Hash

Theme Song: instagram

Sometimes, range queries and sorted order don't matter too much to you, and all you care about is fast lookup. A hash table is a very popular data structure for performing quick lookups used in everything from coding interviews to industry standard indexes. In this assignment, you will implement an extendible hash table.

1 Background Knowledge

1.1 Extendible Hashing

As you know from lecture, extendible hashing is a method of hashing that adapts as you insert more and more data. For direct lookups, it can be a very efficient and lightweight indexing solution. However, it does not support range queries or maintain any sorted order, making it less than ideal as a general purpose indexing scheme.

An extendible hash table has two moving parts: the table and the buckets. The table is essentially an array of pointers to buckets. Each bucket has a *local depth*, denoted d_i , which is the number of trailing bits that every element in that bucket has in common. The table has a *global depth*, denoted d , that is the maximum of the local depths.

A hash table also has an associated hash function. Ideally, this hash function is fast and uniform across the output space, meaning that keys will be uniformly distributed across the buckets. In this implementation, we use xxhash since it is the state of the art, but really, many other hash functions would work just fine. You won't need to call this hash function directly; just use the `Hasher` function in the `hash_subr.go` file.

1.2 Basic Operations

When a hash table performs **lookup**, the search key is hashed, and then the table is used to determine the bucket that we should look at. The last d bits of the search key's hash are used as the key to the table, and then we do a linear scan over the bucket it points to.

When a hash table performs **insertion**, the insertion key is hashed, and then the table is used to determine the bucket that our new tuple should be in. The last d bits of the hash is used as the key to the hash table, after which we add the new tuple to the end of the bucket.

1.3 Splitting

When a value is inserted into a hash table, one of two things can happen: either the bucket doesn't overflow, or the bucket does overflow. If it overflows, then we need to **split** our bucket. To split a bucket, we create a new bucket with the same search key as the bucket that overflowed, but with a "1" prepended to it. We then prepend a "0" to the original bucket's search key. As an example, if a bucket with a local depth of 3 and search key "101" splits, we would be left with two buckets, each of local depth 4 and with search keys "0101" and "1101". Then, we transfer the values from the old bucket to the new bucket, and we are done.

It's worth noting that while we talk about "prepending" as if we are dealing with strings, in actuality, this action is done entirely through the bit-level representation of integers. Think about what you would have to add to a search key to effectively prepend its bit representation with a 1 or a 0.

It is possible that no values belong in either the old or newly created bucket, immediately triggering a second split. This may be a consequence of a bad hash function, but is a totally possible, albeit unlikely, scenario. Don't worry too much about these cases; trust in the hash {-: .

However, if a bucket overflows and ends up with a local depth greater than the global depth, this is no good. We should always maintain a global depth equal to the maximum of the buckets' local depths. To remedy this, we copy the hash table and append it to itself, doubling its size and increasing global depth by 1. Then, we can iterate through and make sure that the buckets are all being pointed to by the correct slots in the hash table.

1.4 Deletion and Fragmentation

When a hash table performs **deletion**, depending on your scheme, the other values may have to be moved over to fill the space it left. A good indexing scheme would use slotted pages to avoid this, but our setup doesn't readily accomodate this. Alternatively, we could populate a free list. There are many ways to handle deletion, since we don't have to maintain any sort order on the bucket level. In this assignment, you should shift over the remaining entries on the right one place left, so that there is no internal fragmenting in the bucket.

---

2 Assignment Spec

2.1 Overview

The following are the files you will need to alter:

- `cmd/bumble/main.go`
- `pkg/db/db.go`
- `pkg/hash/table.go`
- `pkg/hash/bucket.go`

And the following are the files you'll need to read:

- `pkg/db/db.go`
- `pkg/hash/hash.go`
- `pkg/hash/hash_subr.go`
- `pkg/hash/entry.go`

2.2 Subroutine Code

We've provided subroutine code so that you don't have to worry about the fact that all of this data is actually being stored on pages. We handle the marshalling and unmarshalling of data, you handle the actual logic of the hash table.

You will need to read the `hash_subr.go` and `entry.go` files to implement this assignment. The `hash.go` file is just a wrapper for the `db.Index` interface. You don't need to understand exactly how every function works, just understand what they do so that you know when you need to call them. If you ever find yourself manipulating bytes yourself, you have done something wrong and need to leverage the subroutine code more. You shouldn't change the subroutine code unless a TA has instructed you to do so. Note that there are a number of functions that are tagged with future assignment names - you won't need them now.

A common pattern you will see is how we get buckets; we use defer to ensure that the page is returned to the pager eventually.

```
// Get page and put later:
newBucket, err := NewHashBucket(table.pager, bucket.depth)
if err != nil {
    return err
}
defer newBucket.page.Put()
```

2.3 Implementation-specific Details

- Your hash table should allow inserts of duplicate values! Don't worry, non-duplicity through the repl is enforced through the database layer, so there won't be any undefined behaviour. However, we do need to allow duplicate entries for later in the course. One known issue is that is the bucket size is x , and we insert $x + 1$ elements with the same key, this scheme will lead to infinite bucket splits; ignore this for now, because there is no good solution to this problem without involving overflow chains.

2.4 Part 0: Instrumenting Existing Code

Relevant files:

- `cmd/bumble/main.go`
- `pkg/db/db.go`

We have provided you an updated `db` package in your stencil. Once you download the `hash-stencil.zip` folder, you'll see a `db` subfolder and a `hash` subfolder; both of these should be placed as is in the `pkg` folder, replacing the old version of the `db` package. Make sure that your linter doesn't throw any new errors in the `main.go` folder. When finished, your filestructure should look like this:

```
./
    - cmd/
        - bumble/
    - pkg/
        - btree/
        - config/
        - db/
        - utils/
        - hash/
        - list/
        - pager/
        - repl/
```

Next, you will need to uncomment all of the code blocks prefixed with `[HASH]` in your `main.go` (there may be none present; this is a failsafe in case the stencil ever changes in the future). If you're not sure if your `main.go` is working, we've provided a copy that should work below.

2.5 Part 1: Bucket Functions

Implement the following functions:

```
func (bucket *HashBucket) Find(key int64) (entry.Entry, bool)
func (bucket *HashBucket) Insert(key int64, value int64) (bool, error) // ALLOW DUPLICATE ENTRIES
func (bucket *HashBucket) Update(key int64, value int64) error
func (bucket *HashBucket) Delete(key int64) error
func (bucket *HashBucket) Select() ([]entry.Entry, error)
```

Some hints:

- These functions operate on a single bucket; you can safely assume that the key-value pair belongs in this bucket if these functions are called.
- Use `modifyCell`, `updateValueAt`, `updateKeyAt`, and `updateNumKeys`.

2.6 Part 2: Hash Table Functions

Implement the following functions:

```
func (table *HashTable) Find(key int64) (entry.Entry, error)
func (table *HashTable) Split(bucket *HashBucket, hash int64) error
func (table *HashTable) Insert(key int64, value int64) error // ALLOW DUPLICATE ENTRIES
func (table *HashTable) Update(key int64, value int64) error
func (table *HashTable) Delete(key int64) error
func (table *HashTable) Select() ([]entry.Entry, error)
```

Some hints:

- To hash a value, use `Hasher(key, depth)`.
- To get a bucket by its hash, use `table.GetBucket(hash)`. To get it by its page number, use `table.GetBucketByPN(pn)`
- `Split` takes the hash of the bucket that should be duplicated; this makes it easier to be called from `Insert`.
- The maximum size of a bucket is `BUCKETSIZE`.

2.7 Debugging

When using your REPL to debug, we recommend temporarily setting the bucket size to a smaller number. **WARNING: REMEMBER TO REVERT THESE CHANGES BEFORE YOU SUBMIT, OTHERWISE THE AUTOGRADER WILL TIME OUT!**:

```
// var BUCKETSIZE int64 = (PAGESIZE - BUCKET_HEADER_SIZE) / ENTRYSIZE // num entries
var BUCKETSIZE int64 = 8
```

Moreover, if you'd like to use the `mod` operator as the hasher instead of the actual hasher we use, feel free to replace `Hasher` with the following function:

```
func Hasher(key int64, depth int64) int64 {
    return int64(key %powInt(2, depth))
}
```

2.8 Testing

Run our unit tests on Gradescope. We've provided a subset of the tests here.

2.9 Getting started

To get started, download the following stencil packages: db and hash. We've provided an updated `main.go` file here. Part 0 on how to integrate this with the rest of your code.

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!