L-Store Concurrency Control: QueCC

Slides are adopted from Qadah, Sadoghi

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

ECS 165A – Winter 2022

Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
UCDAVIS
UNIVERSITY OF CALIFORNIA

ExpoLab
Creativity Unfolded

ResilientDB

Hardware Trends

Large core counts

Large main-memory


HPE Superdome Flex for SAP HANA Scale-out configuration

HPE Superdome Server
144 physical cores
6TB of RAM

***Image source: https://www.hpe.com/us/en/servers/superdome.html**

Popularity of Key-value Stores

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

High-Contention Workloads

Challenge ???

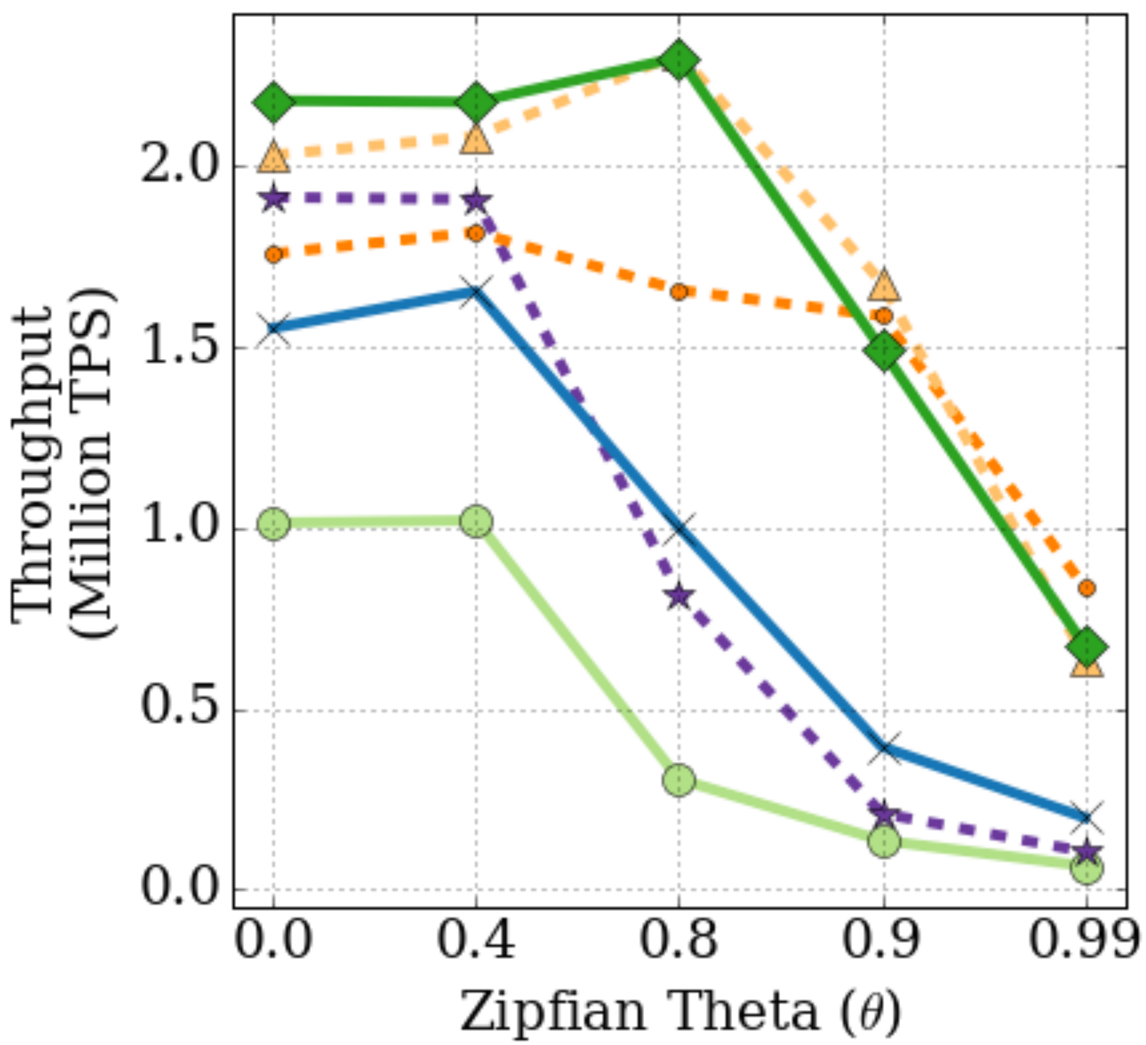
High number of
contented operations

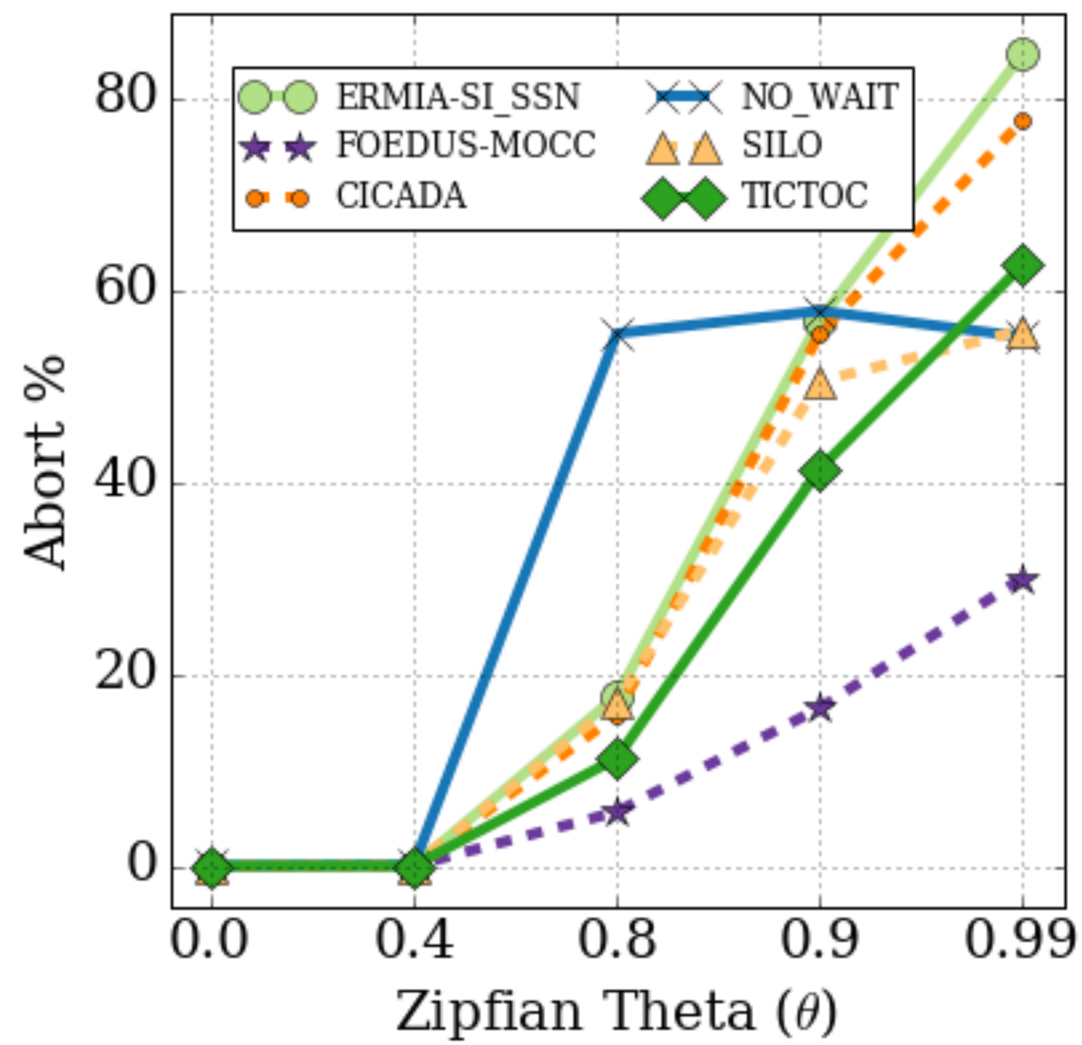
State-of-the-Art Concurrency Control Protocols

- 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

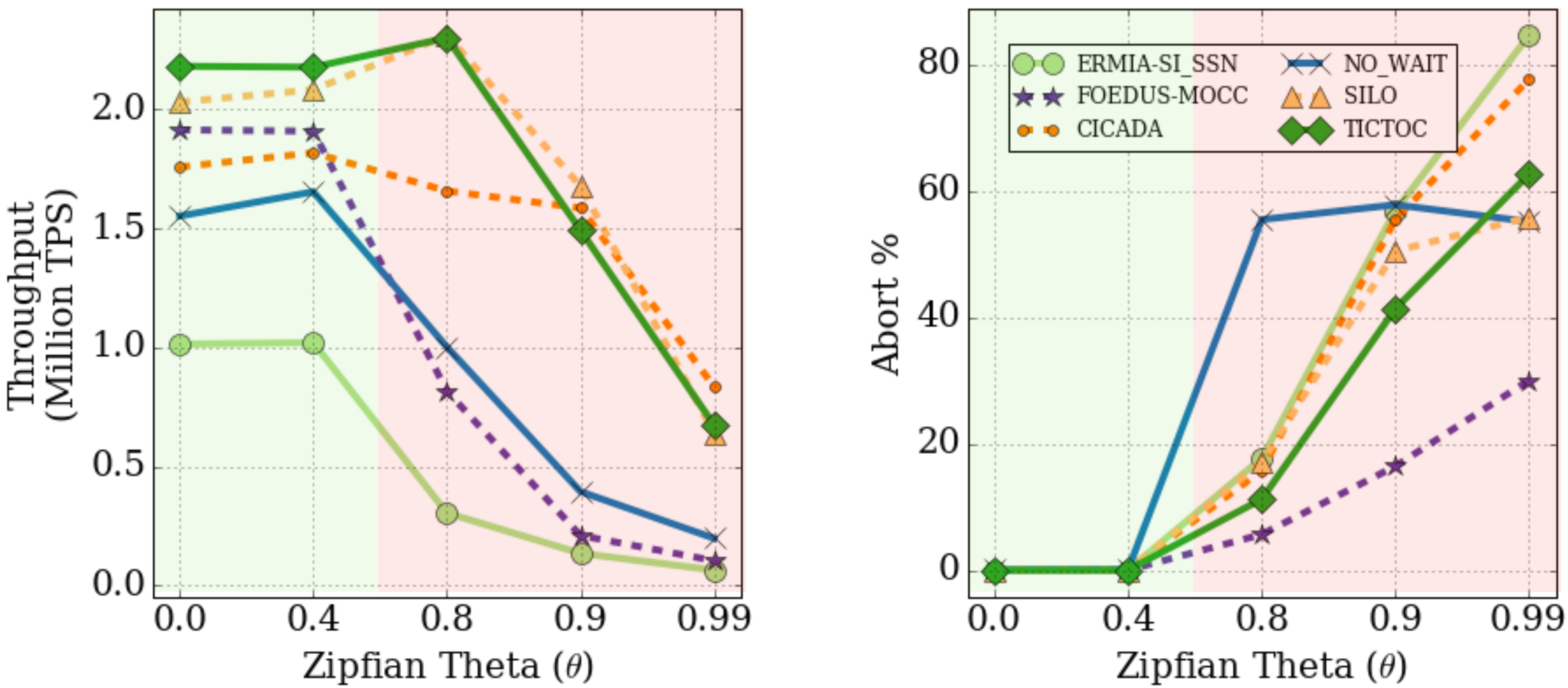
Performance Under High-Contention





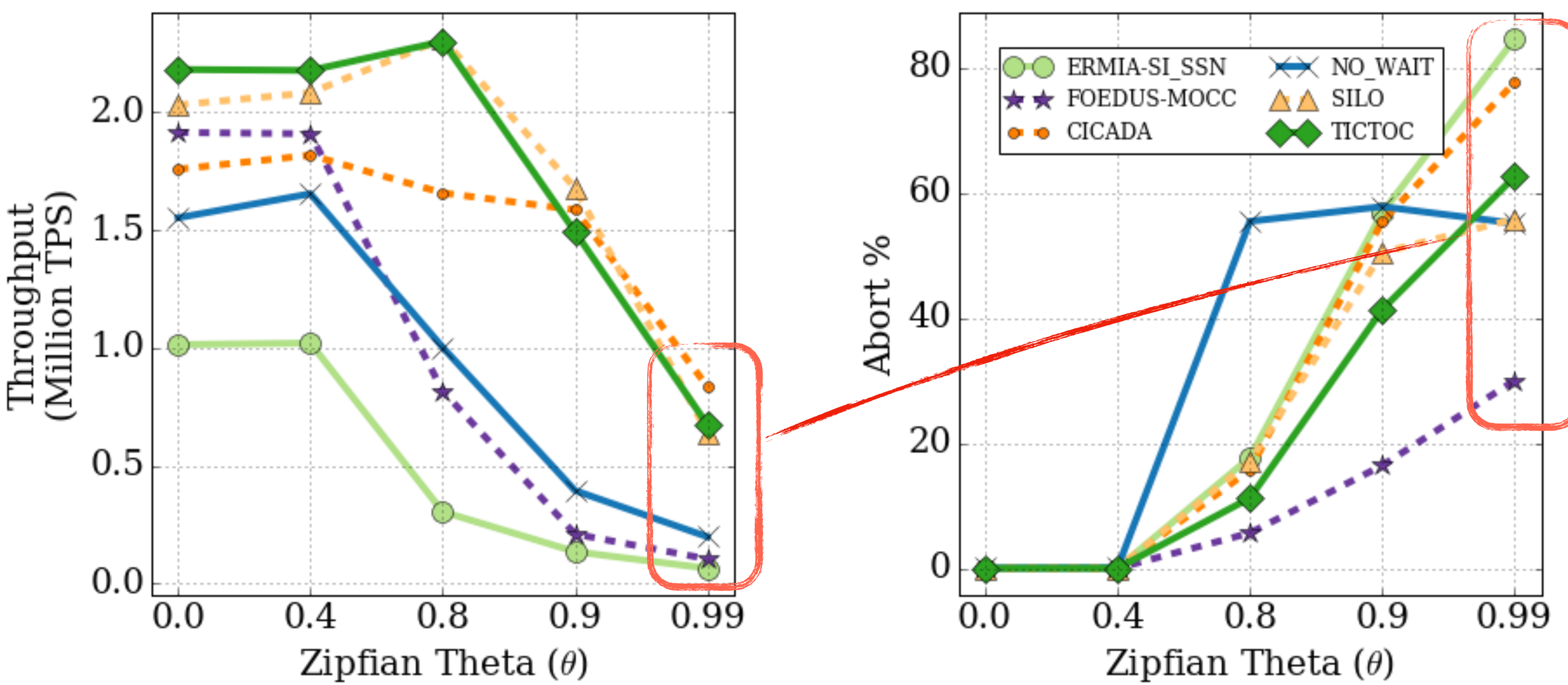
Optimize-for-multi-core concurrency control techniques suffer under high-contention due to increasing abort rate

Performance Under High-Contention

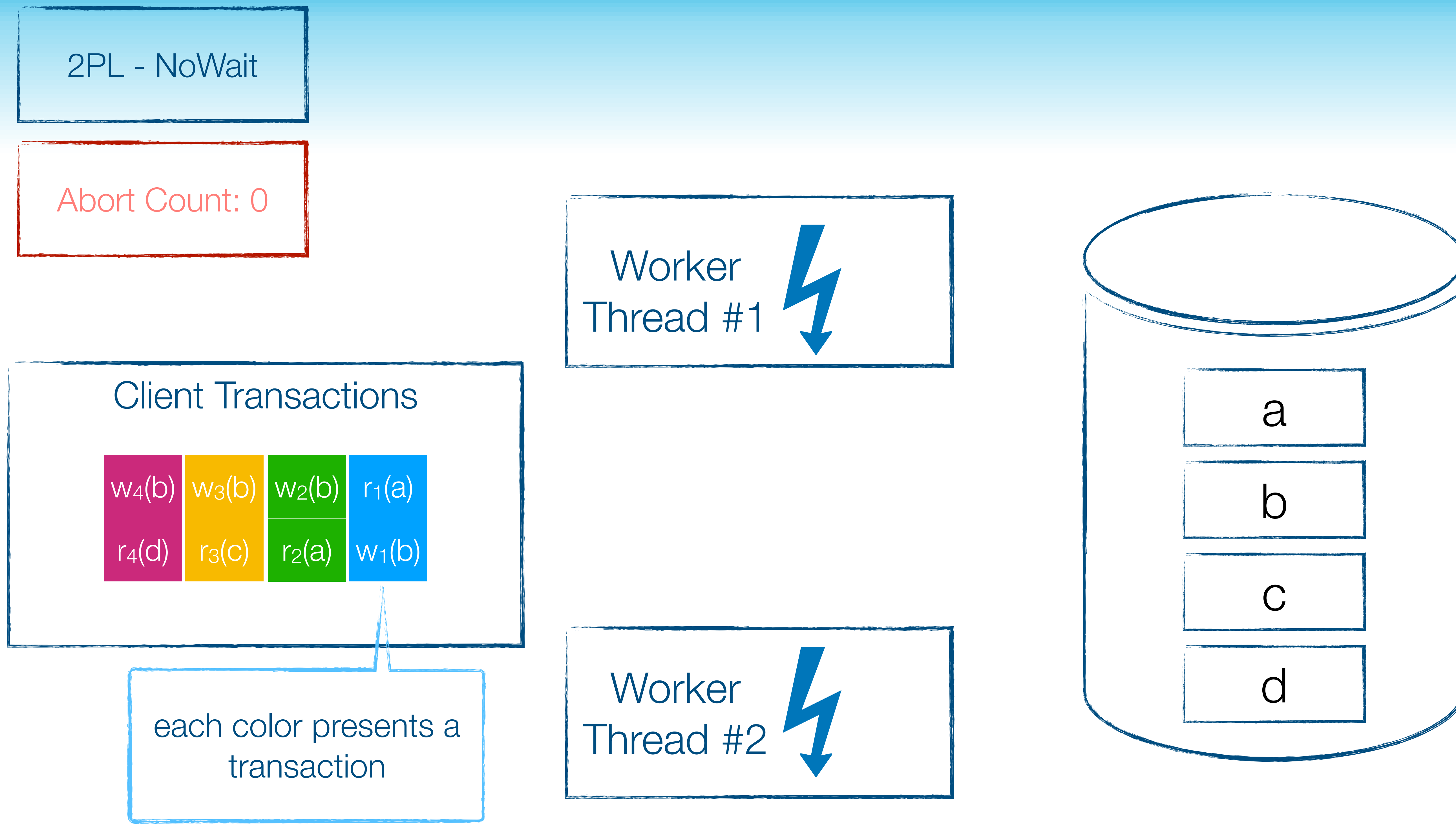


Under high-contention: Non-deterministic aborts dominates

Performance Under High-Contention

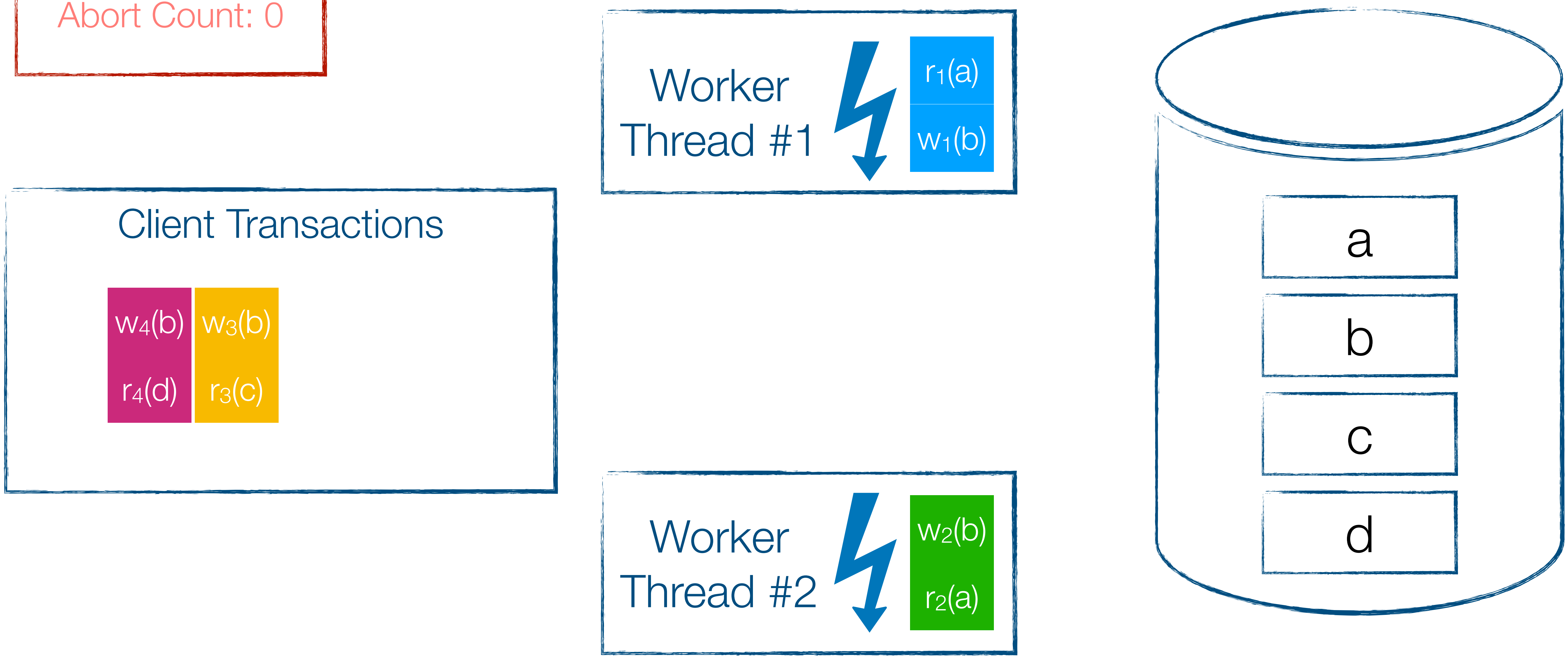


Under high-contention: Non-deterministic aborts dominates


2PL - NoWait
Abort Count: 0
Client Transactions
w4(b) w3(b) w2(b) r1(a)
r4(d) r3(c) r2(a) w1(b)
each color presents a transaction
Worker Thread #1
Worker Thread #2
a
b
c
d

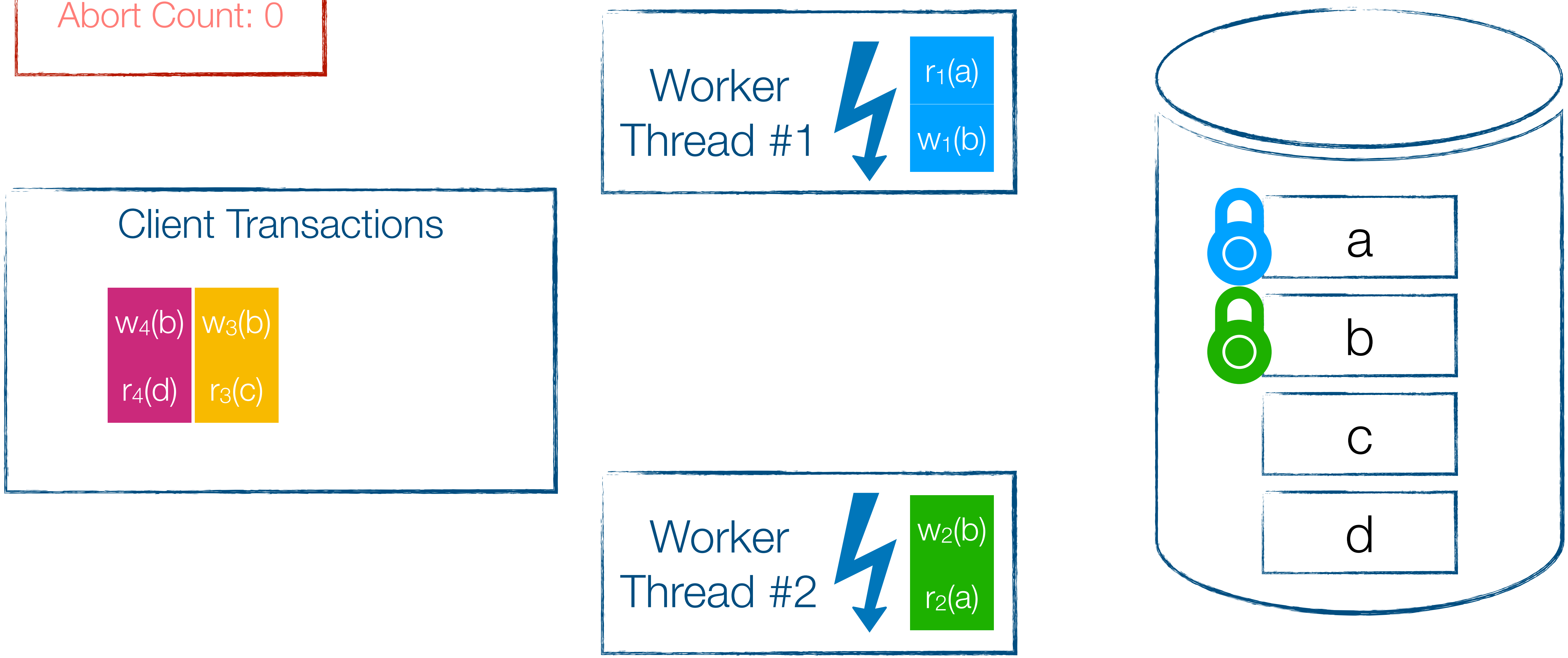
2PL - NoWait

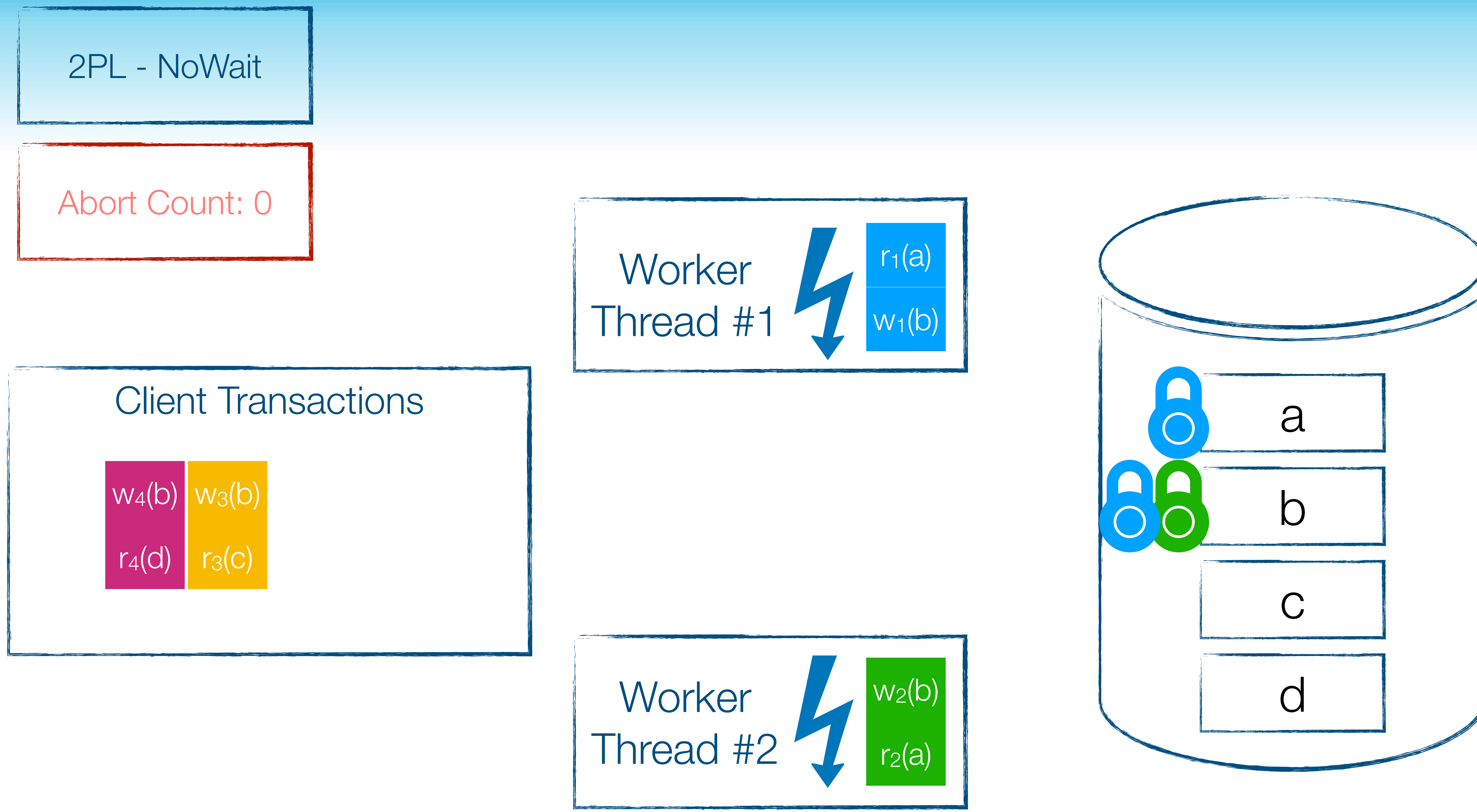
Abort Count: 0



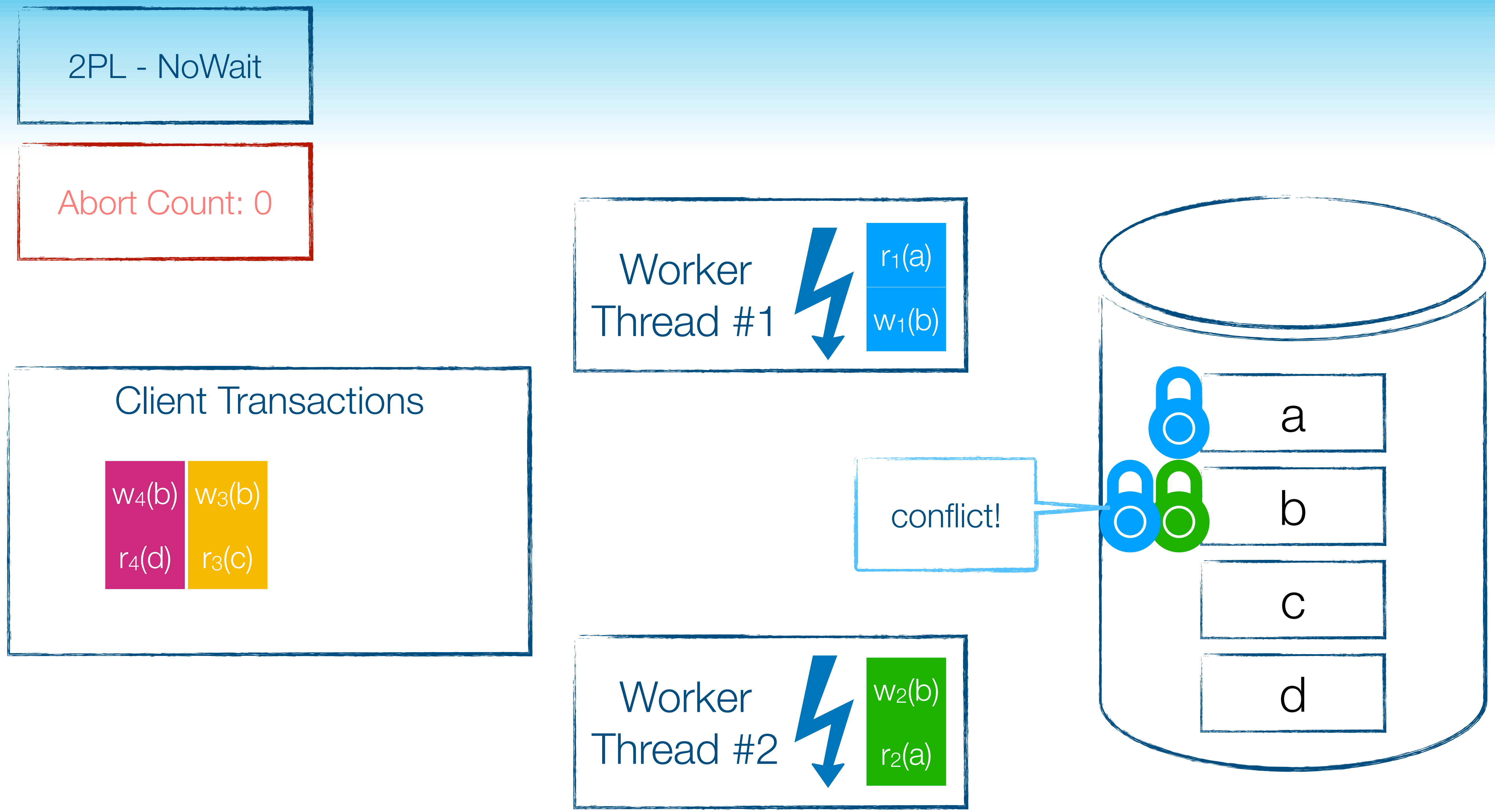
2PL - NoWait

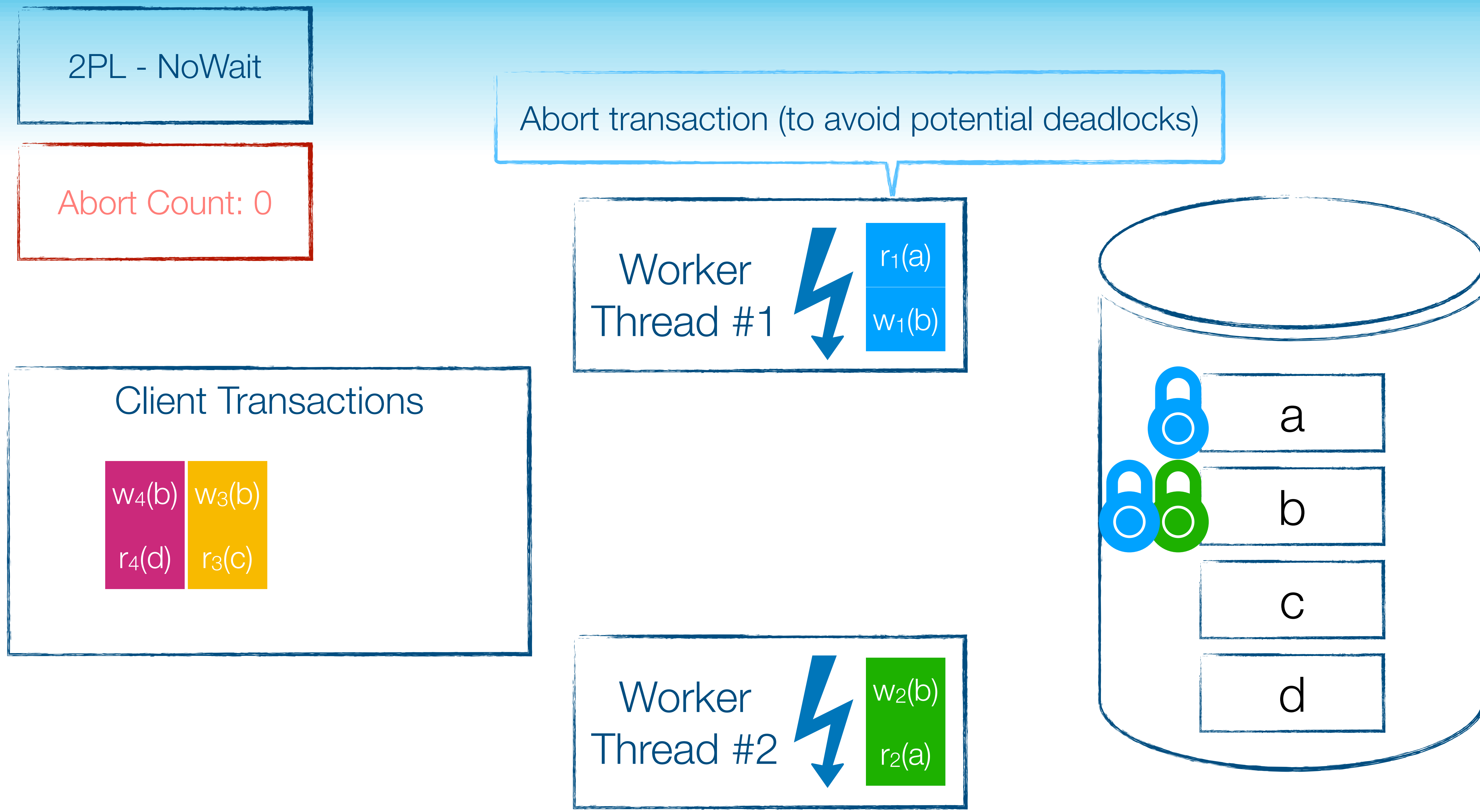
Abort Count: 0





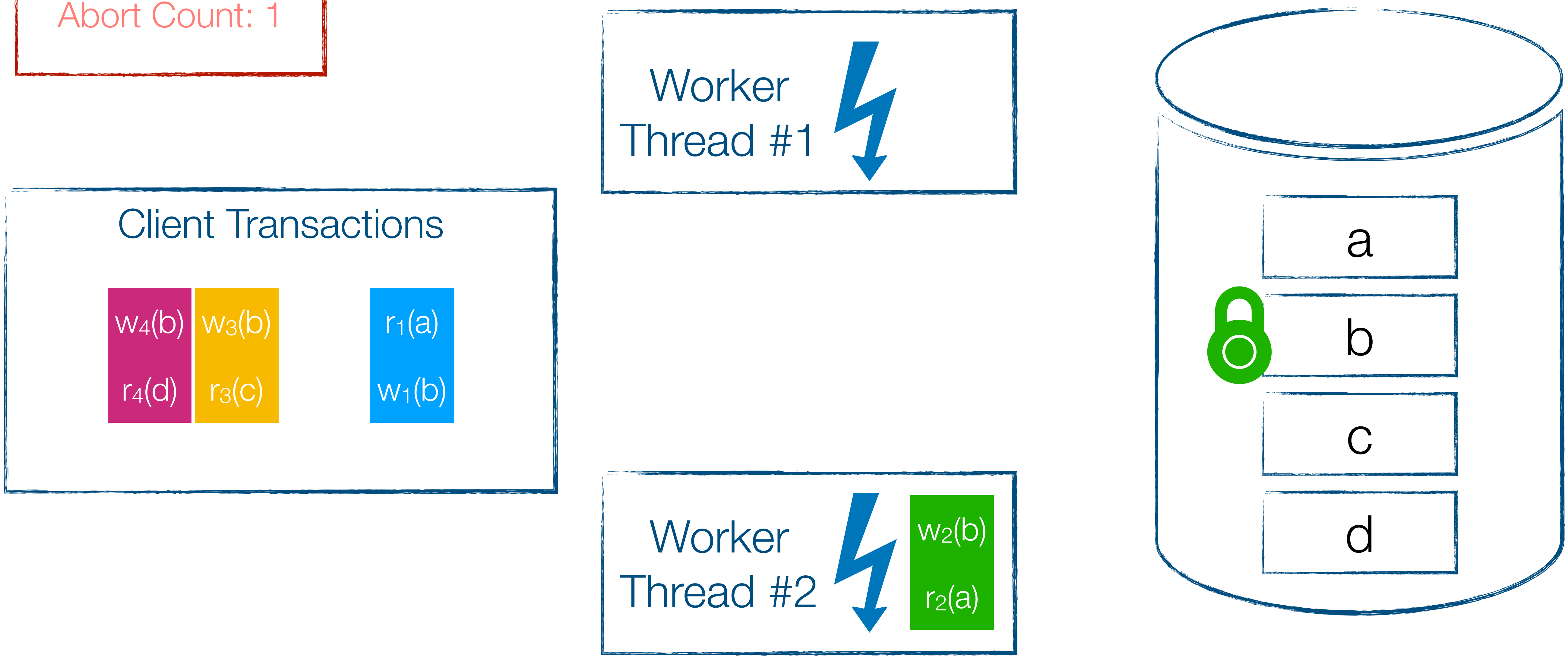
2PL - NoWait
Abort Count: 0
Client Transactions
w4(b)
r4(d)
w3(b)
r3(c)
Worker Thread #1
r1(a)
w1(b)
Worker Thread #2
w2(b)
r2(a)
a
b
c
d


2PL - NoWait
Abort Count: 0
Client Transactions
w4(b)
r4(d)
w3(b)
r3(c)
Worker Thread #1
r1(a)
w1(b)
conflict!
Worker Thread #2
w2(b)
r2(a)
a
b
c
d


2PL - NoWait
Abort Count: 0
Abort transaction (to avoid potential deadlocks)
Worker Thread #1
r1(a)
w1(b)
Client Transactions
w4(b)
r4(d)
w3(b)
r3(c)
Worker Thread #2
w2(b)
r2(a)
a
b
c
d

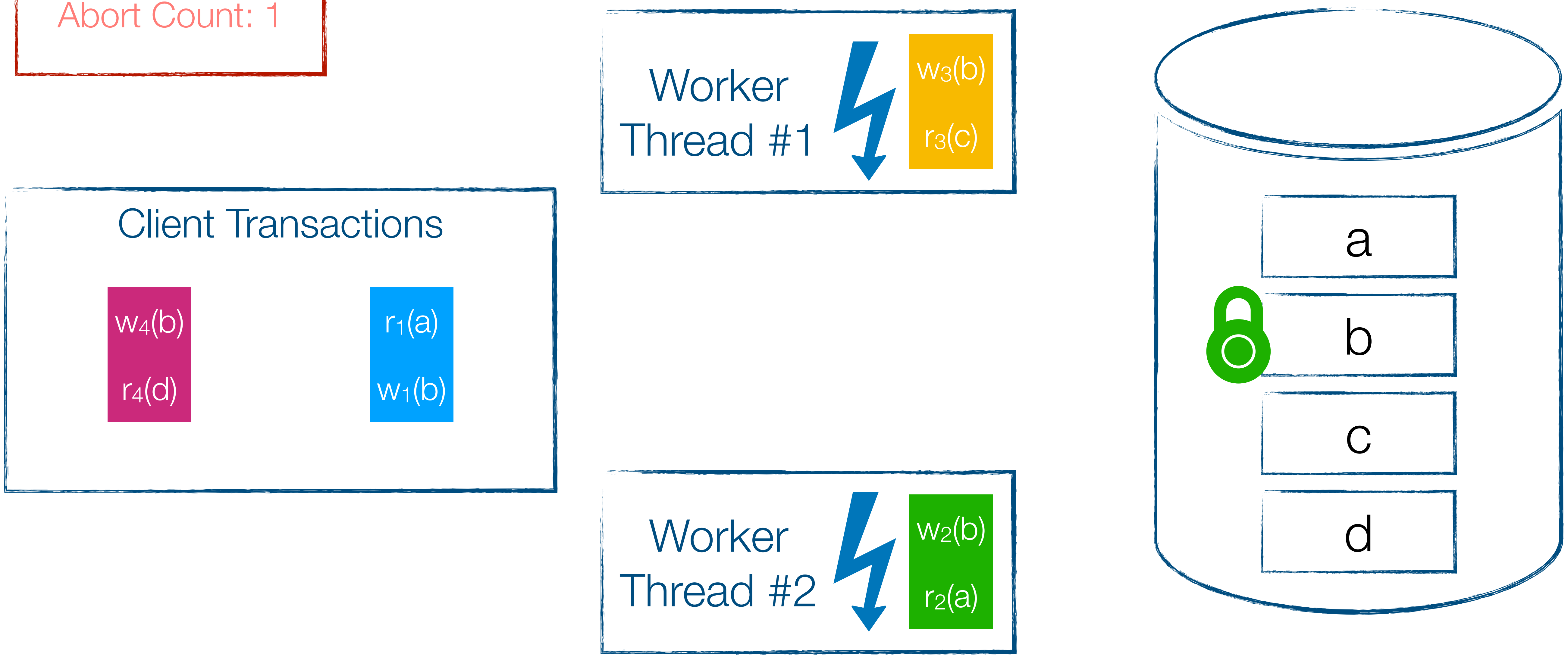
2PL - NoWait

Abort Count: 1



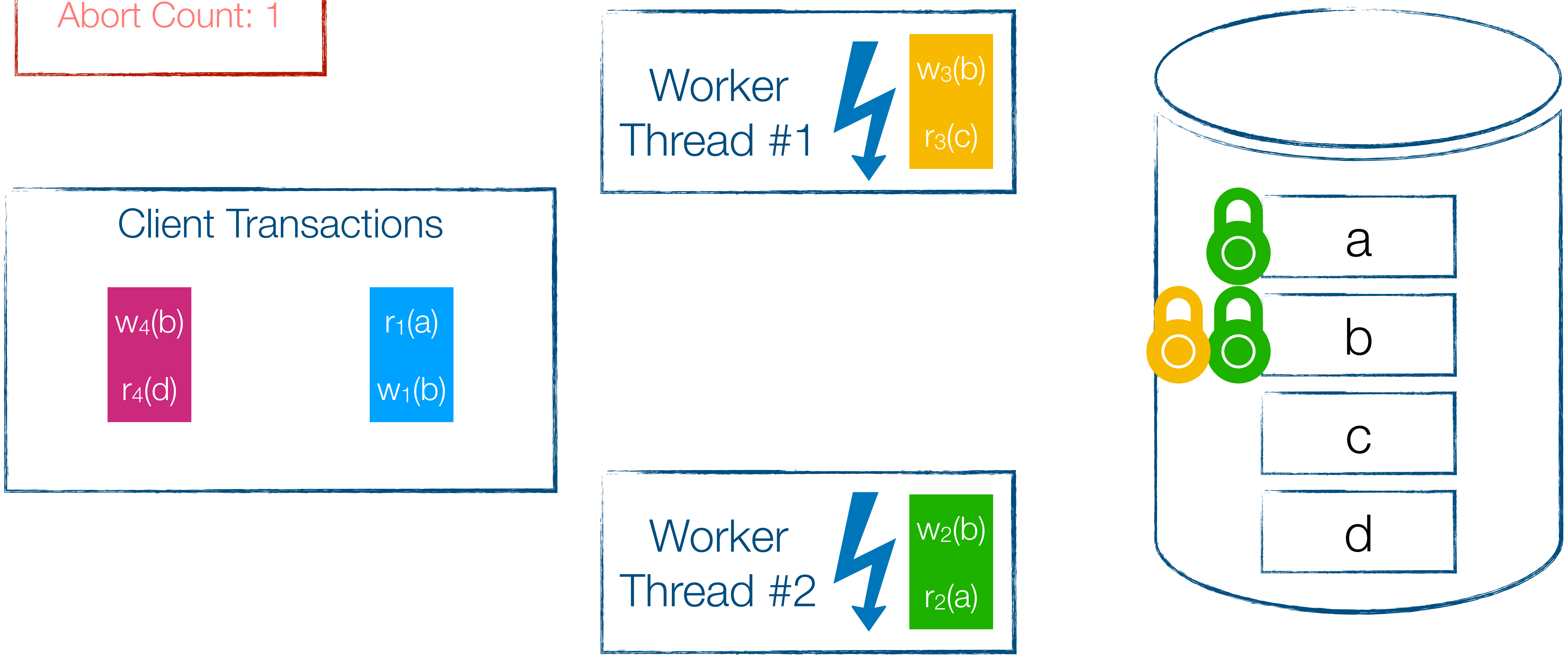
2PL - NoWait

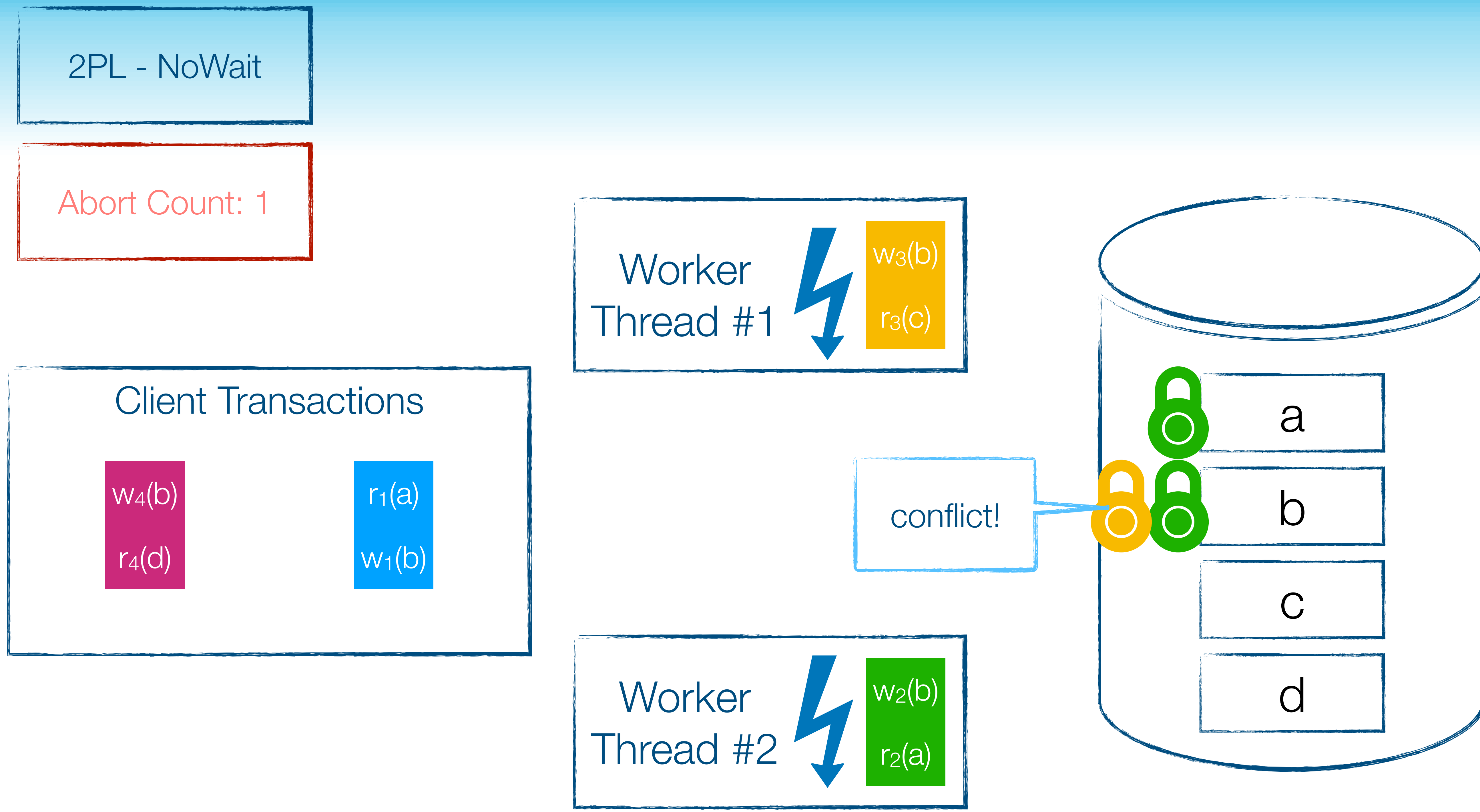
Abort Count: 1

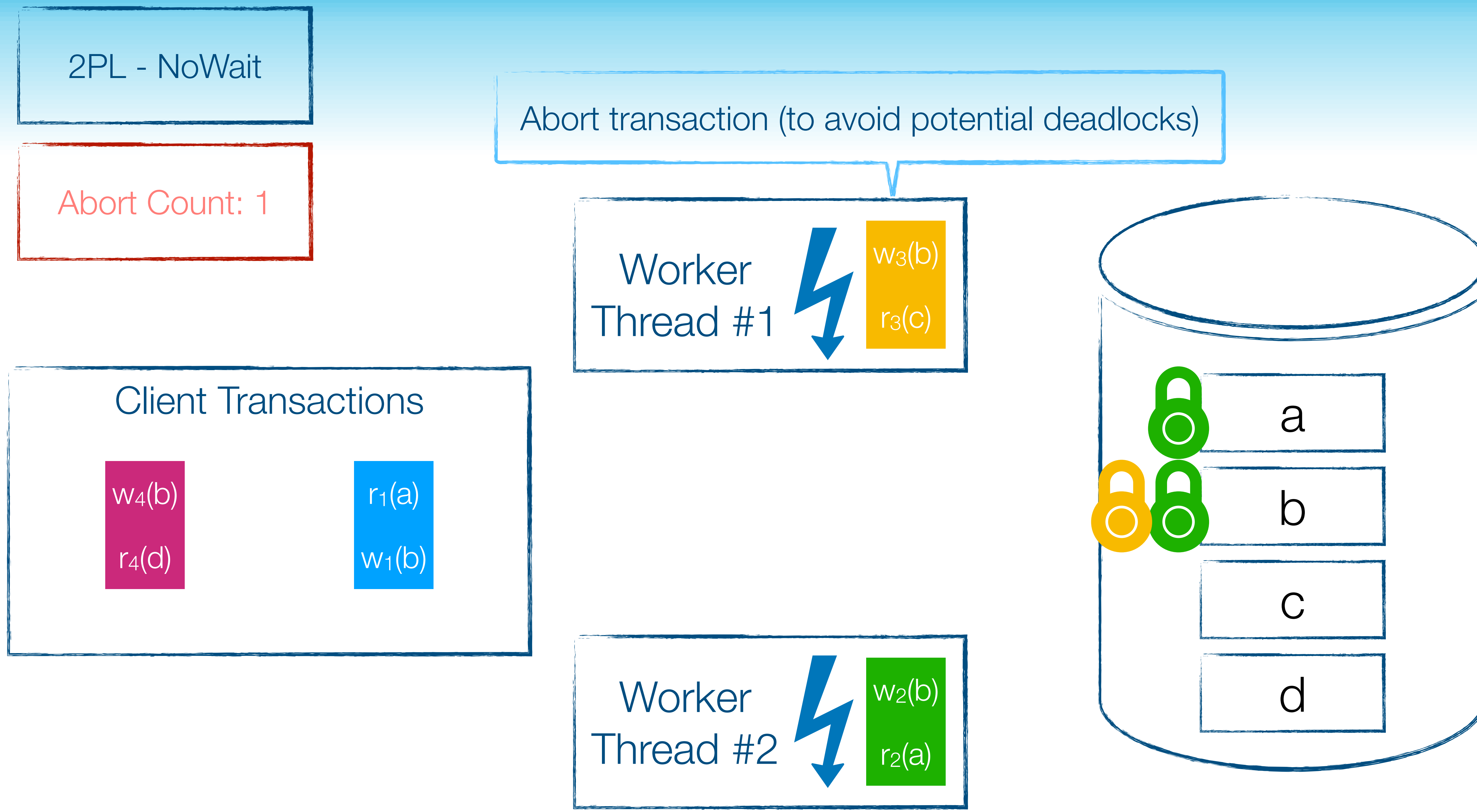


2PL - NoWait

Abort Count: 1

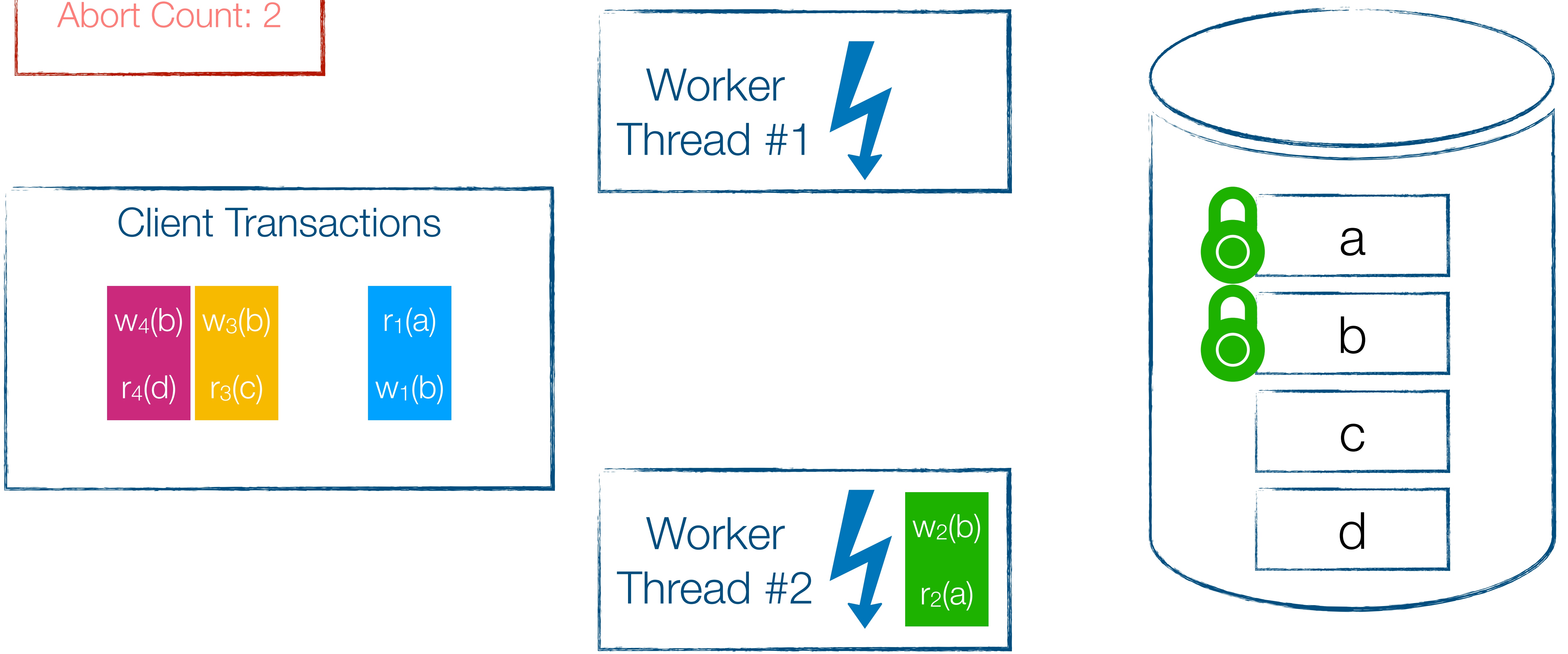


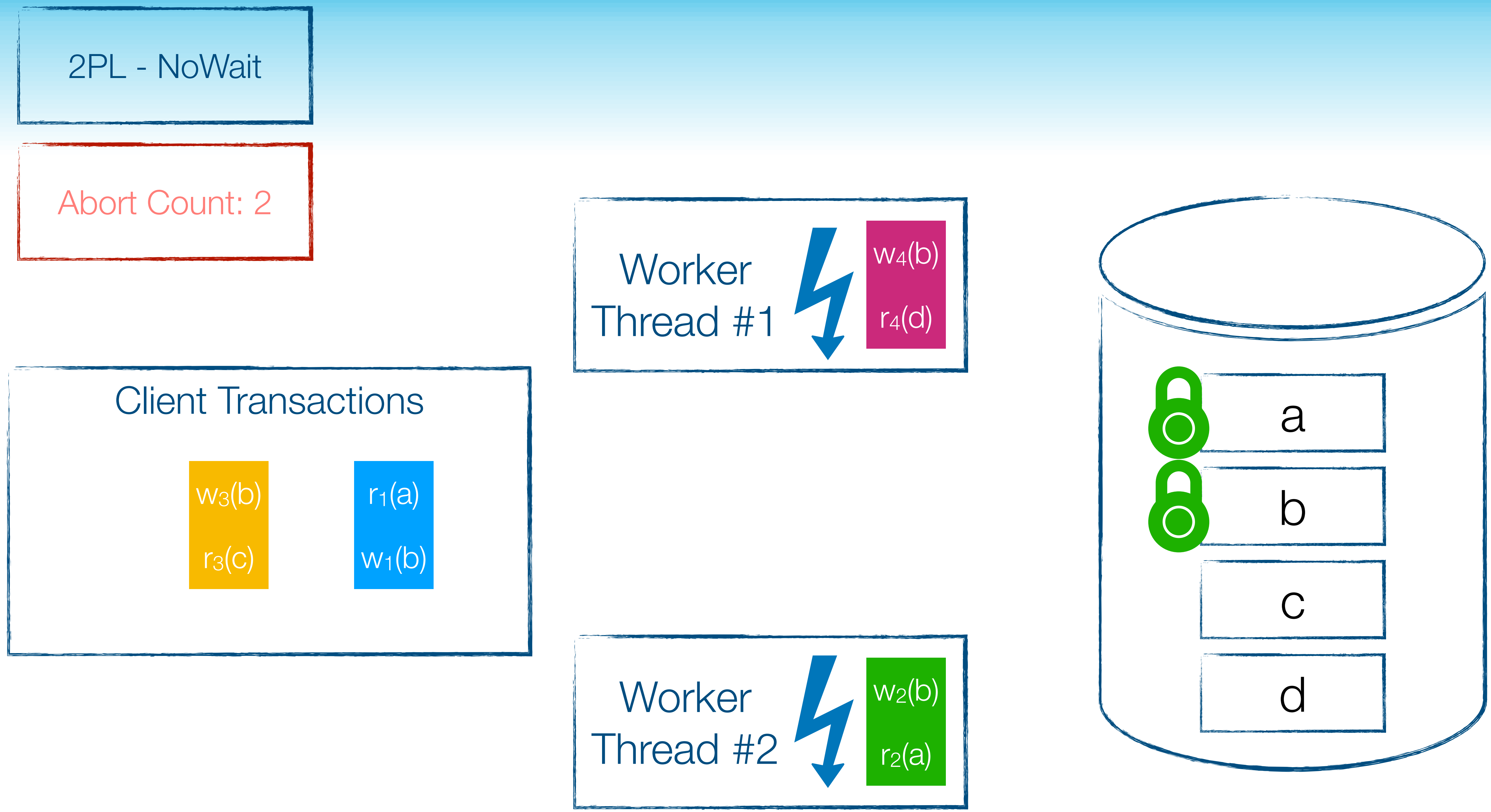

2PL - NoWait
Abort Count: 1
Client Transactions
w4(b)
r4(d)
r1(a)
w1(b)
Worker Thread #1
w3(b)
r3(c)
conflict!
Worker Thread #2
w2(b)
r2(a)
a
b
c
d

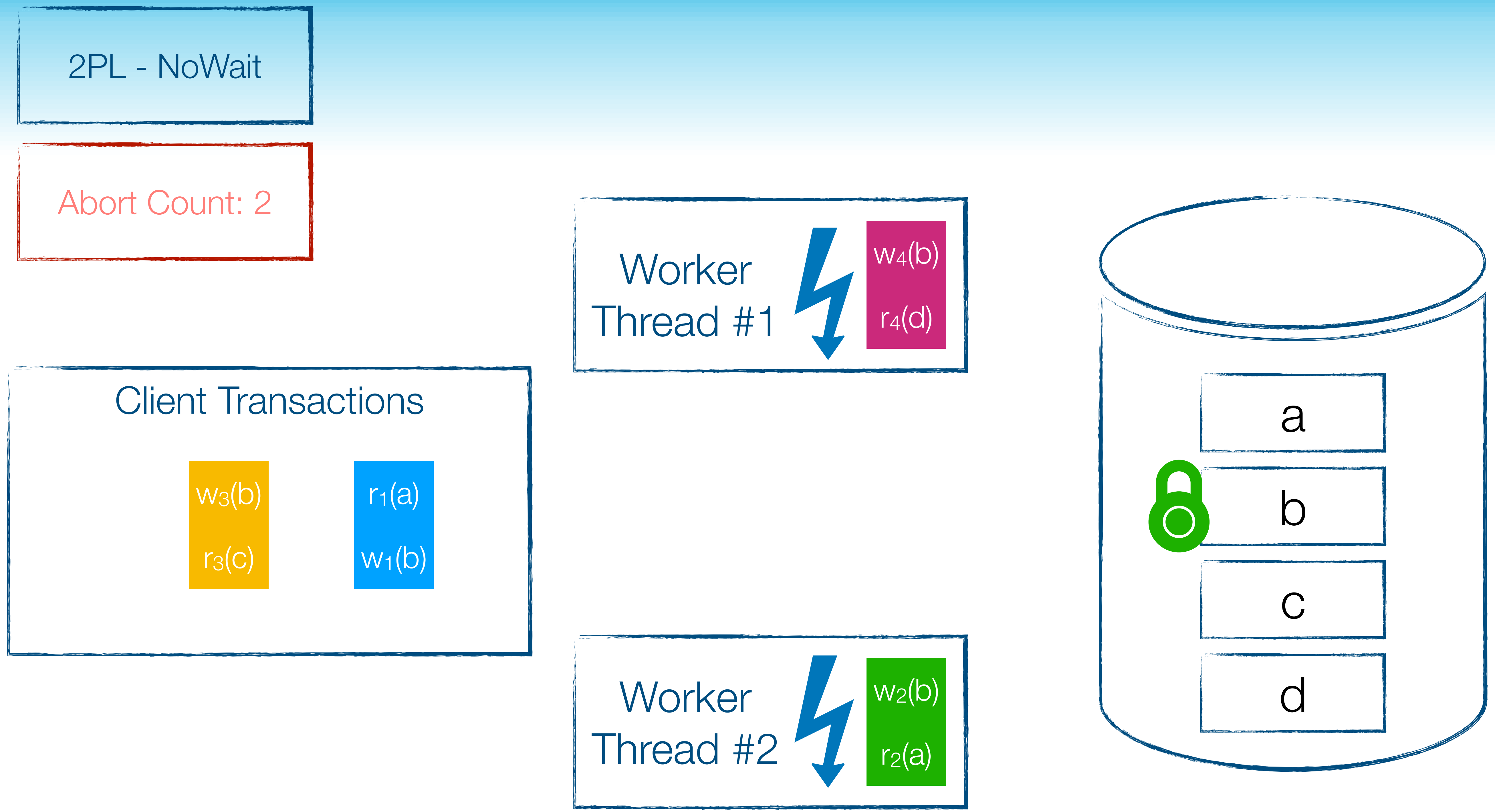

2PL - NoWait
Abort Count: 1
Abort transaction (to avoid potential deadlocks)
Worker Thread #1
w3(b)
r3(c)
Client Transactions
w4(b)
r4(d)
r1(a)
w1(b)
Worker Thread #2
w2(b)
r2(a)
a
b
c
d

2PL - NoWait

Abort Count: 2

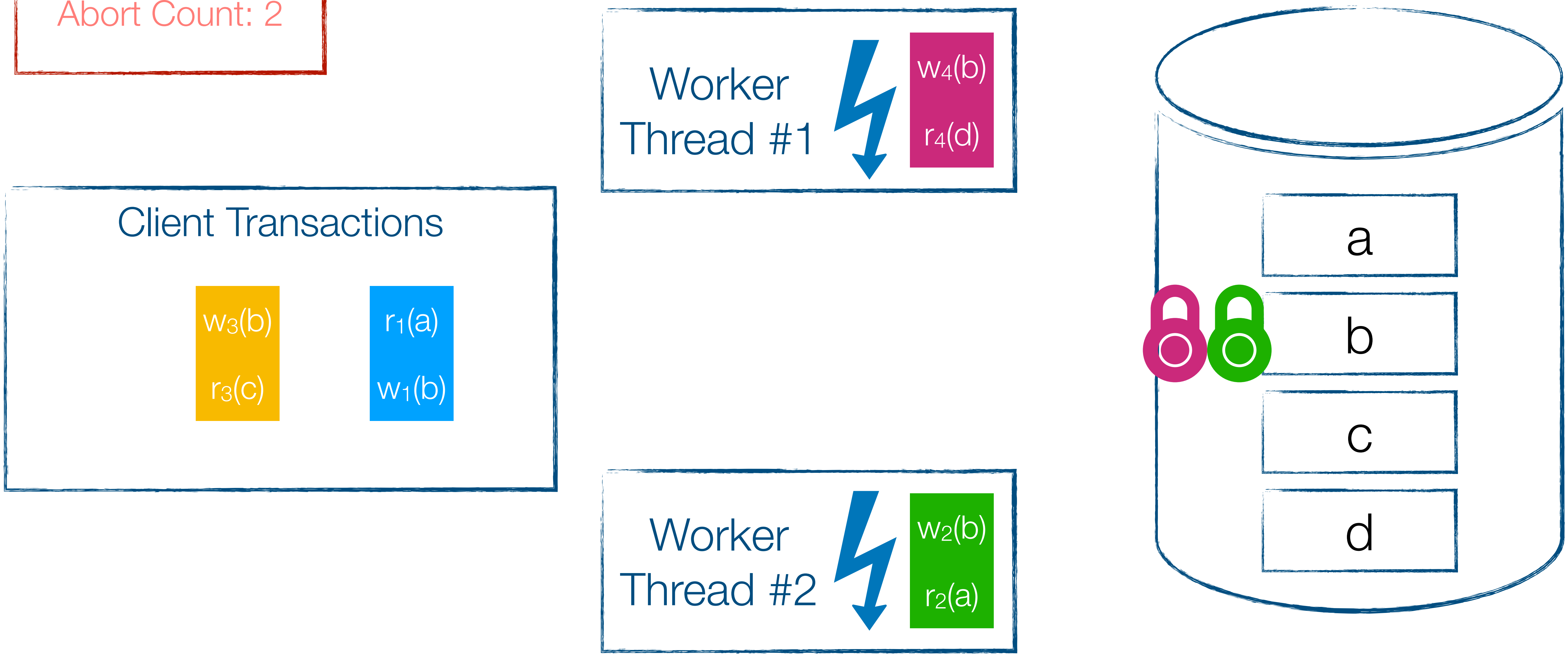


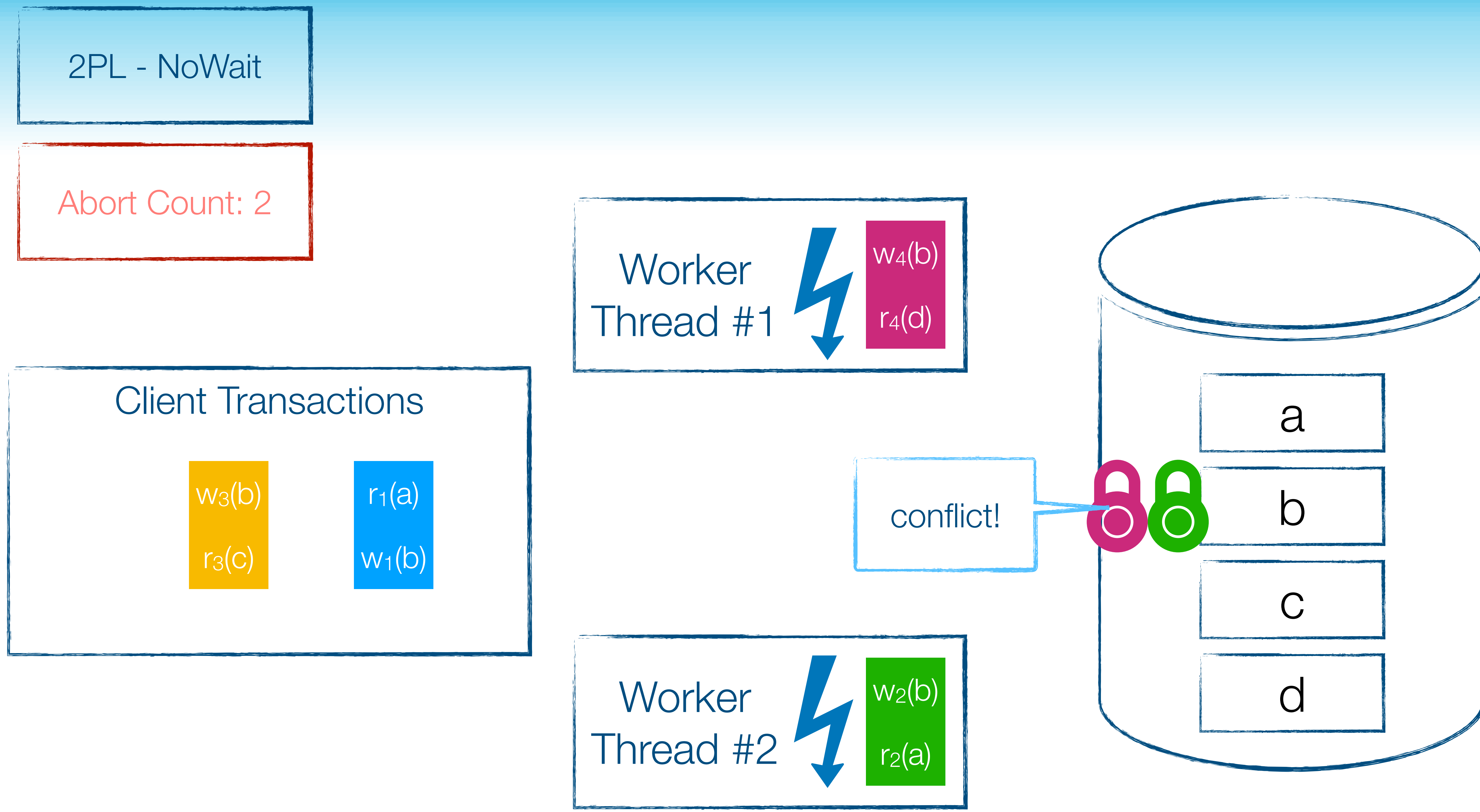

2PL - NoWait
Abort Count: 2
Client Transactions
w3(b)
r3(c)
r1(a)
w1(b)
Worker Thread #1
w4(b)
r4(d)
Worker Thread #2
w2(b)
r2(a)
a
b
c
d

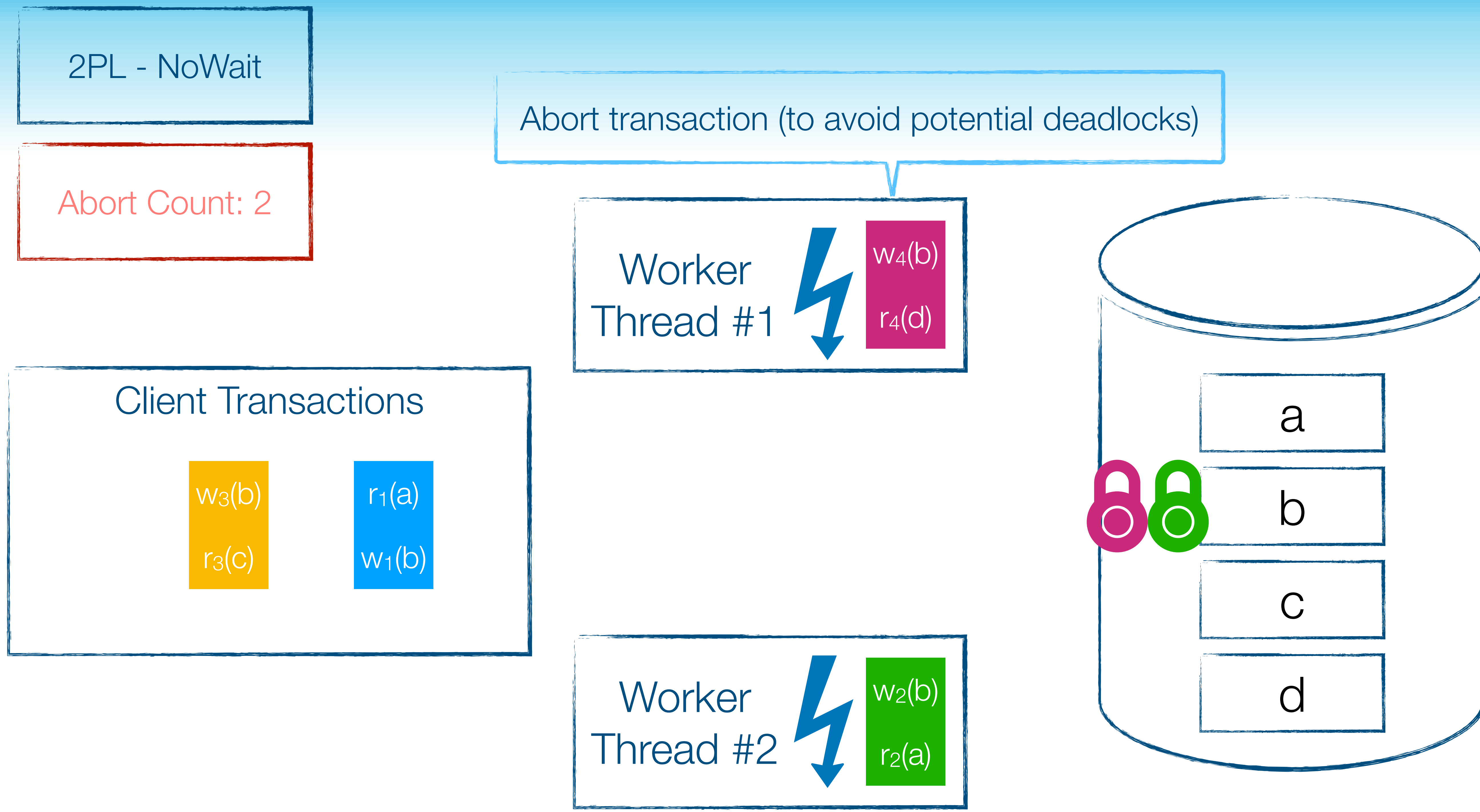

2PL - NoWait
Abort Count: 2
Client Transactions
w3(b)
r3(c)
r1(a)
w1(b)
Worker Thread #1
w4(b)
r4(d)
Worker Thread #2
w2(b)
r2(a)
a
b
c
d

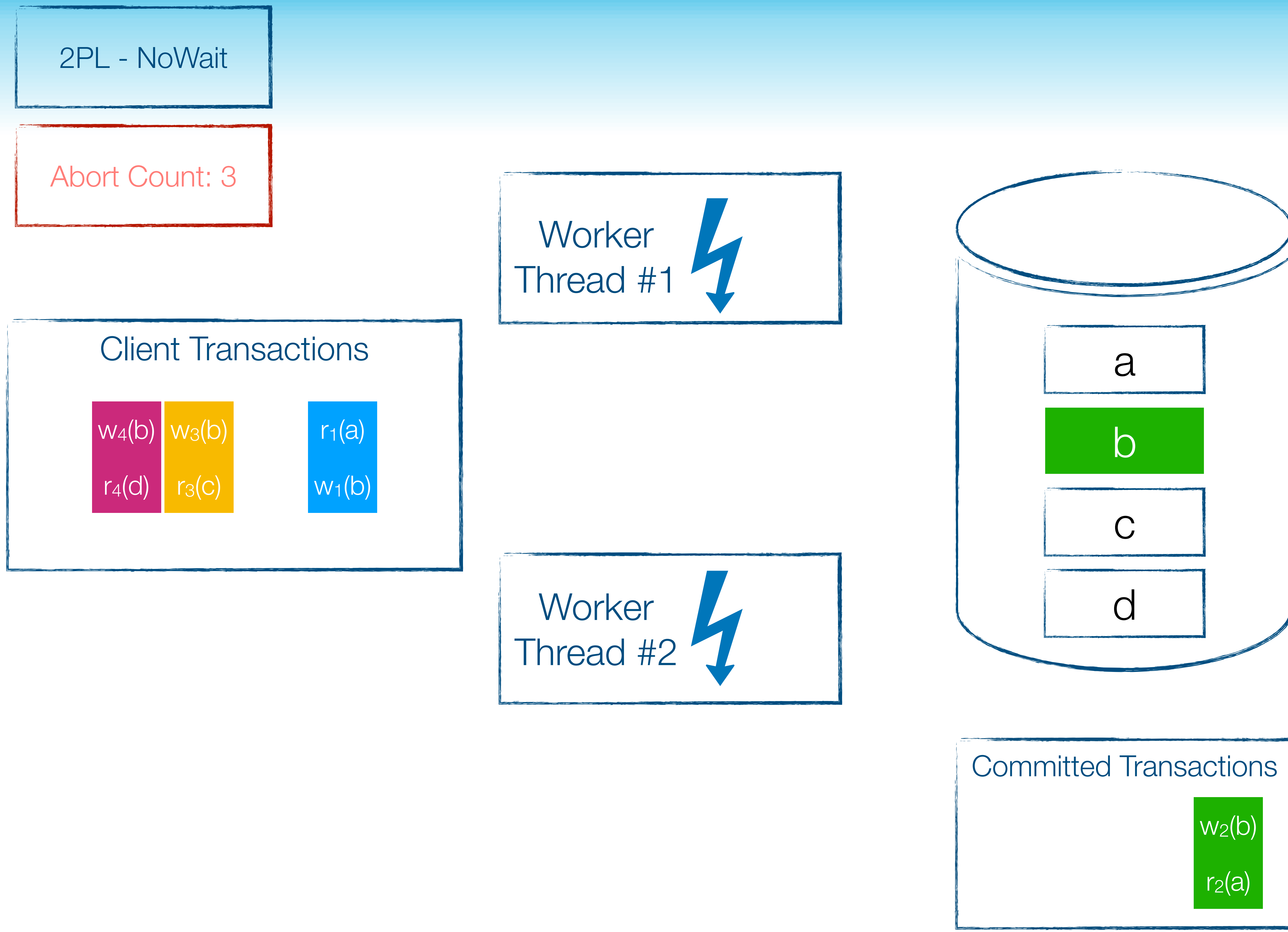
2PL - NoWait

Abort Count: 2



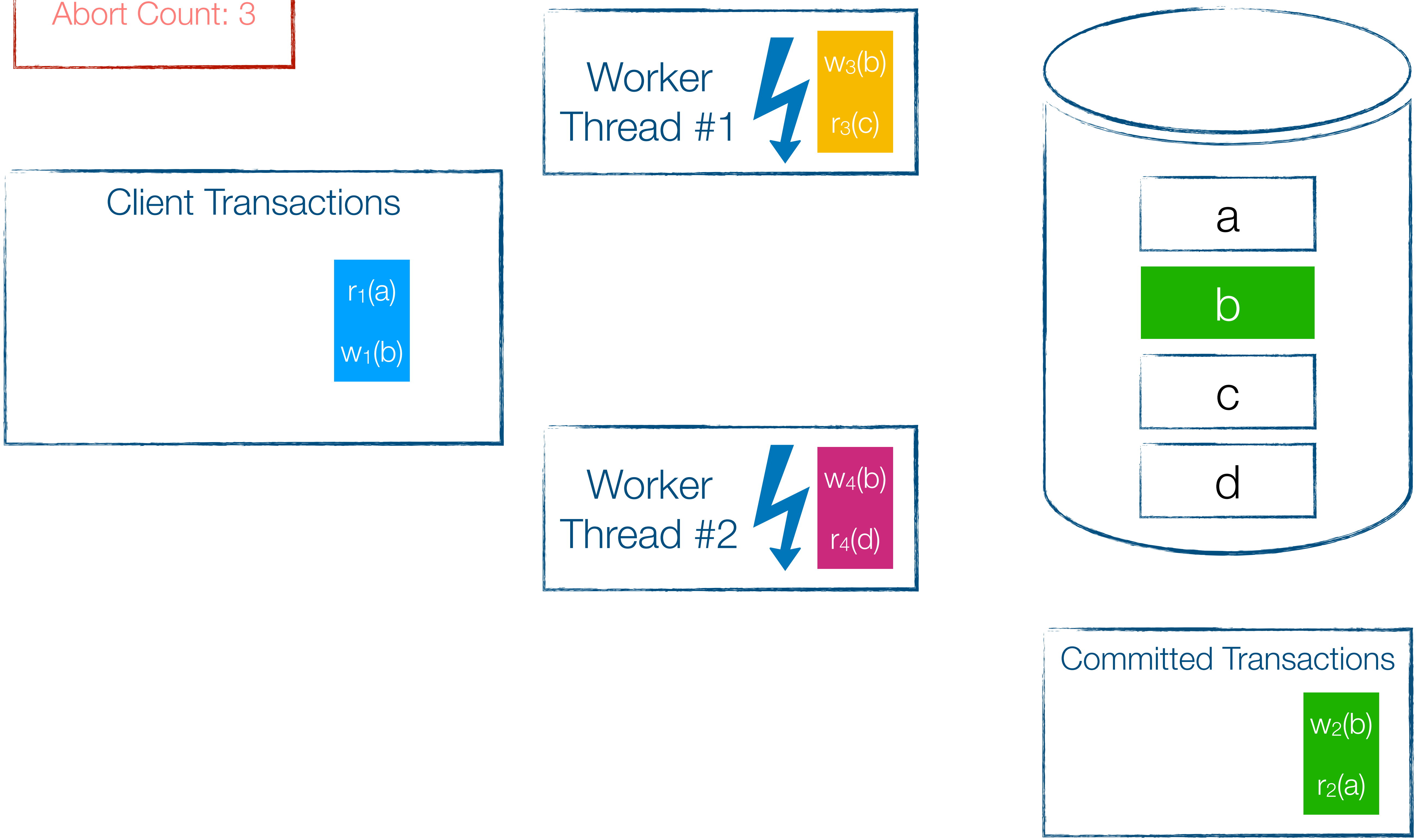

2PL - NoWait
Abort Count: 2
Client Transactions
w3(b)
r3(c)
r1(a)
w1(b)
Worker Thread #1
w4(b)
r4(d)
conflict!
a
b
c
d
Worker Thread #2
w2(b)
r2(a)

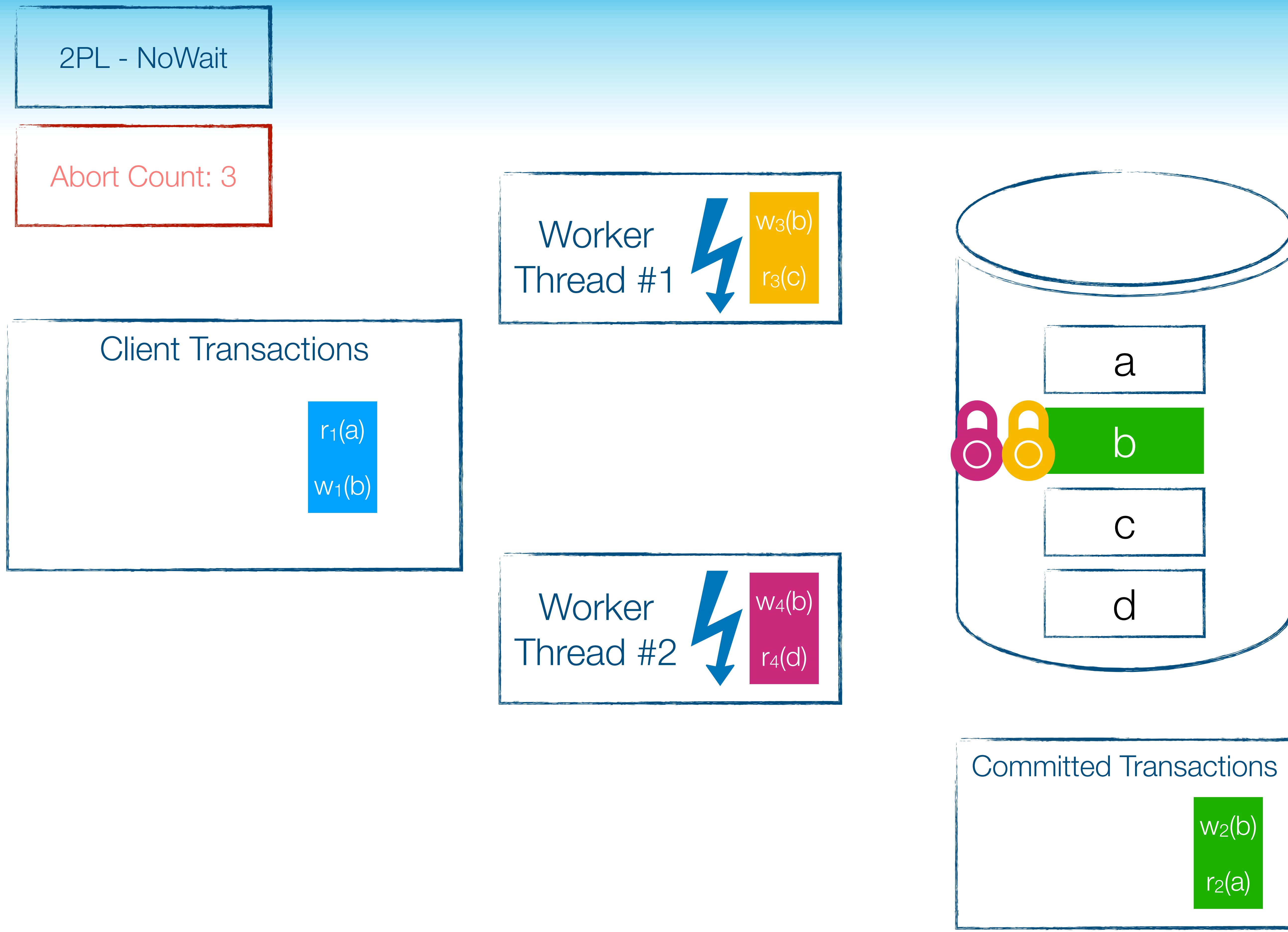

2PL - NoWait
Abort Count: 2
Abort transaction (to avoid potential deadlocks)
Worker Thread #1
$w_4(b)$
$r_4(d)$
Client Transactions
$w_3(b)$
$r_3(c)$
$r_1(a)$
$w_1(b)$
Worker Thread #2
$w_2(b)$
$r_2(a)$
a
b
c
d

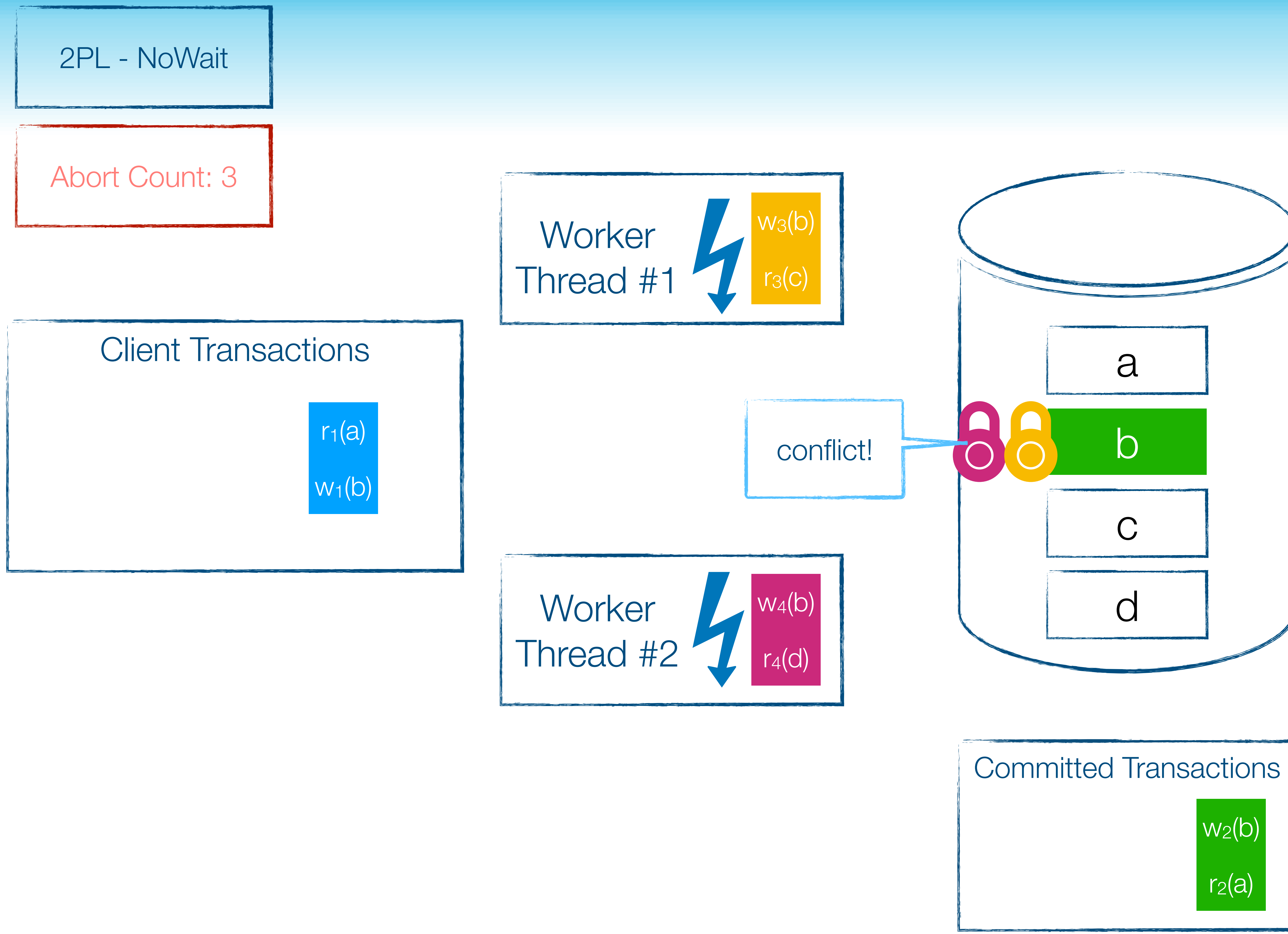

2PL - NoWait
Abort Count: 3
Client Transactions
w4(b)
r4(d)
w3(b)
r3(c)
r1(a)
w1(b)
Worker Thread #1
Worker Thread #2
a
b
c
d
Committed Transactions
w2(b)
r2(a)

2PL - NoWait

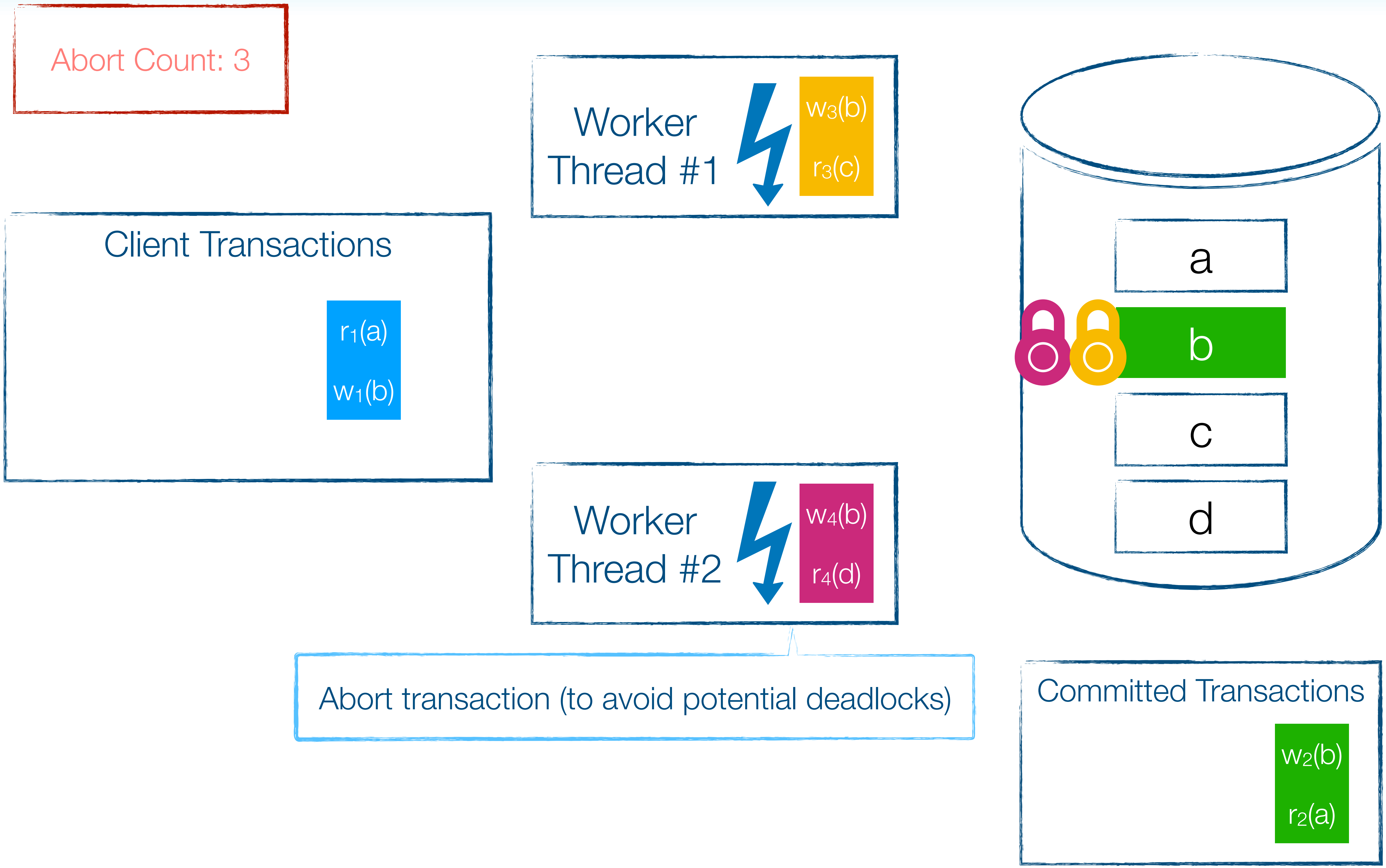
Abort Count: 3



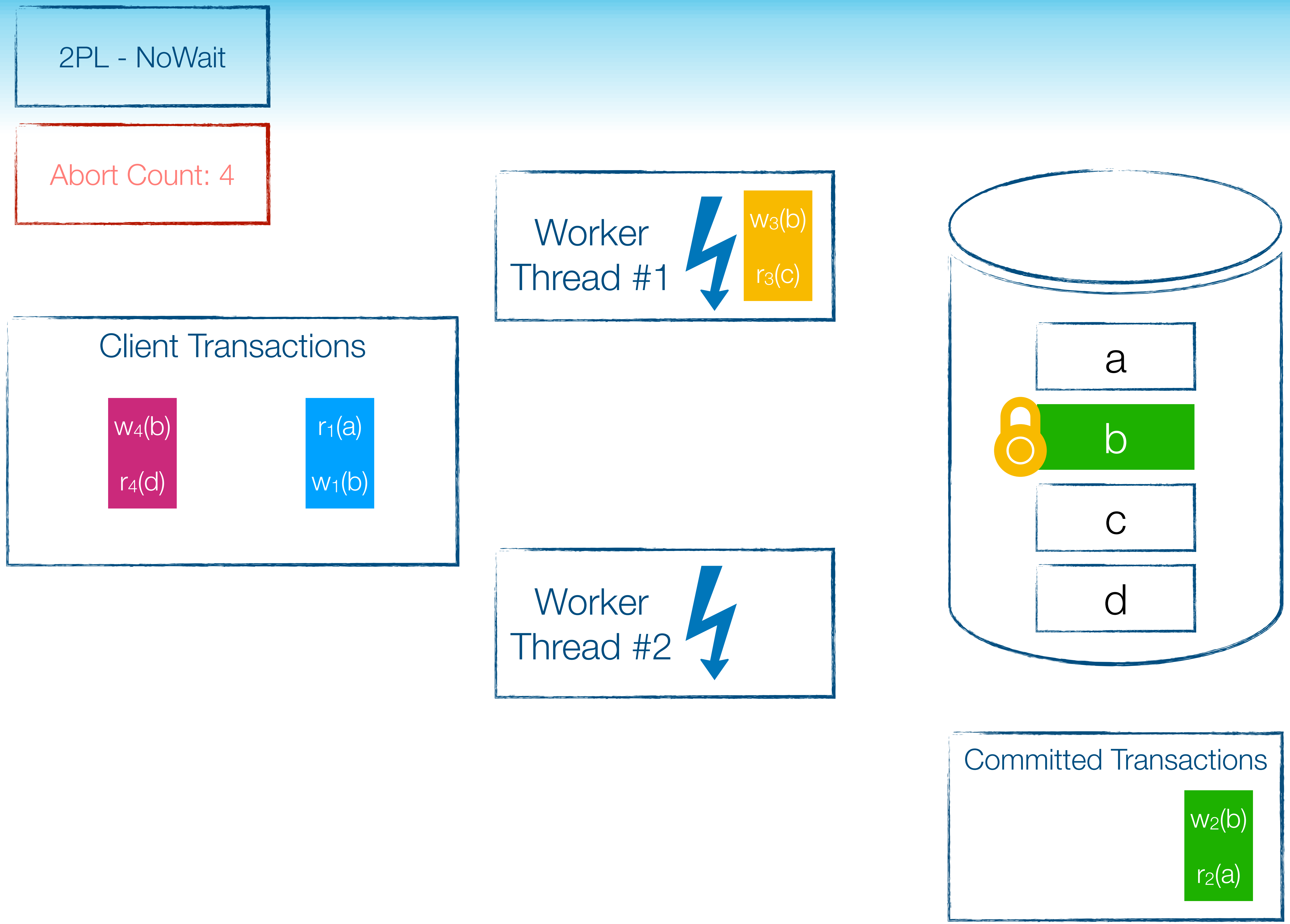

2PL - NoWait
Abort Count: 3
Client Transactions
r1(a)
w1(b)
Worker Thread #1
w3(b)
r3(c)
Worker Thread #2
w4(b)
r4(d)
a
b
c
d
Committed Transactions
w2(b)
r2(a)


2PL - NoWait
Abort Count: 3
Worker Thread #1
w3(b)
r3(c)
Client Transactions
r1(a)
w1(b)
conflict!
a
b
c
d
Worker Thread #2
w4(b)
r4(d)
Committed Transactions
w2(b)
r2(a)

2PL - NoWait



Abort Count: 3
Worker Thread #1
$w_3(b)$
$r_3(c)$
Client Transactions
$r_1(a)$
$w_1(b)$
Worker Thread #2
$w_4(b)$
$r_4(d)$
Abort transaction (to avoid potential deadlocks)
a
b
c
d
Committed Transactions
$w_2(b)$
$r_2(a)$


2PL - NoWait
Abort Count: 4
Worker Thread #1
$w_3(b)$
$r_3(c)$
Client Transactions
$w_4(b)$
$r_4(d)$
$r_1(a)$
$w_1(b)$
Worker Thread #2
a
b
c
d
Committed Transactions
$w_2(b)$
$r_2(a)$

2PL - NoWait

Abort Count: 4

Worker Thread #1

$w_3(b)$

$r_3(c)$

Client Transactions

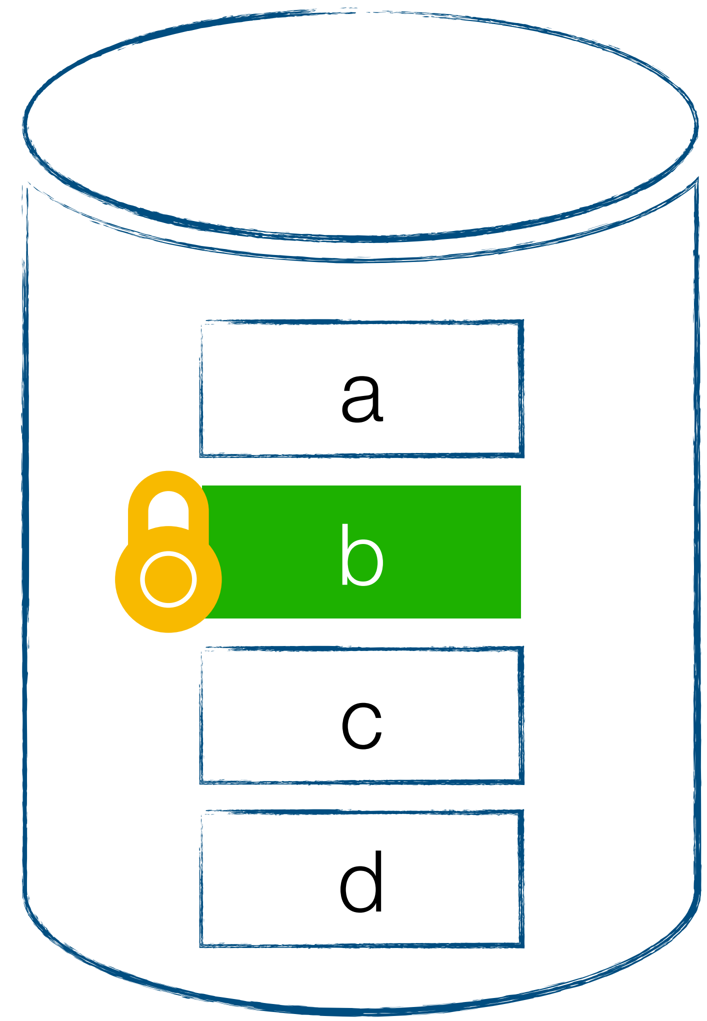
$w_4(b)$

$r_4(d)$

Worker Thread #2

$r_1(a)$

$w_1(b)$



Committed Transactions

$w_2(b)$

$r_2(a)$

2PL - NoWait

Abort Count: 4

Worker Thread #1

$w_3(b)$

$r_3(c)$

Client Transactions

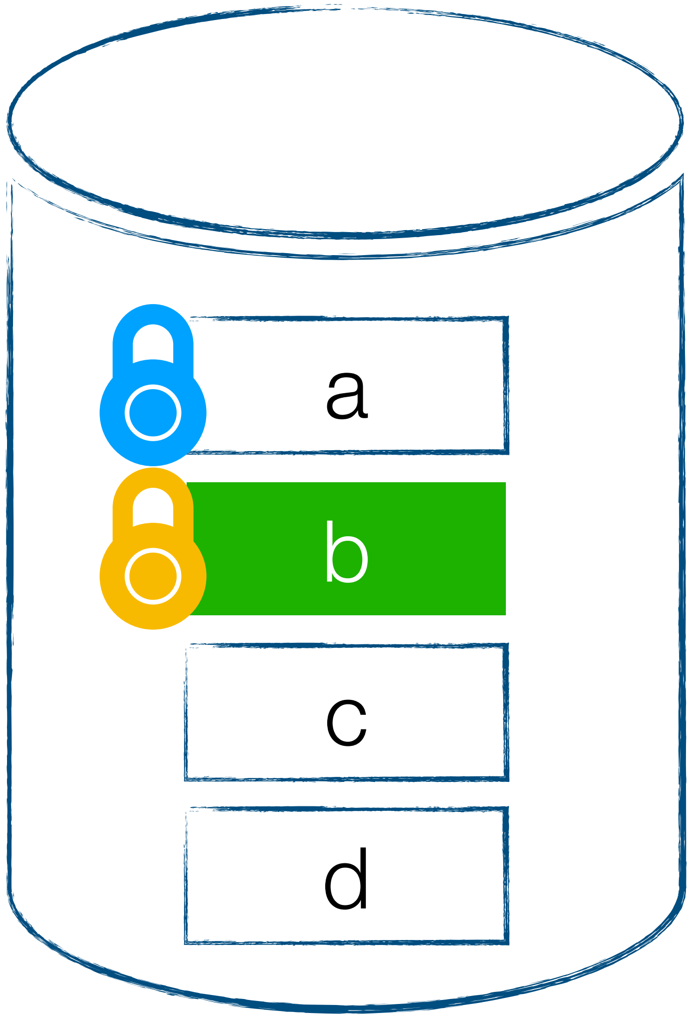
$w_4(b)$

$r_4(d)$

Worker Thread #2

$r_1(a)$

$w_1(b)$



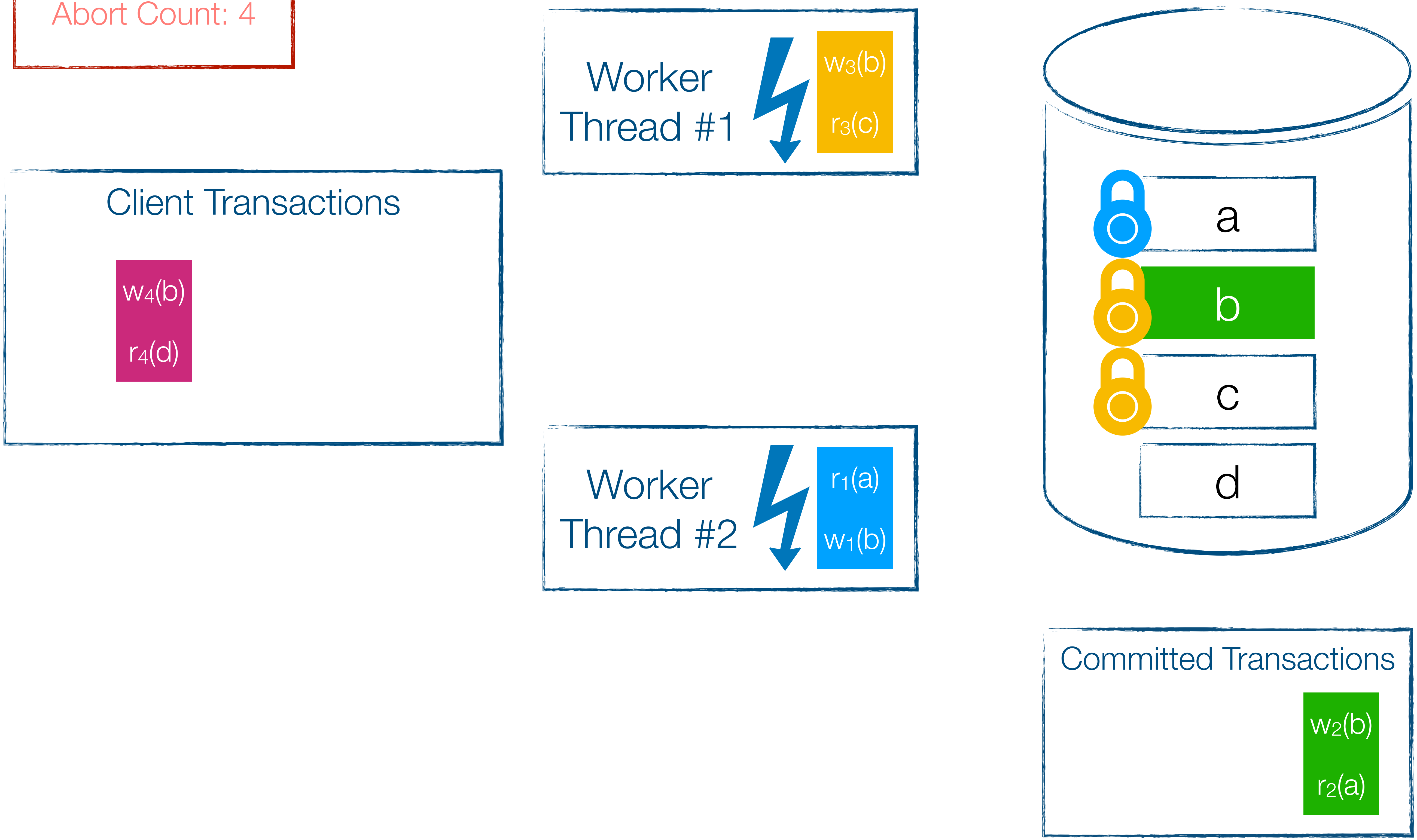
Committed Transactions

$w_2(b)$

$r_2(a)$

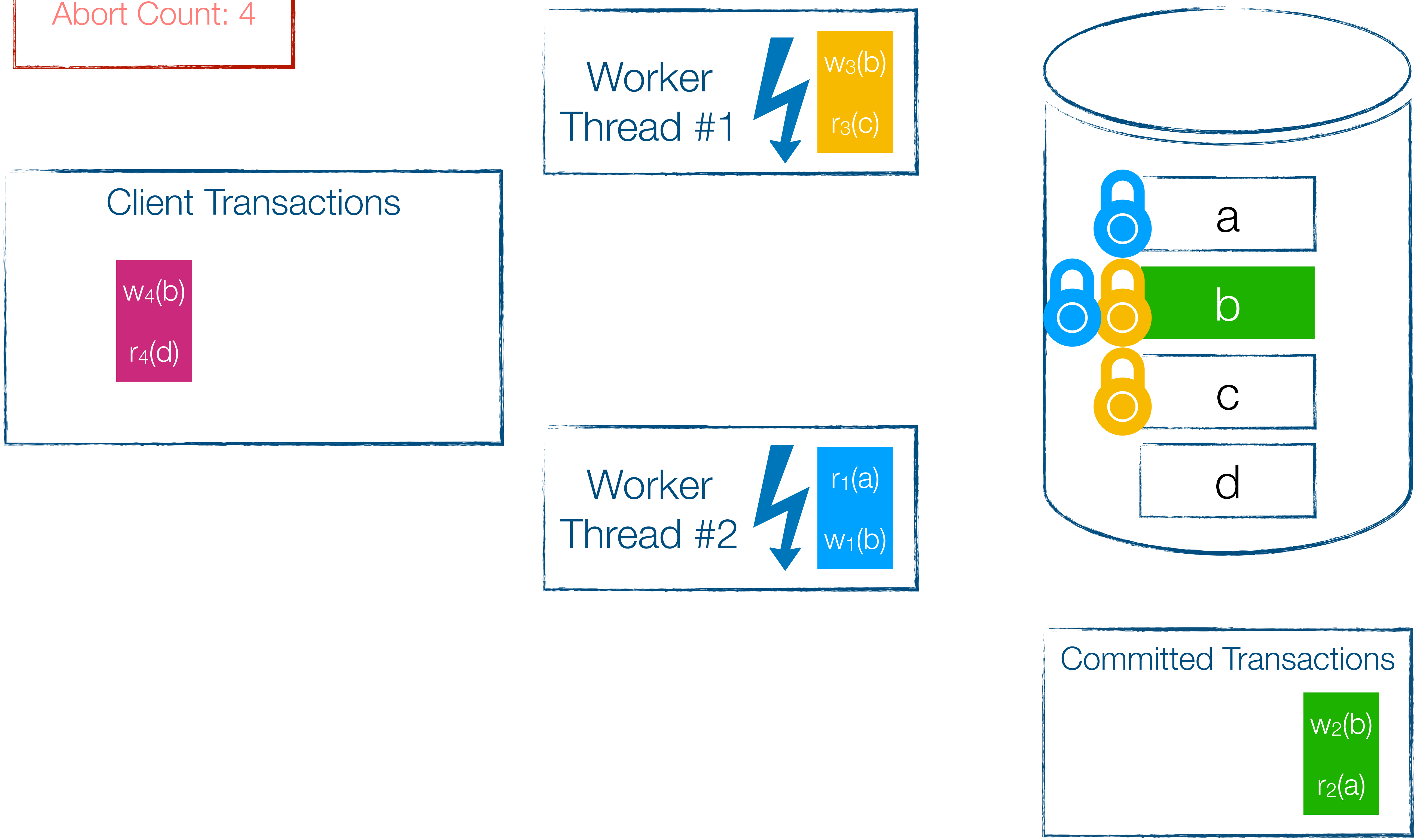
2PL - NoWait

Abort Count: 4

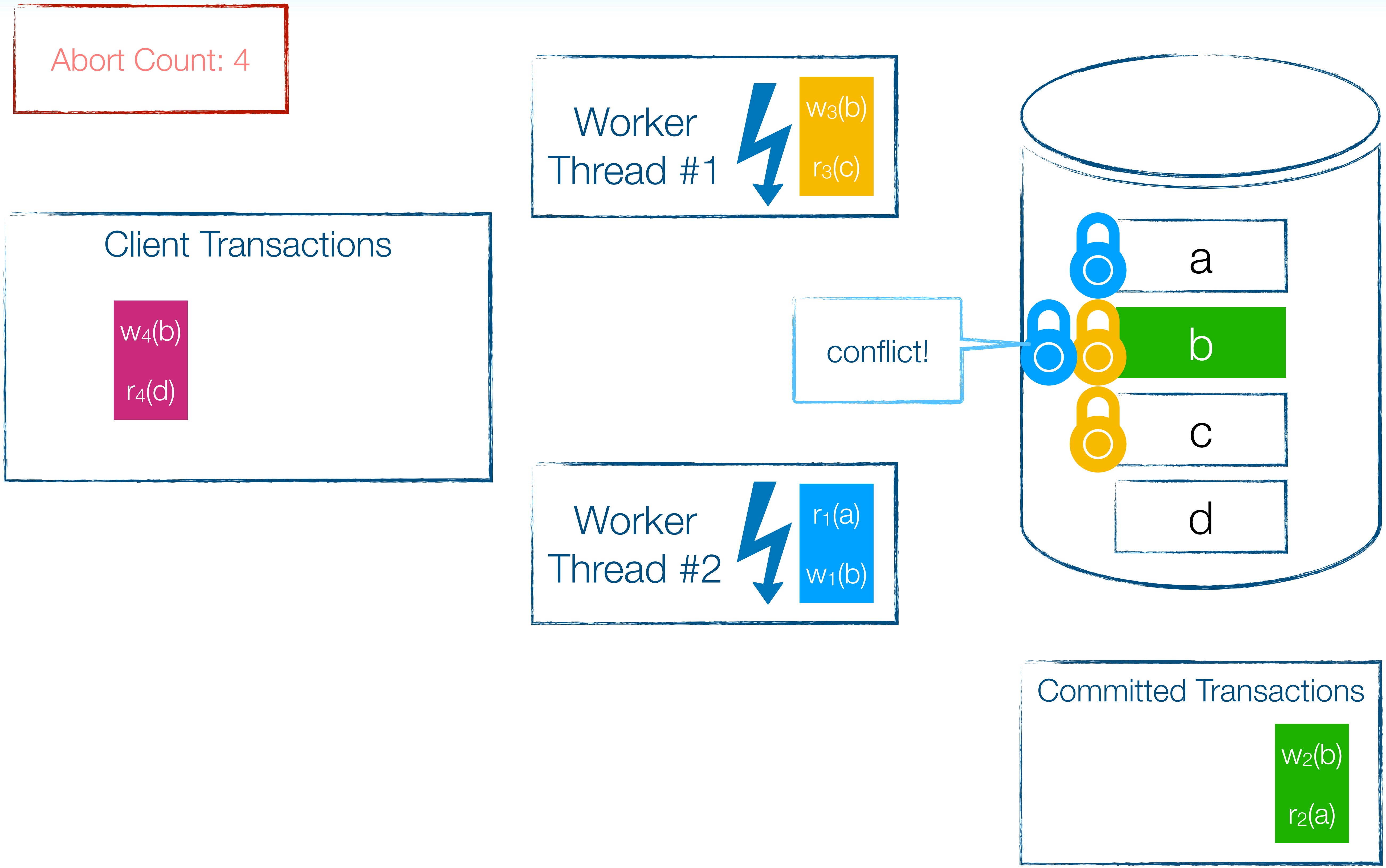


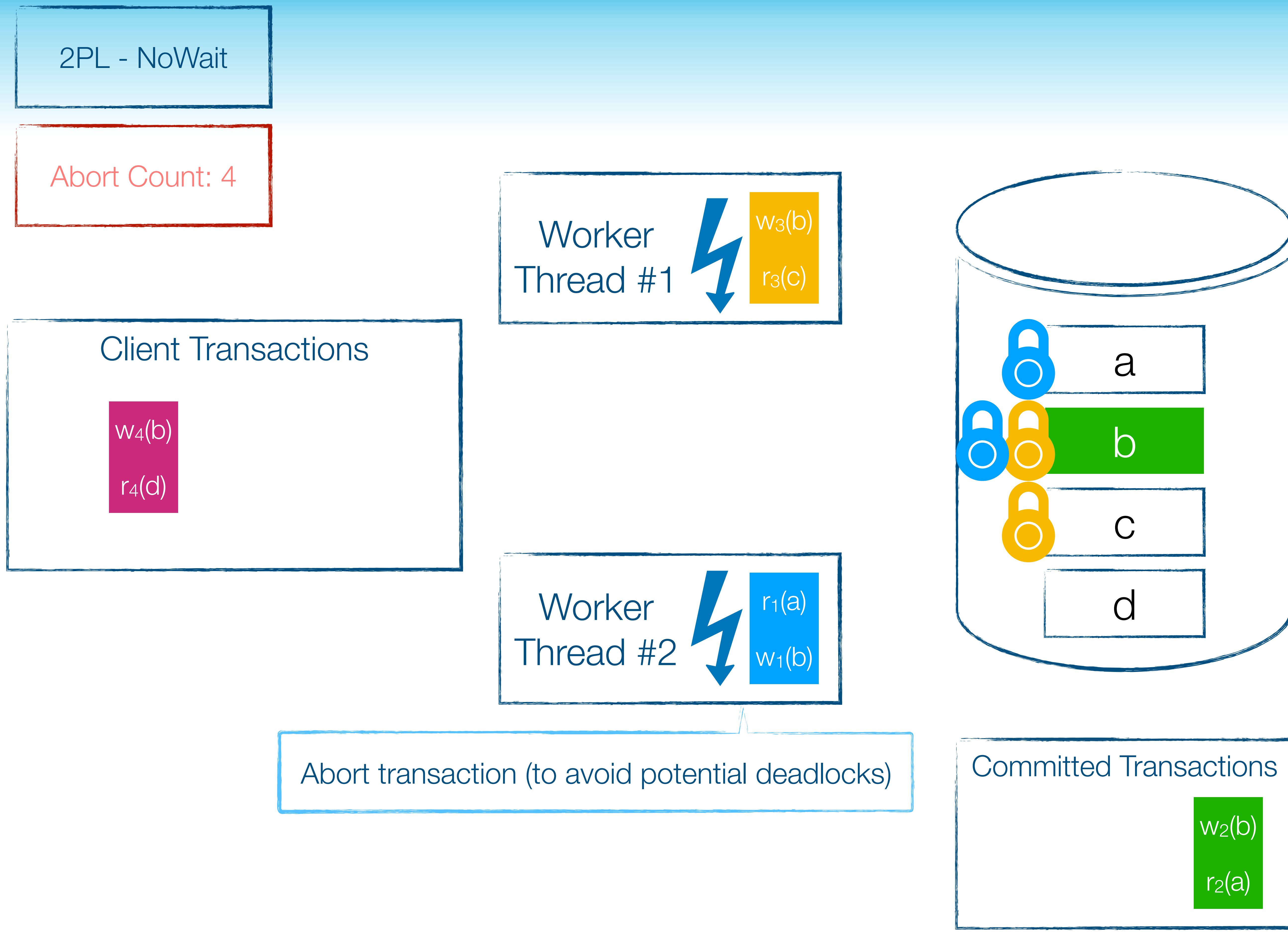
2PL - NoWait

Abort Count: 4



2PL - NoWait


Abort Count: 4
Worker Thread #1
w3(b)
r3(c)
Client Transactions
w4(b)
r4(d)
conflict!
a
b
c
d
Worker Thread #2
r1(a)
w1(b)
Committed Transactions
w2(b)
r2(a)



2PL - NoWait
Abort Count: 4
Client Transactions
$w_4(b)$
$r_4(d)$
Worker Thread #1
$w_3(b)$
$r_3(c)$
Worker Thread #2
$r_1(a)$
$w_1(b)$
Abort transaction (to avoid potential deadlocks)
a
b
c
d
Committed Transactions
$w_2(b)$
$r_2(a)$

2PL - NoWait

Abort Count: 5

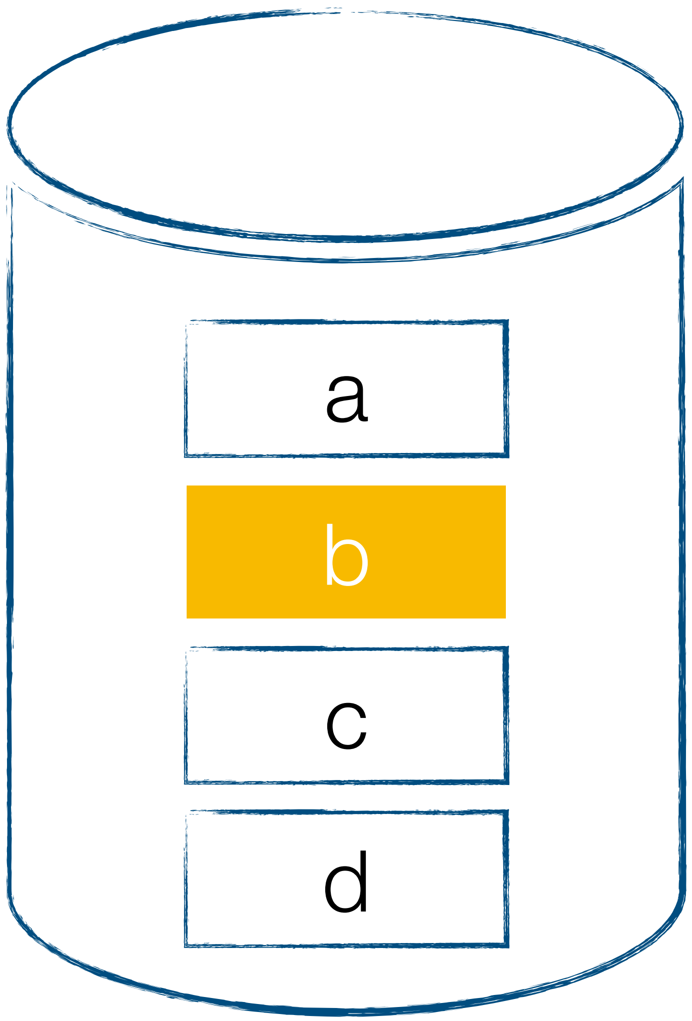
Worker Thread #1

Client Transactions

$w_4(b)$ $r_4(d)$

$r_1(a)$ $w_1(b)$

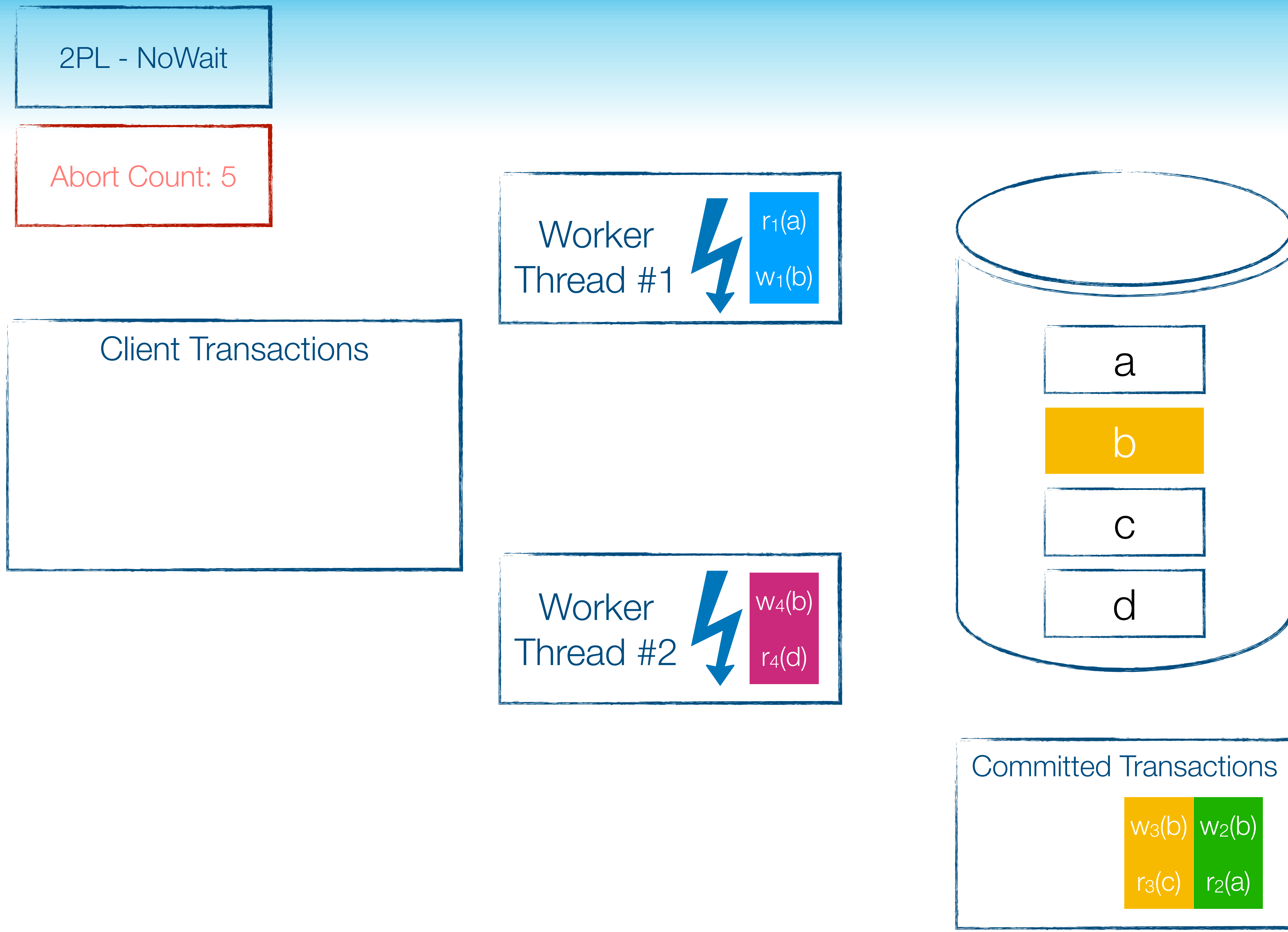
Worker Thread #2



Committed Transactions

$w_3(b)$ $r_3(c)$

$w_2(b)$ $r_2(a)$


2PL - NoWait
Abort Count: 5
Client Transactions
Worker Thread #1
r1(a)
w1(b)
Worker Thread #2
w4(b)
r4(d)
a
b
c
d
Committed Transactions
w3(b)
r3(c)
w2(b)
r2(a)

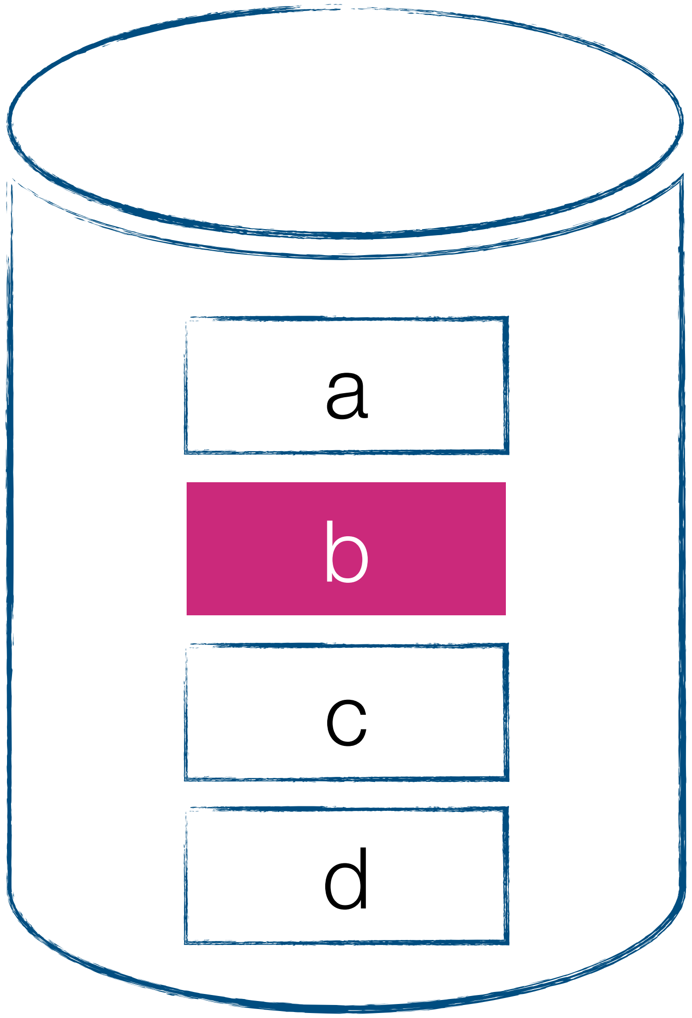
2PL - NoWait

Abort Count: 5

Worker Thread #1

Client Transactions

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

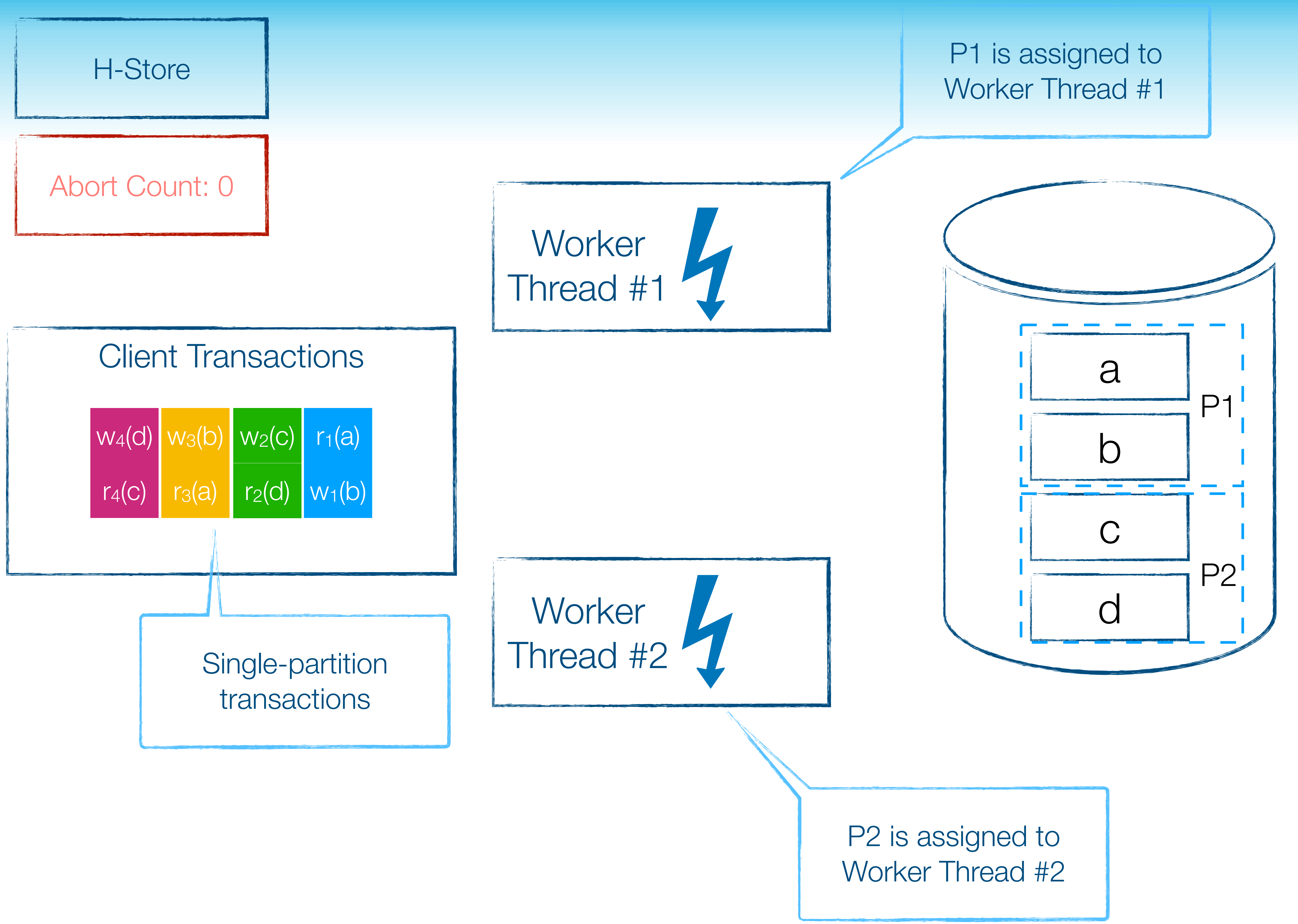
$w_4(b)$	$r_1(a)$	$w_3(b)$	$w_2(b)$
$r_4(d)$	$w_1(b)$	$r_3(c)$	$r_2(a)$

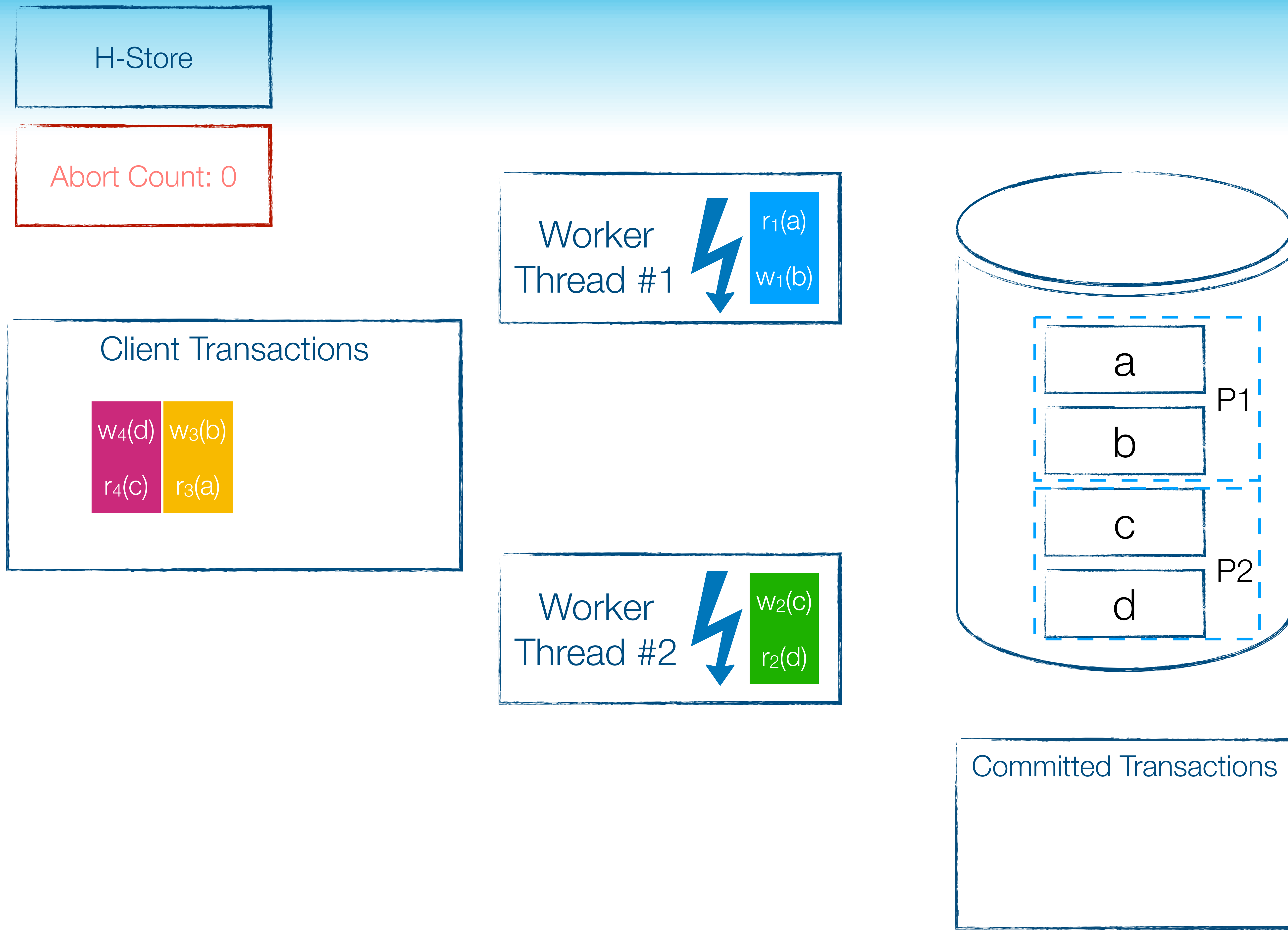
Key Insights

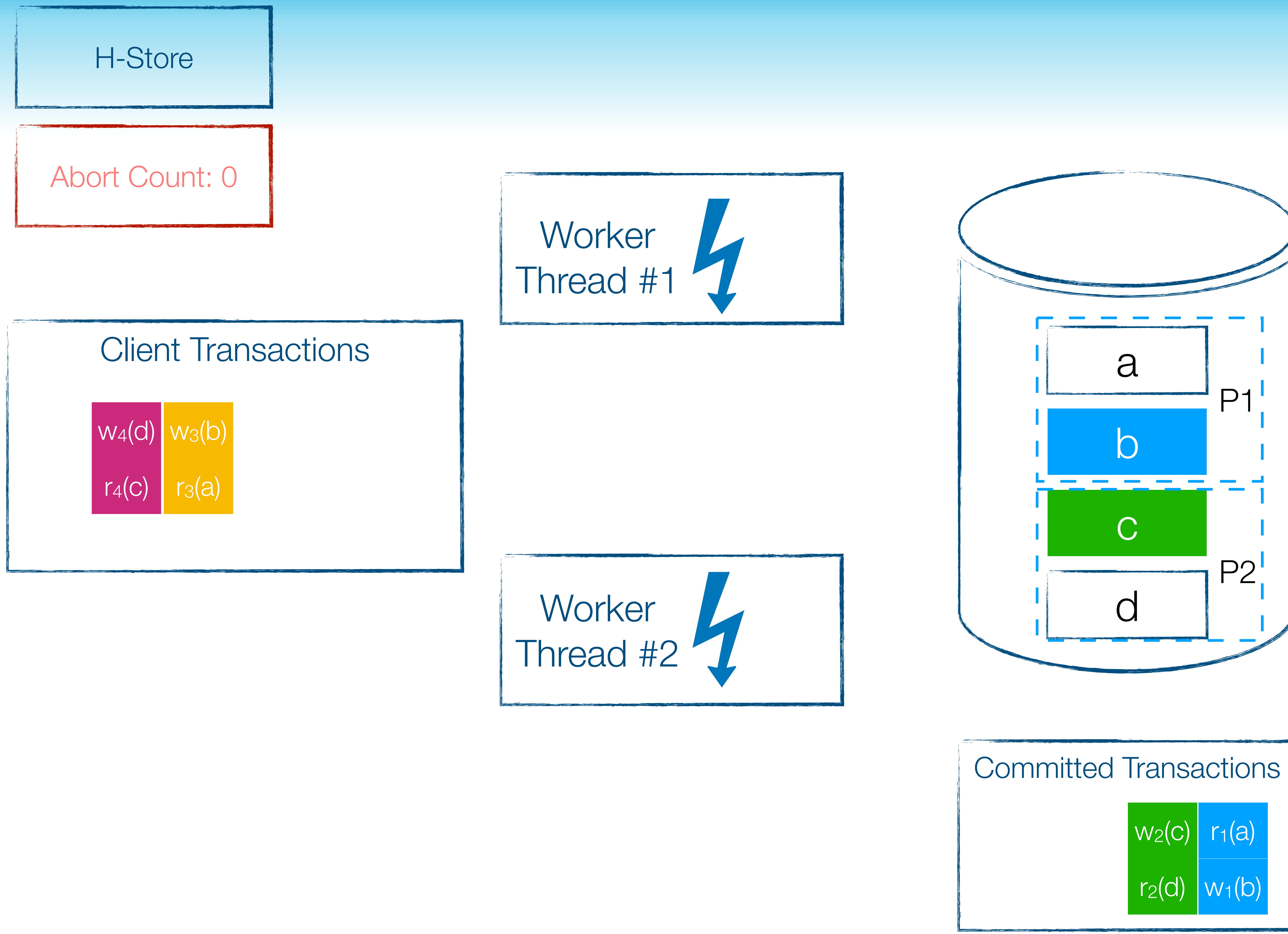
- Many aborts due to high contention
- Non-determinism in CC cause these aborts
- Can we do better?
- Is it possible to eliminate non-deterministic concurrency control from transaction execution?

Deterministic Transaction Execution

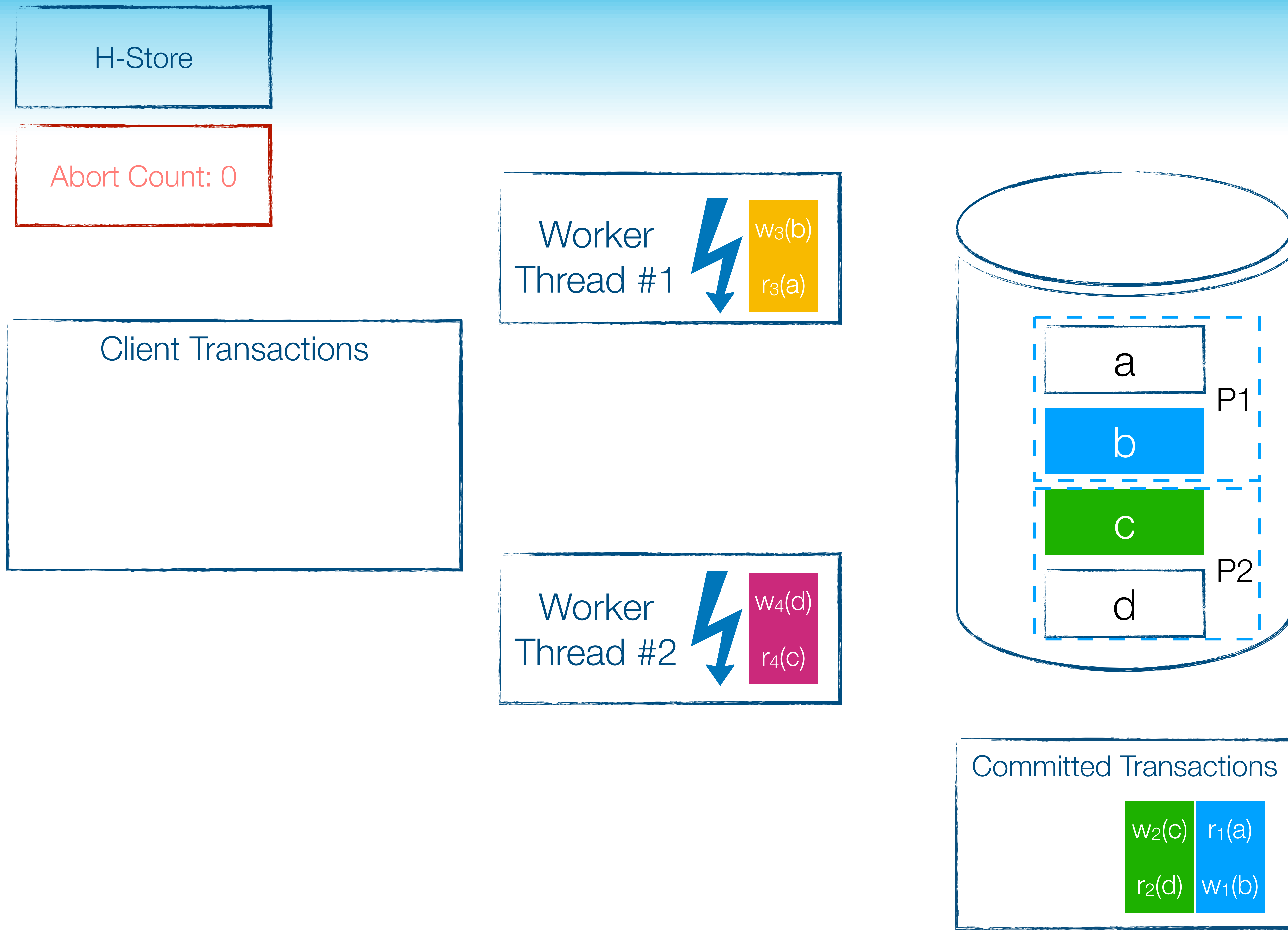
- H-Store [Kallman et al. '08]
- Designed and optimized for horizontal scalability, multi-core hardware and in-memory databases
- Stored procedure transaction model
- Static partitioning of database
- Assigns a single core to each partition
- Execute transaction serially without concurrency control within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

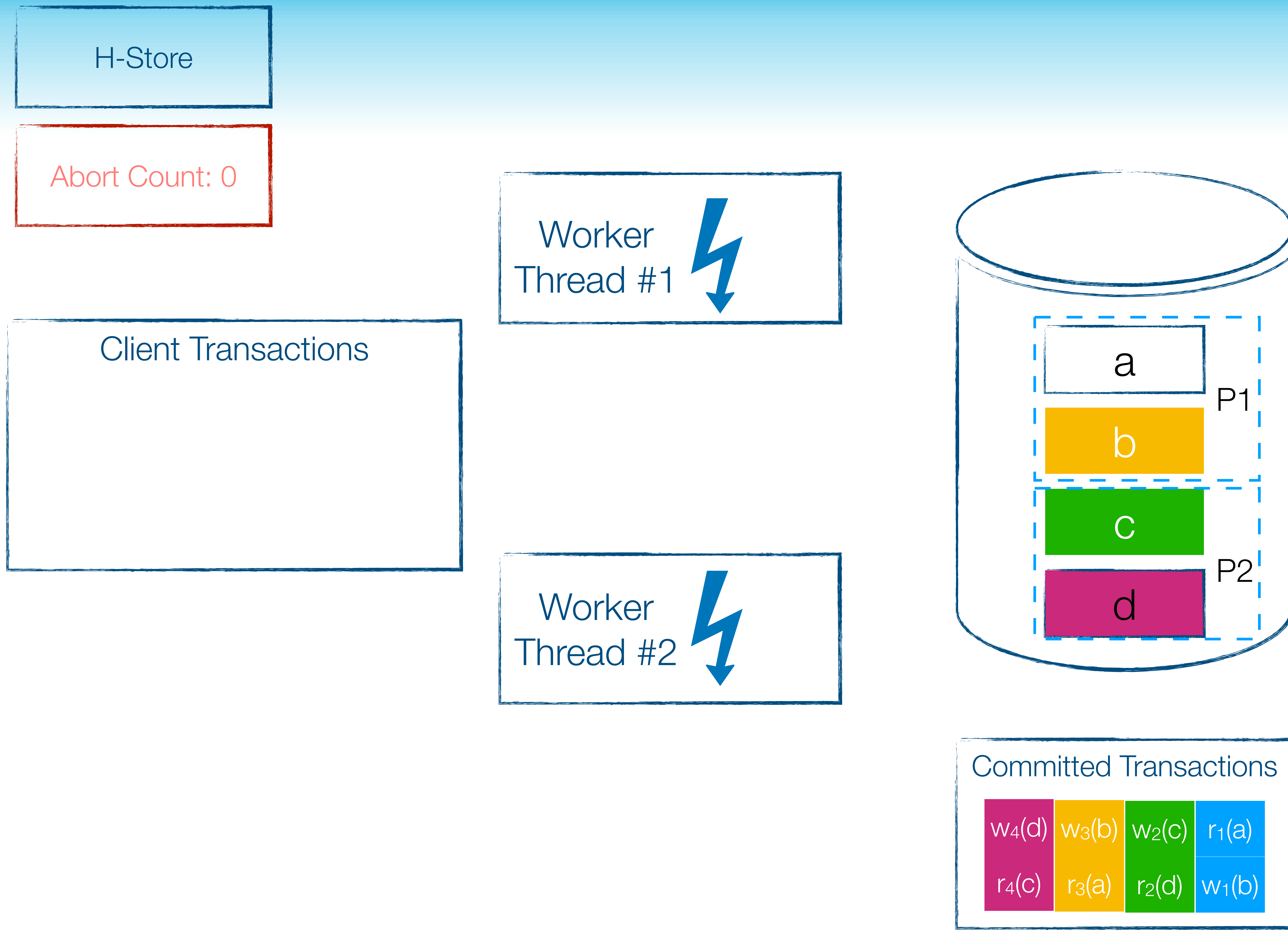

H-Store
Abort Count: 0
Client Transactions
w4(d) w3(b) w2(c) r1(a)
r4(c) r3(a) r2(d) w1(b)
Single-partition transactions
Worker Thread #1
Worker Thread #2
P1 is assigned to Worker Thread #1
P2 is assigned to Worker Thread #2
a
b
P1
c
d
P2


H-Store
Abort Count: 0
Client Transactions
w4(d)
r4(c)
w3(b)
r3(a)
Worker Thread #1
r1(a)
w1(b)
Worker Thread #2
w2(c)
r2(d)
a
b
P1
c
d
P2
Committed Transactions



H-Store
Abort Count: 0
Client Transactions
w4(d)
r4(c)
w3(b)
r3(a)
Worker Thread #1
Worker Thread #2
a
b
P1
c
d
P2
Committed Transactions
w2(c)
r2(d)
r1(a)
w1(b)


H-Store
Abort Count: 0
Client Transactions
Worker Thread #1
w3(b)
r3(a)
Worker Thread #2
w4(d)
r4(c)
a
b
P1
c
d
P2
Committed Transactions
w2(c)
r2(d)
r1(a)
w1(b)


H-Store
Abort Count: 0
Client Transactions
Worker Thread #1
Worker Thread #2
a
b
P1
c
d
P2
Committed Transactions
$w_4(d)$ $w_3(b)$ $w_2(c)$ $r_1(a)$
$r_4(c)$ $r_3(a)$ $r_2(d)$ $w_1(b)$

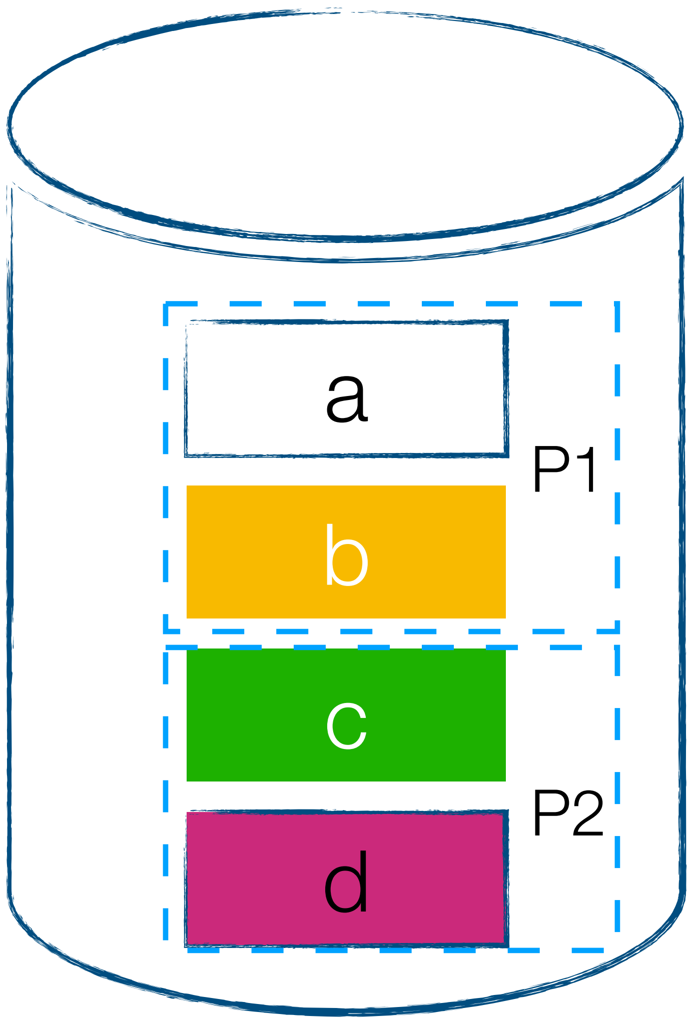
H-Store

Abort Count: 0

Client Transactions

Worker Thread #1

Worker Thread #2



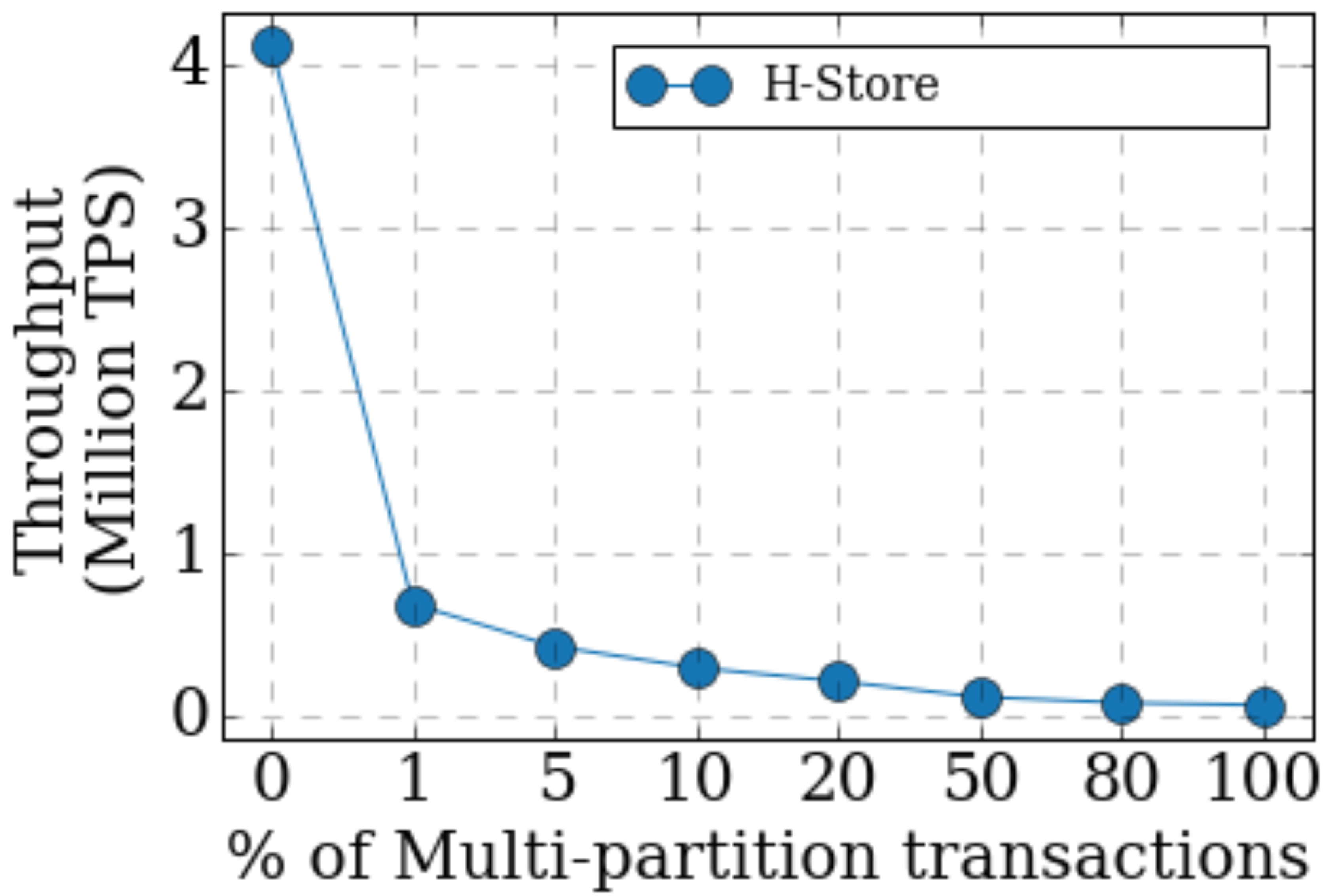
- ✓ Deterministic Execution
- ✓ No aborts because of CC
- ✓ Minimal coordination among threads

◉ 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***

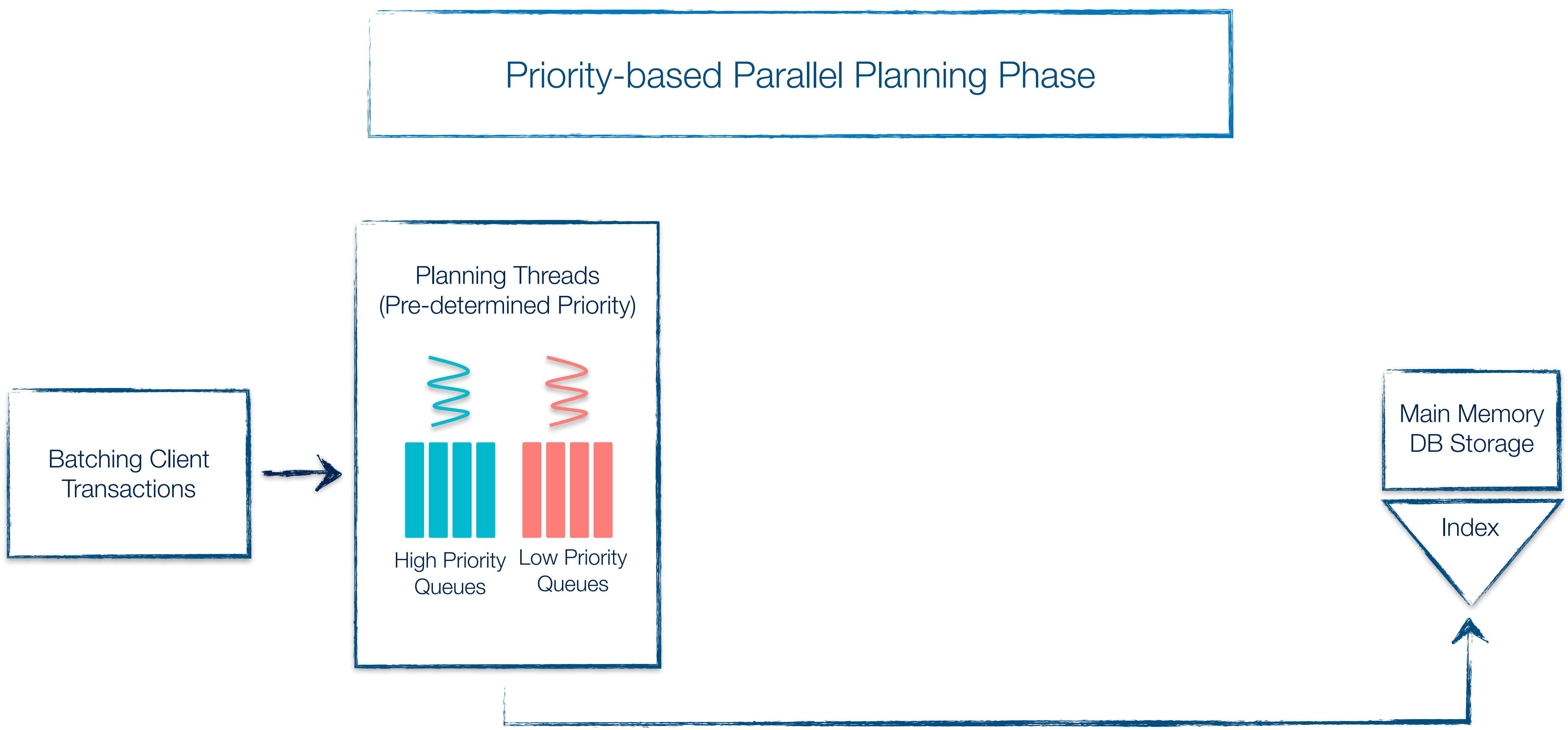
"For each record (or a queue), operations that belong to higher priority queues (created by a higher priority planner) must always be executed before executing any lower priority operations."

QueCC Architecture

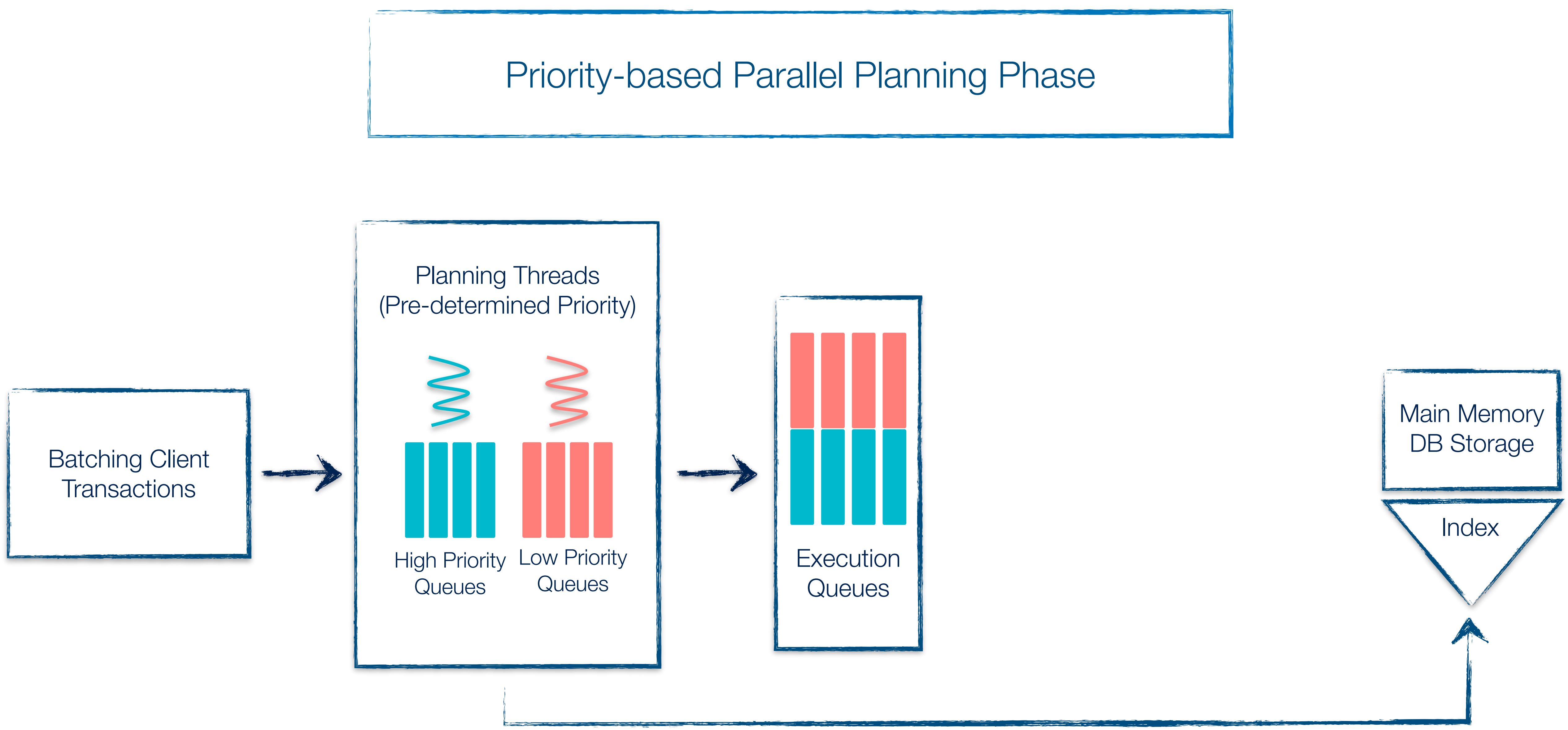
Priority-based Parallel Planning Phase

Batching Client Transactions

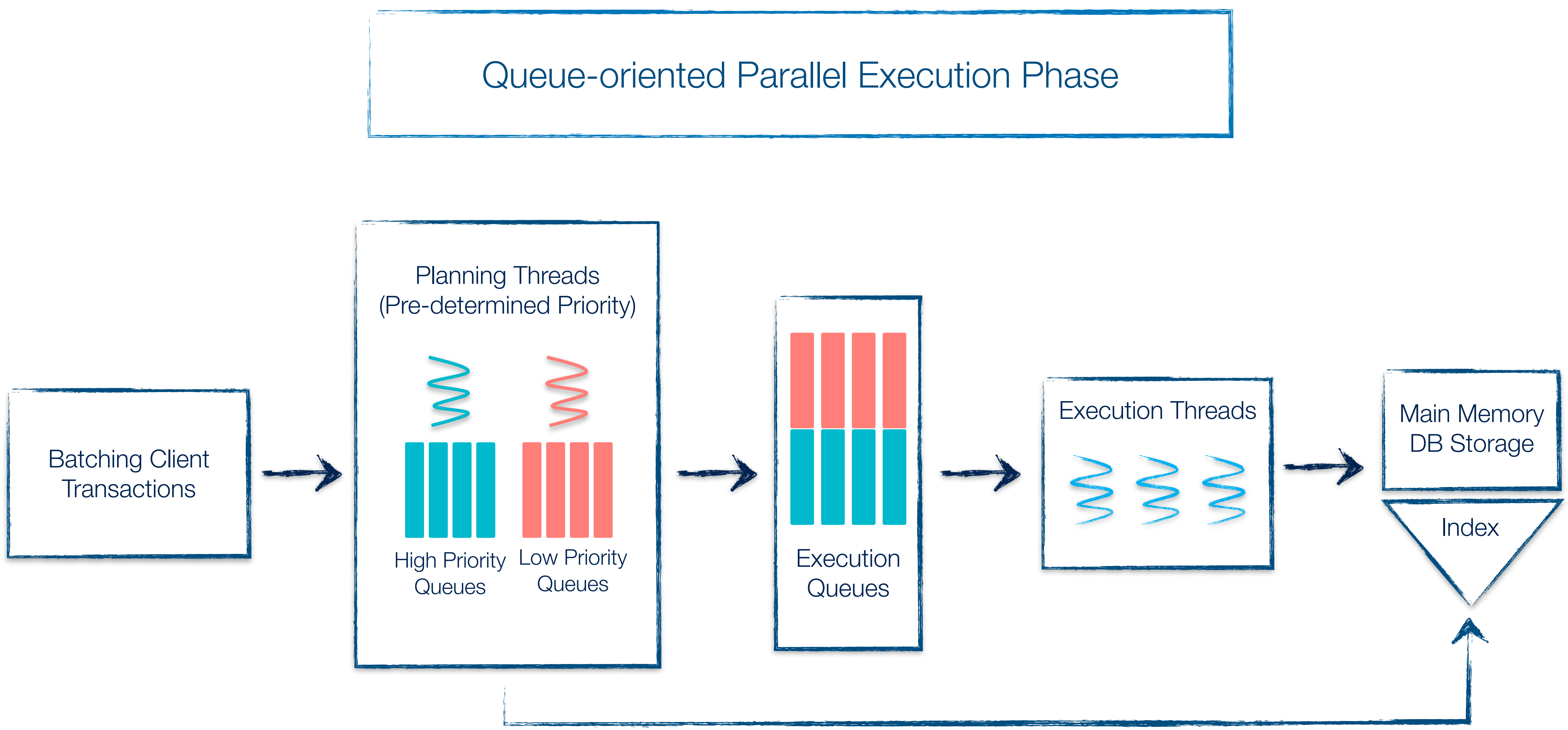
QueCC Architecture

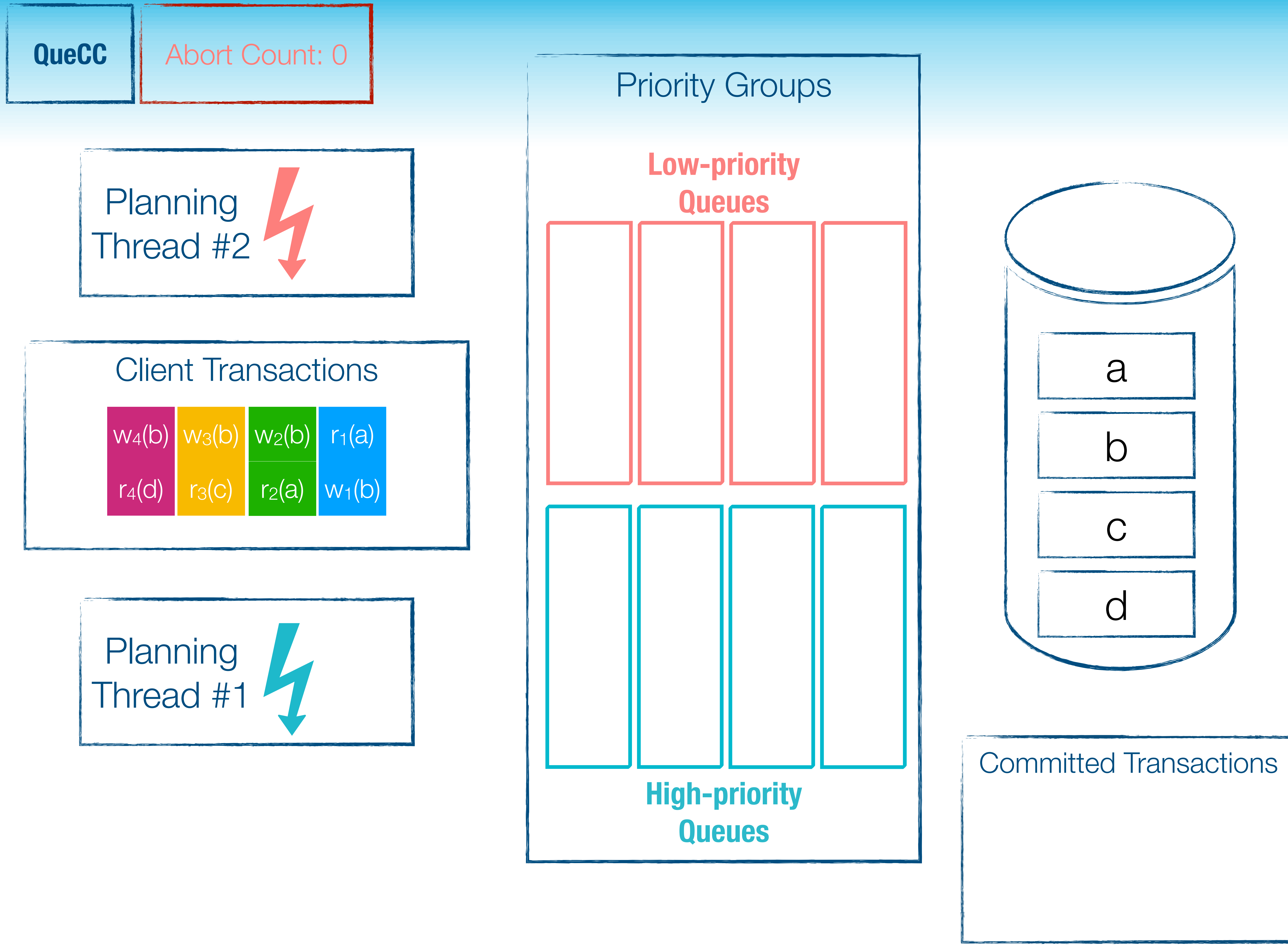


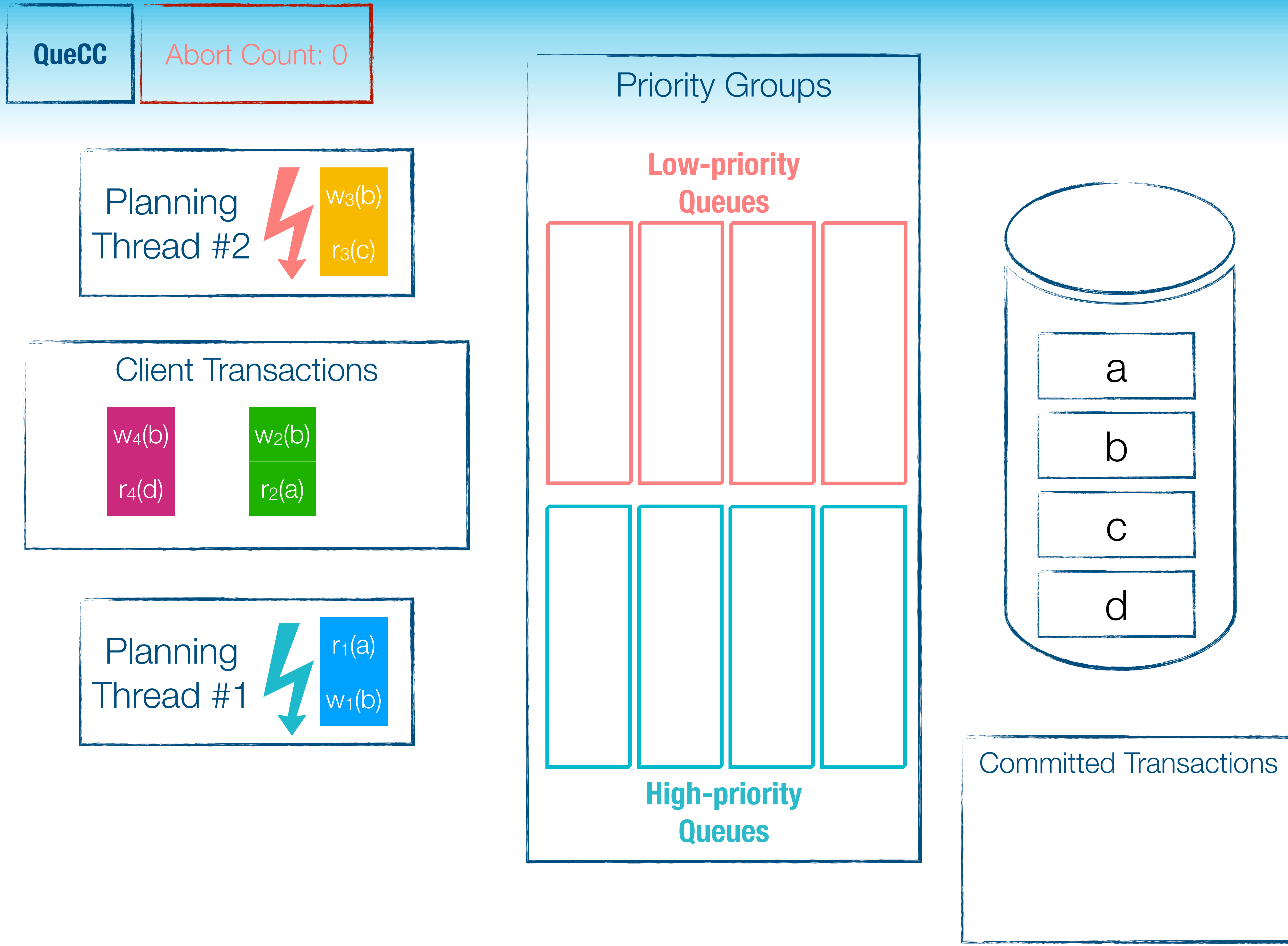
QueCC Architecture

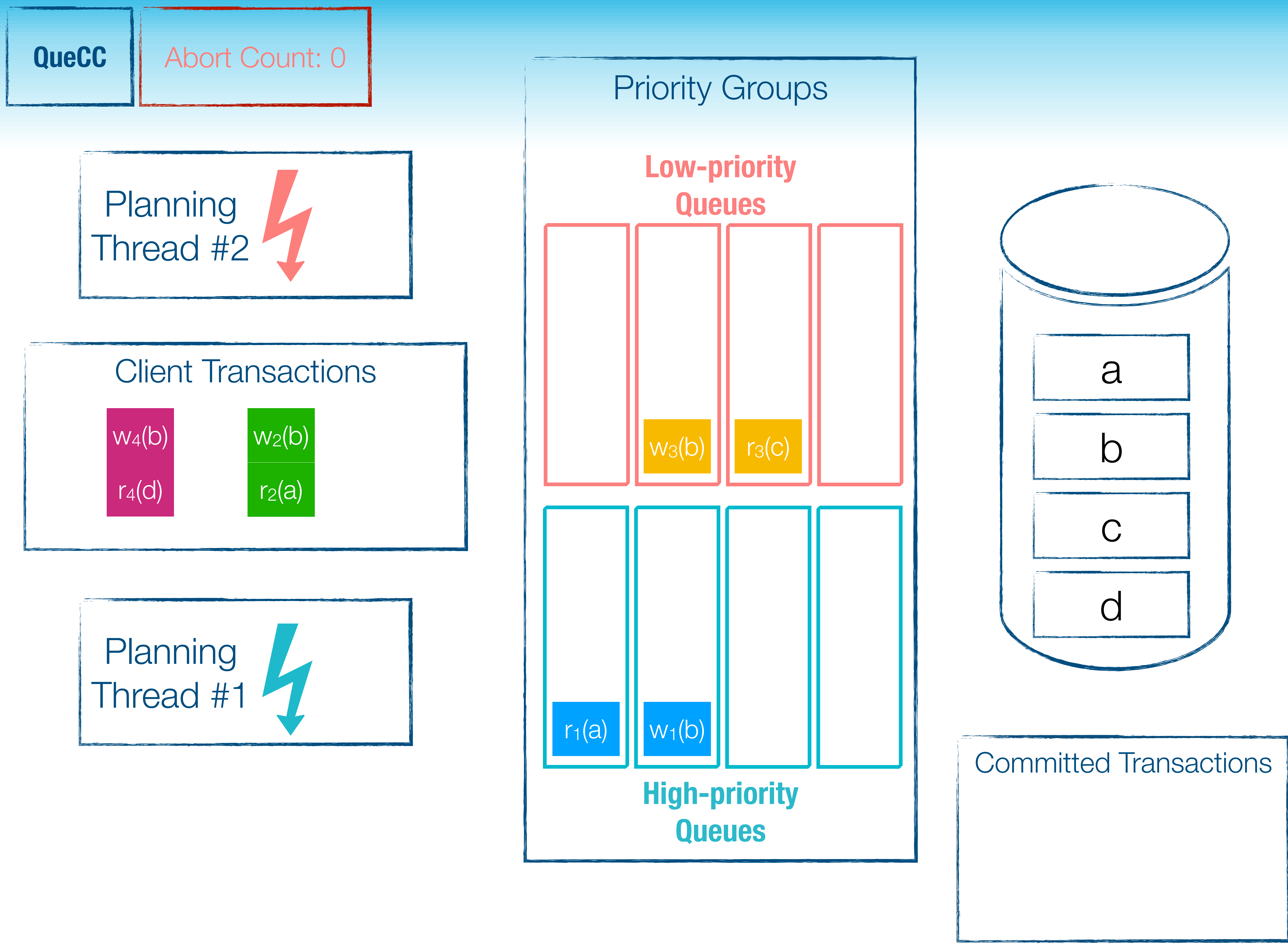


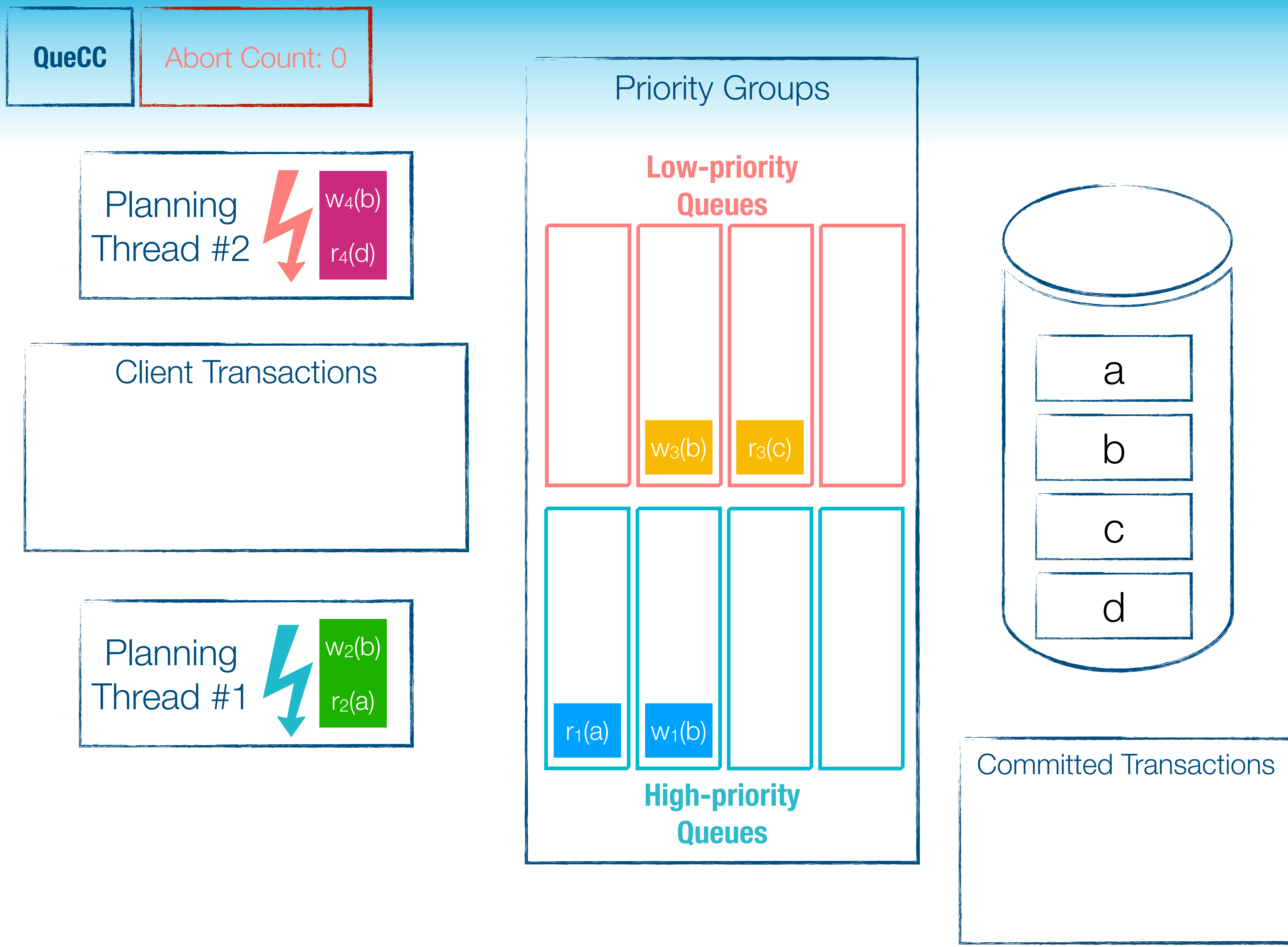
QueCC Architecture

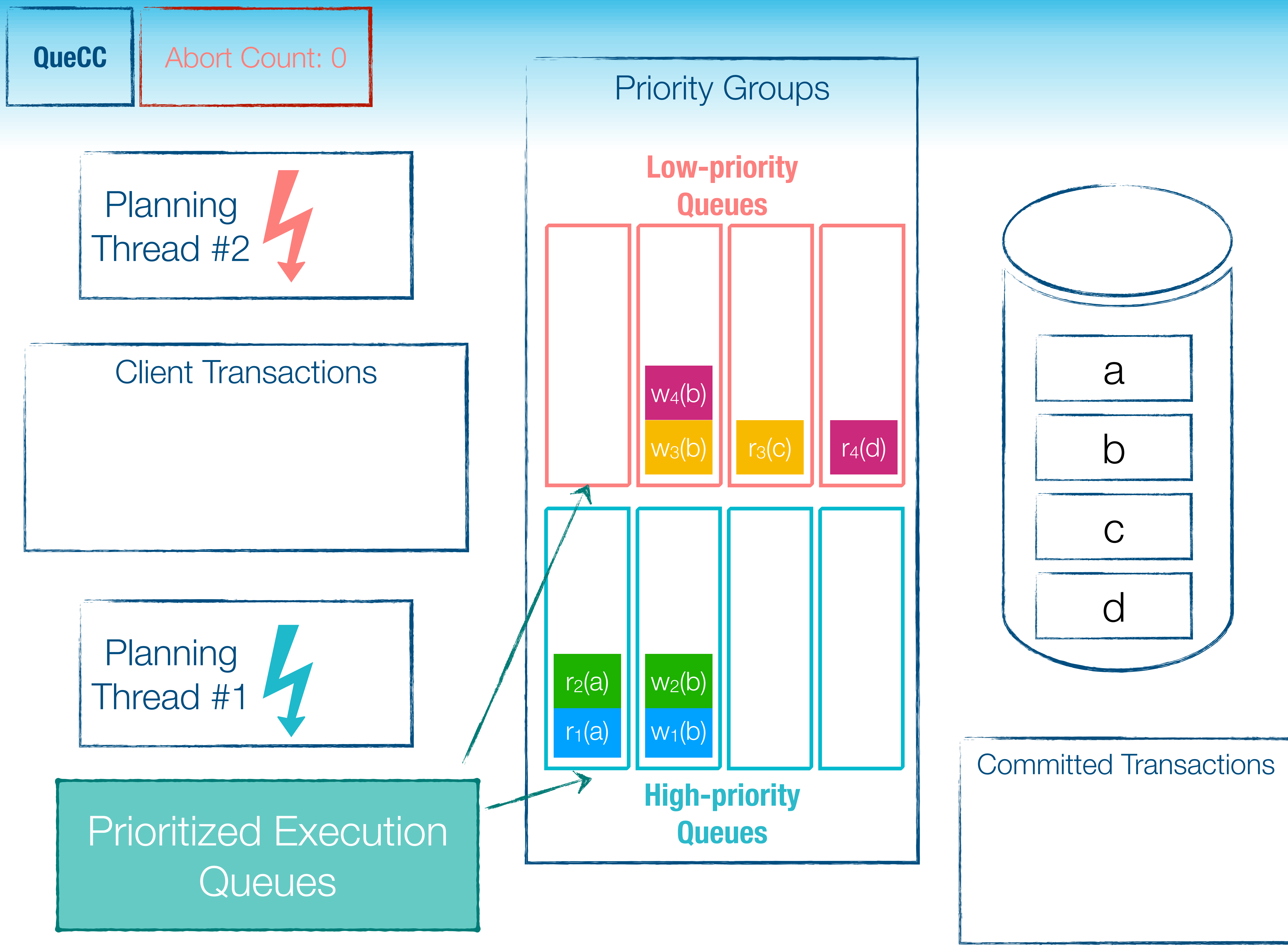


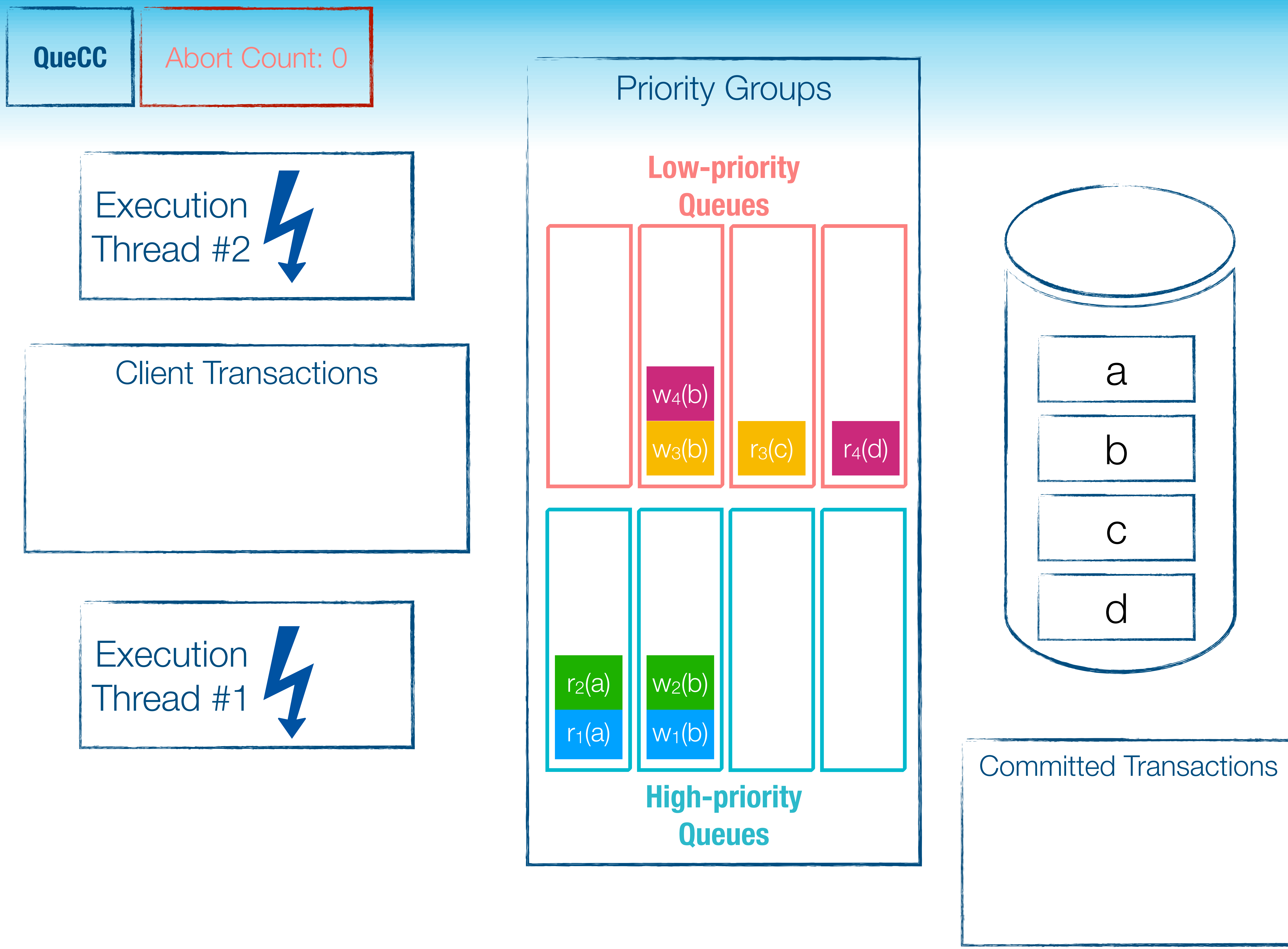

QueCC
Abort Count: 0
Planning Thread #2
Client Transactions
w4(b)
r4(d)
w3(b)
r3(c)
w2(b)
r2(a)
r1(a)
w1(b)
Planning Thread #1
Priority Groups
Low-priority Queues
High-priority Queues
a
b
c
d
Committed Transactions

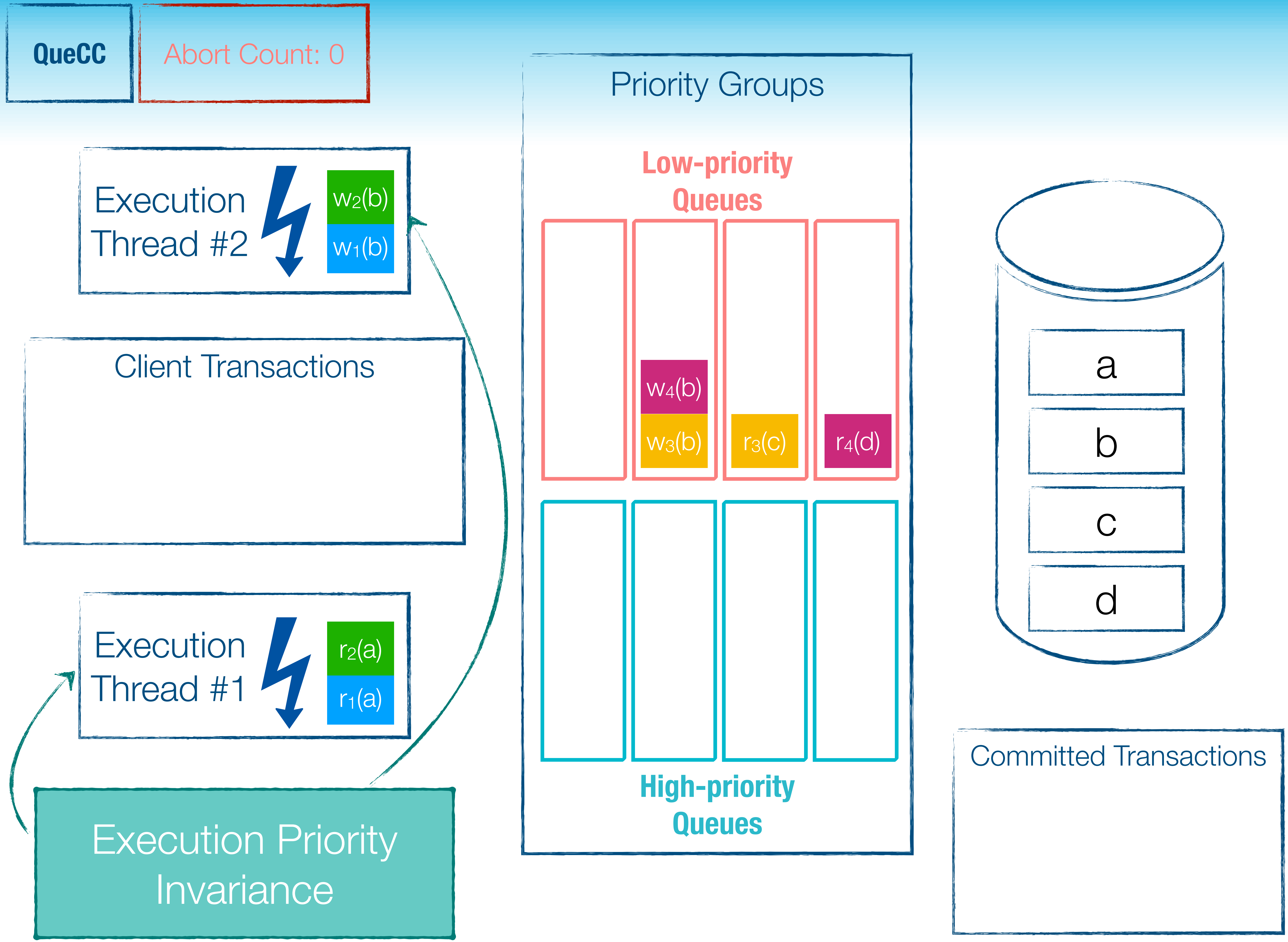

QueCC
Abort Count: 0
Priority Groups
Low-priority Queues
High-priority Queues
Planning Thread #2
w3(b)
r3(c)
Client Transactions
w4(b)
r4(d)
w2(b)
r2(a)
Planning Thread #1
r1(a)
w1(b)
a
b
c
d
Committed Transactions


QueCC
Abort Count: 0
Planning Thread #2
Client Transactions
w4(b)
r4(d)
w2(b)
r2(a)
Planning Thread #1
Priority Groups
Low-priority Queues
w3(b)
r3(c)
r1(a)
w1(b)
High-priority Queues
a
b
c
d
Committed Transactions

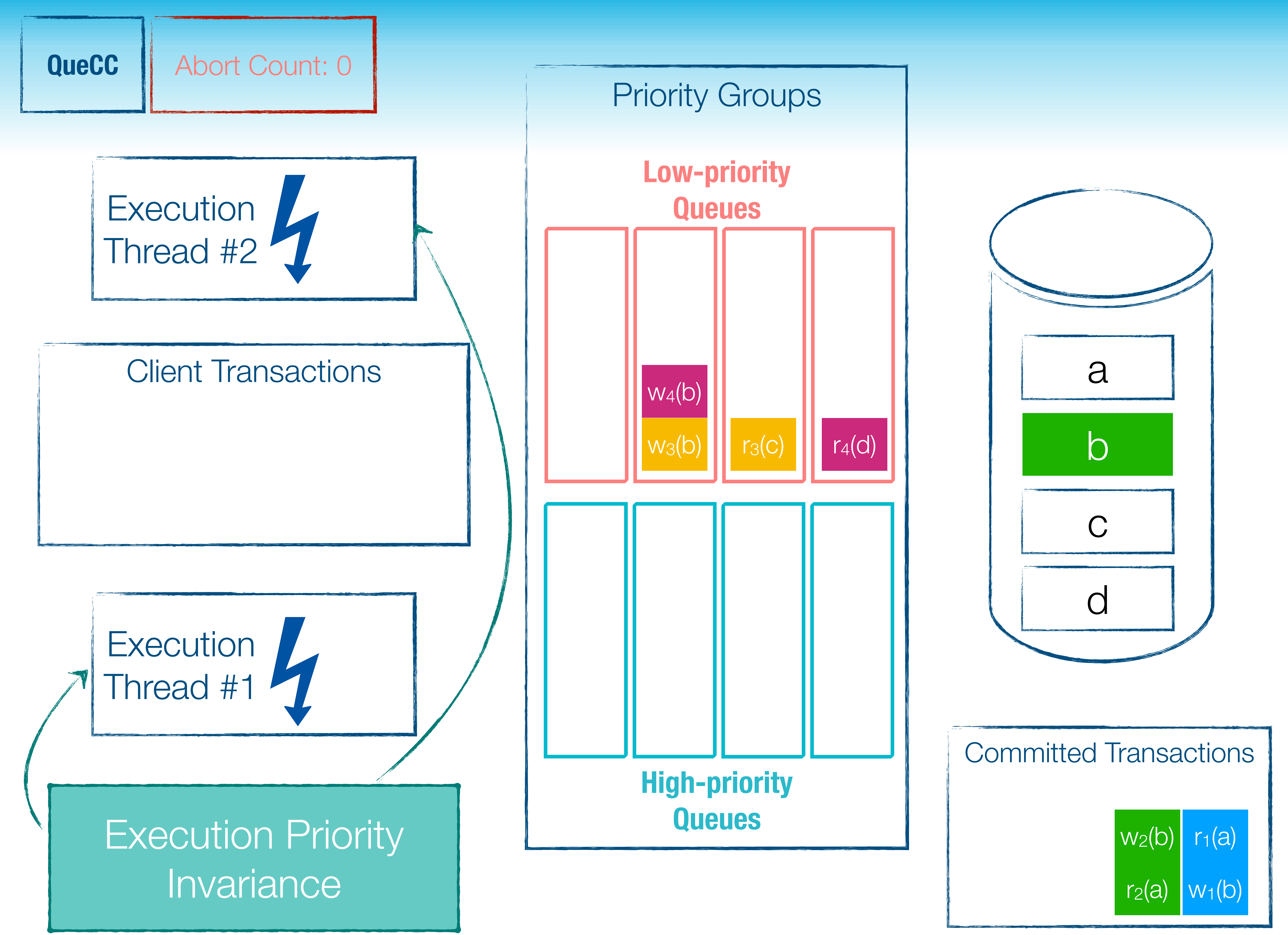

QueCC
Abort Count: 0
Planning Thread #2
w4(b)
r4(d)
Client Transactions
Planning Thread #1
w2(b)
r2(a)
Priority Groups
Low-priority Queues
w3(b)
r3(c)
r1(a)
w1(b)
High-priority Queues
a
b
c
d
Committed Transactions


QueCC
Abort Count: 0
Planning Thread #2
Client Transactions
Planning Thread #1
Prioritized Execution Queues
Priority Groups
Low-priority Queues
$w_4(b)$
$w_3(b)$
$r_3(c)$
$r_4(d)$
$r_2(a)$
$r_1(a)$
$w_2(b)$
$w_1(b)$
High-priority Queues
a
b
c
d
Committed Transactions

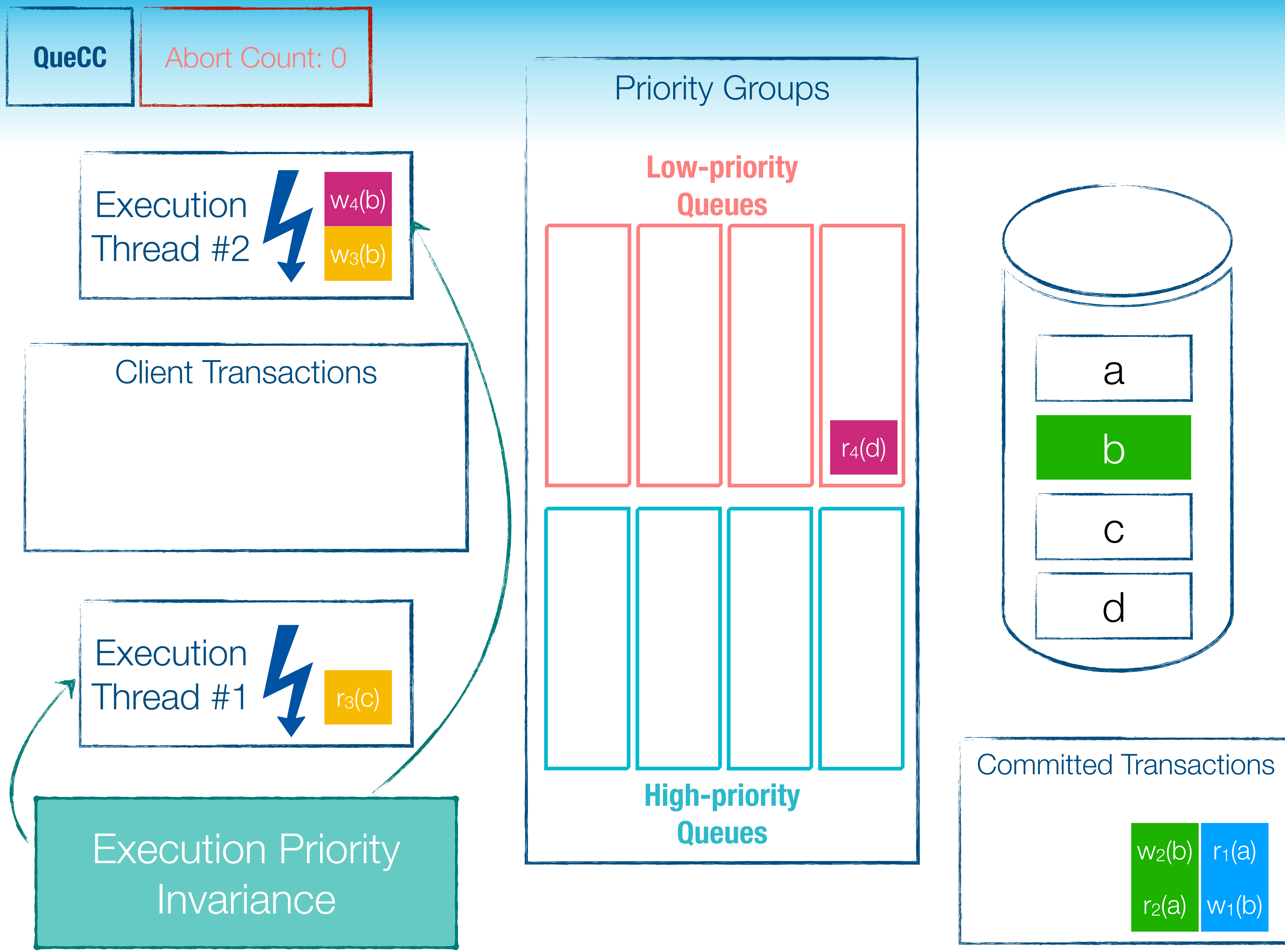

QueCC
Abort Count: 0
Priority Groups
Low-priority Queues
$w_4(b)$
$w_3(b)$
$r_3(c)$
$r_4(d)$
$r_2(a)$
$r_1(a)$
$w_2(b)$
$w_1(b)$
High-priority Queues
Execution Thread #2
Client Transactions
Execution Thread #1
a
b
c
d
Committed Transactions



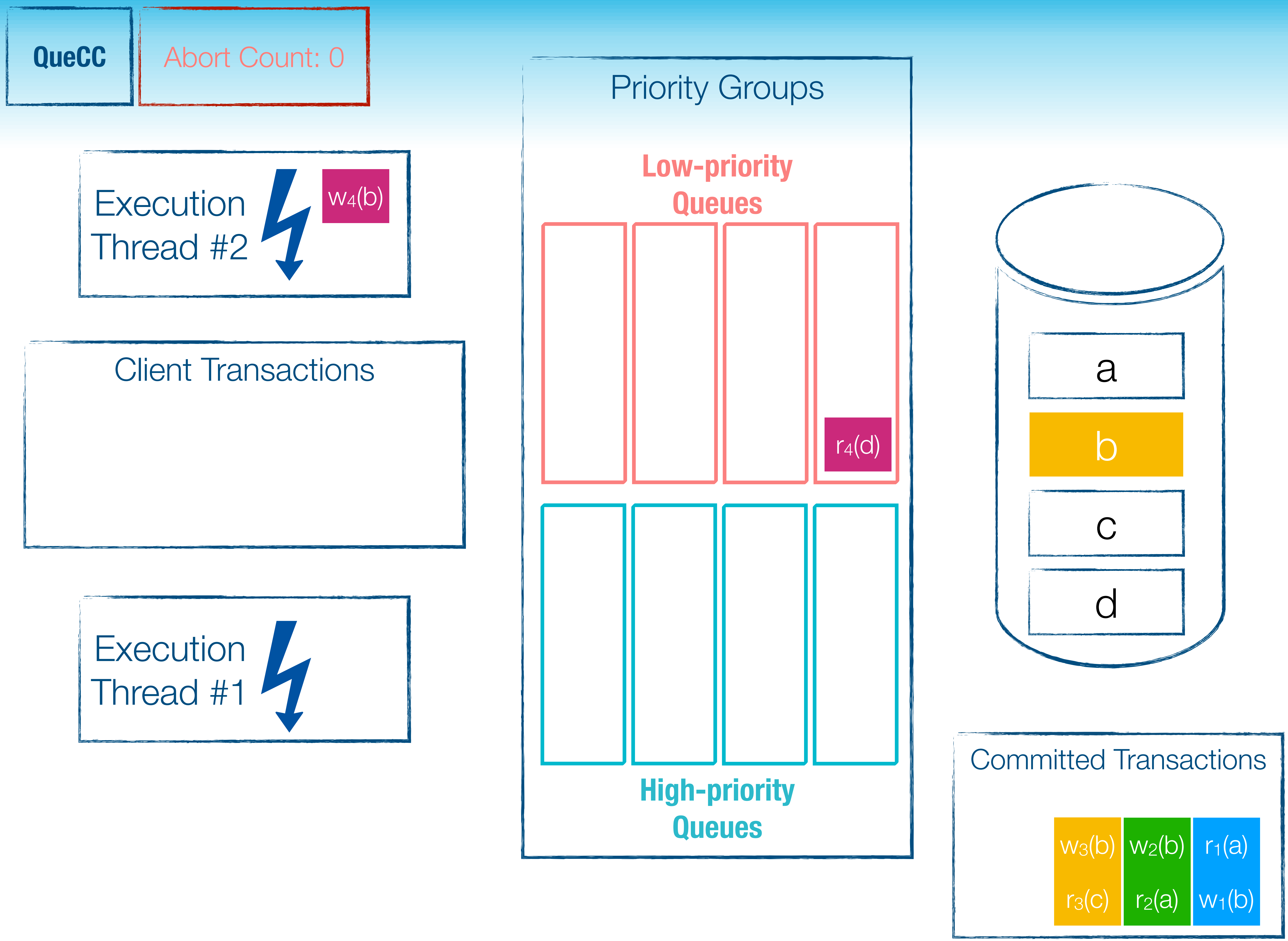
QueCC
Abort Count: 0
Execution Thread #2
w2(b)
w1(b)
Client Transactions
Execution Thread #1
r2(a)
r1(a)
Execution Priority Invariance
Priority Groups
Low-priority Queues
w4(b)
w3(b)
r3(c)
r4(d)
High-priority Queues
a
b
c
d
Committed Transactions

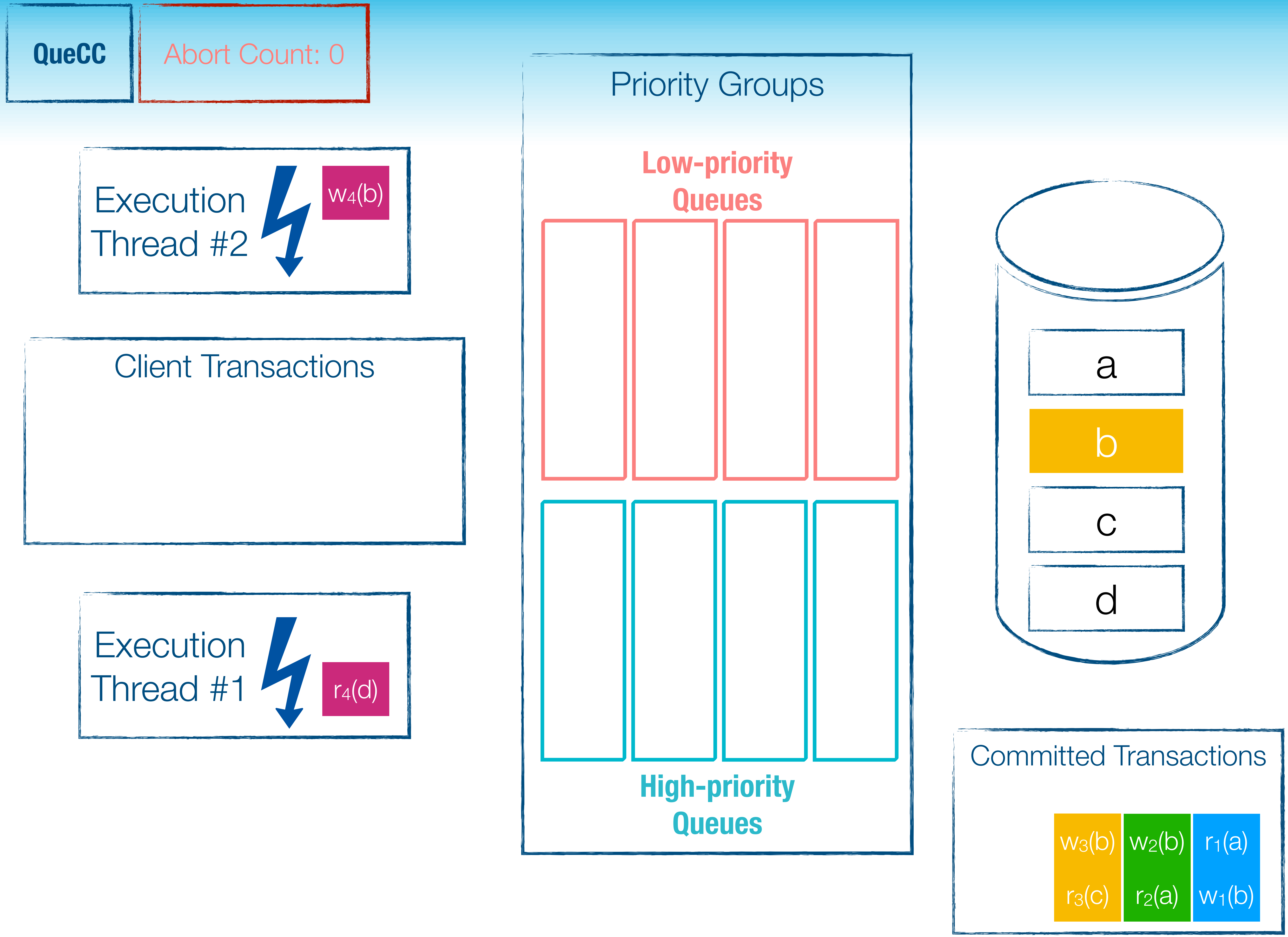


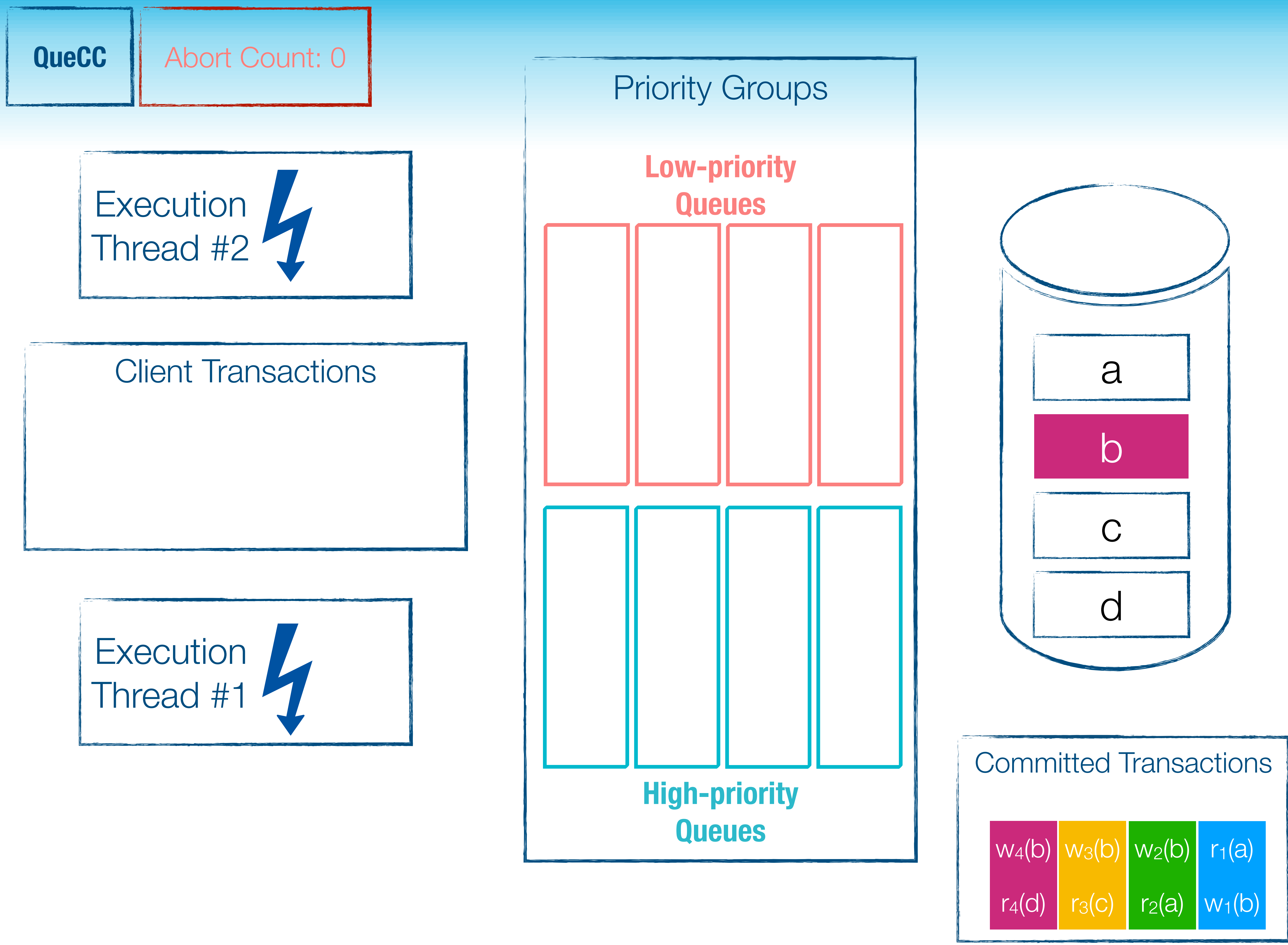
QueCC
Abort Count: 0
Execution Thread #2
Client Transactions
Execution Thread #1
Execution Priority Invariance
Priority Groups
Low-priority Queues
w4(b)
w3(b)
r3(c)
r4(d)
High-priority Queues
a
b
c
d
Committed Transactions
w2(b)
r2(a)
r1(a)
w1(b)



QueCC
Abort Count: 0
Execution Thread #2
w4(b)
w3(b)
Client Transactions
Execution Thread #1
r3(c)
Execution Priority Invariance
Priority Groups
Low-priority Queues
r4(d)
High-priority Queues
a
b
c
d
Committed Transactions
w2(b)
r2(a)
r1(a)
w1(b)

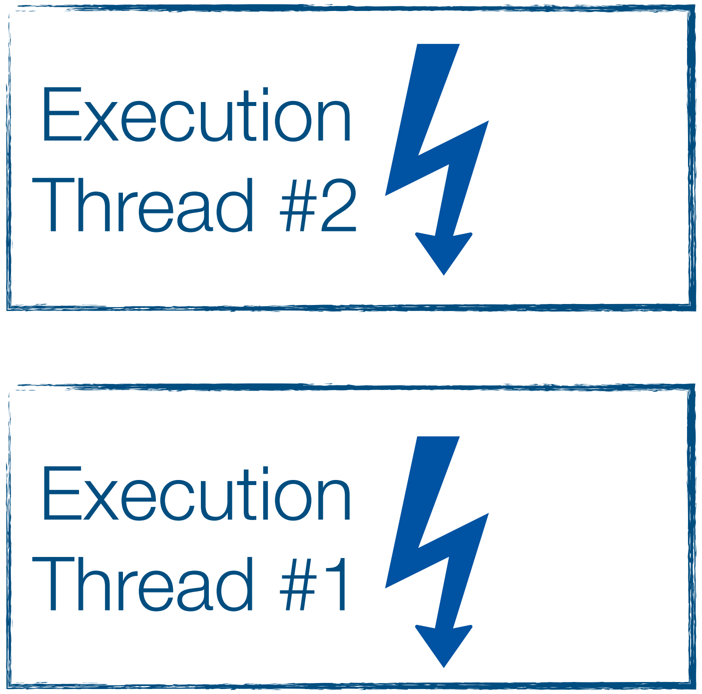

QueCC
Abort Count: 0
Execution Thread #2
$w_4(b)$
Client Transactions
Execution Thread #1
Priority Groups
Low-priority Queues
$r_4(d)$
High-priority Queues
a
b
c
d
Committed Transactions
$w_3(b)$
$r_3(c)$
$w_2(b)$
$r_2(a)$
$r_1(a)$
$w_1(b)$


QueCC
Abort Count: 0
Execution Thread #2
w4(b)
Client Transactions
Execution Thread #1
r4(d)
Priority Groups
Low-priority Queues
High-priority Queues
a
b
c
d
Committed Transactions
w3(b)
r3(c)
w2(b)
r2(a)
r1(a)
w1(b)

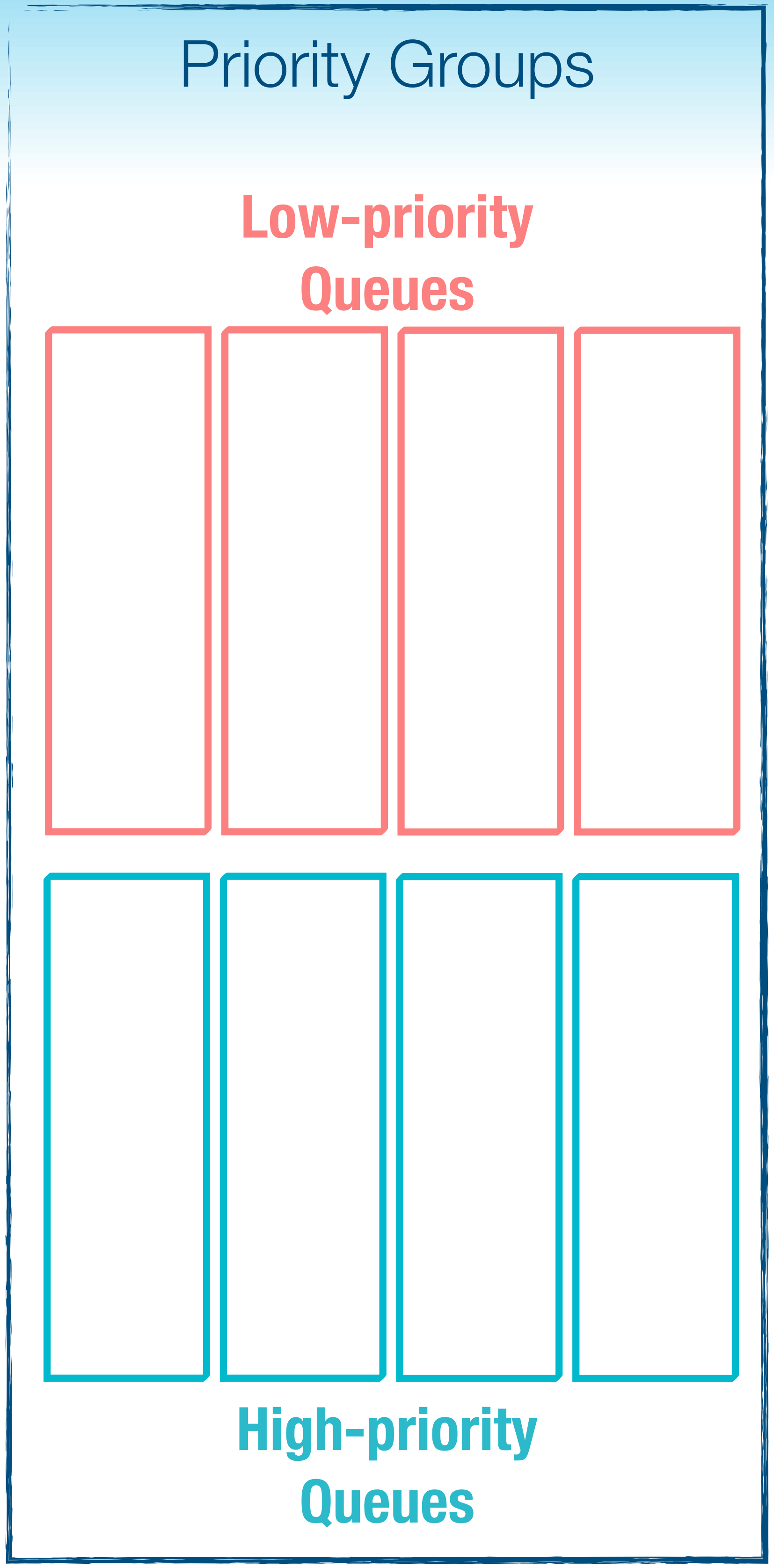

QueCC
Abort Count: 0
Execution Thread #2
Client Transactions
Execution Thread #1
Priority Groups
Low-priority Queues
High-priority Queues
a
b
c
d
Committed Transactions
w4(b) r4(d)
w3(b) r3(c)
w2(b) r2(a)
r1(a) w1(b)

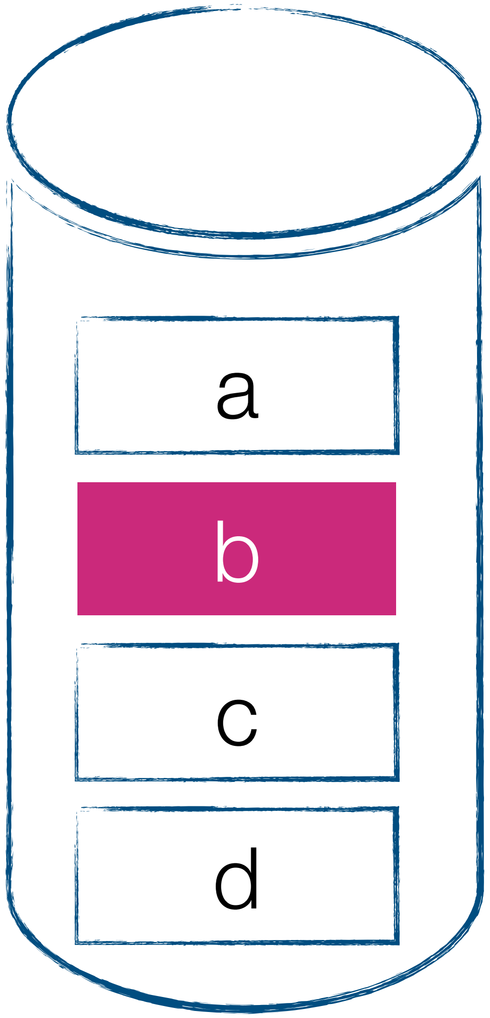
QueCC

Abort Count: 0



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

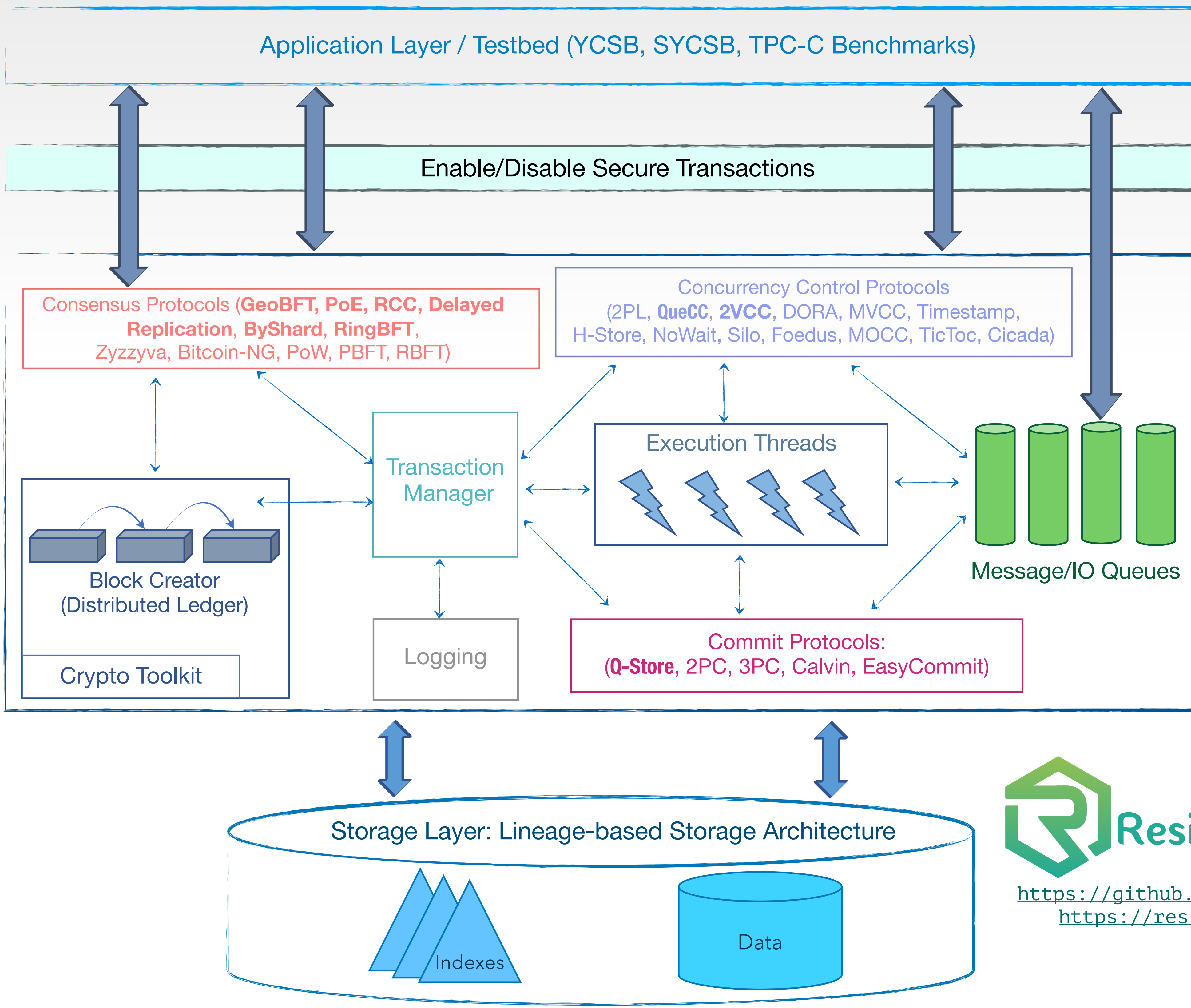




Committed Transactions

$w_4(b)$	$w_3(b)$	$w_2(b)$	$r_1(a)$
$r_4(d)$	$r_3(c)$	$r_2(a)$	$w_1(b)$

ResilientDB Blockchain Fabric



ResilientDB

https://github.com/resilientdb/
https://resilientdb.com/

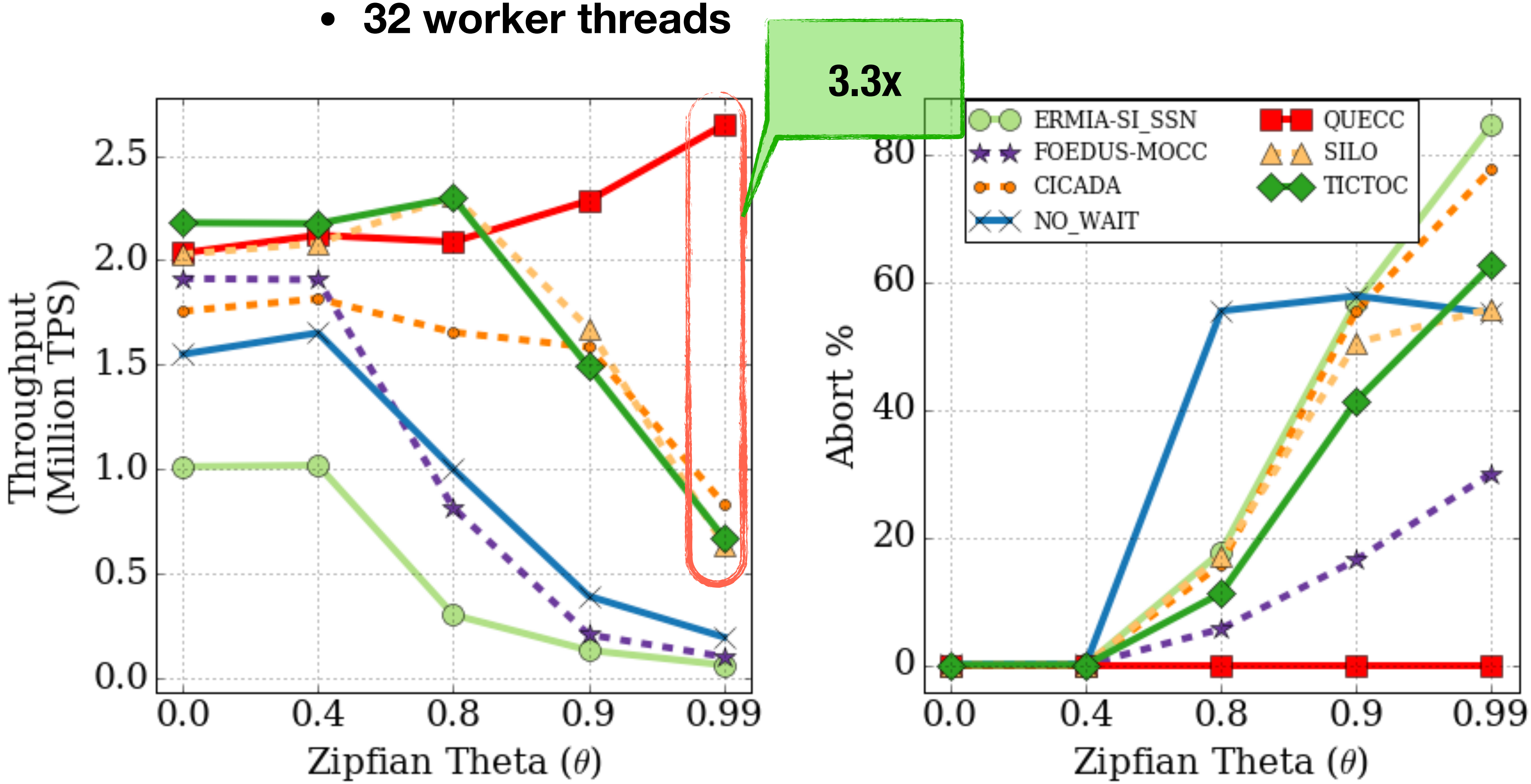
Fault-tolerant Distributed Transactions on Blockchain., S. Gupta, J. Hellings, M. Sadoghi

Evaluation Environment

Hardware	Microsoft Azure instance with 32 core CPU: Intel Xeon E5-2698B v3 *32KB L1 data an 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

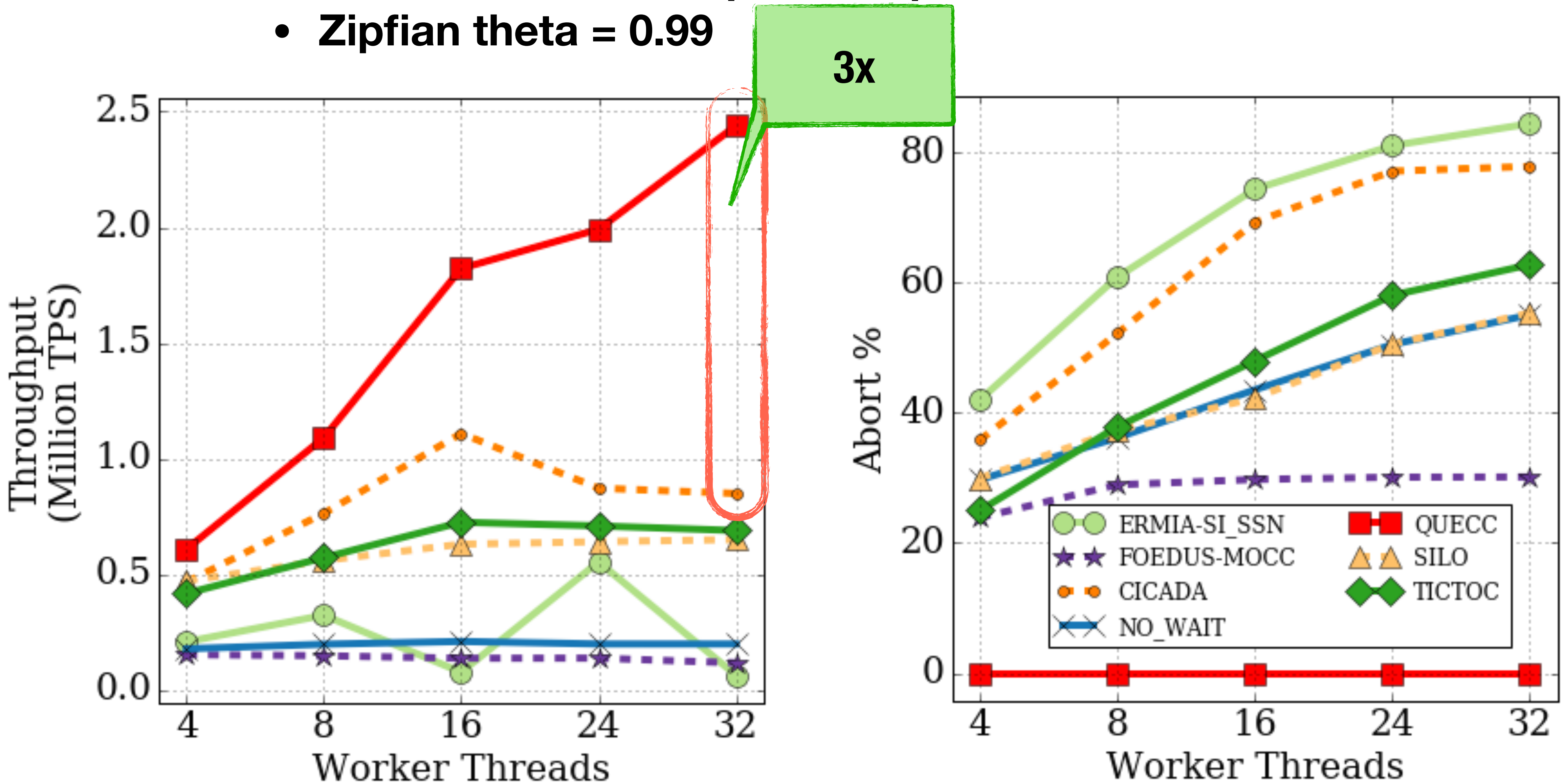
- **5 write and 5 read operation per transaction**
- **32 worker threads**



Workload contention resiliency
Cache locality under high-contention

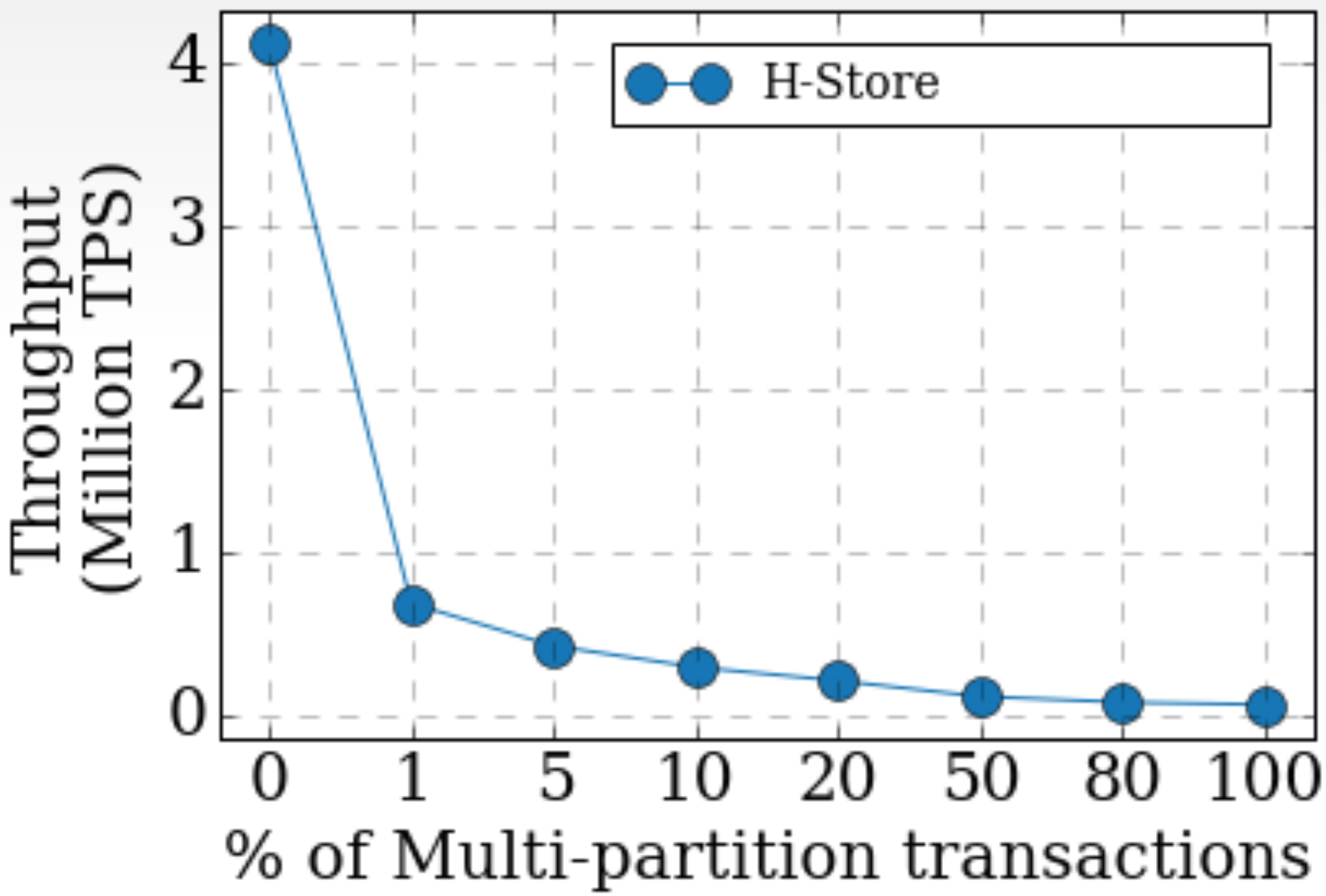
Effect of Varying Worker Threads

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

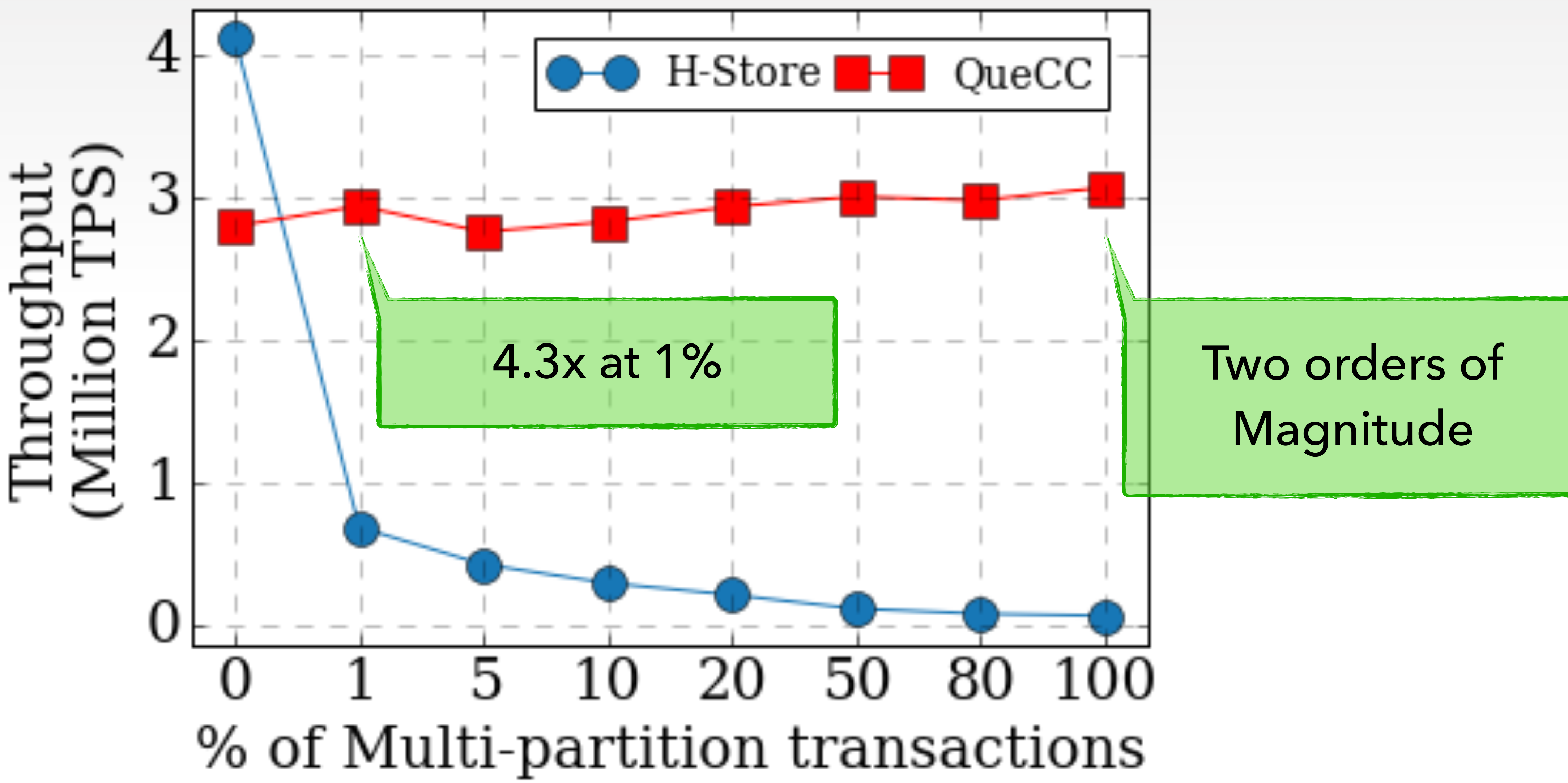


Avoiding thread coordination & eliminating all execution-induced aborts

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



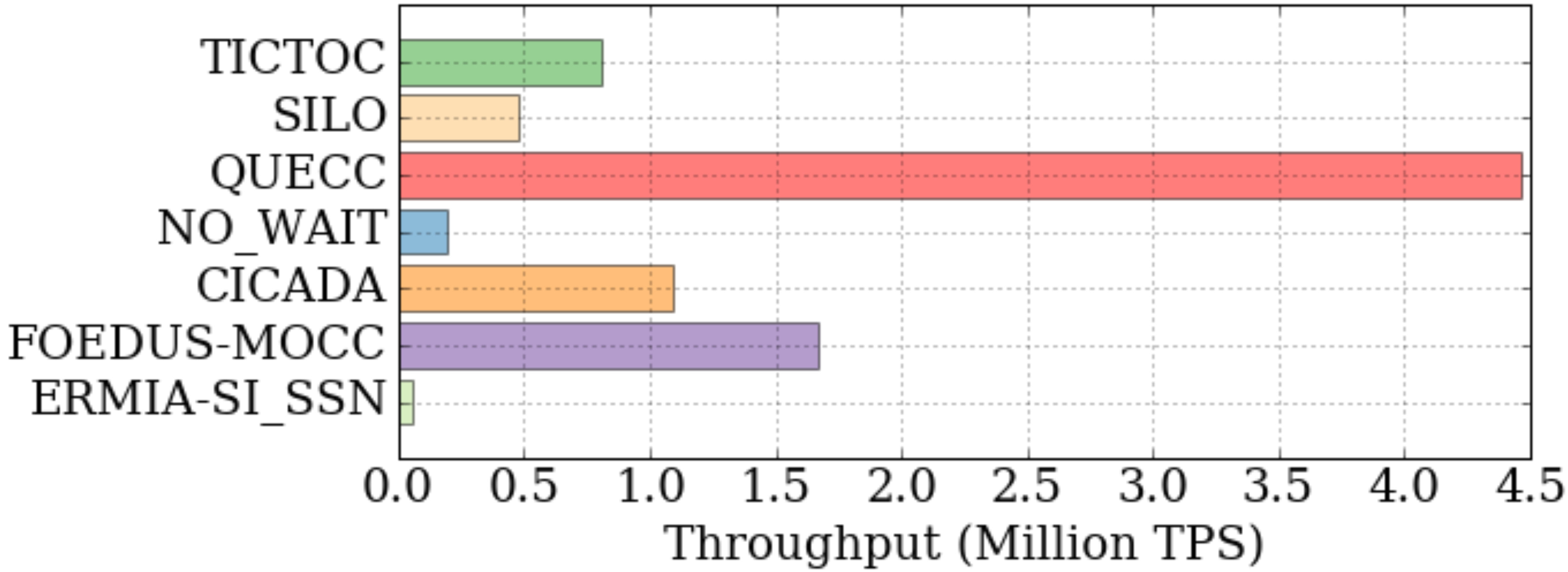
Effect of Increasing Percentage of Multi-Partition Transactions in the Workload



QueCC is not sensitive to multi-partitioning

TPC-C Results

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

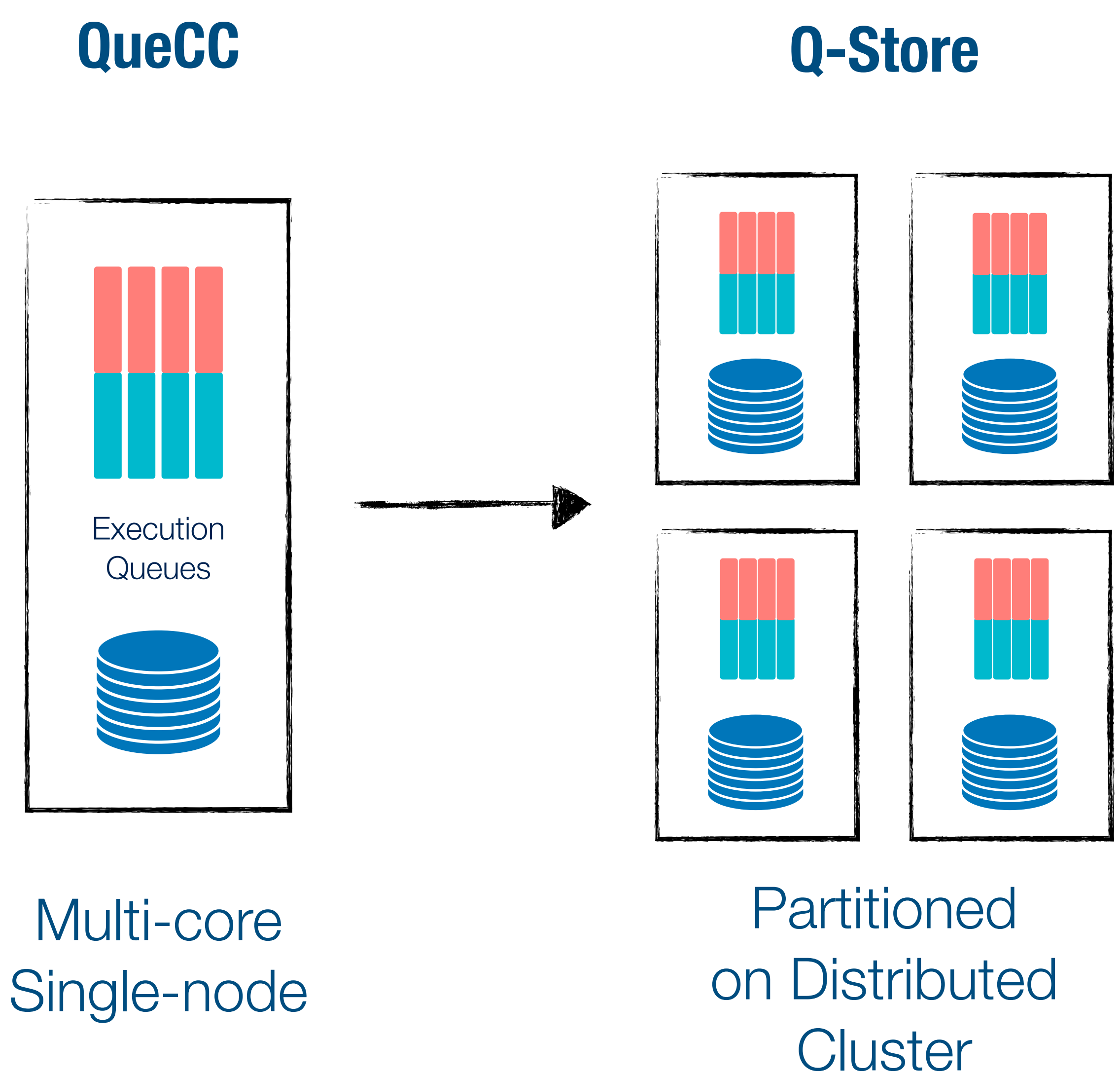


QueCC can achieve up to 3x better performance on high-contention TPC-C workloads

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



Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020

What’s Next: Q-Store

Plan Local and Remote
Execution Queues

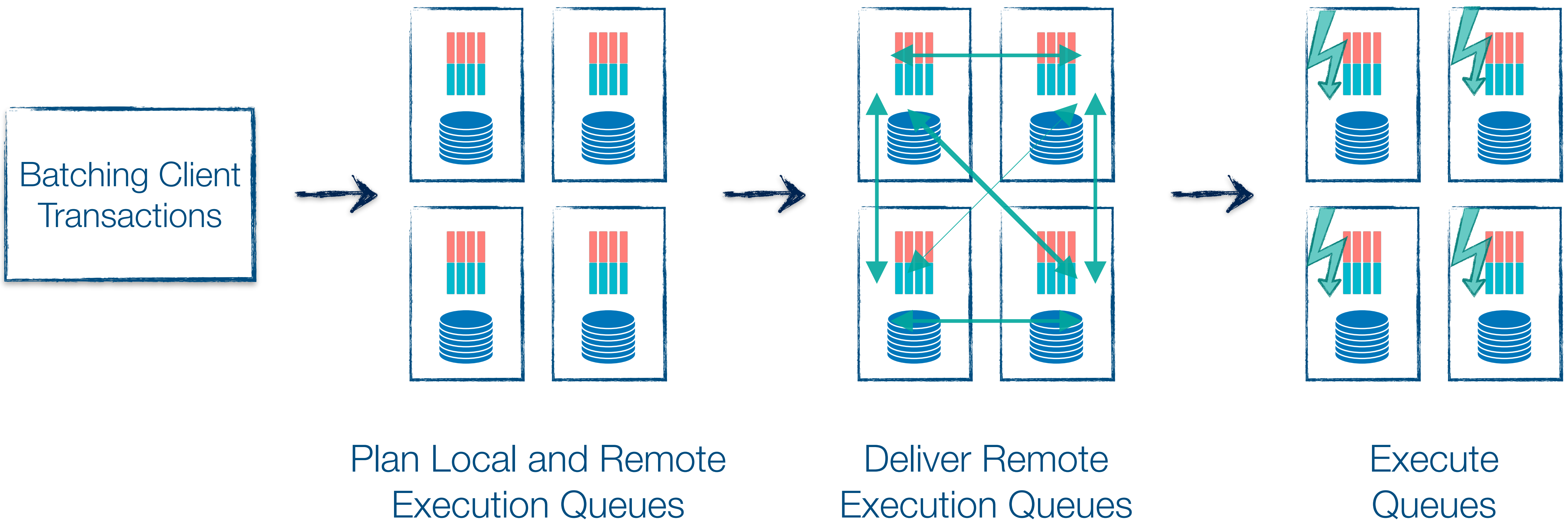
What's Next: Q-Store

Plan Local and Remote Execution Queues

Deliver Remote Execution Queues

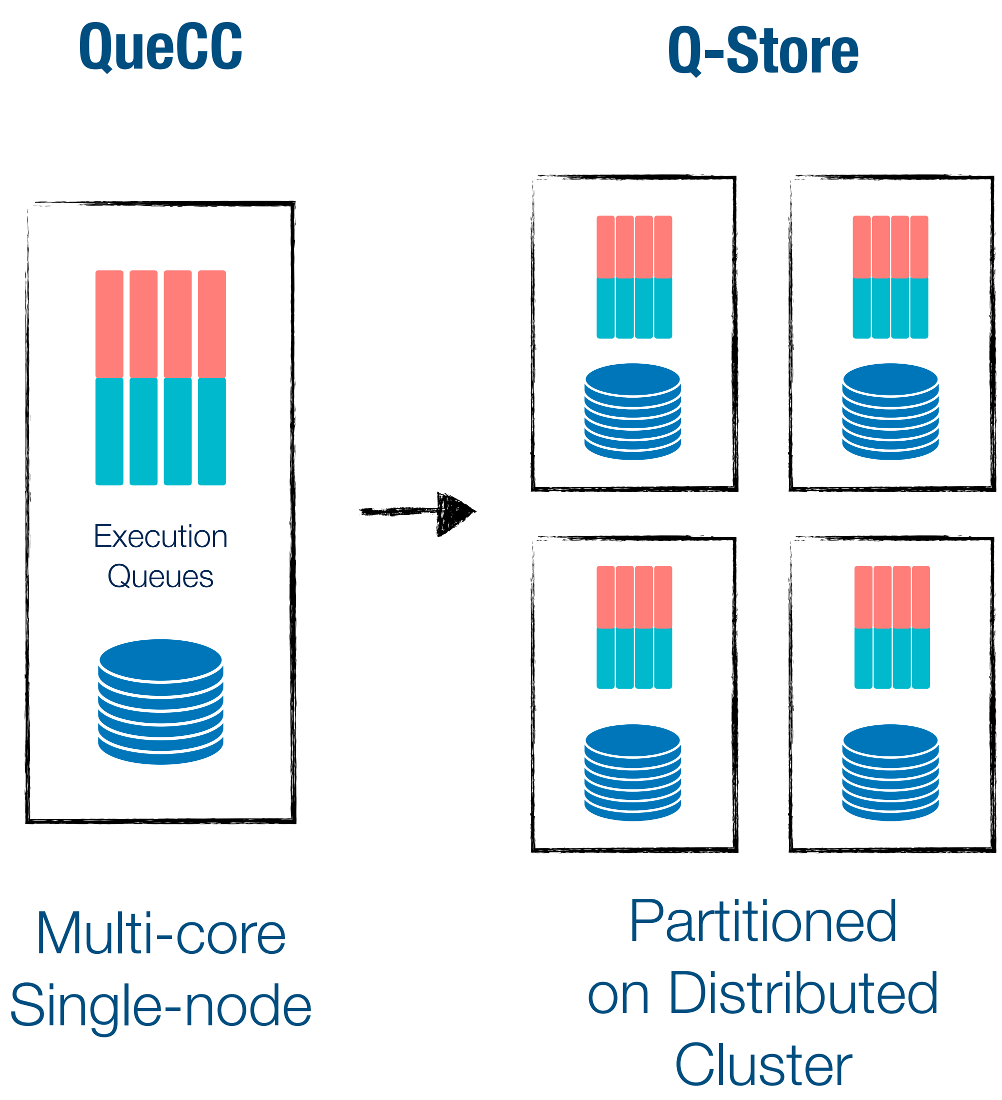
Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020

What's Next: Q-Store



Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020

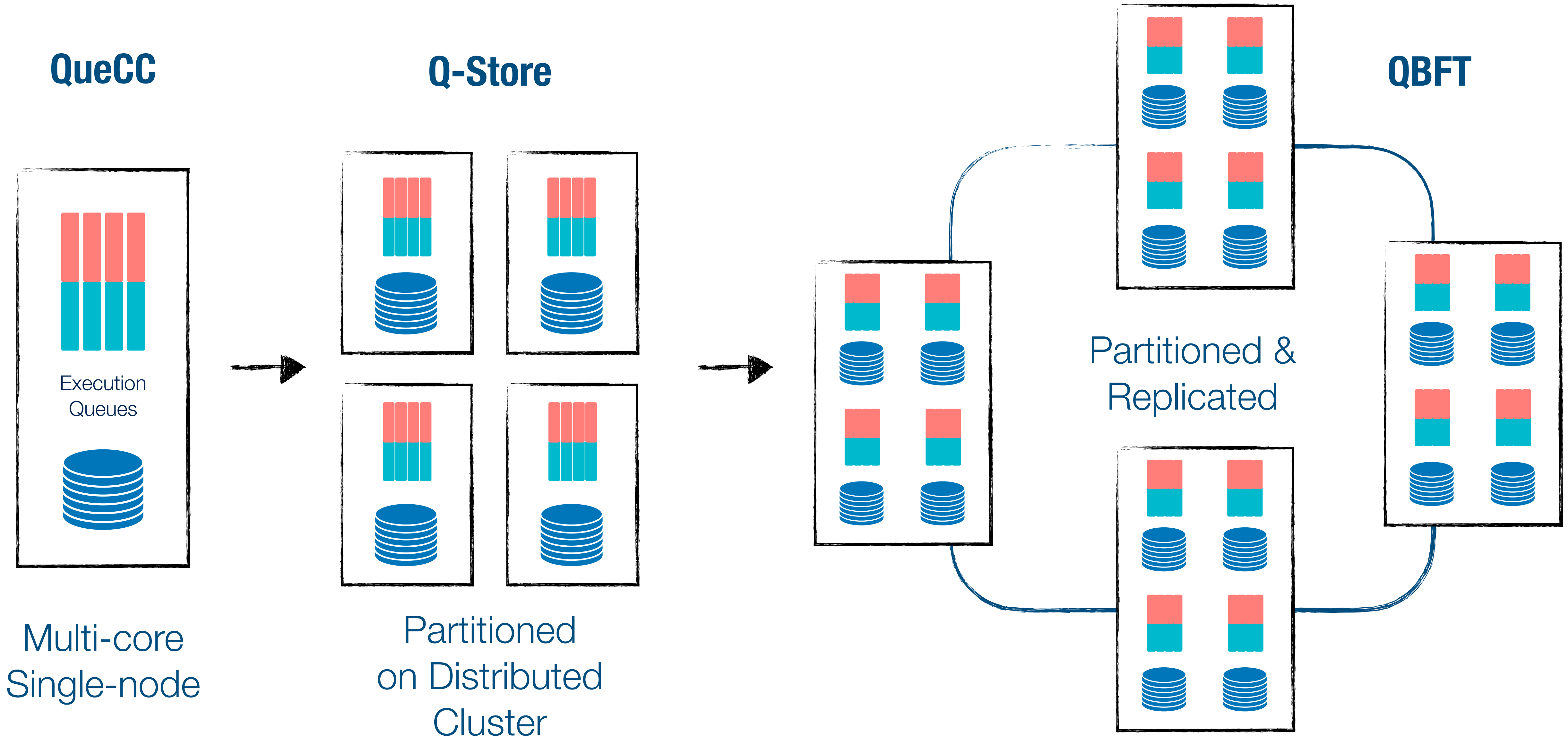
What's Next: Q-Store



- ✓ Parallel and distributed
- ✓ Queue-oriented execution and communication
- ✓ Minimal coordination among nodes and threads

Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication., T. Qadah, S. Gupta, M. Sadoghi, EDBT 2020

What's Next: QBFT



What's Next: QBFT

- ✓ Queue-oriented Byzantine Fault-Tolerance
- ✓ Resilient planning followed by resilient execution