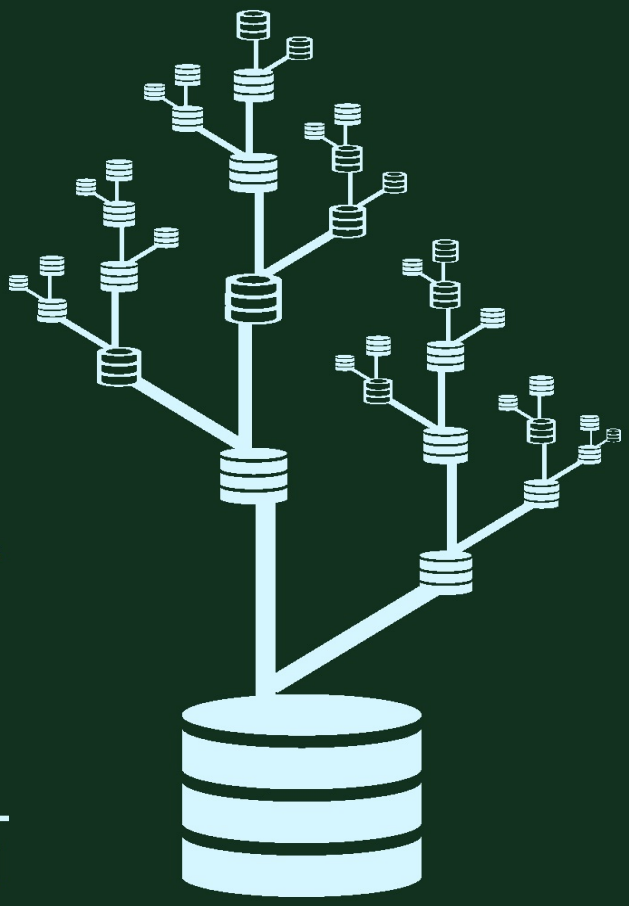
Learn by doing:
From B+Tree to SQL in 3000 lines of code

Build Your Own Database From Scratch in Go

2nd EDITION

B+Tree
Durable KV
Transaction
Relational DB
Query Language



James Smith

build-your-own.org

Build Your Own Database From Scratch in Go

From B+Tree To SQL

James Smith

2024-06-04

Build Your Own Database From Scratch in Go

00. Introduction

Master fundamentals by building your own DB

What to learn?

Complex systems like databases are built on a few simple principles.

1. **Atomicity & durability**. A DB is more than files!
    - Persist data with `fsync`.
    - Crash recovery.
2. **KV store** based on **B-tree**.
    - Disk-based data structures.
    - Space management with a free list.
3. **Relational DB** on top of KV.
    - How tables and indexes are mapped to low-level B-trees.
    - SQL-like **query language**; parser & interpreter.
4. **Concurrency control** for transactions.

Code a database in 3000 LoC, incrementally

It’s amazing that an interesting and broad topic can be captured in 3000 LoC. You may have experience with larger projects, but not all experience is equal.

LoC	Step
366	B+tree data structure.
601	Append-only KV.
731	Practical KV with a free list.
1107	Tables on KV.

LoC	Step
1294	Range queries.
1438	Secondary indexes.
1461	Transactional interfaces.
1702	Concurrency control.
2795	SQL-like query language.

Learn by doing: principles instead of jargon

Database literature is full of confusing, overloaded jargon with no consistent meaning. It’s easy to get lost when reading about it. On the other hand, Feymann once said, “what I can’t build, I don’t understand”. Can you build a database by reading about databases? Test your understanding!

While there is a lot to learn, not all knowledge is equally important, **it takes only a few principles to build a DB**, so anyone can try.

Topic 1: durability and atomicity

More than a data format

Smartphones use SQLite (a file-based DB) heavily. Why store data in SQLite instead of some other format, say JSON? Because you risk data loss if it crashes during an update. The file can end up half-written, truncated, or even missing.

There are techniques to fix this, and they lead to databases.

Durability and atomicity with `fsync`

Atomicity means that data is either updated or not, not in between. Durability means that data is guaranteed to exist after a certain point. They are not separate concerns, because we must achieved both.

The first thing to learn is the `fsync` syscall. A file write doesn't reach disk synchronously, there are multiple levels of buffering (OS page cache and on-device RAM). `fsync` flushes pending data and waits until it's done. This makes writes durable, but what about atomicity?

Topic 2: indexing data structures

Control latency and cost with indexes

A DB turns a query into a result without the user knowing how. But the result is not the only concern, *latency* and *cost* (memory, IO, computation) are also relevant, hence the distinction between analytical (OLAP) and transactional (OLTP).

- OLAP can involve large amounts of data, with aggregation or join operations. Indexing can be limited or non-existent.
- OLTP touches small amounts of data using indexes. Low latency & cost.

The word “transactional” is not about DB transactions, it’s just a funny jargon.

In-memory data structures vs. on-disk data structures

There are extra challenges when putting an indexing data structure on disk. (See my book “Build Your Own Redis” for a much easier in-memory DB).

One of the problems is updating disk data *in-place*, because you have to deal with corrupted states after a crash. Disks are not just slower RAM.

The R in RAM stands for “random”, which is another problem for disk-based data because random access is much slower than sequential access, even on SSDs. So data structures like binary trees are not viable while B-

trees and LSM-trees are OK. *Concurrent* access to data structures is also a topic.

Topic 3: Relational DB on KV

Two layers of DB interfaces

SQL is almost a synonym for database. But SQL is just a user interface, it's not fundamental to a DB. What's important is the **functionalities** underneath.

Another much simpler interface is key-value (KV). You can get, set, and delete a single key, and most importantly, list a range of keys in sorted order. KV is simpler than SQL because it's one layer lower. Relational DBs are built on top of KV-like interfaces called *storage engines*.

Query languages: parsers and interpreters

The last step is easy, despite the larger LoC. Both the parser and the interpreter are coded with nothing but *recursion*! The lesson can be applied to almost any computer language, or creating your own programming language or DSL (See my book "From Source Code To Machine Code" for more challenges).

Build Your Own X book series

X includes Redis, web server and compiler. Read the web version on the website.

https://build-your-own.org

01. From Files To Databases

Let’s start with files, and examine the challenges we face.

1.1 Updating files in-place

Let's say you need to save some data to disk; this is a typical way to do it:

```
func SaveData1(path string, data []byte) error {
    fp, err := os.OpenFile(path, os.O_WRONLY|os.O_CREATE|os.O_TRUNC, 0664)
    if err != nil {
        return err
    }
    defer fp.Close()

    _, err = fp.Write(data)
    if err != nil {
        return err
    }
    return fp.Sync() // fsync
}
```

This code creates the file if it does not exist, or truncates the existing one before writing the content. And most importantly, the data is not persistent unless you call `fsync` (`fp.Sync()` in Go).

It has some serious limitations:

1. It updates the content as a whole; only usable for tiny data. This is why you don't use Excel as a database.
2. If you need to update the old file, you must read and modify it in memory, then overwrite the old file. What if the app crashes while overwriting the old file?
3. If the app needs concurrent access to the data, how do you prevent readers from getting mixed data and writers from conflicting operations? That's why most databases are client-server, you need a

server to coordinate concurrent clients. (Concurrency is more complicated without a server, see SQLite).

1.2 Atomic renaming

Replacing data atomically by renaming files

Many problems are solved by not updating data *in-place*. You can write a new file and delete the old file.

Not touching the old file data means:

1. If the update is interrupted, you can recover from the old file since it remains intact.
2. Concurrent readers won't get half written data.

The problem is how readers will find the new file. A common pattern is to rename the new file to the old file path.

```
func SaveData2(path string, data []byte) error {
    tmp := fmt.Sprintf("%s.tmp.%d", path, randomInt())
    fp, err := os.OpenFile(tmp, os.O_WRONLY|os.O_CREATE|os.O_EXCL, 0664)
    if err != nil {
        return err
    }
    defer func() {
        fp.Close()
        if err != nil {
            os.Remove(tmp)
        }
    }()

    _, err = fp.Write(data)
    if err != nil {
        return err
    }
    err = fp.Sync() // fsync
```

```
    if err != nil {
        return err
    }

    return os.Rename(tmp, path)
}
```

Renaming a file to an existing one replaces it *atomically*; deleting the old file is not needed (and not correct).

Pay attention to the meaning of the jargon, whenever you see “X is atomic”, you should ask “X is atomic *with respect to* what?” In this case:

- Rename is atomic w.r.t. concurrent readers; a reader opens either the old or the new file.
- Rename is NOT atomic w.r.t. power loss; it’s not even durable. You need an extra `fsync` on the parent *directory*, which is discussed later.

Why does renaming work?

Filesystems keep a mapping from file names to file data, so replacing a file by renaming simply points the file name to the new data without touching the old data. That’s why atomic renaming is possible in filesystems. And the operation cost is constant regardless of the data size.

On Linux, the replaced old file may still exist if it’s still being opened by a reader; it’s just not accessible from a file name. Readers can safely work on whatever version of the data it got, while writer won’t be blocked by readers. However, there must be a way to prevent concurrent writers. The level of concurrency is multi-reader-single-writer, which is what we will implement.

1.3 Append-only logs

Safe incremental updates with logs

One way to do incremental updates is to just append the updates to a file. This is called a “log” because it’s append-only. It’s safer than in-place updates because no data is overwritten; you can always recover the old data after a crash.

The reader must consider all log entries when using the log. For example, here is a log-based KV with 4 entries:

```
     0         1         2         3
| set a=1 | set b=2 | set a=3 | del b |
```

The final state is **`a=3`**.

Logs are an essential component of many databases. However, a log is only a description of each update, which means:

- It’s not an indexing data structure; readers must read all entries.
- It has no way to reclaim space from deleted data.

So logs alone are not enough to build a DB, they must be combined with other indexing data structures.

Atomic log updates with checksums

While a log won’t corrupt old data, you still have to deal with the last entry if it gets corrupted after a crash. Many possibilities:

1. The last append simply does not happen; the log is still good.
2. The last entry is half written.
3. The size of the log is increased but the last entry is not there.

The way to deal with these cases is to add a checksum to each log entry. If the checksum is wrong, the update did not happen, making log updates atomic (w.r.t. both readers and durability).

This scenario is about incomplete writes (*torn writes* in DB jargon) that occur before a successful `fsync`. Checksums can also detect other forms of corruption after `fsync`, but that's not something a DB can recover from.

1.4 `fsync` gotchas

After renaming files or creating new files, you must call `fsync` on the parent directory. A directory is a mapping from file names to files, and like file data, it's not durable unless you use `fsync`. See this example of `fsync` on the directory.

Another issue with `fsync` is error handling. If `fsync` fails, the DB update fails, but what if you read the file afterwards? You may get the new data even if `fsync` failed (because of the OS page cache)! This behavior is filesystem dependent.

1.5 Summary of database challenges

What we have learned:

1. Problems with in-place updates.
    - Avoid in-place updates by renaming files.
    - Avoid in-place updates using logs.
2. Append-only logs.
    - Incremental updates.
    - Not a full solution; no indexing and space reuse.
3. `fsync` usage.

What remains a question:

1. Indexing data structures and how to update them.
2. Reuse space from append-only files.
3. Combining a log with an indexing data structure.
4. Concurrency.

02. Indexing Data Structures

2.1 Types of queries

Most SQL queries can be broken down into 3 types:

1. Scan the whole data set. (No index is used).
2. Point query: Query the index by a specific key.
3. Range query: Query the index by a range. (The index is sorted).

There are ways to make scanning fast, such as column-based storage. But a scan is $O(N)$ no matter how fast it is; our focus is on queries that can be served in $O(\log N)$ using data structures.

A range query consists of 2 phases:

1. Seek: find the starting key.
2. Iterate: find the previous/next key in sorted order.

A point query is just seek without iterate; a sorting data structure is all we need.

2.2 Hashtables

Hashtables are viable if you only consider point queries (get, set, del), so we will not bother with them because of the lack of ordering.

However, coding a hashtable, even an in-memory one, is still a valuable exercise. It’s far easier than the B-tree we’ll code later, though some challenges remain:

- How to grow a hashtable? Keys must be moved to a larger hashtable when the load factor is too high. Moving everything at once is prohibitively $O(N)$. Rehashing must be done progressively, even for in-memory apps like Redis.
- Other things mentioned before: in-place updates, space reuse, and etc.

2.3 Sorted arrays

Ruling out hashtables, let's start with the simplest sorting data structure: the sorted array. You can binary search on it in $O(\log N)$. For variable-length data such as strings (KV), use an array of pointers (offsets) to do binary searches.

Updating a sorted array is $O(N)$, either in-place or not. So it's not practical, but it can be extended to other updatable data structures.

One way to reduce the update cost is to split the array into several smaller non-overlapping arrays — nested sorted arrays. This extension leads to B+tree (multi-level n-ary tree), with the additional challenge of maintaining these small arrays (tree nodes).

Another form of "updatable array" is the log-structured merge tree (LSM-tree). Updates are first buffered in a smaller array (or other sorting data structures), then merged into the main array when it becomes too large. The update cost is amortized by propagating smaller arrays into larger arrays.

2.4 B-tree

A B-tree is a balanced n-ary tree, comparable to balanced binary trees. Each node stores variable number of keys (and branches) up to n and $n > 2$.

Reducing random access with shorter trees

A disk can only perform a limited number of IOs per second (IOPS), which is the limiting factor for tree lookups. Each level of the tree is a disk read in a lookup, and n-ary trees are shorter than binary trees for the same number of keys ($\log_n N$ vs. $\log_2 N$), thus n-ary trees are used for fewer disk reads per lookup.

How is the n chosen? There is a trade-off:

- Larger n means fewer disk reads per lookup (better latency and throughput).
- Larger n means larger nodes, which are slower to update (discussed later).

IO in the unit of pages

While you can read any number of bytes at any offset from a file, disks do not work that way. The basic unit of disk IO is not bytes, but sectors, which are 512-byte contiguous blocks on old HDDs.

However, disk sectors are not an application's concern because regular file IOs do not interact directly with the disk. The OS caches/buffers disk

reads/writes in the *page cache*, which consists of 4K-byte memory blocks called *pages*.

In any way, there is a minimum unit of IO. DBs can also define their own unit of IO (also called a page), which can be larger than an OS page.

The minimum IO unit implies that tree nodes should be allocated in multiples of the unit; a half used unit is half wasted IO. Another reason against small *n*!

The B+tree variant

In the context of databases, B-tree means a variant of B-tree called B+tree. In a B+tree, internal nodes do not store values, values exist only in leaf nodes. This leads to shorter tree because internal nodes have more space for branches.

B+tree as an in-memory data structure also makes sense because the minimum IO unit between RAM and CPU caches is 64 bytes (cache line). The performance benefit is not as great as on disk because not much can fit in 64 bytes.

Data structure space overhead

Another reason why binary trees are impractical is the number of pointers; each key has at least 1 incoming pointer from the parent node, whereas in a B+tree, multiple keys in a leaf node share 1 incoming pointer.

Keys in a leaf node can also be packed in a compact format or compressed to further reduce the space.

2.5 Log-structured storage

Update by merge: amortize cost

The most common example of log-structured storage is log-structure merge tree (LSM-tree). Its main idea is neither log nor tree; it's "merge" instead!

Let's start with 2 files: a small file holding the recent updates, and a large file holding the rest of the data. Updates go to the small file first, but it cannot grow forever; it will be merged into the large file when it reaches a threshold.

```
writes => | new updates | => | accumulated data |
              file 1               file 2
```

Merging 2 sorted files results in a newer, larger file that replaces the old large file and shrinks the small file.

Merging is $O(N)$, but can be done concurrently with readers and writers.

Reduce write amplification with multiple levels

Buffering updates is better than rewriting the whole dataset every time. What if we extend this scheme to multiple levels?

```
                |level 1|
                   ||
                   \/
          |------level 2------|
                   ||
                   \/
|-----------------level 3-----------------|
```

In the 2-level scheme, the large file is rewritten every time the small file reaches a threshold, the excess disk write is called *write amplification*, and it gets worse as the large file gets larger. If we use more levels, we can keep the 2nd level small by merging it into the 3rd level, similar to how we keep the 1st level small.

Intuitively, levels grow exponentially, and the power of two growth (merging similarly sized levels) results in the least write amplification. But there is a trade-off between write amplification and the number of levels (query performance).

LSM-tree indexes

Each level contains indexing data structures, which could simply be a sorted array, since levels are never updated (except for the 1st level). But binary search is not much better than binary tree in terms of random access, so a sensible choice is to use B-tree inside a level, that's the "tree" part of LSM-tree. Anyway, data structures are much simpler because of the lack of updates.

To better understand the idea of "merge", you can try to apply it to hashtables, a.k.a. log-structured hashtables.

LSM-tree queries

Keys can be in any levels, so to query an LSM-tree, the results from each level are combined (n-way merge for range queries).

For point queries, Bloom filters can be used as an optimization to reduce the number of searched levels.

Since levels are never updated, there can be old versions of keys in older levels, and deleted keys are marked with a special flag in newer levels (called tombstones). Thus, newer levels have priority in queries.

The merge process naturally reclaims space from old or deleted keys. Thus, it's also called *compaction*.

Real-world LSM-tree: SSTable, MemTable and log

These are jargons about LSM-tree implementation details. You don't need to know them to build one from principles, but they do solve some real problems.

Levels are split into multiple non-overlapping files called SSTables, rather than one large file, so that merging can be done gradually. This reduces the free space requirement when merging large levels, and the merging process is spread out over time.

The 1st level is updated directly, a log becomes a viable choice because the 1st level is bounded in size. This is the "log" part of the LSM-tree, an example of combining a log with other indexing data structures.

But even if the log is small, a proper indexing data structure is still needed. The log data is *duplicated* in an in-memory index called MemTable, which can be a B-tree, skiplist, or whatever. It's a small, bounded amount of in-memory data, and has the added benefit of accelerating the read-the-recent-updates scenario.

2.6 Summary of indexing data structures

There are 2 options: B+tree and LSM-tree.

LSM-tree solves many of the challenges from the last chapter, such as how to update disk-based data structures and resue space. While these challenges remain for B+tree, which will be explored later.

03. B-Tree & Crash Recovery

3.1 B-tree as a balanced n-ary tree

Height-balanced tree

Many practical binary trees, such as the AVL tree or the RB tree, are called *height-balanced trees*, meaning that the height of the tree (from root to leaves) is limited to $O(\log N)$, so a lookup is $O(\log N)$.

A B-tree is also height-balanced; the height is the same for all leaf nodes.

Generalizing binary trees

n-ary trees can be generalized from binary trees (and vice versa). An example is the 2-3-4 tree, which is a B-tree where each node can have either 2, 3, or 4 children. The 2-3-4 tree is equivalent to the RB tree. However, we won't go into the details because they are not necessary for understanding B-trees.

Visualizing a 2-level B+tree of a sorted sequence [1, 2, 3, 4, 6, 9, 11, 12].

```
     [1,   4,   9]
     /     |     \
    v      v      v
[1, 2, 3] [4, 6] [9, 11, 12]
```

In a B+tree, only leaf nodes contain value, keys are duplicated in internal nodes to indicate the key range of the subtree. In this example, node [1, 4, 9] indicates that its 3 subtrees are within intervals [1, 4), [4, 9), and [9, +∞). However, only 2 keys are needed for 3 intervals, so the first key (1) can be omitted and the 3 intervals become (-∞, 4), [4, 9), and (9, +∞).

3.2 B-tree as nest arrays

Two-level nested arrays

Without knowing the details of the RB tree or the 2-3-4 tree, the B-tree can be understood from sorted arrays.

The problem with sorted arrays is the $O(N)$ update. If we split the array into m smaller non-overlapping ones, the update becomes $O(N/m)$. But we have to find out which small array to update/query first. So we need another sorted array of references to smaller arrays, that's the internal nodes in a B+tree.

```
[[1,2,3], [4,6], [9,11,12]]
```

The lookup cost is still $O(\log N)$ with 2 binary searches. If we choose m as $\sqrt{N}$, update become $O(\sqrt{N})$, that's as good as 2-level sorted arrays can be.

Multiple levels of nested arrays

$O(\sqrt{N})$ is unacceptable for databases, but if we add more levels by splitting arrays even more, the cost is further reduced.

Let's say we keep splitting levels until all arrays are no larger than a constant s, we end up with $\log(N/s)$ levels, and the lookup cost is $O(\log(N/s) + \log(s))$, which is still $O(\log N)$.

For insertion and deletion, after finding the leaf node, updating the leaf node is constant $O(s)$ most of the time. The remaining problem is to maintain the invariants that nodes are not larger than s and are not empty.

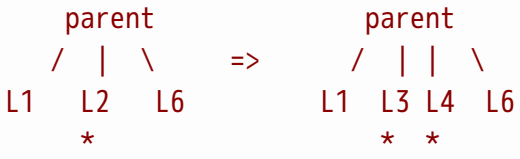
3.3 Maintaining a B+tree

3 invariants to preserve when updating a B+tree:

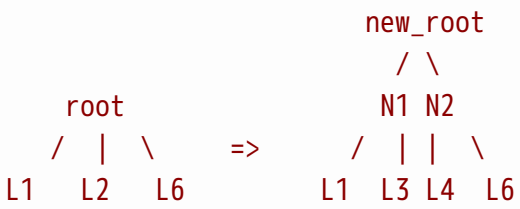
1. Same height for all leaf nodes.
2. Node size is bounded by a constant.
3. Node is not empty.

Growing a B-tree by splitting nodes

The 2nd invariant is violated by inserting into a leaf node, which is restored by splitting the node into smaller ones.



After splitting a leaf node, its parent node gets a new branch, which may also exceed the size limit, so it may need to be split as well. Node splitting can propagate to the root node, increasing the height by 1.



This preserves the 1st invariant, since all leaves gain height by 1 simultaneously.

Shrinking a B-tree by merging nodes

Deleting may result in empty nodes. The 3rd invariant is restored by merging empty nodes into a sibling node. Merging is the opposite of splitting. It can also propagate to the root node, so the tree height can decrease.

When coding a B-tree, merging can be done earlier to reduce wasted space: you can merge a non-empty node when its size reaches a lower bound.

3.4 B-Tree on disk

You can already code an in-memory B-tree using these principles. But B-tree on disk requires extra considerations.

Block-based allocation

One missing detail is how to limit node size. For in-memory B+tree, you can limit the maximum number of keys in a node, the node size in bytes is not a concern, because you can allocate as many bytes as needed.

For disk-based data structures, there are no `malloc/free` or garbage collectors to rely on; space allocation and reuse is entirely up to us.

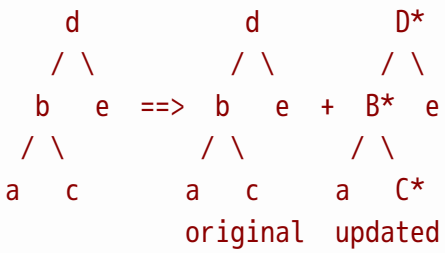
Space reuse can be done with a *free list* if all allocations are of the *same size*, which we'll implement later. For now, all B-tree nodes are the same size.

Copy-on-write B-tree for safe updates

We've seen 3 crash-resistant ways to update disk data: renaming files, logs, LSM-trees. The lesson is **not to destroy any old data during an update**. This idea can be applied to trees: make a copy of the node and modify the copy instead.

Insertion or deletion starts at a leaf node; after making a copy with the modification, its parent node must be updated to point to the new node, which is also done on its copy. The copying propagates to the root node, resulting in a new tree root.

- The original tree remains intact and is accessible from the old root.
- The new root, with the updated copies all the way to the leaf, shares all other nodes with the original tree.



This is a visualization of updating the leaf c. The copied nodes are in uppercase (D, B, C), while the shared subtrees are in lowercase (a, e).

This is called a *copy-on-write* data structure. It's also described as *immutable*, *append-only* (not literally), or *persistent* (not related to durability). Be aware that database jargon does not have consistent meanings.

2 more problems remain for the copy-on-write B-tree:

1. How to find the tree root, as it changes after each update? The crash safety problem is reduced to a single pointer update, which we'll solve later.
2. How to reuse nodes from old versions? That's the job of a free list.

Copy-on-write B-tree advantages

One advantage of keeping old versions around is that we got *snapshot isolation* for free. A transaction starts with a version of the tree, and won't see changes from other versions.

And crash recovery is effortless; just use the last old version.

Another one is that it fits the multi-reader-single-writer concurrency model, and readers do not block the writer. We'll explore these later.

Alternative: In-place update with double-write

While crash recovery is obvious in copy-on-write data structures, they can be undesirable due to the high write amplification. Each update copies the whole path ($O(\log N)$), while most in-place updates touch only 1 leaf node.

It's possible to do in-place updates with crash recovery without copy-on-write:

1. Save a copy of the entire updated nodes somewhere. This is like copy-on-write, but without copying the parent node.
2. `fsync` the saved copies. (Can respond to the client at this point.)
3. Actually update the data structure in-place.
4. `fsync` the updates.

After a crash, the data structure may be half updated, but we don't really know. What we do is blindly apply the saved copies, so that the data structure ends with the updated state, regardless of the current state.

```
| a=1 b=2 |
    ||  1. Save a copy of the entire updated nodes.
    \/
| a=1 b=2 |   +   | a=2 b=4 |
   data           updated copy
    ||  2. fsync the saved copies.
    \/
| a=1 b=2 |   +   | a=2 b=4 |
   data           updated copy (fsync'ed)
    ||  3. Update the data structure in-place. But we crashed here!
    \/
| ??????? |   +   | a=2 b=4 |
   data (bad)     updated copy (good)
```

```
    ||  Recovery: apply the saved copy.
    \/
| a=2 b=4 |   +   | a=2 b=4 |
   data (new)     useless now
```

The saved updated copies are called double-write in MySQL jargon. But what if the double-write is corrupted? It's handled the same way as logs: checksum.

- If the checksum detects a bad double-write, ignore it. It's before the 1st `fsync`, so the main data is in a good and old state.
- If the double-write is good, applying it will always yield good main data.

Some DBs actually store the double-writes in logs, called physical logging. There are 2 kinds of logging: *logical* and *physical*. Logical logging describes high-level operations such as inserting a key, such operations can only be applied to the DB when it's in a good state, so only physical logging (low-level disk page updates) is useful for recovery.

The crash recovery principle

Let's compare double-write with copy-on-write:

- Double-write makes updates *idempotent*; the DB can retry the update by applying the saved copies since they are full nodes.
- Copy-on-write *atomically* switches everything to the new version.

They are based on different ideas:

- Double-write ensures enough information to produce the new version.
- Copy-on-write ensures that the old version is preserved.

What if we save the *original* nodes instead of the updated nodes with double-write? That's the 3rd way to recover from corruption, and it recovers to the old version like copy-on-write. We can combine the 3 ways into 1 idea: **there is enough information for either the old state or the new state at any point**.

Also, some copying is always required, so larger tree nodes are slower to update.

We'll use copy-on-write because it's simpler, but you can deviate here.

3.5 What we learned

B+tree principles:

- n-ary tree, node size is limited by a constant.
- Same height for all leaves.
- Split and merge for insertion and deletion.

Disk-based data structures:

- Copy-on-write data structures.
- Crash recovery with double-write.

We can start coding now. 3 steps to create a persistent KV based on B+tree:

1. Code the B+tree data structure.
2. Move the B+tree to disk.
3. Add a free list.

04. B+Tree Node and Insertion

4.1 Design B+tree nodes

What we will do

The first big step is just the B+tree data structures, other DB concerns will be covered in later chapters. We'll do it from the bottom up.

1. Design a node format that contains all the necessary bits.
2. Manipulate nodes in a copy-on-write fashion (insert and delete keys).
3. Split and merge nodes.
4. Tree insertion and deletion.

The node format

All B+tree nodes are the same size for later use of the free list. Although we won't deal with disk data at this point, a concrete node format is needed because it decides the node size in bytes and *when to split a node*.

A node includes:

1. A fixed-size header, which contains:
    - The type of node (leaf or internal).
    - The number of keys.
2. A list of pointers to child nodes for internal nodes.
3. A list of KV pairs.
4. A list of offsets to KVs, which can be used to binary search KVs.

```
| type | nkeys |  pointers  |   offsets  | key-values | unused |
|  2B  |   2B  | nkeys * 8B | nkeys * 2B |     ...    |        |
```

This is the format of each KV pair. Lengths followed by data.

```
| klen | vlen | key | val |
|  2B  |  2B  | ... | ... |
```

Simplifications and limits

Our goal is to learn the basics, not to create a real DB. So some simplifications are made.

The same format is used for both leaf nodes and internal nodes. This wastes some space: leaf nodes don't need pointers and internal nodes don't need values.

An internal node of n branches contains n keys, each key is duplicated from the minimum key of the corresponding subtree. However, only $n - 1$ keys are needed for n branches, as you'll see in other B-tree introductions. The extra key makes the visualization easier.

We'll set the node size to 4K, which is the typical OS page size. However, keys and values can be arbitrarily large, exceeding a single node. There should be a way to store large KVs outside of nodes, or to make the node size variable. This problem is solvable, but not fundamental. So we'll skip it by limiting the KV size so that they always fit inside a node.

```
const HEADER = 4

const BTREE_PAGE_SIZE = 4096
const BTREE_MAX_KEY_SIZE = 1000
const BTREE_MAX_VAL_SIZE = 3000

func init() {
    node1max := HEADER + 8 + 2 + 4 + BTREE_MAX_KEY_SIZE + BTREE_MAX_VAL_SIZE
    assert(node1max <= BTREE_PAGE_SIZE) // maximum KV
}
```

The key size limit also ensures that an internal node can always host 2 keys.

In-memory data types

In our code, a node is just a chunk of bytes interpreted by this format. Moving data from memory to disk is simpler without a serialization step.

```
type BNode []byte // can be dumped to the disk
```

Decouple data structure from IO

Space allocation/deallocation is required for both in-memory and on-disk data structures. We can abstract this away with callbacks, which is a boundary between the data structure and the rest of the DB.

```
type BTree struct {
    // pointer (a nonzero page number)
    root uint64
    // callbacks for managing on-disk pages
    get func(uint64) []byte // dereference a pointer
    new func([]byte) uint64 // allocate a new page
    del func(uint64)        // deallocate a page
}
```

For an on-disk B+tree, the database file is an array of pages (nodes) referenced by page numbers (pointers). We'll implement these callbacks as follows:

- `get` reads a page from disk.
- `new` allocates and writes a new page (copy-on-write).
- `del` deallocates a page.

We can use fake callbacks (mocks) to test the data structure in memory without the rest of the DB.

4.2 Decode the node format

Since the node type is just a chunk of bytes, we'll define some helper functions to access it.

```
| type | nkeys |  pointers  |   offsets  | key-values | unused |
|  2B  |   2B  | nkeys * 8B | nkeys * 2B |     ...    |        |

| klen | vlen | key | val |
|  2B  |  2B  | ... | ... |
```

Header

```
const (
    BNODE_NODE = 1 // internal nodes without values
    BNODE_LEAF = 2 // leaf nodes with values
)

func (node BNode) btype() uint16 {
    return binary.LittleEndian.Uint16(node[0:2])
}
func (node BNode) nkeys() uint16 {
    return binary.LittleEndian.Uint16(node[2:4])
}
func (node BNode) setHeader(btype uint16, nkeys uint16) {
    binary.LittleEndian.PutUint16(node[0:2], btype)
    binary.LittleEndian.PutUint16(node[2:4], nkeys)
}
```

Child pointers

```
// pointers
func (node BNode) getPtr(idx uint16) uint64 {
    assert(idx < node.nkeys())
    pos := HEADER + 8*idx
```

```
    return binary.LittleEndian.Uint64(node[pos:])
}
func (node BNode) setPtr(idx uint16, val uint64)
```

KV offsets and pairs

The format packs everything back to back. Finding the nth KV can be done by reading each KV pair one by one. To make it easier, we have included an offset list to locate the nth KV in $O(1)$. This also allows binary searches within a node.

Each offset is the *end* of the KV pair relative to the start of the 1st KV. The start offset of the 1st KV is just 0, so we use the end offset instead, which is the start offset of the next KV.

```
// offset list
func offsetPos(node BNode, idx uint16) uint16 {
    assert(1 <= idx && idx <= node.nkeys())
    return HEADER + 8*node.nkeys() + 2*(idx-1)
}
func (node BNode) getOffset(idx uint16) uint16 {
    if idx == 0 {
        return 0
    }
    return binary.LittleEndian.Uint16(node[offsetPos(node, idx):])
}
func (node BNode) setOffset(idx uint16, offset uint16)
```

kvPos returns the position of the nth KV pair relative to the whole node.

```
// key-values
func (node BNode) kvPos(idx uint16) uint16 {
    assert(idx <= node.nkeys())
    return HEADER + 8*node.nkeys() + 2*node.nkeys() + node.getOffset(idx)
}
func (node BNode) getKey(idx uint16) []byte {
    assert(idx < node.nkeys())
```

```
    pos := node.kvPos(idx)
    klen := binary.LittleEndian.Uint16(node[pos:])
    return node[pos+4:][:klen]
}
func (node BNode) getVal(idx uint16) []byte
```

It also conveniently returns the node size (used space) with an off-by-one lookup.

```
// node size in bytes
func (node BNode) nbytes() uint16 {
    return node.kvPos(node.nkeys())
}
```

KV lookups within a node

The “seek” operation is used for both range and point queries. So they are fundamentally the same.

```
// returns the first kid node whose range intersects the key. (kid[i] <= key)
// TODO: binary search
func nodeLookupLE(node BNode, key []byte) uint16 {
    nkeys := node.nkeys()
    found := uint16(0)
    // the first key is a copy from the parent node,
    // thus it's always less than or equal to the key.
    for i := uint16(1); i < nkeys; i++ {
        cmp := bytes.Compare(node.getKey(i), key)
        if cmp <= 0 {
            found = i
        }
        if cmp >= 0 {
            break
        }
    }
    return found
}
```

The function is called `nodeLookupLE` because it uses the **L**ess-than-or-**E**qual operator. For point queries, we should use the equal operator instead, which is a step we can add later.

4.2 Update B+tree nodes

Insert into leaf nodes

Let's consider inserting a key into a leaf node. The 1st step is to use **nodeLookupLE** to get the insert position. Then copy everything to a new node with the extra key. That's copy-on-write.

```
// add a new key to a leaf node
func leafInsert(
    new BNode, old BNode, idx uint16,
    key []byte, val []byte,
) {
    new.setHeader(BNODE_LEAF, old.nkeys()+1) // setup the header
    nodeAppendRange(new, old, 0, 0, idx)
    nodeAppendKV(new, idx, 0, key, val)
    nodeAppendRange(new, old, idx+1, idx, old.nkeys()-idx)
}
```

Node copying functions

nodeAppendRange copies a range of KVs and **nodeAppendKV** copies a KV pair. This must be done in order because these functions rely on the previous offset.

```
// copy a KV into the position
func nodeAppendKV(new BNode, idx uint16, ptr uint64, key []byte, val []byte) {
    // ptrs
    new.setPtr(idx, ptr)
    // KVs
    pos := new.kvPos(idx)
    binary.LittleEndian.PutUint16(new[pos+0:], uint16(len(key)))
    binary.LittleEndian.PutUint16(new[pos+2:], uint16(len(val)))
    copy(new[pos+4:], key)
    copy(new[pos+4+uint16(len(key)):], val)
```

```
    // the offset of the next key
    new.setOffset(idx+1, new.getOffset(idx)+4+uint16((len(key)+len(val))))
}

// copy multiple KVs into the position from the old node
func nodeAppendRange(
    new BNode, old BNode,
    dstNew uint16, srcOld uint16, n uint16,
)
```

Update internal nodes

For internal nodes, the link to the child node is always updated with the copy-on-write scheme, which can become multiple links if the child node is split.

```
// replace a link with one or multiple links
func nodeReplaceKidN(
    tree *BTree, new BNode, old BNode, idx uint16,
    kids ...BNode,
) {
    inc := uint16(len(kids))
    new.setHeader(BNODE_NODE, old.nkeys()+inc-1)
    nodeAppendRange(new, old, 0, 0, idx)
    for i, node := range kids {
        nodeAppendKV(new, idx+uint16(i), tree.new(node), node.getKey(0), nil)
        //               ^position      ^pointer        ^key            ^val
    }
    nodeAppendRange(new, old, idx+inc, idx+1, old.nkeys()-(idx+1))
}
```

Note that the `tree.new` callback is used to allocate the child nodes.

4.3 Split B+tree nodes

Due to the size limits we imposed, a node can host at least 1 KV pair. In the worst case, an oversized node will be split into 3 nodes, with a large KV in the middle. So we may have to split it 2 times.

```
// split a oversized node into 2 so that the 2nd node always fits on a page
func nodeSplit2(left BNode, right BNode, old BNode) {
    // code omitted...
}

// split a node if it's too big. the results are 1~3 nodes.
func nodeSplit3(old BNode) (uint16, [3]BNode) {
    if old.nbytes() <= BTREE_PAGE_SIZE {
        old = old[:BTREE_PAGE_SIZE]
        return 1, [3]BNode{old} // not split
    }
    left := BNode(make([]byte, 2*BTREE_PAGE_SIZE)) // might be split later
    right := BNode(make([]byte, BTREE_PAGE_SIZE))
    nodeSplit2(left, right, old)
    if left.nbytes() <= BTREE_PAGE_SIZE {
        left = left[:BTREE_PAGE_SIZE]
        return 2, [3]BNode{left, right} // 2 nodes
    }
    leftleft := BNode(make([]byte, BTREE_PAGE_SIZE))
    middle := BNode(make([]byte, BTREE_PAGE_SIZE))
    nodeSplit2(leftleft, middle, left)
    assert(leftleft.nbytes() <= BTREE_PAGE_SIZE)
    return 3, [3]BNode{leftleft, middle, right} // 3 nodes
}
```

Note that the returned nodes are allocated from memory; they are just temporary data until `nodeReplaceKidN` actually allocates them.

4.4 B+tree insertion

We've implemented 3 node operations:

- `leafInsert` updates a leaf node.
- `nodeReplaceKidN` updates an internal node.
- `nodeSplit3` splits an oversized node.

Let's put them together for a full B+tree insertion, which starts with key lookups in the root node until it reaches a leaf.

```
// insert a KV into a node, the result might be split.
// the caller is responsible for deallocating the input node
// and splitting and allocating result nodes.
func treeInsert(tree *BTree, node BNode, key []byte, val []byte) BNode {
    // the result node.
    // it's allowed to be bigger than 1 page and will be split if so
    new := BNode{data: make([]byte, 2*BTREE_PAGE_SIZE)}

    // where to insert the key?
    idx := nodeLookupLE(node, key)
    // act depending on the node type
    switch node.btype() {
    case BNODE_LEAF:
        // leaf, node.getKey(idx) <= key
        if bytes.Equal(key, node.getKey(idx)) {
            // found the key, update it.
            leafUpdate(new, node, idx, key, val)
        } else {
            // insert it after the position.
            leafInsert(new, node, idx+1, key, val)
        }
    case BNODE_NODE:
        // internal node, insert it to a kid node.
        nodeInsert(tree, new, node, idx, key, val)
    default:
```

```
        panic("bad node!")
    }
    return new
}
```

leafUpdate is similar to **leafInsert**; it updates an existing key instead of inserting a duplicate key.

```
// part of the treeInsert(): KV insertion to an internal node
func nodeInsert(
    tree *BTree, new BNode, node BNode, idx uint16,
    key []byte, val []byte,
) {
    kptr := node.getPtr(idx)
    // recursive insertion to the kid node
    knode := treeInsert(tree, tree.get(kptr), key, val)
    // split the result
    nsplit, split := nodeSplit3(knode)
    // deallocate the kid node
    tree.del(kptr)
    // update the kid links
    nodeReplaceKidN(tree, new, node, idx, split[:nsplit]...)
}
```

Internal nodes are handled recursively, each call returns an updated node, and the caller will split it if it’s oversized and handle the allocation/deallocation.

4.5 What’s next?

The work is almost done. We just need to add these in the next chapter:

1. Node merging and tree deletion.
2. A high-level interface.
3. Fake node callbacks for tests.

05. B+Tree Deletion and Testing

5.1 High-level interfaces

We'll add the interfaces to use the B+tree as a KV.

```
// insert a new key or update an existing key
func (tree *BTree) Insert(key []byte, val []byte)
// delete a key and returns whether the key was there
func (tree *BTree) Delete(key []byte) bool
```

Most of the details are introduced with the tree insertion, so there's not much more to learn from the deletion. Skip this chapter if you know the principle.

Keep the root node

There is some extra work in maintaining the root node for tree insertions.

- Create the root node if the tree is empty.
- Add a new root if the root node is split.

```
func (tree *BTree) Insert(key []byte, val []byte) {
    if tree.root == 0 {
        // create the first node
        root := BNode(make([]byte, BTREE_PAGE_SIZE))
        root.setHeader(BNODE_LEAF, 2)
        // a dummy key, this makes the tree cover the whole key space.
        // thus a lookup can always find a containing node.
        nodeAppendKV(root, 0, 0, nil, nil)
        nodeAppendKV(root, 1, 0, key, val)
        tree.root = tree.new(root)
        return
    }

    node := treeInsert(tree, tree.get(tree.root), key, val)
```

```
    nsplit, split := nodeSplit3(node)
    tree.del(tree.root)
    if nsplit > 1 {
        // the root was split, add a new level.
        root := BNode(make([]byte, BTREE_PAGE_SIZE))
        root.setHeader(BNODE_NODE, nsplit)
        for i, knode := range split[:nsplit] {
            ptr, key := tree.new(knode), knode.getKey(0)
            nodeAppendKV(root, uint16(i), ptr, key, nil)
        }
        tree.root = tree.new(root)
    } else {
        tree.root = tree.new(split[0])
    }
}
```

Sentinel value

There is a trick when creating the first root: we inserted an empty key. This is called a sentinel value, it's used to remove an edge case.

If you examine the lookup function `nodeLookupLE`, you'll see that it won't work if the key is out of the node range. This is fixed by inserting an empty key into the tree, which is the lowest possible key by sort order, so that `nodeLookupLE` will always find a position.

5.2 Merge nodes

Node update functions

We'll need some new functions for the tree deletion.

```
// remove a key from a leaf node
func leafDelete(new BNode, old BNode, idx uint16)
// merge 2 nodes into 1
func nodeMerge(new BNode, left BNode, right BNode)
// replace 2 adjacent links with 1
func nodeReplace2Kid(
    new BNode, old BNode, idx uint16, ptr uint64, key []byte,
)
```

Merge conditions

Deleting may result in empty nodes, which can be merged with a sibling if it has one. **shouldMerge** returns which sibling (left or right) to merge with.

```
// should the updated kid be merged with a sibling?
func shouldMerge(
    tree *BTree, node BNode,
    idx uint16, updated BNode,
) (int, BNode) {
    if updated.nbytes() > BTREE_PAGE_SIZE/4 {
        return 0, BNode{}
    }

    if idx > 0 {
        sibling := BNode(tree.get(node.getPtr(idx - 1)))
        merged := sibling.nbytes() + updated.nbytes() - HEADER
        if merged <= BTREE_PAGE_SIZE {
            return -1, sibling // left
```

```
        }
    }
    if idx+1 < node.nkeys() {
        sibling := BNode(tree.get(node.getPtr(idx + 1)))
        merged := sibling.nbytes() + updated.nbytes() - HEADER
        if merged <= BTREE_PAGE_SIZE {
            return +1, sibling // right
        }
    }
    return 0, BNode{}
}
```

Deleted keys mean unused space within nodes. In the worst case, a mostly empty tree can still retain a large number of nodes. We can improve this by triggering merges earlier — using 1/4 of a page as a threshold instead of the empty node, which is a soft limit on the minimum node size.

5.3 B+tree deletion

This is similar to insertion, just replace splitting with merging.

```
// delete a key from the tree
func treeDelete(tree *BTree, node BNode, key []byte) BNode

// delete a key from an internal node; part of the treeDelete()
func nodeDelete(tree *BTree, node BNode, idx uint16, key []byte) BNode {
    // recurse into the kid
    kptr := node.getPtr(idx)
    updated := treeDelete(tree, tree.get(kptr), key)
    if len(updated) == 0 {
        return BNode{} // not found
    }
    tree.del(kptr)

    new := BNode(make([]byte, BTREE_PAGE_SIZE))
    // check for merging
    mergeDir, sibling := shouldMerge(tree, node, idx, updated)
    switch {
    case mergeDir < 0: // left
        merged := BNode(make([]byte, BTREE_PAGE_SIZE))
        nodeMerge(merged, sibling, updated)
        tree.del(node.getPtr(idx - 1))
        nodeReplace2Kid(new, node, idx-1, tree.new(merged), merged.getKey(0))
    case mergeDir > 0: // right
        merged := BNode(make([]byte, BTREE_PAGE_SIZE))
        nodeMerge(merged, updated, sibling)
        tree.del(node.getPtr(idx + 1))
        nodeReplace2Kid(new, node, idx, tree.new(merged), merged.getKey(0))
    case mergeDir == 0 && updated.nkeys() == 0:
        assert(node.nkeys() == 1 && idx == 0) // 1 empty child but no sibling
        new.setHeader(BNODE_NODE, 0)          // the parent becomes empty too
    case mergeDir == 0 && updated.nkeys() > 0: // no merge
        nodeReplaceKidN(tree, new, node, idx, updated)
    }
```

```
    return new
}
```

Even if a node becomes empty, it may not be merged if it has no siblings. In this case, the empty node is propagated to its parent and merged later.

5.4 Test the B+tree

The data structure only interacts with the rest of the DB via the 3 page management callbacks. To test the B+tree, we can simulate pages in memory.

```
type C struct {
    tree  BTree
    ref   map[string]string // the reference data
    pages map[uint64]BNode  // in-memory pages
}

func newC() *C {
    pages := map[uint64]BNode{}
    return &C{
        tree: BTree{
            get: func(ptr uint64) []byte {
                node, ok := pages[ptr]
                assert(ok)
                return node
            },
            new: func(node []byte) uint64 {
                assert(BNode(node).nbytes() <= BTREE_PAGE_SIZE)
                ptr := uint64(uintptr(unsafe.Pointer(&node[0])))
                assert(pages[ptr] == nil)
                pages[ptr] = node
                return ptr
            },
            del: func(ptr uint64) {
                assert(pages[ptr] != nil)
                delete(pages, ptr)
            },
        },
        ref:   map[string]string{},
        pages: pages,
    }
}
```

`C.pages` is a map of allocated pages. It's used to validate pointers and read pages. The pointers are actually in-memory pointers, and the B+tree code doesn't care.

To test the B+tree, we first need to update it under various scenarios and then verify the result. The verification is generic, there are 2 things to verify:

1. The structure is valid.
    - Keys are sorted.
    - Node sizes are within limits.
2. The data matches a reference. We used a map to capture each update.

```
func (c *C) add(key string, val string) {
    c.tree.Insert([]byte(key), []byte(val))
    c.ref[key] = val // reference data
}
```

The test cases are left as an exercise. The next thing is B+tree on disk.

06. Append-Only KV Store

6.1 What we will do

We'll create a KV store with a copy-on-write B+tree backed by a file.

```
type KV struct {
    Path   string // file name
    // internals
    fd   int
    tree BTree
    // more ...
}
func (db *KV) Open() error

func (db *KV) Get(key []byte) ([]byte, bool) {
    return db.tree.Get(key)
}
func (db *KV) Set(key []byte, val []byte) error {
    db.tree.Insert(key, val)
    return updateFile(db)
}
func (db *KV) Del(key []byte) (bool, error) {
    deleted := db.tree.Delete(key)
    return deleted, updateFile(db)
}
```

The scope of this chapter is durability + atomicity:

- The file is append-only; space reuse is left to the next chapter.
- We will ignore concurrency and assume sequential access within 1 process.

We'll implement the 3 B+tree callbacks that deal with disk pages:

```
type BTree struct {
    root uint64
    get func(uint64) []byte // read a page
```

```
    new func([]byte) uint64 // append a page
    del func(uint64)        // ignored in this chapter
}
```

6.2 Two-phase update

Atomicity + durability

As discussed in chapter 03, for a copy-on-write tree, the root pointer is updated atomically. Then `fsync` is used to *request* and *confirm* durability.

The atomicity of the root pointer itself is insufficient; to make the whole tree atomic, new nodes must be persisted *before* the root pointer. And **the write order is not the order in which the data is persisted**, due to factors like caching. So another `fsync` is used to ensure the order.

```
func updateFile(db *KV) error {
    // 1. Write new nodes.
    if err := writePages(db); err != nil {
        return err
    }
    // 2. `fsync` to enforce the order between 1 and 3.
    if err := syscall.Fsync(db.fd); err != nil {
        return err
    }
    // 3. Update the root pointer atomically.
    if err := updateRoot(db); err != nil {
        return err
    }
    // 4. `fsync` to make everything persistent.
    return syscall.Fsync(db.fd)
}
```

Alternative: durability with a log

The alternative double-write scheme also has 2 `fsync`'ed phases:

1. Write the updated pages with checksum.
2. `fsync` to make the update persistent (for crash recovery).
3. Update the data in-place (apply the double-writes).
4. `fsync` for the order between 3 and 1 (reuse or delete the double-writes).

A difference with copy-on-write is the order of the phases: the data is persistent after the 1st `fsync`; the DB can return success and do the rest in the background.

The double-write is comparable to a log, which also needs only 1 `fsync` for an update. And it can be an actual log to buffer multiple updates, which improves performance. This is another example of logs in DBs, besides the LSM-tree.

We won't use a log as copy-on-write doesn't need it. But a log still offers the benefits discussed above; it's one of the reasons logs are ubiquitous in databases.

Concurrency of in-memory data

Atomicity for in-memory data (w.r.t. concurrency) can be achieved with a mutex (lock) or some atomic CPU instructions. There is a similar problem: memory reads/writes may not appear in order due to factors like out-of-order execution.

For an in-memory copy-on-write tree, new nodes must be made visible to concurrent readers *before* the root pointer is updated. This is called a *memory barrier* and is analogous to `fsync`, although `fsync` is more than enforcing order.

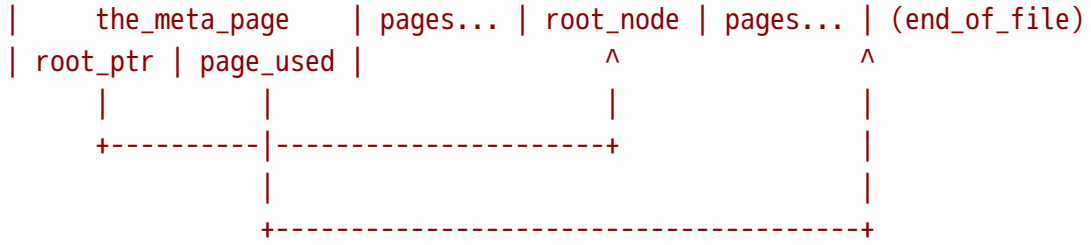
Synchronization primitives such as mutexes, or any OS syscalls, will enforce memory ordering in a portable way, so you don't have to mess with

CPU-specific atomics or barriers (which are inadequate for concurrency anyway).

6.3 Database on a file

The file layout

Our DB is a single file divided into “pages”. Each page is a B+tree node, except for the 1st page; the 1st page contains the pointer to the latest root node and other auxiliary data, we call this the *meta page*.



New nodes are simply appended like a log, but we cannot use the file size to count the number of pages, because after a power loss the file size (metadata) may become inconsistent with the file data. This is filesystem dependent, we can avoid this by storing the number of pages in the meta page.

`fsync` on directory

As mentioned in chapter 01, `fsync` must be used on the parent directory after a `rename`. This is also true when creating new files, because there are 2 things to be made persistent: the file data, and the directory that references the file.

We’ll preemptively `fsync` after potentially creating a new file with `O_CREATE`. To `fsync` a directory, `open` the directory in `O_RDONLY` mode.

```go
func createFileSync(file string) (int, error) {
    // obtain the directory fd
    flags := os.O_RDONLY | syscall.O_DIRECTORY
    dirfd, err := syscall.Open(path.Dir(file), flags, 0o644)
    if err != nil {
        return -1, fmt.Errorf("open directory: %w", err)
    }
    defer syscall.Close(dirfd)
    // open or create the file
    flags = os.O_RDWR | os.O_CREATE
    fd, err := syscall.Openat(dirfd, path.Base(file), flags, 0o644)
    if err != nil {
        return -1, fmt.Errorf("open file: %w", err)
    }
    // fsync the directory
    if err = syscall.Fsync(dirfd); err != nil {
        _ = syscall.Close(fd)  // may leave an empty file
        return -1, fmt.Errorf("fsync directory: %w", err)
    }
    return fd, nil
}
```

The directory fd can be used by **`openat`** to open the target file, which guarantees that the file is from the same directory we opened before, in case the directory path is replaced in between (race condition). Although this is not our concern as we don't expect multi-process operations.

`mmap`, page cache and IO

`mmap` is a way to read/write a file as if it's an in-memory buffer. Disk IO is implicit and automatic with **`mmap`**.

```go
func Mmap(fd int, offset int64, length int, ...) (data []byte, err error)
```

To understand **`mmap`**, let's review some operating system basics. An OS page is the minimum unit for mapping between virtual address and physical address. However, the virtual address space of a process is not fully backed

by physical memory all the time; part of the process memory can be swapped to disk, and when the process tries to access it:

1. The CPU triggers a *page fault*, which hands control to the OS.
2. The OS then …
    1. Reads the swapped data into physical memory.
    2. Remaps the virtual address to it.
    3. Hands control back to the process.
3. The process resumes with the virtual address mapped to real RAM.

`mmap` works in a similar way, the process gets an address range from `mmap` and when it touches a page in it, it page faults and the OS reads the data into a cache and remaps the page to the cache. That's the automatic IO in a read-only scenario.

The CPU also takes note (called a dirty bit) when the process modifies a page so the OS can write the page back to disk later. `fsync` is used to request and wait for the IO. This is writing data via `mmap`, it is not very different from `write` on Linux because `write` goes to the same page cache.

You don't have to `mmap`, but it's important to understand the basics.

6.4 Manage disk pages

We'll use `mmap` to implement these page management callbacks. because it's just convenient.

```
func (db *KV) Open() error {
    db.tree.get = db.pageRead   // read a page
    db.tree.new = db.pageAppend // apppend a page
    db.tree.del = func(uint64) {}
    // ...
}
```

Invoke `mmap`

A file-backed `mmap` can be either read-only, read-write, or copy-on-write. To create a read-only `mmap`, use the `PROT_READ` and `MAP_SHARED` flags.

```
syscall.Mmap(fd, offset, size, syscall.PROT_READ, syscall.MAP_SHARED)
```

The mapped range can be larger than the current file size, which is a fact that we can exploit because the file will grow.

`mmap` a growing file

`mremap` remaps a mapping to a larger range, it's like `realloc`. That's one way to deal with the growing file. However, the address may change, which can hinder concurrent readers in later chapters. Our solution is to add new mappings to cover the expanded file.

```
type KV struct {
    // ...
    mmap struct {
```

```
        total  int      // mmap size, can be larger than the file size
        chunks [][]byte // multiple mmaps, can be non-continuous
    }
}

// `BTree.get`, read a page.
func (db *KV) pageRead(ptr uint64) []byte {
    start := uint64(0)
    for _, chunk := range db.mmap.chunks {
        end := start + uint64(len(chunk))/BTREE_PAGE_SIZE
        if ptr < end {
            offset := BTREE_PAGE_SIZE * (ptr - start)
            return chunk[offset : offset+BTREE_PAGE_SIZE]
        }
        start = end
    }
    panic("bad ptr")
}
```

Adding a new mapping each time the file is expanded results in lots of mappings, which is bad for performance because the OS has to keep track of them. This is avoided with exponential growth, since **mmap** can go beyond the file size.

```
func extendMmap(db *KV, size int) error {
    if size <= db.mmap.total {
        return nil // enough range
    }
    alloc := max(db.mmap.total, 64<<20) // double the current address space
    for db.mmap.total + alloc < size {
        alloc *= 2 // still not enough?
    }
    chunk, err := syscall.Mmap(
        db.fd, int64(db.mmap.total), alloc,
        syscall.PROT_READ, syscall.MAP_SHARED, // read-only
    )
    if err != nil {
        return fmt.Errorf("mmap: %w", err)
    }
```

```
    db.mmap.total += alloc
    db.mmap.chunks = append(db.mmap.chunks, chunk)
    return nil
}
```

You may wonder why not just create a very large mapping (say, 1TB) and forget about the growing file, since an unrealized virtual address costs nothing. This is OK for a toy DB in 64-bit systems.

Capture page updates

The `BTree.new` callback collects new pages from B+tree updates, and allocates the page number from the end of DB.

```
type KV struct {
    // ...
    page struct {
        flushed uint64   // database size in number of pages
        temp    [][]byte // newly allocated pages
    }
}

func (db *KV) pageAppend(node []byte) uint64 {
    ptr := db.page.flushed + uint64(len(db.page.temp)) // just append
    db.page.temp = append(db.page.temp, node)
    return ptr
}
```

Which are written (appended) to the file after B+tree updates.

```
func writePages(db *KV) error {
    // extend the mmap if needed
    size := (int(db.page.flushed) + len(db.page.temp)) * BTREE_PAGE_SIZE
    if err := extendMmap(db, size); err != nil {
        return err
    }
    // write data pages to the file
    offset := int64(db.page.flushed * BTREE_PAGE_SIZE)
```

```
    if _, err := unix.Pwritev(db.fd, db.page.temp, offset); err != nil {
        return err
    }
    // discard in-memory data
    db.page.flushed += uint64(len(db.page.temp))
    db.page.temp = db.page.temp[:0]
    return nil
}
```

pwritev is variant of **write** with an offset and multiple input buffers. We have to control the offset because we also need to write the meta page later. Multiple input buffers are combined by the kernel.

6.5 The meta page

Read the meta page

We'll also add some magic bytes to the meta page to identify the file type.

```
const DB_SIG = "BuildYourOwnDB06" // not compatible between chapters

// | sig | root_ptr | page_used |
// | 16B |    8B    |     8B    |
func saveMeta(db *KV) []byte {
    var data [32]byte
    copy(data[:16], []byte(DB_SIG))
    binary.LittleEndian.PutUint64(data[16:], db.tree.root)
    binary.LittleEndian.PutUint64(data[24:], db.page.flushed)
    return data[:]
}

func loadMeta(db *KV, data []byte)
```

The meta page is reserved if the file is empty.

```
func readRoot(db *KV, fileSize int64) error {
    if fileSize == 0 { // empty file
        db.page.flushed = 1 // the meta page is initialized on the 1st write
        return nil
    }
    // read the page
    data := db.mmap.chunks[0]
    loadMeta(db, data)
    // verify the page
    // ...
    return nil
}
```

Update the meta page

Writing a small amount of page-aligned data to a real disk, modifying only a single sector, is likely power-loss-atomic at the hardware level. Some real databases depend on this. That's how we update the meta page too.

```
// 3. Update the meta page. it must be atomic.
func updateRoot(db *KV) error {
    if _, err := syscall.Pwrite(db.fd, saveMeta(db), 0); err != nil {
        return fmt.Errorf("write meta page: %w", err)
    }
    return nil
}
```

However, atomicity means different things at different levels, as you've seen with `rename`. `write` is not atomic w.r.t. concurrent readers at the system call level. This is likely how the page cache works.

We'll consider read/write atomicity when we add concurrent transations, but we have already seen a solution: In an LSM-tree, the 1st level is the only thing that is updated, and it's duplicated as a MemTable, which moves the concurrency problem to memory. We can keep an in-memory copy of the meta page and synchronize it with a mutex, thus avoiding concurrent disk reads/writes.

Even if the hardware is not atomic w.r.t. power loss. Atomicity is achievable with log + checksum. We could switch between 2 checksumed meta pages for each update, to ensure that one of them is good after a power loss. This is called *double buffering*, which is a rotating log with 2 entries.

6.6 Error handling

Scenarios after IO errors

The bare minimum of error handling is to propagate errors with `if err != nil`. Next, consider the possibility of using the DB after an IO error (`fsync` or `write`).

- Read after a failed update?
  - The reasonable choice is to behave as if nothing happened.
- Update it again after a failure?
  - If the error persists, it's expected to fail again.
  - If the error is temporary, can we recover from the previous error?
- Restart the DB after the problem is resolved?
  - This is just crash recovery; discussed in chapter 03.

Revert to the previous version

There is a survey on the handling of `fsync` failures. From which we can learn that the topic is filesystem dependent. If we read after an `fsync` failure, some filesystems return the failed data as the page cache doesn't match the disk. So reading back failed writes is problematic.

But since we're copy-on-write, this is not a problem; we can revert to the old tree root to avoid the problematic data. The tree root is stored in the meta page, but we never read the meta page from disk after opening a DB, so we'll just revert the *in-memory* root pointer.

```go
func (db *KV) Set(key []byte, val []byte) error {
    meta := saveMeta(db) // save the in-memory state (tree root)
    db.tree.Insert(key, val)
    return updateOrRevert(db, meta)
}

func updateOrRevert(db *KV, meta []byte) error {
    // 2-phase update
    err := updateFile(db)
    // revert on error
    if err != nil {
        // the in-memory states can be reverted immediately to allow reads
        loadMeta(db, meta)
        // discard temporaries
        db.page.temp = db.page.temp[:0]
    }
    return err
}
```

So after a write failure, it's still possible to use the DB in read-only mode. Reads can also fail, but we're using `mmap`, on a read error the process is just killed by `SIGBUS`. That's one of the drawbacks of `mmap`.

Recover from temporary write errors

Some write errors are temporary, such as "no space left". If an update fails and then the next succeeds, the end state is still good. The problem is the intermediate state: between the 2 updates, the content of the meta page *on disk* is unknown!

If `fsync` fails on the meta page, the meta page on disk can be either the new or the old version, while the in-memory tree root is the old version. So the 2nd successful update will overwrite the data pages of the newer version, which can be left in a corrupted intermediate state if crashed.

The solution is to rewrite the last known meta page on recovery.

```
type KV struct {
    // ...
    failed bool // Did the last update fail?
}

func updateOrRevert(db *KV, meta []byte) error {
    // ensure the on-disk meta page matches the in-memory one after an error
    if db.failed {
        // write and fsync the previous meta page
        // ...
        db.failed = false
    }
    err := updateFile(db)
    if err != nil {
        // the on-disk meta page is in an unknown state;
        // mark it to be rewritten on later recovery.
        db.failed = true
        // ...
    }
    return err
}
```

We rely on filesystems to report errors correctly, but there is evidence that they don’t. So can the system as a whole handle errors is still doubtful.

6.7 Summary of the append-only KV store

- File layout for a copy-on-write B+tree.
- Durability and atomicity with `fsync`.
- Error handling.

B+tree on disk is a major step. We just have to add a free list to make it practical.

07. Free List: Recyle & Reuse

The last step of the KV store is to reuse deleted pages, which is also a problem for in-memory data structures.

7.1 Memory management techniques

What we will do

Memory (space) management can be either manual or automatic. A garbage collector is automatic, it detects unused objects without any help from the programmer. The next problem is how to deal with (reuse) unused objects.

We don't need a GC, because in a tree data structure, detecting unused nodes is trivial, as we have already done with the `BTree.del` callback. What we'll do is reimplement those callbacks.

List of unused objects

One of the reasons that disk space is managed in *pages* of the same size is that they become *interchangeable* after they are deleted; the DB can reuse any of them when it needs a page. This is simpler than generic memory management routines such as `malloc`, which deal with arbitrary sizes.

We need to store a list of unused pages, called a *free list* or *object pool*. For in-memory data, this can simply be an array of pointers, or a linked list embedded in objects.

Embedded linked list

The simplest scheme is to use an embedded (intrusive) linked list. The list pointer sits inside the object itself; it borrows space from the object, so no extra space is needed for the data structure.

```
  head
   ↓
[ next | space... ] (unused object 1)
   ↓
[ next | space... ] (unused object 2)
   ↓
  ...
```

However, this conflicts with copy-on-write, as it overwrites *during* an update.

External list

The other scheme is to store pointers to unused pages in an external data structure. The external data structure itself takes up space, which is a problem we'll solve.

Let's say our free list is just a log of unused page numbers; adding items is just appending. The problem is how to remove items so that it doesn't grow infinitely.

7.2 Linked list on disk

Free list requirements

Let's image the free list as a sequence of items, like a log. In a copy-on-write tree, each update requires new nodes and deletes old nodes, so the free list is both added to and removed from per update. If items are removed from the end, then the added items *overwrite* old data, requiring extra crash recovery mechanisms discussed in chapter 03.

If items are removed from the beginning, how do you reclaim the space from the removed items? We're back to the original problem.

To solve the problem, the free list should also be page-based, so that it can manage itself. A page-based list is just a linked list, except that a page can hold multiple items, like a B+tree node. This is also called an *unrolled linked list*.

In summary:

- Our free list is a standalone data structure: a linked list of pages.
  - It will try to get a page from itself when it grows a new node.
  - Removed list nodes are added to itself for reuse.
- Each page can contain multiple items (page numbers).
  - Pages are updated in-place, but it's still *append-only* within a page.
- Items are appended to the tail node and consumed from the head.
  - It's easier to make the tail node append-only this way.

Free list disk layout

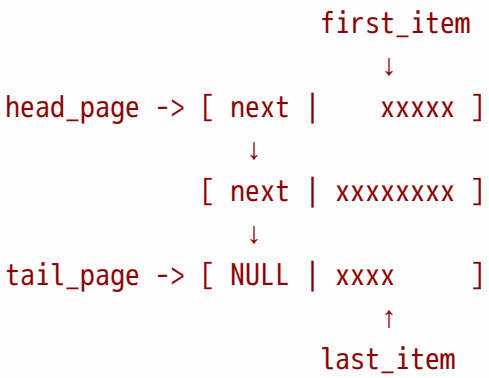
Each node starts with a pointer to the next node. Items are appended next to it.

```
// node format:
// | next | pointers | unused |
// |  8B  |   n*8B   |   ...  |
type LNode []byte

const FREE_LIST_HEADER = 8
const FREE_LIST_CAP = (BTREE_PAGE_SIZE - FREE_LIST_HEADER) / 8

// getters & setters
func (node LNode) getNext() uint64
func (node LNode) setNext(next uint64)
func (node LNode) getPtr(idx int) uint64
func (node LNode) setPtr(idx int, ptr uint64)
```

We also store pointers to both the head node and the tail node in the meta page. The pointer to the tail node is needed for O(1) insertion.



Update free list nodes

Without the free list, the meta page is the only page that is updated *in-place*, which is how copy-on-write made crash recovery easy. Now there are 2 more in-place page updates in list nodes: the next pointer and the appended items.

Although list nodes are updated in-place, no data is overwritten within a page. So if an update is interrupted, the meta page still points to the same data; no extra crash recovery is needed. And unlike the meta page, atomicity is not required.

Following this analysis, the embedded list can also work iff the next pointer is *reserved* in the B+tree node. Here you can deviate from the book. Although this doubles write amplification.

7.3 Free list implementation

Free list interface

```
type KV struct {
    Path   string
    // internals
    fd   int
    tree BTree
    free FreeList // added
    // ...
}
```

FreeList is the extra data structure in **KV**.

```
type FreeList struct {
    // callbacks for managing on-disk pages
    get func(uint64) []byte // read a page
    new func([]byte) uint64 // append a new page
    set func(uint64) []byte // update an existing page
    // persisted data in the meta page
    headPage uint64 // pointer to the list head node
    headSeq  uint64 // monotonic sequence number to index into the list head
    tailPage uint64
    tailSeq  uint64
    // in-memory states
    maxSeq uint64 // saved `tailSeq` to prevent consuming newly added items
}

// get 1 item from the list head. return 0 on failure.
func (fl *FreeList) PopHead() uint64
// add 1 item to the tail
func (fl *FreeList) PushTail(ptr uint64)
```

Like **BTree**, page management is isolated via 3 callbacks:

- `get` reads a page; same as before,
- `new` appends a page; previously used for `BTree`.
- `set` returns a writable buffer to capture in-place updates.
- `del` is not there because the free list manages free pages itself.

```
func (db *KV) Open() error {
    // ...
    // B+tree callbacks
    db.tree.get = db.pageRead      // read a page
    db.tree.new = db.pageAlloc     // (new) reuse from the free list or append
    db.tree.del = db.free.PushTail // (new) freed pages go to the free list
    // free list callbacks
    db.free.get = db.pageRead      // read a page
    db.free.new = db.pageAppend    // append a page
    db.free.set = db.pageWrite     // (new) in-place updates
    // ...
}
```

Free list data structure

Since a node contains a variable number of items up to `FREE_LIST_CAP`, we need to know where the 1st item is in the head node (`headSeq`), and where the items end in the tail node (`tailSeq`).

```
type FreeList struct {
    // ...
    // persisted data in the meta page
    headPage uint64 // pointer to the list head node
    headSeq  uint64 // monotonic sequence number to index into the list head
    tailPage uint64
    tailSeq  uint64
    // in-memory states
    maxSeq uint64 // saved `tailSeq` to prevent consuming newly added items
}
```

`headSeq`, `tailSeq` are indexes into the head and tail nodes, except that they are monotonically increasing. So the wrapped-around index is:

```
func seq2idx(seq uint64) int {
    return int(seq % FREE_LIST_CAP)
}
```

We make them monotonically increasing so that they become a unique identifier of the list position; to prevent the list head from overrunning the list tail, simply compare the 2 sequence numbers.

During an update, the list is both added to and removed from, and when we remove from the head, we cannot remove what we just added to the tail. So we need to …

1. At the beginning of the update, save the original **`tailSeq`** to **`maxSeq`**.
2. During the update, **`headSeq`** cannot overrun **`maxSeq`**.
3. At the beginning of the next update, **`maxSeq`** is advanced to **`tailSeq`**.
4. …

```
// make the newly added items available for consumption
func (fl *FreeList) SetMaxSeq() {
    fl.maxSeq = fl.tailSeq
}
```

Consuming from the free list

Removing an item from the head node is simply advancing **`headSeq`**. And when the head node becomes empty, move to the next node.

```
// remove 1 item from the head node, and remove the head node if empty.
func flPop(fl *FreeList) (ptr uint64, head uint64) {
    if fl.headSeq == fl.maxSeq {
        return 0, 0 // cannot advance
    }
    node := LNode(fl.get(fl.headPage))
    ptr = node.getPtr(seq2idx(fl.headSeq)) // item
    fl.headSeq++
    // move to the next one if the head node is empty
```

```
    if seq2idx(fl.headSeq) == 0 {
        head, fl.headPage = fl.headPage, node.getNext()
        assert(fl.headPage != 0)
    }
    return
}
```

The free list self-manages; the removed head node is fed back to itself.

```
// get 1 item from the list head. return 0 on failure.
func (fl *FreeList) PopHead() uint64 {
    ptr, head := flPop(fl)
    if head != 0 { // the empty head node is recycled
        fl.PushTail(head)
    }
    return ptr
}
```

What if the last node is removed? A linked list with 0 nodes implies nasty special cases. In practice, it's easier to **design the linked list to have at least 1 node** than to deal with special cases. That's why we `assert(fl.headPage != 0)`.

Pushing into the free list

Appending an item to the tail node is simply advancing `tailSeq`. And when the tail node is full, we immediately add a new empty tail node to ensure that there is at least 1 node in case the previous tail node is removed as a head node.

```
func (fl *FreeList) PushTail(ptr uint64) {
    // add it to the tail node
    LNode(fl.set(fl.tailPage)).setPtr(seq2idx(fl.tailSeq), ptr)
    fl.tailSeq++
    // add a new tail node if it's full (the list is never empty)
    if seq2idx(fl.tailSeq) == 0 {
        // try to reuse from the list head
```

```
        next, head := flPop(fl) // may remove the head node
        if next == 0 {
            // or allocate a new node by appending
            next = fl.new(make([]byte, BTREE_PAGE_SIZE))
        }
        // link to the new tail node
        LNode(fl.set(fl.tailPage)).setNext(next)
        fl.tailPage = next
        // also add the head node if it's removed
        if head != 0 {
            LNode(fl.set(fl.tailPage)).setPtr(0, head)
            fl.tailSeq++
        }
    }
}
```

Again, the free list is self-managing: it will try to get a node from itself for the new tail node before resorting to appending.

7.4 KV with a free list

Page management

Now that pages can be reused, reused pages are overwritten in-place, so a map is used to capture pending updates.

```
type KV struct {
    // ...
    page struct {
        flushed uint64            // database size in number of pages
        nappend uint64            // number of pages to be appended
        updates map[uint64][]byte // pending updates, including appended pages
    }
}
```

BTree.new is now **KV.pageAlloc**, it uses the free list before resorting to appending.

```
// `BTree.new`, allocate a new page.
func (db *KV) pageAlloc(node []byte) uint64 {
    if ptr := db.free.PopHead(); ptr != 0 { // try the free list
        db.page.updates[ptr] = node
        return ptr
    }
    return db.pageAppend(node) // append
}
```

KV.pageWrite returns a writable page copy to capture in-place updates.

```
// `FreeList.set`, update an existing page.
func (db *KV) pageWrite(ptr uint64) []byte {
    if node, ok := db.page.updates[ptr]; ok {
        return node // pending update
    }
```

```
    node := make([]byte, BTREE_PAGE_SIZE)
    copy(node, db.pageReadFile(ptr)) // initialized from the file
    db.page.updates[ptr] = node
    return node
}
```

Another change is that we may read a page again after it has been updated, so **`KV.pageRead`** should consult the pending updates map first.

```
// `BTree.get`, read a page.
func (db *KV) pageRead(ptr uint64) []byte {
    if node, ok := db.page.updates[ptr]; ok {
        return node // pending update
    }
    return db.pageReadFile(ptr)
}
func (db *KV) pageReadFile(ptr uint64) []byte {
    // same as `KV.pageRead` in the last chapter ...
}
```

Update the meta page

The meta page now includes free list pointers (head and tail) that are updated atomically along with the tree root.

```
| sig | root_ptr | page_used | head_page | head_seq | tail_page | tail_seq |
| 16B |    8B    |     8B    |     8B    |    8B    |     8B    |    8B    |
```

Remember that the free list always contains at least 1 node, we'll assign an empty node to it when initializing an empty DB.

```
func readRoot(db *KV, fileSize int64) error {
    if fileSize == 0 { // empty file
        // reserve 2 pages: the meta page and a free list node
        db.page.flushed = 2
        // add an initial node to the free list so it's never empty
        db.free.headPage = 1 // the 2nd page
        db.free.tailPage = 1
```

```
        return nil // the meta page will be written in the 1st update
    }
    // ...
}
```

Since **`headSeq`** is blocked by **`maxSeq`**, **`maxSeq`** is updated to **`tailSeq`** *between* updates to allow reuse of pages from the last version.

```
func updateFile(db *KV) error {
    // ...
    // prepare the free list for the next update
    db.free.SetMaxSeq()
    return nil
}
```

We still assume sequential access in this chapter. When we add concurrency later, **`headSeq`** will be blocked by the oldest reader instead.

7.5 Conclusion of the KV store

What we have done:

- File layout for a copy-on-write B+tree.
- Durability and atomicity with `fsync`.
- Managing disk pages with a free list.

That's enough for a KV store with `get`, `set`, `del`. But there is more in part II:

- Relational DB on KV store.
- Concurrent transactions.

08. Tables on KV

8.1 Encode rows as KVs

Indexed queries: point and range

In a relational DB, data is modeled as 2D tables consisting of rows and columns. Users speak their *intent* in SQL and the DB magically delivers results. What's less magic is that while a DB can execute arbitrary queries, not all queries are practical (efficient & scalable) in OLTP workloads, and OLTP always requires users to control *how* queries are executed via proper schema and index design.

How an indexed query is executed boils down to 2 operations:

1. Point query: Find a row by a given key.
2. Range query: Find rows by a range; iterate the result in sorted order.

That's why B+trees and LSM-trees are considered, while hashtables are not.

The primary key as the "key"

Let's consider point queries first. To find a row, there must be a way to uniquely identify the row, that's the primary key, which is a subset of the columns.

```
create table t1 (
    k1 string,
    k2 int,
    v1 string,
    v2 string,
    primary key (k1, k2)
);
```

Intuitively, primary key columns go in the “key” and the rest go in the “value”.

	key	value
t1	k1, k2	v1, v2

Some DBs allow tables without a primary key, what they do is add a hidden, auto-generated primary key.

The secondary indexes as separate tables

Besides the primary key, a table can be indexed in more than 1 way. This is solved by an extra indirection: the secondary indexes.

```
create table t1 (
    k1 string,
    k2 int,
    v1 string,
    v2 string,
    primary key (k1, k2),
    index idx1 (v1),
    index idx2 (v2, v1)
);
```

Logically, each index is like a separate table:

```
create table idx1 (
    -- indexed key (v1)
    v1 string,
    -- primary key (k1, k2)
    k1 string,
    k2 int
);
create table idx2 (
    -- indexed key (v2, v1)
    v2 string,
```

```
    v1 string,
    -- primary key (k1, k2)
    k1 string,
    k2 int
);
```

Which adds an extra key to find the unique row identifier (primary key).

	key	value
t1	k1, k2	v1, v2
idx1	v1	k1, k2
idx2	v2, v1	k1, k2

The primary key is also an index, but with the unique constraint.

Alternative: auto-generated row ID

Some DBs use an auto-generated ID as the “true” primary key, as opposed to the user-selected primary key. In this case, there is no distinction between primary and secondaries; the user primary key is also an indirection.

	key	value
t1	ID	k1, k2, v1, v2
primary key	k1, k2	ID
idx1	v1	ID
idx2	v2, v1	ID

The advantage is that the auto-generated ID can be a small, fixed-width integer, while the user primary key can be arbitrarily long. This means that …

- For ID keys, internal nodes can store more keys (shorter tree).

- Secondary indexes are smaller as they don’t duplicate the user primary key.

8.2 Databases schemas

The table prefix

A DB can contain multiple tables and indexes. We'll prepend an auto-generated prefix to the keys so that they can share a single B+tree. This is less work than keeping multiple trees.

	key	value
table1	prefix1 + columns…	columns…
table2	prefix2 + columns…	columns…
index1	prefix3 + columns…	columns…

The prefix is a 32-bit auto-incrementing integer, you can also use the table name instead, with the drawback that it can be arbitrarily long.

Data types

One advantage of relational DB over KV is that they support more data types. To reflect this aspect, we'll support 2 data types: string and integer.

```
const (
    TYPE_BYTES = 1 // string (of arbitrary bytes)
    TYPE_INT64 = 2 // integer; 64-bit signed
)

// table cell
type Value struct {
    Type uint32 // tagged union
    I64  int64
```

```
    Str  []byte
}
```

The cell `Value` is a tagged union of a particular type.

Records

A `Record` represents a list of column names and values.

```
// table row
type Record struct {
    Cols []string
    Vals []Value
}

func (rec *Record) AddStr(col string, val []byte) *Record {
    rec.Cols = append(rec.Cols, col)
    rec.Vals = append(rec.Vals, Value{Type: TYPE_BYTES, Str: val})
    return rec
}
func (rec *Record) AddInt64(col string, val int64) *Record
func (rec *Record) Get(col string) *Value
```

Schemas

We'll only consider the primary key in this chapter, leaving indexes for later.

```
type TableDef struct {
    // user defined
    Name  string
    Types []uint32 // column types
    Cols  []string // column names
    PKeys int      // the first `PKeys` columns are the primary key
    // auto-assigned B-tree key prefixes for different tables
    Prefix uint32
}
```

Internal tables

Where to store table schemas? Since we're coding a DB, we know how to store stuff; we'll store them in a predefined internal table.

```
var TDEF_TABLE = &TableDef{
    Prefix: 2,
    Name:   "@table",
    Types:  []uint32{TYPE_BYTES, TYPE_BYTES},
    Cols:   []string{"name", "def"},
    PKeys:  1,
}
```

The `def` column is JSON serialized `TableDef`. This is like:

```
create table `@table` (
    `name` string,  -- table name
    `def` string,   -- schema
    primary key (`name`)
);
```

We'll also need to keep some extra information, such as an auto-incrementing counter for generating table prefixes. Let's define another internal table for this.

```
var TDEF_META = &TableDef{
    Prefix: 1,
    Name:   "@meta",
    Types:  []uint32{TYPE_BYTES, TYPE_BYTES},
    Cols:   []string{"key", "val"},
    PKeys:  1,
}
```

8.3 Get, update, insert, delete, create

Point query and update interfaces

Interfaces for reading and writing a single row:

```
func (db *DB) Get(table string, rec *Record) (bool, error)
func (db *DB) Insert(table string, rec Record) (bool, error)
func (db *DB) Update(table string, rec Record) (bool, error)
func (db *DB) Upsert(table string, rec Record) (bool, error)
func (db *DB) Delete(table string, rec Record) (bool, error)
```

`DB` is a wrapper of `KV`:

```
type DB struct {
    Path string
    kv KV
}
```

Query by primary key

The `rec` argument is the input primary key. It's also the output row.

```
// get a single row by the primary key
func dbGet(db *DB, tdef *TableDef, rec *Record) (bool, error) {
    // 1. reorder the input columns according to the schema
    values, err := checkRecord(tdef, *rec, tdef.PKeys)
    if err != nil {
        return false, err
    }
    // 2. encode the primary key
    key := encodeKey(nil, tdef.Prefix, values[:tdef.PKeys])
    // 3. query the KV store
    val, ok := db.kv.Get(key)
```

```
    if !ok {
        return false, nil
    }
    // 4. decode the value into columns
    for i := tdef.PKeys; i < len(tdef.Cols); i++ {
        values[i].Type = tdef.Types[i]
    }
    decodeValues(val, values[tdef.PKeys:])
    rec.Cols = tdef.Cols
    rec.Vals = values
    return true, nil
}
```

The code to handle the columns is just mandane, we'll skip it.

```
// reorder a record and check for missing columns.
// n == tdef.PKeys: record is exactly a primary key
// n == len(tdef.Cols): record contains all columns
func checkRecord(tdef *TableDef, rec Record, n int) ([]Value, error)
```

The next step is encoding and decoding, which can be any serialization scheme.

```
// encode columns for the "key" of the KV
func encodeKey(out []byte, prefix uint32, vals []Value) []byte
// decode columns from the "value" of the KV
func decodeValues(in []byte, out []Value)
```

Read the schema

User-facing interfaces refer to tables by name, so we must get its schema first.

```
// get a single row by the primary key
func (db *DB) Get(table string, rec *Record) (bool, error) {
    tdef := getTableDef(db, table)
    if tdef == nil {
        return false, fmt.Errorf("table not found: %s", table)
```

```
    }
    return dbGet(db, tdef, rec)
}
```

Which is just a query to the internal table `@table` of JSON-encoded `TableDef`s.

```
func getTableDef(db *DB, name string) *TableDef {
    rec := (&Record{}).AddStr("name", []byte(name))
    ok, err := dbGet(db, TDEF_TABLE, rec)
    assert(err == nil)
    if !ok {
        return nil
    }

    tdef := &TableDef{}
    err = json.Unmarshal(rec.Get("def").Str, tdef)
    assert(err == nil)
    return tdef
}
```

We can cache table schemas in memory to reduce the number of queries, since no sane application needs a huge number of tables.

Insert or update a row

There are 3 SQL update statements that differ in how they treat an existing row:

- `INSERT` only adds new rows (identified by the primary key).
- `UPDATE` only modifies existing rows.
- `UPSERT` adds new rows or modifies existing rows.

(Note: `UPSERT` is PostgreSQL specific. In MySQL it's `ON DUPLICATED KEY UPDATE`. In SQLite it's `INSERT OR REPLACE`).

This is implemented by extending `BTree.Insert` with a mode flag.

```
// update modes
const (
    MODE_UPSERT      = 0 // insert or replace
    MODE_UPDATE_ONLY = 1 // update existing keys
    MODE_INSERT_ONLY = 2 // only add new keys
)

type UpdateReq struct {
    tree *BTree
    // out
    Added   bool   // added a new key
    // in
    Key  []byte
    Val  []byte
    Mode int
}

func (tree *BTree) Update(req *UpdateReq)
```

The update function here only deals with a complete row. Partial updates (read-modify-write) are implemented at a higher level (query language).

```
func dbUpdate(db *DB, tdef *TableDef, rec Record, mode int) (bool, error) {
    values, err := checkRecord(tdef, rec, len(tdef.Cols))
    if err != nil {
        return false, err
    }
    key := encodeKey(nil, tdef.Prefix, values[:tdef.PKeys])
    val := encodeValues(nil, values[tdef.PKeys:])
    return db.kv.Update(key, val, mode)
}
```

Create a table

The process of creating a table is rather boring:

1. Read **`@table`** to check for duplicate names.
2. Read the table prefix counter from **`@meta`**.

3. Increase and update the table prefix counter in **`@meta`**.
4. Insert the schema to **`@table`**.

```
func (db *DB) TableNew(tdef *TableDef) error
```

This process involves updating 2 keys, so we’re losing atomicity here. This will be fixed later when we add transactions.

8.4 Conclusion of tables on KV

Tables on KV is not fundamentally different, it’s just extra steps of data serialization and keeping schemas. However, the work is not done yet. The next steps are:

- Range queries.
- Secondary indexes.

09. Range Queries

9.1 B+tree iterator

The iterator interface

The basic operations are *seek* and *iterate* for a range query. A B+tree position is represented by the stateful iterator `BIter`.

```
// find the closest position that is less or equal to the input key
func (tree *BTree) SeekLE(key []byte) *BIter

// get the current KV pair
func (iter *BIter) Deref() ([]byte, []byte)
// precondition of the Deref()
func (iter *BIter) Valid() bool
// moving backward and forward
func (iter *BIter) Prev()
func (iter *BIter) Next()
```

For example, the query for `a <= key` looks like this:

```
for iter := tree.SeekLE(key); iter.Valid(); iter.Prev() {
    k, v := iter.Deref()
    // ...
}
```

Navigate a tree

The position of the current key is needed to find its sibling key inside a node. And if the sibling key is in the sibling node, we need to backtrack to the parent node. Since we don’t use parent pointers, we need the entire path from root to leaf.

```
type BIter struct {
    tree *BTree
    path []BNode  // from root to leaf
    pos  []uint16 // indexes into nodes
}
```

Moving the iterator is like carrying when incrementing a number digit by digit.

```
func (iter *BIter) Next() {
    iterNext(iter, len(iter.path)-1)
}
func iterNext(iter *BIter, level int) {
    if iter.pos[level]+1 < iter.path[level].nkeys() {
        iter.pos[level]++ // move within this node
    } else if level > 0 {
        iterNext(iter, level-1) // move to a sibling node
    } else {
        iter.pos[len(iter.pos)-1]++ // past the last key
        return
    }
    if level+1 < len(iter.pos) { // update the child node
        node := iter.path[level]
        kid := BNode(iter.tree.get(node.getPtr(iter.pos[level])))
        iter.path[level+1] = kid
        iter.pos[level+1] = 0
    }
}
```

Seek to a key

Seeking to a key is like a point query, with the path recorded.

```
// find the closest position that is less or equal to the input key
func (tree *BTree) SeekLE(key []byte) *BIter {
    iter := &BIter{tree: tree}
    for ptr := tree.root; ptr != 0; {
        node := tree.get(ptr)
        idx := nodeLookupLE(node, key)
```

```
        iter.path = append(iter.path, node)
        iter.pos = append(iter.pos, idx)
        ptr = node.getPtr(idx)
    }
    return iter
}
```

nodeLookupLE is for less-than-and-equal, you'll also need other operators.

```
const (
    CMP_GE = +3 // >=
    CMP_GT = +2 // >
    CMP_LT = -2 // <
    CMP_LE = -3 // <=
)
func (tree *BTree) Seek(key []byte, cmp int) *BIter
```

9.2 Order-preserving encoding

Sort arbitrary data as byte strings

Our B+tree deals with string keys of arbitrary bytes. But a column can be of other types, such as numbers, and keys can be multiple columns. To support range queries, serialized keys must be compared w.r.t. their data type.

The obvious way is to replace `bytes.Compare` with a callback that decodes and compares keys according to the table schema.

Another way is to **choose a special serialization format so that the resulting bytes reflect the sort order**. This is the shortcut we'll take.

Numbers

Let's start with a simple problem: how to encode *unsigned integers* so that they can be compared by `bytes.Compare`? `bytes.Compare` works byte by byte until a difference is met. So the 1st byte is most significant in a comparison, if we put the most significant (higher) bits of an integer first, they can be compared byte-wise. That's just big-endian integers.

```
0x0000000000000001 -> 00 00 00 00 00 00 00 01
0x0000000000000002 -> 00 00 00 00 00 00 00 02
                   ...
0x00000000000000ff -> 00 00 00 00 00 00 00 ff
0x0000000000000100 -> 00 00 00 00 00 00 01 00
```

Next, we'll consider *signed integers*, which are represented by two's complement. In two's complement representation, the upper half of

unsigned values is simply offset to negative values. To ensure the correct order, the positive half is swapped with the negative half, which is just flipping the most significant bit.

```
var buf [8]byte
u := uint64(v.I64) + (1 << 63)        // flip the sign bit
binary.BigEndian.PutUint64(buf[:], u) // big endian
```

Some examples:

int64	Encoded bytes
MinInt64	00 00 00 00 00 00 00 00
-2	7f ff ff ff ff ff ff fe
-1	7f ff ff ff ff ff ff ff
0	80 00 00 00 00 00 00 00
1	80 00 00 00 00 00 00 01
MaxInt64	ff ff ff ff ff ff ff ff

So the general ideas are:

- Arranging bits so that more significant bits come first (big-endian).
- Remapping bits to unsigned integers in the correct order.

Exercise for the reader: Apply this to floats (sign + magnitude + exponent).

Strings

The key can be multiple columns. But `bytes.Compare` only works with a single string column, because it needs the length. We cannot simply concatenate string columns, because this creates ambiguity. E.g., `("a", "bc")` vs. `("ab", "c")`.

There are 2 ways to encode strings with lengths, one way is to prepend the length, this requires decoding. Another way is to put a delimiter at the end, such as a null byte. The previous example is encoded as `"a\x00bc\x00"` and `"ab\x00c\x00"`.

The problem with delimiters is that the input cannot contain the delimiter, this is solved by *escaping* the delimiter. We'll use byte 0x01 as the escaping byte, and the escaping byte itself must be escaped. So we'll need 2 transformations:

```
00 -> 01 01
01 -> 01 02
```

Note that the escape sequences still preserve the sort order.

Tuples

A multi-column comparison (tuple) is done column by column until a difference is met. This is like a string comparison, except that each item is a typed value instead of a byte. We can simply concatenate the encoded bytes of each column as long as there is no ambiguity.

9.3 Range query

`Scanner` is a wrapper of the B+tree iterator. It decodes KVs into rows.

```
// within the range or not?
func (sc *Scanner) Valid() bool
// move the underlying B-tree iterator
func (sc *Scanner) Next()
// fetch the current row
func (sc *Scanner) Deref(rec *Record)

func (db *DB) Scan(table string, req *Scanner) error
```

The input is an interval of the primary key.

```
type Scanner struct {
    // the range, from Key1 to Key2
    Cmp1 int // CMP_??
    Cmp2 int
    Key1 Record
    Key2 Record
    // ...
}
```

For open-ended intervals, simply set `Key2` to the maximum/minimum value.

9.4 What we learned

- B+tree iterators.
- Order-preserving encoding.

The next step is to add secondary indexes, which are just extra tables.

10. Secondary Indexes

10.1 Secondary indexes as extra keys

Table schema

As mentioned in chapter 08, secondary indexes are just extra KV pairs containing the primary key. Each index is distinguished by a key prefix in the B+tree.

```
type TableDef struct {
    // user defined
    Name    string
    Types   []uint32   // column types
    Cols    []string   // column names
    Indexes [][]string // the first index is the primary key
    // auto-assigned B-tree key prefixes for different tables and indexes
    Prefixes []uint32
}
```

The first index is used as the primary key as it's also an index.

KV structures

For a secondary index, we could put the primary key in B+tree value, which is used to find the full row. However, unlike the primary key, secondary indexes don't have the unique constraint, so there can be duplicate B+tree keys.

Instead of modifying our B+tree to support duplicates, we can also add the primary key to the B+tree key to make it unique and leave the value empty.

```
create table t1 (
    k1 string,
```

```
    k2 int,
    v1 string,
    v2 string,
    primary key (k1, k2),
    index idx1 (v1),
    index idx2 (v2, k2)
);
```

	key	value
t1	prefix1, k1, k2	v1, v2
idx1	prefix2, v1, k1, k2	(empty)
idx2	prefix3, v2, k2, k1	(empty)

10.2 Using secondary indexes

Select an index by matching columns

To do a range query, we must select an index that matches the query keys, which is stored in the `Scanner` type so that `Scanner.Deref()` can use it.

```
type Scanner struct {
    // the range, from Key1 to Key2
    Cmp1 int // CMP_??
    Cmp2 int
    Key1 Record
    Key2 Record
    // internal
    db     *DB
    tdef   *TableDef
    index  int    // which index?
    iter   *BIter // the underlying B-tree iterator
    keyEnd []byte // the encoded Key2
}
```

An index can be multiple columns. For example, index (a, b) can serve the query $(a, b) > (1, 2)$. It can also serve the query $a > 1$, because this is the same as $(a, b) > (1, +\infty)$. The index selection is just matching the columns.

```
func dbScan(db *DB, tdef *TableDef, req *Scanner) error {
    // ...
    isCovered := func(index []string) bool {
        key := req.Key1.Cols
        return len(index) >= len(key) && slices.Equal(index[:len(key)], key)
    }
    req.index = slices.IndexFunc(tdef.Indexes, isCovered)
    // ...
}
```

Encode missing columns as infinity

In the last example, the query $a > 1$ with the index (a, b) uses only 1 of the columns, so we need to encode the rest as infinity.

Input query	Using the index
$a > 1$	$(a, b) > (1, +\infty)$
$a \le 1$	$(a, b) < (1, +\infty)$
$a \ge 1$	$(a, b) > (1, -\infty)$
$a < 1$	$(a, b) < (1, -\infty)$

This can be done by modifying our order-preserving encoding. First, we'll choose `"\xff"` as +∞ and `""` as -∞. As no columns are encoded as empty strings, we can just ignore the missing columns in the -∞ cases. In the +∞ cases, we'll prepend a tag to each encoded column so that they don't start with `"\xff"`.

```
// order-preserving encoding
func encodeValues(out []byte, vals []Value) []byte {
    for _, v := range vals {
        out = append(out, byte(v.Type)) // *added*: doesn't start with 0xff
        switch v.Type {
        case TYPE_INT64:
            var buf [8]byte
            u := uint64(v.I64) + (1 << 63)        // flip the sign bit
            binary.BigEndian.PutUint64(buf[:], u) // big endian
            out = append(out, buf[:]...)
        case TYPE_BYTES:
            out = append(out, escapeString(v.Str)...)
            out = append(out, 0) // null-terminated
        default:
            panic("what?")
        }
    }
    return out
}
```

We'll prepend the column type code as the tag. This also makes debugging easier since we can now decode stuff by looking at the hexdump.

This is a small extra step to support range queries on prefix columns.

```
// for primary keys and indexes
func encodeKey(out []byte, prefix uint32, vals []Value) []byte {
    // 4-byte table prefix
    var buf [4]byte
    binary.BigEndian.PutUint32(buf[:], prefix)
    out = append(out, buf[:]...)
    // order-preserving encoded keys
    out = encodeValues(out, vals)
    return out
}

// for the input range, which can be a prefix of the index key.
func encodeKeyPartial(
    out []byte, prefix uint32, vals []Value, cmp int,
) []byte {
    out = encodeKey(out, prefix, vals)
    if cmp == CMP_GT || cmp == CMP_LE { // encode missing columns as infinity
        out = append(out, 0xff) // unreachable +infinity
    } // else: -infinity is the empty string
    return out
}
```

10.3 Maintaining secondary indexes

Sync with the primary data

An update may involve multiple B+tree keys with secondary indexes. When a row is changed, we must remove old index keys and insert new ones. To do this, the B+tree interface is extended to return the old value.

```
type UpdateReq struct {
    tree *BTree
    // out
    Added   bool   // added a new key
    Updated bool   // added a new key or an old key was changed
    Old     []byte // the value before the update
    // in
    Key  []byte
    Val  []byte
    Mode int
}
```

Use the new information:

```
func dbUpdate(db *DB, tdef *TableDef, rec Record, mode int) (bool, error) {
    // ...
    // insert the row
    req := UpdateReq{Key: key, Val: val, Mode: mode}
    if _, err = db.kv.Update(&req); err != nil {
        return false, err
    }
    // maintain secondary indexes
    if req.Updated && !req.Added {
        // use `req.Old` to delete the old indexed keys ...
    }
    if req.Updated {
        // add the new indexed keys ...
```

```
    }
    return req.Updated, nil
}
```

Atomicity of multi-key updates

Atomicity is not composable! We lost atomicity when multiple keys are involved, even if individual KV operations are atomic. If the DB crashed or an error occurred while updating a secondary index, it should revert to the previous state.

Achieving this with just get, set, del is tricky, which is why simple KV interfaces are very limiting. Our next step is a *transactional* KV interface to allow atomic operations on multiple keys or even concurrent readers.

10.4 Summary of tables and indexes on KV

- Rows and columns as KVs.
- Range queries.
  - B+tree iterators.
  - Order-preserving encoding.
- Secondary indexes.
  - Index selection.
  - A transactional interface is needed.

11. Atomic Transactions

11.1 The all-or-nothing effect

The secondary indexes from the last chapter require atomic multi-key updates. This is not only necessary for internal DB consistency, but also useful for application data consistency, think of account balances vs. account transactions.

We'll drop the get-set-del interface and add a new one to allow atomic execution of a group of operations. Concurrency is discussed in the next chapter.

Commit and rollback

We'll add interfaces to mark the beginning and end of the transaction. At the end, updates either take effect (commit) or are discarded (rollback) due to errors or user request (`Abort`).

```
// begin a transaction
func (kv *KV) Begin(tx *KVTX)
// end a transaction: commit updates; rollback on error
func (kv *KV) Commit(tx *KVTX) error
// end a transaction: rollback
func (kv *KV) Abort(tx *KVTX)
```

Atomicity via copy-on-write

With copy-on-write, both commit and rollback are just updating the root pointer. This is already implemented as error handling in chapter 06.

```
type KVTX struct {
    db   *KV
```

```
    meta []byte // for the rollback
}

func (kv *KV) Begin(tx *KVTX) {
    tx.db = kv
    tx.meta = saveMeta(tx.db)
}
func (kv *KV) Commit(tx *KVTX) error {
    return updateOrRevert(tx.db, tx.meta)
}
func (kv *KV) Abort(tx *KVTX) {
    // nothing has written, just revert the in-memory states
    loadMeta(tx.db, tx.meta)
    // discard temporaries
    tx.db.page.nappend = 0
    tx.db.page.updates = map[uint64][]byte{}
}
```

Previously, `updateOrRevert()` was called after a single key update. Now it's moved to `KVTX.Commit()`. The B+tree can be updated as many times as needed, it's the root pointer that matters.

```
// previous chapter!!!
func (db *KV) Update(req *UpdateReq) (bool, error) {
    meta := saveMeta(db)
    if !db.tree.Update(req) {
        return false, nil
    }
    err := updateOrRevert(db, meta)
    return err == nil, err
}
```

Alternative: atomicity via logging

In a copy-on-write tree, updates are captured by the root pointer, as opposed to in-place updates where a log is required to capture updates.

The log is used to rollback updates if a transaction is aborted. The problem is that IO errors prevent further updates, so rollback is left to the recovery mechanism, this is also true with copy-on-write (see `updateOrRevert`).

Updates are considered durable once `fsync`'ed in the log. So the DB can return success to the client after only 1 `fsync`, as long as the log is considered for queries and eventually merged into the main datastore.

11.2 Transactional interfaces

Move tree operations to transactions

Tree operations are now associated with a transaction, so they are moved to `KVTX`.

```
func (tx *KVTX) Seek(key []byte, cmp int) *BIter {
    return tx.db.tree.Seek(key, cmp)
}
func (tx *KVTX) Update(req *UpdateReq) bool {
    return tx.db.tree.Update(req)
}
func (tx *KVTX) Del(req *DeleteReq) bool {
    return tx.db.tree.Delete(req)
}
```

Note that these functions no longer return errors because the actual disk update is moved to `KVTX.Commit()`.

Transactional table operations

For the table-based interfaces, just add a wrapper type to `KVTX`.

```
type DBTX struct {
    kv KVTX
    db *DB
}

func (db *DB) Begin(tx *DBTX)
func (db *DB) Commit(tx *DBTX) error
func (db *DB) Abort(tx *DBTX)
```

And move the table operations to that wrapper.

```
func (tx *DBTX) Scan(table string, req *Scanner) error
func (tx *DBTX) Set(table string, rec Record, mode int) (bool, error)
func (tx *DBTX) Delete(table string, rec Record) (bool, error)
```

These operations no longer deal with IO errors, so there is no error handling for updating secondary indexes.

11.3 Optional optimizations

A working relational DB is a major milestone, although it only supports sequential operations. For further challenges, there are some optimizations to consider.

Reduce copying on multi-key updates

Copy-on-write copies nodes from leaf to root in a single update, this is suboptimal for multi-key updates because nodes in intermediate trees are allocated, updated once, and then deleted within a transaction.

The optimization is to copy a node only once within a transaction and use in-place updates on copied nodes.

Range delete

Although we can now do multi-key updates. Deleting a large number of keys, such as dropping a table, is still problematic w.r.t. resource usage. The naive approach to dropping a table is to iterate and delete keys one by one. This reads the entire table into memory and does useless work as nodes are updated repeatedly before being deleted.

Some DBs use separate files for each table, so this is not a problem. In our case, a single B+tree is used for everything, so we can implement a *range delete* operation that frees all leaf nodes with a range without even looking at them.

Compress common prefixes

In any sorted data, nearby keys are likely to share a common prefix. And in typical relational DB usages, multi-column keys also result in shared prefixes. So there’s an opportunity to compress keys within a node.

Prefix compression makes implementation more difficult (fun), especially when the node size is not easily predictable for merge and split.

12. Concurrency Control

12.1 Levels of concurrency

The problem: interleaved readers and writers

Concurrent clients can enter and exit transactions at will, requesting reads and writes in between. To simplify the analysis, let's assume that enter/exit/read/write are atomic steps, so that concurrent TXs are just interleaved steps.

We'll also distinguish read-only TXs from read-write TXs. This is because …

- Concurrent readers are a much easier problem than concurrent writers.
- Many apps are read-heavy, where read performance is more important.

Readers-writer lock (RWLock)

Without knowing how to do concurrency, you can always add a mutex (lock) to serialize any data access. For read performance, you can use a *readers-writer lock* instead. It allows multiple concurrent readers, but only a single writer.

- If there's no writer, nothing can be changed, concurrent readers are OK.
- When a writer wants to enter, it waits until all readers have left.
- Readers are blocked by a writer, but not by other readers.

The usefulness of this is limited; there is no concurrency between writers, and long-running TXs are bad because readers and writers block each other.

Read-copy-update (RCU)

To prevent readers and writers from blocking each other, we can make readers and writers work on their own version of the data.

- There is a pointer to the *immutable* data, readers just grab it as a snapshot.
- A single writer updates its own *copy*, then flips the pointer to it.

We get this level of concurrency for free since we are copy-on-write. But a single writer is still insufficient because the lifetime of a TX is controlled by the client, which can be arbitrarily long.

Optimistic concurrency control

Concurrent writers lead to conflicts, for example:

Seq	TX1	TX2
1	read `a`	
2		write `a := 1`
3	write `b := a`	
4	commit	
5		commit

TX1 depends on the same key that TX2 modifies, so they cannot both succeed.

Note that some seemingly “write-only” operations actually have a read dependency. For example, our update/delete interface reports whether the key is updated/deleted, which depends on the previous state of the key. So the following scenario is also a conflict.

Seq	TX1	TX2
1	write `a := 1`	
2		delete `a`
3	commit	
4		commit

One way to deal with conflicts is to just abort TX when a conflict is detected.

1. TX starts.
2. Reads are on the snapshot, but writes are buffered locally.
3. Before committing, verify that there are no conflicts with committed TXs.
4. TX ends.
    - If there's a conflict, abort and rollback.
    - Otherwise, transfer buffered writes to the DB.

Note that *verify and commit* is an atomic step. This is called *optimistic* concurrency control, because it assumes conflicts are rare and does nothing to prevent them. We'll implement this, but there are alternatives to know about.

Alternative: pessimistic concurrency control

With optimistic concurrency control, TXs cannot progress in case of a conflict, which isn't very helpful from the application's PoV because all they can do is retry in a loop. Another way to deal with conflicts is to prevent them via locking. TXs will acquire locks on their dependencies so that potentially conflicting TXs will wait for each other.

This sounds much nicer, especially in the last example where write/delete can progress without problems. However, this still doesn’t guarantee progress because TXs can now fail with *deadlocks*.

A deadlock is when 2 parties are waiting for each other to release a (different) lock that they own. This also happens for more than 2 parties as long as there is a cycle in the dependency graph. In concurrent programming, locks should be acquired in a predefined order to avoid cycles. This isn’t the case for DBs as the client can grab locks in any order, so a DB must detect and resolve deadlocks, which is a graph problem.

Comparison of concurrency controls

	reader-reader	**reader-writer**	**writer-writer**	**conflict**
RWLock	pass	block	block	-
RCU	pass	pass	block	-
Optimistic	pass	pass	pass	abort
Pessimistic	pass	lock	lock	prevent

12.2 Snapshot isolation for readers

Isolation level refers to how a TX sees changes from other TXs. This is not an issue with copy-on-write since a TX operates on a *snapshot* of the B+tree.

Capture local updates

A transaction keeps a snapshot of the DB and local updates.

- The snapshot is just a root pointer with copy-on-write.
- The local updates are held in an in-memory B+tree.

```
type KVTX struct {
    // a read-only snapshot
    snapshot BTree
    // captured KV updates:
    // values are prefixed by a 1-byte flag to indicate deleted keys.
    pending BTree
    // ...
}
```

Both trees are initialized at the start of the TX.

```
// begin a transaction
func (kv *KV) Begin(tx *KVTX) {
    // read-only snapshot, just the tree root and the page read callback
    tx.snapshot.root = kv.tree.root
    tx.snapshot.get = ... // read from mmap'ed pages ...
    // in-memory tree to capture updates
    pages := [][]byte(nil)
    tx.pending.get = func(ptr uint64) []byte { return pages[ptr-1] }
    tx.pending.new = func(node []byte) uint64 {
        pages = append(pages, node)
```

```
        return uint64(len(pages))
    }
    tx.pending.del = func(uint64) {}
}
```

To represent deleted keys, values in `KVTX.pending` are prefixed by a 1-byte flag.

```
    FLAG_DELETED = byte(1)
    FLAG_UPDATED = byte(2)
```

Read back your own write

Within a TX, the client should be able to read back what it has just written, even if it has not been committed, so queries should consult `KVTX.pending` before `KVTX.snapshot`. That's why writes are held in a B+tree instead of just a list.

```
// point query. combines captured updates with the snapshot
func (tx *KVTX) Get(key []byte) ([]byte, bool) {
    val, ok := tx.pending.Get(key)
    switch {
    case ok && val[0] == FLAG_UPDATED: // updated in this TX
        return val[1:], true
    case ok && val[0] == FLAG_DELETED: // deleted in this TX
        return nil, false
    case !ok: // read from the snapshot
        return tx.snapshot.Get(key)
    default:
        panic("unreachable")
    }
}
```

For range queries, a new iterator type is added to combine both trees.

```
// an iterator that combines pending updates and the snapshot
type CombinedIter struct {
```

```
    top *BIter // KVTX.pending
    bot *BIter // KVTX.snapshot
    // ...
}
```

Version numbers in the free list

Since readers can hold old versions of the DB, the free list cannot give away pages from those versions. We'll solve this by assigning a monotonically increasing version number to each version. This is also called a *timestamp* (logically).

- We keep track of ongoing TXs and the version numbers they're based on.
- Each page added to the free list is associated with the version number.
- The list never gives away pages that are newer than the oldest TX.

This works by checking the version when consuming from the list head. Remember that the free list is a FILO (first-in-last-out), so pages from the oldest version will be consumed first.

Modification 1: Version numbers in **KVTX** and **KV**.

```
type KVTX struct {
    // a read-only snapshot
    snapshot BTree
    version  uint64 // based on KV.version
    // ...
}
type KV struct {
    // ...
    version uint64    // monotonic version number; persisted in the meta page
    ongoing []uint64  // version numbers of concurrent TXs
}
```

Modification 2: Free list augmentation.

```
// | next | pointer + version | unused |
// |  8B  |     n*(8B+8B)     |   ...  |
type FreeList struct {
    // ...
    maxSeq uint64 // saved `tailSeq` to prevent consuming newly added items
    maxVer uint64 // the oldest reader version
    curVer uint64 // version number when committing
}
```

- **`maxVer`** is maintained as the oldest version in **`KV.ongoing`** when a TX exits. It prevents page reuse in addition to the existing **`maxSeq`**.
- **`curVar`** is set to the next version by the writer on commit.

12.3 Handle conflicts for writers

Detect conflicts with history

Modification 1: All reads are added to **`KVTX.reads`** (point and range queries).

```
// start <= key <= stop
type KeyRange struct {
    start []byte
    stop  []byte
}
type KVTX struct {
    // ...
    reads []KeyRange
}
func (tx *KVTX) Get(key []byte) ([]byte, bool) {
    tx.reads = append(tx.reads, KeyRange{key, key}) // dependency
    // ...
}
```

Modification 2: Each successful commit is added to **`KV.history`**.

```
type KV struct {
    // ...
    history []CommittedTX // chanages keys; for detecting conflicts
}
type CommittedTX struct {
    version uint64
    writes  []KeyRange // sorted
}
func (kv *KV) Commit(tx *KVTX) error {
    // ...
    if len(writes) > 0 {
        kv.history = append(kv.history, CommittedTX{kv.version, writes})
    }
```

```
    return nil
}
```

Conflict detection works by checking for overlaps between its dependency and the history that are newer than its base version.

```
func detectConflicts(kv *KV, tx *KVTX) bool {
    for i := len(kv.history) - 1; i >= 0; i-- {
        if !versionBefore(tx.version, kv.history[i].version) {
            break // sorted
        }
        if rangesOverlap(tx.reads, kv.history[i].writes) {
            return true
        }
    }
    return false
}
```

The history is trimmed when the oldest TX exits.

Serialize internal data structures

In the analysis, TXs are simplified as interleaved steps, in reality, these steps can run in parallel threads, which should be serialized as they share the `KV` structure.

```
type KV struct {
    // ...
    mutex   sync.Mutex    // serialize TX methods
}
func (kv *KV) Begin(tx *KVTX) {
    kv.mutex.Lock()
    defer kv.mutex.Unlock()
    // ...
}
func (kv *KV) Commit(tx *KVTX)  // same
func (kv *KV) Abort(tx *KVTX)   // same
```

We can use this lock for all `KVTX` methods. But there are ways to reduce the locking. For example, we don't have to serialize read/write methods because …

- Writes only work on `KVTX.pending`, they never touch `KV`.
- Reads only touch `KV.mmap.chunks`, which are the slices returned by `mmap`.

The commit step may modify `KV.mmap.chunks` by appending, so we'll use a local copy for each TX. This slice is append-only, so a shallow copy is enough.

```
func (kv *KV) Begin(tx *KVTX) {
    kv.mutex.Lock()
    defer kv.mutex.Unlock()
    // read-only snapshot, just the tree root and the page read callback
    tx.snapshot.root = kv.tree.root
    chunks := kv.mmap.chunks // copied to avoid updates from writers
    tx.snapshot.get = func(ptr uint64) []byte { return mmapRead(ptr, chunks) }
    // ...
}
```

This way, read/write methods do not require the lock and can run in parallel. That's good, because reads can trigger page faults and block the thread.

So far, only `Begin`, `Commit`, and `Abort` are serialized. But considering that `Commit` involves IO, we can go further by releasing the lock while waiting for IO to allow other TXs to enter and read-only TXs to exit. The commit step should still be serialized with other commit steps via another lock. This is left as an exercise.

13. SQL Parser

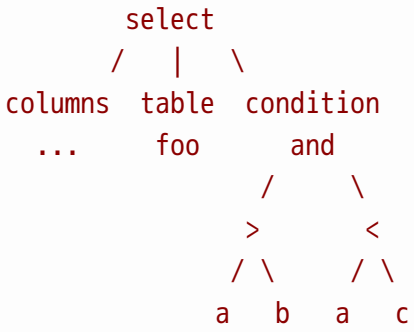
SQL is easily parsed by computers while still looking like English.

13.1 Syntax, parser, and interpreter

Tree representation of computer languages

A query language is a string parsed into a tree structure for further processing.

Example 1: `SELECT ... FROM foo WHERE a > b AND a < c`:



Example 2: Expression `a + b * c`:

SQL is just a particular syntax; there are easier alternatives, such as the pipeline-based PRQL, or even just S-expression.

The S-expression is just nested parentheses, it's the simplest syntax for arbitrary tree structures. You can skip this chapter if you choose S-expression, but SQL isn't much harder, because everything is handled with

nothing but top-down recursion. The lesson in this chapter also applies to most computer languages.

Evaluate by visiting tree nodes

Both `SELECT` and `UPDATE` can contain arithmetic expressions on columns, which are parsed into trees in the last example. Each tree node is an operator, and its subtrees are operands. To evaluate a tree node, first evaluate its subtrees.

```
# pseudo code
def eval(node):
    if is_binary_operator(node):
        left, right = eval(node.left), eval(node.right)
        return node.operator(left, right)
    elif is_value(node):
        return node.value
    ...
```

In retrospect, this is the reason why trees are relevant, because trees represent the evaluation order. A programming language also has control flows, variables, etc., but once you represent it with a tree, the rest should be obvious.

13.2 Query language specification

Statements

Not exactly SQL, just a look-alike.

```
create table table_name (
    a type1,
    b type2,
    ...
    index (c, b, a),
    index (d, e, f),
    primary key (a, b),
);

select expr... from table_name conditions limit x, y;
insert into table_name (cols...) values (a, b, c)...;
delete from table_name conditions limit x, y;
update table_name set a = expr, b = expr, ... conditions limit x, y;
```

Conditions

A SQL DB will choose an index based on the `WHERE` clause if possible, and/or fetch and filter the rows if the condition is not fully covered by the index. This is automatic without any direct user control.

Here we'll deviate from SQL: Instead of `WHERE`, we'll use separate clauses for indexing conditions and filtering conditions,

1. The `INDEX BY` clause selects an index and controls the sort order.

```
-- one of the 3 forms
select expr... from table_name index by a = 1;
```

```
select expr... from table_name index by a > 1;
select expr... from table_name index by a > 1 and a < 5;
-- the last query in descending order
select expr... from table_name index by a < 5 and a > 1;
```

2. The `FILTER` clause then filters the rows.

```
-- the filter condition can be arbitrary
select expr... from table_name index by condition1 filter condition2;
select expr... from table_name filter condition2;
```

Both are optional. And the primary key is selected if the `INDEX BY` is missing.

OLTP workloads often expect predictable performance; a sudden change in the query plan is a production hazard. That's why we make the index selection explicit to save the DB from guessing.

Expressions

An expression is either a …

- column name,
- literal value like numbers or strings,
- binary or unary operator,
- tuple.

```
a OR b
a AND b
NOT a
a = b, a < b, ...   -- comparisons
a + b, a - b
a * b, a / b
-a
```

Which is represented as a tree node.

```
type QLNode struct {
    Type uint32 // tagged union
    I64  int64
    Str  []byte
    Kids []QLNode // operands
}
```

Different operators have different priorities (precedence), as listed above. This complication is avoided in simpler grammars like S-expression. But operator precedence can be handled with simple recursion, as you'll see.

13.3 Recursive descent

Tree node structures

Each statement is divided into smaller parts including the expression node `QLNode`, so they are trees of components.

```
// statements: select, update, delete
type QLSelect struct {
    QLScan
    Names  []string // expr AS name
    Output []QLNode
}
type QLUpdate struct {
    QLScan
    Names  []string
    Values []QLNode
}
type QLDelete struct {
    QLScan
}
// common structure for statements: `INDEX BY`, `FILTER`, `LIMIT`
type QLScan struct {
    Table  string // table name
    Key1   QLNode // index by
    Key2   QLNode
    Filter QLNode // filter expression
    Offset int64  // limit
    Limit  int64
}
```

Split the input into smaller parts

All parsing is *top-down*. The topmost part is a statement, we'll first determine its type, then dispatch the work to the concrete function.

```
func pStmt(p *Parser) (r interface{}) {
    switch {
    case pKeyword(p, "create", "table"):
        r = pCreateTable(p)
    case pKeyword(p, "select"):
        r = pSelect(p)
    // ...
    }
    return r
}
```

pKeyword matches and consumes keywords from the input to determine the next part. Take a look at **pSelect**, its 3 parts are consumed by 3 functions.

```
func pSelect(p *Parser) *QLSelect {
    stmt := QLSelect{}
    pSelectExprList(p, &stmt)   // SELECT xxx
    pExpect(p, "from", "expect `FROM` table")
    stmt.Table = pMustSym(p)    // FROM table
    pScan(p, &stmt.QLScan)      // INDEX BY xxx FILTER yyy LIMIT zzz
    return &stmt
}
```

pSelectExprList is the comma-separated expression list. Each list item is dispatched to **pSelectExpr**. The comma determines where the list ends.

```
func pSelectExprList(p *Parser, node *QLSelect) {
    pSelectExpr(p, node)
    for pKeyword(p, ",") {
        pSelectExpr(p, node)
    }
}
```

pScan is the last part of **SELECT**. It's further divided into 3 smaller parts.

```
func pScan(p *Parser, node *QLScan) {
    if pKeyword(p, "index", "by") {
        pIndexBy(p, node)
    }
    if pKeyword(p, "filter") {
        pExprOr(p, &node.Filter)
    }
    node.Offset, node.Limit = 0, math.MaxInt64
    if pKeyword(p, "limit") {
        pLimit(p, node)
    }
}
```

Without looking at every function, we've already got the idea of parsing:

1. Split the input into smaller and smaller parts until it ends as either an operator, a name, or a literal value.
2. Determine the next part by looking at the next keywords.

`SELECT` decomposed into smaller and smaller parts.

`SELELT a, b`	`FROM`	`foo`	`INDEX BY x`	`FILTER y`	`LIMIT z`
`pSelectExprList`	`pExpect`	`pMustSym`	`pScan`		
`pSelectExpr`			`pIndexBy`	`pExprOr`	`pLimit`
`pExprOr`			...	...	`pNum`
...					

Convert infix operators into a binary tree

`pExprOr` parses an arbitrary expression. Turning `1+2*3-4` into a tree regarding operator precedence is not obvious, as it's just an interleaved list of numbers and operators. So let's start with a simpler problem: only the `+` operator.

```
term
term + term
```

```
term + term + term + ...
```

The expression `left + right` is represented as:

```
    +
   / \
left right
```

The *left* subtree can also represent an expression, so `LL + LR + R` becomes:

```
    +
   / \
  +   R
 / \
LL LR
```

We can add more terms, and it's still a binary tree. Pseudo-code:

```
def parse_terms():
    node = parse_column()
    while consume('+'):
        right = parse_column()
        node = QLNode(type='+', kids=[node, right])
    return node
```

This is described by a simple rule:

```
expr := expr + term
expr := term
```

The *left* subrule `expr` can expand to the rule itself, but the right subrule `term` is the bottommost part that cannot expand any further.

Operator precedence with recursion

Next problem: add the `*` operator. It has a higher priority, so `term` is now expanded by a similar rule, and the bottommost part is now `factor`.

```
expr := expr + term
expr := term
term := term * factor
term := factor
```

Pseudo-code:

```
def parse_terms():
    node = parse_factors()
    while consume('+'):
        right = parse_factors()
        node = QLNode(type='+', kids=[node, right])
    return node

def parse_factors():
    node = parse_column()
    while consume('*'):
        right = parse_column()
        node = QLNode(type='*', kids=[node, right])
    return node
```

Visualization of the 2-rule expansion.

	a	+	b	×	c	-	d
1	term	+		term		-	term
2	factor		factor	×	factor		factor

The `OR` operator has the lowest priority, so `pExprOr` is the topmost function for parsing an expression. It calls `pExprAnd` to handle the next priority, all the way down to the highest priority `pExprUnop`, which calls `pExprAtom` to parse a bottommost part (a name or a literal value).

```
a OR b          -- pExprOr
a AND b         -- pExprAnd
NOT a           -- pExprNot
a = b, a < b    -- pExprCmp
a + b, a - b    -- pExprAdd
```

```
a * b, a / b    -- pExprMul
-a              -- pExprUnop
```

This is called *recursive descent*. In retrospect, it's just divide and conquer, where the "divide" is just checking for the next keyword.

14. Query Language

14.1 Expression evaluation

Both **SELECT** and **UPDATE** contain expressions on columns that need evaluation.

```
type QLEvalContex struct {
    env Record // input row values
    out Value  // output
    err error
}
```

Evaluating a tree is as obvious as it was discussed in the previous chapter.

```
func qlEval(ctx *QLEvalContex, node QLNode) {
    switch node.Type {
    // refer to a column
    case QL_SYM:
        if v := ctx.env.Get(string(node.Str)); v != nil {
            ctx.out = *v
        } else {
            qlErr(ctx, "unknown column: %s", node.Str)
        }
    // a literal value
    case QL_I64, QL_STR:
        ctx.out = node.Value
    // operators
    case QL_NEG:
        qlEval(ctx, node.Kids[0])
        if ctx.out.Type == TYPE_INT64 {
            ctx.out.I64 = -ctx.out.I64
        } else {
            qlErr(ctx, "QL_NEG type error")
        }
    // ...
    }
}
```

`INSERT` contains expressions on constants that are evaluated with an empty `env`.

14.2 Range queries

Set up a range query

Both **`SELECT`**, **`UPDATE`**, and **`DELETE`** can do range queries, the difference is what to do with the results. **`QLScan`** is the common part that represents a range query.

```
type QLScan struct {
    Table  string // table name
    Key1   QLNode // index by
    Key2   QLNode
    Filter QLNode // filter
    Offset int64  // limit
    Limit  int64
}
```

It has 3 phases: **`INDEX BY`**, **`LIMIT`**, and **`FILTER`**. **`Scanner`** implements the **`INDEX BY`**.

```
func qlScanInit(req *QLScan, sc *Scanner) (err error) {
    // convert `QLNode` to `Record` and `CMP_??`
    if sc.Key1, sc.Cmp1, err = qlEvalScanKey(req.Key1); err != nil {
        return err
    }
    if sc.Key2, sc.Cmp2, err = qlEvalScanKey(req.Key2); err != nil {
        return err
    }
    switch { // special handling when `Key1` and `Key2` are not both present
    case req.Key1.Type == 0 && req.Key2.Type == 0: // no `INDEX BY`
        sc.Cmp1, sc.Cmp2 = CMP_GE, CMP_LE // full table scan
    case req.Key1.Type == QL_CMP_EQ && req.Key2.Type == 0:
        // equal by a prefix: INDEX BY key = val
        sc.Key2 = sc.Key1
        sc.Cmp1, sc.Cmp2 = CMP_GE, CMP_LE
    case req.Key1.Type != 0 && req.Key2.Type == 0:
        // open-ended range: INDEX BY key > val
        if sc.Cmp1 > 0 {
```

```
            sc.Cmp2 = CMP_LE // compare with a zero-length tuple
        } else {
            sc.Cmp2 = CMP_GE
        }
    }
    return nil
}
```

Revisit the infinity encoding

`INDEX BY` takes 1 of the 3 forms defined in the last chapter.

1. `a > start AND a < end`: An interval of $(start, end)$.
2. `a > s`: An open-ended interval of $(s, +\infty)$.
3. `a = p`: A prefix of the index.

Let's say the index is (a, b). Queries using a prefix of the index are already handled by the key encoding in chapter 10. So …

- `a = p` is equivalent to `a >= p AND a <= p`, encoded as $(a, b) \geq (p, -\infty)$ and $(a, b) \leq (p, +\infty)$.
- `a > s` is equivalent to `a > s AND () <= ()`, encoded as $(a, b) > (s, -\infty)$ and $(a,) < (+\infty,)$.

Since the use of the empty tuple `()`, `Key1` and `Key2` can now have a different set of columns, so we have to modify the index selection to allow this.

```
func dbScan(tx *DBTX, tdef *TableDef, req *Scanner) error {
    // ...
    covered := func(key []string, index []string) bool {
        return len(index) >= len(key) && slices.Equal(index[:len(key)], key)
    }
    req.index = slices.IndexFunc(tdef.Indexes, func(index []string) bool {
        return covered(req.Key1.Cols, index) && covered(req.Key2.Cols, index)
    })
```

```
    // ...
}
```

14.3 Results iterator

Iterators all the way down

The next phases are `LIMIT` and `FILTER`. The results are consumed from iterators.

```
type RecordIter interface {
    Valid() bool
    Next()
    Deref(*Record) error
}
```

Why use iterators instead of an array of results? Because a DB can potentially work with larger-than-memory data, an iterator doesn't require the results to be ready in memory at once, it can even stream the results as they're produced.

A chain of iterators for a `SELECT`.

Iterator	Out	Role
`BIter`	KV	Iterate through the B+tree.
`KVIter`	KV	Combine the snapshot with local updates.
`Scanner`	Row	Decode records and follow secondary indexes.
`qlScanIter`	Row	Offset, limit, and filter rows.
`qlSelectIter`	Row	Evaluate expressions in `SELECT`.

Transform data with iterators

An iterator takes another iterator as input to transform a stream of items. This is a useful programming pattern.

```
type qlSelectIter struct {
    iter  RecordIter    // input
    names []string
    exprs []QLNode
}

func (iter *qlSelectIter) Valid() bool {
    return iter.iter.Valid()
}
func (iter *qlSelectIter) Next() {
    iter.iter.Next()
}
func (iter *qlSelectIter) Deref(rec *Record) error {
    if err := iter.iter.Deref(rec); err != nil {
        return err
    }
    vals, err := qlEvelMulti(*rec, iter.exprs)
    if err != nil {
        return err
    }
    *rec = Record{iter.names, vals}
    return nil
}
```

qlScanIter is a bit more involved, as some bookkeeping is required for filtering.

```
type qlScanIter struct {
    // input
    req *QLScan
    sc  Scanner
    // state
    idx int64
    end bool
    // cached output item
    rec Record
```

```
    err error
}
```

14.4 Conclusions and next steps

We have multiple interfaces to a persistent, transactional DB:

1. A KV that can be embedded in applications.
2. A relational DB that can be embedded in applications.
3. A SQL-like query language for the relational DB.

Without adding new functionality, we can create a *network protocol* to allow the DB to run in a different process or machine. Network programming in Go is high-level and easy, but you can always learn more with a "from scratch" attitude, which you can find in the "Build Your Own Redis" book.

Since we have a basic parser and interpreter, we can take on compilers. You can create a programming language and compile to machine code instead of merely interpreting it. See the "From Source Code To Machine Code" book.