

When PROC SQL Is Overwhelmed

How to Use the SyncJoin Algorithm to Match Millions of Records Fast

Houliang Li, Frederick, MD

ABSTRACT

To join two huge data sets containing millions of observations each, either in SAS[®] software or an RDBMS product such as Oracle, we usually rely on the built-in query optimizations provided by the SAS SQL procedure or the RDBMS product. In most cases, the built-in optimizations successfully avoid a full Cartesian product join of the tables and accomplish the query efficiently. However, there are multiple conditions where the built-in optimizations become powerless. A cross-table between condition is one of them. This presentation describes the Synchronized Join algorithm, or SyncJoin, which uses the SAS DATA step to implement a join equivalent to the SQL statement containing a cross-table between condition. The SyncJoin algorithm not only finishes the query in a tiny fraction of the time required by both PROC SQL and Oracle, but also consumes a minimal amount of system resources. By constantly synchronizing the two separate passes through the two data sets, the SyncJoin method reduces both processing efforts and runtime to the absolute minimum.

INTRODUCTION

Conceptually, any standard SQL implementation, be it SAS PROC SQL or an RDBMS product such as Oracle, starts with the creation of a temporary Cartesian product of the tables involved. Only a Cartesian product join, which has all the possible combinations of rows from the tables, can satisfy all possible sets of conditions. Subsequent filtering of the combined rows from the Cartesian product, via a where clause, produces the desired results. Needless to say, a single-table query does not create a Cartesian product.

(Please note that throughout the presentation, the following terms are used interchangeably: table and data set, record, row, and observation, field and variable, and key and by variable.)

A Cartesian product join's omnipotence comes with tremendous costs in terms of computing resources. Suppose we need to join two tables, Table1 and Table2. Each table has one million rows, and each row is exactly 512 bytes long. If there are no duplicate variables between Table1 and Table2, then a Cartesian product join of the two tables will create one trillion rows, and each row will be 1024 bytes long, or 1KB. To store this temporary monster requires almost one petabytes (or one million gigabytes) in disk space! Handling such a volume inevitably exhausts other computing resources such as CPU, memory, and I/O.

Fortunately, all standard SQL implementations have built-in query-optimizing mechanisms to avoid such a disastrous outcome. Kent (1997) and Lavery (2005) discussed various optimizations available to SAS PROC SQL. You can also find many articles on query optimizations adopted by Oracle and other popular RDBMS products. When certain conditions make it impossible to optimize a query, an SQL implementation does not really create the full Cartesian product all at once. Instead, it works through the tables portion by portion within the confines of the computing environment, in effect creating a mosaic-like Cartesian product. Only one small piece of the mosaic is "visible", i.e., available for processing, at any given moment during the join operations. Once the piece is fully processed, it is immediately discarded to free up resources for the next piece. The mosaic-like approach may be painfully slow and tedious, but at least it gets the job done.

A where clause that uses the operator "between" to link one variable from one table to at least one variable from another table creates a condition that no query-optimizing engine presently can handle. Both SAS PROC SQL and Oracle surrender to this condition and resort to a full Cartesian product join. Since between conditions like this are quite common in SQL queries, a better solution is highly desirable, especially where large data sets are involved. It is not unusual to see a seemingly simple query containing such a cross-table between condition to run for hours or days in a production environment, consuming all available resources along the way. When that happens, the whole computing system slows to a crawl and other processes are stalled due to the lack of resources.

The following sample code generates the dreaded full Cartesian product join note in the SAS log:

<pre>data table1; var1 = 2; run;</pre>	<pre>data table2; var2 = 1; var3 = var2 + 3; run;</pre>	<pre>proc sql noprint; create table may_run_forever as select * from table1 a, table2 b where a.var1 between b.var2 and b.var3; quit;</pre>
<p>NOTE: The execution of this query involves performing one or more Cartesian product joins that can not be optimized.</p>		

Figure 1

Clearly, the sizes of the data sets are irrelevant as far as triggering a Cartesian product join is concerned. Each table in Figure 1 has only one observation, so the demand on computing resources is minimal. Balloon the observation counts to one million each, and you have an instant nightmare at hand in any computing environment. If we further extend the record length of the observations to hundreds of bytes or more, as is typical of real life data, do not be surprised if you have to wait a couple of days before this innocent, simple query returns from a long journey through the mosaic Cartesian product join. One trillion combined observations always take a lot of computing resources to process, no matter how big or small each mosaic piece is.

BENCHMARK TEST

For the purpose of this presentation, we generate two random data sets: orders and quotes for a fictional stock market that operates seven days a week between 8:00AM and 6:00PM. To simplify matters, we limit the orders and quotes to only one fictional stock – ABCD. For each second during market hours, there are 0 to 10 orders and quotes. Each order is valid only for the second at which it is entered, but each quote can have a randomly assigned life span of between 1 and 5 seconds. An order is either a buy order or a sell order in the stock ABCD, just like a regular customer order. However, a quote has both a buy side and a sell side, reflecting the fact that a market maker (broker/dealer) is obligated to make a market in the stock. The buy price is always lower than the sell price in the same quote, thus allowing the market maker to profit from the price difference. Figure 2 lists all the variables and their formats. Please note that a filler variable in each data set serves to adjust the record length of the observations. You can simply add or remove characters to / from the filler variables to customize the record lengths. For easy calculation of data set file sizes, the orders and quotes both have a record length of exactly 512 bytes.

Orders		Quotes	
Variable	Format	Variable	Format
order_id	8.	quote_id	8.
stock_name	\$8.	stock_name	\$8.
order_time	datetime20.	quote_time	datetime20.
		quote_end_time	datetime20.
customer_id	\$8.	broker_id	\$8.
order_side	\$8.	buy_price	8.2
order_price	8.2	buy_quantity	8.
order_quantity	8.	sell_price	8.2
		sell_quantity	8.
filler_order	\$456.	filler_quote	\$440.

Figure 2

We want to link every order to every quote that is in effect at the order entry time. The matched pair must also have the same price and quantity on the correct buy / sell side. For example, if a buy order is entered into the market at 10:20:30AM for 100 shares at \$10.50 per share, and a quote is entered into the market at 10:20:28AM, has a life span of 4 seconds, and matches the buy order on the sell side, i.e., selling 100 shares at \$10.50, we want to see the order and quote linked to each other. In other words, we are looking for all potential stock transactions between customers and market makers. Because there might be multiple orders that can be matched to the same quote, and multiple quotes that can be matched to the same order, a DATA step match merge does not work in this many-to-many relationship. We have to use SAS PROC SQL or an RDBMS product like Oracle.

In order to see the impact of data volume on the total runtime, the SAS macro in **Appendix** uses a few parameters to control the number of observations generated for both orders and quotes. The actual tested data sets range from 100,000 to 1000,000 observations each. Multiply the numbers by 512 bytes, and we get a rough estimate of the file sizes in bytes. More observations are certainly possible, but the runtime becomes unacceptably long. If you think you have one of the best computing systems around, try ten million observations each and see what happens.

Once the orders and quotes are in place, coming up with an SQL statement is a breeze. Figure 3 displays both the code and the two log notes that SAS PROC SQL generates about this query.

```
proc sql noprint _method;
  create table matches_sql as
  select *
  from orders a,
       quotes (rename=(stock_name=quote_stock_name)) b
  where a.order_time between b.quote_time and b.quote_end_time
        and (a.order_side = "BUY" and
              b.sell_quantity = a.order_quantity and
              b.sell_price = a.order_price
             or
              a.order_side = "SELL" and
              b.buy_quantity = a.order_quantity and
              b.buy_price = a.order_price);
quit;
```

NOTE: The execution of this query involves performing one or more Cartesian product joins that can not be optimized.

NOTE: SQL execution methods chosen are:

```
sqxcrt
  sqxjst
    sqxsrc( WORK.QUOTES(alias = B) )
    sqxsrc( WORK.ORDERS(alias = A) )
```

Figure 3

Executing this relatively simple SQL statement, however, proves increasingly difficult as the number of observations in each data set grows. The table below records the different times required by PROC SQL to execute the same query on the growing data sets. The numbers under orders and quotes indicate the counts of observations for each randomly generated data set. They are about the same for each test.

Orders	Quotes	Time Category	Time
98,781	99,465	Real Time	00:27:00.94
		User CPU Time	00:25:16.08
		System CPU Time	00:01:06.85
		Total CPU Time	00:26:22.93
197,667	197,799	Real Time	01:44:30.85
		User CPU Time	01:39:45.53
		System CPU Time	00:04:07.85
		Total CPU Time	01:43:53.38
296,489	297,046	Real Time	04:02:49.62
		User CPU Time	03:46:27.28
		System CPU Time	00:10:35.98
		Total CPU Time	03:56:63.26

395,122	395,512	Real Time	06:58:22.52
		User CPU Time	06:39:02.28
		System CPU Time	00:17:12.93
		Total CPU Time	06:56:15.21
496,585	496,119	Real Time	11:08:51.66
		User CPU Time	10:29:38.33
		System CPU Time	00:31:06.48
		Total CPU Time	11:00:44.81
990,931	989,682	Real Time	43:38:07.66
		User CPU Time	41:39:39.75
		System CPU Time	01:47:38.08
		Total CPU Time	43:27:17.83

Figure 4

If we compare the times in bold, i.e., the total CPU time needed to join the orders and quotes as their sizes grow, we can see an interesting correlation. When both orders and quotes double in size, from roughly 100,000 to 200,000, the total CPU time quadruples, from about 26 minutes to 104 minutes. Again, when the number of observations doubles from roughly 200,000 to 400,000, the total CPU time quadruples from 104 minutes to 416 minutes. We see the same correlation when the number of observations increases from half a million to one million.

This correlation does not happen by accident. We have purposely generated roughly equal numbers of orders and quotes each time. If N represents the number of observations in each data set, then a Cartesian product will create $(N * N)$ combined rows. When we increase the number of observations to $2N$, the new Cartesian product will now create $(2N * 2N)$ combined rows, or $4 * (N * N)$. This explains the quadrupling of the total CPU time, as there are now four times as many rows to process as before. We can safely assume that the processing time for each row is about the same for large data sets. The quadrupling relationship also holds true when orders and quotes differ significantly in their numbers of observations, as long as they each double their sizes at the same time. Keep doubling the observation counts a few more times, and you are never going to see any results from the query, no matter how cutting edge your computing environment may be. Consider yourself warned about testing ten million observations. Waiting for 44 hours on a single query of million-record data sets is bad enough. Something has to be done!

ANALYSIS

No query-optimizing engine in any current SQL implementation can handle the cross-table between condition. The query and data analyses, routinely conducted by the engine prior to deciding on an optimal execution plan, do not recognize the single most important feature inherent in the data. Yet we know that an order can only be matched to a few quotes within a small time range around the order entry time. It does not make any sense to try to find quotes that fall either way before or after the order entry time, not to mention scanning all the quotes for potential matches.

Let's take a closer look at this data feature. The SAS macro in **APPENDIX** dictates that on one extreme there are 10 quotes for each second and each quote lasts for 5 seconds. That translates into 50 quotes to be matched to an order. On the other extreme is no effective quote at all when an order is entered, therefore no match to make. On average, there are 2.75 quotes for each second and each quote lasts for 3 seconds. The most efficient, or optimal, matching algorithm needs to make about 8 matches ($2.75 * 3 = 8.25$) for every order and no more. This number may change if we modify the parameters used in generating the two data sets, including quote duration and the number of quotes per second, but it will always be a fixed and relatively small number.

However, without the benefit of this inside knowledge about the data, the built-in, general-purpose, query-optimizing engine has to match each order to all quotes when it sees the cross-table between condition. For 100,000 quotes, that translates into a ratio of about 12,000 to 1 in attempted matches per order (100,000 by PROC SQL versus 8.25 by the optimal algorithm). For one million quotes, the ratio of attempted matches increases by tenfold. This is a linear relationship that spells trouble for those users who have to rely on the built-in query optimizations when joining large data sets with cross-table between conditions. Interestingly, the order counts do not affect the ratio of required / attempted matches. As long as there are roughly equal numbers of orders and quotes, we expect to see a linear relationship between the ratio of required CPU times (PROC SQL versus the optimal algorithm) and the number of quotes. The CPU time can be considered as a proxy measure of attempted matches performed by either approach.

Mathematically, if there are M orders and N quotes, then the number of attempted matches performed by a Cartesian product is $(M * N)$, whereas the optimal matching algorithm requires only $(M * K)$ matches, where K represents the

optimal average matches per order, in this case, $K = 8.25$. The ratio of attempted matches, and indirectly the ratio of required CPU times, is $(M * N) / (M * K) = (N / K)$, which is clearly a linear relationship contingent on the number of quotes since K is already fixed. In other words, the more quotes there are, the bigger the performance difference is between the truly optimal matching algorithm and the Cartesian product join chosen by SAS PROC SQL.

But how do we implement the optimal matching algorithm? Is it even possible outside of PROC SQL?

SYNCHRONIZED JOIN – CONCEPTUALIZATION

If we only have to deal with a one-to-many relationship between orders and quotes, we can rely on the DATA step match merge. Instead, we are stuck with a many-to-many relationship, for which the match merge is useless. Still, the DATA step is our only hope outside of PROC SQL. The dilemma here is to replicate the many-to-many relationship that PROC SQL naturally spawns without actually performing the full Cartesian join. We want the benefits but not the costs.

The answer is the Synchronized Join algorithm, or SyncJoin for short. We start by creating a many-to-many relationship using the DATA step, as follows:

```
data many_to_many (drop=nextobs);
  set orders;

  do nextobs = 1 to quotescount;
    set quotes (rename=(stock_name=quote_stock_name)) nobs=quotescount;
    output;
  end;
run;
```

Figure 5

When the DATA step in Figure 5 begins execution, an observation from the orders data set is first brought into the PDV. Then, through a DO loop, observations from the quotes data set are brought in, one at a time, to join the order observation already in the PDV. This in effect combines the order with each quote to form a single, extended observation, which is promptly output to a data set before the next quote is brought in. When the DO loop is finished, the single observation from the orders data set has been matched with all observations from the quotes data set, a perfect one-to-many relationship. It does not stop there. By default, the DATA step execution immediately moves to the next observation in the orders data set and faithfully repeats the process of matching the new order with every quote record. When all observations from the orders data set have been processed, a perfect many-to-many relationship has been replicated. Bravo!

Sadly, the DATA step above also duplicates the Cartesian join. We already know that the fatal flaw with PROC SQL is its indiscriminate matching of all quotes with every order, even though each order should really be matched with a small number of quotes within a tiny time window around the order entry time. If we sort the orders and quotes by order time and quote time respectively, we may be able to avoid bringing in quotes that are either too early or too late relative to the order at hand. In other words, we need the ability to select observations at will from the quotes data set to match each individual order. The POINT= option for the SET statement provides the random access that proves ideal for our purpose. The modified code appears below.

```
data many_to_many;
  set orders;

  do nextobs = 1 to quotescount;
    set quotes (rename=(stock_name=quote_stock_name)) nobs=quotescount
      point=nextobs;
    output;
  end;
run;
```

Figure 6

However, we instantly see a problem with the new code. The pointer variable, **nextobs**, is still looping through all observations of the quotes data set rather than selecting targeted observations close to the order at hand. Nothing has really changed. If we can find a way to direct the pointer variable to appropriate observations in the quotes data set, then we no longer need to loop through all the quotes. On the other hand, due to the many-to-many relationship, we still need a loop to make sure that we have examined all quotes that are reasonably close to the current order. A DO WHILE loop with appropriate conditions is just what we need.

```
data smart_loop (drop=continue);
  set orders;

  continue = 1;

  do while (continue);
    set quotes (rename=(stock_name=quote_stock_name)) nobs=quotescount
      point=nextobs;
    output;
  end;
run;
```

Figure 7

This is a promising framework, but more needs to be done. We need to reset the conditional variable **continue** to 0 at some point when no more eligible quotes exist for the current order. Similarly, we need to adjust the pointer variable **nextobs** constantly so that it only points to those quotes that are truly close to the order at hand. The table below lists six basic scenarios regarding the relative timing of orders and quotes. Each displayed time represents an order or quote's entry time. The quote ending time is not displayed for the quotes. Take a look at Scenario I. An order is entered at 10:20:30AM. It can be matched to five quotes of different times if they each have a life span that encompasses the order entry time. This represents the majority of cases. The "match" column shows the status of the matching: "early" means the quote is too early for the order, "late" denotes a quote entered after the order, "skip" means no eligible quotes at all for the order, and "first" through "fifth" indicates a matched quote.

Scenario I			Scenario II			Scenario III		
Order	Quotes	Match	Order	Quotes	Match	Order	Quotes	Match
	10:20:25	Early		10:20:25	Early	08:00:00	08:00:00	1 st
nextobs→	10:20:26	1st	10:20:30		Skip	nextobs *	08:00:01	Late
	10:20:27	2 nd	nextobs→	10:20:31	Late		08:00:02	Late
	10:20:28	3 rd		10:20:32	Late		08:00:03	Late
	10:20:29	4 th		10:20:33	Late		08:00:04	Late
10:20:30	10:20:30	5 th		10:20:34	Late		08:00:05	Late
	10:20:31	Late		10:20:35	Late		08:00:06	Late
Majority cases – multiple matches for the order			Occasional cases – no match for the order			Boundary cases – same-time first order matched to first quote		
Scenario IV			Scenario V			Scenario VI		
Order	Quotes	Match	Order	Quotes	Match	Order	Quotes	Match
08:00:00		Skip		17:59:54	Early		17:59:54	Early
nextobs→	08:00:01	Late		17:59:55	Early		17:59:55	Early
	08:00:02	Late	nextobs→	17:59:56	1st	nextobs→	17:59:56	1st
	08:00:03	Late		17:59:57	2 nd		17:59:57	2 nd
	08:00:04	Late		17:59:58	3 rd		17:59:58	3 rd
	08:00:00	Late		17:59:59	4 th		17:59:59	4 th
	08:00:01	Late	18:00:00	18:00:00	5 th	18:00:00		
Boundary cases – no same-time first quote for first order			Boundary cases – same-time last order matched to last quote			Boundary cases – no same-time last quote for last order		

*Note: In Scenario III, the **nextobs** variable should point to the first quote at 08:00:00.

Figure 8

The implementation of the optimal matching algorithm needs to cover all six basic scenarios. But that is not enough. You may have noticed a lack of duplicate times in each scenario. With a many-to-many relationship, each of the time entries, i.e., each order or quote, potentially can multiply into up to 10 orders or quotes of the same time. That essentially quadruples the number of scenarios we have to deal with. Take Scenario I, for example. The basic scenario, as listed, is one order matched to 5 quotes of different times. An expansion of this scenario adds three more variations: one order matched to up to 50 quotes of different times; up to 10 orders of the same time matched to 5 quotes of different times; and finally, up to 10 orders of the same time matched to up to 50 quotes of different times.

Another complication is the varying life span of each individual quote. We again use Scenario I as an example. Suppose the second match, the quote at 10:20:27AM, has only a life span of 2 seconds. The optimal algorithm should be smart enough to know that such a quote is not a good match for the order at 10:20:30AM. At the same time, we must ensure that this failed match does not stop us from looking further down the time line in the quotes table. Otherwise, we would have missed the third, fourth, and fifth eligible quotes, which may be good matches.

Finally, we need an effective mechanism to remember the position of the appropriate quote observation to go back to in cases of multiple orders of the same time. Suppose we have another order at 10:20:30AM following the one listed in Scenario I. After successfully matching the listed order to all five displayed quotes, we need to go back to the first matched quote and repeat the matching process for the second order of the same time. Obviously, going back any further than the first matched quote is a waste of computing resources, but going to any quote after the first matched quote will cause lost matches. We have to go straight back to the first matched quote for every order of the same time, up to a total of 10 orders. What a delicate situation! Similarly, when an order can not find a matching quote, we need to make sure that subsequent orders, either of the same time or later, go directly to the most likely quote for a match, again without wasting resources or missing legitimate quotes. The various boundary cases need to be considered in light of this, too.

The following DATA step successfully addresses all of the above issues. After proper sorting, the output should be identical to the table produced by the slow-running PROC SQL in Figure 3.

```
data matches_syncjoin (drop=continue firstmatchobs);
  retain nextobs 1 firstmatchobs;
  set orders;
  by order_time;

  if first.order_time then
    firstmatchobs = 0;

  continue = 1;

  do while (continue);
    set quotes (rename=(stock_name=quote_stock_name)) nobs=quotescount
              point=nextobs;

    if quote_end_time < order_time then do;
      if nextobs < quotescount then
        nextobs + 1;
      else
        continue = 0;
    end;
    else if quote_time > order_time then do;
      continue = 0;

      if firstmatchobs > 0 then
        nextobs = firstmatchobs;
    end;
    else do;
      if firstmatchobs = 0 then
        firstmatchobs = nextobs;

      if order_side = "BUY" and
        sell_quantity = order_quantity and
        sell_price = order_price
      or
```

```

        order_side = "SELL" and
        buy_quantity = order_quantity and
        buy_price = order_price then
        output;

    if nextobs < quotescount then
        nextobs + 1;
    else do;
        continue = 0;

        if firstmatchobs > 0 then
            nextobs = firstmatchobs;
        end;
    end;
end;
run;

```

Figure 9

SYNCHRONIZED JOIN – EXPLANATION

The SyncJoin method, as exemplified by the code above, can be broken down into six steps:

STEP I: Sort the two data sets by the key variables used in the cross-table between condition.

In our case, we have sorted the orders by the order time, and the quotes by the quote entry time. It is important to sort the variables in the same order, preferably ascending. If using the descending order, then the quotes need to be sorted by the quote ending time, not the quote entry time. In that case, the orders also need to be sorted by the order time in descending order. We usually know which one of the two between variables (quote entry time and quote ending time) is bigger. Otherwise, the SyncJoin method will not work! This step is actually completed before entering the DATA step.

STEP II: Pick the data set whose key variable is listed first in the between condition as an anchor.

This is vital when we construct the if-else conditions to compare the key variable values. In our case, order time is listed first in the between condition, so we use the orders data set as the anchor. We can visualize the numerous one-to-many relationships as we work through the anchor data set and link each order to multiple quotes from the second data set.

STEP III: Rename all the common variables in one data set to avoid overwriting.

When an observation from the second data set is brought into the PDV, all common variables already in the PDV (the anchor data set's record) will be overwritten. The key variable from the anchor data set, which appears in the between condition, should never be overwritten. We need its value for current and future comparisons. Other common variables may be overwritten if desired, but it is probably better to preserve their values by renaming them. For simplicity, rename only one data set's common variables.

STEP IV: Retrieve the next observation from the second data set using random access (**POINT=**).

At the very beginning of the DATA step, the pointer variable **nextobs** is initialized to 1 so that the first DO WHILE loop starts with the first observation in the second data set. After that, it is constantly adjusted to select an appropriate observation from the second data set. The pointer variable is also retained so that the next observation from the anchor data set does not have to start from the very beginning of the second data set to look for eligible matches. Instead, it can generally pick up where its predecessor has left off, thus eliminating the indiscriminate, unnecessary matches committed by a Cartesian product join. Obviously, it is essential to point the variable **nextobs** to the correct observation in the second data set once the current observation from the anchor data set (already in PDV) is done.

Here we need to reiterate a key fact. When one observation from the anchor data set sits right next to one observation from the second data set in the PDV, the two observations are in effect joined into an extended temporary record. This fact is crucial to the SyncJoin method. It also accounts for the "Join" part of the SyncJoin name. Conceptually, it is very similar to what a Cartesian product join does when it pairs one

record from one table with one record from the second table. In both cases, an extended temporary record is created for further processing. This similarity should boost our confidence in the SyncJoin algorithm as we attempt to replace the full Cartesian join with a DATA step implementation.

STEP V: Compare the current observation from the anchor data set with the newly retrieved observation from the second data set on key variables to find an initial match.

In our case, we need only to compare the order time with the quote entry time and quote ending time to ensure that the quote is effective for the order. In more complicated SyncJoin implementations, we may need to compare more than one set of key variables to find an initial match. For example, if we have to deal with more than one stock in our orders and quotes, then the first comparison will be on the stock name. Upon a positive match, we will move on to the time comparison. That in turn requires the proper sorting of the data sets on the key variables in **STEP I**, first by stock name, then by time. A between condition's key variable should always be the second by variable in sorting, after the key variable from an equality condition. Due to significant fragmentation of the data after two sets of key variables, it probably does not pay to compare any more sets of key variables, even on data sets with multimillion observations. You would be better off simply constructing an IF statement that incorporates any further key variables which may exist in an SQL statement's where clause.

As you can see from the code in Figure 9, this step is the heart of the SyncJoin method. By comparing the key variables' values, we can determine the relative positions of the two observations currently in the PDV (one from each data set), and see which one is ahead and which one is behind. If the order time overshoots the quote ending time, we know that the current quote is too early (or too old?) for the current order, and the pointer variable `nextobs` is incremented to retrieve the next, potentially later quote during the next iteration of the DO WHILE loop. On the other hand, if the order time precedes the quote entry time, we know there are no more effective quotes for the current order. They have either been matched already with the current order during earlier iterations of the DO WHILE loop, or they are simply nonexistent in the first place. It is time to wrap up the current order and move on to the next order. To do that, we get out of the DO WHILE loop by resetting the conditional variable `continue` to 0, as false. By default, the DATA step then advances its hidden pointer to the next observation in the anchor data set, just as we want. It is noteworthy that throughout the process, the SyncJoin execution always pulls one observation at a time, either through the DATA step default behavior or by incrementing the pointer variable `nextobs` in the DO WHILE loop.

These maneuvers in effect synchronize the observations from the two data sets on the key variables, hence the "Sync" part of the name SyncJoin. They ensure that observations from the two data sets advance in lockstep. Whenever a gap appears, the trailing data set is swiftly brought up to speed with the leading data set. At any given moment, observations from the two data sets are either matched with each other or constantly leapfrogging each other in search of the next match. There is only one single, steady pass through the anchor data set; no observation from the anchor is ever brought into the PDV twice! Similarly, there is generally a single pass through the second data set, with as many loop-back minipasses through narrowly targeted observations as necessary. The loop-back minipasses happen when multiple observations in the anchor data set have duplicate key values that can be matched to observations from the second data set. Put together, these minipasses are just like Cartesian product joins, except they are minimal in scale.

Let's look at an example of minipasses. Suppose there are three orders at 10:20:30AM rather than a single order in Scenario I above. In this case, we need to make three passes through the five potentially eligible quotes, one for each order. The first order always establishes the range of observations in the second data set that subsequent orders need to pass through. Only those identified observations are then brought back into the PDV for matching when subsequent orders of the same time are in the PDV. No loop-back minipass is necessary if there are no duplicate key values in the anchor data set's observations. Scenario I, as it stands in Figure 8, does not require a loop-back minipass because there is only one order.

A successful combination of the two minimalist characteristics in our data set processing – a single pass through the anchor data set, and a single pass through the second data set wherever possible, with only necessary loop-back minipasses – explains the enormous performance improvement that the SyncJoin algorithm achieves over the full Cartesian product joins chosen by SAS PROC SQL or Oracle. The passes through the two data sets have indeed been reduced to the absolute minimum. Any further reduction will inevitably lead to a loss of legitimate matches. This may sound like boasting, but there is really no room for meaningful improvement to the SyncJoin method!

STEP VI: Perform the necessary operations on the matched / joined observations.

Once an initial match is found, the other conditions in the where clause of the original SQL statement can be checked with a simple (or complex) IF statement. As mentioned in **Step V**, if there are more than two sets of key variables to compare, the additional ones can be weaved into the IF statement without any noticeable loss in performance. Don't forget to output the final results to a data set at the end of this step.

Throughout **STEP V** and **STEP VI**, a couple of other important details are also taken care of, where appropriate. In the previous section, we talked about the need to remember where to go back to in cases of multiple orders of the same time. **STEP V** further demonstrated this capability's critical contribution to the minimal-scale Cartesian joins, or minipasses. The retained variable `firstmatchobs`, through the by statement associated with the anchor data set and conditional interactions with the pointer variable `nextobs`, successfully accomplishes the mission. Separately, the temporary variable `quotescount` guarantees that the pointer variable `nextobs` never goes out of bounds with the second data set.

You may have wondered earlier why the scenarios in Figure 8 are all constructed from the perspective of the orders. There is always one order, but there may or may not be one or multiple eligible quotes. This is because we treat the orders data set as the anchor, as explained in **STEP V**. The SyncJoin algorithm always begins with an observation from the anchor data set, before looking for potential matching observations from the second data set.

RESULTS AND DISCUSSION

The SyncJoin algorithm, when tested with the same data sets as those in the **BENCHMARK TEST** in Figure 4, produced exactly the same results in every case, thereby proving its functional equivalence to the PROC SQL code in Figure 3. To level the playing field, the SQL statement does not include an order by clause because the results from the SyncJoin method always need to be sorted separately. The two sets of results are then sorted and passed to the SAS COMPARE procedure for verification. A truly accurate performance comparison should pit the PROC SQL total CPU time against the SyncJoin total CPU time, which includes the presorting in **STEP I**. In the end, the numbers were so terribly lopsided, as the table below attests, that the SyncJoin presorting CPU time was ignored. As a result, the table contains only the CPU times by PROC SQL and the DATA step proper.

Orders	Quotes	Time Category	PROC SQL	SyncJoin	CPU Time Ratio
98,781	99,465	Real Time	00:27:00.94	00:00:03.26	497
		User CPU Time	00:25:16.08	00:00:01.50	1010
		System CPU Time	00:01:06.85	00:00:01.22	54
		Total CPU Time	00:26:22.93	00:00:02.72	581
197,667	197,799	Real Time	01:44:30.85	00:00:06.10	1028
		User CPU Time	01:39:45.53	00:00:02.91	2056
		System CPU Time	00:04:07.85	00:00:02.63	94
		Total CPU Time	01:43:53.38	00:00:05.54	1125
296,489	297,046	Real Time	04:02:49.62	00:00:8.48	1718
		User CPU Time	03:46:27.28	00:00:04.30	3159
		System CPU Time	00:10:35.98	00:00:03.77	168
		Total CPU Time	03:56:63.26	00:00:08.07	1762
395,122	395,512	Real Time	06:58:22.52	00:00:10.90	2302
		User CPU Time	06:39:02.28	00:00:06.10	3924
		System CPU Time	00:17:12.93	00:00:04.30	240
		Total CPU Time	06:56:15.21	00:00:10.40	2401
496,585	496,119	Real Time	11:08:51.66	00:00:14.72	2726
		User CPU Time	10:29:38.33	00:00:07.22	5232
		System CPU Time	00:31:06.48	00:00:06.27	297
		Total CPU Time	11:00:44.81	00:00:13.49	2938
990,931	989,682	Real Time	43:38:07.66	00:00:27.26	5762
		User CPU Time	41:39:39.75	00:00:14.89	10072
		System CPU Time	01:47:38.08	00:00:11.52	560
		Total CPU Time	43:27:17.83	00:00:26.41	5923

Figure 10

Comparing the numbers side by side, one thing that immediately stands out is the performance differences. They are so vast as to be absolutely incredible! With 100,000 orders and quotes, the SyncJoin method required only 2.72 seconds compared with PROC SQL's 26 minutes. When the orders and quotes increased to one million each, the SyncJoin algorithm needed less than half a minute, while PROC SQL took almost two full days! That is a performance ratio approaching 6000 to 1. Can you imagine how long PROC SQL will make you wait if the two data sets have 10 million observations each? If you choose to do the queries in Oracle, you will get excruciatingly sluggish performance too, remarkably similar to what SAS PROC SQL did.

The SyncJoin algorithm does not work wonders with the CPU time alone. It literally requires no extra computing resources in terms of disk space, memory, and I/O. As long as your computing environment can store the two data sets, you are all set. All the algorithm does is pulling one record from the anchor data set and hold it in the PDV, pulling one record from the second data set and comparing it with the anchor data set observation. This process is repeated only as necessary. Technically speaking, the memory and I/O resources may feel a little stress for an extremely short period of time while the algorithm is blasting its way through the data sets. With a Cartesian join, all the resources are likely to be stressed to the limit for the entire, and much longer, duration. In fact, the SyncJoin algorithm was conceived during an episode of extremely stressful computing in the author's Solaris SAS environment, and Oracle did not perform any better than SAS PROC SQL.

A close examination of the performance numbers reveals that the steadily rising performance ratio adheres very well to the mathematical calculations in an earlier section. There is a clearly linear relationship between the number of quotes and the performance ratio, as reflected by the respective total CPU times. Back in the **ANALYSIS** section, we predicted $(N / 8.25)$ as the performance ratio between the optimal matching algorithm and SAS PROC SQL, where N represents the number of quotes. Basically, the more quotes there are, the more PROC SQL and, by extension, Oracle and other RDBMS products, are penalized for resorting to the Cartesian product join. The SyncJoin method is truly the optimal matching algorithm. Otherwise, we might not see the linear relationship. This trend also indirectly confirmed our assumption that each attempted match between an order and a quote costs about the same amount of CPU time with large data sets.

Another predicted relationship, $(M * K)$ as the number of attempted matches by the optimal algorithm, is also confirmed by the results, albeit indirectly. In the **ANALYSIS** section we used M to represent the number of orders while K was a constant number predetermined by the way the data sets are generated, in our case, $K = 8.25$. This linear relationship depends on the number of observations in the orders data set. As a proxy measure of the number of matches attempted, the total CPU time under the SyncJoin method indeed exhibited an unmistakable correlation with the rising orders count.

Although the SyncJoin algorithm is powerful and efficient, it has its own limitations. First and foremost is its narrow applicability. A between condition can be either intra-table or cross-table. All current SQL optimizing engines can handle intra-table between conditions efficiently. The SyncJoin method picks up where the built-in optimizations leave off. It is effective only when the following conditions are met: (1) one variable from one data set is compared with two variables / values from another data set via the between operator, and (2) the two variables / values, which define the value range for the between operator, are closely related to each other in a predictable way. There are only two cases for the second condition: either one variable is created by adding a constant value to the other variable, as in the second DATA step in Figure 1, or one variable is created by adding a varying value to the other variable, like the quote ending time when quotes are generated by the DATA step in **APPENDIX**. If the two variables / values are not related, or their relative values are inconsistent, the SyncJoin algorithm, as currently implemented, probably requires major modifications in order to equal PROC SQL in its thoroughness. The stellar performance we have witnessed so far may suffer as a result.

The SyncJoin implementation in Figure 8 can be tweaked to target one-to-one and one-to-many-relationships. In those expanded capacities, the algorithm performs similarly to the DATA step match merge, but with slightly longer runtime due to the added overhead of checking and resetting temporary variables. Since almost all built-in query-optimizing engines can handle the two relationships rather efficiently, it is better to use SAS PROC SQL or Oracle for the job instead of customizing the SyncJoin implementation. Again, the performances are remarkably similar.

The above caveat presumes that you know the relationship between the two data sets, i.e., whether it is a one-to-one or one-to-many relationship. Furthermore, the relationship should remain constant. But what if you don't know the relationship? What if they change over time? In case of any uncertainty, the current implementation should always be used without modification. Because a many-to-many relationship encompasses both one-to-one and one-to-many relationships, the current version will always work. You may take a small hit in performance from time to time, but you will at least get the correct results every time. More importantly, you will never be stuck with Cartesian joins when an unexpected many-to-many relationship pops up in your data.

If you are ever tempted to modify the SyncJoin implementation to deal with equi-joins only, especially on multiple key variables, don't! It is certainly a good exercise that helps your understanding of the algorithm, but the performance of the modified version is always slightly inferior to the DATA step match merge (if applicable) and standard SQL implementations, again thanks to the added overhead of checking and resetting temporary variables. There is simply no advantage to using the SyncJoin method for equi-joins.

Finally, when you are ready to adapt the sample SyncJoin implementation to your specific query needs, do watch out for infinite loops, a common pitfall accompanying the SET statement's random access option. Writing the key variables from both the anchor and second data sets to the log, as well as the current values of the temporary variables such as `continue`, `nextobs`, and `firstmatchobs`, should instantly alert you to the problem. You know you have fallen into the trap when the SAS log window flashes by like crazy and all the displayed values are identical from one line to the next. It happens to everyone, so don't feel bad.

Aside from the cross-table between condition, there are a number of other conditions that automatically trigger full Cartesian joins in SAS PROC SQL and Oracle. The SyncJoin algorithm provides a good first step towards tackling those conditions. You will be pleasantly surprised by the performance gains you may achieve. In any event, as a SAS programmer who knows the data inside out, you are in a position to find the best solutions to your problems. Yes, you are allowed to beat SAS PROC SQL and Oracle once in a while and get away with it. Someone just did!

CONCLUSION

We can generally trust the built-in query optimizations provided by SAS PROC SQL and other RDBMS products such as Oracle for most of our joining needs. Sometimes, though, we need to come up with our own unique solutions when the standard optimizations fall short. The cross-table between condition is a prime example. The SyncJoin algorithm uses the SAS DATA step to conquer the condition in an innovative and efficient manner. It produces identical results to PROC SQL and, at the same time, successfully avoids the performance nightmares associated with a Cartesian join of large data sets. It also opens the door to more customized solutions based on the DATA step.

REFERENCES

Kent, Paul. 1997. SAS Institute Technical Support Notes TS-553. "SQL Joins – The Long and The Short of It" <<http://support.sas.com/techsup/technote/ts553.html>>

Lavery, Russ. "The SQL Optimizer Project: _Method and _Tree in SAS® 9.1" *Proceedings of the Thirtieth Annual SAS® Users Group International Conference*. April 2005. <<http://www2.sas.com/proceedings/sugi30/101-30.pdf>>

CONTACT INFORMATION

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

Houliang Li
(301) 682-6832
houliang_li@yahoo.com

APPENDIX

```
/*
This macro program first generates the orders and quotes data sets, then joins them
using both the SAS PROC SQL and the SyncJoin algorithm, and finally compares the
results from the two approaches. By systematically changing the parameter values and
invoking the macro repeatedly, the performance differences between the SyncJoin method
and the SAS PROC SQL, in terms of total CPU time required, are clearly demonstrated.
```

Here are the macro parameters:

```
days = <number of days; used to control the number of observations generated>
multipleorder = <yes to allow multiple same-time orders; no for single order>
multiplequote = <yes to allow multiple same-time quotes; no for single quote>
quoteduration = <maximum number of seconds that a quote can last, from 1 to 5>
variableduration = <yes to allow quote duration to vary from 1 to 5; no to fix to 5>
*/
```

```
%macro sql_vs_syncjoin (days=10, multipleorder=yes, multiplequote=yes,
                        quoteduration=5, variableduration=yes);
```

```
options fullstimer;
```

```
/* Generate an orders dataset for the stock ABCD */
```

```
data orders (keep=order_id stock_name order_time customer_id order_side order_price
               order_quantity filler_order);
  format order_id 8. stock_name $8. order_time datetime20. customer_id order_side $8.
         order_price 8.2 order_quantity 8. filler_order $456.;
```

```
  stock_name = "ABCD";
  filler_order = "<Put random characters in the string to make it long or short>";
```

```
  begin = today() * 86400 + "8:00:00"t;
  end = (today() + &days - 1) * 86400 + "18:00:00"t;
```

```
  do time = begin to end;
    random = ranuni(0);
```

```
    %if %upcase(&multipleorder) = YES %then
      order_count = int(10 * random) + 1;
    %else
      order_count = 1;;
```

```
    do counter = 1 to order_count;
      random = ranuni(0);
```

```
      if random < 0.5 then do;
        order_id + 1;
        order_time = time;
        customer_id = "cs_" || compress(int(random * 10000));
```

```
        if random < 0.25 then do;
          order_side = "BUY";
          order_price = 10 + mod(round(random, 0.1), 0.2);
        end;
```

```
        else do;
          order_side = "SELL";
          order_price = 10 - mod(round(random, 0.1), 0.3);
        end;
```

```
        order_quantity = round(1000 * mod(random, 0.3), 100) + 100;
```

```

        output;
    end;
end;

    if timepart(time) = "18:00:00"t then
        time + 50399;
    end;
run;

/* Generate a quotes dataset for the stock ABCD */

data quotes (keep=quote_id stock_name quote_time quote_end_time broker_id buy_price
              buy_quantity sell_price sell_quantity filler_quote);
    format quote_id 8. stock_name $8. quote_time quote_end_time datetime20.
           broker_id $8. buy_price 8.2 buy_quantity 8. sell_price 8.2 sell_quantity 8.
           filler_quote $440.;

    stock_name = "ABCD";
    filler_quote = "<Put random characters in the string to make it long or short>";

    begin = today() * 86400 + "8:00:00"t;
    end = (today() + &days - 1) * 86400 + "18:00:00"t;

    do time = begin to end;
        random = ranuni(0);

        %if %upcase(&multiplequote) = YES %then
            quote_count = int(10 * random) + 1;
        %else
            quote_count = 1;;

        do counter = 1 to quote_count;
            random = ranuni(0);

            if random < 0.5 then do;
                quote_id + 1;
                quote_time = time;

                %if %upcase(&variableduration) = YES %then
                    quote_end_time = quote_time + mod(int(random * 10), &quoteduration);
                %else
                    quote_end_time = quote_time + 4;;

                broker_id = "bk_" || compress(int(random * 10000));
                buy_price = 10 - mod(round(random, 0.1), 0.2);
                sell_price = 10 + mod(round(random, 0.1), 0.3);

                if buy_price = sell_price then
                    sell_price + 0.2;

                if random < 0.125 then do;
                    buy_quantity = 300;
                    sell_quantity = 100;
                end;
                else if random > 0.375 then do;
                    buy_quantity = 100;
                    sell_quantity = 300;
                end;
                else do;
                    buy_quantity = 200;
                    sell_quantity = 200;
                end;
            end;
        end;
    end;
end;

```

```

        output;
    end;
end;

    if timepart(time) = "18:00:00"t then
        time + 50399;
    end;
run;

/* Link orders to effective quotes - Rely on Proc SQL query optimizations */

proc sql noprint _method;

    create table matches_sql as
    select *
    from orders a,
        quotes (rename=(stock_name=quote_stock_name)) b
    where a.order_time between b.quote_time and b.quote_end_time
        and (a.order_side = "BUY" and
            b.sell_quantity = a.order_quantity and
            b.sell_price = a.order_price
        or
            a.order_side = "SELL" and
            b.buy_quantity = a.order_quantity and
            b.buy_price = a.order_price);

quit;

/* Link orders to effective quotes - Use the SyncJoin algorithm */

/* Sort the orders and quotes properly */

proc sort data=orders;
    by order_time;
run;

proc sort data=quotes;
    by quote_time quote_end_time;
run;

/* Match orders with effective quotes */

data matches_syncjoin (drop=continue firstmatchobs);
    retain nextobs 1 firstmatchobs;
    set orders;
    by order_time;

    if first.order_time then
        firstmatchobs = 0;

    continue = 1;

    do while (continue);
        set quotes (rename=(stock_name=quote_stock_name)) nobs=quotescount
            point=nextobs;

        if quote_end_time < order_time then do;
            if nextobs < quotescount then

```

```

        nextobs + 1;
    else
        continue = 0;
    end;
    else if quote_time > order_time then do;
        continue = 0;

        if firstmatchobs > 0 then
            nextobs = firstmatchobs;
        end;
    else do;
        if firstmatchobs = 0 then
            firstmatchobs = nextobs;

            if order_side = "BUY" and
                sell_quantity = order_quantity and
                sell_price = order_price
            or
                order_side = "SELL" and
                buy_quantity = order_quantity and
                buy_price = order_price then
                output;

            if nextobs < quotescount then
                nextobs + 1;
            else do;
                continue = 0;

                if firstmatchobs > 0 then
                    nextobs = firstmatchobs;
                end;
            end;
        end;
    end;
run;

/* Sort the results and compare them */

proc sort data=matches_sql;
    by order_id quote_id;
run;

proc sort data=matches_syncjoin;
    by order_id quote_id;
run;

proc compare data=matches_sql compare=matches_syncjoin;
run;

%mend;

/* Run the macro repeatedly and collect the statistics */

%sql_vs_syncjoin (days=1);
%sql_vs_syncjoin (days=2);
%sql_vs_syncjoin (days=3);
%sql_vs_syncjoin (days=4);
%sql_vs_syncjoin (days=5);
%sql_vs_syncjoin (days=10);

```