Northeastern University
CS5200 – Database Management Systems
Fall 2017, Derbinsky

HW5

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 ZIP file containing a single PDF document 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).

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. However, before you run MariaDB, we need to change its configuration, so edit the `my.ini` file (via the `config` button in XAMPP). Once open, find the line setting the value for the `max_allowed_packet` setting and change it to [at least] `10m`. Now save the file and start the MariaDB Server.

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 `mysql/bin` directory under your XAMPP installation. Once there, execute the following command: `mysql -u root` and you should be brought to a MariaDB prompt. If you encounter an error regarding a password, use `mysql -u root -p` and enter a password when prompted.

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).

3 Data Set

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

```
CREATE TABLE data (
  a0 INT ,
  a1 INT PRIMARY KEY ,
  a2 INT ,
  a5 INT ,
  a10 INT ,
  a50 INT ,
  a100 INT ,
  a500 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 `a`n, the values repeatedly count from 1 . . . 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:

```
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):

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

Finally, load the data into the table. The data we will use will include 25 million rows, and is thus relatively large. You will need to uncompress the supplied data file, and then source just as with the schema:

```
SOURCE path / to / d a t a _ 2 5 m _ 1 0 0 k . sql ;
```

The commands in this file are a set of `INSERT` statements, each adding 100,000 rows (i.e. 0.4%) to the `data` table. After each `INSERT` executes, you should see something like the following:

```
Query OK, 100000 rows affected (1.91 sec)
Records: 100000             Duplicates: 0Warnings: 0
```

If your insert time is consistently greater than 3-5 seconds, stop the process (`CTRL-C`). You likely have another disk-intensive process running and/or an anti-virus. Identify and stop the o↵ending program and restart.

Once the inserts complete, confirm that all the data was inserted successfully via the following queries (note that each will take a non-trivial amount of time, likely 10 – 30 seconds):

```
SELECT COUNT(*) FROM data; -- should return 25,000,000
SELECT SUM(a0) FROM data; -- should return 0
SELECT SUM(a2) FROM data; -- should return 37,500,000
SELECT SUM(a5) FROM data; -- should return 75,000,000
SELECT SUM(a10) FROM data; -- should return 137,500,000
SELECT SUM(a50) FROM data; -- should return 637,500,000
SELECT SUM(a100) FROM data; -- should return 1,262,500,000
SELECT SUM(a500) FROM data; -- should return 6,262,500,000
SELECT SUM(a1000) FROM data; -- should return 12,512,500,000
```

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, `1.91 sec`) for the last 5 queries and enter that number below:

DP INSERT TIME 0) _ **1.602** seconds/100,000 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) **1298** 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:

```
select      count(distinct a2)
from        data
```

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

DP DISTINCT TIME 0) _ 5.01 seconds

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 database):

```
+------+-------------+-------+------+---------------+------+---------+------+----------+-------+
| id   | select_type | table | type | possible_keys | key  | key_len | ref  | rows     | Extra |
+------+-------------+-------+------+---------------+------+---------+------+----------+-------+
|    1 | SIMPLE      | data  | ALL  | NULL          | NULL | NULL    | NULL | 24323773 |       |
+------+-------------+-------+------+---------------+------+---------+------+----------+-------+
```

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 | 24323773 | 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 }	0.00	seconds
{ a2 = 1 }	5.33	seconds
{ a10 = 1 }	4.91	seconds
{ a100 = 1 }	4.79	seconds
{ a1000 = 1 }	4.82	seconds
{ a2 = 1, a1 = 1 }	0.00	seconds
{ a2 = 1, a10 = 1 }	5.60	seconds
{ a2 = 1, a100 = 1 }	5.97	seconds
{ a2 = 1, a1000 = 1 }	5.99	seconds
{ a2 = 1, a1000 = 1, a1 = 1 }	0.00	seconds
{ a2 = 1, a1000 = 1, a10 = 1 }	5.67	seconds
{ a2 = 1, a1000 = 1, a100 = 1 }	5.73	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.

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.

Indexes created on the attributes that has higher selectivity reduces the execution time of the query. Selectivity is nothing but uniqueness of the attribute, primary key has all unique values the its selectivity is very high and hence default indexes make use of priomary keys.

In query 3.2, while finding the count of distinct values, whenever where clause contained of a1 = 1, db made use of the 'index' and the execution was very fast as compared to all other query executions. For all other queries that has non-primary key attributes in where clause, database has to go row by row and check each condition. Thus, for those queries, amount of time taken is directly proportional to number of conditions in the where clase.

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/100k rows)	Space on Disk (MB)	Query Time (seconds)
a2	**2.48**	**1617.7**	**4.25/2.61**
a10	**2.17**	**1646.2**	**4.90/0.63**
a100	**2.61**	**1697.9**	**4.99/0.07**
a1000	**8.12**	**1889.1**	**5.98/0.02**

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.[1]

4.1.2 Create the Index

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

[1] 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 25 million rows of data. Remember to (1) take the average of the last 5 batches of 100k rows, and make this the result in the second column of the ta-ble above and (2) perform the catalog-space query from above to fill in the third column in the ta-ble. 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 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?

For any table 'insertion time' is directly proportional to number of indexes (on high cardinality column) created on that table, as it has to create, mainatain and then insert the data in the table. That's why significant increase can be seen in the average increase time when index attribute has large domain values.

Space used is directly proportional to cardinality of attributes in that table. Thus higher the domain range of an attribute higher the space taken by table in memory.

Query performance can be improved by creating an index on the attribute that has high cardinality. Such indexing decereases the query execution time significantly however the down side is it also increases the insertion time.

4.2 Multiple Single-Attribute Indexes

We now examine the impact of naïvely adding multiple single-attribute indexes to a table. For this section, 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.

DP _INSERT _TIME _ALL) 15.21 seconds/100,000 rows

DP _SPACE _ALL) 2759.67 MB

Now collect data query time with multiple attributes:

{ a2 = 1, a1 = 1 }	0.00	seconds
{ a2 = 1, a10 = 1 }	3.53	seconds
{ a2 = 1, a100 = 1 }	2.78	seconds
{ a2 = 1, a1000 = 1 }	2.59	seconds
{ a2 = 1, a1000 = 1, a1 = 1 }	0.00	seconds
{ a2 = 1, a1000 = 1, a10 = 1 }	1.57	seconds
{ a2 = 1, a1000 = 1, a100 = 1 }	1.03	seconds

4.2.1 Analysis

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 insert-to-query ratio, would you advise this strategy? What about the reverse? Are there particular query loads that would make this indexing strategy more appropriate?

Creating indexes on many attribute had drastic effect on the query performance. The insetion time increased by more than 900% and the space consumption also increased by more than 200%. This is because number of indexes increases the maintainence cost of the table which increases in insertion time and to save the index information, table tajes up more space on disk.

Query time decreased by more than 50% for two non primary key attributes whereas, for three non primary key attributes that are present in index the exection time decreased by more than 250%. Maria DB uses 'index_merge' pattern while multiple attribute query exection. That is db chooses the intersection of the attributes (most selective) in the where clause to expedite the exection process.

When dealing with a database with high insert-to-query ratio, if number of queries executed on database is very high as compared to insert, it is better to include more column from where clause into index. However, if database has more insert queries or the insert-to-query ratio is very minute at that time it is safe to create indexes on only those columns which are very important.

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, we are going to focus on the attribute set {`a2`, `a1000`}: you are to gather insert time/space and query data twice, once for a single index declared as `CREATE INDEX first ON data(a2, a1000);` and the second as `CREATE INDEX second ON data(a1000, a2);`. 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.

7.43

	(a2, a1000)	(a1000, a2)
Insert Time	7.43	7.64
Space on Disk	1892	1803
{ `a2` = 1 }	3.01	5.02
{ a10 = 1 }	4.83	4.89
{ a100 = 1 }	4.57	4.61
{ a1000 = 1 }	4.38	0.03
{ `a2` = 1, `a10` = 1 }	400.3	5.47
{ a10 = 1, a100 = 1 }	5.03	5.12
{ `a2` = 1, `a1000` = 1 }	0.02x	0.03

4.3.1 Analysis

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? Final question: 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`)?

Multi attribute index reduced the insertion time as compared to the multiple single attribute indexes however, it did not benefit the query exection all the time.

If all attributes of the where clause are covered as an index then the execution time is reduced significantly on the other hand when only few columns out of the where clause were present in the index, it did not help the execution at all.

In case of high query-to-insert ration (queries fired are very high as compared to insert) making multiple-attribute index on columns that appear in the where clause most of the time is a good strategy.

Good rule of thumb is to place most selective attributes at the beginning of the index.

Creating redundant indexes has more cons as compared to the minute pros. So I think if (only) a2 is not appearing specifically in where clause of most of the queries then creating single attribute index on will not help.