11

Indexing MySQL

Indexes are a crucial part of any database. Many would even say that database performance without indexing is a myth – and that's not without a reason. Indexes are data structures that facilitate faster access to data by making use of hard drive space in the process thus, they aren't exactly revolutionary, but at the same time, the reason why this book has an entire chapter dedicated to keys (keys are a synonym to indexes) is that when used properly, these keys can unlock doors towards the database performance heaven in ways you couldn't even imagine.

Indexes don't help with everything – they can make `SELECT` queries faster, but at the same time harm the performance of `INSERT`s, `UPDATE`s, and `DELETE`s.

I'll repeat – indexes *can* make queries faster, but there's no guarantee that they *will*. Optimizing `SELECT` query performance is a task too – for your indexes to make `SELECT` queries faster, they have to be used by your database in the first place. This chapter will help you understand how, why, and when to use indexes to make your query performance faster.

Why Index? Indexes Available in MariaDB

Before you do anything in life, it's necessary to have answers to the question of why you do it. The same thing applies to indexes – why do you index data? What makes you think "Indexes may help here?" The answer is simple – indexes act as pointers to data facilitating faster access to that data.

Think of an index as an index in a book – if you want to read a specific chapter, what do you do? You turn to the beginning of the book and observe all of the chapters inside of it, then see what page the chapter begins from to see what page you should turn to. There's a good reason why you do it – if there are more than

400 pages, an index being unavailable would make the task of searching for that chapter much harder, but after you know that the chapter about indexes starts from page 157, you turn to page 157 and start reading.

When it comes to databases, things are a little more complex – different types of database management systems may offer different types of indexes you can choose from. Thus, indexing advice may differ from database to database. At the same time, advice may differ from DBMS to DBMS, but its main aspects remain the same as time goes by – understanding the core principles of indexes will always put you ahead of the curve, and thus, you need to be well-acquainted with the types of indexes that are available for you to use at all times.

At the same time, indexes are not a cure for all of your problems and they may also create problems in the long run, but they're extremely good and adept at solving speed problems when reading data.

The types of indexes in MySQL, MariaDB, and Percona Server are as follows:

1. **B-Tree indexes**: B-Tree indexes refer to Balanced Tree indexes and such indexes are the most frequently used index type within MySQL. Such indexes will be in use once our query is searching for exact matches of data with the equality operator and also if we're using the `LIKE` or `BETWEEN` operators. For queries to make use of such an index, we have to ensure that our queries specify an exact match by using operators like =, <, <=, >, or >=.
2. **R-Tree indexes**: R-Tree indexes are referred to as spatial indexes. Such an index type within MariaDB and its partners is used exclusively to index geographical objects.
3. **Hash indexes**: such indexes can be used only with the `MEMORY` storage engine and they can only be used in conjunction with SQL queries that use the equality operators. Queries using such indexes will be blazing fast, but such indexes can only be held in the memory and not on the disk, so the hash indexes have limited use cases.
4. **Covering indexes**: covering indexes within MariaDB "cover" all of the columns necessary for a query to execute. They're a special type of index

within MariaDB in that if such indexes are implemented correctly, your database will read data from the indexes and not from the disk, thus drastically reducing the time needed to access your data.

5. **Clustered indexes**: clustered indexes in MariaDB are `PRIMARY KEY`s, or if none are present within the table, `UNIQUE` indexes.
6. **Composite indexes**: composite indexes cover multiple columns and thus, they're sometimes referred to as multicolumn indexes. If an index resides on multiple columns at once, it is a composite, or a multicolumn, index.
7. **Prefix indexes**: lastly, prefix indexes allow us to only index a small part of a column. Such practice may be very welcome for those who have a lot of data to work with and don't want to index the entire value in a specific column.

Keep in mind that `PRIMARY KEY` constraints are also indexes, as are `UNIQUE` indexes.

<...>

"Why would it even matter if I create indexes anew or modify data within an existing table?", – you ask. Answers to these questions matter because behind each of them is hidden the functionality of your database:

1) Your use case depicts what type of index may be necessary for you to use.
2) The size of your data set dictates whether indexing makes sense in the first place.
3) The amount of available disk space and other specifics dictate whether indexing is an option, and if it is, directs you to a type of index.
4) Previous experience may help you overcome obstacles related to indexes.
5) Creating tables anew means that your database won't need to make a copy of the table on the disk – altering an existing table means that your database will make a copy of the table, copy all of the data there, perform all of the necessary operations, and swap the two tables.

No matter what the answers to these questions are, if you decide to index, you will have to index columns in a proper way to help your database and the queries within and when you do decide to use indexes, the first question you need an answer to will be *what to index*.

Types of Indexes

Take a look at this graph:

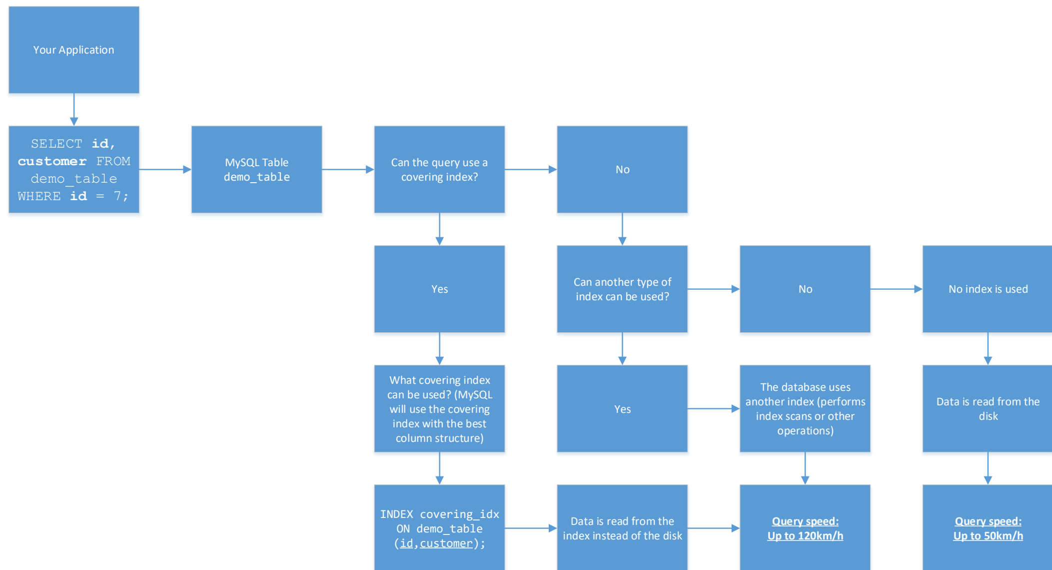


Image 1 – a Covering Index in Action

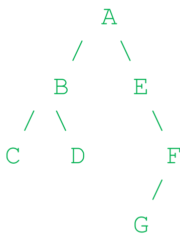
This graph depicts the functionality of a covering index within a database management system. Granted, the illustration is overly simplified (queries may differ and query speed is not measured in km/h – I've added that for illustration purposes), but it gets the point across: *indexes may make your queries faster.*

Indexes may help your query – they may not. There are indeed many index types and each index type helps a specific type of query complete faster. If you've defined an index or two in the past, chances are that you are already aware of the circumstances surrounding index types making you competent in coming up with conclusions and decisions to employ a specific index type, but even if you didn't, index types are not that hard to understand: flick a couple of pages back and refresh your memory around index types and dive into them.

B-Tree Indexes

B-tree indexes refer to balanced tree indexes. B-tree indexes are called that way because an ideal version of such an index would hold all of its leaf nodes at the same level visually speaking, thus providing a balance. In the real world, results are often far from perfect since developers frequently add/remove data concerning the indexed column in question thus paving the way for index fragmentation, but regardless, B-Tree indexes have a firm place in the database world and will likely retain it for years to come.

To understand B-Tree indexes, an understanding of a Binary Search Tree may help. A basic illustration of a binary search tree (BST) can be seen below:



A Binary Search Tree (BST) is a tree-like structure with nodes. Nodes within a BST may have up to two children to the sides and when you see such a structure, you begin to understand why queries using indexes are often significantly faster than those that read data from the disk.

B-tree indexes are binary tree indexes with a twist: such types of indexes allow for their members to have more than two children and that's their primary difference. Here's a basic illustration of a B-tree index:

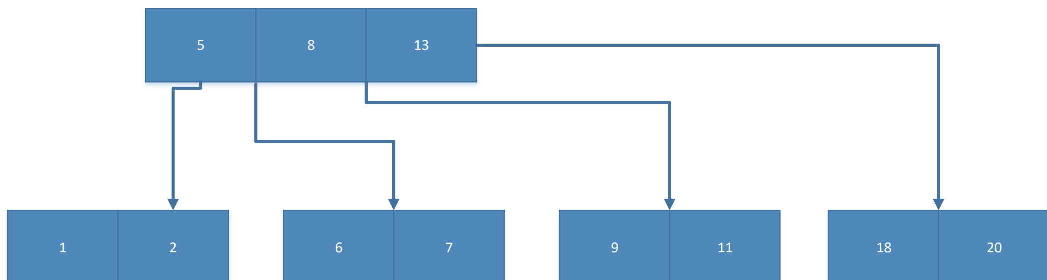


Image 2 – B-Tree Index

B-tree indexes make query performance faster because of their structure allowing them to enable our database to skip traversing through unnecessary rows: as they use pointers to quickly locate data, the intermediate levels of the index can have many more items per node, but regardless of how many items an individual node has, the premise of B-tree indexes remains the same.

In all flavors of MySQL, B-tree indexes can be defined upon table creation or when modifying the structure of a particular table. With that being said, regardless of how you elect to define B-tree indexes, their internal functionality doesn't change. The column that they reside on, their cardinality, and other specifics might but a large majority of those specifics are related to the internals of your queries and database. The internals of your queries dictate what kind of index to choose, too.

To make good use of a B-tree index, use equality operators, `BETWEEN`, or `LIKE` without a preceding wildcard. A B-tree index will also help if you find yourself using `IS [NOT] NULL` clauses, and it will also be helpful when using `AND` clauses.

Summary

Indexing is the backbone behind many modern-world applications: properly defined indexes excel at making `SELECT` queries blazing fast, and indexing can help even if you find yourself obstructed by a variety of factors like low disk space on the hard drive, inability to use certain data types, or other circumstances. Indexes do that by minimizing the amount of readable data in the database, but indexing is not the only way to make query performance skyrocket.

Enter partitions – mini tables inside of your table that split data into easy-to-access chunks further minimizing the scope of accessible data and further improving the performance of certain types of queries.