Performance with in database JSON

Contents

Initial requirements

- SSH private key to Access the database server in the cloud. This private key is provided along with this manual.
- SSH client app, to login to the database server
- Database server public IP

JSON documents in the database

JSON queries

Access the CDB, PDB creation and administration

Below are described the first steps to access the container database (CDB) and create a pluggable database (PDB).

Access to the database server as "**opc**" using ssh. Then gain access to the "oracle" user, and use Sql*Plus to run the following commands:

```
ssh -i id_rsa opc@<db server public ip>

## Gain access to the "oracle" user:

[opc@rdbms21coniaas ~]$ sudo su - oracle

## Use Sql*Plus to show the PDB that will be used for the lab

[oracle@rdbms21coniaas ~]$ sqlplus / as sysdba

SQL*Plus: Release 21.0.0.0.0 - Production on Wed Jan 12 15:44:16 2022
Version 21.3.0.0.0

Copyright (c) 1982, 2021, Oracle.  All rights reserved.


Connected to:
Oracle Database 21c Enterprise Edition Release 21.0.0.0.0 - Production
Version 21.3.0.0.0

SQL> show pdbs
```

```
    CON_ID CON_NAME                        OPEN MODE  RESTRICTED
---------- ------------------------------ ---------- ----------
         2 PDB$SEED                       READ ONLY  NO
         3 ORCLPDB1                       MOUNTED
         5 PDBSOE                         READ WRITE NO
SQL> exit
```

Create a JSON table and populate it

Connect to the PDBSOE PDB, and run the following SQL statements:

```
$ sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe

## Drop the existing tables

drop table OI_JSON_ORDERS purge;
drop table OI_JSON_ORDER_ITEMS purge;

## Create the new tables
## Regular scalar column ID is used as a primary key
## O_JSON column will be used to store JSON documents, and we enforce a check
constraint on it, to check the correct JSON semantic of the inserted values

create table OI_JSON_ORDERS
(
ID number(12),
O_JSON VARCHAR2(4000),
CONSTRAINT O_JSON_insert_pk primary Key (id),
CONSTRAINT O_JSON_check CHECK (O_JSON IS JSON)
);

## Now we will create new data in the OI_JSON_ORDERS table
## We use database JSON API to format a JSON document from the ORDERS
relational table

alter session enable parallel DML;
alter session force parallel query parallel 2;
set timing on

insert /*+ APPEND NOLOGGING */
into OI_JSON_ORDERS (id,O_JSON)
select O.order_id,
       json_object (
       'ORDER_ID' value O.ORDER_ID,
       'ORDER_DATE' value O.ORDER_DATE,
       'ORDER_MODE' value O.ORDER_MODE,
       'CUSTOMER_ID' value O.CUSTOMER_ID,
       'ORDER_STATUS' value O.ORDER_STATUS,
       'ORDER_TOTAL' value O.ORDER_TOTAL,
       'SALES_REP_ID' value O.SALES_REP_ID,
```

```
       'PROMOTION_ID' value O.PROMOTION_ID,
       'WAREHOUSE_ID' value O.WAREHOUSE_ID,
       'DELIVERY_TYPE' value O.DELIVERY_TYPE,
       'COST_OF_DELIVERY' value O.COST_OF_DELIVERY,
    'WAIT_TILL_ALL_AVAILABLE' value O.WAIT_TILL_ALL_AVAILABLE,
    'DELIVERY_ADDRESS_ID' value O.DELIVERY_ADDRESS_ID,
    'CUSTOMER_CLASS' value O.CUSTOMER_CLASS,
    'CARD_ID' value O.CARD_ID,
    'INVOICE_ADDRESS_ID' value O.INVOICE_ADDRESS_ID
    ) as mijson
from orders O;

1429790 rows created.

Elapsed: 00:00:33.64

SQL> commit;

Commit complete.

Elapsed: 00:00:01.50
```

We used the JSON_OBJECT function to format a JSON document out of relational scalar columns.

Use JSON and non JSON tables in the same query

Now let's use SQL to write a query that access data in both a relational and a JSON table:

```
## First we will run a "traditional" query, joining two relational tables.

select W.WAREHOUSE_NAME, sum(O.ORDER_TOTAL)
from ORDERS O,
WAREHOUSES W
where W.WAREHOUSE_ID =  O.WAREHOUSE_ID
and     W.warehouse_name in
('McsRxsWjRxXMFDcobjhEIDdEsO','5eH6XK38SRmNEZCUg43EDIjDICDhbV','PLlypy')
group by W.WAREHOUSE_NAME
order by 1
/

WAREHOUSE_NAME                     SUM(O.ORDER_TOTAL)
---------------------------------- ------------------
5eH6XK38SRmNEZCUg43EDIjDICDhbV                7190505
McsRxsWjRxXMFDcobjhEIDdEsO              7368607
PLlypy                                  7197962

Elapsed: 00:00:00.29

## Now let's write a slightly distinct query to join WAREHOUSES and
OI_JSON_ORDERS table. We can nicely join the two tables on WAREHOUSE_ID, which
is a regular column in the WAREHOUSES table, and $.WAREHOUSE_ID, which is a
field of the JSON document stored in OI_JSON_ORDERS table.
```

```
select W.WAREHOUSE_NAME, sum(to_number(json_value (OI.O_JSON,
'$.ORDER_TOTAL'))) as TOTAL
from    OI_JSON_ORDERS OI,
        WAREHOUSES W
where   W.WAREHOUSE_ID = json_value (OI.O_JSON, '$.WAREHOUSE_ID')
and     W.warehouse_name in
('McsRxsWjRxXMFDcobjhEIDdEsO','5eH6XK38SRmNEZCUg43EDIjDICDhbV','PLlypy')
group by W.WAREHOUSE_NAME
order by 1;

WAREHOUSE_NAME                          TOTAL
---------------------------------- ----------
5eH6XK38SRmNEZCUg43EDIjDICDhbV          7190505
McsRxsWjRxXMFDcobjhEIDdEsO       7368607
PLlypy                           7197962

Elapsed: 00:00:12.84
```

The result of the two query is the same, but the performance of the second query is significantly worst than the performance of the 100% relational query.
In the remaining of the workshop, we will demonstrate several ways to speed-up queries involving JSON documents.

Create a partitioned JSON table

Next we will create and populate partitioned JSON table:

```
## Create an interval partitioned JSON table
## The partition key is a regular column, but could be a JSON field, as we will
see further

create table OI_JSON_ORDER_ITEMS
(
ID number(12),
order_date DATE NOT NULL,
OI_json VARCHAR2(4000),
CONSTRAINT oi_json_insert_pk primary Key (id),
CONSTRAINT OI_json_check CHECK (OI_json IS JSON)
)
PARTITION BY RANGE (order_date)
INTERVAL(NUMTOYMINTERVAL(1,'MONTH'))
(
    PARTITION OIJSON_P0 VALUES LESS THAN (TO_DATE('2007-02-01', 'YYYY-MM-DD'))
);

## Disable in-memory for that table, and populate it

alter table OI_JSON_ORDER_ITEMS no inmemory;
```

```
alter session enable parallel DML;
alter session force parallel query parallel 2;
set timing on

insert /*+ APPEND NOLOGGING */
into OI_JSON_ORDER_ITEMS (id,order_date,OI_json)
select O.order_id, O.order_date,
       json_object (
       'ORDER_ID' value O.ORDER_ID,
       'ORDER_DATE' value O.ORDER_DATE,
       'ORDER_MODE' value O.ORDER_MODE,
       'CUSTOMER_ID' value O.CUSTOMER_ID,
       'ORDER_STATUS' value O.ORDER_STATUS,
       'ORDER_TOTAL' value O.ORDER_TOTAL,
       'SALES_REP_ID' value O.SALES_REP_ID,
       'PROMOTION_ID' value O.PROMOTION_ID,
       'WAREHOUSE_ID' value O.WAREHOUSE_ID,
       'DELIVERY_TYPE' value O.DELIVERY_TYPE,
       'COST_OF_DELIVERY' value O.COST_OF_DELIVERY,
    'WAIT_TILL_ALL_AVAILABLE' value O.WAIT_TILL_ALL_AVAILABLE,
    'DELIVERY_ADDRESS_ID' value O.DELIVERY_ADDRESS_ID,
    'CUSTOMER_CLASS' value O.CUSTOMER_CLASS,
    'CARD_ID' value O.CARD_ID,
    'INVOICE_ADDRESS_ID' value O.INVOICE_ADDRESS_ID,
    'ITEMS' value json_arrayagg (
        json_object (
         'ORDER_ID' value  OI.ORDER_ID,
         'LINE_ITEM_ID' value  OI.LINE_ITEM_ID,
         'PRODUCT_ID' value  OI.PRODUCT_ID,
         'UNIT_PRICE' value  OI.UNIT_PRICE,
         'QUANTITY' value  OI.QUANTITY,
         'DISPATCH_DATE' value  OI.DISPATCH_DATE,
         'RETURN_DATE' value  OI.RETURN_DATE,
         'GIFT_WRAP' value  OI.GIFT_WRAP,
         'CONDITION' value  OI.CONDITION,
         'SUPPLIER_ID' value  OI.SUPPLIER_ID,
         'ESTIMATED_DELIVERY' value  OI.ESTIMATED_DELIVERY
    )
    )
    ) as mijson
from orders O,
    order_items OI
where O.order_id = OI.order_id(+)
group by
O.ORDER_ID,
O.ORDER_DATE,
O.ORDER_MODE,
O.CUSTOMER_ID,
O.ORDER_STATUS,
O.ORDER_TOTAL,
O.SALES_REP_ID,
O.PROMOTION_ID,
O.WAREHOUSE_ID,
O.DELIVERY_TYPE,
```

```
O.COST_OF_DELIVERY,
O.WAIT_TILL_ALL_AVAILABLE,
O.DELIVERY_ADDRESS_ID,
O.CUSTOMER_CLASS,
O.CARD_ID,
O.INVOICE_ADDRESS_ID;


1429790 rows created.

Elapsed: 00:01:47.06
SQL> SQL> commit;

Commit complete.

Elapsed: 00:00:00.13

## Exit the session
exit
```

Now we are ready to start with the performance tests.

Establish a baseline

We are going to retrieve the sum of unit_price, for customer_id= 733116 and date 24-FEB-09. Our baseline is the response time of that query against the relational tables:

```
sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe

set timing on

select sum(OI.unit_price)
from orders O, order_items OI
where O.customer_id = 733116
and trunc(O.order_date) = to_date('24-FEB-09','DD-MON-RR')
and O.order_id = OI.order_id;

SUM(OI.UNIT_PRICE)
------------------
              5146

Elapsed: 00:00:01.16

## Now let's capture the execution plan:

set autotrace traceonly explain statistics

select sum(OI.unit_price)
from orders O, order_items OI
```

```
where O.customer_id = 733116
and trunc(O.order_date) = to_date('24-FEB-09','DD-MON-RR')
and O.order_id = OI.order_id;

Elapsed: 00:00:00.01

Execution Plan
----------------------------------------------------------
Plan hash value: 2506602772

-------------------------------------------------------------------------------
-
--------------------------

| Id  | Operation                      | Name        | Rows  | Bytes
 | Cost (%CPU)| Time       |

-------------------------------------------------------------------------------
-
--------------------------

|   0 | SELECT STATEMENT               |             |     1 |    32
 |     9   (0)| 00:00:01 |

|   1 |  SORT AGGREGATE                |             |     1 |    32
 |           |       |

|   2 |   NESTED LOOPS                       |             |     1 |    32
 |     9   (0)| 00:00:01 |

|   3 |    NESTED LOOPS                |             |     3 |    32
 |     9   (0)| 00:00:01 |

|*  4 |     TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS     |     1 |    22
 |     5   (0)| 00:00:01 |

|*  5 |      INDEX RANGE SCAN                | ORD_CUSTOMER_IX |     2 |
 |     3   (0)| 00:00:01 |

|*  6 |     INDEX RANGE SCAN                 | ITEM_ORDER_IX     |     3 |
 |     2   (0)| 00:00:01 |

|   7 |    TABLE ACCESS BY INDEX ROWID       | ORDER_ITEMS       |     3 |
30
 |     4   (0)| 00:00:01 |

-------------------------------------------------------------------------------
-
--------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   4 - filter(TRUNC(INTERNAL_FUNCTION("O"."ORDER_DATE"))=TO_DATE('24-FEB-
09','DD
-MON-RR'))

   5 - access("O"."CUSTOMER_ID"=733116)
   6 - access("O"."ORDER_ID"="OI"."ORDER_ID")

Note
-----
   - this is an adaptive plan


Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
        10  consistent gets
         0  physical reads
         0  redo size
       579  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

The execution plan uses a couple of indexes, one indexing ORDER.CUSTOMER_ID and one indexing ORDER_ITEMS.ITEM_ID.

Rewrite the query on the JSON table

In the next step, we rewrite the query to access `OI_JSON_ORDER_ITEMS instead of the two relationl tables, ORDERS and ORDER_ITEMS`:

```
set autotrace off

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OIJ.OI_JSON, '$.CUSTOMER_ID') = 733116
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

SUM(ARR.UNIT_PRICE)
```

```
------------------
              5146

Elapsed: 00:00:03.20
```

Pay attention to and review the syntax: we are using native JSON API to access UNIT_PRICE in a nested path.
As expected, the result is the same. But the response time is three times higher. Let's capture the execution plan:

```
set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OIJ.OI_JSON, '$.CUSTOMER_ID') = 733116
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

Elapsed: 00:00:03.25

Execution Plan
----------------------------------------------------------
Plan hash value: 1435571113

-------------------------------------------------------------------------------
-
-------------------------------

| Id  | Operation          | Name             | Rows  | Bytes | Cost (%C
PU)| Time     | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-------------------------------

|   0 | SELECT STATEMENT   |                  |     1 |  1124 |   468K
(3)| 00:00:19 |       |       |

|   1 |  SORT AGGREGATE    |                  |     1 |  1124 |
   |          |       |       |

|   2 |   NESTED LOOPS           |                  |  1167K|  1251M|   468K
(3)| 00:00:19 |       |       |

|   3 |    PARTITION RANGE ALL   |                  | 14298 |    15M| 69606
(1)| 00:00:03 |     1 |1048575|
```

```
|*  4 |     TABLE ACCESS FULL    | OI_JSON_ORDER_ITEMS | 14298 |    15M| 69606
(1)| 00:00:03 |     1 |1048575|

|*  5 |    JSONTABLE EVALUATION |                     |       |       |
   |         |       |       |

--------------------------------------------------------------------------------
-
------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter(TO_NUMBER(JSON_VALUE("OIJ"."OI_JSON" FORMAT JSON ,
'$.CUSTOMER_ID'
 RETURNING

            VARCHAR2(4000) NULL ON ERROR))=733116)
   5 - filter(TRUNC("P"."ORDER_DATE")=TO_DATE('24-FEB-09','DD-MON-RR'))


Statistics
-----------------------------------------------------------
       44  recursive calls
        0  db block gets
   252515  consistent gets
   252160  physical reads
        0  redo size
      580  bytes sent via SQL*Net to client
       52  bytes received via SQL*Net from client
        2  SQL*Net roundtrips to/from client
        3  sorts (memory)
        0  sorts (disk)
        1  rows processed
```

The response time is explained by the "full table scan" access to OI_JSON_ORDER_ITEMS table: this is due to the lack of indexes on that table. Note that the response time is anyway pretty acceptable, as the table contains nearly 1.5 million rows.
In the following steps, we will improve that response time.

Leverage partition pruning

In the previous execution, partition pruning does not occur, because we are using the predicate " and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR')". As ARR.ORDER_DATE is a field in the JSON document, and not the regular column OI_JSON_ORDER_ITEMS.ORDER_DATE, the predicate is applied as a filter, but not an access path.

Writing the query differently will leverage partition pruning and boost the performance:

```
-- Use ORDER_DATE regular column instead of ORDER_DATE in the JSON document

sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe

set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OIJ.OI_JSON, '$.CUSTOMER_ID') = 733116
and OIJ.ORDER_DATE between to_date('24-FEB-09 00:00:00','DD-MON-RR HH24:MI:SS')
and to_date('24-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS');



Elapsed: 00:00:00.05

Execution Plan
----------------------------------------------------------
Plan hash value: 229940177

-------------------------------------------------------------------------------
-
-----------------------------------

| Id  | Operation             | Name            | Rows  | Bytes | Cost
 (%CPU)| Time   | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-----------------------------------

|   0 | SELECT STATEMENT      |                 |    1 |  1130 | 6982
5   (1)| 00:00:03 |   |      |

|   1 |  SORT AGGREGATE       |                 |    1 |  1130 |
       |       |      |      |

|*  2 |   FILTER              |                 |      |      |
       |       |      |      |

|   3 |    NESTED LOOPS       |                 | 64991 |    70M| 6982
5   (1)| 00:00:03 |   |      |

|   4 |     PARTITION RANGE ITERATOR|                 |    8 |  9024 | 6960
6   (1)| 00:00:03 |   KEY |   KEY |
```

```
|*  5 |      TABLE ACCESS FULL       | OI_JSON_ORDER_ITEMS |      8 |  9024 |
6960
6   (1)| 00:00:03 |   KEY |   KEY |

|   6 |     JSONTABLE EVALUATION    |                     |      |      |
       |       |      |      |

--------------------------------------------------------------------------------
-
---------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('24-FEB-09 23:59:59','DD-MON-RR
HH24:MI:SS')>=TO_DATE('24-
FEB-09

             00:00:00','DD-MON-RR HH24:MI:SS'))
   5 - filter(TO_NUMBER(JSON_VALUE("OIJ"."OI_JSON" FORMAT JSON ,
'$.CUSTOMER_ID'
 RETURNING VARCHAR2(4000)

             NULL ON ERROR))=733116 AND "OIJ"."ORDER_DATE"<=TO_DATE('24-FEB-09
23:59:59','DD-MON-RR HH24:MI:SS') AND

             "OIJ"."ORDER_DATE">=TO_DATE('24-FEB-09 00:00:00','DD-MON-RR HH24:M
I:SS'))



Statistics
----------------------------------------------------------
        21  recursive calls
         0  db block gets
      3660  consistent gets
      3646  physical reads
         0  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

The access path is still FULL TABLE SCAN, but after pruning to the corresponding partition.

Create an index on a JSON field

To boost the query even more, we can create an index on a JSON field. We will then benefit from both partitioning and indexing.

```
-- We can index a json field to speed-up the query
-- We would tipically index CUSTOMER_ID
-- Pay attention to the syntax, we use native JSON API to define the index !!!

create index I_CUST_ID  on
OI_JSON_ORDER_ITEMS
(
    json_value (OI_JSON, '$.CUSTOMER_ID'  returning NUMBER(12) error on error
null on empty)
) LOCAL;

Index created.

Elapsed: 00:00:05.14

## Collect standard optimizer statistics on the index

exec dbms_stats.gather_index_stats ('SOE','I_CUST_ID')

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                                NESTED PATH '$.ITEMS[*]'
                                COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OI_JSON, '$.CUSTOMER_ID'  returning NUMBER(12) error on error
null on empty) = 733116
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

SUM(ARR.UNIT_PRICE)
-------------------
              5146

Elapsed: 00:00:00.01

## The response time was boosted, let's review the execution plan

set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                                NESTED PATH '$.ITEMS[*]'
                                COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OI_JSON, '$.CUSTOMER_ID'  returning NUMBER(12) error on error
null on empty) = 733116
```

```
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

Execution Plan
----------------------------------------------------------
Plan hash value: 1296896600

-------------------------------------------------------------------------------
-
-------------------------------------------------

| Id  | Operation                           | Name            | Row
s  | Bytes | Cost (%CPU)| Time      | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-------------------------------------------------

|   0 | SELECT STATEMENT                    |                 |
 1 |  1136 |   404K  (3)| 00:00:16 |       |       |

|   1 |  SORT AGGREGATE                     |                 |
 1 |  1136 |        |          |       |       |

|   2 |   NESTED LOOPS                              |                 |  11
67K|  1265M|   404K  (3)| 00:00:16 |       |       |

|   3 |    PARTITION RANGE ALL                      |                 | 142
98 |   15M|  5785   (1)| 00:00:01 |     1 |1048575|

|   4 |     TABLE ACCESS BY LOCAL INDEX ROWID BATCHED| OI_JSON_ORDER_ITEMS |
142
98 |   15M|  5785   (1)| 00:00:01 |     1 |1048575|

|*  5 |      INDEX RANGE SCAN                       | I_CUST_ID       |  57
19 |      |   65   (0)| 00:00:01 |     1 |1048575|

|*  6 |    JSONTABLE EVALUATION             |                 |
   |      |         |          |       |       |

-------------------------------------------------------------------------------
-
-------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access(JSON_VALUE("OI_JSON" FORMAT JSON , '$.CUSTOMER_ID' RETURNING
NUMBE
R(12,0) ERROR ON ERROR NULL ON

            EMPTY)=733116)
   6 - filter(TRUNC("P"."ORDER_DATE")=TO_DATE('24-FEB-09','DD-MON-RR'))
```

```
Statistics
----------------------------------------------------------
        61  recursive calls
         0  db block gets
       194  consistent gets
         2  physical reads
         0  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

Partition pruning does not occur (PARTITION RANGE ALL), but now the index is used.

We can manage to use both partition pruning and indexing to boost even more the performance.

```
set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                                NESTED PATH '$.ITEMS[*]'
                                COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OI_JSON, '$.CUSTOMER_ID'  returning NUMBER(12) error on error
null on empty) = 733116
and OIJ.ORDER_DATE between to_date('24-FEB-09 00:00:00','DD-MON-RR HH24:MI:SS')
and to_date('24-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS');

Execution Plan
----------------------------------------------------------
Plan hash value: 2462359892

-------------------------------------------------------------------------------
-
-------------------------------------------------------------------

| Id  | Operation                                          | Name
       | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-------------------------------------------------------------------

|   0 | SELECT STATEMENT                                   |
       |     1 |  1134 |    32   (0)| 00:00:01 |     |      |
```

```
|   1 |  SORT AGGREGATE                                      |
        |     1 |  1134 |        |       |      |      |

|*  2 |   FILTER                                             |
        |      |      |          |       |      |      |

|   3 |    NESTED LOOPS                                      |
        |   319 |   353K|    32   (0)| 00:00:01 |      |      |

|   4 |     PARTITION RANGE AND                              |
        |     1 |  1132 |     3   (0)| 00:00:01 |KEY(AP)|KEY(AP)|

|*  5 |      TABLE ACCESS BY LOCAL INDEX ROWID BATCHED WITH ZONEMAP|
OI_JSON_ORD
ER_ITEMS |     1 |  1132 |     3   (0)| 00:00:01 |KEY(AP)|KEY(AP)|

|*  6 |       INDEX RANGE SCAN                                 | I_CUST_ID
        |     1 |    |     1   (0)| 00:00:01 |KEY(AP)|KEY(AP)|

|   7 |     JSONTABLE EVALUATION                             |
        |      |      |          |       |      |      |

---------------------------------------------------------------------------
-
---------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('24-FEB-09 23:59:59','DD-MON-RR
HH24:MI:SS')>=TO_DATE('24-
FEB-09 00:00:00','DD-MON-RR HH24:MI:SS'))

   5 - filter(SYS_ZMAP_FILTER('/* ZM_PRUNING */ SELECT zm."ZONE_ID$", CASE WHEN
BITAND(zm."ZONE_STATE$",1)=1 THEN 1 ELSE CASE WHEN

            (zm."MAX_1_ORDER_DATE" < :1 OR zm."MIN_1_ORDER_DATE" > :2) THEN 3
ELSE 2 END END FROM "SOE"."ZMAP$_OI_JSON_ORDER_ITEMS" zm WHERE

            zm."ZONE_LEVEL$"=0 ORDER BY zm."ZONE_ID$"',SYS_OP_ZONE_ID(ROWID),T
O_DATE('24-FEB-09 00:00:00','DD-MON-RR HH24:MI:SS'),TO_DATE('24-FEB-09

            23:59:59','DD-MON-RR HH24:MI:SS'))<3 AND "OIJ"."ORDER_DATE">=TO_DA
TE('24-FEB-09 00:00:00','DD-MON-RR HH24:MI:SS') AND

            "OIJ"."ORDER_DATE"<=TO_DATE('24-FEB-09 23:59:59','DD-MON-RR HH24:M
I:SS'))

   6 - access(JSON_VALUE("OI_JSON" FORMAT JSON , '$.CUSTOMER_ID' RETURNING
NUMBE
R(12,0) ERROR ON ERROR NULL ON EMPTY)=733116)
```

```
Statistics
----------------------------------------------------------
       440  recursive calls
         5  db block gets
       428  consistent gets
        29  physical reads
       876  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
        35  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

We observe that partition pruning and indexing work together, and boost even more the performance.
Let's build a table to compare the different results:

Test case	Response time	Buffer Gets
Relational query	1,16 s	10
Query on JSON table	3,20 s	252515
JSON table + partition pruning	0,05 s	3660
JSON table + Index on json attribute customer_id	0,01 s	194
JSON table + Index on json attribute customer_id + partition pruning	0,01 s	428

We can speed up queries on the JSON table by using traditional tuning techniques, like indexing or partitioning. This demonstrates that all the performance feature of the Oracle database still apply when using JSON data.
In the following steps, we will go further with indexing, but now let's examine another way to partition a table, based on a JSON attribute.

Partition the table on a JSON attribute

Instead of partitioning the table on a relational scalar column, we could partition on on JSON attribute, and observe the same partition pruning mechanism.
Connect to soe schema and create a JSON table, using a JSON attribute as the partition key:

```
sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe


CREATE TABLE OI_JSON_ORDER_ITEMS_PART
(id NUMBER(12) NOT NULL PRIMARY KEY,
   OI_JSON VARCHAR2(4000),
```

```
   ORDER_DATE DATE GENERATED ALWAYS AS
     (json_value (OI_JSON, '$.ORDER_DATE' RETURNING DATE))
)
PARTITION BY RANGE (ORDER_DATE) INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))
  (
    PARTITION OIJSONPART_P0 VALUES LESS THAN (TO_DATE('2007-02-01', 'YYYY-MM-
DD'))
  );

desc OI_JSON_ORDER_ITEMS_PART
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
-
 ID                                        NOT NULL NUMBER(12)
 OI_JSON                                            VARCHAR2(4000)
 ORDER_DATE                                         DATE
```

We create the ORDER_DATE column as a virtual column on top of the ORDER_DATE attribute onto the JSON document, then we use that virtual column as the partition key, and interval partition the table with it. We observe, with a "desc" command, that the virtual appears as a regular column.

We will now populate this new table:

```
alter table OI_JSON_ORDER_ITEMS_PART no inmemory;

alter session enable parallel DML;
alter session force parallel query parallel 2;
set timing on

insert /*+ APPEND NOLOGGING */
into OI_JSON_ORDER_ITEMS_PART (id,OI_json)
select O.order_id,
       json_object (
       'ORDER_ID' value O.ORDER_ID,
       'ORDER_DATE' value O.ORDER_DATE,
       'ORDER_MODE' value O.ORDER_MODE,
       'CUSTOMER_ID' value O.CUSTOMER_ID,
       'ORDER_STATUS' value O.ORDER_STATUS,
       'ORDER_TOTAL' value O.ORDER_TOTAL,
       'SALES_REP_ID' value O.SALES_REP_ID,
       'PROMOTION_ID' value O.PROMOTION_ID,
       'WAREHOUSE_ID' value O.WAREHOUSE_ID,
       'DELIVERY_TYPE' value O.DELIVERY_TYPE,
       'COST_OF_DELIVERY' value O.COST_OF_DELIVERY,
    'WAIT_TILL_ALL_AVAILABLE' value O.WAIT_TILL_ALL_AVAILABLE,
    'DELIVERY_ADDRESS_ID' value O.DELIVERY_ADDRESS_ID,
    'CUSTOMER_CLASS' value O.CUSTOMER_CLASS,
    'CARD_ID' value O.CARD_ID,
    'INVOICE_ADDRESS_ID' value O.INVOICE_ADDRESS_ID,
    'ITEMS' value json_arrayagg (
        json_object (
         'ORDER_ID' value  OI.ORDER_ID,
```

```
          'LINE_ITEM_ID' value  OI.LINE_ITEM_ID,
          'PRODUCT_ID' value  OI.PRODUCT_ID,
          'UNIT_PRICE' value  OI.UNIT_PRICE,
          'QUANTITY' value  OI.QUANTITY,
          'DISPATCH_DATE' value  OI.DISPATCH_DATE,
          'RETURN_DATE' value  OI.RETURN_DATE,
          'GIFT_WRAP' value  OI.GIFT_WRAP,
          'CONDITION' value  OI.CONDITION,
          'SUPPLIER_ID' value  OI.SUPPLIER_ID,
          'ESTIMATED_DELIVERY' value  OI.ESTIMATED_DELIVERY
    )
    )
    ) as mijson
from orders O,
    order_items OI
where O.order_id = OI.order_id(+)
group by
O.ORDER_ID,
O.ORDER_DATE,
O.ORDER_MODE,
O.CUSTOMER_ID,
O.ORDER_STATUS,
O.ORDER_TOTAL,
O.SALES_REP_ID,
O.PROMOTION_ID,
O.WAREHOUSE_ID,
O.DELIVERY_TYPE,
O.COST_OF_DELIVERY,
O.WAIT_TILL_ALL_AVAILABLE,
O.DELIVERY_ADDRESS_ID,
O.CUSTOMER_CLASS,
O.CARD_ID,
O.INVOICE_ADDRESS_ID;


1429790 rows created.

Elapsed: 00:01:44.14
SQL> SQL> SQL> SQL> commit;

Commit complete.

Elapsed: 00:00:00.07
```

Now we will test the same query, to observe partition pruning:

```
-- Exit and reconnect to avoid parallel query execution !!!

exit
sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe
set timing on

select sum(ARR.UNIT_PRICE)
```

```
from OI_JSON_ORDER_ITEMS_PART OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OIJ.OI_JSON, '$.CUSTOMER_ID') = 733116
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

SUM(ARR.UNIT_PRICE)
-------------------
               5146

Elapsed: 00:00:04.70

set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS_PART OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OIJ.OI_JSON, '$.CUSTOMER_ID') = 733116
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

Elapsed: 00:00:04.68

Execution Plan
----------------------------------------------------------
Plan hash value: 939613720

-------------------------------------------------------------------------------
-
-----------------------------------

| Id  | Operation          | Name                | Rows  | Bytes | Cos
t (%CPU)| Time         | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-----------------------------------

|   0 | SELECT STATEMENT   |                     |    1 |  2006 |   4
92K  (3)| 00:00:20 |         |      |

|   1 |  SORT AGGREGATE    |                     |    1 |  2006 |
      |         |      |      |

|   2 |   NESTED LOOPS           |                     |  1239K|  2372M|   4
92K  (3)| 00:00:20 |         |      |
```

```
|   3 |    PARTITION RANGE ALL   |                       | 15180 | 28M| 697
81   (1)| 00:00:03 |       1 |1048575|

|*  4 |     TABLE ACCESS FULL    | OI_JSON_ORDER_ITEMS_PART | 15180 |   28M|
697
81   (1)| 00:00:03 |       1 |1048575|

|*  5 |    JSONTABLE EVALUATION |                       |     |      |
       |         |     |      |

------------------------------------------------------------------------------
-
------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter(TO_NUMBER(JSON_VALUE("OIJ"."OI_JSON" FORMAT JSON ,
'$.CUSTOMER_ID'
 RETURNING VARCHAR2(4000)

             NULL ON ERROR))=733116)
   5 - filter(TRUNC("P"."ORDER_DATE")=TO_DATE('24-FEB-09','DD-MON-RR'))

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)


Statistics
----------------------------------------------------------
         0  recursive calls
         0  db block gets
    253590  consistent gets
    249728  physical reads
         0  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

Partition pruning does not kick in, because we are not using the virtual column in the predicate. Change the query to use the virtual column:

```
--- WE should use ORDER_DATE virtual column to leverage partition pruning !!!

select sum(ARR.UNIT_PRICE)
```

```
from OI_JSON_ORDER_ITEMS_PART OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OIJ.OI_JSON, '$.CUSTOMER_ID') = 733116
and OIJ.ORDER_DATE between to_date('24-FEB-09 00:00:00','DD-MON-RR HH24:MI:SS')
and to_date('24-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS');

Elapsed: 00:00:00.06

Execution Plan
----------------------------------------------------------
Plan hash value: 3037461420

-------------------------------------------------------------------------------
-
---------------------------------------

| Id  | Operation             | Name                 | Rows  | Bytes |
 Cost (%CPU)| Time     | Pstart| Pstop |

-------------------------------------------------------------------------------
-
---------------------------------------

|   0 | SELECT STATEMENT      |                      |     1 |  2013 |
  3935 (54)| 00:00:01 |       |       |

|   1 |  SORT AGGREGATE       |                      |     1 |  2013 |
           |          |       |       |

|*  2 |   FILTER              |                      |       |       |
           |          |       |       |

|   3 |    NESTED LOOPS       |                      |   250K|   480M|
  3935 (54)| 00:00:01 |       |       |

|   4 |     PARTITION RANGE ITERATOR|                      |    31 | 62341 |
  3091 (68)| 00:00:01 |   KEY |   KEY |

|*  5 |      TABLE ACCESS FULL       | OI_JSON_ORDER_ITEMS_PART |    31 | 62341
|
  3091 (68)| 00:00:01 |   KEY |   KEY |

|   6 |     JSONTABLE EVALUATION    |                      |       |       |
           |          |       |       |

-------------------------------------------------------------------------------
-
---------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('24-FEB-09 23:59:59','DD-MON-RR
HH24:MI:SS')>=TO_DATE('24-
FEB-09 00:00:00','DD-MON-RR

             HH24:MI:SS'))
   5 - filter(TO_NUMBER(JSON_VALUE("OIJ"."OI_JSON" FORMAT JSON ,
'$.CUSTOMER_ID'
 RETURNING VARCHAR2(4000) NULL

             ON ERROR))=733116 AND "OIJ"."ORDER_DATE">=TO_DATE('24-FEB-09 00:00
:00','DD-MON-RR HH24:MI:SS') AND

             "OIJ"."ORDER_DATE"<=TO_DATE('24-FEB-09 23:59:59','DD-MON-RR HH24:M
I:SS'))


Note
-----
   - dynamic statistics used: dynamic sampling (level=2)


Statistics
----------------------------------------------------------
        18  recursive calls
         0  db block gets
      3663  consistent gets
         0  physical reads
         0  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

This illustrates that partitioning might be implemented using a JSON attribute as the partition key.

Create a SEARCH index

Indexing customer_id attribute of the JSON column had a great performance impact on the previous query. But what if we don't use customer_id anymore in the predicate ? What if we cannot anticipate which of the attributes will be used in the predicate ? This is a common case, that can be addressed with a powerful SEARCH index.

```
## Instead of customer_id, we will use card_id in the predicate
```

```
select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OI_JSON, '$.CARD_ID'  returning NUMBER(12) error on error
null on empty) = 1465982
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

SUM(ARR.UNIT_PRICE)
-------------------
              5146

Elapsed: 00:00:07.22

set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OI_JSON, '$.CARD_ID'  returning NUMBER(12) error on error
null on empty) = 1465982
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

Elapsed: 00:00:06.28

Execution Plan
----------------------------------------------------------
Plan hash value: 1435571113

-------------------------------------------------------------------------------
-
-------------------------------

| Id  | Operation          | Name             | Rows  | Bytes | Cost (%C
PU)| Time     | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-------------------------------

|   0 | SELECT STATEMENT   |                  |     1 |  1124 |   468K
(3)| 00:00:19 |       |       |

|   1 |  SORT AGGREGATE    |                  |     1 |  1124 |
```

```
|   |        |       |       |

|   2 |   NESTED LOOPS           |                  |  1167K|  1251M|   468K
(3)| 00:00:19 |       |       |

|   3 |    PARTITION RANGE ALL   |                  | 14298 |    15M| 69598
(1)| 00:00:03 |     1 |1048575|

|*  4 |     TABLE ACCESS FULL    | OI_JSON_ORDER_ITEMS | 14298 |    15M| 69598
(1)| 00:00:03 |     1 |1048575|

|*  5 |    JSONTABLE EVALUATION |                  |       |       |
   |         |       |       |

--------------------------------------------------------------------------------
-
-------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   4 - filter(JSON_VALUE("OI_JSON" FORMAT JSON , '$.CARD_ID' RETURNING
NUMBER(12
,0) ERROR ON ERROR

            NULL ON EMPTY)=1465982)
   5 - filter(TRUNC("P"."ORDER_DATE")=TO_DATE('24-FEB-09','DD-MON-RR'))


Statistics
----------------------------------------------------------
        66  recursive calls
         0  db block gets
    252357  consistent gets
    252160  physical reads
         0  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

As card_id attribute is not indexed, we are back to the full table scan execution plan. Let's build a search index on the JSON column (allow something like 10 minutes to complete):

```
create search index I_JSON_SEARCH on OI_JSON_ORDER_ITEMS(OI_JSON) for JSON;

Index created.

Elapsed: 00:10:09.64
```

Now repeat the query and observe the performance boost:

```
set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where json_value (OI_JSON, '$.CARD_ID') = 1465982
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');

Elapsed: 00:00:00.03

Execution Plan
----------------------------------------------------------
Plan hash value: 1938435827

-------------------------------------------------------------------------------
-
-----------------------------------------

| Id  | Operation                     | Name            | Rows  | Byt
es | Cost (%CPU)| Time       | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-----------------------------------------

|   0 | SELECT STATEMENT              |                 |    1 |  11
36 |   521   (1)| 00:00:01 |       |       |

|   1 |  SORT AGGREGATE               |                 |    1 |  11
36 |         |          |       |       |

|   2 |   NESTED LOOPS                |                 |   584 |   6
47K|   521   (1)| 00:00:01 |       |       |

|*  3 |    TABLE ACCESS BY GLOBAL INDEX ROWID| OI_JSON_ORDER_ITEMS |     7 |
79
24 |   325   (0)| 00:00:01 | ROWID | ROWID |

|*  4 |     DOMAIN INDEX              | I_JSON_SEARCH   |       |
   |    4   (0)| 00:00:01 |       |       |

|*  5 |    JSONTABLE EVALUATION       |                 |       |
   |         |          |       |       |
```

```
--------------------------------------------------------------------------------
-
-------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(TO_NUMBER(JSON_VALUE("OI_JSON" FORMAT JSON , '$.CARD_ID'
RETURNING
 VARCHAR2(4000) NULL ON

             ERROR))=1465982)
   4 -
access("CTXSYS"."CONTAINS"("OIJ"."OI_JSON",'(sdata(FNUM_14173D25B4DD102AB
9B6F2851AEE2420_CARD_ID  = 1465982

             ))')>0)
   5 - filter(TRUNC("P"."ORDER_DATE")=TO_DATE('24-FEB-09','DD-MON-RR'))


Statistics
----------------------------------------------------------
       216  recursive calls
         5  db block gets
       176  consistent gets
         0  physical reads
      1112  redo size
       580  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         2  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

Observe that the SEARCH index (DOMAIN INDEX) is being used, and that the performance has been significantly improved.

Note that we could also use the dotted notation to write the query:

```
select sum(ARR.UNIT_PRICE)
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path
'$.UNIT_PRICE'))
                ) as ARR
where OIJ.OI_JSON.CARD_ID = 1465982
and trunc(ARR.ORDER_DATE) = to_date('24-FEB-09','DD-MON-RR');
```

JSON and analytical queries

We can run analytical queries on top of a JSON column: connect to soe schema and run the following query:

```
sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe
select sum(ARR.UNIT_PRICE*ARR.QUANTITY) as "GrantTotal"
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path '$.UNIT_PRICE',
                                      QUANTITY NUMBER path '$.QUANTITY'))
                ) as ARR
where OIJ.ORDER_DATE between to_date('01-FEB-09 00:00:00','DD-MON-RR
HH24:MI:SS') and to_date('28-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS');

GrantTotal
----------
 278913547

Elapsed: 00:00:00.40

set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE*ARR.QUANTITY) as "GrantTotal"
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
                             COLUMNS (UNIT_PRICE NUMBER path '$.UNIT_PRICE',
                                      QUANTITY NUMBER path '$.QUANTITY'))
                ) as ARR
where OIJ.ORDER_DATE between to_date('01-FEB-09 00:00:00','DD-MON-RR
HH24:MI:SS') and to_date('28-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS');

Elapsed: 00:00:00.38

Execution Plan
----------------------------------------------------------
Plan hash value: 2564720920

-------------------------------------------------------------------------------
-
----------------------------------------

| Id  | Operation                    | Name            | Rows  | Bytes
 | Cost (%CPU)| Time       | Pstart| Pstop |

-------------------------------------------------------------------------------
-
----------------------------------------
```

```
|   0 | SELECT STATEMENT            |                 |     1 |  1132
 |   630K  (1)| 00:00:25 | |      |

|   1 |  SORT AGGREGATE             |                 |     1 |  1132
 |           |       |      |      |

|*  2 |   FILTER                    |                 |       |
 |           |       |      |      |

|   3 |    NESTED LOOPS             |                 |   168M|   177
G|   630K  (1)| 00:00:25 | |      |

|   4 |     PARTITION RANGE AND     |                 | 20624 |    22
M| 69608   (1)| 00:00:03 |KEY(AP)|KEY(AP)|

|*  5 |      TABLE ACCESS FULL WITH ZONEMAP| OI_JSON_ORDER_ITEMS | 20624 |
22
M| 69608   (1)| 00:00:03 |KEY(AP)|KEY(AP)|

|   6 |     JSONTABLE EVALUATION    |                 |       |
 |           |       |      |      |

--------------------------------------------------------------------------------
-
----------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('28-FEB-09 23:59:59','DD-MON-RR
HH24:MI:SS')>=TO_DATE('01-
FEB-09 00:00:00','DD-MON-RR

            HH24:MI:SS'))
   5 - filter(SYS_ZMAP_FILTER('/* ZM_PRUNING */ SELECT zm."ZONE_ID$", CASE WHEN
BITAND(zm."ZONE_STATE$",1)=1 THEN

            1 ELSE CASE WHEN (zm."MAX_1_ORDER_DATE" < :1 OR zm."MIN_1_ORDER_DA
TE" > :2) THEN 3 ELSE 2 END END FROM

            "SOE"."ZMAP$_OI_JSON_ORDER_ITEMS" zm WHERE zm."ZONE_LEVEL$"=0 ORDE
R BY

            zm."ZONE_ID$"',SYS_OP_ZONE_ID(ROWID),TO_DATE('01-FEB-09 00:00:00',
'DD-MON-RR HH24:MI:SS'),TO_DATE('28-FEB-09

            23:59:59','DD-MON-RR HH24:MI:SS'))<3 AND "OIJ"."ORDER_DATE"<=TO_DA
TE('28-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS')

            AND "OIJ"."ORDER_DATE">=TO_DATE('01-FEB-09 00:00:00','DD-MON-RR HH
24:MI:SS'))
```

```
Statistics
----------------------------------------------------------
         17  recursive calls
          0  db block gets
       3662  consistent gets
       3646  physical reads
          0  redo size
        574  bytes sent via SQL*Net to client
         52  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

We observe that the query is resolved by partition pruning, with a fast response time. We can use In-memory database with a JSON table, to leverage in-memory analytical queries. In the following steps, we will populate the in-memory column store with a partition, and observe the result:

Populate In-memory column store

```
--- Use the following query to get the name of the partitions for 2009-02:

select PARTITION_NAME, HIGH_VALUE from user_tab_partitions where table_name =
'OI_JSON_ORDER_ITEMS';

SYS_P2582
TO_DATE(' 2009-03-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS',
'NLS_CALENDAR=GREGORIA'

-- Place that partition into the IMC: replace the partition name by your
partition name !!!

alter table OI_JSON_ORDER_ITEMS modify partition SYS_P2582 inmemory priority
critical;

Table altered.

Elapsed: 00:00:00.01

-- Re-run the analytical query and observe the execution plan

set autotrace traceonly explain statistics

select sum(ARR.UNIT_PRICE*ARR.QUANTITY) as "GrantTotal"
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                             NESTED PATH '$.ITEMS[*]'
```

```
                                COLUMNS (UNIT_PRICE NUMBER path '$.UNIT_PRICE',
                                         QUANTITY NUMBER path '$.QUANTITY'))
                ) as ARR
where OIJ.ORDER_DATE between to_date('01-FEB-09 00:00:00','DD-MON-RR
HH24:MI:SS') and to_date('28-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS');

Elapsed: 00:00:00.40

Execution Plan
----------------------------------------------------------
Plan hash value: 2564720920

-------------------------------------------------------------------------------
-
-------------------------------------------------

| Id  | Operation                             | Name            | Rows
  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-------------------------------------------------

|   0 | SELECT STATEMENT                      |                 |
1 |  1132 |   630K  (1)| 00:00:25 |       |     |

|   1 |  SORT AGGREGATE                       |                 |
1 |  1132 |         |       |     |     |

|*  2 |   FILTER                              |                 |
  |     |          |       |     |     |

|   3 |    NESTED LOOPS                       |                 |   16
8M|   177G|   630K  (1)| 00:00:25 |       |     |

|   4 |     PARTITION RANGE AND               |                 | 2062
4 |    22M| 69608   (1)| 00:00:03 |KEY(AP)|KEY(AP)|

|*  5 |      TABLE ACCESS INMEMORY FULL WITH ZONEMAP| OI_JSON_ORDER_ITEMS |
2062
4 |    22M| 69608   (1)| 00:00:03 |KEY(AP)|KEY(AP)|

|   6 |     JSONTABLE EVALUATION              |                 |
  |     |          |       |     |     |

-------------------------------------------------------------------------------
-
-------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   2 - filter(TO_DATE('28-FEB-09 23:59:59','DD-MON-RR
HH24:MI:SS')>=TO_DATE('01-
FEB-09 00:00:00','DD-MON-RR HH24:MI:SS'))

   5 - inmemory("OIJ"."ORDER_DATE"<=TO_DATE('28-FEB-09 23:59:59','DD-MON-RR
HH24
:MI:SS') AND

             "OIJ"."ORDER_DATE">=TO_DATE('01-FEB-09 00:00:00','DD-MON-RR HH24:M
I:SS'))

       filter(SYS_ZMAP_FILTER('/* ZM_PRUNING */ SELECT zm."ZONE_ID$", CASE WHEN
BITAND(zm."ZONE_STATE$",1)=1 THEN 1 ELSE

             CASE WHEN (zm."MAX_1_ORDER_DATE" < :1 OR zm."MIN_1_ORDER_DATE" > :
2) THEN 3 ELSE 2 END END FROM

             "SOE"."ZMAP$_OI_JSON_ORDER_ITEMS" zm WHERE zm."ZONE_LEVEL$"=0 ORDE
R BY

             zm."ZONE_ID$"',SYS_OP_ZONE_ID(ROWID),TO_DATE('01-FEB-09 00:00:00',
'DD-MON-RR HH24:MI:SS'),TO_DATE('28-FEB-09

             23:59:59','DD-MON-RR HH24:MI:SS'))<3 AND "OIJ"."ORDER_DATE"<=TO_DA
TE('28-FEB-09 23:59:59','DD-MON-RR HH24:MI:SS') AND

             "OIJ"."ORDER_DATE">=TO_DATE('01-FEB-09 00:00:00','DD-MON-RR HH24:M
I:SS'))


Statistics
----------------------------------------------------------
        83  recursive calls
         0  db block gets
      3726  consistent gets
      3646  physical reads
         0  redo size
       574  bytes sent via SQL*Net to client
        52  bytes received via SQL*Net from client
         2  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
         1  rows processed
```

JSON and materialized views

We can create materialized views on top of JSON documents. This will dramatically speed-up the analytical queries. Connect to soe schema, and create a materialized view:

```
sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe

create materialized view fast_mv
build immediate
refresh complete on demand
as
select to_char(OIJ.ORDER_DATE,'YYYYMM') as "Month",
sum(ARR.UNIT_PRICE*ARR.QUANTITY) as "GrantTotal"
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                                NESTED PATH '$.ITEMS[*]'
                                COLUMNS (UNIT_PRICE NUMBER path '$.UNIT_PRICE',
                                         QUANTITY NUMBER path '$.QUANTITY'))
                ) as ARR
group by to_char(OIJ.ORDER_DATE,'YYYYMM');

-- Now compare the performance and scalability metrics if you use either the
base table or the materialized view:

set autotrace traceonly explain statistics
set timing on

select to_char(OIJ.ORDER_DATE,'YYYYMM') as "Month",
sum(ARR.UNIT_PRICE*ARR.QUANTITY) as "GrantTotal"
from OI_JSON_ORDER_ITEMS OIJ,
     json_table(OIJ.OI_JSON,
                '$' COLUMNS (ORDER_DATE DATE path '$.ORDER_DATE',
                                NESTED PATH '$.ITEMS[*]'
                                COLUMNS (UNIT_PRICE NUMBER path '$.UNIT_PRICE',
                                         QUANTITY NUMBER path '$.QUANTITY'))
                ) as ARR
group by to_char(OIJ.ORDER_DATE,'YYYYMM');

64 rows selected.

Elapsed: 00:00:38.50

Execution Plan
----------------------------------------------------------
Plan hash value: 2840653318

-------------------------------------------------------------------------------
-
-------------------------------------------

| Id  | Operation               | Name           | Rows  | Bytes |Tem
pSpc| Cost (%CPU)| Time     | Pstart| Pstop |

-------------------------------------------------------------------------------
-
-------------------------------------------
```

```
|   0 | SELECT STATEMENT        |                 | 46640 | 50M|
    |  1987M  (1)| 21:34:11 |        |      |

|   1 |  HASH GROUP BY                |                 | 46640 | 50M|
 12T|  1987M  (1)| 21:34:11 |        |      |

|   2 |   NESTED LOOPS                |                 |   11G|   12T|
    |   38M  (1)| 00:25:23 |         |      |

|   3 |    PARTITION RANGE ALL         |                 |  1429K|  1538M|
    | 69555   (1)| 00:00:03 |      1 |1048575|

|   4 |     TABLE ACCESS INMEMORY FULL| OI_JSON_ORDER_ITEMS |  1429K|  1538M|
    | 69555   (1)| 00:00:03 |      1 |1048575|

|   5 |    JSONTABLE EVALUATION       |                 |      |      |
    |         |         |      |      |

---------------------------------------------------------------------------
-
---------------------------------------------


Statistics
----------------------------------------------------------
        57  recursive calls
         5  db block gets
    252342  consistent gets
    252160  physical reads
      1032  redo size
      2519  bytes sent via SQL*Net to client
        96  bytes received via SQL*Net from client
         6  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
        64  rows processed
```

This analytical query runs in 38,5 s against the base table. Let's rewrite the query and access to the materialized view:

```
set autotrace traceonly explain statistics
set timing on
select * from fast_mv;

64 rows selected.

Elapsed: 00:00:00.01

Execution Plan
----------------------------------------------------------
Plan hash value: 140868995
```

```
-------------------------------------------------------------------------------
-
---------

| Id  | Operation               | Name | Rows | Bytes | Cost (%CPU)| T
ime    |

-------------------------------------------------------------------------------
-
---------

|   0 | SELECT STATEMENT        |      |    64 |   896 |     3   (0)| 0
0:00:01 |

|   1 |  MAT_VIEW ACCESS INMEMORY FULL| FAST_MV |    64 |   896 |     3   (0)|
0
0:00:01 |

-------------------------------------------------------------------------------
-
---------



Statistics
----------------------------------------------------------
         1  recursive calls
         0  db block gets
         7  consistent gets
         0  physical reads
         0  redo size
      2519  bytes sent via SQL*Net to client
        96  bytes received via SQL*Net from client
         6  SQL*Net roundtrips to/from client
         0  sorts (memory)
         0  sorts (disk)
        64  rows processed
```

This concludes the JSON queries part of the lab. In the next chapter, we will review ingestion functionalities.

JSON documents ingestion

Use Fast Ingest tables for JSON documents

JSON documents are widely used to create a "schema on read" data model. Many IOT devices use JSON format to send the metrics they are collecting. Ingesting these documents in real time can be challenging, depending on the number of IOT sending information concurrently.

In the next steps, we will review a fast ingestion mechanism that can be used for this purpose: Oracle 19c MEMOPTIMIZE FOR WRITE tables. We will compare unitary inserts into a regular table and into a MEMOPTIMIZE FOR WRITE table.

Connect to soe schema, and create a regular JSON table:

```
sqlplus soe/soe@rdbms21coniaas:1521/pdbsoe

create table OI_JSON_REGULAR
(
ID number(12),
OI_json VARCHAR2(4000),
CONSTRAINT oi_json_regular_pk primary Key (id),
CONSTRAINT OI_json_regular_check CHECK (OI_json IS JSON)
);

-- Create a MEMOPTIMIZE FOR WRITE table with JSON column !!!

create table OI_JSON_MEMOPT4WRITE
(
    ID number(12),
    OI_JSON varchar2(4000),
    CONSTRAINT oi_json_MEMOPT_pk primary Key (id),
    CONSTRAINT OI_json_MEMOPT_check CHECK (OI_json IS JSON)
) segment creation immediate memoptimize for write;

-- Now we create a PL/SQL block that inserts row by row into the regular table:

create or replace procedure PC_INS_REGULAR (p_num_rows IN PLS_INTEGER)
IS
    CURSOR c_oi (p_num IN PLS_INTEGER)
    IS
        select id, OI_json
        from OI_JSON_ORDER_ITEMS
        where rownum <= p_num;
begin
    FOR cur in c_oi (p_num_rows)
    LOOP
        insert into OI_JSON_REGULAR (id,oi_json) values (cur.id,cur.oi_json);
        commit;
    END LOOP;
END;
/

-- Now we create a PL/SQL block that inserts row by row into the MEMOPTIMIZE
FOR WRITE table:

create or replace procedure PC_INS_MEMOPT4WRITE (p_num_rows IN PLS_INTEGER)
IS
    CURSOR c_oi (p_num IN PLS_INTEGER)
    IS
        select id, OI_json
        from OI_JSON_ORDER_ITEMS
```

```
        where rownum <= p_num;
begin
    FOR cur in c_oi (p_num_rows)
    LOOP
        insert /*+ memoptimize_write */ into OI_JSON_MEMOPT4WRITE (id,oi_json)
values (cur.id,cur.oi_json);
        commit;
    END LOOP;
END;
/
```

Pay attention to the syntax details that are specific to memoptimize for write tables (highlighted in green).

Now we will use the created procedures to insert some rows in both the regular and the memoptimize for write tables. We will then compare the results in terms of performance and throughput: let's start with 1000 rows.

```
set timing on

--- 1000 rows !!!
truncate table OI_JSON_REGULAR;
truncate table OI_JSON_MEMOPT4WRITE;

exec PC_INS_REGULAR(1000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.23
SQL> exec PC_INS_MEMOPT4WRITE(1000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:18.12
```

The first time we use the memoptimize for write table, a memory alocation is done in the large pool. This explains why the first execution is so slow, but this will occur only once. Let's re-run the second test:

```
truncate table OI_JSON_MEMOPT4WRITE;

Table truncated.

Elapsed: 00:00:00.07
SQL> exec PC_INS_MEMOPT4WRITE(1000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.04
```

Even with only 1000 rows inserted, memoptimize for write table is way faster. Let's try with each time more rows and compare:

```
-- 10.000 rows !!!
truncate table OI_JSON_REGULAR;
truncate table OI_JSON_MEMOPT4WRITE;


SQL> exec PC_INS_REGULAR(10000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:05.30
SQL> exec PC_INS_MEMOPT4WRITE(10000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.70

--- 100.000 rows !!!

truncate table OI_JSON_REGULAR;
truncate table OI_JSON_MEMOPT4WRITE;

SQL> exec PC_INS_REGULAR(100000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:24.83
SQL> exec PC_INS_MEMOPT4WRITE(100000)

PL/SQL procedure successfully completed.

Elapsed: 00:00:08.00

-- Count the rows in each table and check:

select count(*) from OI_JSON_REGULAR;

  COUNT(*)
----------
    100000

select count(*) from OI_JSON_MEMOPT4WRITE;

  COUNT(*)
----------
     99390
```

The count in the memoptimize for write table doesn't match the number of rows inserted (100.000). This is because rows are committed asynchronously in the MEMOPTIMIZE FOR WRITE table.
We might want to use DBMS_MEMOPTIMIZE.WRITE_END procedure to force an immediate flush of the large pool to the table, or just wait for the rows to be eventually flushed automatically. This is important to understand, and might be suitable for your IOT business case (or not).
After some seconds, the missing rows are flushed and we can see them in the table:

```
select count(*) from OI_JSON_MEMOPT4WRITE;

  COUNT(*)
----------
    100000


--- 1.000.000 rows !!!

SQL> exec PC_INS_REGULAR(1000000)

PL/SQL procedure successfully completed.

Elapsed: 00:04:28.81

SQL> select count(*) from OI_JSON_REGULAR;

  COUNT(*)
----------
   1000000


SQL> exec PC_INS_MEMOPT4WRITE(1000000)

PL/SQL procedure successfully completed.

Elapsed: 00:01:29.13

SQL> select count(*) from OI_JSON_MEMOPT4WRITE;

  COUNT(*)
----------
    999875

Elapsed: 00:00:00.15
SQL> exec DBMS_MEMOPTIMIZE.WRITE_END

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.04
SQL> select count(*) from OI_JSON_MEMOPT4WRITE;
```

```
  COUNT(*)
----------
   1000000

Elapsed: 00:00:00.03
```