Paper 099-2009

How to Leverage Oracle Bulk-load for Performance

Jun Cai, Qwest Communications International Inc., Denver, CO

ABSTRACT

Often times, we need to load a large volume of SAS® data into an Oracle database. Bulk-load through the SAS/ACCESS® interface would be the first choice in many cases, but there's a cost associated with dumping SAS data to a text file to accommodate SQL*Loader. SAS V9 further boosts the bulk load by leveraging native Oracle features on load. However, it is crucial to understand how and when to use those functions in order to optimize load for performance. Inappropriate uses of those options may not produce a desired result. Instead, it might jeopardize your load. This paper is intended to explore some of these new features and to discuss the various situations that should be taken into consideration when applying those options in Oracle bulk load. Sample code will be presented to elaborate the benefits gained by appropriate uses of those options, including OR_PARTITION, BL_INDEX_OPTIONS, BL_RECOVERABLE, BL_SUPPRESS_NULLIF and BL_OPTIONS.

This paper is written for SAS developers who possess a basic understanding of SAS/ACCESS and Oracle SQL*Loader.

INTRODUCTION

SQL*Loader is an Oracle utility that provides high performance for data loads. It loads data into an Oracle table from an existing text file, instructed by the control file in which metadata describes how the data will be loaded, including the table name, column data types, formatting, transformation, and so on. In addition, the specified parameters including pre-generated in the PARFILE and from comment line would guild the load as well. As you might know, SQL*Loader is typically used by Oracle professionals to load large volume of data into Oracle databases. Unlike conventional SQL insert, SQL*Loader speeds up the load significantly by building blocks of data in memory and saves these blocks directly into the extents allocated for the table being loaded. Redo log entries are not generated unless the database is in archive log mode (setting the table being loaded to nologging can also reduce redo log entries if the database is in archive log mode). Direct path loads use the field specifications to build whole Oracle blocks of data, and write the blocks directly to the Oracle data files. Direct path load bypasses the database buffer cache and accesses the SGA only for extent management and adjustments of the high-water mark.

Now that it is recently integrated with SAS, SQL*Loader may be launched from within SAS/ACCESS. When the option 'BULKLOAD' is set to 'YES" in SAS/ACCESS, SQL*Loader will be invoked with direct path. Please note that BULKLOAD=YES may have different meaning on different platforms of DBMS. In this paper, we are interested only in discussing Oracle SQL*Loader.

SQL*Loader configuration options are therefore wrapped through SAS/ACCESS interface. You are required to specify SQL*Loader options in SAS code. In order for you to make the best use of Oracle bulk load, you need to possess a good understanding of SQL*Loader as well as its implementation in SAS/ACCESS. As far as I know, there is not a good document provided by SAS to explain the two aspects of Oracle bulk load from a SAS user perspective. This is the very reason why this paper exists for SAS users who are interested in using Oracle bulk load.

For the demonstration purpose, a SAS table sales.sales_activity is created with 239 columns, including a column named as sales_date. The table is populated with 2.1 million rows that may cross (the?) most current 13 months of sales_date. The SAS table is about 2 GB in size. Further, an Oracle table partitioned by sales date is created with the same columns, named as ora_sales.sales_activity. A few indexes are also created on the Oracle table for the demo. All of configurations for Oracle bulk load are benchmarked with the testing data and results will be included at the end of the paper.

The following sample SAS code shows how to invoke SQL*Loader with default settings, assuming that your Oracle database is connected properly.

```
libname sales 'SAS-Data-Library';
libname ora_sales oracle user=xxxxx password=xxxxx path='xxxx' schema=xxxx;

proc append base = ora_sales.sales_activity(BULKLOAD=YES)
            data = sales.sales_activity;
run;
```

When this piece of code runs, the following files will be created: bl_sales_activity.dat (a text file containing data to be loaded), bl_sales_activity.ctl(the control file containing instructions for load), bl_sales_activity.log, bl_sales_activity.bad (if there are any bad data) and bl_sales_activity.dsc (if there are any discarded data). By default, these files will be generated in the current directory, unless the location is otherwise specified. As you might know already, the data file and the control file are generated for you by SAS so that you are able to use the control file without needing to understand the details. When the two files are ready for use, SAS/ACCESS launches SQL*Loader to perform bulk load. The Oracle table will be eventually loaded with SAS data if nothing goes wrong and status may be examined from the log file. Beauty of SAS/ACCESS Oracle bulk load is all of the internal phases are controlled by SAS/ACCESS engine and they are completely transparent to SAS users.

Oracle SQL*Loader came with various options that were not visible in the older versions of SAS. The SAS V9 has enhanced Oracle bulk load by further leveraging the native SQL*Loader functionality, which is now made available through SAS/ACCESS configuration options. Understanding of those functions and having appropriate settings for them are truly critical to run bulk load efficiently and effectively.

In this paper, the following functions will be discussed:

- Load data directly into Oracle table partition
- Load data in parallel
- Build indexes while loading data
- Control recovery mode
- Deal with Nulls

LOAD DATA DIRECTLY INTO ORACLE TABLE PARTITIONS

Large scale data in an Oracle table is usually partitioned based on a selected partition key. When data to be loaded all fits into a particular partition, it is best to direct SQL*Loader to move data to the given partition only, rather than the entire table. This can be achieved by specifying the partition name in the following form:

OR_PARTITION =NAME OF A PARTITION IN A PARTITIONED ORACLE TABLE

In our example, the ora_sale.sales_activity table is partitioned by month as follows:

```
SALES05_10P (MONTH IN OCT05)
SALES05_11P (MONTH IN NOV05)
SALES05_12P (MONTH IN DEC05)
SALES06_01P (MONTH IN JAN06)
SALES06_02P (MONTH IN FEB06)
SALES06_03P (MONTH IN MAR06)
SALES06_04P (MONTH IN APR05)
SALES06_05P (MONTH IN MAY06)
SALES06_06P (MONTH IN JUN06)
SALESACTV_P (MONTH GE TODAY)
```

The following sample code adds rows only to the partition SALES05_10P of ora_sale.sales_activity table.

```
libname sales 'SAS-Data-Library';
libname ora_sales oracle user=xxxxx password=xxxxx path='xxxx' schema=xxxx;

proc append base= ora_sales.sales_activity
                      (OR_PARTITION=SALES05_10P
                       BULKLOAD=YES
                      )
           data= sales.sales_activity;
       where intnx('month',data,0) = '01OCT2005'd;
run;
```

The load is expected to run much faster with the specified partition than if it is a direct load into the table. Without the specified partition name, SQL*Loader would have to check the partition key at a row level to determine where to place data in the table, even though all of the rows extracted from the SAS table have the same date range. Therefore, the key is to have good understanding and control of your source data before load. As you may expect, the load will be slower if your source data has wide date range versus having the specified partition in the target table, due to rejection of records.

LOAD DATA IN PARALLEL

There might be also situations in which multiple partitions need to be loaded at the same time. To further speed up the load, you may instruct bulk load to run in parallel in the following way:

BL_OPTIONS = 'PARALLEL=TRUE'.

Thus, load will be submitted as multiple tasks that run multiple SQL*Loader jobs concurrently. The sample code below shows how to achieve parallel partition load:

```
signon task01 sascmd = '!sascmd';
signon task02 sascmd = '!sascmd';
signon task03 sascmd = '!sascmd';

rsubmit process=task01 wait=no;
 libname sales 'SAS-Data-Library';
 libname ora_sales oracle username=xxxxx password=xxxxx path='xxxxx';

  proc append base=ora_sales.sales_activity
                  (OR_PARTITION=SALESACTV_P
                   BULKLOAD=YES
                   BL_OPTIONS='PARALLEL=TURE'
                   )
         data=sales.sales_activity;
     where sales_date >= today();
  run;
     libname ora_sales clear;
     libname sales clear;

endrsubmit;

rsubmit process=task02 wait=no;
  libname sales 'SAS-Data-Library';
  libname ora_sales oracle username=xxxxx password=xxxxx path='xxxxx';

  proc append base=ora_sales.sales_activity
                  (OR_PARTITION=SALES06_06P
                   BULKLOAD=YES
                   BL_OPTIONS='PARALLEL=TURE'
                   )
          data=sales.sales_activity;
     where sales_date between '01jun06'd and '30jun06'd;
  run;
     libname ora_sales clear;
     libname sales clear;

endrsubmit;

rsubmit process=task03 wait=no;
  libname sales 'SAS-Data-Library';
  libname ora_sales oracle username=xxxxx password=xxxxx path='xxxxx';

  proc append base=ora_sales.sales_activity
                         (OR_PARTITION=SALES06_05P
                          BULKLOAD=YES
                          BL_OPTIONS='PARALLEL=TURE'
                  )
       data=sales.sales_activity;
     where sales_date between '01may06'd and '31may06'd;
  run;
     libname ora_sales clear;
     libname sales clear;
endrsubmit;

waitfor _all_ task01 task02 task03;

rget task01;
rget task02;
rget task03;

signoff task01;
signoff task02;
signoff task03;
```

The three tasks will be submitted as a multi-threaded operation to Oracle. You will find three processes associated with the bulk load on the Oracle side. Thus, the system resources will be maximized to achieve the best performance.

It's important to know data distribution when considering use of parallel feature or function. When your data is evenly or fairly evenly distributed across partitions, you will get the maximum benefit from parallel load, as the time spent on the largest partition will be the time required to complete the multiple tasks as a whole. For example, if you had 200 records for `SALES06_05P`, 300 records for `SALES06_06P`, and 2 million records for `SALES06_P` in the example above, the load as a whole will run to completion when the 2 millions of rows are successfully loaded into the partition `SALES06_P`, although the other two tasks will finish much earlier than the first task. Apparently, you would not gain very much performance through parallel processing in this example, compared with sequential load.

Parallel load can also be used when loading data into a table or a single partition. In following example, 2.1 million rows from the source table first split evenly into three segments, each of which has roughly 700,000 rows. Then three tasks will be created to load three segments in parallel into the Oracle table. Obviously, this is a good use of parallel load in practice:

```
libname sales 'SAS-Data-Library';

data sales.sales_activity;
  set sales.sales_activity;
  load_seg=int(ranuni(0)*3)+1;
run;

signon task01 sascmd = '!sascmd';
signon task02 sascmd = '!sascmd';
signon task03 sascmd = '!sascmd';

rsubmit process=task01 wait=no;
 libname sales 'SAS-Data-Library';
 libname x oracle username=xxxxx password=xxxxx path='xxxxx';

 proc append base=ora_sales.sales_activity
                        (BULKLOAD=YES
                         BL_OPTIONS='PARALLEL=TURE'
                         )
             data=sales.sales_activity;
    where load_seg=1;
 run;
    libname ora_sales clear;
    libname sales clear;

endrsubmit;

rsubmit process=task02 wait=no;
 libname sales 'SAS-Data-Library';
 libname ora_sales oracle username=xxxxx password=xxxxx path='xxxxx';

 proc append base=ora_sales.sales_activity
                        (BULKLOAD=yes
                         BL_OPTIONS='PARALLEL=TURE'
                         )
             data=sales.sales_activity;
    where load_seg =2;
 run;
    libname ora_sales clear;
    libname sales clear;

endrsubmit;

rsubmit process=task03 wait=no;
 libname sales 'SAS-Data-Library';
 libname ora_sales oracle username=xxxxx password=xxxxx path='xxxxx';


 proc append base=ora_sales.sales_activity
                        (BULKLOAD=yes
                         BL_OPTIONS='PARALLEL=TURE'
                         )
```

```
            data=sales.sales_activity;
    where load_seg =3;
 run;
    libname ora_sales clear;
    libname sales clear;

endrsubmit;

waitfor _all_ task01 task02 task03;

rget task01;
rget task02;
rget task03;

signoff task01;
signoff task02;
signoff task03;
```

You might have a question at this point. How should you determine if you need to build multiple tasks for a parallel partition load, or you need to build multiple tasks for parallel segment load? Well, it is primarily based on your data. If your data is very well distributed across partitions, the first approach should be taken. However, if your data is not evenly distributed, like what was shown in our example, you would certainly consider parallel segment load as the second example in the section.

Please note that there are restrictions on Parallel Direct Path Loads enforced by Oracle.

- Indexes may not be maintained at load.
- Referential integrity and CHECK constraints must be disabled at load.
- Triggers must be disabled at load.
- Rows can be appended and inserted. REPLACE and TRUNCATE are not supported.
- When parallel applies to a single partition, you should partition the data first. Otherwise, there is overhead of record rejection due to a partition mismatch, which would slow down the load.

BUILD INDEXES WHILE LOADING DATA

Conventionally, indexes are dropped before the table is loaded. They will be then re-created after the load. The reason: performance! Loading data at the same time when building indexes is generally slower than separating the two tasks sequentially. This is especially true with older versions of SQL*Loader. However, there are scenarios in which loading data and building indexes at the same time are actually more effective using some advanced technologies built in the latest version of SQL*Loader. This is especially true when your data is prepared in a way SQL*Loader can take advantage of the data preparation.

For example, if your source table is sorted already by indexed columns in the target table prior to load, you can instruct the load to build indexes while loading data without an additional sort. As a result, indexes will be built much faster and the entire load will be completed in a shorter time. The key here is the phrase ‘is sorted already’. The idea is to take advantages of your source data quality and characteristics. If your data is not sorted prior to load, it will make little sense to sort your data for this purpose prior to load.

This may be achieved by specifying the following option in your code:

BL_INDEX_OPTIONS=SORTED INDEXES(SEGMENT NAME)

The following sample code shows how to specify the BL_INDEX_OPTION:

```
libname sales 'SAS-Data-Library';
libname ora_sales oracle username=xxxxx password=xxxxx path=xxxxx;

/* truncate oracle table and create index */

proc sql noprint;
       connect to oracle (username=xxxx password=xxxx path=xxxx);
       execute (drop index CUSTOMER_ID) by oracle;
       execute (truncate table SALES_ACTIVITY) by oracle;
       execute (create UNIQUE index CUSTOMER_ID on SALES_ACTIVITY
                     (CUSTNAME, STATE, WTN, ORDERNUM, SALES_DATE)) by oracle;
       disconnect from oracle;
quit;
```

```
   /* append data to oracle table, incoming data have to be sorted by the index fields.
*/
   proc append base= ora_sales.sales_activity
                          (BULKLOAD=YES
                           BL_INDEX_OPTIONS='SORTED INDEXES(CUSTOMER_ID)'
                           )
                 data= sales.sales_activity;
   run;
```

As you might have noticed already, BULKLOAD needs to be set to 'YES' for taking advantage of the indexing option because this is a SQL*Loader function

There may be also situations in which data is not sorted and the assigned addressed memory for load is limited. In this case, you might want to consider the other option for indexing.

BL_INDEX_OPTIONS= SINGLEROW

With this option, load will insert index entry directly into the index, one record at a time, with limited memory required for sort. Note that it is not a good idea to perform single row indexing if you have large volume of data to load.

Following is the example of usage for single row index option:

```
   proc append base= ora_sales.sales_activity
                          (BULKLOAD=YES
                           BL_INDEX_OPTIONS=SINGLEROW
                           )
                 data= sales.sales_activity;
   run;
```

There are tips on "SORTED INDEXES" I would like to share based on my experience.

1) This option can be used only when the Oracle table has existing indexes and the indexed field in the source data is sorted. This option is not intended to help you with creating new indexes on the loaded table. If there are no desired indexes on the target table, you need to create the indexes on the target table first. The following example shows you a way to quickly sort the source data for fulfilling the requirement.

```
   libname sales 'SAS-Data-Library';

   / * Step 1: create a table */
   proc sql;
         create table sales.sales_activity like sales.sales_activity_orginal;
   quit;

   / * Step 2: create index */
   proc datasets lib=sales nolist;
         modify sales_activity;
         index create CUSTOMER_KEY=(CUSTNAME STATE WTN ORDERNUM SALES_DATE);
   run;

   / * Step 3: Using append to indexed table, it takes only 5 minuets and 07 seconds to complete*/
   proc append base = sales.sales_activity
               data = sales.sales_activity_original;
   run;
```

The idea is to sort the source data by creating indexed table. This is a convenient way to sort your source data efficiently and effectively.

2) Parallel load is not valid when the index option is selected, and vice versa. Otherwise, the following error will appear:

```
   SQL*Loader-951: Error calling once/load initialization

   ORA-26002: Table ORA_SALES.SALES_ACTIVIY has index defined upon it.
```

The solution is to select either BL_OPTION = 'PARALLEL=TRUE'' or BL_INDEX_OPTION = 'SORTED INDEX (INDEX)'. Not both.

3) If the index is unique and served as the primary key, specifying 'SORTED INDEXES' will invalidate the unique index even though the load can be completed. The SQL*Loader message would look like:

```
The following index(es) on table SALES_ACTIVITY were processed:

index ORA_SALES.BL_IDX_OPT was made unusable due to:

ORA-01409: NOSORT option may not be used; rows are not in ascending
order
```

That means that the SQL*Loader has to use its own sort to build unique indexes in order to guarantee that data integrity is maintained, regardless of your direction. If you happen to run into the situation, you will have to re-create the unique index.

4) When performing large bulk loads, the best practice is to drop all of indexes, and to load data into the table, then to recreate indexes. This is because loading using direct path with no indexes in most cases provides the best performance, as you can run the load in parallel. As you have seen previously, you cannot run parallel load with building indexes at the same time. After the data has been loaded, you can take advantage of Oracle options such as parallel dml, parallel ddl, and no logging to build indexes using sqlplus or other GUI tools.

5) The unique index for the primary key can not be dropped or following error condition will be generated otherwise:

```
ORACLE execute error:
ORA-02429: cannot drop index used for enforcement of unique/primary key
```

The correct action is to drop the primary key constraint, which automatically drops the unique index for you.

```
proc sql noprint;
 connect to oracle (username=xxxx password=xxxxx path=xxxxx);
      execute (alter table SALES_ACTIVITY
      drop constraint CUSTOMER_KEY) by oracle;
 disconnect from oracle;
quit;
```

CONTROL RECOVERY MODE

I found that many data loads were performed in a recoverable mode, but there was actually no need for that if the source file can be reused when there is a need to re-load data, or the loaded table is backed up somehow. You certainly want to use non-recoverable mode to speed up the load, as logging transactions for recovery is expensive in terms of both space and performance. Running a load with non-recoverable mode, the space use for the transaction log will be significantly reduced and both CPU and memory usage will also be reduced. Both of them will eventually translate into performance gains.

This option may be set as the following:

BL_RECOVERABLE=YES|NO

When this option is set to 'YES', the load process is recoverable. When this option is set to 'NO', the load process is not recoverable. When 'NO' is specified, the keyword 'UNRECOVERABLE' will be automatically added into the SQL*Loader control file before the load.

The sample code below demonstrates the use of BL_RECOVERABLE=NO.

```
libname sales 'SAS-Data-Library';
libname ora_sales oracle user=xxxxx password=xxxxx path='xxxx' schema=xxxx;

proc append base=ora_sales.sales_activity
                        (BULKLOAD=YES
                         BL_RECOERABLE=NO
                         )
             data=sales.sales_activity;
run;
```

After the load completes and before another hot or cold backup is taken, the Oracle database could not recover to the point of the successful load completion, because we did not generate any redo logs. This is why it is important to back up the loaded table right after a successful load. Or the same source file exists for re-load.

One more thing to note about recovery mode is the Oracle table definition. If the Oracle table is already defined for nologging, there is no need for specifying BL_RECOVERABLE=NO since they both basically say the same thing. As

a matter of fact, the following message will be generated if you have both BL_RECOVERABLE=NO and a nologging table.

"Load is UNRECOVERABLE; Invalidation redo is produced."

DEAL WITH NULLS

As we all know, any of functions applying to data is expensive and this is especially true when it applies to data at a row level. NULLIF is one of them. The newly introduced feature for bulk-load allows you to suppress NULLIF clause for the specified columns, if possible, in order to improve performance. The syntax looks like the following:

BL_SUPPRESS_NULLIF=<_ALL_=YES | NO >|(<*COLUMN-NAME-1*=YES | NO > <*COLUMN-NAME-N*=YES | NO >...)

To suppress NULLIF clause for the specified column in the table, column-name should be set to "YES". To not suppress NULLIF clause for the specified column in the table, column-name should be set to "NO". To specify all columns to be suppressed or not suppressed, _ALL_ should be used with "YES" or "NO" respectively.

The following example uses the BL_SUPPRESS_NULLIF= option in the DATA step to suppress the NULLIF for the column product1 and the column product2.

```
libname sales 'SAS-Data-Library';
libname ora_sales oracle user=xxxx pw=xxxxx path='xxxx';

data ora_sales.sales_activity
                        (BULKLOAD=YES
                         BL_SUPPRESS_NULLIF=(PRODUCT1=YES PRODUCT2=YES)
                         );
    set sales.sales_activity;
run;
```

The following example uses the BL_SUPPRESS_NULLIF= option in the DATA step to suppress the NULLIF for all of the columns in the given table.

```
libname sales 'SAS-Data-Library';
libname ora_sales oracle user=xxxx pw=xxxx path='xxxx';

data ora_sales.sales_activity
                        (BULKLOAD=YES
                         BL_SUPPRESS_NULLIF=(_ALL_=YES)
                         );
    set sales.sales_activity;
run;
```

My testing shows that setting BL_SUPPRESS_NULLIF to 'YES' in the first example above saves about 6% of the time in overall load, as the values of the two columns have nulls. However, there are no noticeable performance gains from the second example because the last column in the table has no NULL. Therefore, whether or not to suppress NULLIF is determined by your data. As stated previously in this paper, understanding of your data is always important when making decisions on configurations.

One more thing to note: BL_SUPPRESS_NULLIF=(column1=YES column1=NO) get evaluated from left to right, so if you explicitly or implicitly specify a column name twice, as you see in this example, NO will override YES and consequently NULLIF will not be suppressed in this example.

SOME IMPORTANT NOTES

It is worth noting the cost of data file generation. First, dumping SAS data to a file is expensive in terms of time. In my example, it took about 20 minutes to create the data file, which is roughly one third of the total time spent on the load as a whole. Secondly, as much as three times the SAS table in size is needed to temporarily store the data for load. In this example, I preserved 6GB because the SAS table is about 2GB. With a very large database, this requirement could become a showstopper simply because of space constraint.

The data file will be deleted by default when load is completed. However, BL_DELETE_DATAFILE=NO could be specified in the code to keep the data file. There might be situations in which it is desirable to save the data file for other uses. A situation I ran into quite often was I had to re-run the load due to failure of load for any reason. With the saved data file, I didn't have to start it over again to wait for another 20 minutes in recreating the data file. Instead, I opt to run SQL*Loader out of SAS/ACCESS. For the time being, you cannot re-run SAS/ACCESS with the saved data file and a data file will be re-created anyway. Hopefully we will see this type of SAS enhancement in the near future.

Oracle SQL*Loader may fail for reasons. You need to come up with algorithms to clean up the failed load. The possible solutions include remembering rows just loaded by timestamps and/or number of rows. Remove those newly loaded rows before re-run the load or direct load to skip the first x number of rows in the re-run. Things are getting simpler if your load data into an empty table. In this case, what you can do is to truncate the Oracle table if the table is partially loaded and to re-run the load after the cause of the failure is addressed promptly. Resolving the cause of the failure is beyond the scope of the paper and common causes are data related, including mismatches of your control file with your actual data in the source table and duplicates.

It is crucial to understand how the native DBMS technology works when choosing to use it. Unfortunately, SAS does not provide detailed information helping SAS users. It might be assumed that that would be out of scope from a SAS perspective, and a DBMS manual should be referenced when questions rise. However, this is not a helpful assumption to many SAS developers. Like many other SAS developers, the author of this paper has experienced such frustration when many of functions are not well documented (or documented at all) by SAS. I hope that my experience would be helpful to those who are experiencing some level of frustration.

CONCLUSION

The performance concern with loading large-scaled data across databases has been around ever since. Before SAS introduced support on native DBMS Bulk-Load, there was no easy way to move data efficiently between SAS and DBMS such as Oracle. Not only does SAS/ACCESS make data movement across databases much easier from a development perspective, it also makes data movement across databases more efficient and effective by taking advantages of native DBMS technologies for bulk load.

At the end of the paper, I would like to share some benchmarks I have had. In general, 43% reduction of execution time proves that implementing SAS bulk-load with the native Oracle SQL*Loader functions which are available via SAS/ACCESS are the best choice when it is used properly.

The following table has matrices of configuration options and the resulted performance.

Load Option	**SQL*Loader Loading Elapsed Time (mm:ss.00)**	**Total Elapsed Time(dump file + load) (mm:ss.00)**	**Time Reduction**
BULKLOAD=YES	37:39.86	53:18.00	-
BULKLOAD=YES BL_RECOVERABLE=NO	35:26.57	53:10.42	
BULKLOAD=YES BL_INDEX_OPTIONS='SORTED INDEXES(CUSTOMER_KEY)	30:02.22	49:05.98	8%
BULKLOAD=YES BL_OPTIONS='PARALLEL=TRUE'(six processes by load_seg in parallel)	09:52.78(seg1) 10:25.91(seg2) 09:38.58(seg3) 09:37.97(seg4) 09:35.44(seg5) 10:23.93(seg6)	36:11.00	32%
BULKLOAD=YES BL_OPTIONS='PARALLEL=TRUE' BL_SUPPRESS_NULLIF=(PRODUCT 1=YES PRODUCT=YES) (six processes by load_seg in parallel)	09:35.69(seg1) 09:39.44(seg2) 09:40.67(seg3) 09:34.70(seg4) 08:37.82(seg5) 08:31.91(seg6)	30:44.27	43%

BULKLOAD=YES BL_OPTIONS='PARALLEL=TRUE' OR_PARTITION=SALESACTV_P OR_PARTITION=SALES05_10P OR_PARTITION=SALES06_03P OR_PARTITION=SALES06_04P OR_PARTITION=SALES06_05P OR_PARTITION=SALES06_06P	30:41.60(actv_p) 00:01.53(05_10p) 00:02.07(06_03p) 00:01.00(06_04p) 00:00.86(06_05p) 00:00.88(06_06p)	53:55.32	-26%

Note: The testing was not conducted in a controlled environment, so the results may not be totally accurate. However, it would give you an idea that appropriate use of the configuration options does make difference for performance.

REFERENCES

Dave Moore, "SQL*Loader", http://www.oracleutilities.com/OSUtil/sqlldr.html

SAS V9 Online documents, http://support.sas.com/onlinedoc/913/docMainpage.jsp

ACKNOWLEDGMENTS

I would like to thank Dave Carter, an Oracle database DBA, for providing the SQL*Loader and Oracle database educations, in addition to sharing his experiences on tuning Oracle bulk load for performance. My thanks also go to Heather Tavel for helping with development of sample code, and to Qiang Hou, John Xu, William Lu and Mark Xu, members of Chinese SAS Users Group for sharing their thoughts and experiences when I raised bulk load performance issues. Finally, thanks to Kirk Anderson in add of review. This paper would not be possible without their valuable assistance.

CONTACT INFORMATION

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

Jun Cai
Qwest Communications International Inc.
931 14th Street, 13th floor
Denver, CO 80202
Work Phone: 303 624 2468
Fax:
Email: jun.cai@qwest.com
Web: