ORACLE

# Best Practices For High Volume IoT workloads with Oracle Database 19c

Version 1.01

## PURPOSE STATEMENT

This document provides an overview of features and enhancements included in the release Oracle Database 19c that can help improve the performance of high-volume ingest workloads like IoT. It is intended solely to help you assess the business benefits of using Oracle Database 19c and plan your IT projects.

## INTENDED AUDIENCE

Readers are assumed to have basic knowledge of Oracle Database technologies.

## DISCLAIMER



## TABLE OF CONTENTS

# INTRODUCTION

Over the last ten years, there has been a rapid surge in the adoption of smart devices. Everything from phones and tablets to smart meters and fitness devices connect to the Internet and share data enabling remote access, automatic software updates, error reporting, and sensor readings transmissions. Gartner estimates that by the end of 2021, there will be over 26 billion connected devices.

With these intelligent devices comes a considerable increase in the frequency and volume of data being ingested and processed by databases. This scenario is commonly referred to as the Internet of Things or IoT. Being able to ingest and analyze rapidly increasing data volumes in a performant and timely manner is critical for businesses to maintain their competitive advantage. Determining the best platform to manage this data is a common problem faced by many organizations across different industries.

Some people assume that a NoSQL database is required for an IoT workload because the ingest rate required exceeds a traditional relational database's capabilities. This is simply not true. A relational database can easily exceed the performance of a NoSQL database when properly tuned.

Oracle Database is more than capable of ingesting hundreds of millions of rows per second. It is also the industry-leading database in terms of analytics, high availability, security and scalability, making it the best choice for mission-critical IoT workloads.

The performance of data ingest operations are affected by many variables, including the method used to insert the data, the schema, parallel execution, and the commit rate. The same is true for analytical queries. This paper outlines the best practices to ensure optimal performance when ingesting and analyzing large volumes of data in real-time with Oracle Databases.

# WHAT IS THE INTERNET OF THINGS?

The Internet of Things (IoT) is a world where all physical assets and devices are connected and share information, making life easier and more convenient. Example applications include Smart Meters that record the hourly use of electricity or gas, sensors such as those found on cargo containers that report their location, temperature, and if the door has been opened. Or perhaps wearable technology that analyzes a person's diet, exercise, and sleep.

There are also many industrial examples that have used Oracle Databases for decades to process these types of workloads, long before it was called IoT. For instance, telecoms process tens of millions of Call Detail Records (CDRs) [that are] generated every second on telephone switches across the world. Or connected manufacturing equipment, where every machine in a production line continually sends information about which component is currently being worked on and by whom.

Regardless of how the data is generated, the essential requirements for IoT workloads remain the same, and they are:

» Scalability – Rapid increase in the number of devices and volume of data they generate

» Flexibility – New and or different data may make it necessary to iterate the data model

» Real-time Analytics – Without having the luxury of lengthy ETL processes to cleanse data for downstream reporting

This paper examines each of these requirements and explains how Oracle has developed very sophisticated tuning and management techniques by working with leading telecoms, financial institutions, and manufactures over decades to not only facilitate them but excel at them. Throughout the paper, an analogy of someone going grocery store shopping is used to help explain the reasoning behind each of the tuning techniques recommended.

# SCALABILITY

Scalability is a system's ability to provide throughput in proportion to and limited only by available hardware resources. Oracle Database offers the ability to scale up (increasing hardware capacity within a single server) or scale-out (increasing the number of servers in a cluster). For IoT projects, the consensus is that a scale-out solution is preferred, as it allows for scalability at a lower cost. Oracle Database offers two scale-out architecture options; Real Application Clusters (RAC) and Oracle Sharding.

## Real Application Clusters

**RAC** enables any packaged or custom application to run unchanged across a server pool connected to shared storage. If a server in the pool fails, the database continues to run on the remaining servers. When you need more processing power, simply add another server to the pool without incurring any downtime. This paper assumes the Oracle Database is deployed in a RAC environment consisting of two or more RAC nodes.

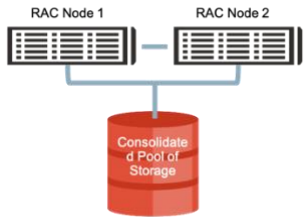


*Figure 1: Simple RAC Architecture with two RAC nodes*

## Oracle Sharding

**Oracle Sharding** is an alternative scalability feature for custom-designed applications that enables distribution and replication of data across a pool of Oracle databases that share no hardware or software. The pool of databases is presented to the application as a single logical database. Applications elastically scale (data, transactions and users) to any level, on any platform, simply by adding additional databases (shards) to the pool.

Sharding divides a database into a farm of independent databases, thereby avoiding scalability or availability edge cases associated with a single database. Data is distributed across shards using horizontal partitioning. Horizontal partitioning splits a database table across shards so that each shard contains the table with the same columns but a different subset of rows. This partitioning is based on a sharding key.

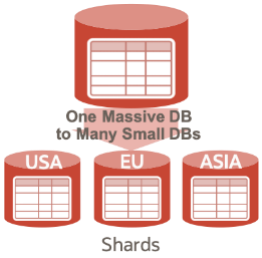


*Figure 2: Oracle Sharding Architecture*

Sharding trades-off transparency in return for massive linear scalability, greater availability, and geographical distribution. With Sharding, the application needs to be shard-aware. That is to say; queries need to provide the shard-key to be directed to the appropriate shard or database. It is also possible to run cross-shard queries.

## Database Configuration

Let's now focus on configuring an Oracle Database to ensure we can scale as we increase the number of nodes in our RAC environment.

### Tablespaces

An essential aspect for loading - and querying - large amounts of data is the space management used for the underlying tables. The goal is to load data as fast and efficiently as possible while guaranteeing the physical data storage will not be detrimental to future data access. You can think of this in the same way you would plan for a large weekly shop at a grocery store. If you know you need to buy many groceries, you select a cart instead of a handbasket when you enter the store. A cart ensures there is plenty of room for everything you need to purchase and allows you to easily access all of your items when you get to the register.

With Oracle Database, space management is controlled on the table level or inherited from the tablespace where the table (or partition) resides. Oracle recommends the usage of `BIGFILE` tablespaces to limit the number of data files to be managed. `BIGFILE` tablespaces are locally managed tablespaces that leverage Automatic Segment Space Management (ASSM) by default.

On the creation of a `BIGFILE` tablespace, the initial data file is formatted. Each time the data file is extended, the extended portion must also be formatted. The extension and formatting of data files is an expensive operation that should be minimized (wait event: Data file init write). Furthermore, formatting is done serially (unless you are on an Oracle Exadata Database Machine where each Exadata Storage Server performs the formatting of its space independently).

Locally managed tablespaces can have two types of extent management: `AUTOALLOCATE` (default) and `UNIFORM`. With `AUTOALLOCATE`, the database chooses variable extent sizes based on the property and size of an object while the `UNIFORM` enforces extents with a pre-defined fixed size. For high rate ingest workloads, such as IoT, Oracle recommends using `AUTOALLOCATE`. However, the default allocation policy of `AUTOALLOCATE` is to start with a tiny extent size, which may be to conservative for a heavy ingest workload like IoT. Therefore, we recommend you specify a larger initial extent size at table creation time (minimum of 8 MB) to force `AUTOALLOCATE` to begin with this value and go up from there. This will avoid having processes wait for new extents to be allocated (wait event: enq: HW – contention).

For optimal performance, Oracle recommends minimizing space allocation operations during loads by specifying auto-extension size on a big file tablespace. The `AUTOEXTEND` parameter controls how much additional space is added to a `BIGFILE` tablespace when it runs out of space. The following example creates the tablespace `TS_DATA` applying the above-discussed best practices:

```
CREATE BIGFILE TABLESPACE ts_data
DATAFILE '/my_dbspace/ts_data_bigfile.dbf' SIZE 1500G
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
```

*Figure 3: Example of a create BIGFILE tablespace command*

Also, grant the user executing the data load an unlimited quota on the tablespace to avoid any accounting overhead every time additional space is requested in the tablespace.

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

*Figure 4: Grant the user who owns the object being loaded into an unlimited quote on the ts_data tablespace*

As tablespaces become full, they should be marked read-only. This will ensure a tablespace will only be backed up once rather than every night, extending the backup process.

### Redo Logs

All of the changes made in an Oracle Database are recorded in the redo logs to ensure that the data is durable and consistent. You should create a minimum of 3 redo log groups to ensure the logging process does not impact the ingest performance. Multiple groups prevent the log writer process (`LGWR`) from waiting for a group to be available following a log switch. A group may be unavailable because a checkpoint has not completed, or the group has not yet been archived (wait event: Log file switch completion).

The redo log files should also be sized large enough to ensure log file switching occurs approximately once an hour during regular activity and no more frequently than every 20 minutes during peak activity. All log files should be the same size. Use the following formula to determine the appropriate size for your logs:

```
Redo log size = MIN(4G,Peak redo rate per minute x 20)
```

*Figure 5: Formula for sizing Redo logs appropriately*

Finally, place the redo logs on high-performance disks. The changes written to the redo logs need to be done in a performant manner (wait event: Log File Parallel Write).

### Memory Settings

Several aspects of a data load operation require memory either from the System Global Area (SGA) and or the Program Global Area (PGA) in the database instance.

#### SGA

The SGA is a group of shared memory structures that contain data and control information for the database instance. Examples of data stored in the SGA include cached data blocks (known as the buffer cache) and a shared SQL area (known as the shared pool). All server and background processes share the SGA. The `SGA_TARGET` and `SGA_MAX_SIZE` parameters control the total size of the SGA.

To ensure optimal performance for Conventional inserts, allocate at least 50% of the available memory for the database to the SGA. The majority of SGA memory should go to the buffer cache and the shared pool.

#### PGA

The PGA is a non-shared, private memory region that contains data and control information exclusively for use by a session process to complete SQL executions, aggregations, and sort operations, amongst other things. Oracle Database creates the PGA inside the session process when the process is started. The collection of individual PGAs is the total instance PGA. The `PGA_AGGREGATE_TARGET` and `PGA_AGGREGATE_LIMIT` parameters control the total amount of memory available for all private process. Allocate a minimum of 20% of the available memory for Oracle to the PGA.

More memory may be required if using parallel Direct Path Loads & External Tables, as each parallel process buffers rows before inserting them into the target table. You should assume each parallel process requires 0.5MB if loading into a non-compressed table and 1 MB if loading into a compressed table.

You should also account for the number of segments that will be loaded into concurrently. Each process allocates a separate buffer in the PGA for each segment they insert into.

## Data loading Mechanisms

Up until now, we have focused on creating a scalable database environment. Let's now switch our attention to data ingestion. With Oracle Database, data can be inserted into a table or partition in two ways: conventional inserts or direct-path inserts. You can think of data ingestion as being analogous to putting items in your cart and paying for them at the grocery store. You would never select one item at a time and pay for it before selecting the next item on your list (single row insert followed by a commit). You would walk through the store, collecting all of the items on your list and go to the check out once (array insert followed by a commit). The same is true when you want to insert data efficiently into the database.

### Conventional inserts

Conventional inserts use the `SQL INSERT` statement to add new rows to either a table or partition. Oracle Database automatically maintains all referential integrity constraints and any other indexes on the table. The database also tries to reuse any existing free

space within the database blocks that already make up the table. All aspects of a conventional `INSERT` statement are recorded in the redo logs in case of a failure. Typically, an `INSERT` command adds one row at a time and is followed by a `COMMIT`, although it is possible to add multiple rows with the `INSERT` command by using the `INSERT ALL`.

A single session can ingest approximately 500 rows per second via a conventional single-row insert, followed by a commit. To ingest large volumes of data via single-row inserts, hundreds of concurrent sessions would need to issue the same insert statement. Having so many concurrent sessions execute the same statement may lead to contention in the shared pool (wait event: cursor: pin S) and at the cluster lock level (wait event: enqueue hash chains latch).

To reduce the contention in the shared pool, for individual statements that are executed thousands of times per second, we recommend marking the insert statement as "hot" via the `MARKHOT` procedure in the `DBMS_SHARED_POOL` package (see figure 6). Marking a SQL statement or PL/SQL package hot enables each session to have its own copy of the statement in the shared pool and relieves the contention at the expense of more shared pool memory usage for this statement.

```
BEGIN
  dbms_shared_pool.markhot ( hash=>'01630e17906c4f222031266c21b49303',
                             namespace=>0,
                             global => TRUE);
END;

/
```

*Figure 6: Using the `DBMS_SHARED_POOL` package to mark the insert statement hot*

To reduce locking overhead, you could consider disabling table-level locking. Disabling the table level lock speeds up each insert statement as we no longer have to secure a shared table-level lock before beginning the transaction. However, it will prevent any DDL commands (`DROP, TRUNCATE, ADD COLUMN` etc.) from occurring on the object.

```
ALTER TABLE Meter_Readings DISABLE TABLE LOCK;
```

*Figure 7: Disable table-level locking*

A more convenient way to reduce locking overhead is to increase the number of transaction slots (ITL slots) in a database block. By default, each database block has 2 ITL slots. However, if you increase the INITRANS attribute on a database table, you can increase the number of ITL slots created. The INITRANS attribute should be set the number of concurrent inserts expected per database block.

## Commit Frequency

To persist any change in the database, you must commit the transaction. Issuing a COMMIT statement after each INSERT generates a large amount of redo, which causes high disk and CPU utilization on the RAC nodes, especially if you have a lot of concurrent sessions (wait event: log file sync and DB CPU). Therefore, it is recommended to COMMIT only after inserting multiple rows, for example, 100 rows. This will reduce the volume of redo generated and CPU consumed and take a single session's ingest rate from 550 rows a second to 7 thousand rows per second.

However, a far more efficient alternative to single row conventional inserts is to take advantage of array inserts and commit after each array insert.

## Array Inserts

Oracle's array interface enables many records or rows to be inserted with a single statement. You can take advantage of array inserts using any Oracle Database application-programming interface (API) regardless of which programing language you use (Python, Java, JavaScript, .NET, PL/SQL, C/C++ etc.). The array interface significantly reduces both the redo generated per row (6X less than single row inserts) and the CPU on the database server (15X less CPU than single row inserts), resulting in 18 thousand inserts per second for a single session on the test environment. Using significantly less CPU per session allows for more concurrent sessions.

Array Inserts also reduces the number of network round trips and context switches when you insert a large volume of data. This reduction can lead to considerable performance gains.

An example of an array insert in Python is available in Appendix B.

## Direct Path Loads & External Tables

An alternative and more efficient approach to conventional inserts is to take advantage of Oracle's direct path loads. A direct path load is preferable over a conventional insert if the data to be ingested arrives in large flat files. A direct path load parses the input data, converts the data for each input field to its corresponding Oracle data type, and then builds a column array structure for the data.

These column array structures are used to format Oracle data blocks and build index keys. The newly formatted database blocks are then written directly to the database, bypassing the standard SQL processing engine and the database buffer cache.

A direct path load is typically achieved by doing a `CREATE TABLE AS SELECT` or an `INSERT AS SELECT` statement from an external table. For the `INSERT AS SELECT` statement to bypass the database buffer cache, you must use the `APPEND` hint, as demonstrated in Figure 8 below. Direct path loads are suitable when data is loaded in "batch mode" every few minutes or more.

### External Tables

Direct path loads typically use external tables, which enable external data (files) to be visible within the database 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 is stored in files outside the database.

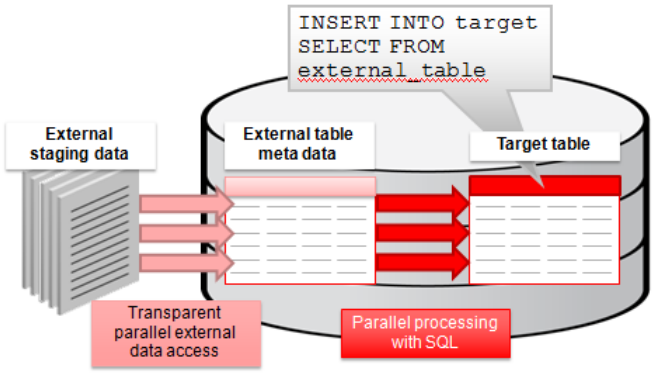


*Figure 8: Architecture and components of external tables*

An external table is created using the standard `CREATE TABLE` syntax, except it requires an additional `ORGANIZATION EXTERNAL` clause. This additional clause specifies information on the type of access driver required (`ORACLE_LOADER, ORACLE_DATAPUMP, ORACLE_HDFS, or ORACLE_HIVE`), access parameters, the name of the directory containing the files, and the definition of the columns within the table.

To ensure a performant and scalable data load using external tables, the access to the external files needs to be fast and parallel execution should be used.

### Location of the external staging files

The staging files should be located on an external shared storage accessible from all RAC nodes in the cluster. The IO throughput of the shared storage directly impacts the load speed, as data can never be loaded faster than it can be read.

To guarantee optimal performance, the staging files should not be placed on the same physical disks used by the database to avoid competing for IO bandwidth. This recommendation does not apply to Oracle Exadata configurations. Oracle Exadata has sufficient IO capacity to have the external data files staged on a Database File System (DBFS) striped across the same Exadata Storage Servers as the database files.

If the shared storage IO throughput is significantly lower than the database's ingest rate, consider compressing the external data files and pre-process the data before loading. Note, this is a tradeoff of CPU resources used to decompress the data versus IO bandwidth; it also imposes some restrictions on parallelism, as discussed in the parallelizing a direct path load section later in the document.

### External data format

The format of the data in the external staging files can also significantly impact load performance. Parsing the column formats and applying character set conversions can be very CPU intensive. We recommend that you use only single-character delimiters for record terminators and field delimiters, as they can be processed more efficiently than multi-character delimiters. It is also recommended that the character set used in the external data file match the database's character set to avoid character set conversion (single-byte or fixed-width character sets are the most efficient).

If data transformations and conversions do need to occur, it's best if Oracle does them as part of the loading process rather than pre-processing the files. This can be done either using SQL within the database or by leveraging external tables' pre-processing capabilities as part of the initial data access through external tables.

### Locking during Load

During a direct path load operation or any parallel DML, Oracle locks the entire target table exclusively. Locking prevents other DML or DDL operation against the table or its partitions; however, the table's data is fully accessible for queries from other sessions. You can prevent acquiring a table-level lock by using the partition extended syntax, which locks only the specified partition.

```
INSERT /*+ APPEND */ INTO Meter_Readings
PARTITION FOR (to_date('25-DEC-2016','dd-mon-yyyy'))
SELECT * FROM   ext_tab_mr_dec_25;
```

*Figure 9: Example of limiting locking during a direct path load operation using partition extended syntax*

### Parallelizing a Direct Path Load

Parallel execution is a commonly used method of speeding up operations by splitting them into smaller sub-tasks. Just as you would split up a large shopping list into two if your spouse went to the grocery store with you, you can take advantage of parallel execution within the database to speed up both data ingestion and queries. Parallel execution in Oracle Database is based on the principles of a coordinator (often called the Query Coordinator – QC for short) and parallel execution (PX) server processes. The QC is the session that initiates the parallel SQL statement, and the PX servers are the individual processes that perform work in parallel on behalf of the initiating session. The QC distributes the work to the PX servers and aggregates their results before returning them to the end-user.

To achieve scalable direct data loads, the external 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 concurrently by the PX server processes.

Oracle can build the parallel granules without any restrictions if it can position itself in an external data file and find the beginning of the next record. For example, when the data file contains single-byte records terminated by a well-known character (a new line or a semicolon). Each external data file is divided into granules of approximately 10 MB in size and distributed among the parallel server processes in a round-robin fashion. For optimal parallel load performance, all files should be similar in size, be multiples of 10MB, and have a minimum size of a few GB.

In this case, there are no constraints on the number of concurrent parallel server processes involved or the Degree of Parallelism (DOP), other than the requested DOP for the statement.

However, when the files' format prevents Oracle from finding record boundaries to build granules (compressed data, etc.) or when the type of media does not support position-able or seekable scans, the parallelization of the loading is defined by the number of data files. Oracle treats each data file as a single entity – and therefore, as a single granule. 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 DOP possible.

### Data Compression and Direct Path Load

Loading large volumes of data always beg the question of whether or not to compress the data during the data load. There is a tradeoff between maximizing the data ingest performance and improved query performance (since less data has to be read from disk) plus space savings. Regarding our grocery store analogy, you should consider compress akin to organizing items in your cart. You fit a lot more groceries in your cart if you spend a little time organizing them rather than just throwing them in.

To load data in a compressed format, you simply have to declare the target table (or partitions) as `COMPRESSED` 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, a feature of Exadata storage

Irrespective of what compression technique is chosen, additional CPU resource will be consumed during the data load operation. However, the benefits of compressing the data will likely far outweigh this cost for an IoT workload as the data is typically ingested once, never changed and queried many times.

## Memoptimized Row Store For Loading Streaming Data

For many IoT workloads, data is continuously streamed into the database directly from a smart device or application. Oracle Database 19c offer an efficient way to ingest streaming data via the Memoptimized Row Store. With Memoptimized Rowstore Fast Ingest, the standard Oracle transaction mechanisms are bypassed, and data is ingested into a temporary buffer in the Large Pool. The content of

the buffer is then periodically written to disk via a deferred, asynchronous process. Since the application doesn't have to wait for the data to be written to disk, the inserts statement returns extremely quickly. A single session can ingest approximately 1.4X more rows per second than with a conventional insert. However, when an array insert is used after batching 100 records on the mid-tier, which is recommended approach, a single session can insert over 1.7X more rows per second. More details on the performance you can expect from the Memoptimized Row Store can be found in Appendix A.

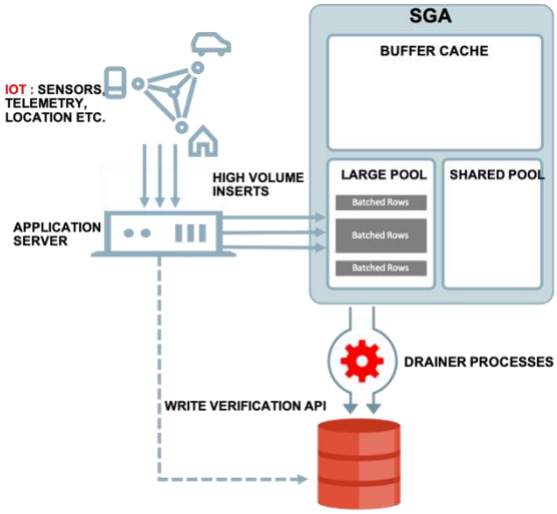


*Figure 10: Example of how Oracle Memoptimized Rowstore Fast Ingest can be used to load IoT data*

Note, you cannot query data until it's persisted to disk, and it is possible to lose data should the database go down before the ingested data has been persisted to disk. This behavior is very different from how transactions are traditionally processed in the Oracle Database, where data is logged and never lost once written/committed to the database. However, many IoT workloads can easily tolerate lost data as they are only interested in the difference between the values over time. For example, how much power was used by this household this month? In this case, you only need two reading, one from the beginning of the month and one from the end to calculate the difference.

However, if an application does need all data points to be persisted, it's the application's responsibility to check all data is persisted before disregarding it to ensure full ACID compliance. An application can confirm data was persisted using the `DBMS_MEMOPTIMIZE` package. It's also possible to force the content of the buffer to be flushed using the `DBMS_MEMOPTIMIZE` package.

To use Memoptimized Rowstore Fast Ingest, you must first enable one or more tables for fast ingest by adding the clause `MEMOPTIMIZE FOR WRITE` to a `CREATE TABLE` or `ALTER TABLE` statement. Then use the `MEMOPTIMIZE_WRITE` hint in all subsequent insert statements.

```
ALTER TABLE Meter_Readings MEMOPTIMIZE FOR WRITE;

INSERT /*+ MEMOPTIMIZE_WRITE */ INTO Meter_Readings
SELECT * FROM   ext_tab_mr_dec_25;
```

*Figure 11: Example of using Memoptimized Rowstore Fast Ingest*

You can monitor the usage of the fast ingest buffers by querying the view `V$MEMOPTIMIZE_WRITE_AREA`.

# FLEXIBILITY

IoT is currently in its infancy, and new use cases come with each new device. Being able to quickly adapt to changes in data formats and efficiently analyze and manage large volumes of data is critical. This section of the paper will discuss the different aspects of Oracle Database that make it possible to handle large volumes of data while still providing a very flexible schema.

## JSON Support

To ensure maximum schema flexibility, IoT data is often sent as JSON[1]. JSON documents allow IoT systems to be effectively schemaless, as each document can contain a different set of attributes and values. Oracle Database 19*c* offers native support for JSON, just as with XML in the past. Oracle is aware of the JSON data structures and persists JSON data in its native structure within the Database. However, unlike XML, there is no new data type for JSON documents. Instead, JSON is stored as text, in any table column, using a VARCHAR2, CLOB or BLOB data type. Using existing data types ensures that JSON data is automatically supported with all existing database functionality, including Oracle Text and Database In-Memory. This allows the data to be ingested and processed in real-time.

It's also extremely easy for existing database users or applications to access information within a JSON document, using the standard dot notation in SQL. The command below extracts the city for each meter reading from within the JSON column stored in the Meter_Reading table.

```
SELECT m.json_column.address.city FROM   Meter_Readings m;
```

*Figure 12: Example of selecting the city from the address stored in a JSON document in the Meter_Readings table*

## Partitioning

Managing terabytes or even petabytes of data demands efficiency and scalability. Oracle Partitioning gives you both of these abilities while being completely transparent to the application queries. Just as a grocery store is divided into different departments (fruit and vegetables, meat, soft drinks, etc.), partitioning allows a table, index or index-organized table to be subdivided into smaller pieces. Each piece is called a partition and has its own name and its own storage characteristics. A partitioned table has multiple pieces that can be managed either collectively or individually from a database administrator's perspective. However, from the perspective of the application, a partitioned table is identical to a non-partitioned table.

Partitioning can provide tremendous benefits to an ingest-heavy workload by improving manageability, availability, and performance.

### Partitioning for manageability

Consider the case where two year's worth of smart meter readings or 100 terabytes (TB) are stored in a table. Each hour a new set of meter readings needs to be loaded into the table, and the oldest hour's worth of data needs to be removed. Suppose the meter readings table is ranged partitioned by the hour. In that case, the new data can be loaded directly into the latest partition, while the oldest hour of data can be removed in less than a second using the following command:

```
ALTER TABLE meter_readings DROP PARTITION MAR_25_2019_08;
```

*Figure 13: Example of how partitioning helps easy the management of large volumes of data by dropping older data.*

Partitioning can also help you compress older data. For example, you can load the data into an uncompressed partition while the rest of the table is stored in a compressed format; after some time, the current partition can also be compressed using an `ALTER TABLE MOVE PARTITION` command.

### Partitioning for performance

Partitioning also helps improve query performance by ensuring only the necessary data will be scanned to answer a query, just as the aisles in a grocery store allow you to access only the goods you are interested in. Let's assume that business users predominately access the meter reading data on a daily basis, e.g. the total amount of electricity used per day. Then RANGE partitioning this table by the hour will ensure that the data is accessed most efficiently. Only 24 partitions out of the 17,520 total partitions (2 years) need to be scanned to answer the business users' query. The ability to avoid scanning irrelevant partitions is known as partition pruning.

Further partition pruning is possible if the `METER_READINGS` table was sub-partitioned. Sub-partitioning allows the data within each partition to be sub-divided into smaller separate pieces. Let's assume the `METER_READINGS` table was sub-partitioned by HASH on `meter_id`, and 32 sub-partitions were specified. A query looking to see how much electricity a given household used on a given day would access only 1/32 of the data from the 24 range partitions that made up that day. Oracle uses a linear hashing algorithm to create sub-partitions. To ensure that the data gets evenly distributed among the hash partitions, it is highly recommended that the

[1] MORE INFORMATION ON JSON CAN BE FOUND AT HTTP://WWW.JSON.ORG/

number of hash partitions is a power of 2 (for example, 2, 4, 8, etc.). In a RAC environment, it should also be a multiple of the number of RAC nodes.

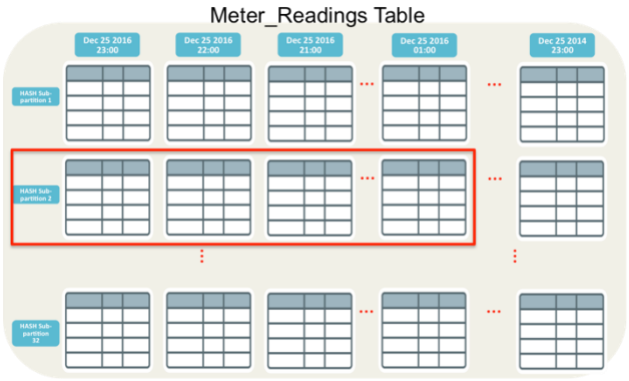


*Figure 14: Query to find how much electricity a household used in one-day accesses only one sub-partition of each hourly partition*

## Partitioning for Affinity

As discussed previously, partitioning helps alleviate table-level lock contention by enabling multiple direct path load operations to occur into the same table at the same time. It also helps minimize cluster-wide communication by allowing different partitions of a table to be affinitized to specific nodes in the cluster. Affinitizing partitions to specific nodes is critical to achieving linear scalability during data ingest operations in a RAC environment. You can think of affinitizing data the same way you organize your shopping list before going to the store. Listing all of the items you need from each department together means you can visit each aisle in the store just one rather than having to hop back forth between the departments looking for everything on your list.

To affinitize partitions to specific RAC nodes, you will need to do two things:

- Create a unique database service for each (sub)partition that connects to just one node
- Sort the incoming data by the (sub)partitioning key so that each array insert contains data for only one (sub)partition

### Creating Unique Database Services

In our example, the `METER_READINGS` table has 32 hash sub-partitions. Therefore, we need to create 32 unique database services, one for each sub-partition. These services will enable the application to connect to a specific node every time it needs to insert data into a particular sub-partition. The services should be evenly distributed across RAC nodes. Assuming we had 8 RAC nodes, we would create four services per node.

```
srvctl add service -d IoT  -s RAC_Node1_Partition1 -r RAC1 -a RAC4

srvctl add service -d IoT  -s RAC_Node1_Partition2 -r RAC1 -a RAC5

                                                    :

srvctl add service -d IoT  -s RAC_Node8_Partition32 -r RAC8 -a RAC1
```

*Figure 15: Statement required to create a unique database service for each of the 32 sub-partitions on an 8-node cluster*

If a node were to fail, each of the services originally attached to that node would failover to a different remaining node, thus ensuring no remaining node would be overwhelmed with additional work.

### Sorting the Incoming Data

To ensure each RAC node only inserts into a sub-set of partitions, we need to create array inserts or external tables that contain data for only one given (sub)partition. To achieve this, the incoming data needs to be sorted based on the partitioning strategy selected. In our example, the `METER_READINGS` table is RANGE partitioned by the hour and sub-partitioned by HASH on meter_id. Therefore, the data must first be sorted on the time_id column and then within each hour, the data must be sorted by hash sub-partition. But how do you determine which meter_ids belong in which hash sub-partition? First, you will need to convert the meter_id to an Oracle

number and then use the HASH() function in the open-source file lookup.c[2]. An example of the code necessary to determine which hash partition a meter_id belongs to is shown in Appendix C.

# REAL-TIME ANALYSIS

The timely analysis of the data captured in an IoT scenario can seriously affect actual business outcomes. It has the potential to:

- Optimize business processing and reduce operational costs
- Predict equipment failures
- Determine new product offerings or services
- Offer differentiated customer experiences

This section of the paper provides an overview of the different technologies offered by Oracle Database to improve real-time Analytics.

## Parallel Execution

Analytics on an IoT workload often require queries to be executed across trillions of data records in real-time. The key to achieving real-time analytics is to utilize all available hardware resources effectively.

As mentioned earlier, parallel execution is a commonly used method of speeding up operations within the database. It enables you to split a task into multiple sub-tasks executed concurrently. The Oracle database supports parallel execution right out-of-the-box. You can also use Automatic Degree of Parallelism (Auto DOP) to control how and when parallelism is used for each SQL statement. Parallel execution is a crucial feature for large-scale IoT deployments and is always recommended.

## Indexing

The most traditional approach to improving database queries' performance is to create indexes on the tables involved in the query, as they typically provide a faster access path to the data. You can think of indexes like signs that hang over every aisle in the grocery store, tell you exactly where you can find the coffee or the cereal. Oracle Database offers various index types, including B-Tree, Reverse Key, Function-Based, Bitmap, Linguistic and Text Indexes.

Partitioned tables can have either local or global indexes. A local index 'inherits' the partitioning strategy from the table. Consequently, each partition of a local index is built for the corresponding partition of the underlying table. This coupling enables optimized partition maintenance; for example, when a table partition is dropped, Oracle simply has to drop the corresponding index partition as well. A global partitioned index is partitioned using a different partitioning-key or partitioning strategy than the table. Decoupling an index from its table partitioning means that any partition maintenance operation on the table automatically causes index maintenance operations.

For IoT workloads that are predominately analytic in nature, Oracle recommends taking advantage of local indexes.

### Overhead of Keeping Indexes Transactionally Consistent

When an index is present on a table, every row inserted into the table must have a corresponding entry inserted into the index. This increases the CPU used per row for conventional inserts, the amount of redo generated per row, and the number of blocks modified per row. Indexes can introduce contention if multiple processes are inserting into the same place on the index. The presence of just one locally partitioned index increases the CPU usage to insert one row by 5x compared to when there is no-index. The volume of redo generated increases by 6x, as all changes to the index and the table need to be logged, and the number of block changes is 20 times higher. This results in a 5X drop in the number of rows you can be inserted per second. If you were to add two additional locally partitioned indexes, the ingest rate drops to 13x fewer rows per seconds than when there are no indexes present.

Direct path load operations using external tables also need to maintain indexes but do so more efficiently, as the index maintenance is not done on a row-by-row basis. Internally, the index maintenance is delayed until after all data is loaded but before committing the transaction and making the loaded rows visible. However, there is still a significant impact on performance when indexes are present.

### Partially Useable Indexes

2 LOOKUP.C IS AN OPEN SOURCE FILE BY BOB JENKINS AVAILABLE AT HTTP://WWW.BURTLEBURTLE.NET/BOB/C/LOOKUP.C, WHICH CONTAINS HASH FUNCTIONS

It is possible to minimize the impact of indexes on data ingestion by taking advantage of partially useable indexes. Partially useable indexes enable the creation of local and global indexes on just a subset of the partitions in a table. By allowing the index to be built only on the stable partitions in the table (older partitions with little or no data ingest), the index's presence will have minimal impact on ingest performance.

Analytic queries that only access data within indexed partitions will use the indexes as a faster access method. Queries that access data only in the non-indexed partitions will have to scan the entire partition. Still, since its being heavily modified, it will likely be in memory in the database buffer cache. Queries that access multiple partitions, some with indexes and some without, can take advantage of the query transformation called Table Expansion[3]. Table Expansion allows the optimizer to generate a plan that uses the index on the read-mostly partitions and full scan on the actively changing partitions.

## Time-Series Analysis

Often with an IoT workload, the business benefit comes from identifying a pattern or an anomaly rather than reviewing the individual entries. Oracle offers a rich set of SQL-based analytical features to support real-time analysis of IoT data. Windowing functions, for example, can be used to compute moving and cumulative versions of SUM, AVERAGE, COUNT, MAX, MIN, and many more functions. They provide access to more than one row of a table without the need to use a self-join. This makes it trivial to compare multiple values from the same device. In the example below, Oracle's built-in LAG function compares the current meter reading to the previous and calculates the difference. Note, each meter reading is stored as an element within a JSON document, hench the extended dot notation in the column names.

```
SELECT m.json_column.meter_id, time_id, m.json_column.value,
       LAG(m.json_column.value,1,0) OVER(ORDER BY m.json_column.meter_id, time_id) AS prev_reading,
       m.json_column.value-LAG(m.json_column.value,1,0) OVER(ORDER BY m.json_column.meter_id, time_id)
AS diff
FROM meter_readings m;
```

*Figure 16: Analytical SQL statement comparing current meter reading to previous and calculating the difference*

You may also want to take advantage of the built-in time series Machine Learning capabilities[4] of the Oracle Database. Time series models estimate the target value for each step of a time window, including up to 30 steps beyond the historical data. This type of predictive analysis allows businesses to prepare for potential spikes in demand, predict failures, determine when would be a good time to do preventive maintenance etc.

## Materialized Views

Pre-summarized and pre-aggregated data offer the potential to improve query performance significantly and overall system scalability by reducing the amount of system resources required by each query. Ideally, summaries and aggregates should be transparent to the application layer to allow them to be optimized and evolved without having to make any changes to the application itself.

Materialized Views [5](MVs) within the database offer the ability to summarize or aggregate data and be completely transparent to the application. A feature called "query rewrite" automatically rewrites SQL queries to access MVs where appropriate, so that materialized views remain transparent to the application.

## Oracle Database In-Memory

If a more ad-hoc approach to analyzing IoT data is required, consider taking advantage of Oracle Database In-Memory[6] (Database In-Memory). With Database In-Memory, IoT data can be populated into memory in a new in-memory optimized columnar format to improve ad-hoc analytic queries' performance.

[3] HTTPS://BLOGS.ORACLE.COM/OPTIMIZER/ENTRY/OPTIMIZER_TRANSFORMATIONS_TABLE_EXPANSION

[4] MORE INFORMATION ON TIME SERIES MACHINE LEARNING CAN BE FOUND IN THE ORACLE REFERENCE GUIDE

[5] MORE INFORMATION ON MATERIALIZED VIEW CAN BE FOUND IN THE ORACLE DATA WAREHOUSING GUIDE

[6] MORE INFORMATION ON IN-MEMORY CAN BE FOUND IN THE WHITEPAPER DATABASE IN-MEMORY WITH ORACLE DATABASE 12C RELEASE 2


Figure 17: Oracle's unique dual-format in-memory architecture

The database maintains complete transactional consistency between the traditional row format and the new columnar format, just as it maintains consistency between tables and indexes. The Oracle Optimizer is fully aware of what data exists in the columnar format. It automatically routes analytic queries to the column format and OLTP operations to the row format, ensuring outstanding performance and complete data consistency for all workloads without any application changes. There remains a single copy of the data on storage (in row format), so there are no additional storage costs or impact on data loading, as no additional redo or undo is generated.

Unlike other In-Memory columnar solutions, not all of the data in the database needs to be populated into memory in the columnar format. With Database In-Memory, only the performance-critical tables or partitions should be populated into memory. This allows businesses to deliver real-time analytics on the data of interest while storing historical data efficiently on disk, at a fraction of the cost. For queries that access data both in the row and the columnar format, the database will use its extensive optimizations across memory, flash and disk to access and aggregate the data.

### Overhead of Keeping IM Column Store Transactionally Consistent

The overhead of keeping the IM column store transactionally consistent varies depending on several factors, including the data ingest method and the in-memory compression level chosen for a table. For example, tables with higher compression levels will incur more overhead than tables with lower compression levels.

## CONCLUSION

With the surge in smart devices' popularity comes a considerable increase in the frequency and volume of data being ingested into databases. Ingesting and analysing rapidly increasing data volumes in a performant and timely manner is critical for businesses to keep their competitive advantages.

Oracle Database is more than capable of ingesting 100s of millions of rows per second and scaling to petabytes of data. It is the best choice for a mission-critical IoT workload given its support for flexible schemas, ultra-fast In-Memory analytic and industry-leading availability.

## APPENDIX A - TEST RESULTS

To demonstrate the benefits of the recommendations outlined in this paper, we conducted tests on an Exadata X6-2 Full Rack (8 compute nodes and 14 storage cells). The database software was Oracle Database 12c Release 2, which was configured using a basic init.ora parameter file, with no underscore parameters. The tablespaces used automatic segment space management and local extent management.

The table used in the tests consisted of eight columns, seven numeric columns and one date column. It was ranged partitioned on one of the numeric columns (32 partitions), and inserts were affinitized by instance. That is to say, inserts on each RAC node only went to a subset of partitions and no other RAC node inserted into those partitions.

Data was inserted via a C program using the OCI (Oracle Call Interface) driver, either one row at a time or via array inserts with an array size of 100 and 5000 rows. The commit frequency was also varied, as listed in the tables below.

As well as varying the insert methods, the tests were run with and without indexes on the table. Initially, one and then three locally partitioned indexes were created. Since the indexes were local and the inserts were affinitized, these indexes had minimal contention. To demonstrate an extremely high-contention case, we created one non-partitioned index on the table's date column. Each insert statement specifies the current time and date in the row, so all of the sessions were trying to modify the same part of the index. In contrast, we created another non-partitioned index that used the partitioning key as the leading column to see the impact of a global index when the sessions would be inserting into a different place in the same index.

## Conventional Insert Results

| ID | Description of Conventional Insert Test | Rows Inserted Per Second on one Full Rack | Rows Inserted Per Second on Each Node |
|---|---|---|---|
| 1 | Array Insert (100 rows), no Index, commit every 100 rows | 100M rows/s | 12.5M rows/s |
| 2 | Array Insert (5000 rows), no Index, commit every 5000 rows | 200M rows/s | 25M rows/s |
| 3 | Single-row insert, no index, commit every 100 rows | 9M rows/s | 1.125M rows/s |
| 4 | Single-row insert, no index, Commit every row | 2.6M rows/s | 325,000 rows/s |
| 5 | Array Insert (100), 1 local partitioned index commit every 100 | 20M rows/s | 2.5M rows/s |
| 6 | Array Insert (100), 3 local partitioned Indexes, commit every 100 | 7.5M rows/s | 937,000 rows/s |
| 7 | Array Insert (100), one non-partitioned global Index on date column which was a high-contention point, commit every 100 rows | 2.5M rows/s | 312,500 rows/s |
| 8 | Array Insert (100 rows), one non-partitioned global Index with partitioning key as the leading edge, commit every 100 rows | 8.7M rows/s | 1.09M rows/s |

Figure 16: Conventional Insert test results for a full Exadata X2-6 rack and each node

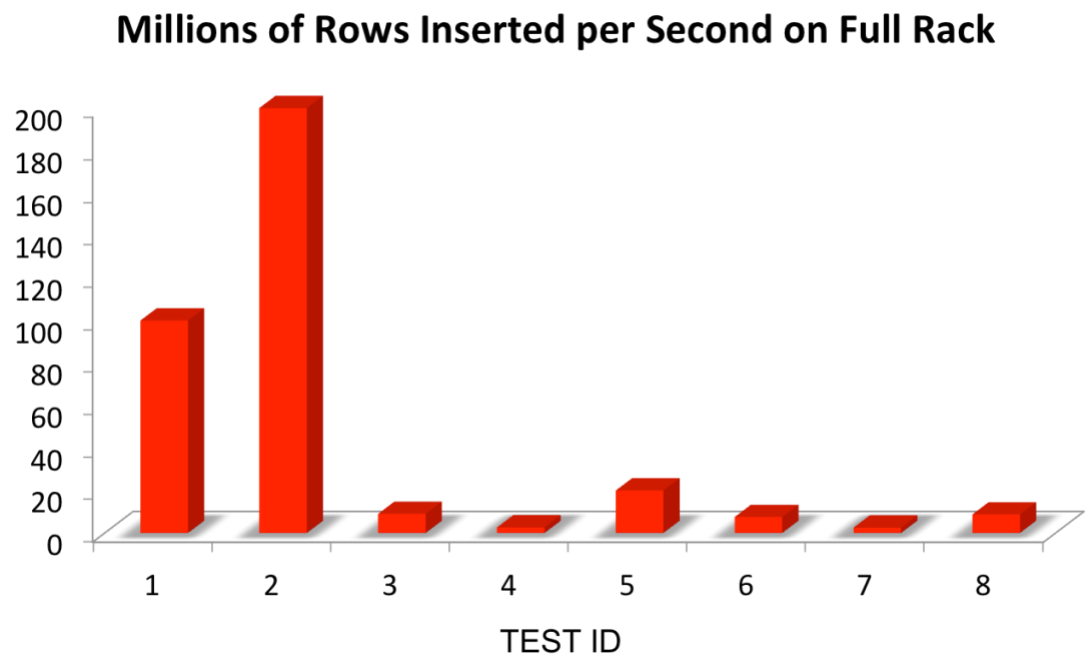


Figure 17: Graph of Insert test results for a full Exadata X2-6 rack

As you can see, by applying some simple application tuning, Oracle can achieve up to 100 million inserts per second using a single rack. Add additional Exadata racks to scale-out this solution.

## Memoptimized Rowstore For Streaming Data Results

The same schema was used for the streaming data tests but only a single node of the Exadata X6-2 was used. Data was inserted via a C program using the OCI (Oracle Call Interface) driver just as before. The inserts done either one row at a time or via array inserts with an array size of 100 rows. As well as varying the insert methods, the tests were run with and without a local index on the table.

| ID | Description of Streaming Data Insert Test | Rows Inserted Per Second per Node |
|---|---|---|
| 1 | Array Insert (100 rows), no Index | 23.2M rows/s |
| 2 | Array Insert (100 rows), with 1 local partitioned Index | 9.5M rows/s |
| 3 | Single-row insert, commit every 100, no index | 1.8M rows/s |
| 4 | Single-row insert, commit every 100, with 1 local partitioned Index | 1.7M rows/s |
| 5 | Single-row insert, commit every row, no index | 446,000 rows/s |
| 6 | Single-row insert, commit every row, with 1 local partitioned Index | 445,000 rows/s |

Figure 18: Memoptimized Ingest test results for a single node of an Exadata X2-6 rack

# APPENDIX B EXAMPLE OF AN ARRAY INSERT IN PYTHON

```
#!/usr/bin/python
#------------------------------------------------------------
# note: download and install cx_Oracle from https://pypi.python.org/pypi/cx_Oracle/5.2.1
# make sure you download the right version
#
# example adapted from http://www.juliandyke.com/Research/Development/UsingPythonWithOracle.php
#------------------------------------------------------------
import cx_Oracle
import sys
def print_exception(msg, exception):
  error, = exception.args
  print '%s (%s: %s)' % (msg, error.code, error.message);
def main():
  username = 'scott'
  password = 'tiger'
  dbname   = 'inst1'

  sqlstmt  = 'insert into Meter_Reading(meterNo, readingDate, loc) values (:mno, :rdate, :loc)';
  try:
```

```
    conn = cx_Oracle.connect(username,password,dbname)
  except cx_Oracle.DatabaseError, exception:
    msg = 'Failed to connect to %s/%s@%s' % (username,password,dbname)
    print_exception(msg, exception)
    exit(1)
  # array insert
  cursor = conn.cursor()
  try:
    cursor.prepare(sqlstmt)
  except cx_Oracle.DatabaseError, exception:
    print_exception('Failed to prepare cursor', exception)
  else:
    data_array = []
    # populate array
    for i in range(1,100):
      data_array.append( (i*10, 'FEBRUARY ' + str(i*10),
                          'LOCATION ' + str(i*10)) )
    try:
      cursor.executemany(sqlstmt, data_array)
      conn.commit()
    except cx_Oracle.DatabaseError, exception:
      print_exception('Failed to insert rows', exception)
    else:
      # no return value for executemany(), assume length of data array
      print 'Inserted %d rows' % len(data_array)
  # close connection
  finally:
    cursor.close()
    conn.close()

#------------------------------------------------------------
# standard template
#------------------------------------------------------------
if __name__ == '__main__':
  main()
```

# APPENDIX C – EXAMPLE OF DETERMINING WHICH HASH PARTITION DATA BELONGS

Below is an example of the code that can be used to determine which hash sub-partition a meter_id belongs to. This sample code takes advantage of the HASH() function provided in the open source file http://www.burtleburtle.net/bob/c/lookup.c. However, we have changed the ub4 to an unsigned int rather than an unsigned long.

For a given meter_id the function will convert the id to an Oracle Number, apply the hash() function from lookup2.c, and keep applying the mask until the value is less than then the number hash partitions. The function then returns the partition number that this meter_id belongs to.

The function takes 4 inputs:

1. id (i.e. meter_id)
2. The number of (sub)partitions (i.e. 32)
3. mask
4. .errhp - OCI error handler (needed to use the OCINumberFromInt() call)

You can use the following command to compile this code:

```
make -f $ORACLE_HOME/rdbms/demo/demo_rdbms.mk build EXE=id2bucket OBJS="id2bucket.o lookup2.o"
```

## Sample Code

```
/* Copyright (c) 2021, Oracle and/or its affiliates. All rights reserved. */

/*
   NAME
     id2bucket
   DESCRIPTION
     Contains a function to return the hash bucket for a given meter_id
   PUBLIC FUNCTION(S)
     id2bucket
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <oci.h>

/* number of partitions */
#define HASH_PARTITIONS 32

/* check for OCI error codes */
void checkerr(errhp, status)
OCIError * errhp;
sword status; {
  text errbuf[512];
  sb4 errcode = 0;

  switch (status) {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid * ) errhp, (ub4) 1, (text * ) NULL, & errcode,
      errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
```

```
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}

/*
 * NAME: getmask
 * DESCRIPTION: return the mask based on the number of partitions
 * PARAMETERS:
 *   int hash_partitions: number of hash partitions
 *
 * RETURNS:
 *   mask - hash mask to use for the id2bucket() function
 */
ub4 get_mask(int hash_partitions) {
  /* find first mask greater than # of hash partitions */
  /* we will only process at most 32k partitions */
  int i;
  for (i = 1; i - 1 < hash_partitions; i = i << 1) {
    /* null */
  }
  /* mask is 0x..fff */
  return i - 1;
}

/*---------------------------------------------------------------
 * NAME: id2bucket
 * DESCRIPTION: for a given id, return the corresponding hash bucket
 * PARAMETERS:
 *   int *id            : meter id
 *   ub4 hash_partitions: number of hash partitions
 *   ub4 hash_mask      : hash mask should be > # of hash partitions
 *   OCIError *errhp    : OCIError handler
 * RETURNS:
 *   bucket - this is the group/hash partition in which this
 *            record should go into (from 0 to HASH_PARTITIONS-1)
 *
 * This uses the hash() function from
 * http://burtleburtle.net/bob/c/lookup.c
 * NOTE: in lookup.c, ub4 should be changed to an unsigned int
 * (not unsigned long int to match oracle specs)
 *---------------------------------------------------------------
 */
sword id2bucket(int id, int hash_partitions, ub4 hash_mask, OCIError * errhp) {
  sword errcode = 0;              /* error code for OCI functions */
  ub1 id_onum[sizeof(OCINumber)]; /* allocate for OCINumber */
  ub4 hashval = 0;                /* hash value */
  ub4 mask = 0;                   /* hash mask */
  sword bucket = -1;              /* bucket id */
  ub1 *bufP = id_onum;            /* data buffer */
  ub4 bufLen;                     /* buffer length */

  /* initialize buffer */
  memset((void *) id_onum, 0, sizeof(OCINumber));

  /* convert id to oracle number */
  errcode = OCINumberFromInt(errhp,
                             (const void *) &id,
                             (uword) sizeof(int),
                             (uword) OCI_NUMBER_UNSIGNED,
                             (OCINumber *) &id_onum);

  /* check for errors in converting to oracle number,
     for any errors, return the error code */
  checkerr(errhp, errcode);
  if (errcode != OCI_SUCCESS) {
```

```
    printf("error\n");
    return errcode;
  }
  bufLen = bufP[0];       /* buffer length is first byte */
  hashval = hash(bufP + 1, bufLen, 0); /* get hash value */
  mask = hash_mask;               /* starting hash mask */

  /* find hash bucket, applying hash mask as required */
  bucket = hashval & mask;
  if (bucket >= hash_partitions)
    bucket = bucket & (mask >> 1);
  return bucket;
}

#ifdef UNIT_TEST
main() {
  OCIEnv   *envhp = NULL;
  OCIError *errhp = NULL;
  sword    errcode = 0;
  int      i;
  /* get mask based on the number of partitions */
  ub4 mask = get_mask(HASH_PARTITIONS);
  /* allocate OCI handles
     note: caller of functions should set this up so we do not have
     to create it on each call to id2bucket()
   */
  errcode = OCIEnvCreate((OCIEnv * * ) & envhp, (ub4) OCI_DEFAULT,
    (dvoid * ) 0, (dvoid * ( * )(dvoid * , size_t)) 0,
    (dvoid * ( * )(dvoid * , dvoid * , size_t)) 0,
    (void( * )(dvoid * , dvoid * )) 0, (size_t) 0, (dvoid * * ) 0);

  if (errcode != 0) {
    (void) printf("OCIEnvCreate failed with errcode = %d.\n", errcode);
    exit(1);
  }
  (void) OCIHandleAlloc((dvoid * ) envhp, (dvoid * * ) & errhp, OCI_HTYPE_ERROR,
    (size_t) 0, (dvoid * * ) 0);

  /* test for 10000 numbers */
  for (i = 0; i < 10000; i++) {
    printf("key %3d: %5d\n", i, id2bucket(i, HASH_PARTITIONS, mask, errhp));
  }

  /* check mask */
  for (i = 1; i < 65536;) {
    printf("partitions: %d, mask: 0x%x\n", i, get_mask(i));
    i = i << 1;
  }
}#

endif.
```

## CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com.
Outside North America, find your local office at oracle.com/contact.

blogs.oracle.com facebook.com/oracle twitter.com/oracle



Best Practices for Implementing High Volume IoT Workloads with Oracle Database
March, 2021