Tuning the Oracle Connector performance in IBM InfoSphere DataStage, Part 1: Overview of the connector tuning process and optimization of fetch, lookup and DML operations

Predrag Maksimovic

March 28, 2013

The Oracle Connector is a connectivity component in IBM® InfoSphere® Information Server. It is utilized by IBM InfoSphere DataStage® and other products in the Information Server suite to perform extract, lookup, load, and metadata import operations on Oracle databases. This article is Part 1 of a series of 2 articles that provide a set of guidelines for tuning the Oracle Connector stages in DataStage parallel jobs with the goal of maximizing their performance.

View more content in this series

Introduction

IBM InfoSphere Information Server is a highly scalable software platform for information integration. It is organized as a suite of products built on top of common metadata and services layers. Working in unison, the Information Server products provide a means to understand, cleanse, transform, and deliver trusted information across the enterprise.

IBM InfoSphere DataStage is a member of this suite and provides extract, transform, and load (ETL) functionality. DataStage users create jobs, which are collections of stages connected to one another via links. The links transport data records between the stages, and the stages perform operations on the records. Each stage in a job is of a particular stage type.

Information Server comes pre-configured with a number of built-in stage types. Generally, stage types can be classified into processing and connectivity stage types. Processing stage types are used to perform transformations on records, and connectivity stage types are used to extract records from external data sources so that they can be processed by the job, or to load records processed by the job to the external data sources. Examples of external data sources include relational databases, enterprise messaging systems, rule management systems, enterprise applications, and file systems.

The Oracle Connector allows DataStage to interact with the Oracle database by leveraging native, high-performing methods. It is used in DataStage jobs to extract records from Oracle database tables so that they can be processed by the job or to deliver records processed by the job to
the Oracle database tables. **Figure 1** shows a simple DataStage job with an Oracle Connector stage (my_ora_fetch) configured to fetch rows from an Oracle database. The fetched records are processed by a Transformer stage (my_transformer) and two identical copies of the processed records are made by a Copy stage (my_copy). One copy of the records is loaded to a DB2 database by a DB2 Connector stage (my_db2_load), and the second copy is loaded to an Oracle database by another Oracle Connector stage (my_ora_load).

**Figure 1. A simple DataStage job with Oracle connector stages**

IBM InfoSphere DataStage leverages several mechanisms to provide high-performance and simply-configured scalability. This article provides information about how those methods work in regard to accessing data in the Oracle database, and how to tune those features in the Oracle Connector to meet specific performance objectives for your project.

The article assumes Information Server Version 9.1. Although most of the included material applies also to previous versions of Information Server, certain concepts such as support for batch processing of anonymous PL/SQL blocks are applicable only to Version 9.1.

The article also assumes the use of DataStage parallel jobs, although some concepts would also apply to DataStage server jobs.

**Parallelism and partitioning**

The terms *parallel* and *partitioned* are referenced throughout this article. This section explains how these two terms apply to DataStage jobs and an Oracle database, providing a baseline of understanding so that you can effectively apply the guidelines in this article.

DataStage runs data integration tasks using two types of parallelism: pipelining and partitioning.

When a DataStage job starts, a single operating system process called the *conductor process* is created for the job and is used to initialize all the stages and links in the job. After the setup phase of the job is completed, the processing phase of the job begins during which the records move through the job and are processed by the stages. The operating system processes created for the stages, and in which the stages are processing records, are called *player processes*.

Parallelism in DataStage is accomplished by running the player processes simultaneously using pipelining and partitioning.
• Pipelining means that player processes for different stages in the job can run simultaneously in an assembly line fashion, passing sets of records downstream through the job.
• Partitioning means that player processes for a single stage can run simultaneously and process distinct subsets of records for that stage.

Pipelining and partitioning are not mutually exclusive. The jobs typically utilize both of them.

An Oracle Connector stage can be configured to run in parallel or sequential execution mode. The Execution mode setting on the Advanced page in the connector stage editor is used to specify this configuration.

When an Oracle Connector stage is running in sequential execution mode, a single player process is created for the stage to fetch records from the database, or to write records to the database, depending on the role that the stage has in the job.

After the player process of the Oracle Connector stage fetches and delivers some records on its output link, the player processes of the next stage in the job will begin processing those records while the Oracle Connector stage player process will simultaneously proceed to fetch more records from the database. The processing of records in pipeline fashion is the default behavior of DataStage. Even if the Oracle Connector stage is running sequentially in a single player processes, player processes of the remaining stages in the job can run simultaneously.

When an Oracle Connector stage is running in parallel execution mode, the partitioning type of parallelism takes place for the stage, so that multiple player processes of that stage are fetching records from the database or writing records to the database simultaneously.

The number of player processes and the locations of the hosts on which they run are specified in the parallel engine configuration file and can further be customized for individual stages on the Advanced page in the connector stage editor. The location of the parallel engine configuration file used by the jobs is specified in the APT_CONFIG_FILE DataStage environment variable.

Let us now take a closer look at an Oracle Connector stage configured to write records to the database. It will have an input link attached to it and the records will be arriving to the stage on that link. The entire set of the records arriving on the link is referred to as the record data set. If this stage is configured to run in parallel execution mode then the records from the input data set will be distributed at runtime by the DataStage engine framework to the player processes of the stage. Each player process will receive a subset (called partition) of records from the input data set and will send it to the database. The algorithm that is used to decide which input record goes to which player process is specified in the Partition type setting on the Partitioning page for the input link in the connector stage editor.

Figure 2 illustrates Oracle connector stage configured to write records to the database in parallel execution mode. Three player processes are created for the stage in this example. The letters P and R in the figure represent player processes and records, respectively. This diagram implies that the upstream stage is running sequentially and that the records are arriving on the link in a single partition. In practice, the upstream stage could also be running in parallel with the records arriving
on the input link in multiple partitions. In that case, depending on the number of the partitions, the number of the player processes created for the Oracle Connector stage and the specified Partition type value for the Oracle Connector stage, the partitions would either be preserved or reorganized into new partitions before being delivered to the player processes of the Oracle Connector stage.

**Figure 2. Oracle connector stage writing records in parallel execution mode**

Let us consider next an Oracle Connector stage configured to fetch records from the database. This stage will have an output link. The rows that the stage fetches from the database will represent the record data set for the output link. If the stage is configured to run in parallel execution mode, this configuration will be changed automatically at runtime to sequential execution mode in order to ensure that multiple player processes of the stage do not end up fetching the same rows from the table. This is what would happen if all the player processes ran the exact same `SELECT` statement specified for the stage. However, if you set the Enable partitioned reads connector property to Yes, then the sequential execution mode will not be enforced any more. Instead, the connector will automatically modify the specified `SELECT` statement by appending a unique `WHERE` clause to it for each player process so that each player process fetches a distinct subset of rows from the database.

Each of the subsets will represent a partition of records provided by the respective player process. When the partitions from all player processes are collected to produce the complete record data set for the output link, this data set will contain the same records that it would have contained had the stage been configured to run in sequential execution mode and execute the original `SELECT` statement in a single player process. The algorithm that the connector uses to determine how to modify the `SELECT` statement for each player process when running in parallel execution mode is specified in the Partitioned reads method connector property.

**Figure 3** illustrates Oracle connector stage configured to fetch records from the database in parallel execution mode and with Enable partitioned reads connector property set to Yes. Three player processes are created for the stage in this example. This diagram implies that the downstream stage is running sequentially and that the records on the output link are joined and delivered to the downstream stage in a single partition. In practice, the downstream stage could also be running in parallel in which case depending on the number of partitions produced by the Oracle Connector stage, the number of player processes for the downstream stage and the Partition type value in the downstream stage, the partitions would either be preserved or
reorganized into new partitions before being delivered to the player processes of the downstream stage.

**Figure 3. Oracle connector stage fetching records in parallel execution mode**

Finally, let us consider an Oracle Connector stage configured to perform sparse lookup operation on the database. This stage will have a reference link leading to a Lookup stage. The *Lookup type* property in the Oracle Connector stage will be set to value *Sparse*. The records arriving on the input link of the Lookup stage will be passed to the Oracle Connector stage on the reference link. The Oracle Connector stage will use the values from these records to initialize parameters of the *WHERE* clause of the parameterized *SELECT* statement that it will then use to perform lookup operation on the database.

Note that the *SELECT* statement with the *WHERE* clause will be prepared only once in each player process as a dynamic statement with bind parameters, but it will then be executed for each input lookup record to fetch the matching records from the database for that record. The records fetched in each lookup call will be passed back to the Lookup stage on the reference link and from there to the output link of the Lookup stage. If the Oracle Connector stage is configured to run in parallel execution mode, the DataStage engine framework will distribute lookup records that arrive on the input link of the Lookup stage across the player processes of the Oracle Connector stage. Each player process will execute the lookup *SELECT* statement for each record that arrives to it. Note that the execution mode in this case is specified in the stage editor of the Lookup stage but applies to both the Lookup and Oracle Connector stages.

**Figure 4** illustrates the Oracle Connector stage configured to perform sparse lookup operation on the database in parallel execution mode. Two player processes are created for the stage in this example. Like in the previous two examples, this diagram implies that the upstream and downstream stages are running sequentially and processing single data set partitions. In practice one or both of those stages could be running in parallel and processing multiple partitions.
Note that the Lookup type property in the Oracle Connector can also be set to value Normal, in which case the Oracle Connector stage will run as if it had an output link and was configured to fetch records from the database. It will run the SELECT statement once as a static statement without any bind parameters, fetch all the rows and provide them to the Lookup stage which will then store them in memory and perform an in-memory lookup operation for each record arriving on the input link and will provide the resulting matching records on the output link.

Refer to the Related topics section for the link to the Information Server Information Center where you can find more information on the DataStage pipelining and partitioning parallelism features.

Next, we will look at how parallelism and partitioning in the Oracle database differ from the parallelism and partitioning in DataStage.

Oracle database can perform the operations requested by the player processes of the Oracle Connector stage in parallel or serial mode. The database executes a particular operation in parallel mode by breaking it up into multiple operations and running those operations simultaneously. Do not confuse the parallel operations in the Oracle database with the parallel execution mode of stages in DataStage jobs. When the connector stage is running in parallel execution mode, each of the player processes of the stage establishes its own connection to the database and submits database operation requests on that connection. The database treats each player process as an independent database client application, and the requests submitted by any of them may end up running as parallel or serial operations in the database. Any combination of parallel/sequential connector stage execution mode and parallel/serial database operation is therefore possible.

A number of factors determine if the database will choose to execute a given operation in parallel or serial mode. In some cases, the request to perform the operation can explicitly indicate how the operation should be executed, as we will see in some examples later in the text when we
discuss performance tuning for DML and bulk load operations. In other cases, the attributes of the database object on which the operation is performed determine how the operation should be executed. Refer to the Related topics section for the link to Oracle Database online documentation where you can find more information about parallel and serial database operations.

Database tables can be partitioned. The rows in a partitioned table are distributed across multiple table sections called partitions. The rule that defines which row belongs to which table partition is specified when the table is created. Refer to the Related topics section for the link to Oracle Database online documentation where you can find more information about partitioned tables.

Do not confuse table partitions with the data set partitions processed by the player processes of the Oracle Connector stage. An Oracle Connector stage configured to read data from a partitioned table or write data to a partitioned table can run in parallel or sequential execution mode. When it is running in parallel execution mode, the data set partitions processed by the stage do not necessarily need to be associated with the table partitions.

It is possible but not mandatory, to configure the Oracle Connector stage so that the number of player processes of the stage matches the number of partitions in the table accessed by the stage. In such a configuration each player process of the stage is accessing a dedicated table partition. When fetching records from a partitioned table, this type of configuration is accomplished by setting the Partitioned reads method connector property to value Oracle partitions. When writing records to a partitioned table, the same configuration is accomplished by specifying the Oracle connector value for the Partition type setting under the Partitioning tab for the input link in the connector stage editor. These special stage configurations are discussed in more detail later in the article in the sections that cover fetch, insert and bulk load operations.

Isolating performance issues

A variety of reasons may result in a less than expected performance of the operations performed by the Oracle Connector. In this section, a list of steps are suggested for analyzing DataStage jobs with Oracle Connector stages with the goal of identifying and isolating performance issues. Rather than strictly following these steps each time you need to analyze DataStage jobs with Oracle Connector stages, treat them as a set of guidelines for developing your own troubleshooting strategy while keeping in mind the specifics of your runtime environment.

Step 1: Check the CC_MSG_LEVEL environment variable

Ensure that the CC_MSG_LEVEL environment variable is not set or that it is set to value 3 or higher.

This environment variable is used for filtering connector messages in the job log. It applies to all connectors including Oracle Connector. Table 1 lists the supported values for this environment variable and their meaning:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When the `CC_MSG_LEVEL` environment variable is not set, its default value is 3. When it is set to value 1 the size of the job log may increase significantly, particularly when the job has multiple connector stages that are processing many records. When you wish to collect detailed job log with trace level messages by setting `CC_MSG_LEVEL` to 1, ensure that you run the job with a small number of records, a few hundred at most. You should never leave this environment variable set to value 1 or even 2 in the production environment. By ensuring that the value is set to 3 or higher or not set at all, not only will you prevent the job log from potentially consuming a lot of space, but you will also ensure that the performance of the job is not affected by the excessive logging activity.

To determine which value is used for this environment variable when the job runs, check the list of environment variables in the `Environment variable settings` message located at the top of the job log, as shown in Figure 5. If the `CC_MSG_LEVEL` is not present in the list that means that it was not explicitly defined and the default value 3 was used in that case.

**Figure 5. Environment variable settings in job log**

<table>
<thead>
<tr>
<th>Type</th>
<th>Occurred</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/7/2013 7:06:26 AM</td>
<td>Starting Job SimpleJob.</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:27 AM</td>
<td>Environment variable settings: ...</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:27 AM</td>
<td>Parallel job initiated</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:27 AM</td>
<td>Parallel job default: NLS map: UTF-8, default locale: OFF</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:27 AM</td>
<td>main_program: IBM InfoSphere DataStage Enterprise Edition 9.1.0.6502 ...</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:27 AM</td>
<td>main_program: conductor: uname: -s:Linux, -r:2.6.18-308.161.e15, -v: #1</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:27 AM</td>
<td>main_program: orchgener: loaded ...</td>
<td></td>
</tr>
<tr>
<td>1/7/2013 7:06:28 AM</td>
<td>my_ora_fetch: The connector connected to Oracle server orcl.</td>
<td></td>
</tr>
</tbody>
</table>

**Step 2: Simplify the job**

Determine whether the Oracle Connector stages in your job are responsible for the less than expected performance of the job. If the job consists of multiple stages, simplify the job by breaking it into self-sustainable segments (subsets of stages) and run each segment separately to check if it is the performance bottleneck of the job.

If a segment with the Oracle Connector stage is found to be the bottleneck, simplify it further by leaving the Oracle Connector stage and its attached links unchanged and replacing the adjacent stages with simpler stages. For example, attach a Peek stage on the output link and a Row Generator stages on the input link of the Oracle Connector stage. If you suspect that the performance issue is caused by specific data values in the data processed by the original
Tuning the Oracle Connector performance in IBM InfoSphere DataStage, Part 1: Overview of the connector tuning process and optimization of fetch, lookup and DML operations

job, then Row Generator stage will not be the best option because this stage generates values automatically. In that case, make a copy of the original segment, replace the Oracle Connector stage with a Data Set stage and run that segment to store the original input data to a data set file. Then modify the segment to use the Data Set stage on the input link of the connector stage and configure it to read data from the previously generated data set file. Once you have ensured that the Oracle Connector stage is the reason for performance issues, proceed to Step 3. Otherwise, identify the problematic job segment and address it.

If all the segments perform satisfactory when you run them individually but the job shows performance issues when you run it as a whole, start reassembling the job from scratch by adding together the job segments. Start with the segments that extract data from the external data source, continue by adding segments with the processing stages and finish with the segments that load the data to external data sources. Keep adding segments until you reach the segment for which you observe sharp decline in job performance. If the identified segment contains an Oracle Connector stage configured to load data to the Oracle database, ensure that Oracle Connector is the culprit by replacing it with another stage such as Peek or Data Set. Note that when you replace Oracle Connector stage with a Peek stage the job will likely run faster because Peek stage does not do much with the data that it receives. You will need to use your best judgment to determine if the time difference when running Oracle Connector and Peek stages is large enough to indicate an issue with the connector stage. You can gain additional insight by replacing the Oracle Connector stage with another stage capable of writing data to the Oracle database, such as for example the ODBC Connector stage.

**Step 3: Determine where the time is spent**

In order to reduce the time it takes for the job to complete, you will need to know where all the time is spent.

Set the `APT_PM_PLAYER_TIMING` environment variable for the job to value `True`. This environment variable is a built-in DataStage environment variable. When the job completes it will report the CPU time that was consumed by all the player processes in the job, including the player processes of the Oracle Connector stages.

If the reported CPU time for the connector player processes is close to the elapsed time for the job, it will serve as an indication that the connector is spending considerable amount of time to process data records or to perform other CPU intensive activities that do not involve round-trips to the database. In that case, refer to the later section in this article that discusses the operation that your stage is performing (fetch, lookup, DML or bulk load). You can also try to run the stage in parallel execution mode while increasing the number of player processes for the stage to better utilize the available CPU resources.

If the CPU time of the connector player processes represents a small portion of the elapsed time for the job, try running the job with a smaller data set. If the job completes quickly this time, proceed to Step 4 which focuses on analyzing database operations that are performed on the records. If on the other hand the job still runs slowly, focus on the database operations executed by the connector but that do not operate on the records directly, such as:
• SQL statements specified in Before SQL statement, Before SQL (node) statement, After SQL statement and After SQL (node) statement connector properties.
• SQL statements performed on the target table as specified in the Table action connector property.
• SQL statements performed before and after bulk loading data as specified in the Perform operations before bulk load and Perform operations after bulk load connector properties.
• SQL statements performed internally by the connector such as the queries issued against the database dictionary views.

Once you have identified the query that is taking a long time to complete, proceed to Step 4.

You can do one of the following two things to collect the statements that the connector issues on the database during the job execution, including the internal queries issued against the database dictionary views:

1. Set the CC_MSG_LEVEL environment variable for the job to value 2 and collect the debug level job log. This log will contain the text of the SQL statements issued by the connector. Also inspect the timestamps of the messages in the log and look for any large time gaps between subsequent messages and focus on the SQL statements logged at those times. Make sure to unset the CC_MSG_LEVEL environment variable or set it to value 3 or higher after you have collected the debug level log, as explained in Step 1.
2. Enable database tracing for the job and inspect the trace after the job completes. Refer to Step 4 for more details.

Step 4: Analyze the statement execution plan

Once you have identified the statement that takes a long time to complete, and the statement is one of SELECT, INSERT, UPDATE or DELETE statements, analyze the execution plan that the optimizer in the database prepared for this statement. You can use the EXPLAIN PLAN statement and DBMS_XPLAN package procedures in the Oracle SQL*Plus tool to obtain the execution plan for your statement without actually running the statement.

If the SQL statement contains bind parameters that use DataStage syntax (ORCHESTRATE.parameter_name), modify the statement to use bind parameters with Oracle syntax (:parameter_name) before running EXPLAIN PLAN on it. For example, suppose that you specified the statement in Listing 1 in the Oracle Connector stage:

Listing 1. Example statement

```sql
UPDATE TABLE1 SET C2=ORCHESTRATE.C2 WHERE C1=ORCHESTRATE.C1;
```

To obtain execution plan for this statement, choose an arbitrary identifier for your statement such as my_update_statement used in this example and run the statement shown in Listing 2:

Listing 2. Statements to explain and display the execution plan

```sql
EXPLAIN PLAN SET STATEMENT_ID='my_update_statement' FOR UPDATE TABLE1 SET C2=:C2 WHERE C1=:C1;
SET PAGESIZE 0;
SET LINESIZE 150;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY('PLAN_TABLE','my_update_statement'));
```
Refer to the Related topics section for the link to Oracle Database online documentation where you can find more information about EXPLAIN PLAN statement and DBMS_XPLAN package.

**Step 5: Trace the connector database activity**

In some cases, the execution plan that you obtained in Step 4 may not match the execution plan for the same statement when it is executed in the job. This can happen for example for a statement that contains a WHERE clause with bind parameters, as in those cases the database may need to perform additional functions on the values passed through the bind parameters, depending on the program variables and settings that the connector used for the bind parameters. If you suspect that this situation may have occurred, you can use the instructions in this step to capture the execution plan for the statement as the connector submits it when the job runs.

To do this you will need to collect the trace file for the player process of the stage at the time when it executed the statement. You will then need to locate this trace file on the database host and to convert it to a file in readable format which will include the execution plan for your statement.

Oracle database provides the tools that can be utilized to capture and process the trace files of client applications connected to the database (the client applications in this case are the player processes of the Oracle Connector stage). The SQL Trace facility or the DBMS_MONITOR built-in PL/SQL package can be used to produce the trace, the trcsess command-line utility can be used to consolidate multiple trace files into a single trace file based on a previously established criteria and the TKPROF command-line utility can be used to convert the trace file to a readable format. Refer to the Related topics section for the link to Oracle database online documentation where you can find more information about these tools.

You can utilize the Before SQL (node) statement and After SQL (node) statement connector properties to start and stop the tracing of the database sessions established by the player processes. You can also utilize them to set the client identifier and tracing identifier values of your choice so that later you can more easily locate the generated trace files. The statements in these two properties are executed in the same database session as the statements that process the records on the link and for which you are trying to capture the execution plans.

For example, specify the statements shown in Listing 3 in the Before SQL (node) statement property to set the tracing identifier my_tid for the database sessions established by the player processes and to start tracing the sessions using the SQL Trace facility. The same identifier will be used by all player processes of the stage because they will all execute the same Before SQL (node) statement.

**Listing 3. Alter session statement to enable SQL Trace**

```
ALTER SESSION SET TRACEFILE_IDENTIFIER = 'my_tid';
ALTER SESSION SET SQL_TRACE TRUE;
```

If you want to ensure that the generated trace files for individual player processes have trace file identifiers that reference the respective player processes (0, 1, 2, and so on), you can take advantage of the CC_ORA_NODE_USE-placeholder and CC_ORA_NODE-placeholder-name environment variables. For example, to generate trace files with identifiers my_tid0, my_tid1 and my_tid2 for the
connector stage configured to run in parallel execution mode in 3 player processes, perform the following steps:

1. Choose a placeholder literal value (for example CURRNODE).
2. Set the CC_ORA_NODE_USE_PLACEHOLDER environment variable to value TRUE (the value is not case-sensitive).
3. Set the CC_ORA_NODE_PLACEHOLDER_NAME environment variable to value CURRNODE (the value is case-sensitive).
4. Specify the two statements shown in Listing 4 in the Before SQL (node) statement property.

**Listing 4. Alter session statement to enable SQL Trace**

```sql
ALTER SESSION SET TRACEFILE_IDENTIFIER = 'my_tidCURRNODE';
ALTER SESSION SET SQL_TRACE TRUE;
```

The remaining instructions in this section assume that you chose to use the statements in Listing 3 to initiate the trace.

The tracing for the sessions will automatically stop when the job ends because all the player processes will disconnect from the database at that time, but if you wish to stop the tracing of these sessions explicitly you can specify the statement shown in Listing 5 in the After SQL (node) statement property:

**Listing 5. Alter session statement to stop SQL Trace**

```sql
ALTER SESSION SET SQL_TRACE FALSE;
```

When the job starts, the statements specified in the Before SQL (node) statement property will be executed by each player process and will start tracing for the database sessions established by the player processes. The statements that process the records will be executed next and will be traced. Before the job ends the statement specified in the After SQL (node) statement property will be executed by each player process to stop the tracing for the database session for that player process.

When the job finishes, go to the trace directory of the database instance to which the connector has connected. The location of this directory can be obtained by running the statement shown in Listing 6 on the instance to which the connector was connected:

**Listing 6. Alter session statement to locate trace directory**

```sql
SELECT VALUE FROM V$DIAG_INFO WHERE NAME='Diag Trace';
```

The trace directory will contain one file for each player process and each of them will have the _my_tid.trc suffix, which is what you will use to identify the trace files that were generated for the job. The prefixes for the trace files will be automatically generated by Oracle and will consist of the Oracle system identifier for the instance (SID), the string "ora" and the server process number. The trace files will contain execution plans for the statements that were executed in the sessions established by the player processes. Use the trcsess utility to combine the generated trace files into a single trace file, such as in Listing 7:
Listing 7. trcsess tool invocation example

```
trcsess output=output_file service=service_name *_my_tid.trc
```

The value `output_file` is the name of the file to generate with the consolidated trace information (for example `my_tid_combo.trc`) and `service_name` is the name of the service to which the connector connected. You can obtain the service name from the `tnsnames.ora` configuration file in the Oracle client installation. Locate the net service name section in this file that corresponds to the value that you specified in the `Server` connector property and then look for the `SERVICE_NAME` parameter value in that section. Alternatively, run the statement shown in Listing 8 on the database after connecting to it from a tool such as SQL*Plus using the same connection settings as the ones that you specified in the connector stage:

Listing 8. Determining service name

```
SELECT SYS_CONTEXT('USERENV', 'SERVICE_NAME') FROM DUAL;
```

Note that when you configure the stage to run in sequential execution mode in a single player process and you use Oracle dedicated server configuration, a single trace file with the `_my_tid.trc` suffix will be generated. In that case, the consolidation of trace files is not applicable. Also in some cases even if multiple trace files were generated you may want to intentionally skip consolidating them and analyze them separately instead. In both cases, you would not need to invoke the `trcsess` utility.

Once you have located or assembled the trace file that you wish to analyze, run the `TKPROF` utility on it to produce the readable output file. For example, run the command shown in Listing 9 to format the consolidated trace file `my_tid_combo.trc` to the readable output file `my_tid_combo.txt`:

Listing 9. tkprof tool invocation example

```
tkprof my_tid_combo.trc my_tid_combo.txt
```

Open the output file in a text editor and examine it. To quickly locate the sections in the file that reference the statements in which you are interested, search the file for the distinguishable words that appear in your statements, such as the table names referenced by the statements.

You can also use `DBMS_MONITOR` package to start and stop tracing of all sessions established by the player processes that have a particular client identifier. Specify the PL/SQL anonymous block shown in Listing 10 in the `Before SQL (node) statement` property to set `my_cid` client identifier on the sessions established by the player processes:

Listing 10. Setting client identifier on the session

```
BEGIN
DBMS_SESSION.SET_IDENTIFIER('my_cid');
END;
```

Before you run the job, enable the tracing for this client identifier by invoking the `CLIENT_ID_TRACE_ENABLE` procedure from SQL*Plus as shown in Listing 11:
Listing 11. Enabling tracing for a client identifier
EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE('my_cid', TRUE, FALSE);

When the job completes, stop the tracing by invoking the CLIENT_ID_TRACE_DISABLE procedure from SQL*Plus as shown in Listing 12:

Listing 12. Disabling tracing for a client identifier
EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE('my_cid');

Go to the trace directory, locate the generated trace files, consolidate them with trcsess as necessary and format them to the readable format using the TKPROF utility as described earlier. Note that in this case instead of service=service_name argument you will be in position to specify clientid=my_cid argument in the trcsess command line when consolidating the trace files. Also in this case specify *.trc for the filter expression.

Keep in mind that the statements specified in Before SQL (node) statement and After SQL (node) statement properties do not run in the same session as the statements specified in Before SQL statement and After SQL statement properties. The statements in the latter pair of properties apply to the database sessions established by the conductor process, which are different from the database sessions established by the player processes and to which the former pair of statements applies. For example, if you issue ALTER SESSION SET SQL_TRACE TRUE; statement as the first statement in the Before SQL statement property, you will be able to capture the trace for the statements that follow that statement in the Before SQL statement property and any other statements performed by the connector stage in the conductor process. If your job contains multiple Oracle Connector stages, they will all share the same conductor process, but they will establish separate database sessions. The produced traces for the sessions in conductor process will not include execution plans for the statements that operate on the records on the link because those statements will run in the database sessions established by the player processes.

Fetch operation

In this section, we will take a look at the Oracle Connector stage configured to execute SELECT statement on the database, fetch the resulting rows and deliver them as records on the output link. We will also look at the important connector properties for tuning fetch operation in the stage. These properties are illustrated in Figure 6

Figure 6. Connector properties that affect the fetch performance
**Array size**

The connector prepares and executes the specified `SELECT` statement once (per player process) and then proceeds to fetch rows from the database and deliver them as records on the output link. To reduce the overhead associated with each round-trip to the database the connector fetches records from the database in arrays of the size specified in the `Array size` connector property. The default value is **2000**, which means it fetches 2000 rows at a time. The optimal array size depends on the available system resources (memory and network primarily) as well as the average record size. In general, a very small array size such as 1 or a very large array size such as 30000 is unlikely to be optimal. As a rule of thumb, the larger the records are on average, the smaller the array size should be. One way to determine the optimal array size would be to run a set of three tests in which you would vary the `Array size` property value while keeping the remaining stage settings unchanged, and then compare the results. For example, run the test with the default array size 2000, then run the test with the order of magnitude smaller array size (200) and then with the order of magnitude larger size (20000). Run each test multiple times until you observe execution time worse than the one in the run before it, and take note of that previous run as the best run for that test. The results collected this way will help you decide if you should continue searching for the optimal array size in the same fashion with the values larger or smaller than 2000.

**Prefetching rows**

In addition to the `Array size` property, which enables fetching of rows in arrays of the specified size, the connector also has the `Prefetch row count` and `Prefetch buffer size` properties that are used to enable the Oracle's row prefetching feature. Prefetching of rows and array fetching both help reduce the number of round-trips to the database. The difference is that with array fetching the connector provides the empty array to the Oracle client and requests the array to be populated with the rows fetched from the database while the prefetching applies to the rows that the Oracle client preemptively fetches from the database in each round-trip in anticipation that the connector will explicitly request them later. The `Prefetch row count` value specifies the number of rows that the Oracle client should prefetch per round-trip and the `Prefetch buffer size` value specifies the size (in kilobytes) of the memory buffer that it should use for prefetching rows. In the latter case Oracle client prefetches as many rows as it can store in this buffer. When both values are specified, the Oracle client uses the value that results in fewer prefetched rows. To disable row prefetching, both property values must be set to 0. If only one is set to 0 the Oracle client performs prefetching based on the other property. Refer to the Related topics section for the link to Oracle Database online documentation for more information about the row prefetching feature.

In general when the job runs, the connector stage submits the specified `SELECT` statement once (per player process), retrieves all the rows represented by that statement in one or more round-trips and delivers them on the output link. To optimize the fetching of the rows from the database you will typically only need to determine and set the optimal `Array size` property value and leave the `Prefetch row count` and `Prefetch buffer size` properties set to their default values (1 and 0, respectively). An example of a scenario in which row prefetching may provide benefits on top of those provided by array fetching is when you have a data migration job that reads records from a table specified through a job parameter and you have runtime column propagation enabled on the output link to instruct the connector to automatically generate the schema on the link at runtime based on the accessed table definition. If the tables that the stage will be accessing in
different job runs vary significantly in terms of the average row sizes in them then a fixed \textit{Array size} property value may not work equally well for all the tables. For example, a small array size may result in good performance for a table with many \texttt{VARCHAR2(4000 BYTE)} columns, but may not provide satisfactory performance for a table consisting of only few \texttt{VARCHAR2(10 BYTE)} columns. Likewise a large array size may perform well for the table with few \texttt{VARCHAR2(10 BYTE)} columns but not so well for the table with many \texttt{VARCHAR2(4000 BYTE)} columns. This is because the connector allocates the array based on the schema definition, so a large array size in combination with the schema definition consisting of many large columns may end up consuming a lot of memory or even cause a memory allocation failure due to insufficient memory resources. In this scenario, you can set the \textit{Array size} to a fixed and relatively small value (for example 20), set the \textit{Prefetch buffer size} to a fixed and relatively large value in kilobytes (for example 4096), and keep the \textit{Prefetch row count} value set to 0. This way you will ensure that the array fetching is enabled in all job runs and that the additional prefetching of rows is performed by the Oracle client with the number of rows prefetched roughly inversely proportional to the size of rows in the table.

**Transaction size**

The \textit{Record count} connector property specifies after how many processed records on the link the connector should commit the transaction and start a new one. The default value is 2000. When fetching rows from the database this means that the connector will by default commit the transaction after every 2000 rows that it fetched and delivered on the output link. At the start of the job the connector ensures that the specified \textit{Record count} value is either 0 or a multiple of the \textit{Array size} value and if necessary adjusts the latter value to ensure this.

Each transaction commits results in a round-trip to the database, which negatively affects the performance. The default value for the \textit{Record count} property is set to 2000 but in most cases when fetching rows from the database you should set it to 0 so that the connector commits the transaction only once (per player process) at the end of the job.

**Transaction waves**

A scenario in which you will want to set the \textit{Record count} property to a value other than 0 is when you want the connector to fetch records and deliver them on the output link in waves of a particular wave size. In this case, the \textit{End of wave} property would need to be set to \textbf{Yes}, and the \textit{Record count} property would need to be set to the number of records to include in each wave. Since the \textit{Record count} value must be a multiple of the \textit{Array size} value, when you specify a small \textit{Record count} value in order to flush records on the link in short waves, the \textit{Array size} will also need to be small which will negatively affect the performance. This is another example where the use of \textit{Prefetch row count} and \textit{Prefetch buffer size} properties could be a used as a strategy for reducing the number of round-trips to the database while keeping the \textit{Array size} value small.

Note that you don’t have to deliver records in waves from the Oracle Connector stage directly. You can group them into waves downstream from the Oracle Connector stage. To do this, connect the Oracle Connector stage to a Wave Generator stage. Set the \textit{Record count} property to 0 and \textit{End of wave} property to \textbf{No} in the Oracle Connector stage and define the wave boundaries in the Wave Generator stage instead. The downside of this approach is that it will result in an additional stage in the job. However, it will allow Oracle Connector to fetch records faster by using
a single transaction and large array and at the same time offer more flexibility for defining waves boundaries by utilizing the options provided by the Wave Generator stage specifically for this purpose.

**Transaction isolation level**

The isolation level for the transactions is specified in the *Isolation level* connector property. The default value is *Read committed* and will be sufficient in most cases, but it is a good idea to also try the remaining two values *Serializable* and *Read only* to check if they result in better performance. Refer to the Related topics section for the link to Oracle Database online documentation for information about transaction isolation levels in Oracle database and apply that information to understand how differences between the isolation levels may play a role in your environment.

If you set *Isolation level* property to *Read only* ensure that the *Record count* property is set to 0. The reason for this is that the connector enforces read only isolation level by executing the `SET TRANSACTION` statement each time a new transaction starts. When the *Record count* is set to a value other than 0, the connector will commit the transaction after the specified number of records and will issue `SET TRANSACTION` statement in the new transaction. This may negatively affect the performance, especially for small *Record count* values since they will imply more frequent transaction commits.

**Fetching rows with the stage in parallel execution mode**

As you saw earlier in the section that covered parallelism and partitioning, in order for the connector stage to run in parallel execution mode and fetch rows from multiple player processes, you will need to set the *Execution mode* stage setting to *Parallel* and set the *Enable partitioned reads* connector property to *Yes*. The `SELECT` statement specified for the stage will then be modified automatically by the connector for all player processes created for the stage so that each player process will fetch a distinct subset (partition) of rows from the database. The *Partitioned reads method* connector property specifies the method that the connector uses to modify the `SELECT` statement for the individual player processes. Refer to the Related topics section for the link to the Information Server Information Center where you can find detailed information about partitioned reads methods supported by the Oracle Connector.

Typically the default *Rowid range* partitioned reads method will provide the best performance. When this method is specified, the connector performs the following steps:

1. Queries the `DBA_EXTENTS` system dictionary view to determine the range of ROWIDs for all the rows that need to be fetched.
2. Divides the range of ROWIDs into the number of sub-ranges that matches the number of player processes of the stage.
3. Modifies the original `SELECT` statement for each player process by adding a `WHERE` clause to it to request from the player process to fetch only the rows with the ROWIDs that belong to the sub-range associated with that player process.

The individual player processes are able to fetch the rows quickly in this case because the `WHERE` clauses are based on ROWID ranges, which enable the database to quickly locate and retrieve the rows from the physical table storage.
In some cases, the queries that the connector performs on the `DBA_EXTENTS` dictionary view may take a long time to complete. The statements that the connector executes can be located in the job log when the `CC_MSG_LEVEL` environment variable is defined for the job and set to value 2. If you determine that the queries on `DBA_EXTENTS` view are running too slowly, provide your database administrator with the statements that you collected from the job log and try to determine why they run slowly. Possible reasons include missing statistics on the tables or unusually large number of database extents, which can happen when extents of fixed small size are used instead of automatically managed tablespaces. If you are unable to determine the reason why the queries on `DBA_EXTENTS` dictionary view are too slow, or if you understand these reasons but are not in a position to address them, or if the database user specified in the connector does not have select access granted on the `DBA_EXTENTS` view, you can use one of the remaining partitioned reads methods as they do not access the `DBA_EXTENTS` view.

The `Modulus` and `Minimum and maximum range` partitioned reads method will result in good performance when the table column that you specify to be used with this method meets the following conditions:

- It is of `NUMBER(p)` type where precision `p` is explicitly defined.
- It has a high cardinality meaning the number of rows with distinct values for this column is high relative to the total number of rows in the table.
- It is indexed. For the `Modulus` method, having a function-based index on the function `MOD(column_name, player_count)` where `column_name` is the name of the column and `player_count` is the number of player processes of the stage could result in additional performance improvements. If the `player_count` is small, you may also consider creating that function-based index as a bitmap index. Refer to the Related topics section for the link to Oracle Database online documentation where you can find more information about the function-based and bitmap types of indexes.

The `Oracle partitions` partitioned reads method may provide considerable performance improvements when the following conditions are met:

- The table is partitioned.
- The number of table partitions is not too large. At runtime the number of player processes of the stage will need to match the number of table partitions. If the stage is configured to run in parallel execution mode in fewer player processes than there are partitions in the table, the connector will automatically increase the number of player processes at runtime to match the number of table partitions. For example, if the table has 50 partitions, there will be 50 player processes running in parallel, each fetching rows from one dedicated table partition. The overhead of starting up this many player processes and establishing that many connections to the database may outweigh the benefits of performing the partitioned reads.
- The rows are evenly distributed across the table partitions. If the rows are concentrated in a small subset of table partitions, then a small subset of player processes will fetch most of the rows and most of the player processes will run idle.

The `Rowid hash` or `Rowid round robin` partitioned reads methods can be used when none of the remaining methods present a good fit. On the upside they do not require access to the
The DBA_EXTENTS dictionary view and do not require the table to have a NUMBER(p) column or to be partitioned but the downside is that the modifications that they make to the SELECT statement involve more SQL functions and therefore slow down the statement execution on the database side. Between these two, the use of Rowid hash method will typically result in better performance.

When you configure the stage with a custom SELECT statement instead of relying on the stage to generate the statement automatically at runtime based on the table name, the connector will inspect the specified statement at runtime and will choose a table from the statement to utilize for making modifications in the WHERE clause for individual player processes. If the specified statement is very complex and references multiple table names, the choice that the connector makes may not be the optimal choice in terms of performance. You can enforce the use of a particular table in the statement by specifying its name in the Table name for partitioned reads connector property.

Another option you can consider when you have a complex custom SELECT statement is to create a view based on that statement and then configure the stage to fetch rows from that view using Modulus or Minimum and maximum range partitioned reads method. This will keep the modified queries simple since the WHERE clause will be appended to the plain SELECT * FROM view_name auto-generated statement. Also, you can consider creating a materialized view for this query, and configuring the stage to fetch rows from that materialized view in which case you will also be able to utilize the Rowid range partitioned reads method.

Run the tests to ensure that your stage configured to run in parallel execution mode and fetch rows from multiple player processes will actually yield higher record throughput than when it is configured to run in sequential execution mode and fetch all rows from a single player process. This is especially important in cases when the query is very complex or when the number of fetched rows is relatively small compared to the total number of rows contained in the tables referenced by the query.

**Lookup operation**

In this section, we will take a look at the Oracle Connector stage configured to perform lookup operations on the database. In this configuration a reference link connects Oracle Connector stage to a Lookup stage. We take a close look at the connector properties that play important role for tuning lookup operation performance in the stage. These properties are illustrated in Figure 7.

**Figure 7. Connector properties that affect lookup performance**

<table>
<thead>
<tr>
<th>Lookup type</th>
<th>Sparse</th>
<th>Lookup mode used by the stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>2000</td>
<td>Number of matching rows to fetch at a time</td>
</tr>
</tbody>
</table>

The connector supports two lookup modes, sparse (direct) and normal (in-memory). The selection is made through the Lookup type property in the connector stage editor.

In sparse lookup mode the connector stage runs the SELECT statement with a WHERE clause. The WHERE clause contains bind parameters (in ORCHESTRATE.param_name form) that are initialized with the key column values of the records arriving on the input link of the Lookup stage (referred to as
the key records). The lookup result set in this case consists of all the records that would have been returned by the specified SELECT statement had the WHERE been omitted. The connector executes the SELECT statement for each key record, fetches the matching records from the lookup result set and provides them to the Lookup stage, which in turn delivers them on its output link.

In normal lookup mode the connector runs the specified SELECT statement once. The statement may still have the WHERE clause but does not have any bind parameters. The lookup result set in this case consists of all the records returned by this statement. The connector provides the entire lookup result set to the Lookup stage which stores it in the memory. The Lookup stage then performs in-memory lookup for each key record arriving on its input link based on the key column values in that record, and provides the matching records on its output link.

The choice between sparse and normal lookup types can have a major effect on the lookup performance. Use the following guidelines to choose between the two:

- How many rows are there in the lookup result set? The smaller the lookup result set is, the more likely that the normal lookup mode will be a better choice since it will be more feasible to store the entire result set in memory. As a rule of thumb, a lookup result set with thousands of rows would probably be a good candidate for normal lookup mode and a lookup result set with millions of rows would probably be better handled in sparse lookup mode.
- Are the lookup key columns indexed in the referenced lookup table? As explained before, in sparse lookup mode the lookup key columns are referenced by the WHERE clause of the SELECT statement. If there is no index defined on these columns in the table, a complete scan of the lookup result set will be performed for each input record, which can have major negative impact on performance, especially when the lookup result set is very large. In this case, normal lookup would probably be a better choice, even for larger result sets, because the connector would run the SELECT statement only once and after that the lookup operation would be performed in memory. If you plan to utilize the connector in sparse lookup mode, make sure that the table index is defined on the columns used as the lookup key columns. The only time when the lack of indexes will not have a major impact on performance is when the lookup result set is very small, but in that case normal lookup type will probably be a better choice anyway.
- If the content of the lookup table changes while the job is running, do you need the lookup operation for the input records to be performed on the most recent rows in the table, or only the rows that existed in the table when the job started? In the former case you would need to use the sparse lookup mode, because in normal lookup mode the entire lookup result set is captured only once when the job starts.

Note that in the lookup scenarios the **Array size** property applies to the fetched records (lookup matches) and not to the input (key) records. In normal lookup mode the connector runs the SELECT statement once and fetches the records using arrays of the specified size. The same guidelines that were presented earlier in the section about **fetch operation** apply in this case. In sparse lookup mode the connector does not buffer the input (key) records in the array but instead executes the SELECT statement separately for each input record that arrives and then fetches the results in the array of the specified size. When the sparse lookup operation results in unique matches (one matching record is found per input record) then the array size value does not have much impact,
but if the matches are not unique then the array size equal or greater than the expected number of matches per input record will help improve the performance because the fetching of the matching records will require fewer database round-trips.

**DML operations**

In this section, we will look at the Oracle Connector stage configured to consume records from the input link and write them to the database using DML statements such as INSERT, UPDATE, DELETE or PL/SQL statements. Now take a close look at the connector properties and environment variables that play important role for tuning DML operation performance in the stage. They are illustrated in [Figure 8](#).

![Figure 8. Connector properties that affect DML performance](#)

### Array size, Transaction size and Transaction isolation level

The connector accepts records on the input link and stores them in an internal array in memory. When the array becomes full the connector submits the array to the database to perform the specified DML operation. The connector repeats this process until there are no more records available on the input link.

The size of the array that the connector uses for this is specified in the **Array size** connector property. The default value is **2000**. To find out the optimal array size value, refer to the earlier section that covered the fetch operation. The guidelines presented in that section for determining optimal array size apply in this case as well.

The **Record count** property specifies after how many records the transaction should be committed. The value **0** means that it should be committed only once after the last record is written to the database. If the **Record count** value is too small it will result in frequent transaction commits and if it is very large will result in long-running transactions in the database when there are many records that need to be processed. Neither of these scenarios is likely to produce optimal performance so typically you will need to do some testing to determine the optimal value for your environment. Essentially the guidelines presented for selecting optimal **Array size** apply in this case as well. Keep in mind that the **Record count** value must either be 0 or a multiple of the **Array size** value.

The **Isolation level** property does not play such an important role in this case as it does for the fetch operation which was discussed earlier, because in this case the connector does not issue SELECT statements so non-repeatable (fuzzy) reads and phantom reads associated with the Read committed isolation level are not applicable. The exception to this is the scenario when the connector stage is running anonymous PL/SQL blocks that perform both SELECT and DML statements on the same table and the stage is configured to run in parallel in multiple player processes. In that case the SELECT statements will be affected by the choice of isolation level.
in terms of the changes that they see and that were made by the remaining player processes. Also note that if the stage is running in parallel execution mode in multiple player processes and is performing `UPDATE` and `DELETE` statements on the table, the job may fail with Oracle error "ORA-08177: Cannot serialize access for this transaction" in the case when two player processes try to update or delete the same row. But as we explain in detail in the **DML operations performed by the stage in parallel execution mode** section, you should prevent the scenario where two player processes try to access the same row in the first place because it may lead to frequent lock waits or even deadlocks.

The **Read only** isolation level is not applicable in this case because the stage is making data changes in the database. The decision to use **Read committed** and **Serializable** isolation level should not be made based on performance requirements but on how you need multiple player processes to see the changes made by each other and this will further depend on what your stages are actually doing with the data. Refer to the **Related topics** section for the link to Oracle Database online documentation where you can find more information about the transaction isolation levels that can help you decide which isolation level is the best choice for your scenario. Unless you have specific reasons to use **Serializable** isolation level, use the default **Read committed** isolation level.

Note that when the records are arriving to the input link of the stage in waves, the **Array size** and **Record count** property values apply to each wave separately. In other words each wave will be treated as a mini-job within the job in regard to how **Array size** and **Record count** property values are applied. For example, if you set **Record count** to 0, the records will be committed to the database after each wave, and not just once after the last wave.

**Array operation modes**

The Oracle Connector uses the Oracle's batch error mode by default when writing records to the database. The connector can also be configured to disable this mode by setting the `CC_ORA_DISABLE_BATCH_ERRORS` environment variable to **TRUE** (the value is not case sensitive so setting it to **True** or **true** will have the same effect).

When the batch error mode is disabled and the connector submits an array of records to the database, the database performs the requested operation on each of the records in the array in the order in which they are stored in the array. When the database encounters an error for a record in the array, it stops the processing and returns error code for that record to the connector. The connector takes the action based on how it was configured to handle that particular error - it can ignore it, send the record to the reject link, issue a warning message or roll back the transaction and cause the job to fail. In cases when the connector does not roll back the transaction, it handles the error and re-submits the portion of the array following the record in error. The connector continues in this fashion until all the records in the array have been processed by the database. It then repeats the entire process with new arrays until there are no more records available on the input link.

When the batch error mode is enabled, the connector works in a similar way but with one important difference. When the database encounters an error for a particular record in the array,
it takes note of the error code and continues to process the remaining records in the array. Once it has processed all the records in the array, it returns back to the connector an array (batch) of status codes indicating the operation outcomes for all the records in the submitted array. The connector inspects the status codes and handles the records in error and proceeds to repeat the entire process with new arrays until there are no more records available on the input link.

In general case, the stage running in batch error mode will result in better performance than the stage that has this mode disabled. Specifically it will perform better in the following scenarios:

- When there are records in the array that are in error and that need to be handled in a fashion other than failing the job. In this case, the number of round-trips to the database per array will be 1 irrespective of the number of records in the array that are in error. On the other hand when the batch error mode is disabled, the number of round trips could in the most extreme case be the same as the size of the array, which will happen when all the records in the array are found to be in error.

- When `UPDATE` or `DELETE` statements are used and the connector needs to know how many rows in the database were updated or deleted by each of the input record. The connector will need this information, for example, when `Update then insert` or `Delete then insert` write modes are specified for the stage or when reject link is defined for the stage and any of the `Row not updated – Update mode`, `Row not updated – Insert then update mode` and `Row not deleted – Delete mode` reject conditions are specified for the reject link. In those cases, the connector will append the `RETURNING COUNT(*)` clause to the statement and as a result it will receive back from Oracle the array (batch) of count values indicating the numbers of rows affected by the individual records in the input array. When the batch error mode is disabled, the connector will not modify `UPDATE` and `DELETE` statements and will not use arrays to submit records for `UPDATE` and `DELETE` statements but will instead submit them one at a time.

However, in the following cases disabling the batch error mode is likely to result in better connector performance:

- When `Write mode` property in the stage is set to value `PL/SQL`. In this case, the `Array size` property will be disabled. In batch error mode the connector will automatically enforce the array size of 1 at runtime. The records are submitted to the database one by one where the specified PL/SQL anonymous block is performed on them one at a time. On the other hand, when the batch error mode is disabled, the `Array size` property will still be disabled but the connector will automatically calculate the optimal array size at runtime and will submit records to the database in arrays of that size. This in turn results in fewer round-trips to the database which will generally lead to improved performance.

- When you include `APPEND_VALUES` hint in the `INSERT` statement for the stage. For example, consider the statement shown in Listing 13.

**Listing 13. Insert statement with APPEND_VALUES hint**

```
INSERT /*+ APPEND_VALUES */ INTO TABLE1 VALUES(
 ORCHESTRATE.COL1,
 ORCHESTRATE.COL2);
```
In this example, the APPEND_VALUES hint will instruct Oracle to perform the insert operation using the Oracle direct path interface, the same interface that the connector utilizes when it is writing records in bulk load mode, which is discussed in more detail later. The APPEND_VALUES hint applies to the VALUES part of the INSERT statement which in turn means that it applies to the records provided by the connector.

If you configure the stage to run in batch error mode and you specify INSERT statement with APPEND_VALUES hint you will receive Oracle error "ORA-38910: BATCH ERROR mode is not supported for this operation".

If you configure the stage to disable the batch error mode then you will be able to use APPEND_VALUES hint. You should not use reject links in this case because the connector will not be in position to detect and process individual records that are in error. Also, in this case you need to make sure that Record count and Array size properties are set to the same value. The reason for this is because every array must be committed before the next array is processed, otherwise an attempt to insert the second array in the same transaction will result in Oracle error "ORA-12838: cannot read/modify an object after modifying it in parallel".

The APPEND_VALUES hint should not be confused with the APPEND hint. Both hints involve executing the INSERT statement as a parallel operation through the direct path interface. The difference is that the APPEND_VALUES hint will apply to the VALUES part of the statement and the APPEND hint will apply to the subquery of the statement. Refer to the Related topics section for the link to Oracle Database online documentation where you can find more information about APPEND_VALUES and APPEND hints. As an example the statement shown in Listing 14 the APPEND hint will result in selecting rows from TABLE2 and inserting them into TABLE1 using direct path interface.

Listing 14. Insert statement with APPEND hint

```
INSERT /*+ APPEND */ INTO TABLE1 SELECT * FROM TABLE2;
```

In summary, the connector can run INSERT statement with APPEND_VALUES hint only when the batch error mode is disabled. It can run INSERT statement with APPEND hint both when the batch error mode is enabled or disabled but in either case the APPEND hint will not affect the inserting of records provided by the connector.

Choosing between Insert then update and Update then insert write modes

When you want each record from the input link to be inserted into the target table if it is not already in the table as determined by the unique constraints defined for the table. To update the row with the values from the record if the row is already present in the table, you can accomplish this by setting the Write mode property to Insert then update or Update then insert value. When deciding which of the two modes to specify, you need to estimate if most of the input records will be inserted as new rows or will result in updates of the existing rows. Insert then update should be chosen in the former case and Update then insert in the latter case in order to reduce the probability of having to execute the follow-up statement for the records (UPDATE statement in Insert then update case and INSERT statement in Update then insert case).
DML operations with the stage in parallel execution mode

When the connector stage is configured to run in parallel execution mode in multiple player processes, the records arriving on the input link of the stage are distributed (partitioned) across the player processes of the stage and each player process writes its partition of records to the database. The Partition type setting for the link determines the strategy that is used to partition the input data set for that link across the player processes. Refer to the Related topics section for the link to the Information Server Information Center where you can find detailed descriptions of the built-in partition types in DataStage.

When multiple player processes are simultaneously writing subsets of records to the database, the performance will generally be better than when a single player process is writing all the records. The optimal number of player processes to use will depend on the available system resources on the DataStage engine and database server hosts.

An important aspect to consider for running multiple player processes for the stage is the data contention on the database side. It is important to understand how likely will the player processes compete for locks on the database side. If there is a chance that two player processes will try to update or delete the same row in the table, that will result in a lock wait for one of the player processes. In many cases, you can prevent this from happening by carefully selecting the Partition type value for the input link. For example, if the target table contains primary key column C1 and the stage is configured to run in Update then insert mode, then by setting the Partition type for the link to use Hash or Modulus function on the input link column C1 you will ensure that all records from the input data set that have the same C1 value will be processed by the same player process. This way you can prevent row level lock waits from happening because each time a player process obtains a lock on a row it will remain the only player process that may try to access that row again and will not need to wait for a lock because it will already hold it.

Failure to prevent the scenario in which active transactions from two player processes try to modify the same table row may further lead to deadlocks. This happens when each of the player processes requests a lock that the other player process holds at that same time. The database will detect this situation and in order to prevent the two player processes from waiting indefinitely on each other to release the locks that they need, it will return Oracle error “ORA-00060: deadlock detected while waiting for resource” to one of the player processes and it will allow the other player process to obtain the lock that it needs and to proceed. However, the player process that has received the error will log it as a fatal error and will cause the entire job to fail.

When there is a chance for the lock waits or deadlocks to occur, the probability for them to occur increases as the number of player processes increases because there will be more concurrently running transactions that compete for the row level locks. The probability also increases as the Record count value increases since it will result in larger transactions for the player processes in terms of the number of table rows affected by the transactions and consequently the number of row level locks acquired by the transactions. The probability further increases if individual input records can result in updating or deleting multiple rows in the target table because that will again make the transactions larger in terms of the table rows affected by them and will increase the probability of two transactions requesting a lock on the same row at the same time.
If the **Record count** property is set to 1 and individual input records update or delete at most one row in the table, then any waits for row level locks will be very short and deadlocks will be eliminated. This is because **Record count** must be either 0 or a multiple of **Array size** value, so in this case each player process will automatically run with the **Array size** of 1 which means that each player process will be requesting one row lock per transaction and releasing it when the transaction is committed. However, it will have a severe negative impact on the performance as it effectively disables the array operation and will commit transactions very often. In this case, it is likely that a better performance would be achieved by running a single player process for the stage but with a larger **Record count** and **Array sizes** values.

Below are some additional things to keep in mind when performing the DML operations with the stage configured in parallel execution mode:

- Check if the table accessed by the connector has a unique or primary key that is referenced by a foreign key of another dependent table. If the player processes will be modifying values in the primary key columns, ensure that the foreign key is indexed to prevent the player processes from requesting full table lock and performing full table scan on the dependent table.

- In addition to configuring the stage to prevent multiple player processes from modifying the same row in the target table, you should also keep in mind the scenarios where player processes are accessing distinct rows in the table but those rows reside in the same table blocks. Each table block has a section in its header called Interested Transaction List (ITL), which keeps track of the transactions that are modifying rows in that block. Each time the transaction from a player process needs to make a change to a row in a block, it is assigned a slot in the ITL of that block.

To ensure that the player processes do not wait to obtain a slot in ITL, the number of slots should be equal or greater than the number of player processes expected to access rows concurrently in the same block. The initial and maximum number of ITL slots for the blocks in the table are specified when the table is created through the **INITRANS** and **MAXTRANS** physical attributes for the table segment. The **MAXTRANS** attribute is deprecated when creating tables starting with Oracle version 10gR1 and is automatically set to 255, which means that it will allow the ITLs for blocks in that table to be expanded to the maximum of 255 slots, which for all practical purposes will be sufficient.

What is more likely than **MAXTRANS** to prevent the ITL from expanding is the available free space in each block. The table attribute **PCTFREE** plays an important role here because when set to a very low value it may result in a block being filled up with the row data to the point where there is no more space left in the block for the ITL to expand when a new slot is needed in it to accommodate a new transaction. This scenario will result in ITL lock wait for the new transaction. In most practical cases the default values for **INITRANS**, **MAXTRANS** and **PCTFREE** attributes will be adequate and will not require modifications.

But if your Oracle Connector stage is modifying many rows in the target table, and you have configured the stage to use a large **Record count** property so that many records are processed per transaction, and there are a lot of player processes, you should make sure that the values for these attributes are configured optimally, especially if your table uses large block sizes (such as 32K). Note that these three attributes also exist for the index segments,
as well as the partition and subpartition segments of partitioned tables and indexes. So if the stage is accessing a table that has indexes, is partitioned, or both, make sure to consider these attributes for all of the segments affected by the DML operations performed by the stage. You can query the V$SEGMENT_STATISTICS dictionary view to check if any ITL waits took place for the blocks in segments accessed by the stage.

- Ensure that your table does not have a bitmap index defined on it. If it does, run a single player process for the stage or drop the index prior to writing rows to the table and rebuild it afterwards.
- Ensure that the table is not accessed by external client applications or other DataStage jobs concurrently with your DataStage job. If such a concurrent access can happen then keep in mind that the considerations listed in this section apply across all the concurrently running client applications and jobs.

### DML operations on a partitioned table

If the table to which the connector is writing records is partitioned then you may be able to achieve significant performance improvements by specifying Oracle connector value for the Partition type setting for the input link. When you do this, the connector stage will take charge of distributing the input link records across its player processes and will do this in accordance with the partition key definition in the target table. The result is that each player process of the stage will receive input records that correspond to a single table partition associated with it. The connector will ensure at runtime that the number of player processes matches the number of table partitions and that each player process is associated with exactly one table partition.

The following is a list of scenarios for which the performance is likely to improve when you choose Oracle connector value for the Partition type setting:

- The table is partitioned using Range, List or Interval partitioning strategy, and is optionally subpartitioned. The remaining table partitioning strategies such as Hash are not supported in combination with the Oracle connector value for Partition type. You can still configure the stage to write records to those tables in parallel execution mode from multiple player processes but you will need to choose a Partition type value other than Oracle connector. If you leave the Oracle connector value set in this scenario, the connector will automatically enforce sequential execution at runtime.
- The rows that are inserted, updated or deleted by the stage are distributed evenly across all table partitions. If that is not the case then the player processes of the stage will not be evenly utilized. For example, if the table is range partitioned on a DATE column and has 10 partitions and all the records that you are inserting in the job end up in the same table partition (for example the partition for the current calendar year) then running the stage with the Oracle connector value for Partition type will result in 10 player processes running in parallel but only one of them will actually be inserting the records.

In this case, it may be better to run the stage in parallel using for example Hash value for the Partition type setting based on the column that corresponds to the DATE partition key column in the target table, plus optionally indicate the name of the partition in the INSERT statement to let the database know that all the records need to be written to that particular partition and that an attempt to write a record to another partition should be treated as error.
• The table does not have too many partitions. Since a player process will be started for each table partition, if the CPU and network resources on the engine tier hosts are not sufficient to effectively utilize all the player processes then the overhead of starting and maintaining the player processes may outweigh the benefits of writing records to the database in parallel. In that case, if you run the stage in parallel in fewer player processes and you set Partition type to a built-in value such as Modulus or Hash you may be able to achieve better performance, especially if you apply Modulus or Hash to the input link columns that correspond to the partition key column of the target table.

• The table columns that constitute the partition key for the table are of DATE or NUMBER\(p\) (no scale) Oracle data types. Additionally, the input link columns that correspond to the DATE table columns are of Date or Timestamp (without Microseconds) DataStage data types, and the input link columns that correspond to the NUMBER\(p\) table columns are of TinyInt, SmallInt, Integer or BigInt DataStage data types. Note that a Decimal\(p\) link column is technically a better match for the NUMBER\(p\) table column than some integer link column, but in this particular case integer link column will likely result in better performance because the connector will be able to partition the input records faster. Other data types for partition key table columns and input link columns may in some cases produce worse results when Partition type is set to Oracle connector value than when it is set to one of the built-in values. The best way to determine this is by running tests and inspecting the CPU times for individual player processes and the elapsed time for the job. Given the partition key columns in the target table, try varying Partition type values as well as the data types for the corresponding link columns and compare the results. When trying different data types for the link columns make sure that the data types you selected are valid for the actual field values in the input records and do not result in data loss. Note that when the Partition type is set to Oracle connector value, the CPU consumption of the connector code responsible for partitioning input link records will be combined and reported as the CPU time for the player processes of the adjacent upstream stage (which is the stage from which the link leads to the Oracle Connector stage).

Conclusion

In this article we covered various aspects of performance tuning of the Oracle Connector stage in DataStage jobs. We discussed the troubleshooting process and provided guidelines for maximizing connector performance through the optimal configuration of connector properties and link column definitions. We also presented some job design techniques that can further help you improve the job performance.

Part 2 of the series will explain how to optimize bulk load operation and will detail considerations for reject links and data types.

In combination with the information listed in the Related topics section and the information you collected through your own research, this article should help you achieve optimal performance of Oracle Connector stages in your DataStage jobs.

Acknowledgments

Many thanks to the following contributors for their valuable input and review of this article:
• Paul Stanley, InfoSphere Senior Architect
• Tony Curcio, InfoSphere Product Manager
• Fayaz Adam, InfoSphere Staff Software Engineer
Related topics

- Use an **RSS feed** to request notification for the upcoming articles in this series. (Find out more about **RSS feeds of developerWorks content**.)
- Explore the **IBM InfoSphere Information Server Version 9.1 Information Center** where you will find technical documentation for Information Server tools and components such as DataStage and Oracle Connector.
- Visit **IBM Information Management Redbooks** page and download redbook titles such as **InfoSphere DataStage Parallel Framework Standard Practices** and **IBM InfoSphere DataStage Data Flow and Job Design**.
- Find details about Oracle Database concepts mentioned in this article in the **Oracle Database Documentation Library 11g Release 2 (11.2)**.

© Copyright IBM Corporation 2013

**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)