**Week 3: B-Trees**

**Overview**

Databases are used to store and query subsets of data.  Most databases are designed to handle datasets larger than the available memory.  While it may be possible to do a naive linear searches through data when held in memory, accessing data on disk is 2-3 orders of magnitude slower.

The tradeoffs chosen by database designers determine the appropriate use cases for a given database.  Relational databases are often optimized for online transaction processing (OLTP) use cases.  They are designed to serve a large number of concurrent users with each user accessing a small number of records at a time. These databases provide transaction mechanisms for ensuring that updates to related records are performed atomically so that data are not left in an inconsistent state.  These databases store data in a manner for which it's easy to directly access individual records.  Online analytics processing (OLAP) databases are designed to handle a smaller number of users running queries that touch large portions of data in each query.  These databases often designed to store runs of data in columns to make it efficient to access large batches.

The data structures used by databases vary from most of the data structures seen in a Data Structures or Algorithms class in that they are designed to be stored on disk.  Databases commonly employ B-trees as search trees. Unlike the widely-studied binary search trees, each node in a B-tree stores multiple keys and has multiple children.  The result is a tree that is relatively short but wide, reducing the number of nodes that need to be read from disk to find a record of interest.

When implementing in-memory data structures, we can rely on the programming language's abstractions.  We can represent each node as a class, use pointers (references) to connect nodes to their children, and let the operating system or run-time manage where the nodes are located in memory.

On-disk data structures require us to make decisions about many of these issues.  Disks transfer data in pages (usually 4 kb), even if you only need a single byte.  As a result, on-disk formats and data structures are designed around accessing 4kb chunks of data in each operation to reduce the number disk accesses needed.  For B-trees, each node is usually designed to take up exactly 4kb of space.

Secondly, we can't use pointers.  Instead, we usually treat the on-disk format as a log.  It will have a 4 kb header page.  The header will store the offset (in units of pages) of the root node of the tree.  For each child, nodes will have a 4-byte integer that stores the offset of that child in the on-disk format (instead of a direct pointer).

Third, the programmer has to decide how to organize the layout of each page, how much storage to allocate to each datum, and manually handle converting between programming language types (ints, strings, etc.) and byte strings. For example, the [Let's Build a Database tutorial](https://cstack.github.io/db_tutorial/) uses a B+-tree structure with the following layouts for internal and leaf nodes, respectively:A screenshot of a computer

AI-generated content may be incorrect.

As such, there are often a few layers of indirection in the implementation of the on-disk B-trees.

**Reflection Questions**

* How is a B-tree organized?
* How and why is it different from a binary search tree?
* How do you perform various operations such as searching, inserting, or updating data?
* If a node has 32 children, how many keys does it store?
* What is the big-O complexity of various operations?
* What is the formal difference between a B-tree and a B+-tree (even though the terms are used interchangeably in practice)?
* Consider B-tree and B+-trees each storing 1024 items with 32 children per node. Assume the trees are balance. How many nodes are in each tree? What are the depths of the trees?
* Why aren't rotations (like those used in AVL and red-black trees) used to balance B-trees?
* If you want to search for an item in a B-tree, how do you search the tree, loading nodes from disk as you go?
* If you update an existing key-value pair, which nodes do you have to update?
* How do you add a key or key-value pair to an existing node? What nodes will be updated if you add a key-value pair, with and without splitting?