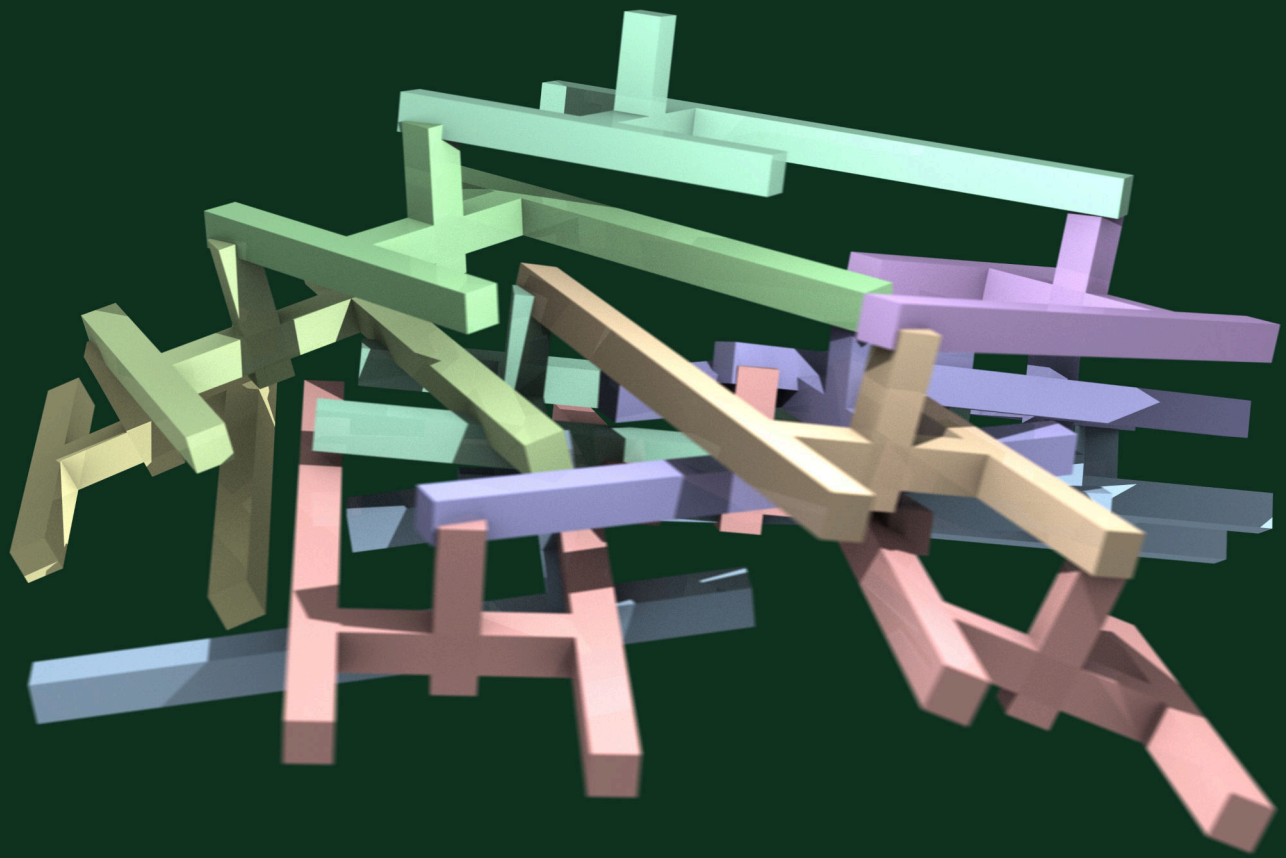
Build Your Own Database From Scratch

Persistence, Indexing, Concurrency



https://build-your-own.org

Build Your Own Database From Scratch

Persistence, Indexing, Concurrency

James Smith

build-your-own.org

2023-08-24

Contents

00. Introduction

0.1 What is This Book About?

Databases are not black boxes. Understand them by building your own from scratch!

This book contains a walk-through of a minimal persistent database implementation. The implementation is incremental. We start with a B-Tree, then a simple KV store, and eventually end with a mini relational DB.

The book focuses on important ideas rather than implementation details. Real-world databases are complex and harder to grasp. We can learn faster and easier from a stripped-down version of a database. And the "from scratch" method forces you to learn deeper.

Although the book is short and the implementation is minimal, it aims to cover three important topics:

1. **Persistence**. How not to lose or corrupt your data. Recovering from a crash.
2. **Indexing**. Efficiently querying and manipulating your data. (B-tree).
3. **Concurrency**. How to handle multiple (large number of) clients. And transactions.

If you have only vague ideas like "databases store my data" or "indexes are fast", this book is for you.

0.2 How to Use This Book?

This book takes a step-by-step approach. Each step builds on the previous one and adds a new concept. The book uses Golang for sample code, but the topics are language agnostic. Readers are advised to code their own version of a database rather than just read the text.

The draft chapters can be accessed at the official website:

https://build-your-own.org

0.3 Topic One: Persistence

Why do we need databases? Why not dump the data directly into files? Our first topic is *persistence*.

Let's say your process crashed middle-way while writing to a file, or you lost power, what's the state of the file?

- Does the file just lose the last write?
- Or ends up with a half-written file?
- Or ends up in an even more corrupted state?

Any outcome is possible. Your data is not guaranteed to persist on a disk when you simply write to files. This is a concern of databases. And a database will recover to a usable state when started after an unexpected shutdown.

Can we achieve persistence without using a database? There is a way:

1. Write the whole updated dataset to a new file.
2. Call `fsync` on the new file.
3. Overwrite the old file by renaming the new file to the old file, which is guaranteed by the file systems to be atomic.

This is only acceptable when the dataset is tiny. A database like SQLite can do incremental updates.

0.4 Topic Two: Indexing

There are two distinct types of database queries: analytical (OLAP) and transactional (OLTP).

- Analytical (OLAP) queries typically involve a large amount of data, with aggregation, grouping, or join operations.
- In contrast, transactional (OLTP) queries usually only touch a small amount of indexed data. The most common types of queries are indexed point queries and indexed range queries.

Note that the word "transactional" is *not* related to database transactions as you may know. Computer jargon is often overloaded with different meanings. This book focuses only on OLTP techniques.

While many applications are not real-time systems, most user-facing software should respond in a reasonable (small) amount of time, using a reasonable amount of resources (memory, IO). This falls into the OLTP category. How do we find the data quickly (in `O(log(n))`), even if the dataset is large? This is why we need indexes.

If we ignore the persistence aspect and assume that the dataset fits in memory, finding the data quickly is the problem of data structures. Data structures that persist on a disk to look up data are called "indexes" in database systems. And database indexes can be larger than memory. There is a saying: if your problem fits in memory, it's an easy problem.

Common data structures for indexing include B-Trees and LSM-Trees.

0.5 Topic Three: Concurrency

Modern applications do not just do everything sequentially, nor do databases. There are different levels of concurrency:

- Concurrency between readers.
- Concurrency between readers and writers, do writers need exclusive access to the database?

Even the file-based SQLite supports some concurrency. But concurrency is easier within a process, which is why most database systems can only be accessed via a "server".

With the addition of concurrency, applications often need to do things atomically, such as the read-modify-write operation. This adds a new concept to databases: transactions.

Part I. Simple KV Store

Build a simple persistent key-value store using the B-tree.

01. Files vs. Databases

This chapter shows the limitations of simply dumping data to files and the problems that databases solve.

1.1 Persisting Data to Files

Let's say you have some data that needs to be persisted to a file; 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)
    return err
}
```

This naive approach has some drawbacks:

1. It truncates the file before updating it. What if the file needs to be read concurrently?
2. Writing data to files may not be atomic, depending on the size of the write. Concurrent readers might get incomplete data.
3. When is the data actually persisted to the disk? The data is probably still in the operating system's page cache after the **`write`** syscall returns. What's the state of the file when the system crashes and reboots?

1.2 Atomic Renaming

To address some of these problems, let's propose a better approach:

```
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 fp.Close()

    _, err = fp.Write(data)
    if err != nil {
        os.Remove(tmp)
        return err
    }

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

This approach is slightly more sophisticated, it first dumps the data to a temporary file, then **rename** the temporary file to the target file. This seems to be free of the non-atomic problem of updating a file directly — the **rename** operation is atomic. If the system crashed before renaming, the original file remains intact, and applications have no problem reading the file concurrently.

However, this is still problematic because it doesn't control when the data is persisted to the disk, and the metadata (the size of the file) may be persisted to the disk before the data, potentially corrupting the file after when the system crash. (You may have noticed that some log files have zeros in them after a power failure, that's a sign of file corruption.)

1.3 fsync

To fix the problem, we must flush the data to the disk before renaming it. The Linux syscall for this is "fsync".

```
func SaveData3(path string, data []byte) error {
    // code omitted...

    _, err = fp.Write(data)
    if err != nil {
```

```
        os.Remove(tmp)
        return err
    }

    err = fp.Sync() // fsync
    if err != nil {
        os.Remove(tmp)
        return err
    }

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

Are we done yet? The answer is no. We have flushed the data to the disk, but what about the metadata? Should we also call the **`fsync`** on the directory containing the file?

This rabbit hole is quite deep and that's why databases are preferred over files for persisting data to the disk.

1.4 Append-Only Logs

In some use cases, it makes sense to persist data using an append-only log.

```
func LogCreate(path string) (*os.File, error) {
    return os.OpenFile(path, os.O_RDWR|os.O_CREATE, 0664)
}

func LogAppend(fp *os.File, line string) error {
    buf := []byte(line)
    buf = append(buf, '\n')
    _, err := fp.Write(buf)
    if err != nil {
        return err
    }
    return fp.Sync() // fsync
}
```

The nice thing about the append-only log is that it does not modify the existing data, nor

does it deal with the `rename` operation, making it more resistant to corruption. But logs alone are not enough to build a database.

1. A database uses additional "indexes" to query the data efficiently. There are only brute-force ways to query a bunch of records of arbitrary order.
2. How do logs handle deleted data? They cannot grow forever.

We have already seen some of the problems we must handle. Let's start with indexing first in the next chapter.

02. Indexing

2.1 Key-Value Store and Relational DB

Although a relational DB supports many types of queries, almost all queries can be broken down into three types of disk operations:

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).

Database indexes are mostly about range queries and point queries, and it's easy to see that a range query is just a superset of point queries. If we extract the functionality of the database indexes, it is trivial to make a key-value store. But the point is that a database system can be built on top of a KV store.

We'll build a KV store before attempting the relational DB, but let's explore our options first.

2.2 Hashtables

Hashtables are the first to be ruled out when designing a general-purpose KV store. The main reason is sorting — many real-world applications do require sorting and ordering.

However, it is possible to use hashtables in specialized applications. A headache of using hashtables is the resizing operation. Naive resizing is `O(n)` and causes a sudden increase in disk space and IO. It's possible to resize a hashtable incrementally, but this adds complexity. Another problem with hashtables is when to resize down; hashtables generally don't shrink automatically to avoid frequent and costly resizing, at the cost of wasted disk space.

2.3 B-Trees

Balanced binary trees can be queried and updated in `O(log(n))` and can be range-queried. A B-tree is roughly a balanced n-ary tree. Why use an n-ary tree instead of a binary tree? There are several reasons:

1. Less space overhead.

Every leaf node in a binary tree is reached via a pointer from a parent node, and the parent node may also have a parent. On average, each leaf node requires 1~2 pointers.

This is in contrast to B-trees, where multiple data in a leaf node share one parent. And n-ary trees are also shorter. Less space is wasted on pointers.

2. Less disk IO.

    - B-trees are shorter, which means fewer disk seeks.
    - The minimum size of disk IOs is usually the size of the memory page (probably 4K). The operating system will fill the whole 4K page even if you read a smaller size. It's optimal if we make use of all the information in a 4K page (by choosing the node size of at least one page).

3. Faster in memory.

    Even when the data is cached in memory and disk IO is out of the equation, due to modern CPU memory caching and other factors, n-ary trees can be faster than binary trees even if their big-O complexity is the same.

We'll use B-trees in this book. But B-trees are not the only option.

2.4 LSM-Trees

Log-structured merge-tree. Here is a high-level overview of how LSM-Tree works.

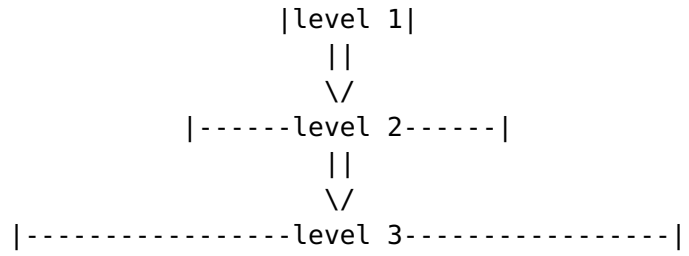
Let's start with 2 files: a small file holding the recent updates and a big file holding the rest of the data. Updates go to the smaller file first, but the file cannot grow forever, it has to be merged with the big file at some point to create a new, bigger file. Compare this to the dumb approach of overwriting the whole database when you update something, this is an improvement because it reduces writes.

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

And how do you query the database? You have to query both files, and the newer (smaller) file has higher priority. For point queries, you can query the small file first, and query the big file if it misses. For range queries, both files are queried simultaneously and the results are merged. Deletion is usually done by putting a mark in the small file to indicate that a key has been deleted. The actual deletion takes place when the files are merged.

Both files contain indexing data structures for queries. The advantage is that you can use simpler data structures because the files aren't updated in place, since the update operations are replaced by the merge operation. Each file can simply be a list of sorted KVs indexed by an array of pointers — easier and less error-prone to implement than B-trees.

Having 2 files is still not optimal regarding the amount of writes when merging files since the data in the big file is written to disk over and over again. Luckily, this idea can be generalized to more than 2 files, and each "file" is usually called a "level". Data goes into the 1st level first, and when the 1st level gets too big, the 1st level is merged into the 2nd level, and the 2nd level is now bigger. Each level is merged into the next bigger and older level when it gets too big.



Why does this scheme writes less than the 2-level scheme? Levels grow exponentially, the multiplier of excess disk write (called write amplification) is **`O(log(n))`** to the data size. For example, you can think of a list of files with exponentially increasing size by the power of two, then you double the size of the 1st file, now the size if the same as the 2nd file, merge it with the 2nd file and then merge it with the 3rd file and etc.

Real databases don't use the power of two ratio between levels because it creates too many levels, which hurts query performance. The size ratio between levels is usually tunable to allow tradeoffs between write amplification and query performance.

Also, real databases usually implement levels as multiple sorted and non-overlapping files instead of one big sorted file. Merges are performed in small parts, which allows for smoother operation. This also reduces the disk space requirements, otherwise, merging the last level would double the disk space usage.

Readers can try to use LSM-trees instead of B-trees after finishing this book. And compare the cons and pros between B-trees and LSM-trees.

03. B-Tree: The Ideas

3.1 The Intuitions of the B-Tree and BST

Our first intuition comes from balanced binary trees (BST). Binary trees are popular data structures for sorted data. Keeping a tree in good shape after inserting or removing keys is what "balancing" means. As stated in a previous chapter, n-ary trees should be used instead of binary trees to make use of the "page" (minimum unit of IO).

B-trees can be generalized from BSTs. Each node of a B-tree contains multiple keys and multiple links to its children. When looking up a key in a node, all keys are used to decide the next child node.

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

The balancing of a B-tree is different from a BST, popular BSTs like RB trees or AVL trees are balanced on the height of sub-trees (by rotation). While the height of all B-tree leaf nodes is the same, a B-tree is balanced by the size of the nodes:

- If a node is too large to fit on one page, it is split into two nodes. This will increase the size of the parent node and possibly increase the height of the tree if the root node was split.
- If a node is too small, try merging it with a sibling.

If you are familiar with RB trees, you may also be aware of 2-3 trees that can be easily generalized as B-trees.

3.2 B-tree and Nested Arrays

Even if you are not familiar with the 2-3 tree, you can still gain some intuition using nested arrays.

Let's start with a sorted array. Queries can be done by bisection. But, updating the array is `O(n)` which we need to tackle. Updating a big array is bad so we split it into smaller arrays. Let's say we split the array into `sqrt(n)` parts, and each part contains `sqrt(n)` keys on average.

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

To query a key, we must first determine which part contains the key, bisecting on the `sqrt(n)` parts is `O(log(n))`. After that, bisecting the key on the part is again `O(log(n))` — it's no worse than before. And updating is improved to `O(sqrt(n))`.

This is a 2-level sorted nested array, what if we add more levels? This is another intuition of the B-tree.

3.3 B-Tree Operations

Querying a B-tree is the same as querying a BST.

Updating a B-tree is more complicated. From now on we'll use a variant of B-tree called "B+ tree", the B+ tree stores values only in leaf nodes, and internal nodes contain only keys.

Key insertion starts at a leaf. A leaf is just a sorted list of keys. Inserting the key into the leaf is trivial. But, the insertion may cause the node size to exceed the page size. In this case, we need to split the leaf node into 2 nodes, each containing half of the keys, so that both leaf nodes fit into one page.

An internal node consists of:

1. A list of pointers to its children.
2. A list of keys paired with the pointer list. Each of the keys is the first key of the corresponding child.

After splitting a leaf node into 2 nodes. The parent node replaces the old pointer and key with the new pointers and keys. And the size of the node increases, which may trigger further splitting.

```
    parent              parent
   /  |  \     =>      /  | |  \
L1   L2   L6         L1  L3 L4  L6
```

After the root node is split, a new root node is added. This is how a B-tree grows.

```
                      new_root
                        / \
    root                N1 N2
   /  |  \     =>      /  | |  \
L1   L2   L6         L1  L3 L4  L6
```

Key deletion is the opposite of insertion. A node is never empty because a small node will be merged into either its left sibling or its right sibling.

And when a non-leaf root is reduced to a single key, the root can be replaced by its sole child. This is how a B-tree shrinks.

3.4 Immutable Data Structures

Immutable means never updating data in place. Some similar jargons are "append-only", "copy-on-write", and "persistent data structures" (*the word "persistent" has nothing to do with the "persistence" we talked about ealier*).

For example, when inserting a key into a leaf node, do not modify the node in place, instead, create a new node with all the keys from the to-be-updated node and the new key. Now the parent node must also be updated to point to the new node.

Likewise, the parent node is duplicated with the new pointer. Until we reach the root node, the entire path has been duplicated. This effectively creates a new version of the tree that coexists with the old version. The LSM-tree we mentioned before is also considered immutable.

There are several advantages of immutable data structures:

1. Avoid data corruption. Immutable data structures do not modify the existing data, they merely add new data, so the old version of data remains intact even if the update is interrupted.
2. Easy concurrency. Readers can operate concurrently with writers since readers can work on older versions unaffected.

Persistence and concurrency are covered in later chapters. For now, we'll code an immutable B+ tree first.

04. B-Tree: The Practice (Part I)

This chapter implements an immutable B+ tree in Golang. The implementation is minimal and thus is easy to follow.

4.1 The Node Format

Our B-tree will be persisted to the disk eventually, so we need to design the wire format for the B-tree nodes first. Without the format, we won't know the size of a node and when to split a node.

A node consists of:

1. A fixed-sized header containing the type of the node (leaf node or internal node) and the number of keys.
2. A list of pointers to the child nodes. (Used by internal nodes).
3. A list of offsets pointing to each key-value pair.
4. Packed KV pairs.

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

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

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

To keep things simple, both leaf nodes and internal nodes use the same format.

4.2 Data Types

Since we're going to dump our B-tree to the disk eventually, why not use an array of bytes as our in-memory data structure as well?

```
type BNode struct {
    data []byte // can be dumped to the disk
}

const (
```

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

And we can't use the in-memory pointers, the pointers are 64-bit integers referencing disk pages instead of in-memory nodes. We'll add some callbacks to abstract away this aspect so that our data structure code remains pure data structure code.

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

The page size is defined to be 4K bytes. A larger page size such as 8K or 16K also works.

We also add some constraints on the size of the keys and values. So that a node with a single KV pair always fits on a single page. If you need to support bigger keys or bigger values, you have to allocate extra pages for them and that adds complexity.

```
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)
}
```

4.3 Decoding the B-tree Node

Since a node is just an array of bytes, we'll add some helper functions to access its content.

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

```
// pointers
func (node BNode) getPtr(idx uint16) uint64 {
    assert(idx < node.nkeys())
    pos := HEADER + 8*idx
    return binary.LittleEndian.Uint64(node.data[pos:])
}
func (node BNode) setPtr(idx uint16, val uint64) {
    assert(idx < node.nkeys())
    pos := HEADER + 8*idx
    binary.LittleEndian.PutUint64(node.data[pos:], val)
}
```

Some details about the offset list:

- The offset is relative to the position of the first KV pair.
- The offset of the first KV pair is always zero, so it is not stored in the list.
- We store the offset to the end of the last KV pair in the offset list, which is used to determine the size of the node.

```
// 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.data[offsetPos(node, idx):])
}
func (node BNode) setOffset(idx uint16, offset uint16) {
    binary.LittleEndian.PutUint16(node.data[offsetPos(node, idx):], offset)
}
```

The offset list is used to locate the nth KV pair quickly.

```
// 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.data[pos:])
    return node.data[pos+4:][:klen]
}
func (node BNode) getVal(idx uint16) []byte {
    assert(idx < node.nkeys())
    pos := node.kvPos(idx)
    klen := binary.LittleEndian.Uint16(node.data[pos+0:])
    vlen := binary.LittleEndian.Uint16(node.data[pos+2:])
    return node.data[pos+4+klen:][:vlen]
}
```

And to determine the size of the node.

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

4.4 The B-Tree Insertion

The code is broken down into small steps.

Step 1: Look Up the Key

To insert a key into a leaf node, we need to look up its position in the sorted KV list.

```
// returns the first kid node whose range intersects the key. (kid[i] <= key)
// TODO: bisect
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 lookup works for both leaf nodes and internal nodes. Note that the first key is skipped for comparison, since it has already been compared from the parent node.

Step 2: Update Leaf Nodes

After looking up the position to insert, we need to create a copy of the node with the new key in it.

```
// 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)
    nodeAppendRange(new, old, 0, 0, idx)
    nodeAppendKV(new, idx, 0, key, val)
    nodeAppendRange(new, old, idx+1, idx, old.nkeys()-idx)
}
```

The **nodeAppendRange** function copies keys from an old node to a new node.

```
// copy multiple KVs into the position
func nodeAppendRange(
    new BNode, old BNode,
    dstNew uint16, srcOld uint16, n uint16,
) {
    assert(srcOld+n <= old.nkeys())
    assert(dstNew+n <= new.nkeys())
    if n == 0 {
        return
    }

    // pointers
    for i := uint16(0); i < n; i++ {
        new.setPtr(dstNew+i, old.getPtr(srcOld+i))
    }
    // offsets
    dstBegin := new.getOffset(dstNew)
    srcBegin := old.getOffset(srcOld)
    for i := uint16(1); i <= n; i++ { // NOTE: the range is [1, n]
        offset := dstBegin + old.getOffset(srcOld+i) - srcBegin
        new.setOffset(dstNew+i, offset)
    }
    // KVs
    begin := old.kvPos(srcOld)
    end := old.kvPos(srcOld + n)
    copy(new.data[new.kvPos(dstNew):], old.data[begin:end])
}
```

The **nodeAppendKV** function copies a KV pair to the new node.

```
// 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.data[pos+0:], uint16(len(key)))
    binary.LittleEndian.PutUint16(new.data[pos+2:], uint16(len(val)))
    copy(new.data[pos+4:], key)
    copy(new.data[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))))
}
```

Step 3: Recursive Insertion

The main function for inserting a key.

```
// insert a KV into a node, the result might be split into 2 nodes.
// 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
}
```

The **`leafUpdate`** function is similar to the **`leafInsert`** function.

Step 4: Handle Internal Nodes

Now comes the code for handling internal nodes.

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

Step 5: Split Big Nodes

Inserting keys into a node increases its size, causing it to exceed the page size. In this case, the node is split into multiple smaller nodes.

The maximum allowed key size and value size only guarantee that a single KV pair always fits on one page. In the worst case, the fat node is split into 3 nodes (one large KV pair in the middle).

```
// split a bigger-than-allowed node into two.
// the second 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.data = old.data[:BTREE_PAGE_SIZE]
        return 1, [3]BNode{old}
    }
    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.data = left.data[:BTREE_PAGE_SIZE]
        return 2, [3]BNode{left, right}
    }
    // the left node is still too large
    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}
}
```

Step 6: Update Internal Nodes

Inserting a key into a node can result in either 1, 2 or 3 nodes. The parent node must update itself accordingly. The code for updating an internal node is similar to that for updating a leaf node.

```
// replace a link with 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)
    }
    nodeAppendRange(new, old, idx+inc, idx+1, old.nkeys()-(idx+1))
}
```

We have finished the B-tree insertion. Deletion and the rest of the code will be introduced in the next chapter.

05. B-Tree: The Practice (Part II)

Following the previous chapter on B-tree implementation.

5.1 The B-Tree Deletion

Step 1: Delete From Leaf Nodes

The code for deleting a key from a leaf node is just like other **`nodeReplace*`** functions.

```
// remove a key from a leaf node
func leafDelete(new BNode, old BNode, idx uint16) {
    new.setHeader(BNODE_LEAF, old.nkeys()-1)
    nodeAppendRange(new, old, 0, 0, idx)
    nodeAppendRange(new, old, idx, idx+1, old.nkeys()-(idx+1))
}
```

Step 2: Recursive Deletion

The structure is similar to the insertion.

```
// delete a key from the tree
func treeDelete(tree *BTree, node BNode, key []byte) BNode {
    // where to find the key?
    idx := nodeLookupLE(node, key)
    // act depending on the node type
    switch node.btype() {
    case BNODE_LEAF:
        if !bytes.Equal(key, node.getKey(idx)) {
            return BNode{} // not found
        }
        // delete the key in the leaf
        new := BNode{data: make([]byte, BTREE_PAGE_SIZE)}
        leafDelete(new, node, idx)
        return new
```

```
    case BNODE_NODE:
        return nodeDelete(tree, node, idx, key)
    default:
        panic("bad node!")
    }
}
```

Step 3: Handle Internal Nodes

The difference is that we need to merge nodes instead of splitting nodes. A node may be merged into one of its left or right siblings. The **`nodeReplace*`** functions are for updating links.

```
// 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.data) == 0 {
        return BNode{} // not found
    }
    tree.del(kptr)

    new := BNode{data: make([]byte, BTREE_PAGE_SIZE)}
    // check for merging
    mergeDir, sibling := shouldMerge(tree, node, idx, updated)
    switch {
    case mergeDir < 0: // left
        merged := BNode{data: 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{data: 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:
        if updated.nkeys() == 0 {
            // kid is empty after deletion and has no sibling to merge with.
            // this happens when its parent has only one kid.
            // discard the empty kid and return the parent as an empty node.
            assert(node.nkeys() == 1 && idx == 0)
            new.setHeader(BNODE_NODE, 0)
            // the empty node will be eliminated before reaching root.
        } else {
            nodeReplaceKidN(tree, new, node, idx, updated)
        }
    }
    return new
}
```

Extra care regarding empty nodes: If a node has no siblings, it cannot be merged, even if all its keys are deleted. In this case, we need to remove the empty node, this will also cause its parent to become an empty node, the empty node will propagate upwords until eventually merged.

```
// merge 2 nodes into 1
func nodeMerge(new BNode, left BNode, right BNode) {
    new.setHeader(left.btype(), left.nkeys()+right.nkeys())
    nodeAppendRange(new, left, 0, 0, left.nkeys())
    nodeAppendRange(new, right, left.nkeys(), 0, right.nkeys())
}
```

Step 4: The Conditions for Merging

The conditions for merging are:

1. The node is smaller than 1/4 of a page (this is arbitrary).
2. Has a sibling and the merged result does not exceed one page.

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

The deletion code is done.

5.2 The Root Node

We need to keep track of the root node as the tree grows and shrinks. Let's start with deletion.

This is the final interface for B-tree deletion. The height of the tree will be reduced by one when:

1. The root node is not a leaf.
2. The root node has only one child.

```
func (tree *BTree) Delete(key []byte) bool {
    assert(len(key) != 0)
    assert(len(key) <= BTREE_MAX_KEY_SIZE)
    if tree.root == 0 {
        return false
    }

    updated := treeDelete(tree, tree.get(tree.root), key)
    if len(updated.data) == 0 {
        return false // not found
    }

    tree.del(tree.root)
    if updated.btype() == BNODE_NODE && updated.nkeys() == 1 {
        // remove a level
        tree.root = updated.getPtr(0)
    } else {
        tree.root = tree.new(updated)
    }
    return true
}
```

And below is the final interface for insertion:

```
// the interface
func (tree *BTree) Insert(key []byte, val []byte) {
    assert(len(key) != 0)
    assert(len(key) <= BTREE_MAX_KEY_SIZE)
    assert(len(val) <= BTREE_MAX_VAL_SIZE)

    if tree.root == 0 {
        // create the first node
        root := BNode{data: 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 := tree.get(tree.root)
    tree.del(tree.root)

    node = treeInsert(tree, node, key, val)
    nsplit, splitted := nodeSplit3(node)
    if nsplit > 1 {
        // the root was split, add a new level.
        root := BNode{data: make([]byte, BTREE_PAGE_SIZE)}
        root.setHeader(BNODE_NODE, nsplit)
        for i, knode := range splitted[: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(splitted[0])
    }
}
```

It does two things:

1. A new root node is created when the old root is split into multiple nodes.
2. When inserting the first key, create the first leaf node as the root.

There is a little trick here. We insert an empty key into the tree when we create the first node. The empty key is the lowest possible key by sorting order, it makes the lookup function **`nodeLookupLE`** always successful, eliminating the case of failing to find a node that contains the input key.

5.3 Testing the B-Tree

Since our data structure code is pure data structure code (without IO), the page allocation code is isolated via 3 callbacks. Below is the container code for testing our B-tree, it keeps pages in an in-memory hashmap without persisting them to disk. In the next chapter, we'll implement persistence without modifying the B-tree code.

```
type C struct {
    tree  BTree
    ref   map[string]string
    pages map[uint64]BNode
}

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

We use a reference map to record each B-tree update, so that we can verify the correctness of a B-tree later.

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

```
func (c *C) del(key string) bool {
    delete(c.ref, key)
    return c.tree.Delete([]byte(key))
}
```

Test cases are left to the reader as an exercise.

5.4 Closing Remarks

This B-tree implementation is pretty minimal, but minimal is good for the purpose of learning. Real-world implementations can be much more complicated and contain practical optimizations.

There are some easy improvements to our B-tree implementation:

1. Use different formats for leaf nodes and internal nodes. Leaf nodes do not need pointers and internal nodes do not need values. This saves some space.
2. One of the lengths of the key or value is redundant — the length of the KV pair can be inferred from the offset of the next key.
3. The first key of a node is not needed because it's inherited from a link of its parent.
4. Add a checksum to detect data corruption.

The next step in building a KV store is to persist our B-tree to the disk, which is the topic of the next chapter.

06. Persist to Disk

The B-tree data structure from the previous chapter can be dumped to disk easily. Let's build a simple KV store on top of it. Since our B-tree implementation is immutable, we'll allocate disk space in an append-only manner, reusing disk space is deferred to the next chapter.

6.1 The Method for Persisting Data

As mentioned in previous chapters, persisting data to disk is more than just dumping data into files. There are a couple of considerations:

1. Crash recovery: This includes database process crashes, OS crashes, and power failures. The database must be in a usable state after a reboot.
2. Durability: After a successful response from the database, the data involved is guaranteed to persist, even after a crash. In other words, persistence occurs before responding to the client.

There are many materials describing databases using the ACID jargon (atomicity, consistency, isolation, durability), but these concepts are not orthogonal and hard to explain, so let's focus on our practical example instead.

1. The immutable aspect of our B-tree: Updating the B-tree does not touch the previous version of the B-tree, which makes crash recovery easy — should the update goes wrong, we can simply recover to the previous version.
2. Durability is achieved via the `fsync` Linux syscall. Normal file IO via `write` or `mmap` goes to the page cache first, the system has to flush the page cache to the disk later. The `fsync` syscall blocks until all dirty pages are flushed.

How do we recover to the previous version if an update goes wrong? We can split the update into two phases:

1. An update creates new nodes; write them to the disk.
2. Each update creates a new root node, we need to store the pointer to the root node somewhere.

The first phase may involve writing multiple pages to the disk, this is generally not atomic. But the second phase involves only a single pointer and can be done in an atomic single

page write. This makes the whole operation atomic — the update will simply not happen if the database crashes.

The first phase must be persisted before the second phase, otherwise, the root pointer could point to a corrupted (partly persisted) version of the tree after a crash. There should be an **`fsync`** between the two phases (to serve as a barrier).

And the second phase should also be **`fsync`**'d before responding to the client.

6.2 mmap-Based IO

The contents of a disk file can be mapped from a virtual address using the **`mmap`** syscall. Reading from this address initiates transparent disk IO, which is the same as reading the file via the **`read`** syscall, but without the need for a user-space buffer and the overhead of a syscall. The mapped address is a proxy to the page cache, modifying data via it is the same as the **`write`** syscall.

`mmap` is convenient, and we'll use it for our KV store. However, the use of **`mmap`** is not essential.

```
// create the initial mmap that covers the whole file.
func mmapInit(fp *os.File) (int, []byte, error) {
    fi, err := fp.Stat()
    if err != nil {
        return 0, nil, fmt.Errorf("stat: %w", err)
    }

    if fi.Size()%BTREE_PAGE_SIZE != 0 {
        return 0, nil, errors.New("File size is not a multiple of page size.")
    }

    mmapSize := 64 << 20
    assert(mmapSize%BTREE_PAGE_SIZE == 0)
    for mmapSize < int(fi.Size()) {
        mmapSize *= 2
    }
    // mmapSize can be larger than the file

    chunk, err := syscall.Mmap(
        int(fp.Fd()), 0, mmapSize,
```

```
        syscall.PROT_READ|syscall.PROT_WRITE, syscall.MAP_SHARED,
    )
    if err != nil {
        return 0, nil, fmt.Errorf("mmap: %w", err)
    }

    return int(fi.Size()), chunk, nil
}
```

The above function creates the initial mapping at least the size of the file. The size of the mapping can be larger than the file size, and the range past the end of the file is not accessible (**SIGBUS**), but the file can be extended later.

We may have to extend the range of the mapping as the file grows. The syscall for extending a **mmap** range is **mremap**. Unfortunately, we may not be able to keep the starting address when extending a range by remapping. Our approach to extending mappings is to use multiple mappings — create a new mapping for the overflow file range.

```
type KV struct {
    Path string
    // internals
    fp   *os.File
    tree BTree
    mmap struct {
        file   int      // file size, can be larger than the database size
        total  int      // mmap size, can be larger than the file size
        chunks [][]byte // multiple mmaps, can be non-continuous
    }
    page struct {
        flushed uint64   // database size in number of pages
        temp    [][]byte // newly allocated pages
    }
}
```

```
// extend the mmap by adding new mappings.
func extendMmap(db *KV, npages int) error {
    if db.mmap.total >= npages*BTREE_PAGE_SIZE {
```

```
        return nil
    }

    // double the address space
    chunk, err := syscall.Mmap(
        int(db.fp.Fd()), int64(db.mmap.total), db.mmap.total,
        syscall.PROT_READ|syscall.PROT_WRITE, syscall.MAP_SHARED,
    )
    if err != nil {
        return fmt.Errorf("mmap: %w", err)
    }

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

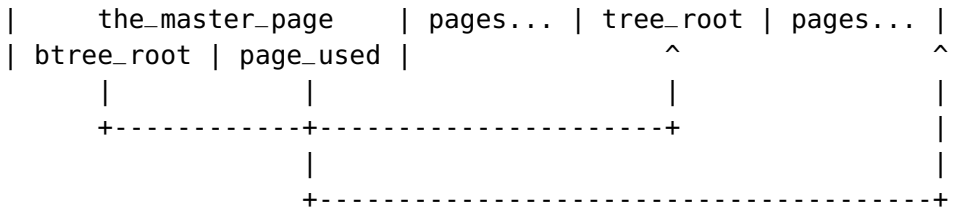
The size of the new mapping increases exponentially so that we don't have to call **mmap** frequently.

Below is how we access a page from the mapped address.

```
// callback for BTree, dereference a pointer.
func (db *KV) pageGet(ptr uint64) BNode {
    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 BNode{chunk[offset : offset+BTREE_PAGE_SIZE]}
        }
        start = end
    }
    panic("bad ptr")
}
```

6.3 The Master Page

The first page of the file is used to store the pointer to the root, let's call it the "master page". The total number of pages is needed for allocating new nodes, thus it is also stored there.



The function below reads the master page when initializing a database:

```
const DB_SIG = "BuildYourOwnDB06"

// the master page format.
// it contains the pointer to the root and other important bits.
// | sig | btree_root | page_used |
// | 16B |     8B     |     8B    |
func masterLoad(db *KV) error {
    if db.mmap.file == 0 {
        // empty file, the master page will be created on the first write.
        db.page.flushed = 1 // reserved for the master page
        return nil
    }

    data := db.mmap.chunks[0]
    root := binary.LittleEndian.Uint64(data[16:])
    used := binary.LittleEndian.Uint64(data[24:])

    // verify the page
    if !bytes.Equal([]byte(DB_SIG), data[:16]) {
        return errors.New("Bad signature.")
    }
    bad := !(1 <= used && used <= uint64(db.mmap.file/BTREE_PAGE_SIZE))
    bad = bad || !(0 <= root && root < used)
    if bad {
        return errors.New("Bad master page.")
    }
```

```
    db.tree.root = root
    db.page.flushed = used
    return nil
}
```

Below is the function for updating the master page. Unlike the code for reading, it doesn't use the mapped address for writing. This is because modifying a page via **`mmap`** is not atomic. The kernel could flush the page midway and corrupt the disk file, while a small **`write`** that doesn't cross the page boundary is guaranteed to be atomic.

```
// update the master page. it must be atomic.
func masterStore(db *KV) error {
    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)
    // NOTE: Updating the page via mmap is not atomic.
    //       Use the `pwrite()` syscall instead.
    _, err := db.fp.WriteAt(data[:], 0)
    if err != nil {
        return fmt.Errorf("write master page: %w", err)
    }
    return nil
}
```

6.4 Allocating Disk Pages

We'll simply append new pages to the end of the database until we add a free list in the next chapter.

And new pages are kept temporarily in memory until copied to the file later (after possibly extending the file).

```
type KV struct {
    // omitted...
    page struct {
```

```
        flushed uint64   // database size in number of pages
        temp    [][]byte // newly allocated pages
    }
}
```

```
// callback for BTree, allocate a new page.
func (db *KV) pageNew(node BNode) uint64 {
    // TODO: reuse deallocated pages
    assert(len(node.data) <= BTREE_PAGE_SIZE)
    ptr := db.page.flushed + uint64(len(db.page.temp))
    db.page.temp = append(db.page.temp, node.data)
    return ptr
}

// callback for BTree, deallocate a page.
func (db *KV) pageDel(uint64) {
    // TODO: implement this
}
```

Before writing the pending pages, we may need to extend the file first. The corresponding syscall is **`fallocate`**.

```
// extend the file to at least `npages`.
func extendFile(db *KV, npages int) error {
    filePages := db.mmap.file / BTREE_PAGE_SIZE
    if filePages >= npages {
        return nil
    }

    for filePages < npages {
        // the file size is increased exponentially,
        // so that we don't have to extend the file for every update.
        inc := filePages / 8
        if inc < 1 {
            inc = 1
        }
        filePages += inc
```

```
    }

    fileSize := filePages * BTREE_PAGE_SIZE
    err := syscall.Fallocate(int(db.fp.Fd()), 0, 0, int64(fileSize))
    if err != nil {
        return fmt.Errorf("fallocate: %w", err)
    }

    db.mmap.file = fileSize
    return nil
}
```

6.5 Initializing the Database

Putting together what we have done.

```
func (db *KV) Open() error {
    // open or create the DB file
    fp, err := os.OpenFile(db.Path, os.O_RDWR|os.O_CREATE, 0644)
    if err != nil {
        return fmt.Errorf("OpenFile: %w", err)
    }
    db.fp = fp

    // create the initial mmap
    sz, chunk, err := mmapInit(db.fp)
    if err != nil {
        goto fail
    }
    db.mmap.file = sz
    db.mmap.total = len(chunk)
    db.mmap.chunks = [][]byte{chunk}

    // btree callbacks
    db.tree.get = db.pageGet
    db.tree.new = db.pageNew
    db.tree.del = db.pageDel
```

```
    // read the master page
    err = masterLoad(db)
    if err != nil {
        goto fail
    }

    // done
    return nil

fail:
    db.Close()
    return fmt.Errorf("KV.Open: %w", err)
}
```

```
// cleanups
func (db *KV) Close() {
    for _, chunk := range db.mmap.chunks {
        err := syscall.Munmap(chunk)
        assert(err == nil)
    }
    _ = db.fp.Close()
}
```

6.6 Update Operations

Unlike queries, update operations must persist the data before returning.

```
// read the db
func (db *KV) Get(key []byte) ([]byte, bool) {
    return db.tree.Get(key)
}

// update the db
```

```
func (db *KV) Set(key []byte, val []byte) error {
    db.tree.Insert(key, val)
    return flushPages(db)
}

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

The **`flushPages`** is the function for persisting new pages.

```
// persist the newly allocated pages after updates
func flushPages(db *KV) error {
    if err := writePages(db); err != nil {
        return err
    }
    return syncPages(db)
}
```

It is split into two phases as mentioned earlier.

```
func writePages(db *KV) error {
    // extend the file & mmap if needed
    npages := int(db.page.flushed) + len(db.page.temp)
    if err := extendFile(db, npages); err != nil {
        return err
    }
    if err := extendMmap(db, npages); err != nil {
        return err
    }

    // copy data to the file
    for i, page := range db.page.temp {
        ptr := db.page.flushed + uint64(i)
        copy(db.pageGet(ptr).data, page)
    }
    return nil
}
```

And the **fsync** is in between and after them.

```
func syncPages(db *KV) error {
    // flush data to the disk. must be done before updating the master page.
    if err := db.fp.Sync(); err != nil {
        return fmt.Errorf("fsync: %w", err)
    }
    db.page.flushed += uint64(len(db.page.temp))
    db.page.temp = db.page.temp[:0]

    // update & flush the master page
    if err := masterStore(db); err != nil {
        return err
    }
    if err := db.fp.Sync(); err != nil {
        return fmt.Errorf("fsync: %w", err)
    }
    return nil
}
```

Our KV store is functional, but the file can't grow forever as we update the database, we'll finish our KV store by reusing disk pages in the next chapter.

07. Free List: Reusing Pages

Since our B-tree is immutable, every update to the KV store creates new nodes in the path instead of updating current nodes, leaving some nodes unreachable from the latest version. We need to reuse these unreachable nodes from old versions, otherwise, the database file will grow indefinitely.

7.1 Design the Free List

To reuse these pages, we'll add a persistent free list to keep track of unused pages. Update operations reuse pages from the list before appending new pages, and unused pages from the current version are added to the list.

The list is used as a stack (first-in-last-out), each update operation can both remove from and add to the top of the list.

```
// number of items in the list
func (fl *FreeList) Total() int
// get the nth pointer
func (fl *FreeList) Get(topn int) uint64
// remove `popn` pointers and add some new pointers
func (fl *FreeList) Update(popn int, freed []uint64)
```

The free list is also immutable like our B-tree. Each node contains:

1. Multiple pointers to unused pages.
2. The link to the next node.
3. The total number of items in the list. This only applies to the head node.

```
|   node1   |     |   node2   |     |   node3   |
+-----------+     +-----------+     +-----------+
| total=xxx |     |           |     |           |
|  next=yyy | ==> |  next=qqq | ==> |  next=eee | ==> ...
|  size=zzz |     |  size=ppp |     |  size=rrr |
|  pointers |     |  pointers |     |  pointers |
```

The node format:

```
| type | size | total | next |  pointers |
|  2B  |  2B  |   8B  |  8B  | size * 8B |
```

```
const BNODE_FREE_LIST = 3
const FREE_LIST_HEADER = 4 + 8 + 8
const FREE_LIST_CAP = (BTREE_PAGE_SIZE - FREE_LIST_HEADER) / 8
```

Functions for accessing the list node:

```
func flnSize(node BNode) int
func flnNext(node BNode) uint64
func flnPtr(node BNode, idx int)
func flnSetPtr(node BNode, idx int, ptr uint64)
func flnSetHeader(node BNode, size uint16, next uint64)
func flnSetTotal(node BNode, total uint64)
```

7.2 The Free List Datatype

The **`FreeList`** type consists of the pointer to the head node and callbacks for managing disk pages.

```
type FreeList struct {
    head uint64
    // callbacks for managing on-disk pages
    get func(uint64) BNode  // dereference a pointer
    new func(BNode) uint64  // append a new page
    use func(uint64, BNode) // reuse a page
}
```

These callbacks are different from the B-tree because *the pages used by the list are managed by the list itself*.

- The **`new`** callback is only for appending new pages since the free list must reuse pages from itself.
- There is no **`del`** callback because the free list adds unused pages to itself.
- The **`use`** callback registers a pending update to a reused page.

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

7.3 The Free List Implementation

Getting the nth item from the list is just a simple list traversal.

```
func (fl *FreeList) Get(topn int) uint64 {
    assert(0 <= topn && topn < fl.Total())
    node := fl.get(fl.head)
    for flnSize(node) <= topn {
        topn -= flnSize(node)
        next := flnNext(node)
        assert(next != 0)
        node = fl.get(next)
    }
    return flnPtr(node, flnSize(node)-topn-1)
}
```

Updating the list is tricky. It first removes **popn** items from the list, then adds the **freed** to the list, which can be divided into 3 phases:

1. If the head node is larger than **popn**, remove it. The node itself will be added to the list later. Repeat this step until it is not longer possible.
2. We may need to remove some items from the list and possibly add some new items to the list. Updating the list head requires new pages, and new pages should be reused from the items of the list itself. Pop some items from the list one by one until there are enough pages to reuse for the next phase.
3. Modify the list by adding new nodes.

```
// remove `popn` pointers and add some new pointers
func (fl *FreeList) Update(popn int, freed []uint64) {
    assert(popn <= fl.Total())
    if popn == 0 && len(freed) == 0 {
        return // nothing to do
    }

    // prepare to construct the new list
    total := fl.Total()
    reuse := []uint64{}
    for fl.head != 0 && len(reuse)*FREE_LIST_CAP < len(freed) {
        node := fl.get(fl.head)
        freed = append(freed, fl.head) // recyle the node itself
        if popn >= flnSize(node) {
            // phase 1
            // remove all pointers in this node
            popn -= flnSize(node)
        } else {
            // phase 2:
            // remove some pointers
            remain := flnSize(node) - popn
            popn = 0
            // reuse pointers from the free list itself
            for remain > 0 && len(reuse)*FREE_LIST_CAP < len(freed)+remain {
                remain--
                reuse = append(reuse, flnPtr(node, remain))
            }
            // move the node into the `freed` list
            for i := 0; i < remain; i++ {
                freed = append(freed, flnPtr(node, i))
            }
        }
        // discard the node and move to the next node
        total -= flnSize(node)
        fl.head = flnNext(node)
    }
    assert(len(reuse)*FREE_LIST_CAP >= len(freed) || fl.head == 0)

    // phase 3: prepend new nodes
    flPush(fl, freed, reuse)
```

```
    // done
    flnSetTotal(fl.get(fl.head), uint64(total+len(freed)))
}
```

```
func flPush(fl *FreeList, freed []uint64, reuse []uint64) {
    for len(freed) > 0 {
        new := BNode{make([]byte, BTREE_PAGE_SIZE)}

        // construct a new node
        size := len(freed)
        if size > FREE_LIST_CAP {
            size = FREE_LIST_CAP
        }
        flnSetHeader(new, uint16(size), fl.head)
        for i, ptr := range freed[:size] {
            flnSetPtr(new, i, ptr)
        }
        freed = freed[size:]

        if len(reuse) > 0 {
            // reuse a pointer from the list
            fl.head, reuse = reuse[0], reuse[1:]
            fl.use(fl.head, new)
        } else {
            // or append a page to house the new node
            fl.head = fl.new(new)
        }
    }
    assert(len(reuse) == 0)
}
```

7.4 Manage Disk Pages

Step 1: Modify the Data Structure

The data structure is modified. Temporary pages are kept in a map keyed by their assigned page numbers. And removed page numbers are also there.

```
type KV struct {
    // omitted...
    page struct {
        flushed uint64 // database size in number of pages
        nfree   int    // number of pages taken from the free list
        nappend int    // number of pages to be appended
        // newly allocated or deallocated pages keyed by the pointer.
        // nil value denotes a deallocated page.
        updates map[uint64][]byte
    }
}
```

Step 2: Page Management for B-Tree

The **pageGet** function is modified to also return temporary pages because the free list code depends on this behavior.

```
// callback for BTree & FreeList, dereference a pointer.
func (db *KV) pageGet(ptr uint64) BNode {
    if page, ok := db.page.updates[ptr]; ok {
        assert(page != nil)
        return BNode{page} // for new pages
    }
    return pageGetMapped(db, ptr) // for written pages
}

func pageGetMapped(db *KV, ptr uint64) BNode {
    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 BNode{chunk[offset : offset+BTREE_PAGE_SIZE]}
        }
        start = end
    }
    panic("bad ptr")
}
```

The function for allocating a B-tree page is changed to reuse pages from the free list first.

```
// callback for BTree, allocate a new page.
func (db *KV) pageNew(node BNode) uint64 {
    assert(len(node.data) <= BTREE_PAGE_SIZE)
    ptr := uint64(0)
    if db.page.nfree < db.free.Total() {
        // reuse a deallocated page
        ptr = db.free.Get(db.page.nfree)
        db.page.nfree++
    } else {
        // append a new page
        ptr = db.page.flushed + uint64(db.page.nappend)
        db.page.nappend++
    }
    db.page.updates[ptr] = node.data
    return ptr
}
```

Removed pages are marked for the free list update later.

```
// callback for BTree, deallocate a page.
func (db *KV) pageDel(ptr uint64) {
    db.page.updates[ptr] = nil
}
```

Step 3: Page Management for the Free List

Callbacks for appending a new page and reusing a page for the free list:

```
// callback for FreeList, allocate a new page.
func (db *KV) pageAppend(node BNode) uint64 {
    assert(len(node.data) <= BTREE_PAGE_SIZE)
    ptr := db.page.flushed + uint64(db.page.nappend)
    db.page.nappend++
    db.page.updates[ptr] = node.data
    return ptr
}

// callback for FreeList, reuse a page.
func (db *KV) pageUse(ptr uint64, node BNode) {
    db.page.updates[ptr] = node.data
}
```

Step 4: Update the Free List

Before extending the file and writing pages to disk, we must update the free list first since it also creates pending writes.

```
func writePages(db *KV) error {
    // update the free list
    freed := []uint64{}
    for ptr, page := range db.page.updates {
        if page == nil {
            freed = append(freed, ptr)
        }
    }
    db.free.Update(db.page.nfree, freed)

    // extend the file & mmap if needed
    // omitted...

    // copy pages to the file
    for ptr, page := range db.page.updates {
        if page != nil {
            copy(pageGetMapped(db, ptr).data, page)
        }
```

```
    }
    return nil
}
```

The pointer to the list head is added to the master page:

```
| sig | btree_root | page_used | free_list |
| 16B |     8B     |     8B    |     8B    |
```

Step 5: Done

The KV store is finished. It is persistent and crash resistant, although it can only be accessed sequentially.

There is more to learn in part II of the book:

- Relational DB on the KV store.
- Concurrent access to the database and transactions.

Part II. Mini Relational DB

Build a mini relational DB on top of the KV store.

08. Rows and Columns

8.1 Introduction

The first step in building a relational DB on top of a KV store is to add tables. A table is just a bunch of rows and columns. A subset of the columns is defined as the "primary key"; primary keys are unique, so they can be used to refer to a row (in queries and secondary indexes).

How does a table fit into a KV store? We can split a row into two parts:

1. Primary key columns go into the "key" part of the KV store.
2. Non-primary key columns go into the "value" part.

This allows us to do both point queries and range queries on the primary key. For now, we'll only consider queries on the primary key, the use of secondary indexes is deferred to a later chapter.

8.2 Data Structures

Below is the definition of rows and cells. For now, we only support two data types (int64 and bytes).

```
const (
    TYPE_ERROR = 0
    TYPE_BYTES = 1
    TYPE_INT64 = 2
)

// table cell
type Value struct {
    Type uint32
    I64  int64
    Str  []byte
}

// table row
type Record struct {
```

```
    Cols []string
    Vals []Value
}

func (rec *Record) AddStr(key string, val []byte) *Record
func (rec *Record) AddInt64(key string, val int64) *Record
func (rec *Record) Get(key string) *Value
```

The definition for the DB and tables:

```
type DB struct {
    Path string
    // internals
    kv     KV
    tables map[string]*TableDef // cached table definition
}

// table definition
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
}
```

To support multiple tables, the keys in the KV store are prefixed with a unique 32-bit number.

Table definitions have to be stored somewhere, we'll use an internal table to store them. And we'll also add an internal table to store the metadata used by the DB itself.

```
// internal table: metadata
var TDEF_META = &TableDef{
    Prefix: 1,
    Name:   "@meta",
    Types:  []uint32{TYPE_BYTES, TYPE_BYTES},
```

```
    Cols:   []string{"key", "val"},
    PKeys:  1,
}

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

8.3 Point Query

Let's implement the point query by the primary key, range queries will be added in the next chapter.

```
// get a single row by the primary key
func dbGet(db *DB, tdef *TableDef, rec *Record) (bool, error) {
    values, err := checkRecord(tdef, *rec, tdef.PKeys)
    if err != nil {
        return false, err
    }

    key := encodeKey(nil, tdef.Prefix, values[:tdef.PKeys])
    val, ok := db.kv.Get(key)
    if !ok {
        return false, nil
    }

    for i := tdef.PKeys; i < len(tdef.Cols); i++ {
        values[i].Type = tdef.Types[i]
    }
    decodeValues(val, values[tdef.PKeys:])

    rec.Cols = append(rec.Cols, tdef.Cols[tdef.PKeys:]...)
```

```
    rec.Vals = append(rec.Vals, values[tdef.PKeys:]...)
    return true, nil
}
```

The procedure is:

1. Verify that the input is a complete primary key. (**checkRecord**)
2. Encode the primary key. (**encodeKey**)
3. Query the KV store. (**db.kv.Get**)
4. Decode the value. (**decodeValues**)

```
// 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) {
    // omitted...
}
```

The method for encoding data into bytes and decoding from bytes will be explained in the next chapter. For now, any serialization scheme will do for this chapter.

```
func encodeValues(out []byte, vals []Value) []byte
func decodeValues(in []byte, out []Value)

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

To query a table we must get its definition 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)
}
```

The definition is stored as a JSON in the internal table **`TDEF_TABLE`**.

```
// get the table definition by name
func getTableDef(db *DB, name string) *TableDef {
    tdef, ok := db.tables[name]
    if !ok {
        if db.tables == nil {
            db.tables = map[string]*TableDef{}
        }
        tdef = getTableDefDB(db, name)
        if tdef != nil {
            db.tables[name] = tdef
        }
    }
    return tdef
}

func getTableDefDB(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
}
```

8.4 Updates

An update can be either insert a new row or replace an existing row. The B-tree interface is modified to support different update modes.

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

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

func (tree *BTree) InsertEx(req *InsertReq)
func (db *KV) Update(key []byte, val []byte, mode int) (bool, error)
```

The function for updating a record via the primary key:

```
// add a row to the table
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)
}
```

Different update modes:

```
// add a record
func (db *DB) Set(table string, rec Record, mode int) (bool, error) {
    tdef := getTableDef(db, table)
    if tdef == nil {
        return false, fmt.Errorf("table not found: %s", table)
    }
    return dbUpdate(db, tdef, rec, mode)
}
func (db *DB) Insert(table string, rec Record) (bool, error) {
    return db.Set(table, rec, MODE_INSERT_ONLY)
}
func (db *DB) Update(table string, rec Record) (bool, error) {
    return db.Set(table, rec, MODE_UPDATE_ONLY)
}
func (db *DB) Upsert(table string, rec Record) (bool, error) {
    return db.Set(table, rec, MODE_UPSERT)
}
```

Deleting a row is similar:

```
// delete a record by its primary key
func dbDelete(db *DB, tdef *TableDef, rec Record) (bool, error) {
    values, err := checkRecord(tdef, rec, tdef.PKeys)
    if err != nil {
        return false, err
    }

    key := encodeKey(nil, tdef.Prefix, values[:tdef.PKeys])
    return db.kv.Del(key)
}

func (db *DB) Delete(table string, rec Record) (bool, error) {
    tdef := getTableDef(db, table)
    if tdef == nil {
        return false, fmt.Errorf("table not found: %s", table)
    }
    return dbDelete(db, tdef, rec)
}
```

8.5 Create New Tables

Three steps:

1. Check the table definition.
2. Allocate the table key prefix.
3. Store the next table prefix and the table definitions.

```
func (db *DB) TableNew(tdef *TableDef) error {
    if err := tableDefCheck(tdef); err != nil {
        return err
    }

    // check the existing table
    table := (&Record{}).AddStr("name", []byte(tdef.Name))
    ok, err := dbGet(db, TDEF_TABLE, table)
    assert(err == nil)
    if ok {
        return fmt.Errorf("table exists: %s", tdef.Name)
    }

    // allocate a new prefix
    assert(tdef.Prefix == 0)
    tdef.Prefix = TABLE_PREFIX_MIN
    meta := (&Record{}).AddStr("key", []byte("next_prefix"))
    ok, err = dbGet(db, TDEF_META, meta)
    assert(err == nil)
    if ok {
        tdef.Prefix = binary.LittleEndian.Uint32(meta.Get("val").Str)
        assert(tdef.Prefix > TABLE_PREFIX_MIN)
    } else {
        meta.AddStr("val", make([]byte, 4))
    }

    // update the next prefix
    binary.LittleEndian.PutUint32(meta.Get("val").Str, tdef.Prefix+1)
    _, err = dbUpdate(db, TDEF_META, *meta, 0)
    if err != nil {
        return err
    }
```

```
    // store the definition
    val, err := json.Marshal(tdef)
    assert(err == nil)
    table.AddStr("def", val)
    _, err = dbUpdate(db, TDEF_TABLE, *table, 0)
    return err
}
```

The prefix numbers are allocated incrementally from the **`next_prefix`** key of the **`TDEF_META`** internal table. The table definitions are stored as a JSON in the **`TDEF_TABLE`** table.

Although we have added table structures, the result is still pretty much a KV store. Some important aspects are missing:

1. Range queries in the next chapter.
2. Secondary indexes in the next next chapter.

09. Range Query

We have implemented table structures on top of a KV store and we're able to retrieve records by primary key. In this chapter, we'll add the capacity to retrieve a range of records in sorted order.

9.1 B-Tree Iterator

The first step is to add the range query to the B-tree. The **`BIter`** type allows us to traverse a B-tree iteratively.

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

// 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()
```

The **`BIter`** is a path from the root node to the KV pair in a leaf node. Moving the iterator is simply moving the positions or nodes to a sibling.

```
func iterPrev(iter *BIter, level int) {
    if iter.pos[level] > 0 {
        iter.pos[level]-- // move within this node
    } else if level > 0 {
        iterPrev(iter, level-1) // move to a slibing node
    } else {
        return // dummy key
    }
```

```
    if level+1 < len(iter.pos) {
        // update the kid node
        node := iter.path[level]
        kid := iter.tree.get(node.getPtr(iter.pos[level]))
        iter.path[level+1] = kid
        iter.pos[level+1] = kid.nkeys() - 1
    }
}

func (iter *BIter) Prev() {
    iterPrev(iter, len(iter.path)-1)
}
```

`BTree.SeekLE` is the function for finding the initial position in a range query. It is just a normal B-tree lookup 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)
        if node.btype() == BNODE_NODE {
            ptr = node.getPtr(idx)
        } else {
            ptr = 0
        }
    }
    return iter
}
```

The **`nodeLookupLE`** function only works for the "less than or equal" operator in range queries, for the other 3 operators (less than; greater than; greater than or equal), the result may be off by one. We'll fix this with the **`BTree.Seek`** function.

```
const (
    CMP_GE = +3 // >=
    CMP_GT = +2 // >
    CMP_LT = -2 // <
    CMP_LE = -3 // <=
)

// find the closest position to a key with respect to the `cmp` relation
func (tree *BTree) Seek(key []byte, cmp int) *BIter {
    iter := tree.SeekLE(key)
    if cmp != CMP_LE && iter.Valid() {
        cur, _ := iter.Deref()
        if !cmpOK(cur, cmp, key) {
            // off by one
            if cmp > 0 {
                iter.Next()
            } else {
                iter.Prev()
            }
        }
    }
    return iter
}

// key cmp ref
func cmpOK(key []byte, cmp int, ref []byte) bool {
    r := bytes.Compare(key, ref)
    switch cmp {
    case CMP_GE:
        return r >= 0
    case CMP_GT:
        return r > 0
    case CMP_LT:
        return r < 0
    case CMP_LE:
        return r <= 0
    default:
        panic("what?")
    }
}
```

9.2 Data Serialization

To support range queries, the serialized primary key must be correctly compared in the KV store. One way to do this is to deserialize the primary key and compare it column by column. What we'll use is another way, to let the serialized key bytes reflect their lexicographic order, that is to say, keys can be compared correctly by **`bytes.Compare`** or **`memcmp`** without deserializing them first. Let's call this technique "order-preserving encoding", it can be used without controlling the key comparison function of the underlying KV store.

For integers, you can easily see that unsigned big-endian integers are order-preserving — the most significant bits come first in big-endian format. And null-terminated strings are also order-preserving.

For signed integers, the problem is that negative numbers have the most significant bit (sign bit) set. We need to flip the sign bit before big-endian encoding them to make negative numbers lower.

```
// order-preserving encoding
func encodeValues(out []byte, vals []Value) []byte {
    for _, v := range vals {
        switch v.Type {
        case TYPE_INT64:
            var buf [8]byte
            u := uint64(v.I64) + (1 << 63)
            binary.BigEndian.PutUint64(buf[:], u)
            out = append(out, buf[:]...)
        case TYPE_BYTES:
            out = append(out, escapeString(v.Str)...)
            out = append(out, 0) // null-terminated
        default:
            panic("what?")
        }
    }
    return out
}

func decodeValues(in []byte, out []Value) {
    // omitted...
}
```

The problem with null-terminated strings is that they cannot contain the null byte. We'll fix this by "escaping" the null byte. `"\x00"` is replaced by `"\x01\x01"`, the escaping byte `"\x01"` itself is replaced by `"\x01\x02"`, this still preserves the sort order.

```
// Strings are encoded as nul terminated strings,
// escape the nul byte so that strings contain no nul byte.
func escapeString(in []byte) []byte {
    zeros := bytes.Count(in, []byte{0})
    ones := bytes.Count(in, []byte{1})
    if zeros+ones == 0 {
        return in
    }

    out := make([]byte, len(in)+zeros+ones)
    pos := 0
    for _, ch := range in {
        if ch <= 1 {
            out[pos+0] = 0x01
            out[pos+1] = ch + 1
            pos += 2
        } else {
            out[pos] = ch
            pos += 1
        }
    }
    return out
}
```

9.3 Range Query

To wrap things up, we'll add the `Scanner` type, which allows us to iterate through a range of records in sorted order.

```
// the iterator for range queries
type Scanner struct {
    // the range, from Key1 to Key2
    Cmp1 int // CMP_??
```

```
    Cmp2 int
    Key1 Record
    Key2 Record
    // internal
    tdef   *TableDef
    iter   *BIter // the underlying B-tree iterator
    keyEnd []byte // the encoded Key2
}

// 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 {
    tdef := getTableDef(db, table)
    if tdef == nil {
        return fmt.Errorf("table not found: %s", table)
    }
    return dbScan(db, tdef, req)
}
```

Initialize the iterator:

```
func dbScan(db *DB, tdef *TableDef, req *Scanner) error {
    // sanity checks
    switch {
    case req.Cmp1 > 0 && req.Cmp2 < 0:
    case req.Cmp2 > 0 && req.Cmp1 < 0:
    default:
        return fmt.Errorf("bad range")
    }

    values1, err := checkRecord(tdef, req.Key1, tdef.PKeys)
    if err != nil {
        return err
    }
    values2, err := checkRecord(tdef, req.Key2, tdef.PKeys)
```

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

    req.tdef = tdef

    // seek to the start key
    keyStart := encodeKey(nil, tdef.Prefix, values1[:tdef.PKeys])
    req.keyEnd = encodeKey(nil, tdef.Prefix, values2[:tdef.PKeys])
    req.iter = db.kv.tree.Seek(keyStart, req.Cmp1)
    return nil
}
```

Moving the iterator:

```
// within the range or not?
func (sc *Scanner) Valid() bool {
    if !sc.iter.Valid() {
        return false
    }
    key, _ := sc.iter.Deref()
    return cmpOK(key, sc.Cmp2, sc.keyEnd)
}

// move the underlying B-tree iterator
func (sc *Scanner) Next() {
    assert(sc.Valid())
    if sc.Cmp1 > 0 {
        sc.iter.Next()
    } else {
        sc.iter.Prev()
    }
}
```

Point queries are just special cases of range queries, so why not get rid of them?

```
// get a single row by the primary key
func dbGet(db *DB, tdef *TableDef, rec *Record) (bool, error) {
    // just a shortcut for the scan operation
```

```
    sc := Scanner{
        Cmp1: CMP_GE,
        Cmp2: CMP_LE,
        Key1: *rec,
        Key2: *rec,
    }
    if err := dbScan(db, tdef, &sc); err != nil {
        return false, err
    }
    if sc.Valid() {
        sc.Deref(rec)
        return true, nil
    } else {
        return false, nil
    }
}
```

We only allow range queries on the full primary key, but range queries on a prefix of the primary key are also legitimate. We'll fix this in the next chapter, along with secondary indexes.

10. Secondary Index

In this chapter, we'll add extra indexes (also known as secondary indexes) to our database. Queries will no longer be restricted to the primary key.

10.1 Index Definitions

The **`Indexes`** and **`IndexPrefixes`** fields are added to the table definition. Like the table itself, each index is assigned a key prefix in the KV store.

```
// table definition
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
    Indexes [][]string
    // auto-assigned B-tree key prefixes for different tables/indexes
    Prefix        uint32
    IndexPrefixes []uint32
}
```

To find a row via an index, the index must contain a copy of the primary key. We'll accomplish this by appending primary key columns to the index; this also makes the index key unique, which is assumed by the B-tree lookup code.

```
func checkIndexKeys(tdef *TableDef, index []string) ([]string, error) {
    icols := map[string]bool{}
    for _, c := range index {
        // check the index columns
        // omitted...
        icols[c] = true
    }
    // add the primary key to the index
    for _, c := range tdef.Cols[:tdef.PKeys] {
        if !icols[c] {
```

```
            index = append(index, c)
        }
    }
    assert(len(index) < len(tdef.Cols))
    return index, nil
}

func colIndex(tdef *TableDef, col string) int {
    for i, c := range tdef.Cols {
        if c == col {
            return i
        }
    }
    return -1
}
```

Indexes are checked and have the primary key appended before creating a new table.

```
func tableDefCheck(tdef *TableDef) error {
    // verify the table definition
    // omitted...

    // verify the indexes
    for i, index := range tdef.Indexes {
        index, err := checkIndexKeys(tdef, index)
        if err != nil {
            return err
        }
        tdef.Indexes[i] = index
    }
    return nil
}
```

Multiple key prefixes are allocated when creating a new table.

```
// create a new table
func (db *DB) TableNew(tdef *TableDef) error {
    if err := tableDefCheck(tdef); err != nil {
        return err
```

```
    }

    // check the existing table
    // omited...

    // allocate new prefixes
    tdef.Prefix = /* omited... */
    for i := range tdef.Indexes {
        prefix := tdef.Prefix + 1 + uint32(i)
        tdef.IndexPrefixes = append(tdef.IndexPrefixes, prefix)
    }

    // update the next prefix
    ntree := 1 + uint32(len(tdef.Indexes))
    binary.LittleEndian.PutUint32(meta.Get("val").Str, tdef.Prefix+ntree)
    _, err = dbUpdate(db, TDEF_META, *meta, 0)
    if err != nil {
        return err
    }

    // store the definition
    // omited...
}
```

10.2 Maintaining Indexes

After updating a row, we need to remove the old row from the indexes. The B-tree interface is modified to return the previous value of an update.

```
type InsertReq 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
}

type DeleteReq struct {
    tree *BTree
    // in
    Key []byte
    // out
    Old []byte
}

func (tree *BTree) InsertEx(req *InsertReq)
func (tree *BTree) DeleteEx(req *DeleteReq)
```

Below is the function for adding or removing a record from the indexes. Here we encounter a problem: updating a table with secondary indexes involves multiple keys in the KV store, which should be done atomically. We'll fix that in a later chapter.

```
const (
    INDEX_ADD = 1
    INDEX_DEL = 2
)

// maintain indexes after a record is added or removed
func indexOp(db *DB, tdef *TableDef, rec Record, op int) {
    key := make([]byte, 0, 256)
    irec := make([]Value, len(tdef.Cols))
    for i, index := range tdef.Indexes {
        // the indexed key
        for j, c := range index {
            irec[j] = *rec.Get(c)
        }
        // update the KV store
        key = encodeKey(key[:0], tdef.IndexPrefixes[i], irec[:len(index)])
        done, err := false, error(nil)
        switch op {
        case INDEX_ADD:
            done, err = db.kv.Update(&InsertReq{Key: key})
        case INDEX_DEL:
```

```
            done, err = db.kv.Del(&DeleteReq{Key: key})
        default:
            panic("what?")
        }
        assert(err == nil) // XXX: will fix this in later chapters
        assert(done)
    }
}
```

Maintaining indexes after updating or deleting a row:

```
// add a row to the table
func dbUpdate(db *DB, tdef *TableDef, rec Record, mode int) (bool, error) {
    // omitted...

    req := InsertReq{Key: key, Val: val, Mode: mode}
    added, err := db.kv.Update(&req)
    if err != nil || !req.Updated || len(tdef.Indexes) == 0 {
        return added, err
    }

    // maintain indexes
    if req.Updated && !req.Added {
        decodeValues(req.Old, values[tdef.PKeys:]) // get the old row
        indexOp(db, tdef, Record{tdef.Cols, values}, INDEX_DEL)
    }
    if req.Updated {
        indexOp(db, tdef, rec, INDEX_ADD)
    }
    return added, nil
}
```

```
// delete a record by its primary key
func dbDelete(db *DB, tdef *TableDef, rec Record) (bool, error) {
    // omitted...

    deleted, err := db.kv.Del(&req)
```

```
    if err != nil || !deleted || len(tdef.Indexes) == 0 {
        return deleted, err
    }

    // maintain indexes
    if deleted {
        // likewise...
    }
    return true, nil
}
```

10.3 Using Secondary Indexes

Step 1: Select an Index

We'll also implement range queries using a prefix of an index. For example, we can do `x < a AND a < y` on the index `[a, b, c]`, which contains the prefix `[a]`. Selecting an index is simply matching columns by the input prefix. The primary key is considered before secondary indexes.

```
func findIndex(tdef *TableDef, keys []string) (int, error) {
    pk := tdef.Cols[:tdef.PKeys]
    if isPrefix(pk, keys) {
        // use the primary key.
        // also works for full table scans without a key.
        return -1, nil
    }

    // find a suitable index
    winner := -2
    for i, index := range tdef.Indexes {
        if !isPrefix(index, keys) {
            continue
        }
        if winner == -2 || len(index) < len(tdef.Indexes[winner]) {
            winner = i
        }
```

```
    }
    if winner == -2 {
        return -2, fmt.Errorf("no index found")
    }
    return winner, nil
}

func isPrefix(long []string, short []string) bool {
    if len(long) < len(short) {
        return false
    }
    for i, c := range short {
        if long[i] != c {
            return false
        }
    }
    return true
}
```

Step 2: Encode Index Prefix

We may have to encode extra columns if the input key uses a prefix of an index instead of the full index. For example, for a query **`v1 < a`** with the index **`[a, b]`**, we cannot use **`[v1] < key`** as the underlying B-tree query, because any key **`[v1, v2]`** satisfies **`[v1] < [v1, v2]`** while violating **`v1 < a`**.

Instead, we can use **`[v1, MAX] < key`** in this case where the **`MAX`** is the maximum possible value for column **`b`**. Below is the function for encoding a partial query key with additional columns.

```
// The range key can be a prefix of the index key,
// we may have to encode missing columns to make the comparison work.
func encodeKeyPartial(
    out []byte, prefix uint32, values []Value,
    tdef *TableDef, keys []string, cmp int,
) []byte {
    out = encodeKey(out, prefix, values)
```

```
    // Encode the missing columns as either minimum or maximum values,
    // depending on the comparison operator.
    // 1. The empty string is lower than all possible value encodings,
    //    thus we don't need to add anything for CMP_LT and CMP_GE.
    // 2. The maximum encodings are all 0xff bytes.
    max := cmp == CMP_GT || cmp == CMP_LE
loop:
    for i := len(values); max && i < len(keys); i++ {
        switch tdef.Types[colIndex(tdef, keys[i])] {
        case TYPE_BYTES:
            out = append(out, 0xff)
            break loop // stops here since no string encoding starts with 0xff
        case TYPE_INT64:
            out = append(out, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff)
        default:
            panic("what?")
        }
    }
    return out
}
```

For the **`int64`** type, the maximum value is encoded as all **`0xff`** bytes. The problem is that there is no maximum value for strings. What we can do is use the **`"\xff"`** as the encoding of the "pseudo maximum string value", and change the normal string encoding to not startk with the **`"\xff"`**.

The first byte of a string is escaped by the **`"\xfe"`** byte if it's **`"\xff"`** or **`"\xfe"`**. Thus all string encodings are lower than **`"\xff"`**.

```
// 1. strings are encoded as null-terminated strings,
//    escape the null byte so that strings contain no null byte.
// 2. "\xff" represents the highest order in key comparisons,
//    also escape the first byte if it's 0xff.
func escapeString(in []byte) []byte {
    // omitted...

    pos := 0
    if len(in) > 0 && in[0] >= 0xfe {
        out[0] = 0xfe
```

```
        out[1] = in[0]
        pos += 2
        in = in[1:]
    }

    // omitted...
    return out
}
```

Step 3: Fetch Rows via Indexes

The index key contains all primary key columns so that we can find the full row. The `Scanner` type is now aware of the selected index.

```
// the iterator for range queries
type Scanner struct {
    // omitted...
    db      *DB
    tdef    *TableDef
    indexNo int    // -1: use the primary key; >= 0: use an index
    iter    *BIter // the underlying B-tree iterator
    keyEnd  []byte // the encoded Key2
}
```

```
// fetch the current row
func (sc *Scanner) Deref(rec *Record) {
    assert(sc.Valid())

    tdef := sc.tdef
    rec.Cols = tdef.Cols
    rec.Vals = rec.Vals[:0]
    key, val := sc.iter.Deref()

    if sc.indexNo < 0 {
        // primary key, decode the KV pair
```

```
        // omitted...
    } else {
        // secondary index
        // The "value" part of the KV store is not used by indexes
        assert(len(val) == 0)

        // decode the primary key first
        index := tdef.Indexes[sc.indexNo]
        ival := make([]Value, len(index))
        for i, c := range index {
            ival[i].Type = tdef.Types[colIndex(tdef, c)]
        }
        decodeValues(key[4:], ival)
        icol := Record{index, ival}

        // fetch the row by the primary key
        rec.Cols = tdef.Cols[:tdef.PKeys]
        for _, c := range rec.Cols {
            rec.Vals = append(rec.Vals, *icol.Get(c))
        }
        // TODO: skip this if the index contains all the columns
        ok, err := dbGet(sc.db, tdef, rec)
        assert(ok && err == nil)
    }
}
```

Step 4: Put All Pieces Together

The **`dbScan`** function is modified to use secondary indexes. And range queries by index prefixes also work now. And it can also scan the whole table without any key at all. (The primary key is selected if no columns are present).

```
func dbScan(db *DB, tdef *TableDef, req *Scanner) error {
    // sanity checks
    // omitted...

    // select an index
```

```
    indexNo, err := findIndex(tdef, req.Key1.Cols)
    if err != nil {
        return err
    }
    index, prefix := tdef.Cols[:tdef.PKeys], tdef.Prefix
    if indexNo >= 0 {
        index, prefix = tdef.Indexes[indexNo], tdef.IndexPrefixes[indexNo]
    }

    req.db = db
    req.tdef = tdef
    req.indexNo = indexNo

    // seek to the start key
    keyStart := encodeKeyPartial(
        nil, prefix, req.Key1.Vals, tdef, index, req.Cmp1)
    req.keyEnd = encodeKeyPartial(
        nil, prefix, req.Key2.Vals, tdef, index, req.Cmp2)
    req.iter = db.kv.tree.Seek(keyStart, req.Cmp1)
    return nil
}
```

Step 5: Congratulations

We have implemented some major features of our relational DB: tables, range queries, and secondary indexes. We can start adding more features and a query language to our DB. However, some major aspects are still missing: transactions and concurrency, which will be explored in later chapters.

11. Atomic Transactions

A transaction allows multiple updates to be performed atomically (all or nothing). In the last chapter, updating a row can result in multiple KV updates (due to secondary indexes), which are not atomic and can lead to corruption if interrupted (not crash-resistant). Implementing transactions fixes this.

For now, we'll only consider sequential execution, and leave concurrency for the next chapter.

11.1 KV Transaction Interfaces

The first step is to add the KV transaction type.

```
// KV transaction
type KVTX struct {
    // later...
}
```

There are 2 ways to end a transaction:

1. By committing the transaction to persist changes.
2. Or by aborting the transaction to rollback.

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

The methods for reading and updating the KV store are moved to the transaction type. Note that these methods can no longer fail because they do not perform IOs, IO operations are performed by committing the transaction, which can fail instead.

```
// KV operations
func (tx *KVTX) Get(key []byte) ([]byte, bool) {
    return tx.db.tree.Get(key)
}
func (tx *KVTX) Seek(key []byte, cmp int) *BIter {
    return tx.db.tree.Seek(key, cmp)
}
func (tx *KVTX) Update(req *InsertReq) bool {
    tx.db.tree.InsertEx(req)
    return req.Added
}
func (tx *KVTX) Del(req *DeleteReq) bool {
    return tx.db.tree.DeleteEx(req)
}
```

11.2 DB Transaction Interfaces

Similarly, we'll also add the transaction type for **DB**, which is a wrapper around the KV transaction type.

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

func (db *DB) Begin(tx *DBTX) {
    tx.db = db
    db.kv.Begin(&tx.kv)
}
func (db *DB) Commit(tx *DBTX) error {
    return db.kv.Commit(&tx.kv)
}
func (db *DB) Abort(tx *DBTX) {
    db.kv.Abort(&tx.kv)
}
```

And the read and update methods are also moved to the transaction type.

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

Modifications to the DB code are mostly changing the arguments of functions, which will be omitted in the code listing.

11.3 Implementing the KV Transaction

The transaction type saves a copy of the in-memory data structure: the pointer to the tree root and the pointer to the free list head.

```
// KV transaction
type KVTX struct {
    db *KV
    // for the rollback
    tree struct {
        root uint64
    }
    free struct {
        head uint64
    }
}
```

This is used for rollbacks. Rolling back a transaction is simply pointing to the previous tree root, which can be done trivially even if there is an IO error while writing B-tree data.

```
// begin a transaction
func (kv *KV) Begin(tx *KVTX) {
    tx.db = kv
    tx.tree.root = kv.tree.root
    tx.free.head = kv.free.head
}
```

```
// rollback the tree and other in-memory data structures.
func rollbackTX(tx *KVTX) {
    kv := tx.db
    kv.tree.root = tx.tree.root
    kv.free.head = tx.free.head
    kv.page.nfree = 0
    kv.page.nappend = 0
    kv.page.updates = map[uint64][]byte{}
}

// end a transaction: rollback
func (kv *KV) Abort(tx *KVTX) {
    rollbackTX(tx)
}
```

Committing a transaction is not much different from how we persisted data before, except that we have to roll back on errors in the first phase of a commit.

```
// end a transaction: commit updates
func (kv *KV) Commit(tx *KVTX) error {
    if kv.tree.root == tx.tree.root {
        return nil // no updates?
    }

    // phase 1: persist the page data to disk.
    if err := writePages(kv); err != nil {
        rollbackTX(tx)
        return err
    }

    // the page data must reach disk before the master page.
    // the `fsync` serves as a barrier here.
    if err := kv.fp.Sync(); err != nil {
        rollbackTX(tx)
        return fmt.Errorf("fsync: %w", err)
    }

    // the transaction is visible at this point.
    kv.page.flushed += uint64(kv.page.nappend)
```

```
    kv.page.nfree = 0
    kv.page.nappend = 0
    kv.page.updates = map[uint64][]byte{}

    // phase 2: update the master page to point to the new tree.
    // NOTE: Cannot rollback the tree to the old version if phase 2 fails.
    //       Because there is no way to know the state of the master page.
    //       Updating from an old root can cause corruption.
    if err := masterStore(kv); err != nil {
        return err
    }
    if err := kv.fp.Sync(); err != nil {
        return fmt.Errorf("fsync: %w", err)
    }
    return nil
}
```

There are not many changes in this chapter, because we have left out an important aspect — concurrency — which will be explored in the next chapter.

12. Concurrent Readers and Writers

12.1 The Readers-Writer Problem

To support concurrent requests, we can first separate transactions into read-only transactions and read-write transactions. Readers alone can always run concurrently as they do not modify the data. While writers have to modify the tree and must be serialized (at least partially).

There are various degrees of concurrency that we can support. We can use a readers-writer lock (RWLock). It allows the execution of either:

- Multiple concurrent readers.
- A single writer.

The RWLock is a practical technique and is easy to add. However, thanks to the use of immutable data structures, we can easily implement a greater degree of concurrency.

With a RWLock, readers can be blocked by a writer and vice versa. But with immutable data structures, a writer creates a new version of data instead of overwriting the current version, this allows concurrency between readers and a writer, which is superior to the RWLock. This is what we'll implement.

Our database will support:

- Multiple concurrent readers.
- Concurrency between multiple readers and a single writer.
- Writers are fully serialized.

Note that it's possible to implement a greater degree of concurrency by only serializing writers partially. For example, if a read-write transaction reads a subset of data and then uses that data to determine what to write, we might be able to perform read operations concurrently and serialize only the final commit operation. However, this introduces new problems: even if the commit operation is serialized, writers may submit conflicting commits, so we need extra mechanisms to prevent or detect conflicts. We won't do that in this book.

12.2 Analysing the Implementation

3 major changes:

First, we'll split the transaction type into two, for both read-only and read-write transactions respectively. The B-tree type is moved to the transaction type (a snapshot). Reads from one transaction won't be affected by other transactions.

```
type KVReader struct {
    // later...
}
func (kv *KV) BeginRead(tx *KVReader)
func (kv *KV) EndRead(tx *KVReader)
func (tx *KVReader) Get(key []byte) ([]byte, bool)
func (tx *KVReader) Seek(key []byte, cmp int) *BIter
```

Next, we'll consider the use of mutexes. Writers are fully serialized, so a single mutex for writers would do. Some fields are updated by writers and read by readers, such as the latest tree root and the **mmap** chunks. These fields need another mutex to protect them.

```
type KV struct {
    // omitted...
    mu      sync.Mutex
    writer  sync.Mutex
    // omitted...
}
```

Lastly, the free list needs a redesign because we cannot reuse a page that is still reachable by a reader. Our solution is to assign an auto-incrementing version number to each version of the B-tree, and the free list stores the version number along with the page number when a page is freed. Only the free page with a version number smaller than all current readers can be reused.

```
// the in-memory data structure that is updated and committed by transactions
type FreeListData struct {
    head uint64
    // later...
}
```

```
type FreeList struct {
    FreeListData
    // for each transaction
    version   uint64   // current version
    minReader uint64   // minimum reader version
    // later...
}

// try to remove an item from the tail. returns 0 on failure.
// the removed pointer must not be reachable by the minimum version reader.
func (fl *FreeList) Pop() uint64
// add some new pointers to the head and finalize the update
func (fl *FreeList) Add(freed []uint64)
```

12.3 Concurrent Transactions

Part 1: Modify the `KV` type

1. The B-tree type is moved to the transaction type and only a root pointer remains here. Likewise, the free list is also moved.
2. The data structure and code for managing disk pages are also moved to the transaction type.
3. Added mutexes. The **`writer`** mutex is for serializing writers and the **`mu`** mutex is for protecting data fields.
4. Added version numbers. And a list of ongoing readers for tracking the minimum active version (for the free list). The reader list is maintained as a heap data structure so that the minimum version is the first element.

```
type KV struct {
    Path string
    // internals
    fp *os.File
    // mod 1: moved the B-tree and the free list
    tree struct {
        root uint64
```

```
    }
    free FreeListData
    mmap struct {
        // same; omitted...
    }
    // mod 2: moved the page management
    page struct {
        flushed uint64 // database size in number of pages
    }
    // mod 3: mutexes
    mu      sync.Mutex
    writer  sync.Mutex
    // mod 4: version number and the reader list
    version uint64
    readers ReaderList // heap, for tracking the minimum reader version
}

// implements heap.Interface
type ReaderList []*KVReader
```

Part 2: Add the Read-Only Transaction Type

```
// read-only KV transactions
type KVReader struct {
    // the snapshot
    version uint64
    tree    BTree
    mmap    struct {
        chunks [][]byte // copied from struct KV. read-only.
    }
    // for removing from the heap
    index int
}
```

The B-tree type is moved into **`KVReader`**, and so is the page management function **`pageGetMapped`**. The **`version`** and **`index`** fields are for the **`ReaderList`** heap. We also take a copy of the **`mmap`** chunks because it's modified by writers.

```
func (kv *KV) BeginRead(tx *KVReader) {
    kv.mu.Lock()
    tx.mmap.chunks = kv.mmap.chunks
    tx.tree.root = kv.tree.root
    tx.tree.get = tx.pageGetMapped
    tx.version = kv.version
    heap.Push(&kv.readers, tx)
    kv.mu.Unlock()
}

func (kv *KV) EndRead(tx *KVReader) {
    kv.mu.Lock()
    heap.Remove(&kv.readers, tx.index)
    kv.mu.Unlock()
}

// callback for BTree & FreeList, dereference a pointer.
func (tx *KVReader) pageGetMapped(ptr uint64) BNode
```

Part 3: Add the Read-Write Transaction Type

The **`KVTX`** extends the **`KVReader`** so that it gets all the read methods. And like the B-tree type, the free list and the page management data structure are also moved from the **`KV`** type.

```
// KV transaction
type KVTX struct {
    KVReader
    db   *KV
    free FreeList
    page struct {
        nappend int // number of pages to be appended
        // newly allocated or deallocated pages keyed by the pointer.
        // nil value denotes a deallocated page.
        updates map[uint64][]byte
    }
}
```

To start a read-write transaction, the writer lock must be acquired. Then we can initialize the B-tree type and the free list type. No additional locks are needed to access fields from the **`KV`** type because the writer is the only thread that can modify anything. Except for the reader list (which is modified by readers).

The **`tx.page*`** functions are simply moved out of the **`KV`** type.

```
// begin a transaction
func (kv *KV) Begin(tx *KVTX) {
    tx.db = kv
    tx.page.updates = map[uint64][]byte{}
    tx.mmap.chunks = kv.mmap.chunks

    kv.writer.Lock()
    tx.version = kv.version

    // btree
    tx.tree.root = kv.tree.root
    tx.tree.get = tx.pageGet
    tx.tree.new = tx.pageNew
    tx.tree.del = tx.pageDel
    // freelist
    tx.free.FreeListData = kv.free
    tx.free.version = kv.version
    tx.free.get = tx.pageGet
    tx.free.new = tx.pageAppend
    tx.free.use = tx.pageUse

    tx.free.minReader = kv.version
    kv.mu.Lock()
    if len(kv.readers) > 0 {
        tx.free.minReader = kv.readers[0].version
    }
    kv.mu.Unlock()
}
```

Rolling back a transaction is now a no-op, because nothing in the **`KV`** type is modified until commit.

```
// end a transaction: rollback
func (kv *KV) Abort(tx *KVTX) {
    kv.writer.Unlock()
}
```

The `KV` type is only modified a transaction is committed (for the tree root and the free list head).

```
// end a transaction: commit updates
func (kv *KV) Commit(tx *KVTX) error {
    defer kv.writer.Unlock()

    // phase 1: persist the page data to disk.
    // omitted...

    // the transaction is visible at this point.
    // save the new version of in-memory data structures.
    kv.page.flushed += uint64(tx.page.nappend)
    kv.free = tx.free.FreeListData
    kv.mu.Lock()
    kv.tree.root = tx.tree.root
    kv.version++
    kv.mu.Unlock()

    // phase 2: update the master page to point to the new tree.
    // omitted...
}
```

Both functions release the writer lock in the end, obviously.

12.4 The Free List

The free list is changed from a FILO (first-in-last-out) to a FIFO (first-in-first-out); pages freed by newer versions are added to the list head, and reused pages are removed from the list tail. This keeps the free list in sorted order (by version number).

To avoid reusing a page that a reader is still reading, reused pages must be from a version no newer than any reader. That's why we design the free list to be sorted by version number.

Unfortunately, switching to FIFO introduces some complications. We have to traverse the list to access the other end of the list (the tail). We'll keep a reference to each node in memory so that we only have to traverse the list once.

```
// the in-memory data structure that is updated and committed by transactions
type FreeListData struct {
    head uint64
    // cached pointers to list nodes for accessing both ends.
    nodes []uint64 // from the tail to the head
    // cached total number of items; stored in the head node.
    total int
    // cached number of discarded items in the tail node.
    offset int
}

type FreeList struct {
    FreeListData
    // for each transaction
    version   uint64   // current version
    minReader uint64   // minimum reader version
    freed     []uint64 // pages that will be added to the free list
    // callbacks for managing on-disk pages
    get func(uint64) BNode  // dereference a pointer
    new func(BNode) uint64  // append a new page
    use func(uint64, BNode) // reuse a page
}
```

Version numbers are added to free list nodes and to the master page:

```
| type | size | total | next |  pointer-version-pairs |
|  2B  |  2B  |   8B  |  8B  |       size * 16B       |
```

The master page format:

```
| sig | btree_root | page_used | free_list | version |
| 16B |     8B     |     8B    |     8B    |    8B   |
```

The oldest version of all readers was obtained at the beginning of a transaction.

```
// begin a transaction
func (kv *KV) Begin(tx *KVTX) {
```

```
    // omitted...

    tx.free.minReader = kv.version
    kv.mu.Lock()
    if len(kv.readers) > 0 {
        tx.free.minReader = kv.readers[0].version
    }
    kv.mu.Unlock()
}
```

It is checked when reusing pages from the list tail.

```
// try to remove an item from the tail. returns 0 on failure.
// the removed pointer must not be reachable by the minimum version reader.
func (fl *FreeList) Pop() uint64 {
    fl.loadCache()
    return flPop1(fl)
}

func flPop1(fl *FreeList) uint64 {
    if fl.total == 0 {
        return 0
    }

    // remove one item from the tail
    assert(fl.offset < flnSize(fl.get(fl.nodes[0])))
    ptr, ver := flnItem(fl.get(fl.nodes[0]), fl.offset)
    if versionBefore(fl.minReader, ver) {
        // cannot use; possibly reachable by the minimum version reader.
        return 0
    }
    fl.offset++
    fl.total--

    // discard the empty node and move to the next node
    // omitted...

    return ptr
}
```

```
// a < b
func versionBefore(a, b uint64) bool {
    return int64(a-b) < 0
}
```

Adding things to the list head is more complicated than before, but nothing special is required. We'll skip the code listing.

```
// add some new pointers to the head and finalize the update
func (fl *FreeList) Add(freed []uint64)
```

12.5 Closing Remarks

We have shown a great advantage of immutable data structures, which is easy concurrency between readers and a writer. Read-only transactions can run as long as needed, the only downside is that long-running readers prevent page reuse. However, read-write transactions are expected to be short, because writers are fully serialized. This degree of concurrency may be sufficient for some use cases.

We have explored 3 major topics in this book: persistence, indexing, and concurrency. Now it's time to add a user interface to our database — a query language — which is the topic of the next chapter.

13. Query Language: Parser

The last thing to add to our database is a query language. A query language exposes all the functionality we have implemented as a human interface.

13.1 The Grammar

13.1.1 Statements

The grammar is designed to look like SQL, which looks like English.

```
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;
```

13.1.2 Conditions

However, the **`conditions`** in our language differ from SQL. Unlike SQL, which uses the **`WHERE`** clause to select rows, we separate conditions into *indexing conditions* and *non-indexing conditions*.

1. The **`INDEX BY`** clause explicitly selects the index for the query. It represents an indexed point query or an indexed range query, and the range can be either open-ended or closed. It also controls the order of the rows.

```
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;
```

2. The **FILTER** clause selects rows without using indexes. Both the **INDEX BY** and the **FILTER** clauses are optional.

```
select expr... from table_name index by condition1 filter condition2;
select expr... from table_name filter condition2;
```

13.1.3 Expressions

The language also contains arbitrary expressions in the **SELECT** statement, **FILTER** conditions, and the **UPDATE** statement. Expressions are just recursive binary or unary operations.

Below is a list of operators in our language ordered by operator precedence.

```
-a
a * b, a / b
a + b, a - b
a = b, a < b, ...   -- all comparisons
NOT a
a AND b
a OR b
(a, b, c, ...)      -- tuple
```

13.2 Operator Precedence and Recursion

Let's start by parsing expressions. Expressions are just trees, so let's define the tree structure first.

```
// syntax tree
type QLNode struct {
```

```
    Value // Type, I64, Str
    Kids []QLNode
}

// syntax tree node types
const (
    QL_UNINIT = 0
    // scalar
    QL_STR = TYPE_BYTES
    QL_I64 = TYPE_INT64
    // binary ops
    QL_CMP_GE = 10 // >=
    QL_CMP_GT = 11 // >
    // more operators; omitted...
    // unary ops
    QL_NOT = 50
    QL_NEG = 51
    // others
    QL_SYM  = 100 // column
    QL_TUP  = 101 // tuple
    QL_ERR  = 200 // error; from parsing or evaluation
)
```

Like the structure itself, the process for parsing an expression is also recursive. Let's start with simple examples.

Consider a subset of the language consisting only of additions and column names:

```
a
a + b
a + b + c + ...
```

The pseudo-code for parsing the language looks like this:

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

Now we add the multiplication operator, which has a different precedence. Let's revise the expression `a + b`, the subexpression `a` or `b` could be a multiplication, which should be applied before the addition. (e.g.: when the `b` is `c * d`). We'll add a level of recursion to handle this:

```
def parse_add():
    node = parse_mul()
    while parse('+'):
        right = parse_mul()
        node = QLNode(type='+', kids=[node, right])
    return node

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

Notice that the `parse_add` recurses into the `parse_mul` for subexpressions, which recurses into the `parse_column`. From there we can see the pattern:

1. Each operator precedence is parsed by a level of recursion.
2. Operators that were applied first are parsed at the bottom of the recursion.

13.3 Parsing Expressions

Let's translate the pseudo-code above into some real code.

The `Parser` structure stores the current position in the input when parsing. Every parsing function takes it as an argument.

```
type Parser struct {
    input []byte
    idx   int
    err   error
}
```

The highest level of parsing is the tuple expression (e.g.: `(a, b, c, ...)`), followed by the `OR` operator, then followed by the `AND` operator, etc.

```
func pExprTuple(p *Parser, node *QLNode) {
    kids := []QLNode{{}}
    pExprOr(p, &kids[len(kids)-1])
    for pKeyword(p, ",") {
        kids = append(kids, QLNode{})
        pExprOr(p, &kids[len(kids)-1])
    }
    if len(kids) > 1 {
        node.Type = QL_TUP
        node.Kids = kids
    } else {
        *node = kids[0] // not a tuple
    }
}
```

13.3.1 Parsing Keywords

The `pKeyword` function matches one or more words from the input and advances the position.

```
// match multiple keywords sequentially
func pKeyword(p *Parser, kwds ...string) bool {
    save := p.idx
    for _, kw := range kwds {
        skipSpace(p)
        end := p.idx + len(kw)
        if end > len(p.input) {
            p.idx = save
            return false
        }

        // case insensitive matach
        ok := strings.EqualFold(string(p.input[p.idx:end]), kw)
        // token is terminated
```

```
        if ok && isSym(kw[len(kw)-1]) && end < len(p.input) {
            ok = !isSym(p.input[end])
        }
        if !ok {
            p.idx = save
            return false
        }

        p.idx += len(kw)
    }
    return true
}
```

The **skipSpace** function does what its name says. The **isSym** thing is explained later.

13.3.2 Generalization

The **pExprOr** should recurse into the **AND** operator (**pExprAnd**) according to the precedence list. But there are many precedences, so let's generalize this.

```
func pExprOr(p *Parser, node *QLNode) {
    pExprBinop(p, node, []string{"or"}, []uint32{QL_OR}, pExprAnd)
}

func pExprBinop(
    p *Parser, node *QLNode,
    ops []string, types []uint32, next func(*Parser, *QLNode),
) {
    assert(len(ops) == len(types))
    left := QLNode{}
    next(p, &left)

    for more := true; more; {
        more = false
        for i := range ops {
            if pKeyword(p, ops[i]) {
                new := QLNode{Value: Value{Type: types[i]}}
                new.Kids = []QLNode{left, {}}
                next(p, &new.Kids[1])
```

```
                left = new
                more = true
                break
            }
        }
    }

    *node = left
}
```

The **pExprBinop** is the generalized function for parsing binary operators. It takes a list of operators of equal precedence and tries to parse with each of them. The parser for the next precedence is parameterized via the **next** argument.

List of binary parsers ordered by precedence:

```
func pExprOr(p *Parser, node *QLNode) {
    pExprBinop(p, node, []string{"or"}, []uint32{QL_OR}, pExprAnd)
}
func pExprAnd(p *Parser, node *QLNode) {
    pExprBinop(p, node, []string{"and"}, []uint32{QL_AND}, pExprNot)
}
func pExprNot(p *Parser, node *QLNode) // NOT a
func pExprCmp(p *Parser, node *QLNode) // a < b, ...
func pExprAdd(p *Parser, node *QLNode) // a + b, a - b
func pExprMul(p *Parser, node *QLNode) {
    pExprBinop(p, node,
        []string{"*", "/", "%"}, []uint32{QL_MUL, QL_DIV, QL_MOD}, pExprUnop)
}
func pExprUnop(p *Parser, node *QLNode)
```

The **pExprNot** and **pExprUnop** are unary operators. They are much easier than binary operators.

```
func pExprUnop(p *Parser, node *QLNode) {
    switch {
    case pKeyword(p, "-"):
        node.Type = QL_NEG
        node.Kids = []QLNode{{}}
```

```
        pExprAtom(p, &node.Kids[0])
    default:
        pExprAtom(p, node)
    }
}
```

13.3.3 Parsing Names

The **`pExprAtom`** function is the deepest level. It parses either a column name, an integer, a string, or a pair of parentheses, which recurses back into the highest level function **`pExprTuple`**.

```
func pExprAtom(p *Parser, node *QLNode) {
    switch {
    case pKeyword(p, "("):
        pExprTuple(p, node)
        if !pKeyword(p, ")") {
            pErr(p, node, "unclosed parenthesis")
        }
    case pSym(p, node):
    case pNum(p, node):
    case pStr(p, node):
    default:
        pErr(p, node, "expect symbol, number or string")
    }
}
```

The **`pErr`** function stores an error in the **`Parser`** structure. To keep the code concise, the parser continues execution even after an error. That's why you don't see any error handling here.

The **`pSym`** function is for parsing a column name. It's just matching characters against a rule. This can also be done with a regular expression.

```
func pSym(p *Parser, node *QLNode) bool {
    skipSpace(p)

    end := p.idx
    if !(end < len(p.input) && isSymStart(p.input[end])) {
        return false
    }
    end++
    for end < len(p.input) && isSym(p.input[end]) {
        end++
    }
    if pKeywordSet[strings.ToLower(string(p.input[p.idx:end]))] {
        return false // not allowed
    }

    node.Type = QL_SYM
    node.Str = p.input[p.idx:end]
    p.idx = end
    return true
}
```

Help functions for the **pSym**:

```
var pKeywordSet = map[string]bool{
    "from":   true,
    "index":  true,
    "filter": true,
    "limit":  true,
}

func isSym(ch byte) bool {
    r := rune(ch)
    return unicode.IsLetter(r) || unicode.IsNumber(r) || r == '_'
}

func isSymStart(ch byte) bool {
    return unicode.IsLetter(rune(ch)) || ch == '_' || ch == '@'
}
```

13.4 Parsing Statements

Below is the structure for the **`SELECT`** statement.

```
// common structure for queries: `INDEX BY`, `FILTER`, `LIMIT`
type QLScan struct {
    Table  string
    // INDEX BY xxx
    Key1   QLNode // comparison, optional
    Key2   QLNode // comparison, optional
    // FILTER xxx
    Filter QLNode // boolean, optional
    // LIMIT x, y
    Offset int64
    Limit  int64
}

// stmt: select
type QLSelect struct {
    QLScan
    Names  []string // expr AS name
    Output []QLNode // expression list
}
```

And all other statements:

```
// stmt: update
type QLUpdate struct {
    QLScan
    Names  []string
    Values []QLNode
}

// stmt: insert
type QLInsert struct {
    Table  string
    Mode   int
    Names  []string
    Values [][]QLNode
}
```

```
// stmt: delete
type QLDelete struct {
    QLScan
}

// stmt: create table
type QLCreateTable struct {
    Def TableDef
}
```

Statements can be distinguished by a few starting words.

```
func pStmt(p *Parser) interface{} {
    switch {
    case pKeyword(p, "create", "table"):
        return pCreateTable(p)
    case pKeyword(p, "select"):
        return pSelect(p)
    case pKeyword(p, "insert", "into"):
        return pInsert(p, MODE_INSERT_ONLY)
    case pKeyword(p, "replace", "into"):
        return pInsert(p, MODE_UPDATE_ONLY)
    case pKeyword(p, "upsert", "into"):
        return pInsert(p, MODE_UPSERT)
    case pKeyword(p, "delete", "from"):
        return pDelete(p)
    case pKeyword(p, "update"):
        return pUpdate(p)
    default:
        pErr(p, nil, "unknown stmt")
        return nil
    }
}
```

We'll use the **SELECT** statement as the only example. The parser is divided into several components.

```
func pSelect(p *Parser) *QLSelect {
    stmt := QLSelect{}

    // SELECT xxx
    pSelectExprList(p, &stmt)

    // FROM table
    if !pKeyword(p, "from") {
        pErr(p, nil, "expect `FROM` table")
    }
    stmt.Table = pMustSym(p)

    // INDEX BY xxx FILTER yyy LIMIT zzz
    pScan(p, &stmt.QLScan)

    if p.err != nil {
        return nil
    }
    return &stmt
}
```

Let's zoom into the **`pSelectExprList`** function, which consists of finer and finer components.

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

func pSelectExpr(p *Parser, node *QLSelect) {
    expr := QLNode{}
    pExprOr(p, &expr)
    name := ""
    if pKeyword(p, "as") {
        name = pMustSym(p)
    }

    node.Names = append(node.Names, name)
```

```
    node.Output = append(node.Output, expr)
}
```

The rest of the code should be trivial at this point. We'll learn how to execute the parsed statements in the next chapter.

14. Query Language: Execution

14.1 Introduction

To execute a statement, the statement is translated into function calls to the existing DB interfaces. For example, the **`CREATE TABLE`** statement is translated into a simple function call:

```
func qlCreateTable(req *QLCreateTable, tx *DBTX) error {
    return tx.TableNew(&req.Def)
}
```

Other statements are not as simple, but they are still glue code that does not have much functionality on its own. Executing a **`SELECT`** statement is more complicated:

```
func qlSelect(req *QLSelect, tx *DBReader, out []Record) ([]Record, error) {
    // records
    records, err := qlScan(&req.QLScan, tx, out)
    if err != nil {
        return nil, err
    }

    // output
    for _, irec := range records {
        orec := Record{Cols: req.Names}
        for _, node := range req.Output {
            ctx := QLEvalContex{env: irec}
            qlEval(&ctx, node)
            if ctx.err != nil {
                return nil, ctx.err
            }
            orec.Vals = append(orec.Vals, ctx.out)
        }
        out = append(out, orec)
    }
    return out, nil
}
```

It does 2 things:

1. Fetching rows based on some conditions (**`qlScan`**).
2. Transforming rows into output by evaluating expressions (**`qlEval`**).

Both of these things are also part of some other statements. We'll look at expression evaluation first.

14.2 Expression Evaluation

There are 3 places where we need to evaluate expressions against a row. They are the expression list of the **`SELECT`** statement, the conditions in the **`FILTER`** clause, and the values of the **`UPDATE`** statement.

The **`qlEval`** function is for such tasks. To keep the code concise, the result of an evaluation (either a scalar value or an error) and the current row are put in the **`QLEvalContex`** structure.

```
// for evaluating expressions
type QLEvalContex struct {
    env Record // optional row values
    out Value
    err error
}

// evaluate an expression recursively
func qlEval(ctx *QLEvalContex, node QLNode)
```

The **`qlEval`** function evaluates subexpressions recursively and then performs the calculation of the operator. Column names and literal values (integers and strings) are obvious to handle.

```
func qlEval(ctx *QLEvalContex, node QLNode) {
    if ctx.err != nil {
        return
    }

    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
    // more; omitted...
    default:
        panic("not implemented")
    }
}
```

Operators are also easy to handle. We also do type checking when evaluating an expression. Listing the **`QL_NEG`** operator as an example:

```
    // unary ops
    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")
        }
```

The **`qlErr`** function saves an error in the **`QLEvalContex`**.

14.3 Fetching Rows

Fetching rows is another thing a query needs to do.

```
// execute a query
func qlScan(req *QLScan, tx *DBReader, out []Record) ([]Record, error) {
    sc := Scanner{}
    err := qlScanInit(req, &sc)
    if err != nil {
```

```
        return nil, err
    }

    err = tx.Scan(req.Table, &sc)
    if err != nil {
        return nil, err
    }

    return qlScanRun(req, &sc, out)
}

// create the `Scanner` from the `INDEX BY` clause
func qlScanInit(req *QLScan, sc *Scanner) error
// fetch all rows from a `Scanner`
func qlScanRun(req *QLScan, sc *Scanner, out []Record) ([]Record, error)
```

14.3.1 Initializing the Iterator

First, we need to translate the **`INDEX BY`** clause into the **`Record`** type, which is used by the **`Scanner`** iterator. An **`INDEX BY`** clause is either:

1. A point query.
2. An open-ended range (a single comparison).
3. A closed range (two comparisons of different directions).

Corresponding examples:

```
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 clause goes to the **`Key1`** and **`Key2`** fields of **`QLScan`**.

```
type QLScan struct {
    Table  string
    Key1   QLNode // first range or a point query
```

```
    Key2   QLNode // second range
    Filter QLNode
    Offset int64
    Limit  int64
}
```

The translation is just some type checkings and type conversions.

```
// create the `Scanner` from the `INDEX BY` clause
func qlScanInit(req *QLScan, sc *Scanner) error {
    if req.Key1.Type == 0 {
        // no `INDEX BY`; scan by the primary key
        sc.Cmp1, sc.Cmp2 = CMP_GE, CMP_LE
        return nil
    }

    var err error
    sc.Key1, sc.Cmp1, err = qlEvalScanKey(req.Key1)
    if err != nil {
        return err
    }

    if req.Key2.Type != 0 {
        sc.Key2, sc.Cmp2, err = qlEvalScanKey(req.Key1)
        if err != nil {
            return err
        }
    }

    if req.Key1.Type == QL_CMP_EQ && req.Key2.Type != 0 {
        return errors.New("bad `INDEX BY`")
    }
    if req.Key1.Type == QL_CMP_EQ {
        sc.Key2 = sc.Key1
        sc.Cmp1, sc.Cmp2 = CMP_GE, CMP_LE
    }
    return nil
}
```

The **`qlEvalScanKey`** function is for converting a comparison operator to the **`Record`** type

and the **`CMP_??`** enumeration. Nothing interesting there either.

```
func qlEvalScanKey(node QLNode) (Record, int, error)
```

14.3.2 Iterating Rows

We need to deal with the **`LIMIT`** and the **`FILTER`** clauses when iterating rows.

```
// fetch all rows from a `Scanner`
func qlScanRun(req *QLScan, sc *Scanner, out []Record) ([]Record, error) {
    for i := int64(0); sc.Valid(); i++ {
        // `LIMIT`
        ok := req.Offset <= i && i < req.Limit

        rec := Record{}
        if ok {
            sc.Deref(&rec)
        }

        // `FILTER`
        if ok && req.Filter.Type != 0 {
            ctx := QLEvalContex{env: rec}
            qlEval(&ctx, req.Filter)
            if ctx.err != nil {
                return nil, ctx.err
            }
            if ctx.out.Type != TYPE_INT64 {
                return nil, errors.New("filter is not of boolean type")
            }
            ok = (ctx.out.I64 != 0)
        }

        if ok {
            out = append(out, rec)
        }
        sc.Next()
    }
```

```
    return out, nil
}
```

14.4 Executing Statements

The code for the **SELECT** statement is already listed. Let's add the **DELETE** statement, which is not much different.

```
func qlDelete(req *QLDelete, tx *DBTX) (uint64, error) {
    records, err := qlScan(&req.QLScan, &tx.DBReader, nil)
    if err != nil {
        return 0, err
    }

    tdef := getTableDef(&tx.DBReader, req.Table)
    pk := tdef.Cols[:tdef.PKeys]
    for _, rec := range records {
        key := Record{Cols: pk}
        for _, col := range pk {
            key.Vals = append(key.Vals, *rec.Get(col))
        }
        deleted, err := tx.Delete(req.Table, key)
        assert(err == nil && deleted) // deleting an existing row
    }
    return uint64(len(records)), nil
}
```

Other statements should be obvious at this point.

14.5 Further Steps

As you can see, this chapter does not add much. The implementation of our query language is mostly glue code for what we have already implemented. Before reaching this point, it may have been a mystery to you how a database turns SQL into rows. Now that you have

a better understanding of databases, there are some other aspects that you may want to explore.

You can try to add more features to our database, such as joins, group bys, and aggregations, which are common in analytical queries. Getting these things to work should not be difficult at this point.

You can also build a client API and a server for our database. A server process is needed anyway for managing concurrent access. To do this, you'll need to learn network programming. Although networking in Golang is fairly easy and high-level, there is also the "*from scratch*" method if you are willing to learn more. My other book "Build Your Own Redis From Scratch" is for learning network programming from scratch and some data structures, which can be found on the official website.

This book is part of the "*Build Your Own X*" book series. You can find more books on the official website, such as the Redis book and the compiler book "From Source Code To Machine Code".

https://build-your-own.org/