www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 6 Issue 2 Feb. 2017, Page No. 20183-20191
Index Copernicus Value (2015): 58.10, DOI: 10.18535/ijecs/v6i2.03

Analysis and Evaluation of Techniques for Managing Unstructured and Semi-Structured Data in a MapReduce Platform

Dina Darwish
International Academy for Engineering and Media Science, Multimedia and Internet Department, Egypt
dina.g.darwish@gmail.com

Abstract: ***The increasing demand for large-size data mining and data analysis applications drives both industry and academia to create new types of highly scalable data-intensive computing platforms. MapReduce is one of the most popular platforms in which the dataflow is in the form of a directed acyclic graph of operators. This paper presents a modified version of the MapReduce framework that is developed to manage unstructured and semi-structured data. Since, almost most kinds of database systems are designed to manage well-structured data requiring users to design a schema before storing and querying data. However, there are significant amount of unstructured data and semi-structured data that cannot be effectively managed this way. In this paper, we develop the engineering principles and practices to manage unstructured and semi-structured data in a MapReduce platform. Having a single data platform for managing both well-structured data, unstructured and semi-structured data is beneficial to users; this approach reduces significantly integration, migration, development, maintenance, and operational issues. The Hadoop environment is used to write SQL/XML schemas first, then, all commands are translated to Hadoop as MapReduce jobs. The efficiency of using this method in MapReduce software is discussed and evaluated.***

Keywords: unstructured and semi-structured data, SQL/JSON schema, SQL/XML schema, Hadoop MapReduce framework.

1. Introduction

Having a huge role over the last 20 years in digital world, the software will remain a driving force for its continued enhancements and updates. The innovation is based on software as a key component [1]. It allows presenting different features and services with short turnaround times and high speed to market. As a result, software has become an essential factor in industry and a basis for innovation [2]. Designing a software is not a simple work, because it needs to be developed in close cooperation with R&D units and domains built in universities, companies, factories and others.

Software engineering research and innovation presents new ways, techniques, mechanisms, languages and tools which enhance software production and engineering in itself. Industry will remain skilled, capable and competitive in delivering software and software-based products to their customers and markets through continued software engineering research and innovation.

One important aspect of software engineering when speaking about Big Data is how to design the software systems. Implementing Big Data technologies to develop and create large scalable data systems makes a significant software architecture challenge for software architects. The challenge comes from the scale factor where software architects must deal with issues related to distributed systems. Problems like data replication, data consistency, temporary failures, communications latencies and concurrent processing require to be solved in the system design. These problems are amplified in Big Data systems, where these systems need to dynamically extend to make use of data distributed geographically.

Parallel database systems, which all share a common architectural design, have been commercially used for nearly two decades, and there are a lot of these systems in the marketplace, such as; Teradata, Microsoft SQL Server, Vertica, DB2 with the database partitioning feature, and Oracle. These systems are robust, as they use high performance computing platforms.

The increased focus on the 'schema first, data later' technique has prevented relational database systems (RDBMSs) from being the ideal platform for dealing with unstructured or semi-structured data. Instead, unstructured or semi-structured data support has been adopted in specialized database systems, but, some found that it is inadequate to support schema evolution [3] to manage data whose structure varies a lot over time.

For managing unstructured documents in database management systems (DBMSs), content management systems are frequently used to store documents such as files with text index providing keyword search [4, 5]. MarkLogic NoSQL system [6] is known for implementing XQuery as query language for managing document-oriented semi-structured data like XML. MongoDB [7] based NoSQL systems with JSON specific query language became a popular choice for managing JSON data as JSON became the data-centric semi-structured data format. Polygolt storage with NoSQL [8] and some other specialized NoSQL based database systems [9] are becoming known. In [10], a new technique that depends on the paradigm (data first, schema later/never) for managing unstructured and semi-structured data in relational database systems, has been explained in details to treat the problems that appeared during handling these kinds of data. Due to an explosion in the number of massive-scale data intensive applications both in industry and in the sciences, the need for highly parallel data processing platforms have appeared.

Today, NoSQL and MapReduce are used frequently compared to parallel data processing platforms, because they provide efficient storage, representation and query of Big Data. However, apart from large, long-standing batch jobs, many Big Data queries require only small, short and increasingly interactive jobs. MapReduce [11] is a famous framework used in programming commodity computer clusters to implement large-scale data processing in a single pass. A MapReduce cluster can expand to thousands of nodes in a fault-tolerant manner. Although parallel database systems [12] may provide these data analysis applications, they are expensive, difficult to administer, and lack fault-tolerance for long-running queries [13]. Hadoop [14], an open-source MapReduce implementation, has been used by Yahoo, Facebook, and other companies for large-scale data analysis. Programmers can realize their applications simply by using a map function and a reduce function inside the MapReduce framework to transform and aggregate their data, respectively. Many algorithms can be implemented using the MapReduce model, such as word counting, equi-join queries, and inverted list building [11]. MapReduce constitutes an attractive environment for developers to create new techniques for handling it due to previous advantages.

In this paper, the goal is to enable MapReduce to manage unstructured and semi-structured data along with the structured data and, thereby providing all the advanced data management services that have been designed over many years for dealing with it. The main contribution of this paper is a detailed analysis and evaluation of the techniques used to manage unstructured and semi-structured data, the issues of importance when applying these techniques, and the potential of handling unstructured, semi-structured data and structured data in a MapReduce platform are thoroughly analyzed and evaluated.

The Outline of the Paper is as follows: Section 2 presents a description for Hadoop and MapReduce platforms. Section 3 provides detailed analysis and evaluation of techniques for managing unstructured and semi-structured data in MapReduce platform. Section 4 presents and discusses the experimental results. Section 5 presents the conclusions and future work.

2. Description of Hadoop and MapReduce Platforms

2.1. HadoopDB architecture

2.1.1. HadoopDB's Components

HadoopDB, whose architecture is shown in Figure 1, expands the Hadoop framework and consists of the following four components [21]:

(a) Database Connector

The interface that connects between independent database systems residing on nodes in the cluster and Task Trackers is called the Database Connector. It expands Hadoop's Input Format class and constitutes a part of the Input Format Implementations library. Each MapReduce job feeds the connector with the SQL query and connection parameters, such as: which JDBC driver to use, query fetch size and other query tuning parameters. The Database Connector has been assigned the role of connecting to the database, processing the SQL query and providing results as key-value pairs. The Database Connector can create connection to any JDBC-compliant database that exists in the cluster. However, different databases need different read query optimizations.

(b) Catalog

The catalog keeps meta information about the databases. The meta information can be as follows: (a) connection parameters such as database location, driver class and credentials, (b) metadata such as data sets included in the cluster, copies locations, and data partitioning properties. The current implementation of the HadoopDB catalog enables saving its meta information in the form of an XML file in HDFS. This file can be reached by the Job Tracker and Task Trackers to acquire information needed to schedule tasks and execute data required by a query.

(c) Data Loader

The Data Loader is assigned the following tasks for (a) repartitioning data on a given partition key upon loading, (b) breaking single node data into multiple smaller partitions called chunks and (c) then loading the single-node databases with the chunks. The Data Loader has two main components: Global Hasher and Local Hasher. The Global Hasher processes a custom made MapReduce job over Hadoop that reads raw data files stored in HDFS and creates new partitions from them as many as the number of nodes in the cluster. The repartitioning job does not include the overhead of organizing in order typical MapReduce jobs. The Local Hasher role is to copy a partition from HDFS into the local file system of each node, and secondarily, to partition the file into smaller sized chunks depending on the maximum chunk size setting.

(d) SQL to MapReduce to SQL (SMS) Planner

HadoopDB presents a parallel database front-end to data analysts making them able to execute SQL queries. The SMS planner expands Hive [15]. Hive converts HiveQL, a variant of SQL, into MapReduce jobs that can be linked to tables stored as files in HDFS. The MapReduce jobs constitutes of lots of relational operators (such as filter, select (project), join, aggregation) that work as iterators: each operator sends a data tuple to the next operator after executing it. Since each table is saved as a separate file in HDFS.

2.1.2. Hadoop Implementation

In the middle of HadoopDB exists the Hadoop framework. Hadoop is composed of two layers [21]: (a) a data storage layer named the Hadoop Distributed File System (HDFS) and (b) a data processing layer named the MapReduce Framework. HDFS is a block-structured file system controlled by a central Name Node. Individual files are divided into blocks of the same size and distributed across multiple Data Nodes in the cluster. The Name Node work includes keeping metadata about the size and location of blocks and their copies. The MapReduce Framework has a simple master-slave architecture. The master works as a single Job Tracker and the slaves or worker nodes work as Task Trackers. The Job Tracker manages the runtime scheduling of MapReduce jobs and keeps information on each Task Tracker's load and available resources. Each job is divided into Map tasks depending on the number of data blocks that need processing, and Reduce tasks.

The Job Tracker assigns tasks to Task Trackers depending on their locations and load balancing. It reaches locality by matching a Task Tracker to Map tasks that execute data inside it. It performs load balancing by ensuring all available Task Trackers are assigned tasks. Task Trackers regularly inform the Job Tracker with their status through heartbeat messages. The Input Format library constitutes the interface between the storage and processing layers. Input Format implementations scan text/binary files or make connections to arbitrary data sources, and convert the data into key-value pairs that Map tasks can execute.

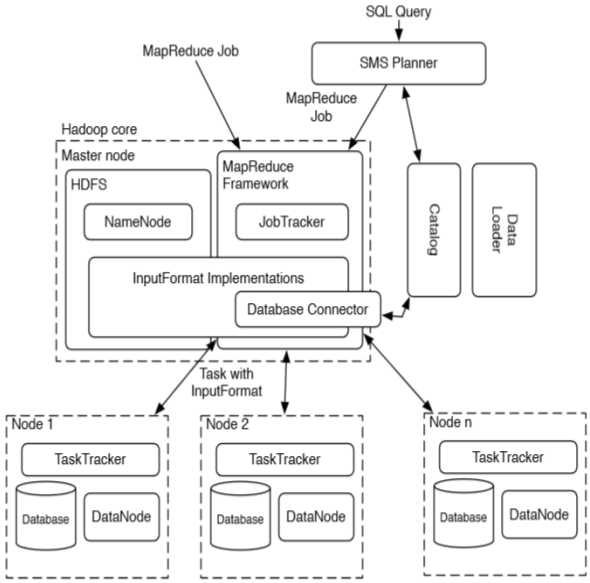

Figure 1: The architecture of HadoopDB

2.2. MapReduce framework

MapReduce was introduced by Dean et. al. in 2004 [11]. Shortly, MapReduce executes data distributed across many nodes in a shared-nothing cluster via three basic processes. First, a set of Map tasks are executed in a parallel way by each node in the cluster without having to communicate with other nodes. Next, data is divided into new partitions across all nodes of the cluster. Finally, a set of Reduce tasks are processed in a parallel way by each node on the partition receving it. This can be followed by a certain number of additional Map-repartition-Reduce cycles as needed. MapReduce does not provide a query execution plan that determines which nodes will execute which tasks in advance; instead, this can be determined during runtime. This gives MapReduce the ability to adjust to node failures and slow nodes by giving more tasks to faster nodes and redistributing tasks from failed nodes. MapReduce also examines the output of each Map task inside the local disk to decrease the quantityt of work that has to be redone after a failure. MapReduce best achieves the fault tolerance and ability to work in heterogeneous environment characteristics. It reaches fault tolerance by finding and redistributing Map tasks of failed nodes to other nodes in the cluster, specially nodes with copies of the input Map data. It has the ability to operate in a heterogeneous environment via redundant task execution. Tasks that takes a long time to finish compared to slow nodes has to be processed inside other nodes that have finished their assigned tasks. MapReduce contains a flexible query interface; Map and Reduce functions are considered arbitrary computations written in a general-purpose language. Therefore, each task can do anything on its input, just as long as its output follows the conventions determined by the model. In general, most MapReduce-based systems, such as Hadoop, which provides the systems-level details of the MapReduce platform, do not allow declarative SQL. Many of the performance enhancing tools that are implemented by database systems can not be realized if not obligating the user to first design and load data before execution. The fault tolerance and ability to operate in different environment characteristics of MapReduce could be integrated with the performance of parallel databases systems. Figure 2 [11] shows the MapReduce programming model, where a Hadoop application can handle Hadoop distributed file system (HDFS) in a cluster.

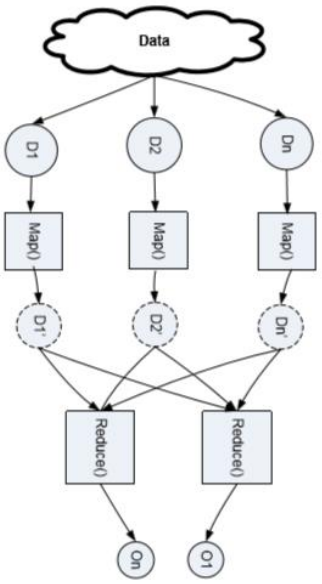

Figure 2: The MapReduce programming model

3. Approach for Managing Unstructured and Semi-Structured Data Using MapReduce

3.1 Managing unstructured and semi-structured data in a MapReduce platform

There is a huge need to reengineering Hadoop platform to handle new challenging data management requirements, such as unstructured or semi-structured data, that were not addressed in the original Hadoop MapReduce data management paradigm. It is better to create a single system which avoids the complexity of multiple architectures instead of having users to manage multiple systems. Management of unstructured and semi-structured data has challenged the fundamental assumption that a schema must exist to store,

index and query data first. then, there is a need to think out-of-the schema. Indeed, management of unstructured and semi-structured data requires thinking how to store, query and index data without making schema definition first. To achieve this goal, there is a work done to extend technology for managing user defined object types, functions and indexes. Implementing extensibility ideas leads us to the current engineering principles and practices for managing unstructured and semi-structured data in MapReduce. The idea in this paper is inspired by the research done on RDBMS [10]. In the following sections, a detailed analysis and description of storage, query, and update and index principles in a MapReduce platform is going to be explained. And, experimental results using different simulation methods such as HiveQL, JSON and XQuery were analyzed to discuss the efficiency of applying this idea.

3.1.1. Storage technique for management of unstructured and semi-structured data in a MapReduce environment

The design of most of the database platforms including MapReduce provides a clean separation between structure and data. In most database platforms, a schema has to be determined before data can be loaded. A collection of unstructured and semi-structured data, has a small number of common attributes accompanied by a large variety of non-common attributes, that are managed using JSON and XML documents, then, they are transformed into MapReduce jobs in a MapReduce platform. The structure cannot be separated easily from data content because the structure changes a lot from instance to instance. The instance schema is included in each unstructured or semi-structured data (UNSED) instance so that each UNSED instance is self-contained, and can be distributed to different layers. Schema based on data of a UNSED collection are not treated as central dictionary data but can be computed dynamically from all unstructured or semi-structured data (UNSED) instances saved in a UNSED collection.

Unlike the schema based on structured data, inside the schema based on UNSED data, there does not exit a set of finite size of dimension D such that every element of a set of data S can be expressed as a linear combination of elements from set D. The schema based on structured data may have bounded dimensions with unlimited number of elements as formal schema definition, while, schema based on unstructured and semi-structured data has unlimited dimensions.

All the dimensions are called schemas. Managing UNSED data cannot be done by separating the data from its schema. Each element in this case has to keep track of its dimensions and the corresponding value. Each element is represented by a vector of dimension and value (name value pair). Then, management of unstructured or semi-structured data requires store, query and index both schema and data together. Then, after implementing the storage technique using XML documents or JSON objects, these queries were transformed into Map and Reduce jobs in a MapReduce environment.

UNSED instances are stored in a UNSED collection using document-object-store model, where both structure and data are saved together for each UNSED instance, so each UNSED instance is self-descriptive and does not rely on a central schema. New structures can be added on a per-record basis without dealing with schema architecture. Adding a new domain UNSED is done by storing into existing SQL datatype, without the need to add a new SQL type and this enables the new UNSED domain to acquire full operational data capability support.

A data-guide can be derived from UNSED collections in the form of virtual tables, to know the complete structures of the data which helps to create queries over UNSED collection. UNSED management with data-guide supports the paradigm of "storage without schema but query with schema". For master-detail hierarchical structures existing in UNSED instances, a table architecture must be created. They can be built as secondary structures on top of the primary hierarchical UNSED storage.

Storage technique in the MapReduce depends on the following steps. UNSED instances containing both schema and data are stored in the UNSED collections in the data loader inside the MapReduce environment. Also, schemas created inside UNSED instances are saved in schema collections in the catalog inside the MapReduce environment. Then, all saved schemas are transformed using SMS planner into MapReduce jobs, that are executed inside the MapReduce platform.

Figure 3 shows the storage technique for management of unstructured and semi-structured data in a MapReduce environment.

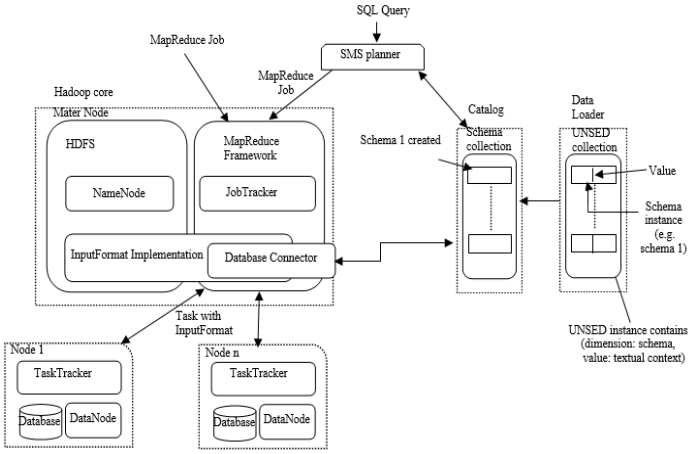

Figure 3: Storage technique for management of UNSED in a MapReduce environment

3.1.2. Query and Update technique for management of unstructured and semi-structured data in a MapReduce environment

A UNSED collection is saved as a table of UNSED instances. A UNSED instance itself is domain specific and typically owns its domain-specific query language. The domain-specific query language is XQuery for UNSED of XML documents. The domain-specific query language is the SQL/JSON path language for UNSED of JSON objects.

In the domain-specific query language, querying or updating UNSED can be performed by surfing through document-object structures, A UNSED instance is not distributed into tables since hierarchies in a UNSED are dynamic. Then, it is obvious to express hierarchical traversal of UNSED as path navigation with value predicate constructs in the UNSED domain language. There must exist ability of performing full context aware text search, UNSED instances may be document centric with mixture of textual content and structures. There is a

DOI: 10.18535/ijecs/v6i2.03

considerable quantity of full text content in UNSED that can be subject to full text query. Full text search can be embedded within a context identified by path navigation into the UNSED instance. Also, querying or updating can be done by projecting, transforming object component and developing new document or object, results of path navigational queries can be partitions or called fragments of UNSED. The new UNSED fragments can be created by extracting components of existing UNSED and combining them through creation and transformation.

While a UNSED domain-specific query and update language constitutes as an intra-document query language, SQL can be implemented as an inter-document query language. The query of UNSED depends on positioning SQL as a set-oriented language to present access to a set of UNSED instances by integrating the set based algebra supported by SQL. By positioning SQL as a Set Query Language, SQL achieves the needed constructs to express set algebra operators, such as selection, projection, join, group by, aggregation, union, intersection and difference among UNSED instances. Query and update techniques in the MapReduce is performed as follows.

Some UNSED instances saved in the UNSED collections in the data loader inside the MapReduce environment are retrieved according to the required query or update commands. The retrieved UNSED instances are gathered in a fragment of UNSED collection created after the query or update command is executed. After that, schemas contained in the extracted fragment of UNSED collection are saved in schema collections in the catalog inside the MapReduce environment as virtual schemas. Then, all virtual schemas saved inside the schema collections are converted using SMS planner into MapReduce jobs, that are executed inside the MapReduce platform.

Figure 4 shows query and update techniques for management of unstructured and semi-structured data in a MapReduce environment.

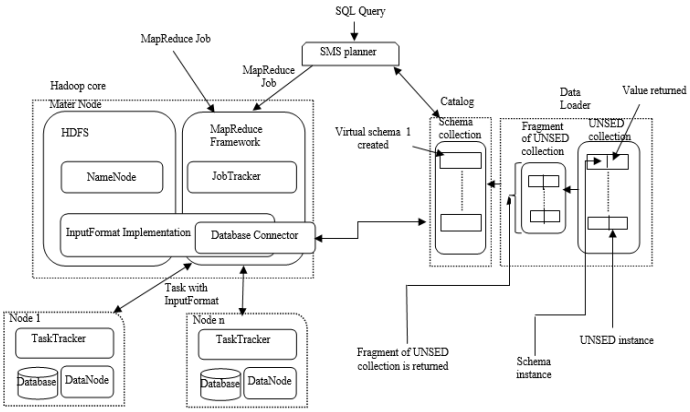

Figure 4: Query and update techniques for management of UNSED in a MapReduce environment

3.1.3. Indexing technique for management of unstructured and semi-structured data in a MapReduce environment

UNSED indexing must be able to achieve performance for both predefined query access pattern and ad-hoc query access pattern. In a pre-defined query access pattern in the context of UNSED, a projection out of UNSED in the form of a set of scalar value projections or in the form of UNSED table for a set of views can be determined, because users are aware of the partial schema within UNSED derived from data-guide. This is known as 'data first, schema later as index' indexing approach.

Meanwhile, in the ad-hoc query access pattern in the context of UNSED, users do not own any prior knowledge of the UNSED, then a UNSED search index is required to make efficient evaluation of the existence of UNSED using ad-hoc query search. This is known as 'data first, schema never' search index approach, which is similar to full text index search index.

The columnar index embeds efficient range queries over the columnar projection of UNSED values and returns a set of DOCIDs, each of which is an ordinal number that defines a row of the base document-object-store table including UNSED that satisfy the range query. The original UNSED can be provided from the primary document-object-store table using the DOCID returned by the columnar index. To support range queries over multiple scalar values extended from UNSED via multiple UNSED values, multiple columnar indexes, each of which links to a UNSED value, can be implemented.

The idea of table index [16] is used, to efficiently execute UNSED table queries. Table index can be presented in two physical forms in dealing with UNSED. In a classical row store, table index can internally keep master-detail relational tables to carry the results calculated by evaluation of UNSED tables. The master-detail table is connected by internally generated primary foreign key so that the column values in the master table are not repeatedly saved in detail tables. Besides, the tuple-store model can be used to determine columnar index in the form of UNSED values and UNSED tables to index UNSED.

To manage an ad-hoc query, a search index over a UNSED table without having users to determine what path structures or values require to be indexed is created. A search index indexes everything in a UNSED collection. Search indices can be created depending on classical inverted index that indexes all keywords in a document to present ad-hoc keyword search capability [17]. This achieves the basic full text search capability over document centric XML documents or JSON objects. Unlike classical inverted indices that index only keywords in a document, a generalized inverted index is expanded to index hierarchical path structures inside UNSED to provide path-aware full text search scalar range value search workload queries.

Each UNSED document in a UNSED collection indexed by the search index is determined by an ordinal number as a DOCID. The DOCIDs of all documents containing the keyword are saved in an organized way using delta-compression inside a posting list, so that efficient pre-sorted merge join is accomplished on the posting lists to efficiently manage multi-keywords searches and phrase searches linked by AND, OR and NOT Boolean predicates [18]. Classical inverted indices can be expanded to support efficient processing of path query and path aware full text search [19] that is a common query for XML full text search.

Each distinct XML tag is indexed and saved as an entry in inverted index with the posting list saving not only the DOCIDs of documents including the XML tags, but also the range of tag open and close positions inside the document. In the same method, an inverted index can be expanded to index JSON objects saved in JSON collection table [20]. Like XML tree nodes, JSON objects and arrays create nested hierarchical relationship that can be indexed using their positions inside the JSON object.

Indexing technique in the MapReduce can be accomplished according to the following steps. Each UNSED instance containing both schema and data has a DOCID, as mentioned before, these UNSED instances are stored in the UNSED collections in the data loader inside the MapReduce environment. A fragment of UNSED collection is created according to the indexing required. This fragment contains the UNSED instances in need. Then, schemas residing inside UNSED instances are stored with their DOCIDs in schema collections in the catalog inside the MapReduce environment. Then, all saved schemas in the catalog are transformed using SMS planner into MapReduce jobs, that are executed inside the MapReduce platform. Figure 5 shows indexing technique for management of unstructured and semi-structured data in a MapReduce environment.

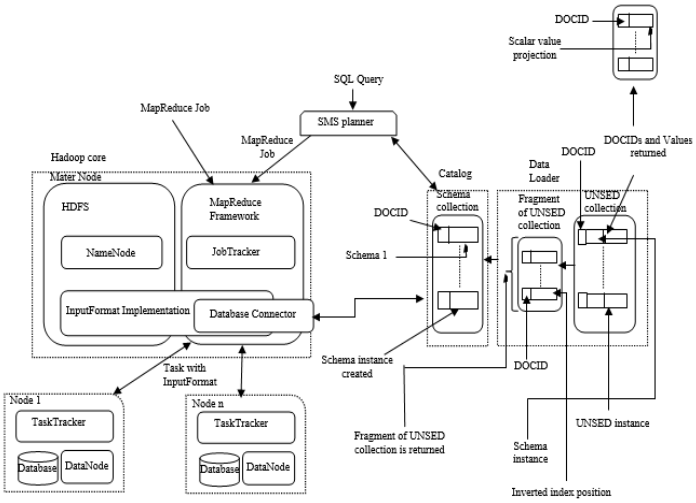

Figure 5: Indexing technique for management of UNSED in a MapReduce environment

4. Experimental Results

In this section, the execution times of these three methods were analyzed and evaluated. The TCP-H benchmark dataset was used to compute the results using HiveQL, JSON, XQuery.

The following table (table 1) shows the total execution times of running TCP-H dataset using five proposed queries with HiveQL, JSON, XQuery to compare, analyze and evaluate the fastest execution method, and the differences of execution times between the three methods. Let's consider the following queries:

Query 1:

Using HiveQL:

```
Select n_nationkey, n_name, n_regionkey, n_comment from
nation;
```

Using JSON:

```
CREATE TABLE json_nation (json string);
LOAD DATA INPATH '/user/datafiles/nation.json' OVERWRITE
INTO TABLE json_nation;
CREATE TABLE jt_nation AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valueregnat from json_nation;
SELECT get_json_object(jt_nation.valueregnat,'$.id') AS id,
get_json_object(jt_nation.valueregnat,'$.N_NATIONKEY') AS
n_nationkey,
get_json_object(jt_nation.valueregnat,'$.N_NAME') AS n_name,
get_json_object(jt_nation.valueregnat,'$.N_REGIONKEY') AS
n_regionkey,
get_json_object(jt_nation.valueregnat,'$.N_COMMENT') AS
n_comment FROM jt_nation;
```

Using XQuery:

```
xquery version "1.0";
for $i in 0 to 25
   let $y := doc("nation.xml")/nations/id[$i]
     return $y/(n_nationkey,n_name,n_regionkey,n_comment)
```

Query 2:

Using HiveQL:

```
select n.n_nationkey, n.n_name, r.r_regionkey , r.r_name
from region r join nation n on
n.n_regionkey = r.r_regionkey;
```

Using JSON:

```
CREATE TABLE json_region ( json string );
LOAD DATA INPATH '/user/datafiles/region.json' OVERWRITE
INTO TABLE json_region;
CREATE TABLE jt_region AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valueregreg from json_region;
SELECT
get_json_object(jt_nation.valueregnat,'$.N_NATIONKEY') AS
n_nationkey,
get_json_object(jt_nation.valueregnat,'$.N_NAME') AS n_name,
get_json_object(jt_region.valueregreg,'$.R_REGIONKEY') AS
r_regionkey,
get_json_object(jt_region.valueregreg,'$.R_NAME') AS r_name
FROM jt_region join jt_nation on
get_json_object(jt_nation.valueregnat, '$.N_REGIONKEY') =
get_json_object(jt_region.valueregreg, '$.R_REGIONKEY');
```

Using XQuery:

```
xquery version "1.0";
let $regions := doc('region.xml')
   let $y := doc('nation.xml')
   for $i in 0 to 25
      for $j in 0 to 5
       where  $y/nations/id[$i]/n_regionkey =
$regions/regions/id[$j]/r_regionkey
          return <result>{$y/nations/id[$i]/n_nationkey}
and {$y/nations/id[$i]/n_name} and
{$regions/regions/id[$j]/r_regionkey} and
{$regions/regions/id[$j]/r_name}</result>
```

Query 3:

Using HiveQL:

```
select
  s_suppkey, s_name, s_address, s_acctbal, s_comment
from nation n join supplier s on
  s.s_nationkey = n.n_nationkey and n.n_name = 'GERMANY';
```

Using JSON:

```
CREATE TABLE json_nation (json string);
CREATE TABLE json_supplier (json string);
LOAD DATA INPATH '/user/datafiles/nation.json' OVERWRITE
INTO TABLE json_nation;
LOAD DATA INPATH '/user/datafiles/supplier_v.json'
OVERWRITE INTO TABLE json_supplier;
CREATE TABLE jt_nation AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valueregnat from json_nation;
CREATE TABLE jt_supplier AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valuesupp from json_supplier;
SELECT get_json_object(jt_supplier.valuesupp,'$.S_SUPPKEY'),
get_json_object(jt_supplier.valuesupp,'$.S_NAME'),
get_json_object(jt_supplier.valuesupp,'$.S_ADDRESS'),
get_json_object(jt_supplier.valuesupp,'$.S_ACCTBAL'),
get_json_object(jt_supplier.valuesupp,'$.S_COMMENT')
FROM jt_nation join jt_supplier on
get_json_object(jt_nation.valueregnat, '$.N_NATIONKEY') =
get_json_object(jt_supplier.valuesupp, '$.S_NATIONKEY') and
```

```
get_json_object(jt_nation.valueregnat, '$.N_NAME') =
'GERMANY';
```

Using XQUERY:

```
xquery version "1.0";
let $suppliers := doc('supplier.xml')
   let $y := doc('nation.xml')
   for $i in $suppliers/suppliers
       for $j in $y/nations
        where  $y/nations/id[$j]/n_nationkey =
$suppliers/suppliers/id[$i]/s_nationkey and
               $y/nations/id[$j]/n_name = 'GERMANY'
          return
<result>{$suppliers/suppliers/id[$i]/s_suppkey} and
{$suppliers/suppliers/id[$i]/s_name} and
{$suppliers/suppliers/id[$i]/s_address} and
{$suppliers/suppliers/id[$i]/s_acctbal} and
{$suppliers/suppliers/id[$i]/s_comment}</result>
```

Query 4:

Using HiveQL:

```
select s_suppkey from supplier
where not s_comment like '%Customer%Complaints%';
```

Using JSON:

```
CREATE TABLE json_supplier (json string);
LOAD DATA INPATH  '/user/datafiles/supplier_v.json'
OVERWRITE INTO TABLE json_supplier;
CREATE TABLE jt_supplier AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valuesupp from json_supplier;
SELECT get_json_object(jt_nation.valuesupp,'$.S_SUPPKEY') AS
ps_suppkey FROM jt_supplier
where not get_json_object(jt_supplier.valuesupp,
'$.S_COMMENT') like '%Customer%Complaints%';
```

Using XQUERY:

```
xquery version "1.0";
let $suppliers := doc('supplier.xml')
    for $i in $suppliers/suppliers
      where  not $suppliers/suppliers/id[$i]/s_comment like
'%Customer%Complaints%'
          return
<result>{$suppliers/suppliers/id[$i]/s_suppkey}</result>
```

Query 5:

Using HiveQL:

```
select s_suppkey, s_acctbal, s_name, n_name, s_address,
s_phone, s_comment
from nation n join region r on
    n.n_regionkey = r.r_regionkey and r.r_name = 'EUROPE'
  join supplier s on   s.s_nationkey = n.n_nationkey
  where s_acctbal > 50.0;
```

Using JSON:

```
CREATE TABLE json_nation (json string);
CREATE TABLE json_region (json string);
CREATE TABLE json_supplier (json string);
LOAD DATA INPATH  '/user/datafiles/nation.json' OVERWRITE
INTO TABLE json_nation;
LOAD DATA INPATH  '/user/datafiles/region_v.json' OVERWRITE
INTO TABLE json_region;
LOAD DATA INPATH  '/user/datafiles/supplier_v.json'
OVERWRITE INTO TABLE json_supplier;
CREATE TABLE jt_nation AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valueregnat from json_nation;
CREATE TABLE jt_region AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valueregreg from json_region;
CREATE TABLE jt_supplier AS
select
regexp_replace(regexp_replace(json,'\\}\\,\\{','\\}\\\n\\{')
,'\\[|\\]','') as valuesupp from json_supplier;
SELECT get_json_object(jt_supplier.valuesupp,'$.S_SUPPKEY'),
get_json_object(jt_supplier.valuesupp,'$.S_ACCTBAL'),
get_json_object(jt_supplier.valuesupp,'$.S_NAME'),
get_json_object(jt_nation.valueregnat,'$.N_NAME'),
get_json_object(jt_supplier.valuesupp,'$.S_ADDRESS'),
get_json_object(jt_supplier.valuesupp,'$.S_PHONE'),
get_json_object(jt_supplier.valuesupp,'$.S_COMMENT')
FROM jt_nation join jt_region on
get_json_object(jt_nation.valueregnat, '$.N_REGIONKEY') =
get_json_object(jt_region.valueregreg, '$.R_REGIONKEY')
and get_json_object(jt_region.valueregreg, '$.R_NAME') =
'EUROPE'
join jt_supplier on
get_json_object(jt_nation.valueregnat, '$.N_NATIONKEY') =
get_json_object(jt_supplier.valuesupp, '$.S_NATIONKEY')
where get_json_object(jt_supplier.valuesupp,'$.S_ACCTBAL') >
50.0;
```

Using XQUERY:

```
xquery version "1.0";
let $suppliers := doc('supplier.xml')
   let $y := doc('nation.xml')
      let $z = doc('region.xml')
   for $i in $suppliers/suppliers
      for $j in $y/nations
         for $m in $z/regions
       where  $y/nations/id[$j]/n_nationkey =
$suppliers/suppliers/id[$i]/s_nationkey and
             $y/nations/id[$j]/n_regionkey =
$z/regions/id[$m]/r_regionkey and
$suppliers/suppliers/id[$i]/s_acctbal > 50.0
          return
<result>{$suppliers/suppliers/id[$i]/s_suppkey} and
{$suppliers/suppliers/id[$i]/s_acctbal} and
{$suppliers/suppliers/id[$i]/s_name} and
{$y/nations/id[$j]/n_name} and
{$suppliers/suppliers/id[$i]/s_address} and
{$suppliers/suppliers/id[$i]/s_phone} and
{$suppliers/suppliers/id[$i]/s_comment}</result>
```

Using the three techniques, execution times are computed for the five proposed queries. For query 1, HiveQL was executed in 4 seconds and 530 milliseconds, JSON was executed in 7 seconds 880 milliseconds, which is a little bit longer than HiveQL, and XQuery was executed in 1 seconds and 500 milliseconds, which is the shortest time. In query 2, HiveQL took 15 seconds and 350 milliseconds to finish, JSON took a very close time from HiveQL with fractions in milliseconds, which is 15 seconds and 490 milliseconds, then XQuery took the shortest time with 2 seconds 400 milliseconds. For query 3, JSON took a little bit longer with 34 seconds and 640 milliseconds, followed by, HiveQL with 8 seconds and 320 milliseconds, and XQuery took the shortest time of 3 seconds and 60 milliseconds. For query 4, JSON took the longer time with 15 seconds and 590 milliseconds, HiveQL took a little bit shorter time of 12 seconds and 750 milliseconds, then, XQuery took the shortest time at 2 seconds and 980 milliseconds. For query 5, JSON took a very close time from HiveQL, with 7 seconds and 980 milliseconds, and 7 seconds and 330 milliseconds respectively, then, XQuery took a shorter time with 2 seconds and 560 milliseconds. After analyzing the results obtained in table 1 for all the five queries, XQuery showed the better execution time, followed by HiveQL, then, finally, JSON query. But, in general, the difference in execution times between HiveQL and JSON is small. And, this showed that JSON and XQuery are preferred to be used in managing unstructured, semi-structured, as well as structured data, due to their good execution times reasonably compared to HiveQL.

Table 1: Execution times using different methods

	HiveQL	*JSON*	*XQuery*
Query 1	4 sec 530 msec	7 sec 880 msec	1 sec 500 msec
Query 2	15 sec 350 msec	15 sec 490 msec	2 sec 400 msec
Query 3	8 sec 320 msec	34 sec 640 msec	3 sec 60 msec
Query 4	12 sec 750 msec	15 sec 590 msec	2 sec 980 msec
Query 5	7 sec 330 msec	7 sec 980 msec	2 sec 560 msec

Figure 6, Figure 7, Figure 8, Figure 9 and Figure 10 show execution times of queries 1, 2, 3, 4 and 5. These figures are derived from table 1. The execution times of the three methods are represented in seconds. In each figure, the execution time is

represented by the x-axis, and the HiveQL, JSON and XQuery methods used in execution are represented by the y-axis. These figures compare the execution times of the three methods.

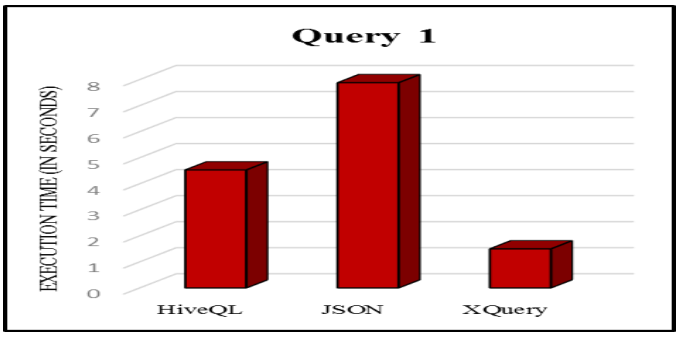

Figure 6: Execution times of query 1

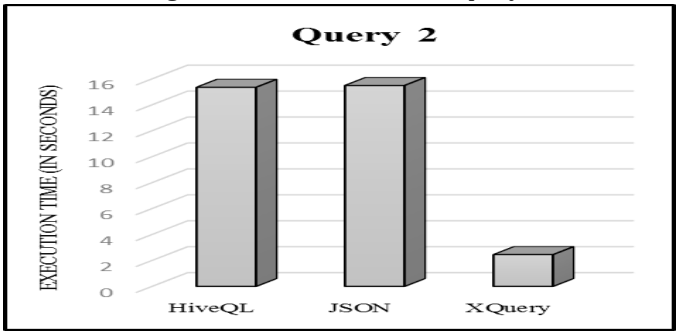

Figure 7: Execution times of query 2

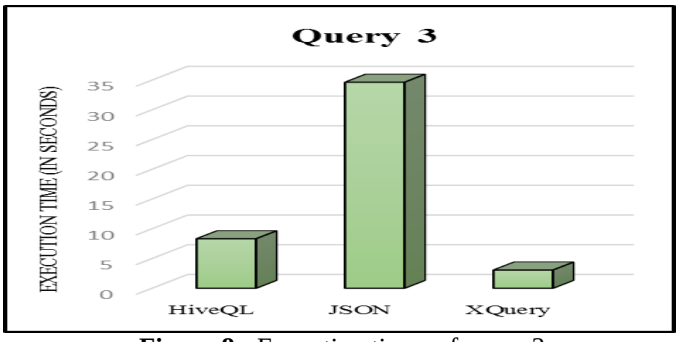

Figure 8: Execution times of query 3

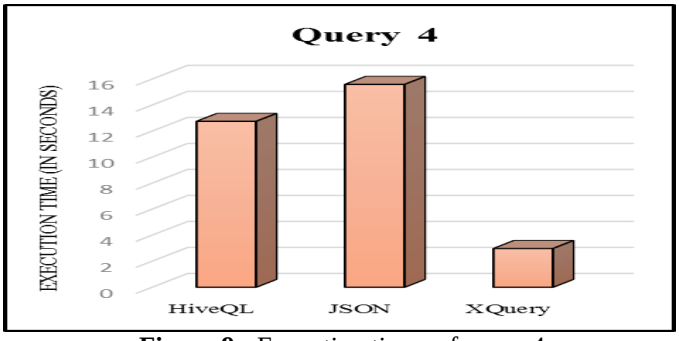

Figure 9: Execution times of query 4

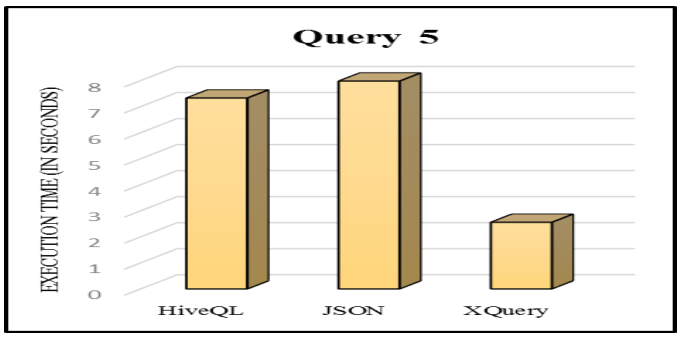

Figure 10: Execution times of query 5

5. Conclusions and Future Work

A model for managing unstructured and semi-structure is proposed in this paper using JSON and XQuery. It was concluded from simulation that JSON and XQuery provided good execution times compared to HiveQL. It is better to create a single interface to manage structured, unstructured and semi-structured data. In the future work, there is a need to make a unified model that all kinds of environments and platforms and that can be used to manage all types of data; such as structured, unstructured and semi-structured data.

References

[1] A. Arora, L. G. Branstetter, M. Drev, "Going Soft: How the Rise of Software-Based Innovation Led to the Decline of Japan's IT Industry and the Resurgence of Silicon Valley," MIT Press Journals, July 2013.

[2] ISTAG, "Software Technologies: The Missing Key Enabling Technology - Toward a Strategic Agenda for Software Technologies in Europe," July 2012. [Online]. Available: http://cordis.europa.eu/fp7/ict/docs/istag-soft-tech-wgreport2012.pdf.

[3] Mohan C., "History repeats itself: sensible and Nonsense SQL aspects of the NoSQL hoopla," EDBT, 2013.

[4] G. Salton, M. McGill., "Introduction to Modern Information Retrieval," McGraw-Hill, New York, 1983.

[5] J. Zobel, A. Moffat, "Inverted files for text search engines. ACM Computing," Surveys, Volume 38 Issue 2, 2006.

[6] MarkLogic. [Online]. Available: http://www.marklogic.com/

[7] MongoDB. [Online]. Available: http://www.mongodb.org/

[8] P. J. Sadalage, M. Fowler, "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence," August 18, 2012.

[9] R. Cattell, "Scalable SQL and NoSQL data stores," SIGMOD Record, Volume 39 Issue 4, pp. 12-27, 2010.

[10] Zhen Hua Liu, Dieter Gawlick, "Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL," Oracle Corporation, USA.

[11] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: Simplified data processing on large clusters," in OSDI, pp. 137–150, 2004.

[12] David J. DeWitt, Jim Gray, "Parallel database systems: The future of high performance database systems," Commun. ACM, Volume 35 Issue 6, pp. 85–98, 1992.

[13] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Michael Stonebraker, "A comparison of approaches to large-scale data analysis," in SIGMOD Conference, pp. 165–178, 2009.

[14] Hadoop. [Online]. Available: http://hadoop.apache.org/. [Accessed: July 7, 2010.]

[15] Facebook, "Hive." [Online]. Available: http:// issues.apache.org/jira/browse/HADOOP-3601.
[16] Z. H. Liu, M. Krishnaprasad, H. J. Chang, V. Arora, "XMLTABLE Index - An Efficient Way of Indexing and Querying XML Property Data," ICDE, 2007.
[17] J. Zobel, A. Moffat, "Inverted files for text search engines," ACM Computing Surveys, Volume 38 Issue 2, 2006.
[18] I. Rae, A. Halverson, J. Naughton, "In-RDBMS Inverted Indexes revisited," ICDE, pp. 352-363, 2014.
[19] Z.H. Liu, Y. Lu, H. Chang, "Efficient Support of XQuery Full Text in SQL/XML Enabled RDBMS," ICDE, 2014.
[20] Z. H. Liu, B. Christoph Hammerschmidt, D. McMahon, "JSON data management: supporting schema-less development in RDBMS," SIGMOD Conference, pp. 1247-1258, 2014.
[21] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, Alexander Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads," VLDB 2009, August 24-28 2009.

Author Profile

Dina Darwish received the B.Sc. in 2004 and the M.Sc. in 2006 with honors degrees from Arab Academy for Science and Technology, Computer Engineering Department, Egypt. She received the Ph.D. degree from Cairo University, Egypt, in 2009. Her main interests include communications systems, wireless and computer networks, internet-of-things technology, database, software engineering and multimedia systems.

She became an Assistant Professor at Multimedia and Internet department in 2009, International Academy of Engineering and Media Science, Egypt. She is now an Associate Professor and chair of the Multimedia and Internet department, International Academy of Engineering and Media science, Egypt, since 2015.