JSON Performance features in Oracle 12c Release 2

ORACLE WHITE PAPER | MARCH 2017

Disclaimer

Table of Contents

Introduction

JSON (JavaScript Object Notation) is a lightweight data interchange format. JSON is text based which makes it easy for humans to read and write and for machine to read and parse and generate. JSON being text based makes it completely language independent. JSON has gained a wide popularity among application developers (specifically web application developers) and is used as a persistent format for application data. JSON is schema-less, which makes it particularly attractive to developers, enabling the applications to make changes without requiring corresponding changes to the storage schema. Detailed JSON specification can be found at http://json.org.

Oracle 12c Release 1 provides flexibility of NoSQL data store along with other traditional relational data with the power of SQL based analytics and reporting. This release also introduces an extension to SQL which enables storing, indexing and querying of JSON data. It also provides APIs which offer application developers a NoSQL development experience. These enhancements along with traditional SQL reporting and analytics makes Oracle 12c an ideal platform to store JSON content.

This whitepaper covers performance enhancements in the latest release, Oracle 12c Release 2, for efficient querying of JSON content. A brief introduction on JSON querying in Oracle 12c is given in the next section as a refresher. A separate whitepaper on "Oracle as a document store" covers storing, loading and querying JSON content in greater detail. Benchmark results are presented to quantify the gains seen by the performance features. In this paper Oracle 12c is used to refer to both Oracle 12c Releases 1 and 2 unless otherwise indicated.

Brief Introduction to JSON and Querying JSON in Oracle 12c

JSON is a light weight data interchange format (http://www.json.org). JSON consists of a collection of key/value pairs and an ordered list of values (similar to array, vector, list etc.). A sample JSON document in text form is shown below.

```
{
  "PONumber"               : 1438,
  "Reference"              : "SKVIS-20140421",
  "Requestor"              : "Skar Viswa",
  "User"                   : "SKVIS",
  "CostCenter"             : "A50",
  "ShippingInstructions" : { "name"    : "Skar Viswa",
                             "Address": {
                                 "street"   : "200 Sporting Green",
                                 "city"     : "South San Francisco",
                                 "state"    : "CA",
                                 "zipCode" : 99236,
                                 "country" : "United States of America"
                              },
                                 "Phone" : [ { "type": "Office","number": "909-555-7307"
},
                                             { "type": "Mobile","number": "415-555-1234" }
]
                           },
   "Special Instructions" : null,
   "AllowPartialShipment" : false,
   "LineItems"             : [ { "ItemNumber" : 1,
                                "Part"        : { "Description" : "Mission Impossible",
                                                  "UnitPrice"    : 19.95,
                                                  "UPCCode"      : 13131092705
                               },
                               "Quantity"    : 9.0
                             }, {
                                  "ItemNumber" : 2,
                                  "Part"        : { "Description" : "Lethal Weapon",
                                                    "UnitPrice"    : 21.95,
                                                    "UPCCode"      : 85391628927 },
                                  "Quantity": 5.0
                                 }]
```

JSON documents can contain one or more key-value pairs. Values can be scalars, arrays and/or objects. Scalar values can be strings, numbers, Booleans or nulls. There are no date, time or other scalar data types. An object consists of one or more key-value pairs. Arrays are ordered collections of values. The elements of an array can be of different types.

Storing JSON

In Oracle Database 12c JSON is stored in standard columns of data type CLOB, BLOB, standard VARCHAR2(4K) and extended VARCHAR2(32K). The standard data type being used for JSON means that all enterprise-level functionality like replication and high-availability are automatically available for JSON data as well. To enable storing JSON, a constraint called 'IS JSON' is introduced, which tells the Oracle database that the column contains a valid JSON document. All the enterprise-level security features are automatically available on JSON data. This whitepaper will not cover the details of loading JSON data into relational table.

JSON Path Expressions

JSON Path expressions are expressions used to navigate a JSON document. JSON Path expression is analogous to XPath for XML. JSON Path expressions contain the set of keys that need to be navigated in order to reach a particular item. A JSON Path can be used to reference a value of a particular key, an object, an array, or an entire document. In Oracle, an entire JSON document is referenced using the $ symbol. All JSON Path expressions start with the $ symbol. The following table demonstrates some JSON Path expressions based on the PurchaseOrder example given in the earlier section.

JSON Path Expression	Result	Remarks
$.PONumber	1438	Scalar; Number value
$.Requestor	Skar Viswa	Scalar; String value
$.ShippingInstructions.Phone	`[ {"type": "Office","number": "909-555-7307" }, { "type": "Mobile","number": "415-555-1234" } ]`	Array value
$.ShippingInstructions.Address	`{ "street": "200 Sporting Green", "city" :"South San Francisco", "state":"CA", "zipCode":99236, "country":"United States of America" }`	Object

Querying JSON

Oracle Database 12c Release 2 has support for JSON queries using SQL. The SQL/JSON enhancements allow SQL queries on JSON data. The enhanced operators JSON_VALUE, JSON_EXISTS, JSON_QUERY, JSON_TABLE and JSON_TEXTCONTAINS allow JSON path expressions to be evaluated on columns containing JSON data. These operators let JSON data to be queried just like relational data. The following examples showcase some of the JSON operators.

JSON_VALUE operator enables a JSON Path expression to be used to return a scalar value based on the JSON path of a key. JSON_VALUE operator can be used in the select list or the predicate list as a filter in the WHERE clause. The following example demonstrates the use of JSON_VALUE.

```
select JSON_VALUE(PO_DOCUMENT ,'$.LineItems[0].Part.UnitPrice' returning
NUMBER(5,3)) UNIT_PRICE
from J_PURCHASEORDER p
where JSON_VALUE(PO_DOCUMENT ,'$.PONumber' returning NUMBER(10)) = 1438
/

UNIT_PRICE
----------------
19.95
```

JSON_EXISTS operator checks whether a key-value pair exists in the JSON document for the specified JSON Path expression. It returns a true or false based on the presence of the key-value pair. The following example demonstrates the use of JSON_EXISTS. The query assumes that J_PURCHASEORDER table has a column PO_DOCUMENT containing JSON data.

```
 select count(*) from J_PURCHASEORDER
 where JSON_EXISTS(PO_DOCUMENT ,'$.ShippingInstructions.Address.state')
/
 COUNT(*)
 --------
      435
```

JSON_TABLE a very useful operator that facilitates relational access to JSON data. JSON_TABLE creates an inline view of the JSON data. JSON_TABLE operator contains one or more columns specified via JSON path expressions. The following example demonstrates the use of JSON_TABLE.

```
select M.*
  from J_PURCHASEORDER p,
       JSON_TABLE(
       p.PO_DOCUMENT ,
       '$'
       columns
           PO_NUMBER NUMBER(10) path '$.PONumber',
           REFERENCE VARCHAR2(30 CHAR) path '$.Reference',
           REQUESTOR VARCHAR2(32 CHAR) path '$.Requestor',
           USERID VARCHAR2(10 CHAR) path '$.User',
           COSTCENTER VARCHAR2(16 CHAR) path '$.CostCenter',
           TELEPHONE VARCHAR2(16 CHAR) path '$.ShippingInstructions.Phone[0].number'
        ) M
  where PO_NUMBER > 1437 and PO_NUMBER < 1440
/

PO_NUMBER REFERENCE        REQUESTOR    USERID COSTCENTER TELEPHONE
---------- --------------- ------------ ------ ---------- ----------
      1438 SKVIS-20140421    Skar Viswa  SKVIS        A50 909-555-7307
      1440 SKVIS-20140422    Skar Viswa  SKVIS        A50 909-555-9119
```

JSON_TABLE can also be used to generate an inline view from nested objects in JSON.

Indexing JSON

Oracle Database 12c supports indexing on JSON documents. Functional indexes can be created on specific keys or combination of keys. A search (full-text) index can also be created on the entire JSON documents. These indexes are used to optimize query operations that use SQL/JSON operators. Functional indexes are built using JSON_VALUE operators. Functional indexes on key values support both bitmap and B-Tree index format.

The following example demonstrates creation of a functional index on the PONumber key in the PurchaseOrder document.

```
create unique index PO_NUMBER_IDX
  on J_PURCHASEORDER (
       JSON_VALUE(
              PO_DOCUMENT, '$.PONumber' returning number(10) error on error
       )
  )
/

Index created.
```

Oracle Database 12c supports indexing the entire JSON document using a search index which is based on Oracle Full-Text index. The search index incorporates not only the values but the key names as well and also allows Full text searches over the JSON documents. The syntax below shows creating the index in Oracle Database 12c Release 1. In Release 2, a new simpler syntax has been introduced to create the search index which supersedes the older syntax.

```
exec CTX_DDL.SET_SEC_GRP_ATTR('json_group','json_enable','t');
exec ctx_ddl.drop_preference('live_st');
exec ctx_ddl.create_preference('live_st', 'BASIC_STORAGE');

create index PO_DOCUMENT_INDEX on J_PURCHASE_ORDER(PO_DOCUMENT) indextype is
ctxsys.context parameters('sync (on commit) section group json_group storage live_st
memory 1G') parallel 24;
```

The sync (on commit) enables the index to be updated every time insert and update operations are committed to the JSON documents. For documents with very frequent updates this will have a performance impact. This option is more suitable for read-mostly documents. The explain plan for a query will show if the context index is being utilized for executing the query.

The optimizer is aware of the functional index and search index and uses the appropriate index with a lesser cost, even though either could be used to satisfy the query.

NoBench Benchmark

NoBench benchmark was used to evaluate the performance of the enhanced JSON features. NoBench is a benchmark suite that evaluates the performance of several classes of queries over JSON data in NoSQL and SQL databases. Details of Nobench benchmark can be found at http://paperhub.s3.amazonaws.com/cb2e514d67256700c7eaeec9ac36d174.pdf.

Benchmark Details

The Nobench benchmark was configured with 16 million documents. Each document is of approximately 600 bytes. A typical benchmark document is shown below:

```
{
     "_id" : ObjectId("53214c880dca4a8c46f0dd1f"),
    "num" : 28483498,
    "bool" : false,
    "nested_obj" : {
                    "num" : 60483498,
                    "str" : "GBRDCMJRGAYDCMJQGEYDCMJRGAYDCMJRGEYDCMBRGAYTA==="
      },
      "dyn2" : "GBRDCMJQGEYTAMBRGAYTAMBRGEYTCMJRGAYTAMJQGEYA====",
      "dyn1" : 28483498,
      "nested_arr" : [ "take", "walked" ],
      "str2" : "GBRDCMJRGAYDCMJQGEYDCMJRGAYDCMJRGEYDCMBRGAYTA===",
      "str1" : "GBRDCMJQGEYTAMBRGAYTAMBRGEYTCMJRGAYTAMJQGEYA====",
      "thousandth" : 498,
      "sparse_980" : "GBRDCMBRGA======",
      "sparse_981" : "GBRDCMBRGA======",
      "sparse_982" : "GBRDCMBRGA======",
      "sparse_983" : "GBRDCMBRGA======",
      "sparse_984" : "GBRDCMBRGA======",
      "sparse_985" : "GBRDCMBRGA======",
      "sparse_986" : "GBRDCMBRGA======",
      "sparse_987" : "GBRDCMBRGA======",
      "sparse_988" : "GBRDCMBRGA======",
      "sparse_989" : "GBRDCMBRGA======"
}
```

Key “num” and “nested_obj.num” are unique across all the documents. Key “str1” is also unique across all the documents. “sparse_XX0” through “sparse_XX9” are sparse keys and same sparse keys repeat after every hundred documents. The range of values for sparse columns are “sparse_000” through “sparse_999”.

The benchmark consists of 10 queries, shown in the table below. The queries are executed by a single user and the response times are measured. The benchmark was modified to add four extra queries Q1a Q1b, Q2a and Q2b to include the MAX aggregation function on scalar and nested fields.

NoBench Queries

Query	**Query description**
Q1	`SELECT str,num FROM nobench_main, JSON_TABLE(jobj, '$' columns str varchar(128) path '$.str1', num NUMBER path '$.num') where rownum < 200;`
Q1a	`SELECT MAX( JSON_VALUE(jobj, '$.str1')) FROM nobench_main;`
Q1b	`SELECT MAX( JSON_VALUE(jobj, ‘$.str2’)) FROM nobench_main;`
Q2	`SELECT nested_str, nested_num FROM nobench_main, JSON_TABLE(jobj, '$' columns nested_str varchar(128) path '$.nested_obj.str', nested_num NUMBER path '$.nested_obj.num') where rownum < 200;`
Q2a	`SELECT MAX(JSON_VALUE(jobj, '$.nested_obj.str')) FROM nobench_main;`
Q2b	`SELECT MAX(JSON_VALUE(jobj, '$.nested_obj.num')) FROM nobench_main;`
Q3	`SELECT sparse_xx0, sparse_yy0 FROM nobench_main, JSON_TABLE(jobj, '$' columns sparse_xx0 varchar(128) path '$.sparse_XX0', sparse_yy0 varchar(128) path '$.sparse_XX9') WHERE (JSON_EXISTS(jobj, '$.sparse_XX0') OR JSON_EXISTS(jobj, '$.sparse_XX9'));`
Q4	`SELECT sparse_xx0, sparse_yy0 FROM nobench_main, JSON_TABLE(jobj, '$' columns sparse_xx0 varchar(128) path '$.sparse_XX0', sparse_yy0 varchar(128) path '$.sparse_YY0') WHERE (JSON_EXISTS(jobj, '$.sparse_460') OR JSON_EXISTS(jobj, '$.sparse_670'));`
Q5	`SELECT jobj FROM nobench_main WHERE JSON_VALUE(jobj format json, '$.str1' ) = :1;`
Q6	`SELECT jobj FROM nobench_main WHERE JSON_VALUE(jobj, '$.num' RETURNING NUMBER) BETWEEN :1 AND :2;`
Q7	`SELECT jobj FROM nobench_main WHERE JSON_VALUE(jobj, '$.dyn1' RETURNING NUMBER) BETWEEN :1 AND :2;`
Q8	`SELECT jobj FROM nobench_main WHERE json_textcontains(jobj, '$.nested_arr', 'accelerate');`
Q9	`SELECT jobj FROM nobench_main WHERE JSON_VALUE(jobj, '$.sparse_456' ) = 'dontfindme';`
Q10	`SELECT count(*) FROM nobench_main WHERE JSON_VALUE(jobj, '$.num' RETURNING NUMBER) BETWEEN :1 AND :2 GROUP BY JSON_VALUE(jobj,`

```
                          '$.thousandth')
```

The queries set consists of wide range of queries including single key selects, range scans, aggregation, and selection of documents based on sparse fields.

Performance Enhancements for JSON data

JSON with In-Memory Columnar Store (IMC)

Oracle Database 12c Release 1 introduced In-Memory Columnar option (also known as DBIM – Database In-Memory option) which is an in-memory columnar store to store tables and materialized views to speed up analytical queries that typically scan large number of records. In Oracle Database 12c Release 2 this feature is enhanced to support JSON in memory, where JSON is loaded into in-memory store in a binary format to speed up SQL/JSON queries that scan large number of JSON documents. The binary format is optimized to efficiently evaluate SQL/JSON path queries. The JSON in-memory feature uses the same Oracle SGA memory configured to store In-Memory columnar store and thus uses the same configuration parameters designed for Oracle In-Memory columnar feature.

The JSON In-Memory store feature benefits the following use cases:

» Non-DML intensive workload with SQL/JSON analytical queries based on JSON_TABLE functions and JSON_QUERY(), JSON_VALUE() and JSON_EXISTS() operators scanning a large number of JSON documents.

» Each JSON document is less than 32K in size. If the JSON document is larger than 32K, then it won't benefit from the JSON In-Memory feature.

» If there are functional indexes on the JSON column, the optimizer is intelligent enough to choose In-Memory scan or functional index on queries based on the selectivity of the query.

» In addition to storing entire JSON documents, the Database also allows caching of JSON_VALUE() or JSON_QUERY() expressions and JSON_TABLE() constructs. Users can create virtual columns on frequently queried JSON_VALUE(), JSON_QUERY() operators to project top level scalar values from a JSON column and have them loaded into Oracle In-Memory store. In-Memory virtual columns assume users have knowledge of the most frequently queried JSON operators. Ad-hoc SQL/JSON operators will still benefit from the JSON In-Memory store.

Configuring In-Memory store for JSON documents

The JSON In-Memory feature uses the same Oracle SGA memory configured to store In-Memory columnar store and thus uses the same configuration parameters designed for Oracle In-Memory columnar feature.

The following Oracle parameters need to be set in the init.ora/spfile parameter file:

```
compatible = 12.2.0.0
inmemory_expressions_usage =   STATIC_ONLY
inmemory_size              =   35433480192
inmemory_virtual_columns   =   ENABLE
```

The varchar2/CLOB/BLOB column storing JSON documents musts have the “IS JSON” check constraint to designate the column as a JSON column. The table can be loaded into In-Memory store by executing the following SQL commands:

```
alter table nobench_main add str1_vc as (JSON_VALUE(jobj format json, '$.str1'));
alter table nobench_main add num_vc as (JSON_VALUE(jobj format json,
                 '$.num' RETURNING NUMBER));
alter table nobench_main add nstr_vc  as (JSON_VALUE(jobj format json,
                 '$.nested_obj.str'));
alter table nobench_main add dyn1_vc  as (JSON_VALUE(jobj format json,
                 '$.dyn1' RETURNING NUMBER));
alter table nobench_main add constraint j_c check (jobj is json);
alter table nobench_main  inmemory;
select count(*) from nobench_main;
```

Virtual columns are created on the frequently used JSON operators. The virtual columns will also be loaded into the In-Memory store. In the above example, four virtual columns are created. Functional indexes are also created on these columns. Setting the inmemory_virtual_columns to ENABLE, enables storing of all the virtual columns of all the tables in the In-Memory store. The default value for this parameter is MANUAL. To manually store selective virtual columns the following statement can be issued, where vc1 and vc2 are previously created virtual columns.

```
alter table nobench_main inmemory inmemory(vc1,vc2);
```

A check constraint is added on the jobj column which holds the JSON document. The In-Memory store for the table is enabled by the alter table <table_name> inmemory command. The last command checks to make sure the table is fully loaded in memory. This query may return immediately, however the population of the In-Memory binary format of the JSON documents in the In-Memory store happens in the background.

To verify that the JSON documents are loaded into the memory, the following queries can be executed to check the In-Memory store segments.

```
SQL> select distinct SEGMENT_NAME, POPULATE_STATUS from gv$IM_SEGMENTS where
    SEGMENT_NAME='NOBENCH_MAIN';

SEGMENT_NAME
--------------------------------------------------------------------------------
INMEMORY_SIZE POPULATE_STAT
------------- -------------
NOBENCH_MAIN
  2.5547E+10 COMPLETED

SQL> select pool, alloc_bytes, used_bytes, populate_status from v$inmemory_area;

POOL                      ALLOC_BYTES USED_BYTES POPULATE_STATUS
-------------------------- ----------- ---------- --------------------------
1MB POOL                            2.8158E+10 2.5680E+10 DONE
64KB POOL                    722678579   30015488 DONE

SQL> select substr(column_name, 1, 13) as NM, HIDDEN_COLUMN from user_tab_cols
     where table_name = 'NOBENCH_MAIN';

 NM                             HID
 ---------------------------    ------------
 JOBJ                           NO
 SYS_NC00002$                   YES
 SYS_NC00003$                   YES
 SYS_NC00004$                   YES
 SYS_NC00005$                   YES
 SYS_IME_OSON_0001000000019160 YES
```

The query on gv$IM_SEGMENTS should show the POPULATE_STATUS on the table as "COMPLETED". Running the second query on v$inmemory_area shows the amount of In-Memory store being used. The inmemory_size parameter can be adjusted to fit the data in memory.

To verify that the JSON in-memory format is successfully loaded , first verify that the virtual column with name like SYS_IME_OSON_XXXXXXX is displayed when the table columns are inspected.

The query execution plan should show that the in-memory column is being used, as shown below. The query execution plan should show the TABLE ACCESS as INMEMORY_FULL on the table being queried. Also the "SYS_IME_OSON_XXXXX"[RAW,32767] parameter should be displayed in the column projection section of the explain plan.

```
SELECT MAX( JSON_VALUE(jobj, '$.str2')) FROM nobench_main;
Plan hash value: 1125102308
----------------------------------------------------
| Id  | Operation                     | Name         |
----------------------------------------------------
|   1 |  SORT AGGREGATE               |              |
|   2 |   TABLE ACCESS INMEMORY FULL| NOBENCH_MAIN |
----------------------------------------------------


Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - (#keys=0) MAX(JSON_VALUE("JOBJ" FORMAT JSON , '$.str2' RETURNING
       VARCHAR2(4000) NULL ON ERROR , "NOBENCH_MAIN"."SYS_IME_OSON_0001000003BB
       569C"))[4000]
   2 - "JOBJ"[VARCHAR2,4000],"NOBENCH_MAIN"."SYS_IME_OSON_0001000003BB5
       69C"[RAW,32767]
```

Creating a functional index on a JSON expression implicitly creates a virtual column. Any such virtual columns associated with functional indexes are automatically stored in the In-Memory store once the In-Memory option is enabled on the table. No extra steps are needed.

Benchmark Results

The NoBench benchmark was run with the In-Memory option enabled and with all the indexes disabled. Virtual columns were created and loaded into IMC on the expressions as described in the table above. With In-Memory option, the queries are sped up by scanning the In-Memory binary format of JSON and by scanning the virtual columns where applicable. The following graph shows the speed up achieved by In-Memory option. The y-axis of the graph is shown in log-scale to accommodate the large range of speedups in the graph.

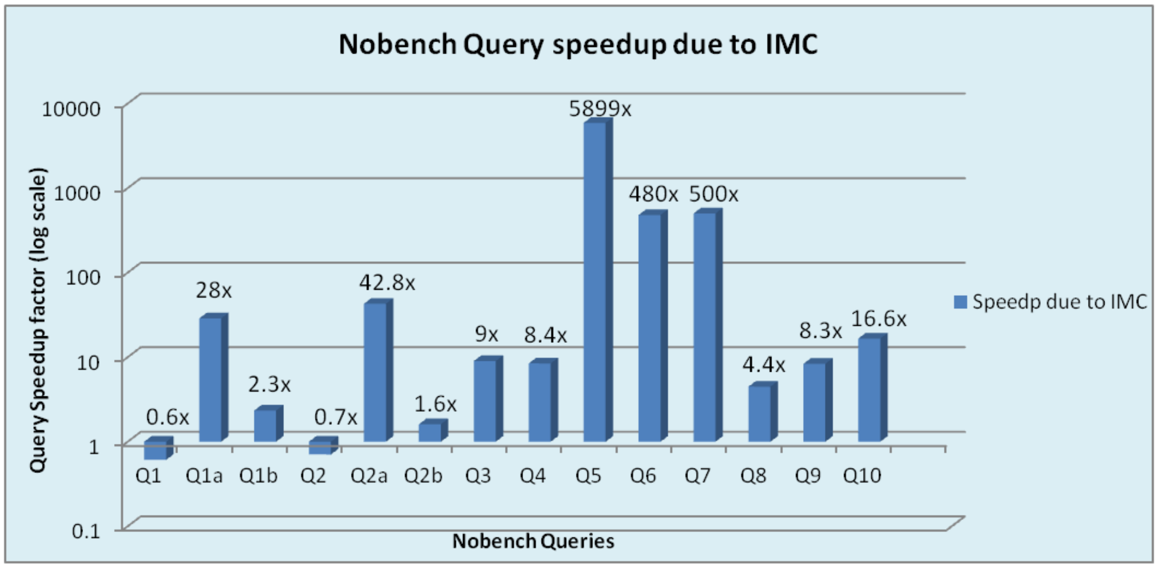

Figure 1

From the graph above, it can be clearly seen that In-Memory storage of the binary format of JSON and caching of the virtual column gives significant speedup in query execution times. In this test no functional indexes or JSON Search index was built, this was done to study the impact of In-Memory caching alone. The queries with high selectivity show speed-up due to scanning of the virtual columns (configured on JSON expressions on queried field) in the In-Memory store.

In-Memory Store Sizing

The In-Memory area for the Oracle Database 12c Release 2 has to be configured via an init.ora parameter. The In-Memory store has to be sized for the JSON text documents, in-memory binary format of JSON and any other virtual columns that are configured. The size of the binary format is same as the size of the original JSON text documents. To illustrate an example, for the above NoBench benchmark with 16 million documens of size 600 bytes each, the In-Memory store was sized as follows:

Total In-Memory Store = JSON Documents + JSON binary format + Virtual Columns
= 16000000 * 600 + 16000000 * 600 + <size of the virtual columns>

Functional and JSON Search Index

Functional indexes yield fast query performance with minimal DML overheads. Based on the query requirement, building functional indexes on JSON Path expressions will result in faster query execution. The following functional indexes were configured for this benchmark based on the query definitions.

```
create index j_get_str1 on nobench_main(JSON_VALUE(jobj format json, '$.str1'));
create index j_get_num on nobench_main(JSON_VALUE(jobj format json, '$.num' RETURNING
NUMBER));
create index j_get_nstr on nobench_main(JSON_VALUE(jobj format json,
'$.nested_obj.str'))
create index j_get_dyn1 on nobench_main(JSON_VALUE(jobj format json, '$.dyn1'
RETURNING NUMBER))
```

In cases where the predicate is not known (adhoc queries) or the keys being queried are sparse across the documents, instead of building multiple functional indexes, building a search index across all the documents in a table would be more efficient. The optimizer can utilize the search index to search and retrieve the documents that match. The following sample query demonstrates the use of the search index. This is especially suited for documents that are updated infrequently, where the overhead of maintaining the search index during updates and inserts will be less.

```
select sparse_xx0, sparse_yy0 FROM nobench_main,
      JSON_TABLE(jobj, '$' columns sparse_xx0 varchar(128) path '$.sparse_450',
                                   sparse_yy0 varchar(128) path '$.sparse_459'
                )
 WHERE (JSON_EXISTS(jobj, '$.sparse_450') OR JSON_EXISTS(jobj, '$.sparse_459'));


Execution Plan
Plan hash value: 3007270940

---------------------------------------------------------------
| Id  | Operation                    | Name                  |
---------------------------------------------------------------
|   1 |  NESTED LOOPS                |                       |
|   2 |   TABLE ACCESS BY INDEX ROWID| NOBENCH_MAIN          |
|*  3 |    DOMAIN INDEX              | NOBENCH_JSON_VCHAR_IDX |
|   4 |   JSONTABLE EVALUATION       |                       |
---------------------------------------------------------------

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

   3 - access("CTXSYS"."CONTAINS"("NOBENCH_MAIN"."JOBJ",'(HASPATH(/sparse_450))
                  or (HASPATH(/sparse_459))')>0)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) VALUE(A0)[128], VALUE(A0)[128]
   2 - "JOBJ"[VARCHAR2,4000]
```

In the query mentioned above, the keys being queried are sparsely populated across the JSON documents. Building a large number of functional indexes on the sparse keys would be impractical. A Search Index is a better option as shown in the query plan above.

Benchmark Results

The NoBench benchmark was run to study the impact of search index and the functional index. The following graph shows the speedup with functional and JSON search index compared to table scans. In this test IMC was turned off and only functional indexes and JSON search index were configured to study the impact of indexes alone. The queries with high selectivity benefit greatly with functional indexes (query Q5) and JSON search index (Q3, Q4, Q8 and Q9). Q1a and Q2a are aggregation queries using max() function, which benefit with the presence of functional index. Query execution times for these queries are greatly reduced due to the functional indexes built on the functional expressions. The y-axis of the graph is shown in log-scale to accommodate the large range of speedups in the graph.

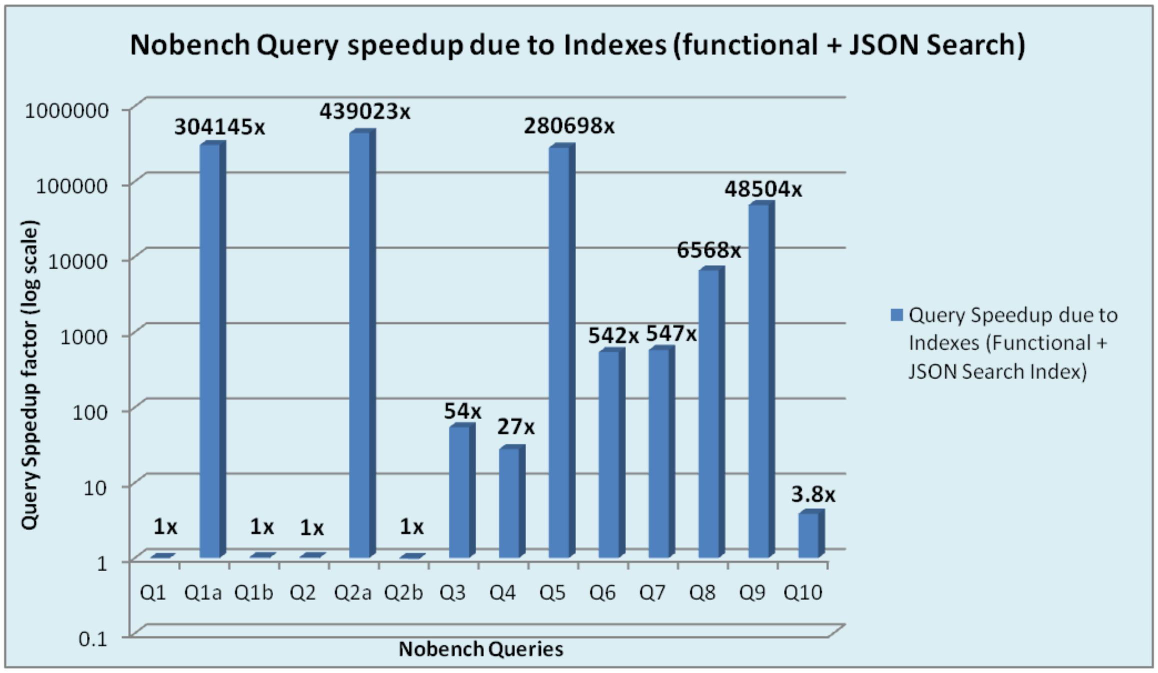

Figure 2

As it is evident from the graph above (Figure 2), JSON functional indexes and search index can be used to speedup queries with full search capability. One thing to bear in mind is the performance overhead of updating the search index during DML statements on the JSON document. The full Search index is well suited for JSON documents with infrequent updates.

In-Memory Store and Functional and JSON Search Indexes

Indexes and IMC can be configured together to derive benefits of both the features. The following graph (Figure 3) shows the speedup due to enabling IMC and configuring indexes (both functional and JSON Search index) compared to configuring only IMC..

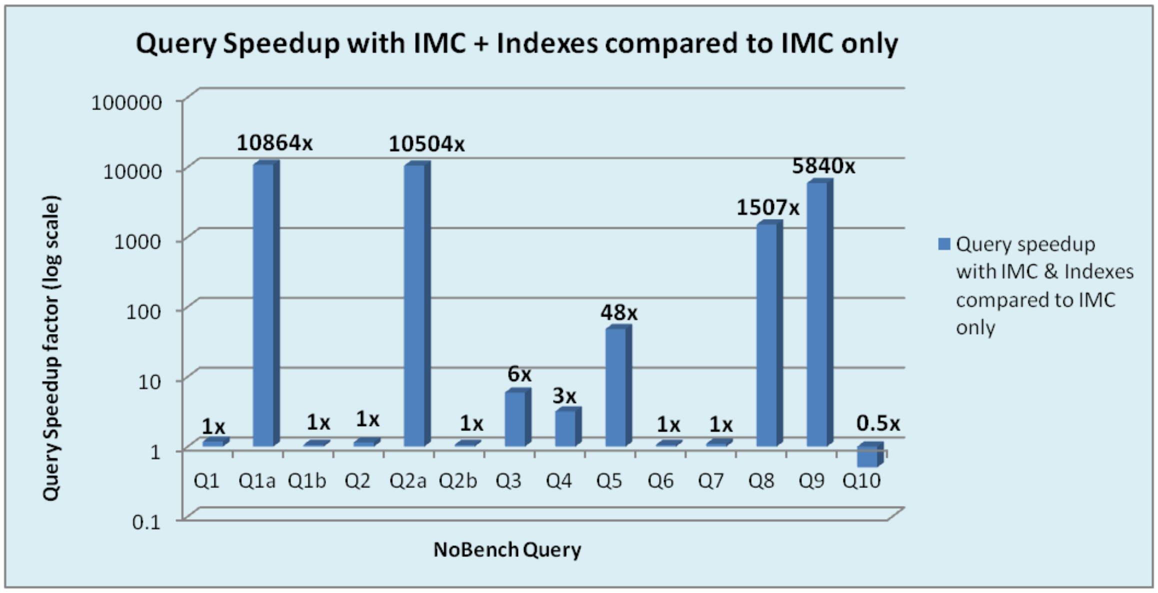

Figure 3

By configuring indexes based on selectivity, a further speedup can be seen for these queries, whereas still enjoying the speedup due to IMC for queries with low selectivity. The optimizer automatically picks the right plan (IMC vs Indexes) based on the query selectivity.

Performance Guidelines

The following section describes the performance guidelines while storing and retrieving JSON data in Oracle 12c Release 2 database.

- » JSON functional Indexes yield fast query performance with minimal DML overheads.
- » In-Memory store can be used to boost the performance of JSON queries across the board with minimal DML overhead.
    - » IMC improves performance on JSON data by using a post parsed binary representation in memory for fast scans. This features helps queries with low selectivity.
    - » IMC virtual columns are automatically created for any functional index expressions. No further steps are required other than just turning on IMC. The Optimizer is intelligent enough to pick either functional index or IMC scan of the virtual column based on the selectivity.
- » The JSON search index provides full search capability (both on keys and values) of JSON documents. Combined with functional index, the JSON search index provides the best performance in terms of response times. However the search index maintenance can add to the DML overhead. So the JSON search index is most suitable for read-mostly workloads.
- » For documents smaller than 4000 bytes, storing them in varchar2 columns gives the maximum performance. For documents smaller than 32KB (and larger than 4000 bytes) extended varchar2 gives the best performance. Documents larger than 32KB can be stored in BLOB columns.

Conclusion

Oracle Database 12c Release 2 provides performance enhancements for SQL/JSON processing which results in significant performance speed up for queries. The JSON full-text search index provides speed up for adhoc queries where the keys being looked up are not known a priori. It can also be used for cases where sparse keys are queried or where full text searches are needed. The Oracle Database 12c Release 2 In-Memory store feature has been enhanced to support an in-memory binary format of JSON for efficient SQL/JSON operations. In addition to loading the whole JSON document, virtual columns can also be loaded into the In-Memory store. Any virtual columns underlying functional indexes are automatically loaded into the In-Memory store. The optimizer is intelligent enough to pick a full In-Memory store scan or functional index based on the selectivity of the query.

White Paper JSON Performance features in Oracle 12c Release 2
December 2016