H-Store And VoltDB

One Database In Two Universes

Yiqun (Ethan) Zhang

December 3, 2018

Carnegie Mellon University

yzhang@voltdb.com

https://www.linkedin.com/in/**yzhang1991**

VOLTDB

H-Store

fast transactions

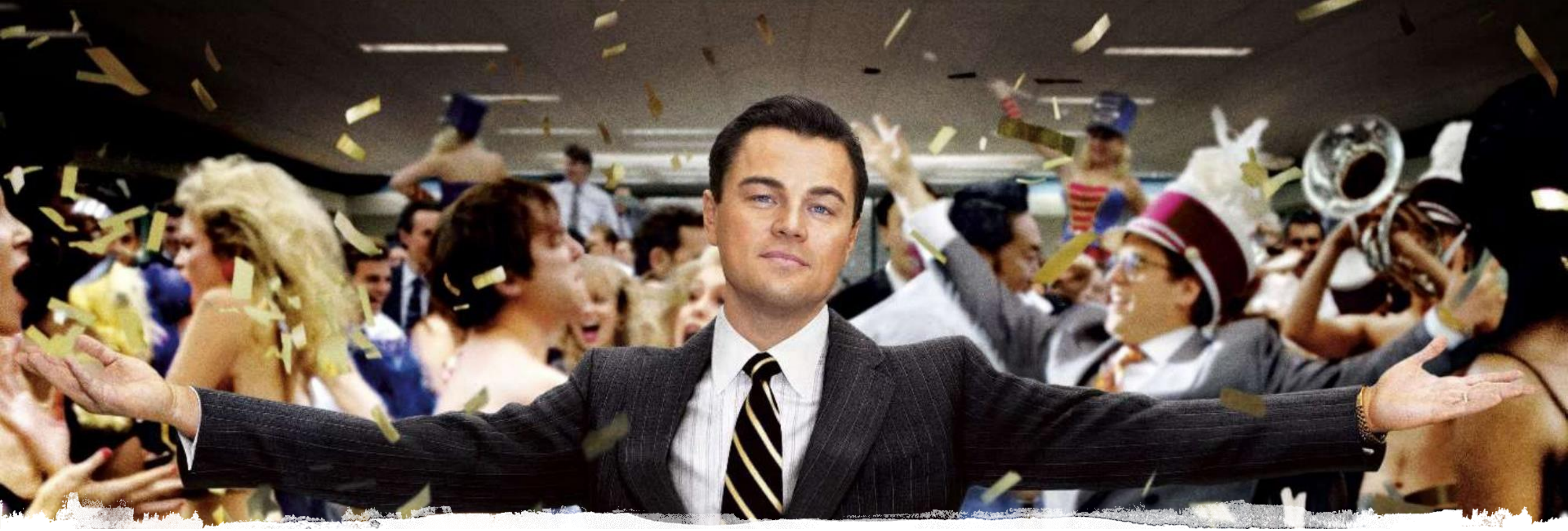
VOLTDB

AGENDA

- History

AGENDA

- Architectural Overview



AGENDA

- How VoltDB diverged from H-Store

AGENDA

- New research followed H-Store

read-only

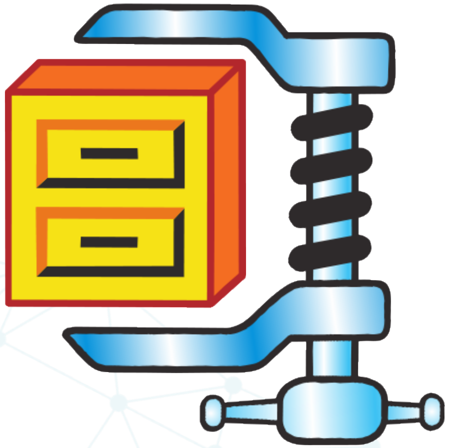
long-running

OLAP

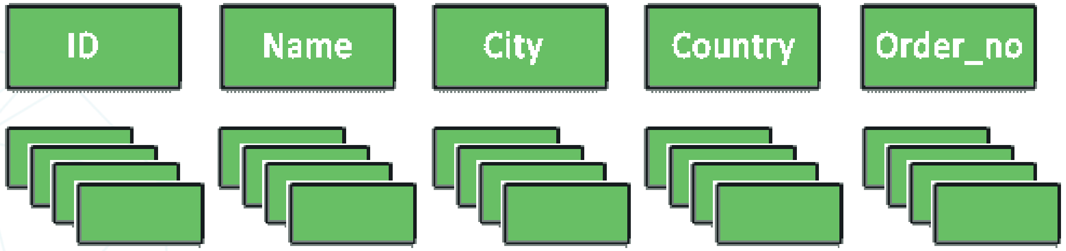
complex joins

exploratory queries

compression

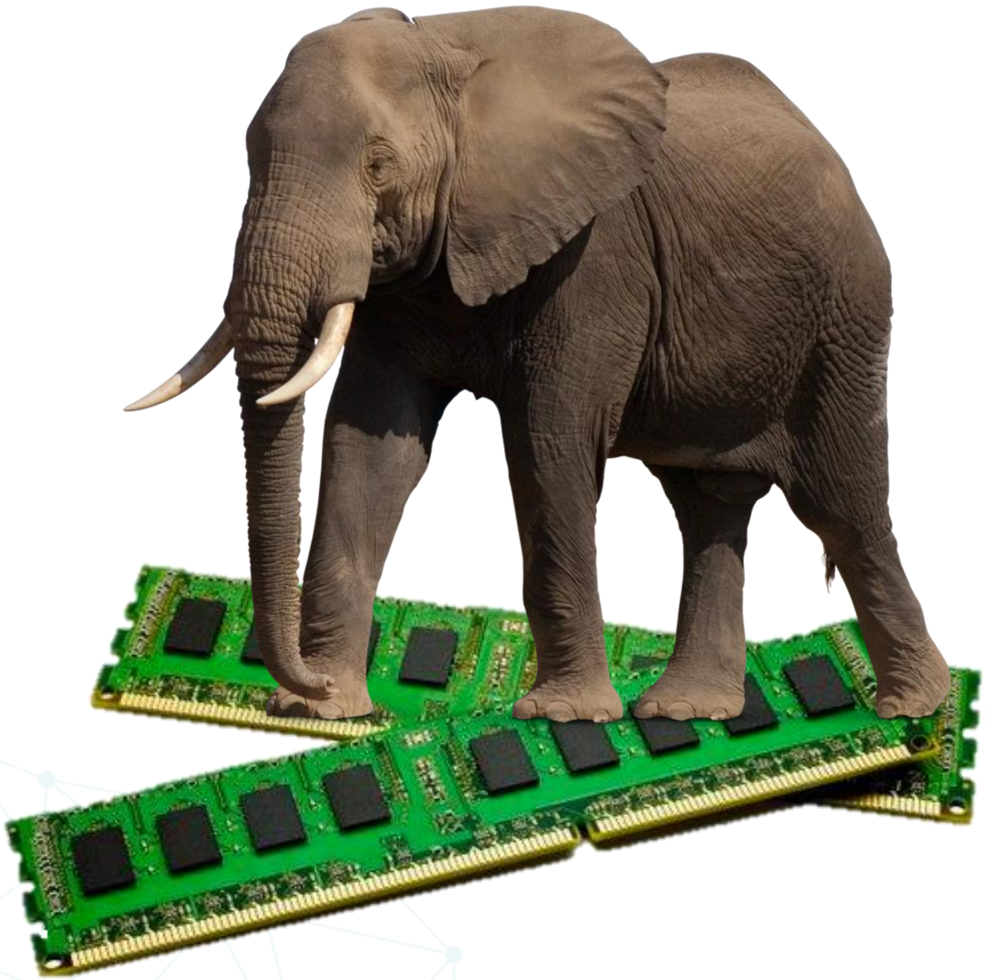


column-store



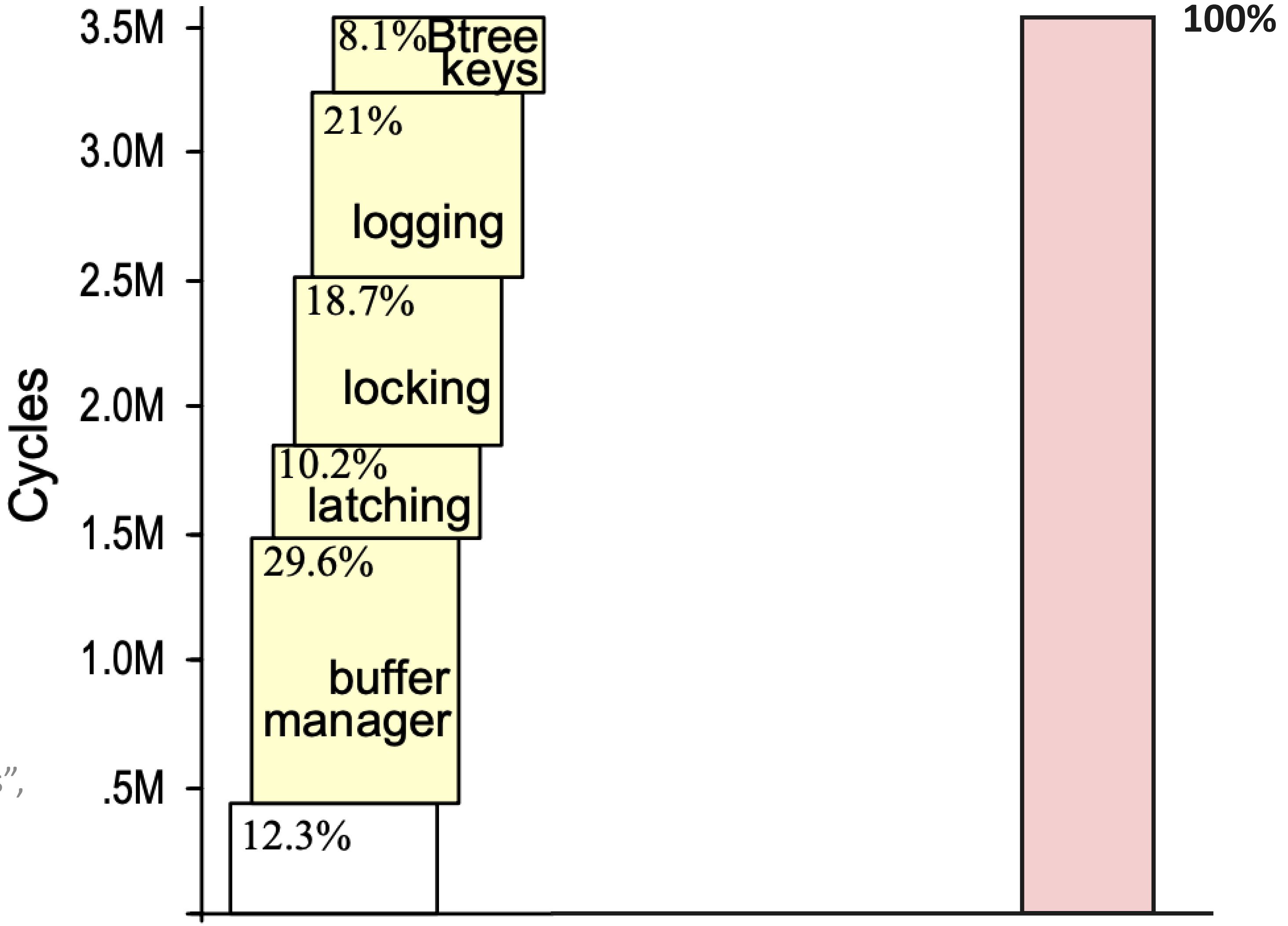
What if state fits in memory?

- A lot of data sets do fit in memory
- 100 MB per warehouse in TPC-C
- Even data for 1,000 such warehouses can still fit!



VOLTDB

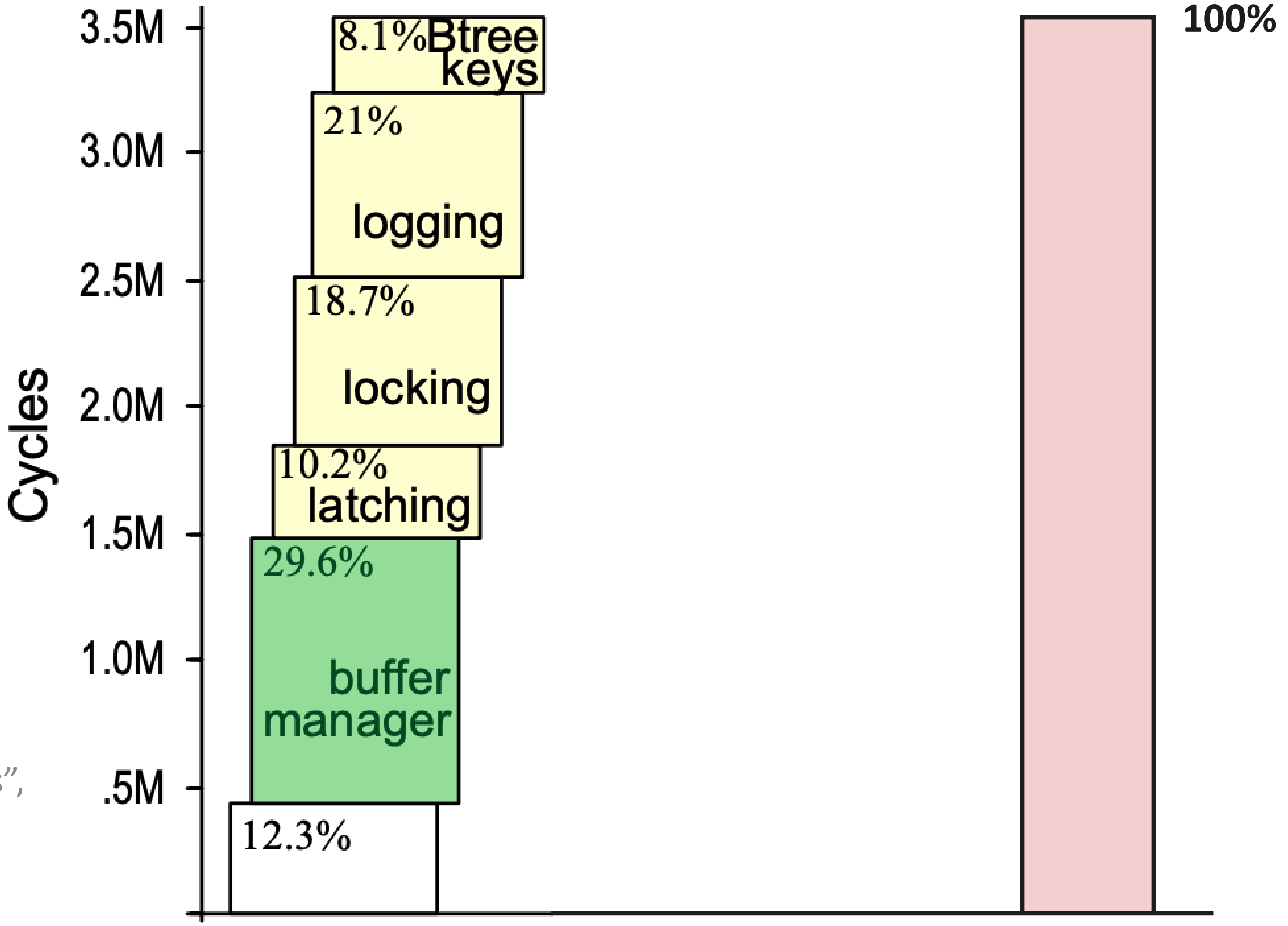
Where did we spend our time?



CPU Cycle Breakdown for Shore on TPC-C New Order

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

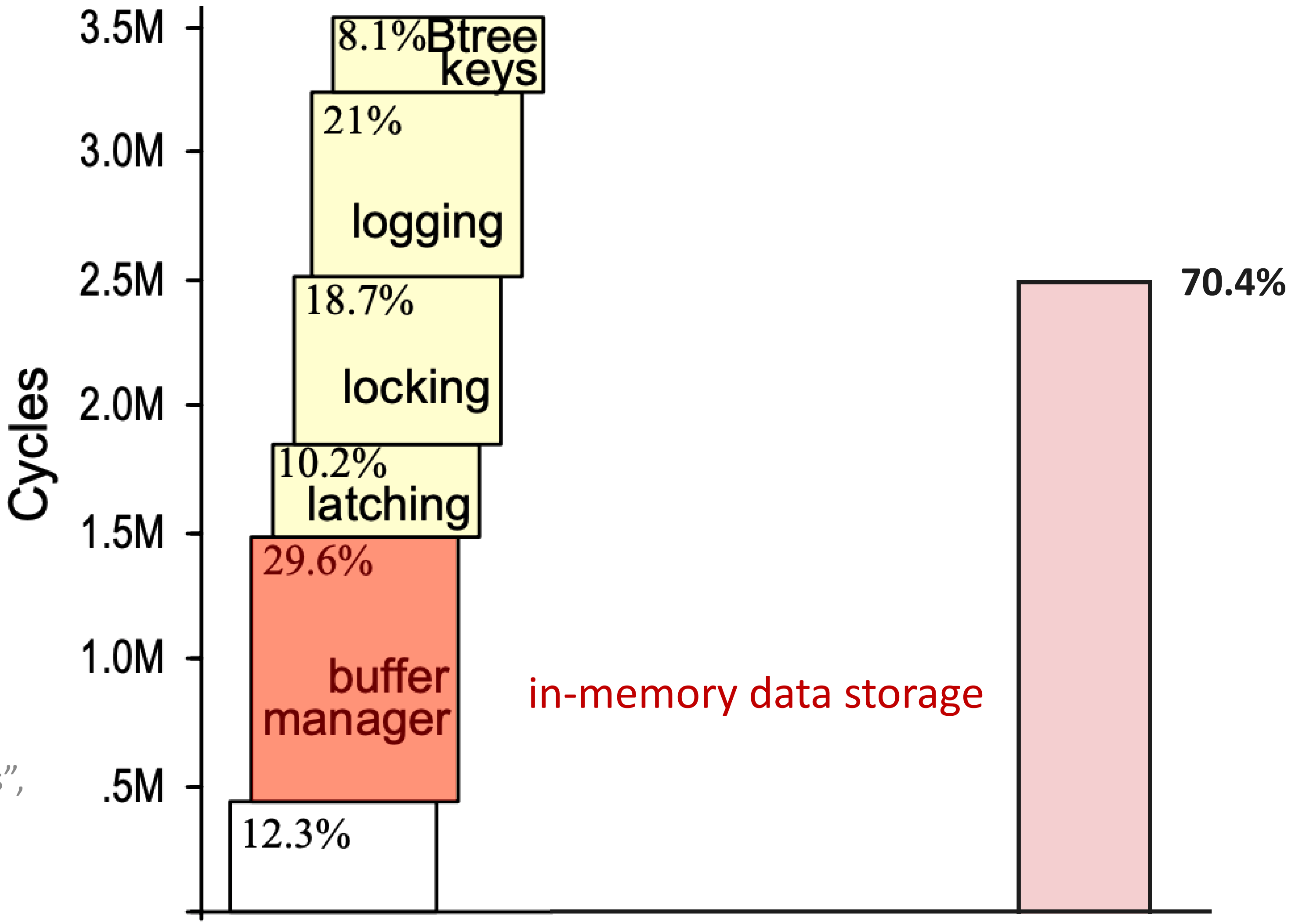
Where did we spend our time?



CPU Cycle Breakdown for Shore on TPC-C New Order

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

Where did we spend our time?

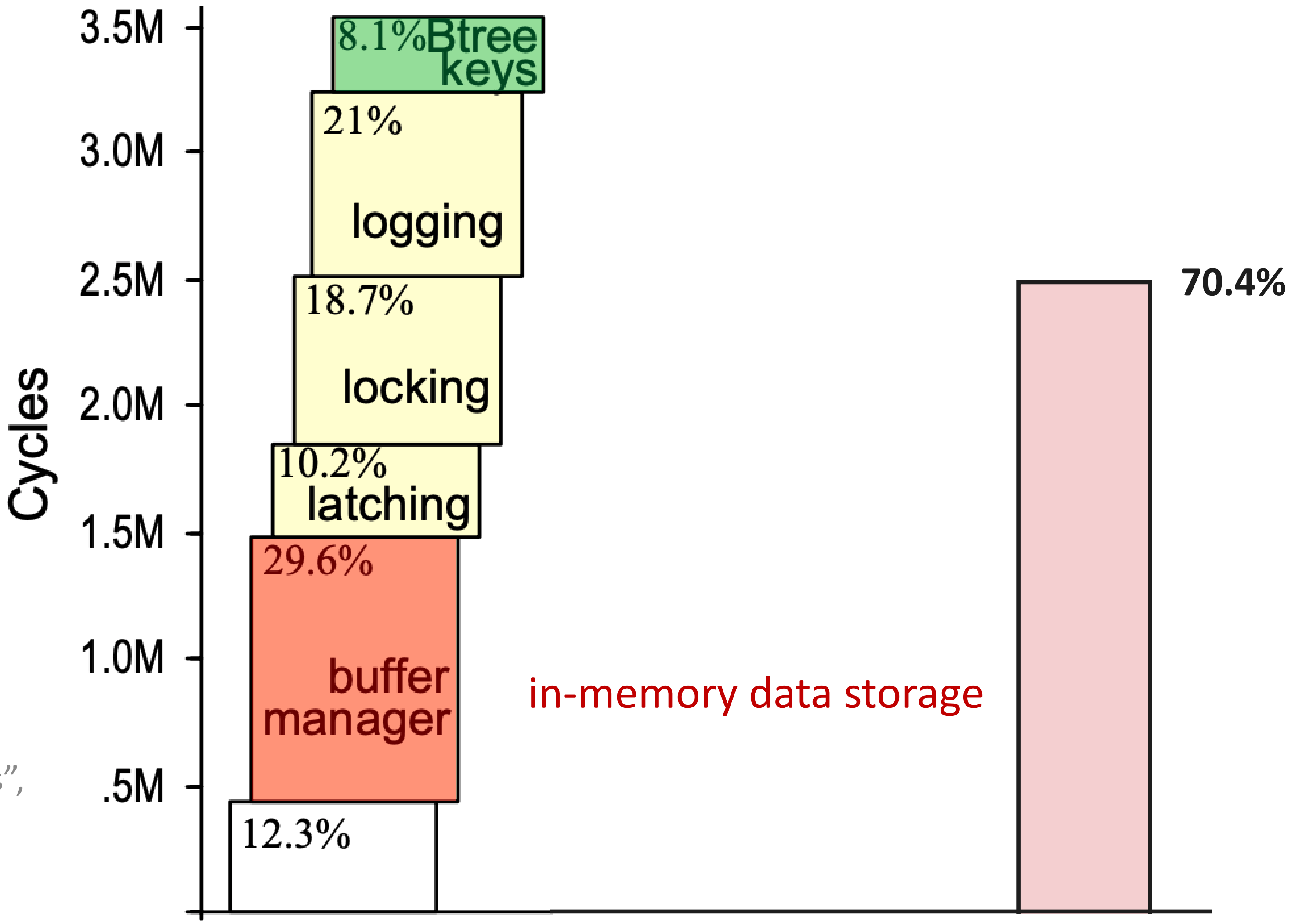


CPU Cycle Breakdown for Shore on TPC-C New Order

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

VOLTDB

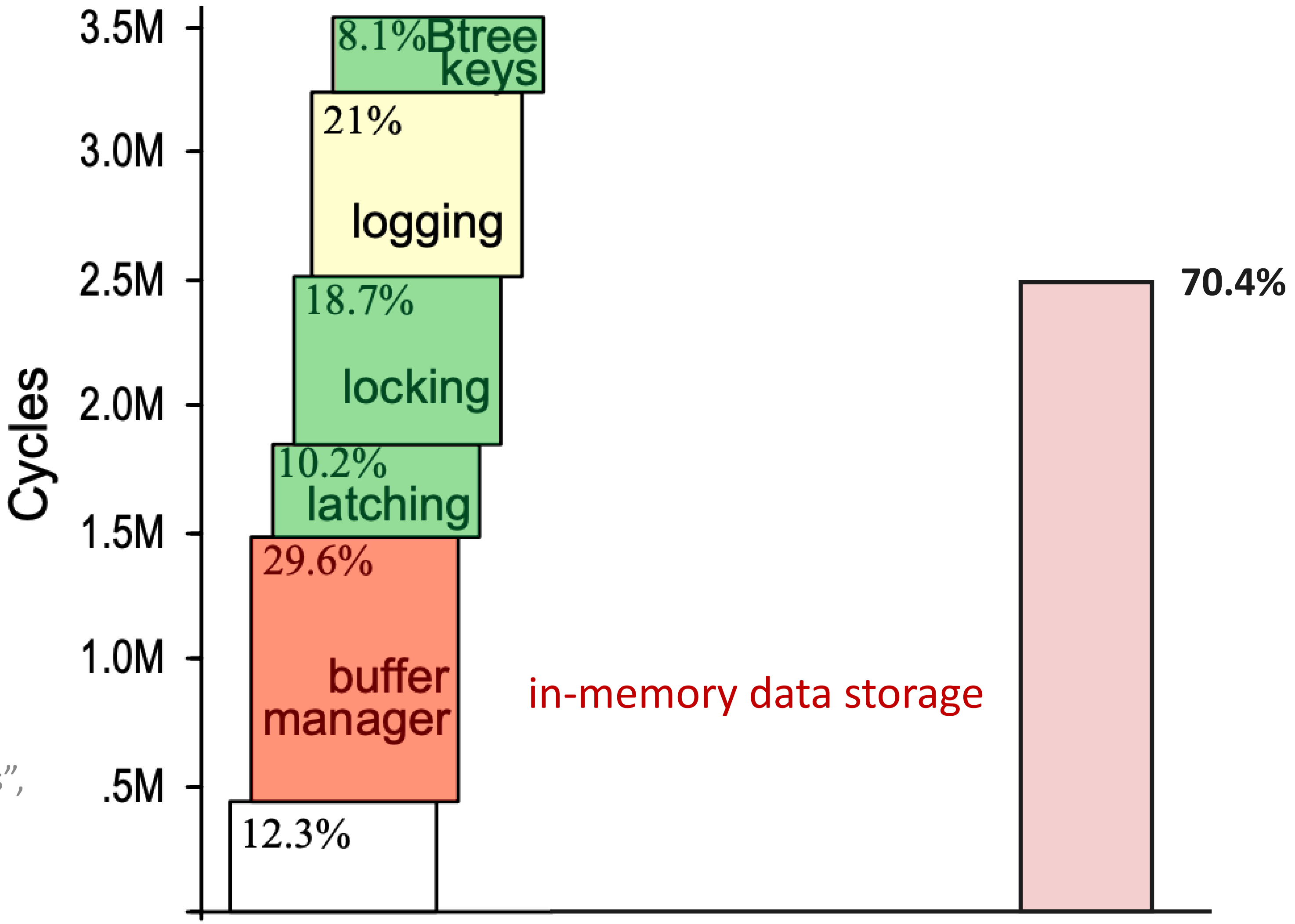
Where did we spend our time?



CPU Cycle Breakdown for Shore on TPC-C New Order

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

Where did we spend our time?



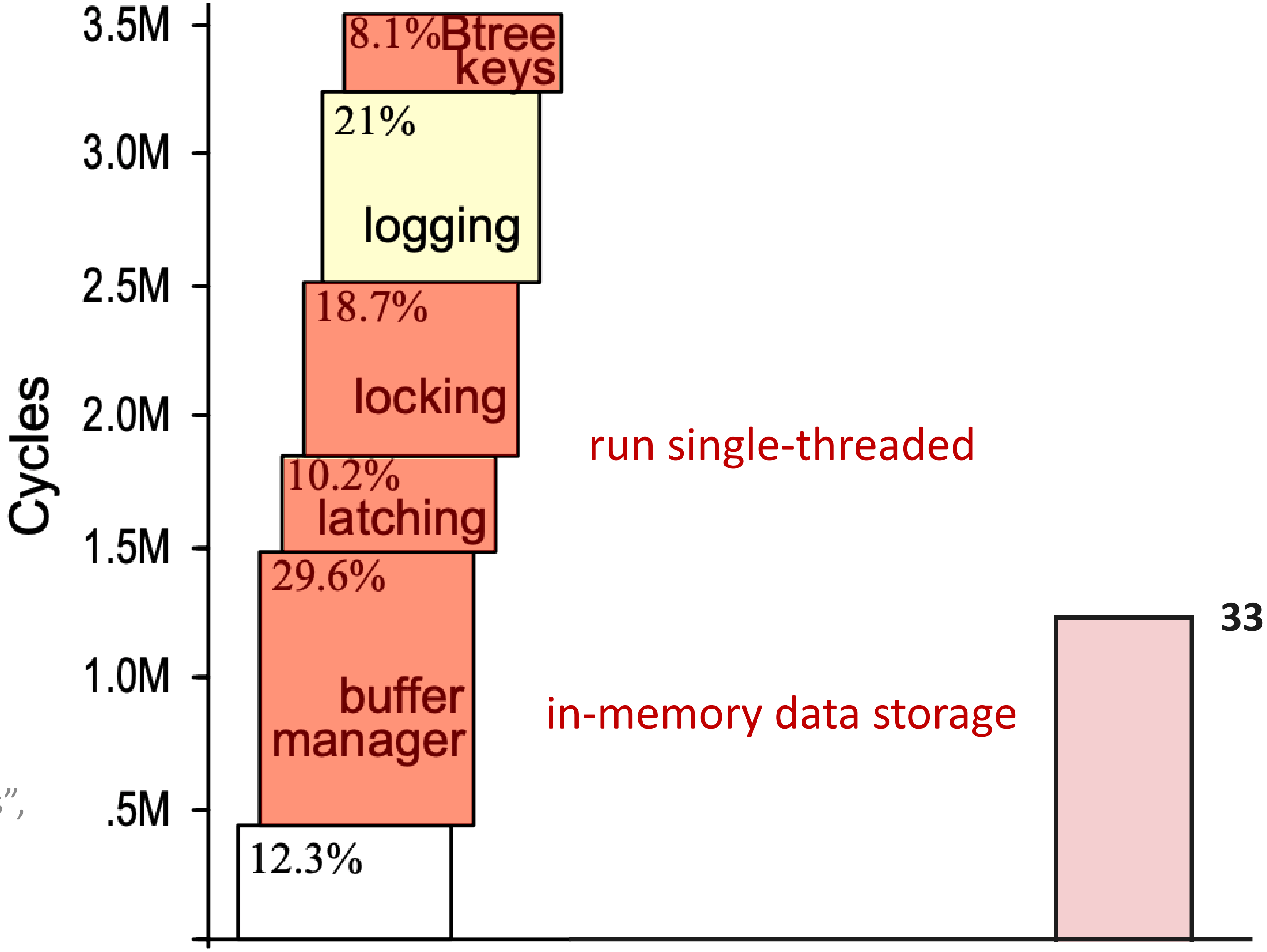
CPU Cycle Breakdown for Shore on TPC-C New Order

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

OLTP transactions are short-lived

- The heaviest TPC-C transaction:
  - reads/writes ~200 records;
  - can be finished in less than 1 millisecond;
  - CPU is not the bottleneck.

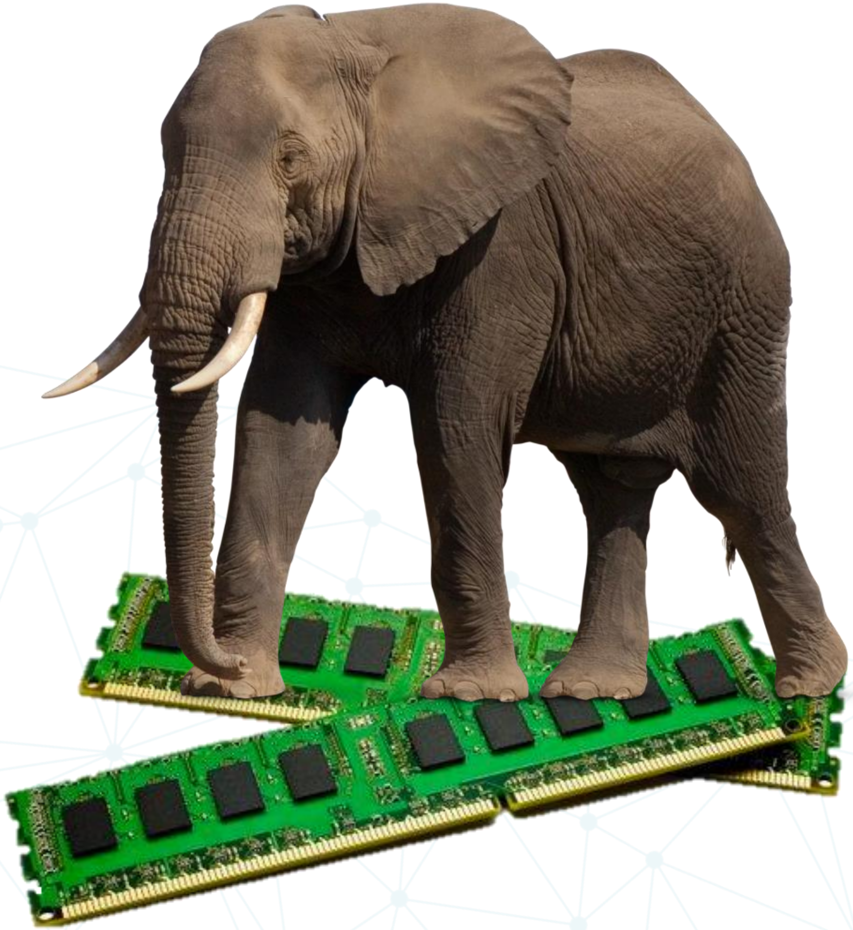
Where did we spend our time?



CPU Cycle Breakdown for Shore on TPC-C New Order

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

Single-threaded problems



- Waiting on users leaves CPU idle.
- Single-threaded does not jive with the multicore world.

VOLTDB

Transactions are repetitive

- Queries are known in advance;
- Control flows are settled in advance too.
- External transaction control can be converted into pre-compiled stored procedures with structured control code intermixed with parameterized SQL commands on the server.

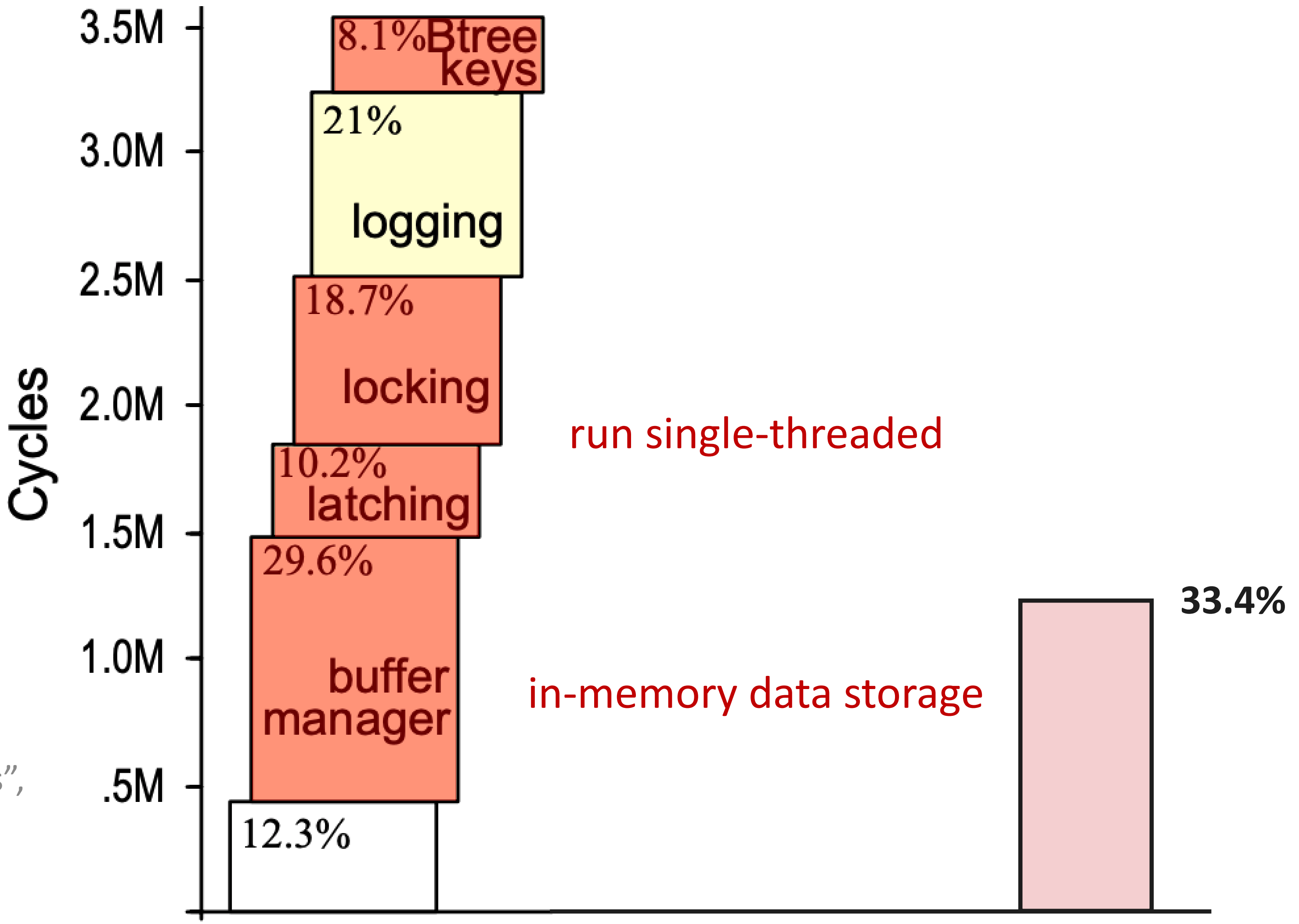
Waiting on users external transaction control

- Don't
- External transaction control and performance are not friends;
- Use server-side transactional logic;
- Move the logic to data, not the other way around;

Using ALL the cores

- Partitioning data is a requirement for scale-out.
- Single-threaded is desired for efficiency.
  Why not partition to the core instead of the node?
- Concurrency via scheduling, not shared memory.

Where did we spend our time?

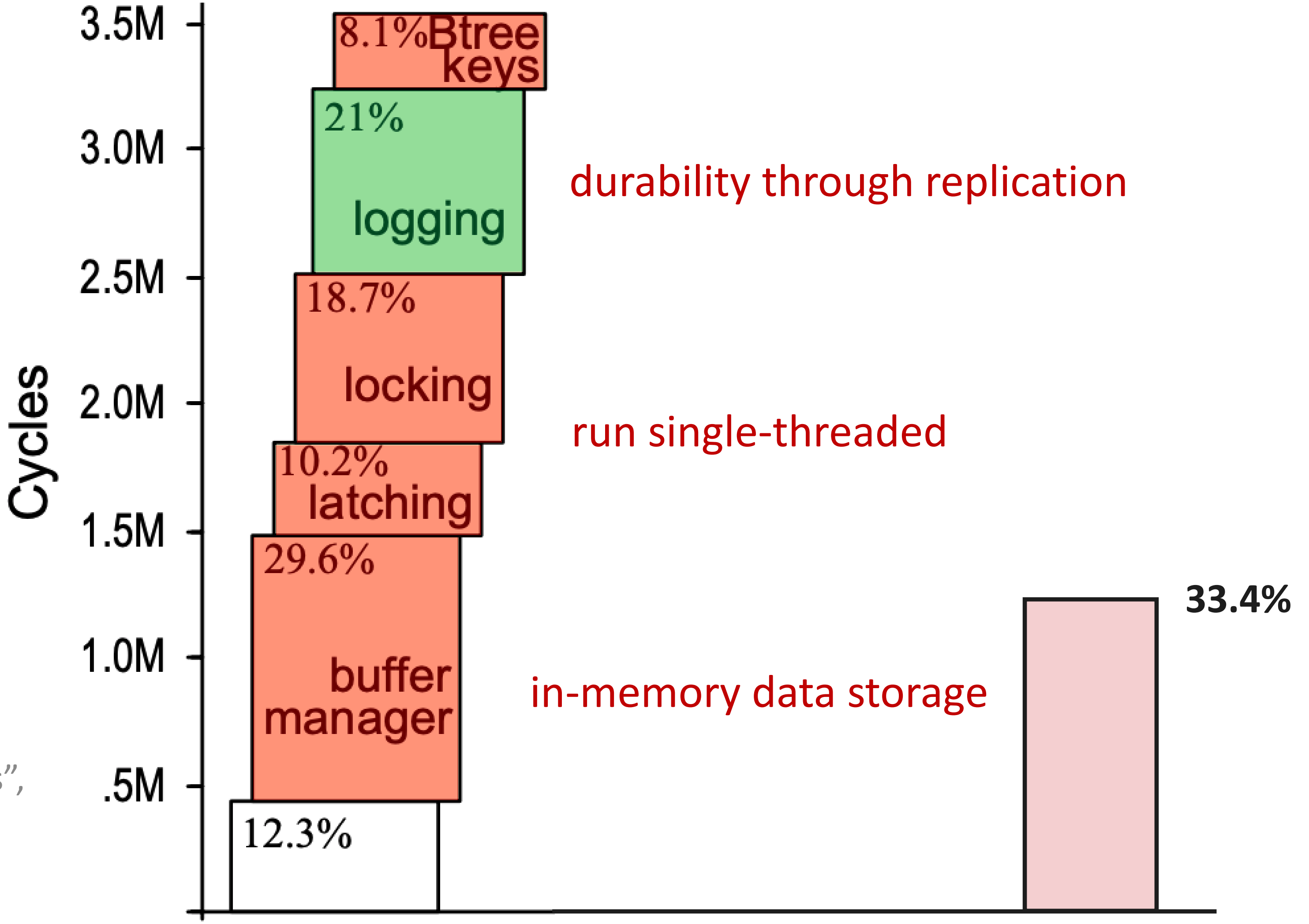


CPU Cycle Breakdown for Shore on TPC-C New Order

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

VOLTDB

Where did we spend our time?

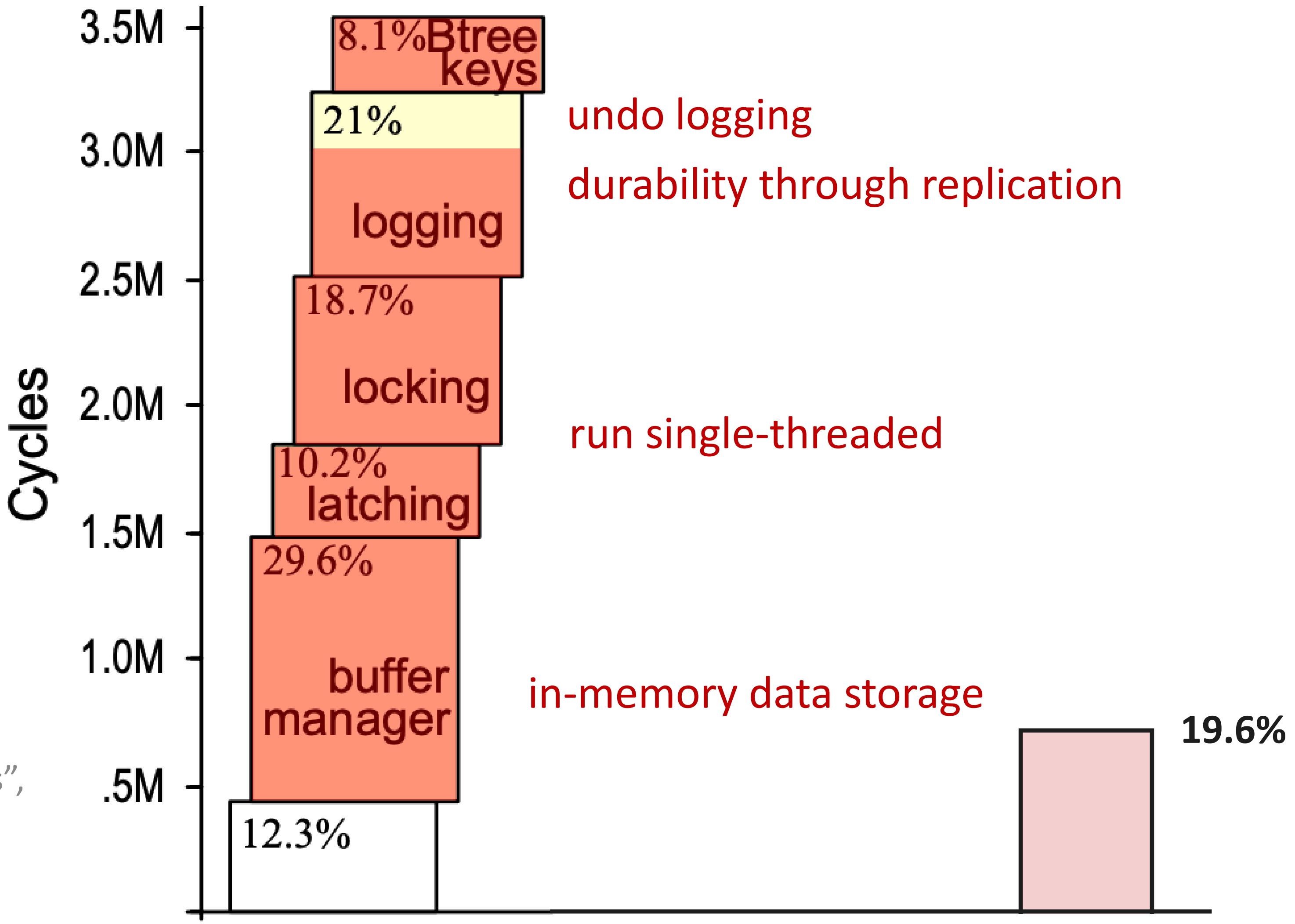


CPU Cycle Breakdown for Shore on TPC-C New Order

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

VOLTDB

Where did we spend our time?



CPU Cycle Breakdown for Shore on TPC-C New Order

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

VOLTDB

What did we end up building?

- In-memory relational SQL database.
- No external transaction control – Stored Procedures
- Single-threaded engines run in parallel.
- Partitioned to the core.
- Concurrency via Scheduling, not shared memory.
- Serializable ACID.
- Durability through Replication

Architecture

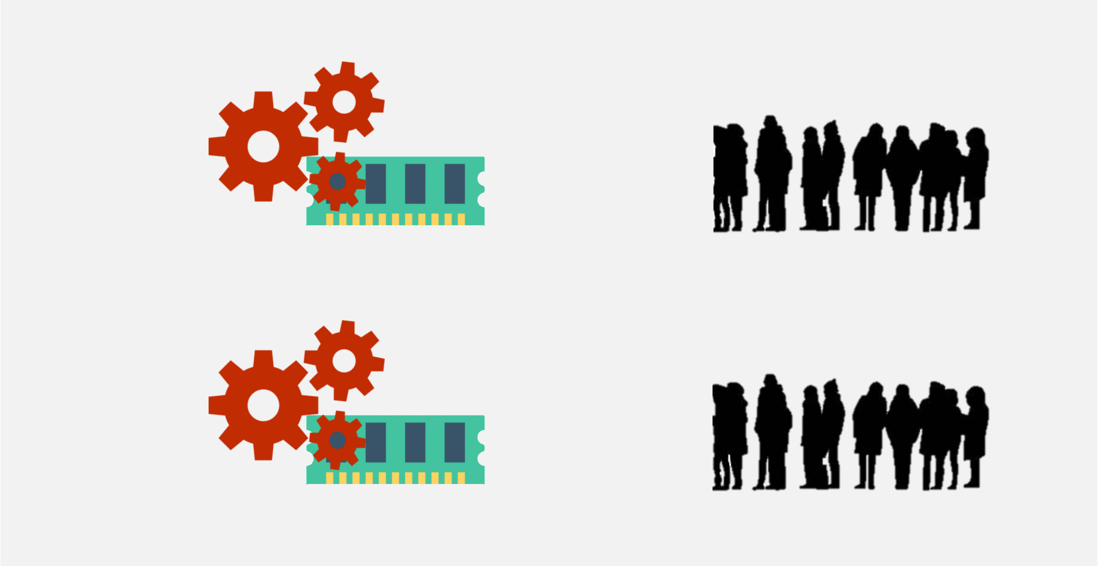

WORLD OF WARCRAFT
MISTS of PANDARIA
©2012 Blizzard Entertainment, Inc. All rights reserved.

Run in parallel

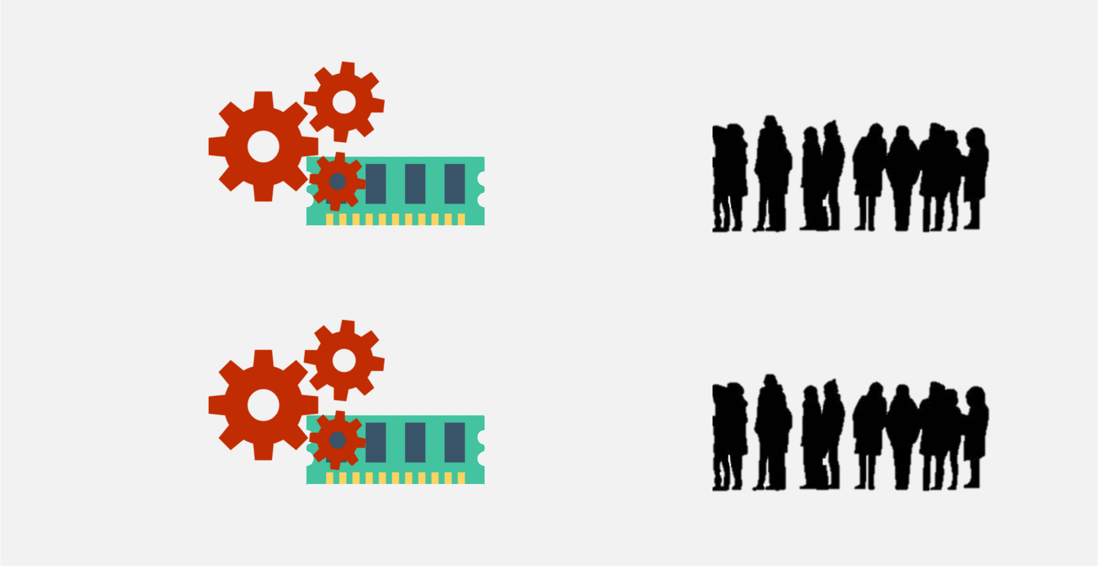
In-memory store

Single-threaded engine

Node #1

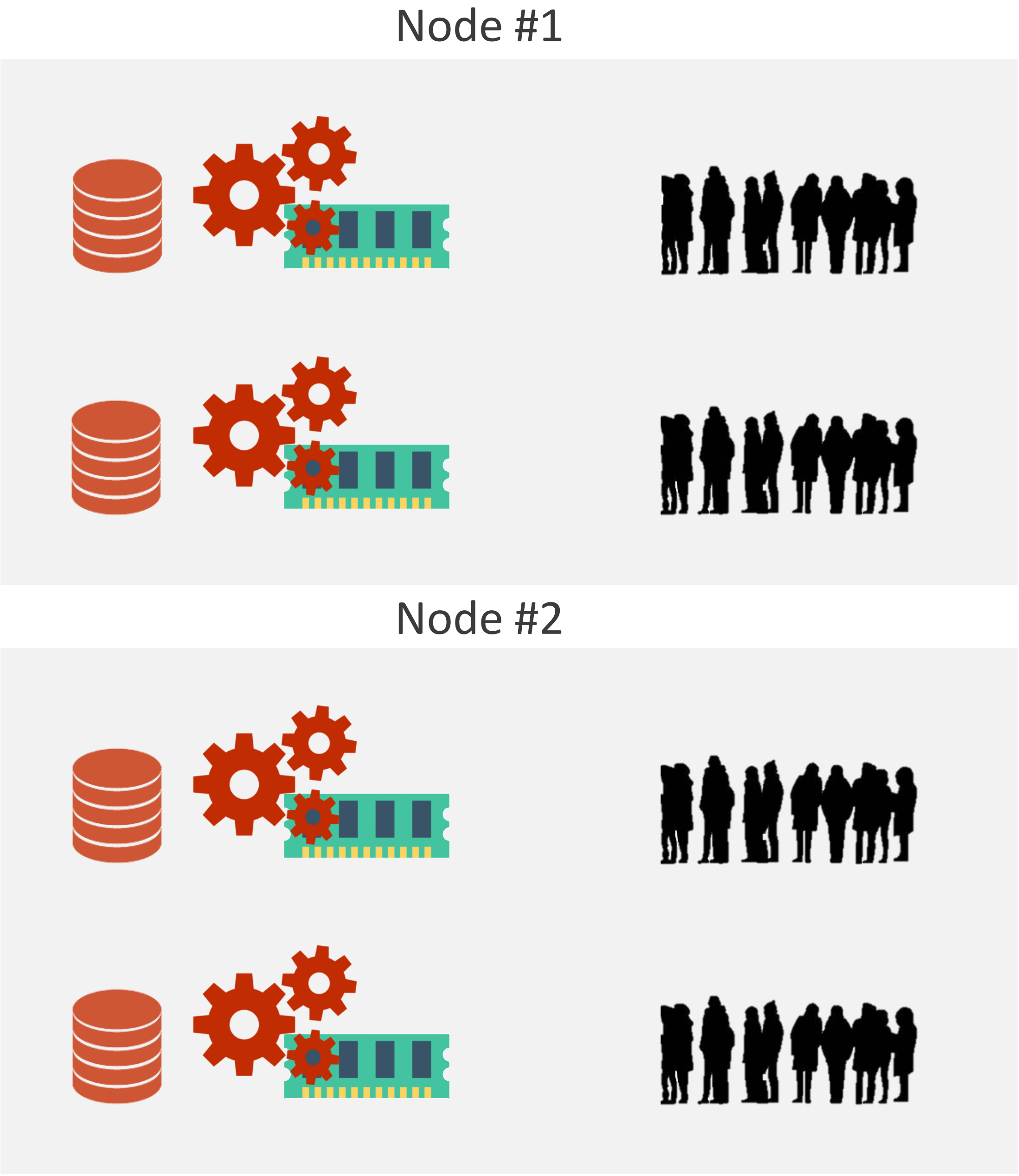


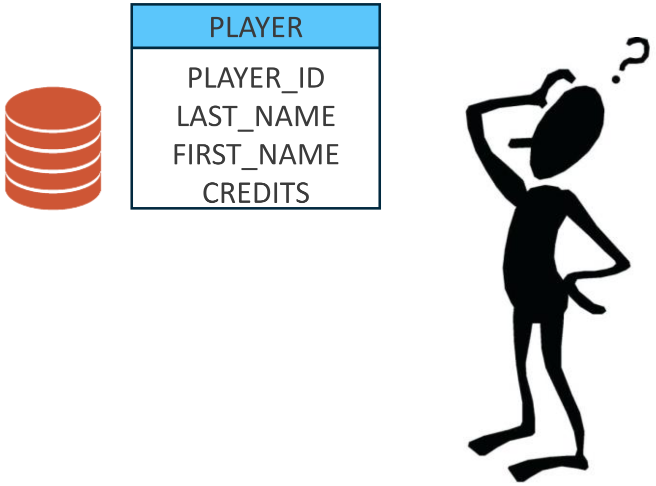
Node #2



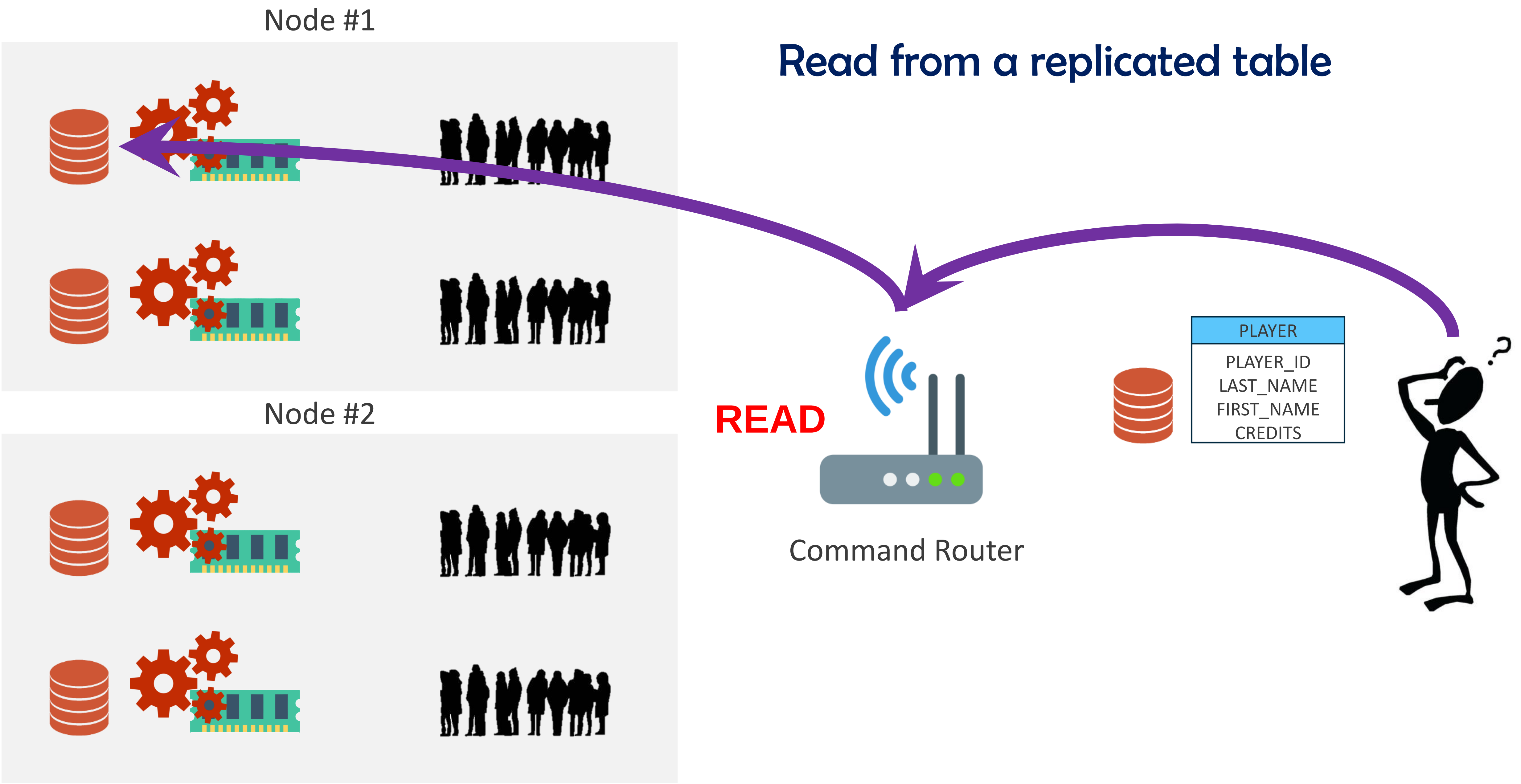
PLAYER
PLAYER_ID LAST_NAME FIRST_NAME CREDITS

Replicated table

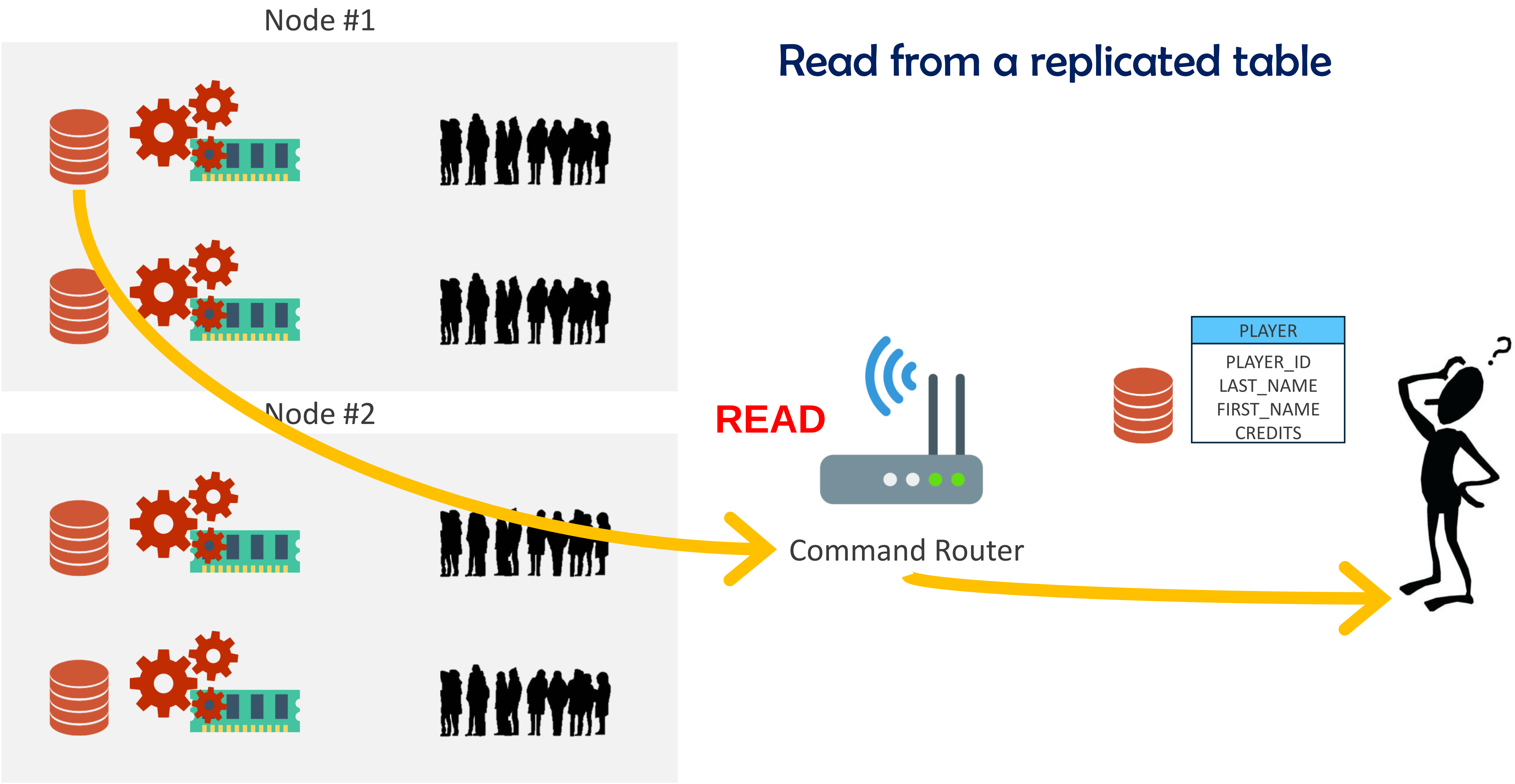




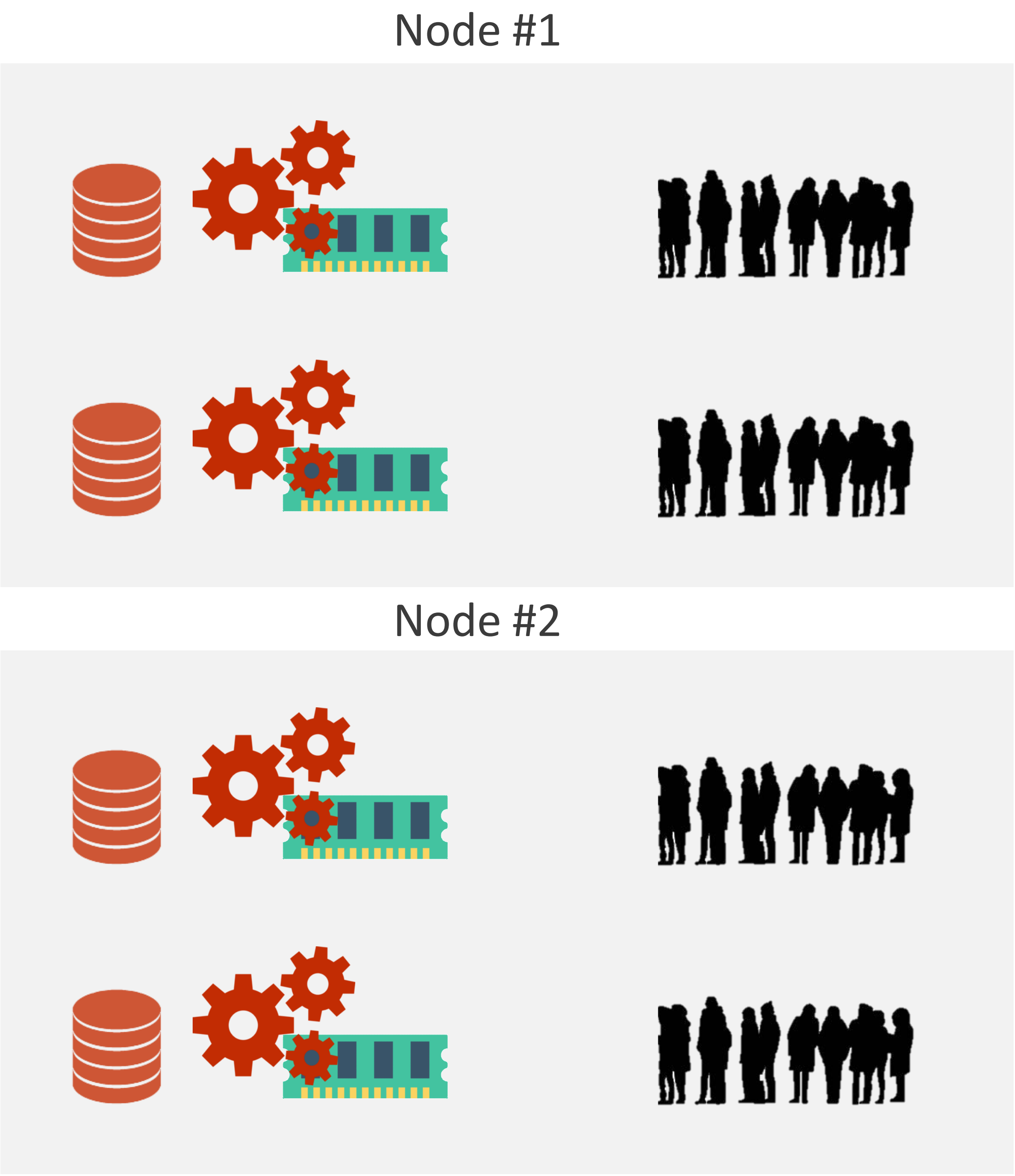
Read from a replicated table

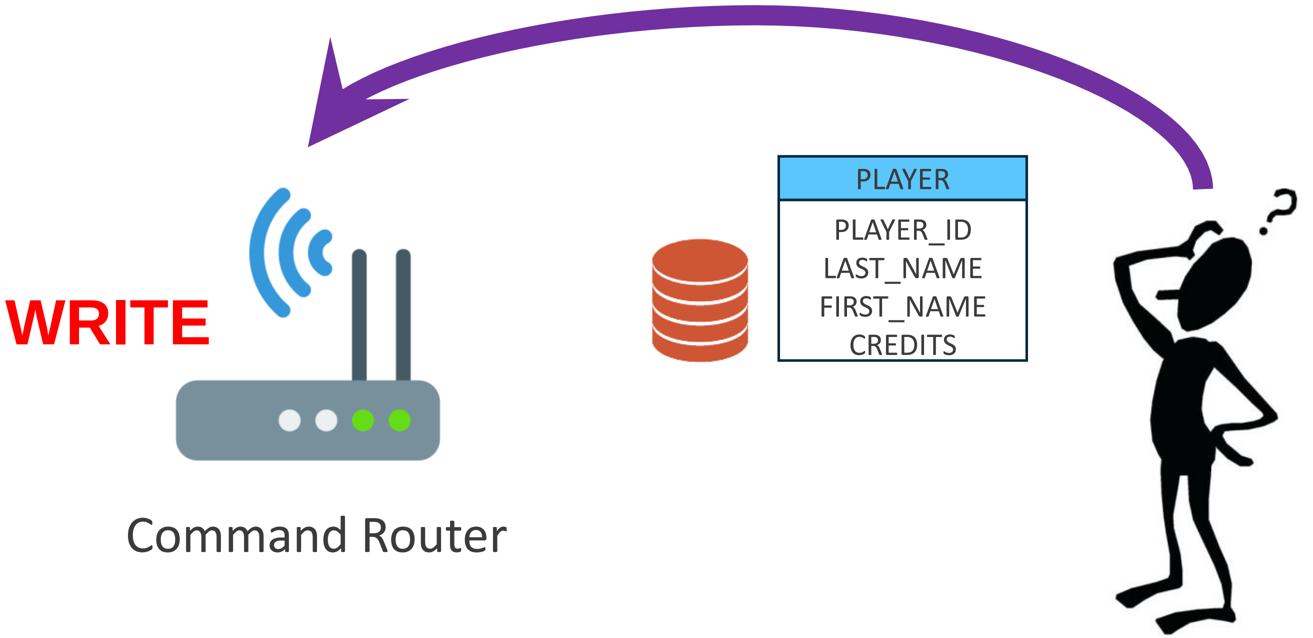


Read from a replicated table

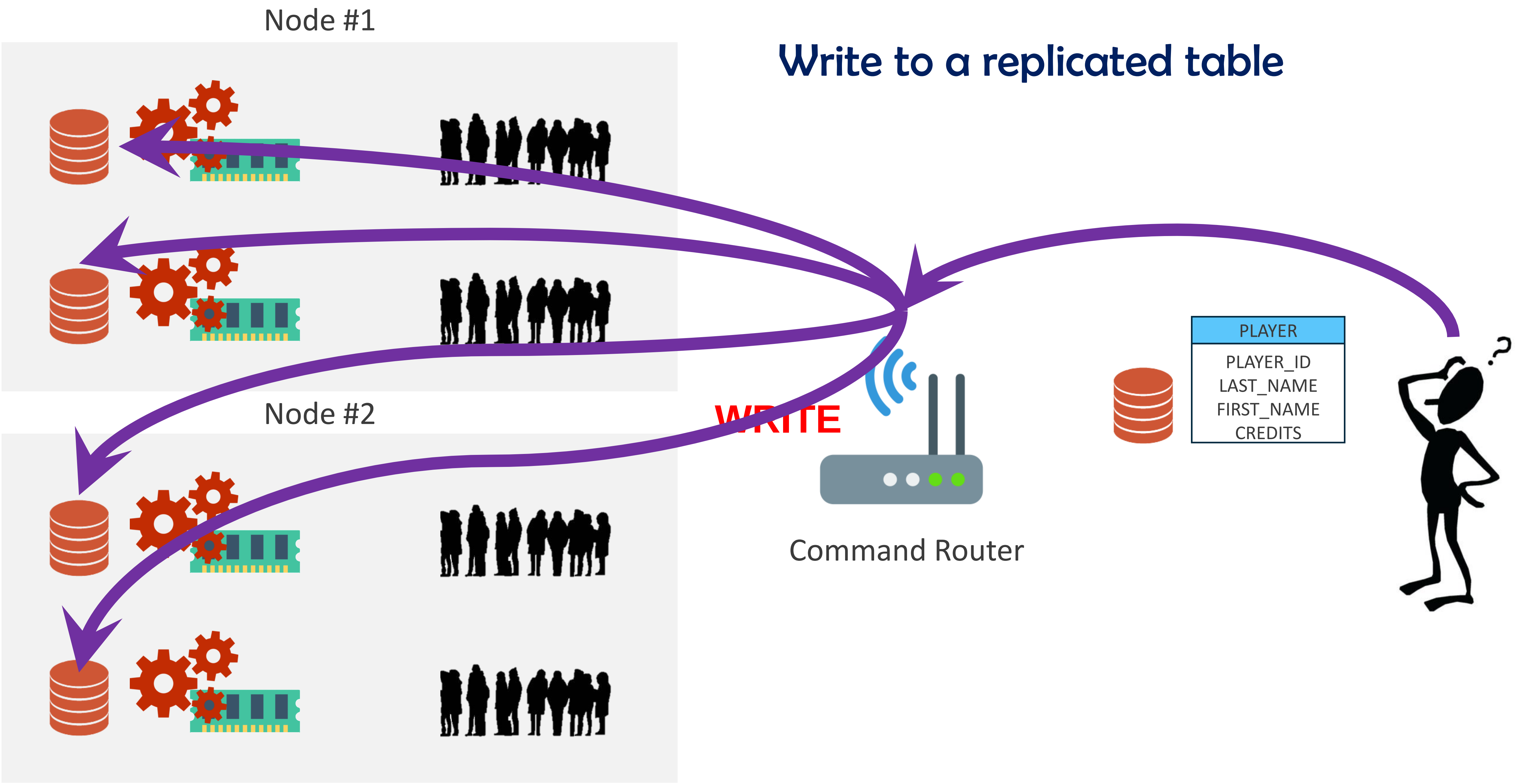


Write to a replicated table

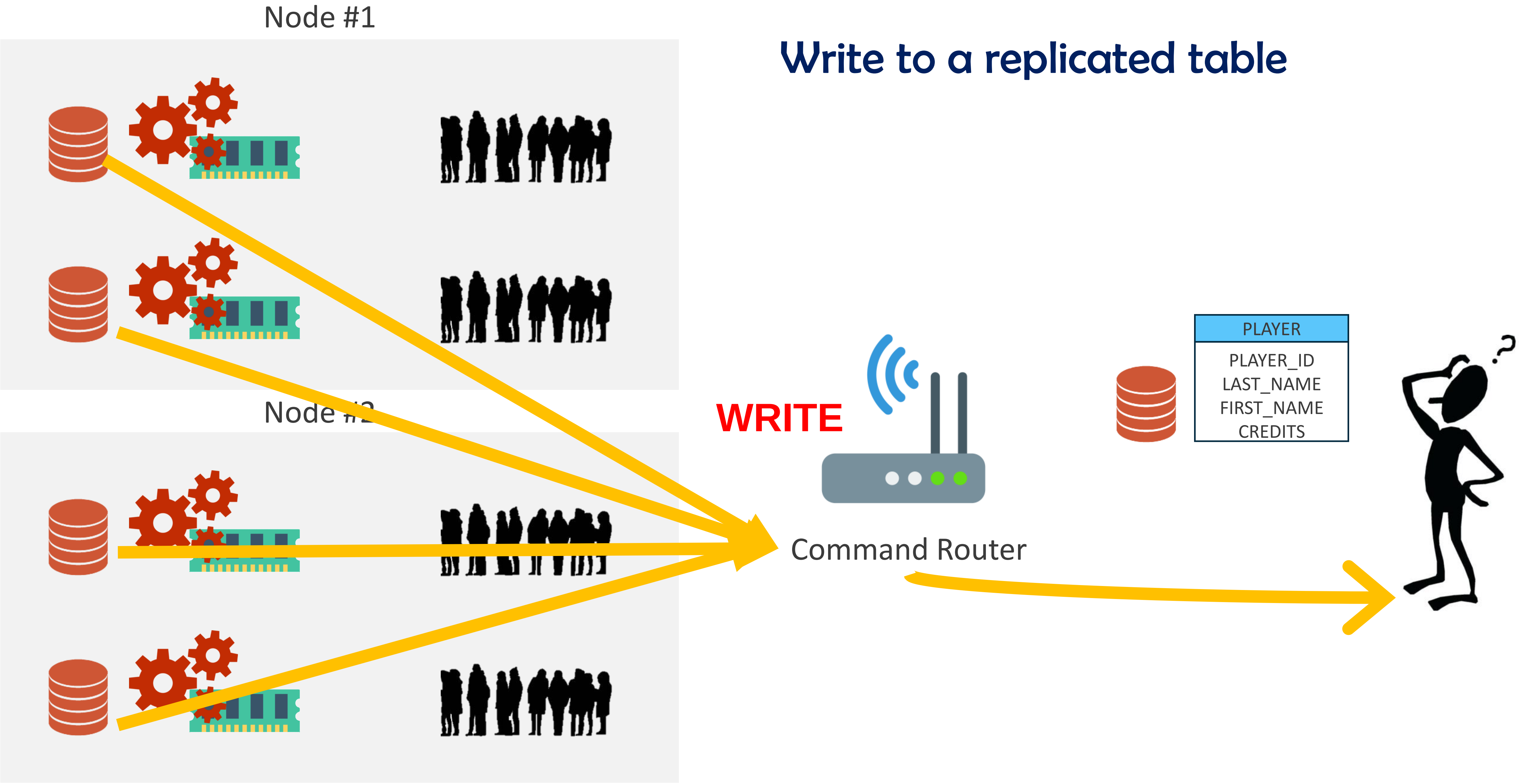




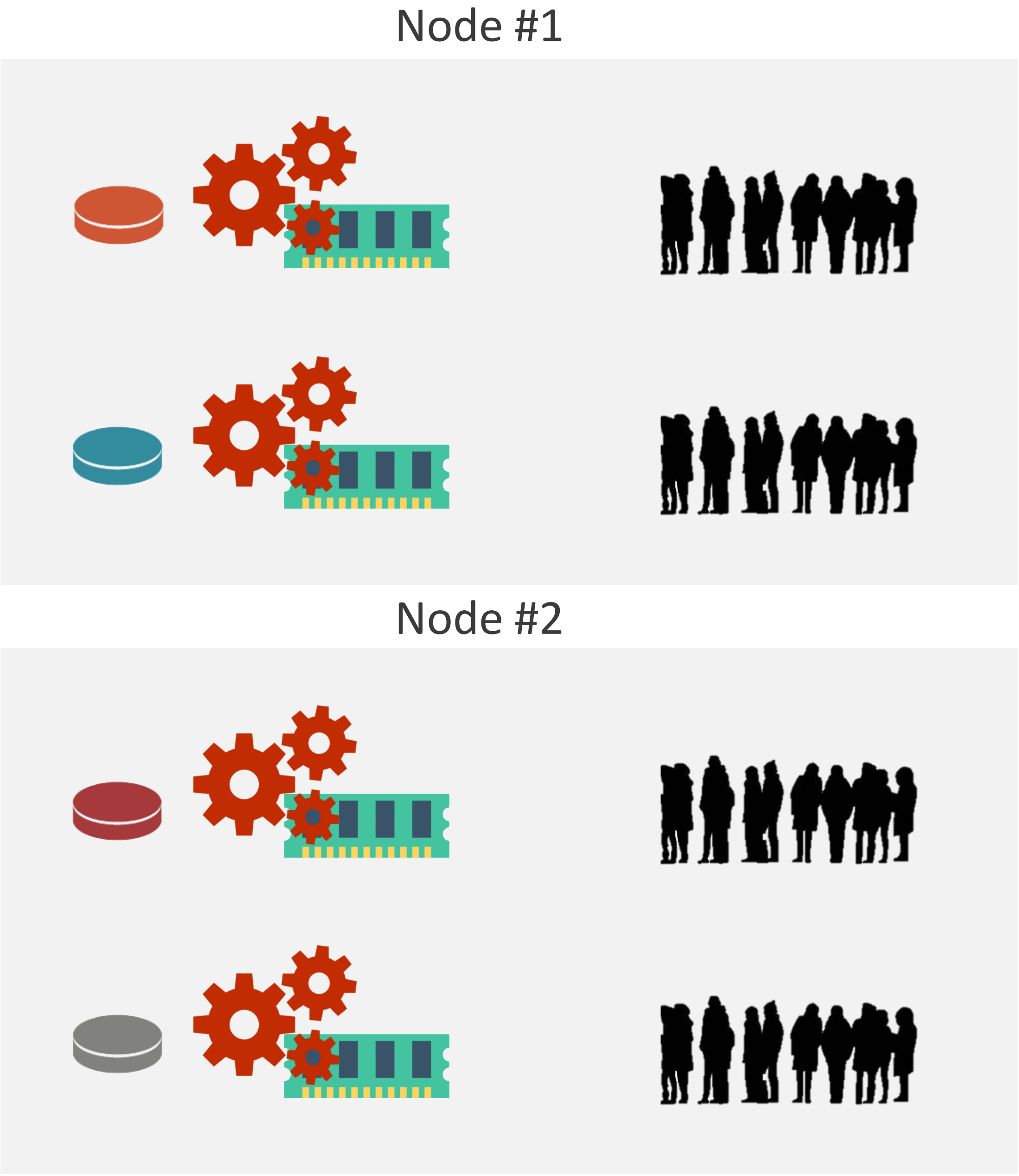
Write to a replicated table



Write to a replicated table



Partitioned Table



PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

VOLTDB

Single Partition Read

select * from PLAYER where PLAYER_ID = 687

Node #1

123, Brown, Joe, 100

456, Silvers, Phil, 77

234, Green, Peter, 41

567, Brown, Mary, 68

Node #2

345, White, Betty, 94

687, Black, Mark, 55

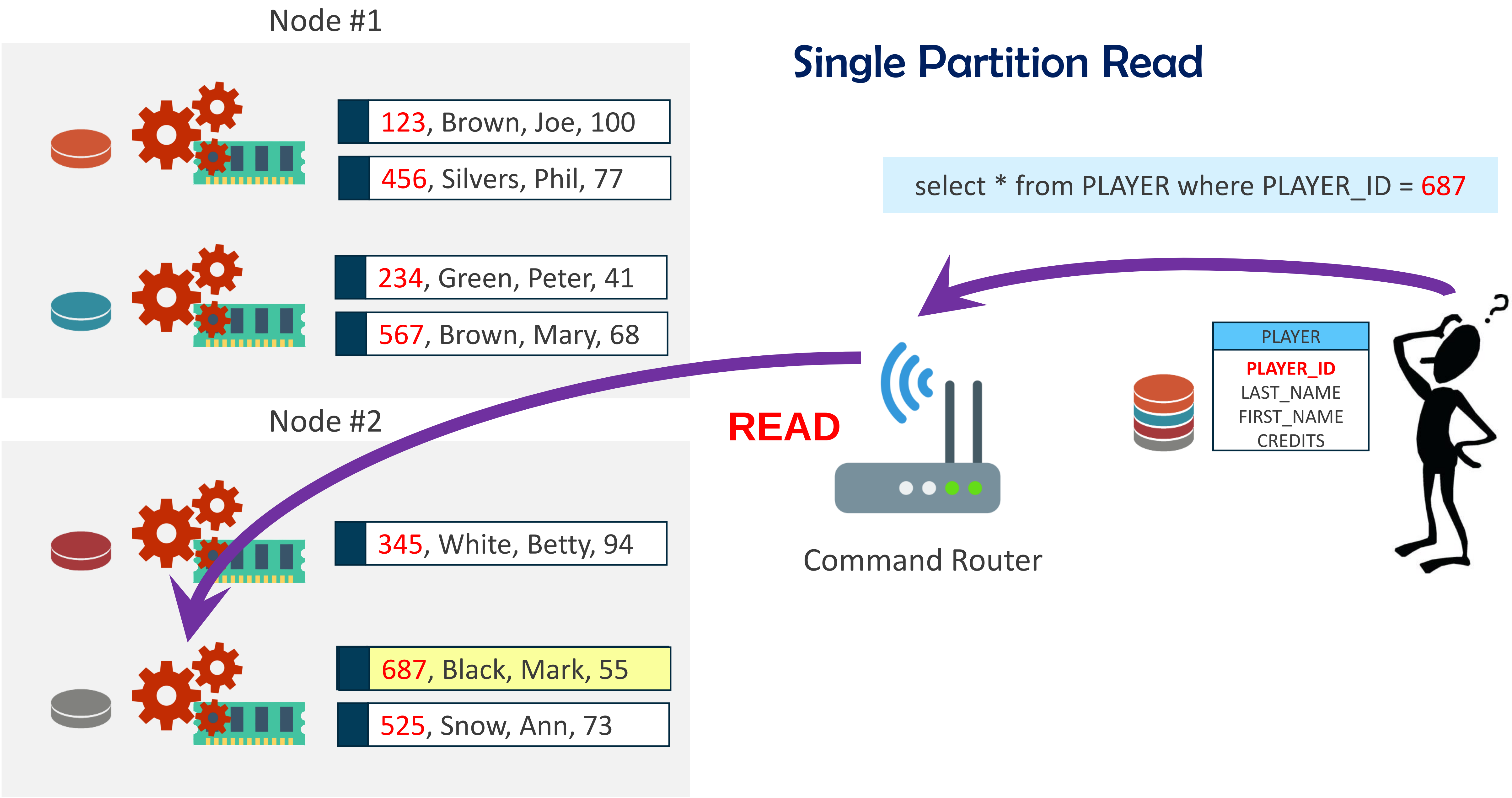
525, Snow, Ann, 73

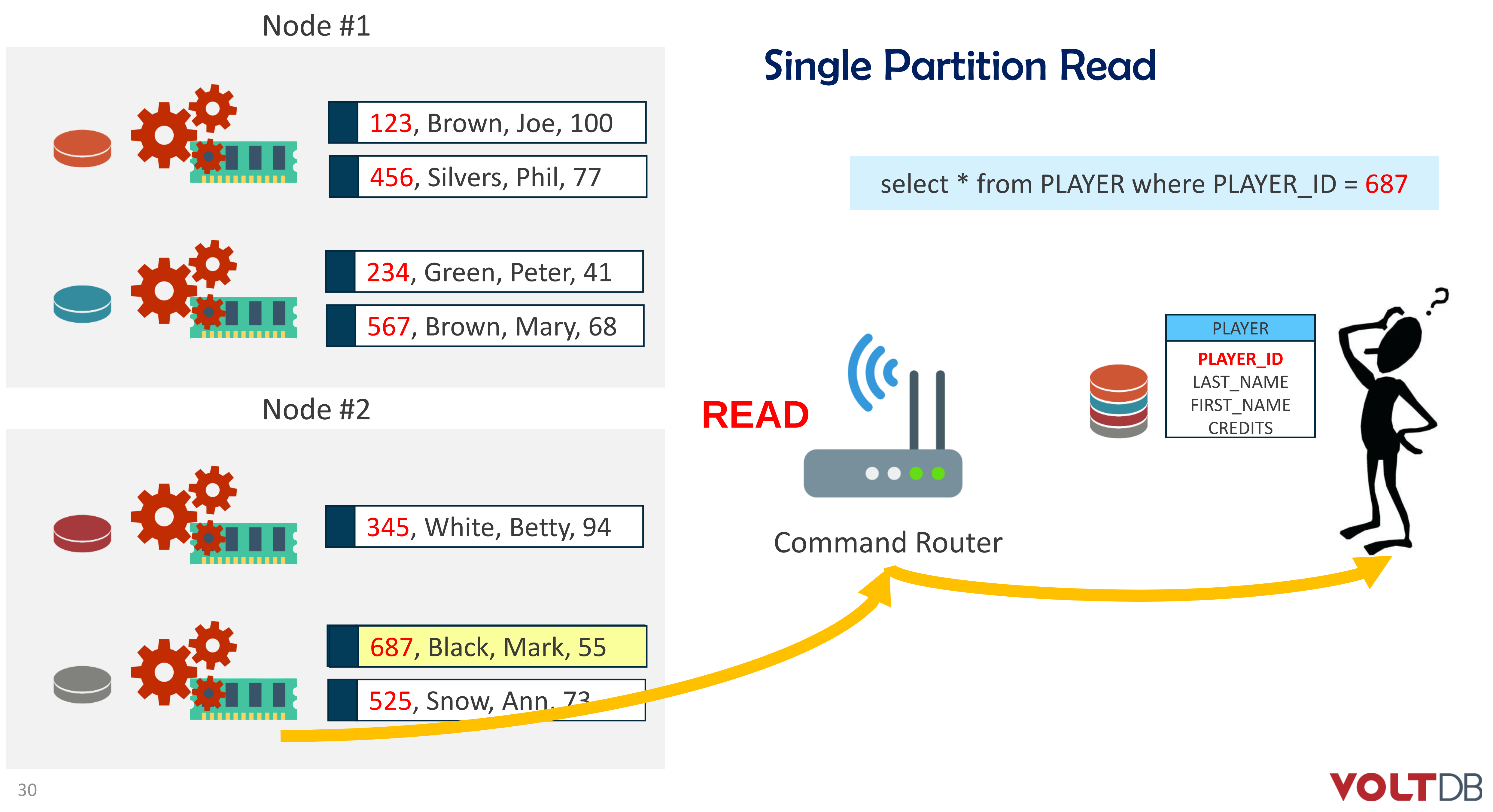
READ

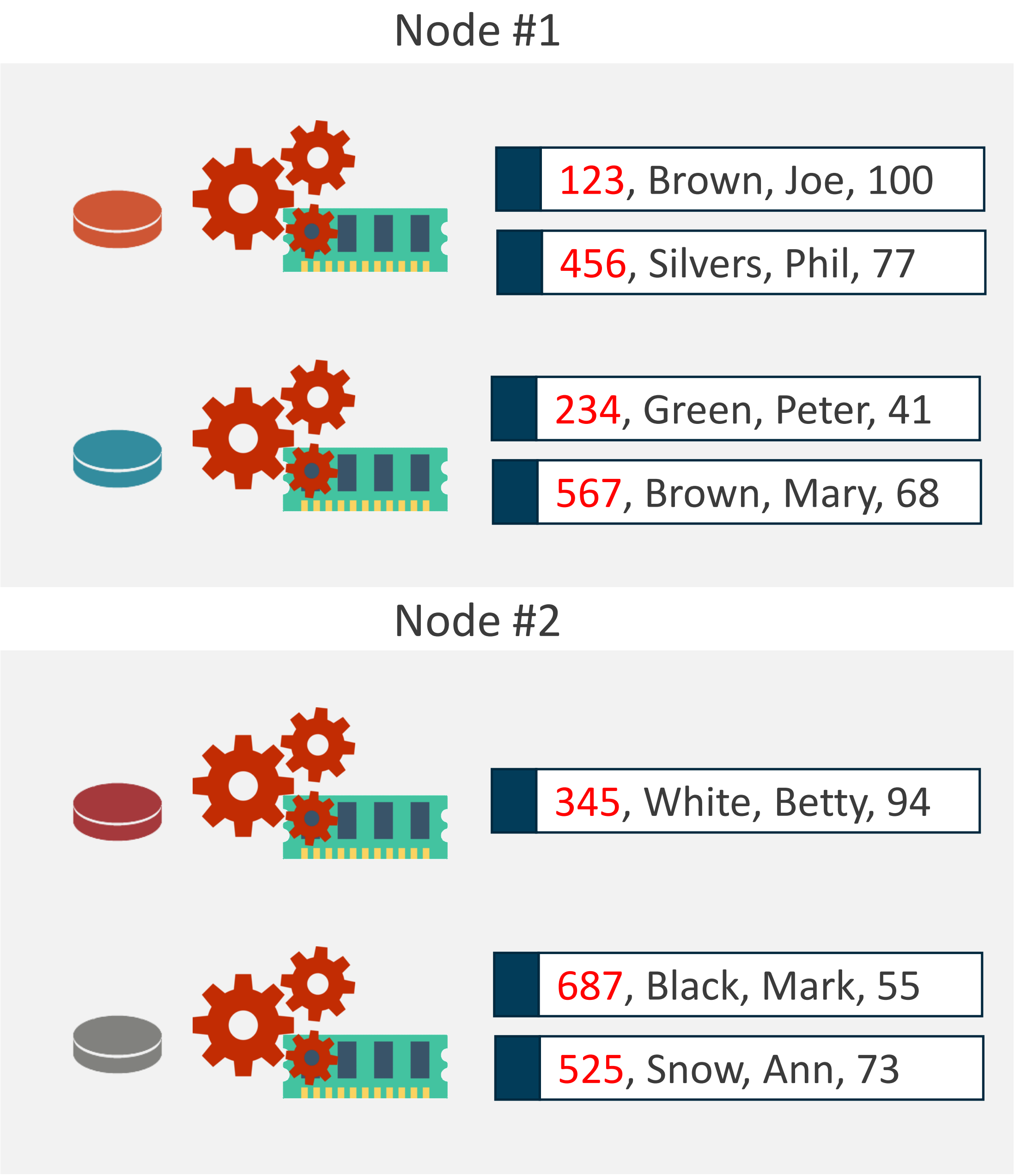
Command Router

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

Single Partition Read

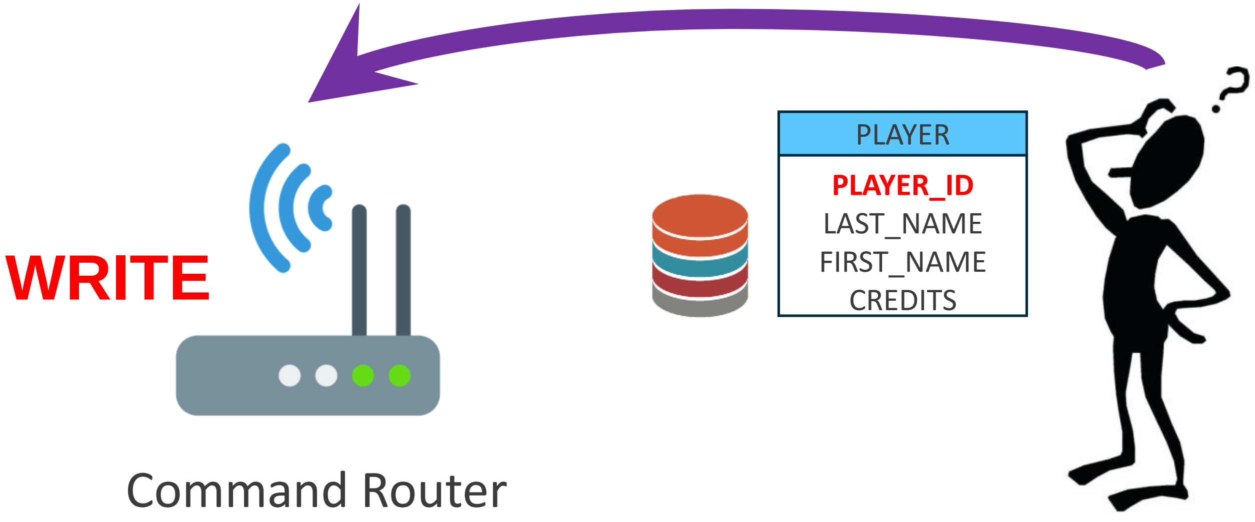



Node #1
Single Partition Read
123, Brown, Joe, 100
456, Silvers, Phil, 77
select * from PLAYER where PLAYER_ID = 687
234, Green, Peter, 41
567, Brown, Mary, 68
PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS
Node #2
READ
Command Router
345, White, Betty, 94
687, Black, Mark, 55
525, Snow, Ann, 73
VOLTDB

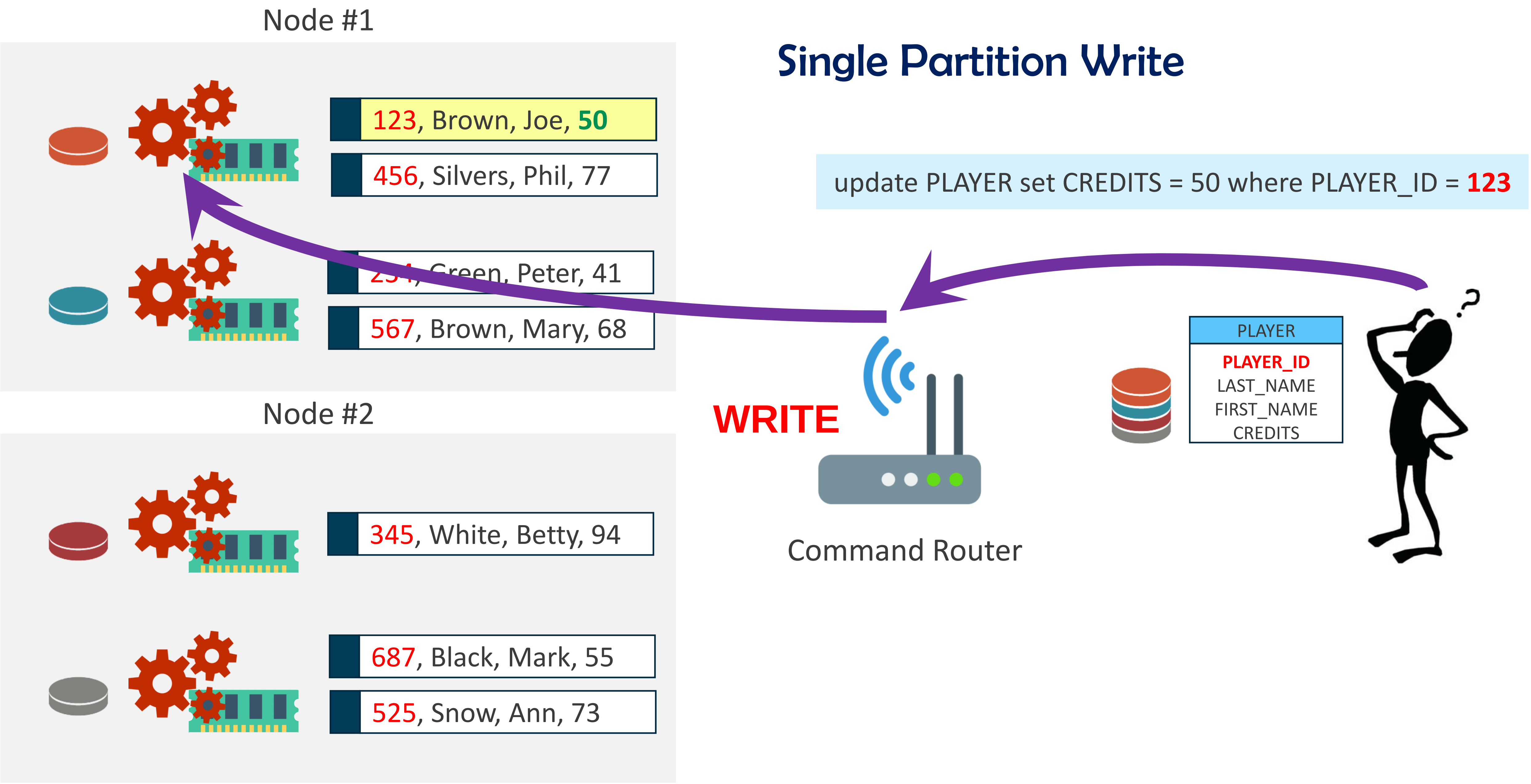


Single Partition Write

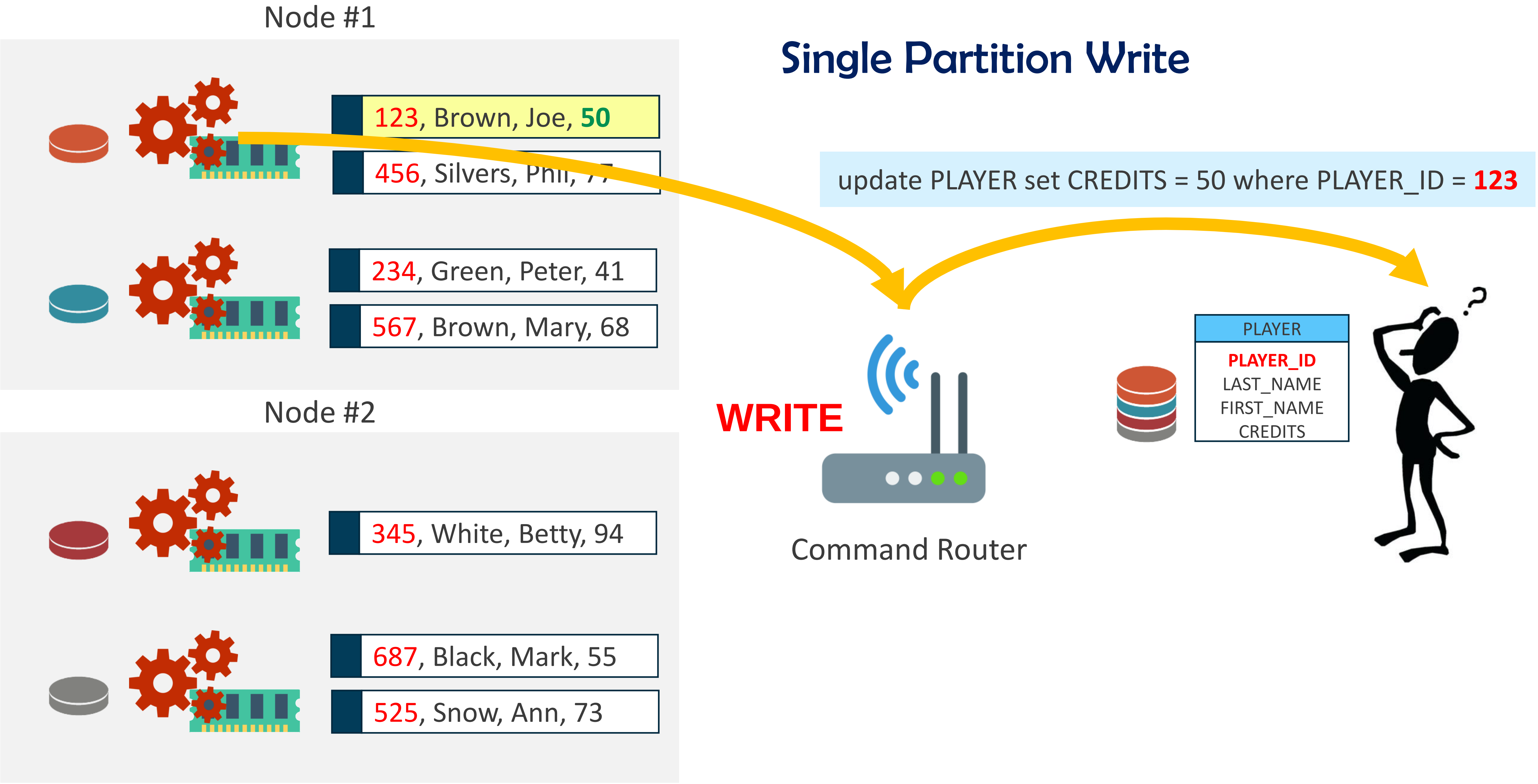
update PLAYER set CREDITS = 50 where PLAYER_ID = **123**



Single Partition Write

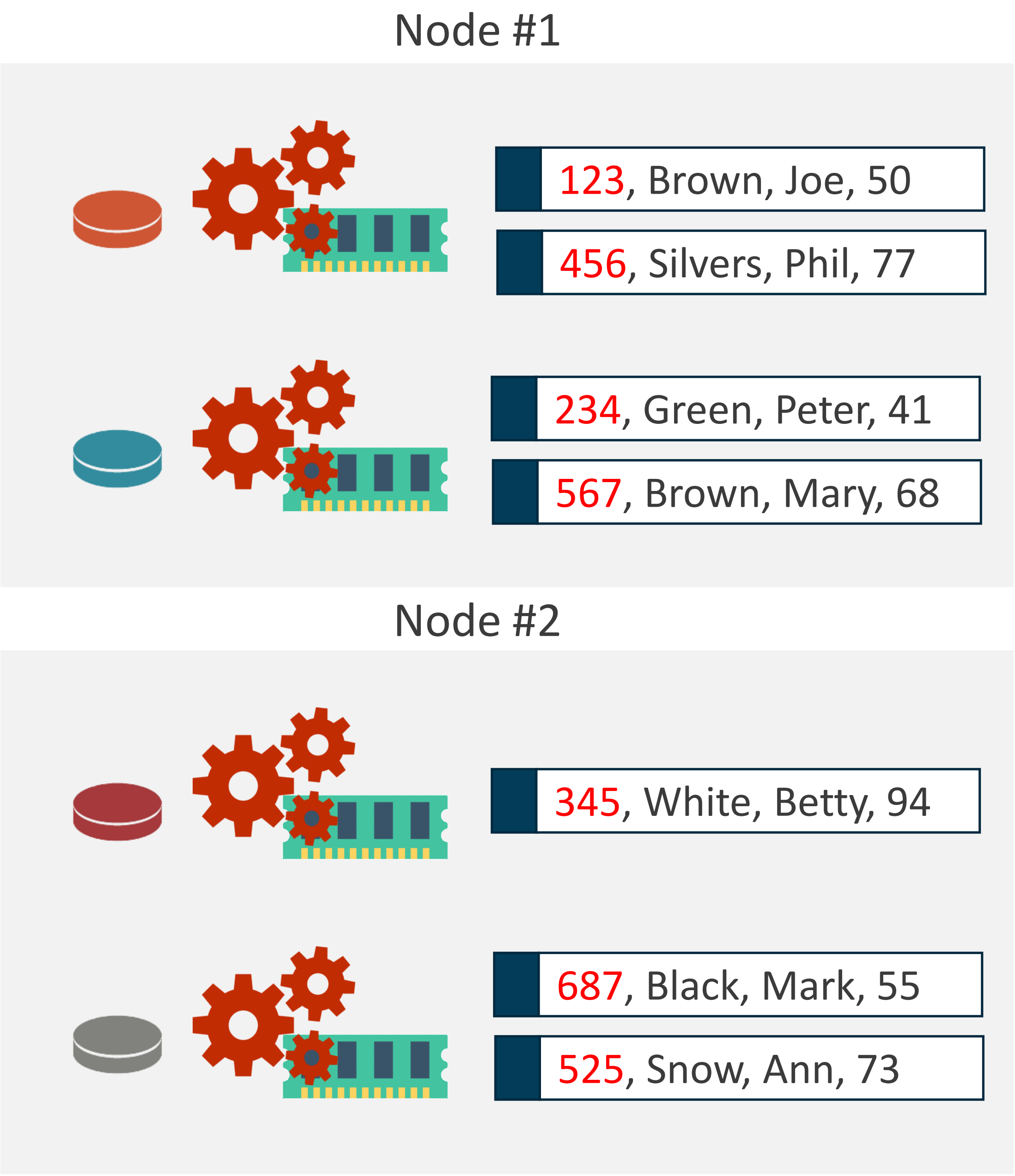


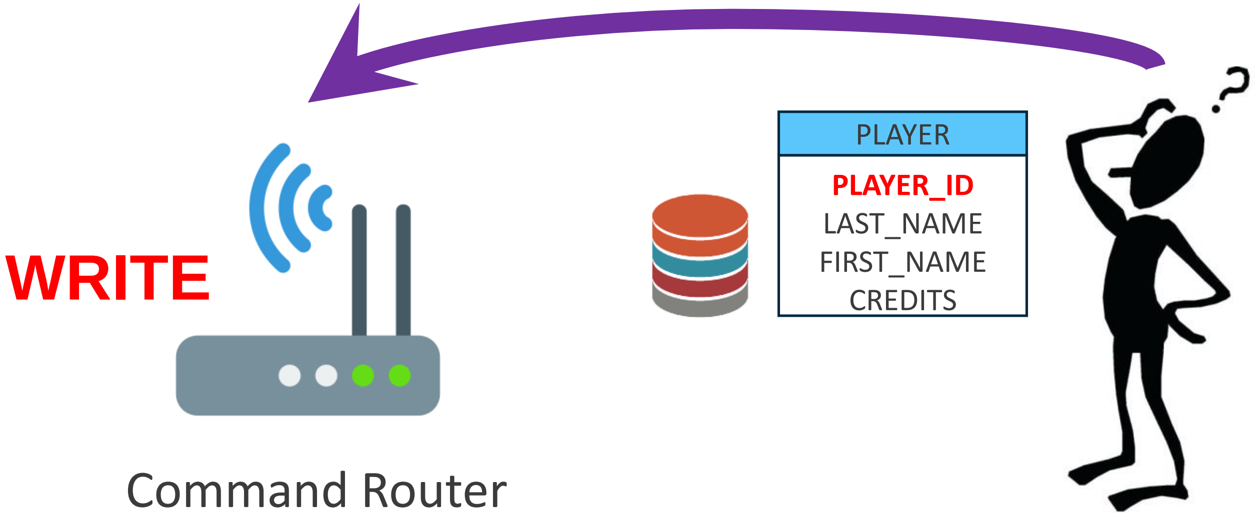
Single Partition Write



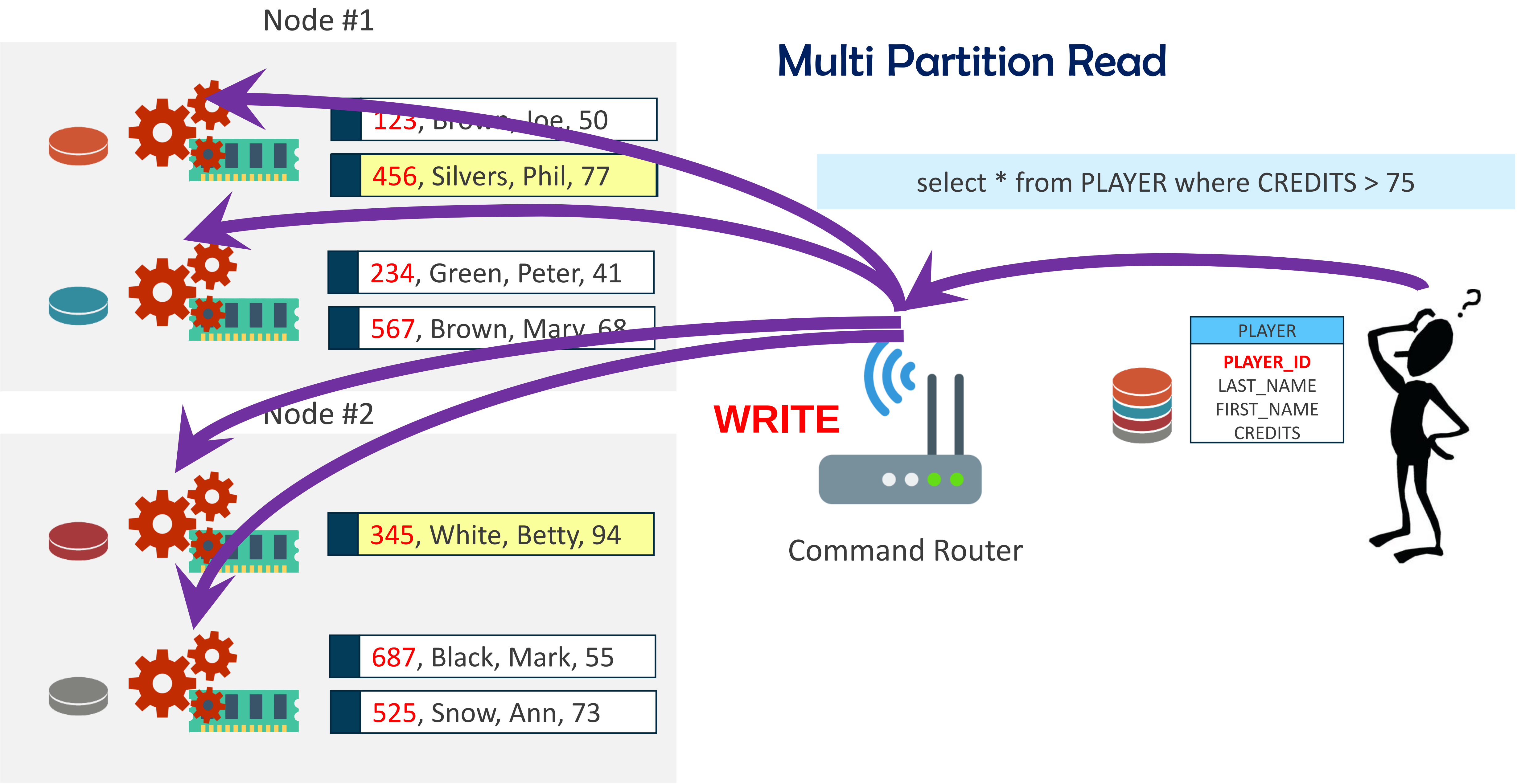
Multi Partition Read

select * from PLAYER where CREDITS > 75

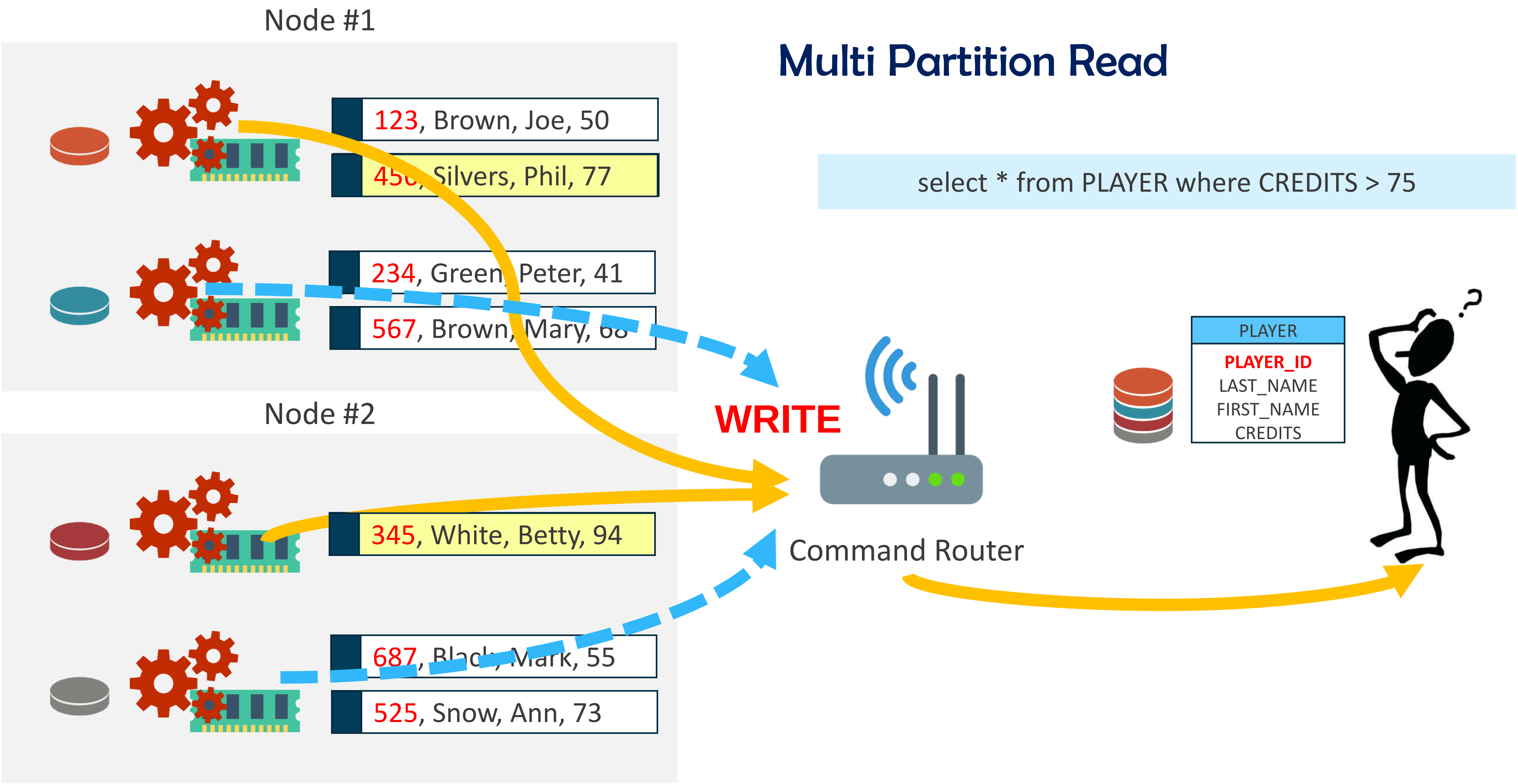




Multi Partition Read

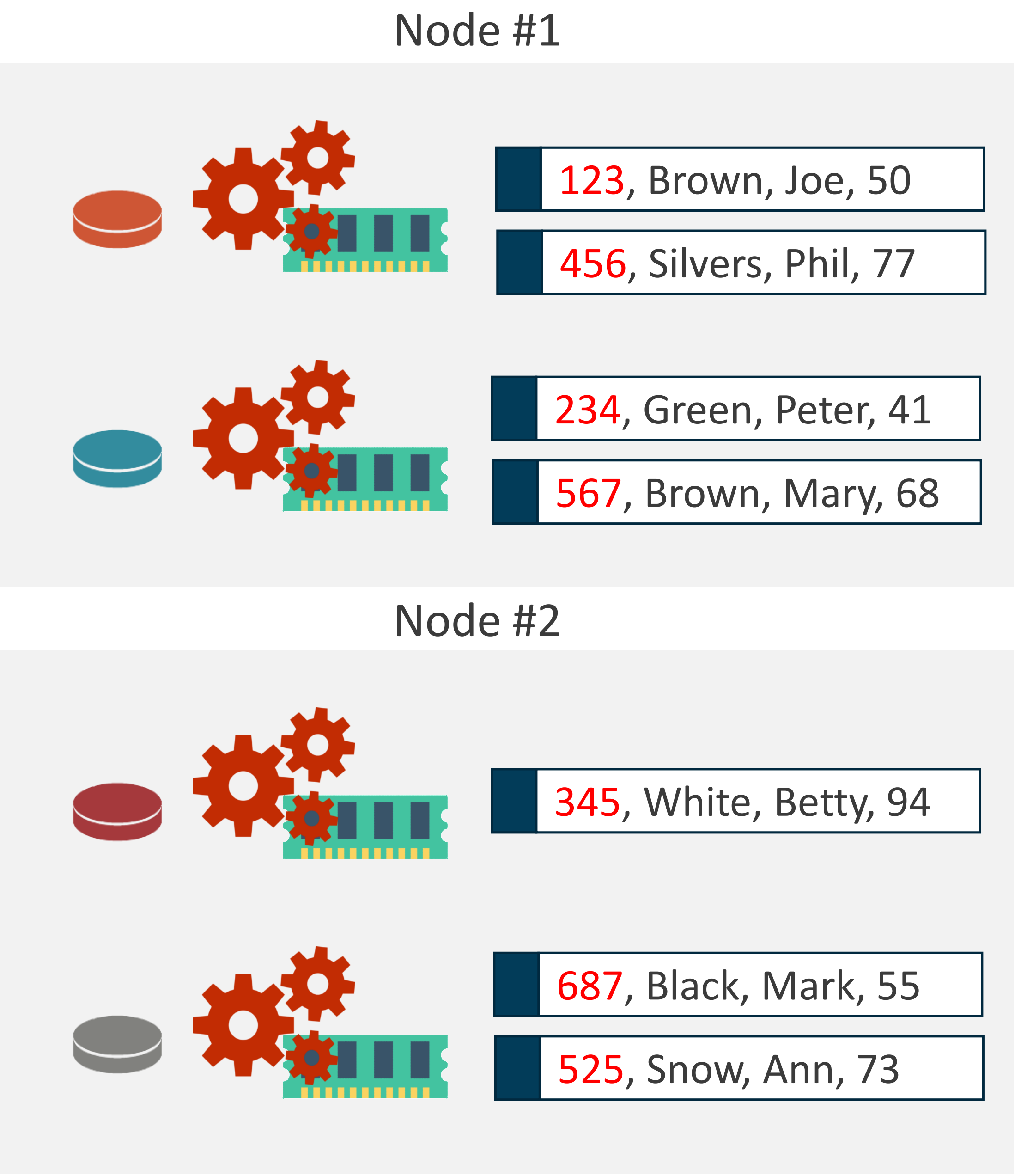


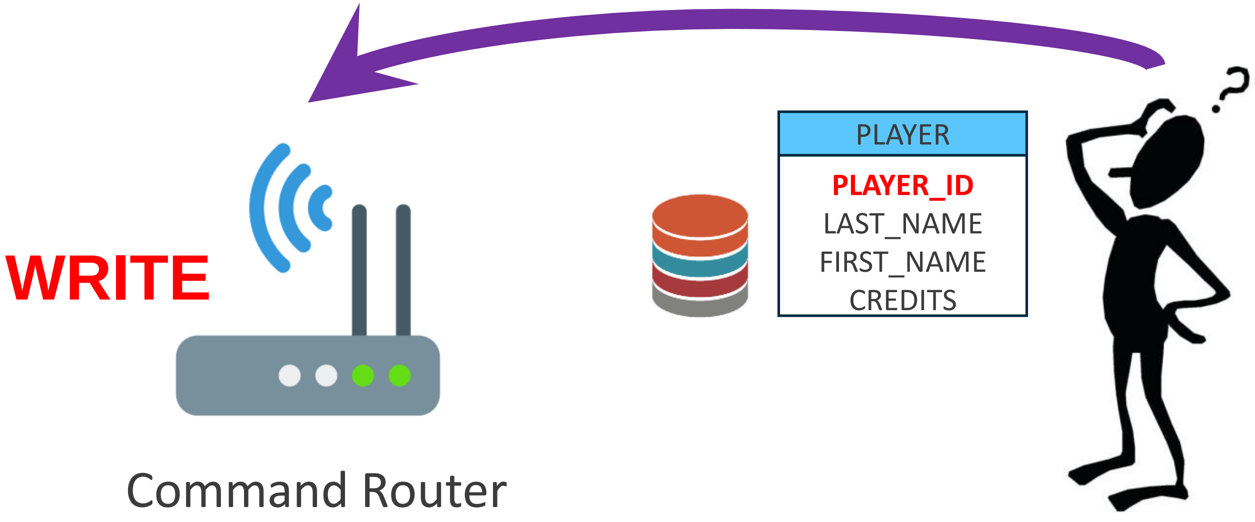
VOLTDB


Multi Partition Read
Node #1
123, Brown, Joe, 50
456, Silvers, Phil, 77
234, Green, Peter, 41
567, Brown, Mary, 68
Node #2
345, White, Betty, 94
687, Black, Mark, 55
525, Snow, Ann, 73
select * from PLAYER where CREDITS > 75
WRITE
Command Router
PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

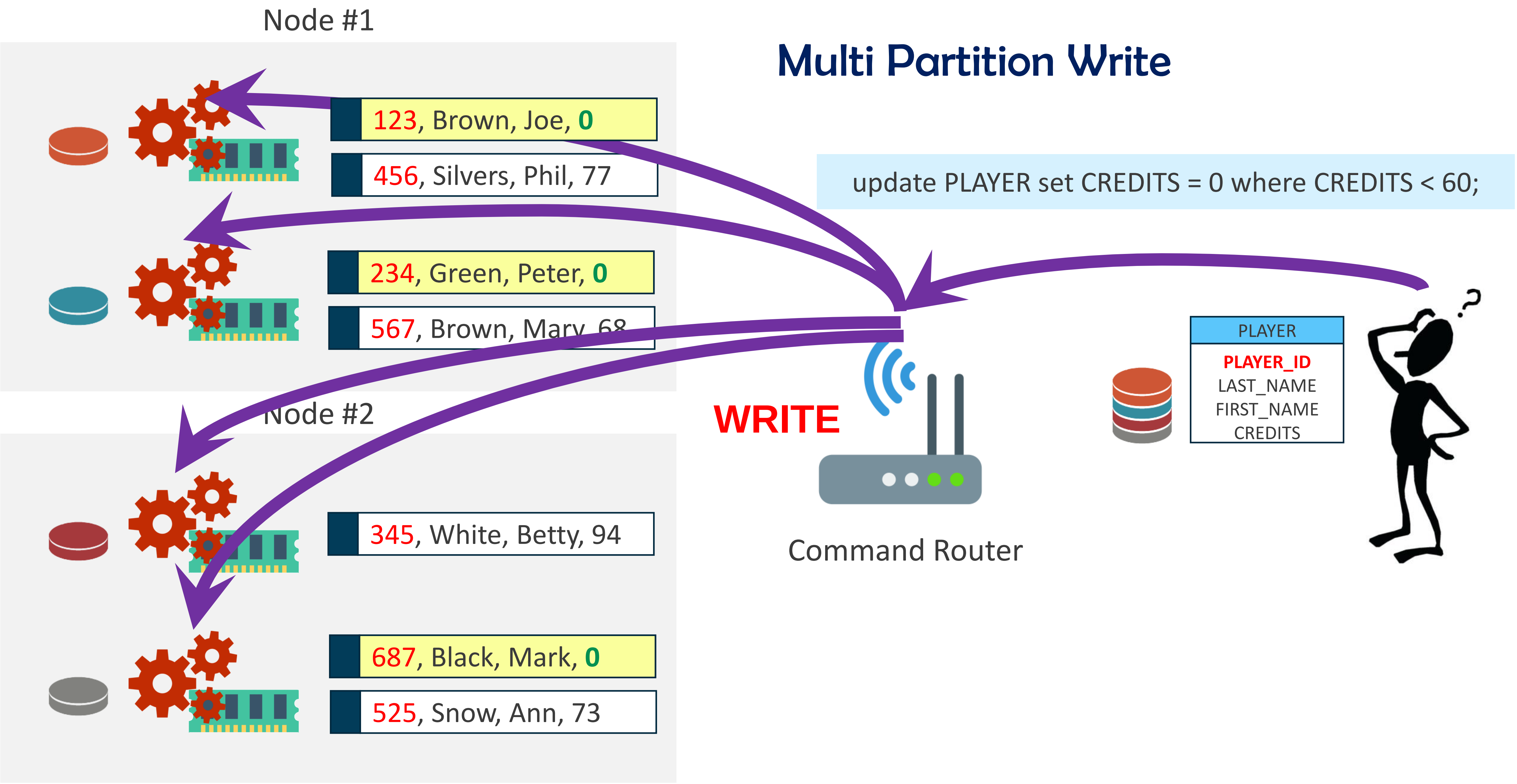
Multi Partition Write

update PLAYER set CREDITS = 0 where CREDITS < 60;

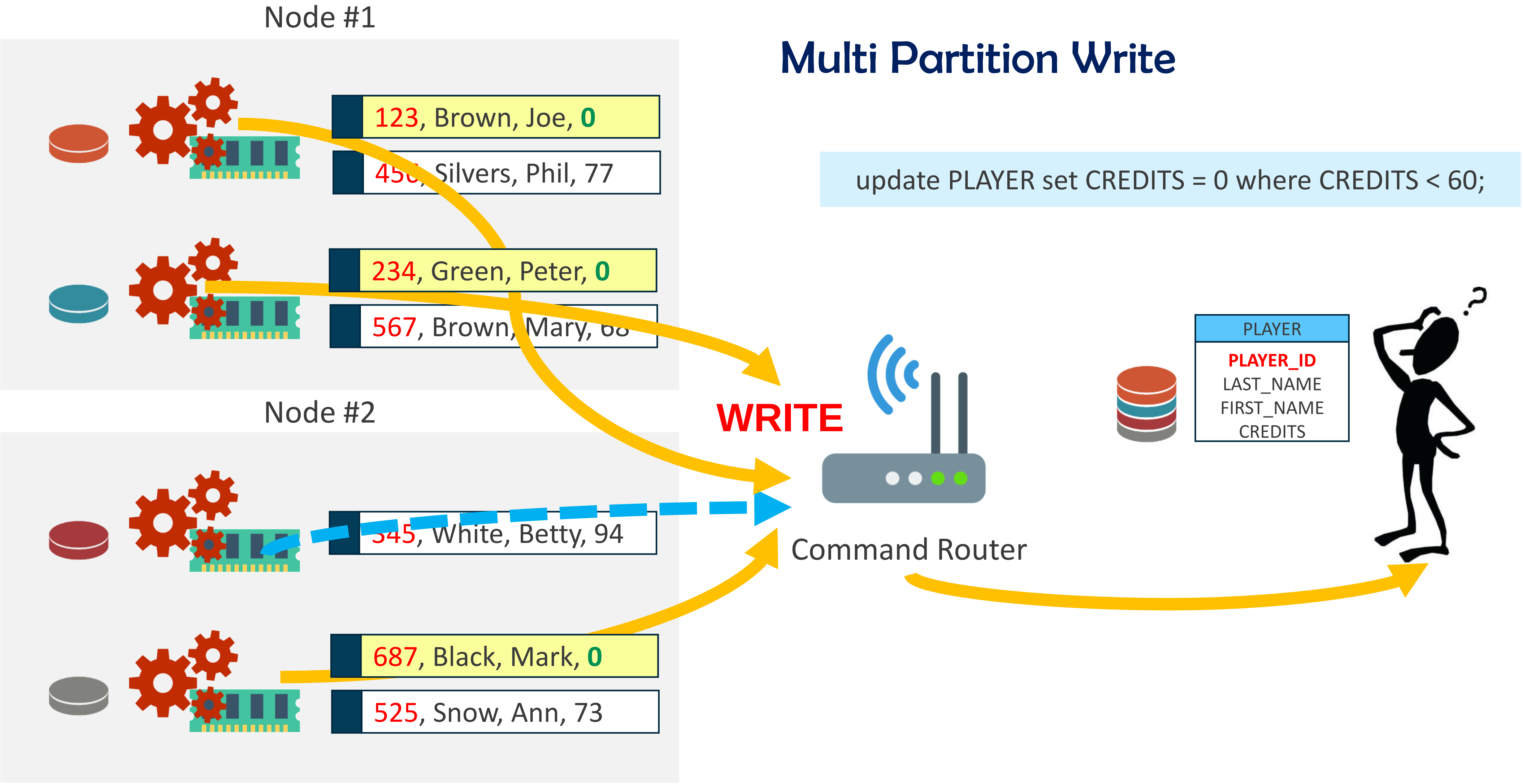




Multi Partition Write



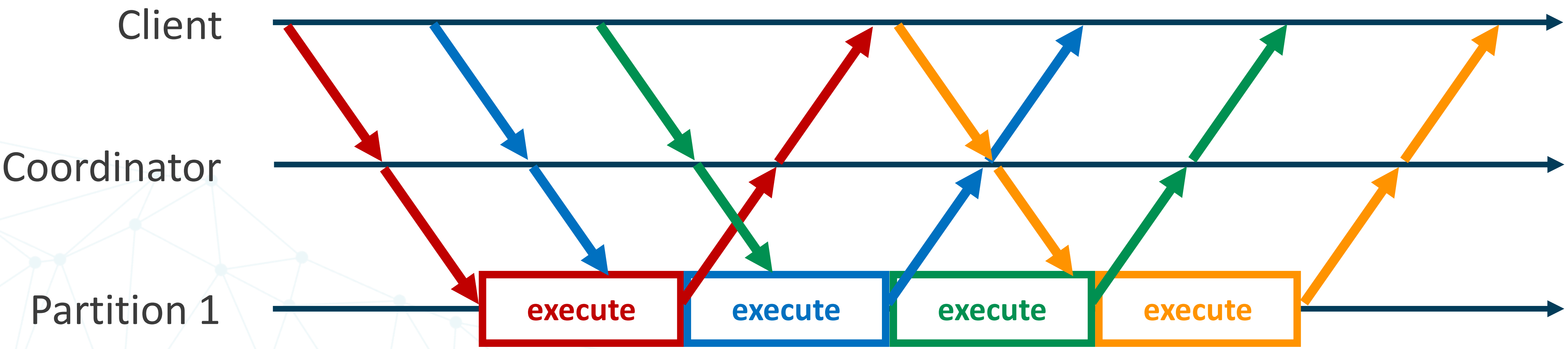
Multi Partition Write



Multi Partition Writes

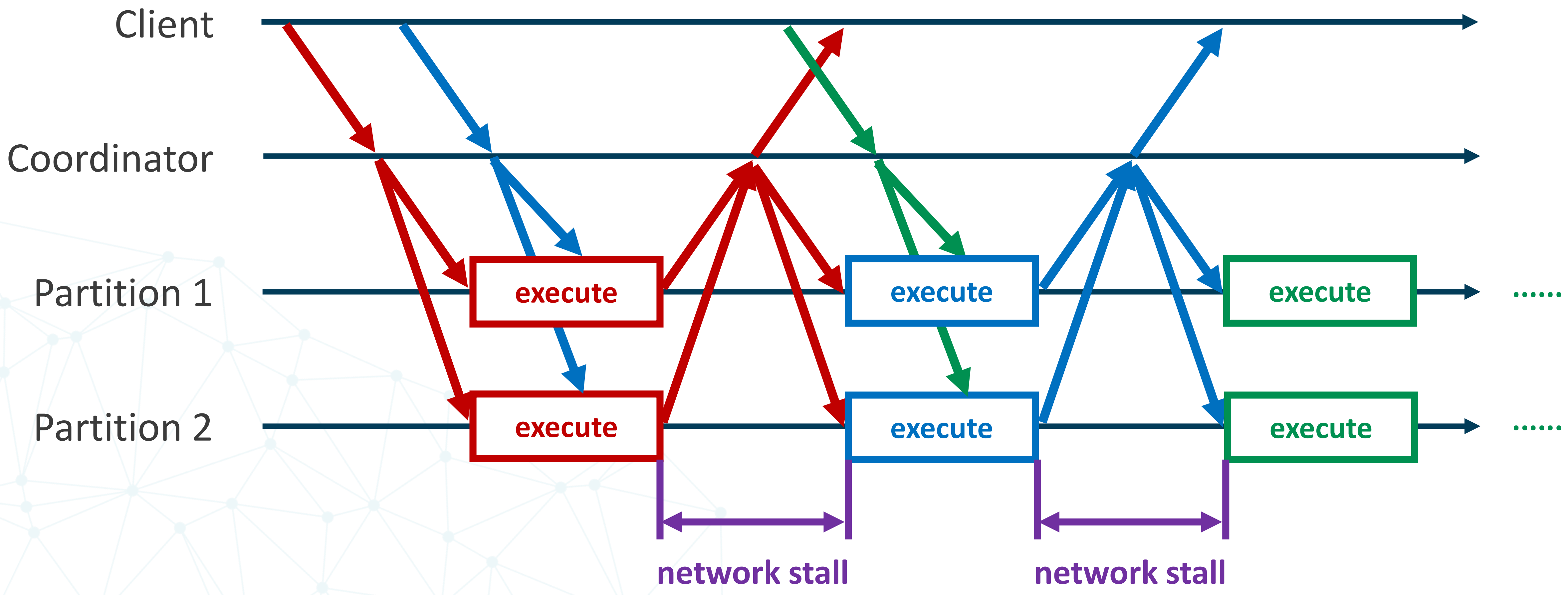
- Need two-phase commit.
- Simple solution – block until the transaction finishes.
- Introduces network stall - **BAD**.

Single Partition case

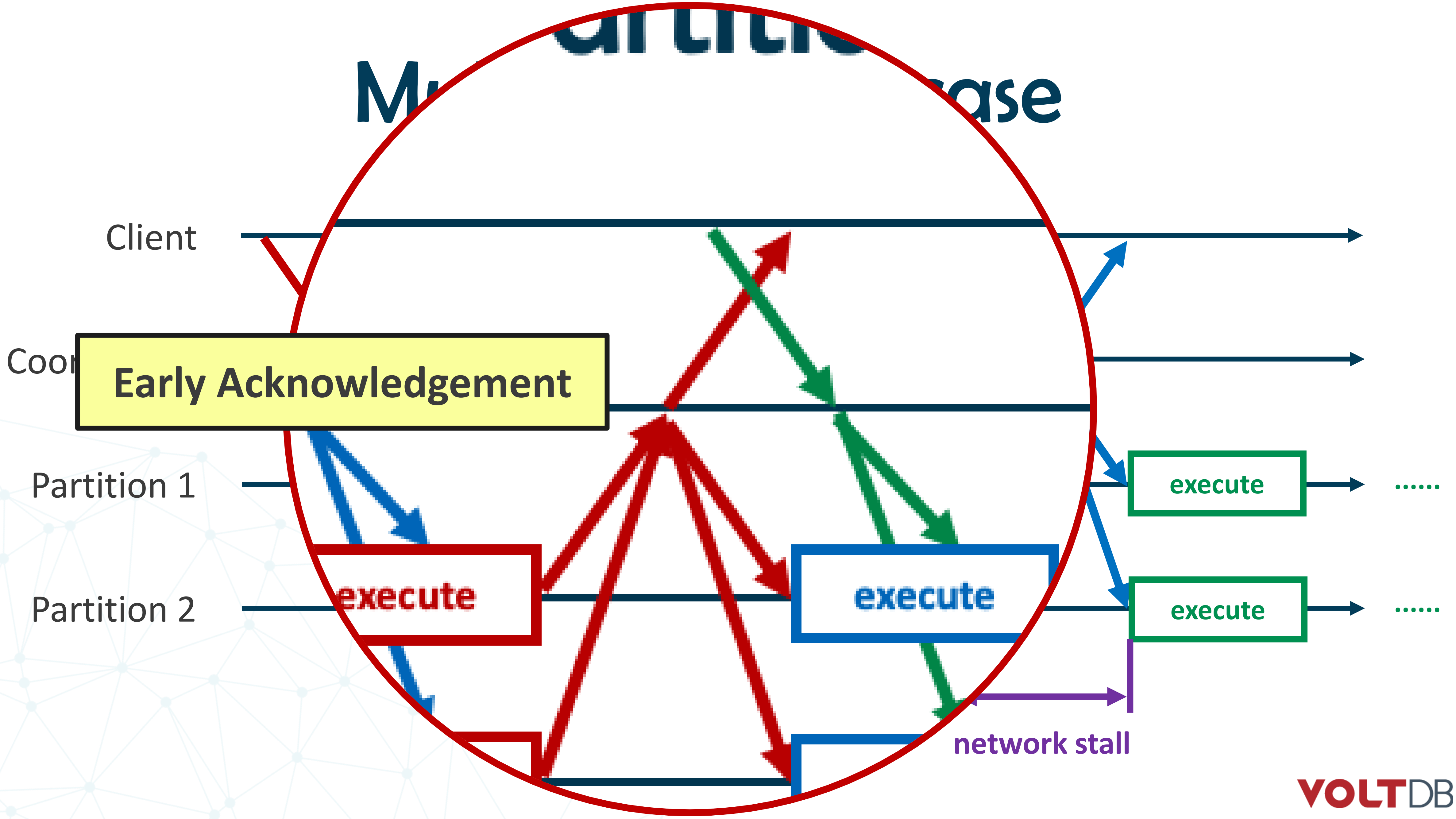


VOLTDB

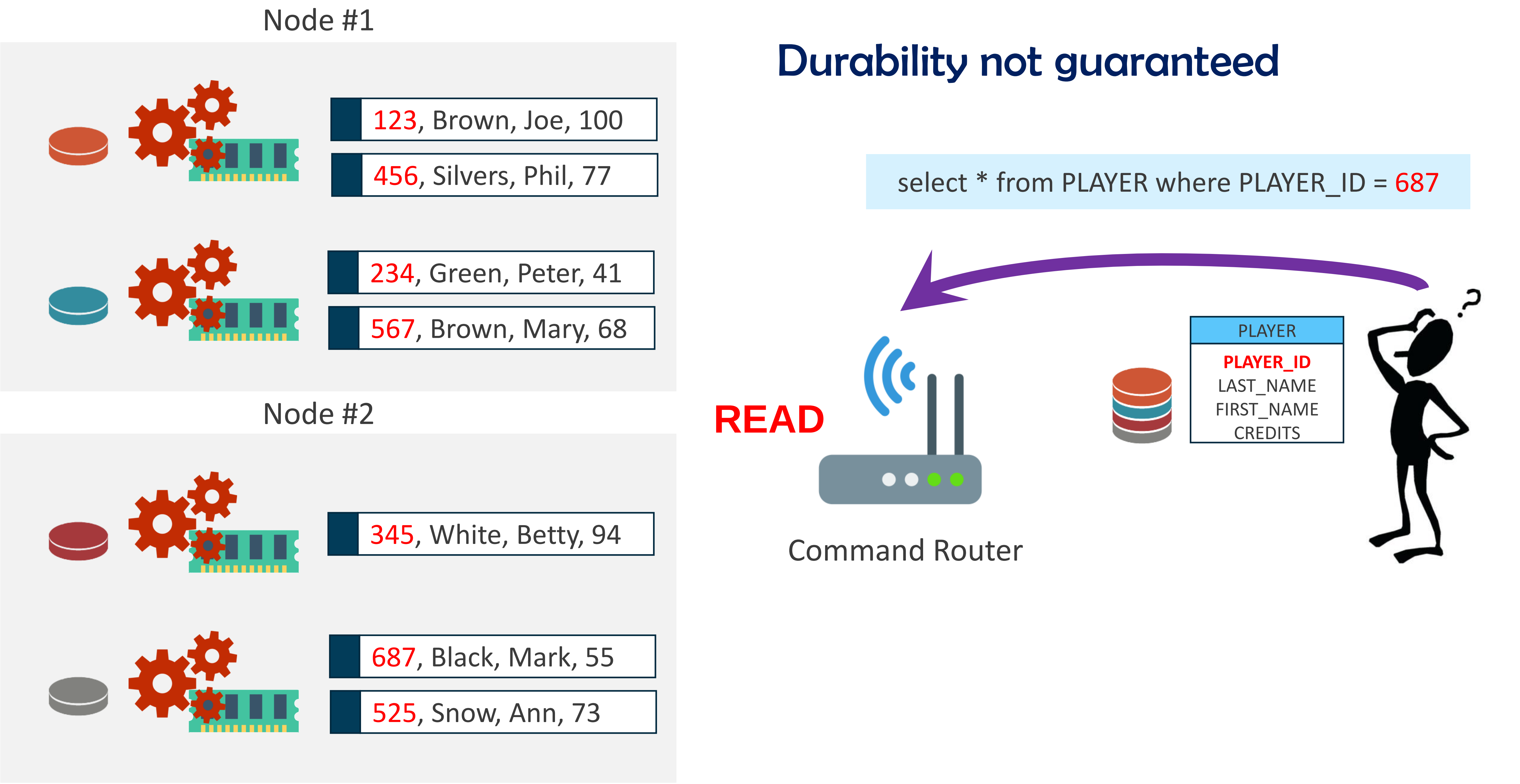
Multi Partition case



VOLTDB

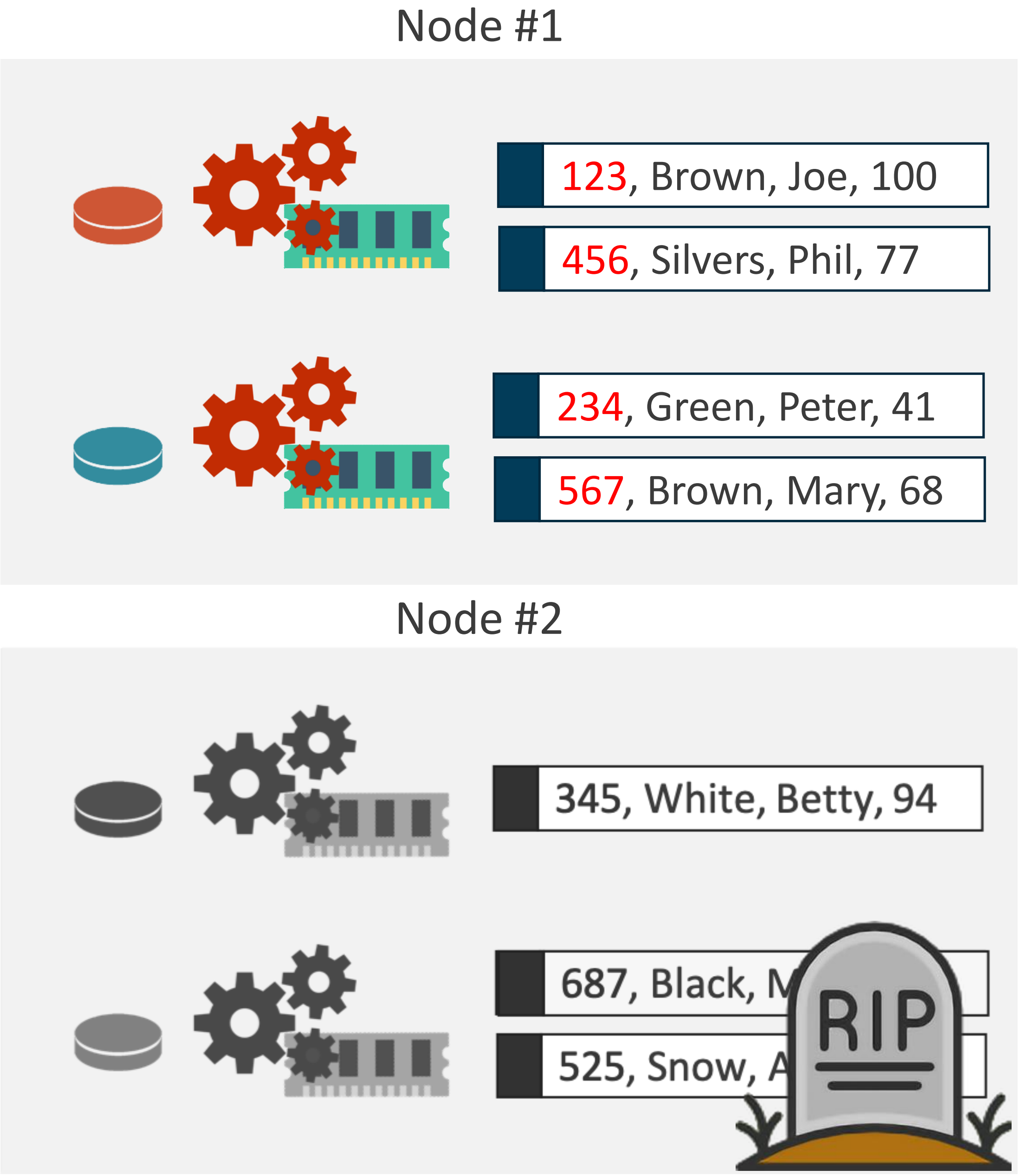


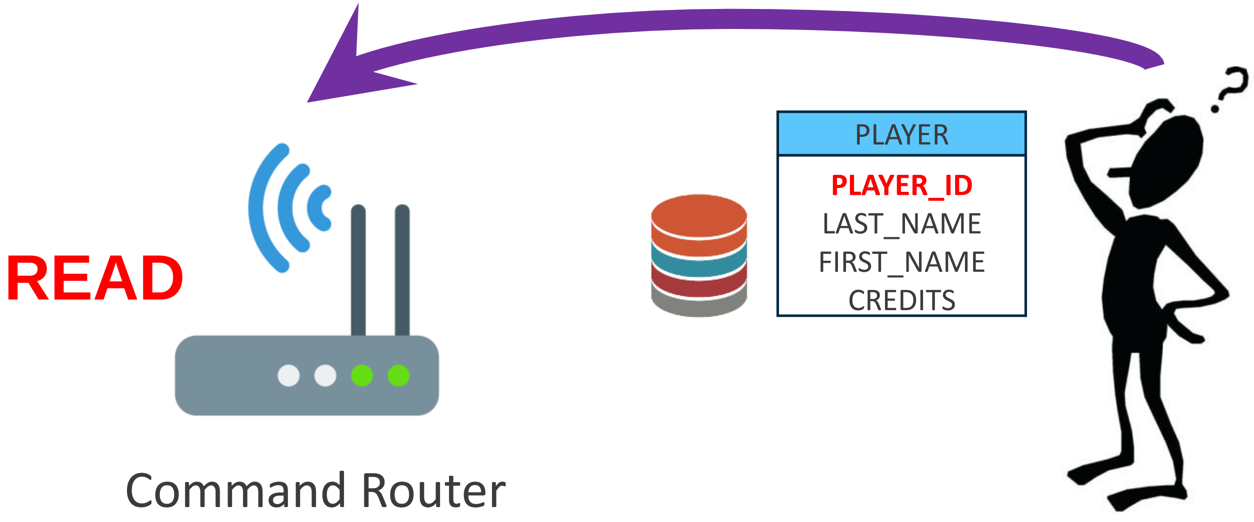
Client
Coor
Early Acknowledgement
Partition 1
Partition 2
execute
execute
execute
execute
......
......
network stall


Durability not guaranteed
Node #1
123, Brown, Joe, 100
456, Silvers, Phil, 77
234, Green, Peter, 41
567, Brown, Mary, 68
Node #2
345, White, Betty, 94
687, Black, Mark, 55
525, Snow, Ann, 73
select * from PLAYER where PLAYER_ID = 687
READ
Command Router
PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

Durability not guaranteed

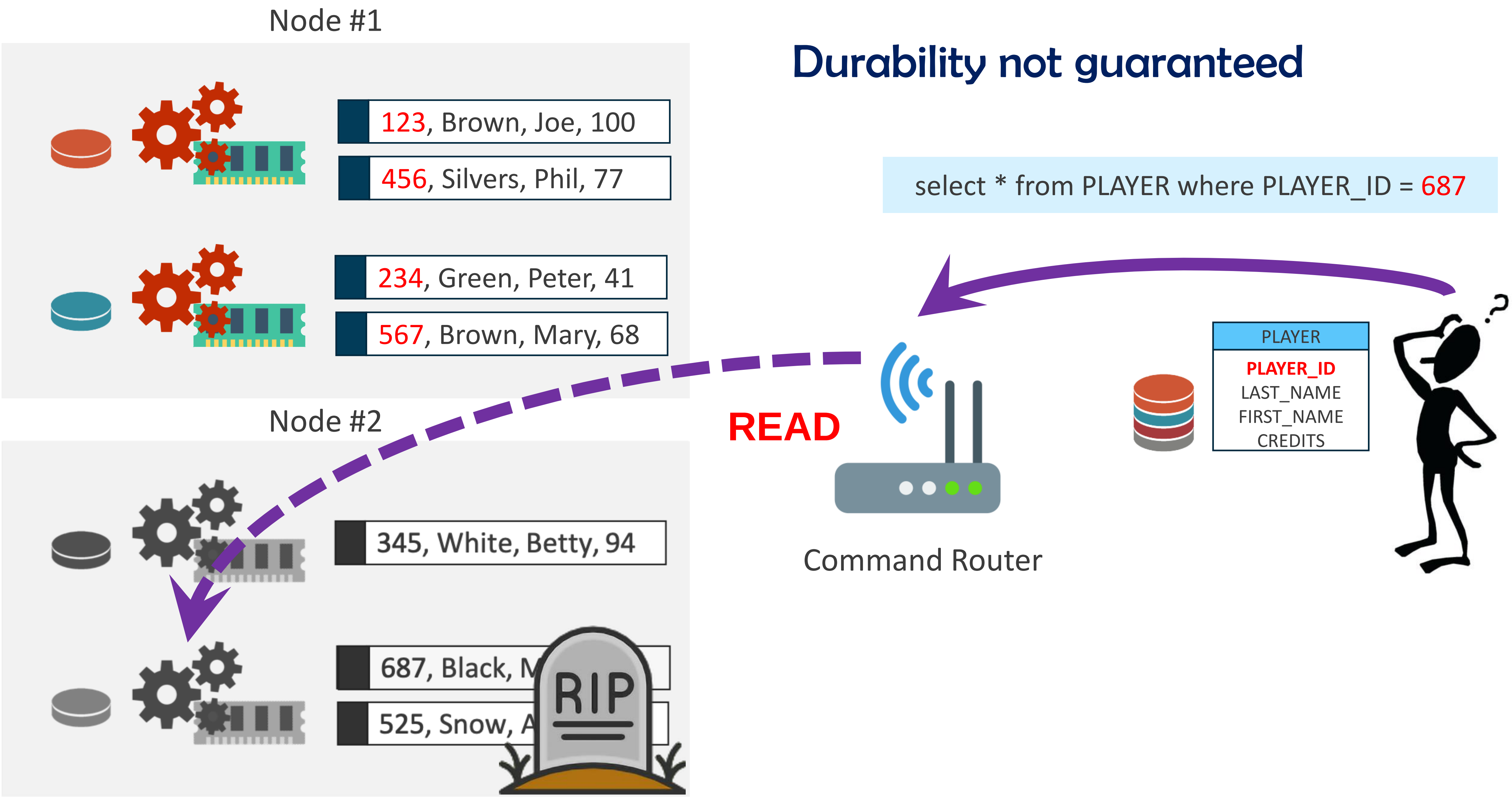
select * from PLAYER where PLAYER_ID = 687





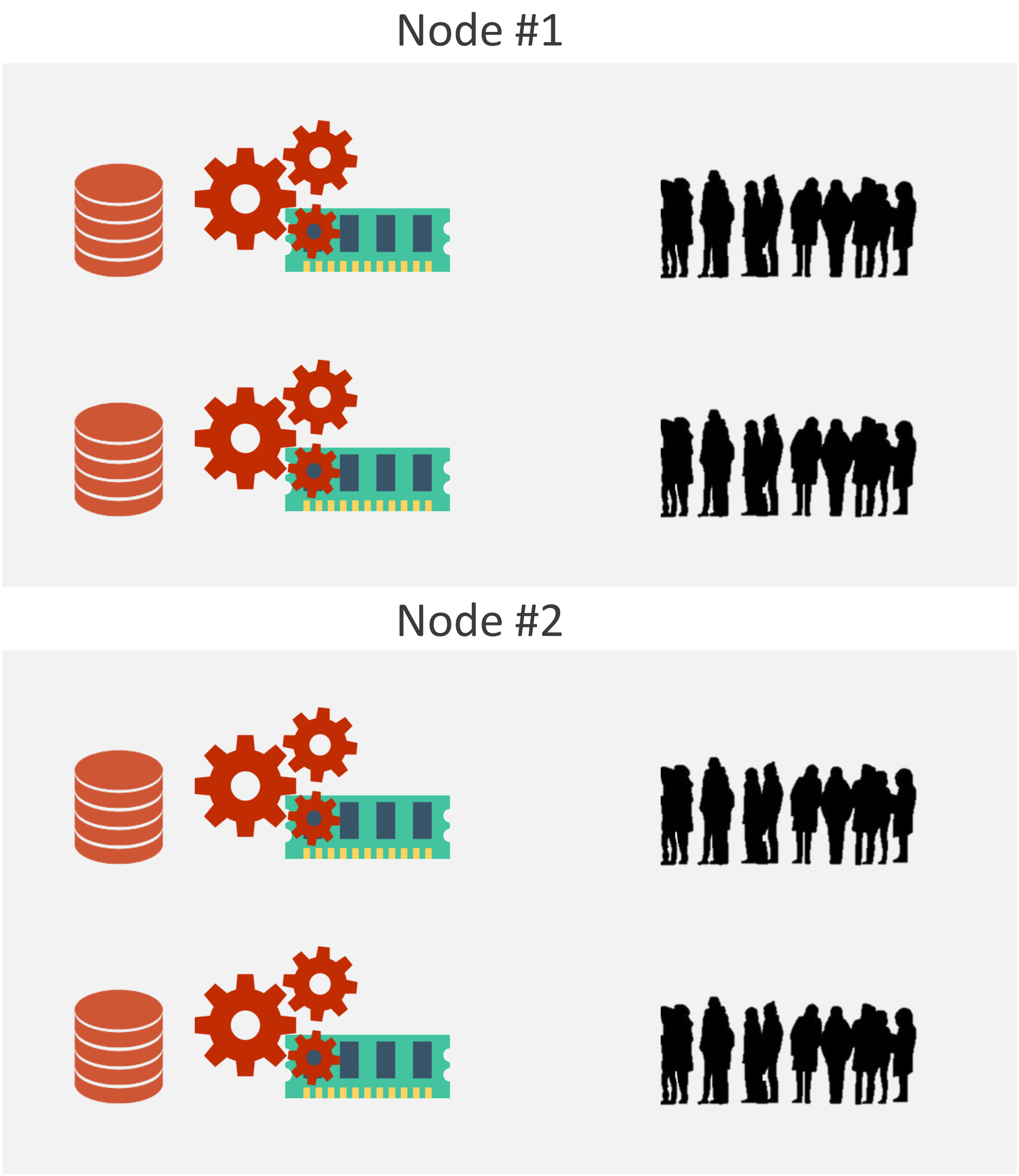
VOLTDB

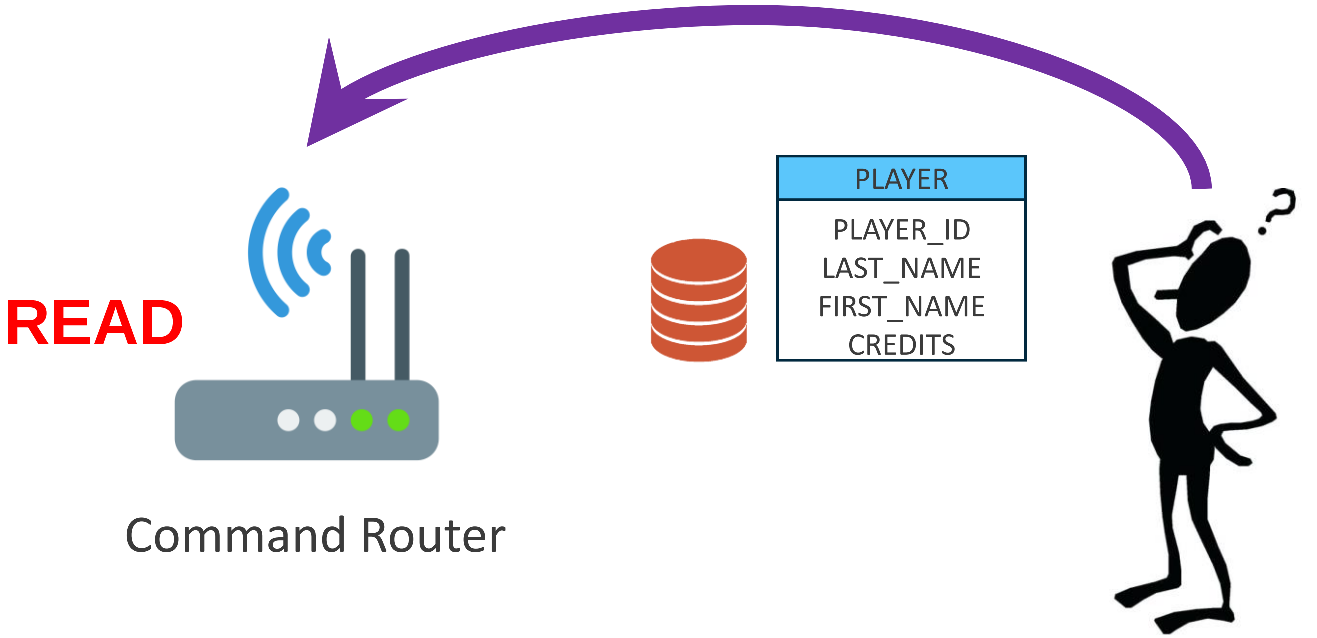
Durability not guaranteed



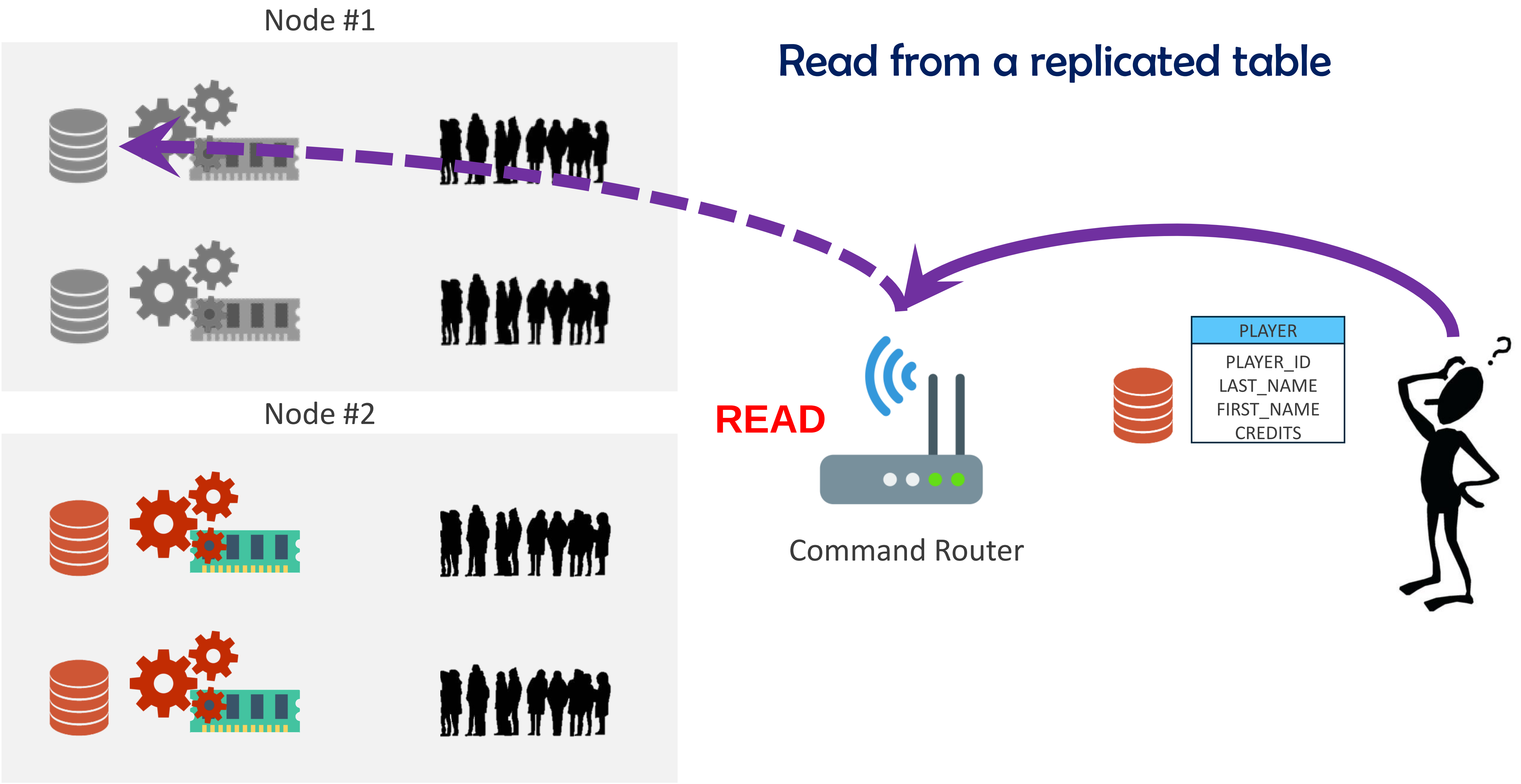
VOLTDB

Read from a replicated table

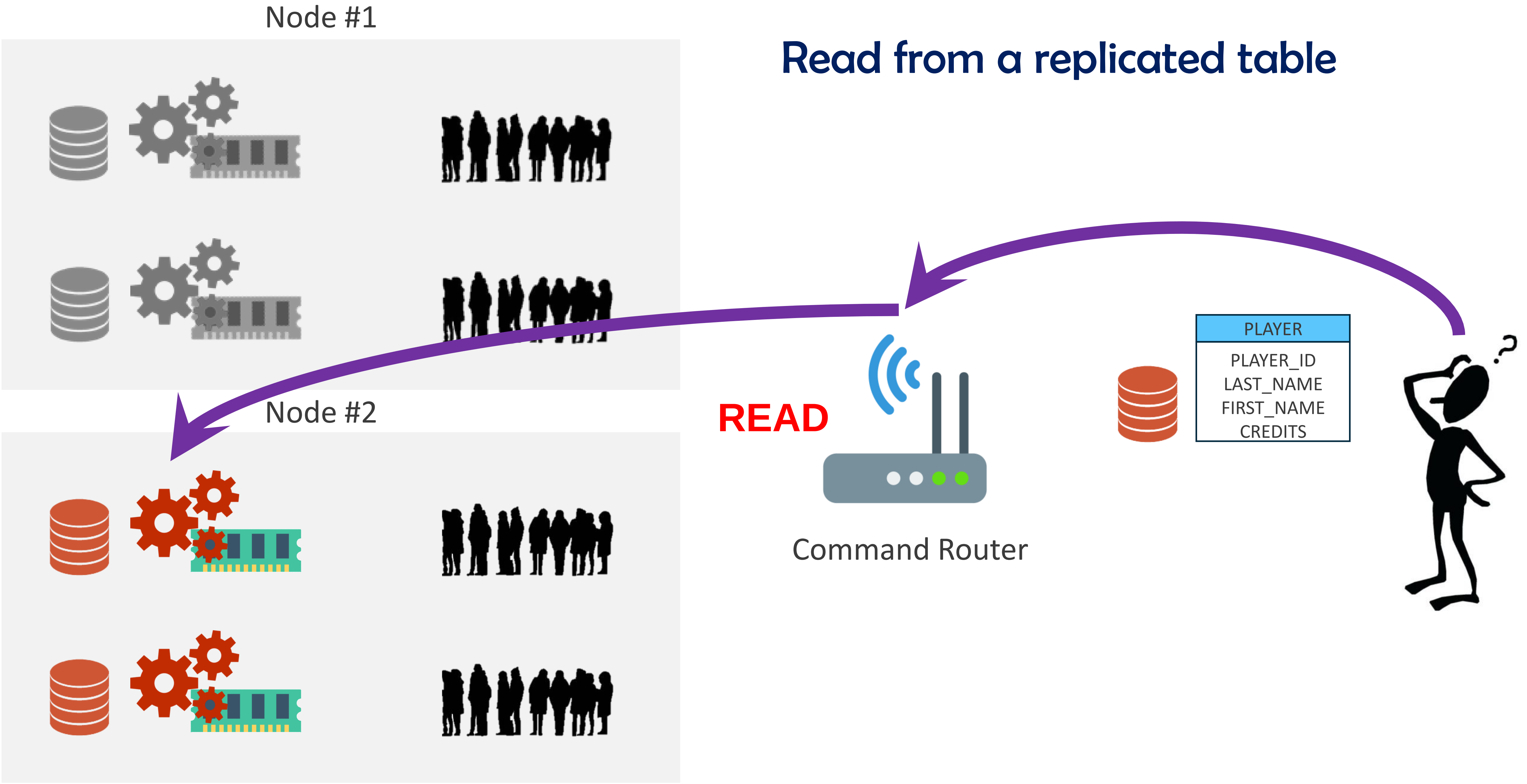




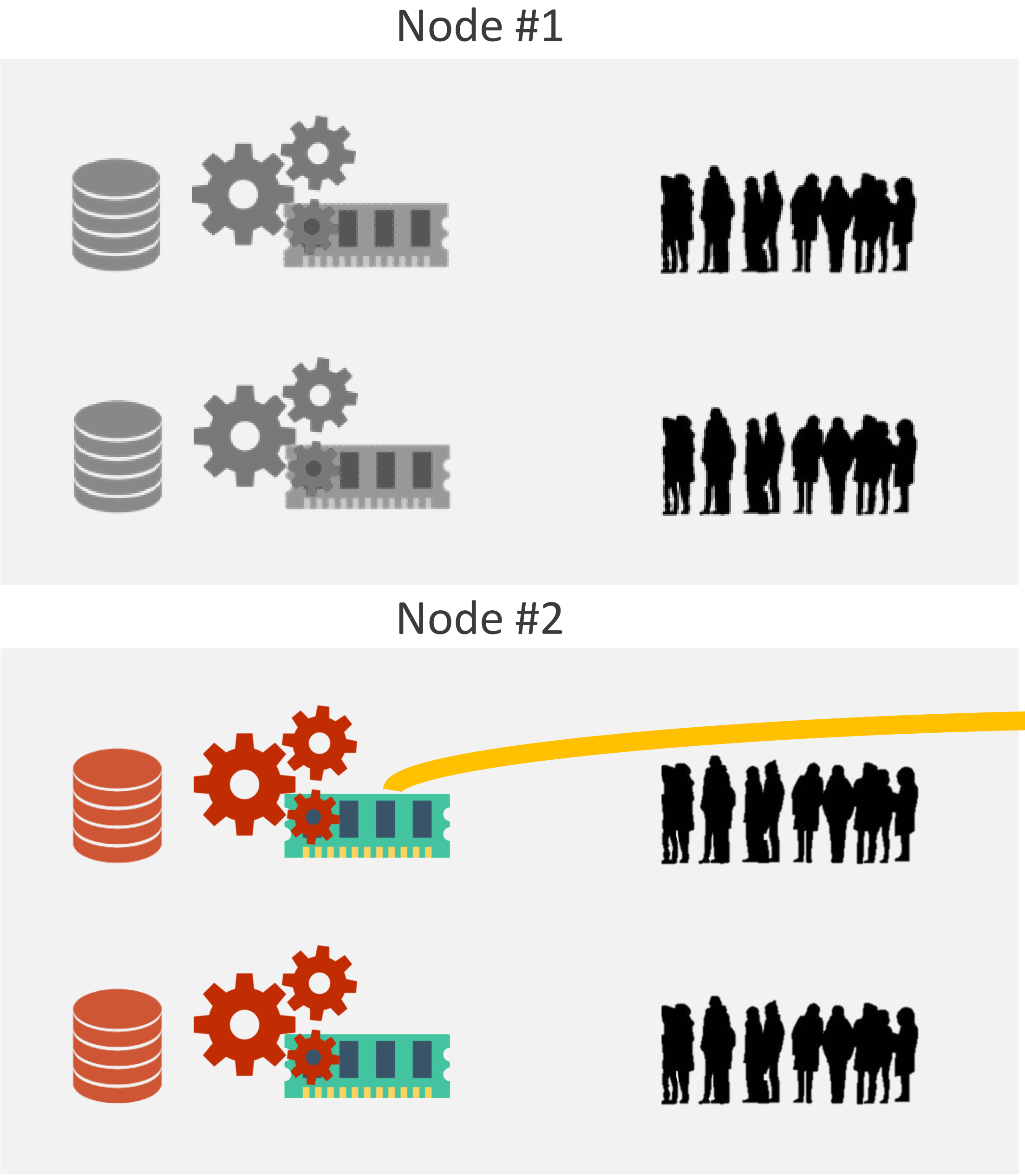
Read from a replicated table

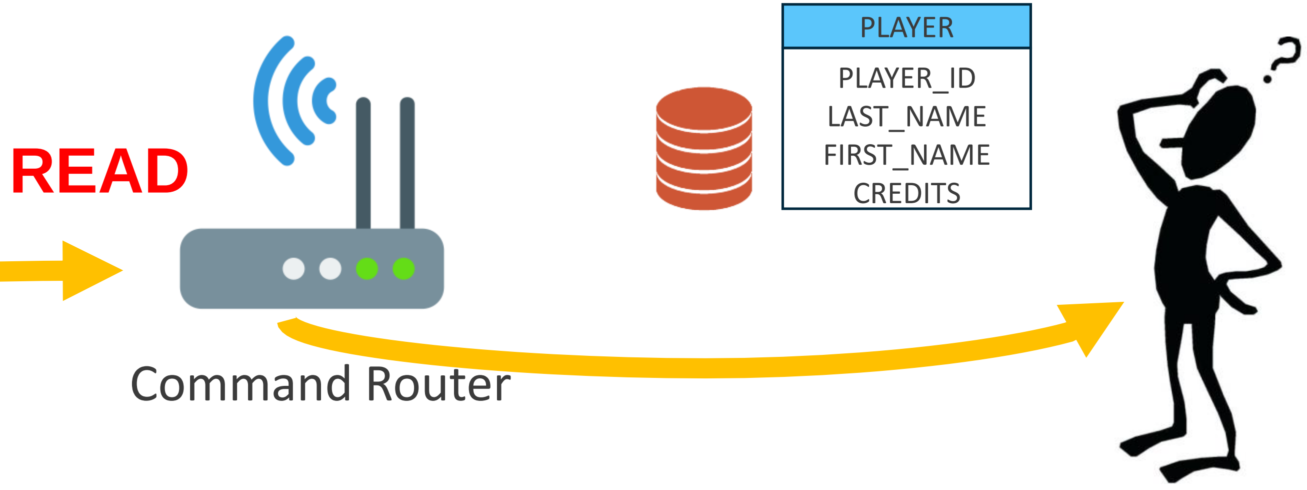


Read from a replicated table



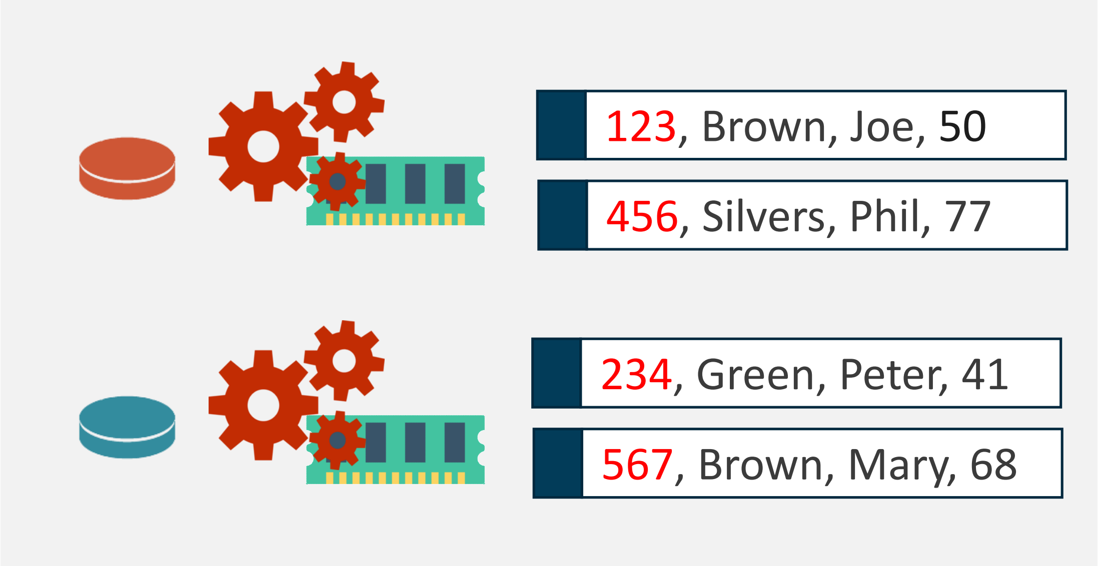
Read from a replicated table





Replication!

Node #1

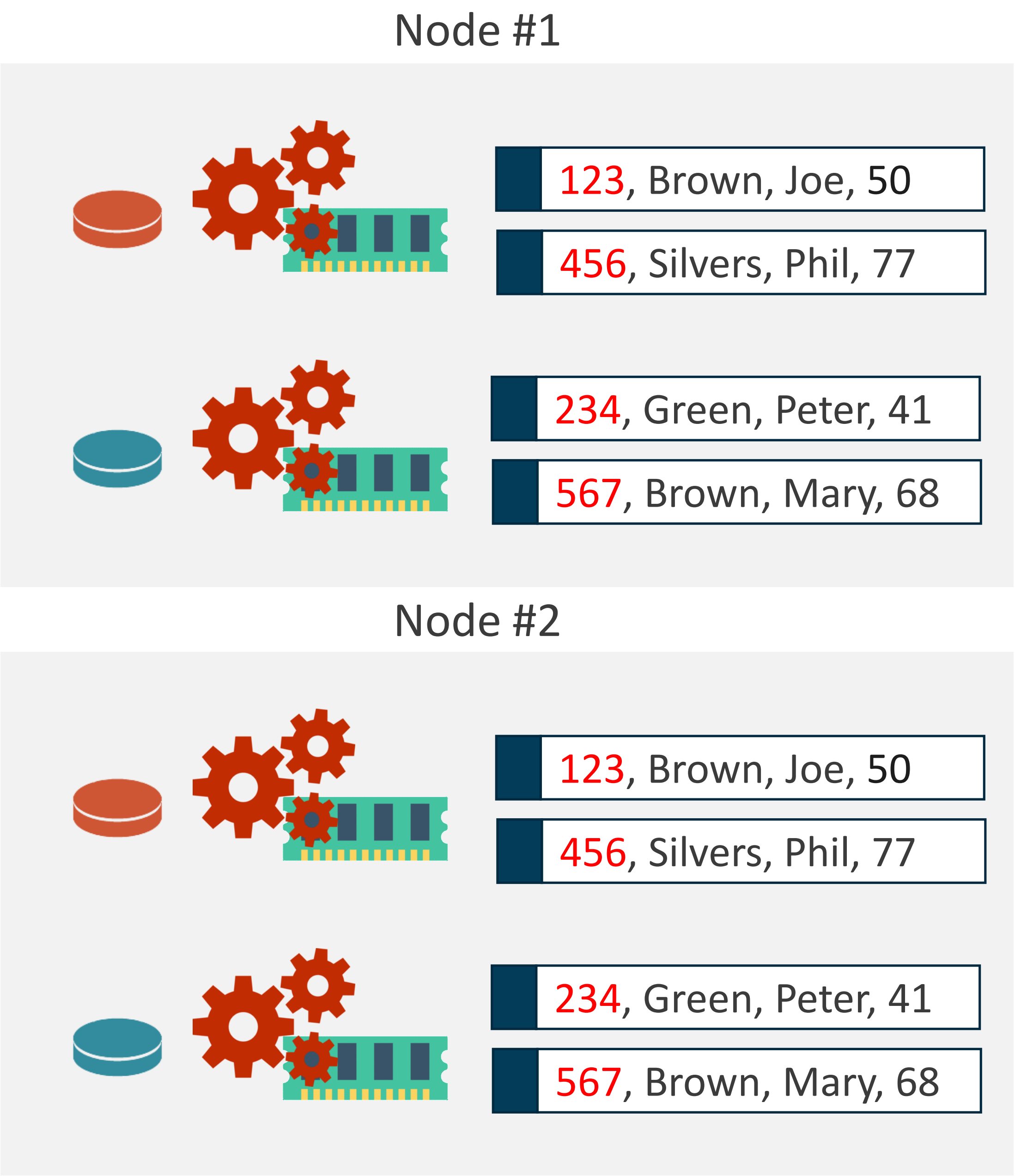


Durability through replication

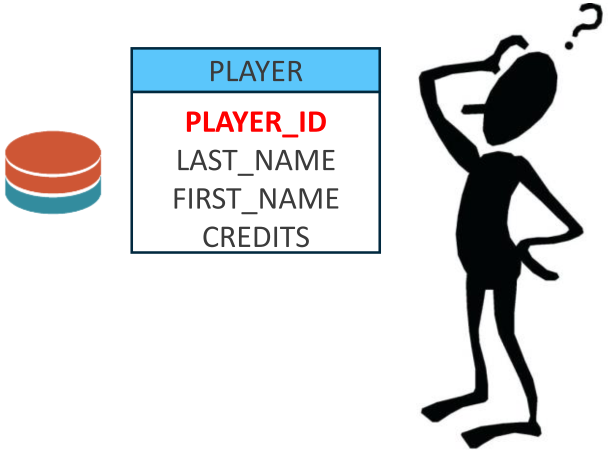
Command Router

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

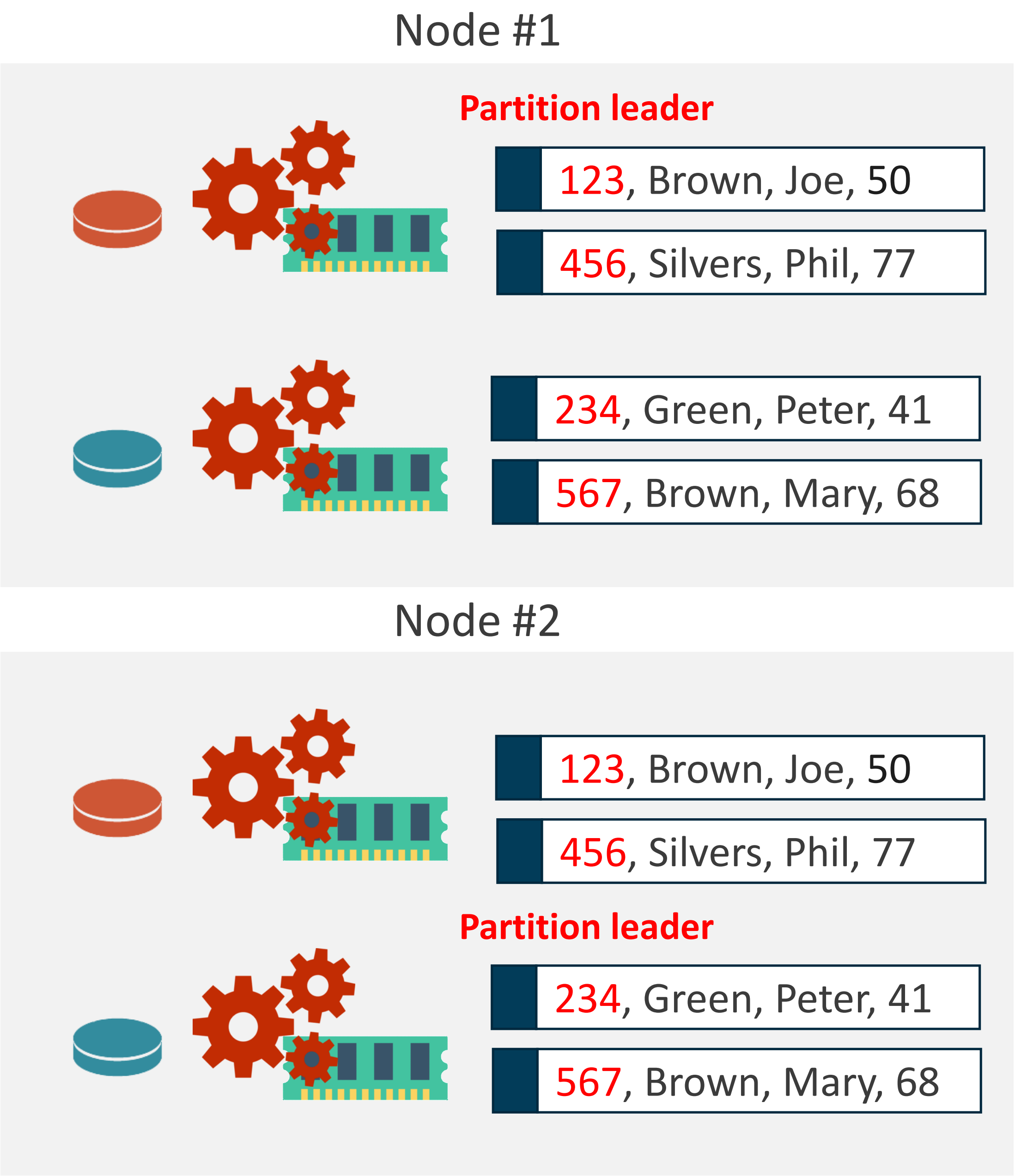
Durability through replication

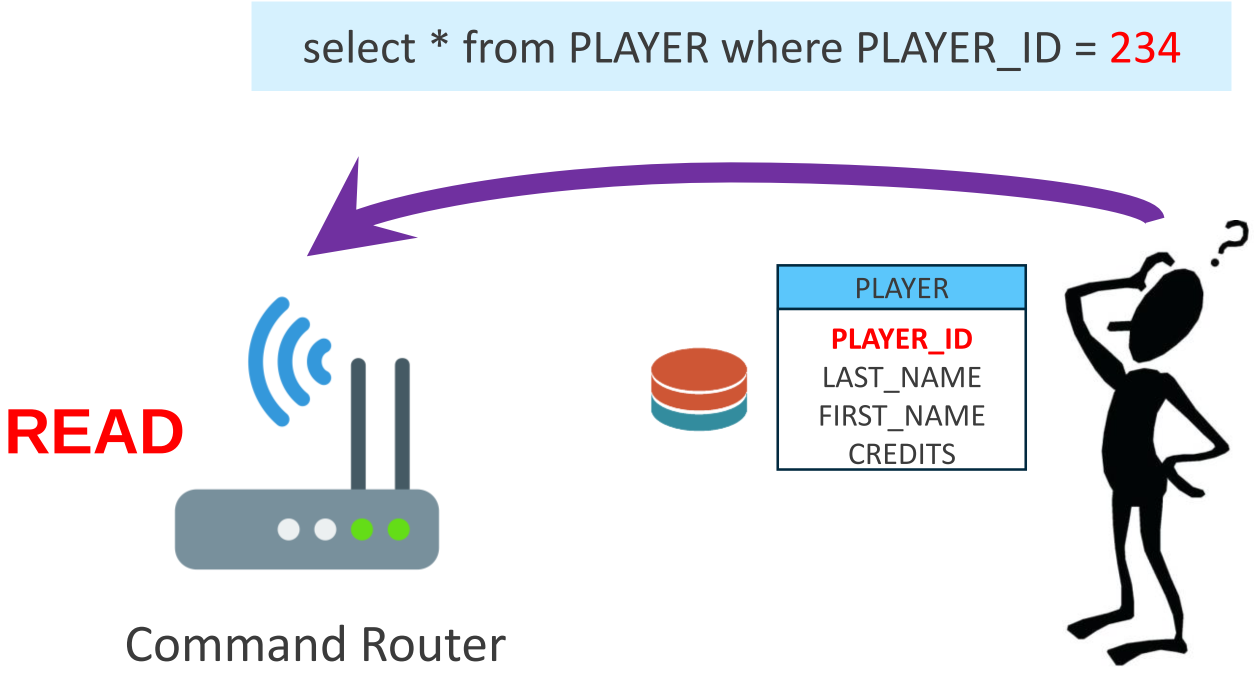


Command Router



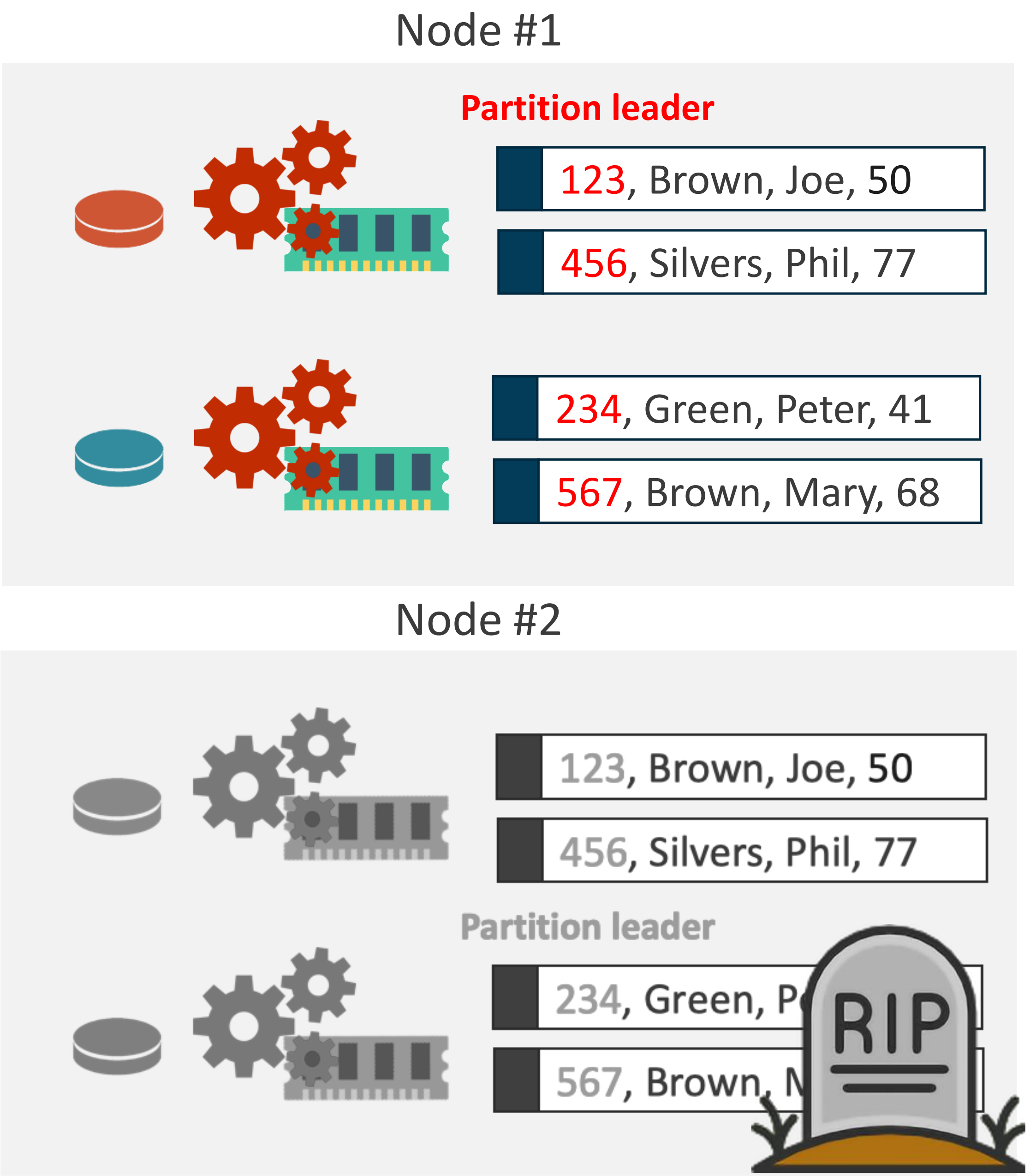
Durability through replication

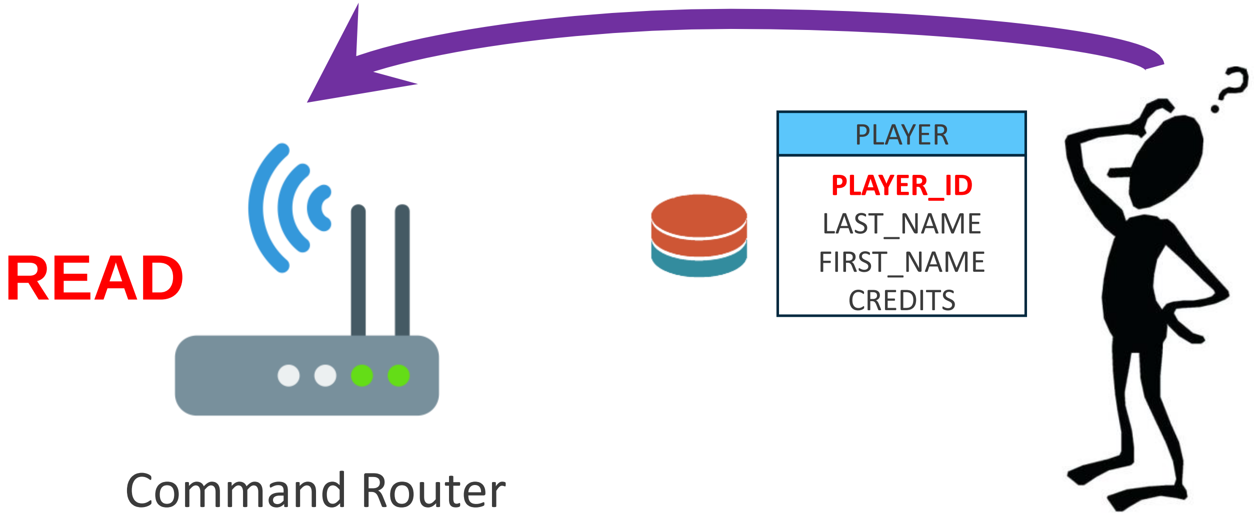




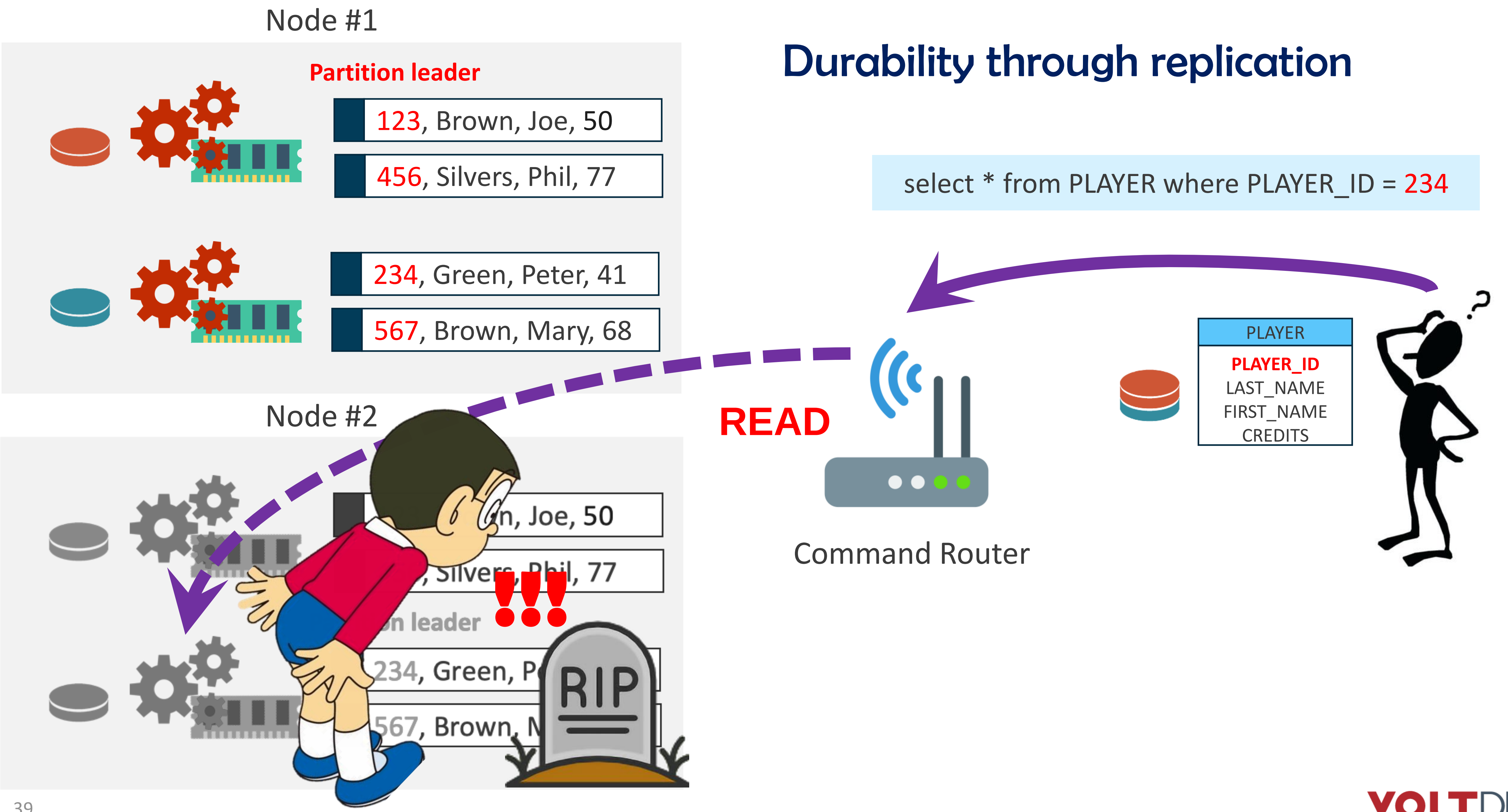
Durability through replication

select * from PLAYER where PLAYER_ID = 234

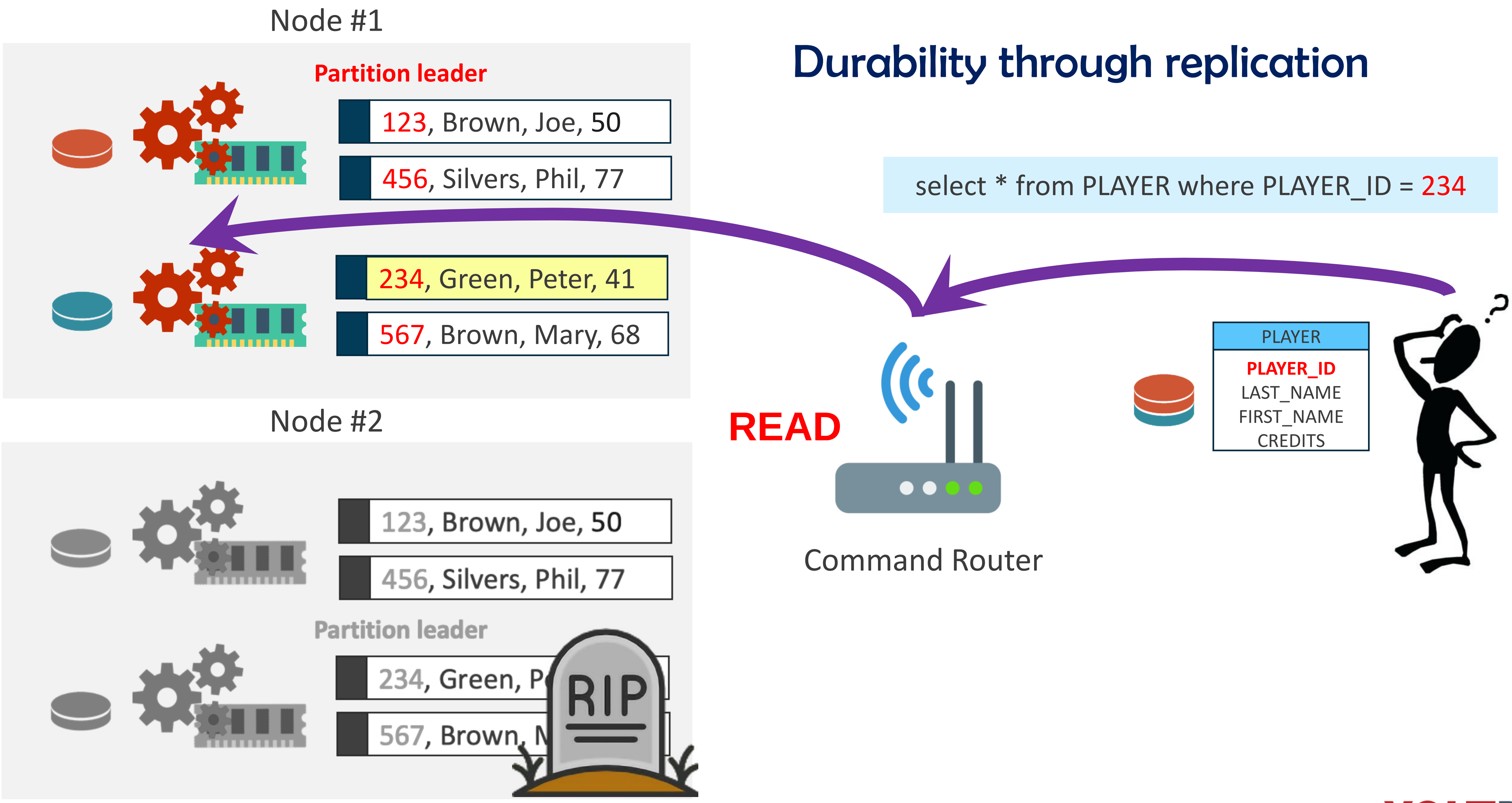




Durability through replication

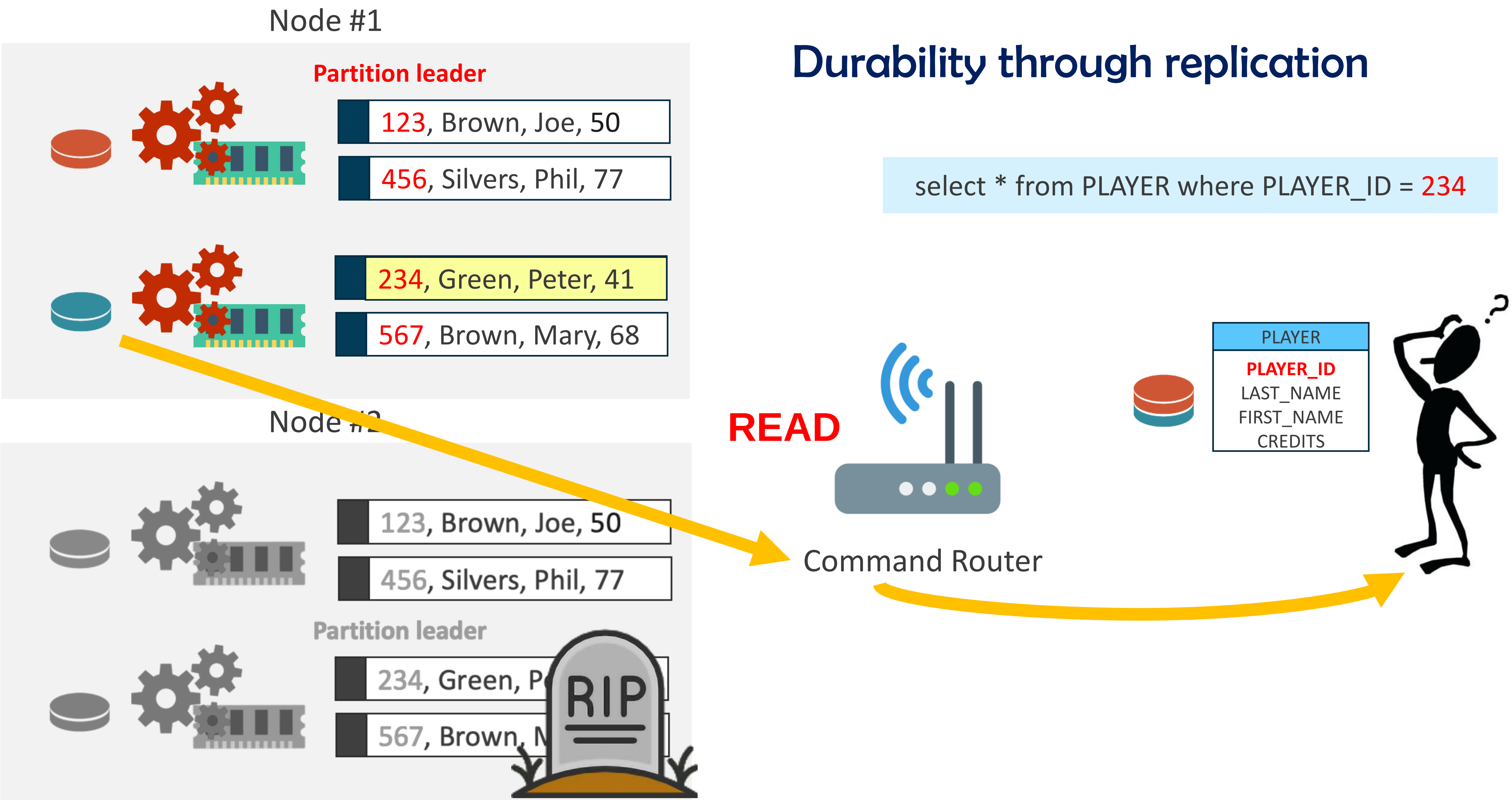


Durability through replication



VOLTDB

Durability through replication



VOLTDB

ACTIVE VS. PASSIVE

Approach #1: Active-Active

→ A txn executes at each replica independently.

→ Need to check at the end whether the txn ends up with the same result at each replica.

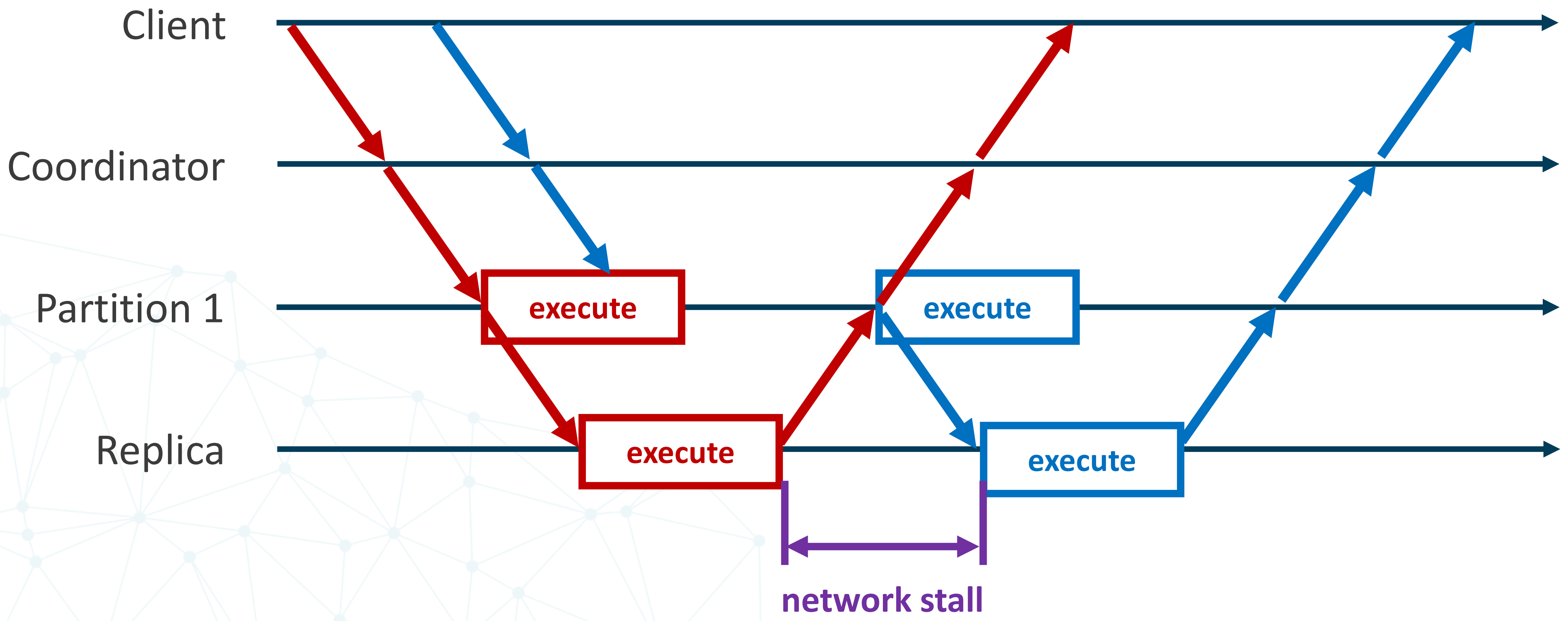
Approach #2: Active-Passive

→ Each txn executes at a single location and propagates the changes to the replica.

→ Not the same as master-replica vs. multi-master

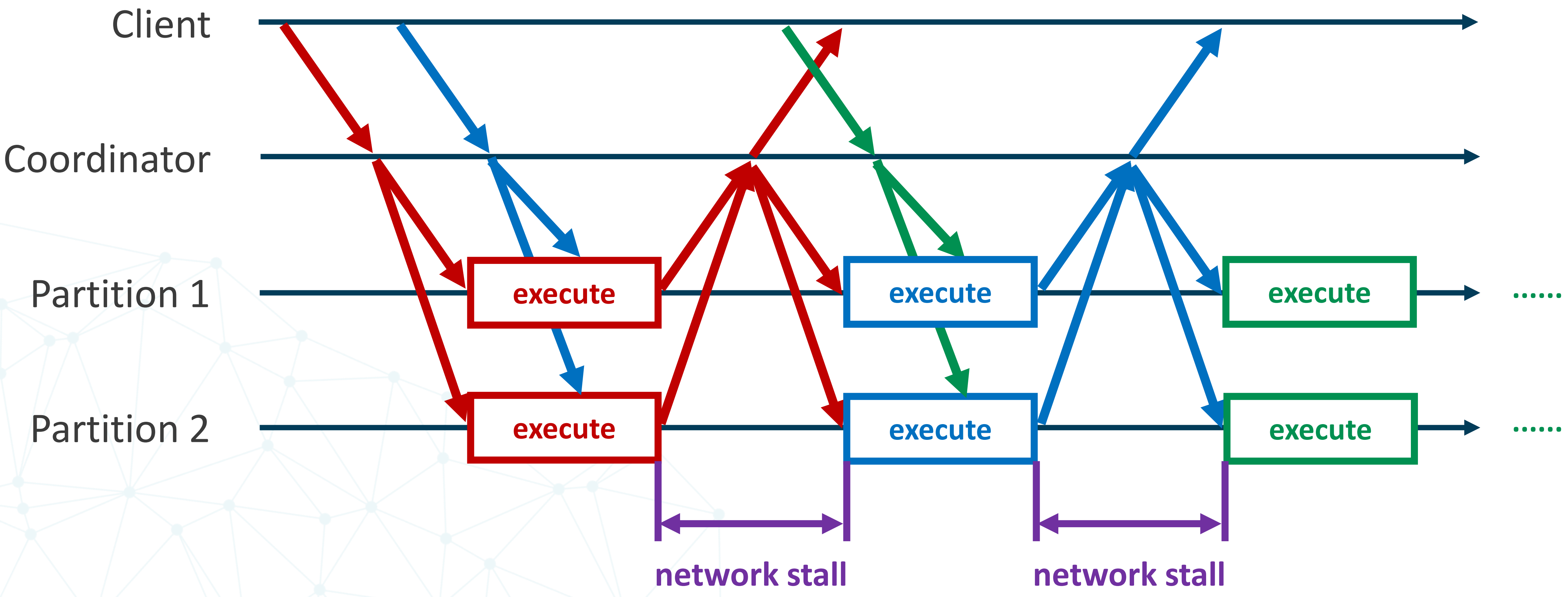
https://15445.courses.cs.cmu.edu/fall2018/slides/23-distributedoltp2.pdf

Active-Active Replication



VOLTDB

Recall that for the Multi Partition case…



VOLTDB

SP + Replication as bad as MP?

SP + Replication (K-safety) blocks K + 1 partitions

still has parallelism

MP blocks **ALL** partitions

NO parallelism

Determinism in Active-Active Replication

- Running the same transaction against several replicas.
- How do you ensure they end up with the same result?

Query Order

```
CREATE TABLE t (val INT);
```

val

```
CREATE TABLE t (val INT);
```

val

Query Order

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
```

val
1

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
```

val
1

Query Order

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
```

val
1
2

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
UPDATE t SET val = val * 10;
```

val
10

Query Order

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
UPDATE t SET val = val * 10;
```

val
10
20

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
UPDATE t SET val = val * 10;
INSERT INTO t VALUES (2);
```

val
10
2

Tuple Order

```
CREATE TABLE t (val INT);
```

val

```
CREATE TABLE t (val INT);
```

val

Tuple Order

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
```

val
1

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
```

val
1

Tuple Order

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
```

val
1
2

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
```

val
2
1

Tuple Order

```
DELETE FROM t LIMIT 1 ORDER BY val;
```

```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
DELETE FROM t LIMIT 1;
```

val
2

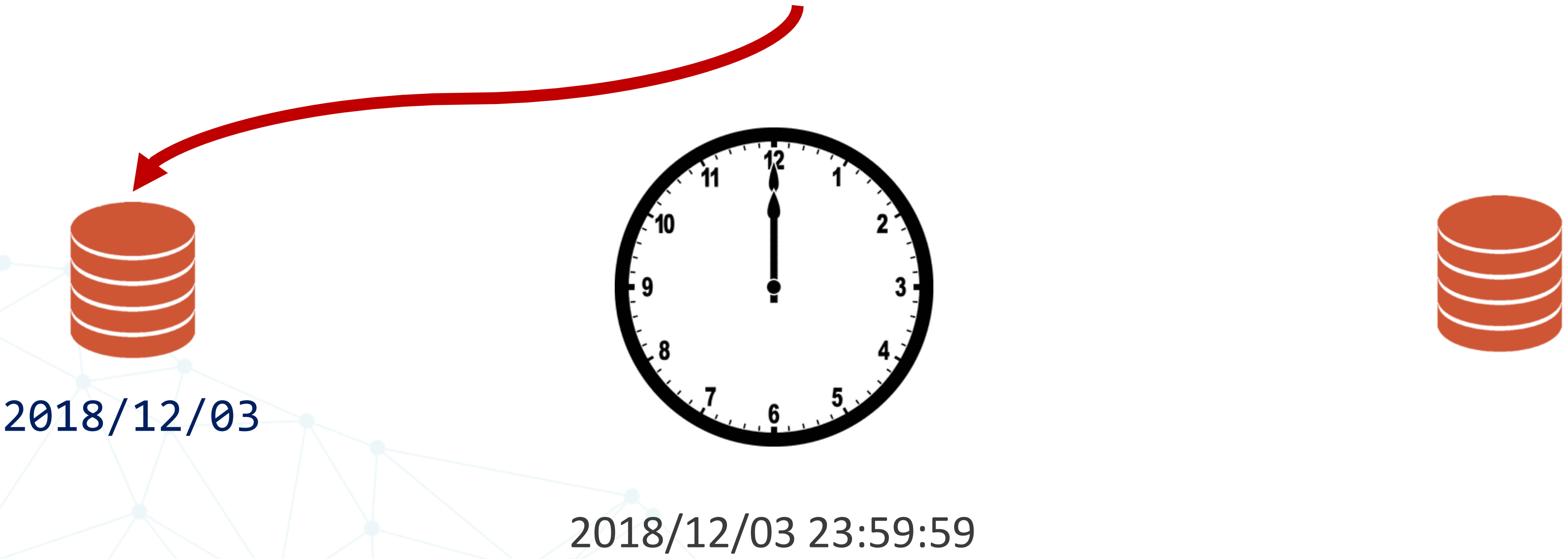
```
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
DELETE FROM t LIMIT 1;
```

val
1

VOLTDB

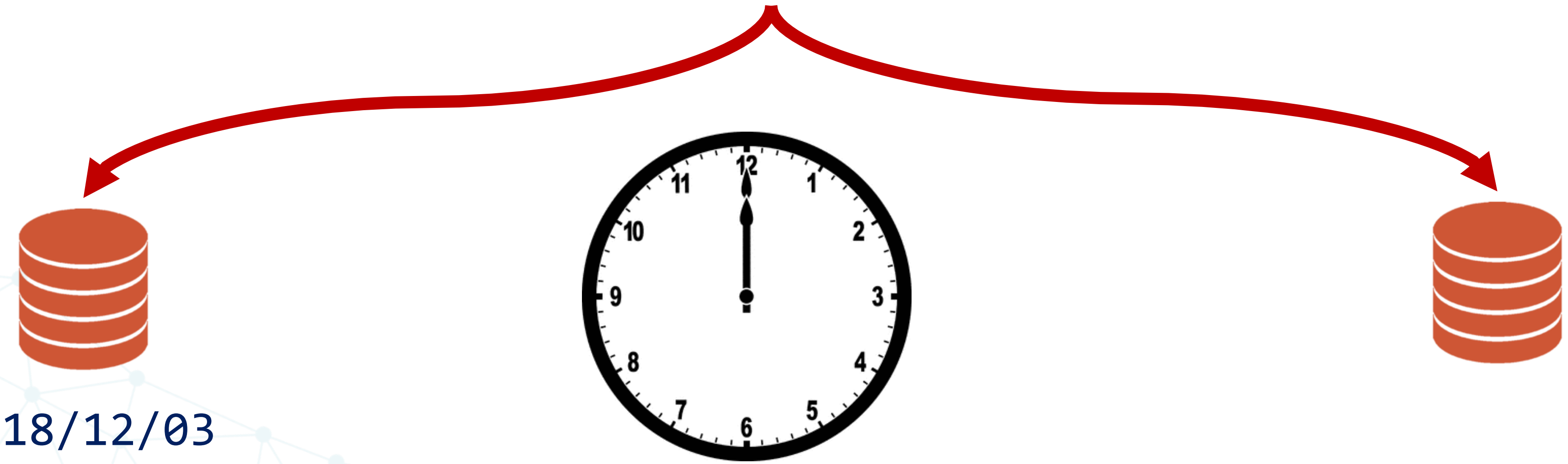
Function Determinism

```
INSERT INTO t VALUES ( TODAY() );
```



Function Determinism

```
INSERT INTO t VALUES ( TODAY() );
```



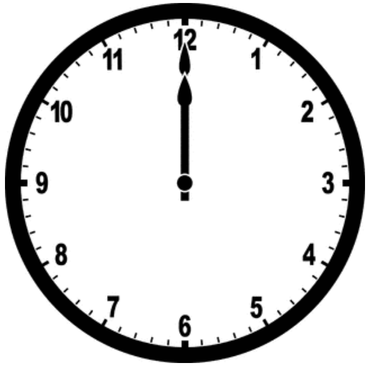
2018/12/03

2018/12/04 00:00:00

Function Determinism

```
INSERT INTO t VALUES ( '2018/12/03' );
INSERT INTO t VALUES ( TODAY() );
```

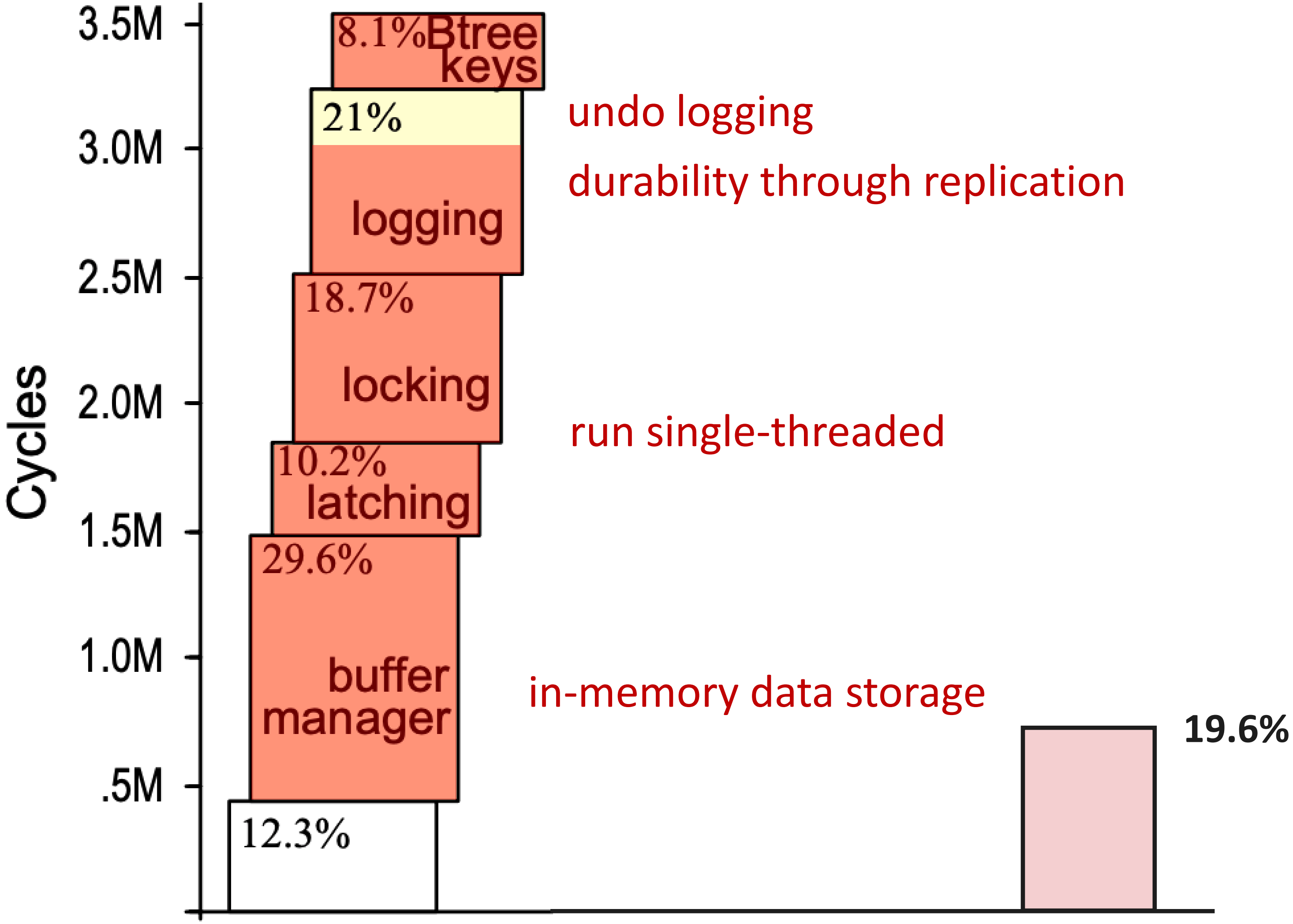
2018/12/03



2018/12/04 00:00:00

2018/12/04

VOLTDB



undo logging

durability through replication

run single-threaded

in-memory data storage

What did we have to change? – except logos

#1 Disk-based durability

- No one had any interest whatsoever in in-memory-only OLTP.


Reality
Theory

Durability - Command Logging

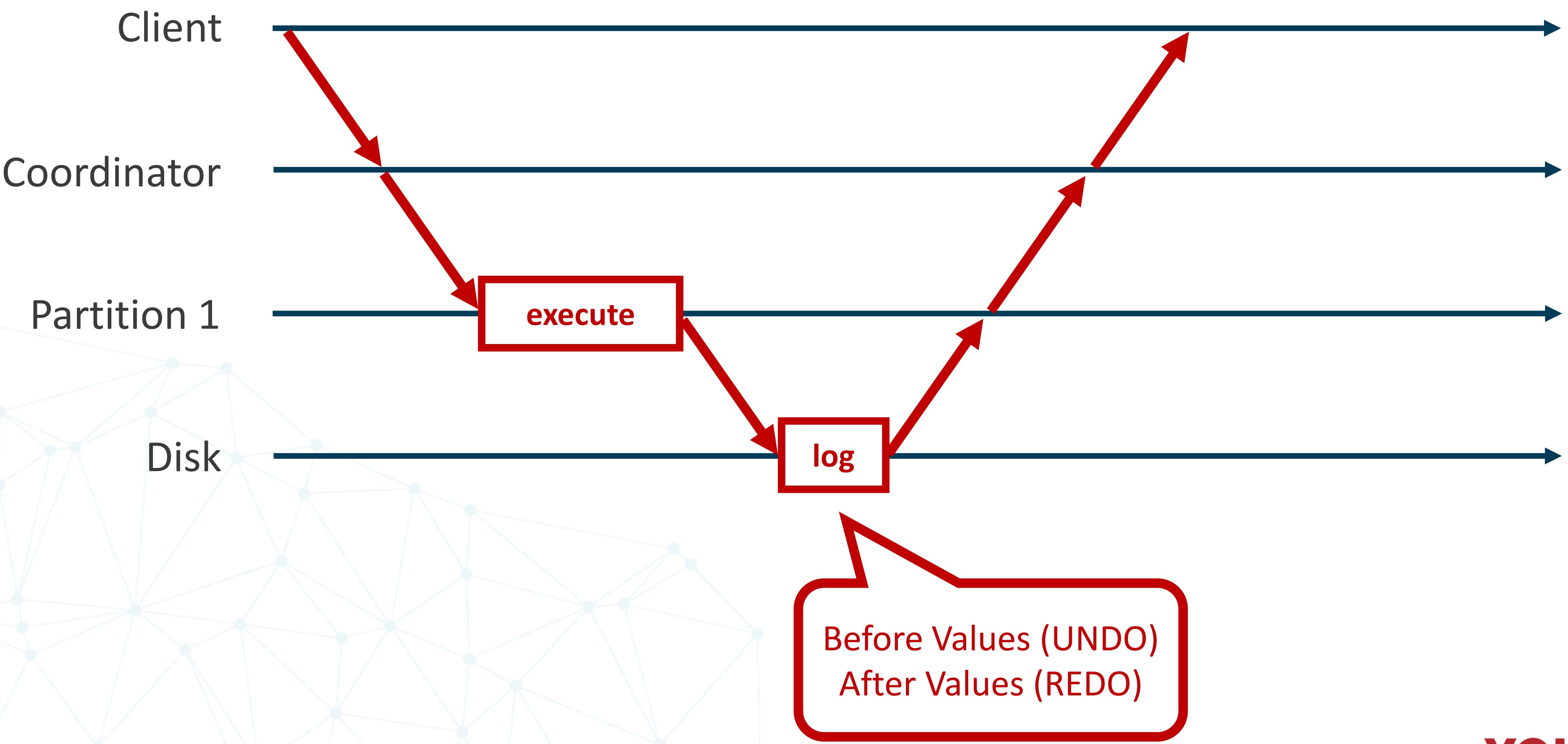
- Deterministic, Serializable operations written to the command log on disk.
- Replay operations on the same starting state in the fixed order reproduces the same ending state.
- **Serializable Isolation:** a performance trick, rather than a performance compromise.

Why log the command?

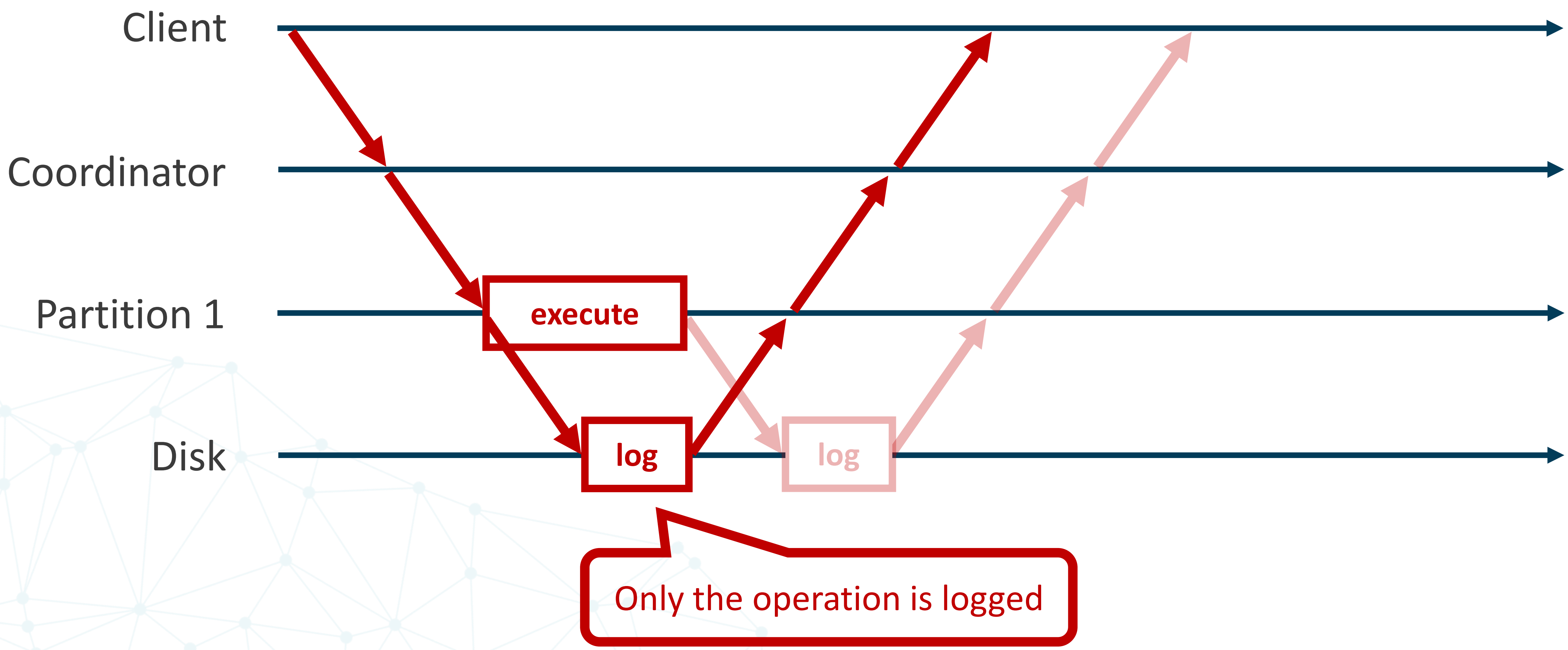
- Bounded Size - throughput
- Latency

VOLTDB

Write-Ahead Logging

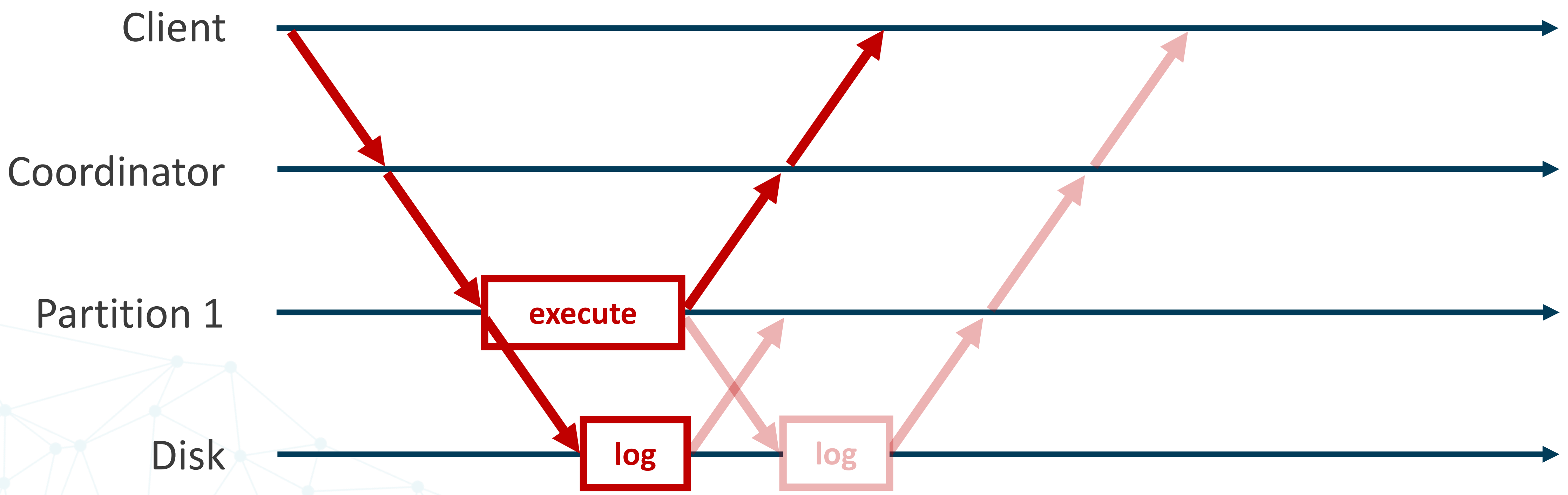


Command Logging (Sync)



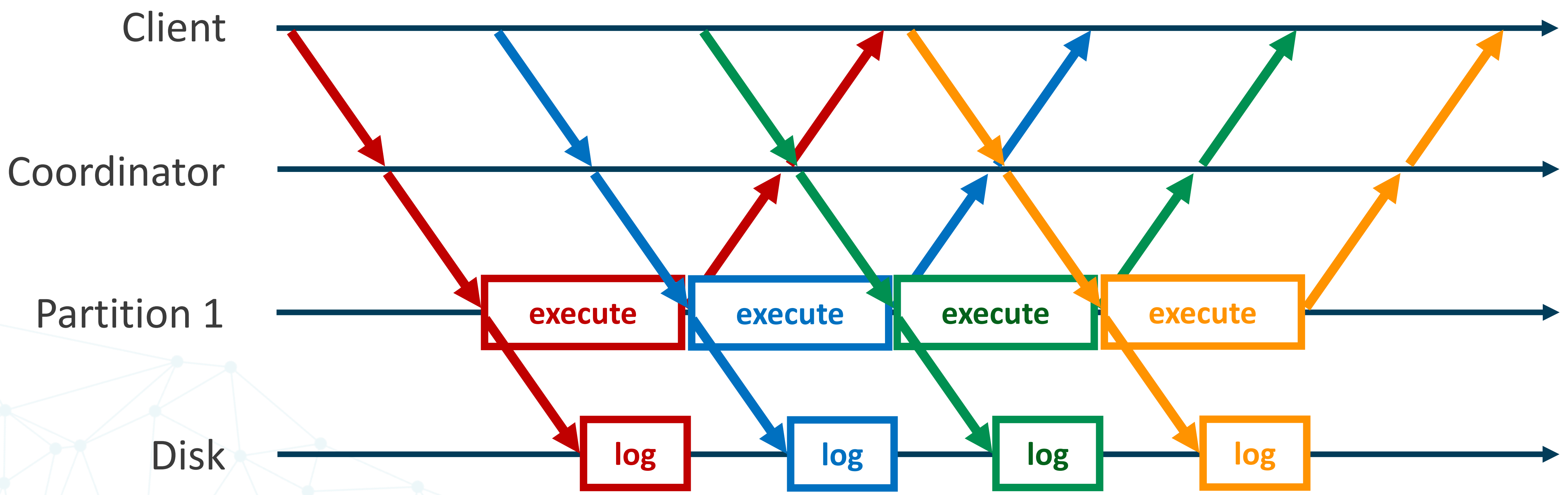
VOLTDB

Command Logging (Async)



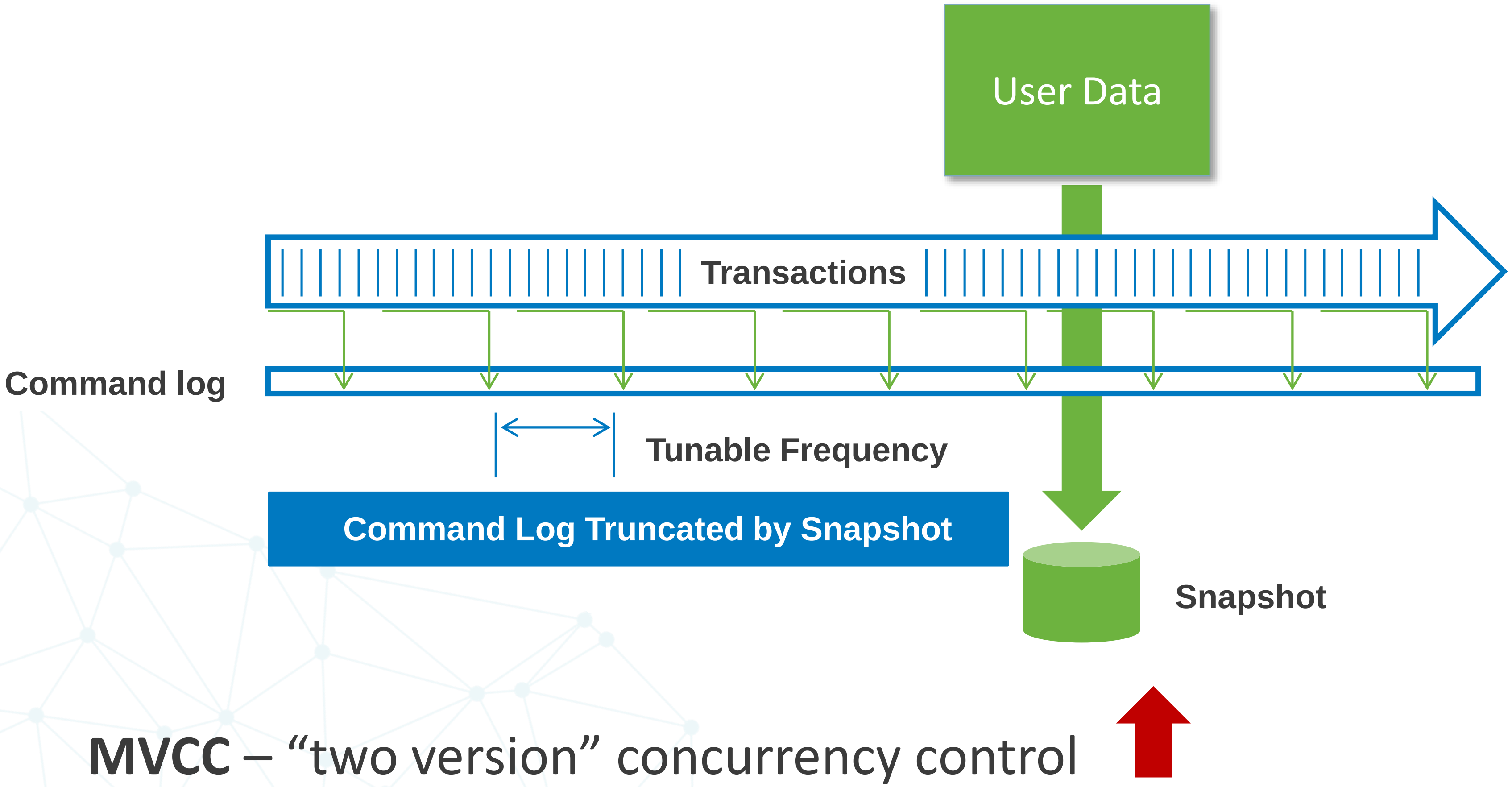
VOLTDB

Command Logging (Async)



Back Pressure mechanism to make sure the command log does not fall too far behind.

Checkpoint Snapshot

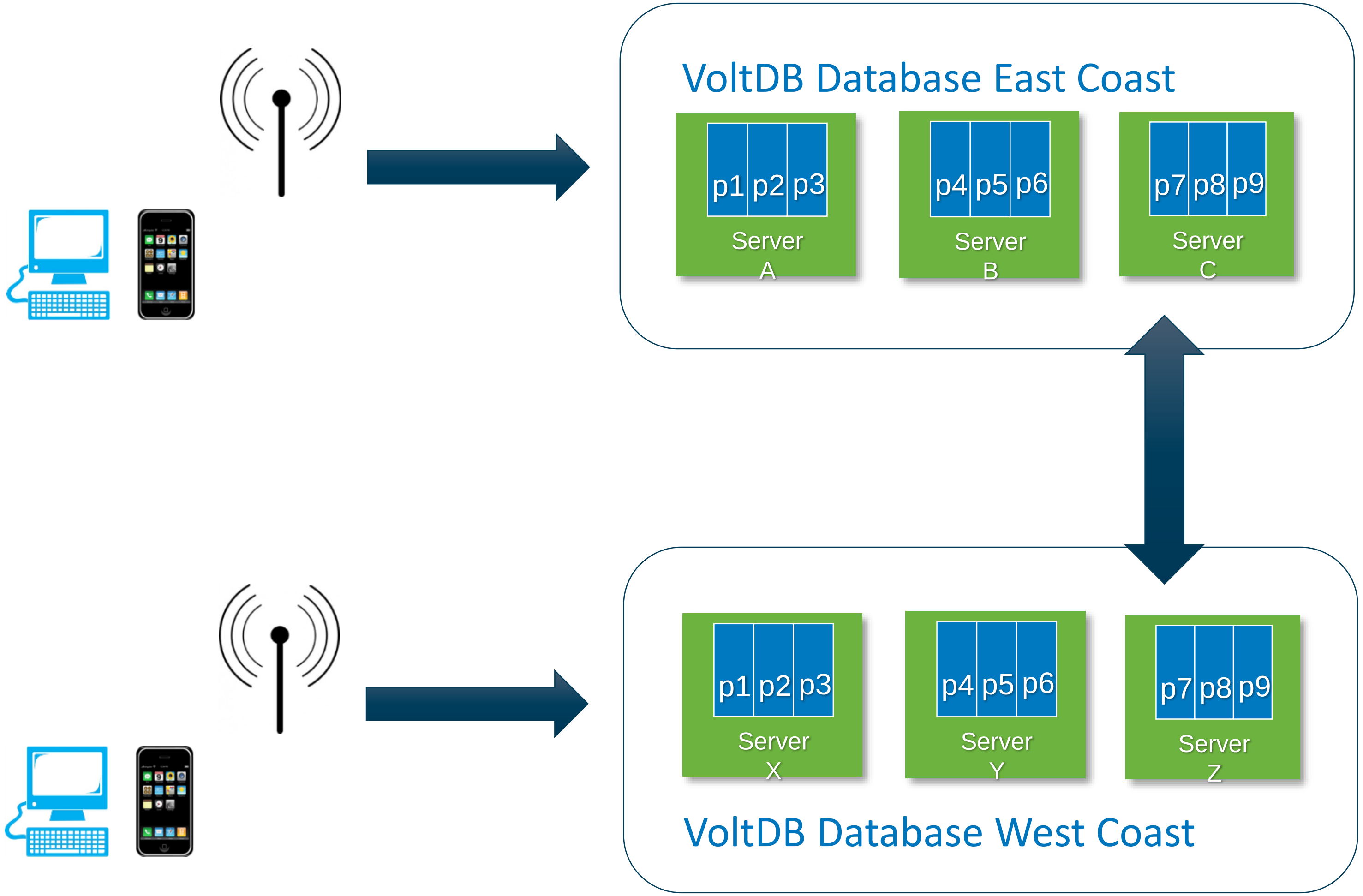


MVCC – "two version" concurrency control

VOLTDB

#2 Cross Datacenter Replication

- Durability
- Geographically Dispersed Datacenters
- Active-Passive and Active-Active



- Active-Active Geo Datacenter Replication
- Asynchronous Replication
- Conflict Detection
- Different Cluster Topologies

#3 Memory Fragmentation

- Long running clusters used more memory
- Memory usage doesn't shrink after data deletion

VOLTDB

Bucketing and Compaction

Tuple Storage

20% full

40% full

60% full

80% full

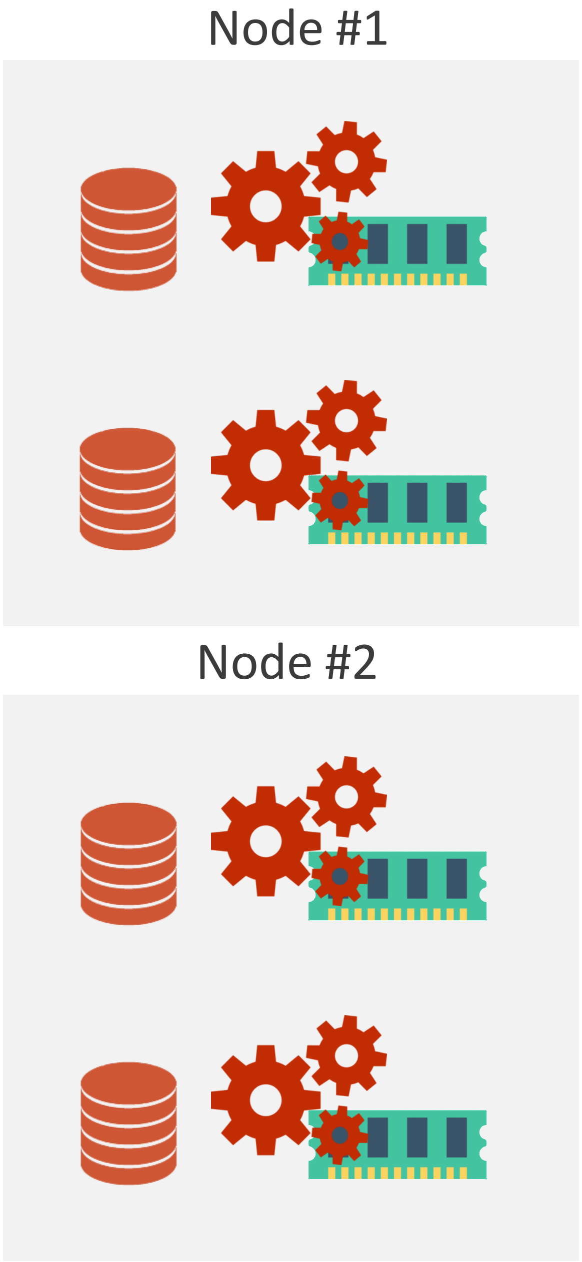
Index Swap the node for deletion with something at the end of the allocated storage, fixing links up when needed.

VOLTDB

#4 Shared Replicated Table

- Space efficiency
- Engine Complexity

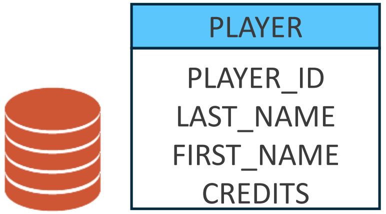
VOLTDB



Replicated table

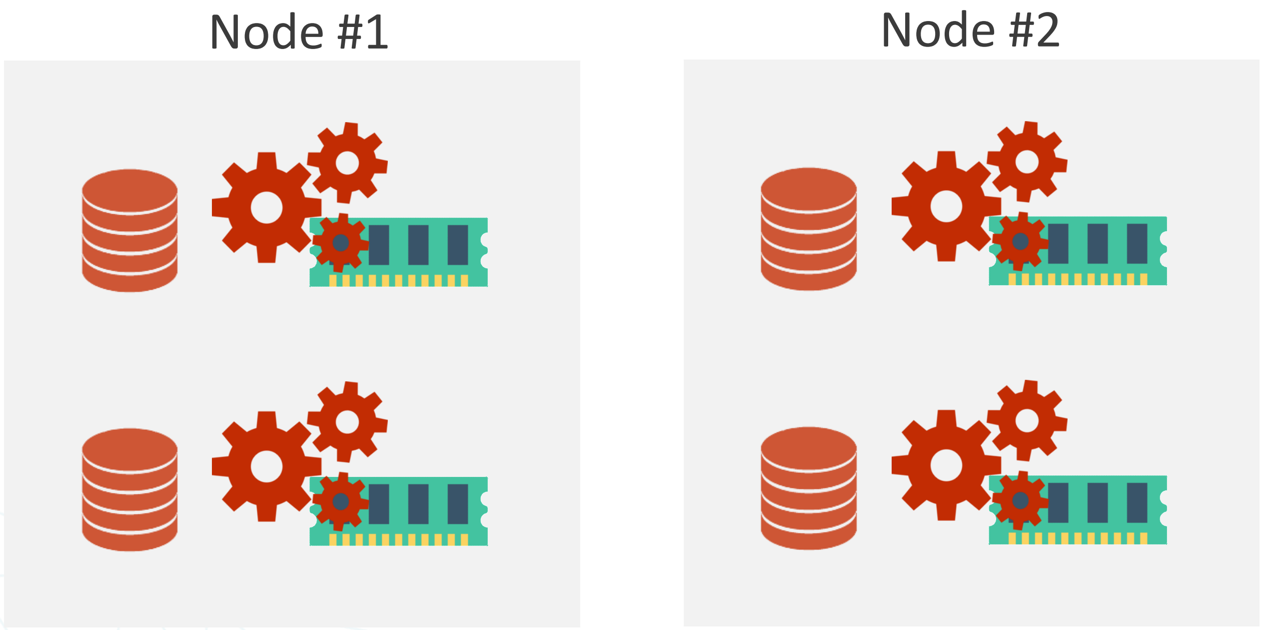
A cluster configuration from a customer:

- 48 CPU cores (sites)
- 512 GB RAM
- 10Gbps ethernet
- 6 nodes
- k-safety = 1



A 100 MB replicated table takes
100 x 48 x 6 = 28,800 MB

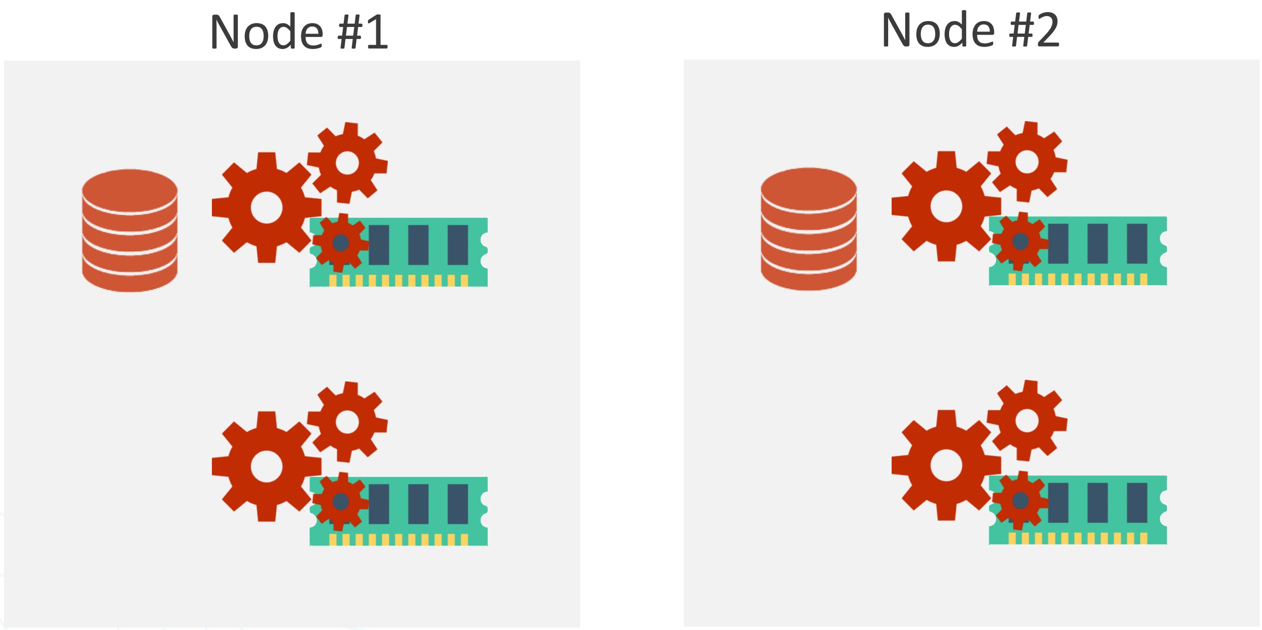
SRT saved significant memory space



A 100 MB replicated table takes
100 x 48 x 6 = 28,800 MB

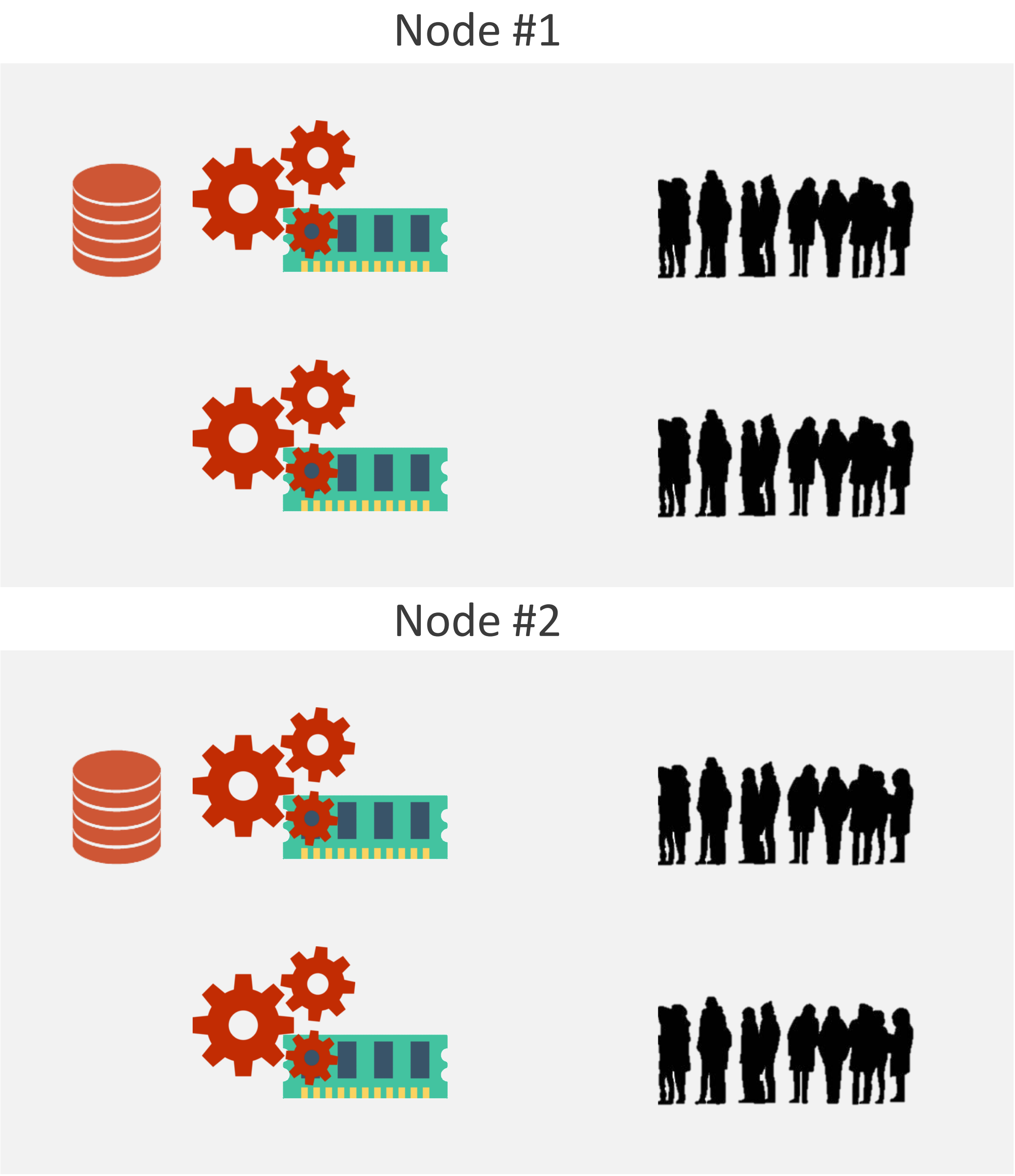
VOLTDB

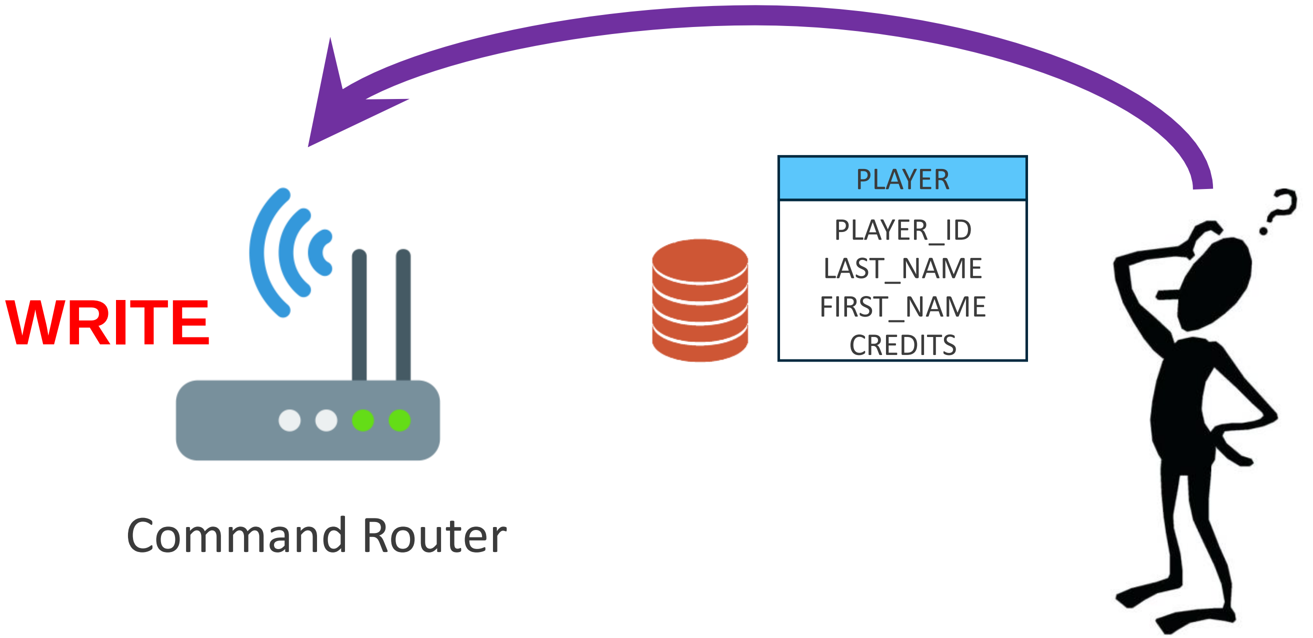
SRT saved significant memory space



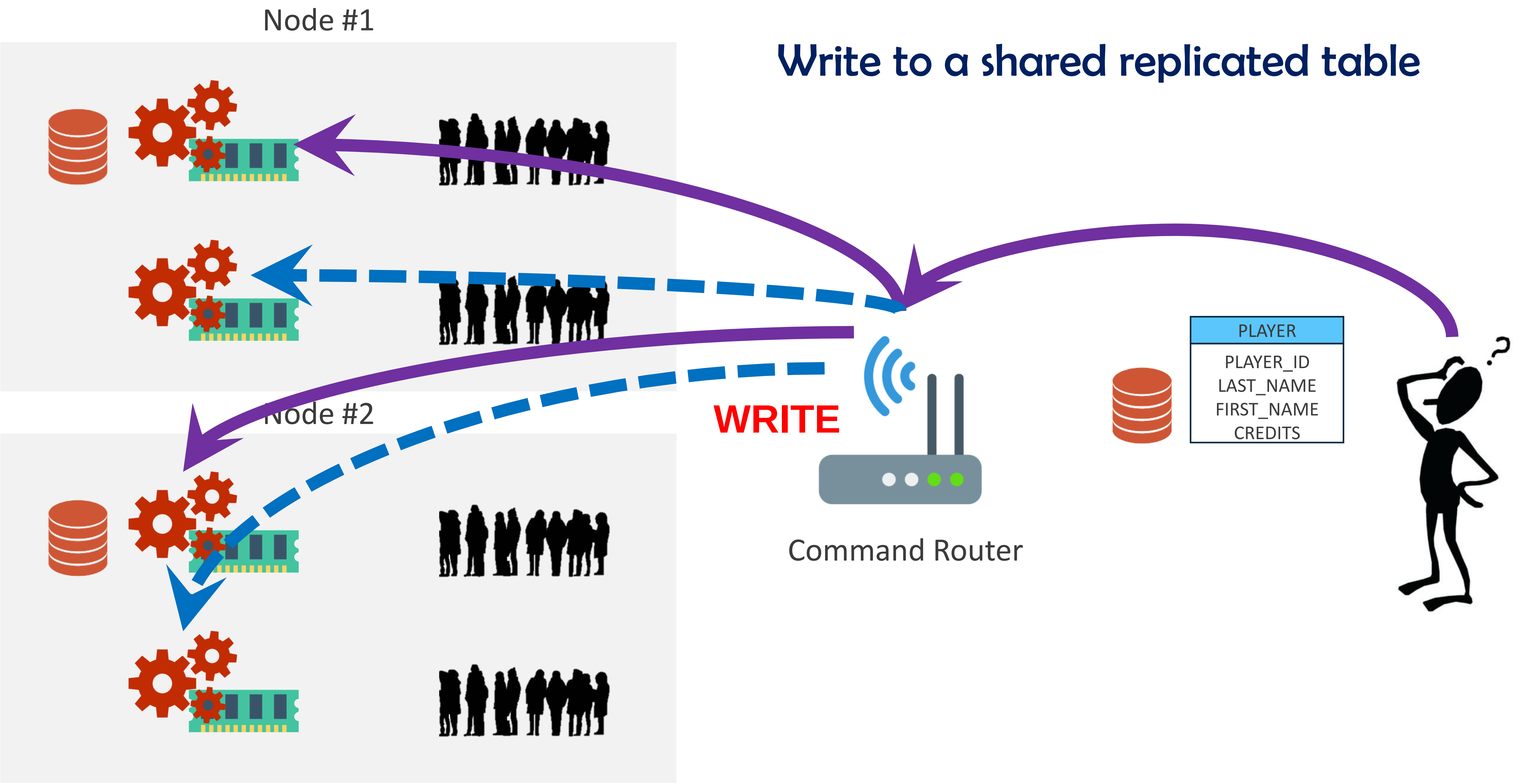
A 100 MB replicated table takes
100 x 6 = 600 MB

Write to a shared replicated table

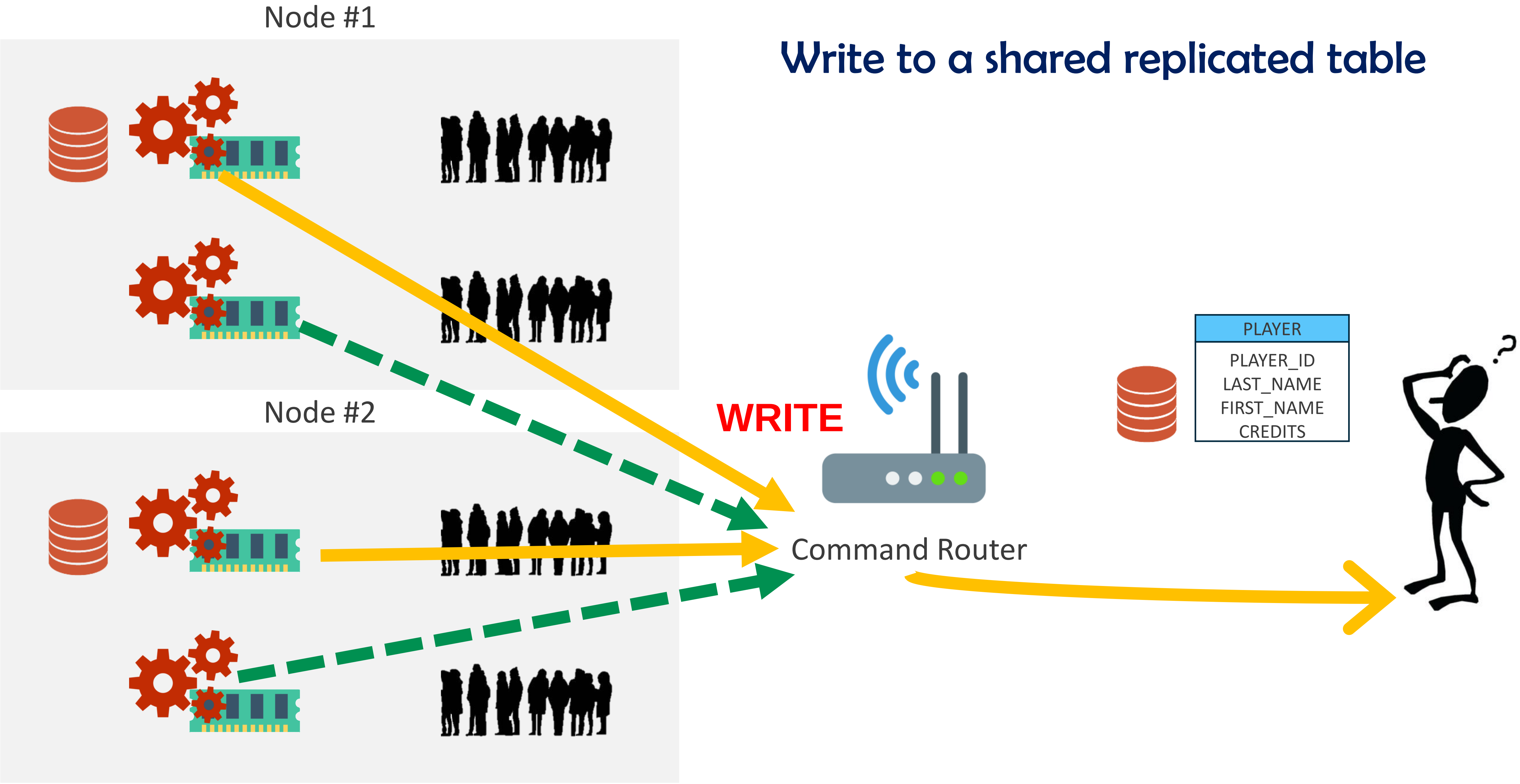




Write to a shared replicated table



Write to a shared replicated table

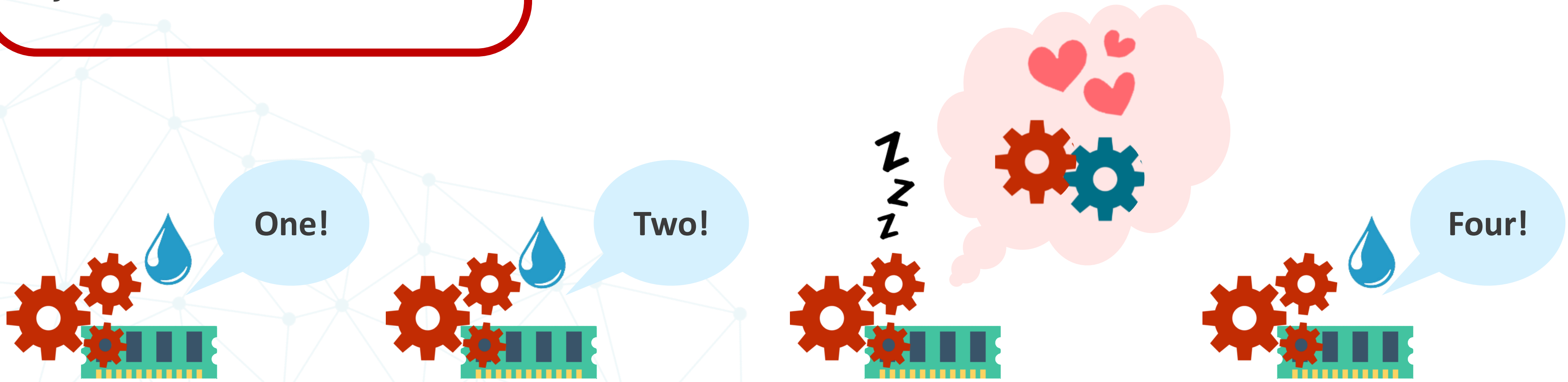


Latches in the execution engine

```
latch.countDown();
if (isLowestSite()) {
    latch.await();
    doWrite();
}
```

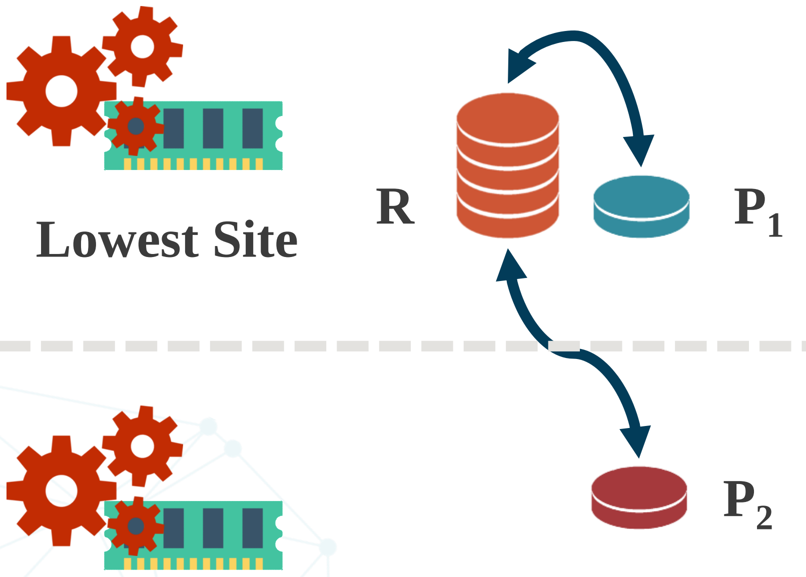
DEADLOCK

- Current transaction cannot finish
- Next transaction cannot begin



Engine Memory Context Switch

Partitioned Table P join Replicated Table R:



VOLTDB

#5 Materialized Views

- One of things that enables the streaming power in VoltDB.

VOLTDB

```
SELECT c1, COUNT(*), SUM(c2+c3) FROM T WHERE ...
```

Without Materialized Views:

NETWORKING	TXN OVERHEAD	ADD TUPLE IN MEM	x 500K/s
NETWORKING	TXN OVERHEAD	→ QUERY DASHBOARD →	x 1K/s

With Materialized Views:

NETWORKING	TXN OVERHEAD	ADD TUPLE IN MEM	UPDATE VIEW	x 500K/s
NETWORKING	TXN OVERHEAD	QUERY VIEW	x 1K/s	

#6 Importer/Exporters

- When you process transactions at extremely high velocity, the problem starts to look like stream processing a little bit.

Summary: AT HIGH VELOCITY

- Nobody wants black-box state. Real-time understanding has value.
- OLTP apps smell like stream processing apps.
- Processing and state management go well together.
- Adding features to a fast/stateful core is easier than reinventing wheels.

VOLTDB

#7 More SQL

- User-Defined Functions
- Common Table Expressions
- Better planning via Calcite (In Progress)
- and more…

Things that were changed

- Disk-based Durability
- Cross Datacenter Replication
- Memory Fragmentation
- Shared Replicated Tables
- Materialized Views
- importers and Exporters
- More SQL

New Research Directions

- Stream Processing capabilities - S-Store
- Larger-than-memory data management
- Improve Multi Partition Transaction Performance

H-Store -> S-Store: Stream Processing

S-Store: A Streaming NewSQL System for Big Velocity Applications

Ugur Cetintemel[1], Jiang Du[2], Tim Kraska[1], Samuel Madden[3], David Maier[4], John Meehan[1], Andrew Pavlo[5], Michael Stonebraker[3], Erik Sutherland[4], Nesime Tatbul[2,3], Kristin Tufte[4], Hao Wang[3], Stanley Zdonik[1]

[1]Brown University [2]Intel Labs [3]MIT [4]Portland State University [5]CMU

ABSTRACT

First-generation streaming systems did not pay much attention to state management via ACID transactions (e.g., [3, 4]). S-Store is a data management system that combines OLTP transactions with stream processing. To create S-Store, we begin with H-Store, a main-memory transaction processing engine, and add primitives to support streaming. This includes triggers and transaction workflows to implement push-based processing, windows to provide a way to bound the computation, and tables with hidden state to implement scoping for proper isolation. This demo explores the benefits of this approach by showing how a naïve implementation of our benchmarks using only H-Store can yield incorrect results. We also show that by exploiting push-based semantics and our implementation of triggers, we can achieve significant improvement in transaction throughput. We demo two modern applications: (i) leaderboard maintenance for a version of "American Idol", and (ii) a city-scale bicycle rental scenario.

1. INTRODUCTION

Managing high-speed data streams generated in real time is an integral part of today's big data applications. In a wide range of domains from social media to financial trading, there is a growing need to seamlessly support incremental processing as new data is generated. At the same time, the system must ingest some or all of this data into a persistent store for on-demand transaction or analytical processing. In particular, this is true for applications in which high-velocity data updates a large persistent state such as leaderboard maintenance or online advertising. In these situations, there is need to manage streaming and non-streaming state side by side in a way that ensures transactional integrity and performance.

Today's stream processing systems lack the required transactional robustness, while OLTP databases do not provide native support for data-driven processing. In this work, our goal is to build a single, scalable system that can support both stream and transaction processing at the same time. We believe that modern distributed main-memory OLTP platforms, also known as *NewSQL* systems [5], provide a suitable foundation for building such a system, since (i) they are more lightweight than their traditional disk-based counterparts, (ii) like streaming engines, they offer lower latency via in-memory processing; and (iii) they provide strong support for state and transaction management. Thus, we introduce S-Store, a streaming OLTP system that realizes our goal by extending the H-Store DBMS [6] with streams.

We propose to demonstrate the S-Store streaming NewSQL system and several of its novel features that include:

Architecture: S-Store makes a number of fundamental architectural extensions to H-Store that generally apply to making any main-memory OLTP system stream-capable. Thus, the first goal of this demonstration is to highlight our architectural contributions.

Transaction Model: S-Store inherits H-Store's ACID transaction model and makes several critical extensions to it. The streaming nature of the data requires dependencies between transactions, and S-Store provides ACID guarantees in the presence of these dependencies. We will show how our extended model ensures transactional integrity.

Performance: S-Store's native support for streams not only makes application development easier and less error-prone, but also boosts performance by removing the need to poll for new data and by reducing the number of round-trips across various layers of the system. We demo these features by comparing H-Store and S-Store.

Applications: S-Store can support a wide spectrum of applications that require transactional processing over both streaming and non-streaming data. The demo will present a select set of these applications, highlighting different technical features of the system as well as its support for diverse workloads.

The rest of this paper provides an overview of the S-Store system and the details of our demo scenarios.

2. S-STORE SYSTEM OVERVIEW

S-Store belongs to a new breed of stream processing systems designed for high-throughput, scalable, and fault-tolerant processing over big and fast data across large clusters. Like its contemporaries such as Twitter Storm/Trident [9] or Spark Streaming [10], S-Store supports complex computational workflows over streaming and non-streaming data sets. S-Store is unique in that all data access in S-Store is SQL-based and fully transactional.

S-Store builds on the H-Store NewSQL system [6]. H-Store is a high-performance, in-memory, distributed OLTP system designed for shared-nothing clusters. It targets OLTP workloads with short-lived transactions, which are pre-defined as parameterized stored procedures (i.e., SQL queries embedded in Java-based control code) that are invoked by client requests at run time. As with most distributed database systems, a good H-Store design partitions the database in a way that processes most of the transactions in a single-sited manner, minimizing the number of distributed transactions and reducing the overhead of coordination across multiple partitions [8].

Proceedings of the VLDB Endowment, Vol. 7, No. 13

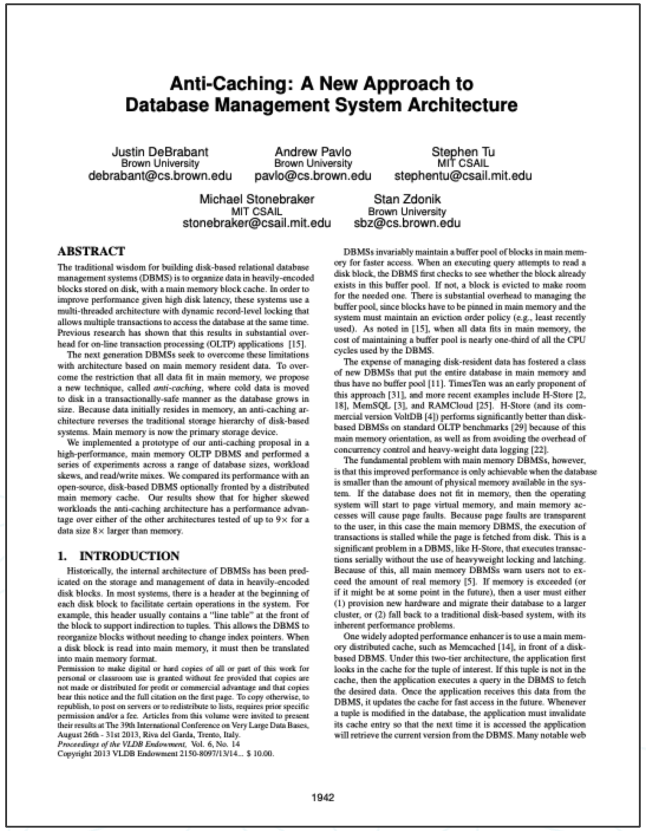
1633

- New constructs for streams:
  - Window: finite chunks of state over (possibly unbounded) streams.
  - Trigger: computations to be invoked for newly generated data.
  - Workflow: computation pipelines of dependent transactions.
- Tuple TTL (Time-To-Live) – **VoltDB 8.2**

Cetintemel, Ugur, et al. "S-Store: a streaming NewSQL system for big velocity applications." Proceedings of the VLDB Endowment 7.13 (2014): 1633-1636.

VOLTDB

Larger than memory data management



Anti-Caching: A New Approach to Database Management System Architecture

Justin DeBrabant
Brown University
debrabant@cs.brown.edu

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Stephen Tu
MIT CSAIL
stephentu@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

The traditional wisdom for building disk-based relational database management systems (DBMS) is to organize data in heavily-encoded blocks stored on disk, with a main memory block cache. In order to improve performance given high disk latency, these systems use a multi-threaded architecture with dynamic record-level locking that allows multiple transactions to access the database at the same time. Previous research has shown that this results in substantial overhead for on-line transaction processing (OLTP) applications [15].

The next generation DBMSs seek to overcome these limitations with architecture based on main memory resident data. To overcome the restriction that all data fit in main memory, we propose a new technique, called *anti-caching*, where cold data is moved to disk in a transactionally-safe manner as the database grows in size. Because data initially resides in memory, an anti-caching architecture reverses the traditional storage hierarchy of disk-based systems. Main memory is now the primary storage device.

We implemented a prototype of our anti-caching proposal in a high-performance, main memory OLTP DBMS and performed a series of experiments across a range of database sizes, workload skews, and read/write mixes. We compared its performance with an open-source, disk-based DBMS optionally fronted by a distributed main memory cache. Our results show that for higher skewed workloads the anti-caching architecture has a performance advantage over either of the other architectures tested of up to 9× for a data size 8× larger than memory.

1. INTRODUCTION

Historically, the internal architecture of DBMSs has been predicated on the storage and management of data in heavily-encoded disk blocks. In most systems, there is a header at the beginning of each disk block to facilitate certain operations in the system. For example, this header usually contains a "line table" at the front of the block to support indirection to tuples. This allows the DBMS to reorganize blocks without needing to change index pointers. When a disk block is read into main memory, it must then be translated into main memory format.

Proceedings of the VLDB Endowment, Vol. 6, No. 14

DBMSs invariably maintain a buffer pool of blocks in main memory for faster access. When an executing query attempts to read a disk block, the DBMS first checks to see whether the block already exists in this buffer pool. If not, a block is evicted to make room for the needed one. There is substantial overhead to managing the buffer pool, since blocks have to be pinned in main memory and the system must maintain an eviction order policy (e.g., least recently used). As noted in [15], when all data fits in main memory, the cost of maintaining a buffer pool is nearly one-third of all the CPU cycles used by the DBMS.

The expense of managing disk-resident data has fostered a class of new DBMSs that put the entire database in main memory and thus have no buffer pool [11]. TimesTen was an early proponent of this approach [31], and more recent examples include H-Store [2, 18], MemSQL [3], and RAMCloud [25]. H-Store (and its commercial version VoltDB [4]) performs significantly better than disk-based DBMSs on standard OLTP benchmarks [29] because of this main memory orientation, as well as from avoiding the overhead of concurrency control and heavy-weight data logging [22].

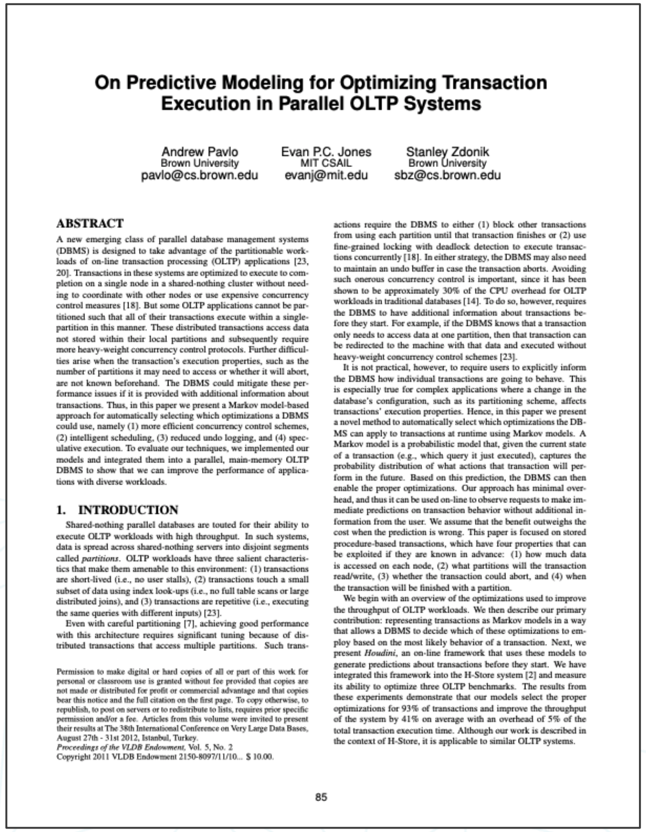
The fundamental problem with main memory DBMSs, however, is that this improved performance is only achievable when the database is smaller than the amount of physical memory available in the system. If the database does not fit in memory, then the operating system will start to page virtual memory, and main memory accesses will cause page faults. Because page faults are transparent to the user, in this case the main memory DBMS, the execution of transactions is stalled while the page is fetched from disk. This is a significant problem in a DBMS, like H-Store, that executes transactions serially without the use of heavyweight locking and latching. Because of this, all main memory DBMSs warn users not to exceed the amount of real memory [5]. If memory is exceeded (or if it might be at some point in the future), then a user must either (1) provision new hardware and migrate their database to a larger cluster, or (2) fall back to a traditional disk-based system, with its inherent performance problems.

One widely adopted performance enhancer is to use a main memory distributed cache, such as Memcached [14], in front of a disk-based DBMS. Under this two-tier architecture, the application first looks in the cache for the tuple of interest. If this tuple is not in the cache, then the application executes a query in the DBMS to fetch the desired data. Once the application receives this data from the DBMS, it updates the cache for fast access in the future. Whenever a tuple is modified in the database, the application must invalidate its cache entry so that the next time it is accessed the application will retrieve the current version from the DBMS. Many notable web

- More often than not, OLTP workloads have **hot** and **cold** portions of the database.
- General approach:
  - Identify cold tuples (online/offline)
  - Evict cold tuples to disk (when? track?)
  - Tuple retrieval (how? granularity?)
  - Tuple merge (when?)
- A lot of implementations:
  - H-Store, MemSQL,
    Hekaton (SQL Server In-Memory), etc.

DeBrabant, Justin, et al. "Anti-caching: A new approach to database management system architecture."
Proceedings of the VLDB Endowment 6.14 (2013): 1942-1953.

Smarter Scheduling



On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Evan P.C. Jones
MIT CSAIL
evanj@mit.edu

Stanley Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

A new emerging class of parallel database management systems (DBMS) is designed to take advantage of the partitionable workloads of on-line transaction processing (OLTP) applications [23, 20]. Transactions in these systems are optimized to execute to completion on a single node in a shared-nothing cluster without needing to coordinate with other nodes or use expensive concurrency control measures [18]. But some OLTP applications cannot be partitioned such that all of their transactions execute within a single-partition in this manner. These distributed transactions access data not stored within their local partitions and subsequently require more heavy-weight concurrency control protocols. Further difficulties arise when the transaction's execution properties, such as the number of partitions it may need to access or whether it will abort, are not known beforehand. The DBMS could mitigate these performance issues if it is provided with additional information about transactions. Thus, in this paper we present a Markov model-based approach for automatically selecting which optimizations a DBMS could use, namely (1) more efficient concurrency control schemes, (2) intelligent scheduling, (3) reduced undo logging, and (4) speculative execution. To evaluate our techniques, we implemented our models and integrated them into a parallel, main-memory OLTP DBMS to show that we can improve the performance of applications with diverse workloads.

1. INTRODUCTION

Shared-nothing parallel databases are touted for their ability to execute OLTP workloads with high throughput. In such systems, data is spread across shared-nothing servers into disjoint segments called *partitions*. OLTP workloads have three salient characteristics that make them amenable to this environment: (1) transactions are short-lived (i.e., no user stalls), (2) transactions touch a small subset of data using index look-ups (i.e., no full table scans or large distributed joins), and (3) transactions are repetitive (i.e., executing the same queries with different inputs) [23].

Even with careful partitioning [7], achieving good performance with this architecture requires significant tuning because of distributed transactions that access multiple partitions. Such transactions require the DBMS to either (1) block other transactions from using each partition until that transaction finishes or (2) use fine-grained locking with deadlock detection to execute transactions concurrently [18]. In either strategy, the DBMS may also need to maintain an undo buffer in case the transaction aborts. Avoiding such onerous concurrency control is important, since it has been shown to be approximately 30% of the CPU overhead for OLTP workloads in traditional databases [14]. To do so, however, requires the DBMS to have additional information about transactions before they start. For example, if the DBMS knows that a transaction only needs to access data at one partition, then that transaction can be redirected to the machine with that data and executed without heavy-weight concurrency control schemes [23].

It is not practical, however, to require users to explicitly inform the DBMS how individual transactions are going to behave. This is especially true for complex applications where a change in the database's configuration, such as its partitioning scheme, affects transactions' execution properties. Hence, in this paper we present a novel method to automatically select which optimizations the DBMS can apply to transactions at runtime using Markov models. A Markov model is a probabilistic model that, given the current state of a transaction (e.g., which query it just executed), captures the probability distribution of what actions that transaction will perform in the future. Based on this prediction, the DBMS can then enable the proper optimizations. Our approach has minimal overhead, and thus it can be used on-line to observe requests to make immediate predictions on transaction behavior without additional information from the user. We assume that the benefit outweighs the cost when the prediction is wrong. This paper is focused on stored procedure-based transactions, which have four properties that can be exploited if they are known in advance: (1) how much data is accessed on each node, (2) what partitions will the transaction read/write, (3) whether the transaction could abort, and (4) when the transaction will be finished with a partition.

We begin with an overview of the optimizations used to improve the throughput of OLTP workloads. We then describe our primary contribution: representing transactions as Markov models in a way that allows a DBMS to decide which of these optimizations to employ based on the most likely behavior of a transaction. Next, we present *Houdini*, an on-line framework that uses these models to generate predictions about transactions before they start. We have integrated this framework into the H-Store system [2] and measure its ability to optimize three OLTP benchmarks. The results from these experiments demonstrate that our models select the proper optimizations for 93% of transactions and improve the throughput of the system by 41% on average with an overhead of 5% of the total transaction execution time. Although our work is described in the context of H-Store, it is applicable to similar OLTP systems.

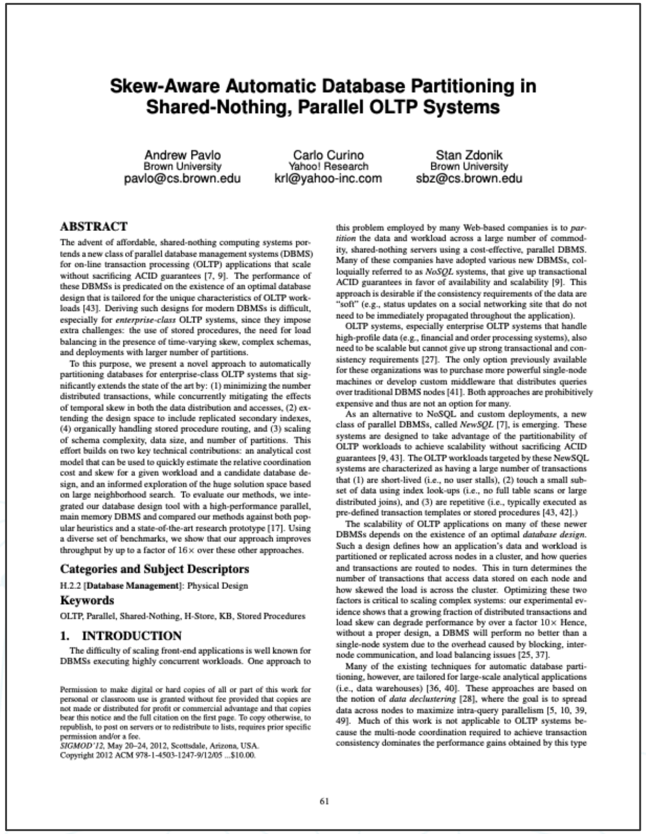
Proceedings of the VLDB Endowment, Vol. 5, No. 2

- Use data-heavy node as coordinator
  - reduces data movement
- N-Partition instead of All-Partition
- Disable undo logging when possible (SP only)
- Speculative concurrency control
  - Execute other transactions speculatively while waiting for commit/abort.
- Use Markov model for transaction behavior forecast.

Pavlo, Andrew, et al. "On predictive modeling for optimizing transaction execution in parallel OLTP systems."
Proceedings of the VLDB Endowment5.2 (2011): 85-96.

VOLTDB

Smarter Partitioning



Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Carlo Curino
Yahoo! Research
krl@yahoo-inc.com

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

The advent of affordable, shared-nothing computing systems portends a new class of parallel database management systems (DBMS) for on-line transaction processing (OLTP) applications that scale without sacrificing ACID guarantees [7, 9]. The performance of these DBMSs is predicated on the existence of an optimal database design that is tailored for the unique characteristics of OLTP workloads [43]. Deriving such designs for modern DBMSs is difficult, especially for *enterprise-class* OLTP systems, since they impose extra challenges: the use of stored procedures, the need for load balancing in the presence of time-varying skew, complex schemas, and deployments with larger number of partitions.

To this purpose, we present a novel approach to automatically partitioning databases for enterprise-class OLTP systems that significantly extends the state of the art by: (1) minimizing the number distributed transactions, while concurrently mitigating the effects of temporal skew in both the data distribution and accesses, (2) extending the design space to include replicated secondary indexes, (4) organically handling stored procedure routing, and (3) scaling of schema complexity, data size, and number of partitions. This effort builds on two key technical contributions: an analytical cost model that can be used to quickly estimate the relative coordination cost and skew for a given workload and a candidate database design, and an informed exploration of the huge solution space based on large neighborhood search. To evaluate our methods, we integrated our database design tool with a high-performance parallel, main memory DBMS and compared our methods against both popular heuristics and a state-of-the-art research prototype [17]. Using a diverse set of benchmarks, we show that our approach improves throughput by up to a factor of 16× over these other approaches.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design

Keywords

OLTP, Parallel, Shared-Nothing, H-Store, KB, Stored Procedures

1. INTRODUCTION

The difficulty of scaling front-end applications is well known for DBMSs executing highly concurrent workloads. One approach to this problem employed by many Web-based companies is to *partition* the data and workload across a large number of commodity, shared-nothing servers using a cost-effective, parallel DBMS. Many of these companies have adopted various new DBMSs, colloquially referred to as *NoSQL* systems, that give up transactional ACID guarantees in favor of availability and scalability [9]. This approach is desirable if the consistency requirements of the data are "soft" (e.g., status updates on a social networking site that do not need to be immediately propagated throughout the application).

OLTP systems, especially enterprise OLTP systems that handle high-profile data (e.g., financial and order processing systems), also need to be scalable but cannot give up strong transactional and consistency requirements [27]. The only option previously available for these organizations was to purchase more powerful single-node machines or develop custom middleware that distributes queries over traditional DBMS nodes [41]. Both approaches are prohibitively expensive and thus are not an option for many.

As an alternative to NoSQL and custom deployments, a new class of parallel DBMSs, called *NewSQL* [7], is emerging. These systems are designed to take advantage of the partitionability of OLTP workloads to achieve scalability without sacrificing ACID guarantees [9, 43]. The OLTP workloads targeted by these NewSQL systems are characterized as having a large number of transactions that (1) are short-lived (i.e., no user stalls), (2) touch a small subset of data using index look-ups (i.e., no full table scans or large distributed joins), and (3) are repetitive (i.e., typically executed as pre-defined transaction templates or stored procedures [43, 42].)

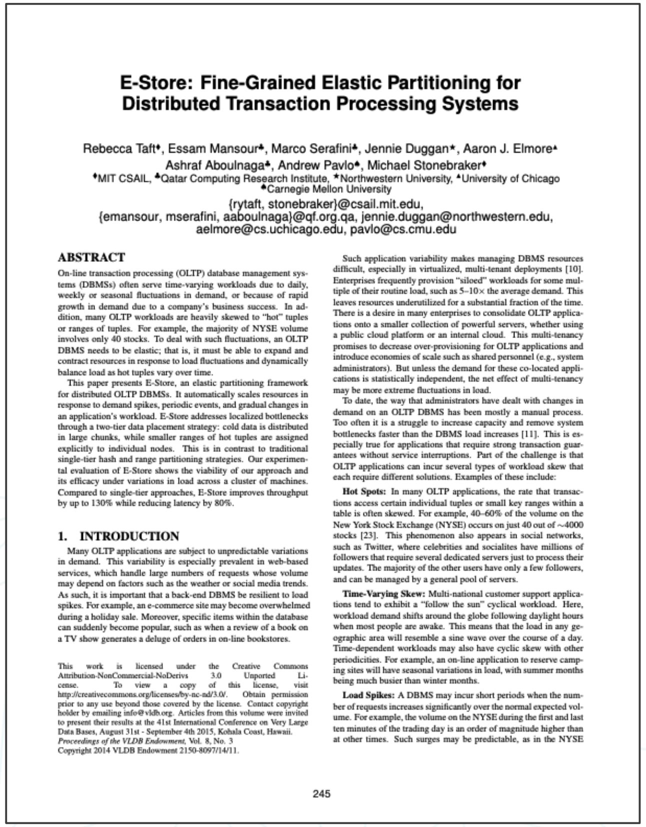
The scalability of OLTP applications on many of these newer DBMSs depends on the existence of an optimal *database design*. Such a design defines how an application's data and workload is partitioned or replicated across nodes in a cluster, and how queries and transactions are routed to nodes. This in turn determines the number of transactions that access data stored on each node and how skewed the load is across the cluster. Optimizing these two factors is critical to scaling complex systems: our experimental evidence shows that a growing fraction of distributed transactions and load skew can degrade performance by over a factor 10× Hence, without a proper design, a DBMS will perform no better than a single-node system due to the overhead caused by blocking, inter-node communication, and load balancing issues [25, 37].

Many of the existing techniques for automatic database partitioning, however, are tailored for large-scale analytical applications (i.e., data warehouses) [36, 40]. These approaches are based on the notion of *data declustering* [28], where the goal is to spread data across nodes to maximize intra-query parallelism [5, 10, 39, 49]. Much of this work is not applicable to OLTP systems because the multi-node coordination required to achieve transaction consistency dominates the performance gains obtained by this type

- Partition database to reduce the number of distributed transactions.
- Large-Neighborhood Search with sample workload trace.
- Skew-aware Cost Model
- Replicated secondary index

Pavlo, A., Curino, C., & Zdonik, S. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (pp. 61-72). ACM.

Elastic Partitioning: E-Store



E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems

Rebecca Taft♦, Essam Mansour♣, Marco Serafini♣, Jennie Duggan★, Aaron J. Elmore▲
Ashraf Aboulnaga♣, Andrew Pavlo♠, Michael Stonebraker♦
♦MIT CSAIL, ♣Qatar Computing Research Institute, ★Northwestern University, ▲University of Chicago
♠Carnegie Mellon University
{rytaft, stonebraker}@csail.mit.edu,
{emansour, mserafini, aaboulnaga}@qf.org.qa, jennie.duggan@northwestern.edu,
aelmore@cs.uchicago.edu, pavlo@cs.cmu.edu

ABSTRACT

On-line transaction processing (OLTP) database management systems (DBMSs) often serve time-varying workloads due to daily, weekly or seasonal fluctuations in demand, or because of rapid growth in demand due to a company's business success. In addition, many OLTP workloads are heavily skewed to "hot" tuples or ranges of tuples. For example, the majority of NYSE volume involves only 40 stocks. To deal with such fluctuations, an OLTP DBMS needs to be elastic; that is, it must be able to expand and contract resources in response to load fluctuations and dynamically balance load as hot tuples vary over time.

This paper presents E-Store, an elastic partitioning framework for distributed OLTP DBMSs. It automatically scales resources in response to demand spikes, periodic events, and gradual changes in an application's workload. E-Store addresses localized bottlenecks through a two-tier data placement strategy: cold data is distributed in large chunks, while smaller ranges of hot tuples are assigned explicitly to individual nodes. This is in contrast to traditional single-tier hash and range partitioning strategies. Our experimental evaluation of E-Store shows the viability of our approach and its efficacy under variations in load across a cluster of machines. Compared to single-tier approaches, E-Store improves throughput by up to 130% while reducing latency by 80%.

1. INTRODUCTION

Many OLTP applications are subject to unpredictable variations in demand. This variability is especially prevalent in web-based services, which handle large numbers of requests whose volume may depend on factors such as the weather or social media trends. As such, it is important that a back-end DBMS be resilient to load spikes. For example, an e-commerce site may become overwhelmed during a holiday sale. Moreover, specific items within the database can suddenly become popular, such as when a review of a book on a TV show generates a deluge of orders in on-line bookstores.

Proceedings of the VLDB Endowment, Vol. 8, No. 3

Such application variability makes managing DBMS resources difficult, especially in virtualized, multi-tenant deployments [10]. Enterprises frequently provision "siloed" workloads for some multiple of their routine load, such as 5–10× the average demand. This leaves resources underutilized for a substantial fraction of the time. There is a desire in many enterprises to consolidate OLTP applications onto a smaller collection of powerful servers, whether using a public cloud platform or an internal cloud. This multi-tenancy promises to decrease over-provisioning for OLTP applications and introduce economies of scale such as shared personnel (e.g., system administrators). But unless the demand for these co-located applications is statistically independent, the net effect of multi-tenancy may be more extreme fluctuations in load.

To date, the way that administrators have dealt with changes in demand on an OLTP DBMS has been mostly a manual process. Too often it is a struggle to increase capacity and remove system bottlenecks faster than the DBMS load increases [11]. This is especially true for applications that require strong transaction guarantees without service interruptions. Part of the challenge is that OLTP applications can incur several types of workload skew that each require different solutions. Examples of these include:

Hot Spots: In many OLTP applications, the rate that transactions access certain individual tuples or small key ranges within a table is often skewed. For example, 40–60% of the volume on the New York Stock Exchange (NYSE) occurs on just 40 out of ~4000 stocks [23]. This phenomenon also appears in social networks, such as Twitter, where celebrities and socialites have millions of followers that require several dedicated servers just to process their updates. The majority of the other users have only a few followers, and can be managed by a general pool of servers.

Time-Varying Skew: Multi-national customer support applications tend to exhibit a "follow the sun" cyclical workload. Here, workload demand shifts around the globe following daylight hours when most people are awake. This means that the load in any geographic area will resemble a sine wave over the course of a day. Time-dependent workloads may also have cyclic skew with other periodicities. For example, an on-line application to reserve camping sites will have seasonal variations in load, with summer months being much busier than winter months.

Load Spikes: A DBMS may incur short periods when the number of requests increases significantly over the normal expected volume. For example, the volume on the NYSE during the first and last ten minutes of the trading day is an order of magnitude higher than at other times. Such surges may be predictable, as in the NYSE

245

- Two-tiered partitioning:
  - Individual hot tuples
  - Large blocks of colder tuples
- Tuple-level monitoring
- Tuple placement planning
- Online reconfiguration

Taft, Rebecca, et al. "E-store: Fine-grained elastic partitioning for distributed transaction processing systems." Proceedings of the VLDB Endowment 8.3 (2014): 245-256.

VOLTDB

Thank you

VOLTDB