at008

Picking Up Where the SQL Optimizer Leaves Off

Howard Schreier, U.S. Dept. of Commerce, Washington DC

ABSTRACT

Structured Query Language (SQL) is an extremely versatile and useful component of Base SAS® software. With SQL, one can sometimes express the solution to a problem in one code statement. Unfortunately, in some cases that code is prohibitively inefficient. This paper is a case study showing ways in which the inefficient code can be modified and tuned to the extent that a solution can be computed.

GENESIS AND DESCRIPTION OF THE PROBLEM

The problem we will examine was brought to my attention by a researcher's inquiry on SAS-L in early 2001. The research project had gathered data on the movement of individual fish through a body of water, and also on conditions in the water. Both the fish-location observations and the water-condition observations incorporated spatial coordinates. The task at hand was one of matching, to find the closest water-condition observation for each fish-location observation.

The problem is illustrated, in two dimensions, in Figure 1. The data actually provide coordinates in three dimensions, and the code presented in this paper treats three dimensions, but the diagrams throughout the paper are in two dimensions; this is sufficient to present the concepts.

In Figure 1, each red dot represents a fish data point and each blue triangle represents a water data point. So the task is to find the closest triangle to each dot. There is a circle drawn around each dot, passing through the closest triangle. So each of these circles has a dot at its center, with one triangle along its circumference but no triangles in its interior. Note that in fact ties are possible, though not illustrated here; a tie would be characterized by two or more triangles on the circumference of a circle.

There some things to notice. First, although each dot must be matched to exactly one triangle, a triangle can (as explained above) be matched to two or more dots, or to no dots at all. Second, the analysis and solution for any one observation in the fish dataset is unrelated to the analysis and solution for any other observation in the fish dataset.

So much for the concept. The real problem of course is presented numerically, to be solved numerically. In both datasets, each observation consists of an identification (ID) variable and three cartesian coordinates (X, Y, and Z).

Regarding terminology: I use traditional SAS terms more or less interchangeably with relational database terms. So a dataset is equivalent to a table, an observation equivalent to a row, and a variable equivalent to a column. When referring to the data involved in the problem at hand, I use “point” to refer to an observation from either the water dataset or the fish dataset and “pair” to refer to one of each.

NAIVE SOLUTION

There is a rather simple yet correct SQL solution to find the best (closest) matches:

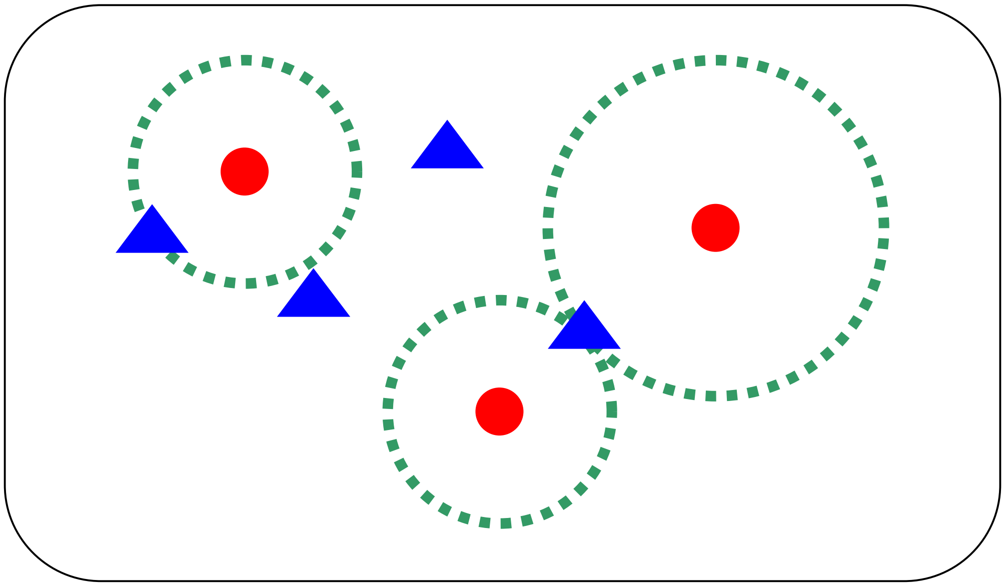

Figure 1. Red dots represent fish points. Blue triangles represent water points. The problem is to identify the water point closest to each fish point.

```
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 expression which populates the DISTANCE column is the pythagorean formula for the distance between a fish point and a water point. The FROM clause joins the two tables; that is, it pairs each row in the fish dataset with each row in the water dataset. The GROUP BY clause serves to separate and compartmentalize the analyses for the individual fish data points. The HAVING clause stipulates that among all of the pairwise distances involving a particular fish point, the smallest is wanted.

The actual datasets include 650 thousand observations on fish location and 30 million observations on water conditions, so the issue is one of performance. To solve the problem, the SQL processor must form the Cartesian product which pairs each fish data point with each water data point and calculate all pairwise distances before it can proceed to determine the set (one for each fish data point) of minima. So the "front end" of the query (the join) will yield nearly 20 trillion pairings to be processed through the "back end" of the query (the summarization). To gauge this, I ran the query against small fractions of the data. Processing time increased, as expected, with the size of the tables. The largest of my tests used exactly one tenth of one percent of each of the two tables; in other words, one millionth of the total volume which would be generated in crossing the complete tables. It took nearly a half hour to process this, on a 1.6 megahertz Pentium 4 computer system with 512 megabytes of RAM. A quick back-of-the-envelope extrapolation suggests that the full problem would take nearly a half million hours, or 50+ years.

Moreover, another limitation arises: the temporary disk footprint of the query. The intermediate result passed to the back end of the query is written to a file in the WORK library, one which is managed internally by SQL and is ordinarily transparent to the user. The entire yield of the join (around 20 trillion rows) would require on the order of 3 **petabytes** (3 million gigabytes) of disk storage, so obviously the query would fail after processing only a tiny fraction of the data.

Of course I was running these tests on a fairly ordinary office computer, circa 2002. On a more powerful hardware platform, in particular with a high-performance disk system, and with some partitioning of the fish table, the resource requirements would be less daunting. But it would still be prohibitively expensive to proceed.

OPTIONS

We are dealing with what is known in the computer science field as the "Nearest Neighbor" problem; see Gionis. PROC SQL offers what is a fine solution in theory, but not in practice. So what are the options?

We can look elsewhere in SAS for a more efficient solution. SQL is after all a very general tool suitable for use on a wide variety of problems. Typically, the more specific a software tool is, in terms of the tasks to which it is applicable, the more opportunity there is to optimize. However, neither SAS/OR® nor SAS/STAT® provides an efficient solution. PROC FASTCLUS does provide a mechanism for identifying a nearest neighbor data point, but it is even less scaleable than the naive SQL solution which we already have.

We can look outside of SAS. See Skiena. Given a suitable tool, we could export the data from SAS, run the specialized software to solve the problem, then import the results for subsequent processing with SAS.

Another possibility is to find an efficient algorithm, then implement it within SAS, perhaps using the DATA step language.

Finally, we might modify the code in ways which permit SQL to employ some of its optimization features and to implement a few shortcuts which go beyond those capabilities. That is the course which we will follow, and the remainder of this paper explores some possibilities.

PRELIMINARIES

We can begin with a mundane bit of optimization. The formula for the distance includes the calculation of the square root of the sum of squares. But that is a monotonic function, so we can work in terms of squared distance without affecting the results. If the distance as such is needed, it can be calculated later for the selected pairings only. Meanwhile, the query becomes:

```
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, water
 group by FishID
having distance_squared
=min(distance_squared);
```

where the highlighting indicates the changes.

We can also get rid of duplicate data points (those having distinct ID values but identical coordinates). That shrinks the problem a bit, and the duplicates can always be reintroduced at the end if they are needed. However, the datasets actually have relatively few such duplicates, so I have not bothered to carry out this process. Note that deleting duplicates from the input data does not eliminate the possibility in the output of multiple matches for a given fish data point. It's possible for a fish data point to be equidistant from two (or more) **distinct** water data points.

Finally, we can divide the fish dataset into any number of pieces and process each one separately. This can make the overall computation process more manageable, and it certainly can help reduce the disk footprint. But fundamentally it just rearranges the huge join into a number of somewhat less huge joins without reducing the aggregate size. I call this "parallel partitioning" and will make use of it later.

These measures are very limited and can at best be just part of the solution to the scale problem we confront. Without more powerful techniques, the needed results cannot be computed using PROC SQL.

VARIANT: UPPER BOUND ON DISTANCE

Consider a variant of the given problem, one which will ultimately be useful in devising a solution. Specifically, recall that the requirement in the original problem is to find the nearest water point for each fish point, no matter how far away it is, and suppose instead that we are to find the nearest water point only if there is such a point within some specified distance. In other words, assume that there is some upper bound stipulated.

The modified problem is illustrated in Figure 2. The circles drawn around each fish point (red dot) all have the same radius (10 units, for example). One fish point is matched to a single water point (blue triangle), one is matched to two water points, and one is not matched to any. The SQL code for the problem with the upper bound is:

```
  %let radius=10;
```

```
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, water
 where calculated
  distance_squared<&RADIUS**2
 group by FishID
 having distance_squared=
    min(distance_squared);
```

The added WHERE clause does in fact reduce the number of data point pairs which are eligible for consideration when the HAVING clause is processed. But because the expression for DISTANCE_SQUARED involves data from both input tables, that happens "downstream" from the join; SQL will still have to step through all possible fish-water pairings. Also, keep in mind that the addition of an upper bound changes the results of the query, so even if there were a performance gain, the modified query does not provide a correct solution to the original problem. However, the addition of this WHERE clause stipulating a maximum distance does provide some opportunities for increasing the efficiency of the query.

With an upper bound on the distance, we may be able to to shrink the problem. If a water point is more than 10 units from **every** fish point, then it is not going to be matched to **any** fish point and will have no presence in the result set. Consider Figure 3. Suppose that all of the fish points (red dots) lie within the inner (dotted-lined) rectangle. The green-shaded, round-cornered region represents a margin of 10 units surrounding the inner rectangle. Any water point (blue triangle) lying outside that margin is thus extraneous, given the upper bound on the distance measure. The same is true with respect to the square-cornered, solid-lined rectangle, which is a mathematically

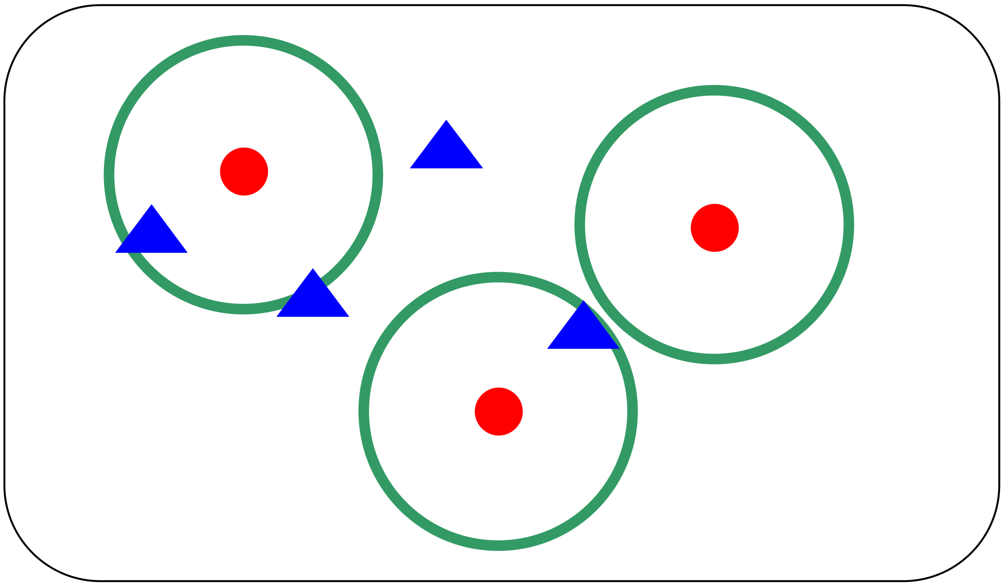

Figure 2. Red dots represent fish points. Blue triangles represent water points. Circles have the same radius. The problem is to identify the water point closest to each fish point within the specified radius.

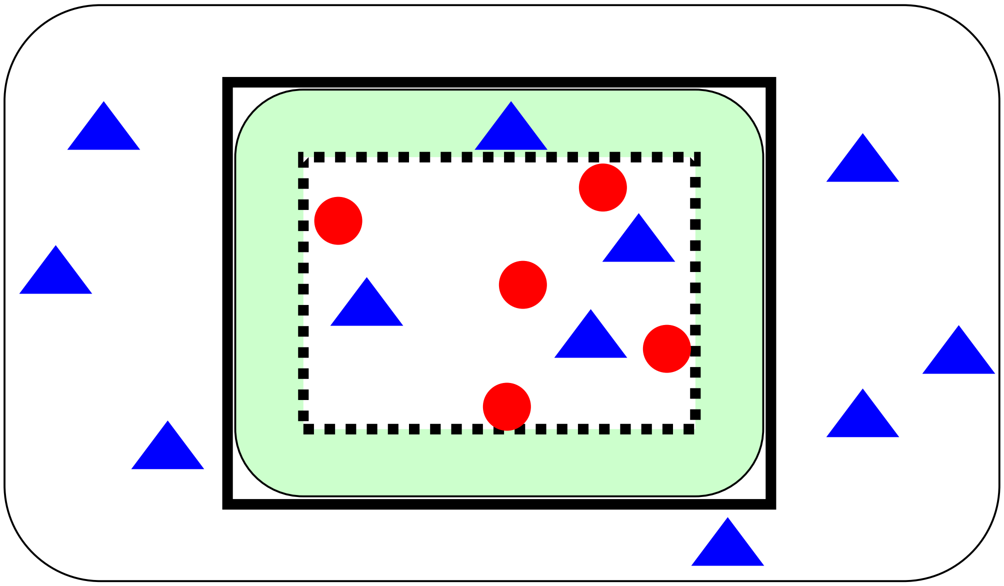

Figure 3. Red dots represent fish points. Blue triangles represent water points. All fish points lie within the inner rectangle. The margin between the inner and outer rectangles equals the specified upper bound on distance, so water points beyond the outer rectangle are extraneous.

simple approximation of the round-cornered one.

Here is code implementing this idea:

```
%let radius=10;

select max(x), max(y), max(z),
       min(x), min(y), min(z)
 into :maxx,  :maxy,  :maxz,
      :minx,  :miny,  :minz
 from fish;

create table watersubset as
 select * from water
  where &MINX-&RADIUS <= x <=
        &MAXX+&RADIUS
    and &MINY-&RADIUS <= y <=
        &MAXY+&RADIUS
    and &MINZ-&RADIUS <= z <=
        &MAXZ+&RADIUS;

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, watersubset as water
 where calculated
  distance_squared<&RADIUS**2
 group by FishID
 having distance_squared=
    min(distance_squared);
```

The first SELECT statement finds the extreme fish coordinates in each dimension, and stores them in macrovariables. It corresponds to drawing the inner rectangle in the diagram. The WATERSUBSET table is based on these values and on the radius (upper bound on distance). Its rows correspond to the water points lying inside the outer rectangle in the diagram. WATERSUBSET then replaces the full table of water points in the final statement.

Remember that the WHERE clause in the second statement (which creates WATERSUBSET) is logically redundant, so this modification does not introduce any change in results. All of the rows filtered out represent points too distant from any fish point to be considered, and would eventually be caught by the WHERE clause in the third statement. However, by computing the boundaries of the “box” and explicitly coding the WHERE clause using these boundaries, we allow SQL to shed these data points when it initially processes the WATER table, ahead of the join and thus reducing the size of the join. It never has to match them with fish points and calculate distances; hence I refer to the technique as “anticipatory subsetting”. Moreover, each extraneous water point is deleted once and for all, rather than being examined again and again, in the context of each fish point.

The degree to which this technique actually benefits performance depends on the spatial distribution of the data. After all, in Figure 3 there might few (or even no) blue triangles outside of the outer rectangle.

Before turning away from anticipatory subsetting, I want to explain that there is a reason for having WATERSUBSET materialize as a table, instead of making it a view or even rolling the two CREATE statements into one. Because it is a table, its size is known to SQL before the processing of the third statement begins, and this can be advantageous in a way which will be explained later.

Now look at Figure 4, which introduces the concept for a different optimization “hook”. The red dot again represents some fish data point. It is at the center of a circle whose radius is R, the maximum distance to be considered (we had specified R=10, but now we will be more abstract and use a variable). The grid of squares divides the two-dimensional space in intervals of R. In other words, if the

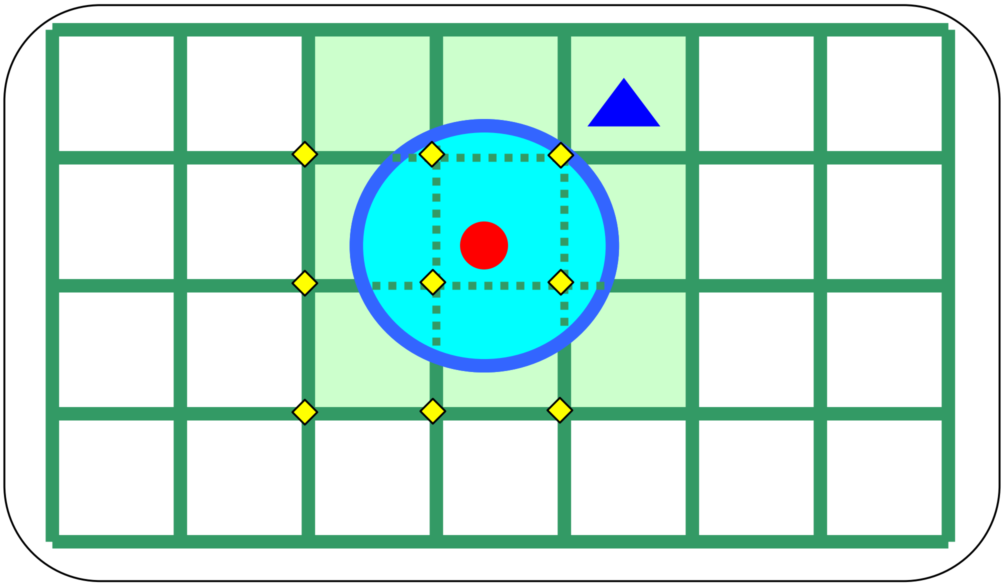

Figure 4. The red dot represents a fish point. The blue triangle represents a water point. Sides of the squares equal the radius of the circle. If the dot is in the square shown, match-eligible triangles must be within the 3x3 array of squares.

radius of the circle changes, the size of the squares changes as well.

Consider that the fish data point is in some particular square and the water data points which are close enough to be eligible for matching are in either the same square or an adjacent square (possibly diagonally adjacent). We can arbitrarily designate each square using the coordinates of its lower left corner; the lower left corners of the nine squares in question are marked with yellow diamonds.

If the coordinates of the red dot are X and Y, the coordinates of the lower left corner of the square in which it is located are R*int(X/R) and R*int(Y/R), where “int” is the SAS INT function, which returns integer parts (int(2.3) returns 2, int(-2.3) returns –3, etc.). This formula simply implements a so-called step function, which maps continuous values to discrete ones. So the lower left corner coordinates of the squares in which we must look for match-eligible WATER points are R*int(X/R) ± R and R*int(Y/R) ± R, where “±” is shorthand in the sense that a±b means a+b or a or a-b. Using subscripts, “f” for fish and “w” for water, we can express this as:

$$R*\mathrm{int}(X_w/R) = R*\mathrm{int}(X_f/R) \pm R$$
$$R*\mathrm{int}(Y_w/R) = R*\mathrm{int}(Y_f/R) \pm R$$

Dividing through by R:

$$\mathrm{int}(X_w/R) = \mathrm{int}(X_f/R) \pm 1$$
$$\mathrm{int}(Y_w/R) = \mathrm{int}(Y_f/R) \pm 1$$

To be able to code this in SQL, we first have to find a way to implement the “±” notation. This can be done by introducing a third table (OFFSETS) into the join, one which simply enumerates the possible offsets; geometrically, these correspond to the possible forms of adjacency of the tinted squares. Here is that table:

xoffset	**yoffset**
-1	-1
-1	0
-1	1
0	-1
0	0
0	1
1	-1
1	0
1	1

Note that this table of offsets is for the two-dimensional example in the diagram. In three dimensions there is a third column (ZOFFSET) and a total of 27 rows.

The code incorporating this logic is:

```
%let radius=10;

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, 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);
```

You can see that we’ve added three conditions to the WHERE clause. Together they are redundant in that they follow, as we have shown, from the inequality. This is equivalent to observing that, in Figure 4, any point within the circle is within the shaded large square. In other words, expanding the WHERE clause in this fashion does not change the results which are returned.

Having a third table in the join does not change the internal process much. SQL will bind OFFSETS to one or the other of the larger tables, then join that intermediate result to the remaining table. See Kent.

The inclusion of OFFSETS in the join increases the theoretical size of the join by a factor of 27. However, it also makes possible dramatically improved performance. This is due to the internals of SQL processing, and in particular to the selection of methods for implementing a join. Kent provides much detail; I will just offer a general explanation.

The improved performance is due to the fact that the added conditions (in the WHERE clause) are equalities of the type which make this a so-called “equi-join”. In processing an equi-join, the SQL processor can identify and locate the rows in one table which match a particular row (or set of rows) in the other table, without having to exhaustively examine all row pairs and discard those which do not match. If this in fact allows SQL to bypass a lot of non-matching and thus irrelevant fish-water pairs, it can save time and resources.

I refer to the modification just described as the “redundant equi-join condition”. Like anticipatory subsetting, it will prove useful in crafting a practical solution to the given problem. The two techniques can be combined. Here is the code:

```
%let radius=10;

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;
```

```
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);
```

This code is in fact the “building block” we will use to solve the original problem, so it is worthwhile at this point to have some test results. I generated some simple test data: 10,000 fish points and 10,000 water points. I set the radius to 25, which is a small value relative to the span of the data points. I first ran the naive version of the bounded query; it took 3:50 (3 minutes and 50 seconds). When the anticipatory subsetting technique was applied, this dropped to 2:50 (which result was almost completely predetermined by the extent to which the two datasets were spatially separated). Applying the redundant equi-join condition without anticipatory subsetting resulted in use of just 33 seconds. Finally, applying both techniques together (that is, using our building block), the time was a bit less, about 25 seconds.

BACK TO THE ORIGINAL PROBLEM

Let’s summarize the discussion up to this point. We began with a problem which has a theoretically sound SQL solution which does not scale at all well relative to the needs. We turned to a variant of that problem, one which changes the “rules” and thus produces different results, but which opens the door to techniques which improve performance without further altering results.

The difference between the initial problem and the variant is that the latter returns only a subset of minimum-distance fish-water pairs, those for which the distance is less than the specified upper bound. So it should be possible to solve the initial (unbounded) problem iteratively. The process would be along these lines:

1. Place the fish data points to be matched into a table; call it UNKNOWNS.
2. Set an initial upper bound.

3. Solve for the nearest-neighbor pairings with distances within that bound, using the building-block code.
4. Delete from UNKNOWNS the fish points which have been matched as a result
5. Increase the upper bound
6. Repeat steps 3 through 5 as many times as needed.

I refer to this strategy as "concentric partitioning", because it does break the problem into pieces and the iterations do involve successively larger spheres (circles in our diagrams) centered around given points. Concentric partitioning is distinct from and different from the parallel partitioning which can be used to subdivide the problem "up front". In particular, concentric partitioning is inherently sequential, with the result of each iteration determining the input to the next.

Notice that unlike parallel partitioning, which basically just rearranges the workload by subdividing the fish dataset, concentric partitioning may be able to effectively shrink it. That would happen if, at each radius, we can efficiently match a relatively large number of rows from UNKNOWN. Then the size of UNKNOWN passed to the next iteration would be reduced, making it less costly.

The degree to which this actually works will certainly depend on the given data, but it will also depend to a great extent on the progression of radii. Let's try to see heuristically how, for a particular iteration, the choice of the radius value affects the efficiency of the process.

Look at Figure 5. Remember that the size of the squares adjusts to match the radius of the circles, that a group of 9 squares in a 3x3 array represents the boundary implemented by the equi-join, the circle represents the boundary implemented by the computationally more expensive inequality condition, and that the summary process (invoked by the GROUP BY and HAVING clauses) picks a single point when the are multiple points within a circle. The three red dots represent three fish points which have not yet been matched to water points.

The leftmost fish point (red dot) depicts the ideal situation. There is a single water point (blue triangle) within the circle, so the solution for this particular fish point can be found with a minimum of computation. Then this fish point will be excluded from processing when the radius is increased. Thus, its distances from the quite possibly large number of more distant water points never get computed; contrast this with the naive query, where every fish-water pair must be examined.

The middle fish point represents a near-worst case, because it is not matched to a water point at this radius and must be kept for processing at the next radius.

The rightmost fish point represents an intermediate situation. There are multiple potential matches provided by the equi-join (that is, within the 3x3 subgrid), so the inequality and the HAVING clause have to screen out all but one, but then this fish point will have been matched and it drops out of the set of fish points carried forward. To the extent that a large number of fish points have large numbers of "candidate" water points provided by the equi-join, some of the massiveness of the naive solution remains.

Now consider what would happen if we chose a smaller radius. It should reduce the number of and size of the multiple candidate situations, thus reducing the volume of computation. That's good. It would also likely increase the number of fish points which are not matched and must be processed again at the next radius. That's bad.

With a larger radius, we get opposite effects. There are more and larger multicandidate cases, which increases the

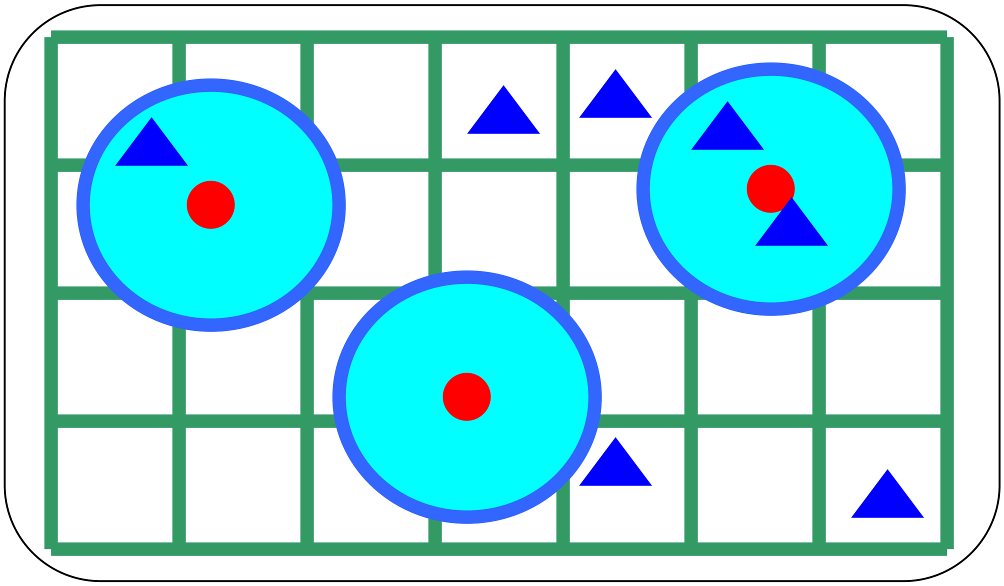

Figure 5. Red dots represent fish points. Blue triangles represent water points. The sides of the squares equal the radii of the circles. Smaller radii lead to fewer, but more efficient, matches. Larger radii lead to more, but less efficient, matches.

computing burden. On the other hand, more fish points should be matched. At an extreme, the radius would be so large that the redundant equi-join condition would admit all water points, and the query would essentially behave like the naive one.

So what is needed, in choosing each successive radius, is something of a happy medium. I am not prepared to offer any systematic or analytic approach to deriving such a sequence. Nevertheless, concentric partitioning is the last piece of the puzzle. We can now envision an overall strategy for solving the original problem. In brief, this strategy entails use of parallel partitioning to break the problem into pieces of more manageable size, then processing each of the resulting segments by use of concentric partitioning to apply the building-block code with a sequence of successively larger radii.

TUNING AND TESTING

It's time to see whether the strategy we have devised can be used to mitigate the problems of scale to a degree which will let us solve the given problem at full scale in an acceptable amount of time.

First, it's important to explore issues pertaining to the way in which SQL processes equi-joins. Kent covers the ground in some detail, so I will only try to summarize.

In recent releases of SAS software, SQL actually has three methods available to evaluate equi-joins. One involves sorting (if necessary) and coordinated sequential processing resembling that done by the MERGE statement in the SAS DATA step. The second relies on indexes, if any exist. The third, known as hashing, is memory-based. Hashing tends to provide the best performance, but its applicability is limited by the amount of memory available. It is difficult to generalize with complete certainty about the relative performance of the various methods, because data characteristics are always a factor. However, any of the three optimized methods will usually outperform a brute force evaluation. For this problem, indexing is not a good candidate (for reasons explained by Kent), and we can expect hashing to outperform sequential matching. So we want to facilitate the use of hashing. Basically, this means reducing the size of the joins to be performed.

Kent also describes an undocumented option, BUFFERSIZE=, which can be exploited to make more memory available for hashing. I found that a value of 2048000 was appropriate in the situations I tested. This however should absolutely not be taken as a rule of thumb.

Testing requires data. Not having access to the original research, I synthesized what I considered to be more or less plausible and realistic tables of fish and water data points. Figures 6 through 8 present some two-dimensional views of these datasets. For both datasets, the Z coordinates are uniformly distributed along the interval 100 to 110, so there is not much to see from that perspective.

Figure 6 is a sample of 3,000 of the fish data points and Figure 7 is a sample of 3,000 of the water data points. Figure 8 provides an overlay of the boundaries; the areas shaded in blue indicate the range of X and Y coordinates for the water data and the red dotted-line encompasses the corresponding range for the fish data.

All of the X and Y coordinates were generated with a normal distribution, but the means and variances differ. I did that, and also designed the boundaries, to create some asymmetries and discontinuities, which in turn should yield some outliers.

Now we can turn to the implementation of the proposed strategy for solving the problem. The coding task entailed development of some relatively straightforward macros (available as part of a code package distributed along with this paper, or on request from the author). The real issues were: (1) deciding how many parallel partitions to form, and on what basis to do the partitioning; and (2) specifying the progression of radii for each resulting segment.

Let's also review the immediate objectives. They are: (1) to keep the individual joins small enough, relative to the amount of memory available, that the hash method will be used; (2) to keep the intermediate result sets yielded by the joins small enough to fit within the disk space available for the WORK library; (3) to avoid an excessive amount of overhead processing; and (4) to keep the aggregate number of pairings formed small (in other words, to avoid the essence of the difficulty found in the naive solution). Objectives (1) and (2) are essentially yes/no propositions which can be viewed as hurdles. Objective (3) is unlikely to be a problem except in extreme situations. That points to (4) as the final determinant of performance.

The first decision involved the parallel partitioning. I eventually chose to subdivide the problem into 50 segments of roughly equal size. As explained earlier, this is done by subsetting the fish table. I did this on the basis of quantiles of the X coordinate in order to make the subsets compact in terms of their spatial data. This serves to leverage the effect of the anticipatory subsetting technique. To see this, look at Figure 3 and consider that making the inner rectangle narrower, perhaps much narrover, can only increase the number of blue triangles lying outside the outer rectangle.

I found that coding the SQL so that WATERSUBSET is created as a table rather than as a view was necessary for anticipatory subsetting to work as intended. The sequence of methods to be used in processing a statement (including any views referenced within the statement) is determined before processing begins. The choice of join methods (that is, whether hashing is feasible) depends on the sizes

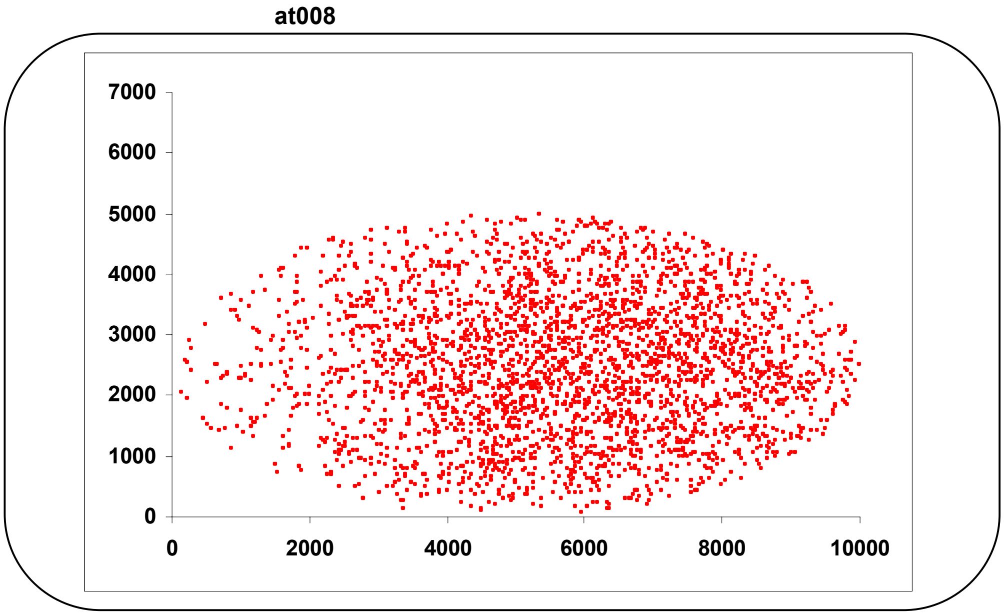

Figure 6. Red dots represent X-Y coordinates of 3,000 randomly chosen fish points.

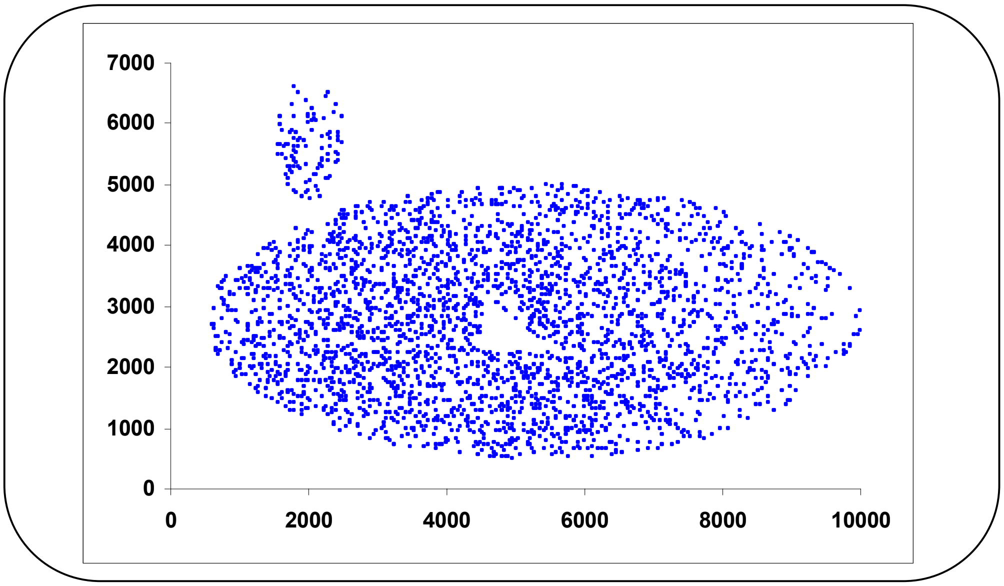

Figure 7. Blue dots represent X-Y coordinates of 3,000 randomly chosen water points.

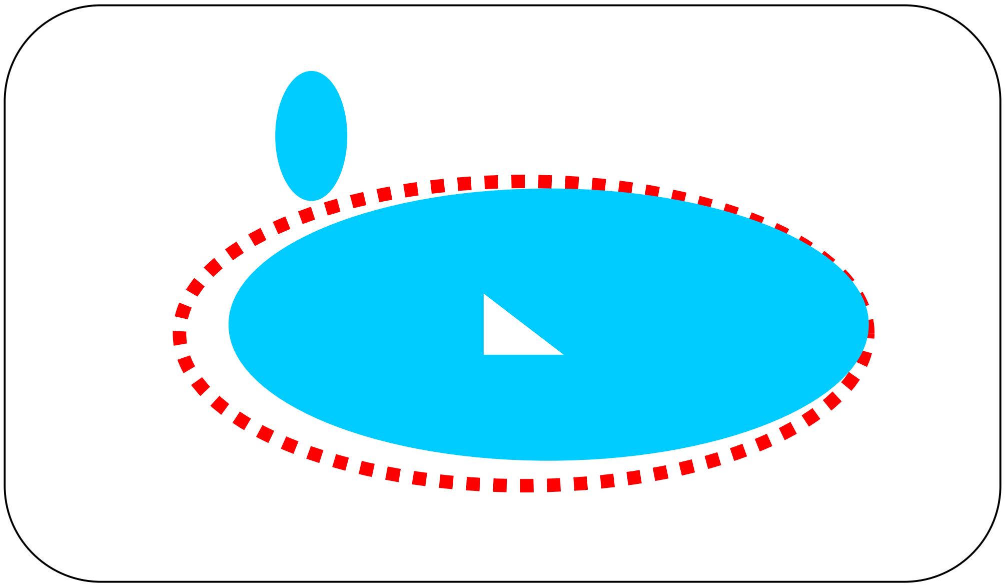

Figure 8. In this overlay based on Figures 6 and 7, blue-tinted areas are the X-Y range of water data and the red broken line encloses the X-Y range of fish data.

of the row sets being joined. So if WATERSUBSET is a view, SQL has to guess how big it will be in deciding on a join method. I found that it tended to guess conservatively; in some cases this leads SQL to not use the hash method. So creating a table lets SQL consider the actual size of the subset, which can increase the chances that hashing will be used.

Selecting values for the radii was probably the most difficult aspect of the whole problem. As we concluded earlier, in terms of objective (4), they should not be too small and should not be too large. That's of little operational help. So I began by concentrating on objective (2), the disk footprint issue. Another way of saying this is that I was seeking an acceptable solution, not an optimal one.

In brief, here is the procedure I followed. I took a sample of 2,500 fish points and 500 water points, performed a simple join to form all 1.25 million pairings, and calculated the distance for each. Then I assigned each of these distances, on the basis of its X coordinate from the fish dataset and identified by its ID number from the fish dataset, to one of the 50 parallel segments. So, on average, I had 25,000 distances for each segment.

By observing the growth of temporary files on my hard disk during some earlier tests, I estimated that about 150 bytes of disk space were used for each row passed to the back end of the query. I had about 10 gigabytes of space available. So that meant that about 65 to 70 million rows could be accommodated in each query. Since my sample contained 1.25 million of the roughly 20 trillion distances in the full-scale matrix, each row of the sample represented 16 million distances. Thus I wanted to step through my sample four rows at a time to find radius values which would not cause the hard disk to fill up during processing.

So for each of the 50 segments, I considered the sample distances in increasing order. Then I removed the four smallest, and captured the value of the largest among these. Then I removed from the residual all distances involving the fish ID values from the set of four. This simply reflects the way in which the concentric partitioning technique serves to shrink the problem. I repeated the process until the sample distance matrix was exhausted. The result was a plan providing a separate sequence of increasing radii for each of the 50 segments. Typically, the sequence for a given segment consisted of 15 to 20 values.

In practice, I found some deficiencies with this approach. I had not actually sampled the distance matrix. Rather, I had sampled the fish and water datasets and constructed a distance matrix from those two samples. A consequence is that the relatively short distances were underrepresented. Outliers at the other extreme also tended to be missed.

If I had proceeded, the initial radii for at least some of the segments would have been too large, and, at the other end, many fish points would have been left unmatched. So I arbitrarily adjusted my plan in two ways: I subdivided the initial radius into five (one ninth the originally derived value, two ninths, one third, two thirds, and the originally derived value itself). I also added a number of radii at the high end, repeating the interval between the highest originally derived value and the second highest. I also included code to save any unmatched fish points in a table of "hard cases" for later analysis.

This set of seat-of-the-pants decisions proved adequate: I was successful in computing matches for all 650 thousand fish data points. It took a total of 62 hours. This time was not evenly distributed however. Even though the 50 segments were nearly equal in size (with about 13,000 fish points each), the computing time for individual segments varied from a low of 13 minutes to a high of eight hours.

The building-block code was run a total of 926 times. The aggregate size of these joins was about 39 trillion fish-water pairings. While this superficially appears to exceed the size of the utterly infeasible brute-force join, keep in mind that it is enlarged 27-fold by the presence of the table of offsets. In fact, the figure would have been many times greater if not for the reduction effect of the anticipatory subsetting technique. The aggregate size gets as large as it does because of the iterative processing entailed by the concentric partitioning. It is really only a curiosity, however, because the equi-join methods effectively bypass almost all of these pairings.

It's more enlightening to look at the number of rows passed to the back end of the query. I wasn't able to capture that information exactly, but based on my capacity analysis I do not think it exceeded 70 billion. That's a big number, but not in comparison to the 20 trillion which would have been involved without use of the techniques we have discussed. This demonstrates the benefit of the concentric partitioning.

Incidentally, I believe that the anticipatory subsetting provided at best marginal benefits in this particular exercise. That's because the fish data segments were so small (about 13,000 rows each), that the size of the water tables was non-critical to the choice of join method. However, this might not be the case with moderately different problem parameters, and the overhead involved is relatively modest.

This obviously was very much an *ad hoc* process. There is room for possibly more sophisticated and analytically grounded methods in selecting values for all of the parameters.

SUMMARY AND CONCLUSIONS

We began with a simple problem and a simple solution, one which unfortunately was impractical due to severe problems of scale. By elaborating on the code and applying a series of techniques, some simple and others more intricate, we devised a practical solution using SQL.

In the process of doing the experimentation underlying this paper, I concluded that the given problem at its given scale pushed the performance envelope for a simple office computer. Of course that envelope shifts with time. The SAS-L posting which initiated my thinking appeared early in 2001. Using the machine on my desk at that time, I probably could not have completed the task. On the other hand it will probably be less daunting in 2006, when I have perhaps four gigabytes of memory, a 6GHz processor, multithreading, and a half-terabyte disk.

References

Baxter, Shelley, "Fuzzy conditional merge on 3 variables", comp.soft-sys.sas, 22 January 2001

Gionis, Aristides, "Computational Geometry: Nearest Neighbor Problem" (http://theory.stanford.edu/~rajeev/CS361/lecture12-scribe.pdf), Stanford University

Kent, Paul, "SQL Joins -- The Long and The Short of It" (http://support.sas.com/techsup/technote/ts553.html), TS-553, SAS Institute Inc.

SAS Institute Inc., *SAS SQL Procedure User's Guide*, Version 8

Skiena, Steven, "Nearest Neighbor Search" (http://www.cs.sunysb.edu/~algorith/files/nearest-neighbor.shtml), State University of New York, Stony Brook

Contact Information

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

Howard Schreier
U.S. Department of Commerce
Stop H-2815
Washington DC 20230

(202) 482-4180

Howard.Schreier@mail.doc.gov