Databases and Info Systems

Advanced SQL - Indexes

Dr. Seán Russell
sean.russell@ucd.ie,

School of Computer Science,
University College Dublin

April 17, 2020

Specific Information

- Note: much of the information in this lecture is general and applies to all/most relational database management systems (RDBMS)
- However, some of the information will be specific to MySQL and to the default engine Innodb
- The general information about indexes and how they work should apply to all RDBMS
- But the specific information about how they are used to optimise queries may be implemented differently in other systems

Table of Contents

Big Databases

- Databases often contain hundreds of thousands or millions of rows of data
- Lets have a look at what kind of problems this might cause when performing queries
- We will look at an example table:

```
students(student_id, given_name, family_name, dob, class, email)
```

How big is Big

- This is a subjective opinion and is subject to change over time
- This is an approximation of what size a database is considered

Small	10^5 or fewer records
Medium	10^5 to 10^7 records
Large	10^7 to 10^9 records
Very Large	Greater than 10^9 records

- Lets assume that we have a big university, with 100,000 student records, then look at some example queries
- We are also assuming for now that there is no optimisation happening in the search
    1. Look up a student by ID
    2. Search for students by family name
    3. List all students in a class

- Look up a student by ID
  - E.g.

```
SELECT * FROM students WHERE student_id = "06373313";
```

  - **Without any optimisation**, we would need to search through the rows 1 by 1
  - On average if we have n rows, we will find the row we want in $\frac{n}{2}$ checks
  - This is $O(n)$ or linear time

- Search for students by family name
  - E.g.

```
SELECT * FROM students WHERE family_name = "Smith";
```

  - **Without any optimisation**, we would need to search through the rows 1 by 1
  - Because we are searching for all students with this name, we must search all n rows
  - This is $O(n)$ or linear time

- List all students in a class
  - E.g.

```
SELECT * FROM students WHERE class = "SE1";
```

  - **Without any optimisation**, we would need to search through the rows 1 by 1
  - Because we are searching for all students with this name, we must search all n rows
  - This is $O(n)$ or linear time

- MySQL provides the ability to understand how our queries are being executed
- This is done by adding the keyword `EXPLAIN` before the start of the query
- The results will give us some information about how efficiently the query is being executed

```
EXPLAIN SELECT * FROM students WHERE student_id =
    "06373313";
```

```sql
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

`id`

- `id` is a sequential number for this part of the query
- Where there are multiple parts to the query, such as when using joins or subqueries these are used to identify the different parts of the query

```
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

select_type

- This attributes describes they type of this part of the query
- There are many different types, below are a list of the more common one you will see
  - **SIMPLE** Simple SELECT (not using UNION or subqueries)
  - **PRIMARY** Outermost SELECT
  - **SUBQUERY** First SELECT in subquery

table and partitions

- `table` the name of the table that this row of the output is describing
- This may also describe the union of two the results of other rows (described in this explain) or the result of a subquery (also described in this explain)
- `partitions` describes which partitions are used in this query
  - Partitions are similar (but not quite the same) to the idea of file sharding in NoSQL

```sql
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

possible_keys and key

- The `possible_keys` column indicates the indexes from which MySQL can choose to find the rows in this table.
- Note that this column is totally independent of the order of the tables as displayed in the output from EXPLAIN.
- That means that some of the keys in `possible_keys` might not be usable in practice with the generated table order.
- The `key` column indicates the key (index) that MySQL actually decided to use.

```
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

keys_len and ref

- The key_len column indicates the length of the key that MySQL decided to use.
- The value of key_len enables you to determine how many parts of a multiple-part key MySQL actually uses.
- The ref column shows which columns or constants are compared to the index named in the key column to select rows from the table.

```sql
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

rows and filtered

- The `rows` column indicates the number of rows MySQL believes it must examine to execute the query
    - This number is an estimate
- The `filtered` column indicates an estimated percentage of table rows that will be filtered by the table condition
- The maximum value is 100, which means no filtering of rows occurred
- Values decreasing from 100 indicate increasing amounts of filtering

```sql
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

type

- This references how MySQL has chosen to perform the query
- This can refer to how it is joining two tables together or how the data is filtered to speed up the query
- There are many different values that are possible here, here are some of the most common:
  - `system/const`
  - `eq_ref`
  - `ref`
  - `index`
  - `ALL`

type - system/const

- The table has at most one matching row, which is read at the start of the query
- Because there is only one row, values from the column in this row can be regarded as constants by the rest of the optimizer.
- `const` (and `system`) tables are very fast because they are read only once
- These are used when you compare all parts of a PRIMARY KEY or UNIQUE index to constant values

type - eq_ref

- One row is read from this table for each combination of rows from the previous tables
- Other than the `system` and `const` types, this is the best possible join type
- `eq_ref` can be used for **indexed** columns that are compared using the = operator

type - ref

- All rows with matching index values are read from this table for each combination of rows from the previous tables
- `ref` is used if the join uses only a leftmost prefix of the key or if the key is not a PRIMARY KEY or UNIQUE index

type - index

- The index join type is the same as `ALL`, except that the index tree is scanned.
- An index-only scan usually is faster than `ALL` because the size of the index usually is smaller than the table data

type - all

- A full table scan is done for each combination of rows from the previous tables.
- This is normally not good if the table is the first table not marked const, and usually very bad in all other cases
- Normally, you can avoid ALL by adding indexes that enable row retrieval from the table based on constant values or column values from earlier tables.

```sql
EXPLAIN SELECT * FROM students WHERE student_id = "06373313";
```

```
+----+-------------+----------+------------+-------+---------------+---------+---------
| id | select_type | table    | partitions | type  | possible_keys | key     | key_len
+----+-------------+----------+------------+-------+---------------+---------+---------
|  1 | SIMPLE      | students | NULL       | const | PRIMARY       | PRIMARY | 32
+----+-------------+----------+------------+-------+---------------+---------+---------
1 row in set, 1 warning (0.01 sec)
```

```
+-------+------+----------+-------+
| ref   | rows | filtered | Extra |
+-------+------+----------+-------+
| const |    1 |   100.00 | NULL  |
+-------+------+----------+-------+
```

```sql
EXPLAIN SELECT * FROM students WHERE family_name = "Smith";
```

```
+----+-------------+----------+------------+------+---------------+------+---------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len |
+----+-------------+----------+------------+------+---------------+------+---------+
|  1 | SIMPLE      | students | NULL       | ALL  | NULL          | NULL | NULL    |
+----+-------------+----------+------------+------+---------------+------+---------+
1 row in set, 1 warning (0.00 sec)
```

```
+------+--------+----------+-------------+
| ref  | rows   | filtered | Extra       |
+------+--------+----------+-------------+
| NULL | 112568 |    10.00 | Using where |
+------+--------+----------+-------------+
```

```
EXPLAIN SELECT * FROM students WHERE class = "SE1";
```

```
+----+-------------+----------+------------+------+---------------+------+---------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len |
+----+-------------+----------+------------+------+---------------+------+---------+
|  1 | SIMPLE      | students | NULL       | ALL  | NULL          | NULL | NULL    |
+----+-------------+----------+------------+------+---------------+------+---------+
1 row in set, 1 warning (0.00 sec)
```

```
+------+--------+----------+-------------+
| ref  | rows   | filtered | Extra       |
+------+--------+----------+-------------+
| NULL | 112568 |    10.00 | Using where |
+------+--------+----------+-------------+
```

Table of Contents

File Storage

- The biggest limitation on the speed of queries in a database is reading the information from files on the hard disk
- As such, we must have a basic understanding of how databases are stored in files
- Deep understanding is not required, but we must know enough to understand how and why indexes improve performance

- Databases are typically stored on disk organised as files of records
  - Often a file per table with all rows (called records) in the same file
- There are several primary file organisations that can be used, these determine how the file records are physically placed on the disk
  - A **heap file** (unordered file)
  - A **sorted file** (sequential file)
  - A **hash file**
- We often use secondary organisations to allow efficient access based on other fields

Heap Files

- This is the simplest and most basic organisation
- Records are placed in the file in the order that they are inserted
- Inserting a new record to a hep file is very efficient (just add it to the end)
- Searching for a record requires a linear search

Sorted Files

- Records are placed in the file based on the values in one or more of their fields
- Inserting and deleting are slowed because we must always maintain the correct order
- Searching for a record requires a binary search

Hash Files

- Similar to operation of Hash maps and has tables
- Requires a hashing function to map records to their location on disk
- Can be very efficient, but difficult to implement

Other Fields

- The default engine in MySQL (InnoDB) uses sorted files based on the primary key of the table
- This means that if we are searching for something based on the key, it will be really fast
    - This is why our first example query only required checking 1 row
- But what about when we need to search based on the other attributes in the table?
- We can use an index to help with this

- An index is an extra **access structure**
- These are additional files that we use to help find the rows we want in table
- We can create an index for any attribute in the table
  - Or on multiple attributes
  - Or we can have multiple indexes that are for the same attributes

- An index in SQL works just like an index at the back of a book
- These list the page numbers where important terms appear
- This way we can find information quickly

Index

Indexes in Databases

- An index is a data structure that stores values and a pointer to the location that rows can be found in the data file
- Indexes are usually used with ordered files and use either a tree structure or are based on hashing
- This allows us to find the correct value in logarithmic or constant time
  - InnoDB uses B-Trees

Index File

key	pointer
07375301	
07375701	
10376301	
11378701	
16379601	
18377201	
18377401	
90371701	
90373801	
90376301	

Data File

student_id	given_name	family_name	dob	class	email
07375301	Charles	Andrew	1995-12-11	STAT3	Charles.Andrew@qq.com
07375701	Liam	Webb	1991-11-05	STAT7	Liam.Webb@qq.com
10376301	Billy	Stevenson	1998-08-09	MATH3	Billy.Stevenson@qq.com
11378701	Sam	Vincent	1995-08-11	EENG7	Sam.Vincent@qq.com
16379601	Bailey	Russell	1989-08-31	CS6	Bailey.Russell@qq.com
18377201	Naomi	Little	1999-07-10	AGG2	Naomi.Little@qq.com
18377401	Joel	Harrison	1995-06-28	AGG4	Joel.Harrison@qq.com
90371701	Sophia	Andrew	1997-09-01	IOT7	Sophia.Andrew@qq.com
90373801	Louie	Wells	1999-05-27	FIN8	Louie.Wells@qq.com
90376301	Mohammad	Russell	1985-05-17	MATH3	Mohammad.Russell@qq.com

Primary Index

- Every table has a **primary index** (sometimes called a clustered index)
- This is based on the primary key or our table
  - This is why it is important to choose a primary key
  - The last slide was an example of a primary index
- If we don't not choose a primary key, MySQL will create an index using a hidden automatically incremented number

Secondary Indexes

- We can also add additional indexes to our tables
- These will allow us to get the same access benefits when querying other attributes
  - These are called a **secondary index**
- Secondary indexes can be created using candidate keys or non-unique attributes

Unique Secondary Index

Secondary Index File

key	pointer
Bailey.Russell@qq.com	
Billy.Stevenson@qq.com	
Charles.Andrew@qq.com	
Joel.Harrison@qq.com	
Liam.Webb@qq.com	
Louie.Wells@qq.com	
Mohammad.Russell@qq.com	
Naomi.Little@qq.com	
Sam.Vincent@qq.com	
Sophia.Andrew@qq.com	

Data File

student_id	given_name	family_name	dob	class	email
07375301	Charles	Andrew	1995-12-11	STAT3	Charles.Andrew@qq.com
07375701	Liam	Webb	1991-11-05	STAT7	Liam.Webb@qq.com
10376301	Billy	Stevenson	1998-08-09	MATH3	Billy.Stevenson@qq.com
11378701	Sam	Vincent	1995-08-11	EENG7	Sam.Vincent@qq.com
16379601	Bailey	Russell	1989-08-31	CS6	Bailey.Russell@qq.com
18377201	Naomi	Little	1999-07-10	AGG2	Naomi.Little@qq.com
18377401	Joel	Harrison	1995-06-28	AGG4	Joel.Harrison@qq.com
90371701	Sophia	Andrew	1997-09-01	IOT7	Sophia.Andrew@qq.com
90373801	Louie	Wells	1999-05-27	FIN8	Louie.Wells@qq.com
90376301	Mohammad	Russell	1985-05-17	MATH3	Mohammad.Russell@qq.com

- In this example, we can find a single student immediately based on their email
- This can only be achieved because a secondary index is created for the unique attribute `email`

Non-Unique

- When an attribute (or set of attributes) is not unique, we can still get some benefit from indexes
- A non-unique index is sometimes known as a **clustering index**
- In this case, each unique value in the index is connected all of the records that it matches

Clustering Index File

key	pointer
Andrew	
Harrison	
Little	
Russell	
Stevenson	
Vincent	
Webb	
Wells	

Data File

student_id	given_name	family_name	dob	class	email
07375301	Charles	Andrew	1995-12-11	STAT3	Charles.Andrew@qq.com
07375701	Liam	Webb	1991-11-05	STAT7	Liam.Webb@qq.com
10376301	Billy	Stevenson	1998-08-09	MATH3	Billy.Stevenson@qq.com
11378701	Sam	Vincent	1995-08-11	EENG7	Sam.Vincent@qq.com
16379601	Bailey	Russell	1989-08-31	CS6	Bailey.Russell@qq.com
18377201	Naomi	Little	1999-07-10	AGG2	Naomi.Little@qq.com
18377401	Joel	Harrison	1995-06-28	AGG4	Joel.Harrison@qq.com
90371701	Sophia	Andrew	1997-09-01	IOT7	Sophia.Andrew@qq.com
90373801	Louie	Wells	1999-05-27	FIN8	Louie.Wells@qq.com
90376301	Mohammad	Russell	1985-05-17	MATH3	Mohammad.Russell@qq.com

- To create an index we have two options:
  1. Include it in the `CREATE TABLE` statement
  2. Use the `CREATE INDEX` statement
- Additionally, some indexes are created automatically

Automatically Created indexes

- When you define a primary key, the primary index is automatically created
  - This is why it is important to always create a primary key
- When you specify a unique constraint, a secondary index is created
  - This index is used to check if values are unique before inserting or changing them
- Foreign key constraints require an index to work, when you define foreign key constraints a secondary index is automatically created
  - This can be unique or non-unique
  - These are used to maintain the constraints and help improve join speed

- A secondary index can be added during the `CREATE TABLE` statement
- It is added in the constraints section using the keyword `INDEX`
- The list of attributes that are used to make the index are placed in brackets after it
- Lets assume that we knew that we would regularly be searching for students by family name and often ordering by family name and then first name

Indexes at Creation

```sql
CREATE TABLE students (
  student_id CHAR(8) PRIMARY KEY,
  given_name VARCHAR(30) NOT NULL,
  family_name VARCHAR(30) NOT NULL,
  dob DATE NOT NULL,
  class CHAR(8) NOT NULL,
  email VARCHAR(60) NOT NULL UNIQUE,
  INDEX (family_name, given_name)
)
```

1. `PRIMARY KEY` constraint creates primary index
2. `UNIQUE` constraint creates a secondary index
3. `INDEX` keyword creates a clustering index

Adding Indexes Later

- Lets assume that we discover that the query to find all the students in a class used often and is very slow
- We can add an index to speed this up
- We use the CREATE INDEX command

```
CREATE INDEX index_name ON table_name(column_name);
```

- E.g.

```
CREATE INDEX class_index ON students(class);
```

```sql
EXPLAIN SELECT * FROM students WHERE family_name = "Smith";
```

```
+----+-------------+----------+------------+------+-------------------------
| id | select_type | table    | partitions | type | possible_keys
+----+-------------+----------+------------+------+-------------------------
|  1 | SIMPLE      | students | NULL       | ref  | family_name,name_index
+----+-------------+----------+------------+------+-------------------------
1 row in set, 1 warning (0.00 sec)
```

```
+-------------+---------+-------+------+----------+-------+
| key         | key_len | ref   | rows | filtered | Extra |
+-------------+---------+-------+------+----------+-------+
| family_name | 122     | const |  197 |   100.00 | NULL  |
+-------------+---------+-------+------+----------+-------+
```

- This query now only looks at the rows with the correct family name

```
EXPLAIN SELECT * FROM students WHERE class = "SE1";
```

```
+----+-------------+----------+------------+------+---------------+-------------
| id | select_type | table    | partitions | type | possible_keys | key
+----+-------------+----------+------------+------+---------------+-------------
|  1 | SIMPLE      | students | NULL       | ref  | class_index   | class_index
+----+-------------+----------+------------+------+---------------+-------------
1 row in set, 1 warning (0.00 sec)
```

```
+---------+-------+------+----------+-----------------------+
| key_len | ref   | rows | filtered | Extra                 |
+---------+-------+------+----------+-----------------------+
| 32      | const | 1224 |   100.00 | Using index condition |
+---------+-------+------+----------+-----------------------+
```

- This query now only looks at the rows with the correct class

- To see the indexes that exist for a table in MySQL, we use the SHOW INDEX command

```
SHOW INDEX FROM table_name;
```

- This will give us the following information:
  - The name of the index
  - The columns involved and their order
  - If the index is unique or not
  - How many distinct values in the index (estimated)
  - The type of the index

Showing Indexes

```sql
SHOW INDEX FROM students;
```

```
+----------+------------+------------+--------------+-------------
| Table    | Non_unique | Key_name   | Seq_in_index | Column_name
+----------+------------+------------+--------------+-------------
| students |          0 | PRIMARY    |            1 | student_id
| students |          0 | email      |            1 | email
| students |          1 | family_name |           1 | family_name
| students |          1 | family_name |           2 | given_name
| students |          1 | class_index |           1 | class
+----------+------------+------------+--------------+-------------
```

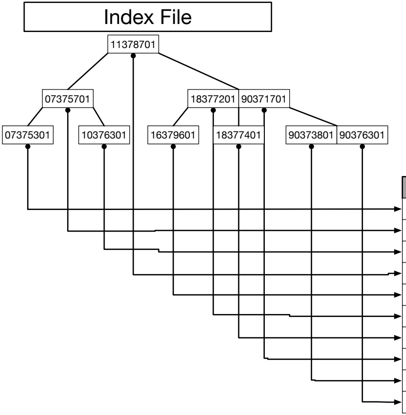
```
+-----------+-------------+------------+---------+------------+
| Collation | Cardinality | Index_type | Visible | Expression |
+-----------+-------------+------------+---------+------------+
| A         |      108000 | BTREE      | YES     | NULL       |
| A         |       99184 | BTREE      | YES     | NULL       |
| A         |         460 | BTREE      | YES     | NULL       |
| A         |       66790 | BTREE      | YES     | NULL       |
| A         |          82 | BTREE      | YES     | NULL       |
+-----------+-------------+------------+---------+------------+
```

- There are different types of indexes, but the one used most often is the B-Tree

- A B-Tree is a self balancing tree based on the Binary Search Tree you learned about in DSA 2
  - It is similar in some ways to AVL and Splay Trees

B-Trees

- Where B-Trees differ from these is the B-Trees are not binary
  - Each node can contain more than one value and have more than 2 children
  - If a node can contain x values, it can have $x + 1$ children
- B-Trees are useful when your data is too big to fit in memory
- This is designed to reduce the number of times you have to read from the hard drive



Data File

student_id	given_name	family_name	dob	class	email
07375301	Charles	Andrew	1995-12-11	STAT3	Charles.Andrew@qq.com
07375701	Liam	Webb	1991-11-05	STAT7	Liam.Webb@qq.com
10376301	Billy	Stevenson	1998-08-09	MATH3	Billy.Stevenson@qq.com
11378701	Sam	Vincent	1995-08-11	EENG7	Sam.Vincent@qq.com
16379601	Bailey	Russell	1989-08-31	CS6	Bailey.Russell@qq.com
18377201	Naomi	Little	1999-07-10	AGG2	Naomi.Little@qq.com
18377401	Joel	Harrison	1995-06-28	AGG4	Joel.Harrison@qq.com
90371701	Sophia	Andrew	1997-09-01	IOT7	Sophia.Andrew@qq.com
90373801	Louie	Wells	1999-05-27	FIN8	Louie.Wells@qq.com
90376301	Mohammad	Russell	1985-05-17	MATH3	Mohammad.Russell@qq.com

- Indexes can speed up our queries, but too many indexes may slow the database down or use too much space
- So how do we know when we should consider adding indexes?
- To answer this question, we need to consider the responsiveness of our information system
    - Remember our database is designed to support the information system

Responsiveness

- If your information system has an acceptable response time, then you do not need to add any indexes
- If some parts of your system are too slow, the queries involved need to be looked at
    - Look at the explain output for the query
    - If there are a large number of rows and the type is `ALL` you should consider adding an index