Lazy Evaluation for Concurrent OLTP and Bulk Transactions

Lesley Wevers
Marieke Huisman Maurice van Keulen

March 4, 2014

Bitcoin bank Flexcoin shuts down after massive theft

Some $600,000 worth of bitcoins were stolen from the site

March 5, 2014

Yet another exchange hacked: Poloniex loses around $50,000 in bitcoin

Firm can't cover losses: "All balances will temporarily be deducted by 12.3%."

On March 4th, 2014, about 12.3% of the BTC on Poloniex was stolen.

How Did It Happen?

The hacker found a vulnerability in the code that takes withdrawals. Here's what happens when you place a withdrawal:

1. Input validation.
2. Your balance is checked to see if you have enough funds.
3. If you do, your balance is deducted.
4. The withdrawal is inserted into the database.
5. The confirmation email is sent.
6. After you confirm the withdrawal, the withdrawal daemon picks it up and processes the withdrawal.

The hacker discovered that if you place several withdrawals all in practically the same instant, they will get processed at more or less the same time. This will result in a negative balance, but valid insertions into the database, which then get picked up by the withdrawal daemon.

What happened?

	balance
...	...
Alice	10
...	...

withdraw 10

Request 1

Request 2

Request 3

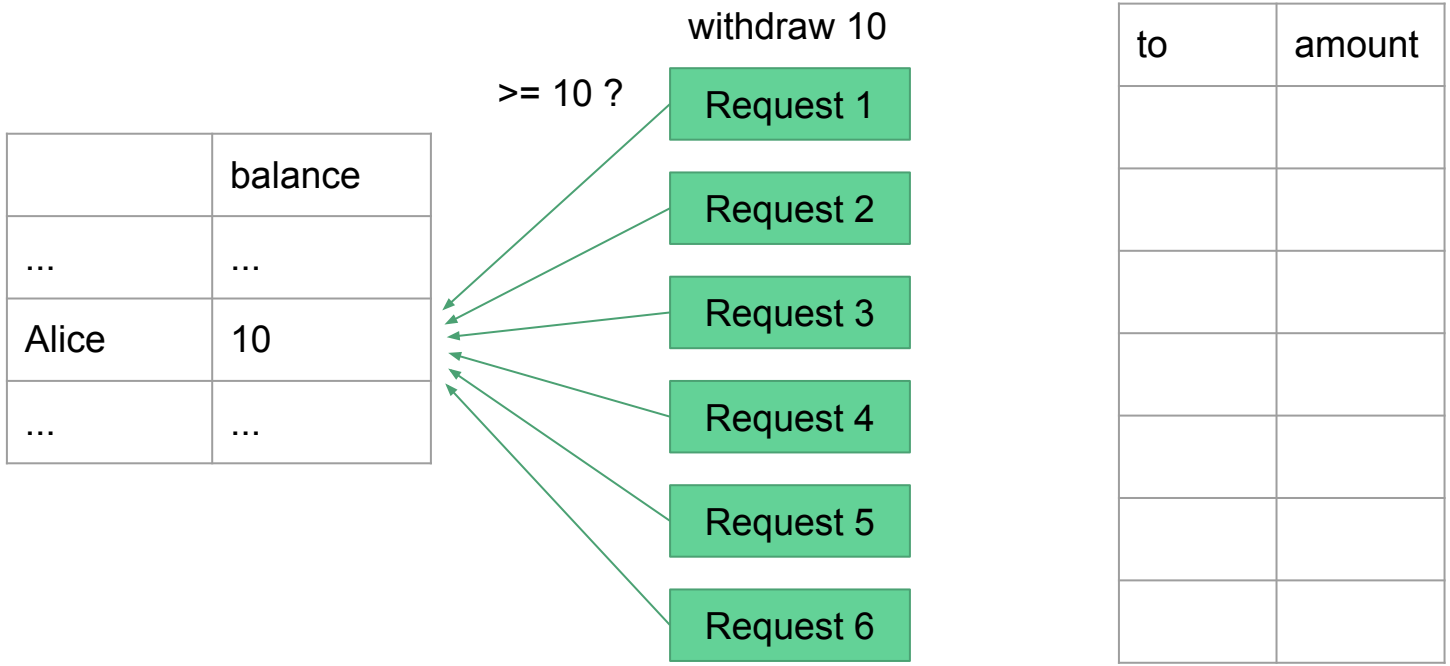
Request 4

Request 5

Request 6

to	amount

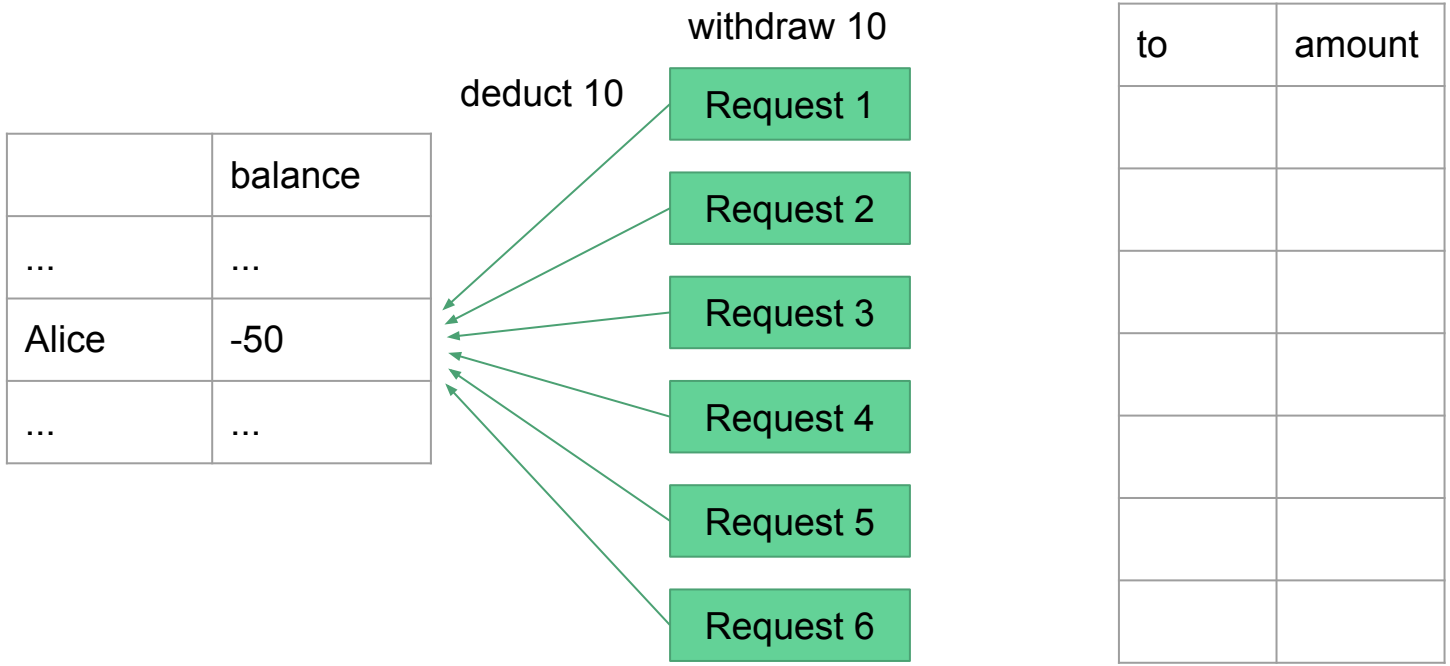
What happened?



	balance
...	...
Alice	10
...	...

to	amount

What happened?

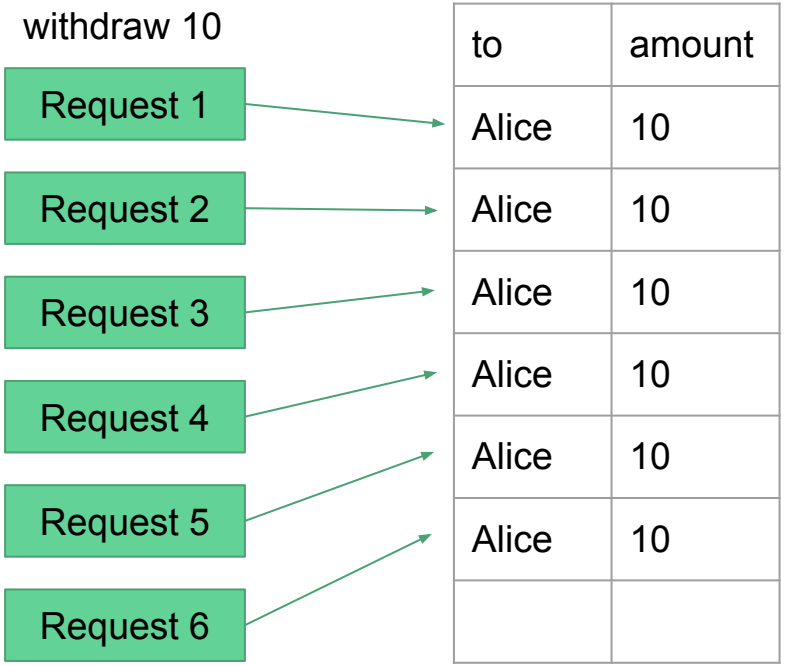


	balance
...	...
Alice	-50
...	...

to	amount

What happened?

	balance
...	...
Alice	-50
...	...



Transactions & Correctness

Databases provide **transactions** to ensure correct execution of database operations.

```
def withdraw(account, amount):
    atomic:
        if(balance[account] < amount)
            abort
        balance[account] -= amount
        withdrawals += (account, amount)
```

Correctness guarantees:

Atomicity

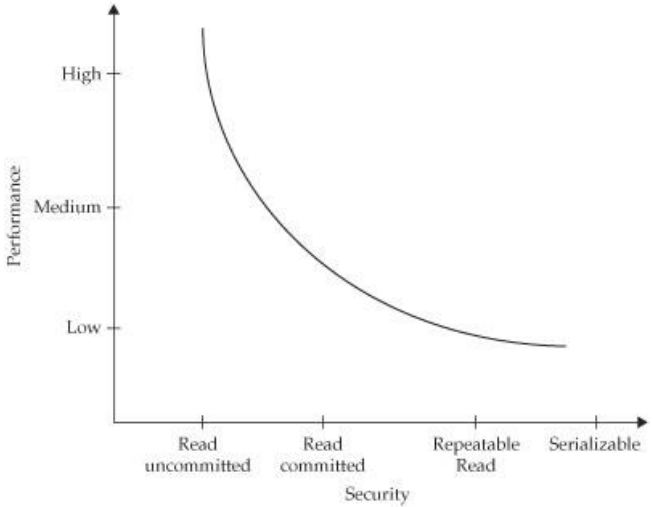
Consistency

Isolation

Durability

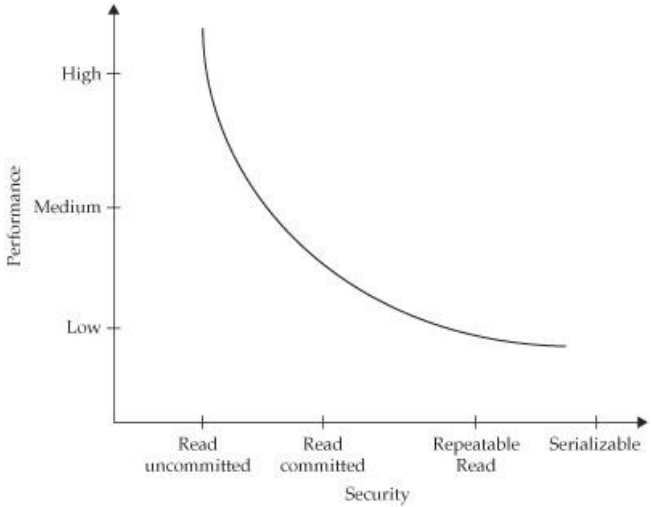
Weak Isolation

High volume database systems often trade correctness for performance.



Weak Isolation

High volume database systems often trade correctness for performance.



Can we make transactions faster without sacrificing correctness?

Transactional Workloads in Database Systems

	Variety	Footprint	Access pattern
OLTP Online transaction processing	Regular	Small	Read / write
OLAP Online analytical processing	Ad-hoc	Large / Everything	Read-only

Transactional Workloads in Database Systems

	Variety	Footprint	Access pattern
OLTP Online transaction processing	Regular	Small	Read / write
OLAP Online analytical processing	Ad-hoc	Large / Everything	Read-only
OLBP? Online bulk processing	Ad-hoc	Large / Everything	Read / write

Concurrency Control

Two-phase locking

- Lock data elements while executing the transaction.
- Do not release locks until the transaction has finished.

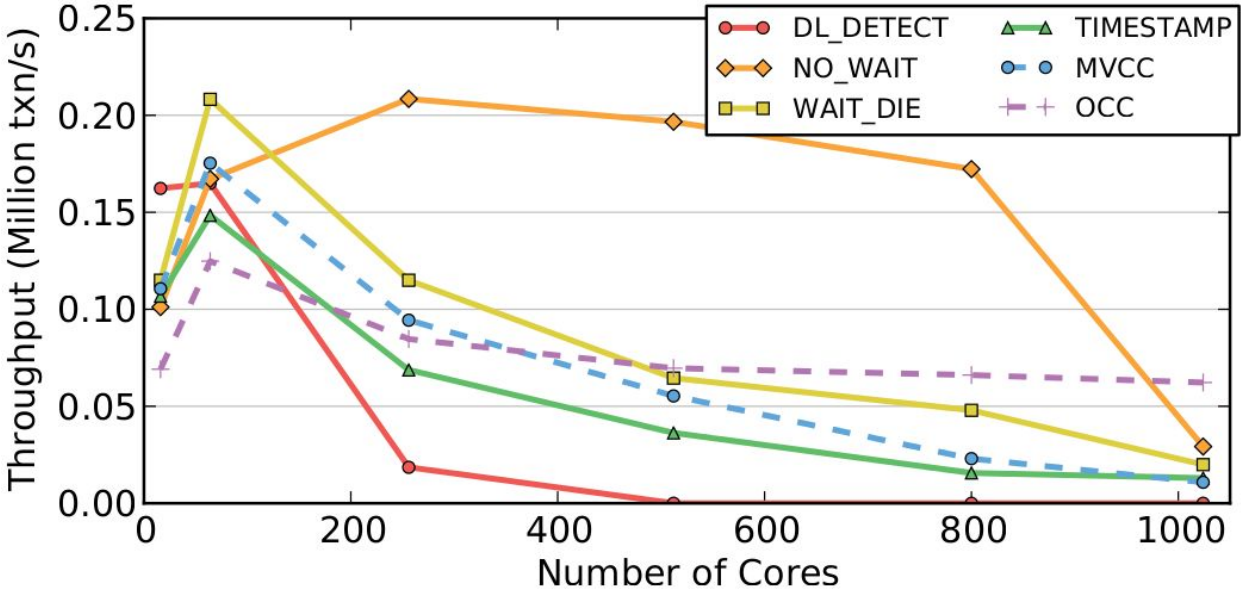
Optimistic concurrency control

- Check validity of a transaction after executing.
- Abort and re-execute if inconsistent.

Multi-version concurrency control

- Perform read-only transactions on a snapshot.

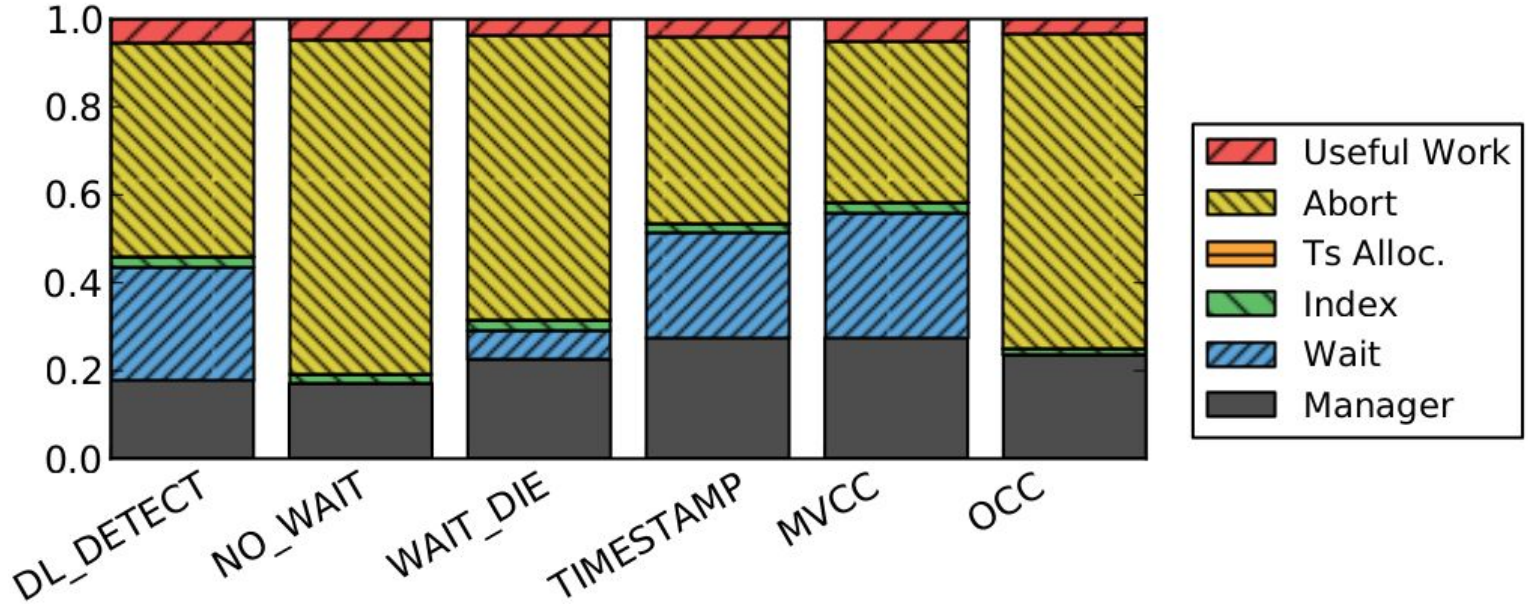
Multi-Core Scalability



The YCSB benchmark under high contention.

Xiangyao Yu et al. **Staring Into The Abyss: An Evaluation Of Concurrency Control With One Thousand Cores**. VLDB 2014.

Runtime Breakdown



Xiangyao Yu et al. **Staring Into The Abyss: An Evaluation Of Concurrency Control With One Thousand Cores**. VLDB 2014.

Contention

Sources of Contention in OLTP transactions

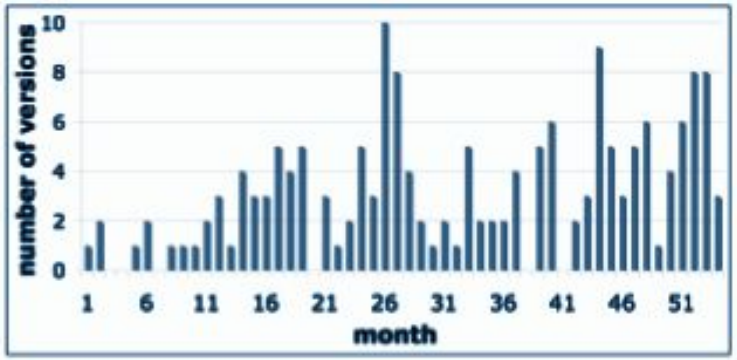
- Counters
- Summary data
- Popular items
- Newest / oldest items
- Correlated access
- Zipfian access skew

Bulk Updates

Bulk updates content with most or all concurrent transactions

- Bulk operations are often required in schema transformations

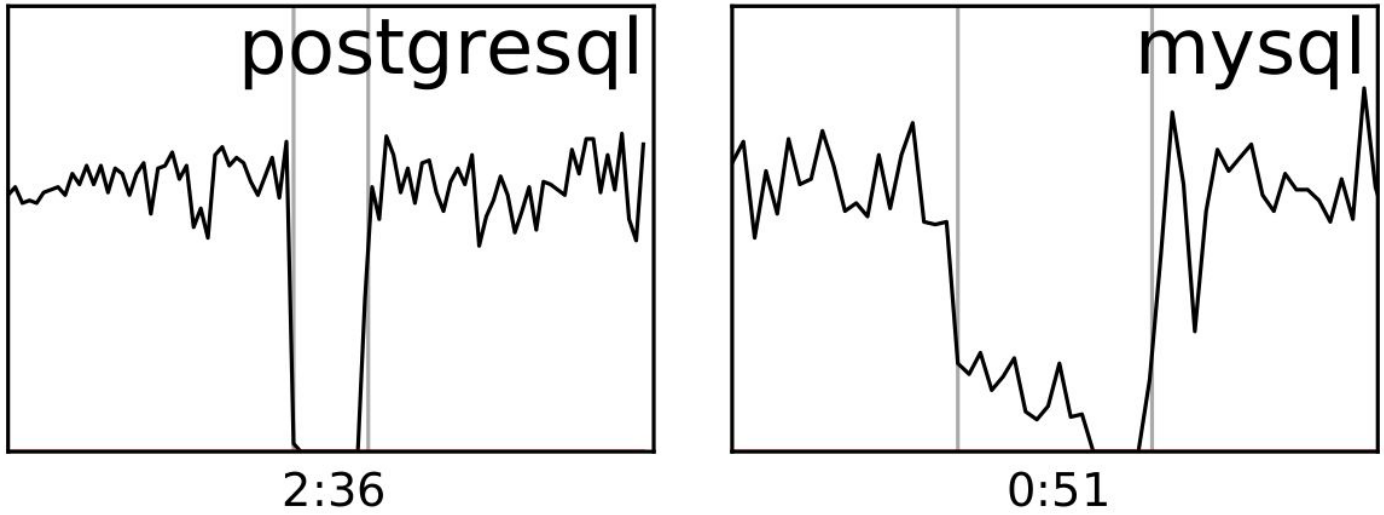
WikiMedia schema revisions:



- 90% require a write lock.
- Largest took 22 hours to complete for wikipedia.

Curino et al. http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Schema_Evolution_Benchmark

Bulk Updates



Bulk updates block all concurrent transactions, or are executed incorrectly.

L. Wevers et al. **A Benchmark for Online Non-Blocking Schema Transformations**. DATA 2015.
L. Wevers et al. **Analysis of the Blocking Behaviour of Schema Transformations in Relational Database Systems**. ADBIS 2015.

Contributions

Existing concurrency control mechanisms:

- Cannot execute contended workloads concurrently
- Block concurrent transactions during bulk operations

We have developed a concurrency control mechanism that can:

- Significantly improve parallelism in contended workloads
- Execute bulk operations without blocking

Lazy Transactions

Lazy Transactions

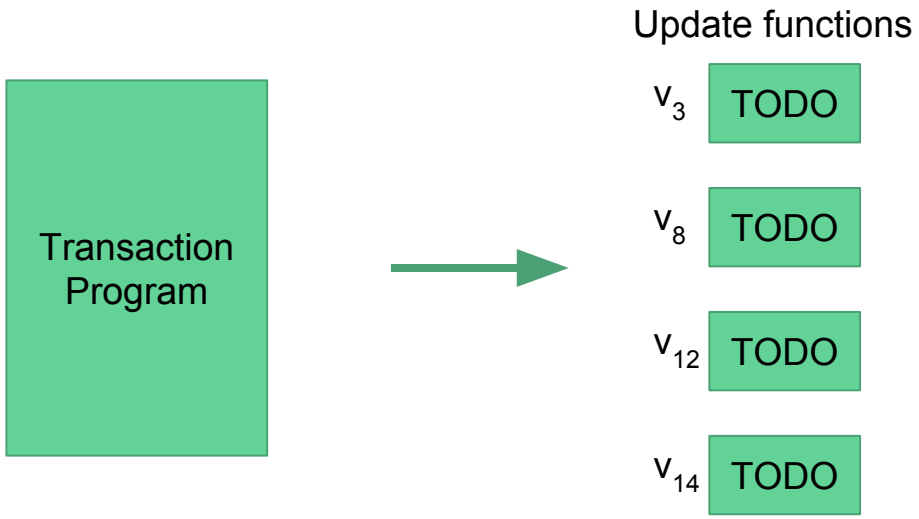
Main Ideas

- Re-order operations within a transaction based on demand from readers
- Use results from partially executed transactions

Mechanism: move responsibility of evaluation from writers to readers

- Transactions commit before executing their operations
- Readers can prioritize evaluation of those operations that they need

Mechanism



Split a transaction into **update functions** on variables.

Mechanism

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO

Mechanism

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

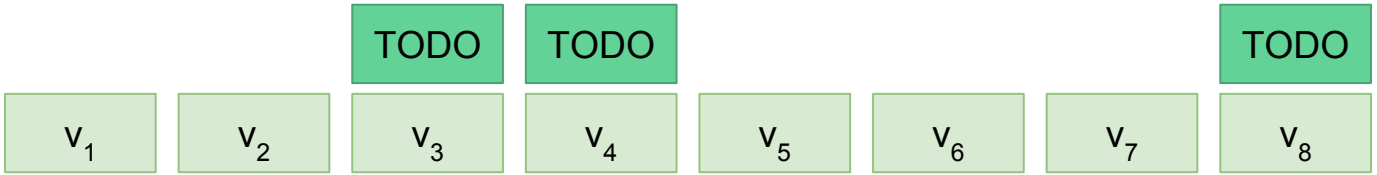
Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO



Mechanism

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

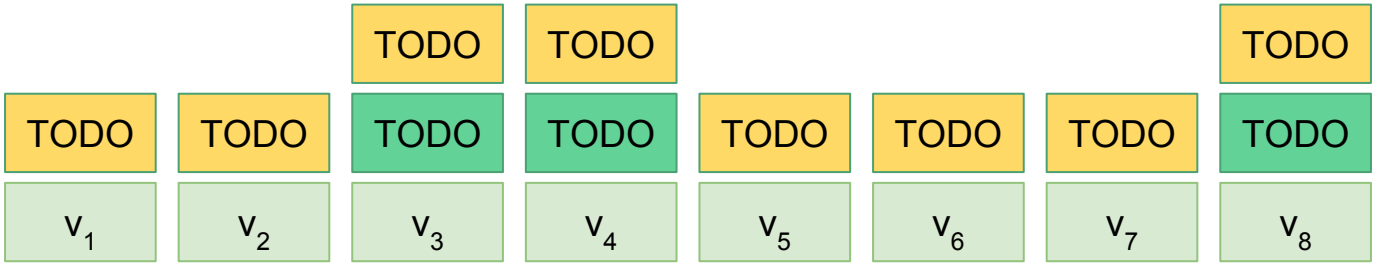
Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO



Mechanism

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

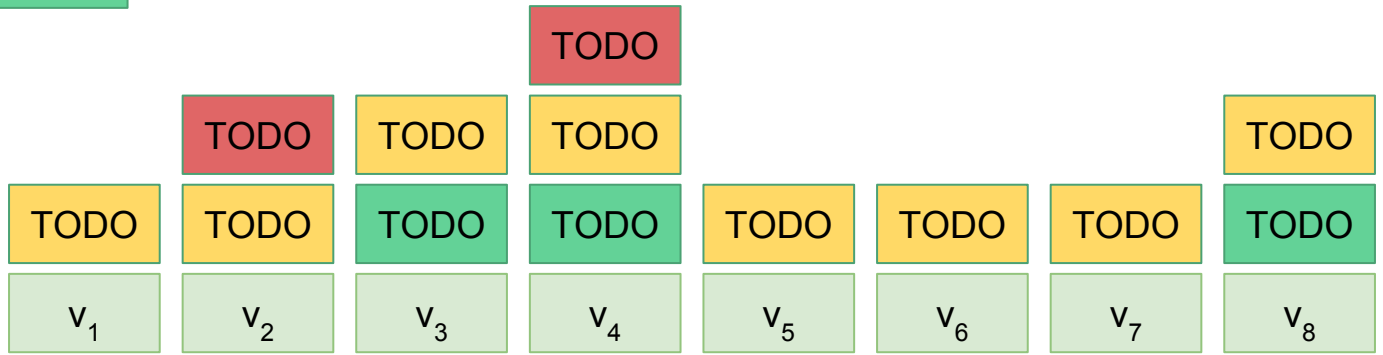
Transaction 2

$v_{1..8}$ TODO

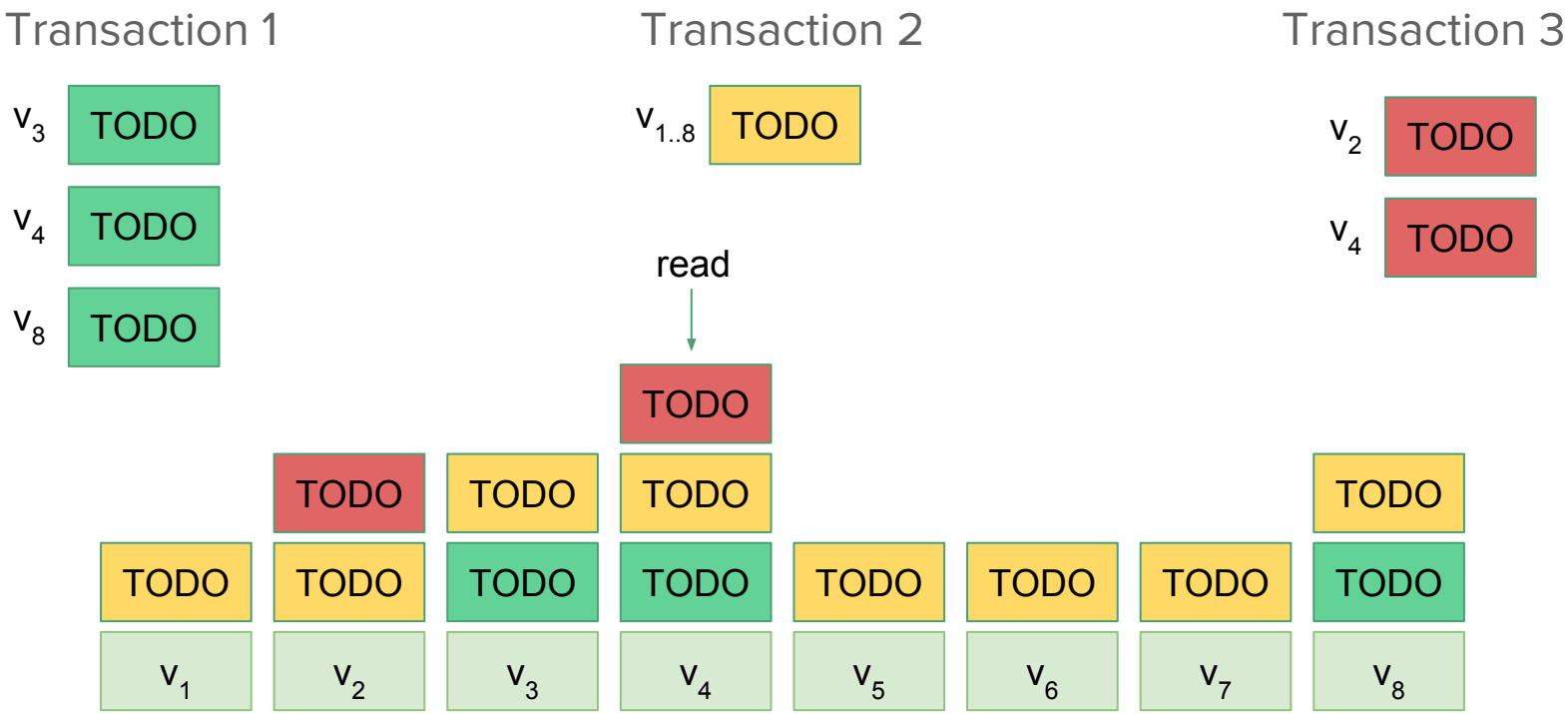
Transaction 3

v_2 TODO

v_4 TODO



Mechanism


Transaction 1
v3 TODO
v4 TODO
v8 TODO
Transaction 2
v1..8 TODO
read
Transaction 3
v2 TODO
v4 TODO
TODO
TODO TODO TODO TODO
TODO TODO TODO TODO TODO TODO TODO TODO
v1 v2 v3 v4 v5 v6 v7 v8

Mechanism

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

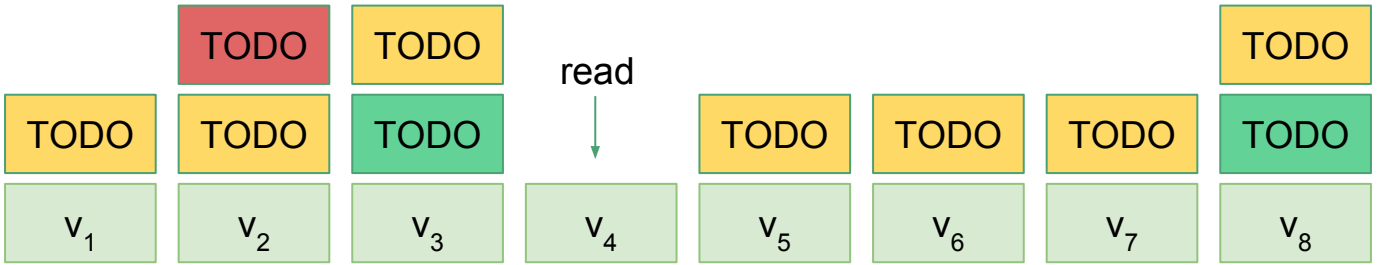
Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO



Encoding Transactions

Transaction Execution Phases

1. **Preparation Phase**
   - Prepare update functions
   - **Optimistic reads** on a snapshot
2. **Commit Phase**
   - Queue update functions
   - **Blocking reads** on the current state
3. **Lazy Phase**
   - Work performed inside update functions
   - **Lazy reads** on a snapshot
4. **Result Phase**
   - Compute observable results from a snapshot

Lazy Reads

commit:

```
update(y, get(x))
```

commit:

```
s <- snapshot
update(y, v => s.get(x))
```

Optimistic Reads

commit:

```
update(get(y), a)
```

prepare:

```
x <- get(y)
```

commit:

```
s <- snapshot
update(x, v =>
    if(x == s.get(y))
        a
    else
        v
)
```

Reading transaction results

commit:

```
update(x, v)
return get(x) + get(y) + get(z)
```

commit:

```
update(x, v)
s <- snapshot
```

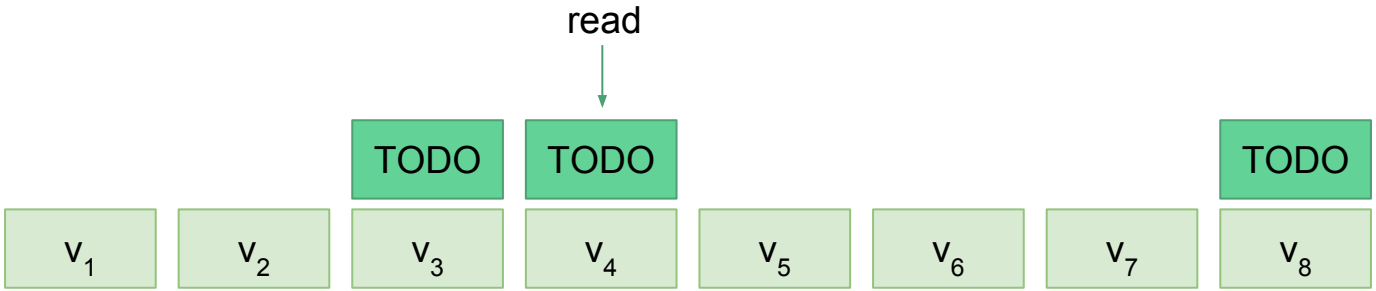
result:

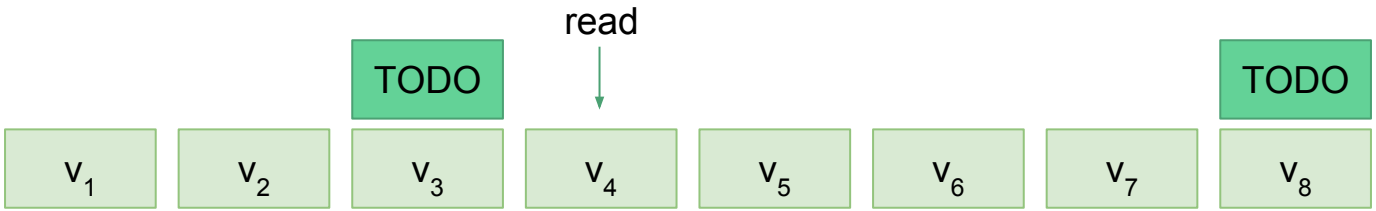
```
return s.get(x) + s.get(y) + s.get(z)
```

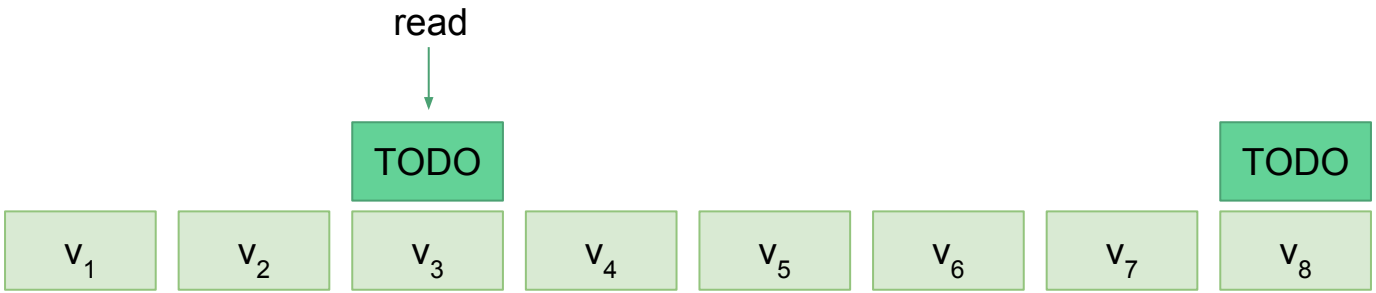
Read-only

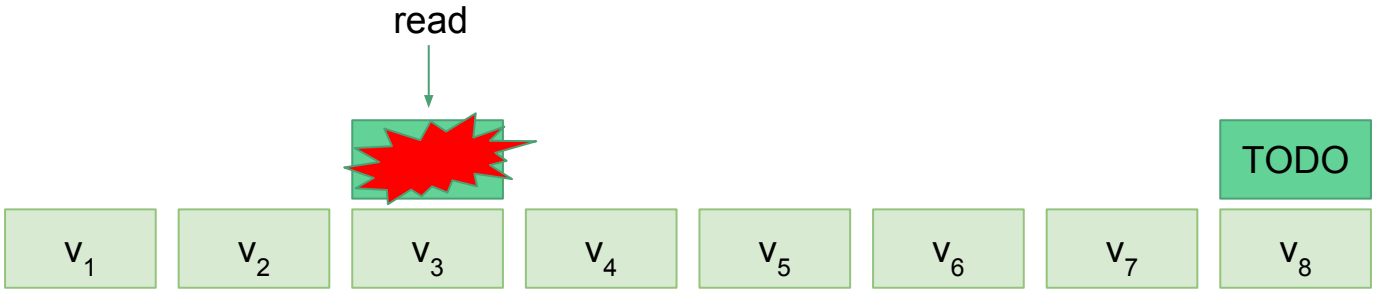
prepare:

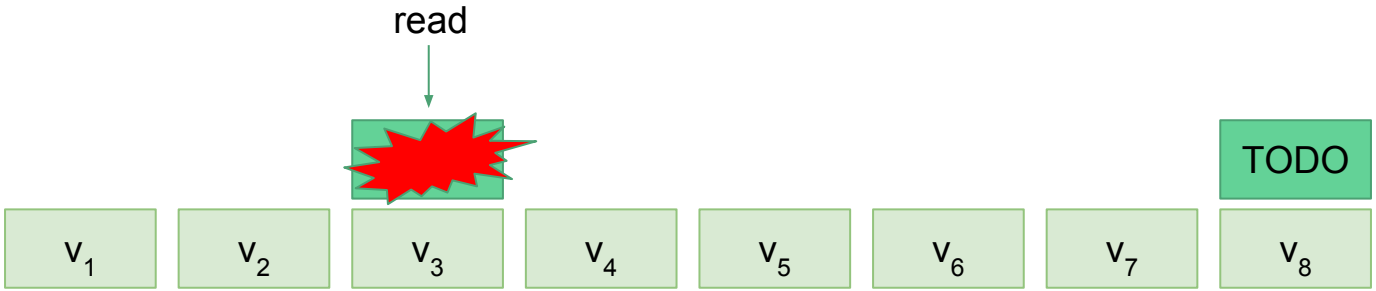
```
s <- snapshot
return s.get(x) + s.get(y) + s.get(z)
```


read
TODO
TODO
TODO
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8


read
TODO
TODO
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8


read
TODO
TODO
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8


read
TODO
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8



How do we deal with errors?

Aborts

If a transaction **aborts**: all update functions must return the original value
If a transaction **commits**: all update functions must not fail

Check for errors before committing

commit:

```
s <- snapshot
lazy check = s.get(x) > 10
update(x, v => if(check) f(v) else v)
update(y, v => if(check) g(v) else v)
update(z, v => if(check) h(v) else v)
```

Lazy Indexes & Bulk Operations

Lazy Indexes

Indexes are the basic building block of database management systems

- Access by primary key
- Secondary indexes

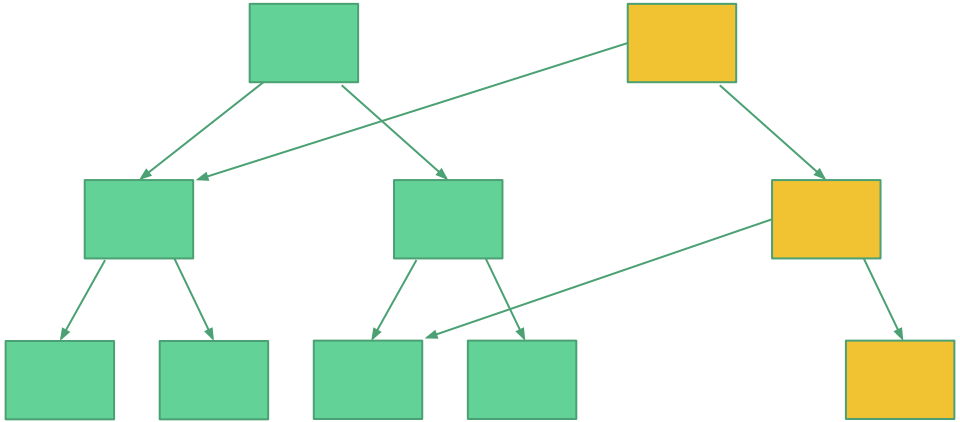
Goal: develop an index structure for lazy transactions

- Single-key and bulk operations
- Sequential access by writers to queue lazy operations
- Concurrent access by readers, which evaluate lazy operations
- Snapshots, for lazy reads

Immutable Data Structures

We use immutable data structures to provide snapshots

- Instances cannot be mutated: any instance is a snapshot
- Updates create a new version of the data structure
- Unmodified parts are shared (copy-on-write)



How can we atomically write many update functions without blocking?

Transaction 1

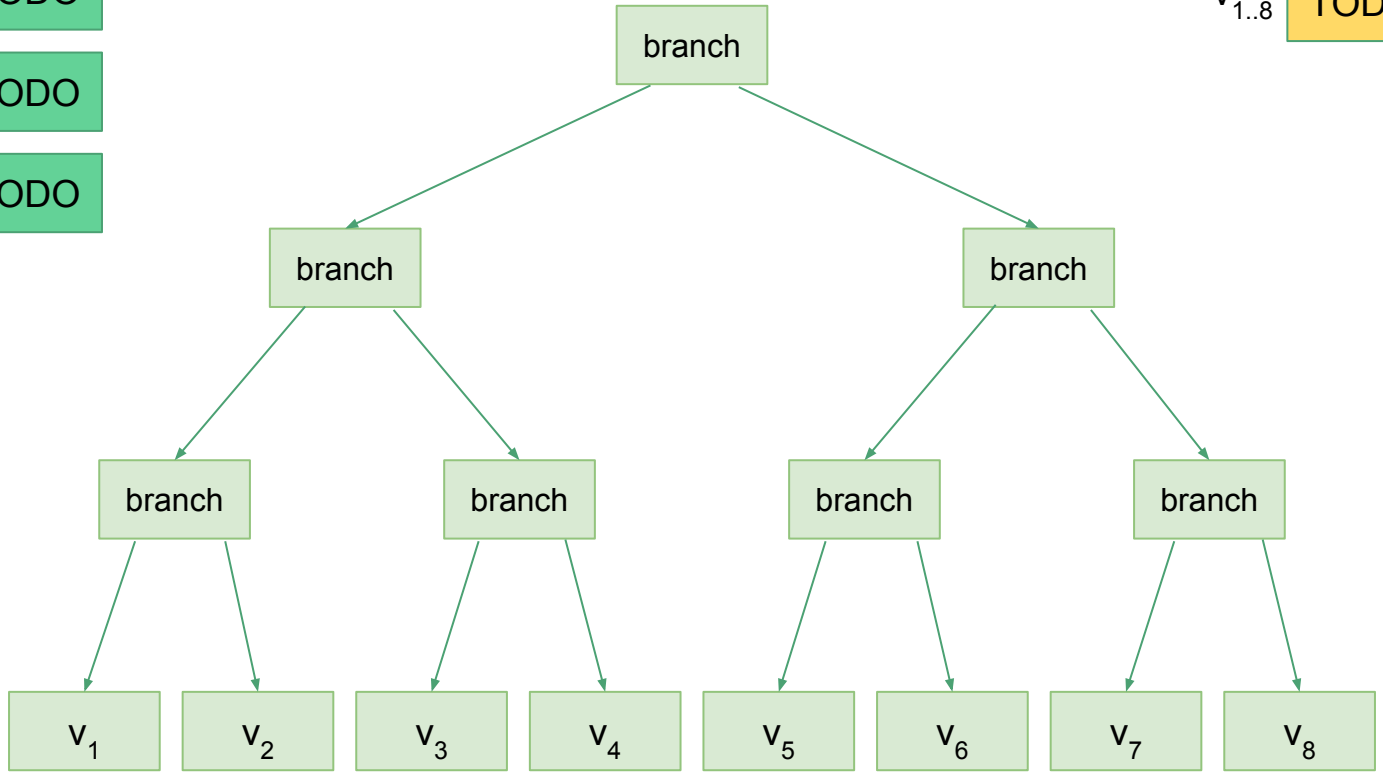
v_3 TODO

v_4 TODO

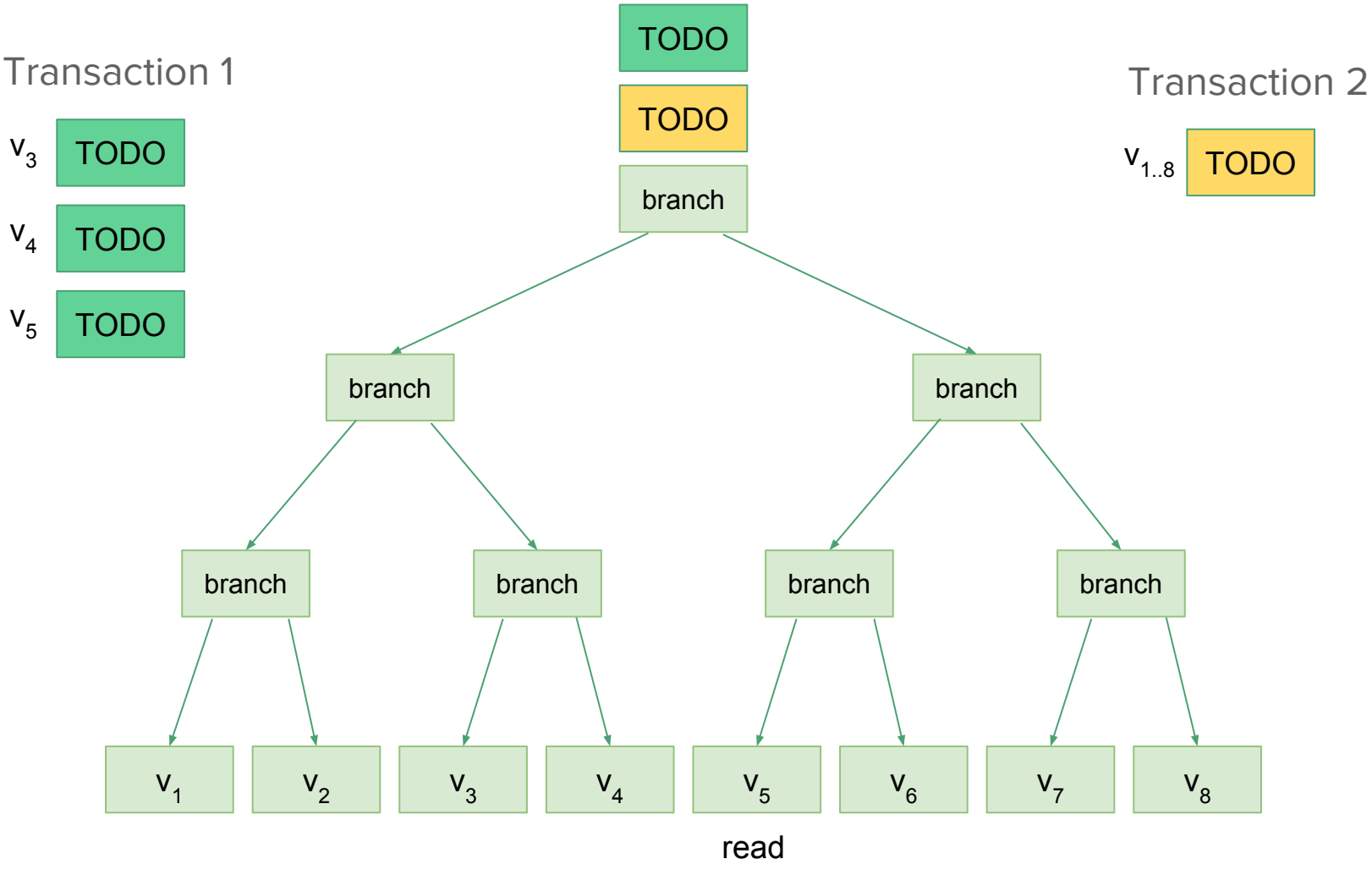
v_5 TODO

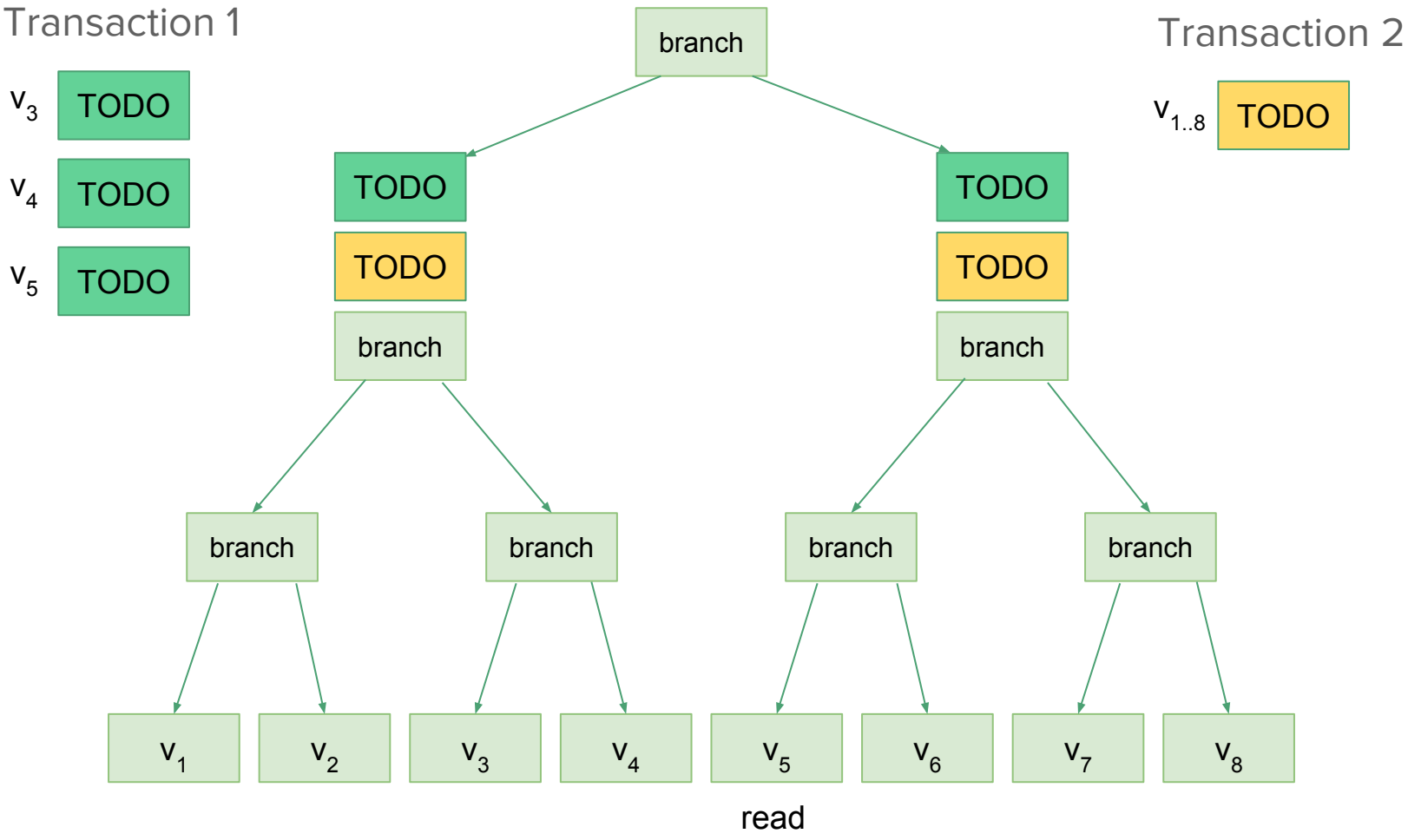
Transaction 2

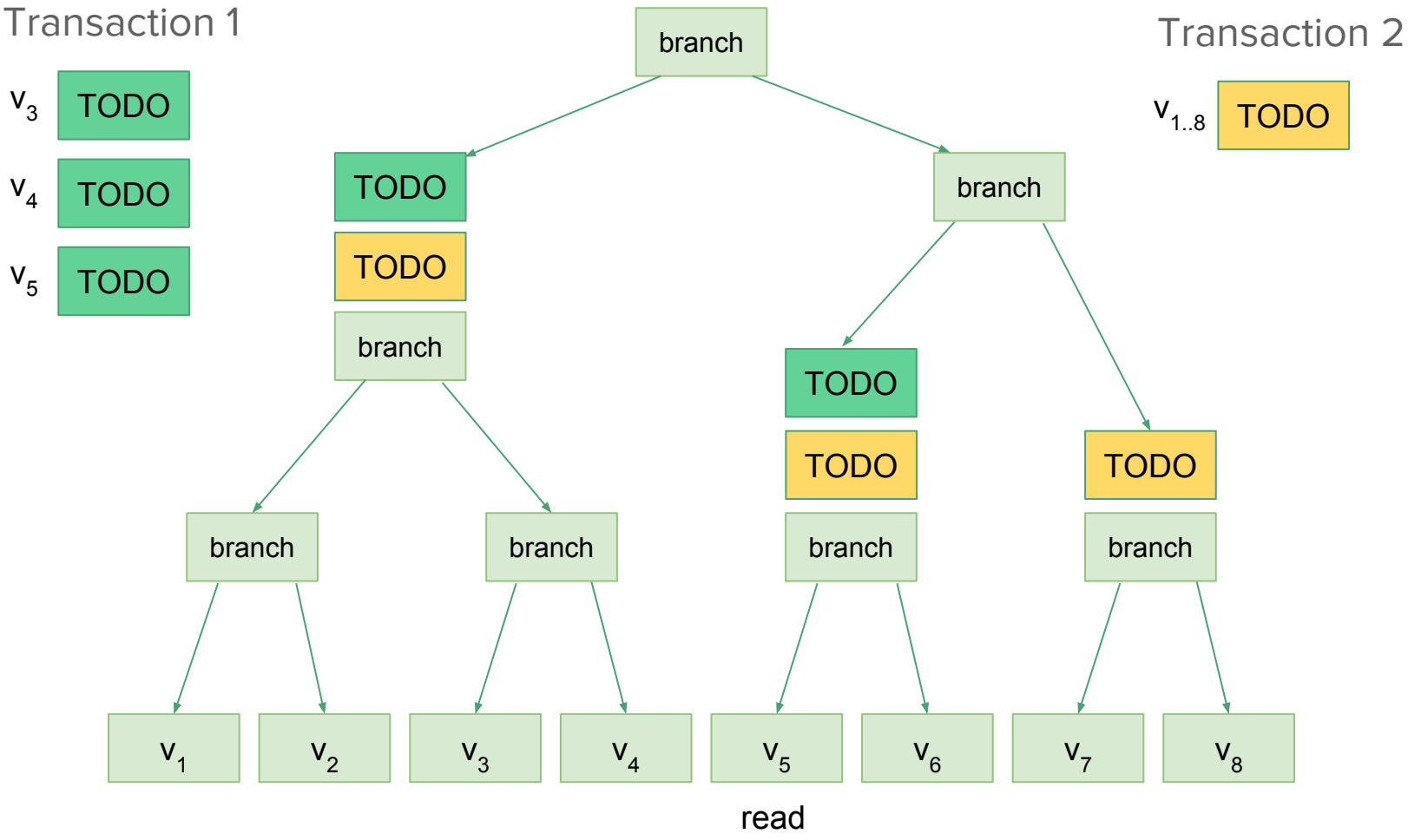
$v_{1..8}$ TODO

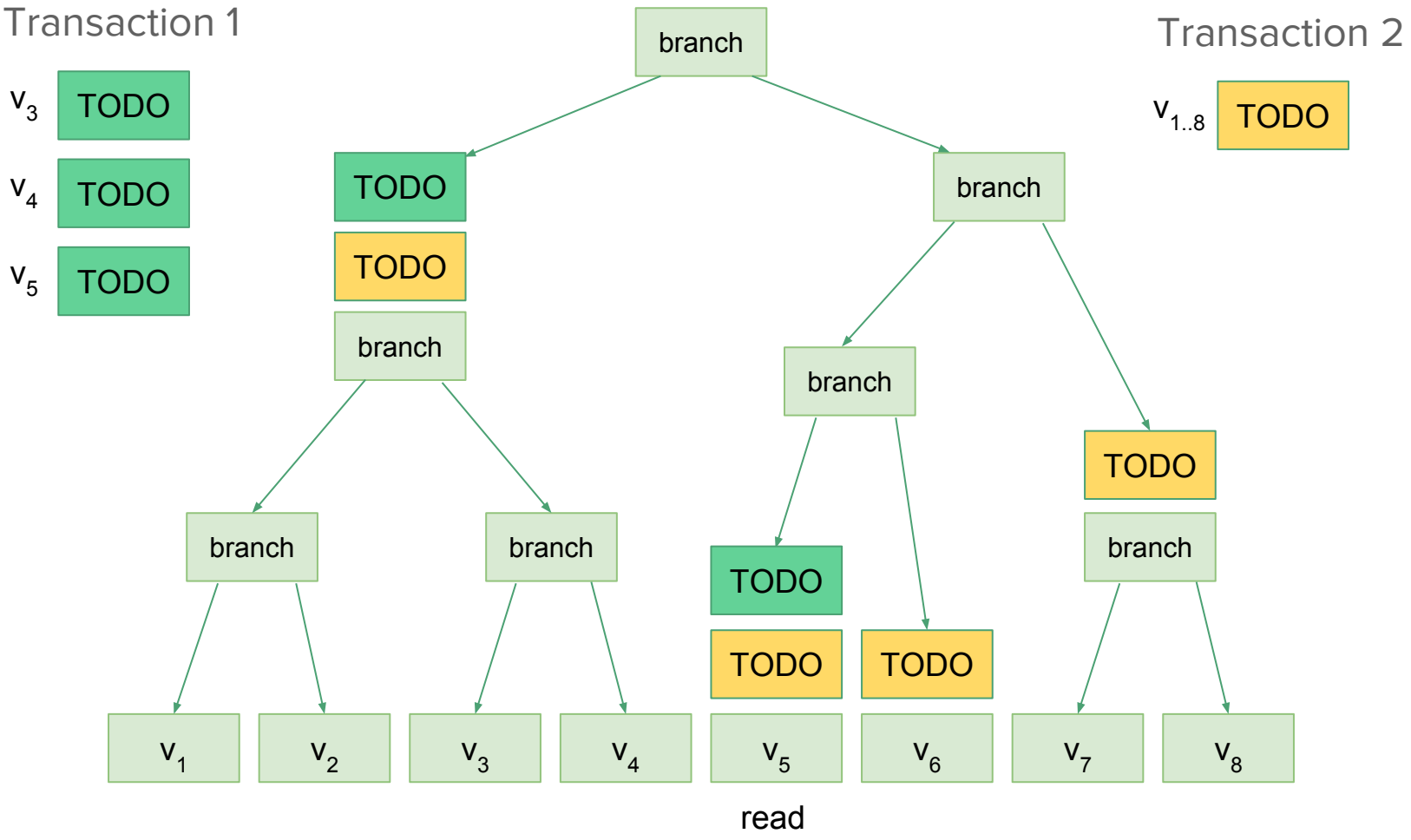


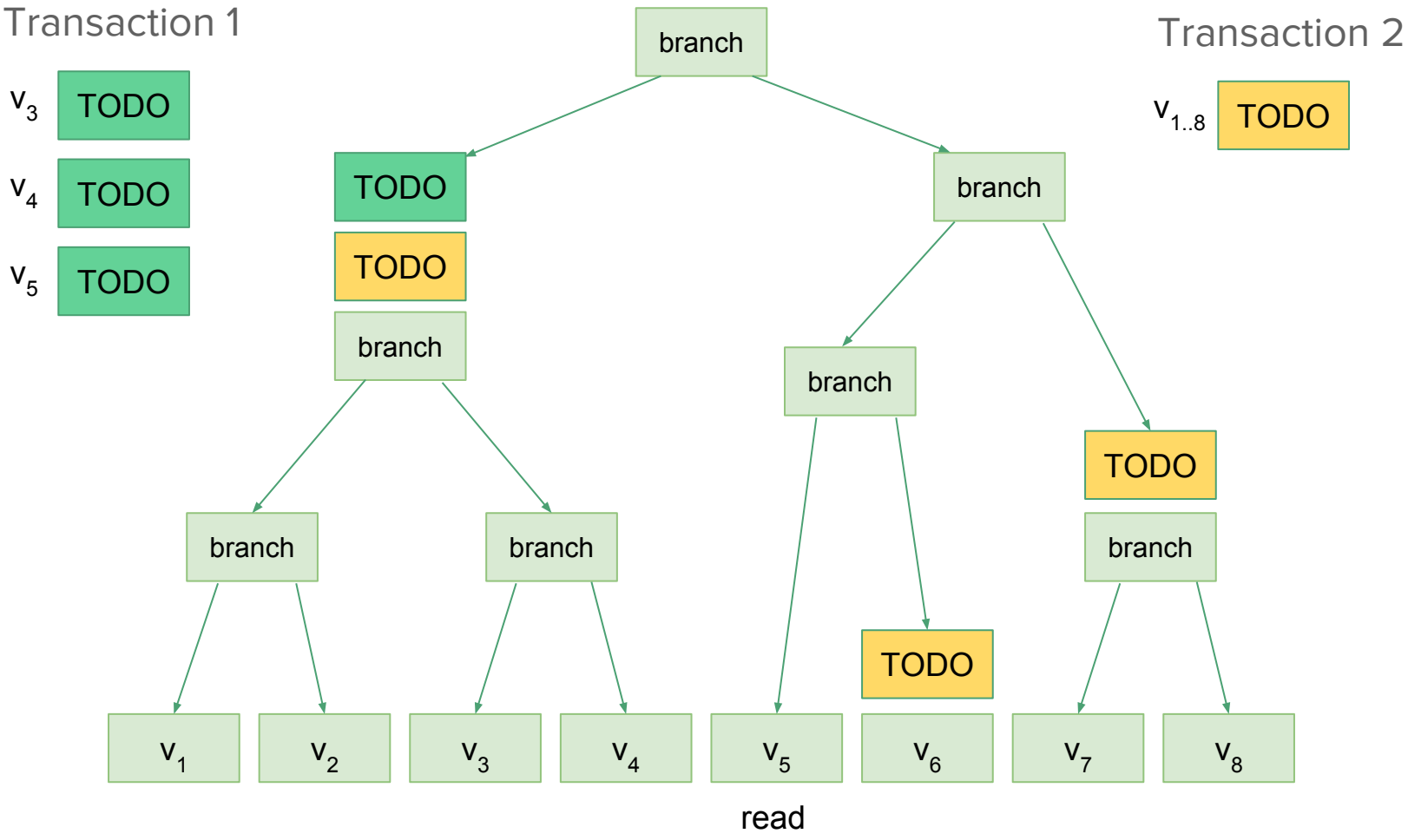
read


Transaction 1
v_3 TODO
v_4 TODO
v_5 TODO
TODO
TODO
branch
Transaction 2
$v_{1..8}$ TODO
branch
branch
branch
branch
branch
branch
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
read


Transaction 1
v_3 TODO
v_4 TODO
v_5 TODO
branch
TODO
TODO
branch
TODO
TODO
branch
branch
branch
branch
branch
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
Transaction 2
$v_{1..8}$ TODO
read


Transaction 1
v_3 TODO
v_4 TODO
v_5 TODO
Transaction 2
$v_{1..8}$ TODO
branch
TODO
TODO
branch
branch
TODO
TODO
branch
TODO
branch
branch
branch
branch
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
read


Transaction 1
v3
TODO
v4
TODO
v5
TODO
Transaction 2
v1..8
TODO
branch
TODO
TODO
branch
branch
branch
branch
branch
TODO
branch
branch
TODO
TODO
TODO
branch
v1
v2
v3
v4
v5
v6
v7
v8
read


Transaction 1
v_3 TODO
v_4 TODO
v_5 TODO
Transaction 2
$v_{1..8}$ TODO
branch
TODO
TODO
branch
branch
branch
branch
TODO
branch
branch
branch
TODO
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
read

Tries

Trie[K,V]

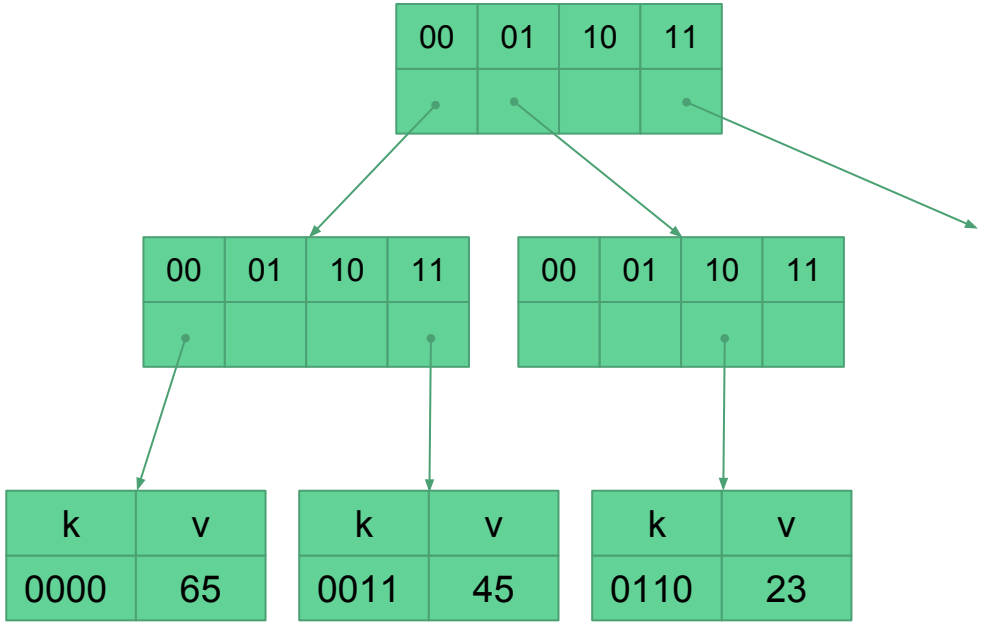
Empty

Leaf(k : K, v : V)

Branch(children : Array[Trie[K,V]])

Benefits

- Static balancing scheme
- Wide branches
- Ordered
- Snapshots: copy on write



Lazy Tries

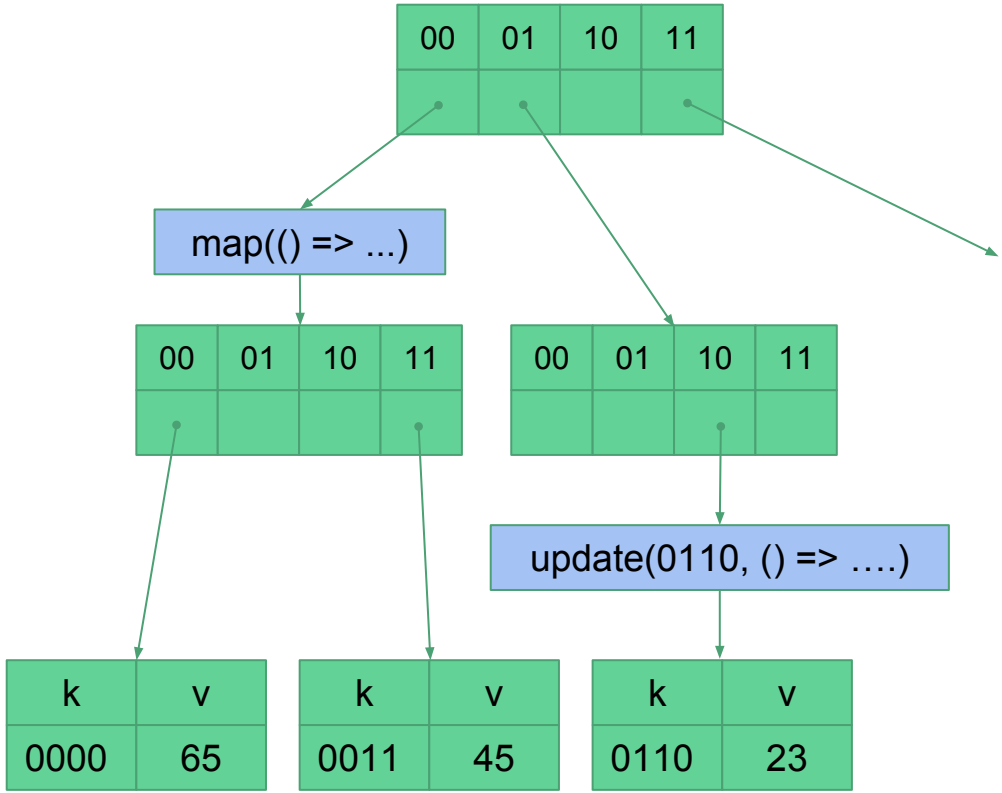
Trie[K,V]

Empty

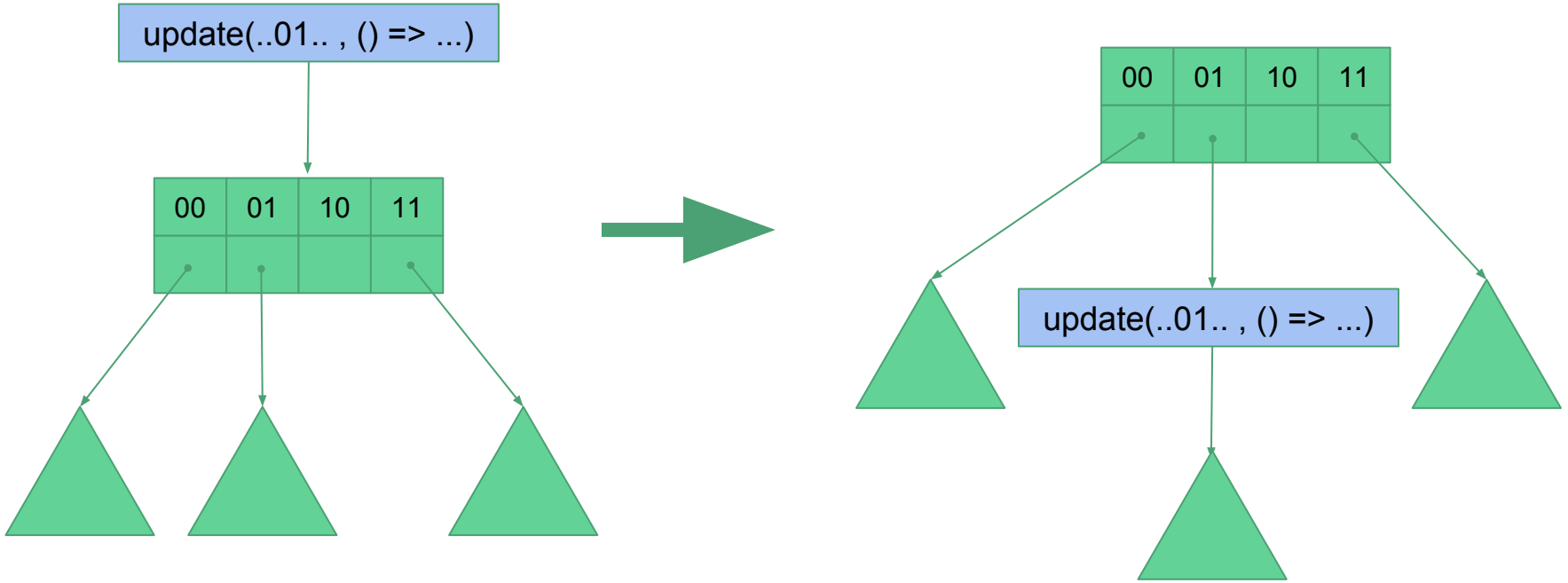
Leaf(k : K, v : **Lazy[** V **]**)

Branch(

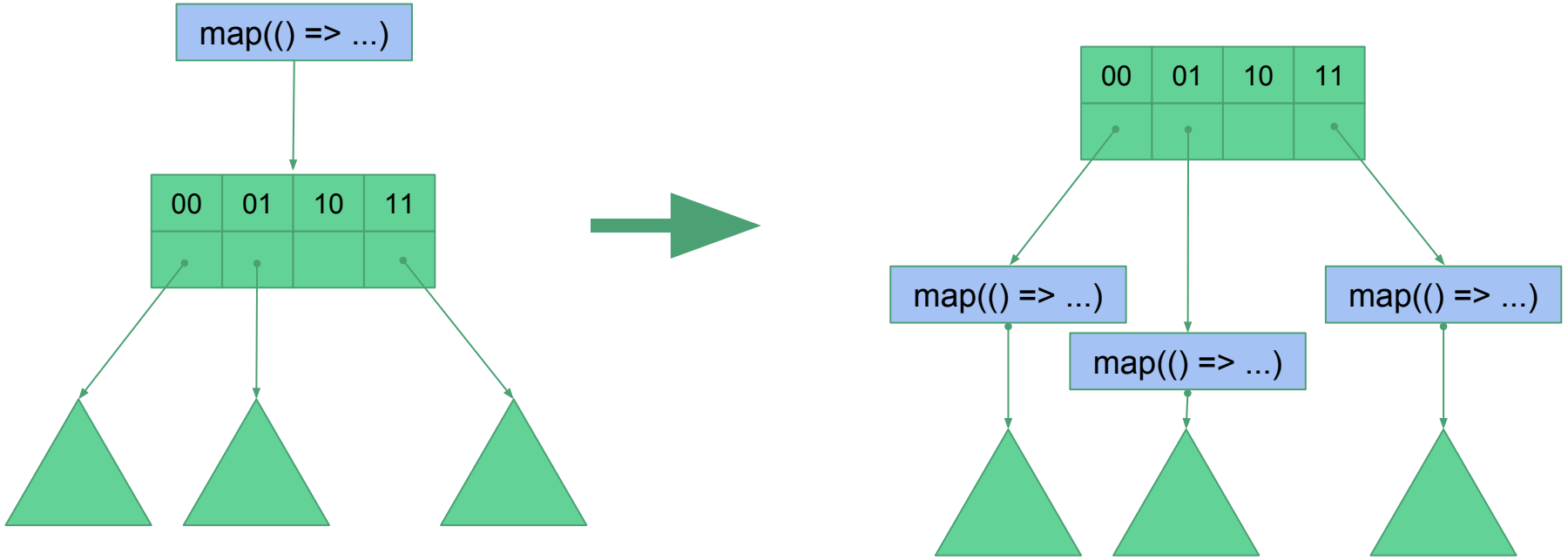
children : Array[**Lazy[** Trie[K,V] **]**])



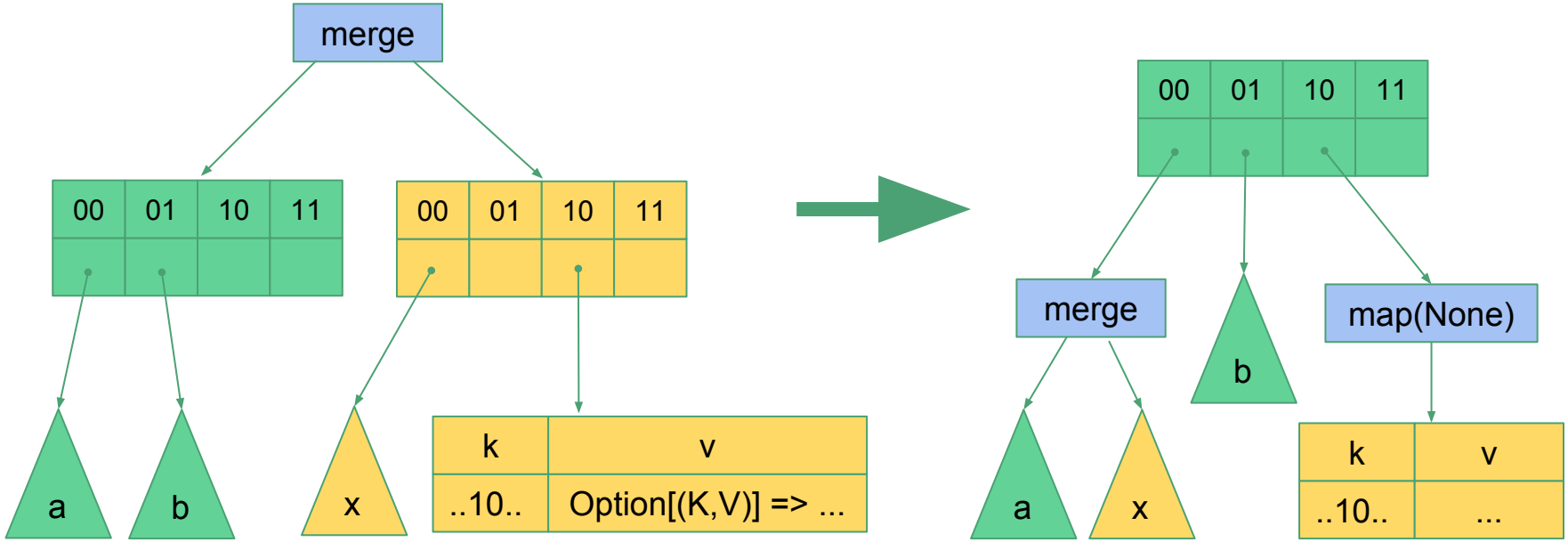
Single key updates



Bulk range updates



Bulk random-access updates



Scheduling Lazy Operations

Immediate Evaluation

commit:

```
update(x, v => ...)
update(y, v => ...)
update(z, v => ...)
s <- snapshot
```

result:

```
s.get(x)
s.get(y)
s.get(z)
```

Scheduling Opportunities

Delay evaluation of operations that are not needed immediately

+ Better temporal locality
+ Lower write latency
+ Can avoid work
− Higher read latency
− Higher memory consumption

Execute lazy operations in batches

+ Better spatial locality
+ Amortization of scheduling overhead
− Higher read latency

Experimental Results

Experimental Setup

We have implemented a lazy trie in Scala (JVM)

How does our system handle:

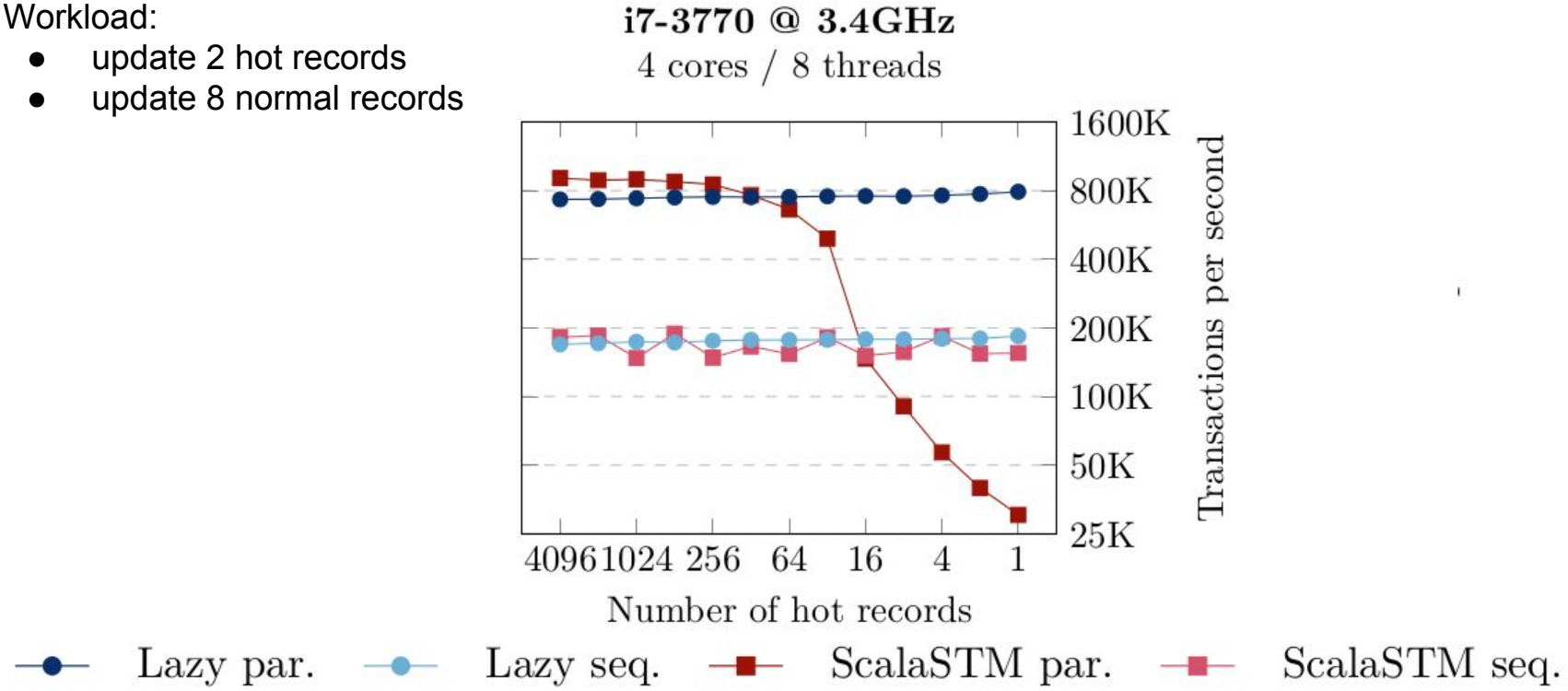
- Highly contended workloads
- Bulk operations: effect on concurrent transactions
- Realistic workloads: TPC-C

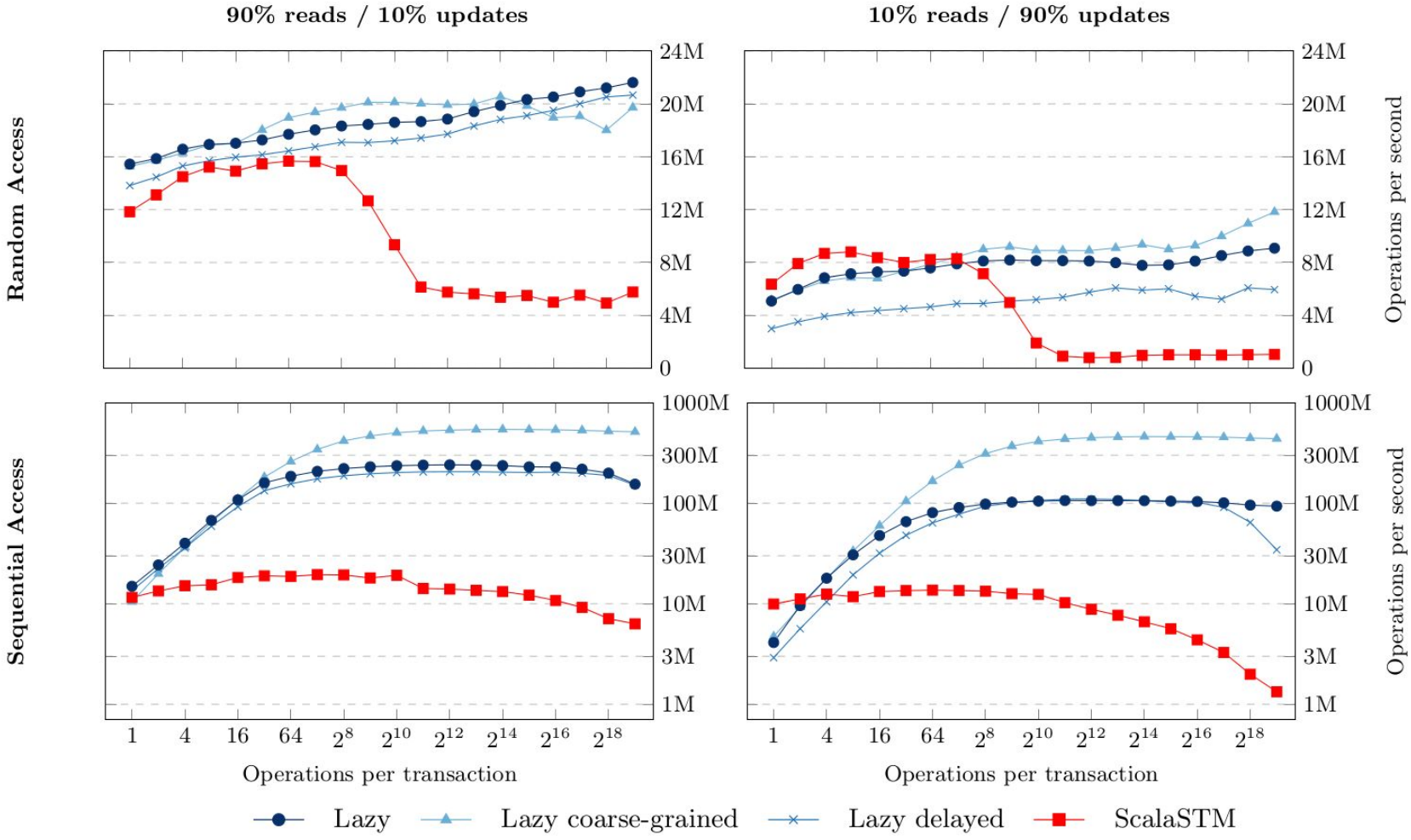
We compare against ScalaSTM

- Software transactional memory for Scala
- Transactional maps
- Optimistic concurrency control

Workload:

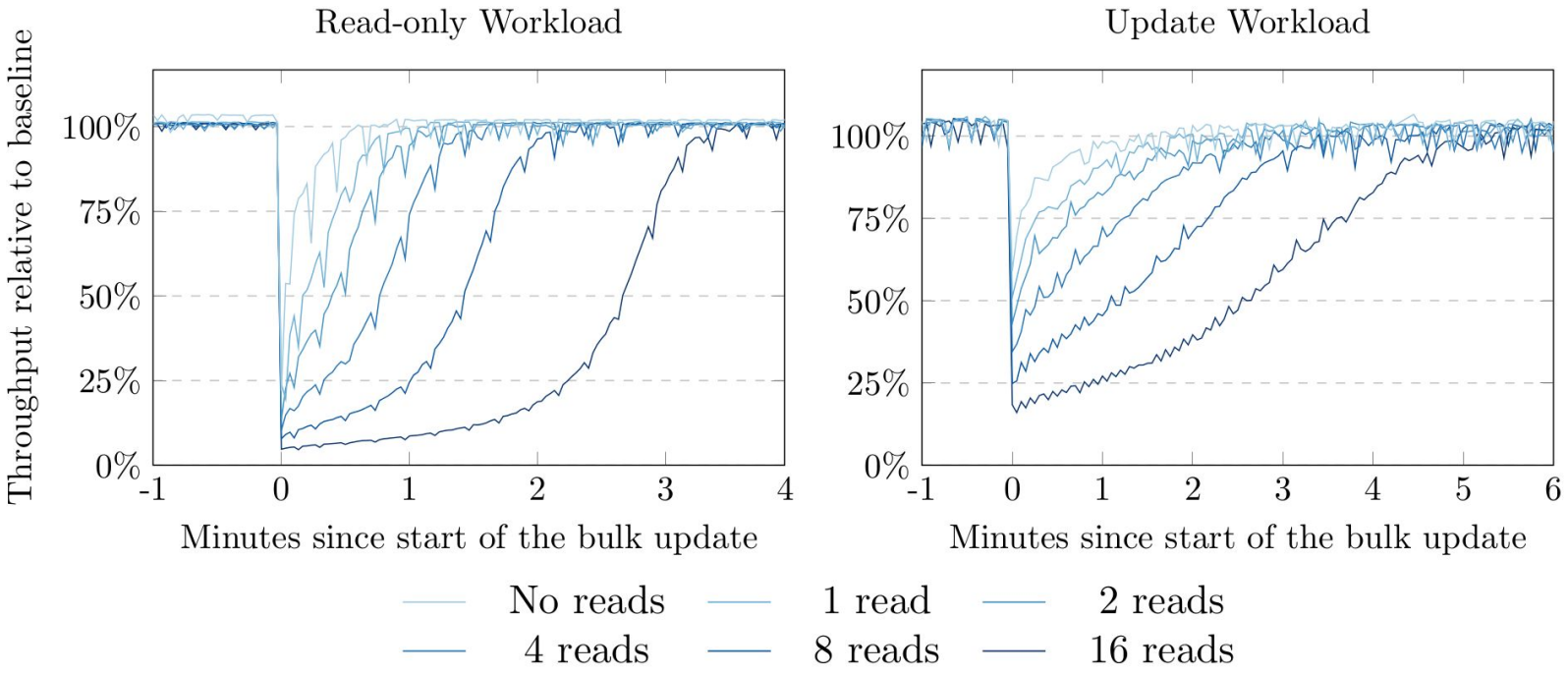
- update 2 hot records
- update 8 normal records

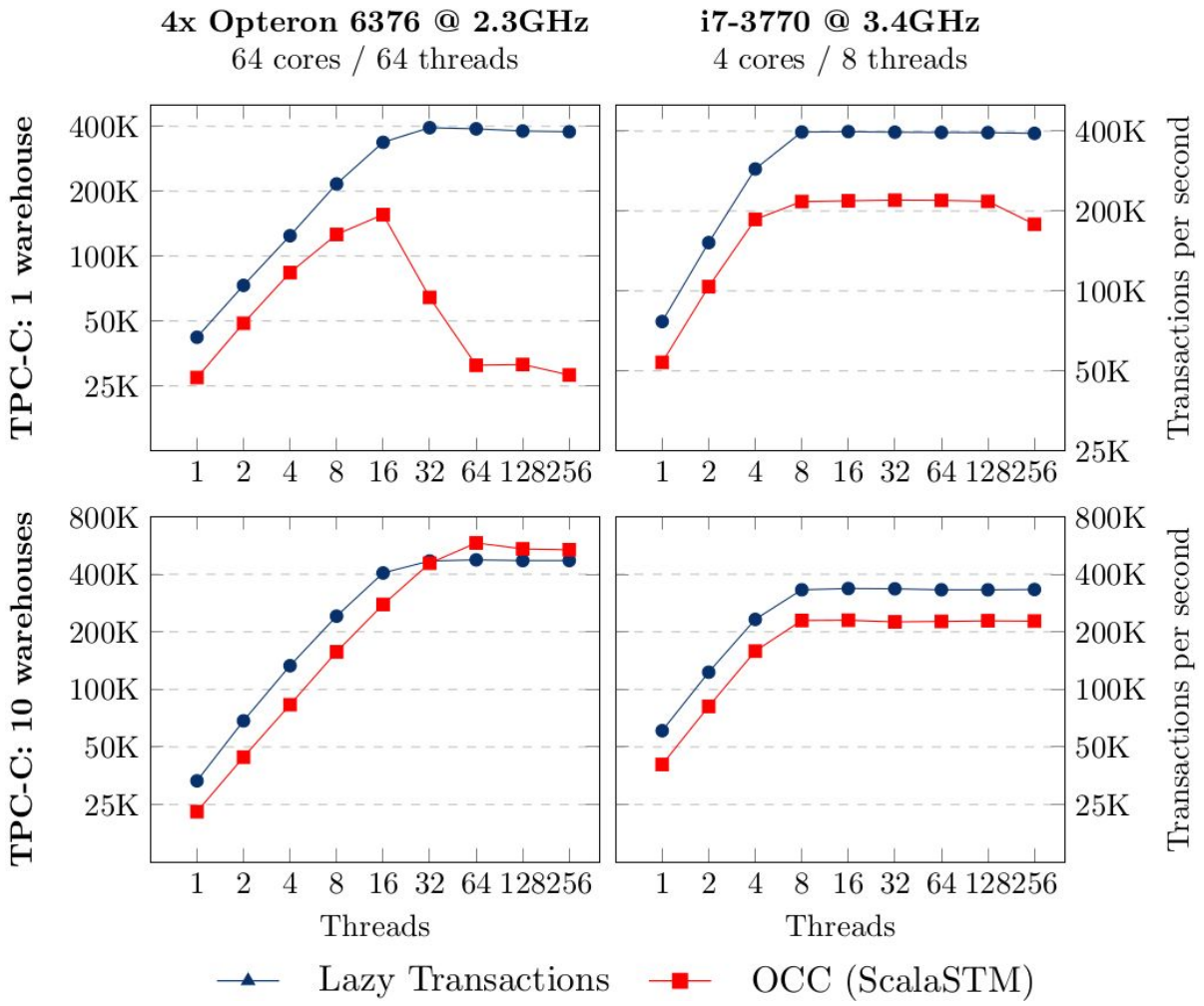


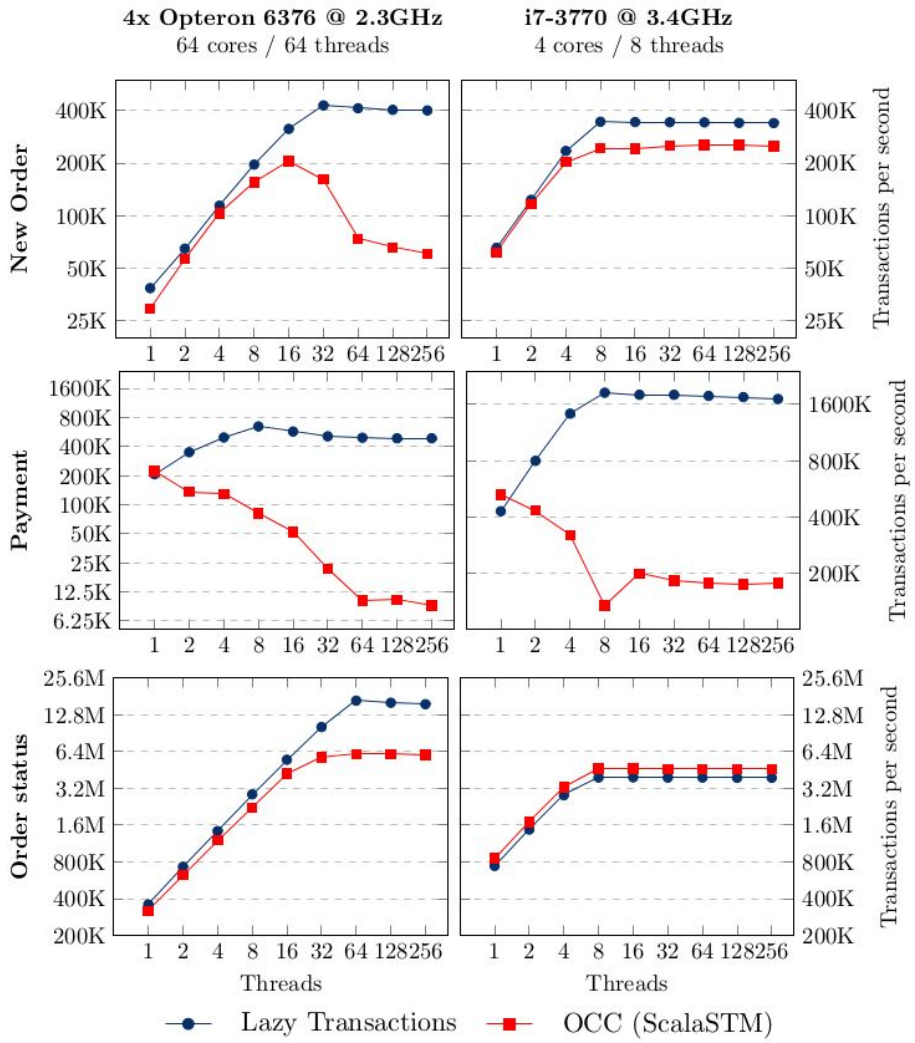

90% reads / 10% updates
10% reads / 90% updates
Random Access
Sequential Access
Operations per second
24M
20M
16M
12M
8M
4M
0
1000M
300M
100M
30M
10M
3M
1M
1
4
16
64
2^8
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}
Operations per transaction
Lazy
Lazy coarse-grained
Lazy delayed
ScalaSTM


1 read per write
10 reads per write
Lazy Reads
Optimistic Reads
24M
20M
16M
12M
8M
4M
0
Operations per second
1
4
16
64
2^8
2^{10}
2^{12}
2^{14}
2^{16}
2^{18}
Writes per transaction
Lazy
Lazy coarse-grained
Lazy delayed
ScalaSTM

Effect of bulk updates on concurrent OLTP workload




4x Opteron 6376 @ 2.3GHz
64 cores / 64 threads
i7-3770 @ 3.4GHz
4 cores / 8 threads
TPC-C: 1 warehouse
TPC-C: 10 warehouses
Transactions per second
Threads
400K
200K
100K
50K
25K
800K
1 2 4 8 16 32 64 128256
Lazy Transactions
OCC (ScalaSTM)


4x Opteron 6376 @ 2.3GHz
64 cores / 64 threads
i7-3770 @ 3.4GHz
4 cores / 8 threads
New Order
Payment
Order status
Transactions per second
Threads
Lazy Transactions
OCC (ScalaSTM)

Conclusions

Conclusions

Lazy transactions execute contended transactions concurrently

- Non-blocking bulk updates
- Nearly ideal scaling on single-processor multi-core systems

Current concurrency control mechanisms cannot do this

- This is surprising, considering there has been 40 years of DB research

Requires core changes to database architecture

- Transactions are be submitted as programs

Future Work

- Scalability on NUMA systems: solving root contention
- Lazy relational schema transformations
- Lazy transactions for on-disk databases
- Language support for lazy transactions

Further Information

Read the paper

Lesley Wevers, Marieke Huisman and Maurice van Keulen. Lazy Evaluation for Concurrent OLTP and Bulk Transactions.

Implementation and benchmarks

https://github.com/utwente-fmt/lazy-transactions-IDEAS16