ABSTRACT

As a part of achieving specific targets, business decision making involves processing and analyzing large volumes of data that leads to growing enterprise databases day by day. Considering the size and complexity of the databases used in today's enterprises, it is a major challenge for enterprises to re-engineer their applications that can handle large amounts of data. Compared to traditional relational databases, non-relational NoSQL databases are better suited for dynamic provisioning, horizontal scaling, significant performance, distributed architecture and developer agility benefits. This project is intended to implement an existing algorithm for data migration from MySQL to MongoDB and extending the same algorithm to improve its performance metric i.e., time taken for data migration. It evaluates the query processing time by executing the same set of queries in MySQL as well as MongoDB and represent results graphically by plotting data size vs query processing time. The time taken for data migration by the existing algorithm as well as that of the extended algorithm will be obtained.

INTRODUCTION

A relational database is meant for structured data, which is organized in the form of tables. This can lead to high complexity when it comes to unstructured data. It is known that SQL queries are not well suited for the object-oriented data structures which are used in most of the applications now. Another problem that relational databases cannot handle is related to an exponentially increasing amount of data, so called big data problem. Therefore, to address all the above stated issues we are motivated to migrate to a NoSQL database. NoSQL databases deals well with both structured and unstructured data thus providing a framework to avoid scalability and distributed database issues. MongoDB is schema-less in which structure of a single object is clear, no complex joins, deep query ability (MongoDB supports dynamic queries on documents using a document-based query language that is almost as powerful as SQL) , tuning, ease to scale-out, conversion/mapping of application objects to database objects is not needed and uses internal memory for storing the working set, enabling faster access to data as referred in [5]. This motivated us to choose MongoDB for NoSQL database.

The report is organized in the following way: The section Related work, discusses the research work that has been explored for the project and the algorithm cited in [Chao sien, 2015]. The section Work and Results, presents our extended algorithm for mapping and migrating a relational database to NOSQL, the application of this algorithm and the results of the algorithm. The conclusion and potential future work is discussed in the last section of this report followed by the references.

RELATED WORK

A. SQL versus NoSQL

[Boicea, 2012] deals with a comparison between MongoDB, a document-oriented NoSQL system, and Oracle Database, an object-relational Database Management System. The main differences between SQL and NoSQL systems are described, specifically schema and model disparities (relational database model versus flexible document-oriented model), the query and syntax differences (SQL versus API calls, Javascript and REST). The authors used datasets from various entities like craigslist, Disney interactive media group, Sourceforge, The New York Times etc. Then, performance comparisons were conducted for INSERT, UPDATE and DELETE operations for different sizes of datasets. These comparisons show that MongoDB is more efficient for inserting huge amounts of data than Oracle, although MongoDB needs more time than Oracle to insert small amounts of data.

Furthermore, the performance of updating in MongoDB is nearly unaffected by the number of updated records, while Oracle's response time increases with the number of updated records. Being specific, Oracle took 94 milliseconds, 1343 milliseconds and 27782 milliseconds to update 10000, 100000 and 1000000 records respectively, while MongoDB took a time of 1, 2 and 3 milliseconds for the same number of records as results obtained from [Boicea, 2012]. The same phenomenon is recapitulated while deleting records. MongoDB shows almost a constant response time, while Oracle's response time is again increasing. The authors then concluded their analysis by stating that MongoDB works great for applications that need flexible data structures, while SQL fulfills the requirements for a complex but rigid data structure. [Chao-Hsien, 2015] also covers a short part comparing SQL to NoSQL systems and gets more into detail when it comes to the benefits of NoSQL systems for Content Management Systems (CMS) like WordPress, Joomla.

The authors state that NoSQL systems support the trend to run CMS in a cloud-based environment due to their ability to scale 5 horizontally, which means “processing several instances on different servers simultaneously” ([Chao-Hsien, 2015]). Due to the increasing popularity of NoSQL systems, there are tons of other papers that deal with the comparison of SQL and NoSQL systems.

B. Transforming a schema from SQL to NoSQL

Liana et al. [8] presented a framework that implements an algorithm of automatic mapping of MySQL to MongoDB. The algorithm uses the MySQL INFORMATION_SCHEMA that provides access to database metadata and represents the entity relationship in MongoDB using embedding and linking strategies (structure mode). To implement the existing algorithm, we require information about foreign keys for which we must access metadata from INFORMATION_SCHEMA.TABLES and REFERENTIAL_CONSTRAINTS as referred in [10].

Steps involved in the Algorithm:

1. Get the structure mode from the user
2. If structure mode is Linking, create collection for each table in the relational database using create collection. Get each row of child record and add the record in the document with the ID same as its foreign key
3. If structure mode is Embedding, create collection for the table that has the parent key i.e, the table which has the parent for the foreign key. Get the child record and insert as array in the document of the collection which has the parent key.

The given algorithm is shown in Figure 1:

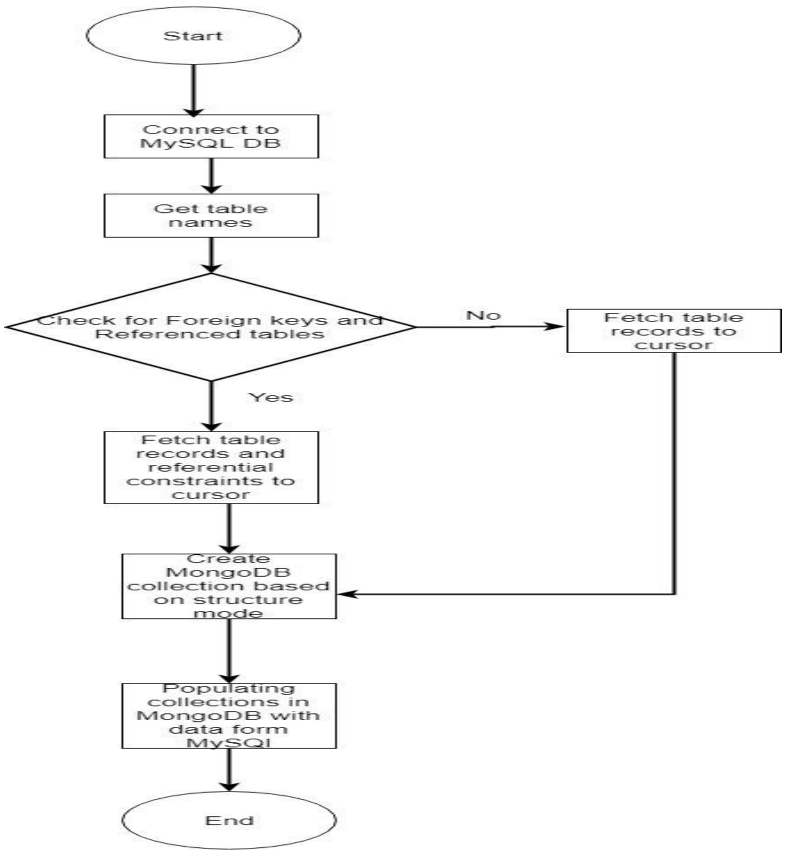

Start
Connect to MySQL DB
Get table names
Check for Foreign keys and Referenced tables
No
Fetch table records to cursor
Yes
Fetch table records and referential constraints to cursor
Create MongoDB collection based on structure mode
Populating collections in MongoDB with data form MySQl
End

Figure 1: Existing Algorithm

But we have come across a few shortcomings with the algorithm [Chao-Hsien, 2015] as noted below:

- In this algorithm, the type of structure mode (embedding/linking) must be explicitly specified.
- The same structure mode will be followed for the creation of collections in MongoDB without considering the relationship between the tables in relational database.

The extended algorithm will overcome these drawbacks by considering the six conditions for creating the collections in MongoDB which will be discussed in detail in the extended section.

EXTENDED WORK

In the extended algorithm, the collections are created using the relationship between the tables in relational database. In the extended algorithm, the use of metadata plays a crucial role. To get the metadata from the MySQL database, the INFORMATION_SCHEMA is used. The data gathered would be analyzed to understand the schema of the afore mentioned database.

The method of embedding, which is to infuse data from one table into another in the collections, is used in the algorithm through which the relationship criteria is retained throughout the process of migration and into the MongoDB collections, moving away from the algorithm cited in [Chao-Hsien, 2015], which uses the data structure linked lists.

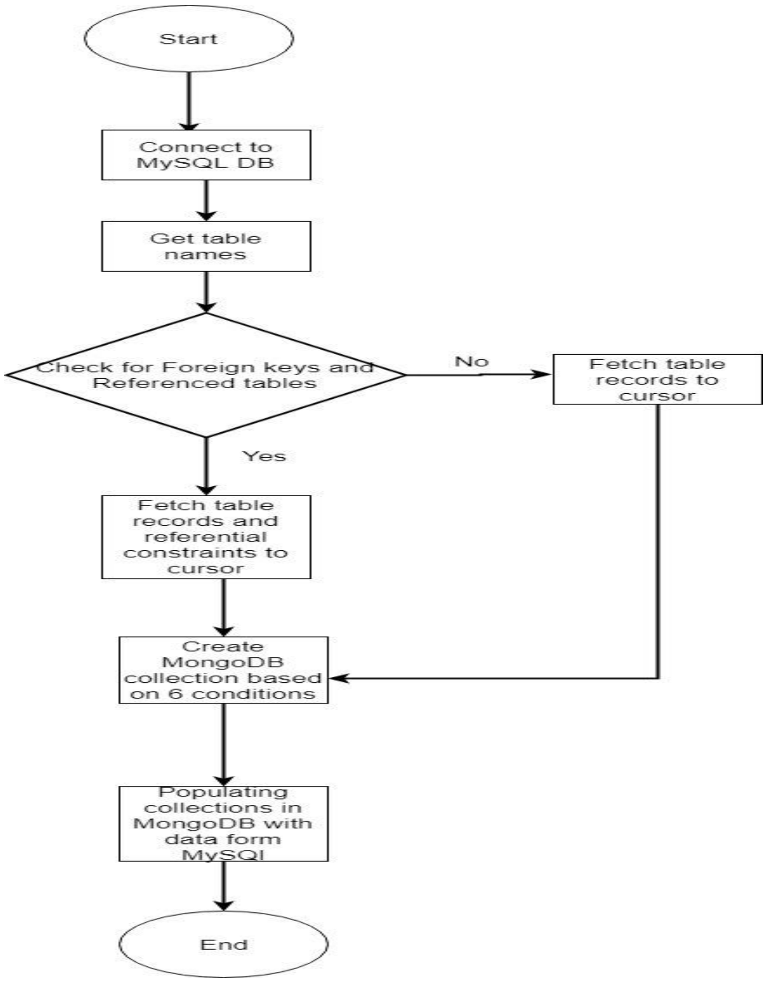

Start
Connect to MySQL DB
Get table names
Check for Foreign keys and Referenced tables
No
Fetch table records to cursor
Yes
Fetch table records and referential constraints to cursor
Create MongoDB collection based on 6 conditions
Populating collections in MongoDB with data form MySQl
End

Figure 2: Extended Algorithm

Conditions to be satisfied for creating MongoDB Collections: (6 conditions as mentioned in flow chart)

1. If the table is not referred by other tables, it will be represented by a new MongoDB collection.
2. If the table has no foreign keys, but is referred by another table, it will be represented by a new MongoDB collection.
3. If the table has one foreign key and is referenced by another table, it will be represented by a new MongoDB collection. In our framework, for this type of tables we use linking method, using the same concept of foreign key.
4. If the table has one foreign key but is not referred by another table, the algorithm uses one way embedding model. So, the table is embedded in a collection that represents the table from which the foreign key was present.
5. If the table has two foreign keys and is not referred by another table, it will be represented using the two-way embedding model.
6. If the table has three or more foreign keys, so it is the result of a N:M ternary, quaternary relationships, the algorithm uses a linking model, with foreign keys that refer all the tables initially implied in that relationship and already represented as MongoDB collections.

IMPLEMENTAION STRATEGY

The extended algorithm is presented in the following steps:

1) User should specify the MySQL database which need to be migrated.
2) The schema from the MySQL database is gathered from INFORMATION_SCHEMA explained in [MySQL 5.7 Reference Manual] which maintained in the MySQL server.

The information such as the referential constraints, data type of the columns and such, can be gathered.

3) Once the schema details are obtained, the algorithm then checks each table for foreign keys and/or tables which refer to this table.
4) The tables which are not referred by any other tables and have no foreign keys, form a MongoDB collection.
5) Tables which are referred by other tables but have no foreign keys form a MongoDB collection.
6) A table with a single foreign key and references from other tables exist then a MongoDB collection is created.
7) For a table with one foreign key and no other table referencing it, one-way embedding defined in the description above is used, where the current table is embedded into the table referencing it.
8) For a table with two foreign keys and no other table referencing it, the two-way embedding method defined in the description above is used.
9) For a table with three or more foreign keys Multiway embedding method defined in the description above is used.

TOOLS AND SOFTWARES USED:

We used Python for developing and implementing the extended algorithm and the algorithm cited in [Chao-Hsien, 2015]. Python modules like pymongo, MongoClient, MySQLdb are also used. The latest versions of mongoDB (MongoDB v3.6.2) and MySQL (MySQL 5.7.18.) are used for the performance testing.

RESULTS

The employees dataset which is collected from MySQL has six relations which is described below:

- employees – (emp_no,birth_date,first_name,last_name,gender,hire_date)
- departments – (dept_no,dept_name)
- dept_emp – (emp_no,dept_no,from_date,to_date)
- dept-manager – (dept_no,emp_no,from_date,to_date)
- salaries – (emp_no,salary,from_date,to_date)
- titles – (emp_no,title,from_date,to_date)

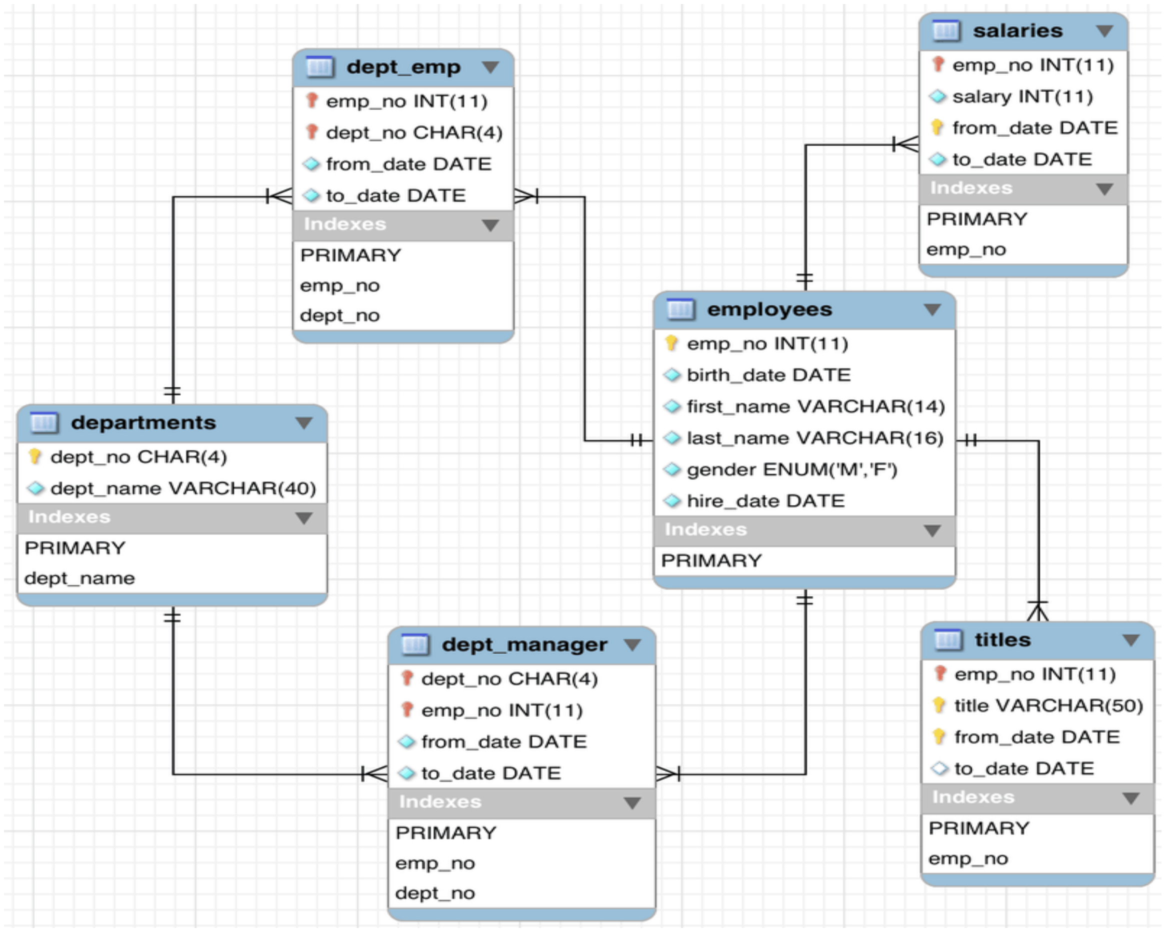
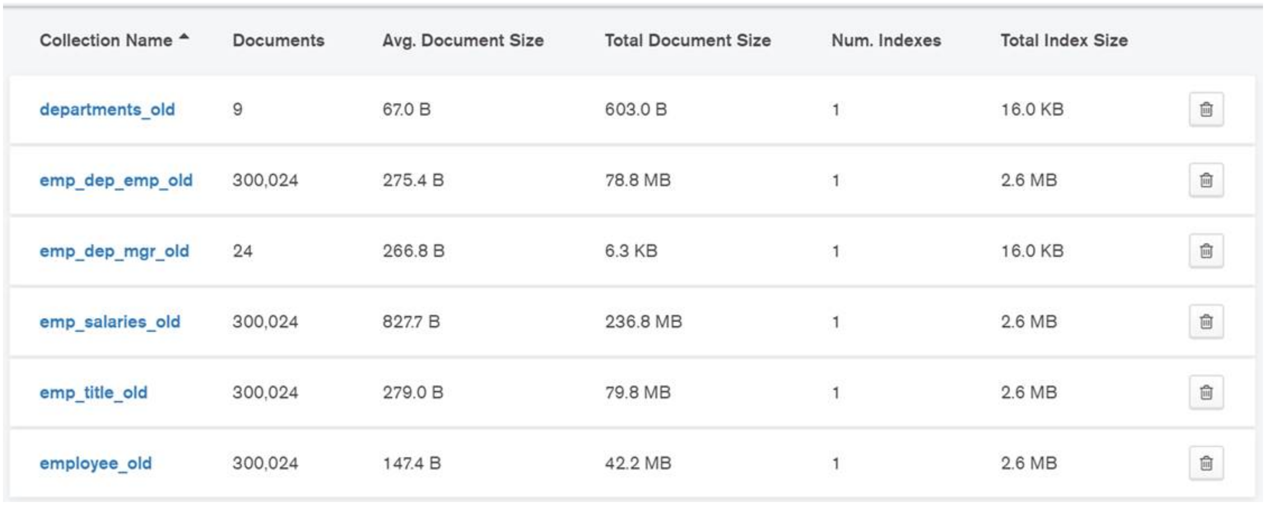


Figure 3: Database schema for Employee dataset

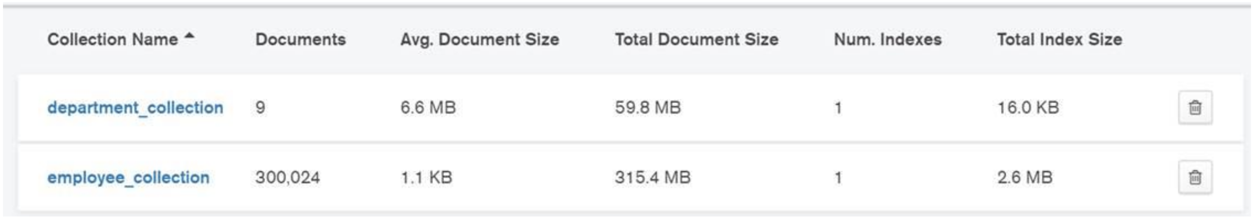
CORRESPONDING COLLECTIONS IN MONGODB

EXISTING ALGORITHM



Collection Name	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size
departments_old	9	67.0 B	603.0 B	1	16.0 KB
emp_dep_emp_old	300,024	275.4 B	78.8 MB	1	2.6 MB
emp_dep_mgr_old	24	266.8 B	6.3 KB	1	16.0 KB
emp_salaries_old	300,024	827.7 B	236.8 MB	1	2.6 MB
emp_title_old	300,024	279.0 B	79.8 MB	1	2.6 MB
employee_old	300,024	147.4 B	42.2 MB	1	2.6 MB

EXTENDED ALGORITHM

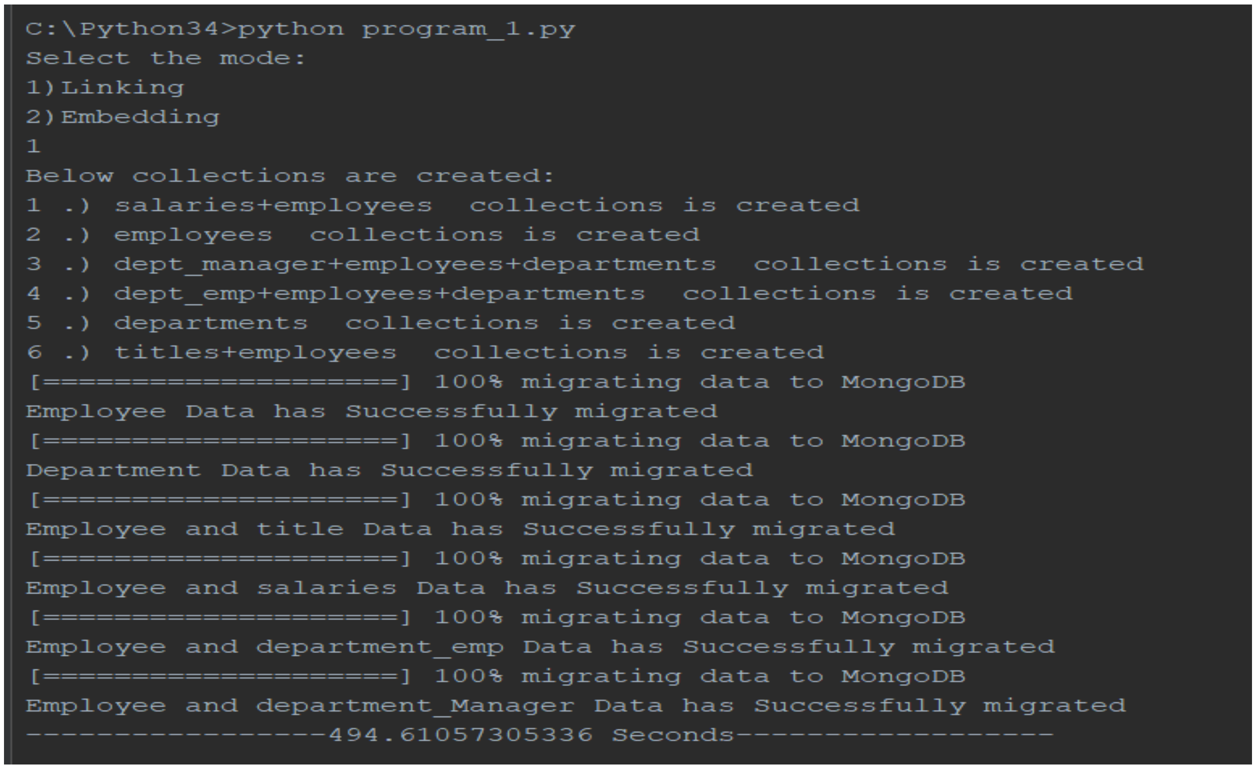


Collection Name	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size
department_collection	9	6.6 MB	59.8 MB	1	16.0 KB
employee_collection	300,024	1.1 KB	315.4 MB	1	2.6 MB

DESCRIPTION OF THE RELATIONS

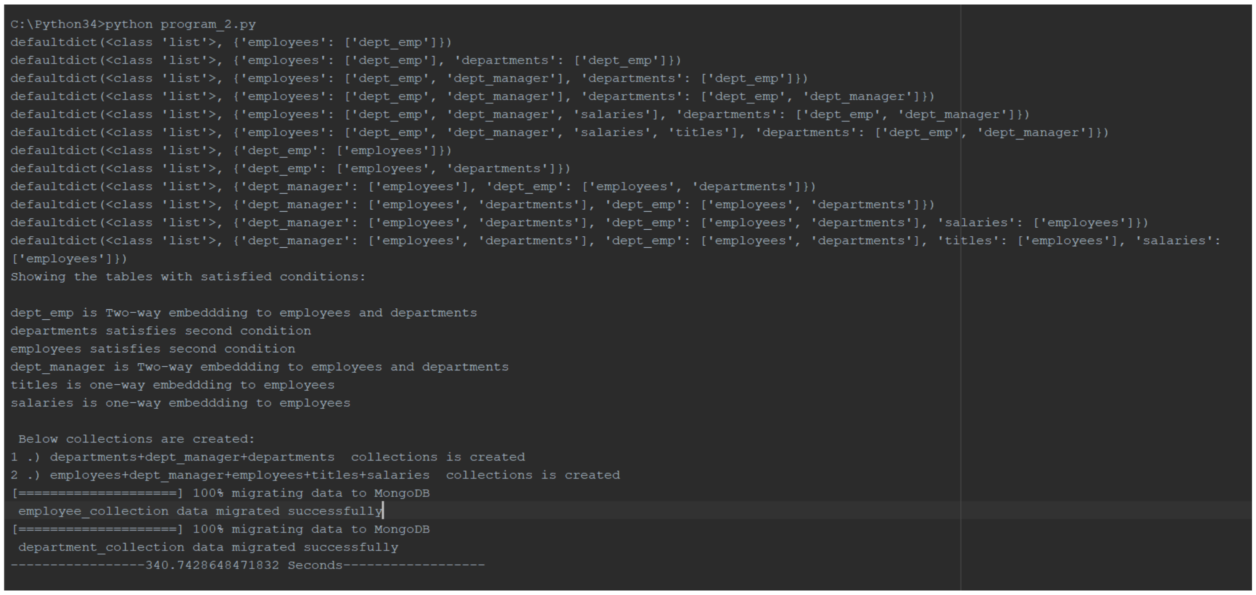
The employees table stores the emp_no, their first and last name, gender, birth_date and hire_date. The department table has the unique dept_no and dept_name. Each employee works for a department and this data such as employee number and the department to which they belong along with their start and end dates is stored. Each department has a manager and their term start and end as manager is stored. Each employees salary for each working month is stored. The title of each employee is stored along with their title term start and end date.

Execution of the existing algorithm:



```
C:\Python34>python program_1.py
Select the mode:
1)Linking
2)Embedding
1
Below collections are created:
1 .) salaries+employees  collections is created
2 .) employees  collections is created
3 .) dept_manager+employees+departments  collections is created
4 .) dept_emp+employees+departments  collections is created
5 .) departments  collections is created
6 .) titles+employees  collections is created
[====================] 100% migrating data to MongoDB
Employee Data has Successfully migrated
[====================] 100% migrating data to MongoDB
Department Data has Successfully migrated
[====================] 100% migrating data to MongoDB
Employee and title Data has Successfully migrated
[====================] 100% migrating data to MongoDB
Employee and salaries Data has Successfully migrated
[====================] 100% migrating data to MongoDB
Employee and department_emp Data has Successfully migrated
[====================] 100% migrating data to MongoDB
Employee and department_Manager Data has Successfully migrated
-----------------494.61057305336 Seconds------------------
```

Execution of the extended algorithm:



```
C:\Python34>python program_2.py
defaultdict(<class 'list'>, {'employees': ['dept_emp']})
defaultdict(<class 'list'>, {'employees': ['dept_emp'], 'departments': ['dept_emp']})
defaultdict(<class 'list'>, {'employees': ['dept_emp', 'dept_manager'], 'departments': ['dept_emp']})
defaultdict(<class 'list'>, {'employees': ['dept_emp', 'dept_manager'], 'departments': ['dept_emp', 'dept_manager']})
defaultdict(<class 'list'>, {'employees': ['dept_emp', 'dept_manager', 'salaries'], 'departments': ['dept_emp', 'dept_manager']})
defaultdict(<class 'list'>, {'employees': ['dept_emp', 'dept_manager', 'salaries', 'titles'], 'departments': ['dept_emp', 'dept_manager']})
defaultdict(<class 'list'>, {'dept_emp': ['employees']})
defaultdict(<class 'list'>, {'dept_emp': ['employees', 'departments']})
defaultdict(<class 'list'>, {'dept_manager': ['employees'], 'dept_emp': ['employees', 'departments']})
defaultdict(<class 'list'>, {'dept_manager': ['employees', 'departments'], 'dept_emp': ['employees', 'departments']})
defaultdict(<class 'list'>, {'dept_manager': ['employees', 'departments'], 'dept_emp': ['employees', 'departments'], 'salaries': ['employees']})
defaultdict(<class 'list'>, {'dept_manager': ['employees', 'departments'], 'dept_emp': ['employees', 'departments'], 'titles': ['employees'], 'salaries':
['employees']})
Showing the tables with satisfied conditions:

dept_emp is Two-way embeddding to employees and departments
departments satisfies second condition
employees satisfies second condition
dept_manager is Two-way embeddding to employees and departments
titles is one-way embeddding to employees
salaries is one-way embeddding to employees

 Below collections are created:
1 .) departments+dept_manager+departments  collections is created
2 .) employees+dept_manager+employees+titles+salaries  collections is created
[====================] 100% migrating data to MongoDB
 employee_collection data migrated successfully
[====================] 100% migrating data to MongoDB
 department_collection data migrated successfully
-----------------340.7428648471832 Seconds------------------
```

Thus, the time taken for data migration from MySQL to MongoDB by the extended algorithm is less compared to the time taken for migration by the existing algorithm.

EXAMPLE QUERIES:

MYSQL

Query 1 - Select count(*) from departments;

Query 2 - select s.salary from salaries s,employees e where s.emp_no=e.emp_no and e.first_name="Georgi";

Query 3- select count(*) ,dept_no from dept_emp group by dept_no having count(*)>1;

MONGODB

EXISTING ALGORITHM

Query 1 - db.departments_old.count();

Query 2 - db.employee_old.find({"first_name":"Georgi"},{"salaries":1});

Query 3 - db.employee_old.aggregate([{ $group: { _id: "$dept_emp.dept_no", count: { $sum: 1 } } }, { $match: { count: { $gt: 1 } } }]);

EXTENDED ALGORITHM

Query 1 - db.department_collection.count();

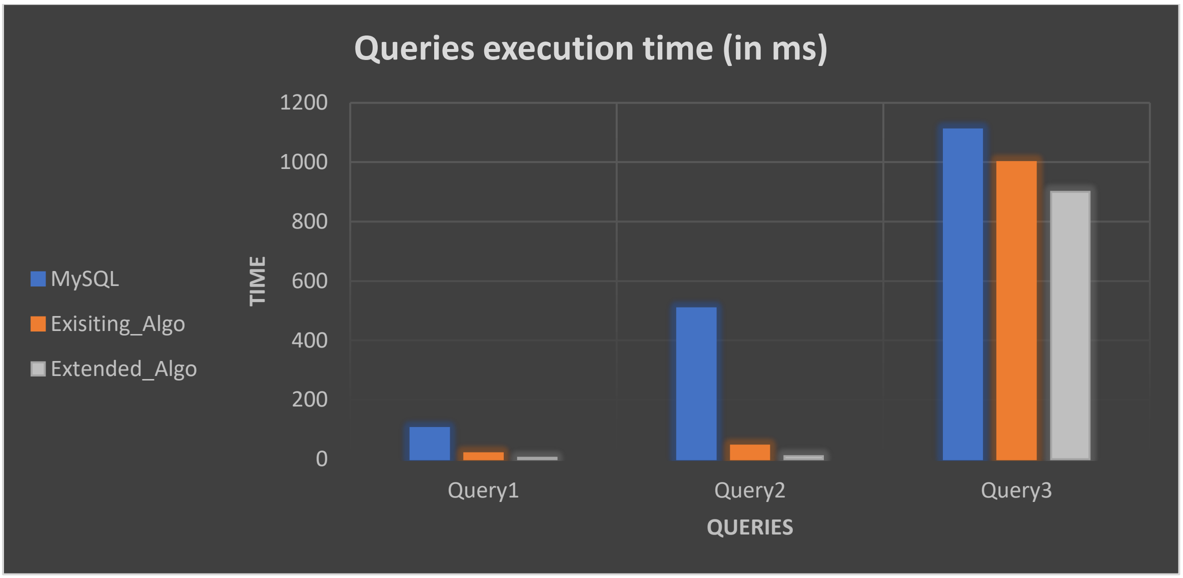
Query 2 - db.employee_collection.find({"first_name":"Georgi"},{"salaries":1});

Query 3 - db.employee_collection.aggregate([{ $group: { _id: "$dept_emp.dept_no", count: { $sum: 1 } } }, { $match: { count: { $gt: 1 } } }]);

GRAPHICAL REPRESENTATION OF THE QUERY EXECUTION TIME

The above queries were executed and the time for executing the queries were measured and recorded as follows,

Time for Query1	Time for Query2	Time for Query3	Database
104.6	508.4	1109.9	MySQL
20	46	1000	Existing Algorithm
5	10	900	Extended Algorithm



The results are varying for the existing and extended algorithm and the extended algorithm took less time for execution when compared to the existing algorithm execution time. The query 3 involves group by expression which is performed on dept_emp relation (331603 records) so it takes more execution time.

RANGE QUERIES

MySQL

Query 1 - SELECT * FROM employees WHERE emp_no =234432;

Query 2 - SELECT * FROM employees e, salaries s WHERE e.emp_no =s.emp_no AND s.salary =105471;

Query 3 - SELECT * FROM employees e,departments d, dept_emp de WHERE e.emp_no = de.emp_no AND de.dept_no = d.dept_no AND d.dept_name='Sales' ;

MongoDB

EXISTING ALGORITHM

Query 1 - db.employee_old.find({emp_no:234432});

Query 2 - db.emp_salaries_old.find({"salaries.salary" :105471});

Query 3 - db.emp_dep_emp_old.find({"dept_emp.dept_name":"Sales"});

EXTENDED ALGORITHM

Query 1 - db.employee_collection.find({emp_no:234432});

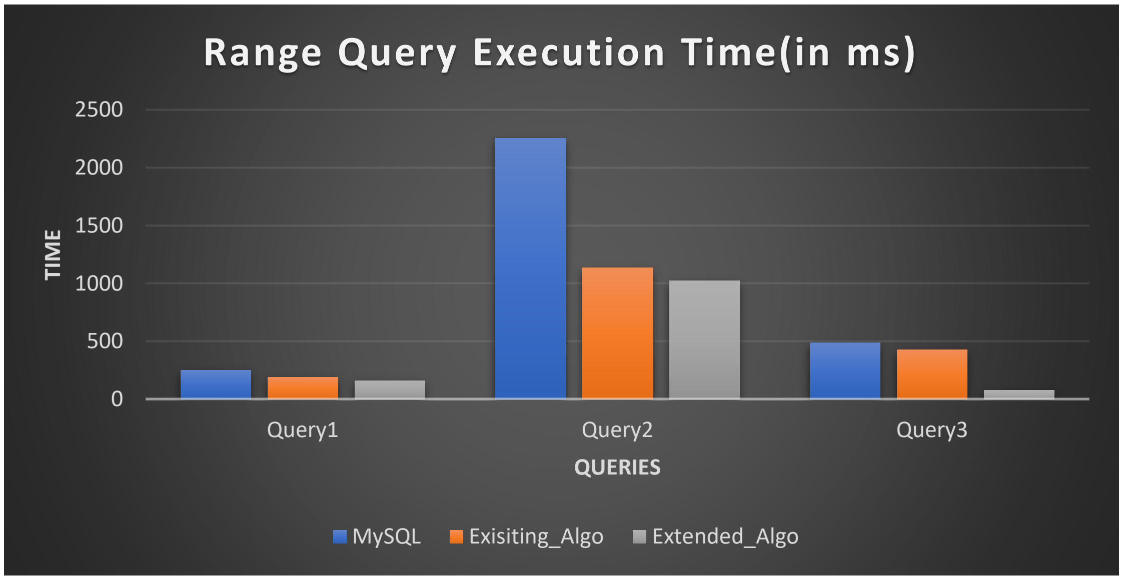
Query 2 - db.employee_collection.find({"salaries.salary" :105471});

Query 3 - db.department_collection.find({"dept_name":"Sales"});

GRAPHICAL REPRESENTATION OF RESULTS FOR RANGE QUERIES

The above queries were executed and the time for executing the queries were measured and recorded as follows,

Time for Query1	Time for Query2	Time for Query3	Database
251	2253	487	MySQL
190	1133	428	Existing Algorithm
158	1022	77	Extended Algorithm



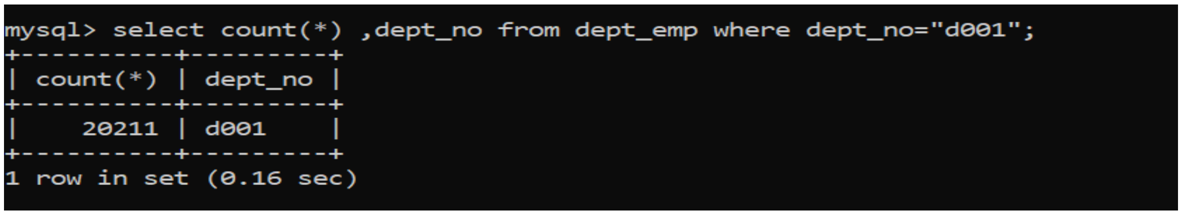
The query performance for all the queries is maximum in the MySQL environment and minimum in the extended algorithm. Since the second query is dealing with the large amount of data in the salaries table in MySQL, it varies with such significant difference.

The two-way embedding helped create only two collections compared to creating a collection for each table present in the MySQL database which reduced the time taken to execute both the lookup and range queries with a least time possible with the MongoDB environment created with the extended algorithm.

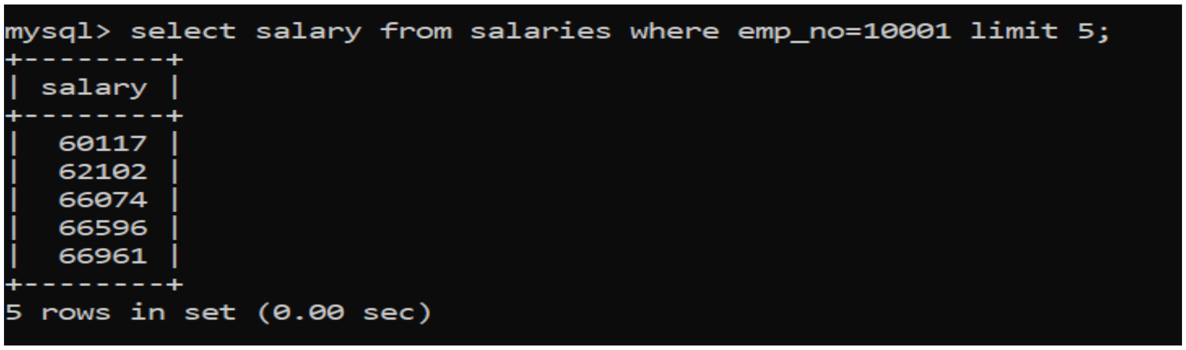
VERIFYING THE CORRECTNESS OF THE ALGORITHM

The correctness is measured by executing the same set of retrieval queries, approximately 25 times, in the MySQL database as well as the database created by the existing algorithm and the extended algorithm. Verify the displayed result by comparing the results retrieved.

MySQL



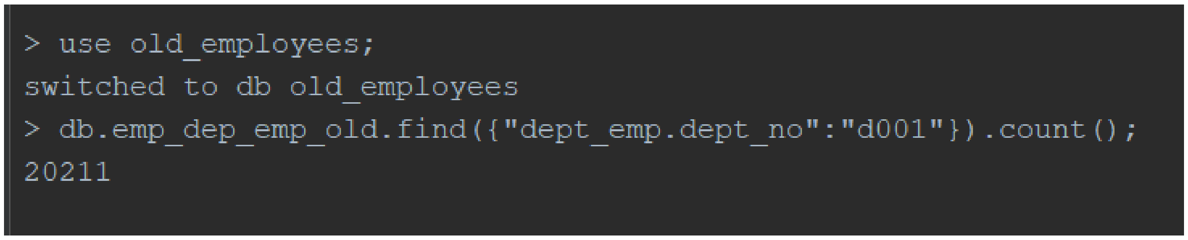
```
mysql> select count(*) ,dept_no from dept_emp where dept_no="d001";
+----------+---------+
| count(*) | dept_no |
+----------+---------+
|    20211 | d001    |
+----------+---------+
1 row in set (0.16 sec)
```



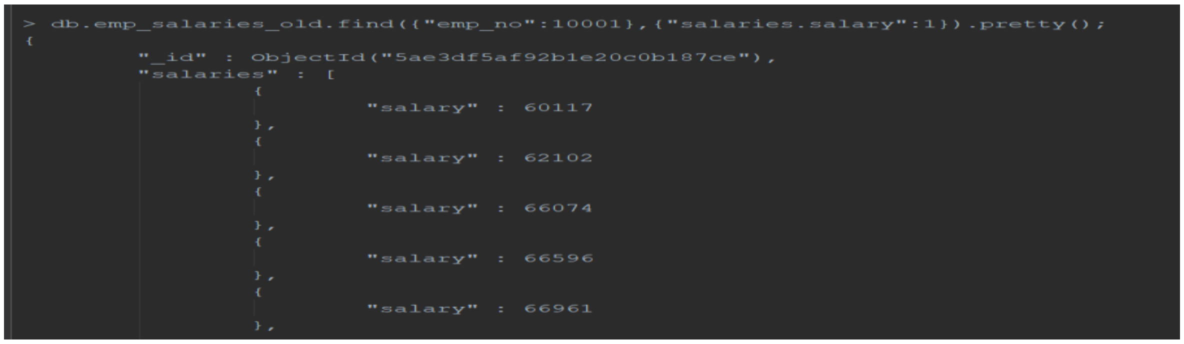
```
mysql> select salary from salaries where emp_no=10001 limit 5;
+--------+
| salary |
+--------+
|  60117 |
|  62102 |
|  66074 |
|  66596 |
|  66961 |
+--------+
5 rows in set (0.00 sec)
```

MONGODB

EXISTING ALGORITHM

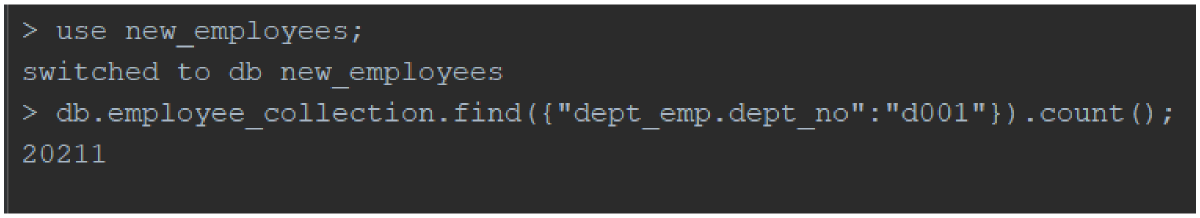


```
> use old_employees;
switched to db old_employees
> db.emp_dep_emp_old.find({"dept_emp.dept_no":"d001"}).count();
20211
```

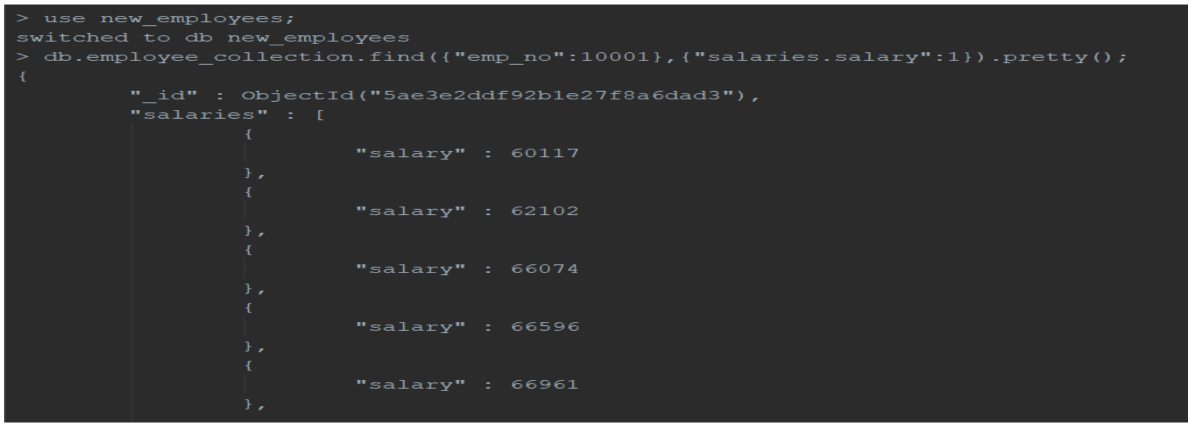


```
> db.emp_salaries_old.find({"emp_no":10001},{"salaries.salary":1}).pretty();
{
        "_id" : ObjectId("5ae3df5af92b1e20c0b187ce"),
        "salaries" : [
                {
                        "salary" : 60117
                },
                {
                        "salary" : 62102
                },
                {
                        "salary" : 66074
                },
                {
                        "salary" : 66596
                },
                {
                        "salary" : 66961
                },
```

EXTENDED ALGORITHM



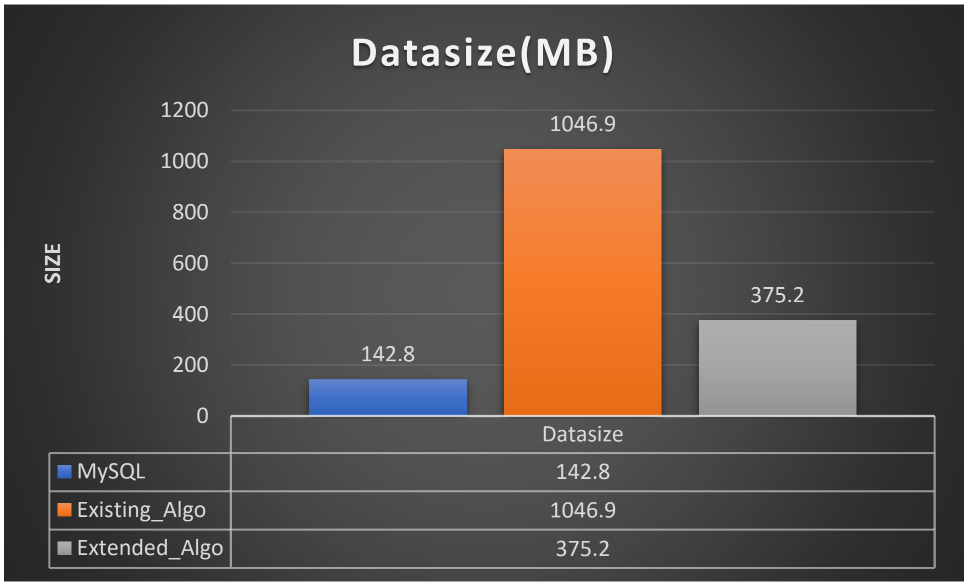
```
> use new_employees;
switched to db new_employees
> db.employee_collection.find({"dept_emp.dept_no":"d001"}).count();
20211
```



```
> use new_employees;
switched to db new_employees
> db.employee_collection.find({"emp_no":10001},{"salaries.salary":1}).pretty();
{
        "_id" : ObjectId("5ae3e2ddf92b1e27f8a6dad3"),
        "salaries" : [
                {
                        "salary" : 60117
                },
                {
                        "salary" : 62102
                },
                {
                        "salary" : 66074
                },
                {
                        "salary" : 66596
                },
                {
                        "salary" : 66961
                },
```

DATASIZE COMPARISON RESULTS

The data size before migration in MySQL database is measured far lower than the data size measured after migration in MongoDB. Though the extended algorithm occupies lesser space than the existing algorithm in this test case, the results might vary depending on the number of foreign keys and number of referenced tables present in the MySQL database. The total number of mongoDB objects created when the existing algorithm is used is 1.2 million but the mongoDB objects created using the extended algorithm is 0.3 million.



	Datasize
🟦 MySQL	142.8
🟧 Existing_Algo	1046.9
⬜ Extended_Algo	375.2

CONCLUSION AND FUTURE WORK:

The implementation of the extended algorithm proved that it is better than the algorithm cited in [Chao-Hsien, 2015] in terms of performance. Though the amount of space occupied by the data after the migration to mongoDB is significantly higher than its MySQL counterpart, the efficiency in terms of the time taken for execution of queries varies significantly. The algorithm and the testing strategies after implementation, have been presented in detail. The results obtained from testing the algorithm have been encouraging.

In the future, using diverse types of data to test the algorithm with varying types of relationships would help further understand the efficiency of the algorithm. Using more complex datasets to measure the efficiency of the algorithm could provide more comparisons with other migration algorithms. Developing the algorithm, to migrate from other relational databases like Oracle, MS SQL Server and PostgreSQL.

REFERENCES:

[1] A. Boicea, F. Radulescu, and L.I. Agapin, "MongoDB vs Oracle --Database Comparison," Proceedings of the 3rd International Conference on Emerging Intelligent Data and Web Technologies (EIDWT), Sept. 2012, pp.330-335.

[2] Giridhara Gopalan M, Prasanna C, Srihari Krishna YV, Shanthini B, Arulkumar A, “MYSQL TO CASSANDRA CONVERSION ENGINE”, IEEE 3rd International Conference on Sensing, Signal processing and Security, Oct.2017, pp.503-508.

[3] Shady Hamouda, Zurinahni Zainol, “Document-Oriented Data Schema for Relational Database Migration to NoSQL”, International Conference on Big Data Innovations and Applications, Aug.2017, pp. 43-50.

[4] Uwe Hohenstein, “Supporting Data Migration between Relational and Object-Oriented Databases Using a Federation Approach”, Database Engineering and Application Symposium,2000 International, Aug.2002, pp.371-379.

[5] Tianyu Jia, Xiaomeng Zhao, Zheng Wang, Dahan Gong and Guiguang Ding, “Model Transformation and Data Migration from Relational Database to MongoDB”, IEEE International Congress on Big Data, Oct.2016, pp. 60-67.

[6] Girts Karnitis and Guntis Arnicans, “Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data Transformation”, 7th International Conference on Computational Intelligence, Communication Systems and Networks, Jun.2015, pp. 113-118.

[7] Alza A. Mahmood,” Automated Algorithm for Data Migration from Relational to NoSQL Databases”, Al-Nahrain Journal for Engineering Sciences, Feb.2018, pp. 60-65.

[8] Liana Stanescu, “Automatic Mapping of MySQL Databases to NoSQL MongoDB”, Proceedings of the Federated Conference on Computer Science and Information Systems, Nov.2016, pp. 837–840.

[9] Rocha, L., Vale, F., Cirilo, E., Barbosa, D., and Mourao, F., "A Framework for Migrating Relational Datasets to NoSQL", Procedia Computer Science, 2015, pp. 2593-2602.

[10] MySQL 5.7 Reference Manual, https://dev.mysql.com/doc/refman/5.7/en/.