Performance Evaluation of Analytical Queries on a Stand-alone and Sharded Document Store

Aarthi Raghavendra and Karen C. Davis
Electrical Engineering and Computing Systems Department
University of Cincinnati
Cincinnati, OH USA 45221-0030
1-513-556-2214
raghavai@mail.uc.edu, karen.davis@uc.edu

ABSTRACT
Numerous organizations perform data analytics using relational databases by executing data mining queries. These queries include complex joins and aggregate functions. However, due to an explosion of data in terms of volume, variety, veracity, and velocity known as Big Data [1], many organizations such as Foursquare, Adobe, and Bosch have migrated to NoSQL databases [2] such as MongoDB [3] and Cassandra [4]. We investigate the performance impact of analytical queries on a NoSQL document store. We benchmark the performance of MongoDB [3], a cross-platform document-oriented database, in a stand-alone environment and a sharded environment. The TPC-DS benchmark [5] is used to generate data of different scales and selected data mining queries are executed in both the environments. Our experimental results show that along with choosing the environment, data modeling in MongoDB also has a significant impact on query execution times. Analytical query performance is best when data is stored in a denormalized fashion. When the data is sharded, due to multiple query predicates in an analytical query, aggregating data from a few or all nodes proves to be an expensive process and hence performs poorly when compared to the alternative process of executing the same in a stand-alone environment.

CCS Concepts
• Information systems→Database management system engines • Database query processing→query optimization.

Keywords
document databases; MongoDB; sharded query performance; analytical queries.

1. INTRODUCTION
Relational database systems have been the foundation for enterprise data management for over 30 years. Many organizations use a relational platform to perform data analysis by running data mining queries against a database. With an estimated growth in enterprise data to 35ZB by 2020 [6] along with growing user loads, organizations are adopting newer technologies such as NoSQL databases to store data. Among the types of NoSQL databases [7] (key-value store, column-oriented, document store, and graph), we have chosen MongoDB, a cross-platform document-oriented database against which we execute data mining queries. It provides features such as aggregation, secondary indexing, sharding and replication. Parker et al. [8] compare the runtime performance of MongoDB with SQL Server for a modest-sized database (3 tables, at most 12,416 tuples total) and conclude that the former performs equally well or better than SQL Server except when aggregation is utilized. However, the impact of data modeling and deployment environments for aggregation operations were not explored in detail.

In this paper, we investigate the performance of complex data mining queries against datasets of different sizes. We use a stand-alone and distributed data organization known as sharding [9]. In a sharded database, data is split into chunks and distributed across the cluster nodes. A query run against such a system can target either one, a few, or all the nodes and the result from each of the nodes is aggregated and displayed to the user.

In Section 2, we outline features of MongoDB such as data modeling, indexing, sharding, and aggregation. In Section 3, we discuss the TPC-DS benchmarking standard that is used to generate datasets of varying sizes as well as the criteria used to select data mining queries for our study. In Section 4, we describe the hardware and software configurations of the systems used to conduct the experiments. In Section 5, algorithms for migrating relational data and translating SQL queries to MongoDB are presented. Section 6 outlines the experimental procedures implemented on the stand-alone and sharded environments and discusses our findings. We conclude with a synopsis of the contributions and future work in Section 7.

2. MongoDB/BIG DATA BENCHMARKING
We give an overview of MongoDB concepts and terms, followed by a discussion of relevant benchmarking efforts to date.

2.1 MongoDB
MongoDB is a cross-platform document-oriented database classified under the aegis of the NoSQL databases. It is coined from the term *huMONGOus* for its support of big data management. Key features of MongoDB include high performance, high availability, and automatic scaling [10]. It is schema-less or has a flexible schema. Unlike SQL where the structure of the data is defined prior to loading into the database, MongoDB does not enforce any rigid structure. The flexible structure is achieved by storing data in BSON format [11], which is a binary-encoded serialization of JSON-like [13] key and value pairs.

A document is composed of key-value pairs, and is the basic unit of data in MongoDB. The value of these fields can be another document, array, and array of documents. A group of documents is called a collection. Since documents do not dictate a specific

format, a collection can have documents with each having a varying number of fields and types of values, thereby giving it a flexible schema.

An example of a document is given in Figure 1. Every document in a collection has a unique *_id* field that acts as the primary key. Unless explicitly specified by the application, MongoDB uses a special 12-byte BSON type, called *ObjectId*, as the default value for the *_id* field. The *ObjectId* is generated from the timestamp, machine ID, process ID, and a process-local incremental counter that guarantees uniqueness [10].

```
{
   _id: ObjectId (“5480abb8986c9d3197f6682c”),
   customer_id: 23,
   customer_address: {
         apartment_number: 26,
         street_name: “Whitfield”,
         state: “CA”,
         country: “United States”
   }
   customer_name: “Earl Garrison”,
   birth_date: “9/25/1979”,
   email_id: earl.garrison@G3sM4P.com
}
```

Figure 1: **Document Structure**

2.1.1 Data Modeling

An application that uses a relational database management system models data by declaring a table’s structure and its relationships to other tables prior to inserting data into it. Similarly, data modeling in MongoDB focuses on the document structure and relationships between data. Since MongoDB does not support joins, the two concepts that facilitate data modeling are embedding and referencing [10].

We illustrate data modeling techniques in MongoDB through an example. Consider two entities, *Book* and *Publisher* and a one-to-many relationship between them.

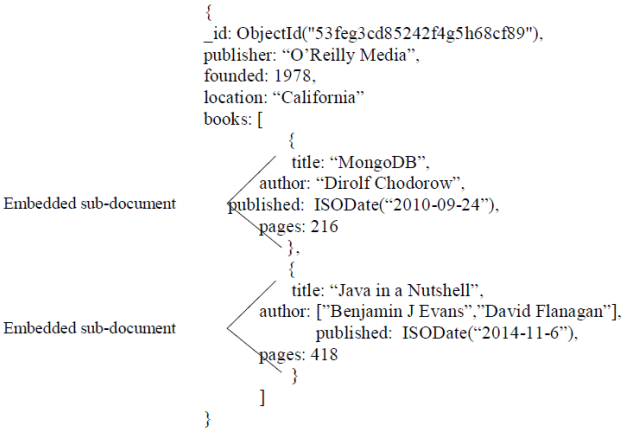

Figure 2: Embedded Data Model

Figure 2 illustrates the embedded data model design for the one-to-many relationship between *Publisher* and *Book* entities. Embedding represents a relationship by encapsulating related information in a single document or structure. It depicts a *contains* relationship between entities since related pieces of information is stored in the same database record. The one-to-many relationship between a *Publisher* and a *Book* can be modeled by embedding the book data entities in the publisher data. This provides good performance for read operations as related data can be retrieved in a single database operation. For example, an application can retrieve complete publisher information in one query and new books published by the publisher can be added as embedded sub-documents to the books array. This ensures no repetition of the publisher details per book, thereby reducing redundant data. However, if the size of the document exceeds the 16 MB limit, data has to be split and stored in separate documents. In such cases, the referenced data model can be adopted.

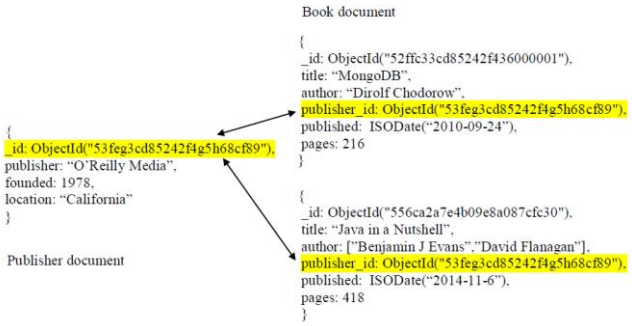

Figure 3: Referenced Data Model

Figure 3 illustrates the referenced data model design for the one-to-many relationship between *Publisher* and *Book* entities. References represent relationships between data by associating or referencing one document to another. This is achieved by storing the *_id* of one document as the value for another field in the other document. The one-to-many relationship between *Publisher* and *Book* can be modeled by keeping the publisher and book information in two separate collections and the relationship can be enforced by storing the *Publisher* reference inside the *Book* document. In doing so, a query to retrieve complete publisher information would have to make multiple requests to the server as follow-up queries are necessary to resolve the references. However, it is a suitable choice to model entities that are independent of each other. Also, if two or more entities are related but complex, then the complexity can be reduced by breaking down the data into multiple documents. Table 1 provides a comparison between the two data models.

Table 1: Embedded and Referenced Data Model Comparison

	Embedded Data Model	**Referenced Data Model**
Normalization	Denormalized	Normalized
Definition	All related data is encapsulated in a single document resulting in fewer queries to complete operations.	Related data is stored in separate collections and relationship is enforced by referencing one document to another.
When to use?	To model contains and one-to-many relationships between entities.	To model many-to-many relationships and for large hierarchical data sets.
Read Operation	Retrieve data in a single database operation.	Multiple requests to server to resolve references.
Write Operation	Atomic as single write operation can write or update the document.	Since data is being split, multiple write operations are required that are not atomic collectively.
Data Growth	Updating or inserting fields can increase the document size resulting in relocation on the disk.	Updating and inserting fields is done in separate collections.

For fast and efficient data access, indexes can be created to locate data quickly and perform random lookups.

2.1.2 Indexing

An index is a special on-disk data structure that allows the database application to retrieve data faster by not scanning the entire collection. It uses keys built from one or more fields in a document. MongoDB implements indexing by storing the keys in a B-Tree data structure which helps in finding rows associated with the keys quickly and efficiently. Indexes in MongoDB are defined at the collection level on any field or sub-field of the document. MongoDB supports 7 different types of indexes [10]. In our work,

we use the *default_id* and *compound* indexes. All collections are indexed on the *_id* field by default. A compound index is created on multiple fields of a document with a specific sort order for each field. If a collection has a compound index on *PersonID* and *Age*, the index sorts first by *PersonID* and then within each *PersonID* value, sorts by *Age*. Therefore, the order of the fields in the compound index should be declared appropriately based on the application needs.

2.1.3 Sharding

A database system deployed on a single server can experience heavy loading due to high query rates or large datasets. When the demand on the server spikes drastically, alternate means should be identified to keep the system online and robust. This scaling issue is addressed in database systems by implementing vertical scaling or horizontal scaling [15]. MongoDB implements horizontal scaling, or sharding [10]. It is the process of splitting data into chunks and distributing the chunks across multiple servers, known as shards. The data is partitioned at a collection level and stored on multiple shards such that all shards put together make up a single logical database.

A sharded cluster in MongoDB [10] has 3 components:

1. A *shard* is either a single *mongod* instance or a *replica set* [10] that stores data. The mongod is a daemon process that starts the MongoDB server and handles data requests and other background operations. Replica set is a feature of MongoDB that ensure redundancy by storing the same data on multiple servers.
2. A *config server* is a mongod instance that stores the metadata of the cluster. It maintains a mapping of the chunks to the shards.
3. A *query router* is a *mongos* instance that is responsible for directing read and write operations from the client application layer to the specific shard or shards. The mongos is a routing service to which the application queries are directed internally, which then uses the metadata information stored in the config server to locate the target shard or shards and consolidates all the information returned from the various shards before displaying it to the user.

While deploying the sharded cluster for our research, we encountered issues that affect the application and cluster performance. Most issues are caused by the number of instances of each of the components deployed in the cluster. We discuss the issues faced and methods adopted to avoid them.

In a sharded environment the number of instances of each of the components determines the robustness of the cluster. For read intensive applications, having multiple query routers helps balance the application needs rather than having a single query router that can be easily overloaded due to high frequency of read operations. Based on the cluster and application needs, query routers can be added to the cluster on the fly by establishing connections to the config servers and the shards.

The number of config servers and shards is of greater importance since they perform all the application critical operations. Deploying a cluster with multiple config servers enables data accessibility and avoids a single point of failure. Similar to config servers, the number of shards can also cause potential problems if they are not aligned with the application needs. For write intensive applications, shards can exceed their capacity and be exhausted quickly if data is continually written to it. Therefore the capacity of a shard should be decided before deploying the cluster based on the amount of the data to be stored on them. If the number of shards are too few, data resides on just a few shards leading to exhaustion problems. Having more shards reduces the amount of data on each shard and resources such as RAM and CPU cannot be used effectively. The number of shards in a cluster can be calculated based on the following factors [14].

1. The sum of disk storage space across shards should be greater than the required storage size. For example, if the application data is 1.5TB and the disk storage available per server is 256GB, then the number of shards needed can be calculated as 1.5TB/256GB ~ 6 shards.
2. The sum of RAM across shards should be greater than the working set of the sharded cluster. The working set is the segment of client data that is accessed most frequently by the application. For read intensive applications, storing the entire working set in the RAM results in faster read operations. If the working set memory requirement is more than the available RAM, the operating system needs to perform frequent IO operations to the disk to retrieve the information, thereby drastically slowing down the system. The working set size is the size of the frequently accessed collections and their indexes. For a working set of 200GB and server RAM of 64GB, the number of shards can be calculated as 200GB/64GB ~ 4 shards.

Other factors include disk throughput and operations per second. We calculate the number of shards needed in our cluster based on disk storage and RAM. Since the data load is a write intensive process, each server needs to have a sufficient amount of disk space to the store the data that is continually written. In doing so the server resources such as CPU, memory, and disk are utilized effectively without being overloaded. Also, since we focus on analytical query performance, read operations should be optimized to achieve best results. For fast read operations, all the collections and indexes related to the query should reside in the RAM to avoid disk IO operations. For this purpose, along with disk storage, we also take the server RAM into consideration for calculating the number of shards.

Data distribution effects the application read and write performance. If a considerable amount of data resides on a single shard, it can lead to a server crash or latency issues. Similarly, if too little data resides on each shard, the server resources are not fully utilized. In MongoDB, distribution of data across multiple cluster members is determined by the shard key. A shard key [10] is either an indexed field or an indexed compound field that is present in all documents of a collection. MongoDB uses the shard key to divide a collection into small non-overlapping ranges called chunks and the default chunk size is 64 MB. MongoDB uses range-based partitioning and hash-based partitioning for dividing shard key values into chunks.

If the shard key is a numeric value, MongoDB can use range-based partitioning [10], where documents with nearby shard key values reside in the same chunk and therefore on the same shard. This distributes data evenly with an overhead for efficient range queries.

MongoDB outlines the following strategies for selecting a shard key:

1. High cardinality: The cardinality of a shard key refers to the number of different values associated to it. A shard key with high cardinality has low probability of creating jumbo chunks.
2. Compound shard key: If a collection does not have a field which can serve as an optimal shard key, additional fields can be added to produce a more ideal key.

3. Hashed shard key: For a shard key with high cardinality, a hashed index on the field can ensure even distribution of data across the shards.

We utilize high cardinality and compound shard keys in our implementation.

2.2 Benchmarking Big Data

Among the many definitions for big data, we adopt the definition given by Dumbill: "Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the strictures of your database architectures." [1] Big data is typically characterized by 4V properties (i.e., volume, velocity, variety, and veracity [15]). Therefore in order to benchmark big data, a standard should be chosen that satisfies the 4V properties at least partially, if not completely. The synthetic data generator should meet the following criteria.

1. *Volume* refers to the ability to generate data of various scaling factors as inputs of typical workloads. The volume of data generated can range from GBs to PBs.
2. *Velocity* refers to data generation rates.
3. *Variety* refers to the support for generating diverse data types, which include structured data, unstructured data, and semi-structured data.
4. *Veracity*, with respect to benchmarking, is the ability to keep the synthetic data generated aligned with the characteristics of real world data [15]. The credibility of the benchmark is dependent on how well the raw data features are preserved in the generated data.

Han et al. [15] compare big data benchmarks such as TPC-DS [16] BigBench [17], and Hibench [18] in terms of data generation and benchmarking techniques. We use this paper as a reference to choose our benchmarking standard for generating data sets of various scale factors and data mining queries of varying complexities. We briefly discuss the state-of-the-art which includes terms introduced in relation to each of the 4V properties in order to compare and categorize the existing big data benchmarks. Table 3 gives basic definitions of the terminology.

Table 2: Terms Used to Categorize Big Data Benchmarks based on 4V Properties [15]

Property	Terms	Definition
Volume	Scalable	Generate data sets of variable sizes or scale factors
	Partially scalable	Generate fixed-size data as inputs
Velocity	Semi-controllable	Data generation rate is controlled, however data updating frequency is not considered
	Uncontrollable	Both data generation rate and updating frequency are not considered
Variety	Structured data	Tables
	Unstructured data	Texts, graphs and videos
	Semi-structured data	Web logs and resumes
Veracity	Partially considered	Data is generated by using both traditional synthetic distributions such as a Gaussian distribution and a realistic distribution obtained from real world data
	Un-considered	Data is randomly generated using statistical distributions

We conduct a performance evaluation of MongoDB by executing analytical queries on datasets of two different sizes. Therefore, we need a benchmark that can generate scalable datasets, and real world data in order to achieve a realistic result. MongoDB is an appropriate choice of database for unstructured data rather than tabular data. However, we are interested in the performance of MongoDB when tabular data is denormalized and modeled in a way that better suits MongoDB; we compare the performance of analytical queries against a normalized and denormalized data model. Therefore, among the big data benchmarks [15], we choose a benchmark that satisfies a scalable volume, semi-controllable velocity, structured variety, and partially considers veracity. The 3 benchmarks that satisfy our needs are TPC-DS [16], BigBench [17] and Bigdatabench [19]. Since we are studying the performance evaluation of analytical queries, we need the benchmark to be able to generate data that supports joins between entities and queries containing varying aggregate functions. BigBench benchmark provides a limited number of data mining queries and Bigdatabench provides a dataset consisting of only two tables, whereas TPC-DS provides a dataset of 24 tables and a query set of 100 queries, most of which support aggregate functions. Therefore, we choose TPC-DS as our big data benchmark

3. TPC-DS AND QUERY SELECTION

The underlying business model of the TPC-DS schema is a retail product supplier that follows a snowflake schema [16]. It is a database architecture where a central table called fact table is linked to multiple other tables called dimension tables. A fact table typically has two types of columns, the foreign key columns and measures columns. The foreign key columns reference the primary key of the dimension tables, and the measures columns hold data used for calculations and analysis.

Table 3: Query Features

Features/Queries	Query 7	Query 21	Query 46	Query 50
Number of tables	5	4	6	5
Number of aggregation functions	4	2	2	5
Number of group by/order by clauses	1	1	1	1
Number of conditional constructs	0	3	0	5
Number of correlated subquery(s)	0	0	1	0

Table 4: Table Details for Datasets 1GB and 5GB

Table	Number of Records	
	1GB	5GB
Call_center	6	14
Catalog_page	11,718	11,718
Catalog_returns	144,067	720,174
Catalog_sales	1,441,548	7,199,490
customer	100,000	27,7000
customer_address	50,000	138,000
customer_demographics	1,920,800	1,920,800
date_dim	73,049	73,049
household_demographics	7,200	7,200
Income_band	20	20
inventory	11,745,000	49,329,000
item	18,000	54,000
promotion	300	388
Reason	35	39
Ship_mode	20	20
store	12	52
store_returns	287,514	1,437,911
store_sales	2,880,404	14,400,052
Time_dim	86,400	86,400
warehouse	5	7
Web_page	60	122
Web_returns	71,763	359,991
Web_sales	719,384	3,599,503
Web_site	30	34

The TPC-DS benchmark [5] has a total of 7 fact tables and 17 dimension tables. Among the 24 tables, the representative queries we selected utilize 3 fact tables (*Store_Sales*, *Store_Returns*, and

Inventory) and a total of 9 dimension tables. Among the 4 query classes supported by TPC-DS we choose the data mining class. Among the 23 queries available in that class, we select 4 queries which meet three or more of these criteria: (1) join of 4 or more tables, (2) aggregation functions such as *sum()* and *avg()*, (3) *group by* and *order by* clauses, (4) conditional constructs such as *case*, and (5) correlated subquery using the *from* clause.

We select 4 queries that satisfy the criteria: Query 7, Query 21, Query 46, and Query 50. Table 3 summarizes the criteria met by each query. Table 4 lists the number of records in the tables for datasets of sizes 1GB and 5GB.

4. EXPERIMENTAL PLATFORM

For setting up the stand-alone and sharded environments, we used the Amazon Web Services (AWS), a cloud-computing platform that provides on-demand delivery of IT resources [20]. We rented virtual computers through the Amazon Elastic Compute Cloud (EC2) web service for application deployment and experimental set-up. We boot the Red Hat Enterprise Linux AMI (Amazon Machine Image) to create our virtual machines [20]. AWS provides the capability of starting, stopping, and terminating instances as needed, whereby active servers are charged by the hour.

TPC-DS is chosen as our benchmark for generating data and analytical queries. We use datasets of sizes 1GB and 5GB for conducting our research. However, the 1GB and 5GB text data when migrated to MongoDB increases to 9.94GB and 41.93GB respectively, an increase by a factor of nearly nine compared to the original dataset size. Therefore, we need machine(s) that can accommodate datasets with a minimum size of 10GB. Hence, an EC2 instance is chosen such that the RAM is greater than the working set, the portion of data that is accessed often by the application server [10]. A RAM that fits all the indexes and working set ensures faster processing there by reducing random disk IO.

The MongoDB stand-alone environment uses the m4.4xlarge[1] instance that is capable of storing both the 9.94GB and 41.93GB datasets. The MongoDB sharded environment is a 5 node cluster where every machine/instance has the same configuration. For application deployment on the MongoDB cluster we use the t2.large[1] instance for the 9.94GB dataset and the m4.xlarge[1] instance for the 41.93GB dataset.

4.1 Hardware/Software Configuration

This section discusses the hardware configurations of all the AWS machines utilized for the deployment of the stand-alone and sharded environments. Table 5 illustrates the machine configurations for the MongoDB sharded and stand-alone environments.

Table 5: Machine Hardware Configurations

	9.94GB sharded environment	41.93GB sharded environment	stand-alone environment
Number of machines	5	5	1
Instance Type	t2.large	m4.xlarge	m4.4xlarge
vCPUs	2	4	16
RAM	8 GB	16GB	64GB
Storage	EBS-Only	EBS-Only	EBS-Only
Processor	Intel Xenon Processor	2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors	2.4 GHz Intel Xeon® E5-2676 v3 (Haswell) processors

Two different MongoDB sharded environments are created for the 9.94GB and 41.93GB datasets. Each of the sharded environments have 5 machines. Since each sharded environment supports a dataset of specific size, the machine configurations for both the environments differ. For example, the 41.93GB sharded environment has more powerful machines than the 9.94GB sharded environment since it has more data. Only one stand-alone system is setup for both the 9.94GB and 41.93GB datasets. All the AWS machines have the same software configuration: Red Hat Enterprise Linux 7.1 and MongoDB version 3.0.2.

4.2 MongoDB Cluster Architecture

A MongoDB sharded system consists of 3 components: shards, config servers, and query routers. We determine the number of instances of each component by taking the 9.94GB dataset as an example. The MongoDB sharded cluster test architecture [10] is used as a reference for creating our sharded system. The test architecture uses 1 config server, 1 query router, and 1 or more shards based on the application needs. Our MongoDB sharded system has 1 config server and 1 query router similar to the test architecture. However, the number of shards is decided taking into consideration the data to be stored on it, which in this case is 9.94 GB.

We use the disk storage and RAM as factors for deciding the number of shards in the cluster. Among the two factors, RAM is given priority as it reduces random disk IO thereby improving read performance. Therefore, a system that is capable of accommodating data, indexes, and other running applications in the RAM is chosen. Among the available AWS machines, those with a RAM storage of either 4GB or 8GB best suit our application needs. A machine with less RAM would require deploying more systems in the cluster, increasing operational costs and a machine with RAM higher than 8GB would make sharding insignificant for the 9.94GB dataset.

The RAM consumption of the operating system and other applications typically does not exceed 2GB. If an AWS machine with 4GB RAM is chosen, only 2GB space would be available for storing data and indexes, hence requiring 5 machines (9.94GB/2GB). On the other hand, if an AWS machine with 8GB RAM is chosen, 6GB space would be available for storing data and indexes, thus requiring 2 machines (9.94GB/6GB). However, we use 3 machines with 8GB RAM as shards to accommodate not only the data but also indexes and the intermediate and final query collections. Therefore, the MongoDB sharded system deployed on AWS has 3 shards, 1 config server, and 1 query router.

Figure 4 illustrates the organization of the sharded system. Each node in the sharded system is named after their functionality namely, Shard1, Shard2, Shard3, ConfigServer, and AppServer/ QueryRouter. Since each node has a specific functionality, the processes running on them differ. The 3 shards are responsible for storing data and therefore run the mongod instance. The config server stores the metadata of the cluster and runs a mongod instance. The query router is a mongos instance, a routing service to which the application queries are directed to. It internally makes use of the config server metadata to direct queries to the appropriate shards. The application server and query router are deployed on the same AWS machine because the query router is a lightweight process and does not utilize much of the server resources. The two segmented lines represent the actual flow path of query or any related read operation and the continuous line indicates the observed flow path.

[1] AWS machine nomenclature

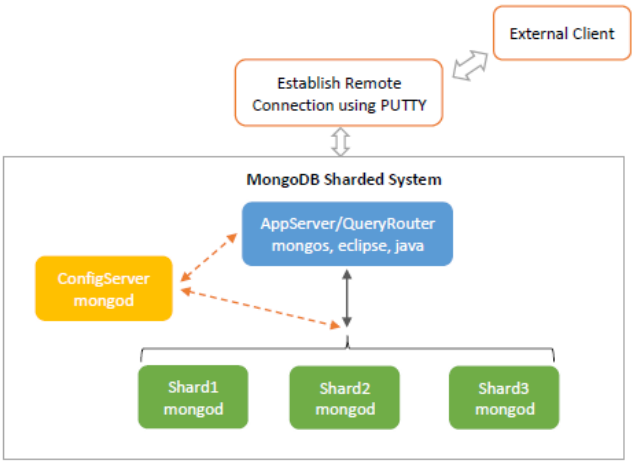

Figure 4: MongoDB Cluster Architecture

5. PERFORMANCE STUDY

In this section, we describe the algorithms developed for data loading and translation of a SQL query to a Mongo query using Java as the programming language[2] and we describe the experimental setup. Table 6 illustrates the different experimental setups based on the choice of dataset sizes, data models, and deployment environments. The experiments are conducted by taking into consideration the two dataset sizes, two data models, and two deployment environments.

Table 6: Experiments

Dataset Size	Data Model	Deployment Environment	Experiment Number
9.94GB	Normalized	Sharded system	Experiment 1
		Stand-alone system	Experiment 2
	Denormalized	Stand-alone system	Experiment 3
41.93GB	Normalized	Sharded system	Experiment 4
		Stand-alone system	Experiment 5
	Denormalized	Stand-alone system	Experiment 6

5.1 Migrating Data into MongoDB

We develop an algorithm to migrate the TPC-DS generated data files into MongoDB in the JSON format. The same algorithm is used to load datasets of sizes 1GB and 5GB, which translate to 9.94GB and 41.93GB, respectively, when loaded into MongoDB. The increase in size is due to the data storage in JSON format. Each document has key-value pairs where the key corresponds to the table column. Each key is repeated in every document in the collection, hence drastically increasing the size of the dataset. The following points summarize the flow of data starting with its creation using the TPC-DS benchmark up to its migration into MongoDB:

1. TPC-DS generates a .dat file for each table. The column values are delimited by the '|' operator.
2. The databases in MongoDB for 1GB and 5GB TPC-DS datasets are called *Dataset_1GB* and *Dataset_5GB*, respectively. Steps 3 and 4 correspond to the *Dataset_1GB* database, however they work analogously for the *Dataset_5GB* database.
3. Each table in the TPC-DS schema corresponds to a separate collection in the *Dataset_1GB* database. Hence, *Dataset_1GB* contains a total of 24 collections, representing the 7 fact tables and 17 dimension tables. The records in each table correspond to the documents in the respective collections.
4. Since data in MongoDB in stored in the JSON format, the column names of the table correspond to the keys in the JSON document and the column values of the table correspond to the respective key values in the document.

The pseudocode of the data migration algorithm is illustrated in Figure 5. It takes a data file as input and generates the MongoDB collection as output. The algorithm makes use of the HashMap data structure in Java [21], a tabular structure that maps keys to values. It uses hashing to generate a hashcode for every key, which determines the object location in the table.

Algorithm: ***SQL to MongoDB Data Migration***

Input: data file: *T*
Output: MongoDB collection
1: BEGIN
2: Create a new collection *collection*
3: Create a HashMap $H<k,v> = \{(k, v) \mid 0 \le k \le n \wedge v$=column name$\}$ where n = number of table columns
4: for each line *l* in *T* till *EOF* do
5: Create a new MongoDB document $document_{(k, v)}$ *where k = key and v = value*
6: Split *l* on the '|' delimiter and store returned array of strings in *A*.
7: for element A_i in array *A* do
8: $document_k = H_v$ where $H_k = i$
9: $document_v = A_i$
10: **end for**
11: Insert $document_{(k, v)}$ in *collection*
12: **end for**
13: END

Figure 5: Data Migration Algorithm

5.2 Translating SQL Queries to MongoDB

The SQL queries in our study differ from one another in aspects such as numbers of joined tables, aggregate functions, where clauses, from clauses, group by and order by clauses, and correlated and uncorrelated subqueries. Manually translating each of these queries into a Mongo query can be tedious and error-prone. We create algorithms for translating an analytical SQL query to its equivalent Mongo query. These algorithms focus on executing the selected queries that support aggregate functions, where clauses, from clauses, group by, and order by clauses.

Performance of queries or any read operation is greatly dependent on the data model that supports the application needs. In a relational database system, data models are broadly classified into normalized and denormalized data models. A normalized data model has a higher number of foreign keys and hence a higher number of joins. On the other hand, in a denormalized data model, data is consolidated into one or a few tables, resulting in fewer foreign keys and hence fewer joins. We are interested in the query performance on MongoDB when run against both the normalized and denormalized data models.

In order to execute analytical queries against a denormalized model, the collections in the MongoDB database should be denormalized. During data load the TPC-DS data files are migrated into individual collections, therefore on initial load the data is completely normalized. We develop an algorithm for creating a denormalized collection from a set of given fact collection and dimension collections. In this approach, all the dimension collections are joined to the fact collection based on their foreign key relationships. In MongoDB, joining a dimension collection to a fact collection is equivalent to embedding the dimension collection documents in the fact collection. To understand the structure of a denormalized collection in MongoDB, consider the

[2] code is available at https://github.com/raghavai/Performance-Evaluation-of-Analytical-Queries-on-a-Stand-alone-and-Sharded-Document-Store

store_sales fact collection connected to multiple dimension collections such as *time_dim*, *store*, and *item*. The foreign keys such as *ss_sold_date_sk*, *ss_item_sk*, and *ss_cdemo_sk* are replaced by the actual document in the document collection.

Figure 6 illustrates pseudocode for the algorithm to create a denormalized fact collection. It takes in a fact collection and a set of dimension collections as input and outputs the denormalized fact collection. An *EmbedDocuments* method, shown in Figure 7, is called on every dimension collection, which embeds the dimension collection documents in the fact collection.

Algorithm: ***Create Denormalized Collection***

Input: Fact Collection and Dimension Collection(s)
Output: Denormalized Fact Collection
1: BEGIN
2: Retrieve fact collection *F* and dimension collections $D_1, D_2, D_3, \ldots, D_n$
3: **for** each dimension collection D_i do
4: ***EmbedDocuments*** (F, D_i)
5: **end for**
6: END

Figure 6: Denormalized Collection Creation Algorithm

Algorithm: ***EmbedDocuments (F, D)***

Input: Fact Collection *F* and Dimension Collection *D*
Output: Embedded Fact Collection
1: BEGIN
2: Create an empty HashMap *H<k,v>*
3: Retrieve all documents in *D* using *find()* and store returned cursor in *cursor*
4: **while** *cursor.hasNext()* do
5: Create a copy called *doc* of *cursor.next()*
6: Remove *_id* field in *doc*
7: Insert into $H = \{(k, v) \mid k$ = primary key of $D \wedge v = doc\}$
8: **end while**
9: **for** each entry *E* in *H.entrySet()* do
10: Update F = {*(query, update, upsert, multi)* |
query = where primary key of $D = E.getKey() \wedge$
update = set primary key of $D = E.getValue() \wedge$
upsert = false $\wedge$
multi = true}
11: **end for**
12: END

Figure 7: Embedding Documents Algorithm

5.3 Query Translation Algorithm for Normalized Data Model

An analytical query against a normalized data model typically queries data from one or more tables. In SQL, this is accomplished through join operations. The SQL query optimizer plays a vital role in identifying an optimal query plan with low cost. For example, if a query contains join operations, where clauses, and a group by clause, the query optimizer decides the best possible execution order of the operations that yields a query plan that has a low cost and execution time.

In MongoDB, query optimization is limited to the usage of indexes and efficient read operations. Join operations cannot be optimized as MongoDB does not support joins. To overcome this hurdle, we develop an algorithm that simulates join operations and executes queries on the fly. All the queries used here implement the select-from-where template. Therefore, the algorithm is optimized for queries that follow this template. The algorithm does not take into consideration the details of the query predicates, aggregation operations, and sequence of join operations, but only follows a predetermined order of execution:

1. Query all the dimension collections based on their respective where clauses.
2. Perform a semi-join of the fact collection with the filtered dimension collection documents, i.e., obtain only those fact collection documents whose foreign keys are present in the filtered dimension collection documents. For example, a semi-join of the *store_sales* and *customer_address* collections would result in a collection containing only those *store_sales* documents whose foreign key is referenced in the *customer_address* collection. Store the semi-joined fact collection in an intermediate collection.
3. Embed dimension collection documents in the intermediate collection documents. To improve performance embed only those dimension collection documents whose attributes are used in query aggregation.
4. Perform aggregation operations against the embedded intermediate collection and store the final query results in an output collection.

We illustrate the pseudocode for the query translation algorithm against a normalized data model in Figure 8. It takes the fact collection(s) and dimension collections related to the query as input and outputs the final query collection. In addition to the final query collection, an intermediate collection is created which holds the semi-joined fact collection documents. The algorithm uses the *ArrayList* data structure in java [23], a dynamic array that can grow and shrink as needed. It is used in the process of performing a semi-join on the fact collection. The *EmbedDocuments* method illustrated in Figure 7 is called on the intermediate collection and a dimension collection during the embedding process. To improve performance, only those dimension collections whose attributes are used in query aggregation are embedded. After the embedding process, the MongoDB aggregation framework is used to execute the aggregation operations in the query.

Algorithm: ***Query Translation Algorithm against Normalized Data Model***

Input: Fact Collection and Dimension Collection(s) associated with query *Q*
Output: Query output collection
1: BEGIN
2: Retrieve fact collection *F* and dimension collections $D_1, D_2, D_3, \ldots, D_n$
3: Create an empty ArrayList for each dimension collection $A_1, A_2, A_3, \ldots, A_n$
4: **for** each dimension collection D_i do
5: Filter D_i based on its respective query *where* conditions and store the primary key of filtered documents in A_i
6: **end for**
7: Perform semi-join on *F* using MongoDB *$in* operator on the respective arrays. Store the filtered documents in an intermediate collection *I*.
8: **for** each D_i whose attributes are used in the query aggregation do
9: ***EmbedDocuments*** (I, D_i)
10: **end for**
11: Perform aggregate operations, group by, and order by clauses on the embedded intermediate collection *I*
12: Store the aggregated documents in a new collection
13: END

Figure 8: Query Translation Algorithm for the Normalized Data Model

6. EXPERIMENT RESULTS

Table 7 summarizes the data load times for each table for both the dataset sizes. We also show query runtimes executed on each of the experimental setups. Table 8 illustrates the selectivity of the queries, i.e., the proportion of data retrieved for both the 9.94GB and 41.93GB datasets. Table 9 summarizes the query execution runtimes for each of the experiments. Every query is run 5 times on each experimental setup. For each run data is cached in the memory. Among the 5 runtimes we obtain, Table 9 presents the best results. Hours are denoted by *h*, minutes by *m*, and seconds by *s*.

We analyze the results of the data load times for the 1GB and 5GB datasets below.

1. For tables having the same number of records in both the datasets, the data load times are nearly identical. This can be observed for tables *catalog_page, customer_demographics, date_dim, household_demographics, income_band, ship_mode,* and *time_dim*.
2. For tables with an unequal number of records in both the datasets, the ratio of the number of records is nearly identical to the ratio of their load times. This can be observed for tables *call_center, catalog_returns, catalog_sales, customer, customer_address, inventory, item, promotion, reason, store, store_returns, store_sales, warehouse, web_page, web_returns, web_sales,* and *web_site*.

Table 7: Data Load Times

TPC-DS Data File	1GB Dataset Load Times	5GB Dataset Load Times
Call_center	0.40s	0.50s
Catalog_page	3.39s	3.41s
Catalog_returns	27.51s	2m13.02s
Catalog_sales	4m50.00s	24m50.00s
Customer	16.07s	50.37s
Customer_address	7.87s	23.68s
Customer_demographics	4m40.00s	5m3.35s
Date_dim	14.34s	17.19s
Household_demographics	1.02s	1.03s
Income_band	0.04 $x10^{-1}$s	0.06$x10^{-1}$s
Inventory	24m44.13s	1h53m51.00s
Item	3.65s	11.45s
Promotion	0.05s	0.08s
Reason	0.06 $x10^{-1}$s	0.02s
Ship_mode	0.04 $x10^{-1}$s	0.03 $x10^{-1}$s
Store	0.04 $x10^{-1}$s	0.02s
Store_returns	47s	4m19.23s
Store_sales	8m18.03s	45m30.65s
Time_dim	11.90s	13.16s
Warehouse	0.03 $x10^{-1}$s	0.02 $x10^{-1}$s
Web_page	0.01s	0.02s
Web_returns	12.04s	1m8.15s
Web_sales	2m22.80s	12m55.86s
Web_site	0.008s	0.01s
total	47m20.14s	3h31m53.72s

Table 8: Query Selectivity

	Query 7	Query 21	Query 46	Query 50
9.94GB	0.60MB	0.34MB	2.48MB	0.003MB
41.93GB	2.28MB	1.55MB	10.42MB	0.01MB

Table 9: Query Execution Runtimes

	Query 7	Query 21	Query 46	Query 50
Experiment 1	15.71s	33.77s	3m18.00s	26.08s
Experiment 2	7.30s	26.84s	63.93s	52.61s
Experiment 3	0.62s	0.17s	3.43s	1.25s
Experiment 4	37.02s	2m39.00s	11m5.00s	1m57.00s
Experiment 5	22.55s	1m47.00s	6m16.00s	4m36.00s
Experiment 6	2.71s	0.52s	11.12s	5.12s

Figures 9 and 10 illustrate the query performance on the 9.94GB and 41.93GB dataset taking into consideration the data models and deployment environments.

We analyze the query execution runtimes below.

1. Experiments 3 and 6 have the fastest query runtimes in their respective dataset sizes. It can be observed that the two experiments correspond to the denormalized data models running on stand-alone systems. This indicates that denormalized data models outperform their normalized counter parts. Also, the algorithm implemented to execute queries against a denormalized data model handles scaling effectively, for the scales used in our experiments.
2. Among the experiments conducted against a normalized data model, stand-alone systems are observed to be faster than sharded systems. The runtimes for the Queries 7, 21, and 46 for Experiments 1, 2 and 4, 5 support this statement.
3. Query 50 is observed to be faster on the sharded system. This does not indicate that the sharded system is slower or faster than a stand-alone system. It depends on the type of the query sent to either of the systems. If a query includes a shard key, the mongos routes the query to a specific shard rather than broadcasting the query to all the shards in the cluster, enhancing the query performance, which is the case for Query 50. Thus, we can infer that Queries 7, 21, and 46 which are faster on a stand-alone system are being broadcasted on the sharded system, resulting in slower runtimes.

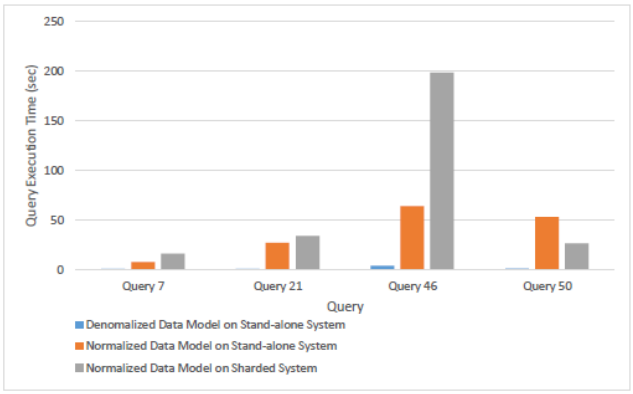

Figure 9: A Comparison of Query Execution Times for 9.94GB Dataset

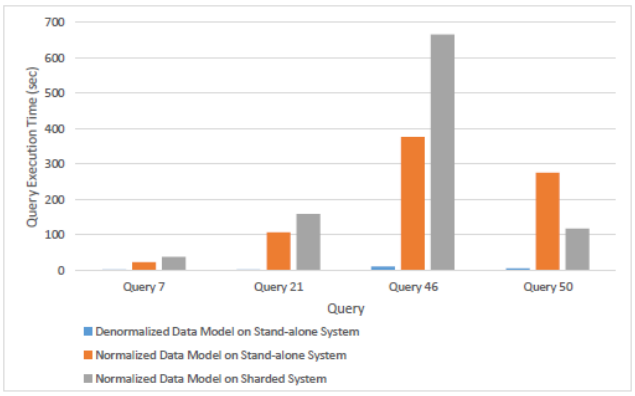

Figure 10: A Comparison of Query Execution Times for 41.93GB Dataset

There have been other studies that benchmarked the performance of MongoDB [8, 25, 26, 27, 28]. However, the majority of them performed the MongoDB benchmark against popular databases such as Oracle, Cassandra, HBase, and Couchbase but none of them study the performance of MongoDB based on the deployment environments, data modeling, and scalable datasets. We specifically focus on modeling relational data in a NoSQL environment and tune it in different ways and study why one data

model outperforms the other. Also, we base our conclusions on running complex analytical queries on each of the data models.

Based on the results obtained from executing analytical queries on different experimental setups, we conclude that performance of queries on MongoDB is influenced by the data model and deployment environment. Given the two choices of normalized and denormalized data models, a decision has to be made based on the amount of data stored in each document. If the size of a document does not exceed 16MB, a denormalized data model should be chosen over a normalized data model. Our experiments indicate that a denormalized data model results in faster query times than its normalized counterpart irrespective for the two dataset sizes we investigate. Queries against a normalized data model are slower since join operations followed by the embedding procedure are expensive. It is also expensive in terms of storage consumption due to the creation of intermediate collections.

Given the two choices of a stand-alone and sharded deployment environment, a decision has to be made based on application needs, available resources, and types of queries to be executed. A sharded system is an appropriate choice when an application deployed on stand-alone system becomes resource intensive resulting in an increase in data volumes and read/write throughput. Another scenario considers the high cost incurred from running a stand-alone system as compared to a sharded system. When operating with a sharded system, we have to be mindful of the following aspects. Firstly, the shard key is immutable, i.e., it cannot be altered after the collection is sharded. Secondly, queries have to be structured to use a shard key to prevent broadcasting across all the shards in the cluster. This allows the mongos to route the query to a specific shard providing better performance. On the other hand, if a stand-alone system is utilized within its resource limits, it can be equally effective at processing queries as compared to a sharded system with equivalent resources. Since it does not encounter the constraint of a shard key, indexing can be applied to any field and a wide variety of queries can be directed to the system. Based on the experimental results we can conclude that a stand-alone system is a suitable choice when queries with varying predicates are directed to the system and a sharded system is a suitable choice when specific queries containing the shard key are directed to the cluster.

7. CONCLUSIONS AND FUTURE WORK

In this section, we summarize the contribution of our research and elaborate possible extensions for future work.

We address the performance impact organizations may face if they choose to run complex analytical queries on a stand-alone and sharded document store. We highlight the importance of data modeling coupled with deployment environments. Different experimental setups are implemented to evaluate the combination of a data model and a deployment environment and its impact on query performance and scaling.

For conducting performance evaluation of analytical queries, we use TPC-DS as our chosen benchmark to generate data and analytical queries. We develop a data loading algorithm to migrate data generated by the TPC-DS benchmark into MongoDB for performing query analysis. TCP-DS generated SQL queries employ join operations and SQL operations. Since MongoDB does not support joins, we develop an algorithm to simulate join operations and perform aggregation operations.

We assess the performance of stand-alone MongoDB system with a MongoDB sharded system and conclude that it is dependent on deployment environments, data models, dataset sizes, and query types. Our results indicate that the denormalized data model speeds up queries by a significant amount when deployed on a stand-alone environment. The trend in execution times remains the same with the increase in scalability we investigated.

With the help of the algorithms proposed here, a tool can be created that migrates SQL-like data into MongoDB. Since MongoDB lacks the support for join operations, the query translation algorithm developed here can be used as a basis for a tool to streamline join and aggregation operations.

There are many different ways that this study could be both broadened and deepened. Specifically, the preliminary experiments described here could be extended by implementing a denormalized data model on a sharded system and comparing its performance to a denormalized data model implemented on a stand-alone system. Other ways of organizing the data could be considered. Herrero et al. consider conceptual, logical, and physical design of NoSQL databases and propose techniques that allow for varying amounts of denormalization depending on factors such as query workload [29]. Truică et al. study the performance of three document databases (including MongoDB) and three relational databases for create, read, update, and delete operations [30]. The replication of data considered in their approach would be interesting to consider as part of a future study using benchmark queries.

The research work can be furthered in different aspects such as varying the dataset sizes, increasing the scalability, and using different benchmarks that are suited for schema-less data.

MongoDB can be implemented on both a sharded and stand-alone system. For the case of a stand-alone system, MongoDB is thread safe; multi-threading is handled by the database on the client side. MongoDB allows operations to lock at either the collection level or database level. In a collection level lock, only a single thread is allowed to perform write operations to the collection. Aggregation queries usually involve multiple collections that are queried individually followed by the aggregation operations. Since MongoDB uses collection level locks, individual threads can be used to query each collection in parallel and then perform aggregation on a single thread that runs after the completion of the other threads. In our research, the entire query was performed on a single thread; using multithreading may reduce the query execution times. The same concept can be used in the sharded environment where individual collections reside on different shards and multiple threads can be issued to query each collection in parallel. Parallelizing can also be performed for aggregation over collection subsets using multiple threads on a sharded database.

The performance of MongoDB can be tested further by employing a more varied dataset in terms of number of fields and datatypes, and also by using a different benchmark intended for a document store database. Different denormalized data models could be deployed on the sharded cluster and its performance can be studied by varying the number of shards and the number of mongos instances, and implementing multi-threading. Additional metrics could be collected such as the amount of data read from disk and transmitted. All of the possible extensions could be executed in a more recent MongoDB engine such as WiredTiger [31], which introduces enhanced features including compression, column-oriented management techniques, and additional indexing styles.

8. REFERENCES

[1] (January 11, 2012). *What is big data?*. Available: https://beta.oreilly.com/ideas/what-is-big-data.

[2] A. Moniruzzaman and S. A. Hossain, "NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv Preprint arXiv:1307.0191*, 2013.

[3] (2013). *MongoDB: A Document Oriented Database*. Available: http://www.mongodb.org/about/.

[4] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35-40, 2010.

[5] (April, 2012). *TPC BENCHMARK ™ DS*. Available: http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf.

[6] C. Baru, M. Bhandarkar, R. Nambiar, M. Poess and T. Rabl, "Setting the direction for big data benchmark standards," in *Selected Topics in Performance Evaluation and Benchmarking*, Springer, 2013, pp. 197-208.

[7] A. Nayak, A. Poriya and D. Poojary, "Type of NoSQL databases and itscomparison with relational databases," *International Journal of Applied Information Systems*, vol. 5, 2013.

[8] Z. Parker, S. Poe and S. V. Vrbsky, "Comparing NoSQL Mongodb to an SQL db," in *Proceedings of the 51st ACM Southeast Conference*, 2013, pp. 5.

[9] (April 9, 2015). *Sharding*. Available: http://docs.mongodb.org/manual/sharding/.

[10] (June 04, 2015). *MongoDB Documentation Release 3.0.3*. Available: http://docs.mongodb.org/master/MongoDB-manual.pdf.

[11] *BSON*. Available: http://bsonspec.org/.

[12] N. Nurseitov, M. Paulson, R. Reynolds and C. Izurieta, "Comparison of JSON and XML data interchange formats: A case study." *Caine*, vol. 2009, pp. 157-162, 2009.

[13] D. Pritchett, "Base: An acid alternative," *Queue*, vol. 6, pp. 48-55, 2008.

[14] (2014, September 25). *Sharding Methods for MongoDB*. Available: http://www.slideshare.net/mongodb/sharding-v-final.

[15] R. Han, X. Lu and J. Xu, "On big data benchmarking," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, Springer, 2014, pp. 3-18.

[16] M. Poess, R. O. Nambiar and D. Walrath, "Why you should run TPC-DS: A workload analysis," in *Proceedings of the 33rd International Conference on very Large Data Bases*, 2007, pp. 1138-1149.

[17] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte and H. Jacobsen, "BigBench: Towards an industry standard benchmark for big data analytics," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1197-1208.

[18] S. Huang, J. Huang, J. Dai, T. Xie and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proceedings of the 2010 IEEE 26th International Conference On Data Engineering Workshops (ICDEW)*, 2010, pp. 41-51.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi and S. Zhang, "Bigdatabench: A big data benchmark suite from internet services," in *Proceedings of 2014 IEEE 20th International Symposium On High Performance Computer Architecture (HPCA)* , 2014, pp. 488-499.

[20] (2015). *AWS Documentation*. Available: http://aws.amazon.com/documentation/.

[21] *HashMap (Java Platform SE 7)*. Available: http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html.

[22] *MongoDB Java Driver Documentation*. Available: http://mongodb.github.io/mongo-java-driver/3.0/.

[23] *ArrayList (Java Platform SE 7)*. Available: http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html.

[24] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *Roedunet International Conference (RoEduNet), 2011 10th*, 2011, pp. 1-5.

[25] M. Fotache and D. Cogean, "NoSQL and SQL databases for mobile applications. Case Study: MongoDB versus PostgreSQL," *Informatica Economica*, vol. 17, pp. 41-58, 2013.

[26] V. Abramova and J. Bernardino, "NoSQL databases: MongoDB vs. Cassandra," in *Proceedings of the International C* Conference on Computer Science and Software Engineering*, 2013, pp. 14-22.

[27] *TPC Documentation as of 04/27/2015*. Available: http://www.tpc.org/information/current_specifications.asp.

[28] (Aug 26, 2013). *Using TPC-DS to generate RDBMS performance and benchmark data.* Available: http://www.innovation-brigade.com/index.php?module=Content&type=user&func=display&tid=1&pid=3.

[29] V. Herrero, A. Abello and O. Romero, "NOSQL design for analytical workloads: variability matters," in *Proceedings of the 35th International Conference on Conceptual Modeling (ER)*, 2016, pp. 50-64.

[30] C.-O. Truică, I.I. Bucur and A. Boicea, "Performance evaluation for CRUD operations in asynchronously replicated document oriented database," in *Proceedings of the International Conference on Control Systems and Computer Science*, 2015, pp. 191-196.

[31] (January 15, 2017), WiredTiger, Available: http://www.wiredtiger.com/.