Hand-in 4

The effects of indexes, documented experimentally

Jacob Hartmann, Stefan Jacobsen, Villads Nielsen, Thomas Richardy

Creation of Large, Related Tables

We chose to create three tables containing 2 million rows of generated data. The three tables were named Person, Places and Vehicles as an example of a use case. An attempt was made to create tables of 28 million rows of generated data, but the query times this resulted in were generally unmanageable.

Time Consuming Queries

Query 1.

```
SELECT Person.personName, Vehicles.serialNumber, Places.address
        FROM Person
           INNER JOIN Vehicles ON Person.personName = Vehicles.serialNumber
           INNER JOIN Places ON Person.personName = Places.address;
```

Query 2.

```
SELECT Persons.personName, Vehicles.serialNumber, Places.address
       FROM Persons
       INNER JOIN Vehicles
       INNER JOIN Places
              ON Persons.personId = Vehicles.vehicleId
              AND SUBSTRING(Persons.personName, 1, 3) ='9ge'
              AND SUBSTRING(Places.address, 1, 3) ='9ge';
```

912 rows in set (122.242 sec (Duration) / 338.366 sec (Fetch))

Query 3.

```
SELECT Person.personName, Vehicles.serialNumber, Places.address
        FROM Persons
        INNER JOIN Vehicles
        INNER JOIN Places
    ON SUBSTRING(Persons.personName, 1, 3) ='9ge' AND SUBSTRING(Places.address, 1, 3)
='9ge' OR SUBSTRING(Vehicles.serialNumber,1,3)='9ge';
```

(See following questions for in depth explanation)

All three queries are designed to show why indexes are a good idea in some situations - situations where a rather large amount of data needs to be evaluated if the data isn't sorted.

Define reasonable indexes for your tables.

Index 1.
```
CREATE INDEX partOfPersonName ON Persons(personName(10));
```

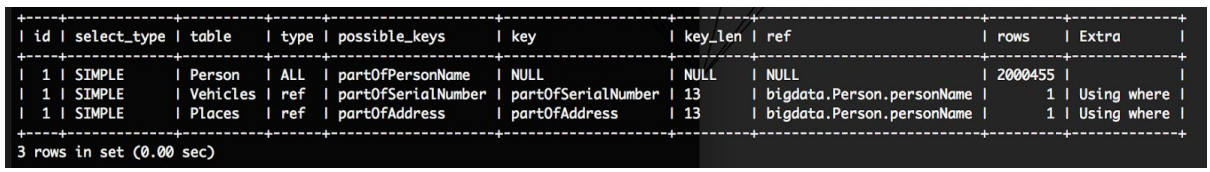
Index 2.
```
CREATE INDEX partOfSerialNumber ON Vehicles(serialNumber(10));
```

Index 3.
```
CREATE INDEX partOfAddress ON Places(address(10));
```

All indexes have been created with the assumption that you would want to find matches between the first ten symbols of persons' names, serial numbers and addresses faster. Furthermore, all indexes have been created under the assumption that each of the areas they define, are concerned with information that would be accessed often, but result in large query times if queried without indexes.

The Effects of the Indexes

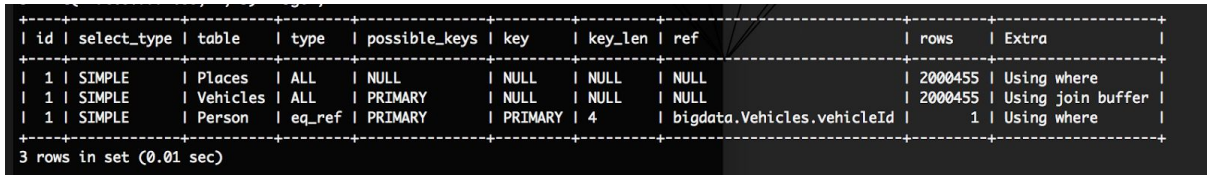
Query 1.
```
EXPLAIN SELECT Person.personName, Vehicles.serialNumber, Places.address
        FROM Person
           INNER JOIN Vehicles ON Person.personName = Vehicles.serialNumber
           INNER JOIN Places ON Person.personName = Places.address;

   EXPLAIN EXTENDED SELECT partOfPersonName.personName,
partOfserialNumber.serialNumber, partOfAddress.Address
```



```
+----+-------------+---------+------+-------------------+-------------------+---------+-------------------------+---------+-------------+
| id | select_type | table   | type | possible_keys     | key               | key_len | ref                     | rows    | Extra       |
+----+-------------+---------+------+-------------------+-------------------+---------+-------------------------+---------+-------------+
|  1 | SIMPLE      | Person  | ALL  | partOfPersonName  | NULL              | NULL    | NULL                    | 2000455 |             |
|  1 | SIMPLE      | Vehicles| ref  | partOfSerialNumber| partOfSerialNumber| 13      | bigdata.Person.personName |       1 | Using where |
|  1 | SIMPLE      | Places  | ref  | partOfAddress     | partOfAddress     | 13      | bigdata.Person.personName |       1 | Using where |
+----+-------------+---------+------+-------------------+-------------------+---------+-------------------------+---------+-------------+
3 rows in set (0.00 sec)
```

Query 2.

```
EXPLAIN SELECT Persons.personName, Vehicles.serialNumber, Places.address
        FROM Persons
        INNER JOIN Vehicles
        INNER JOIN Places
                ON Persons.personId = Vehicles.vehicleId
                AND SUBSTRING(Persons.personName, 1, 3) ='9ge'
                AND SUBSTRING(Places.address, 1, 3) ='9ge';
```

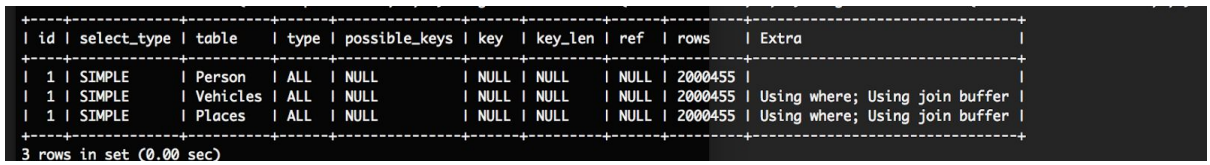


id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Places	ALL	NULL	NULL	NULL	NULL	2000455	Using where
1	SIMPLE	Vehicles	ALL	PRIMARY	NULL	NULL	NULL	2000455	Using join buffer
1	SIMPLE	Person	eq_ref	PRIMARY	PRIMARY	4	bigdata.Vehicles.vehicleId	1	Using where

3 rows in set (0.01 sec)

Query 3.

```
EXPLAIN SELECT Person.personName, Vehicles.serialNumber, Places.address
        FROM Persons
        INNER JOIN Vehicles
        INNER JOIN Places
   ON SUBSTRING(Persons.personName, 1, 3) ='9ge' AND SUBSTRING(Places.address, 1, 3)
='9ge' OR SUBSTRING(Vehicles.serialNumber,1,3)='9ge';
```



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	Person	ALL	NULL	NULL	NULL	NULL	2000455	
1	SIMPLE	Vehicles	ALL	NULL	NULL	NULL	NULL	2000455	Using where; Using join buffer
1	SIMPLE	Places	ALL	NULL	NULL	NULL	NULL	2000455	Using where; Using join buffer

3 rows in set (0.00 sec)

Show, using comparisons of time measurements, that the existence of indexes speeds up some queries. (at least two queries)

For query 1, we see a dramatic change between the indexed tables and the non-indexed. The indexed returns an empty set in 8.12 seconds, whereas the non-indexed didn't return an answer for more than 15 minutes.

For query 2, the query is answered in 16.22 seconds, with 3840 rows from the indexed tables, and doesn't return an answer in 20+ minutes for the unindexed database.

Query 3, is our special case. The query doesn't gain anything from our indexing, and thereby doesn't return an answer for 20+ minutes for both an indexed and unindexed database.

Explain in each case how the index helps the query evaluator.

Generally the indexing helps the query evaluator by narrowing down the possible rows in each table. This means that the query doesn't need do a full table search, which is the least optimal way of querying any table - unless this is more efficient than first having to create an index and then do an evaluation. But instead needs only to query the index, compare the index to the query, and then make a decision based on this.

As we can see in the EXPLAIN printout for our first query, this shows us that the query can use all of our indexings, and that it will use two references. This is almost an optimal query.

As for our second query, the EXPLAIN printout shows us that this query is able to both use the the primary keys of two tables Person and Vehicle, and one unique key for our Person table, which should be the most optimal way of querying an indexed table.

Last but not least, we can see in the EXPLAIN printout for our third query, that the query does a full table search on all of the tables, which will make it slow, even though we've indexed our database! The interesting thing in this query is that the ON clause, along with the two truth operators in use, the AND and OR, makes the query non-responsive to the indexing of the database, despite the fact that it doesn't look a lot different from our second query.