Theme Song: end of THE WORLD

What happens when our database crashes? What if we lose power for a moment or run out of memory? In dramatic terms, *how can our database be prepared for the end of THE WORLD*? The answer is recovery! In this assignment, you'll make your database crash-tolerant using write-ahead logging and recovery.

---

1 Background Knowledge

1.1 Write-Ahead Logging

When our database crashes, we lose everything that was stored in volatile memory. Write-ahead logging aims to solve this problem by storing enough information to recreate the state of the database after a crash.

In write-ahead logging, we write a log specifying the action we undertook *before* we actually do it. For example, if we inserted (10, 10) into the database, we would write a log that might look like "< Tx1, INSERT, 10, 10, tablename >", indicating that transaction 1 inserted the value (10, 10) into table "tablename". If we do this for every write action a database undertakes, then we will be able to recover just by *replaying* the log!

You might be tempted to remark that write-ahead logging seems redundant and slow; why don't we just save the state of the database each time? The nature of write-ahead logging makes writing a log much less expensive than flushing a page: in short, appending a string to a file is much cheaper than mutating a data structure and flushing the corresponding pages.

1.2 Checkpoints

Let's say we've crashed and decide to recover our database by replaying history. You may have noticed that it is rather inefficient to replay a database's entire history. Especially since our database already has data stored on disk, we want to do the minimum amount of work required to restore the database.

To achieve this, we use **checkpoints**. A checkpoint is a point in the logs where we can be sure that all data in memory was flushed to disk. In other words, there is no need to replay information up to a checkpoint, since it already exists on disk. A checkpoint log also contains information specifying which transactions are currently active; as you will see soon, this is imperative for recovery, since it gives us information about what instructions should actually be undone.

When we call `checkpoint` on the REPL, we first flush all pages to disk, then write a checkpoint log to disk. This order is important; if we wrote the log first, but crashed during the flush itself, our recovery algorithm would not work!

1.3 Log Types

Now that we understand checkpoints, we should introduce the other kinds of logs. There are five kinds of logs in our database: TABLE logs, EDIT logs, START logs, COMMIT logs, and CHECKPOINT logs.

TABLE logs signify the creation of a table. The structure of a TABLE log is `< create tblType table tblName >`, where `tblType` is either `hash` or `btree`, and `tblName` is the name of the table.

EDIT logs signify actions that have modified the state of the database; for example, before a database insertion, deletion, or update, an EDIT log is written to disk. The structure of an EDIT log is `< Tx, table, INSERT|DELETE|UPDATE, key, oldval, newval >`, where `Tx` is the transaction that undertook this edit, `table` is the affected table, `key` is the affected key, and `oldval` and `newval` after the before and after results. Note that `oldval` xor `newval` can be null - think about why!

START logs signify the beginning of a new transaction. The structure of a START log is `< Tx start >`.

COMMIT logs signify the end of a running transaction. The structure of a COMMIT log is `< Tx commit >`.

CHECKPOINT logs list the currently running transactions and guarantee that all memory has been flushed to disk. The structure of a CHECKPOINT log is `< Tx1, Tx2... checkpoint >`.

Using these logs, we can define a recovery algorithm that can recovery from any database crash. We've handled the serialize and deserialize functions for you in the code; this will be useful, though, if you decide to debug by looking at your WAL.

1.4 The Recovery Algorithm

Now that we have all of the groundwork we need, let's formalize the recovery algorithm. It goes as follows:

1. Seek backwards through the logs to the most recent checkpoint and note which transactions are currently active.
2. Replay all actions from the most recent checkpoint to the end of the log, keeping track of which transactions are active.
3. Undo all transactions that have failed to commit.

We encourage you to think about the following cases and why this algorithm properly restores the database:

- A transaction began before a checkpoint and commited after.
- A transaction began before a checkpoint and never committed.
- A transaction began after a checkpoint and commited after.
- A transaction began after a checkpoint and never commited.
- A transaction began before a checkpoint and commited after, but some of its results were already written out to disk.
- A transaction begin before a checkpoint and committed after, and another transaction with the same name began after as well, and never committed.
- And so on...

---

2 Assignment Spec

2.1 Overview

In this project, you'll implement crash recovery! After this final piece, you will have fully completed a disk-oriented, multi-indexed, fully-concurrent and fully-crash tolerant database! Congratulations!

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

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

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

- `pkg/recovery/log.go`
- `pkg/recovery/reader.go`

2.2 New REPL Commands

Our REPL now supports the following commands:

- `checkpoint` - writes all pages to disk and writes a checkpoint log.
- `abort` - rolls back the current transaction.
- `crash` - crashes the database. Note that you cannot use `Ctrl-C` or `Ctrl-D` to simulate a crash, since both of those safely flush to disk.

Note that you will need transaction management from the Concurrency assignment; however, we will not be testing to ensure that your database can recover from a multi-threaded workload.

2.3 Part 0: Instrumenting Existing Code

Relevant files:

- `cmd/bumble/main.go`

Once you download `recovery.zip`, you'll see a `recovery` subfolder; this should be placed as is in the `pkg` folder. You'll also want to replace the `db` package with the one linked below. Next, you'll want to replace your `main.go` with the one provided below. 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/
        - recovery/
```

Next, add/change the following line in your `config/default.go` file:

```
// Name of log file.
const LogFileName = "./db.log"
```

You may have to insert `config.LogFileName` in `bumble_stress` to compile.

Next, add the following code to your `page.go` file:

```
// [RECOVERY] Grab the update lock.
func (page *Page) LockUpdates() {
    page.updateLock.Lock()
}

// [RECOVERY] Release the update lock.
func (page *Page) UnlockUpdates() {
    page.updateLock.Unlock()
}
```

Next, add/replace the following code to your `pager.go` file:

```
// TODO: Replace the following function
// GetFileName returns the file name.
func (pager *Pager) GetFileName() string {
    return filepath.Base(pager.file.Name())
}

// [RECOVERY] Block all updates.
func (pager *Pager) LockAllUpdates() {
    pager.ptMtx.Lock()
    for _, page := range pager.pageTable {
        page.GetKey().(*Page).LockUpdates()
    }
}

// [RECOVERY] Enable updates.
func (pager *Pager) UnlockAllUpdates() {
    for _, page := range pager.pageTable {
        page.GetKey().(*Page).UnlockUpdates()
    }
    pager.ptMtx.Unlock()
}
```

And finally, run `go get github.com/otiai10/copy`. Now you should be ready to code!

2.4 Part 1: Logging

Implement the following functions:

```
func (rm *RecoveryManager) Table(tblType string, tblName string)
func (rm *RecoveryManager) Edit(clientId uuid.UUID, table db.Index, action Action, key int64,
    oldval int64, newval int64)
func (rm *RecoveryManager) Start(clientId uuid.UUID)
func (rm *RecoveryManager) Commit(clientId uuid.UUID)
func (rm *RecoveryManager) Checkpoint()
```

Some notes:

- The `clientId` should be the value passed from the `REPLConfig`. This value should not be nil; if it is, you may have a bug in your REPL.
- For each log, update the corresponding `txStack` entry. The `txStack` tracks logs for each client individually to allow selective rollbacks. You should be calling either `make`, `append` or `delete` on values in the map.
- When a transaction begins, it should have exactly one entry in the stack corresponding to the startLog you've just written.
- When a transaction commits, you can delete all of its data in the map.
- `TABLE` logs don't need to be added to any stacks.

- Use `rm.writeToBuffer(log.toString())` to write a log. All logs should be written once.
- In `Checkpoint`, you should flush all pages to disk before writing a checkpoint log. To do so, you will have to go through each table in `rm.d.GetTables()` and run `table.GetPager().FlushAllPages()`. Write the log **AFTER** flushing to disk!
- Remember to lock the respective Pagers using `LockAllUpdates()` and `UnlockAllUpdates()`.

2.5 Part 2: Recovery

Implement the following functions:

```
func (rm *RecoveryManager) Recover() error
func (rm *RecoveryManager) Rollback(clientId uuid.UUID) error
```

Some notes:

- Use `rm.readLogs()` to get the logs and most recent checkpoint position from disk.
- Use the `Redo` and `Undo` functions we've provided to undo and redo logs. We highly recommend reading through them to get a sense of what the system does in each case.
- The `Rollback` method should iterate backwards through the actions in a transaction's `txStack` - this should indicate to you that you need to add logs to this stack at some point.
- Don't roll back a transaction if the transactions logs are not well formed; i.e. the first log is not a `START` log, a log doesn't parse, etc.
- `Commit` to **both** the RecoveryManager and TransactionManager when `Rollback` ends so that both the logs and system know that this transaction has ended..
- The `Recovery` method should implement the algorithm we described above in section titled "The Recovery Algorithm".
- You'll notice some new functions floating around; namely, `Prime` and `Delta`. These two functions change our database into something a bit more like a copy-on-write data structure. In short, these functions make sure that, when we recover, we recover from a snapshot of the database taken at the most recent checkpoint; otherwise, the internal representation of our B+Trees or Hash Tables may be corrupted due to our rolling Pager flushes. If you're interested in learning more, come to our hours and ask!

2.6 Part 3: Optimization (Optional)

Since this is the last assignment, we'd like to challenge you to take our database to the next level somehow! Given the length of Databases are complex creatures that often support many features. Sometimes they span millions of lines of code. From this rich field of possibilities, we have some suggestions for potential optimizations/improvements you could undertake; if any of them seem interesting to you, come talk with us about it. We'd love to hear about what you end up building! If you do any optimization, please write about it in your README.

- Database log compaction
- Linear probe tables
- External merge sort
- Variable-length records
- Support for more data types
- Database table statistics
- Query trees
- Query plans and optimizations

2.7 Testing

Run our unit tests on Gradescope.

2.8 Getting started

To get started, download the following stencil packages: recovery, db. 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!