Paper 239-2012

Optimizing that which "… cannot be optimized ….."

Superfast SAS SYSTEM® Searches and Fuzzy Linkage of Large Datasets

Sigurd W. Hermansen, Westat, Rockville, MD, USA

ABSTRACT

Some of the more successful innovations in the SAS System® have percolated up through the SAS-L listserv and other sounding boards for SAS users. This review of large-scale search and fuzzy linkage methods focuses on questions and commentaries that have led to new techniques and methods. In turn, SAS has incorporated some of these innovations into later versions of the SAS System.

KEYWORDS

Fuzzy linkage, SQL, outer join, hash index, object, query optimization, method, search space, Cartesian product, logical decomposition, intersect, conjunctive, disjunctive, union.

INTRODUCTION

Dateline: Fri, 13 Mar 1998 17:52:07
"Self, Karsten" <kself@VISA.COM>
Re: Subsetting very large sasdataset (Related)

Karsten's post of a database search problem on the SAS-L listserv provoked much thought about and discussion of how to optimize searches of databases.

Karsten (the sig line of his messages read "What part of "gestalt" don't you understand?") asked for help in finding: "… ways of restricting the number of potential matches" of key and demographic fields to corresponding fields in a very large database. Specifically,

- *Hash or key numeric fields such that transposes and near-misses are keyed with identical or similar values. Should be suitable for SSN;*
- *Hash or key text fields so that they may be searched readily for similar words and/or text elements. Should be suitable for name and address data".*

At the time, SAS Proc SQL JOIN's followed an optimization strategy that did not differ materially from that of a SAS Data step MERGE: sort datasets being joined on the JOIN key and search the ordered datasets for matches on the key. This strategy has two major limitations. First, of particular concern in 1998, it either fails or takes too long when one or more of the datasets joined are too large to sort efficiently. Second, it does not select a limited number of potentially matching rows in datasets for closer comparison unless the rows match on a specific key value. One would have to sort large datasets many times to search for matches on multiple selection criteria.

Perhaps of more interest retrospectively, Karsten waxed prescient when he suggested

"The ultimate in index/data response would be to include a ramdisk in the search path and place the index on the ramdisk partition – essentially loading it in memory. This technique, along with the use of SAS formats for value lookup, is used extensively in very large database and datawarehouse applications, where sever memory may be measured in gigabytes."

He missed the mark, though, when he discounted the value of homespun methods for optimizing database searches: "Building your own key-value lookup system introduces additional layers of complexity and maintenance, and introduces more opportunities for things to go wrong. I am not saying the technique is never justified or that the results could not improve on what SAS has built in. I am saying that your odds of improving on the available methods, and not breaking your application under the weight of complexity, are slim." Within that same year events would prove him wrong.

Dateline: Tue, 15 Dec 1998 21:15:40
pdorfma@FL6612MAILEX4.UCS.ATT.COM

XMAS SASTip: Quick Table Lookup by Hashing

In one fell swoop, Paul Dorfman demonstrated convincingly that a SAS Data step solution for many basic database search problems performed far better than SAS indexes, "big formats", and SAS Proc SQL. Roll your own "Hashing" not only outperformed standard methods that I and many others considered the best that the SAS System had to offer, this programming feat straight out of Knuth's classic ***Art of Computer Programming, Volume 3: Sorting and Searching*** *was much faster than commercial DBMS static indexes. As we soon discovered, programs that build indexes on the fly and use them to reduce search spaces to manageable proportions would surpass Karsten's fondest dream for a database search method.*

During the next year, Paul developed SAS hash index macroprograms for Westat that continue to be used to link or deduplicate very large datasets. SAS introduced the Java equivalent of a RAM disk, the Hash Object, circa 2004. Many programmers on the bleeding edge made good use of Paul's innovations before SAS had built them into Version 9. The pages that follow will feature examples of SAS programs written by users that have improved on search improvement methods built into SAS and, in some instances, have led to important improvements in the SAS System.

So when the SAS System runs into a roadblock and tells you "NOTE: The execution of this query involves performing one or more Cartesian product joins that can not be optimized." don't despair. Recent history shows that optimizing that "… which can not be optimized …" can and does happen.

THE SCHRIER SOLUTION TO THE FUZZY MATCH PROBLEM

Dateline: Tue, 30 Jan 2001 16:13:44 -0500 howard_schreier@ITA.DOC.GOV
Fuzzy conditional merge on 3 variables

Howard responded to a plea for help with a fishy problem. A researcher was attempting to link geospatial coordinates for fish and water conditions. He hoped to find the closest water condition observation to an observed fish location. He began his search for a solution with theoretically sound, brute force SQL solution that implements the Pythagorean formula for the distance between two coordinates:

```
create table nearest as
select FishID, WaterID,
sqrt((fish.x-water.x)**2
+(fish.y-water.y)**2
+(fish.z-water.z)**2
     ) as distance
from fish, water
group by FishID
having distance=min(distance);
```

The only problem is that for large numbers of observations of fish and water conditions, this Cartesian product solution has to evaluate trillions of possible pairings of fish and water condition observations. Howard painstakingly reduces the search space dimensions from trillions to manageable numbers by placing bounds on the distance between any one fish's location and the location of a water conditions measure. See http://www.nesug.org/proceedings/nesug03/at/at008.pdf *.*

The solution does not matter here as much as the concepts of a search space that, if very large, makes the search for a solution interminateably long, and of a search space reduction method that leads to a more timely solution. The solution combined anticipatory subsetting -that is, computing the maximum and minimum of each of the geospatial coordinates (x,y,z) of fish locations and limiting locations of water condition measure to those close to fish locations – e.g.,

```
select max(x), max(y), max(z),
 min(x), min(y), min(z)
into :maxx, :maxy, :maxz,
     :minx, :miny, :minz
from fish;
create view watersubset as
select * from water
where &MINX-&RADIUS <= x <=
&MAXX+&RADIUS
and &MINY-&RADIUS <= y <=
&MAXY+&RADIUS
and &MINZ-&RADIUS <= z <=
&MAXZ+&RADIUS; ,
```

with offsets. A prescribed radius of a unit sphere around the fish location and a table of all possible cubes that overlap the unit sphere limits the pairings of fish location and water condition locations to a small subset of all possible pairings. A table named offsets, consisting of geographic coordinate values that differ by one unit, combines with a prescribed radius value to

reduce the search space to points at the corners of cubes that enclose the sphere. The offsets table forces links between fish locations and locations of water conditions to a few integer values instead of a multitude of fractional values:

```
create table nearest as
select FishID,WaterID,
(fish.x-water.x)**2
+(fish.y-water.y)**2
+(fish.z-water.z)**2 as
distance_squared
from fish,offsets,watersubset as water
where calculated
distance_squared < &RADIUS**2
  and int( fish.x/&RADIUS)
      +xoffset = int(water.x/&RADIUS)
  and int( fish.y/&RADIUS)
      +yoffset = int(water.y/&RADIUS)
  and int( fish.z/&RADIUS)
      +zoffset = int(water.z/&RADIUS)
group by FishID
having distance_squared
       =min(distance_squared);
```

The best choice of a value of a radius around each fish location depends on the distribution of fish and water conditions measures. Good choices of successively greater radii and the offsets method translates a solution that would take 50+ years to run on a small machine to one that could run in no more than a few hours.

Howard's program demonstrates that when a programmer knows time, space, or other bounds on an abstract search space, one can frame a solution to fit within those bounds. Now try at home a simpler case of anticipatory subsetting. Say that a downtown shopping area has 400 stores and 10,000 people shop there each day. What is the possible number of stores the shoppers will visit? Unconstrained, all possible pairings of shoppers and stores adds up to 4,000,000:

```
data stores;
do store = 1 to 400; output; end;
data shoppers;
do shopper = 1 to 10000; output;
end;
proc sql;
   create table storeShopperPairings as
   select store,shopper
   from stores,shoppers;
```

If we take into account a time constraint on shoppers that limits the expected number of visits per shopper to 40 or less during one day, a better number for an upper limit on the number of store-shopper pairings would be far fewer than four million. Selecting shopper-store pairings at random in sets averaging 40 stores,

```
create table storeShopperPairings as
   select distinct shopper,store
   from stores,shoppers
   where ranuni(23467) < 1/10
   order by shopper ;
```

the number of possible pairings decreases to around 400,000. Further, if we can assume that the absolute difference in store numbers approximates the distance between them, a query captures the distances in a table:

```
create table storeDistancesApart as
   select r1.store as store1,
          r2.store as store2,
    abs(store1 - store2) as distance
   from stores as r1,stores as r2;
```

This table can serve much the same purpose as an offset in that it can be used to constrain a search space. For instance,

```
create table
expectedStoreShopperPairings as
  select distinct shopper,
     store11 as store1,store2,distance
  from (select r1.shopper as shopper,
r1.store as store11,r2.store as store12
from storeShopperPairings as r1,
storeShopperPairings as r2
where r1.shopper = r2.shopper and
r1.store <= r2.store ) as r12
   left join storeDistancesApart as r3
    on store11 = r3.store1
    and store12 = r3.store2
group by shopper
     having distance = max(distance)
order by distance;
```

The query executes in less than a minute with the constraints in place. Without them it ties up a desktop for at least an hour (until I lost patience).

THE BOROWIAK SOLUTION TO THE LEFT JOIN OPTIMIZATION PROBLEM

```
Dateline: Fri, 17 Feb 2006 17:23:05 -
0500 Ken Borowiak
<evilpettingzoo97@AOL.COM>
          For large datasets, skip the
SQL solution for a hash based solution.
```

While a SAS SQL INNER JOIN (e.g., … from R1 inner join R2 on R1.ID = R2.ID …) benefits from hash index optimization and races ahead to a solution, a LEFT JOIN of the same R1 and R2

with the same ON condition crawls behind it. The time required for a hashed INNER JOIN increases proportionally with the number of rows or observations in the larger of R1 and R2. The LEFT JOIN begins by sorting both R1 and R2 and time required for it increases disproportionately as the larger of R1 and R2 increases.

```
proc sql _method ;
create table R3 as
select * from R1
left join R2
on R1.person=R2.person
and R1.id=r2.id;
quit;
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxjm
            sqxsort
            sqxsrc( WORK.SUBSETME(alias=T2))
              sqxsort T..
```

Ken recogized that the SQL left join ON condition either succeeds or fails for each pairing of rows in two datasets.If it succeeds, the join proceeds as if it were an inner join and joined the two rows in the dataset. If it fails,the program joins null values for each RHS dataset variable to values in the LHS dataset.Ken's method indexes a key value and other variable value in one table, and scans the other table for key matches to the index. If the key in a row in the other table matches the index, the program writes data from both datasets to an output dataset. If the key does not match, the program writes data from the LHS table and missing values of variables in the RHS table to the output dataset.

```
/* 'left look-up' using the DATA step
Hash Object - */
data hlj1;
if 0 then set VerySmall ;
declare hash VS(hashexp:7,
dataset:'VerySmall');
VS.definekey('customer','id');
VS.definedata(all:'Y');
VS.definedone();

do until(eof);
       set Big end=eof;
       if VS.find()=0 then output;
       else do;
              call missing(of b1--c);
              output;
            end;
end;
stop; run;
```

Note that, when an attempt to find a key in Big that matches the same key in VerySmall fails, the call of the missing function sets values of variables in VerySmall to missing before output'ting the row containing the key value.

The Borowiak article in http://www.nesug.org/proceedings/nesug06/dm/da07.pdf. Ken concludes that typical large-scale LEFT JOIN's take 30% - 40% longer to run than its hash object mimic.

A LEFT JOIN OPTIMIZATION BASED ON A LOGICAL DECOMPOSITION

```
Dateline: Sun, 16 Nov 2003 3:55 PM
tin-shun-jimmy chan
SAS-L@LISTSERV.UGA.EDU
Subject: merging two dataset
```

Even though Jimmy Chan's question had to do more with why a LEFT JOIN could yield more rows than found in the LHS dataset (answer: when the RHS dataset has enough multiples of key values matching LHS key values in an ON clause), the extended answer to the question provides some interesting insights into how workarounds work well under some conditions but not others.

We know from the prior section that the SAS SQL compiler does not select a potentially more efficient method for executing the LEFT JOIN query, but it does for the INNER JOIN query. All the more puzzling because the LEFT JOIN logically breaks down to an INNER JOIN query and another query or queries…. This basic Venn Diagram illustrates how a set of key values in a LHS dataset might intersect with a set of key values in a RHS dataset:

Figure 1: Key sets

The yield of a LEFT JOIN includes the rows identified by keys in the cross-hatched intersection of LHS and RHS, plus those identified by the set complement of the RHS: that part of the LHS not in the LHS – RHS intersection. (This simple description doesn't take into account multiples of the key values in the datasets.)

Now suppose that we separate out the LEFT JOIN intersection potentially to take advantage

of the efficiency of a hash index. Should the RHS dataset keys fit easily into memory, a LEFT JOIN will execute at close to the time required to read the LHS dataset.

```
   create table intersect as
   select R1.customer as
customer,R1.ID as ID,R2.c as c
   from vwBig_keyed as R1 inner join
        vwSubsetKeyed as R2 on
R1.key=R2.key;
```

The LEFT JOIN solution now requires a UNION of intersect and the rows of Big that do not have one of the key values in intersect.

```
   create table solution as
   select * from
   (select * from intersect)
        outer union corr
   (select * from Big
      where
compress(put(customer,z8.))||compres
s(put(ID,z8.)) NOT IN
      (select
compress(put(customer,z8.))||compres
s(put(ID,z8.))
       from intersect)
    );
```

Borowiak's hash solution to the LEFT JOIN tends to perform better than the standard LEFT JOIN query or the queries that implement a logical decomposition of the LEFT JOIN. Should the platform have less memory available than required by the hash solution, it will fail whereas the standard and alternative LEFT JOIN's will succeed eventually. Between the two SQL query methods, the standard query works faster when sorting of the LHS and RHS datasets takes less time than it takes to read the LHS dataset twice.

For example, the alternative method may work faster on wider datasets that take longer to sort. In one test based Borowiak's synthetic data with number of variables ratcheted up to ten times the original number, the standard method took just over 30 minutes of elapsed time on a standard Windows desktop, while the alternative method took just under 13 minutes on the same machine.

A SOLUTION TO A DISJUNCTIVE (OR) QUERY OPTIMIZATION PROBLEM

```
Dateline: 30 Nov 2000 14:19:33 GMT Perry
Bratis pbratis@MY-DEJA.COM
Merge/Join Efficiency
```

Perry posted a request for help with the task of making a query of this form run faster:

```
create view vwsuball as
select soundex(LN) as sLN, soundex(FN) as sFN,
     substr(LN,1,3) as LN3, substr(FN,1,3) as FN1,
     LN, FN,DOB,SSN,sex
from LNSUBS.submissionall ;
create view vwLNSub as
select soundex(LN) as sLN, soundex(FN) as sFN,
     substr(LN,1,3) as LN3, substr(FN,1,3) as FN1,
     LN, FN,DOB,SSN,sex
from LNSUBS.lexisnexis_submission ;
/* Multiple disjunctive condition query. */
create table LNSUBS as
select * from vwsuball as R1 full join vwLNSub as
R2
    on  R1.SSN = R2.SSN
    OR (R1.sLN = R2.sLN and R1.sFN = R2.sFN
and R1.sex = R2.sex)
    OR (R1.LN3 = R2.LN3 and R1.FN1 = R2.FN1
and R1.DOB = R2.DOB)
    OR (R1.FN = R2.FN and R1.DOB = R2.DOB) ;
```

When fed very large datasets, this form of program takes an excessive amount of time to execute, if it actually terminates normally at all. The SAS SQL compiler often politely advises that the query cannot be optimized. The OR condition(s) in the ON clause specifies different sort or index keys. In a sense the query is asking the compiler to conduct independent searches and to combine the results of each. While similar in concept to the fuzzy fish linkage, this problem does not have a geographic distance to minimize when searching for a nearest neighbor.

Initial VIEW's, vwsuball and vwLNSub, define the same attributes and functions of attributes in each of two datasets. With these VIEW's as a starting point, a hash object program offers an alternative to what the SAS SQL compiler tells us cannot be optimized.

Another VIEW, vwhashMatches, builds four indexes of identifiers and stores keys and other attributes of the vwLNSub virtual table in memory. New to SAS V9.2, the multidata keyword prompts the program to index multiple instances of keys (definekey) and their related attributes (definedata). The program indexes four alternative keys and data read from vwLNSub:

```
/* Based on Dorfman (2007) and Ray - Secosky
(2008). */
data vwhashMatches (keep=SSN SSNM LN LNM FN
FNM DOB DOBM sex sexM)
    / view=vwhashMatches;
          if 0 then set vwLNSub ;
    dcl hash hk
(dataset:'vwLNSub',multidata:'y',hashexp:4) ;
      hk.definekey   ('SSN') ;
      hk.definedata  ('LN','FN','DOB','sex', 'SSN') ;
          hk.definedone () ;

    dcl hash hh
(dataset:'vwLNSub',multidata:'y',hashexp:4) ;
    hh.definekey   ('sLN','sFN','sex') ;
    hh.definedata  ('LN','FN','DOB','sex', 'SSN') ;
       hh.definedone () ;

    dcl hash hi
(dataset:'vwLNSub',multidata:'y',hashexp:4) ;
    hi.definekey   ('LN3','FN1','DOB') ;
    hi.definedata  ('LN','FN','DOB','sex', 'SSN') ;
       hi.definedone () ;

    dcl hash hj
(dataset:'vwLNSub',multidata:'y',hashexp:8) ;
   hj.definekey   ('DOB','FN') ;
   hj.definedata  ('LN','FN','DOB','sex', 'SSN') ;
       hj.definedone () ;
```

With keys indexed and data linked, the program begins scanning the virtual table vwsuball. For each row in the second dataset, it assigns variables in vwsuball to new variables. The h.find () method next looks up the key values on each of the four indexes, and, for each found (rc=0), it assigns indexed values to variables named in the argument list of each h*.definedata method* ***('LN','FN','DOB','SEX', 'SSN')****, and outputs values of those and the other variables in vwsuball to the virtual table vwhashMatches. The program then uses the h*.has_next(result:r) method to search for any other key values in the indexes and, if found, outputs values of indexed data variables and other variables in vwsuball to vwhashMatches.*

```
    do until (eof2) ;
      set vwsuball end = eof2 ;

          LNM=LN;
          FNM=FN;
          DOBM=DOB;
          SSNM=SSN;
          sexM=sex;

          /* h*.find loop.. */
          rc = hk.find ();
          if (rc = 0)
           then do; output ;
                hk.has_next(result: r);
                do while(r ne 0);
                  hk.find_next();
                  output;
                  hk.has_next(result: r);
                end;
              end;
          .... < Repeat  loop using hh, hi, and hj
indexes.>
    end ;
run ;
```

The VIEW's finally materialize in a SQL query that combines selected data from the two sources, "scores" the linked pairs of identifying values, and creates a table of values from linked rows:

```
/* Combine hash search results, compare pairings of
identifiers, and compute overall similarity scores. */
   create table LNSubs.DeDupd as
   select distinct sum(     max( (SSN = SSNM),
                    0.2 * %spedis(SSN,SSNM) ),
                    0.4 * %spedis(
%fixSuffixDash(LN),%fixSuffixDash(LNM) ),
                    0.3 * %spedis(
%fixSuffixDash(FN),%fixSuffixDash(FNM)  ),
                    0.3 * (DOB = DOBM)
       ) as Score, max(calculated Score) as maxScore,
SSN,SSNM,LN,FN,LNM,FNM,
         DOB,DOBM,sex,sexM
    from vwhashMatches
    where Calculated Score >= 1
    group by SSN having maxScore >= 1
                    and Score=maxScore
    order by maxScore descending,SSN,Score
descending;
```

The weighted sum of different match measures reflects to some extent the similarity of a pair of identifiers and the importance of that pair of identifiers in a correct match and in a non-match. Note that logical equality [e.g., (SSN = SSNM)] and a fuzzy measure (a function of a SAS SPEDIS() comparison) both require two arguments and, in this context, one value from each of the data sources. Note also that the SQL query does not call for a join of the two data sources. So where does the pairing occur?

Recall that the h.find () and h*.find_next () methods look up a key value and, if found, link through that key to data values pushed into a satellite index by the h*.definedata method. These indexed variables become available alongside rows of variables SET from data source two.*

Multidata, disjunctive (OR), hash object indexes and searches closely approximate a multiple disjunctive condition query of the type presented at the beginning of this section. That query joined all pairings of rows in vwLNSub (n=300,258) and vwsuball (n=564,113) subject to OR conditions into a table of 27,034,304 rows (a small fraction of the trillions of possible pairings of rows in the two tables). It took almost 5 hours to run under SAS 9.2 on a fast Linux server. The hash object program and "scoring" query imposed additional score constraints at the row-pair level and the SSN group level. For the same two data sources, it generated 285,900 linked pairs, much closer to the expected number of matching records. The program took less than 17 minutes to run on the same Linux SAS server.

CONCLUSIONS

A variety of large scale database search and fuzzy linkage problems have better solutions than those that the SAS SQL compiler can find, or than those that programmers can easily find on their own. Some problems have naïve solutions that would take far more than a reasonable amount of time to run. Many fuzzy matching problems fall in this group. A bounded search method such as that presented by Schrier reduces search space to manageable dimensions.

Some data linkage problems have better solutions that lie outside the scope of the SAS SQL compiler or the Data step. SAS developers have to put a premium on methods that work reliably across many different contexts, and avoid methods that may fail when applied by programmers with less experience. Borowiak demonstrates that a method with a somewhat higher risk of failure may, in the hands of a skilled programmer, lead to better solutions. An alternative LEFT JOIN method takes advantage of knowledge of logical components of outer joins. Moreover, a simple extension of SAS hash object techniques explored by Secosky and Ray approximates another of those naïve solutions, disjunctive outer joins, that cannot be optimized by the SAS SQL compiler. SAS-L listserv participants and other user groups interact with SAS developers to improve both the SAS System and the experience of using it.

REFERENCES

H. Schreier Picking Up Where the SQL Optimizer Leaves Off, NESUG 16, Arlington VA, 2003
K. Borowiak A Hash Alternative to the PROC SQL Left Join, NESUG, Philadelphia, PA 2006
P. Dorfman, G. Snell. Hashing: Generations. SUGI 28, Seattle, WA, 2003.
P. Dorfman, L. Shajenko. Data Step Programming Using the Hash Objects, NESUG, Baltimore, MD 2004.
R. Ray, J. Secosky Better Hashing in SAS® 9.2, SGF, San Antonio, TX, 2008

ACKNOWLEDGMENTS

Paul Dorfman, Michael Raithel, Stan Legum, and Mike Rhoads suggested improvements of content and presentation; despite their best efforts, the author has sole responsibility for any errors or oversights.

DISCLAIMERS

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

CONTACT INFORMATION

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

Sigurd W. Hermansen
Westat
1650 Research Blvd.
Rockville, MD 20850
Work Phone: 301.251.4268
E-mail: hermans1@westat.com