A Concurrent Persistent Functional Language

Towards Practical Functional Databases

Lesley Wevers [1] Ander de Keijzer [2] Marieke Huisman [1]

[1]University of Twente, Netherlands

[2]Windesheim University of Applied Sciences, Netherlands

IFL 2012, Oxford, UK
1 September, 2012

Transactions

Transactions

- There is a global *state*.

Transactions

Transactions

- There is a global *state*.
- A *transaction* is a collection of *operations* on the global state.

Transactions

Transactions

- There is a global *state*.
- A *transaction* is a collection of *operations* on the global state.

ACID Properties

Atomic: Either *all or none* of the operations of a transaction are executed.

Transactions

Transactions

- There is a global *state*.
- A *transaction* is a collection of *operations* on the global state.

ACID Properties

Atomic: Either *all or none* of the operations of a transaction are executed.

Consistent: After each transaction, the system is in a consistent state.

Transactions

Transactions

- There is a global *state*.
- A *transaction* is a collection of *operations* on the global state.

ACID Properties

Atomic: Either *all or none* of the operations of a transaction are executed.

Consistent: After each transaction, the system is in a consistent state.

Isolated: It seems as if the transactions are executed one by one (serializability and recoverability).

Transactions

Transactions

- There is a global *state*.
- A *transaction* is a collection of *operations* on the global state.

ACID Properties

Atomic: Either *all or none* of the operations of a transaction are executed.

Consistent: After each transaction, the system is in a consistent state.

Isolated: It seems as if the transactions are executed one by one (serializability and recoverability).

Durable: The effect of a transaction is permanent.

Transaction Processing Systems

Examples

- Banking

Transaction Processing Systems

Examples

- Banking
- Ticket reservation

Transaction Processing Systems

Examples

- Banking
- Ticket reservation
- Inventarisation systems

Transaction Processing Systems

Examples

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

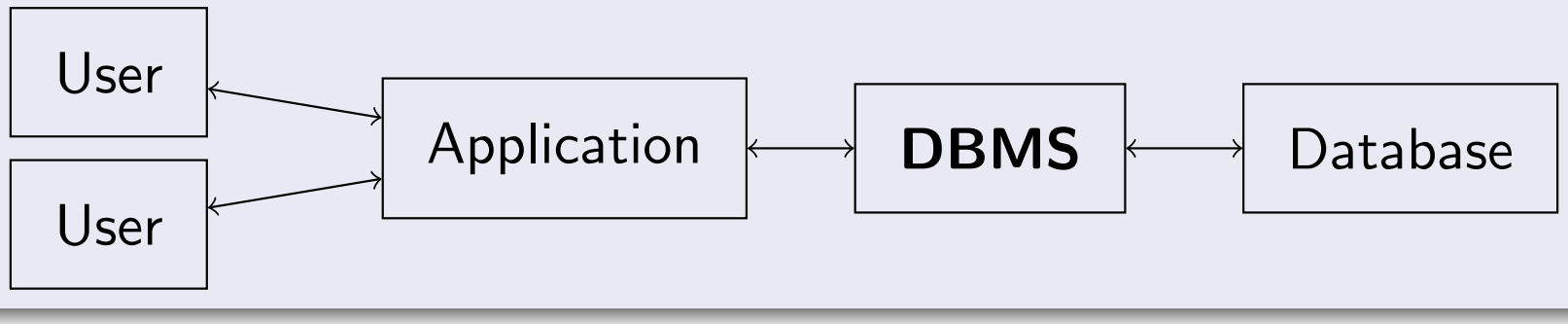
Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.
- Everyone wants a quick response.
- We want to handle a lots of data.

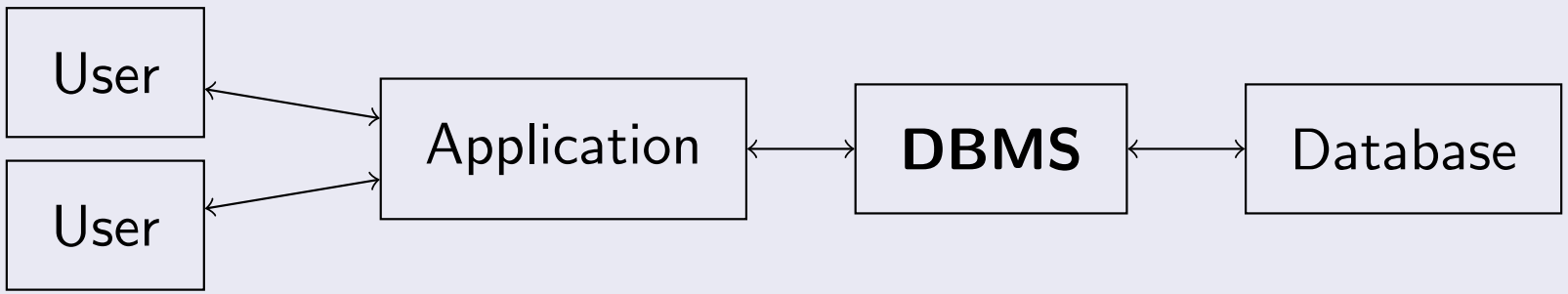
Current Approach

Traditional Architecture



Current Approach

Traditional Architecture

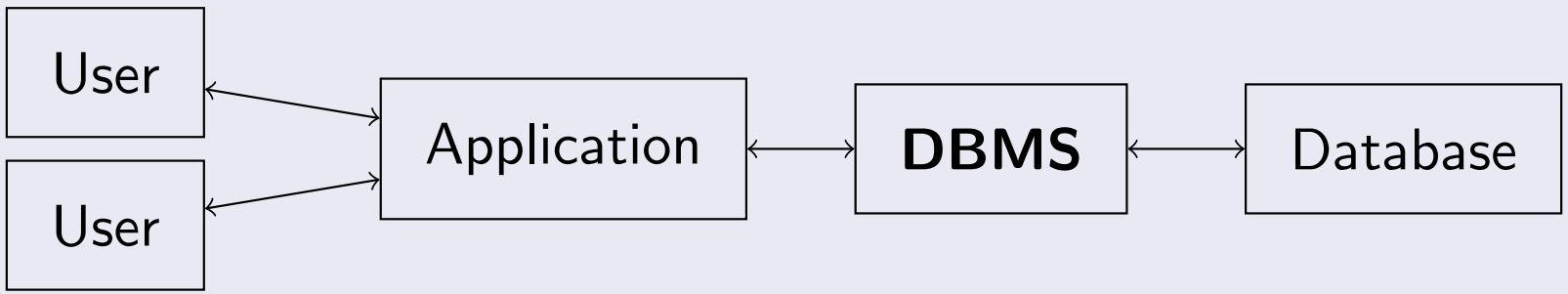


Database Management System

- Optimised to handle large amounts of data.

Current Approach

Traditional Architecture

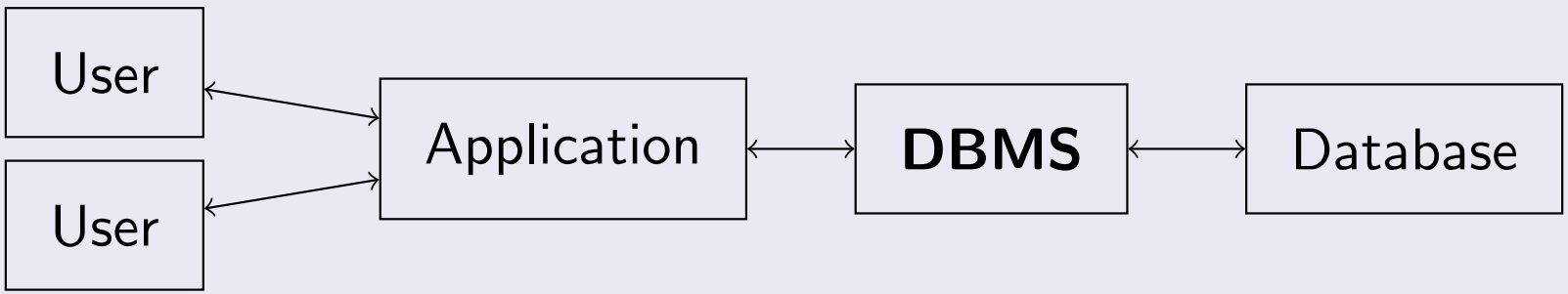


Database Management System

- Optimised to handle large amounts of data.
- Interface to query and manipulate data.

Current Approach

Traditional Architecture

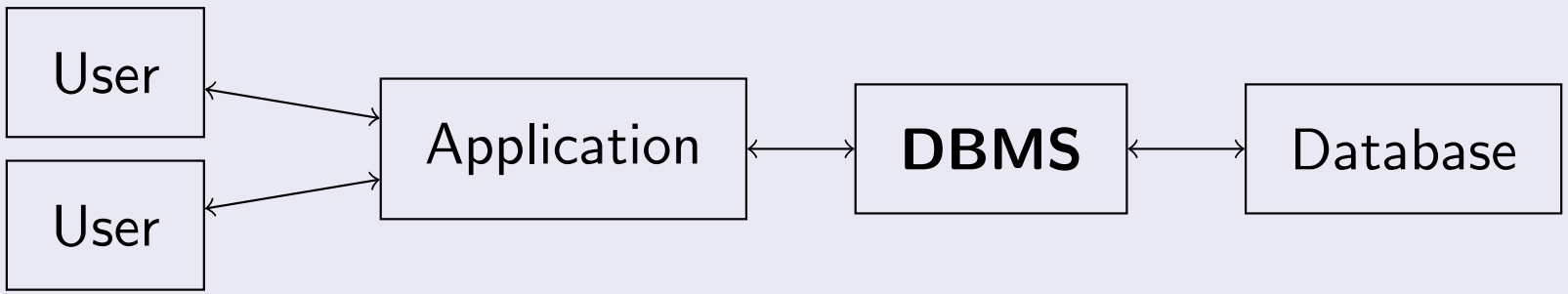


Database Management System

- Optimised to handle large amounts of data.
- Interface to query and manipulate data.
- Transactions.

Current Approach

Traditional Architecture



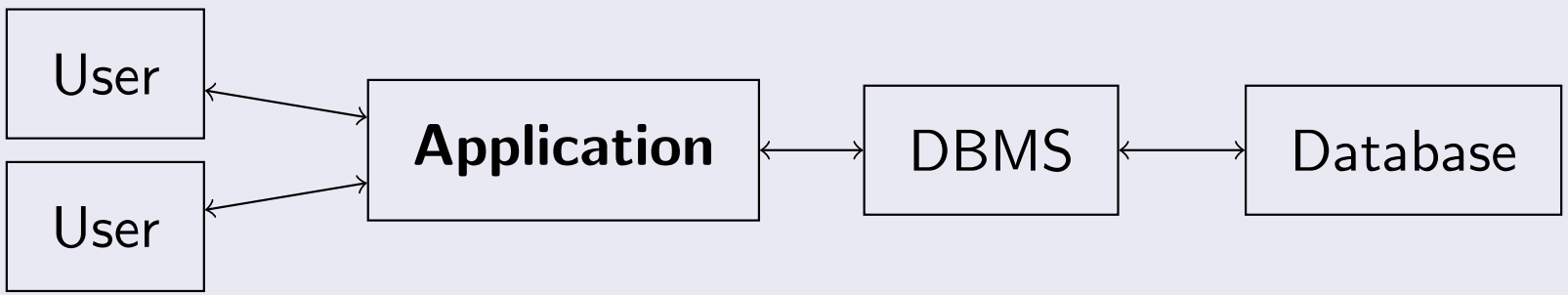
Database Management System

- Optimised to handle large amounts of data.
- Interface to query and manipulate data.
- Transactions.

Most DBMS's only partially support isolation of transactions due to efficiency reasons.

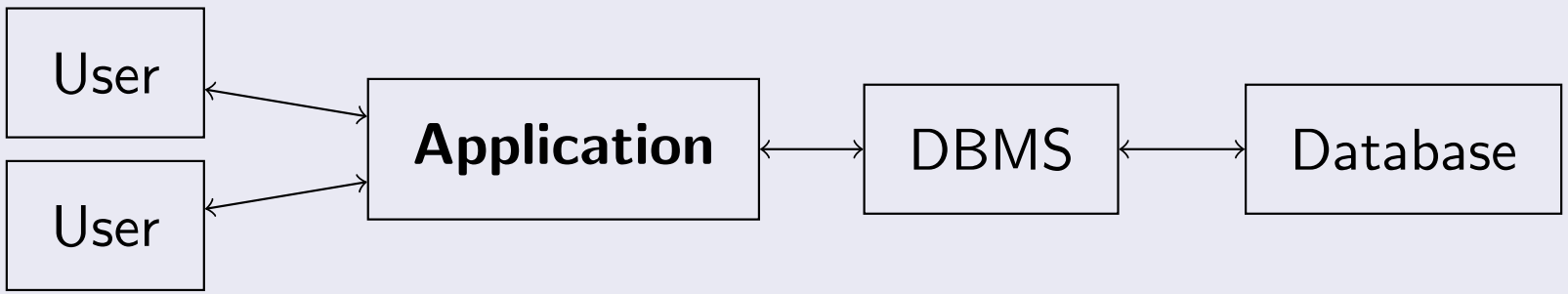
Current Approach

Traditional Architecture



Current Approach

Traditional Architecture

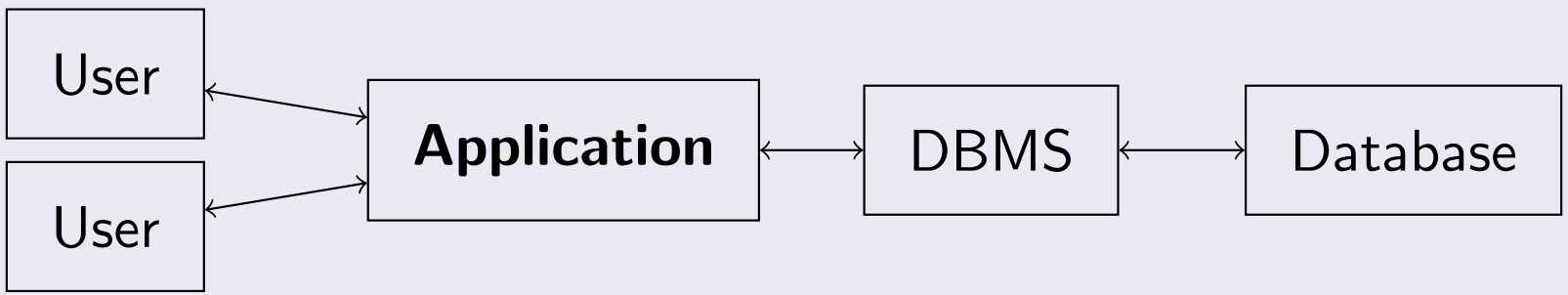


Application

- Interface to the outside world.

Current Approach

Traditional Architecture

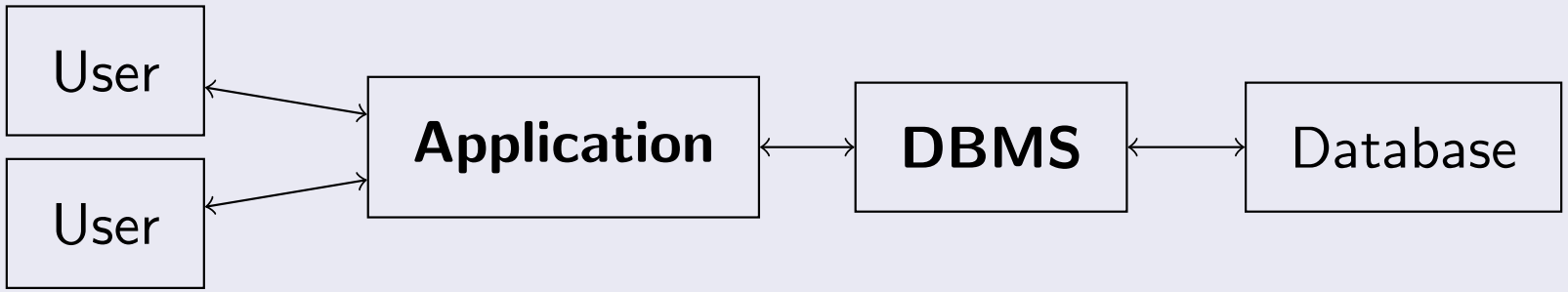


Application

- Interface to the outside world.
- Can enforce additional security constraints.

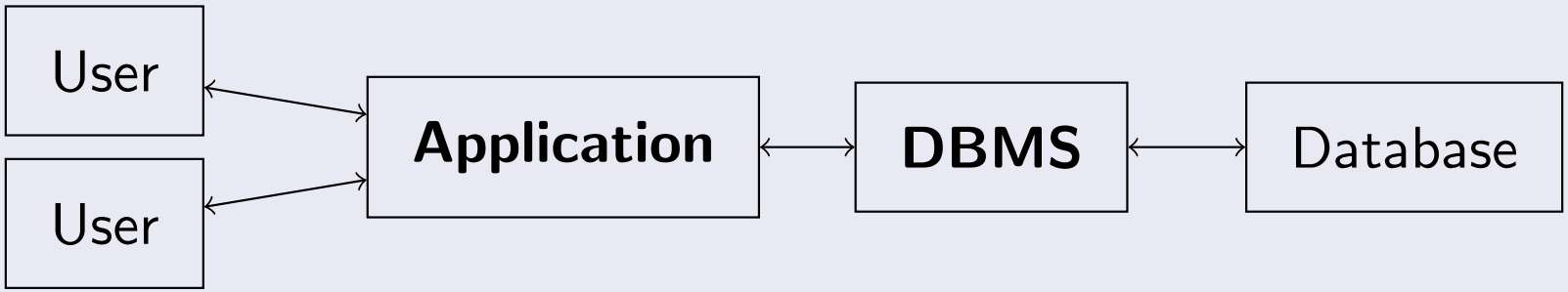
Current Approach

Traditional Architecture



Current Approach

Traditional Architecture

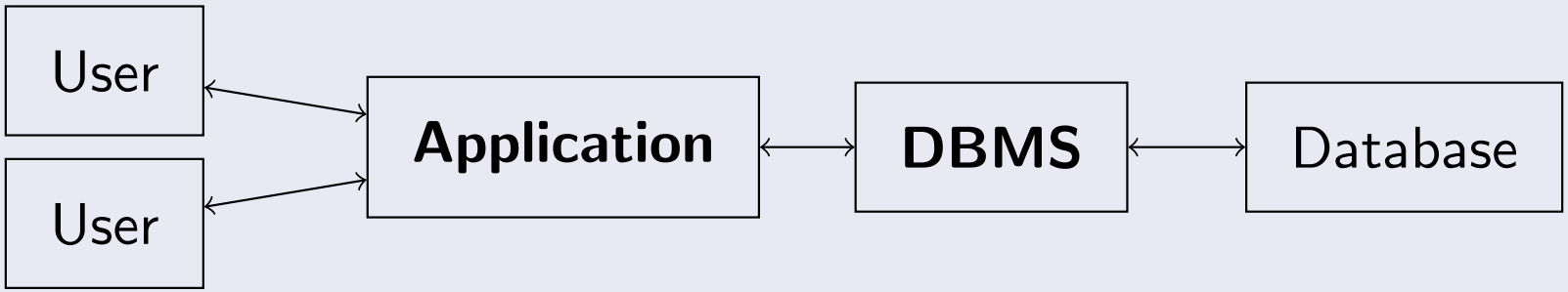


Limitations

- Application and DBMS have different type system.

Current Approach

Traditional Architecture

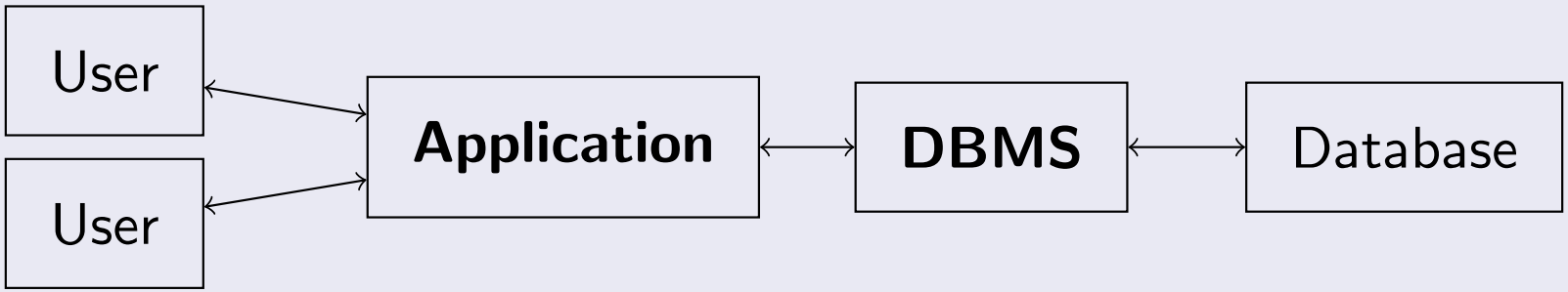


Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.

Current Approach

Traditional Architecture

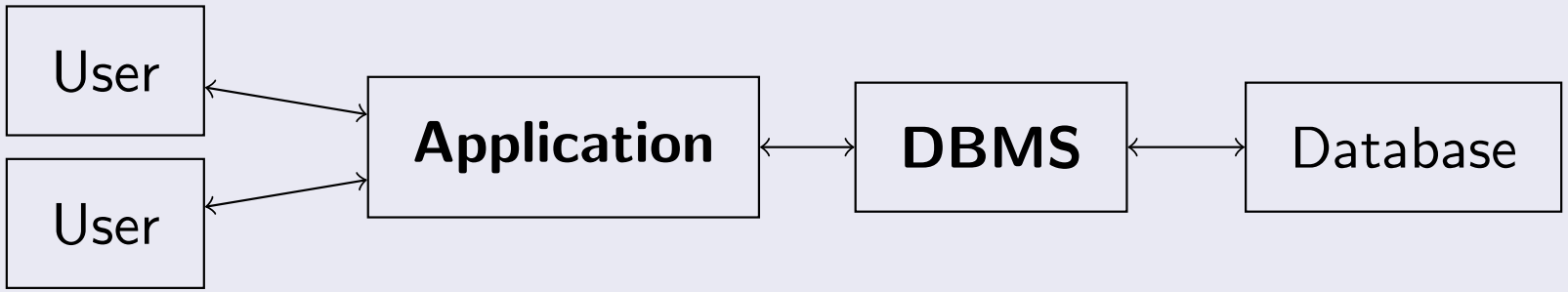


Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.

Current Approach

Traditional Architecture

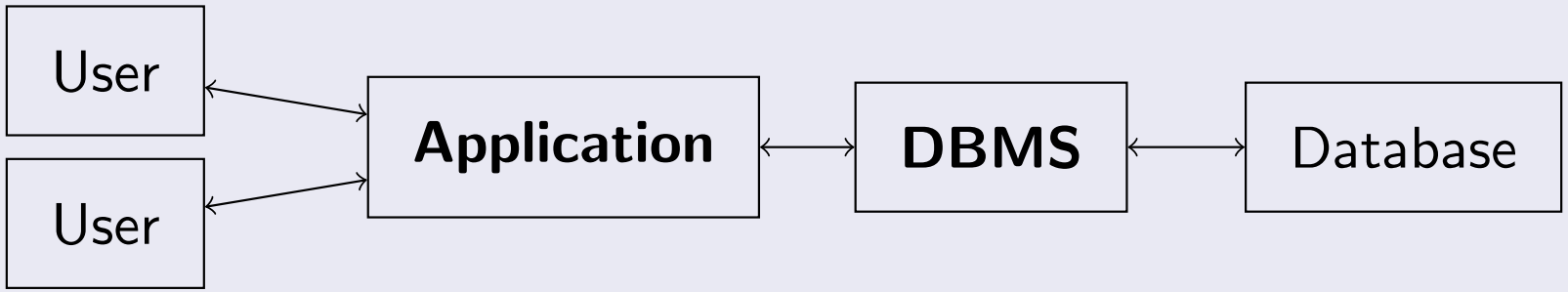


Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.
- DBMS's are vulnerable to command injection attacks.

Current Approach

Traditional Architecture



Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.
- DBMS's are vulnerable to command injection attacks.
- System as a whole is difficult to verify.

Functional Transaction Processing

Functional Transaction Processing

- A *transaction function*: $State \rightarrow State \times Result$.

Functional Transaction Processing

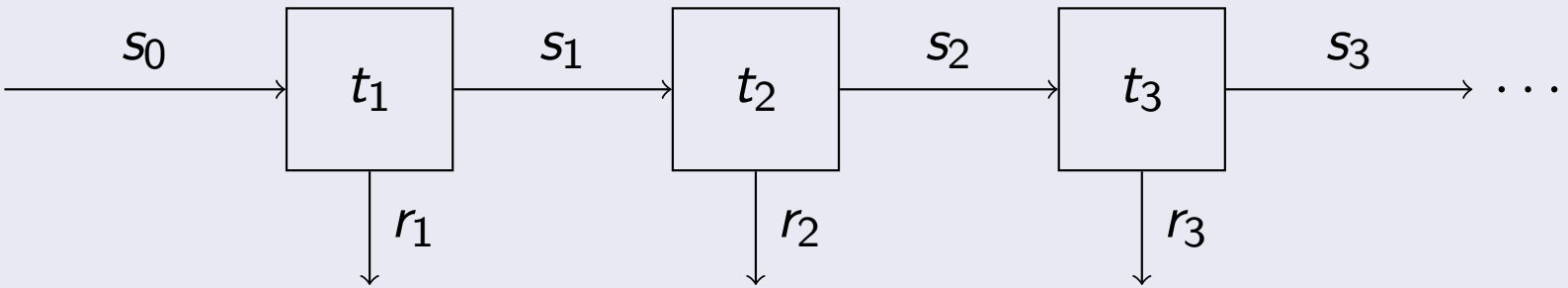
Functional Transaction Processing

- A *transaction function*: $State \rightarrow State \times Result$.
- A *transaction manager*: $State \times [Transaction] \rightarrow [Result]$.

Functional Transaction Processing

Functional Transaction Processing

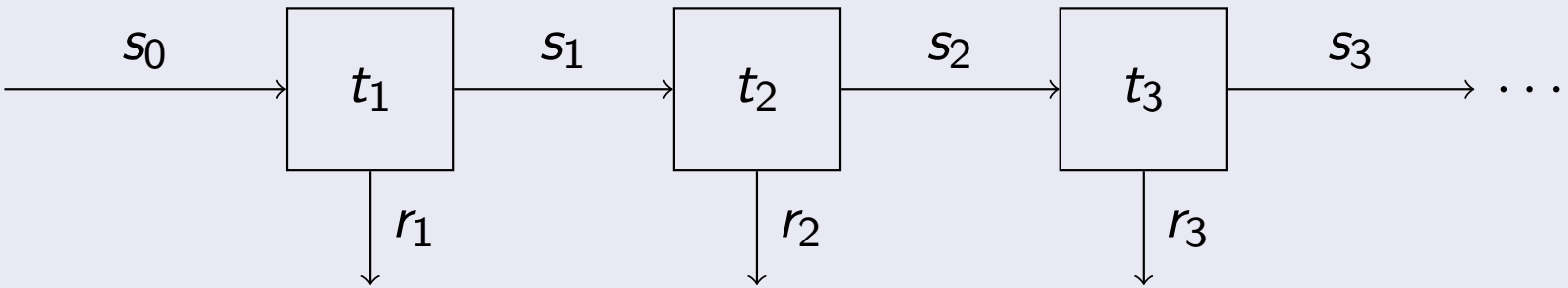
- A *transaction function*: $State \rightarrow State \times Result$.
- A *transaction manager*: $State \times [Transaction] \rightarrow [Result]$.



Functional Transaction Processing

Functional Transaction Processing

- A *transaction function*: $State \to State \times Result$.
- A *transaction manager*: $State \times [Transaction] \to [Result]$.



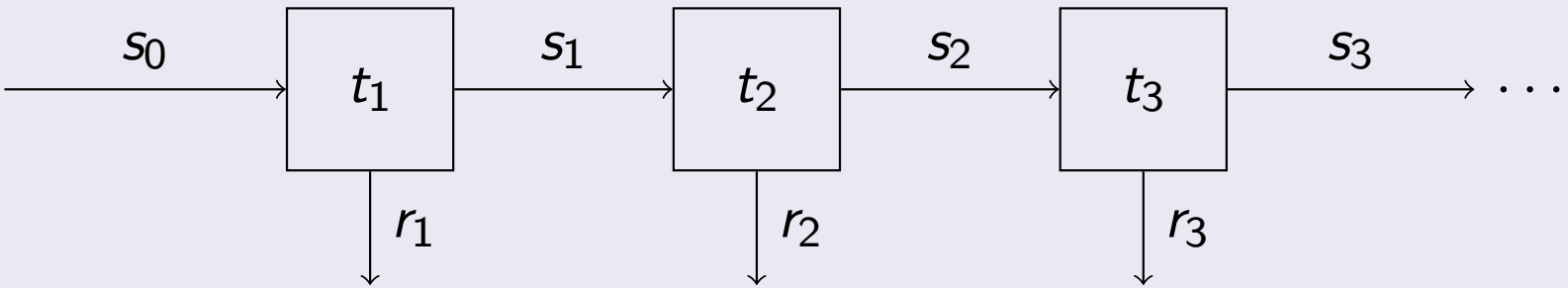
Correctness (ACID)

- Atomicity and isolation hold trivially for *total* transactions.

Functional Transaction Processing

Functional Transaction Processing

- A *transaction function*: $State \rightarrow State \times Result$.
- A *transaction manager*: $State \times [Transaction] \rightarrow [Result]$.



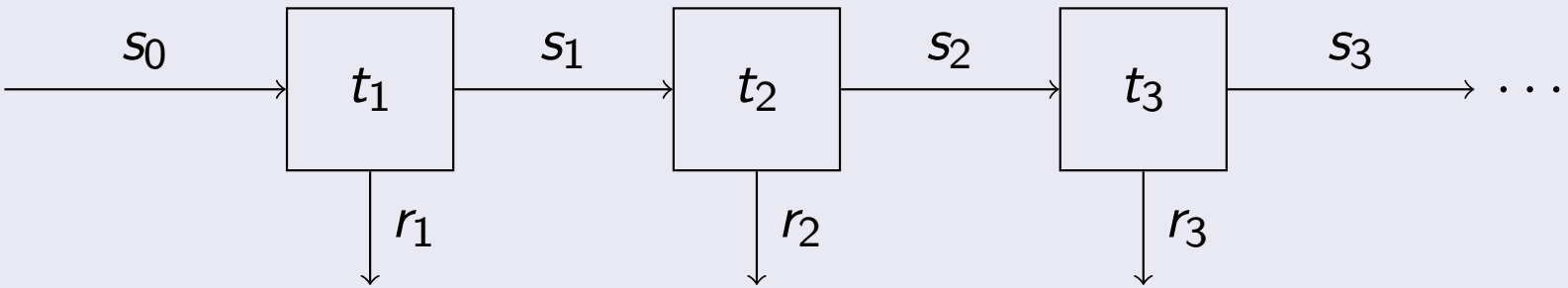
Correctness (ACID)

- Atomicity and isolation hold trivially for *total* transactions.
- A transaction must enforce consistency rules.

Functional Transaction Processing

Functional Transaction Processing

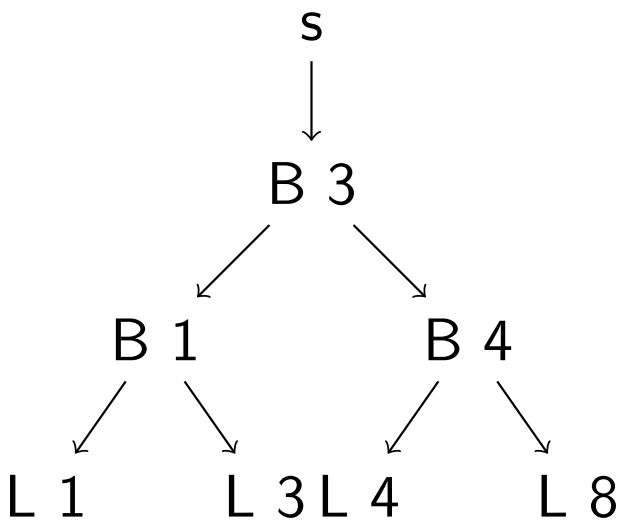
- A *transaction function*: $State \to State \times Result$.
- A *transaction manager*: $State \times [Transaction] \to [Result]$.



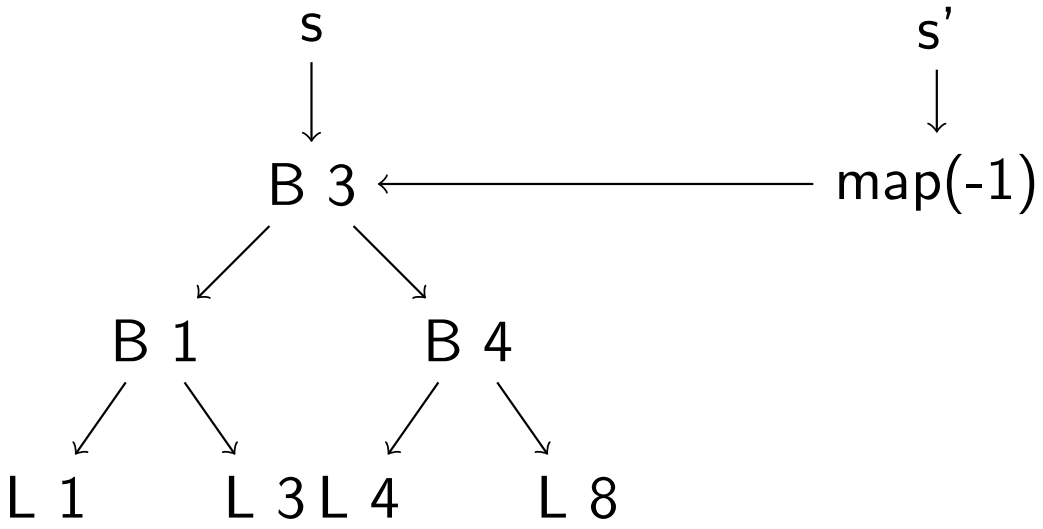
Correctness (ACID)

- Atomicity and isolation hold trivially for *total* transactions.
- A transaction must enforce consistency rules.
- Implementation can easily support durability.

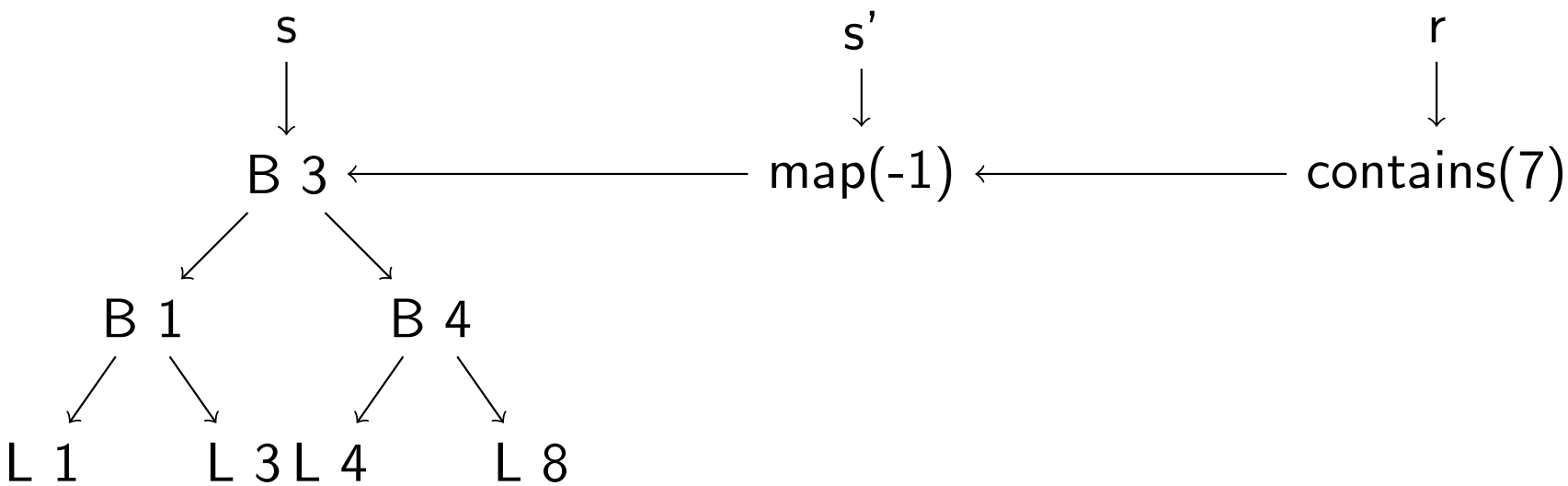
Functional Transaction Processing



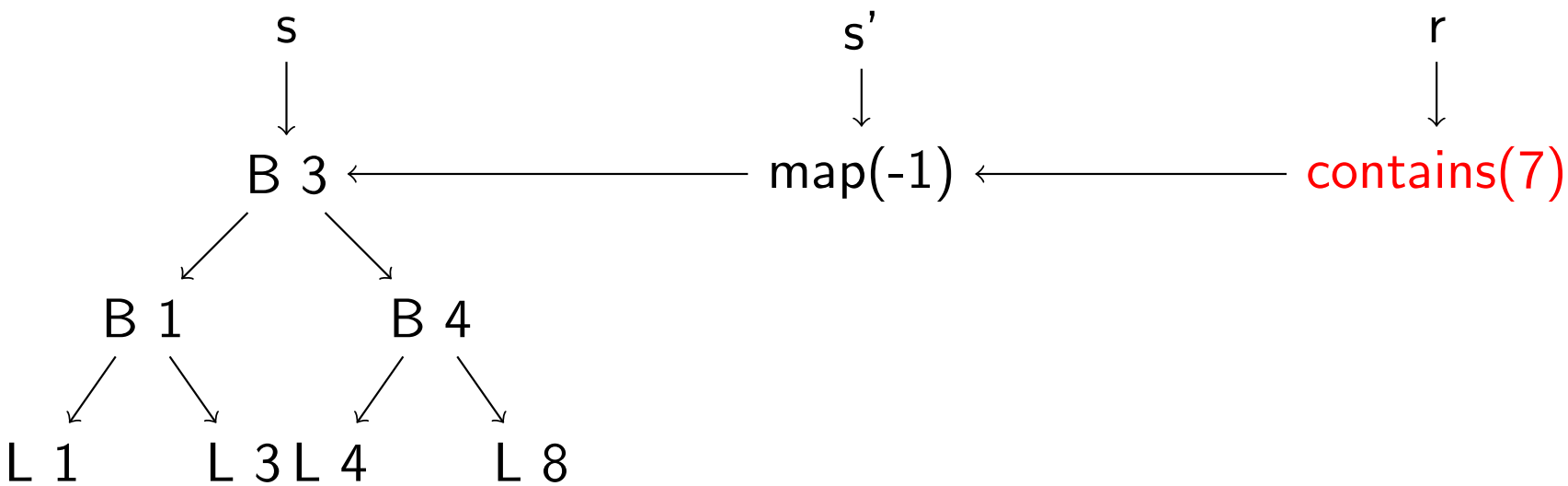
Functional Transaction Processing



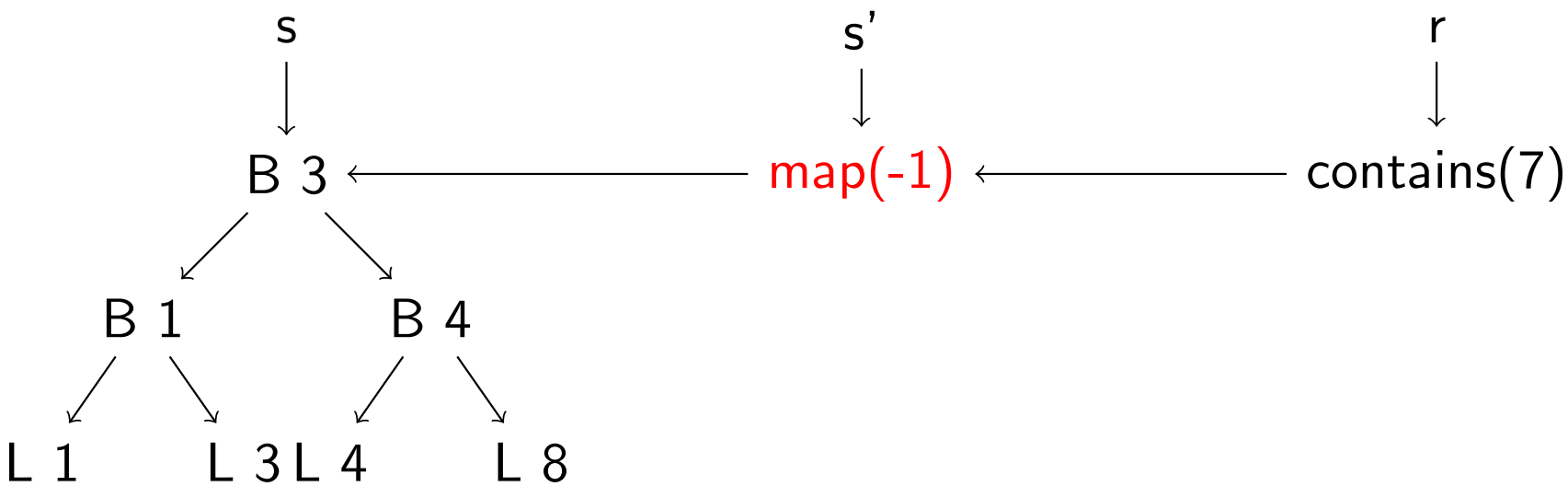
Functional Transaction Processing



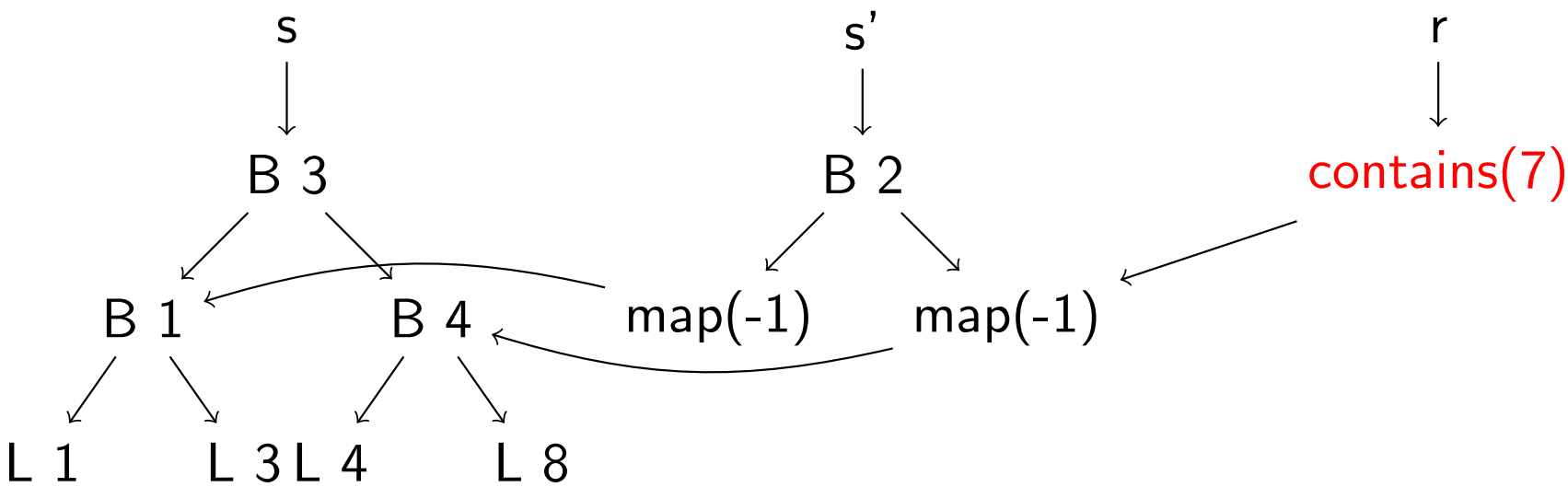
Functional Transaction Processing



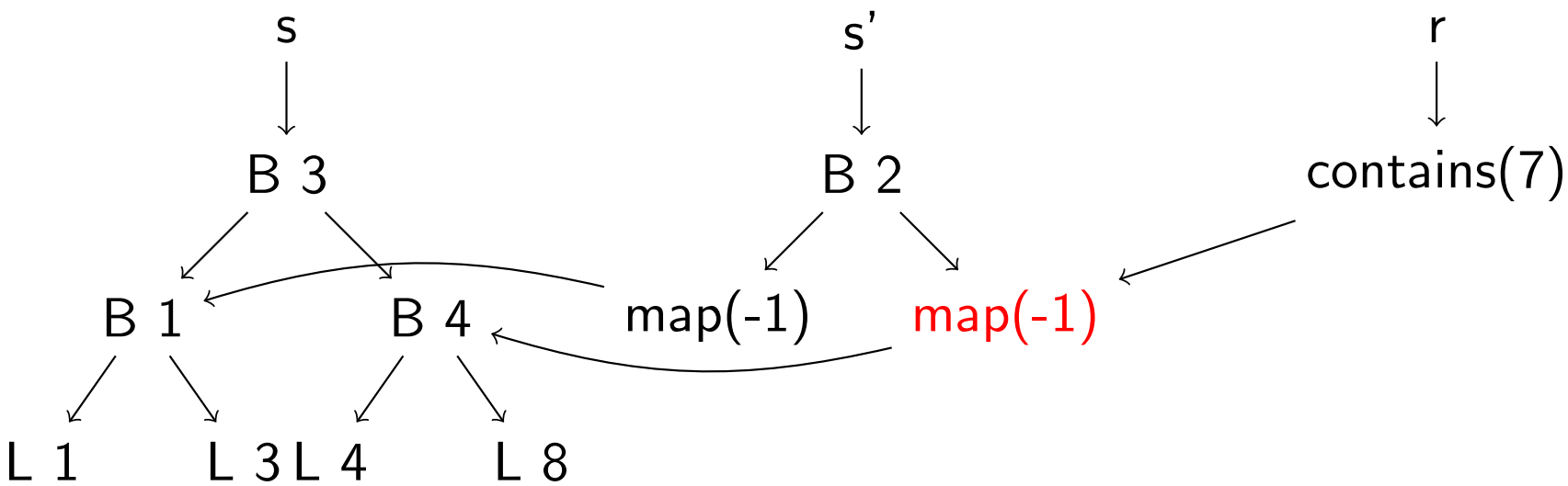
Functional Transaction Processing



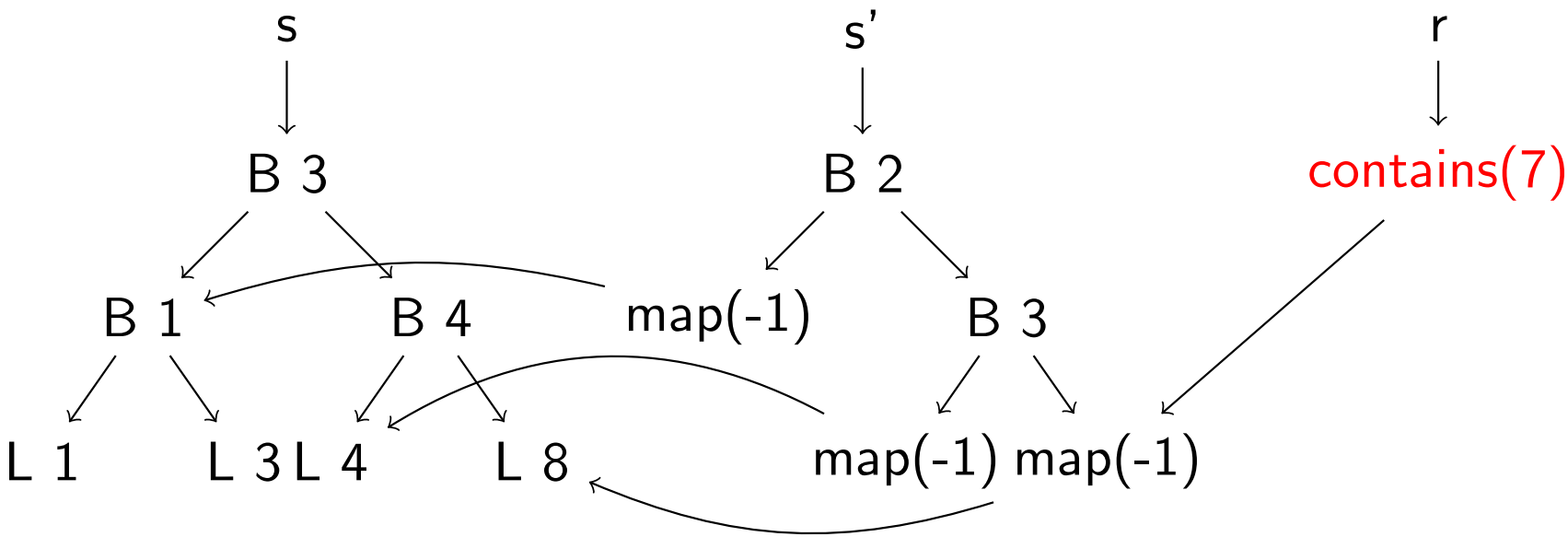
Functional Transaction Processing



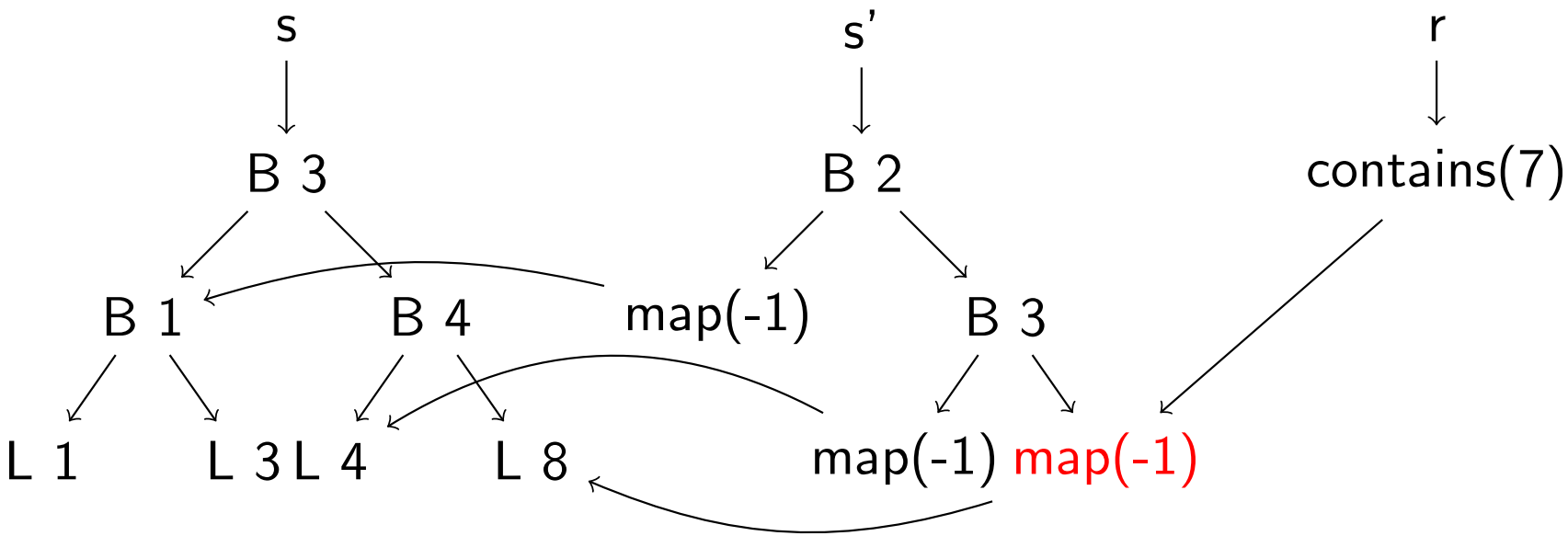
Functional Transaction Processing



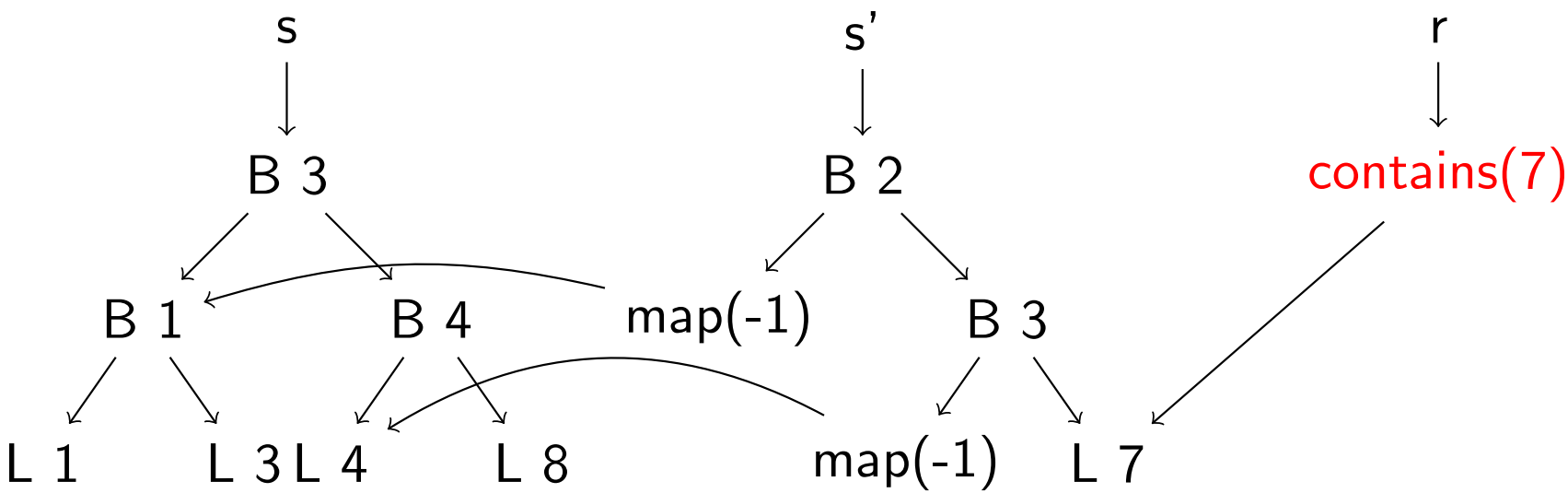
Functional Transaction Processing



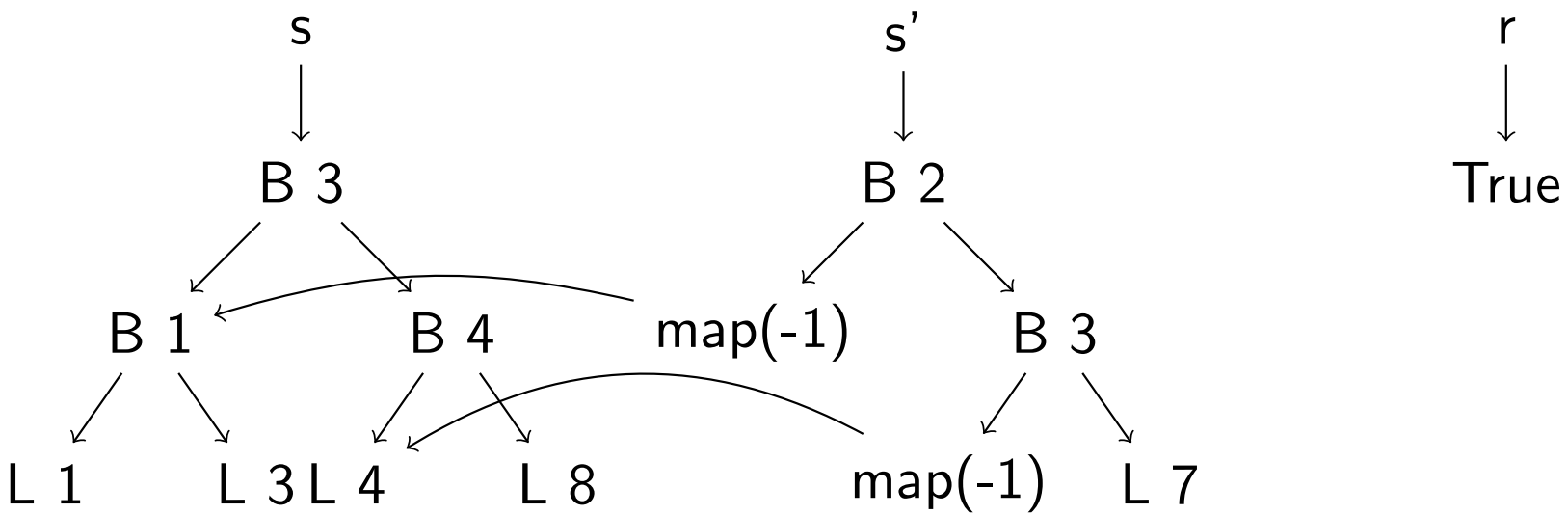
Functional Transaction Processing



Functional Transaction Processing



Functional Transaction Processing



Transactional Functional Language

Transactional Functional Language

Transactional Functional Language

Transactional Functional Language

A state is a set of bindings $x = E$, where:

- x is a *name*.
- E is an *expression*.

Transactional Functional Language

Transactional Functional Language

A state is a set of bindings $x = E$, where:

- x is a *name*.
- E is an *expression*.

A transaction is a set of bindings $x = E$, where

- x is a variable.
- E is an expression.

Transactional Functional Language

Transactional Functional Language

A state is a set of bindings $x = E$, where:

- x is a *name*.
- E is an *expression*.

A transaction is a set of bindings $x = E$, where

- x is a variable.
- E is an expression.

Transaction Variables

- *Current state variables*: `x`, `y`, `z`, ...

Transactional Functional Language

Transactional Functional Language

A state is a set of bindings $x = E$, where:

- x is a *name*.
- E is an *expression*.

A transaction is a set of bindings $x = E$, where

- x is a variable.
- E is an expression.

Transaction Variables

- *Current state variables*: `x`, `y`, `z`, ...
- *Next state variables*: `x'`, `y'`, `z'`, ...

Transactional Functional Language

Transactional Functional Language

A state is a set of bindings $x = E$, where:

- x is a *name*.
- E is an *expression*.

A transaction is a set of bindings $x = E$, where

- x is a variable.
- E is an expression.

Transaction Variables

- *Current state variables*: `x`, `y`, `z`, ...
- *Next state variables*: `x'`, `y'`, `z'`, ...
- *Result variable*: `result`

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

Transactional Functional Language

Example

s_0: `names  = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'`

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'` $\rightarrow r_1$: `["dave", "alice", "bob"]`

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'` $\rightarrow r_1$: `["dave", "alice", "bob"]`

s_1: `names = ["dave", "alice", "bob"]`

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'` $\rightarrow r_1$: `["dave", "alice", "bob"]`

s_1: `names = ["dave", "alice", "bob"]`

t_2:
```
length' = λ list . case list of
  [] -> 0
  [x:xs] -> 1 + length' xs
```

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'` $\rightarrow r_1$: `["dave", "alice", "bob"]`

s_1: `names = ["dave", "alice", "bob"]`

t_2:
```
length' = λ list . case list of
  [] -> 0
  [x:xs] -> 1 + length' xs
```

s_2:
```
names = ["dave", "alice", "bob"]
length = λ list . case list of
  [] -> 0
  [x:xs] -> 1 + length xs
```

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'` $\rightarrow r_1$: `["dave", "alice", "bob"]`

s_1: `names = ["dave", "alice", "bob"]`

t_2:
```
length' = λ list . case list of
  [] -> 0
  [x:xs] -> 1 + length' xs
```

s_2:
```
names = ["dave", "alice", "bob"]
length = λ list . case list of
  [] -> 0
  [x:xs] -> 1 + length xs
```

t_3: `result = length names`

Transactional Functional Language

Example

s_0: `names = ["alice", "bob"]`

t_1: `names' = "dave" : names`

`result = names'` $\rightarrow r_1 :$ `["dave", "alice", "bob"]`

s_1: `names = ["dave", "alice", "bob"]`

t_2: `length' =` λ `list . case list of`

```
  [] -> 0
  [x:xs] -> 1 + length' xs
```

s_2: `names = ["dave", "alice", "bob"]`

`length =` λ `list . case list of`

```
  [] -> 0
  [x:xs] -> 1 + length xs
```

t_3: `result = length names` $\rightarrow r_3 : 3$

Implementation

Goals

- Execute transactions concurrently.

Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.

Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.
- Allow bindings to be created dynamically.

Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.
- Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.

Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.
- Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
- Store state in persistent memory.

Prototype

We implemented a prototype in Java.

Implementation - Graph Reduction

Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.

Implementation - Graph Reduction

Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
- We have a reduction graph.

Implementation - Graph Reduction

Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
- We have a reduction graph.
- On application of a binding we instantiate its template.

Implementation - Graph Reduction

Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
- We have a reduction graph.
- On application of a binding we instantiate its template.

Adaptations to support dynamic bindings

We resolve references to template graphs statically.

- Templates are anonymous, and can be garbage collected.

Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on eachother.

Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on eachother.
- Execute transactions *in parallel* using multiple processors.

Result sharing and randomisation

Reduction threads:

Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on eachother.
- Execute transactions *in parallel* using multiple processors.

Result sharing and randomisation

Reduction threads:

- Share results between one another.

Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on eachother.
- Execute transactions *in parallel* using multiple processors.

Result sharing and randomisation

Reduction threads:

- Share results between one another.
- Take random reduction "paths".

Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

- Inserted before every redexes.

Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

- Inserted before every redexes.
- Reference to either redex or result, which is updated on reduction.

Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

- Inserted before every redexes.
- Reference to either redex or result, which is updated on reduction.
- Algorithm that ensures that updating maintains sharing between threads.

Implementation - Graph Reduction

Sharing Results

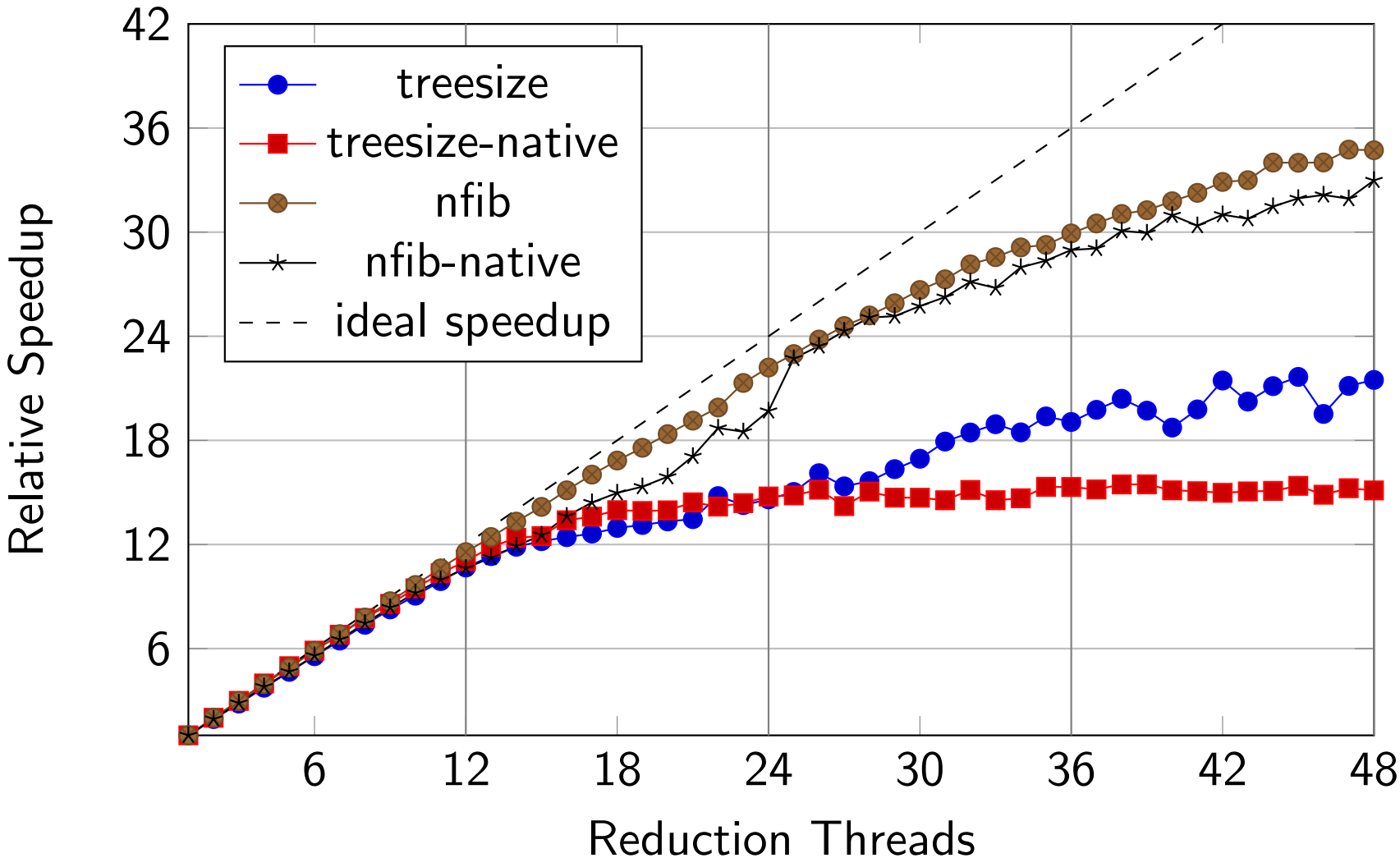
Special nodes to share results:

- Inserted before every redexes.
- Reference to either redex or result, which is updated on reduction.
- Algorithm that ensures that updating maintains sharing between threads.

Randomisation

- We reduce strict-function arguments in a random order:
- We reduce data elements in a random order.

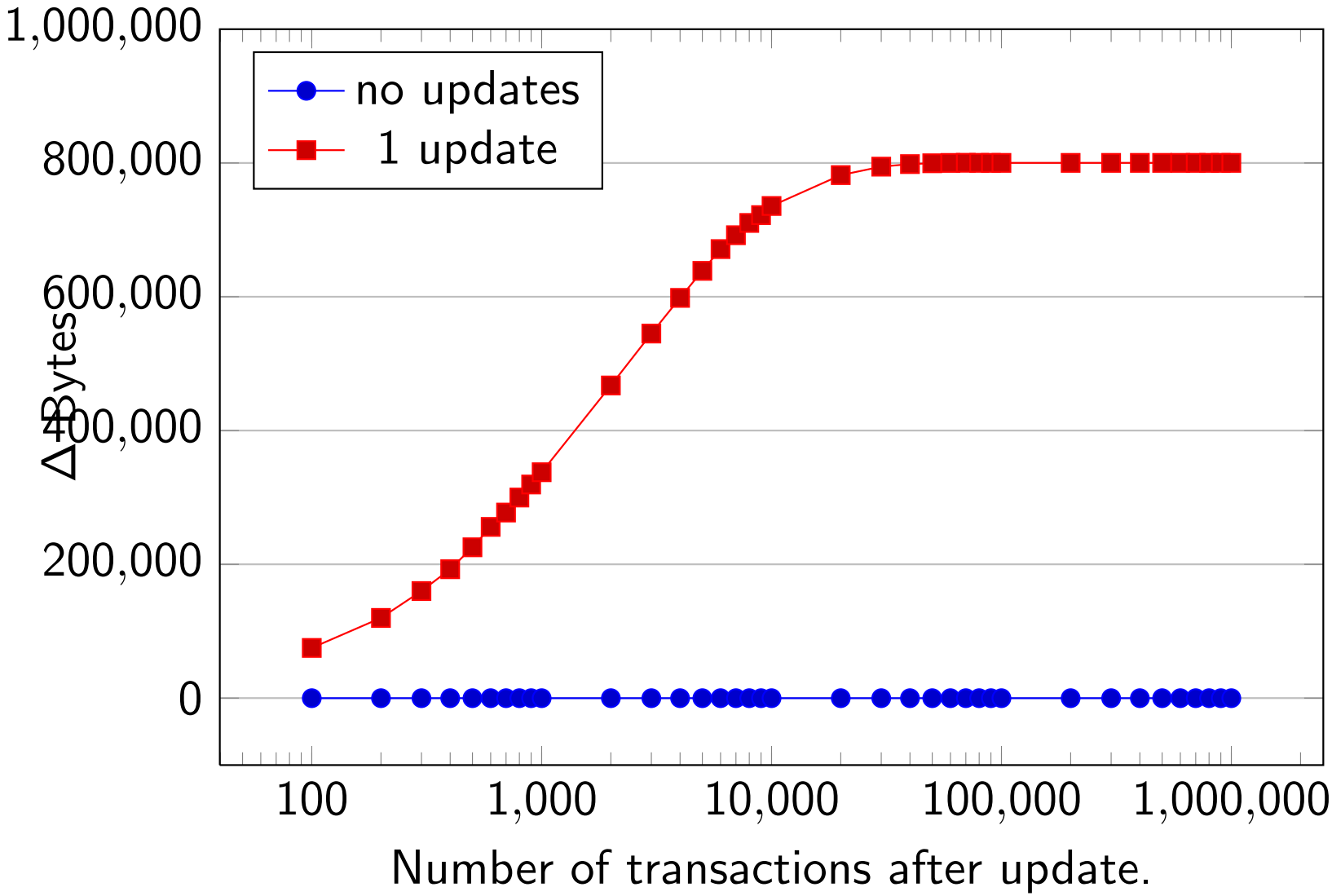
Evaluation


Relative Speedup
Reduction Threads
treesize
treesize-native
nfib
nfib-native
ideal speedup
6
12
18
24
30
36
42
48

Evaluation

	treesize	treesize-native	nfib	nfib-native
Serial	2666 ms	819 ms	3294 ms	626 ms
Parallel	3243 ms	1291 ms	4162 ms	819 ms
Overhead	21.6%	57.6%	26.4%	30.1%

Evaluation


1,000,000
800,000
600,000
400,000
200,000
0
ΔBytes
no updates
1 update
100
1,000
10,000
100,000
1,000,000
Number of transactions after update.

Persistence - Journaling

Journaling

Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

Persistence - Journaling

Journaling

Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

Checkpointing

Journaling is not enough:

- The log can grow very large.

Persistence - Journaling

Journaling

Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

Checkpointing

Journaling is not enough:

- The log can grow very large.
- Long recovery times.

Persistence - Journaling

Journaling

Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

Checkpointing

Journaling is not enough:

- The log can grow very large.
- Long recovery times.
- The size of the state is limited to main-memory.

Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.

Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.

Advantages

- We can snapshot computations.

Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.

Advantages

- We can snapshot computations.
- We can maintain sharing.

Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.

Advantages

- We can snapshot computations.
- We can maintain sharing.

Disadvantages

- The state is limited to main-memory.

Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.

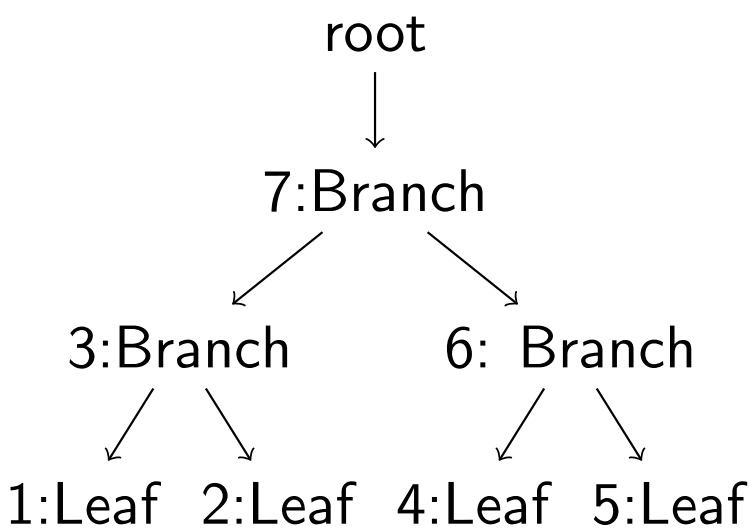
Advantages

- We can snapshot computations.
- We can maintain sharing.

Disadvantages

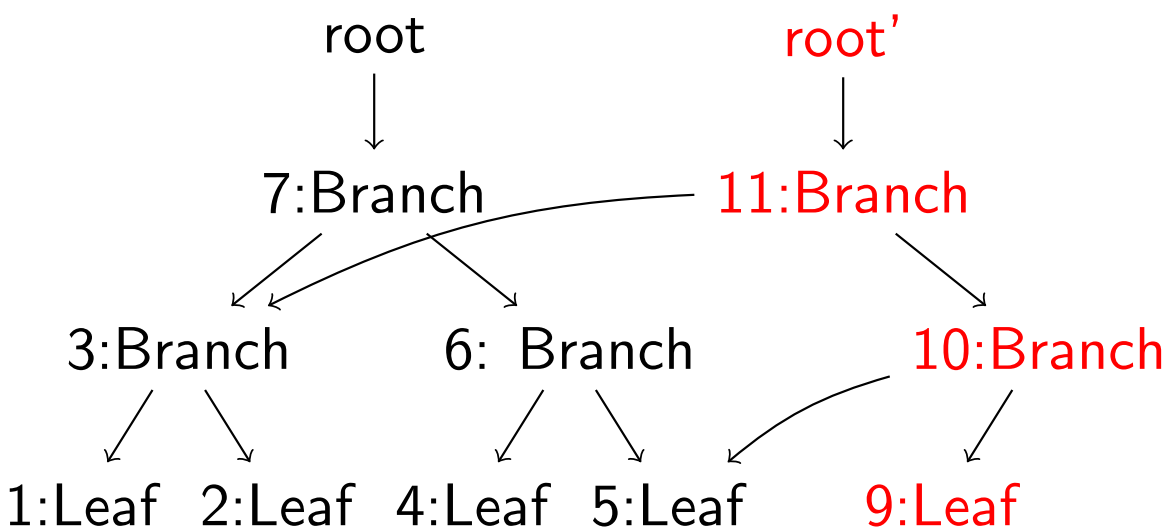
- The state is limited to main-memory.
- Recovery can take a while.

Persistence - Log-Structured Storage



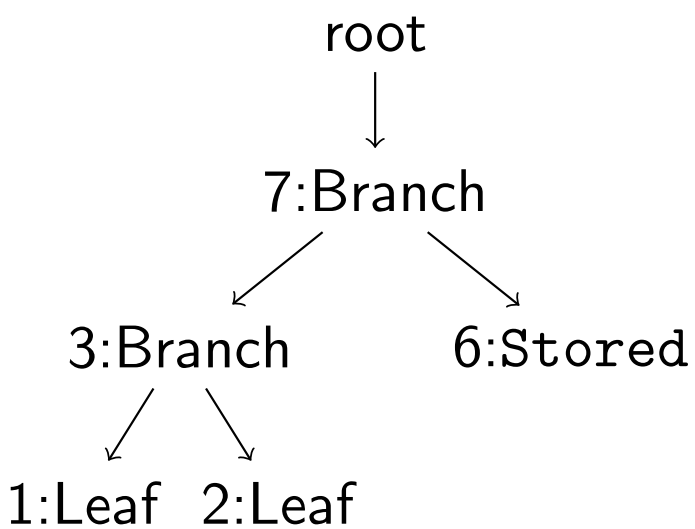

```
1: Leaf
2: Leaf
3: Branch 1 2
4: Leaf
5: Leaf
6: Branch 4 5
7: Branch 3 6
8: Root 7
```

Persistence - Log-Structured Storage




```
1 : Leaf
2 : Leaf
3 : Branch 1 2
4 : Leaf
5 : Leaf
6 : Branch 4 5
7 : Branch 3 6
8 : Root 7
9 : Leaf
10 : Branch 9 5
11 : Branch 3 10
12 : Root 12
```

Persistence - Log-Structured Storage




```
1: Leaf
2: Leaf
3: Branch 1 2
4: Leaf
5: Leaf
6: Branch 4 5
7: Branch 3 6
8: Root 7
```

Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.

Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
- Fast recovery.

Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
- Fast recovery.

Disadvantages / Complications

- Checkpointing of computations is expensive.

Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
- Fast recovery.

Disadvantages / Complications

- Checkpointing of computations is expensive.
- Garbage collection is needed.

Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
- Fast recovery.

Disadvantages / Complications

- Checkpointing of computations is expensive.
- Garbage collection is needed.
- Maintaining sharing is expensive.

Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
- Fast recovery.

Disadvantages / Complications

- Checkpointing of computations is expensive.
- Garbage collection is needed.
- Maintaining sharing is expensive.
- We need to maintain locality of reference.

Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Advantages

- We can checkpoint computations.

Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Advantages

- We can checkpoint computations.
- States can be larger than main-memory.

Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
- Fast recovery.

Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
- Fast recovery.

Disadvantages

- Garbage collection is still needed.

Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
- Fast recovery.

Disadvantages

- Garbage collection is still needed.
- There is no sharing in the reduced heap.

Conclusions

Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.

Conclusions

Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.

Conclusions

Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
- Allow bindings to be created dynamically.

Conclusions

Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
- Allow bindings to be created dynamically.
- New approach to parallel graph reduction.

Conclusions

Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
- Allow bindings to be created dynamically.
- New approach to parallel graph reduction.
- Investigated method for storing states in persistent memory.

Future Work

Future Work

(1) Resolving thunk leaks in lazily constructed states.

(2) Implementation of storage mechanisms.

(3) Practical use of system.

(4) Scheduling of reduction threads.