Here is a summary of all the performance best practices we discussed in this course. This list is not complete by any means, but it contains guidelines that can help solve a lot of performance problems. If you're interested in learning more about this topic, take a look at the additional resources later in this document.

1. Smaller tables perform better. Don't store the data you don't need. Solve today's problems, not tomorrow's future problems that may never happen.

2. Use the smallest data types possible. If you need to store people's age, a TINYINT is sufficient. No need to use an INT. Saving a few bytes is not a big deal in a small table, but has a significant impact in a table with millions of records.

3. Every table must have a primary key.

4. Primary keys should be short. Prefer TINYINT to INT if you only need to store a hundred records.

5. Prefer numeric types to strings for primary keys. This makes looking up records by the primary key faster.

6. Avoid BLOBs. They increase the size of your database and have a negative impact on the performance. Store your files on disk if you can.

7. If a table has too many columns, consider splitting it into two related tables using a one-to-one relationship. This is called *vertical partitioning*. For example, you may have a customers table with columns for storing their address. If these columns don't get read often, split the table into two tables (users and user_addresses).

8. In contrast, if you have several joins in your queries due to data fragmentation, you may want to consider denormalizing data. Denormalizing is the opposite of normalization. It involves duplicating a column from one table in another table (to reduce the number of joins) required.

9. Consider creating summary/cache tables for expensive queries. For example, if the query to fetch the list of forums and the number of posts in each forum is expensive, create a table called forums_summary that contains the list of forums and the number of posts in them. You can use events to regularly refresh the data in this table. You may also use triggers to update the counts every time there is a new post.

10. Full table scans are a major cause of slow queries. Use the EXPLAIN statement and look for queries with type = ALL. These are full table scans. Use indexes to optimize these queries.

11. When designing indexes, look at the columns in your WHERE clauses first. Those are the first candidates because they help narrow down the searches. Next, look at the columns used in the ORDER BY clauses. If they exist in the index, MySQL can scan your index to return ordered data without having to perform a sort operation (filesort). Finally, consider adding the columns in the SELECT clause to your indexes. This gives you a *covering* index that covers everything your query needs. MySQL doesn't need to retrieve anything from your tables.

12. Prefer composite indexes to several single-column index.

13. The order of columns in indexes matter. Put the most frequently used columns and the columns with a higher cardinality first, but always take your queries into account.

14. Remove duplicate, redundant and unused indexes. Duplicate indexes are the indexes on the same set of columns with the same order. Redundant indexes are unnecessary indexes that can be replaced with the existing indexes. For example, if you have an index on columns (A, B) and create another index on column (A), the latter is redundant because the former index can help.

15. Don't create a new index before analyzing the existing ones.

16. Isolate your columns in your queries so MySQL can use your indexes.

17. Avoid SELECT *. Most of the time, selecting all columns ignores your indexes and returns unnecessary columns you may not need. This puts an extra load on your database server.

18. Return only the rows you need. Use the LIMIT clause to limit the number of rows returned.

19. Avoid LIKE expressions with a leading wildcard (eg LIKE '%name').

20. If you have a slow query that uses the OR operator, consider chopping up the query into two queries that utilize separate indexes and combine them using the UNION operator.

Additional Reading

High Performance MySQL: Optimization, Backups, and Replication by Baron Schwartz

Relational Database Index Design and the Optimizers by Tapio Lahdenmaki