervisor ids. The WHERE clause is then modified to use the employee id values contained in the macro, thereby allowing use of the index.

Modified Query

Step 1

```
PROC SQL NOPRINT;
  SELECT DISTINCT  """||(USER_ID)||""""
  INTO: UCODES
  SEPARATED BY ","
  FROM IDTABLE I;
WHERE        I.SUPERVISOR_ID IN (‘XXJZX’,’XXREM’,’XXHJK’,’XXBHO’,’XXHRC’);
```

Step 2

```
SELECT
        H.OUTCOME,
        H.USER_ID,
        I.SUPERVISOR_ID,
        A.ADDRESS
FROM
        IDTABLE I
INNER JOIN
        HISTORY  H
ON
        I.USER_ID = H.USER_ID
INNER JOIN
        ADDRESS A
ON
        H.CASE_NUM = A.CASE_NUM

WHERE
        I.USER_ID IN (&UCODES));
```

2)- Chop that Mega-Query into Manageable Pieces
(Read ONLY the data that you need, WHEN you need it)

One principle taught in the SAS Efficiencies class is to read ONLY the data that you need, WHEN you need it. This is easy to accomplish in traditional data step programming. However, much SQL I have seen comes in the form of giant mega-queries that get every field needed for the final product in one long (and likely long-running) fell swoop. This next tip is brought to you courtesy of the Tin Woodsman.

Building on the contact data example above, suppose mailing address information is needed for ***a few*** of the phone contactees who were just identified, for a follow up mail survey in order to assess customer service.
As you can guess, it is going to be more efficient to look for 100 addresses based on 100 case numbers than it is to pull all the address information for all the contacts in the first place, as was shown above. The new 'chopped-up' process is shown below:

Modified Query

Step 1

```
PROC SQL NOPRINT;
  SELECT DISTINCT  """||(USER_ID)||"""
  INTO: UCODES
  SEPARATED BY ","
  FROM IDTABLE I;
WHERE        I.SUPERVISOR_ID IN ('XXJZX','XXREM','XXHJK','XXBHO','XXHRC');
```

Step 2 (Make this a temporary SAS data set so you can retrieve case numbers)

```
CREATE TABLE CONTACTS AS
SELECT
        H.OUTCOME,
        H.USER_ID,
        H.CASE_NUM,
        I.SUPERVISOR_ID
FROM
        IDTABLE I
INNER JOIN
        HISTORY  H
ON
        I.USER_ID = H.USER_ID
WHERE
        I.USER_ID IN (&UCODES));
```

(additional code for address information has been removed)

Step 3 (Pull a random sample for the desired number of follow-up contacts- as a temporary data set)

```
DATA MAIL (DROP=K N) ;
RETAIN K 100 N;
 IF _N_=1 THEN N=TOTAL;
SET CONTACTS NOBS=TOTAL;
IF RANUNI(0) <= K/N THEN DO;
OUTPUT MAIL;
K=K-1;
END;
N=N-1;
IF K=0 THEN STOP;
```

Step 4 (Create another macro variable holding the case_numbers of the 100 phone contacts)

```
PROC SQL NOPRINT;
  SELECT DISTINCT  """||(CASE_NUM)||"""
  INTO: CCODES
  SEPARATED BY ","
  FROM MAIL ;
```

Step 5 (Retrieve address information for contacts)

```
CREATE TABLE ADDRESS2 AS
SELECT
        A.CASE_NUM,
        A.ADDRESS

FROM
        ADDRESS A
WHERE
        A.USER_ID IN (&CCODES));
```

Step 6 (Merge Information in the)

```
PROC SORT DATA=CONTACTS;
BY CASE_NUM;

PROC SORT DATA=ADDRESS2;
BY CASE_NUM;

DATA FINAL;
MERGE ADDRESS2 CONTACTS;
BY CASE_NUM;
```

3) – Find the courage to divide and conquer

A major limitation of the macro technique described so far is the Pass Through facility/DB2 32K byte transfer limitation for the values of the macro variable. One solution is to create a macro program (this is the part requiring courage, ala the Cowardly Lion) that divides the primary dataset containing the ids into ***n*** smaller data sets. Then an allowable number of ids can be read, loaded into ***k*** macro variables, and passed to a large 'transactional' dataset ***j*** number of times. This technique is described in more detail in several SUGI Papers (included in the references).

Basic Split of the Data

```
%MACRO SPLIT1(NUM);
DATA _NULL_;
IF 0 THEN SET DAARS NOBS=COUNT;
CALL SYMPUT('NUMOBS',PUT(COUNT,8.));
RUN;

%LET M=%SYSEVALF(&NUMOBS/&NUM,CEIL);

DATA %DO J=1 %TO &M; DAARS_&J %END;;
SET DAARS;

%DO I=1 %TO &M;

IF %EVAL(&NUM*(&I-1))<_N_<= %EVAL(&NUM*&I) THEN OUTPUT DAARS_&I;

%END;
RUN;
```

Macro to hold ID's for future use in WHERE clause

```
%DO I=1 %TO &M;

PROC SQL NOPRINT;
  SELECT DISTINCT  """||(CASE_NUM)||"""
  INTO: ACODES_&I
  SEPARATED BY ","
  FROM DAARS_&I;
  QUIT;
 %END;        RUN;
```

Read ‘transactional’ data from a large table for the selected ID’s

```
%DO I=1 %TO &M;
 PROC SQL;
  CREATE TABLE HISTCALL_&I AS
  SELECT * FROM CONNECTION TO DB2
      (SELECT
            CASE_NUM,    USER_ID,
            COLL_ACTIVITY_CODE AS INOUT,
            PLACE_REVIEW_CODE AS RESULT,
            PROM_TO_PAY_IND AS PROMTOPAY,
            PARTY_OUTCOME_CODE AS PARTY,
            DATE(SEQUENCE_NUMBER) AS HISTDATE
       FROM CACS.CO_CASE_COLL_HIST
       WHERE
          CASE_NUM IN (&&ACODES_&I)
        AND  LOCATION_CODE='000001'
        AND  SEQUENCE_NUMBER >=  &DC_DATE1X
        AND  COLL_ACTIVITY_CODE IN ('OC','IC','AC')
        AND  PARTY_OUTCOME_CODE IN ('1','E',' '));
 QUIT;
%END;   RUN;
```

Put all the small datasets back into one large one for the reporting task

```
DATA ALL;
 SET
 %DO J=1 %TO &M;  HISTCALL_&J %END;;
RUN;

%MEND SPLIT1 ;

%SPLIT1(400);
```

Final Words

Perhaps the most important consideration to writing efficient queries is to try to use indices whenever possible. This may require creative thinking on your part, such as adding additional pre-processing steps, or including fields not critical to data selection in a WHERE clause, just because coding for those fields will trigger index usage. Also beware of code in the WHERE clause that interferes with index usage- for example, using date transformations or character functions on fields that are part of an index.

General SAS programming efficiency rules also apply to SQL pass-through coding. Reading ‘ONLY the data you need, WHEN you need it’ was illustrated by the example in this paper, but it is only one of many best practices to be mindful of. Learn to exploit the special nifty PROC SQL features- such as the creation of macro variables using SELECT INTO:. Finally, explore ways to circumvent resource limitations- splitting large datasets into smaller ones is an example of this strategy in action.

Remember, if a query runs for too long, someone is probably going to have to release the flying monkeys!

References

Baum, L. Frank., The Wonderful Wizard of Oz, 1900, George M. Hill Company, Chicago, IL.

SAS Programming Tips: A Guide to Efficient SAS Processing, 1990, SAS Institute, Inc., Cary, NC.

Satchi, Thiru., ‘An SQL List Macro to Retrieve Data from Large SAS/DB2 Databases’, SAS Users Group International (SUGI) 24, 1999

Sridharma, Selvaratnam., ‘Splitting a Large SAS Data Set’, SAS Users Group International (SUGI) 28, 2003