Paper 7580-2016

Architecting Data Management: Seven Principles Using SAS®, DataFlux® and SQL

Bob Janka, Modern Analytics

ABSTRACT

Seven principles frequently manifest themselves in data management projects. These principles improve two aspects: the development process, which enables the programmer to deliver better code faster with fewer issues; and the actual performance of programs and jobs. Using examples from Base SAS®, SAS® Macro Language, DataFlux®, and SQL, this paper will present seven principles: environments; job control data; test data; improvement; data locality; minimal passes; and indexing.

Readers with an intermediate knowledge of SAS DataFlux Management Platform and/or Base SAS and SAS Macro Language as well as SQL will understand these principles and find ways to use them in their data management projects.

INTRODUCTION

During my current long-term consulting engagement for a financial services institution, which I will call "FINCO", I repeatedly encounter situations where one or more Data Management principles manifest. Many are these are essential in enterprise data management. The scale of work for Enterprise Data Management typically involves daily updates to tens or hundreds of tables, where we see daily data volumes in the range of 100,000 records all the way up to 100 million records.

In my experience, I find that these principles span all sorts of clients and companies. If a company is not yet operating at the "enterprise" scale, its management probably hopes to reach that scale sometime in the not so distant future. Utilizing several of these principles will help the company better prepare for that future.

Some key motivations for these principles are improvements in quality, delivery times, and/or performance of the data management projects. These principles are not an exclusive list, but represent some of the more common data management principles.

Many of these principles are present in some aspect of SAS Data Integration Studio and/or DataFlux Data Management Platform. Although these platforms are typically used by companies that need Enterprise Data Management solutions, this paper will present illustrative examples that use Base SAS®, including SAS® Macro and PROC SQL, and a few examples using DataFlux Data Management Studio.

OVERVIEW OF DATAFLUX DATA QUALITY (DQ) PROCESS AND DQMART

At "FINCO", I and several other team members design, develop and maintain the technology portions of a Data Quality process using DataFlux Data Management Platform (DMP). Another team, the Data Management staff, work with the Data Stewards in the Lines of Business to identify Key Business Elements (KBE), profile those fields, and develop business rules to ensure data quality in those fields.

To profile a KBE field, the Data Steward will sample the data for a few business DATE_ID values and determine several key aspects: missing counts (aka how many NULL values), data type (character, numeric, date, etc.), and frequency counts. Most SAS programmers will notice that this mirrors the mantra "Know Thy Data". Let me repeat, "Know Thy Data".

After profiling the KBE fields, the Data Steward then writes English version of the Business Rules. For one KBE, the business rule might be as simple as "Field X cannot be NULL or empty". For another KBE, the rule might be this: "Field Y must be one of these values: Past Due, Current, Paid Off". The Data Management staff then develop the DataFlux version of each business rule using the DataFlux Expression Engine Language.

Our DataFlux process jobs run a collection of business rules, grouped into a DataFlux data job, on the specified table and business DATE_ID. DataFlux moves data differently than SAS: The data input node

reads 1 record at a time and passes it to the next node in the job. In the Monitor Node, each of the appropriate business rules run on their respective fields in the record. If a business rule "fails", then the Monitor data job writes certain portions of the record, usually including the business rule field and any pass-through fields, such as unique identifier, grouping fields, and/or balance fields, to the DataFlux repository as a "trigger" record. After the monitor phase completes, the Report phase extracts the "trigger" records, and generates 2 result tables: one with each failed record (DETAILS) and one which aggregates the records (SUMMARY). The summary table adds up the balance fields, grouped by rule field and the grouping fields.

On the other hand, each SAS step (DATA or PROC) reads *all* input records, processes all of them, and then writes all output records.

"FINCO" is definitely at the Enterprise Data Management level, with over 50 source tables updated daily. Many have data volumes less than 10000 records while some have well over 1 million records processed every day. My fellow DQ technology team members and I have developed a small number of DataFlux process job streams which enable us to quickly add a new source table without changing any code in the process jobs.

Throughout this paper, I advocate a process of examine, hypothesize, test, and act.

- Examine your code for potential improvements.
- Hypothesize an improvement.
- Try out the improvement.
- If it yields the desired results, implement it. If not, then restart at Examine.

ENVIRONMENTS PRINCIPLE:

USE SEPARATE ENVIRONMENTS IN SEQUENCE TO DEVELOP AND TEST CODE, AND THEN TO GENERATE PRODUCTION RESULTS.

BENEFIT:

*FIND AND RESOLVE PROBLEMS *BEFORE* WE PUT CHANGES INTO PRODUCTION USE.*

Like many larger companies, "FINCO" has three shared environments: DEV, TEST, and PROD. Each environment accesses different databases for the source and results data. The source data usually points to the ETL environment at the same level. Thus, as the ETL developers add a new table at their DEV database, we can point our DataFlux jobs at DEV to read that new table.

DataFlux developers use local environments on their workstations to develop process and data jobs independent of each other. They then 'check-in' or commit these changes to a versioning control system, e.g. Subversion. At each shared environment in succession, they coordinate changes in the environment using "check-out" or "update".

Other version control systems include RCS, CVS, Git, and Visual SourceSafe. Two main benefits derive from use of version control systems. First, we can quickly revert to a previous version that is known to work if we ever commit changes that have an unexpected issue not caught during development or testing. Second, we can compare two versions to see what actual changes are present in the two versions, along with any explanatory notes included with those changes. A very frequent example is the comparison of a new set of changes to the current shared version. If the developer only sees the expected changes, then she has a higher confidence that her new changes will not break any existing functionality.

We leverage the macro variable facility in DataFlux to set some common values on a per-environment basis. This allows us to easily promote a job from one environment to the next, knowing that it will configure itself to use the corresponding source and target data locations. In DataFlux, we use 3 common values:

- Environment
    - %% ENV_IS%%

- Source DataSourceName (DSN)
    - %%SRC_DSN%%
- Destination DataSourceName (DSN)
    - %%DEST_DSN%%

Here is an example of the ENV_macros.cfg file for DataFlux:

```
#
#  Sample DataFlux ENV_ environment macro variables
#
#  Prefix, ENV_ , identifies where macro variable resides

#  On developer PC, environment is set to userid
#
ENV_IS=bjanka
#
#  On Linux shared server, set to one of these:
#  ENV_IS=DEV
#  ENV_IS=TEST
#  ENV_IS=PROD

#  OS specific path separator
#
#  These are the Windows flavors
ENV_FS_SEP=\
#
#  These are the Linux flavors:
#  ENV_FS_SEP=/

#  Environment specific database DSNs
#
#  DSNs include SOURCE and DQMART
#
ENV_DSN_SOURCE=DF_SOURCE
ENV_DSNTYPE_SOURCE=ODBC
#
ENV_DSN_DQMART=DQMART_DEV
ENV_DSNTYPE_DQMART=TKTS
```

In Base SAS, we can emulate these variables by using the SAS Macro facility. For example, I wrote a simple macro called %LOAD_ENV() that takes an optional Level_Env parameter. If no parameter is specified, then it defaults to “DEV”.

The readers can try this out for themselves by specifying a different “autoexec.sas” script when they invoke SAS. For example, you might have an “autoexec_dev.sas” for the DEV environment and an “autoexec_tst.sas” for the TEST environment. In the “autoexec_ENV.sas” file, call the %LOAD_ENV() macro with the specified level_env value (“DEV” in “autoexec_dev.sas” and “TST” in “autoexec_tst.sas”).

```
/*
 *  SAS Macro:  load_env()
 *
 *  Purpose:    Demonstrate how to load macro variables for
 *              DEV, TEST, PROD environments.
 *
 *  Author:          Bob Janka
 *  Date:            2016-03-16
```

```
 */

%macro  load_env(env_level=DEV);
  /*  %let var_path = "C:\SAS\&env_level\env_vars.sas";  */

  %global  SRC_DSN  DEST_DSN  META_DSN;

  %IF &env_level = DEV %then %DO;
     %let  SRC_DSN = DEV_SRC;
     %let DEST_DSN = DEV_DEST;

     libname   &SRC_DSN.  base "C:\SAS\&env_level\data_src";
     libname  &DEST_DSN.  base "C:\SAS\&env_level\data_dest";
  %END;

  %ELSE %IF &env_level = TST %then %DO;
    %let  SRC_DSN = TST_SRC;
    %let DEST_DSN = TST_DEST;

    libname   &SRC_DSN.  base "C:\SAS\&env_level\data_src";
    libname  &DEST_DSN.  base "C:\SAS\&env_level\data_dest";
  %END;

  %ELSE %DO;
    %put load_env:  Unknown environment level: "&env_level";
  %END;

%MEND;
options mprint;

%load_env(env_level = DEV);

*  %load_env(env_level = TST);
*  %load_env(env_level = BAR);
*  %load_env();
```

```
50   %load_env(env_level = DEV);

MPRINT(LOAD_ENV):   libname DEV_SRC base "C:\SAS\DEV\data_src";
NOTE: Libref DEV_SRC was successfully assigned as follows:
      Engine:        BASE
      Physical Name: C:\SAS\DEV\data_src

MPRINT(LOAD_ENV):   libname DEV_DEST base "C:\SAS\DEV\data_dest";
NOTE: Libref DEV_DEST was successfully assigned as follows:
      Engine:        BASE
      Physical Name: C:\SAS\DEV\data_dest
```

Output 1. Output from SAS Macro, %LOAD_ENV()

Base SAS provides a system option, SYSPARM, that allows us to specify an environment value for a set of programs. If we need to specify multiple different values in the SYSPARM option, we could introduce separator characters, such as a colon (:), and then write a macro to parse out the different strings and assign those to known global macro variables for use by later macros and modules.

The specifics are less important than the concept: use macro variables to set per-environment values that allow you to point your job or program to different input data and save the results in separate locations.

JOB CONTROL PRINCIPLE:

USE INTERNAL CONTROL TABLE IN DATABASE TO CONTROL JOB RUNS.

BENEFIT:

RE-USE CODE FOR MULTIPLE SETS OF MOSTLY SIMILAR INPUT DATA.

Job control tables provide the capability to run the same job or program with different inputs and track those individual runs. Using a job control table enables us to write the job once and then re-use it for each new source table. If you put any table-specific processing into the monitor data job, then for each new source table, you need only do the following:

- Create table-specific monitor data job
    - This runs the business rules for that specific table. Each table gets its own monitor data job.
- Add 1 record to control table, DF_META_TABLES
- Add k records to control table, DF_META_FIELDS,
    - Where k is the number of rule fields PLUS the number of pass-through fields.
    - Each record in this control table specifies one of these fields.

We have 2 control tables: one that manages control data for each source table (DF_META_TABLES) and another that manages control data for each KBE field in those source tables (DF_META_FIELDS). We configure our jobs to take 2 job-level parameters: a source table name and a business DATE_ID. The job looks up the source table name in our control table, DF_META_TABLES, extracts the table-specified metadata, and then starts processing that source table.

The job has 3 phases: a Count phase, a Monitor phase, and a Report phase. The Count phase submits a query to the source table that does a simple count, filtering on only those records with the specified business DATE_ID. For most source tables, we expect data on a daily basis. If the count is zero, then the job quits with an error message.

For the Monitor phase, the job queries the control table, DF_META_FIELDS, to get a list of the fields for the specified source table. The job then invokes the table-specific data job to run the business rules for that source table. Each thread builds up a query to the source table, using the list of source fields and a filter on the business DATE_ID.

In the Report phase, the job uses the table-specific information from the control table, DF_META_TABLES, to determine which source fields to use for the Balance and grouping fields. It then submits another query to the source table to aggregate the day's data by the grouping fields and summing up the Balance field. These aggregated records provide the TOTAL record counts and TOTAL balances, by grouping variable, for the day's source data. It next extracts the trigger records from the DataFlux repository. Each of these records, including the Unique_ID field, are written to the results table, DQ_DETAIL. These records provide an exact list of which source records failed which business rules and the values that caused these failures. The same trigger records are summarized by grouping variable and then combined with the TOTAL counts and balances to populate the results table, DQ_SUMMARY.

In the SAS code shown below, it first creates the control tables and populates them with some example control data. It then shows some lookups of these control data. In the second control table, DF_META_FIELDS, the SQL_FIELD_EXPRSSN usually contains a simple expression, consisting solely of the source field name. It could contain a more complex expression that combines multiple source fields into a single composite monitor field. The example shows a numeric expression that adds two source fields to yield an amount field.

SAS program shown here:

```
/* Job Control Data - Setup */

%load_env();

/*  DQMart – DF_META_TABLES            */

data &DEST_DSN..DF_META_TABLES;
  attrib  PRODUCT_GROUP  length = $10;
  attrib  SRC_TABLE      length = $20;
  attrib  SRC_CATEGORY   length = $10;
  attrib  SRC_AMOUNT     length = $20;

  INPUT  PRODUCT_GROUP $ SRC_TABLE $ SRC_CATEGORY $ SRC_AMOUNT;
  DATALINES;    /*  aka CARDS;  */
LOANS     BUSINESS_LOANS     REGION  BUSINESS_BALANCE
LOANS     PERSONAL_LOANS     BRANCH  PERSONAL_BALANCE
DEPOSITS  ACCOUNTS           CHK_SAV NET_BALANCE
;
PROC PRINT data = &DEST_DSN..DF_META_TABLES;
run;

/*  DQMart – DF_META_FIELDS            */

data &DEST_DSN..DF_META_FIELDS;
  attrib  PRODUCT_GROUP     length = $10;
  attrib  SRC_TABLE         length = $20;
  attrib  TECH_FIELD        length = $10;
  attrib  SQL_FIELD_EXPRSN  length = $200;

  INPUT  PRODUCT_GROUP $ 1-10 SRC_TABLE $ 11-30 TECH_FIELD $ 31-40
SQL_FIELD_EXPRSN & $ 41-101;

  DATALINES;    /*  aka CARDS;  */
LOANS     BUSINESS_LOANS      REGION    REGION
LOANS     BUSINESS_LOANS      BALANCE
Coalesce(Principal,0)+Coalesce(Interest,0) as PERSONAL_BALANCE
LOANS     BUSINESS_LOANS      Field1    Field1
LOANS     BUSINESS_LOANS      Field2    Field2
;
/* NOTE: 2nd and 3rd datalines above are actually all one line */

PROC PRINT data = &DEST_DSN..DF_META_FIELDS;
run;

/*  Job Control Data - Usage */

/*  Get fields to process for source table */

proc sql noprint;
  select
     SQL_FIELD_EXPRSN
   into
       :SRC_FIELDS SEPARATED by " , "
  from &DEST_DSN..DF_META_FIELDS
  where
    SRC_TABLE = 'BUSINESS_LOANS'
  order by
```

```
    TECH_FIELD
   ;
quit;

%PUT  &SRC_FIELDS;

%LET sql_text_00 = select ;
%LET sql_text_01 = from &SRC_DSN..&SRC_TBL ;
%LET sql_text_02 = where ;
%LET sql_text_03 =   date_var = '20150704' ;

%LET sql_text =
  &sql_text_00
   &SRC_CAT
  ,&SRC_AMT
  &sql_text_01
  &sql_text_02
  &sql_text_03
  ;
%put &sql_text_00;
%put &sql_text_01;
%put &sql_text_02;
%put &sql_text_03;
%put &sql_text ;

%LET sql_text_flds =
  &sql_text_00
  &SRC_FIELDS
  &sql_text_01
  &sql_text_02
  &sql_text_03
  ;
%put &sql_text_flds ;
```

```
             The SAS System          16:08 Monday, July 13, 2015   1

    PRODUCT_                  TECH_
Obs  GROUP     SRC_TABLE     FIELD  SQL_FIELD_EXPRSN
1   LOANS   BUSINESS_LOANS REGION  REGION
2   LOANS   BUSINESS_LOANS BALANCE Coalesce(Principal,0)+ Coalesce
(Interest,0) as PERSONAL_BALANCE
3   LOANS   BUSINESS_LOANS Field1  Field1
4   LOANS   BUSINESS_LOANS Field2  Field2
```

Output 2. Listing Output from SAS blocks, Job Control Data – Setup & Usage

```
92   /*
93    *  Get fields to process for source table
94    */
95
96   proc sql noprint;
97     select
98        SQL_FIELD_EXPRSN
99       into
100         :SRC_FIELDS SEPARATED by " , "
```

```
101    from &DEST_DSN..DF_META_FIELDS
102    where
103      SRC_TABLE = 'BUSINESS_LOANS'
104    order by
105      TECH_FIELD
106      ;
NOTE: The query as specified involves ordering by an item that doesn't
appear in its SELECT clause.
107  quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.06 seconds
      cpu time            0.01 seconds
108
109  %PUT  &SRC_FIELDS;
coalesce(Principal,0)+coalesce(Interest,0) as PERSONAL_BALANCE , Field1 ,
Field2 , REGION
110
111
112  %LET sql_text_00 = select ;
113  %LET sql_text_01 = from &SRC_DSN..&SRC_TBL ;
114  %LET sql_text_02 = where ;
115  %LET sql_text_03 =   date_var = '20150704' ;
116
117  %LET sql_text =
118    &sql_text_00
119     &SRC_CAT
120    ,&SRC_AMT
121    &sql_text_01
122    &sql_text_02
123    &sql_text_03
124    ;
126  %put &sql_text_00 ;
select
127  %put &sql_text_01 ;
from DEV_SRC.BUSINESS_LOANS
128  %put &sql_text_02 ;
where
129  %put &sql_text_03 ;
date_var = '20150704'
130  %put &sql_text ;
select    REGION       ,BUSINESS_BALANCE       from DEV_SRC.BUSINESS_LOANS
where   date_var = '20150704'
131
132  %LET sql_text_flds =
133    &sql_text_00
134    &SRC_FIELDS
135    &sql_text_01
136    &sql_text_02
137    &sql_text_03
138    ;
140  %put &sql_text_flds ;
select   coalesce(Principal,0)+ coalesce(Interest,0) as PERSONAL_BALANCE ,
Field1 , Field2 , REGION
  from DEV_SRC.BUSINESS_LOANS   where   date_var = '20150704'
```

Output 3. Log Output from SAS block, Job Control Data – Setup & Usage

TEST DATA PRINCIPLE:

ADD SUPPORT TO REDIRECT SOURCE AND/OR TARGET DATA FOR TESTING.

BENEFIT:

USE DIFFERENT SETS OF TEST OR SAMPLE DATA WHEN TESTING PROGRAMS.

Before we can explore performance improvements, we need to know that the DataFlux jobs and SAS programs are behaving correctly. Make them do the "right" things *before* you try to speed them up!

In the previous principle of Environments, we set some environment defaults. In this principle, we extend our jobs and programs to allow run-time changes that override those defaults.

Many jobs and programs read data from a source, process that data, and then write out results to a destination. Our DataFlux jobs at "FINCO" do this also. We add a node at the beginning of the job that reads 3 key job-level parameters: SRC_DSN and DEST_DSN (where DSN is DataSourceName, which anyone familiar with ODBC will probably recognize) and TABLE_NAME.

The job looks for override values at run-time. If no values are provided, then the job looks up the default values in the environment settings.

We also added an optional fourth job-level parameter: TABLE_PREFIX. If not provided, we detect the default value and set the job-level parameter to NULL. If it is provided, then we set the job-level parameter to its value. Our table references now look this in SAS (periods are necessary to terminate SAS macro variable references in text strings):

&SRC_DSN..&TABLE_PREFIX.&TABLE_NAME.

For "FINCO", some of the rule fields never have any "bad" data in the live source tables. How can we show that the rule logic works for these fields? By creating an alternate set of source tables with test data. When we create this test data, we want to include both "good" data and "bad" data. For many rules, the logic is fairly simple: "The rule field must NOT be NULL or blank". We can sample one record from the live source table as our "good" data. We can then copy that record and set the values for the various rule fields in that table to NULL. This should yield 1 "FAILED" and 1 "PASSED" instance for each rule on that table. If the logic uses something else, like this: "The rule field must be numeric and not zero", we can set the "bad" data to a zero (0). In any case, we want to force each rule to "pass" at least 1 record and "fail" at least 1 record. For some tables, we had several test records to cover more complex rule logic.

Where do we put the test data? At "FINCO", we do not have write access to the source database. So, we create the test tables in our DQMart schema and use a table prefix of "TD_". This allows us to separate out test data from sample data "SD_" which we used to help develop rule logic for some other source tables.

One advantage to using test or sample data is that these trial runs will complete a LOT faster with 10-100 records compared to some live source tables which yield over 100,000 records on a daily basis.

SAS program shown here:

```
/*  Test Data - Setup  */

data &DEST_DSN..TD_LOAN;
  attrib  Field_X          length =   4;
  attrib  Field_Y          length =   4;
  attrib  Expect_X         length = $ 8;
  attrib  Expect_Y         length = $ 8;

  INPUT  Field_X  Field_Y  Expect_X  Expect_Y;

  DATALINES    /*  aka CARDS;  */;
  .      0  fail  fail
  9      1  pass  pass
  ;
```

```
proc print data=&DEST_DSN..TD_LOAN;
run;

/*  Test Data - Usage  */
%put ;
%let table_prefix = ;
%put  Data = &dest_dsn..&table_prefix.&src_tbl;

%put ;
%let table_prefix = TD_;
%put  Data = &dest_dsn..&table_prefix.&src_tbl;

%put ;
%let table_prefix = SD_;
%put  Data = &dest_dsn..&table_prefix.&src_tbl;
```

```
Obs    Field_X    Field_Y    Expect_X    Expect_Y
 1        .          0         fail        fail
 2        9          1         pass        pass
```

Output 4. Listing Output from SAS block, Test Data - Setup

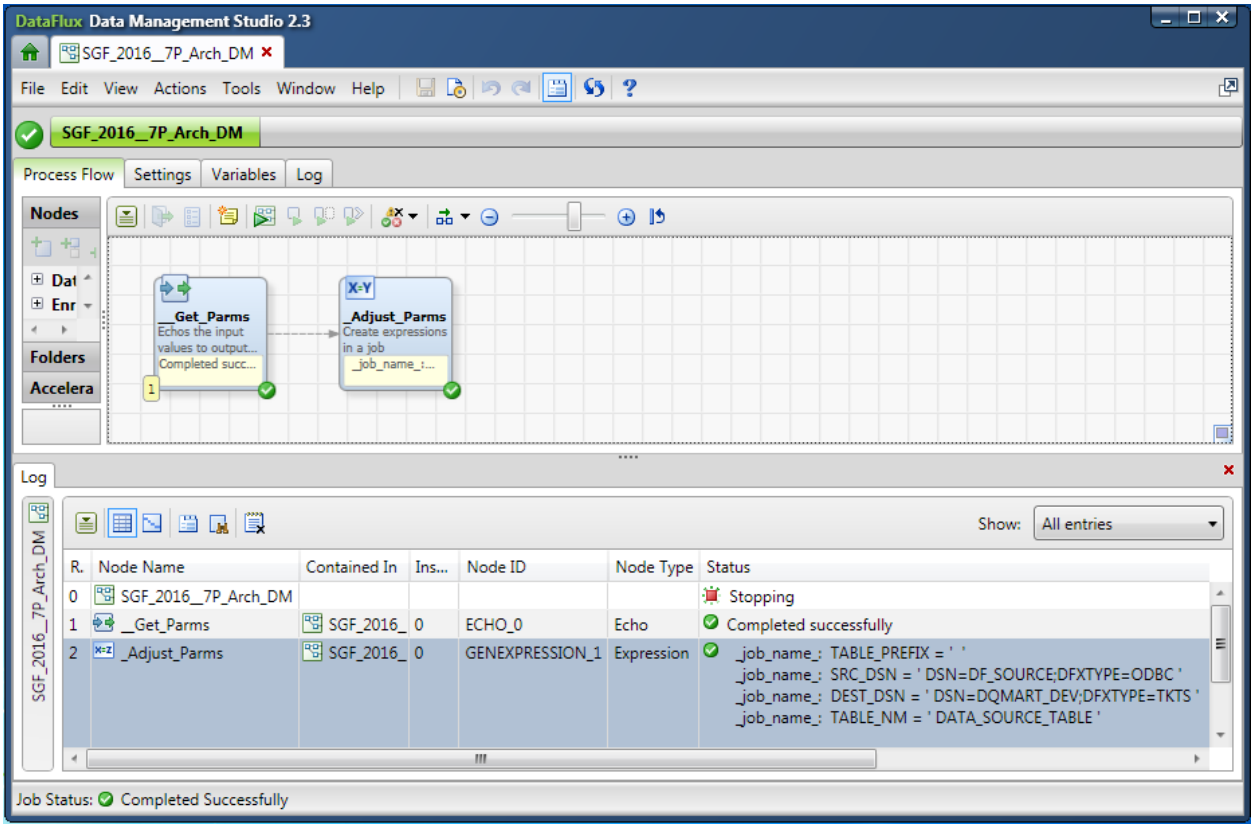
```
323  proc print data=&DEST_DSN..TD_LOAN;
324  run;

NOTE: There were 2 observations read from the data set DEV_DEST.TD_LOAN.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

333  %let table_prefix = ;
334  %put  Data = &dest_dsn..&table_prefix.&src_tbl;
Data = DEV_DEST.BUSINESS_LOANS
335
336  %let table_prefix = TD_;
337  %put  Data = &dest_dsn..&table_prefix.&src_tbl;
Data = DEV_DEST.TD_BUSINESS_LOANS
338
339  %let table_prefix = SD_;
340  %put  Data = &dest_dsn..&table_prefix.&src_tbl;
Data = DEV_DEST.SD_BUSINESS_LOANS
341  %put ;
```

Output 5. Log Output from SAS block, Test Data - Setup

DataFlux jobs use a node structure. Macro variables pass control values between nodes. Display 1 is a screenshot from DataFlux Data Management Studio for a sample job with 2 nodes. It shows the status for both nodes, including some debugging information from the print() function. The logmessage() function sends its output to the job log. Together, these functions provide debugging information across all environments.

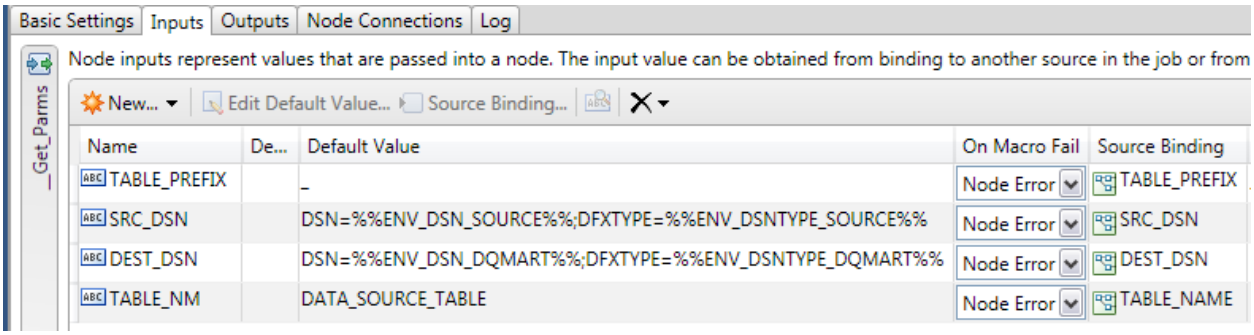


Display 1. DataFlux log for job with 2 nodes, "__Get_Parms" & "_Adjust_Parms"

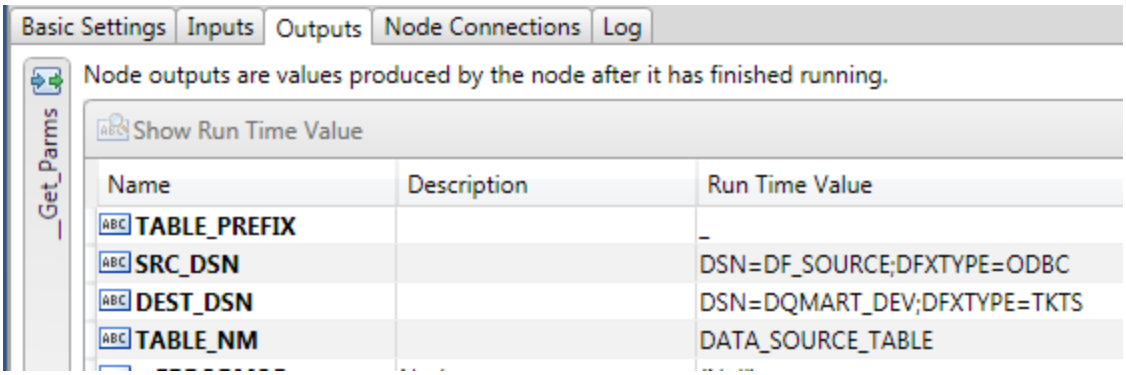
In Displays 2 & 3, we see the respective input and output macro variable values for the first node, "__Get_Parms". As the first node in the DataFlux job, we set the default-values for the job-level macro variables (aka job parameters). The tokens with the pairs of double-percent-signs, %%__%%, reference the environment macro variables setup in the ENV_macros.cfg at invocation, as described earlier in the section on the Environment Principle.

In this node, the node macro variables bind their input values to their respective job-level macro variables specified at job invocation. Here is the list for this job, in the order users and developers are most likely to override:

- TABLE_NAME
- SRC_DSN
- DEST_DSN
- TABLE_PREFIX



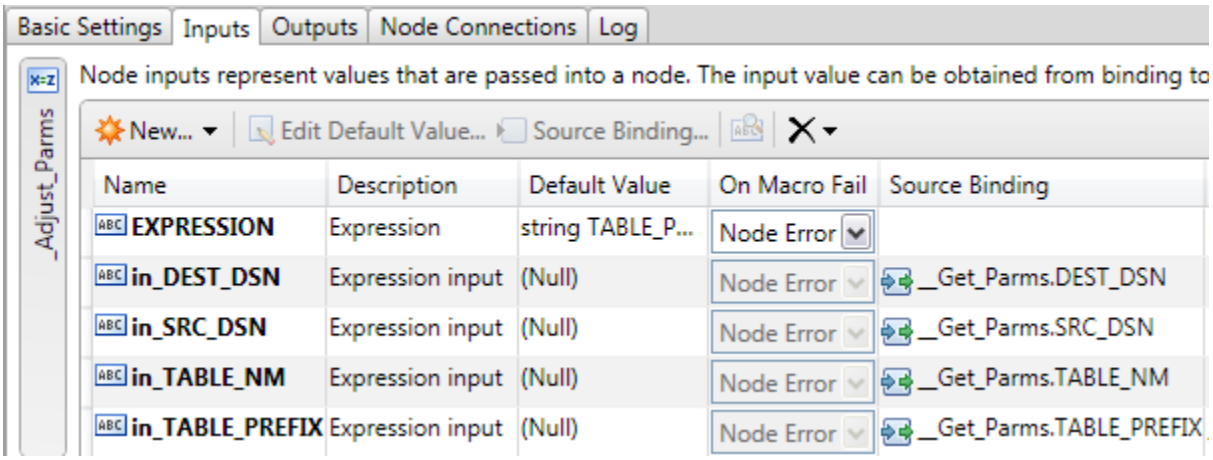
Display 2. DataFlux inputs for node, "__Get_Parms"



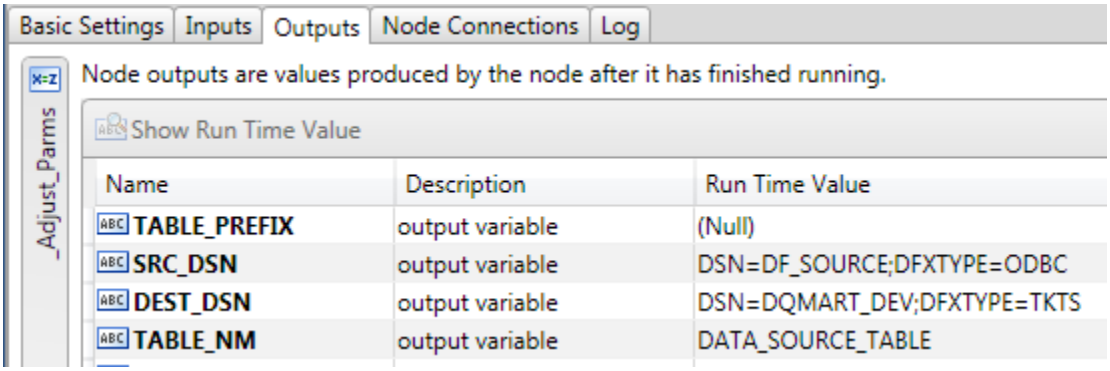
Display 3. DataFlux outputs for node, "__Get_Parms"

In the next two displays, 4 & 5, we see the respective input and output macro variable values for the second node, "_Adjust_Parms". Here, we see that the source values for the node macro variables are "bound" to the outputs from the first node, "__Get_Parms".

Any subsequent nodes would bind the input values for their node macro variables to the output from this node, "_Adjust_Parms". Thus, we can control the remainder of job by changing the job-level parameters.



Display 4. DataFlux inputs for node, "_Adjust_Parms"



Display 5. DataFlux outputs for node, "_Adjust_Parms"

In DataFlux, we use a default value of a single underscore, '_', to track whether some override values are NULL or not. When the values propagate through the Source Binding, a NULL could be either an override value OR a missing value. Here is the DataFlux Expression code for node, "_Adjust_Parms":

```
//  Expression Node:  _Adjust_Parms

string   TABLE_PREFIX
string   SRC_DSN
string   DEST_DSN
string   TABLE_NM

// * * * * * * * * * * * * * * * * * * * *

if ( in_TABLE_PREFIX == "_" )  then
  begin
    //  default value of "_"  -->  do not use Table_Prefix
    TABLE_PREFIX = NULL
  end
else
  begin
    //  non-default value -->  *do* use specified Table_Prefix
    TABLE_PREFIX = in_TABLE_PREFIX
  end

logmessage( " _job_name_:  TABLE_PREFIX = ' "  &  TABLE_PREFIX  &  " ' " )
print( "_job_name_:  TABLE_PREFIX = ' "  &  TABLE_PREFIX  &  " ' " )

//  Pass-through macro variables
//
SRC_DSN = in_SRC_DSN
logmessage( " _job_name_:  SRC_DSN = ' "  &  SRC_DSN  &  " ' " )
print( " _job_name_:  SRC_DSN = ' "  &  SRC_DSN  &  " ' " )

DEST_DSN = in_DEST_DSN
logmessage( " _job_name_:  DEST_DSN = ' "  &  DEST_DSN  &  " ' " )
print( " _job_name_:  DEST_DSN = ' "  &  DEST_DSN  &  " ' " )

TABLE_NM = in_TABLE_NM
logmessage( " _job_name_:  TABLE_NM = ' "  &  TABLE_NM  &  " ' " )
print( " _job_name_:  TABLE_NM = ' "  &  TABLE_NM  &  " ' " )
```

LOGGING QUERY STATEMENTS

An associated suggestion in this principle involves the logging of any SQL queries generated by the job or program and submitted to the source database. SAS programmers are lucky in that PROC SQL usually writes to the SAS log the query it submits to the database. If you are encountering issues with the submitted queries in SAS, you can extract the exact SQL query text from the log and submit in a database client, which eliminates any SAS overhead or issues from your troubleshooting. These external client submissions often yield more detailed error messages that help the developer pinpoint the problem in the query.

In DataFlux, we build up text blocks containing the generated SQL queries and then submit those in the job. We added a few statements to write out those blocks to the job log which enable our troubleshooting. Details of exactly how we do this in DataFlux exceed the scope of this paper.

IMPROVEMENT PRINCIPLE:

USE THE PARETO PRINCIPLE (AKA 20/80 RULE) REPEATEDLY AS NEEDED.

BENEFIT:

FOCUS IMPROVEMENT EFFORTS ON MOST SIGNFICANT FACTORS.

- Stay alert for opportunities to improve
    - Improvements can be in the code
    - Improvements can also be in the development process

The first three principles focus on the development process while the last three principles focus on the performance of programs and jobs.

In our development, we start with “good enough” code that meets the minimum quality and performance requirements. Over time, as we modify the code to add some of the inevitable changes required with new source data that has different needs, we strive to improve both the quality and performance.

We now expand our focus to a broader topic, that of continual improvement. When we have an opportunity to review the existing code base for jobs or programs, we want to take a few minutes and see if there are any improvements that we might have identified during production runs. Perhaps our jobs are taking longer amounts of time to complete.

Often times, we have pressures to deliver jobs or programs by a particular deadline. This forces us to choose what items we have to do before the deadline and which items can wait for a future change opportunity. If we keep these items in a list somewhere, then the next time we are making changes to the job or program, we can review them for inclusion.

Use the 20/80 rule: where 20% of factors explain (or yield) 80% of results. One way to view this is that we can usually find a few items to improve for now and leave others for later. Another thing to keep in mind is that we don’t have to fix *everything* *all of the time*. Often, we get “good enough” results by resolving a few critical items.

Find the one or few factors that explain a majority of the issues.

Resolve these and observe the results.

To help us focus on the key aspects of our computations, let me introduce a concept from Computer Science, namely Order Notation (aka Big-O Notation). This notation helps us compare algorithms and even full programs by assessing costs involved in getting data, processing it, and saving the results.

The full details of Order Notation are beyond the scope of this paper, but we can get a general sense of some common cost assessments. There are 4 common examples:

- O(**N**)
    - Linear time to perform operations.
- O(log**N**)
    - Logarithmic time to perform operations
- O(**N** * log**N**)
    - **N** times Logarithmic time to perform operations.
- O(**N** * **N**)
    - Quadratic time (**N** times **N**) to perform operations.

Each of these examples assess the costs of performing a particular set of operations based on the number of items going those operations. For instance, when **N** < 10, the differences are fairly small between Linear and Logarithmic. When **N** exceeds one thousand (1 000) or one million (1 000 000), then the differences are quite dramatic.

Here is a graph where **N** varies from 10 to 100 by 10, including the O(**N** * **N**) quadratic case.

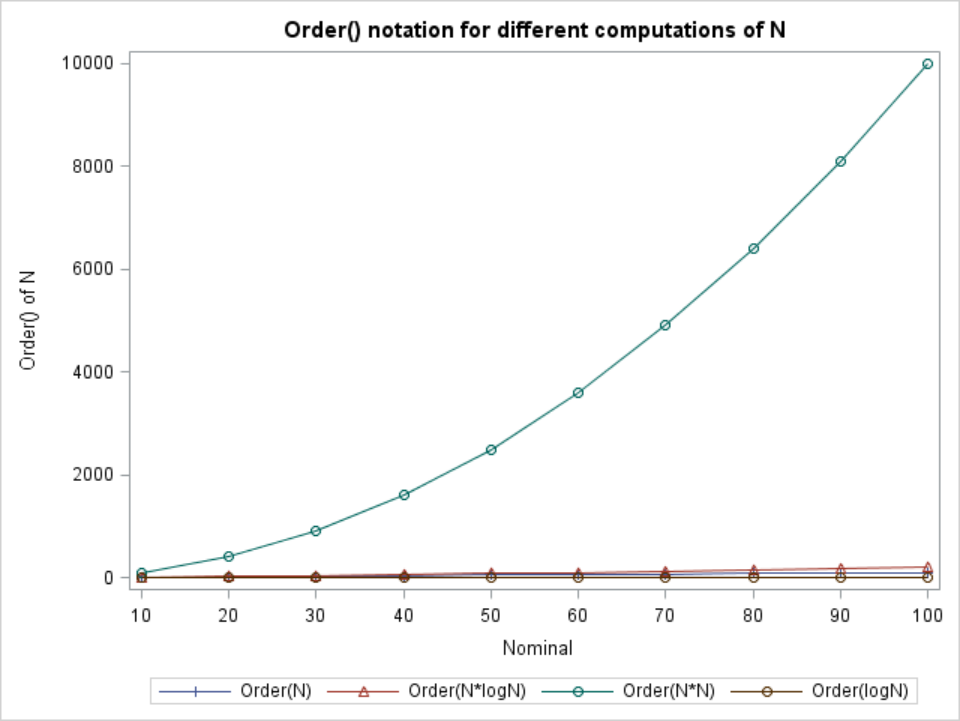


Figure 1. Order() notation for different computations of N

Since the O(**N** * **N**) increases greatly with larger values for **N**, this squeezes together the other 3 Order Notation expressions. To help clarify their relative increases, I show a second graph <u>without</u> the O(**N** * **N**) values.

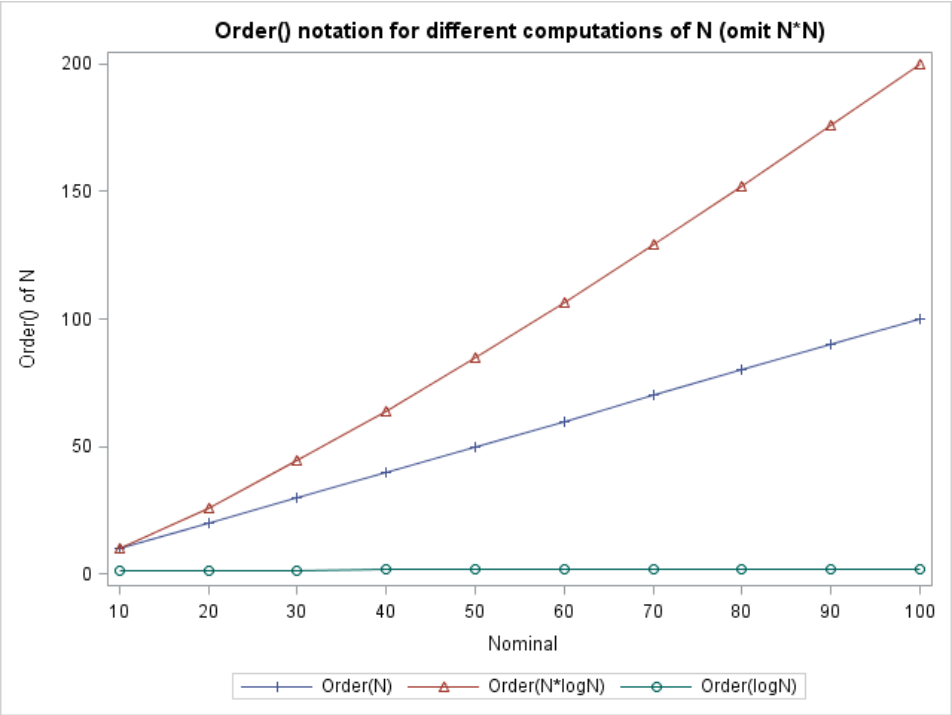


Figure 2. Order() notation for different computations of N (omit N*N)

Why do we want to use Order Notation? A key teaching example is the difference between efficient and inefficient sorting algorithms. Many efficient sorting algorithms perform in **N** * logarithmic time at O(**N** * log**N**). These include algorithms such as quicksort and mergesort. Inefficient sorting algorithms, such as insertion sort and bubble sort, tend to perform in quadratic time at O(**N** * **N**).

Binary searches or heap operations are a common example of logarithmic time at O(log**N**).

What is a key learning from Order Notation? The costs of certain classes of operations are more expensive than others. Data Management professionals should strive to reduce these classes, which are most likely sorts. Although we cannot eliminate them entirely, we have several strategies to consider:

- Combine multiple sorts into a single sort.
- Defer sorting until after data is pruned.
- Sort keys to the data instead of the data itself.

As we explore the next few Data Management principles, I will refer to the corresponding Order notations.

For SAS programmers, I strongly encourage you to set option FULLSTIMER. This gives you details on how much CPU and I/O is used for each DATA or PROC step. If you extract these details with the name of the PROC or DATA step, then you sort these details to find which steps consume the most amount of time in your program. If you want to assess the Order Notation for your program, try running multiple tests with different scales of observations, say 10, 100, 1000. You can then plot your total time to see which curve approximates the run-times.

DATA LOCALITY PRINCIPLE:

MOVE COMPUTATIONS AS CLOSE AS POSSIBLE TO DATA.

BENEFIT:

IMPROVE PERFORMANCE BY REDUCING DATA TRANSFER TIMES.

In many Enterprise Data Management environments, data moves between multiple database and compute server. This introduces overhead in moving data across a network, which is usually 10-1000 times slower than moving data internally in a server. Efficient data management dictates that we process the data as close to the source as possible to reduce the network overhead.

One example for data locality is pushing computations into the database, especially any summarizations and/or aggregations. If we look at the effort required to pull data from a source database into SAS, then run PROC summary, and then save results, we can use this Order Notation expression:

T = k * O(**N**) + Summary(N) + k * O(M)

where: k is a constant factor representing the overhead of moving data over the network.
N is the number of source records
M is the number of result records.
Summary(**N**) is the time to run PROC SUMMARY.

If we are summarizing to a much smaller number of result records, then it is more efficient to move that summarization to the database as much as possible. Here is an example using a common SQL concept of subqueries:

```
/*  Data Locality Principle  */

proc sql;
  select
     SQ.count_rows
    ,SQ.sum_actual
    ,SQ.prodtype
    ,SQ.product
    ,(SQ.sum_actual / SQ.count_rows )  as  Avg_sales
```

```
  from (
    select
      count( * )  as  count_rows
     ,sum(actual)  as  sum_actual
     ,prodtype
     ,product
   from sashelp.prdsal2
   group by
      prodtype
     ,product
   ) SQ
   ;
quit;
```

Our new Order Notation expression would now look like this:

T = Summary(**N**) + k * O(M)

Where we have eliminated the first factor of “k * O(**N**)”.

Summary(**N**) is the time to run the SQL summarization query

(assume: PROC SUMMARY and SQL summarization query take roughly the same amount of time).

In DataFlux, all SQL queries are passed directly to the source database. In Base SAS, PROC SQL has two modes: implicit pass-through and explicit pass-through. The mode is determined by how you connect to the database. If you create a libname and then use SAS libname.dataset notation in your SQL query, PROC SQL will implicitly pass the query pieces to the source database via a SQL query translator. If you use a connect statement, then PROC SQL directly connects to the source database in “explicit” mode. For many queries which use generic SQL expressions, there should be little difference. If you find that you need a database-specific function or expression, then you must use the connect() statement and “explicit” mode. Be aware that this code now locks into the SQL dialect of that source database. If you try to change databases, say from Oracle to Netezza or SQL Server to Teradata, then you will have to review these queries and possibly rewrite them to work with the new dialect.

```
                            Product

count_rows  sum_actual  Type        Product     Avg_sales
      5760     3710252  FURNITURE   BED          644.1409
      5760     3884188  FURNITURE   SOFA         674.3382
      5760     3695384  OFFICE      CHAIR        641.5597
      5760     3718359  OFFICE      DESK         645.5484
```

Output 6. Listing Output from SAS block, Data Locality Principle

```
230  proc sql;
231    select
232       SQ.count_rows
233      ,SQ.sum_actual
234      ,SQ.prodtype
235      ,SQ.product
236      ,(SQ.sum_actual / SQ.count_rows )  as  Avg_sales
237    from (
238      select
```

```
239         count( * )  as  count_rows
240        ,sum(actual)  as  sum_actual
241        ,prodtype
242        ,product
243      from sashelp.prdsal2
244      group by
245         prodtype
246        ,product
247      ) SQ
248      ;
249  quit;

NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Output 7. Log Output from SAS block, Data Locality Principle

MINIMAL PASSES PRINCIPLE:

TOUCH EACH OBSERVATION OR RECORD AS FEW TIMES AS POSSIBLE.

BENEFIT:

IMPROVE PERFORMANCE BY REDUCING MULTIPLE READS OF DATA.

The principle of Minimal Passes is similar to Data Locality, but deals more with avoiding extraneous computations. If we can eliminate or combine expensive steps in our DataFlux jobs or SAS programs, then the jobs and programs will run faster and handle more data in a particular window of time.

A very common example of this deals with sorting data as we process it. If we examine a job or program and find several sort steps, then these are prime candidates for combining into a single step.

Another example is using multiple data steps in SAS programs to transform data and/or perform calculations. If we have several data steps that each do 1 or 2 operations on the same dataset, then we can and should combine those into a single data step that does all of the operations in sequence on the data.

One example we encountered during our DataFlux development involved the inclusion of TOTAL values in the summary data. The business team asked that we report the TOTAL count, TOTAL sum and several variations of these, including split TOTALS of negative, positive, and NULL values. Our original DataFlux job, inherited from the original developer, performed 3 separate database operations:

- INSERT the summary of the FAILED records into the results table
- UPDATE these summary records with the pre-computed TOTAL values.
- UPDATE the “passed” counts and sums for these summary records by subtracting FAILED from TOTAL (PASSED = TOTAL – FAILED).

When we were adding a new set of source tables to the control data, we had to make some changes to the DF process job to accommodate some new features. At this time, we examined the triple steps of INSERT, UPDATE, UPDATE and re-wrote this to pre-compute the TOTALS up front, then combine those into the FAILED summaries and compute the PASSED values *before* we ran the database INSERT step. This reduced our processing significantly as shown by the following BEFORE and AFTER Order Notations:

- Where
    - **N** = number of records in daily set
    - M = number of records in entire results

- BEFORE
    - INSERT + GenTOTALS + UPDATE + UPDATE
    - **(N *** O(logM) **)** + (**N** * O(logM)) + (**N** * O(logM))
- AFTER
    - INSERT + GenTOTALS
    - **N *** O(logM)

INDEXING PRINCIPLE:

LEVERAGE INDEXES WHERE APPROPRIATE TO IMPROVE RUN-TIME PERFORMANCE.

BENEFIT:

IMPROVE PERFORMANCE BY REDUCING RECORD LOOKUP TIMES.

Indexes on data are a very powerful tool if used wisely. This principle will help identify some places to use indexes.

Every modern database, including SAS datasets, supports indexes. I won't go into the specific details on which index type to use where. Rather, I want to share some examples where it is definitely useful.

A key assumption about many data management environments is the concept of write few times, read many times. In other words, most accesses to the data are READ queries, especially when compared to WRITEs. Other environments might not benefit as much from indexes.

Indexes are very useful in these cases as they can dramatically improve run-times for the READ queries at some smaller cost for updating the indexes during WRITE queries.

The important thing is to carefully pick your index fields. In a previous example, we discussed replacement of a triple step of INSERT / UPDATE / UPDATE with a single step of INSERT. We might improve performance even more if we identify the field much often used in the WHERE clauses of the READ queries and create indexes on that field.

A "primary key" field by definition will have an index created. In our DataFlux DQMart, we found that our 2 results tables, DQ_SUMMARY and DQ_DETAIL, did not have a single field or even a combination of fields that yielded a unique reference to each result row.

Instead, we noticed that RUN_ID field present in each table, while not unique, did group together a collection of result records. We worked with our database team DBA to create an index on the RUN_ID field for each table. The improvement in our process job run-time was noticeable. A welcome, but unexpected, side effect was that the data pull which occurs after we write out our DQ results *also* improved. Two other uses improved their run-times: troubleshooting queries which we submit when tracking down reported issues in the data and the periodic data purges when we delete result records past their retention dates.

Another instance where the Indexing Principle can help is when submitting queries to our source data tables. Often, these queries include a filter on the business DATE_ID. By adding an index on this field, we find that the our queries run much faster.

The Order Notations for both of the above examples shows the difference between non-indexed and indexed filters:

Non-indexed:

T = M * O(**N**)

where:

N = total number of records in table
M = number of records in subset specified by filter (usually M << **N**).

Indexed:

$$T = M * O(\log \mathbf{N})$$

where:

N = total number of records in table
M = number of records in subset specified by filter (usually M << **N**).

CONCLUSION

This paper describes 7 principles that help improve development of Data Management programs and jobs. The first three principles, Environments, Job Control, and Test Data, benefit the quality of the code. The last three principles, Data Locality, Minimal Passes, and Indexing positively impact the performance of those programs and jobs. The middle principle, Improvement, binds all of these efforts together.

Use one or more of these principles in your development project to improve your quality and performance!

ACKNOWLEDGMENTS

The author thanks his many colleagues, former and current, who have graciously shared their knowledge, experience, and friendship. The author is especially grateful to the members of San Diego SAS Users Group who provided valuable feedback and support.

RECOMMENDED READING

- *PROC SQL: Beyond the Basics using SAS ®, Second Edition.*
- *The Little SAS® Book: A Primer, Fifth Edition.*
- *Carpenter's Complete Guide to the SAS Macro Language, Second Edition.*

CONTACT INFORMATION

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

Bob Janka, MBA

Modern Analytics
6310 Greenwich Dr, Ste 240
San Diego, CA 92122

Bob.Janka@computer.org

www.modernanalytics.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.