SCALE OUT AND CONQUER:

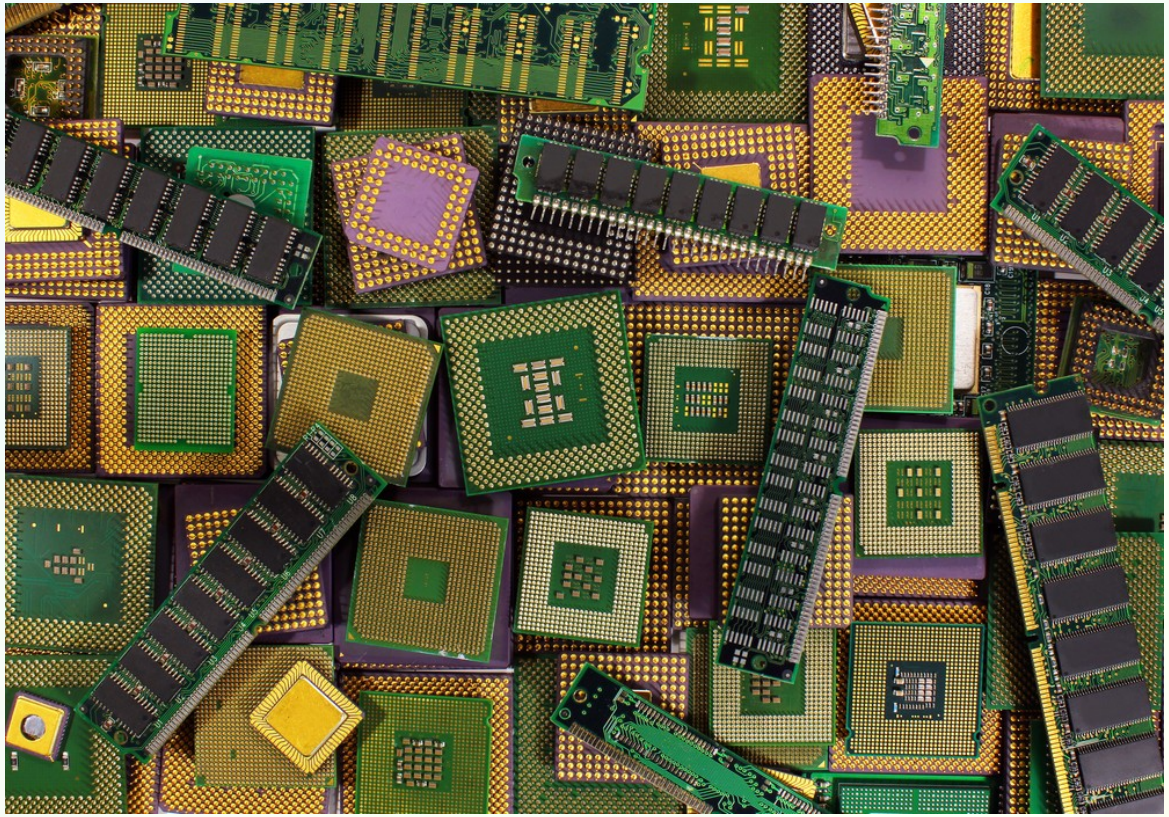
ARCHITECTURAL DECISIONS BEHIND DISTRIBUTED IN-MEMORY SYSTEMS

VLADIMIR OZEROV
YAKOV ZHDANOV

WHO?

Yakov Zhdanov:
- GridGain's Product Development VP
- With GridGain since 2010
- Apache Ignite committer and PMC
- Passion for performance & scalability
- Finding ways to make product better
- St. Petersburg, Russia

WHY IN-MEMORY?



PLAN

1. Data partitioning and affinity functions examples

PLAN

1. Data partitioning and affinity functions examples
2. Data affinity colocation

PLAN

1. Data partitioning and affinity functions examples
2. Data affinity colocation
3. Synchronization in distributed systems

PLAN

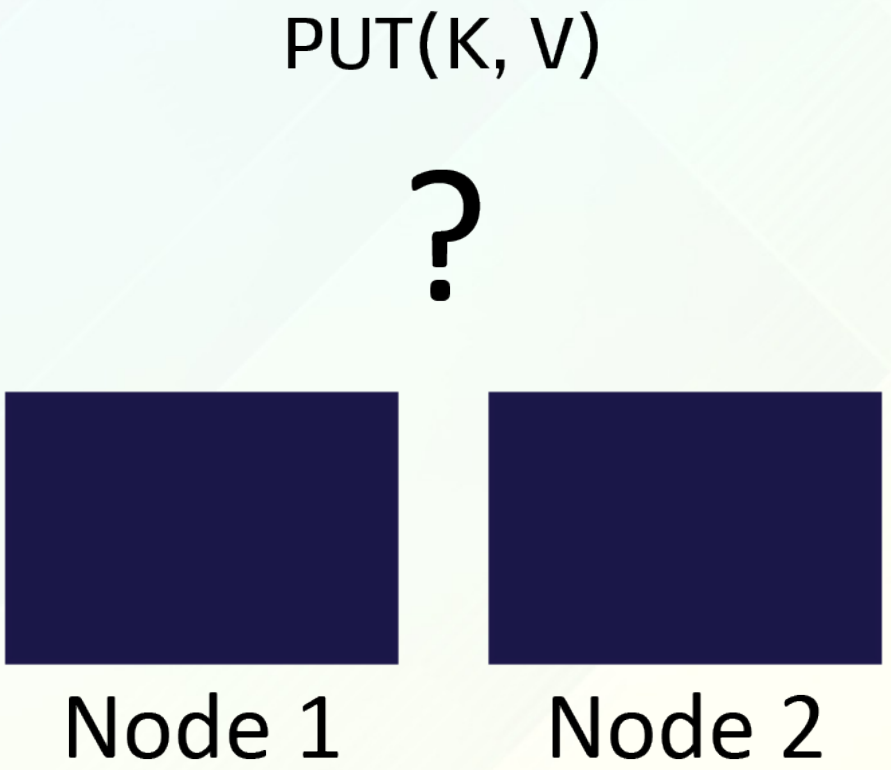
1. Data partitioning and affinity functions examples
2. Data affinity colocation
3. Synchronization in distributed systems
4. Multithreading: local architecture

PLAN

1. Data partitioning and affinity functions examples
2. Data affinity colocation
3. Synchronization in distributed systems
4. Multithreading: local architecture

WHERE?

On which node of the cluster does the key reside?



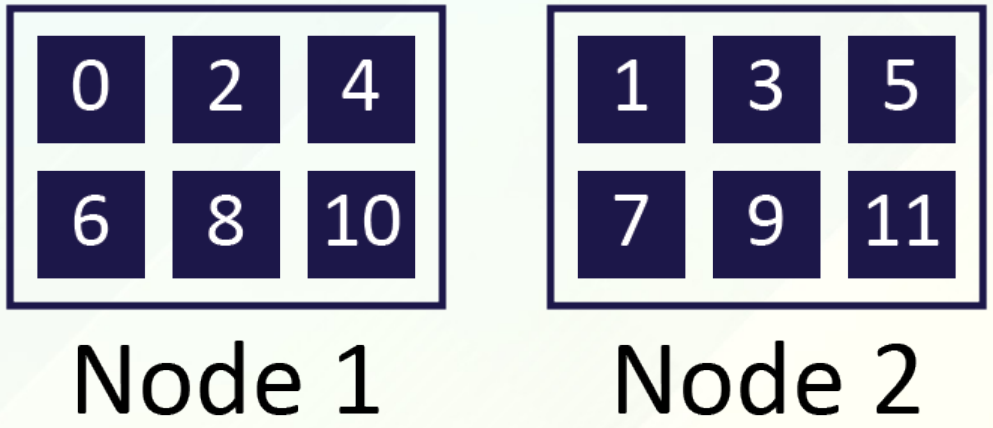
AFFINITY

Partition → Node

AFFINITY

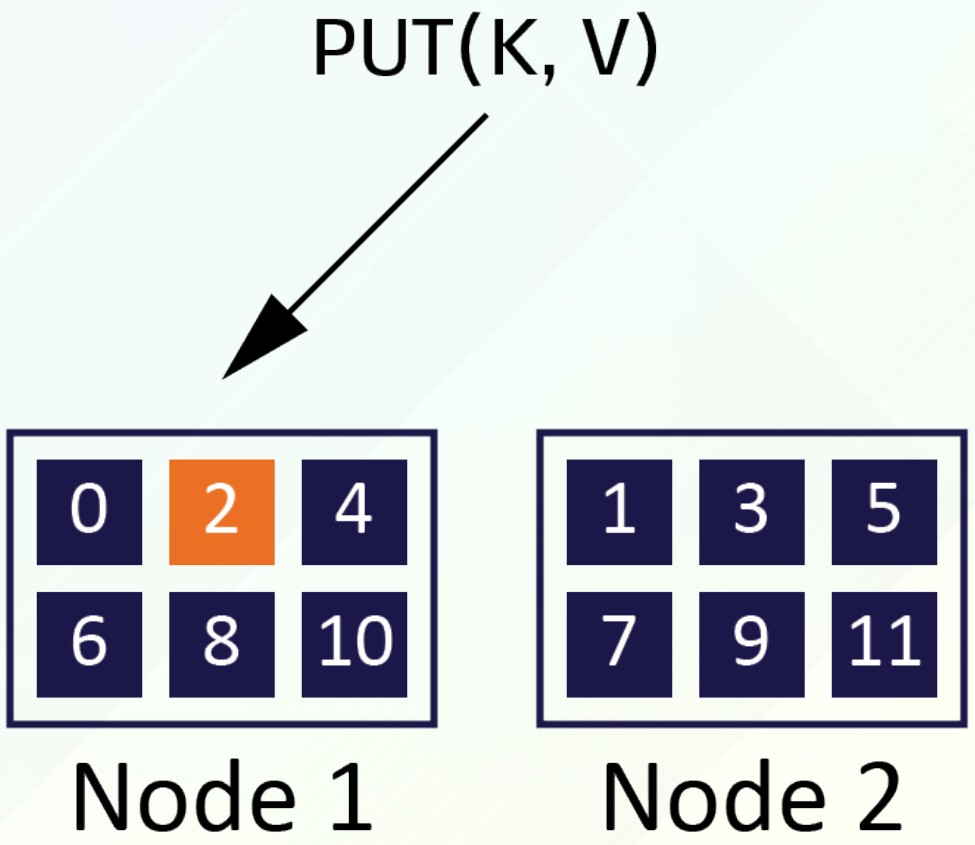
PUT(K, V)

?

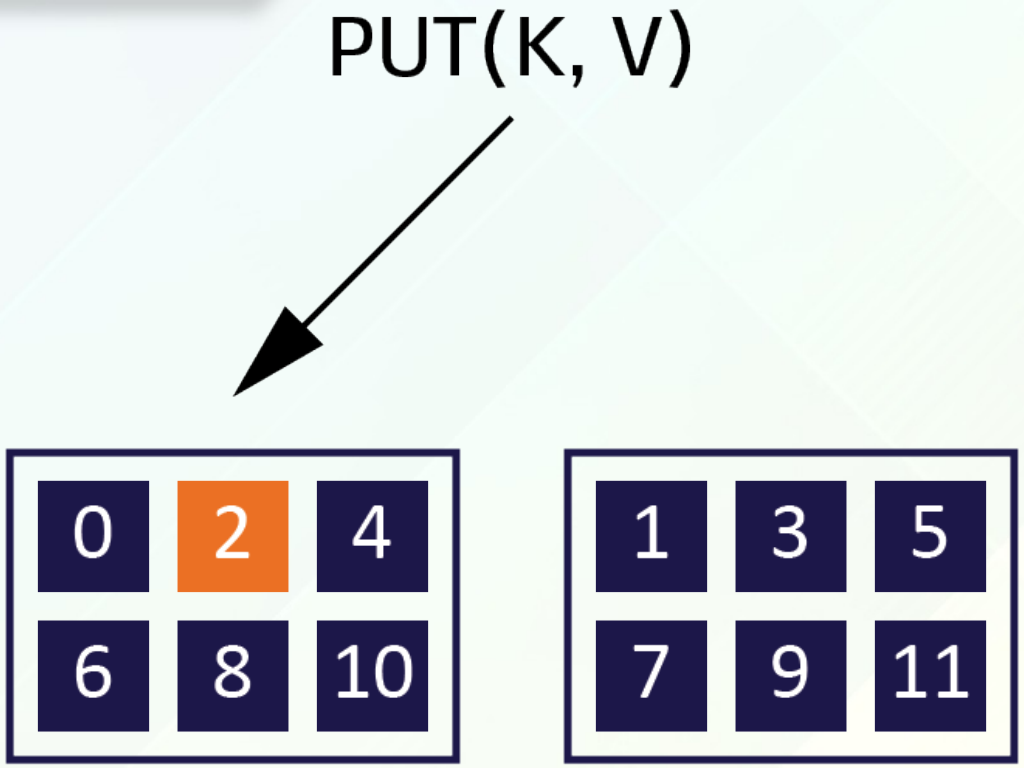


AFFINITY

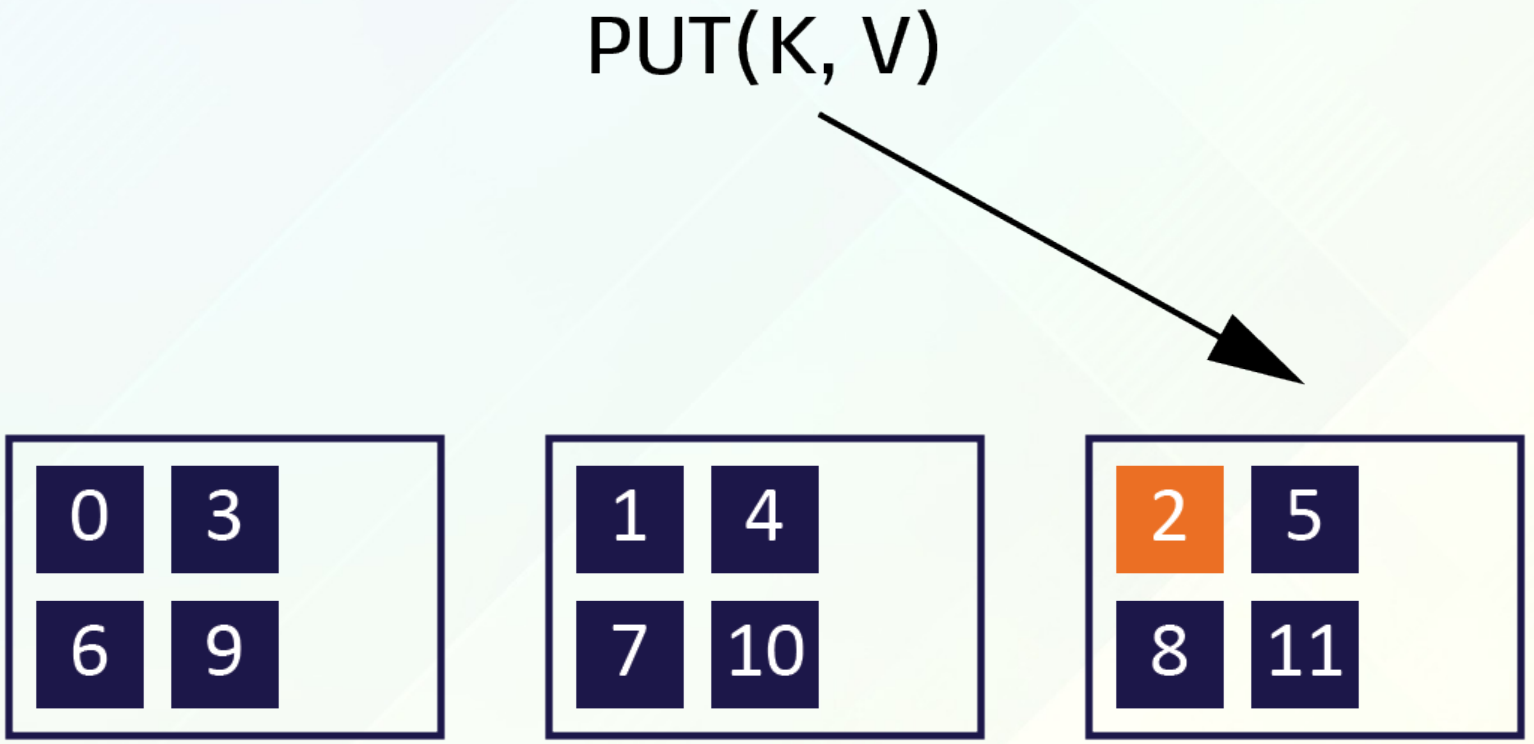
AFFINITY



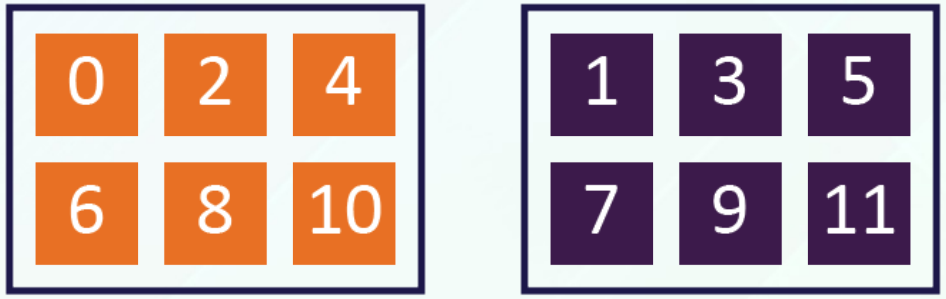
NAIVE AFFINITY



NAIVE AFFINITY

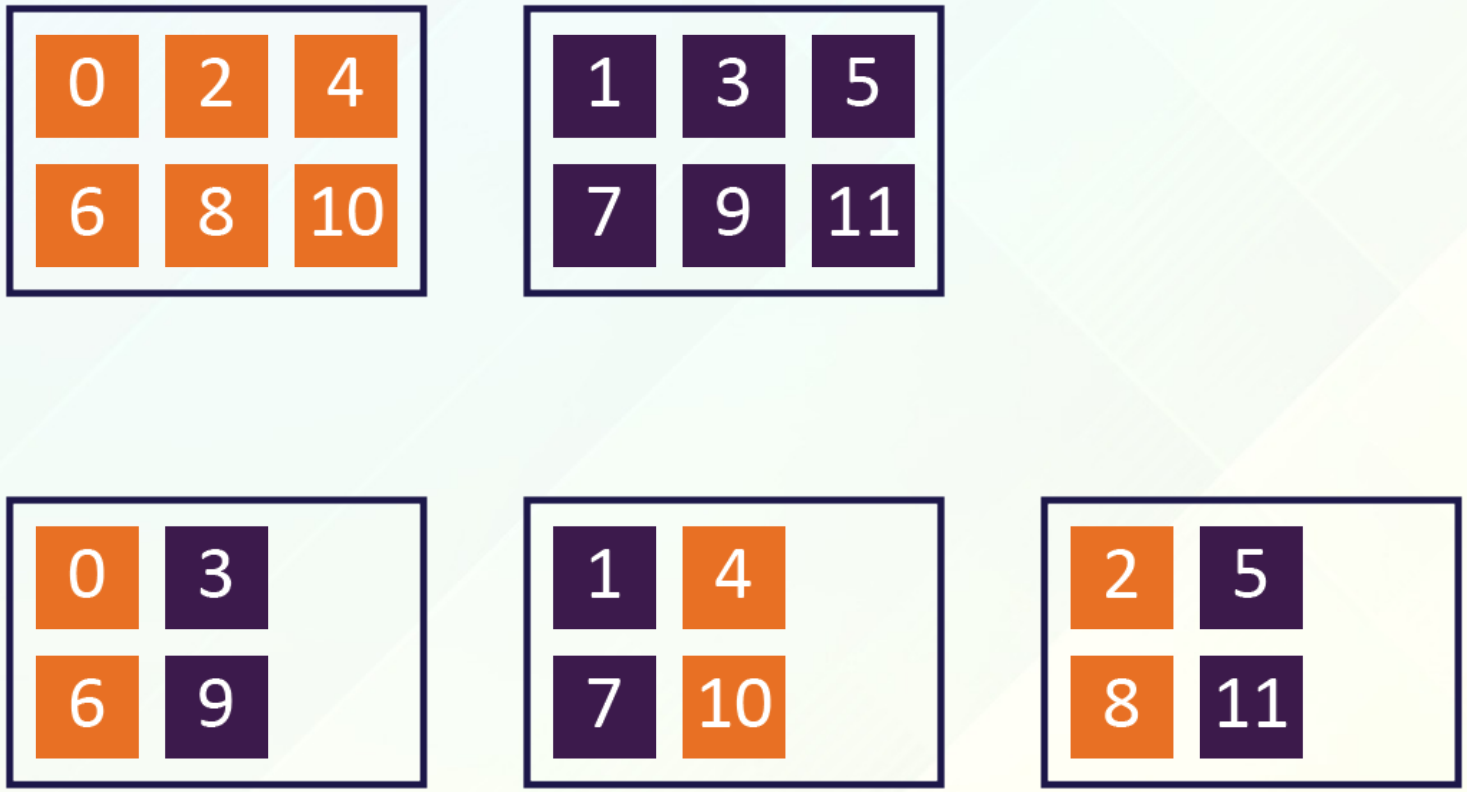


NAIVE AFFINITY

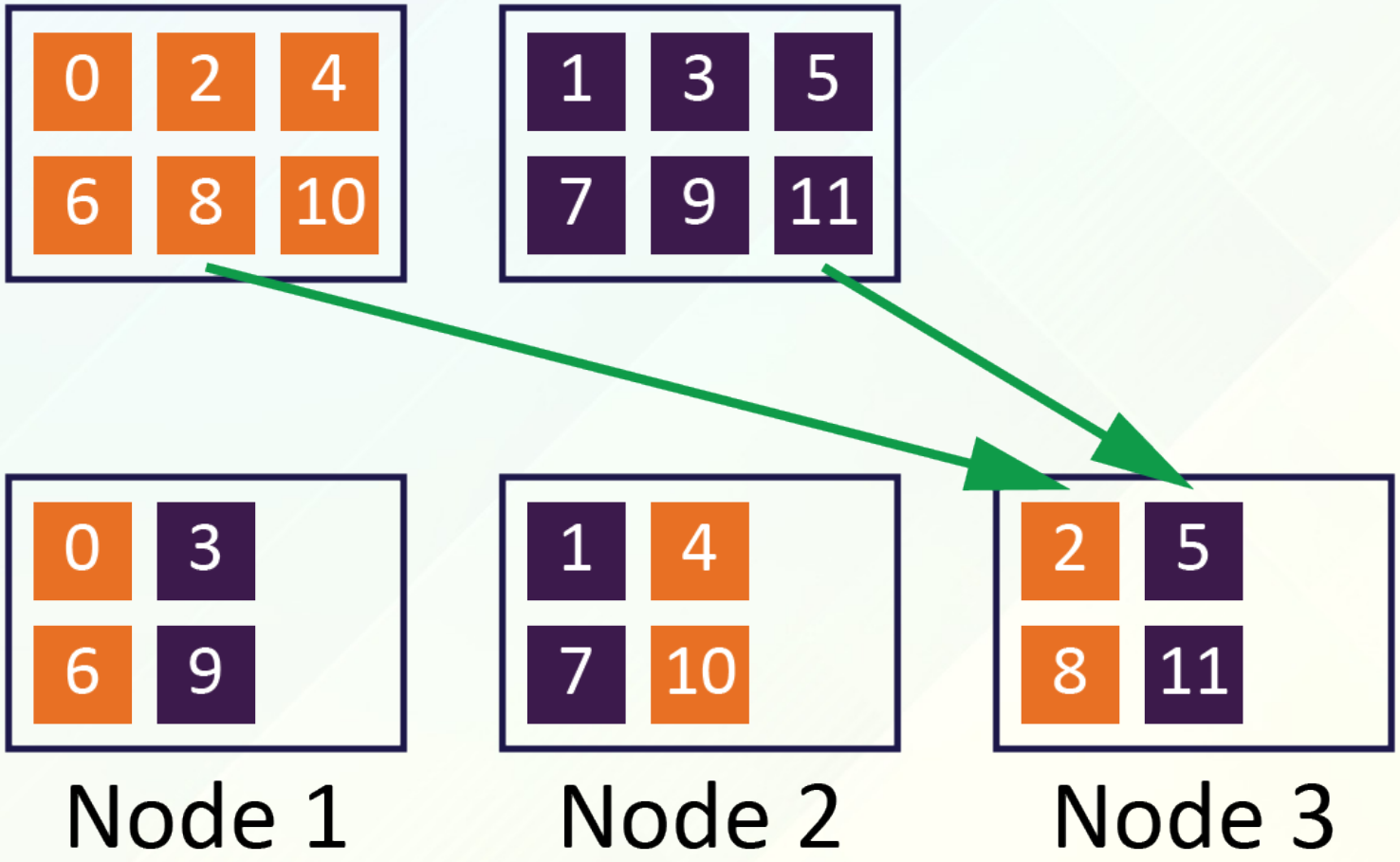


Node 1 Node 2

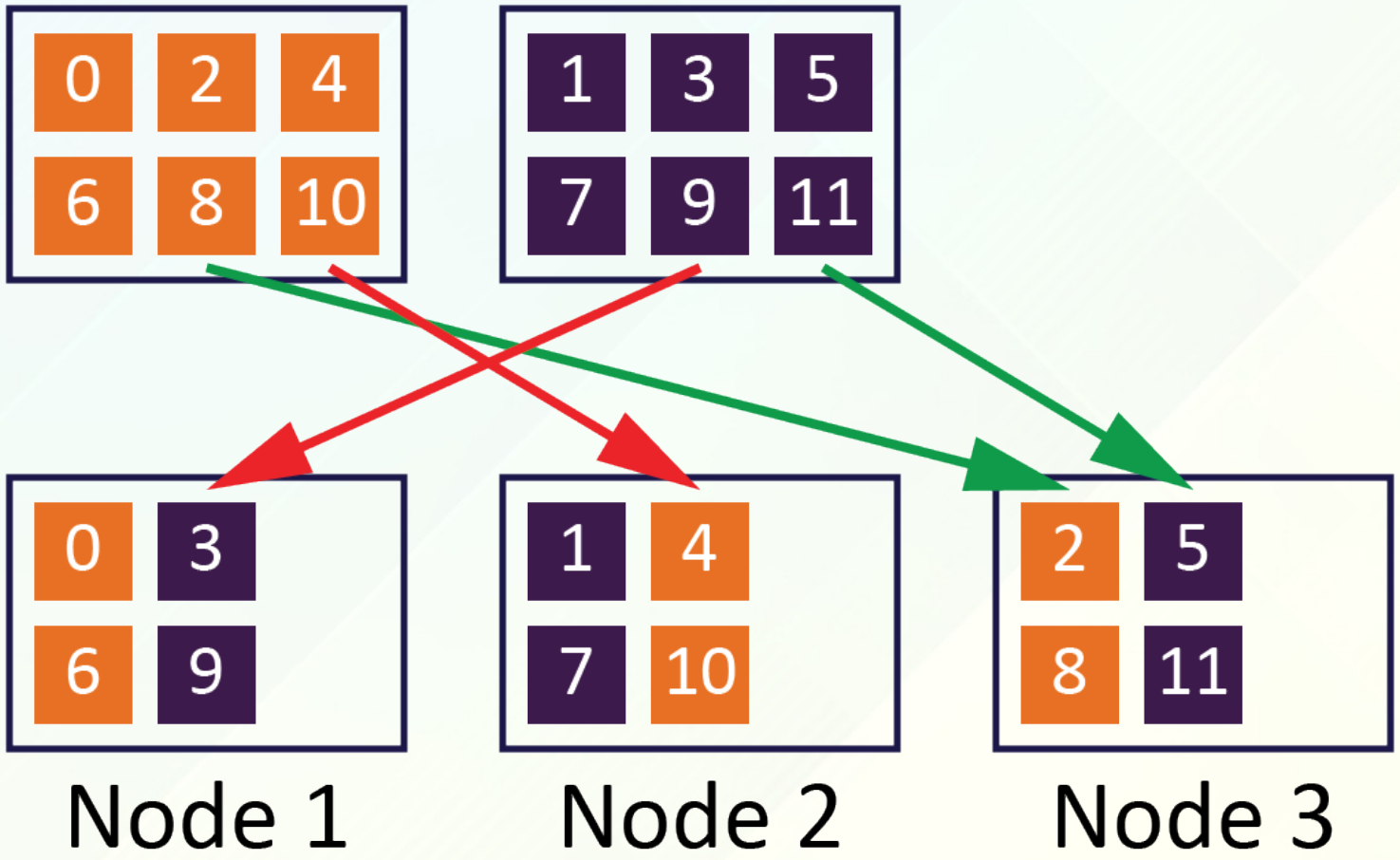
NAIVE AFFINITY



NAIVE AFFINITY



NAIVE AFFINITY



NAIVE AFFINITY

Problem: partition to node mapping depends on nodes count.

NODE = F (PARTITION, **NODES_COUNT**);

AFFINITY: BETTER ALGORITHMS

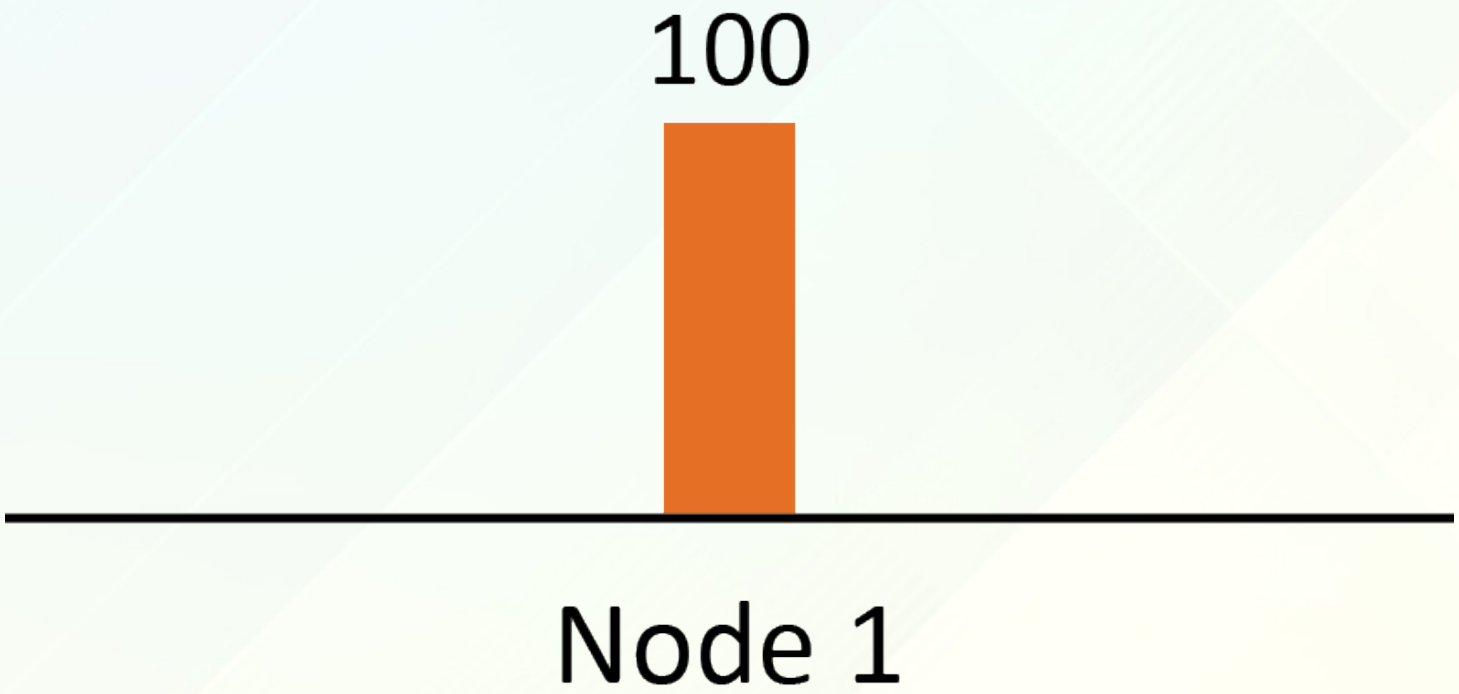
- Consistent hashing [1]
- Rendezvous hashing (or highest random weight - HRW) [2]

[1] https://en.wikipedia.org/wiki/Consistent_hashing
[2] https://en.wikipedia.org/wiki/Rendezvous_hashing

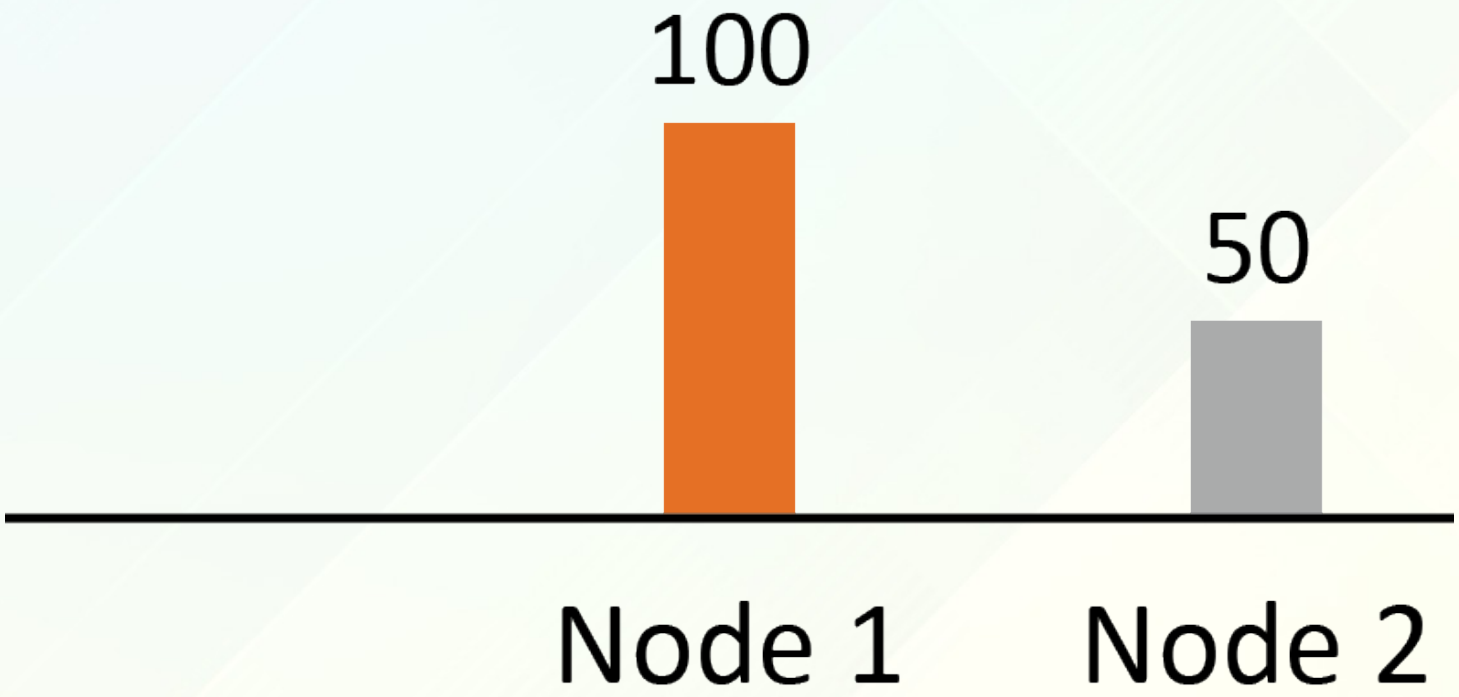
RENDEZVOUS AFFINITY

WEIGHT = W(PARTITION, NODE);

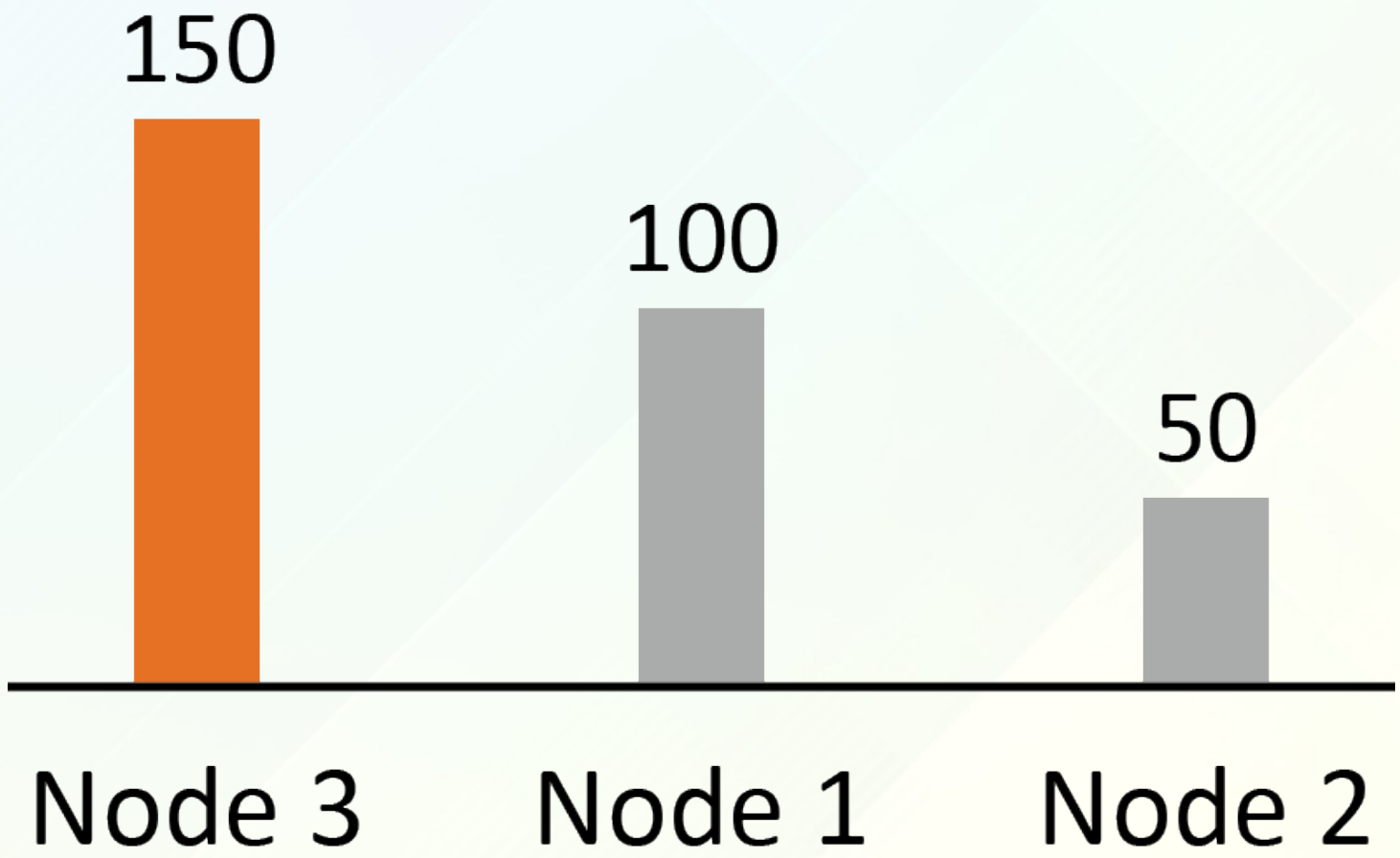
RENDEZVOUS AFFINITY



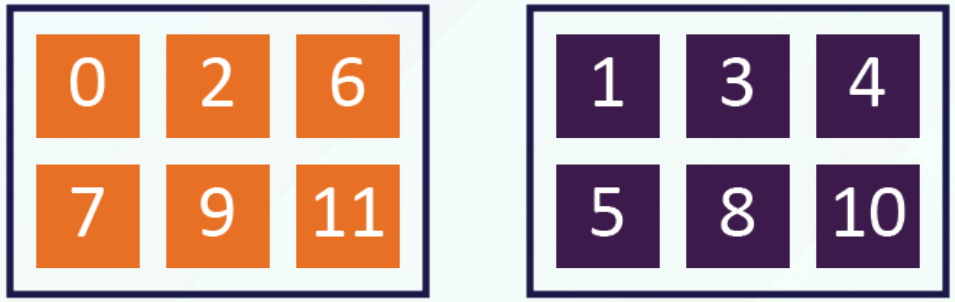
RENDEZVOUS AFFINITY



RENDEZVOUS AFFINITY

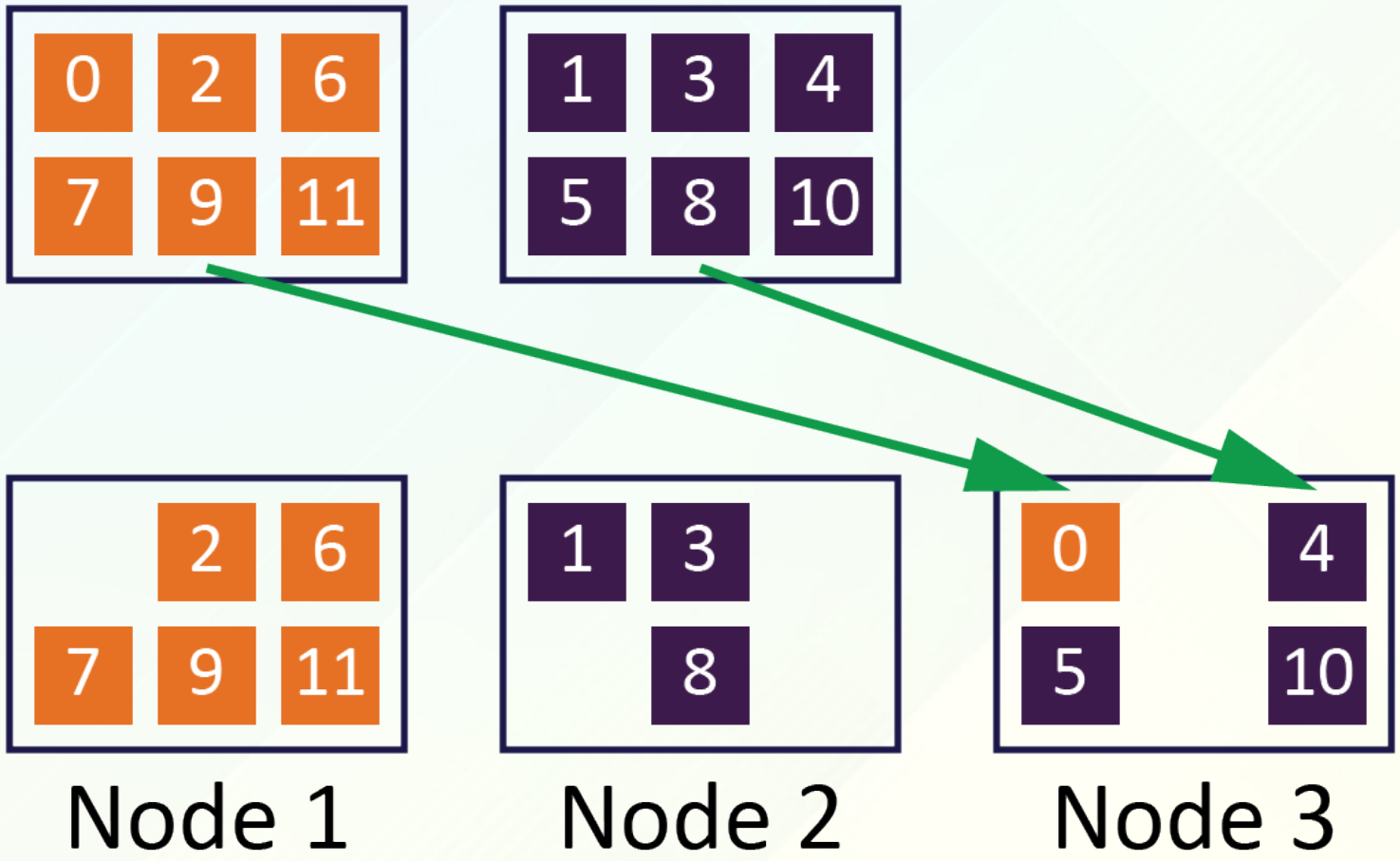


RENDEZVOUS AFFINITY

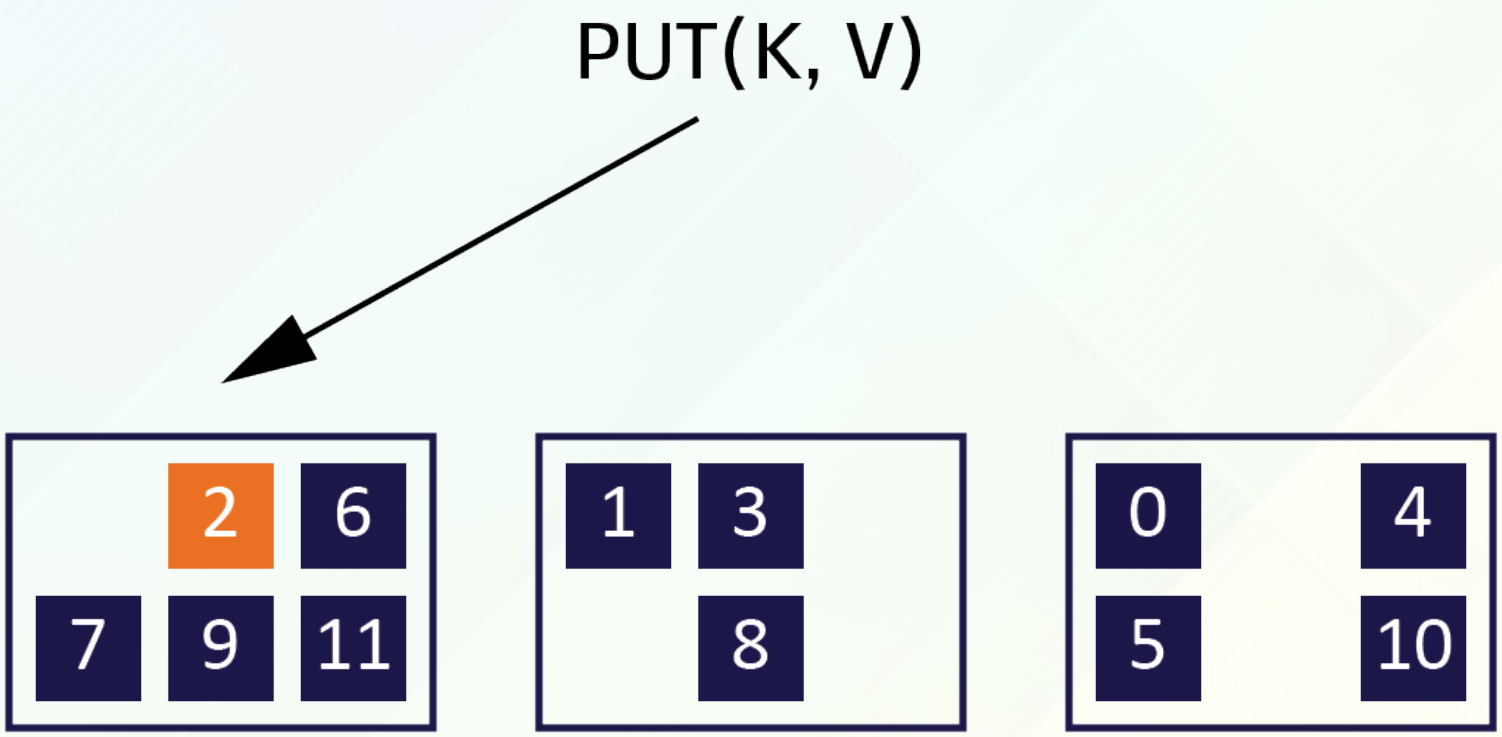


Node 1 Node 2

RENDEZVOUS AFFINITY



RENDEZVOUS AFFINITY: EVEN DISTRIBUTION?



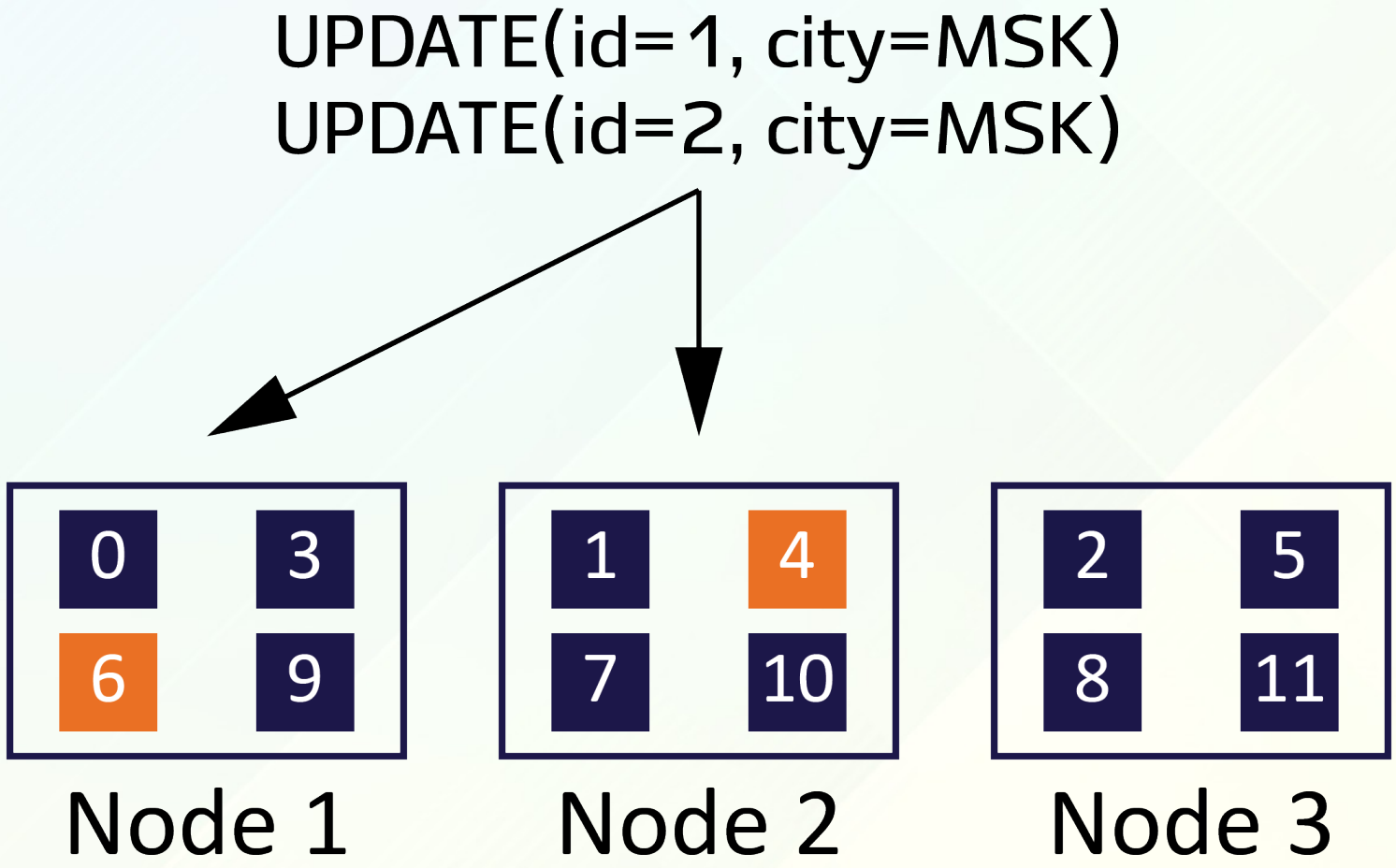
PLAN

1. Data partitioning and affinity functions examples
2. Data affinity colocation
3. Synchronization in distributed systems
4. Multithreading: local architecture

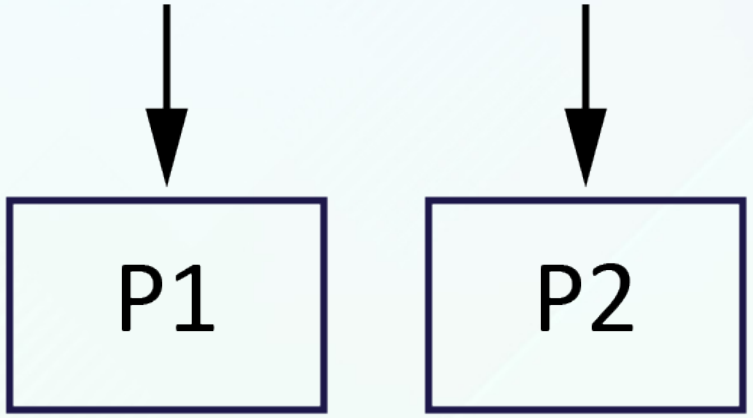
TRANSACTIONS: NO COLOCATION

```
1: class Customer {
2:         long id;
3:         City city;
4: }
```

TRANSACTIONS: NO COLOCATION

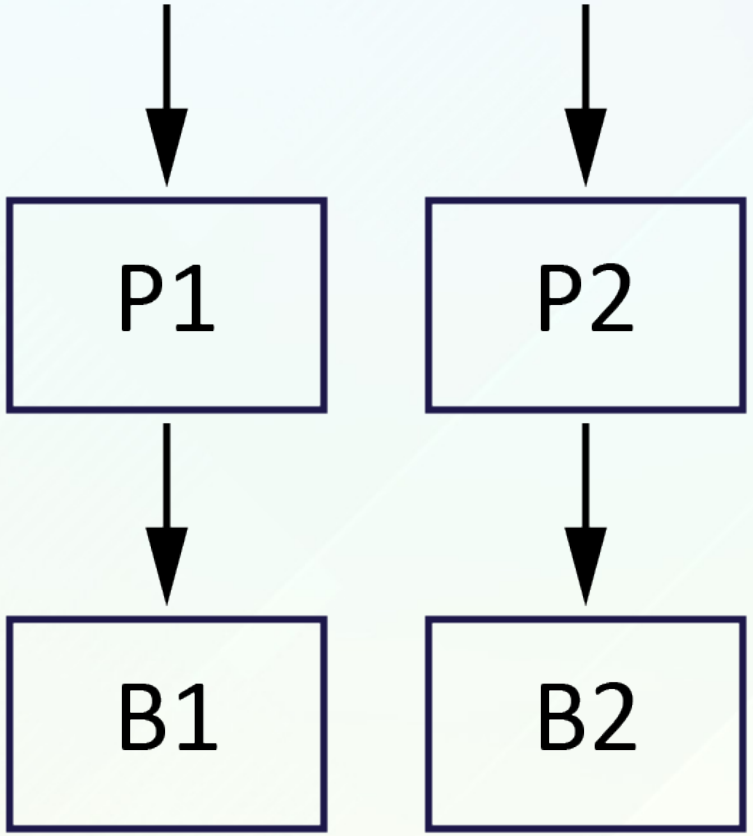


TRANSACTIONS: NO COLOCATION



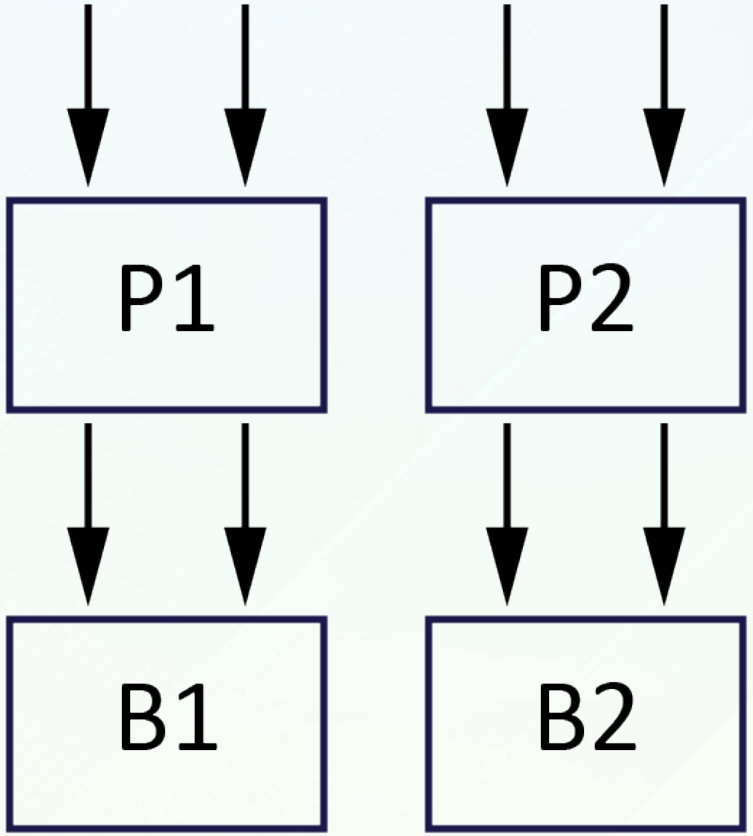
2 (2 nodes)

TRANSACTIONS: NO COLOCATION



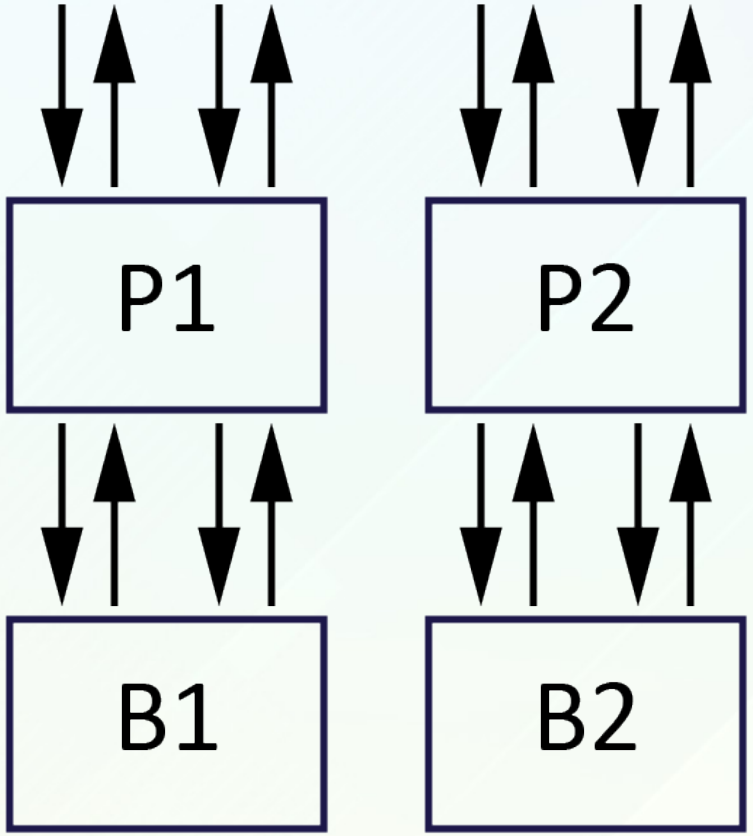
2 (2 nodes)
2 (primary + backup)

TRANSACTIONS: NO COLOCATION



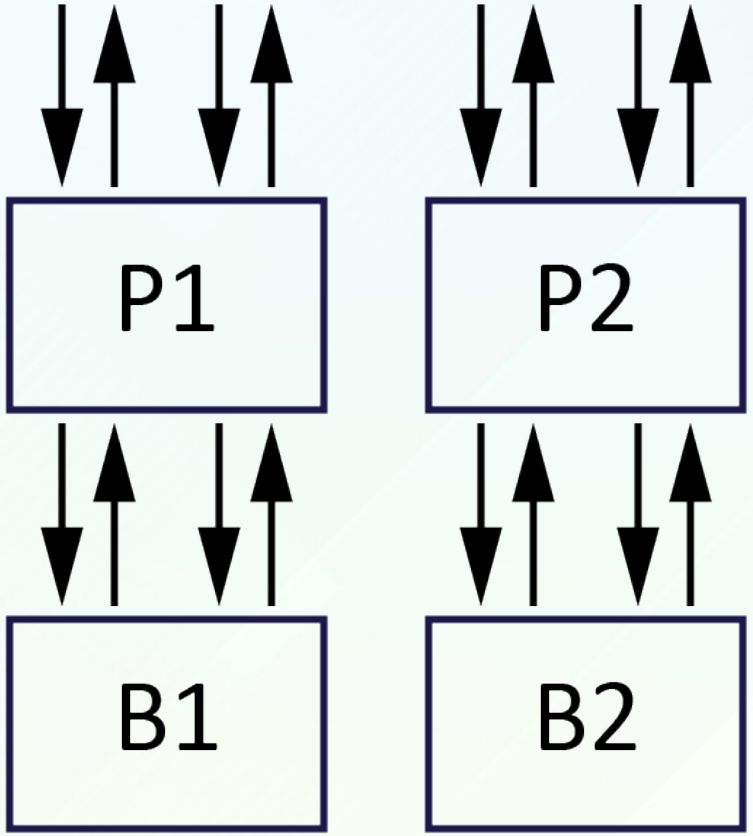
2 (2 nodes)
2 (primary + backup)
2 (two-phase commit)

TRANSACTIONS: NO COLOCATION



2 (2 nodes)
2 (primary + backup)
2 (two-phase commit)
2 (request-response)

TRANSACTIONS: NO COLOCATION



2 (2 nodes)
2 (primary + backup)
2 (two-phase commit)
2 (request-response)

16
Messages

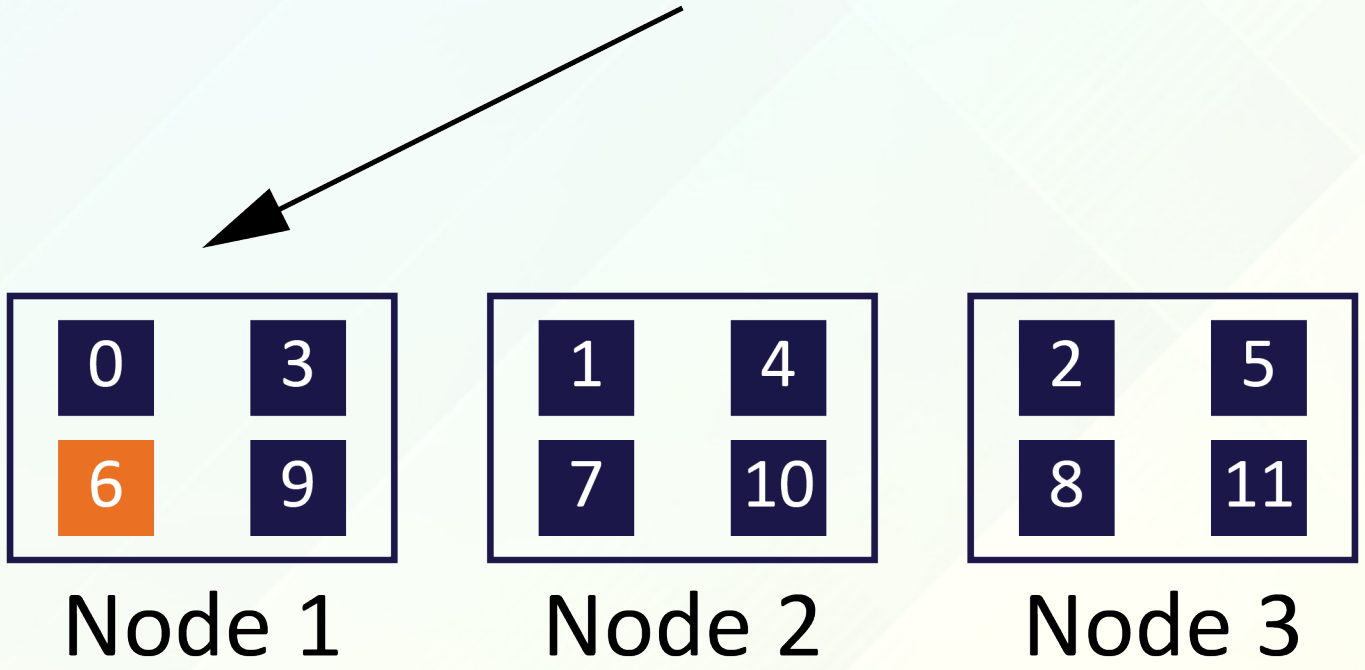
TRANSACTIONS: NO COLOCATION

TRANSACTIONS: WITH COLOCATION

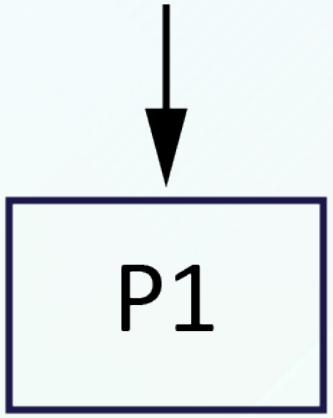
```
1: class Customer {
2:     long id;
3:
4:     @AffinityKeyMapped
5:     City city;
6: }
```

TRANSACTIONS: WITH COLOCATION

UPDATE(id=1, city=MSK)
UPDATE(id=2, city=MSK)

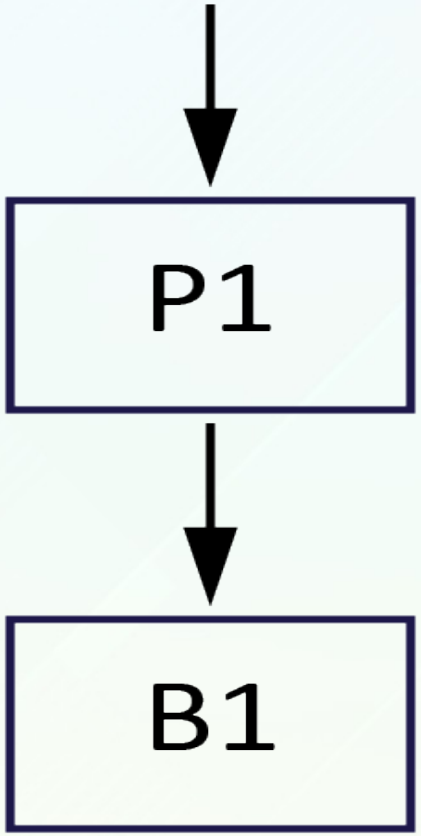


TRANSACTIONS: WITH COLOCATION



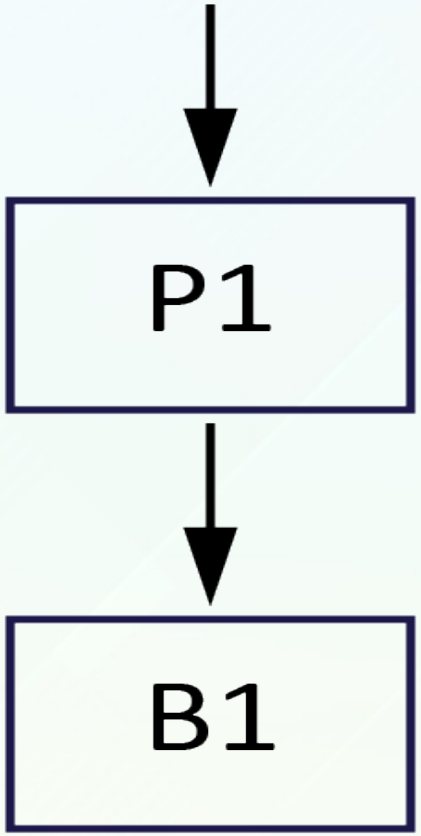
1 (1 node)

TRANSACTIONS: WITH COLOCATION



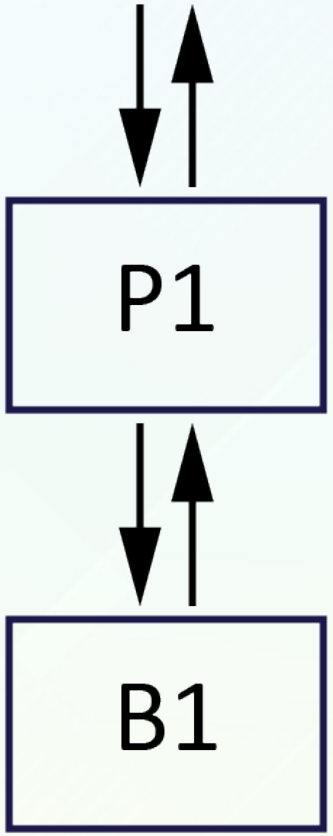
1 (1 node)
2 (primary + backup)

TRANSACTIONS: WITH COLOCATION



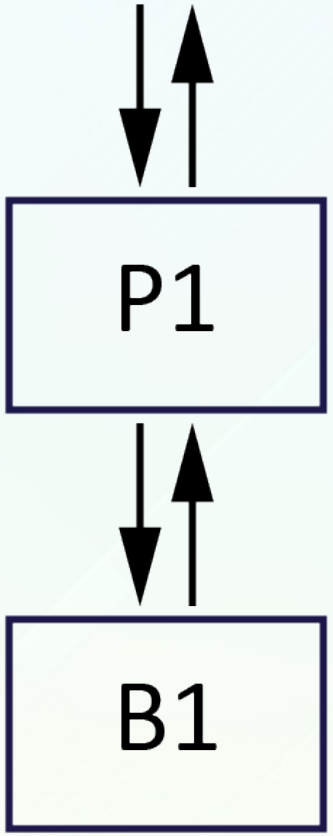
1 (1 node)
2 (primary + backup)
(one-phase commit)

TRANSACTIONS: WITH COLOCATION




```
1 (1 node)
2 (primary + backup)
  (one-phase commit)
1 (request-response)
```

TRANSACTIONS: WITH COLOCATION



1 (1 node)
2 (primary + backup)
(one-phase commit)
1 (request-response)

4 Messages

TRANSACTIONS: COLOCATION VS NO COLOCATION

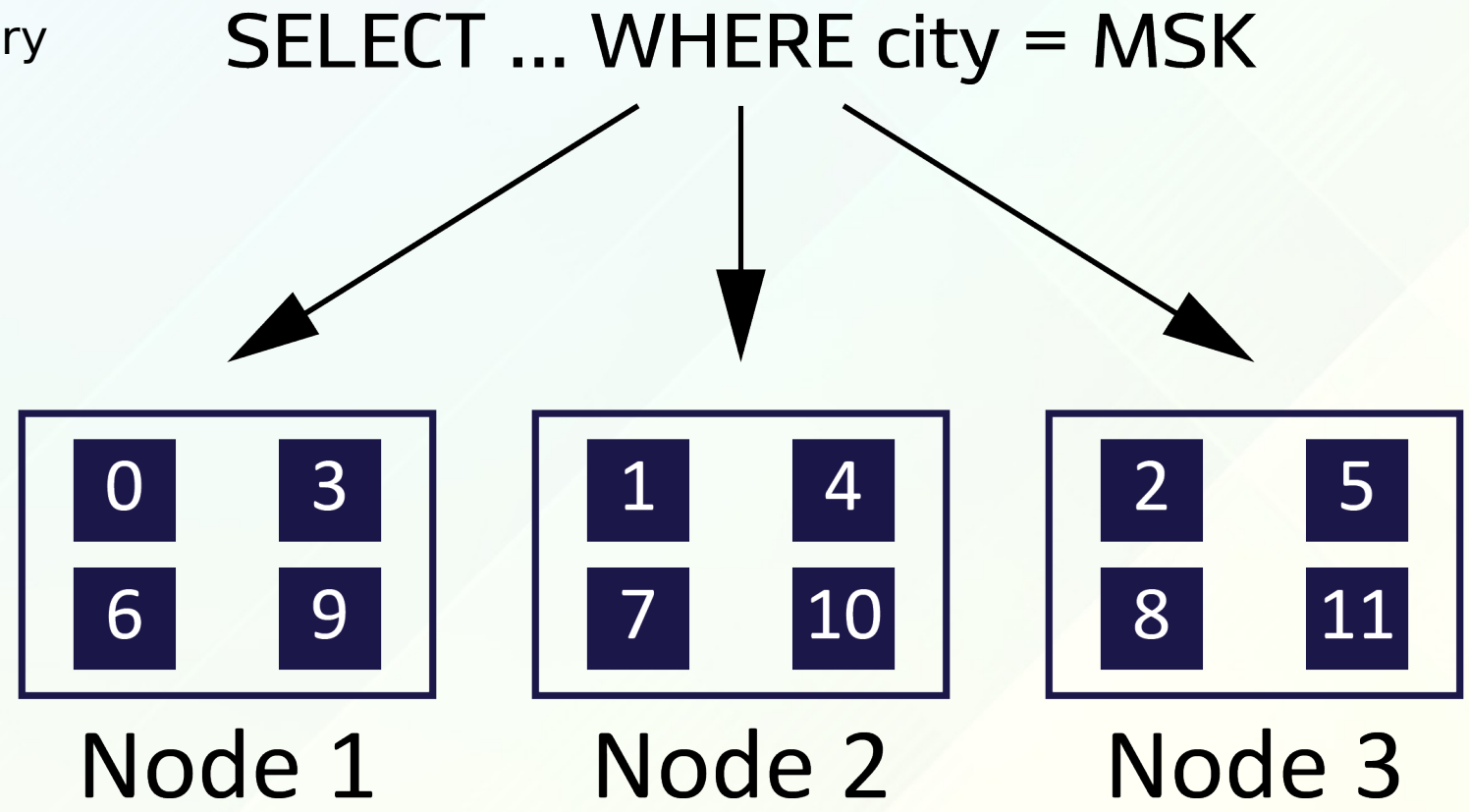
4 Messages

VS

16 Messages

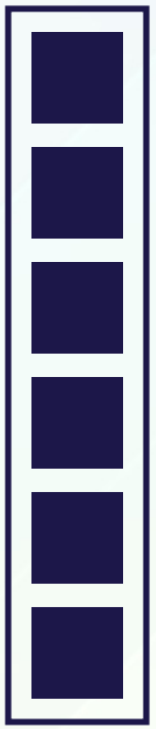
SQL

Let’s run a query on our data



SQL

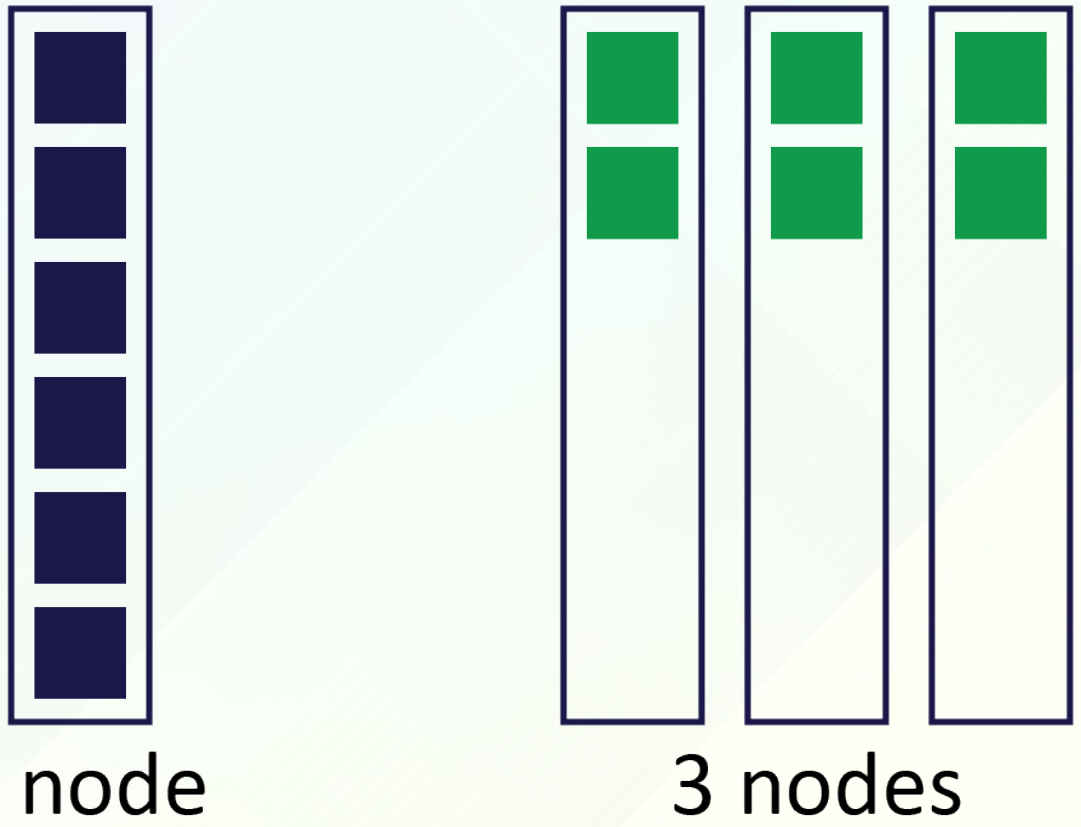
No colocation:
FULL SCAN



1 node

SQL

No colocation:
FULL SCAN



No colocation:
FULL SCAN

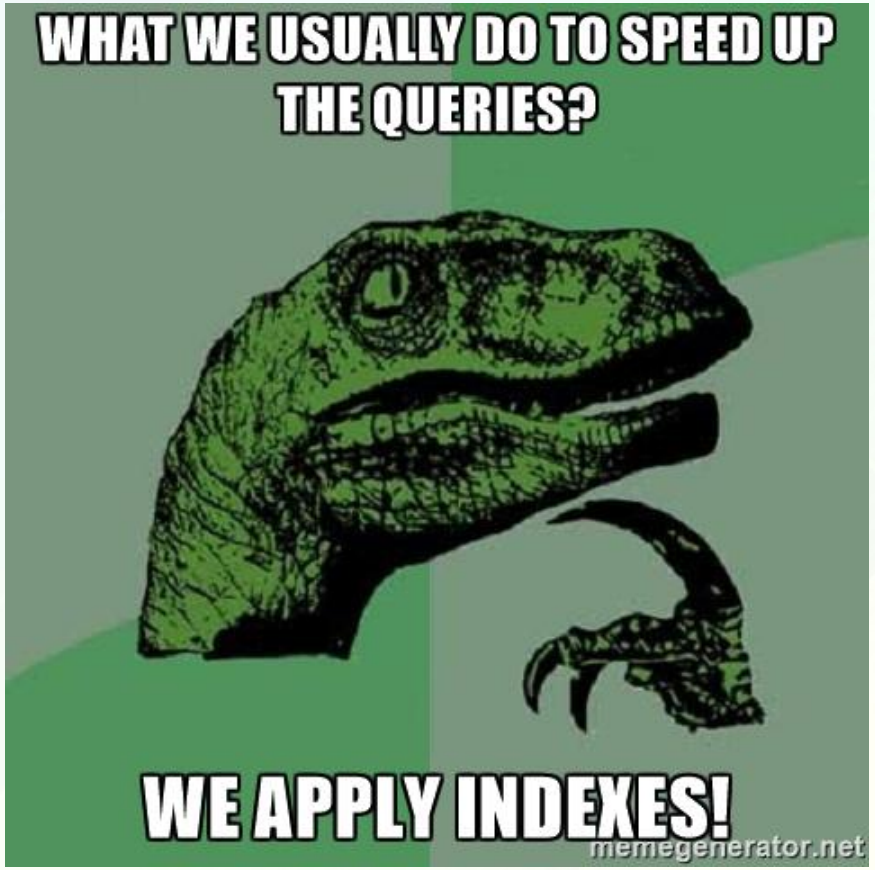
1/3x Latency

No colocation:
FULL SCAN

1/3x Latency

3x Capacity

SQL


WHAT WE USUALLY DO TO SPEED UP THE QUERIES?
WE APPLY INDEXES!
memegenerator.net

SQL

1 node

SQL

1 node

N nodes

What about complexity?

$$\log 1\_000\_000 \approx 20$$

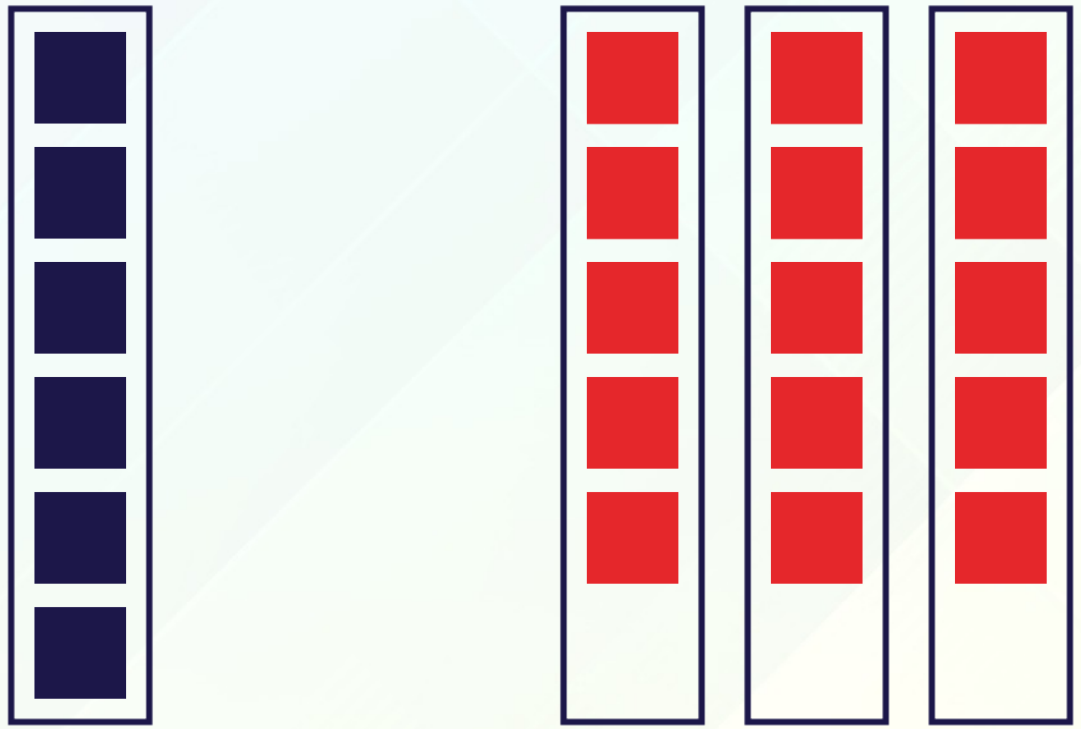
What about complexity?

```
log 1_000_000 ≈ 20
```

vs

```
log   333_333 ≈ 18
log   333_333 ≈ 18
log   333_333 ≈ 18
```

SQL: INDEXED



1 node

3 nodes

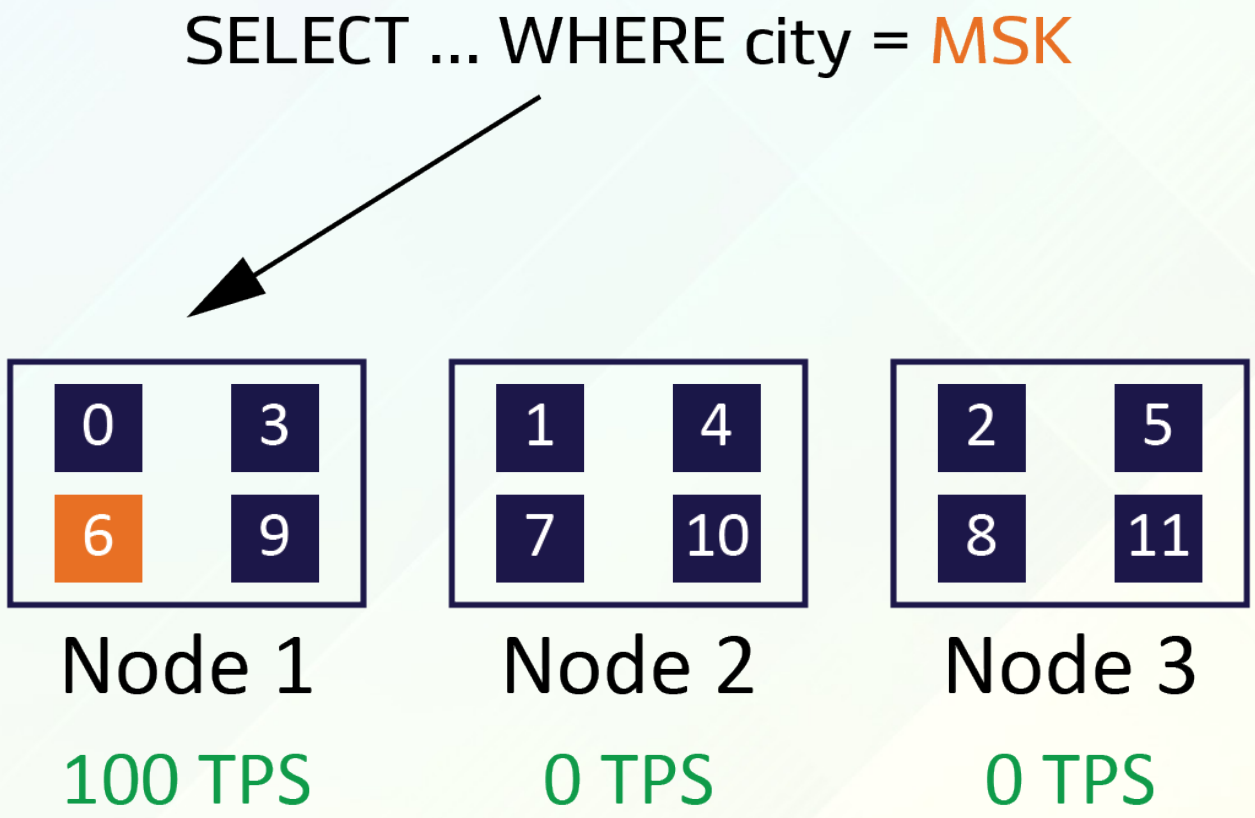
No colocation:
INDEXED QUERY

Same latency!

Same capacity!

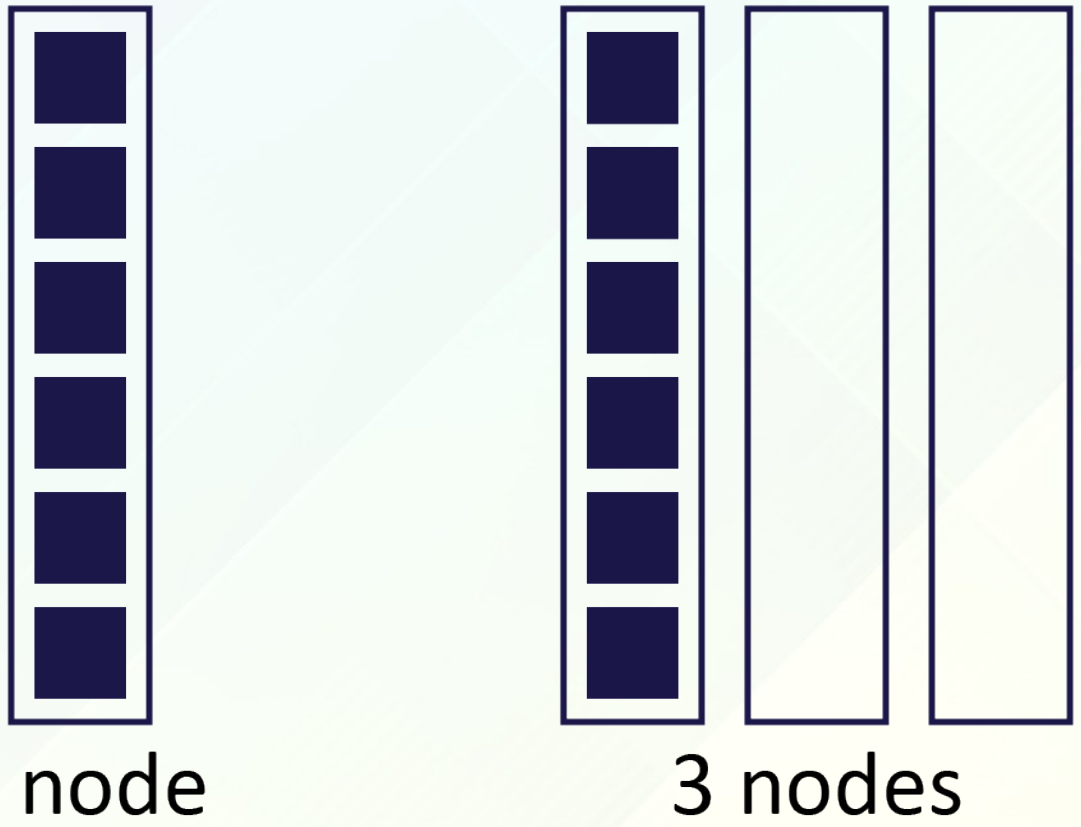
SQL: INDEX AND COLOCATION

Colocation:
INDEXED QUERY



SQL

Colocation:
INDEXED QUERY



SQL: INDEX AND COLOCATION

Colocation:
INDEXED QUERY

Same latency

But 3x capacity!

SQL: EVEN DISTRIBUTION WITH COLOCATION?

Node 1

100 TPS

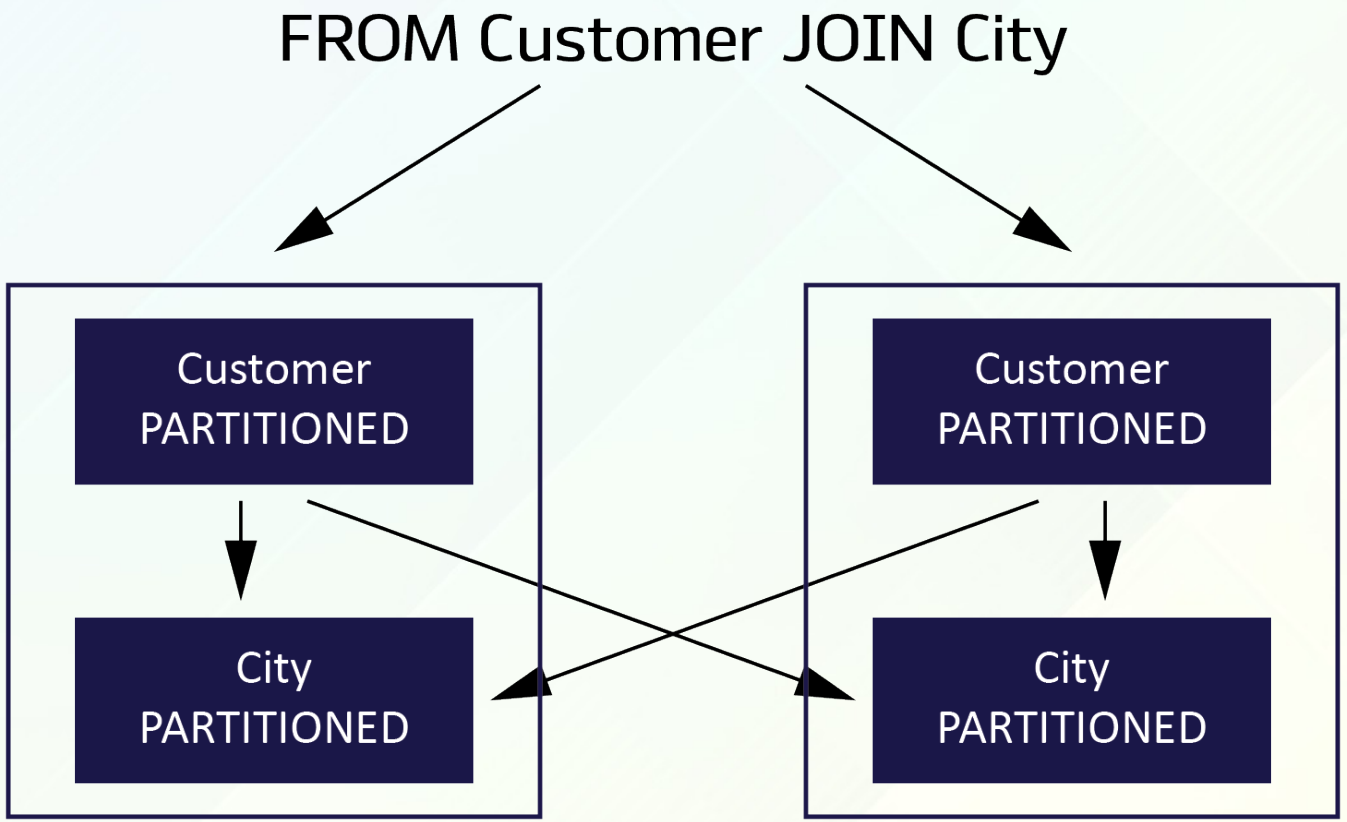
Node 2

50 TPS

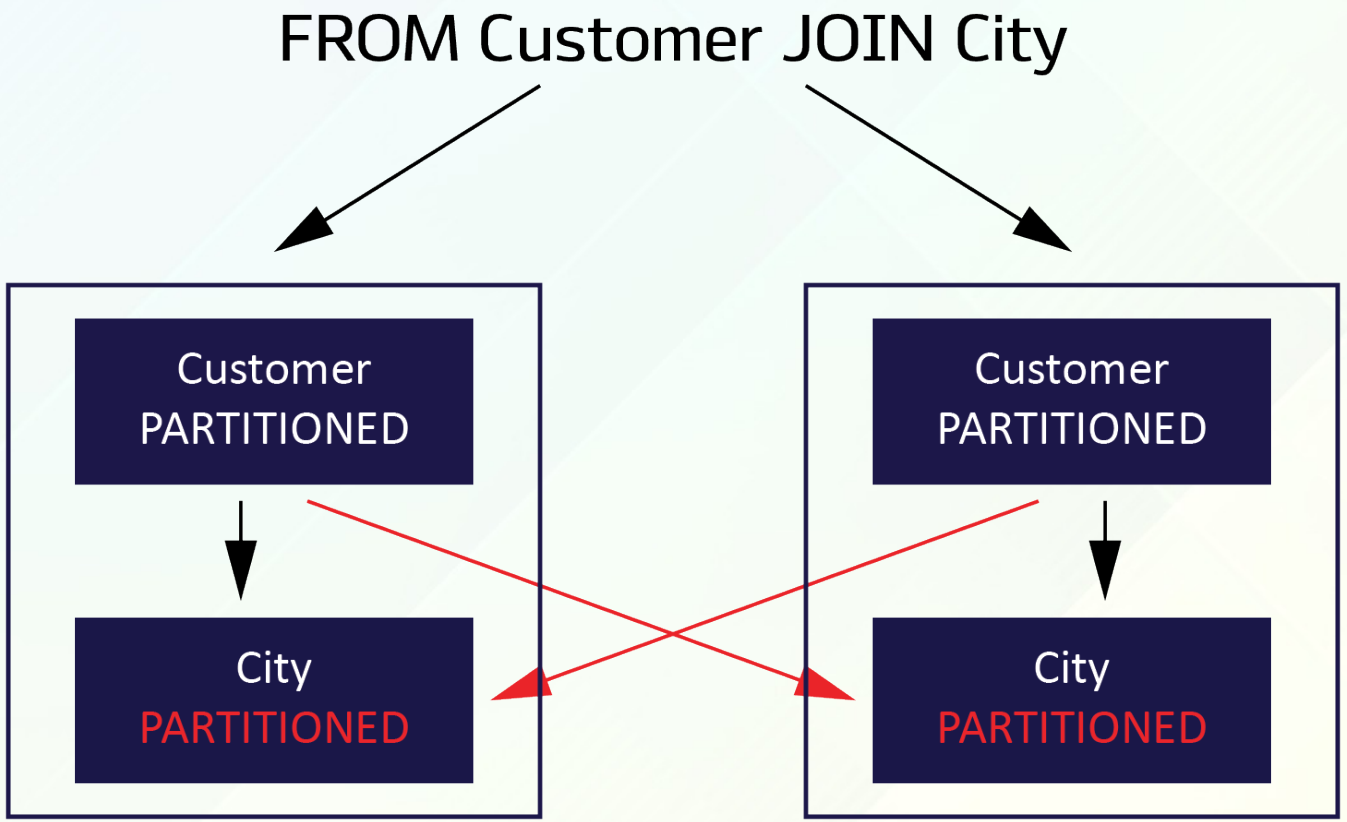
Node 3

10 TPS

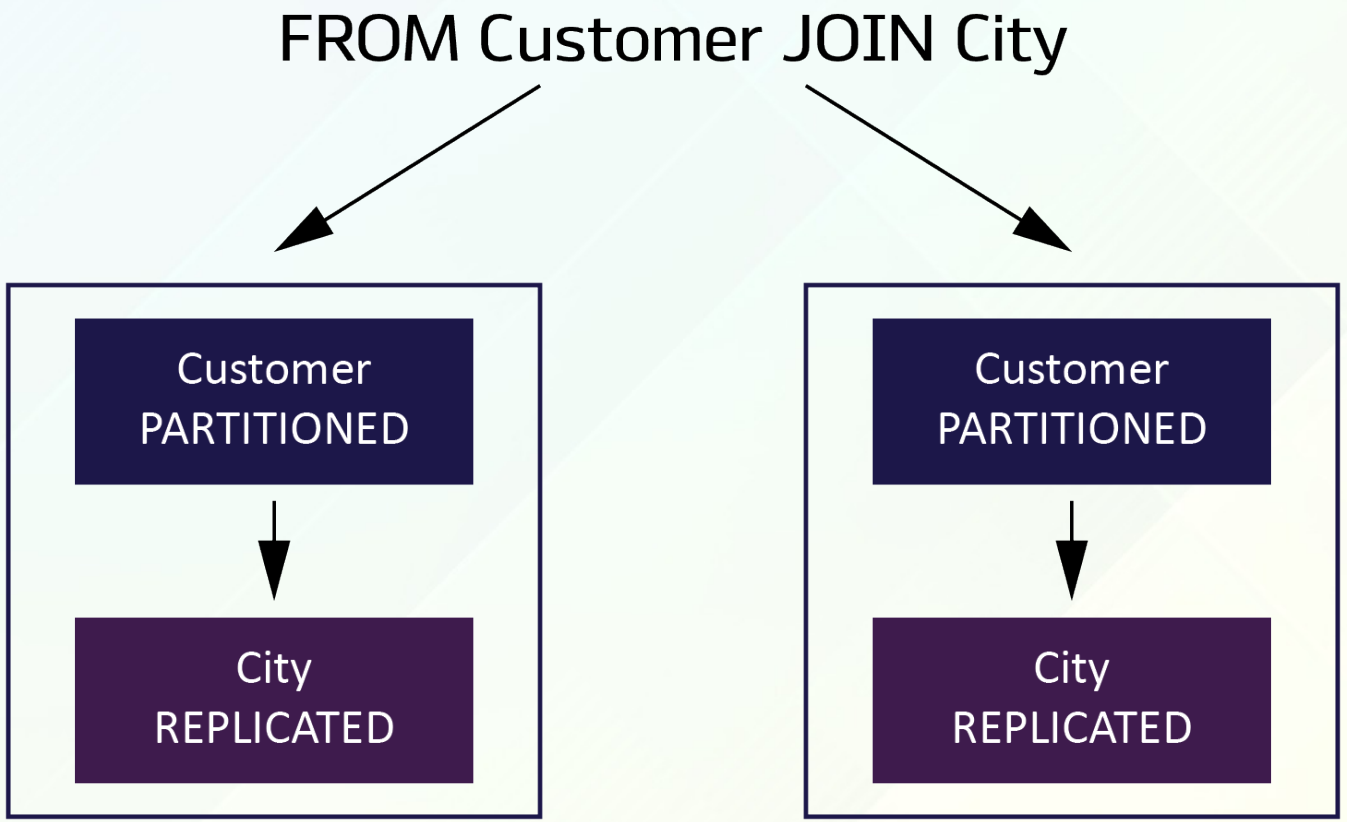
SQL: JOINS IN DISTRIBUTED ENVIRONMENT



SQL: JOINS WITH COLOCATION



SQL: JOINS WITH REPLICATION



PLAN

1. Data partitioning and affinity functions examples
2. Data affinity colocation
3. Synchronization in distributed systems
4. Multithreading: local architecture

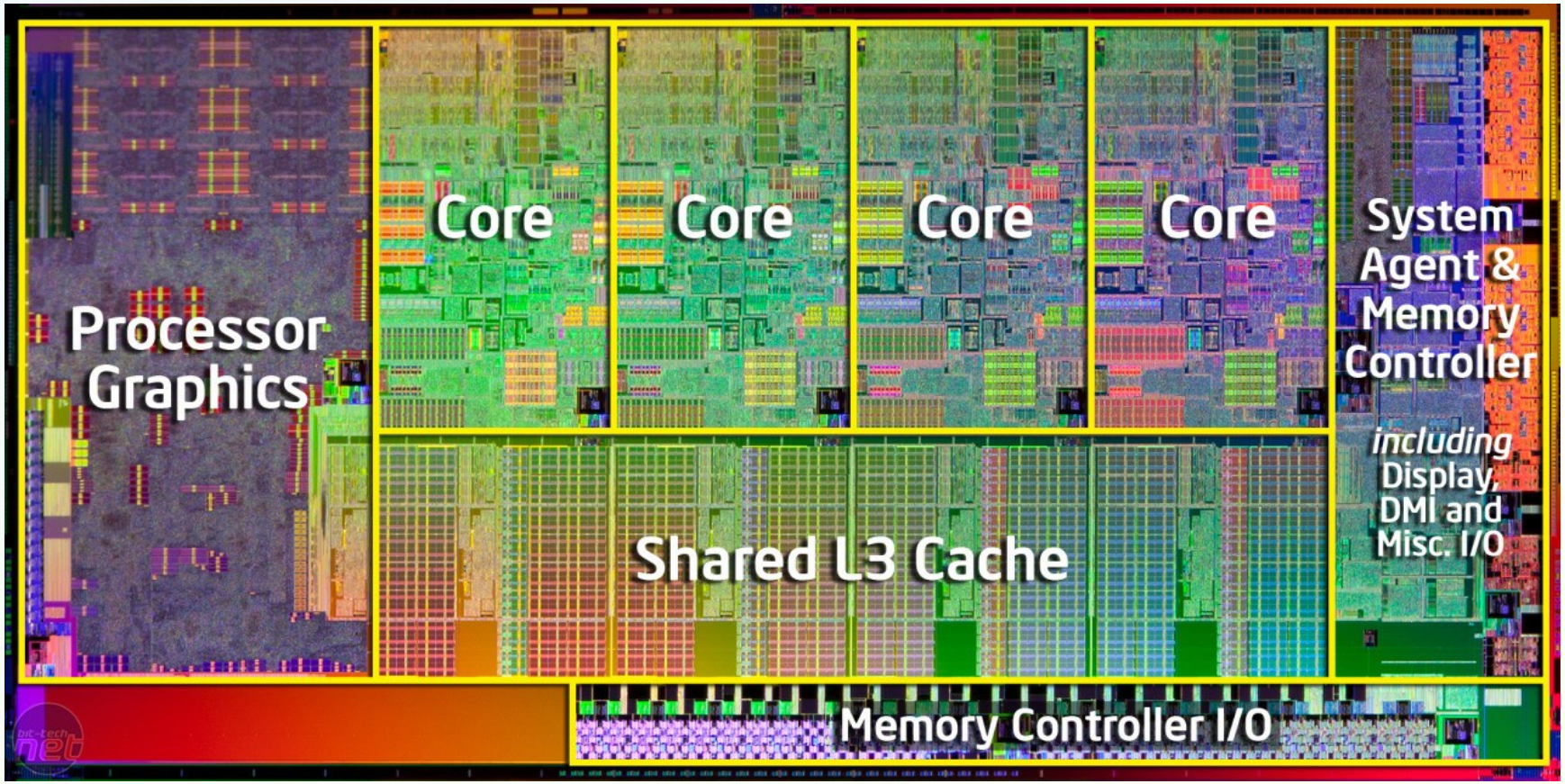
SYNCHRONIZATION: LOCAL COUNTER

```
1: AtomicLong ctr;
2:
3: long getNext() {
4:     return ctr.incrementAndGet();
5: }
```

SYNCHRONIZATION: LOCAL (RE-INVENTING A BICYCLE)

```
 1: AtomicLong ctr;
 2: ThreadLocal<Long> localCtr;
 3:
 4: long getNext() {
 5:     long res = localCtr.get();
 6:
 7:     if (res % 1000 == 0)
 8:         res = ctr.getAndAdd(1000);
 9:
10:     localCtr.set(++res);
11:
12:     return res;
13: }
```

SYNCHRONIZATION: LOCAL



SYNCHRONIZATION: DISTRIBUTED

SYNCHRONIZATION: COUNTER IN THE CLUSTER

Local implementation: millions ops/sec

Distributed implementation: thousands ops/sec

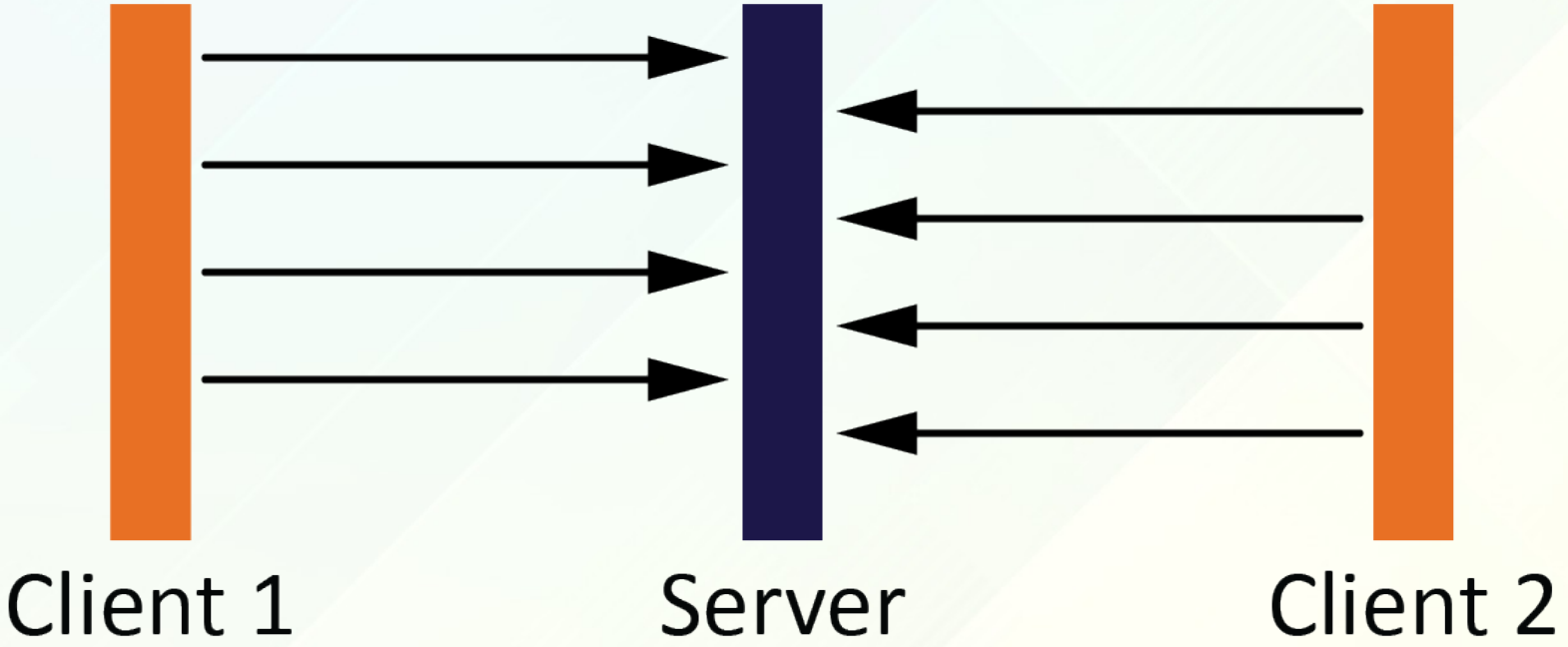
SYNCHRONIZATION: COUNTER IN THE CLUSTER

Proper requirements:

- Unique
- Monotonously growing
- 8 bytes

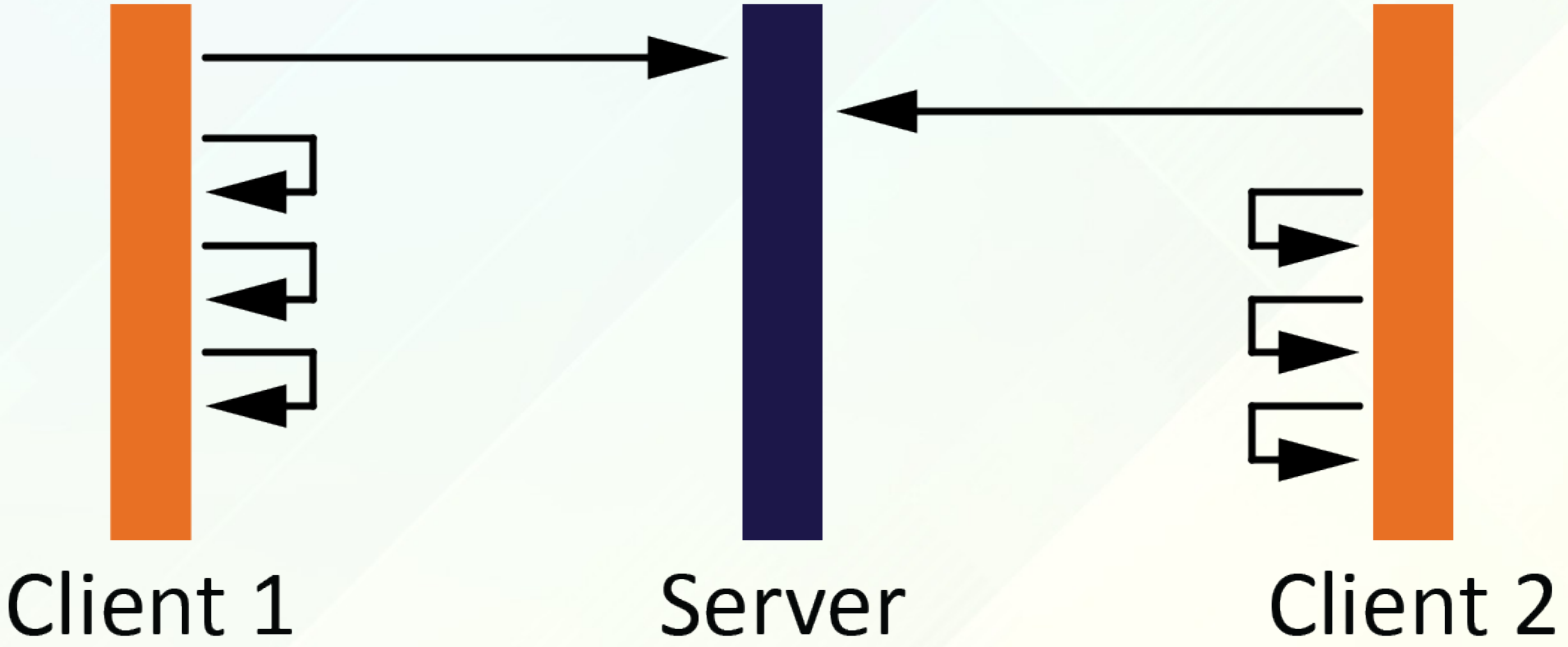
SYNCHRONIZATION: COUNTER IN THE CLUSTER

Requirements: unique, monotonous, 8 bytes.



SYNCHRONIZATION: COUNTER IN THE CLUSTER

Requirements: unique, ~~monotonous~~, 8 bytes.



SYNCHRONIZATION: COUNTER IN THE CLUSTER

Requirements: unique, ~~monotonous~~, ~~8 bytes~~.

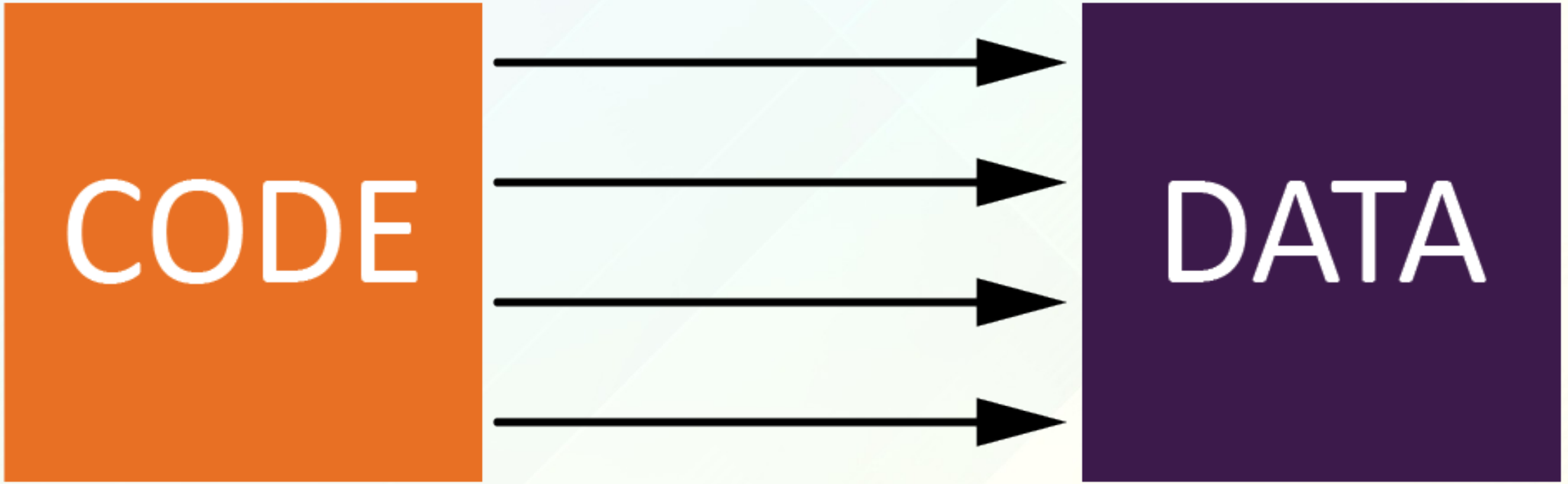
SYNCHRONIZATION: COUNTER IN THE CLUSTER

Requirements: unique, ~~monotonous~~, ~~8 bytes~~.

See also: `org.apache.ignite.lang.IgniteUuid`

SYNCHRONIZATION AS FRICTION FOR A CAR

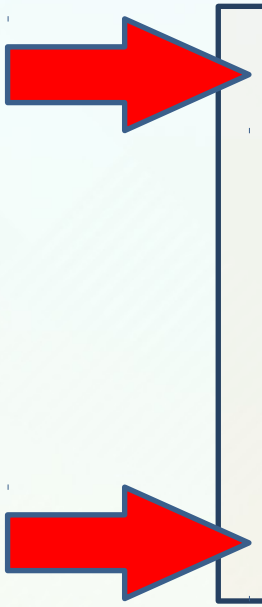
SYNCHRONIZATION: DATA TO CODE



SYNCHRONIZATION: DATA TO CODE

```
1: Account acc = cache.get(accKey);
3:
3: acc.add(100);
4:
5: cache.put(accKey, acc);
```

SYNCHRONIZATION: DATA TO CODE



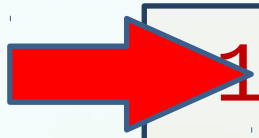
```
1: Account acc = cache.get(accKey);
3:
3: acc.add(100);
4:
5: cache.put(accKey, acc);
```

SYNCHRONIZATION: CODE TO DATA

SYNCHRONIZATION: CODE TO DATA

```
1: cache.invoke(accKey, (entry) -> {
1:     Account acc = entry.getValue();
3:
3:     acc.add(100);
4:
5:     entry.setValue(acc);
6: });
```

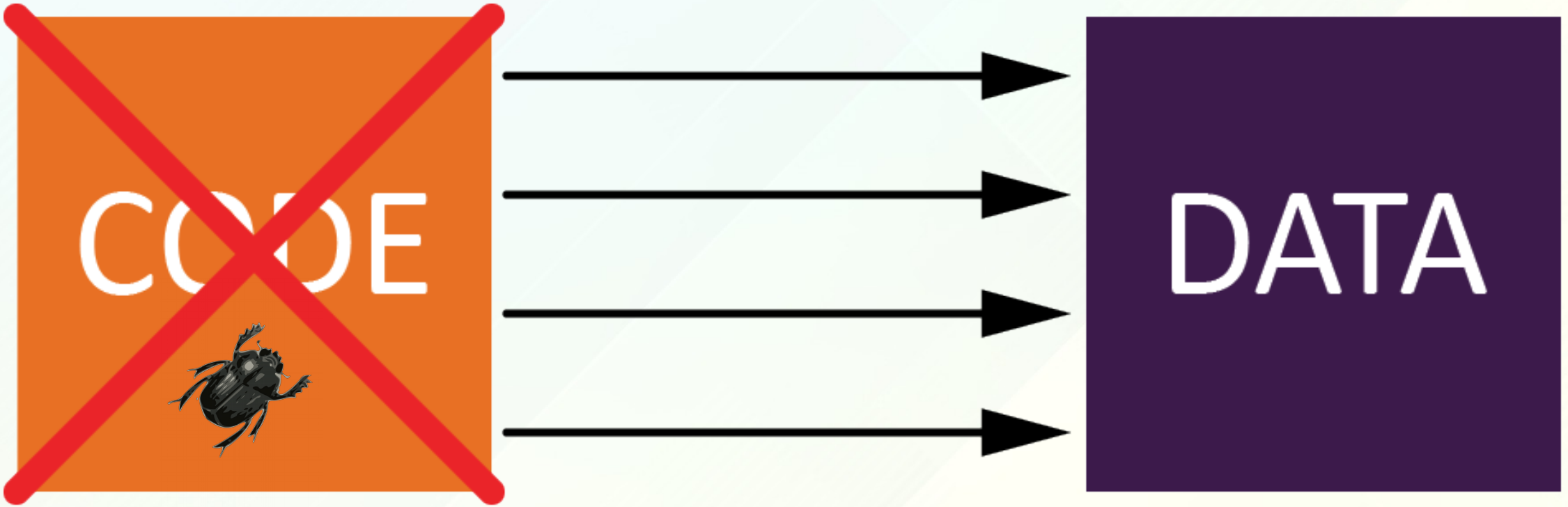
SYNCHRONIZATION: CODE TO DATA



```
1: cache.invoke(accKey, (entry) -> {
1:     Account acc = entry.getValue();
3:
3:     acc.add(100);
4:
5:     entry.setValue(acc);
6: });
```

SYNCHRONIZATION: DATA TO CODE

What if we have a bug?!

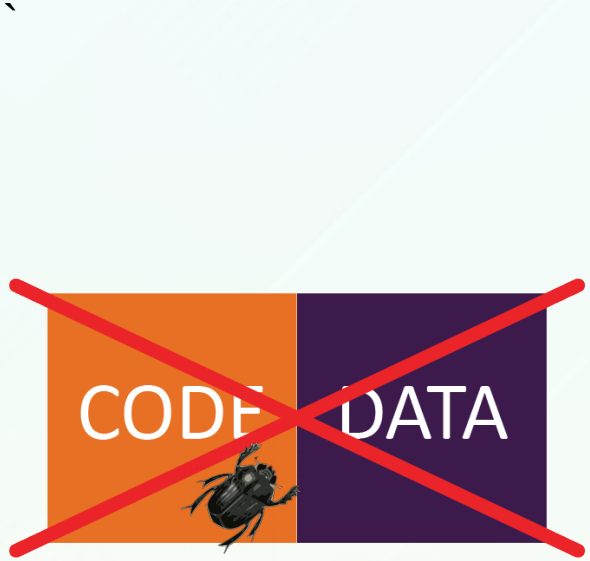


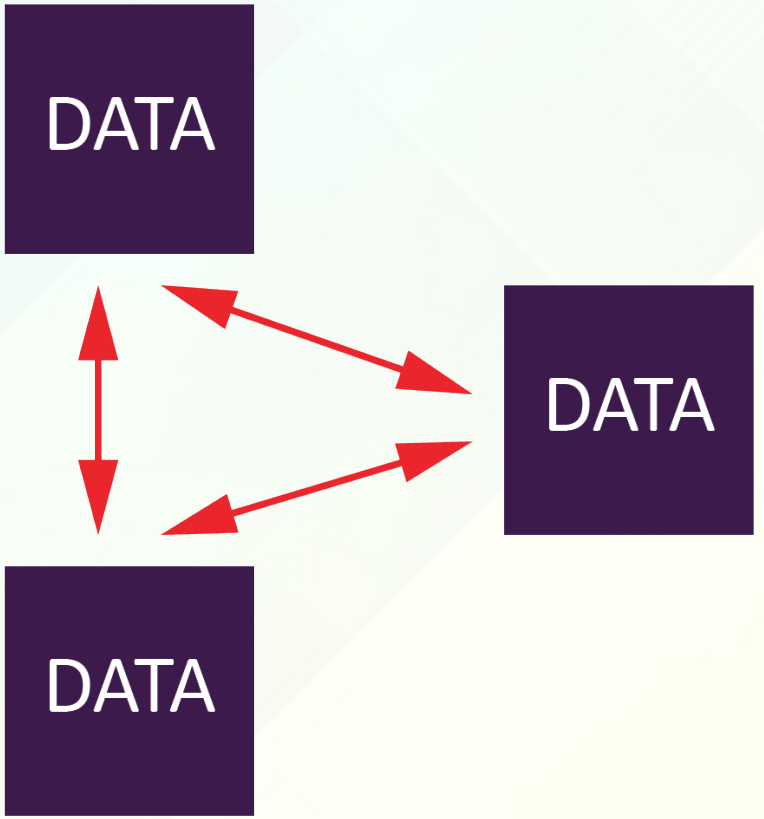
SYNCHRONIZATION: CODE TO DATA

What if we have a bug?!

SYNCHRONIZATION: CODE TO DATA

What if we have a bug?!

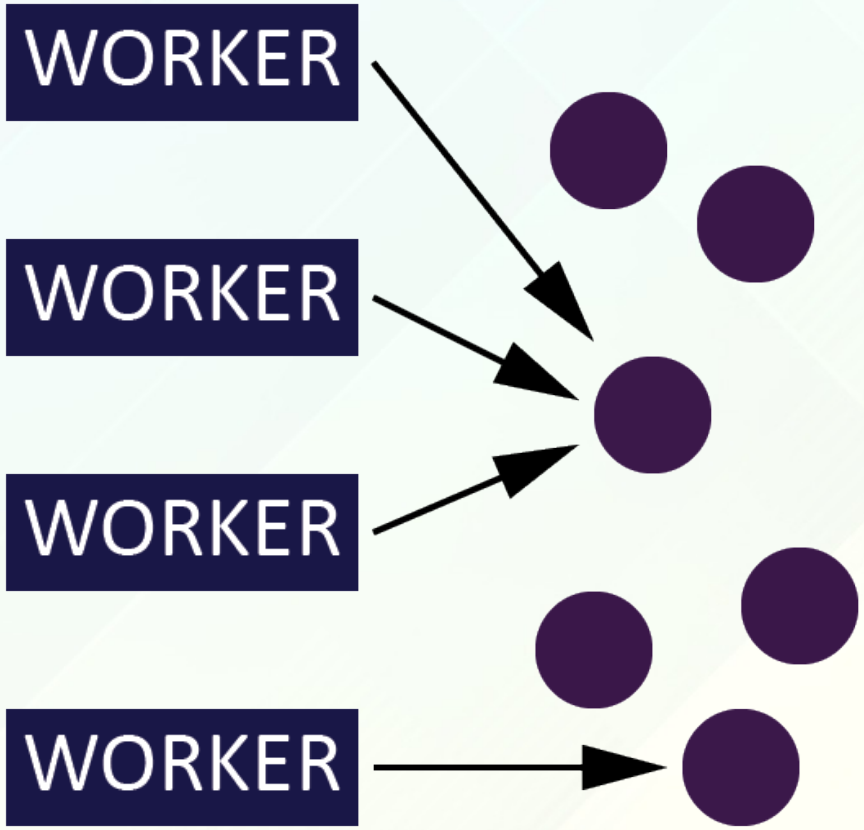




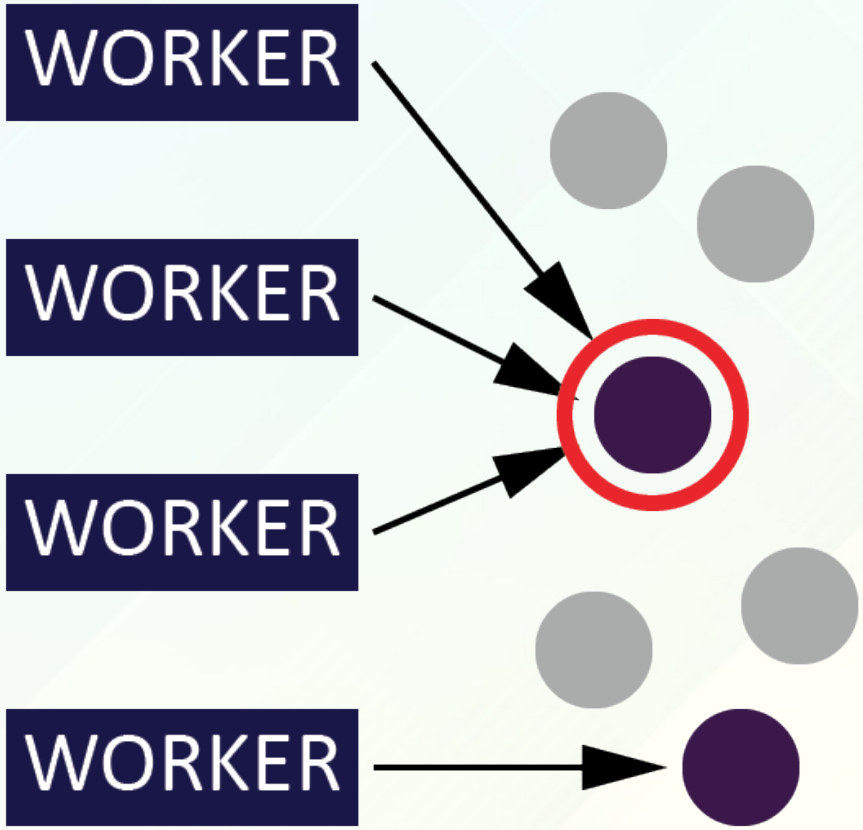
PLAN

1. Data partitioning and affinity functions examples
2. Data affinity colocation
3. Synchronization in distributed systems
4. Multithreading: local architecture

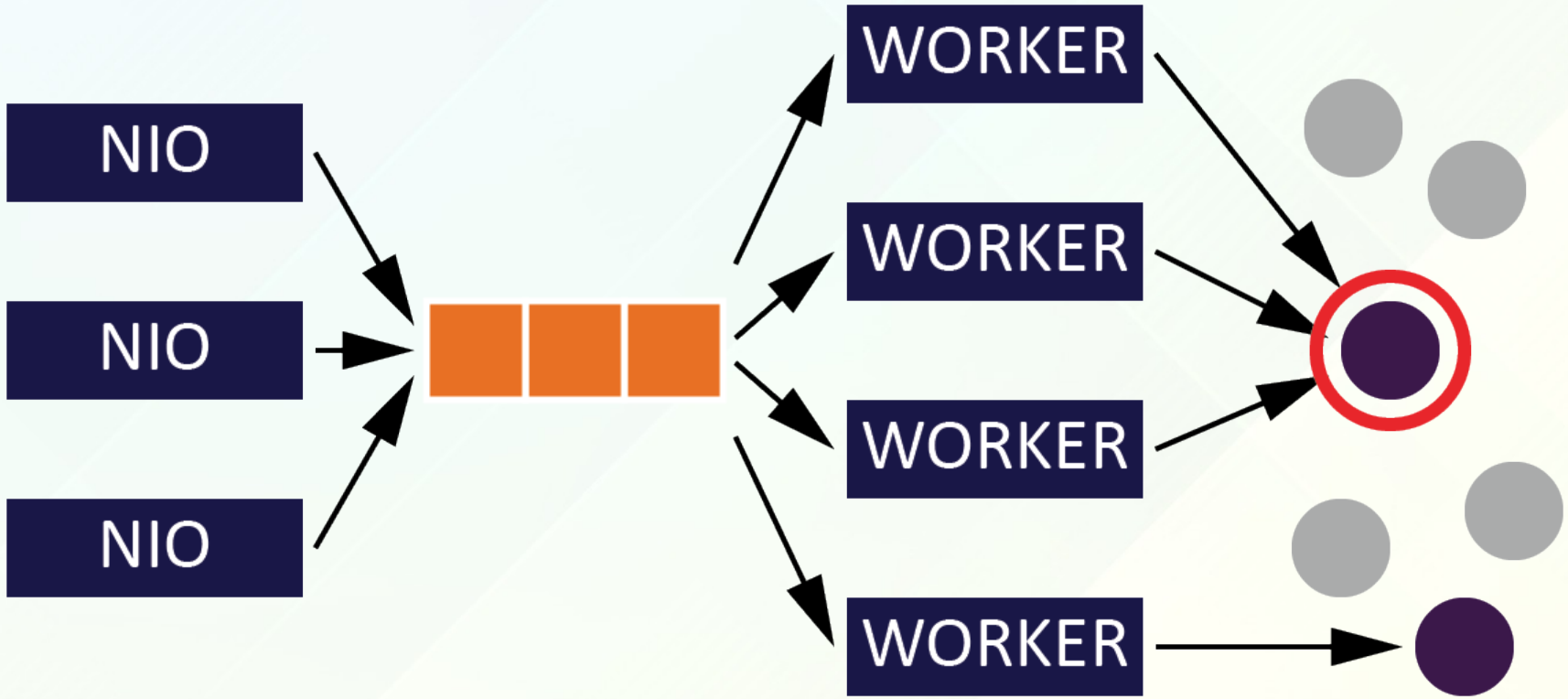
LOCAL TASKS DISTRIBUTION



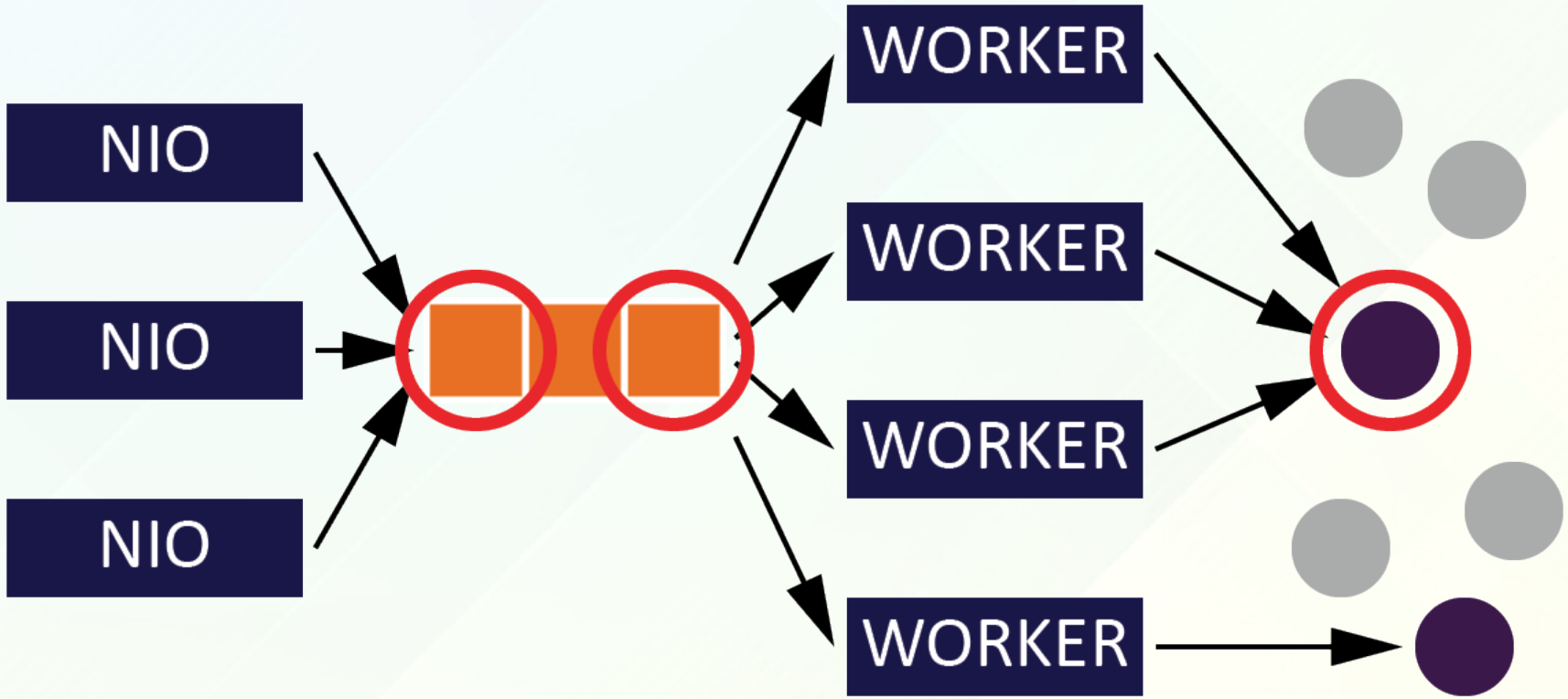
LOCAL TASKS DISTRIBUTION



LOCAL TASKS DISTRIBUTION

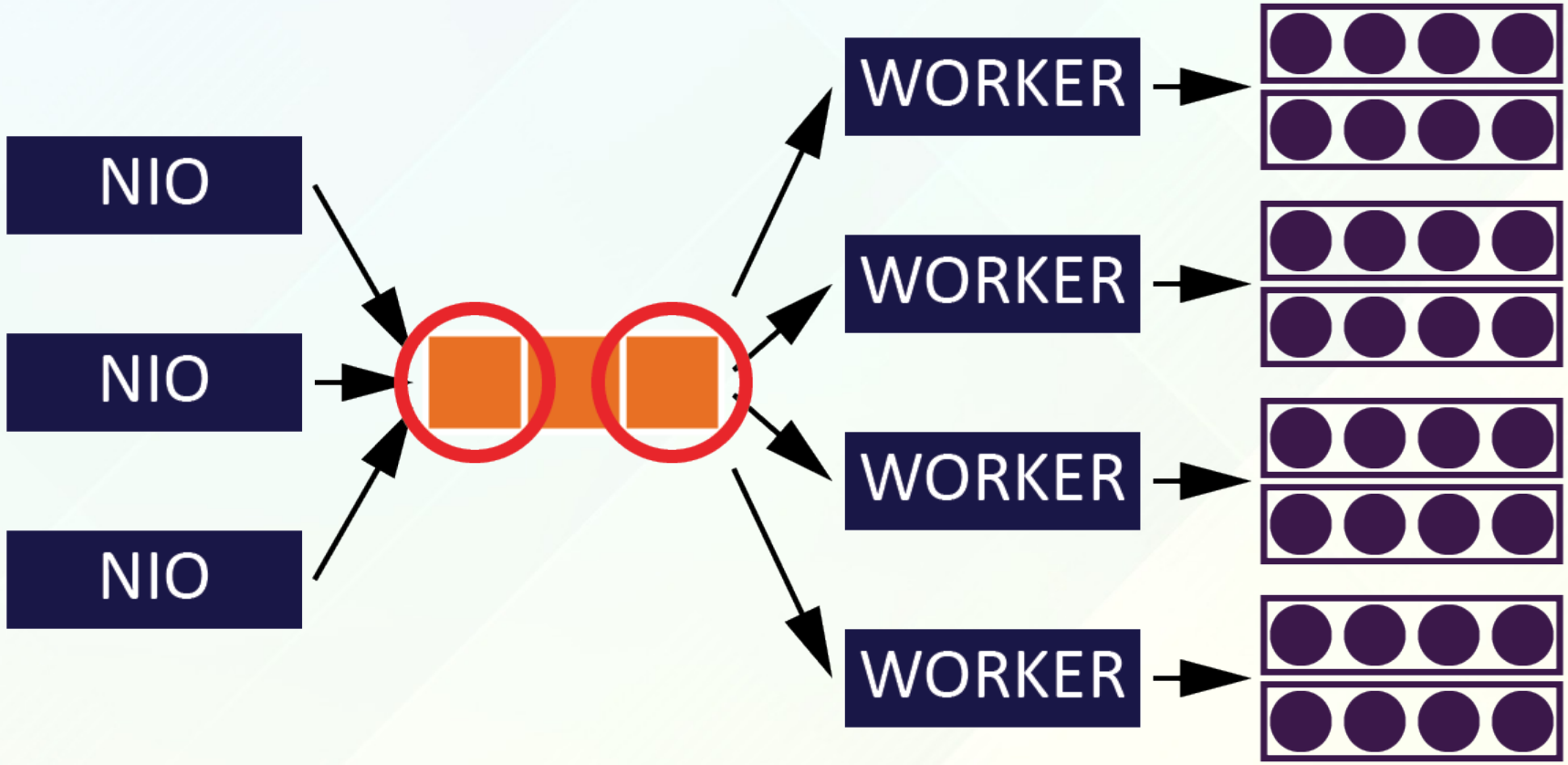


LOCAL TASKS DISTRIBUTION

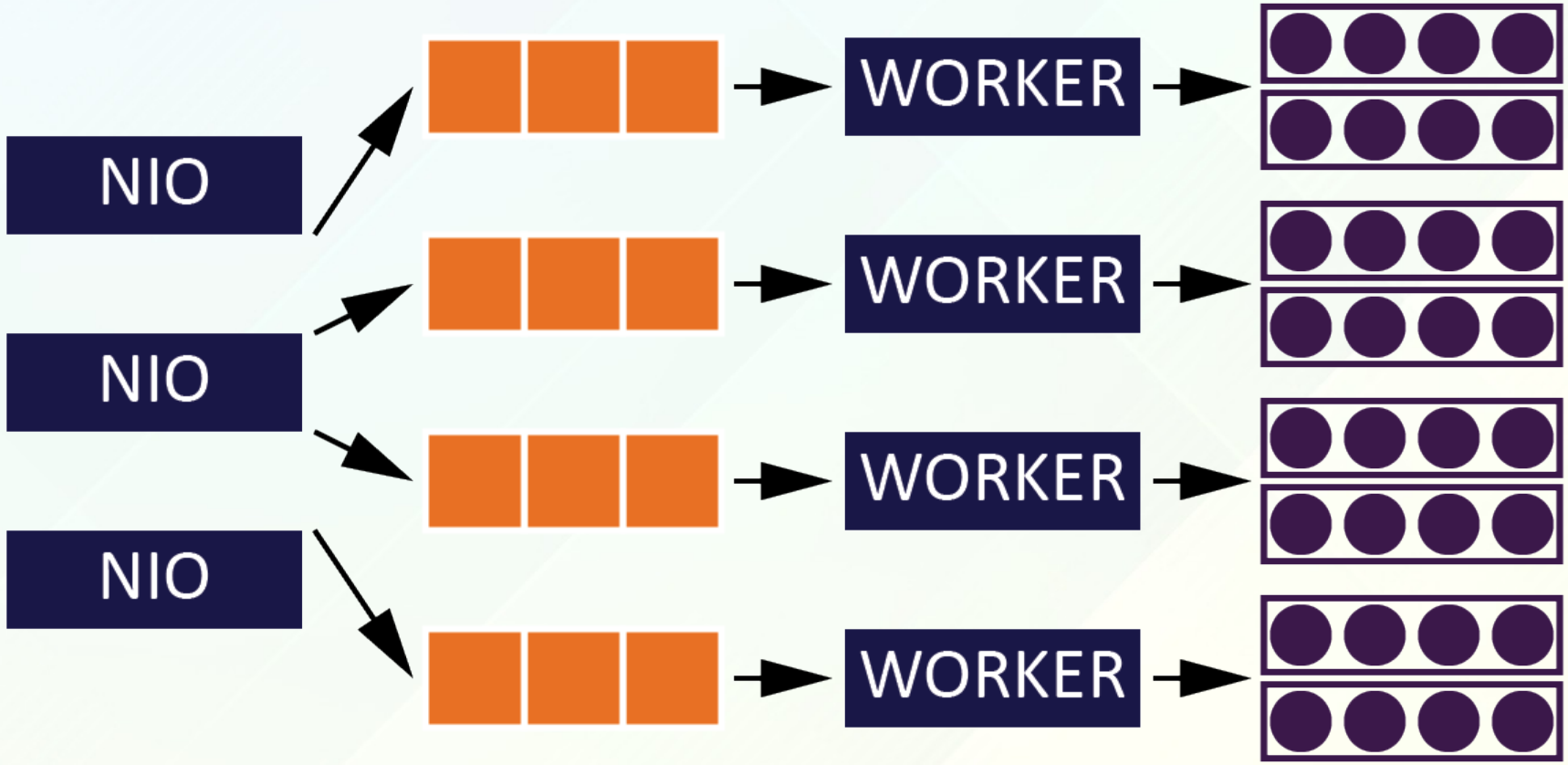


LOCAL TASKS DISTRIBUTION

LOCAL TASKS DISTRIBUTION: THREAD PER PARTITION



LOCAL TASKS DISTRIBUTION: THREAD PER PARTITION



LOCAL TASKS DISTRIBUTION: THREAD PER PARTITION

Node 1

100 TPS

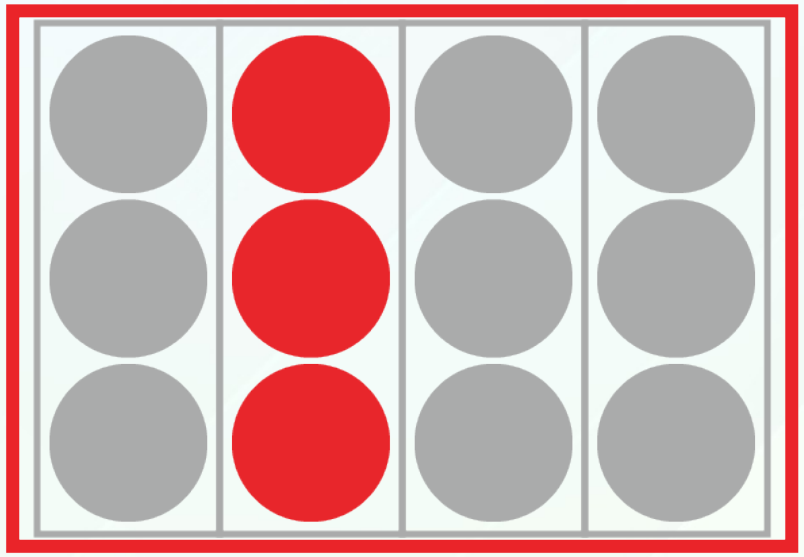
Node 2

50 TPS

Node 3

10 TPS

LOCAL TASKS DISTRIBUTION: THREAD PER PARTITION



Node 1

100 TPS

SPB

Node 2

50 TPS

NSK

Node 3

10 TPS

LESSONS LEARNED

1) **Data partitioning:** balance and stability

LESSONS LEARNED

1) **Data partitioning:** balance and stability
2) **Colocation:** balance and efficiency

LESSONS LEARNED

1) **Data partitioning:** balance and stability
2) **Colocation:** balance and efficiency
3) **Data model:** should be adopted accordingly

LESSONS LEARNED

1) **Data partitioning:** balance and stability
2) **Colocation:** balance and efficiency
3) **Data model:** should be adopted accordingly
4) **Synchronization:** delicate and only when really needed

LESSONS LEARNED

1) **Data partitioning:** balance and stability
2) **Colocation:** balance and efficiency
3) **Data model:** should be adopted accordingly
4) **Synchronization:** delicate and only when really needed
5) **Thread per partition:** can improve simple operations, but also may slow down complex ones

CONTACTS

yzhdanov@gridgain.com

http://ignite.apache.org

dev@ignite.apache.org
user@ignite.apache.org

QUESTIONS?

ANY QUESTIONS?