Theme Song: Wannabe

The most powerful feature of a relational database is the ability to combine your data to extract meaningful results. We use advanced queries to select, filter, and join our tables to get the insights we desire. In this assignment, you will implement a join algorithm to allow your database to return more interesting results.

---

1 Background Knowledge

1.1 Hash Joins

We've explored many join algorithms in class; now it's time to dive deep into one join algorithm: the hash join. In particular, this implementation uses the grace hash join, which is much more efficient than the classic hash join when the hash table is too large to fit into memory.

When performing a classic hash join algorithm on two tables, we have to find tuples that match on some key or value across both tables. To do so, the classic hash join creates a hash table index over the inner table, and then scans the outer table, probing the inner table for corresponding values. However, due to the nature of hash functions, these tuples are distributed at random in the inner hash tables. If the hash table is too large to reside in memory, then we have to fetch data from disk at random, which greatly increases our disk i/o cost because pages will keep getting churned through our buffer cache. So we turn to the grace hash join.

Let's say we have two tables, denoted L and R . We'll say WLOG that L is our outer relation. Grace hash join will first extract the entirety of L and R into two hash tables using the same hash function with the join key as the search key; concretely, if we were joining L 's value on R 's key, then we would create one hash table with all of L 's tuples, hashed on their value, and another hash table with all of R 's tuples, hashed on their key. Buckets are written out to disk to avoid running out of memory. Critically, the two hash tables are made to have the same global depth, even if this means expanding one table without splitting any buckets.

After partitioning our two relations and resizing our hash tables so that we have the same number of entries in our global index, we then load pairs of buckets into memory and match up pairs of tuples. Since we used the same hash function, if there was a value in L that matched with a value in R, they would be in the same bucket pair; because otherwise, their hash keys must have been different, so they wouldn't have been in match buckets. We repeat this matching process for every pair of buckets until we have finished iterating through the buckets.

The main issue to worry about is duplicity; if we ever come across the same pair of buckets twice, we may end up outputting duplicate join results. Thus, it is important to be sure that we never process the same pair of buckets twice; keeping track of which pairs of buckets we have seen will solve this problem. (It's actually a good exercise to think through when this case would occur).

You may have noticed that each pair of buckets has a disjoint result set (think about what would happen if result sets weren't disjoint), indicating that there is room for parallelization. Our stencil code implements parallelized joins for you using channels.

1.2 Bloom Filters

One inefficiency in a hash join is that you have to iterate through an entire bucket to be sure that the value you're looking for isn't present. To do that for every search value is really expensive - if we could speed up our search time somehow, that would be a huge win! Unfortunately, maintaining a bucket ordering doesn't really make sense just for search, and building an index over such a small data structure is bloated.

Bloom filters are a lightweight solution to the sparse search problem. It is a probabilistic data structure that can tell us whether an element is definitely not in a set, or maybe in a set. See a visual simulation of a bloom filter here.

A bloom filter has two parts: a bitset (essentially an array of m bytes) and a set of n hashes (in this assignments, $n = 2$). When we insert an element into a set, we hash it n times and take those values modulo m - call these values h_i . We then set each h_i -th bit to 1 in our bitset. As a concrete example, let's say the string "hello" hashed to 3 and 5. Then, we would set the 3rd and 5th bit (0-indexed) to 1.

Next, when we want to check if a particular element is in the set, we hash it n times and take those values module m - call these values h_i . Now, if at least one of the h_i -th bits are 0, then we know for certain that the element in question is not in the set; if it were, then all of these bits would be 1. If they are all 1, then we know that it is possible that the element in question is in the set. But we can't say for sure that the element is present, since the bits may have been set by a combination of other elements that set all of those bits to 1. However, with a large enough bitset and enough hashes, we can avoid collisions as much as possible, making this a nearly constant-time check for inclusion!

In this implementation, we use this bitset implementation, as well as xxhash from before, alongside murmurhash. We found our preferred bloom filter parameters using this calculator, but you are welcome to experiment and find your preferred space-time tradeoffs.

---

2 Assignment Spec

2.1 Overview

In this project, you'll implement the grace hash join algorithm, and then speed it up using bloom filters!

The following are the files you'll need to alter:

- `cmd/bumble/main.go`
- `pkg/query/hash_join.go`
- `pkg/query/bloom_filter.go`

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

- `pkg/query/query_repl.go`

2.2 New REPL Commands

Our REPL now supports the following command:

- `join <table1> <key|val> on <table2> <key|val>` - prints the result of table1 joined on table2.

2.3 Part 0: Instrumenting Existing Code

Relevant files:

- `cmd/bumble/main.go`

Once you download `query.zip`, you'll `query` subfolder; this should be placed as is in the `pkg` folder. 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/
        - query/
```

Next, you will need to uncomment all of the code blocks prefixed with `[QUERY]` in your `main.go`. If you're not sure if your `main.go` is working, we've provided a copy that should work below.

2.4 Part 1: Grace Hash Join

Implement the following functions:

```
func buildHashIndex(sourceTable db.Index, useKey bool) (tempIndex *hash.HashIndex, dbName string,
    err error)
func probeBuckets(ctx context.Context, resultsChan chan EntryPair, lBucket *hash.HashBucket,
    rBucket *hash.HashBucket, joinOnLeftKey bool, joinOnRightKey bool) error
```

Some hints:

- Use the index-agnostic `Cursor` to get all of the values you need!
- Be cognizant of whether you're joining on keys or values; it's fine to temporarily transform entries by flipping their keys and values to store them in the hash index, just be sure to flip them back before returning the results.
- Use `sendResult` to return results from `probeBuckets` - this allows the function to work with our concurrent join setup. Use `sendResult` like this: `sendResult(ctx, resultsChan, EntryPair{l: left, r: right})`.
- Be sure to handle all four combinations of key-value joining.

2.5 Part 2: Bloom Filters

Implement the following functions:

```
func CreateFilter(size int64) *BloomFilter
func (filter *BloomFilter) Insert(key int64)
func (filter *BloomFilter) Contains(key int64) bool
```

And then use your bloom filter to speed up your `probeBuckets` function.

Some hints:

- We will use two hash functions. Please use `hash.XxHasher` and `hash.MurmurHasher` as your two hash functions.
- The `bitset` package will be useful; check out `bitset.New`, `bits.Set`, and `bits.Test`.

2.6 Testing

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

2.7 Getting started

To get started, download the following stencil package: query. We've provided an updated `main.go` file here. See 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!