SESUG Paper 261-2018

Tips for Pulling Data from Oracle® Using PROC SQL® Pass-Through

John J. Cohen, Advanced Data Concepts, LLC, Newark, DE

ABSTRACT

For many of us a substantial portion of our data reside outside of SAS®. Often these are in DBMS (Data Base Management Systems) such as Oracle, DB2, or MYSQL. The great news is that the data will be available to us in an already-structured format, with likely a minimum of reformatting optimized database effort required. Secondly, these DBMS' come with an array of manipulation tools of which we can take advantage. The not so good news is that the syntax required for pulling these data may be somewhat unfamiliar to us.

We will offer several tips for making this process smoother for you, including how to leverage a number of the DBMS tools. We will take advantage of the robust DBMS engines to do a lot of the preliminary work for us, thereby reducing memory/work space/sort space, data storage, and CPU cycles required of the SAS server – which is usually optimized for analytical work while being relatively weaker (than the DBMS) at the heavy lifting required in an increasingly Big Data environment for initial data selection and manipulation. Finally, we will make our SAS Administrators happy by reducing some of the load in that environment.

INTRODUCTION

Databases (or more precisely, DMBS – database management systems) are software applications used to store, manage, and retrieve data. These will be designed to optimize some number of tasks, some of which may work at cross-purposes and therefore require decisions regarding priorities and appropriate tradeoffs relative to current needs. These tasks include data storage (at acceptable costs), data retrieval (at acceptable/efficient speeds), data integrity (including backups and back outs, redundancy, and the like), assuring appropriate security and privacy, ability to support real-time processing (say, credit card transactions or web click streams), multiple file and data types (character versus numeric, structured data in rows and columns versus unstructured data such as Word documents, .jpg files, html pages, xml files, etc.). And as the size of our data grows exponentially, our ability to pull our data from databases becomes a critical component of getting our work done.

As SAS users we care about data storage and particularly data retrieval. We will introduce certain concepts which will allow us to leverage a number of the constructs resident within DBMS' to facilitate this process. We will also discuss certain approaches to structuring our SAS programs in order to optimize performance. While the examples here will be specific to SAS and Oracle, the concepts and much of the programming syntax will be easily transferable to many other DBMS environments.

WHAT IS THE SQL PASS-THROUGH FACILITY?

The SQL pass-through facility in SAS allows us to connect directly to our DBMS and send SQL (Structured Query Language) statements directly to the DBMS. The SAS/ACCESS engine for the DBMS in question must be installed. This means that SQL (native to the DBMS) or SAS PROC SQL statements can be executed within the DBMS. (In theory, there will usually not be much of a difference.)

Because our data are coming from a structured database, we will inherit much of the variable definition information (column_name becomes variable name, the data characteristics such as character or numeric, data lengths, etc.). You will need to experiment with how formatting passes across to SAS (for currency or date fields and the like) and make the appropriate adjustments. Regardless, you will be well ahead of reading in raw data.

A simple example might look much like the following in Figure 1:

Figure 1 – SQL Pass-Through – Simple Example

```
PROC SQL;
   connect to oracle (user=myusername password=mypassword);
   create table MD_customers as
   select *
      from connection to oracle
        (select * from customers
         where state = 'MD');
   disconnect from oracle;
quit;
```

Those already familiar with SAS PROC SQL will find the above example to be generally familiar. The three highlighted program lines allow us to read from the external DBMS almost as if it were another SAS dataset, with the "connect to oracle" serving as if it were a LIBNAME statement, the "from connection to oracle" uses that LIBNAME statement, and then we close our connection with "disconnect from oracle". (Leaving a connection open can wreak havoc and result in your name appearing on a multitude of "naughty lists". So just don't do it.)

We are reading from the Oracle table **customers**, selecting all columns (the asterisk) where the value of the column **state** is "MD". Our result will be a SAS dataset with all the variables from the Oracle table where the customer's state is Maryland. We can now continue with our analysis in SAS, but our initial data pull was executed in Oracle.

ORACLE 001

Because our predominant interest is in SAS, we will only briefly discuss some of the constructs in Oracle of which are of interest – a fuller exposition on Database Design otherwise might warrant a course designation of "101". Note also that in a complex, sophisticated DBMS, many of the architectural decisions regarding implementation of individual tables is very much NOT a "one size fits all" approach. For each new Oracle table you want to read in, you will need important information regarding how the table is set up. And you will need to benchmark performance on a regular basis to identify optimal – or at least efficient – program design in the SAS environment. Look to examples from your colleagues, documentation as available specific to your installation, and expect to need to develop a good working relationship with the DBAs and Data Governance teams.

We will explore the following concepts in turn:

- Indexes
- Partitions
- Parallel processing
- *Normalized* data
- Views

ACCESSING ORACLE INDEXES

Much as in SAS, indexes are created in Oracle to reduce the cost of searching for the desired rows (or observations). Individual columns (variables) are identified as key to a particular table (dataset) and the physical location of the particular row is noted in a small "look up" table (the index table). When all of the rows with a particular indexed value (let's say, where State = "MD") are desired, the initial search is made of the index table, a very small table in comparison to the large data table. In turn the index table rows selected will point to the exact physical location of the rows of interest in the much larger data table, and then only those rows are pulled in entirety from the large data table.

Depending on the structure of the data table, this may speed up our data pull somewhat, substantially, or sometimes even not at all. Testing our results under a variety of conditions (with multiple concurrent users or none, various sizes of the data tables, varying proportions of the data being selected, etc.) will be needed to get a feel for the benefits of accessing the DBMS index. Because the index has been created and is resident on the DBMS and because this initial data pull is executed on the DBMS, the load on SAS is minimal. In Figure 2 we see an example. We will access the index (on the column State) using a ***hint***.

Figure 2 – Accessing the Index

```
PROC SQL;
   connect to oracle (user=myusername password=mypassword);
   create table MD_customers as

   select *
      from connection to oracle
        (select /*+ index(c) */
                c.customer_name, c.state, c.amount_spent
            from customers c
            where state = 'MD' and year = '2016');
   disconnect from oracle;
```

The ***hint*** to Oracle is indicated with the "slash-asterisk-plus" and "asterisk-slash" surrounding the text, in this case suggesting the use of the index in table **C**. If our hint syntax is incorrect, Oracle will simply treat this as a comment (no harm, no foul, but also no error report). But if our syntax IS correct, then our data retrieval may prove to be substantially faster. Note that we point explicitly to the table in question with a symbolic "c" in the index hint as well as in our list of columns (**c**.state, etc.), and define the symbolic in our "from" statement (from customers **c**). This will be of particular relevance when we have more than one table in our query.

In this instance we knew that there was an index on the column **state**, so we listed that column in our "where" clause first. There might also be an index on the column **year** -- or not. For optimal data retrieval, columns being "queried" with indexes should be list first, followed by columns without indexes. We can identify which columns have indexes with a query much like that in Figure 3.

Figure 3 – Finding Columns with Indexes

```
PROC SQL;
   connect to oracle (user=myusername password=mypassword);
   create table index_list as
   select *
      from connection to oracle
         (Select table_name, index_name, column_name, column_position
             from ALL_IND_COLUMNS
             where table_name like 'CREDIT_CARD%'
             order by table_name, index_name, column_position);
   disconnect from oracle;
quit;
```

The partial results will look much like Figure 4.

Figure 4 – Sample of Index List

TABLE_NAME	INDEX_NAME	COLUMN_NAME	COLUMN_POSITION
CREDIT_CARD_DEMOGRAPHY	CCD_AUTH_USER_FLAG_IX3	AUTH_USER_FLAG	1
CREDIT_CARD_DEMOGRAPHY	CCD_CUST_ACCNT_ID	CUSTOMER_ACCOUNT_ID	1
CREDIT_CARD_DEMOGRAPHY	CCD_MEMBER_ID	MEMBER_ID	1
CREDIT_CARD_DEMOGRAPHY	CCD_OPEN_DT_IX1	OPEN_DT	1

Here we see any of four columns which individually or in combination can be used to speed up our data pull, depending on our needs. Note that while it may be convenient to exclude (or include) all NULL values for a particular column, Oracle does NOT store null values in indexes, so a query such as:

where AUTH_USER_FLAG is not NULL

will NOT use the index on that column. Instead, convert the value (in Oracle!) first:

where NVL(AUTH_USER_FLAG, 'N/A') is not = 'N/A'

ACCESSING ORACLE PARTITIONS

Partitions, much like indexes, are created in order to speed up data pulls. However, here the data are stored physically in contiguous locations based on the column(s) being used. If the data architects identify a Date column (say, for daily transaction files) or State as being critical criteria for searching this particular table, then accessing via partition will limit the search to only the physical location where that portion of the data are stored. In other words, if once again I only need data for Maryland, my query will only reference that partition.

Figure 5 – Accessing Partitions

```
PROC SQL;
   connect to oracle (user=myusername password=mypassword);
   create table new_customers as
   select *
      from connection to oracle
         (Select *
             from customers partition(PTN_MD)
             where open_date >= '01-JAN-2016');
   disconnect from oracle;
quit;
```

By limiting our query to searching just a single partition, the efficiency of our data pull should be greatly improved. We can identify which tables are partitioned as suggested in Figure 6.

Figure 6 – Identifying Partitions

```
PROC SQL;
   connect to oracle (user=myusername password=mypassword);
   create table partition_list as
   select *
      from connection to oracle
         (select TABLE_NAME, PARTITION_NAME
             from ALL_TAB_PARTITIONS
             where TABLE_NAME like 'DAILY%'
             order by TABLE_NAME, PARTITION_NAME);
   disconnect from oracle;
quit;
```

You will need to confirm that these are the correct table and column names for you query, but most Oracle installations will use much of the standard naming conventions and structures “as is”, and only customizing their proprietary data. So a little poking around should be sufficient to getting you on your way. In Figure 7 we will see the typical results.

Figure 7 – Sample Partition List

TABLE_NAME	PARTITION_NAME
DAILY_TRANSACTIONS	PTN_D20160623
DAILY_TRANSACTIONS	PTN_D20160624
DAILY_TRANSACTIONS	PTN_D20160625
DAILY_TRANSACTIONS	PTN_D20160626
DAILY_TRANSACTIONS	PTN_D20160627
DAILY_TRANSACTIONS	PTN_D20160628
DAILY_TRANSACTIONS	PTN_D20160629
DAILY_TRANSACTIONS	PTN_D20160630
DAILY_TRANSACTIONS	PTN_D20160701
DAILY_TRANSACTIONS	PTN_D20160702
DAILY_TRANSACTIONS	PTN_D20160703
DAILY_TRANSACTIONS	PTN_D20160704
DAILY_TRANSACTIONS	PTN_D20160705

Here a query looking for transaction data on June 28th might include the following:

from DAILY_TRANSACTIONS(PTN_D20160628)

PARALLEL PROCESSING

Including a Parallel ***hint*** will allow us potentially to leverage the multi-processor environment common in large database installations. We again use the hint syntax and must specify which table is to be queried in parallel fashion and the number of processors being requested. (Each installation will likely have a suggested number, and access is dependent on the number of other users also active at a given time.) In Figure 8 we will request parallel processing for our query which will return a count of the number of active customers in our table.

Figure 8 – Parallel Processing

```
PROC SQL;
   connect to oracle (user=myusername password=mypassword);
      select *
      from connection to oracle
        (select /*+ PARALLEL (c,8) */
                Count(*)
            from customers c
            where active = 'YES');
   disconnect from oracle;
```

For table customers labelled “c” we are requesting 8 processors to execute our query. Note that in this instance we are NOT creating a table/SAS dataset and instead only expect to see a number (the count) returned to us.

NORMALIZED DATA

Oracle, among other attributes, is a relational database system. Data architects will analyze each new collection of data for ingestion as a table – as well as reassess existing tables – and determine appropriate ways to store these data. A key consideration in this exercise is to determine which potential columns are “stand alone” or shared. Thus, for efficiency of storage and retrieval a goal will be to retain data in a way that minimizes “superfluous” data. The effort will be to design a collection of tables which contain all of the incoming data, but in a simplified (*normalized*) fashion.

In a simple illustration, we are tracking customer purchases by merchant. In a simple form, we have customer ID, name, address, city, state, zip, date of purchase, purchase amount, merchant id, and merchant name. At a reporting and analytics stage we likely need all of these elements displayed for each transaction. But the efficiency of storing all of these in a single table is sub-optimal. More likely we would consider breaking these data into three separate, simpler tables and joining the selected rows at reporting time. In Figure 9 we see what the data might resemble at reporting time.

Figure 9 – De-Normalized Table

customer_id	name	address	city	state	zip	date	merchant_id	amount	merchant_name
00123	John Doe	1313 Mockingbird Ln	Perth Amboy	NJ	08534	1-Jun	776	14.95	ebay

In contrast, within the database we might expect something more like the following in Figures 10A, 10B, and 10C.

Figure 10A – Customer_Demography Table

customer_id	name	address	city	state	zip
00123	John Doe	1313 Mockingbird Ln	Perth Amboy	NJ	08534
14005	Jane Roe	17 Cleveland Ct	xx	xx	xx
17623	Herman Eutics	xx	xx	xx	xx
09786	xx	xx	xx	xx	xx
23456	xx	xx	xx	xx	xx

Figure 10B – Purchases Table

customer_id	date	merchant_id	amount
00123	1-Jun	776	14.95
14005	2-Jun	145	127.5
23456	2-Jun	643	97.66
00123	3-Jun	776	1,245.97
00123	4-Jun	776	76.5
17623	7-Jun	776	14.95

Figure 10C – Merchant Table

merchant_id	merchant_name
145	amazon.com
643	Superfresh
776	ebay

These same data are stored as three separate tables, each complete in as of itself. As needed, the rows/observations of interest can be retrieved and joined to create the appropriate view of the data at reporting time. Although this example is likely much simpler than would be encountered in an actual work setting, we can further illustrate the challenge. Figure 11 suggests the nature of the analysis.

Figure 11 – Analysis of Data

metric	customers	purchases	merchants
observations	100	10,000	500
variables	6	4	2
Space required (cells)	**600**	**40,000**	**1,000**

In this hypothetical database we have 100 customers, each row or observation consisting of 6 columns or variables. In this example we will assume that all columns are equal, so a simple calculation suggests that we will need 600 "cells" to store (and retrieve from) the customer table. Similarly the purchases (10,000 transactions so far year-to-date) by 4 columns suggests 40,000 cells. Finally, the merchant table will take 500 observations by 2 columns or 1,000 cells. The total storage requirement will be the sum of the individual tables, or **41,600** (600 + 40,000 + 1,000).

In contrast, the de-normalized table in Figure 9 contains only 10 columns (no need to store customer_id nor merchant_id in two locations each). But with 40,000 rows, this means we will need **400,000** cells to store this version of our data. (As my dog might say to me, "41,600 versus 400,000 – you do the math.")

DISCUSSION

We would expect much of the data stored in the DBMS to be of this nature – perhaps not fully-normalized, but certainly stored across multiple tables. As a result, many of our analyses will require that we join tables in advance of any subsequent work in SAS. Executing as much of this heavy lifting in the DBMS is one goal. The second, nevertheless, is to make our SQL queries as efficient as possible. This means that in addition to using the ***hints***, we also need to ***engineer*** our queries. Certain powerful tools exist to assist us (such as ***explain plans***), but these are beyond the scope of this paper. We will instead suggest certain rules in determining the order and content of your

- Always list the most selective query, condition, index, or partition first, then the second most, etc.
    - Note that if you tend to copy-and-paste some of the same programs with any frequency, the desired result may vary considerably over each new instance. (We might want only Maryland customers in one run, but only customers in July in another – implying a different "first" selection criteria for each run.)
- When multiple tables are to be joined, there can be a cost to joining all in one run.
    - Instead join the first two, then that result with the third, etc.
- Tailor your results to emerge from Oracle in as final/polished form as possible.
    - Formatting/re-coding variables, although this can be tricky.
    - Sort order.
    - Benchmark, validate, test.
        - Never assume that a random sample of, say, one run will necessarily easily generalize to all possible circumstances under which you might run your program.

In Figure 12 we suggest one approach to creating the de-normalized version of the data appropriate for reporting and analysis.

Figure 12 – Joining Tables (without subqueries)

```
PROC SQL;                    /** join Customer_Demography with Purchases **/
   connect to oracle (user=myusername password=mypassword);
      create table temp1 as
      select *
      from connection to oracle
        (select c.*,
                p.date, p.merchant_id, p.amount
            from Customer_Demography c,
                 Purchases p
            where c.customer_id = p.customer_id);
   disconnect from oracle;
quit;


PROC SQL;                    /** join temp1 with Merchant **/
   connect to oracle (user=myusername password=mypassword);
      create table analysis as
      select *
      from connection to oracle
        (select t.*,
                m.merchant_name
            from temp1  t,
                 Merchant m
            where t.merchant_id = m.merchant_id
      order by t.customer_id);
   disconnect from oracle;
quit;
```

Views

Much as in SAS, views in Oracle can perhaps best be thought of as templates through which one can "view" the physical data stored in tables. The data are not stored in the view, which is created "on the fly" each time someone executes a query against it. These are thus efficient with space. They also can embody specific, validated ways of looking at certain data and can incorporate selection criteria (only certain transaction or contract types), standard data transformations and definitions, and can automatically join codes to their descriptions in reference tables or handle issues around the display or masking of privacy fields (such as Credit Card Number or Social Security Number). As standard, understood views of the underlying tables, these can move the analyst far along the work stream as the programming required to replicate these results is already created, validated, and stored for use. No additional data storage requirements are created as these only exist on demand, then disappear at the end of a users' session. A frequent naming convention will resemble **V_mytablename** or **mytablename_V**. It is not unusual for end user access to be limited to the views.

There are also challenges in using these. As views consist of queries (the end user's SQL pass-through query is querying the results of the query which creates the view), the physical result set will surface at the conclusion of the view-creation query. In contrast the underlying tables are physically present immediately for the SQL pass-through query to execute against. In practice this may mean that views can be slower to access than the underlying tables.

Other challenges include that some of the nice query tools (indexes and partitions) are associated with the underlying tables but **NOT** with the views. Applying some of the same engineering principles (query first against the most limiting selection criteria, then the next, etc.), list columns in the index order, select by partitions are said to help with views as well. But you will need to benchmark your results in this regard.

CONCLUSION

As more of our data are stored in DBMS other than the familiar SAS dataset, there is value in becoming familiar with using some of the powerful selection and retrieval tools available therein. We have illustrated a number of such tools as well as suggested programming considerations using the SAS SQL pass-through facility to Oracle. As we may be querying against enormous datasets, almost every execution of a given program may be a candidate for a modicum of computer engineering/fine tuning. In ever-changing conditions, on-going attention to benchmarking is appropriate. Note that installation-specific conventions should be adhered to, and always leverage the learnings of colleagues.

REFERENCES AND RESOURCES

Agnihotri, Kunal and Borowiak, Ken, 2014, "Tips for Creating SAS®-Based Applications for Oracle Clinical", Proceedings of the PharmaSUG 2014 Conference, San Diego, California. Available at:

http://pharmasug.org/proceedings/2014/AD/PharmaSUG-2014-AD19.pdf

Date, C. J., 2000, **An Introduction to Database Systems**, 7th ed., Addison-Wesley.

SAS Institute, Inc., 2010, "SQL Pass-Through Facility Specifics for Oracle", **SAS/ACCESS(R) 9.2 for Relational Databases: Reference**, Fourth Edition, Cary, NC, SAS Institute, Inc.

http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#a003113595.htm

Schacherer, Christopher W. and Detry, Michelle A., 2010, "PROC SQL: From SELECT to Pass-Through SQL ",Proceedings of the SouthCentral SAS Users Group, Austin, Texas. Available at:

http://www.scsug.org/SCSUGProceedings/2010/Schacherer_2/PROC_SQL_%20From_SELECT_to_Pass-Through_SQL.pdf

Wikipedia, "Database", retrieved 8/31/2016. Available at:

https://en.wikipedia.org/wiki/Database

Williams, Christianna, 2015 , "PROC SQL for PROC SUMMARY Stalwarts", Proceedings of SAS Global Forum 2015, Dallas, Texas. Available at:
http://support.sas.com/resources/papers/proceedings15/3154-2015.pdf

TRADEMARKS

CONTACT INFORMATION

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

Name:	John J. Cohen
Enterprise:	Advanced Data Concepts, LLC
City, State:	Newark, DE
Work Phone:	(302) 559-2060
E-mail:	jcohen1265@aol.com