DBMS Assignment

Supreeth Baliga, 180801

Data:

The corresponding files for the respective sections according to my roll number are as follows:

1. A-100.csv, B-100-3-4.csv
2. A-100.csv, B-100-5-0.csv
3. A-100.csv, B-100-10-3.csv
4. A-1000.csv, B-1000-5-0.csv
5. A-1000.csv, B-1000-10-0.csv
6. A-1000.csv, B-1000-50-0.csv
7. A-10000.csv, B-10000-5-3.csv
8. A-10000.csv, B-10000-50-0.csv
9. A-10000.csv, B-10000-500-1.csv

Size of Data of Input Files:

- A-100.csv: 100 tuples with 2 attributes, Total Size: 789 B
- A-1000.csv: 1000 tuples with 2 attributes, Total Size: 9.6 KB
- A-10000.csv: 10000 tuples with 2 attributes, Total Size: 116 KB
- B-10000-500-1.csv: 2532790 tuples with 3 attributes, Total Size: 57 MB
- B-10000-50-0.csv: 259148 tuples with 3 attributes, Total Size: 5.6 MB
- B-10000-5-3.csv: 34994 tuples with 3 attributes, Total Size: 735 KB
- B-1000-10-0.csv: 5902 tuples with 3 attributes, Total Size: 113 KB
- B-1000-50-0.csv: 26345 tuples with 3 attributes, Total Size: 526 KB
- B-1000-5-0.csv: 3449 tuples with 3 attributes, Total Size: 66 KB
- B-100-10-3.csv: 598 tuples with 3 attributes, Total Size: 11 KB
- B-100-3-4.csv: 249 tuples with 3 attributes, Total Size: 4.3 KB
- B-100-5-0.csv: 333 tuples with 3 attributes, Total Size: 5.8 KB

Configuration Details:

The laptop is dual booted with Windows 10 and Linux. Below are the details of the Linux operating system since this is what was used to run the databases.

OS: Ubuntu 18.04.5 LTS x86_64
Processor: Intel® Core™ i7-8550U CPU @ 1.80GHz × 8
RAM: 8046MB
HDD: 2TB
SDD: 128GB
GPU: AMD Radeon R7 M260/M265

Directory Structure:

DBMS_Assignment
- **data/** (Contains all the csv files)
- **scripts/**
    - **main.py** (Parses time_stats.txt of all databases and builds the avg, std. dev and graphs)
    - **ExecTimeTable.txt** (The table created by main.py of the execution times)
    - **Images/** (Contains all the created graphs from main.py)
    - **Sqlite3/**
        - **Sqlite3.sh** (Script for running all the queries on all 9 datasets for sqlite3)
        - **time_stats.text** (Contains the execution time of all queries done by Sqlite3.sh)
    - **MongoDB/**
        - **MongoDb.sh** (Script for running all the queries on all 9 datasets for MongoDB)
        - **time_stats.text** (Contains the execution time of all queries done by MongoDb.sh)
    - **MariaDB/**
        - **MariaDb.sh** (Script to run all the queries on all datasets for MariaDB without index)
        - **time_stats.text** (Contains the execution time of all queries done by MariaDb.sh)
    - **MariaDbIndex/**
        - **Maria_Db_Index.sh** (Script to run all the queries on 9 datasets for MariaDB with index)
        - **time_stats.text** (Contains execution times of all queries done by Maria_Db_Index.sh)

How to Run:

1. Go inside each of the folders (Sqlite3, MongoDB, MariaDB, MariaDbIndex) and run the corresponding bash files in the following way:
   "$ *bash Sqlite3.sh > time_stats.txt 2> dump.txt"*
   And similarly for other folders. This will create the time_stats.txt file in each of the folders. This is the relevant file. Other text files created are just for running purpose, we won't be using them for our analysis.
2. Come to the scripts/ folder and start a new virtual environment and install the following modules using pip:
    - numpy
    - matplotlib
    - tabulate
3. Once this is done. Run "$ *python main.py > ExecTimeTable.txt*". This will display the execution time table in the ExecTimeTable..txt file and create the relevant graphs in the Images folder.

Points to note:

1. I have created 4 bash scripts for each of the databases which notes down all the execution times for the 9 csv files in the time_stats.txt file of their corresponding folders. Then I have executed main.py to calculate the necessary mean and standard deviations and to plot graphs.
2. All the times are noted in seconds (in mongo, they are in ms, but then I have converted it to seconds while calculating average and standard deviation).
3. For MariaDB without index, I have not created the primary key and foreign key constraints to make sure that implicit indices are not created.

4. Instead of interleaving the queries, I have cleared the cache after each query so that each query is independent of the previous one. (Note that sqlite3 does not cache queries and hence, it wasn't needed in sqlite3.) This way each query is run independently and in a consistent environment.
5. The execution times include the time needed to print the result of the query to a file.
6. In each database, Query 2 sorting is done according to the ASCII value of the characters. Hence, I have added the keyword 'BINARY' in MariaDB (with and without index) to enforce the same.
7. As cleared by sir, I have added explicit indices only in MariaDB with index and not in Sqlite3.
8. All the queries were run in the linux terminal.
9. To see the techniques used to get the query times, please look at the respective bash scripts.
10. For creating an index on B3 in MariaDB, I assumed the max length of the string to be 32000 bytes.

Queries:

1. Sqlite3

a. Setting up the tables and importing

```
.mode csv
.headers on
.timer on
drop table if exists A;
create table A(
     A1 integer primary key,
     A2 text
);
.import ../../data/A-100.csv A
drop table if exists B;
create table B(
     B1 integer primary key,
     B2 integer,
     B3 text,
     foreign key (B2) references A(A1)
);
.import ../../data/B-100-3-4.csv B
```

b. Query 1

```
SELECT * FROM A WHERE A1 <= 50;
```

c. Query2

```
SELECT * FROM B ORDER BY B3 ASC;
```

d. **Query3**

```
SELECT AVG(vals) FROM
(SELECT B2, COUNT(B1) AS vals FROM B
GROUP BY B2);
```

e. **Query4**

```
SELECT B1, B2, B3, A2 FROM
(B INNER JOIN A ON B.B2 = A.A1);
```

2. **MariaDB without Index**
   a. **Setting up the tables and importing**

```
CREATE DATABASE IF NOT EXISTS test;
USE test;
drop table if exists B;
drop table if exists A;
create table A(
     A1 integer,
     A2 text
);
LOAD DATA LOCAL INFILE '../../data/A-100.csv'
INTO TABLE A
FIELDS TERMINATED BY ','
IGNORE 1 LINES;

create table B(
     B1 integer ,
     B2 integer,
     B3 text
);
LOAD DATA LOCAL INFILE '../../data/B-100-3-4.csv'
INTO TABLE B
FIELDS TERMINATED BY ','
IGNORE 1 LINES;
```

b. **Query 1**

```
SELECT * FROM A WHERE A1 <= 50;
```

c. **Query2**

```
SELECT * FROM B ORDER BY BINARY B3 ASC;
```

d. **Query3**

```
SELECT AVG(vals) FROM
(SELECT B2, COUNT(B1) AS vals FROM B
GROUP BY B2) AS K;
```

e. **Query4**

```
SELECT B1, B2, B3, A2 FROM (B INNER JOIN A ON B.B2
= A.A1);
```

3. **MariaDB with Index**
   a. **Setting up the tables and importing**

```
CREATE DATABASE IF NOT EXISTS test;
USE test;
drop table if exists B;
drop table if exists A;
create table A(
     A1 integer primary key,
     A2 varchar(32000)
);
LOAD DATA LOCAL INFILE '../../data/A-100.csv'
INTO TABLE A
FIELDS TERMINATED BY ','
IGNORE 1 LINES;
CREATE INDEX IF NOT EXISTS A_idx_1 on A(A2(32000));
create table B(
     B1 integer primary key,
     B2 integer,
     B3 varchar(32000),
     foreign key (B2) references A(A1) ON DELETE
CASCADE
);
LOAD DATA LOCAL INFILE '../../data/B-100-3-4.csv'
```

```
INTO TABLE B
FIELDS TERMINATED BY ','
IGNORE 1 LINES;
CREATE INDEX IF NOT EXISTS B_idx_1 on B(B2);
CREATE INDEX IF NOT EXISTS B_idx_2 on B(B3(32000));
# Note: Indexes not explicitly created for fields
which already have implicit index on them
```

b. Query 1

```
SELECT * FROM A WHERE A1 <= 50;
```

c. Query2

```
SELECT * FROM B USE INDEX(B_idx_2) ORDER BY BINARY B3
ASC;
```

d. Query3

```
SELECT AVG(vals) FROM
(SELECT B2, COUNT(B1) AS vals FROM B USE
INDEX(B_idx_1)
GROUP BY B2) AS K;
```

e. Query4

```
SELECT B1, B2, B3, A2 FROM
(B USE INDEX(B_idx_1,B_idx_2)
INNER JOIN A USE INDEX(A_idx_1) ON B.B2 = A.A1);
```

4. MongoDB

a. Setting up the tables and importing

```
use test
db.A.drop()
db.B.drop()
```

Bash:

```
mongoimport --type csv -d test -c A --headerline
../../data/A-100.csv
```

```
mongoimport --type csv -d test -c B --headerline
../../data/B-100-3-4.csv
```

b. Query 1

```
db.A.find({A1 : {$lte : 50}}, {A1: true, A2: true,
_id: false})
```

c. Query2

```
db.B.aggregate(
     [
     {$sort: {B3 : 1} },
     {$project: {
          _id: false,
          B1: true,
          B2: true,
          B3: true
     }}
     ], { "allowDiskUse" : true }
)
```

Note: `allowDiskUse is done to avoid Memory Limit Exceeded Error`

d. Query3

```
db.B.aggregate(
     [
     {$group: {_id: "$B2", total: {$sum: 1}}},
     {$group: {_id: null, answer: {$avg:
"$total"}}},
     {$project: {
          _id: false,
          answer: true
     }}
     ]
)
```

e. Query4

```
db.B.aggregate(
	[
	{$lookup: {
		from: "A",
		localField: "B2",
		foreignField: "A1",
		as: "temp"
	}},
	{$unwind: {
		path: "$temp",
		preserveNullAndEmptyArrays: false
	}},
	{$project:{
		_id: false,
		B1: true,
		B2: true,
		B3: true,
		A2: "$temp.A2"
	}}
	]
)
```

Execution Times:

I have included the text file which contains the required table of the average and standard deviations of the query runtimes (ExecTimeTable.txt). I'm attaching a screenshot of the same over here.

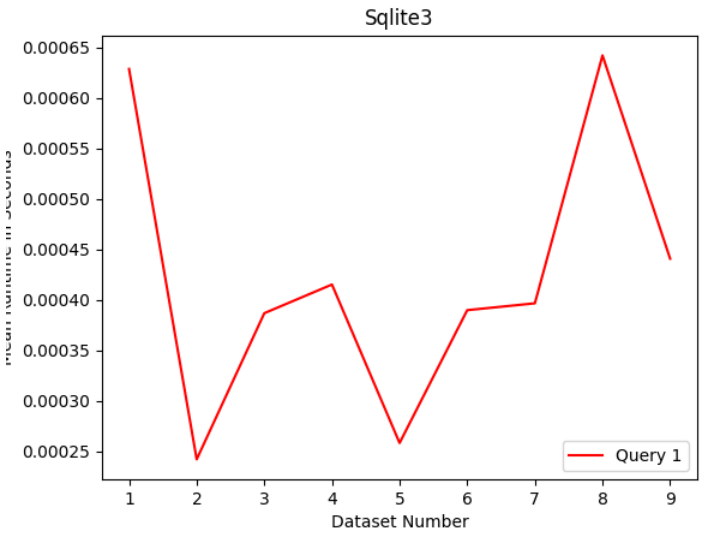
		1	2	3	4	5	6	7	8	9
Query 1	Sqlite3	Avg:0.000629 Dev:0.000045	Avg:0.000242 Dev:0.000019	Avg:0.000387 Dev:0.000193	Avg:0.000415 Dev:0.000149	Avg:0.000258 Dev:0.000052	Avg:0.000390 Dev:0.000203	Avg:0.000397 Dev:0.000161	Avg:0.000642 Dev:0.000016	Avg:0.000441 Dev:0.000107
Query 1	MariaDB(without index)	Avg:0.000437 Dev:0.000140	Avg:0.000358 Dev:0.000158	Avg:0.000442 Dev:0.000206	Avg:0.000926 Dev:0.000194	Avg:0.001069 Dev:0.000447	Avg:0.000966 Dev:0.000266	Avg:0.009111 Dev:0.002883	Avg:0.006487 Dev:0.004061	Avg:0.016261 Dev:0.002240
Query 1	MariaDB(with index)	Avg:0.000613 Dev:0.000202	Avg:0.000679 Dev:0.000022	Avg:0.000414 Dev:0.000054	Avg:0.000447 Dev:0.000200	Avg:0.000690 Dev:0.000029	Avg:0.000457 Dev:0.000044	Avg:0.000365 Dev:0.000173	Avg:0.000473 Dev:0.000150	Avg:0.000684 Dev:0.000030
Query 1	MongoDB	Avg:0.000000 Dev:0.000000	Avg:0.000200 Dev:0.000400	Avg:0.000000 Dev:0.000000	Avg:0.002200 Dev:0.000400	Avg:0.001600 Dev:0.000490	Avg:0.001800 Dev:0.000400	Avg:0.019200 Dev:0.004578	Avg:0.016600 Dev:0.001356	Avg:0.019000 Dev:0.001095
Query 2	Sqlite3	Avg:0.001074 Dev:0.000265	Avg:0.000600 Dev:0.000038	Avg:0.000904 Dev:0.000082	Avg:0.004569 Dev:0.000379	Avg:0.009133 Dev:0.000529	Avg:0.027107 Dev:0.005961	Avg:0.048162 Dev:0.004810	Avg:0.272296 Dev:0.006396	Avg:3.387400 Dev:0.504971
Query 2	MariaDB(without index)	Avg:0.001248 Dev:0.000237	Avg:0.001078 Dev:0.000037	Avg:0.002111 Dev:0.000634	Avg:0.010204 Dev:0.000820	Avg:0.017532 Dev:0.000740	Avg:0.128800 Dev:0.006053	Avg:0.208715 Dev:0.099069	Avg:4.634506 Dev:0.652406	Avg:598.669006 Dev:120.104970
Query 2	MariaDB(with index)	Avg:0.001233 Dev:0.000004	Avg:0.003139 Dev:0.000061	Avg:0.003364 Dev:0.000021	Avg:0.027032 Dev:0.006243	Avg:0.018249 Dev:0.001451	Avg:0.098529 Dev:0.004032	Avg:0.138778 Dev:0.012923	Avg:11.474452 Dev:1.260811	Avg:465.443281 Dev:56.994926
Query 2	MongoDB	Avg:0.001400 Dev:0.000490	Avg:0.001800 Dev:0.000400	Avg:0.002400 Dev:0.001020	Avg:0.020000 Dev:0.003950	Avg:0.029400 Dev:0.002332	Avg:0.097000 Dev:0.007563	Avg:0.135400 Dev:0.030037	Avg:0.858400 Dev:0.163818	Avg:5.142800 Dev:0.057763
Query 3	Sqlite3	Avg:0.000499 Dev:0.000034	Avg:0.000301 Dev:0.000014	Avg:0.000391 Dev:0.000010	Avg:0.001106 Dev:0.000006	Avg:0.002584 Dev:0.000381	Avg:0.005874 Dev:0.000456	Avg:0.017272 Dev:0.003906	Avg:0.067814 Dev:0.002633	Avg:0.712279 Dev:0.037158
Query 3	MariaDB(without index)	Avg:0.000549 Dev:0.000029	Avg:0.000623 Dev:0.000087	Avg:0.000687 Dev:0.000004	Avg:0.002675 Dev:0.000466	Avg:0.004394 Dev:0.000928	Avg:0.015940 Dev:0.009912	Avg:0.033841 Dev:0.011531	Avg:0.106185 Dev:0.010568	Avg:1.190861 Dev:0.025082
Query 3	MariaDB(with index)	Avg:0.000500 Dev:0.000008	Avg:0.000661 Dev:0.000347	Avg:0.001081 Dev:0.000155	Avg:0.003615 Dev:0.000065	Avg:0.005070 Dev:0.000026	Avg:0.016743 Dev:0.000034	Avg:0.006837 Dev:0.000204	Avg:0.041784 Dev:0.008903	Avg:0.481280 Dev:0.101399
Query 3	MongoDB	Avg:0.000600 Dev:0.000490	Avg:0.001400 Dev:0.000490	Avg:0.002000 Dev:0.000632	Avg:0.013600 Dev:0.001020	Avg:0.022400 Dev:0.004409	Avg:0.076600 Dev:0.009851	Avg:0.068400 Dev:0.005083	Avg:0.310800 Dev:0.008588	Avg:1.527000 Dev:0.010450
Query 4	Sqlite3	Avg:0.000626 Dev:0.000042	Avg:0.000453 Dev:0.000037	Avg:0.000676 Dev:0.000047	Avg:0.002555 Dev:0.000426	Avg:0.006140 Dev:0.001283	Avg:0.018565 Dev:0.003718	Avg:0.032616 Dev:0.001897	Avg:0.197352 Dev:0.020698	Avg:1.541680 Dev:0.022145
Query 4	MariaDB(without index)	Avg:0.002297 Dev:0.000136	Avg:0.003089 Dev:0.000118	Avg:0.004785 Dev:0.000201	Avg:0.184768 Dev:0.021838	Avg:0.290041 Dev:0.020386	Avg:1.187696 Dev:0.055453	Avg:16.173417 Dev:0.292429	Avg:115.316307 Dev:2.947362	Avg:1305.059231 Dev:240.390164
Query 4	MariaDB(with index)	Avg:0.001054 Dev:0.000017	Avg:0.000846 Dev:0.000018	Avg:0.003562 Dev:0.000184	Avg:0.011196 Dev:0.003695	Avg:0.016019 Dev:0.006574	Avg:0.058095 Dev:0.000642	Avg:0.077365 Dev:0.001927	Avg:0.295508 Dev:0.010720	Avg:5.564380 Dev:0.137387
Query 4	MongoDB	Avg:0.033600 Dev:0.002577	Avg:0.055000 Dev:0.011009	Avg:0.095400 Dev:0.026058	Avg:2.505000 Dev:0.189336	Avg:4.127400 Dev:0.035149	Avg:21.731800 Dev:2.367429	Avg:224.477000 Dev:4.274376	Avg:1068.205000 Dev:231.599288	Avg:9466.376200 Dev:26.402389

Trends and Graphs:

1. Comparing the execution times of the queries per individual database

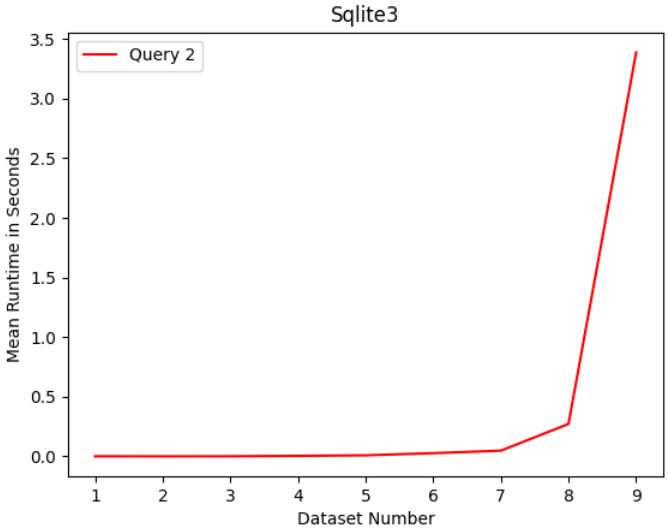
a. Sqlite3

i. For Query 1



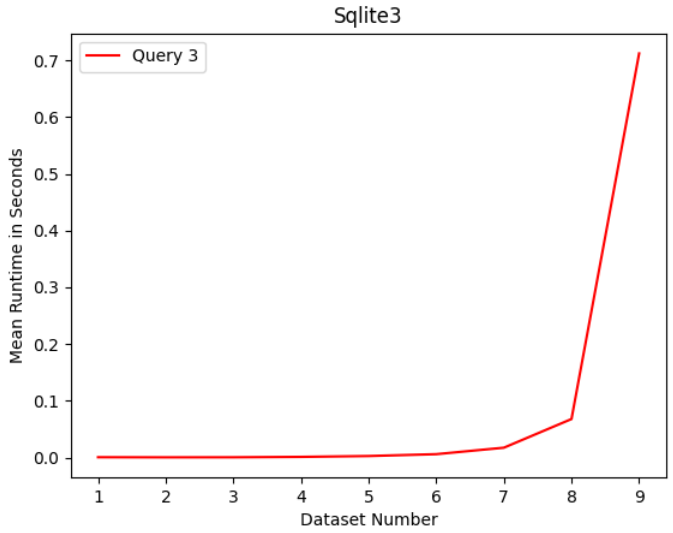
Inference: Since the query is based on the primary key which has an implicit index built upon it, the query time does not scale much according to the size of the data set. Secondly, since the query time is very less, even little interruptions from the system show a huge error in execution time as can be seen from the figure.

ii. For Query 2



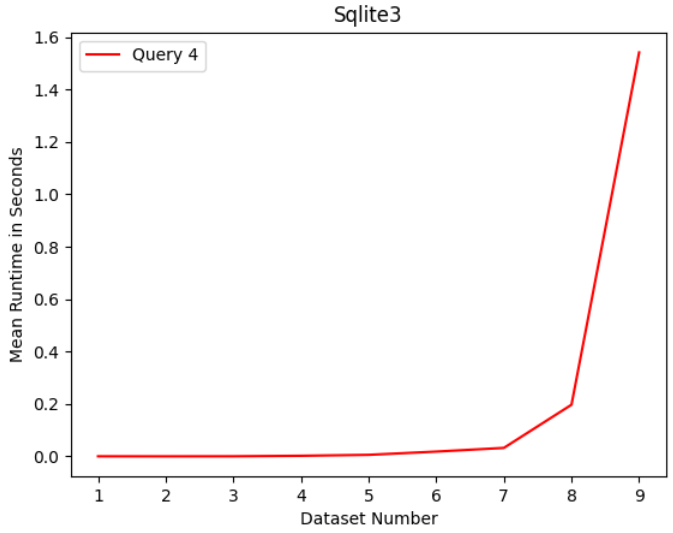
Inference: A clear trend can be seen that the execution time increases as the size of the data set increases.

iii. For Query 3



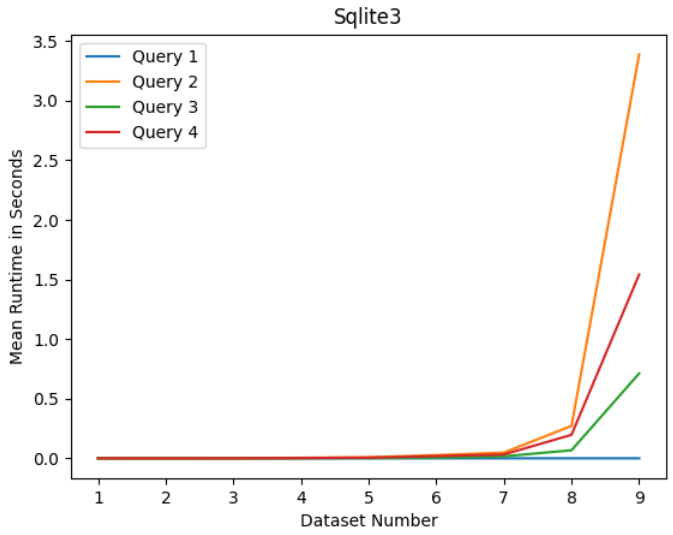
Inference: A clear trend can be seen that the execution time increases as the size of the data set increases.

iv. For Query 4



Inference: A clear trend can be seen that the execution time increases as the size of the data set increases.

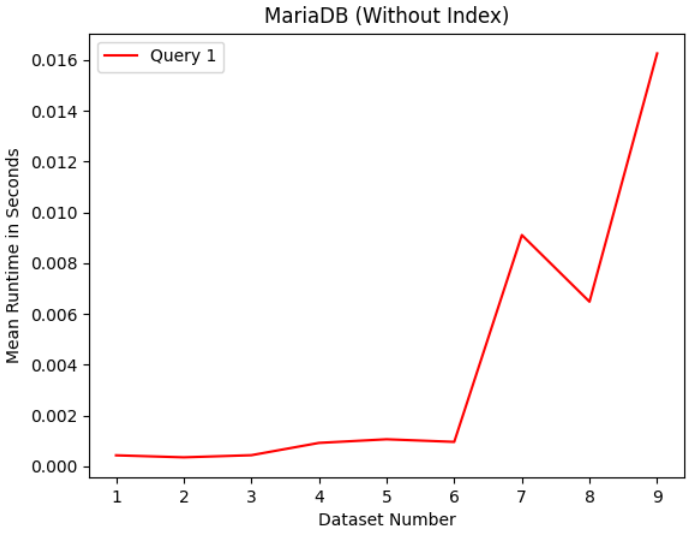
v. For all Queries comparison



Inference: We can clearly see a distinction between the execution times of the queries as the size of the dataset increases. Query1 takes the least time as it just involves the smaller dataset (A) and also just has to work with the already indexed primary key. Query 2 takes the most time since it involves sorting and SQLite3 is exactly not built for maintaining an order between the tuples since it is based on the relational algebra model which involves working with sets (multisets for SQL), although, the absolute runtimes are still not large. Query 3 again involves simple aggregation and hence does not take much time. Query 4 takes more time than 1 and 3 because it involves the complex join operation. However, it takes less time than query 2 since SQLite3 is built for maintaining relations between tables because of which join is in general, much faster than sorting since we have created the foreign key constraint.

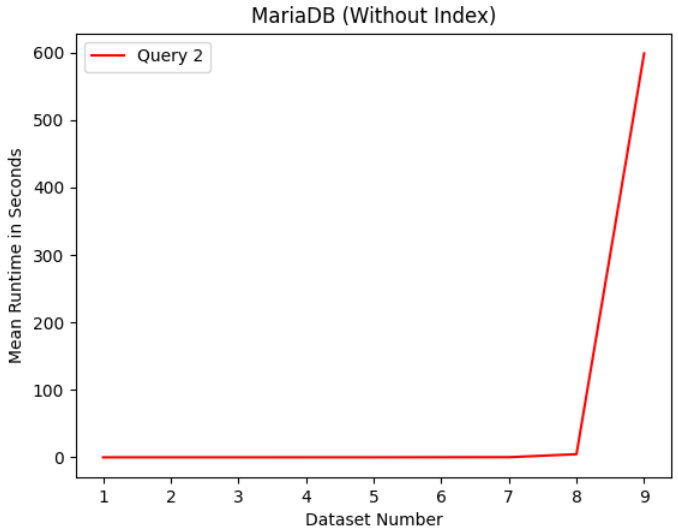
b. MariaDB (Without Index)

i. For Query 1



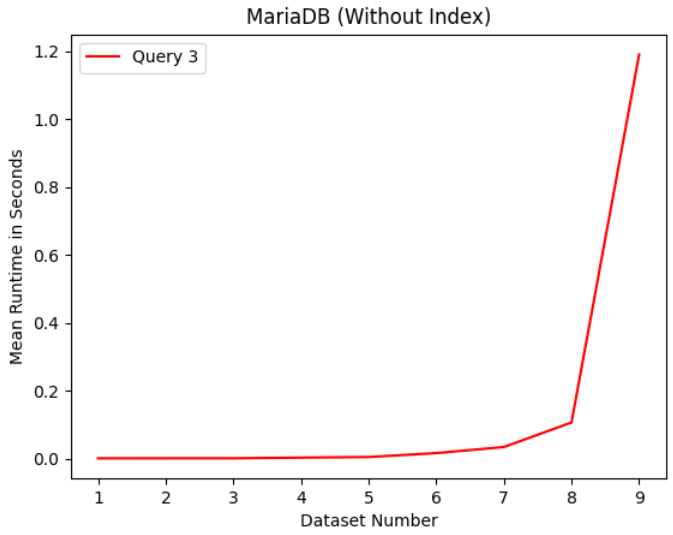
Inference: Since in this case, there is no primary key declared, a rough trend can be seen that the query time increases with the increase in size of the dataset.

ii. For Query 2



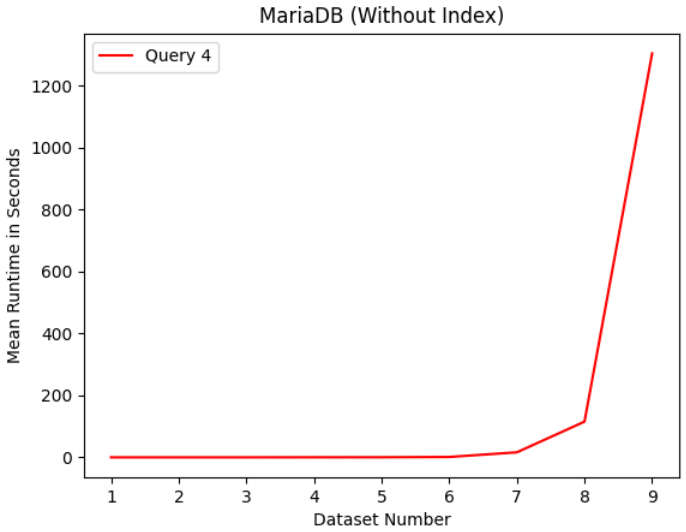
Inference: Similar to the sqlite3 case, execution time increases with increase in size of the dataset. Note that MariaDB by default sorts strings without discriminating between lowercase and uppercase characters. This takes a lot less time. However, since we have used the 'Binary' keyword to sort according to ASCII values, this is taking a lot more time.

iii. For Query 3



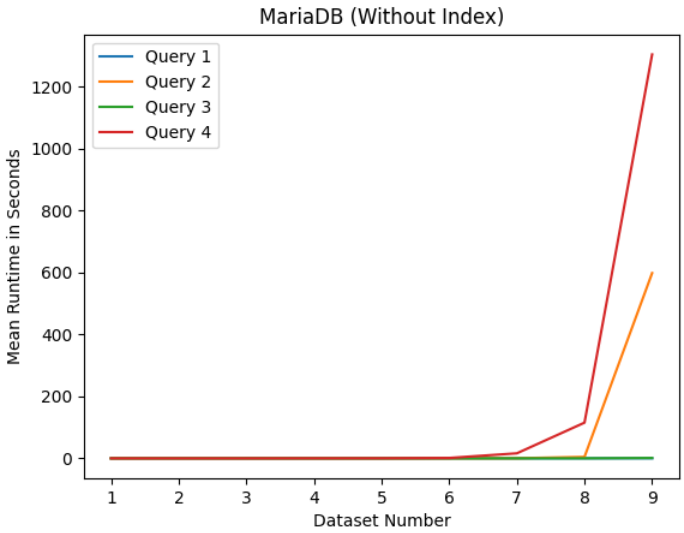
Inference: Similar to the sqlite3 case, execution time increases with increase in size of the dataset.

iv. For Query 4



Inference: Similar to the sqlite3 case, execution time increases with increase in size of the dataset.

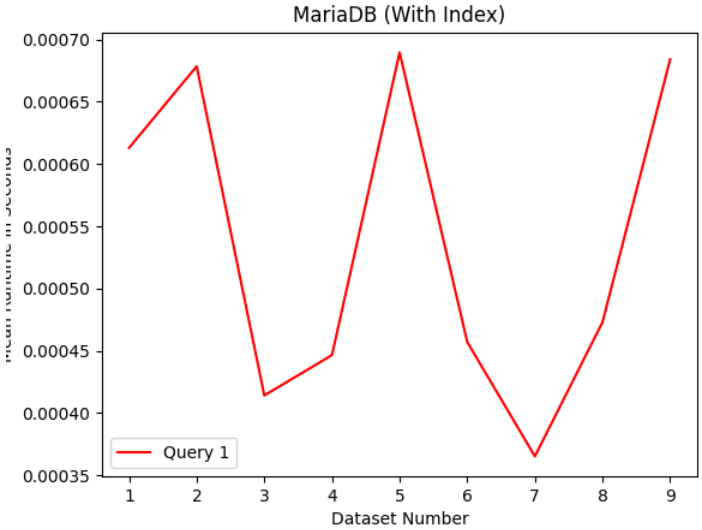
v. **For all Queries comparison**



Inference: Query 1 and Query 3 behave in a similar way as to SQLite3. However, Queries 2 and 4 take much more time since there is no indexing. Here, because of no foreign key constraint, join becomes a much more time consuming operation than sorting and hence, Query 4 has a far larger execution time than Query 2.

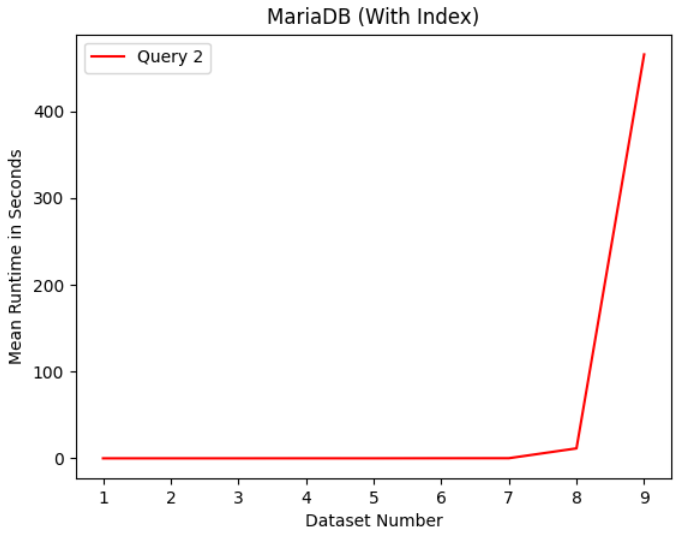
c. **MariaDB (With Index)**

i. **For Query 1**



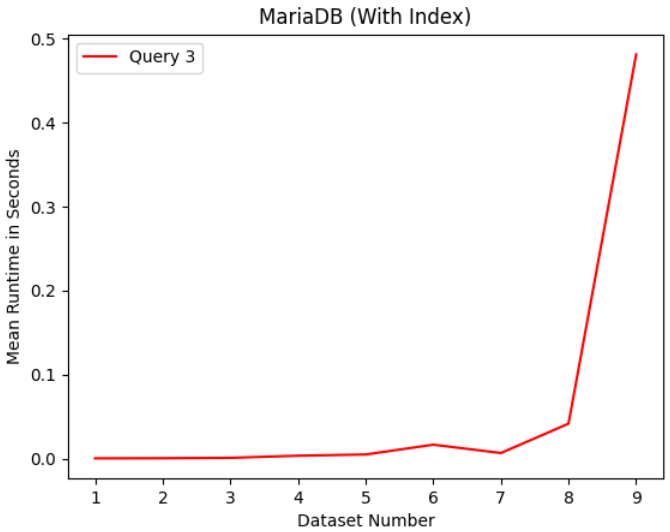
Inference: Similar to the SQLite3 case since now it is again indexed because of the primary key and hence, no clear trend. And since low execution times, even little interruptions can cause huge errors.

ii. For Query 2



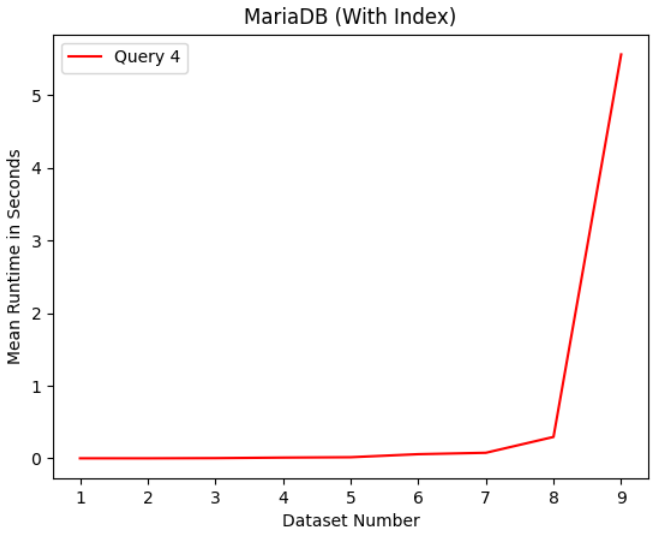
Inference: Similar to the sqlite3 case, execution time increases with increase in size of the dataset. Note that MariaDB by default sorts strings without discriminating between lowercase and uppercase characters. This takes time comparable to SQLite3. However, since we have used the 'Binary' keyword to sort according to ASCII values, this is taking a lot more time as compared to SQLite3.

iii. For Query 3



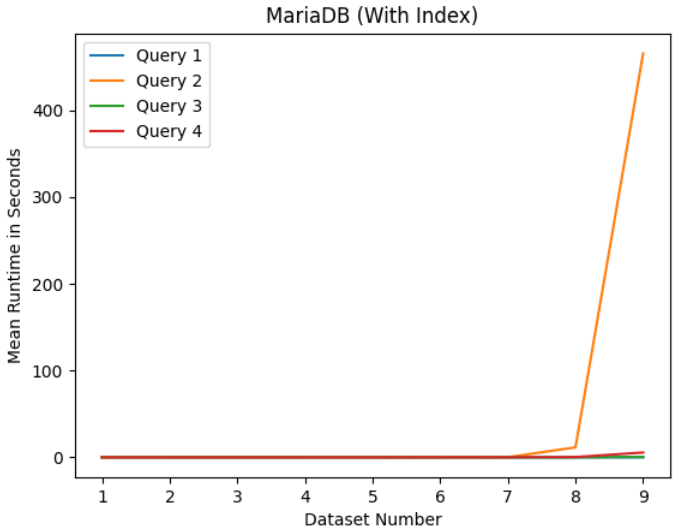
Inference: Similar to the sqlite3 case, execution time increases with increase in size of the dataset.

iv. For Query 4



Inference: Similar to the sqlite3 case, execution time increases with increase in size of the dataset.

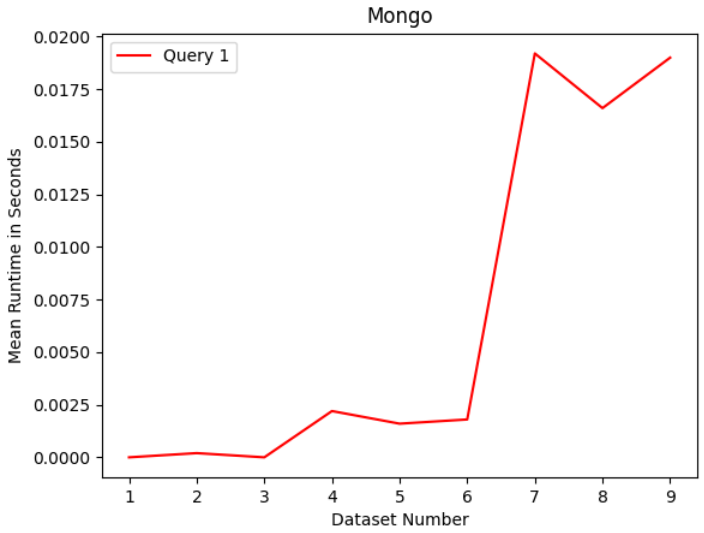
v. For all Queries comparison



Inference: Query 1,3 and 4 are similar to what happened in SQLite3 since we have the primary key and foreign key constraints applied here again. However, the time for sorting (Query 2) is much more over here because of the use of the ‘Binary’ keyword. Sorting by default is done by not differentiating between lowercase and uppercase characters. But to enforce sorting by ASCII values, the ‘Binary’ keyword is used which increases the time for sorting by a lot.

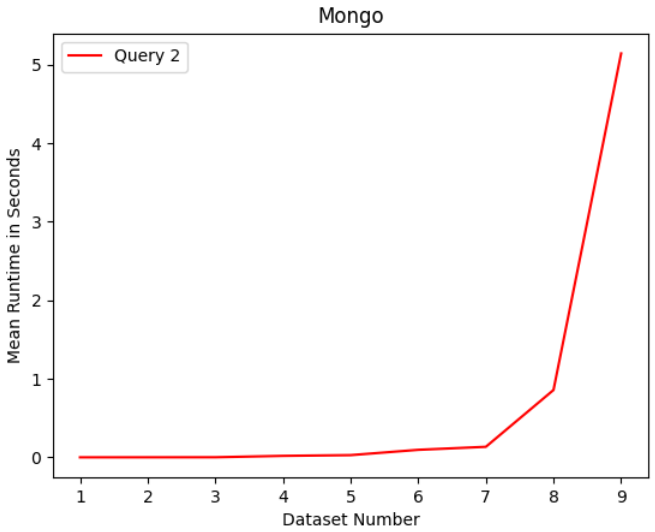
d. MongoDB

i. For Query 1



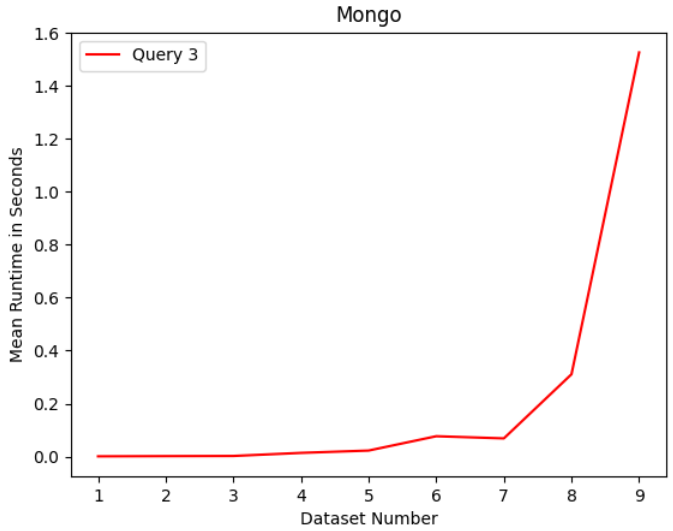
Inference: We can see a very rough trend of the execution time increasing with increase in size of the A collection.

ii. For Query 2



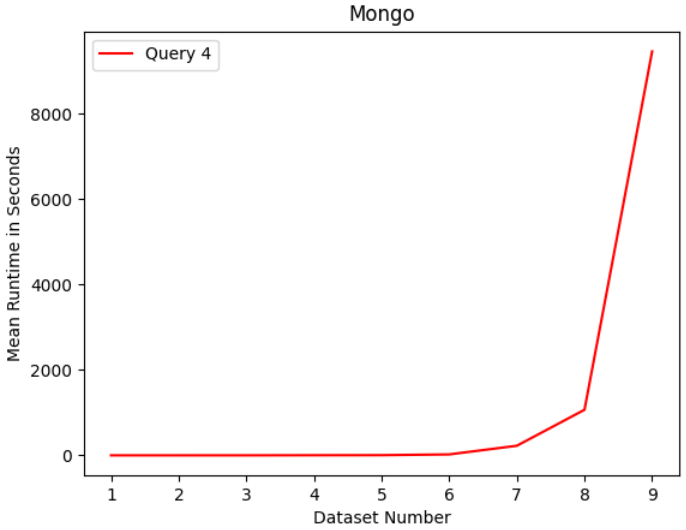
Inference: We can see a clear trend that the execution time increases with increase in size of datasets.

iii. For Query 3



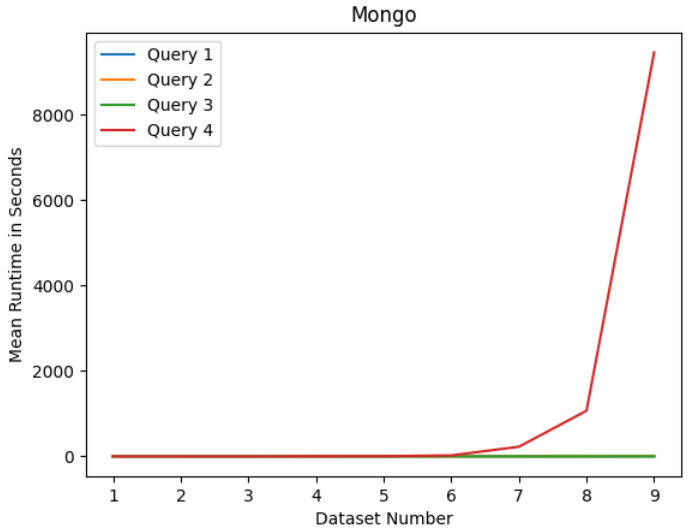
Inference: We can see a rough trend that the execution time increases with increase in size of datasets.

iv. For Query 4



Inference: We can see a clear trend that the execution time increases with increase in size of datasets.

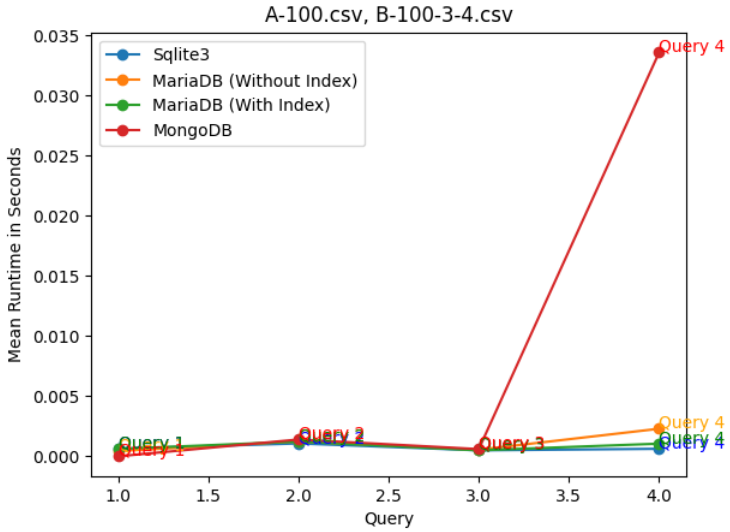
v. For all Queries comparison



Inference: The lines for queries 1,2,3 almost overlap because of the huge amount of difference in execution times between them and that of query 4. The large amount of time taken by query 4 can be explained from the fact that MongoDB is not exactly built for the join operation which takes a huge time here. Queries 1,2 and 3 are single collection queries and hence, don't have huge execution times whereas query 4 involves operations on 2 collections which explains the behaviour.

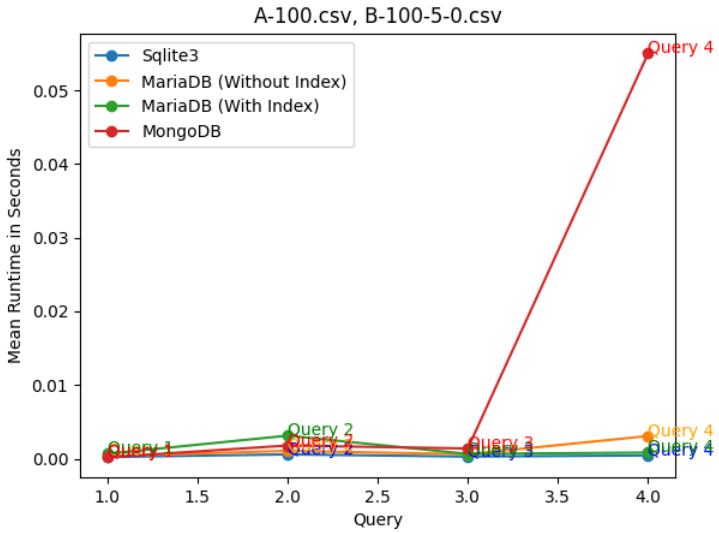
2. Comparing the execution times of all queries per dataset

a. A-100.csv, B-100-3-4.csv



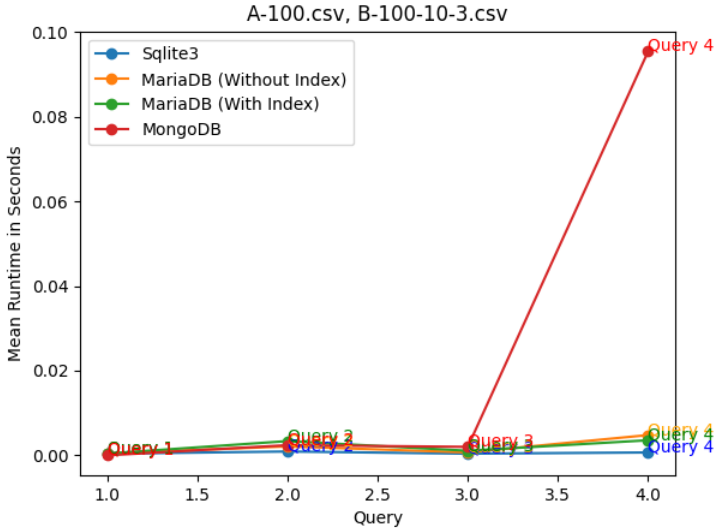
Inference: We can see the huge difference for query 4 in MongoDB and the SQL database SQLite3 and MariaDB which is logical since join operations are much faster in relational databases than in no-sql databases like MongoDB. For the other operations not much difference is seen since the datasets are small.

b. A-100.csv, B-100-5-0.csv



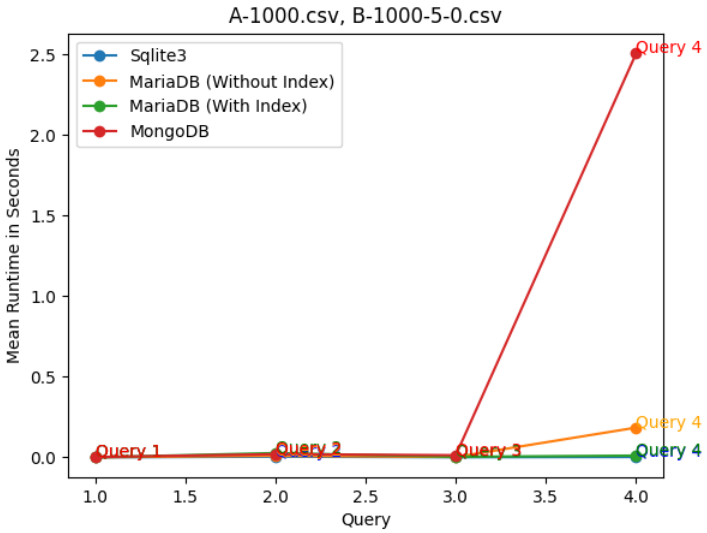
Inference: Same as the inference of (a)

c. A-100.csv, B-100-10-3.csv



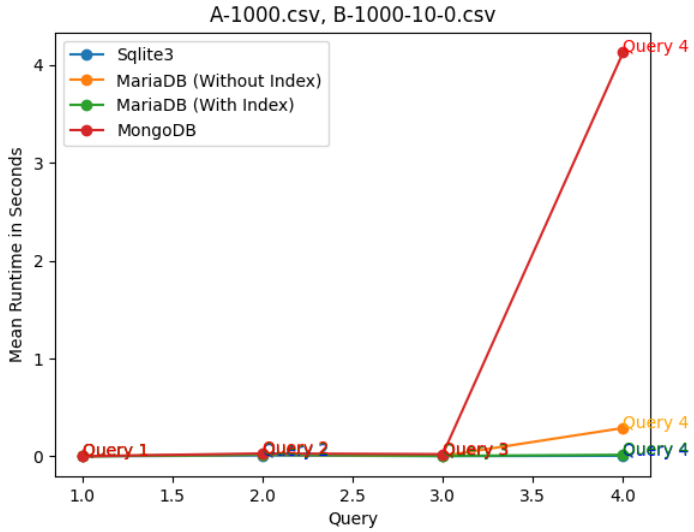
Inference: Same as the inference of (a)

d. A-1000.csv, B-1000-5-0.csv



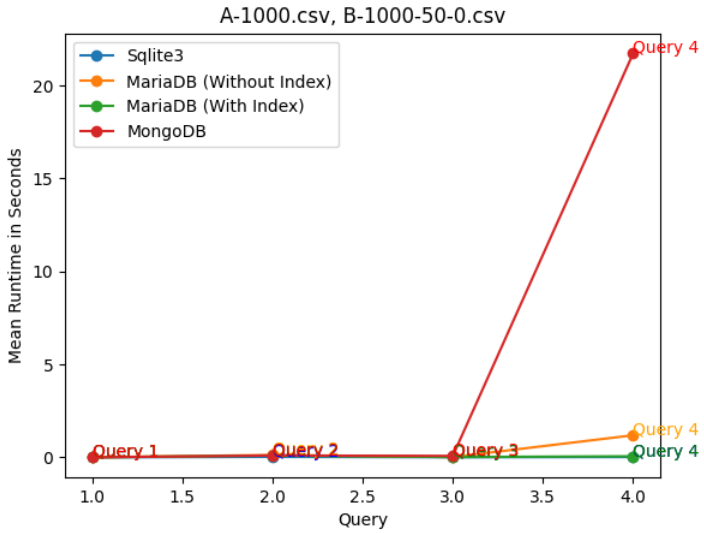
Inference: Same as the previous inference for queries 1,2 and 3. For query 4, apart from the huge gap shown by mongoDB, now we can also see the difference in time created by mariaDB without index since we did not have any foreign key constraints for it unlike SQLite3 and MariaDB with index. Hence, it takes a bit more time than the other 2 relational databases.

e. A-1000.csv, B-1000-10-0.csv



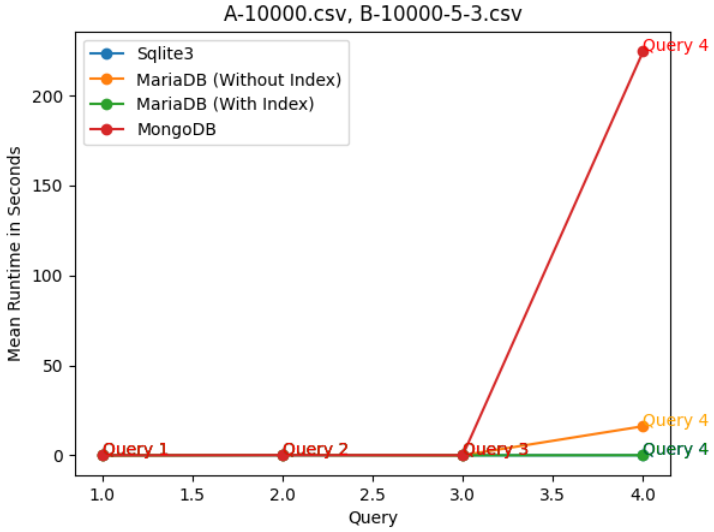
Inference: Same as the inference for (d).

f. A-1000.csv, B-1000-50-0.csv



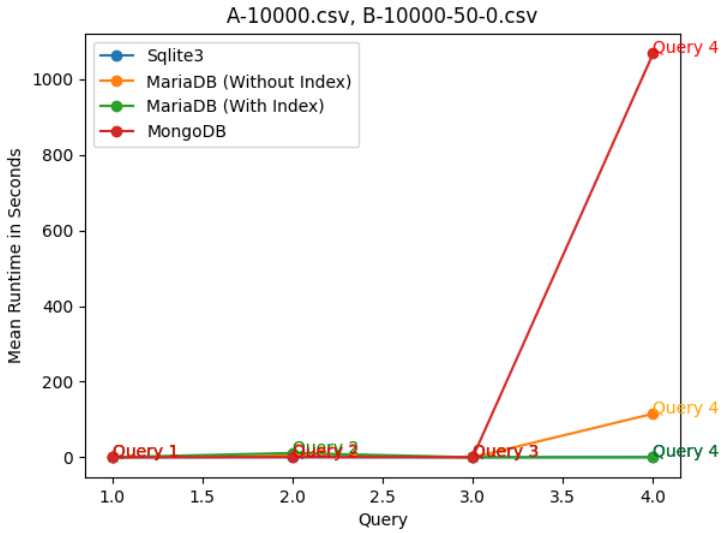
Inference: Same as the inference for (d).

g. A-10000.csv, B-10000-5-3.csv



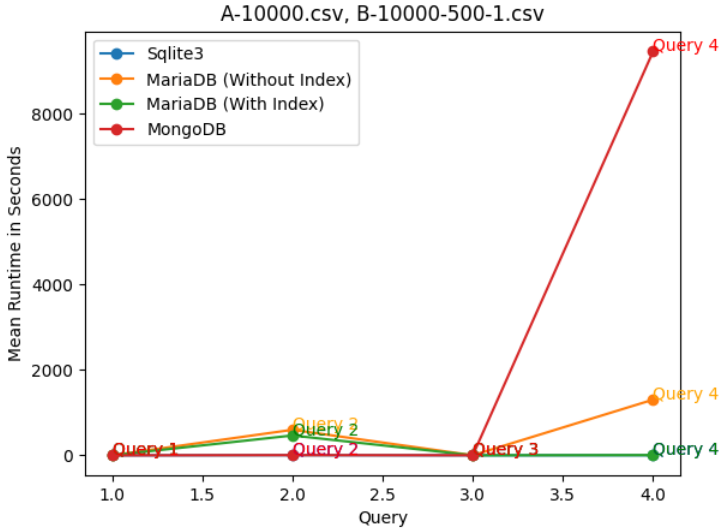
Inference: Same as the inference for (d).

h. A-10000.csv, B-10000-50-0.csv



Inference: Same as the inference for (d).

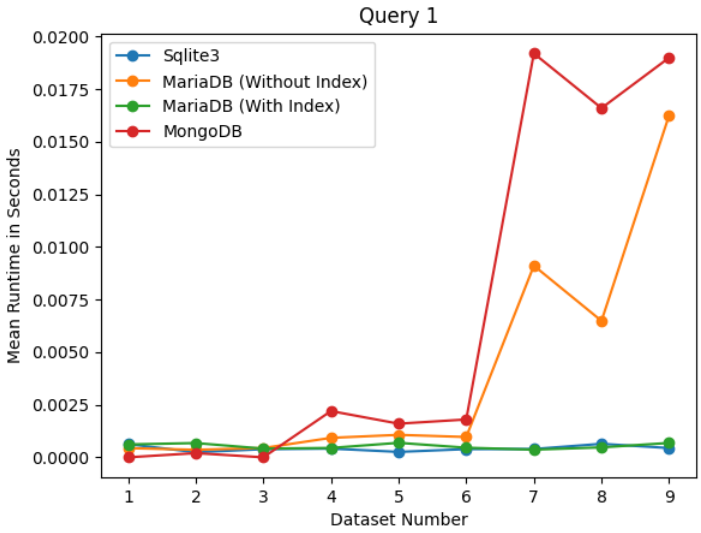
i. A-10000.csv, B-10000-500-1.csv



Inference: Queries 1,3 and 4 show the same trend as (d). But now, we can see the distinction between execution times of query 2 for MongoDB, SQLite and the remaining two relational databases because of the large size of the dataset. This is consistent with what we discussed before that MariaDB is slower when it comes to the sorting operation with ‘Binary’ keyword we have used. If we had not used the ‘Binary’ keyword, then their execution times would be comparable to that of SQLite3.

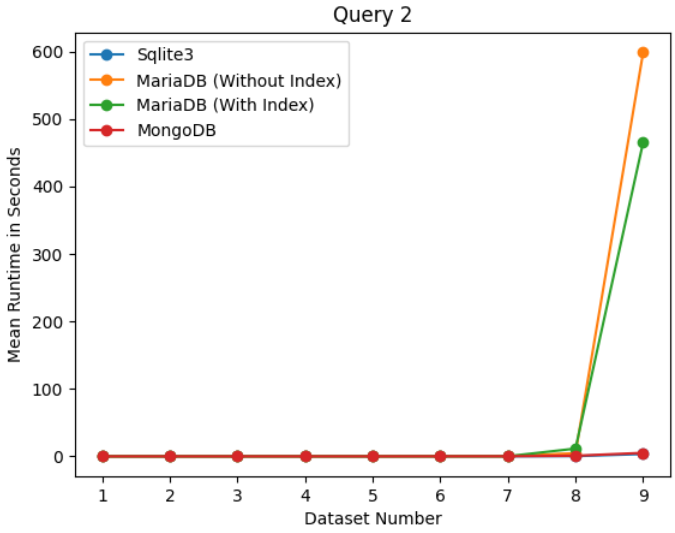
3. Comparing the execution times of all queries per query

a. Query 1



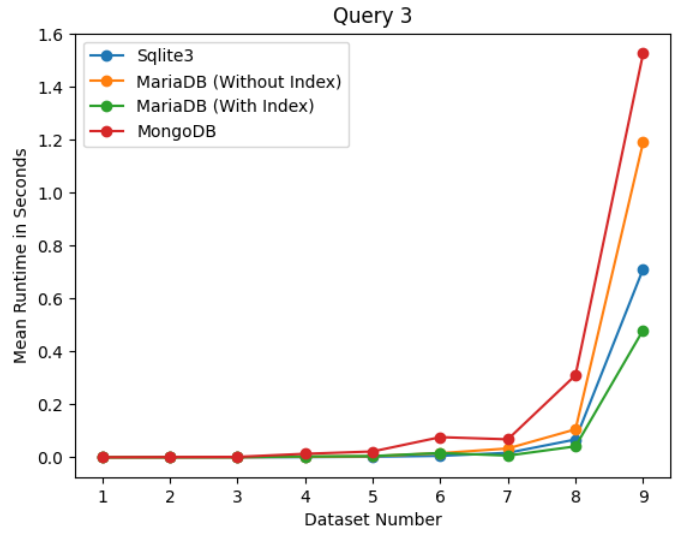
Inference: Since this query takes very little time in all databases, even small interruptions cause a huge error. Because of this no clear trend can be seen. A few things we do notice are that mariaDB without index takes more time than indexed MariaDB and SQLite3 which is very logical since there is no indexing. It also seems that MongoDB takes more time than relational databases for this query as the size of the dataset increases although this can be because we have taken the printing times into consideration. MongoDB outputs in batches which we have to call again and again whereas the remaining three output all resulting tuples at once.

b. Query 2



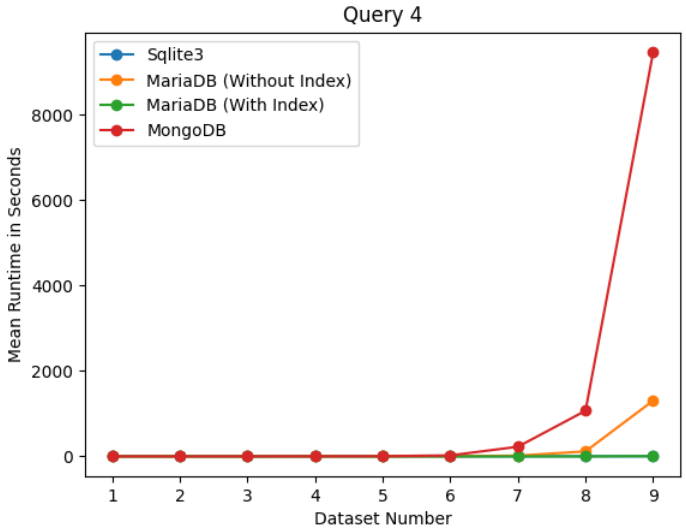
Inference: As discussed previously, MongoDB and SQLite3 take very less time for sorting as compared to MariaDB. Between the two MariaDB databases, the indexed one takes less time since we have built an index on the B3 attribute. Note that MariaDB by default sorts strings without discriminating between lowercase and uppercase characters. This takes time comparable to that of SQLite3. However, since we have used the 'Binary' keyword to sort according to ASCII values, this is taking a lot more time than SQLite3.

c. Query 3



Inference: Through this we can understand the trend shown by the aggregation (or group by) operation in all the databases. As the size of the dataset increases, we can see that MongoDB takes the longest time followed by MariaDB (without index), SQLite3 and lastly, indexed MariaDB. However, the difference in times is very minute as compared to that of query 4.

d. Query 4



Inference: As expected and discussed before, MongoDb takes the highest time for join operations followed by non-indexed MariaDB since there are no foreign key constraints. Sqlite3 and Indexed MariaDB take the least time because of the implicit index created because of the foreign key constraint.

Conclusion:

1. The first thing to conclude would be that when we measure the time taking into account the time taken for printing, it is substantially more than what it would be to just run the query. This is quite logical because printing the output of the query would require the use of various I/O buffers as well because of which the time increases.

2. Secondly, in this assignment, we wrote bash scripts to populate the tables and to run the queries. This would add a little bit of time overhead as compared to manually running the queries inside the respective database system shells, although, it is very insignificant.

3. Talking of scalability, we know that Big Data systems that exist today all use NoSQL databases. Hence, it would be naturally better to choose MongoDB with distributed systems as our database for larger datasets. However, if the operations involve the join functionality or any operation that requires the use of multiple collections, then it would be better to go for relational databases as they are built to capture the relationships between datasets. Although, we can design the collections in MongoDB in such a way that we wouldn't require many join operations.

4. Since we are running the queries on our local laptop and not on some dedicated server, the execution times for multiple attempts may be skewed because of the other processes that run concurrently with our queries. Which is why the standard deviations become a bit significant in case of smaller dataset queries. Using dedicated servers would reduce the execution times since such servers are used solely for the purpose of running the queries. To reduce the amount of interference, one technique was to restart the laptop once to ensure no application background tasks are running. Secondly, I kept only the terminal open. Lastly, I kept the laptop undisturbed during query executions. Although this does not make a great difference, it is a step towards getting more accuracy by achieving some form of isolation.

5. Another thing is about caching. If we do not clear the cache after each query is run, the database remembers certain things to reduce the runtime of the successive queries. But since we want to get independent runtimes of the queries, I cleared the cache after every query was run. This ensured that each query was independent of the previous ones.

6. Based on the runtimes that we saw, we can say that the following databases are better for each of the queries:
    a. Query 1 (Required only selection) - SQLite3, Indexed MariaDB, MongoDB
    b. Query 2 (Sorting) - SQLite3, MongoDB
    c. Query 3 (Aggregation/Group By) - SQLite3, Indexed MariaDB
    d. Query 4 (Join) - SQLite3, Indexed MariaDB

7. Since my laptop had a GPU as well, I tried to search a bit to check if database queries make use of GPU as well during execution. Even though I could not get any concrete answer, one thing is for sure that using the GPU would improve the runtime efficiency of the database since it adds a lot more parallel processing. This is evidently giving rise to a new form of databases which run on GPUs instead of solely CPUs. Unfortunately, I could not determine if my laptop uses the GPU to run the database queries but popular opinion on the internet is that in local machines, databases run solely on CPUs.

8. Having a better processor helps as well. The execution times I have mentioned are of the queries that I ran on this laptop (details in the “Configuration Details” section on Page 1 of this report). Just for the sake of comparison, I ran the same queries in a device which had a lower-end Intel i5 processor and as expected, the runtimes on the i5 processor came out to be larger than what I obtained from this device (i7 processor).

9. Lastly, I would like to point out that we cannot classify any database system as the best one for all applications. Each database is built to handle certain tasks optimally. So, it is better to decide what database to use after understanding what its application is going to be. For example, if we want to capture the relationships in the data, then it is better to go for a relational database model than using something like MongoDB. No wonder, because of such vivid applications, there are so many varieties of database systems emerging.

Submission:

I have included the bash scripts, the output of the bash scripts, the main.py file, the output execution time table created by main.py. The directory structure is similar to the one mentioned in the “Directory Structure” section in this report on Page 2. Be sure to create the “Images/” folder in the “scripts/” folder and then subfolders “Part1/”, “Part2/” and “Part3/” inside the “Images/” folder before running main.py.