Assignment 1

Name: Ottor Mills

ID: 620098373

Prerequisites

Ensure that docker and docker-compose are installed on the host system.

$ docker

expected output:

```
[ottor@fedora Assignment_1]$ docker

Usage:  docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default "/home/ottor/.docker")
  -c, --context string     Name of the context to use to connect to the daemon (overrides DOCKER_HOST env var an
  -D, --debug              Enable debug mode
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default "/home/ottor/.docker/ca.pem")
      --tlscert string     Path to TLS certificate file (default "/home/ottor/.docker/cert.pem")
      --tlskey string      Path to TLS key file (default "/home/ottor/.docker/key.pem")
      --tlsverify          Use TLS and verify the remote
  -v, --version            Print version information and quit
```

Ensure that nodejs is installed and the command “node” is recognized successfully.

Automatic Setup:

To get both the MySQL and Mongo database up and running quickly navigate to the “database” folder run the “start-databases” shell file. Once both databases have been started they can be populated using the “populator” package. Navigate to the “populator” directory and follow the README file. The database population process will commence.

Manual Setup:

Part 1

i)

Using Docker, create a new MySQL container (rudimentary plain text password) using the command:

$ docker pull mysql && docker run --name mysql_620098373 -p 3306:3306 -e MYSQL_ROOT_PASSWORD=root -d mysql

expected output:

```
[ottor@fedora Assignment_1]$ docker pull mysql && docker run --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=root -d mysql
Using default tag: latest
latest: Pulling from library/mysql
Digest: sha256:4fcf5df6c46c80db19675a5c067e737c1bc8b0e78e94e816a778ae2c6577213d
Status: Image is up to date for mysql:latest
docker.io/library/mysql:latest
5756ce0c2e560ca9acaa80b48bc12a3a7104621ed1cad3c7f2ff5f303d6755cb
```

Now connect to the newly created database using the MySQL database using shell included in the container using the following command. You will need to provide the connection password (root):

$ docker exec -it mysql_620098373 mysql -u root -p

expected output:

```
[ottor@fedora Assignment_1]$ docker exec -it mysql mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.26 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Once connected to the database server create the database called “cab_company” and create the necessary tables using the queries below:

```
CREATE DATABASE IF NOT EXISTS cab_company;
use cab_company;

CREATE TABLE IF NOT EXISTS employees (
   eid INT(11) NOT NULL AUTO_INCREMENT,
   first_name VARCHAR(50) NOT NULL,
   middle_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,
   gender VARCHAR(1) NOT NULL,
   phone VARCHAR(15) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   PRIMARY KEY(eid)
);

CREATE TABLE IF NOT EXISTS vehicles (
   vid INT(11) NOT NULL AUTO_INCREMENT,
   eid INT(11) NOT NULL,
   yr INT(4) NOT NULL,
   brand VARCHAR(50) NOT NULL,
   model VARCHAR(50) NOT NULL,
   vin VARCHAR(50) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   PRIMARY KEY(vid),
   UNIQUE(eid),
   FOREIGN KEY(eid) REFERENCES employees(eid)
);

CREATE TABLE IF NOT EXISTS accounts (
   aid INT(11) NOT NULL AUTO_INCREMENT,
   eid INT(11) NOT NULL,
   bic VARCHAR(12),
   account_number VARCHAR(12),
   bitcoin_address VARCHAR(64) NOT NULL,
   ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
   PRIMARY KEY(aid),
   FOREIGN KEY(eid) REFERENCES employees(eid)
);
```

expected output:

```
mysql> CREATE DATABASE cab_company;
Query OK, 1 row affected (0.60 sec)

mysql> use cab_company
Database changed
mysql> CREATE TABLE IF NOT EXISTS employees (
    ->     eid INT(11) NOT NULL AUTO_INCREMENT,
    ->     first_name VARCHAR(50) NOT NULL,
    ->     middle_name VARCHAR(50) NOT NULL,
    ->     last_name VARCHAR(50) NOT NULL,
    ->     gender VARCHAR(1) NOT NULL,
    ->     phone VARCHAR(15) NOT NULL,
    ->     ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    ->     PRIMARY KEY(eid)
    -> );
Query OK, 0 rows affected, 1 warning (0.58 sec)

mysql>
mysql> CREATE TABLE IF NOT EXISTS vehicles (
    ->     vid INT(11) NOT NULL AUTO_INCREMENT,
    ->     eid INT(11) NOT NULL,
    ->     yr INT(4) NOT NULL,
    ->     brand VARCHAR(50) NOT NULL,
    ->     model VARCHAR(50) NOT NULL,
    ->     vin VARCHAR(50) NOT NULL,
    ->     ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    ->     PRIMARY KEY(vid),
    ->     UNIQUE(eid),
    ->     FOREIGN KEY(eid) REFERENCES employees(eid)
    -> );
Query OK, 0 rows affected, 3 warnings (1.00 sec)

mysql>
mysql> CREATE TABLE IF NOT EXISTS accounts (
    ->     aid INT(11) NOT NULL AUTO_INCREMENT,
    ->     eid INT(11) NOT NULL,
    ->     bic VARCHAR(12),
    ->     account_number VARCHAR(12),
    ->     bitcoin_address VARCHAR(64) NOT NULL,
    ->     ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
    ->     PRIMARY KEY(aid),
    ->     FOREIGN KEY(eid) REFERENCES employees(eid)
    -> );
Query OK, 0 rows affected, 2 warnings (0.91 sec)

mysql>
```

ii)

Using Docker, create a new Mongo container using the command:

$ docker pull mongo && docker run --name mongo_620098373 -p 27017:27017 -d mongo

expected output:

```
[ottor@fedora Assignment_1]$ docker pull mongo && docker run --name mongo -p 27017:27017 -d mongo
Using default tag: latest
latest: Pulling from library/mongo
Digest: sha256:71348de35c2edef847906cbcf175d12e15244cdc37fc4ae5661fde361f4e4893
Status: Image is up to date for mongo:latest
docker.io/library/mongo:latest
0696600af01112a780868baafacb505dd82c638bfada0531caf2e7b565a7cf0d
[ottor@fedora Assignment_1]$
```

Now test to the newly created MongoDB using the shell included in the container using the following command:

$ docker exec -it mongo_620098373 mongo

expected output

```
[ottor@fedora Assignment_1]$ docker exec -it mongo mongo
MongoDB shell version v5.0.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("f79218b7-d61f-493a-92f8-f8914b14a277") }
MongoDB server version: 5.0.3
================
Warning: the "mongo" shell has been superseded by "mongosh",
which delivers improved usability and compatibility.The "mongo" shell has been deprecated and will be removed in
an upcoming release.
We recommend you begin using "mongosh".
For installation instructions, see
https://docs.mongodb.com/mongodb-shell/install/
================
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
        https://docs.mongodb.com/
Questions? Try the MongoDB Developer Community Forums
        https://community.mongodb.com
---
The server generated these startup warnings when booting:
        2021-10-06T05:40:36.839+00:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
        2021-10-06T05:40:36.839+00:00: Soft rlimits for open file descriptors too low
        2021-10-06T05:40:36.839+00:00:         currentValue: 1024
        2021-10-06T05:40:36.839+00:00:         recommendedMinimum: 64000
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

Unlike MySQL, MongoDB automatically creates its storage buckets on demand. Therefore this no further action is needed as it related to schemas.

a) Completed in script file

b) Completed in script file

c) Completed in script file

d) Completed in script file

e) Using inner joins and variables:

i-) Finding all employees with a vehicle of year 2012 or more with bic equal to EILARSJ1

$$\pi_{employees.eid}\ \sigma_{vechs.yr >= 2012\ AND\ acc.bic = \text{"EILARSJ1"}}\ (employees \bowtie_{employees.eid = acc.eid} \rho_{acc}\ \pi_{eid, bic}\ accounts \bowtie_{employees.eid = vechs.eid} \rho_{vechs}\ \pi_{eid, yr}\ vehicles)$$

rows accessed:

employees ⋈ accounts
accesses = 10000 * 10000 rows = 1 * 10 ^ 8
returns 10000

employees ⋈ accounts ⋈ vehicles
accesses = 10000 * 10000
returns 10000

total tuples accessed = 1 * 10 ^ 8 + 1 * 10 ^ 8
= 2 * 10 ^ 8

ii) Finding all Females that drive a BMW or Toyota

FEMS $\leftarrow \pi_{eid} \sigma_{employees.gender = "F"} employees$

$\sigma_{\text{eid IN FEMS AND brand = "BMW" OR brand = "Toyota"}}$ vehicles

FEMS which were randomly generated
accesses = 10000
returns 5000

$\sigma_{\text{eid IN FEMS AND brand = "BMW" OR brand = "Toyota"}}$ vehicles
accesses = 5000 * 10000 = 50000
returns 457 tuples

total tuples accessed = 60000

f)

FEMS which were randomly generated

Total records in employees collection = 10000
therefore accesses = 10000
returns 5000

Using the employee ids:

$\sigma_{\text{eid IN FEMS AND brand = "BMW" OR brand = "Toyota"}}$ vehicles
accesses = 5000 * 10000 = 50000
returns 457 tuples

Part 2 – Results are in seconds

How results were obtained for MySQL queries:

```
mysql> SHOW PROFILES;
+----------+------------+--------------------------------------+
| Query_ID | Duration   | Query                                |
+----------+------------+--------------------------------------+
|       82 | 0.00449250 | SELECT * FROM employees LIMIT 5000   |
|       83 | 0.00365400 | SELECT * FROM employees LIMIT 5000   |
|       84 | 0.00472400 | SELECT * FROM employees LIMIT 5000   |
|       85 | 0.00379275 | SELECT * FROM employees LIMIT 5000   |
|       86 | 0.00351725 | SELECT * FROM employees LIMIT 5000   |
|       87 | 0.00663450 | SELECT * FROM employees LIMIT 10000  |
|       88 | 0.00850675 | SELECT * FROM employees LIMIT 10000  |
|       89 | 0.00654050 | SELECT * FROM employees LIMIT 10000  |
|       90 | 0.00744300 | SELECT * FROM employees LIMIT 10000  |
|       91 | 0.00864300 | SELECT * FROM employees LIMIT 10000  |
|       92 | 0.00722025 | SELECT * FROM employees LIMIT 10000  |
|       93 | 0.00746850 | SELECT * FROM employees LIMIT 10000  |
|       94 | 0.00668000 | SELECT * FROM employees LIMIT 10000  |
|       95 | 0.00843475 | SELECT * FROM employees LIMIT 10000  |
|       96 | 0.00908450 | SELECT * FROM employees LIMIT 10000  |
+----------+------------+--------------------------------------+
```

High precision seconds using PROFILES

How results were obtained for Mongo queries:

```
var before = Date.now();
db.employees.find().limit(10000);
var after = Date.now();
print(after - before);
```

Due to MongoDB’ s JavaScript engine there is a lack of precision. Only 3 decimal places could be obtained.

a)

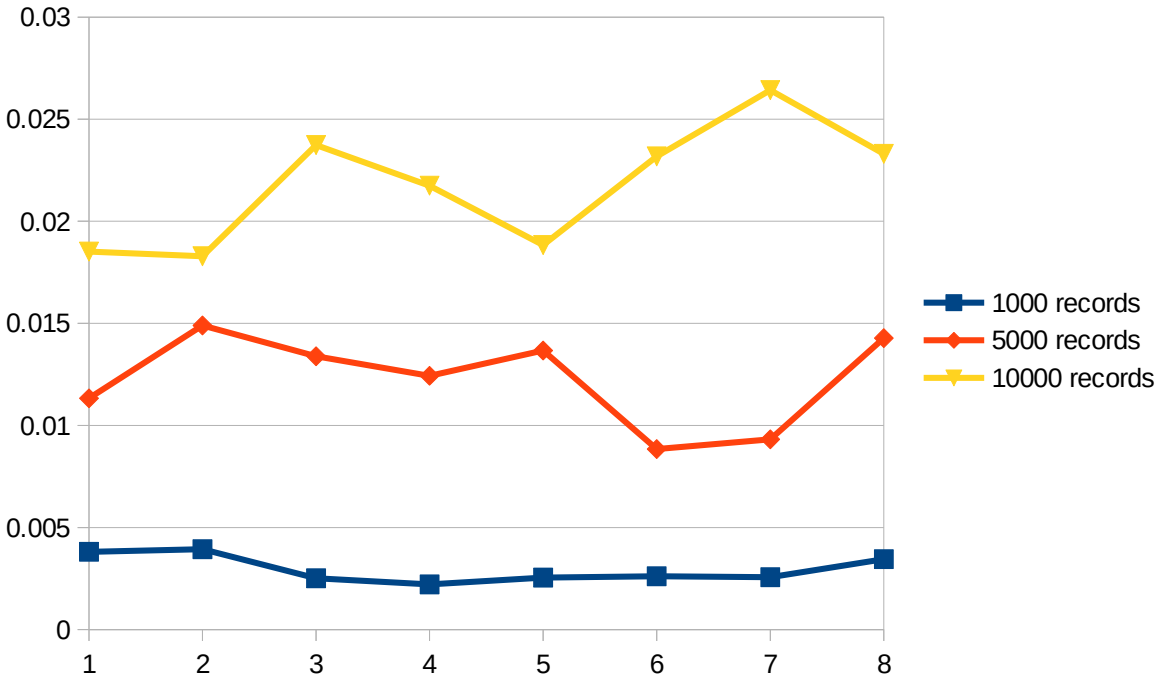
i) MySQL Select Comparisons

Query:

SELECT * FROM employees JOIN accounts ON employees.eid = accounts.eid LIMIT x;

Results in seconds:

1000 records	5000 records	10000 records
0.003811	0.0113285	0.01851125
0.00393825	0.01489625	0.01828825
0.0025165	0.013385	0.0237305
0.00221475	0.012432	0.02174325
0.002547	0.0136665	0.01883525
0.00261225	0.0088395	0.02317775
0.002566	0.009318	0.02642275
0.0034505	0.0142755	0.0233005



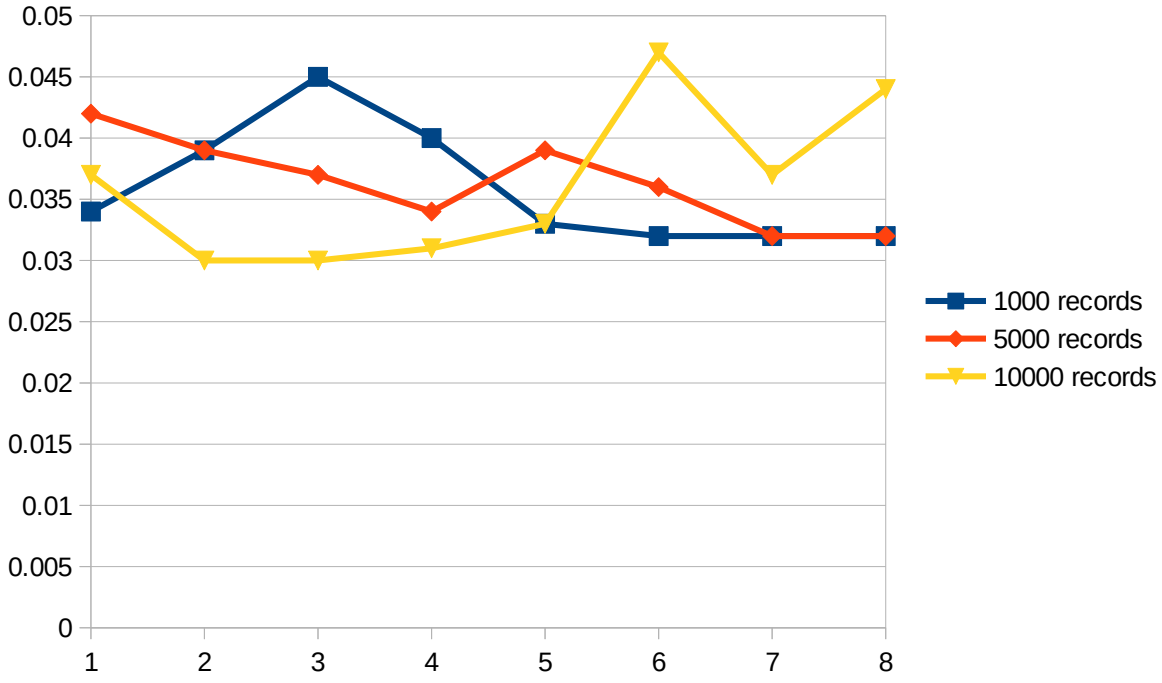
ii) Mongo Select Comparisons

Query:

```
 var before = Date.now();
db.getCollection("employees").aggregate([  {    $lookup: {      from: "accounts",      localField: "eid",
foreignField: "eid",      as: "Accounts",    },  },  {    $unwind: "$Accounts",  },  {    $lookup:
{      from: "vehicles",      localField: "eid",      foreignField: "eid",      as: "Vehicles",    },  },
{    $unwind: "$Vehicles",  },  { $limit : xx } ]);
var after = Date.now();
print(after - before);
```

Results in seconds:

1000 records	5000 records	10000 records
0.034	0.042	0.037
0.039	0.039	0.03
0.045	0.037	0.03
0.04	0.034	0.031
0.033	0.039	0.033
0.032	0.036	0.047
0.032	0.032	0.037
0.032	0.032	0.044



b)

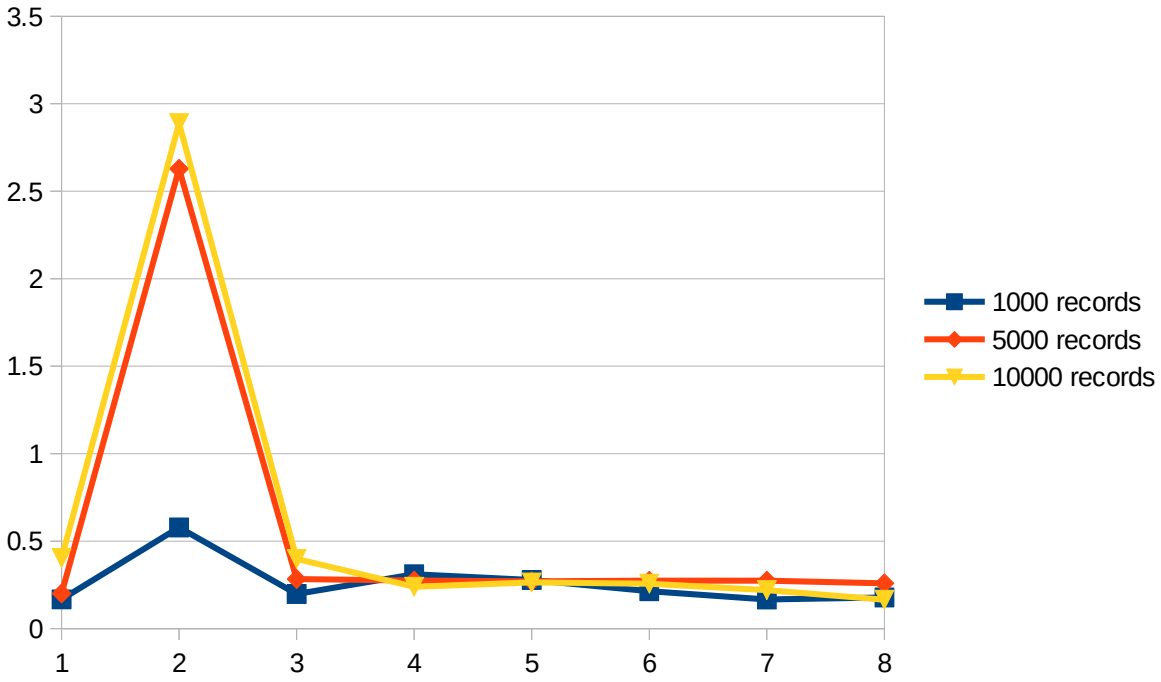
i) MySQL Update Comparisons

Query:

UPDATE accounts set account_number = (SELECT FLOOR(RAND()*(99999999-10000000)+10000000)) WHERE aid IN (1,2,3…..X)

Results:

1000 records	5000 records	10000 records
0.167442	0.20119	0.40385925
0.57863125	2.62825325	2.888748
0.19817075	0.28356375	0.39863275
0.311176	0.27481475	0.2388895
0.27798575	0.27043025	0.2641875
0.214141	0.273987	0.25694725
0.16690525	0.27425875	0.22003275
0.17877175	0.25938325	0.165112



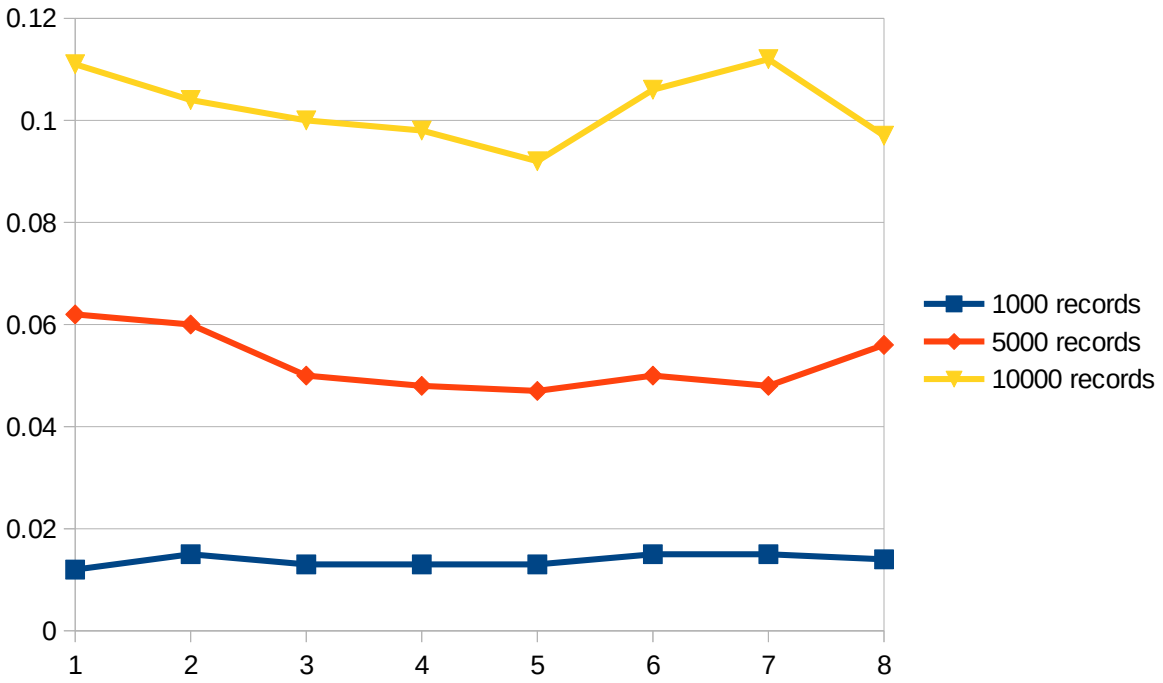
ii) Mongo Update Comparisons

Query:

```
var elements = db.accounts.find().limit(x).map(elm => elm._id);
var before = Date.now();
db.accounts.updateMany({_id: {$in: elements}}, {$set: {account_number: getRandomInteger(100000, 999999)}});
var after = Date.now();
print(after - before);
```

Results:

1000 records	5000 records	10000 records
0.012	0.062	0.111
0.015	0.06	0.104
0.013	0.05	0.1
0.013	0.048	0.098
0.013	0.047	0.092
0.015	0.05	0.106
0.015	0.048	0.112
0.014	0.056	0.097



c)

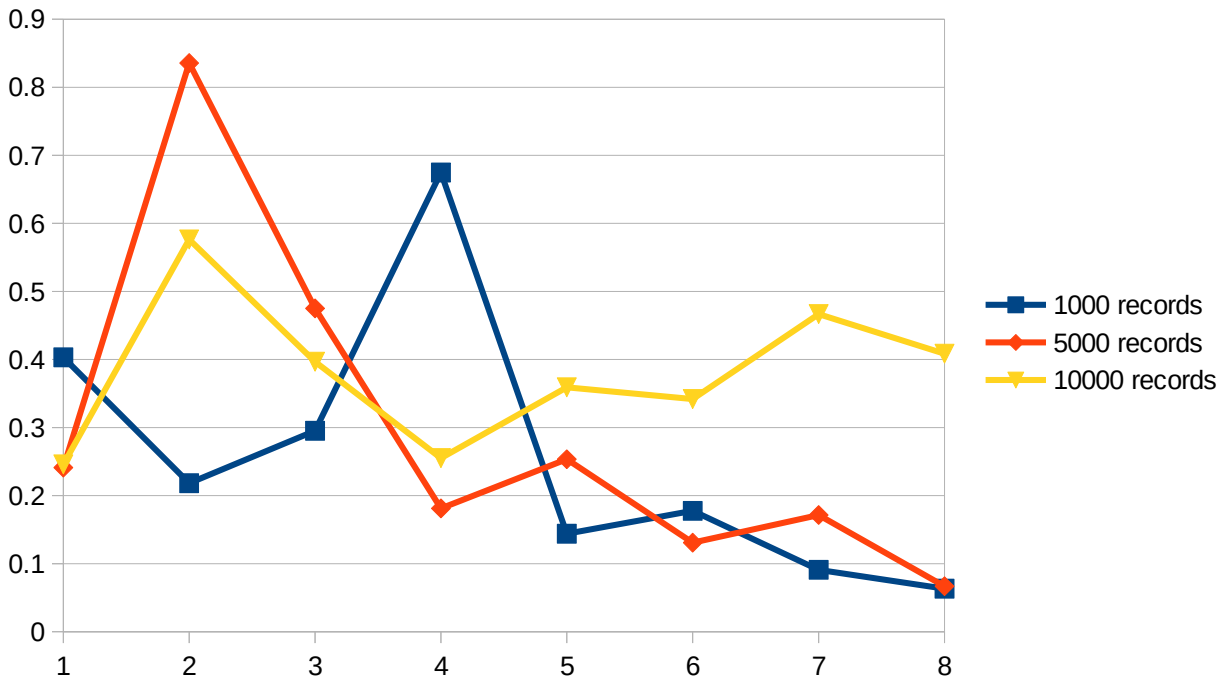
i) MySQL Delete Comparisons

Query:

DELETE FROM employees LIMIT x

Results:

1000 records	5000 records	10000 records
0.403291	0.241239	0.245849
0.2182955	0.835596	0.57639925
0.29531225	0.47511875	0.396801
0.6747075	0.1811305	0.2550835
0.14395925	0.2534605	0.359126
0.17762625	0.13087275	0.34195925
0.090962	0.171532	0.46672075
0.06319175	0.0669575	0.4085665



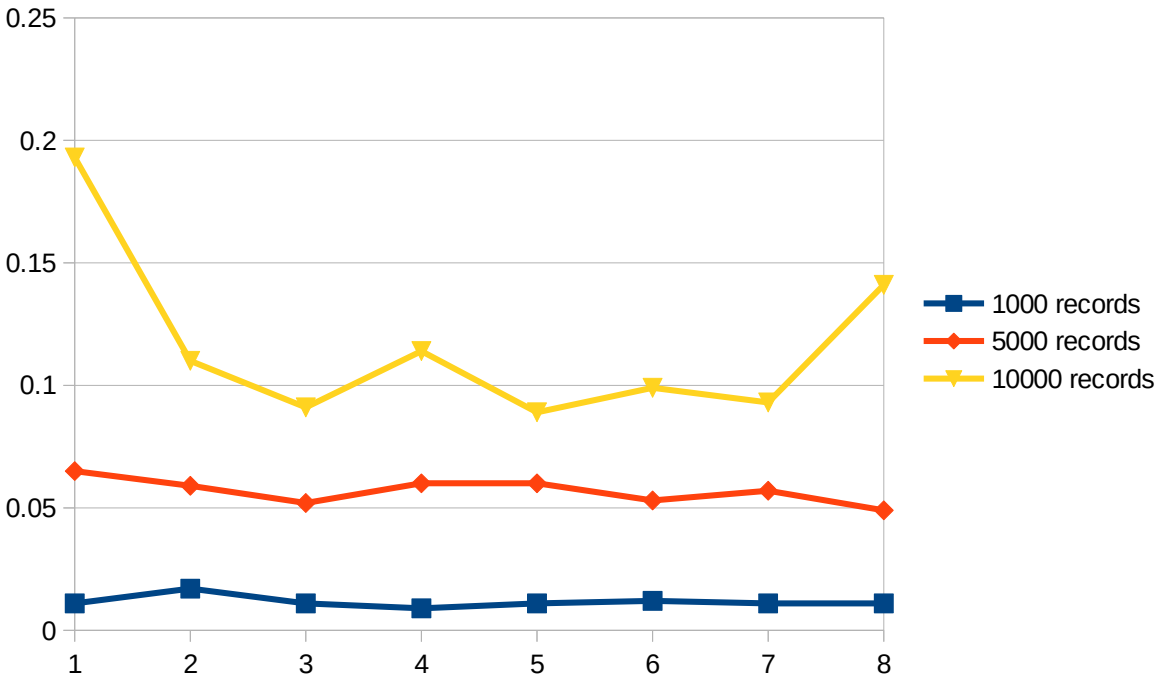
ii) Mongo Delete Comparisons

Query:

```
var elements = db.accounts.find().limit(5000).map(elm => elm._id);
var before = Date.now();
db.accounts.deleteMany({_id: {$in: elements}}, {$set: {account_number: getRandomInteger(100000, 999999)}});
var after = Date.now();
print(after – before);
```

Results:

1000 records	5000 records	10000 records
0.011	0.065	0.193
0.017	0.059	0.11
0.011	0.052	0.091
0.009	0.06	0.114
0.011	0.06	0.089
0.012	0.053	0.099
0.011	0.057	0.093
0.011	0.049	0.141



d)

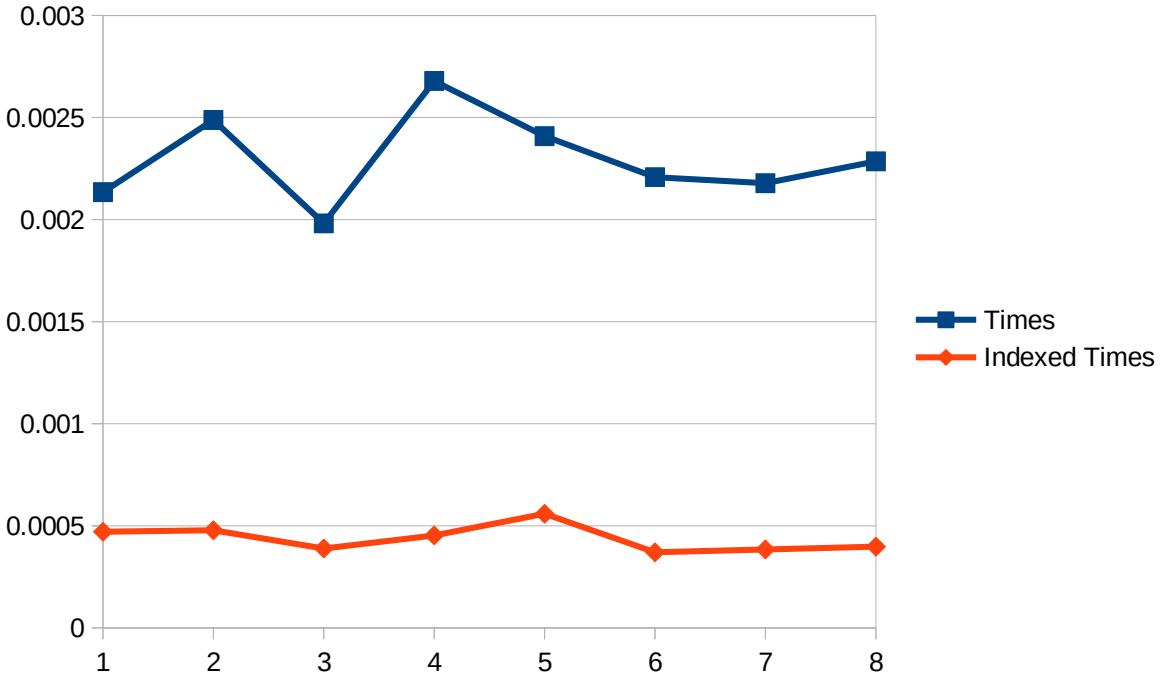
i) MySQL Selection results with indexes

Query:

SELECT COUNT(*) FROM employees WHERE first_name = "Anna";

CREATE INDEX employee_firstname_index ON employees (first_name);

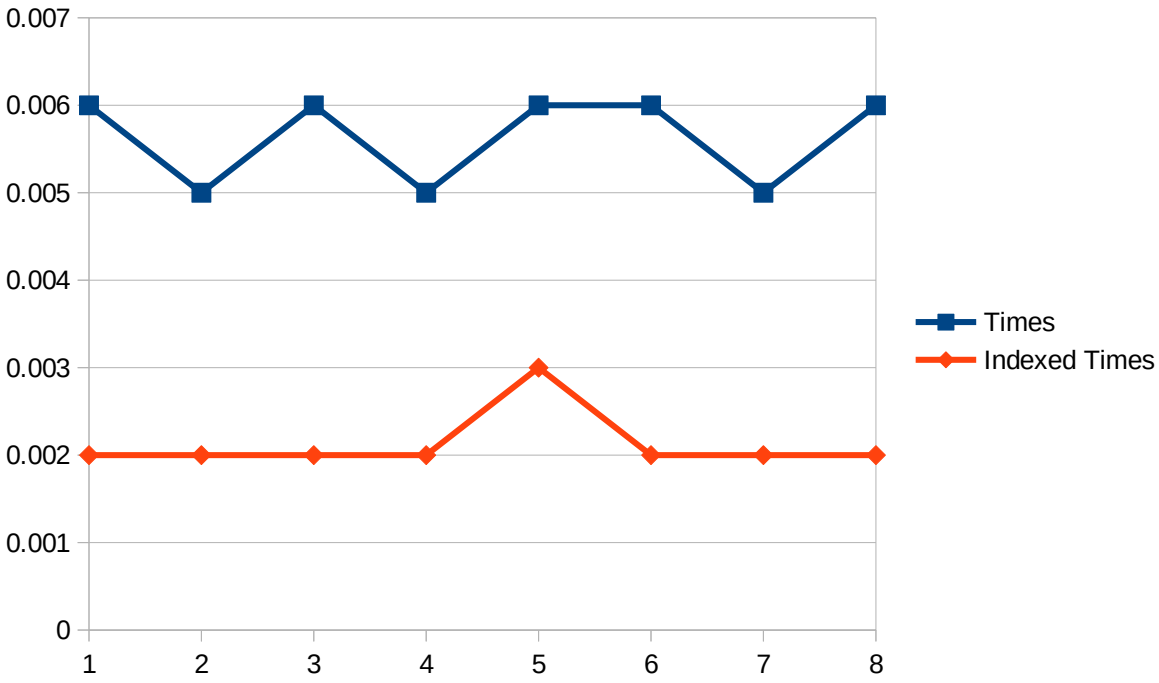
Times	Indexed Times
0.00213475	0.00047075
0.00248775	0.00047825
0.0019815	0.000389
0.00267875	0.00045275
0.0024085	0.00055875
0.002208	0.00037025
0.0021785	0.000384
0.0022855	0.00039775



ii) Mongo Selection results with indexes
var before = Date.now();

```
db.getCollection('employees').find({"firstName": "Anna"});
var after = Date.now();
print(after - before);
db.employees.createIndex({"firstName": 1});
```

Times	Indexed Times
0.006	0.002
0.005	0.002
0.006	0.002
0.005	0.002
0.006	0.003
0.006	0.002
0.005	0.002
0.006	0.002

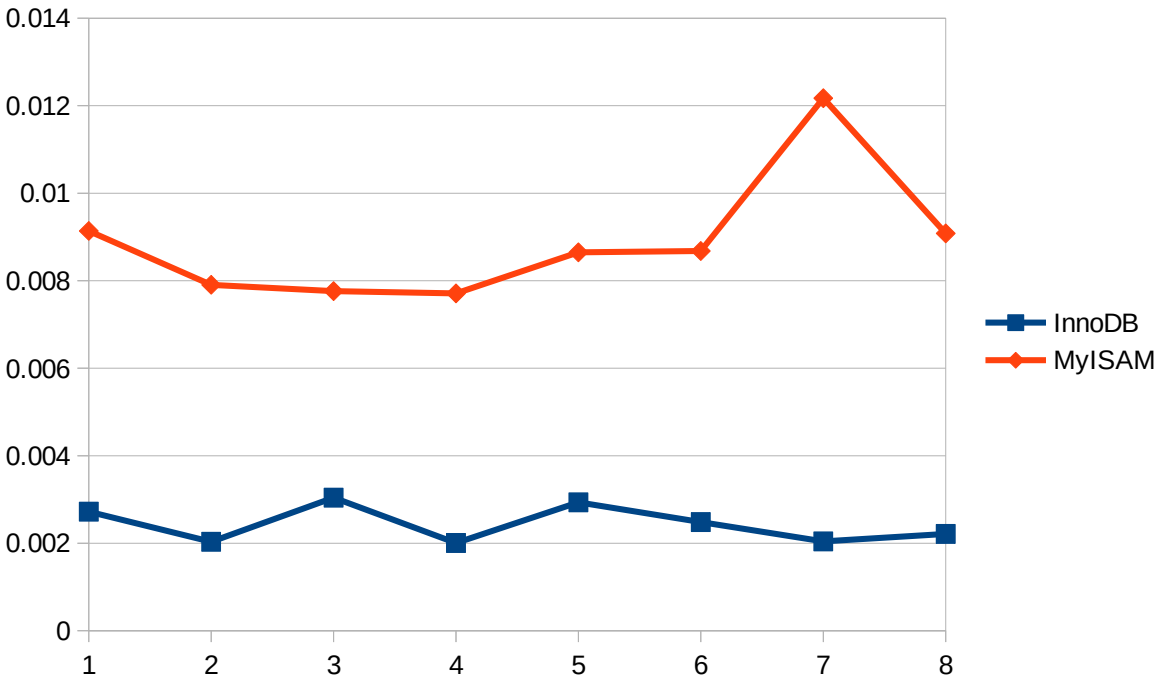


iii) MySQL Storage engine comparison

Query:

SELECT COUNT(*) FROM vehicles WHERE brand = "BMW"

InnoDB	MyISAM
0.00272125	0.00913925
0.00203175	0.0079065
0.00304025	0.00776275
0.0020045	0.0077105
0.00293575	0.008649
0.00248525	0.00868
0.002043	0.01217125
0.002213	0.00908075

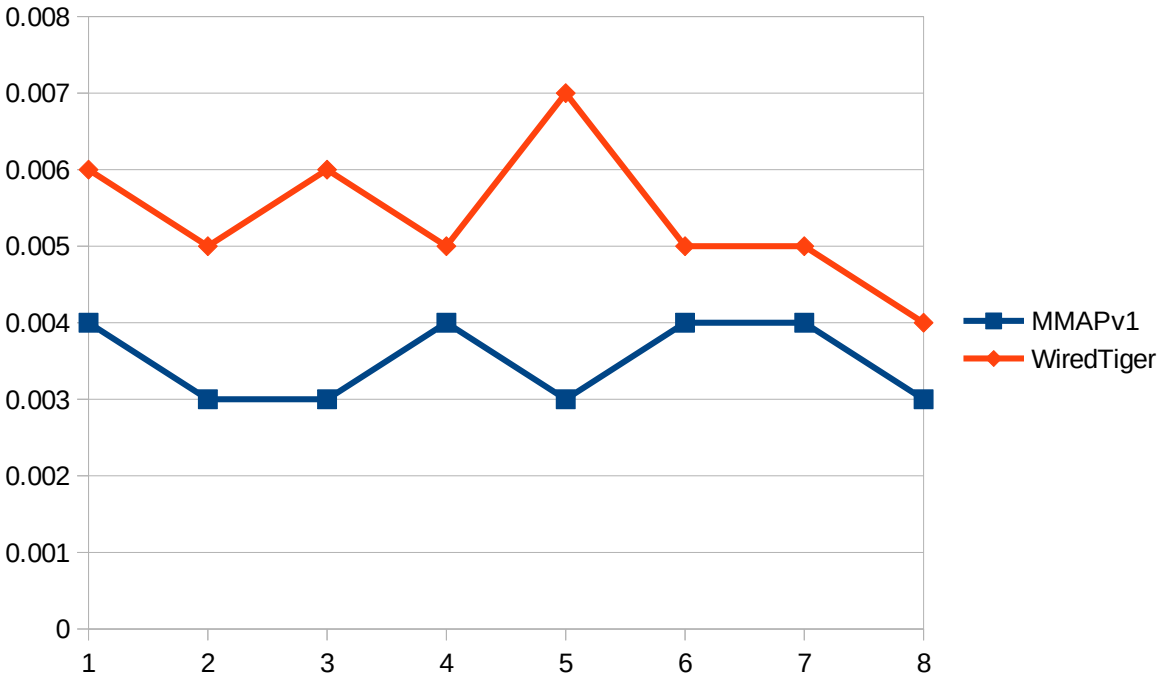


iv) Mongo Storage engine comparison

```
var before = Date.now();
db.getCollection('vehicles').find({"brand": "BMW"}).count();
```

```
var after = Date.now();
print(after – before);
```

MMAPv1	WiredTiger
0.004	0.006
0.003	0.005
0.003	0.006
0.004	0.005
0.003	0.007
0.004	0.005
0.004	0.005
0.003	0.004



e)

a) based on the populator script, the insertion time of the Mongo database was observed to be faster with insertions than that of the MySQL database

Proposed reason – The possible reason for this is that in a Mongo database data is written unstructured and as is to collections whereas in a MySQL database the data is split into various columns which can add some overhead to the operation.

b) From the selection query results it seems that the rate at which the MySQL queries become slower as the number of records being selected increases is greater than the said rate of the Mongo queries.

Proposed reason – Since data is stored unstructured in NoSQL databases (as mentioned before) it is accessed in the same manner which therefore means faster read times

c) From the deletion query results it seems that the Mongo query times increase at a much faster rate as the number of records being deleted increases when compared to the MySQL query times.

Proposed reason – This could be because MySQL has relationships. The relevant fields are all tied to a single primary key which results in faster field referencing and therefore removal

d) From the update query results it seems that the MySQL query times remained relatively constant as the number of records being updated increases in contract to the Mongo queries which became slower as the number of records being updated increased.

Proposed reason – MySQL could have faster updates because the fields being updated are structured unlike with Mongo database. This means that once the rows that match the update query are found the actual modification of the target field is a very simple process.

f) When comparing Relational to Non-Relational databases have many pros and cons. Relational databases offer rich, transnational, relationships between data stored across tables which allows for the joining of related tables in a single query. NoSQL databases are susceptible to inconsistencies in record attributes, therefore aggregations from multiple collections may yield unexpected results. In relational databases records that have relationships with records in another table have to live on the same database server in order for joins or subqueries to work, resulting in reduced the scaling potential. Data stored in NoSQL databases on the other hand have flexible schema which can be adapted to suit retrieval requirements and possibly prevent the querying multiple collections. This makes NoSQL databases easier to shard since each database instance does not need to have an identical schema. To place greater emphasis on the level of scalability associated with NoSQL databases, Apache Cassandra, a NoSQL DBMS, uses schema that is stricter than MongoDB but more flexible than a relational database. It avoids the master-slave relationship commonly seen among nodes with database scaling deployments as the master node is typically centralized and will eventually become a bottleneck in the

architecture. Instead, Apache Cassandra utilizes a decentralized, interconnected ring architecture with hashed ranges assigned to each node. Therefore if a query is sent to a random node the node can identify the node that contains the requested information using a hashing algorithm. If an Apache Cassandra deployment has multiple nodes in its ring and a range query is executed (greater than or less than operations) then Apache Cassandra will query multiple nodes for records matching the range. Not only does this consume bandwidth but it will also add a network delay to the query time resulting is significantly slower ranged queries.