Test of Oracle JSON support in the view of CMS JSON data

August 2016

Author:
Sartaj Singh Baveja

Supervisor(s):
Katarzyna Maria Dziedziniewicz-Wójcik
Valentin Kuznetsov

CERN openlab Summer Student Report 2016

15 years CERN openlab

Abstract

Oracle has introduced native support for Javascript Object Notation (JSON) data in its 12c release with relational database features, including transactions, indexing, declarative querying and views.

The requirements for the CMS WMArchive project, whose goal is to reliably store its Workflow and Data Management framework job report (FWJR) documents, include storing deep nested JSON structures, running queries over them and aggregating data in an effective way.

The objective of this project is to assess, evaluate and test the capabilities and performance of Oracle JSON with respect to the currently used solution, MongoDB. The comparison is based on functionality, read/write rates and indexing.

Initially, JSON documents are created by randomizing a sample CMS FWJR document and inserted into both MongoDB and Oracle to evaluate the performance. Then, the data stored in these databases is queried with and without indexes. Performance is then evaluated and a comparison is made. Other performance metrics such as CPU Usage, data and index size are also compared.

Table of Contents

1 Introduction 4
1.1 What is JSON? 4
1.2 Importance of JSON 5
2 CMS WMArchive Project 6
2.1 Introduction 6
2.2 Architecture 7
2.3 Data Collection 7
2.4 Requirements 7
3 Current Solution: MongoDB 9
4 JSON in Oracle Database 10
4.1 Oracle Database 12.1 Release 10
4.2 Oracle Database 12.2 Beta 13
4.3 Issues with Oracle 12.1 Release 14
5 Procedure 15
5.1 Generating JSON Documents 15
5.2 Inserting Data 15
5.2 Querying Data 19
5.3 Creating Indexes 20
5.4 Indexed vs Non Indexed Queries 21
5.5 CPU Performance 26
6. Comparison 27
7. Conclusion ....Error! Bookmark not defined.
8. Appendix 29
9. References 32

1 Introduction

1.1 What is JSON?

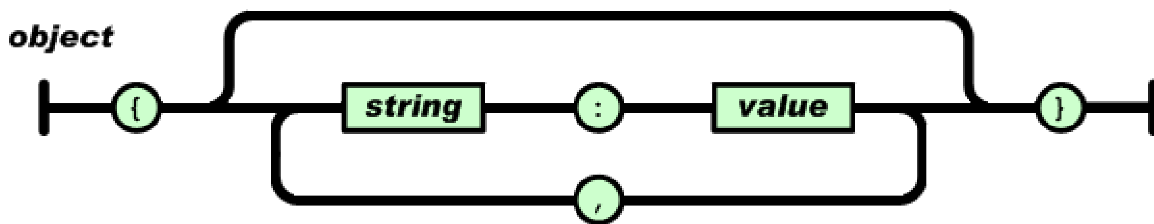
JSON (JavaScript Object Notation) is a lightweight data-interchange format, which is easy for humans to read and write. Because it is (almost a subset of) JavaScript notation, JSON can often be used in JavaScript programs without any need for parsing or serializing.

Although it was defined in the context of JavaScript, JSON is in fact a language-independent data format. A variety of programming languages can parse and generate JSON data. It is often used for serializing structured data and exchanging it over a network, typically between a server and web applications.

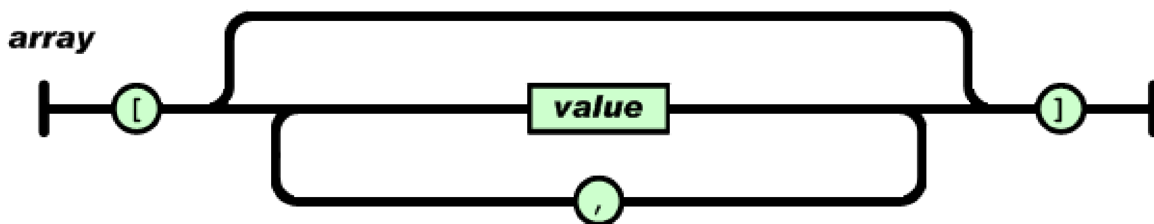
JSON is built on two structures:

- A collection of name/value pairs
- An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

An *object* is an unordered set of name/value pairs. An object begins with **{** (left brace) and ends with **}** (right brace). Each name is followed by**:** (colon) and the name/value pairs are separated by **,** (comma).



An *array* is an ordered collection of values. An array begins with **[** (left bracket) and ends with **]** (right bracket). Values are separated by **,** (comma).



An example of a json structure would be :

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

1.2 Importance of JSON

JSON is becoming really popular these days because of the following reasons :

- Lightweight data interchange format used in web services responses
- Easy to parse by different programming languages
- Almost a subset of JavaScript notation and thus, can be used in JavaScript programs without any need for serializing or parsing

2 CMS WMArchive Project

2.1 Introduction

The Compact Muon Solenoid (CMS) is one of the two general purpose particle physics detectors operated at the LHC. It is designed to explore the frontiers of physics and provide physicists with the ability to look at the conditions presented in the early stage of our Universe. More then 3,000 physicists from 183 institutions representing 38 countries are involved in the design, construction and maintenance of the experiments.

Experiment of this magnitude requires a vast and complex distributed computing and data model. CMS spans more than a hundred data centers in a three-tier model and generates around 10 petabytes (PB) of data each year in real data, simulated data and meta-data.

This information is stored and retrieved from relational and non-relational data-sources, such as relational databases, document databases, blogs, wikis, file systems and customized applications.

WMAgent is a set of services used by CMS to provide a grid workload management system. It is comprised of:

- A SQL based database (default MySQL) to keep the WMAgent state
- A non-relational database for storage of job reports and state information
- The RequestManager, which handles the creation and approval of requests for new work
- The WorkQueue, a multi-level system that deals with finding and injecting files when the system is ready for new work
- The Agent which creates, submits, and evaluates the actual jobs

Retrieving the log file for any particular job is a difficult and time consuming process involving several database lookups:

- Groups of log files are saved in a single archive file
- Retrieving the correct archive file
- Extracting the correct log file from that archive
- It takes multiple tries to finally get the log file

The WMArchive project aims at storing and further analyzing job log files to better understand the processing time of the job and help planning on future processing and resource requirements. WMArchive should also make it easier to generate reports for reviews or meetings.

2.2 Architecture

- WMAgent/JobStateMachine will be the data source of WMArchive.
- WMArchive should not be a burden to WMAgent
- WMArchive should be robust, losing data is not acceptable
- WMArchive should be flexible enough to allow adding or removing new information
- Quick document inserting is more important that data retrieving

2.3 Data Collection

Everyday, there are 200K-300K documents being generated by dozens of WMAgents running at different sites. These documents represent the **CMS Workflow and Data Management Framework Job Report documents (FWJR)**. Each document is around 12kB in size which means that 3GB of data is produced every single day.

Every document is in a JSON data format whose structure is not subject to change and has a deep nested structure

2.4 Requirements

WMArchive should implement the following requirements for both failed and succeeded jobs:

- WMAgent Framework Job Report (FWJR) will be the source of WMArchive.
- when a job fails and it is resubmitted again, two FWJR will be created in WMArchive as two separated docs.
- gather as many statistics as possible for each job
    - date and time of the job ran
    - CMSSW version used
    - CPU information
    - LFN of the log file
- it should include all the information monitored in WMStats
- store log file location for all the jobs
- provide flexible queries

Here's an example of the CMS FWJR document

```
# CMS FWJR data structure
{"meta_data": {"agent_ver": "1.0.14.pre5",
               "fwjr_id": "1-0",
               "host": "a.b.com",
               "jobtype": "Processing",
               "jobstate": "success",
               "ts": 1456500229},
 "LFNArray": ["/store/file1.root",
              "/store/file2.root",
              "/lfn/fallbackfile.root", "/lfn/skipedfile.root"],
 "LFNArrayRef": ["fallbackFiles",
                 "outputLFNs",
                 "lfn",
                 "skippedFiles",
                 "inputLFNs"],
 "PFNArray": ["root://file1.root",
              "root://file2.root",
              ],
 "PFNArrayRef": ["inputPFNs", "outputPFNs", "pfn"],
 "steps": [{"name": "cmsRun1",
            "analysis": {},
            "cleanup": {},
            "logs": {},
            "errors": [
                  {
                      "details": "An exception",
                      "type": "Fatal Exception",
                      "exitCode": 8001
                  }
              ],
            "input": [{"catalog": "",
                       "events": 6893,
                       "guid": "E8099605-8853-E011-A848-0030487A18F2",
                       "input_source_class": "PoolSource",
                       "input_type": "primaryFiles",
                       "lfn": 0,
                       "module_label": "source",
                       "pfn": 0,
                       "runs": [{"lumis": [164, 165],
                                 "runNumber": 160960}]}],
            "output": [{"StageOutCommand": "rfcp-CERN",
                        "acquisitionEra": "CMSSW_7_0_0_pre11",
                        "adler32": "e503b8b9",
                        "applicationName": "cmsRun",
                        "applicationVersion": "CMSSW_7_0_0_pre11",
                        "async_dest": "",
                        "branch_hash": "c1e135af4ac2eb2b803bb6487be2c80f",
                        "catalog": "",
                        "cksum": "2641269665",
                        "configURL": "https://hostname/couchdb",
                        "events": 0,
                        "globalTag": "GR_R_62_V3::All",
                        "guid": "ECCFE421-08CB-E511-9F4C-02163E017804",
                        "inputDataset": "/Cosmics/Run2011A-v1/RAW",
                        "inputLFNs": [0],
                        "inputPFNs": [0],
```

3 Current Solution: MongoDB

Initially, CouchDB was chosen to implement the WMArchive DB backend. However, it was not possible to have flexible queries and large data storage. Therefore, MongoDB was chosen.

MongoDB is a free and open-source cross-platform document-oriented database. Classified as a NoSQL database, MongoDB avoids the traditional table based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster.

Some of the features of MongoDB are :

1. **Ad Hoc Queries**

   MongoDB supports field, range queries, regular expression searches. Queries can return specific fields of documents and also include user-defined JavaScript functions. Queries can also be configured to return a random sample of results of a given size.

2. **Indexing**

   Any field in a MongoDB document can be indexed – including within arrays and embedded documents. Primary and secondary indices are available.

3. **Replication**

   MongoDB provides high availability with replica sets. A replica set consists of two or more copies of the data. Each replica set member may act in the role of primary or secondary replica at any time. All writes and reads are done on the primary replica by default. Secondary replicas maintain a copy of the data of the primary using built-in replication.

4. **Scaling**

   MongoDB scales horizontally using sharding. The data is split into ranges and distributed across multiple shards. MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

5. **Aggregation**

   MapReduce can be used for batch processing of data and aggregation operations. The aggregation framework enables users to obtain the kind of

results for which the SQL GROUP BY clause is used. The aggregation framework includes the $lookup operator which can join documents from multiple documents.

4 JSON in Oracle Database

4.1 Oracle Database 12.1 Release

Unlike relational data, JSON data can be stored, indexed, and queried *without any need for a schema* that defines the data. Oracle Database supports JSON natively with relational database features, including transactions, indexing, declarative querying, and views.

JSON data has often been stored in NoSQL databases such as Oracle NoSQL Database and Oracle Berkeley DB. These allow for storage and retrieval of data that is not based on any schema, but they do not offer the rigorous consistency models of relational databases.

To compensate for this shortcoming, a relational database is sometimes used in parallel with a NoSQL database. Applications using JSON data stored in the NoSQL database must then ensure data integrity themselves.

Native support for JSON by Oracle Database obviates such workarounds. It provides all of the benefits of relational database features for use with JSON, including transactions, indexing, declarative querying, and views.

Oracle Database queries are declarative. You can join JSON data with relational data. And you can project JSON data relationally, making it available for relational processes and tools. You can also query, from within the database, JSON data that is stored outside the database in an external table. You can access JSON data stored in the database the same way you access other database data.

JSON data is stored in Oracle Database using SQL data types **VARCHAR2, CLOB** and **BLOB**. Oracle recommends that you always use an *is_json* check constraint to ensure that column values are valid JSON instances.

In SQL, you can access JSON data stored in Oracle Database using the following:

- Functions json_value, json_query, and json_table.
- Conditions json_exists, is json, is not json, and json_textcontains.

- A dot notation that acts similar to a combination of json_value and json_query and resembles a SQL object access expression, that is, attribute dot notation for an abstract data type (ADT)

```
CREATE TABLE j_purchaseorder
   (id          RAW (16) NOT NULL,
    date_loaded TIMESTAMP WITH TIME ZONE,
    po_document CLOB
    CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

Example 39-2 Simple SQL Query of JSON Data

```
SELECT po.po_document.Requestor FROM j_purchaseorder po;
```

Oracle JSON Path Expressions

Oracle Database provides SQL access to JSON data using Oracle JSON path expressions. An Oracle JSON path expression selects zero or more JSON values that match, or satisfy, it.

Oracle SQL condition **json_exists** returns true if at least one value matches, and false if no value matches. If a single value matches, then SQL function **json_value** returns that value if it is scalar and raises an error if it is non-scalar. If no value matches the path expression then json_value returns SQL NULL.

Oracle SQL function **json_query** returns all of the matching values, that is, it can return multiple values.

```
SELECT json_query(po_document, '$.ShippingInstructions.Phone[*].type'
                               WITH WRAPPER)
  FROM j_purchaseorder;
```

Example 39-13 Projecting an Entire JSON Array as JSON Data

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$'
         COLUMNS (requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
                  ph_arr    VARCHAR2(100 CHAR) FORMAT JSON
                            PATH '$.ShippingInstructions.Phone')) AS "JT";
```

Wrapper Clause for Oracle SQL Functions

Oracle SQL functions json_query and json_table accept an optional wrapper clause, which specifies the form of the value returned by json_query or used for the data in a json_table relational column.

The wrapper clause takes one of these forms:

- **WITH WRAPPER** – Use a string value that represents a JSON array containing *all* of the JSON values that match the path expression. The order of the array elements is unspecified.
- **WITHOUT WRAPPER** – Use a string value that represents the *single* JSON *object* or *array* that matches the path expression. Raise an error if the path expression matches either a scalar value (not an object or array) or more than one value.
- **WITH CONDITIONAL WRAPPER** – Use a string value that represents *all* of the JSON values that match the path expression.

Simple Dot Notation Access to JSON Data

A simple dot-notation syntax is provided for queries, as an alternative to using the more verbose but more flexible Oracle SQL functions json_query and json_value. The dot notation is designed to return JSON values whenever possible.

The behavior of a query using the dot notation is different from both json_query and json_value. In effect, it combines their behavior to return one or more JSON values whenever possible.

Example 39-18 JSON Dot-Notation Query Compared with JSON_VALUE

```
SELECT po.po_document.PONumber FROM j_purchaseorder po;

SELECT json_value(po_document, '$.PONumber') FROM j_purchaseorder;
```

Example 39-19 JSON Dot-Notation Query Compared with JSON_QUERY

```
SELECT po.po_document.ShippingInstructions.Phone FROM j_purchaseorder po;

SELECT json_query(po_document, '$.ShippingInstructions.Phone')
  FROM j_purchaseorder;
```

4.2 Oracle Database 12.2 Beta

Oracle Database 12c Release 2 (12.2.0.1) is an improvement over the earlier release. It has the following new features :

a) Simple Dot-Notation Syntax Supports Array Access

You can now access arrays and their elements using the simple dot-notation

syntax.

b) Path Expression Enhancements

JSON path expressions can now include filter expressions that must be satisfied by the matching data and transformation methods that can transform it.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$?(@.LineItems.Part.UPCCode == 85391628927
                                      && @.LineItems.Quantity > 3)');
```

c) SQL/JSON Functions and Conditions Added to PL/SQL

SQL/JSON functions json_value, json_query, json_object, and json_array, as well as SQL/JSON condition json_exists, have been added to the PL/SQL language as built-in functions (json_exists is a Boolean function in PL/SQL).

```
SELECT json_object('title'       VALUE job_title,
                   'salaryRange' VALUE json_array(min_salary, max_salary))
  FROM jobs;

JSON_OBJECT('TITLE'ISJOB_TITLE,'SALARYRANGE'ISJSON_ARRAY(MIN_SALARY,MAX_SALARY))
--------------------------------------------------------------------------------
{"title":"President","salaryRange":[20080,40000]}
{"title":"Administration Vice President","salaryRange":[15000,30000]}
{"title":"Administration Assistant","salaryRange":[3000,6000]}
```

d) Search Enhancements

You can create a JSON search index. Range search is now available for numbers and JSON strings that can be cast as built-in date and time types.

Example 25-12 Creating a JSON Search Index

```
CREATE SEARCH INDEX po_search_idx ON j_purchaseorder (po_document) FOR JSON;
```

e) JSON Columns In the In-Memory Column Store

You can now store JSON columns in the in-memory column store, to improve query performance.

f) SQL/JSON Functions for Generating JSON Data

You can now construct JSON data programmatically using SQL/JSON functions json_object, json_array, json_objectagg, and json_arrayagg.

4.3 Issues with Oracle Release 12.1

1. Simplified dot notation doesn't work on accessing array elements in 12.1 whereas it works in 12.2

In Oracle 12.1

```
SELECT test.doc.LFNArray[0] from test11 test;
```

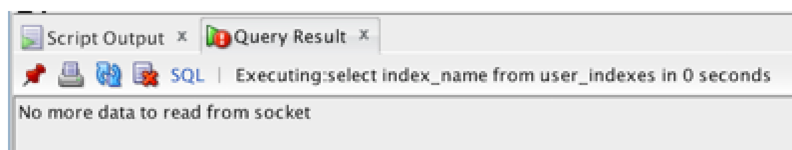
```
ORA-00923: FROM keyword not found where expected
00923. 00000 -  "FROM keyword not found where expected"
*Cause:
*Action:
```

In Oracle 12.2

```
SELECT test.doc.LFNArray[0] from test8 test;
```

	LFNARRAY
1	/store/mc/Run660/file0.root
2	/store/mc/Run661/file0.root
3	/store/mc/Run662/file0.root
4	/store/mc/Run663/file0.root
5	/store/mc/Run664/file0.root

2. Faced ORA-600 and ORA-7445 errors stating "No Data to be read from socket" in 12.1



3. Oracle Error "Missing right parentheses" occurred due to some issue in Release 12.1. Queries such as "**SELECT count(*) FROM table**" were giving the same error output. Due to this, the entire table had to be dropped

```
ORA-00907: missing right parenthesis
00907. 00000 -  "missing right parenthesis"
*Cause:
*Action:
Error at Line: 2 Column: 16
```

5 Procedure

5.1 Generating JSON Documents

Initially, based on a sample CMS FWJR document, various fields of the document were randomized by a python script. For this, the mongo client from the ***pymongo*** library was imported for MongoDB and ***cxOracle*** was installed for Oracle. Then, some fields were modified such as LFNArray, PFNArray, wmaid, site among a few.

The steps for installing ***cxOracle*** on MacOSX were documented here : https://gist.github.com/sartaj10/03936b3dc5f9d0499f93e06cc12eb52e

```
125  def randomizeDoc(doc, idx, index, x):
126      newdoc = copy.deepcopy(doc)
127      del newdoc['_id']
128
129      newdoc['wmaid'] = get_random_string(32)
130
131      for i in range(len(newdoc['steps'])):
132          storage_key = newdoc['steps'][i]['performance']['storage']
133
134          storage_key['writeTotalMB'] = round(random.uniform(200,400), 2)
135          storage_key["readPercentageOps"] = random.uniform(1, 2)
136          storage_key["readMBSec"] = random.uniform(0,1)
137
138          steps_key = newdoc['steps'][i]
139
140          steps_key['site'] = 'T' + str(random.randint(1,5)) + '_US_FNAL_Disk'
141          output_length = len(steps_key['output'])
142
```

5.2 Inserting Data

5.2.1 MongoDB

One Million(1M) JSON documents were randomly generated by the Python script and inserted into MongoDB using a row by row insert.

- db.collection.insert()
  Inserts a document or documents into a collection.

```
1.  def loadFiles(db, doc):
2.
3.      for idx in range(1000000):
4.          newdoc = dict(doc)
5.          del newdoc["_id"]
6.
7.          newdoc["wmaid"] = idx
8.          ........
9.          db.production.insert(newdoc)
```

To achieve better performance, we switched to *Bulk Inserts.* For this, we used the *initialize_ordered_bulk_op()* method.

- db.collection.initialize_ordered_bulk_op()

  Initializes and returns a new Bulk() operations builder for a collection. The builder constructs an ordered list of write operations that MongoDB executes in bulk.

```
1. def init():
2.     for i in range(4):
3.         bulkInsert(db,doc,i)
4.
5. def bulkInsert(db, doc, index):
6.     bulk = db.production.initialize_ordered_bulk_op()
7.     x = 250000
8.
9.     for idx in range(x):
10.        newdoc = randomizeDoc(doc, idx, index, x)
11.        bulk.insert(newdoc)
12.
13.    result = bulk.execute()
```

5.2.2 Oracle

First, a connection to the Oracle database was made using cxOracle which is a Python extension module that enables access to Oracle databases. Insertion was both carried out using row-by-row insert as well as bulk inserts.

a) Row inserts,

```
1. for i in range(1000000):
2.     json_doc = generateJSON(doc, i)
3.     cursor.execute("INSERT INTO testDocument VALUES (:input)", input = json_doc)
```

- cx_Oracle.Cursor.execute(statement, [parameters])

  This method can accept a single argument - a SQL statement - to be run directly against the database. Bind variables assigned through the parameters can be specified as a dictionary, sequence, or a set of keyword arguments. This method returns a list of variable objects if it is a query, and None when it's not.

To improve the time, we used *bind variables* and *prepare statements.*

Bind variables are core principles of database development. They do not only make programs run faster but also protect against SQL injection attacks. By using bind variables you can tell Oracle to parse a query only once. Otherwise, when run one-by-one, each need to be parsed separately which adds extra overhead to your application.

```
query1 = cursor.execute('SELECT * FROM employees WHERE
department_id=:dept_id AND salary>:sal', named_params)
```

When binding, you can first **prepare** the statement and then execute None with changed parameters. Oracle will handle it such that one prepare is enough when variables are bound.

```
1. cursor.prepare("INSERT INTO testDocument VALUES (:input)")
2. for i in range(1000000):
3.     json_doc = generateJSON(doc, i)
4.     cursor.execute(None, input = json_doc)
```

b) Bulk Inserts

Large insert operations don't require many separate inserts because Python fully supports inserting many rows at once with the cursor.executemany method. Limiting the number of execute operations improves program performance a lot and should be the first thing to think about when writing applications heavy on INSERTs.

```
1. def batch_insert(cursor, doc, db):
2.     cursor.prepare("INSERT INTO testDocument VALUES (:1)")
3.
4.     for j in range(4):
5.         document = []
6.         for i in range(250000):
7.             json_doc = generateJSON(doc, j, i)
8.             row = (json_doc,)
9.             document.append(row)
10.
11.         cursor.executemany(None, document)
12.         db.commit()
```

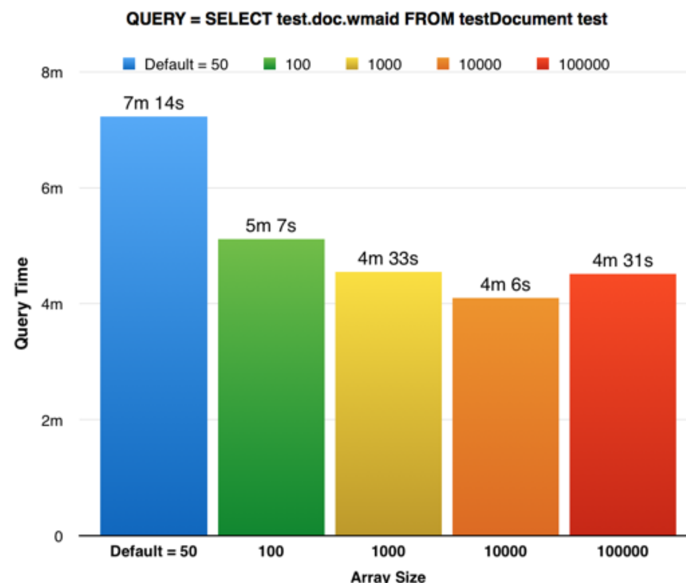


Figure: Variation of time as the BindArraySize variable is changed from the default value i.e. 50 till 100,000 when number of rows in the table are 1M

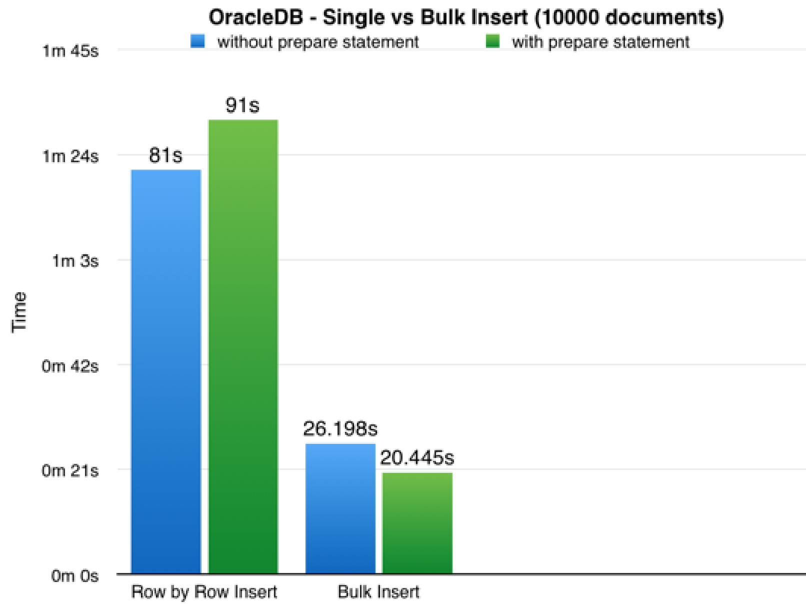


Figure: Variation of Insert Time with/without the bind variable when number of rows to be inserted is 10,000

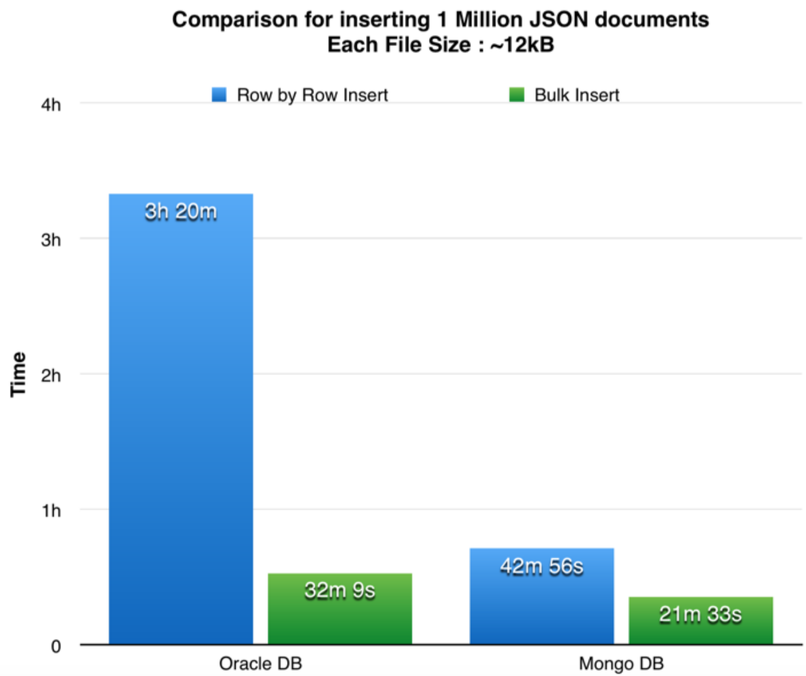


Figure: Comparison of injection rate of 1M JSON Documents in both Oracle and MongoDB (row-by-row and bulk insert)

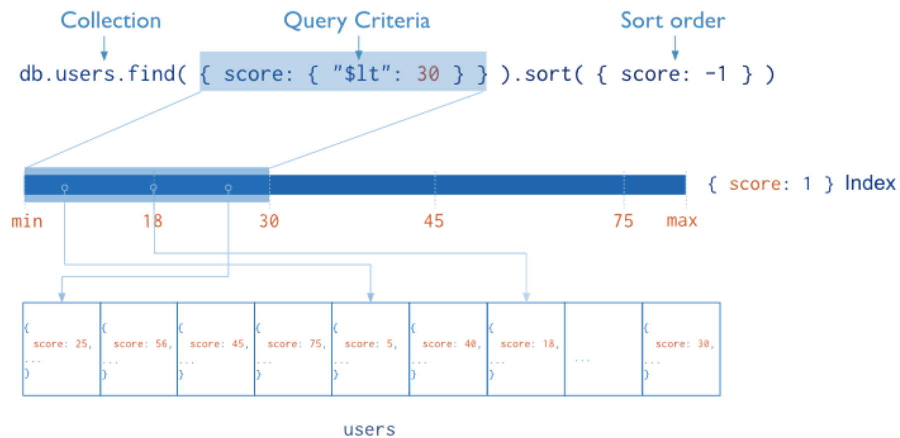
5.2 Querying Data

Queries were run on the data stored in both Oracle as well as MongoDB. A few queries were executed to :

- Search for a specific string
- Aggregate data
- Find records based on provided pattern
- Comparison Query Operators
- Logical Query Operators

Initially, these queries were run without creating indexes. Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

The following diagram illustrates a query that selects and orders the matching documents using an index:



MongoDB provides the db.collection.find() method to read documents from a collection. The db.collection.find() method returns a cursor to the matching documents.

For the db.collection.find() method, you can specify the following optional fields:

- Query filter to specify which documents to return.
- Query projection to specifies which fields from the matching documents to return. The projection limits the amount of data that MongoDB returns to the client over the network.

```
def query(db):
    cursor = db.production.find({'PFNArray':'root://test.ch/Run123/file0.root'})
    cursor = db.production.find({'steps.output.runs.runNumber':2})
    cursor = db.production.find({'steps.site':'T2_US_FNAL_Disk'})
    cursor = db.production.aggregate([
        ])
    cursor = db.production.find({'wmaid':'88JEntUcP6G5rbCGudE07rakfWjfg5rg'})
    cursor = db.production.find({'PFNArray':{'$regex':'^root://test.ch/Run214/'}})
    cursor = db.production.find({'LFNArray':{'$regex':'^/store/mc/Run727/'}})
    cursor = db.production.find({"$or":[
                                        ]
                                })
    cursor = db.production.find({'steps.performance.storage.writeTotalMB':{'$gte': 200, '$lte': 250}})
```

Figure: Writing queries for MongoDB

```
77  # Starts with Operator / PFN Array Regex
78
79  select t.doc.LFNArray , t.doc.PFNArray
80    from test3 t
81      where json_exists(
                          t.doc,
                          '$?(@.PFNArray starts with $str)'
                          passing 'root://test.ch/Run1' as "str"
                          );
86
```

Figure: Writing Query for Oracle

5.3 Creating Indexes

To create indexes in MongoDB, db.collection.createIndex(**keys, options**) where keys is a document containing the key value pairs of the fields which need to be indexed.

```
def createIndex(db):
    db.production.create_index([("steps.performance.storage", pymongo.ASCENDING)])
```

In case of Oracle, there are various types of indexes that can be created. Bitmap Indexes are used when there are only few possible values for the field in your data.

Function Based JSON_VALUE or JSON_QUERY indexes can also be created with a returning data type and an ERROR on ERROR clause. The use of ERROR ON ERROR here means that if the data contains a record that either doesn't have that field or has that field with a *non-number* value then index creation fails.

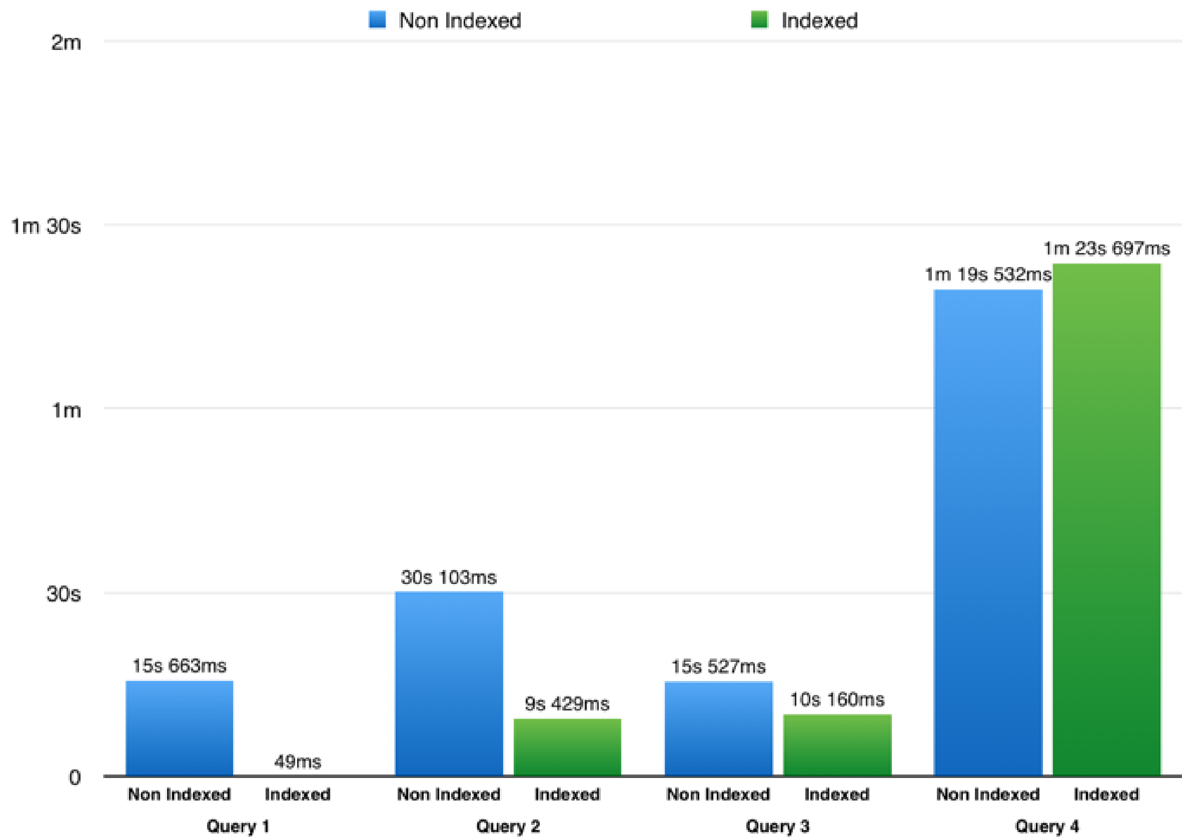
```
SQL> create index wmaid on testDocument test (test.doc.wmaid);

Index created.

Elapsed: 00:13:42.61
```

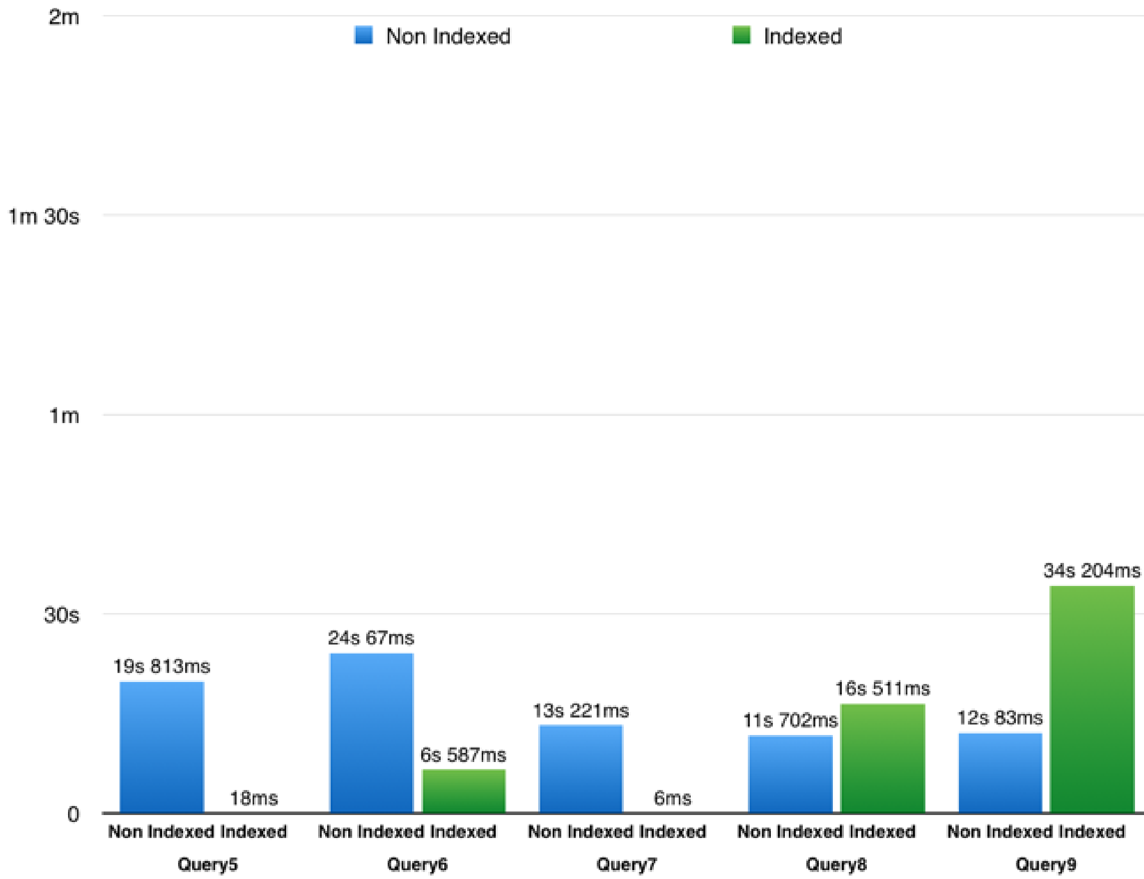
```
create index ind_lfn on testDocument json_query(doc, '$.LFNArray');
```

5.4 Indexed vs Non Indexed Queries

5.4.1 MongoDB

The following results were obtained when a few queries were run on MongoDB with and without an index.

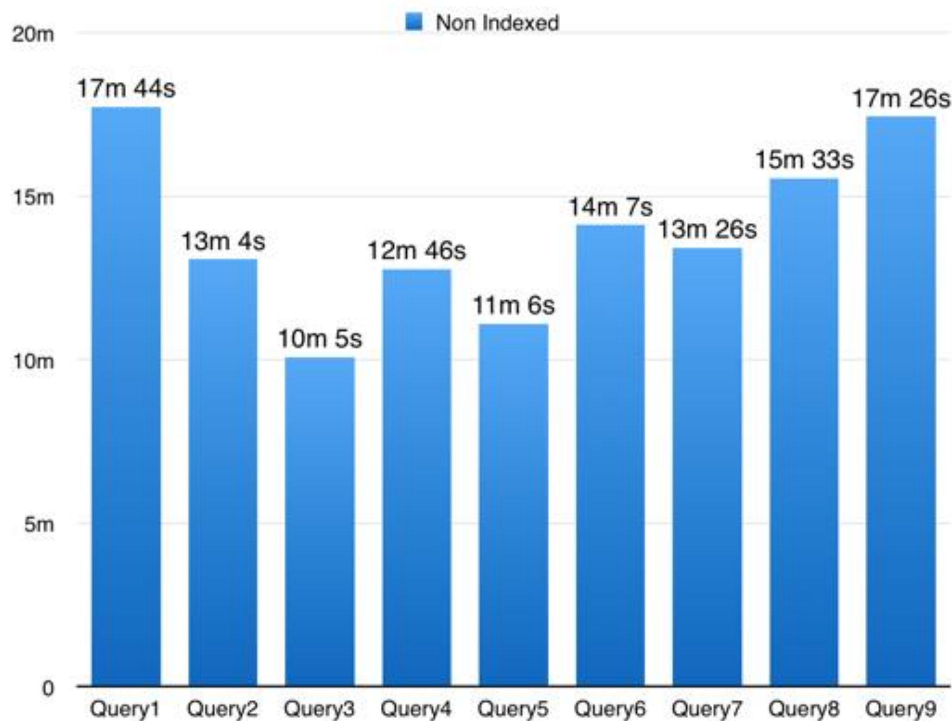



Query 5: Find a string
Query 6/7: Find records based on provided pattern
Query 8,9: Logical / Comparison Query Operators
2m
1m 30s
1m
30s
0
Non Indexed
Indexed
19s 813ms
18ms
24s 67ms
6s 587ms
13s 221ms
6ms
11s 702ms
16s 511ms
12s 83ms
34s 204ms
Non Indexed Indexed
Query5
Non Indexed Indexed
Query6
Non Indexed Indexed
Query7
Non Indexed Indexed
Query8
Non Indexed Indexed
Query9

5.4.2 Oracle Database

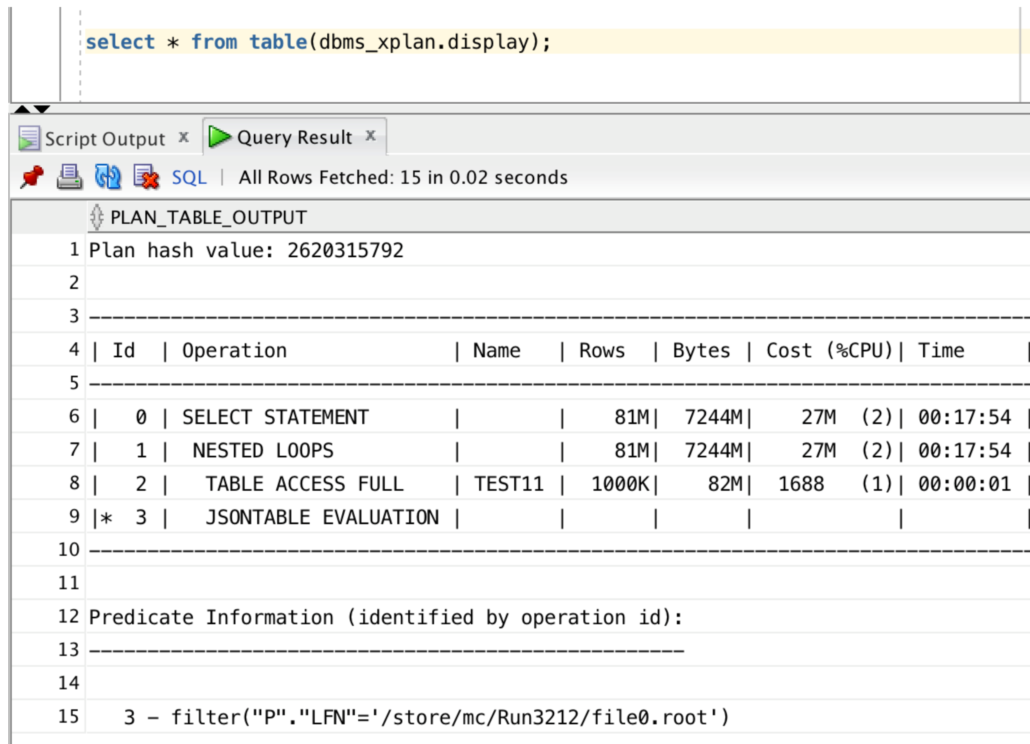
The following results were obtained when the queries were run in Oracle.

Query 1/2/3: Search for specific string
Query 4: Aggregate data
Query 5: Find a string
Query 6/7: Find records based on provided pattern
Query 8,9: Logical / Comparison Query Operators



When the same queries were run **after creating indexes**, the Oracle Optimizer chose the Full Table Scan path rather than executing the query using the index.

```
create index ind_lfn on test11 json_query(doc, '$.LFNArray');

explain plan for SELECT M.*
  FROM test11 p,
       json_table(
        p.doc,
        '$'
        columns (
          wmaid varchar2(2000 char) path '$.wmaid',
          meta_data varchar2(2000 char) format json with wrapper path '$.meta_data',
          nested path '$.LFNArray[*]'
          columns (
            lfn varchar2(2000 char) path '$'
          )
        )
      ) M
WHERE lfn = '/store/mc/Run3212/file0.root';
```



```
select * from table(dbms_xplan.display);
```

Script Output × | Query Result ×

SQL | All Rows Fetched: 15 in 0.02 seconds

```
PLAN_TABLE_OUTPUT
1  Plan hash value: 2620315792
2
3  ----------------------------------------------------------------------------------
4  | Id  | Operation              | Name   | Rows  | Bytes | Cost (%CPU)| Time     |
5  ----------------------------------------------------------------------------------
6  |   0 | SELECT STATEMENT       |        |   81M|  7244M|    27M   (2)| 00:17:54 |
7  |   1 |  NESTED LOOPS          |        |   81M|  7244M|    27M   (2)| 00:17:54 |
8  |   2 |   TABLE ACCESS FULL    | TEST11 | 1000K|    82M|  1688   (1)| 00:00:01 |
9  |*  3 |   JSONTABLE EVALUATION |        |      |       |            |          |
10 ----------------------------------------------------------------------------------
11
12 Predicate Information (identified by operation id):
13 ---------------------------------------------------
14
15    3 - filter("P"."LFN"='/store/mc/Run3212/file0.root')
```

After analyzing the 10053 Trace, it was seen that the optimizer determined that the cost of computing the result with Index was higher than the cost of computing without an index

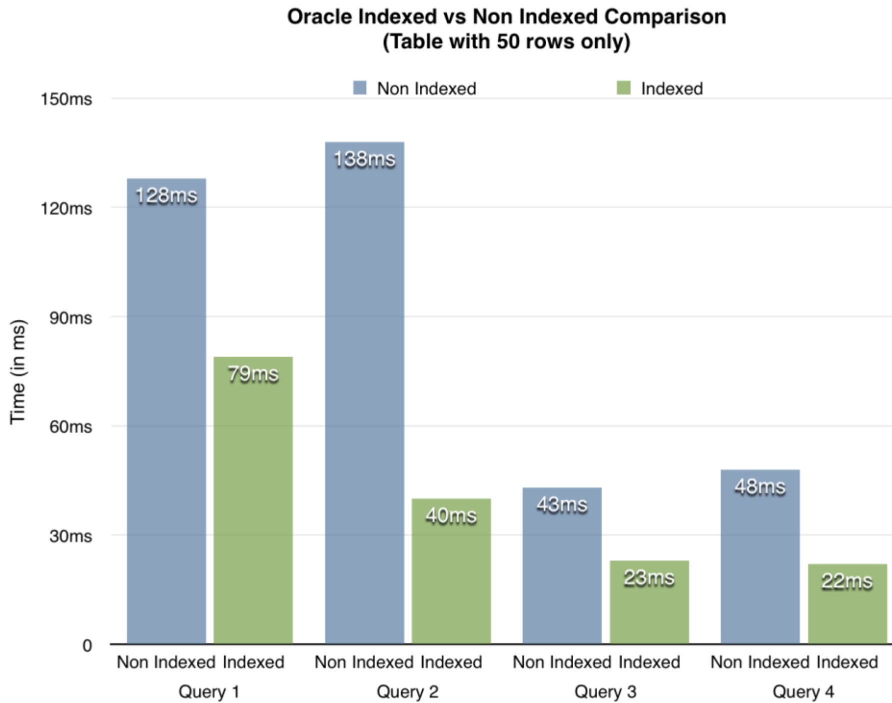
```
alter session set events '10053 trace name context forever, level 1';

SELECT VALUE FROM V$DIAG_INFO WHERE NAME = 'Default Trace File';

exit;
```

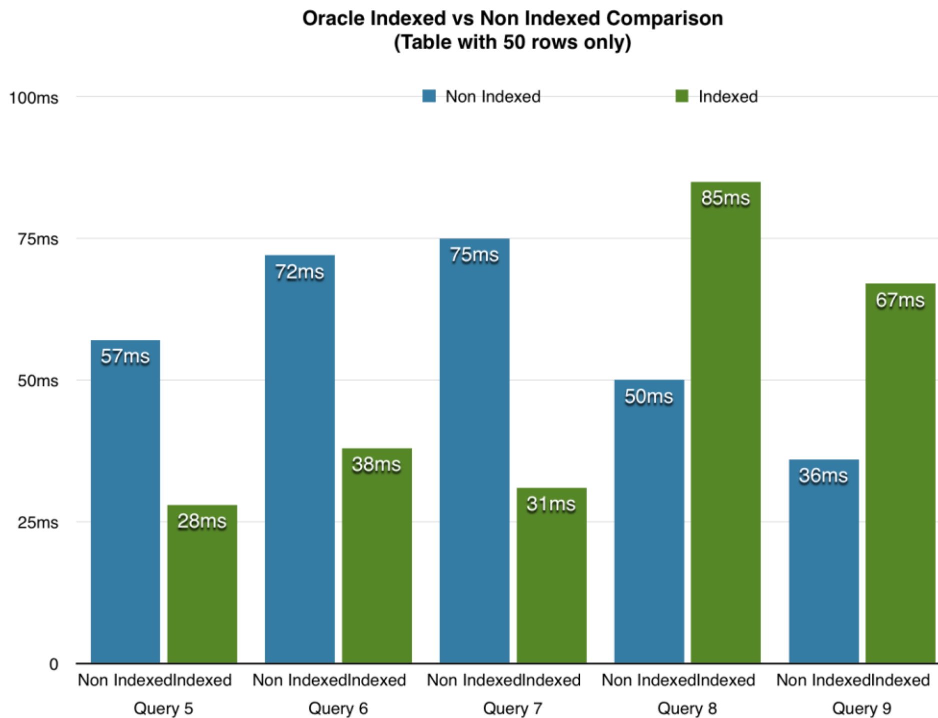
Next, hints were given to the optimizer to use the available index.

```
explain plan for SELECT /*+ index(test11 ind_lfn) */ M.*
  FROM test11 p,
       json_table(
        p.doc,
        '$'
        columns (
          wmaid varchar2(2000 char) path '$.wmaid',
          meta_data varchar2(2000 char) format json with wrapper path '$.meta_data',
          nested path '$.LFNArray[*]'
          columns (
            lfn varchar2(2000 char) path '$'
          )
        )
      ) M
WHERE lfn = '/store/mc/Run3212/file0.root';
```

Indexing on smaller tables

In a table of 50 rows, it was observed from the ***explain plan*** that Oracle Optimizer selected the index for calculating the results. Therefore, tests were performed on the same queries with and without indexes on this table.



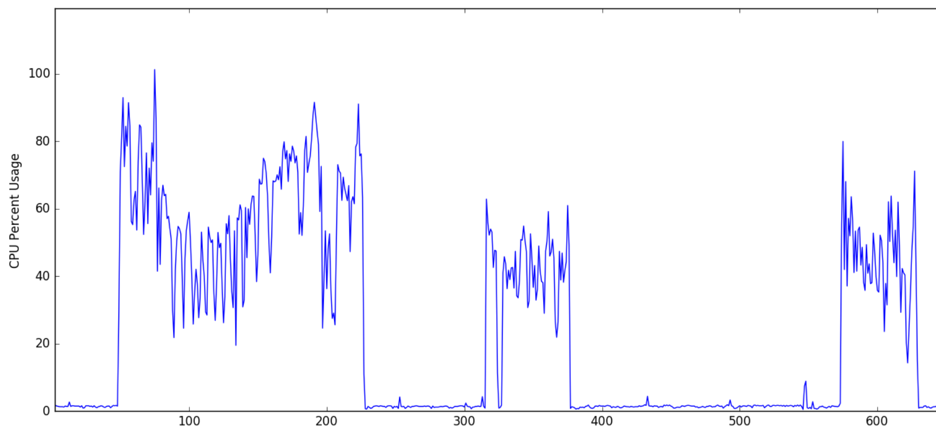


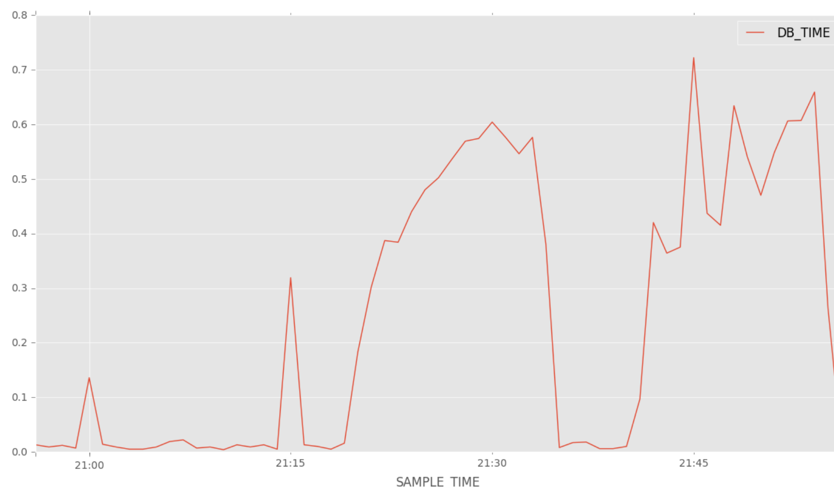
5.5 CPU Performance

To monitor the CPU Performance, the psutil library was used. The process id was noted down and passed to a python script that calculated the CPU Usage and stored it in a file after every time interval of 1second.

```
def get_statistics(pid):
    process = psutil.Process(pid)

    file_cpu = open('../Performance Statistics/cpu_stats.txt','w')
    file_memory = open('../Performance Statistics/mem_stats.txt','w')

    elapsed_time = 0
    while 1:
        cpu_usage = process.cpu_percent()
        memory_usage = (process.memory_info()[0])/(1024*1024)
        time.sleep(1)
```

To obtain the graph, pyplot, a module of matplotlib was used and it read the data from a text file. For Oracle, *pandas* was used and matplotlib displayed the graph.





6. Comparison

Both MongoDB and Oracle's JSON Library perform good in terms of read/writes and querying. A table shows below the difference between both the technologies.

MongoDB	Oracle JSON
Higher Data Size 1.3 GB for 1M documents	**Lower** Data Size 54 MB for 1M documents
223 MB Index Size	**1.14 GB** Index Size
Queries are **not** atomic	Queries **are** atomic
Lower Insert Time	**Higher** Insert Time
Indexing **works**	Indexing **not straightforward**
No such errors	**Internal errors** such as "No Data to be read from socket" occur frequently

7. Conclusion

Oracle's JSON Library looks promising and it was able to perform most of the tasks successfully. Insertion of numerous JSON documents into Oracle didn't seem to be a problem. Along with that, it was able to perform different kinds of queries ranging from searching a string to regular expressions. However, there were some issues with the usage of indexes in the queries.

MongoDB on the other hand performed well on both insertion of documents as well as running queries and creating indexes. It showed a better insert time than Oracle and ran queries faster. However, it was seen that there were some areas when the CPU Usage was really high and also, it used larger storage space to store the data.

8. Appendix

a) **Query** : Find specific string in PFNArray

MongoDB :

```
1.  cursor = db.production.find({'PFNArray':'root://test.ch/Run123/file0.root'})
```

Oracle :

```
1.  SELECT M.*
2.    FROM testDocument p,
3.         json_table(
4.          p.doc,
5.          '$'
6.          columns (
7.            wmaid varchar2(2000 char) path '$.wmaid',
8.            meta_data varchar2(2000 char) format json with wrapper path '$.meta_da
    ta',
9.            nested path '$.PFNArray[*]'
10.           columns (
11.             pfn varchar2(2000 char) path '$'
12.           )
13.         )
14.       ) M
15. WHERE pfn = 'root://test.ch/Run16/file0.root';
```

b) **Query** : Find Records for provided run number

MongoDB :

```
1.  cursor = db.production.find({'steps.output.runs.runNumber':2})
```

Oracle :

```
1.  SELECT M.*
2.    FROM testDocument p,
3.         json_table(
4.          p.doc,
5.          '$'
6.          columns (
7.            names varchar2(2000 char) format json with wrapper path '$.LFNArray',

8.            nested path '$.steps.output.runs[*]'
9.            columns (
10.             runNumber VARCHAR path '$.runNumber'
11.           )
12.         )
13.       ) M
14.   WHERE runNumber = 2;
```

c) **Query** : Find Records Based on provided site

MongoDB :

```
1.  cursor = db.production.find({'steps.site':'T2_US_FNAL_Disk'})
```

Oracle :

```
1.  SELECT M.*
2.    FROM testDocument p,
3.        json_table(
4.          p.doc,
5.          '$'
6.          columns (
7.              site varchar2 (2000 char) path '$.steps.site'
8.            )
9.        ) M
10.   WHERE site = 'T2_US_FNAL_Disk';
```

d) **Query** : Aggregate Data

MongoDB :

```
1.  cursor = db.production.aggregate([
2.                      { "$unwind" : "$steps"},
3.                      { "$group" : { "_id": None,
4.                                     "sum_totalMB": { "$sum" : "$steps.performance
    .storage.writeTotalMB"},
5.                                     "max_cp": { "$max" : "$steps.performance.cp.T
    otalJobCP" },
6.                                     "avg_eventTime": { "$avg" : "$steps.performan
    ce.cp.AvgEventTime"},
7.                                     "max_valueRss" : { "$max" : "$steps.performan
    ce.memory.PeakValueRss"}
8.                                   }
9.                      }
10.                 ])
```

Oracle :

```
1.  SELECT SUM(M.totalMB) totalMB_sum,
2.         MAX(M.totalCP) totalCP_max,
3.         AVG(M.avgEventTime) eventTime_avg,
4.         MAX(M.peakValueRss) Rss_max
5.    FROM testDocument p,
6.         json_table(
7.          p.doc,
8.          '$.steps.performance'
9.          columns (
10.           totalMB number path '$.storage.writeTotalMB',
11.           totalCP number path '$.cp.TotalJobCP',
12.           avgEventTime number path '$.cp.AvgEventTime',
13.           peakValueRss number path '$.memory.PeakValueRss'
14.         )
15.       ) M;
```

e) **Query** : Find Specific wmaid

MongoDB :

```
1. cursor = db.production.find({'wmaid':'88JEntUcP6G5rbCGudEO7rakfWjfg5rg'})
```

Oracle :

```
1. SELECT test.doc.wmaid FROM testDocument test WHERE test.doc.wmaid = '88JEntUcP6G
   5rbCGudEO7rakfWjfg5rg';
```

f) **Query** : Find Records based on provided PFN pattern

MongoDB :

```
1. cursor = db.production.find({'PFNArray':{'$regex':'^root://test.ch/Run214'}})
```

Oracle :

```
1. select t.doc.LFNArray , t.doc.PFNArray
2.   from testDocument t
3.     where json_exists(
4.                         t.doc,
5.                         '$?(@.PFNArray starts with $str)'
6.                         passing 'root://test.ch/Run214' as "str"
7.                         );
```

g) **Query** : Find records based on provided LFN pattern

MongoDB :

```
1. cursor = db.production.find({'LFNArray':{'$regex':'^/store/mc/Run727'}})
```

Oracle :

```
1. select t.doc.PFNArray
2.   from testDocument t
3.     where json_exists(
4.                         t.doc,
5.                         '$?(@.LFNArray starts with $str)'
6.                         passing '/store/mc/Run727' as "str"
7.                         );
```

h) **Query** : Logical Query Operators

MongoDB :

```
1. cursor = db.production.find({"$or":[
2.                            {"PFNArray": { "$regex" : "^root://test.ch/Run430/"} },
3.                            {"LFNArray": { "$regex" : "^/store/mc/Run121/"} }
4.                                       ]
5.                              })
```

Oracle :

```
1. select t.doc.LFNArray, t.doc.PFNArray
2.   from testDocument t
3.     where json_exists(
4.                         t.doc,
5.                         '$?(@.LFNArray starts with "/store/mc/Run1" ||
6.                             @.PFNArray starts with "root://test.ch/Run4")'
7.                         );
```

i) **Query** : Comparison Query Operators

MongoDB :

```
1. cursor = db.production.find({'steps.performance.storage.writeTotalMB':{'$gte': 2
   00, '$lte': 250}})
```

Oracle :

```
1. select test.doc.steps.performance.storage.writeTotalMB
2.   from testDocument test
3.     where json_exists(test.doc,
4.                         '$.steps.performance.storage?(@.writeTotalMB > 390
5.                                                        && @.writeTotalMB < 400)'
6.                        );
```

9. References

JSON in Oracle database :
https://docs.oracle.com/database/121/ADXDB/json.htm#ADXDB6246

MongoDB Documentation: https://docs.mongodb.com/

CMS MongoDB : https://www.mongodb.com/customers/cern-cms

CMS WMArchive Project : https://twiki.cern.ch/twiki/bin/view/ITSDC/WMArchive

WMArchive Presentation : https://goo.gl/I6P7bQ

cx_Oracle: http://www.oracle.com/technetwork/articles/dsl/prez-python-queries-101587.html

Wikipedia : http://www.wikipedia.com