Picking Up Where the SQL Optimizer Leaves Off

Howard Schreier
U.S. Dept. of Commerce

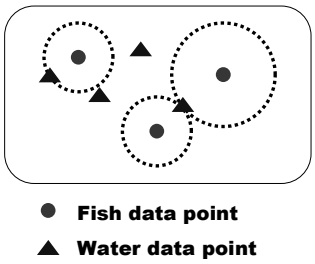
Purpose

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.

Case study, matching spatial data

- First table: locations of fish in a body of water
- Second table: points where water conditions measured
- Task: find closest water-condition point for each fish-location point

Diagram (two dimensions)



SQL code (three dimensions)

```
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);
```


Scale is the issue

- 650 thousand fish points
- 30 million water points
- Almost 20 **trillion** rows in intermediate result
- Requirements
  - 50 years
  - disk footprint for WORK library: 3 petabytes

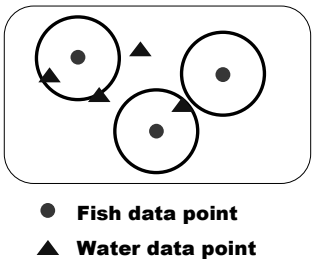
Preliminary bit of optimization

```
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);
```


Parallel partitioning

- Divide fish table into pieces; solve problem for each separately; combine results
- Makes task no smaller, but more manageable

Variant: Upper bound on distance



Variant: Upper bound on distance

```
%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
```


Variant: Upper bound on distance

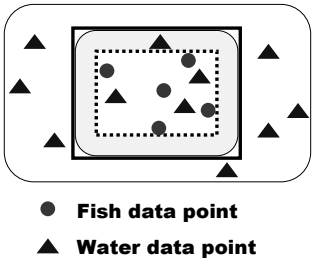
```
where calculated
  distance_squared<&RADIUS**2
 group by FishID
 having distance_squared=
    min(distance_squared);
```


Consequences

- Size of join unchanged
- Smaller intermediate result passed to “back end”
- Wrong answer (but subset of correct answer)

Useful nevertheless

Anticipatory subsetting



Anticipatory subsetting

```
%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;
```


Anticipatory subsetting

```
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;
```


Anticipatory subsetting

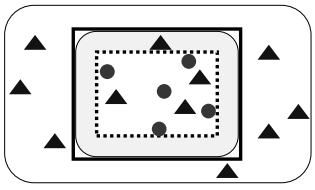
```
create table nearest as
 select FishID, WaterID,
        (fish.x-water.x)**2+
        (fish.y-water.y)**2+
        (fish.z-water.z)**2
         as distance_squared
 from fish, watersubset as water
```


Anticipatory subsetting

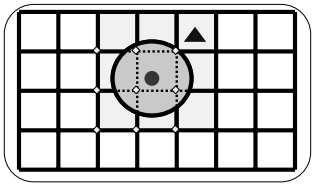
```
 where calculated
  distance_squared<&RADIUS**2
 group by FishID
 having distance_squared=
    min(distance_squared);
```


Consequences

- No change in results, since restriction is redundant
- Reduction in size of join (maybe)



Redundant equi-join condition



● Fish data point

▲ Water data point

Redundant equi-join condition

- Coordinates of LL corner of small square with fish point:
  R*int(X/R), R*int(Y/R)
- Coordinates of LL corners of 9 small squares:
  R*int(X/R) ± R, R*int(Y/R) ± R
- So
  R*int(X_w/R) = R*int(X_f/R) ± R
  R*int(Y_w/R) = R*int(Y_f/R) ± R
- Dividing through by R:
  int(X_w/R) = int(X_f/R) ± 1
  int(Y_w/R) = int(Y_f/R) ± 1

OFFSETS table implements ± operator

```
xoffset  yoffset
     -1       -1
     -1        0
     -1        1
      0       -1
      0        0
      0        1
      1       -1
      1        0
      1        1
```


Redundant equi-join condition

```
%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
```


Redundant equi-join condition

```
 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);
```


Consequences

- No change in results, since restriction is redundant (as the circle is within the large square)
- Increases theoretical size of join by factor of 27
- Makes it possible to avoid complete brute-force evaluation

The Genuine BRANNOCK Device

"Designed in 1927, The Brannock Device® foot-measuring device is a must in all retail footwear stores."

http://www.brannock.com/

25

Building block code

- Based on variant with upper bound
- Incorporates
  - anticipatory subsetting
  - redundant equi-join condition

26

Building block 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;
```

27

Building block code

```
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;
```

28

Building block code

```
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
```

29

Building block code

```
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);
```

30

Performance

- Generated test data tables with 10,000 rows in each (fish, water)
- 3:50 using brute force
- 2:50 with anticipatory subsetting
- 0:33 with redundant equi-join condition (only)
- 0:25 with both techniques

Review

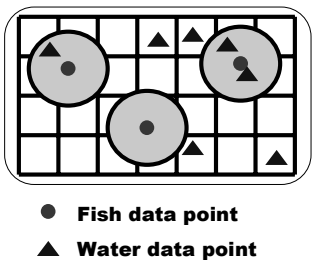
- Original problem has simple SQL solution
- That solution does not scale well
- Variant problem
  - Returns only a subset of the correct result
  - Does permit performance tuning

Concentric partitioning

1. Set value for upper bound (&RADIUS)
2. Solve using building block code; result is subset of the correct result
3. Remove fish points successfully matched
4. Increase upper bound (&RADIUS)
5. Repeat (2) through (4) as many times as necessary

The devil is in the details.

Concentric partitioning: choosing radii



Concentric partitioning: choosing radii

- Ideal: one-to-one
- Too small: no match-ups
- Too large: inadequate performance gain over the naive code
- Medium

Strategy overview

- Apply parallel partitioning to divide problem into pieces of manageable size
- Use concentric partitioning to solve each piece
- Each concentric partitioning iteration uses the building block SQL code (which incorporates anticipatory subsetting and redundant equi-join condition)

Equi-join methods in PROC SQL

- Coordinated sequential processing
- Index-based processing
- Hashing
  - Preferred method for this problem
  - Limited by memory availability

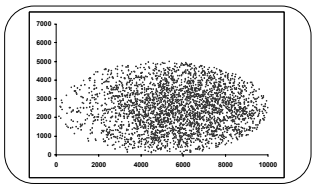
Providing memory for hashing

- Partition data table(s) to reduce size of joins, and thus amount of memory needed
- Use (undocumented) option BUFFERSIZE= to increase available memory

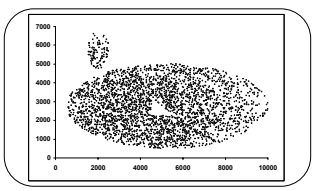
Data for testing

- Synthetic
- Full scale
  - 650 thousand fish points
  - 30 million water points
- Realistic (asymmetries, discontinuities, outliers)

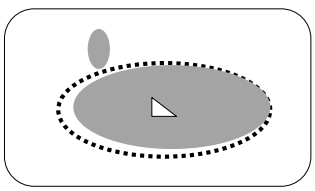
Sample fish points (X-Y projection)



Sample water points (X-Y projection)



Overlay



SAS coding

- Based on building block
- Other code (mostly SQL) for generating data, sampling, partitioning, capturing performance metrics, other housekeeping
- Macro facility, especially for looping
- Validated against results of brute-force solution

Decisions

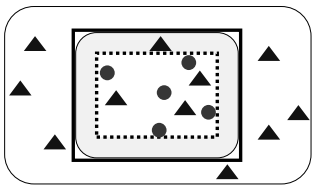
- Number and shape of parallel partitions
- Progression of radii

Objectives

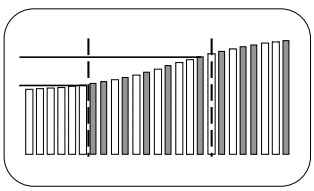
- Small joins (to enable hashing)
- Small result sets from joins (to manage disk footprint)
- Avoid excessive overhead
- Limit aggregate number of pairings considered (<< 20 trillion)

Parallel partitions

- 50
- Based on quantiles of X coordinate



Concentric partitions: radii



Results

- All 650,000 fish data points matched
- Building block code run 926 times
- Aggregate processing time was approximately 62 hours
- Times for the 50 parallel partitions (each with approximately 13,000 fish data points)
  - Minimum: 13 minutes
  - Maximum: 8 hours
- Aggregate size of joins: approximately 39 trillion
- Estimated 70 billion rows passed to “back end”

About the speaker

Howard Schreier

U.S. Dept. of Commerce
Washington DC 20230

(202) 482-4180

Howard_Schreier@ita.doc.gov