



L-Store Concurrency Control: QueCC

Slides are adopted from Qadah, Sadoghi

QueCC - A Queue-Oriented, Control-Free Concurrency Architecture, ACM Middleware 2018

ECS 165A



Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science

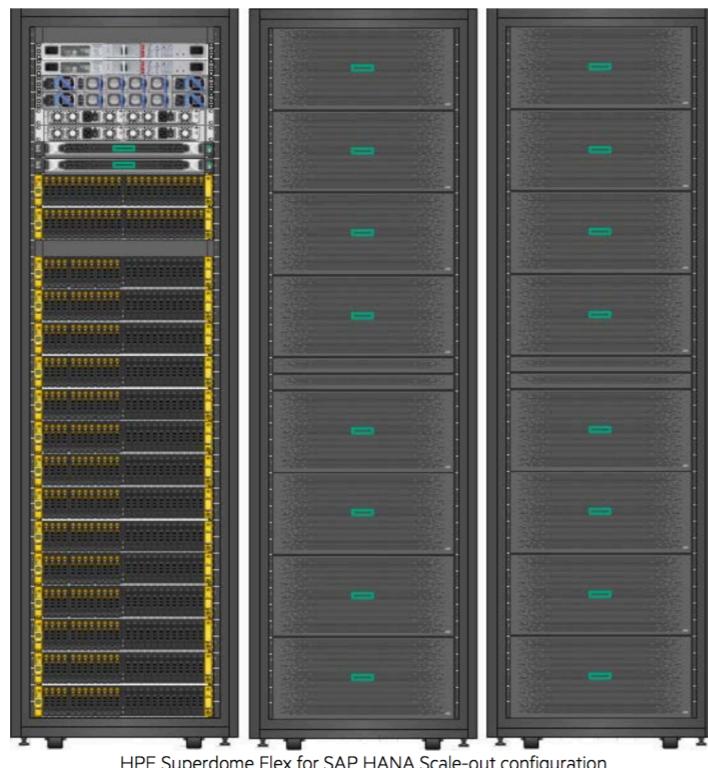
UCDAVIS
UNIVERSITY OF CALIFORNIA



Hardware Trends

Large core counts

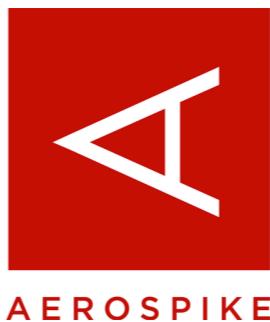
Large main-memory



HPE Superdome Server
144 physical cores
6TB of RAM

Popularity of Key-value Stores

- No multi-statement transactions
- Weak consistency
- Weak isolation

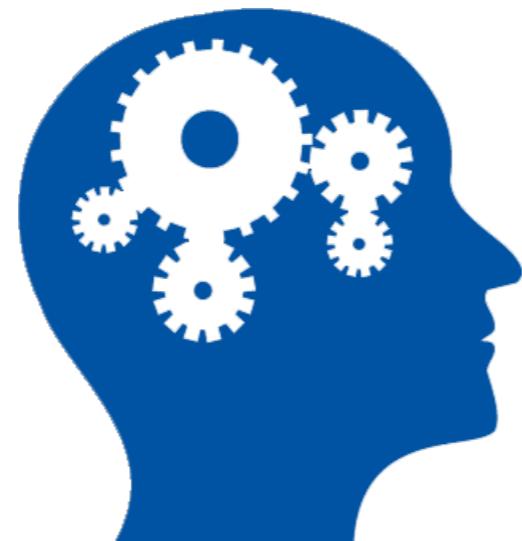


ORACLE®
NOSQL DATABASE



High-Contention Workloads

Challenge ???



High number of
contented operations

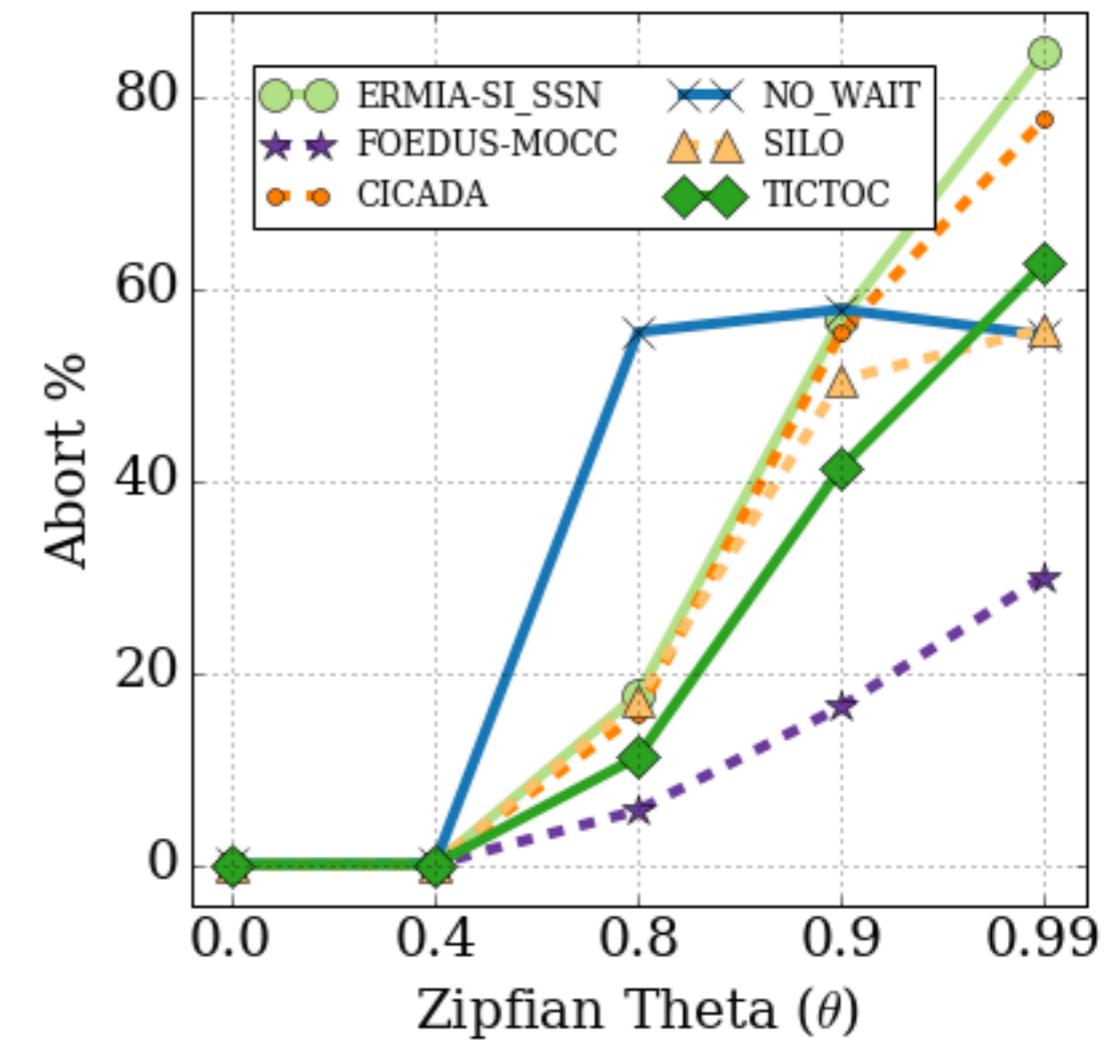
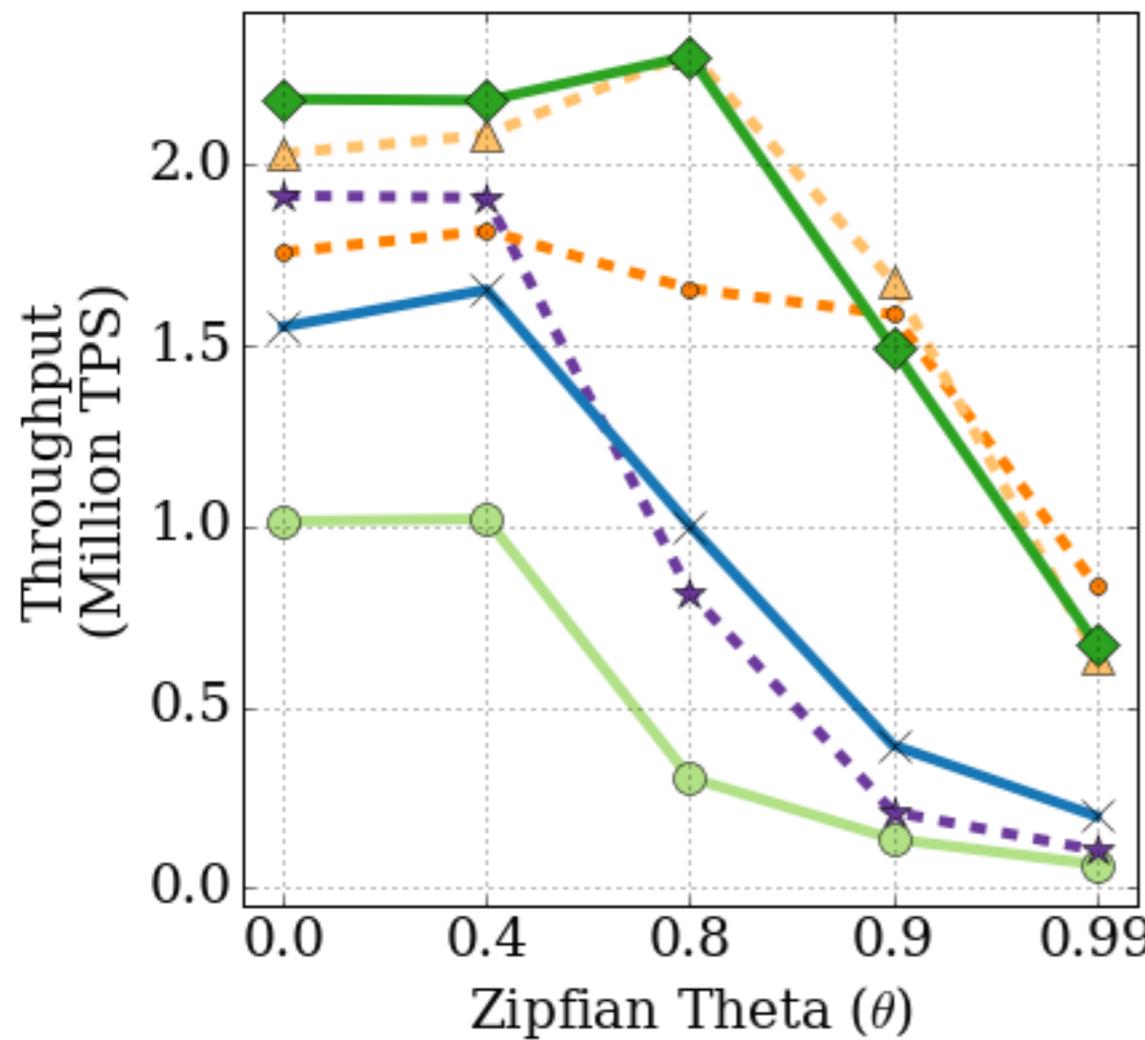


State-of-the-Art Concurrency Control Protocols

- Optimized for multi-core hardware and main-memory databases
- Non-deterministic

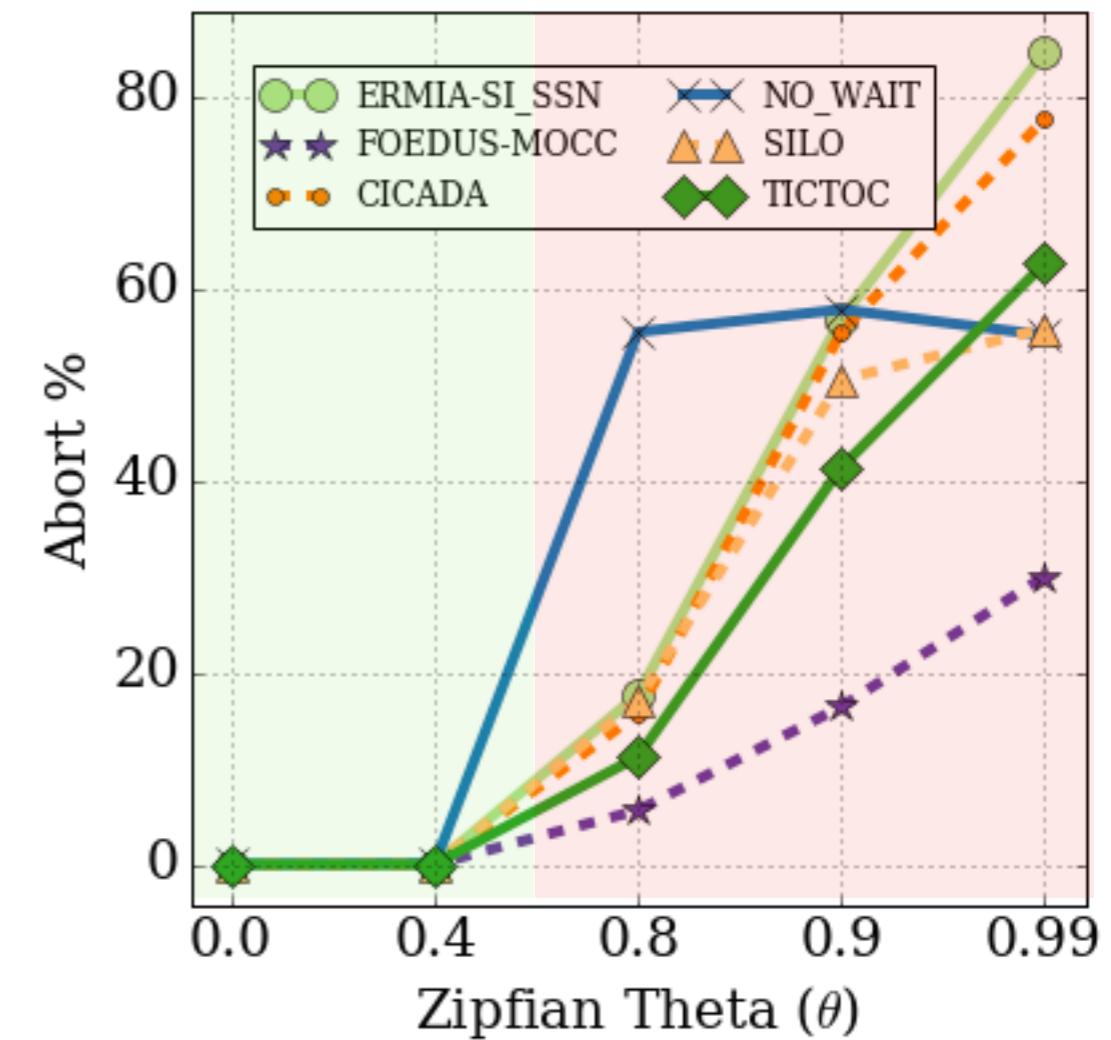
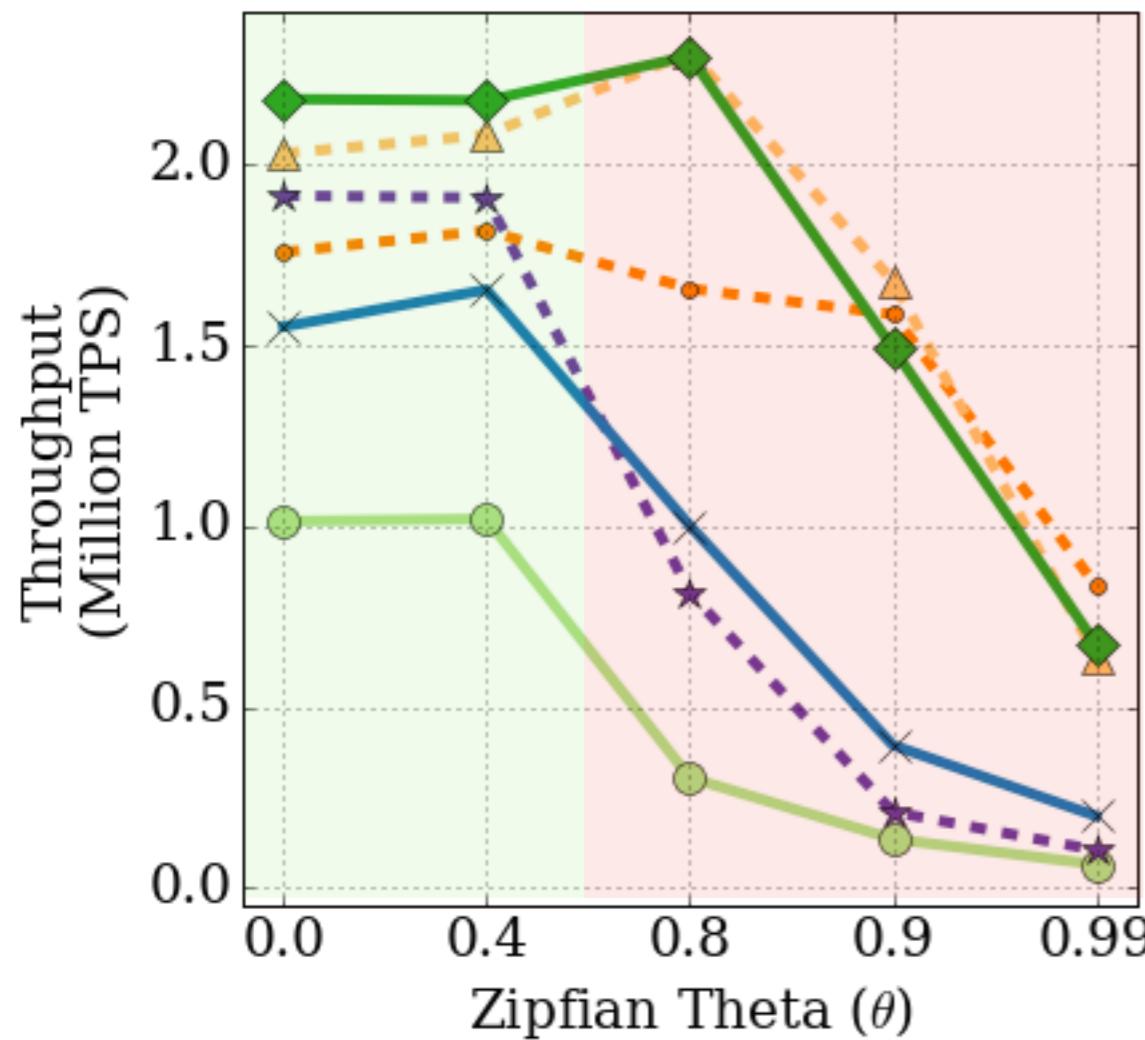
CC	Class	Year
SILO	Optimistic CC	SOSP '13
TICTOC	Timestamp Ordering	SIGMOD '16
FOEDUS-MOCC	Optimistic CC	VLDB '16
ERMIA	MVCC	SIGMOD '16
Cicada	MVCC	SIGMOD '17

Performance Under High-Contention



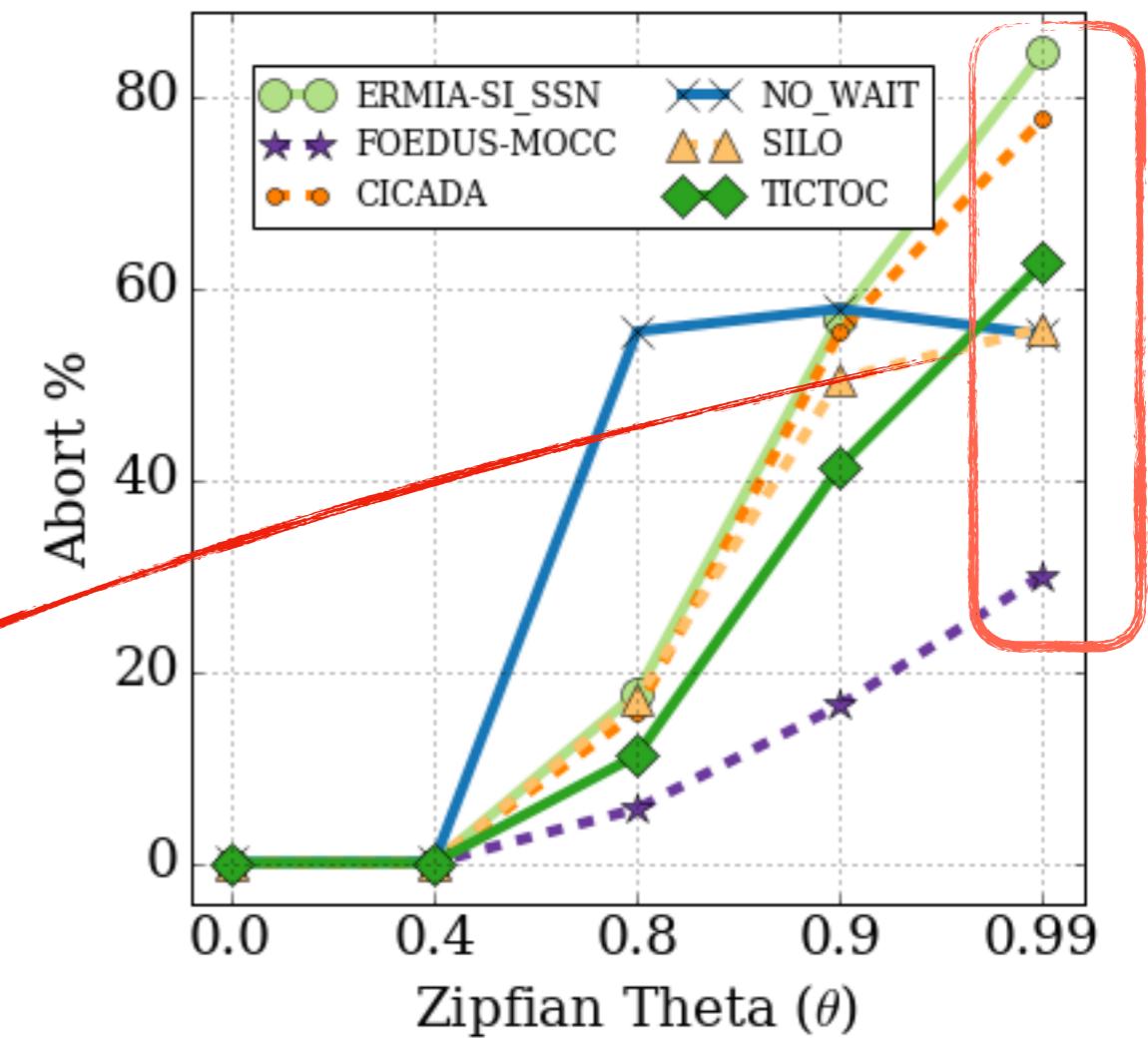
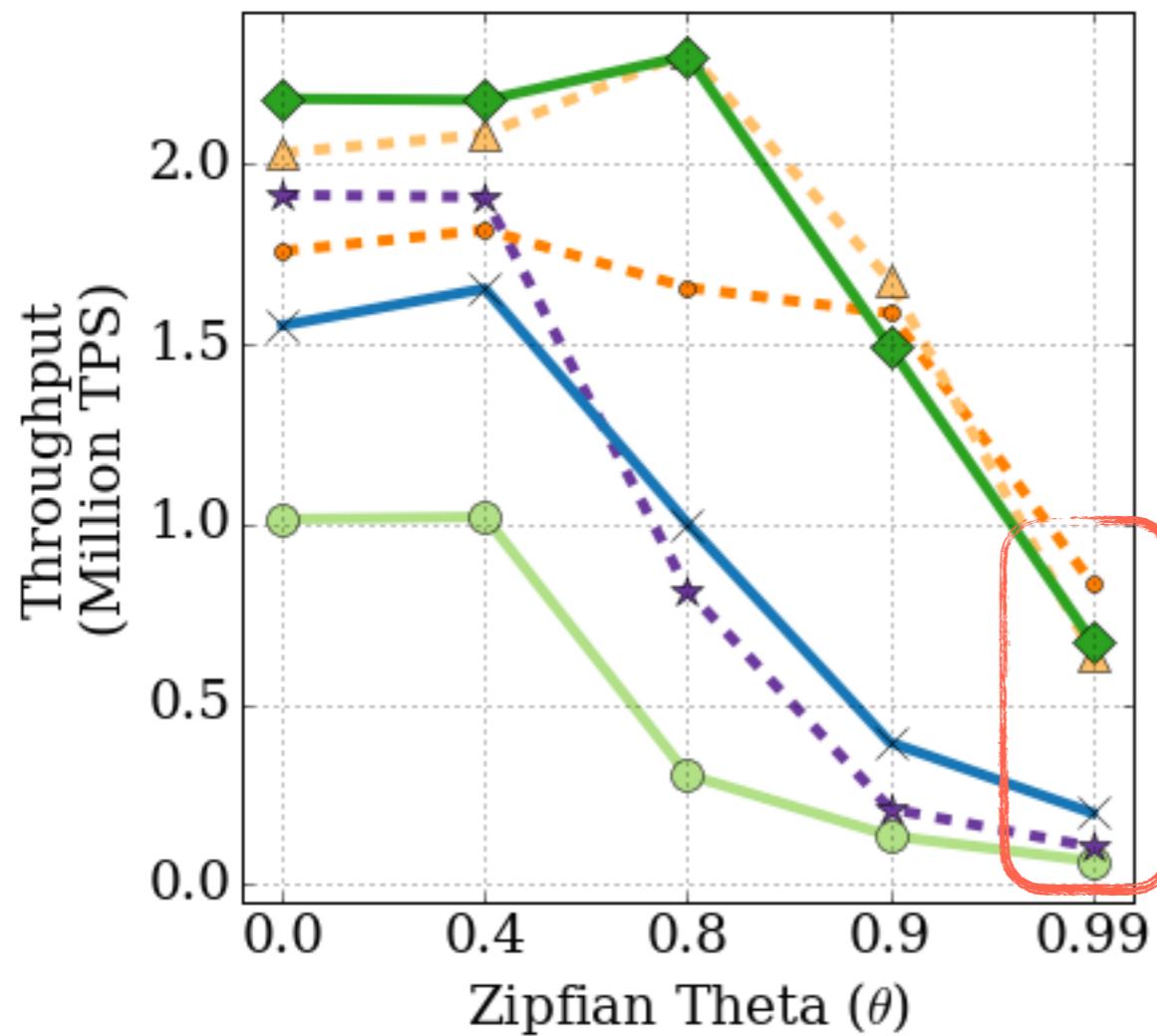
Optimize-for-multi-core concurrency control techniques suffer under high-contention due to increasing abort rate

Performance Under High-Contention



Under high-contention: Non-deterministic aborts dominates

Performance Under High-Contention



Under high-contention: Non-deterministic aborts dominates

2PL - NoWait

Abort Count: 0

Client Transactions

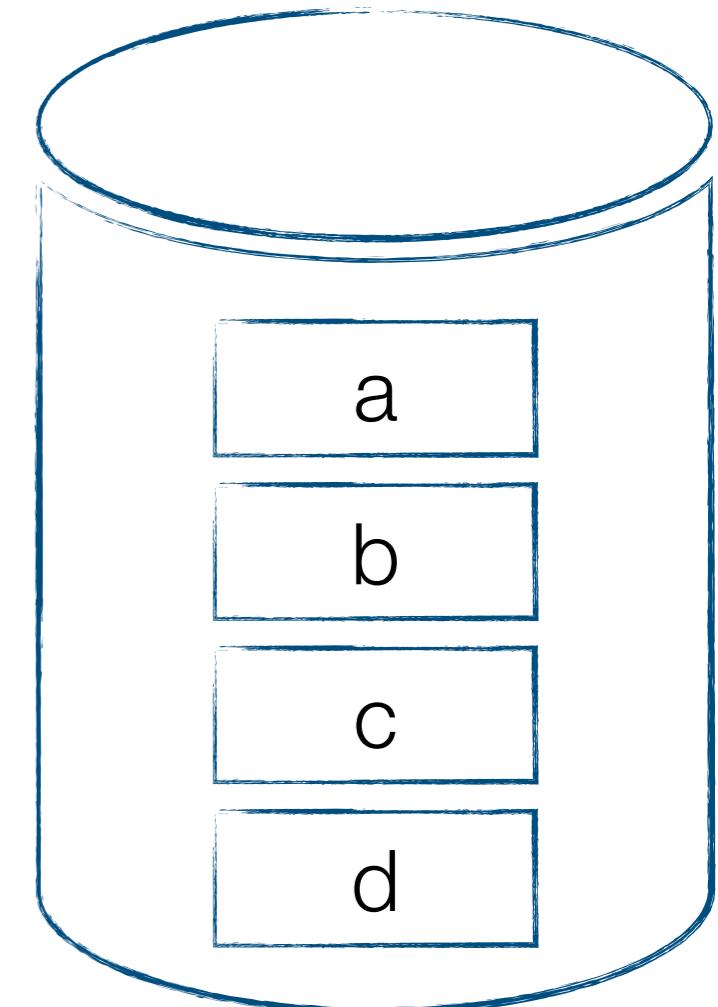
w ₄ (b)	w ₃ (b)	w ₂ (b)	r ₁ (a)
r ₄ (d)	r ₃ (c)	r ₂ (a)	w ₁ (b)

each color presents a transaction

Worker Thread #1



Worker Thread #2



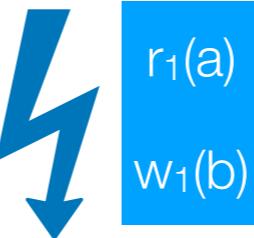
2PL - NoWait

Abort Count: 0

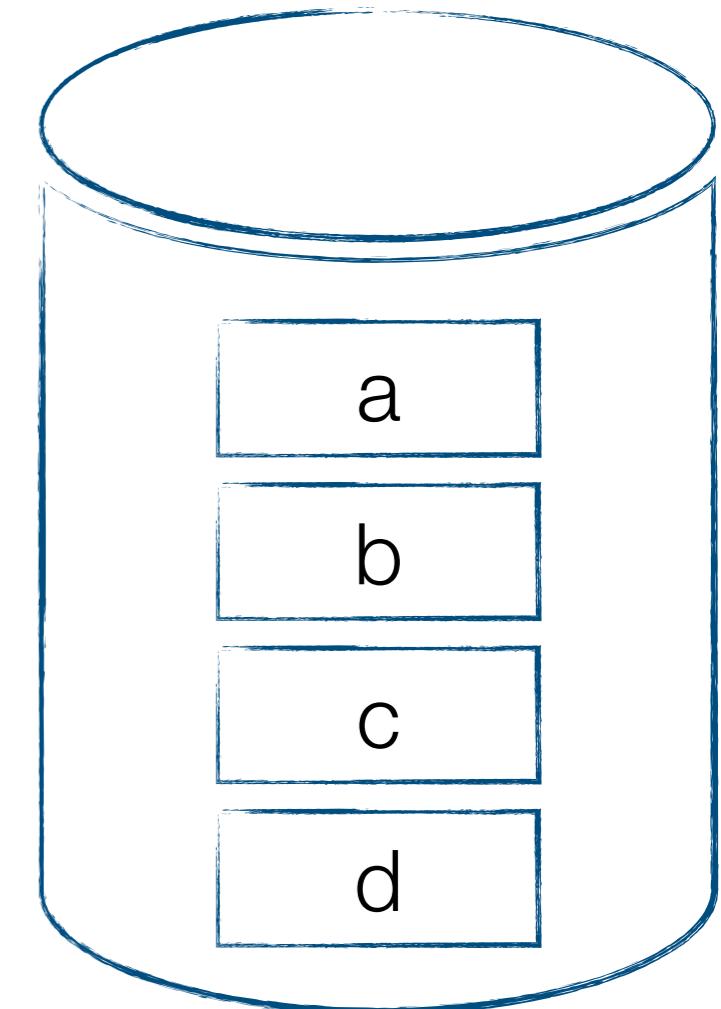
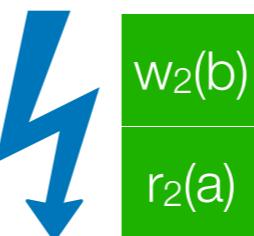
Client Transactions

w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

Worker Thread #1



Worker Thread #2



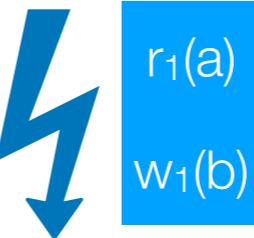
2PL - NoWait

Abort Count: 0

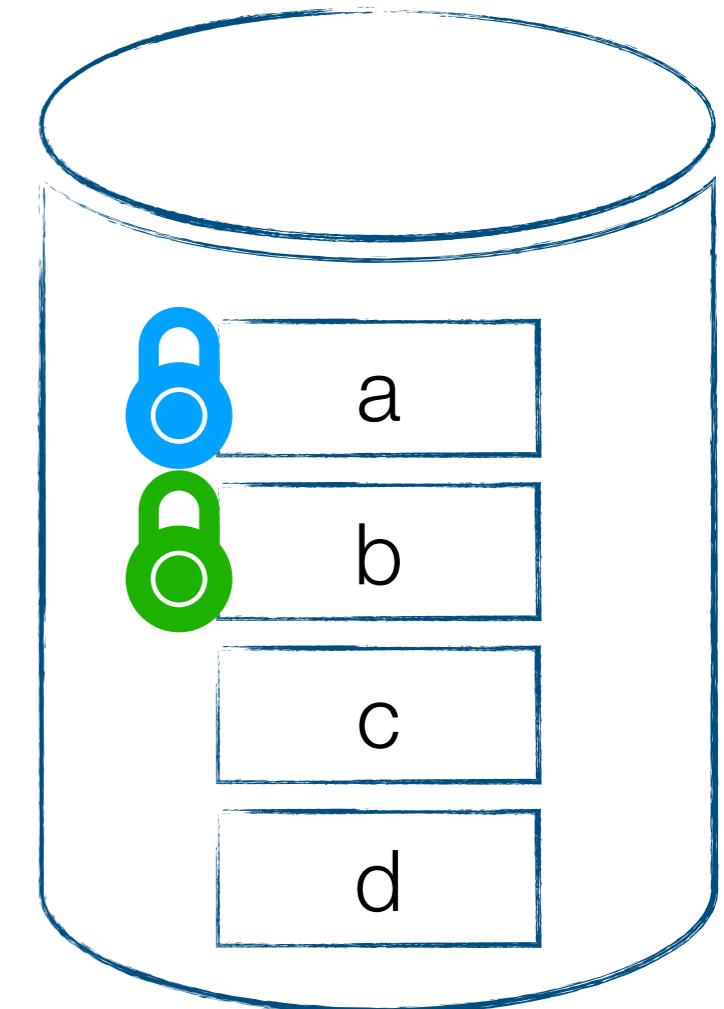
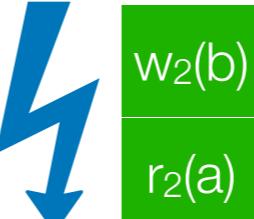
Client Transactions

w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

Worker
Thread #1



Worker
Thread #2



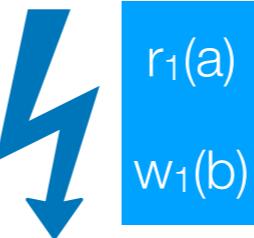
2PL - NoWait

Abort Count: 0

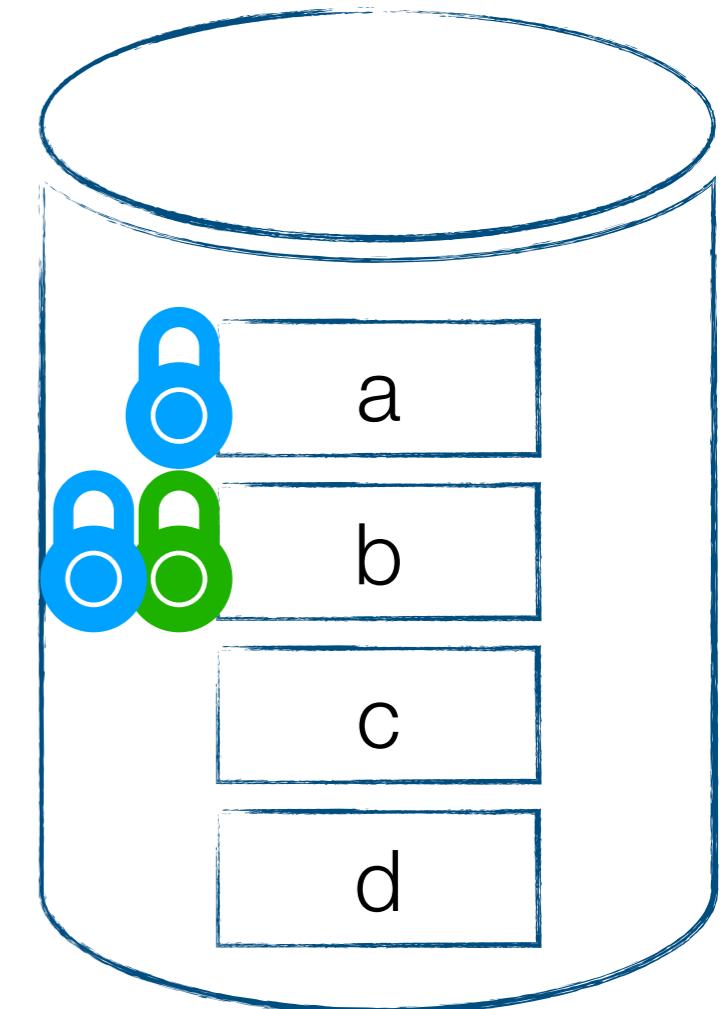
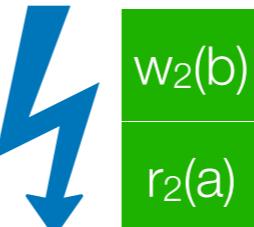
Client Transactions

w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

Worker Thread #1



Worker Thread #2



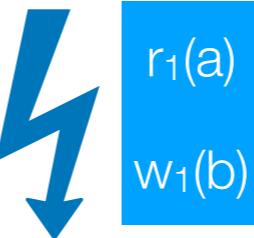
2PL - NoWait

Abort Count: 0

Client Transactions

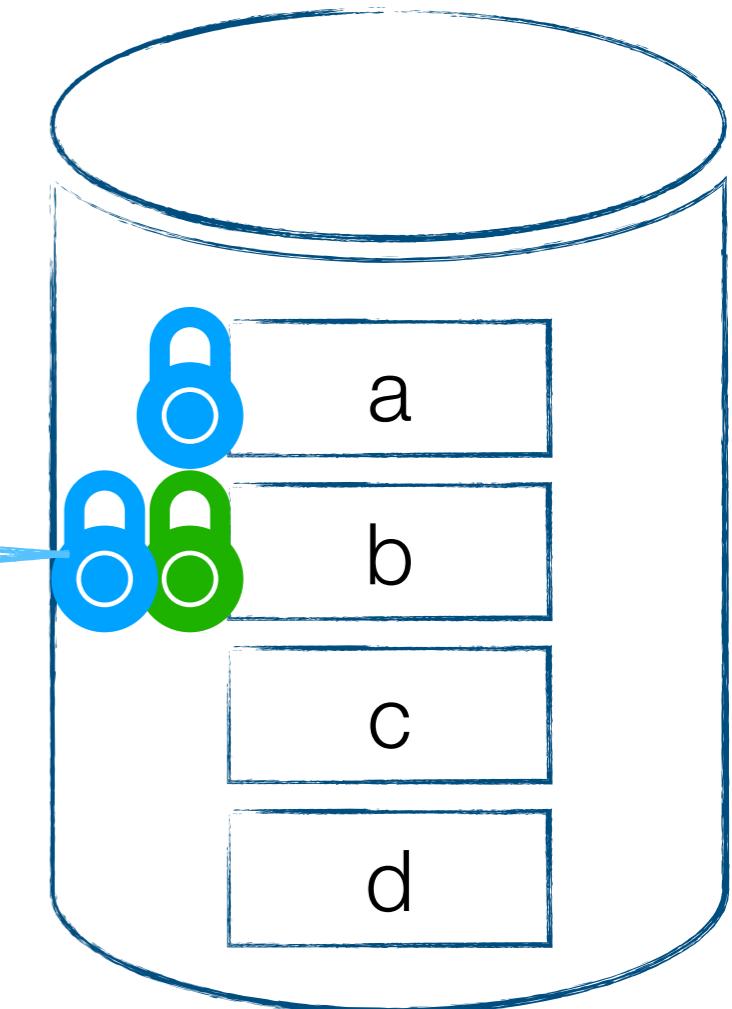
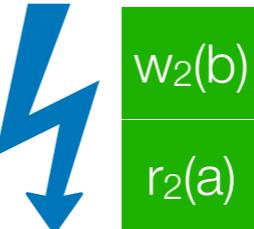
w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

Worker Thread #1



conflict!

Worker Thread #2



2PL - NoWait

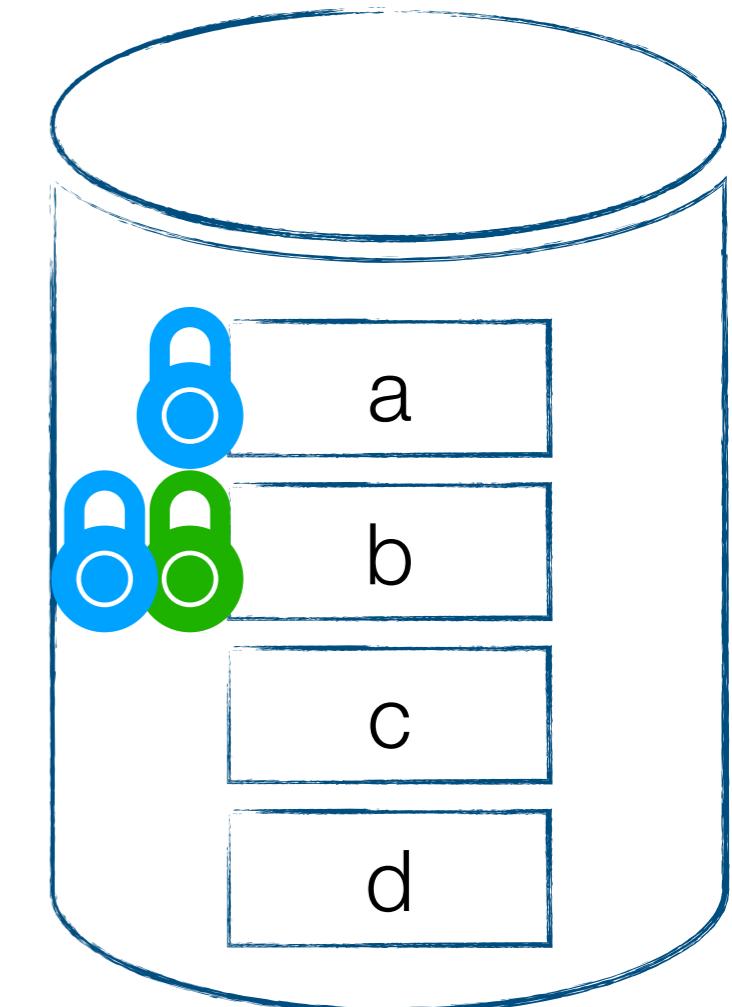
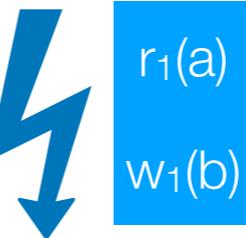
Abort Count: 0

Abort transaction (to avoid potential deadlocks)

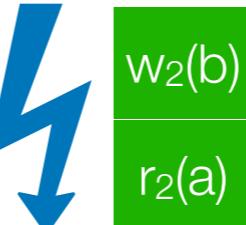
Client Transactions

w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

Worker
Thread #1



Worker
Thread #2



2PL - NoWait

Abort Count: 1

Client Transactions

w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

r₁(a)

w₁(b)

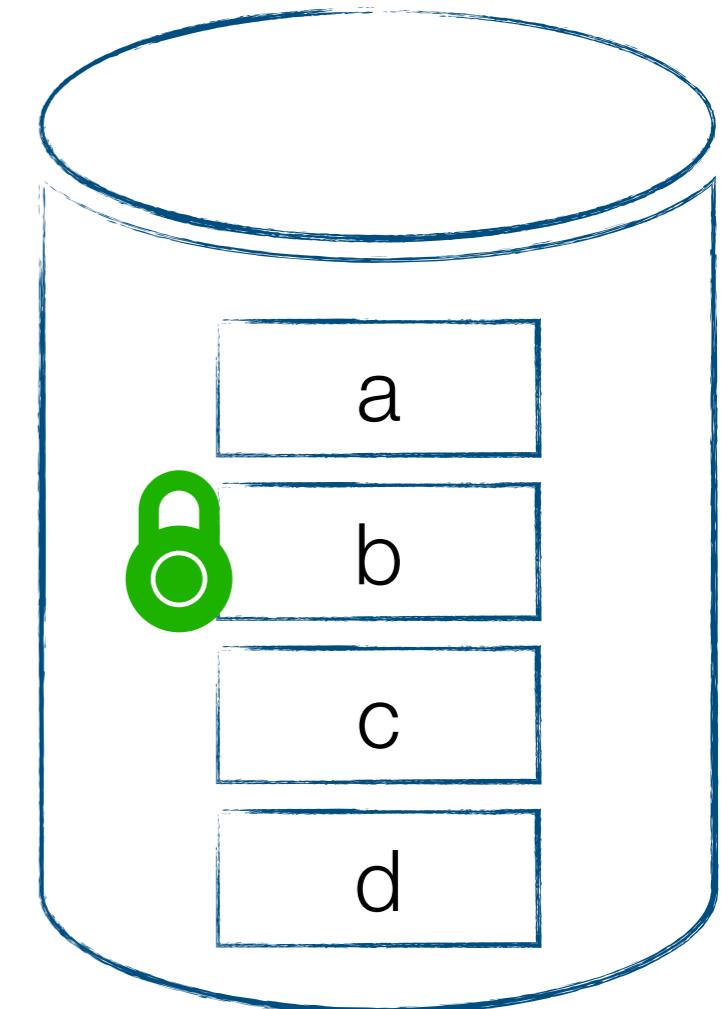
Worker
Thread #1



Worker
Thread #2



w ₂ (b)
r ₂ (a)



2PL - NoWait

Abort Count: 1

Client Transactions

w₄(b)
r₄(d)

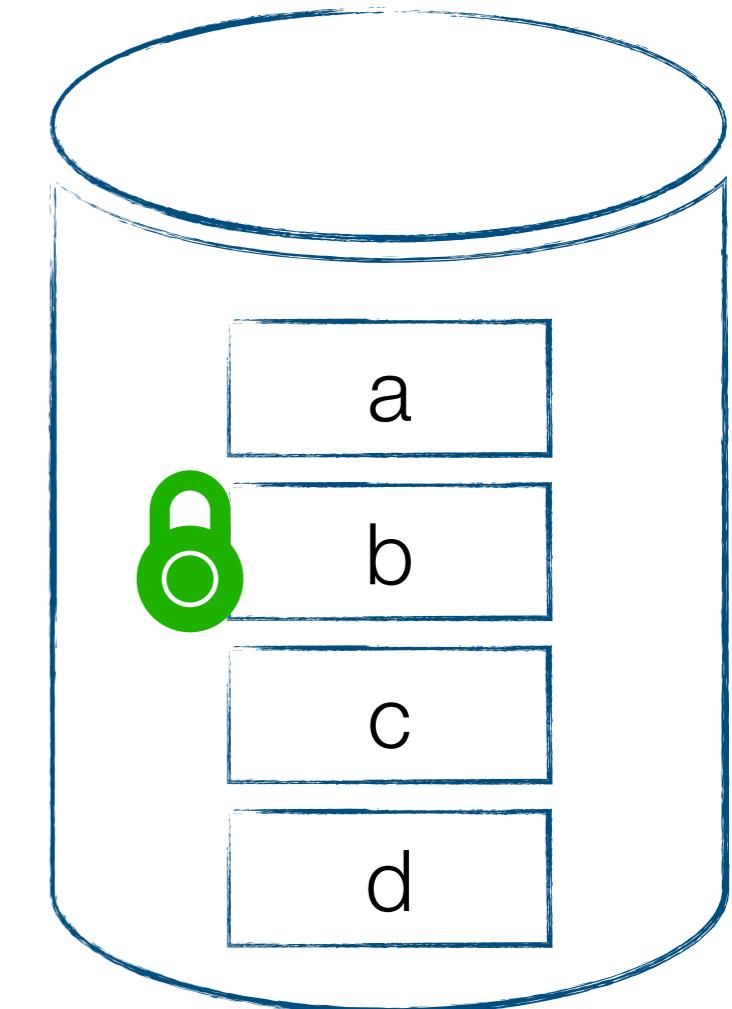
r₁(a)
w₁(b)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

w₂(b)
r₂(a)



2PL - NoWait

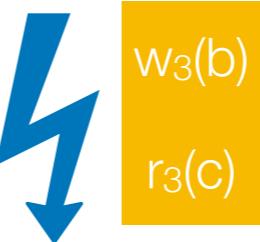
Abort Count: 1

Client Transactions

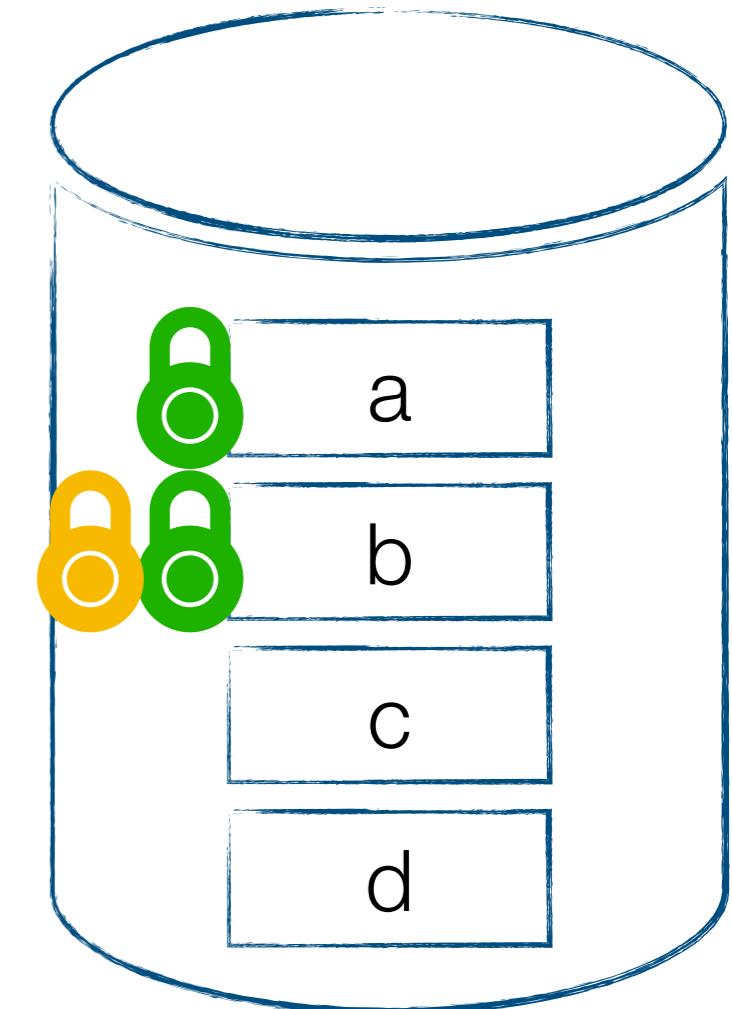
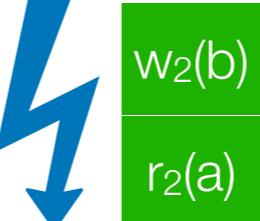
w₄(b)
r₄(d)

r₁(a)
w₁(b)

Worker
Thread #1



Worker
Thread #2



2PL - NoWait

Abort Count: 1

Client Transactions

w₄(b)
r₄(d)

r₁(a)
w₁(b)

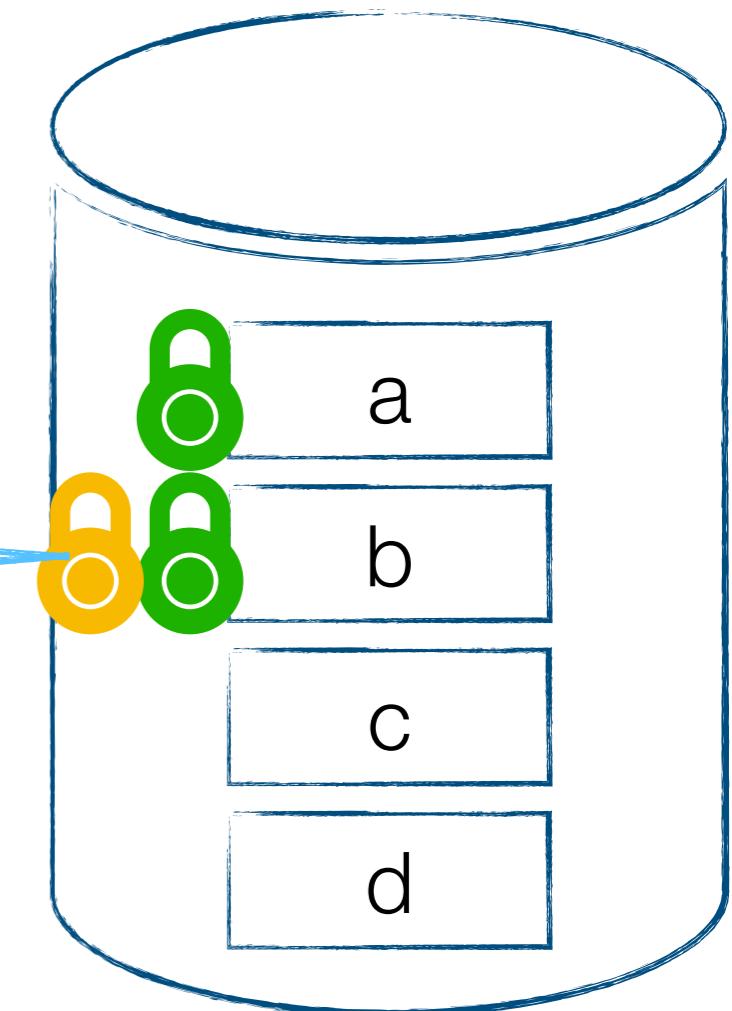
Worker
Thread #1

w₃(b)
r₃(c)

conflict!

Worker
Thread #2

w₂(b)
r₂(a)

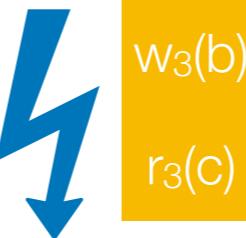


2PL - NoWait

Abort Count: 1

Abort transaction (to avoid potential deadlocks)

Worker
Thread #1

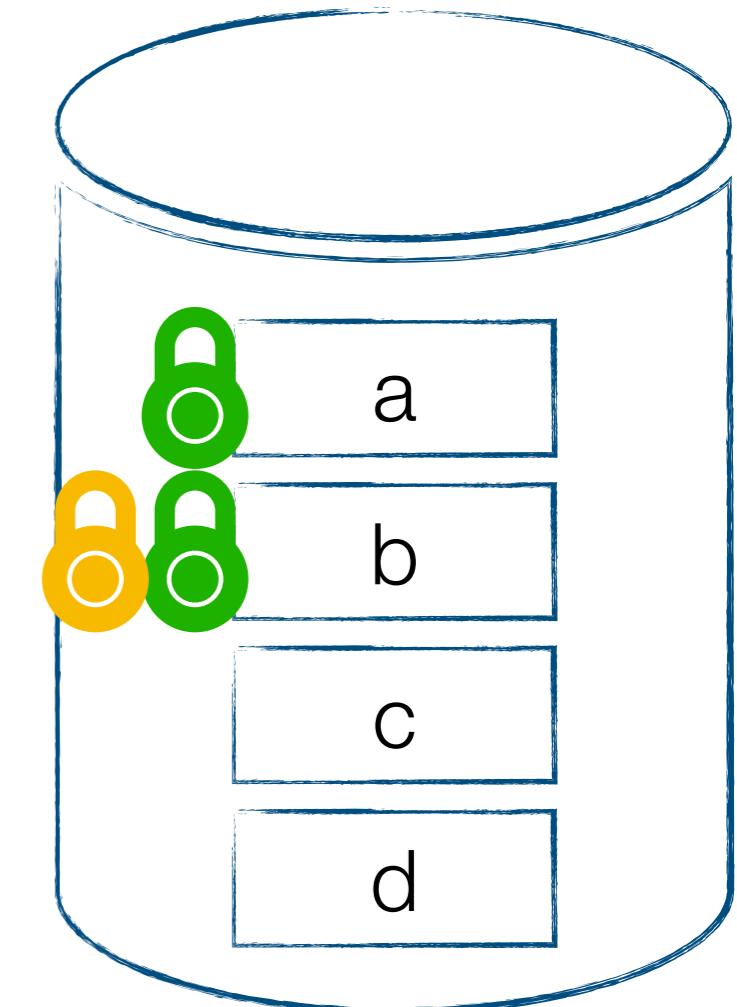
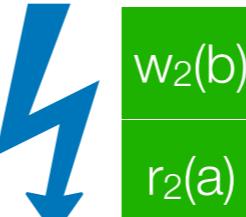


Client Transactions

w₄(b)
r₄(d)

r₁(a)
w₁(b)

Worker
Thread #2



2PL - NoWait

Abort Count: 2

Client Transactions

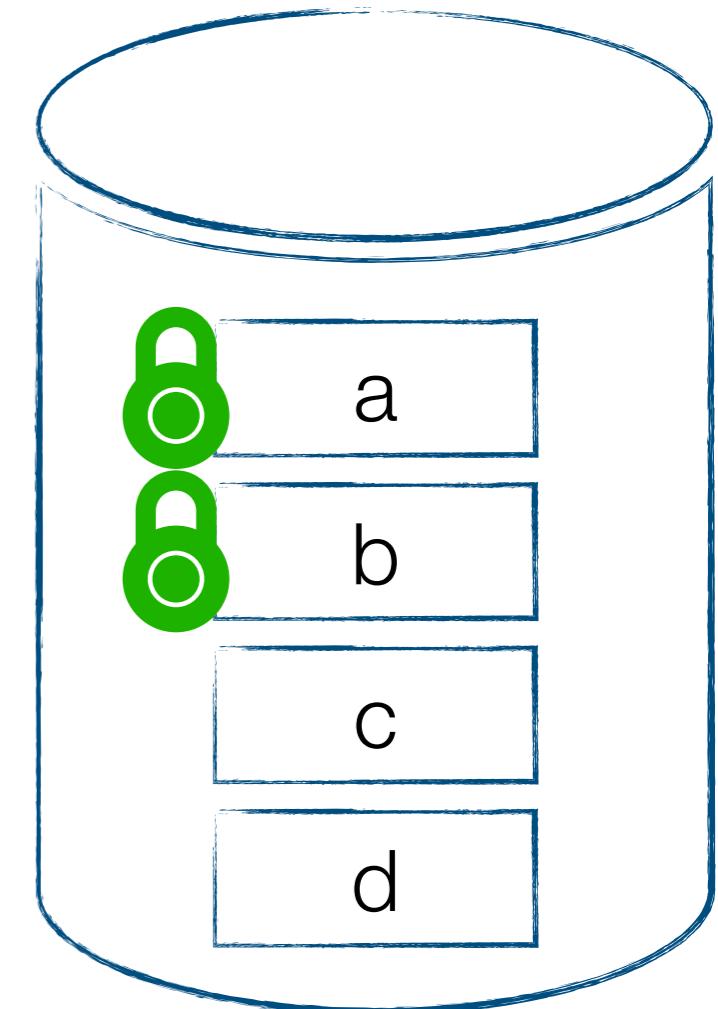
w ₄ (b)	w ₃ (b)
r ₄ (d)	r ₃ (c)

r ₁ (a)
w ₁ (b)

Worker Thread #1



Worker Thread #2



2PL - NoWait

Abort Count: 2

Client Transactions

w₃(b)
r₃(c)

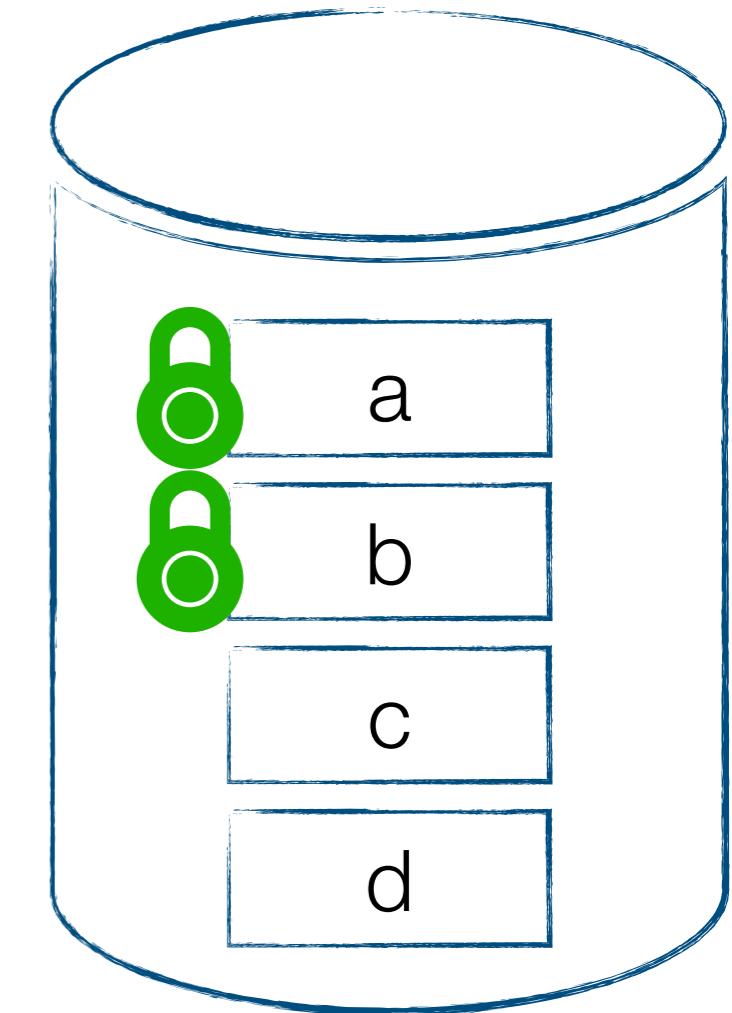
r₁(a)
w₁(b)

Worker
Thread #1

w₄(b)
r₄(d)

Worker
Thread #2

w₂(b)
r₂(a)



2PL - NoWait

Abort Count: 2

Client Transactions

w₃(b)
r₃(c)

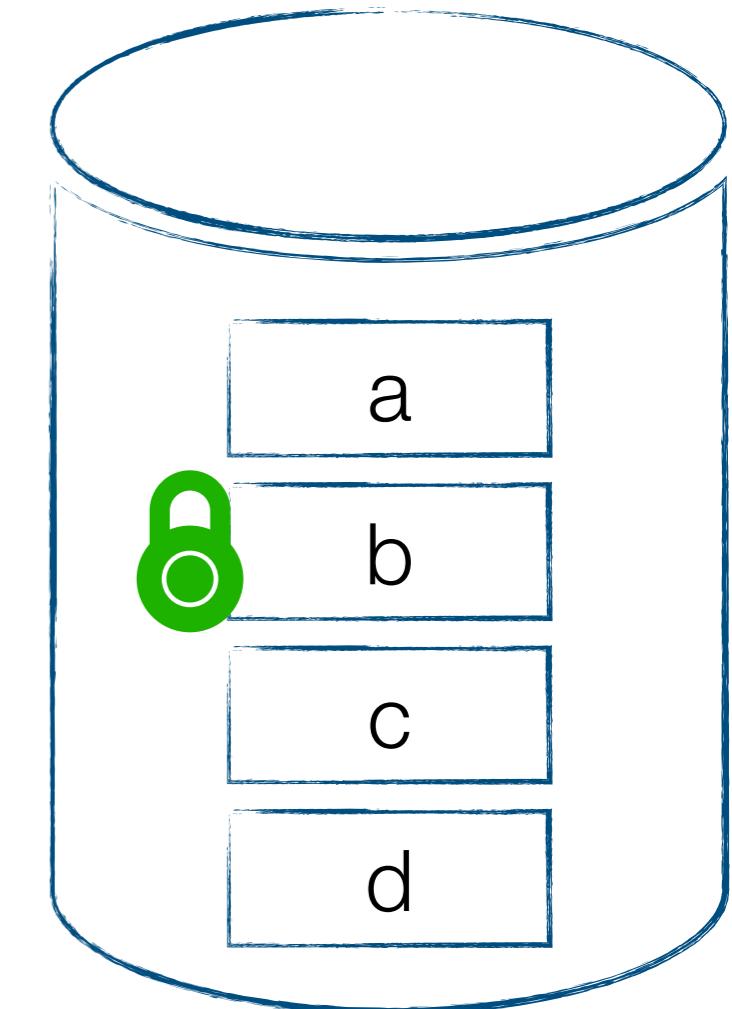
r₁(a)
w₁(b)

Worker
Thread #1

w₄(b)
r₄(d)

Worker
Thread #2

w₂(b)
r₂(a)



2PL - NoWait

Abort Count: 2

Client Transactions

w₃(b)
r₃(c)

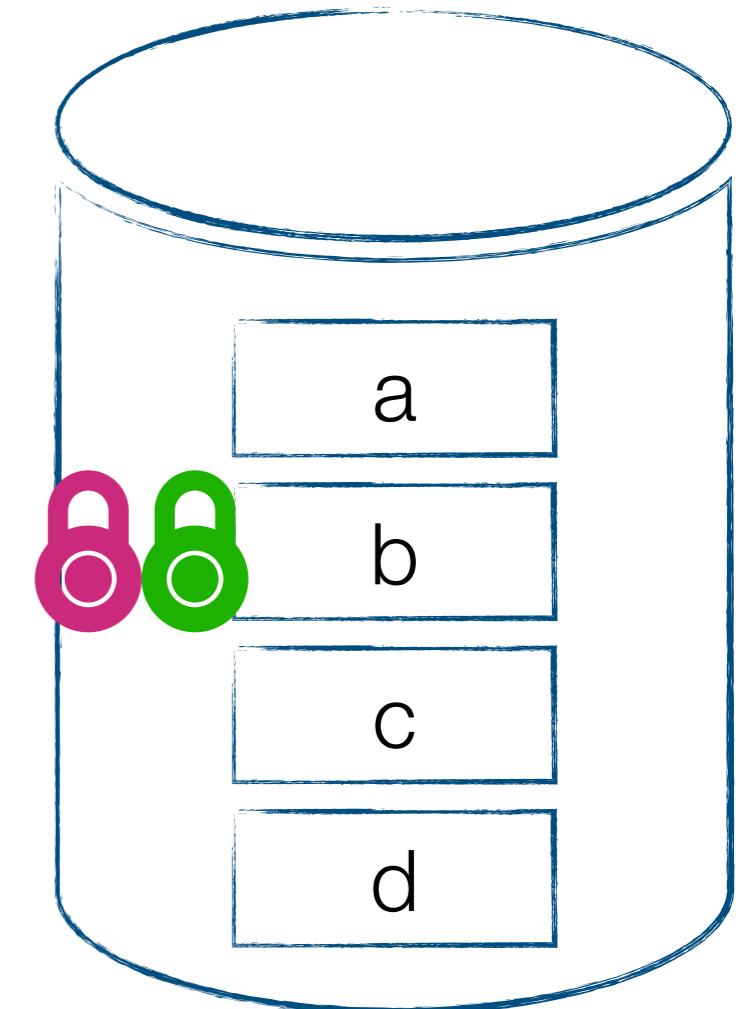
r₁(a)
w₁(b)

Worker
Thread #1

w₄(b)
r₄(d)

Worker
Thread #2

w₂(b)
r₂(a)



2PL - NoWait

Abort Count: 2

Client Transactions

w₃(b)
r₃(c)

r₁(a)
w₁(b)

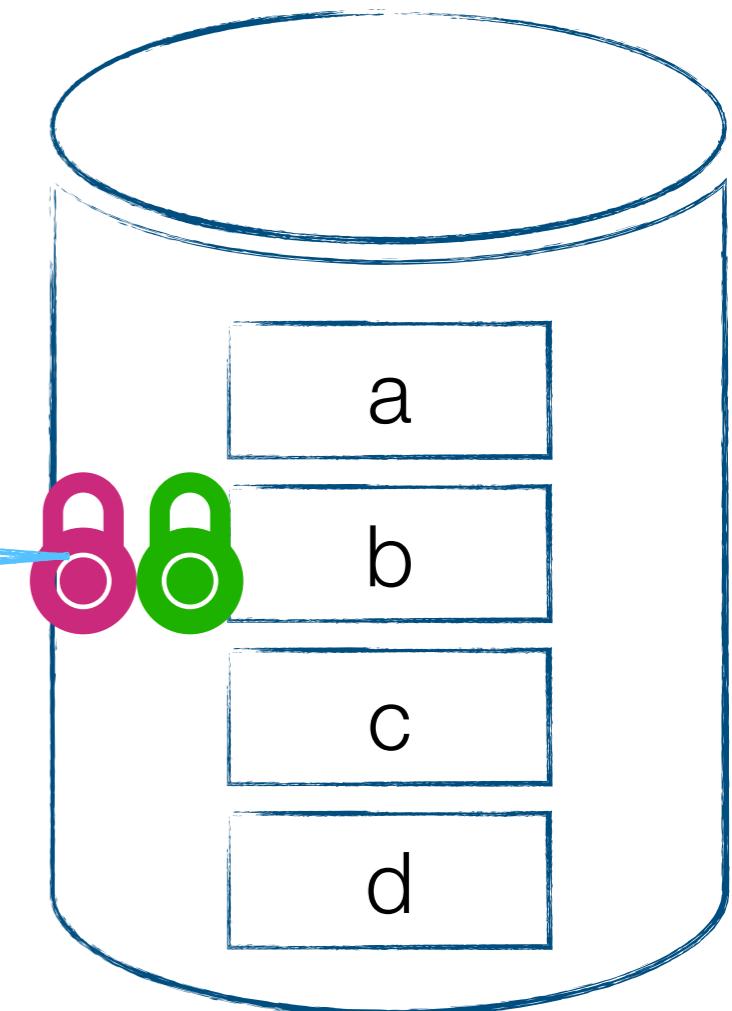
Worker
Thread #1

w₄(b)
r₄(d)

conflict!

Worker
Thread #2

w₂(b)
r₂(a)



2PL - NoWait

Abort Count: 2

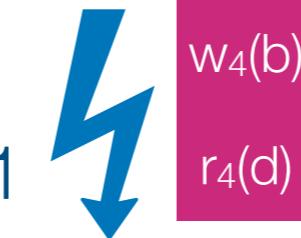
Abort transaction (to avoid potential deadlocks)

Client Transactions

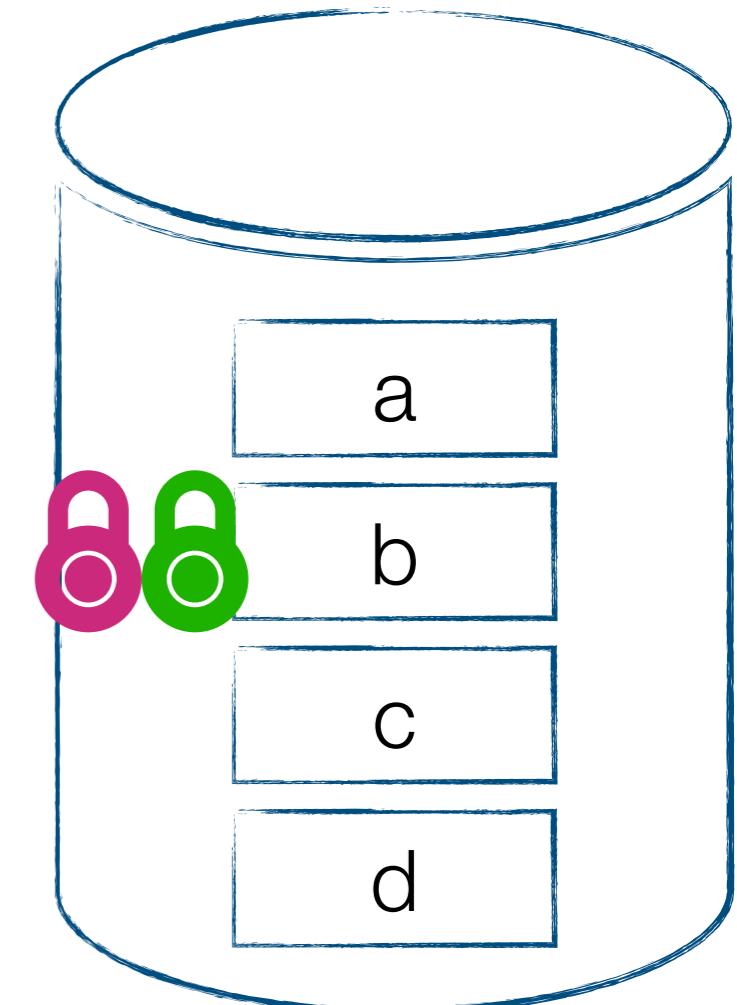
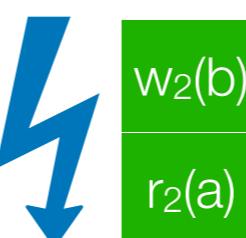
w₃(b)
r₃(c)

r₁(a)
w₁(b)

Worker
Thread #1



Worker
Thread #2



2PL - NoWait

Abort Count: 3

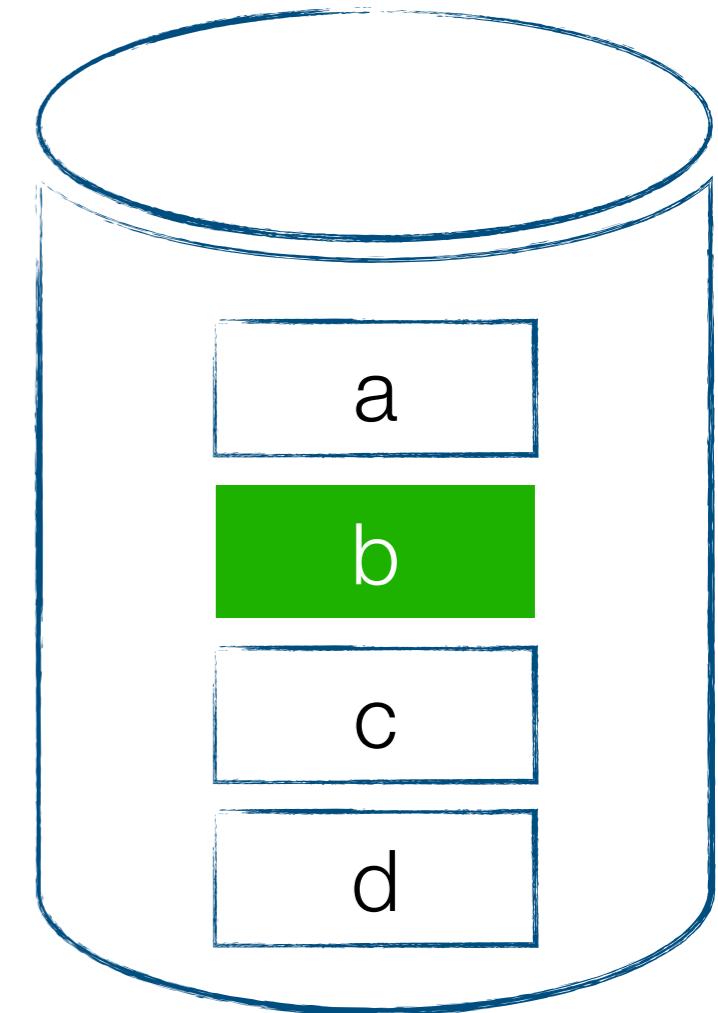
Client Transactions

w ₄ (b)	w ₃ (b)	r ₁ (a)
r ₄ (d)	r ₃ (c)	w ₁ (b)

Worker
Thread #1



Worker
Thread #2



Committed Transactions

w ₂ (b)
r ₂ (a)

2PL - NoWait

Abort Count: 3

Client Transactions

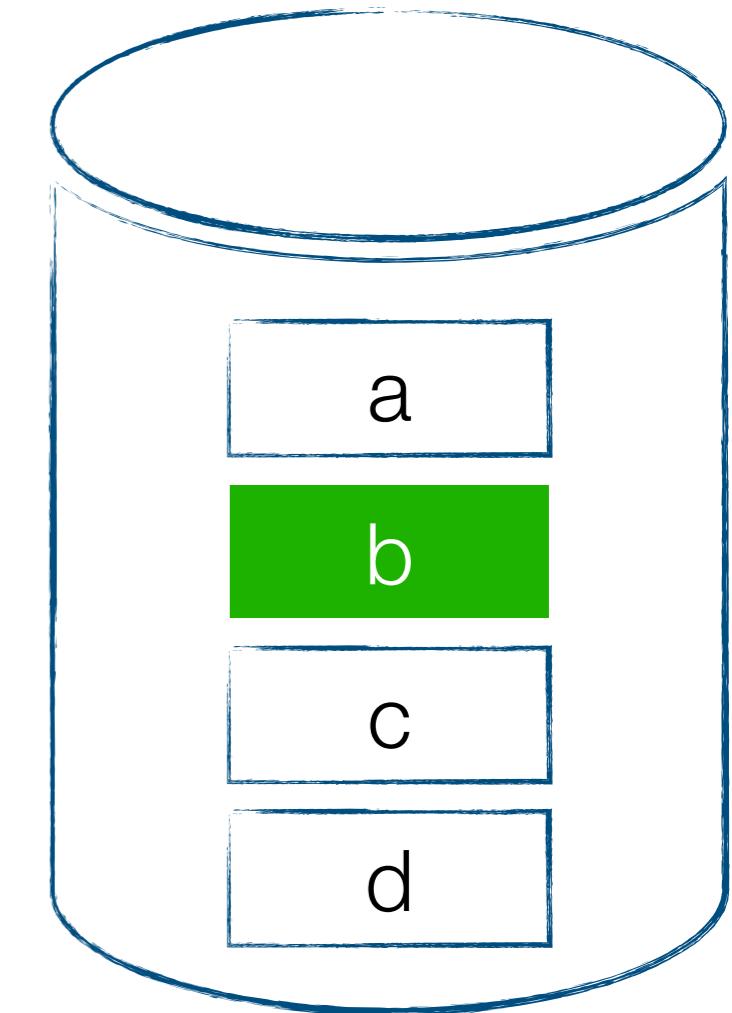
r₁(a)
w₁(b)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

w₄(b)
r₄(d)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 3

Client Transactions

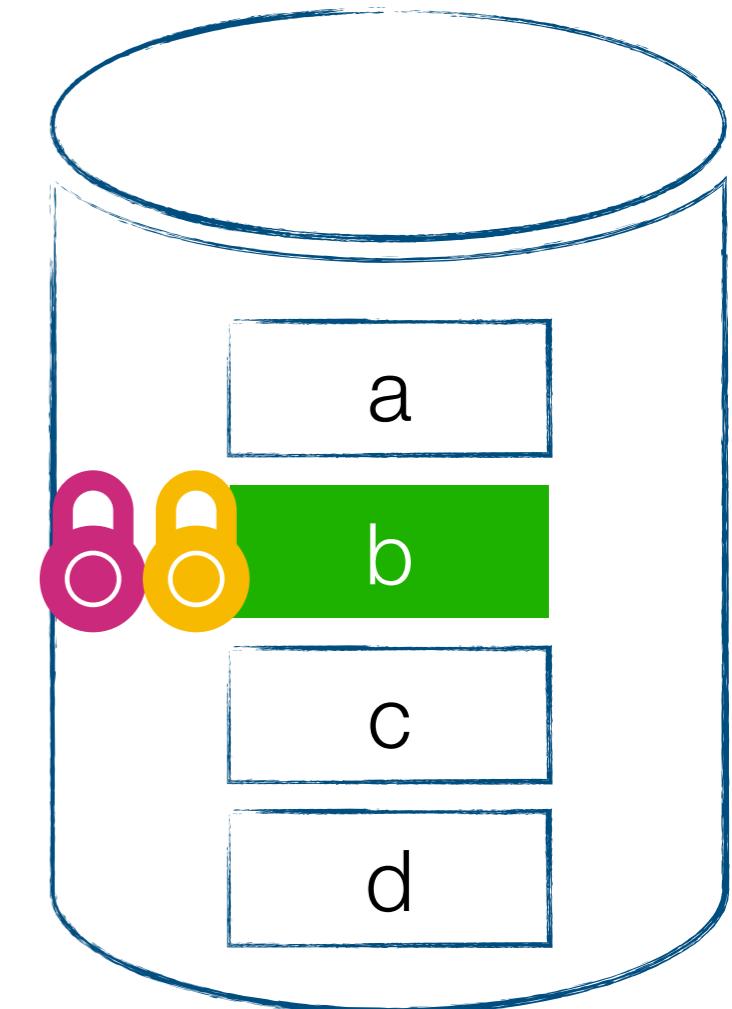
r₁(a)
w₁(b)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

w₄(b)
r₄(d)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 3

Client Transactions

r₁(a)
w₁(b)

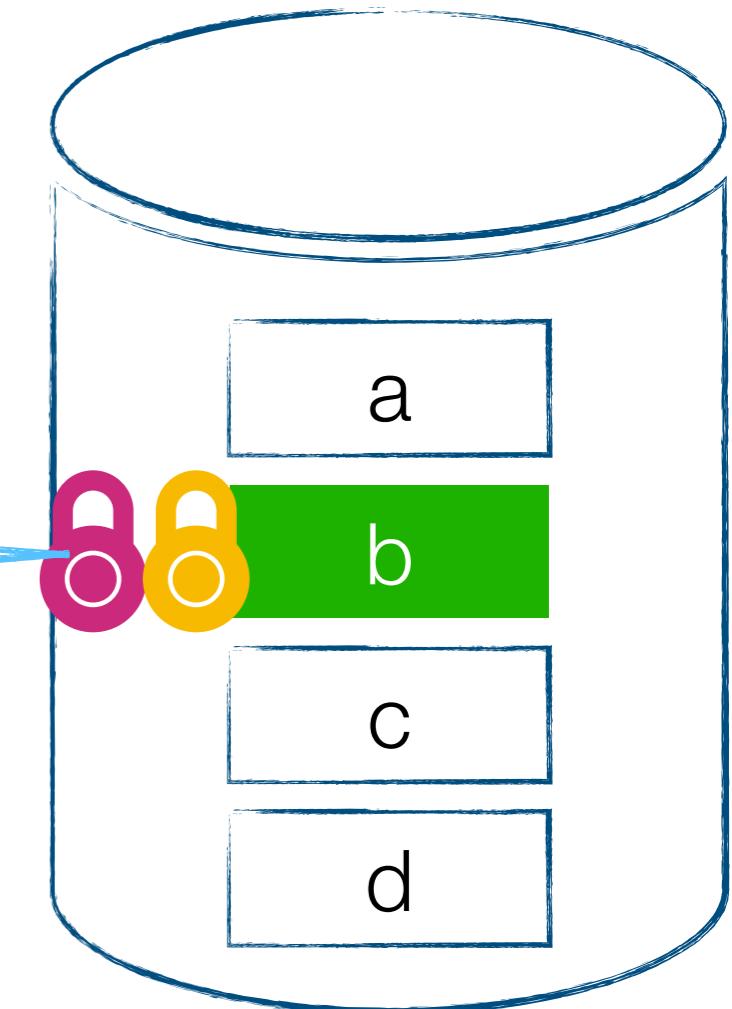
Worker
Thread #1

w₃(b)
r₃(c)

conflict!

Worker
Thread #2

w₄(b)
r₄(d)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 3

Client Transactions

r₁(a)
w₁(b)

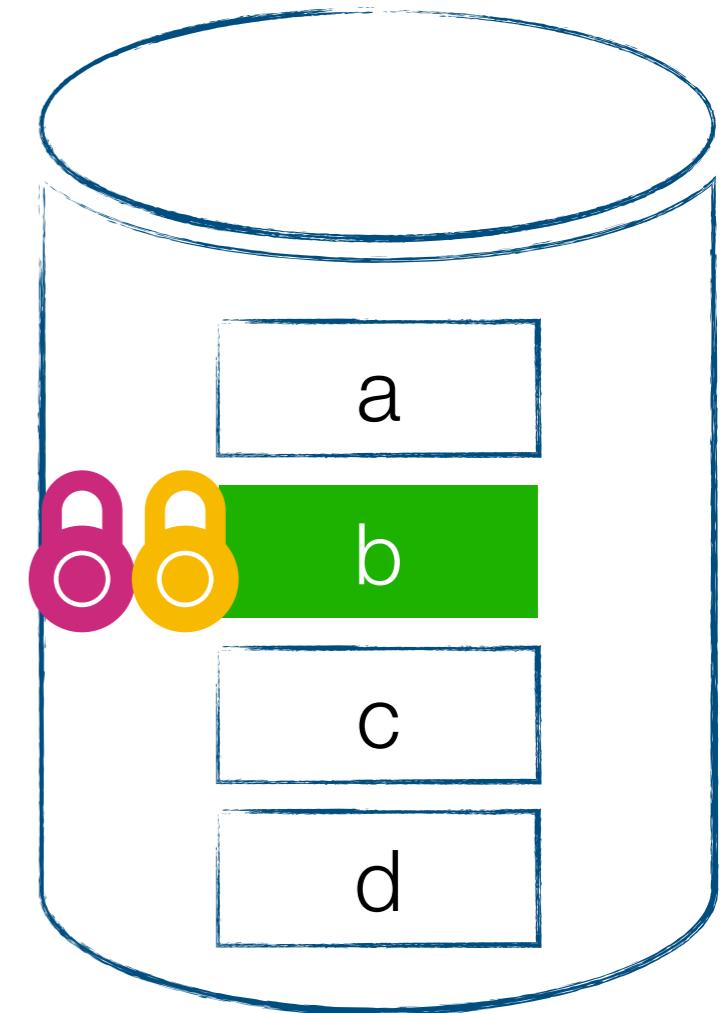
Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

w₄(b)
r₄(d)

Abort transaction (to avoid potential deadlocks)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

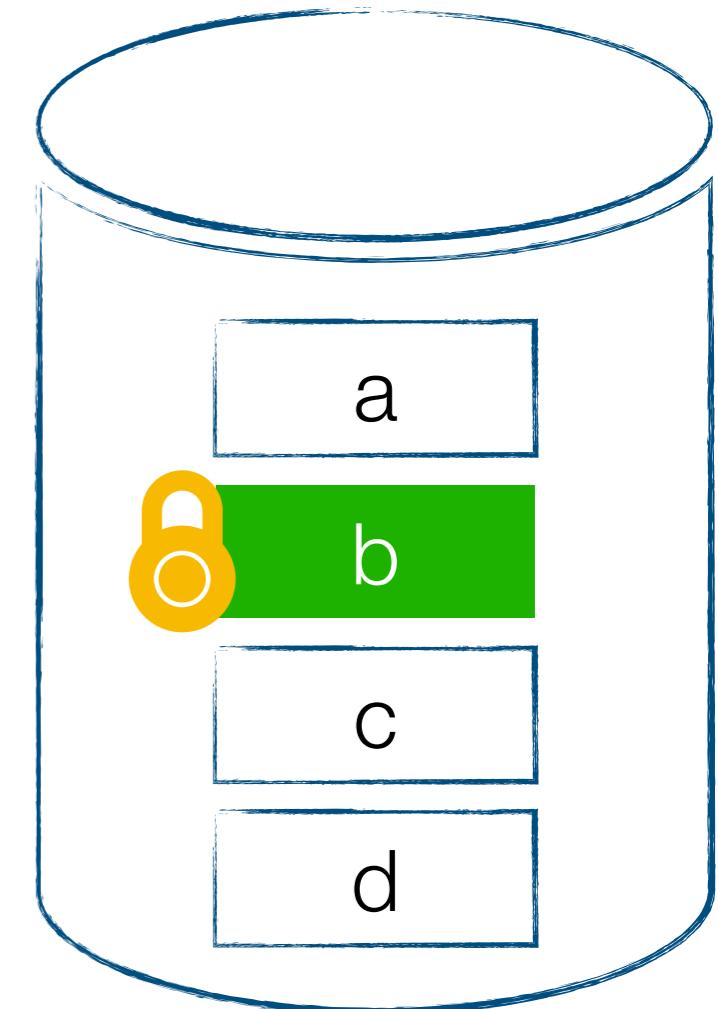
w₄(b)
r₄(d)

r₁(a)
w₁(b)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

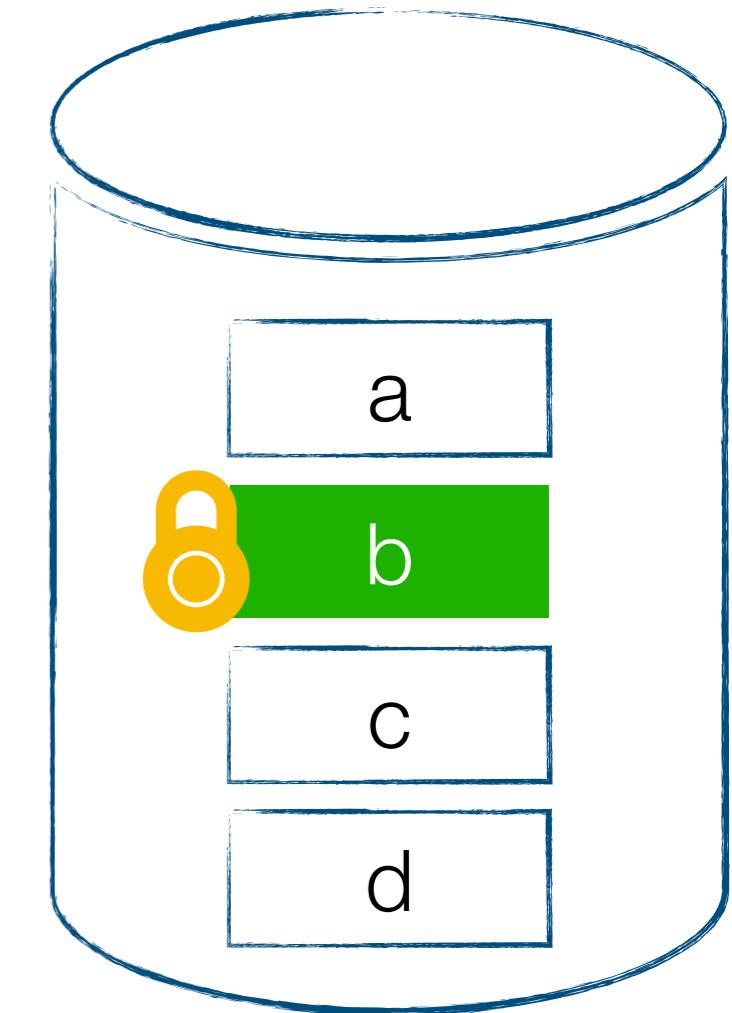
w₄(b)
r₄(d)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

r₁(a)
w₁(b)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

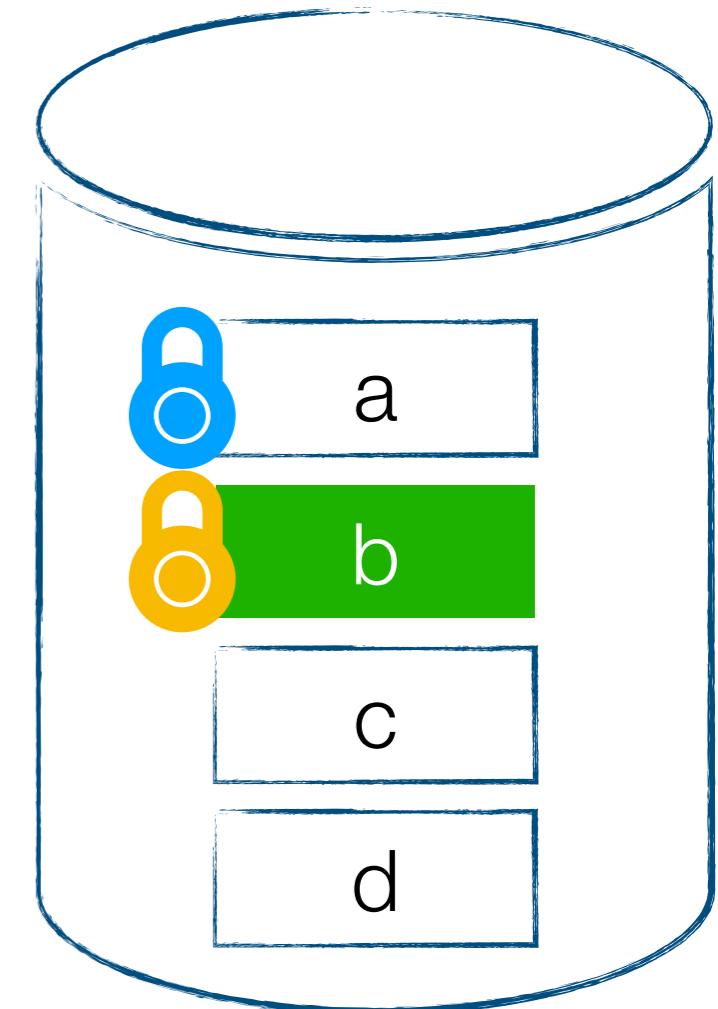
w₄(b)
r₄(d)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

r₁(a)
w₁(b)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

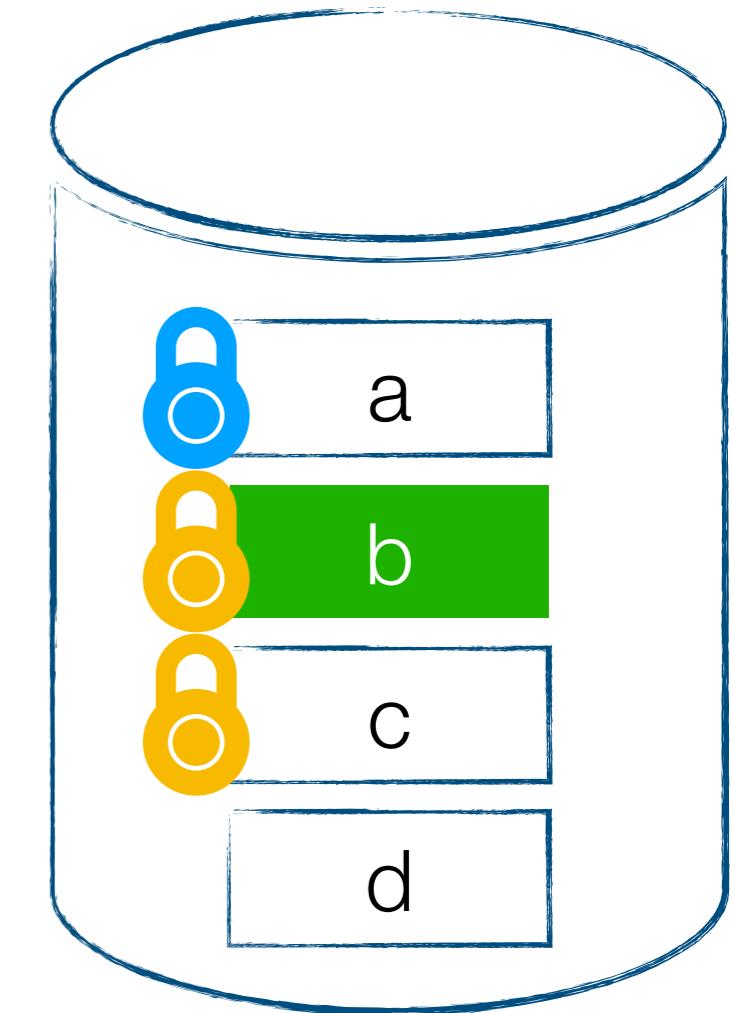
w₄(b)
—
r₄(d)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

r₁(a)
w₁(b)



Committed Transactions

w₂(b)
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

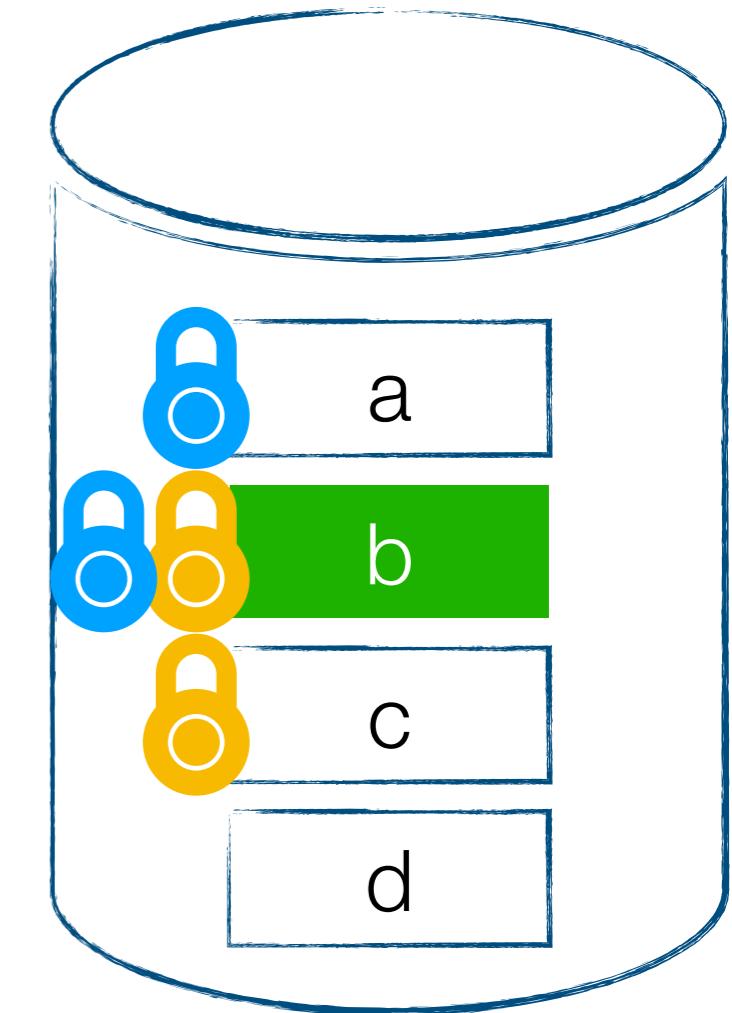
w₄(b)
—
r₄(d)

Worker
Thread #1

w₃(b)
r₃(c)

Worker
Thread #2

r₁(a)
w₁(b)



Committed Transactions

w₂(b)
—
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

w₄(b)
—
r₄(d)

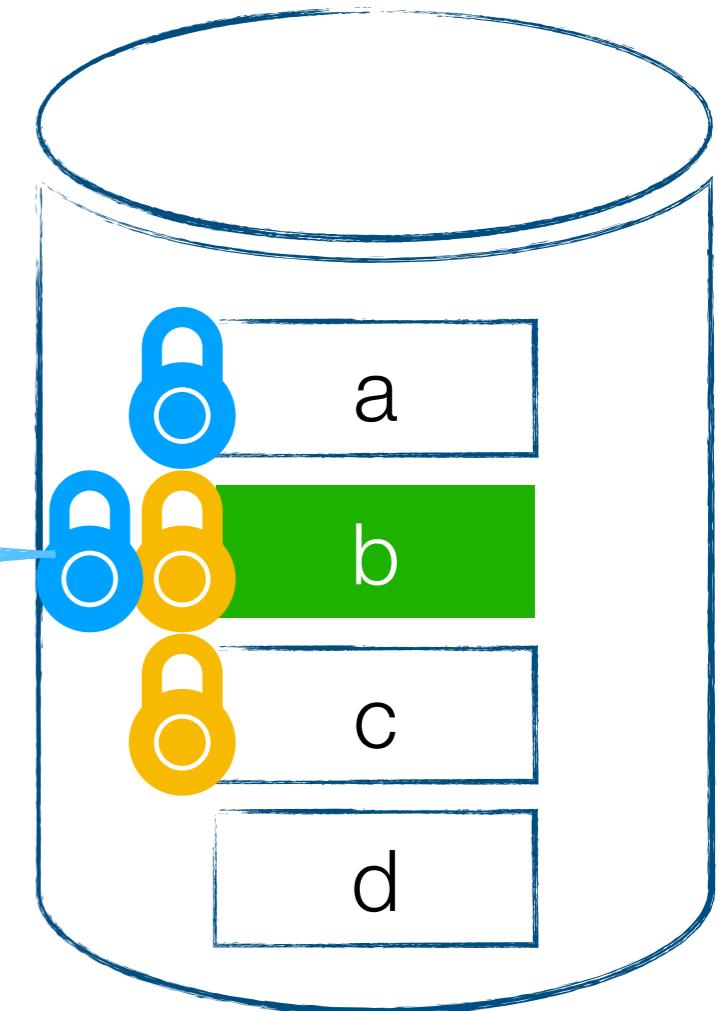
Worker
Thread #1

w₃(b)
r₃(c)

conflict!

Worker
Thread #2

r₁(a)
w₁(b)



Committed Transactions

w₂(b)
—
r₂(a)

2PL - NoWait

Abort Count: 4

Client Transactions

w₄(b)
—
r₄(d)

Worker
Thread #1

w₃(b)
r₃(c)

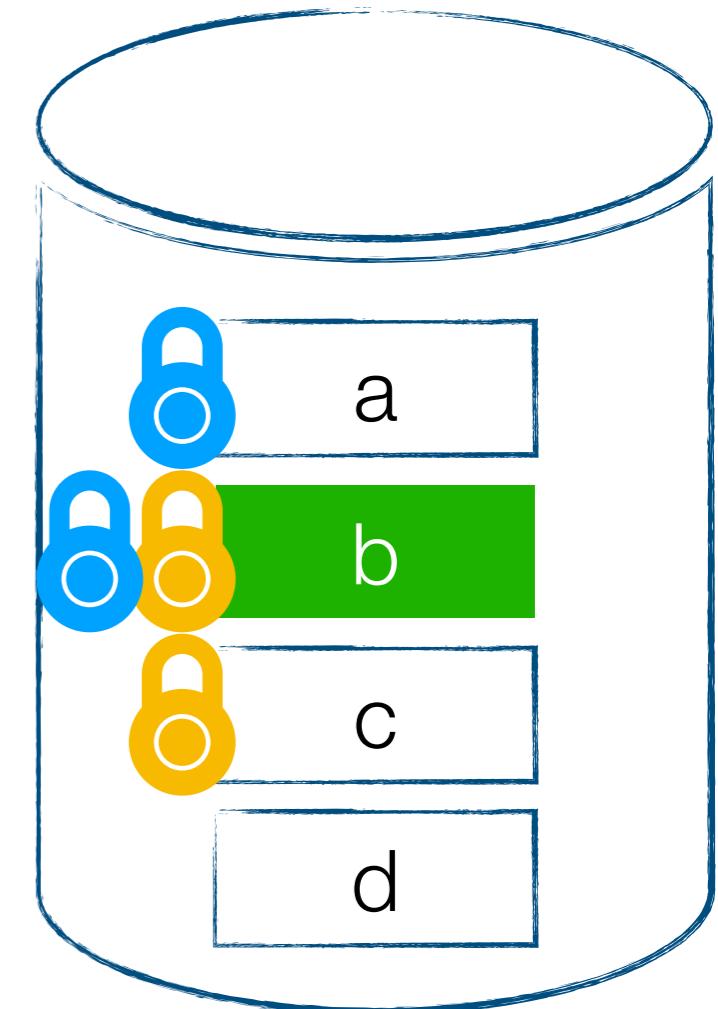
Worker
Thread #2

r₁(a)
w₁(b)

Abort transaction (to avoid potential deadlocks)

Committed Transactions

w₂(b)
r₂(a)



2PL - NoWait

Abort Count: 5

Client Transactions

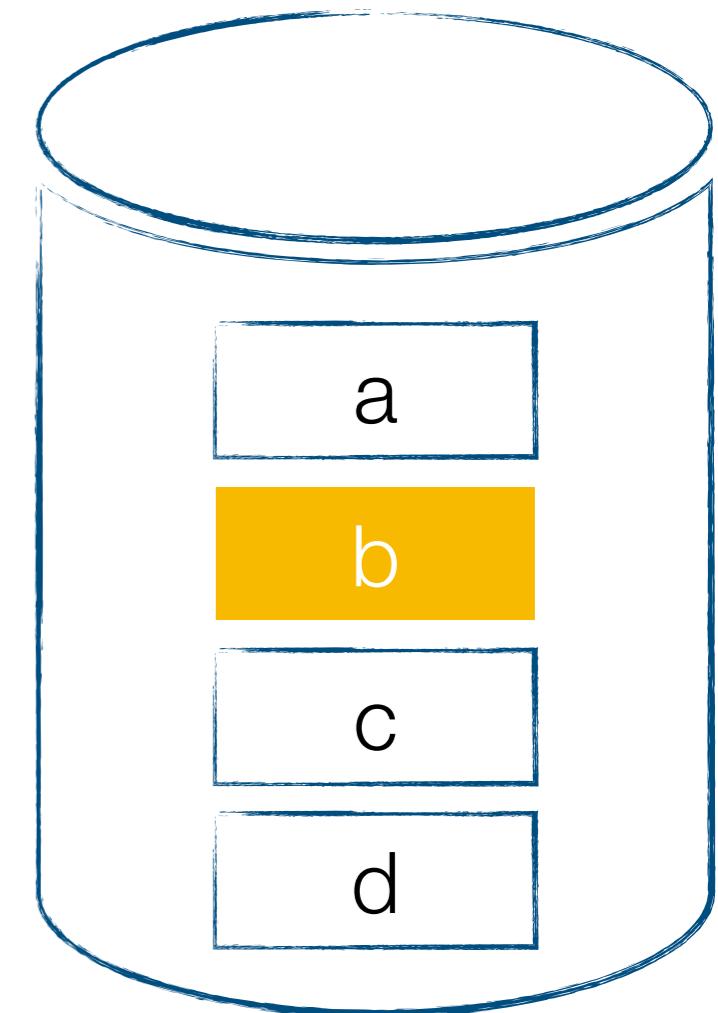
w₄(b)
r₄(d)

r₁(a)
w₁(b)

Worker
Thread #1



Worker
Thread #2



Committed Transactions

w ₃ (b)	w ₂ (b)
r ₃ (c)	r ₂ (a)

2PL - NoWait

Abort Count: 5

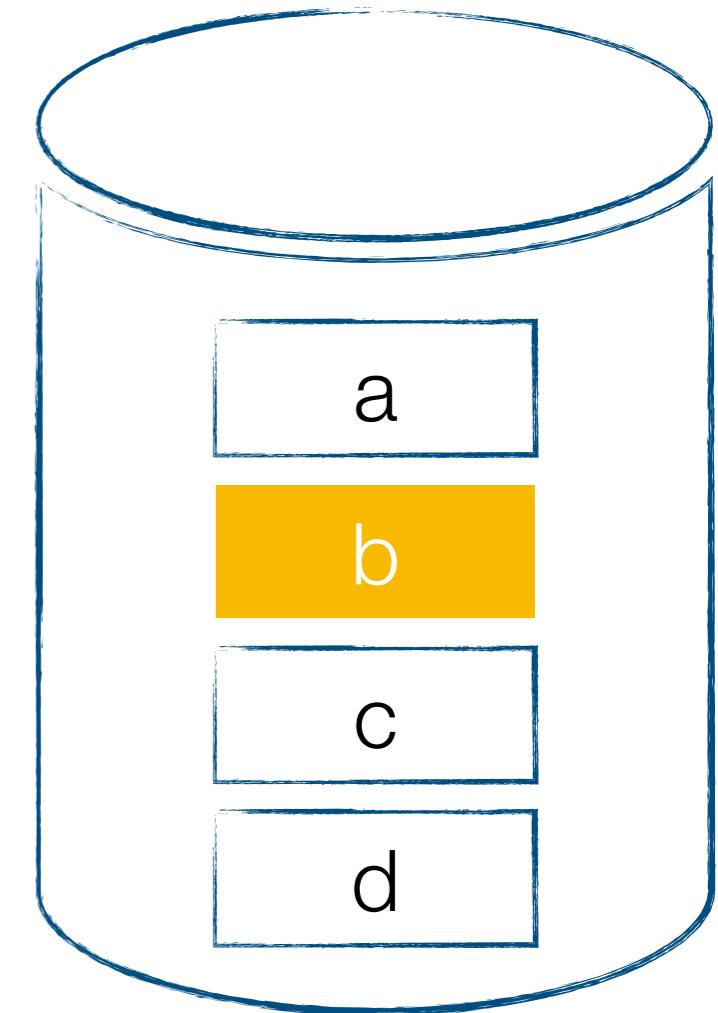
Client Transactions

Worker
Thread #1

 r₁(a)
w₁(b)

Worker
Thread #2

 w₄(b)
r₄(d)



Committed Transactions

w ₃ (b)	w ₂ (b)
r ₃ (c)	r ₂ (a)

2PL - NoWait

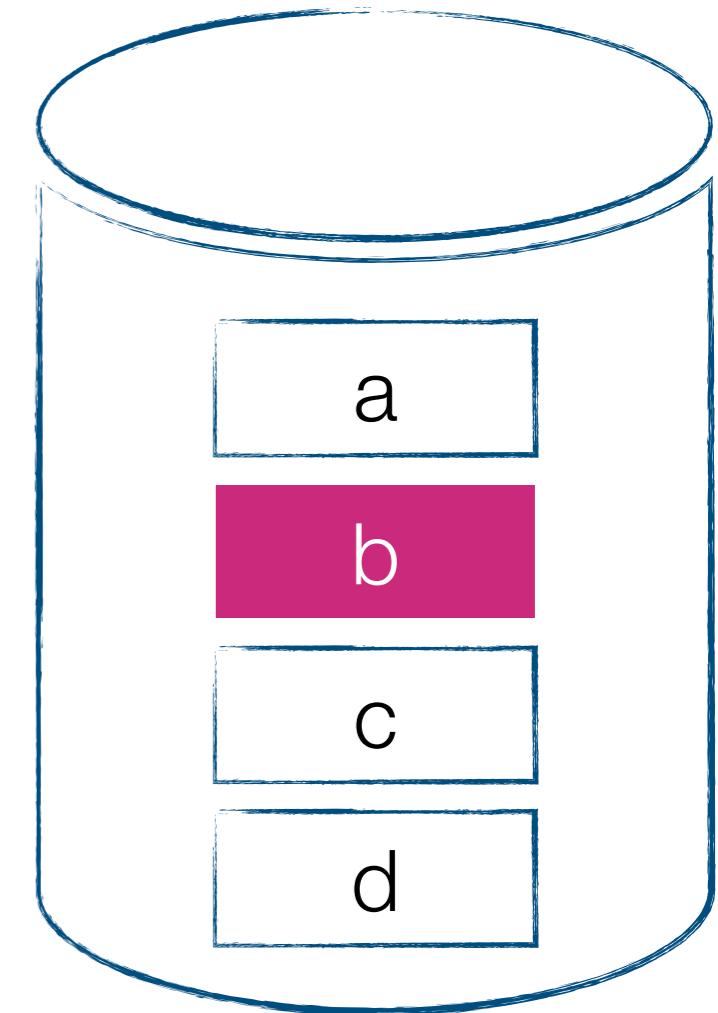
Abort Count: 5

Client Transactions

Worker
Thread #1



Worker
Thread #2



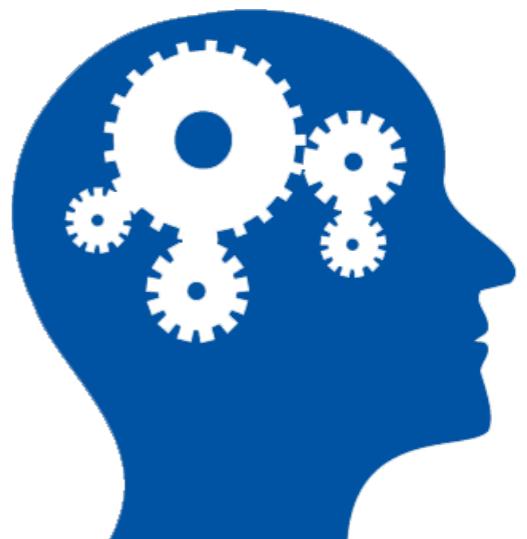
- Eventually transactions commit in some serial order!
- Many aborts due to high contention on record b
- Non-determinism in CC cause these aborts
- Wasted work

Committed Transactions

w ₄ (b)	r ₁ (a)	w ₃ (b)	w ₂ (b)
r ₄ (d)	w ₁ (b)	r ₃ (c)	r ₂ (a)

Key Insights

- Many aborts due to high contention
- Non-determinism in CC cause these aborts
- Can we do better?
- Is it possible to eliminate non-deterministic concurrency control from transaction execution?



Deterministic Transaction Execution

- H-Store [Kallman et al. '08]
- Designed and optimized for horizontal scalability, multi-core hardware and in-memory databases
- Stored procedure transaction model
- Static partitioning of database
- Assigns a single core to each partition
- Execute transaction serially without concurrency control within each partition

H-Store

Abort Count: 0

Client Transactions

w ₄ (d)	w ₃ (b)	w ₂ (c)	r ₁ (a)
r ₄ (c)	r ₃ (a)	r ₂ (d)	w ₁ (b)

Single-partition
transactions

Worker
Thread #1

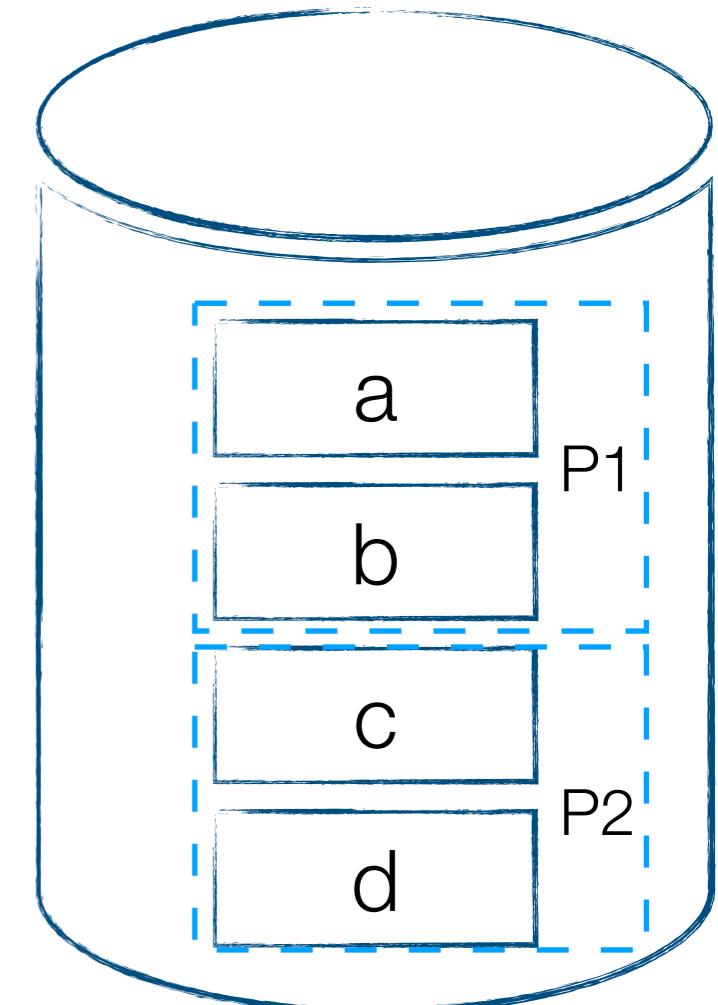


P1 is assigned to
Worker Thread #1

Worker
Thread #2



P2 is assigned to
Worker Thread #2



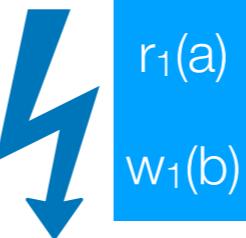
H-Store

Abort Count: 0

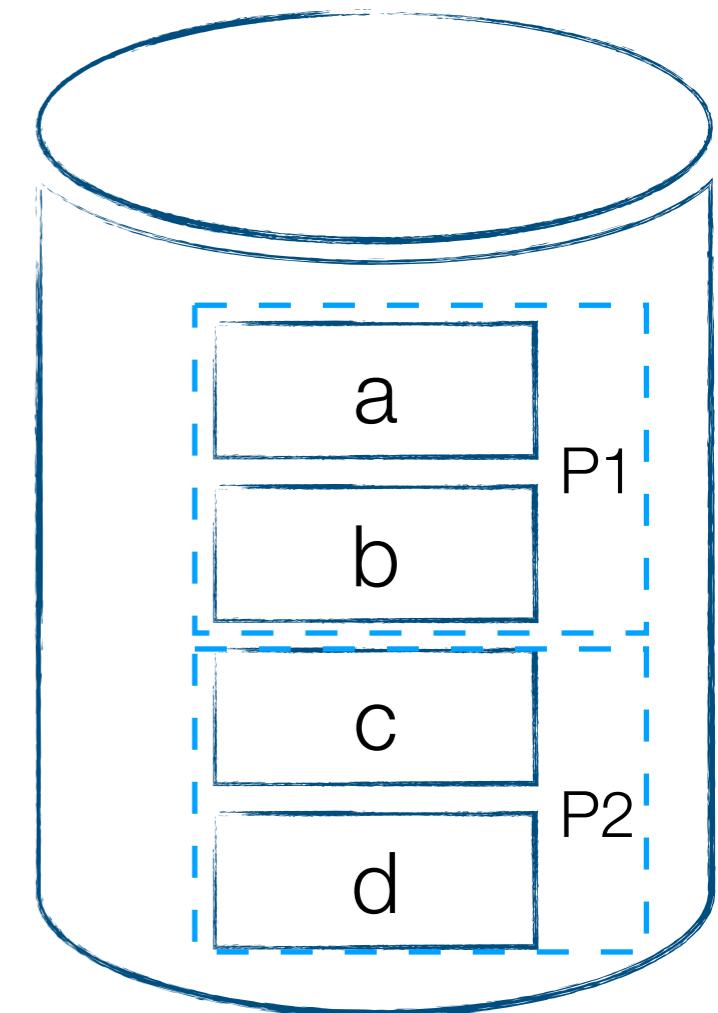
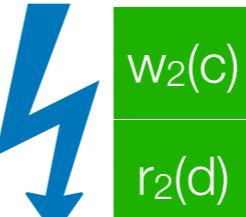
Client Transactions

w ₄ (d)	w ₃ (b)
r ₄ (c)	r ₃ (a)

Worker Thread #1



Worker Thread #2



Committed Transactions

H-Store

Abort Count: 0

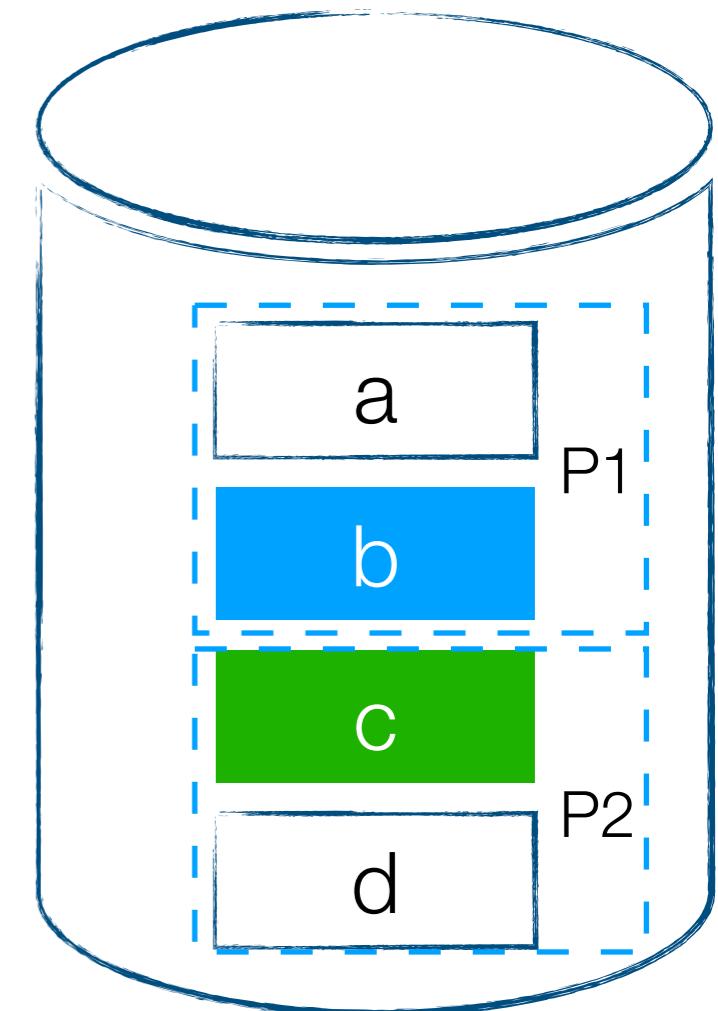
Client Transactions

w ₄ (d)	w ₃ (b)
r ₄ (c)	r ₃ (a)

Worker Thread #1



Worker Thread #2



Committed Transactions

w ₂ (c)	r ₁ (a)
r ₂ (d)	w ₁ (b)

H-Store

Abort Count: 0

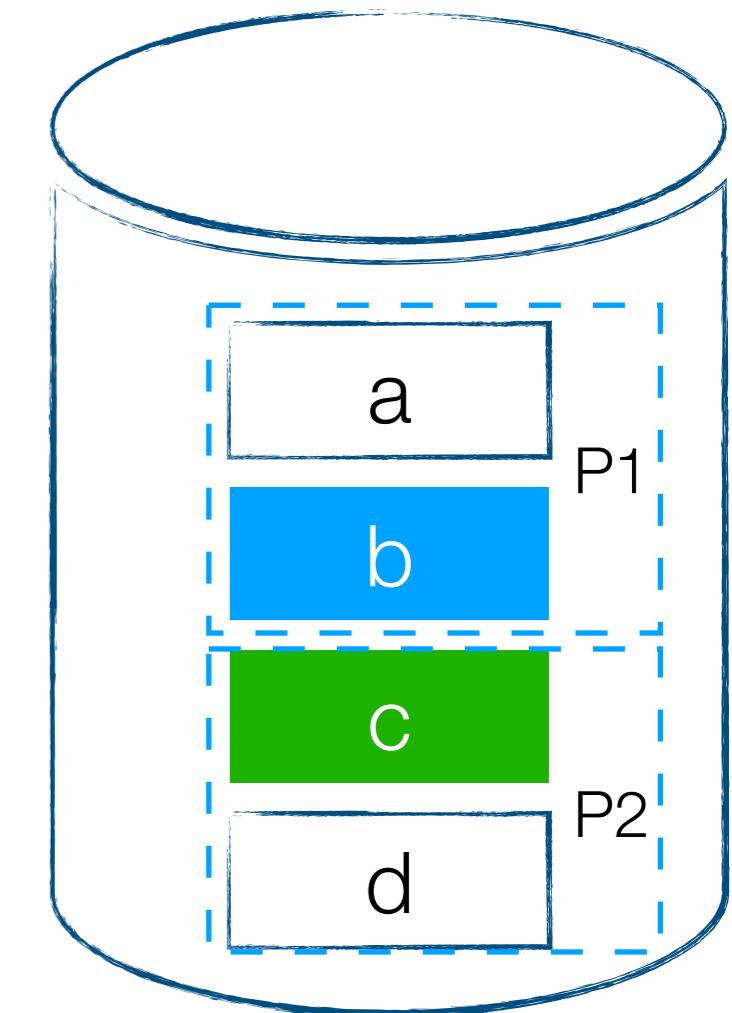
Client Transactions

Worker
Thread #1

w₃(b)
r₃(a)

Worker
Thread #2

w₄(d)
r₄(c)



Committed Transactions

w ₂ (c)	r ₁ (a)
r ₂ (d)	w ₁ (b)

H-Store

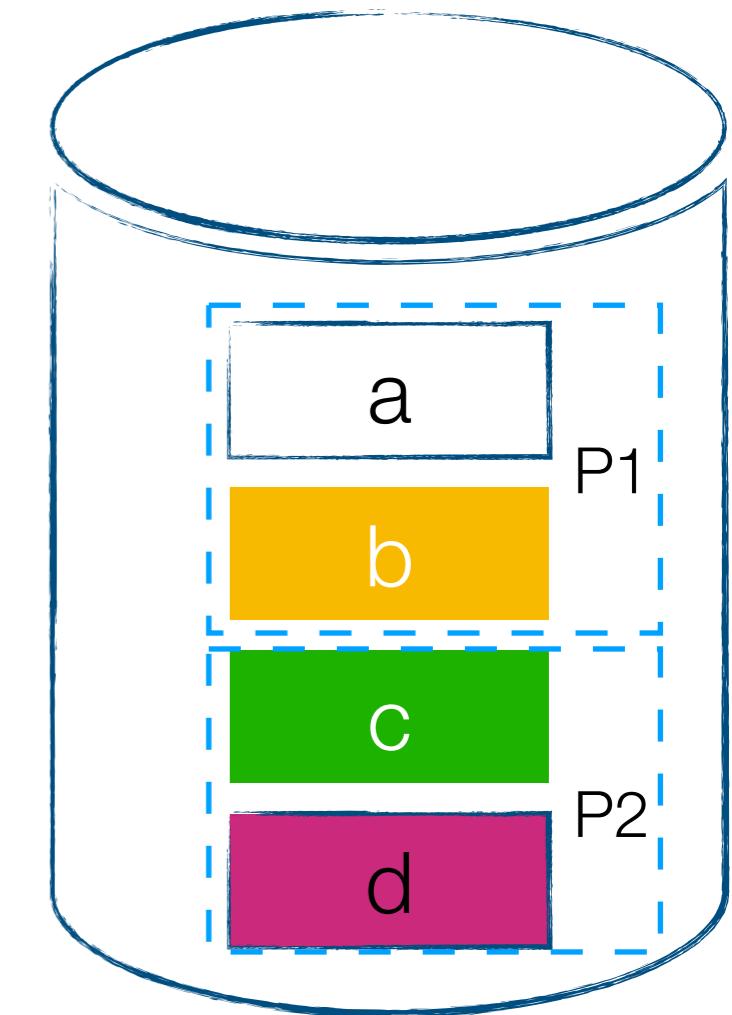
Abort Count: 0

Client Transactions

Worker
Thread #1



Worker
Thread #2



Committed Transactions

w ₄ (d)	w ₃ (b)	w ₂ (c)	r ₁ (a)
r ₄ (c)	r ₃ (a)	r ₂ (d)	w ₁ (b)

H-Store

Abort Count: 0

Client Transactions

Worker Thread #1

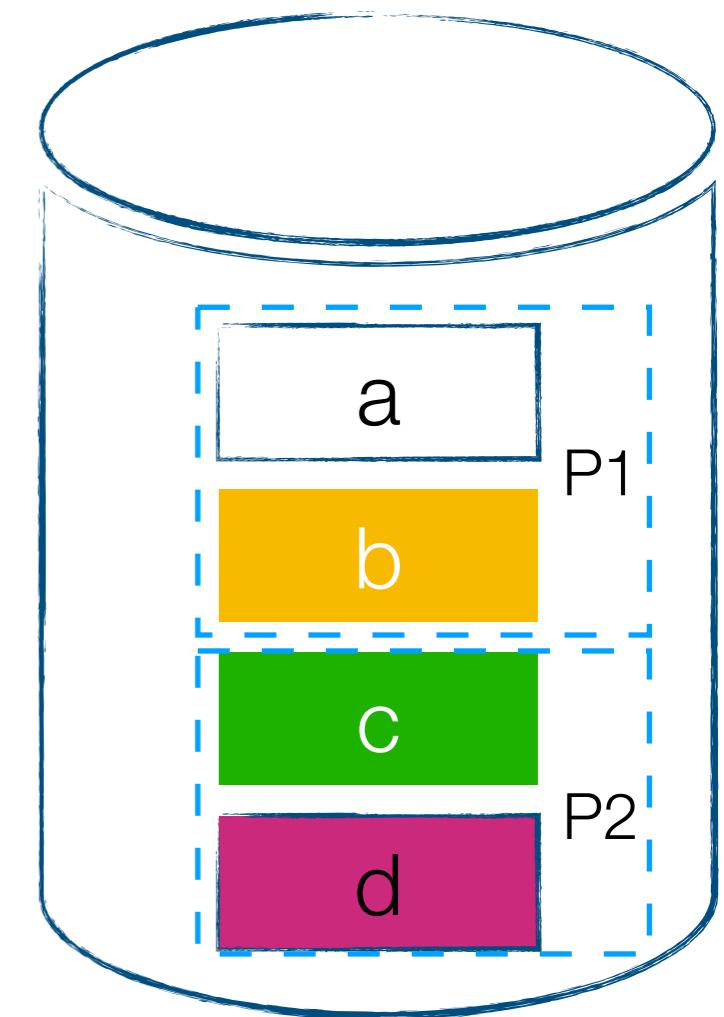


Worker Thread #2



- ✓ Deterministic Execution
- ✓ No aborts because of CC
- ✓ Minimal coordination among threads

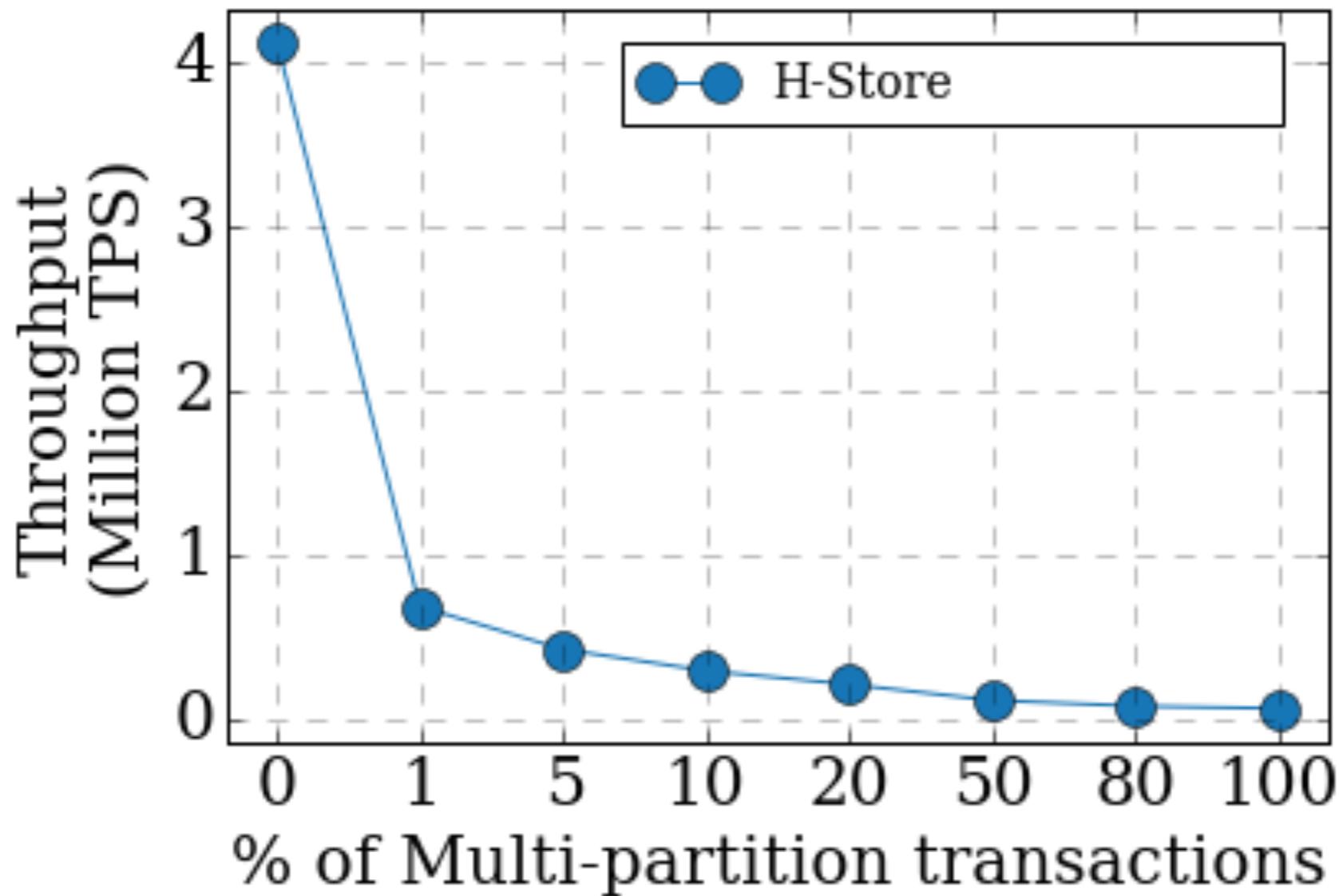
● Performs well only when transactions are single-partitioned



Committed Transactions

w ₄ (d)	w ₃ (b)	w ₂ (c)	r ₁ (a)
r ₄ (c)	r ₃ (a)	r ₂ (d)	w ₁ (b)

Effect of Increasing Percentage of Multi-Partition Transactions in the Workload



H-Store is sensitive to the percentage of multi-partition transactions in the workload

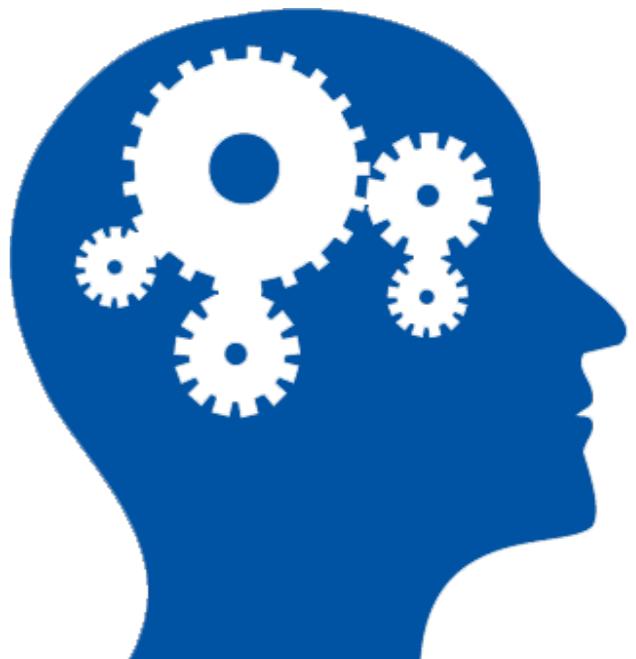
Can We Do Better?

Our motivations are

- Efficiently exploits multi-core and large main-memory systems
- Provide serializable multi-statement transactions for key-value stores
- Scales well under high-contention workloads

Desired Properties

- Concurrent execution over shared data
- Not limited to partitionable workloads
- Without any concurrency controls



Is it possible to have concurrent execution over shared data without having any concurrency controls?

Introducing: QueCC

Queue-Oriented, Control-Free, Concurrency Architecture

A two parallel & independent phases of priority-driven planning & execution

Phase 1: Deterministic priority-based planning of transaction operations in parallel

- *Plans take the form of **Prioritized Execution Queues***
- Execution Queues inherits predetermined priority of its planner
- Results in a deterministic plan of execution

Phase 2: Priority driven execution of plans in parallel

- Satisfies the ***Execution Priority Invariance***

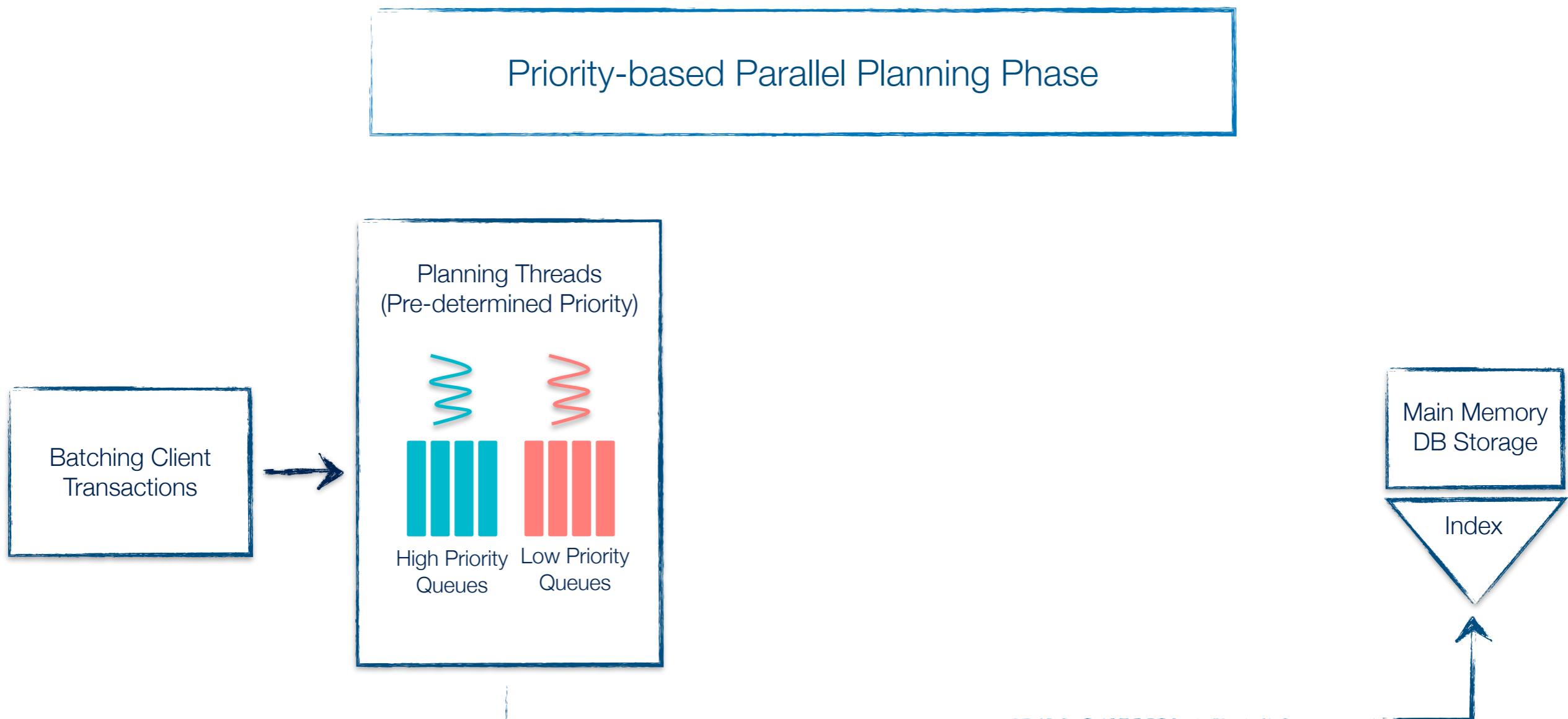
“For each record (or a queue), operations that belong to higher priority queues (created by a higher priority planner) must always be executed before executing any lower priority operations.”

QueCC Architecture

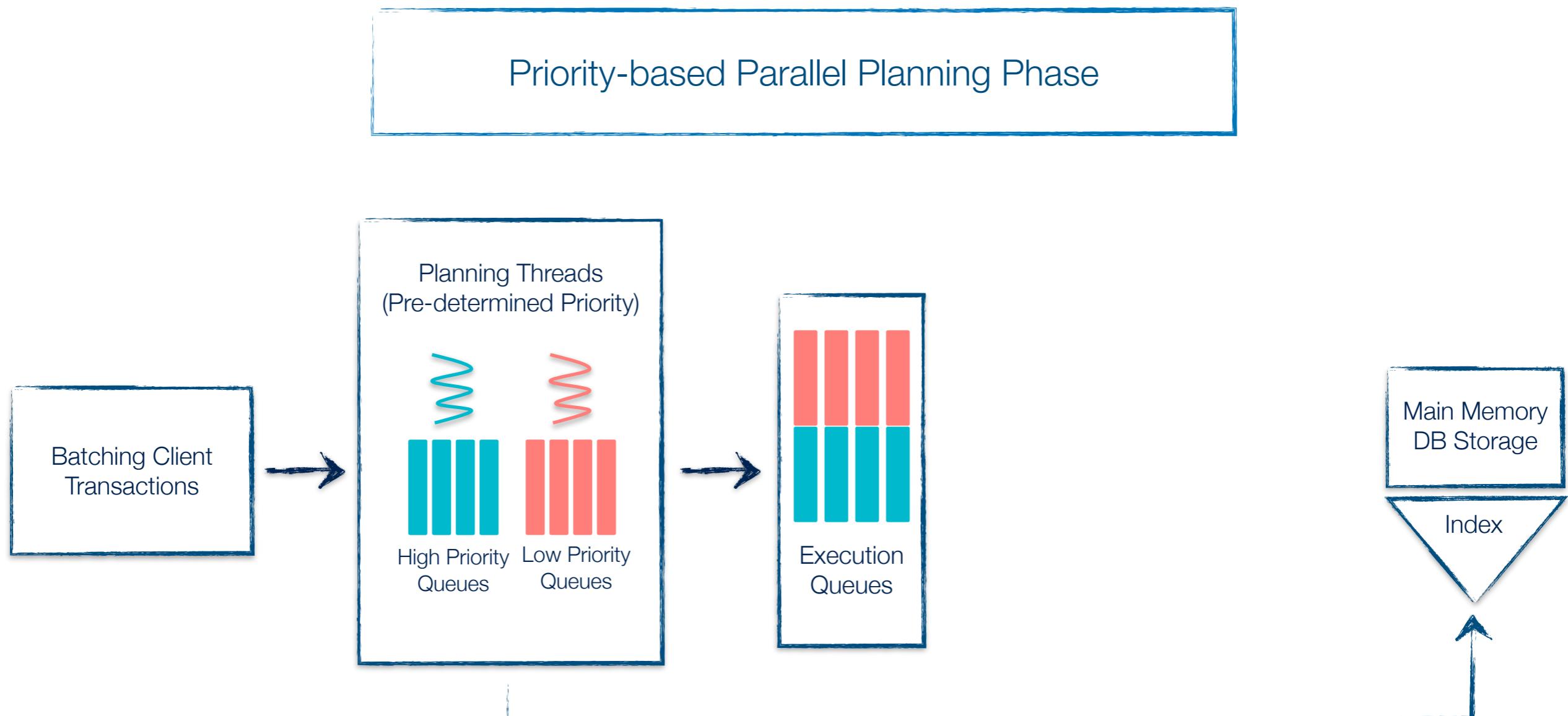
Priority-based Parallel Planning Phase

Batching Client
Transactions

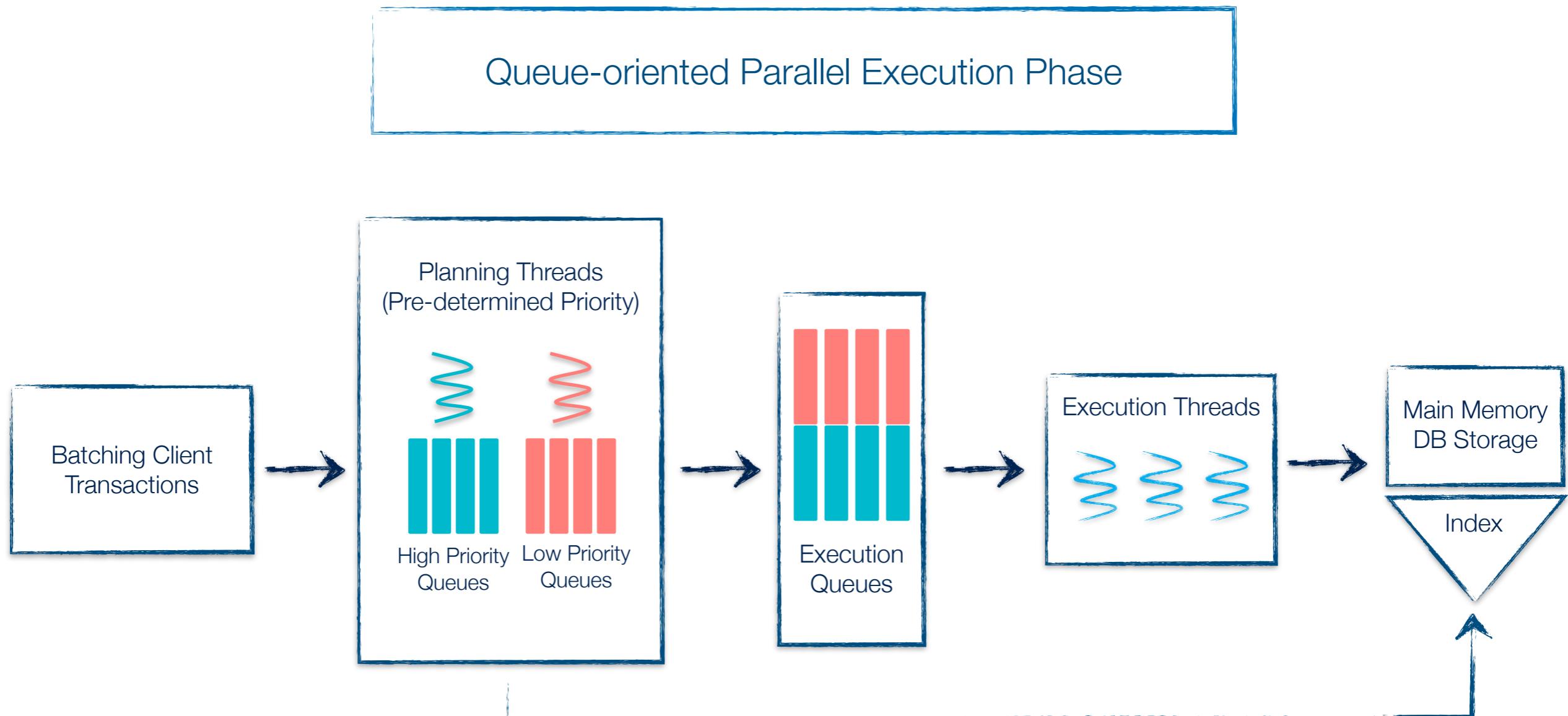
QueCC Architecture



QueCC Architecture

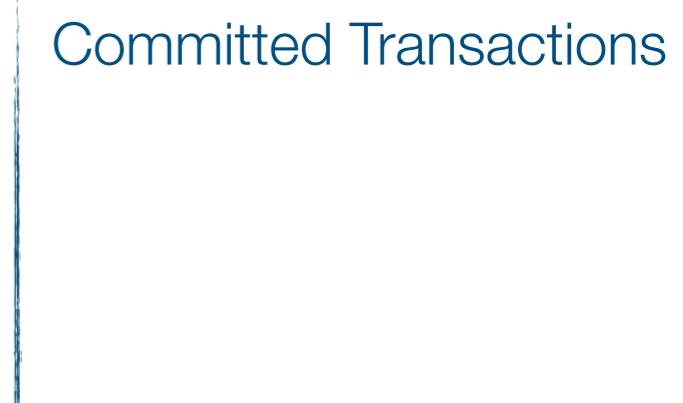
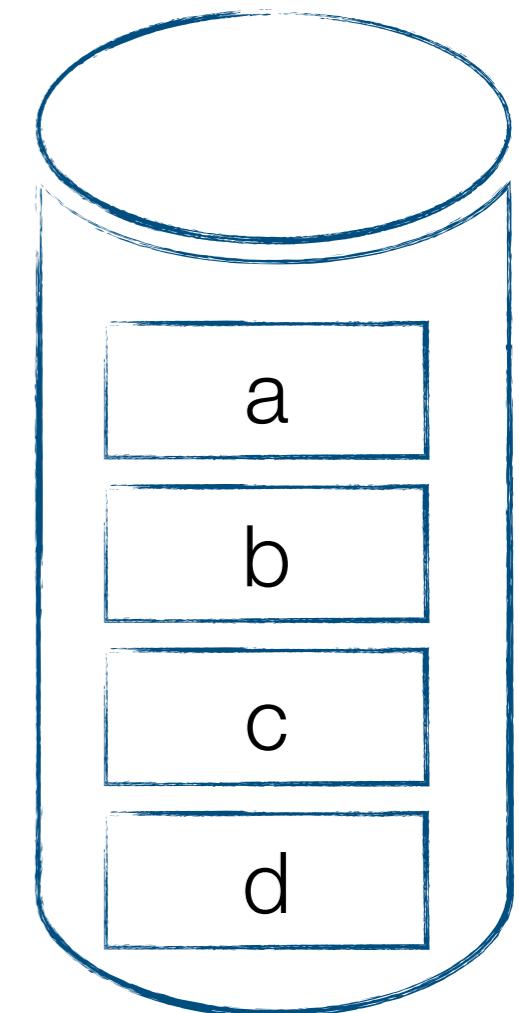
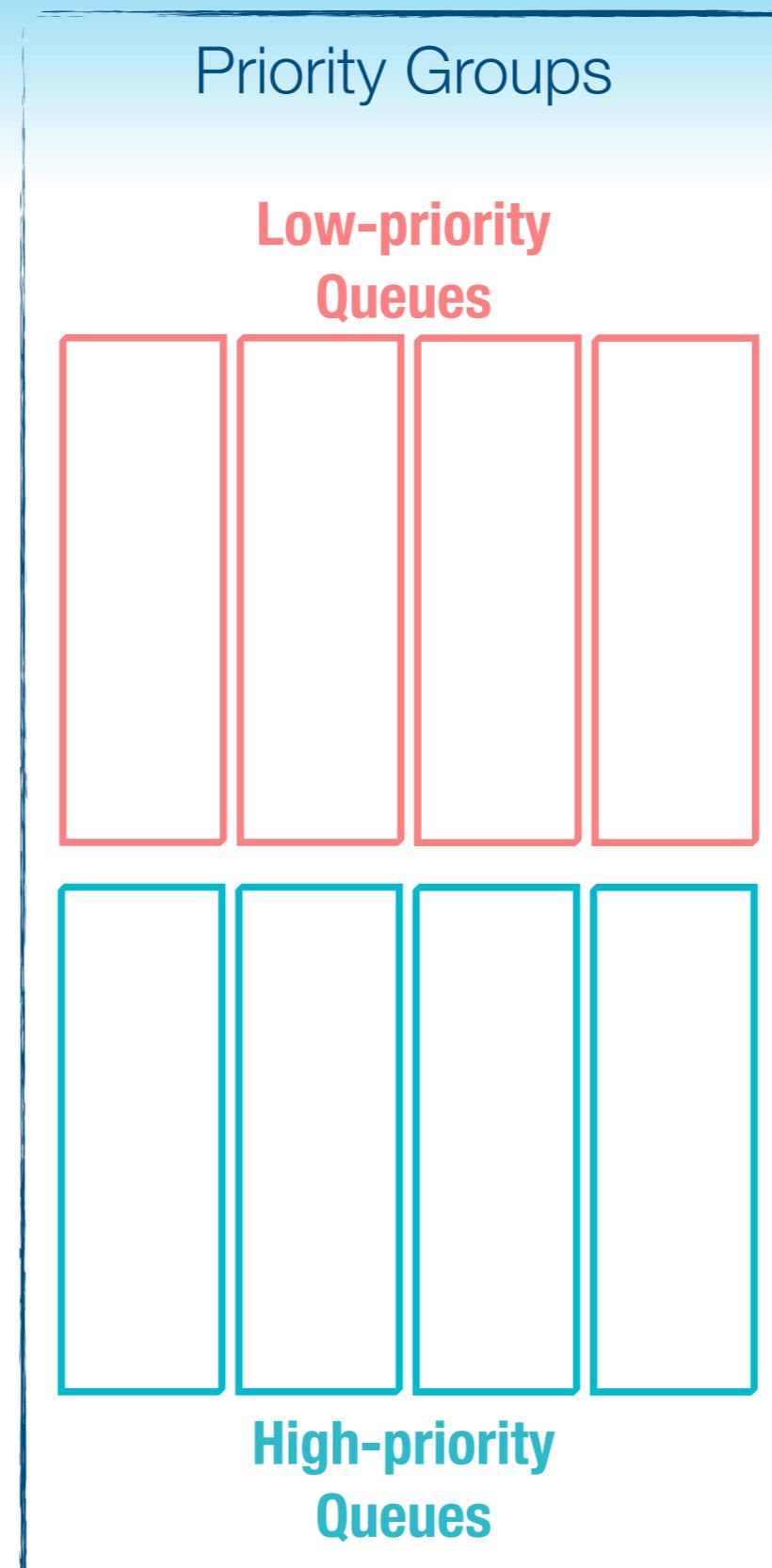
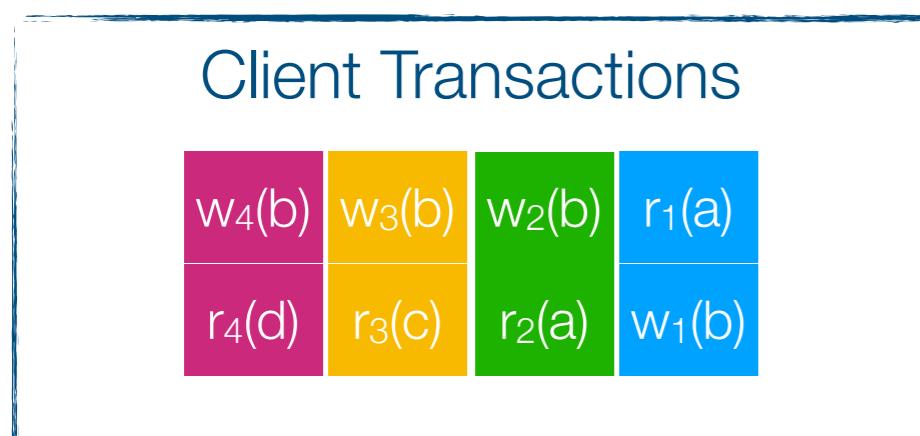


QueCC Architecture



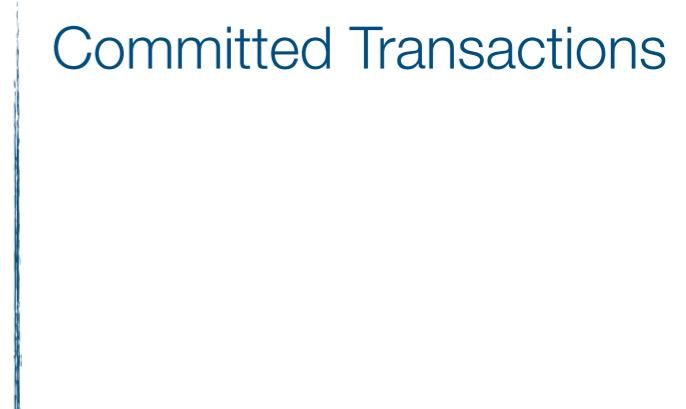
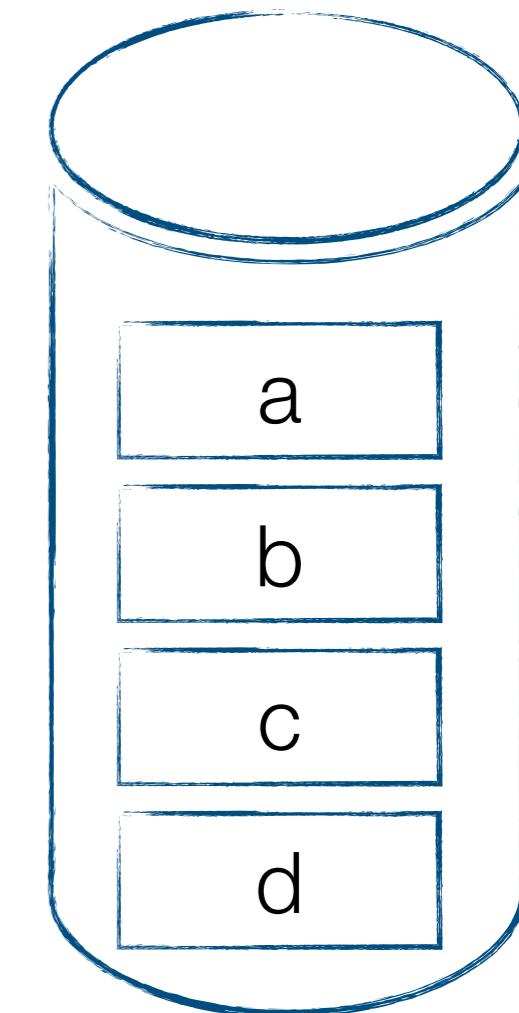
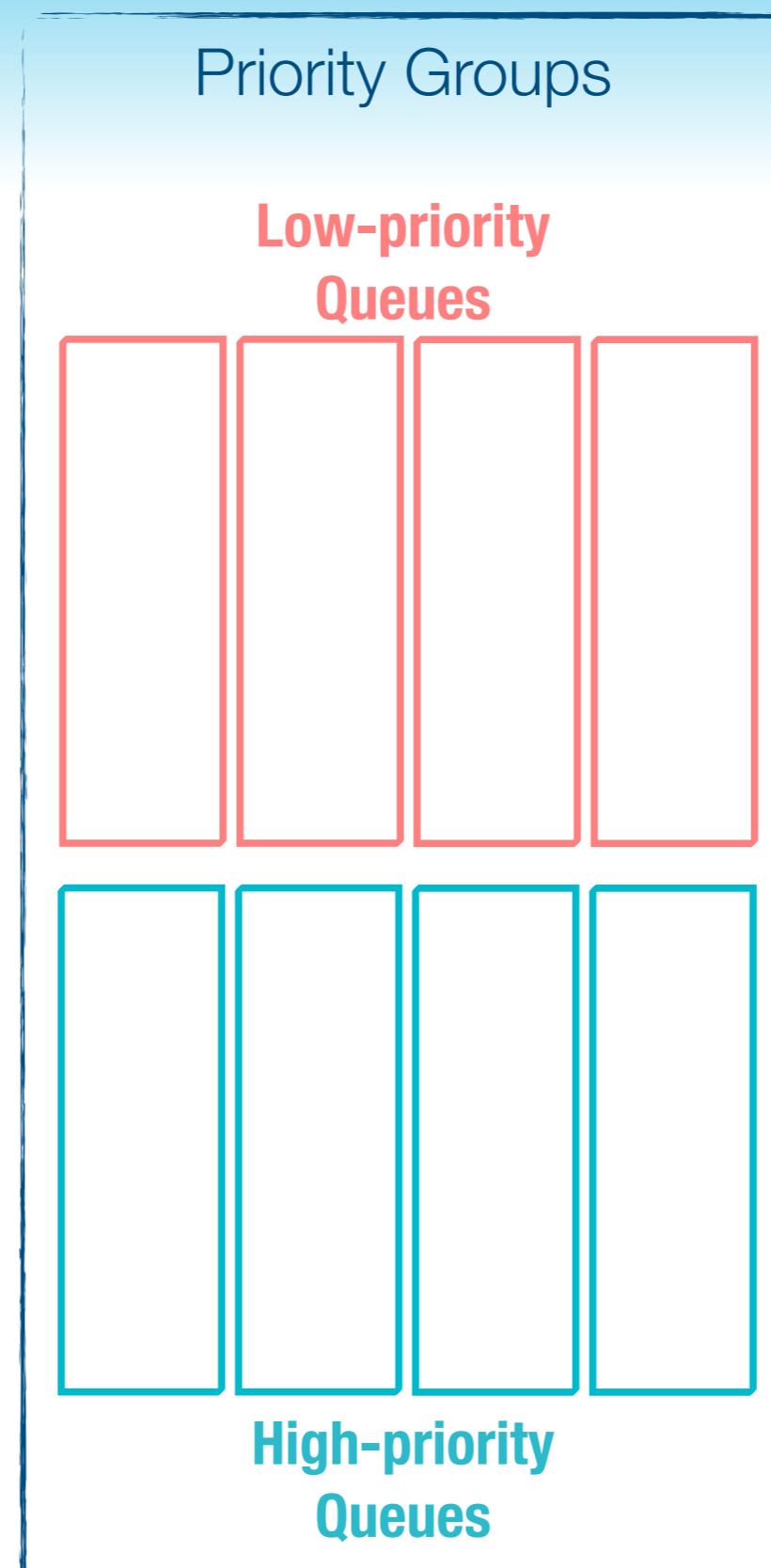
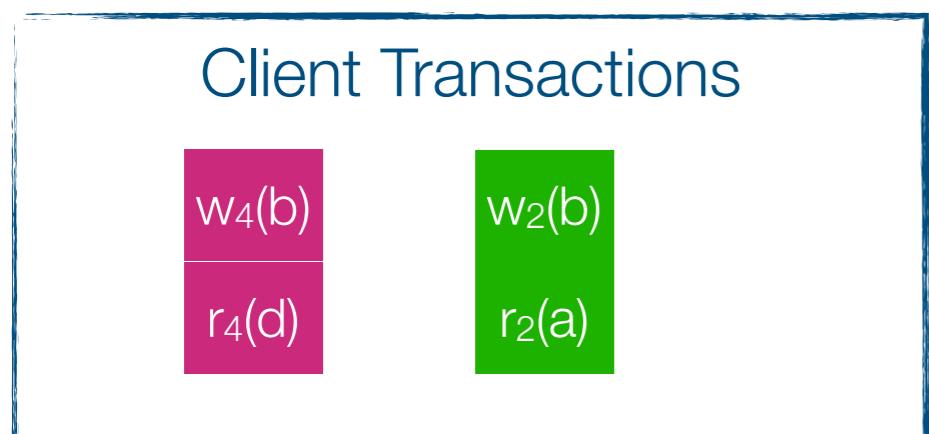
QueCC

Abort Count: 0



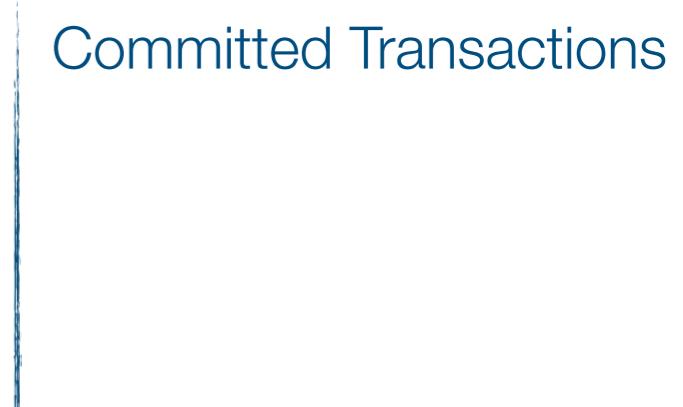
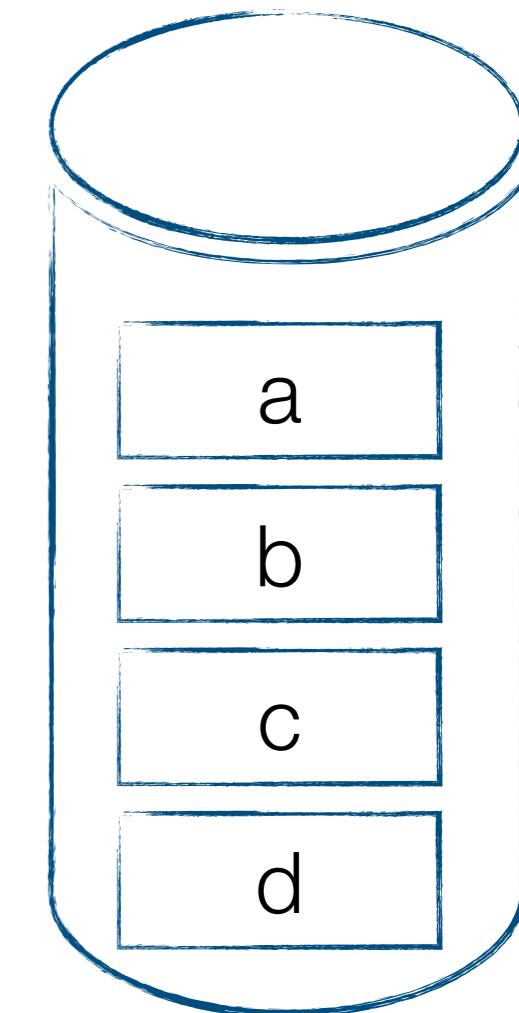
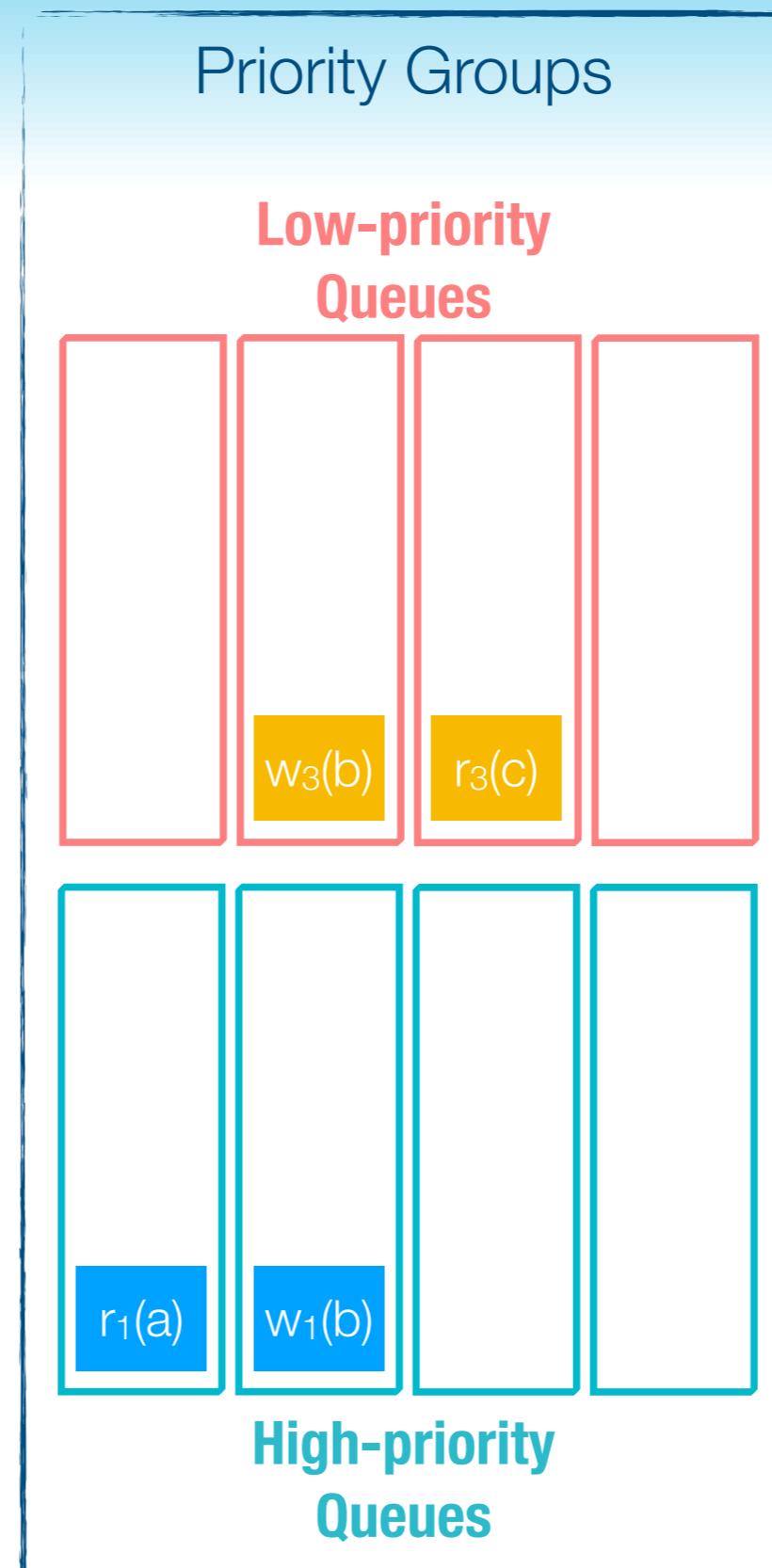
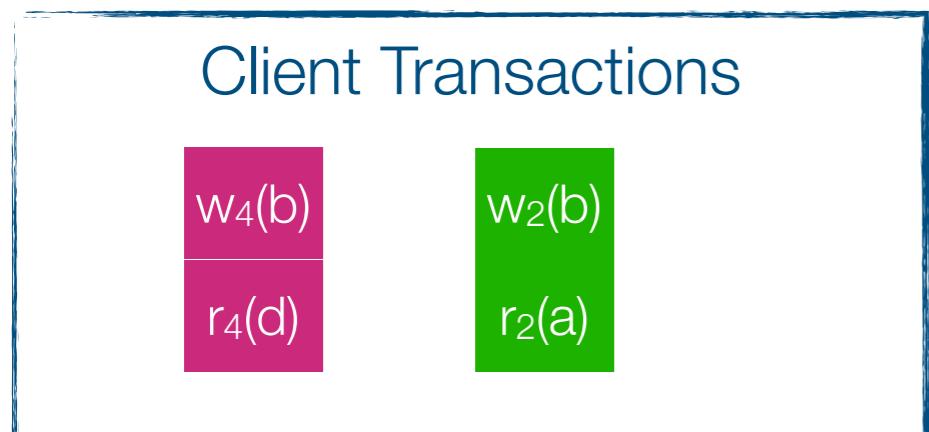
QueCC

Abort Count: 0



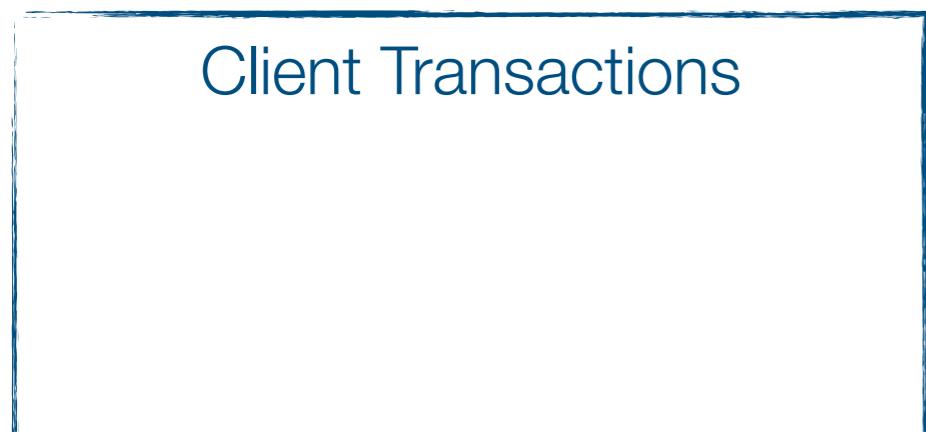
QueCC

Abort Count: 0



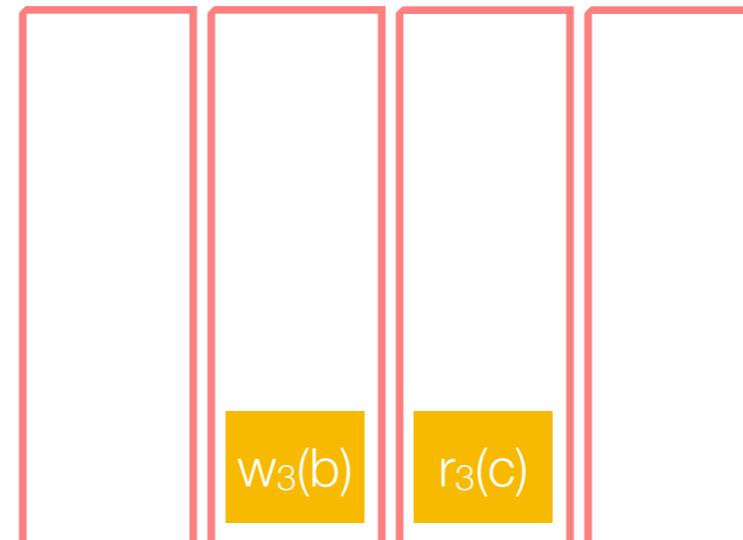
QueCC

Abort Count: 0

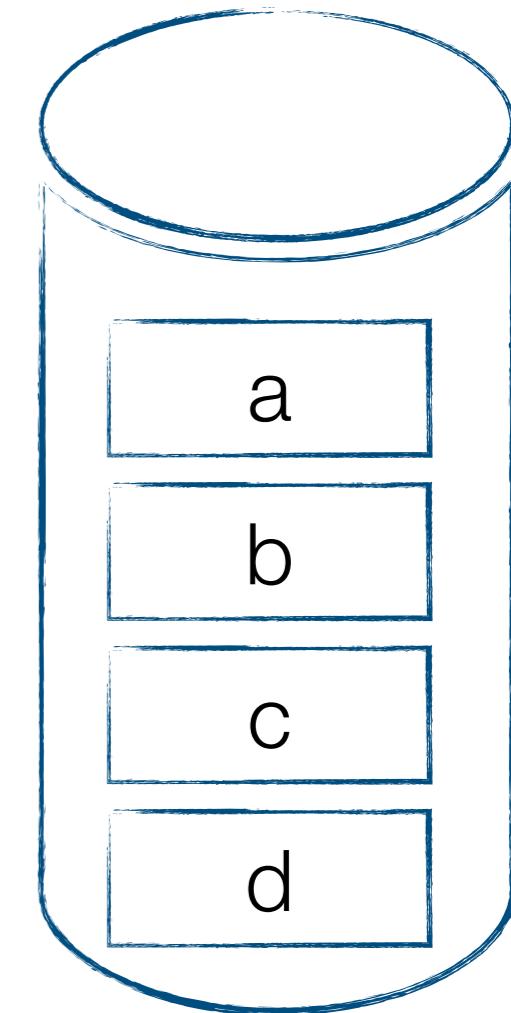
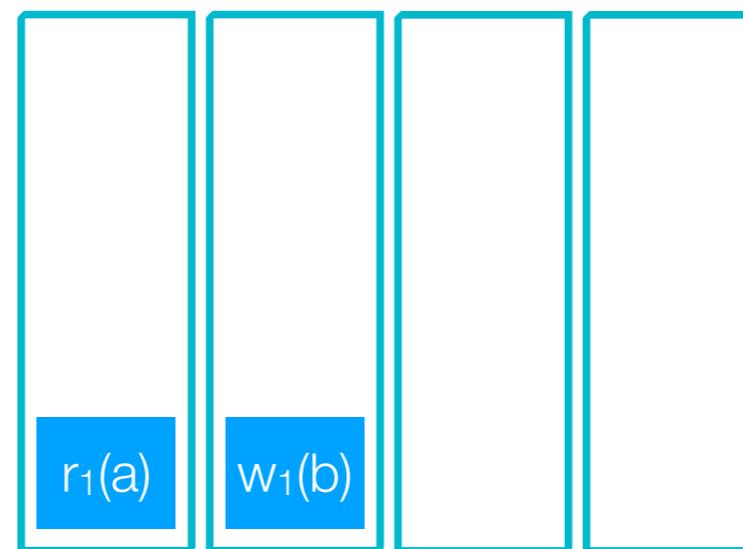


Priority Groups

Low-priority Queues



High-priority Queues



Committed Transactions

QueCC

Abort Count: 0

Planning Thread #2



Client Transactions

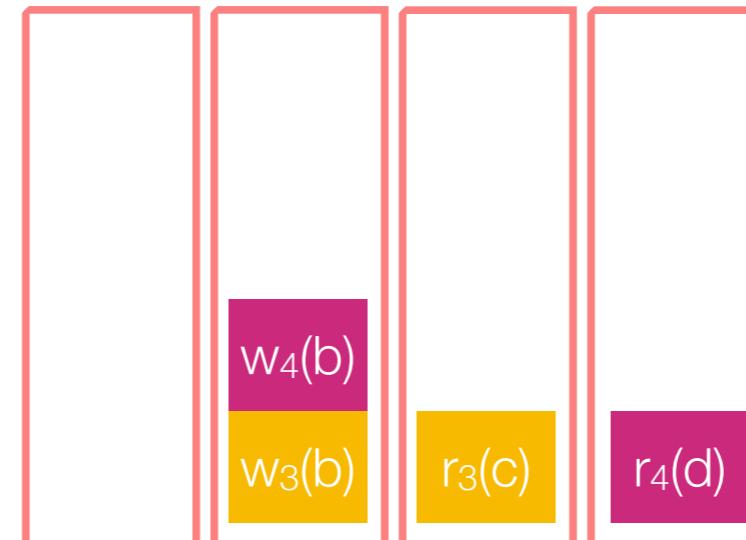
Planning Thread #1



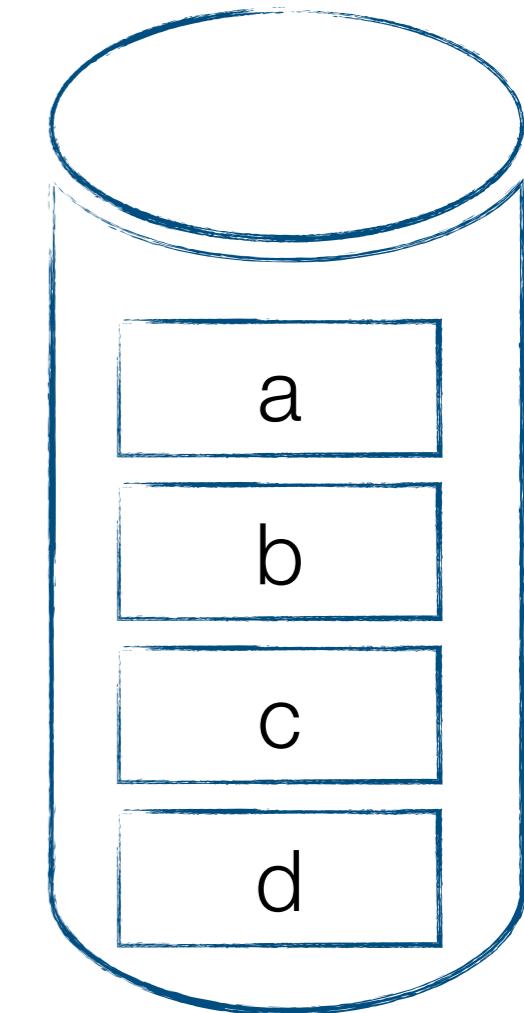
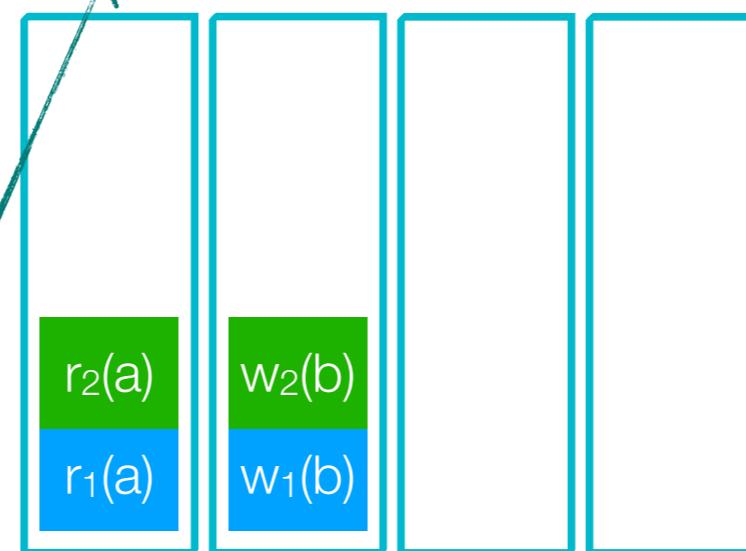
Prioritized Execution
Queues

Priority Groups

Low-priority
Queues



High-priority
Queues



Committed Transactions

QueCC

Abort Count: 0

Execution Thread #2



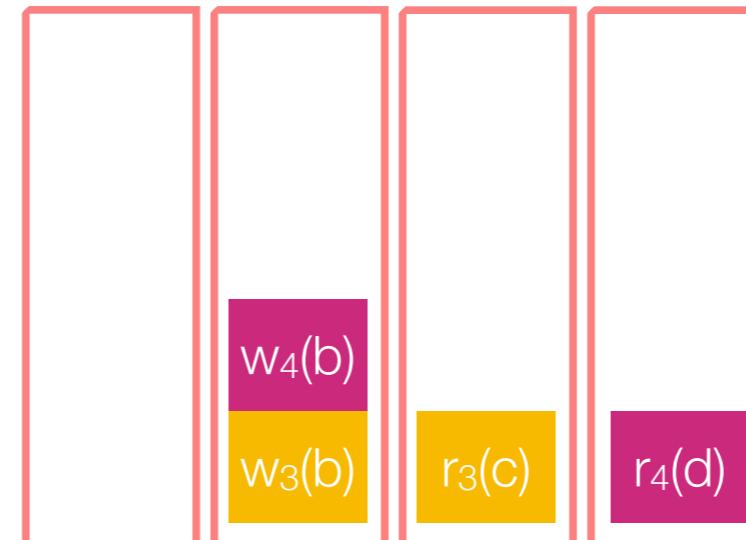
Client Transactions

Execution Thread #1

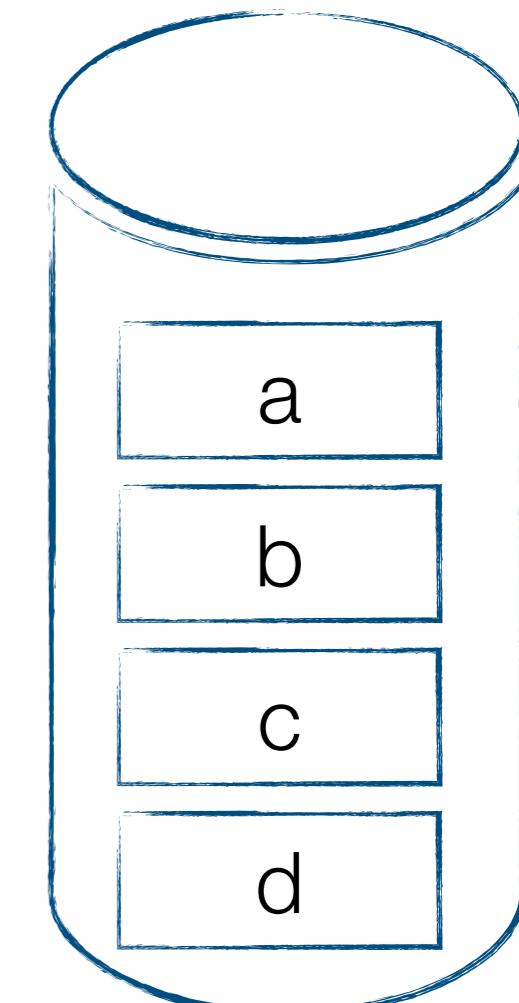
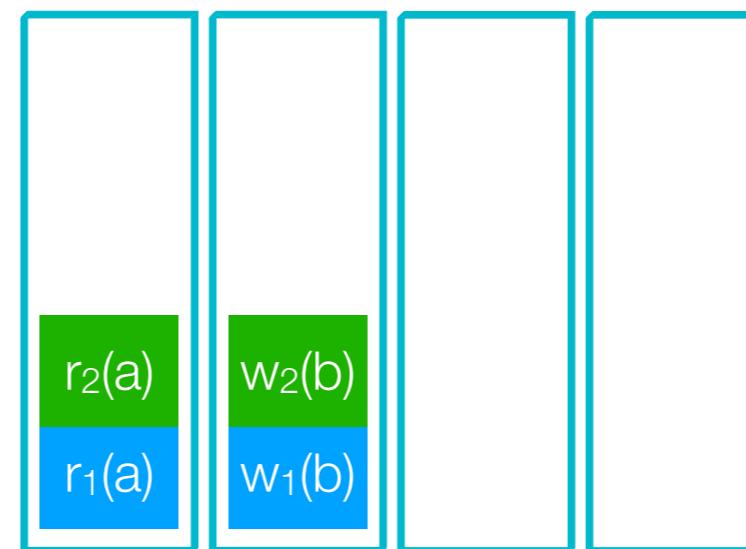


Priority Groups

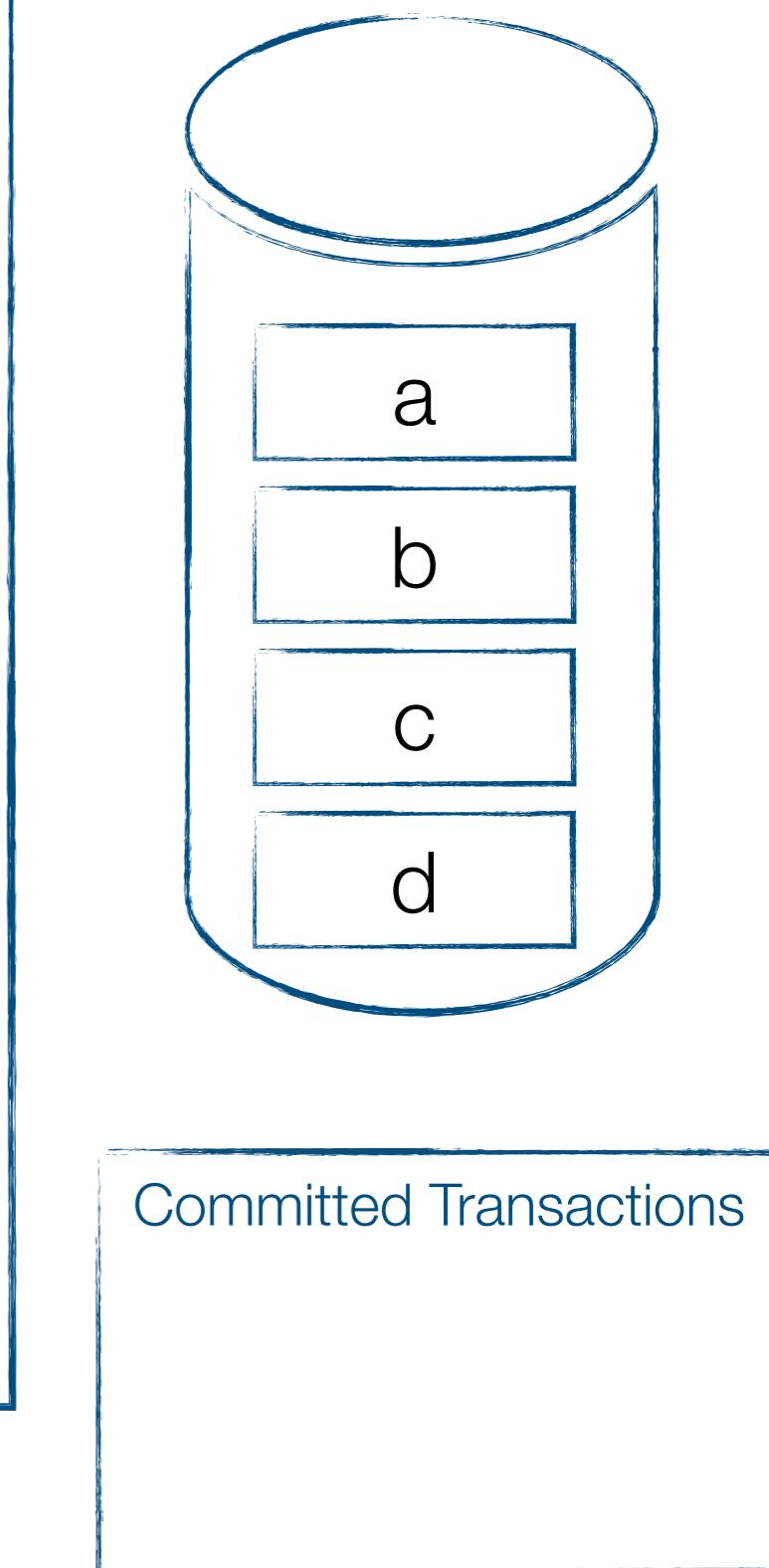
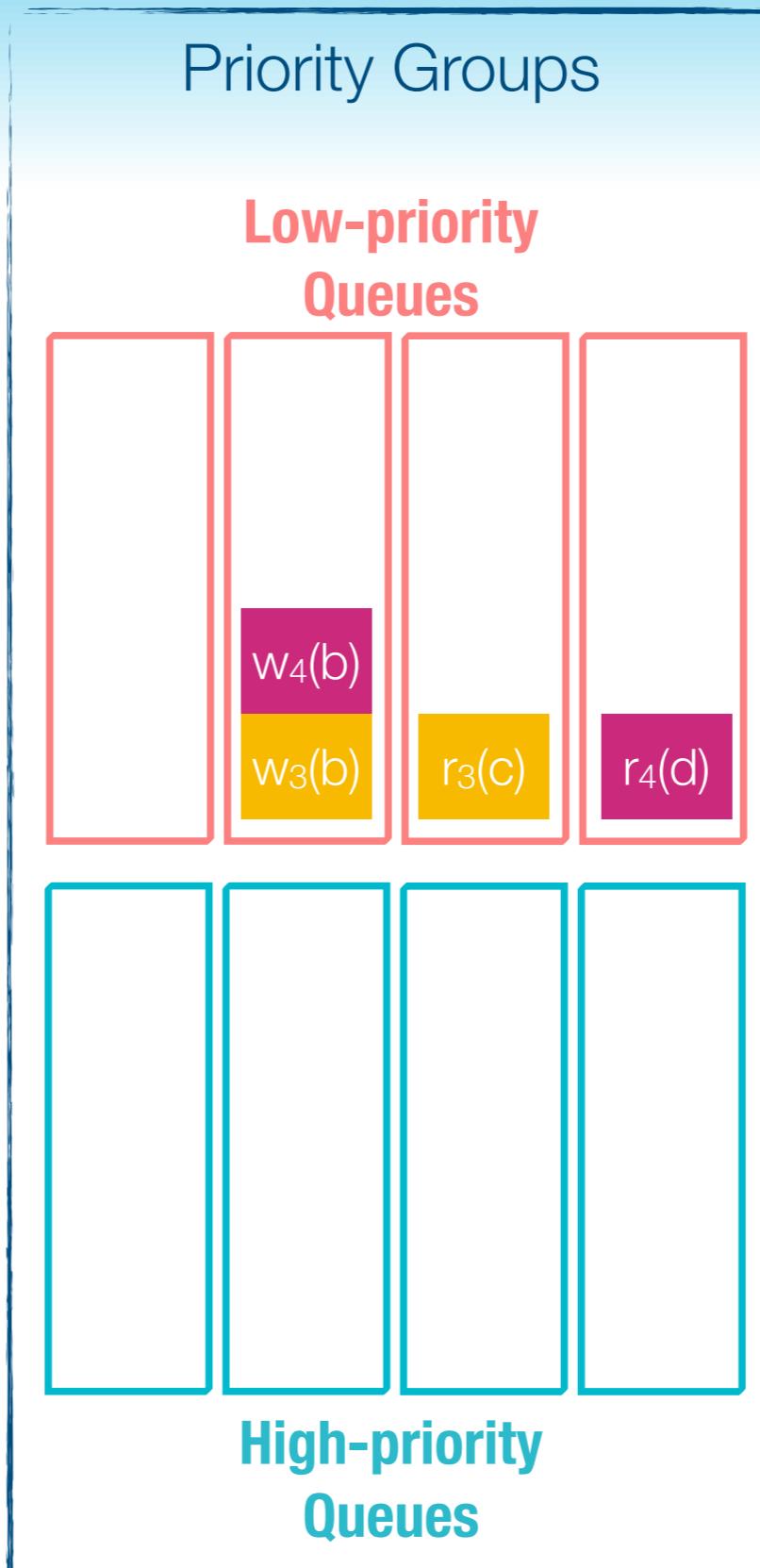
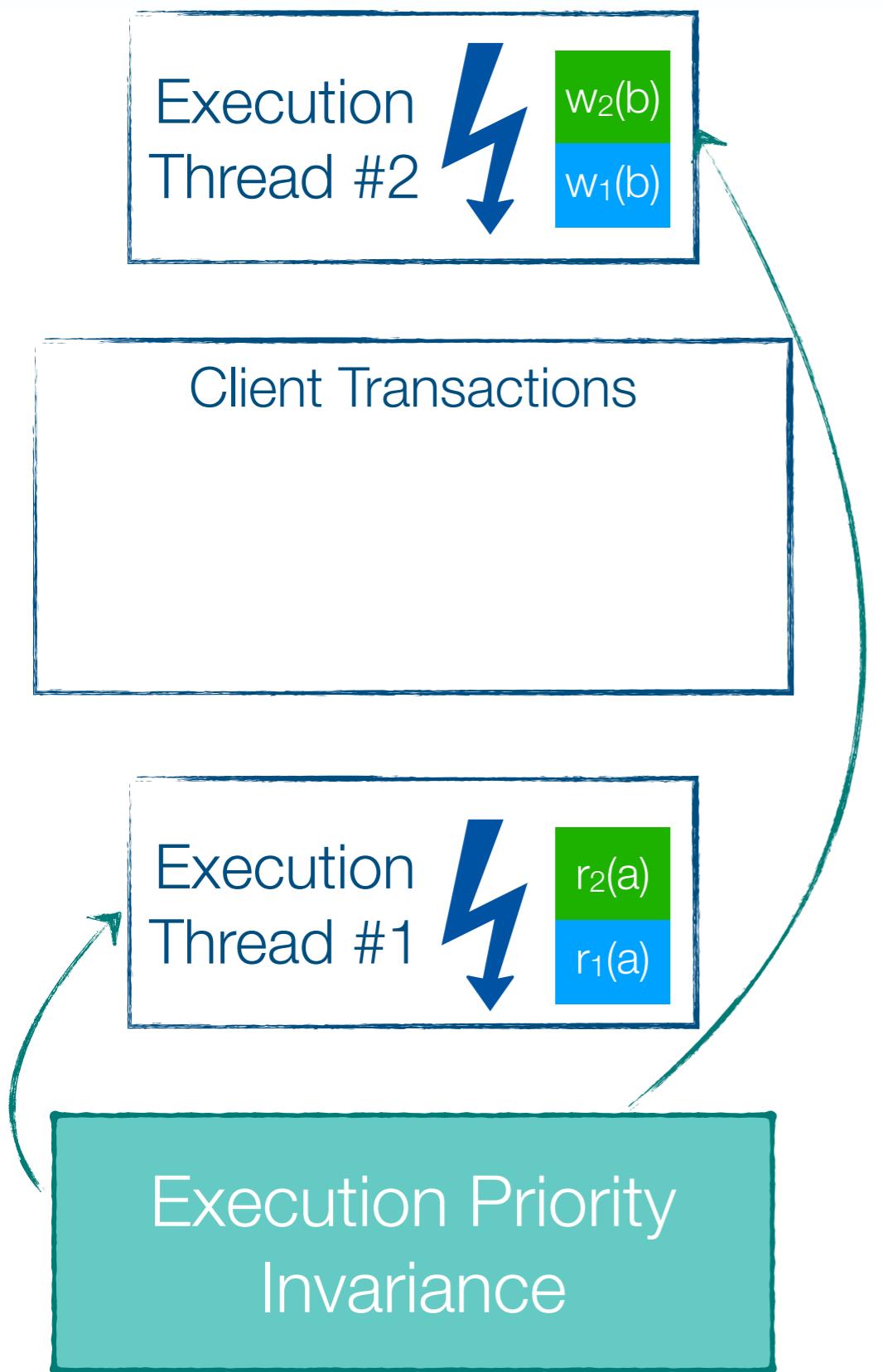
Low-priority
Queues



High-priority
Queues



Committed Transactions



QueCC

Abort Count: 0

Execution Thread #2



Client Transactions

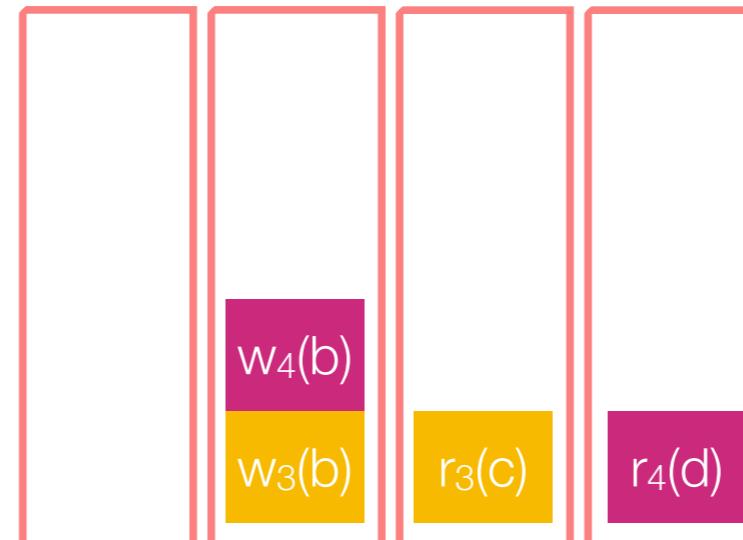
Execution Thread #1



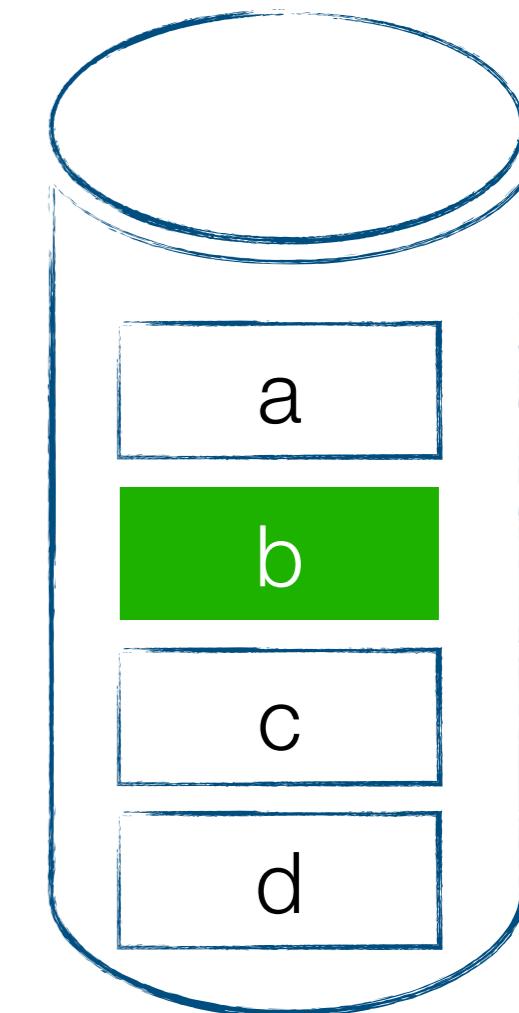
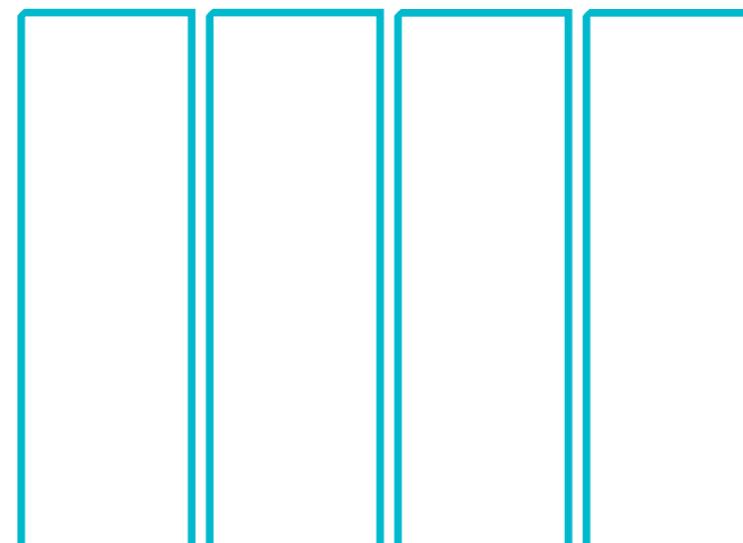
Execution Priority Invariance

Priority Groups

Low-priority
Queues



High-priority
Queues

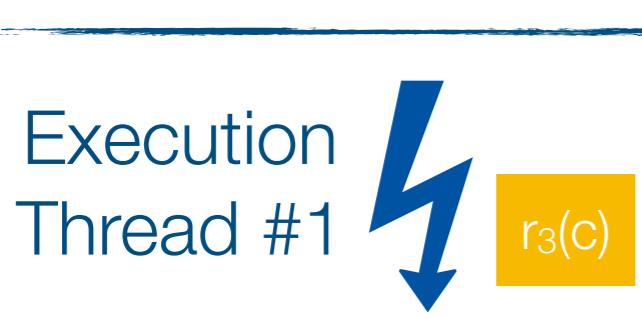
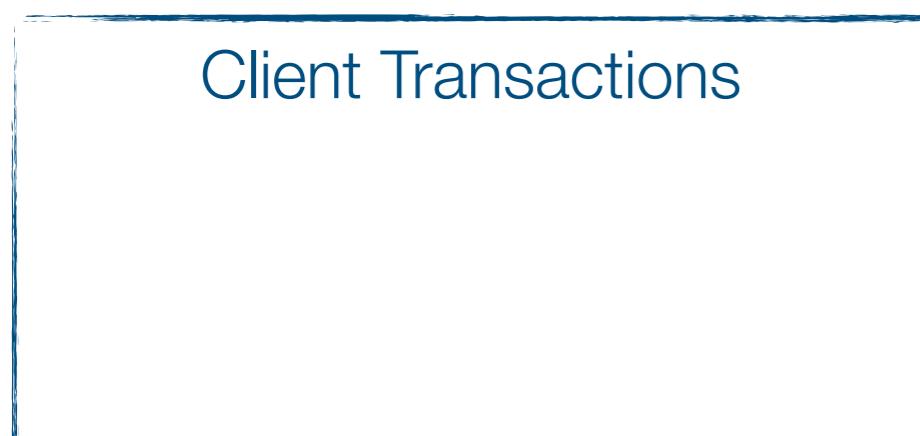


Committed Transactions

$w_2(b)$ $r_1(a)$
 $r_2(a)$ $w_1(b)$

QueCC

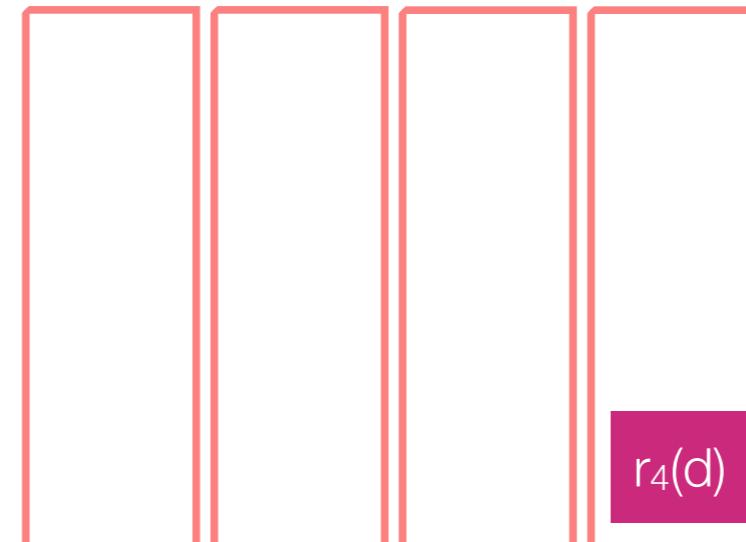
Abort Count: 0



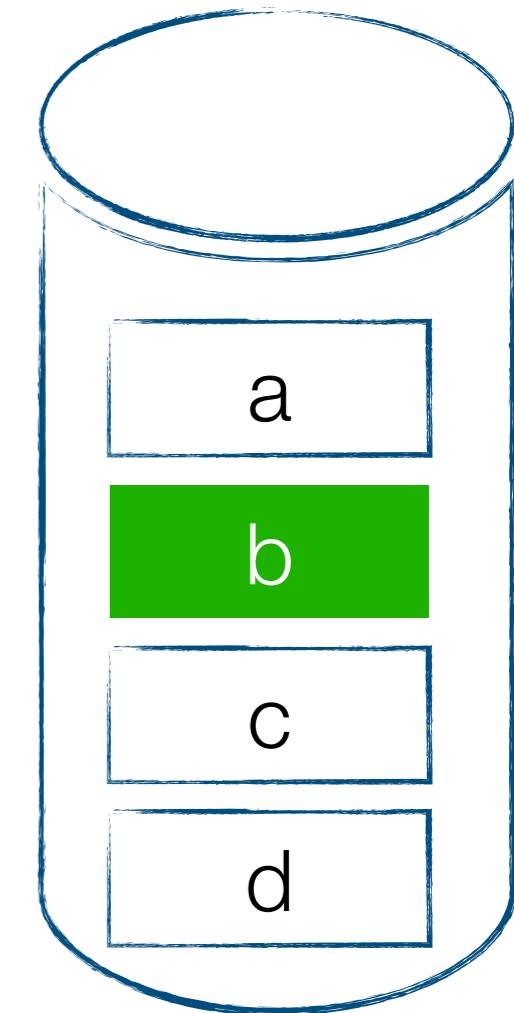
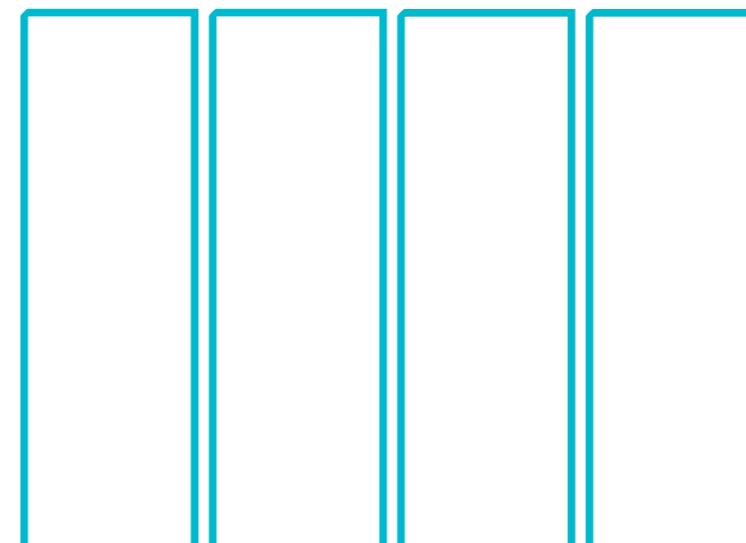
Execution Priority
Invariance

Priority Groups

Low-priority
Queues



High-priority
Queues

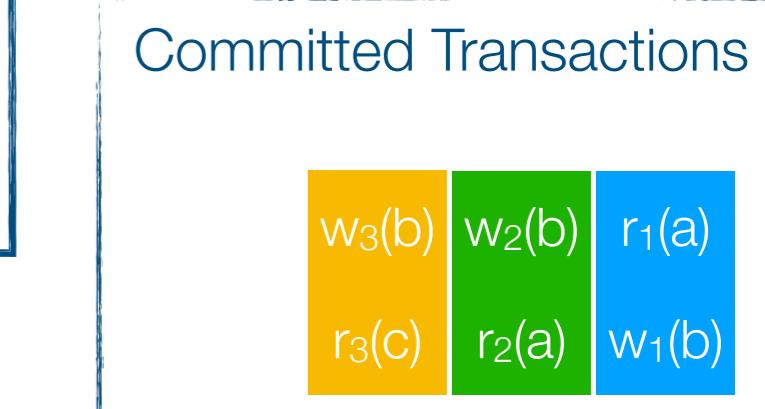
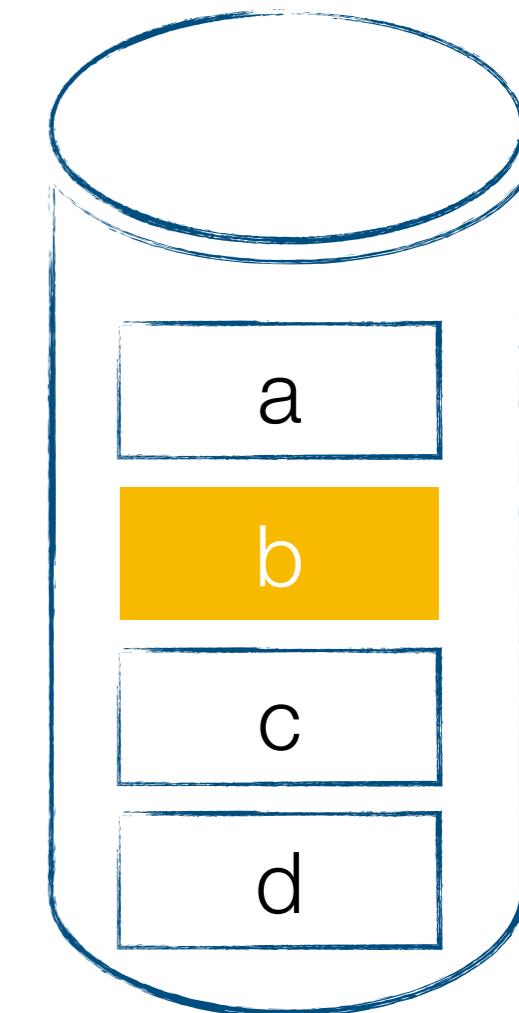
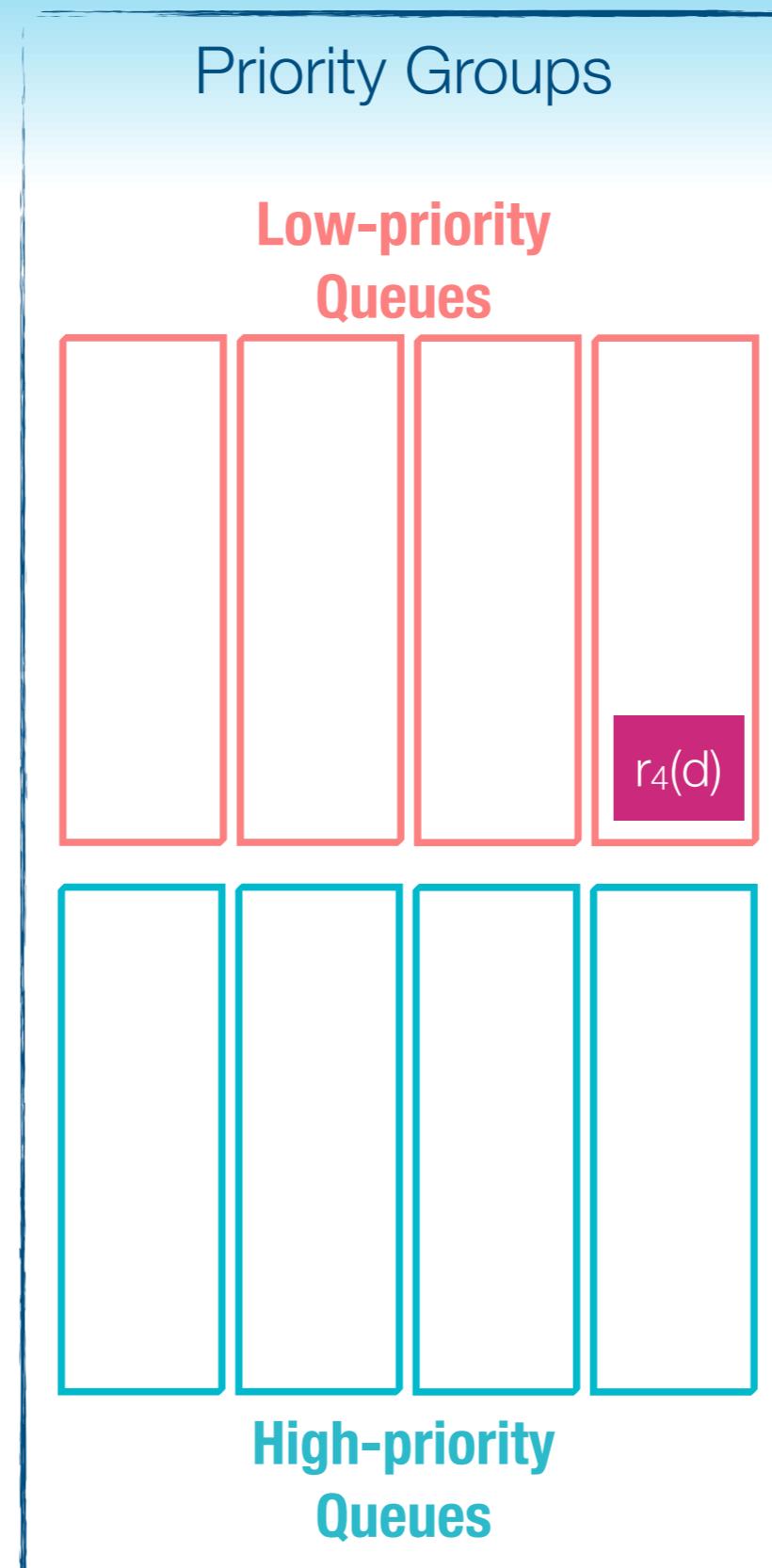
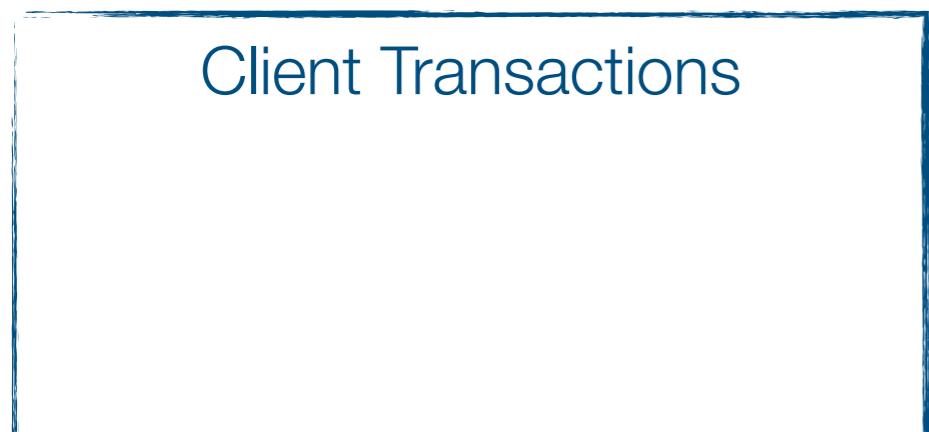


Committed Transactions

w₂(b) r₁(a)
r₂(a) w₁(b)

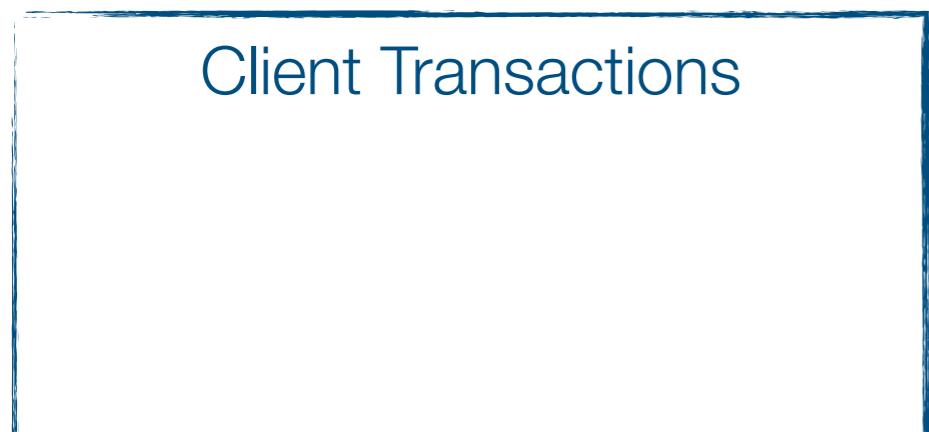
QueCC

Abort Count: 0



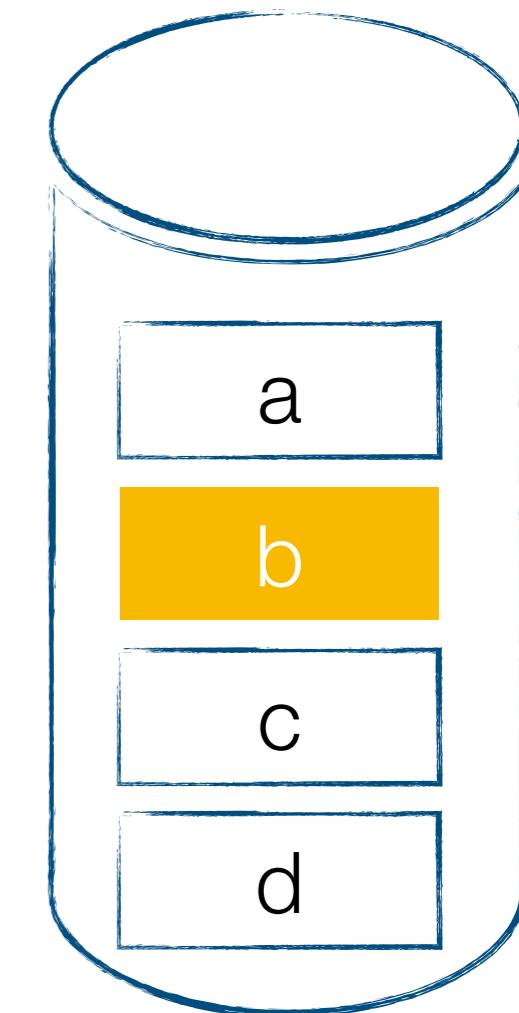
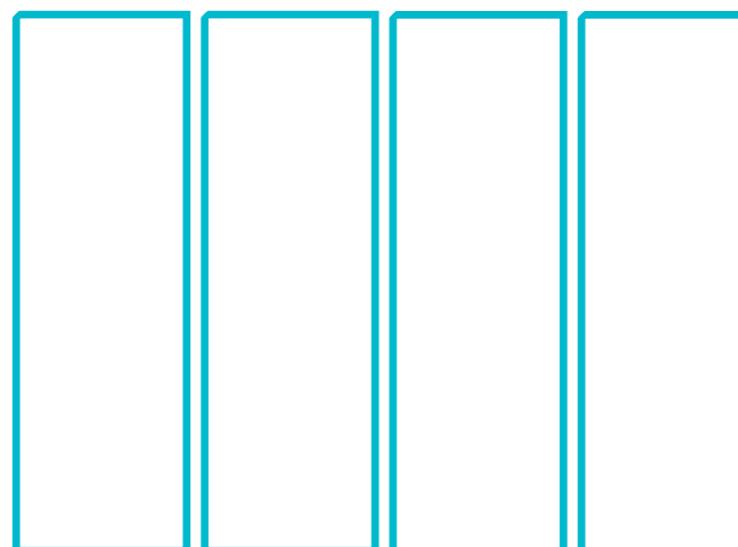
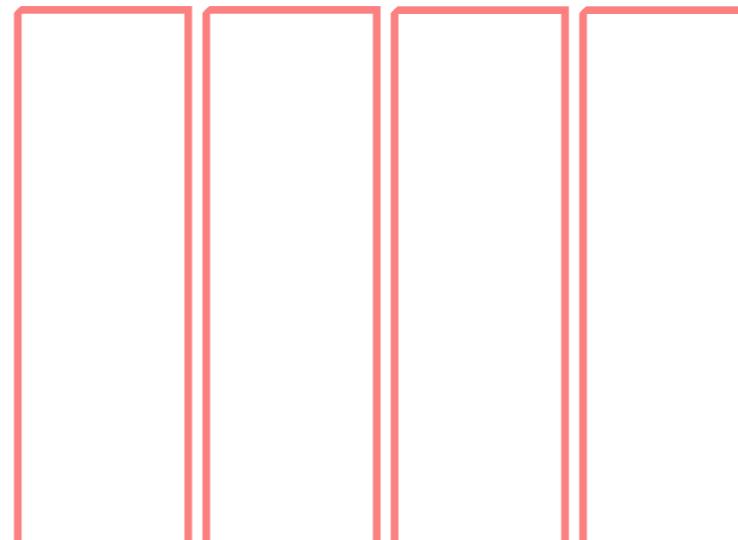
QueCC

Abort Count: 0



Priority Groups

Low-priority Queues

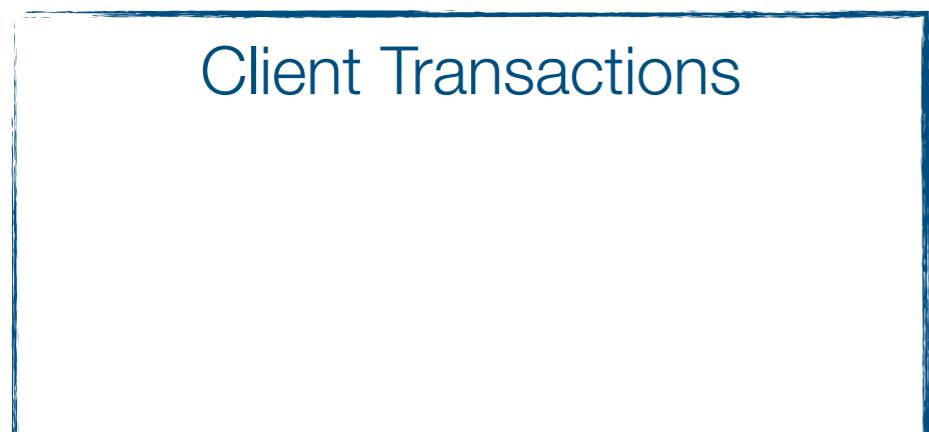


Committed Transactions

w ₃ (b)	w ₂ (b)	r ₁ (a)
r ₃ (c)	r ₂ (a)	w ₁ (b)

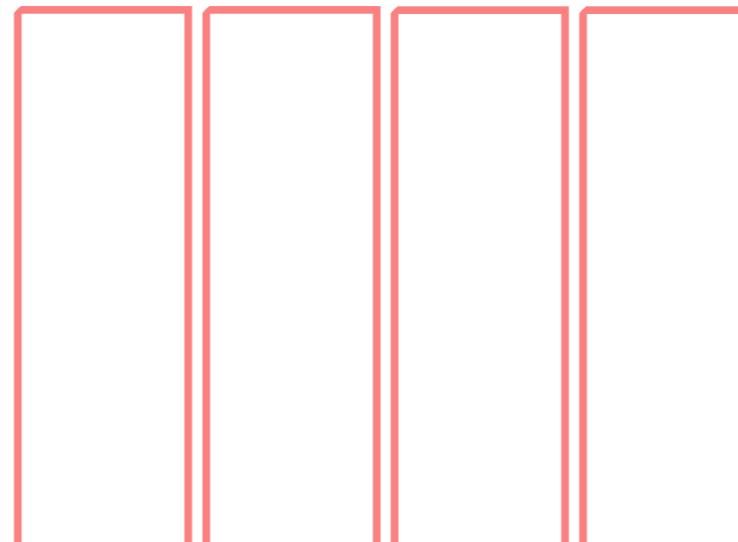
QueCC

Abort Count: 0

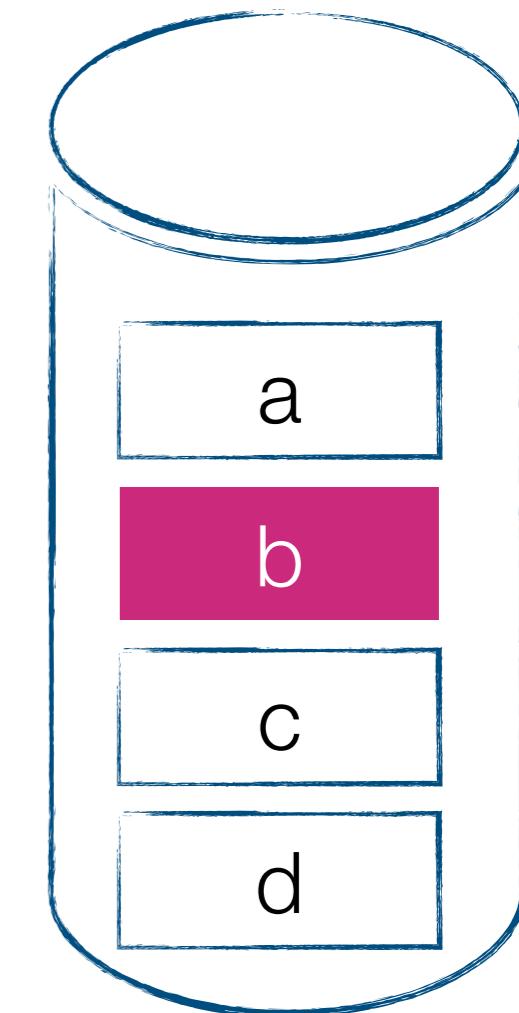
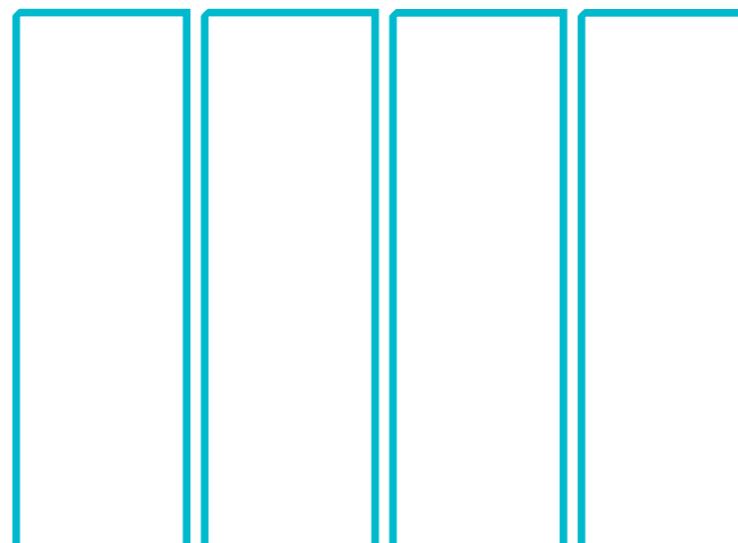


Priority Groups

Low-priority Queues



High-priority Queues



Committed Transactions

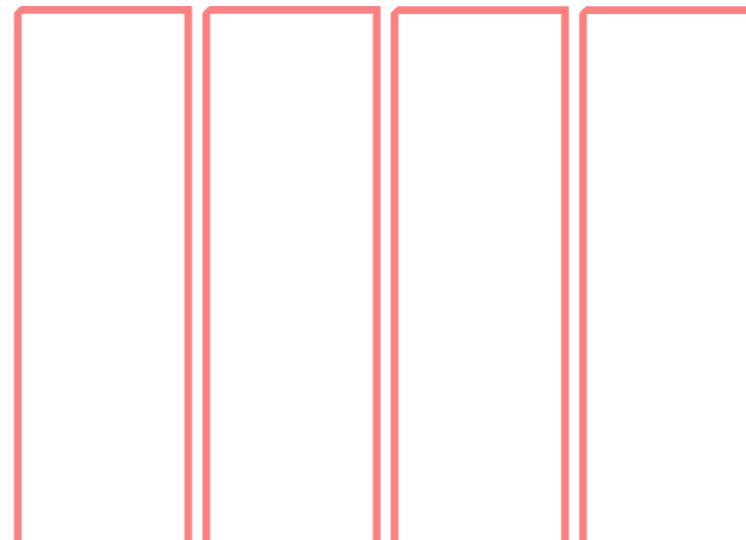
w ₄ (b)	w ₃ (b)	w ₂ (b)	r ₁ (a)
r ₄ (d)	r ₃ (c)	r ₂ (a)	w ₁ (b)



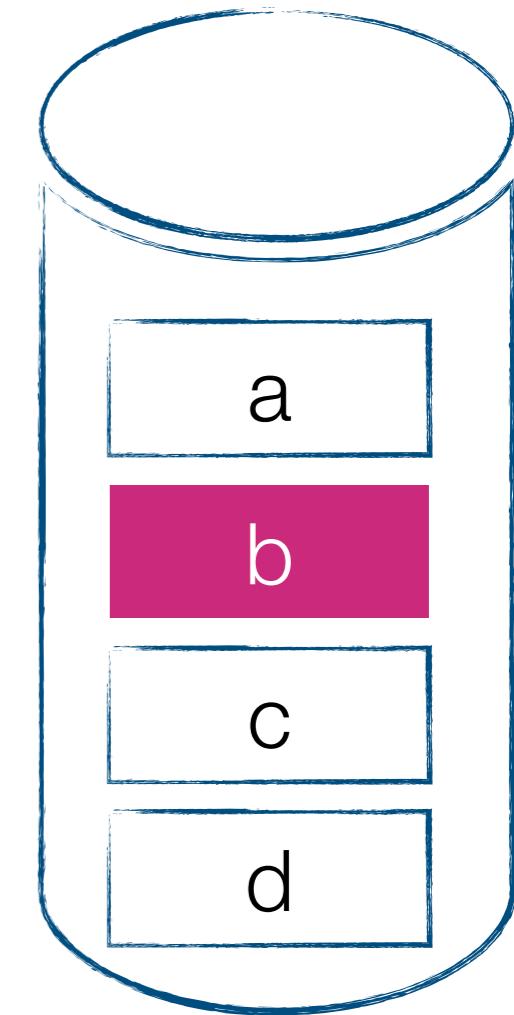
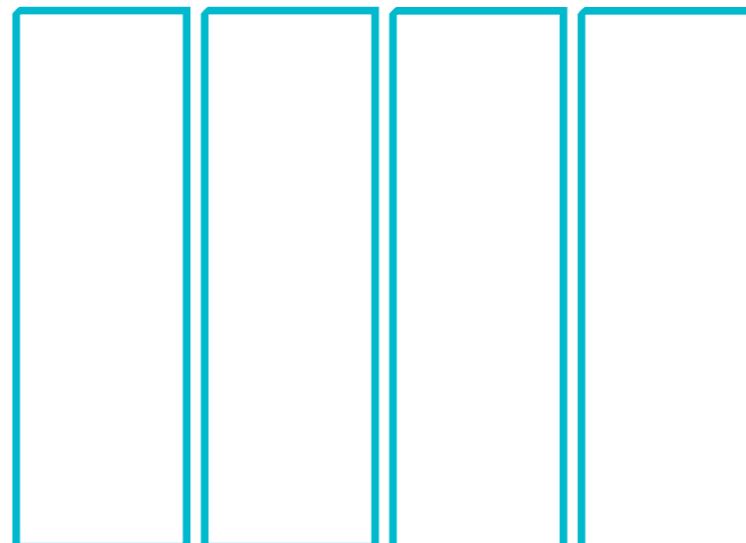
- ✓ Deterministic Execution
- ✓ No aborts because of CC
- ✓ Minimal coordination among threads
- ✓ Not sensitive to multi-partition transactions
- ✓ Exploits Intra-transaction parallelism

Priority Groups

Low-priority Queues



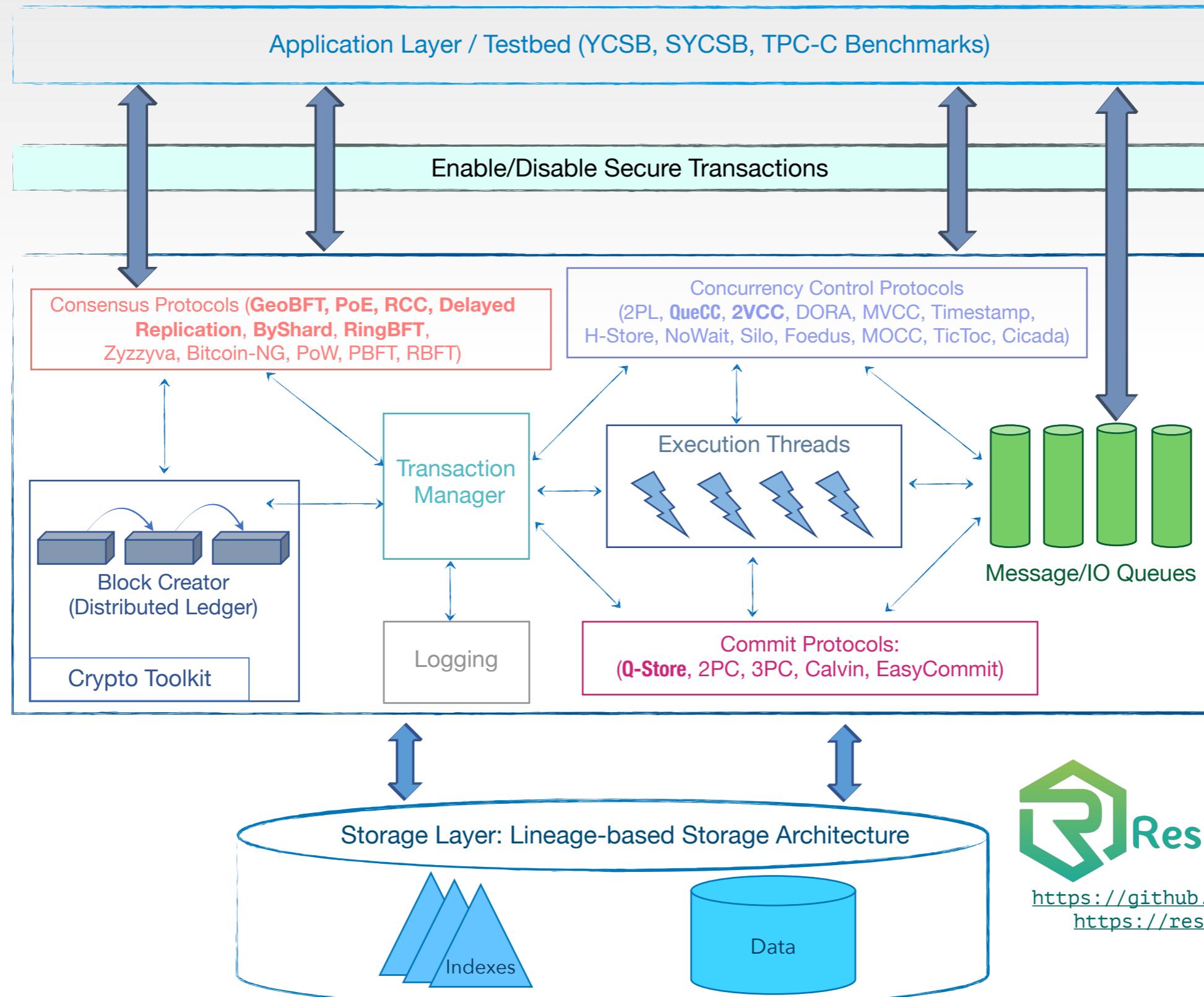
High-priority Queues



Committed Transactions

w ₄ (b)	w ₃ (b)	w ₂ (b)	r ₁ (a)
r ₄ (d)	r ₃ (c)	r ₂ (a)	w ₁ (b)

ResilientDB Blockchain Fabric



 **ResilientDB**
<https://github.com/resilientdb/>
<https://resilientdb.com/>

Evaluation Environment

Hardware

Microsoft Azure instance with 32 core
CPU: Intel Xeon E5-2698B v3
32KB L1 data and instruction caches
256KB L2 cache
40MB L3 cache
RAM: 448GB

Workload

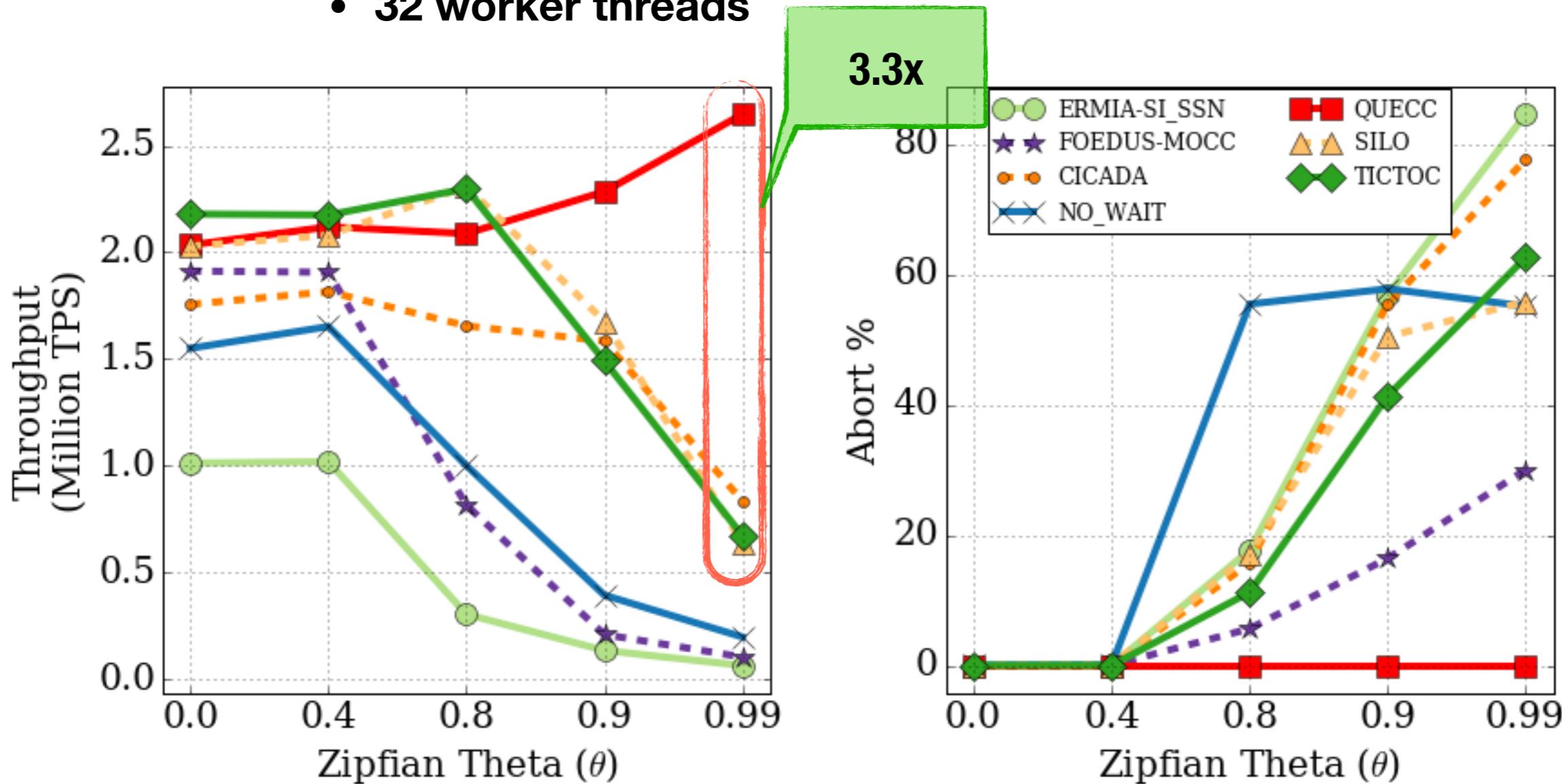
YCSB: 1 table, 10 operations, 50% RMW, Zipfian distribution
TPCC: 9 tables, Payment and NewOrder, 1 Warehouse

Software

Operating System: Ubuntu LTS 16.04.3
Compiler: GCC with -O3 compiler optimizations

Effect of Varying Contention

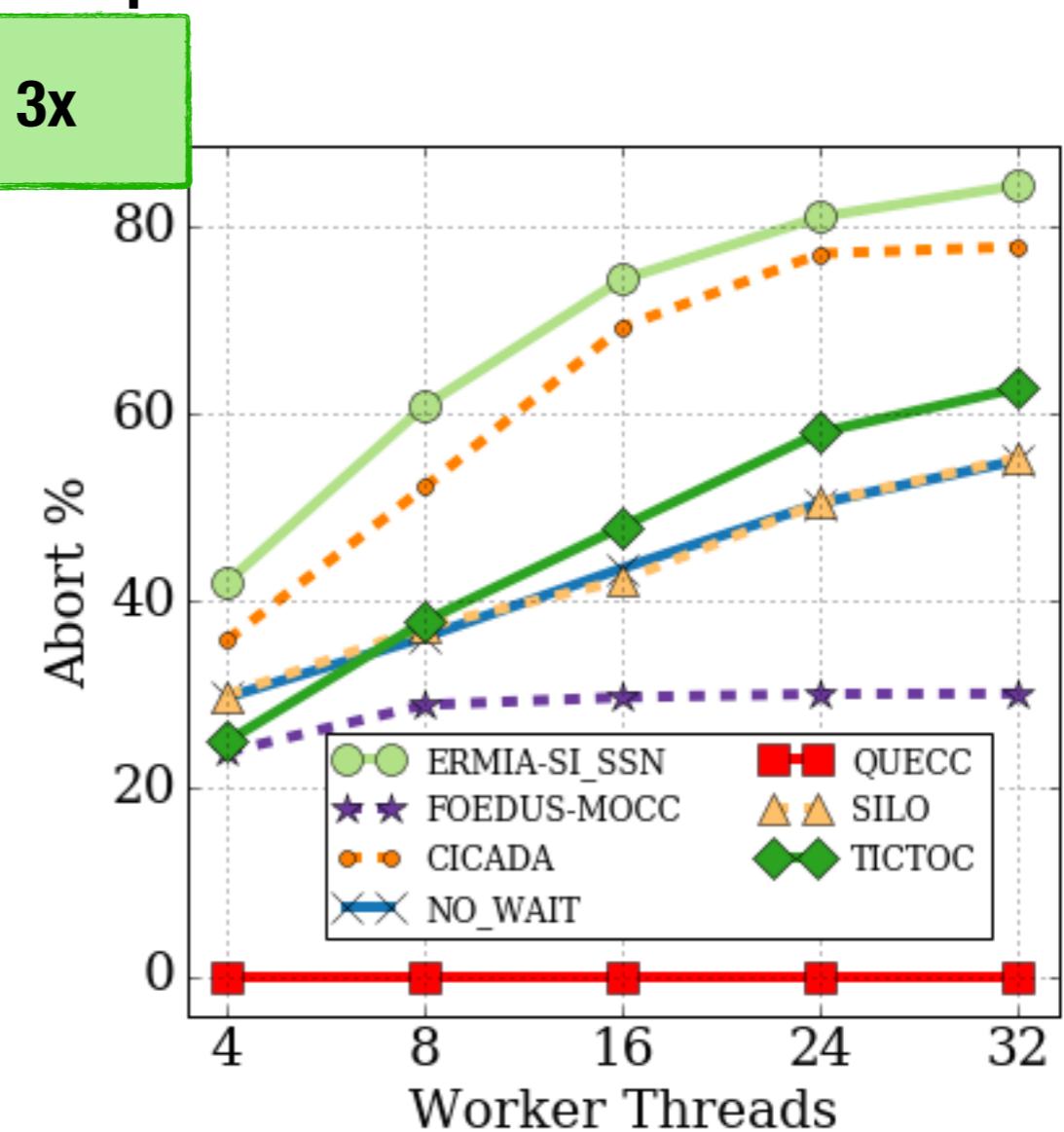
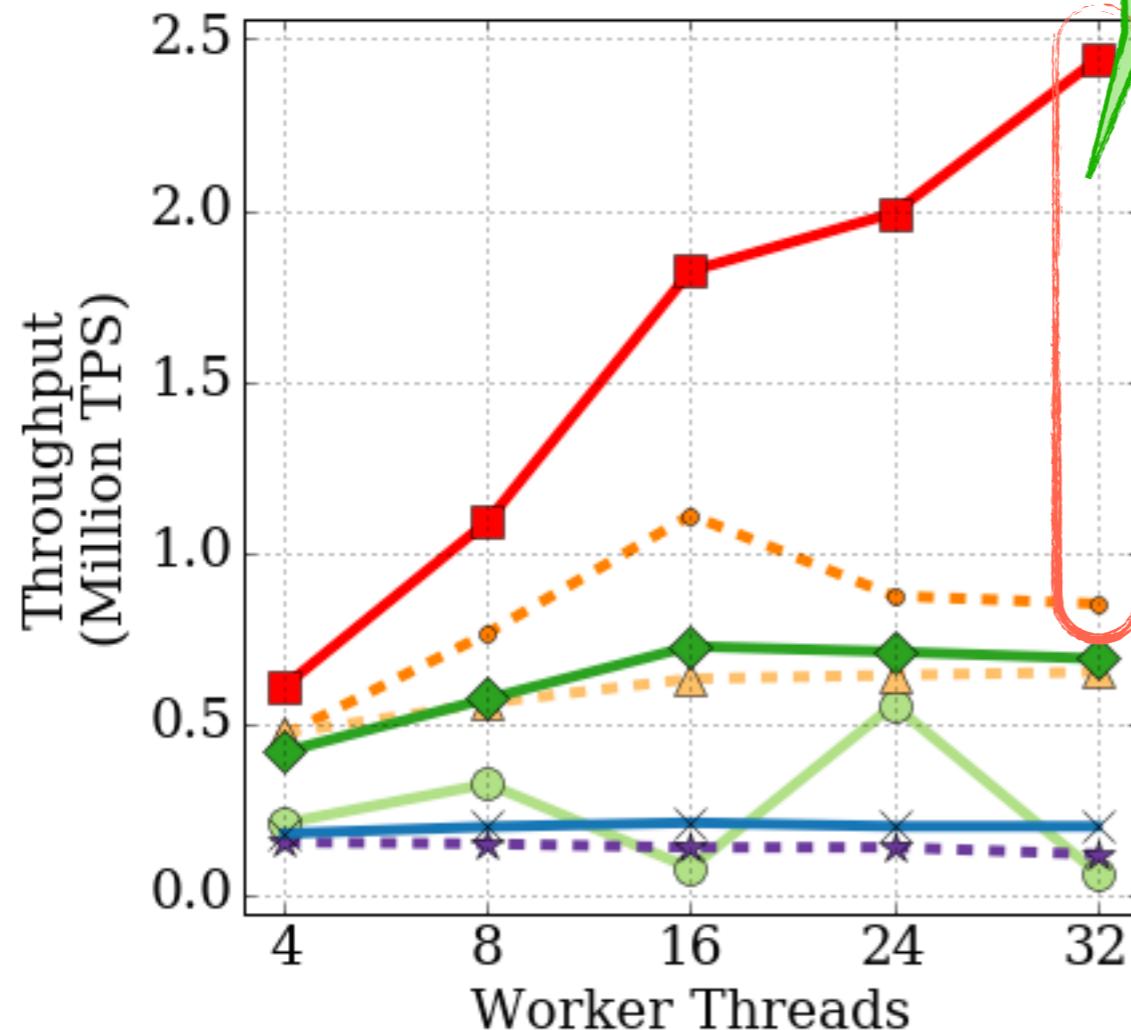
- 5 write and 5 read operation per transaction
- 32 worker threads



Workload contention resiliency
Cache locality under high-contention

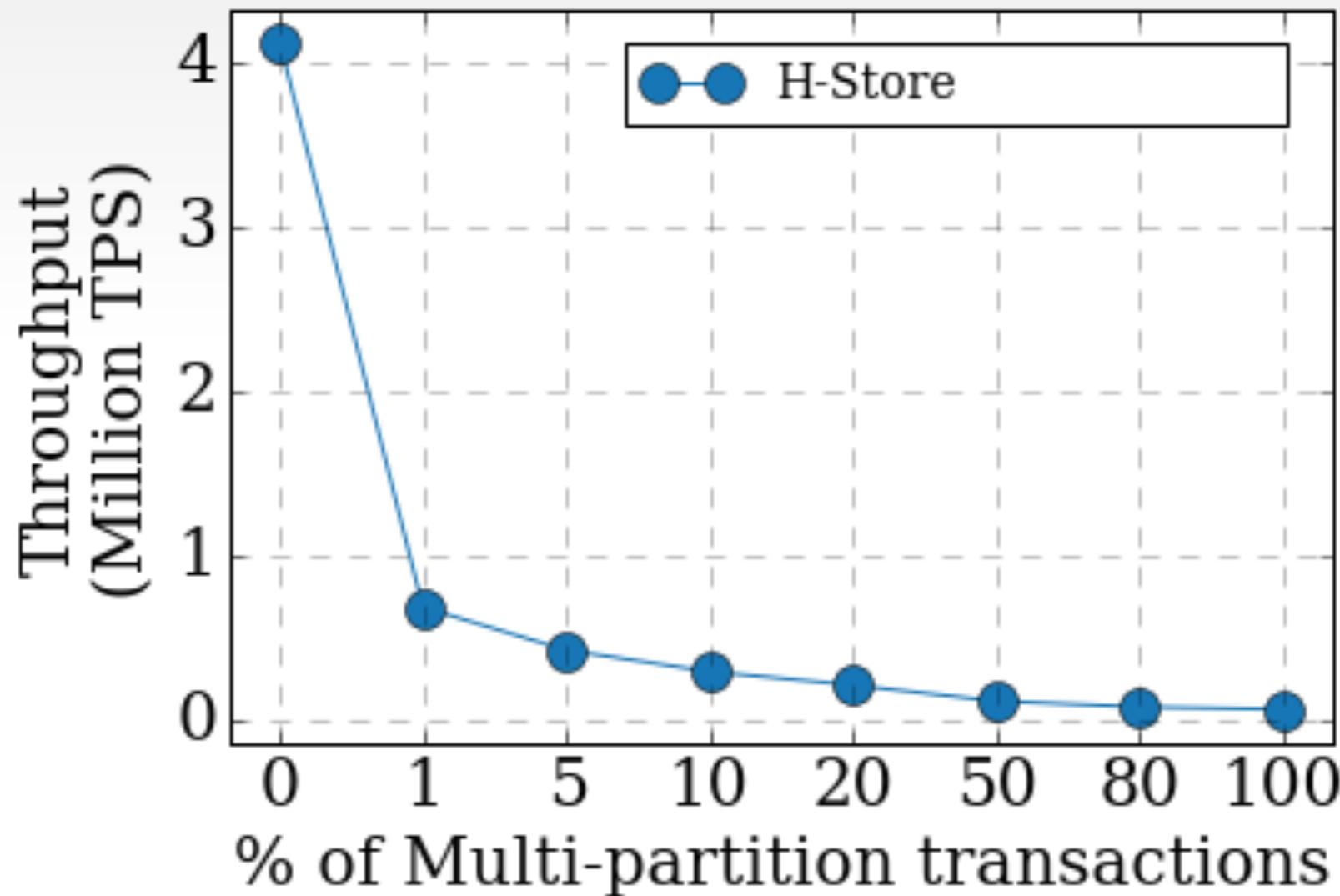
Effect of Varying Worker Threads

- 5 write and 5 read operation per transaction
- Zipfian theta = 0.99

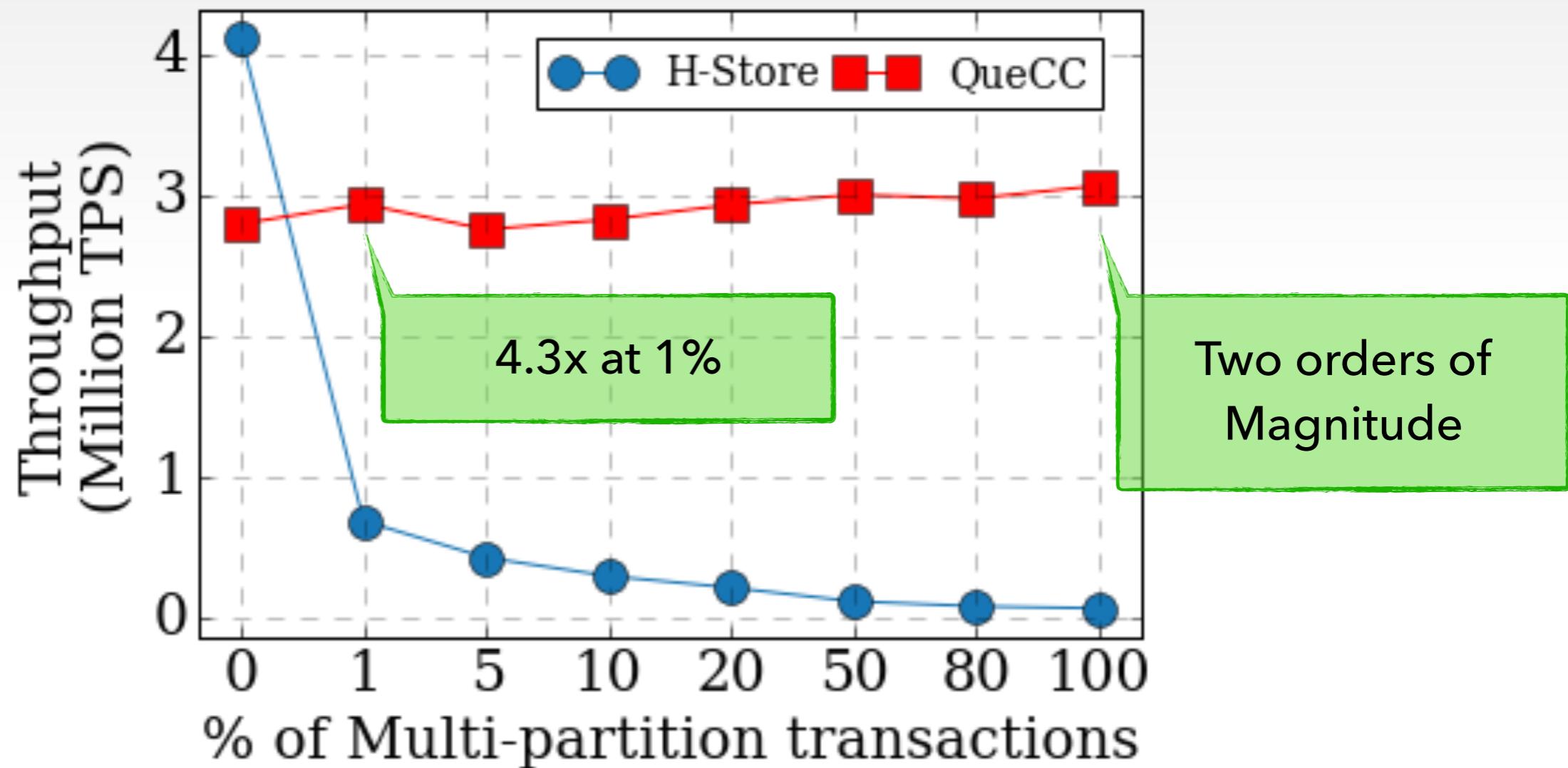


Avoiding thread coordination & eliminating all execution-induced aborts

Effect of Increasing Percentage of Multi-Partition Transactions in the Workload



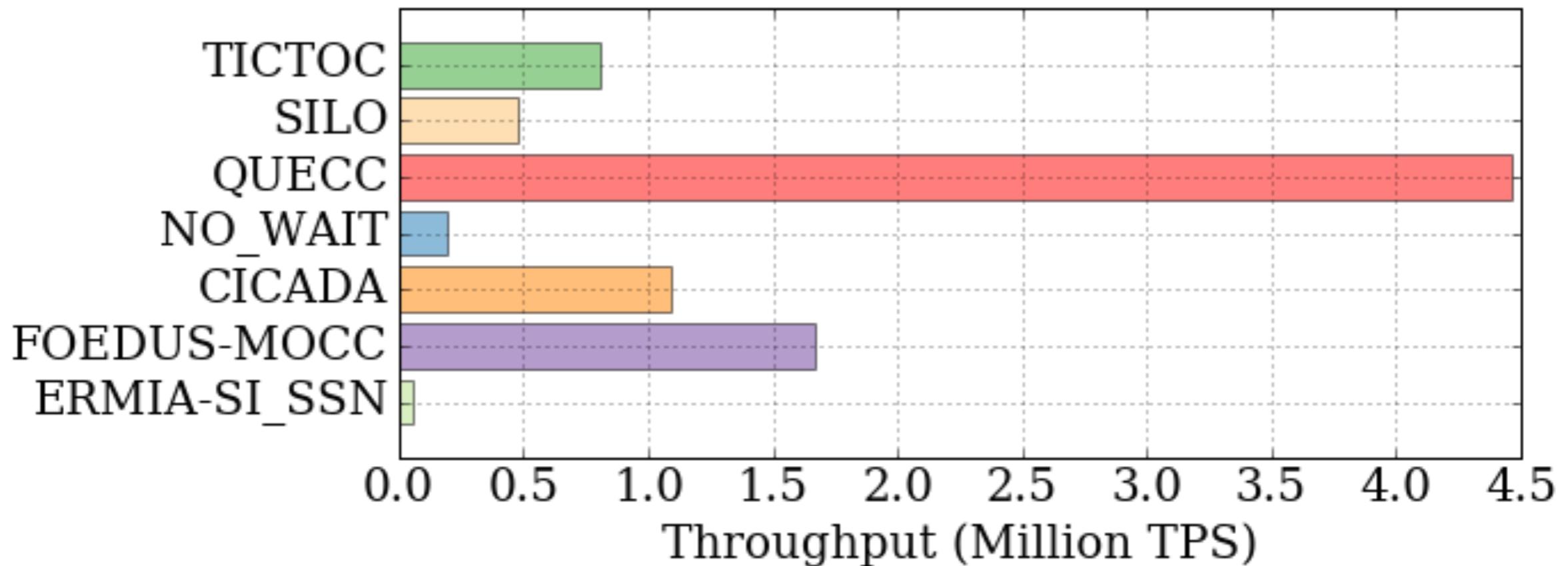
Effect of Increasing Percentage of Multi-Partition Transactions in the Workload



QueCC is not sensitive to multi-partitioning

TPC-C Results

1 Warehouse (highly contended workload)
50% Payment + 50% NewOrder transaction mix



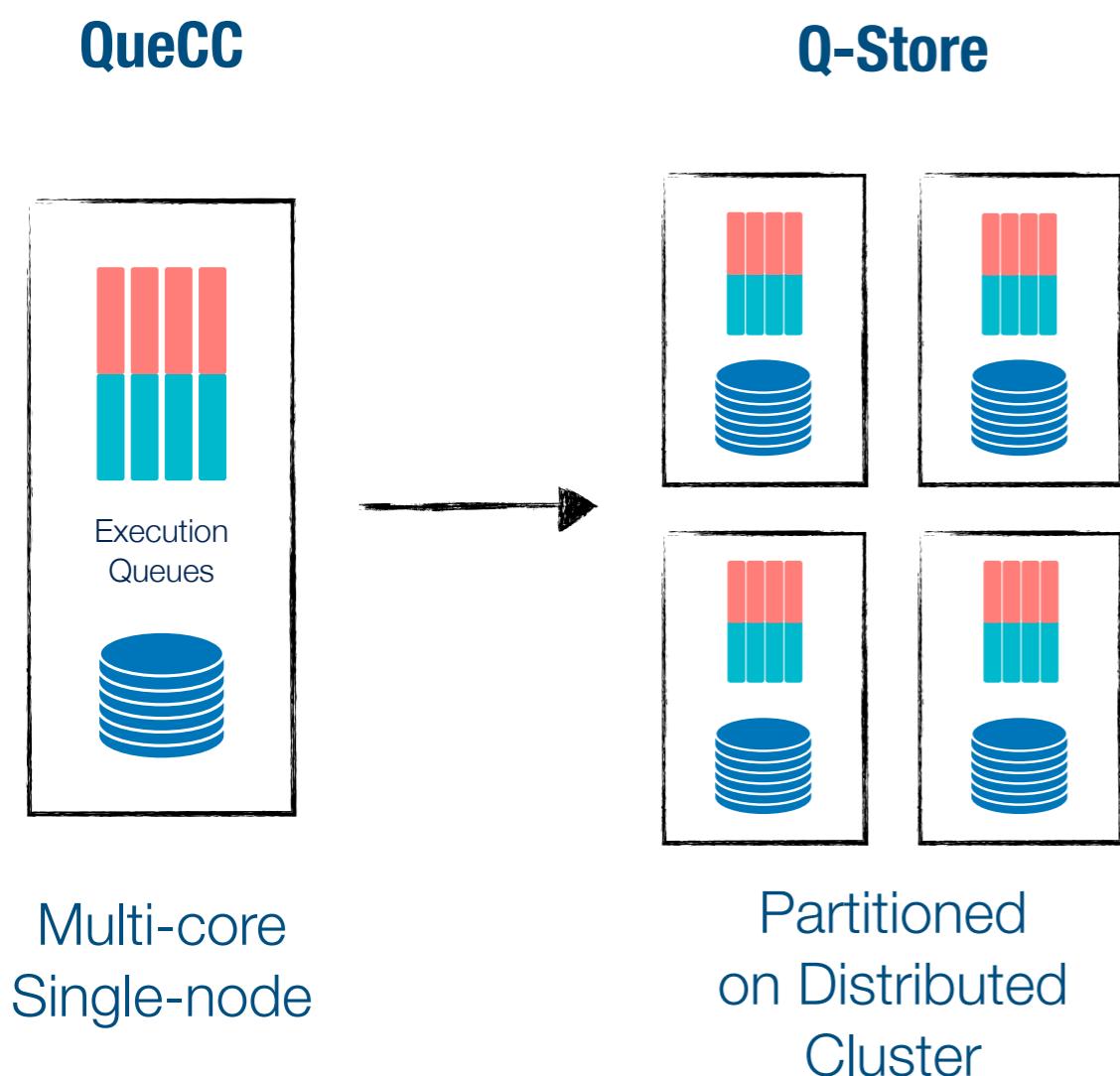
QueCC can achieve up to 3x better performance on high-contention
TPC-C workloads

QueCC Conclusions

- ✓ Efficient, parallel and deterministic in-memory transaction processing
- ✓ Eliminates almost all aborts by resolving transaction conflicts *a priori*
- ✓ Works extremely well under high-contention workloads



What's Next: Q-Store

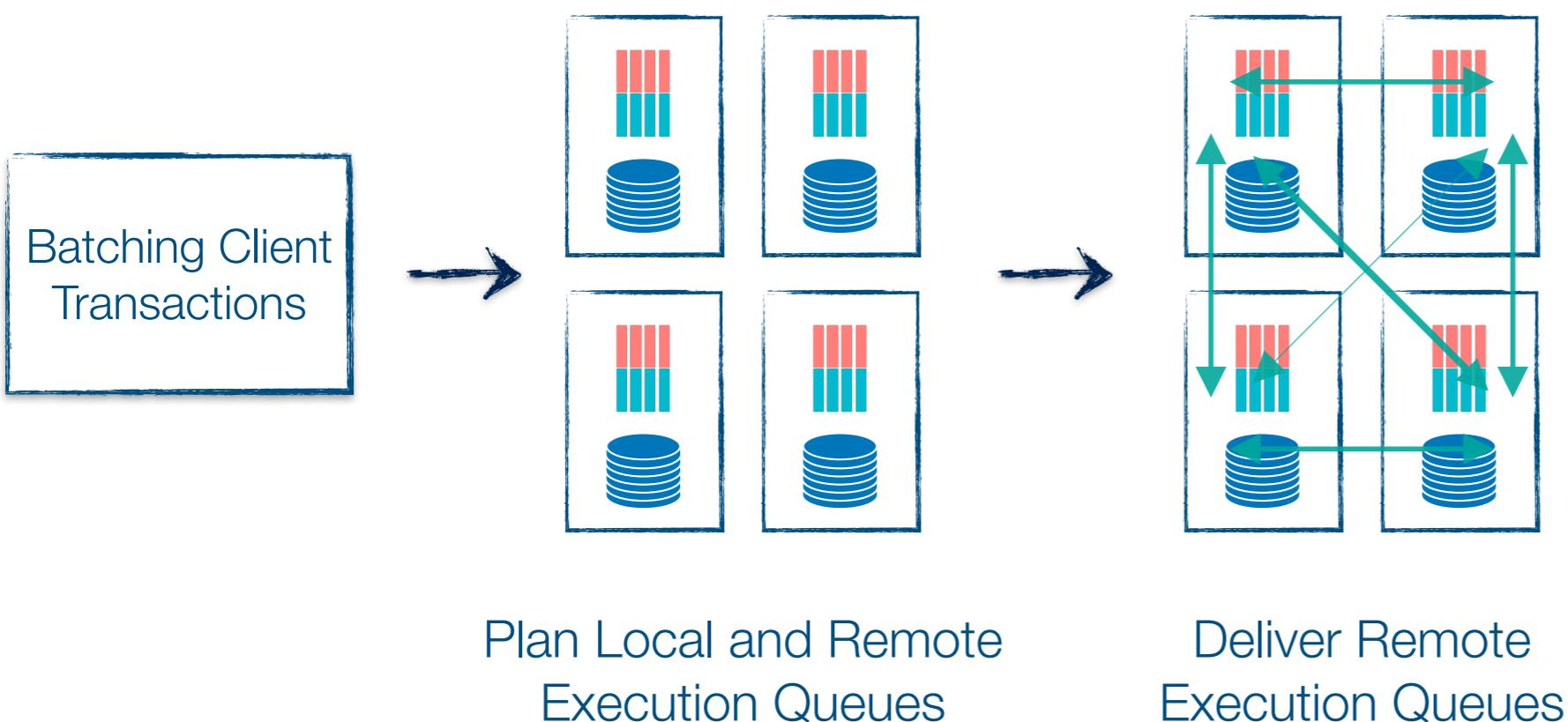


What's Next: Q-Store



Plan Local and Remote
Execution Queues

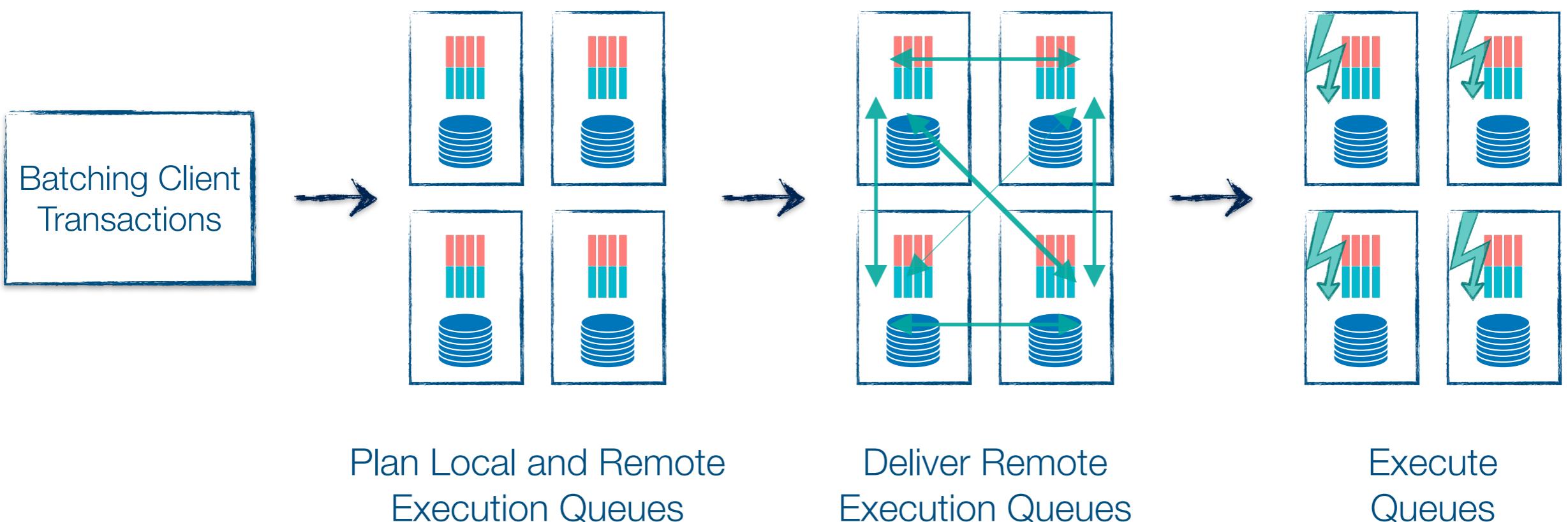
What's Next: Q-Store



Plan Local and Remote
Execution Queues

Deliver Remote
Execution Queues

What's Next: Q-Store

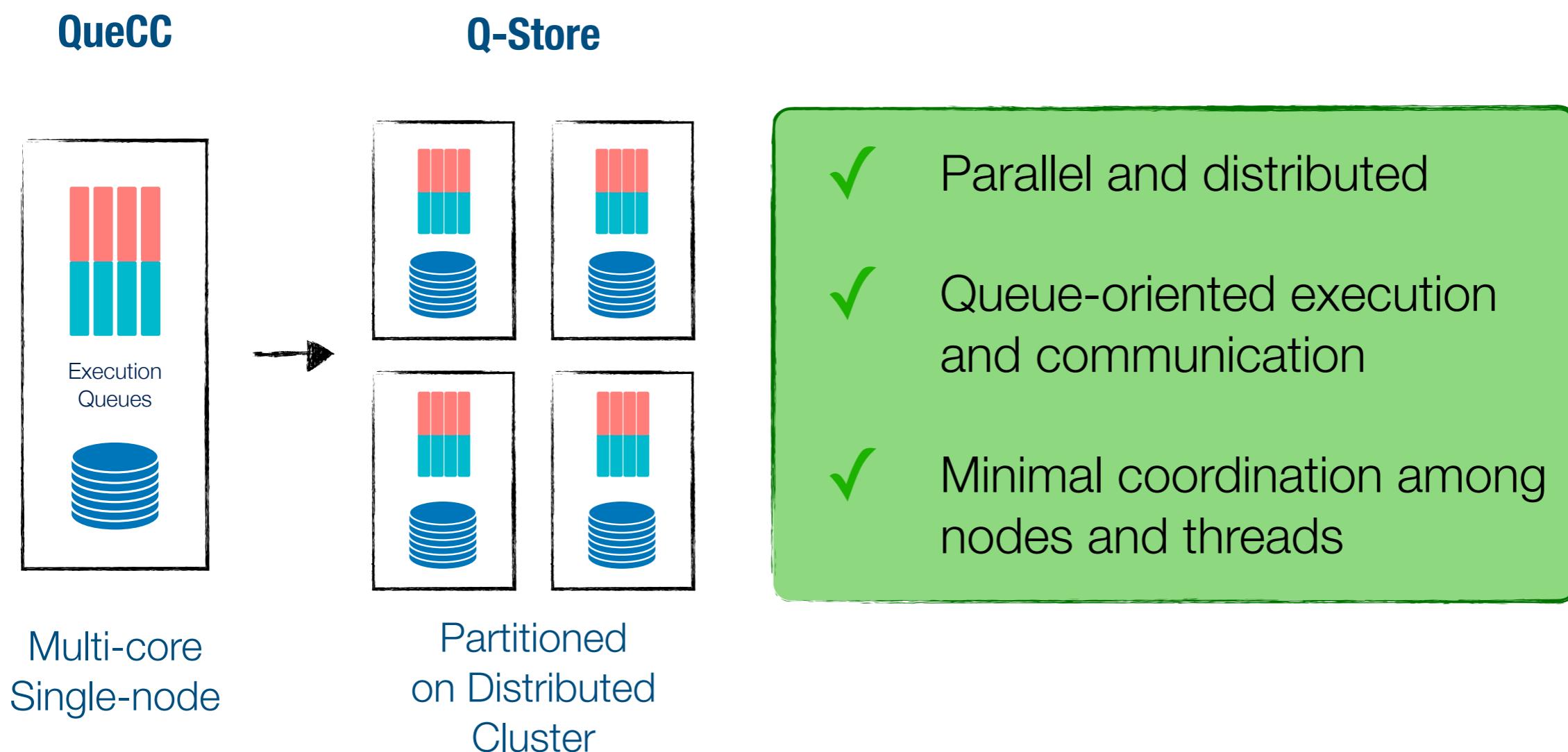


Plan Local and Remote
Execution Queues

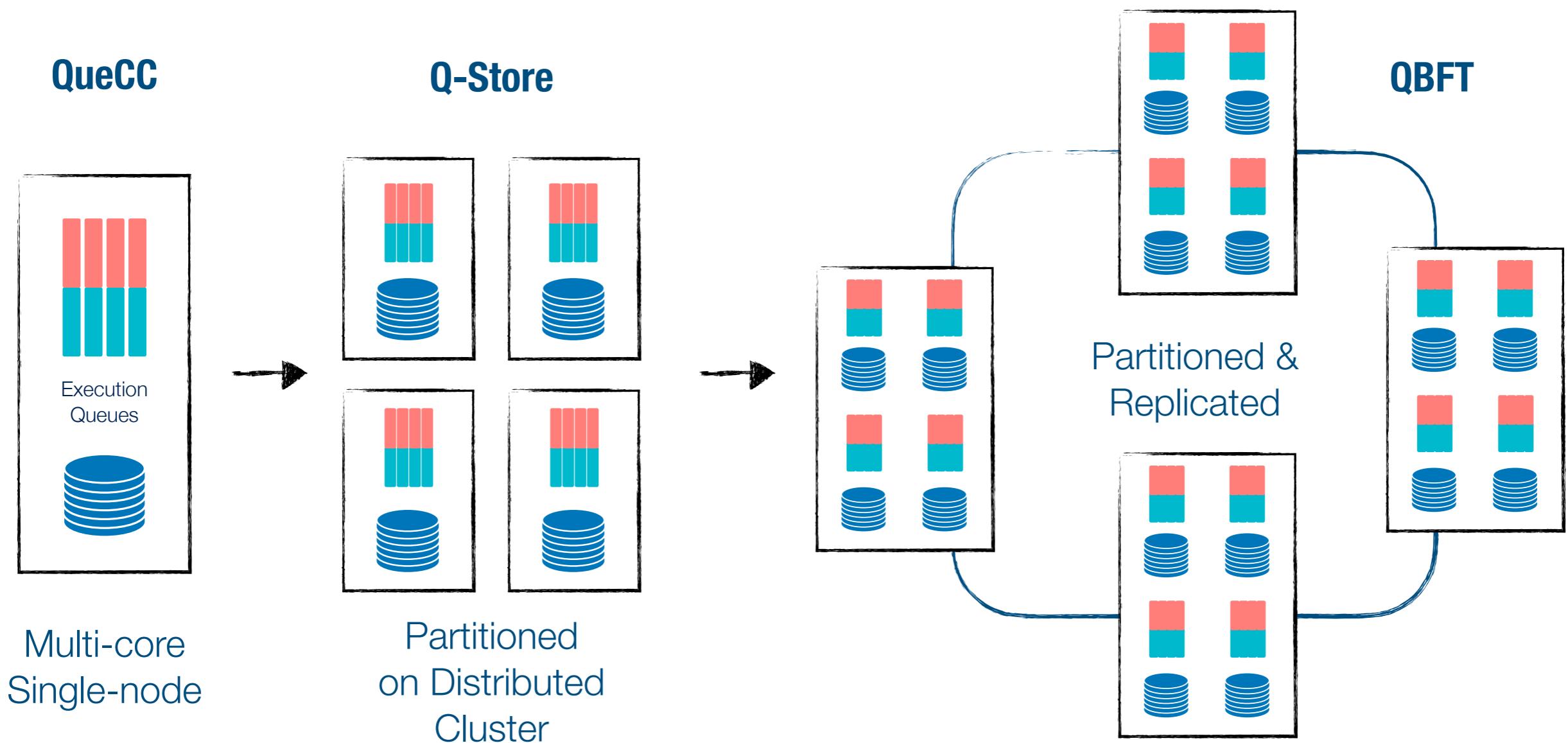
Deliver Remote
Execution Queues

Execute
Queues

What's Next: Q-Store



What's Next: QBFT



What's Next: QBFT



Queue-oriented
Byzantine Fault-
Tolerance



Resilient planning
followed by resilient
execution

