Lazy Evaluation for Concurrent OLTP and Bulk Transactions

Lesley Wevers Marieke Huisman Maurice van Keulen
University of Twente

IDEAS'16

March 4, 2014

Bitcoin bank Flexcoin shuts down after massive theft

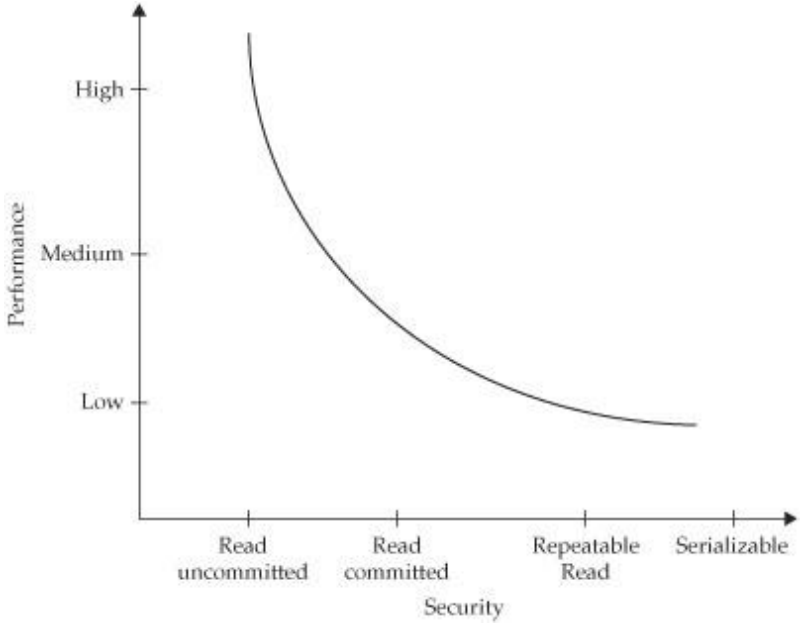
Some $600,000 worth of bitcoins were stolen from the site

March 5, 2014

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

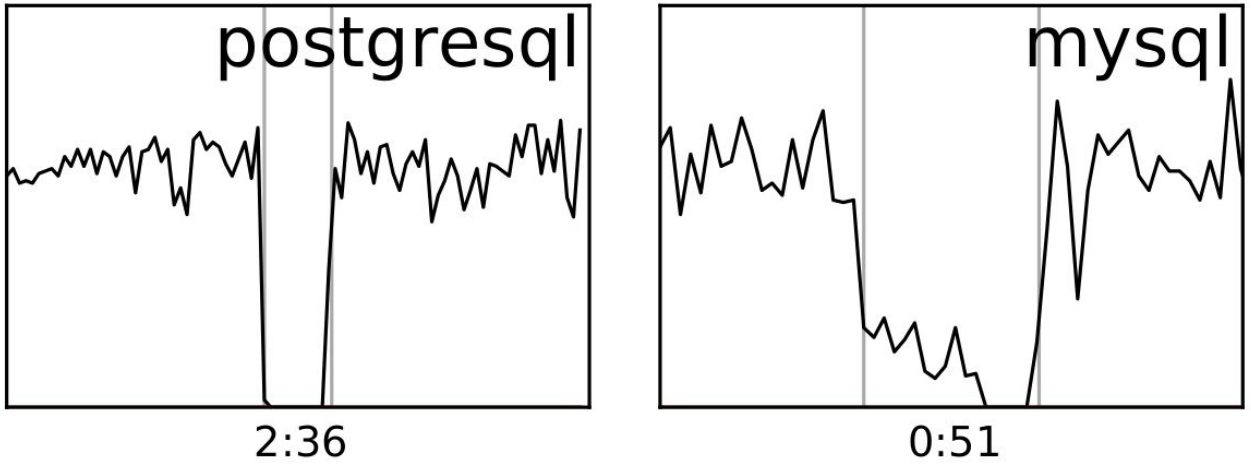
Firm can't cover losses: "All balances will temporarily be deducted by 12.3%."

Problem 1: Correctness vs Concurrency



Can we get more **concurrency** without sacrificing **correctness**?

Problem 2: Availability

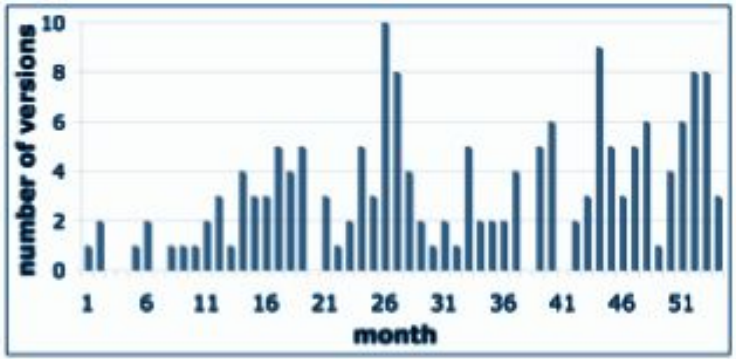


Bulk updates block all concurrent transactions.

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.

Problem 2: Availability

WikiMedia schema revisions:



- 90% required 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

Proposed Solution

Lazy Evaluation of Transactions

1. Splitting Transactions into Parts

```
if x >= 10:
        x -= 10
        y += 10
```

x =

```
if x >= 10
        then x - 10
        else x
```

y =

```
if x >= 10
        then y + 10
        else y
```

1. Splitting Transactions into Parts

```
if x >= 10:
        x -= 10
        y += 10
```

x = TODO

y = TODO

2. Atomically Queueing Parts

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

2. Atomically Queueing Parts

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

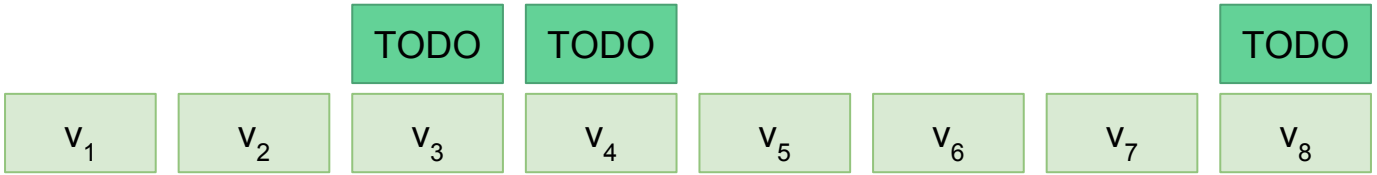
Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO



2. Atomically Queueing Parts

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

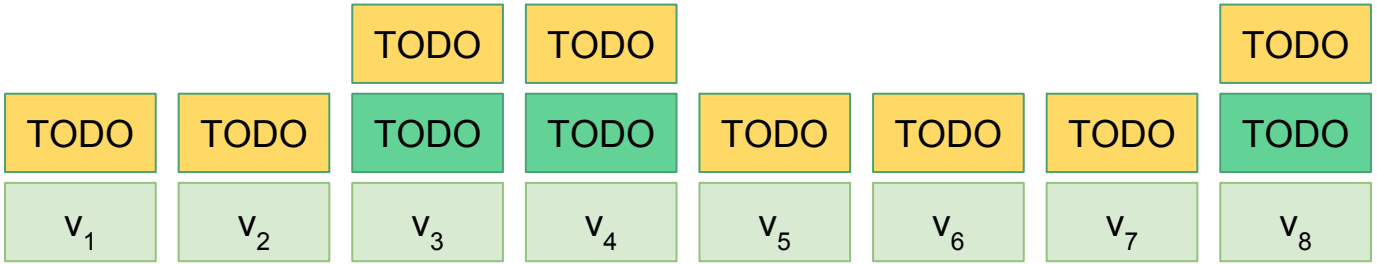
Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO



2. Atomically Queueing Parts

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

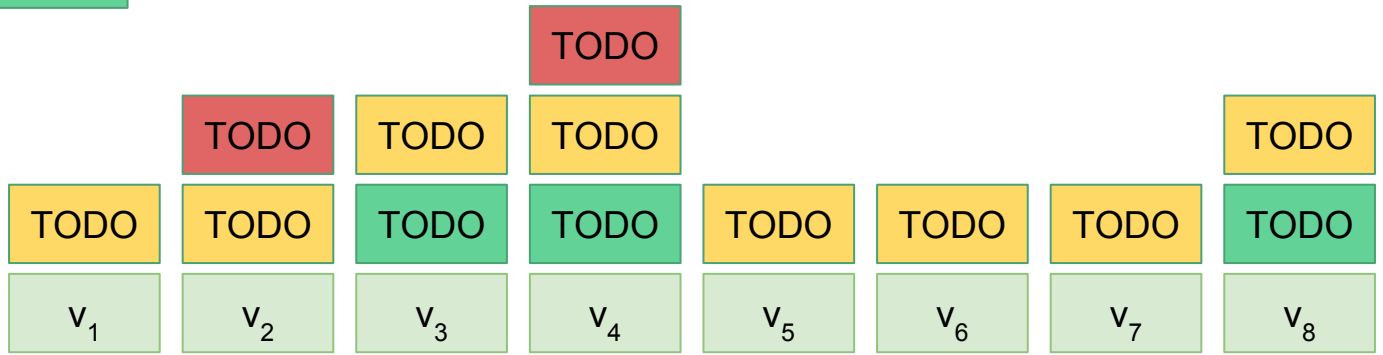
Transaction 2

$v_{1..8}$ TODO

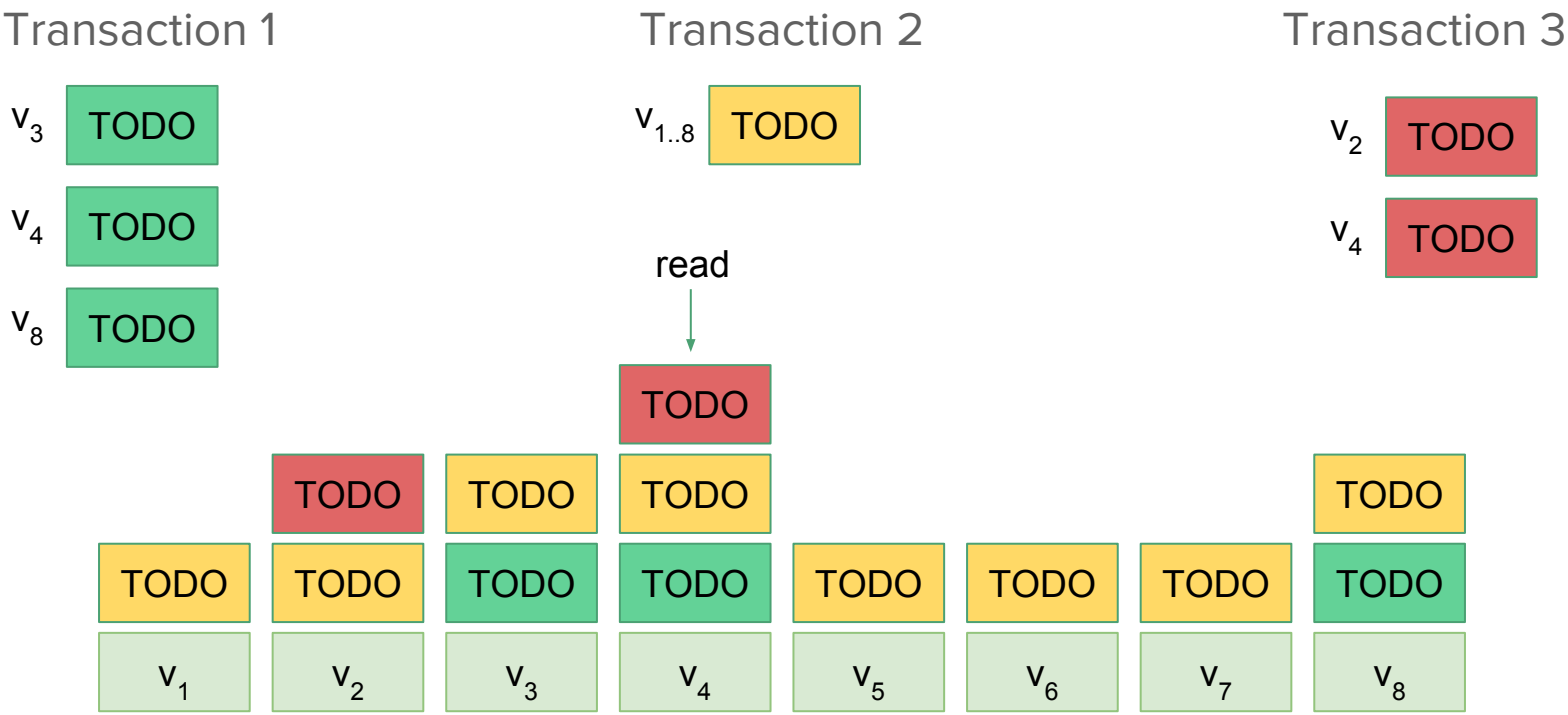
Transaction 3

v_2 TODO

v_4 TODO



3. Evaluate Parts on Demand



3. Evaluate Parts on Demand

Transaction 1

v_3 TODO

v_4 TODO

v_8 TODO

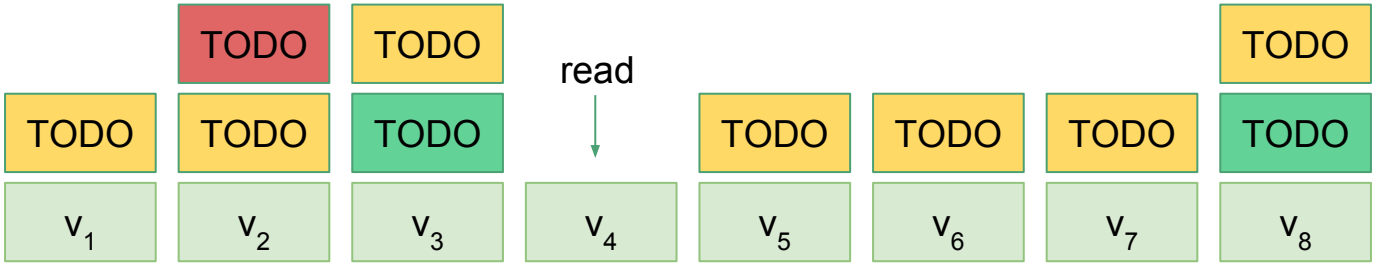
Transaction 2

$v_{1..8}$ TODO

Transaction 3

v_2 TODO

v_4 TODO



Contributions

Lazy evaluation of transactions

- More concurrency, without sacrificing correctness
- Non-blocking bulk operations

Outline

1. Atomically queuing (many) parts without blocking
2. Transaction execution phases
3. Evaluation strategies
4. Experimental evaluation

Atomically Queueing Parts without Blocking

Transaction 1

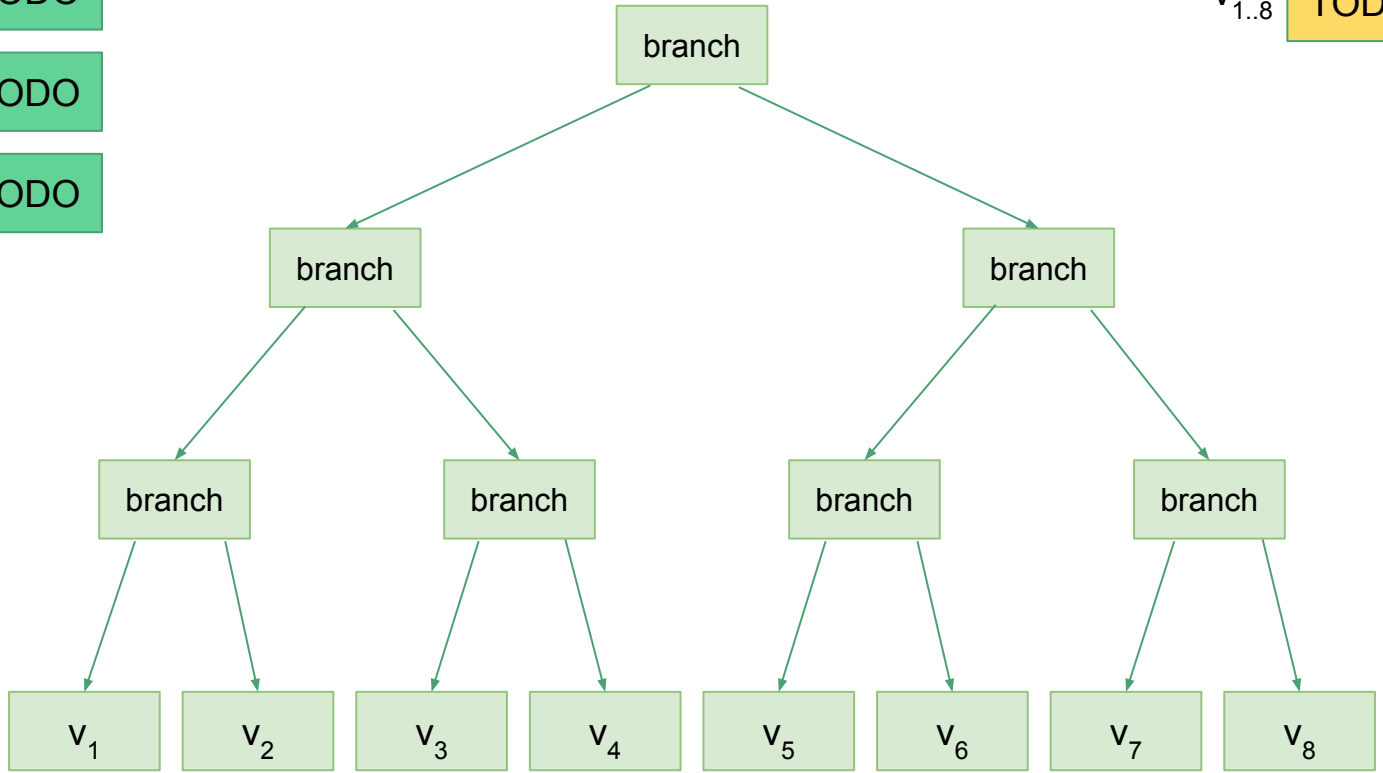
v_3 TODO

v_4 TODO

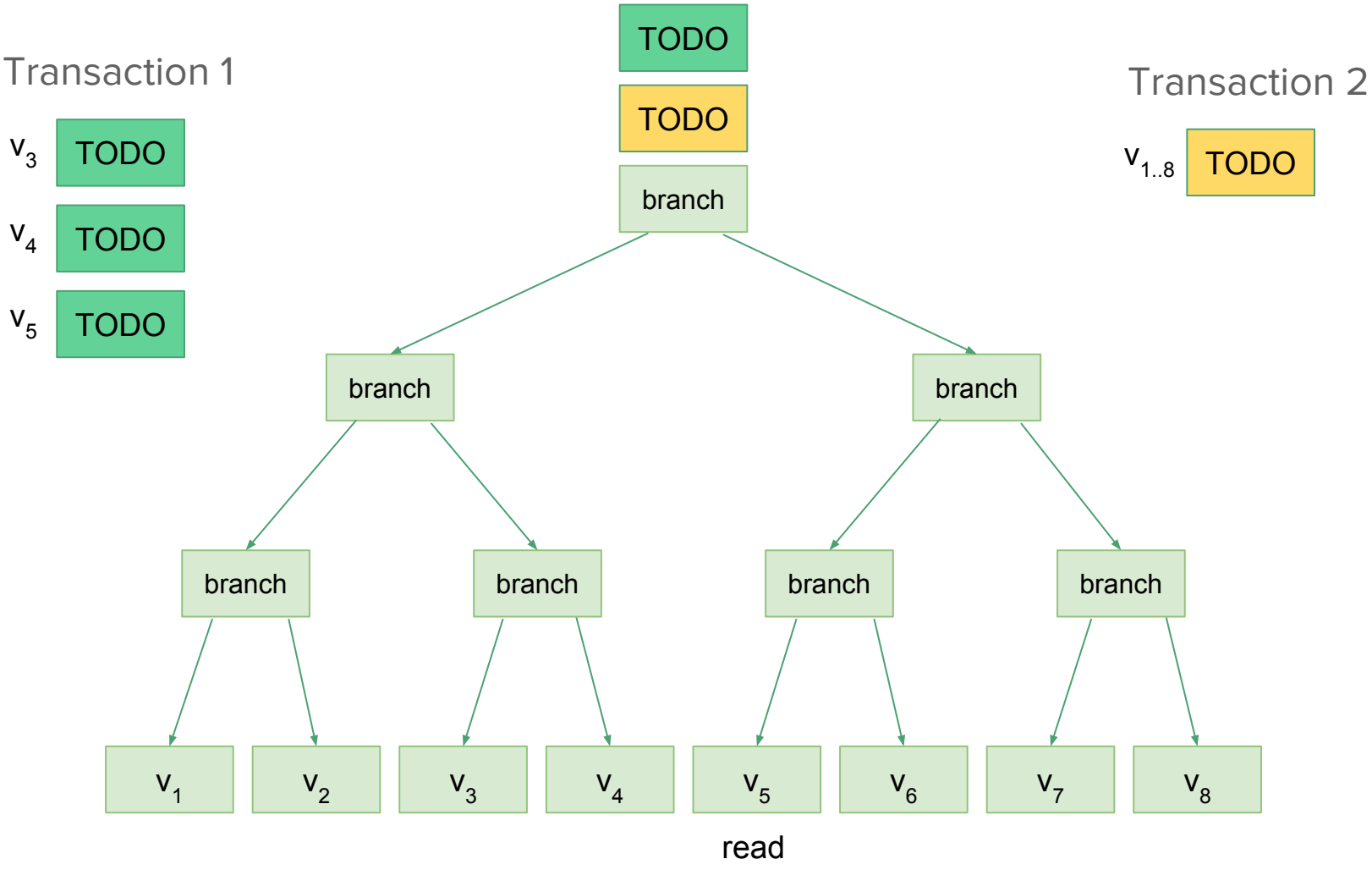
v_5 TODO

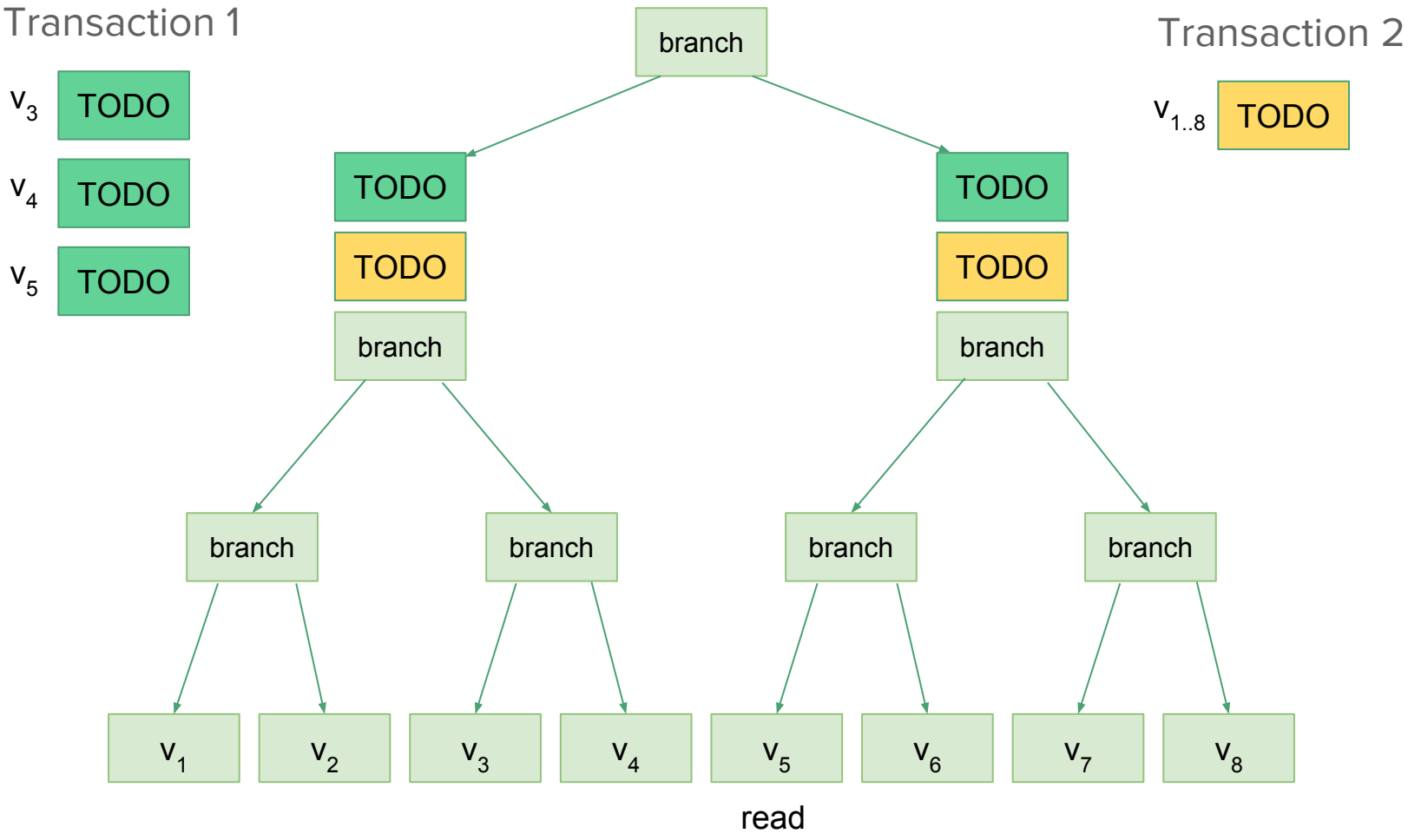
Transaction 2

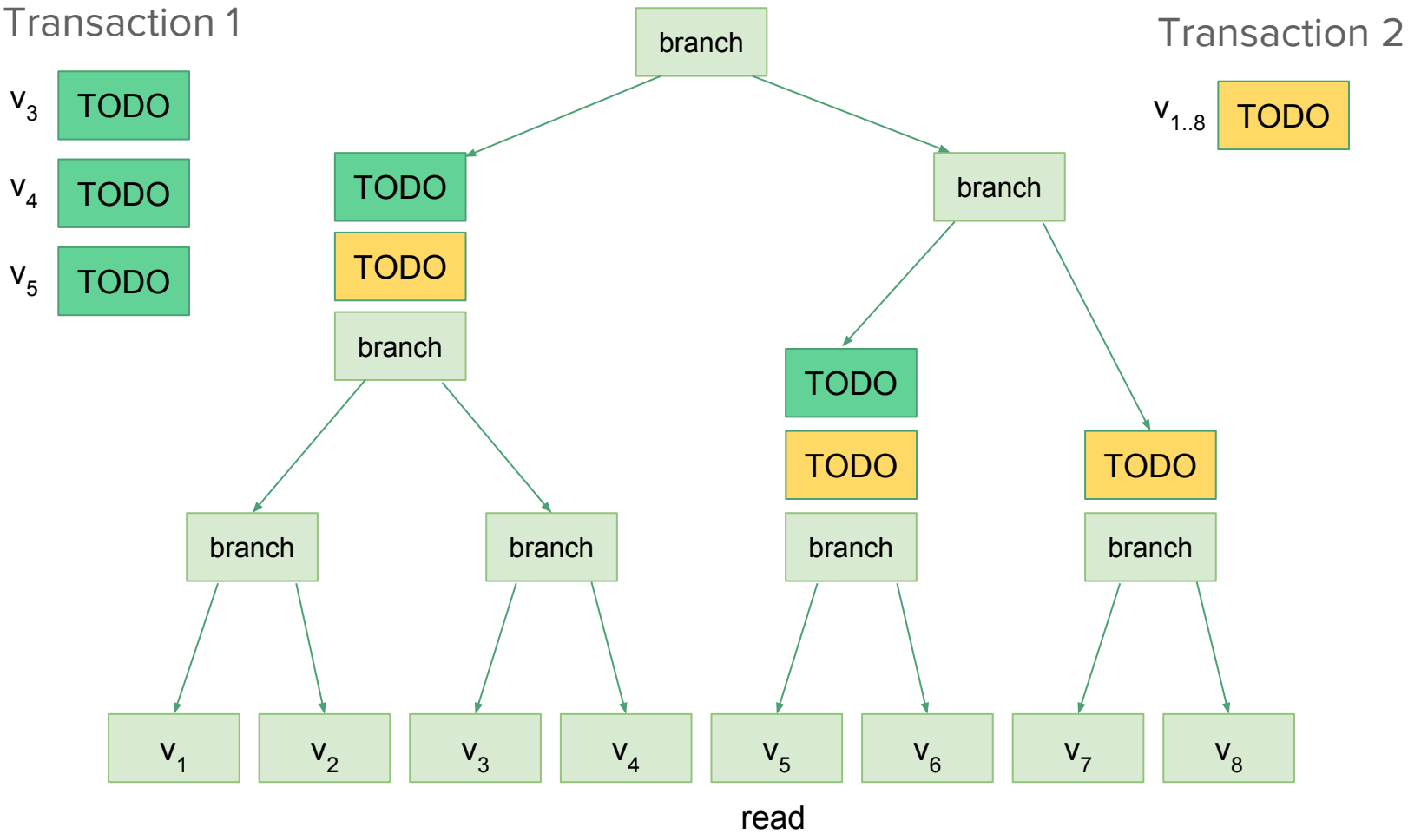
$v_{1..8}$ TODO

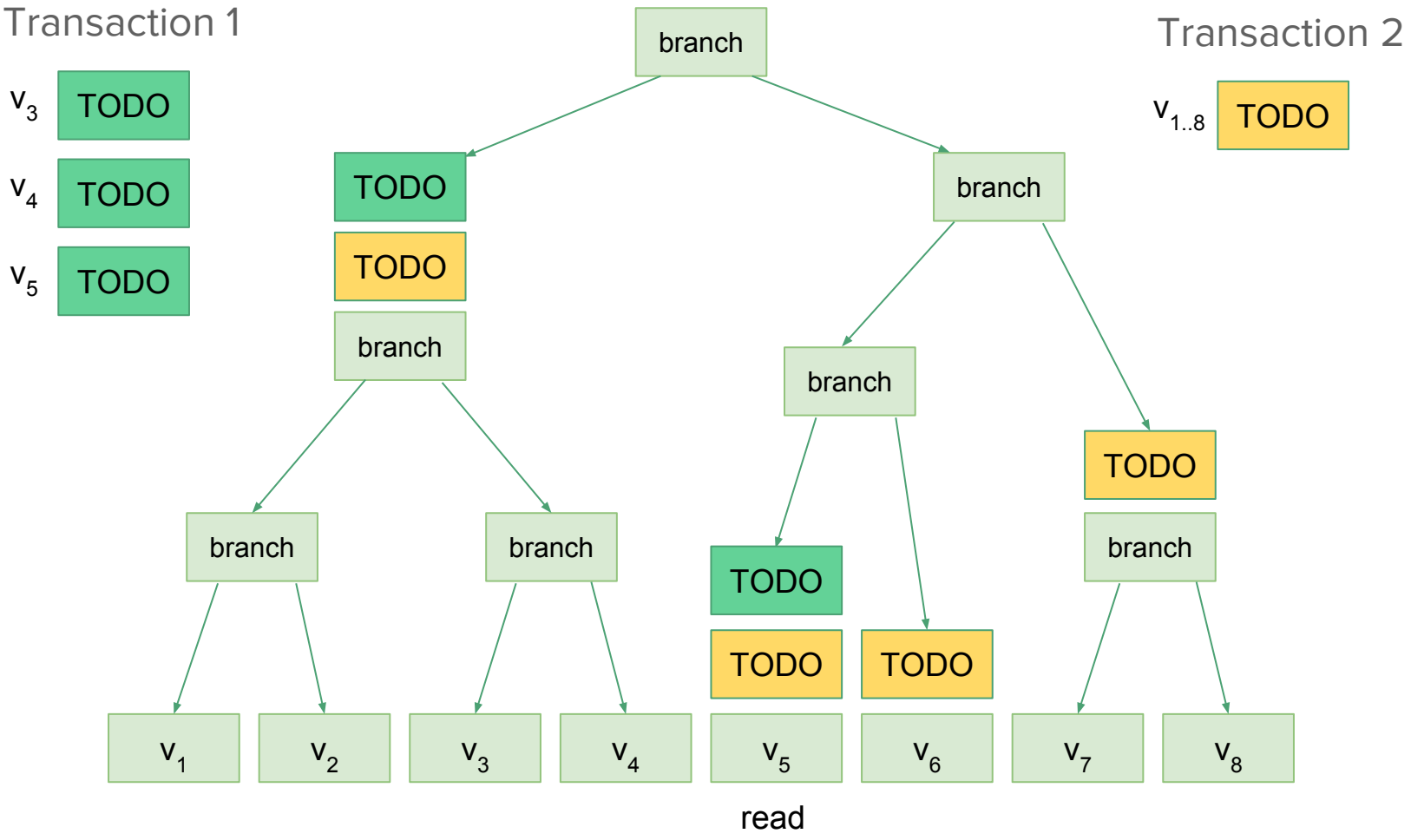


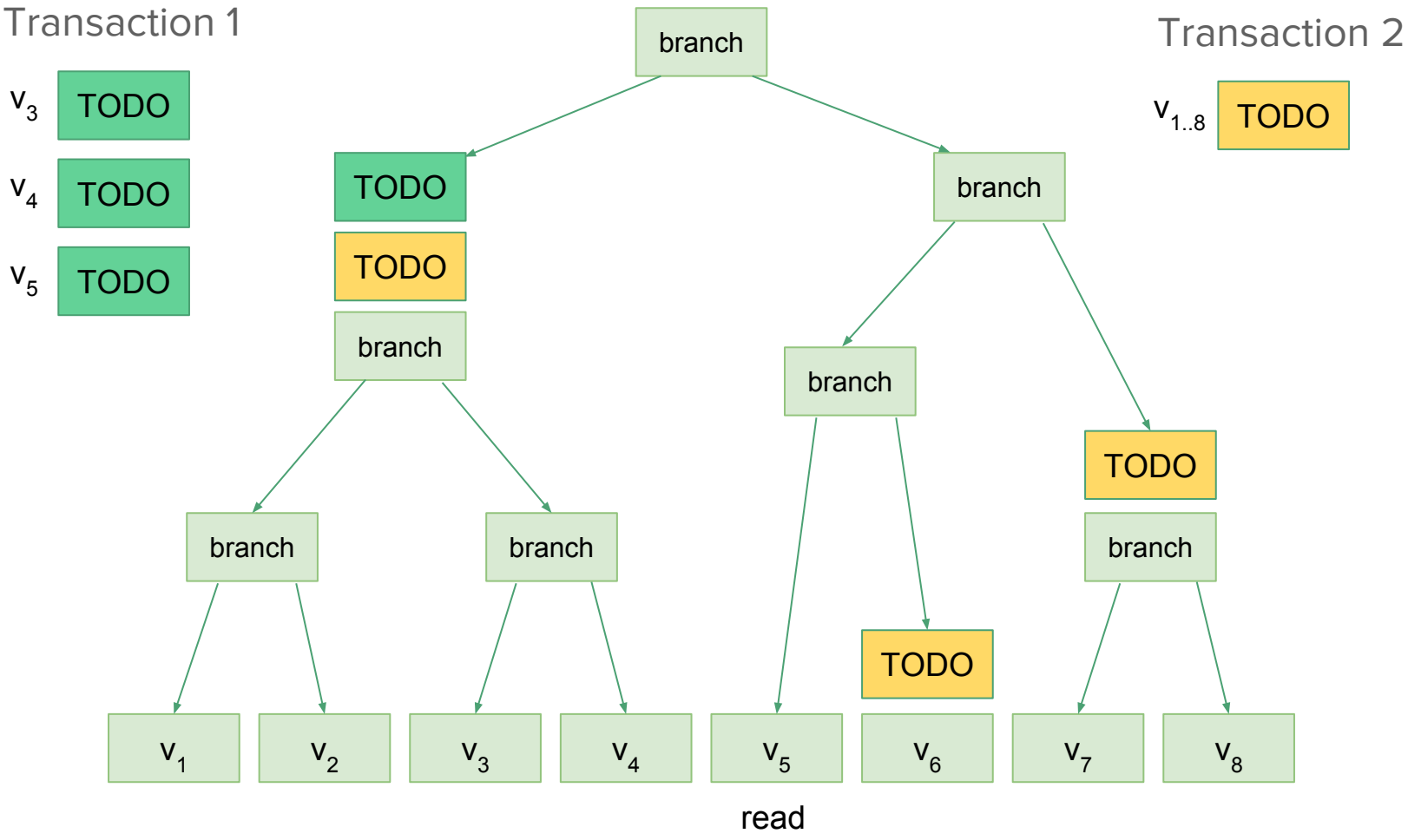
read


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

Transaction Execution Phases

Transaction Execution Phases

1. **Preparation Phase**
    - Determine where parts should be written
    - Inconsistent **optimistic reads**

Transaction Execution Phases

1. **Preparation Phase**
   - Determine where parts should be written
   - Inconsistent **optimistic reads**
2. **Commit Phase**
   - Atomically queue all parts
   - **Blocking reads** on the current state

Transaction Execution Phases

1. **Preparation Phase**
   - Determine where parts should be written
   - Inconsistent **optimistic reads**
2. **Commit Phase**
   - Atomically queue all parts
   - **Blocking reads** on the current state
3. **Lazy Phase**
   - Work performed inside parts
   - **Lazy reads** on a snapshot
   - Validation of optimistic reads

Transaction Execution Phases

1. **Preparation Phase**
   - Determine where parts should be written
   - Inconsistent **optimistic reads**
2. **Commit Phase**
   - Atomically queue all parts
   - **Blocking reads** on the current state
3. **Lazy Phase**
   - Work performed inside parts
   - **Lazy reads** on a snapshot
   - Validation of optimistic reads
4. **Result Phase**
   - **Reads on a snapshots obtained in the commit phase**

Evaluation Strategies

Evaluation Strategies

Trade-offs between **latency**, **throughput** and **memory consumption**.

When do we evaluate update functions?

- Delay as much as possible
- Evaluate immediately after committing a transaction

Granularity of evaluation

- Evaluate no more than strictly necessary
- Evaluate multiple operations together

Experimental Results

Experimental Setup

We have implemented a prototype in Scala

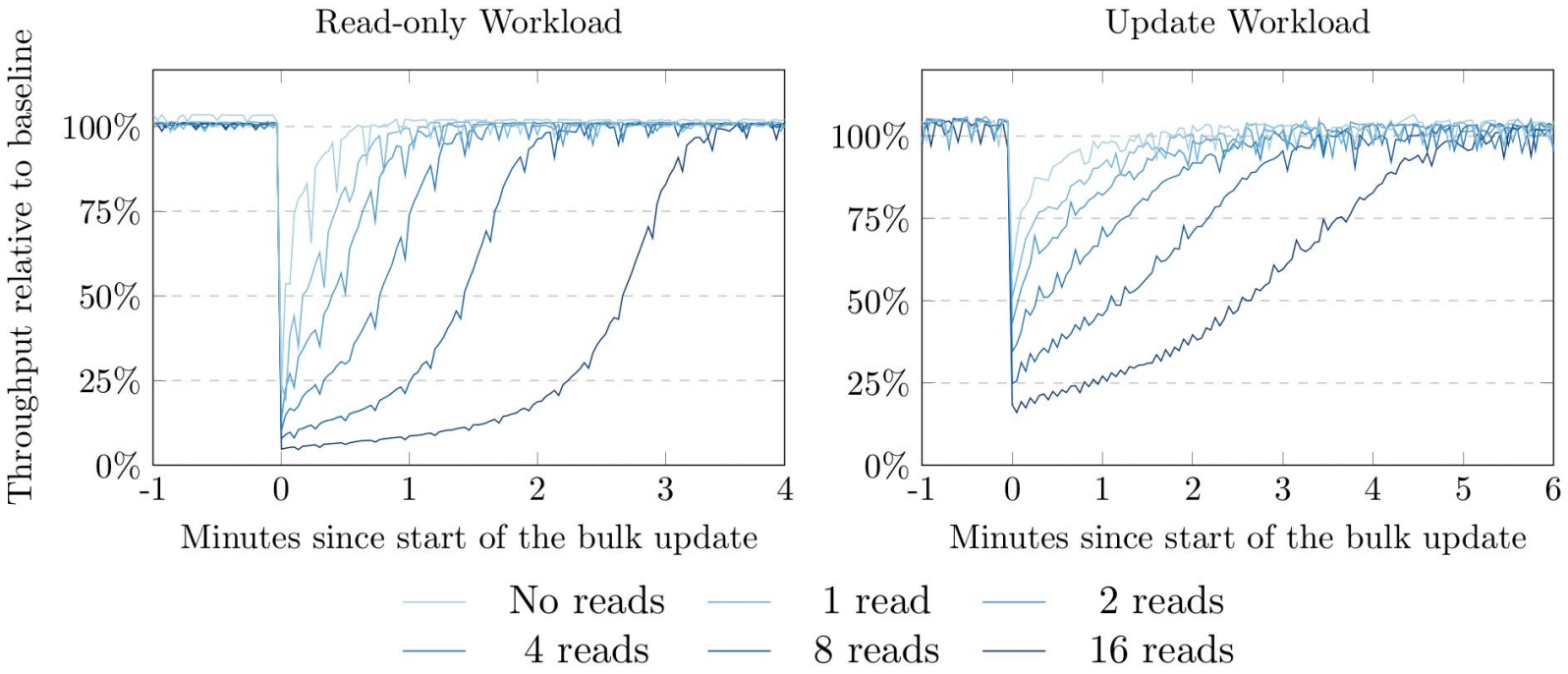
We show that lazy transactions can perform

- Bulk operations without blocking a concurrent OLTP workload
- More transactions concurrently in OLTP workloads

We compare against ScalaSTM

- Software transactional memory for Scala
- Optimistic concurrency control

Effect of bulk updates on a concurrent OLTP workload



Benchmarks in the Paper

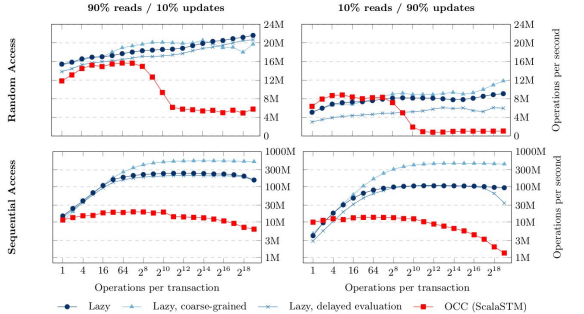


Figure 4: The effect of the size of transactions on throughput, where each transaction is read-only or update-only. On the left, transactions are read dominated, while on the right they are update dominated. Note the logarithmic axis on the sequential access plots.

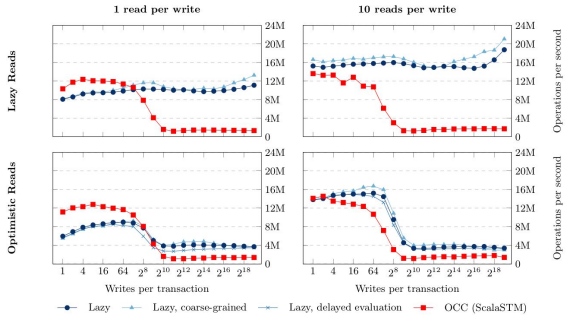
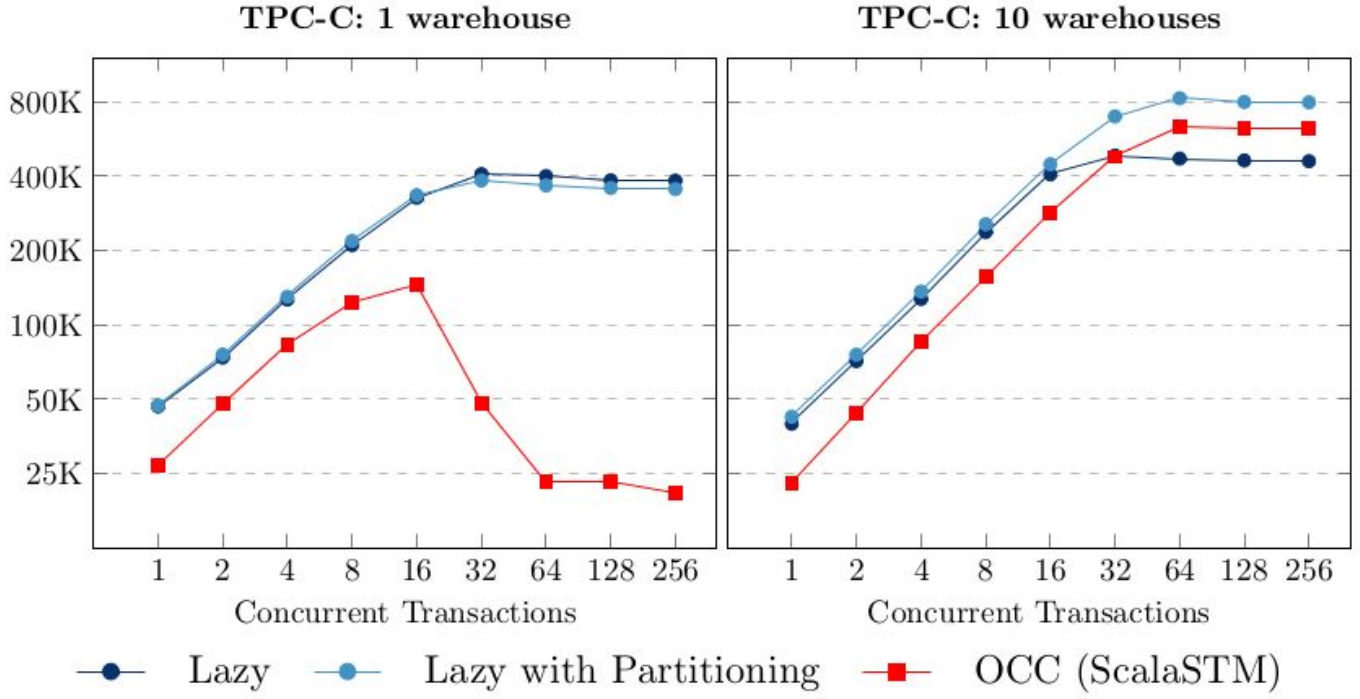


Figure 5: The effect of the size of transactions on throughput, where transactions perform a mixed read / write workload using lazy or optimistic reads.

TPC-C



4x Opteron 6376
64 cores / 64 threads

Discussion & Conclusions

Discussion

Lazy evaluation:

- Re-orders operations within a transaction based on demand from readers
- Allows using results from partially executed transactions

Concurrency is fundamentally limited by data dependencies:

- Lazy evaluation comes closer to this limit than 2PL and OCC
- Often better than 2PL and OCC, in the worst case similar

Laziness require transactions to be submitted to the DBMS as programs

Future Work

- Automatic splitting of transaction programs
- Alternative protocols to queue operations atomically
- Evaluation strategies
- Lazy relational schema transformations

Conclusion

Lazy evaluation of transactions allows:

- More concurrent transactions, without loss of correctness
- Better use of multi-core systems
- Non-blocking bulk updates

Implementation and benchmarks:

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

For more information, read our paper:

Lazy Evaluation for Concurrent OLTP and Bulk Transactions