Theme Song: I Will Wait

A database running on a single thread, supporting a single client, is not particularly interesting. We want to be able to run multiple operations at the same time and serve multiple clients to boot. In this assignment, we'll implement both fine-grain and resource-level locking to ensure that your database is safe to use concurrently.

Note: This is a challenging assignment with a lot of moving parts. Moreover, you won't be able to use many late days on this assignment. Start early and code incrementally. Starting at the last minute is a ***very*** *bad idea. Please do not take this advice as a challenge.*

---

1 Background Knowledge

1.1 Fine-Grain Locking

You may recall mutexes and reader-writers locks from the Pager assignment - in this assignment, we'll expand our usage of locks to make our B+Tree and hash table thread-safe using fine-grain locking.

Fine-grain locking is a locking technique that involves locking a part of a data structure to ensure safe access rather than locking the entire data structure. In B+Trees, this means only locking the nodes that we're currently looking at, and in a hash table, this means only locking the buckets we're currently looking at. Clearly this is desirable - now we can have multiple tenants safely traversing and modifying the data structure at the same time, leading to huge speed ups. However, getting fine-grain locking right is incredibly nuanced, as you'll learn in this assignment.

1.1.1 Fine-Grain Locking on Pages

In the stencil code that we've provided, `Page`s have locks. Since the `Page` represents logical units in both hash tables and B+Trees, this `Lock` method will be instrumental in implementing the following two fine-grain locking schemes.

1.1.2 Fine-Grain Locking on Hash Tables

Hash tables are rather simple structures to lock; the only entrypoints into a bucket are through the lookup table. Therefore for each read or write, we only need to lock the lookup table, then the bucket!

On reads, we first acquire a read lock on the lookup table, then find our bucket. Next, we acquire a read lock on our bucket, then release our read lock on the lookup table, read our value, then release our read lock from the bucket.

On writes, we first acquire a read lock on the lookup table, then find and lock our bucket. We could have grabbed a write lock, but there's no need to grab a write lock on the lookup table unless we are sure that we are going to split; this is called **optimistic locking**, and can reduce unnecessary waiting for locks. After we've grabbed a write lock on the bucket, we check if we could potentially split; if so, we grab a write lock on the lookup table and complete our insert and split. If we don't split, we simply insert. Afterwards, we release all of our locks. You aren't required to perform optimistic locking in this assignment - it's perfectly find just to grab the write lock from the get go. However, do ensure that you release the write lock if you don't need to hold onto it - otherwise, it's not fine-grain locking at all!

1.1.3 Fine-Grain Locking on B+Trees

B+Trees are much more difficult structures to lock. There are few major concerns. Firstly, the structure of the tree can change under us as nodes split. Secondly, we don't want to be overly pessimistic in our locking, since holding a write lock on our root node locks all other clients out of the tree. Thirdly, table scans and queries get especially complicated, especially with resource locking below. And so on.

For sanity's sake, we will not be implementing fine-grain locking for selects, cursors, printing, or joins; we will focus on the main B+Tree operations: reading and writing. Primarily, we employ a scheme known as latch-crabbing or lock-crabbing, which ensures that the structure of the database won't change underneath us as we acquire locks. The following is a brief overview of lock-crabbing.

On reads, we first acquire a read lock on the root node. Then, we find the child node we want to go to, and grab a read lock on the child node. Only after locking the child node do we unlock the parent (root) node. This is called lock-crabbing, and is how we ensure that the shape of the tree doesn't change underneath us. Consider the case where we release the lock on the root before grabbing one on the child. In that split second, another thread could split the root node, making the child node obsolete. Crabbing avoids this issue entirely.

On writes, we first acquire a write lock on the root node. Then, we find the child node we want to go to, and grab a write lock on this child node. We only release the write lock on the root node if we can be sure that our child node will not split; if it can, then we hold onto the lock. As we recurse down, we hold locks on all parents that could potentially be affected by a node split. Eventually, we are guaranteed to unlock everything either after perfoming the write at a leaf node, or after a split is propagated up the tree. Because this algorithm is rather complicated, we've written a help doc here. Please use this and the associated helper functions in `btree_subr.go` when implementing locking!

1.2 Transactions

Transactions are a way of grouping multiple actions into one, ACID-compliant package. That is to say, we are guarananteed that either all of the actions in a transaction succeed or none of them succeed, and that they are isolated from other transactions. Transactions acquire locks on the resources they are accessing to be sure that they can read and write safely. Critically, notice that the nature of transaction-level locks and data structure-level locks are very different. Transaction locks are completely unaware of the underlying representation of the data; we're only concerned in logical units to preserve the integrity of the external view of the database. On the other hand, data structure-level locks are completely unaware of the data its locking; only the structure of how the data is stored. Thus, these two locking schemes are completely orthogonal to one another, and yet, are both essential for a database serving multiple tenants concurrently.

1.2.1 Strict 2PL

Our transactions will adhere to strict two-phase locking. That is, we will acquire locks as we need them, but we will hold on to all of them until after we have committed our changes to the database. **One great corrolary of this scheme is that we can be absolutely sure that there will not be cascading rollbacks**; that is, if a transaction aborts, no other transaction will have to roll back because of it! This makes our lives a lot easier when implementing aborts, but does mean that our transactions may wait unnecessarily for resources to open.

1.2.2 Deadlock Avoidance

We want to be sure that our transactions don't end up creating a deadlock; one way to do this is by detecting cycles in a "waits-for" graph. While a transaction is waiting for another transaction to free a particular resource, we

should add an edge between it and the offending transaction to the "waits-for" graph. If a cycle is detected in this graph, that means that we have deadlocked, and will not be able to proceed without dying. The last transaction to join the graph should be the one that rolls back. Critically, remember to remove edges between transactions once transactions die or are otherwise no longer waiting for a resource - otherwise, you may detect deadlocks that don't exist!

---

2 Assignment Spec

2.1 Overview

In this project, you'll implement fine-grain locking on Hash and B+Tree, then resource-level locking, and finally implement deadlock detection and avoidance! Note that the assignment parts are somewhat isolated from each other; feel free to work out of order on this assignment.

2.2 New REPL Commands

The transaction REPL now supports two new commands:

- `transaction [begin|commit]` - either starts or ends a transaction. Each client can have up to 1 transaction running at a time.
- `lock <table> <key>` - grabs a write lock on a resource. Useful for debugging.

2.3 Multiple Clients

Since we need to deal with multiple clients, we need to run BumbleBase as a server-client application rather than just as a command-line application. Running `./bumble -p transaction` should now start a server at port 8335. Then, run `./bumble_client -p 8335` to connect to the database and start running queries as normal! Using a tool like tmux might help you manage multiple terminals.

In our implementation, each client can have up to one transaction running at a time. To simulate multiple transactions, we need multiple clients; hence, now, our database supports multiple clients through `./bumble_client`. To begin a transaction, we run `transaction begin`, and to end it, `transaction commit`. Commands issued without an active transaction will be treated as a transaction of one action (`transaction begin`, [action], `transaction commit`).

2.4 Stress Testing

Because it can be useful to clobber your database to detect deadlocks or unsafe behaviour using a shotgun approach, we've provided a new executable named `bumble_stress` to help with this. We've also provided a set of sample workloads in the `workloads/` directory to run with `bumble_stress` - poke through `workloads/README.md` to get a sense of what each workload is doing, and feel free to make your own workloads!

To stress test your database, build and run `./bumble_stress -index=<btree|hash> -workload=<w.txt> -n=<n> -verify`. The workload file should essentially mimic what would normally be piped through `STDIN`, separated by newlines. The numthreads argument will specify the number of threads that will run the workload - to be clear, we split the

workload across `n` threads, not duplicate the workload for `n` threads, meaning each line will only be run once, but on different threads. The `index` flag determines which kind of index you'll be stress testing. Lastly, the `verify` flag runs a verification check at the end of the stress test to ensure that the datastructure is still consistent after the run. No `project` flag is required.

Stress testing is an especially experimental feature in the course. As a result, we will not be evaluating your code using this stress testing mechanism. Treat it as a way to discover bugs in your implementation and learn more, not as a metric for completion.

2.5 Part 0: Instrumenting Existing Code

Relevant files:

- `cmd/bumble/main.go`
- `pkg/pager/page.go`
- `pkg/hash/bucket.go`
- `pkg/hash/table.go`
- `pkg/btree/btree_subr.go`
- `pkg/btree/btree.go`

Once you download `concurrency.zip`, you'll see a `concurrency` subfolder; this should be placed as is in the `pkg` folder. As well, once you download `bumble_client.zip` and `bumble_stress.zip`, you'll see `bumble_client` and `bumble_stress` subfolders respectively; these should be placed as is in the `cmd` folder. Lastly, replace the corresponding files in your project with the files in the `patch.zip` folder provided below (i.e. replace `btree.go`, `btree_subr.go`, and `hash_subr.go`, and add the `btree/verify.go` and `hash/verify.go` files to the corresponding packages). 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/
        - bumble_client/
        - bumble_stress/
    - pkg/
        - btree/
        - config/
        - db/
        - utils/
        - hash/
        - list/
        - pager/
        - repl/
        - query/
        - concurrency/
```

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

Next, you'll want to change the `test` target in your `Makefile` to include the `-race` flag. This will signal to the Go testing framework that you'd like to detect data races in your tests. You should end up with something like this:

```
test:
    go test ./test/* -v -race
```

Lastly, add the following function to your `repl.go` file:

```
// Run the REPL.
func (r *REPL) RunChan(c chan string, clientId uuid.UUID, prompt string) {
```

```
    // Get reader and writer; stdin and stdout if no conn.
    writer := os.Stdout
    replConfig := &REPLConfig{writer: writer, clientId: clientId}
    // Begin the repl loop!
    io.WriteString(writer, prompt)
    for payload := range c {
        // Emit the payload for debugging purposes.
        io.WriteString(writer, payload+"\n")
        // Parse the payload.
        fields := strings.Fields(payload)
        if len(fields) == 0 {
            io.WriteString(writer, prompt)
            continue
        }
        trigger := cleanInput(fields[0])
        // Check for a meta-command.
        if trigger == ".help" {
            io.WriteString(writer, r.HelpString())
            io.WriteString(writer, prompt)
            continue
        }
        // Else, check user commands.
        if command, exists := r.commands[trigger]; exists {
            // Call a hardcoded function.
            err := command(payload, replConfig)
            if err != nil {
                io.WriteString(writer, fmt.Sprintf("%v\n", err))
            }
        } else {
            io.WriteString(writer, "command␣not␣found\n")
        }
        io.WriteString(writer, prompt)
    }
    // Print an additional line if we encountered an EOF character.
    io.WriteString(writer, "\n")
}
```

2.6 Part 1: Fine-Grain Locking - Hash Tables

Relevant files:

- `pkg/hash/bucket.go`
- `pkg/hash/table.go`

Add locking the following functions:

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

Some hints: - Be careful whether you use the read or write locks! - Using `defer` can save a lot of headache if you know for certain that a function will unlock some resource when it returns. - Depending on your scheme, you may/should not have to add any locking to `Split` - think about what you should lock to keep the whole table safe, though! - Both `getBucket` and `getBucketByPN` now take a second parameter, which can be one of `NO_LOCK`, `READ_LOCK`, or `WRITE_LOCK` - do this instead of locking the page directly. Don't forget to unlock the page when you are done!

2.7 Part 2: Fine-Grain Locking - B+Trees

Relevant files:

- pkg/btree/node.go
- pkg/btree/btree.go
- pkg/btree/btree_subr.go

```
func (node *LeafNode) insert(key int64, value int64, update bool) Split
func (node *LeafNode) delete(key int64)
func (node *LeafNode) get(key int64) (value int64, found bool)
func (node *InternalNode) insert(key int64, value int64, update bool) Split
func (node *InternalNode) delete(key int64)
func (node *InternalNode) get(key int64) (value int64, found bool)
```

Some hints: - Before anything else, **read through the new helper methods in** `btree_subr.go`! It will save you a LOT of time if you understand what they do. - We handle super node handling for you in the `btree.go` file. - Check out the `Insert` implementation in `btree.go` to see what else we handle for you. - We guarantee these are the only functions you need to change. - We check whether we need to unlock parents at the begin of all of these function calls. - Calling `getChildAt` with the second parameter set to `true` will lock the child page for you. After calling `getChildAt` and checking for errors, you should immediately call `initchild` to set the parent pointers correctly. - For all leaf node functions, we will definitely need to eventually unlock the current node and all parent nodes. Use the `force` parameter with the `unlockParent` function to do this. You should also take a look at `unlock`. - Lastly, think about whether we should unlock all parents in each case of a `split`. Critically, you'll have to look into the result of `node.insertSplit` in each `insert` call and react appropriately.

2.8 Part 3: Transactions

Relevant files:

- pkg/concurrency/transaction.go
- pkg/concurrency/lock.go

First, implement the following functions:

```
func (tm *TransactionManager) Lock(clientId uuid.UUID, table db.Index, resourceKey int64, lType
    LockType) error
func (tm *TransactionManager) Unlock(clientId uuid.UUID, table db.Index, resourceKey int64, lType
    LockType) error
```

Some notes:

- Transactions are identified by their `clientId`; whenever a new client connects to the server, it is assigned an ID, and since each client can only have one transaction at a time, its ID serves as a uuid for its transaction, should it exist.
- There are two abstractions surrounding how we represent locks. Firstly, a `Resource` is a unique identifier for a specific tuple in a database using its table and key. Secondly, a `LockType` helps us keep track of which lock is currently being held on a particular resource. A `Transaction` keeps track of which locktypes it is holding.
- A `Transaction` should hold either a read or a write lock on a resource; never both and certainly not multiple of each. Thus, if a transaction requests a duplicate lock, ignore the duplicate, and if a transaction requests a write lock on a resource when it holds a read lock on the same resource, upgrade the lock carefully.
- Remember to read-lock the `tmMtx` when you call `GetTransaction`!

- Remember to lock `Transaction`s when you access their resources!
- `Lock` should check what lock we want to request, add an edge to our `pGraph`, check if there is a cycle and throw an error if there is, lock the resource if we can, and remove an edge from our `pGraph`.
- `Unlock` should remove the lock from our transaction's resources and unlock the resource. Be sure to keep track of what locktype the resource had.

2.9 Part 4: Deadlock Avoidance

Relevant files:

- `pkg/concurrency/transaction.go`
- `pkg/concurrency/deadlock.go`

Implement the following function and instrument your prior transactions code with it:

```
func (g *Graph) DetectCycle() bool
```

Feel free to use any cycle detection algorithm you like. We recommend depth-first search. You'll most likely have to define some helper functions to complete this function.

2.10 Testing

Run our unit tests on Gradescope. We've provided a subset of the tests here. Moreover, try out the `bumble_stress` executable on a few workloads.

2.11 Getting started

To get started, download the following stencil package: concurrency, bumble_client, bumble_stress, patch, and workloads. 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!