Persistent Functional Languages: towards Functional Relational Databases

Lesley Wevers
Supervised by Marieke Huisman & Maurice van Keulen

FMT & DB, University of Twente
Persistence

Data that outlives the execution of a program:

- Websites
- Information Systems
- Operating Systems
- Version Control Systems
- ...
...
Serialization to files on disk

Ad-hoc queries?
Schema transformations?
Concurrent operations?
...

Database Management Systems

Ad-hoc queries
Schema transformations
Concurrent operations
Share data between programs
Very large states

Query Optimization
Parallelism
Data Integrity
Enforce Constraints
Replication
...


Difficulties using DBMS’s

Forces the program into the database model:
- Data model mapping
- Type mapping

Verification is difficult:
- Type checking
- Testing
- Formal verification
Weak Points of DBMS’s

Largely fixed function, e.g.:

- Fixed data model
- Fixed data types
- Fixed index types

DBMS’s can’t really optimize database updates

- Database program execution is not under the control of the DBMS.
Persistent Languages

Ideal solution: Integrate a programming language with the features of a DBMS.

Not much success so far:
- Incompatible semantic models
- Optimization is a problem
Functional Persistent Languages

XQuery shows that functional languages are:

- Compatible with databases.
- Optimizable and parallelizable.

Using functional languages for the *updating* of databases has not really been explored.
Transactions

A transaction is a collection of operations, which execution satisfies the ACID properties:

- Atomicity
- Consistency
- Isolation
- Durability
Functional Transactions

type Transaction :: DB -> (DB, Result)
Functional Transaction Processing

tm :: DB -> [Transaction] -> [Result]
tm s (tx:txs) =
    let (ns, r) = tx(s) in
    r : (tm ns txs)
Functional Transaction Processing

data Maybe a = Just a | Nothing

incr s = (s+1, Nothing)
read s = (s, Just s)

> tm 0 [incr, incr, read] [Nothing, Nothing, Just 2]
Functional States

data Tree k v
  = Branch k (Tree k v) (Tree k v)
  | Leaf k v

```
Branch "bob"
  Branch "alice"
    Leaf "alice" 100
  Branch "eve"
    Leaf "bob" 50
    Leaf "eve" 25
    Leaf "dan" 25
```

Functional Updates

- Branch “bob”
  - Branch “alice” Leaf “alice” 100
  - Leaf “bob” 50
- Branch “eve” Leaf “eve” 25
  - Leaf “dan” 25
- Branch “bob”
  - Branch “eve” Leaf “eve” 50
Persistence

Simple persistence model:
● Journal transaction before executing.
● Recover state from latest snapshot by replaying journaled transactions from the initial state.
● Snapshot the state to clear the journal.
Constraints and Aborts

Enforce a constraint \( \text{check} : \text{DB} \rightarrow \text{Bool} \) over the state:

\[
\begin{align*}
\text{let } (ns, r) &= \text{tx}(s) \text{ in} \\
&\quad \text{if check(ns) then (ns, r) else (s, Error)}
\end{align*}
\]

Abort by returning the original state.
Transactions

This model satisfied the ACID properties:

- Atomic
- Consistent
- Isolated
- Durable

But how do we execute transactions in parallel?
Concurrent Transaction Execution

Idea: Evaluate states lazily.

update s = (map f s, Nothing)
contains k s = (s, contains k s)

tm (Branch ... ) [contains a, update, contains b, ... ]
Concurrent Transaction Processing

\[ \text{tm (Branch ...)} \ [\text{contains a, update, contains b, ...}] \]
Concurrent Transaction Processing

```
Branch[contains a, update, contains b, ...]
  /       /
/        /
Branch   Branch
  |       |
Leaf     Leaf
  |       |
Leaf     Leaf
  |       |
Leaf     Leaf
```

```
Concurrent Transaction Processing

- Cons
  - contains a
  - Branch
    - Branch
      - Leaf
    - Branch
      - Leaf
    - Branch
      - Leaf
  - tm
    - [update, contains b, ...]
Concurrent Transaction Processing

- Cons
  - contains a
  - Branch
    - Branch
      - Leaf
    - Branch
      - Leaf
  - Nothing
    - Cons
      - map f
    - tm
      - [contains b, ...]
Concurrent Transaction Processing

The diagram represents a data structure with the following properties:

- The topmost node is a `Cons` node.
- The `Cons` node contains `a`.
- Below `a`, there is a `Branch` node.
- The `Branch` node is connected to two `Leaf` nodes, each containing `b`.
- There is another `Branch` node below the `Leaf` nodes.
- The `Branch` node is connected to a `Cons` node.
- The `Cons` node contains `b`.
- The `Cons` node is the result of applying the `map f` function to the structure.
- The `Leaf` nodes are connected to the `Branch` nodes.

The diagram illustrates a recursive data structure where each node can either be a `Cons` or a `Leaf`, and the `Cons` nodes can contain values or further `Branch` nodes.
Concurrent Transaction Processing

contains a

map f

contains b

tm
Concurrent Transaction Processing

- Cons
  - contains a
  - Nothing
    - Branch
      - Branch
        - Leaf
          - map f
          - map f
        - Leaf
        - Leaf
      - map f
      - map f
      - contains b
      - tm
    - Leaf
  - Cons
  - Cons

[...]
Concurrent Transaction Processing

- Cons
  - Branch
    - Branch
      - Leaf
    - Branch
      - Leaf
    - Branch
      - Leaf
  - Branch
    - map f
      - Leaf
    - map f
      - Leaf
    - map f
      - Leaf
  - contains b
    - Cons
      - tm
        - [...]
Concurrent Transaction Processing

contains a

Nothing

.contains b

map f

map f

map f

[...]
Concurrent Transaction Processing

```
contains a

Branch
Branch
Branch
Branch

Leaf
Leaf
Leaf
Leaf
...
map f
map f
map f

Cons
Cons
Cons
Cons

Nothing

True

tm
```

[...]

Limitations of lazy evaluation

- Concurrency is limited by data dependencies
e.g. if c then a else b
can not be evaluated until c is evaluated
- Transaction functions must be total
- Memory requirements
Future work: Memoization

Remember results of function applications:

- Optimistic execution & retrying transactions
- Aggregate functions:
  - sum, min, max, …
  - constraint checks
- Materialized views
Persistent Functional Languages

ACID-State for Haskell implements many of these ideas, however:

- There are no ad-hoc transactions:
  - We can’t do schema changes on the fly
  - We can’t share the state with other programs
- GHC not optimized for this use case
  - State is limited to main memory
  - Task scheduling not optimized for latency
Persistent Functional Language

Goals:

● Functional transaction processing
● Ad-hoc transactions
● Stored transactions (domain specific API)
Binding Model

The state consists of a set of bindings.

A transaction can atomically:

● Create, update and delete bindings
● Evaluate an expression in the current state
Persistent Functional Language

Demo
Conclusions

We have seen:
● Functional languages for transaction processing
● Persistent functional languages

Future work:
● Optimistic concurrency control & Memoization
● Online Schema Changes
● Modelling relational databases
● Verification of constraints
Functional Persistent Languages

New possibilities:
- New methods of concurrency control
- Verification of database software
Concurrent Transaction Execution

Latency > Throughput
Concurrency > Parallelism
Transactions should be able to make progress.
Avoid transactions blocking each other.
Blocking: Heavy computations, IO
Functional States

We model states using algebraic data types, e.g.:

```haskell
data List a
  = Cons a (List a)
  | Nil

data Tree k v
  = Branch k (Tree k v) (Tree k v)
  | Leaf k v
```
Combined Approach

```java
var state = new AtomicReference(initial_state)
def execute(tx : S -> (S, R, R)) : R = {
    var ns, r, f
    do {
        val s = state.get()
        (ns, r, f) = tx(s)
        reduce(f)
    } while(!state.compareAndSet(s, ns))
    return r
}
```
Application:
Online Schema Transformations

Current database systems can only do schema transformations offline.

We want to perform schema changes lazily
Future Work:
Modelling Relational Databases

Model: relations, relational operations, indices, constraints, …
Querying using list comprehensions
Specifying updates conveniently
Concurrent updates
Typing relational operations
Future Work:
Verifying Database Software

Runtime constraint verification to eliminate runtime checks.

Example?
Future Work: Optimistic Execution

var state = new AtomicReference(initial_state)
def execute(tx : S -> (S, R, F)) : R = {
    var ns, r, f
    do {
        val s = state.get()
        (ns, r, f) = tx(s)
        reduce(f)
    } while(!state.compareAndSet(s, ns))
    return r
}