Paper CC10

Date Dimension

Christopher Johnson, BrickStreet Insurance

ABSTRACT

Intuition would suggest that it is more efficient to perform simple calculations as needed than to store calculations in a table for reference. However, in some circumstances, creating lookup tables can save both programmer and CPU time. Dates present a particular difficulty in any programming language. This paper will present a data structure that can simplify date manipulations while gaining efficiency.

INTRODUCTION

When writing a program in any language, a programmer should have the goal of writing the most efficient program possible. However, efficiency can have many meanings, as seen in Figure 1 below. Depending on the situation and project requirements, one may want to minimize the CPU usage, minimize memory usage, or simplify program coding. Generally, the best result comes from trying to balance these concerns so that no particular issues presents a bottleneck to the program. In the code that follows, we will reduce CPU and programming time with a small sacrifice in memory.

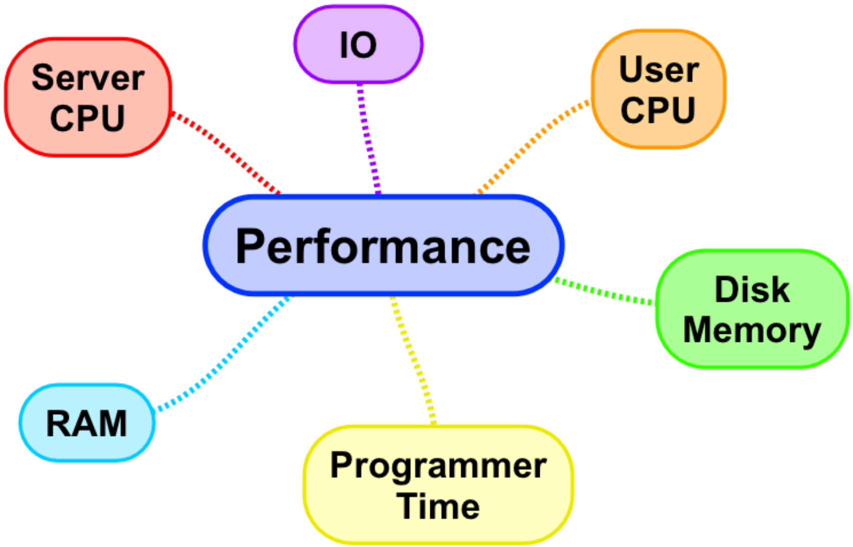


Figure 1. Components of Program Efficiency

One of the first hurdles to overcome when learning a new programming language is how to handle dates. Formats can vary from text to numeric, from mm/dd/yyyy to ddmmmyyyy or yyyyddmm and more. Syntax for date functions vary widely. For example, in SAS®, dates should be references as ‘mmddyyyy’d. When pulling a date from Oracle, you get a timestamp that must be dealt with using the DATEPART function. SAS functions like INTNX are unique to SAS, and the syntax must be learned. More concerning is that formulas to derive values such as the number of quarters between two dates can become, long, nested, inefficient, and error prone. We will derive a lookup table for some common situations and test the resulting efficiency.

COMMON CALCULATIONS

Using an insurance application, we will look at a common set of data calculations. These calculations will be compared to our data dimension later to determine the efficiency of each.

In the code below, we use a PROC SQL statement to perform calculations of a table of values called DATA. This table contains three variables: claim_num, date_id, and accident_date. We must calculate the quarter of date_id (qop), the quarter of accident_date (qoi), and the number of quarters between the two (qtr). In this example, I worked with 2,624,176 observations.

```
/*Calculate values*/
proc sql;
```

```
       create table calculations as
       select claim_num,
              date_id,
              accident_date,
              input(case when month(date_id) <= 3 then '31mar'
                     when month(date_id) <= 6 then '30jun'
                     when month(date_id) <= 9 then '30sep'
                     else '31DEC' end
                     || put(year(date_id),4.), date9.) format=mmddyy8. as qop,
              input(case when month(accident_date) <= 3 then '31mar'
                     when month(accident_date) <= 6 then '30jun'
                     when month(accident_date) <= 9 then '30sep'
                     else '31DEC' end
                            || put(year(accident_date),4.), date9.)
                            format=mmddyy8. as qoi,
              ((((year(calculated qop)-year(calculated qoi))*12)
                     +(month(calculated qop)-month(calculated qoi)))/3) + 1 as
                     qtr
       from data;
quit;
```

CODING A DATE DIMENSION

A dimension is a table of values utilizing a unique key and a variety of characteristic variables. The following code use two DATA steps to create a date dimension that stores many commonly used date characteristics. It takes advantage of such SAS functions as INTNX, DAY, MONTH, YEAR, NLDATE, and WEEKDAY in order to maximize the efficiency of the calculations. However, these calculations only need to occur once. The first DATA step iteratively creates the dates to be used for the key, and the second DATA step calculates the needed values for each.

In the code below, there are two values that need to be set: the begin date and the end date for the iterations. I have chosen to create a dimension covering 200 years (73,049 observations). However, this can be increased or decreased depending on the need of the user, and this will affect program efficiency.

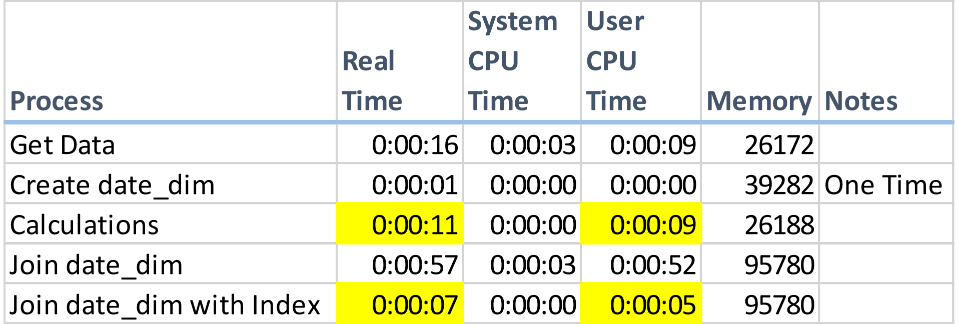
```
/*Create date_dim*/
data testing.date_values;
       format date_key date10.;
       do date_key = '01jan1900'd to '31dec2099'd;
       output;
       end;
run;
data date_dim;
       set testing.date_values;
       format date_key date10.
              day month 2. year 4. julian 3.
              weekday weekdayname $9.
              fst_dd_mth fst_dd_qtr fst_dd_hf fst_dd_yr
              lst_dd_mth lst_dd_qtr lst_dd_hf lst_dd_yr date10.;
       day = day(date_key);
       month = month(date_key);
       year = year(date_key);
       julian = date_key - input('01jan' || put(year(date_key), 4.), date9.)+ 1;
       weekday = weekday(date_key);
       weekdayname = nldate(date_key,'%a');
       fst_dd_mth = intnx('month', date_key, 0, 'beginning');
       fst_dd_qtr = intnx('quarter', date_key, 0, 'beginning');
       fst_dd_hf = intnx('semiyear', date_key, 0, 'beginning');
       fst_dd_yr = intnx('year', date_key, 0, 'beginning');
       lst_dd_mth = intnx('month', date_key, 0, 'end');
       lst_dd_qtr = intnx('quarter', date_key, 0, 'end');
       lst_dd_hf = intnx('semiyear', date_key, 0, 'end');
       lst_dd_yr = intnx('year', date_key, 0, 'end');
run;
```

`The following code performs the same calculations as our initial PROC SQL above, however, it uses two joins and one subquery to our date dimension.  This code is both noticeably shorter and simpler to write than the original code.`

```
        /*Join date_dim*/
        proc sql;
                create table join as
                select data.claim_num,
                        data.date_id,
                        data.accident_date,
                        qoidate.lst_dd_qtr as qoi,
                        qopdate.lst_dd_qtr as qop,
                        (select count(*) as count
                        from testing.date_dim
                        where date_key between qoidate.lst_dd_qtr and qopdate.lst_dd_qtr
                                and date_key = lst_dd_qtr) as qtr
                from data,
                        date_dim qoidate,
                        date_dim qopdate
                where data.accident_date = qoidate.date_key
                        and data.date_id = qopdate.date_key;
        quit;
```

BENCHMARKING

Now we will look at the efficiency of both methods. I have used the SAS Option FULLSTIMER to calculate the following values.



Process	Real Time	System CPU Time	User CPU Time	Memory	Notes
Get Data	0:00:16	0:00:03	0:00:09	26172	
Create date_dim	0:00:01	0:00:00	0:00:00	39282	One Time
Calculations	0:00:11	0:00:00	0:00:09	26188	
Join date_dim	0:00:57	0:00:03	0:00:52	95780	
Join date_dim with Index	0:00:07	0:00:00	0:00:05	95780	

Figure 2. Benchmarking Results

In Figure 2, we can see the statistics for pulling the original data, creating the data dimension, calculating the values using functions, and pulling the values from our dimension (with and without indexing the dimension). Getting the data and creating the dimension can be neglected, as these are one time calculations.

The results were compared on a Windows client machine running SAS Enterprise Guide with the calculations running on a separate SAS server. Both the server and client CPU times are shown. Notice that running pulling the values from the dimension resulting in about a 40% reduction in both real time and CPU time. We also need to point out that without indexing the dimension, both values are dramatically increased. Also note that the memory usage was increased using the dimension, but not to the point of becoming a problem for our system.

Furthermore, we could continue by joining the date dimension to the original data using a hash object, which does further reduce the time, however, this does not allow the calculation of the number of quarters, and so will be omitted.

CONCLUSION

Our results show that calculating and storing the data values resulted in a reduction in both programmer and CPU time. The code is also simpler to read and less prone to programmer error. However, this approach may not be beneficial in a situation where the dimension is large enough to cause a reduction in performance due to memory issues.

CONTACT INFORMATION

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

Christopher Johnson
BrickStreet Insurance
400 Quarrier Street
Charleston, WV 25301
(304) 941-1000 Ext. 5359
(304) 941-1186
Christopher.Johnson@BrickStreet.com
www.BrickStreet.com