Dynamo

Big Data Systems

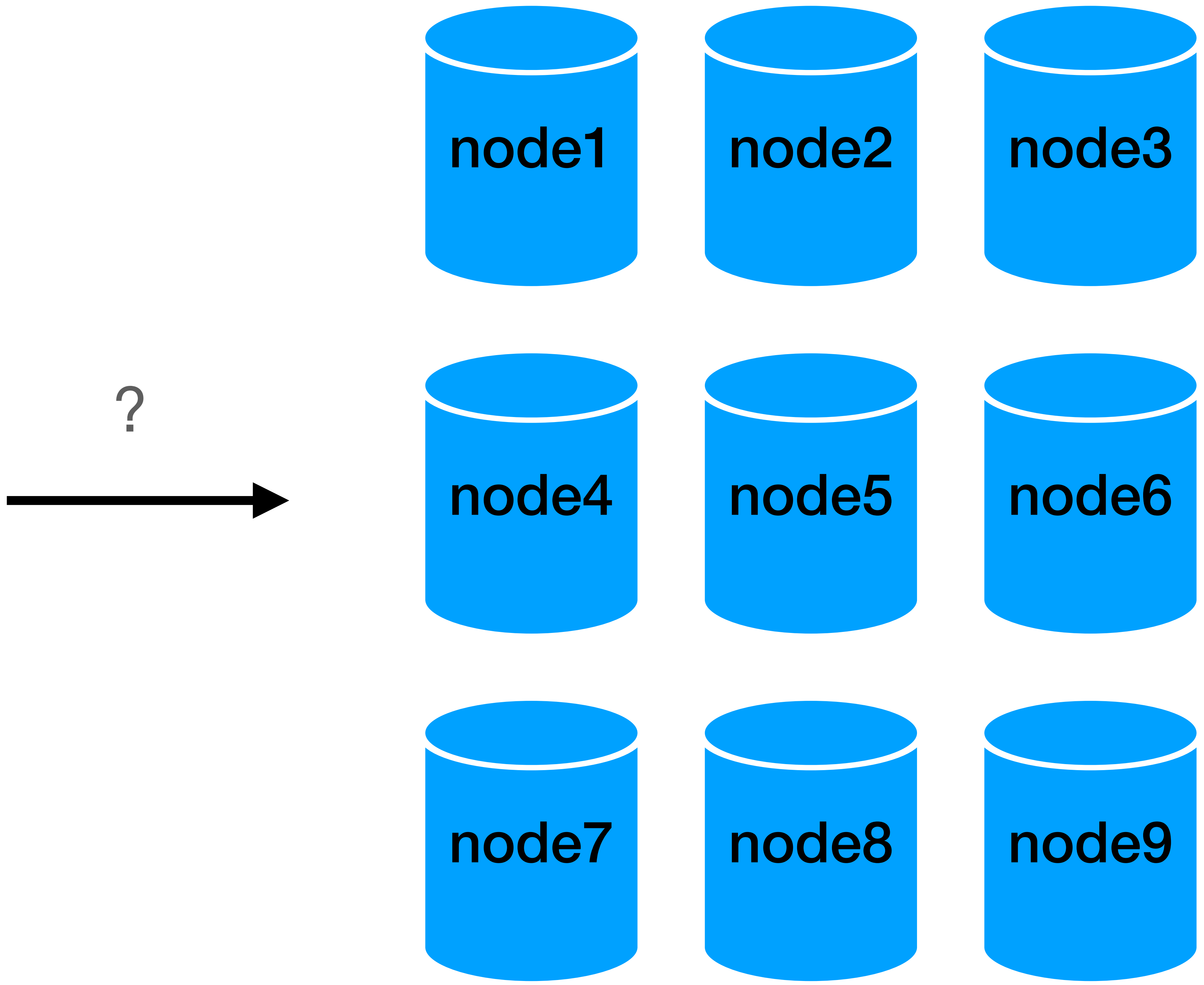
Dr. Rubi Boim

A quick reminder / motivation

Previously - Going distributed

- **Not trivial…** :)

- Starting with:
  - Data fragmentation
  - Data distribution
  - Data replication



Data fragmentation (horizontal)

user_id	fname	lname	city	country	account	brithdate
101	Rubi	Boim	Tel Aviv	Israel	Normal	<null>
102	Tova	Milo	Tel Aviv	Israel	Premium	<null>
103	Lebron	James	Los Angeles	USA	Premium	30/12/1984
104	Michael	Jordan	Chicago	USA	Normal	17/02/1963

- Choose an attribute
- Assign a “range” to each “node”

node1

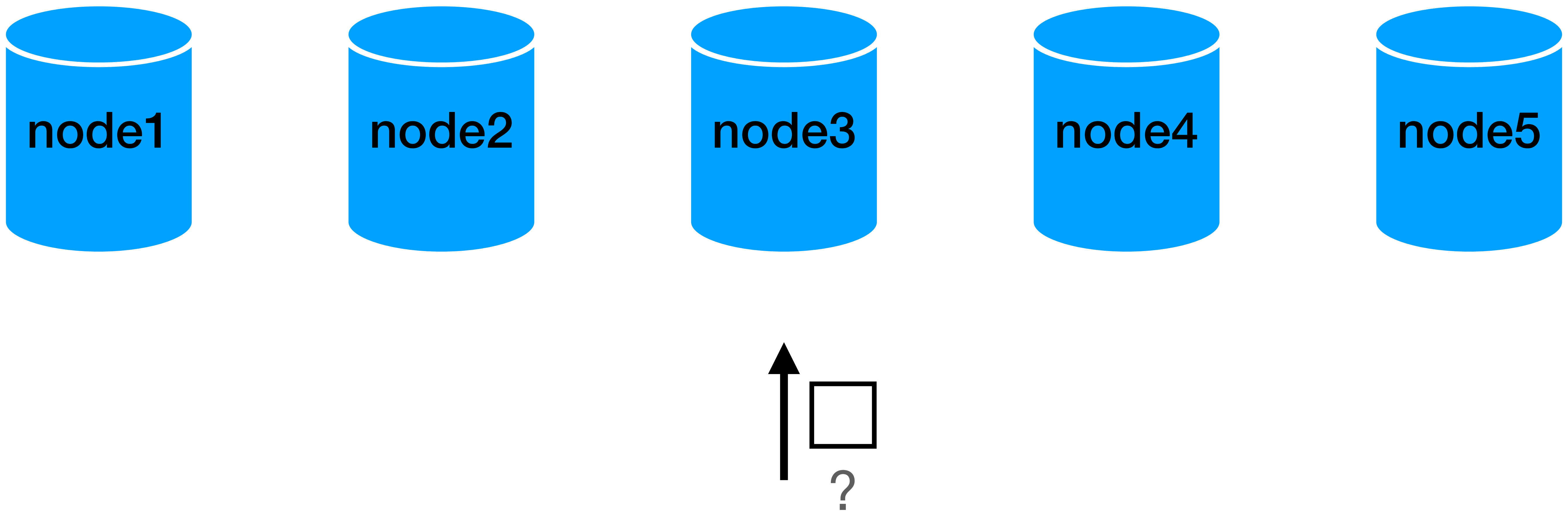
user_id	fname	lname	city	country	account	brithdate
101	Rubi	Boim	Tel Aviv	Israel	Normal	<null>
104	Michael	Jordan	Chicago	USA	Normal	17/02/1963

node2

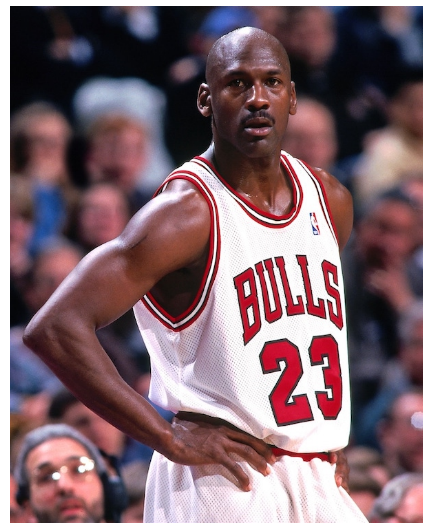
user_id	fname	lname	city	country	account	brithdate
102	Tova	Milo	Tel Aviv	Israel	Premium	<null>
103	Lebron	James	Los Angeles	USA	Premium	30/12/1984

Data distribution

- How can the DB decide where the data is located?

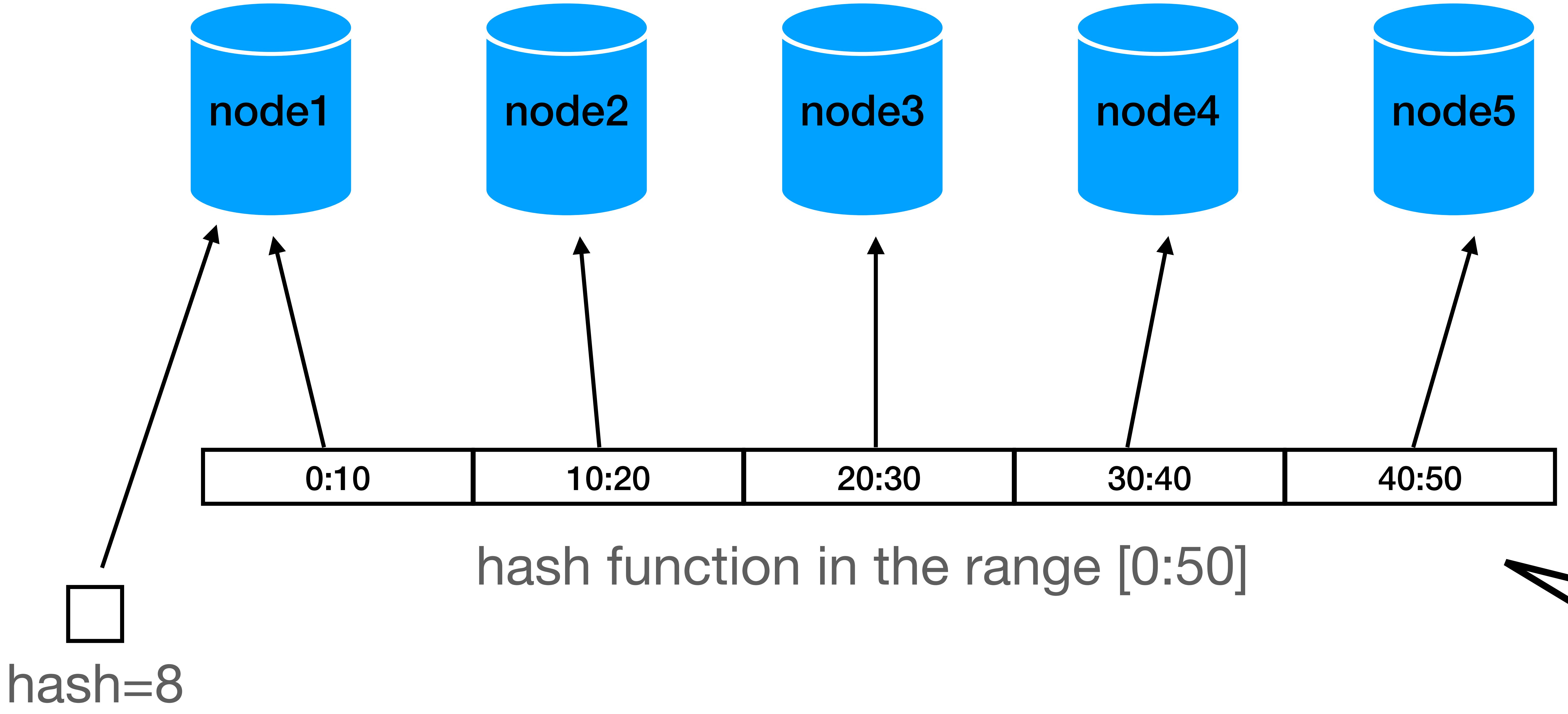


INSERT INTO users VALUES(x,y,z)



add new data /
query existing data

Data distribution- Range on hashes

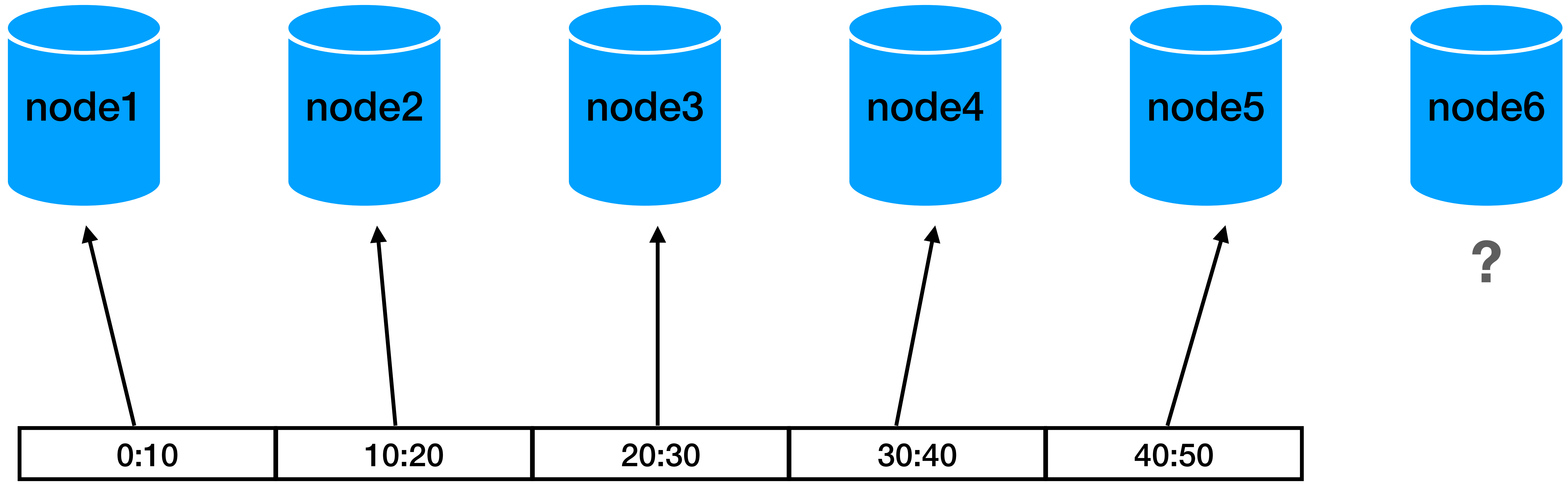


hash function in the range [0:50]

pros / cons?

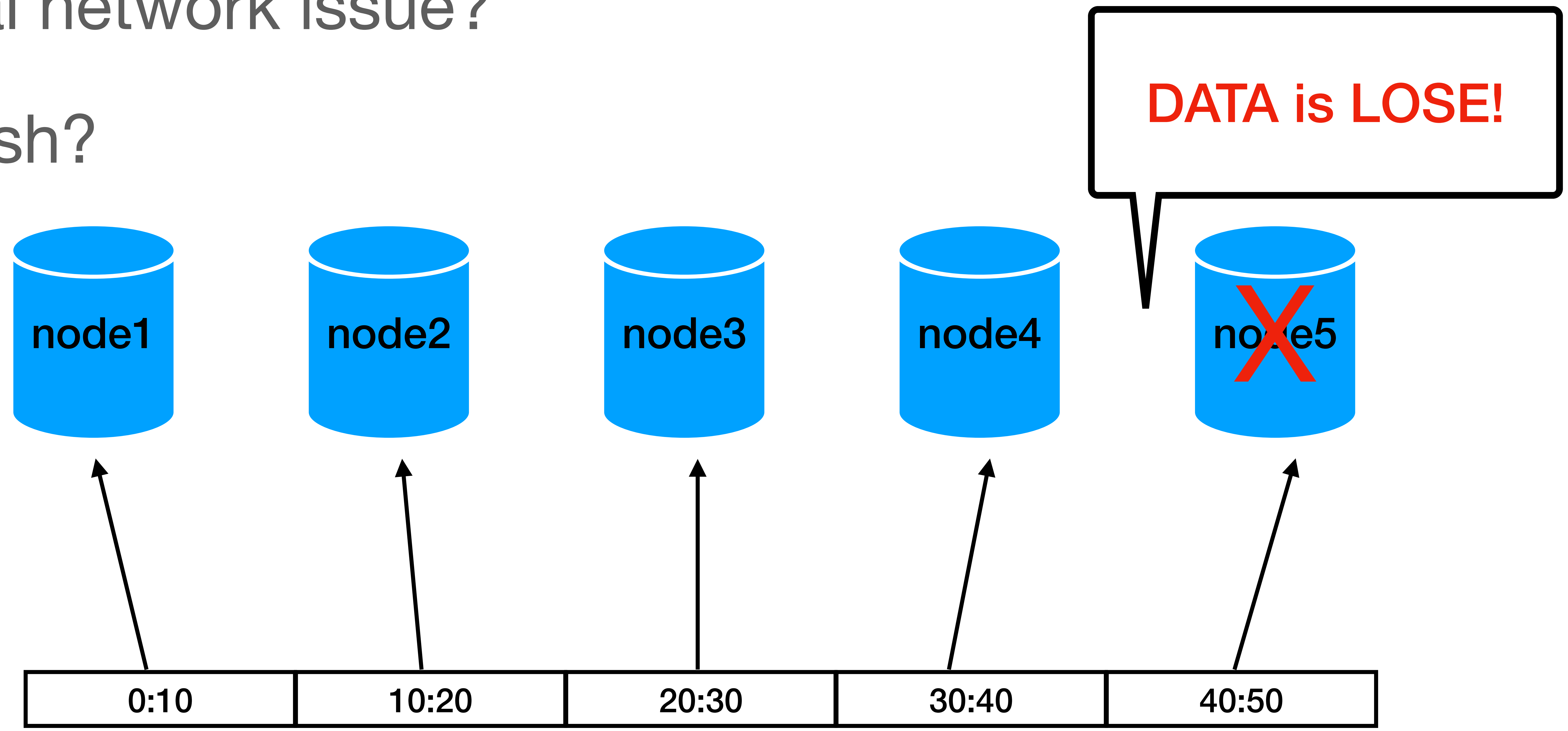
Data distribution - scaling

- What happens if we want to add a node?
  - new data?
  - existing data?



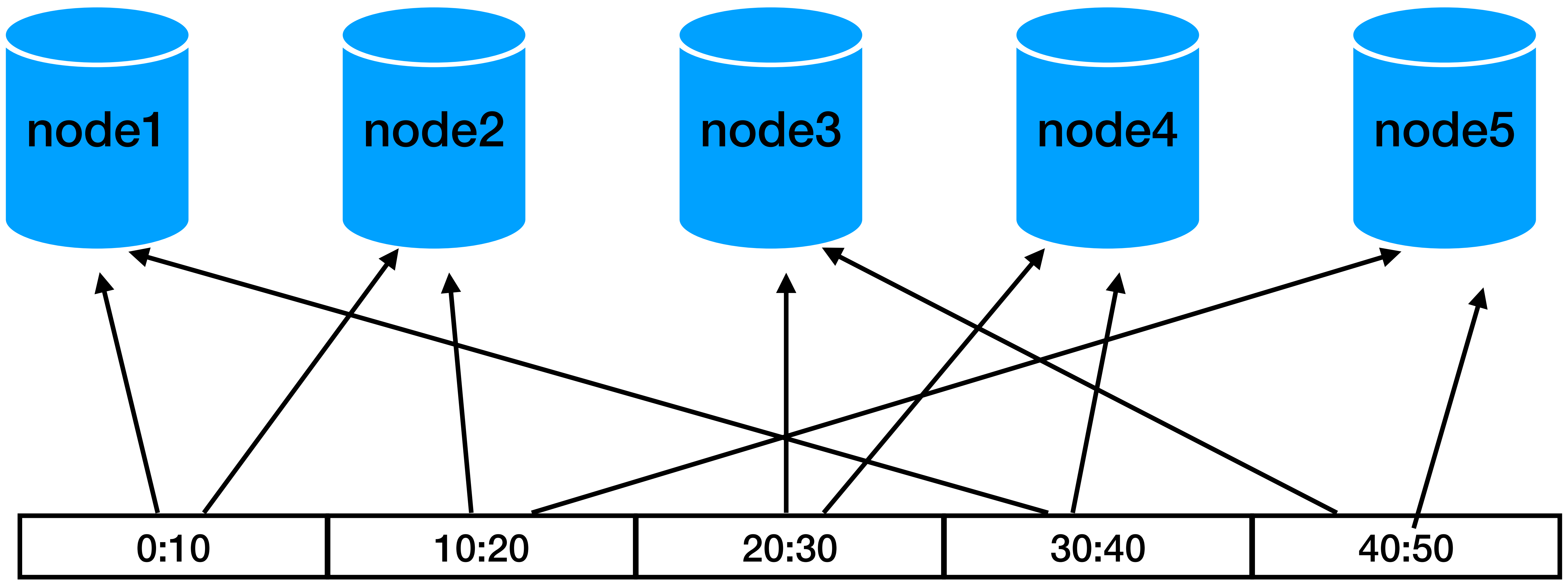
Stuff happens

- What happens if a node fails?
  - temporal network issue?
  - disk crash?



Data replication

- (re)distribute among all nodes



replication factor = 2

How do we manage all this?

and much more

Dynamo

- Create by Amazon in 2007
  paper: Dynamo: Amazon’s Highly Available Key-value Store
- The techniques developed here are used in many other systems
  not just NoSQL and not just by Amazon

Requirement: Key-Value store

- `put(key, object)`
- `get(key)`

- Sounds simple.
- How would you implement it? Single server?

Dynamo topics for today

- **Requirements**
- Partition algorithm
- Replication
- Data versioning
- `get()` and `put()` execution
- Failures
- Ring membership

Requirements (1)

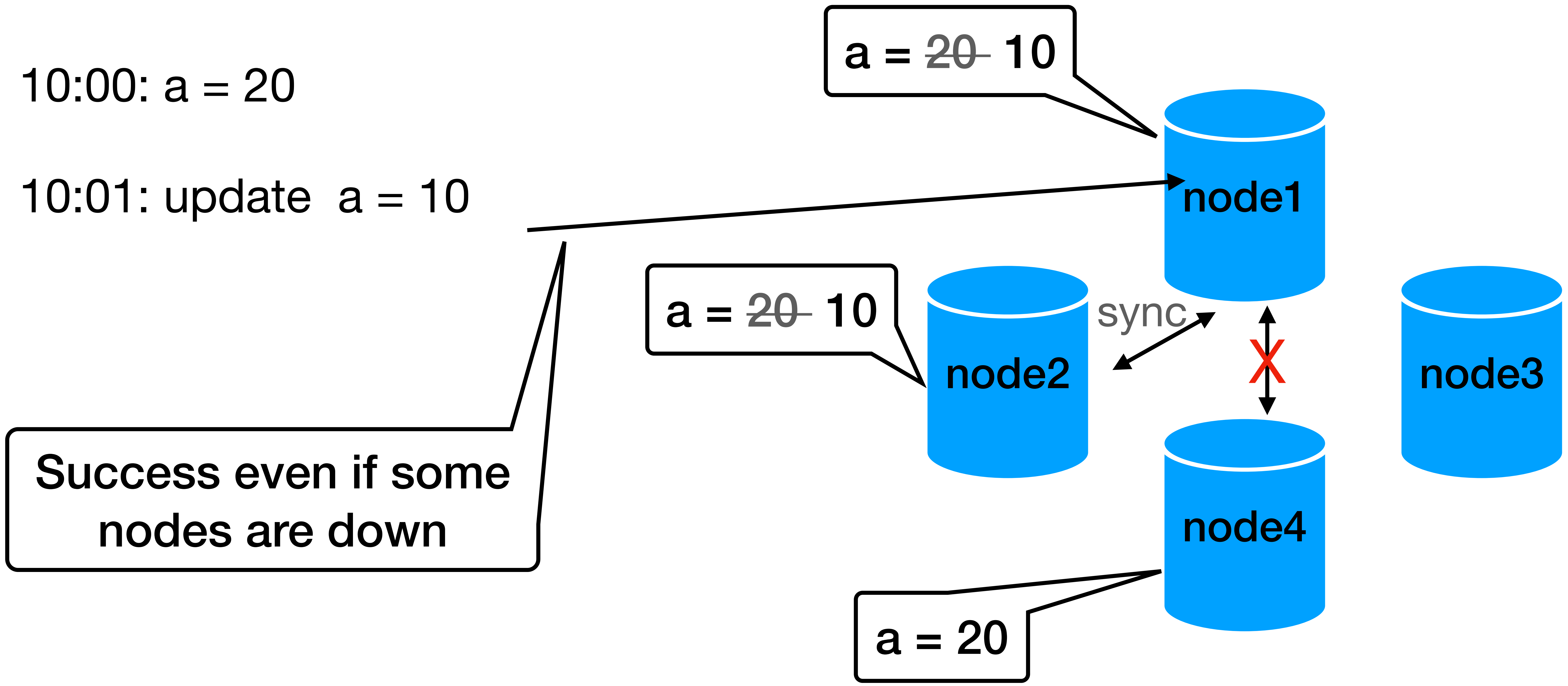
Incremental scalability

- scale out one node at a time
- support thousands of servers, multi data centers

Requirements (2)

Highly available

- “always writable” data store



10:00: a = 20

10:01: update a = 10

Success even if some nodes are down

a = ~~20~~ 10

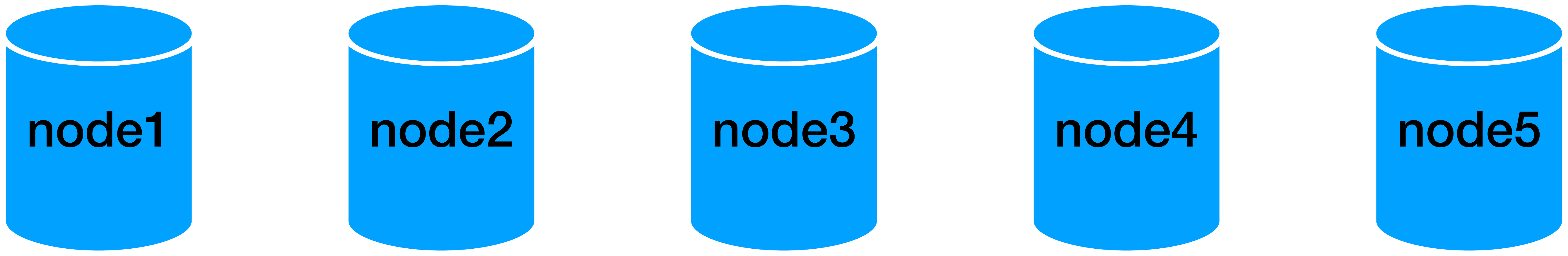
a = ~~20~~ 10

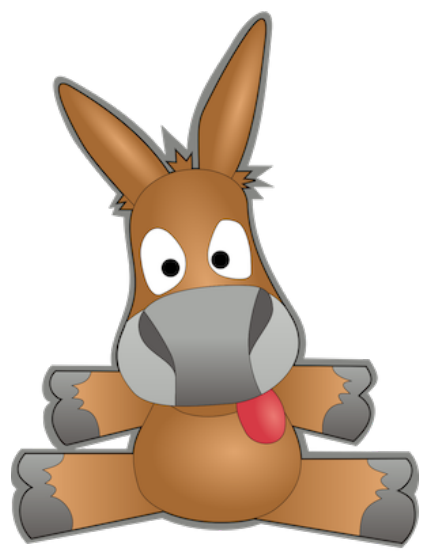
a = 20

Requirements (3)

Decentralized / Symmetry

- all nodes are equal, **no master** / SPOF

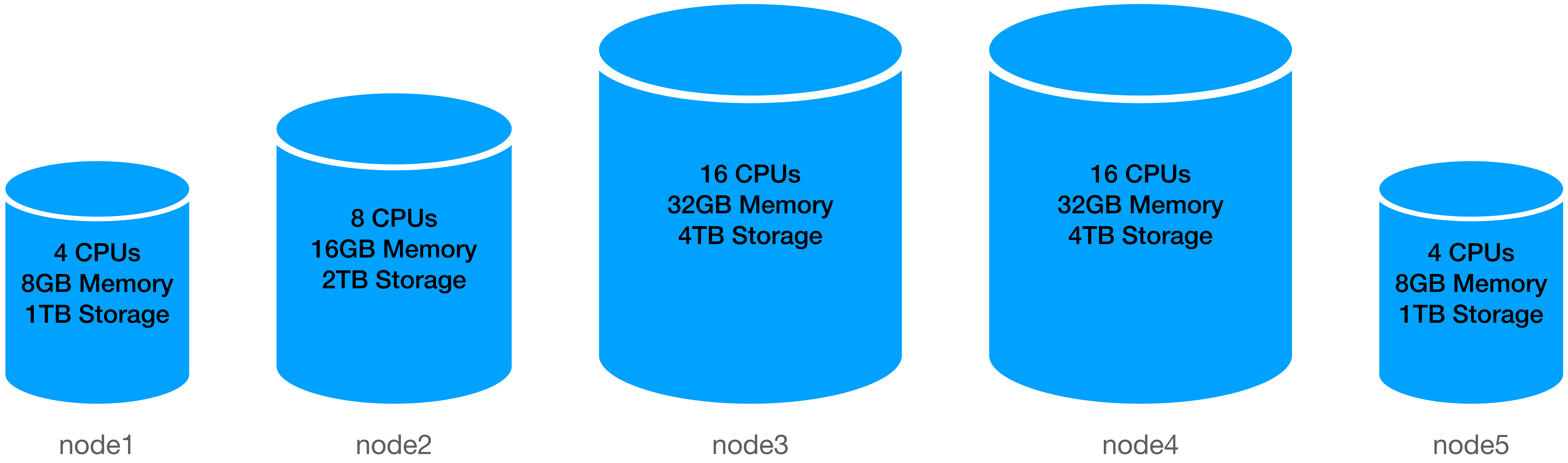




Requirements (4)

Node heterogeneity

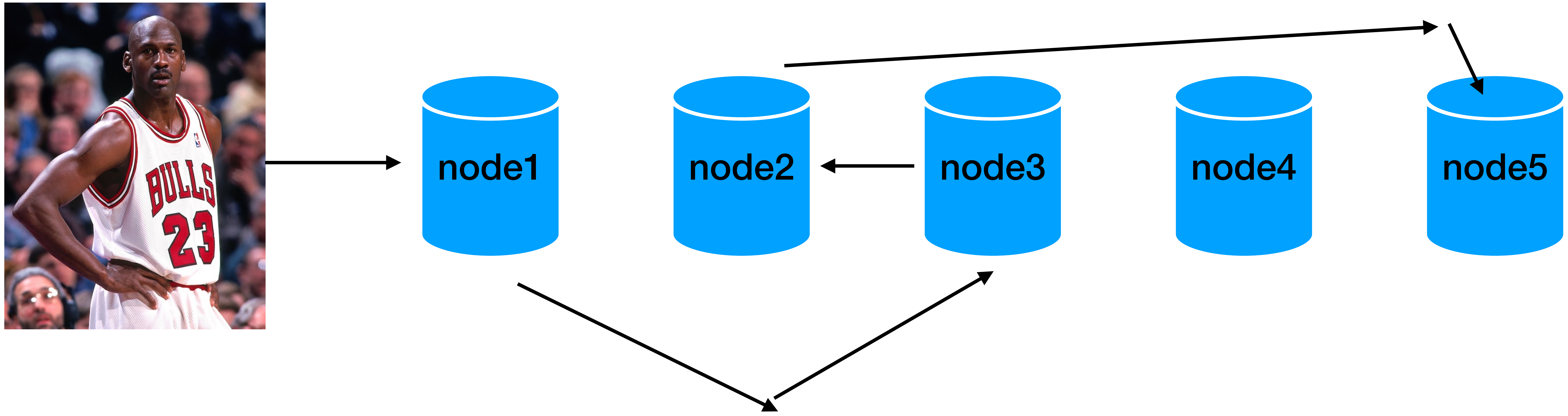
- work distribution must be proportional to the capabilities of each node



Requirements (5)

Performance

- 99.9% with 300 milliseconds response
  —> **avoid** routing request through multiple nodes as used in P2P DHT (distributed hash table) such as Chord or Pastry



Requirements (all together)

- Incremental scalability

  scale out one node at a time
  support thousands of servers, multi data centers

- Highly available
  "always writable" data store

- Decentralized / Symmetry
  all nodes are equal, **no master** / SPOF

- Node heterogeneity
  work distribution must be proportional to the capabilities of each node

- Performance
  99.9% with 300 milliseconds response
  —> avoid routing request through multiple nodes as
  used in P2P DHT (distributed hash table) such as Chord or Pastry

Requirements: Interface

- `put(key, context, object)`
- `get(key)`
  - `context` = system metadata / versioning (opaque to the user)
  - `get` returns <u>all versions</u> of the associated object
    * we will later see when can we have multi versions

Dynamo topics

- Requirements
- **Partition algorithm**
- Replication
- Data versioning
- `get()` and `put()` execution
- Failures
- Ring membership

Partitioning algorithm (1)

- Scale incrementally —>
  a mechanism is required to dynamically partition the data over a set of nodes

- How do we match nodes and keys (hashes)?

Partitioning algorithm (1) - side note

Ring —> Xbox360 technical problems

In December 2021 Microsoft started to sell “Red Ring of Death” posters…

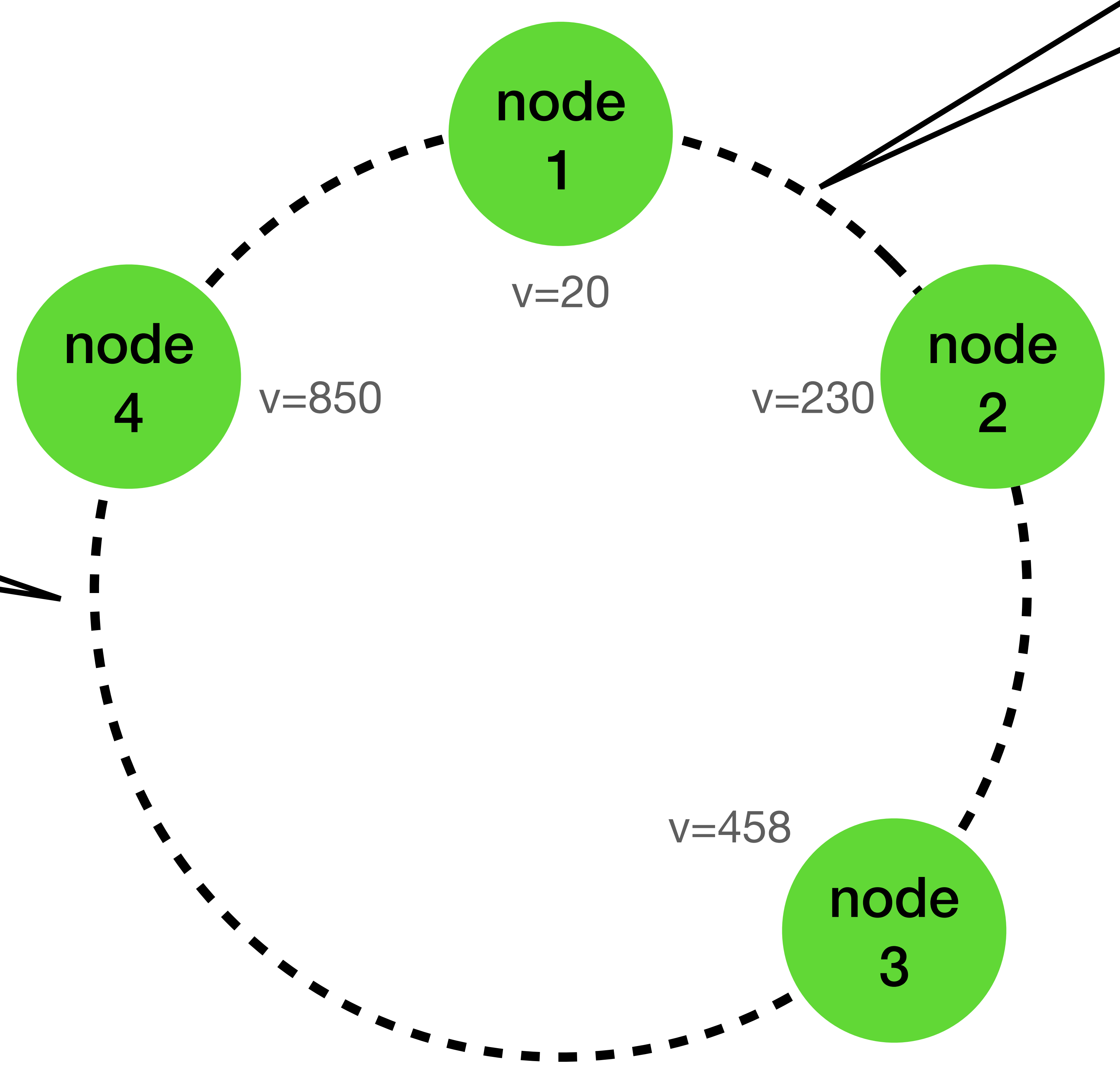
Partitioning algorithm (2)

Consistent hashing

- Hash function output is treated as a “ring”
- Each node is assigned a random value within the space (“location on the ring”)
- Assignment to a node is done by taking the hash of the key and “walking (clockwise) on the ring till a node”

Partitioning algorithm (3)

Consistent hashing



Range is [0:1000] for the example

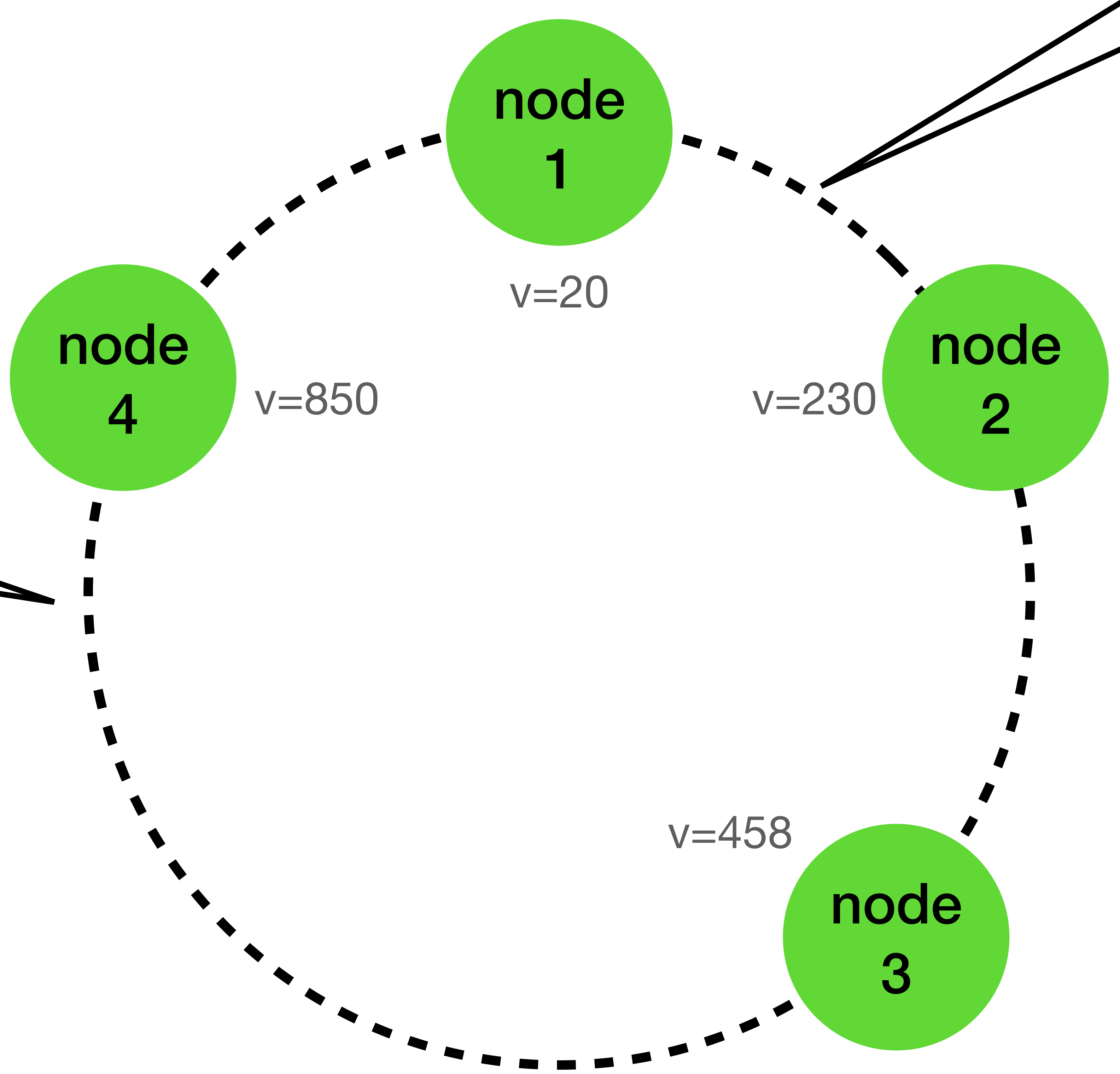
Each node is responsible to the range between it and its predecessor

Partitioning algorithm (3)

Consistent hashing

Range is [0:1000] for the example

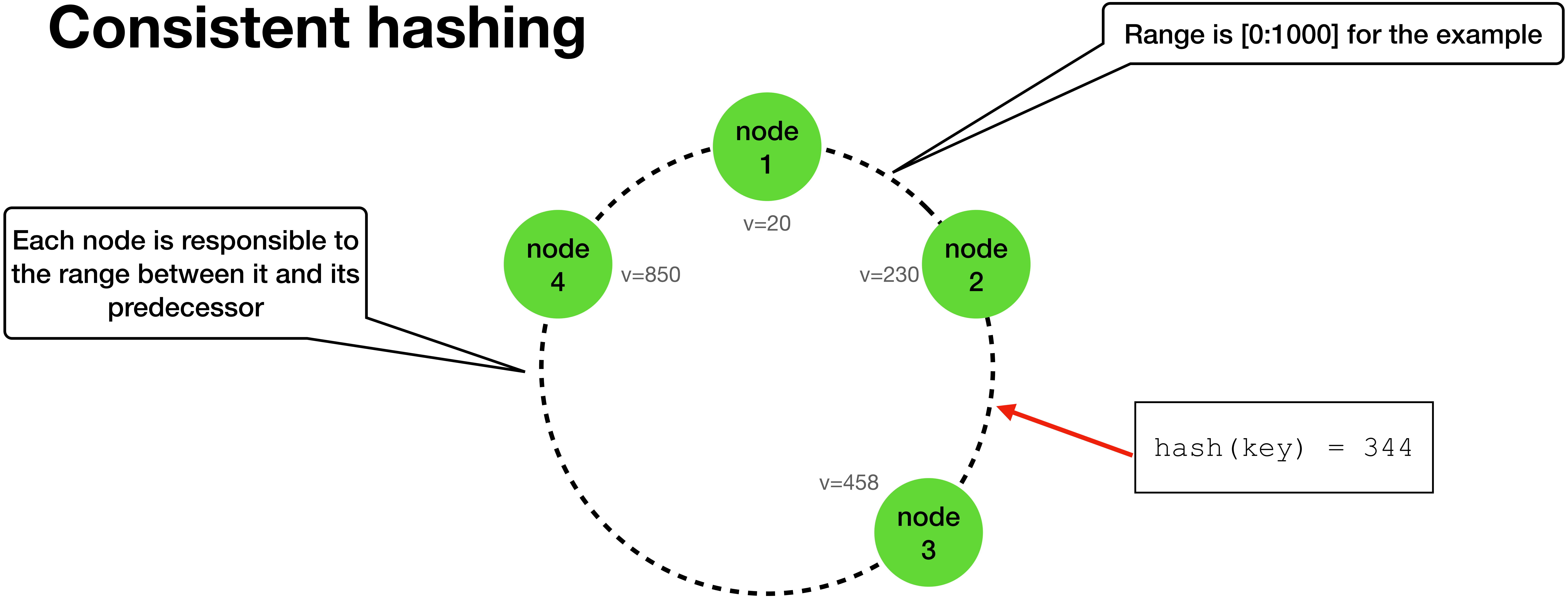
Each node is responsible to the range between it and its predecessor



```
hash(key) = 344
```

Partitioning algorithm (3)

Consistent hashing



Range is [0:1000] for the example

Each node is responsible to the range between it and its predecessor

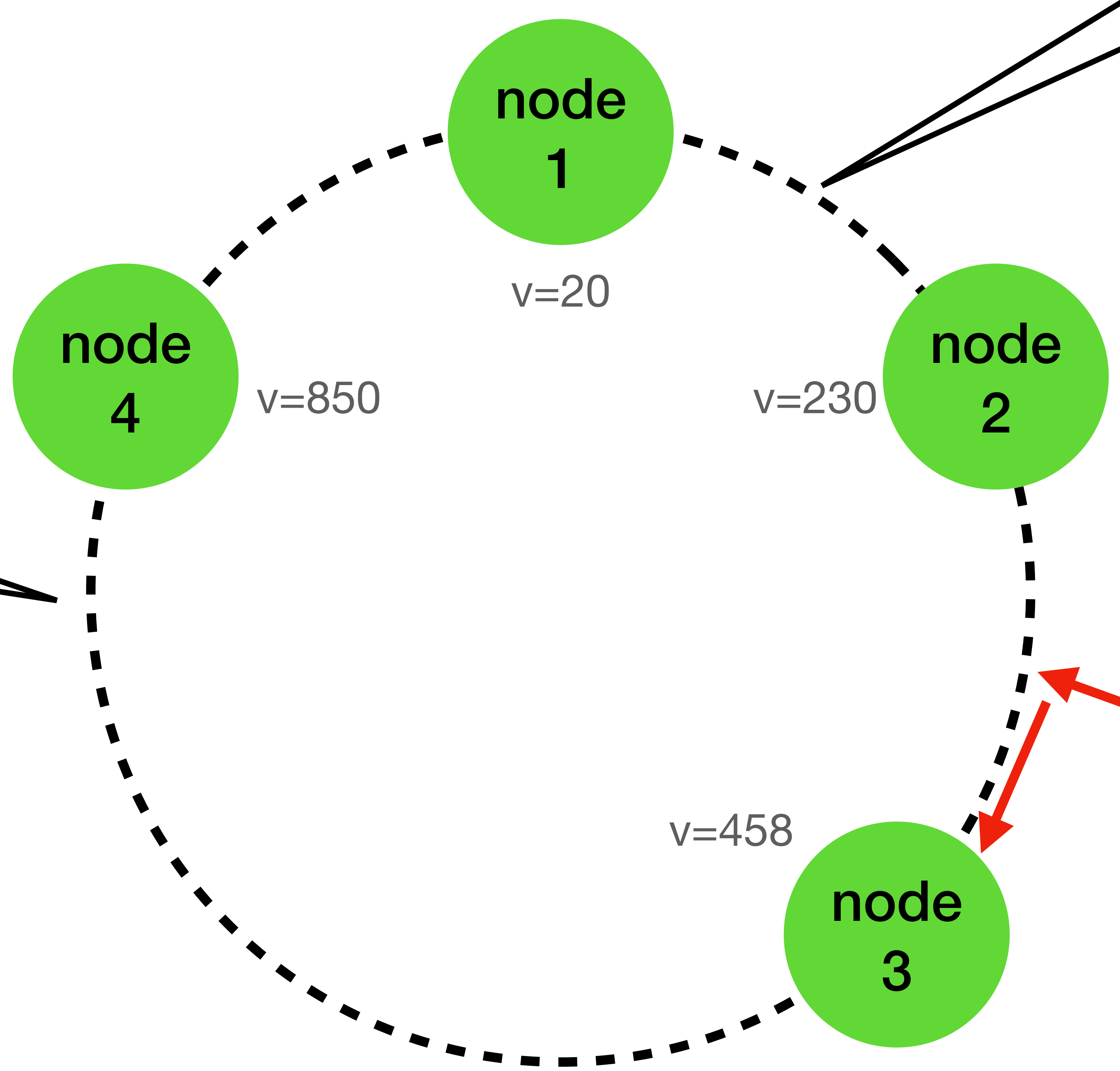
`hash(key) = 344`

Partitioning algorithm (3)

Consistent hashing

Range is [0:1000] for the example

Each node is responsible to the range between it and its predecessor



```
hash(key) = 344
```

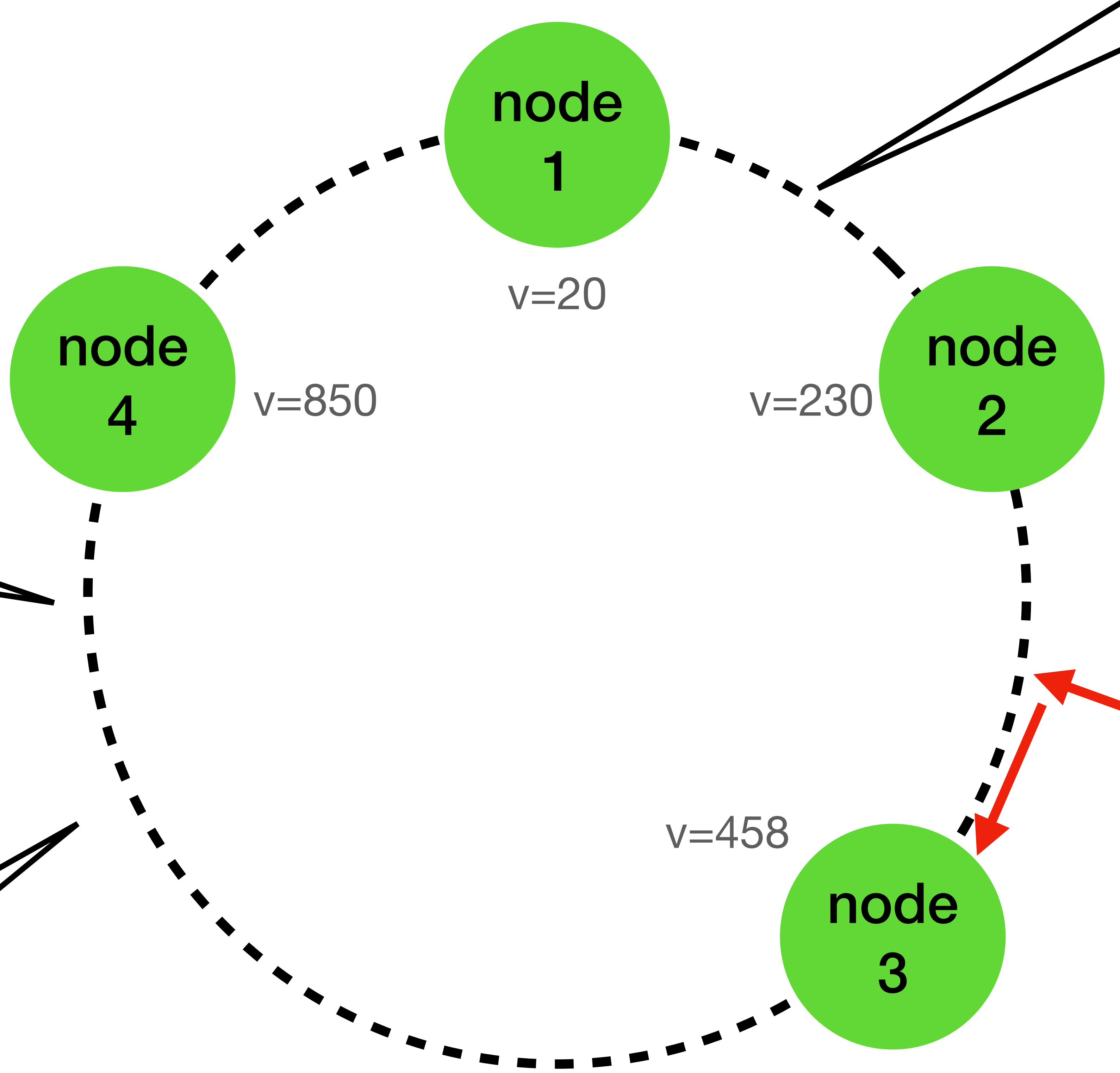
assigned to node3

Partitioning algorithm (3)

Consistent hashing

Range is [0:1000] for the example

Each node is responsible to the range between it and its predecessor



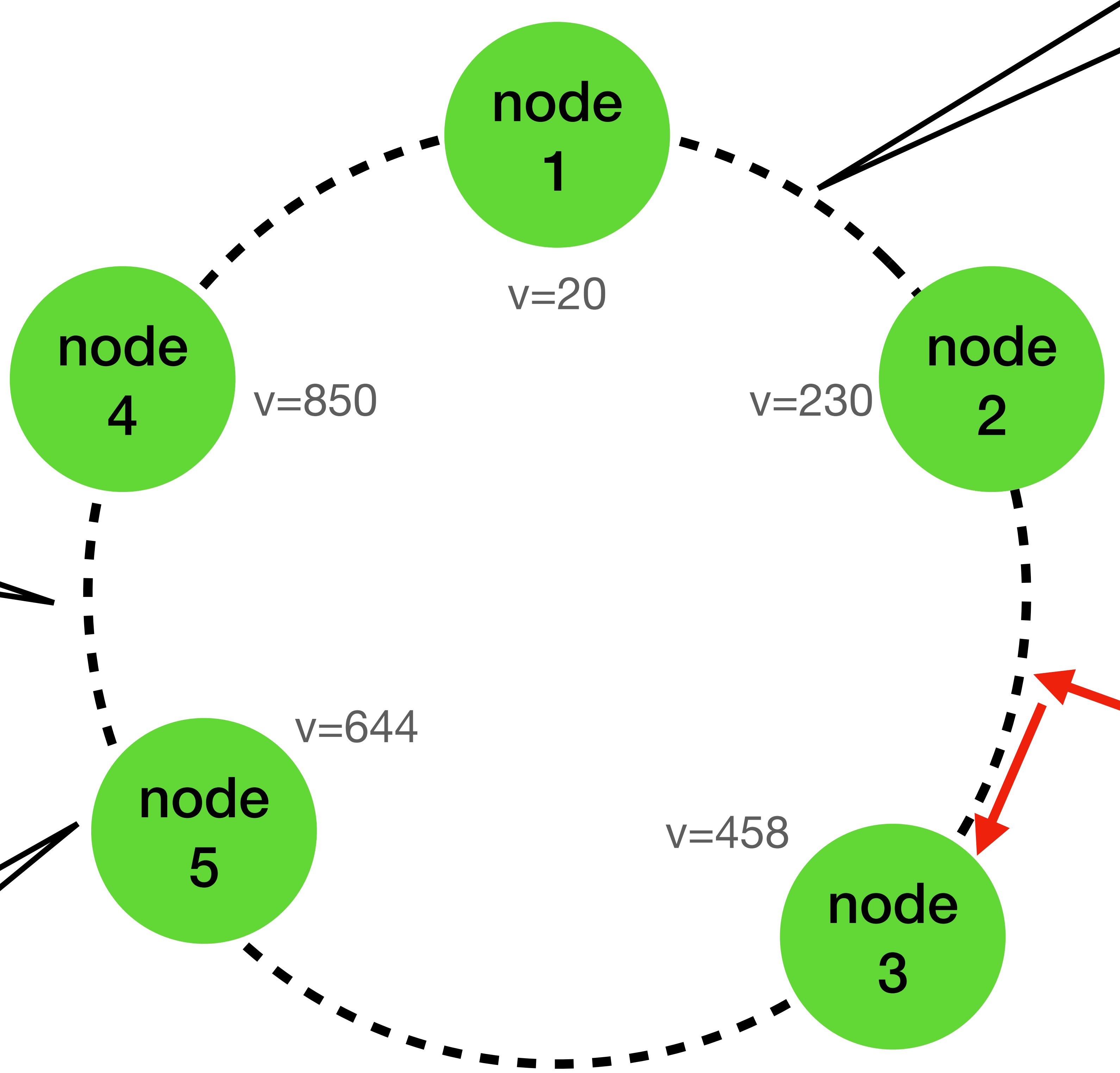
```
hash(key) = 344
```

assigned to node3

adding / removing a node only affect its neighbors

Partitioning algorithm (3)

Consistent hashing



Range is [0:1000] for the example

Each node is responsible to the range between it and its predecessor

```
hash(key) = 344
```

assigned to node3

adding / removing a node only affect its neighbors

Partitioning algorithm (4)

Consistent hashing - challenges

- Random positioning —> non uniform data distribution
- Node heterogeneity is not supported
  node hardware is not considered

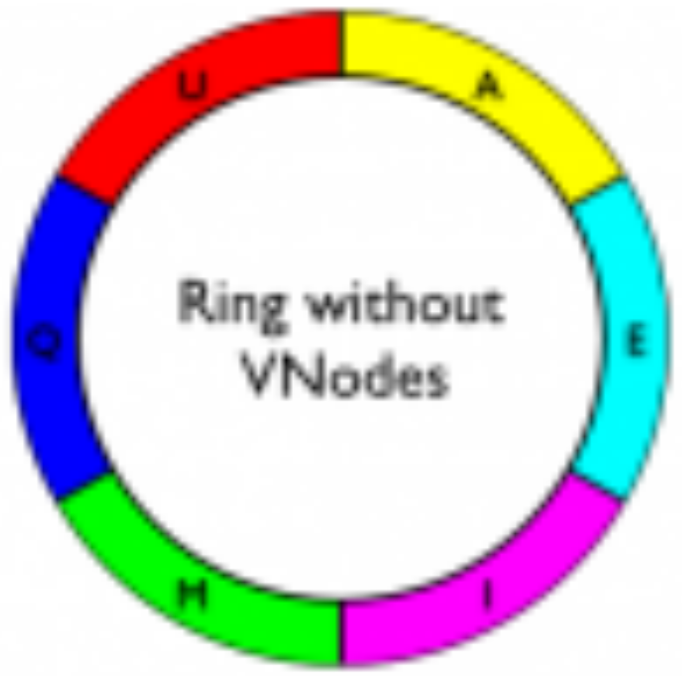
Partitioning algorithm (5)

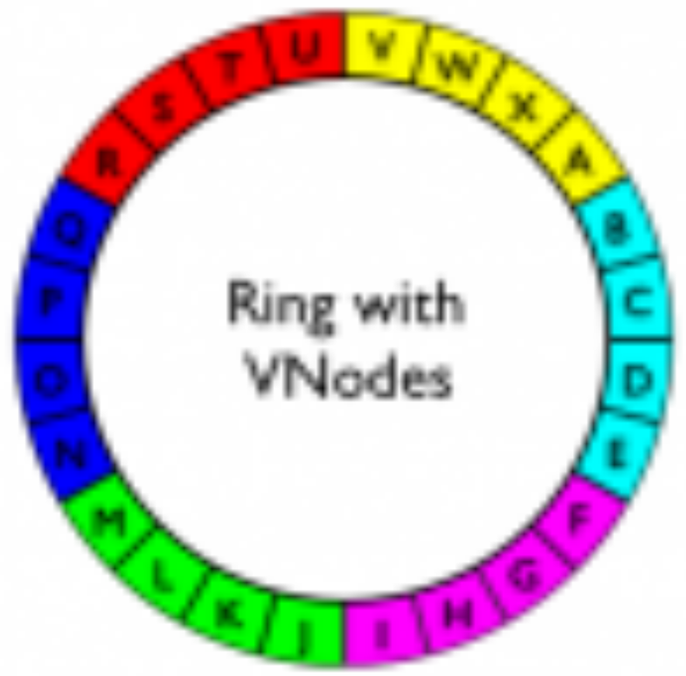
Dynamo consistent hashing

- Instead of a single “token” for a node, ,map vnodes

  vnode looks like a “normal” node
  each node manage several vnodes

- Basically the idea is to split the range into smaller pieces

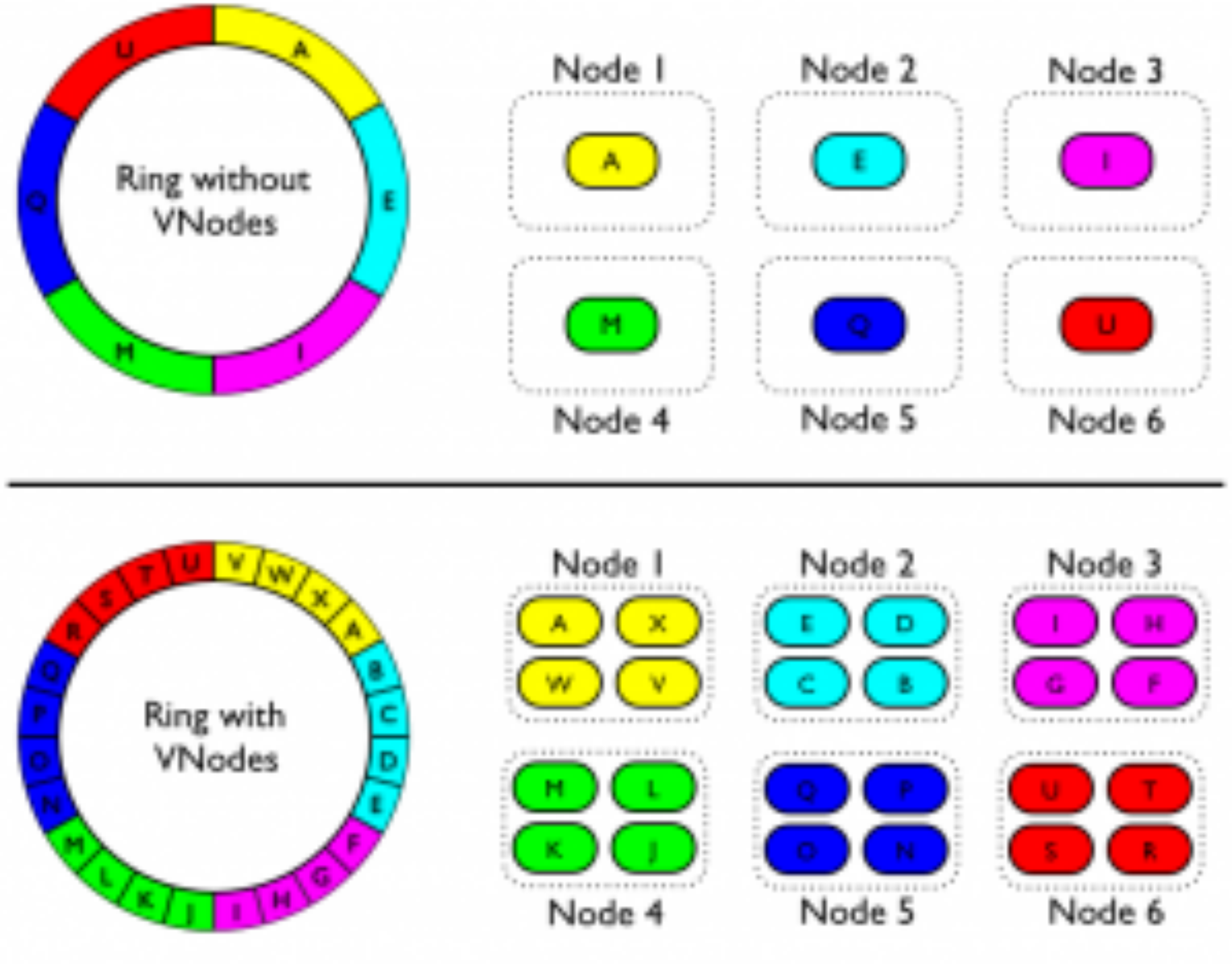




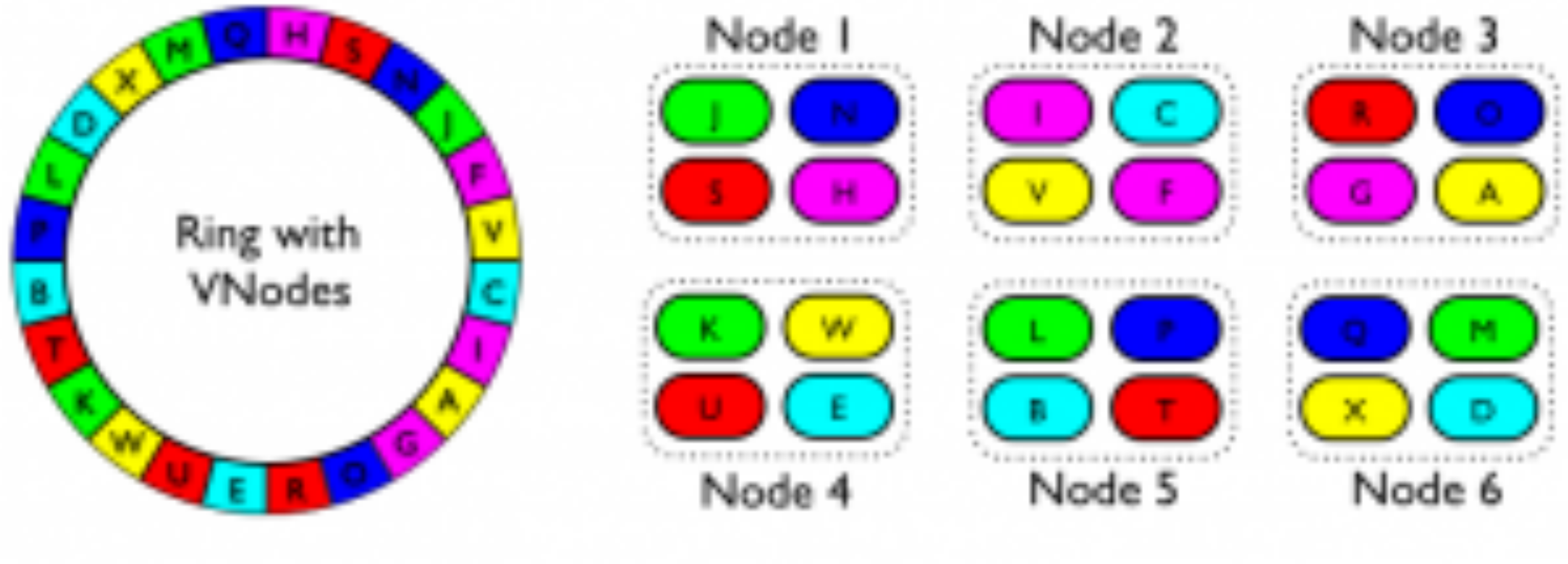
* Images from DataStax website

Partitioning algorithm (6)

Dynamo consistent hashing



VNodes before shuffling

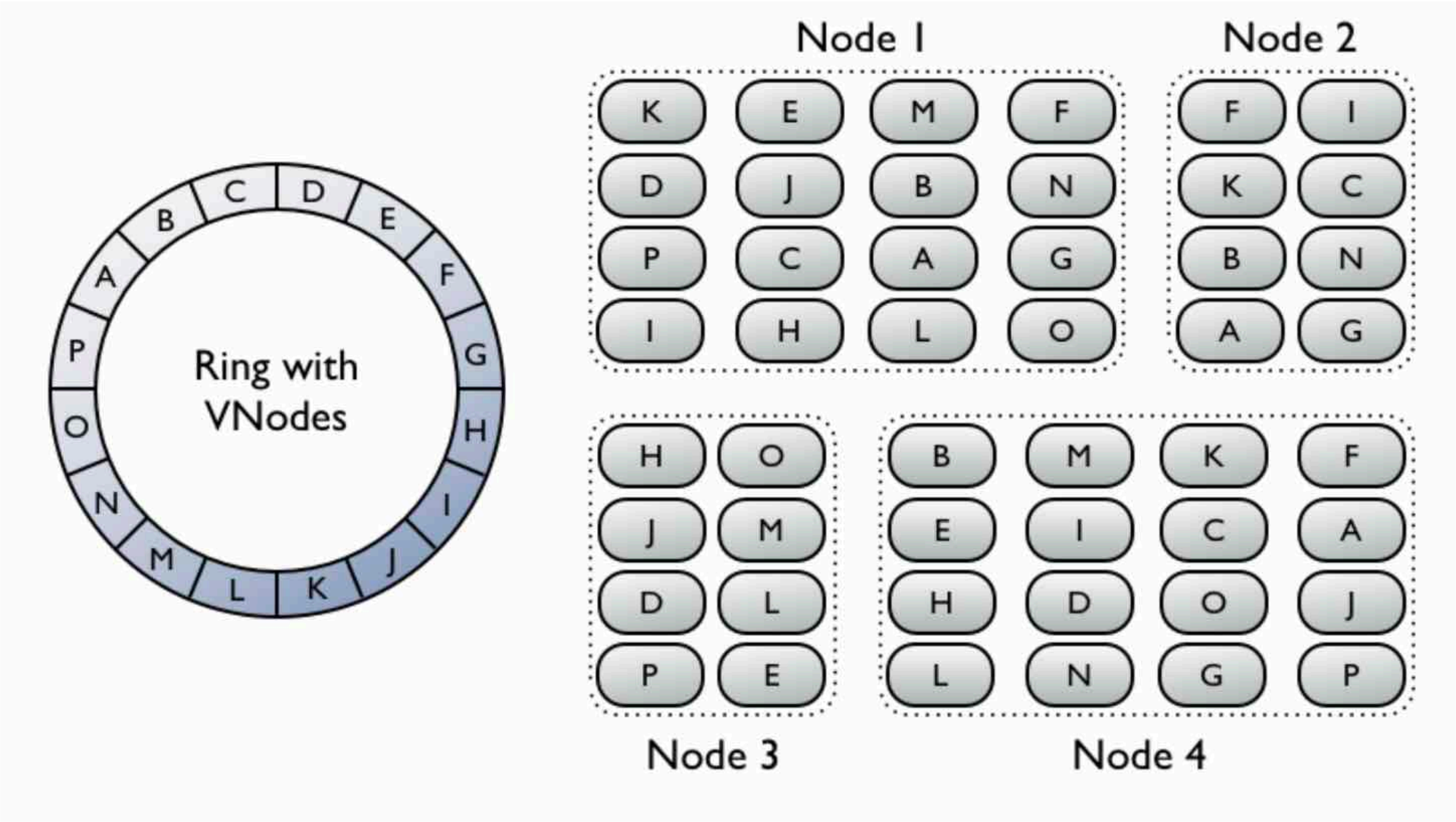


VNodes after shuffling

* Images from DataStax website

Partitioning algorithm (7)

Dynamo consistent hashing - node heterogeneity example



* Images from DataStax website

Partitioning algorithm (8)

Dynamo consistent hashing

- With vnodes:

—> data is distributed more evenly

—> #vnodes for each node is proportional to its hardware

—> If we add/remove a node, the load is now distributed among much mode nodes

Partitioning algorithm (9)

Dynamo consistent hashing - final note

- There are several options for assigning the range / node
  - Random
  - Equal size partitions, random tokens per nodes
  - Equal size partitions, equal tokens per nodes

- Not the focus for this presentation
  see the paper for more info actual load distribution

Dynamo topics

- Requirements
- Partition algorithm
- **Replication**
- Data versioning
- `get()` and `put()` execution
- Failures
- Ring membership

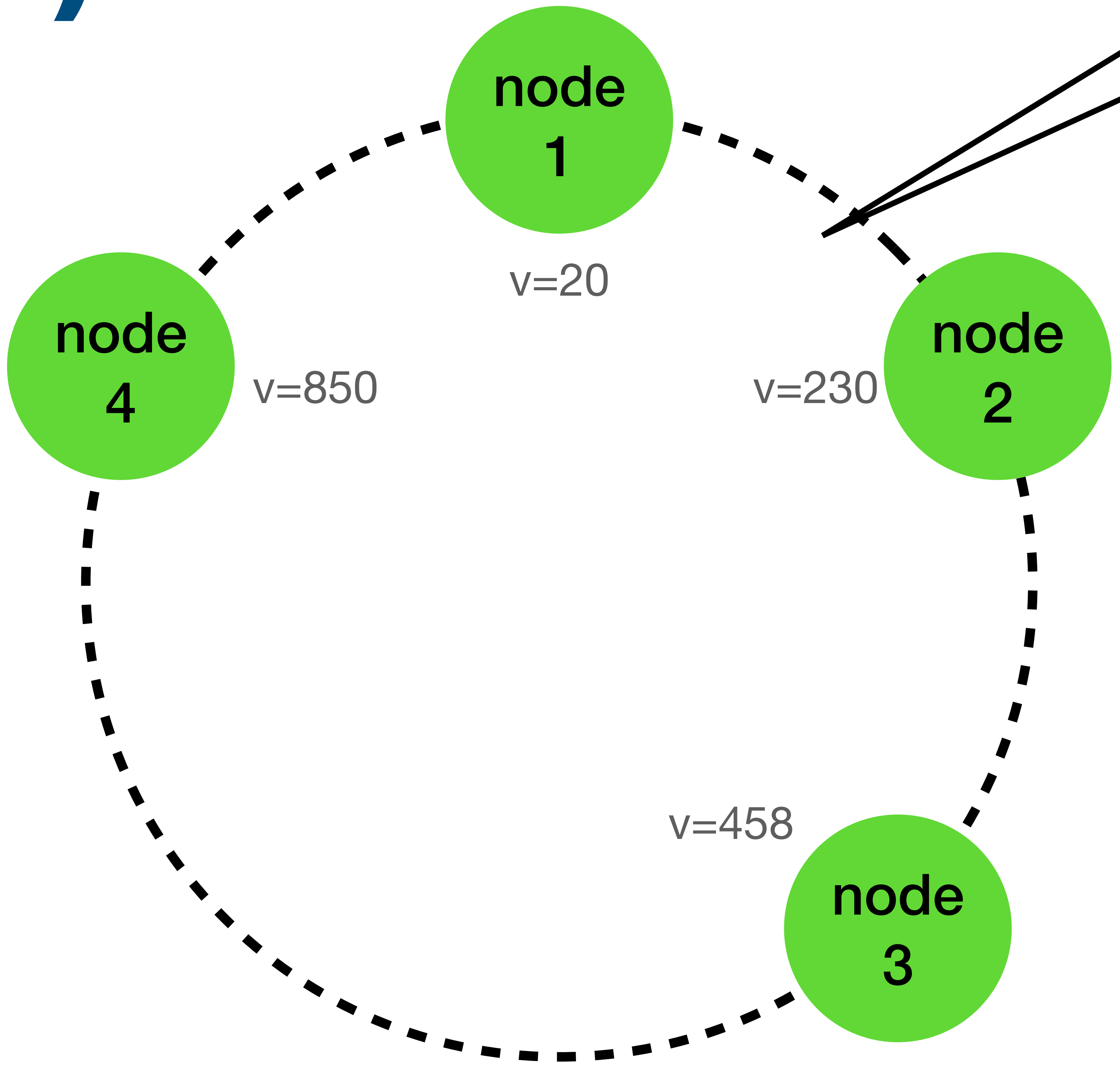
Replication (1)

- To achieve High availability and Durability, Dynamo replicates its data on N nodes (configurable)
- A key is assigned to a coordinator

  coordinator = the mapped node from the consistent hashing
- The coordinator stores locally + on the next N-1 nodes

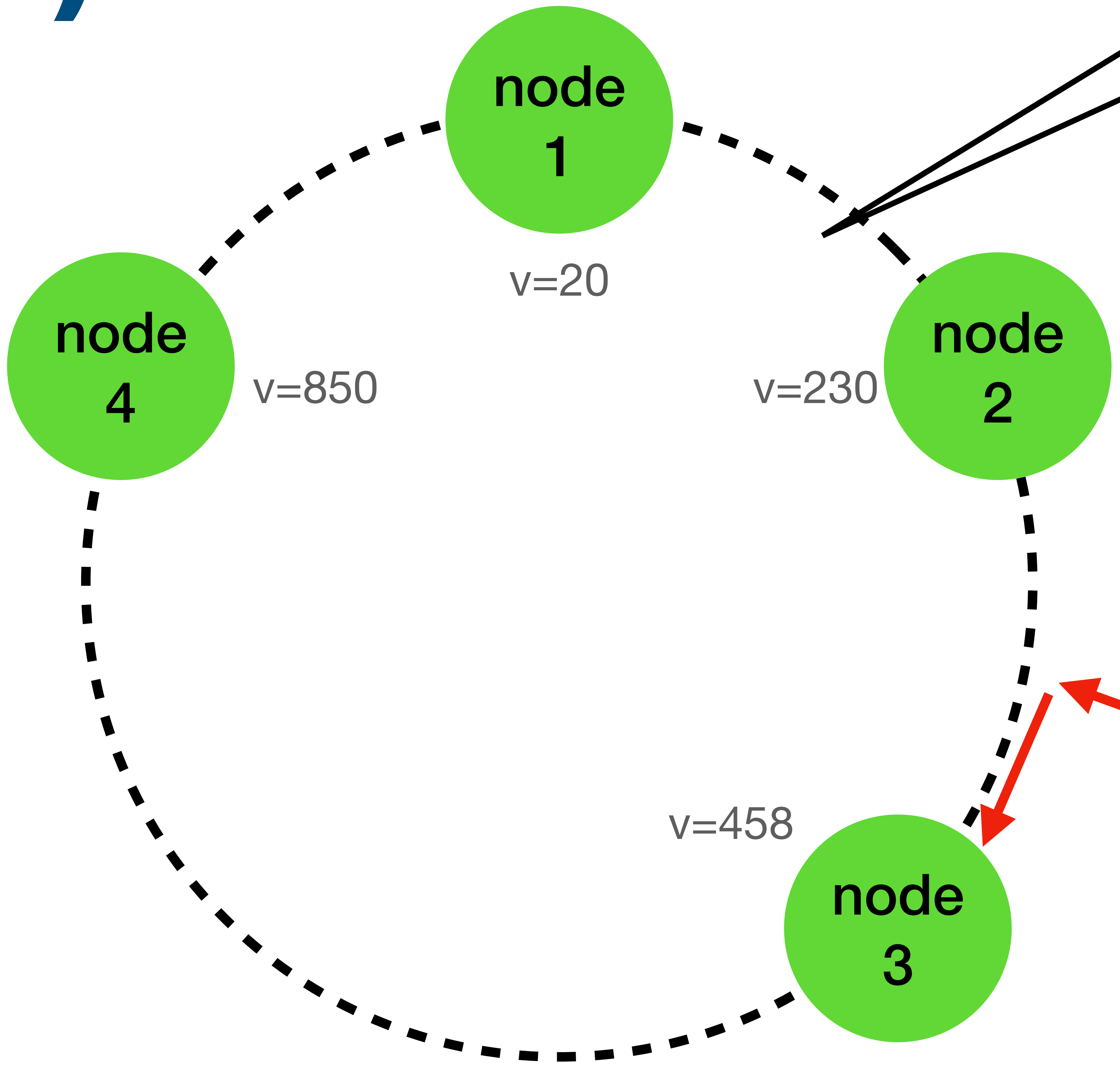
  automatically skips vnodes of “existing” nodes as we want to store the data on N physically different nodes

Replication (2)



Range is [0:1000] for the example,
N=3

Replication (2)

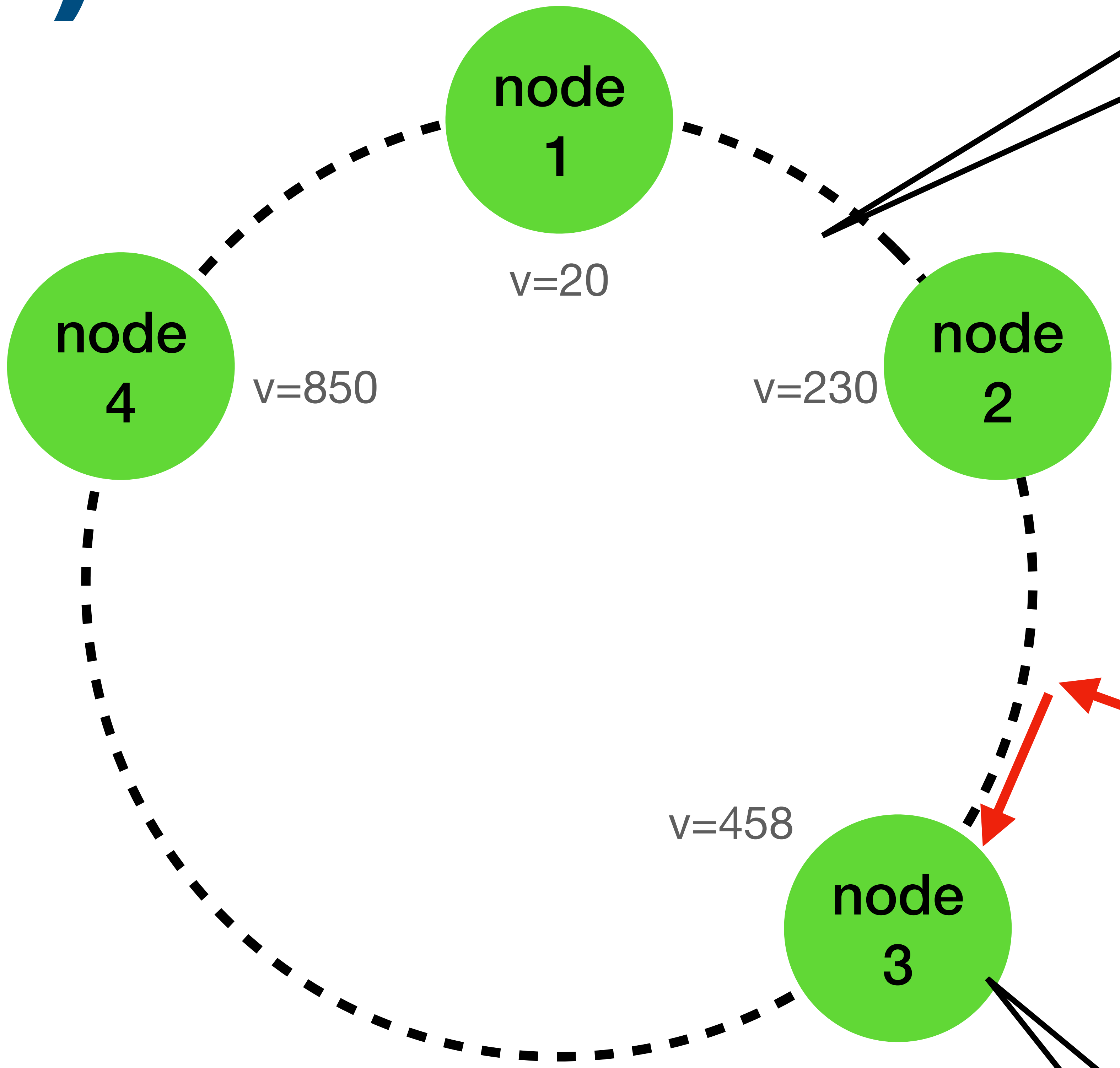


Range is [0:1000] for the example,
N=3

`hash(key) = 344`

(assigned to node3)

Replication (2)



Range is [0:1000] for the example,
N=3

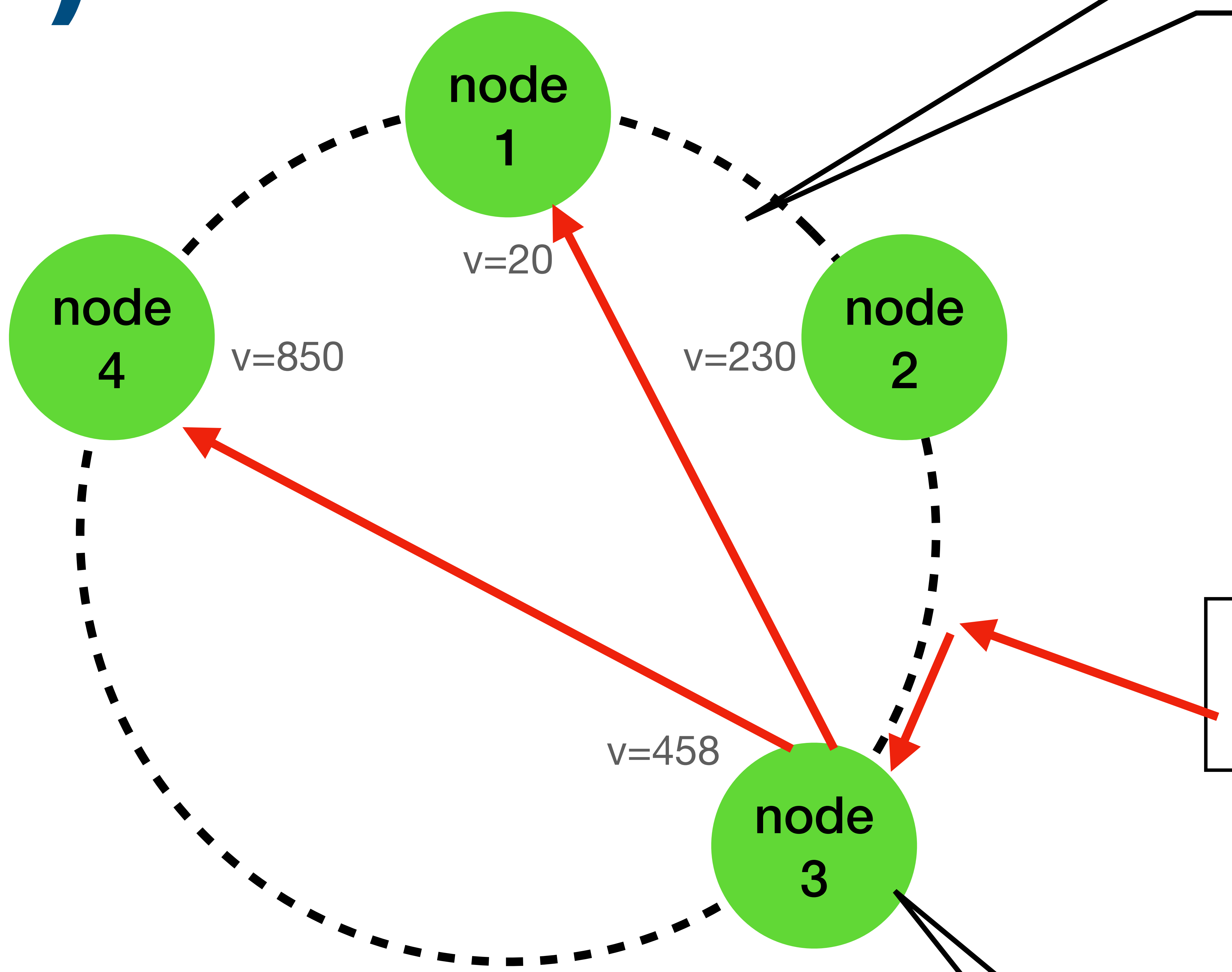
```
hash(key) = 344
```

(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

Replication (2)

Range is [0:1000] for the example,
N=3



```
hash(key) = 344
```

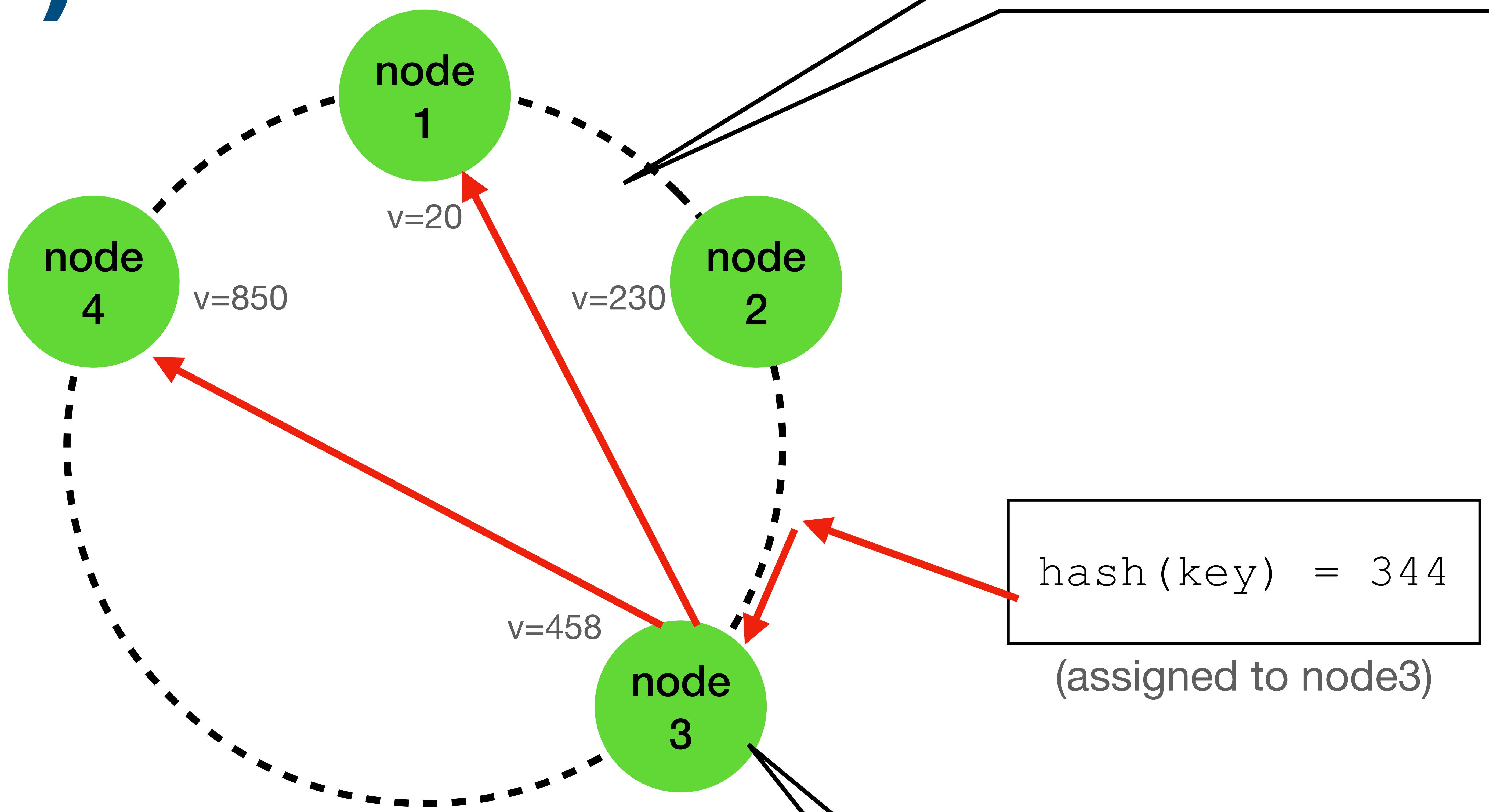
(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

Replication (2)

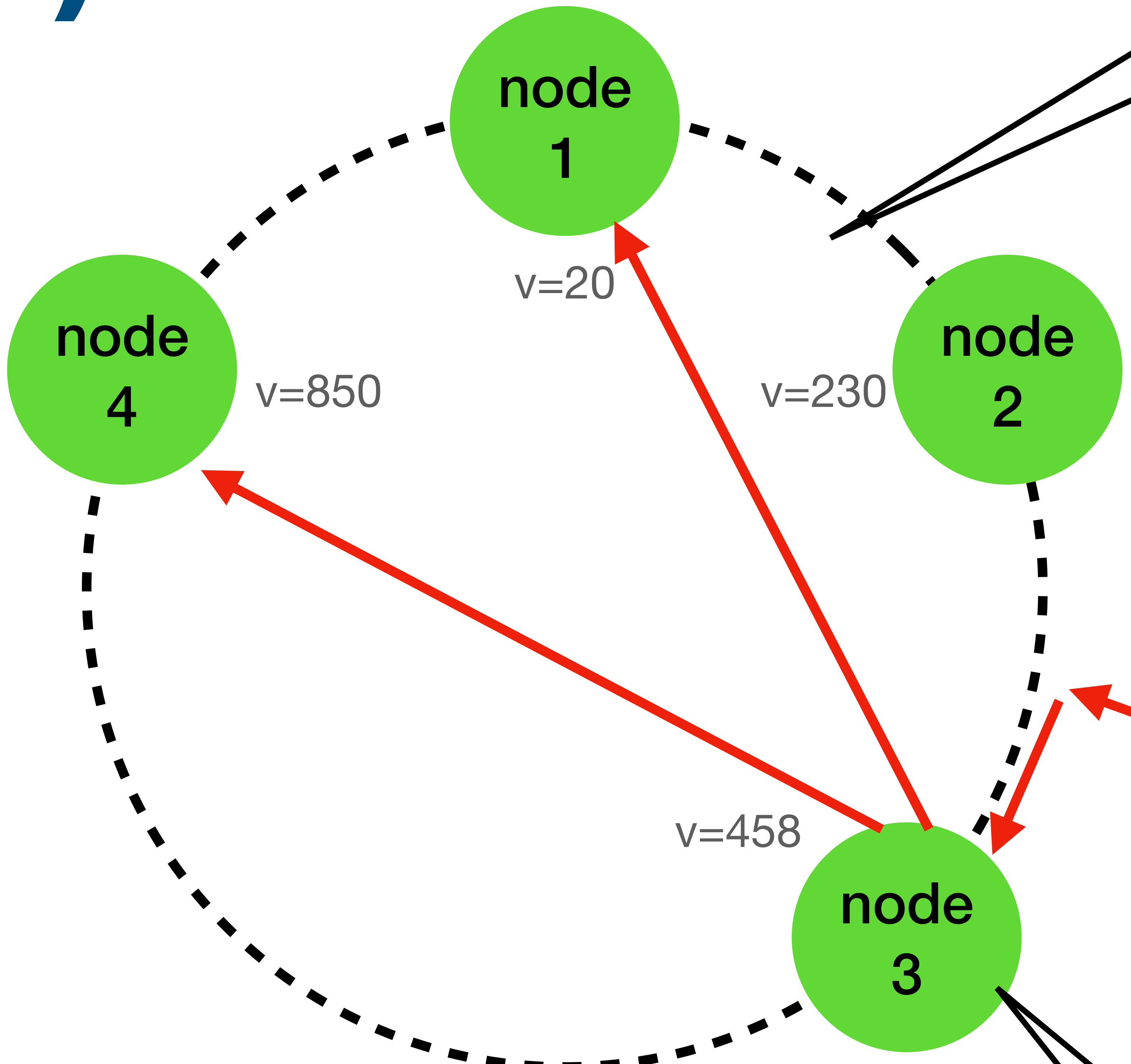
Range is [0:1000] for the example, N=3

NOTE - All values in the range between [node2:node3] will be stored on node3, node4 and node1



node3 is the coordinator. It will store the value locally and on node4 and node1

Replication (2)



Range is [0:1000] for the example, N=3

NOTE - All values in the range between [node2:node3] will be stored on node3, node4 and node1

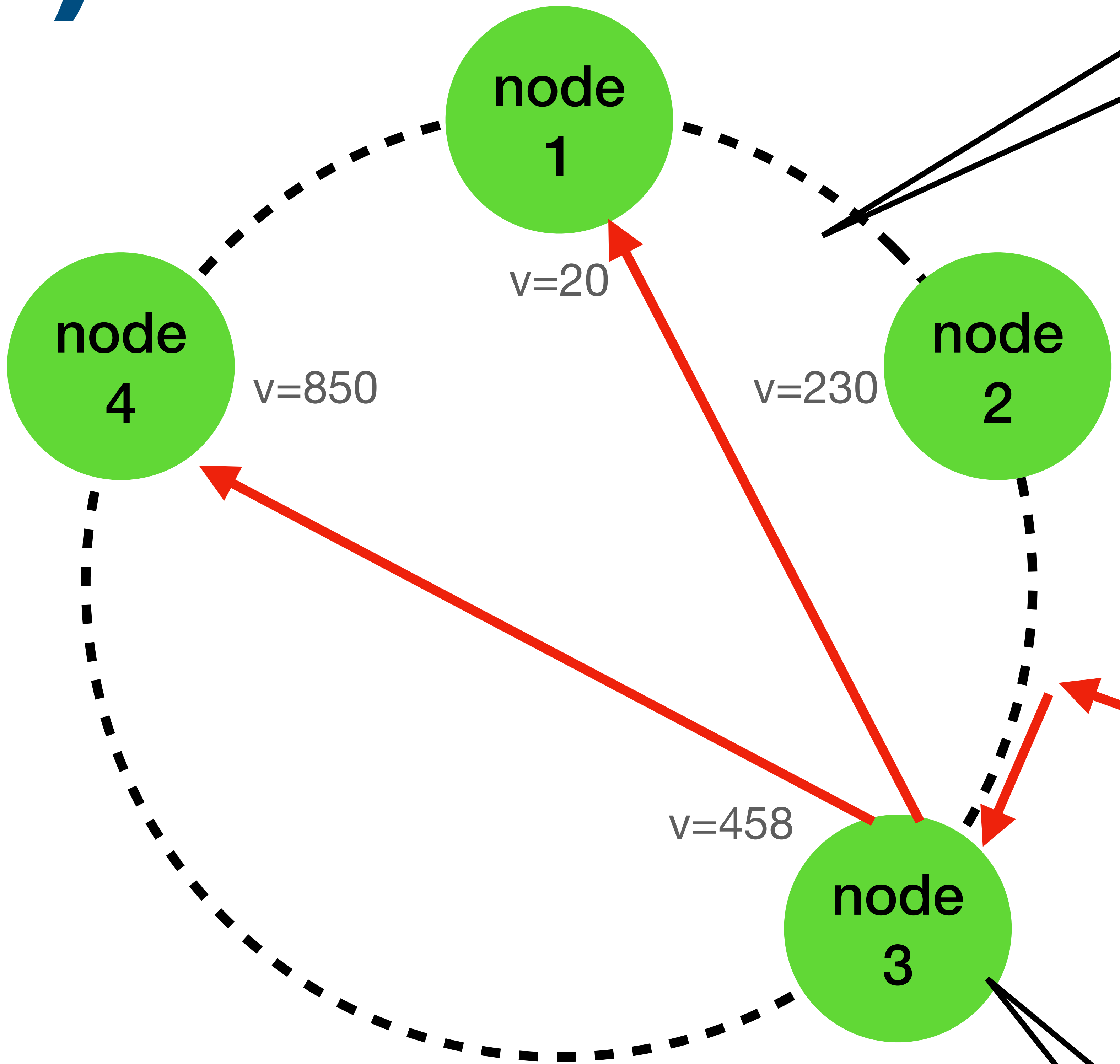
```
hash(key) = 344
```

(assigned to node3)

As all nodes “know” the “ring”, for each key any node “knows” on which nodes that data is stored (“**preference list**”)

node3 is the coordinator. It will store the value locally and on node4 and node1

Replication (2)



Range is [0:1000] for the example, N=3

NOTE - All values in the range between [node2:node3] will be stored on node3, node4 and node1

```
hash(key) = 344
```

(assigned to node3)

As all nodes “know” the “ring”, for each key any node “knows” on which nodes that data is stored (“**preference list**”)

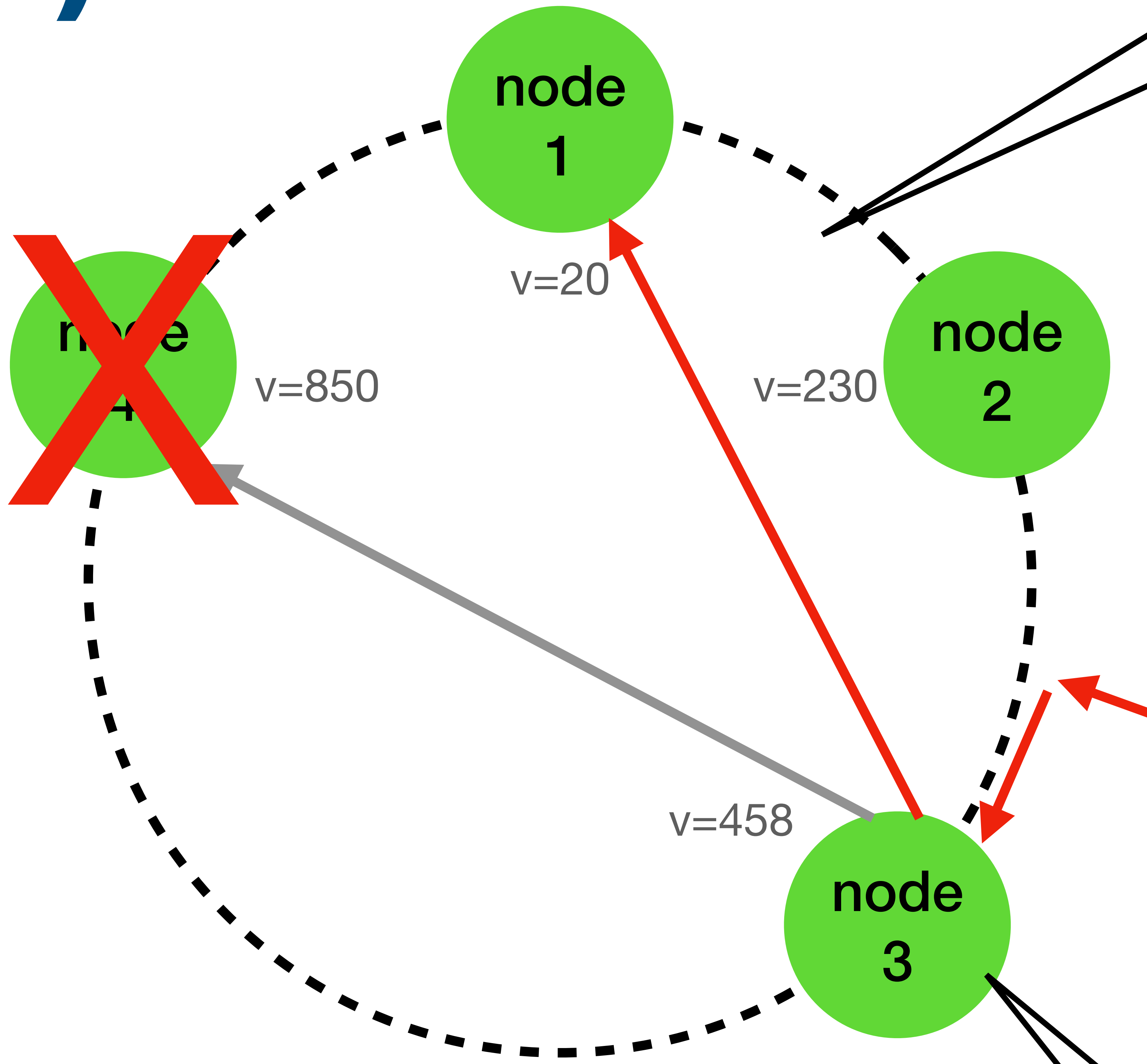
node3 is the coordinator. It will store the value locally and on node4 and node1

A “preference list” can contain more than N nodes in order to handle “fail nodes”. For example, if node4 fails, that value will be stored on node3, node1 and node2

Replication (2)

Range is [0:1000] for the example, N=3

NOTE - All values in the range between [node2:node3] will be stored on node3, node4 and node1



```
hash(key) = 344
```

(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

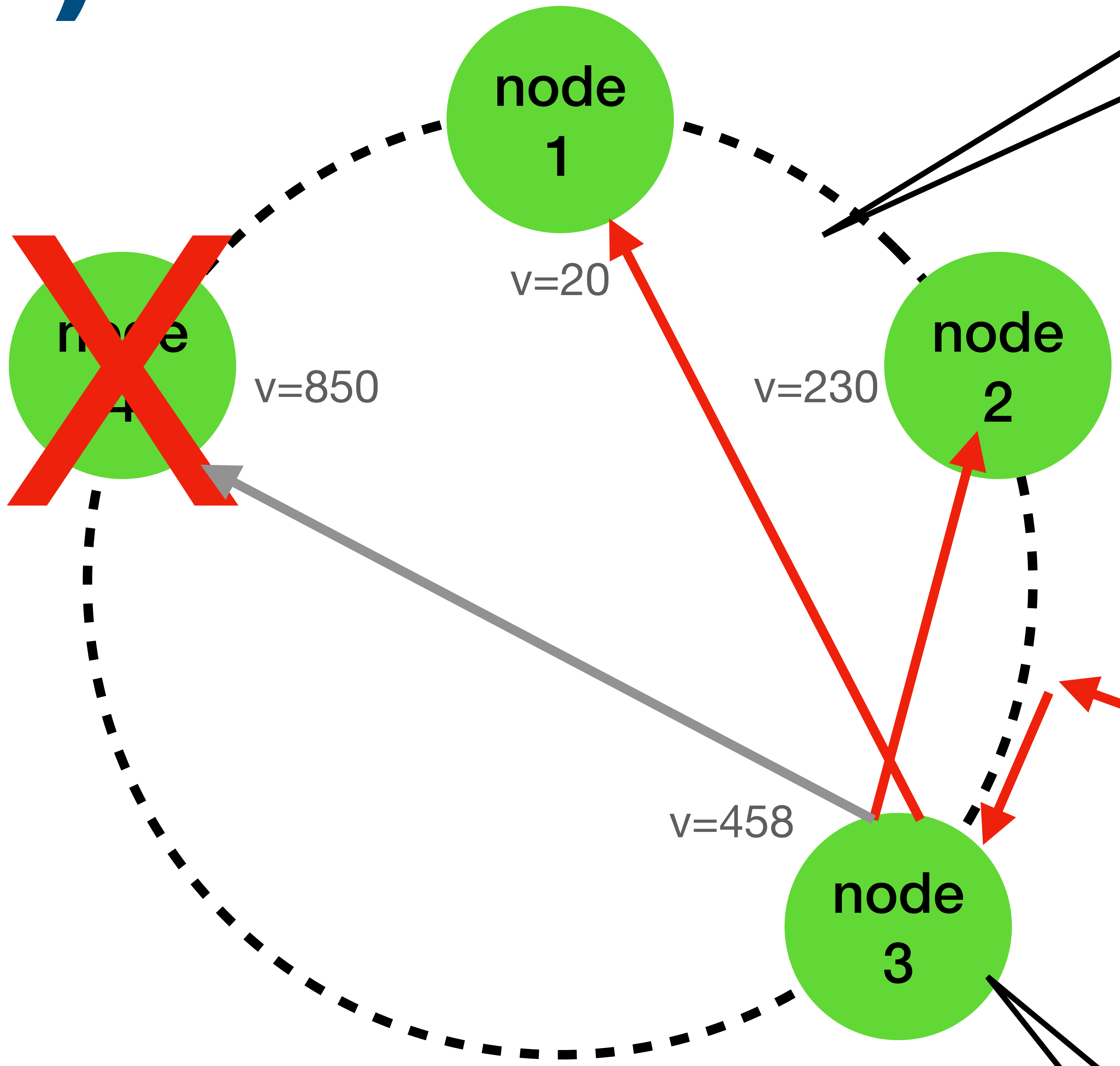
As all nodes “know” the “ring”, for each key any node “knows” on which nodes that data is stored (“**preference list**”)

A “preference list” can contain more than N nodes in order to handle “fail nodes”. For example, if node4 fails, that value will be stored on node3, node1 and node2

Replication (2)

Range is [0:1000] for the example, N=3

NOTE - All values in the range between [node2:node3] will be stored on node3, node4 and node1



```
hash(key) = 344
```

(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

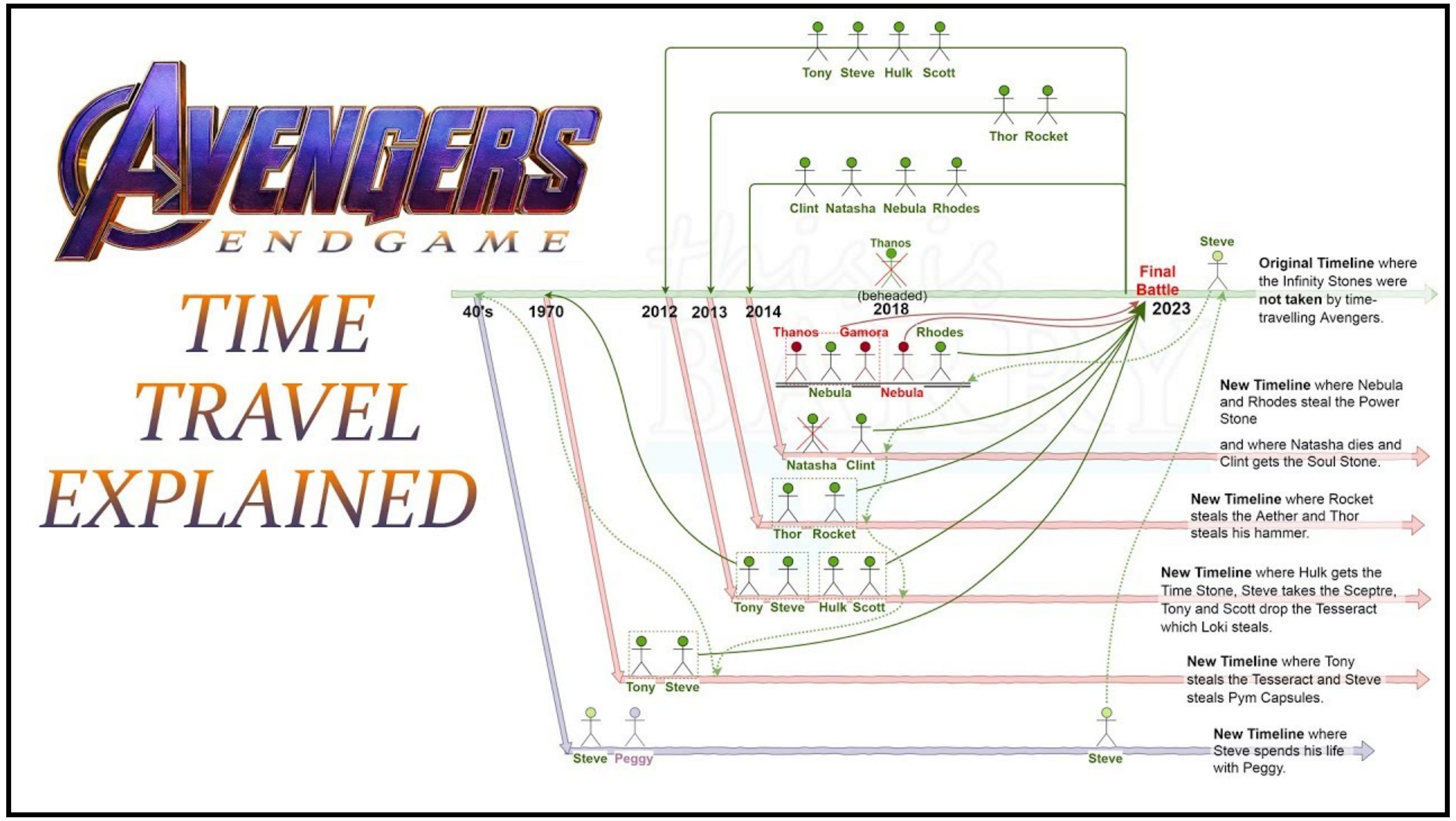
As all nodes “know” the “ring”, for each key any node “knows” on which nodes that data is stored (“preference list”)

A “preference list” can contain more than N nodes in order to handle “fail nodes”. For example, if node4 fails, that value will be stored on node3, node1 and node2

Dynamo topics

- Requirements
- Partition algorithm
- Replication
- **Data versioning**
- `get()` and `put()` execution
- Failures
- Ring membership

Data versioning


AVENGERS
ENDGAME
TIME
TRAVEL
EXPLAINED
Tony Steve Hulk Scott
Thor Rocket
Clint Natasha Nebula Rhodes
Thanos
(beheaded)
Steve
Final Battle
40's
1970
2012
2013
2014
2018
2023
Original Timeline where the Infinity Stones were not taken by time-travelling Avengers.
Thanos Gamora Rhodes
Nebula Nebula
New Timeline where Nebula and Rhodes steal the Power Stone
and where Natasha dies and Clint gets the Soul Stone.
Natasha Clint
New Timeline where Rocket steals the Aether and Thor steals his hammer.
Thor Rocket
New Timeline where Hulk gets the Time Stone, Steve takes the Sceptre, Tony and Scott drop the Tesseract which Loki steals.
Tony Steve Hulk Scott
New Timeline where Tony steals the Tesseract and Steve steals Pym Capsules.
Tony Steve
New Timeline where Steve spends his life with Peggy.
Steve Peggy
Steve

https://www.youtube.com/watch?v=kn2IoDzI8L0

Reminder: Requirements: Interface

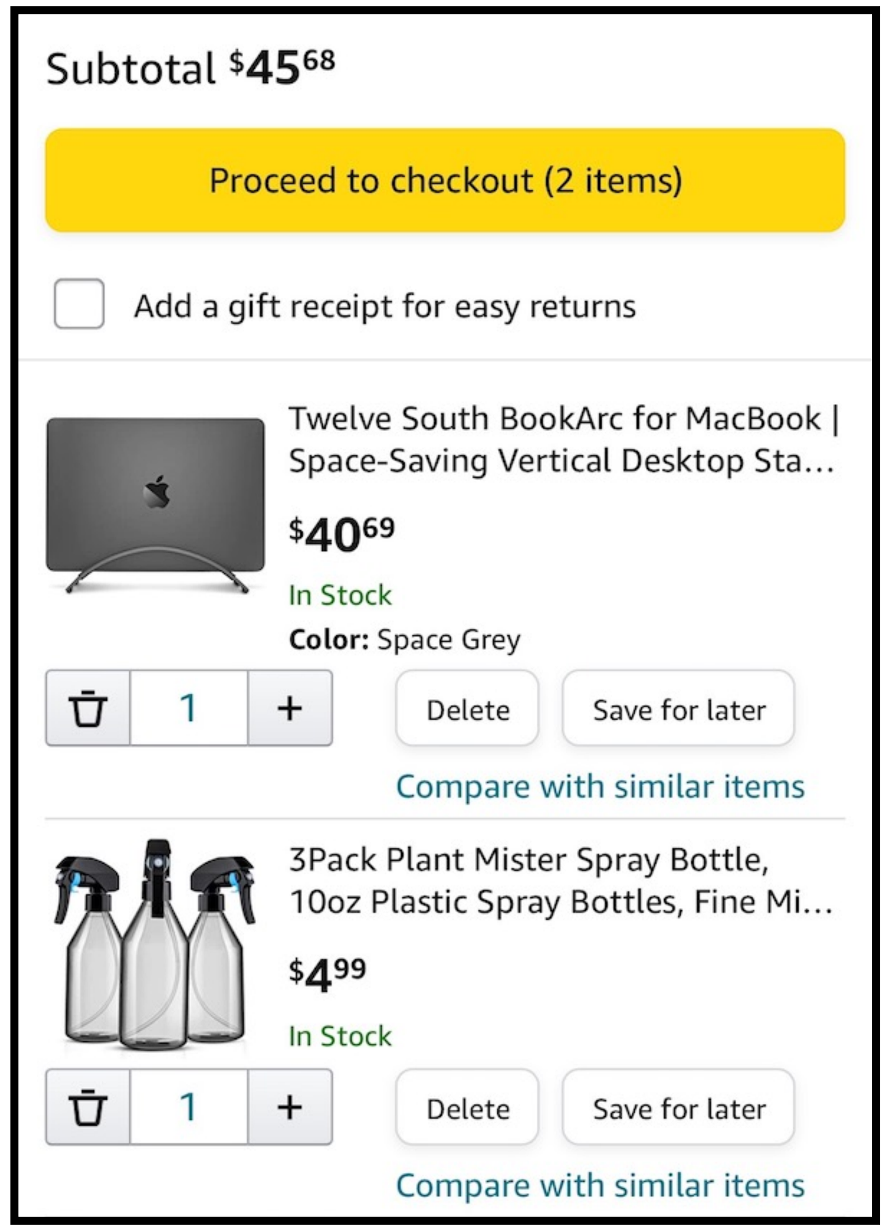
- `put(key, context, object)`
- `get(key)`
  - `context` = system metadata / versioning (opaque to the user)
  - `get` returns <u>all versions</u> of the associated object
    * we will later see when can we have multi versions

Data versioning (1)

- Dynamo provides "Eventual consistency"
- A `put()` may returned before updating all replicas
- A subsequent `get()` may return not latest value
- If no node fails, there is a bound on the propagation time
- **On node failures,** it may take a while, and the problems begins

Data versioning (2) - motivation

- Apps that can tolerate some “inconsistencies”
  for example, shopping cart
- **“Add to cart” should never fails**
- If previous value is unavailable, we should still be able to add a new item
  and **“merge”** the “old” cart once available
- Both add/delete from cart are translated to `put()`
  each update is a new **immutable** version of the data
- On conflicts, the client app “reconcile” by a merge
  this guarantees that an added item is never lost
  but deleted items can resurface



Subtotal $\$45^{68}$

Proceed to checkout (2 items)

□ Add a gift receipt for easy returns

Twelve South BookArc for MacBook | Space-Saving Vertical Desktop Sta...

$\$40^{69}$

In Stock

Color: Space Grey

1 + Delete Save for later

Compare with similar items

3Pack Plant Mister Spray Bottle, 10oz Plastic Spray Bottles, Fine Mi...

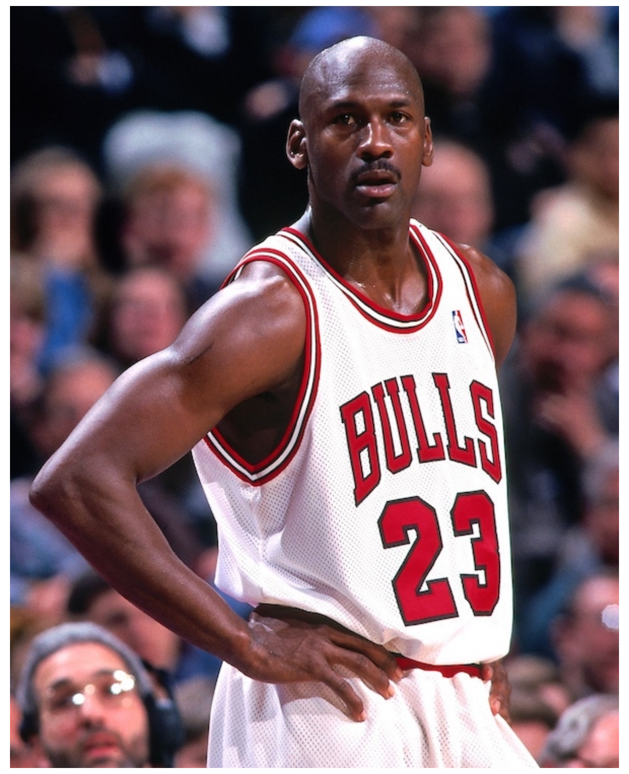
$\$4^{99}$

In Stock

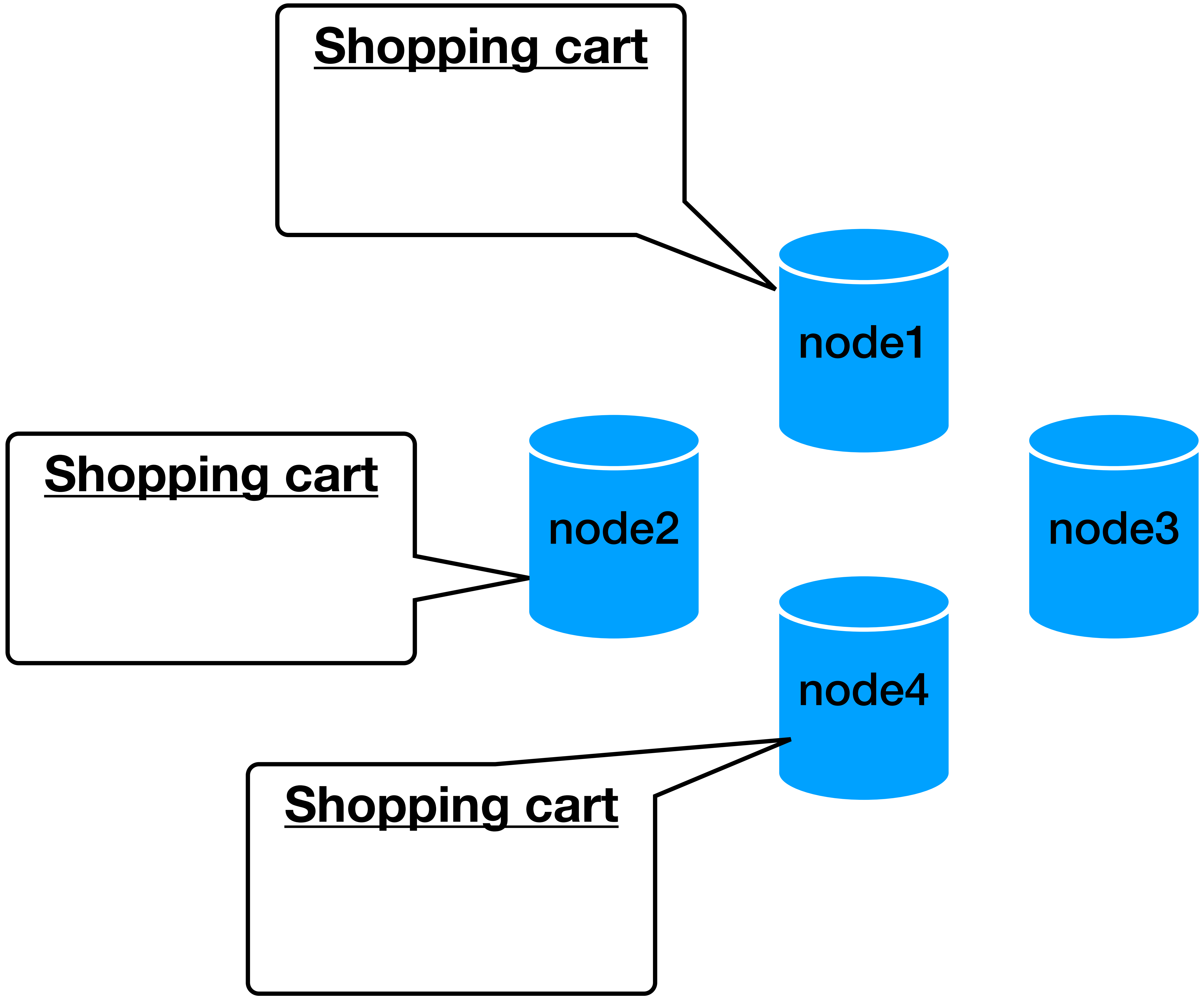
1 + Delete Save for later

Compare with similar items

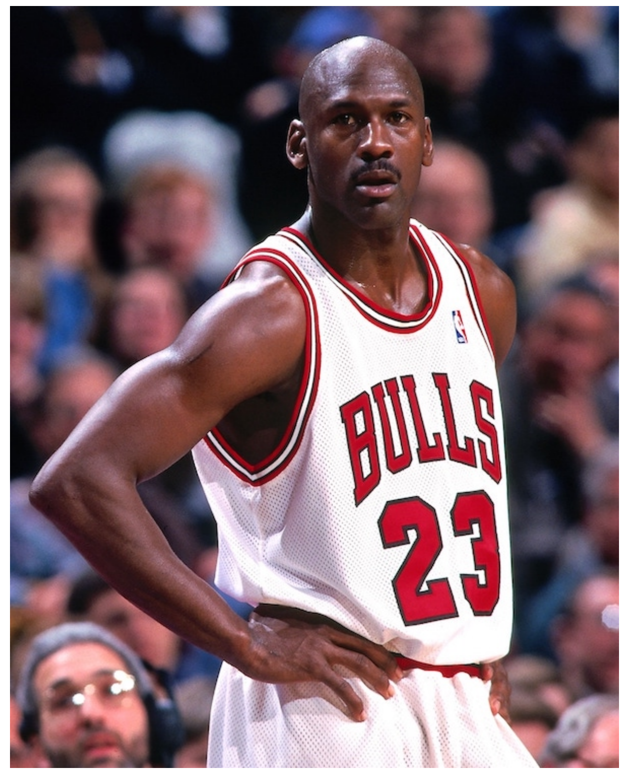
Data versioning (2) - motivation example



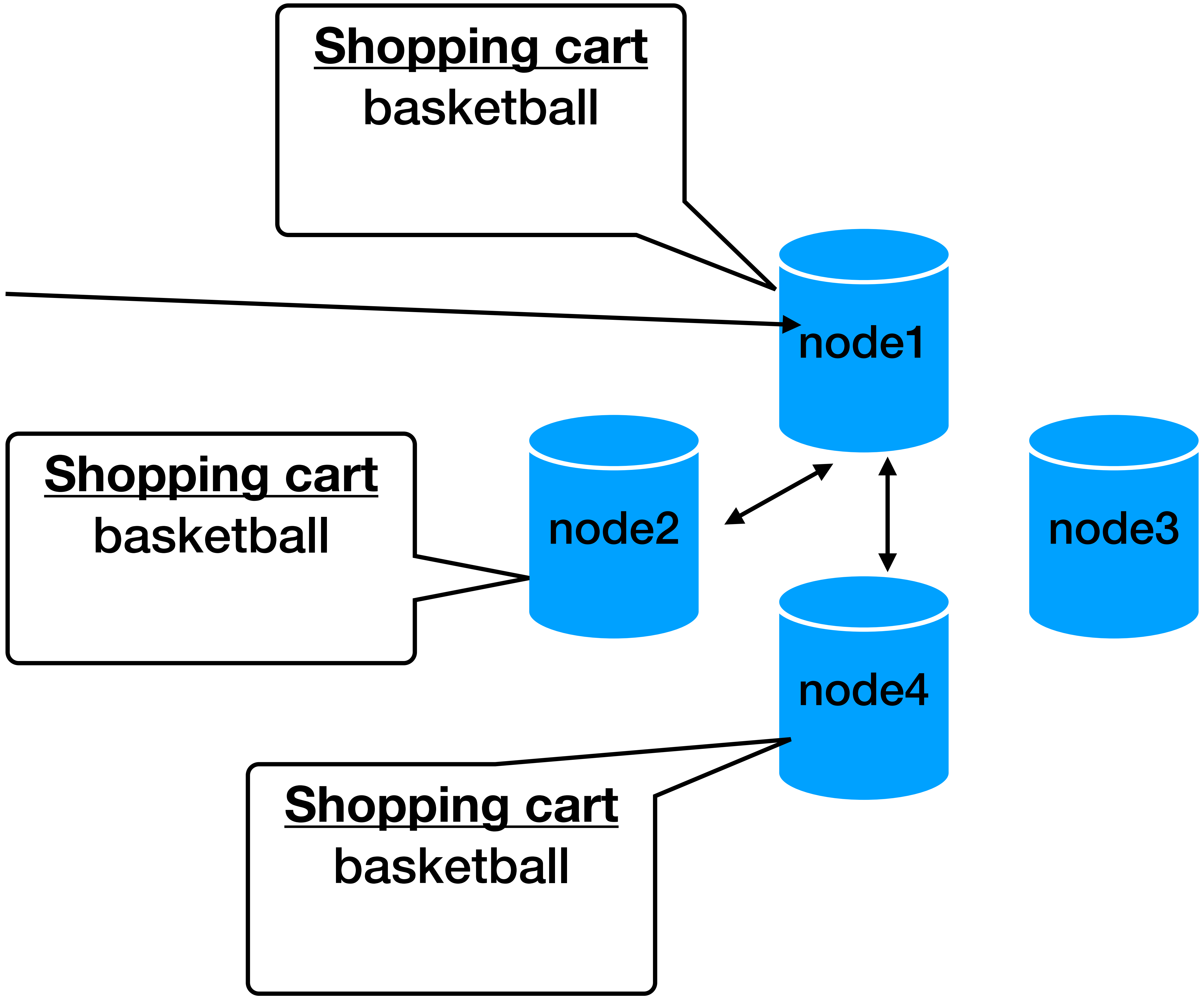
10:00: empty cart



Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball



Shopping cart
basketball

node1

Shopping cart
basketball

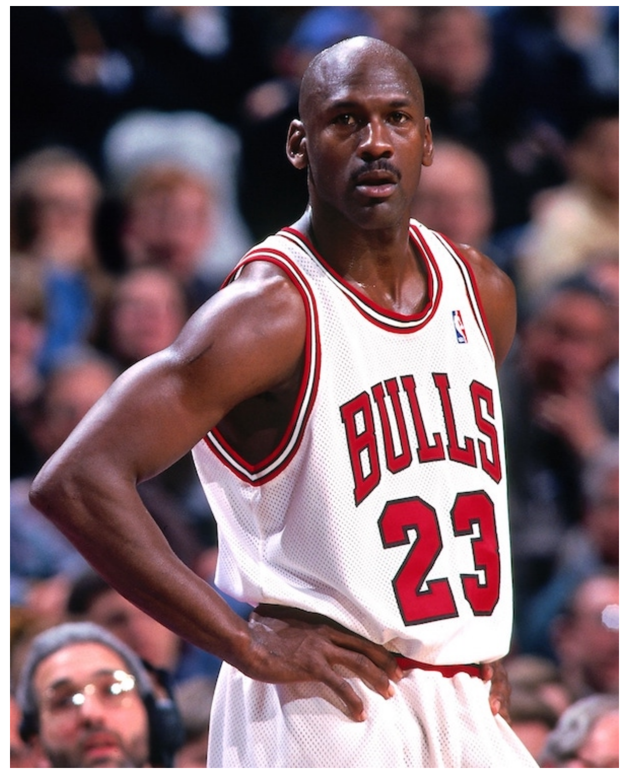
node2

node3

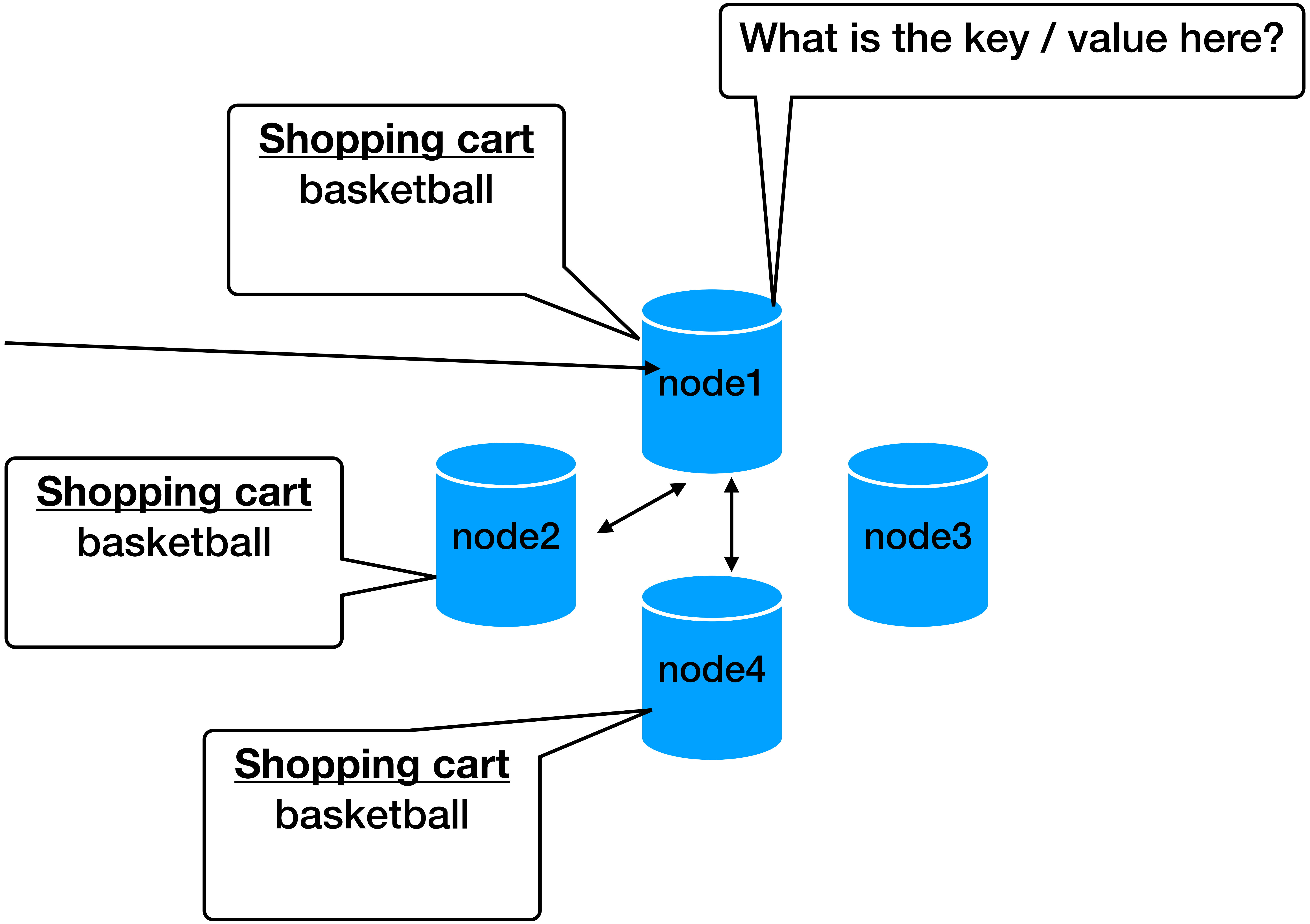
node4

Shopping cart
basketball

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball



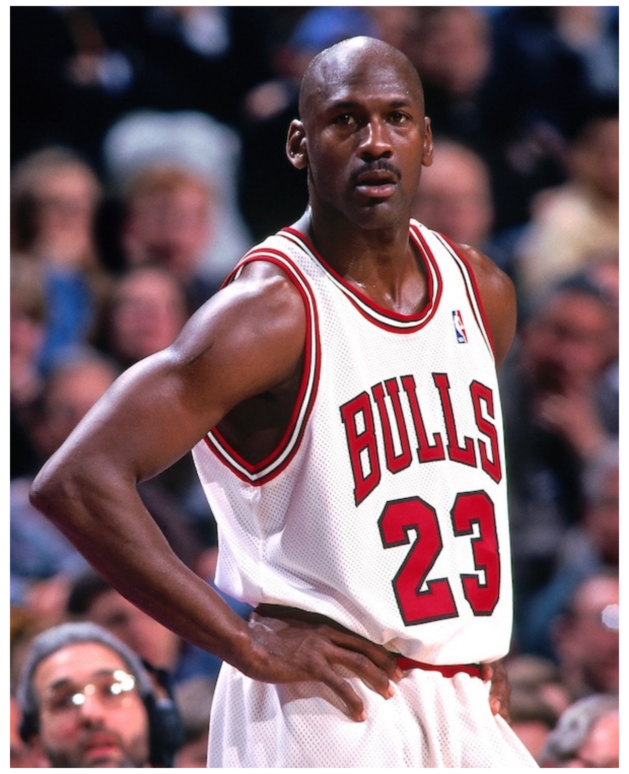
What is the key / value here?

Shopping cart
basketball

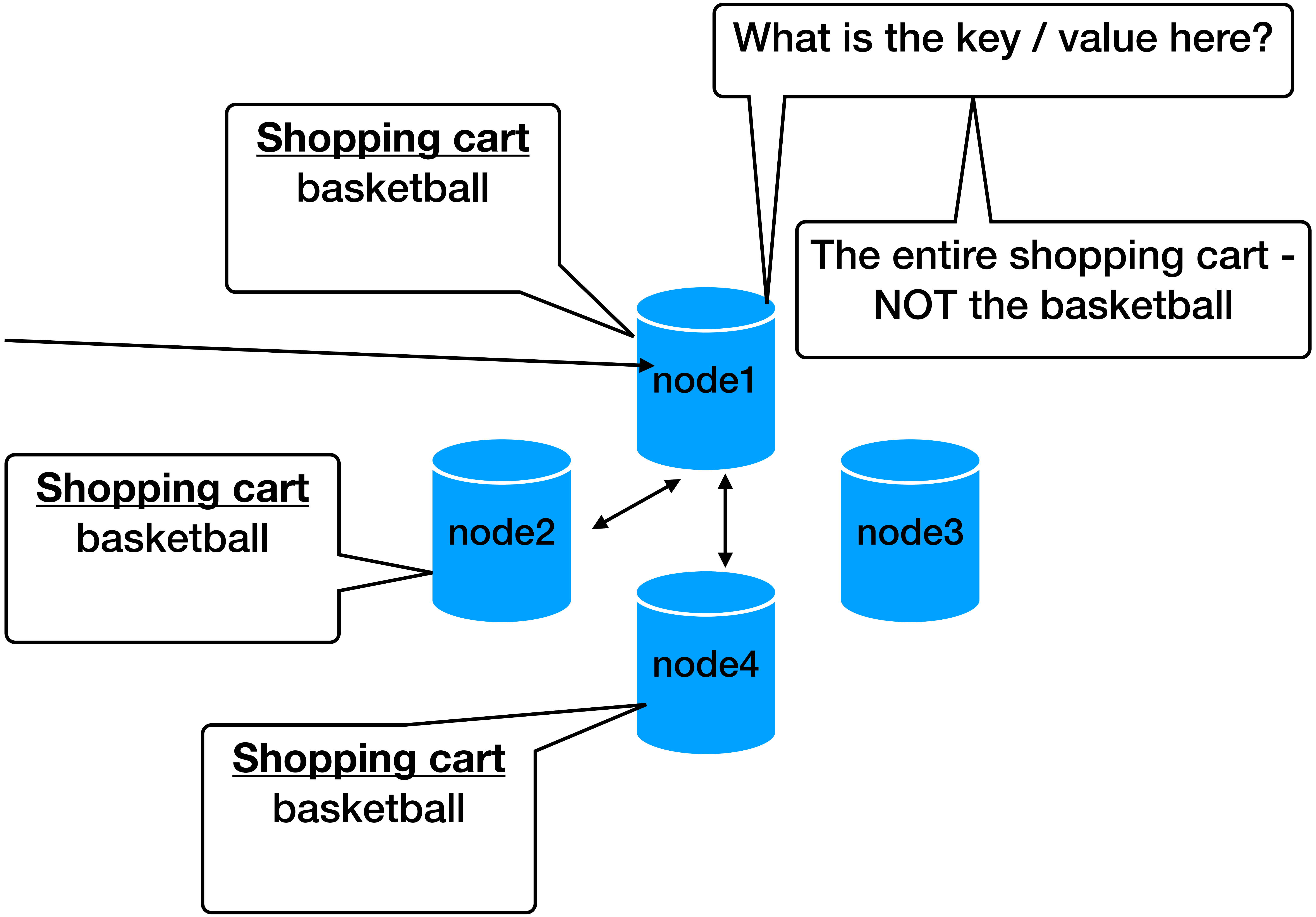
Shopping cart
basketball

Shopping cart
basketball

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball



What is the key / value here?

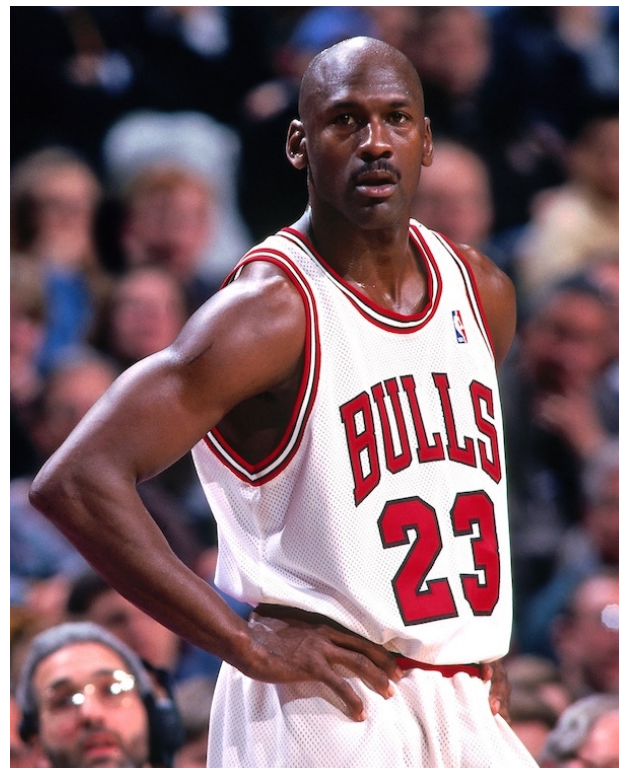
The entire shopping cart - NOT the basketball

Shopping cart
basketball

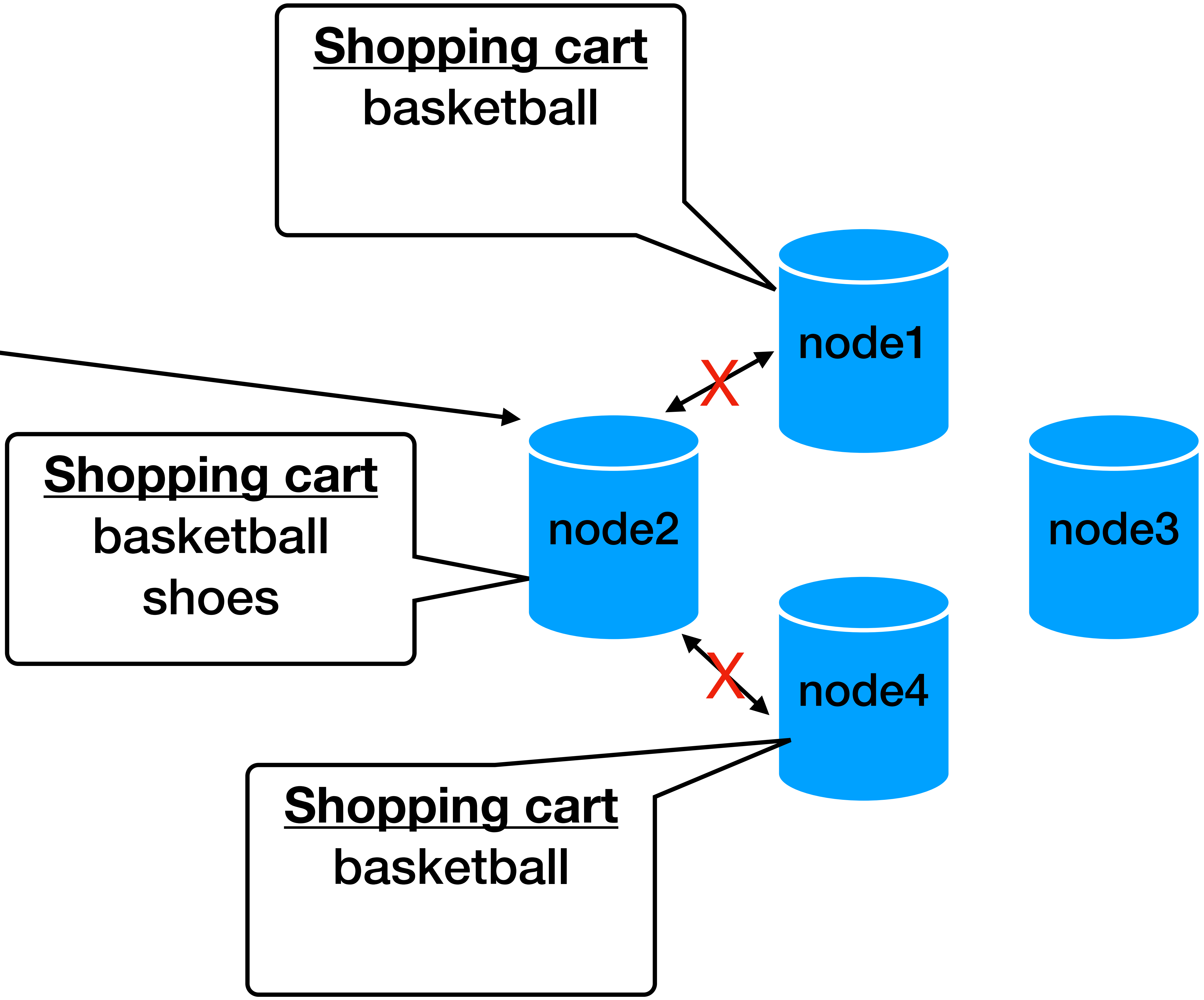
Shopping cart
basketball

Shopping cart
basketball

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes



Shopping cart
basketball

node1

Shopping cart
basketball
shoes

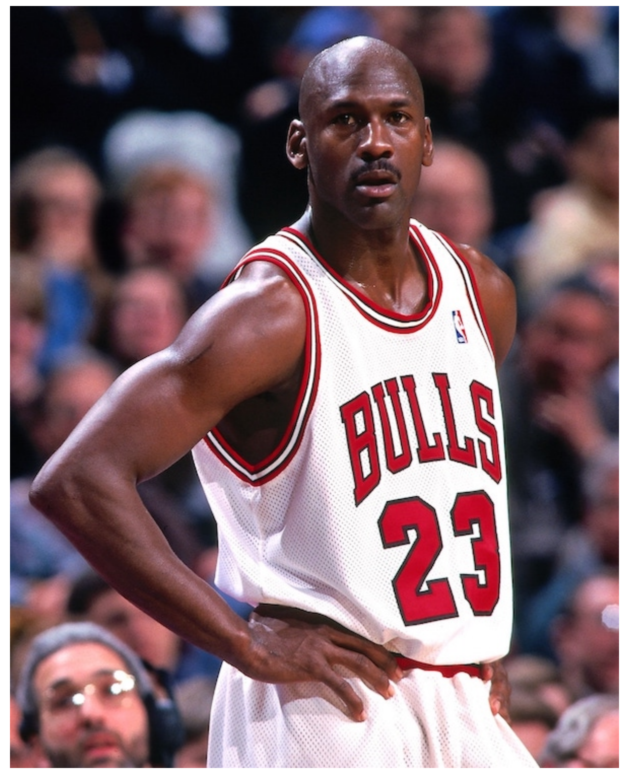
node2

node3

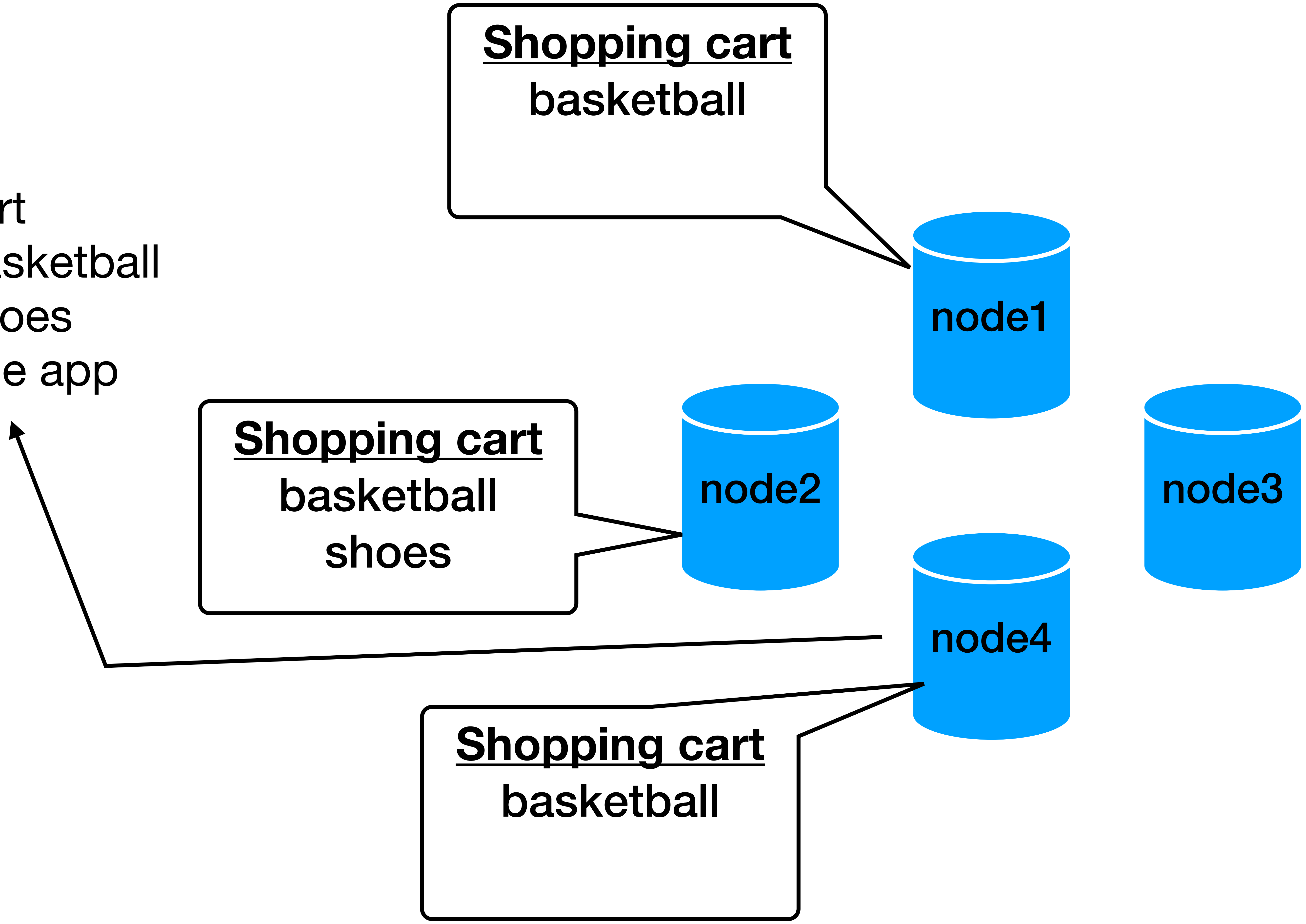
node4

Shopping cart
basketball

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app



Shopping cart
basketball

node1

Shopping cart
basketball
shoes

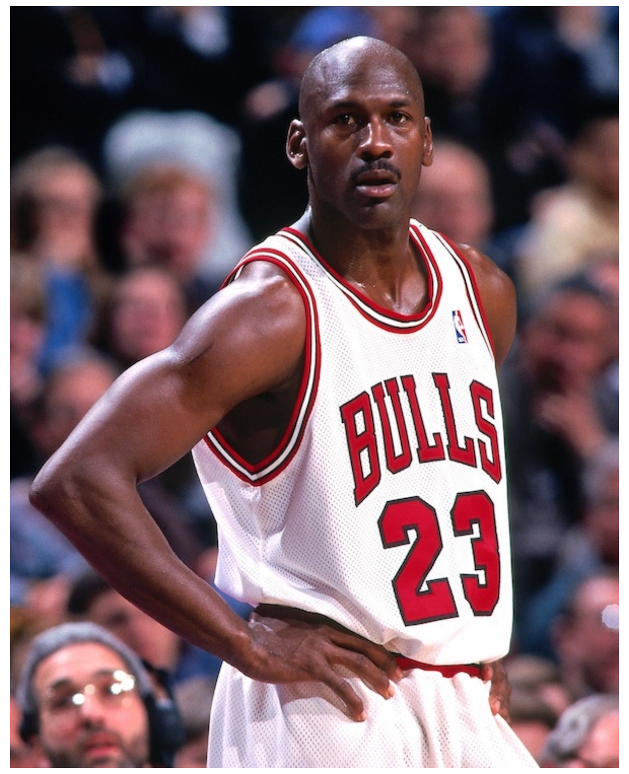
node2

node3

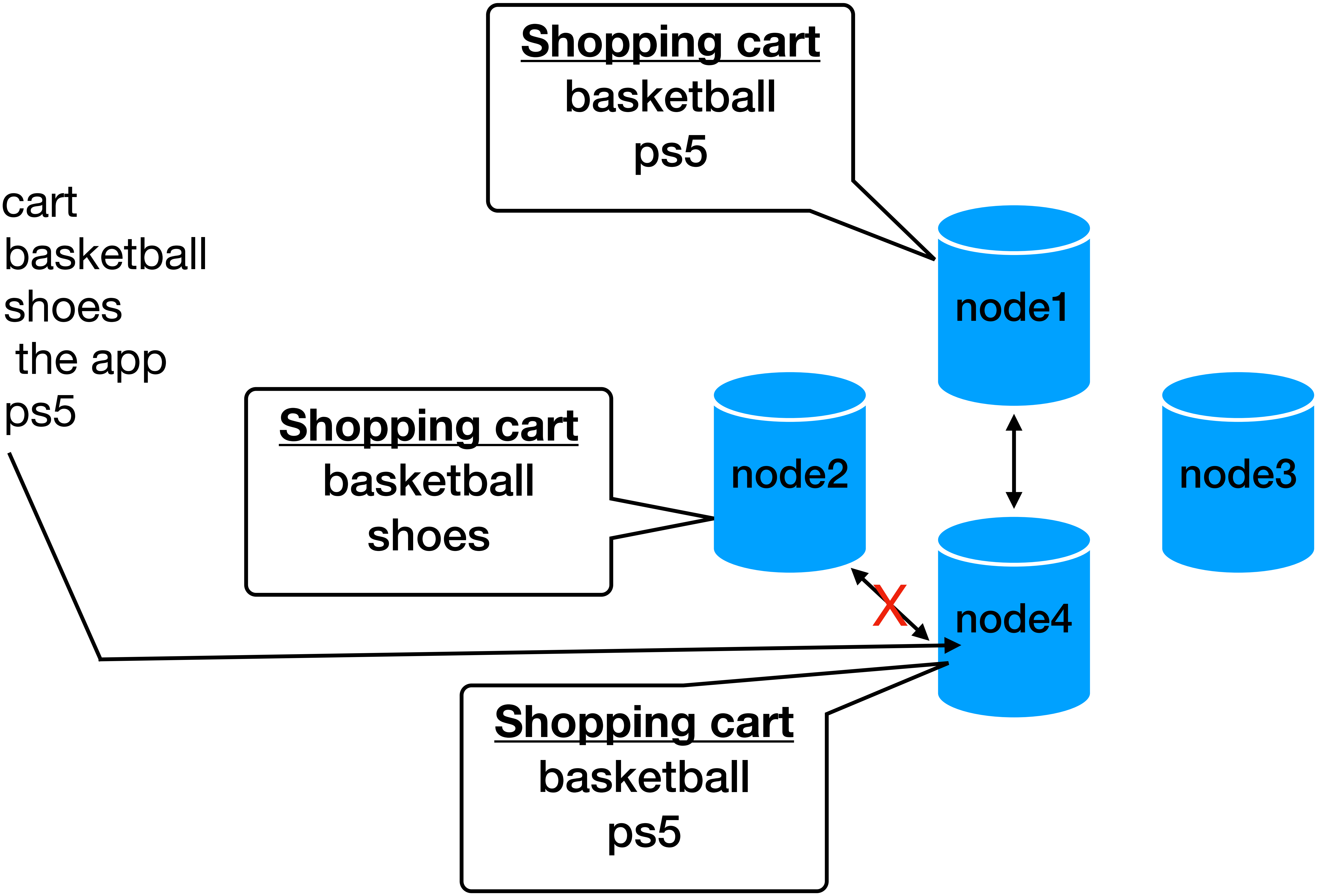
node4

Shopping cart
basketball

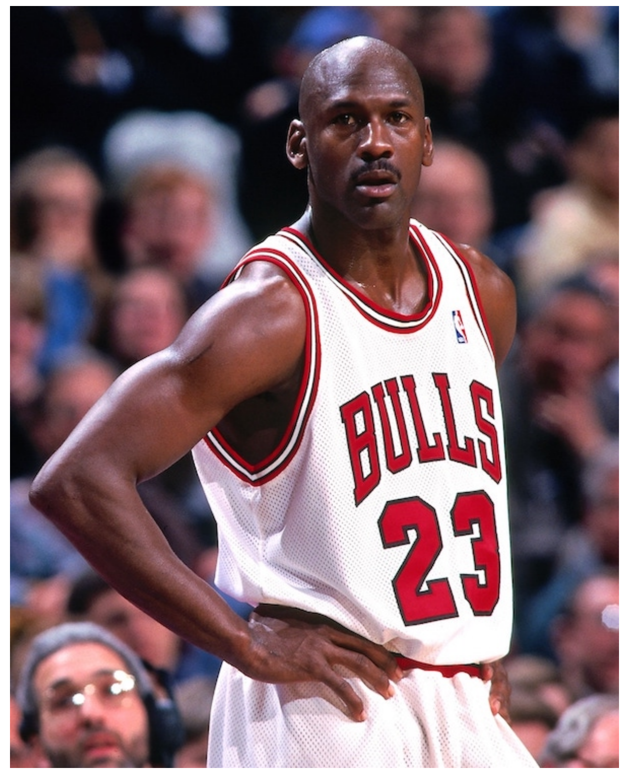
Data versioning (2) - motivation example



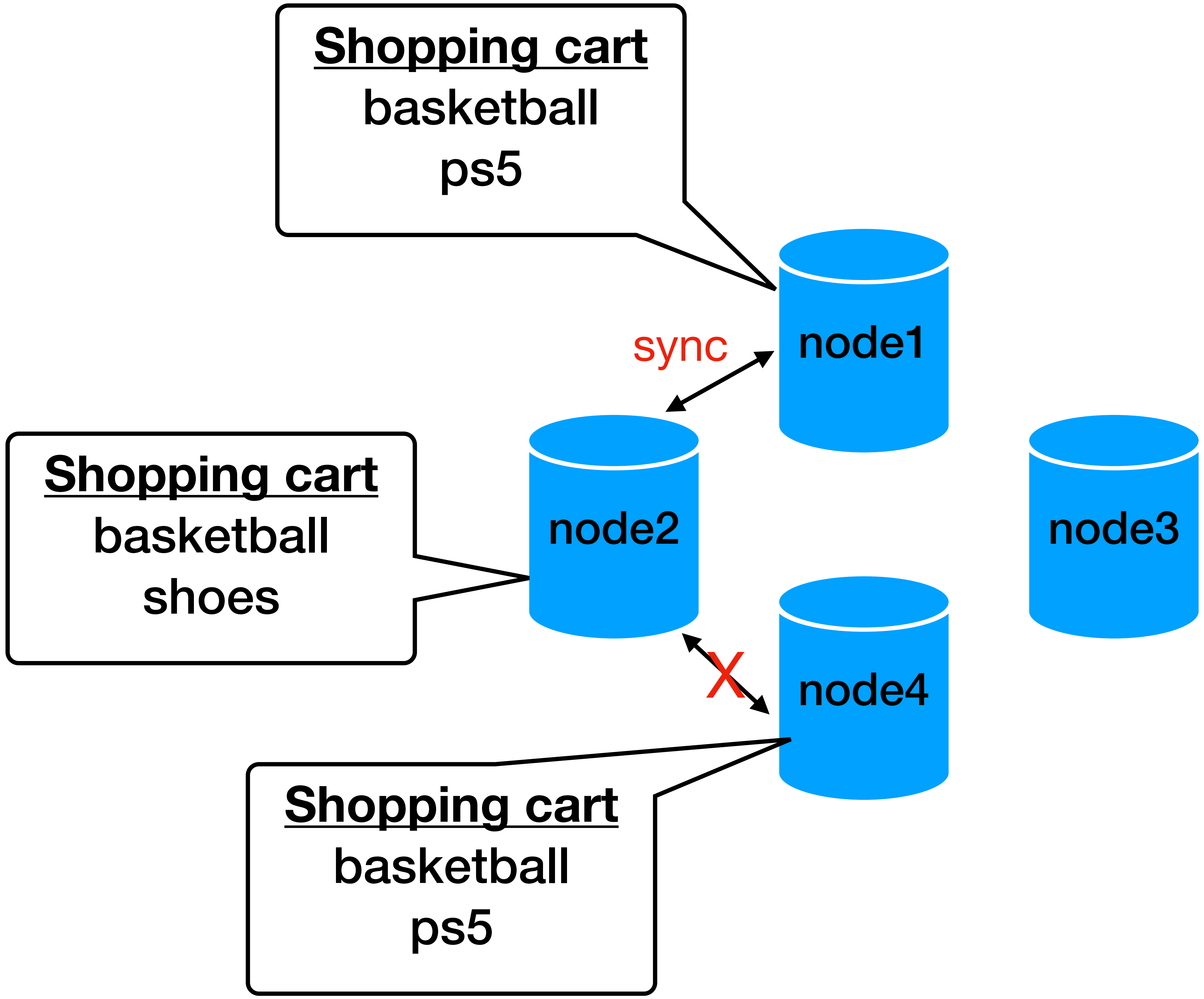
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



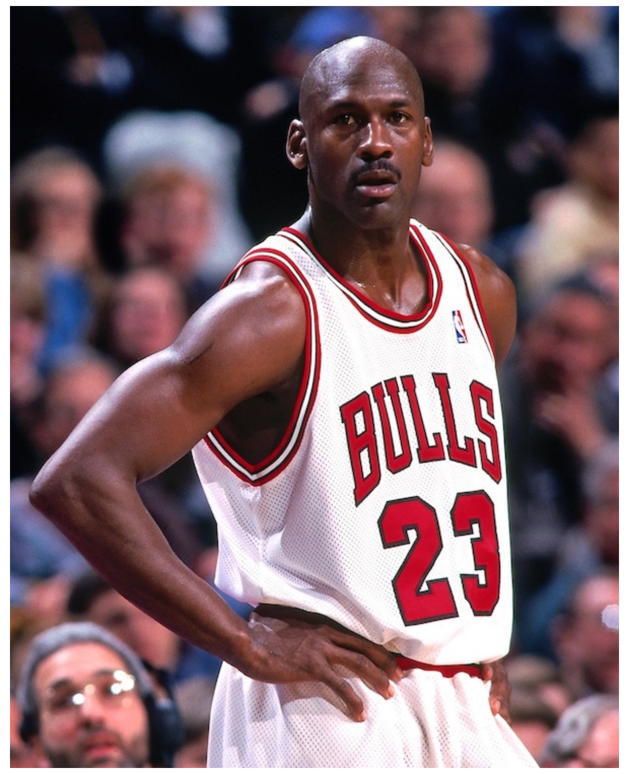
Data versioning (2) - motivation example



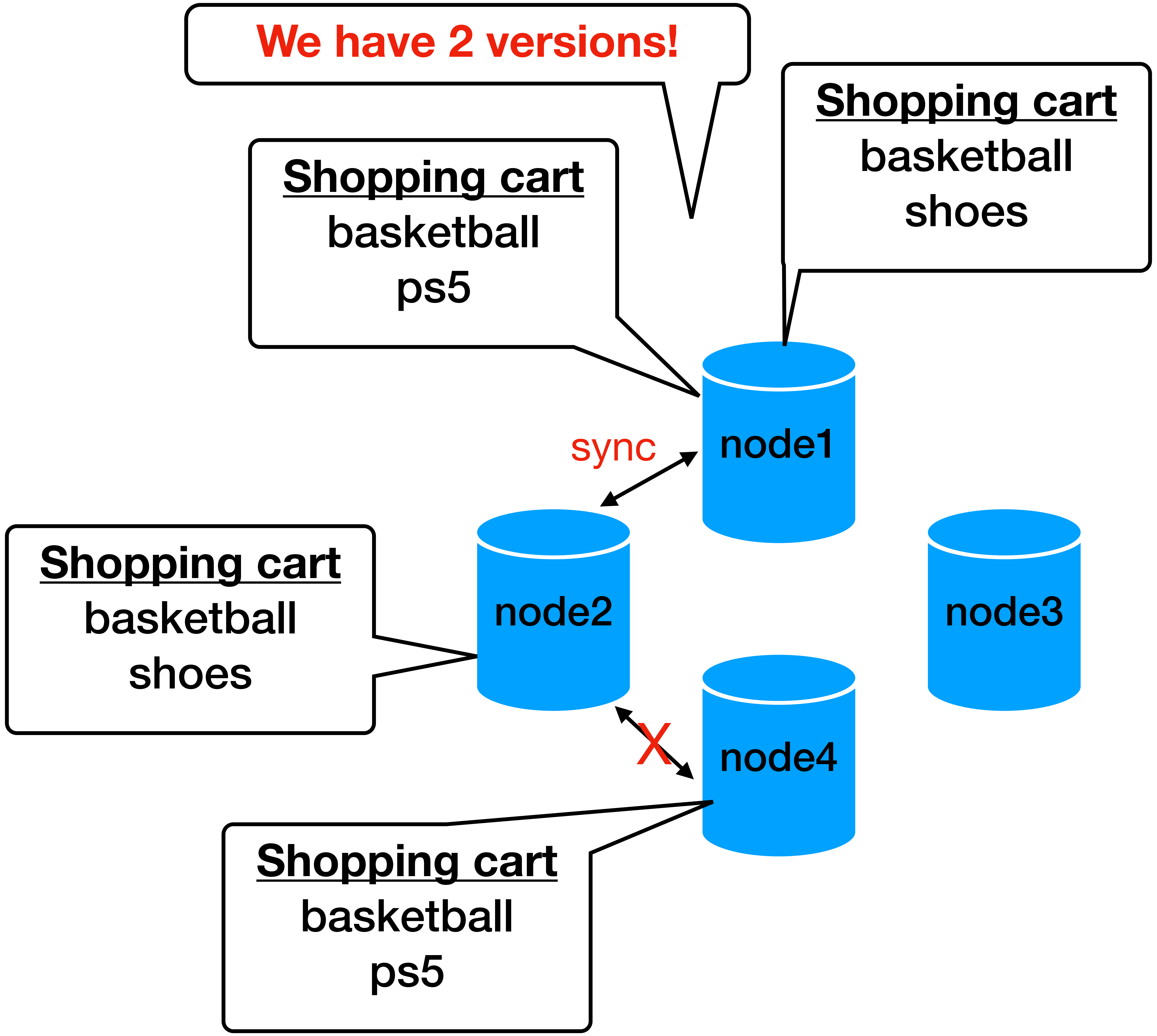
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



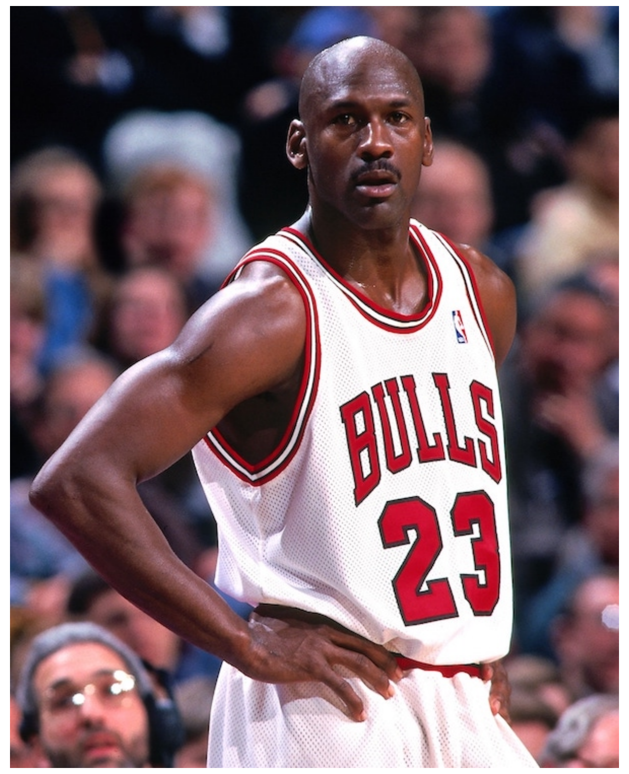
Data versioning (2) - motivation example



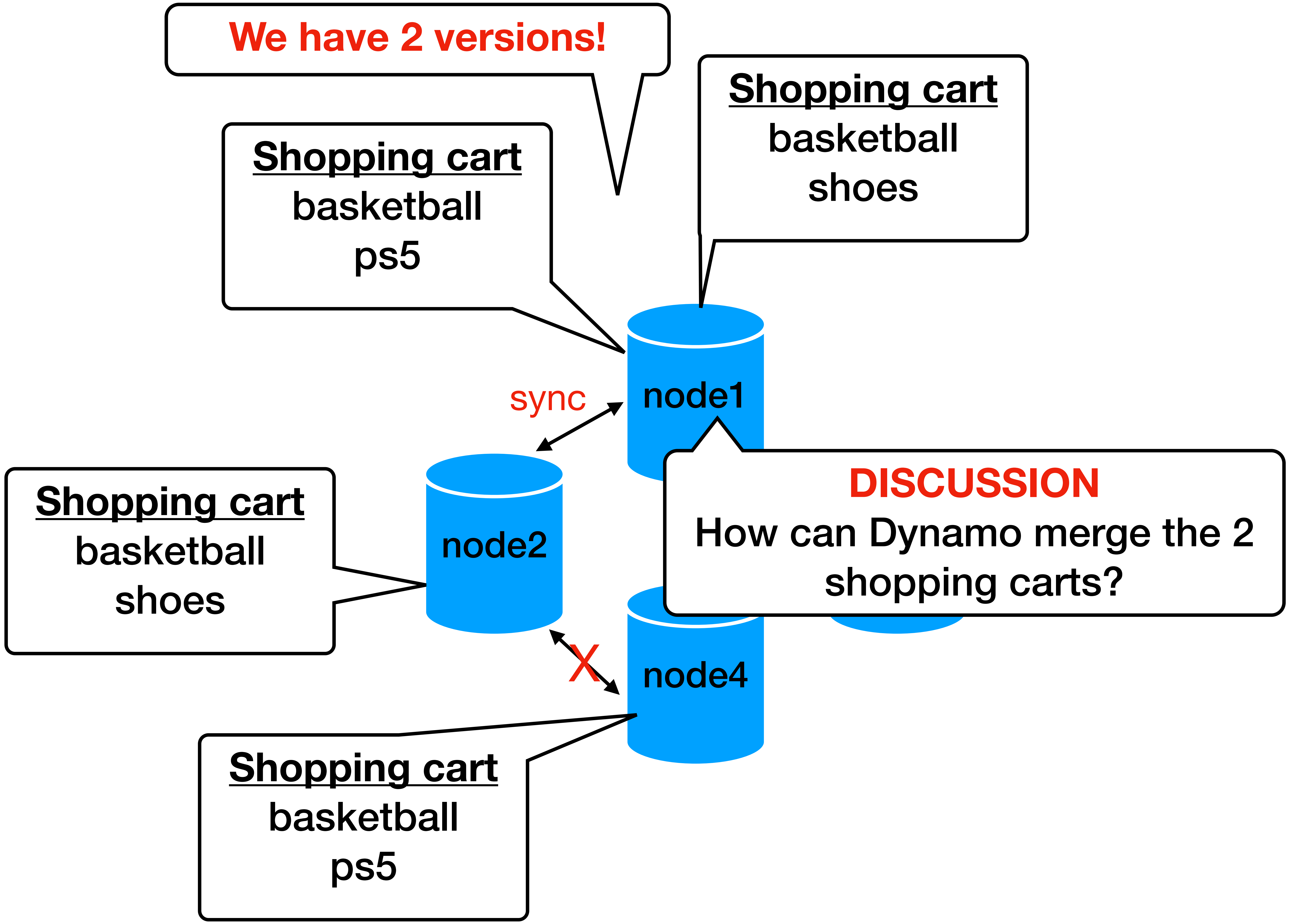
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



We have 2 versions!

Shopping cart
basketball
ps5

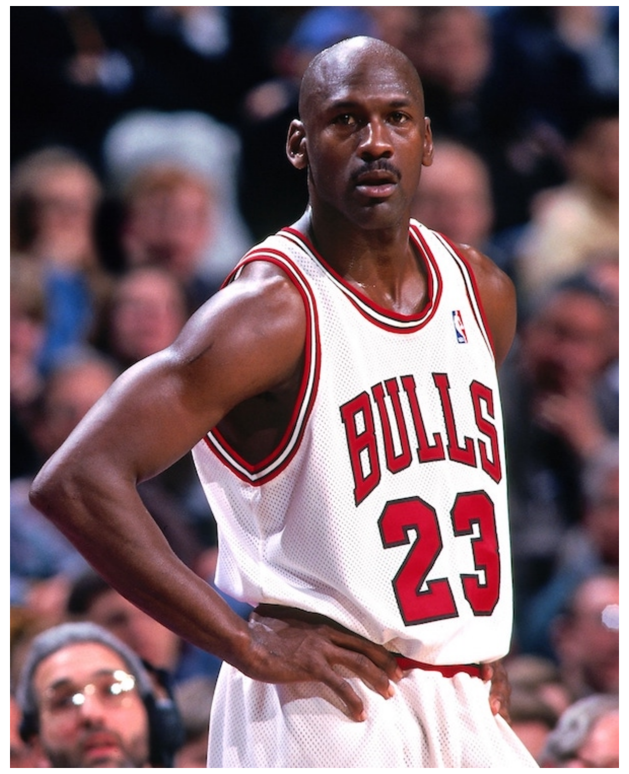
Shopping cart
basketball
shoes

Shopping cart
basketball
shoes

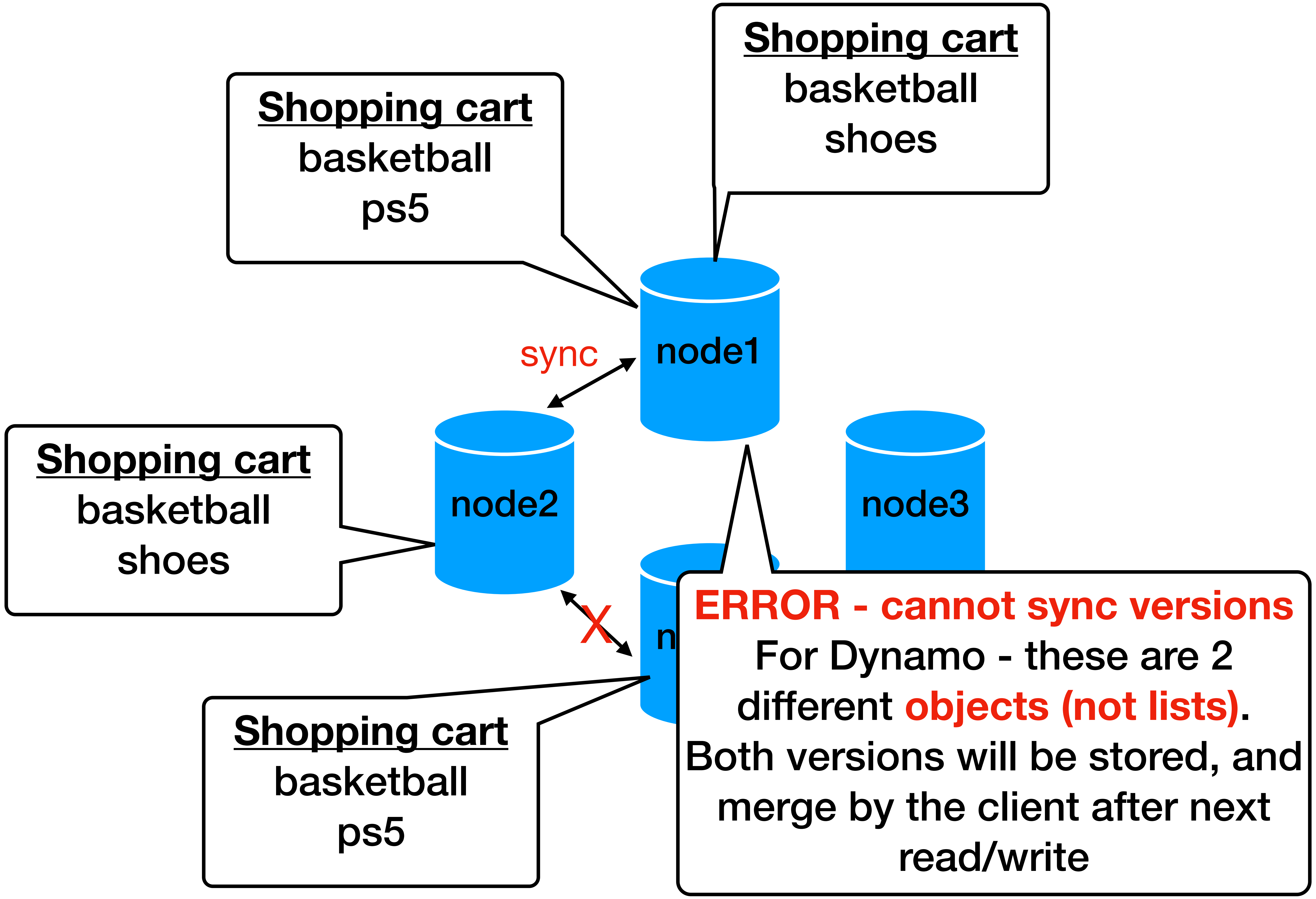
Shopping cart
basketball
ps5

DISCUSSION
How can Dynamo merge the 2 shopping carts?

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



Shopping cart
basketball
ps5

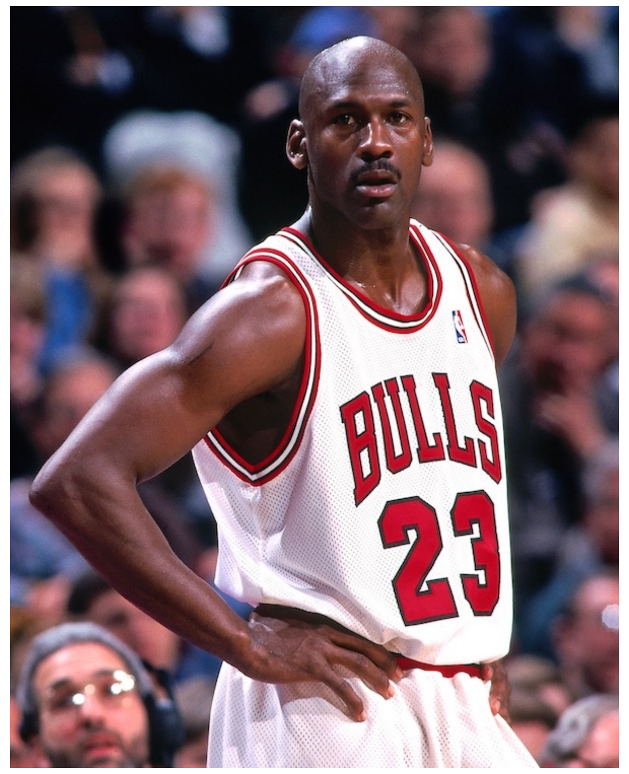
Shopping cart
basketball
shoes

Shopping cart
basketball
shoes

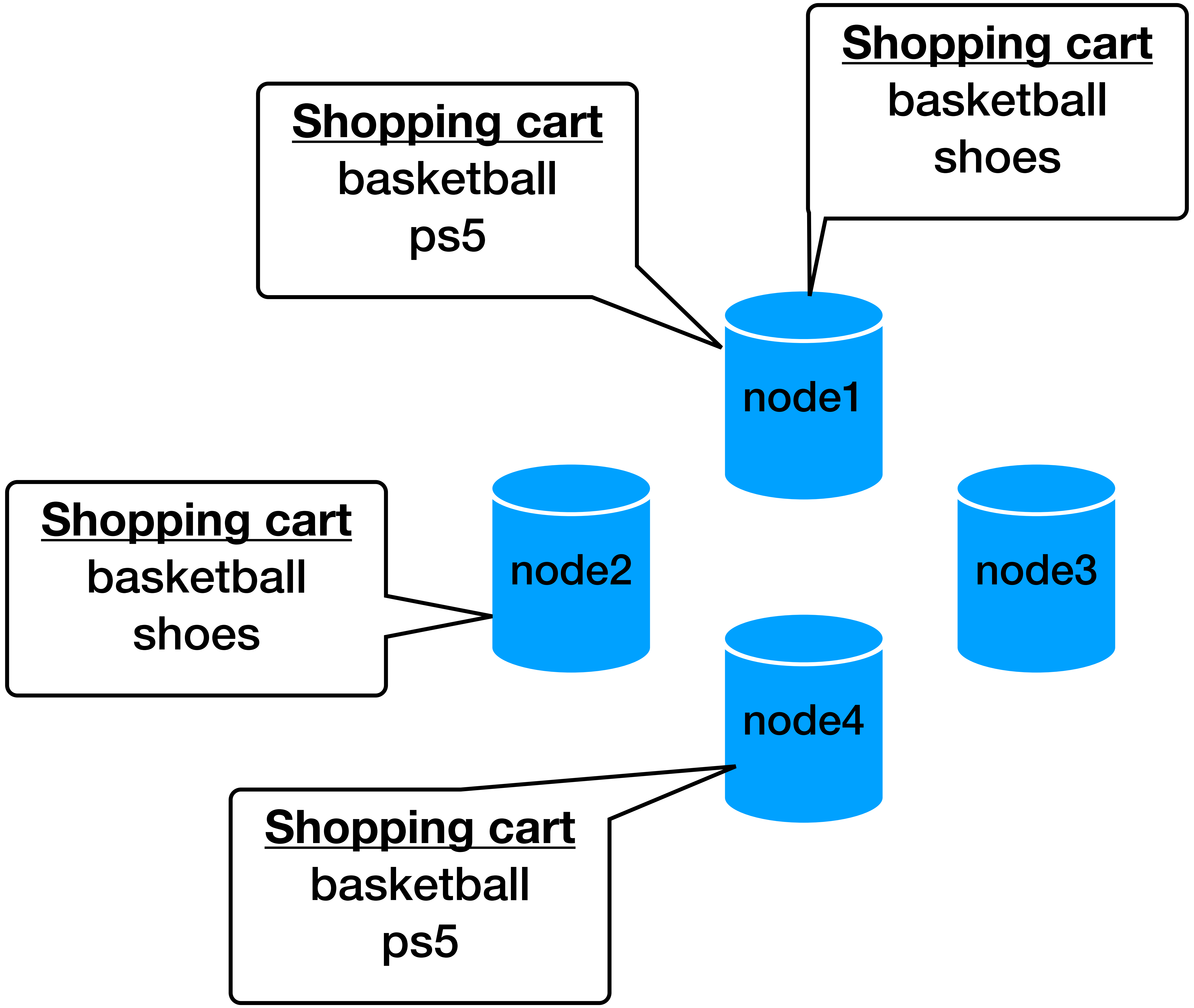
Shopping cart
basketball
ps5

ERROR - cannot sync versions
For Dynamo - these are 2 different **objects (not lists)**.
Both versions will be stored, and merge by the client after next read/write

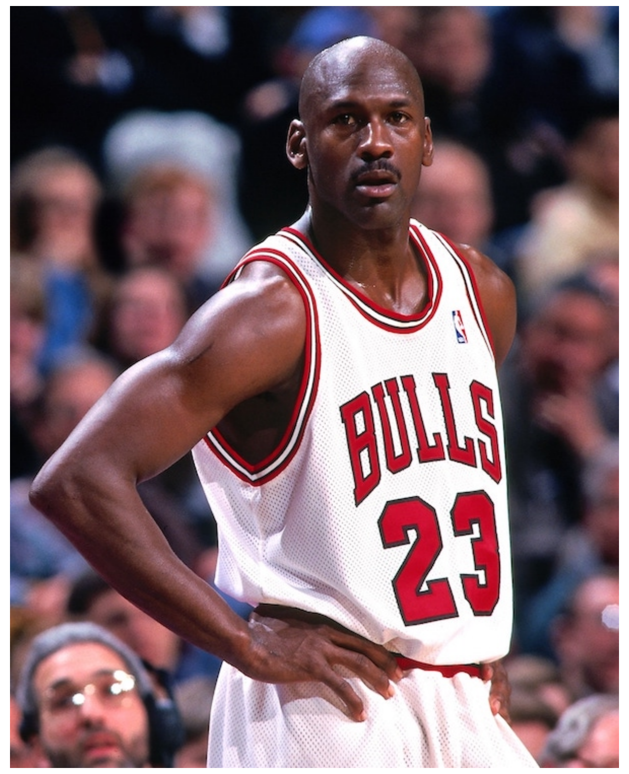
Data versioning (2) - motivation example



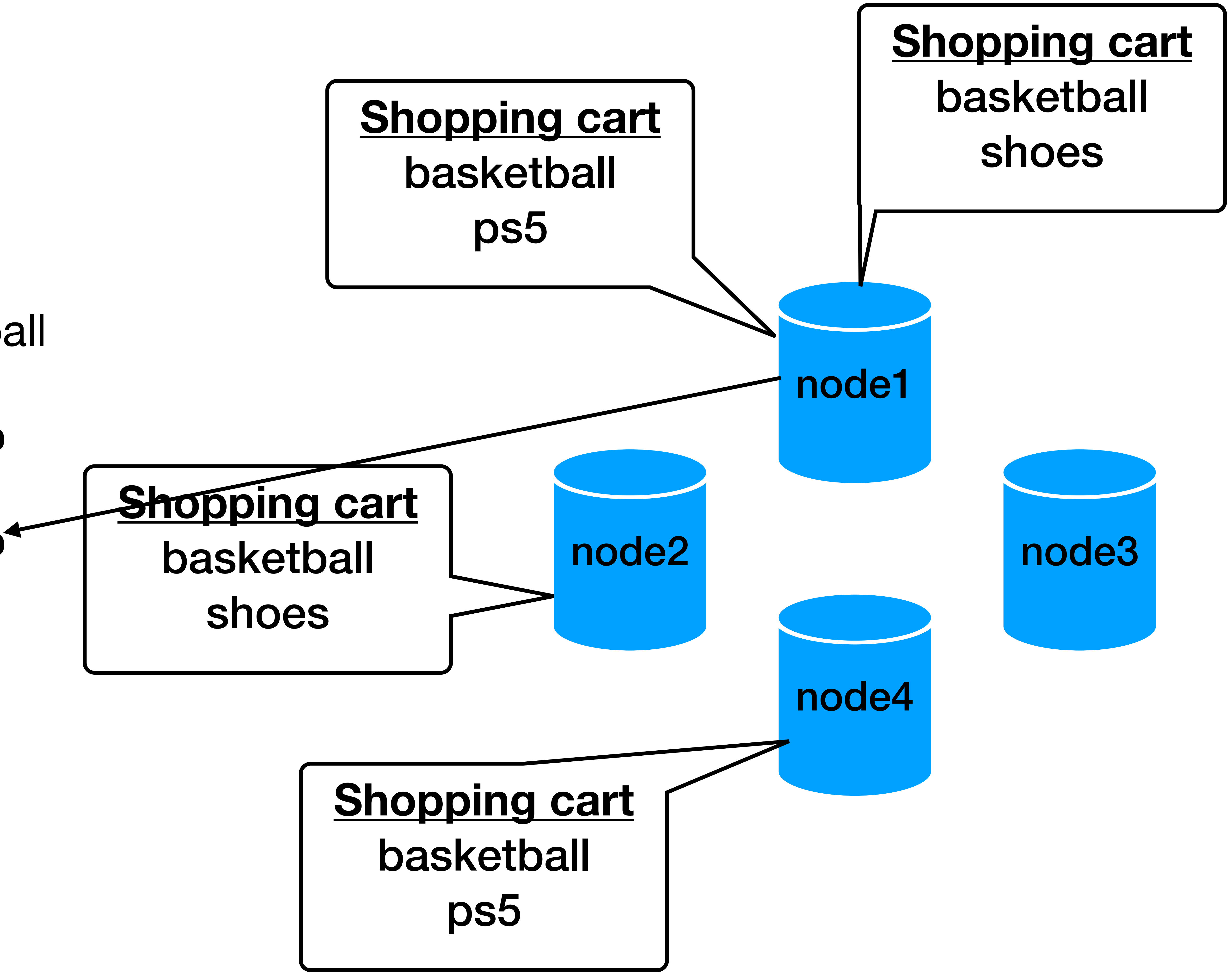
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



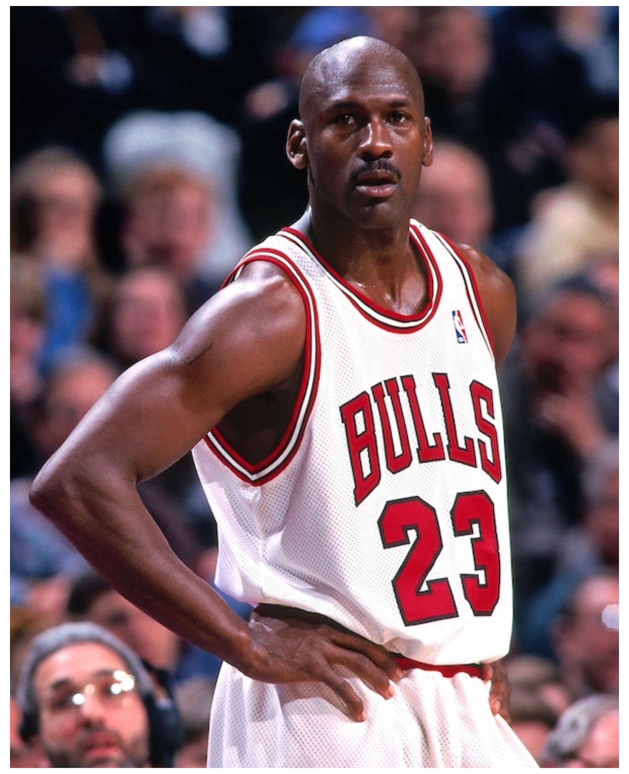
Data versioning (2) - motivation example



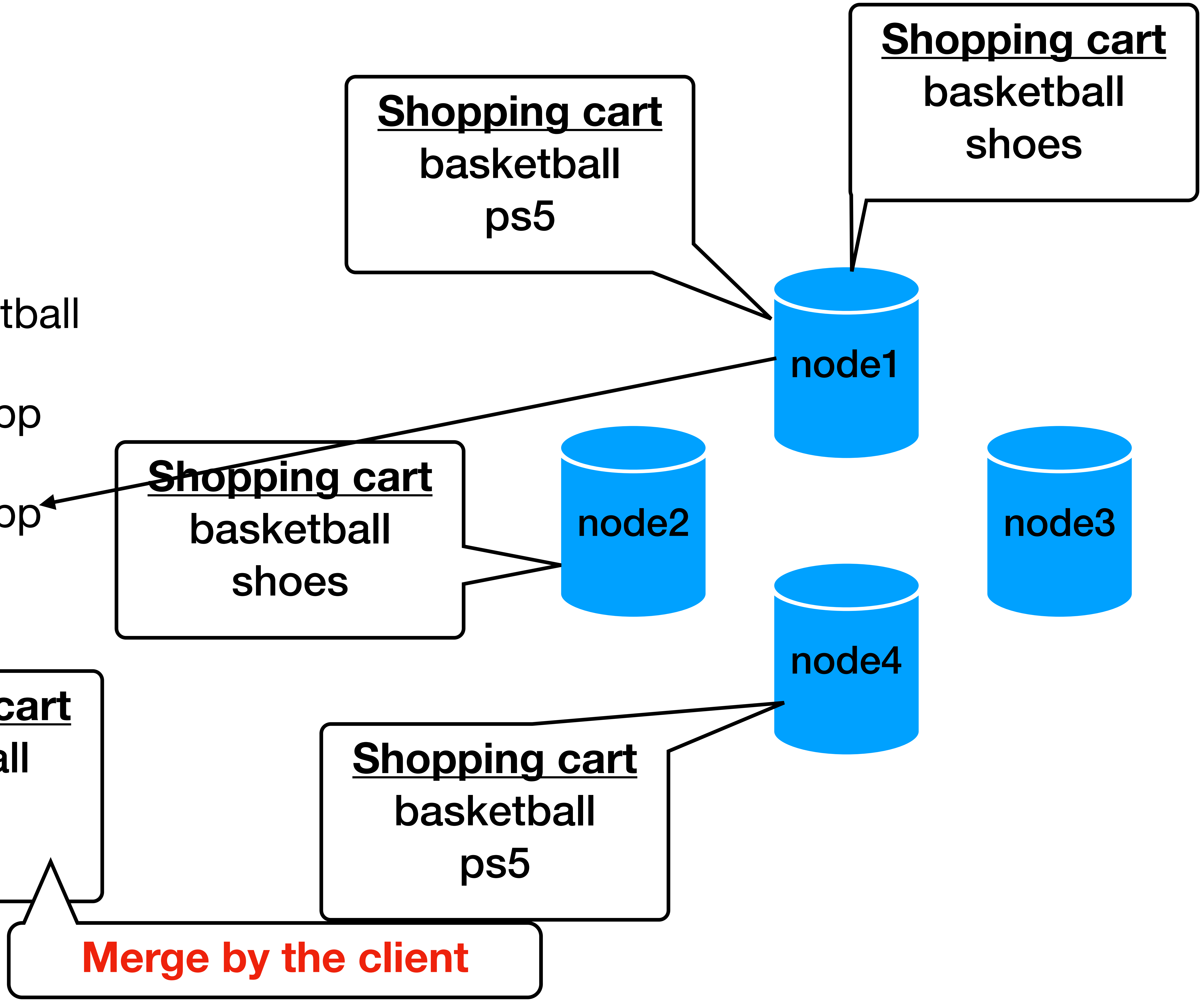
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app



Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app



Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

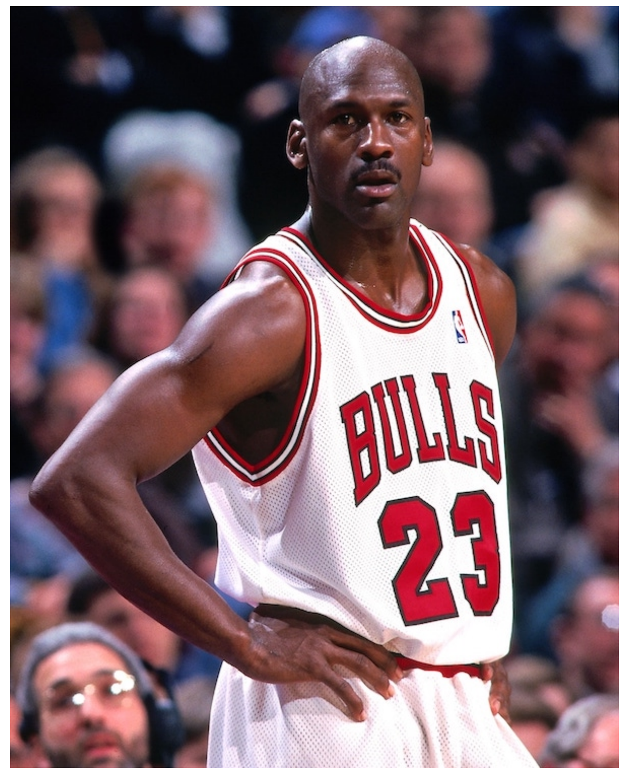
Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

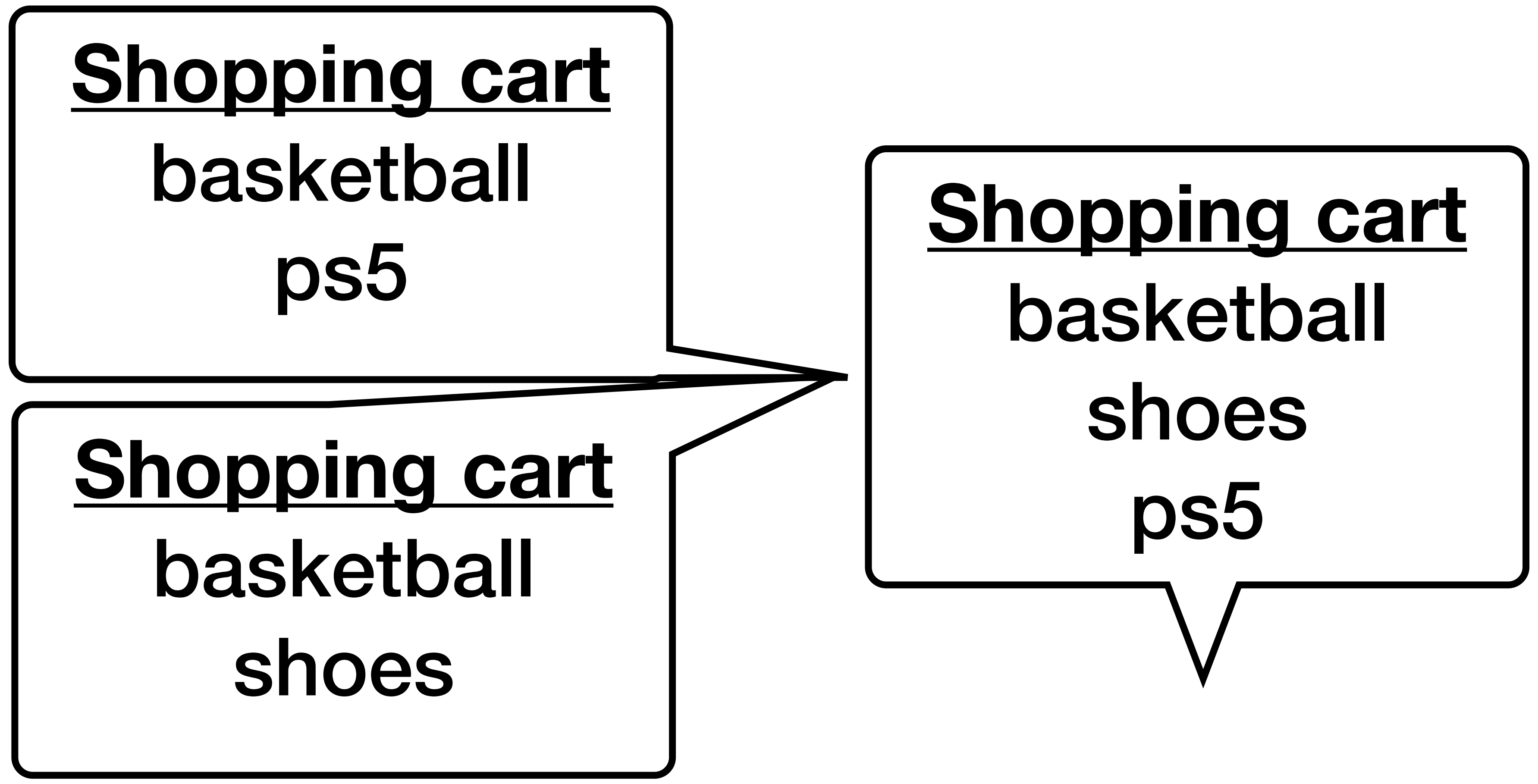
Shopping cart
basketball
shoes
ps5

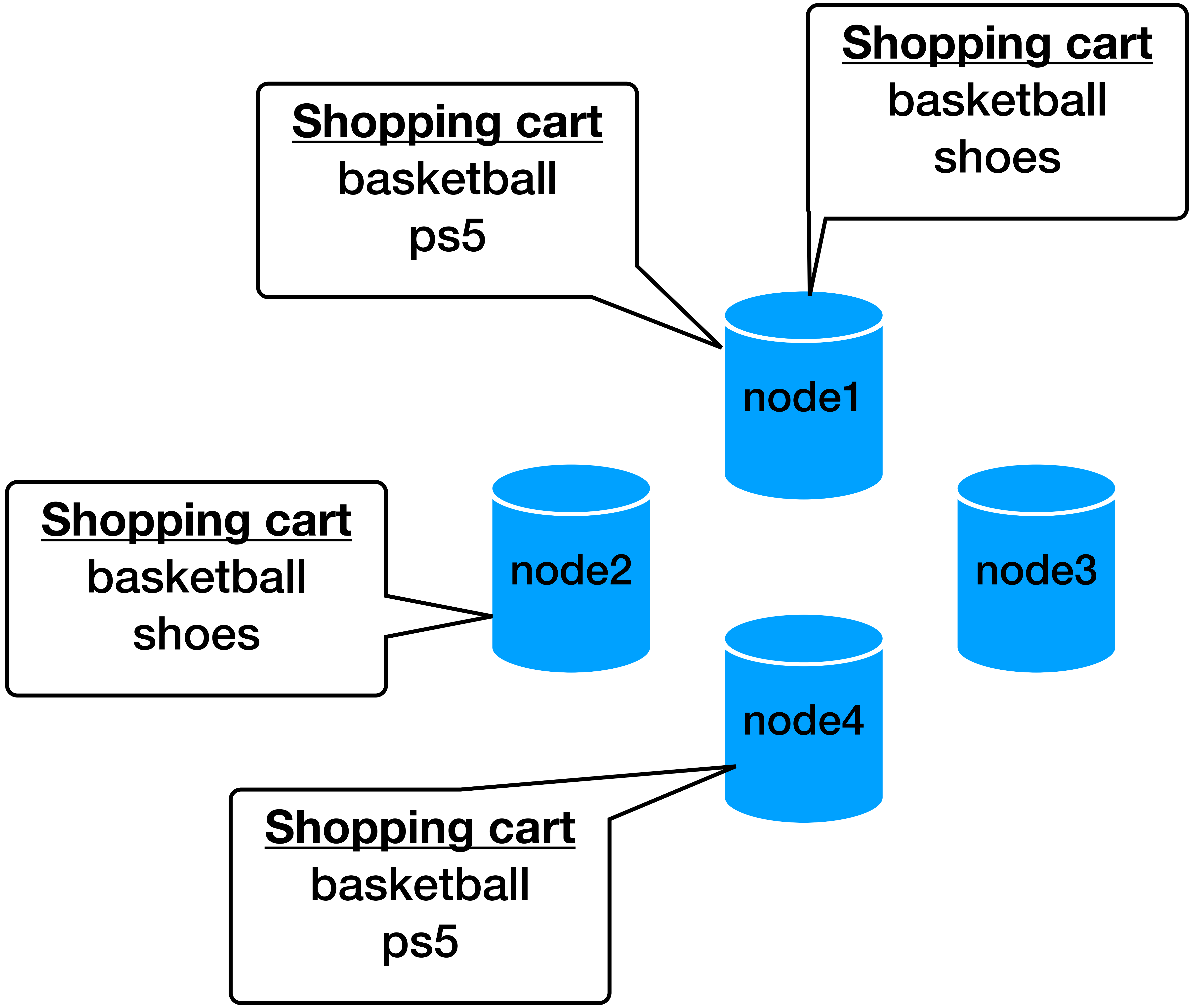
Merge by the client

Data versioning (2) - motivation example

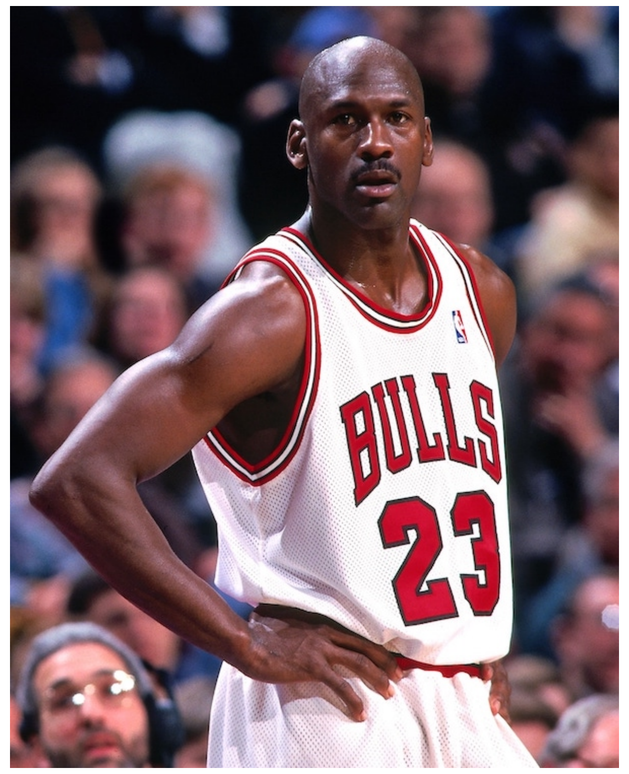


10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

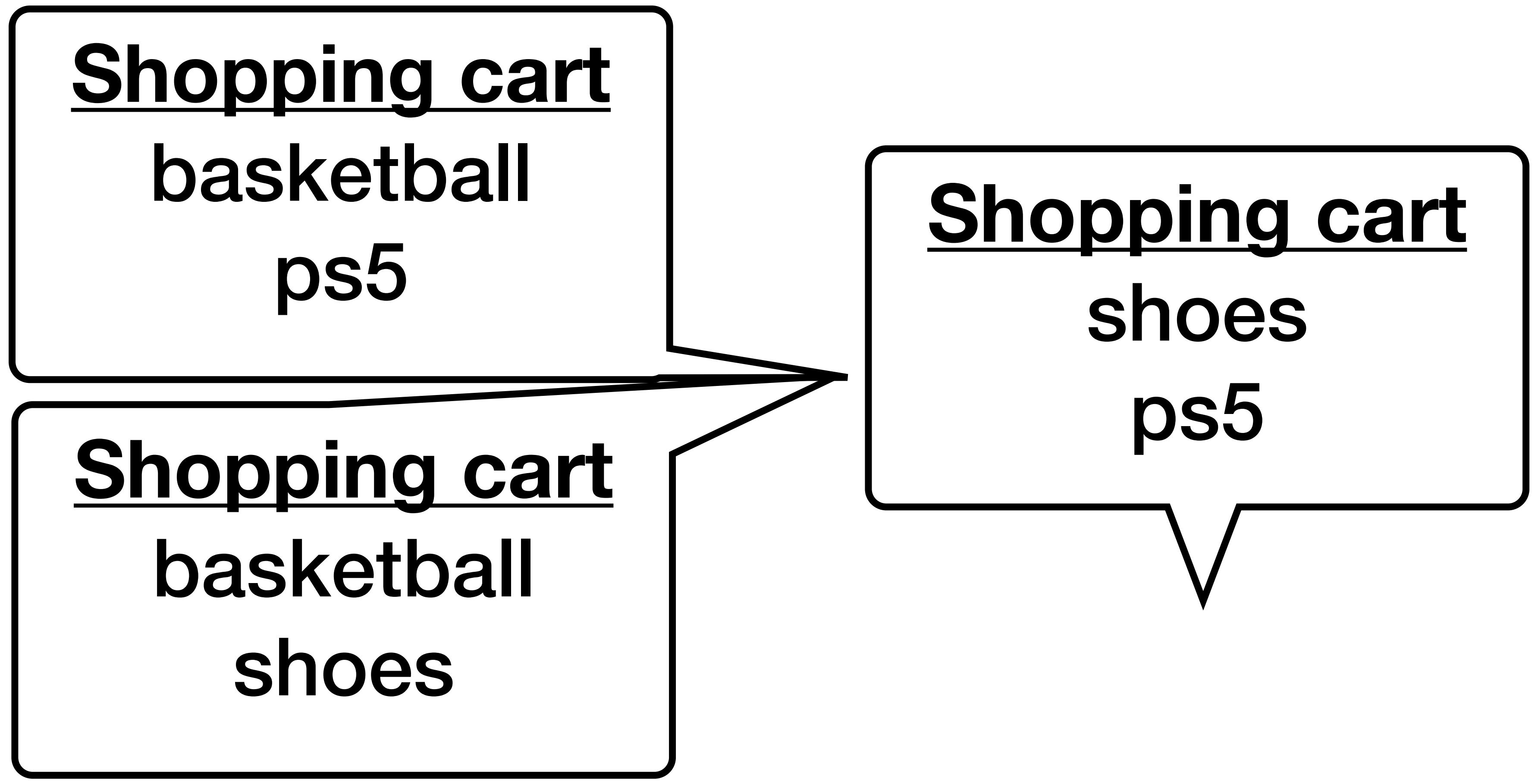




Data versioning (2) - motivation example



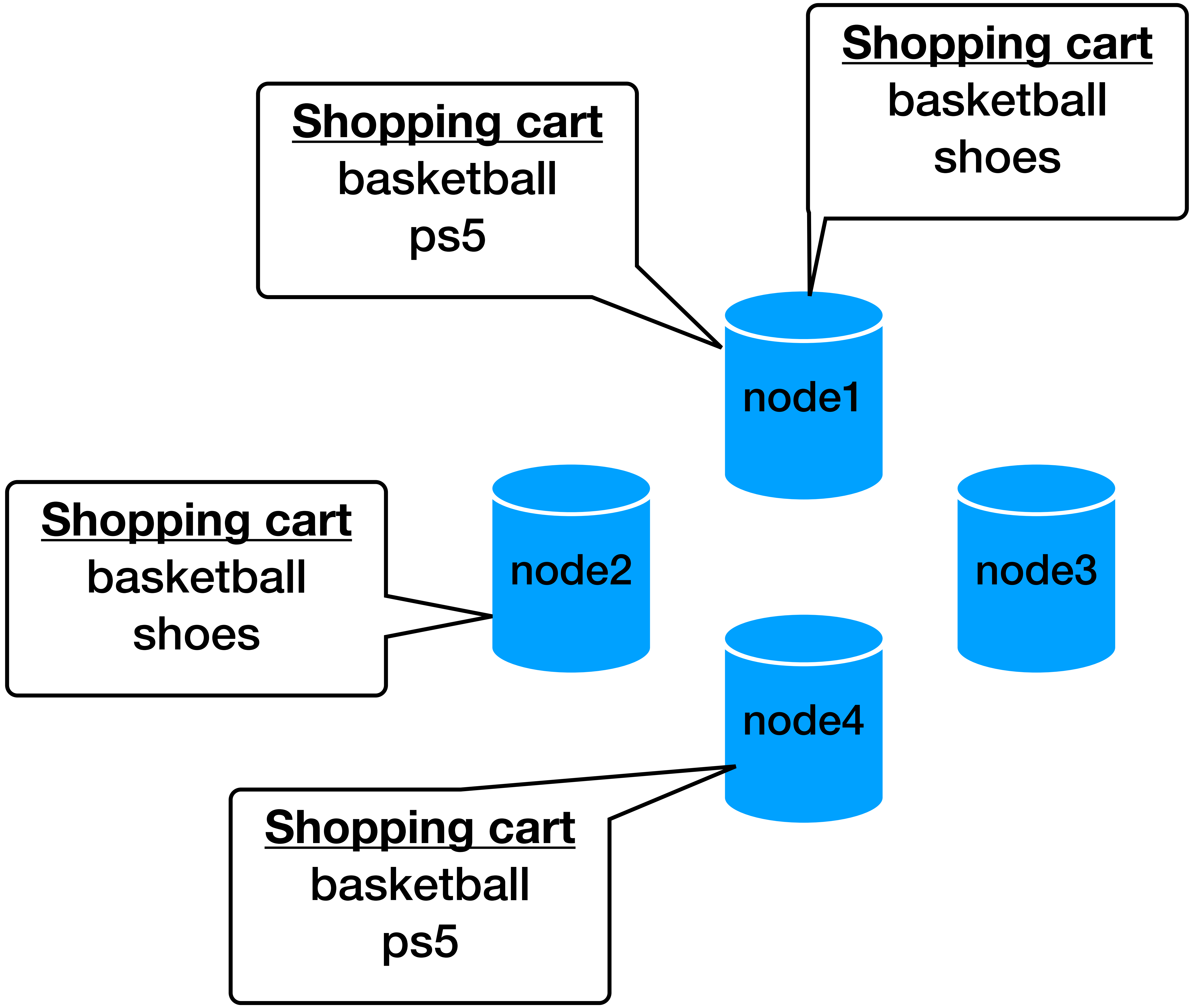
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
shoes
ps5



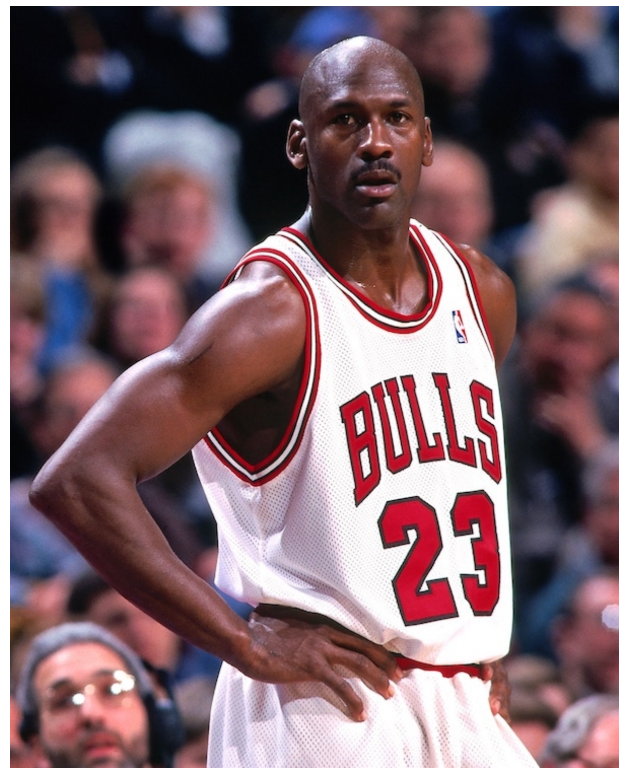
Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

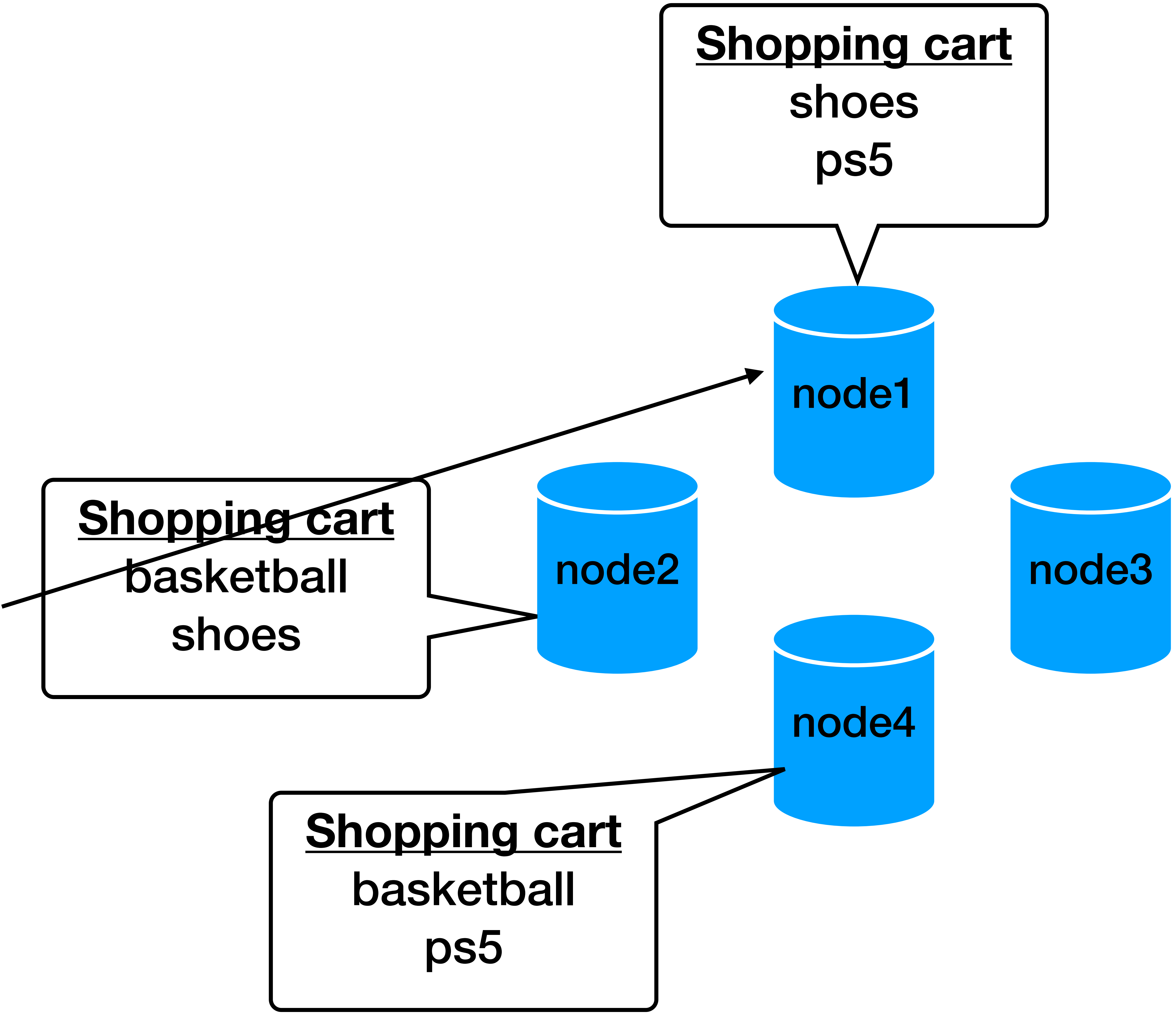
Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

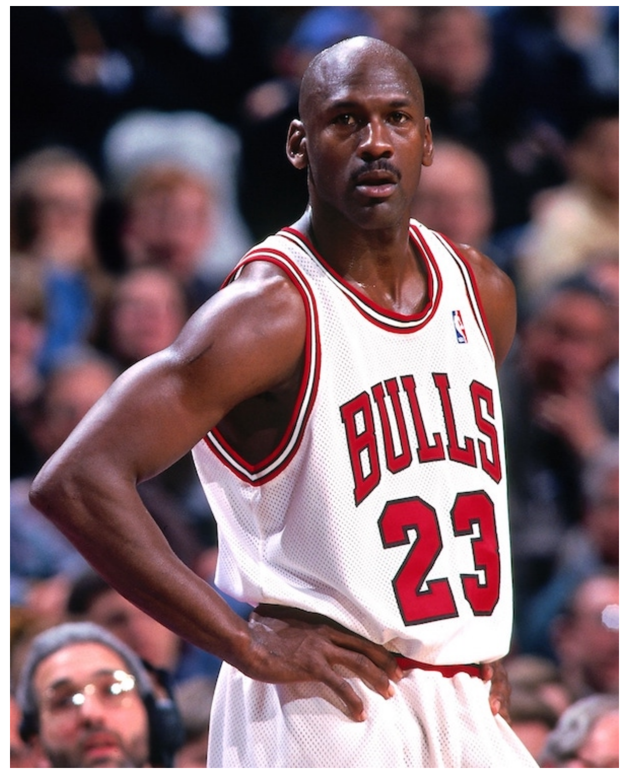


Shopping cart
shoes
ps5

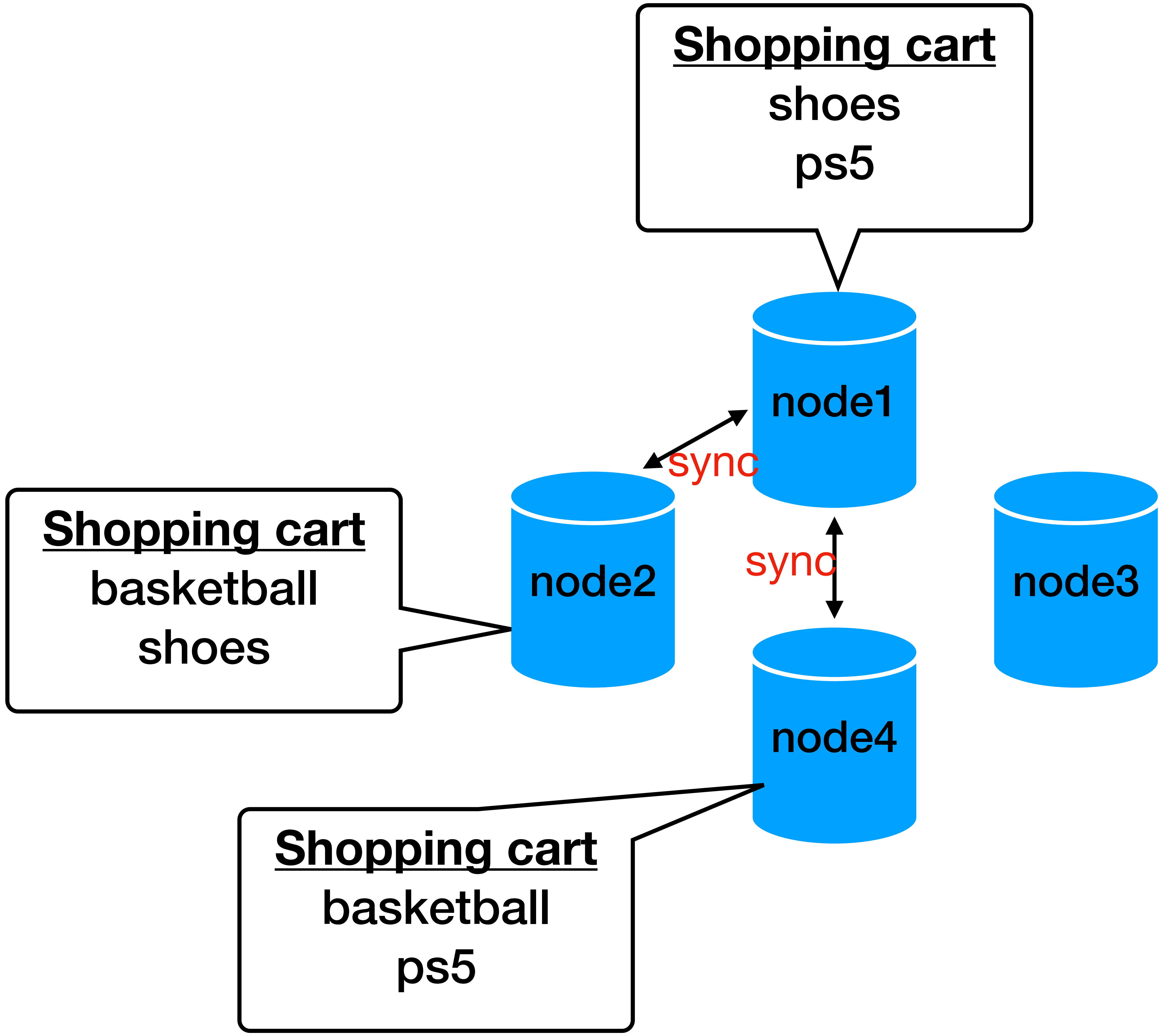
Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

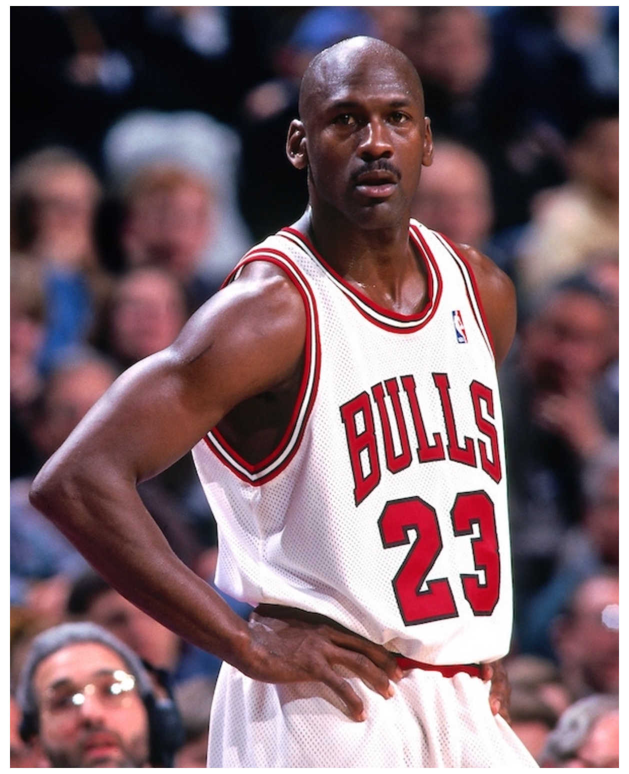


Shopping cart
shoes
ps5

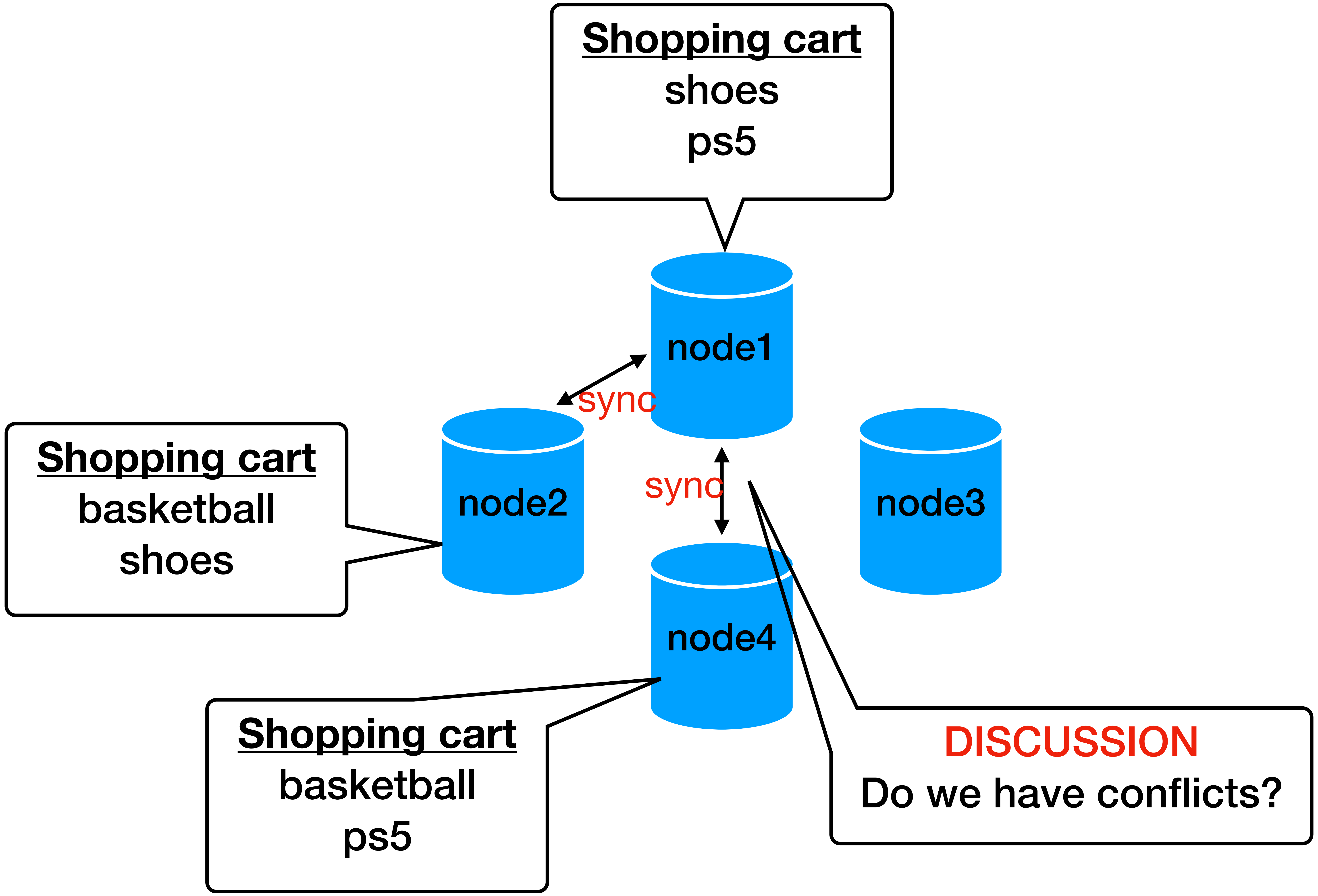
Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Data versioning (2) - motivation example

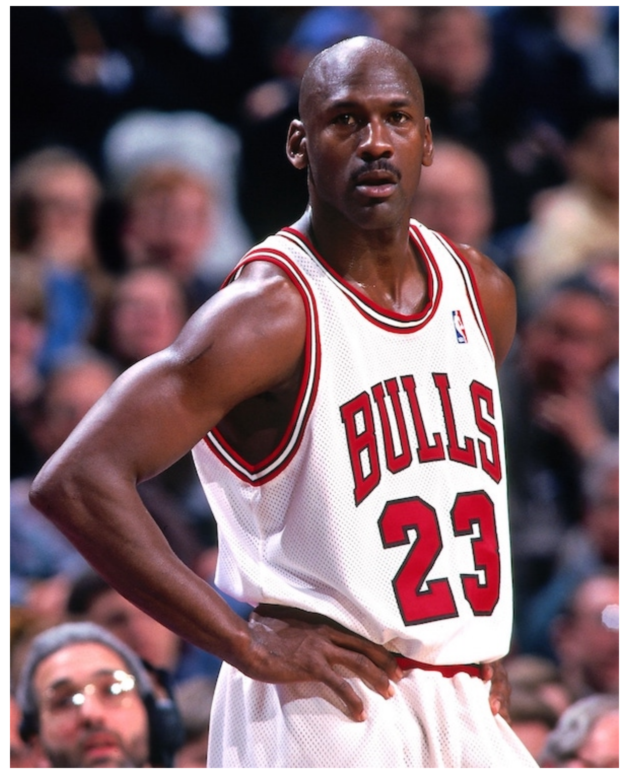


10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

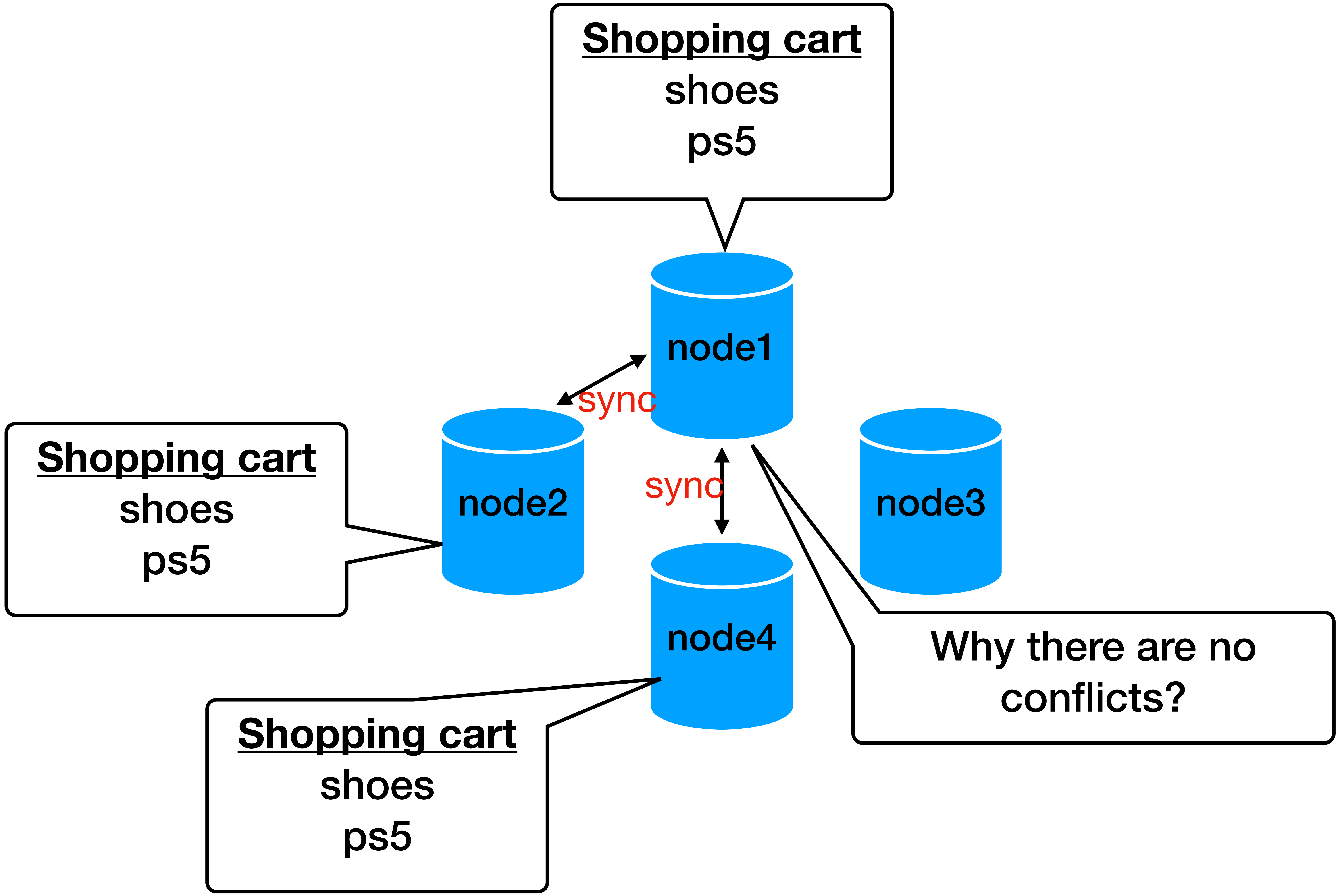


DISCUSSION
Do we have conflicts?

Data versioning (2) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



Why there are no conflicts?

Data versioning (3) - Vector clocks

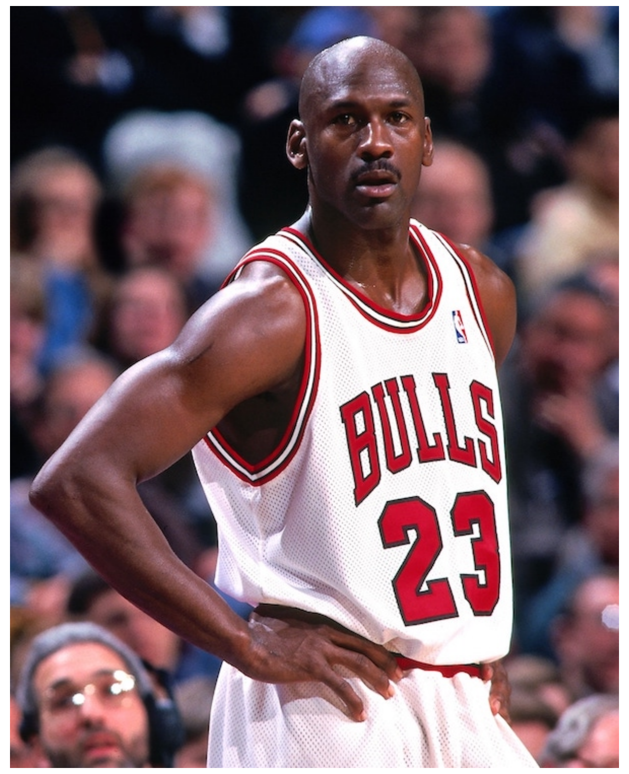
- Used to capture causality between versions
  (of the same object)
- Vector clock = a list of `[node, counter]` pairs
  one list is a attached to every version of every object

```
IF      all the counters on the first object’s clocks <=
        all the counters on the second object
THEN
        first is ancestor of the second and can be forgotten
ELSE
        there is a conflict, the client should reconcile
```

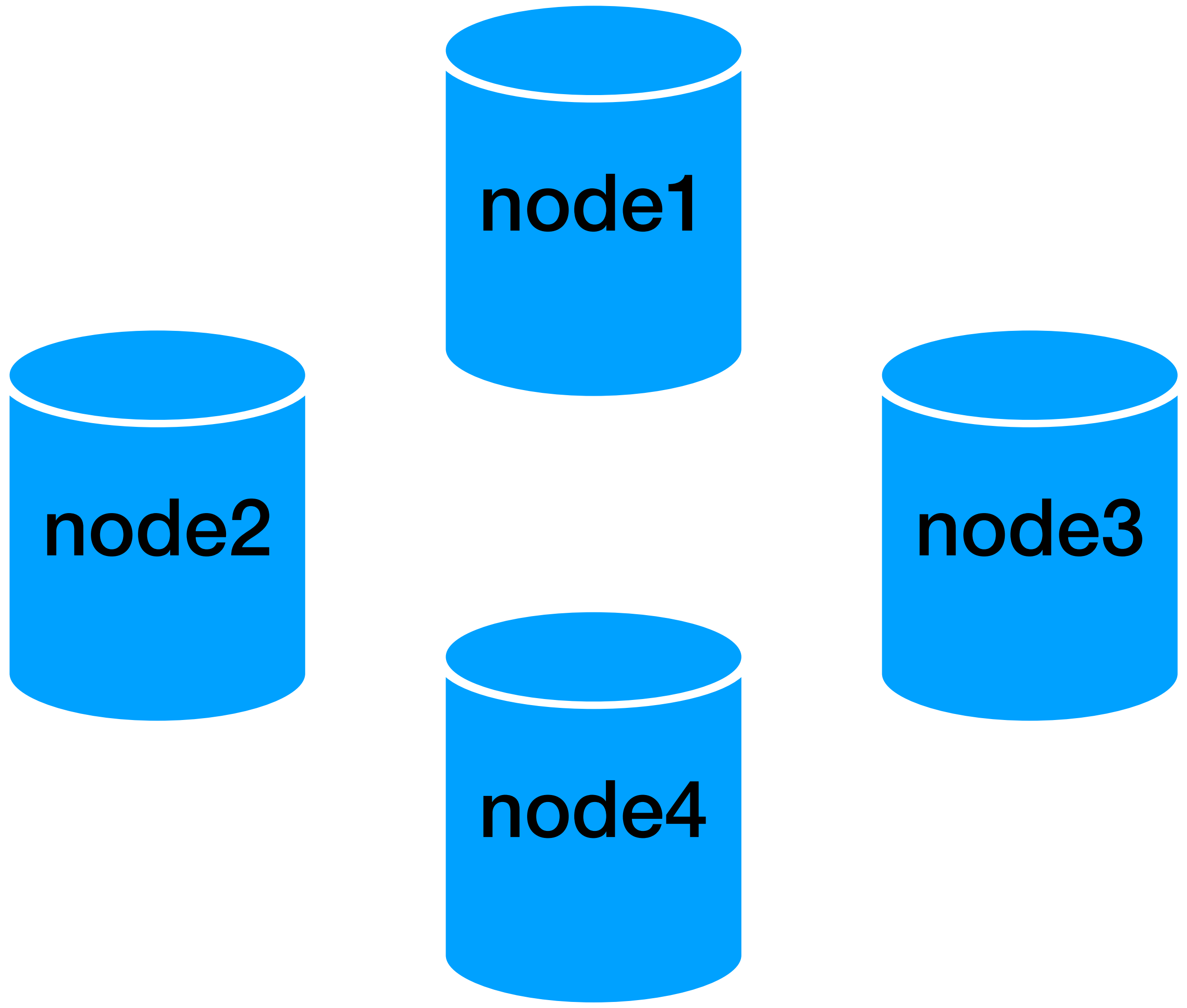
Data versioning (4) - Interface

- `put(key, context, object)`
- `get(key)`
  - `get` returns all versions of the associated object AND a `context`
  - `context` = system metadata / versioning (opaque to the user) holds the vector clocks
- If the response of a `get()` contained multiple versions, the next update (with the retrieved `context`) will reconcile the versions

Data versioning (5) - motivation example

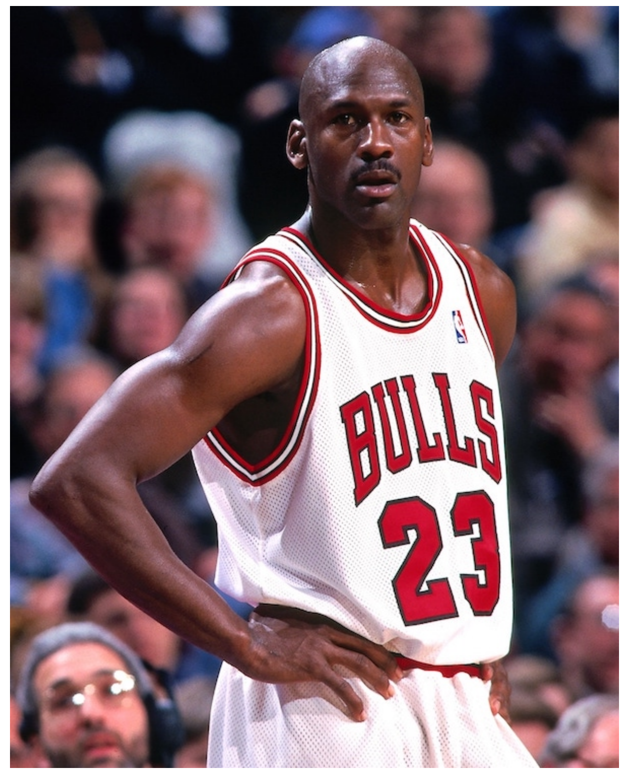


10:00: empty cart

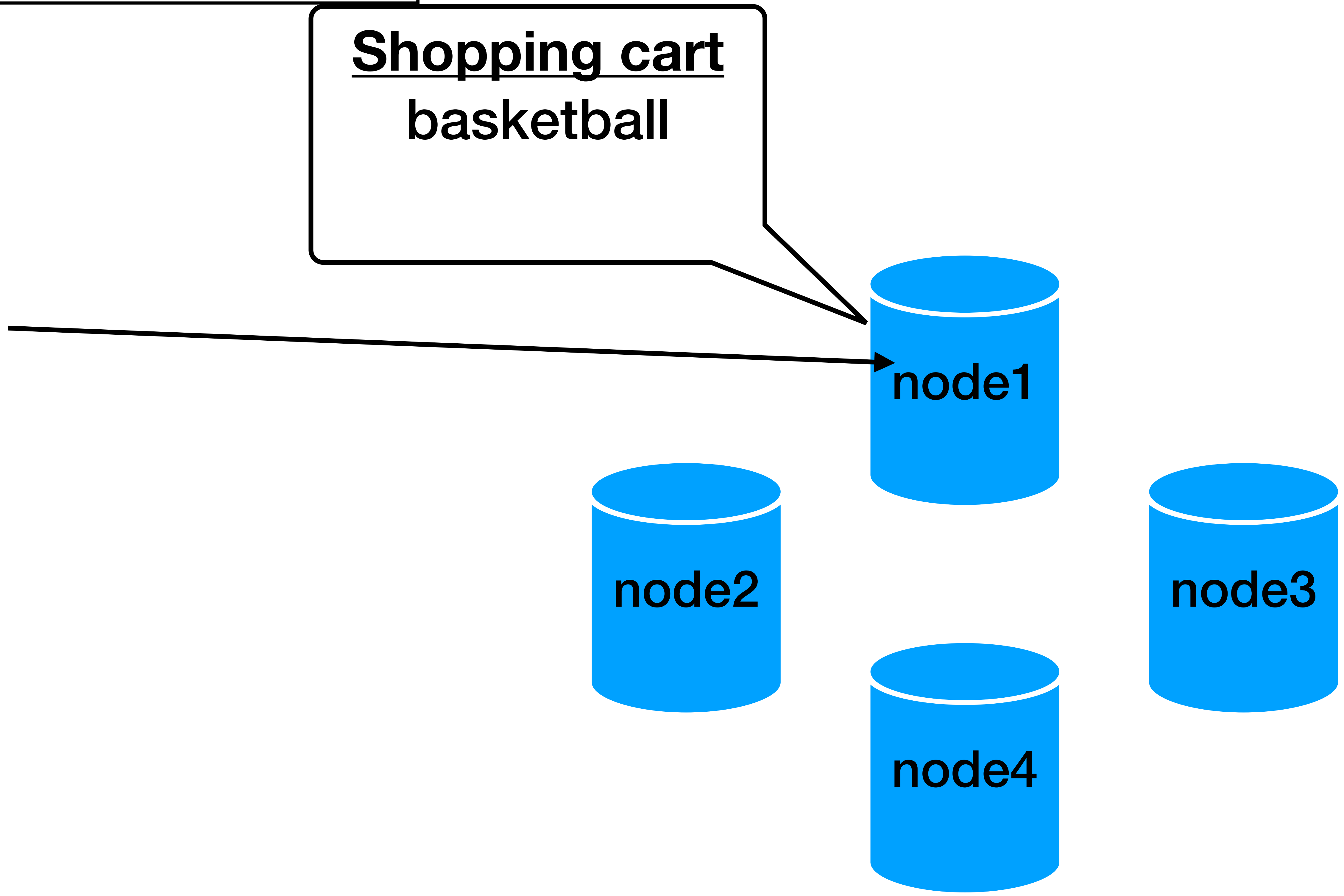


Data versioning (5) - motivation example

D1 `([node1,1])`



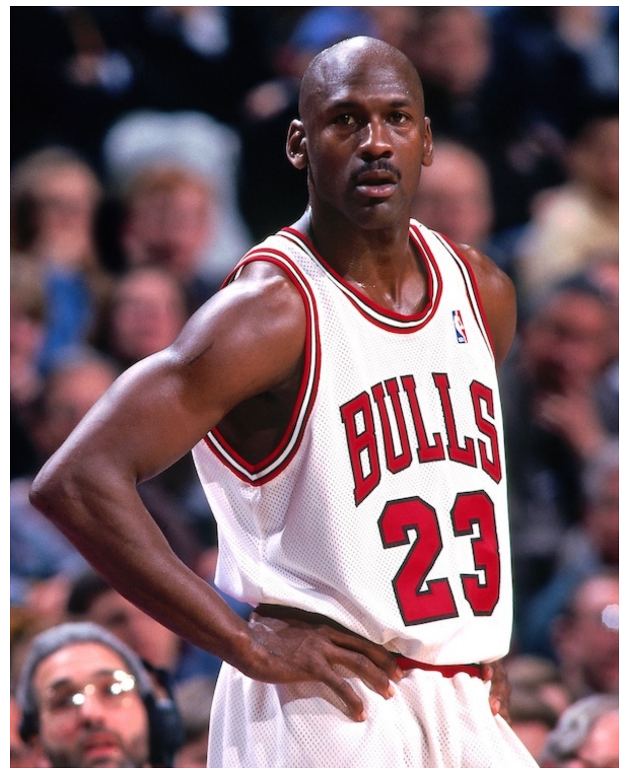
10:00: empty cart
10:01: added basketball



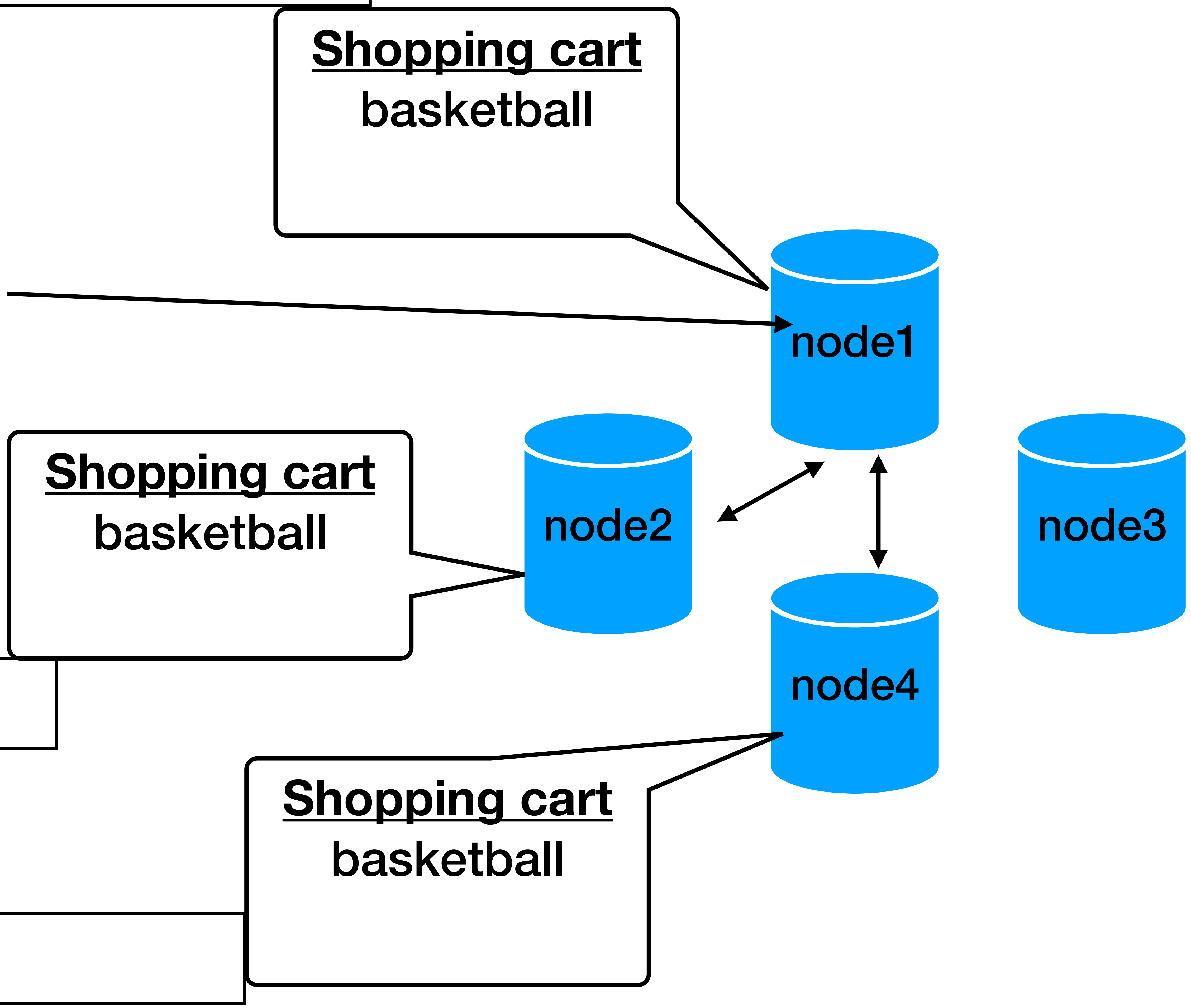
<u>Shopping cart</u>
basketball

Data versioning (5) - motivation example

D1 ([node1,1])



10:00: empty cart
10:01: added basketball



Shopping cart
basketball

Shopping cart
basketball

D1 ([node1,1])

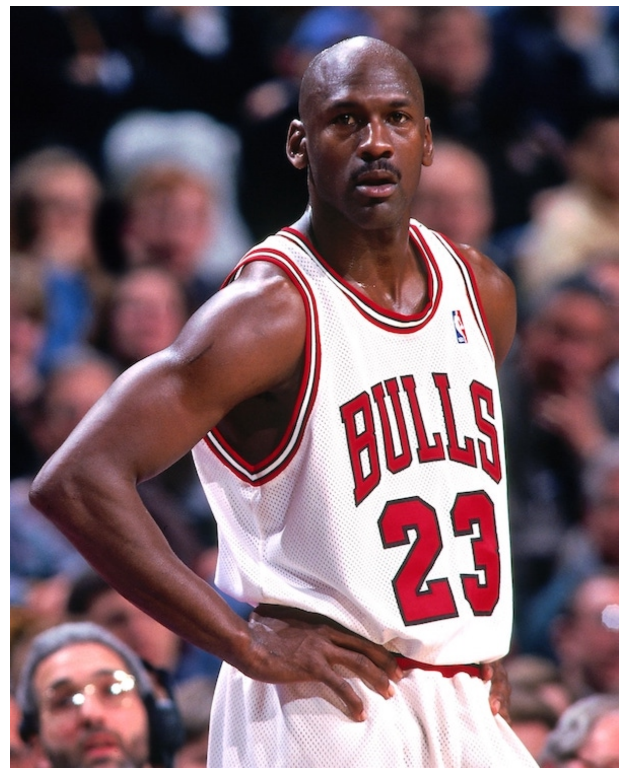
Shopping cart
basketball

D1 ([node1,1])

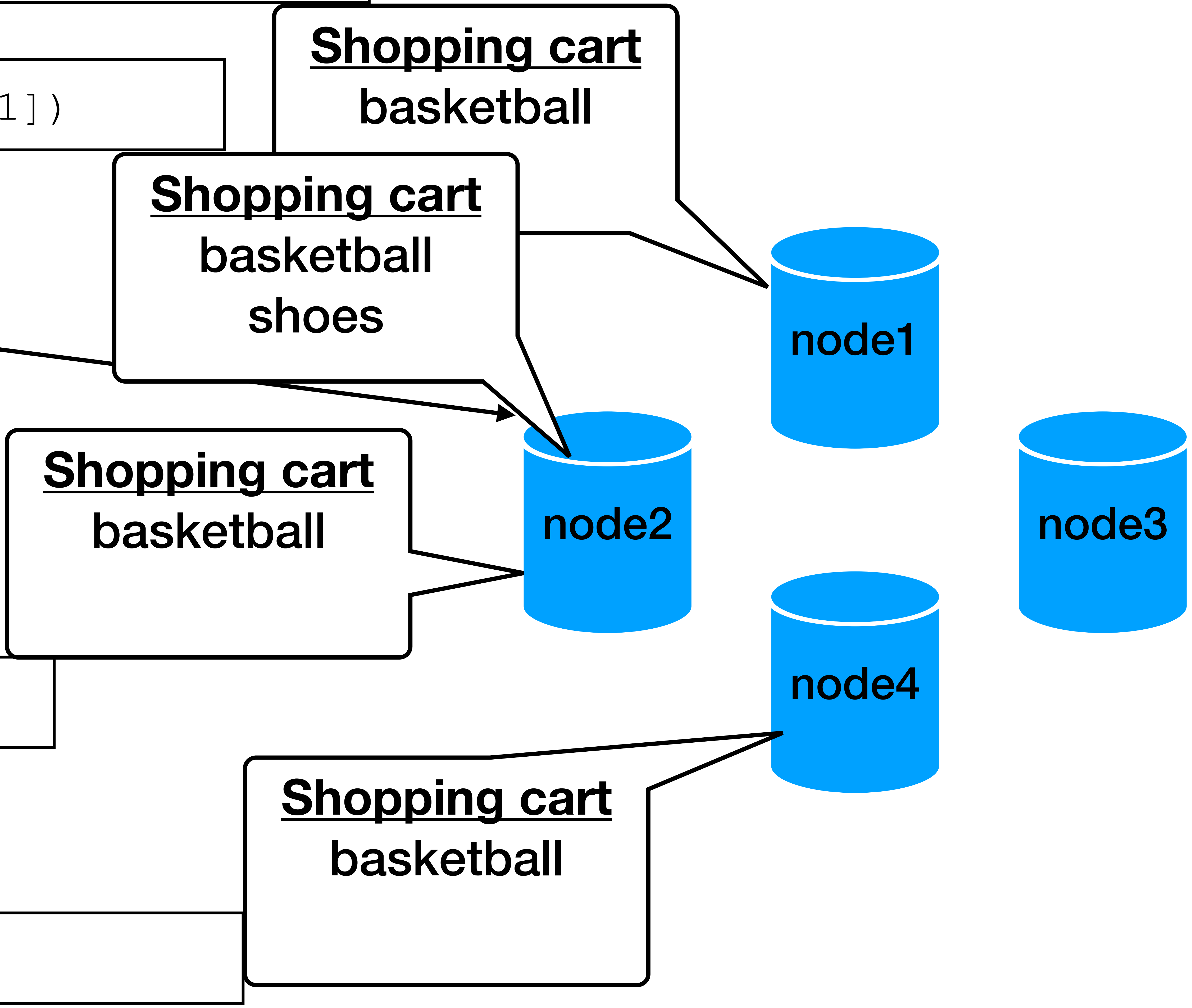
Data versioning (5) - motivation example

D1 `([node1,1])`

D2 `([node1,1],[node2,1])`



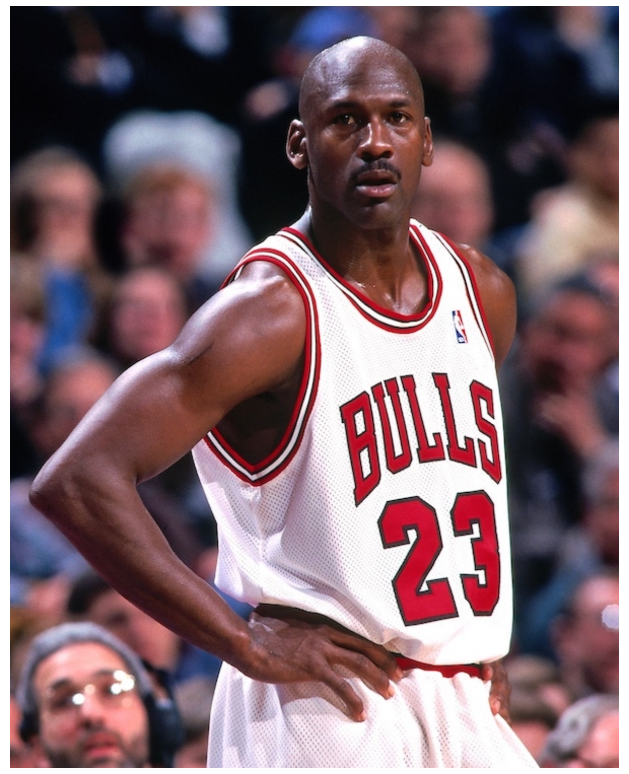
10:00: empty cart
10:01: added basketball
10:02: added shoes



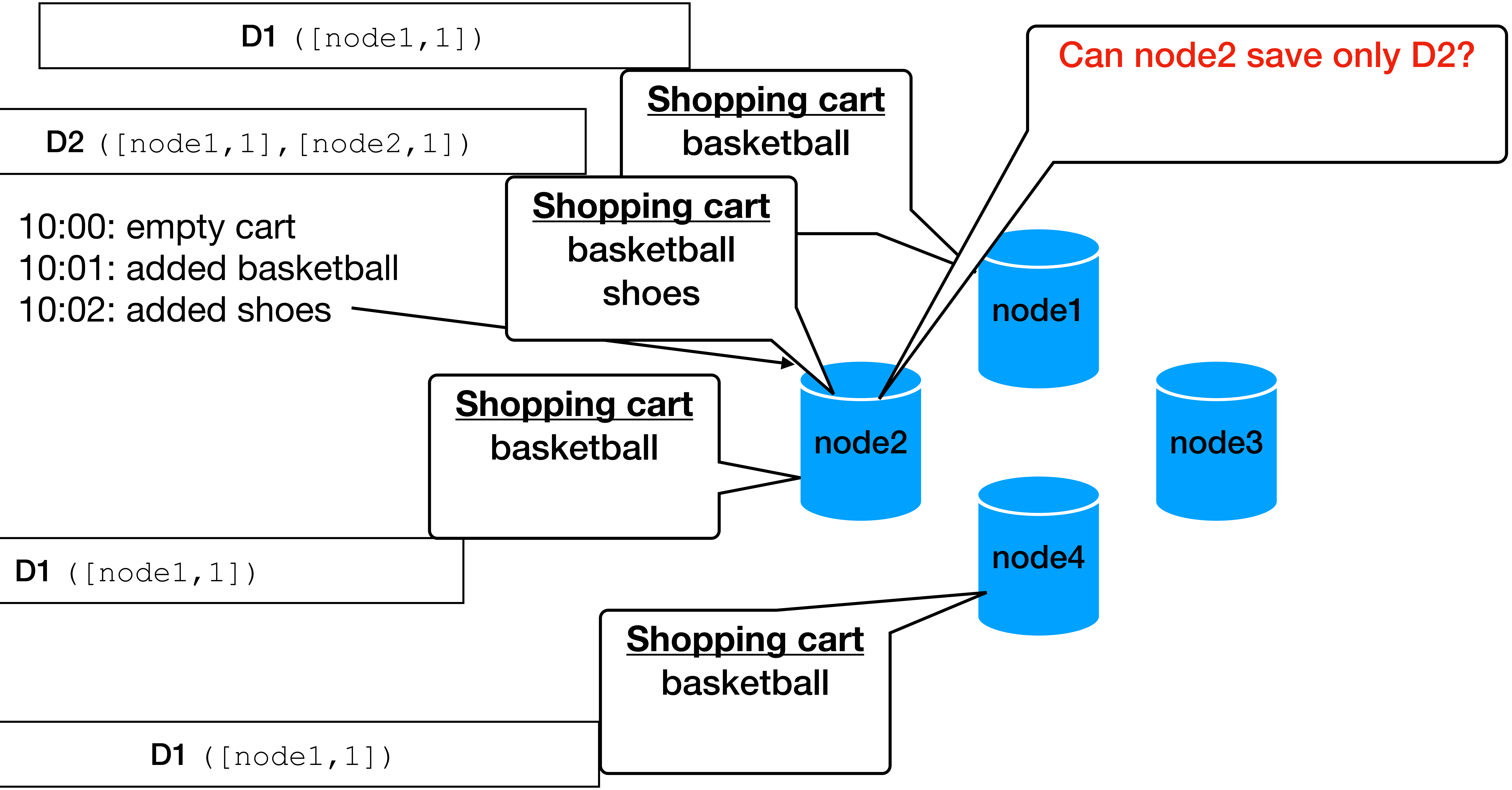
D1 `([node1,1])`

D1 `([node1,1])`

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes



D1 ([node1,1])

D2 ([node1,1],[node2,1])

D1 ([node1,1])

D1 ([node1,1])

Can node2 save only D2?

Data versioning (5) - motivation example

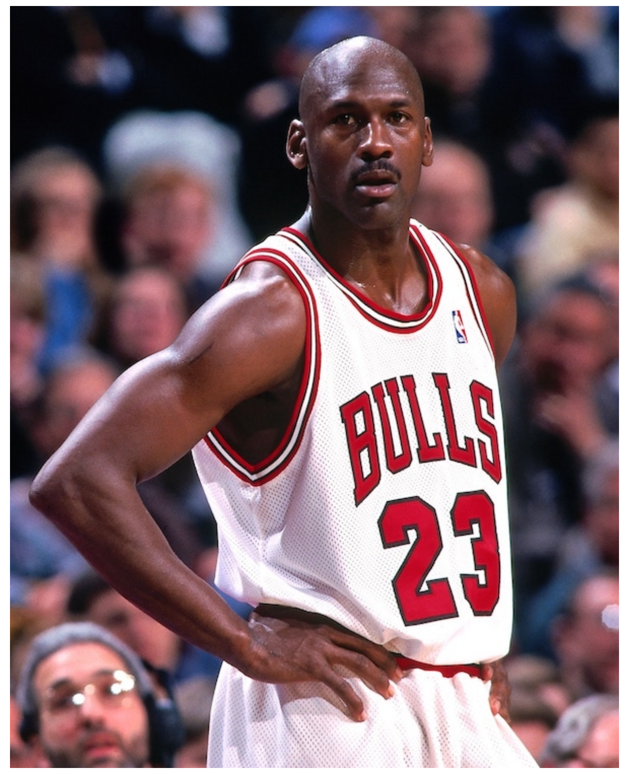
D1 `([node1,1])`

D2 `([node1,1],[node2,1])`

<u>Shopping cart</u>
basketball

Can node2 save only D2?

<u>Shopping cart</u>
basketball



10:00: empty cart
10:01: added basketball
10:02:

```
IF      all the counters on the first object's clocks <=
        all the counters on the second object
THEN
        first is ancestor of the second and can be forgotten
ELSE
        there is a conflict, the client should reconcile
```

D1 `([node1,1])`

node4

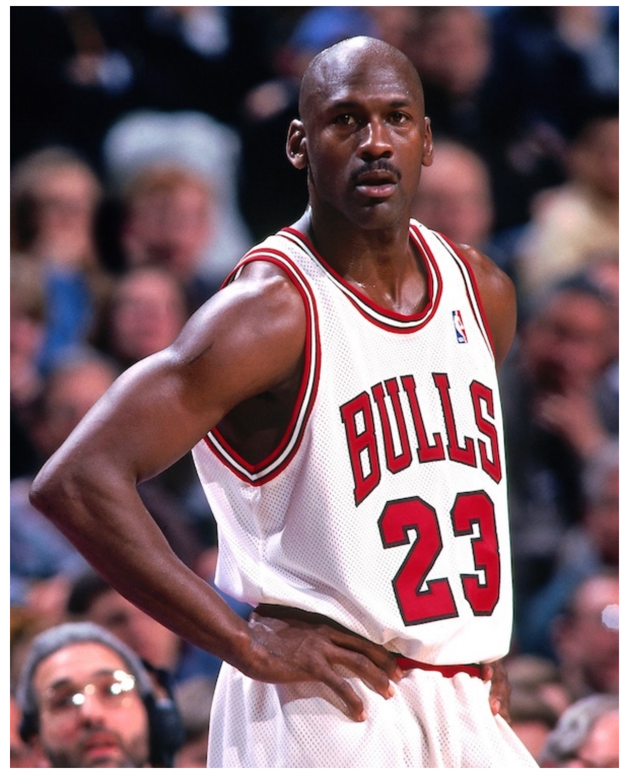
<u>Shopping cart</u>
basketball

D1 `([node1,1])`

Data versioning (5) - motivation example

D1 `([node1,1])`

D2 `([node1,1],[node2,1])`

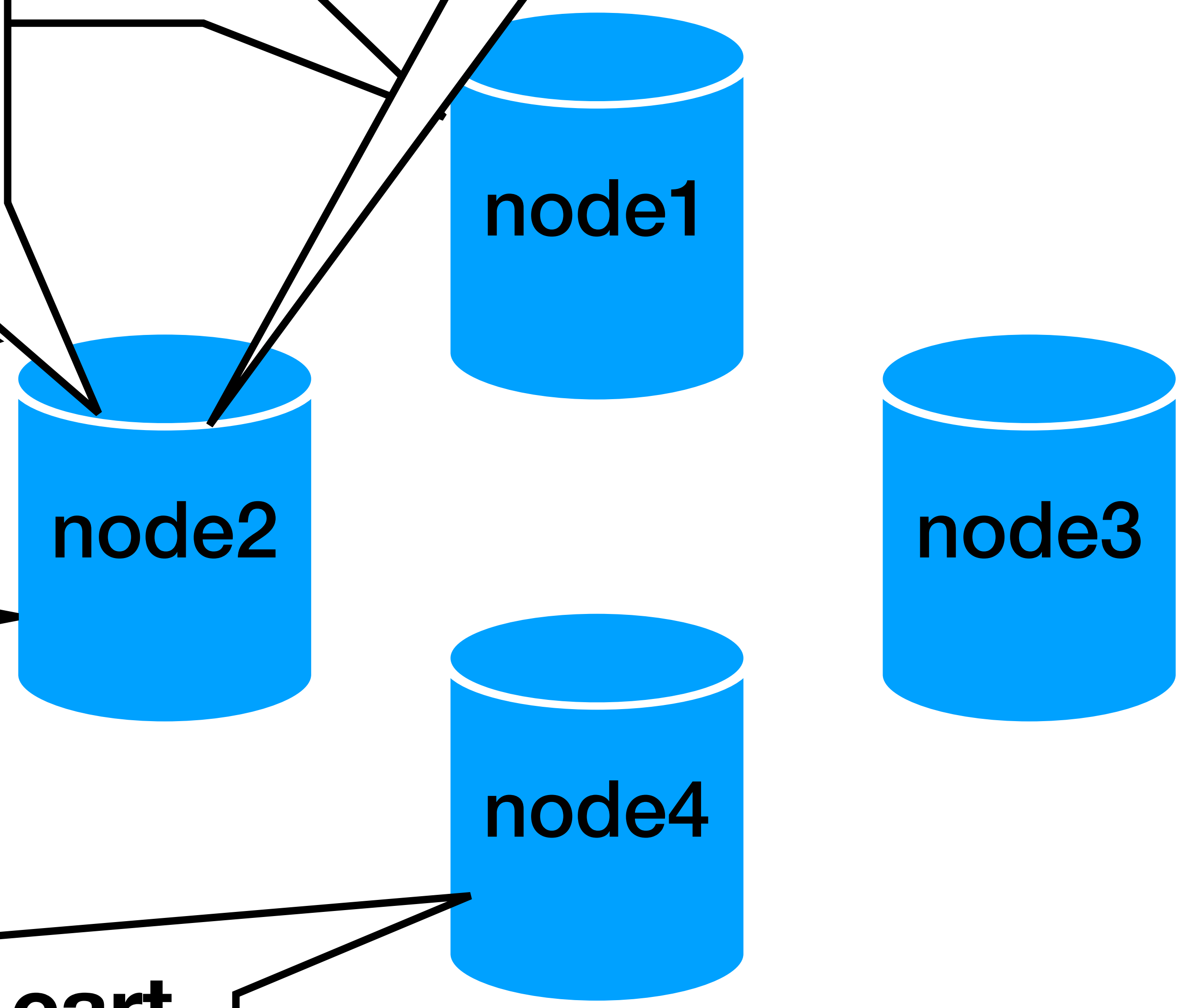


10:00: empty cart
10:01: added basketball
10:02: added shoes

Shopping cart
basketball

Shopping cart
basketball
shoes

Can node2 save only D2?
YES!



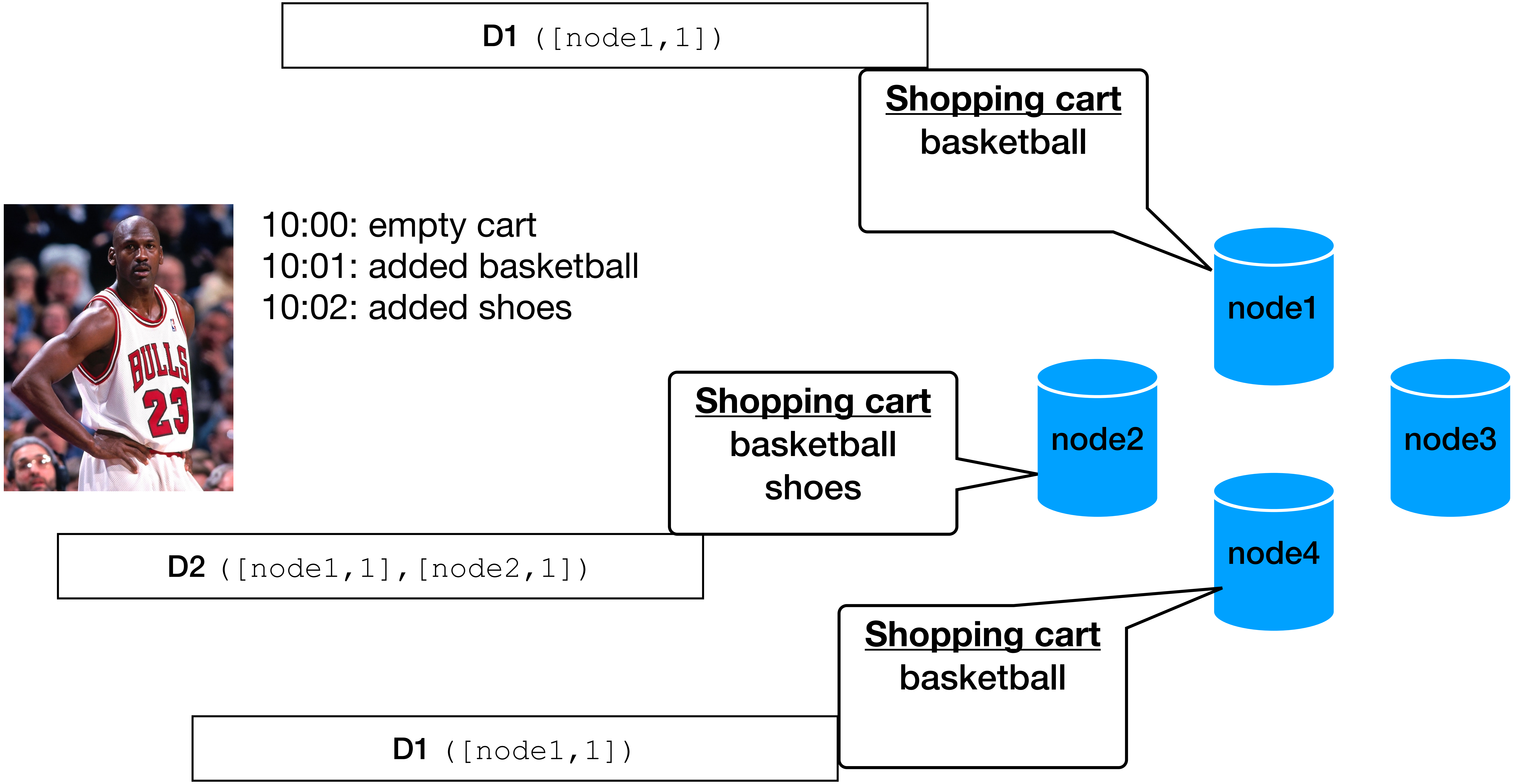
Shopping cart
basketball

D1 `([node1,1])`

Shopping cart
basketball

D1 `([node1,1])`

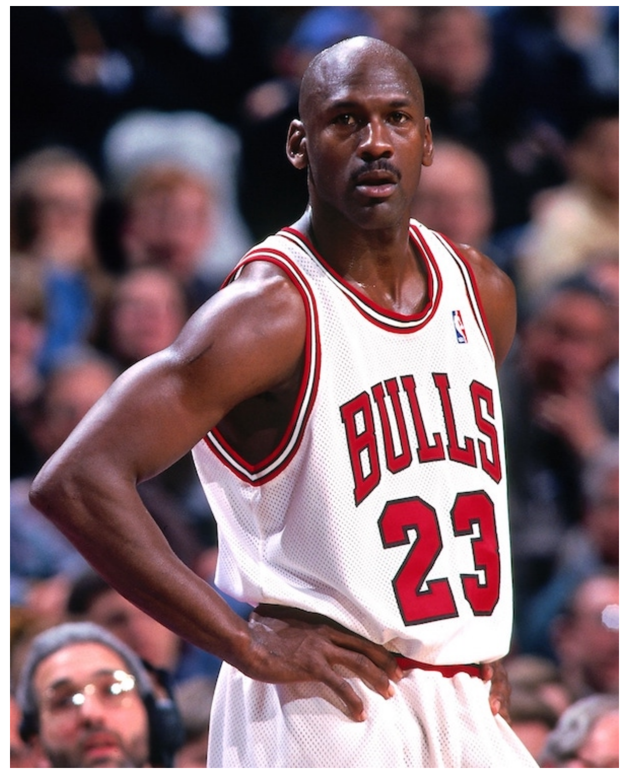
Data versioning (5) - motivation example



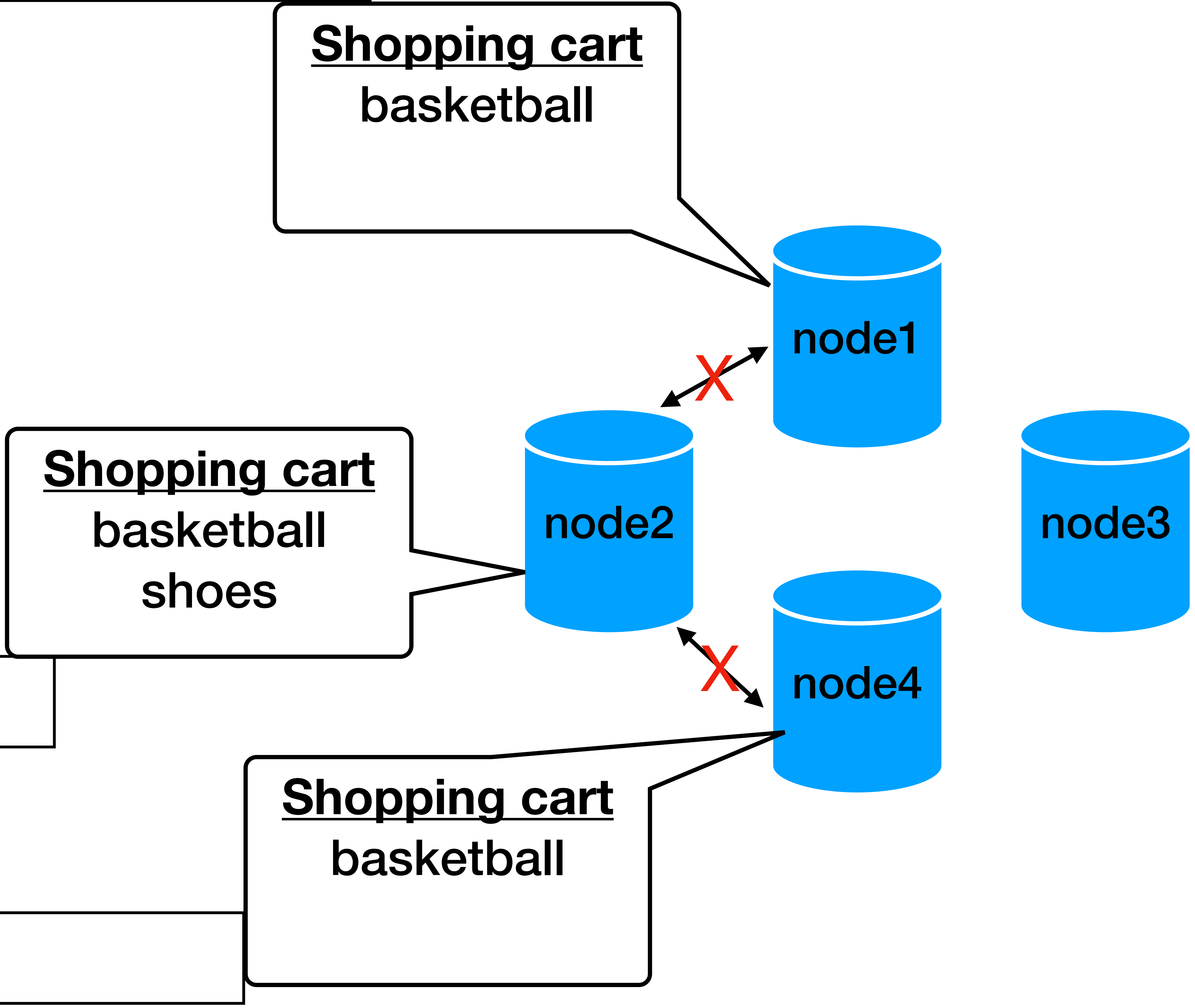
10:00: empty cart
10:01: added basketball
10:02: added shoes

Data versioning (5) - motivation example

D1 `([node1,1])`



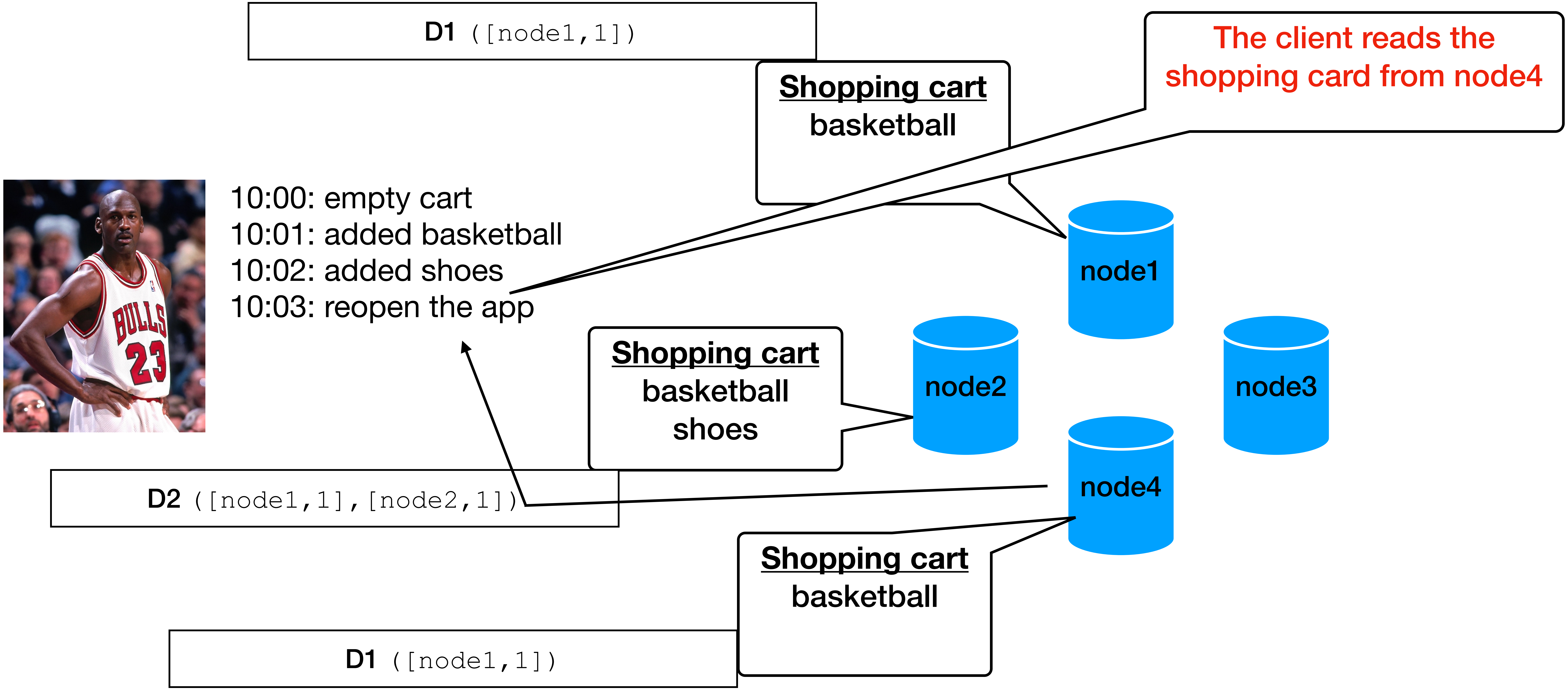
10:00: empty cart
10:01: added basketball
10:02: added shoes



D2 `([node1,1],[node2,1])`

D1 `([node1,1])`

Data versioning (5) - motivation example



D1 ([node1,1])

Shopping cart
basketball

The client reads the shopping card from node4

10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app

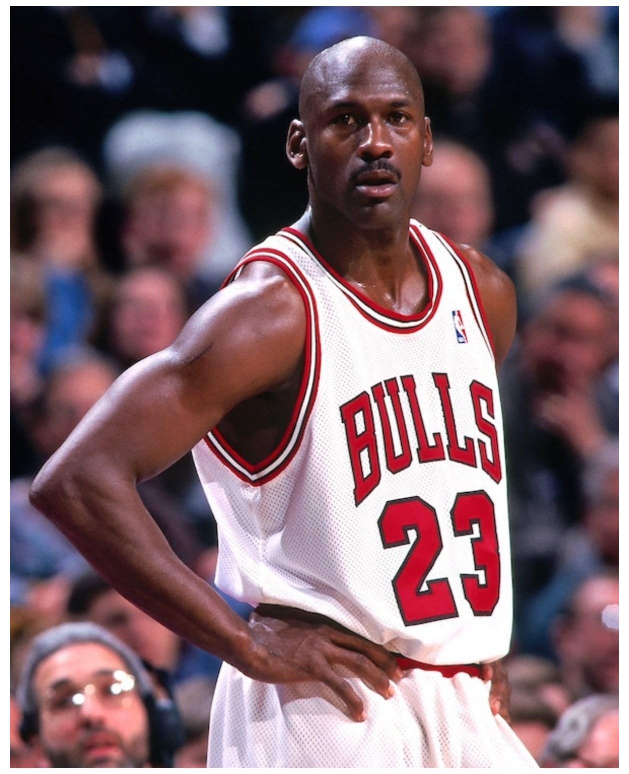
Shopping cart
basketball
shoes

D2 ([node1,1],[node2,1])

Shopping cart
basketball

D1 ([node1,1])

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

D1 `([node1,1])`

Shopping cart
basketball

node1

Shopping cart
basketball
shoes

node2

node3

D2 `([node1,1],[node2,1])`

node4

Shopping cart
basketball

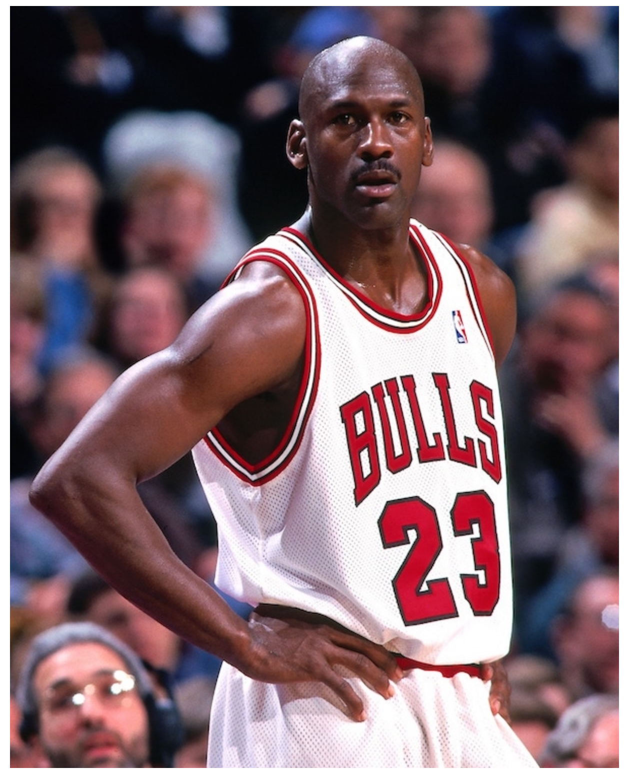
Shopping cart
basketball
ps5

D1 `([node1,1])`

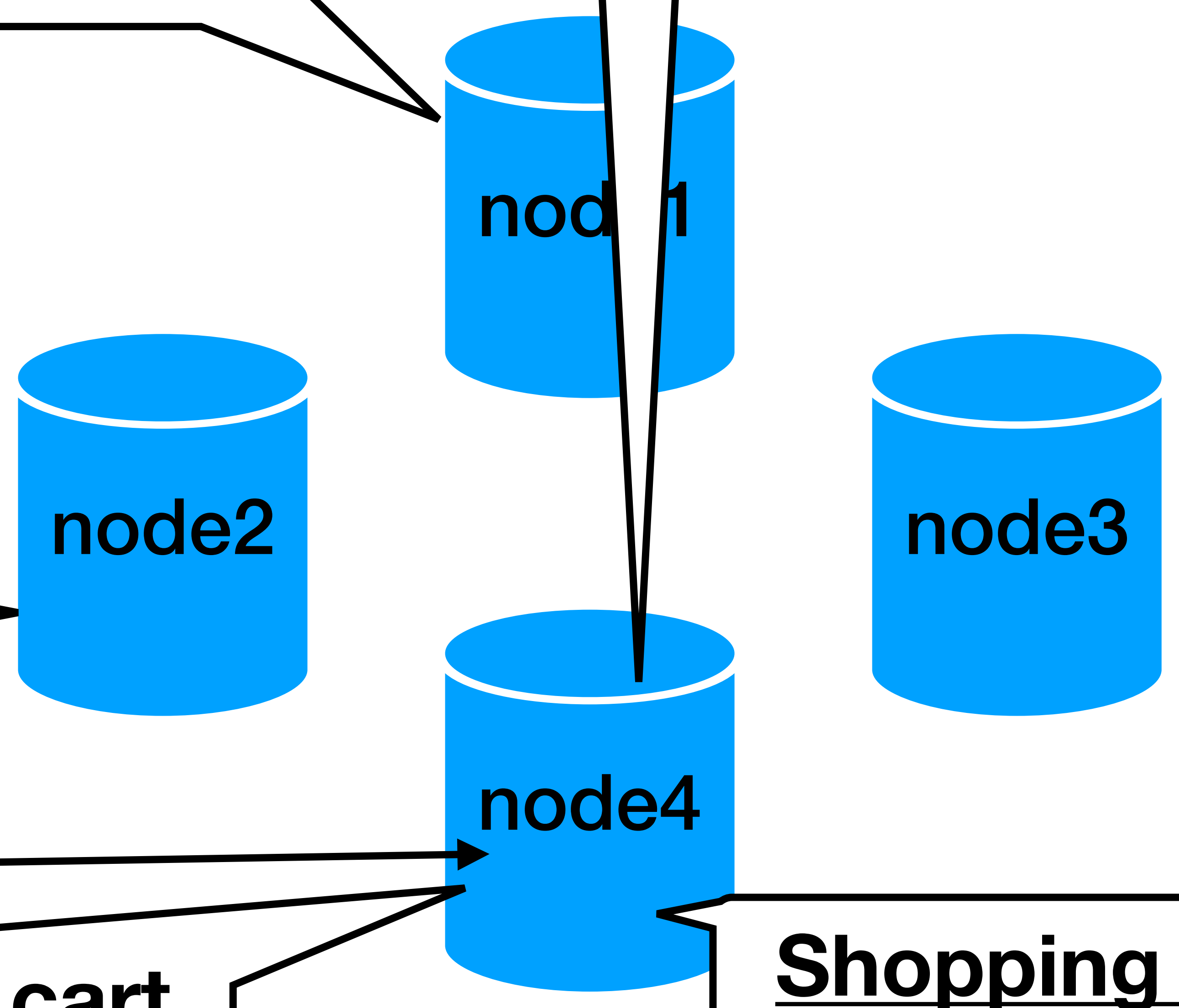
D3 `([node1,1],node4,1])`

Data versioning (5) - motivation example

D1 ([node1,1])



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



Can node4 save only D3?

Shopping cart
basketball

Shopping cart
basketball
shoes

D2 ([node1,1],[node2,1])

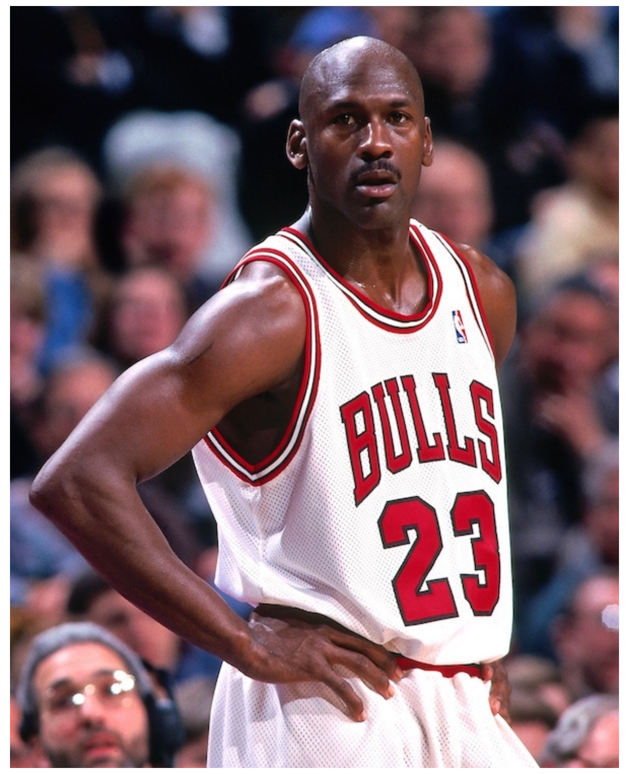
Shopping cart
basketball

Shopping cart
basketball
ps5

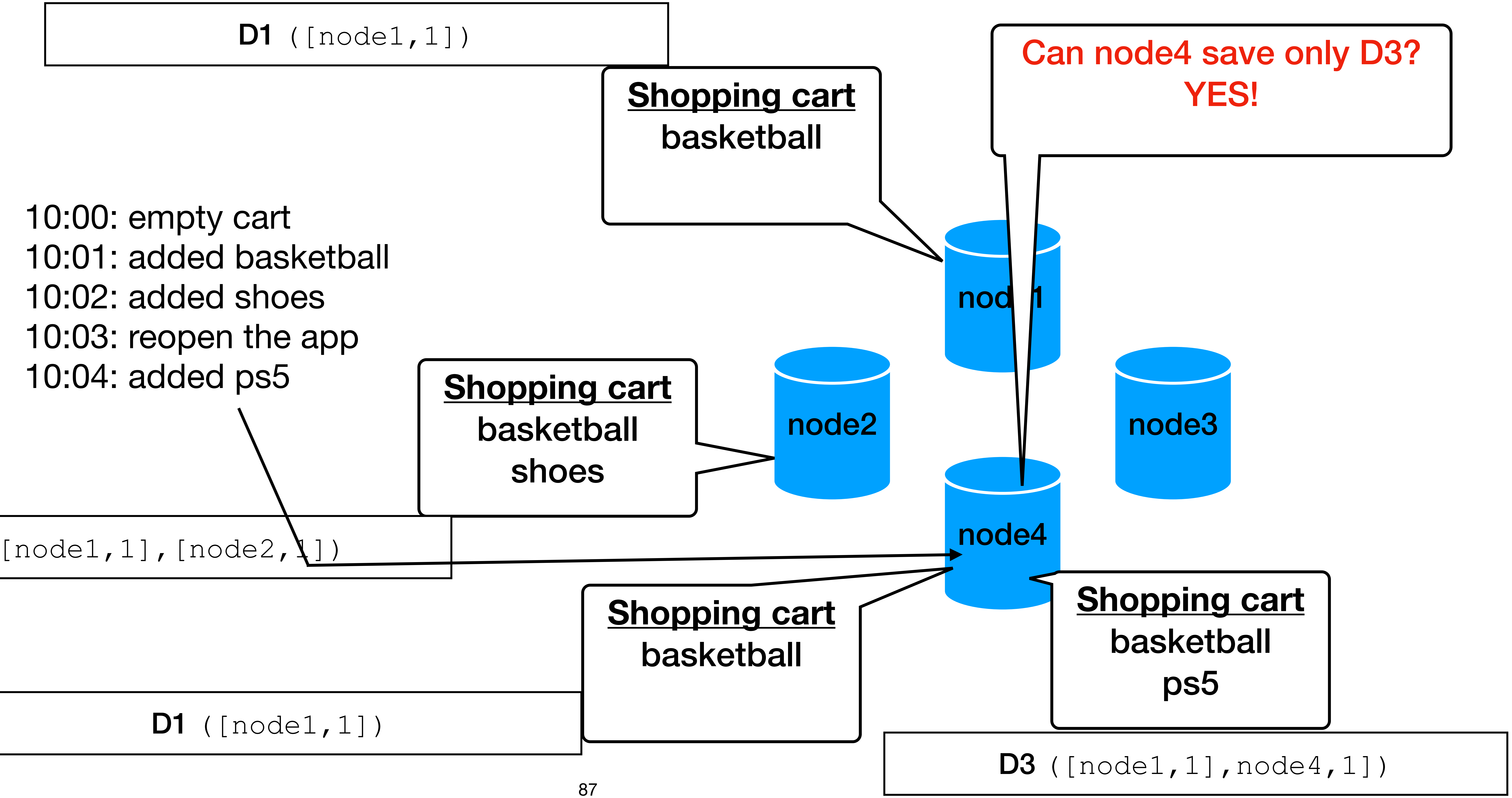
D1 ([node1,1])

D3 ([node1,1],node4,1])

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



D1 `([node1,1])`

Shopping cart
basketball

Can node4 save only D3?
YES!

Shopping cart
basketball
shoes

D2 `([node1,1],[node2,1])`

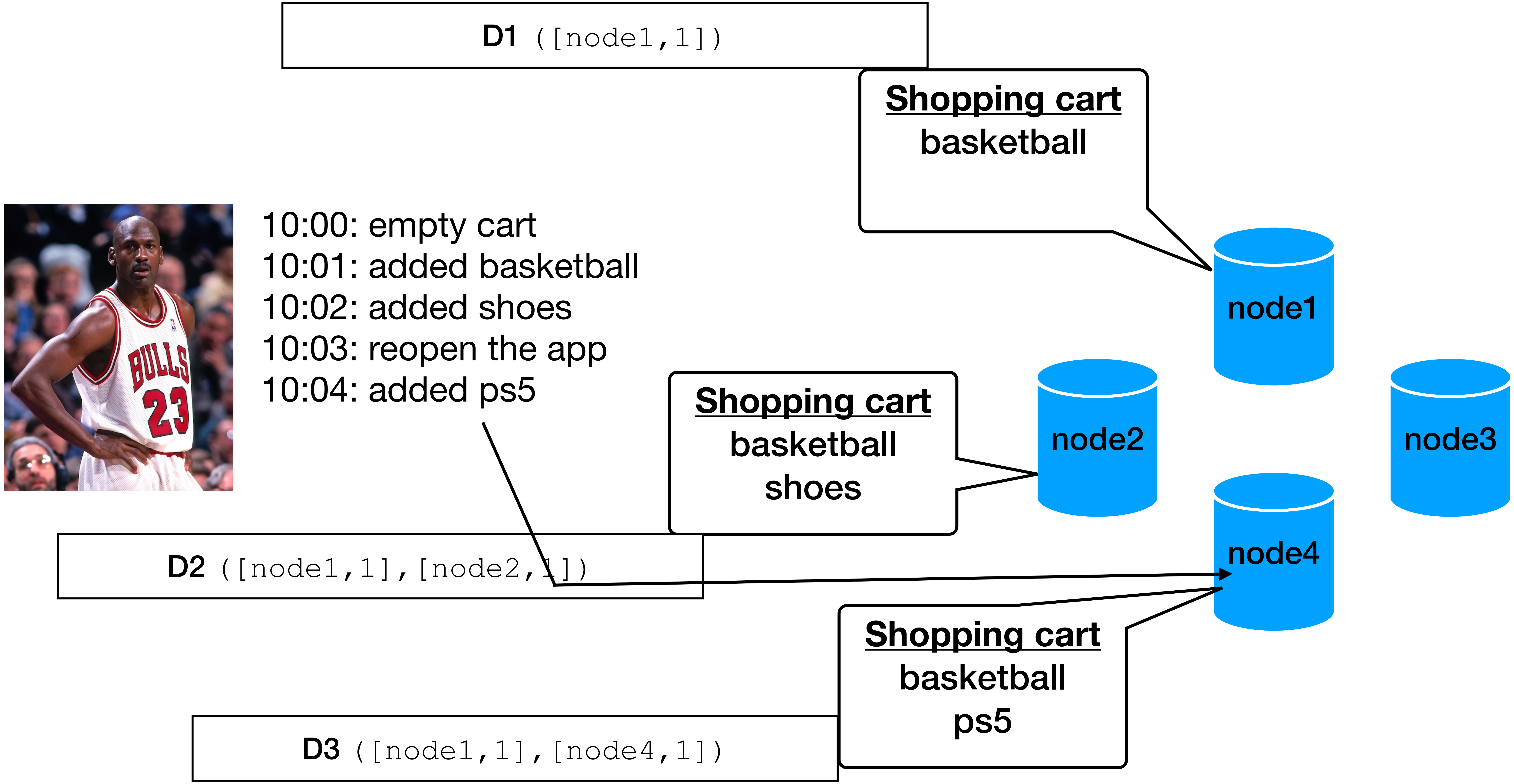
Shopping cart
basketball

Shopping cart
basketball
ps5

D1 `([node1,1])`

D3 `([node1,1],node4,1])`

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

D1 `([node1,1])`

Shopping cart
basketball

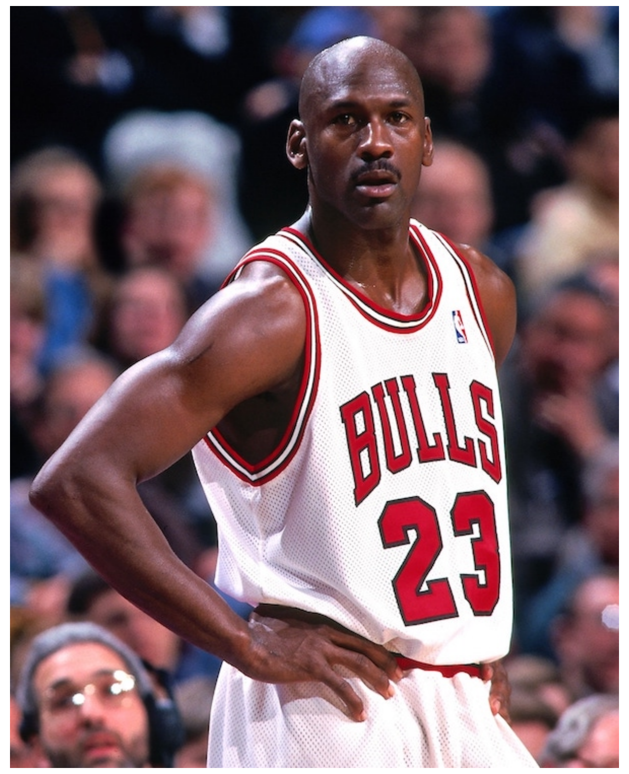
D2 `([node1,1],[node2,1])`

Shopping cart
basketball
shoes

D3 `([node1,1],[node4,1])`

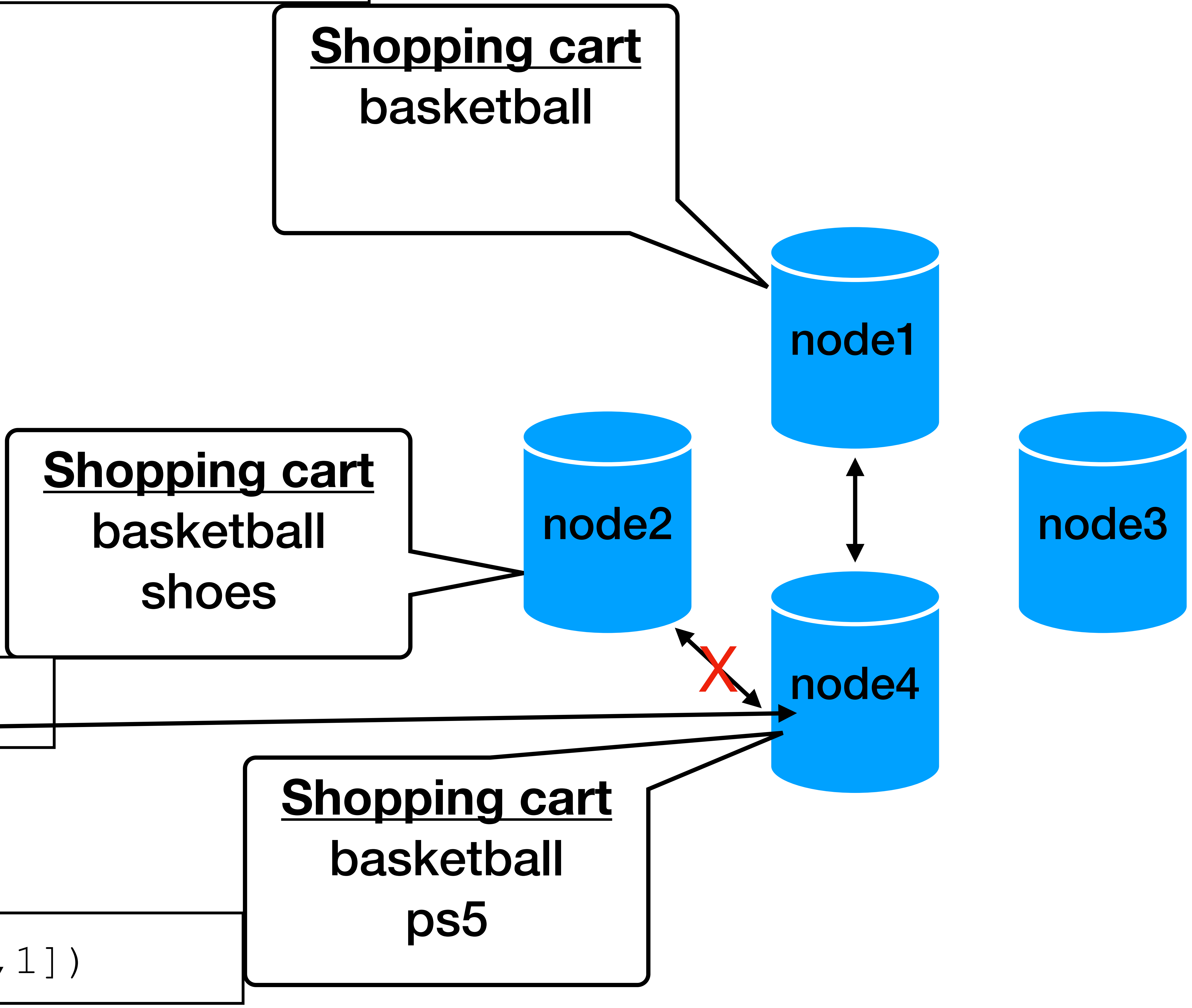
Shopping cart
basketball
ps5

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

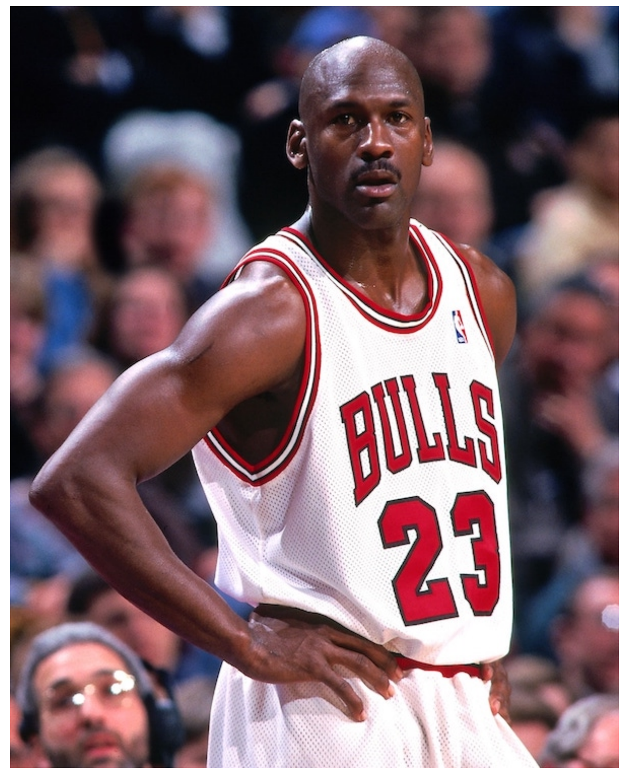
D1 `([node1,1])`



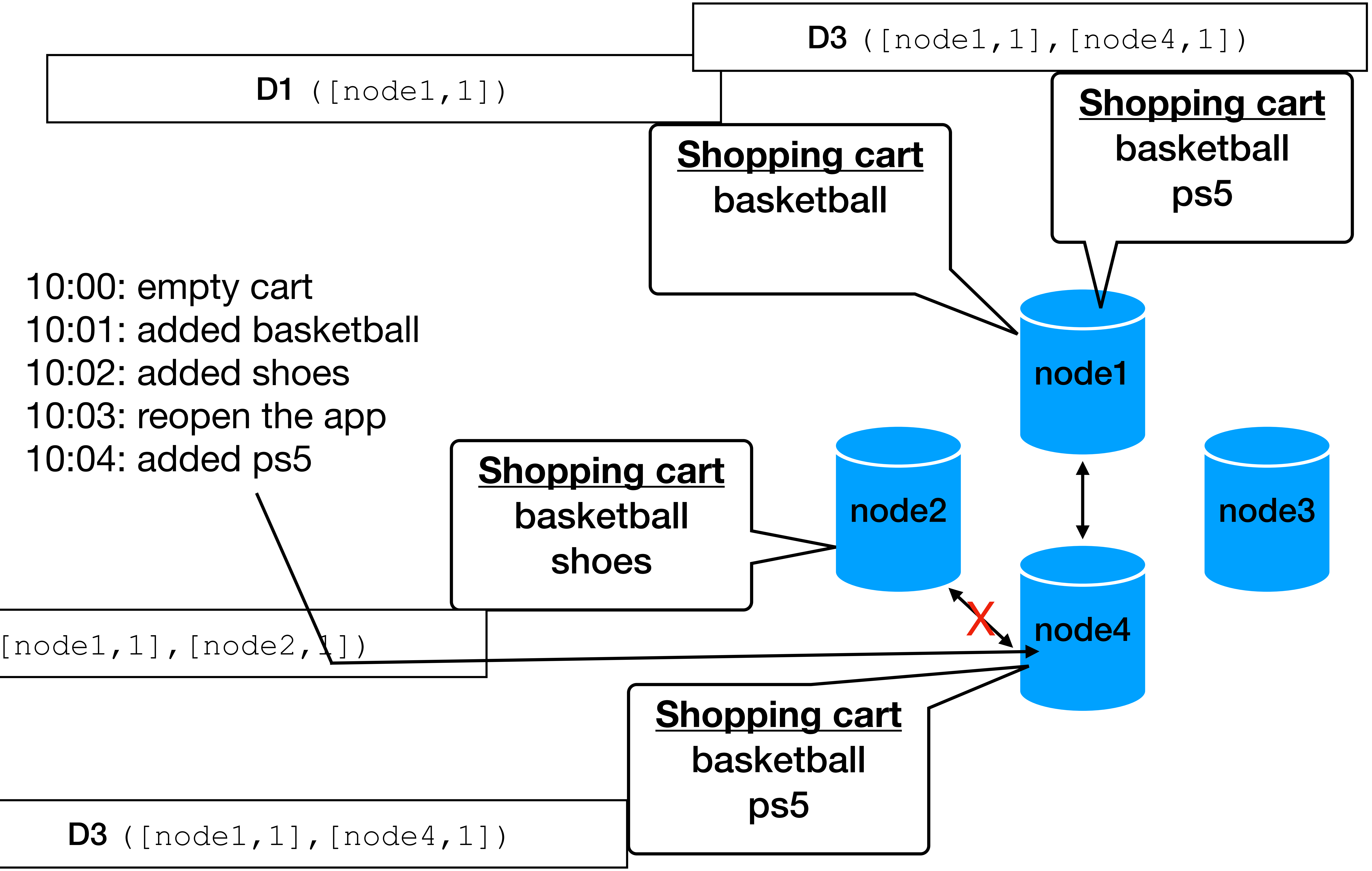
D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



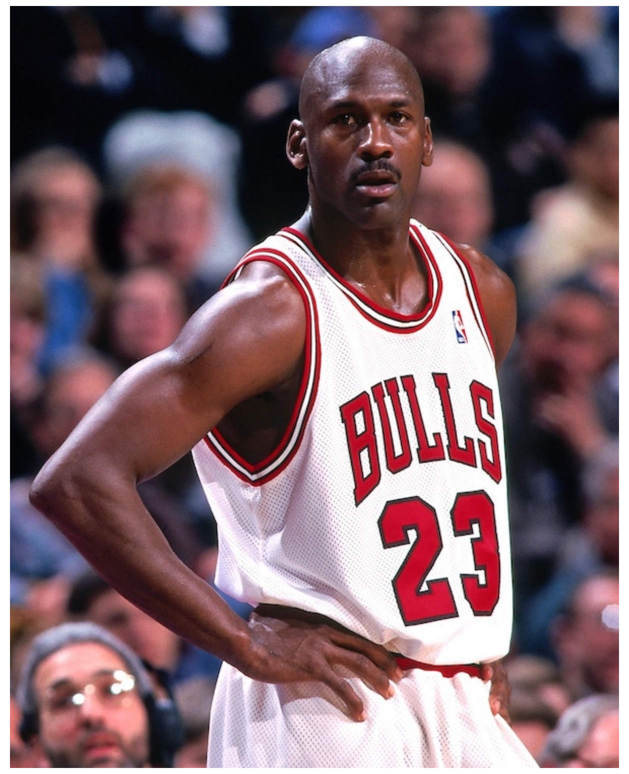
D1 `([node1,1])`

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`

D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example



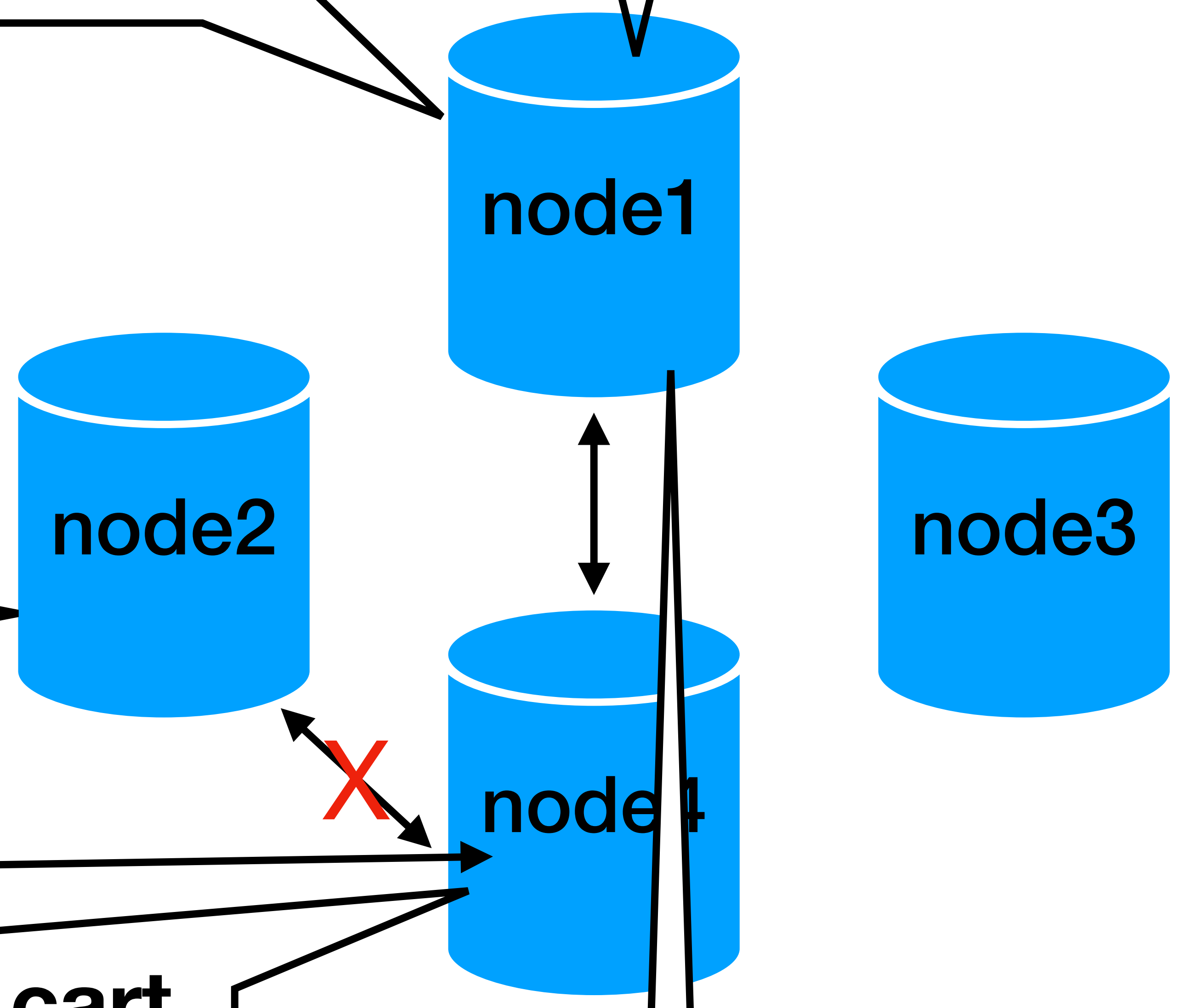
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

D1 `([node1,1])`

D3 `([node1,1],[node4,1])`

Shopping cart
basketball

Shopping cart
basketball
ps5



Shopping cart
basketball
shoes

D2 `([node1,1],[node2,1])`

Shopping cart
basketball
ps5

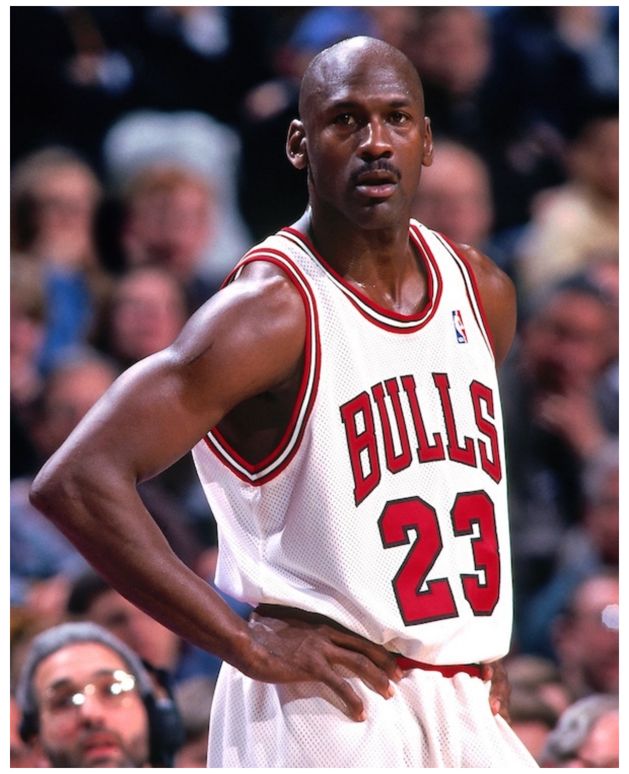
D3 `([node1,1],[node4,1])`

Can node1 save only D3?

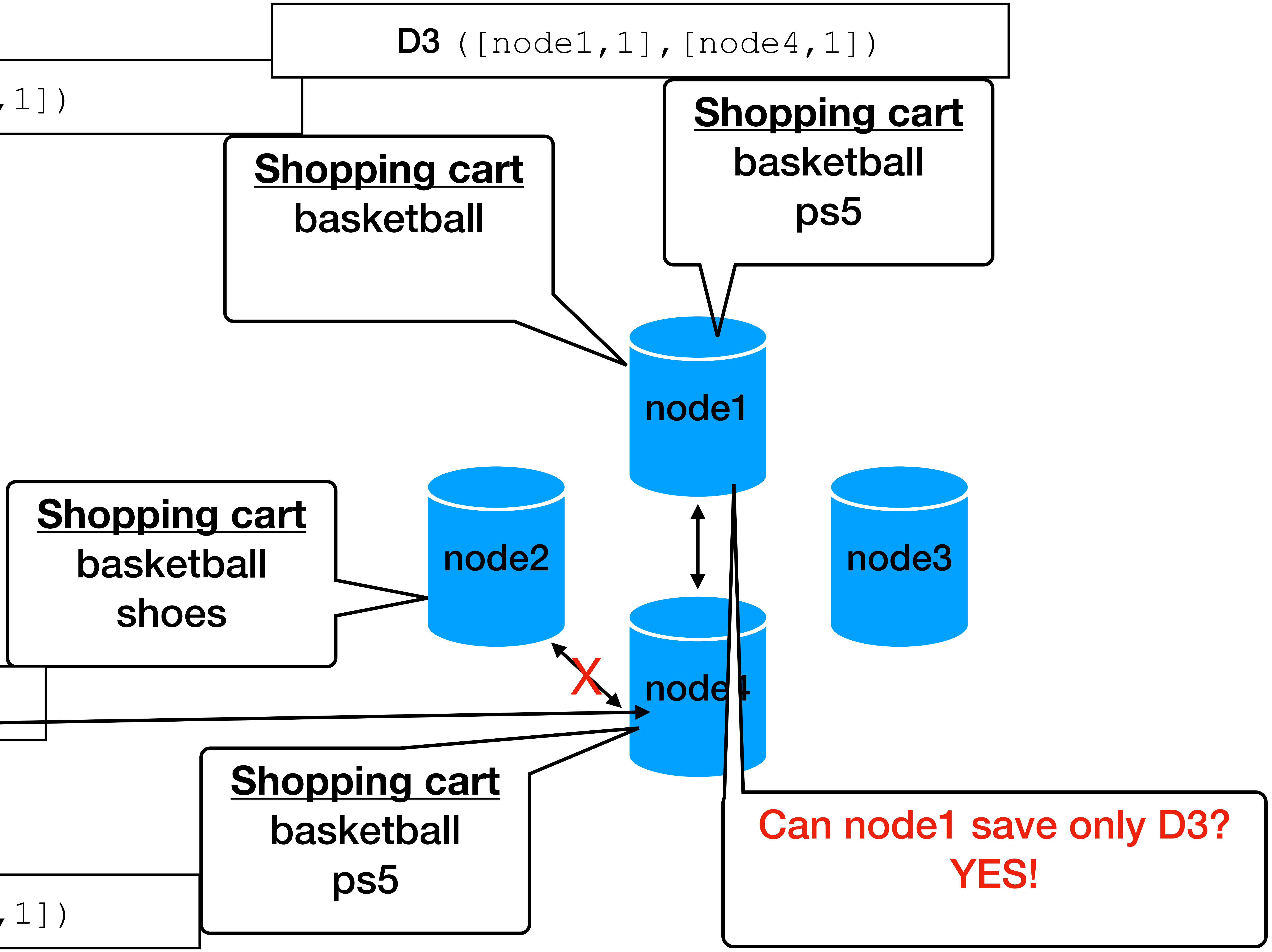
Data versioning (5) - motivation example

D1 `([node1,1])`

D3 `([node1,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



Shopping cart
basketball

Shopping cart
basketball
ps5

node1

node2

node3

node4

Shopping cart
basketball
shoes

D2 `([node1,1],[node2,1])`

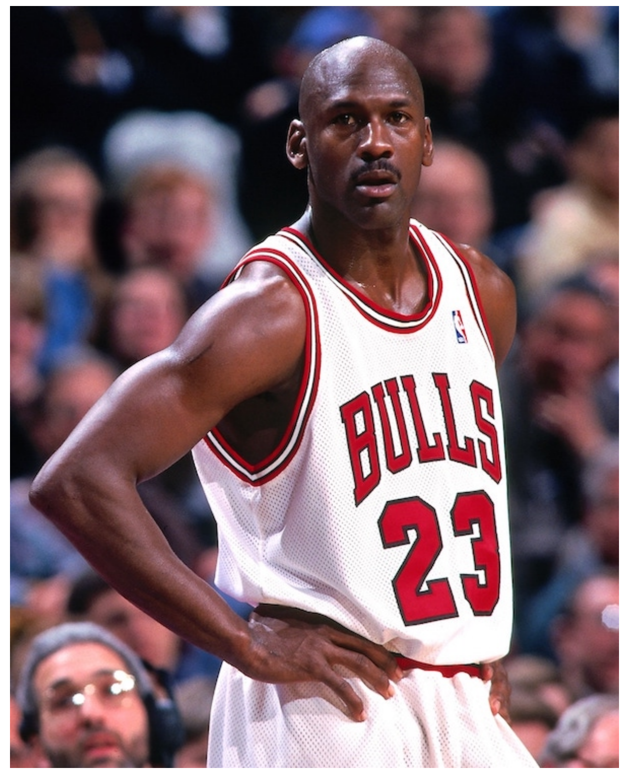
Shopping cart
basketball
ps5

D3 `([node1,1],[node4,1])`

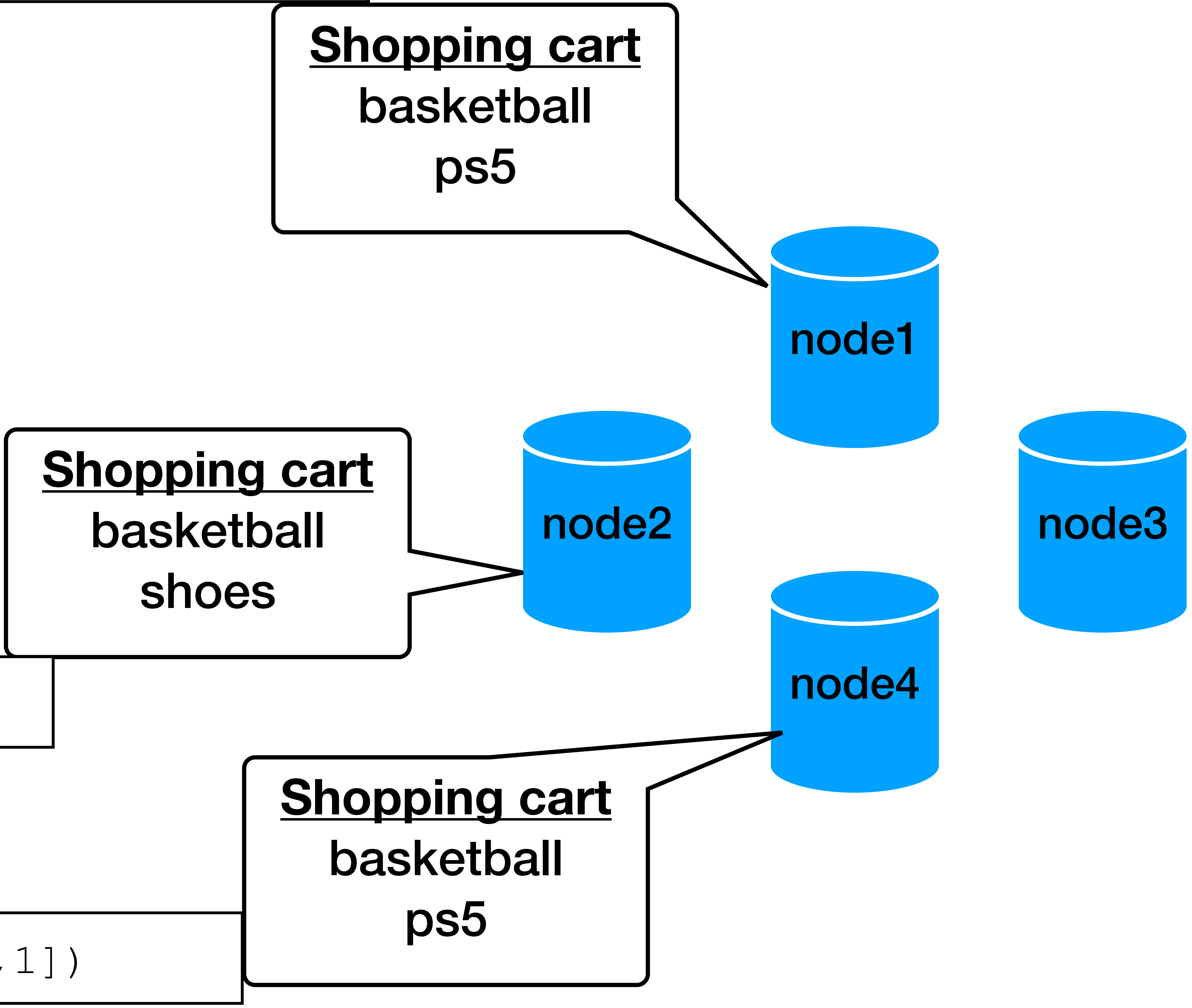
Can node1 save only D3?
YES!

Data versioning (5) - motivation example

D3 `([node1,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



Shopping cart
basketball
ps5

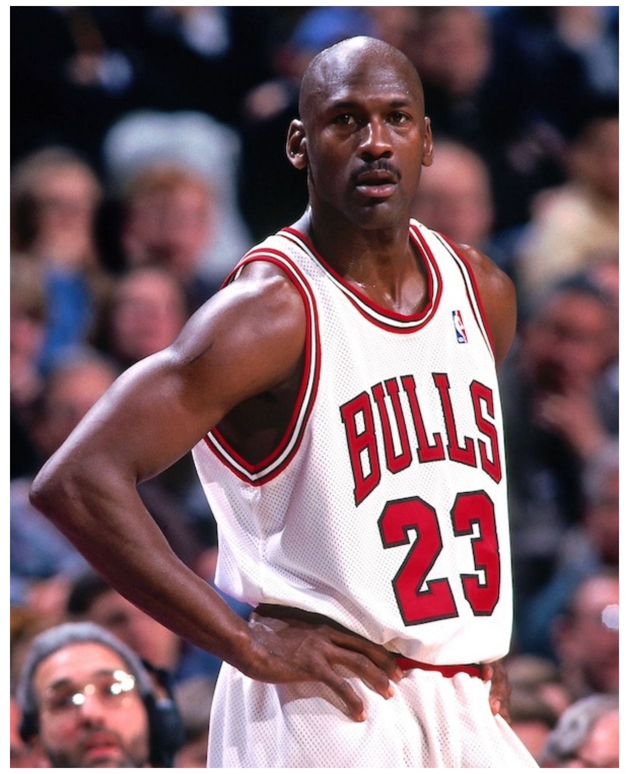
Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

D2 `([node1,1],[node2,1])`

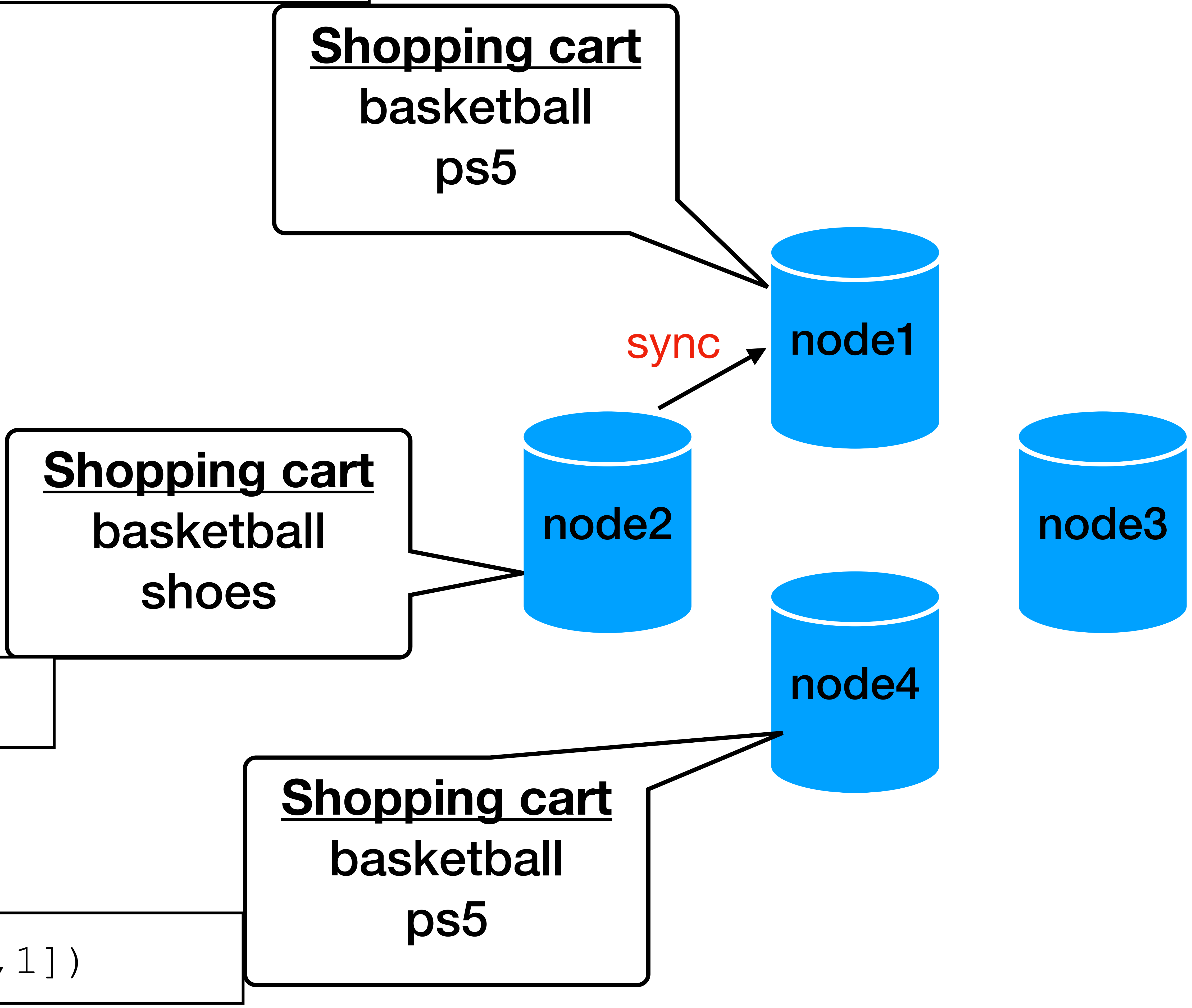
D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

D3 `([node1,1],[node4,1])`



Shopping cart
basketball
ps5

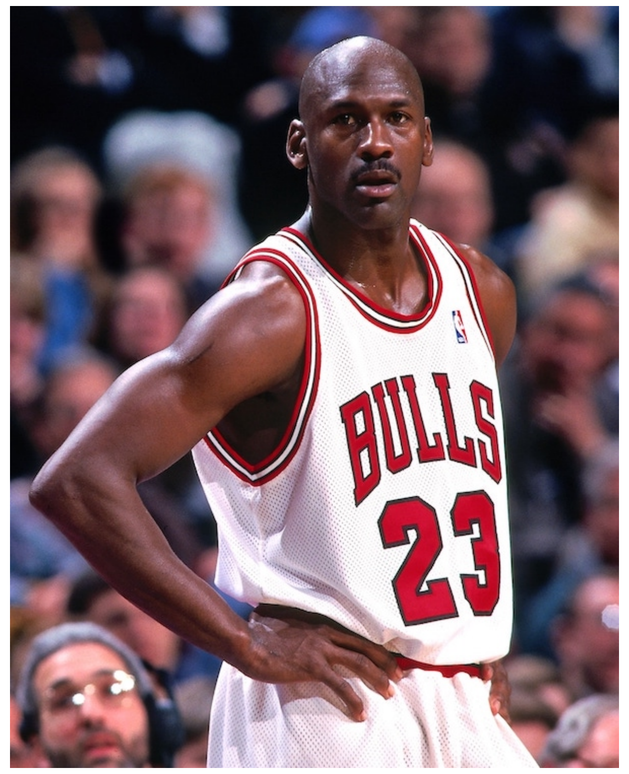
Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

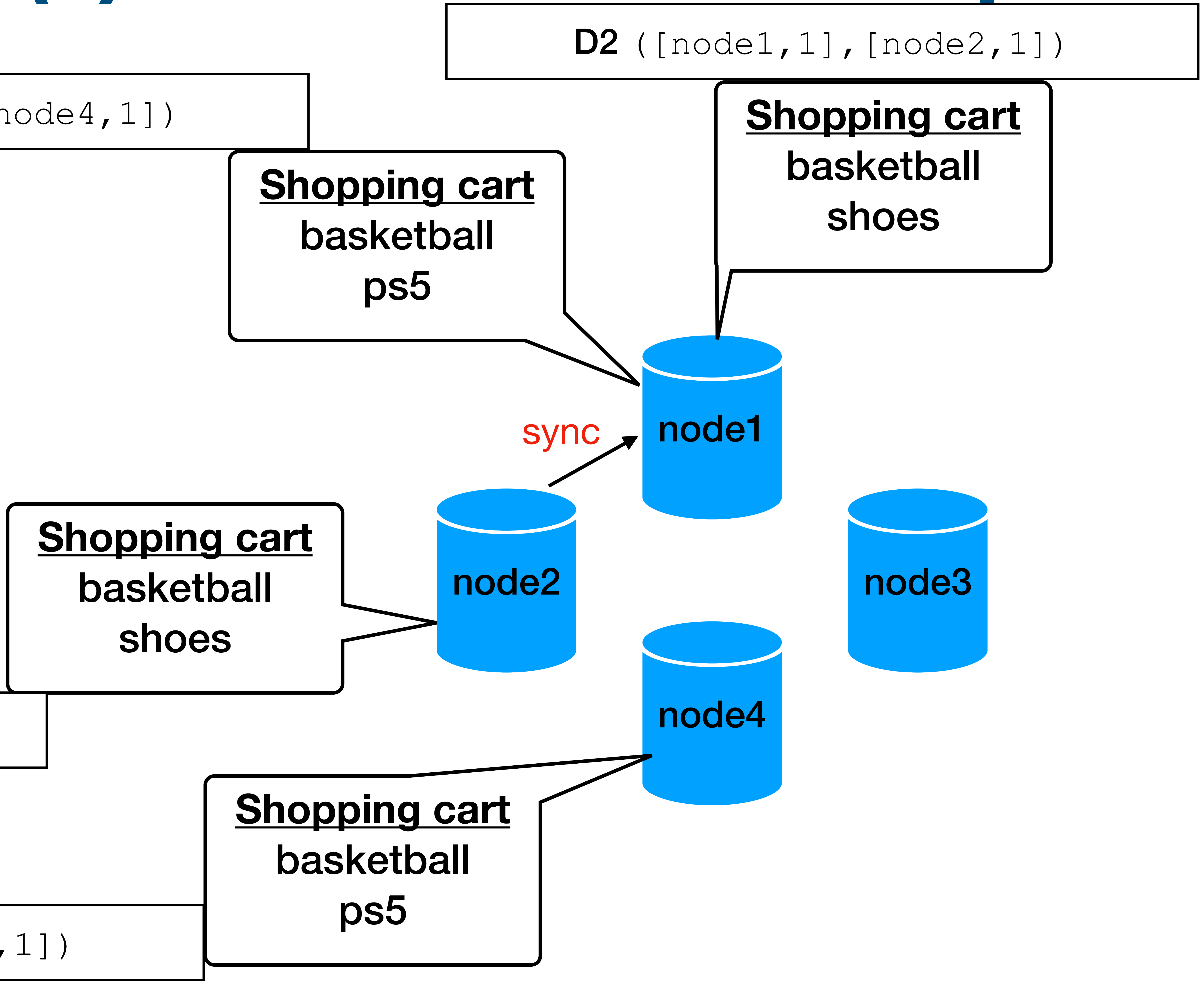
D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



D2 ([node1,1],[node2,1])

Shopping cart
basketball
shoes

D3 ([node1,1],[node4,1])

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

D2 ([node1,1],[node2,1])

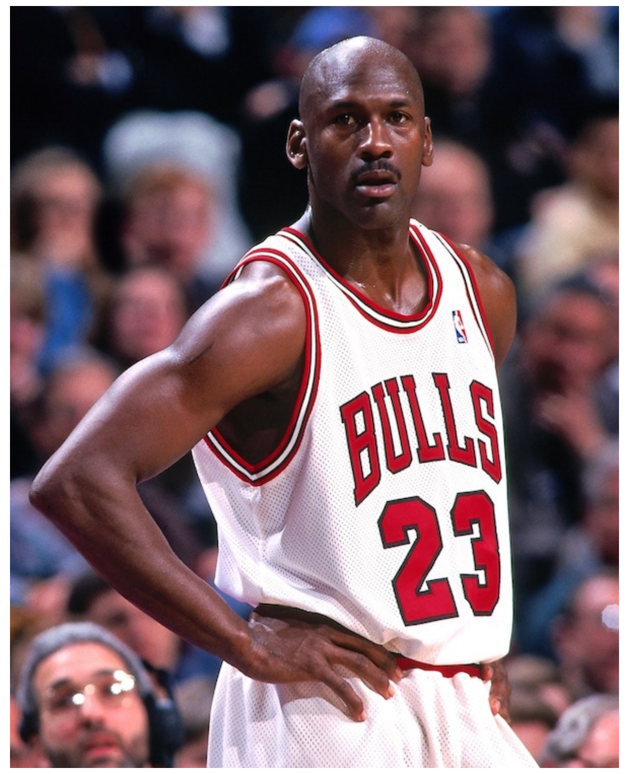
Shopping cart
basketball
ps5

D3 ([node1,1],[node4,1])

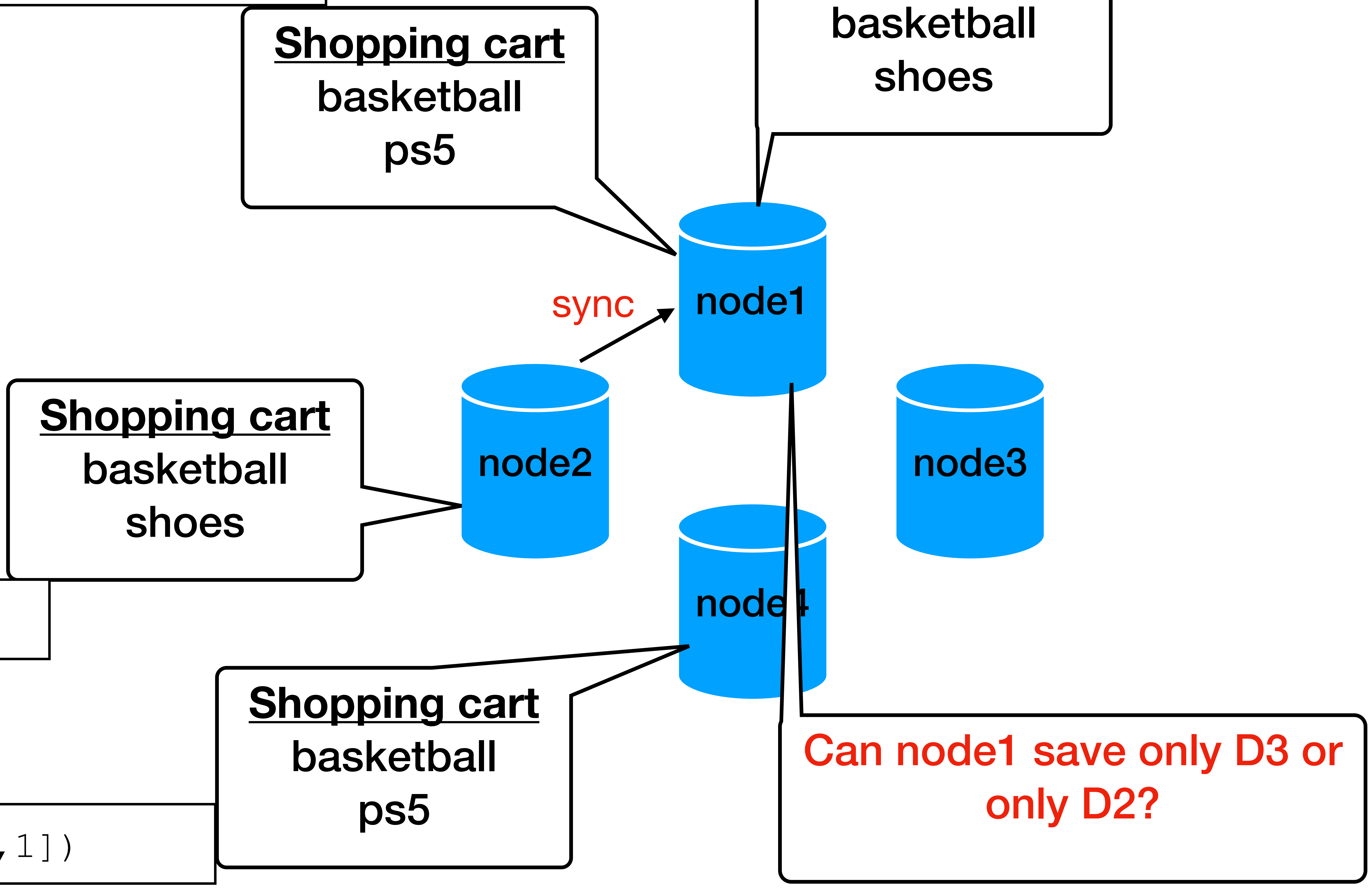
Data versioning (5) - motivation example

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

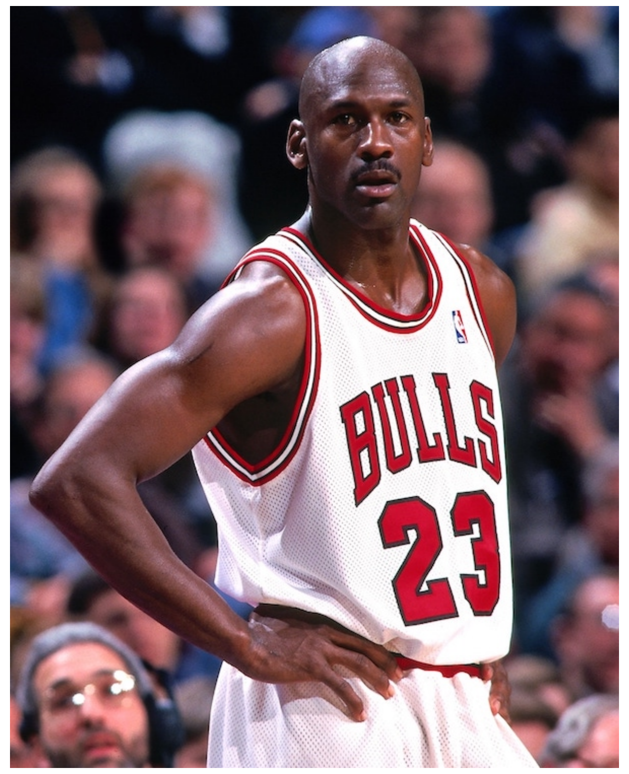


D2 `([node1,1],[node2,1])`

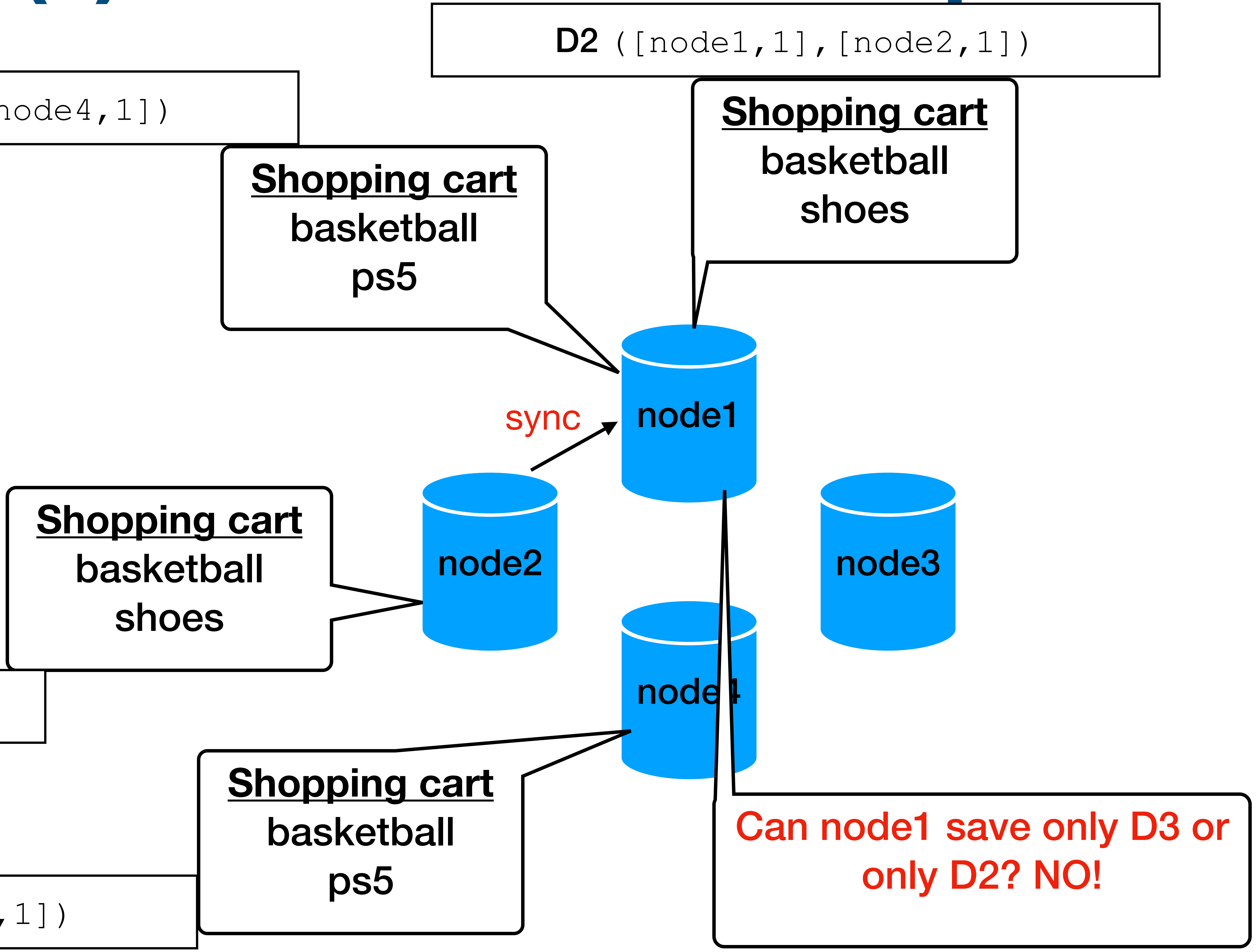
D3 `([node1,1],[node4,1])`

Can node1 save only D3 or only D2?

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5



D2 `([node1,1],[node2,1])`

Shopping cart
basketball
shoes

D3 `([node1,1],[node4,1])`

Shopping cart
basketball
ps5

sync

node1

node2

node3

node4

Shopping cart
basketball
shoes

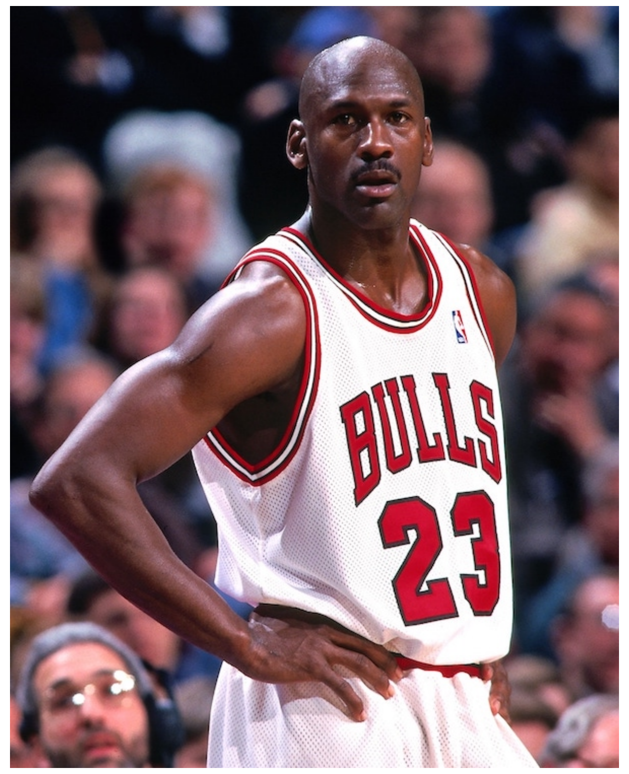
D2 `([node1,1],[node2,1])`

Shopping cart
basketball
ps5

D3 `([node1,1],[node4,1])`

Can node1 save only D3 or only D2? NO!

Data versioning (5) - motivation example

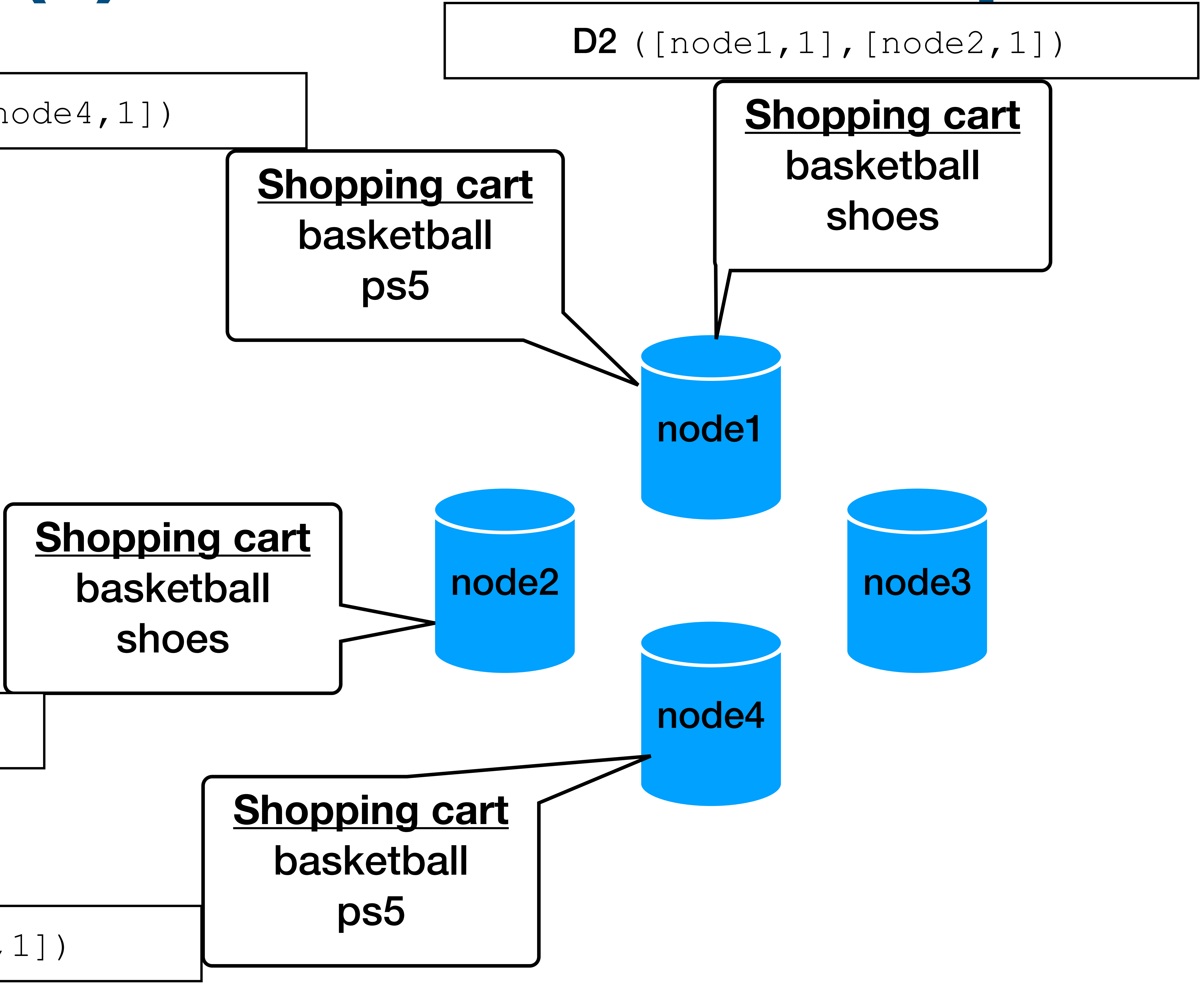


10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5

D3 ([node1,1],[node4,1])

D2 ([node1,1],[node2,1])

D3 ([node1,1],[node4,1])



D2 ([node1,1],[node2,1])

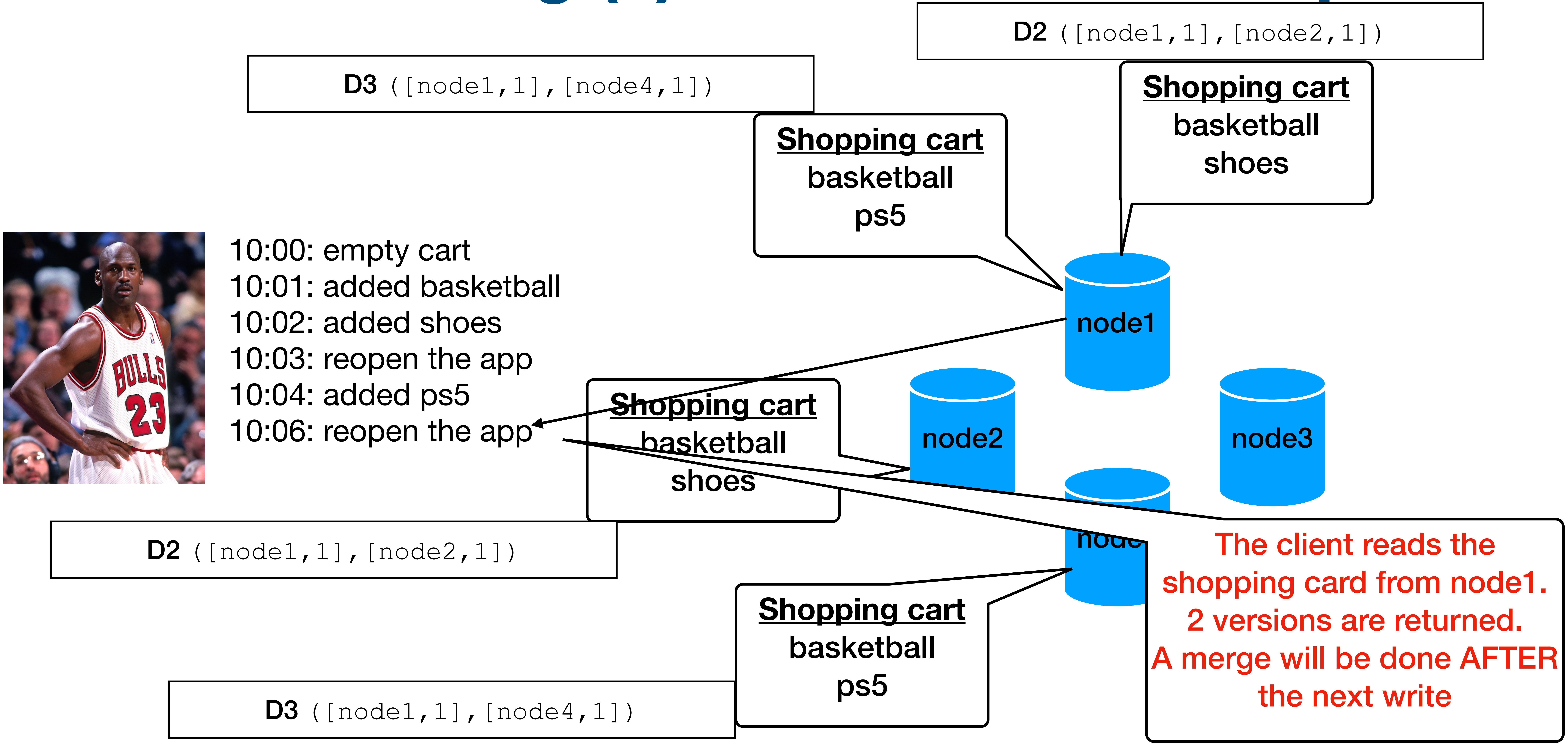
Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Data versioning (5) - motivation example



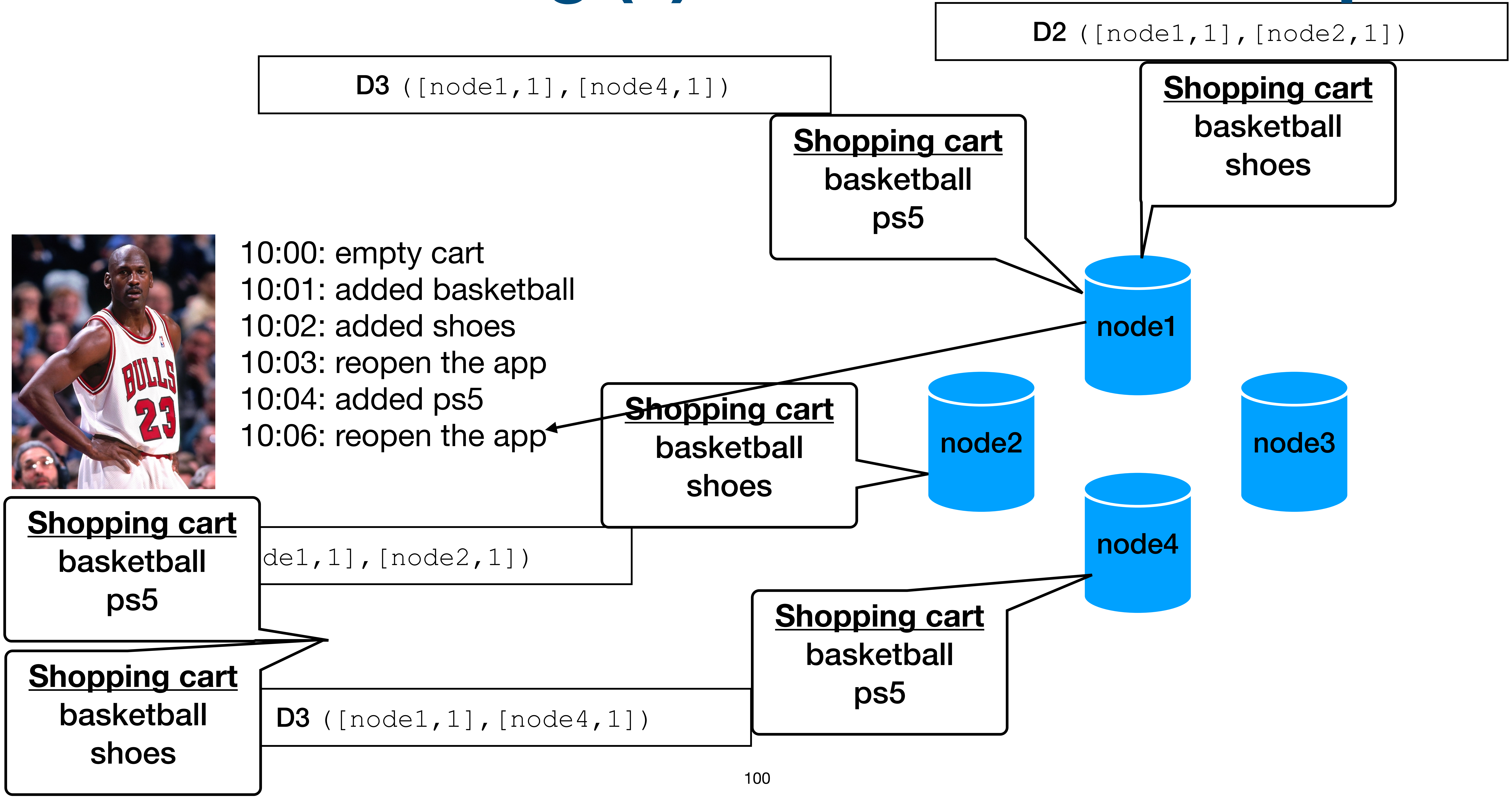
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app

D2 ([node1,1],[node2,1])
D3 ([node1,1],[node4,1])
Shopping cart basketball ps5
Shopping cart basketball shoes
node1
node2
node3
node
Shopping cart basketball shoes
D2 ([node1,1],[node2,1])
Shopping cart basketball ps5
D3 ([node1,1],[node4,1])

The client reads the shopping card from node1. 2 versions are returned. A merge will be done AFTER the next write

Data versioning (5) - motivation example

10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app



D2 ([node1,1],[node2,1])

Shopping cart
basketball
shoes

D3 ([node1,1],[node4,1])

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

de1,1],[node2,1])

Shopping cart
basketball
shoes

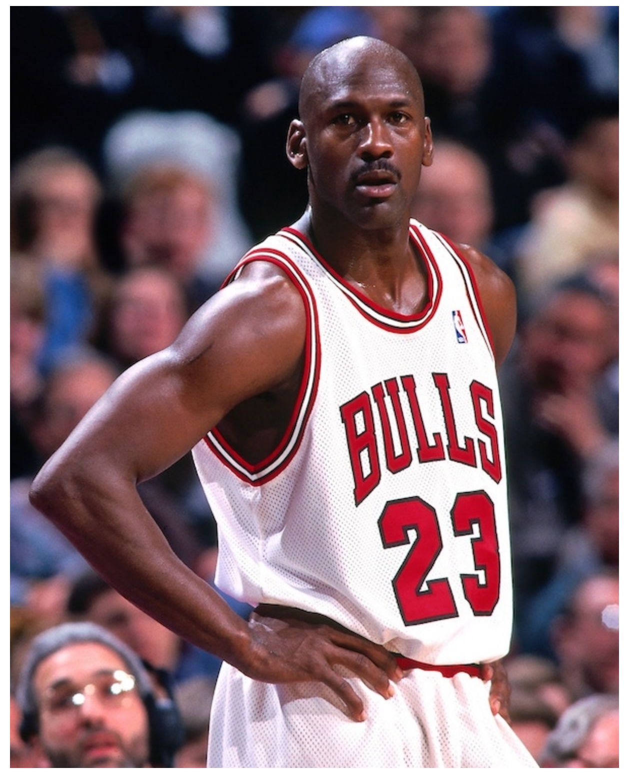
D3 ([node1,1],[node4,1])

Shopping cart
basketball
ps5

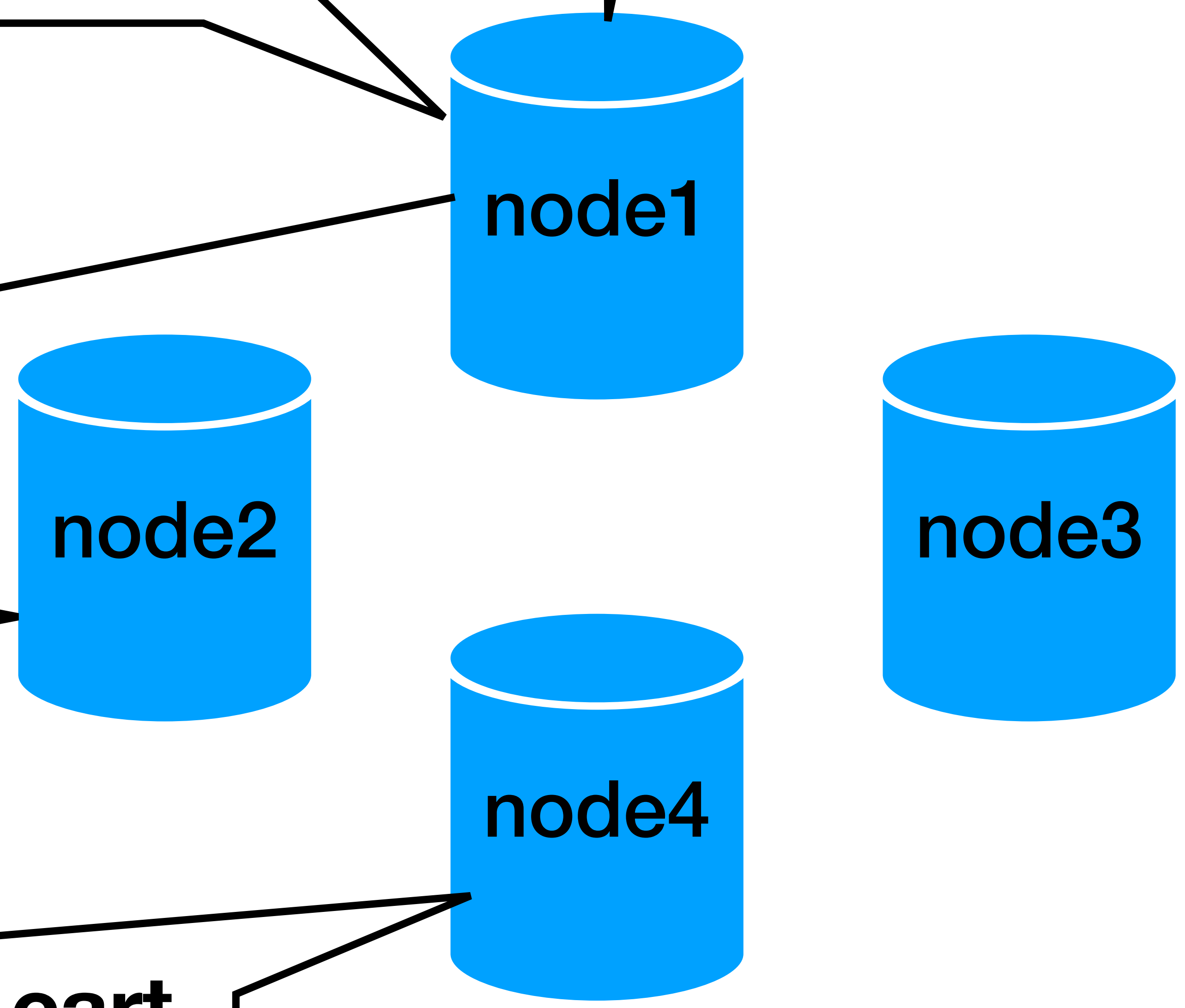
Data versioning (5) - motivation example

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app



Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

`de1,1`

Shopping cart
basketball
shoes

Shopping cart
basketball
shoes
ps5

D3 `(` `,1])`

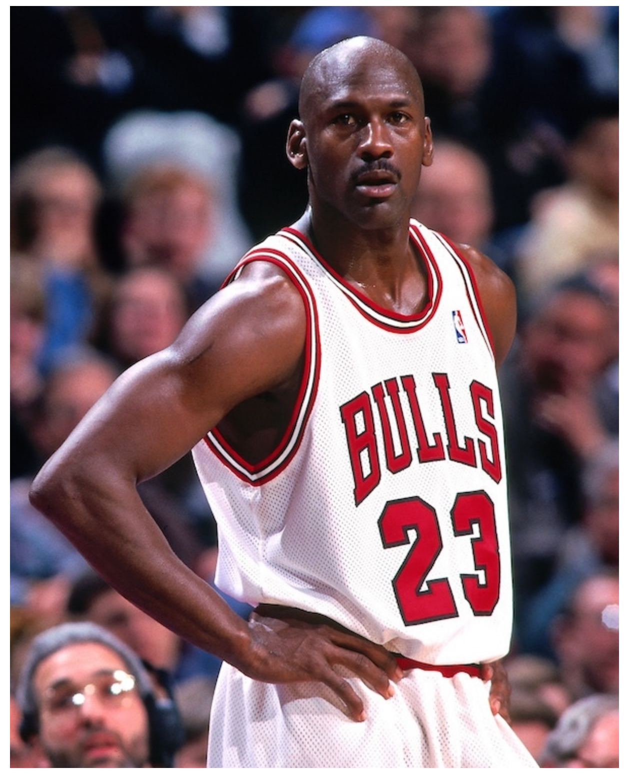
Shopping cart
basketball
ps5

Merge by the client

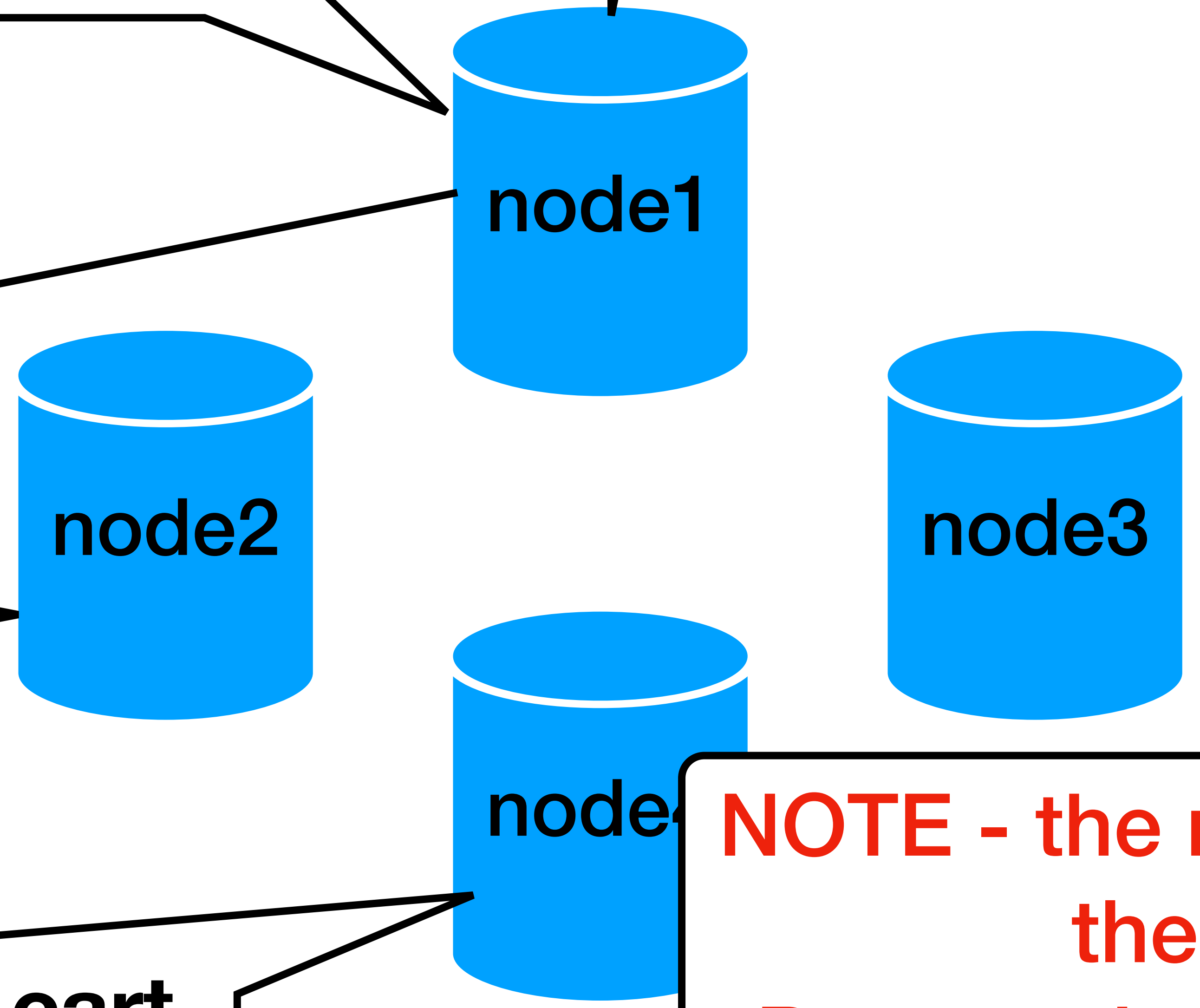
Data versioning (5) - motivation example

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app



Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

`de1,1`

Shopping cart
basketball
shoes

Shopping cart
basketball
shoes
ps5

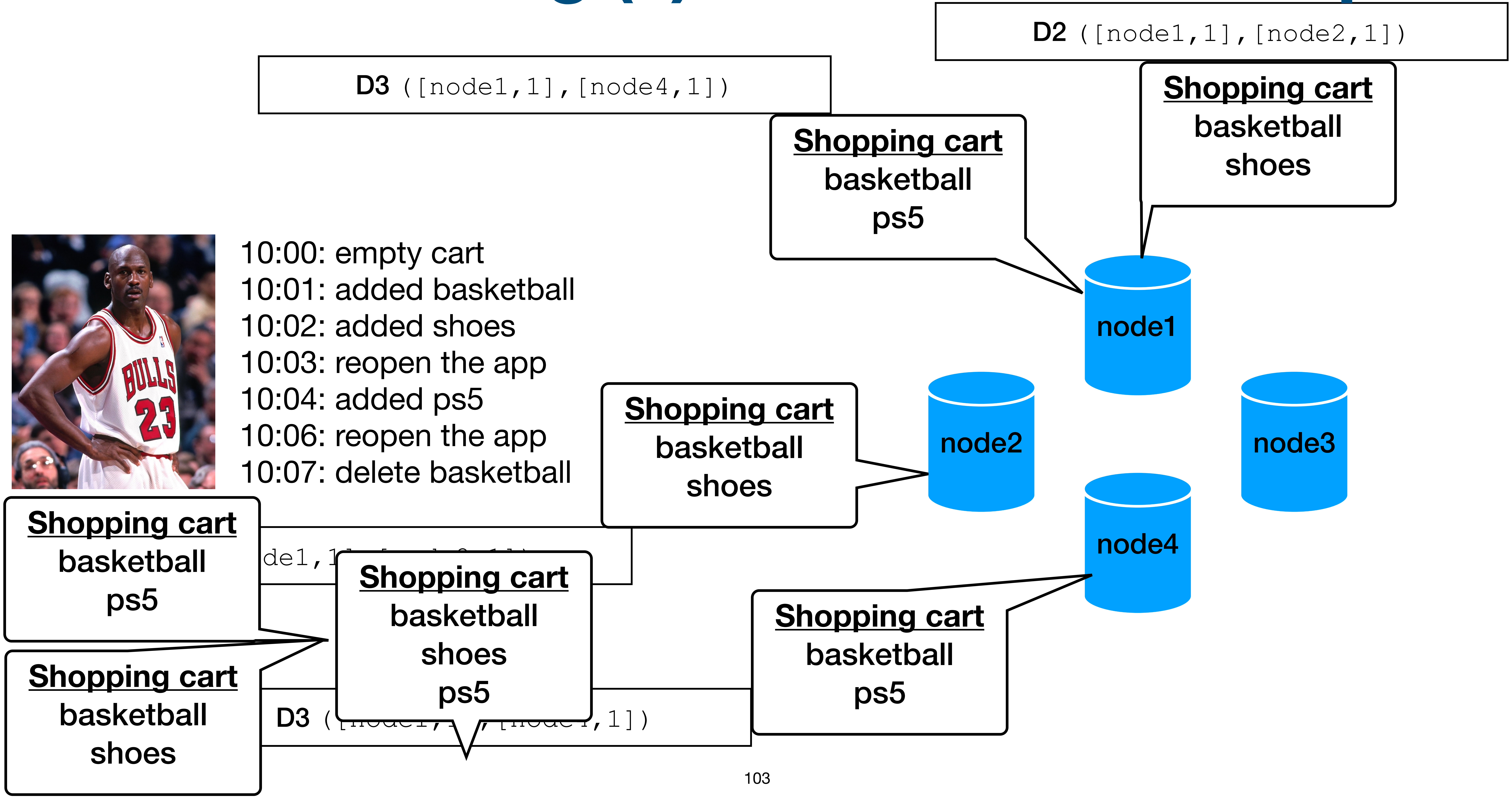
D3 `(` `,1])`

Shopping cart
basketball
ps5

Merge by the client

NOTE - the merge is still on the client.
Dynamo has no idea yet - only after the next write

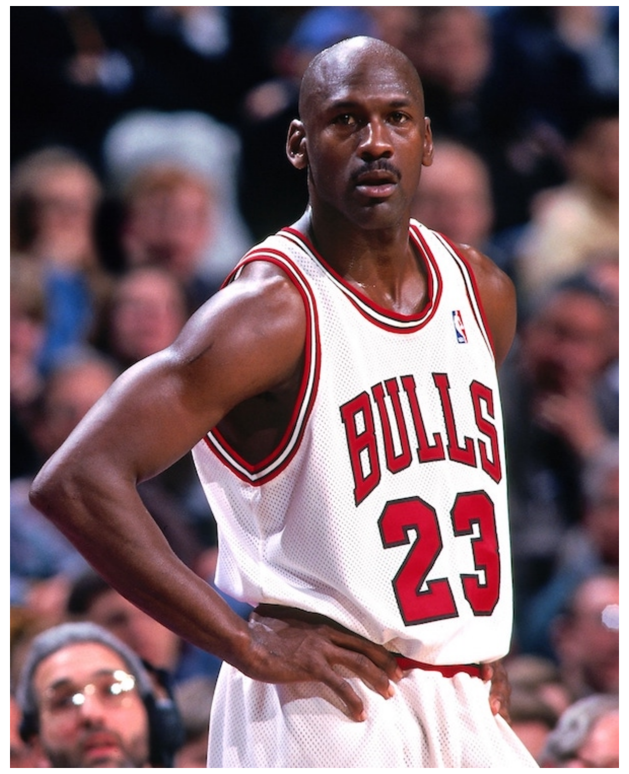
Data versioning (5) - motivation example



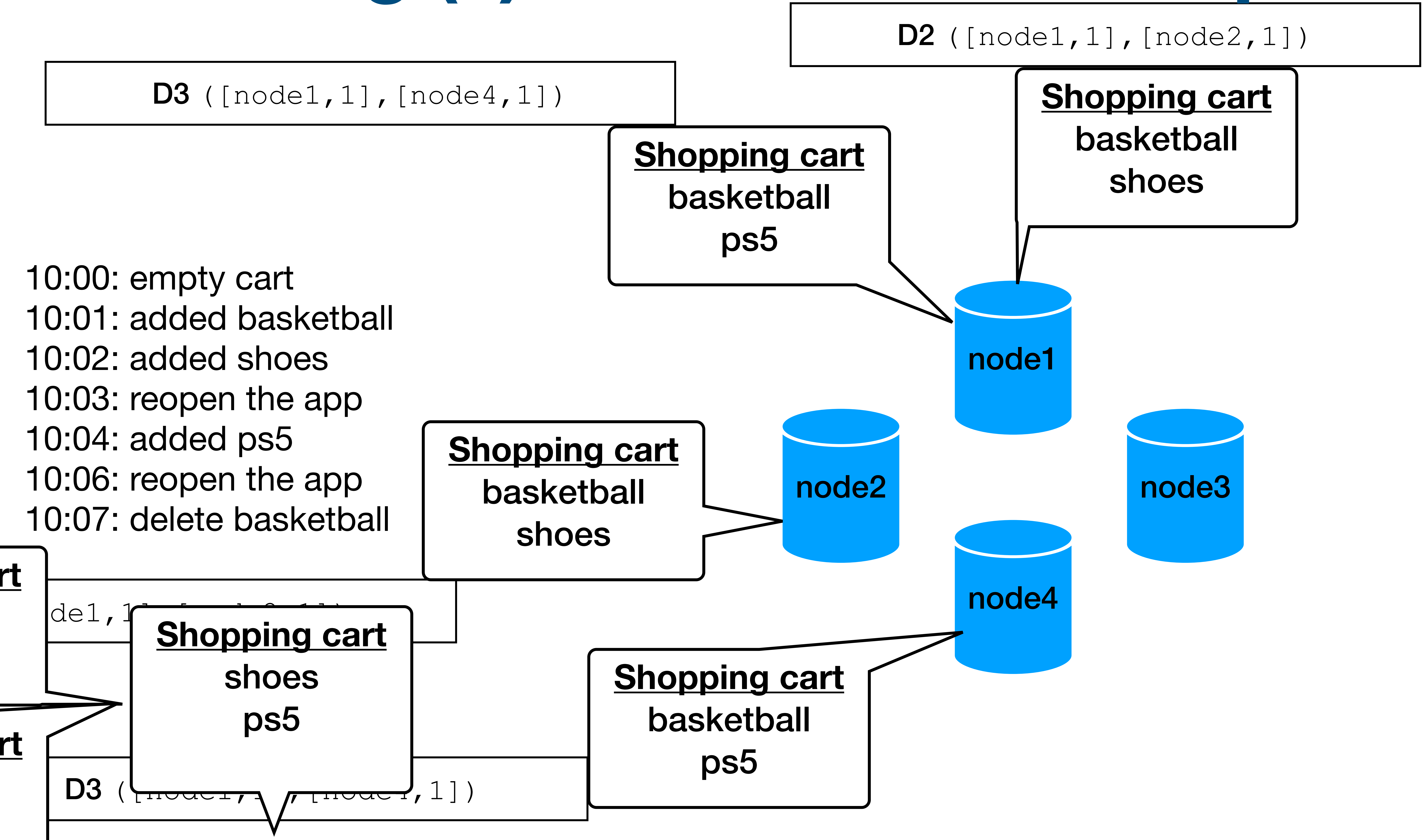
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

D2 ([node1,1],[node2,1])
D3 ([node1,1],[node4,1])
Shopping cart basketball shoes
Shopping cart basketball ps5
node1
Shopping cart basketball shoes
node2
node3
node4
Shopping cart basketball ps5
Shopping cart basketball ps5
de1,1
Shopping cart basketball shoes ps5
Shopping cart basketball shoes
D3 (,1])

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
shoes
ps5

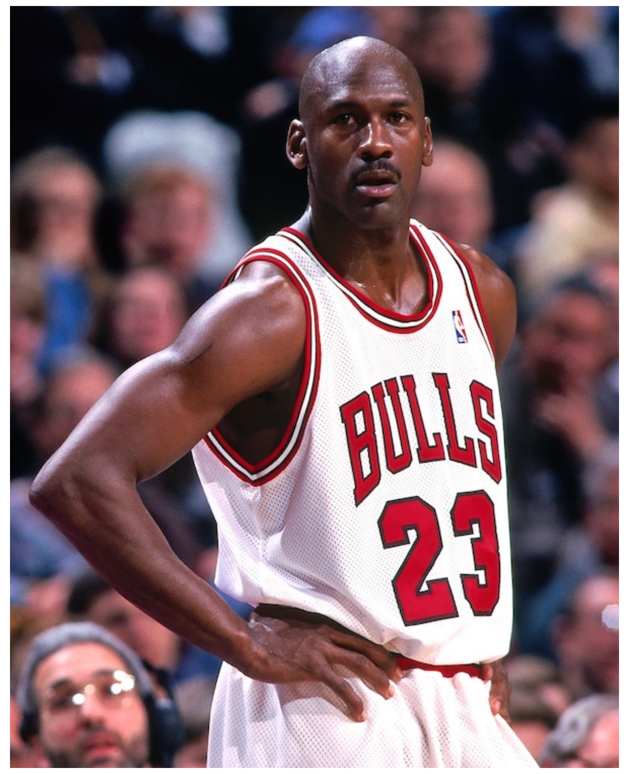
`de1,1],[node2,1])`

D3 `([node1,1],[node4,1])`

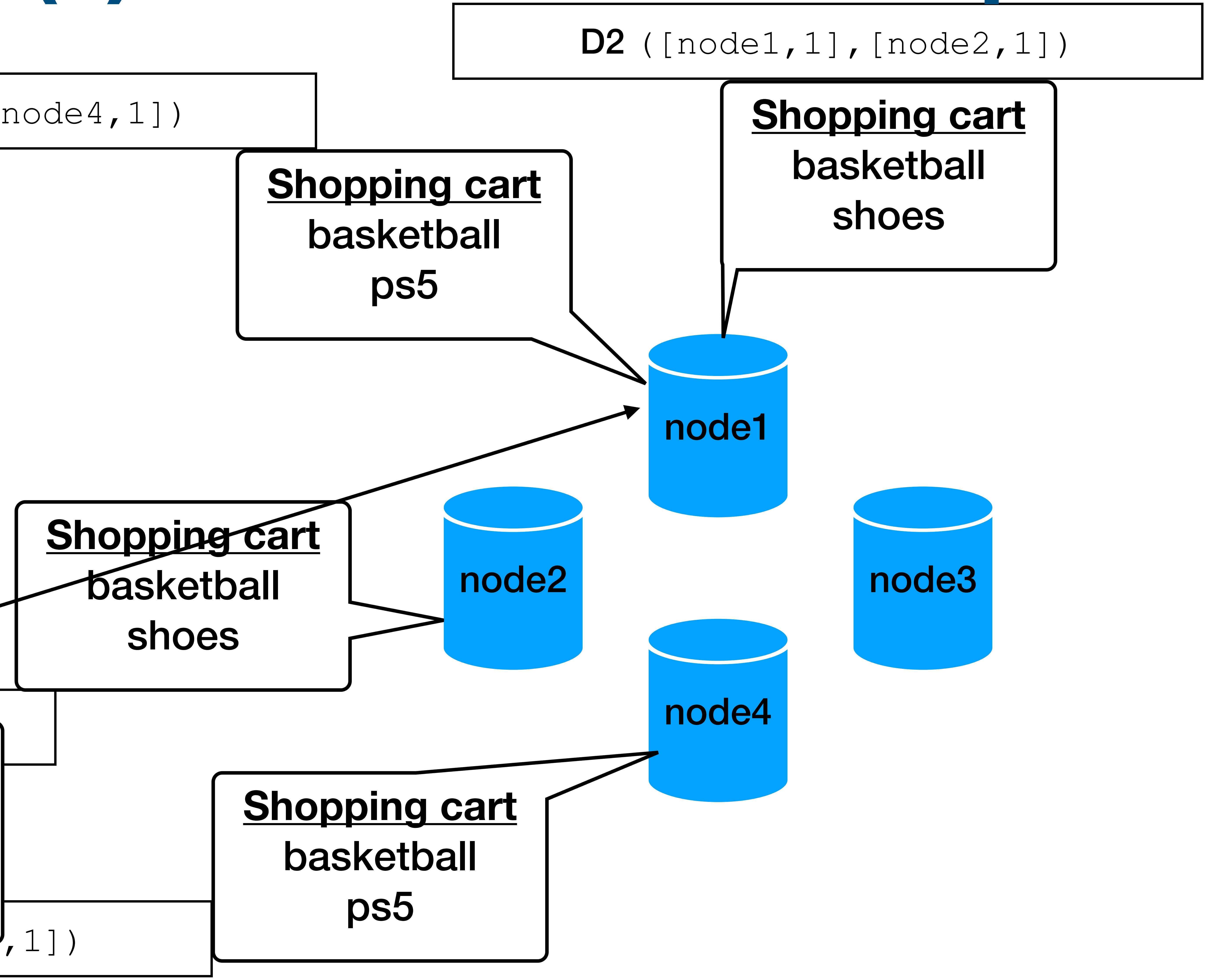
Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



D2 `([node1,1],[node2,1])`

Shopping cart
basketball
shoes

D3 `([node1,1],[node4,1])`

Shopping cart
basketball
ps5

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
basketball
ps5

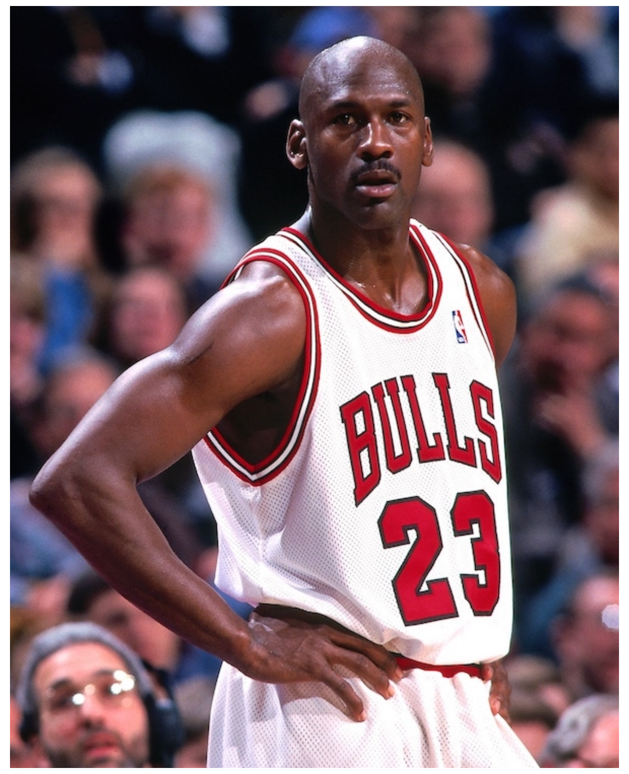
Shopping cart
basketball
shoes

`de1,1],[node2,1])`

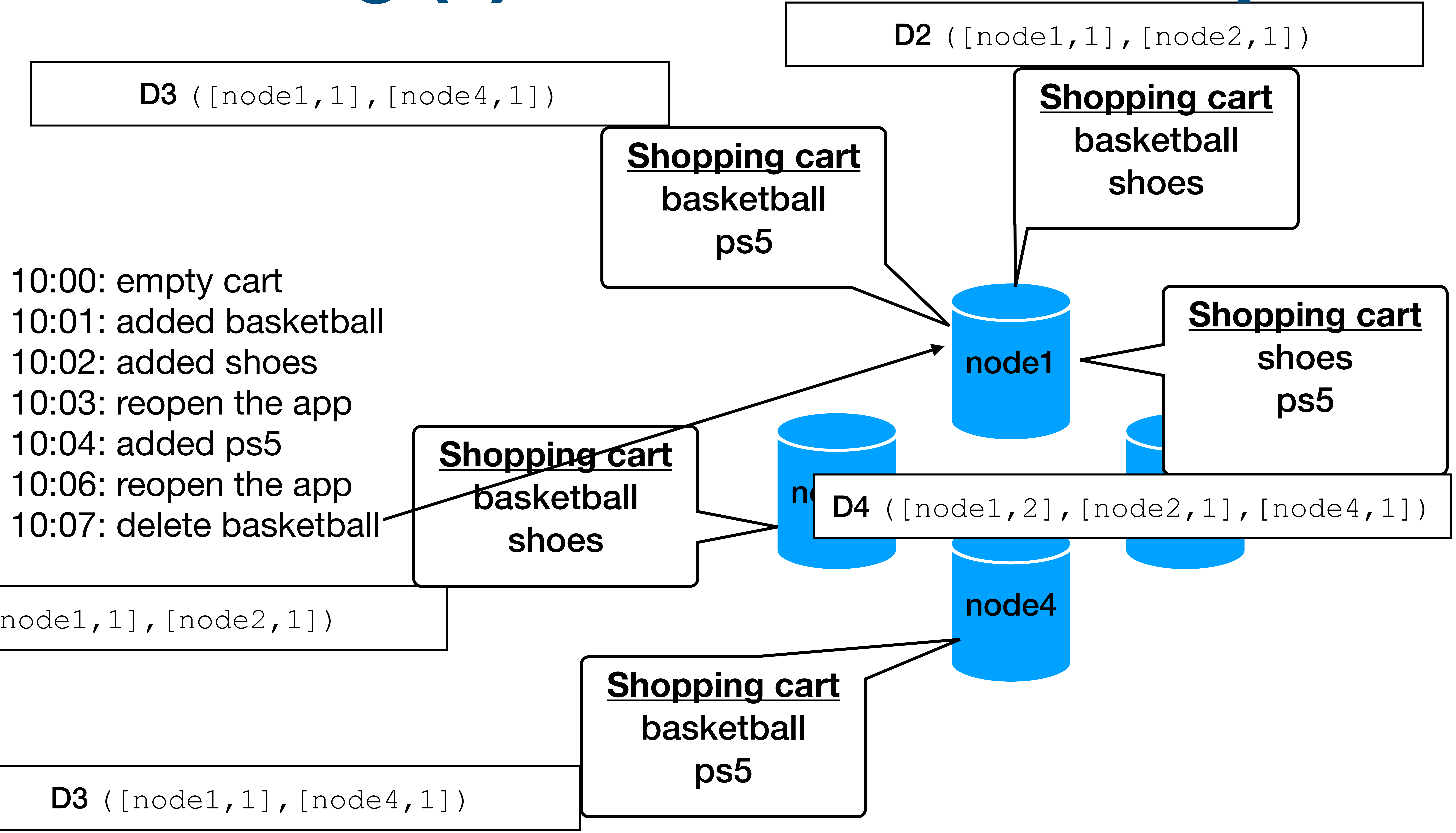
Shopping cart
shoes
ps5

D3 `(` `,1])`

Data versioning (5) - motivation example



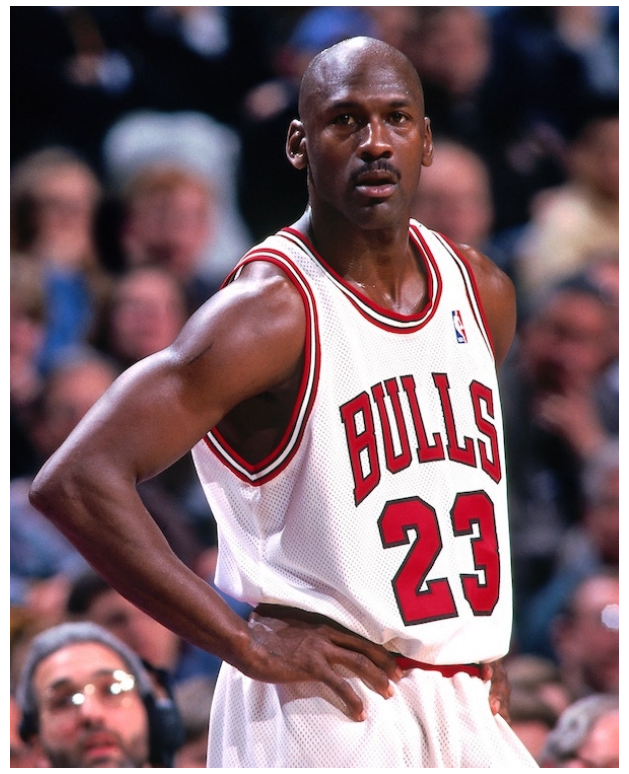
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



D2 ([node1,1],[node2,1])
D3 ([node1,1],[node4,1])
Shopping cart basketball shoes
Shopping cart basketball ps5
node1
Shopping cart shoes ps5
Shopping cart basketball shoes
n
D4 ([node1,2],[node2,1],[node4,1])
D2 ([node1,1],[node2,1])
node4
Shopping cart basketball ps5
D3 ([node1,1],[node4,1])

Data versioning (5) - motivation example

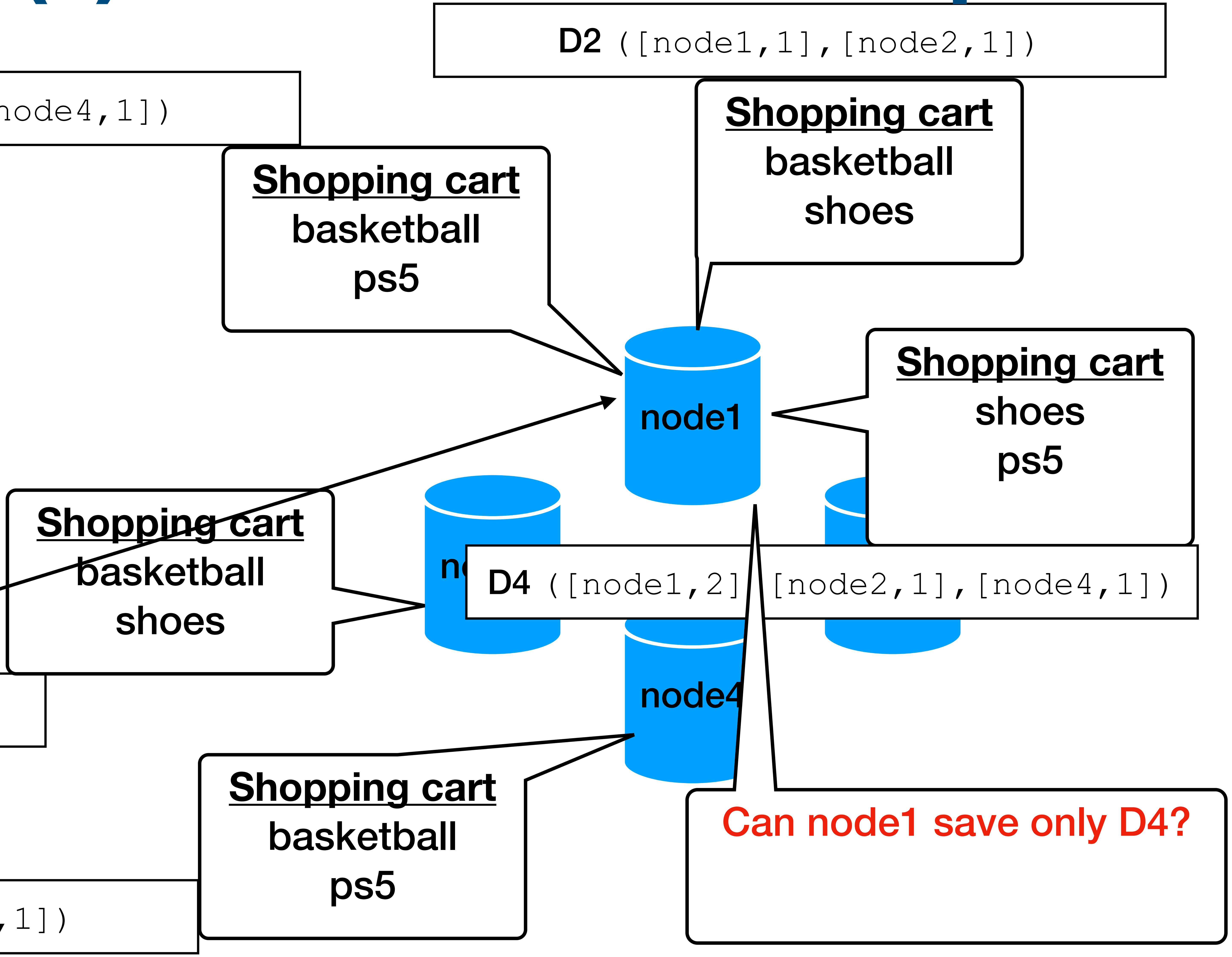
D3 `([node1,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`



D2 `([node1,1],[node2,1])`

Shopping cart
basketball
shoes

Shopping cart
basketball
ps5

Shopping cart
shoes
ps5

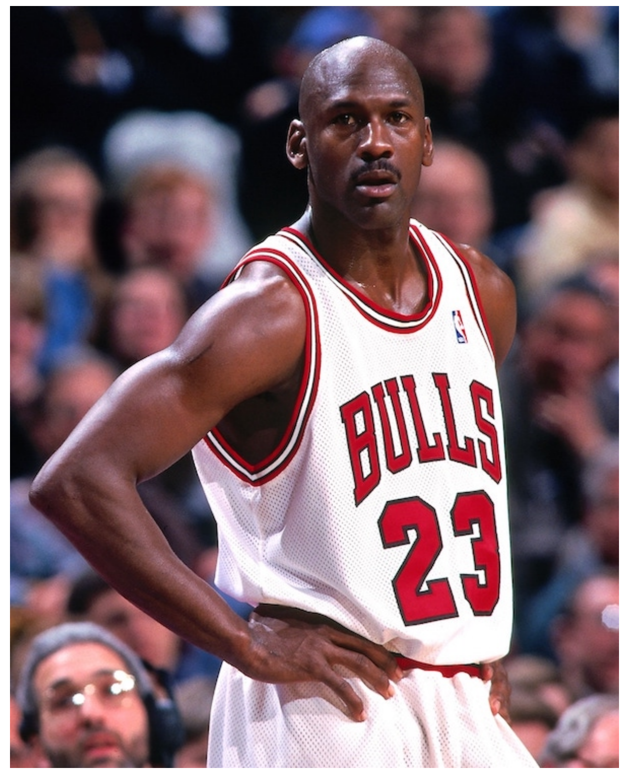
Shopping cart
basketball
shoes

D4 `([node1,2],[node2,1],[node4,1])`

Shopping cart
basketball
ps5

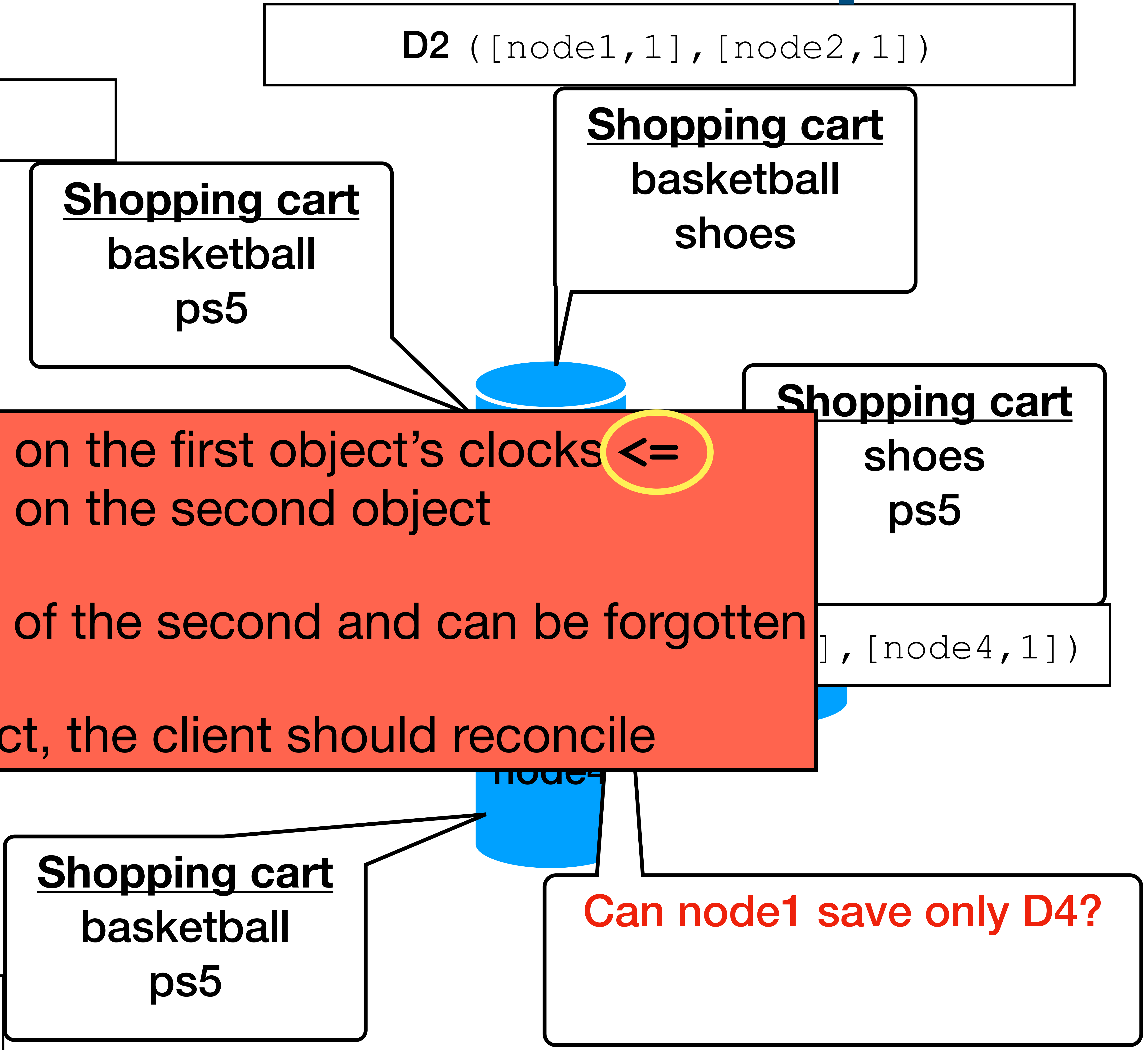
Can node1 save only D4?

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02:
10:03:
10:04:
10:06:
10:07:

```
IF      all the counters on the first object’s clocks <=
        all the counters on the second object
THEN
        first is ancestor of the second and can be forgotten
ELSE
        there is a conflict, the client should reconcile
```



D2 ([node1,1],[node2,1])

Shopping cart
basketball
shoes

D3 ([node1,1],[node4,1])

Shopping cart
basketball
ps5

Shopping cart
shoes
ps5

],[node4,1])

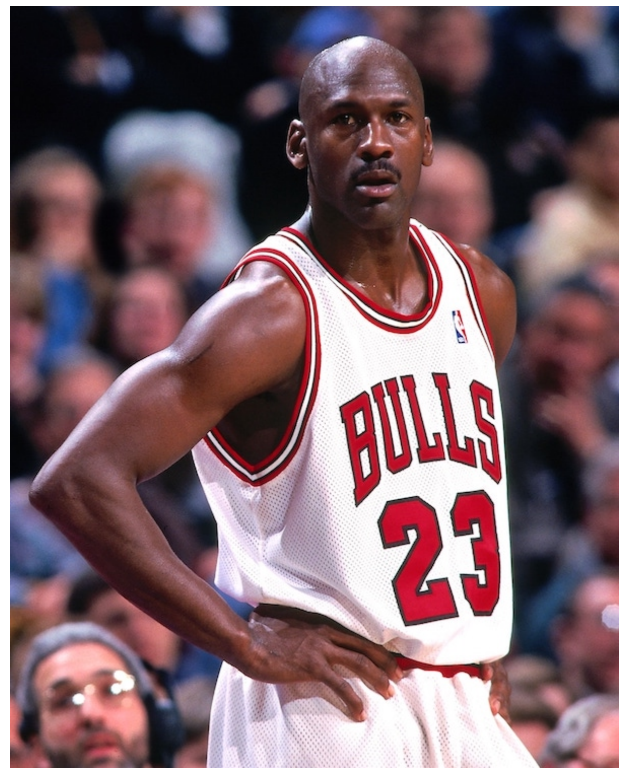
D2 ([node1,1],[node2,1])

Shopping cart
basketball
ps5

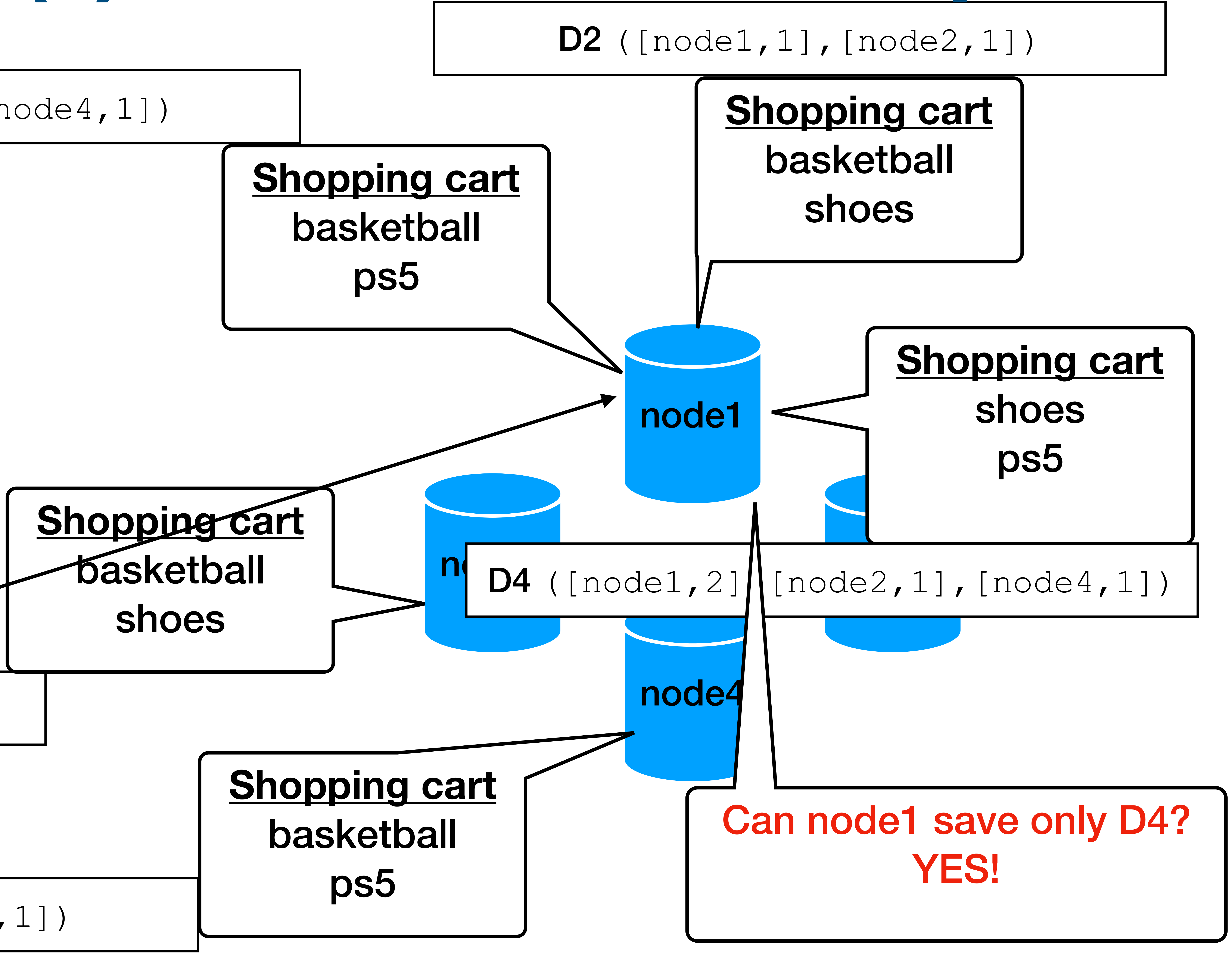
D3 ([node1,1],[node4,1])

Can node1 save only D4?

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



D2 ([node1,1],[node2,1])

Shopping cart
basketball
shoes

D3 ([node1,1],[node4,1])

Shopping cart
basketball
ps5

Shopping cart
shoes
ps5

D4 ([node1,2],[node2,1],[node4,1])

Shopping cart
basketball
shoes

D2 ([node1,1],[node2,1])

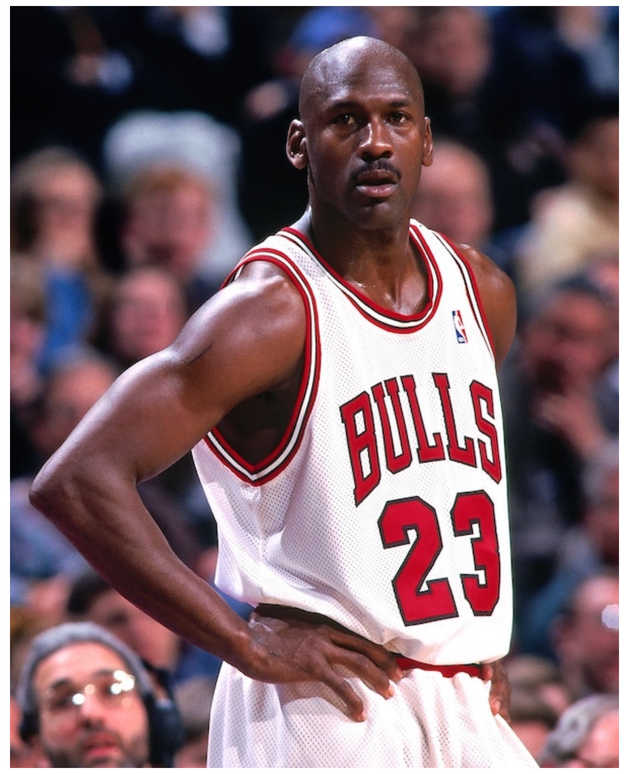
Shopping cart
basketball
ps5

D3 ([node1,1],[node4,1])

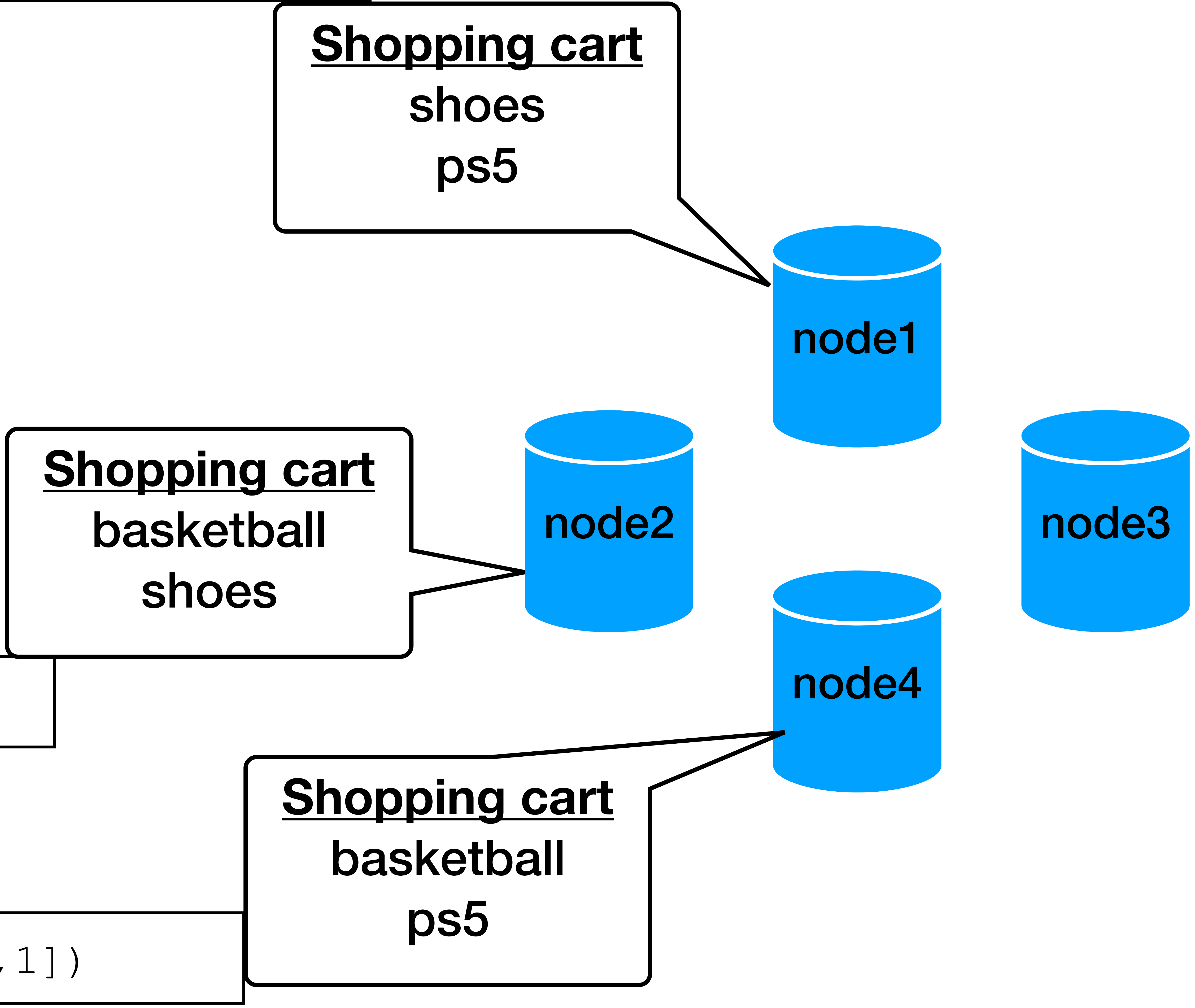
Can node1 save only D4?
YES!

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



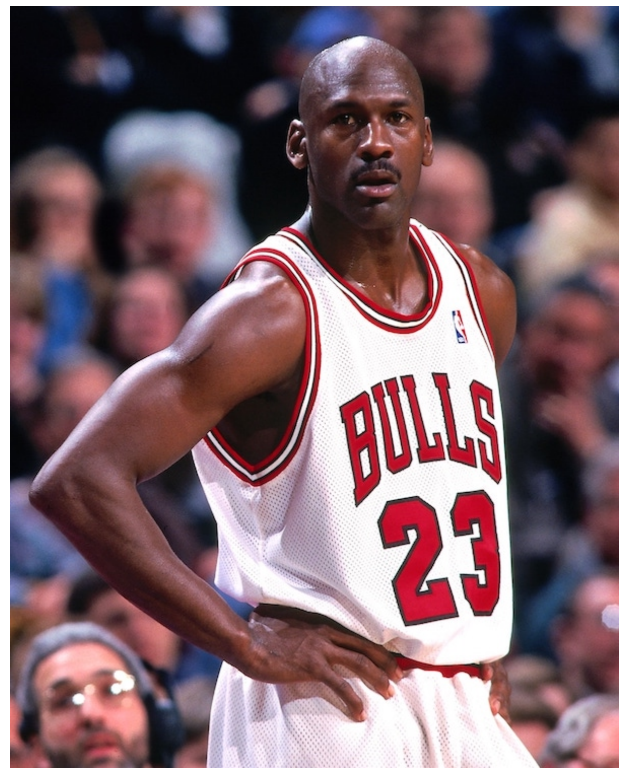
D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`

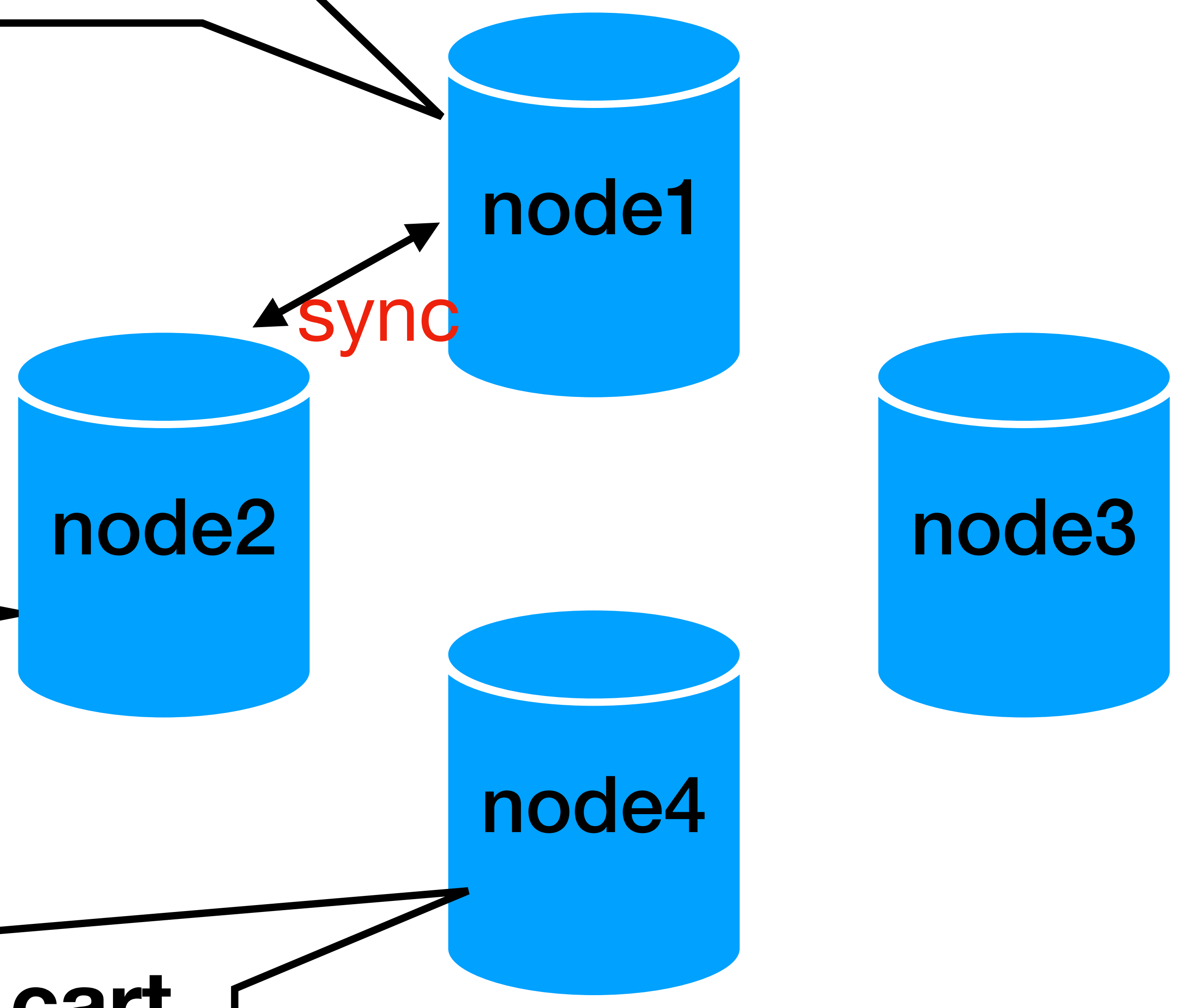
Shopping cart
shoes
ps5



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

Shopping cart
basketball
shoes

D2 `([node1,1],[node2,1])`

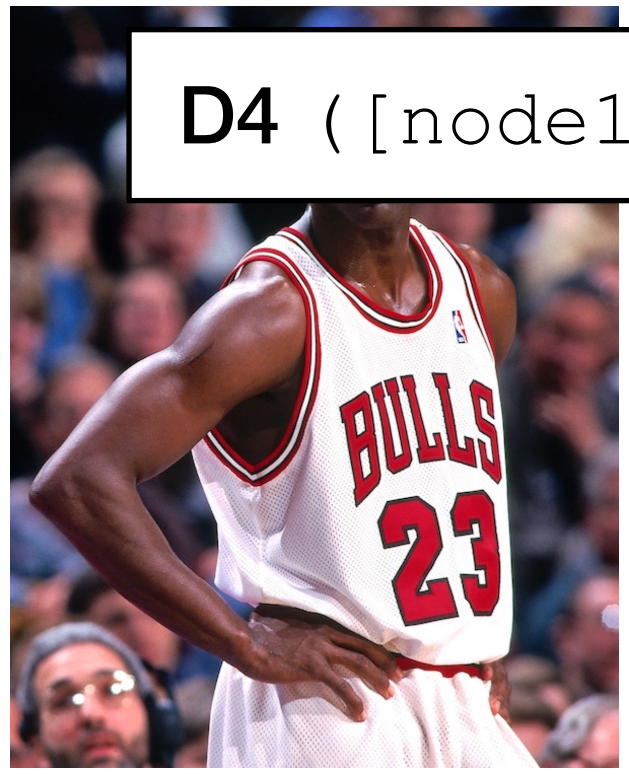


Shopping cart
basketball
ps5

D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`

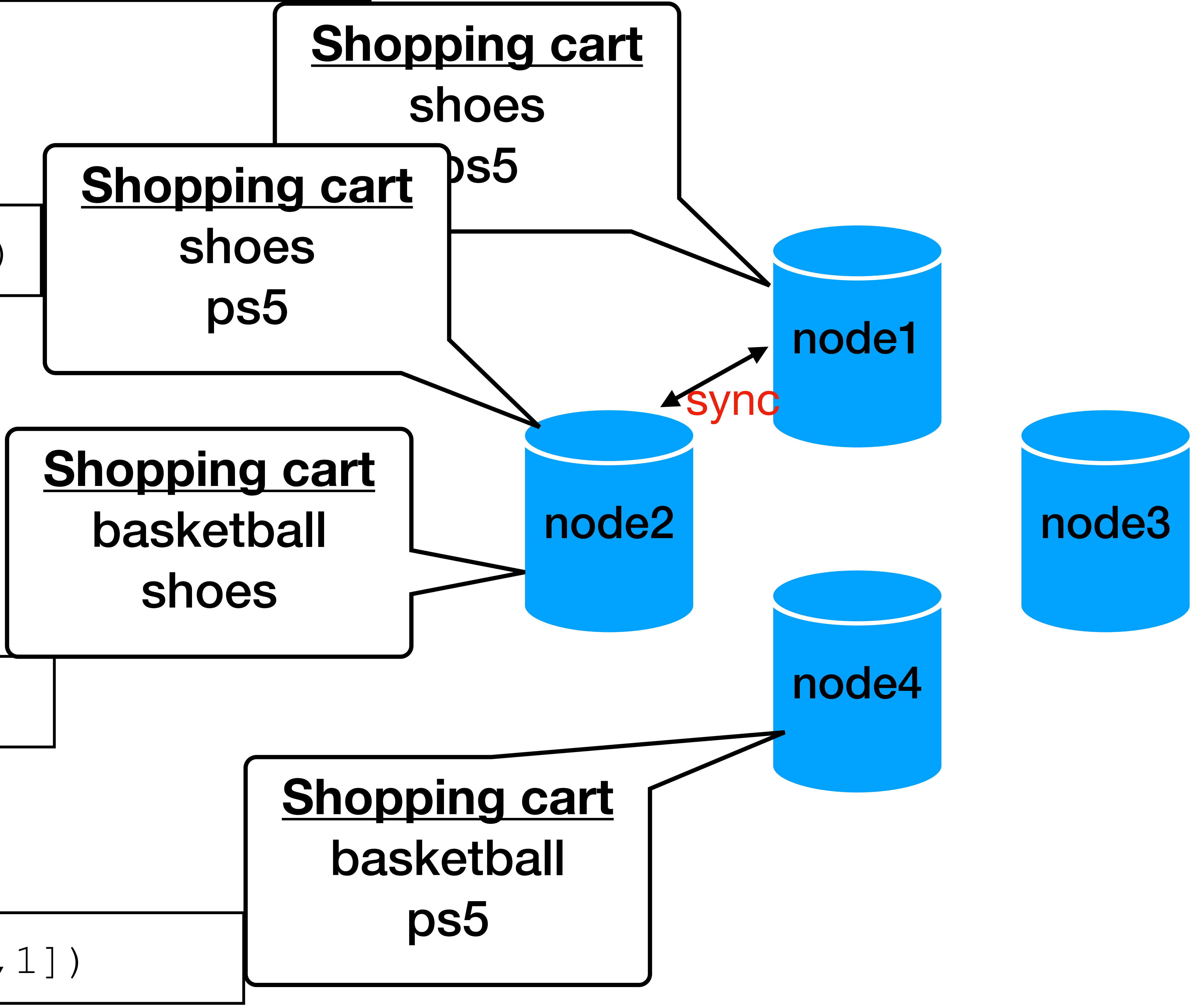


D4 `([node1,2],[node2,1],[node4,1])`

10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`



Shopping cart
shoes
ps5

Shopping cart
shoes
ps5

Shopping cart
basketball
shoes

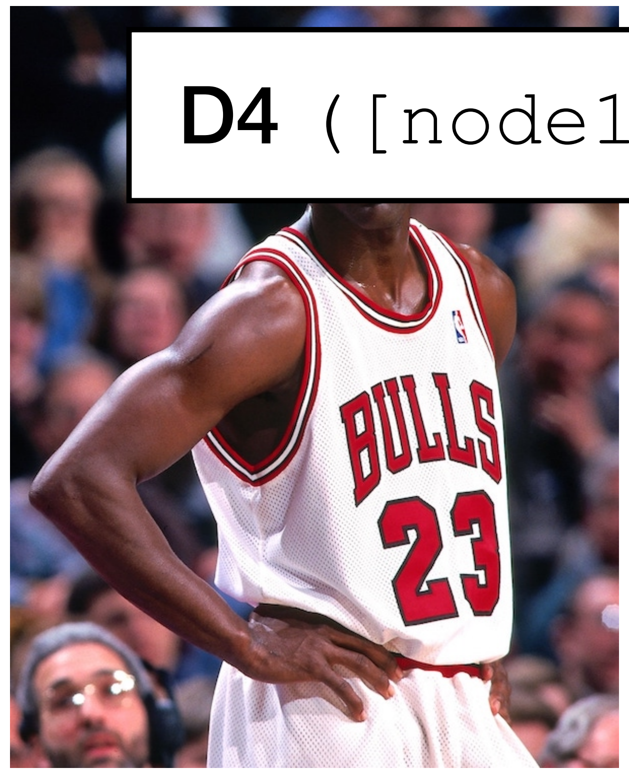
Shopping cart
basketball
ps5

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`

Can node2 save only D4?

Shopping cart
shoes
ps5

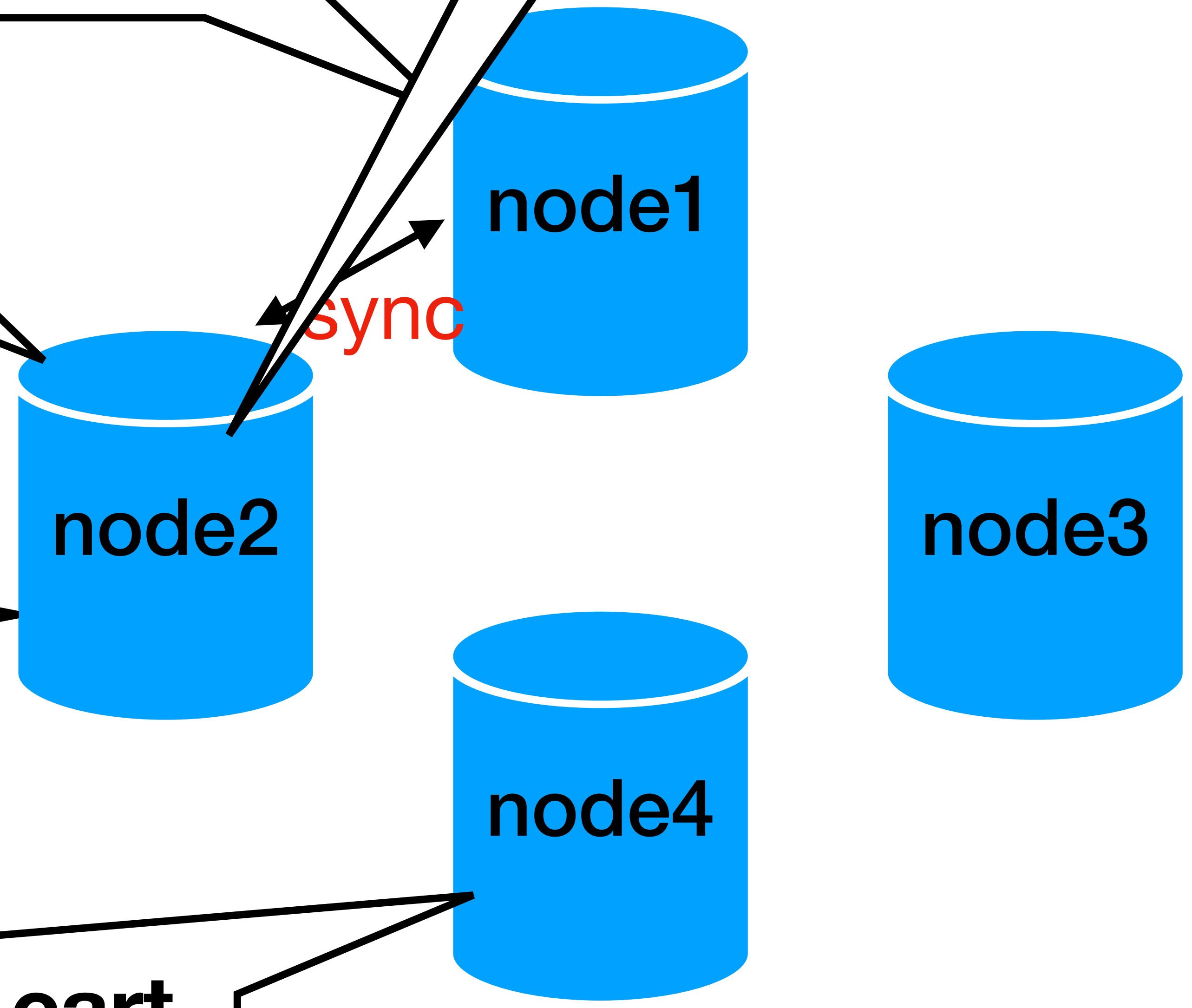


D4 `([node1,2],[node2,1],[node4,1])`

10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

Shopping cart
shoes
ps5

Shopping cart
basketball
shoes



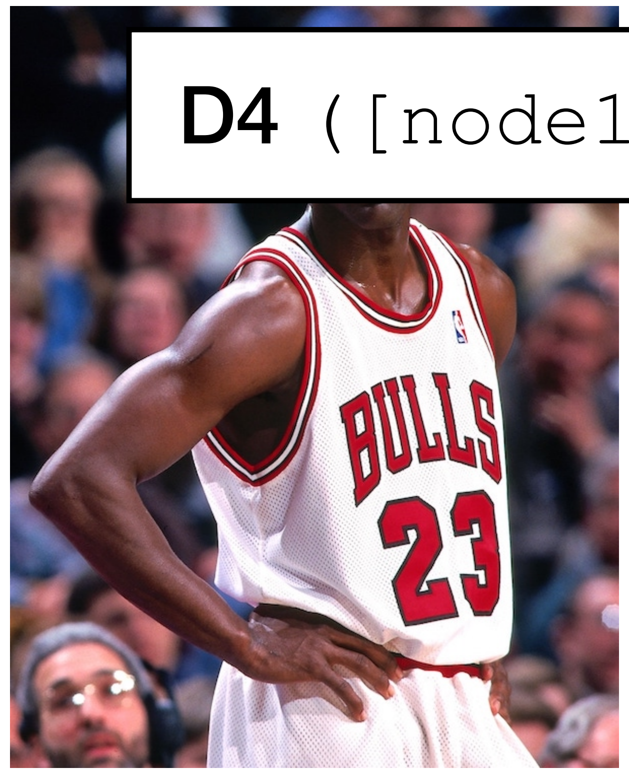
D2 `([node1,1],[node2,1])`

Shopping cart
basketball
ps5

D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`

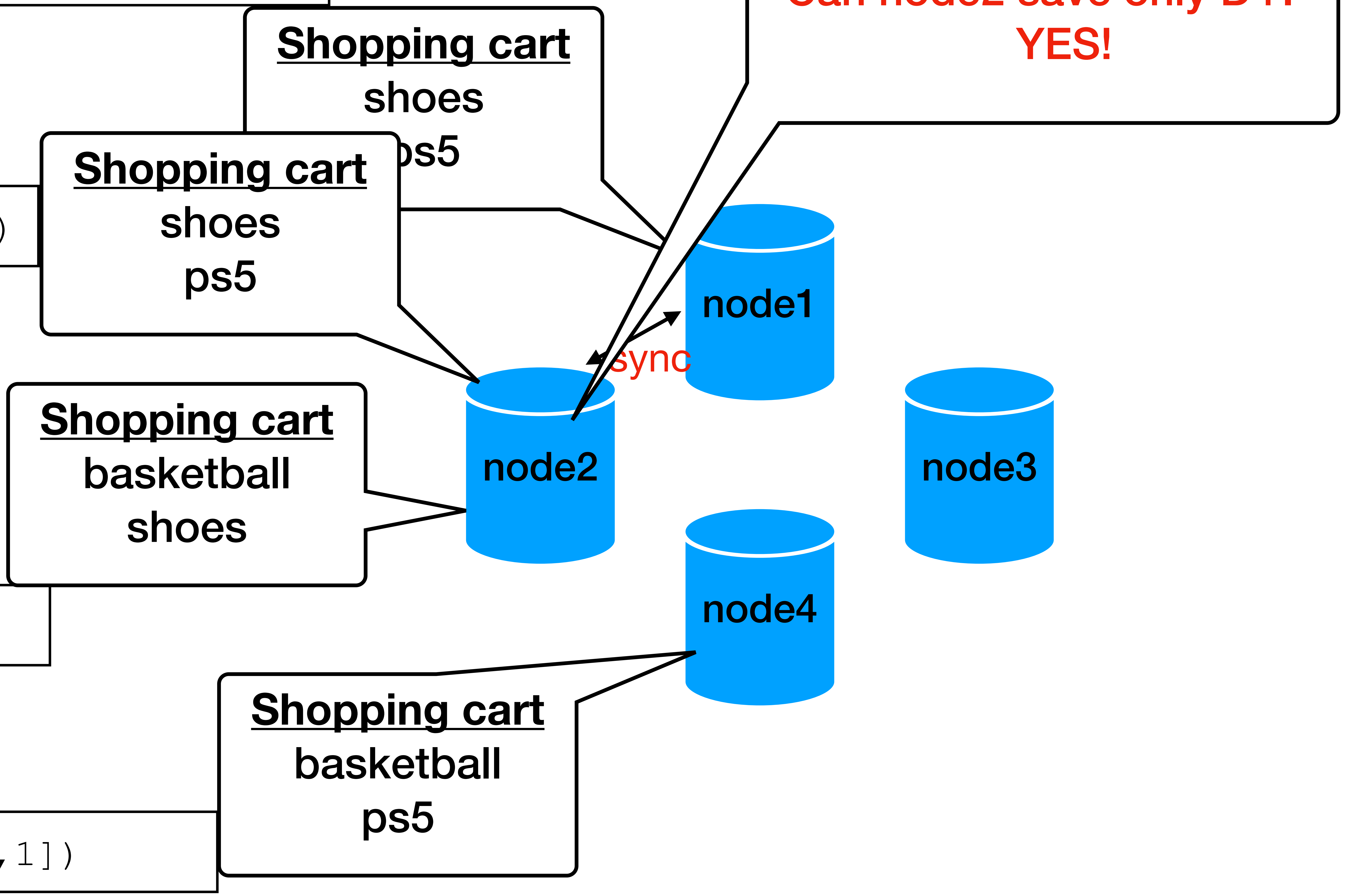


D4 `([node1,2],[node2,1],[node4,1])`

10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

D2 `([node1,1],[node2,1])`

D3 `([node1,1],[node4,1])`



Shopping cart
shoes
ps5

Shopping cart
shoes
ps5

Shopping cart
basketball
shoes

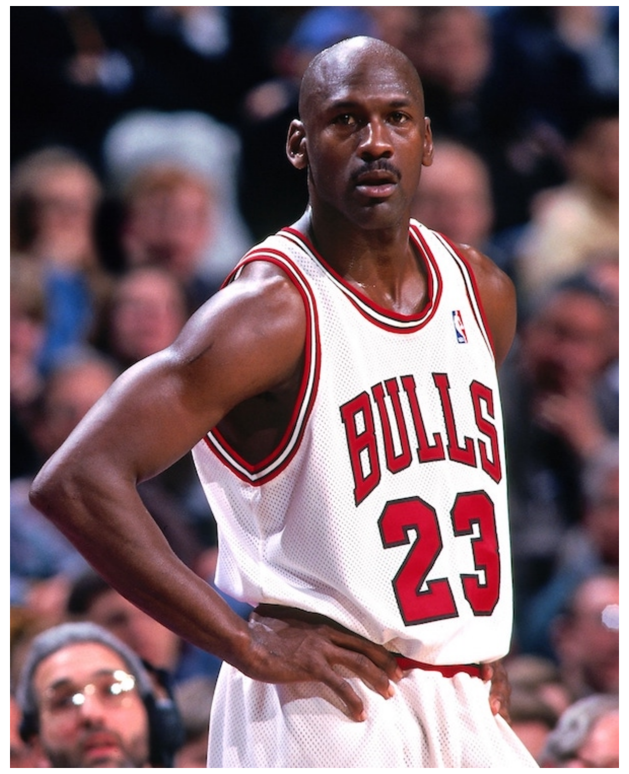
Shopping cart
basketball
ps5

Can node2 save only D4?
YES!

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`

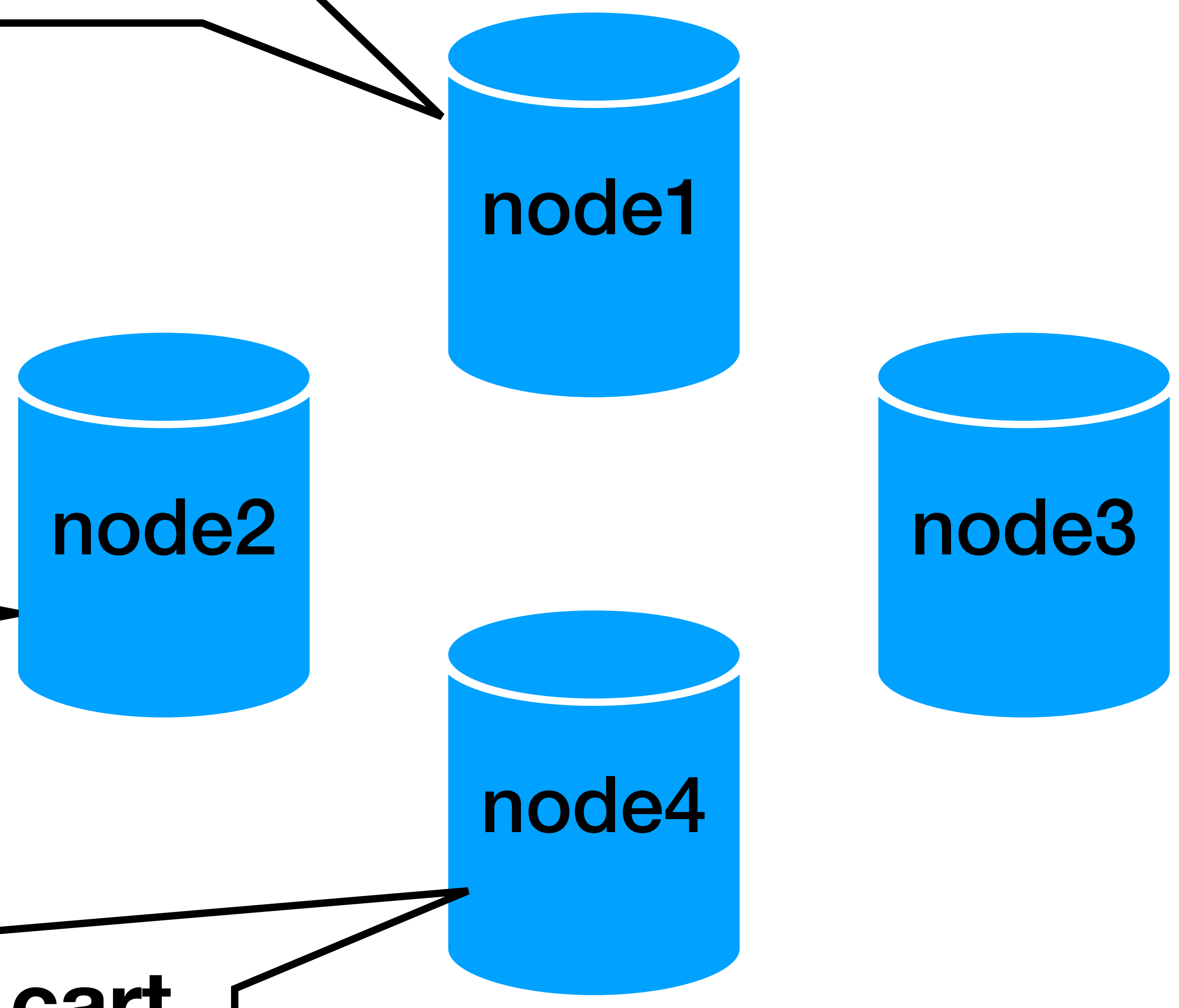
Shopping cart
shoes
ps5



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

Shopping cart
shoes
ps5

D4 `([node1,2],[node2,1],[node4,1])`

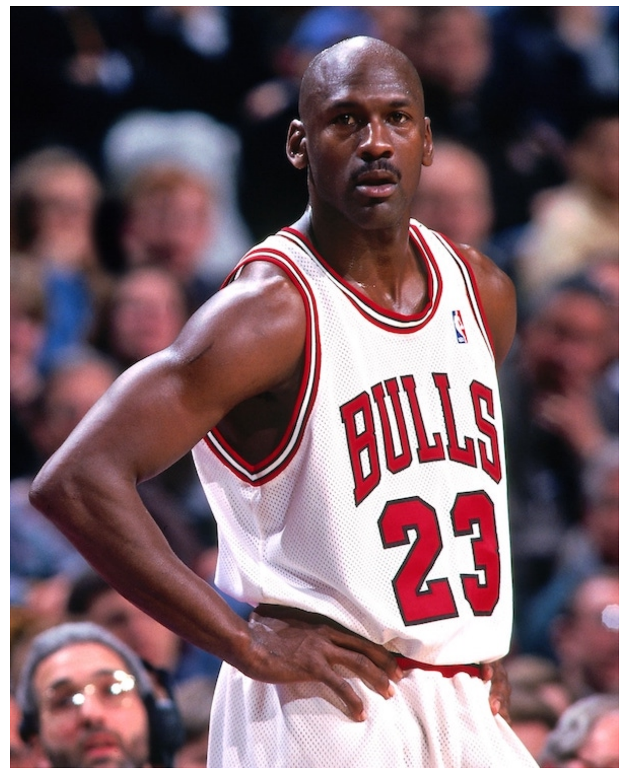


Shopping cart
basketball
ps5

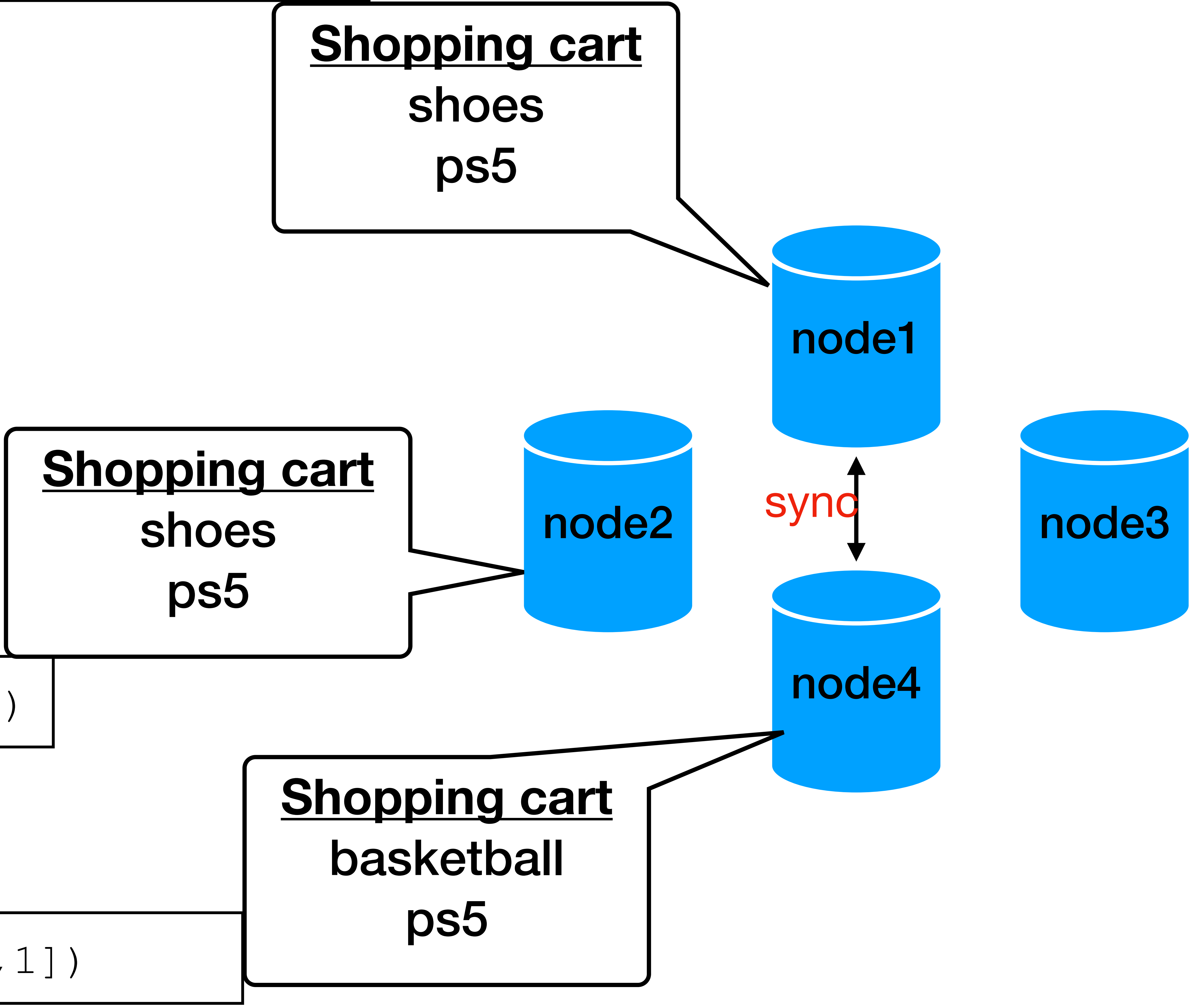
D3 `([node1,1],[node4,1])`

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

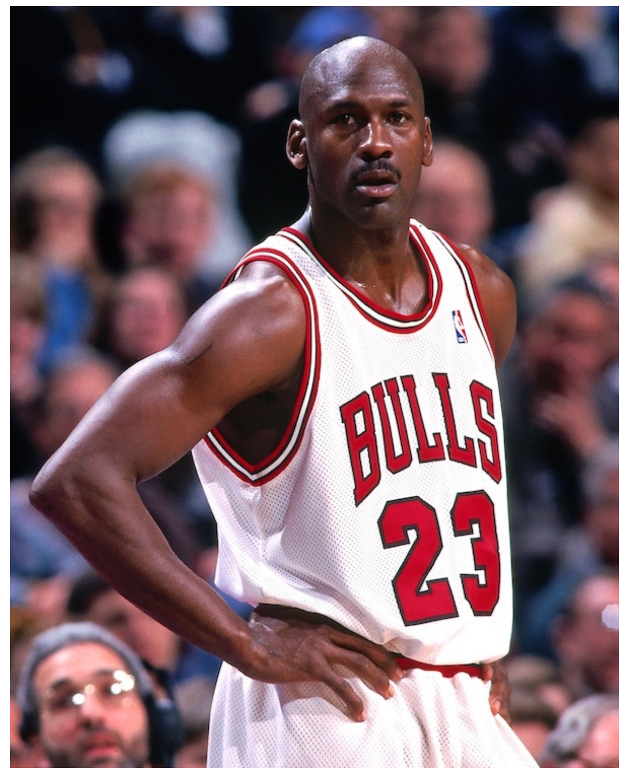


D4 `([node1,2],[node2,1],[node4,1])`

D3 `([node1,1],[node4,1])`

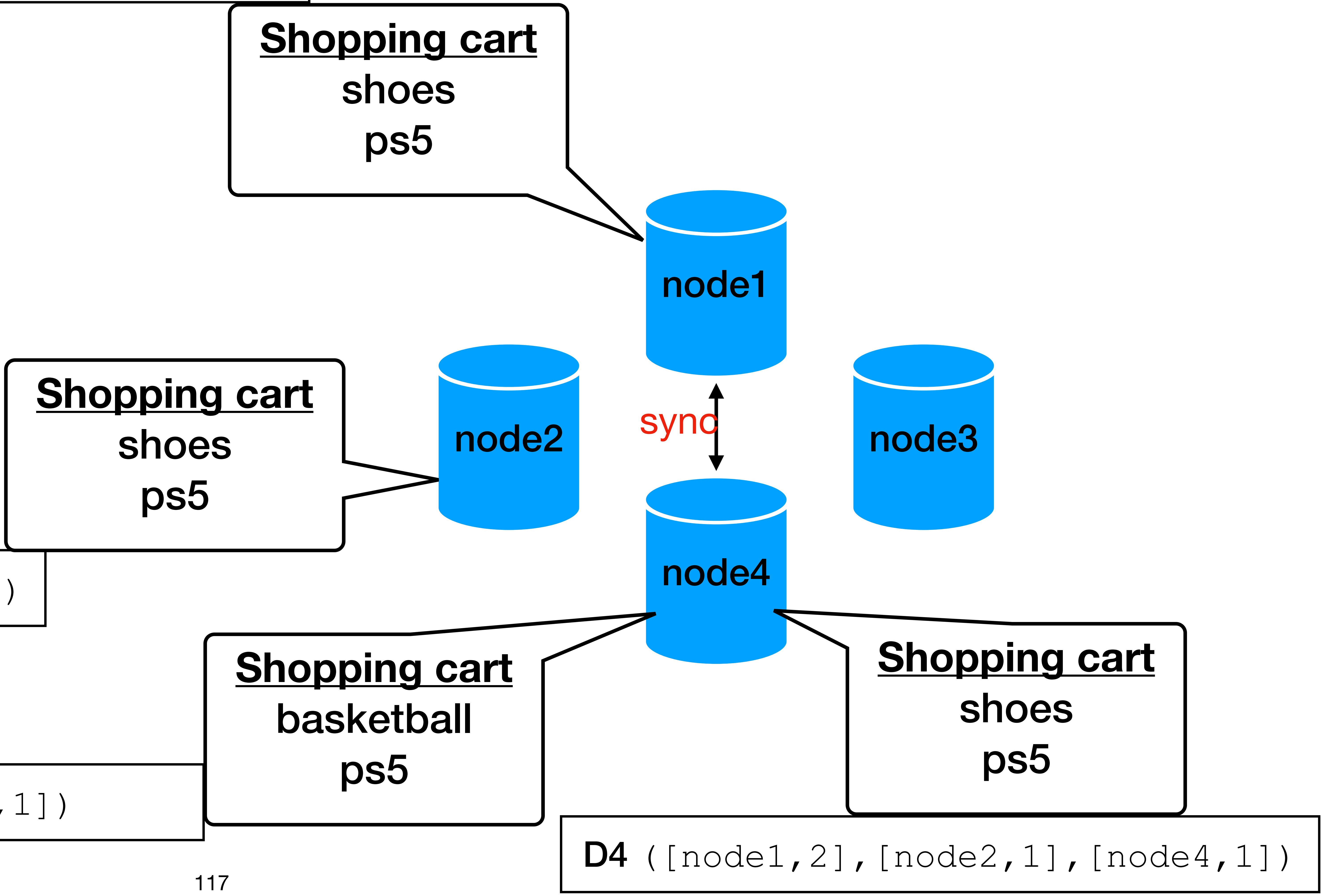
Data versioning (5) - motivation example

D4 ([node1,2],[node2,1],[node4,1])



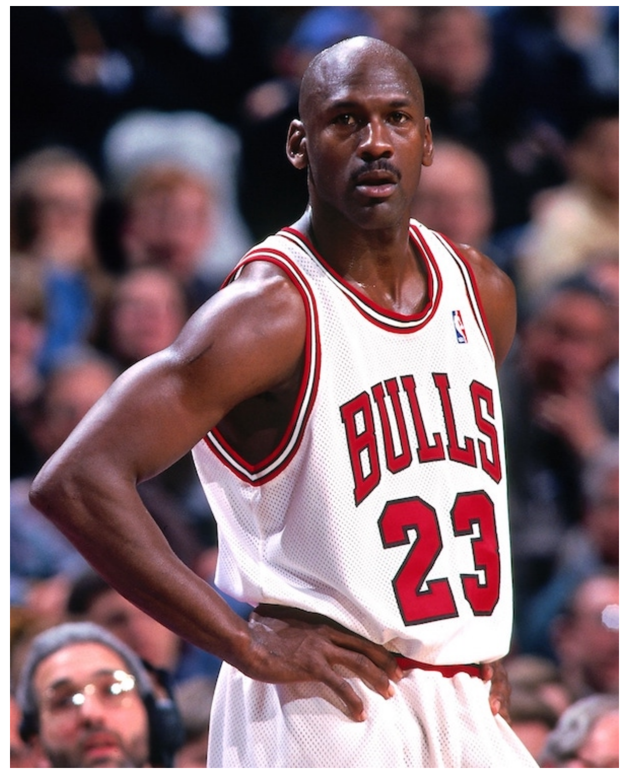
10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball

D4 ([node1,2],[node2,1],[node4,1])

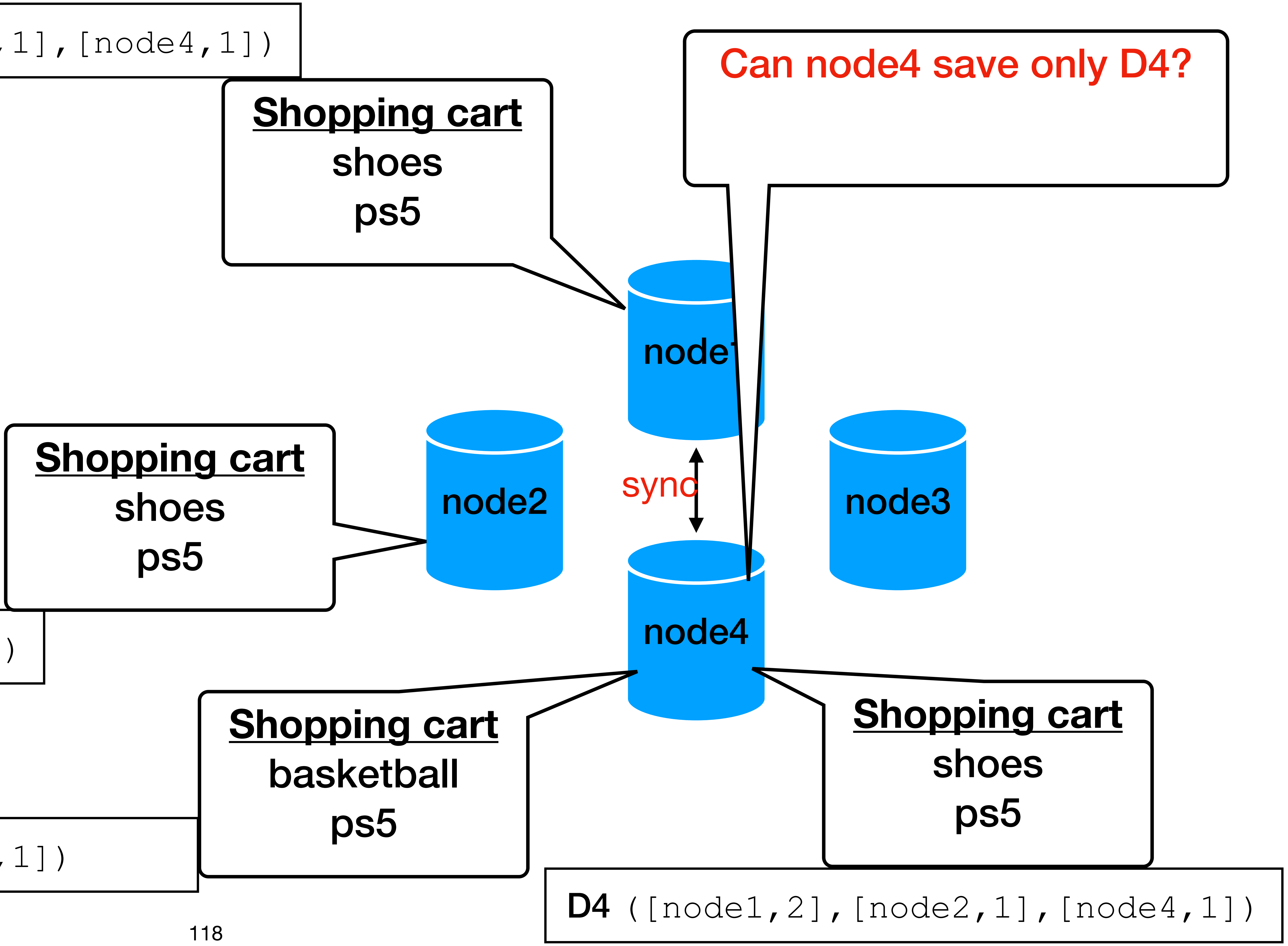


D3 ([node1,1],[node4,1])

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



D4 ([node1,2],[node2,1],[node4,1])

Shopping cart
shoes
ps5

Can node4 save only D4?

node1

node2

sync

node3

node4

Shopping cart
shoes
ps5

D4 ([node1,2],[node2,1],[node4,1])

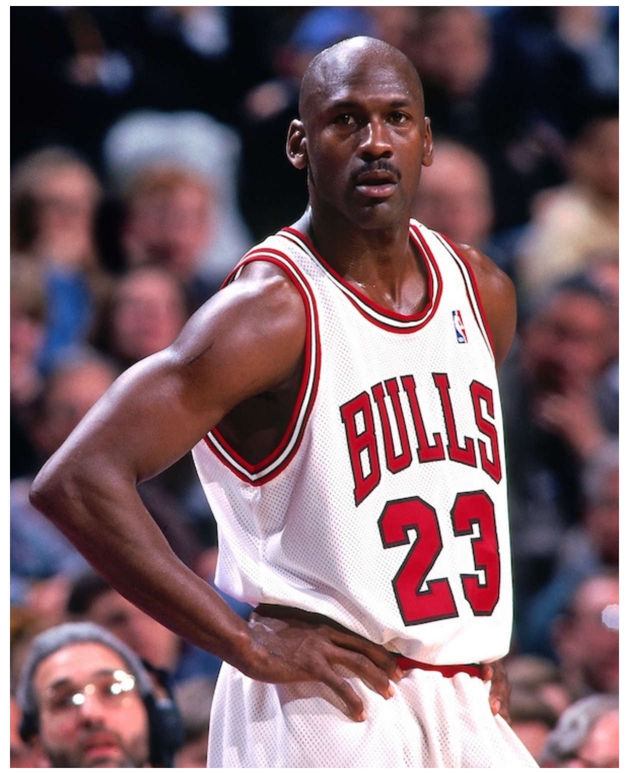
Shopping cart
basketball
ps5

D3 ([node1,1],[node4,1])

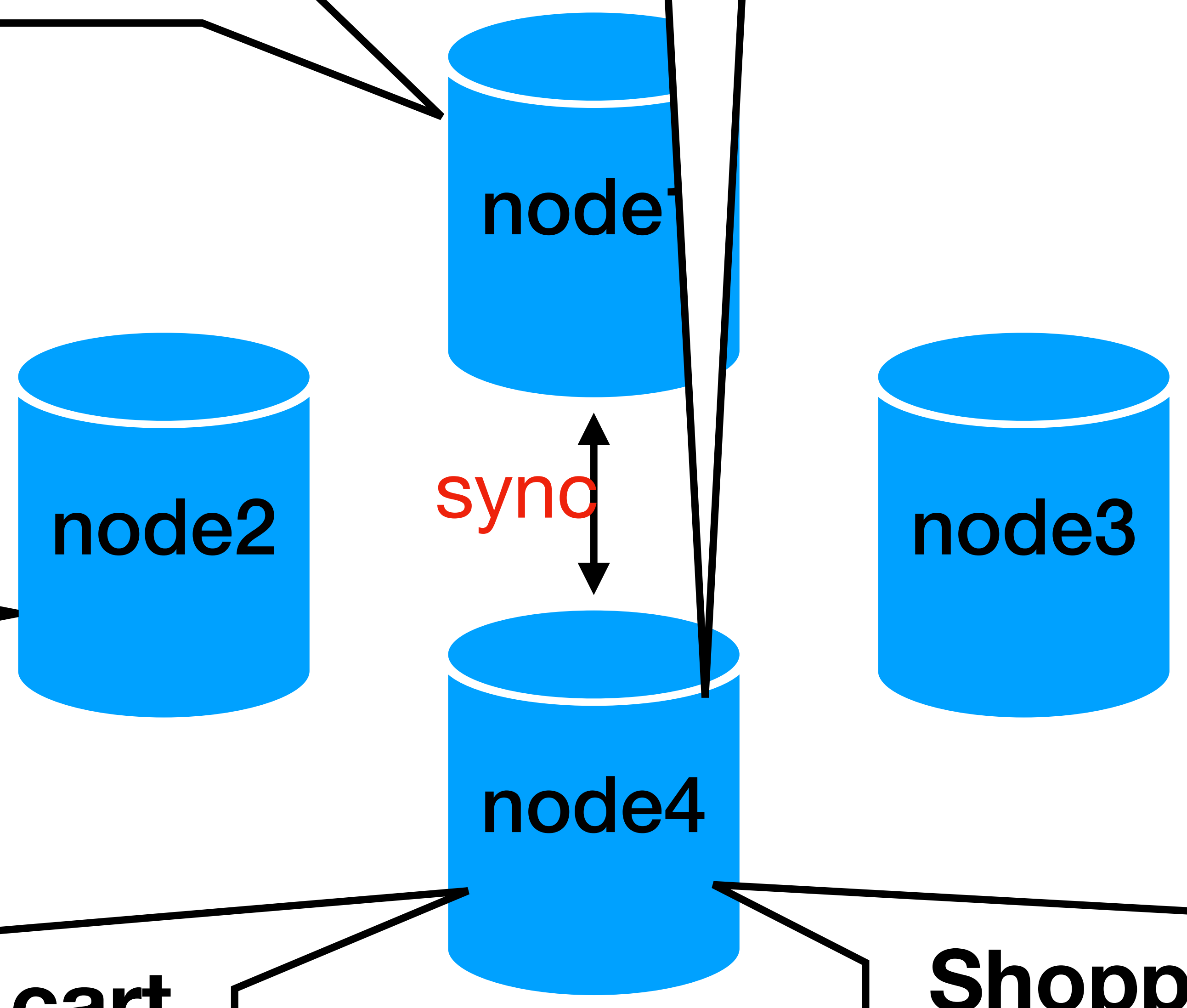
Shopping cart
shoes
ps5

D4 ([node1,2],[node2,1],[node4,1])

Data versioning (5) - motivation example



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



D4 ([node1,2],[node2,1],[node4,1])

Shopping cart
shoes
ps5

Can node4 save only D4?
YES!

Shopping cart
shoes
ps5

D4 ([node1,2],[node2,1],[node4,1])

Shopping cart
basketball
ps5

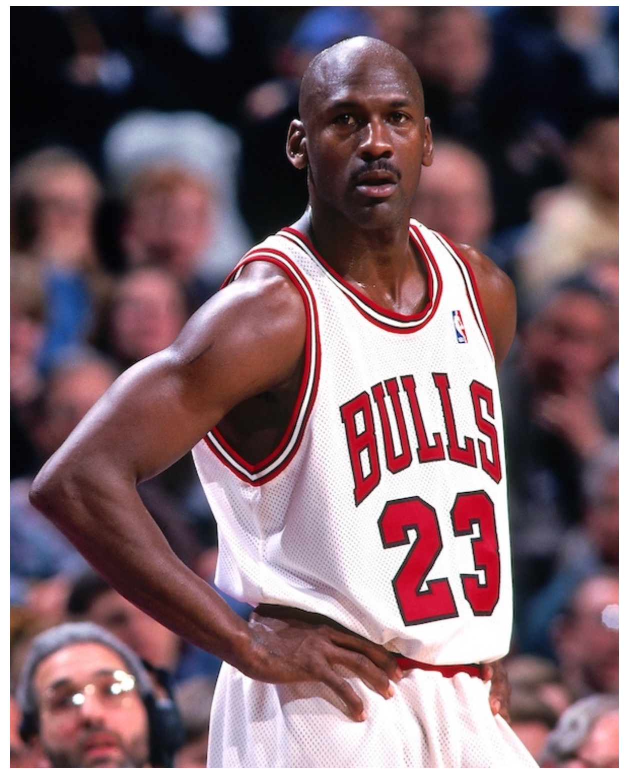
D3 ([node1,1],[node4,1])

Shopping cart
shoes
ps5

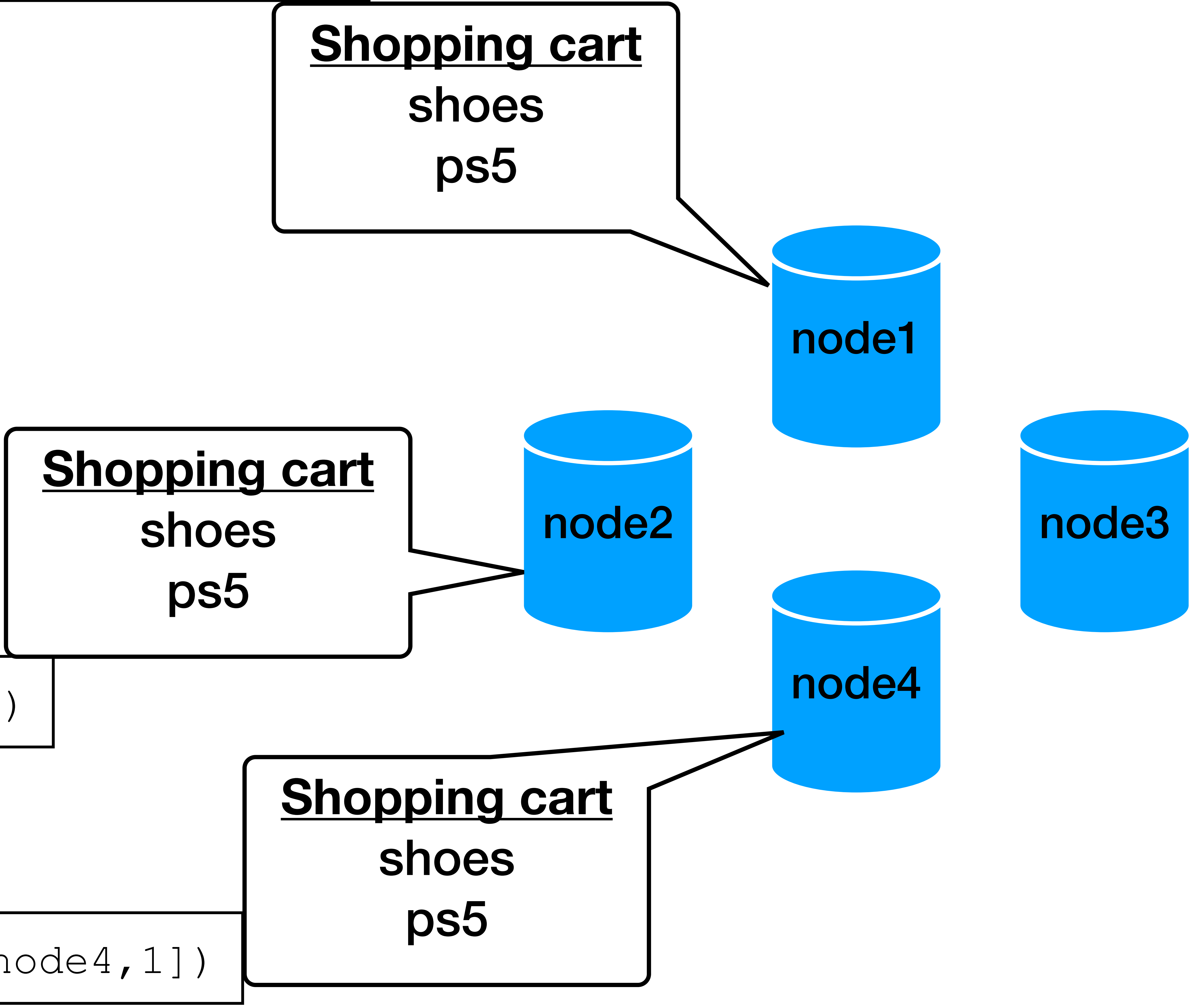
D4 ([node1,2],[node2,1],[node4,1])

Data versioning (5) - motivation example

D4 `([node1,2],[node2,1],[node4,1])`



10:00: empty cart
10:01: added basketball
10:02: added shoes
10:03: reopen the app
10:04: added ps5
10:06: reopen the app
10:07: delete basketball



Shopping cart
shoes
ps5

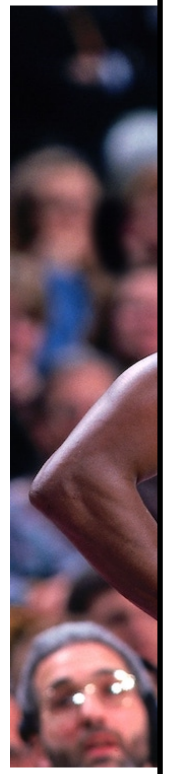
Shopping cart
shoes
ps5

Shopping cart
shoes
ps5

D4 `([node1,2],[node2,1],[node4,1])`

D4 `([node1,2],[node2,1],[node4,1])`

Data versioning (5) - motivation example



מה כבר נגמר?!

https://www.youtube.com/watch?v=9jI-IFmLi_E

D4 ([node1,2],[node2,1],[node4,1])

Data versioning (5) - example (paper)

- If you want another example, check the extra slides

Data versioning (5) - Vector clocks size

- In theory, the size of the vector clocks can grow if many servers coordinate the write
  "preference list"

- In practice, it is always handled by one of the top N

- Amazon added a threshold (10) that above that,
  the oldest pair gets removed
    - can lead for reconciliation problems
    - this problem has not surfaced in production (according to Amazon)

Data versioning (6) - Nerd note

- You can NOT currently understand this slide yet
  but try to remember it for the future

- Cassandra does NOT use vector clocks

- It use a "simple” timestamp mechanism - **“last write wins”**
  - Clocks are naturally not 100% sync between different nodes
    but Cassandra has a mechanism to try and sync them all the time

- Works in practice because the “key-value” will be brake into smaller parts
  For example - each item of the shopping cart will have a different “key-value”

Bonus clip

https://www.youtube.com/watch?v=cMalJkGJzYU

Dynamo topics

- Requirements
- Partition algorithm
- Replication
- Data versioning
- **`get()` and `put()` execution**
- Failures
- Ring membership

`get()` and `put()` execution (1)

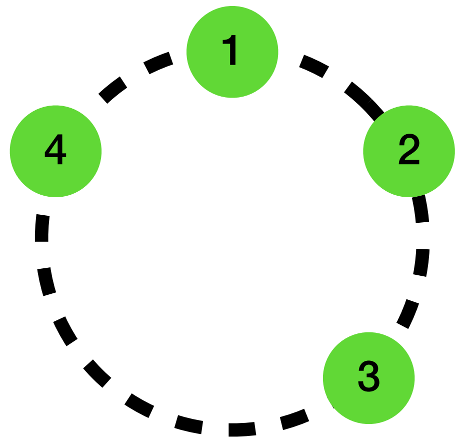
The client can initiate an HTTP call by

- (1) via a load balancer

  + the client is unaware of any dynamo logic

  - more latency as another forwarding step may be required

  (if the reached node is NOT part of the top N nodes in the preference list)



- (2) via a partition aware client driver

  + lower latency

  - client need to maintain the logic / sync with the ring nodes

`get()` and `put()` execution (2)

Consistency

- Dynamo uses a quorum protocol
  just like the one we saw in the CAP theorem

- **N** #nodes that store replicas of the data
- **W** #replicas that need to acknowledge the receipt of the update
- **R** #replicas that are contacted for a read

$$W + R > N$$

(2,2,3 is a common setting)

`get()` and `put()` execution (3)

For `put()` the coordinator

- Writes the data + the new vector clock locally
- Send it to N-1 nodes from the preference list
- Waits for W-1 to return success

In a failure free environment

For `get()` the coordinator

- Request all versions from the N-1 nodes in the preference list
- Wait for R response to return success
  if more than 1 version returned, return all versions for the client to reconcile

Dynamo topics

- Requirements
- Partition algorithm
- Replication
- Data versioning
- `get()` and `put()` execution
- **Failures**
- Ring membership

Failures

- Temporary (from milliseconds to 3 hours)
- Permanet

Failures - Temporary (1)

- In a cloud environment there are (possibly) frequent temporal errors
  network partitions, vm fails, power…
- Temporal = from seconds to minutes (3 hours max)
- Can easily cause an availability issue (“strict quorum”)
  can you think of an example?

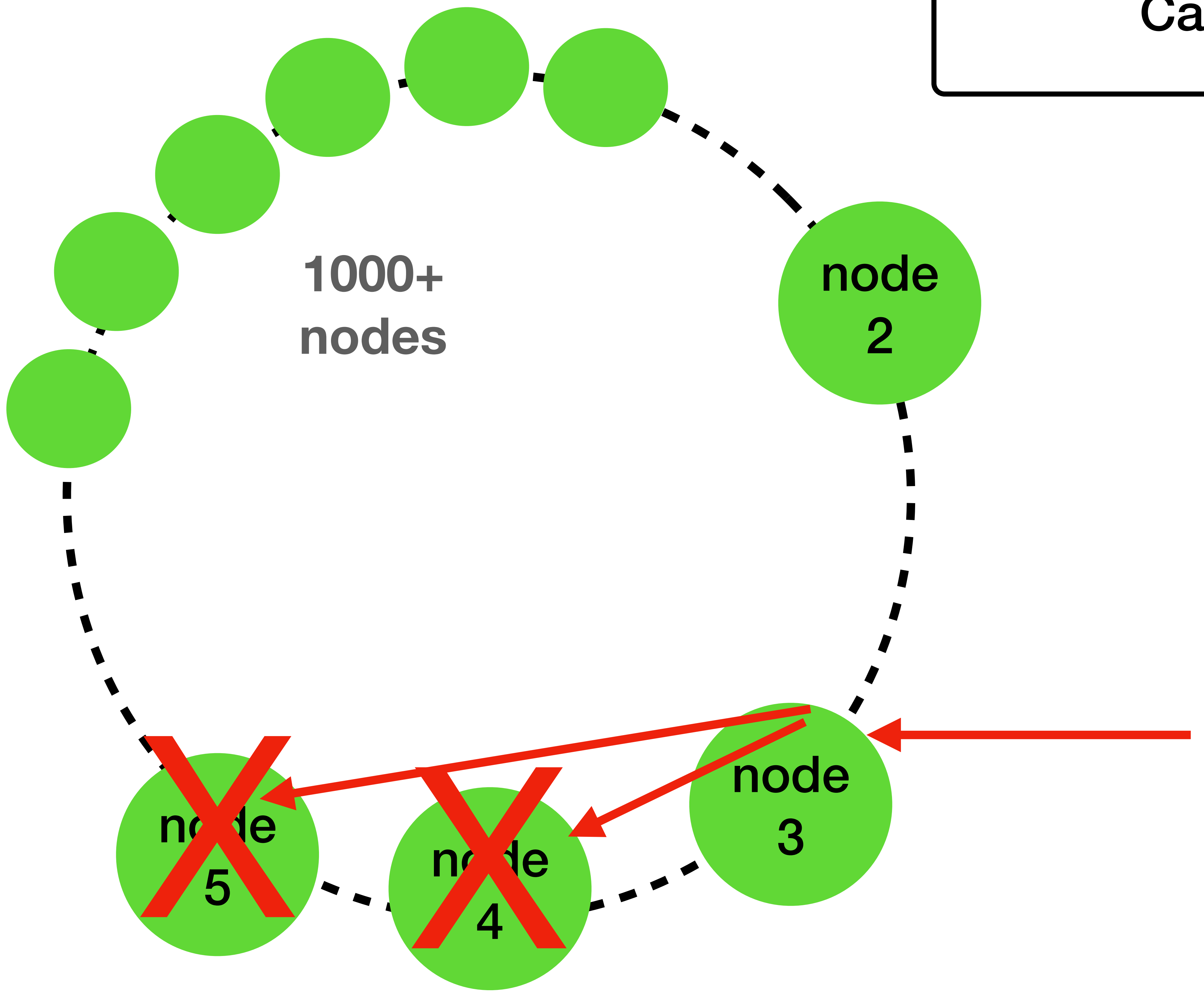
Strict = the nodes which are ″mapped″ to store the data

Failures - Temporary (1)

Strict quorum “problem”

If node4 and node5 are down, we can NOT complete the write.

Can we use the other nodes?

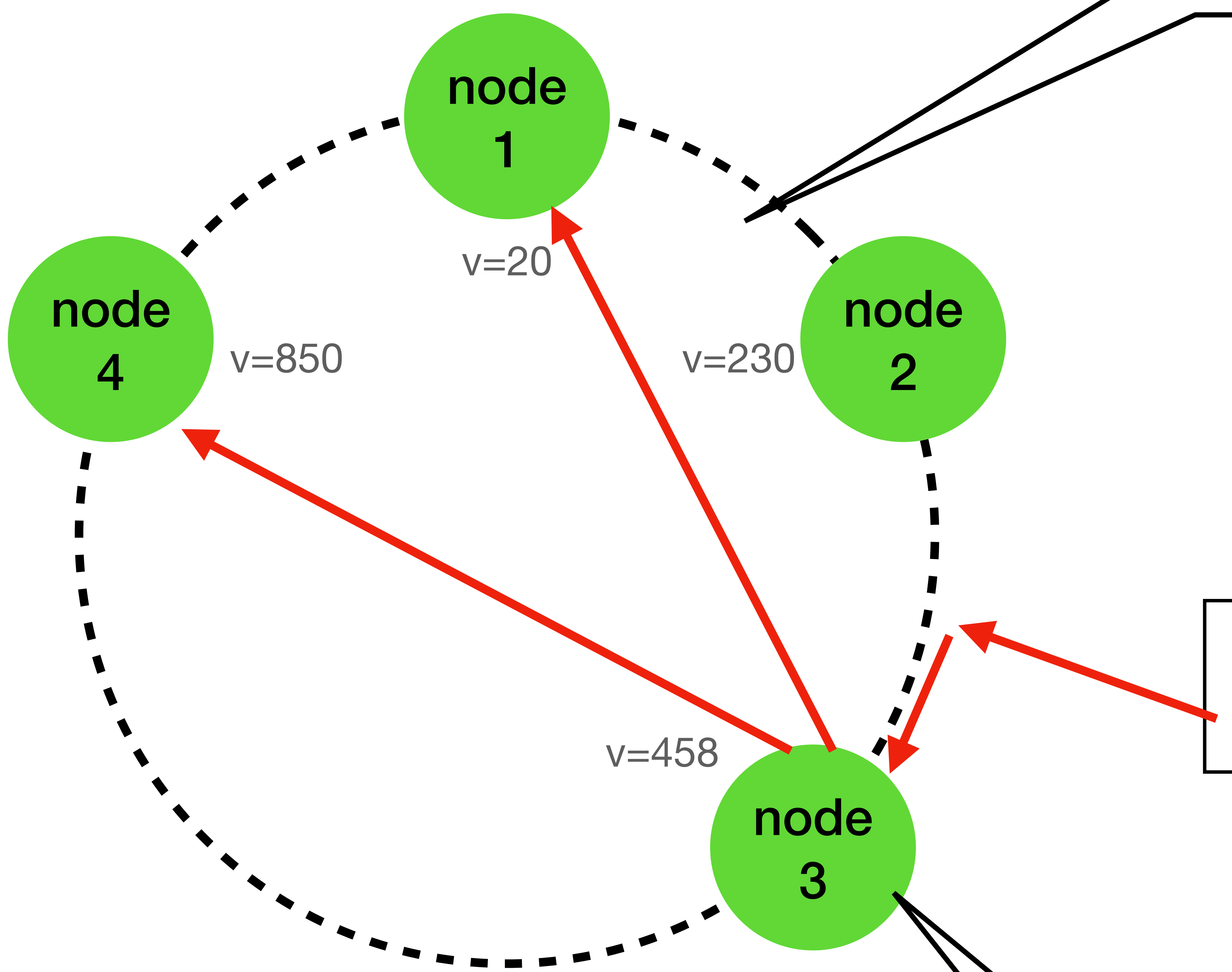


Failures - Temporary (2)

Hinted handoff

- Sloppy quorum - all reads/writes are performed on the first N healthy nodes from the preference list
  may not be the first N nodes if some fail
- On nodes failures, we use the next nodes (on the ring) as replicas and store an additional “hint” on the metadata
  suggesting which node was originally intended to be written
- These hinted handoffs will be stored on a separate local list, and will be used to update the failed nodes once are back online

Failures - Temporary (3) - example



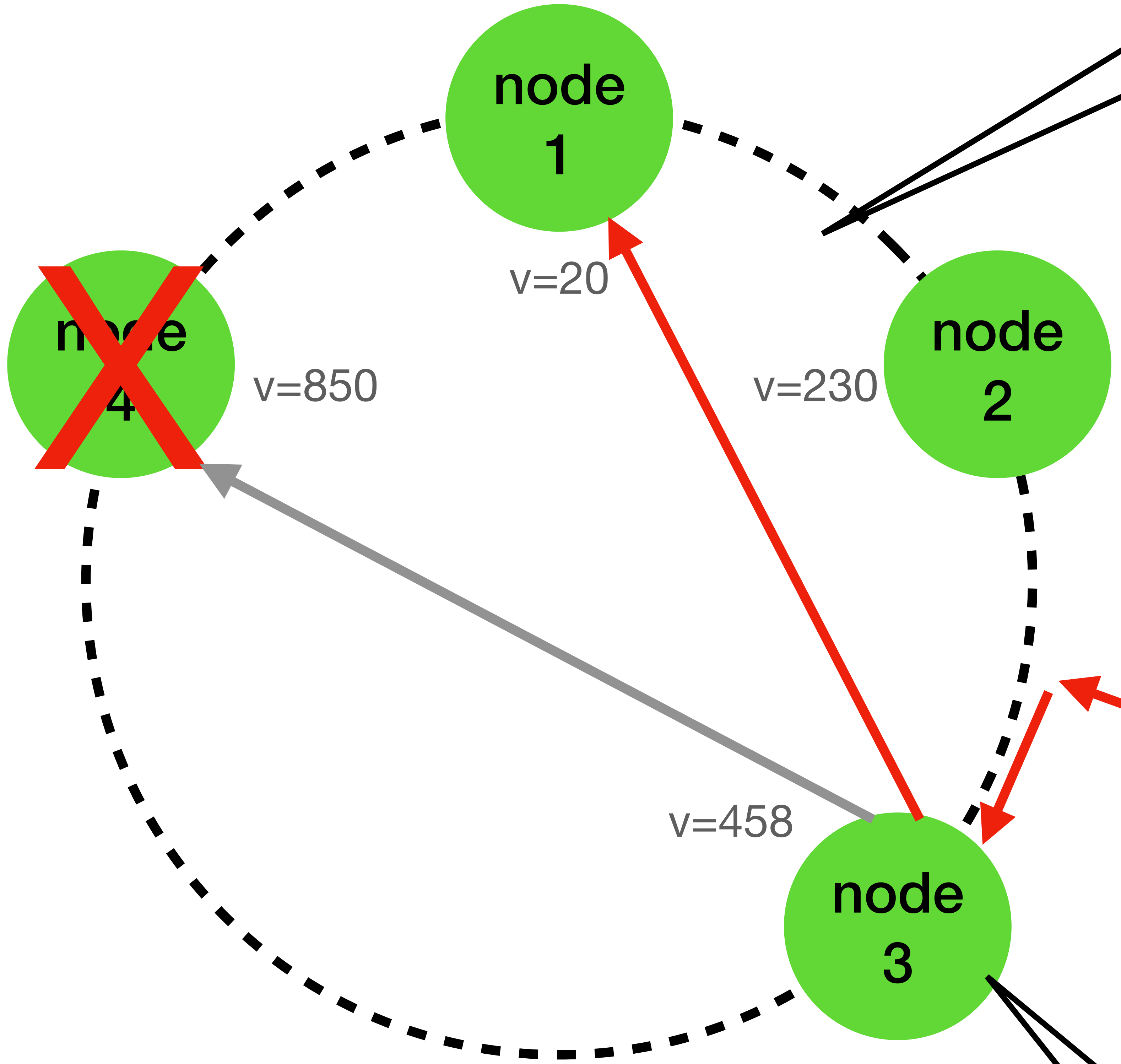
Range is [0:1000] for the example, N=3

```
hash(key) = 344
```

(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

Failures - Temporary (3) - example



Range is [0:1000] for the example, N=3

hash(key) = 344

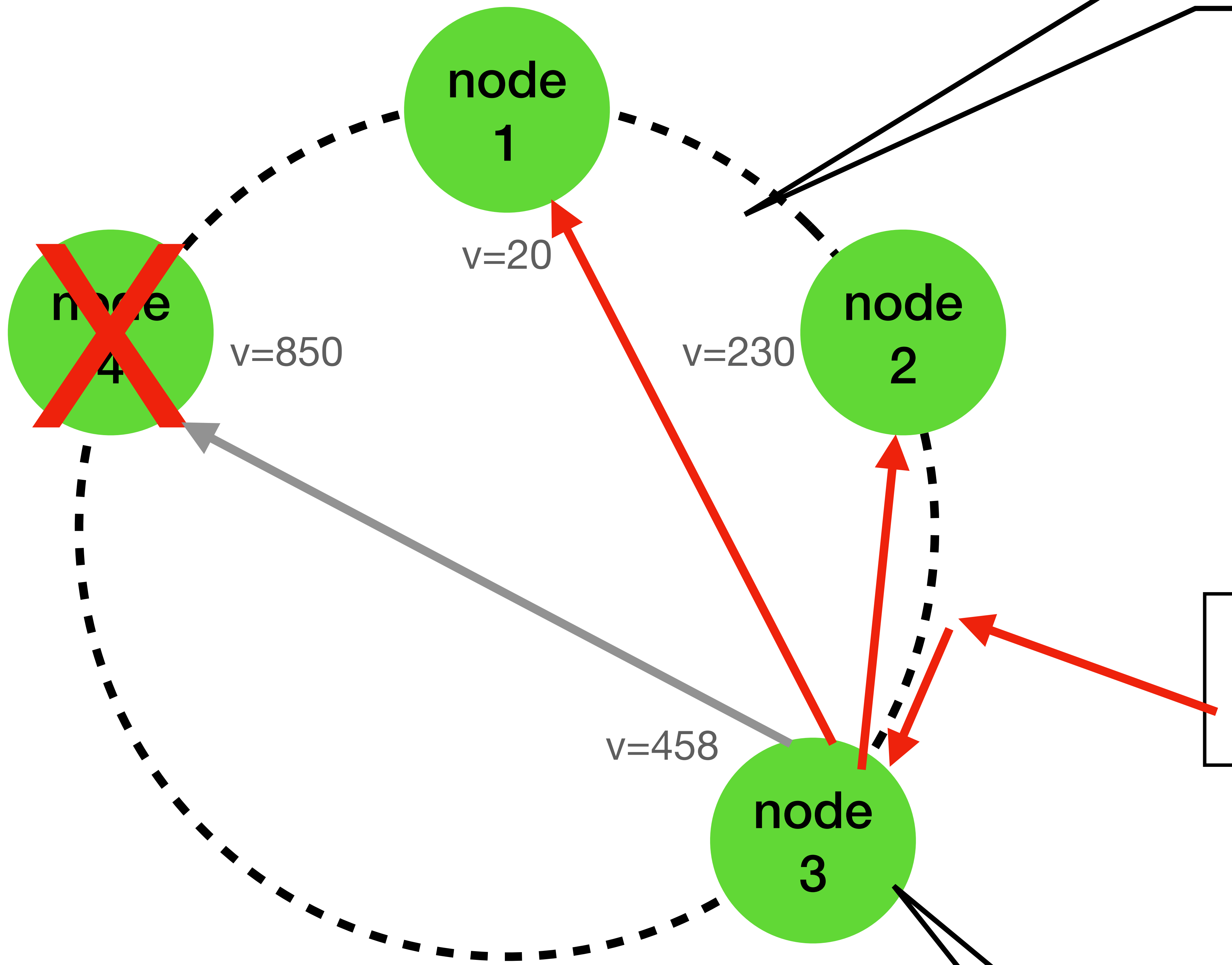
(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

Failures - Temporary (3) - example

Range is [0:1000] for the example, N=3

Assume node4 fails. To maintain durability (N=3), we write on node2

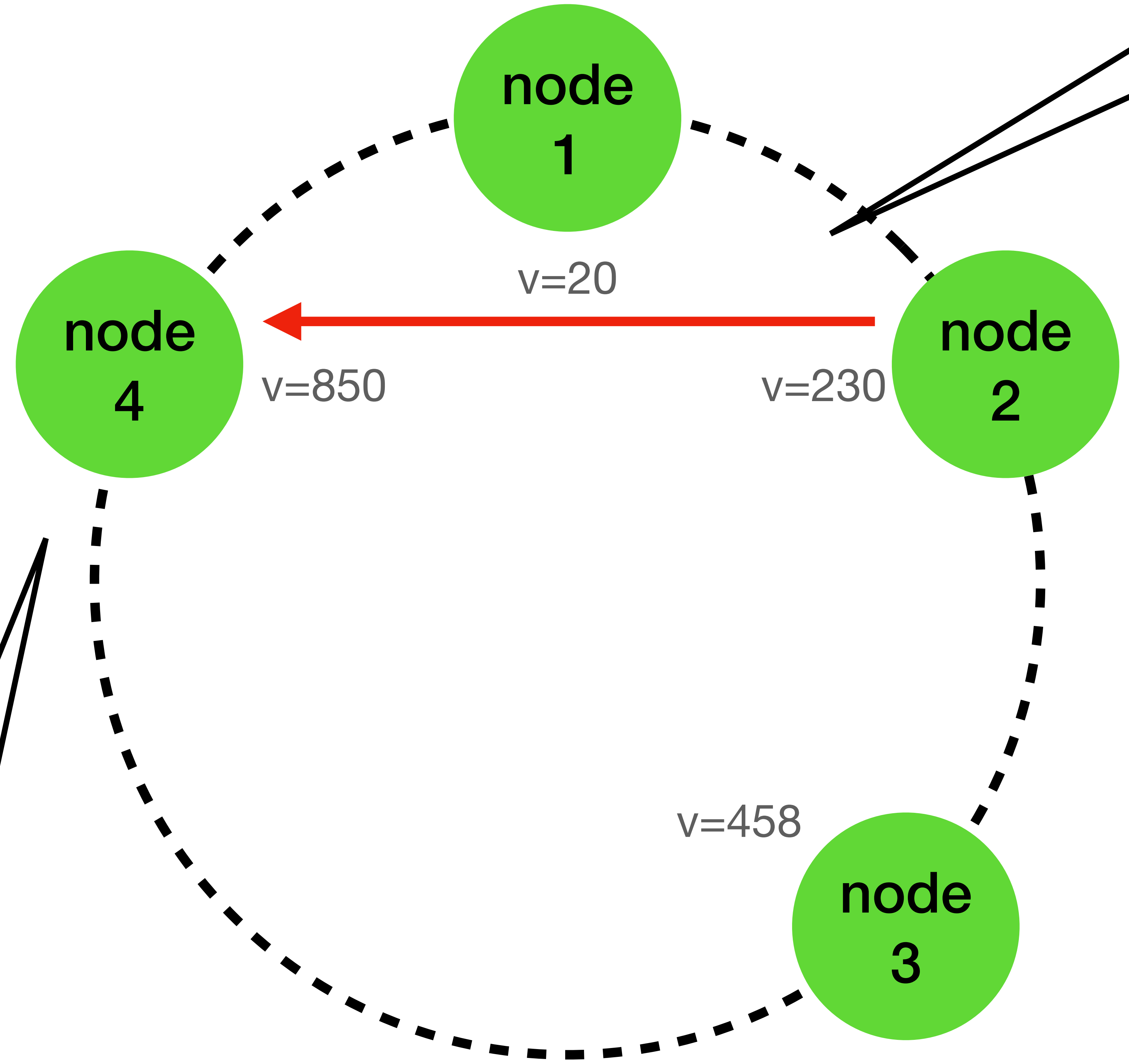


```
hash(key) = 344
```

(assigned to node3)

node3 is the coordinator. It will store the value locally and on node4 and node1

Failures - Temporary (3) - example



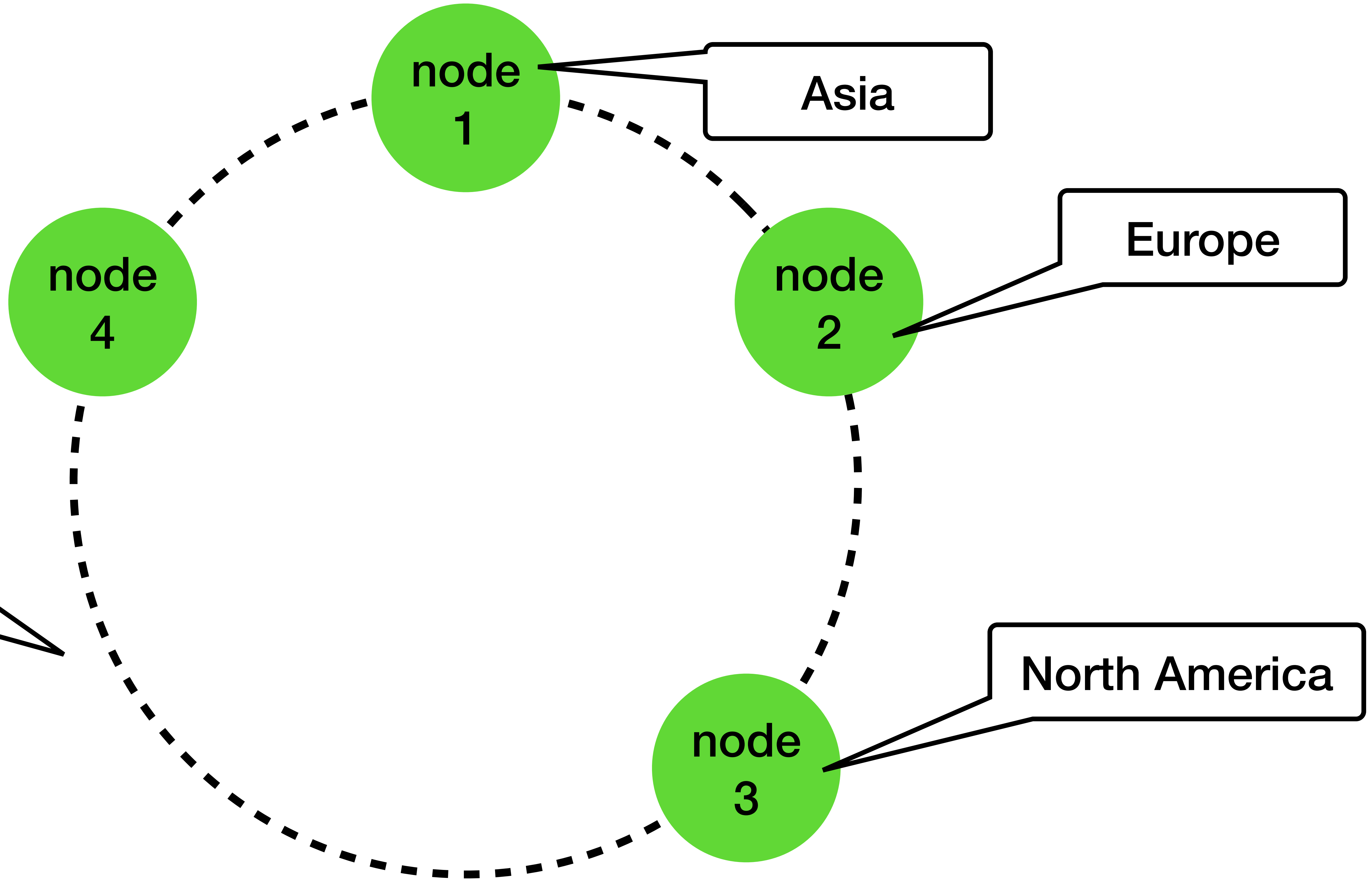
Range is [0:1000] for the example, N=3

Assume node4 fails. To maintain durability (N=3), we write on node2

Once node4 resumes, node2 will update node4 and delete the data

Failures - Temporary (4)

- It is crucial for an highly available system to be able of handing the failure of an entire data center
  power outages, cooling/network failures, natural disasters…



Dynamo can be configured such that the preference list is spread among different data centers

Failures - Permanent (1)

Hinted handoff works best when

- Node failures are transient
- System membership churn is low

What to do when

- The node with the hinted replicas fails
- Other durability threats

Failures - Permanent (2)

Anti entropy (replica synchronization)

- A protocol to keep replicas synchronized

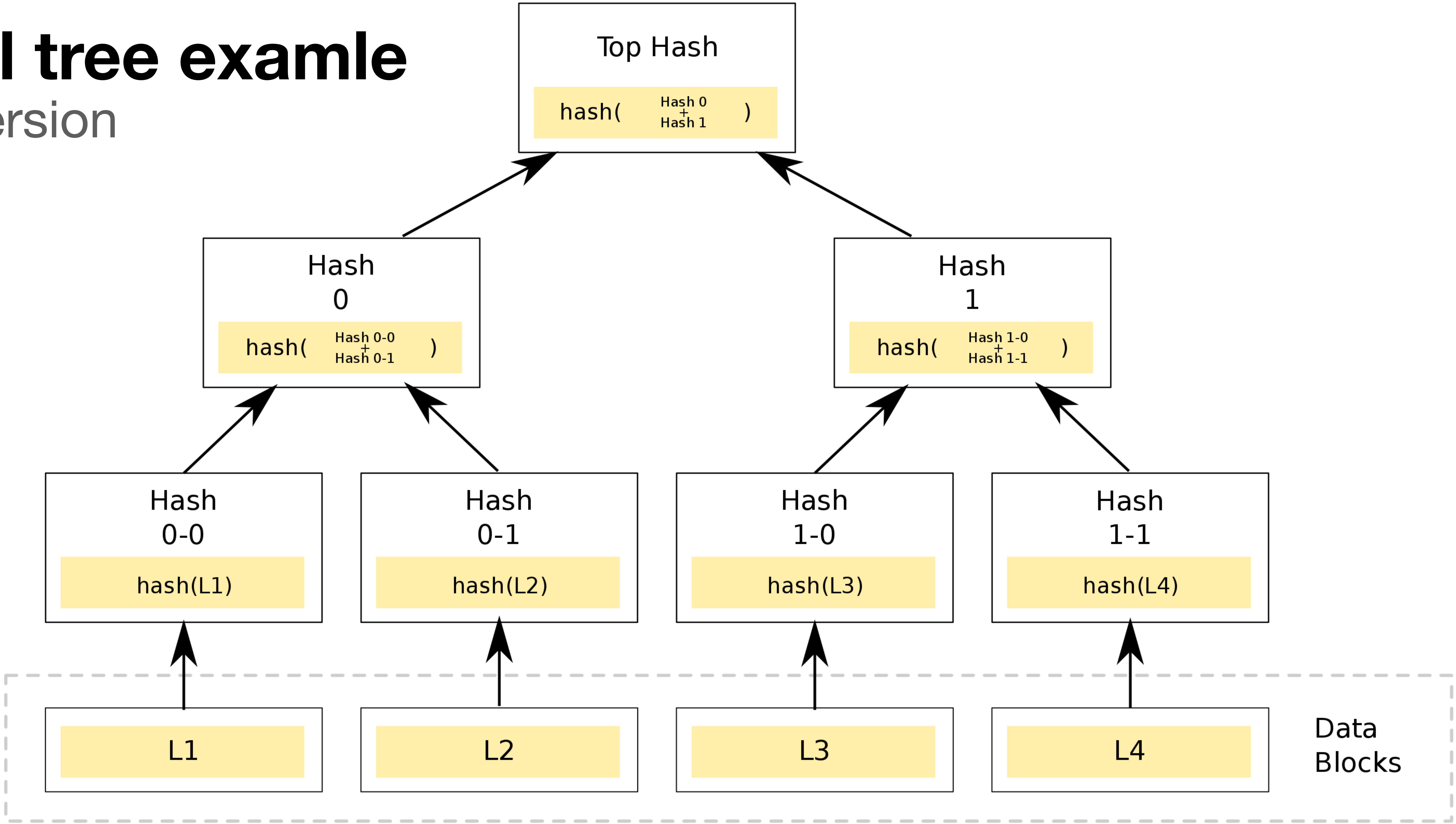
To detect inconsistencies between replicas and to minimize the amount of transferred data, Dynamo uses Merkle trees:

A Merkle tree is a hash tree where leaves are hashes of the values of individual keys. Parent nodes higher in the tree are hashes of their respective children

Failures - Permanent (3)

Merkel tree examle

binary version



Failures - Permanent (4)

Dynamo uses Merkel tree as follows

- Each node maintain a separate Merkel tree for each key range
  the set of keys covered by a virtual node
- Nodes can compare each matching range by exchanging the matching tree roots
- On “out of sync” - nodes can exchange only the subset of their children to avoid transmitting all data

Dynamo topics

- Requirements
- Partition algorithm
- Replication
- Data versioning
- `get()` and `put()` execution
- Failures
- **Ring membership**

Ring membership

Assumption

- Node outages are often transient
- Permanent departures are rare
  - —> do not automatically rebalanced the ring when (temporal) error occurs

- To add / remove nodes (which rebalance the ring) use an explicit mechanism (via API)

Ring membership - Gossip protocol

- Recall we do not have a master node (fully distributed)
- When a node is added/removed (and thus the ring changes), a gossip based protocol is used to update the ring status
  —> eventually consistent view of the ring
- **Gossip protocol**: every second each node contact a random different node and the two nodes “reconcile” their ring membership view
  also used for other Dynamo needs

Ring membership - Failure detection (1)

- Used to avoid communicating with unreachable nodes
  during get() and put()

Local notion of failure (decentralized)

- Node A may consider node B failed
  if B does not response to A’s message
- But node C can consider node B alive
  if B is responsive to C’s message

Ring membership - Failure detection (2)

- Under normal operation, Node A can quickly discover that node B is unresponsive when B fails to respond to a message
  derived from `put()` / `get()` calls

- A periodically retires to B are made to check for B's recovery

- If 2 nodes are not "near" in the ring, neither needs to know whether the other is reachable and responsive

Dynamo topics

- Requirements
- Partition algorithm
- Replication
- Data versioning
- `get()` and `put()` execution
- Failures
- Ring membership