Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases

Evan P. C. Jones

Databases: Critical infrastructure

- E-commerce
- Banking
- Airline reservations
- Web applications

OnLine Transaction Processing

Interactive user requests:

- Short transactions
- Use indexes to access few records
- Queries known in advance
- Reads and writes

Industry trends changing OLTP

- Separate OLTP and data warehouse systems
- Memory is cheap
- Clusters are cheap

Dtxn

Framework for building fault-tolerant distributed databases, specialized for memory-resident OLTP workloads

Novel features

Reusable infrastructure for OLTP databases

Speculative concurrency control

Live migration using a cache-based approach

Novel features

Reusable infrastructure for OLTP databases

Speculative concurrency control

Live migration using a cache-based approach

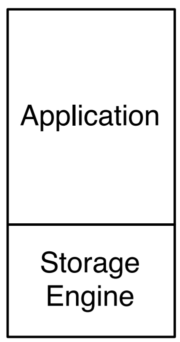
Example: NewBank storage

Use Dtxn to build a customized key/value store with transactions

Operations:

- get(account)
- put(account, balance)
- transfer(source, destination)

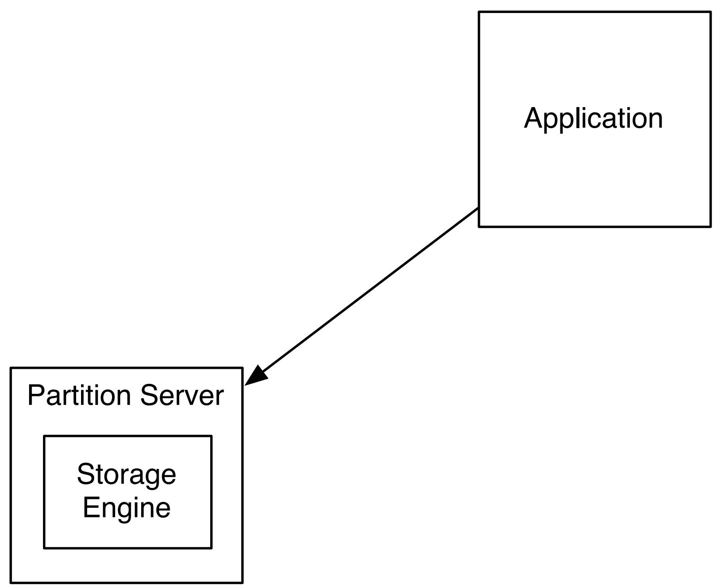
V1: In-memory storage library



Storage Engine implements:

- Operations (e.g. get, put, transfer)
- Ability to undo a set of operations (rollback)

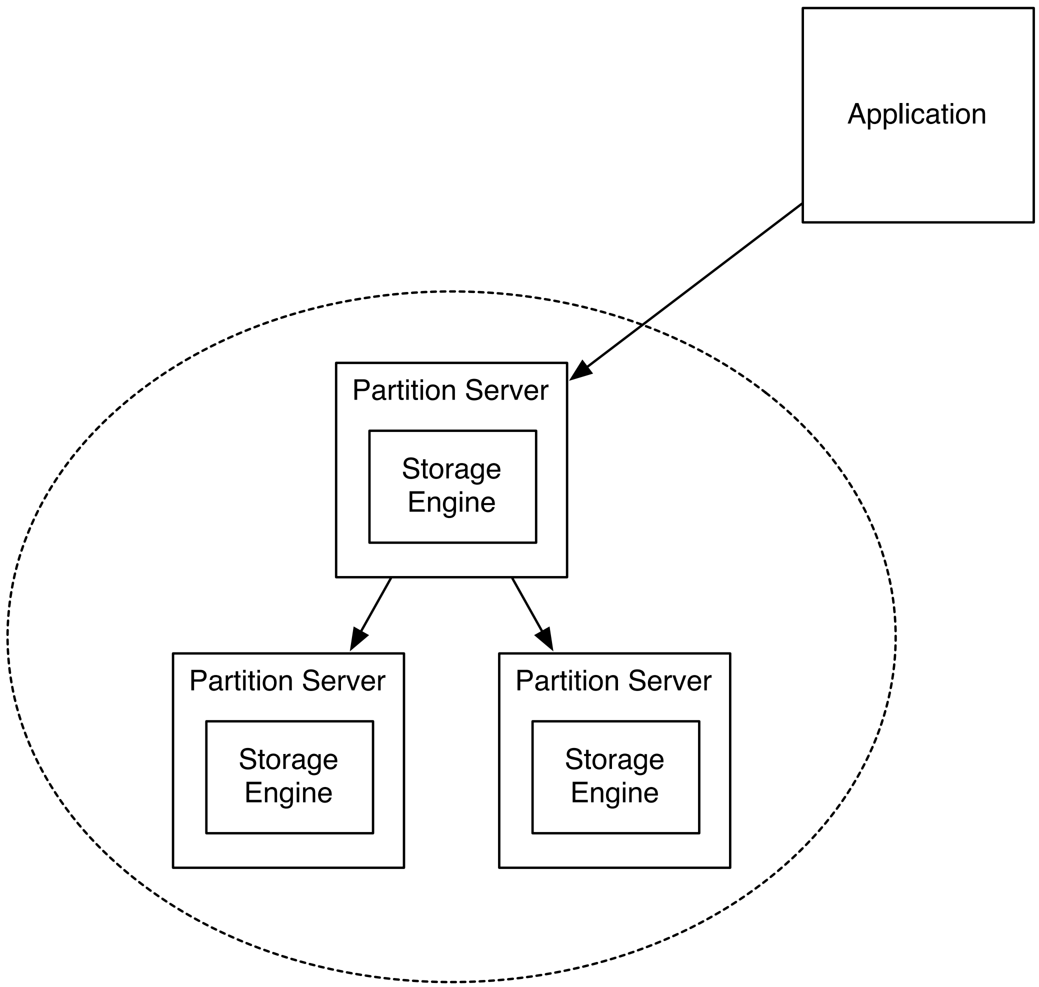
V2: Single server



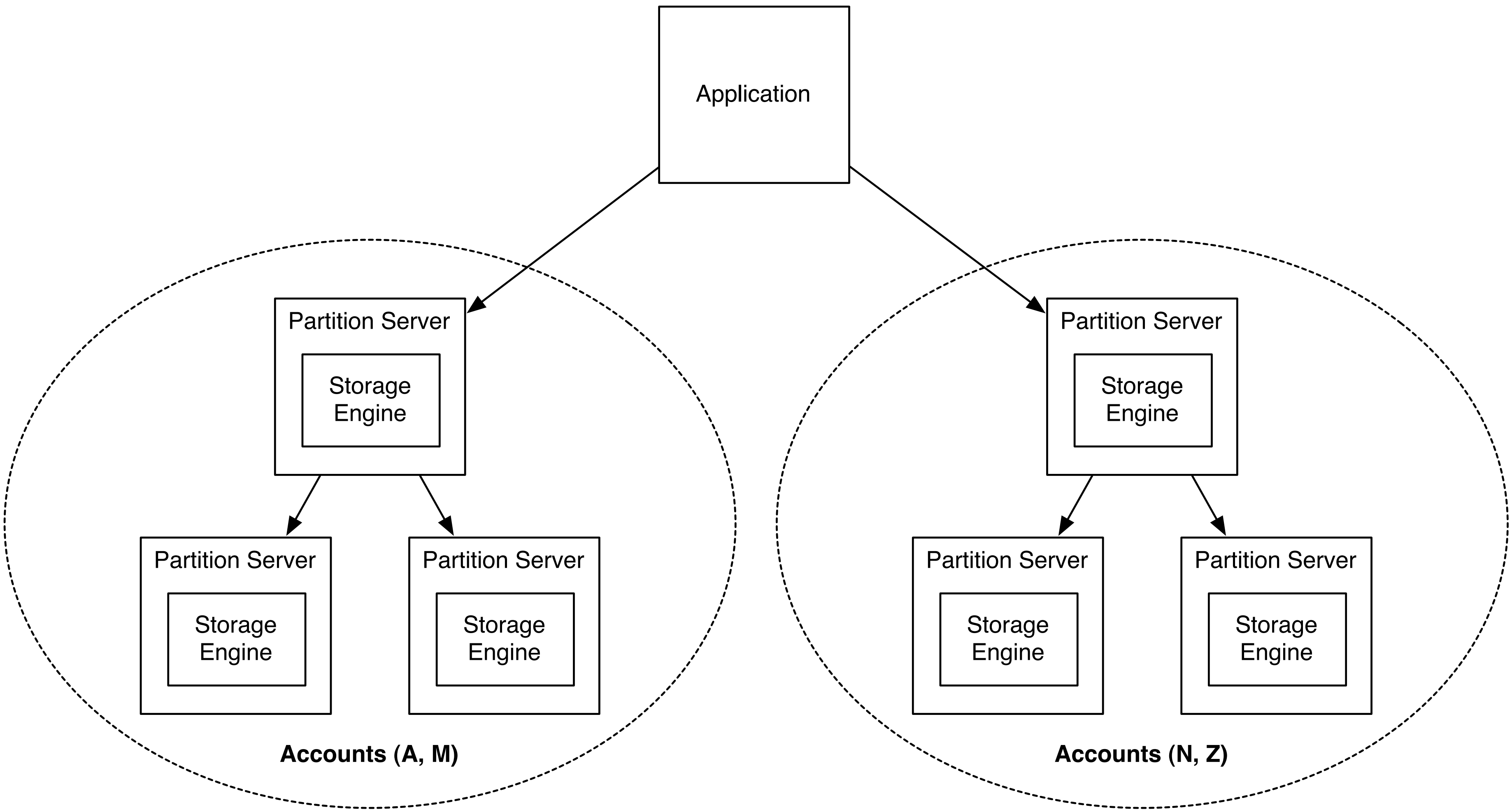
Partition server provides:

- Concurrency control
- Durability

V3: Replicated server



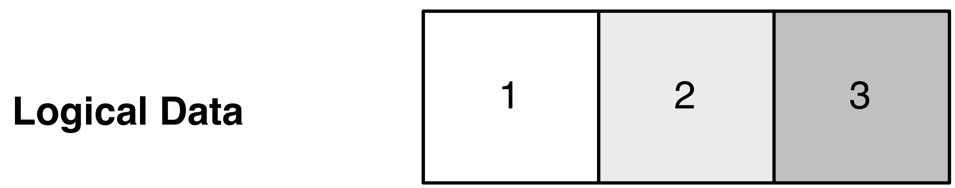
Dtxn distributed system



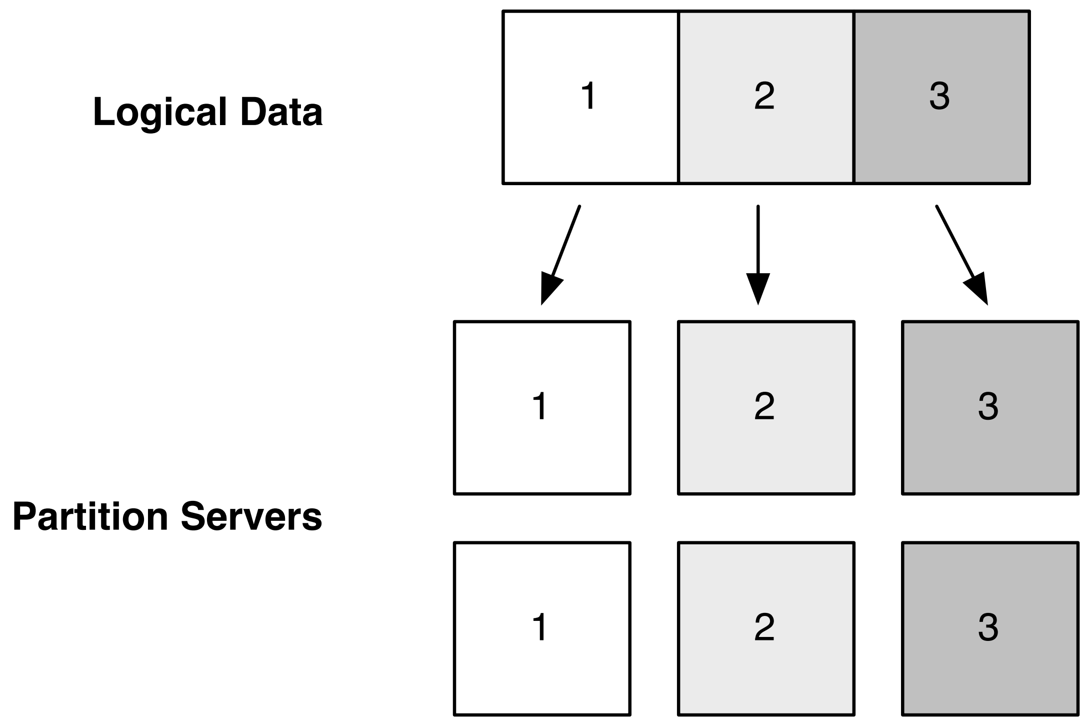
Distributing data: partitions

Logical Data

Distributing data: partitions



Distributing data: partitions



Partition: Logical container of data

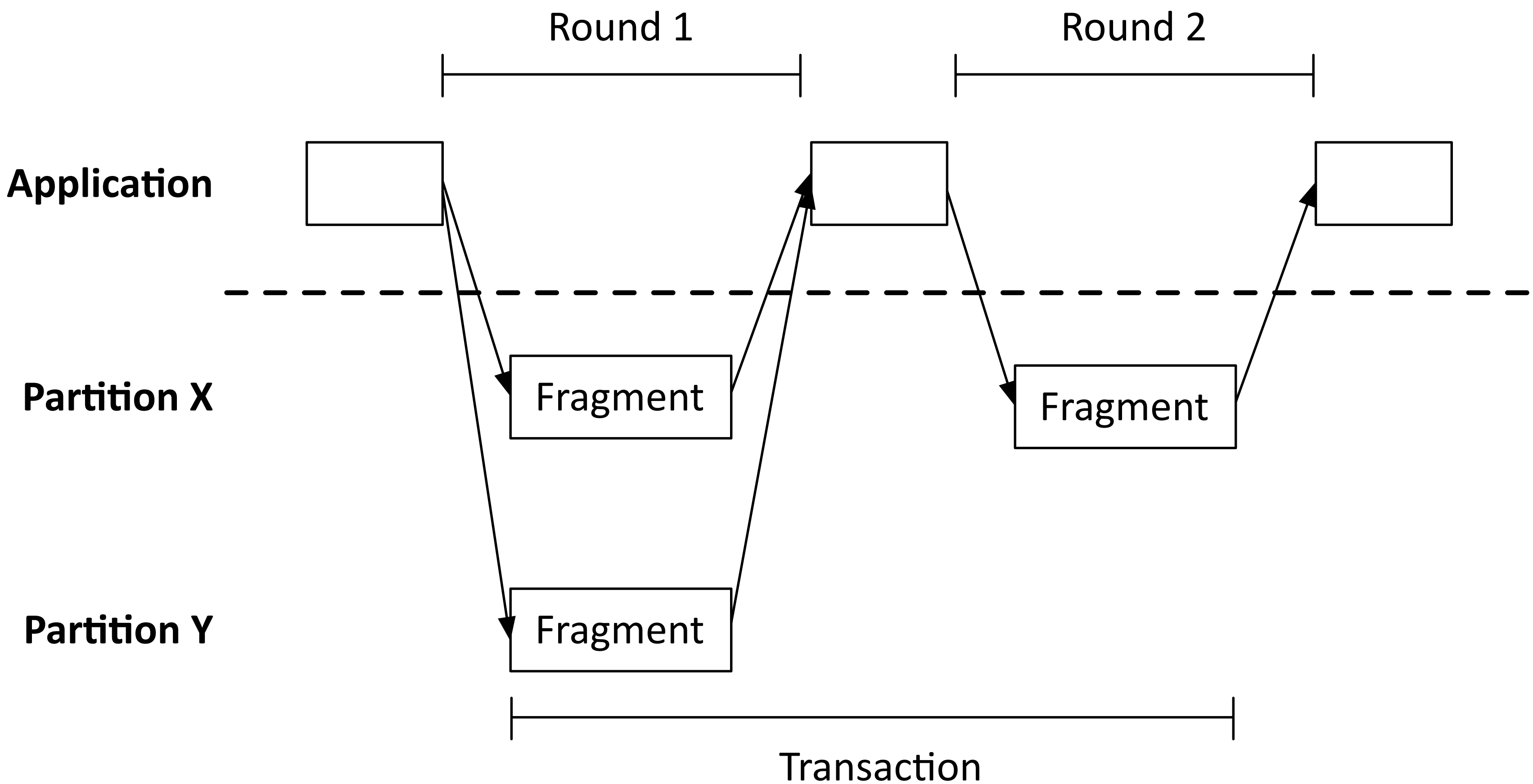
Partition Server: Process storing a partition

Transaction API

Applications issue transactions to Dtxn:
Transaction coordinators

- Dtxn handles distribution and fault-tolerance
- Two-phase commit for distributed transactions
- Send a *round* of work to Dtxn at a time
- A round contains *fragments*, one fragment per partition

Transactions, fragments, rounds



Applications

H-Store design: Specialized OLTP database

- Main memory
- Single threaded execution
- Optimized for stored procedures

Relational Cloud: Distributed SQL database

- MySQL storage engine
- Traditional SQL interface

Novel features

Reusable infrastructure for OLTP databases

Speculative concurrency control

Live migration using a cache-based approach

Serializable consistency

Transactions do not see effects of other concurrent transactions

Makes life easier for application developers

Main memory overhead

Traditional lock-based concurrency control is expensive for main memory databases:

30-40% of CPU time

Problem: Acquiring/releasing a lock costs the same as accessing the data

Source: Harizopoulos, Abadi, Madden and Stonebraker, "OLTP Under the Looking Glass", SIGMOD 2008

Speculative Concurrency Control

Eliminate overhead of fine-grained locks

Eliminate undo logs

Up to 2X faster than locking for appropriate workloads

Why support concurrency?

Use idle resources:

disk stalls	main memory
user stalls	stored procedures

Physical resources:

multiple CPUs	partition per core
multiple disks	

Long running txns: don't do them

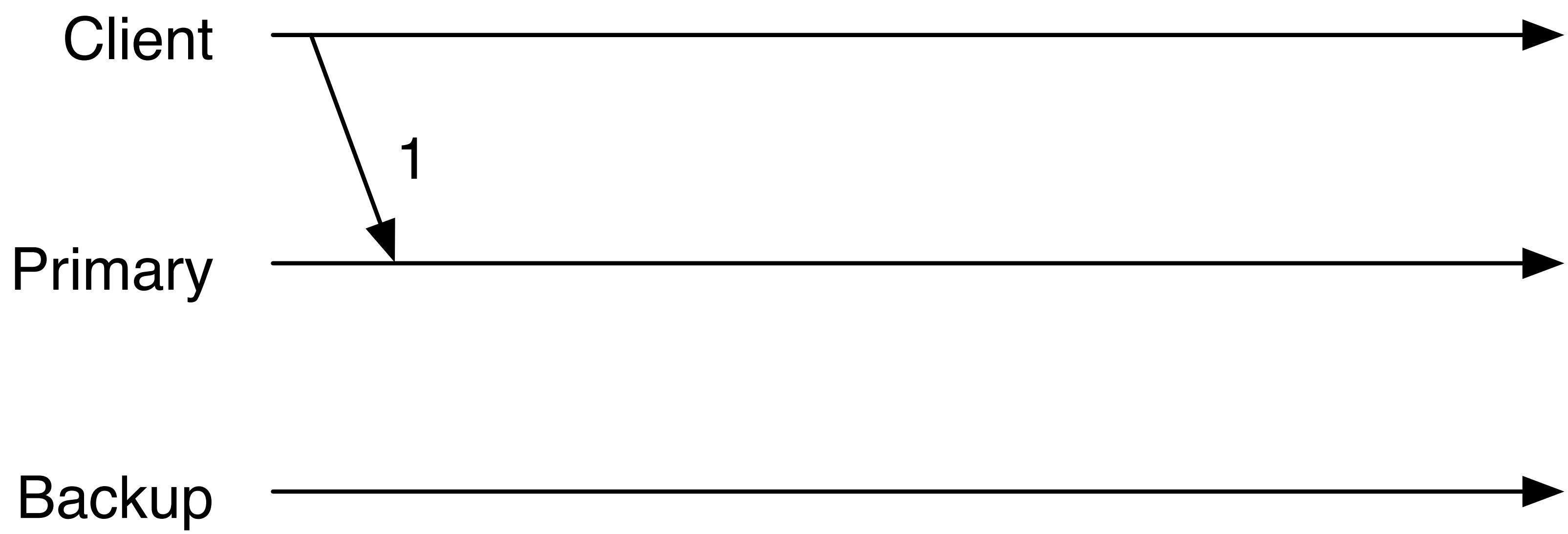
Alternative: one txn at a time

Execute transaction from beginning to end, with no locks, undo logs, or stalls

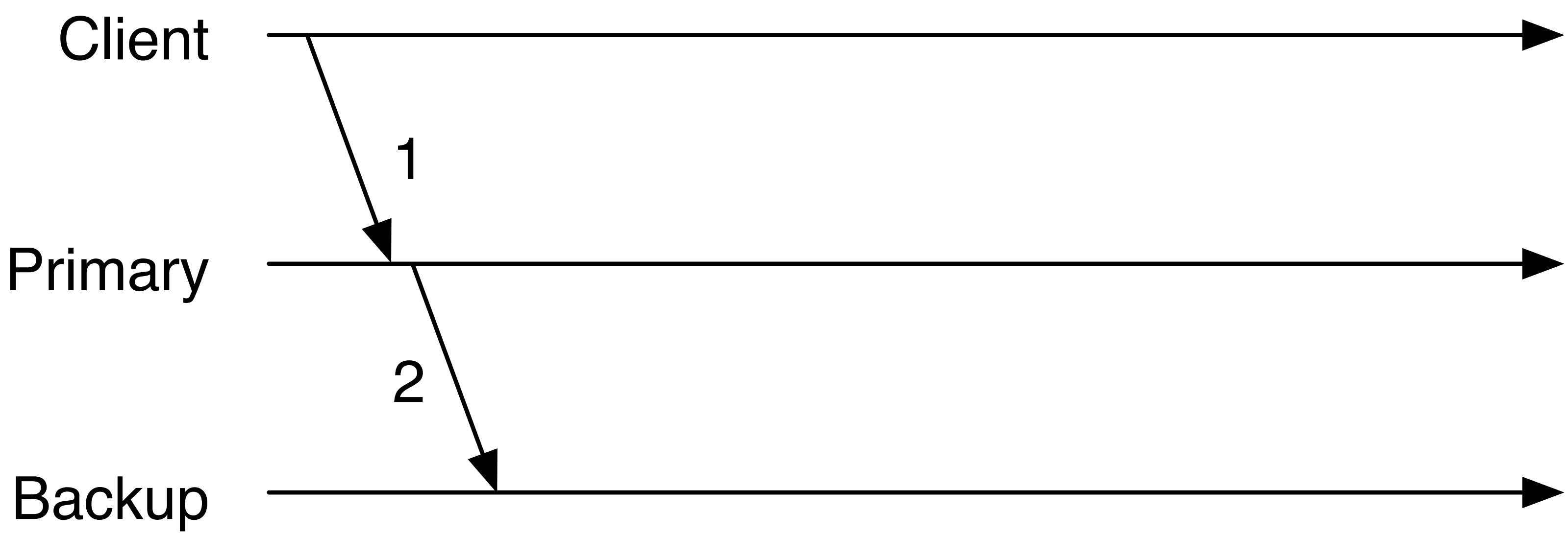
Example:

- Single round, single partition transactions (e.g. balance transfer on same partition)

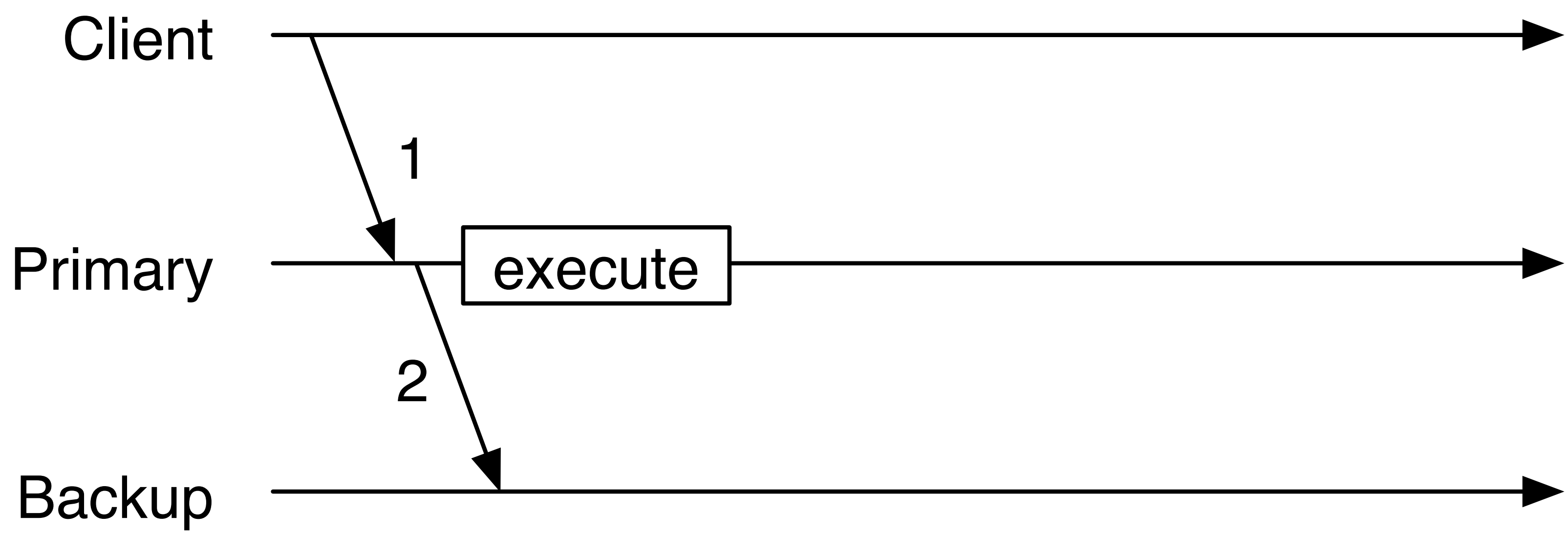
Single partition transaction



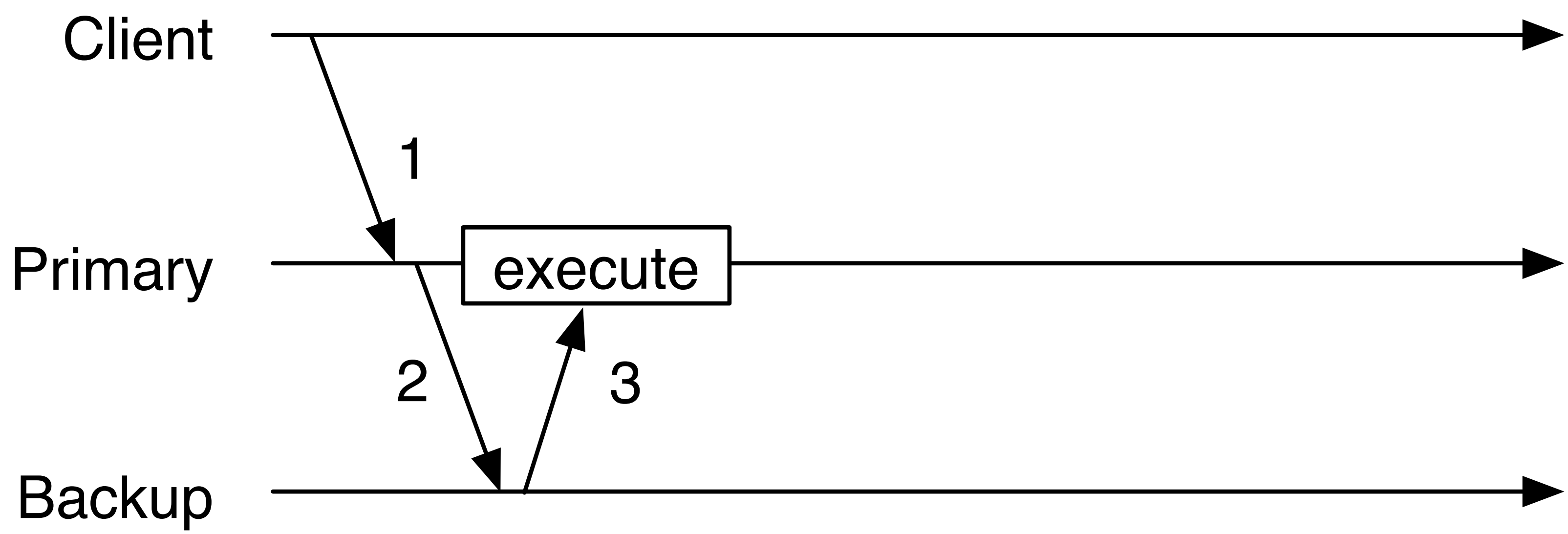
Single partition transaction



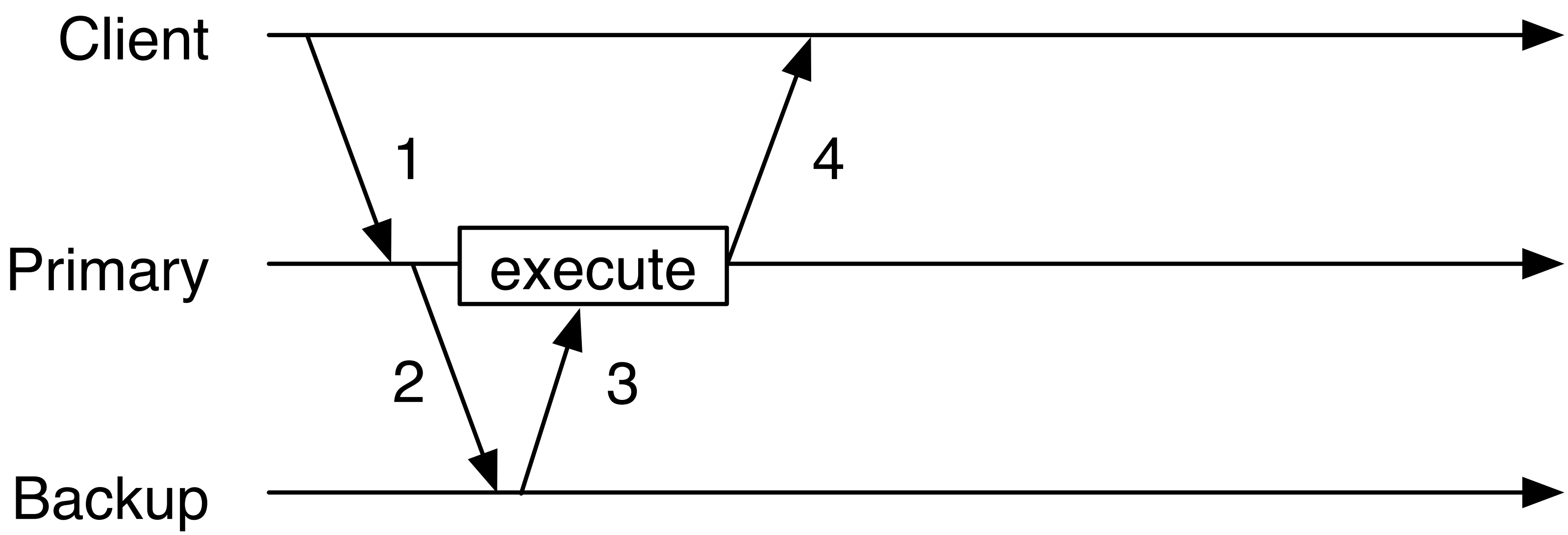
Single partition transaction



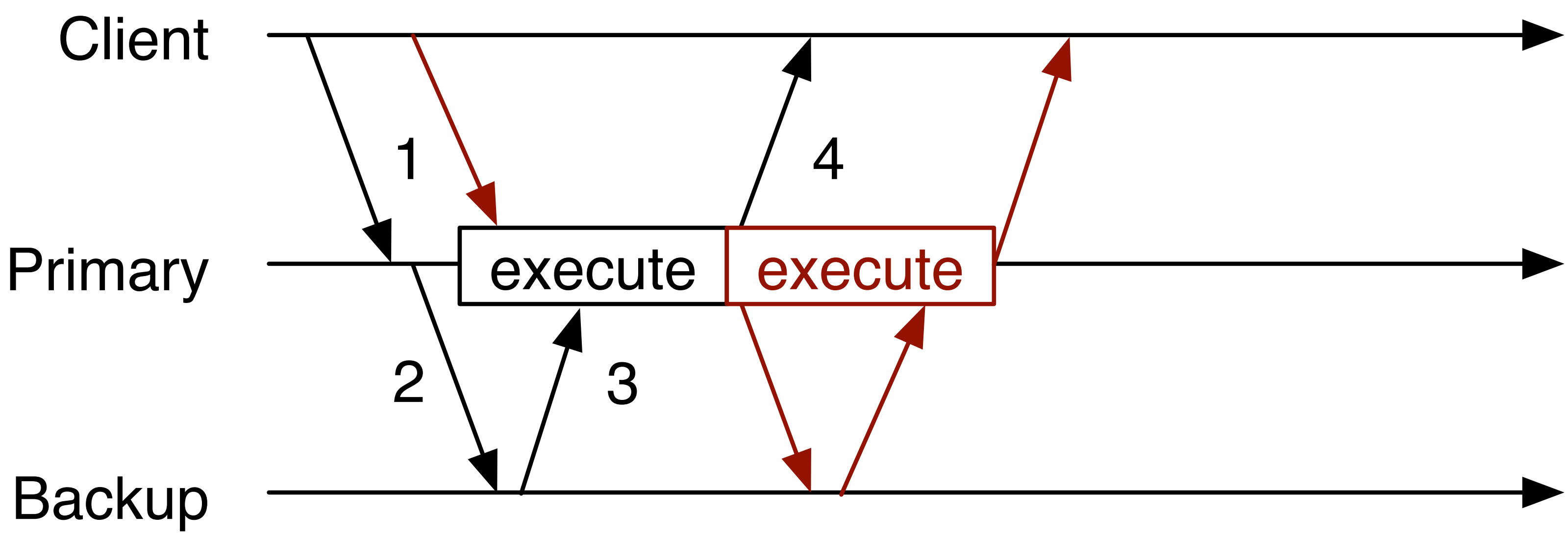
Single partition transaction



Single partition transaction



Single partition transaction



What about distributed txns?

Many OLTP applications are *mostly* partitionable

e.g. TPC-C: 11% multi-partition transactions

Simple example:

- One round, multiple partitions
- e.g. transfer $\$x$ between accounts

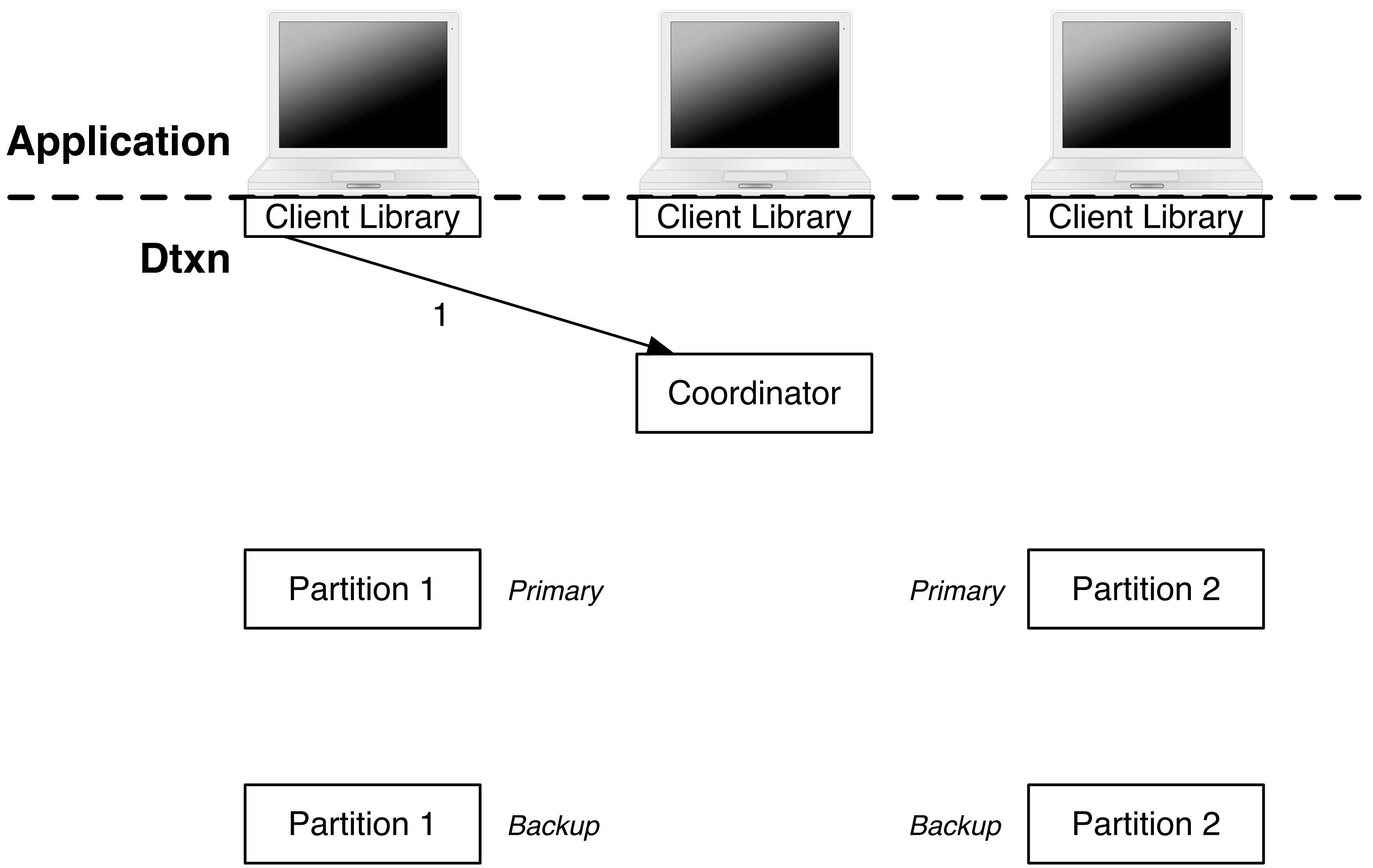
Simple solution: block

Wait until the distributed transaction completes

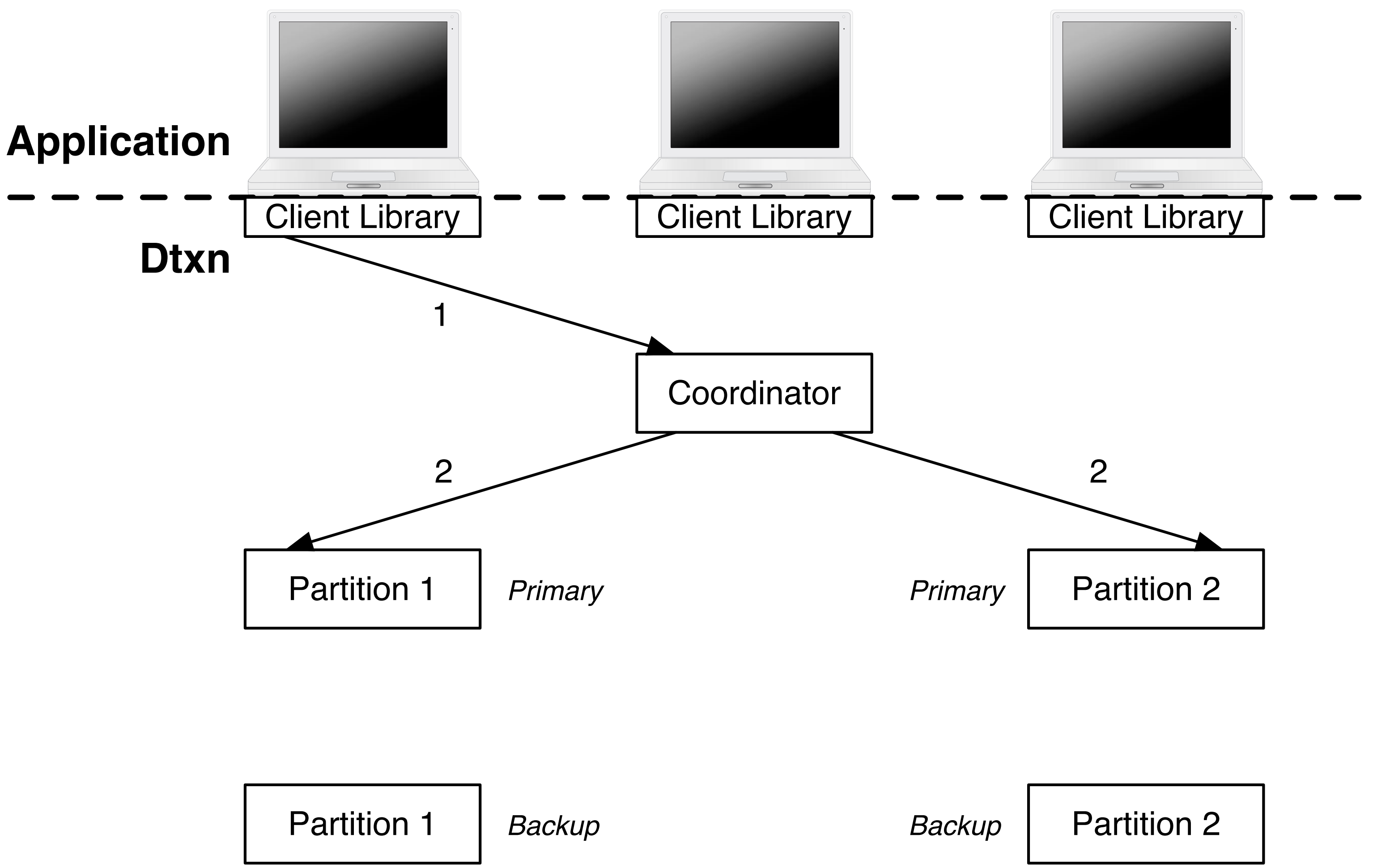
Need to order distributed transactions:

- Send all through a central coordinator

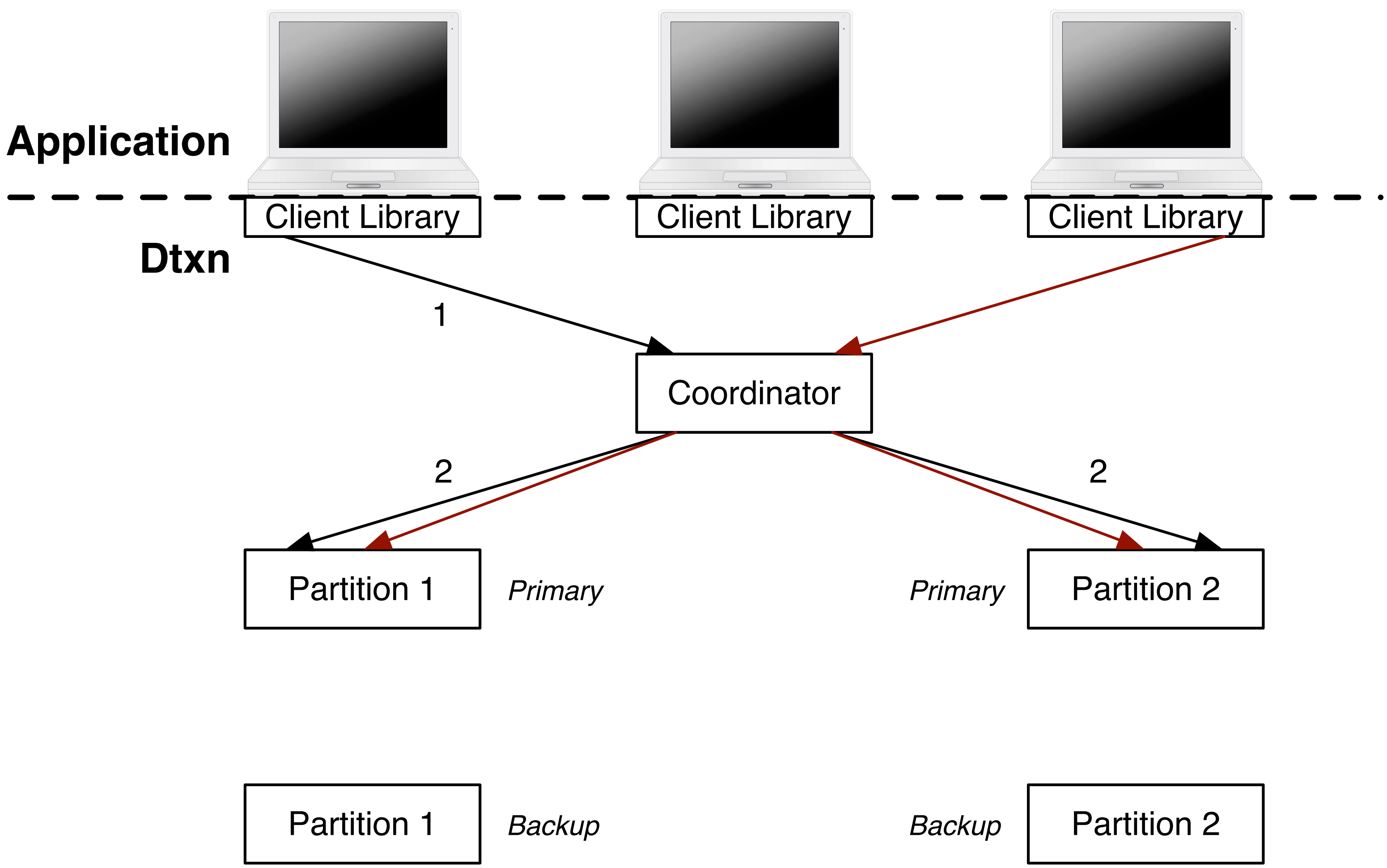
Blocking multi-partition



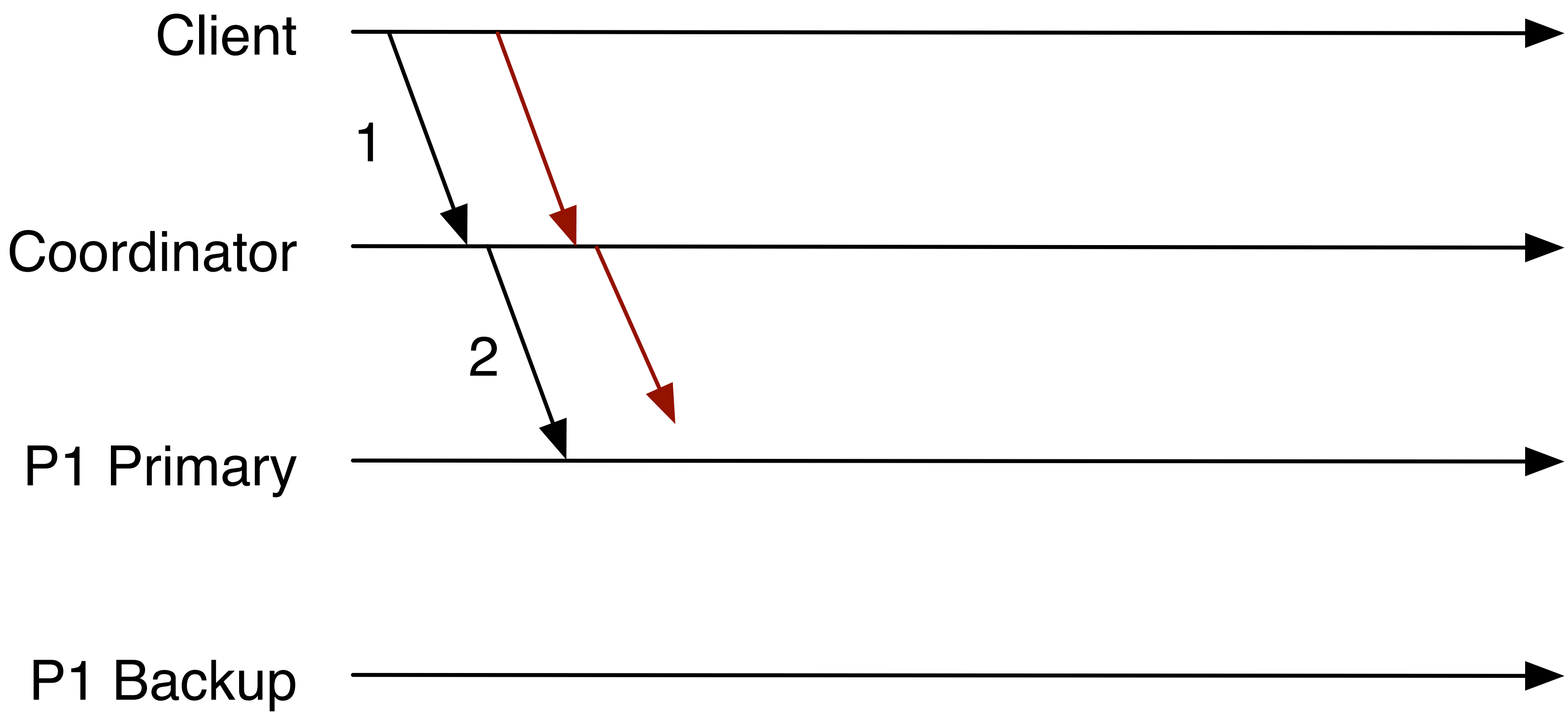
Blocking multi-partition



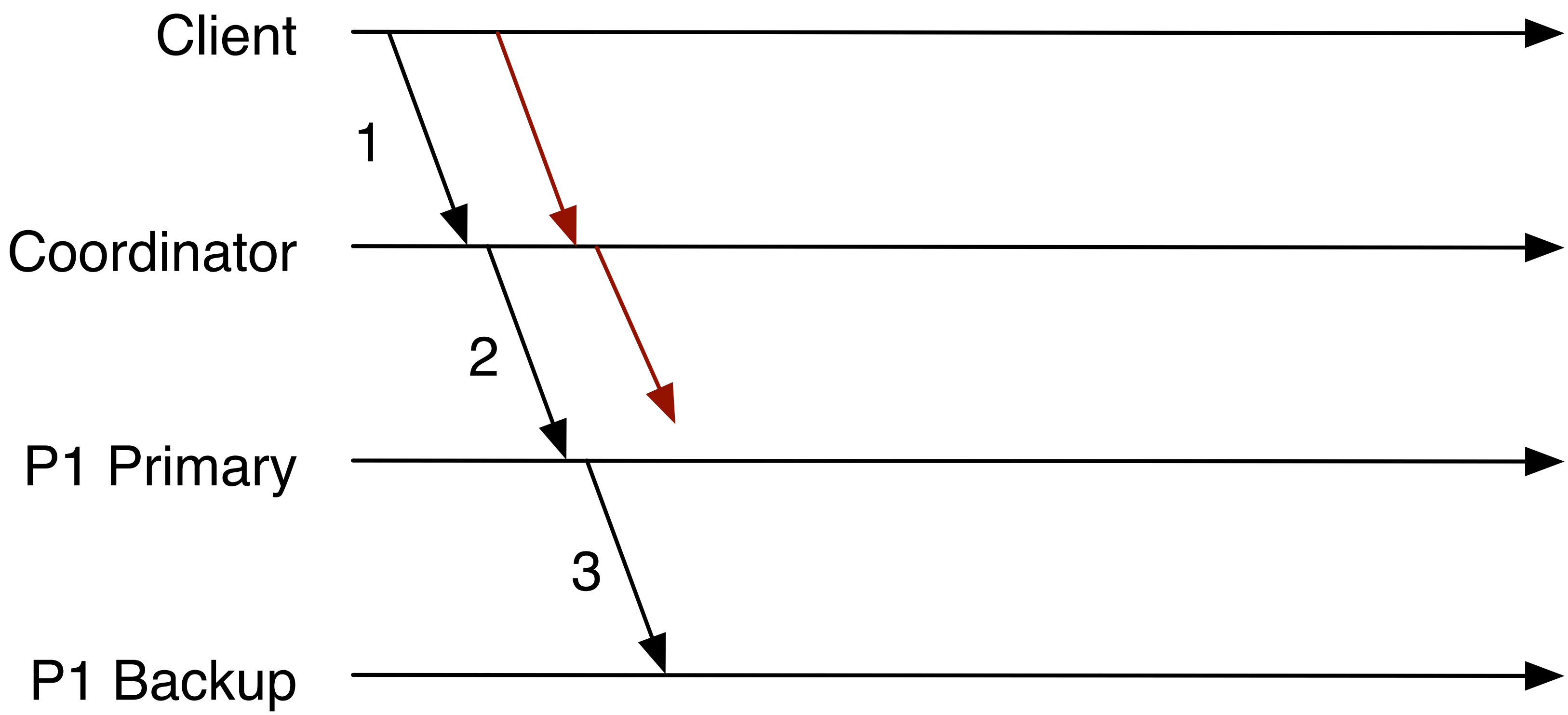
Blocking multi-partition



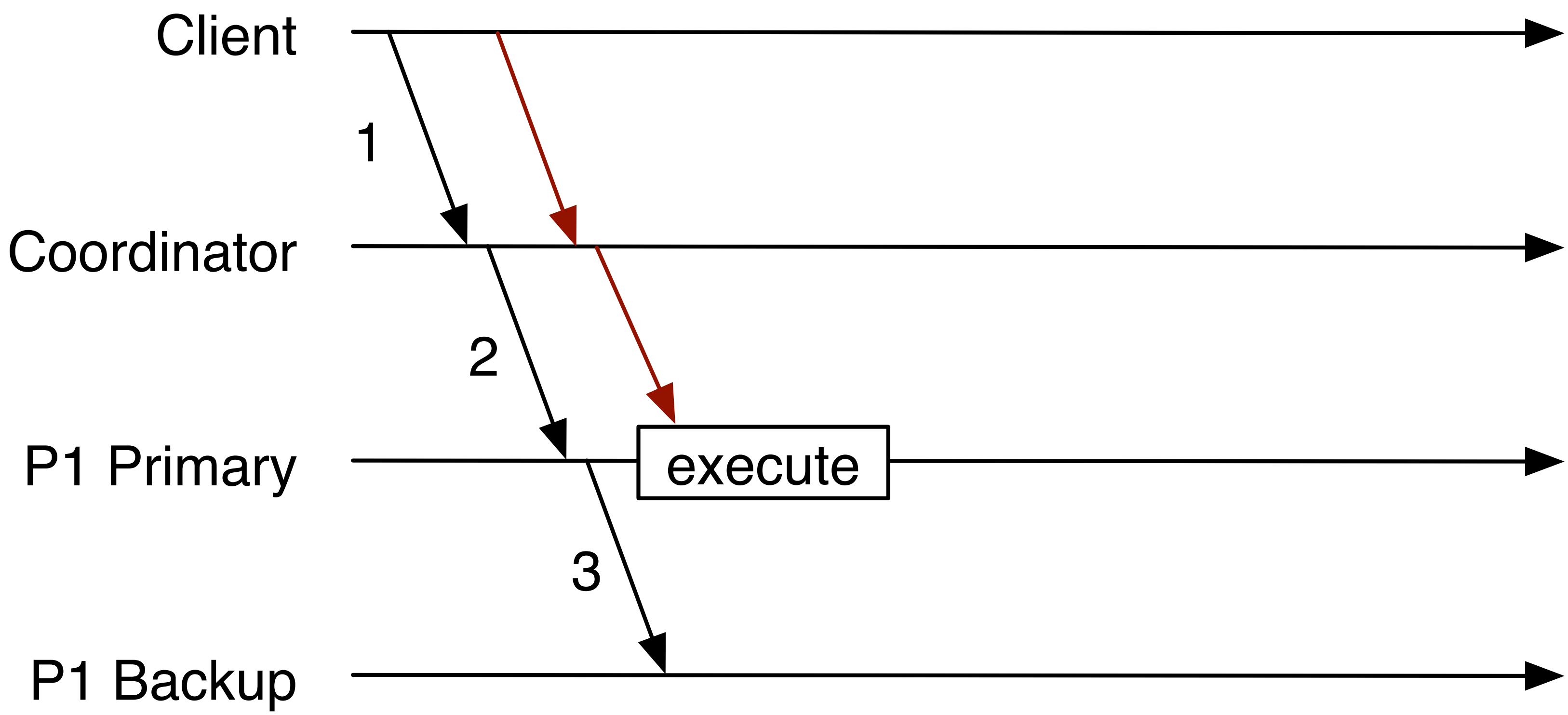
Blocking multi-partition



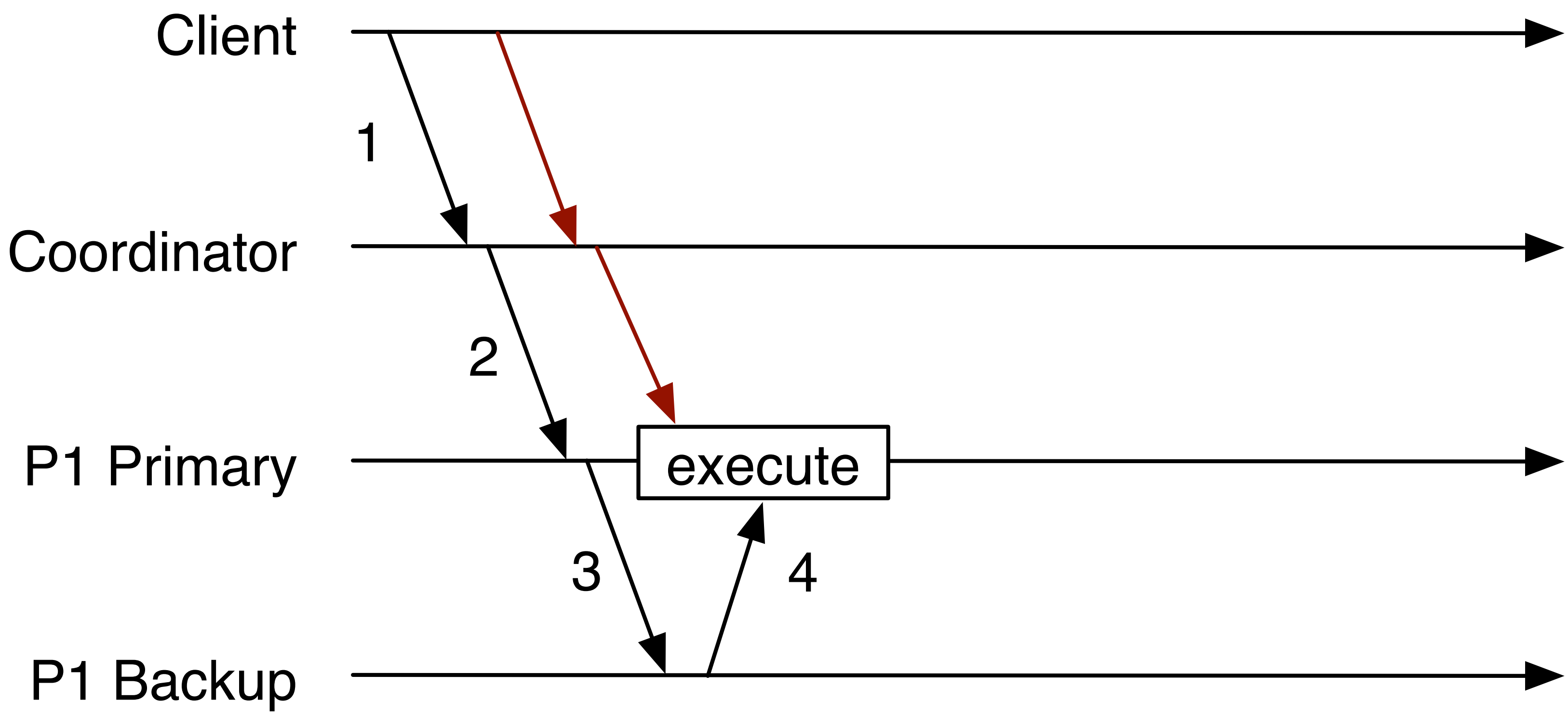
Blocking multi-partition



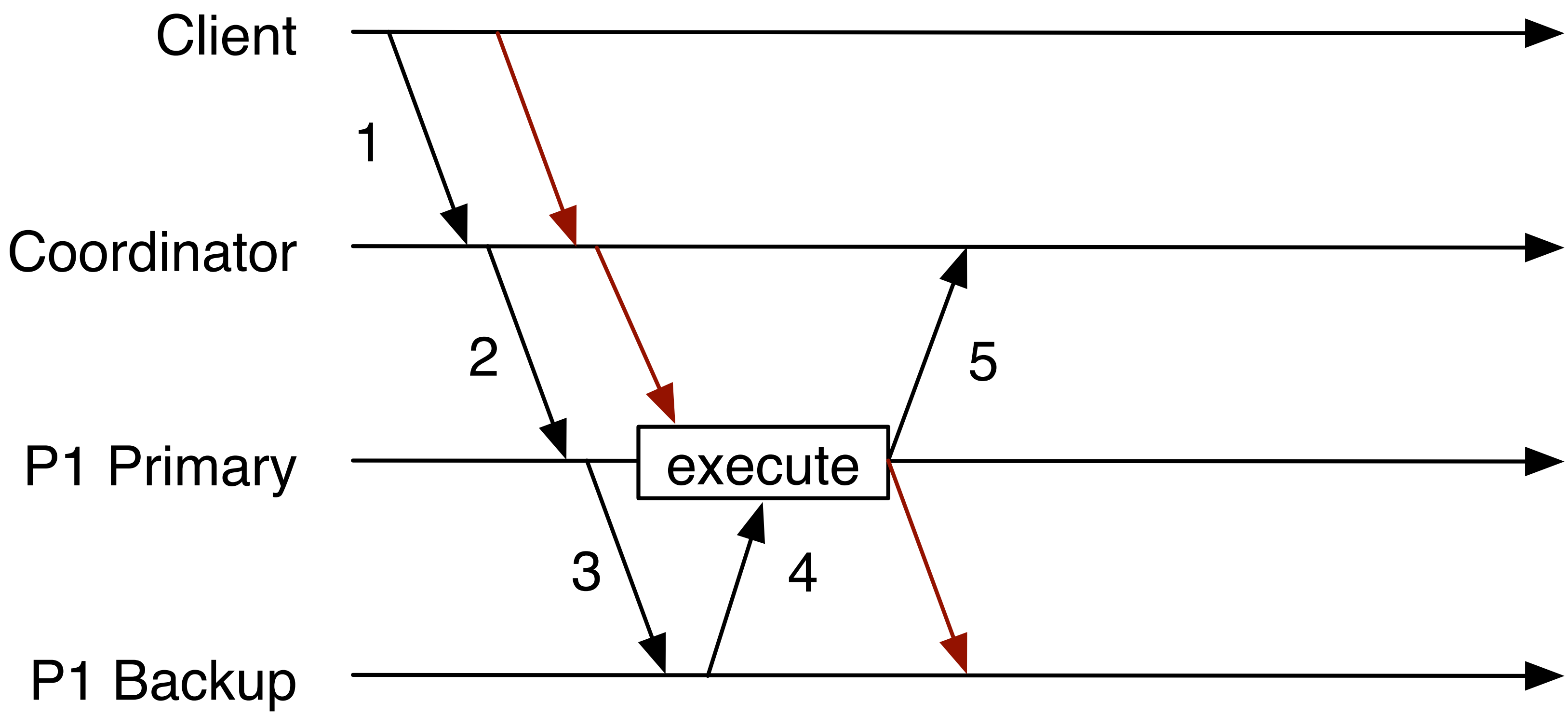
Blocking multi-partition



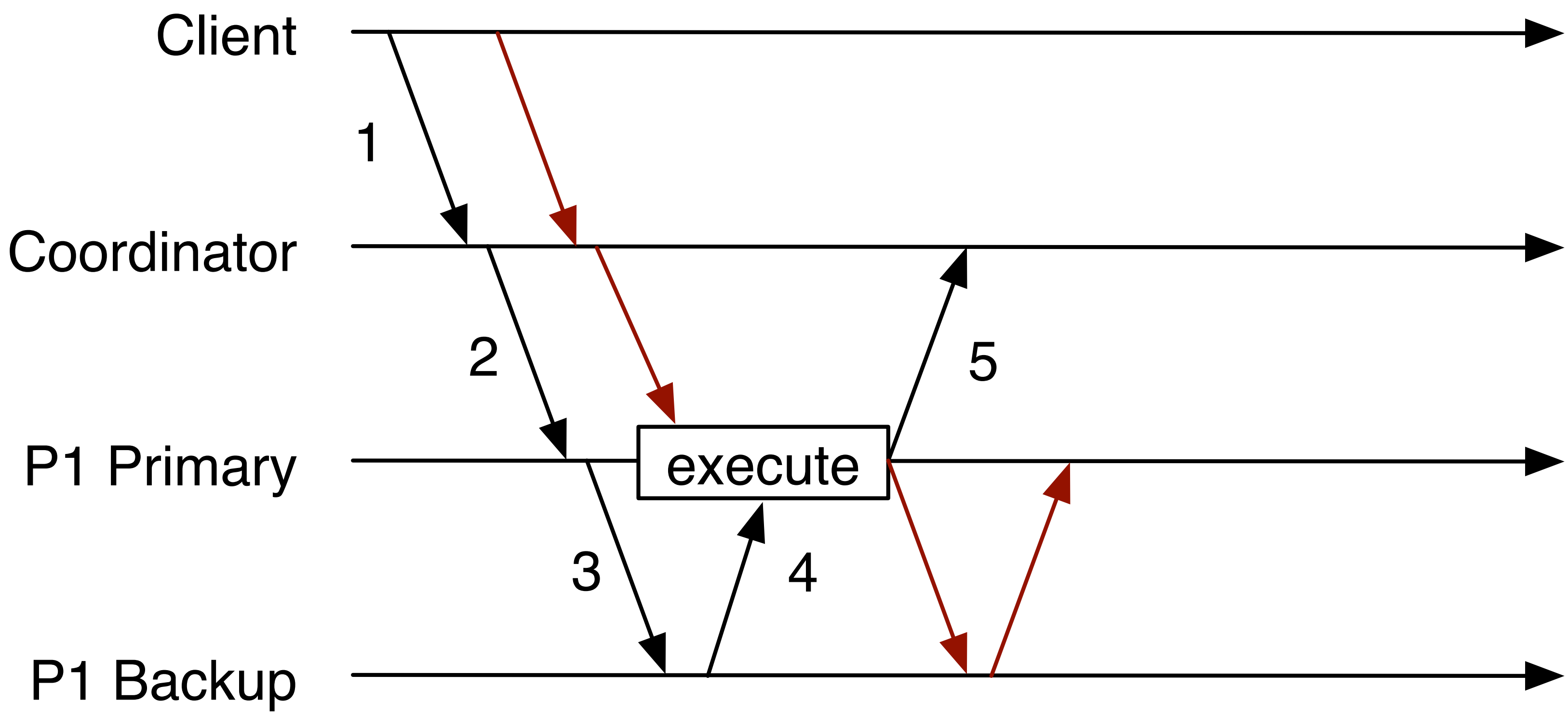
Blocking multi-partition



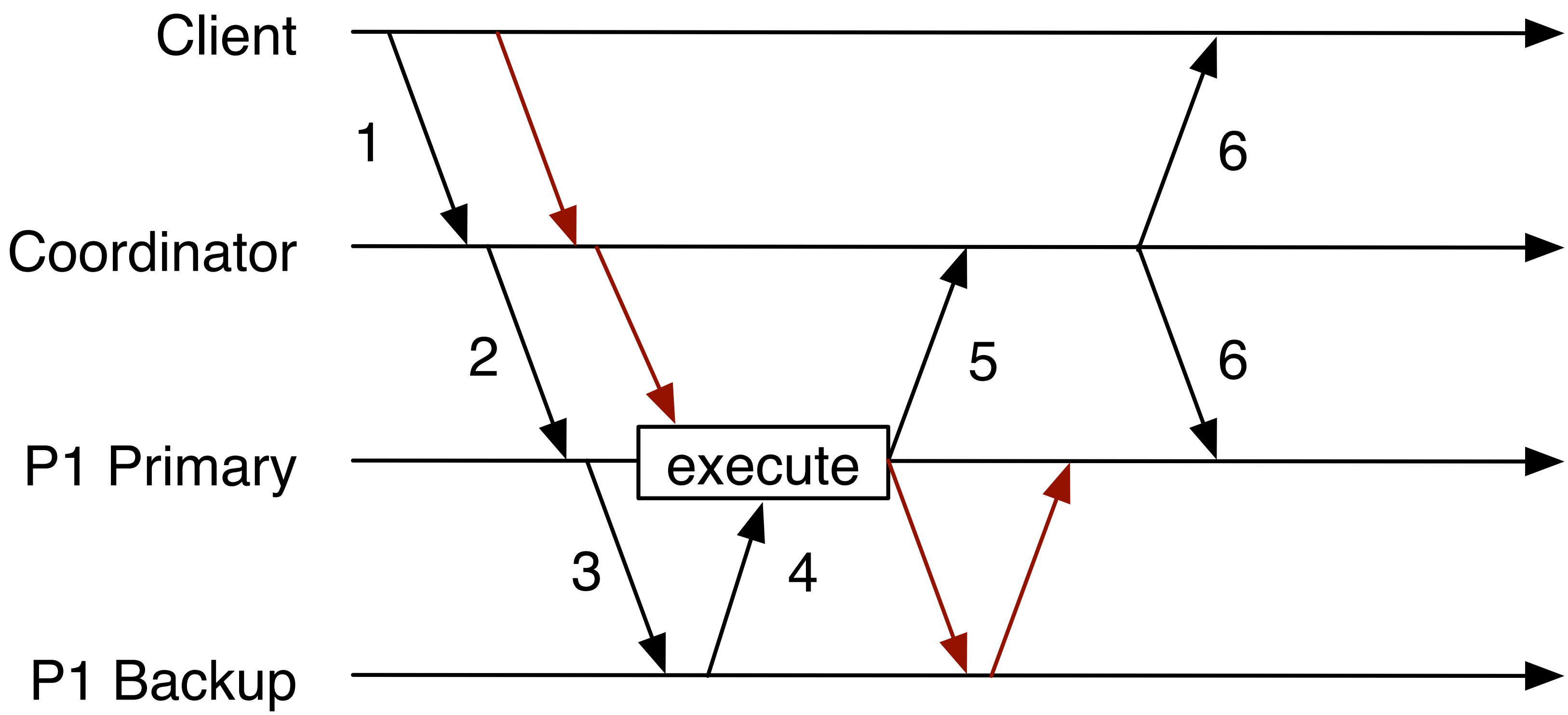
Blocking multi-partition



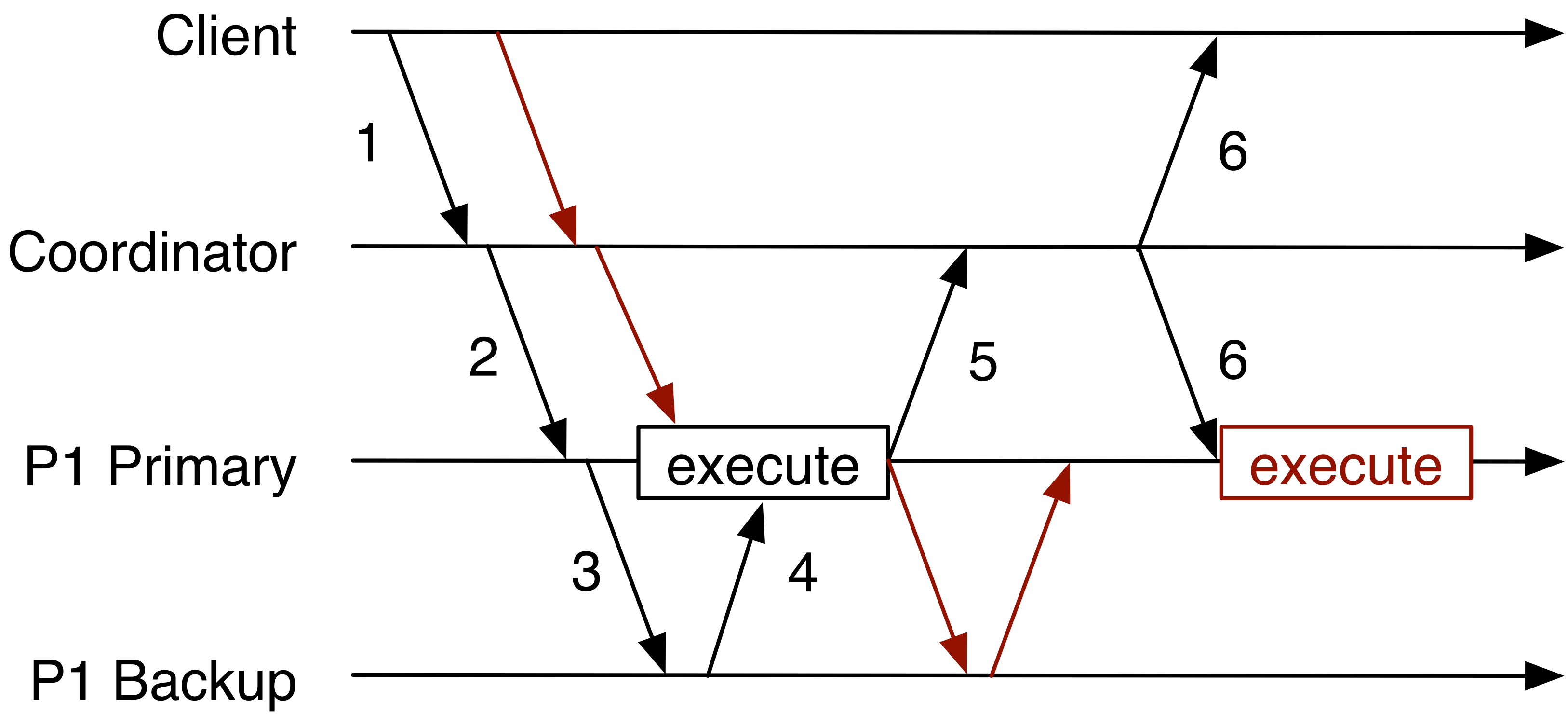
Blocking multi-partition



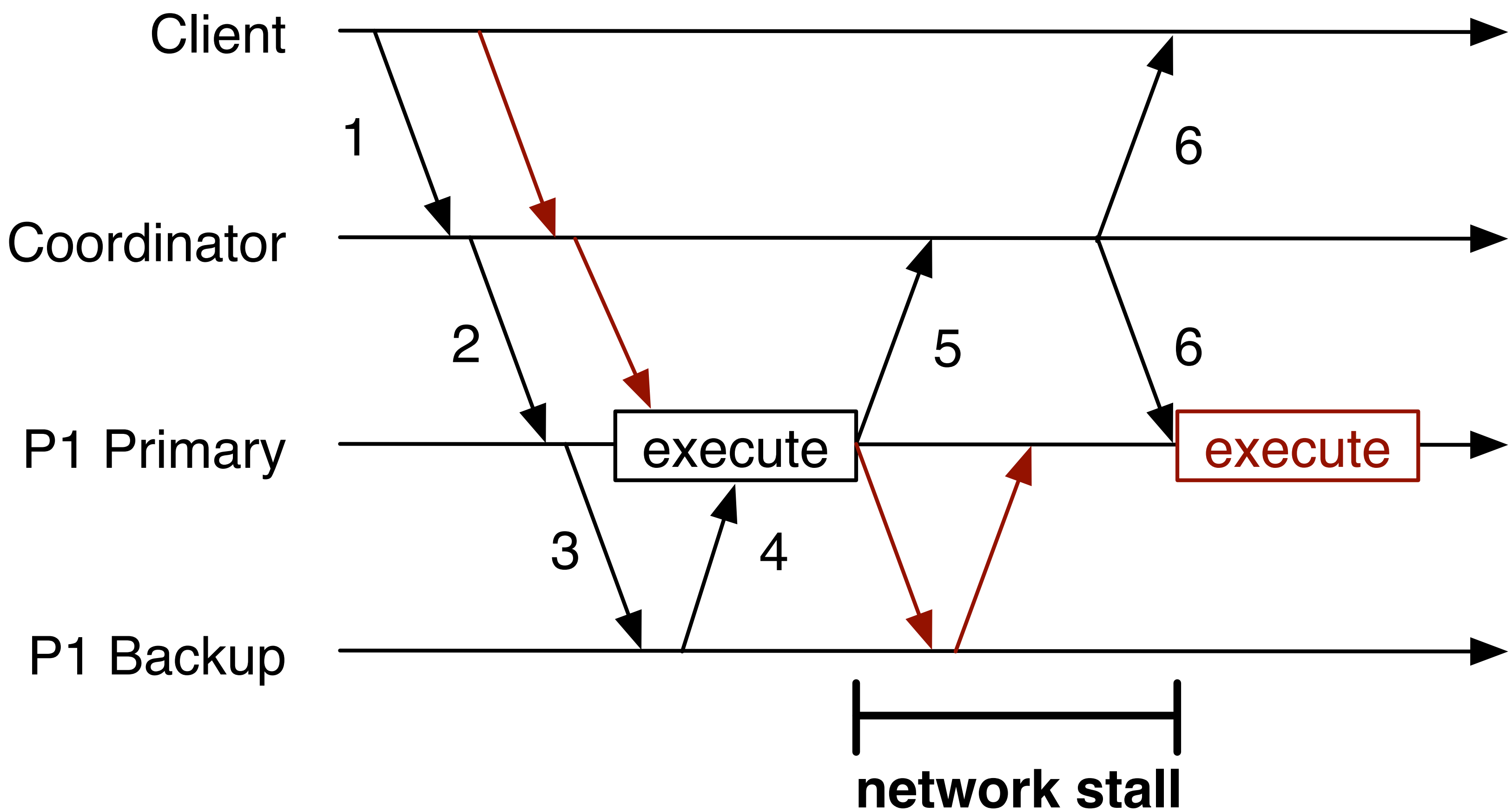
Blocking multi-partition



Blocking multi-partition



Blocking multi-partition



Solutions: Two-phase locking

+ Execute non-conflicting txns during stall

– Locking overhead

– Deadlocks

Solutions: Speculative CC

While waiting for commit/abort, speculatively execute other transactions

+ No locking overhead

– Need global transaction order

– Cascading aborts

Solutions: Speculative CC

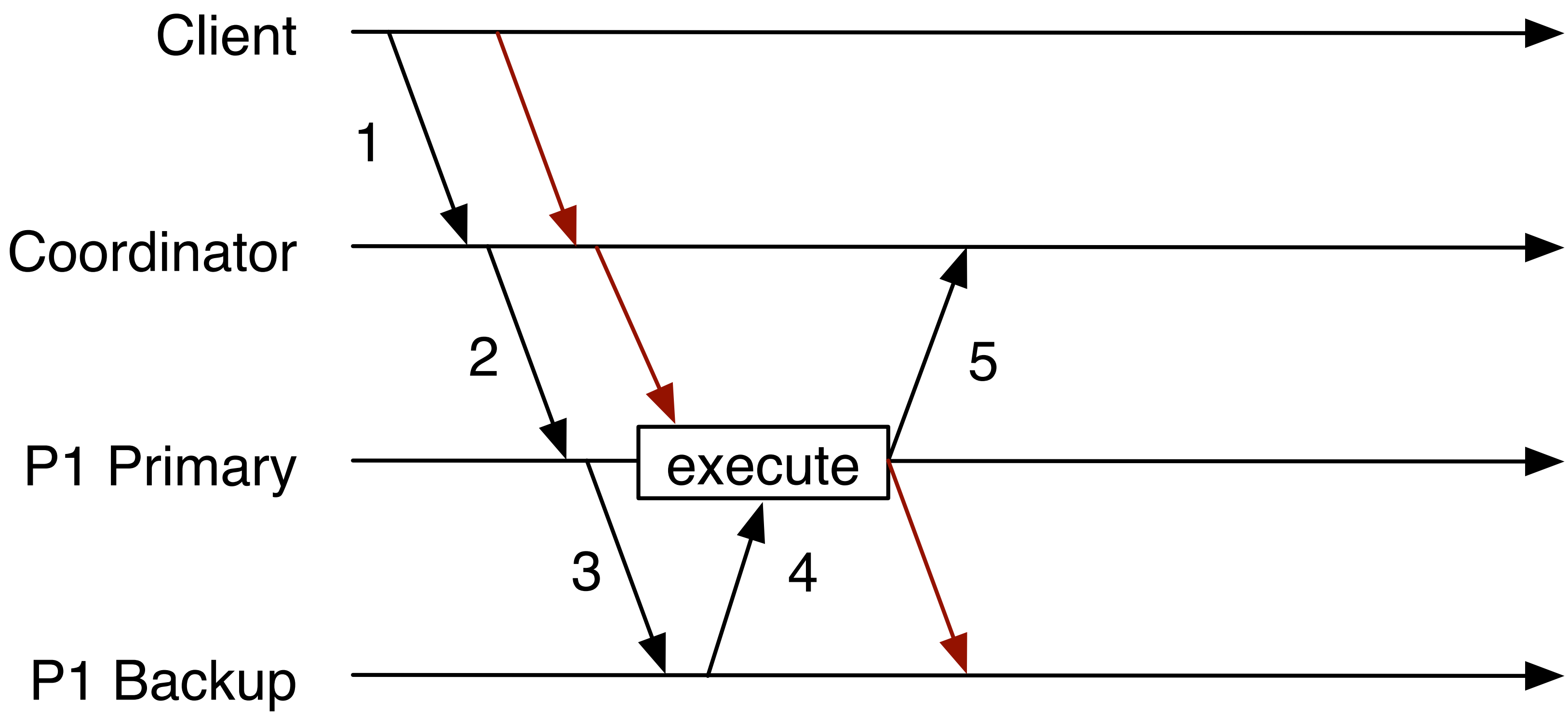
While waiting for commit/abort, speculatively execute other transactions

+ No locking overhead

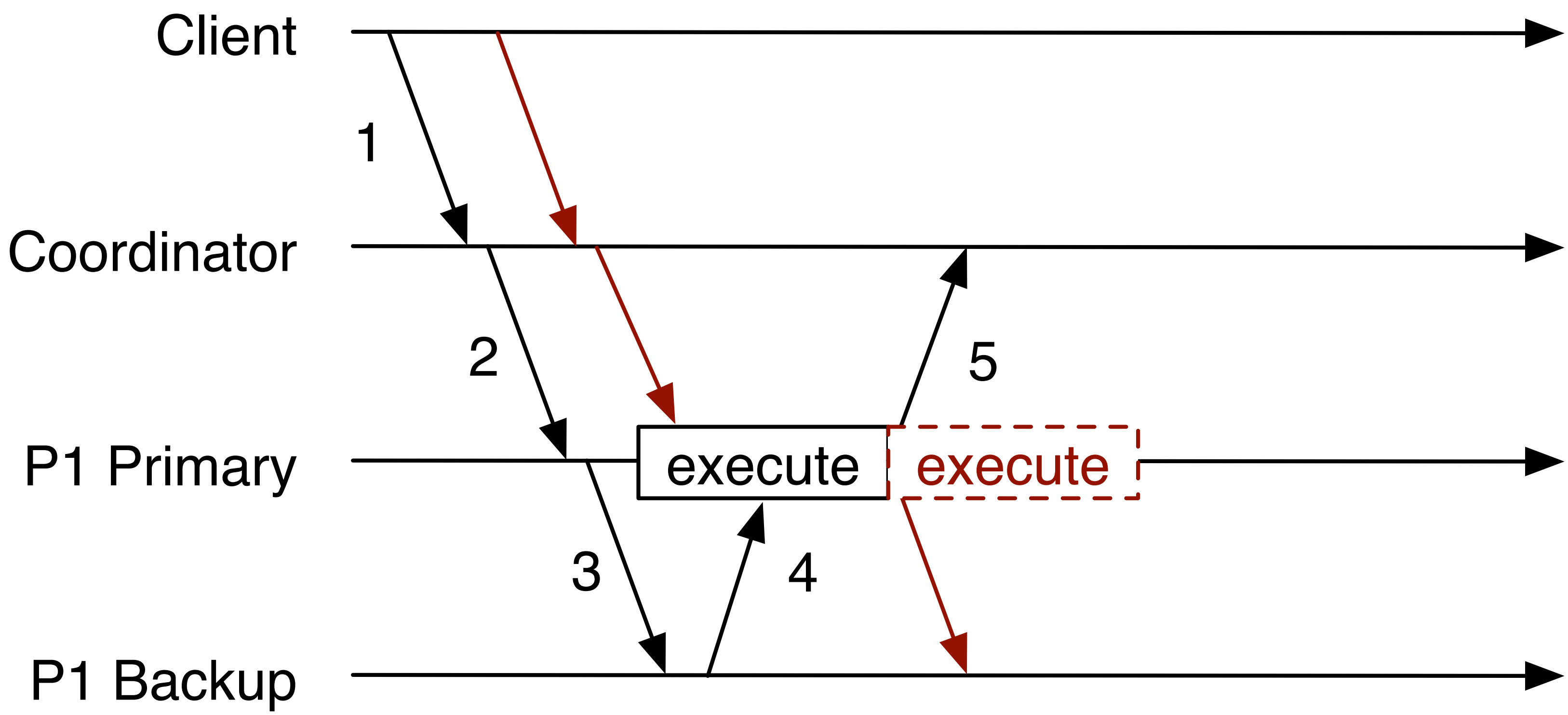
– Need global transaction order

– Cascading aborts

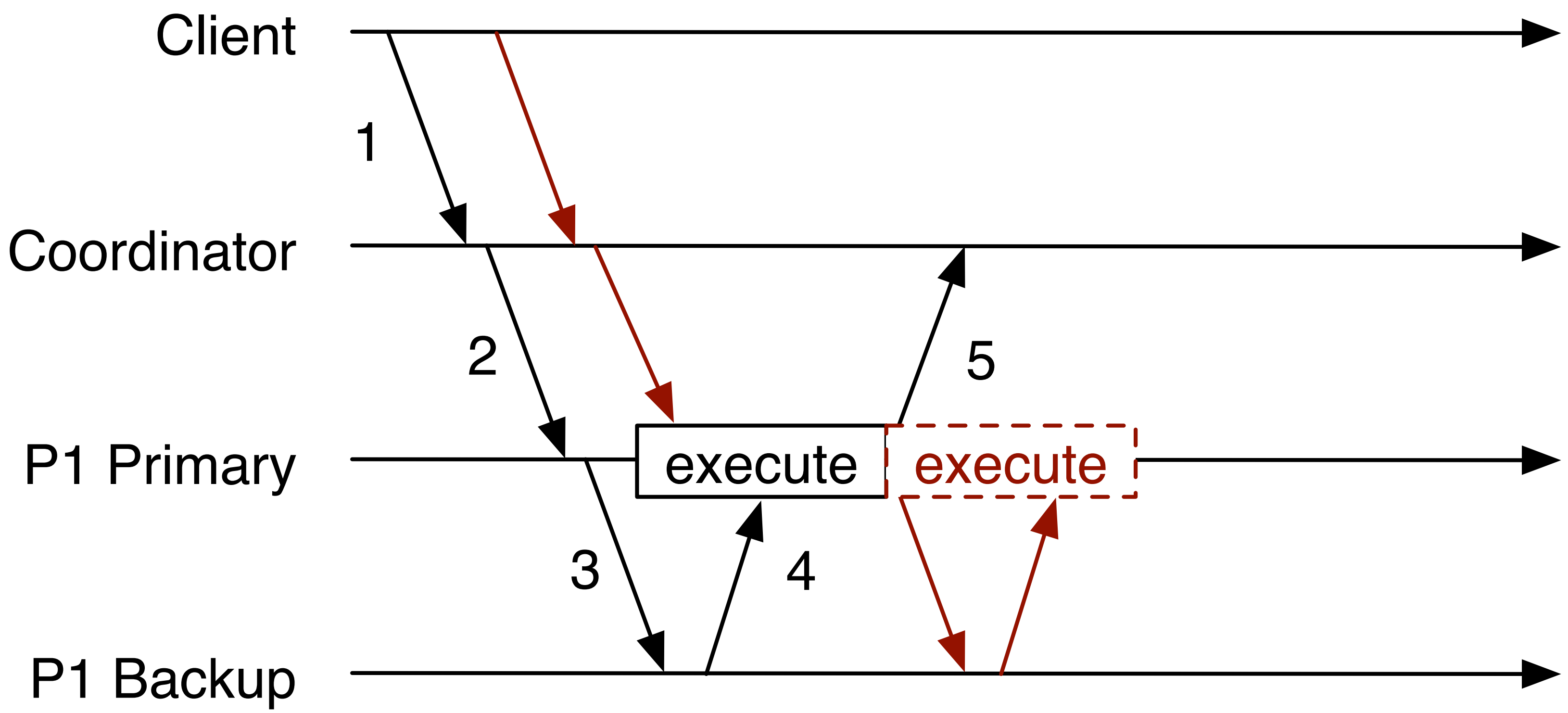
Speculative multi-partition



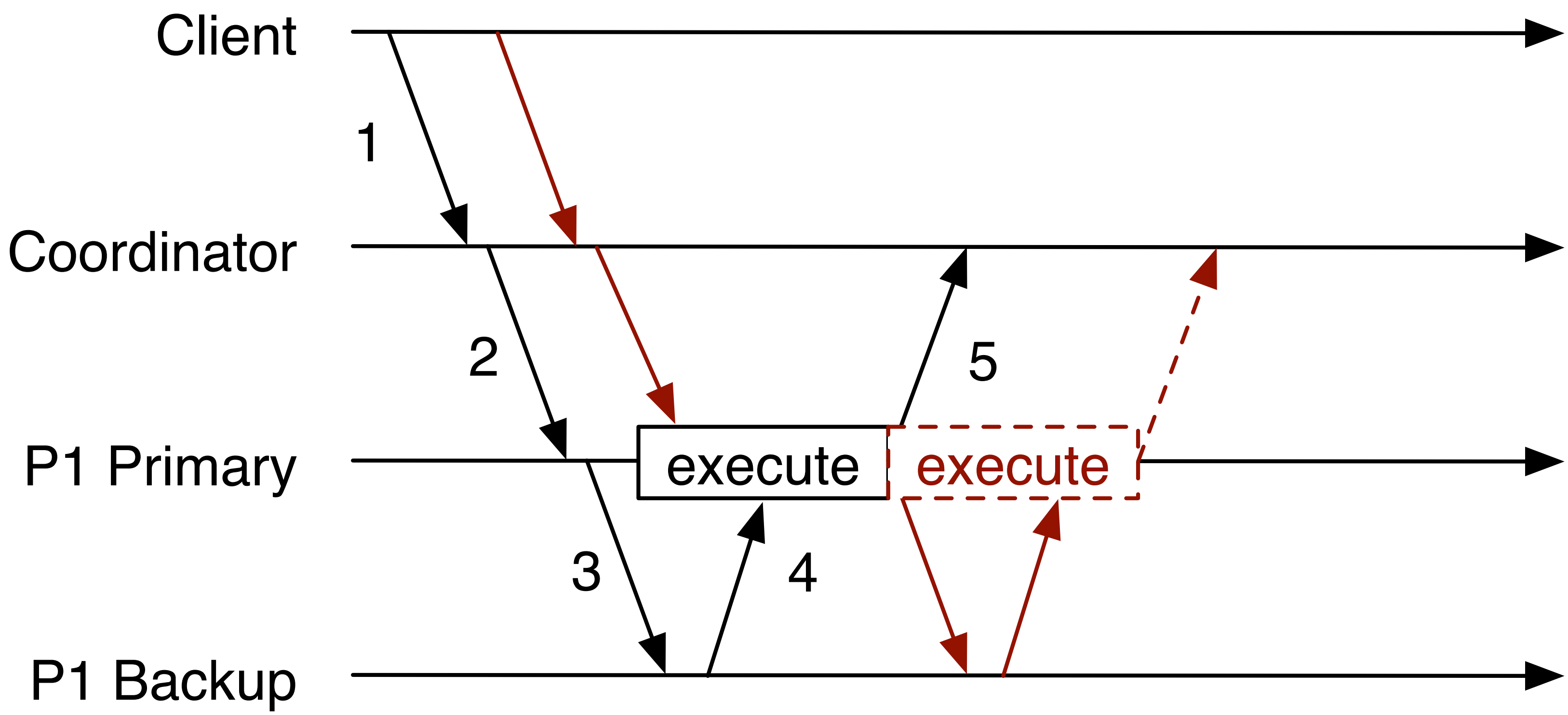
Speculative multi-partition



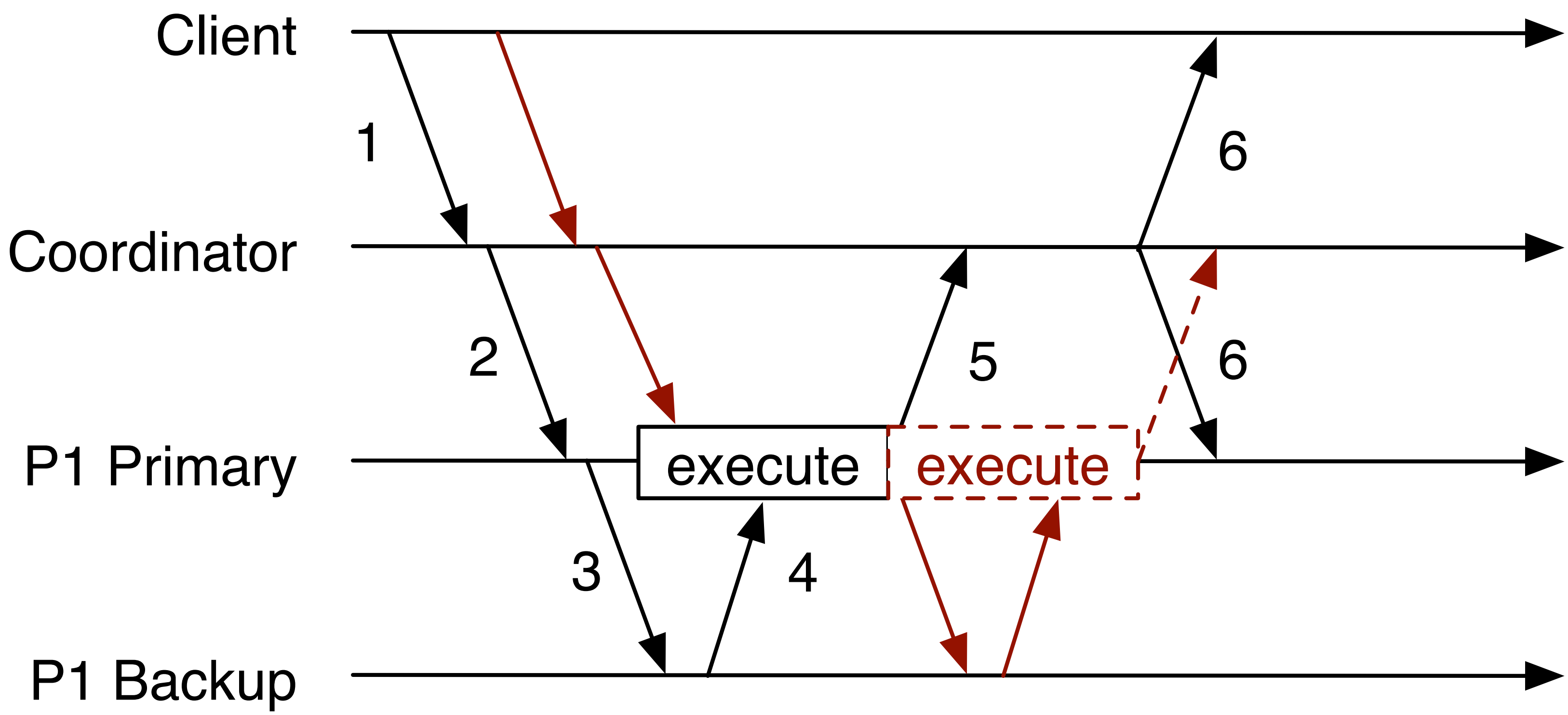
Speculative multi-partition



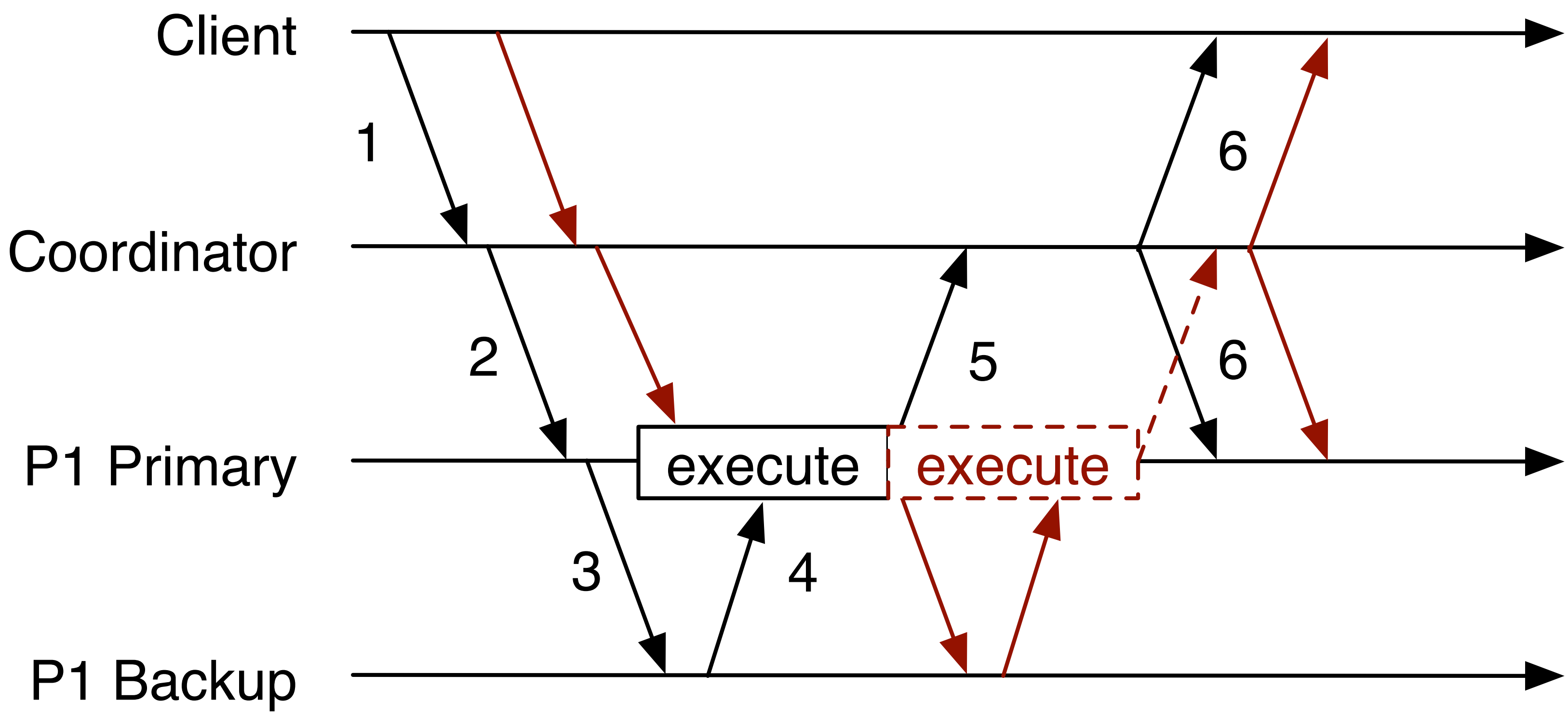
Speculative multi-partition



Speculative multi-partition



Speculative multi-partition



Speculation limitation

Transactions with multiple rounds: need network stall

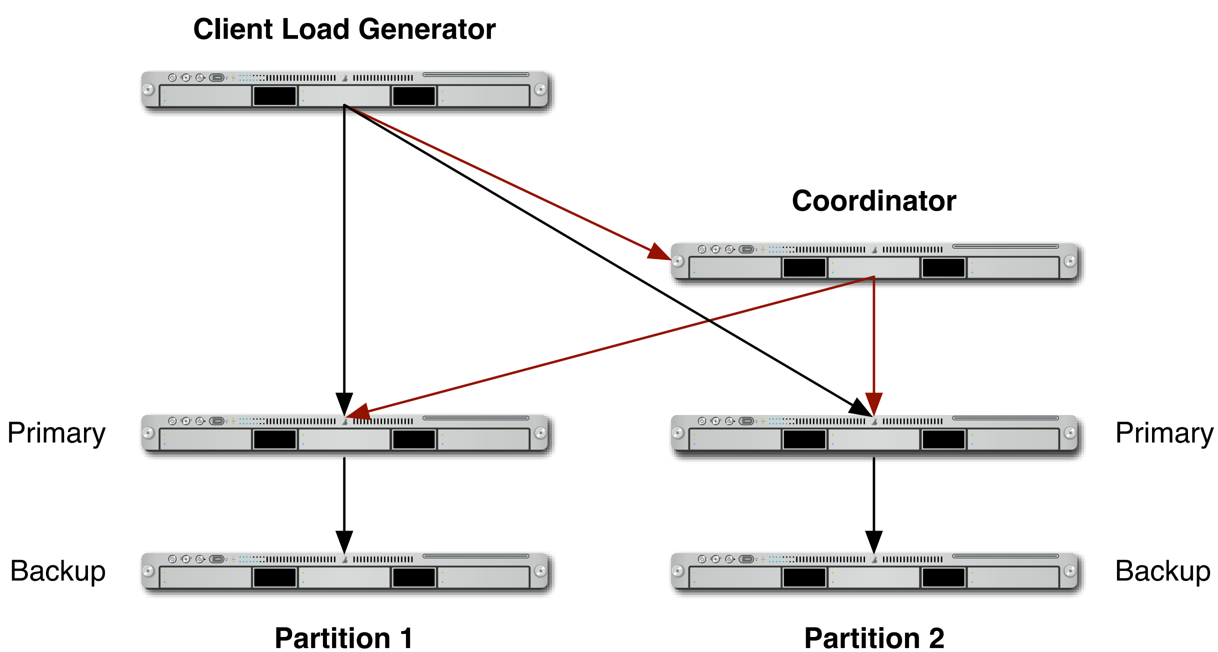
Example:

1. Read x on partition 1, y on partition 2
2. Update x = f(x, y); y = f(x, y)

Microbenchmark

Two partitions of a single table

`(id INTEGER PRIMARY KEY, value INTEGER)`



Microbenchmark

Single partition transaction:

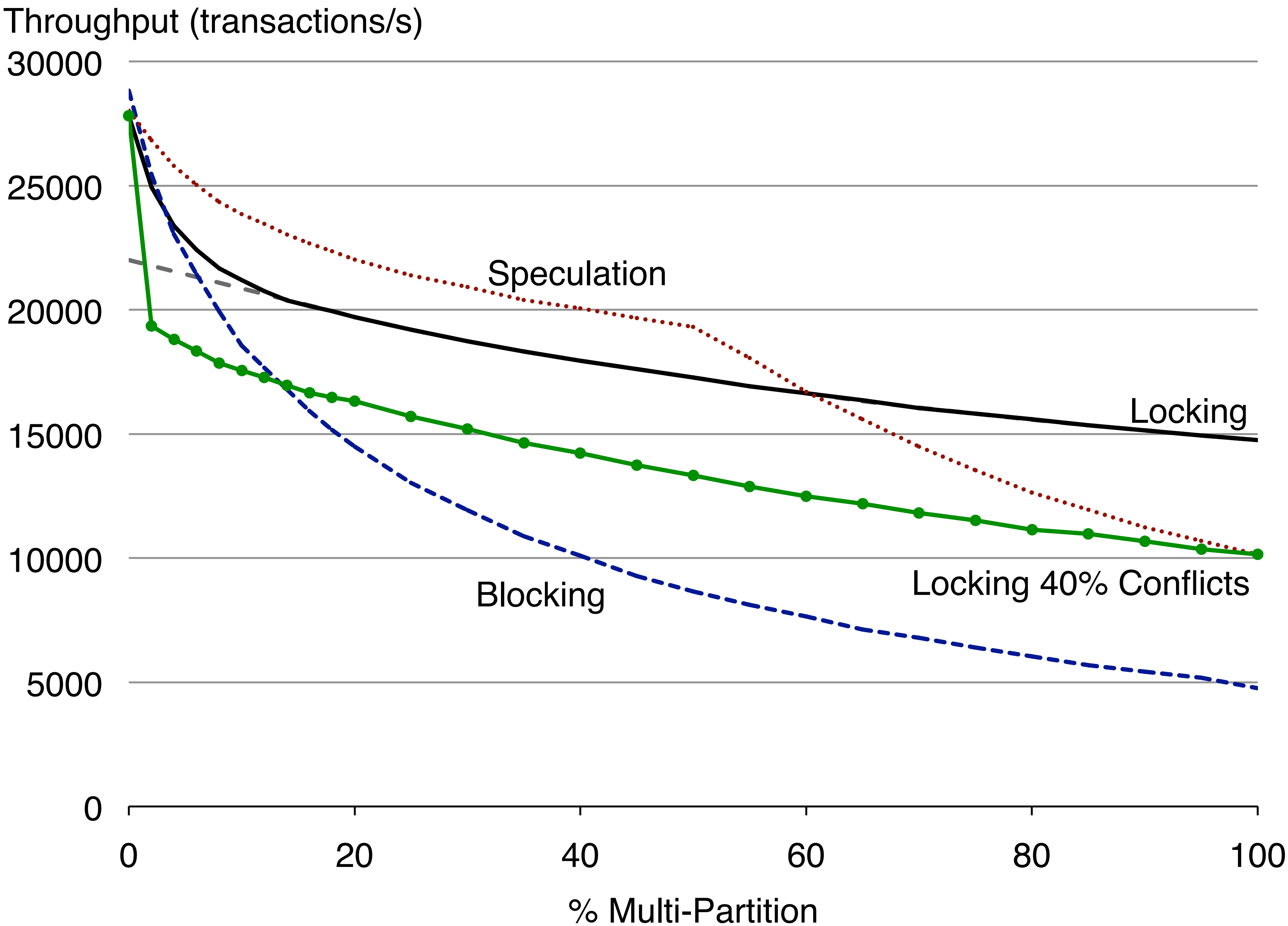
read/write keys on one partition

Multi-partition transaction:

access half keys from each partition

single partition work = multi-partition work

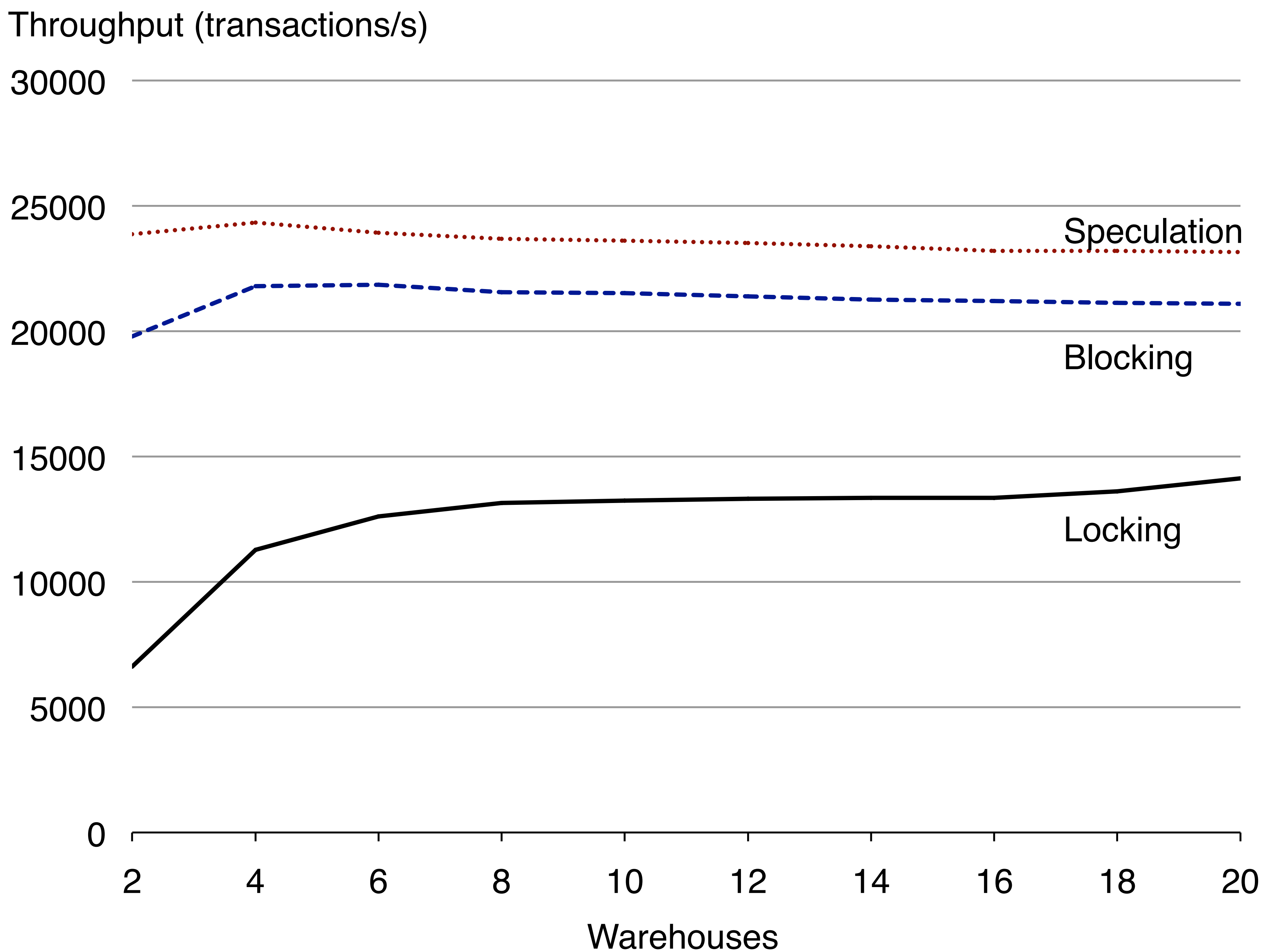
No deadlocks, no aborts, no conflicts



Throughput (transactions/s)
30000
25000
20000
15000
10000
5000
0
0
20
40
60
80
100
% Multi-Partition
Speculation
Locking
Locking 40% Conflicts
Blocking

TPC-C Based Benchmark

- ~11% multi-partition transactions
- More complex locking
- Many conflicts
- Some deadlocks
- Some aborts


Throughput (transactions/s)
30000
25000
20000
15000
10000
5000
0
Speculation
Blocking
Locking
2
4
6
8
10
12
14
16
18
20
Warehouses

Speculative CC

Better for "mostly partitionable" apps on main memory DBs

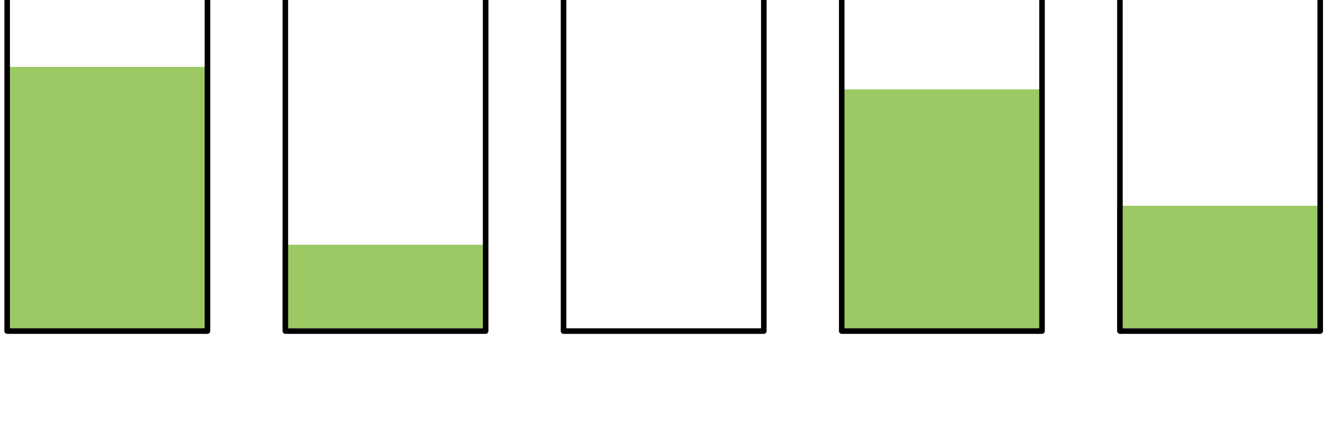
- Up to 2X throughput
- No locking overhead
- No deadlocks

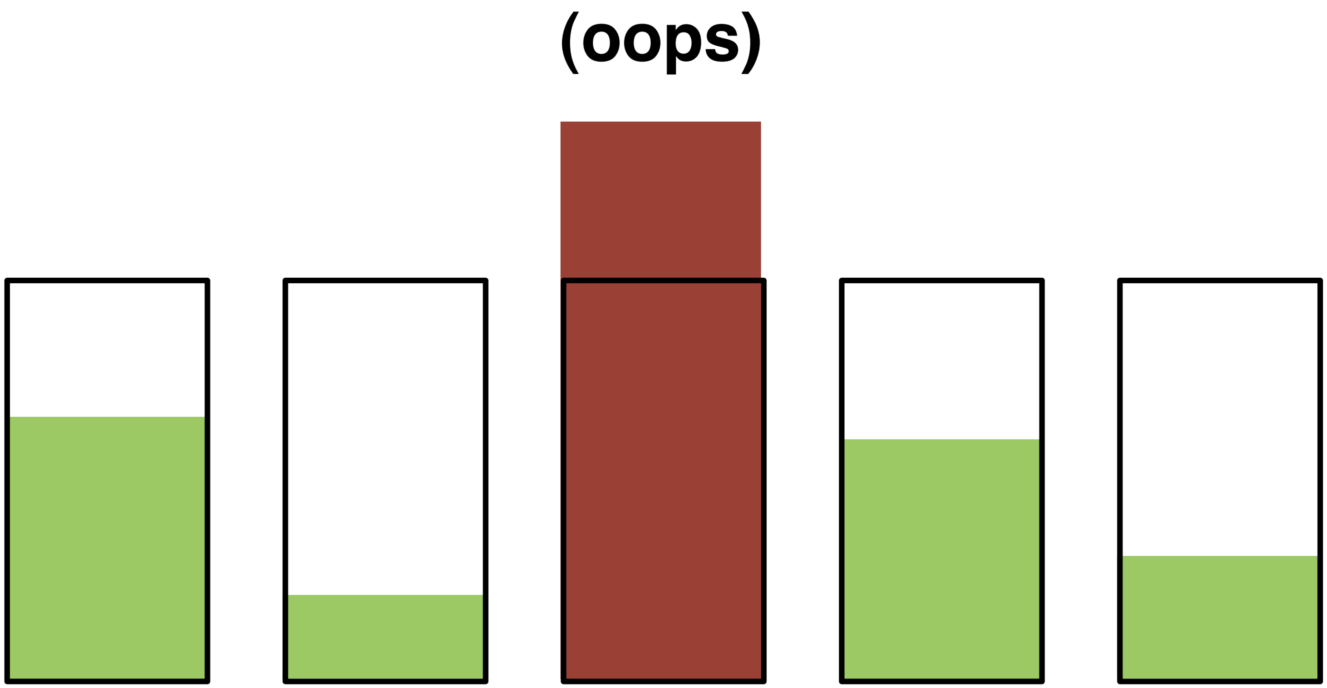
Novel features

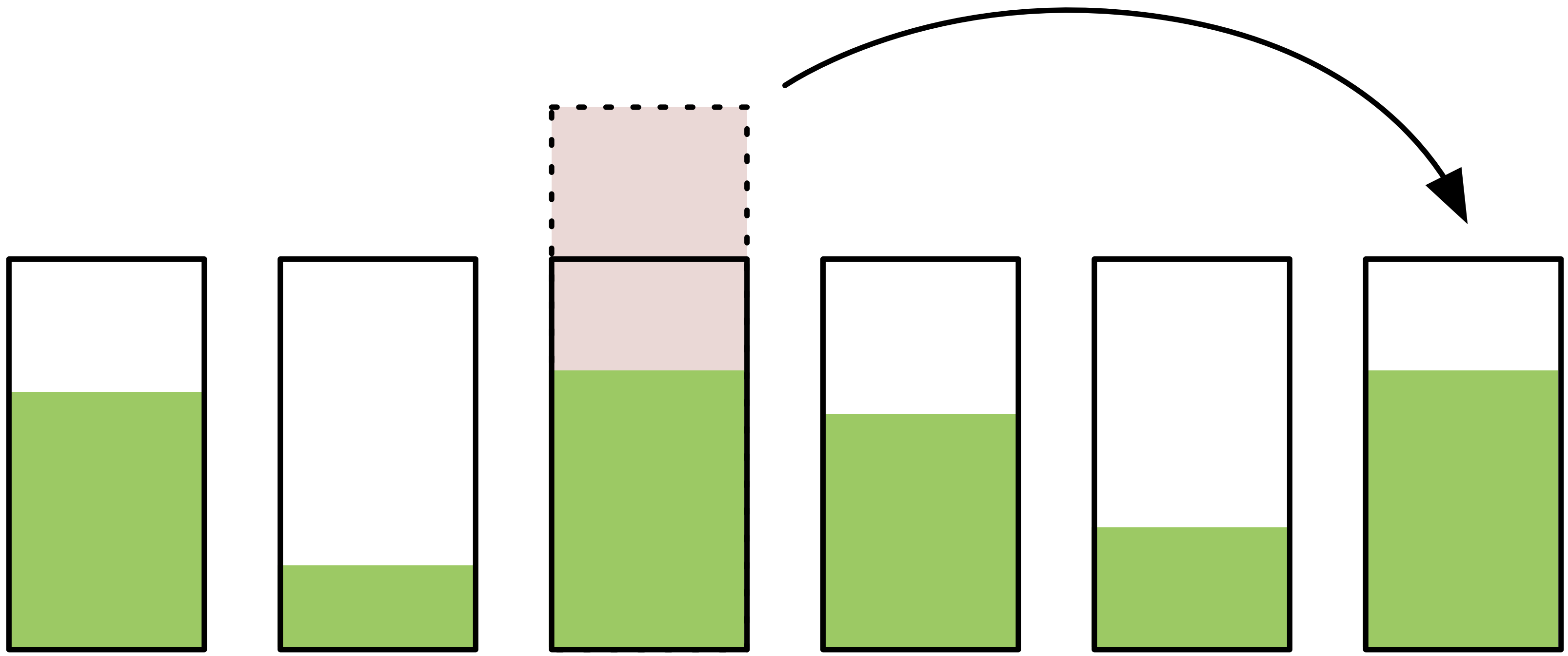
Reusable infrastructure for OLTP databases

Speculative concurrency control

Live migration using a cache-based approach




(oops)



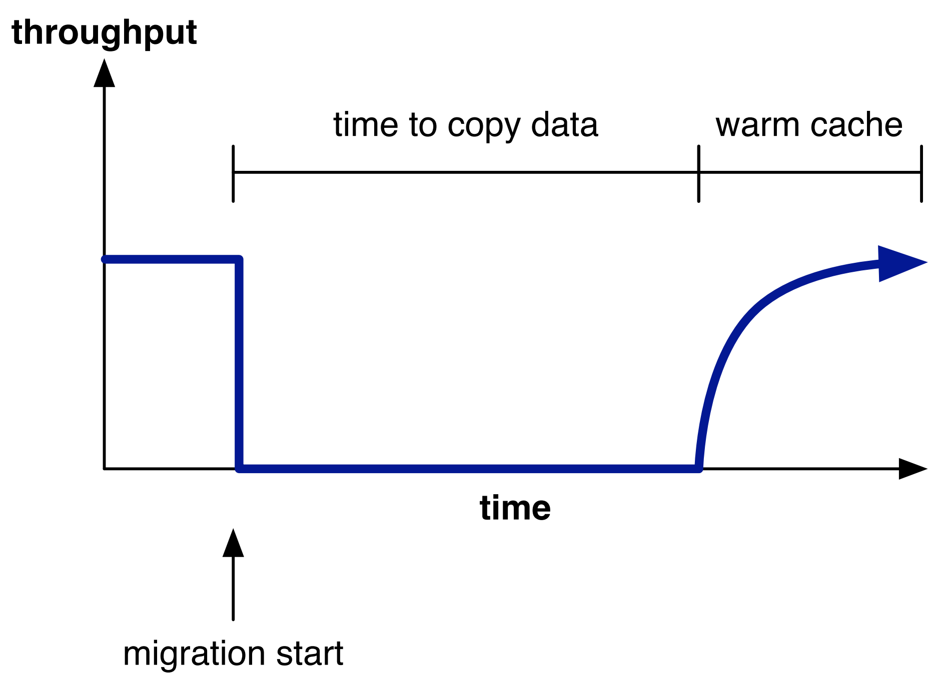
Live migration: elastic scalability

Move data from a *source* partition to a *destination* partition

- No impact on the partition being migrated
- No impact on other operations on the source
- Partial migration: move *part* of a partition
- Quickly use the destination's resources

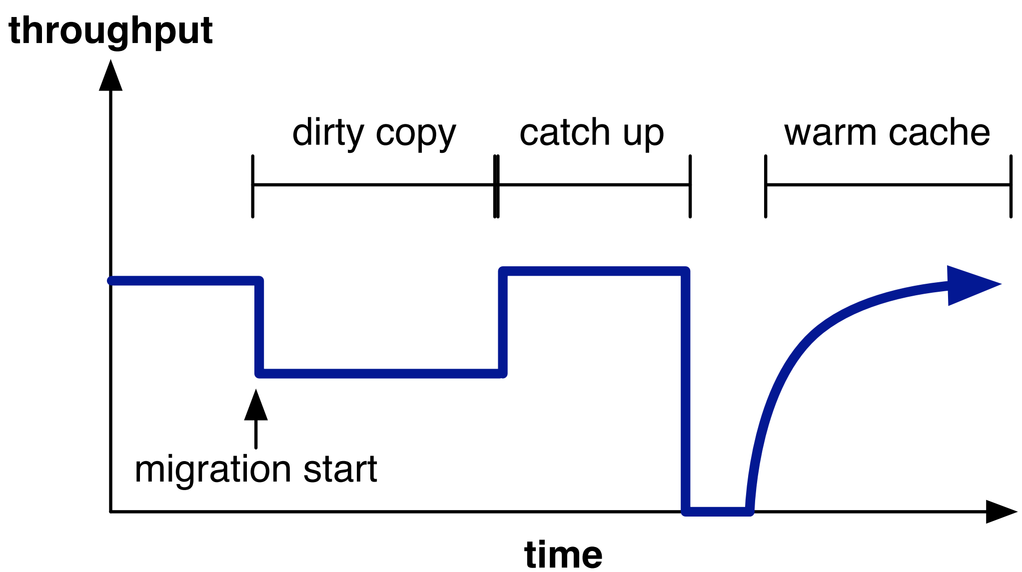
Today: Stop and copy

- Shut down the partition
- Copy data to another machine
- Restart and redirect requests



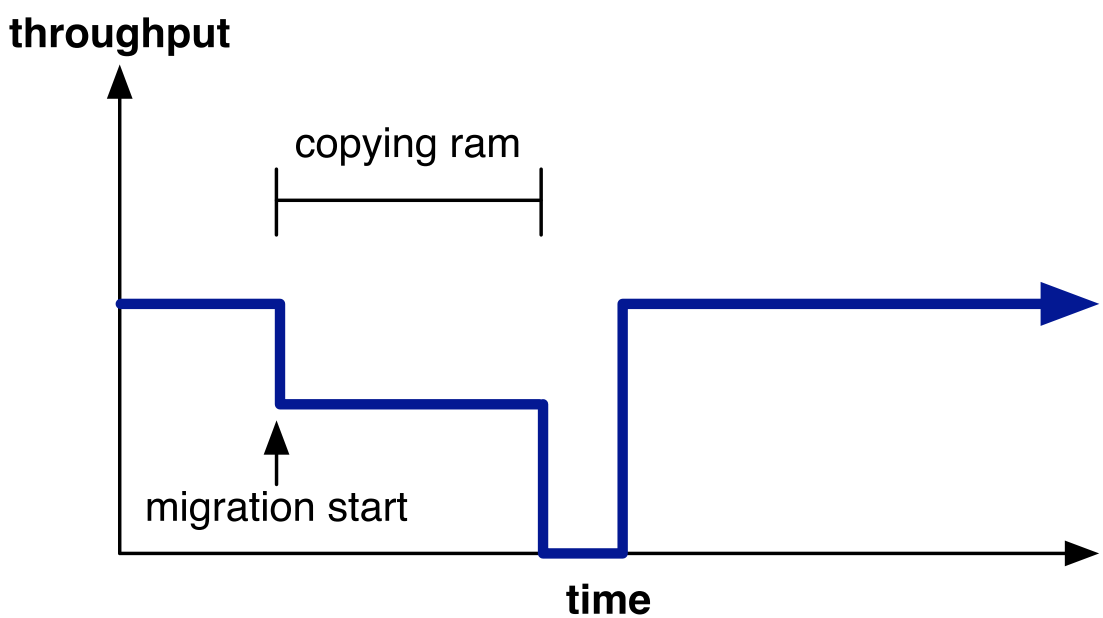
Today: Replica failover

- Copy the partition while running
- Catch up on the destination
- Pause, finish catching up, fail over



Today: Virtual machines

- Copies memory pages iteratively
- Pause, copy last few pages
- Restart on destination



Wildebeest: Dtxn's live migration

Logical "copy on demand" policy: switch immediately, fetch missing data as needed

- Immediately reduces load
- Permits partial migration

Disadvantage:

- Requires integration with the storage engine
- Used Relational Cloud (MySQL)

Migration process

1. Prepare destination
2. Drain the source
3. Redirect to the destination
4. Destination fetches tuples on demand
5. Clean up migration

Complicated parts

Query rewriting:

- What data to fetch from the source?

Range tracking

- Has this data already been fetched?

Physical vs Logical

Physical: Efficiently copy everything

Logical: Rebuilds indicies on the destination

- Permits copying referenced data only
- Destination should have excess capacity
- Enables partial migration

Range Tracking

Must remember what data has been migrated

Record a set of migrated ranges for each index (called a *range set*)

- Check range set for query
- If data is local: execute locally
- If not: fetch tuples, insert, then execute locally

Query rewriting

Rewrite queries to fetch missing data from source

- SELECT x, y, z -> SELECT *
- Joins: Decompose into multiple queries
- Deletes: Must be executed on both source and destination

Fault tolerance and recovery

Basic principle: The state of the partition is the source plus the changes in the destination

- Failure of either source *or* destination causes partition to be unavailable

Optimization: Batching

Instead of fetching one tuple at a time, fetch a group.

- Larger batches = migration completes faster
- Larger batches = more impact on source/destination

Optimization: Batching

Instead of fetching one tuple at a time, fetch a group.

- Rewrite query: id = x -> x <= id LIMIT batch
- Larger batches = migration completes faster
- Larger batches = more impact on source/destination

Write-behind caching

When inserting migrated data, don't log for durability

- Recovery: data is durable on the source
- Updates: logged on destination as usual
- Improves performance for read-only queries
- Implementation: main-memory, non-durable cache engine on top of the normal engine

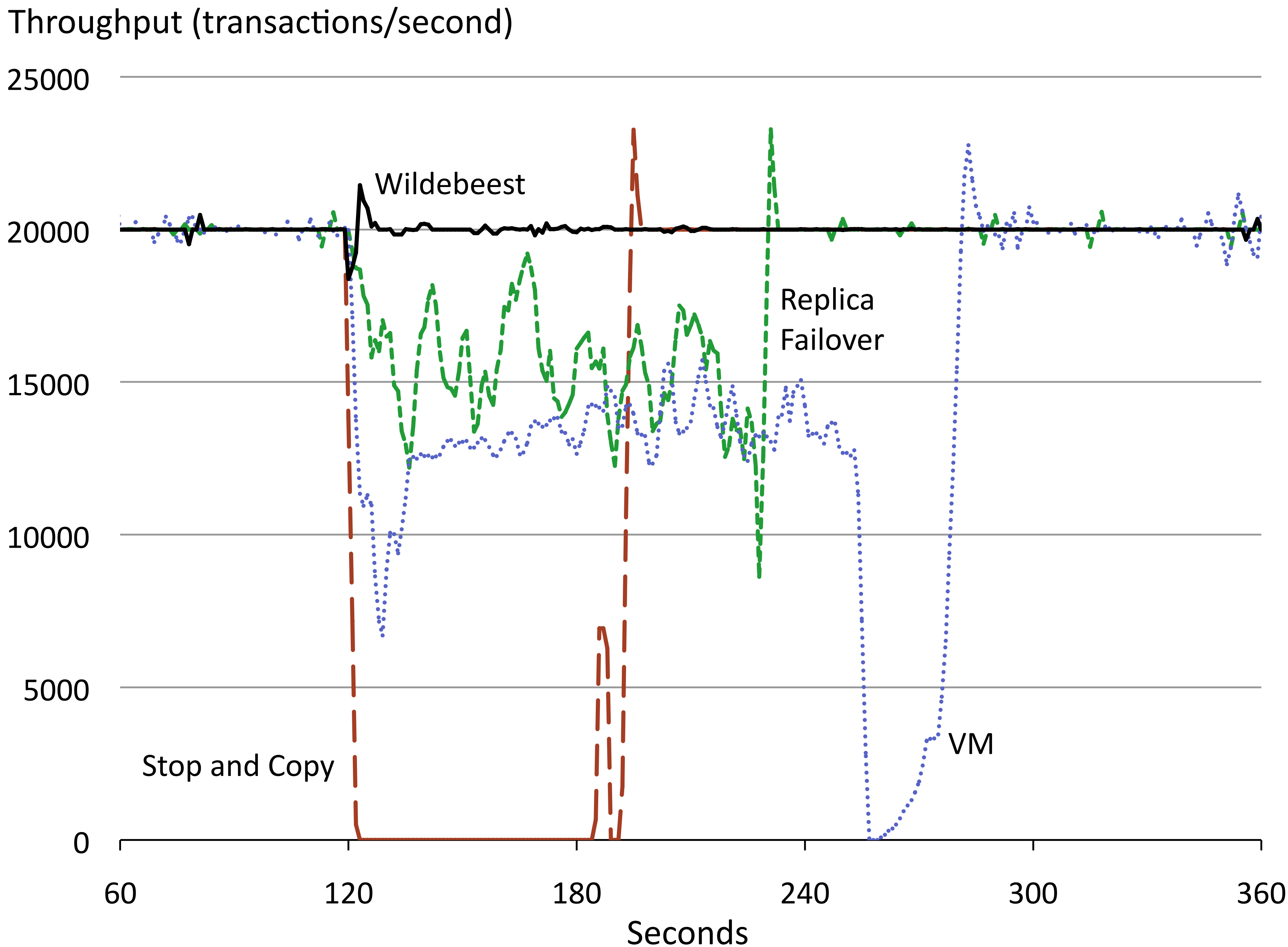
Experiments!

- Relational Cloud: MySQL backends
- YCSB-B: Fetches single rows with Zipfian distribution. 95% read, 5% update.

- 100M rows
- 6.5 GB data
- MySQL sized to store this in RAM
- Trigger migration after 2 minutes (120 s)

Complete migration; rate limited

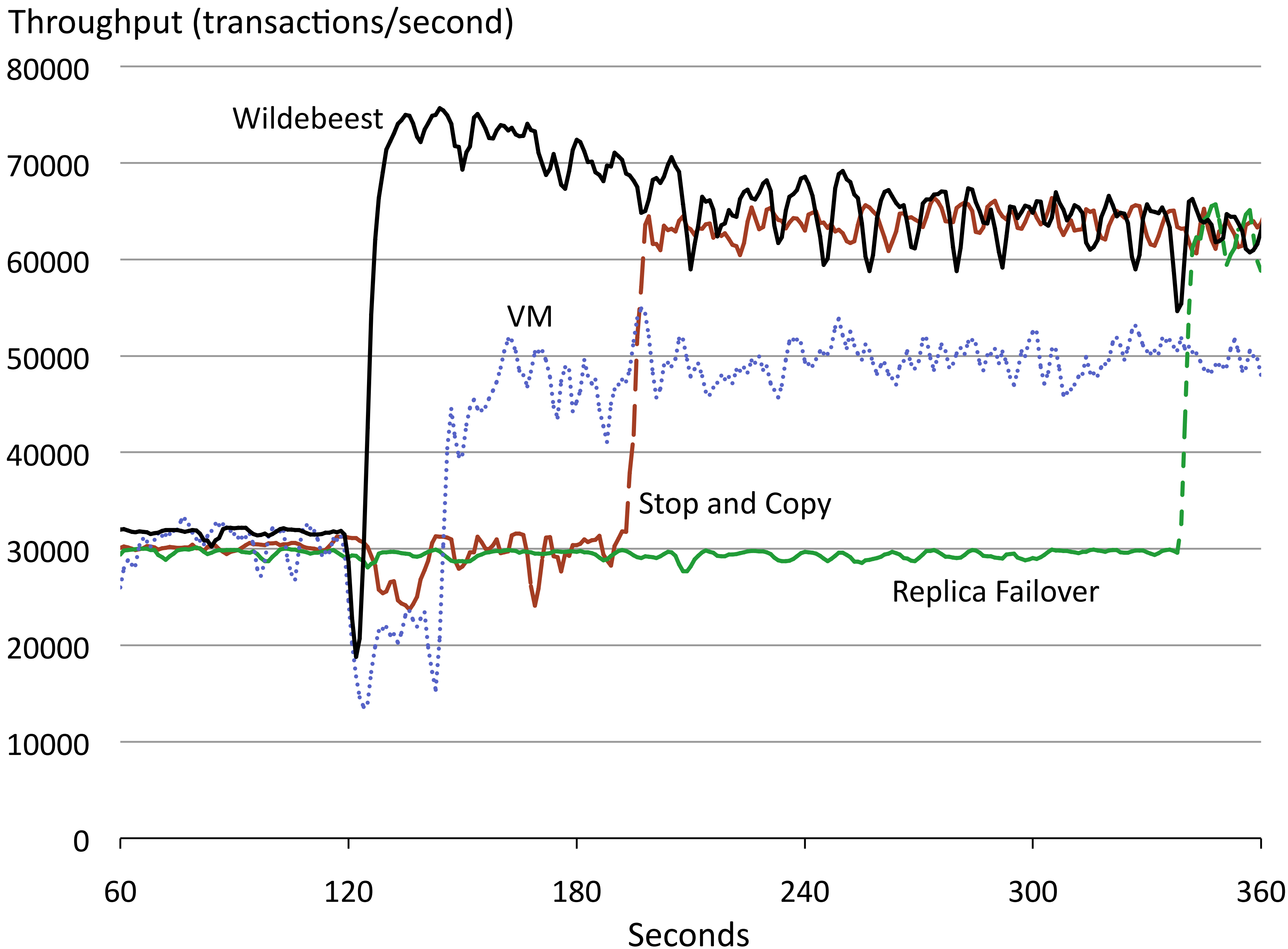
- Move the entire partition
- Limit client request rate to 20,000 transactions per second



Throughput (transactions/second)
25000
20000
15000
10000
5000
0
Wildebeest
Replica
Failover
VM
Stop and Copy
60
120
180
240
300
360
Seconds

Scale out scenario

- Partition is overloaded (maximum throughput)
- Move half the data to another machine
- VM: Start with pre-partitioned data



Throughput (transactions/second)
80000
70000
60000
50000
40000
30000
20000
10000
0
Wildebeest
VM
Stop and Copy
Replica Failover
60
120
180
240
300
360
Seconds

Conclusions

Logical migration: ideal for partial migration

Fetch on demand: responsive, with minimum impact on source

Wildebeest live migration allows distributed databases to be scaled on demand

Summary

Dtxn is a framework for building fault-tolerant distributed databases, specialized for memory-resident OLTP workloads

- Reusable infrastructure for OLTP databases
- Speculative concurrency control
- Live migration using a cache-based approach