**MySQL Theory**

Transactions

A transaction is a group of SQL queries that are treated automatically, as a single unit of work. If the database engine can apply the entire group of queries to a database, it does so, but if any of them cannot be done because of a crash or other reason, none of them is applied.

START TRANSACTION;

// queries here;

COMMIT;

Storage engines

To determine what storage engine a particular table uses, use the SHOW TABLE STATUS cmd: SHOW TABLE STATUS LIKE ‘user’ \g

MyISAM engine

MyISAM is the MySQL default engine. It provides a good compromise between performance and useful features; such as full-table indexing, compression and spatial functions. MyISAM does not support transactions or row-level locks.

InnoDB engine

Designed for transactional processing.

Converting table from one engine to another

ALTER TABLE mytable ENGINE = innodb;

**Schema optimization and indexing**

Choosing optimal data types

MySQL supports a large variety of data types and choosing the correct type to store your data is crucial to getting good performance.

1. Smaller is usually better – in general, try to use the smallest data type that can correctly store and represent your data, because they use less space on the disk, in memory and in the CPU cache
2. Simple is good – e.g. integers are cheaper that characters
3. Avoid NULL if possible – its harder for MySQL to optimize queries that refer to nullable columns. Consider using zero, a special character or an empty string instead. If you are planning to index columns, avoid making them nullable if possible.

Data types

*whole numbers*: tinyint, smallint, mediumint, bigint

*floats:* float, double

*string:* varchar, char

*date + time:* datetime (8 bytes), timestamp(4 bytes)

Indexing basics

Indexes are data structures that help MySQL retrieve data efficiently. They are critical for good performance. If you index more than one column, the column order is very important, because MySQL can only search efficiently on a leftmost prefix of the index. Most SQL indexes (primary key, unique, index and fulltext) are stored in B-trees.

Index and table maintenance

Once you have created tables with proper data types and added indexes, your work isn’t over. You still need to maintain your tables and indexes to make sure they perform well.

3 goals:

1. Finding and fixing corruption
2. Maintaining accurate index statistics
3. Reducing fragmentation

Corruption can only happen from crashes etc… use CHECK TABLE *tablename* to see if corrupt. Use REPAIR TABLE to repair, or:

ALTER TABLE tablename ENGINE = *sameEngineAsBefore*

**Normalization**

Normalized updates are usually faster than denormalized tables. Normalized tables ensure no duplicated data so therefore less data to change. Normalized tables are usually smaller so they are better on memory.

1NF

1. Each table cell should contain a single value
2. Each record needs to be unique

2NF

1. Be in 1NF
2. Single column primary key

3NF

1. Be in 2NF
2. No transitive functional dependency, i.e. we have relationships between tables

**Query performance optimization**

The most basic reason a query doesn’t perform well is because its working with too much data.

2 ways to analyze a poorly performing query:

1. Find out whether your application is retrieving more data than you need (look at columns and rows)
2. Find out whether the MySQL server is analyzing more rows than it needs

Use LIMIT to only retrieve the rows you need. Only retrieve the columns you need data from.

COUNT()

Works in 2 ways:

1. Counts values and rows
2. Counts number of rows in a result

For (2) always use COUNT(\*) and never use the column name in the parenthesis.

Optimizing JOIN queries

Make sure there are indexes on the columns in the ON clauses.

Ensure any GROUP BY or ORDER BY expressions refer only to columns from a single table, so MySQL can try to use an index for that operation.

**Scale and high availability**

Load balancing places a *load balancer* in front of the servers which then routes incoming connections to the least-busy available server.

Placing all your data in a single MySQL instance is not an approach that will scale well. Sooner or later you will hit performance bottlenecks caused by an increase load on the server.

Scaling vertically/up – buy more powerful servers

Scaling horizontally/out – divide work across many servers

Data sharding

This is the most common and successful approach for scaling todays very large MySQL apps. Shard/split the data in to smaller pieces and store them on different nodes.

Clustering

A clustered system consists of several hosts on a local area network configured to appear as a single server.