Business / Technical Brief

JSON in Oracle Database: Performance Considerations

On-Premises and Cloud
(including Autonomous Database),
SODA (Simple Oracle Document Access API),
Oracle Database API for MongoDB

November 2022, Version 2.0

Table of contents

Purpose

This document provides an overview of performance tuning best practices for JavaScript Object Notation (JSON) stored and processed in Oracle Databases. Applying these best practices will allow Developers, DBA's, and Architects to proactively avoid performance issues and ensure the applications and systems they design operate at peak performance.

The hyperlinks through out this document provide access to documentation, additional information, examples, and free hands-on training.

JSON Developer Guide:

19c, 21c

Brief Introduction to JSON in the Oracle Database

In 2014, Oracle released Oracle 12.1.0.2, which added native JSON support across all Oracle Database Editions. Before this release, JSON was often stored in NoSQL databases which lacked functionality and a data consistency model, which forced developers to add additional code to ensure data integrity.

To compensate for NoSQL shortcomings, developers included using relational databases or other data storage technologies, for example to run analytical queries. The inclusion of native JSON support in 2014 eliminated the need for these additional specialized data storage technologies, which greatly accelerates development by eliminating integration work, simplifying deployments, reducing risk, and reducing cost.   Additionally, storing, processing, and analyzing JSON using standardized SQL operators significantly reduces adoption time, required skills and empowers non-developers to easily work with JSON data.

The Oracle Database provides Native JSON support. JSON works with all Oracle Database Capabilities including Options, Oracle management Packs, Frameworks, Architectures, and Security. JSON stored in the Oracle database also benefits from the Performance, Scalability, Availability, Extensibility, Portability, and Security of the Oracle Database. Access to JSON stored in the Oracle Database is the same as access other database access methods, including OCI, .NET, and JDBC.

Additional information regarding JSON in the Oracle Database can be found in the JSON Developer's Guide.

Performance Features and Techniques – an in-depth view

The following section describes the features discussed in the workload section in more detail:

Storing JSON Data in Oracle Databases for optimal performance

JSON can be stored using columns whose data types are VARCHAR2, CLOB, BLOB, or JSON. Whichever type you use, you can manipulate JSON data as you would any other data of those types.

- For Oracle 21c, it is recommended to use the native JSON type, optimized for queries and efficient (partial) updates. An IS JSON check constraint can be defined on the JSON columns to enforce correct JSON syntax and can be disabled (not dropped) if the application can guarantee JSON correctness.
- For Oracle 19c, it is recommened to use the native BLOB data type, which is also optimized for queries and efficient updates.

- CLOBs are also supported, but should be avoided as CLOBS typically require twice storage space (and disk reads) due to UCS2 encoding.
- VARCHAR2s fields are also supported and can be considered if the maximum JSON document size is known or the JSON Documents are already stored in VARCHAR2 fields or if the simplicity of working with VARCHAR2 is preferred. VARCHAR2 values can hold up to 32 bytes.

Workload Types and Data Access Patterns

Database workloads can be classified as operational or analytical. Operational workloads, also known as online transaction processing systems, or OLTP, are transaction-oriented, have many users, and are designed for immediate response; for example, an automated teller machine (ATM) for a bank. OLTP systems support all data manipulation types. Typical operations involve transactions that insert or update data using a minimal number of rows. The performance goals for OLTP systems are transactional speed, throughput, and Database Concurrency. In contrast, Analytical workloads, such as online analytical processing (OLAP), data warehouses, and data lakes, are built for data analysis, have fewer users, and are designed to process large volumes of data. Typical operations include processing thousands or millions of rows using complex resource-intensive queries which join and aggregate data across many tables. OLAP systems are optimized for query.

“Native JSON support is significant because it used to be the case that one had to choose between more efficient JSON management in a pure-play DBMS, or the ability to integrate JSON data with other data, such as relational data…now, that choice is no longer necessary, because Oracle Database features both JSON efficiency and integrated data management."

Carl W Olofson IDC
bit.ly/nativeJSON_IDC

JSON document retrieval by key (OLTP)

Your workload selects individual JSON documents based on a relational column (key), with the JSON data stored in a second (payload) column. A primary key constraint on the key column enforces unique key values and also indexes them for fast lookups. If the key is not random (for example, using a sequence or identity column), then the index may become a hot spot in highly transactional systems because concurrent/subsequent inserts hit the same index block. **Hash Partitioning** the index on the key column will distribute inserts evenly to all partitions. SODA and MongoDB collections automatically have a primary key column – no further action is required for key-based document lookups.

JSON document retrieval by field value(s) (OLTP)

Here, one or few documents are selected by field values inside the JSON document. Path expressions in JSON_VALUE or JSON_EXISTS operators define the values. If the same path expressions are used repeatedly, a **function-based index** using JSON_VALUE is recommended. Indexing your field values of interest ensures the best possible performance by replacing full table scans with index lookups for data retrieval.

While indexing single fields within a JSON document can be easily accomplished, the indexing of arrays is more challenging. Function-based indexes cannot index array values (the function can only return one value per JSON data); in releases prior to Oracle 21c, **materialized views** can be used as an alternative: the materialized view expands the array into a relational column with multiple row entries that are then indexed as normal columns. Oracle’s comprehensive query rewrite framework automatically rewrites SQL statements against the JSON document to use the materialized view for fast data retrieval. With Oracle 21c, you index values in JSON arrays natively, using new **multi-value index** capabilities introduced in this release.

JSON document retrieval with full text search (OLTP, OLAP)

Some workloads only know the values of interest without knowing the path expressions to the fields within the JSON document, such as in ad-hoc queries on arbitrary documents. Oracle provides a **JSON Search Index** to improve the performance of such workloads. With JSON Search Indexes, the SQL/JSON operator JSON_TEXTCONTAINS allows selecting rows based on text search criteria, including word stemming and fuzzy search.

Extraction of JSON value for reporting or analytics (OLAP)

In reporting or analytical use cases, JSON data is mapped to the relational model for further processing using SQL. Commonly used SQL operations are joins (with other JSON or relational data), aggregates (sum, averages, window functions), or machine learning (classification, prediction). The SQL/JSON operator JSON_TABLE allows the mapping from JSON to the relational model. Whenever possible, Oracle Database optimizes multiple JSON query operators into a single JSON_TABLE statement (shown in the query execution plan).

For highly selective analysis (only a few JSON documents are selected based on field filter criteria) the access can be optimized with **indexes**. If many (but not all) rows are accessed, and indexes are no longer selective enough, **partitioning** the data should be considered to prune irrelevant partitions from the query. It is also recommended to leverage **parallel execution** whenever you work on large data volumes. The SQL/JSON operator JSON_TABLE can be parallelized without any limitations.

Suppose you run the same transformation from JSON to relational repeatedly, for example, a daily report or dashboard queries. In that case, a **materialized view** avoids the repeated execution of the same JSON_TABLE transformation at runtime altogether by materializing intermediate results in the view. JSON_TABLE materialized views are fast-refreshable so that they get efficiently and automatically refreshed after inserts or updates. A materialized view can also be used together with **Oracle Database In-Memory** to benefit from in-memory columnar compression and fast SIMD scans. This greatly improves performance, especially for analytical queries.

JSON Generation (OLTP, OLAP)

Oracle database added SQL/JSON operators to generate new JSON data from relational data and query results. Typical use cases are to modify the shape of one JSON document or to return the result of an analytical query as a JSON data extract. When only a few rows are accessed, then indexes provide fast access to them. If the JSON generation is built on many rows, then **materialized views** should be considered with the caveat that fast refresh is only supported in limited cases for JSON generation.

Performance Features and Techniques – an in-depth view

The following section describes the performance relevant features in more detail and with examples. In general, **normal SQL tuning techniques apply:** you can leverage the skills you already know.This shortens the learning curve and removes the fear DBA's and database managers have regarding adopting JSON in the Oracle Database. The main idea behind the tuning techniques is to reduce the data that needs to be read and processed:

Function-based Indexes

Function-based indexes can be created on specific keys or a combination of keys and optimize query operations that use SQL/JSON operators on the same keys. Function-based indexes are built using JSON_VALUE operators and support both bitmap and B-Tree index format.

The following creates a (unique) functional index on the `PONumber` key of our sample JSON document, accessed by the path expressions `'$.PONumber'`. The example assumes the JSON data is stored in a column called 'data' of a table called 'purchaseorder'.

```
create unique index PO_NUMBER_IDX on PURCHASEORDER po(
  json_value(po.DATA, '$.PONumber' returning number
                                   null on empty error on error));
```

Functional Indexes 19c, 21c

The `PONumber` values will be extracted (and indexed) as numbers. This affects range queries (numeric ordering instead of alphabetical ordering) and avoids data type conversions at runtime for mathematical operations or comparisons. Missing values will be indexed as SQL NULL value.

The following query uses the simplified JSON syntax. Because of the 'number()' item method, the index is used for data retrieval, as the plan shows.

```
select data from PURCHASEORDER po
where po.data.PONumber.number() = 200;

-----------------------------------------------------
| Id  | Operation                    | Name          |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |               |
|   1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER |
|*  2 |   INDEX UNIQUE SCAN          | PO_NUMBER_IDX |
-----------------------------------------------------
```

"Independent benchmarking of JSON databases based on the Yahoo! Cloud System benchmark revealed that Oracle is by far the leader in the space, outperforming all competitors [...]"

Accenture technical report:
Increase agility and cut bdevelopment time with JSON and Oracle, 2021
https://accntu.re/3lezy00

Multi-Value Index

A multi-value index is recommended if a path expression can select more than one value – this is common when accessing values inside a JSON array. The following creates a multi-value index on the field '`UPCCode` in the JSON array '`LineItems`' of our sample JSON document. The values are indexed as strings.

```
create multivalue index UPCCODE_INDEX on PURCHASEORDER po(
po.data.LineItems.Part.UPCCode.string());
```

JSON Multi-Value Index 21c

The multi-value index also uses B-Trees but is slightly slower than the functional index because resulting ROWIDs need deduplication. Therefore, if a path expression is known to return at most one value, the function-based index should be preferred. Multi-value indexing was introduced in Oracle 21c (for earlier reasons, materialized views can be used to accelerate access to arrays). The following query uses the multi-value index:

```
select data from PURCHASEORDER po where
po.data.LineItems.Part.UPCCode.string() = '13131092705';
-------------------------------------------------------------
| Id  | Operation                            | Name          |
-------------------------------------------------------------
|   0 | SELECT STATEMENT                     |               |
|*  1 |  TABLE ACCESS BY INDEX ROWID BATCHED| PURCHASEORDER |
|*  2 |   INDEX RANGE SCAN (MULTI VALUE)     | UPCCODE_INDEX |
-------------------------------------------------------------
```

JSON Search Index

Oracle Database supports *indexing* an entire JSON document using a search index, which is based on Oracle Full-Text index. The search index incorporates not only all values but their field names as well and allows full-text searches. The following creates a JSON Search index on 'purchaseorder'.

JSON Search Index: 19c, 21c

JSON Search Index blog

JSON DataGuide: 19c, 21c

```
create search index PO_FULL_IDX on PURCHASEORDER po (po.data) for json
parameters('SYNC (EVERY "FREQ=SECONDLY; INTERVAL=1") DATAGUIDE OFF');
```

The 'parameters' clause specifies that the index is asynchronous and gets synchronized every second. It is also possible to sync the index with every transaction commit, but this increases the cost of index maintenance and reduces the throughput of concurrent DML. The JSON Search index can also discover schema changes during DML operations with a feature called *JSON Dataguide* – it allows for example to auto-generate JSON_Table views. The clause 'DATAGUIDE OFF' disables this schema discovery and therefore reduces the costs of the JSON Search Index during DML operations.

The underlying data structure of a JSON Search index is posting lists, which are typically slower than B-Tree indexes. If the JSON Search index is used together with function-based indexes or the multi-value index, then those will be preferred by the Optimizer whenever possible. Because a JSON Search index indexes the entire JSON data, the size of this index will be significantly larger than other indexes, typically in the range of 20%-30% of the original data. JSON Search indexes support values inside JSON arrays and also full-text search operations.The following selects all documents with a 'Description' field that contains both the word 'Magic' and 'Christmas'. Instead of '{and}', one could also use '{near}' or '{not(…)}'. More information about the capabilities of JSON search indexes can be found in the documentation

```
select data from PURCHASEORDER po
where JSON_TEXTCONTAINS(po.data, '$.LineItems.Part.Description', 'Magic
{and} Christmas');
```

The query execution plan shows a JSON Search index as a 'Domain Index':

```
-----------------------------------------------------
| Id  | Operation                     | Name          |
-----------------------------------------------------
|   0 | SELECT STATEMENT              |               |
|   1 |  TABLE ACCESS BY INDEX ROWID| PURCHASEORDER |
|*  2 |   DOMAIN INDEX                | PO_FULL_IDX   |
-----------------------------------------------------
```

For workloads with many DML operations, it may be beneficial to use a single JSON search index over a large number of functional and multi-value indexes to reduce the amount of index maintenance (index synchronization after DML). Further optimization strategies are listed in the blog referenced on the right.

Materialized Views

You can use *materialized views* to improve the performance of frequent queries that access many rows (not key-based lookups that are index driven). A materialized view persist the result of a query. Subsequent queries that partially or fully match the query of the materialized view access the materialized data without having to re-run the original query (space is traded for speed).

JSON_Table

Materialized Views: 19c, 21c

In this document, we focus primarily on JSON_TABLE materialized views. The following creates a materialized view including values from the 'LineItems' array of our sample JSON document. As previously mentioned, with the use of materialized views one can index JSON array values in Oracle 19c, where multi-value JSON indexes are not available.:

```
create materialized view PO_MV build immediate
  refresh fast on statement with primary key as
    select po.id, jt.*
    from PURCHASEORDER po,
         json_table(po.data, '$' error on error null on empty
         columns (
              po_number  NUMBER        PATH '$.PONumber',
              userid     VARCHAR2(10)  PATH '$.User',
              NESTED                   PATH '$.LineItems[*]'
              columns (
                  itemno      NUMBER         PATH '$.ItemNumber',
                  description VARCHAR2(256)  PATH '$.Part.Description',
                  upc_code    NUMBER         PATH '$.Part.UPCCode',
                  quantity    NUMBER         PATH '$.Quantity',
                  unitprice   NUMBER         PATH '$.Part.UnitPrice')))
jt;
```

The transformation of our array values into multiple rows in our materialized view allows us to create an additional (secondary) index on fields of our JSON array as follows:

```
CREATE INDEX mv_idx ON PO_MV(upc_code, quantity);
```

SQL/JSON queries on the base table will now transparently rewrite to use materialized views and its indexes whenever possible. The following query is an example where Oracle automatically rewrites the query to use the materialized view and its secondary index, as seen in the execution plan:

```
select data from PURCHASEORDER po
where JSON_EXISTS(po.data, '$.LineItems[*]?(@.Part.UPCCode == 1234)');

--------------------------------------------------------------------
| Id  | Operation                                    | Name          |
…
|   4 |     MAT_VIEW ACCESS BY INDEX ROWID BATCHED| PO_MV         |
|*  5 |      INDEX RANGE SCAN                       | MV_IDX        |
--------------------------------------------------------------------
```

To keep the materialized view (MV) in sync with the underlying data (after DML) we created our materialized view as 'fast refreshable on statement'. This automates the process of refreshing and keeps the materialized view and the base table data consistent all the time. An in-depth discussion of the various refresh mechanisms of materialized views is out of the scope of this paper. Please consult the documentation for further details.

"The ability to run all the critical enterprise database loads—from analytical to transactional loads—in autonomous fashion, as well as support for ML, Graph, IoT, JSON and more, sets the Oracle Autonomous Database apart in the market for databases right now. Would you rather have nine specialized databases, each with its own separate security profile and management learning curve, or a single database that operates with all types of datasets autonomously?"

Holger Mueller, Constellation
bit.ly/ADB_Constellation

Oracle Partitioning

You can patition a table of with documents like you normally would to improve performance: Partitioning enables tables and indexes to be subdivided into individual smaller physical objects, so-called *partitions*. The data placement within a partitioned table is identified by a *partitioning key.* This key can be a relational column or a field from the JSON data. From the perspective of the application, a partitioned table is identical to a non-partitioned table.

JSON Partitioning: 19c, 21c
Partitioning Concepts: 19c, 21c

The following example creates a range partitioned table with the partition key extracted from the JSON document stored in column 'data', using a JSON_VALUE-based virtual column called 'po_num_vc';

```
CREATE TABLE part_j (id VARCHAR2 (32) NOT NULL PRIMARY KEY,
                     data JSON,
                     po_num_vc NUMBER GENERATED ALWAYS AS
                    (json_value (data, '$.PONumber' RETURNING NUMBER)))
PARTITION BY RANGE (po_num_vc)
   (PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (2000));
```

Queries filtering on the JSON field `'$.PONumber'` - the JSON field used as virtual column partitioning key - will transparently benefit from Oracle Partitioning: an optimization technique called partition pruning automatically excludes all irrelevant partitions, partitions that are known not to contain any data relevant for a query.

The following sample query only needs to access the first partition since the equality predicate of the query can only find matching records in this very partition. This is shown in the execution plan, with columns Pstart and Pstop both being 1.

```
select data from part_j
where json_value (data, '$.PONumber' RETURNING NUMBER) = 500;

-----------------------------------------------------------------
| Id  | Operation           | Name   | Time     | Pstart| Pstop |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT    |        | 00:00:01 |       |       |
|   1 |  PARTITION RANGE ALL|        | 00:00:01 |     1 |     1 |
|*  2 |   TABLE ACCESS FULL | PART_J | 00:00:01 |     1 |     1 |
-----------------------------------------------------------------
```

Oracle Partitioning has various mechanisms to partition a table, which are omitted here for space reasons. Please consult the documentation for more details. In general, using a relational column as partition key for larger JSON documents (average >32kb) is generally more performant during DML than using a JSON_VALUE virtual column because the latter requires the extraction of the partitioning key from the JSON prior of writing to the right partition.

Parallel Execution

JSON operations (for example queries or bulk updates) can be parallelized by processing JSON documents using multiple processes. This yields a more efficient use of hardware resource and is key for large-scale data processing.

Parallel Execution: 19c, 21c

Large data warehouses should always use parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution.

Parallel execution supports both queries and DML (inserts, updates). There are multiple ways to enable and configure parallel execution. For example, Oracle Autonomous Database automatically selects parallelism depending on the consumer group chosen for a connection. For databases that control parallelism manually, you can enable parallelism on a session level or decorate individual objects. For example, the following enables a *degree of parallelism* of 8 for our table 'purchaseorder'.

```
alter table PURCHASEORDER parallel 8;
```

If parallel execution is used, then the execution plan will show lines with '*PX'.*

```
|   1 |  PX COORDINATOR         |
|   2 |   PX SEND QC (ORDER)    |
```

Oracle In-Memory Columnar Storage

JSON data can be stored in the In-Memory Column Store (IM column store) to improve query performance. JSON values up to a size of 32 KBytes can directly be loaded and processed in-memory – together with other relational columns. Often, not all the values in the JSON document are relevant for an analytical query. In this case, memory can be used more efficiently by just moving the relevant JSON fields separately in memory: either by using virtual columns or an intermediate materialized view.

In-Memory JSON: 19c, 21c

In-Memory Guide: 19c, 21c

The following example adds a virtual column to our table 'purchaseorder' that extracts the 'zipCode' field from the order's address. The virtual column is added, and the table is enabled for in-memory processing.

```
alter table PURCHASEORDER add (ZIP varchar2(4000) generated always as
(JSON_VALUE(data, '$.ShippingInstructions.Address.zipCode.number()')));

alter table PURCHASEORDER inmemory;
```

The following analytical sample query counts the number of orders by zipCode and makes use of fast in-memory processing, as seen in the execution plan.

```
select zip, count(1) from PURCHASEORDER group by zip ;
-----------------------------------------------------
| Id  | Operation                   | Name          |
-----------------------------------------------------
|   0 | SELECT STATEMENT            |               |
|   1 |  HASH GROUP BY              |               |
|   2 |   TABLE ACCESS INMEMORY FULL| PURCHASEORDER |
-----------------------------------------------------
```

"We heavily use JSON whenever faced with unpredictable data from external APIs or custom user extensions. We decided to use Oracle Database as a document store that also supports SQL analytics over JSON and Blockchain."

Peter Merkert, CTO Retraced

www.retraced.co

Oracle Exadata Database Machine

Exadata accelerates JSON performance: queries with table and index scans can offload data search and retrieval processing to the Exadata Storage Servers. This offloading happens automatically and transparently for JSON operators, for example JSON_VALUE or JSON_EXISTS if used in the WHERE clause of a query. JSON documents up to 4KB can be offloaded to the Exadata Storage Server. Larger documents will be processed in the database.

The *STORAGE* term in an execution plan shows that offloading is done:

```
-----------------------------------------------------
| Id  | Operation                   | Name          |
-----------------------------------------------------
|*  3 |    TABLE ACCESS STORAGE FULL| PURCHASEORDER |
-----------------------------------------------------
```

Oracle Exadata Database Machine: doc

Oracle Real Application Clusters

Oracle Real Application Clusters (RAC) allow customers to run a single Oracle Database across multiple servers in order to maximize availability and enable *horizontal scalability* while accessing *shared storage*.

Using Oracle Real Application Clusters is transparent to the processing of JSON documents, and any SQL/JSON processing will automatically benefit.

Oracle Real Application Clusters doc: 19c, 21c

Oracle Sharding

Oracle Sharding is also a *horizontal scaling* technique, but unlike RAC, it uses a *shared-nothing* architecture. Sharding allows JSON documents to scale to massive data and transactions volume and support data sovereignty. JSON documents are distributed to the individual database table shards according to the sharding key, which can be a relational column or a JSON field.

Oracle Sharding Doc:19c, 21

Sharded JSON collections: doc

Using Oracle Sharding is transparent to the processing of sharded JSON documents. For many operations the processing of sharded documents will take place only on the database owning a specific shard of the documents whereas cross-shard queries will transparently collect and aggregate result data from all relevant shards.

This concludes the JSON performance tuning features in Oracle Database. The following summarized performance relevant topics for the JSON Document Store APIs (MongoDB collections and SODA collections):

Performance Tips for SODA Collections

Oracle Database offer APIs that allow to access JSON data as *collections*: The *Oracle Database API for MongoDB* and the *Simple Oracle Document Access API - SODA*. Conceptually, JSON collections stores JSON data (called documents) in automatically generated tables (so that SQL access is also possible). SODA supports the same storage options as regular tables with JSON data, and the same recommendations apply: use BLOB on Oracle 19c and the native JSON type on Oracle 21c.

SODA API

Database API for MongoDB

Users will typically work with JSON collections using native language drivers, for example, SODA for Java or SODA for Python. SODA native language drivers generally provide more throughput (operations per second) than the REST driver (SODA for REST).

It is recommended to configure the SODA drivers as follows:

- Enable **SODA Metadata Cache**
  The SODA driver needs to know the metadata of each JSON collection (the column names, types, etc.). By enabling the metadata cache, roundtrips to the database can be saved, improving latency and throughput.
- Enable **Statement Cache**
  Statement caching improves performance by caching executable statements that are used repeatedly, such as in a loop or in a method that is called repeatedly. For Java, the statement cache is enabled using JDBC.
- For load-balanced systems: **turn off DNS caching**
  Load balancing allows to distribute SODA operations across different nodes. If DNS caching is turned on, then all connections are likely to use the same node and nullifying the load balancing. For Java, the following system property should be set: `inet.addr.ttl=0`

The database performance tuning techniques also apply to SODA: for example, **SODA collections can be partitioned or sharded,** and queries can be accelerated using **indexes and/or materialized views**. SODA operations are translated automatically to equivalent SQL operations: for example, a

SODA query becomes a SELECT with a JSON_EXISTS operator in the WHERE clause.

The SQL operations can be retrieved from the v$sql database view or by **enabling logging in the SODA driver** directly: In Java,the standard package for logging is used – it can be enabled for SODA as follows:

```
java –classpath "..." –Doracle.soda.trace=true –
      Djava.util.logging.config.file=logging.properties <program>
```

- `‘oracle.soda.trace=true’` enables the logging of SQL statements.
- ‘logging.`java.util.logging.config.file’`defines the path to the java.util.logging configuration file, which allows different logging levels: FINEST is the most verbose logging level.

Further Information – Links

Oracle XE

Oracle Standard Edition

Oracle Enterprise Edition

Oracle Exadata Cloud Service

Oracle Exadata Cloud at Customer

Oracle Exadata Database Machine

Oracle Database Cloud Service

Oracle Autonomous JSON

Oracle Autonomous Transaction Processing

Oracle Autonomous Data Warehouse