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

TeReSe, Vrije Universiteit, Amsterdam
14 November 2012

Transaction Processing Systems

Transaction Processing Systems

- Banking

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.
- Everyone wants a quick response.

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.
- Everyone wants a quick response.
- Transactions can be very large.

Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

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

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.

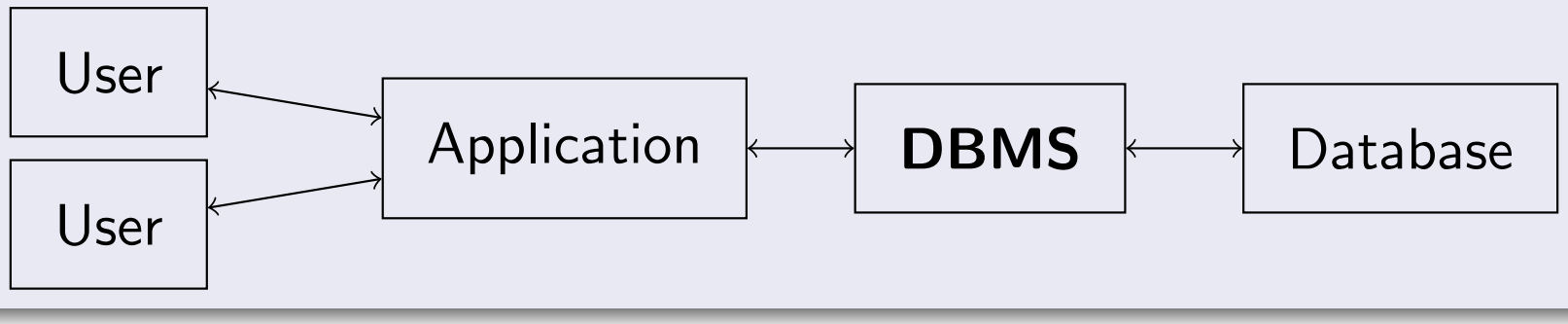
Consistent: After each transaction, the system is in a consistent state.

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

Durable: The effect of a transaction is permanent.

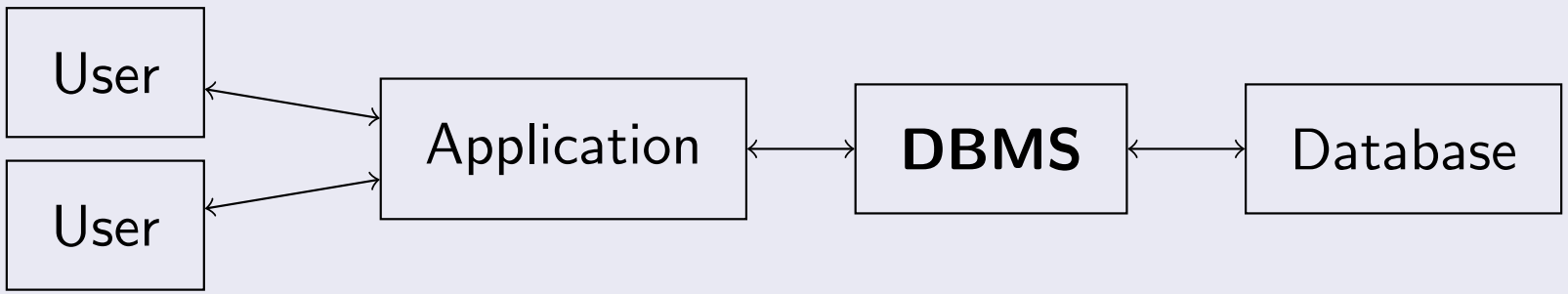
Current Approach

Traditional Architecture



Current Approach

Traditional Architecture

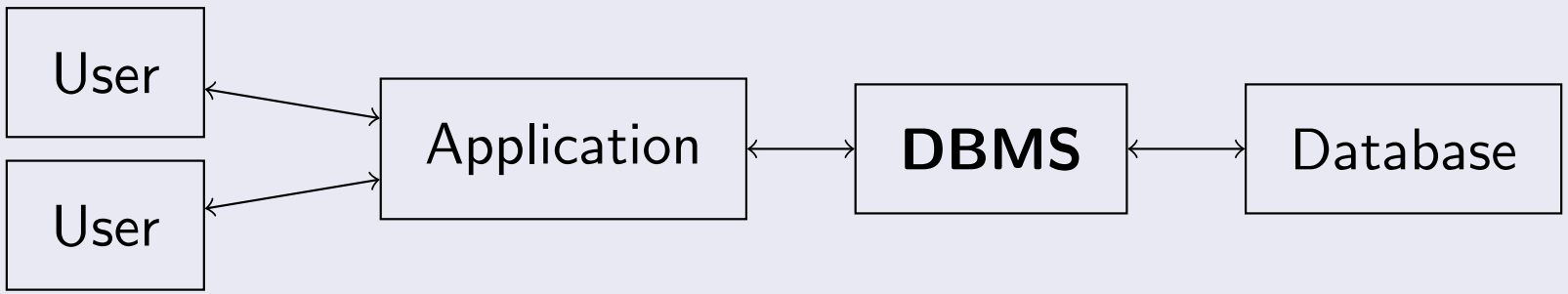


Database Management System

- Optimised to handle large amounts of data.

Current Approach

Traditional Architecture

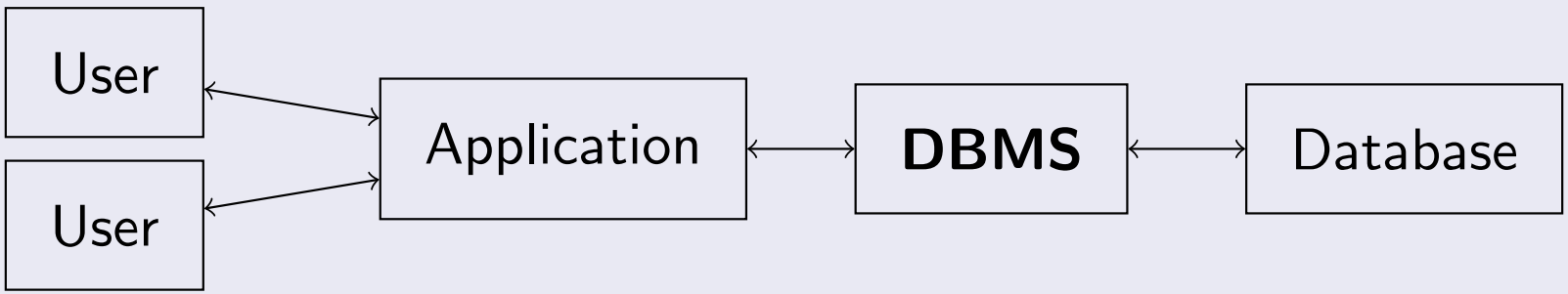


Database Management System

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

Current Approach

Traditional Architecture

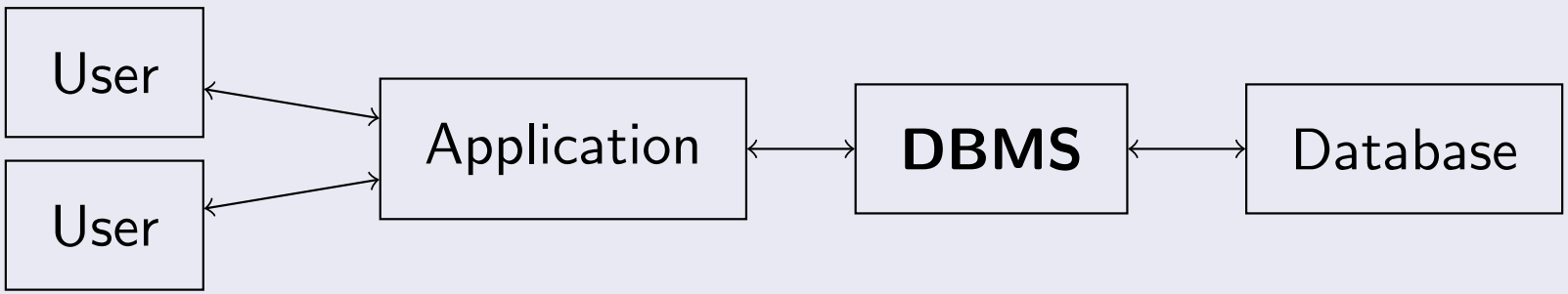


Database Management System

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

Current Approach

Traditional Architecture



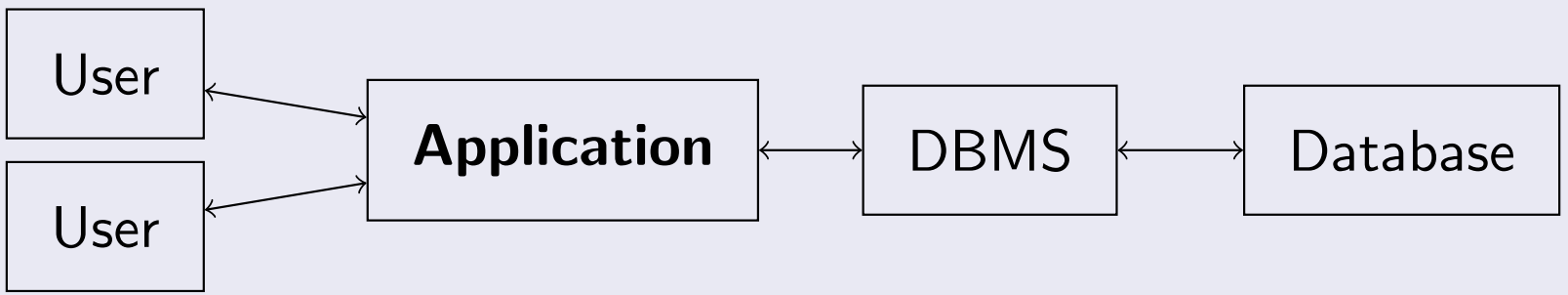
Database Management System

- 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.

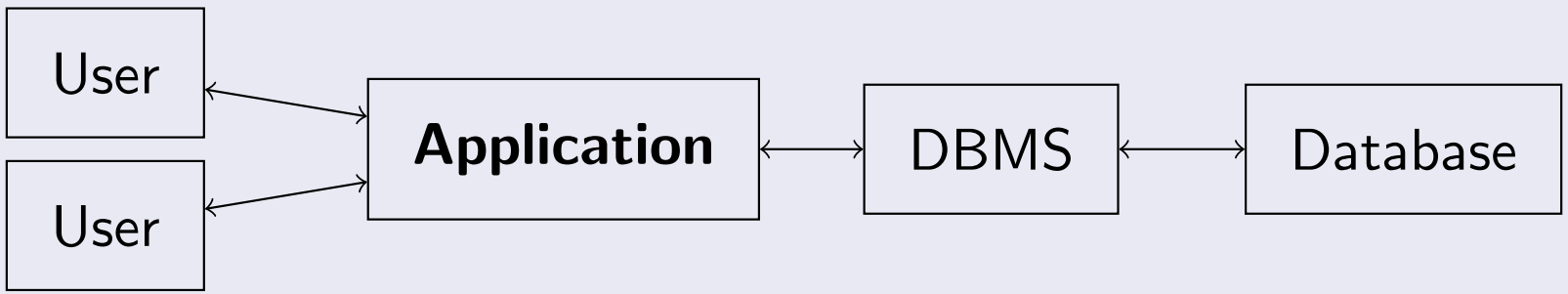
Current Approach

Traditional Architecture



Current Approach

Traditional Architecture

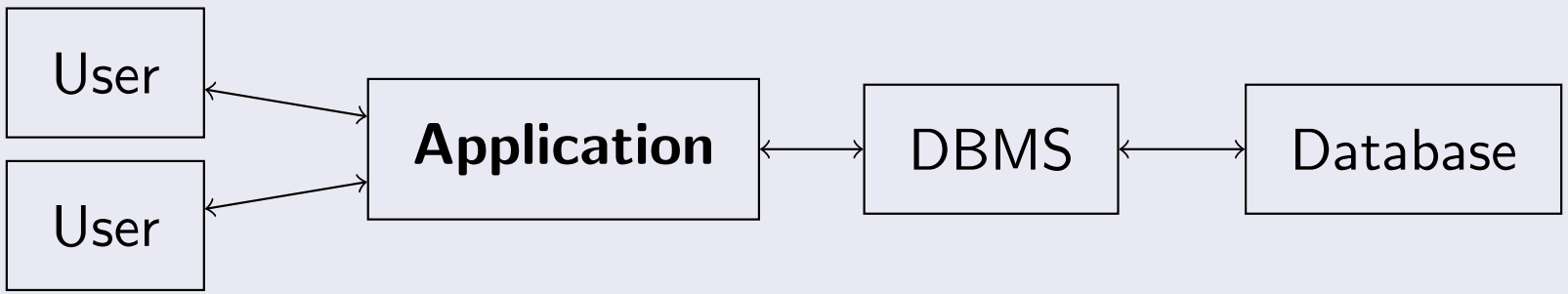


Application

- Interface to the outside world.

Current Approach

Traditional Architecture

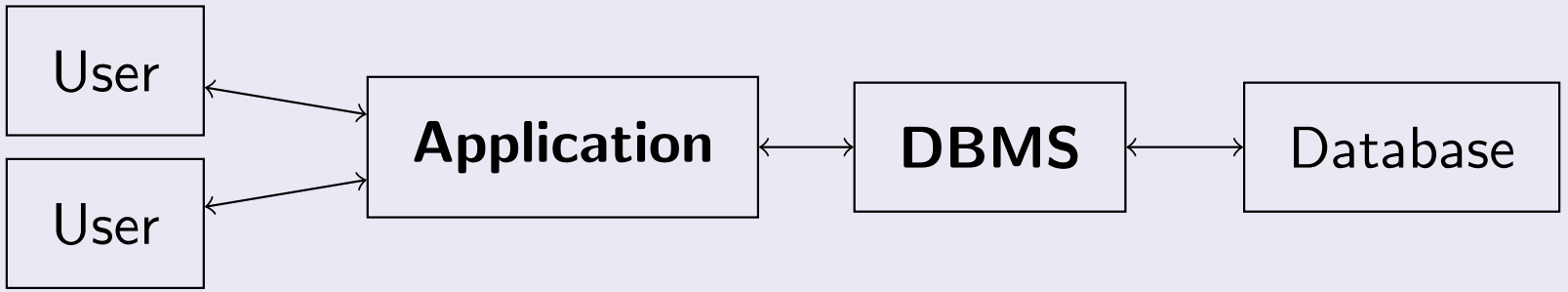


Application

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

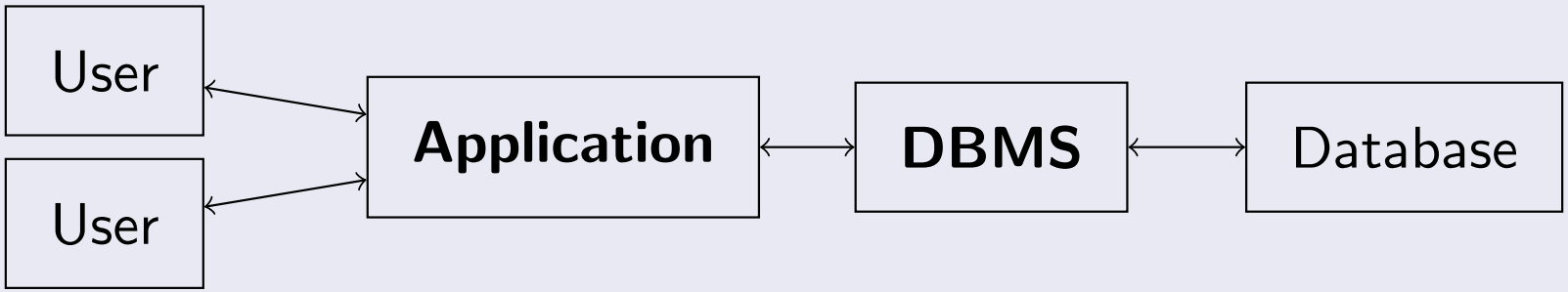
Current Approach

Traditional Architecture



Current Approach

Traditional Architecture

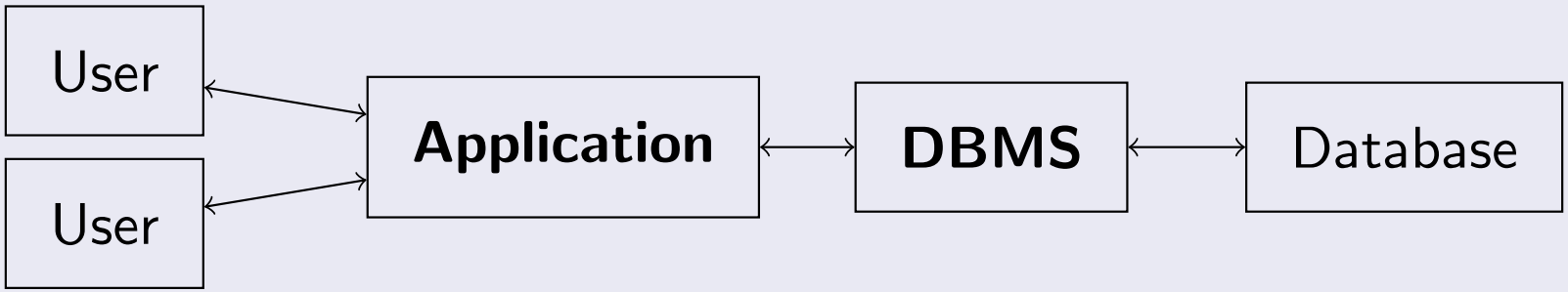


Limitations

- Application and DBMS have different type system.

Current Approach

Traditional Architecture

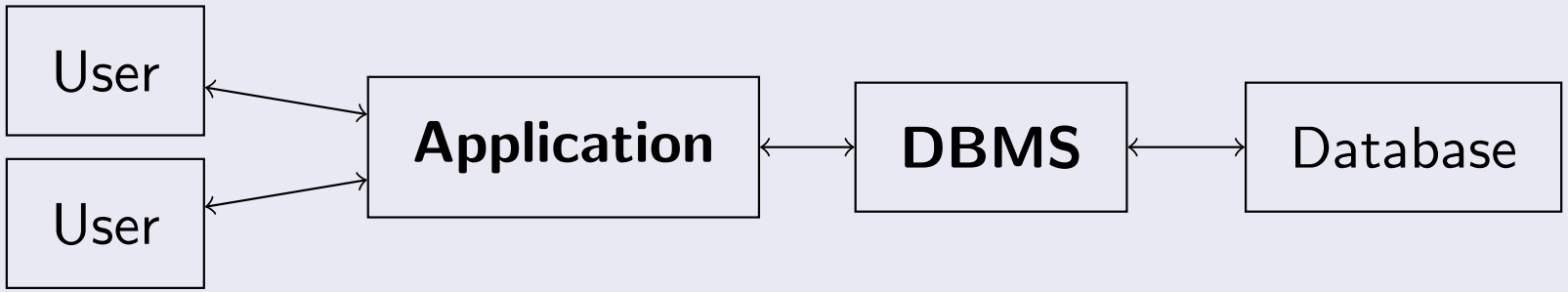


Limitations

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

Current Approach

Traditional Architecture

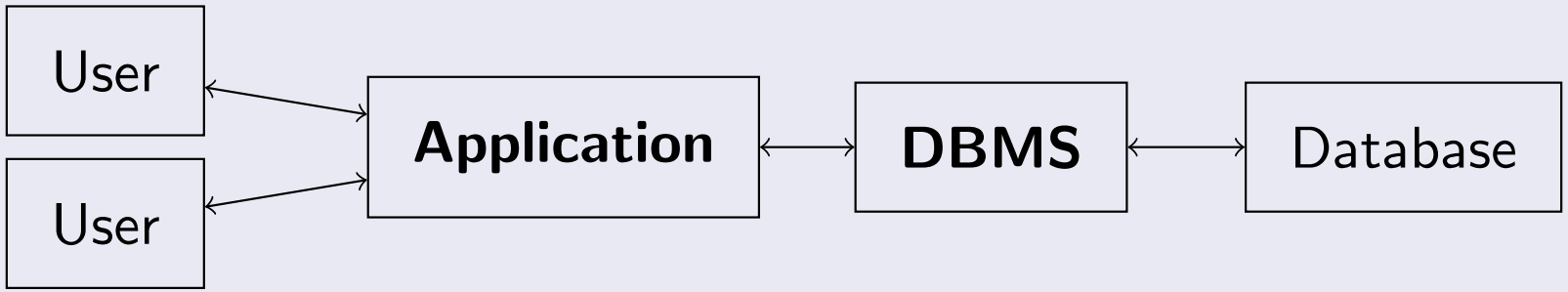


Limitations

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

Current Approach

Traditional Architecture

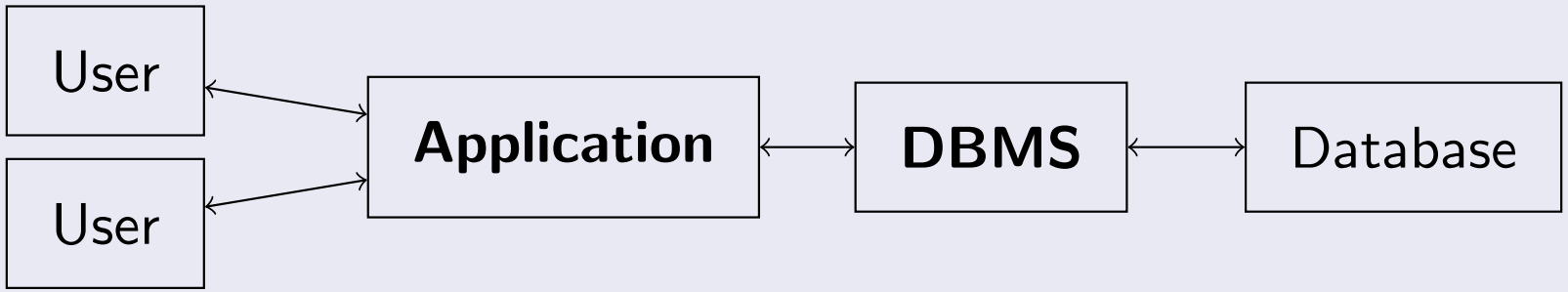


Limitations

- 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

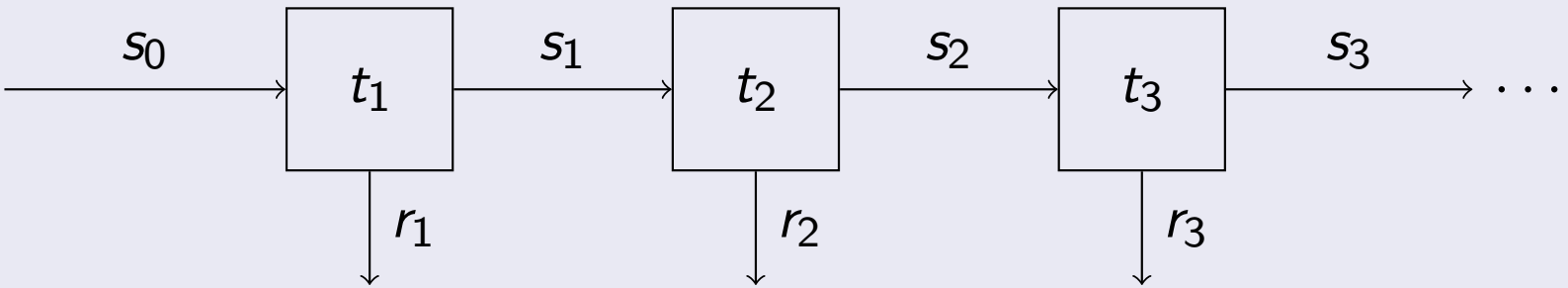
Functional Transaction Processing

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

Functional Transaction Processing

Functional Transaction Processing

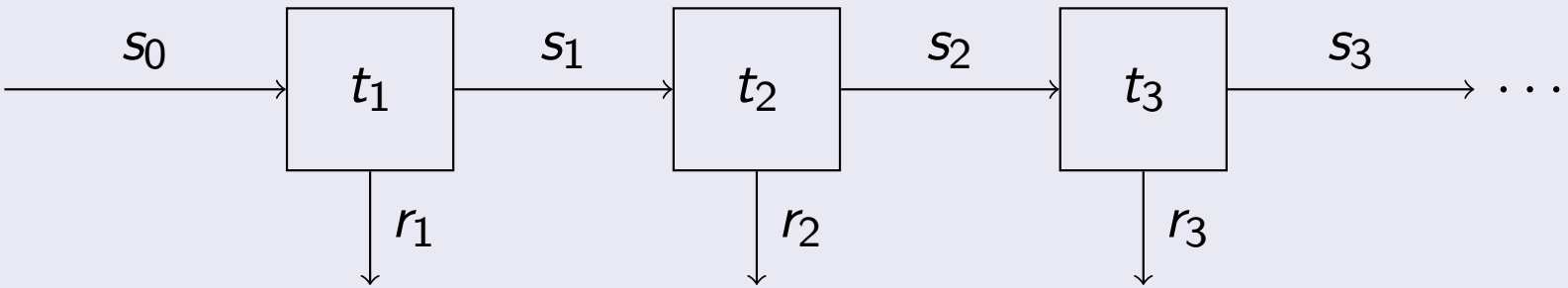
- A *transaction function*: $State \rightarrow State \times Result$.
- A *transaction manager*: $State \times [Transaction] \rightarrow [Result]$.



Functional Transaction Processing

Functional Transaction Processing

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



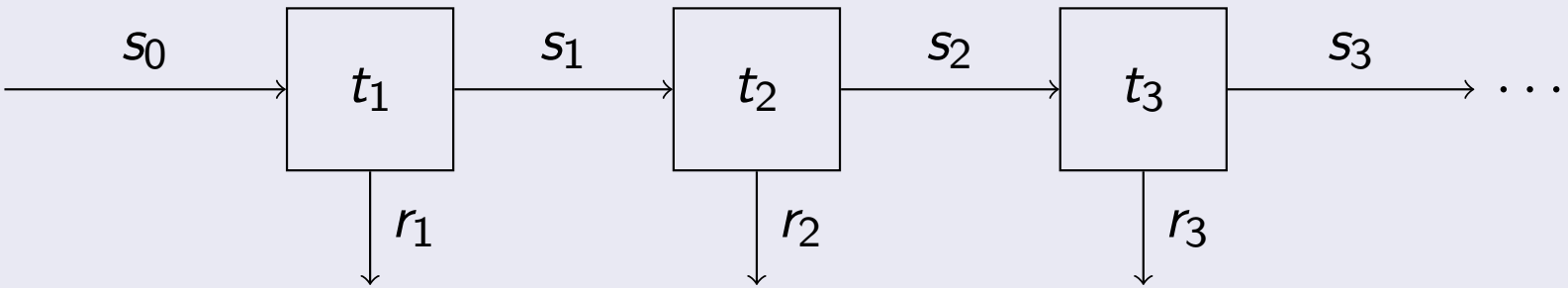
Correctness (ACID)

- 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*: $State \times [Transaction] \rightarrow [Result]$.



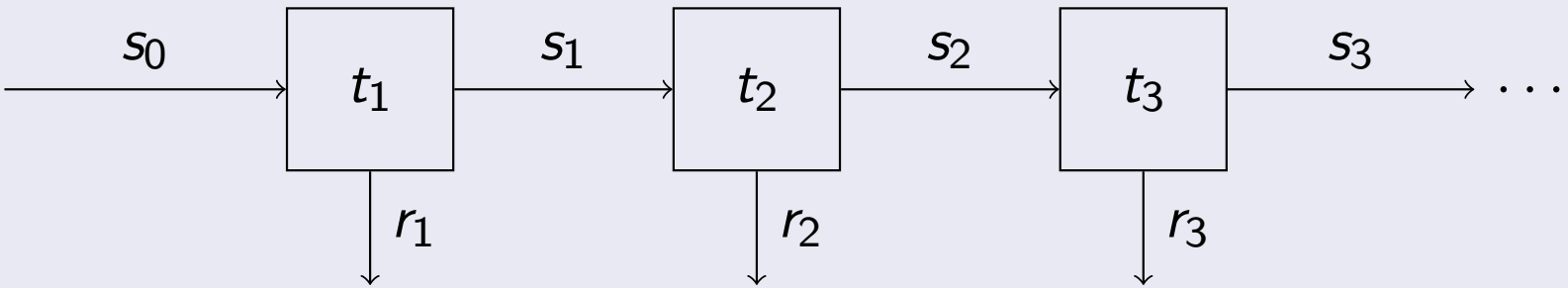
Correctness (ACID)

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

Functional Transaction Processing

Functional Transaction Processing

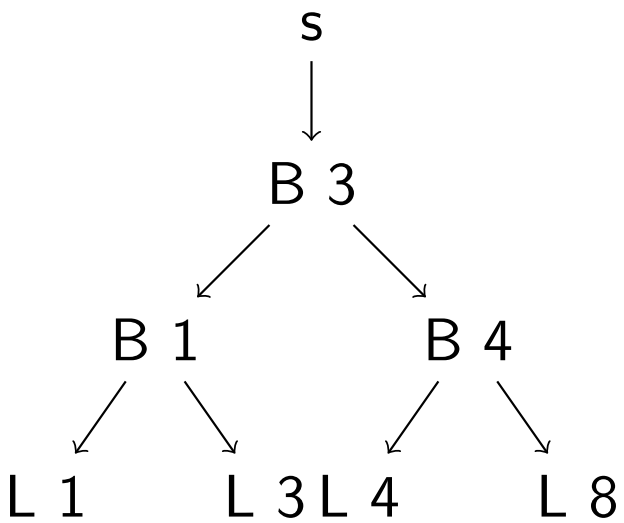
- A *transaction function*: $State \rightarrow State \times Result$.
- A *transaction manager*: $State \times [Transaction] \rightarrow [Result]$.



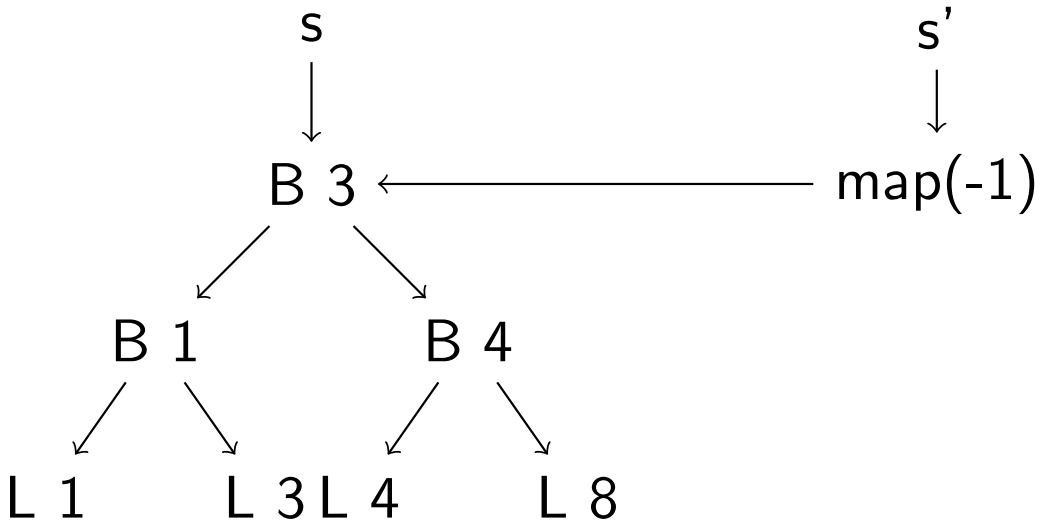
Correctness (ACID)

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

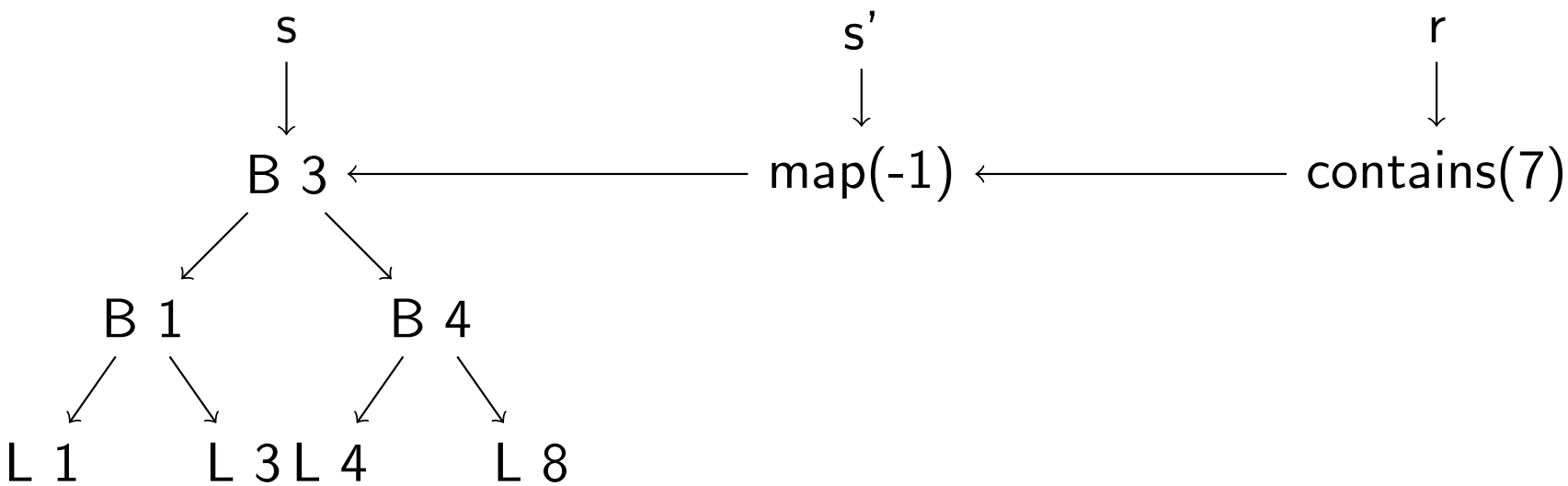
Functional Transaction Processing



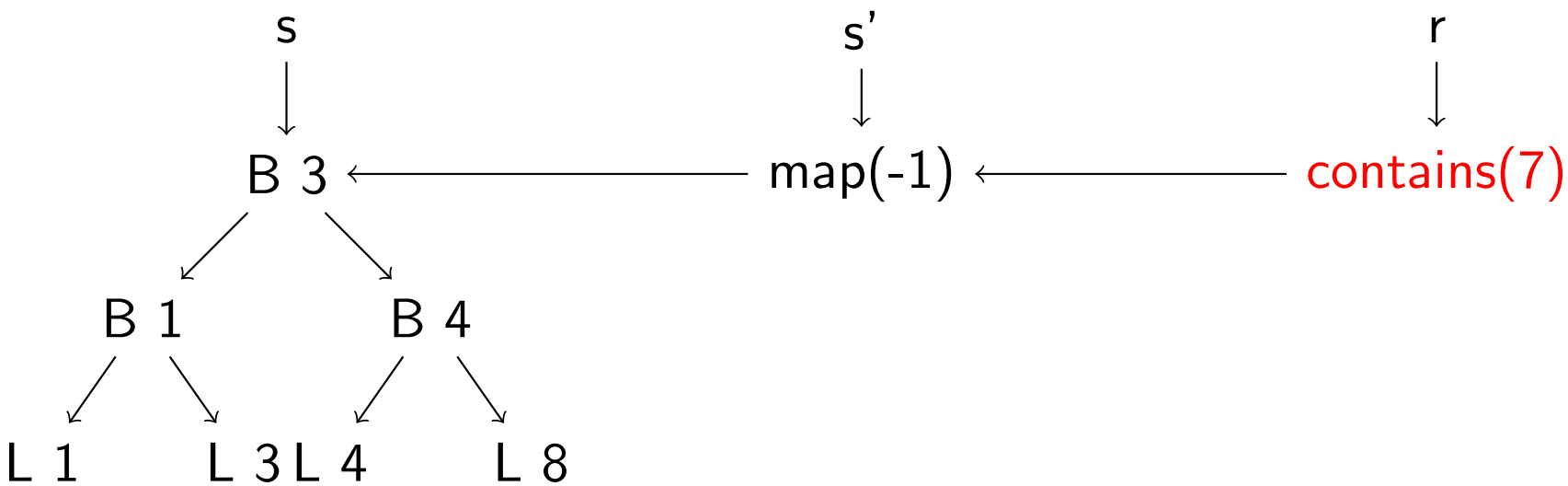
Functional Transaction Processing



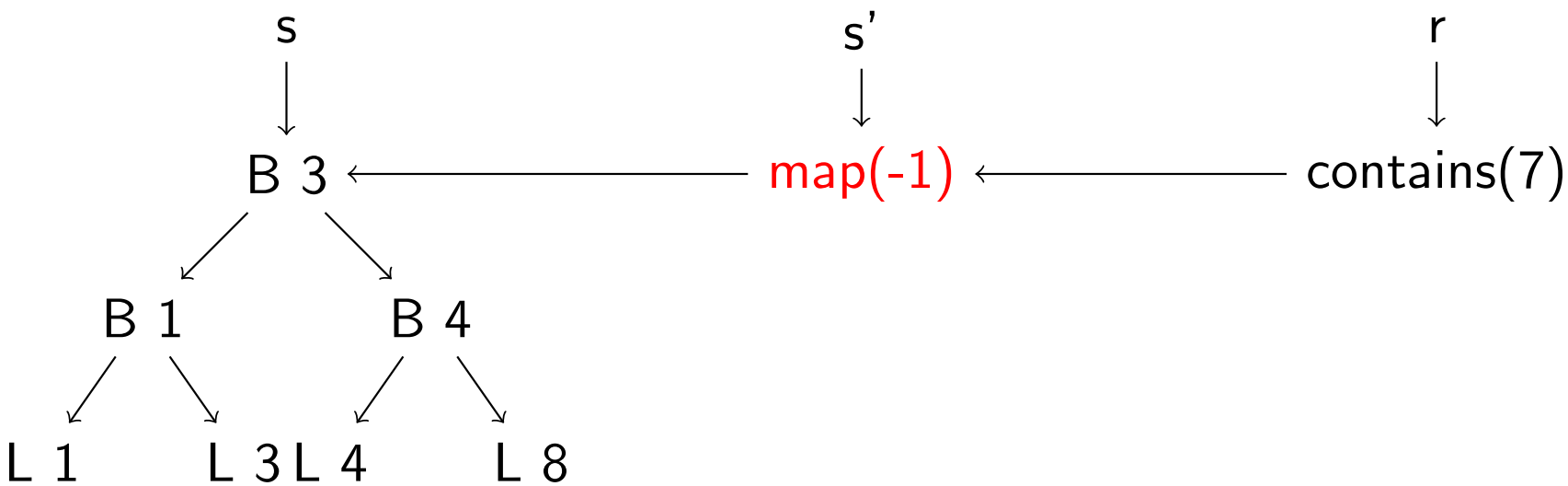
Functional Transaction Processing



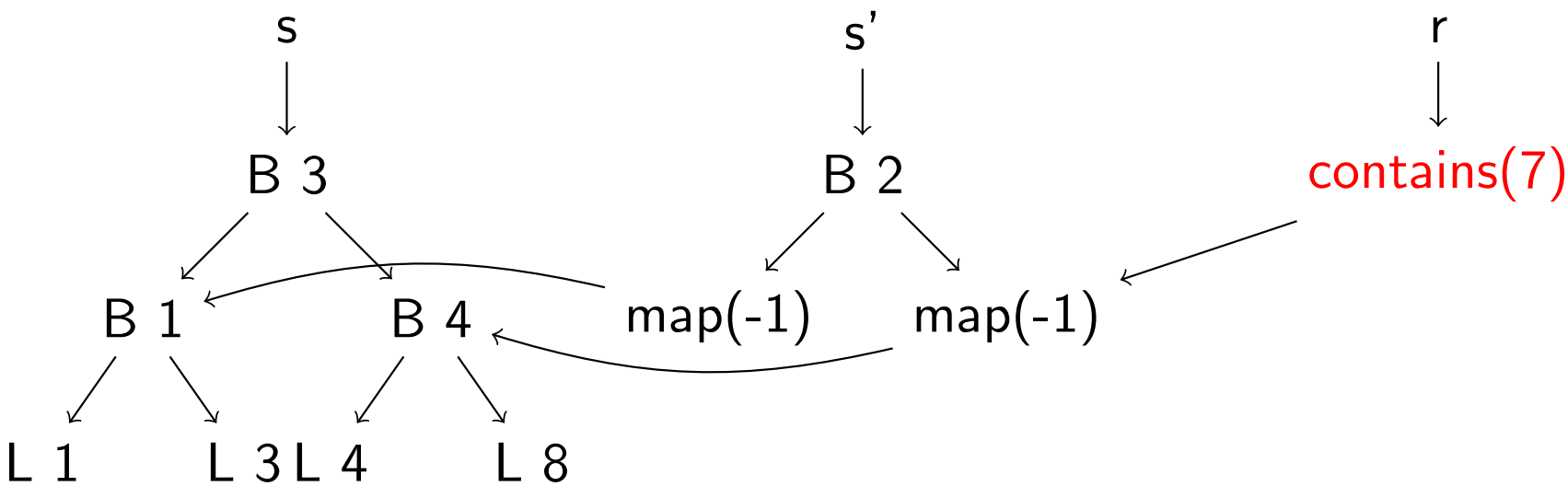
Functional Transaction Processing



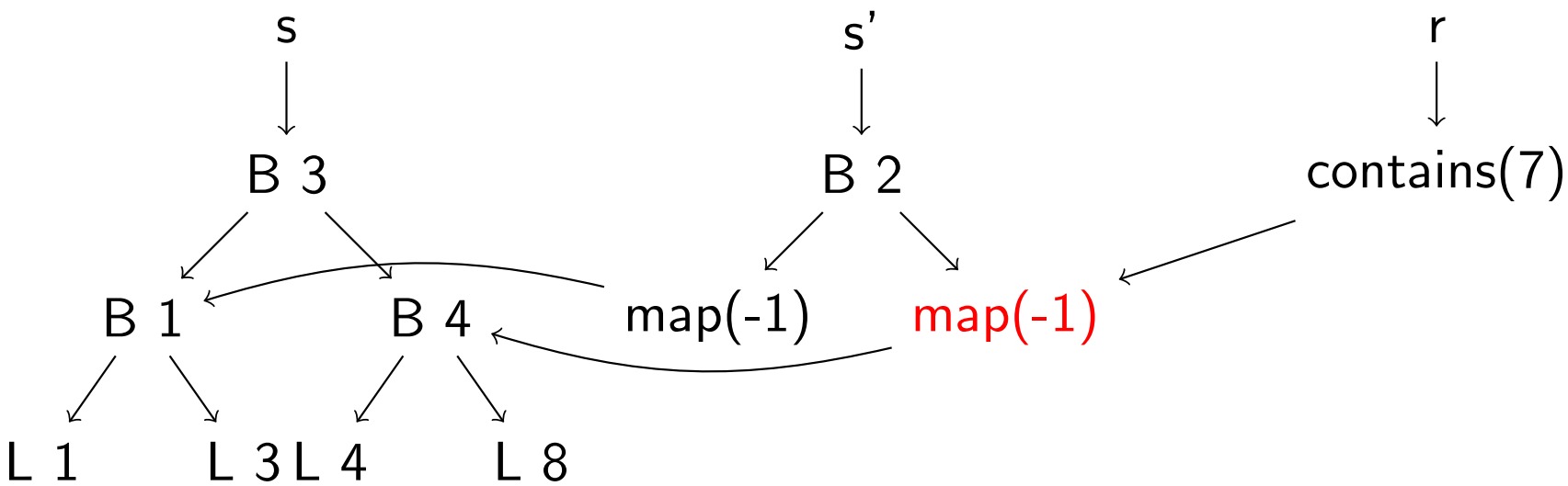
Functional Transaction Processing



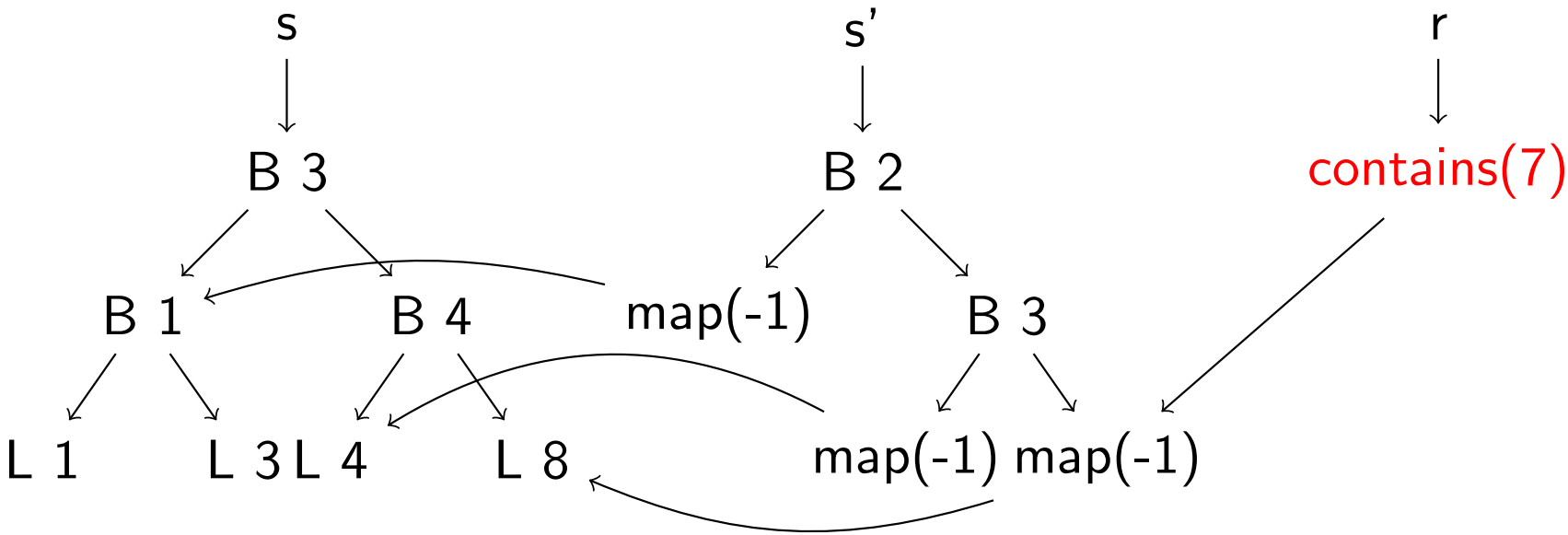
Functional Transaction Processing



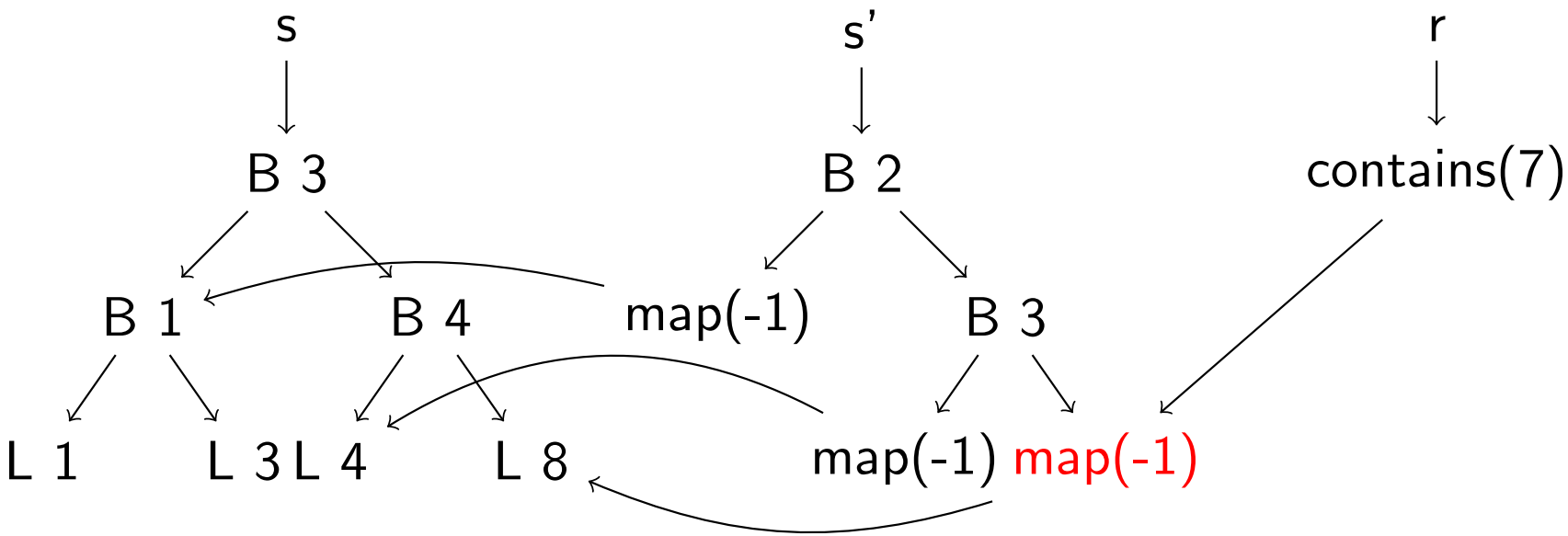
Functional Transaction Processing



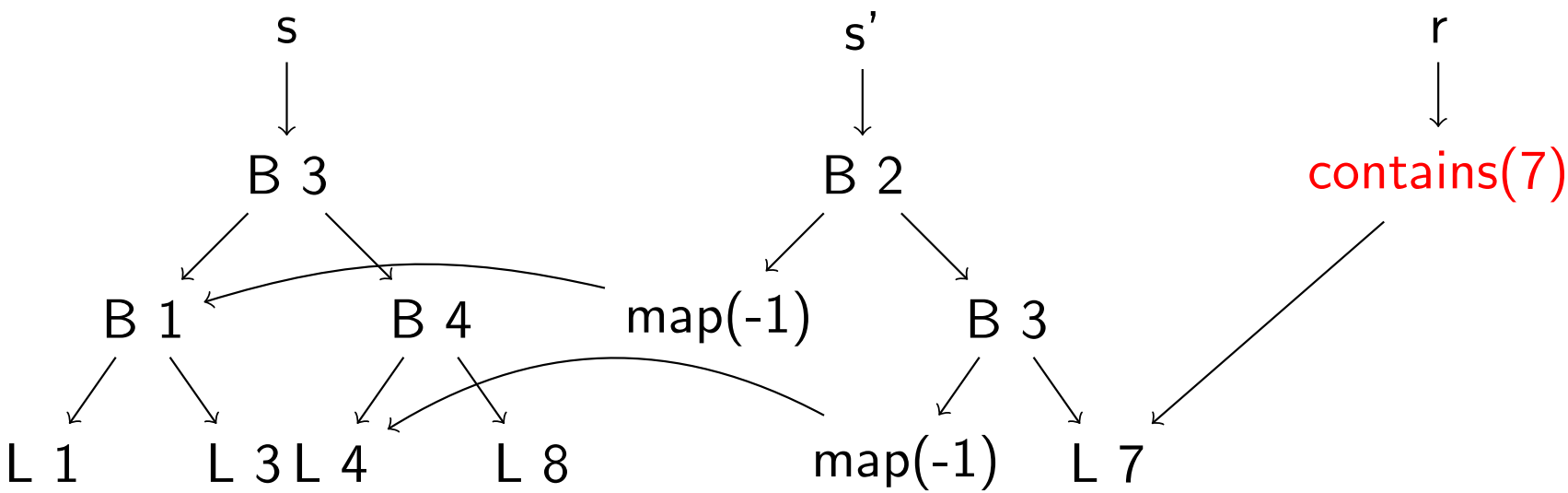
Functional Transaction Processing



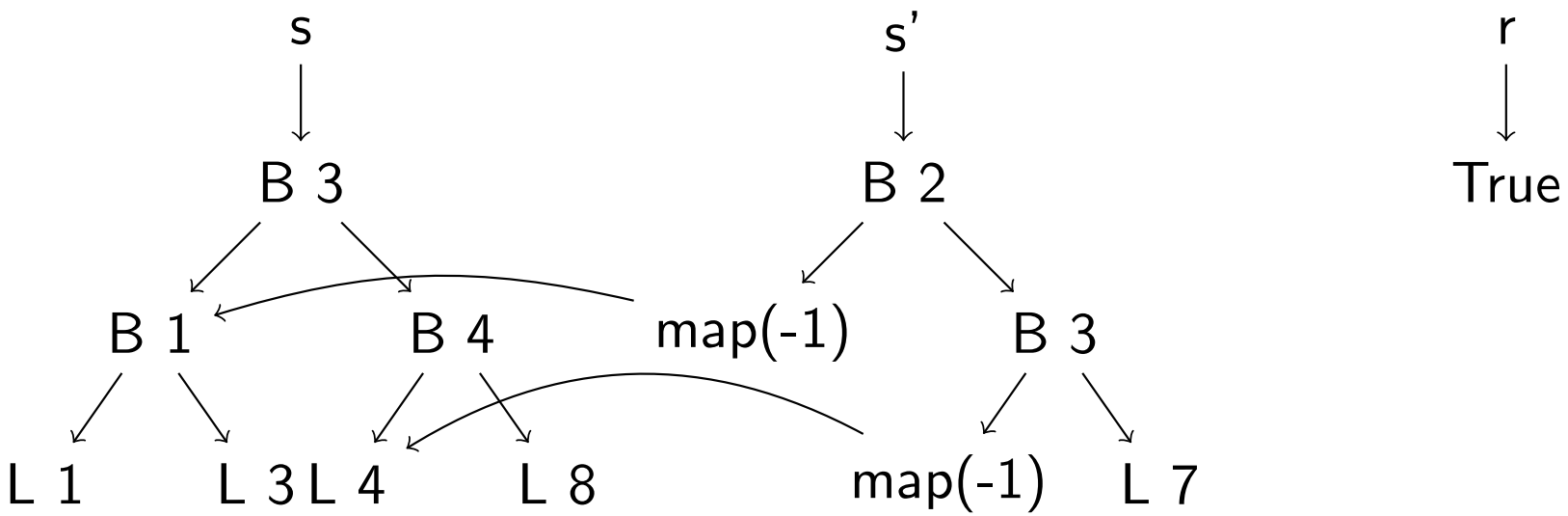
Functional Transaction Processing



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

- Transactional functional language.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.
- Store states in persistent memory.

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.
- Store states in persistent memory.
  - Durability

Implementation

Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.
- Store states in persistent memory.
  - Durability
  - States larger than main memory.

Current Implementation

- Implementation in Java.

Implementation

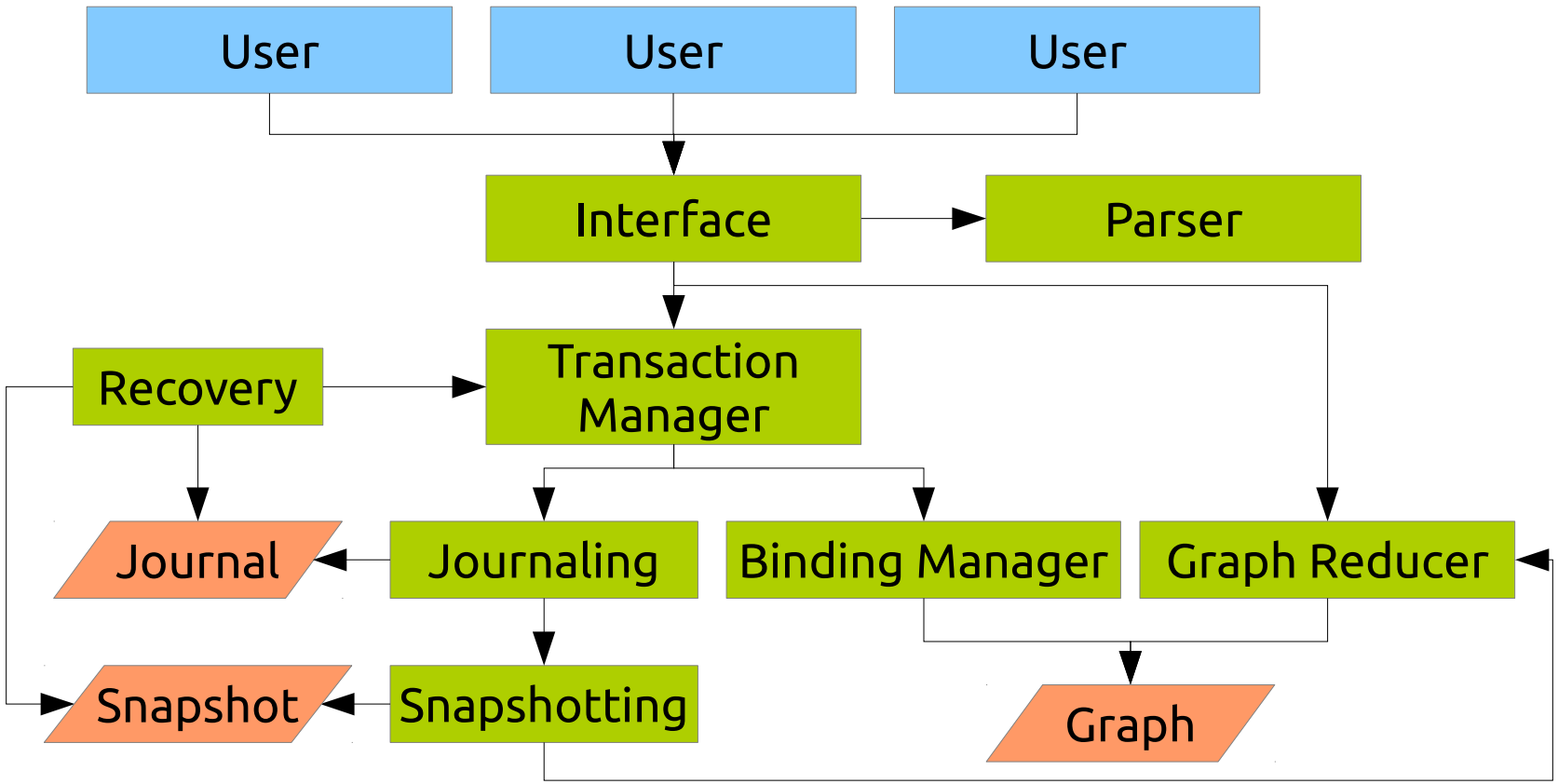
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.
- Store states in persistent memory.
  - Durability
  - States larger than main memory.

Current Implementation

- Implementation in Java.
- Only main memory states.

Architecture



Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.
- For every supercombinator we construct a *template graph*.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.
- For every supercombinator we construct a *template graph*.
  - Each template graph has a name.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.
- For every supercombinator we construct a *template graph*.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.
- For every supercombinator we construct a *template graph*.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.
- Reduction is done in a *reduction graph*.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.
- For every supercombinator we construct a *template graph*.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.
- Reduction is done in a *reduction graph*.
  - Initialised with graph of expression to be reduced.

Graph Reduction

Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.E$:
  - E is not a supercombinator.
  - Any λ-abstraction in E is a supercombinator.
- For every supercombinator we construct a *template graph*.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.
- Reduction is done in a *reduction graph*.
  - Initialised with graph of expression to be reduced.
  - Beta-reduction: $(f\ E) \rightarrow T_f[f := E]$, where T_f is the template graph associated with x.

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

Solution: Anonymous Templates

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

Solution: Anonymous Templates

- We do not maintain a mapping of names to templates.

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

Solution: Anonymous Templates

- We do not maintain a mapping of names to templates.
- We resolve references to template graphs statically.

Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

Solution: Anonymous Templates

- We do not maintain a mapping of names to templates.
- We resolve references to template graphs statically.
- Garbage collection cleans up unused templates.

Graph Reduction

Scheduling

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency Minimize latency of individual transactions.

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency Minimize latency of individual transactions.

Parallelism Maximize overal system throughput.

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency Minimize latency of individual transactions.

Parallelism Maximize overal system throughput.

Distributing Work

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency Minimize latency of individual transactions.

Parallelism Maximize overal system throughput.

Distributing Work

- Keep workers busy.

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency Minimize latency of individual transactions.

Parallelism Maximize overal system throughput.

Distributing Work

- Keep workers busy.
- Avoid contention between workers.

Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency Minimize latency of individual transactions.

Parallelism Maximize overal system throughput.

Distributing Work

- Keep workers busy.
- Avoid contention between workers.
- *Minimise latency of transactions.*

Graph Reduction - Distributing Work

Randomisation

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- Fast random number generator

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- Fast random number generator
- Coordination between workers

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- ☐ Fast random number generator
- ☐ Coordination between workers ← we implemented this

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- Fast random number generator
- Coordination between workers ← we implemented this

Orders

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- ☐ Fast random number generator
- ☐ Coordination between workers ← we implemented this

Orders

- ☐ Permutation

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- Fast random number generator
- Coordination between workers ← we implemented this

Orders

- Permutation
- Left to right / right to left

Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- ☐ Fast random number generator
- ☐ Coordination between workers ← we implemented this

Orders

- ☐ Permutation
- ☐ Left to right / right to left ← we implemented this

Graph Reduction - Distributing Work

```
whnf(Add(left, right, left_to_right) : Node) : Node {
  left_to_right ← !left_to_right;
  if(left_to_right) {
    l ← whnf(left); r ← whnf(right);
  } else {
    r ← whnf(right); l ← whnf(left);
  }
  return Int(l.value + r.value);
}
```

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results $\rightarrow$ Duplicate (large) computations.

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results $\rightarrow$ Duplicate (large) computations.

Solution Strategies

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results $\rightarrow$ Duplicate (large) computations.

Solution Strategies

- Avoid duplicate computations.

Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results → Duplicate (large) computations.

Solution Strategies

- Avoid duplicate computations.
- Ensure sharing of duplicate results.

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

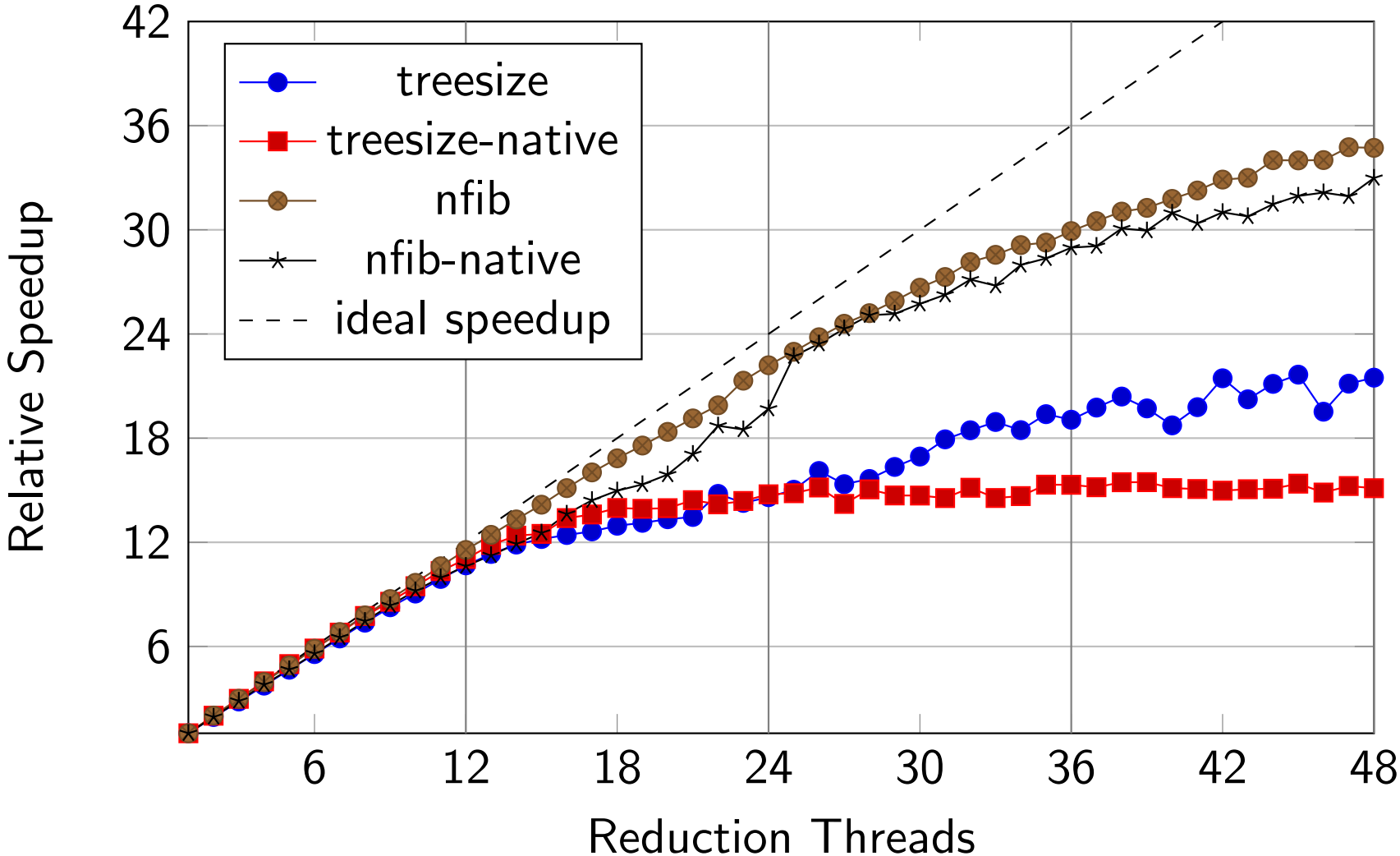
Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
  local ← shared;
  reduced ← reduce(local);
  while(local ≠ reduced) {
    if(compareAndSet(shared, local, reduced)):
      local ← reduced;
    } else {
      local ← shared;
    }
    reduced ← reduce(local);
  }
  return local;
}
```

Graph Reduction - Evaluation

TODO: Worker always makes progress on given task Bad for sequential computations

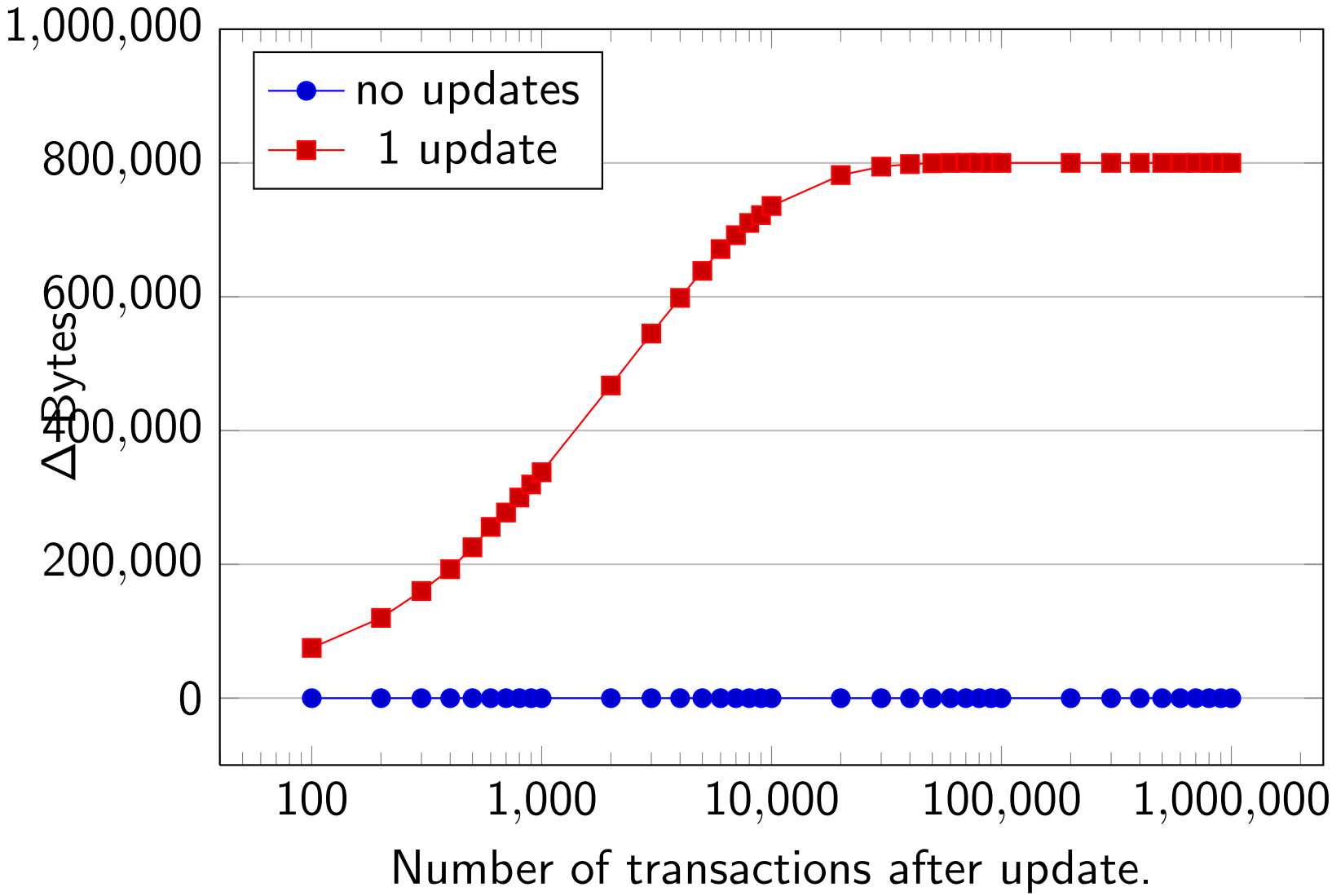
Evaluation


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
Δ Bytes
no updates
1 update
100
1,000
10,000
100,000
1,000,000
Number of transactions after update.

Conclusions

Results

We designed a functional language for transaction processing:

- Prototype implementation.

Conclusions

Results

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.

Conclusions

Results

We designed a functional language for transaction processing:

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

Conclusions

Results

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

Results

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.

Conclusions

- Persistent functional languages are feasible.
- There are still open problems.

Future Work

Future Work

(1) Lazy evaluation leads to space leaks.

(2) Practical use of system.

(3) Handling run-time errors.

(4) Concurrent data structures.

(5) Optimistic concurrency control.