MySQL B+ tree

A typical B+tree

Why use B+tree?

B+tree is used for an obvious reason and that is speed. As we know that there are space limitations when it comes to memory, and not all of the data can reside in memory, and hence majority of the data has to be saved on disk. Disk as we know is a lot slower as compared to memory because it has moving parts. So if there were no tree structure to do the lookup, then to find a value in a database, the DBMS would have to do a sequential scan of all the records. Now imagine a data size of a billion rows, and you can clearly see that sequential scan is going to take very long.

But with B+tree, its possible to store a billion key values (with pointers to billion rows) at a height of 3, 4 or 5, so that every key lookup out of the billion keys is going to take 3, 4 or 5 disk accesses, which is a huge saving.

The reason why B+tree is chosen over other tree structures is that B+trees tend to be very shallow, and since every lookup translates to a disk access, the number of disk accesses required to fetch a value is directly proportional to the height of the tree, so the shallower the tree, the less number of disk accesses.
How is B+tree structured?

B+trees are normally structured in such a way that the size of a node is chosen according to the page size. Why? Because whenever data is accessed on disk, instead of reading a few bits, a whole page of data is read, because that is much cheaper.

Let us look at an example,
Consider InnoDB whose page size is 16KB and suppose we have an index on a integer column of size 4 bytes, so a node can contain at most $16 \times 1024 / 4 = 4096$ keys, and a node can have at most 4097 children.

So for a B+tree of height 1, the root node has 4096 keys and the nodes at height 1 (the leaf nodes) have $4096 \times 4097 = 16781312$ key values.

This goes to show the effectiveness of a B+tree index, more than 16 million key values can be stored in a B+tree of height 1 and every key value can be accessed in exactly 2 lookups.

How important is the size of the index values?

As can be seen from the above example, the size of the index values plays a very important role for the following reasons:

- The longer the index, the less number of values that can fit in a node, and hence the more the height of the B+tree.
- The more the height of the tree, the more disk accesses are needed.
- The more the disk accesses the less the performance.
- So the size of the index values have a direct bearing on performance!

MySQL Index on Integer type
MySQL Index on VarChar type
MySQL Index on Two Columns Integer and VarChar

Showing that MySQL is actually using your index (EXPLAIN)

EXPLAIN helps you to do this by showing how MySQL is approaching your query.

use ap;
desc invoice_line_items;
select * from invoice_line_items;
explain extended select * from invoice_line_items
where invoice_id = 1;

explain select * from invoice_line_items
where invoice_sequence = 1;
explain select * from invoice_line_items
where invoice_id = 1 and
(invoice_sequence = 1 or invoice_id = 2);

explain select * from invoice_line_items
where invoice_id = 78 and invoice_sequence = 2;

If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to find 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).
MySQL cannot use the index to perform lookups if the columns do not form a leftmost prefix of the index. Suppose that you have the SELECT statements shown here:
SELECT * FROM tbl_name WHERE col1=val1;
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;

SELECT * FROM tbl_name WHERE col2=val2;
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
If an index exists on (col1, col2, col3), only the first two queries use the index. The third and fourth queries do involve indexed columns, but (col2) and (col2, col3) are not leftmost prefixes of (col1, col2, col3).
Levels are numbered starting from 0 at the leaf pages, incrementing up the tree.
Pages on each level are doubly-linked with previous and next pointers in ascending order by key.
Records within a page are singly-linked with a next pointer in ascending order by key.
Infimum represents a value lower than any key on the page, and is always the first record in the singly-linked list of records.
Supremum represents a value higher than any key on the page, and is always the last record in the singly-linked list of records.
Non-leaf pages contain the minimum key of the child page and the child page number, called a "node pointer".
InnoDB table format is Barracuda with 'compact' record structure, non-compressed.

Table created with: CREATE TABLE t (i INT NOT NULL, s CHAR(10) NOT NULL, PRIMARY KEY(i)) ENGINE=InnoDB;
Table populated with: INSERT INTO t (i, s) VALUES (0, 'A'), (1, 'B'), (2, 'C');
Record size: 5 (header) + 4 (PK) + 6 (TRX_ID) + 7 (ROLL_PTR) + 10 (non-key fields) = 32 bytes