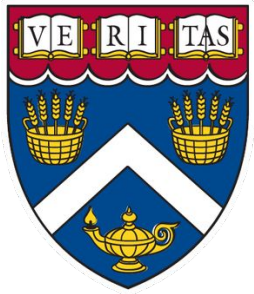
CSCI E-59 Designing & Developing Relational and Non-Relational Databases

Harvard University Extension, Spring 2022

Greg Misicko



Lecture 8 - Subqueries and Indexes

@GregMisicko

Agenda

- Subqueries
- Indexes
  - How can we optimize our search queries?
- Control Flow Functions:
  - CASE statements
  - IF/THEN (different from the IF/THEN we'll see in Stored Procedures)

Next Section

Monday, March 28, 8pm ET - Anthony
Monday, April 4, 8pm ET - Anthony
Monday, April 11, 8pm ET - Anthony

Homework 5 to be released tomorrow

What Are Subqueries

We've used subqueries in previous examples, but we've never dug into them explicitly. In particular we never talked about when we might choose or need to use them over a JOIN.

A subquery is a query nested within a SELECT, INSERT, UPDATE or DELETE statement, or within another subquery.

You'll often find that a subquery can be replaced with a JOIN, however there are situations where a subquery is your only option.

When you have a choice, is one option better than the other? Subqueries are generally not easy to interpret, and multiple level subqueries only get more confusing. Perhaps more concerning is that the performance of subqueries can be significantly worse than a JOIN, which optimizes the combination of data from multiple tables.

Deep thoughts on subqueries vs. JOINS:
https://stackoverflow.com/questions/4799820/when-to-use-sql-sub-queries-versus-a-standard-join

Subqueries

- a subquery is a query (a SELECT statement) inside of another query
- the first query is referred to as the outer query
- the second query is referred to as the inner query
- the inner query is processed first
- the output of the inner query is used as the input to the outer query
- the entire SQL statement can be referred to as a nested query
- an inner query can return one or more values

Subqueries

One example of a scenario where you cannot use JOIN to combine data is when using aggregate functions in a WHERE clause. For example, if we want to find the minimum value of our student's GPA's, it is easy to use the MIN() function to retrieve it. However, what if we wanted to know all students who have the lowest GPA value out of all students? There would seem to be two steps required to do this:

1. identify the minimum GPA
2. identify all students who have the minimum GPA

How can we combine two queries such as those together?

Subqueries: WHERE

Review note: The WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables

First of all, what is the lowest GPA in our records?

```
SELECT MIN(gpa) FROM students;
```

```
mysql> select min(gpa) from students;
+----------+
| min(gpa) |
+----------+
|      3.5 |
+----------+
1 row in set (0.00 sec)
```

Which gives us only the lowest GPA in our table.

Second, select all students. This will obviously list all columns and all rows for us:

```
SELECT * FROM students;
```

Finally, we can use these two queries together. We find the lowest gpa, and then select only the records which have a gpa equal to that value:

```
SELECT * FROM students WHERE gpa = (SELECT MIN(gpa) from students);
```

Subqueries: SELECT

Review note: SELECT is used to retrieve zero or more rows of a result set. SELECT identifies the columns you wish you retrieve. You can apply aggregate functions to your SELECT columns.

Suppose we wanted to know how each students GPA relates to the overall over GPA's of all students.

We could construct a query that selects our students and their GPA's, plus an average of all student's GPA's.

```
select last_name, gpa, (select avg(gpa) from students) from students;
```

Performance: Indexing

What is Indexing?

Let’s say that you are using a text book on the subject of databases and want to find out where in the book information for “Indexing” is located.

You could flip through all of the pages of the book, skimming for that topic, but this would not be the fastest or most efficient way of searching for that info.

Instead you would turn to the Index at the back of the book, where information is organized so that you can quickly locate the topic you are looking for.

Think of database indexing in a similar way - frequently accessed data is stored in a way where the lookup is optimized.

@GregMisicko

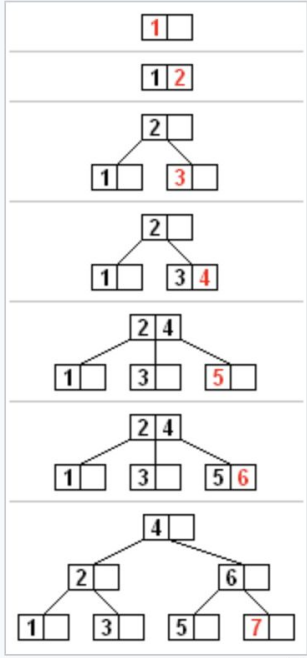
Indexing

Two of the most commonly used data structures for indexing are:

Hash Index: A hash index is an ordered list of hash values, and is probably a familiar concept to software developers. A hash algorithm is used to create a hash value from a key column which points to an entry in a hash table, which points to the location of the data row.

B-tree Index: In computer science, a B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree generalizes the binary search tree, allowing for nodes with more than two children.[2] Unlike other self-balancing binary search trees, the B-tree is well suited for storage systems that read and write relatively large blocks of data, such as discs. It is commonly used in databases and file systems. (https://en.wikipedia.org/wiki/B-tree)

Trivia: Bayer and McCreight (the inventors of the B-Tree) never explained what, if anything, the B stands for: Boeing, balanced, broad, bushy, and Bayer have been suggested.

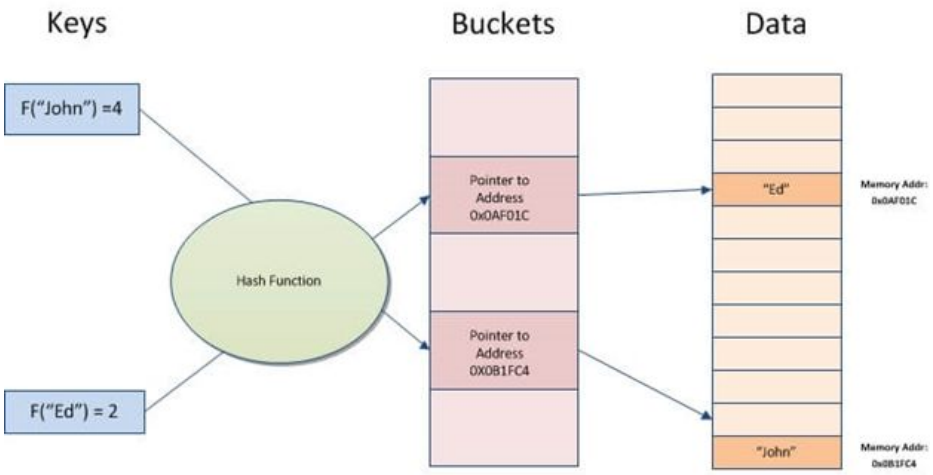


A B Tree insertion example with each iteration. The nodes of this B tree have at most 3 children (Knuth order 3).

Hash Indexes

https://codingsight.com/hash-index-understanding-hash-indexes

Hash Indexes work by creating an index key from the value and then locating it based on the resulting hash.



Indexing

The type of data structure used by your database is determined by the engine you are running. In MySQL the default is the B-Tree index, but other options are possible.

Storage Engine	Allowed Index Types
InnoDB	BTREE
MyISAM	BTREE
MEMORY/HEAP	HASH, BTREE

And according to https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html
Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; MEMORY tables also support hash indexes; InnoDB uses inverted lists for FULLTEXT indexes

Indexing

For a simple non-indexed SELECT statement, you should expect that the Big-O performance is going to be O(n), meaning that the performance will be directly related to the number of rows in the table as the lookup runs down all rows for matches.

What about other operations?
An INSERT statement would be the same speed regardless of how much data is in your database, and an UPDATE would be roughly equivalent to a SELECT because it also needs to search through your records.

“Indexing” organizes your data for more efficient lookups. Just like the index at the back of a textbook is organized to improve search efficiency, SQL databases use various types of data structures to store your information in a way that makes it faster to find what you are looking for.

Rusty on your Big-O notations?
https://www.bigocheatsheet.com/

Indexing Performance

Indexes improve the performance of our lookups, however there is an impact to the performance of our INSERT operations due to the maintenance required for our data structures. UPDATE can be impacted as well, although it probably gets advantages due to the improvements gained in its SELECT operation due to indexing.

Indexes can provide a significant boost to performance when retrieving data, so for SELECT statements that you expect to run frequently you can consider using indexes to provide speed improvements. However, there are some cases where you would not want to use indexing:

- on small tables (tables containing few rows of data) : Indexes are less important for queries on small tables, or big tables where report queries process most or all of the rows. When a query needs to access most of the rows, reading sequentially is faster than working through an index. Sequential reads minimize disk seeks, even if not all the rows are needed for the query (https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html)
- on columns that you're really not SELECT-ing from often. It's pretty common for people to over-index: the problem with this is the fact that it wastes space, and can slow down the RDBMS which is constantly checking to see which indexes it can use on SELECT statements
- on tables that are heavy on INSERT, UPDATE and DELETE operations as the indexes need to be updated after they occur

Indexing

Let’s see how indexing works.

First of all, we can list our indexes using the following command:

```
SHOW INDEX FROM <table>;
```

or

```
SHOW INDEXES FROM <table>;
```

You will notice when running this command on a table with a primary key, a **primary index** already exists. Any additional indexes you create on the same table are referred to as **secondary indexes**.

Indexing

Let’s see how indexing works.

First it helps to measure the performance of a query before we create an index (particularly since in this example I am working with a very small data set). We can see what the relative performance of our SELECT statement is by using EXPLAIN:

```
EXPLAIN SELECT <columns> FROM <table>;
```

```
mysql> explain select first_name from students;
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | students | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    7 |   100.00 | NULL  |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

MySQL EXPLAIN:
https://dev.mysql.com/doc/refman/8.0/en/explain-output.html

Indexing

To create an index at the time when you are creating a new table, you can identify it up front:

```
CREATE TABLE students
(
  student_id             INT unsigned NOT NULL AUTO_INCREMENT, # Unique ID for the record
  first_name             VARCHAR(20) NOT NULL,                # Student first name
  middle_initial         CHAR(1),                             # Optional middle initial
  last_name              VARCHAR(20) NOT NULL,                # Student last name
  gpa                    DECIMAL(2,1) NOT NULL                # Student grade point average
  CHECK(gpa <= 4 AND gpa >=0),                                # Set a constraint to ensure data integrity
  PRIMARY KEY           (student_id),                         # Make the StudentID the primary key
  INDEX (last_name);
);
```

Indexing

Or, you can create an index on an existing table:

```
CREATE INDEX <name> ON <table>( <column, or columns> );
```

You can, of course, remove indexes as well:

```
DROP INDEX <name> ON <table>;
```

There are certain things to consider when dropping an index that we won't cover here:
https://www.mysqltutorial.org/mysql-index/mysql-drop-index/

Indexing

Let's use the following sets of commands to try out indexing. We'll work with the database being used currently for Homework 5.

See which indexes already exist for our customers table:

```
show index from customers;
```

Let's say we are going to want to optimize searching by state.
How many rows are scanned to search for a specific state?

```
explain select first_name, last_name, state from customers where state='CA';
```

Now let's create a secondary index on state:

```
create index idx_state on customers(state);
```

Do we see that there would be a noticeable performance improvement (fewer rows scanned)?

```
explain select first_name, last_name, state from customers where state='CA';
```

Indexing: Multi-Column Indexes

https://dev.mysql.com/doc/refman/8.0/en/multiple-column-indexes.html

MySQL can create composite indexes (that is, indexes on multiple columns). An index may consist of up to 16 columns.

If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on (col1, col2, col3), you have indexed search capabilities on (col1), (col1, col2), and (col1, col2, col3).

Indexing: Multi-Column Indexes

https://dev.mysql.com/doc/refman/8.0/en/multiple-column-indexes.html

For example:
Say we create a composite index such as:

```
CREATE INDEX idx_name ON customers(first_name, last_name);
```

The following queries will benefit from the ‘idx_name’ index. Why?

```
SELECT * FROM customers WHERE first_name='Mark';

SELECT * FROM customers
  WHERE first_name='Mark' AND last_name='Benton';

SELECT * FROM customers
  WHERE first_name='Mark'
  AND (last_name='Benton' OR last_name='Garrett');

SELECT * FROM customers
  WHERE first_name='Mark'
  AND last_name >='F' AND last_name < 'N';
```

What happens when our WHERE clause begins with a last_name lookup?

Indexing: Index Prefixes

You don’t necessarily need to index the entire string you are trying to optimize. It may be enough to only index the leading characters of a field. By reducing the amount of information you are storing in you indexes you can save significant storage space.

For example:

```
CREATE INDEX idx_last_name
ON customers(last_name(2));
```

This doesn’t only provide us benefits if we are searching on the first two letters alone, we’ll still see benefits from searching on full last names.

Indexing: Invisible Indexes

It’s often a good idea to use EXPLAIN to determine whether or not you are getting benefits from an index in your queries.

You may also disable an index by using the INVISIBLE clause in order to see if your queries perform any differently when they are disabled:

ALTER TABLE table_name ALTER INDEX index_name [VISIBLE | INVISIBLE];

https://www.mysqltutorial.org/mysql-index/mysql-invisible-index/

Indexing: Hints

FORCE and USE

Without going into too much detail:
You may have multiple indexes which could provide benefits to your SELECT and UPDATE statements. Your RDBMS should do a very effective job identifying which index or indexes will provide you with the best performance. However, you have the ability to tell it that you know better, by using FORCE and USE.

For more details:
https://dev.mysql.com/doc/refman/8.0/en/index-hints.html

Indexing: FULLTEXT Indexes for Full-Text Search

https://dev.mysql.com/doc/refman/8.0/en/blob.html

Remember the TEXT data type? It can be used for large blocks of text, such as a product description or advertising statement. FULLTEXT indexes are created on text-based columns (CHAR, VARCHAR, or TEXT columns) to help speed up queries and Data Manipulation Language (DML) operations on data contained within those columns, omitting any words that are defined as stopwords.

FULLTEXT indexes allow you to index this large text object more efficiently.

https://dev.mysql.com/doc/refman/8.0/en/innodb-fulltext-index.html

Indexing: Cardinality

Remember the term Cardinality as we applied it to relationships? It refers to the number of items in a group, and when referring to indexing it identifies how many index entries we have.

When we look at index details, you will see that there is a column which identifies the cardinality of each index:

```
mysql> show index from customers;
+-----------+------------+---------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table     | Non_unique | Key_name      | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-----------+------------+---------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| customers |          0 | PRIMARY       |            1 | customer_id | A         |        1445 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| customers |          1 | idx_state     |            1 | state       | A         |           3 |     NULL |   NULL | YES  | BTREE      |         |               | YES     | NULL       |
| customers |          1 | idx_name      |            1 | first_name  | A         |        1265 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| customers |          1 | idx_name      |            2 | last_name   | A         |        1444 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| customers |          1 | idx_last_name |            1 | last_name   | A         |         145 |        2 |   NULL |      | BTREE      |         |               | YES     | NULL       |
+-----------+------------+---------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
```

Control Flow Functions

https://dev.mysql.com/doc/refman/5.6/en/case.html
These functions will feel comfortable to programmers, as they give you the ability to implement IF/THEN/ELSE logic.

CASE: this will be familiar to programmers; it works like a collection of IF, THEN, ELSE statements.

```
CASE case_value
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    …
    [ELSE else_result]
END
```

https://dev.mysql.com/doc/refman/8.0/en/if.html
IF: again very obvious to programmers. IF a condition is met, then set a result to a specified value. There are more advanced ways of using an IF statement than we will cover here (within functions, specifically), but a common example of the IF statement is when it is used with aggregate functions such as SUM or COUNT:

```
IF(expr,if_true_expr,if_false_expr);
e.g.:
SELECT SUM(IF(gpa='4.0',1,0)) from students;
```