An Oracle White Paper
April 2014

Performant and scalable data loading with Oracle Database 12c

Introduction ... 3
Data loading with external tables ... 4
Prepare for data loading ... 5
    Considerations for the external staging files ... 5
    Database objects necessary for data loading ... 9
    Database parameter settings for optimal load performance ... 13
    Ensure sufficient parallelism for the load operation ... 15
    Parallel loading in a Real Application Clusters (RAC) environment 16
    Considerations for the target table ... 17
Load the data ... 20
    DML versus DDL ... 20
    Locking during Load ... 21
    Activate parallelism for a load operation ... 21
    Partition Exchange Loading ... 22
    Index Maintenance during load ... 25
    Analyzing and monitoring your load operation ... 26
Post-load activities ... 29
    Statistics maintenance ... 29
Conclusion ... 32
Appendix A: Summary of best practices recommendations ... 33
    Settings Summary ... 33
Appendix B: Advanced topics ... 35
    Memory consideration for load ... 35
    Distribution Methods of Parallel Load ... 36
    The impact of data distribution (skew) in external data files ... 38
    Accessing remote data staging files using Oracle external tables 41

Introduction

The main goal when loading data into a data warehouse, whether it is an initial or incremental load, is often to get the data into the data warehouse in the most performant manner. Oracle Database 12c supports several embedded facilities for loading data including: external tables, SQL*Loader, and Oracle Data Pump. This paper focuses on loading data using the most scalable method for a data warehouse. It is by no means a complete guide for data loading with Oracle.

This paper is divided into four sections; after an overview of the load architecture, each section addresses a key aspect of data loading.

- Data loading with external tables
- Prepare for data loading
- Load the data
- Post-load activities

A lot of data loading scenarios, where some resources have to be reserved for concurrent end user activities while loading, will benefit from this paper as well: by applying the discussed best practices recommendations and scaling down the allocated resources for the data load, concurrent end users should be able to complete their activates with zero impact from the data load.

Data loading with external tables

Oracle's external table functionality enables you to use external data as a virtual table that can be queried directly and in parallel without requiring the external data to be first loaded in the database.

The main difference between external tables and regular tables is that an external table is a read-only table whose metadata is stored in the database but whose data in stored in files outside the database. The database uses the metadata describing external tables to expose their data as if they were relational tables, so you always have to have the external data accessible when using an external table. Since the data resides outsides the database in a non-Oracle format you cannot create any indexes on them either. Figure 1 illustrates the architecture and the components of external tables.

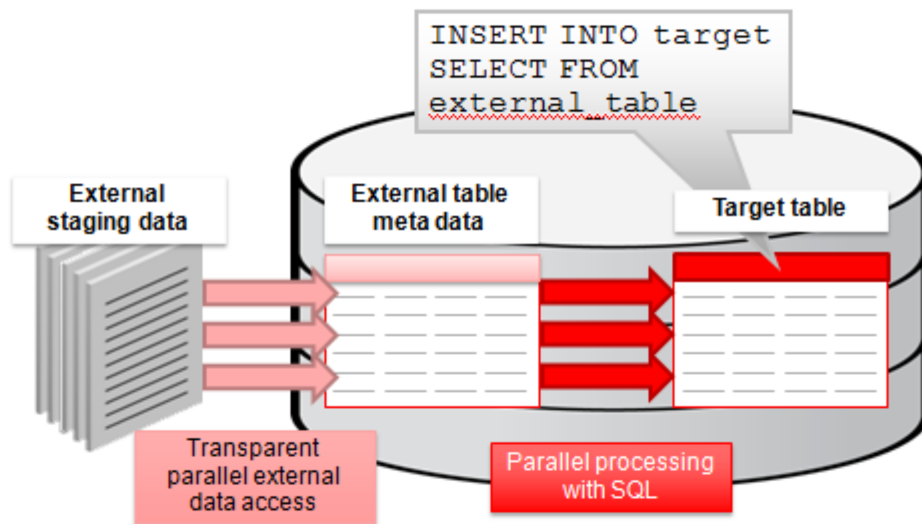

Figure 1: Architecture and components of external tables

To ensure a performant and scalable data loading using external tables requires each individual component and operation to scale. The goal is to avoid bottlenecks in the data flow from the external staging files into the target table, namely:

- The access of the external staging files will perform and scale
- The proper setup of the target database to perform highly scalable and parallel operations
- The optimal parallel data loading for your business

This document will discuss these main three areas in more detail.

Prepare for data loading

Oracle's best performing approach to loading data in a high-performant parallel manner is through the use of flat files and external tables. An external table allows you to access data in external sources (staging files) as if it were a table in the database. This means that external files can be queried directly and in parallel using the full power of SQL, PL/SQL, and Java.

To load data you have to provide the following: one or more external data files (a.k.a. staging files) in a known format that is accessible from the machine(s) where the database instance(s) runs on, plus two objects in the database: a directory object, creating a logical pointer to a location outside the database where the external data files reside and an external table, holding the metadata of both the data format in the external files as well as the internal database representation of this "table".

Considerations for the external staging files

Location of the external staging files

Users should provide an external shared storage for the staging files. We refer to it as shared storage since on a system running Real Application Clusters (RAC) the storage has to be accessible from all cluster nodes. The IO throughput of the shared storage has a direct impact on the load speed: you will never be able to load data faster than it is read from the shared storage.

To guarantee optimal performance, the staging files should not be placed on the same disks that are used by RDBMS data files since reading the external data files will compete with the data insertion of the database. You might encounter an IO bottleneck if the external data files and the database files are not physically separated. This recommendation does not apply for the Oracle Exadata Database Machine: having the external data files staged within DBFS striped across the same Exadata Storage Servers than the database files provides sufficient IO capabilities and you will not encounter an IO bottleneck.

If the shared storage IO throughput is significantly lower than the ingest rate of the database consider compressing the external data files and pre-process the data before loading, as discussed in 'pre-processing of the data'. Note, however, that this approach represents a trade-off by substituting lack of IO bandwidth with additional CPU processing for the decompression of the data; it also imposes some restrictions on how to use parallelism, as discussed in the 'manual setup for parallelism' section later in the document.

Impact of the external data format on performance

The format of the data in the external staging files can have a significant impact on the performance of the data load. The impact mainly relates to the overhead of parsing the column formats and applying character set conversions; these kinds of operations require additional CPU resources that could have been used for data loading otherwise.

The following list describes the impact of common data format settings on the load performance, in approximate order of their impact on the load performance:

- Single-character delimiters for record terminators and field delimiters are faster to process than multi-character delimiters.
- Having the character set in the external data file match the character set of the database is faster than a character set conversion.
- Single-byte character sets are the fastest to process.
- Fixed-width character sets are faster to process than varying-width character sets.
- Fixed-length records are processed faster than records terminated by a string.
- Fixed-length fields are processed faster than delimited fields.
- Byte-length semantics for varying-width character sets are faster to process than character-length semantics.

Environments that control the format of the staging files should ensure to choose the most performant data format settings for their situation. If the format is pre-determined, it is often preferable that any transformation and conversion is done by Oracle, as part of the loading process, rather than by pre-processing the files manually prior to the load. This can be done either by using SQL within the database or by leveraging the pre-processing capabilities of external tables as part of the initial data access through external tables.

Pre-processing of the data

Pre-processing before the load operation cannot be avoided in all cases or is sometimes even a necessary part of the data flow of the data warehouse architecture. For example, when massive amounts of data have to be transferred over a slow network and compressing the data files is chosen to optimize the data transfer. Oracle Database 12c allows an external program to be specified as part of the external table definition. For example the unzip program can be invoked transparently as a pre-processor before consuming the data. With Oracle's built-in pre-processing in place the database is accessing the compressed external files and the de-compression will take place (outside the database) transparently as part of the data access by the database, without the need to re-stage the data in an uncompressed format before consuming it.

A requirement for the pre-processing program is that it accepts a single argument as input – the locations defined in the external table definition – and which sends its output to the standard output (stdout)[1]. Setting up the pre-processing of data is part of the external table definition, as discussed later in this paper.

[1] A more complex example of how to set up a pre-processing program is discussed in the appendix

[2] The size of 10MB is not fixed and adjusted adaptively in many cases, but still provides a good

While the decompression of external data files is the most common use case for pre-processing data, the framework is flexible enough to be used in other circumstances. For example, awk could be used to modify data of an external staging file.

Note that if the pre-processor functionality is used, the parallelization of the access of the underlying external staging files has some restrictions: for example, the content of a compressed file is not known unless the whole file is completely decompressed. As a consequence, only one parallel process can work on a single individual file. Thus the maximum degree of parallelism is constrained by the number of external files. To ensure proper parallelism, follow the best practices as discussed in the next section.

When to process data in parallel

Invoking parallel processing requires more internal "administrative work" of the database compared to processing data in serial. For example, if you want 128 processes working on a single task, Oracle has to allocate the processes, delegate work, and coordinate the parallel processing.

In the case of parallel loads a generic rule of thumb is that you should have at least approximately 30 seconds of work for every process involved to warrant parallel processing. For example, if you expect an average load processing of 150MB/sec per process, you should have approximately 30 sec * (150 MB/sec) = 4.5 GB per process to warrant parallel processing; otherwise serial processing is more suitable. Note that your mileage will vary because the data processing rate depends on the IO capabilities of you IO subsystem and also on the data format of the external data files.

Impact of the data format on parallel processing

In order to support scalable parallel data loads, the external staging files must be processed in parallel. From a processing perspective this means that the input data has to be divisible into units of work - known as granules that are then processed in parallel.

The database provides automatic parallelism out-of-the-box with no architectural limit of the parallel processing capabilities. However, under some circumstances the degree of parallelism is constrained, for example due to pre-processing of the data, in which case the user has to take some actions to ensure the degree of parallelism desired.

Automatic transparent parallelism

Whenever Oracle can position itself in any place in an external data file and be able to find the beginning of the next record, Oracle can build the parallel granules without any restrictions and automatic parallelism will take place; the degree of parallelism is not constrained by anything other than the requested degree of parallelism and you can have many parallel processes working concurrently on the same external file (intra-file parallelism). For example, when single-byte records are terminated by a well known character, for example a new line or a semicolon, you can choose any degree of parallelism, irrespective of whether you access one, two, or many data files.

With automatic parallelism, Oracle divides each external data file into granules of approximately 10 MB in size[2]. If there are multiple files, each is granulized independently and the granules are given to the access drivers[3] for scans in round robin fashion. For optimal performance you should have multiple files similar in size, with an approximate size that is a multiple of 10MB, and a minimal size in the order of few GB for parallel loading. Having multiple files may be advantageous if they can be located on independent disk drives to increase overall scan throughput. Otherwise, if scan IO throughput is not a bottleneck, there is no significant performance difference between a single large file and multiple smaller files.

Manual setup for parallel processing

When the format of the files prevents Oracle from finding record boundaries (to build granules) or when the type of media does not support position-able (or seekable) scans, the parallelization of the loading does not allow automatic transparent parallelism. Under these circumstances Oracle cannot break down an external staging file into granules, but has to treat each individual data file as a single entity – and therefore as a single granule. So the number of parallel processes that can work concurrently on a data load depend on the number of data files to being loaded. The parallelization of such data loads has to be done by providing multiple staging files, and the total number of staging files will determine the maximum degree of parallelism possible.

The following file formats require manual setup for parallelism

- Records are stored using VAR encoding
- Records use a multi-byte character sets and the end-of-record delimiter cannot be uniquely identified without processing the whole file
- Pre-processing of the data files is necessary, for example to deal with compressed or otherwise encoded data

The following resident media are also mandating manual setup for parallelism

- Tapes
- Pipes

For optimal performance in these scenarioes the size of the files should be similar to avoid skew in processing the data. For example, if we assume two external staging files, one of them being twice as big as the other, the parallel process working on the larger file will work approximately twice as long as the other process. In other words, the second process will be idle for half of the processing time.

---

[2] The size of 10MB is not fixed and adjusted adaptively in many cases, but still provides a good approximation for the purpose of this discussion.

[3] The access driver is the piece of code that does the external access of the data and makes it accessible for further processing inside the database.

Having three files similar in size and three processes working would cut the processing time in half in this scenario by investing 50% more resources.

For optimal performance, Oracle recommends to having N * DOP external staging files similar in size available for parallel processing where DOP is the requested degree of parallelism for the load and N > 1. This guarantees a balanced parallel data load and avoids a skew in the processing time of an individual parallel process. For example, loading with a parallel degree of 4 should have 8, 12, or 16 external data files or any other multiple of the number 4.

If you cannot ensure files similar in size, order the files in the LOCATION clause of the external table definition in descending order of their sizes to minimize skew in processing as much as possible[4].

Database objects necessary for data loading

Create directory objects

To define an external table users define **Oracle directory objects** that contain the paths to the **OS directories** holding the staging files, log files (information about the external data access is written there), bad files (information about data access errors is written there), and finally in Oracle 11.2 directories where the pre-processing programs reside, as shown in Figure 2.

For example

```
REM directory object to point to the external data files
CREATE DIRECTORY data_dir1 AS '/dbfs/staging/hourly/data';

REM for the log files of the external table access
CREATE DIRECTORY log_dir   AS '/dbfs/staging/hourly/log';

REM for the rejected records of the external table access
CREATE DIRECTORY bad_dir   AS '/dbfs/staging/hourly/bad';

REM directory to point to the pre-processing executable
CREATE DIRECTORY exec_dir AS '/myloadfiles/exec';
```

Oracle directory objects are important for Oracle controlled security. Appropriate privileges (READ, WRITE or EXECUTE) must be given to the Oracle directories for a database user to be able to read, write and execute files from them.

[4] A more thorough discussion of the most optimal layout of a data file can be found in the 'advanced topics' section.

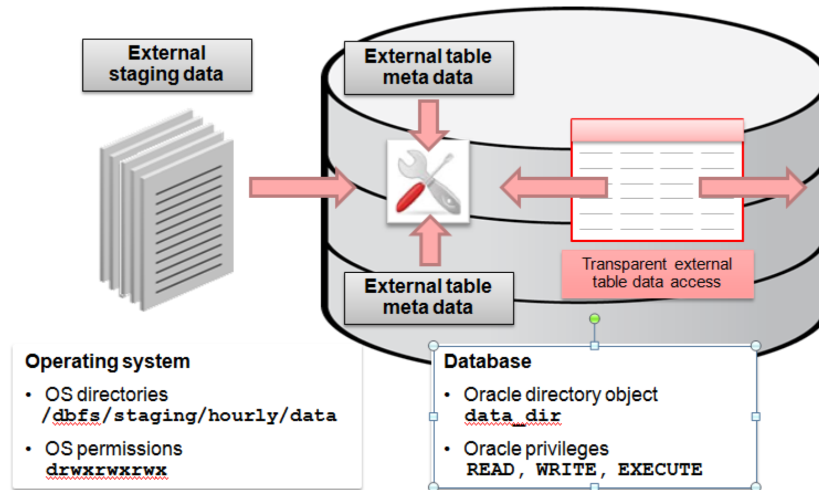

Figure 2: Operating system-database relationship for external tables

Recommendations for optimal setup:

- Staging files, logging files, bad files and pre-processor programs should all reside in different OS directories. This will allow DBA to create Oracle directory objects for each of the OS directories with the appropriate Oracle privileges (inside the database). The system administrator must make sure that OS-user 'oracle' (the owner of the Oracle software) has appropriate OS-permissions to read, write, write, and execute on the data, logging files, bad files, and execute images correspondingly.
- The DBA should grant to the loading database user READ privileges for the Oracle directories with staging files, WRITE privileges for Oracle directories with the logs and bad files and EXECUTE privileges to directories with pre-processor programs (if applicable).
- Any files generated by the access driver, including log files, bad files or discard files, should be written to an Oracle directory object that is separate from the Oracle directory objects containing staging files and directory objects containing the preprocessor. The DBA and the system administrator need to work together to create an Oracle directory and ensure it has the proper protections, both inside and outside the database (on the OS level). The database user may need to access these files to resolve problems in staging files; hence the DBA and OS system manager need determine a way for the database user to read them.
- Note that the DBA as well as any user with the CREATE ANY DIRECTORY or DROP ANY DIRECTORY privilege also has full access to all Oracle directory objects. These privileges should be used sparingly, if at all.

If the pre-processing functionality is leveraged, additional precautions should take place:

- The system administrator should always create a separate OS directory to hold the preprocessors; you might need multiple OS directories if different sets of Oracle users will use different preprocessors. The system administrator also must make sure that OS-user 'oracle' has appropriate OS-permissions to execute the preprocessor program.
- The system administrator should protect the preprocessor program from write access by any OS user to prevent an accidental overwriting or deletion of the program.
- The DBA should create an Oracle directory object for each of the OS directories that contain the preprocessor programs. The DBA should grant EXECUTE privilege to those Oracle directory objects only to the database users that require the use of those files.
- The DBA should NOT grant WRITE privilege on the Oracle directory object containing the preprocessor programs to anyone. This will prevent a database user from accidentally or maliciously overwriting the preprocessor program.

For example,

```
REM external data files – read only
GRANT READ ON DIRECTORY data_dir1 TO sh;
GRANT READ ON DIRECTORY data_dir2 TO sh;

REM log and bad files – write only
GRANT WRITE ON DIRECTORY log_dir TO sh;
GRANT WRITE ON DIRECTORY bad_dir TO sh;

REM pre-processor executable – read and execute
GRANT READ EXECUTE ON DIRECTORY exec_dir TO sh;
```

Create an External Table

An external table is created using the standard CREATE TABLE syntax except it requires an additional clause, ORGANIZATION EXTERNAL. This additional clause specifying the following information (which is necessary to access the external data files):

- Type of the access driver used for reading the external staging files. It can be ORACLE_LOADER or ORACLE_DATAPUMP. For data warehousing loading the most common case is using ORACLE_LOADER.
- Access parameters that define the behavior of the access driver, including the description of the external data format, for example record delimiters, field separators, and default value for missing fields.
- The directory objects that contain the staging files and define the location for the log and bad files for the load.
- The definition of the columns as seen by the database.

The following SQL command creates an external table pointing to two external staging files (‘sales_1.csv’,’sales_2.csv’) located in directory ‘data_dir1’. The additional metadata for the external table is shown in bold:

```
CREATE TABLE sales_external (…)
ORGANIZATION EXTERNAL
(  TYPE ORACLE_LOADER
   DEFAULT DIRECTORY data_dir1
   ACCESS PARAMETERS
   (  RECORDS DELIMITED BY NEWLINE
      BADFILE bad_dir: 'sh%a_%p.bad'
      LOGFILE log_dir: 'sh%a_%p.log'
      FIELDS TERMINATED BY '|'
      MISSING FIELD VALUES ARE NULL )
      LOCATION ( data_dir1:'sales_1.csv',
              data_dir1:'sales_2.csv' ))
PARALLEL
REJECT LIMIT UNLIMITED;
```

If the staging files were compressed and the pre-processing option would have to be leveraged, the syntax would change as follows. The additional pre-processing clause is shown in bold:

```
CREATE TABLE sales_external (…)
ORGANIZATION EXTERNAL
(   TYPE ORACLE_LOADER
   DEFAULT DIRECTORY data_dir1
   ACCESS PARAMETERS
   (  RECORDS DELIMITED BY NEWLINE
      PREPROCESSOR exec_dir: 'zcat'
      BADFILE bad_dir: 'sh%a_%p.bad'
      LOGFILE log_dir: 'sh%a_%p.log'
      FIELDS TERMINATED BY '|'
      MISSING FIELD VALUES ARE NULL )
          LOCATION ( data_dir1:'sales_1.csv.gz',
                     data_dir1:'sales_2.csv.gz' ))
PARALLEL
REJECT LIMIT UNLIMITED;
```

For more information on creating external tables refer to the Oracle Database 12c Utilities Guide.

Database parameter settings for optimal load performance

If you want to load data in parallel, you not only have to ensure that your hardware is capable of performing a parallel load without any significant bottleneck, you have to ensure that the database leverages all of the available resources, potentially even for a single load operation[5].

On a balanced system, data loading will normally become CPU and memory bound. Consequently, the database settings for parallelism and memory allocation have a large impact on the load performance.

The following section will discuss the basic settings to ensure parallel loading.

Parallel settings of the database

A sufficient number of parallel execution servers have to be provided by the system. The following is recommended to ensure optimal parallel load, assuming that your hardware is providing the necessary resources for your planned parallel processing (note that all parameters are instance-specific and have to be aggregated in a cluster environment):

- `PARALLEL_MIN_SERVERS:` Controls the minimal number parallel execution servers that are spawned at instance startup and made available all the time. Note that in a Real Application Clusters environment, the total number of processes in the cluster that are spawned at startup time is the sum of all parallel_min_servers settings. To ensure the startup of a single parallel load can occur without a delay for process allocation, you should set this value to a minimum of 2*<your "normal" degree of parallelism for data load>.
- `PARALLEL_MAX_SERVERS`: Controls the maximum number parallel execution servers that are provided by an instance; this number will never be exceeded. Note that in a Real Application Clusters environment, the total number of available processes in the cluster is the sum of all parallel_max_servers settings. Leave this parameter as it is set by default by the Oracle database.[6].
- `PARALLEL_EXECUTION_MESSAGE_SIZE`: The message buffer size used for communication between parallel execution servers. This parameter should be kept at 16K, the default in Oracle Database 12c.
- `PARALLEL_DEGREE_POLICY:` To leverage Oracle's new automatic parallelism, you should set this parameter to AUTO.
- `PARALLEL_DEGREE_LIMIT:` The maximum degree of parallelism used when Auto DOP is enabled. If you are using AutoDOP, ensure that the setting of this parameter does not limit the requested degree of parallelism. For example, if this parameter is set to a value of 16, no operation

[5] This paper assumes a balanced hardware system that is capable to leverage parallel processing.

[6] We assume that your hardware platform is capable of supporting the default number of parallel processes

running under Auto DOP will be capable of allocating more parallel execution servers, irrespective of the size of the operation.

- `PARALLEL_SERVER_TARGET`: When using AutoDOP, the maximum number of parallel execution servers that can be used before statement queuing kicks in. When AutoDOP is used exclusively, then this parameter should be set close to PARALLEL_MAX_SERVERS, e.g. 90%+ of its value.

For a discussion of Oracle's parallel execution architecture and all parallel initialization parameters, see the Parallel Execution Fundamentals in Oracle Database 12c on oracle.com

Memory Settings of the database

On typical systems deployed for data warehousing, you normally see a minimum of 4GB of physical memory per core. Many load operations consume a lot of memory for things like the message buffers for the parallel execution servers' communication, memory buffers for compressing data, or the load buffers for data insertion (which increases with the degree of parallelism and the number of partitions to being loaded). In the case of automatic index maintenance you may also need a significant amount of memory for data sorting. Having a per-core memory less than 4GB can lead to suboptimal load performance.

An Oracle database has two main memory areas; the SGA (System Global Area) and the PGA (Program Global Area). The SGA is a group of shared memory structures that contain data and control information for one Oracle Database instance. The SGA is shared by all server and background processes. Examples of data stored in the SGA include cached data blocks and shared SQL areas. The PGA is a non-shared, private memory region that contains data and control information exclusively for use by an Oracle process. The PGA is created by Oracle Database when an Oracle process is started. The collection of individual PGAs is the total instance PGA.

In data warehousing environments the usage of PGA is much more important than the usage of the SGA, especially where the amount of data stored far exceeds the amount of memory available. For optimal performance, it is therefore recommended to start with the following simple memory setting rules:

- `PGA_AGGREGATE_TARGET`: This parameter controls the amount of memory that is available for all private process operations, such as aggregations, sorts, or load buffers. Allocate a minimum of 60% of the available memory for Oracle to the PGA.
- `SGA_AGGREGATE_TARGET`: This parameter sets the total size of the SGA, including components such as the shared pool and the buffer cache. Allocate the rest of 40% of the available memory for Oracle to the SGA.
- `DB_CACHE_SIZE`: For data warehousing workloads caching of database blocks is not critical since parallel query uses direct IO rather than IO via a buffer cache.

For a complete discussion of Oracle's memory architecture and the various components, see the 'memory management' section in the Oracle Database Administrator's Guide.

Ensure sufficient parallelism for the load operation

In addition to having the database set up for parallel processing you have to ensure that your actual load operation runs in parallel. Doing so can either be done by manually setting the requested degree of parallelism for a given object, session, or statement - or by leveraging Oracle's Automatic Parallelism capabilities (Auto DOP).

With Auto DOP, the database decides if a statement (Query, DML, or DDL) should use parallelism and what degree of parallelism will be chosen for a given operation. The chosen degree of parallelism is derived from the statement's resource requirements; statements requiring little resources will execute serially, while once needing significant resources will run with a potentially high degree of parallelism.

For optimal performance it is recommended that data warehousing loads are run with either Auto DOP enabled or by manually setting a degree of parallelism for the load operation. How to activate parallelism is described in more detail later in this document. Loading external data files through external table is supported with the new Auto DOP functionality. If no statistics are present on the external table, the Optimizer takes the total size of the external staging files into account for the derivation of the degree of parallelism. In addition, ensure that the user that is performing the actual load operation is not constrained by any database or instance setting that prevents the load operation from getting the necessary (or requested) resources, such as a conflicting Database Resource Manager directive.

For a detailed discussion of Oracle's parallel execution architecture, see the Parallel Execution Fundamentals in Oracle Database 12c on oracle.com.

Parallel loading in a Real Application Clusters (RAC) environment

In a RAC environment, multiple nodes (machines) are clustered together in a shared everything architecture where every node can access all the data. Loading data in a RAC environment offers you choices of how to load the data in parallel: you can run the parallel data load (and all its processes) on a single node, on the whole cluster, or only on a subset of the nodes of a cluster.

Parallel load from multiple nodes

By default, all participating nodes (instances) of a RAC environment are eligible to run a given parallel operation. Out of the box, the Oracle database will allocate parallel execution servers on the whole system with the premise to (A) ensure an equally loaded system and (B) to ensure that as many processes as possible run locally together on a single node.

Oracle RAC services[7] can be used to limit a parallel operation to run only on a subset of database nodes. Constraining the number of nodes participating in a parallel operation has an impact on the total number of available parallel server processes and therefore the maximum degree of parallelism possible. For optimal performance you have to include as many nodes as necessary to achieve the requested degree of parallelism.

As discussed earlier in the 'considerations for external data files' section, a load involving multiple nodes in a RAC environment requires the usage of a cluster file system (or an externally mounted device, e.g. through NFS) to ensure that all nodes participating in the load are capable of accessing the same external staging files. For optimal performance you have to ensure that the storage hosting the external staging files is capable of satisfying the aggregated IO requirements of all loading processes across all nodes. On Oracle Exadata Database Machine DBFS is the most commonly used shared file system.

If multiple nodes are participating in a parallel load, some data or message transfer between the varying nodes will become necessary. This communication can be as little as synchronizing the space management of the various participating nodes or as much as redistributing the whole data set, for example for automatic index maintenance or the load into partitioned tables. For optimal performance you have to provide a sufficiently sized RAC interconnect, ideally providing the same IO throughput characteristics as your IO subsystem[8]. If your RAC interconnect is undersized then do not consider a parallel load across multiple nodes; your load is unlikely to scale.

Parallel load from a single node

[7] A detailed description of how to leverage services in a Real Application Clusters environment can be found in the Real Application Clusters Administration and Deployment Guide

[8] The Oracle Exadata Database Machine is an example that has these characteristics

For systems with an undersized interconnect or systems with large individual nodes (a.k.a. “fat nodes’, a system with many CPUs and a large amount of memory) you might want to consider loading from an individual node only. Loading from an individual node minimizes the interconnect communication to a bare minimum, down to zero in the most optimal case.

Considerations for the target table

An important aspect for loading - and querying - large amounts of data is the space management used for DW tables. You want to ensure to load data as fast and efficient as possible, while guaranteeing a physical data storage that is not detrimental to the future data access. Space management is either controlled on a table level or inherited from the tablespace level where the table (or partition) will reside later on.

Tablespace attributes

Oracle recommends the usage of BIGFILE tablespaces to limit the number of data files that have to be managed by the data warehouse administrator. BIGFILE tablespaces are locally managed tablespaces and any space allocation should be done leveraging Automatic Segment Space Management (ASSM). Oracle also provides older forms of segment and space management – namely dictionary managed tablespaces and manual segment space management - but those are less efficient than locally managed tablespaces and not recommended for DW.

Locally managed tablespaces can have two types of extend management: AUTOALLOCATE and UNIFORM. With AUTOALLOCATE, the database chooses variable extent sizes based on the property and the size of an object while the UNIFORM setting enforces extents with a pre-defined fixed size. Oracle recommends using the AUTOALLOCATE extent policy.

For performance reasons, most load operations are leveraging a direct path operation where the load process directly formats and writes Oracle blocks instead of going through the buffer cache. Such a loading process (or multiple one in the case of a parallel load) allocates extents and fills them with data as it loads. If a small amount of data is loaded the UNIFORM allocation may cause space wastage since extents may not be fully filled. This effect is amplified for a parallel load with a high degree of parallelism where many loader processes insert data; every loader process will allocate its own extent of UNIFORM size, and you might end up with many partially-filled large extents. This not only wastes space, it also will negatively affect subsequent queries that scan the table since they have to scan the whole table, including all extents that are not fully filled. AUTOALLOCATE on the other hand will dynamically adjust the size of an extent and trim the extent after the load in case it is not fully loaded. By default Oracle starts allocation of space conservatively with a small extent size for nonpartitioned

tables (64k) and a larger extent size for partitioned tables (8M); the extent size will then gradually increase based on the observed allocation patterns and the size of an object[9].

Table attributes

In cases where you are loading large volumes of data in a highly parallel fashion into an empty table, the default allocation policy of AUTOALLOCATE might be too conservative since it implies too many small initial extents. You should manually set a larger initial extent size at table creation time; it is recommended to set the initial and next extent size to at minimum 8M for tables that will become large, even for nonpartitioned tables.

Other considerations

Minimize the impact of space management

When a tablespace is created its data files are formatted. Each time a data file is extended, the extended portion must be formatted. This is an expensive and operations and it serializes space allocations – all processes that want to allocate more space than currently available have to wait for the process that initiated the data file extension. Furthermore, formatting is done serially (unless you are on an Oracle Exadata Database Machine where each Exadata Storage Server performs the formatting of its disks independently).

For optimal performance we recommend to minimize space allocation operations during the data loading process by pre-allocating large sized data files (in orders of terabytes) prior to the load. Alternatively, specify a large auto-extension size for the data files to minimize the space operations for extending data files; the time between such operations should never be measured in seconds.

For example, the following creates tablespace TS_DATA applying the above-discussed best practices:

```
CREATE BIGFILE TABLESPACE ts_data
DATAFILE '/my_dbspace/ts_data_bigfile.dbf' SIZE 1500G REUSE
AUTOEXTEND ON NEXT 15G MAXSIZE UNLIMITED
LOGGING                              -- this is the default
EXTENT MANAGEMENT LOCAL AUTOALLOCATE –- this is the default
SEGMENT SPACE MANAGEMENT AUTO;        -- this is the default
```

Minimize the impact of space quotas

A large data load consumes a lot of space and therefore has to allocate many extents. This stresses updates to the tablespace accounting information in cases where a user has a tablespace quota enabled. Tablespace quotas restrict the amount of space a user can consume in a given tablespace, so for every

[9] The larger initial extent size for partitioned tables together with deferred segment creation represent the optimal space allocation approach for tables with sparsely populated partitions or subpartitions

requested space management operation, the system has to ensure that a user will not go over quota. While in a single user case this does not sound like a lot of work, remember that for a massively parallel load, a single user is represented by many parallel server processes, each of them doing space management.

Using a large initial extent size (8MB) already reduces updates to this accounting information. However, the need to update the tablespace quota information can be eliminated completely if the tablespace quota is set to UNLIMITED for the user doing the load operation or if this user has the UNLIMITED TABLESPACE privilege. The former can be done using “ALTER USER user_name TABLESPACE QUOTAS ts_name UNLIMITED”, for example:

```
ALTER USER sh QUOTAS UNLIMITED ON ts_data;
```

Granting unlimited quota to the user that owns the object to being loaded is recommended for performing large loads.

Consider the usage of NOLOGGING

If the redo recovery of the table after load is not critical, we recommend to use the CREATE TABLE or INDEX with the NOLOGGING clause. This minimizes the redo information generated during direct loads[10]. For an index, Oracle minimizes redo logging only for CREATE INDEX and it logs redo for index maintenance during a direct load INSERT. Conventional DML statements are unaffected by the NOLOGGING clause and generate redo. Note that using the NOLOGGING clause will disable the table from being recovered through normal database backup and recovery procedures, but such operations can be very well integrated and coped with by following the Maximum Availability Architecture by leveraging RMAN’s incremental backup capabilities after running an operation with NOLOGGING. For further details see the Oracle Maximum Availability Architecture (MAA) page on oracle.com.

Consider the usage of compression

Loading large volumes of data always beg the question of whether or not to compress the data right at data load time. However, any kind of compression always imposes additional CPU processing on the loading process. Since loads most commonly are CPU and memory bound, applying compression always has a negative impact on the load performance.

You have to make the decision between maximum performance and immediate space saving (note that compression also provides a benefit for scanning the data since less data has to be read from disk).

For optimal load performance consider loading the data in an uncompressed format and to apply compression later on in the lifecycle of a table (or partition). Especially with Oracle Partitioning the

[10] Direct loads (a.k.a.[10] direct path operations) are invoked by either using a CREATE TABLE AS SELECT or an INSERT /*+APPEND*/ statement.

separation between "older" data and the new data to being loaded is an easy accomplishment: the majority of the older existent data is stored in a compressed format in separate partitions (or subpartitions) while the most recent partition(s) are kept in an uncompressed format to guarantee optimal load performance. For example, you can load the data of the most recent day into an uncompressed partition while the rest of the table is stored in a compressed format; after some time this partition can also be compressed (and be potentially moved to a different storage tier) by using an ALTER TABLE MOVE PARTITION [ONLINE] command.

If you want to load data in a compressed format right away you simply have to declare the target table (or partitions) as COMPRESSED to reduce space and scan cost. Oracle offers the following compression algorithms:

- COMPRESS/COMPRESS FOR DIRECT_LOAD OPERATIONS – block level compression, for direct path operations only
- COMPRESS FOR ALL OPERATIONS – block level compression, for direct path operations and conventional DML, part of the Advanced Compression Option
- COMPRESS FOR [QUERY|ARCHIVE] [HIGH|LOW] – columnar compression, for direct path operations only, feature of Exadata Database Machine

Irrespective of what compression is applied, no further action than declaring an object as compressed is necessary. For example, it is not necessary to impose ordering of columns in the CREATE TABLE statement for compression. Compression permutes columns within blocks automatically to get the best compression ratio for block level compression algorithms[11]. Furthermore, with Oracle Exadata Hybrid Columnar Compression, the storage format will be transparently converted to a hybrid columnar format.

Load the data

DML versus DDL

Data can be loaded using DDL or DML operations. In the case of a DDL statement, the target table will be created as part of the load process itself using a CREATE TABLE AS SELECT (CTAS) command. In the case of using a DDL statement, the target table is always empty (since it is not existent prior to the "load"), and there will be no indexes, thus no index maintenance[12]. A CTAS operation does a direct path insert (a.k.a. direct load) where the load process directly writes formatted

[11] Note that the efficiency of compression can potentially increase with the size of the data blocks since compression is performed with it.

[12] The only exceptions are primary and unique constraints when defined as part of the CTAS command; these constraints require index maintenance for constraint enforcement.

Oracle data blocks on disk; it bypasses the buffer cache and the conventional SQL layer and is the fastest way of loading data.

Alternatively, data can be loaded into an existing table, either using an INSERT APPEND or MERGE APPEND statement; the target table can either be empty or already contain data, and it can have indexes. Using the APPEND hint invokes a direct load operation for DML. Note that in the case of a MERGE statement, only the insert portion of the statement will be a direct load operation; the update of already existent data will be done using a conventional update operation since you actually change existing records.

For performance reasons it is recommended to load data using a direct load operation whenever possible.

Locking during Load

During a direct path load operation or any parallel DML, Oracle locks the target table exclusively. If you specify a single partition to be loaded using the partition extended syntax, only this partition is locked. Locking prevents other DML or DDL operation against the table or its partition; however, the data in the table is fully accessible for queries of other sessions.

If a load occurs into a single top-level partition of a simple or composite (two-level) partitioned table, it is recommended to indicate the partition using the partition extended syntax to reduce locking.

Activate parallelism for a load operation

Parallelizing the load

After ensuring that a database environment provides sufficient parallelism for your operation, as discussed earlier in this paper, you have to ensure that your load is actually running in parallel.

If you use Auto DOP, the DOP will be automatically derived by Oracle based on the more expensive operation of the statement: the SELECT or the INSERT (CREATE TABLE) portion. If you are using manual DOP, you have to ensure the desired degree of parallelism will be used by the SELECT and/or the INSERT portion of the statement. The SELECT portion is derived following the rules for parallelizing a SQL operation[13], and the degree of parallelism for the INSERT (CREATE TABLE) portion is derived by the parallel decoration in the CREATE TABLE command or the parallel decoration of the target table respectively. For the statement overall, the higher degree of parallelism for both the SELECT and INSERT branch is chosen.

[13] For rules of how to parallelize a query, see the Parallel Execution Fundamentals in Oracle Database 12c on oracle.com

A sample CTAS statement could look as follows:

```
CREATE TABLE <target_table>
PARALLEL 8
AS SELECT /*+ PARALLEL(64) */ *
   FROM <external_table>;
```

This CTAS command will run with a degree of parallelism of 64, assuming the system provides 64 parallel server processes and the user session is not constrained to a lower degree of parallelism.

Parallelizing DDL

A CTAS operation can automatically run in parallel. As soon as you either enable AutoDOP or follow the rules of manual parallelization, the operation will run in parallel, for both the SELECT and the CREATE TABLE part of the statement.

Parallelizing DML

By default, DML operations will not run in parallel; you have to explicitly invoke parallel DML capabilities for the loading session by issuing an ALTER SESSION command. Note that this command has to be the first operation of a transaction and has to be committed before you can access the loaded data. A sample SQL flow to insert data into a test table in parallel would look as follows:

```
ALTER SESSION ENABLE PARALLEL DML;
INSERT /*+ APPEND */ INTO test
SELECT * FROM <external_table>;
COMMIT;
```

The degree of parallelism of this operation is chosen as discussed in the previous section.

Partition Exchange Loading

The majority – if not all - Oracle data warehousing environment leverage Oracle Partitioning for their larger tables, for both performance and manageability reasons. In the case of data insertion into a partitioned table, the data can be inserted into the target table right away or “inserted” by leveraging the ‘Partition Exchange Loading’ (PEL) technique. Unlike a direct load into the target table, PEL offers a greater degree of flexibility in terms of index maintenance and “point of data publication” for the end users (the moment an end user can see the data). PEL provides a framework to “load” data quickly and easily with minimal impact on the business users by using the EXCHANGE PARTITION command. It allows you to swap the data of a standalone non-partitioned table into a particular partition in your partitioned target table. No data is physically moved - the data dictionary is simply updated to reset the pointer from the partition to being exchanged with the pointer of the standalone non-partitioned table to being loaded and vice versa. Because there is no physical movement of data, a partition exchange does not generate any redo and undo, making it a sub-second operation and far less likely to impact performance compared with any traditional data-movement approaches such as INSERT.

While it is obvious that the data has to be loaded initially before you can actually do a partition exchange (you won't get around loading the data), there are a number of benefits:

- **Flexible data loads**: Since the data is loaded into a standalone table, you have more flexibility for how and when you are loading and preparing your data. For example, you can start loading data as soon as it arrives from a source system without the risk of making the newly inserted data available to the end users too early. Unless you do the final partition exchange command, no data is visible to the end user. For example, you can load data in several individual transactions and run additional data validity checks before doing the final exchange operation.
- **Better control of data publication**: The data "loading" is a matter of sub-seconds, compared to the time needed to do the full data load into the target table. This allows a more precise data publication and provides better control. For example, you can issue the partition exchange at a given point in time and do not have to rely on the commit of the load, whenever it might take takes place.
- **Flexibility for index maintenance**: A partition exchange operation requires both the partitioned table and the standalone table to have the same logical shape. This includes any indexes. However, since the indexes have to be present only at the time when the actual partition exchange is taking place, you can decouple the index maintenance (or creation) from the initial data load if required.

The following illustrates a sample Partition Exchange Loading scenario.

Let's assume we have a large table called SALES, which has daily range partitions and is sub-partitioned by hash on the cust_id column. At the end of each business day, data from the online sales system needs to be loaded into the SALES table in the warehouse.

The staging table would be created using the following command:

```
CREATE TABLE tmp_sales (…)
TABLESPACE ts_data
STORAGE (INITIAL 8M NEXT 8M)
PARALLEL
NOLOGGING
SUBPARTITION BY HASH (cust_id) PARTITIONS 8;
```

Note that the logical shape of this table matches the shape of the target table and that the partitioning strategy (hash) matches the sub-partitioning strategy of the target table.

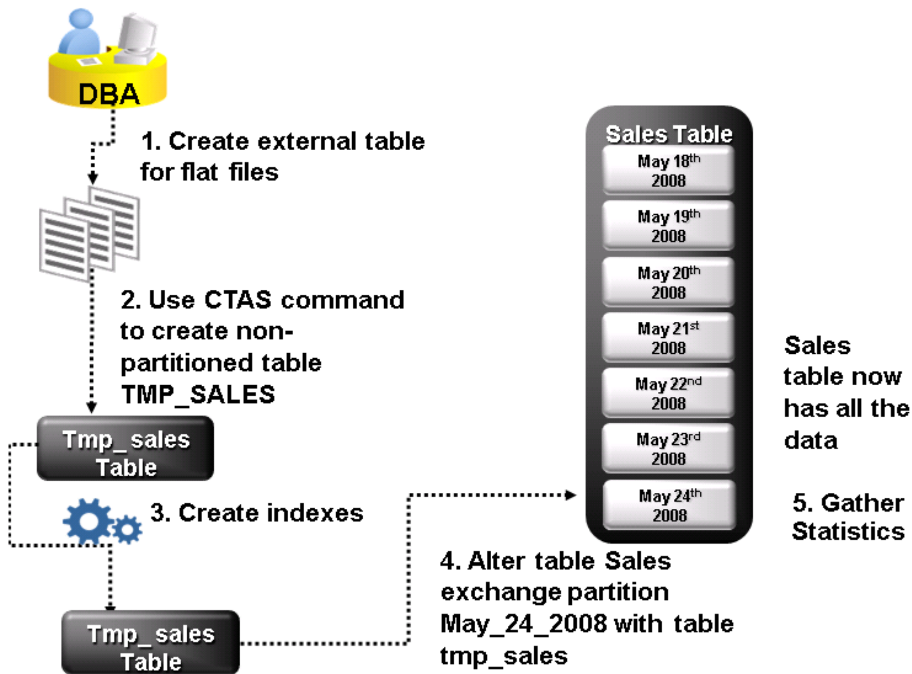

Figure 3: Steps of a partition exchange load

This staging table could have been loaded during the CREATE TABLE command if we had added an 'AS SELECT' clause or after its creation using one or multiple DML commands, like:

```
INSERT /*+ APPEND */ INTO tmp_sales
SELECT * FROM sales_external_day_2014_JAN_01;
```

After the data is loaded into the staging tables, local indexes corresponding to those on the target table SALES have to be built on the staging table, e.g.

```
CREATE BITMAP INDEX sales_prod_bix_stage ON tmp_sales(prod_id)
TABLESPACE ts_index PARALLEL NOLOGGING COMPRESS;
```

You could have chosen to create all indexes right away prior to the data loading and rely on Oracle's automatic index maintenance while loading the data. However, in the case of many indexes the index creation after data loading is recommended to optimize the resource usage – specifically memory – by creating all indexes individually at the end of the data load. Unlike doing index maintenance as part of the data load, another benefit of creating the indexes afterwards is that a CREATE INDEX command automatically and transparently collects the index statistics as part of the index creation. In addition, a subsequent CREATE INDEX can be done in NOLOGGING mode to avoid redo generation; an index maintenance as part of the data load is done using conventional index maintenance and requires redo and undo logging.

Finally the staging table is exchanged into the target table:

```
ALTER TABLE sales EXCHANGE PARTITION day_2014_jan_01 WITH tmp_sales INCLUDING
INDEXES;
```

The former standalone hash partitioned table now becomes a range partition in our target composite range-hash partitioned table.

Note that by default Oracle is recursively checking the validity of the table to ensure that its data content does not violate the partition boundaries. If it is guaranteed that the staging table does not violate the partition key mappings of the exchange partition, the WITHOUT VALIDATION option can be used to speed up the exchange, for example

```
ALTER TABLE sales EXCHANGE PARTITION day_2014_jan_01 WITH
tmp_sales INCLUDING INDEXES WITHOUT VALIDATION;
```

Use the without validation clause with great caution: you can introduce data integrity violations.

After exchange, statistics should be collected on the target table as explained in Post-load activities section below.

Index Maintenance during load

Direct load operations using external tables – serial or parallel – can maintain global and local indexes while loading data. Internally, the index maintenance is delayed until after all data is loaded but before committing the transaction and making the loaded rows visible[14].

However, depending on whether you are loading data into an empty table or into a table that already contains data, your index strategy should vary, irrespective of what is technically doable.

To achieve optimal performance you should apply the following recommendations

- If you are loading data into an initially empty table, you should always create any index after the data load.
- If you are loading data into a table that already contains data, you should consider invalidating indexes and rebuilding them after the load as discussed below.

The decision to invalidate indexes prior to loading depends on type of the indexes (btree vs bitmap), the number of indexes, the percentage of data added or changed during the load, and the requirement of index availability. Choosing index availability is most likely a trade-off with performance. However, if the business requirement is to have all indexes available at any given point in time, you will load the

---

[14] Note that while this is true for parallel operations inside the database, it does not hold true for SQL*Loader. SQL*Loader imposes a lot of restrictions and requires invalidation of indexes in many cases, like a parallel load.

data and leverage the automatic index maintenance of a direct load operation and keeping all indexes available all the time, irrespective of how long it takes.

If optimal performance is the goal and index availability is **not** a business requirement, the following is recommended:

Btree indexes

If there is only a small number of indexes present and if we are loading more than approximately 5% of the size of a table or partition, it is beneficial from a performance perspective to invalidate local indexes and to rebuild them after load. For a significant number of btree indexes, invalidate and rebuilt is always recommended; the time to maintain indexes does not scale linearly with their number. Oracle sorts the inserted data in one chunk on all the columns of all indexes, and sort operations do not scale linearly with their size (note that this statement is based on the assumption that we have sufficient IO to re-scan the data for the rebuild - and we do this n times for n indexes if you rebuild them after a load)

Bitmap indexes

It is recommended to invalidate and rebuild bitmap indexes regardless of their number or size of the load provided we load more than 1MB.

For example, to mark all local index partitions of our table SALES as unusable:

```
ALTER TABLE sales
MODIFY PARTITION day_2008_JAN_01
UNUSABLE LOCAL INDEXES;
```

To rebuild these local index partitions after the data load:

```
ALTER TABLE sales
MODIFY PARTITION day_2008_JAN_01
REBUILD UNUSABLE LOCAL INDEXES;
```

Analyzing and monitoring your load operation

Using EXPLAIN PLAN

To analyze a load operation you can analyze it prior to running it, using EXPLAIN PLAN.

There are four important items in an EXPLAIN PLAN for a load statement: whether (or not) a load runs in parallel, the degree of parallelism used, , the estimated size of the load, and the distribution method. For example given this load statement:

```
EXPLAIN PLAN FOR
INSERT  /*+ APPEND */ INTO sales
SELECT  *
FROM   sales_external;
```

Its EXPLAIN PLAN may look as follows:

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

---------------------------------------------------------------------------------------
|Id| Operation                      | Name        | Bytes|    TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------
|0 | INSERT STATEMENT               |             |  100G|        |      |            |
|1 |  PX COORDINATOR                |             |      |        |      |            |
|2 |   PX SEND QC (RANDOM)          | :TQ10000    |  100G|  Q1,00 | P->S | QC (RAND)  |
|3 |    LOAD AS SELECT              | SALES       |      |  Q1,00 | PCWP |            |
|4 |     PX BLOCK ITERATOR          |             |  100G|  Q1,00 | PCWC |            |
|5 |      EXTERNAL TABLE ACCESS FULL| SALES_EXTER|  100G|  Q1,00 | PCWP |            |
---------------------------------------------------------------------------------------

Note
-----
   - Computed Degree of Parallelism is 24
   - Degree of Parallelism of 24 is derived from scan of object
     SH.SALES_EXTERNAL
```

Figure 4: Sample execution plan for a simple load operation

Ensure that the actual data load is running in parallel (Id #3 in Figure 4). You can do this by ensuring that you do not see a parallel-to-serial distribution before the load actually takes places ('P->S' in the 'IN-OUT' column for Ids higher than the load Id). You can also verify a parallel load by having the final 'PX SEND QC' operation (Id #2 in Figure 4) above the load operation.

The plan output will show you the actual degree of parallelism the statement will run with. In this example the DOP was determined by Auto DOP based on the size of the SALES_EXTERNAL table; you will get appropriate notes when the degree of parallelism is chosen manually and will also see whether the degree of parallelism was constrained, for example through the setting of parallel_degree_limit. Ensure that the degree of parallelism as seen in the Notes section is as requested (or desired). If you are not using AutoDOP, the explain plan shows the estimated number of bytes to be loaded; this can be used for verification of the manual computation of a degree of parallelism, as discussed in section 'When to process in parallel'.

For the plan in Figure 4 no data redistribution has taken place between reading the data (SELECT) and inserting the data (LOAD); the same parallel server process that is doing the scan is inserting its portion of the data into the SALES table. You can identify this COMBINED operation by analyzing the 'PQ Distrib' (distribution) column: the only "distribution" you are seeing is 'QC (RAND)' which means the parallel server processes are communicating with the QC (Query Coordinator) that they finished their portion of the workload, and they do this communication in a random fashion.

If we had a data distribution between the read and the insert operation, it could look as follows:

```
----------------------------------------------------------------------------------------
|Id| Operation                         | Name       | Bytes|     TQ |IN-OUT| PQ Distrib  |
----------------------------------------------------------------------------------------
|0 | INSERT STATEMENT                  |            |  100M|        |      |             |
|1 |  PX COORDINATOR                   |            |      |        |      |             |
|2 |   PX SEND QC (RANDOM)             | :TQ10000   |  100M| Q1,00  | P->S |  QC (RAND)  |
|3 |    LOAD AS SELECT                 | SALES      |      | Q1,00  | PCWP |             |
|4 |    PX RECEIVE                     |            |  100M| Q1,01  | PCWP |             |
|5 |      PX SEND RANDOM LOCAL         | :TQ10000   |  100M| Q1,00  | P->P | RANDOM LOCAL|
|6 |       PX BLOCK ITERATOR           |            |  100M| Q1,00  | PCWC |             |
|7 |        EXTERNAL TABLE ACCESS FULL| SALES_EC |  100M| Q1,00  | PCWP |             |
----------------------------------------------------------------------------------------

Note
-----
   - Computed Degree of Parallelism is 24
   - Degree of Parallelism of 24 is derived from scan of object
     SH.SALES_EXTERNAL
```

Figure 5: Sample execution plan for a simple load operation with data redistribution

In Figure 5 you see data redistribution between the read and the insert of the data, in Id #5: a parallel-to-parallel redistribution is taking place, using a RANDOM-LOCAL distribution mechanism[15]. A reason for data distribution could be for example the load into a partitioned table.

Monitoring the load operation

When a load operation is taking place, you can use SQL Monitoring to analyze the ongoing load and to track the progress in real time. Figure 6 shows the output of a sample load operation.

When a parallel load operation is ongoing, the CPU and IO utilization should be monitored. If default DOP or AutoDOP was used, then a significant CPU utilization (>90%) should be seen uniformly across all CPUs since loads are typically CPU bound. There should be no significant skew of CPU utilization. If there is less then 90% of CPU utilization and high IO utilization, this is a sign that system may not have enough IO bandwidth. If there is a skew of individual CPU utilization, this can be a sign of clustering of external data on partition key. See Appendix B, impact of data skew for further details. In this case it might be beneficial to enforce random local distribution through a statement hint.

[15] For rules of how to parallelize a query, see the Parallel Execution Fundamentals in Oracle Database 12c on oracle.com

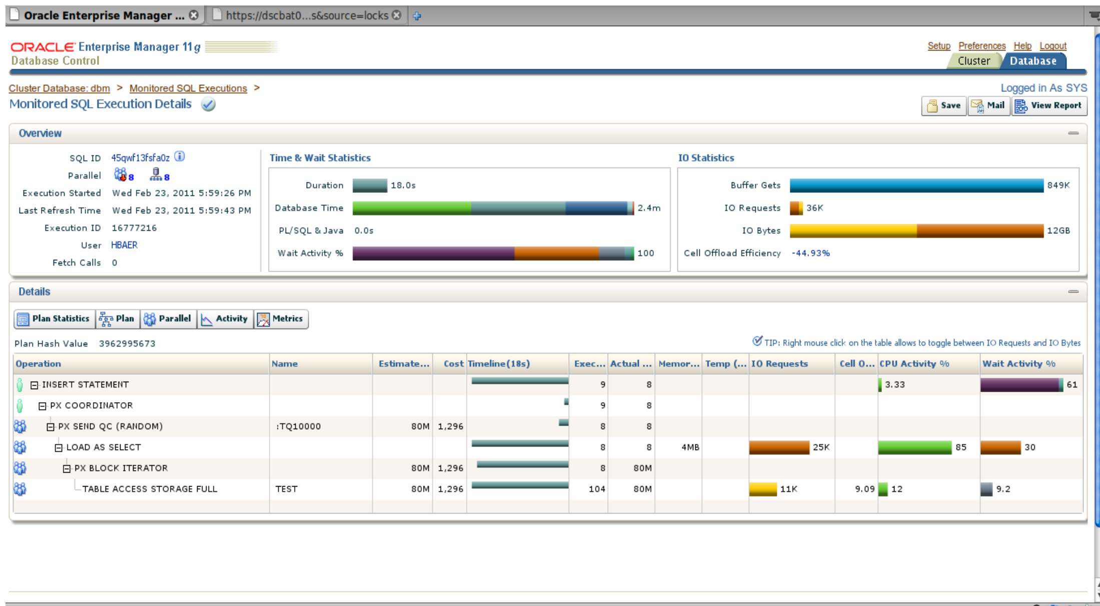

Figure 6: Sample execution plan for a simple load operation with SQL Monitoring

Post-load activities

Statistics maintenance

The Oracle Optimizer relies on accurate statistics, so whenever you add a significant amount of data to a table you should update its statistics; by default the statistics of a table are considered 'STALE' when you change more than 10 % of its data content. Note that "your mileage will vary" and that you will encounter situation where you may want to collect statistics earlier, e.g. when the cardinality or the min-max boundaries of a column change dramatically.

For non-partitioned tables you have to collect the statistics for the whole table again. For partitioned objects the preferred method is to rely on incremental statistics collection or to alternatively copy statistics from another partition on the same table ; both of these methods are explained later in this document.

Irrespective of whether (or not) you are dealing with partitioned or non-partitioned table, Oracle strongly recommends to collect statistics with the 'AUTO_SAMPLE_SIZE' for optimal performance and accuracy of your statistics.

Disabling automatic statistics collection for periodically loaded large tables

By default, Oracle collects statistics for stale objects every night in the maintenance window to keep the statistics up-to-date[16]. Assuming that in a data warehousing environment all large tables are loaded periodically and that its data does not change between the loads, Oracle recommends that auto collection of statistics is turned off for just these tables and that the statistics management is done manually after each load.

To disable automatic statistics collection for an individual table you can “lock” its statistics, for example

```
execute dbms_stats.lock_table_stats('SH', 'SALES');
```

To collects statistics on a table with locked statistics, you have to set the FORCE parameter of gather_table_stats to TRUE, for example,

```
execute dbms_stats.gather_table_stats('SH','SALES', force=>TRUE);
```

Index statistics for invalidated (rebuild) indexes

If indexes are invalidated throughout the load and rebuilt afterwards, the index rebuilt process will automatically calculate statistics on the rebuilt indexes. If statistics are collected after a load has finished and indexes are rebuilt, there is no need to re-calculate statistics on the index. This can be controlled by setting the CASCADE option when collecting statistics, for example,

```
exec dbms_stats.gather_table_stats('SH', 'SALES', force=>TRUE,
                                                  cascade=>FALSE);
```

Incremental statistics management for partitioned tables

Typically an incremental load in a data warehouse modifies a one or a small number of partitions; especially for larger tables, as the amount of new data loaded is normally small compared to the overall size of the table. While statistics collection of the newly loaded (or updated) data partitions is normally a relatively inexpensive, re-gathering global statistics is not. Re-gathering global statistics will require a full table scan, which could consume a large amount of system resources and time. If you chose not to re-gather global statistics you would have to rely on less accurate aggregated global statistics as some global statistics - like number of distinct values of a column - are not easily derivable from its partition statistics. Oracle’s incremental statistics collection for partitioned tables automatically tracks what partitions have changed since the last statistics collection and provides incremental global statistics maintenance: the database only collects statistics on the changed partitions and incrementally computes and updates the global statistics of the table without re-scanning the entire table. The database maintains a so-called partition synopsis (per partition) while computing individual partition statistics;

[16] For more details about the maintenance window see the Oracle documentation

this data is stored in the SYSAUX tablespace and used for the computation of the global table statistics.

For optimal statistics collection of partitioned tables, Oracle recommends to enable incremental statistics maintenance for partitioned tables. You can enable this functionality on a database or at a table level. For example, enabling it on a table level you have to set three statistics parameters: INCREMENTAL, ESTIMATE_PERCENT, and AUTO_SAMPLE_SIZE[17].

You should enable incremental statistics before loading data or at least before collecting statistics for the first time. For example, to enable incremental statistics for table SALES

```
exec dbms_stats.set_table_prefs('SH', 'SALES',
                          'INCREMENTAL', 'TRUE');
exec dbms_stats.set_table_prefs('SH', 'SALES',
                          'ESTIMATE_PERCENT', 'AUTO_SAMPLE_SIZE');
exec dbms_stats.set_table_prefs(null, 'SALES',
                          'GRANULARITY', 'AUTO');
```

Statistics management with COPY_STATS

There is a special consideration for continual incremental loading into a partition with concurrent query access where you either do not have a window of opportunity to collect statistic prior to using the object or the statistics gets stale very quickly. For example, when you add a new partition to a time-based range partitioned table and data begins to be continuously loaded, but the data is accessible for the end users right away: while rows are still being loaded into this new partition, users begin to query the newly inserted data. Partition statistics do not exist since the partition was just created and cannot be created since the data is not completely loaded yet.

In this situation Oracle recommends to using the copy table statistics procedure to pre-populate the statistics of the newly added partition with statistics of a similar existing partition from the same table. This procedure copies the statistics of a source [sub] partition to a destination [sub] partition. It also copies the statistics of the dependent objects, for example columns, local (partitioned) indexes etc. It also adjusts the minimum and maximum values of the partitioning column at the global level by using the high bound partitioning value as the maximum value of the first partitioning column (you can have concatenated partition columns) and high bound partitioning value of the previous partition as the minimum value of the first partitioning column for range partitioned table. It can optionally scale some of the statistics like number of blocks, number of rows etc. of the destination.

[17] For details see the [Oracle Database Performance Tuning Guide]

Conclusion

With External Tables and its transparent access from within the database, Oracle Database12c provides a scalable and performant data loading infrastructure to load large volumes of data within the shortest period of time. Together with the automatic DOP capabilities of Oracle Database 12c you can leverage the massively parallel capabilities of the Oracle Database and Oracle Real Application Clusters. Loading terabytes of data within an hour or less on the right hardware infrastructure, like an Oracle Exadata Database Machine, never was easier and more performant.

Appendix A: Summary of best practices recommendations

This section provides a quick summary of recommendations for data loading into a Data Warehouse. The summary is divided into a parameter setting summary where we explain important settings for various parameters pertaining to Data Warehouse loads.

Settings Summary

Parameter type	Setting
Memory	At least 4 GB of physical memory per core
System Parameters (parallel query)	• parallel_execution_message_size=16K • parallel_min_servers =2*<expected “normal” degree of parallelism> • parallel_max_servers: leave at default
Degree of Parallelism (DOP)	• Use AutoDOP whenever possible • Ensure sufficient DOP with manual control of DOP
System Parameters (memory)	• pga_aggregate_target = 60% of available memory for rdbms • sga_aggregate_target = 40% of available memory for rdbms
Tablespace parameters	• Use ASSM • Use BIGFILE • Use AUTOALLOCATE with very large AUTOEXTEND ON NEXT • User with UNLIMITTED QUOTA on the tablepace
Table parameters	• Consider COMPRESSION • Consider NOLOGGING • Ensure PARALLEL decoration • Use large INITIAL and NEXT extent (e.g. 8M)
External Files	• Create separate OS directories for data, logging, bad, and pre-processor files. Log and bad directories typically share the same OS directory • Ensure the proper OS permissions are set to allow the Oracle OS user to read, write and execute as appropriate. • Create separate Oracle directory objects for data, logging, bad, and pre-processor files and appropriate privileges • Use formats of external data that enable automatic granulation

	(defined record terminators and column delimiters) for parallelism unless data pre-processing is used. • The ORACLE_LOADER format is the most common format for the external data files • External files should be of similar size. If not order them by size in LOCATION clause. • For compressed external data use pre-processor to decompress it. This implies manual granulation. For this option, number of external files should be at least the requested DOP. • Layout of the data in external files should not cause temporal skew when loading into partitioned tables. • Use direct path insert • Choose appropriate index maintenance

Appendix B: Advanced topics

The performance of load depends on the available memory, the number of partitions to being loaded, the skew of the data, the degree of parallelism of the load, the distribution method used and the throughput of the filesystem the access drivers read from. Users have (some) control over these influencing factors.

Memory consideration for load

This is a background section useful for manual hinting of load statements and determining the degree of parallelism for a load.

Loading into a partitioned table with many partitions can be a memory consuming operation. The memory comes both from the SGA (mostly for messages/communication between parallel processes) and the PGA (mostly for load buffers). A loader process needs about $LOAD_{buf_size}$ = 0.5MB of memory for each uncompressed partition it actively loads into. If you are loading into a partition that is compressed, the needed memory is approximately 1MB per partition. The memory is used to buffer rows before inserting them into the target table, allowing for good space utilization and low insert overhead. Note that the memory allocation happens in a lazy fashion: a loader process allocates a buffer for a partition only when it processes a row belonging to it. It then keeps the allocated memory buffer until the end of load. If a loader process inserts data into many partitions it will need a load buffer for every partition it is inserting into, potentially stressing memory as a resource. For example, if a load process inserts data into 1000 compressed partitions, it will consume 1000MB of memory for load buffers. If the load is happening with a degree of parallelism of 10, there will be 10 load processes and they will consume 10GB of PGA memory in aggregate.

Oracle Database 12c minimizes the memory usage by providing different distribution methods based on the degree of parallelism of a load (DOP), the number of partitions loaded (P_{num}), the available memory in PGA target (M_{pga}), and the number of Oracle instances (N_{nodes}) involved in the load operation. For example, if the number of active partitions is small and the DOP is moderate, then a very efficient **COMBINE** method is used. It is efficient since it does not require any redistribution of rows at all. Each parallel process may load into all active partitions hence we require P_{num} *DOP*1MB pga memory per load (in case of a compressed table). For example, when loading 10 partitions with DOP 10, we require only 10*10*1MB=100 MB.

If the number of active partitions is significantly larger than DOP we would quickly exhaust the available PGA memory. In this case Oracle maps partitions to each loading process and uses the **Partitioning Key distribution** to distribute rows read by the scanning processes. In this case each parallel process serves $\lceil P_{num} / DOP \rceil$ partitions and this requires only $\lceil P_{num} / DOP \rceil$*1MB memory. For example, if we load 1000 partitions with DOP=10, this requires 100MB of memory. The COMBINE method would require 10GB. Partitioning Key distribution method conserves memory over the COMBINE method, but requires row redistribution.

The **Random-Local** distribution method is a compromise between COMBINE and Partition Key distribution.

We note a subtle difference between term 'loaded partitions' and 'active partitions'. Loaded partitions are determined statically from the loaded table. For a partitioned table it is all its partitions. For a table partition extended syntax, these are sub-partitions of the top-level partitions. Active partitions, on the other hand, designate partitions that have rows loaded into them.

Users can determine the distribution method and approximate the amount of memory (M_{total}) consumed using the following approximate formulas. Assume P_{num} is the number of loaded partitions. If

$$M_{total}=P_{num} * DOP * 1MB <= 0.25 * M_{pga}$$

then COMBINE method is used, i.e., we use this method if the will consume less then a quarter of PGA target. Otherwise if

$$P_{num} > 8 * DOP \text{ or } Num_{HASH-PARTITIONS} >= DOP$$

then Partitioning Key Distribution is used. The factor of 8 is an internal heuristic signifying that number of active partitions is significantly larger than the DOP. In this case the total PGA memory used is

$$M_{total} = \lceil P_{num} / DOP \rceil * 1MB$$

Also, if the table you are loading into is compressed using Exadata Hybrid Columnar Compression, then Partitioning Key distribution is always chosen.

If none of the above is holds, Oracle constructs groups of partitions for Random-Local distribution. If the number of groups is N_{sub}, each serves $\lceil P_{num}/ N_{sub} \rceil$ partitions, and requires $\lceil P_{num}/ N_{sub} \rceil$*1MB memory. We construct as many groups as not to exceed quarter of PGA. The number of groups N_{sub} is governed by this (approximate) formula: $(P_{num}/ 0.25*M_{pga})*1MB <= N_{sub}$.

Note that in a corner case when there is memory shortage, we may reduce number of groups to DOP in which case random-local is equivalent to the Partition Key Distribution.

Distribution Methods of Parallel Load

This is a background section useful for manual hinting of load statements and determining the degree of parallelism for a load.

During parallel load there are two types of parallel processes: the scanning parallel processes and the loader parallel processes. The scanning processes must pass or distribute their data into the loader process.

Oracle Database 12c offers four methods to exchange the data between scanning and loader processes to optimize the usage of memory and speed of the load:

Combine: The scan is combined with the load and both operations are executed by an individual parallel process, without the need of any data redistribution. Note that with this method, if the target table is partitioned, the parallel process will have to handle all partitions that are going to be present in the scanned data. In the extreme cases a single parallel process has to load data into all partitions of the target table. A sample plan using combine is shown in Figure 4 earlier in this document.

Random: This method assumes that each loader process can insert data into the entire target table. If the target table is partitioned, this implies that each loader process might insert data into all partitions. The scanning process will map rows in a round robin fashion to all available loader processes. The row is placed into a buffer designated to a loader process, and when the buffer is filled it will be sent.

You might want to choose to enforce one over the other if FILTER predicates are applied to the data loading portion; A FILTER predicate might introduce a data skew that is not known to the database, but to the user. For example if you are loading compressed files in parallel and you know that the FILTER predicate will vary the data to being inserted significantly between the files you might want to enforce a random distribution.

Random-Local: This method is only applicable when the target table is partitioned. It assumes that an individual loader process works only on a subset of partitions. Loader processes are divided into groups. Each group serves a set of unique set of partitions. For example, given a degree of parallelism of 10, inserting into a target table with 100 partitions, Oracle may divide the loader processes into two groups. The first group of five loader processes will load into partitions 1-50, and the second group into partitions 51-100. Note that in this case, each loader process will load into at most 50 partitions. Within a group rows are distributed randomly to its loader processes, thus the name of this distribution method. The scanning processes know the mapping and place rows into loader buffers based on the partitioning column for the individual groups.

Partitioning Key Distribution: This method is only applicable when the target table is partitioned (applies also to insert into a single top-level partition with sub-partitions). Partitions of the target table are mapped to the loader processes. For example, given a degree of parallelism of 10 (and hence 10 loader processes) and a target table with 100 range partitions, each loader process will serve 10 unique partitions. The scanning process will map each scanned row to a loader process based on the partitioning column and will place it in a buffer designated to that process. When the buffer is filled, it will be sent to the designated loader process.

Choices of the distribution method have significant ramifications for needed memory (see Section 'Memory Consideration for Load') and for handling of skew during load, and hence its scalability.

Oracle Database 12c determines the distribution method automatically based on the available memory and the DOP of the load. As discussed earlier, the distribution methods are visible in the explain plan for the statement (see Section 'Explain Plan for Load'). In addition, users can force a distribution using hints, as discussed later in this document.

Note that "Random-Local" distribution can morph to the "Partition Key Distribution" if the number of slave groups is equal to the number of loader slaves, i.e., if Oracle assigned the number of groups to be the DOP of the load. On the other opposite, if there is only one group, this distribution becomes the same as the "Random Distribution".

In general, COMBINE method is recommended and will be chosen while loading into a non-partitioned table, into a single or very few partitions of a partitioned table. Partition Distribution is recommended and will be chosen when loading into many partitions. Random-local is recommended

and will be chosen when loading into many partitions and when there is a data skew on the partition key. This is explained in the next section.

Applying parallel distribution hints for the load

In Oracle Database 12c, any parallel CTAS as well as a parallel INSERT /*+APPEND */… AS SELECT accept the PQ_DISTRIBUTE hint that will force a distribution method between the scanning and loader processes. For example,

```
INSERT /*+ APPEND PARALLEL(sales, 16)PQ_DISTRIBUTE(sales, NONE)*/
INTO sales
SELECT /*+ PARALLEL(sales_external, 16) */
FROM sales_external;


CREATE /*+ PQ_DISTRIBUTE(sales, NONE) */ TABLE sales
NOLOGGING PARALLEL   PARTITION BY (..) AS
SELECT /*+ PARALLEL(sales_external, 16) */
FROM sales_external;
```

The formal specification for the hint is:

```
/*+ PQ_DISTRIBUTE(table, DISTRIBUTION) */
```

where DISTRIBUTION defines the method to force between the producer (scanner) and the consumer (loader) processes:

- NONE: no distribution, e.g. combine the scanning and loader slaves
- PARTITION: use the partitioning of the table being loaded to distribute the rows.
- RANDOM: distribute the rows in a round-robin fashion.
- RANDOM_LOCAL: distribute the rows from the producers to a set of slaves that are responsible for maintaining a given set of partitions.

The impact of data distribution (skew) in external data files

Whether or not data is pre-sorted within a single external data file or between several external data files, its sorting can have an impact on the load behavior. For example, the data might contain a date field that is mapped into a date-range partition column in the target table and several staging files contain the data sorted by the partitioning column. Loading such data in parallel into the partitioned target table potentially requires the load operation to redistribute the data based on the partitioning key; for example, if we have tens of thousands of partitions and a degree of parallelism of 128, each parallel loader process might become exclusively responsible for loading data in some of the partitions (instead of having all processes loading in all partitions, consuming a large amount of memory load buffers). If the data is large and sorted you might end up loading partitions "in a sequence" – all data scanned belongs to a single partition at a time - and it will cause temporal imbalance of work among the parallel loader processes.

The following section will illustrate this behavior for two typical common scenarios of how external data files are structures. We further assume that these data files can be loaded using automatic parallelism (parallel access within a data file), as discussed earlier in this document.

Case 1: Clustered and overlapping external data files

The external data files being loaded are sorted or clustered on the partition key of target table and the keys overlap. For example, consider the following scenario:

- The target table we are loading into is a range partitioned table with daily partitions for 8 years worth of data
- Ten staging files f1, …, f10, of similar size with a size of 100MB each
- Staging files contain data for 10 partitions of the target table, for example DAY_2011_JAN_01, DAY_2011_JAN_02, …, DAY_2011_JAN_10.
- Each of the files contains data for the entire 10 days
- Data is sorted within a data file on the partitioning key of the target table.
- Similar number of rows for each partition

In this case when the parallel granules (units of work) are created, each granule will contain similar ranges of the partitioning key. For example, we will have 10 granules G1.1, G1.2, ...G1.10 for file f1, 10 granules G2.1, G2.2, ...G.2.10 for file f2, and so on.

If we are loading this data with a degree of parallelism of 10, we will have 10 scan processes starting to process all 10 files concurrently, resulting in first scanning G1.1, G2.1, ..G.10.1, followed by G1.2, G2.2,…G2.10, followed by G3.1, G3.2, ..G3.10, and so on. If the load operation uses a **Partition Key Distribution** method, then only one of the ten parallel load processes will be active at a time, resulting in temporal load imbalance. For example, when scanning granules G1.1, G2.1, ..G.10.1 only rows for day_2008_JAN_01 may be seen.

Figure 7: Clustered and overlapping files

In this case a better distribution method is one that allows multiple loader processes to insert into a partition. If there is enough memory, COMBINE is preferred; otherwise, Random Local or Random should be chosen.

Oracle has no knowledge of data distribution in the files; the choice of the distribution method is determined by the available memory and the shape of the target table. Since we insert into a table with 365*8=2920 partition, Oracle will conservatively estimate that we need approximately 10*2920*0.5MB=14GB of memory when choosing a COMBINE. In our example it chose Partition Key Distribution where only 2920*0.5MB = 1.4GB would be needed.

Knowing the data we are going to load and the fact that the chosen distribution method is suboptimal, it is safe to hint a different data distribution method. In our case, using COMBINE with a degree of parallelism of 10 and 10 active partitions, we will need 10*10*0.5MB = 50MB of PGA memory, so COMBINE is a suitable approach. You can hint the COMBINE distribution method by using `/*+ PQ_DISTRIBUTE(<table_name>, NONE) */`.

Case 2: Clustered and not overlapping external data files

The external data files being loaded are sorted or clustered on the partition key of target table and the keys do not overlap. For example, consider the following scenario:

- The target table we are loading into is a range partitioned table with daily partitions for 8 years worth of data
- Ten staging files f1, …, f10, of similar size with a size of 100MB each
- Staging files contain data for 10 partitions of the target table, for example DAY_2011_JAN_01, DAY_2011_JAN_02, …, DAY_2011_JAN_10.
- Data files contain not overlapping data ranges; file f1 contains only JAN_01 data, file f2 only JAN_02 data, etc.

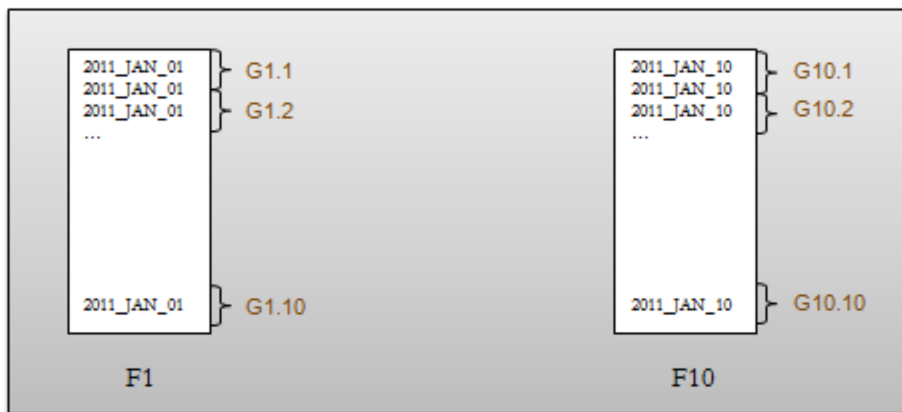

Figure 8: Clustered and not overlapping files

In this case, all distribution methods are appropriate. Hence, if there is enough PGA memory for the requested degree of parallelism of the load and for the number of partitions affected, then COMBINE is preferred. Otherwise, Partition Key, Random local, or random methods are acceptable in that order.

Case 3: Non-clustered external data files

External data files have random distribution of the partitioning key. For example, assume similarly to Case 1, there are multiple staging files f1, f2, ..,f10. However, data is distributed randomly within the files with respect to the partitioning key of the sales table. All distribution methods are appropriate as explained in the Case 2.

Case 4: Unbalanced file sizes

There are multiple staging files, f1, f2, …, f10 but they differ significantly in sizes. However, even though sizes of files are not equal, the number of rows designated to each partition is. For example, assume that f1, f2,..f9 have 10MB, but f10 is significantly bigger than 100MB. File f10 may be or may not be clustered on the partitioning key of the sales table. In this case, all distribution methods are appropriate as explained in the Case 2.

Case 5: Unbalanced size of partitions.

In this case number of rows designated to different partitions are significantly different, hence data imbalance. Assume that f1, f2,..f9 have 100MB, but f10 is significantly bigger 1GB. File f10 may be or may not be clustered on the partitioning key of the sales table. Assume that data in file f10 is designated to one partition, say day_2008_JAN_10. Files f1, f2, ..f9 have equal number of rows for partitions JAN_01 to JAN_09. Population of partitions is not balanced. In this case, all distribution methods except for Partitioning Key Distribution are appropriate as explained in the Case 2. Oracle does not know about partition imbalance, hence the load plan needs to be checked against Partitioning Key Distribution.

Accessing remote data staging files using Oracle external tables

Typically external tables are created with the data staging files stored on the local machine. The problem at hand is to provide a mechanism that enables the access of staging files stored on a remote machine via external tables. You can use the pre-processor capability of external tables to ssh into the remote machine and use the "local data file" specified in the access parameters of the external table as a proxy for the remote data file.

The following will outline the steps to enable a remote access of data staging files through external table. For the purpose of this sample setup we will refer to the machine (or node) where the external table access will happen as the "local machine", while the "remote machine" is the machine (or node) where the actual physical file(s) will reside.

The remote "data access"

The mechanism provided in this example does not require the user to move data or even have remote file access, e.g. a network file system. However, ssh will be used to make a remote connection and to "read" the remote file content; in external table words this means to invoke "some program" on the

remote side to stream the data to the standard output so that it can be read as external table data stream.

Pre-requirement ssh equivalence

ssh needs to be set up to allow the Oracle OS user (the OS user running the database) on the local machine to access the remote machine(s). You have to ensure a proper setup of ssh user equivalence for all participating machines, which enables to login with ssh on a remote machine without specifying a password. Please refer to your operating system documentation of how to set up ssh and ssh equivalence.

Setting up the external table as proxy on the local machine

The external table setup on the local machine requires both Oracle directory objects and data staging files of being existent. Local data staging files are necessary since the external table access driver checks for these files and requires them to exist. In our example we will rely on one file only and use this as proxy identifier, as discussed later in this example.

Assume that our directory structure for the local machine is the following

OS DIRECTORY	ORACLE DIRECTORY OBJECT	PURPOSE
/dbfs/staging/data	DATA_DIR	*directory where the proxy data file will reside*
/dbfs/staging/log	LOG_DIR	*directory where the log and bad files will reside*
/dbfs/staging/exec	EXEC_DIR	*directory where the preprocessor script will reside*

The external table sample setup would look as follows:

```
create table xtab (<col definition>)
organization external(
type oracle_loader
default directory datadir
access parameters(
records delimited by newline
badfile log_dir:'foo.bad'
logfile log_dir:'foo.log'
PREPROCESSOR execdir:'ora_xtra_pp.sh' …)
LOCATION('foo.dat'))
reject limit unlimited;
```

Note that this external table points to a local dummy file ‘foo.dat’, residing in /dbfs/staging/data ; as mentioned before, this file must exist as empty file. Log and bad files are written locally (/dbfs/staging/log), and a local shell script ‘ora_xtra_pp.sh’ (in /dbfs/staging/exec/) is invoked for pre-processing. In fact, the “pre-processing script is doing all the work, and no data is read locally at all.

Setting up the pre-processor

As mentioned earlier, our pre-processor shell script is written in a way that it acts as a proxy for different remote files; residing in different locations, using different OS users; by mapping the remote file name to the local dummy file name (which is obviously the only implicit "input parameter" that an external table access provides). The external table definition controls what remote file to access, so you can have multiple external tables using the same pre-processor script to access different data.

The mapping file 'ora_xtra_map.txt ' that is used by the pre-processor shell script 'ora_xtra_pp.sh' has to be created in addition to the actual pre-processor shell script; for our example the mapping file has to reside in the same directory than the pre-processor shell script.. It identifies the remote file name, the remote node, the OS user to connect to the remote node, the remote directory, and the remote command to be executed on the file.

The format of mapping file 'ora_xtra_map.txt' is

```
        file:rusr:rhost:rdir:rcmd
```

where

- **file**: name of the remote data staging file. Note that the name has to match the local proxy file name to initiate a remote access of the file
- **rusr**: name of the remote user used to ssh into the remote system
- **rhost**: name of the remote host where actual data file is stored
- **rdir**: name of the directory where the data file resides on the remote machine (excluding file name). The remote directory path is completely independent of how the local oracle directory is created
- **rcmd**: command to be executed on the file mentioned (example 'cat', 'zcat').

Example entries in mapping file 'ora_xtra_map.txt' could be as follows:

```
foo.dat:ellison:san_francisco:/home/ellison/xt_data:cat
bar.dat:mueller:munich:/home/mueller/xt_data:zcat
```

Depending on the external table definitions, different remote files will read. For example, the external table definition shown earlier identified a file named 'foo.dat', which will lead to a remote access of 'foo.dat' on remote machine 'san_francisco' through ssh equivalence with user 'ellison'. An external table pointing to proxy file 'bar.dat" would access 'bar.dat' on remote machine 'munich' using OS user 'mueller'.

The following shows the sample pre-processor script.

```
#! /bin/sh
PATH=/bin:/usr/bin export PATH

# ora_xtra_pp: ORAcle eXternal Table Remote file Access Pre- Processor

# Format for the Map File
# Consists of five fields separated using a :
# Syntax
# file:rusr:rhost:rdir:rcmd
#
# Example
# foo.dat:ellison:san_francisco:/home/ellison/xt_data:cat
# get filename component of LOCATION, the access driver
# provides the LOCATION as the first argument to the preprocessor

proxy_file_name=`basename $1`
data_dir_name=`dirname $1`

# Flag is set if the file name in the Map file matches the proxy file name
# where our data file is stored.

flag_dirf=0

# loops through the map file and fetches details for ssh
# username, hostname and remote directory

file_not_found_err='ora_xtra_pp: Map file missing. Needed ora_xtra_map.txt in data
directory.'

if [ -e $data_dir_name/ora_xtra_map.txt ]
then
  while read line
  do
    map_file_name=`echo $line | cut -d : -f1`
    if [ $map_file_name = $proxy_file_name ]
    then
      rdir=`echo $line | cut -d : -f4`
      rusr=`echo $line | cut -d : -f2`
      rhost=`echo $line | cut -d : -f3`
      rcmd=`echo $line | cut -d : -f5`
      flag_dirf=1
      break
    fi
  done <$data_dir_name/ora_xtra_map.txt
else
  echo $file_not_found_err 1>&2
  echo $data_dir_name 1>&2
  exit 1
fi

if [ $flag_dirf = 1 ]
then
  ssh -q $rusr@$rhost $rcmd $rdir/$proxy_file_name
```

Performant data loading
with Oracle Database 12c
April 2014
Author: Hermann Baer
Contributing Authors: Andy Witkowski, Allen Brumm

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Copyright © 2014, Oracle and/or its affiliates. All rights reserved.

This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0114

Hardware and Software, Engineered to Work Together