HiDb: A Haskell In-Memory Relational Database

Rohan Puttagunta Arun Debray Susan Tu

CS240H

rohanp | adebray | sctu @stanford.edu

June 11, 2014

Abstract

We describe our experience implementing an in-memory relational database in Haskell that supports the standard CRUD (create, read, update, delete) operations while providing the requisite ACID (atomicity, consistency, isolation, durability) guarantees. We rely on Haskell's STM module to provide atomicity and isolation. We use a combination of STM, Haskell's type system, and dynamic type-checking in order to enforce consistency. We implement undo-redo logging and eventual disk writes to provide durability. We also provide a Haskell library which clients can use to connect to and send transactions to the database. We found that while the STM module greatly eased the implementation of transactions, the lack of support for de-serializing data into a dynamic type was not ideal.

1 Introduction

Haskell's `STM` module provides an alternative to lock-based programming that we suspected would be particularly well-suited to database implementation. `STM` shields the programmer from having to worry about correctness in a concurrent environment because it builds up actions but only commits them if all pointers contained in `TVar`s have not been changed since they have been read for that transaction. Uncommitted transactions can be retried, which is exactly what we need in a database implementation. At the same time, the module provides an appropriate level of granularity; if the value in a `TVar` that was not read for a transaction changes as the transaction is being built up, the transaction will still commit. Moreover, the module's `atomically :: STM a -> IO a` function provides a clean abstraction with which to group actions into a single atomic action, suggesting that `STM` would make it simple to group database operations into transactions [1].

The fact that `TVar`s hold pointers to data structures in memory informed our choice to implement an in-memory database. In-memory databases are fast (since they obviate the need for slow disk seeks and writes), useful for small applications whose data can be held in memory, and gaining in popularity as memory becomes cheaper. One example of a popular in-memory database is Redis, a key-value store that can store data types that are more complex than traditional SQL types [2]. In contrast, we chose to implement a fairly traditional relational database, primarily because we wanted to write a database that could be swapped in for any of the most popular databases (e.g., Oracle, MySQL, PostgreSQL).

2 Database Operations

2.1 Supported Syntax

The database operations that we support are `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`, `SELECT`, `INSERT`, `SHOW TABLES`, `UPDATE`, and `DELETE`. Although we did not finish the parser, we intended to support the following syntax for specifying these operations:

- Each individual command must go on one line.
- The list of supported types is `integer`, `boolean`, `real`, `char(`n`)` (which denotes an array of characters), and `bit(`n`)` (a bit array).
- For `CREATE TABLE`, the syntax is `CREATE TABLE` *tablename* `(`*fieldname*$_1$ *type*$_1$`, ...` *fieldname*$_n$ *type*$_n$`)`. Each type name should be one of the above specified names, and each fieldname should have no spaces or parentheses in it. In addition, one of the types may be followed by `PRIMARY KEY`.
- For `DROP TABLE`, the syntax is `DROP TABLE` *tablename*. The given tablename should not contain spaces.

- For `ALTER TABLE`, there are two choices.
  - `ALTER TABLE` *tablename* `ADD` *fieldname typename*, where the tablename and fieldname do not contain spaces, and the typename is one of the ones specified above.
  - `ALTER TABLE` *tablename* `DROP` *fieldname*, with the same constraints on the tablename and fieldname as above.
- For `SELECT`, the syntax is `SELECT` *fieldnames* `FROM` *tablename* `WHERE` *conditions*, where the table name has the same restrictions as usual, the fieldnames should be a comma-separated list of column names (but without spaces), and the conditions as follows:
  - All types may be compared with `>`, `<`, `>=`, `<=`, and `==`. Array types additionally have `in` and `contains`.
  - Comparisons may be stacked with `and` and `or`, and grouped within parentheses.
  - Spaces are allowed in condition statements.

  That is, they must follow the following context-free grammar. Here, $name_1$ and $name_2$ denote column names or constants of the corresponding type, and *op* is one of `>`, `<`, `>=`, `<=`, `==`, `in`, and `contains`.

  $$P \to name_1 \ op \ name_2$$
  $$Q \to (Q) \mid P \mid Q \texttt{ and } Q \mid Q \texttt{ or } Q$$

  Then, a valid condition is represented by Q.
- For `INSERT`, the syntax is `INSERT INTO` *tablename* `VALUES` *values*, where *values* should be a list separated by commas (spaces are allowed) corresponding to an entire new row (i.e. they should correspond to the types of the corresponding columns).
- `SHOW TABLES` has no other syntax.
- For `UPDATE`, the syntax is `UPDATE` *tablename* `SET` *assignments* `WHERE` *conditions*, where the conditions are as for a `SELECT` call, and the assignment is of the form *fieldname*=*const*, for a constant that is of the type held in that column. In particular, there should not be spaces in the assignment clause.
- For `DELETE`, the syntax is `DELETE FROM` *tablename* `WHERE` *conditions*, where the conditions are as for a `SELECT` call.

2.2 Implementation

Each operation is implemented as a Haskell function. Operations which make changes to the database, should they succeed, should return lists of `LogOperation`s, where the `LogOperation` datatype represents log entries and we use different constructors for different types of entries (see durability section below for further discussion). Since we cannot perform IO from within STM, the calling function is responsible for actually writing these `LogOperation`s, which are instances of `Read` and `Show` to allow for easy serializability and de-serializability, to the in-memory log. If the operation was malformed (for example, the referenced table does not exist, or the user failed to specify the value for a column for which there is no default value), then we return some error message to the user. For operations such as `SELECT` that do not modify the database, we return a `String` that can be written back to the user.

It is interesting to note that while being forced to return everything that needs to be written to the log resulted in some cumbersome type signatures, Haskell's explicit separation of pure and impure computations is also essentially what makes a safe implementation of `STM` possible; because all computation done within `STM` is effectless, there is no danger of performing an action with side-effects multiple times as the transaction is retried. In the context of our database, there is no danger of writing the same `LogOperation` to the log multiple times.

In our Haskell implementation of `SELECT`, we choose to return a `Table` rather than a `String` because while we did not implement this functionality in this verison of HiDb, in theory could we perform futher operations involving the returned table. We also chose to make use of a `Row` datatype, which is a wrapper around a `Fieldname -> Maybe Element` function. This allows us to use functions of the type `Row -> STM(Bool)` as the condition for whether a row of a table should be deleted or updated. It also allows us to express an update as `Row -> Row`.

One interesting implication of implementing these operations in Haskell is that the laziness of the language essentially makes our transactions lazy as well; the `IO` actions are not evaluated until we attempt to write the corresponding `LogOperation`s, until we attempt to write the results back to the user (i.e., the string resulting from a `SELECT`), or until another transaction does something that requires thunks from the earlier transaction to be evaluated.

3 Parsing

In order to provide a usable end program, we need a server that can respond to queries and manage a database.

Upon connecting to a client, this server devotes a thread to listening to instructions from a client, which are received in blocks. Since there may be multiple clients acting at the same time, as well as possible other threats to durability (e.g. power outages), it would be optimal for all of these commands to be run in one atomic block, so that they can be easily undone and redone.

However, a few commands alter the structure of the database itself, i.e. `CREATE TABLE`, `DROP TABLE`, and

`ALTER TABLE`. These commands change the types of fields or the tables themselves, and thus should go in their own atomic actions.

Thus, the first step in parsing is to split the client's block of actions into sections that can be executed atomically: as many commands as possible that don't alter the structure of the database, followed by one that does, then as many as possible that don't, and so on. The following code provides a simplified implementation of this.

Listing 1: Grouping the client's issued commands.

```
executeRequests :: TVar Database -> TVar
    ActiveTransactions -> Log -> TransactionID
    -> [String] -> IO (Maybe ErrString)
-- base case
executeRequests _ _ _ _ [] = return Nothing
-- altersTable checks whether a command is one
    of the aforementioned ones
-- that alters the structure of the database.
executeRequests db transSet logger tID cmds = do
  case findIndex altersTable cmds of
    Just 0 -> do -- the first request alters the
        table.
      result <- actReq [head cmds]
      recurseReq $ tail cmds
    Just n -> do -- the (n+1)st request alters
        the table, but the first n don't.
      actReq $ take n cmds
      recurseReq $ drop n cmds
    -- if nothing changes the table, then we can
        just do everything.
    Nothing -> actReq cmds
  where actReq = atomicAction db transSet logger
      tID
      recurseReq = executeRequests db transSet
          logger (incrementTId tID)
```

Once this is done, the server has blocks of commands that can be issued atomically. This allows us to take advantage of the STM framework provided by Haskell, using the `atomically` function to convert all of the atomic actions on the database into one `IO` action that will execute atomically.

The return type of these operations is of the form `STM (Either ErrString [LogOperation])`, and these two cases are handled differently:

- If the database operation returns `Left err`, then the rest of the block should stop executing, and the error is reported to the user.
- If the database operation succeeded, returning `Right logOp`, then the resulting statements should be written to the log, to power the undo/redo logging of the whole database.

That atomicity was the easiest aspect of the parser/server was one of the primary advantages of using Haskell.

Finally, the `parseCommand` function pattern-matches on the different possible commands and chooses the correct operation to execute. This is where the bulk of the explicit parsing was done; however, in order to simplify this function, we made some simplifying constraints on the subset of SQL accepted by this parser, as detailed in Section 2.1 above.

One trick in parsing was that the database is strongly typed, but the client sends over strings; in theory, the server has to read data from strings into an arbitrary type specified at runtime. Haskell's type system does not make this very easy, and we found it much easier to restrict to several basic types and pattern-match against a `TypeRep` stored along with every database column. Then, there are only a few types to check, and these can be done without too much code.

The types we used are `Int`, `Real` (backed by a `Double`), `Bool`, and `Char` and bit arrays; these were chosen to represent the fundamental SQL types. It would not be hard to add more types, since this just involves adding one step to each pattern-matching function for the types.

4 Storage: Data Structures

Listing 2: Main data structures for the database.

```
data Database = Database { database :: Map
    Tablename (TVar Table) }

data Table
  = Table { rowCounter :: Int
          , primaryKey :: Maybe Fieldname
          , table :: Map Fieldname Column
          }

data Column
  = Column { default_val :: Maybe Element
           , col_type :: TypeRep
           , column :: TVar(Map RowHash (TVar
               Element))
           }

data Element = forall a. (Show a, Ord a, Eq a,
    Read a, Typeable a) =>
  Element (Maybe a) -- Nothing here means that
      it's null
```

Note that we needed to use the `Existential Quantification` language extension in our definition of `Element`, which allows us to put any type that is an instance of `Show`, `Ord`, `Eq`, `Read`, and `Typeable` into `Element`. Note that this means that we do not statically enforce the fact that every column should contain elements of just one type. It seems that it is not possible to statically enforce this in Haskell, since when the user attempts an insert, there is no way to know statically (since the user only provides the input at runtime) whether this input will be of the same type as the other elements in the column. We therefore must enforce the type of columns at run-time in the parsing stage (as previously discussed).

We use multiple layers of `TVar`s in order to allow operations that touch different parts of the database to proceed at the same time. For example, having each `Table` be stored in a `TVar` means that if client A updates a row in one table, client B can concurrently update a row in another table. Moreover, consider what happens if we attempt changes that affect the database at different levels of granularity: Suppose we concurrently attempt two operations, O_1 and O_2, where O_1 is the insertion of a new column into a table and O_2 is the updating of a value in an existing column in the same table. If O_1 completes first, then O_2 will have to be retried because the pointer (which had to be read for O_2 to occur) in the `TVar` surrounding the `Table` type will have changed. However, if O_2 completes first, then O_1 will complete without issue because the only `TVar` that O_2 changed are ones that did not need to be read for O_1.

5 Durability

Almost by definition, a database needs to maintain its stored data between sessions; a crash cannot erase all of the data. As such we cannot rely solely on in-memory storage. We must push our database to disk periodically to ensure that we have a relatively recent and persistent copy of our data at all times. We call these periodic pushes *checkpoints*.

5.1 Checkpoint

The most naive checkpointing algorithm would simply push all of the tables in memory to disk. However, this push might capture partially completed transactions, which might lead to inconsistent or corrupted data. The simple solution would be to block new transactions from starting when we want to run a checkpoint, and then do the push to disk once the transactions in progress finish. In this manner, we can ensure that no partially completed transaction is stored on disk. The obvious drawback is that we must stop all operations when we want to run a checkpoint, and then we must wait for the disk to finish its writes. This algorithm is clearly unacceptable as stopping in-memory operations for disk operations destroys the point of having an in-memory database.

Checkpointing algorithms that do not require all transactions to stop are called *quiescent checkpointing*. Quiescent checkpoints necessarily push partially completed checkpoints to disk, but we can compensate by logging the operations we do. By pushing this log to disk, we can theoretically repair partially completed transactions. Formalizing this notion a bit, we log an operation whenever we modify an element in some table by storing the quadruple consisting of the table identifier, the element identifier, the old value, and the new value. If we ensure that the log corresponding to an operation is pushed to disk before the operation is pushed to disk, we can identify partially completed transactions that have been pushed to disk and then repair any inconsistencies.

The only need for this repairing is when the database is starting up, at which point it reads the stored data on disk. Once read in, there are two possible types of repairs to do for partially completed transactions - transactions that are completely logged on disk might need to be redone and transactions that are partially logged on disk might need to be undone. To distinguish between these two types of repairs, we must scan the log from the end. When we see a log entry corresponding to a transaction commit, we add the transaction to a list of transactions to be redone. When we see log entries for modifying an element of a table in a transaction not marked to be redone, we must undo the operation. After completing this first pass, we rescan the log from the start to the end, redoing transactions that we marked in the first pass. Undoing and redoing a modification is as simple as setting the element to the old and new logged value, respectively.

As currently described, the log will continuously grow, getting overwhelmingly large before long - the final detail to discuss is shrinking the log. We can delete a transaction in the log after we are sure that the entire transaction has been pushed to disk. In particular, if we start a checkpoint after a transaction has committed, we can remove that transaction from the log once the checkpoint terminates. As such, we trim and flush (again) the log after finishing a checkpoint.

5.2 Serializing

As our objects representing the database are wrapped in `TVar`s, we cannot define a method like `show` that converts a `Table` to a `String` without using an unsafe function such as `unsafePerformIO`. We can, however, define a function with the type `Table -> IO String`. Similarly we also can define a function with the type `String -> IO (TVar Table)`.

Since we couldn't use automatically derived instances of `show` and `read`, we were faced with two prospects - manually write the encoding for our Objects or to somehow convert them into data structures that can derive `show` and `read`. The latter route seemed safer and much easier, so we defined two (private) data structures `ShowableTable` and `ShowableColumn` that derive `show` and `read`. Converting between our regular `Table` and `Column` data structures and their showable variants is simply a matter of extracting all of the necessary variables from their `TVar` containers.

The last problem with respect to serialization is serializing the existentially quantified data structure `Element`. We must know the type of the element stored in the `Element` if we want to `read` it. As we only use three different types in our database, we simply prepend the type in `show` and then check the prefix to identify the

type during `read`. We use a similar procedure to serialize the types of a column (stored as a `TypeRep`).

6 Summary & Future Work

We found that Haskell's support for lightweight threads and the `STM` module were, as expected, very well-suited to database implementation. However, we also need dynamic types, which presents difficulties given Haskell's unusually strong type system. Furthermore, we found it difficult to implement the Operation module cleanly; the code indeed appears quite imperative due the constant use of `do` notation (which we could have replaced with `>>=`, but we do not believe that would have made the code any easier to read).

Given that we were unable to complete the parser and were ultimately only able to test the operations and the serialization and the deserialization of the database, logical next steps would be to complete the parser and to make it more robust. After that, implementing operations such as `INNER JOIN`, specifications such as `FOREIGN KEY`, and just generally fleshing out our implementation to include the full set of traditional SQL operations would be a logical next step. Another logical direction of exploration would be to compare the performance of this database to the relational databases whose functionality we tried to emulate.

Code is available at `https://github.com/susanctu/Haskell-In-Memory-DB/`. Tests verifying that operations and rehydration work are in `Main.hs`.

References

[1] Harris T., Marlow S. Jones S., Herlihy M. "Composable Memory Transactions". `http://research.microsoft.com/pubs/67418/2005-ppopp-composable.pdf`

[2] Redis. `http://redis.io/`