Paper 072-2013

SAS-Oracle Options and Efficiency: What You Don't Know Can Hurt You

John E. Bentley, Wells Fargo Bank

ABSTRACT

SAS/Access® engines allow us to read, write, and alter almost any relational database. Using the engine right out of the box works OK, but there are a host of options and techniques that if properly used can improve performance, sometimes greatly. In some cases though an incorrect option value can degrade performance and you may not even know it. This paper will review SAS/Access for Oracle engine options that can impact performance. Although the focus is on Oracle, most of the options reviewed are cross-database. All levels of SAS® programmers, Enterprise Guide® users, and Oracle and non-Oracle database users should find the paper useful.

INTRODUCTION

SAS Institute and Oracle Corporation have a cooperative relationship going back over 25 years to SAS Version 6. The shared goal in the partnership is to insure that their shared customers fully benefit from an integrated and optimized SAS-Oracle environment.

An on-going joint initiative is a "modernize and optimize" program in which SAS and Oracle staff team up to provide architectural recommendations that take advantage of the latest SAS Analytical Solutions and Oracle Exastack, Solaris, and WebLogic technologies. SAS and Oracle also partner to offer "SAS Grid-in-a-Box", a package that combines SAS Grid Manager with Oracle's Exalogic System. With SAS Version 9.2, a half-dozen Base procedures were enhanced to use Oracle in-database processing for significant performance improvement—the aggregations and analytics are run inside the database and only the much smaller results set is returned to SAS.

Advances like in-database processing are still in the future for many if not most SAS users. Today, using SAS to read from and write to Oracle tables is probably the most way SAS and Oracle are used together. In this paper we'll start with a short overview of SAS/Access for Oracle and then address specific challenges you might find when working with relational database data. Solutions will of course be suggested.

In the SAS environment, we work with libraries, data sets, observations, and variables. Oracle calls these objects schema, tables, rows, and columns. Observations and variables are also known as records and fields. In this paper the names will be used interchangeably.

A QUICK OVERVIEW OF SAS AND ORACLE

One of the results of the SAS-Oracle partnership is that SAS easily works with Oracle tables via SAS/Access for Oracle. Tools like SQL*Plus, SQL Navigator, or Toad for Oracle not are really necessary but can be useful for simple exploratory analysis of an Oracle table. With SAS/Access, we can use the Oracle LIBNAME Engine to do probably 90% or more of work we need to do with Oracle. For the other 10% SAS's SQL Pass-Through Facility allows queries written with native Oracle extensions as well as Oracle DDL, DCL, and TCL statements to be passed directly to the database so you can do things like generate statistics and grant permissions.

Using SAS/Access for Oracle you can join not only SAS data sets and Oracle tables, but an Oracle table to an Oracle table. If you have a second SAS/Access product like Access to DB2, you can join Oracle and DB2 tables, with or without a SAS data set.

This paper is not an introduction to SAS/Access, so we'll just say that SAS/ACCESS for Oracle provides access to Oracle data via three methods—

- A LIBNAME statement with the Oracle engine allows an Oracle table to be referenced using a two-level name in a DATA Step or PROC just as you would with a SAS data set. You do not need to know Oracle SQL syntax or extensions because the SAS code is translated into Oracle behind the scenes.
- The CONNECT TO enables the SQL Pass-Through Facility that allows native Oracle SQL to be coded and passed directly to the database for processing. This insures that the database can optimize queries and take advantage of indexes. Even better, SQL Pass-Through can execute all Oracle data definition, data manipulation, data control, and transaction control statements.
- The ACCESS and DBLOAD procedures are no longer recommended but still support indirect Access for Oracle tables.

Task	*How to do it*
Read a table or view	• LIBNAME statement • Pass-Through Facility • View descriptors
Create a table and SELECT or process fields in an existing table	• LIBNAME statement • SQL Pass-Through Facility • PROC DBLOAD
Update, delete, or insert rows into a table	• LIBNAME statement • SQL Pass-Through Facility • View descriptors
Append rows to a table	• LIBNAME statement with APPEND procedure • PROC DBLOAD with the APPEND option • SQL Pass-Through Facility
List tables in a schema	LIBNAME statement and • the SAS Explorer window • PROC DATASETS • PROC CONTENTS • PROC SQL and dictionary tables
Delete tables or views	• LIBNAME statement, PROC SQL and the DROP Table statement • LIBNAME statement, PROC DATASETS and the DELETE statement • PROC DBLOAD with DROP TABLE statement • SQL Pass-Through Facility's EXECUTE statement
Grant or revoke privileges, Create a savepoint Rollback transactions Call a PL/SQL program	• SQL Pass-Through Facility EXECUTE statement

Table 1. SAS/Access for Oracle, Common Tasks

CONNECTING TO ORACLE

This paper won't go into details, but when specifying a LIBNAME for Oracle for the first time, the best approach is to get libref code from someone who already has it working. (Does that sound familiar?) If you can't get one, the *SAS/Access Supplement for Oracle* lays it out very well and it's actually quite straightforward. The References section has a link to a SAS Tech Note with some examples.

You can connect SAS on a desktop directly to an Oracle database or you can work with the SAS/Windows Editor and remote submit to SAS on a server that connects to the database. Either way, Oracle Client software must be installed on the platform that connects to the database and Client must have a driver that matches the SAS version that's running.

- SAS 9.1 requires the Oracle Client 8.1.7 or higher
- SAS 9.2 and 9.3 require the Oracle Client 10.2 or higher

If SAS won't connect to Oracle, the error messages are very obvious. For example,

```
libname orcDB oracle path='orcl4' schema='test_hm' user=tarzan pass='Cheetah#1';

ERROR: The SAS/ACCESS Interface to ORACLE cannot be loaded. The SASORA code
appendage could not be loaded.
```

In this case, the problem is that the SAS executable SASORA can't be found and you should double-check to see that SAS/ACCESS for Oracle is properly installed and available to your session. Other troubleshooting steps might include checking that the LIBNAME PATH= option is correct—here you place the same Oracle path designation that you use to connect using one of the other tools like SQL Navigator. Also see if you can establish an Oracle session using a something like SQL Navigator. Always try support by friendship—find someone who can connect and see how they do it.

CHALLENGE #1 – DIFFERENT NAMING CONVENTIONS

While SAS and Oracle work well together, there are important basic differences in naming conventions that you need to know.

- SAS and Oracle both allow long names for tables and fields but the maximum length is different—32 versus 30 characters.
- The naming rules themselves are different. Oracle names must begin with a letter and can include underscore (_), #, and $ unless the name is enclosed in double quotes. Then it can begin with a number and include any special characters and blanks. This is similar to the rarely seen (for the author anyway) SAS name literal that is a quoted string followed by the letter n, such as 'December 2012 balance'n;
- SAS is case insensitive when it comes to names. It stores names as case-sensitive for readability but ignores case during processing. EMPLOYEE and employee are the same variable. But in Oracle, double-quotes make the name case sensitive—employee, "employee', "Employee", "EMPLOYEE", "eMpLoYee" are all valid field names that can be in the same table and are all valid tables in the same schema.
- Both SAS and Oracle have specific words that are reserved and cannot be used for table or field names. Oracle's list is quite extensive and differs greatly from SAS's. The Reference section has a couple useful links listing reserved words.

What happens when SAS encounters an Oracle field name that is non-standard for SAS? The default behavior is to replace the 'invalid' character with an underscore (_) and then appends a number to preserve uniqueness. Oracle fields balance$ and balance% from the same table would become balance_1 and balance_2 in SAS.

And what about Oracle? Well, it simply throws a very obvious error when you try to use an invalid name. There's nothing to be done about that but there are two SAS options presented in Table 1 that tell Oracle how to handle what it sees as a non-standard name being passed from SAS.

OPTIONS THAT HELP WITH NAMING ISSUES

If you expect to be querying Oracle columns that are not valid SAS variable names then you can use either the DQUOTE=ANSI option with PROC SQL or the VALIDVARNAME=ANY option with a LIBNAME. This will allow queries like this to run—

```
libname orcDB oracle path='orcl4' schema='test_hm' user=tarzan pass='Cheetah#1';

proc sql dquote=ansi;
       select sum ('TOTAL_Cost$') as total_cost
       from orcDB.activity;
```

Option	***Valid Values***	***Description/Explanation***
DQUOTE=	ANSI | SAS	A PROC SQL option, overrides the VALIDVARNAME option if present. The value ANSI allows SAS to treat values enclosed in double-quotes as a variable name. This allows SAS to use non-standard Oracle column names. The default is SAS and values in double-quotes are treated as character strings.
VALIDVARNAME=	V7 | UPCASE | ANY	Specifies the naming rules for SAS names to use during the session. The value ANY allows compatibility with RDBMS names that contain blanks and special characters. Names will be expressed as a *name literal*, enclosed in single quotes and having a training n. 'December 2012 balance'n
PRESERVE_NAMES=	YES | NO	Combines both PRESERVE_COL_NAMES= AND PRESERVE_TAB_NAMES=.
PRESERVE_COL_NAMES=	YES | NO	Controls how Oracle handles non-standard column coming in from SAS when a table is created. • The default is NO and Oracle converts column names using its internal naming conventions (e.g., uppercase) • YES retains the case of the word and, if you're using SAS name literals, special characters.

Option	*Valid Values*	*Description/Explanation*
PRESERVE_TAB_NAMES=	YES | NO	Controls how Oracle handles non-standard table SAS-specified table name. The impact of the valid values is the same as with preserve_col_names. When using a LIBREF to display a list of Oracle tables in the SAS explorer window, PRESERVE_TAB_NAMES= controls whether or not nonstandard Oracle table names are displayed. To see them, specify YES.

Table 2. Options for Handing Naming Conventions

CHALLENGE #2 – READING ORACLE TABLES

Where the author works, the bulk of our Oracle activity involves using it as a data source—reading data from Oracle tables. Our databases contain a mix of tables that are basically flat files (lots of redundant data) and first, second, and third normal forms that can require complex joins and self- and recursive joins. Many of the tables—especially those created and maintained by smaller departmental teams—are not partitioned or don't have indexes. Queries return results sets ranging from a few hundred or thousand records to 30+ million rows. Ad-hoc analytic jobs run any time during the day but production jobs generally run at night between about 7 pm and 7 am.

The mechanics of SAS querying a database are simple. SAS statements whether in PROC SQL, a DATA Step, or a PROC like FREQ or MEANS are converted into SQL by the SAS/Access software and passed to the database. The database executes the SQL and produces a results set consisting of rows and columns. The results set is then passed back to SAS for further processing or, less often, written to an Oracle table.

The very first option to be aware of and consider changing is READBUFF=. This value sets the number of records read into the memory buffer and returned to SAS in a single fetch operation. The default is 250 and for a narrow results set (one that has a comparatively short record length) that may be too few. On the other hand, for a wide results set it may be correct. Adjusting the number of records held in the buffer can improve or degrade performance. Keep in mind that increasing the size of the buffer uses more memory and you may run into constraints or diminishing returns. For production jobs, this option is worth experimenting with to find the optimal size.

To see the current value of READBUFF or any option, use PROC OPTIONS.

```
proc options option=readbuff define value lognumberformat;
run;
```

THREADED READS

An important 'trick' to extracting a large volume of data efficiently from Oracle is to use threaded reads. The operative word here is 'large'. Threaded reads significantly increase performance only when there is a whole lot of data being read—we're talking hundreds of megabytes or gigabytes—and they have some potential to actually slow things down. They're also a double-edged sword. (Nothing is free, right?) They can greatly reduce the elapsed time of a SAS step but increase the load on the database. They reduce the time it takes to move data across the network but use more bandwidth during that time. This paper cannot provide comprehensive instructions for using threaded reads, so for the details look at the *SAS/Access for Relational Databases: Reference.* There are also a number of SAS Conference papers that address threaded reading.

Starting with Version 9, SAS has the ability to retrieve a database query results set using multiple simultaneous data streams, a technique called threaded reading. Multiple database connections are spawned, used, and then closed. For very large amounts of data, the performance improvement is amazing. All things being equal, using four threads will return the data in one-quarter the time it takes for a single thread.

SAS's threaded reading capability is enabled in the SAS configuration file, at invocation, or with system options. The option THREADS= enables SAS PROCS and the DATA Step to take advantage of multiple CPUs even when databases aren't involved. The option CPUCOUNT= specifies the maximum number of CPUs to use. Finally, there is also an option that controls when SAS uses threads to read databases: DBSLICEPARM=.

DBSLICEPARM= can be set at the system, libref, or procedure level and the rules of precedence apply. When the options THREADS is specified, by default DBSLICEPARM= is set to THREADED_APPS. With this value the SQL, MEANS, SUMMARY, REG, and SORT Procedures and some others are eligible for threaded read processing. The value ALL extends threaded reading eligibility to steps that only read data, specifically the DATA Step and PROC PRINT. The NONE value turns off threaded reading even when THREADS is enabled. NONE remains in effect until THREADED_APPS or ALL is issues. There is a second argument for DBSLICEPARM= that overrides the value of CPUCOUNT to limit the number of threads.

In this example, SAS will attempt to always perform threaded database reads using four threads.

```
OPTIONS THREADS CPUCOUNT=4 DBSLICEPARM=ALL;
```

With threading enabled, SAS *tries* to make the database partition the results set and if successful assigns one thread per partition. 'Partitioning' here means breaking the results set into a number of separate results sets. How to partition the data is handled behind the scenes by SAS automatically adding a WHERE clause to the SQL so that the query itself become multi-threaded, resulting in multiple results sets. The WHERE applies the MOD function to generate a partitioning strategy. But as the documentation points out, MOD can't always be used and it's not guaranteed to produce the best approach. For details, see the "Autopartitioning Techniques" section of the SAS/Access documentation and also read up on the MOD function.

Threaded reads will not automatically occur in certain situations:

- If all the columns eligible for the MOD function are used in WHERE criteria.
- If no MODable columns are in the table, e.g., the table is all character fields.
- When a BY statement is used.
- When the OBS= or FIRSTOBS= options are used.
- If the NOTHREADS options is set.
- If DBSLICEPARM=NONE.

But don't worry… if SAS can't automatically generate a partitioning WHERE clause or you don't like what it does there is an options that allow the user to define the partitioning strategy: DBSLICE=.

DBSLICE= is a table-level option, not library-level. It allows coding a custom WHERE clause to pass to the database but requires that you are very familiar with the distribution of the data. Be very careful here: *if you specify criteria that omits or duplicates rows then it will return an incorrect results set if there are no syntax errors*. Also, it is important that your partitions contain approximately the same numbers of records. The largest partition determines how fast the read finishes.

For example, if you have a field that can be aggregated into four fairly-equal sized groups, then the code might look like this. The rule is to keep it simple.

```
data transact; set orcDB.transactions
          (dbslice=("sgmt_cd in ('A','D')"      "sgmt_cd in ('B','C')"
                    "sgmt_cd in ('E','F','H')"  "sgmt_cd='G');
run;
```

TRACING DATABASE ACTIVITY

Once you've told SAS to use threaded reads, how do you track what it does? Well, as you might expect there's an option for that too. SASTRACE= writes information to the log showing where threaded reads happen along with information about the performance impact.

SASTRACE= must be set at invocation, configuration or as a system option and it has valid values that determine what output is captured. There are six valid values and they can be combined. Notice that the values are delimited by single quotes and how commas are used.

Value	***Description***
',,,d,'	Only SQL statements set to the database are sent to the log.
',,d,'	Metrics for routine database calls are sent to the log. These include function enters and exits along with parameters and return codes. Most useful for troubleshooting database issues.
'd,'	Captures information for all database calls including connection info, API and Client calls, column bindings, errors, and row processing. Useful for troubleshooting. Can produce a lot of log pages.
',,,db'	New with Version 9.2. Returns a briefer version of the information produced by 'd,'.
',,,s'	Produces a summary of timing information for the database calls. Very useful for seeing how well the database is working.
',,,sa'	In addition to a summary of database timing information, detailed timings for each call is written to the log.
',,t,'	All threaded information is sent to the log. This includes the number of threads spawned, the number of records returned by each thread, and the thread exit code if it fails. Useful to see how well the partitioning strategy worked.

Value	*Description*
OFF	Turns off tracing.

Table 3. SASTRACE Valid Values

Along with SASTRACE= you should specify NOSTSUFFIX to suppress diagnostic information and make the log easier to read. The SAS/ACCESS documentation has a good example of what the output looks like if the diagnostic info isn't suppressed.

By default, SAS Trace information is routed to the server-side standard output location, which is probably not what you want. With the SASTRACELOC= system option you can direct the info to the SAS Log or a separate log file.

SASTRACELOC= stdout | SASLOG | FILE '*path and file-name*'

Along with SASTRACE= you should use the NOSTSUFFIX to suppress diagnostic information and make it easier to read. The online documentation has a good example of what the output looks like if the diagnostic info isn't suppressed.

Here is an example of the Options that will turn on and control threaded database reads and the output that SASTRACE returns. It shows that the threaded read autopartitioning strategy isn't optimal for this particular query.

```
options threads cpucount=4 dbsliceparm=all
        sastrace=',,t,' sastraceloc=saslog nostsuffix;

169  data custType5a; set hemijb.cust_sum(where=(cust_type='C5a') and cust_cd=5);
170  run;

ORACLE:  DBSLICEPARM option set and 4 threads were requested
ORACLE:  No application input on number of threads.
ORACLE:  Thread 1 contains 128141 obs.
ORACLE:  Thread 2 contains 11586 obs.
ORACLE:  Thread 3 contains 127128 obs.
ORACLE:  Thread 4 contains 132832 obs.
ORACLE:  Threaded read enabled. Number of threads created: 4
```

NULL vs. MISSING

Oracle and SAS treat missing data differently and that can cause apparent discrepancies during data validation checks. Oracle has a special value called Null. Null means 'absence of value' and it doesn't equal or not equal anything including another Null. A Null isn't evaluated in a comparison operation. In an operation like `where x < 10`, Oracle ignores records where x is Null but SAS would include missing values in the results set.

When SAS reads an Oracle Null it interprets it as a missing value. That's fine, but to Oracle a Null (`where state is null`) is not the same thing as a blank value (`where state=' '`). SAS interprets both Nulls and blanks coming from Oracle as missing values. Applying a format is during the read doesn't solve the problem because the format is applied after the data is in SAS and has already been 'converted' to a missing value.

If telling Nulls from blanks is important, it must be solved on the Oracle side. Nulls can also impact WHERE clauses and CASE statements. Oracle does not consider Null values in a WHERE clause or a CASE statement unless they are specifically referenced. Consider `case when x='Y' then 1 else 0 end as y`; In SAS, the value Y for a missing X would be 0 but Oracle will assign a 0 when X is blank and a Null when X is Null.

Be very careful working with data that contains Nulls. Here's a query using SQL Pass-Through that shows an approach if you have a need to distinguish between blanks and Nulls.

```
proc sql threads;
     connect to oracle (path='orcl4' user=tarzan pass='Cheetah#1')
     create table testread as
        select * from connection to oracle
          (select customer_id
                , (when customer_type=' ' then ' '
                   when customer_type is null then 'X'
                   else customer_type end) as customer_type) end as cust_type
          from wfjbent.cust_base;
     disconnect from oracle;
   quit;
```

SUMMARY OF OPTIONS FOR READING TABLES

PROC SQL has its own rich set of options and if you write SQL you should take the time to review them. Here are some of the read-related options the author finds most useful.

Option	*Valid Values*	*Description/Explanation*
READBUFF=	integer value	Specifies the number of database rows to return from the results set in one fetch operation. A reasonable higher number usually improves performance by returning data faster at the expense of higher memory usage
DBSLICE=	Two or more properly specified WHERE clauses.	A data set option. Provides a user-supplied WHERE clause for partitioning the results set so that threaded reads can return the data to SAS.
DBSLICEPARM=	NONE | ALL | THREADED_APPS	Controls when threaded read processing is applied. The default is THREADED_APPS.
DBSASTYPE	<col-name-1>=<'SAS-data-type'>	A data set option that specifies a data type to override the default SAS data type during processing. If for example a field has an Oracle data type of decimal(20) by default SAS will display it in scientific notation. `set X(sasdatatype=(var1='char20'));` lands var1 as a character string.
DBSASLABEL=	COMPAT | NONE	• When creating a data set, by default the value COMPAT causes SAS to write the database column name as the label. The value NONE leaves the label blank. • When reporting via PROC SQL, the value NONE ignores an alias assigned to the column and uses the column name itself as the header.

Table 4. Options for Reading Oracle Data

CHALLENGE #3 – WRITING TO ORACLE TABLES

Writing SAS data to Oracle is simple but it can also be time consuming. Proper use of certain options is guaranteed to speed things up. Writing to an Oracle table is by default a single-threaded operation. There is an easy technique to change that to a multi-threaded operation but first let's look at the default because there are many situations where that is entirely appropriate.

USE BUFFERS

A simple libref with the Oracle engine and a DATA Step or PROC SQL are all that are needed to create an Oracle table, add rows, or modify existing records data. Here are three basic examples that will work just fine but might take a l-o-n-g time if there are lots and lots and lots of rows.

```
libname orcDB oracle path='orcl4' schema='test_hm' user=tarzan pass='Cheetah#1';

** Create a table containing January customer data. ;
data orcDB.jan_data; set year_to_date_data(where=(mon=1));
run;

** Create a table containing February customer data. ;
proc sql;
  create table orcDB.feb_data as
     select *
     from year_to_date_data
     where mon=2;
quit;
```

```
** Revise the December customer data.  Libref revised is a SAS library. ;
proc sql;
  update orcDB.dec_data as orc_table
     set total_refund_amt=
          (select sas_ds.total_refund_amt
           from revised.adjusted_dec_refunds as sas_ds
           where sas_ds.cust_id=orc_table.cust_id)
     where exists
          (select 1
           from revised.adjusted_dec_refunds as sas_ds
           where sas_ds.cust_id=orc_table.cust_id);
```

The default buffer value is probably not optimal (depending on the number of records) because by default SAS uses buffer values that guarantee the processing will work if the code is syntactically correct but are not optimized for the amount of data involved. Loads and updates will work, but performance may leave much to be desired. Changing a couple options, however, may make a big difference.

The first two examples will use the SQL Insert operation to load the tables, even the DATA Step. Inserting records, especially lots of long records, is a slow process for Oracle even though it does it at a phenomenal speed. Pick a number—1000. At 1000 inserts per second, it still takes over 16 minutes to insert 1,000,000 records. Add to this the time it takes to execute a commit operation that permanently writes the new records to the table. By default, SAS tells Oracle to issue a commit after every 10 inserts so for a million records that's 100,000 commit operations. If it takes .001 seconds to run a commit, 100,000 commits add another minute and a half to processing. So we're up to about 17 minutes.

The example above that updates an Oracle table will use SQL Update operations and updates are slower than the Inserts because we have to find which records to update. In fact, we probably don't want to run this update as coded here because of the need to join SAS with Oracle. We'll revisit that topic in Challenge #5.

For both of these examples, performance can be perhaps dramatically improved by using options that control how many records are inserted or updated in a single operation. INSERTBUFF= and UPDATEBUFF= control how many records are processed "at once", and DBCOMMIT= controls how many records are processed before a commit is issued. The default values are low enough to insure that the processing will work but not high enough in probably most cases to get good performance. Here is a libref that should allow inserts and updates to work faster when there is a large number of records.

```
libname orcDB oracle path='orcl4' schema='test_hm'
        user=tarzan pass='Cheetah#1' insertbuff=5000 updatebuff=25;
```

The Somerville and Cooper paper "Optimizing SAS Access to an Oracle Database in a Large Data Warehouse" (see References section) has tables showing the impact of different values for INSERTBUFF= and UPDATEBUFF=. Trial-and-error is often not the fastest way to figure something out, in some cases it is the only way. The buffer options' optimal values are based on how many records you're working with and how wide they are. So for production jobs it would be useful to experiment with different values to find what works best for that particular job.

PARTITION NEW TABLES AND TURN OFF LOGGING

In Oracle, partitioned tables are faster to read and update so it behooves us to partition new tables when we create them. Large Oracle tables (large is site-specific) should be organized into smaller pieces—partitioned—based on a key such as customer number. All data related to that key is in the same partition and that makes data retrieval/update faster. To a program, a partitioned table is no different from a non-partitioned table, but Oracle doesn't partition a table unless it's told to.

DBCREATE_TABLE_OPTS= is a valuable option because it allows us to pass native Oracle syntax that is added to the `CREATE TABLE` statement and causes thing like partitioning to happen. Generally speaking, partitioning large tables is important because it

- Provides faster performance by lowering query times from minutes to seconds;
- Improves manageability by creating smaller 'chunks' of data; and
- Enables more cost-efficient use of storage.

Partitioning improves performance after the table is created. Turning off logging on the other hand speeds up the table creation process itself. Logging takes space by creating rollback files and adds time writing log entries so turning it off reduces time but at the cost of not being able to rollback records inserted in error. Turning off logging does not affect the ability of subsequent data manipulation statements like UPDATE and DELETE to generate rollback files.

This code example partitions the dataset we're creating and turns off logging.

```
proc sql;
  create table orcDB.feb_data
     (dbcreate_table_opts=nologging
                          partition by hash (cust_id) partitions 4) as
     select cust_id
          , <field list>
          , case when eom_bal=. then ' '
                 else eom_bal end as eom_bal
     from year_to_date_data(nullchar=no nullcharval=' ')
     where mon=2;
quit;
```

HANDLE MISSING VALUES PROPERLY

By default Oracle loads a Null when it finds a SAS missing value. This may be what you want, but two data set-level options are used together control handling SAS missing character values when loading or updating an Oracle table.

NULLCHAR=NO tells Oracle to load a special value instead of Null and the special value is specified with the NULLCHAR= option. To recode missing numeric values when nulls are not wanted you must use an `IF-THEN-ELSE` or `CASE` statement on the SAS side before the data is sent to Oracle.

In this example we use a CASE statement and options to load blanks instead of Nulls for missing values.

```
proc sql;
  create table orcDB.feb_data as
     select <field list>
          , case when eom_bal=. then ' '
                 else eom_bal end as eom_bal
     from year_to_date_data(nullchar=no nullcharval=' ')
     where mon=2;
quit;
```

If we wanted character field-specific special missing then we would use a `CASE` statement to recode the character Nulls.

SUMMARY OF OPTIONS FOR WRITING TO ORACLE

Bulkload Option	***Valid Values***	***Description/Explanation***
INSERTBUFF=	a positive-integer	Specifies the number of rows moved to the buffer and processed in a single database insert operation. The default value is 10.
UPDATEBUFF=	a positive-integer	Specifies the number of rows moved to the buffer and processed in a single database insert operation. The default value is 1.
DBCOMMIT=	a positive-integer	• Causes a Commit operation (permanent writing) after a specified number of rows have been processed. If the value is less than INSERTBUFF or UPDATEBUFF then the COMMIT= overrides the other value. A value of 0 causes the commit to be issued only after all rows have been processed. • This option must be coordinated with ERRLIMIT=. The DBCOMMIT value overrides the ERRLIMIT value. It is possible that a rollback initiated by ERRLIMIT might not include bad records already committed.
DBCREATE_TABLE_OPTS	*Oracle-specific syntax*	Specifies Oracle code to be passed to the database engine as part of the CREATE TABLE statement. Useful for controlling partitioning and logging.

Bulkload Option	*Valid Values*	*Description/Explanation*
DBLABEL=	YES | NO	Specifies whether to use SAS variable names or the variable labels for column names when creating a table. Default is NO, use variable names.
DBNULL=	_ALL_=YES | NO | *column_name*=YES|NO	Indicates whether or not Null is a valid variable for a column. When YES, applies a constraint that rejects the record and reports an error when a Null value is found. _ALL_ specifies that constraint is applied to all columns, or specific columns can be named. Default is _ALL_=NO.
ERRLIMIT=	a positive-integer	• Specifies the number of errors allowed before SAS stop processing and issues a rollback command. The value 0 causes SAS to process all rows regardless of the number of errors. • This option must be coordinated with DBCOMMIT=. The DBCOMMIT value overrides the ERRLIMIT value. It is possible that a rollback initiated by ERRLIMIT might not include bad records already committed.
NULLCHAR=	SAS | YES | NO	Indicates how missing SAS character values are handled during insert and update processing.
NULLCHARVAL=	*'character string'*	When NULLCHAR=SAS or YES, defines the character string to use instead of a Null value.

Table 5. Options for Writing to Oracle Tables

BULK LOADING—NOT A SILVER BULLET BUT CLOSE

Bulk loading gets around the performance restriction imposed by Insert operations when creating a new table or inserting records into an existing table, and SAS supports Oracle's bulk loading facility (SQL*Loader) via the BULKLOAD=YES option. Bulk loading provides amazing load performance compared to not using it simply because without it you're doing single-threaded insert operations. Your mileage may vary but the author recommends using the bulk loading facility when you have more than maybe 10,000 records of more than a half-dozen fields.

The bulk loader is available with both a DATA Step and PROC SQL for creating a new table, and PROC APPEND is best for adding rows to an existing table. In its most basic form, it looks like the examples below.

```
libname orcDB oracle path='orcl4' schema='test_hm'
                        user=tarzan pass='Cheetah#;

data orcDB.finalCustomerUpdate(bulkload=yes);
  set customerUpdateStep5;
  <...process 100,000 observations each with 242 variables...>
run;
```

If you're not a SQL fan and you have to do complex data modifications and transformation the easiest way to do it is in two steps. Use a DATA step to manipulate the data and save a work data set and then run PROC SQL or PROC APPEND to load the data.

```
proc sql;
  create table orcDB.productOwnership_&_when(bulkload=yes) as
     select *
     from work.finalCustomerUpdate;
quit;

proc append base=orcDB.productOwnership
                (bulkload=yes bl_index_option='sorted indexes(product)')
            data= work.finalCustomerUpdate;
run;
```

PROC APPEND is a powerful way to add a large amount of data to an existing table, such as a transaction or history table. Appending with bulk loading is fast and it places the data in the correct partition.

Oracle indexes are by default dropped and rebuilt when bulk loading with PROC APPEND. If your data is already sorted by one of the index keys, you can specify the BL_INDEX_OPTION= as we did in the example and the index will be rebuilt as the table is loaded. The performance gain can be significant. If your data isn't already sorted or you have many indexes then it doesn't necessarily make sense to sort prior to loading.

PROC SQL by default creates a new table without partitions or indexes regardless of whether or not bulk loading is used. Using bulk loading still allows the DB_CREATE_TABLE_OPTS= option to be used for partitioning and turning off logging, and indexing can be done using Oracle commands executed via the Pass-Through Facility's EXECUTE (…) syntax. The example below creates an index for a table created with PROC SQL.

```
proc sql;
  connect to oracle path='orcl4' user=tarzan pass='Cheetah#1');

  create table test_hm.productOwnership_&_when
     (bulkload=yes
      dbcreate_table_opts=nologging partition by hash (cust_id) partitions 4) as
     <complex query>
  ;

   execute (
     create index on orcDB.productOwnership_&_when (product_cd, cust_cat_cd)
   ) by oracle;

   disconnect from oracle;
quit;
```

Behind the scenes, when you specify BULKLOAD=YES a flat file of the data and a record layout file are created and landed by default in the current directory. A control file with Oracle specifications for loading the table such as the location of the flat file and record layout is also produced. The Oracle SQL*Loader utility is then invoked and executes the control file. During loading a log of the bulk loading activity is created. This log is different from the one we turn off with DBCREATE_TABLE_OPTS=NOLOGGING.

In the SAS program, if you get an error that says something like Oracle can't find the bulk loader—it will be obvious—then contact a DBA and ask that the SQL*Loader executable be made available to your user id.

Bulk Loading Options

BULKLOAD=YES supports quite a few options that can add or subtract from performance and Jun Cai's paper "How to Leverage Oracle Bulk-load for Performance Contact Information" (see references) does a great job of explaining them. Most options apply to all RDBMS, but some are specific to one or more databases. The table below lists some that the author finds useful for Oracle. The SAS/ACCESS documentation is very good and provides generic usage examples.

Bulkload Option	***Valid Values***	***Description/Explanation***
BL_CONTROL=	<path-and-filename.ext>	Names a file to capture the Oracle SQL*Loader DDL statements.
BL_LOG=	<path-and-filename.ext>	Names a file to capture a log of the statistics and messages produced by Oracle during the load.
BL_DATAFILE=	<path-and-filename.ext>	Names the file that contains the data to be loaded to an Oracle table.
BL_BADFILE=	<path-and-filename.ext>	Names the file to hold rejected records. Empty if zero problem records.
BL_DISCARDFILE=	<path-and-filename.ext>	Names a file to hold records that do not meet any WHERE criteria specified in the SQL. If an error is later found in the criteria, the discard file can be loaded.
BL_DELETE_DATAFILE=	YES | NO	Controls if the data file is deleted after the load is completed. Default is Yes but the file is retained if the load completely fails. Caution: disk space management says do not set this to NO unless there is a good reason.

Bulkload Option	*Valid Values*	*Description/Explanation*
BL_DIRECT_PATH=	YES | NO	Sets the SQL*Loader option for using the Direct or Conventional path loading technique. YES forces the Direct Path to be used and that is significantly faster but restrictions apply. NO forces the Conventional Path technique. The default is YES if the Direct Path is available but that depends on your SQL*Loader installation. Best to speak with your DBA about it.
BL_INDEX_OPTIONS=	SINGLEROW | SORTED INDEXES	Used with PROC APPEND. When the data is sorted by an index key specify the SORTED INDEXES value to optimize performance.
BL_LOAD_METHOD=	INSERT | APPEND | REPLACE | TRUNCATE	Specifies the method to use when bulk loading a table. • INSERT requires a new or empty table; • APPEND adds rows to an existing table. • REPLACE deletes all rows in existing table, and then loads new rows. • TRUNCATE uses the Oracle truncate command to drop and replace data,. Faster than REPLACE. REPLACE and TRUNCATE are valid only with PROC APPEND. Defaults: INSERT when creating a table, APPEND when loading an existing table.
BL_OPTIONS=	ERRORS= and LOAD=	Allows two options to be passed directly to the SQL*Loader facility. • ERRORS= specifies the number of insert errors allowable before aborting the load. Default is 50. • LOAD= specifies the number of records to load. This is valuable during development and testing for loading only a test table of a few dozen records without creating a subset or sample data set.
BL_PRESERVE_BLANKS=	YES | NO	Determines how SQL*Loader handles character missing (blank) values. YES causes the field to be filled with blanks. NO insures that missing values are inserted as NULL values. Default is NO. Caution: YES causes trailing blanks to be inserted. 5-character field will be filled with 5 blanks, not 1. A code of 'AF5' loaded into a 5-character field will be loaded as 'AF5 '.

Table 6. Bulk Loading Options

Writing the control, log and data files and the bad and discarded record files to a specific directory is useful during development and testing QA and might be required during production for validation. By default, the files are written to the 'current' directory and are deleted after a successful run. The default name form is BL_<*table-name*>_ <*unique_id*>.<*ext*>. *table-name* is the name of the table being loaded/appended, and *unique-id* prevents collisions during parallel loads and overwriting when the file is retained between loads. The default extensions are—

- control=.ctl
- log=.log
- data= .dat
- bad= .bad
- discard=.dsc

If you retain the BL_files, use a retention strategy to manage the disk space. BL_DELETE_DATAFILE=NO can be a dangerous setting. When large amounts of data are loaded on a regular basis, the server file space can fill up very quickly. It's also possible to set BL_DELETE_DATAFILE=NO during development and test but then forgetting to

remove it or set it to yes when putting the code into production. If a month later jobs suddenly started failing with a message of 'File system full', people will be ticked.

CHALLENGE #4 – JOINING A SAS DATA SET TO AN ORACLE TABLE

Joining a SAS data set to an Oracle table can be tricky and the documentation is helpful. The author's general rule is to minimize moving data and do the processing where the most data is. But when working with database data you also have to consider where do you want the results set to end up? If you're creating an Oracle table then you might not want to pull the source data into SAS, do the join, and then return the results set to Oracle. If the results set is going into a SAS data set, you might not want to send the SAS data to Oracle to do the join there. But then again the size/amount of data comes into play also… you probably don't want to move the larger of the data sources but sometimes you might. Tricky, eh?

SAS/Access for Oracle by default will always try to pass a join to the database, even when one of the data sources is a SAS dataset. Only when the join can't be passed to the database, such as outer join that also has a WHERE clause, the Oracle data is brought into the SAS environment and joined there. So for optimal performance, avoid outer joins and the SAS data set should be smaller than the Oracle table.

When moving data in either direction, SAS parses the query and moves only the data elements that are SELECTed plus the fields needed to complete the join. Unfortunately, it moves all the records. So if the SAS data set has more records than the Oracle table, performance will suffer. Performance will take a double hit if you need to save the results set back as a SAS data set.

There are, as you expect by now, options that allow us to override this default behavior. But incorrectly specifying these options can degrade performance rather than improve it, so spend some time doing basic exploratory data analysis and looking at the SAS/Access documentation.

As long as the SAS data set is smaller than the Oracle table (number of records) and you want the results set to end up in an Oracle table or the results set is "small", the default behavior is probably tolerable. But telling SAS which is the larger of the data sources is useful because usually that's the one you don't want to move. The data set option DBMASTER=YES identifies the largest data source and forces the data from the other data source to be moved. The option is valid in both PROC SQL and DATA Steps. In this example we force the join to happen in SAS because DBMASTER=YES is attached to the SAS data set.

```
proc sql;
  create table work.refund_data
     select base.cust_id, base.join_dt, base.ytd_tot_amt,
          , (case when refnd.cust_id is null then 0
             else refnd.ytd_refund_amt end) as ytd_refund_amt
     from orcDB.cust_w_refunds as refnd
        , sasData.customer_master(dbmaster=yes) as base
     where base.cust_id=refnd.cust_id;
quit;
```

If the Oracle table is larger than the SAS data set and it has an index on the join key column, the option DBKEY= will improve performance by forcing Oracle to use the index during the join in the Oracle environment. Be careful with this option—the documentation clearly warns that "[p]erformance can be severely degraded without an index" when the option is used.

```
proc sql;
  create table matches
     select dat.cust_id, dat.join_dt, dat.ytd_tot_amt, data.ytd_refund_amt
     from work.cust_w_refundsas base
        , orcDB.master_data(dbkey=cust_id)  as dat
     where base.cust_id=dat.cust_id;
quit;
```

Sometimes you want the join to take place in SAS, and MULTI_DATASRC_OPT= forces the join to happen there regardless of the numbers of records in the data sources. The documentation says that it results in better performance than the DBKEY= option. Restrictions are that it is available only for equi-joins, works only with PROC SQL, is a LIBNAME option, and has only two valid values—NONE and IN_CLAUSE.

The IN_CLAUSE value forces a SELECT COUNT(*) against each table to determine which is the smaller table. If you know which data source is smaller, you can save some time by specifying DBMASTER= to avoid the SELECT COUNT(*) full table scan. Based on the values returned by COUNT(*), the larger and smaller data sources are identified.

An IN clause is constructed containing all the unique values of the join key from the smaller table *or the SAS data set*. This is important—when a SAS data set is one of the data sources, it always provides the keys for the IN clause. The IN clause criteria (a list of the unique join keys) is passed to Oracle along with the data set's WHERE criteria and only those records that meet the IN clause constraint are returned to SAS for the join. This greatly reduces the size of the data set passed to SAS.

MULTI_DATASRC_OPT= has great potential for improving performance in the right situation, but the downside is that the IN clause is limited to 4500 unique values. This means that it probably can't be used for joins that use an account, customer, or household key.

All three of the options here have potential, but only in certain circumstances and they're a bit tricky to use. To simplify things, unless the amount of data is only a few tens of thousands of records the author's standard solution is to bulk load SAS data into an Oracle table, run the join there, and then keep the results in an Oracle table or download them into SAS for further processing. Sometimes performance can be sacrificed for simplicity and reliability.

DBINDEX=	YES | NO | '*index name*'	Detects and verifies that an Index is available for a table. If it can be used then join performance may be improved. • YES= search for a usable index and formulate a WHERE clause that will utilize it in the join. • NO= do not perform an automated search. • '*index name*' provides the name of a specific index to use.
DBMASTER=	YES	Designates which table is larger. The smaller table is passed to the larger table's environment for the join.
MULTI_DATASRC_OPT=	NONE | IN_CLAUSE	Improves equi-join performance by generating an IN clause containing join key values from the smaller table or the SAS data set.

Table 7. Options for Joining SAS and Oracle tables.

CHALLENGE #5 – DATA SECURITY

Because so much of the contents of a database can be sensitive in one way or another, security is increasingly an issue. For a database, security means restricting read access to those with a need to know and write access to the very minimum number of people. SAS preserves and honors all the data security provided by an Oracle database. It doesn't over-ride any security features to restrict or control data access, modification, or deletion. SAS can, however, add a level of security to whatever exists on the database side.

USE VIEWS AND PASSWORDS

Views are a great approach for protecting data from unauthorized access. A view is a subset of a data table or set of data tables defined by SQL and stored as a 'virtual' table. It takes very little storage space because only the definition (the underlying SQL) is stored. Whenever the view is referenced, the definition (SQL) is processed to return the results set. This way, the data tables themselves are not exposed to the outside world. Many databases restrict user queries to views and the users may not even know it! Kirk Lafler's *Proc SQL: Beyond the Basics Using SAS®* has a good chapter on views.

SAS can easily use PROC SQL to create views pointing to Oracle tables and with them the Oracle security remains in place and must complied with. For example, Oracle can assign users to a 'role' and then grant the role a specific level of access. This is more efficient than managing privileges for dozens or hundreds of individual user ids.

SAS has options for initializing and passing Read, Write, and Alter passwords. READ= and WRITE= passwords apply to the underlying physical table and prevent it from being read or updated via a view. They don't however prevent the view itself from being replaced. For both a view and a table it's the ALTER= password that protects it from being modified, deleted, or replaced.

In this example, we create a view from an Oracle table that contains sensitive demographic data such as age, gender, race, and religion. To prevent access to those fields, we simply do not select them when we create the view. Not selecting fields for a view does nothing to affect their availability in the underlying physical table or in other views.

Here is basic syntax for creating a view with a SAS read password.

```
libname orcDB oracle path='orcl4' schema='test_hm' user=tarzan pass='Cheetah#1';
libname mrkting '/home/marketing/mtg/data';

proc sql;
  create view mrkting.v_marketing_demogs(read=eyeball)
     select customer_id, <list of non-sensitive fields>
     from orcDB.demogs
     where <selection criteria>;
quit;
```

This query reads the table we just created. It pulls date from the Oracle table orcDB.demogs. Without the read password, we'd get an error—`ERROR: Invalid or missing READ password`. If we gave the correct SAS password but on the Oracle side didn't have read privileges we'd get an error saying "Ora`-00942 table or view does not exist`."

```
proc sql;
     select customer_id, <list of fields>
     from mrkting.v_marketing_demogs(read=eyeball);
quit;
```

If we don't assign SAS-level passwords when a view is created, then afterwards we can use PROC DATASETS to add them. This example adds write and alter to prevent unauthorized changes.

```
proc datasets library=customer memtype=view;
  modify orcDB.v_marketing_demogs(write=pencil alter=eraser);
run;
```

MASK PASSWORDS

Passwords are often a big issue among SAS users. Not the password itself, although changing it every 30 or 60 days can be troublesome, but most corporate IT departments I think have a policy against hardcoding passwords into programs.

PROC PWENCODE or SAS macro variables go a long way to solving this particular problem. Paraphrasing the SAS documentation, PWENCODE uses encoding to *disguise* the password. During encoding, characters are translated on another character or set of characters using a unique lookup table. Decoding changes the word back to its original value. Encryption on the other hand actually transforms the data value of the password using mathematical operations and usually a "key" value. Decrypting returns the word to its original value. Encoding is obviously a weaker form of security but serves for what we need. Here is an example PROC PWENCODE and the encoded string it creates.

```
proc pwencode in='Cheetah#1' out='/users/apps/shared_code/cust_db_cd.std';
run;

{sas002}bXkgc51GF9wAzc3dvcmQ=
```

Each encoded password string is stored in its own text file designated by the out= option. So if you have three passwords you must use three PROC PWENCODE's each with a different out= value. To use the value, you read in the text file and assign the encoded value to a macro variable. Three passwords require three DATA _NULL_ statements. Here is one.

```
data _null_;
   infile '/users/apps/shared_code/cust_db_cd.std' obs=1 length=l;
   input @;
   input @1 line $varying1024. l;
   call symput('_dbPswd',substr(line,1,l));
run;

libname orcDB oracle path='orcl4' schema='test_hm'
                     user=tarzan pass="&_dbPswd" validvarname=any;
```

The obvious downside here is that the code for PROC PWENCODE and the DATA _NULL_ must reside somewhere. A personal autoexec.sas might be a good place for the DATA _NULL_, but then only that person can run the programs that use the passwords. If you run routines using a shared production account then having it in the

autoexec.sas solves that problem but then the file location is still available. But hopefully not too many people go poking around in the shared autoexec.sas.

One solution might be to save the PROC PWENCODE code in an innocuously named text file stored in a shared directory so it can be run each time the password must be updated. The DATA _NULL_ could be in a similarly named text file that is %INCLUDED by the shared autoexec.sas with the NOSOURCE2 option. Although the files would need UNIX group read permissions because they're shared, an Access Control List (ACL) lets you control file-specific permissions within the group. You'll probably have to get your Sys Admin involved but it's very doable and might be more common than you think. The References section has a link to information on ACLs.

This is only one technique for hiding passwords. This author used compiled data sets and there are quite a few others that can be found in SAS Conference papers.

CONCLUSION

SAS and Oracle are a powerful combination. We've covered here some of the SAS Options and techniques that improve their performance and minimize problems. Options really can make a difference, even simple ones like adjusting a buffer size and using bulk loading into Oracle. SASTRACE is a requirement for seeing what the database is doing and troubleshooting perceived SAS-Oracle performance problems. There are many other options and you really should flip thru the documentation specific to your version of SAS.

REFERENCES, RESOURCES, AND RECOMMENDED READING

- IBM. Access Control Lists. Available at http://www.ibm.com/developerworks/aix/library/au-acl/index.html?cmp=dw&cpb=dwaix&ct=dwnew&cr=dwnen&ccy=zz&csr=05121friend
- Lafler, Kirk Paul. 2004. *Proc SQL: Beyond the Basics Using SAS®*. Chapter 8. Cary NC: SAS Institute.
- Levin, Lois. 2004. *"Methods of Storing SAS® Data into Oracle Tables."* *Proceedings of the SAS Users Group International Conference 29*. Cary NC: SAS Institute. (PROC DBLOAD and the BULKLOAD option)
- Oracle Corporation. Reserved Words. Available at http://docs.oracle.com/cd/B19306_01/em.102/b40103/app_oracle_reserved_words.htm
- Plemmons, Howard. 2010. "What's New in SAS/Access." *Proceedings of the SAS Global 2010 Conference*. Cary NC: SAS Institute.
- Plemmons, Howard. 2009. "Top Ten SAS DBMS Performance Boosters for 2009." *Proceedings of the SAS Global 2009 Conference*. Cary NC: SAS Institute.
- Qai, Jun. 2009. "How to Leverage Oracle Bulk-load for Performance Contact Information." *Proceedings of the SAS Global 2009 Conference*. Cary NC: SAS Institute. (Bulk loading using SAS MP/CONNECT®.)
- Rhoads, Mike. 2009. *"*Avoiding Common Traps When Accessing RDBMS Data.*"* *Proceedings of the SAS Global 2009 Conference*. Cary NC: SAS Institute.
- Rhodes, Diane. 2007. "Talking to Your RDBMS Using SAS/Access®*."* *Proceedings of the SAS Global 2007 Conference*. Cary NC: SAS Institute.
- Somerville, Claire and Clive Cooper. 1998. *"*Optimizing SAS® Access to an Oracle Database in a Large Data Warehouse*."* *Proceedings of the SAS Users Group International Conference 29*. Cary NC: SAS Institute. (Older but valid examples of effectively using DATA Step, Procs, and Indexes.)
- SAS Institute. 2008. *SAS/Access Supplement for Oracle*. Cary NC: SAS Institute. (Be sure to use the documentation specific to your version of SAS. An RDBMS-specific supplement is available for each system SAS supports.)
- SAS Institute. 2008. *SAS(R) 9.2 Language Reference: Dictionary*. Cary NC: SAS Institute.
- SAS Institute. Connecting SAS to Oracle. Available at http://support.sas.com/techsup/technote/ts703.pdf
- SAS Institute. Reserved Words. Available at http://support.sas.com/documentation/cdl/en/mcrolref/62978/HTML/default/viewer.htm#p0y43hj7lzhq1gn1r68h65wzljbt.htm
- SAS Institute. PROC SQL Options. Available at http://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a002473669.htm

If you run into a problem Mike Rhoads paper would be a good one to start with. His References section provides a very good list of pre-2009 SAS Conference papers.

ACKNOWLEDGMENTS

I certainly didn't figure out all this on my own. I'm extremely grateful to the technical writers and web developers at SAS Institute for the online documentation, the patient people at the SAS Help Desk who answer my questions, SAS users who give up their precious time to write papers and present at SAS Conferences, and the folks I work with who share their wide-ranging knowledge. Thanks also to my enlightened bosses over the years who had the vision and foresight to send me to SAS training and conferences. Thanks everybody, I couldn't have done this without you.

CONTACT INFORMATION

John E. Bentley
Wells Fargo Bank
Charlotte NC 28226
John.bentley@wellsfargo.com