B-trees and database indexes

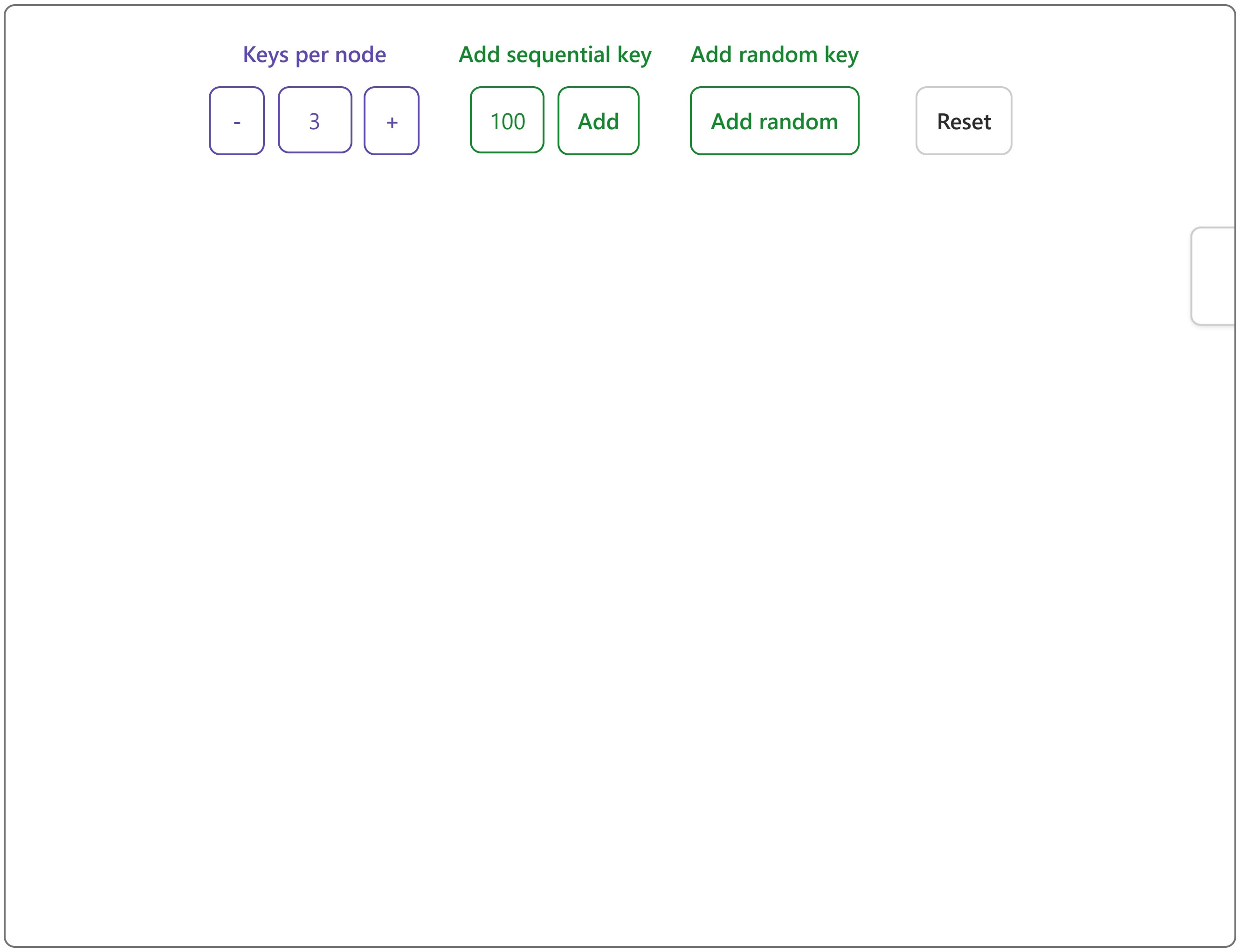
By Ben Dicken | September 9, 2024

What is a B-tree?

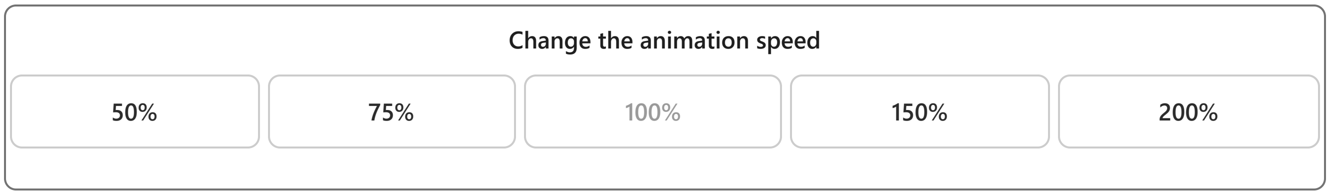
The B-tree plays a foundational role in many pieces of software, especially database management systems (DBMS). MySQL, Postgres, MongoDB, Dynamo, and many others rely on B-trees to perform efficient data lookups via **indexes**. By the time you finish this article, you'll have learned how B-trees and B+trees work, why databases use them for indexes, and why using a UUID as your primary key might be a bad idea. You'll also have the opportunity to play with *interactive* animations of the data structures we discuss. Get ready to click buttons.

Computer science has a plethora of data structures to choose from for storing, searching, and managing data on a computer. The B-tree is one such structure, and is commonly used in database applications. B-trees store pairs of data known as *keys* and *values* in what computer programmers call a tree-like structure. For those not acquainted with how computer scientists use the term "tree" it actually looks more like a root system.

Below you'll find the first *interactive* component of this blog. This allows you to visualize the structure of a B-tree and see what happens as you add key/value pairs and change the number of key/value pairs per node. Give it a try by clicking the Add or Add random button a few times and try to get an intuitive sense of how it works before we move on to the details.



If the animations above are too fast or slow, you can adjust the animation speed of everything that happens with the B-trees in this article. Adjust below:



Every B-tree is made up of **nodes** (the rectangles) and **child pointers** (the lines connecting the nodes). We call the top-most node the **root node**, the nodes on the bottom level **leaf nodes**, and everything else **internal nodes**.

The formal definition of a B-tree can vary depending on who you ask, but the following is a pretty typical definition.

A B-tree of order K is a tree structure with the following properties:

* **Each node in the tree stores N key/value pairs, where N is greater than 1 and less than or equal to K.**

* **Each internal node has at least N/2 key/value pairs (an internal node is one that is not a leaf or the root).**

* **Each node has N+1 children.**

* **The root node has at least one value and two children, unless it is the sole node.**

* **All leaves are on the same level.**

The other key characteristic of a B-tree is ordering. Within each node the elements are kept in order. Any child to the left of a key must only contain other keys that are less than it. Children to the right must have keys that are greater than it.

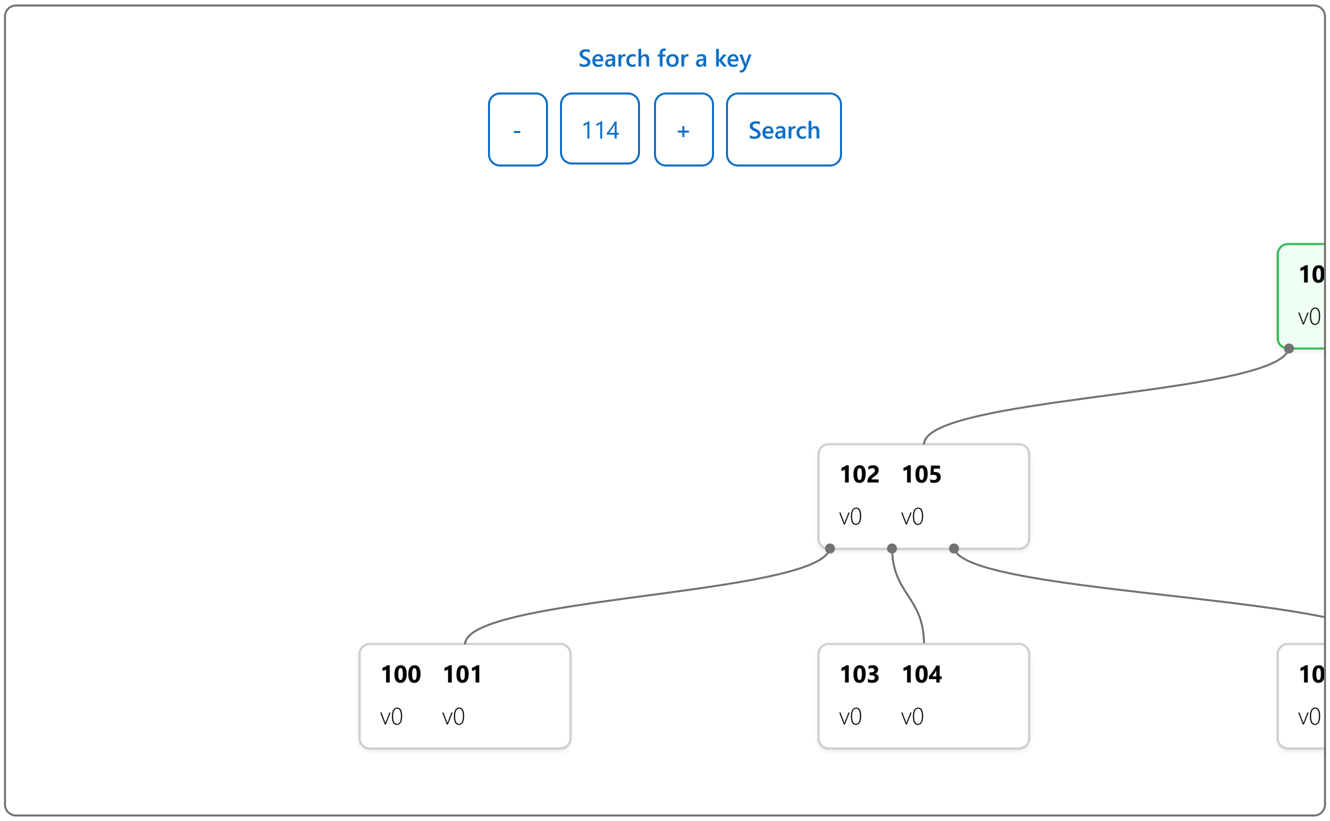
This enforced ordering means you can **search for a key** very efficiently. Starting at the **root node**, do the following:

1. Check if the node contains the key you are looking for.

2. If not, find the location in the node where your key would get inserted into, if you were adding it.

3. Follow the **child pointer** at this spot down to the next level, and repeat the process.

When searching in this way, you only need to visit *one* node at each level of the tree to search for one key. Therefore, the fewer levels it has (or the shallower it is), the faster searching can be performed. Try searching for some keys in the tree below:



B-trees are uniquely suited to work well when you have a *very large* quantity of data that also needs to be persisted to long-term storage (disk). This is because each node uses a fixed number of bytes. The number of bytes can be tailored to play nicely with **disk blocks**.

Reading and writing data on hard-drive disks (HDDs) and solid-state disks (SSDs) is done in units called **blocks**. These are typically byte sequences of length 4096, 8192, or 16384 (4k, 8k, 16k). A single disk will have a capacity of many millions or billions of blocks. RAM on the other hand is typically addressable on a per-byte level.

This is why B-trees work so well when we need to organize and store *persistent* data on disk. Each node of a B-tree can be sized to match the size of a disk block (or a multiple of this size).

The number of values each node of the tree can store is based on the number of bytes each is allocated and the number of bytes consumed by each key / value pair. In the example above, you saw some pretty small nodes — ones storing 3 integer values and 4 pointers. If our disk block and B-tree node is 16k, and our keys, values, and child pointers are all 8 bits, this means we could store 682 key/values with 683 child pointers per node. A three level tree could store over 300 *million* key/value pairs (682 × 682 × 682 = 317,214,568).

The B+Tree

B-trees are great, but many database indexes use a "fancier" variant called the B+tree. It's similar to a B-tree, with the following changes to the rules:

* **Key/value pairs are stored only at the leaf nodes.**

* **Non-leaf nodes store only keys and the associated child pointers.**

There are two additional rules that are specific to how B+trees are implemented in MySQL indexes:

* **Non-leaf nodes store N child pointers instead of N+1.**

* **All nodes also contain "next" and "previous" pointers, allowing each level of the tree to also act as a doubly-linked list.**

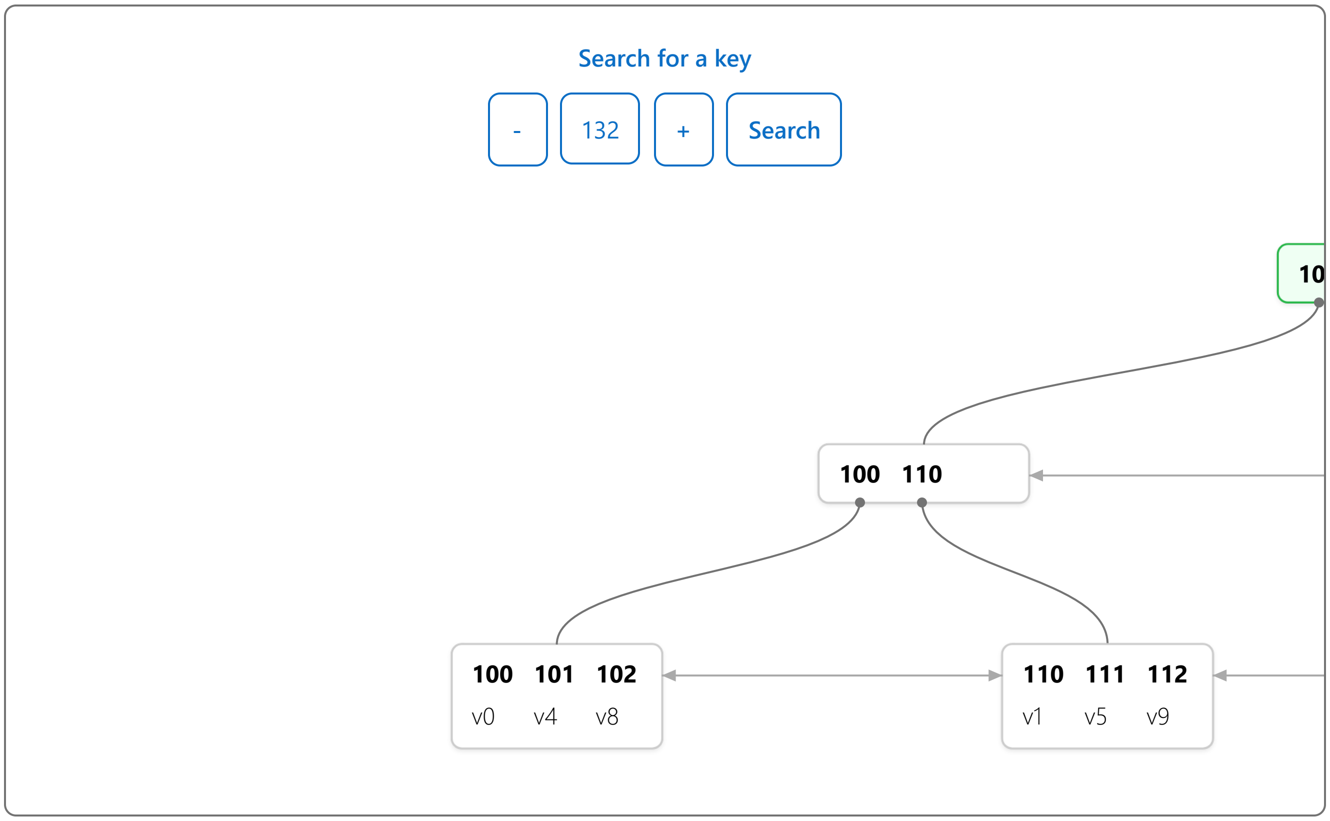
Here's another visualization showing how the B+tree works with these modified characteristics. This time you can individually adjust the number of keys in inner nodes and in the leaf nodes, in addition to adding key/value pairs.

Why are B+trees better for databases? There are two primary reasons.

1. Since inner nodes do *not* have to store values, we can fit more keys per inner node! This can help keep the tree shallower.

2. All of the *values* can be stored at the same level, and traversed in-order via the bottom-level linked list.

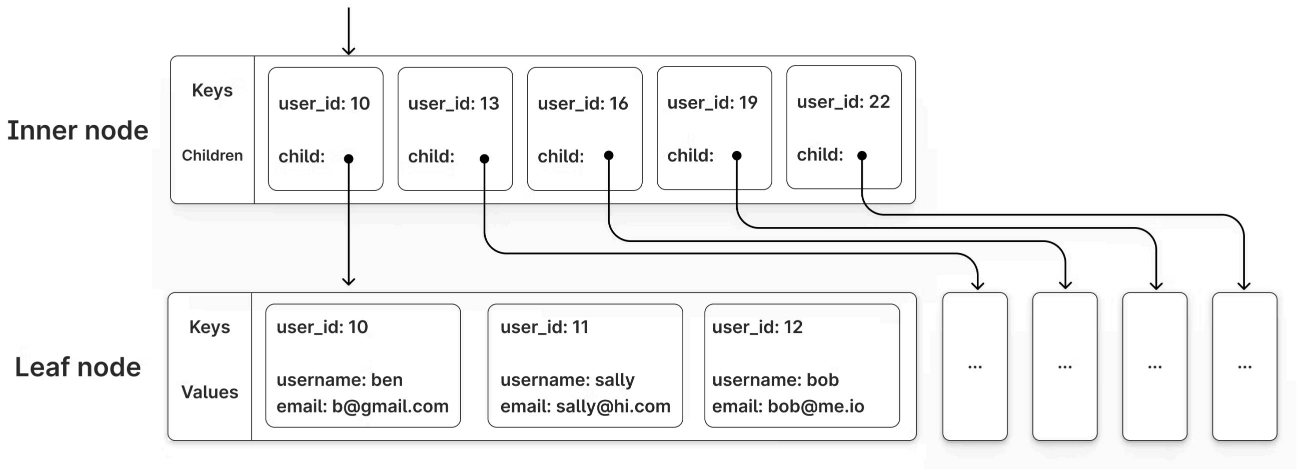
Go ahead and give searching on a B+tree a try as well:



B+trees in MySQL

MySQL, arguably the world's most popular database management system, supports multiple storage engines. The most commonly used engine is InnoDB which relies *heavily* on B+trees. In fact, it relies *so* heavily on them that it actually stores *all table data* in a B+tree, with the table's primary key used as the tree key.

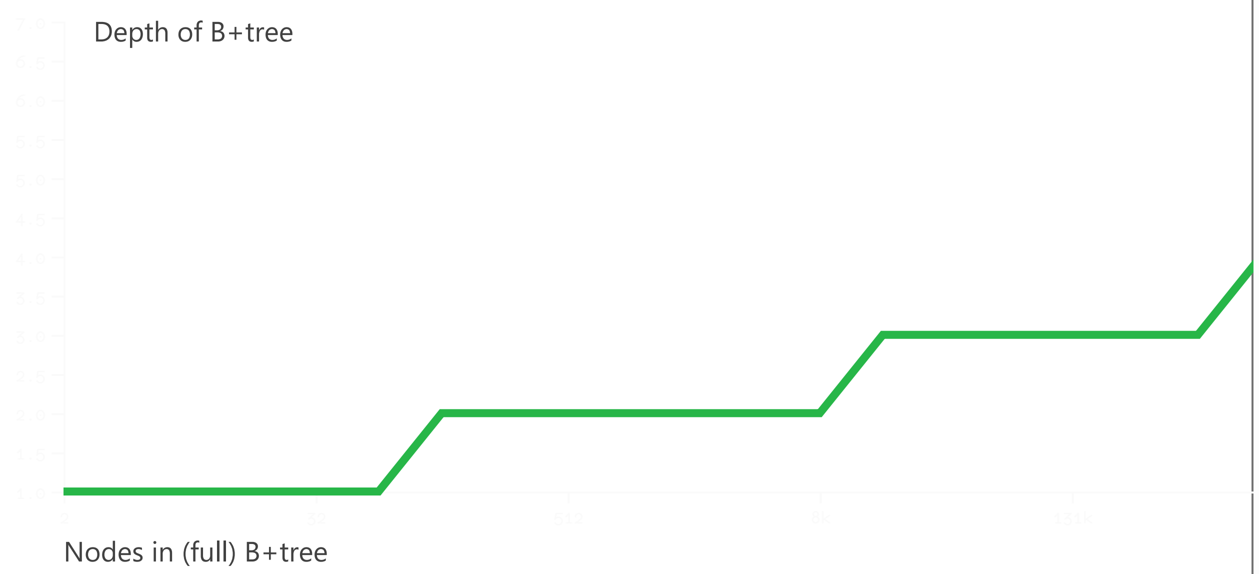
Whenever you create a new InnoDB table you are required to specify a primary key. Database administrators and software engineers often use a simple auto-incrementing integer for this value. Behind the scenes, MySQL + InnoDB creates a B+tree for each new table created. The keys for this tree are whatever the primary key was set to. The values are the remaining column values for each row, and are stored only in the leaf nodes.



The size of each node in these B+trees is set to 16k by default. Whenever MySQL needs to access a piece of data (keys, values, whatever), it loads the entire associated page (B+tree node) from disk, even if that page contains other keys or values it does not need.

The number of rows stored in each node depends on how "wide" the table is. In a "narrow" table (a table with few columns), each leaf could store hundreds of rows. In a "wide" table (a table with many columns), each leaf may only store a single-digit number of rows. InnoDB also supports rows being larger than a disk block, but we won't dig into that in this post.

Use the visualization below to see how the number of keys in each inner node and in each leaf node affect the depth of the tree. The deeper the tree, the slower it is to look up elements. Thus, we want shallow trees for our databases!



Keys per inner node: 100

Keys per leaf node: 100

It's also common to create secondary indexes on InnoDB tables — ones on columns other than the primary key. These may be needed to speed up WHERE clause filtering in SQL queries. An additional persistent B+tree is constructed for each secondary index. For these, the key is the column(s) that the user selected the index to be built for. The values are the **primary key** of the associated row. Whenever a secondary index is used for a query:

1. A search is performed on the secondary index B+tree.

2. The primary keys for matching results are collected.

3. These are then used to do additional B+tree lookup(s) on the main table B+tree to then find the actual row data.

Consider the following database schema:

```
CREATE TABLE user (
  user_id BIGINT UNSIGNED AUTO_INCREMENT NOT NULL,
  username VARCHAR(256) NOT NULL,
  email VARCHAR(256) NOT NULL,
  PRIMARY KEY (user_id)
);
CREATE INDEX email_index ON user(email);
```

This will cause two B+tree indexes to be created:

* One for the table's primary key, using `user_id` for the key and the other two columns stored in the values.

* Another for the `email_index`, with `email` as the key and `user_id` as the value.

When a query like this is executed:

```
SELECT username FROM user WHERE email = 'x@planetscale.com';
```

This will first perform a lookup for `x@planetscale.com` on the `email_index` B+tree. After it has found the associated `user_id` value it will use that to perform another lookup into the primary key B+tree, and fetch the username from there.

Overall, we'd like to always minimize the number of blocks / nodes that need to be visited to fulfill a query. The fewer nodes we have to visit, the faster our query can go. The **primary key** you choose for a table is pivotal in minimizing the number of nodes we need to visit.

Insertions

The way your table's data is arranged in a B+tree depends on the key you choose. This means your choice of `PRIMARY KEY` will impact the layout on disk of all of the data in the table, and in turn performance. Choose your `PRIMARY KEY` wisely!

Two common choices for a primary key are:

* An integer sequence (such as `BIGINT UNSIGNED AUTO_INCREMENT`)

* A UUID, of which there are many versions.

Let's first consider the consequences of using a UUIDv4 primary key. A UUIDv4 is a mostly-random 128 bit integer.

We can simulate this by inserting a bunch of random integers into our B+tree visualization. On each insertion, all of the visited nodes will be highlighted green. You can also control the percentage of keys to keep in the existing node when a split occurs. Give it a try by clicking the Add random button several times. What do you notice?

A few observations:

1. The nodes visited for an insert are unpredictable ahead of time.

2. The destination leaf node for an insert is unpredictable.

3. The **values** in the leaves are not in order.

Issues 1 and 2 are problematic because over the course of many insertions we'll have to visit many of the nodes (pages) in the tree. This excessive reading and writing leads to poor performance. Issue 3 is problematic if we intend to ever search for or view our data in the order it was inserted.

The same problem can arise (albeit in a less extreme way) with some other UUIDs as well. For example, UUID v3 and v5 are both generated via hashing, and therefore will not be sequential and have similar behavior to inserting randomly. Alternatively, UUIDv7 actually does a good job of overcoming some of these challenges.

Let's consider using a sequential `BIGINT UNSIGNED AUTO_INCREMENT` as our primary key instead. Try inserting sequential values into the B+tree instead:

This mitigates all of the aforementioned problems:

1. We always follow the right-most path when inserting new values.

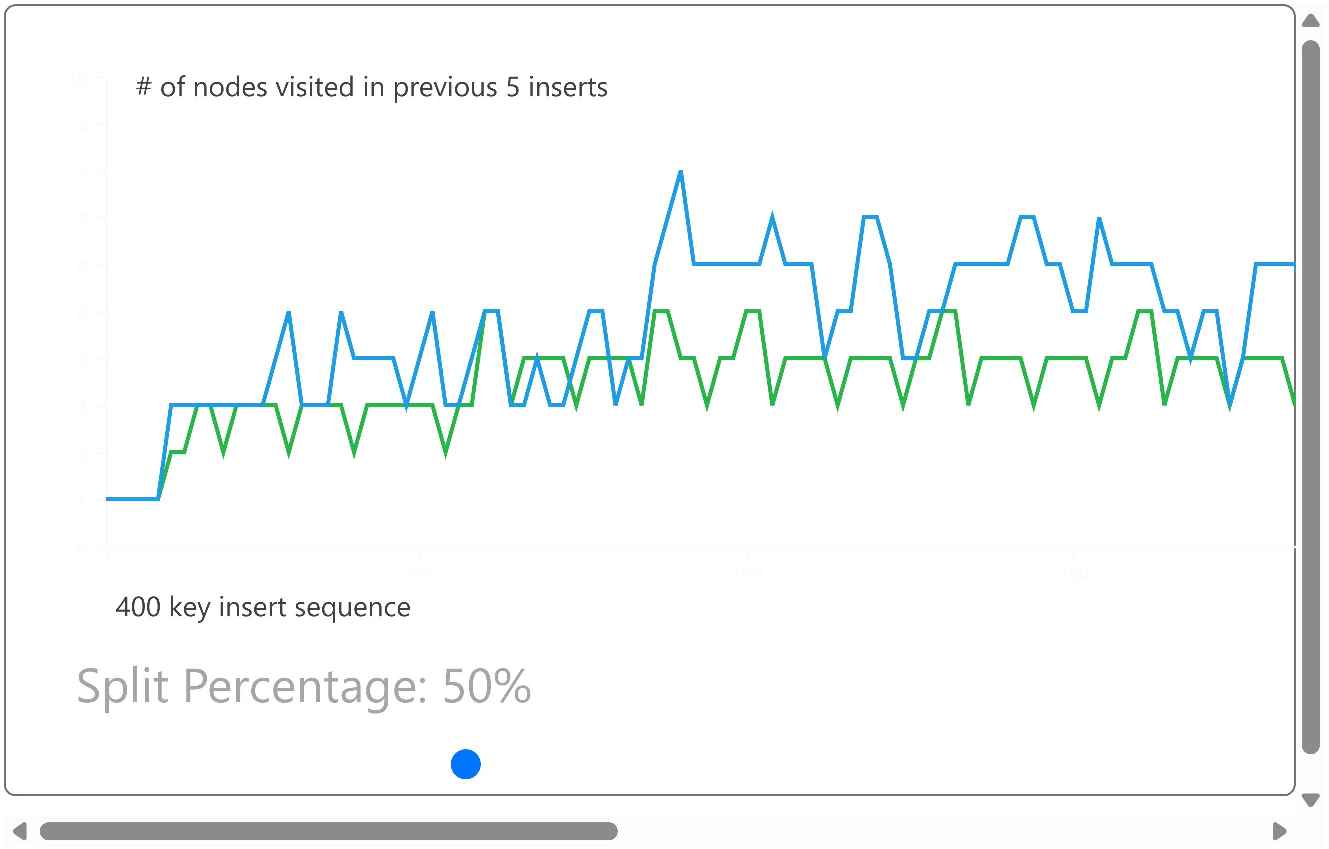
2. Leaves only get added on the right side of the tree.

3. At the leaf level, data is in sorted order based on when it was inserted.

Because of 1 and 2, many insertions happening in sequence will revisit the same path of pages, leading to fewer I/O requests when inserting a lot of key/value pairs.

The bar chart below shows the number of unique nodes visited for the previous 5 inserts on the two B+trees above. Assuming trees of the same depth, you should see random one being slightly higher, meaning worse performance.

If you are curious about the effect of the split percentage on sequential vs random insert patterns, check out the interactive visualization below. Use the slider to set the split percentage. The line graph will update to show how many nodes needed to be visited for the prior 5 at various points in a 400-key insertion sequence. Notice that in most cases, the sequential inserts require much fewer node visits than random inserts, and are also more predictable.

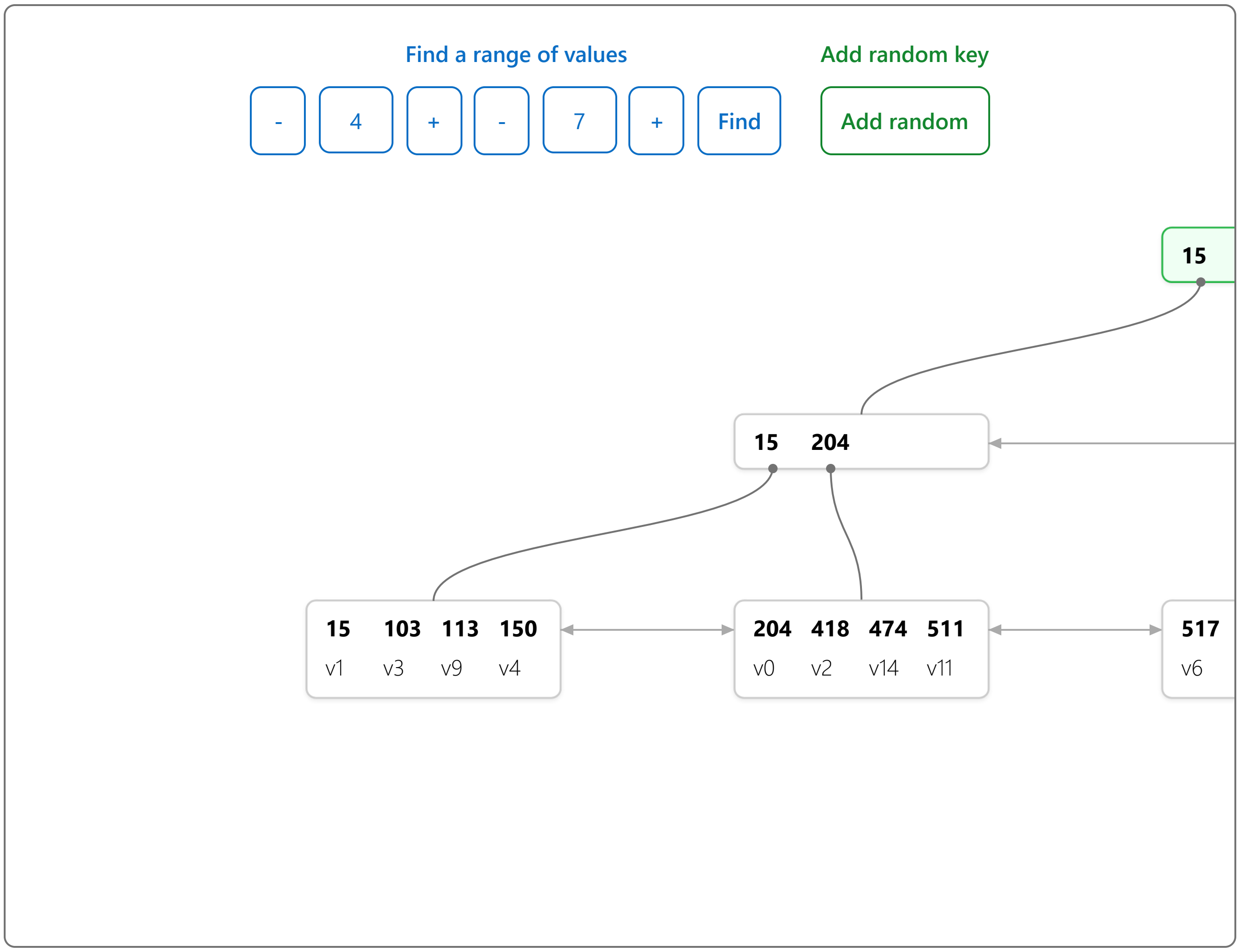


Data order

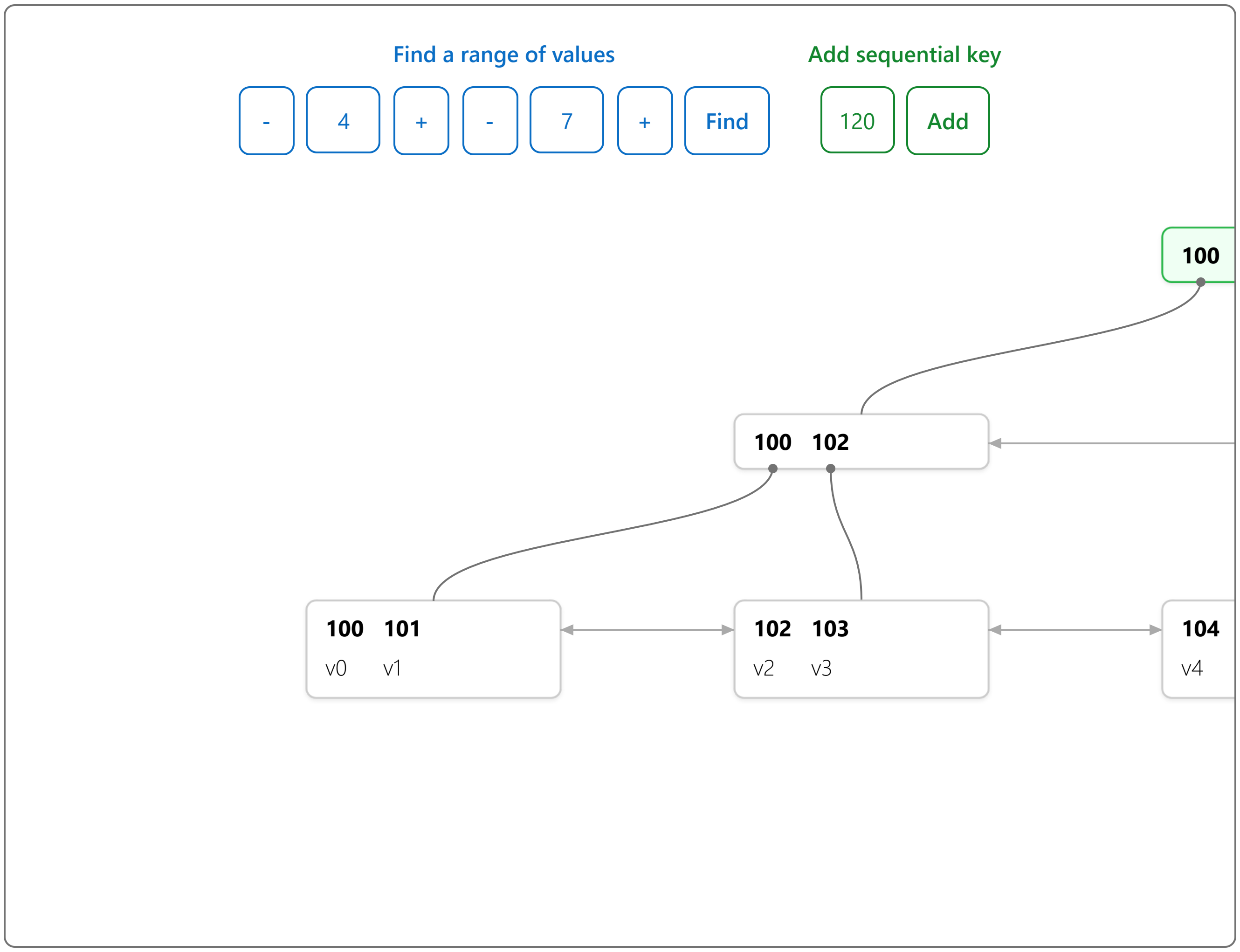
It's common to search for data from a database in time-sequenced order. Consider viewing the timeline on X, or a chat history in Slack. We typically want to see the posts and chat messages in time (or reverse-time) sequences. This means we'll often read chunks of database that are "near" each other in time. These queries take the form:

```
SELECT username, message_text, ...
FROM post
  WHERE sent > $START_DATETIME
  AND sent < $END_DATETIME
  ORDER BY sent DESC;
```

Consider what this would be like if we have UUIDv4s for our primary key. In the B+tree below, a bunch of random keys and corresponding values have been inserted into the table. Try finding ranges of values. What do you see?



Notice that the value sequences are spread out across many non-sequential leaf nodes. On the other hand, consider finding sequentially inserted values instead.



In such cases, all pages with the search results will be next to each other. It's even possible to search for several rows, and all of them will be next to each other in a single page. For this variety of query pattern, we can mitigate the number of pages that need to be read using a sequential primary key.

Primary key size

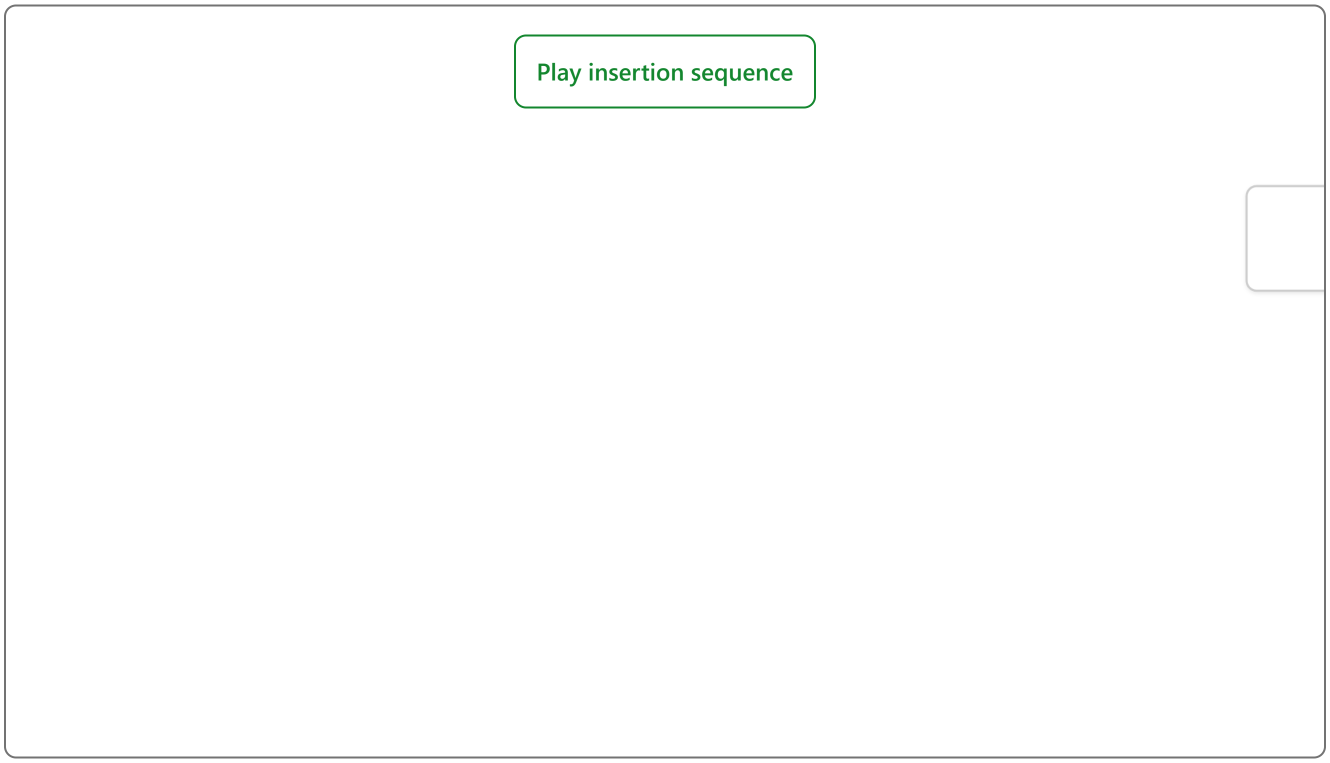
Another important consideration is key size. We always want our primary keys to be:

1. Big enough to never face exhaustion

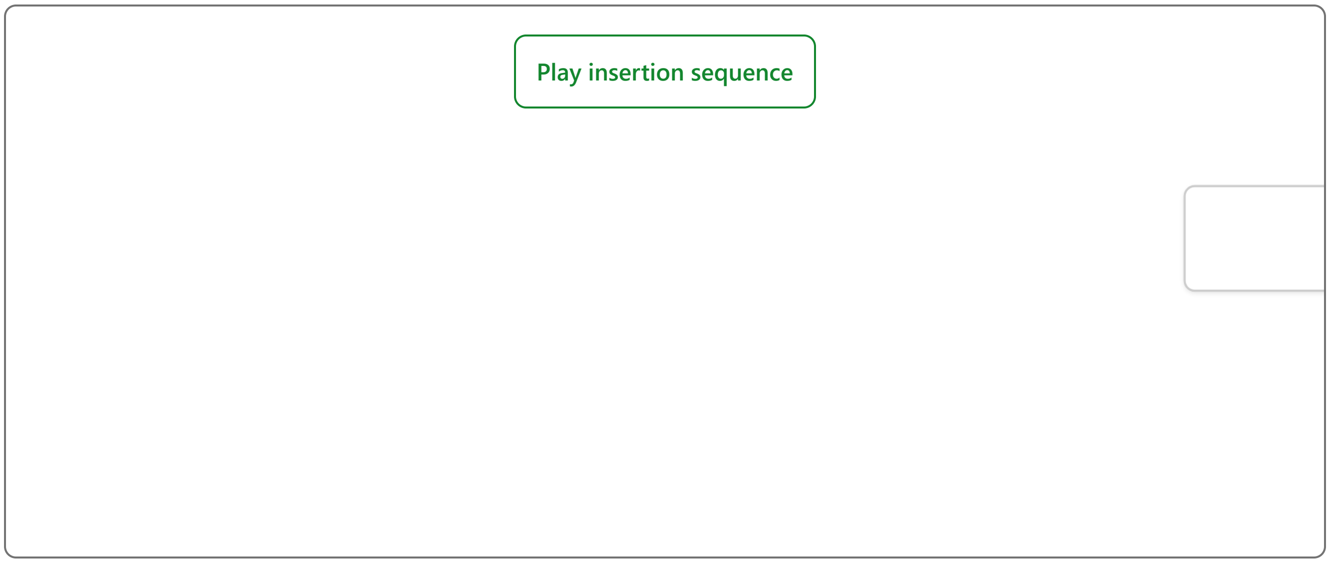
2. Small enough to not use excessive storage

For integer sequences, we can sometimes get away with a MEDIUMINT (16 million unique values) or INT (4 billion unique values) for smaller tables. For big tables, we often jump to BIGINT to be safe (18 sextillion possible values). BIGINTs are 64 bits (8 bytes). UUIDs are typically 128 bits (16 bytes), twice the size of even the largest integer type in MySQL. Since B+tree nodes are a fixed size, a BIGINT will allow us to fit more keys per-node than UUIDs. This results in shallower trees and faster lookups.

Consider a case where each tree node is only 100 bytes, child pointers are 8 bytes, and values are 8 bytes. We could fit 4 UUIDs (plus 4 child pointers) in each node. Hit the play insertion sequence button below to see the inserts.



If we had used a BIGINT instead, we could fit 6 keys (and corresponding child pointers) in each node instead. This would lead to a shallower tree, better for performance.

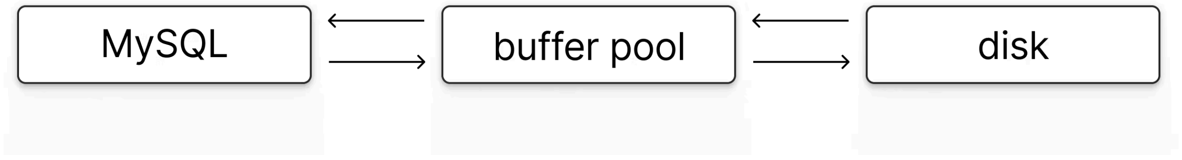


Pages and InnoDB

Recall that one of the big benefits of a B+tree is the fact that we can set the node size to whatever we want. In InnoDB, the B+tree nodes are typically set to 16k, the size of an **InnoDB page**.

When fulfilling a query (and therefore traversing B+trees), InnoDB does not read individual rows and columns from disk. Whenever it needs to access a piece of data, it loads the entire associated page from disk.

InnoDB has some tricks up its sleeve to mitigate this, the main one being the **buffer pool**. The buffer pool is an in-memory cache for InnoDB pages, sitting between the pages on-disk and MySQL query execution. When MySQL needs to read a page, it first checks if it's already in the buffer pool. If so, it reads it from there, skipping the disk I/O operation. If not, it finds the page on-disk, adds it to the buffer pool, and then continues query execution.



The buffer pool *drastically* helps query performance. Without it, we'd end up doing significantly more disk I/O operations to handle a query workload. Even with the buffer pool, minimizing the number of pages that need to be visited helps performance (1) because there's still a (small) cost to looking up a page in the buffer pool, and (2) it helps reduce the number of buffer pool loads and evictions that need to take place.

Other situations

Here, we mostly focused on comparing a sequential key to a random / UUID key. However, the principles shown here are useful to keep in mind no matter what kind of primary or secondary key you are considering.

For example, you may also consider using a `user.created_at` timestamp as a key for an index. This will have similar properties to a sequential integer. Insertions will generally always go to the right-most path, unless legacy data is being inserted.

Conversely, something like a `user.email_address` string will have more similar characteristics to a random key. Users won't be creating accounts in email-alphabetical order, so insertions will happen all over the place in the B+tree.

Conclusion

This is already a long blog post, and yet, much more could be said about B+trees, indexes, and primary key choice in MySQL. On the surface it may seem simple, but there's an incredible amount of nuance to consider if you want to squeeze every ounce of performance out of your database. If you'd like to experiment further, you can visit the dedicated interactive B+tree website. If you want a regular B-tree, go here instead. I hope you learned a thing or two about indexes!

Special thanks to Sam Rose for early review.