SQL Query Optimization in Distributed NoSQL Databases for Cloud-Based Applications

Aristeidis Karras[1], Christos Karras[1], Antonios Pervanas[1], Spyros Sioutas[1], and Christos Zaroliagis[1,2(✉)]

[1] Computer Engineering and Informatics Department, University of Patras, 26504 Patras, Greece
{akarras,c.karras,pervanas,sioutas}@ceid.upatras.gr, zaro@ceid.upatras.gr

[2] Computer Technology Institute and Press "Diophantus", Patras University Campus, 26504 Patras, Greece

Abstract. A method for query optimization is presented by utilizing Spark SQL, a module of Apache Spark that integrates relational data processing. The goal of this paper is to explore NoSQL databases and their effective usage in conjunction with distributed environments to optimize query execution time, in order to accommodate the user complex demands in a cloud computing setting that necessitate the real-time generation of dynamic pages and the provision of dynamic information.

In this work, we investigate query optimization using various query execution paths by combining MongoDB and Spark SQL, aiming to reduce the average query execution time. We achieve this goal by improving the query execution time through a sequence of query execution path scenarios that split the initial query into sub-queries between MongoDB and Spark SQL, along with the use of a mediator between Apache Spark and MongoDB. This mediator transfers either the entire database from MongoDB to Spark, or transfers a subset of the results for those sub-queries executed in MongoDB. Our experimental results with eight different query execution path scenarios and six difference database sizes demonstrate the clear superiority and scalability of a specific scenario.

Keywords: Big Data and the Cloud · Query Optimization · SparkSQL · NoSQL databases · Indexes · Big Data Analytics for Cloud computing

1 Introduction

Data mining and analytics sectors have drawn much attention in our days by both academic and businesses communities in order to handle massive datasets. With modern libraries and existing systems such as Hadoop [4,30], which is a frequently used cloud platform for data mining, the efficient management of big

L. Foschini and S. Kontogiannis (Eds.): ALGOCLOUD 2022, LNCS 13799, pp. 21–41, 2023.
https://doi.org/10.1007/978-3-031-33437-5_2

data is no longer a promise. Several machine learning methods based on the MapReduce [14] architecture have gained popularity as they can be deployed on the cloud with the use of Apache Spark [6]. In contrast, when similar algorithms are implemented using MapReduce, intermediate results are written to the Hadoop Distributed File System (HDFS) [4] and read from there. However, this requires a considerable amount of time for disc I/O operations as well as vast amounts of resources for communication and storage.

Cloud computing can enhance analytics, machine learning, and other possible directions as the data are stored in a cloud provider and not locally. However, traditional relational databases face many challenges when employed in a cloud setting. There is a constant demand for high concurrent database read/write operations. In cloud computing, the complex demands of users necessitate the real-time generation of dynamic pages and the provision of dynamic information; as a result, the database concurrency rate is excessively high and tends to receive thousands of reading requests per second. It is difficult for a relational database to accommodate tens of thousands of SQL data write requests, and the hard drive cannot support the load. Additionally, there is a huge demand for the efficient storage and access of massive data. The massive data created dynamically, for relational databases in a cloud computing environment, has resulted in storing hundreds of millions of records in a table, making it exceedingly inefficient to execute an SQL query.

In contrast, complicated SQL queries that need multi-table lookup operations have led to the development of flexible systems such as the one presented here. In a system that contains massive amounts of data, we could issue several connected queries across big tables, intelligent data processing, and extensive data reporting. Although simple conditional paging queries on a single table with a primary key are often employed in cloud computing scenarios, they produce an extensive load to the environment, hence, we should seek for other options.

Despite the fact that the prevalence of relational databases (RDBMS) indicates that users often prefer making declarative queries, the relational method is inadequate for many big data applications. Initially, users want to extract, transform and load to/from multiple semi or unstructured data sources, which requires specialized programming. Secondly, customers might do complex analytics, such as machine learning and graph processing, which are difficult to be performed in RDBMS. Particularly, the majority of data pipelines shall ideally be defined using both relational queries and complicated procedural methods. Up to now, such kinds of systems, relational and procedural, have remained essentially separate, requiring users to choose between the two methods.

For the aforementioned reasons, we mainly focus in this work on distributed databases for query optimization including Spark SQL [6] and MongoDB [27] and show how to utilize both relational and procedural models in MongoDB and Spark SQL, using Hadoop [4,30,34]. With the use of a MongoDB connector for Apache Spark, the preceding connection occurs in order to perform speedy and complex queries. Spark SQL is an extension of Spark for structured data processing. Spark SQL allows users to effortlessly combine relational and proce-

dural APIs, rather than requiring them to choose between the two. Furthermore, frameworks like Hadoop, Apache Spark, and Apache Storm [7], as well as distributed data storages such as HDFS and HBase [5], are gaining popularity since they are designed to make the processing of extremely massive volumes of data almost straightforward. Such systems appear to have a great deal of interest, and therefore, libraries (such as MLlib of Apache Spark) that enable the development and application of Machine Learning methods in the cloud are noteworthy.

Spark SQL bridges the gap between the relational and procedural models by contributing in two ways. Spark SQL offers a DataFrame (DF) API that may conduct relational operations on external data sources as well as the own distributed collections of Spark. MongoDB is utilized for speedy index queries. The API provides Spark applications with extensive relational/procedural interaction. DFs are collections of structured records that can be modified using either the procedural API of Spark or the new relational APIs that enable more efficient optimizations. They may be constructed directly from distributed Java/Python object collections, allowing relational processing in current Spark applications.

In this work, we utilize Spark SQL along with MongoDB to efficiently perform complex queries and improve their runtime. We investigate query optimization using various query execution paths by combining MongoDB and Spark SQL, aiming to minimize the average query execution time. We improve the query execution time by splitting the query into sub-queries, considering various scenarios that split sub-queries between MongoDB and Spark SQL, along with the use of the connector between Apache Spark and MongoDB. This mediator transfers either the entire database from MongoDB to Spark, or transfers a subset of the results for those sub-queries executed in MongoDB. Our experimental results with eight different query execution path scenarios and six difference database sizes (ranging from 500,000 to 20,000,000 records) demonstrate the clear superiority and scalability of a specific scenario.

The remainder of the paper is organized as follows. In Sect. 2 the fundamental elements of Spark, Resilient Distributed Datasets (RDDs), and MongoDB are presented. Section 3 describes the implementation of several query execution plans in MongoDB and Spark. Section 4 highlights the experimental results and their findings. Section 5 discusses the idea of sharding for further improvements on the query performance over huge data sets. Finally, conclusions and future directions of this work are presented in Sect. 6.

2 Preliminaries

Big Data refers to the deluge of digital data from a variety of digital sources, including sensors, scanners, smartphones, videos, e-mails, and social media. These data include texts, photos, videos, and sounds, as well as their combinations. In the big data era, applications require a combination of processing algorithms, data sources, and storage formats to accomplish a common goal which is big data processing. Nowadays this has turned toward big data warehouses [32] and high-performance computing environments that can handle geospatial

big data [21] among others. The initial systems built for these types of workloads, such as MapReduce which is offered by Apache Spark, provide users with a strong yet procedural programming interface. However, such systems are difficult to program and need manual tuning by the user to get optimal performance. As a consequence, a number of innovative technologies aimed to deliver a more productive user experience by providing relational interfaces to large amounts of data. Systems like Asterix, Hive, Dremel, and Shark [9,26,31,33] all use declarative queries to deliver more robust automated optimizations.

Apache Spark which is utilized in this work is a distributed cluster computing engine with APIs in Scala, Java, and Python and libraries for streaming, graph processing, and machine learning [28]. It is one of the most widely-used systems with a language-integrated API similar to DryadLINQ [18], and the most active open-source project for big data processing. Spark offers a functional programming API similar to other systems [11,18], where users manipulate distributed collections called Resilient Distributed Datasets (RDDs) [34]. Each RDD is a set of Java or Python objects partitioned throughout a cluster. RDDs can handle operations like map, filter, and reduce, which take functions in the programming language and transfer them to nodes on the cluster. An example of a Scala code that counts lines starting with "ERROR" within a text file is given below (Listing 1):

Listing 1: Scala Example Code

```
lines = spark.textFile ("␣hdfs␣://...")
errors = lines.filter(s => s. contains ("␣ERROR␣"))
println(errors.count ())
```

The preceding example constructs an RDD of strings named lines by reading an HDFS file, which then transforms it using a filter to obtain another RDD, named errors, and then performs a count on this data. RDDs are fault tolerant meaning that the system can recover lost data using the lineage graph of the RDDs by rerunning operations such as the filter above to rebuild missing partitions. They can also explicitly be cached in memory or on disk to support iteration [34]. One final note about the API is that RDDs are evaluated lazily. Each RDD represents a "logical plan" to compute a dataset, but Spark waits until certain output operations, to launch a function. This allows the engine to perform some simple query optimization, such as pipelining operations.

In particular, Spark will pipeline reading lines from the HDFS file by applying the filter and computing a running count, so that it never needs to materialize the intermediate lines and error results. Although such optimizations are extremely useful, they are also limited because the engine does not understand the structure of the data in RDDs which are Java/Python objects or the semantics of user functions that contain arbitrary code. Nonetheless, the most basic data processing paradigms are relational queries that RDDs cannot manage. To address this, Apache Spark requires a number of higher-level libraries. Spark SQL is one

of the innovative components of the Apache Spark Framework that combines relational processing with the functional programming API of Apache Spark. It enables Apache Spark developers to use the advantages of relational processing.

Spark SQL allows a seamless mix of SQL Queries within the environment of Apache Spark. Spark SQL is capable to perform data processing on structured data, or on Resource Description Framework (RDFs) stores, or in DataFrames (DFs). RDF is a graph-based data model, composed of triples (s, p, o); such a triple denotes a directed arc (s, o) with label p. RDFs can be applied to matrix computations [13] as well as to knowledge graph representations [2]. Spark SQL can support batch processing [3] of RDFs in a matter of seconds. It can also support storage, partitioning, indexing, and information retrieval in the spectrum of Big Data [12]. A DF is a distributed collection of data organized into named columns. Users can use a DataFrame API to perform various relational operations on both external data sources and Spark's built-in distributed collections without providing specific procedures for processing data.

Transiting from traditional SQL-based approaches to NoSQL techniques requires layers that convert relational databases to key-value stores. Numerous studies have suggested alternative layers to convert relational databases to NoSQL; however, the majority of them focused on just one or two models of NoSQL and assessed their layers on a single node, not in a distributed environment. Therefore, Spark-based layers that are able to map relational databases to NoSQL storage have emerged [1]. Of course, the necessity here is to utilize a connector that takes advantage of both distributed computing engines such as Spark and the exceptional speed that MongoDB has to offer as per searches in documents.

MongoDB [27] is a document-based NoSQL datastore that is commercially maintained by 10gen. MongoDB in particular is among the most promising databases existing because of its nature and its superior performance. Despite being a non-relational database, MongoDB provides several relational database functions, such as sorting, secondary indexing, range queries, and nested document querying. Operators like create, insert, read, update and remove as well as manual indexing, indexing on embedded documents and indexing on location-based data are also supported. In such systems, data are kept in documents, which are entities that offer structure and encoding for the managed data. Each page is effectively an associative array containing a scalar value, lists, or arrays nested inside arrays. Every document has a unique special key called "ObjectId" that is used for explicit identification, but this key and the document it corresponds to are conceptually comparable to a key-value pair.

Documents in MongoDB are serialised as Javascript Object Notation (JSON) objects and saved using a binary encoding of JSON known as BSON. MongoDB, like other NoSQL systems, has no schema limits and can allow semi-structured data, as well as multi-attribute lookups on records that may contain multiple types of key-value pairings [22]. Documents are often semi-structured files such as XML, JSON, YALM, and CSV. There are two methods for storing data: a) nesting documents inside each other, which may accommodate one-to-one

or many-to-many relationships, and b) reference to documents, in which the referred document is only obtained when the user requests data from this document.

Cloud computing can be integrated with MongoDB databases along with modern technologies such as the Internet of Things (IoT) for streaming applications [16], or for IoT Data Management on the Cloud [15]. Cloud-based applications that promote and support smart cities and overall well-being in societies can enhance information management as a service [10].

3 Query Execution Plans

3.1 Indexing in MongoDB

Having previously discussed the use of Apache Spark and Spark SQL, we shall now provide a simple example of constructing an index and demonstrate how it influences the query runtime. For this purpose, we shall use the following example (Listing 2) of a MongoDB database, consisting of one million records.

Listing 2: Index Construction in MongoDB

```
{
    "_id":{"$oid":"61a6540c3838fe02b81e5338"},
    "Region":"Sub-Saharan␣Africa",
    "Country":"South␣Africa",
    "Item␣Type":"Fruits",
    "Sales␣Channel":"Offline",
    "Order␣Priority":"M",
    "Order␣Date":{"$date":"2012-07-26T21:00:00.000Z"},
    "Order␣ID":443368995,
    "Ship␣Date":{"$date":"2012-07-27T21:00:00.000Z"},
    "Units␣Sold":1593,
    "Unit␣Price":9.33,
    "Unit␣Cost":6.92,
    "Total␣Revenue":14862.69,
    "Total␣Cost":11023.56,
    "Total␣Profit":3839.13
}
```

Instead of storing the data in the form of tables with columns and rows, the data is stored as documents. Each document can be one of the relational matrices of the numerical values, or the overlapping interrelated arrays or matrices. These documents are serialized as JSON objects and stored internally using JSON binary encryption known as BSON in MongoDB. The data is partitioned and stored on several servers called shard servers for simultaneous access and effective read/write operations.

Assume that the following SQL query (Listing 3) is to be executed within the given database.

Listing 3: SQL Query

```
SELECT Country, Region, Unit Price, Unit Cost
    FROM sales
    WHERE Unit Price > 600
    AND Unit Cost < 510
    ORDER BY Region
```

The aforementioned query is well formatted in SQL, making it easy to comprehend. In order to execute the query in MongoDB, we make use of mongosh, a component of the MongoDB Compass tool[1] to construct the database.

The previous query can now be executed utilizing an equivalent function (Listing 4):

Listing 4: MongoDB Aggregation Function

```
db.myBigCollection.aggregate([{$project: {
    Country: 1, Region: 1, 'Unit␣Price': 1,
    'Unit␣Cost': 1}},
    {$match: {'Unit␣Price': {$gt: 600},
    'Unit␣Cost': {$lt: 510}}},
    {$sort: {Region: 1}}])
    .explain()
```

By utilizing the *explain*() function, we observed an average query execution time of 860 milliseconds (ms) for the specific database.

To improve the execution time of a certain query by creating an index, it is reasonable to believe that this index should be based on the columns "Unit Price" and "Unit Cost" on which the majority of the searches is performed.

Utilizing the following command (Listing 5), one compound index for the "Unit Price" and "Unit Cost" values are constructed in ascending order:

Listing 5: MongoDB Index Construction

```
db.myBigCollection.createIndex({"Unit␣Price": 1,
"Unit␣Cost": 1})
```

Now, the preceding query is re-executed and measured in terms of time. The execution time has been drastically lowered, varying from 250 to 270 ms. An

[1] Available at: https://www.mongodb.com/products/compass.

additional single-field index, depending on the field that is being used to sort the data, may be established. The following command (Listing 6) constructs a new index based on the "Region" field:

Listing 6: MongoDB New Index Construction

```
db.myBigCollection.createIndex({"Region": 1})
```

If the same query is re-executed utilising both indexes constructed, the average query execution time drops further to 220 ms. This demonstrates the significance of indexes, since the average execution time of a very basic query was lowered to roughly one fourth with the proper use of indexes.

3.2 Integration of MongoDB and Apache Spark

In this subsection, the information about MongoDB is applied to examine various instances of the MongoDB-Spark integration described in the previous Section. We will determine how to use the connection and how to apply our indexing methods, using the database and indexes described previously.

To highlight the differences among Spark SQL and MongoDB in terms of query execution, different operations must be considered. In general, MongoDB tends to be quicker for `INSERT/UPDATE` operations [17], while SQL appears to be faster for `SELECT` operations, but this is not a general rule. To investigate this problem, an identical database using DataFrames is constructed in Spark. We will execute the query from Sect. 3, and monitor its execution time.

Recall that without indexing, it took MongoDB an average of 860 ms to perform the query. Spark SQL executes the identical query in 310 ms without indexing, which is much faster than MongoDB. This already is a significant improvement in terms of time. The main reason that the execution time can be further improved in Spark SQL using indexing is that Apache Spark does not necessarily allow indexing in the same way as SQL does. Apache Spark is compatible with a range of data storage formats, some of which enable indexing while others do not. For instance, Spark along with PostgreSQL enables the usage of PostgreSQL indexes.

Having observed that Spark SQL executes certain queries faster than MongoDB, it becomes pretty clear that it is better to utilise a URL to get the data, rather than recreating a database in Apache Spark. Initially, Apache Spark is executed, including the link package named MongoDB Connector for Spark[2]. The initial objective here is to access the database generated previously in MongoDB and to transfer it to Spark. Using the following command (Listing 7), the data is transferred into a DataFrame, denoted by df.

[2] Available at: https://www.mongodb.com/docs/spark-connector/current/.

Listing 7: Dataframe Creation from MongoDB to Apache Spark

```
val df = spark.sqlContext.read.format
("com.mongodb.spark.sql.DefaultSource")
.option("uri",
"mongodb://127.0.0.1/myDb.myBigCollection")
.load()
```

Once the data are imported, a temporary SQL view of the "sales" DataFrame can be constructed utilizing the following command (Listing 8).

Listing 8: Temporary SQL View

```
df.createOrReplaceTempView("sales")
```

At this point, Spark SQL can be utilized to execute numerous queries on the database. We execute the query from Sect. 3 and measure its execution time for evaluation. To execute and measure the execution time of the query, the `spark.sql()` and `spark.time()` methods are used respectively as follows (Listing 9).

Listing 9: Spark SQL Query Time Measurement Command

```
spark.time(spark.sql(
SELECT Region, Country, ‘Unit Price‘, ‘Unit Cost‘
    FROM sales
    WHERE ‘Unit Price‘ > 600 AND ‘Unit Cost‘ < 510
    ORDER BY Region).show())
```

The average execution time of the aforementioned query is 580 ms, which is much slower than the 220 ms of MongoDB. This is due to the fact that the connection transfers data in real-time, resulting in a significant increase in the average execution time required to move data from MongoDB to Spark. In particular, the entire database is transferred from MongoDB to Spark, while the query is executed, and the results are derived at the end.

Therefore, we should consider how we might save time by moving the database so that the query execution times are not that lengthy. One way to improve the query time is to execute the query on MongoDB and transfer only the results to Spark. That is, instead of transferring the entire database in the DataFrame, the portion of the database is simply transferred that pertains to the given query. This is done by utilizing the following commands (Listing 10).

Listing 10: MongoDB Query Execution and Transferring the Results to Spark

```
val df = spark.sqlContext.read.format
("com.mongodb.spark.sql.DefaultSource")
.option("uri",
"mongodb://127.0.0.1/myDb.myBigCollection")
.option("pipeline",
{$project: {Country: 1, Region: 1,
'Unit␣Price': 1, 'Unit␣Cost': 1}},
{$match: {'Unit␣Price': {$gt: 600 },
'Unit␣Cost': {$lt: 510}}}, {$sort: {Region: 1}}).load()
```

Once the necessary information in the DataFrame exists, the results can be examined. After creating a temporary SQL view of the DataFrame with the same name "sales" (for convenience), a single query is executed to return all fields, as the DataFrame includes the required information. This is done through the following query (Listing 11):

Listing 11: Spark SQL Query Execution

```
spark.time(spark.sql(SELECT * FROM sales).show())
```

As anticipated, the average query execution time now drops to 180 ms. This time is lower than that of the MongoDB (220 ms) and this is due to the following reasons.

Recall first the two query scenarios. In the first scenario, the query is executed in MongoDB and the results are reported in MongoDB. In the second scenario, the query is executed in MongoDB, the data are transferred to Spark, and then the results are reported there. The obvious question is how the query execution time of the second scenario turns out to be faster than that of the first scenario, given the fact that the second scenario (and its corresponding execution path) requires more time due to the transfer of data.

The reason appears to be in the speed at which the query is executed using a `SELECT` operation in Spark SQL against the operations of MongoDB, as previously noted. Performing more experiments in the whole database in both MongoDB and Spark SQL (after transferring it), it appears that Spark SQL performs the `SELECT` operation significantly faster. In the second scenario, the complete database transfer is not required, but only a tiny portion of it that contains the results which are sent after the queries. Hence, the overall execution time will be much less. In the particular example used, around 83,000 records are returned out of the total of one million records in the database.

A subsequent question is whether the query execution time can be further reduced by exploiting the speed of the `SELECT(Spark SQL)` operation against that of the `$project(MongoDB)` operation. To investigate this idea, we divide the query into sub-queries. In particular, we split the query so that the `WHERE($match)` and the `ORDER BY($sort)` operations are executed in MongoDB, while the operation `SELECT($project)` is executed on Spark.

To execute the operations `WHERE($match)` and `ORDER BY($sort)` in MongoDB the following commands (Listing 12) are used.

Listing 12: MongoDB WHERE and ORDER BY Query Execution

```
val df = spark.sqlContext.read.format
("com.mongodb.spark.sql.DefaultSource")
.option("uri",
"mongodb://127.0.0.1/myDb.myBigCollection")
.option("pipeline", {$match: {'Unit␣Price': {$gt: 600},
'Unit␣Cost': {$lt: 510}}}, {$sort: {Region: 1}}).load()
```

To execute the operation `SELECT($project)` in SparkSQL the following commands (Listing 13) are used.

Listing 13: Spark SQL SELECT Command Execution

```
spark.time(spark.sql(SELECT Region, Country,
‘Unit Price‘, ‘Unit Cost‘ FROM sales).show())
```

Measuring now the average query execution time, we observe that it has been further reduced to approximately 105 ms. Spark SQL appears to be faster than MongoDB when executing the operation `SELECT from $project`.

Based on this additional improvement, a natural attempt would be to migrate the `ORDER BY($sort)` portion of the query to Spark SQL. This is done in two steps.

First, the operation `WHERE($match)` is executed in MongoDB using the following commands (Listing 14).

Listing 14: MongoDB WHERE Command Execution

```
val df = spark.sqlContext.read.format
("com.mongodb.spark.sql.DefaultSource")
.option("uri",
"mongodb://127.0.0.1/myDb.myBigCollection")
.option("pipeline", {$match: {'Unit␣Price':
{$gt: 600}, 'Unit␣Cost': {$lt: 510}}})
.load()
```

Then, the operations `SELECT($project)` and `ORDER BY($sort)` are executed in Spark SQL using the following commands (Listing 15).

Listing 15: Spark SQL SELECT and ORDER BY Execution

```
spark.time(spark.sql(SELECT Region, Country,
‘Unit Price‘, ‘Unit Cost‘ FROM sales ORDER BY
Region).show())
```

Measuring the query execution time of this experiment, we observed that the average time did not improve but rather increased significantly to 530 ms. This implies that the `ORDER BY($sort)` method in MongoDB appears to be sufficiently faster.

The discussion in this section demonstrates the need to consider various query execution scenarios and measuring the corresponding query execution times in order to recommend some best cases/practices. We do this in Sect. 4 where various scenarios are analysed and their query execution times are reported.

4 Experimental Results

In this Section we present the experimental results by running various scenarios of query execution paths on different database sizes and measure the average query execution time.

We considered the eight query execution path scenarios shown in Table 1, in which one part of the query is executed in MongoDB and the other part in Spark SQL (examples of such query scenarios were presented in Sect. 3.

The aforementioned scenarios were executed on six different database sizes in order to investigate the scalability of the specific query execution scenarios.

We initiated the database size to 500,000 records and doubled the size for generating the next database instance up to 20,000,000 records.

The average query times per scenario and database size are reported in Tables 2 to 7. The fastest query times are highlighted in bold.
We observe the following across all results (cf. Tables 2 to 7).

Table 1. Scenarios of Query Execution Paths.

Scenario	MongoDB	Spark SQL
1	Entire Query Execution	–
2	Entire Database Transfer	Entire Query Execution
3	WHERE($match) + ORDER BY($sort)	SELECT($project)
4	WHERE($match)	SELECT($project) + ORDER BY($sort)
5	ORDER BY($sort)	SELECT($project) + WHERE($match)
6	SELECT($project)	WHERE($match) + ORDER BY($sort)
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)
8	SELECT($project) + ORDER BY($sort)	WHERE($match)

Table 2. Average query execution time (in ms) per scenario for 500,000 records.

Scenario	MongoDB	Spark SQL	Avg Qtime (ms)
1	Entire Query Execution	–	121
2	Entire Database Transfer	Entire Query Execution	311
3	WHERE($match) + ORDER BY($sort)	SELECT($project)	72
4	WHERE($match)	SELECT($project) + ORDER BY($sort)	289
5	ORDER BY($sort)	SELECT($project) + WHERE($match)	**48**
6	SELECT($project)	WHERE($match) + ORDER BY($sort)	672
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)	518
8	SELECT($project) + ORDER BY($sort)	WHERE($match)	127

Table 3. Average query execution time (in ms) per scenario for 1,000,000 records.

Scenario	MongoDB	Spark SQL	Avg Qtime (ms)
1	Entire Query Execution	–	180
2	Entire Database Transfer	Entire Query Execution	580
3	WHERE($match) + ORDER BY($sort)	SELECT($project)	105
4	WHERE($match)	SELECT($project) + ORDER BY($sort)	530
5	ORDER BY($sort)	SELECT($project) + WHERE($match)	**55**
6	SELECT($project)	WHERE($match) + ORDER BY($sort)	850
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)	690
8	SELECT($project) + ORDER BY($sort)	WHERE($match)	210

Table 4. Average query execution time (in ms) per scenario for 2,000,000 records.

Scenario	MongoDB	Spark SQL	Avg Qtime (ms)
1	Entire Query Execution	–	337
2	Entire Database Transfer	Entire Query Execution	1429
3	WHERE($match) + ORDER BY($sort)	SELECT($project)	184
4	WHERE($match)	SELECT($project) + ORDER BY($sort)	1185
5	ORDER BY($sort)	SELECT($project) + WHERE($match)	**59**
6	SELECT($project)	WHERE($match) + ORDER BY($sort)	1338
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)	1129
8	SELECT($project) + ORDER BY($sort)	WHERE($match)	369

Table 5. Average query execution time (in ms) per scenario for 5,000,000 records.

Scenario	MongoDB	Spark SQL	Avg Qtime (ms)
1	Entire Query Execution	–	670
2	Entire Database Transfer	Entire Query Execution	8800
3	WHERE($match) + ORDER BY($sort)	SELECT($project)	1270
4	WHERE($match) + SELECT($project)	ORDER BY($sort)	6300
5	ORDER BY($sort)	SELECT($project) + WHERE($match)	**65**
6	SELECT($project)	WHERE($match) + ORDER BY($sort)	3048
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)	2438
8	SELECT($project) + ORDER BY($sort)	WHERE($match)	844

Table 6. Average query execution time (in ms) per scenario for 10,000,000 records.

Scenario	MongoDB	Spark SQL	Avg Qtime (ms)
1	Entire Query Execution	–	1237
2	Entire Database Transfer	Entire Query Execution	11469
3	WHERE($match) + ORDER BY($sort)	SELECT($project)	2543
4	WHERE($match) + SELECT($project)	ORDER BY($sort)	12894
5	ORDER BY($sort)	SELECT($project) + WHERE($match)	**102**
6	SELECT($project)	WHERE($match) + ORDER BY($sort)	5671
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)	4179
8	SELECT($project) + ORDER BY($sort)	WHERE($match)	1636

Table 7. Average query execution time (in ms) per scenario for 20,000,000 records.

Scenario	MongoDB	Spark SQL	Avg Qtime (ms)
1	Entire Query Execution	–	3659
2	Entire Database Transfer	Entire Query Execution	Out of memory
3	WHERE($match) + ORDER BY($sort)	SELECT($project)	6784
4	WHERE($match) + SELECT($project)	ORDER BY($sort)	Out of memory
5	ORDER BY($sort)	SELECT($project) + WHERE($match)	**285**
6	SELECT($project)	WHERE($match) + ORDER BY($sort)	11074
7	SELECT($project) + WHERE ($match)	ORDER BY($sort)	8514
8	SELECT($project) + ORDER BY($sort)	WHERE($match)	4855

The `ORDER BY($sort)` operation in MongoDB is exceptionally fast, faster than any other operation.

The `SELECT($project)` operation in Spark SQL is faster compared to the same operation in MongoDB.

The combination of the `SELECT($project)` and `WHERE($match)` operations in Spark SQL are exceptionally fast, faster than any other operation.

Scenario 5 is the fastest across all database sizes, due to the above facts.

Scenario 3 is the second fastest scenario for database sizes up to $2,000,000$ records, followed by scenarios 1 and 8 (cf. Tables 2 to 4).

As soon as the database size exceeds 2,000,000 records (cf. Tables 5 to 7), the transfer of a large amount of data between MongoDB and Spark begins to have a significant effect on the query execution time. This is also evident by the query execution time of scenario 2, in which the entire database is transferred to Sparl SQL, and which has the largest value, while the memory exceeded its limit in the case of the database with 20,000,000 records (cf. Table 7).

For database sizes beyond 2,000,000 records, scenario 1 (i.e., just run the entire query in MongoDB) is the second fastest, followed by scenarios 8 and 3 (cf. Tables 5 to 7).

Scenario 5 has an exceptional scalability not only because it is the fastest across all database sizes, but also due to the very good scaling of the average query execution time as the size of the database doubles from one instance to the next (cf. Tables 2 to 7).

The log-scaled results across all 8 different test scenarios and the 6 different database sizes are shown in Figs. 1 and 2.

Figure 1 presents the results across all databases sizes for scenarios 1–4. As we can see, scenarios 1 and 2 have similar behavior across all database sizes. The only difference appears in the case of scenario 2 and database size of 20,000,000 records, where the memory exceeded its limits. Scenario 3 remains the fastest execution plan across all database sizes, while we see an increase in time at 20,000,000 records. Lastly, scenario 4 has similar performance to scenario 2, but once again when the size of the database reaches 20,000,000 records the memory exceeded its limits.

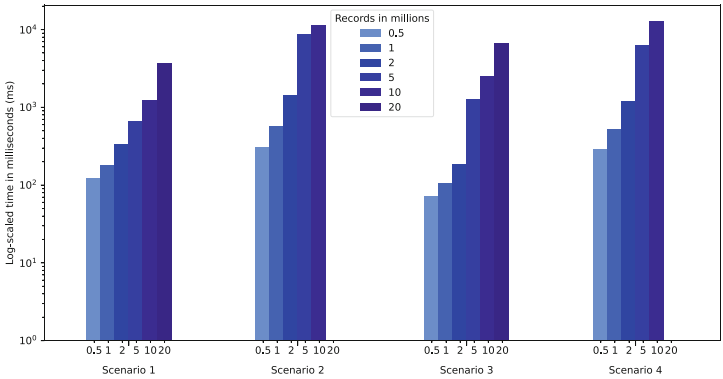

Fig. 1. Query Runtime for Scenarios 1-4 for 0.5, 1, 2, 5, 10 and 20 million records.

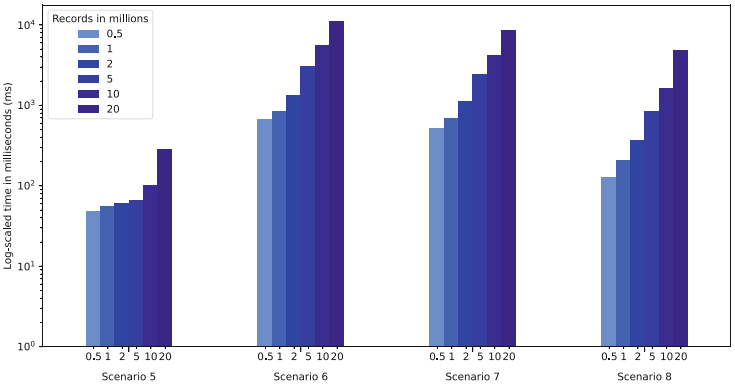

Fig. 2. Query Runtime for Scenarios 5-8 for 0.5, 1, 2, 5, 10 and 20 million records.

Figure 2 presents the results across all databases sizes for scenarios 5–8. As we can see, scenario 5 is the fastest as per query execution time across all database sizes. Scenario 6 appears to be the slowest. Scenario 8 is the second best. Both scenarios 7 and 8 have similar behavior across all database sizes. We also observed that scenarios 5–8 did not cause the memory to reach its limits.

5 Further Extensions

In this section, we shall discuss a technique known as *sharding* that can be used to further improve the query performance of huge data sets.

Recall that MongoDB stores data as documents, instead of storing data as tables with columns and rows. Every document may be represented by one of the relational matrices of numerical values or the overlapping connected arrays or matrices. These documents are serialised as JSON objects and saved internally using JSON binary encryption (known as BSON in MongoDB).

The data are partitioned and stored on many servers known as *shards* or *shard servers* to facilitate simultaneous read/write operations.

This connection integrates MongoDB with Apache Spark using a cluster assignment function $C : X \rightarrow \{1, 2, \ldots.., K\}$, where K refers to the number of clusters across all documents, X refers to a set of N objects (documents), and $d \in \mathbb{R}_0^+$ refers to a distance function (symmetric, non-negative and obeying the triangle inequality) between all pairs of objects in X.

Then, the goal is to partition X into K disjoint sets

$$X_1, X_2, \ldots, X_K$$

such that $\sum_{x,x' \in X_p} d(x, x')$ is minimized for each $1 \leq p \leq K$, while the distance $d(y, y')$ between any two points $y \in X_i$ and $y' \in X_j$, $i \neq j$, is maximized.

The number of all possible distinct cluster assignments $S(N, K)$ is given by

$$S(N,K) = \frac{1}{K!}\sum_{p=1}^{K}(-1)^{K-p}\binom{K}{p}p^N \tag{1}$$

The function $S(N, K)$ can be used to determine the optimal cluster assignment function C for a given set of data, by finding the value of p that minimizes the value of $S(N, K)$. In the context of sharding, this could be used to find the optimal number of shards (corresponding to clusters) for a database or any other distributed system by minimizing the number of shards needed to store a given amount of data.

Sharding is a way to distribute data across multiple devices, to deal with applications that use huge databases and structures. A database may have a mix of sharded collections and unsharded collections.

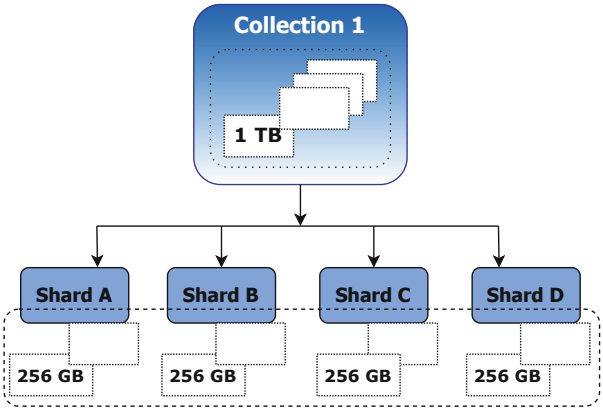

Fig. 3. Sharding Phase of MongoDB

Sharding in MongoDB uses subsets of data which are later moved from one shard to another; cf. Fig. 3. One way to identify which subset is being moved is by the selected key. For example, if we were to split a collection of users based on the field `username`, then the data is split into *chunks* (parts of a file) of predefined ranges e.g., [“a”,“f”)[3]. Then“a”, “charlie”, and “ezbake” could be in the set, but “f” could not.

A MongoDB shard cluster is comprised of two or more shards, one or more configuration servers, and an arbitrary number of routing processes. Each component is detailed below.

[3] The standard range notation is used where “[” and “]” denote inclusive bounds and “(” and “)” denote exclusive bounds.

- *Shard*: each shard consists of one or more servers and uses MongoDB processes to store data. Each shard in a production environment will consist of a replica set to ensure availability and automated fail-over.
- *Configuration server*: it stores the metadata of the cluster, which includes basic information about each shard server and the chunks it contains.
- *Mongos (Routing Processes)*: they concernt the routing and coordination processes. When MongoDB receives a request from a client, it routes the request to the appropriate server and merges the results before sending them back to the client.

Sharded sets are divided into clusters and spread throughout the shards, using a cluster assignment function, as discussed above. Unsharded collections are stored on the main shard.

MongoDB measures the theoretical maximum collection size as follows. Let B_{max} be the maximum BSON document size (in MB) and let Y_{avg} be the average size of shard key values (in bytes). Then, the maximum number M of splits is given by $M = B_{max}/Y_{avg}$. Assuming a chunk size of H (in MB), we have that the maximum collection size M_B (in MB) is given by

$$M_B = \frac{M \cdot C}{2} \tag{2}$$

The size of the chunks, which is the basic unit of data movement in sharded clusters, also plays a significant role in the performance of operations such as migrations. Adjusting the chunk size can help to balance the trade-offs between the need for data movement and the need to keep chunks small enough to prevent hotspots[4].

An additional technique concerns *zone sharding* that allows the assignment of ranges of shard keys to different shards, or a group of shards. This technique can be used to distribute data based on access patterns; for instance, assigning frequently accessed data to a specific set of shards can improve query performance. Furthermore, complex queries can be split and executed on specific shards based on their complexity and the capacity of the selected shard.

In order to achieve optimal performance in MongoDB, it is essential to constantly monitor and optimize the sharding configuration by considering the usage of sharding in the query execution plan, monitoring and optimizing the sharding configuration, choosing the right shard key, indexes, chunk size, and using techniques like zone sharding. These steps can greatly improve the performance of MongoDB in a large and complex data environment.

6 Conclusions and Future Work

We presented an approach for query optimization in terms of average query execution time for NoSQL databases and Spark SQL. The query execution path

[4] Hotspots in sharded clusters refer to situations where a specific chunk of data receives a disproportionate amount of read and write operations, causing performance issues.

scenarios that were examined demonstrate that our results are promising. By examining the aforementioned database instances and scenarios, the objective of this work was to determine how the connection among MongoDB and Apache Spark operates and therefore to investigate potential optimization possibilities using the connector and the indexing algorithms offered by MongoDB. One of our findings is that the `SELECT` operation in Spark SQL is typically faster compared to the same operation in MongoDB.

To further substantiate this finding, one could investigate as many potential scenarios as possible, in order to either discover the optimal answer to a given question, or to detect optimization tendencies. Naturally, the integration of all conceivable scenarios and conditions is endless and therefore it is impossible to map all feasible improvements for each specific instance. This work can be considered as a useful step forward to SQL query optimization in distributed systems utilizing NoSQL databases. Based on our current approach, our outcomes and the evaluated methodologies, we also believe that this work can be further expanded.

Future directions include collaborating with major organisations, businesses, and cooperatives that can provide a portion of their vast amounts of real-world data, in order to develop a variety of optimization models based on the current work. Hence, it will be possible to detect broad optimization tendencies based on the used databases and the frequency of queries. Thus, the database administrators (used to establish their own database) will be able in the future to utilize these models and adjust their database and query path execution plans to them.

The preceding directions may be performed automatically by using a smart query optimizer as the ones presented in [8,19,20,23–25,29]. However, the implementation of such a tool should combine query evaluation and optimization methods along with machine learning techniques. We strongly believe that these methods would be interesting to be used on specific use cases, where after several experiments the appropriate cost functions can be found in order to create one highly efficient query execution scheduler able to scale and adapt.

To further improve the query execution time, one approach is to distribute a given complex query to sharded queries on RDDs (cf. Sect. 5) based on the operations contained within it so as to improve the time, and then to collect the sub-results from RDDs and merge them to construct the answer to the initial query. Ultimately, a fine-tuning direction would be to utilize modified indexes such as R-trees, Quad-trees, kD-trees and LSM-trees, which have been already implemented, for integration with this work rather than using the MongoDB B-tree index.

References

1. Abdel-Fattah, M.A., Mohamed, W., Abdelgaber, S.: A comprehensive spark-based layer for converting relational databases to NoSQL. Big Data Cogn. Comput. **6**(3), 71 (2022). https://doi.org/10.3390/bdcc6030071
2. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.-C.N.: A survey of RDF stores & SPARQL engines for querying knowledge graphs. VLDB J. **31**, 1–26 (2021). https://doi.org/10.1007/s00778-021-00711-3

3. Anusha, K., Usha Rani, K.: Performance evaluation of spark SQL for batch processing. In: Venkata Krishna, P., Obaidat, M.S. (eds.) Emerging Research in Data Engineering Systems and Computer Communications. AISC, vol. 1054, pp. 145–153. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-0135-7_13
4. Apache: Hadoop. https://hadoop.apache.org/. Accessed 17 Jan 2023
5. Apache: HBase. http://hbase.apache.org/. Accessed 17 Jan 2023
6. Apache: Spark. https://spark.apache.org/. Accessed 17 Jan 2023
7. Apache: Storm. https://storm.apache.org/. Accessed 17 Jan 2023
8. Babcock, B., Chaudhuri, S.: Towards a robust query optimizer: a principled and practical approach. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 119–130 (2005)
9. Behm, A., Behm, A., et al.: ASTERIX: towards a scalable, semistructured data platform for evolving-world models. Distrib. Parall. Databases **29**(3), 185–216 (2011)
10. Celesti, A., et al.: Information management in IoT cloud-based tele-rehabilitation as a service for smart cities: Comparison of NoSQL approaches. Measurement **151**, 107218 (2020). https://doi.org/10.1016/j.measurement.2019.107218
11. Chambers, C., et al.: Flumejava: easy, efficient data-parallel pipelines. ACM SIGPLAN Notices **45**(6), 363–375 (2010)
12. Chawla, T., Singh, G., Pilli, E.S., Govil, M.: Storage, partitioning, indexing and retrieval in big RDF frameworks: a survey. Comput. Sci. Rev. **38**, 100309 (2020). https://doi.org/10.1016/j.cosrev.2020.100309
13. Chen, Y., Özsu, M.T., Xiao, G., Tang, Z., Li, K.: GSmart: an efficient SPARQL query engine using sparse matrix algebra - full version. arXiv preprint arXiv:2106.14038 (2021)
14. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
15. Eyada, M.M., Saber, W., El Genidy, M.M., Amer, F.: Performance evaluation of IoT data management using MongoDB versus MySQL databases in different cloud environments. IEEE Access **8**, 110656–110668 (2020). https://doi.org/10.1109/ACCESS.2020.3002164
16. Gupta, A., Jain, S.: Optimizing performance of real-time big data stateful streaming applications on cloud. In: 2022 IEEE International Conference on Big Data and Smart Computing (BigComp), pp. 1–4 (2022). https://doi.org/10.1109/BigComp54360.2022.00010
17. Győrödi, C., Győrödi, R., Pecherle, G., Olah, A.: A comparative study: MongoDB vs. MySQL. In: 2015 13th International Conference on Engineering of Modern Electric Systems (EMES), pp. 1–6. IEEE (2015)
18. Isard, M., Yu, Y.: Distributed data-parallel computing using a high-level programming language. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 987–994 (2009)
19. Izenov, Y., Datta, A., Rusu, F., Shin, J.H.: COMPASS: Online sketch-based query optimization for in-memory databases. In: Proceedings of the 2021 International Conference on Management of Data, pp. 804–816 (2021)
20. Karras, A., Karras, C., Samoladas, D., Giotopoulos, K.C., Sioutas, S.: Query optimization in NoSQL databases using an enhanced localized R-tree index. In: Pardede, E., Delir Haghighi, P., Khalil, I., Kotsis, G. (eds.) Information Integration and Web Intelligence, pp. 391–398. Springer Nature Switzerland, Cham (2022)

21. Li, Z.: Geospatial big data handling with high performance computing: current approaches and future directions. In: Tang, W., Wang, S. (eds.) High Performance Computing for Geospatial Applications. GE, vol. 23, pp. 53–76. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47998-5_4
22. Makris, A., Tserpes, K., Andronikou, V., Anagnostopoulos, D.: A classification of NoSQL data stores based on key design characteristics. Procedia Comput. Sci. **97**, 94–103 (2016). https://doi.org/10.1016/j.procs.2016.08.284, 2nd International Conference on Cloud Forward: From Distributed to Complete Computing
23. Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., Kraska, T.: Bao: making learned query optimization practical. ACM SIGMOD Rec. **51**(1), 6–13 (2022)
24. Marcus, R., et al.: Neo: a Learned Query Optimizer. Proc. VLDB Endow. **12**(11), 1705–1718 (2019). https://doi.org/10.14778/3342263.3342644
25. Markl, V., Lohman, G.M., Raman, V.: LEO: An autonomic query optimizer for DB2. IBM Syst. J. **42**(1), 98–106 (2003)
26. Melnik, S., et al.: Dremel: interactive analysis of web-scale datasets. Proceed. VLDB Endow. **3**(1–2), 330–339 (2010)
27. MongoDB Inc.: MongoDB. https://www.mongodb.com/. Accessed 24 Dec 2022
28. Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z.: Big data analytics on Apache Spark. Int. J. Data Sci. Anal. **1**(3), 145–164 (2016). https://doi.org/10.1007/s41060-016-0027-9
29. Sellami, R., Defude, B.: Complex queries optimization and evaluation over relational and NoSQL data stores in cloud environments. IEEE Trans. Big Data **4**(2), 217–230 (2017)
30. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2010)
31. Thusoo, A., et al.: Hive-a petabyte scale data warehouse using Hadoop. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), pp. 996–1005. IEEE (2010)
32. Vaisman, A., Zimányi, E.: Recent Developments in Big Data Warehouses. In: Data Warehouse Systems. Data-Centric Systems and Applications, pp. 561–631. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-662-65167-4_15
33. Xin, R.S., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: SQL and rich analytics at scale. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 13–24 (2013)
34. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pp. 15–28 (2012)