BASIC DATABASE STRUCTURE AND QUERY TUNING

Index, please!

INDEXES

- An index is a fancy way of describing a structure used in a database to speed up searches. Consider:
  - `SELECT account_id FROM account`

    `WHERE balance = 12345.67;`
- Without an index, the database would have to crawl through every single row in the account table to locate each account with a current balance of 12345.67. This is called a *full table scan*, and it is one of the most expensive database operations out there, to be avoided at almost all costs!

INDEXES

- Fortunately, with an *index*, the database can locate the records many, many times faster (often many orders of magnitude faster)

- How the heck?

- Internally, an index uses one of several different complex computer science data structures to boil down and organize the information, allowing for rapid searches.

- How cool is that?!

WHY BOTHER WITH TABLES THEN?

- You might think “Why don’t we just use indexes instead of tables?”

- Well, interestingly enough, the index won’t actually hold the data in the table - it holds a reference to the *row* that holds the data.

- So while you might build an index on the balance column of the account table, when the database uses that index to quickly find every account with a balance of 12345.67, it’s actually just getting referred back to unique rows in the table which it knows have the right value.

INDEX EVERYTHING?

- Now you might think “Why don’t we index everything? Every column of every table. Index it all so any search I might do will be extremely fast!”
- On the outside, this seems like a fair idea. But, there are several considerations:
  - Indexes only help with certain *types* of searches (such as range searches and equality searches)
  - Indexes take up disk space
  - Indexes have to be updated any time the table is changed
  - Indexes can only be created on certain data types in certain engines

WHAT TO INDEX, THEN?!

- Frustrated, you might wonder “Well, what should I index, then?!”
- There is no perfect answer to this question, as it depends on:
  - Types of common queries
  - Data types
  - Size of table
- Rough idea: index identifier fields and consider indexing fields that are regularly searched by equality or range.

CREATING INDEXES

- Creating an index on an existing table is actually quite simple:
  - `CREATE INDEX name ON table ( columns... )`
- Example:
  - `CREATE INDEX bal_idx ON account ( balance );`
- But normally, we create indexes with table creation...

CREATING INDEXES

```
CREATE TABLE account (

    id          INT UNSIGNED NOT NULL,

    balance   DECIMAL( 15, 4 ),

              INDEX ( balance )

)
```

VIEWING INDEXES

- To see the indexes defined on a table:
  - `SHOW INDEX FROM table`
- Try:
  - `SHOW INDEX FROM movie;`

LAB

1) Create indexes on the actor and director tables for the *_id columns. Does it make sense to have an index for each field separately, or one combined index? Justify your answer.

2) Browse the documentation on creating indexes and some of the advanced topics presented therein.

EXPLAIN

- In a DBA's lifelong quest to tune `SELECT` queries, it is often helpful to see what the MySQL optimizer is doing.
- This is what the EXPLAIN command does:
  - `EXPLAIN select_statement`
- Example:
  - `EXPLAIN SELECT * FROM movie;`

EXPLAIN OUTPUT

- Section 7.2.2 covers the output in detail, but in short form:
- **Select type** - categorizes select
- **Table** - name of table being explained
  - The order indicates MySQL's read order
  - Remember, the FROM clause does not specify order
  - The order is chosen by MySQL's built-in optimizer, otherwise can be “hinted”

EXPLAIN OUTPUT

- **Type** - join type; how efficiently MySQL scans, ordered from best to worst:
  - *system* – fastest type
  - *eq_ref* - "=" referenced by a primary key or unique key (1 row)
  - *ref* - "=" by non-unique key (multiple rows)
  - *range* - references by <> or complex ranges
  - *index*
  - *ALL* - full table scan – slowest type

EXPLAIN OUTPUT

- **Possible keys** - which columns the optimizer could have used as indexes
- **Key** - index MySQL actually selected
- **Key length** - used key length in bytes
  - Check expected length is used for multiple column indexes
- **Ref** - the column or constant, key is matched against
  - Nulls in these columns indicate they are targets for improvement

EXPLAIN OUTPUT

- **Rows** - estimation of rows read
- **Extra** – extra information
  - *using index* - A covering index is used – a good thing!
  - *using where* - a where clause is used for filtering
  - *using filesort* - external sort is used
  - *using temporary* - temporary table will be used

EXPLAIN

- The output from EXPLAIN can be analyzed to find:
  - Candidates for indexes, as scans are bad!
  - Possible query rewriting, maybe even breaking into multiple queries in some cases.
  - Playing with join ordering.
- See section 7.2.1 for a preliminary discussion of optimization techniques for EXPLAIN output, and google for further details.

LAB

1) Put together a few SELECT statements, at least one or two including joins. Use EXPLAIN to see how the optimizer would execute your queries.

2) Bonus: Try to create some indexes that improve your execution plan reported by EXPLAIN.

3) Bonus bonus: Write a short script to insert a few hundred thousand fake records into the MovieCollection database. Then experiment with EXPLAIN and indexing to boost search performance.

QUERY PROFILER

- Way beyond a simple EXPLAIN, the query profiler can provide exceptionally detailed metrics on the actual execution of a statement.
  - Shows resource usage of the execution of a query
  - Introduced in mysql-server 5.0.37
  - Run on a per-session basis
  - Stores results in information_schema.PROFILING, a memory table unique to the session which is destroyed at disconnect

ENABLING PROFILING

- To enable the query profiler, just set a session variable:
  - `SET profiling = 1;`
- The profiler has a finite set of queries it can track data on:
  - 15 queries saved by default
  - Max is 100
  - `SET profiling_history_size = 100`

USING THE PROFILER

- To see all stored profiles:
  - `SHOW PROFILES;`
- And to view the metrics on a given query id:
  - `SHOW PROFILE type FOR QUERY query_id`
- Where type is one of the following...

PROFILE TYPES

- **ALL** – all information
- **BLOCK IO** – disk I/O mostly
- **CONTEXT SWITCHES** – (in)voluntary context switches
- **CPU** – CPU time, both system and user
- **IPC** – counts for msgs sent and received
- **MEMORY** – doesn't work
- **PAGE FAULTS** – counts for major/minor page faults
- **SOURCE** – shows functions from source code
- **SWAPS** – swap counts

LAB

1) Enable profiling for your mysql instance.

2) Experiment with a few queries and familiarize yourself with the profiler output. Research a bit into the documentation for the profiler and the current limitations.

ANALYZE

- The `ANALYZE` statement tells the server to re-analyze an index, computing a new key distribution and storing this information in the index summary.

- The key distribution is used by the optimizer in deciding the order for `JOIN`'s connected with something besides a constant.

- What?!

- Yeah, I know. Short of a looong discussion on index design, suffice it to say that performing a periodic `ANALYZE` on a very dynamic table will improve the optimizer's performance.

ANALYZE SYNTAX

- The syntax is simple:
  - `ANALYZE TABLE table`

OPTIMIZE

- As data is deleted and updated in a database, holes start to form in the underlying storage due to the changes.
- Normally, the storage engines continue refilling most of the holes, but this creates a largely fragmented table.
- The `OPTIMIZE TABLE` statement asks the storage engine to clean house:
  - `OPTIMIZE TABLE table`

SLOW QUERIES

- All of this talk about improving query performance.. If only there were some way to easily and quickly identify these slow queries..
- Oh wait, there is! That's what the slow query log is used for!

SLOW QUERIES

- The slow query log is not enabled by default. To enable it, add the following to `/etc/my.cnf` under the `[mysqld]` section:
  - `log-slow-queries=file`
  - `long_query_time=seconds`
- Records all queries taking longer than `long_query_time` to execute. The default is 10 seconds.
- These queries are often prime candidates for optimization!
- Use `mysqldumpslow` to summarize the log file.

LAB

1) Analyze, then optimize all tables in MovieCollection.

2) Enable the slow query log on your server, then simulate some slow queries using the `SLEEP(seconds)` function.

3) Use `mysqldumpslow` to view a report on your "slow" queries.

```
slideshow.end();
```