Homework 5

The goal of this assignment is for you to empirically discover and justify the guidelines regarding usage of indexes within a large database application. Towards this end, you will be using the MariaDB relational database management system as an experimental platform: for a large data set provided to you, you will be measuring the amount of time and disk space various database operations require given different combinations of indexes. All work in this assignment should be done using the `MariaDB Console` application.

For this assignment, you are to submit a **single PDF file** with all of your data as well as written analysis in response to questions posed in the assignment. You will be evaluated on the completeness of the data, the correctness/insightfulness of your analysis (with respect to your data), as well as the professionalism of your writeup (including aesthetics, spelling/grammar, and organization). You must typeset all responses – hand-written/drawn work will receive 0% credit.

START THIS ASSIGNMENT EARLY. Much of the time involved goes to collecting data in various indexing situations, and to collect clean data your computer should not be working on other tasks (e.g. web browsing). Give yourself plenty of time to gather this data.

1 Preliminaries

For this assignment you will only need to run the MariaDB Server, not Apache.

2 Using the MariaDB Console

The console application is named `mysql` (or `mysql.exe` on Windows). You should start the terminal program on your platform (command prompt, or `cmd`, on Windows), and navigate to the `bin` directory under your XAMPP installation[1]. Once there, execute the following command[2]: `mysql -u root` and you should be brought to a MariaDB prompt.

Unfortunately the console application does not have autocompletion for file/directory names, and so you should place all files required for this assignment in an easy to type location, such as `/tmp` (or `c:\temp` on Windows).

When you are done, you can type `quit` to exit the console.

---

[1]Defaults: Windows (`c:\xampp\mysql\bin`), Mac (`/Applications/XAMPP/bin`)
[2]For Mac: `./mysql -u root`

3 Data Set

Provided with this assignment is a base schema (`schema.sql`), reproduced below:

```sql
CREATE TABLE data (
  a0 INT,
  a1 INT PRIMARY KEY,
  a2 INT,
  a5 INT,
  a10 INT,
  a100 INT,
  a1000 INT
);
```

The data that will populate this schema has been specially constructed for this assignment:

- The value of `a0` is always 0

- The value of `a1` is unique (first row $= 1$, second row $= 2$, ...)

- For all remaining fields an, the values repeatedly count from $1 \ldots n$

3.1 Loading the Dataset

At the beginning of this assignment, you should create a new database (e.g. `hw5`) and then switch to using it:

```sql
CREATE DATABASE hw5;
USE hw5;
```

You should then load the base schema to create the `data` table (use forward slashes in the path, even on Windows):

```sql
SOURCE path/to/schema.sql;
```

Finally, load the data into the table. The data we will use will include 10 million rows, and is thus relatively large. You will need to uncompress the supplied data file, and then source just as with the schema:

```sql
SOURCE path/to/data_10m_10k.sql;
```

The commands in this file are a set of `INSERT` statements, each adding 10,000 rows (i.e. 0.1%) to the `data` table. After each `INSERT` executes, you should see something like the following:

```
Query OK, 10000 rows affected (0.034 sec)
Records: 10000  Duplicates: 0  Warnings: 0
```

If your insert time is consistently greater than 1 sec, stop the process (`CTRL-C`). You likely have another disk-intensive process running and/or an anti-virus. Identify and stop the offending program and restart.

Once the inserts complete, confirm that all the data was inserted successfully via the following queries (note that each will take $\sim 1 - 2$ seconds or more, basically just to perform a table scan):

```
SELECT COUNT(*) FROM data; -- 10,000,000
SELECT SUM(a0) FROM data; -- 0
SELECT SUM(a2) FROM data; -- 15,000,000
SELECT SUM(a10) FROM data; -- 55,000,000
SELECT SUM(a100) FROM data; -- 505,000,000
SELECT SUM(a1000) FROM data; -- 5,005,000,000
```

Right now this table has no indexes (other than the automatic primary key on `a1`), so we will use these insert times as a baseline for later comparison. Take the average of the reported time (in the example above, `0.034 sec`) for the last 5 queries and enter that number below:

DP_INSERT_TIME_0) _____ **seconds/10,000 rows**

We also need to record the approximate space being used by the table – again, having minimal indexes, this measurement will serve as a baseline for future comparison. We will query the MariaDB catalog for this information using the following query:

```
SELECT ( ( data_length + index_length ) / power( 1024 , 2 ) ) AS tablesize_mb
FROM information_schema.tables
WHERE table_schema='hw5' and table_name='data';
```

The query adds memory used for the table itself, as well as any indexes, and divides by an appropriate constant to convert to MB from bytes. Record the result of your query:

DP_SPACE_0) _____ **MB**

3.2 Selectivity Analysis and EXPLAIN

For this data set you know that, by construction, each field has a fixed cardinality (i.e. number of distinct values). In general, however, you will not know this fact ahead of time. In the space below, write the SQL query that gets the number of distinct values for the a2 column:

DP_DISTINCT_TIME_0) _____ **seconds**

Now run this query three times and record the average time below:

Now modify this query to get the number of distinct values for the a1000 column, run three times, and compare to **DP_DISTINCT_TIME_0** – these should not be radically different. We can confirm this last hypothesis by looking at how MariaDB executed these queries via the EXPLAIN command (simply type EXPLAIN before your query and execute). The output should look something like the following (the value for rows may be slightly different – it's just a heuristic estimate by the DBMS):

```
+------+-------------+-------+------+---------------+------+---------+------+---------+-------+
| id   | select_type | table | type | possible_keys | key  | key_len | ref  | rows    | Extra |
+------+-------------+-------+------+---------------+------+---------+------+---------+-------+
|    1 | SIMPLE      | data  | ALL  | NULL          | NULL | NULL    | NULL | 9719061 |       |
+------+-------------+-------+------+---------------+------+---------+------+---------+-------+
```

The takeaways from this output are that the query is scanning the data table and not making use of any index. By contrast, if you execute the same query for the a1 column, your timing data should be different and the output of the EXPLAIN command will look something like the following:

```
+------+-------------+-------+-------+---------------+---------+---------+------+---------+-------------+
| id   | select_type | table | type  | possible_keys | key     | key_len | ref  | rows    | Extra       |
+------+-------------+-------+-------+---------------+---------+---------+------+---------+-------------+
|    1 | SIMPLE      | data  | index | NULL          | PRIMARY | 4       | NULL | 9719061 | Using index |
+------+-------------+-------+-------+---------------+---------+---------+------+---------+-------------+
```

MariaDB is reporting that for the non-indexed attributes (at this point, all of them aside from a1), the query was executed using a *table scan*. For the primary key, however, an index was used (though every row still had to be touched to generate the result set).

3.3   Baseline Query Performance

Now we are going to obtain query performance for a set of simple queries with single-attribute and multi-attribute `WHERE` clauses. Later we will compare these results with various indexing schemes.

All of our test queries will be aggregations, in order to control for large/different result set sizes. The basic format will be . . .

```
SELECT AVG(a1), COUNT(*) FROM data WHERE ...
```

For each set of attribute=value pairs below, perform the corresponding query three times and report the average query time. In each case, execute the query above, but with the `WHERE` condition indicated, noting that {x=a, y=b} should be interpreted as `x=a AND y=b`.

{ a1 = 1 }   _____ seconds

{ a2 = 1 }   _____ seconds

{ a10 = 1 }   _____ seconds

{ a100 = 1 }   _____ seconds

{ a1000 = 1 }   _____ seconds

{ a2 = 1, a1 = 1 }   _____ seconds

{ a2 = 1, a10 = 1 }   _____ seconds

{ a2 = 1, a100 = 1 }   _____ seconds

{ a2 = 1, a1000 = 1 }   _____ seconds

{ a2 = 1, a1000 = 1, a1 = 1 }   _____ seconds

{ a2 = 1, a1000 = 1, a10 = 1 }   _____ seconds

{ a2 = 1, a1000 = 1, a100 = 1 }   _____ seconds

You shouldn't encounter wildly different times between runs for the same query. If you do, stop any other processes on your machine that may be diverting CPU usage and retry. If you are surprised by a result, run the `EXPLAIN` command to understand the query plan.

3.3.1 Analysis

Based upon these results, what are your preliminary conclusions regarding query performance on this data set? You should comment on characteristics including prime/non-prime, selectivity, query plan (i.e. the result of using the `EXPLAIN` statement), and query length (i.e. number of attributes in the query). You are welcome to report additional results/analysis to bolster your conclusions.

4 Adding Indexes

We begin with single-attribute indexes, and then move into multiple indexes and multi-column indexes.

4.1 Individual Single-Attribute Indexes

We are now ready to add an index and assess its impact. The goal will be to populate the following table with experimental results:

	Insert Time (sec/10k rows)	**Space on Disk** (MB)	**Query Time** (seconds)
a2			/
a10			/
a100			/
a1000			/

To obtain these results, we are going to take a sequence of steps. This text will detail the steps for `a2` (the first row), then you will repeat for the remaining attributes (in each subsequent row).

4.1.1 Clear the Table

To see what impact the index will have on time to insert the rows, first drop the table (`DROP TABLE data;`) – this may take a minute or two. Now re-create the table using the schema file as we did at the beginning of the assignment.[3]

4.1.2 Create the Index

Add a new index on the column of interest (`CREATE INDEX foo ON data(a2);`).

---

[3]You could also just delete the data in the table via the `DELETE` command, but often this takes more time than just dropping the table as a whole.

4.1.3 Input the Data

Execute the `SOURCE` command above for the 10 million rows of data. Remember to (1) take the average of the last 5 batches of 10k rows, and make this the result in the second column of the table above and (2) perform the catalog-space query from above to fill in the third column in the table above. It would aid your analysis to note the percentage increase in time between these values and **DP_INSERT_TIME_0**/**DP_SPACE_0** above.

4.1.4 Perform Baseline Query

Check to see how the index changes the time to run your cardinality query (look back and compare to **DP_DISTINCT_TIME_0**) and single-attribute query (e.g. `WHERE a2=1`) – remember to average over three runs (the first may be large, due to loading from memory). You may also retry other queries that contain the attribute and see how they change as a result of the query – in cases that interact with the primary key, look at the output of `EXPLAIN` to see which index the query optimizer chose to utilize (see the `key` column).

4.1.5 Analysis

Repeat these steps for the remaining attributes (remembering to change `a2` in each situation for the attribute in row of the table) and then analyze the results. Your analysis should focus on the relative costs of adding a single index as compared to query benefits. Did there seem to be a trend as to which indexes cost/improved performance more? And for which type of query?

4.2 Multiple Single-Attribute Indexes

We now examine the impact of naïvely adding multiple single-attribute indexes to a table. For this section, drop the table and then add individual indexes for ALL of the columns we have been examining (`a2`, `a10`, `a100`, and `a1000`) and report on insert time and table space.

To confirm all the indexes have been added, execute: `SHOW INDEX FROM data;`

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| data  |          0 | PRIMARY  |            1 | a1          | A         |           0 | NULL     | NULL   |      | BTREE      |         |               |
| data  |          1 | foo2     |            1 | a2          | A         |           0 | NULL     | NULL   | YES  | BTREE      |         |               |
| data  |          1 | foo10    |            1 | a10         | A         |           0 | NULL     | NULL   | YES  | BTREE      |         |               |
| data  |          1 | foo100   |            1 | a100        | A         |           0 | NULL     | NULL   | YES  | BTREE      |         |               |
| data  |          1 | foo1000  |            1 | a1000       | A         |           0 | NULL     | NULL   | YES  | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
```

As with the prior section, now add the data to collect insert time/space data. . .

DP_INSERT_TIME_ALL) _____ **seconds/10,000 rows**

DP_SPACE_ALL) _____ **MB**

and collect data query time with multiple attributes. . .

{ a2 = 1, a1 = 1 } _____ seconds

{ a2 = 1, a10 = 1 } _____ seconds

{ a2 = 1, a100 = 1 } _____ seconds

{ a2 = 1, a1000 = 1 } _____ seconds

{ a2 = 1, a1000 = 1, a1 = 1 } _____ seconds

{ a2 = 1, a1000 = 1, a10 = 1 } _____ seconds

{ a2 = 1, a1000 = 1, a100 = 1 } _____ seconds

4.2.1 Analysis

Now discuss your results, covering at least a response to all the following questions. What was the overall impact of adding these indexes in terms of insert time/space (in terms of percentage increase)? To what extent did query time decrease in the test queries? For the multi-attribute queries, was there a pattern for when MariaDB chose which index(es)? If you were dealing with a database that had a high write-to-read ratio, would you advise this strategy? What about the reverse? Are there particular query loads that would make this indexing strategy more appropriate?

4.3   Multi-Attribute Indexes

We now focus on indexes that use multiple attributes, and in particular the degree to which order affects applicability of the index, as well as speed. First, drop the table and re-create the schema; next, focus on the attribute set {a2, a1000}: collect all the data in the table's first column below for a single index declared as `CREATE INDEX first ON data(a2, a1000);`. Second, drop the table and re-create the schema; now `CREATE INDEX second ON data(a1000, a2);` and collect the second column's data. Note that the order does matter, as you will see – both in terms of query-time benefits, as well as when the index can be used.

	(a2, a1000)	(a1000, a2)
Insert Time		
Space on Disk		
{ a2 = 1 }		
{ a10 = 1 }		
{ a100 = 1 }		
{ a1000 = 1 }		
{ a2 = 1, a10 = 1 }		
{ a10 = 1, a100 = 1 }		
{ a2 = 1, a1000 = 1 }		

4.3.1   Analysis

Now discuss your results, covering at least a response to all the following questions. How did the multi-attribute index compare to multiple single-attribute indexes? What were the trends you discovered with respect to when a multi-attribute index could apply for a query (in full or part)? Under what situations would you use a multi-attribute query (with respect to insert/query ratio, query load)? What is a good rule of thumb for ordering attributes? Would there be any advantage to having a single-attribute index on (a2) if there were a multi-attribute index on (a2, a1000) vs. (a1000, a2)?