Optimizing Transaction Execution in a Purely Functional Database

Towards Database Maintenance without Downtime

Lesley Wevers
University of Twente

Database maintenance downtime

Gmail by Google

Sorry... account maintenance underway

We're currently performing maintenance on your account. You won't be able to log in while maintenance is underway, but your account data and messages are safe. Unfortunately, we can't predict exactly how long this will take.

If this maintenance lasts more than 24 hours, please contact us at gmail-maintenance@google.com.

Example: Changing a database table

name	postal_num	postal_alpha
Alice	6382	ZB
Bob	9832	KW

name	postal
Alice	6382 ZB
Bob	9832 KW
Eve	63729

Ideally, in SQL we do the following:

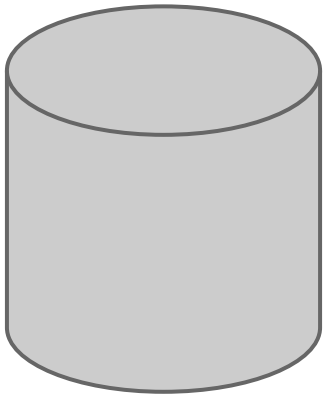
```
BEGIN TRANSACTION;
ALTER TABLE users ADD COLUMN postal;
UPDATE users SET postal = postal_num + " " + postal_alpha;
ALTER TABLE users DROP COLUMN postal_num;
ALTER TABLE users DROP COLUMN postal_alpha;
UPDATE STORED PROCEDURE create_user ...
UPDATE STORED PROCEDURE get_user_details ...
COMMIT;
```

Transactions

A transaction declaratively guarantees:

- Atomicity
- Consistency
- Isolation
  - Serializability
  - Recoverability
- Durability (for persistent systems)

```
BEGIN TRANSACTION;
ALTER TABLE users ADD COLUMN postal;
UPDATE users SET postal = postal_num + " " + postal_alpha;
ALTER TABLE users DROP COLUMN postal_num;
ALTER TABLE users DROP COLUMN postal_alpha;
UPDATE STORED PROCEDURE create_user ...
UPDATE STORED PROCEDURE       get_user_details ...
COMMIT;
```



Source of the downtime problem

Transactions can be aborted at any time:

- By choice of the client
- Due to concurrency conflicts
- Hardware failures

Recoverability:

- A client may not commit if it has read any data of an uncommitted transaction.

Our approach

- Send database programs to the database.
- A program can commit before it is executed.
- Optimize transaction execution to avoid or minimize downtime using techniques from functional languages.

Functional Transactions

Transaction :

```
DB -> (DB, Result)
```

Transaction manager:

```
DB -> [Transaction] -> [Result]
```

Example

```
data User = User String String Integer
type DB = [User]

add_user :: User -> DB -> (DB, ())
add_user u db = (u : db, ())

count_users :: DB -> (DB, Boolean)
count_users db = (db, length db)
```

Database states

- States are constructed from bulk data types, and can implement many kinds of data models.

- Copies can be created cheaply through sharing
  - Updates create new versions of the database
  - Reads have a stable snapshot
  - Allows experimenting with changes

Lazy evaluation of database states

Uses

- Postpone computing parts of database updates
- Effects of updates can be visible immediately
- Don’t do unnecessary work in queries

Limitations

- Blocks when there are data dependencies
- Transactions can not abort
- Memory requirements

Memoization

Uses

- Keep views up to date cheaply
- Share work between transactions
- Reduce cost of retrying failed transactions

Limitations

- Only works for divide-and-conquer functions
- Potentially a large amount of memory is required

Persistent Functional Language

System featuring a **persistent state** that can be accessed through **functional transactions**, and that can be used to:

- Implement database management systems
- Implement database programs
- Ad-hoc query and update databases

Persistent Functional Language

A state consists of a set of named bindings, e.g.:

```
users = [“alice”, “bob”]
length = \list -> …
```

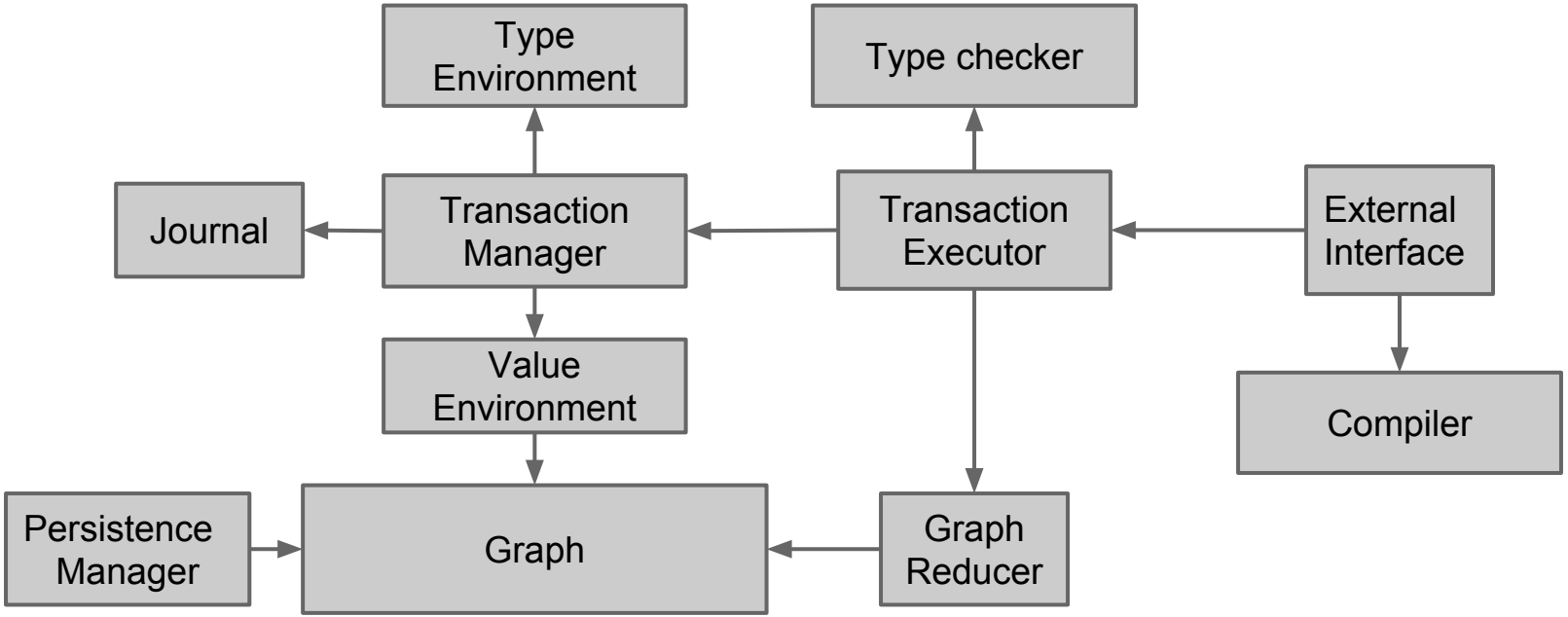
Transactions can:

- Change the value of bindings in the state
- Evaluate expressions with values from the state

Example Transaction

```
transaction {
    persistent {
        users = “eve” : users’
    }
    length list = case
        Nil -> 0
        Cons x xs -> 1 + length xs
    result = length users
}
```

Implementation



Typing Transactions

Hindley Milner + Algebraic Data Types

- Every binding in the state has a type
- Transactions can define new types:
  - Persistent: Available in this and future transactions
  - Non-persistent: Available only in the current transaction
- Transactions can refer to types defined in:
  - The current state, e.g.: `List`
  - The previous state, e.g.: `List’`
- An algebraic type definition creates constructor functions

Changing types

Assuming the state:

```
data User = User String Int String
users = [User “alice” 6382 “ZB”, ...]
```

We can change the structure of users:

```
transaction { persistent {
    data User = User String String
    users = map users’
        (User’ n pn pa -> User n (pn + “ “ + pa))
} }
```

Rule: A binding in the next state may not have a type that is being overwritten

Type Checking

Type checking is performed server side

- We can not trust the client
- The client does not have the type environment

Optimistic implementation:

- Type check on a snapshot of the type environment
- Assume types do not change during type checking
- Does not block access to the database

Ongoing and Future Work

- Benchmark based on TPC-C
- Language support for optimization strategies
- More optimization strategies
- Modelling a functional relational database