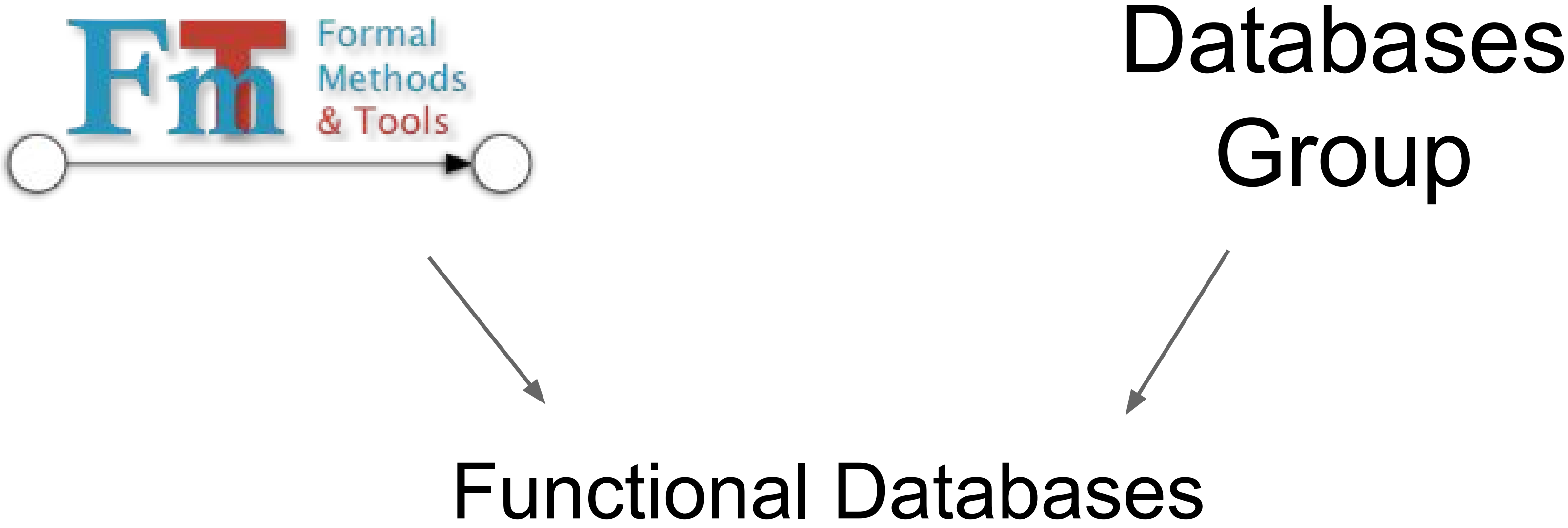
Lazy Transactional Operations on Bulk Data

Lesley Wevers

Marieke Huisman   Maurice van Keulen

CWI, 23 February 2015



FMT Formal Methods & Tools

Databases Group

Functional Databases

Functional Programming

Functional Programming

Imperative programming

- Execute instructions that mutate state

Functional programming

- Evaluate expressions that produce values
  - users.map(u => u.name)
  - users.filter(u => u.age < 18)
  - users.reduce(max, u => u.age)
  - users.orderBy(u => u.name)

Pure functional programming

Pure functions: **stateless** and **deterministic**

- Lazy evaluation (call-by-need)
- Concurrency and parallelism
- Partial evaluation
- Rewriting
- Memoization

FP in the Context of Databases

Functional languages are used for querying:

- XQuery
- Relational algebra

Can we also use functional languages to optimize transaction processing?

Applications

Lazy non-blocking schema transformations

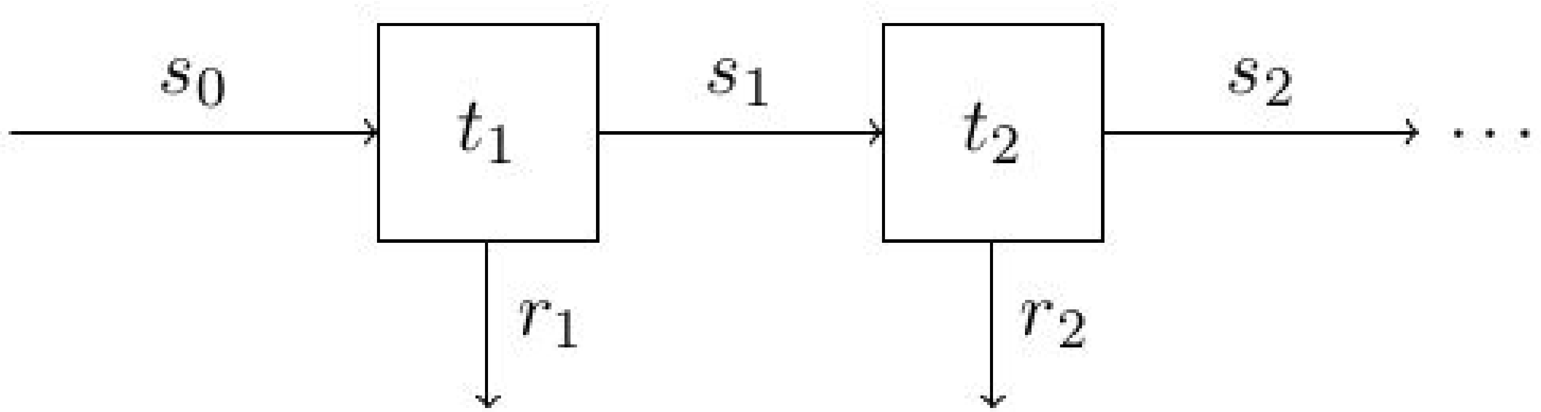
- Immediate access to results
- Lazy transformations are composable

Persistent functional languages

- Flexible data modelling
- Optimization of transactions

Functional Transaction Processing

Functional Transaction Processing



Persistent Functional Language

```
users = relation(name, age)
```

Persistent Functional Language

```
users = users
  + (name: “alice”, bday: 26/02/1987)
  + (name: “bob”, bday: 08/09/1985)
```

Persistent Functional Language

users

name	bday
alice	26/02/1987
bob	08/09/1985

Persistent Functional Language

```
users.map(name, age: years(now - bday))
```

name	age
alice	27
bob	29

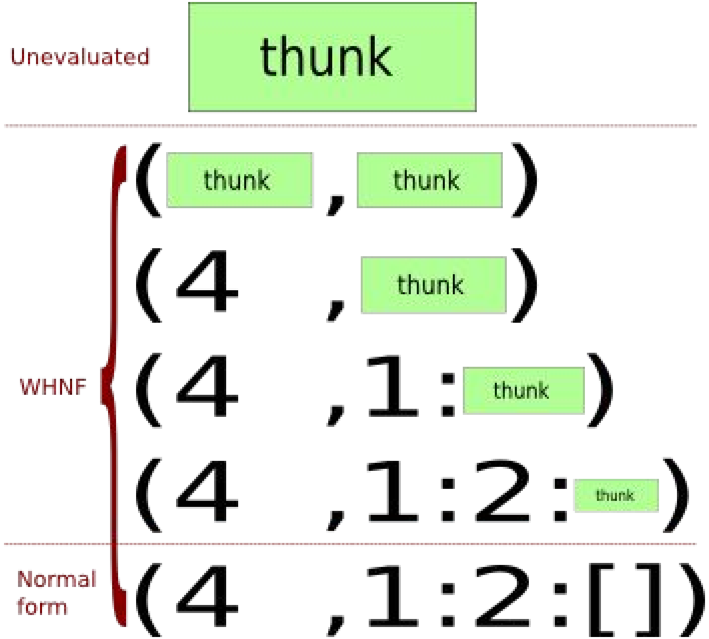
Persistent Functional Language

```
users = users.map(name, btime: bday.toTimestamp)
```

Lazy Transaction Processing

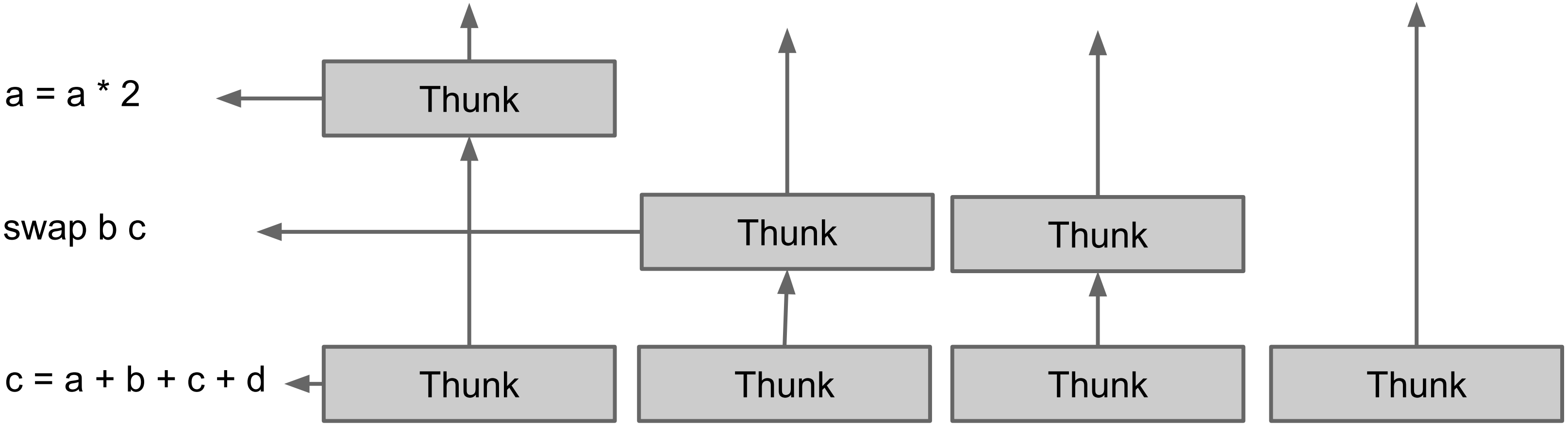
Lazy Evaluation

Suspend computations which results are not immediately needed.



Laziness in Mutable Databases

a	b	c	d
95	43	72	56



a = a * 2

swap b c

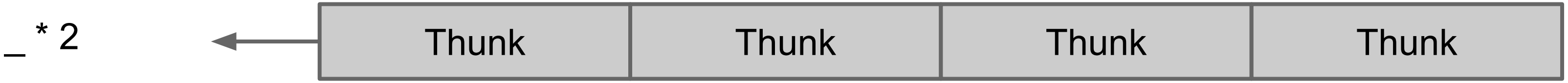
c = a + b + c + d

Laziness in Functional Databases

	a	b	c	d
	95	43	72	56
a = a * 2	Thunk			
swap b c		Thunk	Thunk	
c = a + b + c + d			Thunk	

Lazy Bulk Operations

a	b	c	d
95	43	72	56

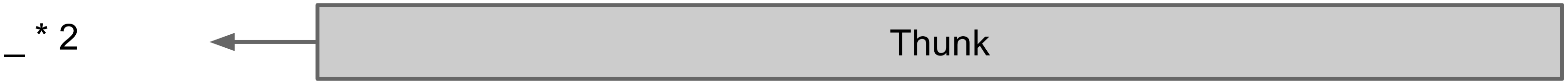


_ * 2

Thunk Thunk Thunk Thunk

Divide and Conquer Lazy Operations

a	b	c	d
95	43	72	56

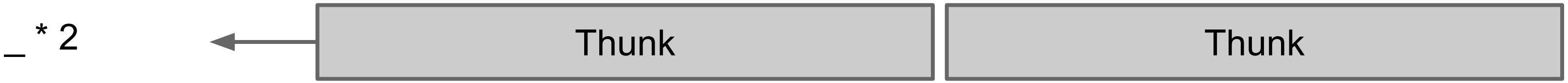


_ * 2

Thunk

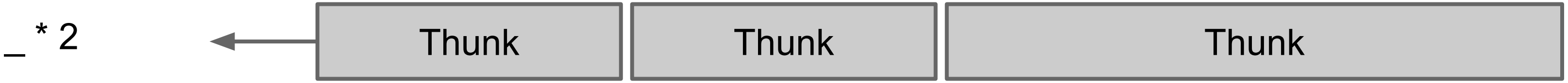
Divide and Conquer Lazy Operations

a	b	c	d
95	43	72	56



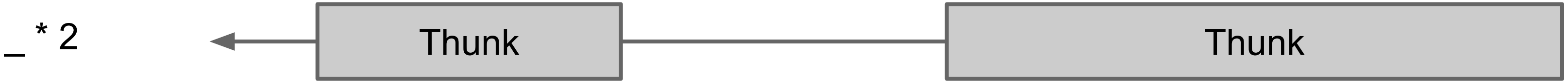
Divide and Conquer Lazy Operations

a	b	c	d
95	43	72	56



Divide and Conquer Lazy Operations

a	b	c	d
95	86	72	56



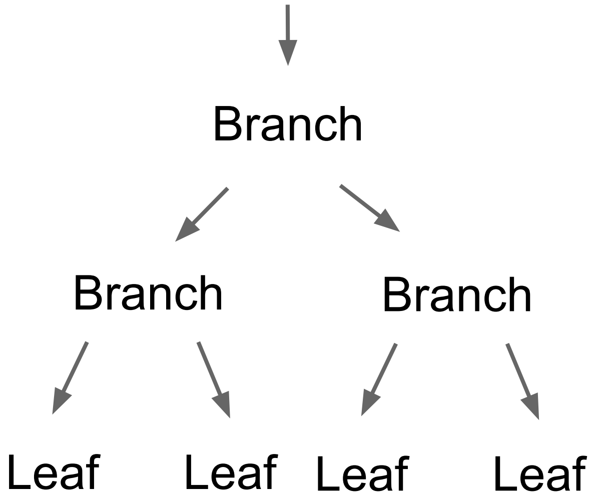
Why Lazy Transactions?

- Provides parallelism
- Allows for lazy bulk operations
- Improved performance*
    - Temporal load balancing
    - Avoiding work
    - Locality of reference
    - Reduced contention footprint

* Jose M. Faleiro, Alexander Thomson. Lazy Evaluation of Transactions in Database Systems. VLDB, 2014.

Implementation

Persistent data structures



Branch

Branch

Branch

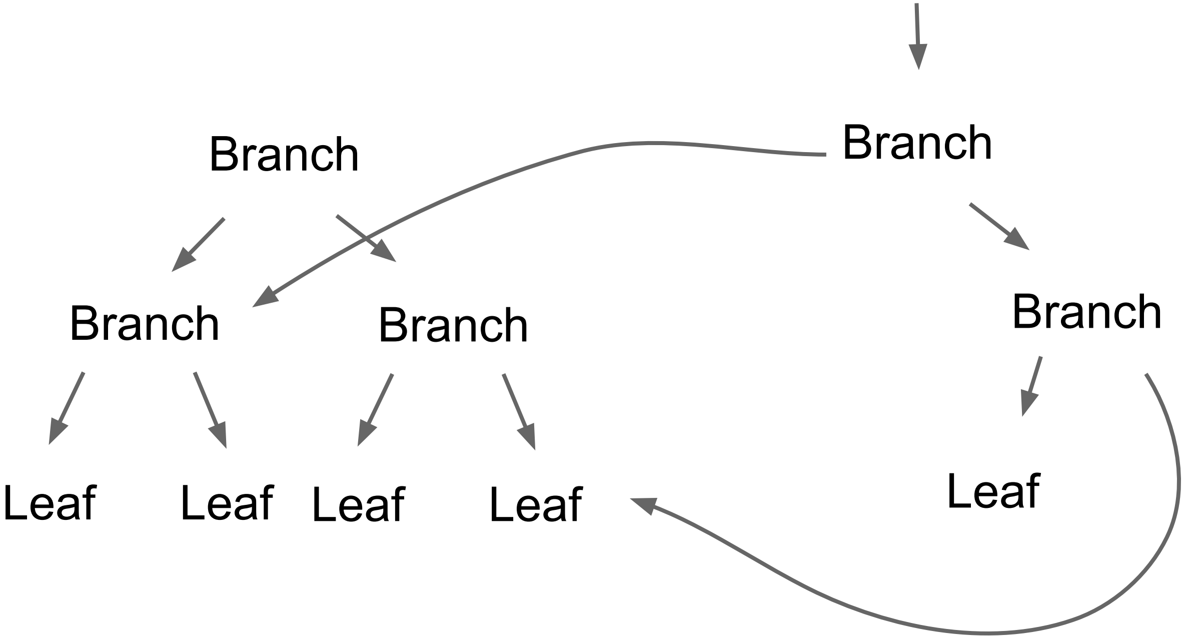
Leaf

Leaf

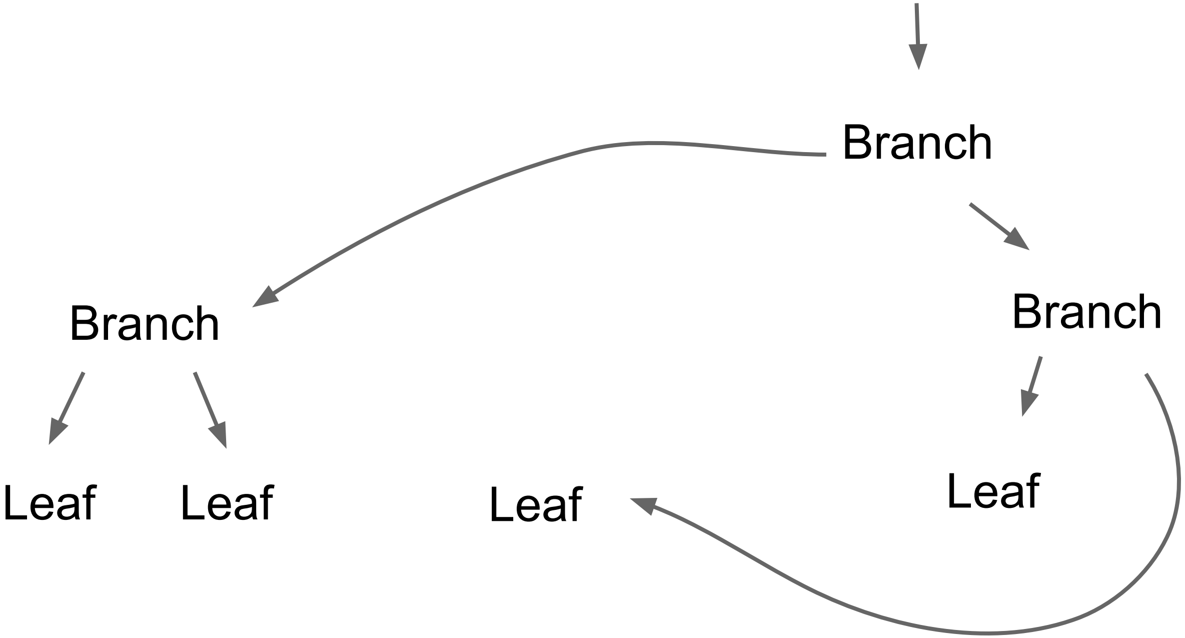
Leaf

Leaf

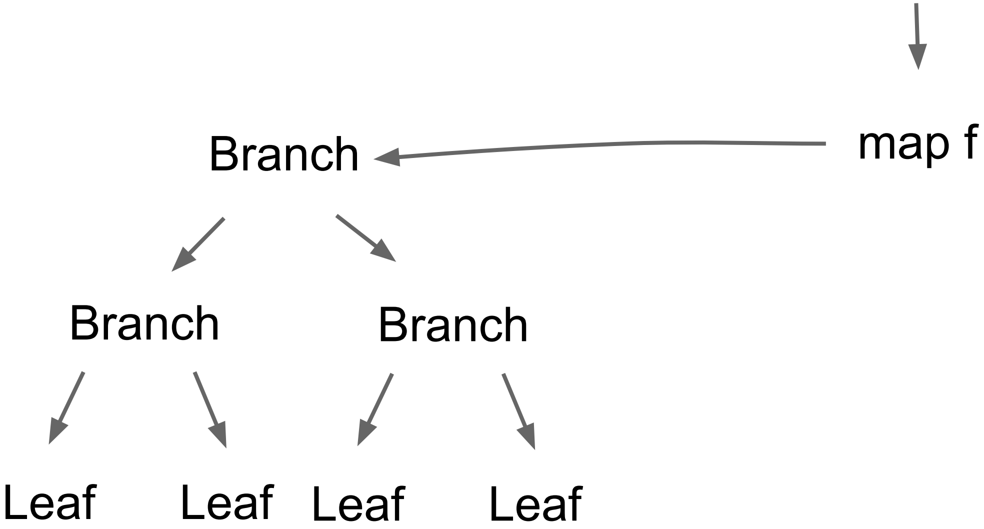
Persistent data structures



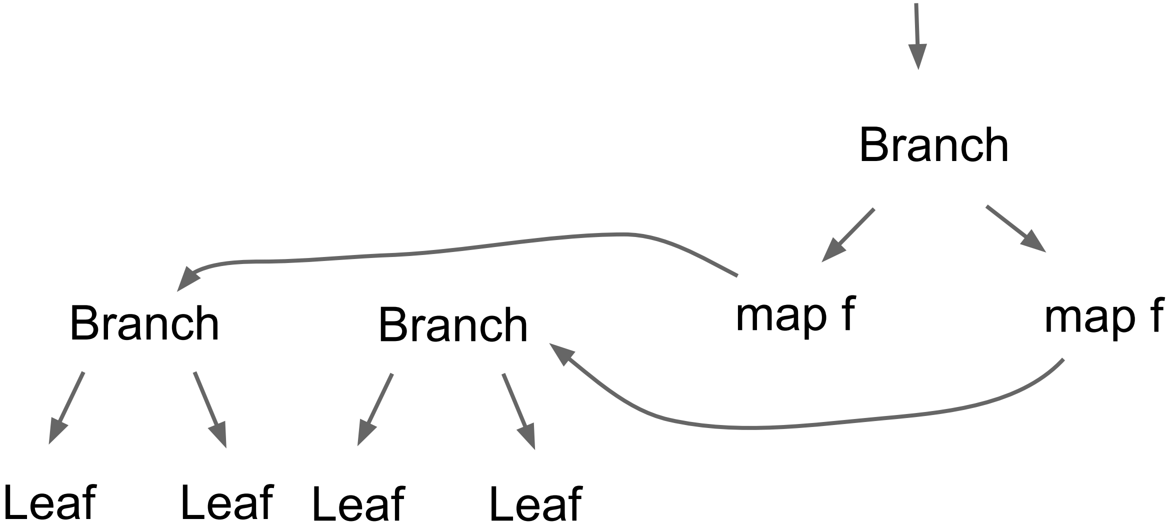
Persistent data structures



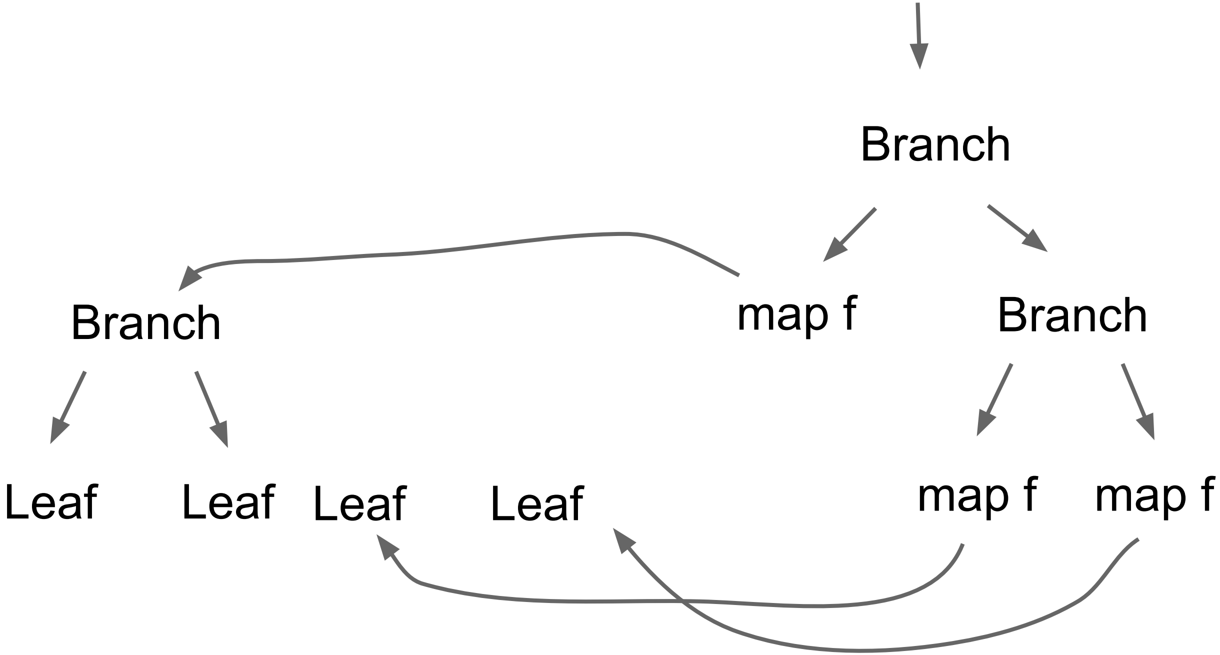
Graph rewriting



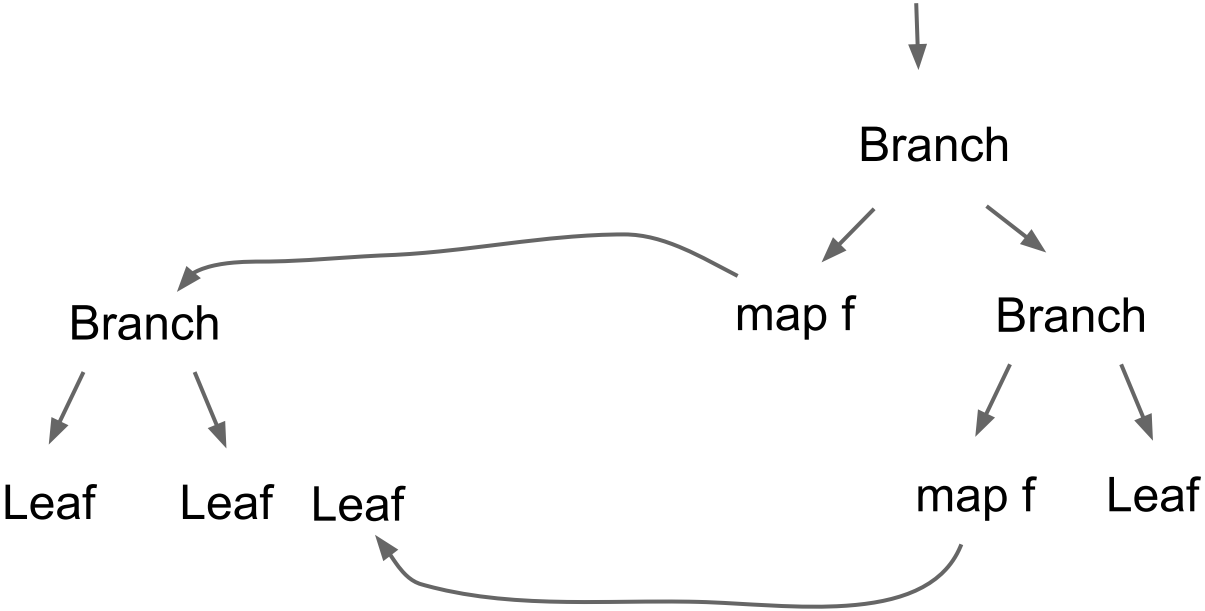
Graph rewriting



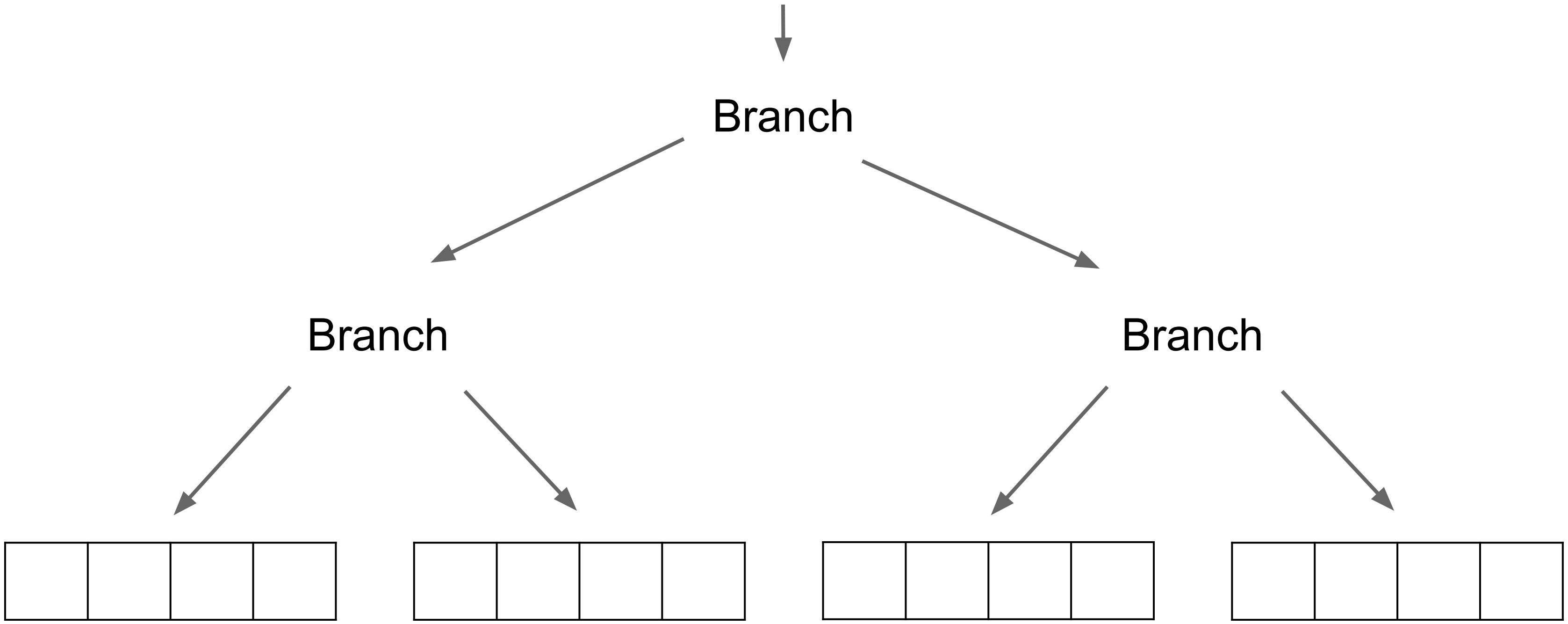
Graph rewriting



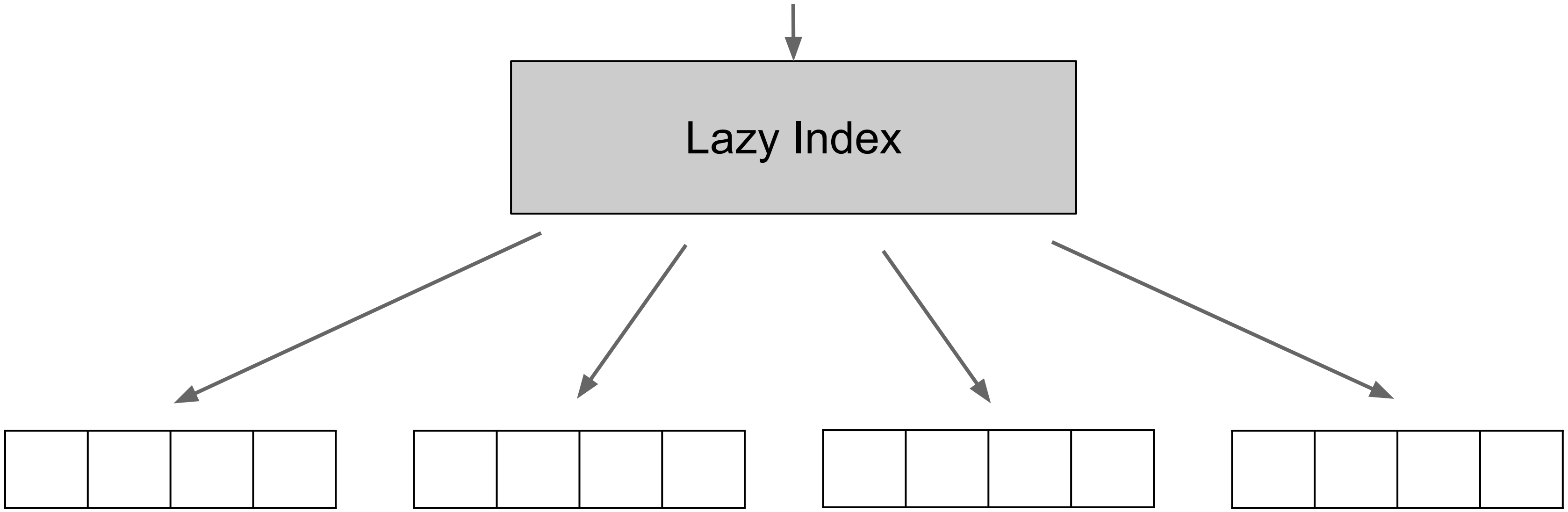
Graph rewriting



Bulk Data



Bulk Data



On-disk Storage

Storage orientation

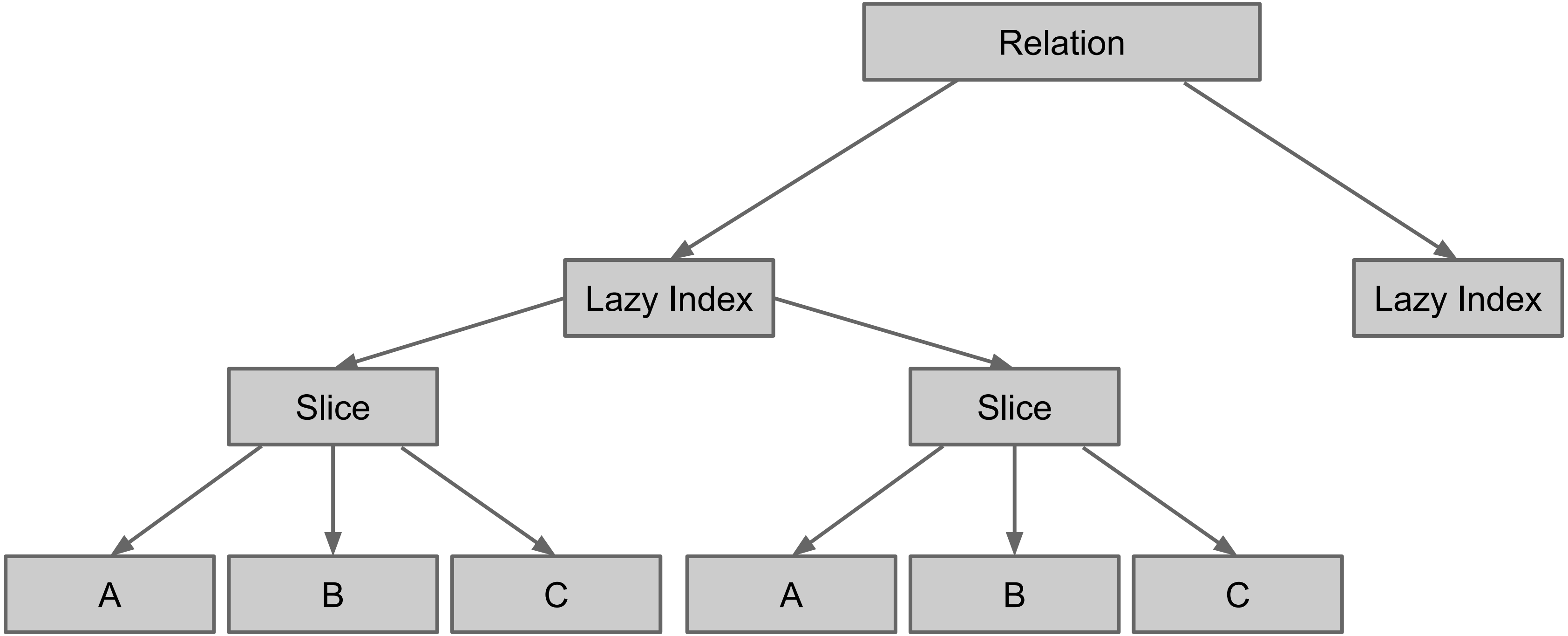
Column oriented storage is good for:

- Projections
- Aggregates
- Single-column updates

Row oriented storage is good for:

- Inserts, deletes and multi-column updates

Architecture



Durability

Idea sketch:

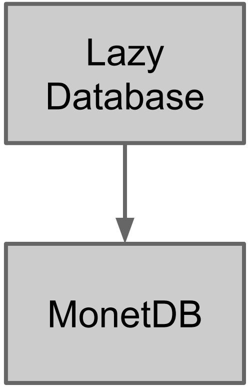
- Journal transaction functions
- Take a snapshot of the index at regular intervals
- When taking a snapshot, commit the MonetDB state to flush blocks to disk.

Using MonetDB

Using MonetDB for column storage

Idea:

- MonetDB stores columns
- MonetDB performs bulk operations
- We provide laziness



Questions

Can MonetDB handle this use case?

- We may create many temporary columns
- What size of columns should we use?
- How would MonetDB cope with OLTP workloads in this approach?

Questions

How do we perform joins?

- MonetDB does not know if there are operations pending on a node.
- Solution sketch: We request data from MonetDB, and perform the join ourselves.
- Is there a better approach?

Questions

Or should we use alternative approaches:

- Implement laziness in MAL?
- Implement laziness inside MonetDB?
- Use a lower level storage system?
- Build our own storage system?

Functional languages for databases

- Integrate programming and databases
  - Optimize transactions
  - Flexible data modelling
- Immutable data structures provide isolation
- Lazy database updates
  - Concurrency control through data dependencies
  - Non-blocking schema transformations

Online Schema Transformations

Basic Schema Changes

Creating, removing and changing:

- Relations
- Columns
- Indices
- Constraints

Complex Schema Changes

Complex transformations:

- Changing the type of primary key
- Changing the cardinality of relationships
- Splitting and merging of tables
- Moving data between tables
- Any combination of basic transformations

Current database systems

	PostgreSQL	MySQL
Simple Changes	Mixed results	Mixed results
Complex Changes	Correct but blocking	Online but incorrect

Solution Direction

Lazy schema transformations:

- A transformation is a view on the old schema.
- Transform data on demand when accessed.

How this better meets the requirements:

- Updates are immediately visible.
- Lazy schema changes are composable.