GridGain

Architects' Guide for Apache Ignite ACID Transactions and Consistency

Ivan Rakov

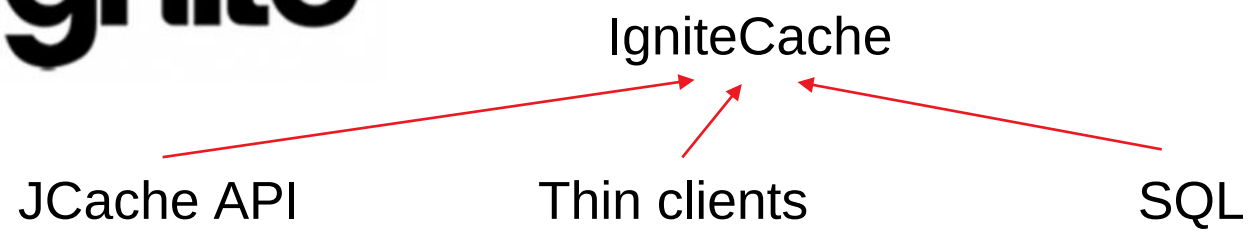
April 29, 2020

April 29, 2020

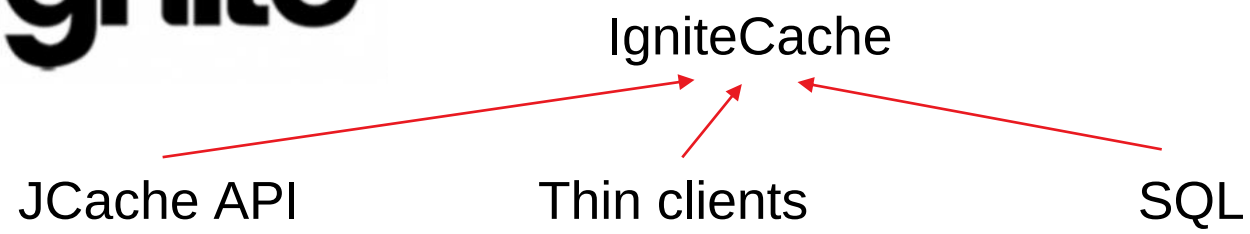
Ivan Rakov

- Work at GridGain Systems
  - Leading data consistency dev team
- Apache Ignite Committer

Stores your data in a distributed way

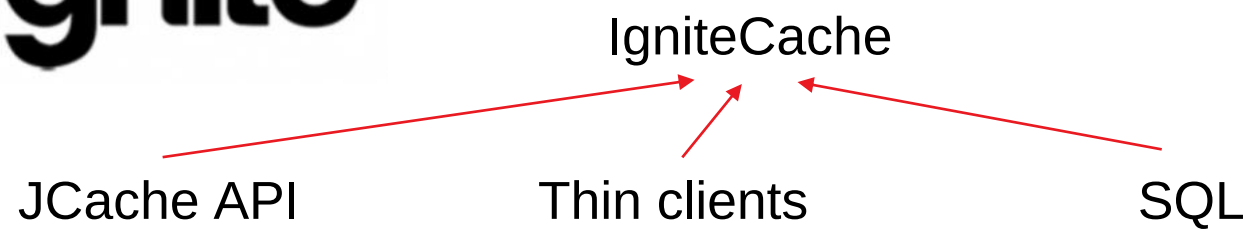


Stores your data in a distributed way



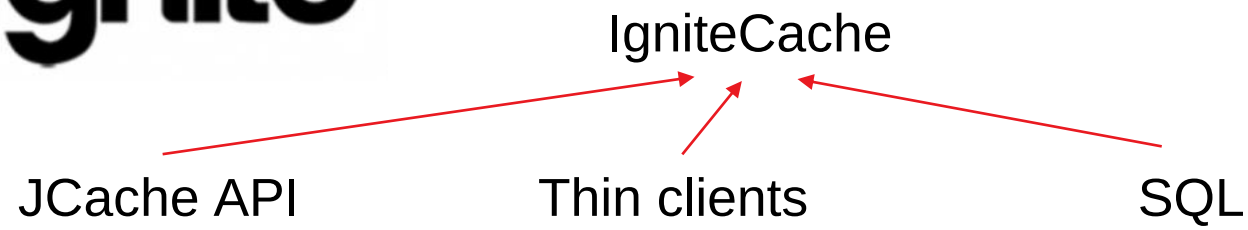
- Trade-off: data safety / consistency vs. performance

Stores your data in a distributed way



- Trade-off: data safety / consistency vs. performance
- Decision should depend on the use case
  - Caching for external storage
  - Reliable (K, V) storage
  - Business-critical transactions processing

Stores your data in a distributed way



- Trade-off: data safety / consistency vs. performance
- Decision should depend on the use case
  - Caching for external storage
  - Reliable (K, V) storage
  - Business-critical transactions processing
- Flexible Ignite configuration allows to adapt for every case

Agenda

The most important configuration settings

- Data replication modes PARTITIONED, REPLICATED

Agenda

The most important configuration settings

- Data replication modes PARTITIONED, REPLICATED
- Data sync guarantees

Agenda

The most important configuration settings

- Data replication modes PARTITIONED, REPLICATED
- Data sync guarantees
- Data consistency

Agenda

The most important configuration settings

- Data replication modes PARTITIONED, REPLICATED
- Data sync guarantees
- Data consistency
- Data storage
  - In-memory / disk
  - Capacity
  - Disk-based consistency

Agenda

The most important configuration settings

- Data replication modes
- Data sync guarantees
- Data consistency
- Data storage
  - In-memory / disk
  - Capacity
  - Disk-based consistency

Data Replication

- cacheConfiguration.setCacheMode(mode);

Data Replication

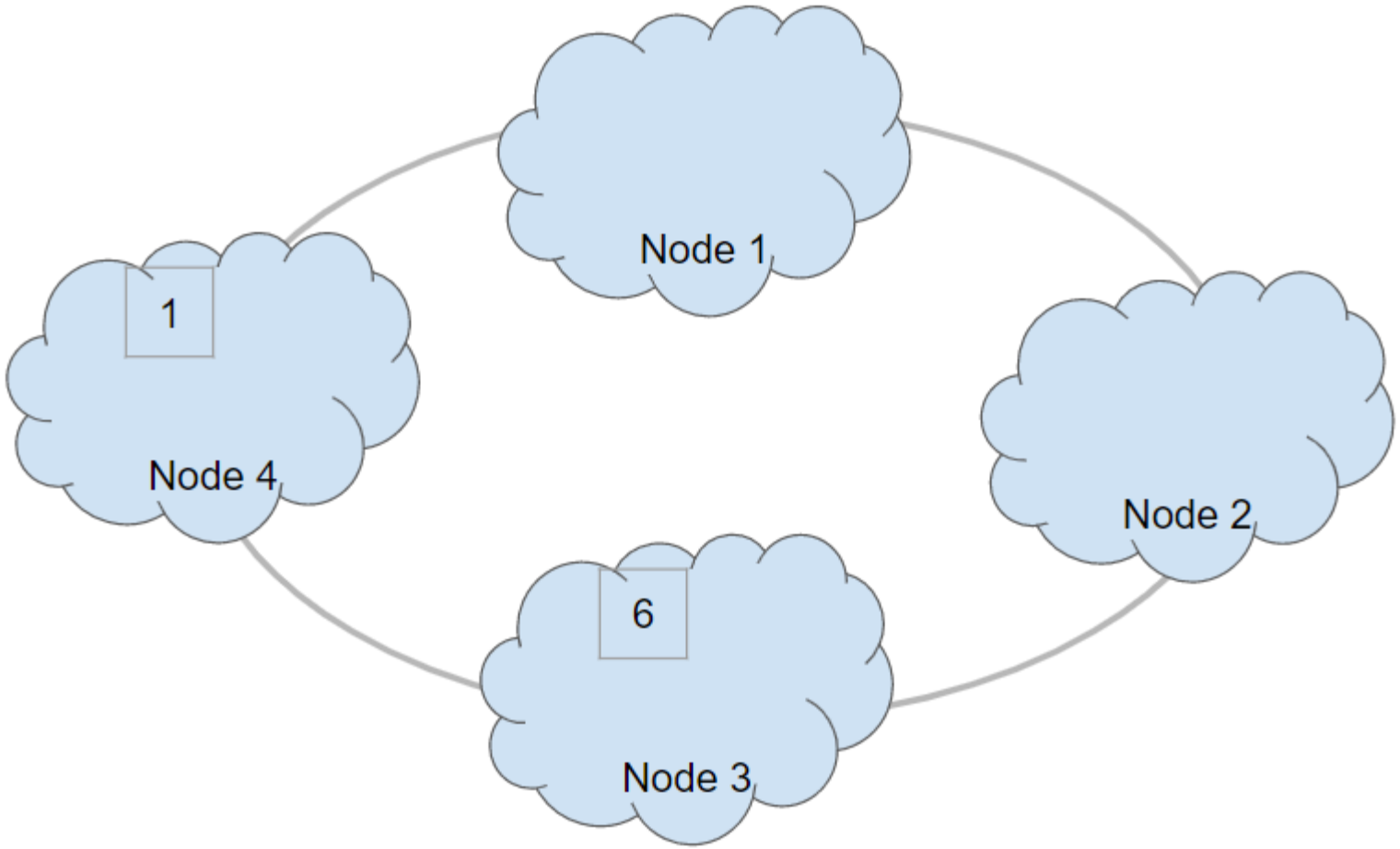
- cacheConfiguration.setCacheMode(mode);
  - PARTITIONED
    - Data is partitioned
    - Number of copies for every partition can be specified

Data Replication

- cacheConfiguration.setCacheMode(mode);
  - PARTITIONED
    - Data is partitioned
    - Number of copies for every partition can be specified
  - REPLICATED
    - Every node keeps whole data set

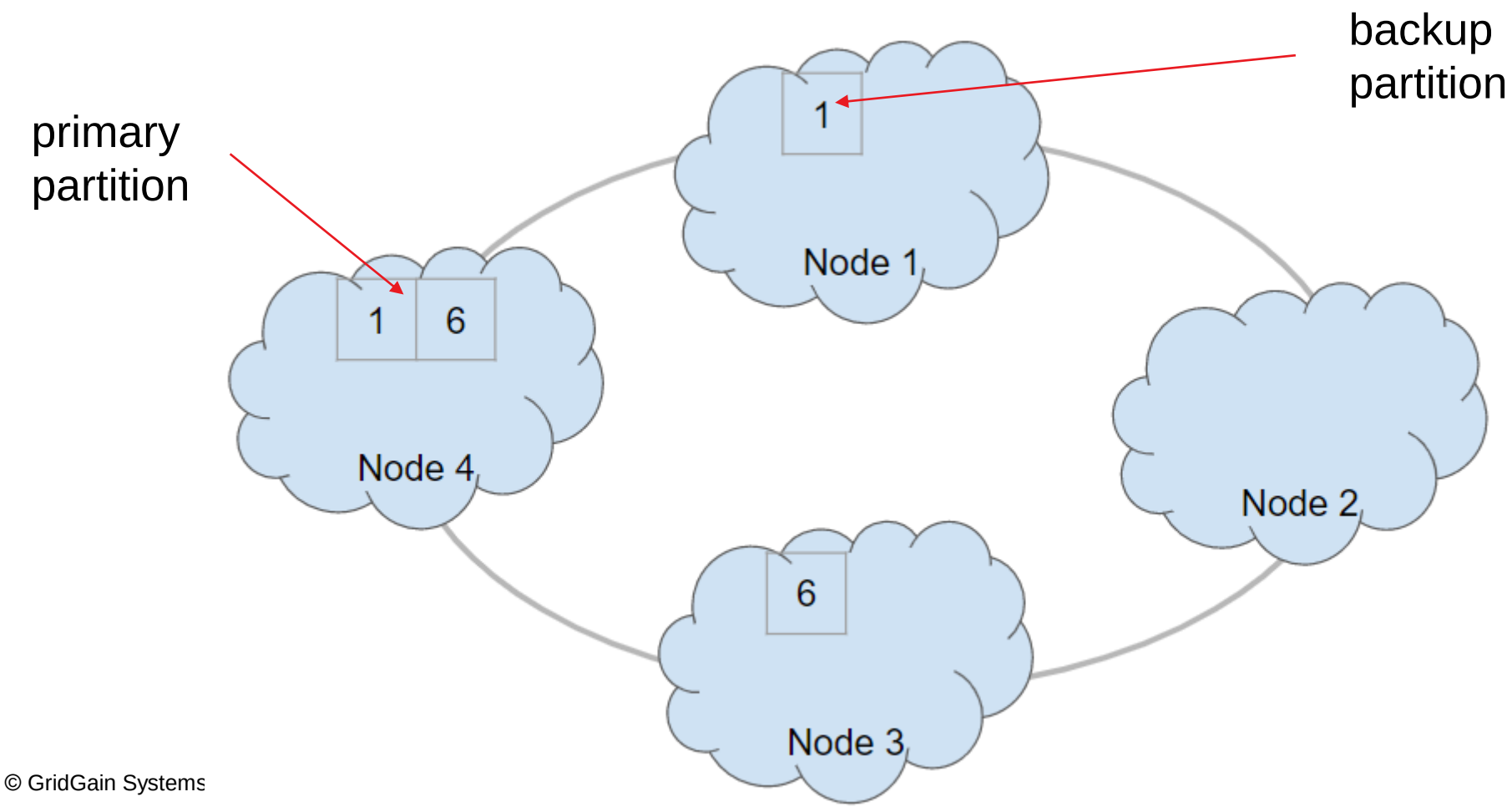
Partitioned Cache

- By default, every partition is present in one copy

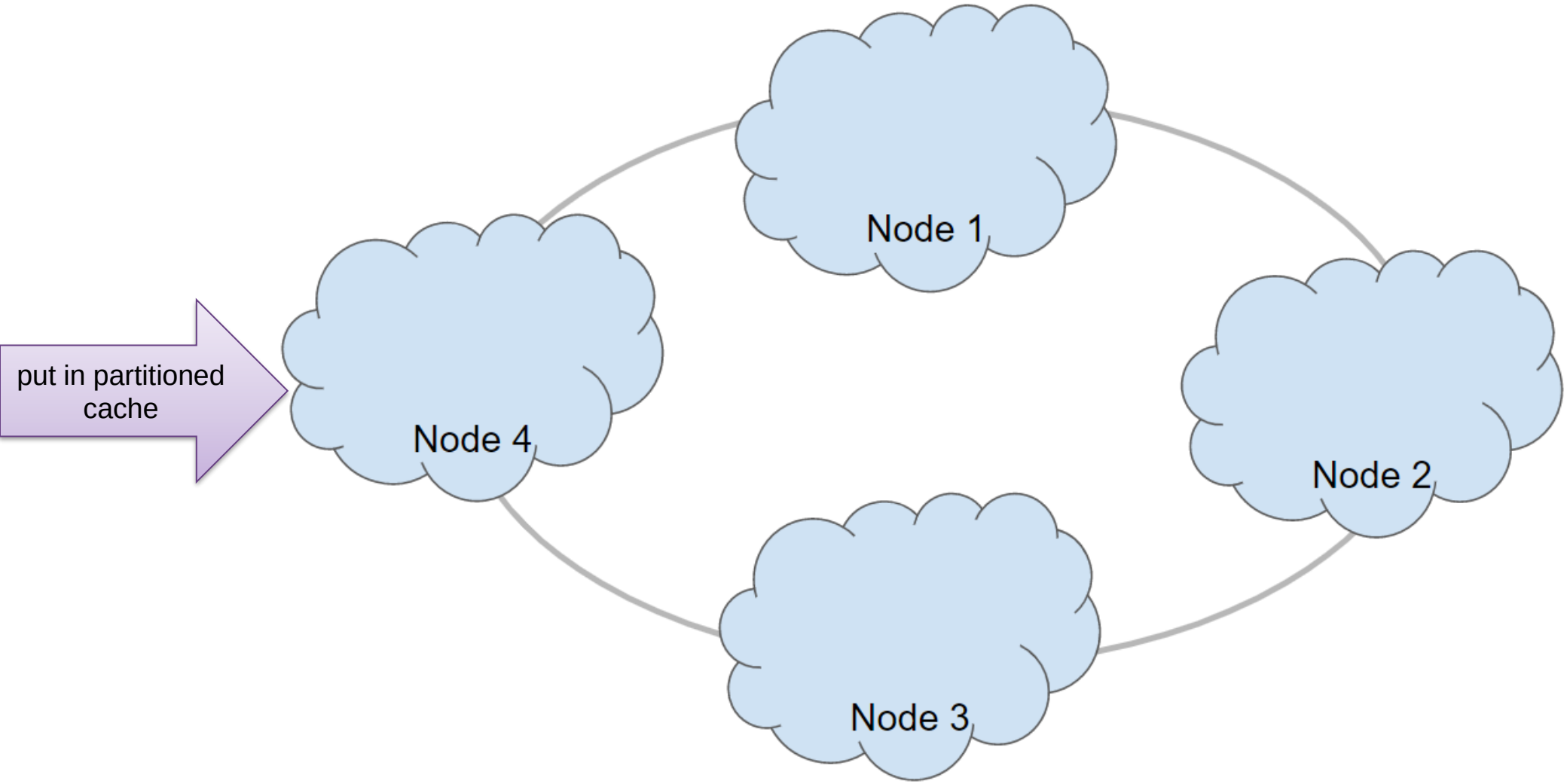


Partitioned Cache

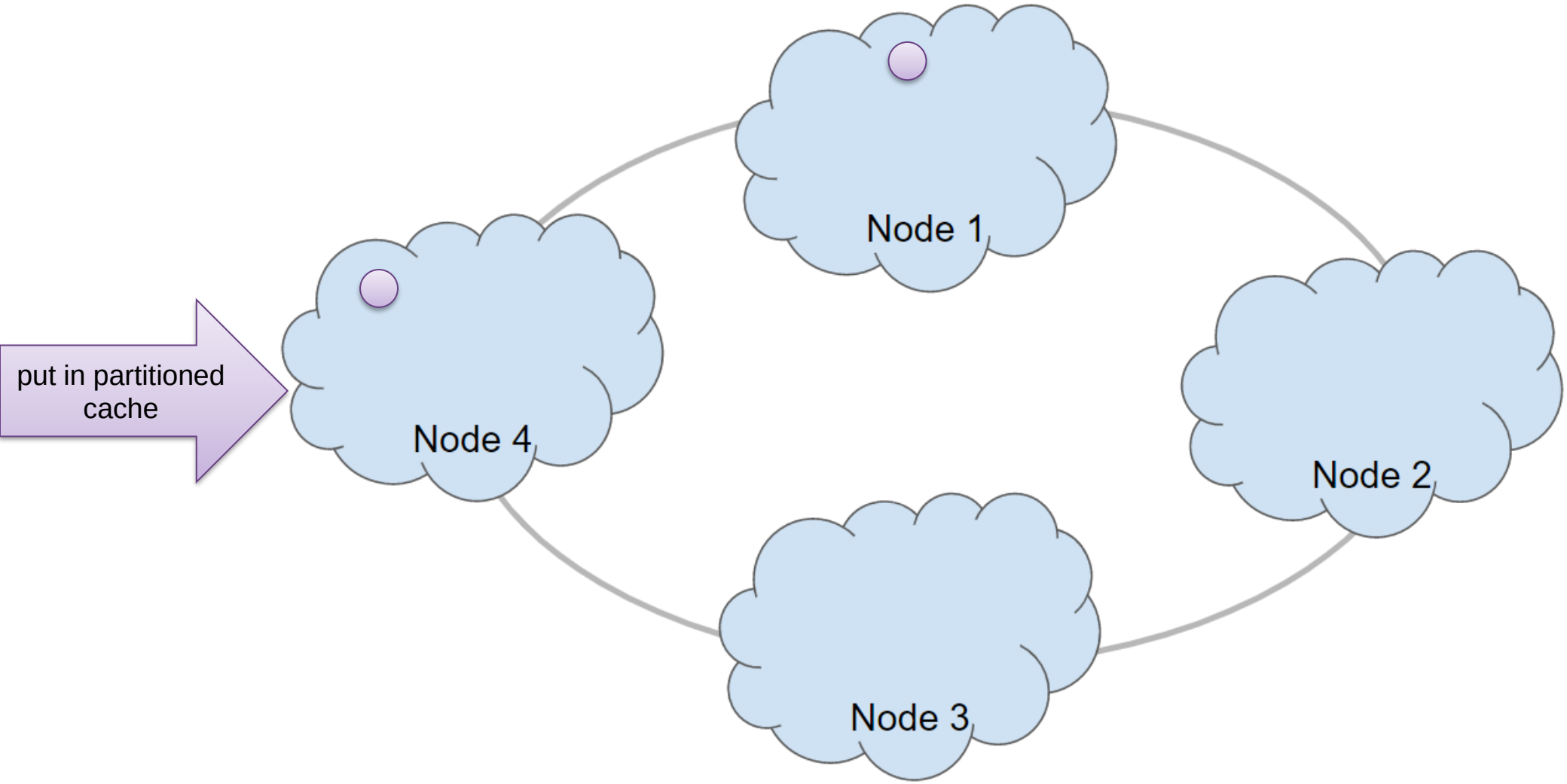
- cacheConfiguration.setBackups(1);



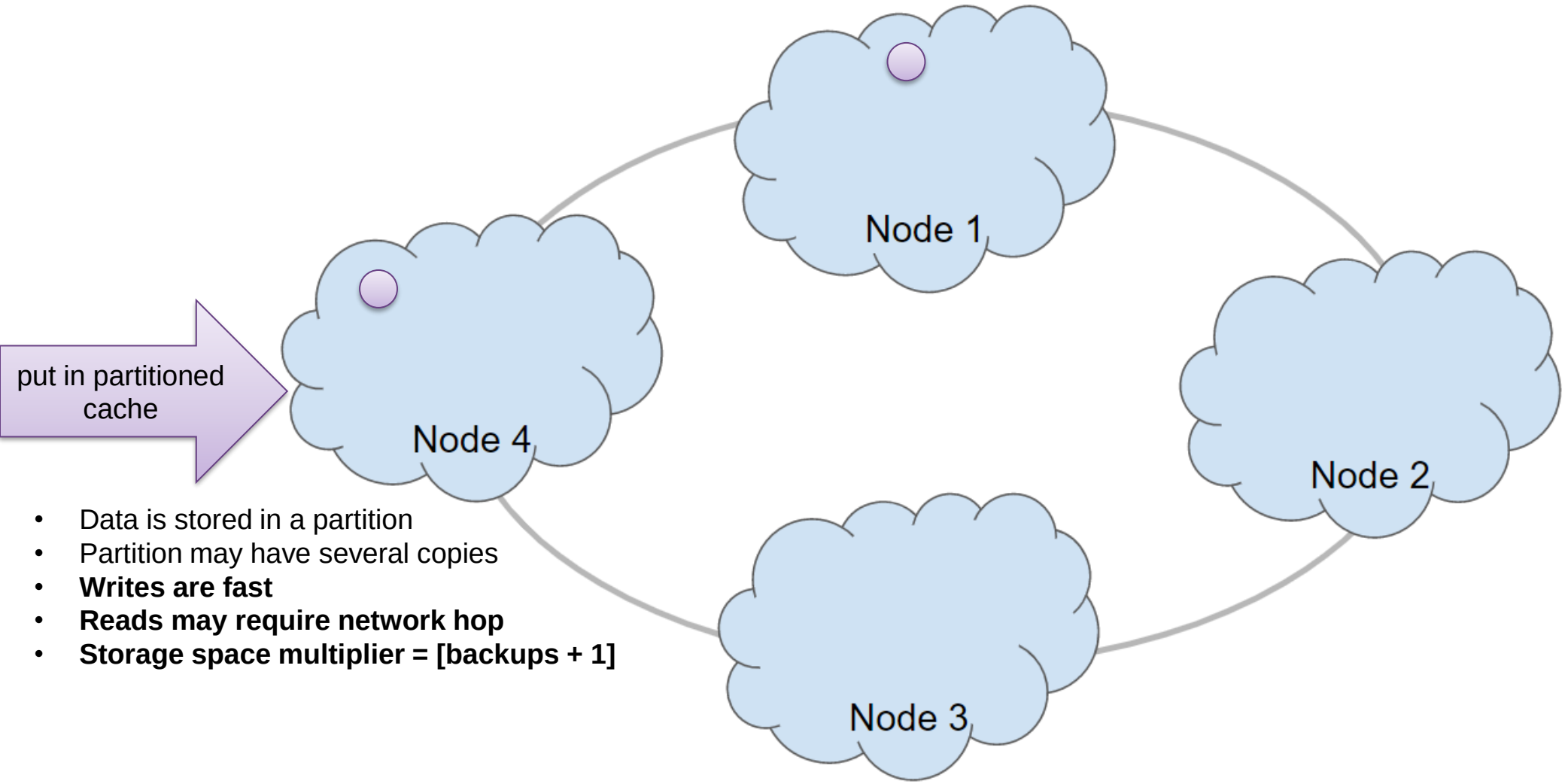
Partitioned vs Replicated Cache



Partitioned vs Replicated Cache

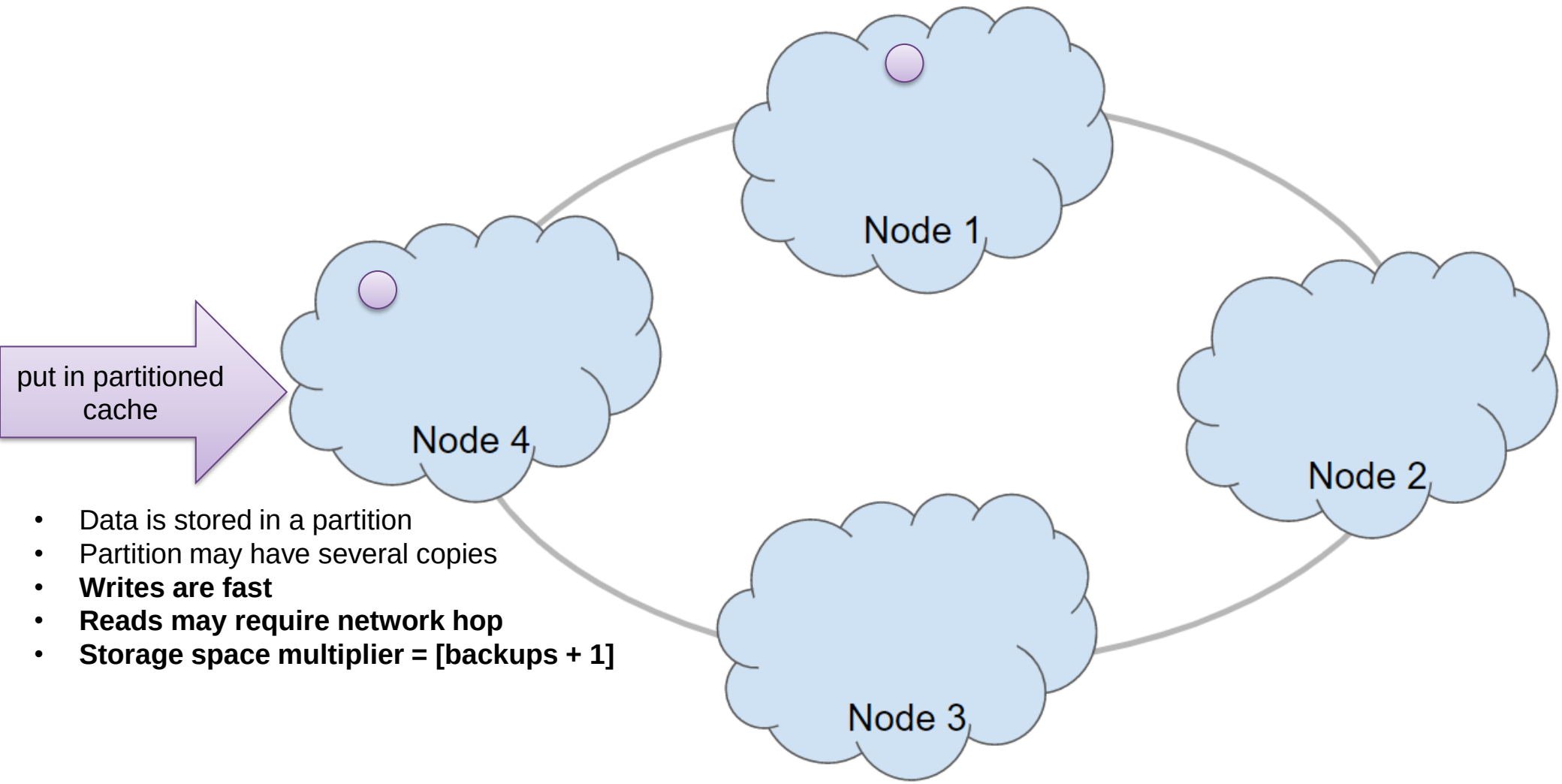


Partitioned vs Replicated Cache



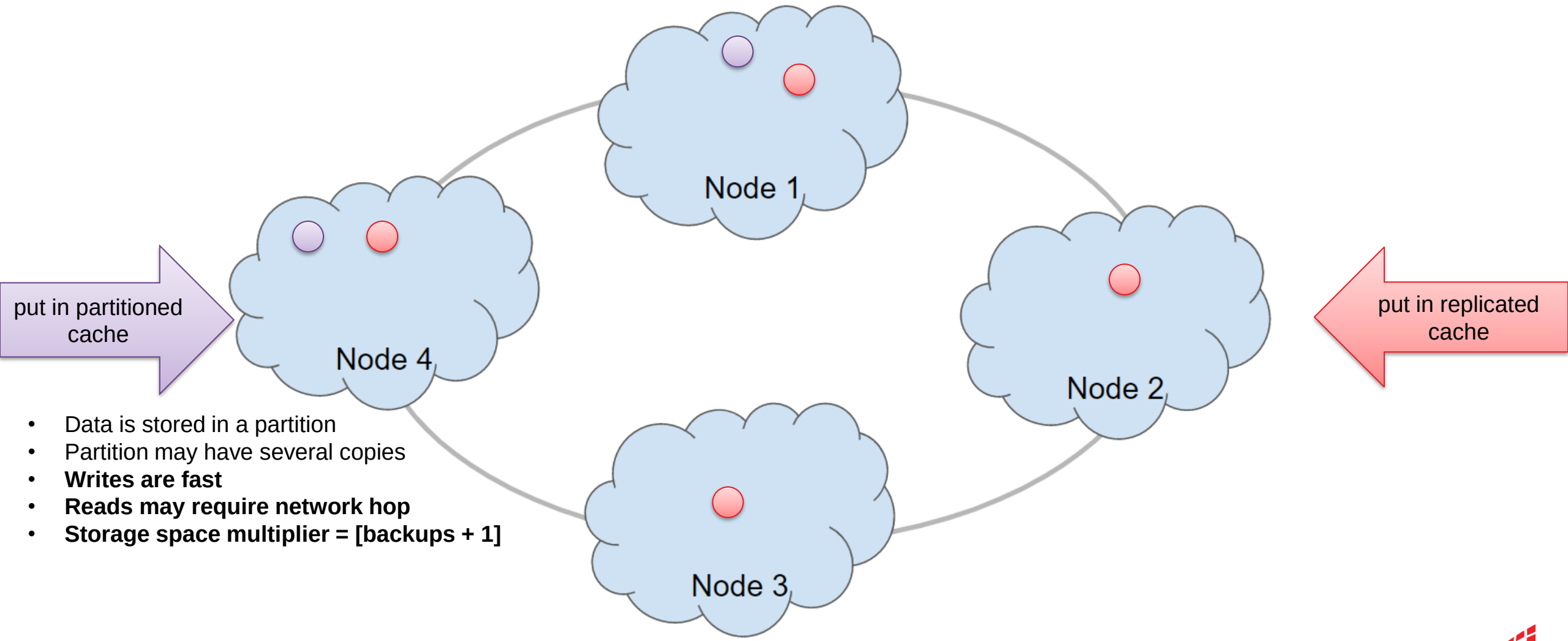
- Data is stored in a partition
- Partition may have several copies
- **Writes are fast**
- **Reads may require network hop**
- **Storage space multiplier = [backups + 1]**

Partitioned vs Replicated Cache



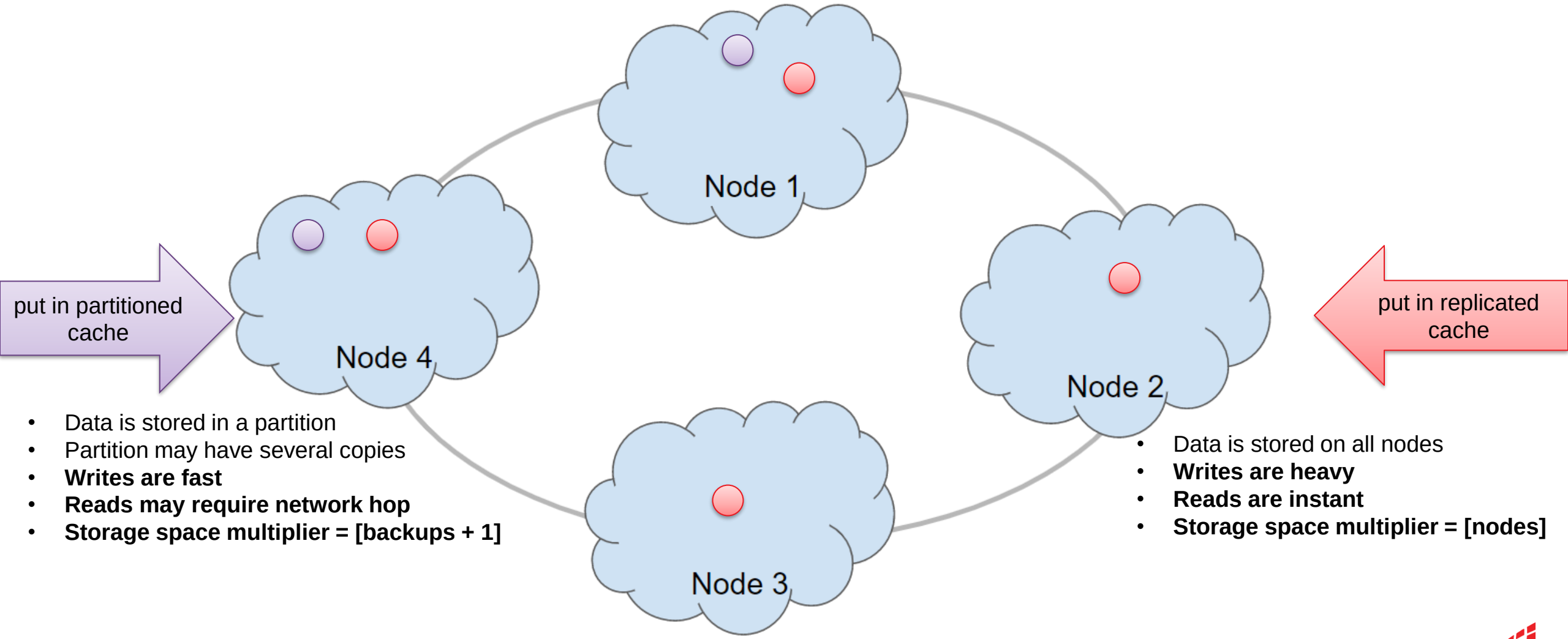
- Data is stored in a partition
- Partition may have several copies
- **Writes are fast**
- **Reads may require network hop**
- **Storage space multiplier = [backups + 1]**

Partitioned vs Replicated Cache



- Data is stored in a partition
- Partition may have several copies
- **Writes are fast**
- **Reads may require network hop**
- **Storage space multiplier = [backups + 1]**

Partitioned vs Replicated Cache



- Data is stored in a partition
- Partition may have several copies
- **Writes are fast**
- **Reads may require network hop**
- **Storage space multiplier = [backups + 1]**

- Data is stored on all nodes
- **Writes are heavy**
- **Reads are instant**
- **Storage space multiplier = [nodes]**

Agenda

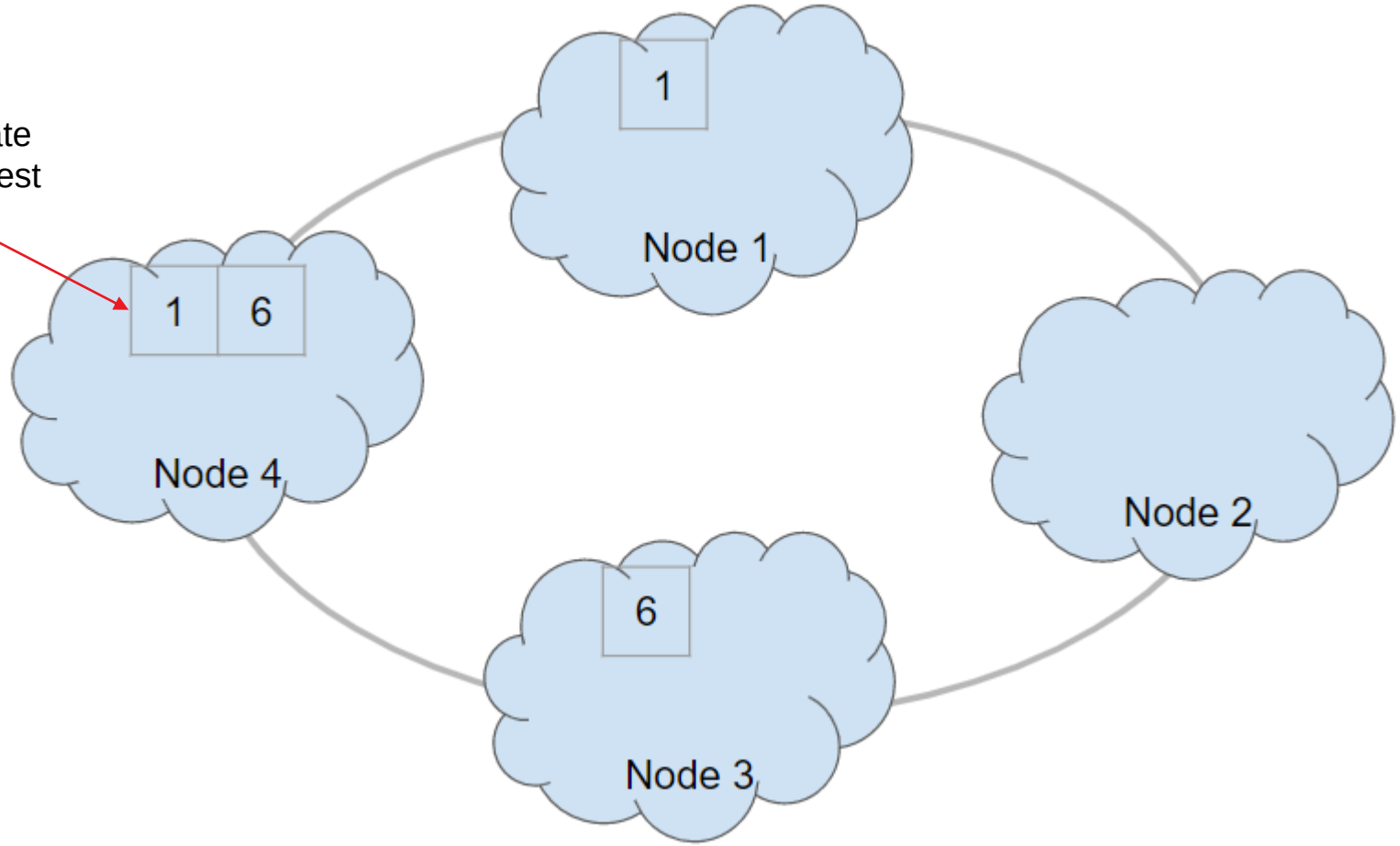
The most important configuration settings

- Data replication modes
- Data sync guarantees
- Data consistency
- Data storage
  - In-memory / disk
  - Capacity
  - Disk-based consistency

Data Sync Guarantees

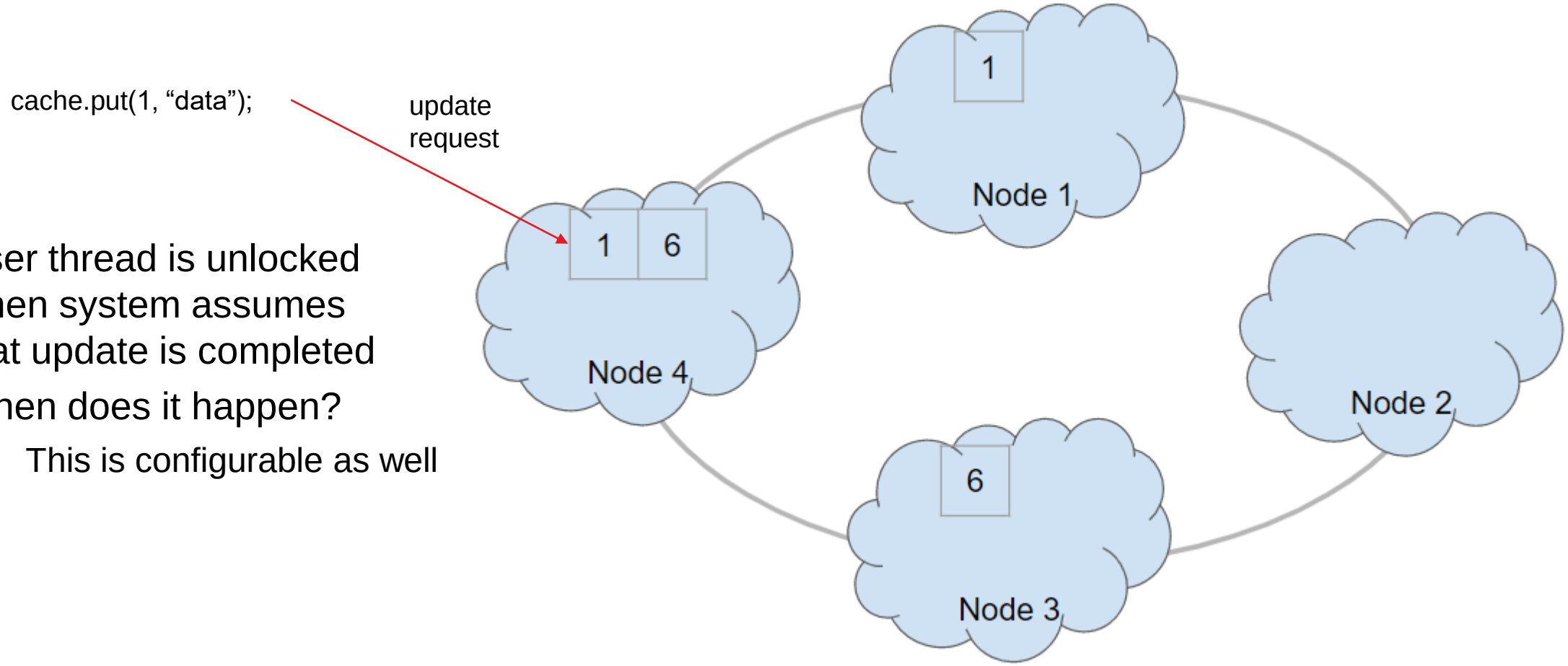
cache.put(1, “data”);

update request



- User thread is unlocked when system assumes that update is completed

Data Sync Guarantees



- User thread is unlocked when system assumes that update is completed
- When does it happen?
  - This is configurable as well

Data Sync Guarantees

- cacheConfiguration.setWriteSynchronizationMode(mode);

Data Sync Guarantees

- cacheConfiguration.setWriteSynchronizationMode(mode);
- PRIMARY_SYNC
  - Client will wait for data update completion on primary node

Data Sync Guarantees

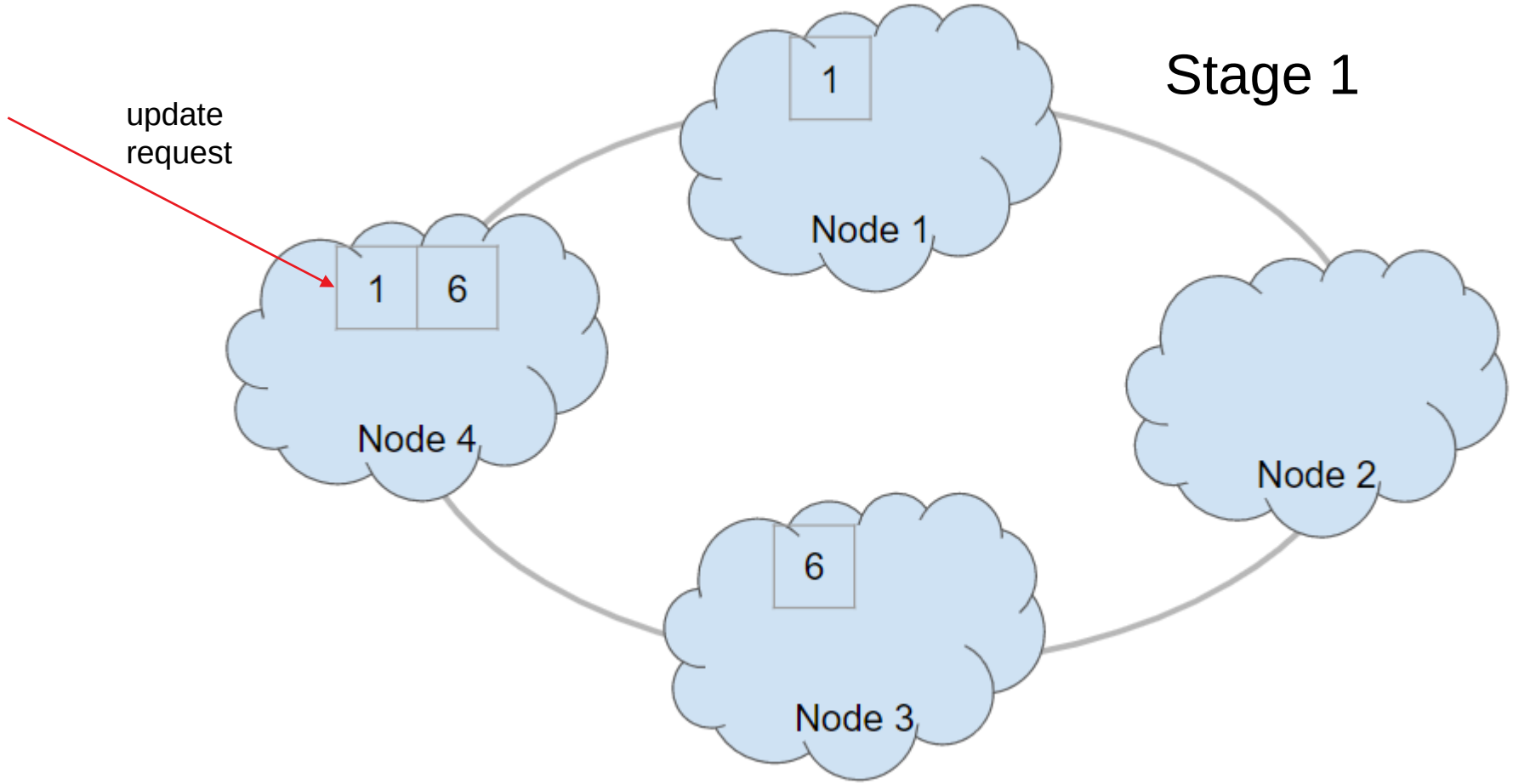
- cacheConfiguration.setWriteSynchronizationMode(mode);
- PRIMARY_SYNC
  - Client will wait for data update completion on primary node
- FULL_SYNC
  - Client will wait for data update completion on all participating nodes

Data Sync Guarantees

- cacheConfiguration.setWriteSynchronizationMode(mode);
- PRIMARY_SYNC
  - Client will wait for data update completion on primary node
- FULL_SYNC
  - Client will wait for data update completion on all participating nodes
- FULL_ASYNC
  - Client doesn’t wait for data update completion

Data Sync Guarantees

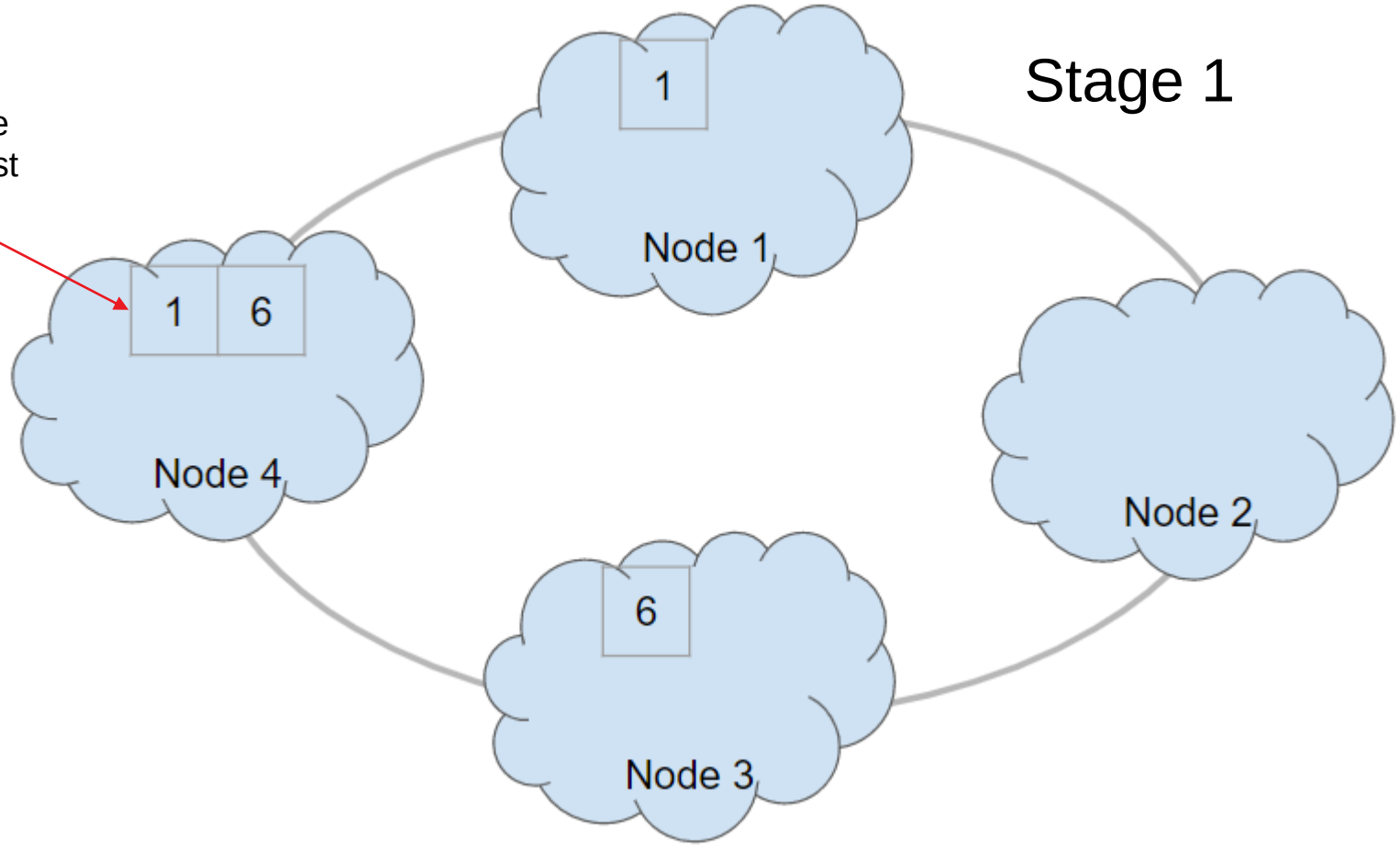
cache.put(1, “data”);



Data Sync Guarantees

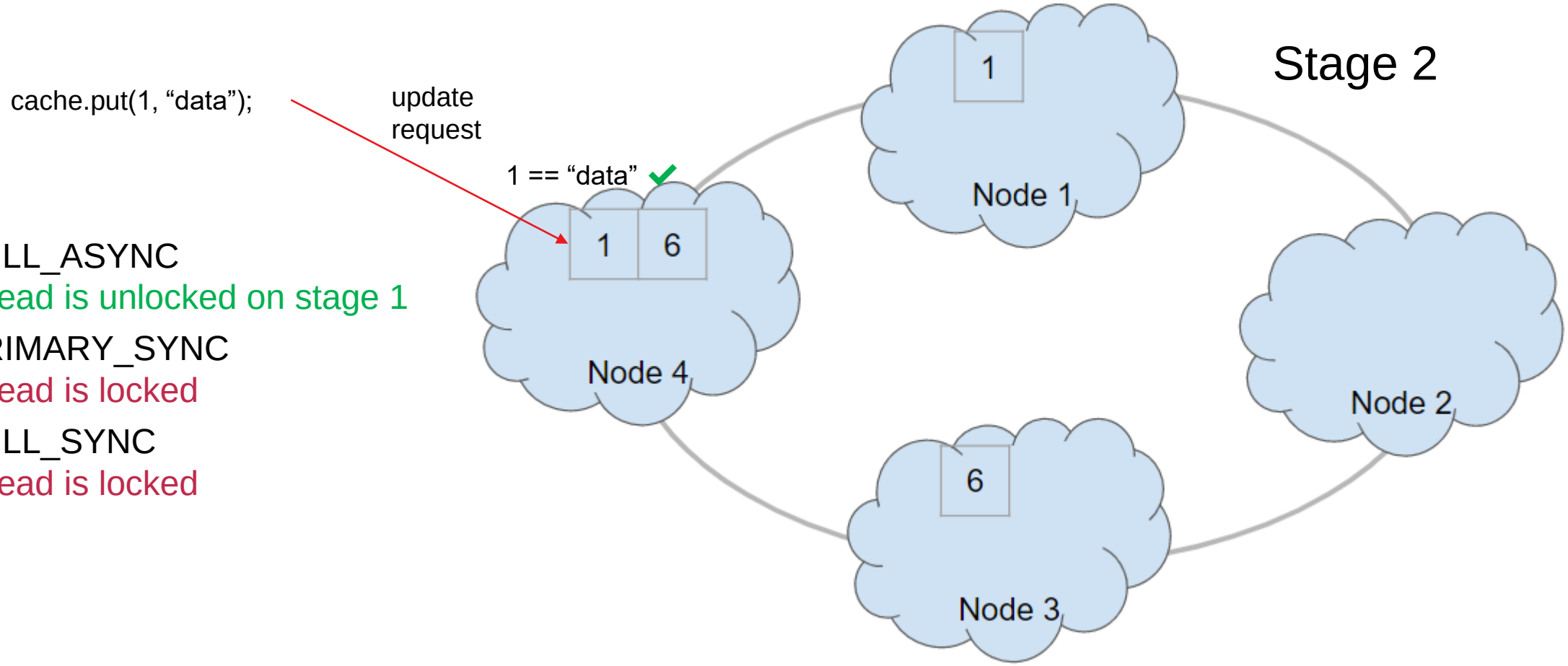
cache.put(1, “data”);

update request



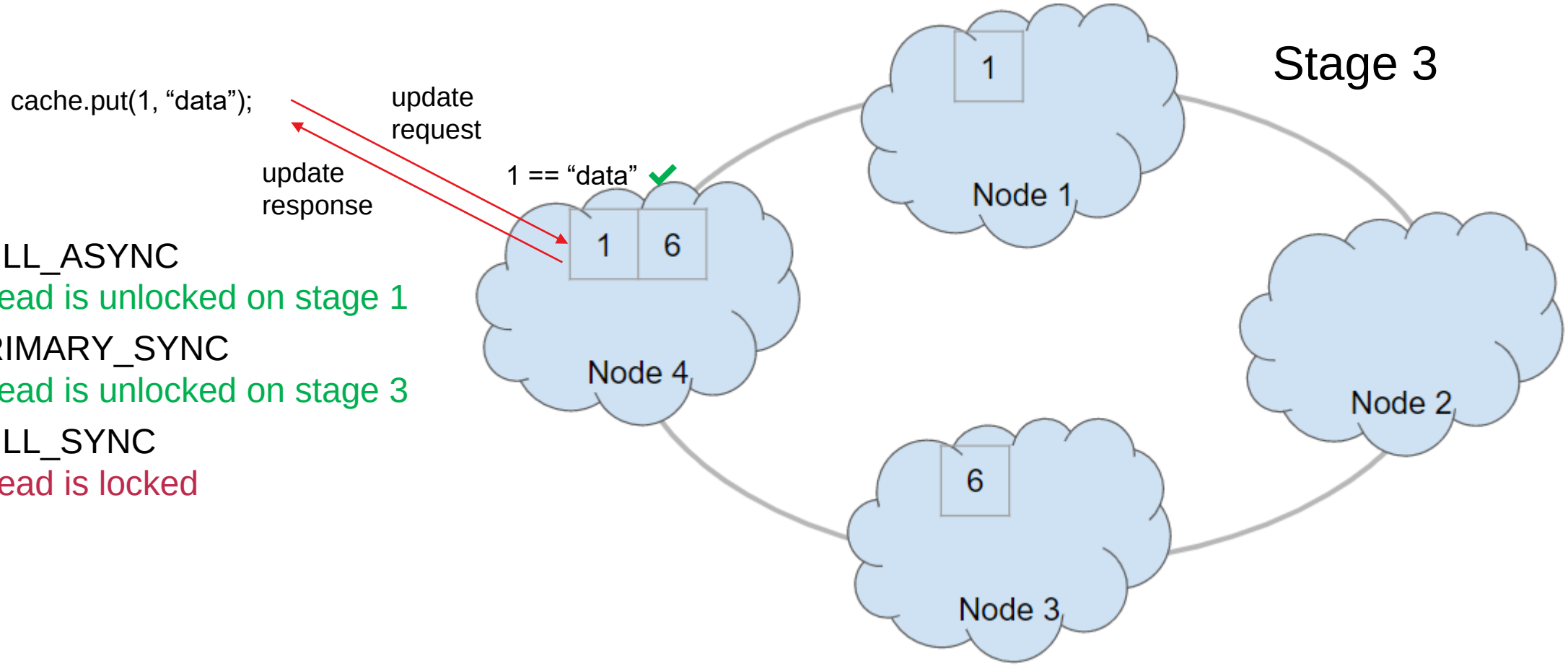
- FULL_ASYNC
  thread is unlocked on stage 1
- PRIMARY_SYNC
  thread is locked
- FULL_SYNC
  thread is locked

Data Sync Guarantees



- FULL_ASYNC
  thread is unlocked on stage 1
- PRIMARY_SYNC
  thread is locked
- FULL_SYNC
  thread is locked

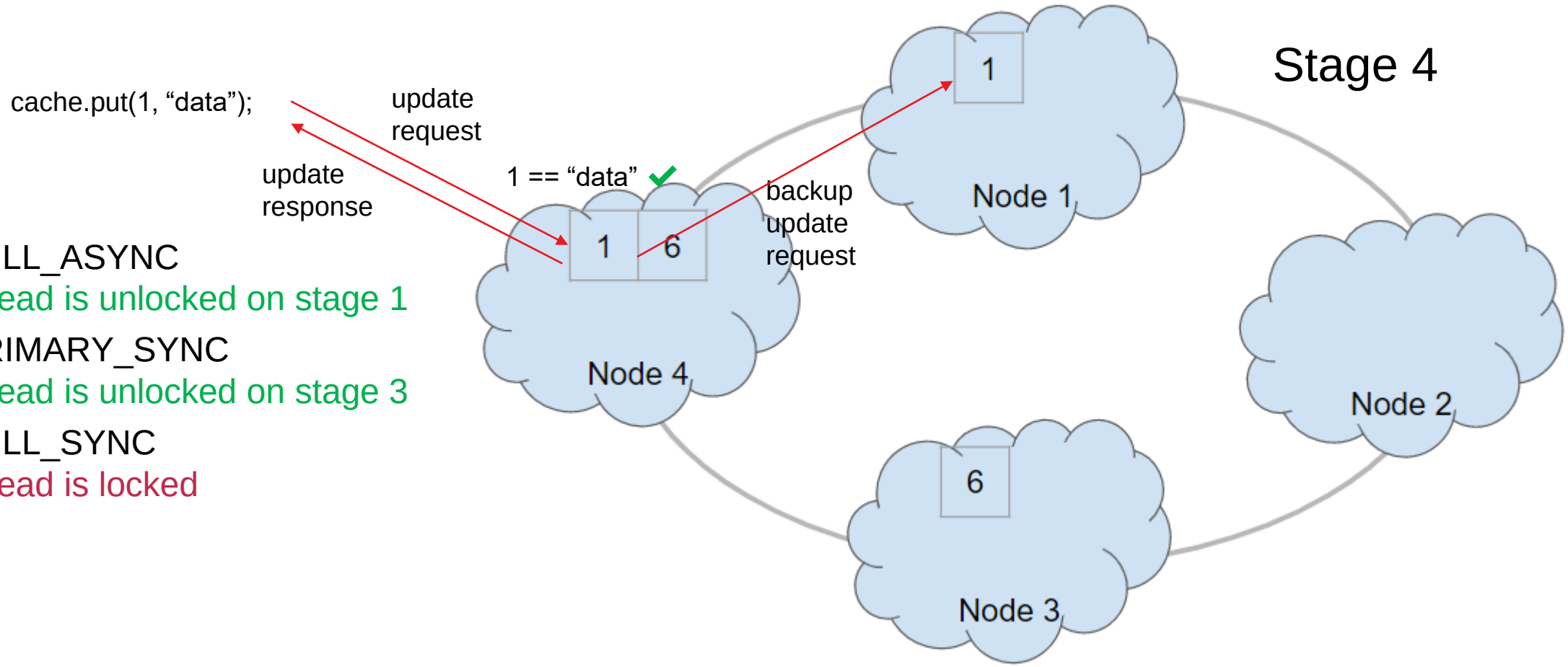
Data Sync Guarantees



- FULL_ASYNC
  thread is unlocked on stage 1
- PRIMARY_SYNC
  thread is unlocked on stage 3
- FULL_SYNC
  thread is locked

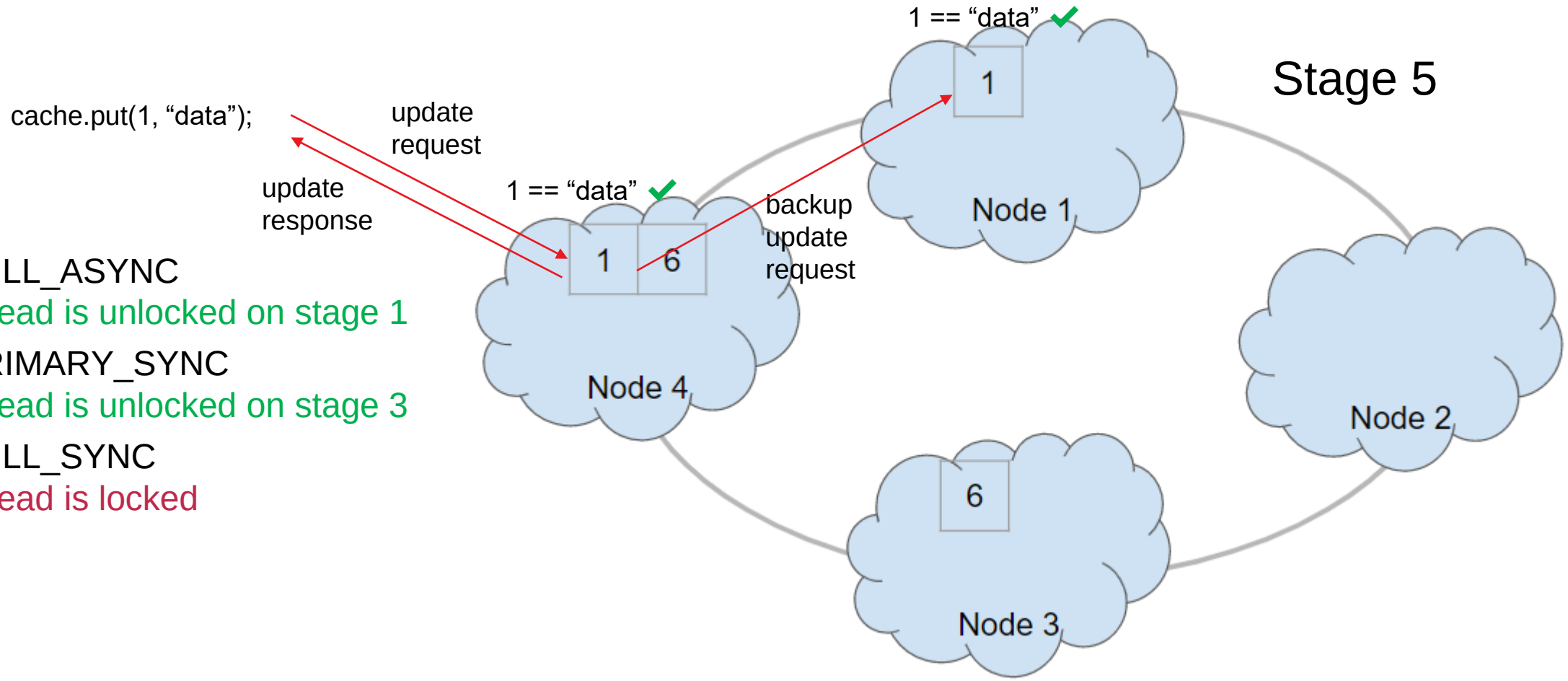
Data Sync Guarantees

- FULL_ASYNC
  thread is unlocked on stage 1
- PRIMARY_SYNC
  thread is unlocked on stage 3
- FULL_SYNC
  thread is locked

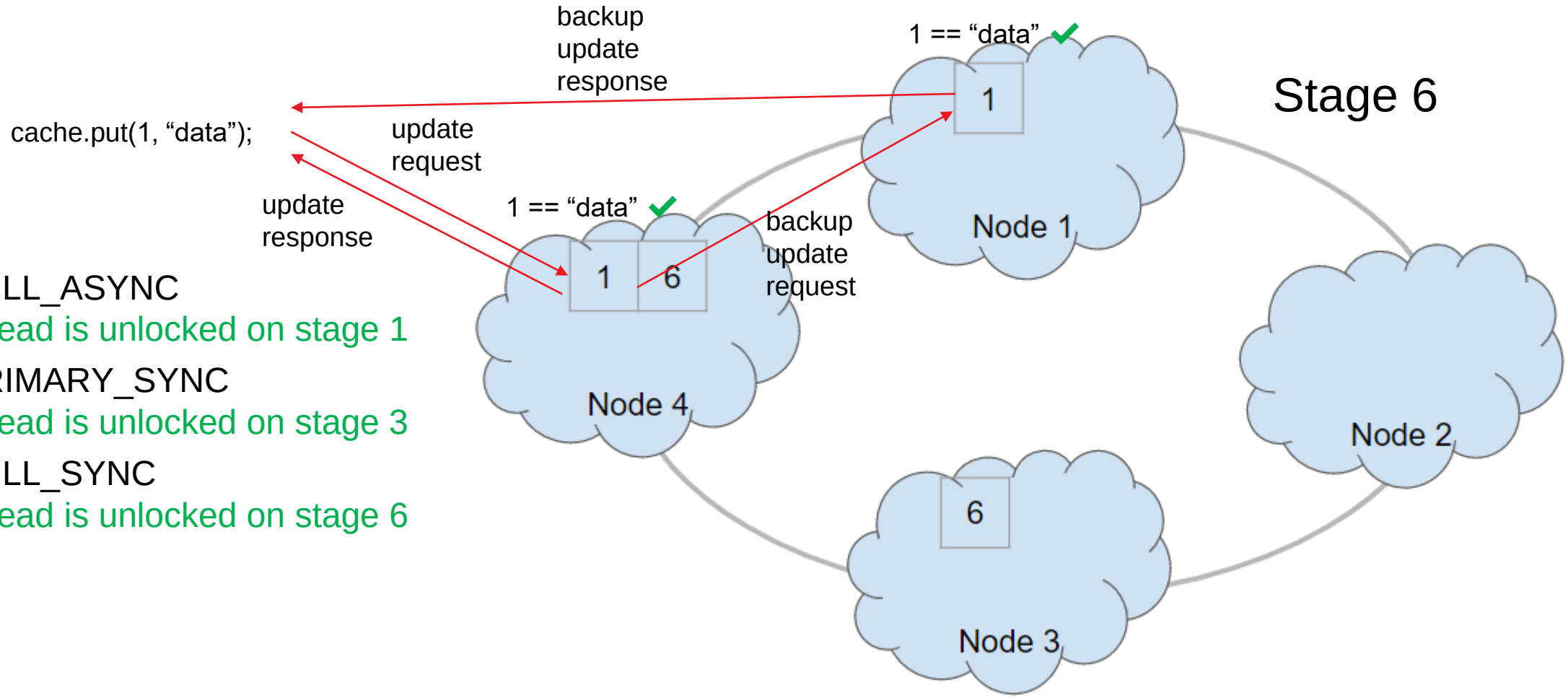


Data Sync Guarantees

- FULL_ASYNC
  thread is unlocked on stage 1
- PRIMARY_SYNC
  thread is unlocked on stage 3
- FULL_SYNC
  thread is locked



Data Sync Guarantees



- FULL_ASYNC
  thread is unlocked on stage 1
- PRIMARY_SYNC
  thread is unlocked on stage 3
- FULL_SYNC
  thread is unlocked on stage 6

Data Sync Guarantees

- cacheConfiguration.setReadFromBackup(**true** / false);

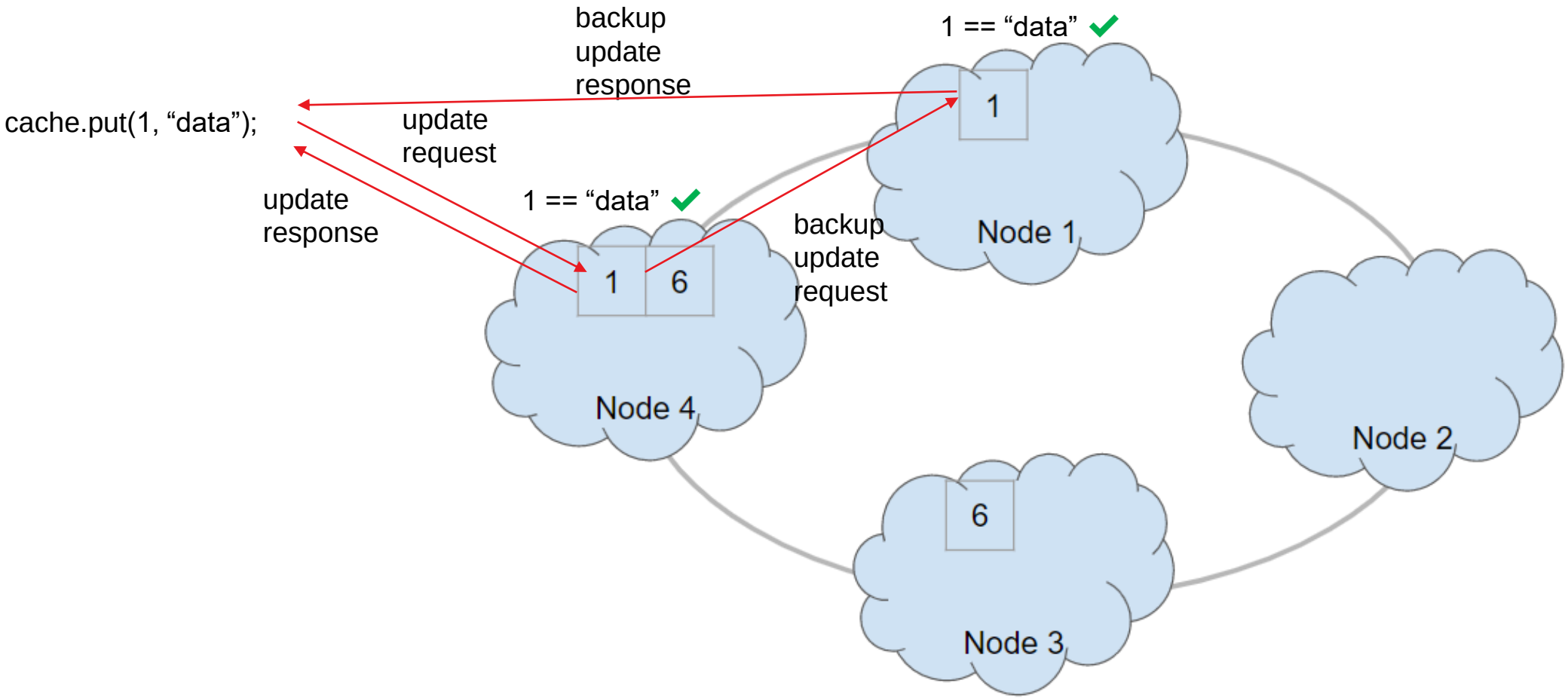
Data Sync Guarantees

- cacheConfiguration.setReadFromBackup(**true** / false);
  - Read operations are balanced between all partition copies

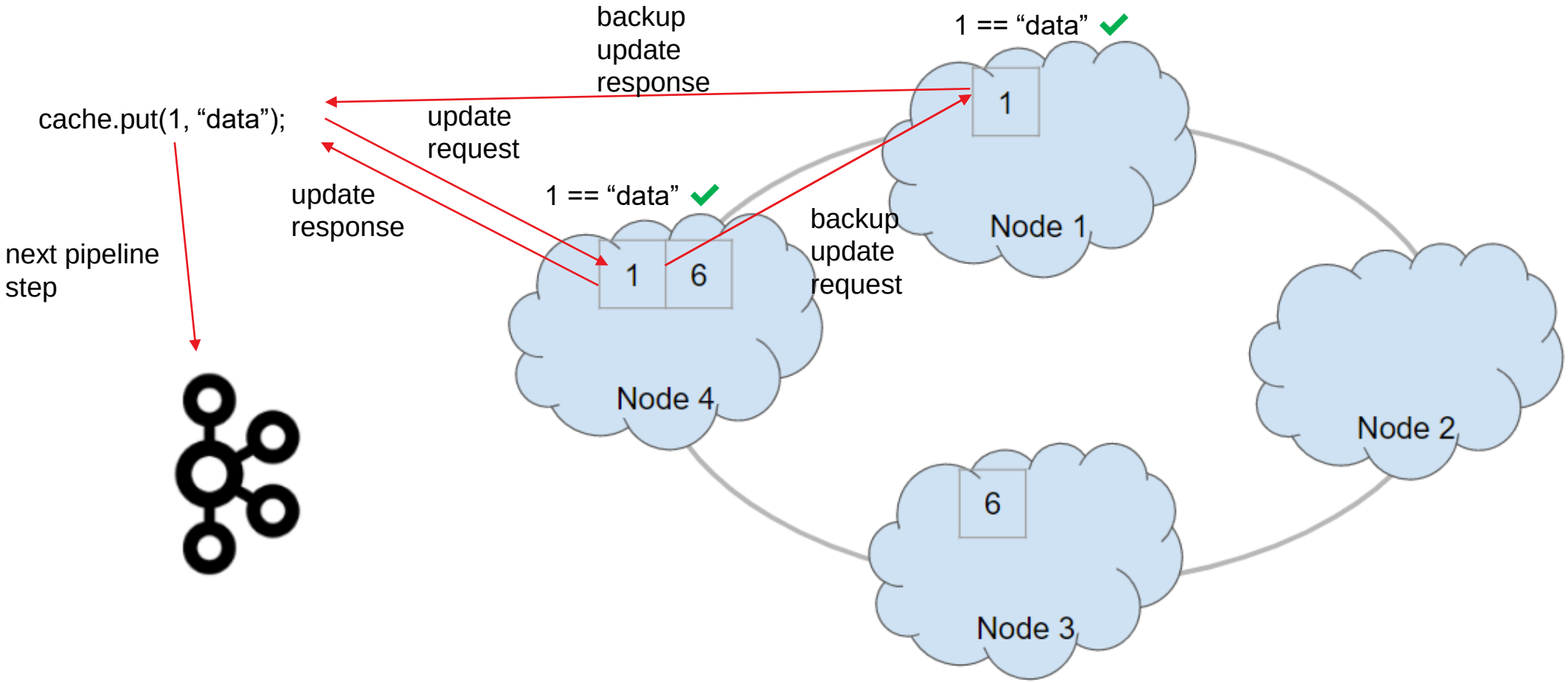
Data Sync Guarantees

- cacheConfiguration.setReadFromBackup(**true** / false);
  - Read operations are balanced between all partition copies
  - On REPLICATED caches reads are performed locally

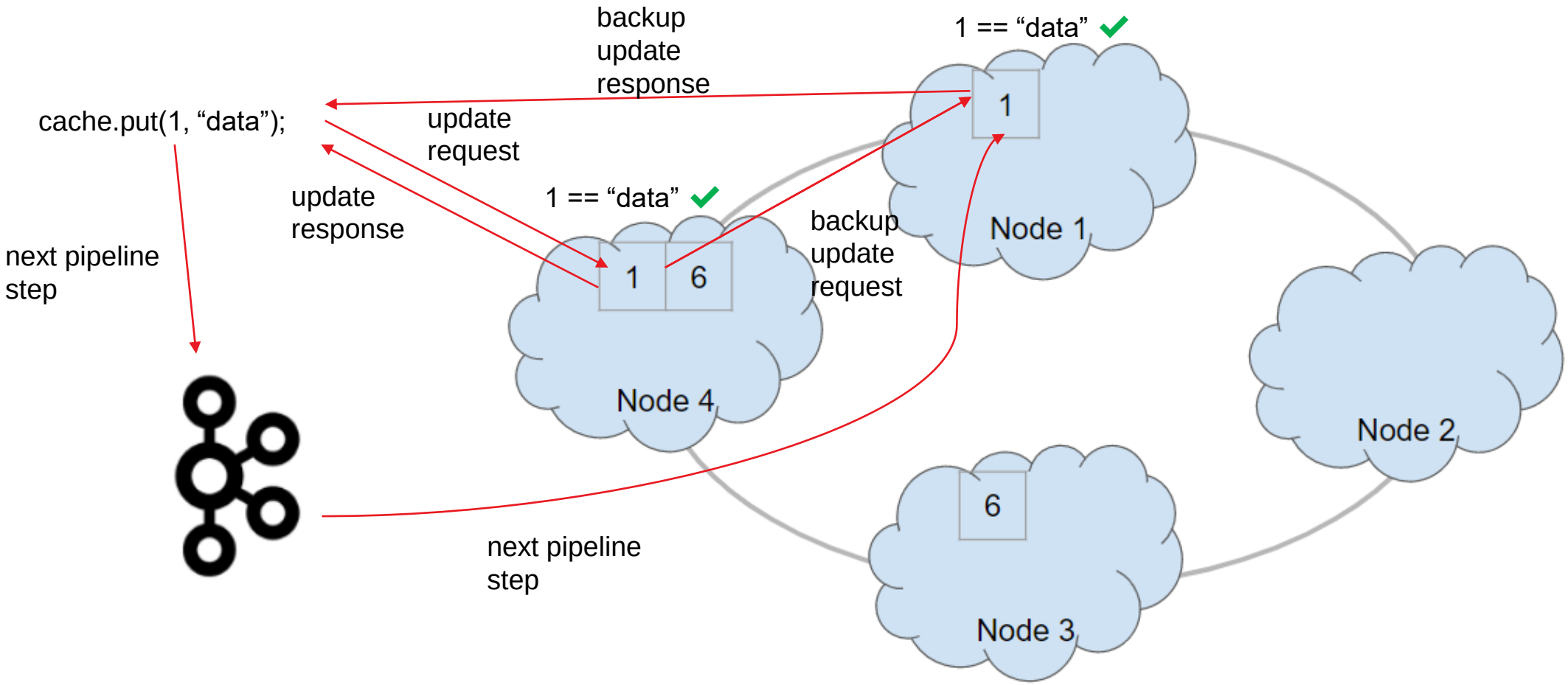
Why FULL_SYNC may be required for readFromBackup=true



Why FULL_SYNC may be required for readFromBackup=true



Why FULL_SYNC may be required for readFromBackup=true



Agenda

The most important configuration settings

- Data replication modes
- Data sync guarantees
- Data consistency
- Data storage
  - In-memory / disk
  - Capacity
  - Disk-based consistency

Data Consistency

- cacheConfiguration.setAtomicityMode(mode);

Data Consistency

- cacheConfiguration.setAtomicityMode(mode);
- ATOMIC
  - Only entry-level atomicity is guaranteed

```
cache.putAll(batch);
org.apache.ignite.cache.CachePartialUpdateException:
Failed to update keys (retry update if possible).: [7]
```

  - Higher performance

Data Consistency

- cacheConfiguration.setAtomicityMode(mode);
- ATOMIC
  - Only entry-level atomicity is guaranteed

```
cache.putAll(batch);
org.apache.ignite.cache.CachePartialUpdateException:
Failed to update keys (retry update if possible).: [7]
```

  - Higher performance
- TRANSACTIONAL
  - ACID guarantees
  - Cross-partition, cross-cache
  - Failover-safe (still ACID if some of participant nodes fail)

Transactional Caches

Use case 1: atomic batch update

```
try (Transaction tx = ignite.transactions().txStart(
  OPTIMISTIC, READ_COMMITTED, 300/*timeout*/, 0) {
  txCache.putAll(batch);
  ...
  tx.commit();
}

or

tx.putAll();
```

Transactional Caches

Use case 2: exclusive update

```
try (Transaction tx = ignite.transactions().txStart(
PESSIMISTIC, READ_COMMITTED, 300 /*timeout*/, 0) {
     lastUserExecutionCache.put(user1, UUID.randomUuid());
     // all similar operations with user1 are locked

     transfersCache.put(newTransferId, user1TransferData);
     auditCache.put(newAuditId, user1OperationData);
     …
     tx.commit();
}
```

Transactional Caches

Use case 3: safe money transfer

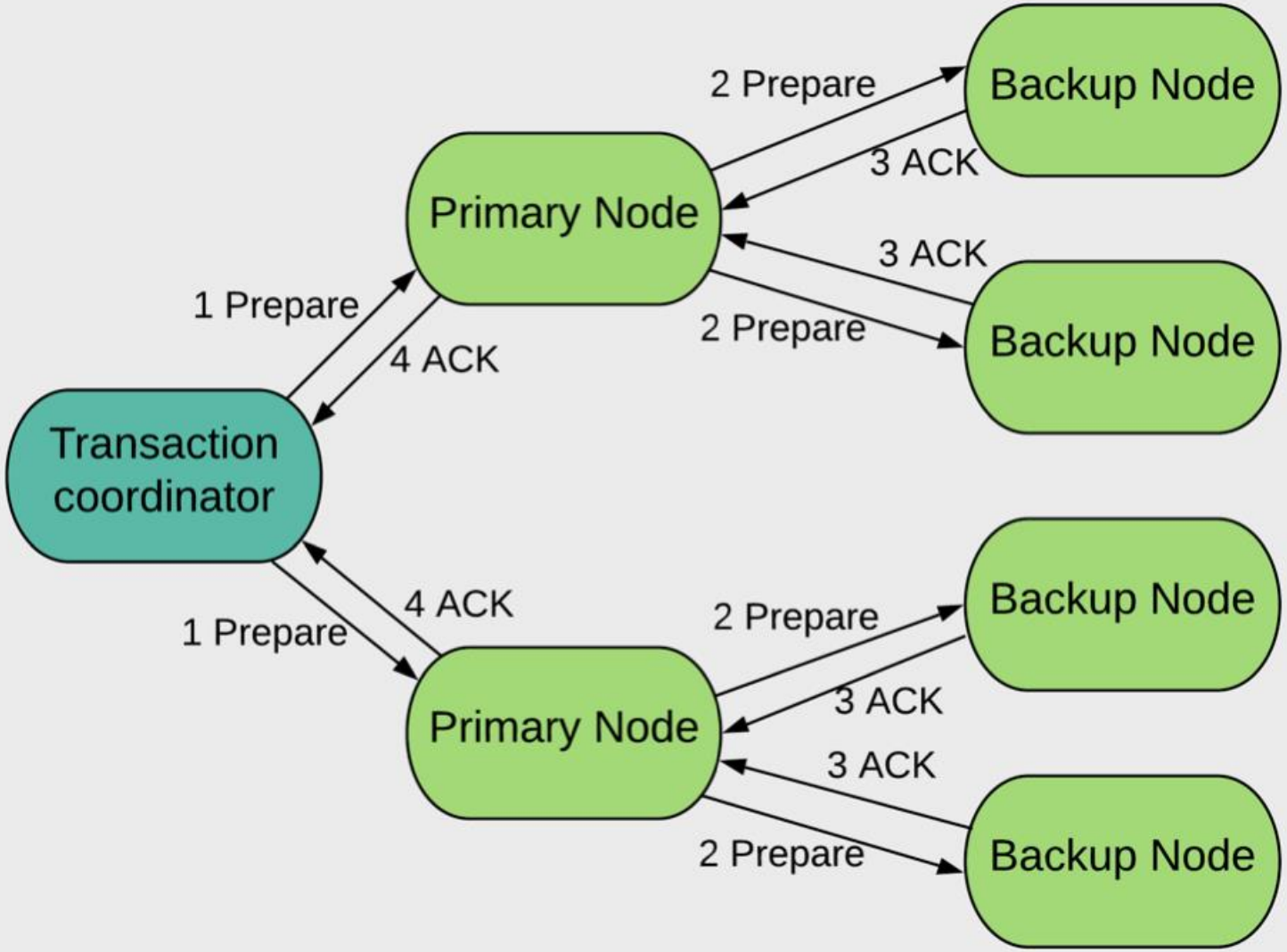
```
try (Transaction tx = ignite.transactions().txStart(
PESSIMISTIC, REPEATABLE_READ, 300 /*timeout*/, 0) {
     int balance1 = cache.get(acc1);
     int balance2 = cache.get(acc2);

     cache.put(acc1, balance1 - 100);
     cache.put(acc2, balance2 + 100);

     tx.commit();
}
```

Transactional Caches

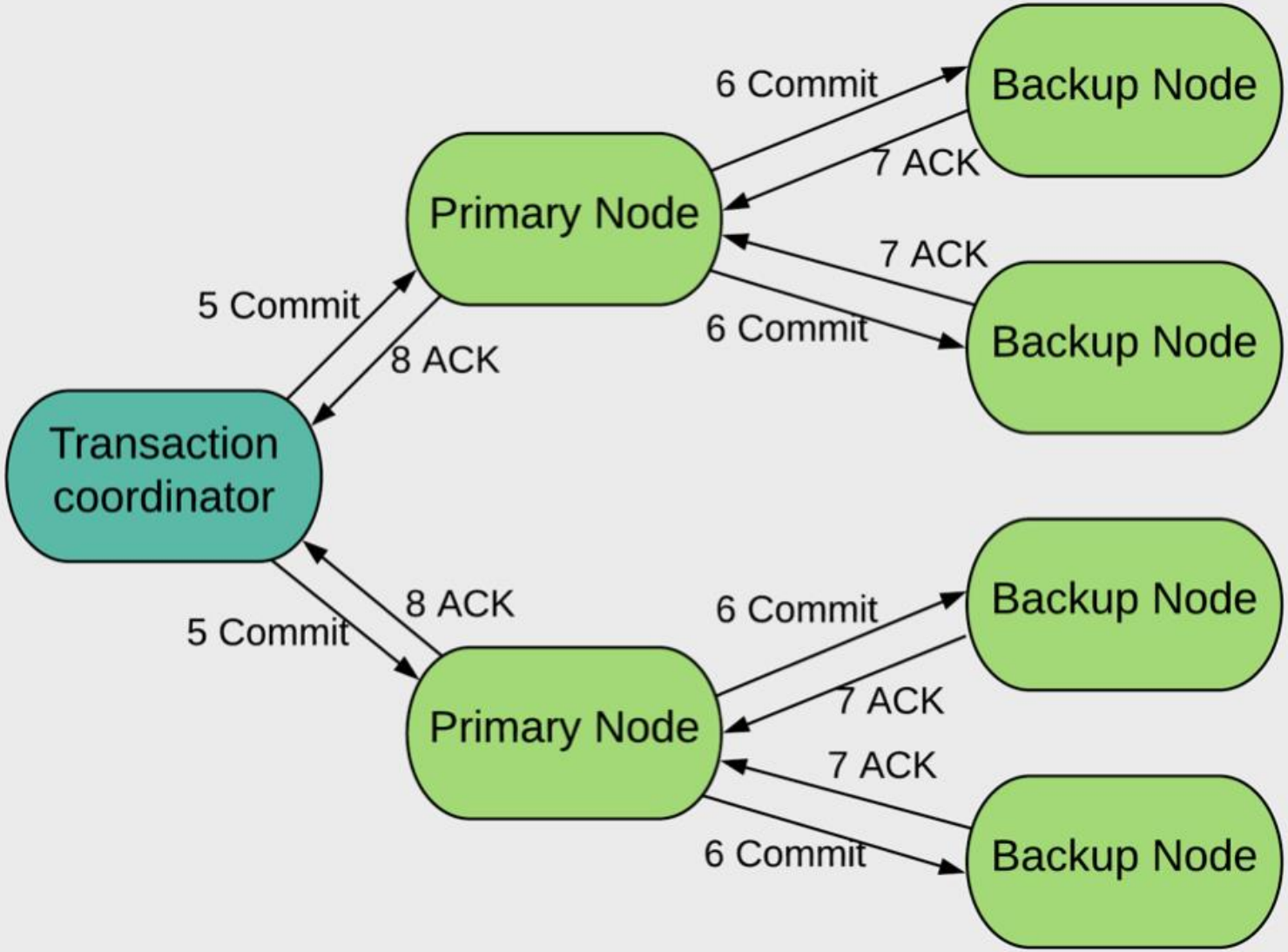
	OPTIMISTIC / READ_COMMITTED*	PESSIMISTIC / READ_COMMITTED	PESSIMISTIC / REPEATABLE_READ	OPTIMISTIC / SERIALIZABLE
ACID guarantees	✔	✔	✔	✔
Locks on write (exclusive update is possible)	x	✔	✔	✔ (optimistic lock)
Locks on read (money transfer is possible)	x	x	✔	✔ (optimistic lock)
Can be forcibly rolled back on concurrent update	x	x	x	✔ (if optimistic locking fails)
Automatic resolution of deadlocks caused by application	x	x	x	✔

* Batch putAll also has OPTIMISTIC / READ_COMMITTED guarantees

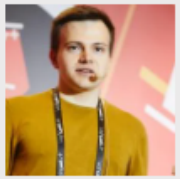
Transactional Caches: two-phase commit under the hood



Transactional Caches: two-phase commit under the hood



Transactions: to be continued!



Wednesday, June 10, 2020

Ivan Rakov

Moving Apache Ignite into Production: Best Practices for Distributed Transactions

The Apache Ignite transactional engine can execute distributed ACID transactions which span multiple nodes, data partitions, and caches/tables. This key-value API differs slightly from traditional SQL-based transactions but its reliability and flexibility lets you achieve an optimal balance between consistency and performance at scale by following several guidelines.

ATOMIC vs TRANSACTIONAL: what to choose

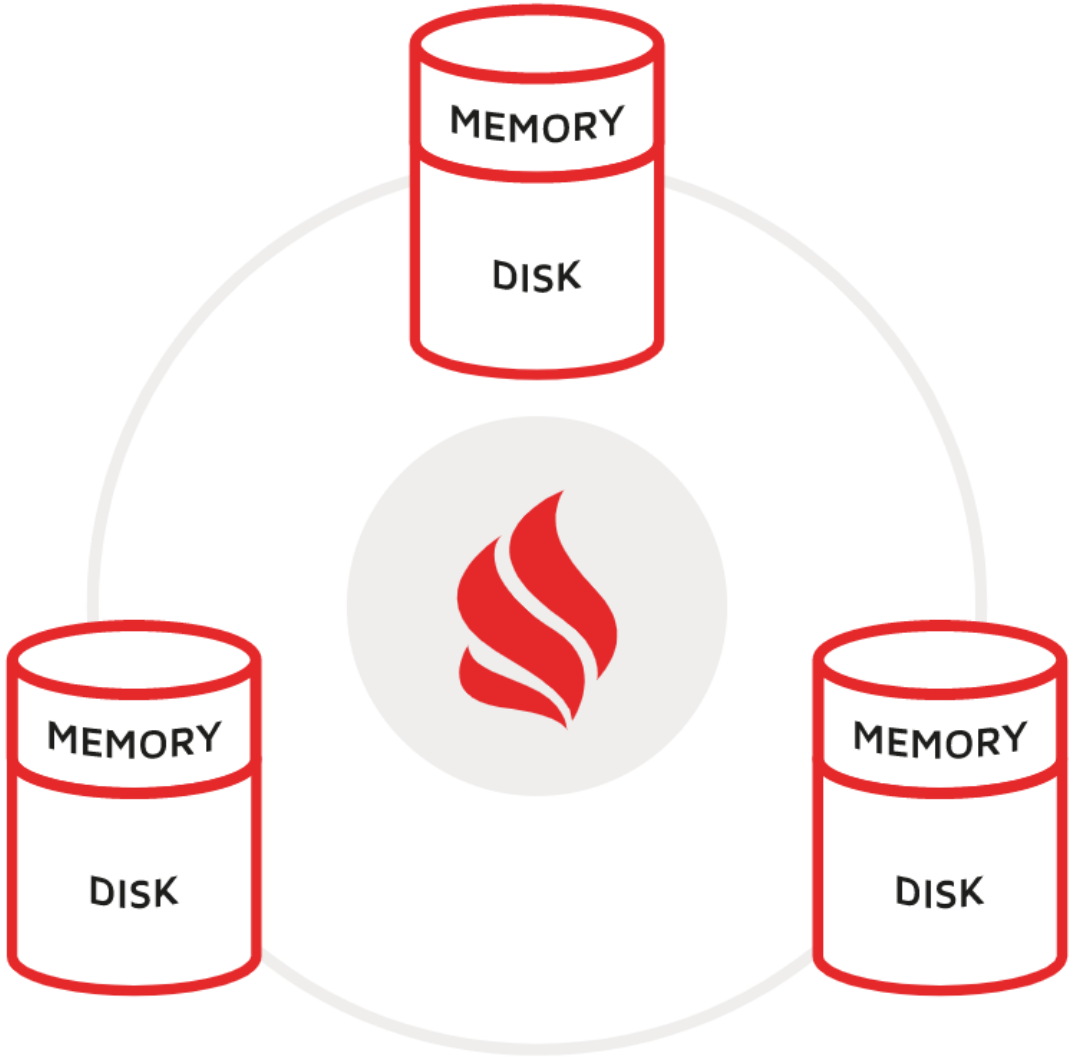
- ATOMIC
  - Choose if performance is crucial
- TRANSACTIONAL
  - Must be chosen if stronger than entry-level consistency may be needed

Agenda

The most important configuration settings

- Data replication modes
- Data sync guarantees
- Data consistency
- Data storage
  - In-memory / disk
  - Capacity
  - Disk-based consistency

Apache Ignite: multi-tier storage



Apache Ignite: multi-tier storage

- In-memory caches
  - Data are stored in preconfigured offheap region
  - Overflow causes IgniteOutOfMemoryException unless eviction mode is set

Apache Ignite: multi-tier storage

- In-memory caches
  - Data are stored in preconfigured offheap region
  - Overflow causes IgniteOutOfMemoryException unless eviction mode is set
- Persistent caches
  - Data are stored in preconfigured offheap region and are synced with disk
  - Overflow causes replacement of “cold” pages from offheap to disk

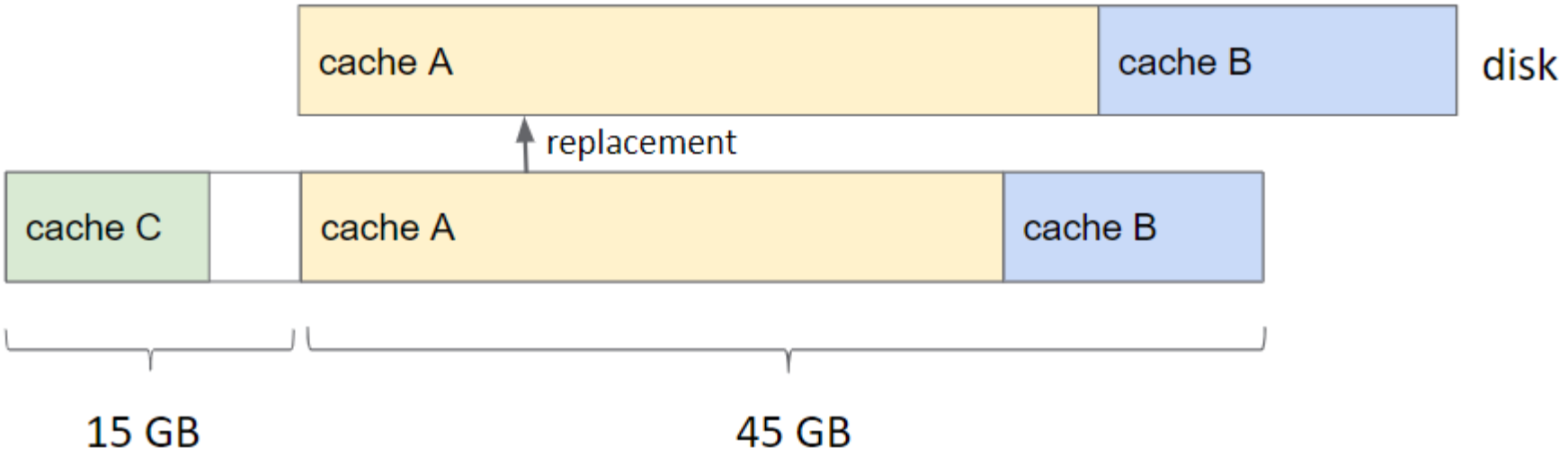
Use case: hot and cold data

In-memory region for hot data, persistent region for cold data

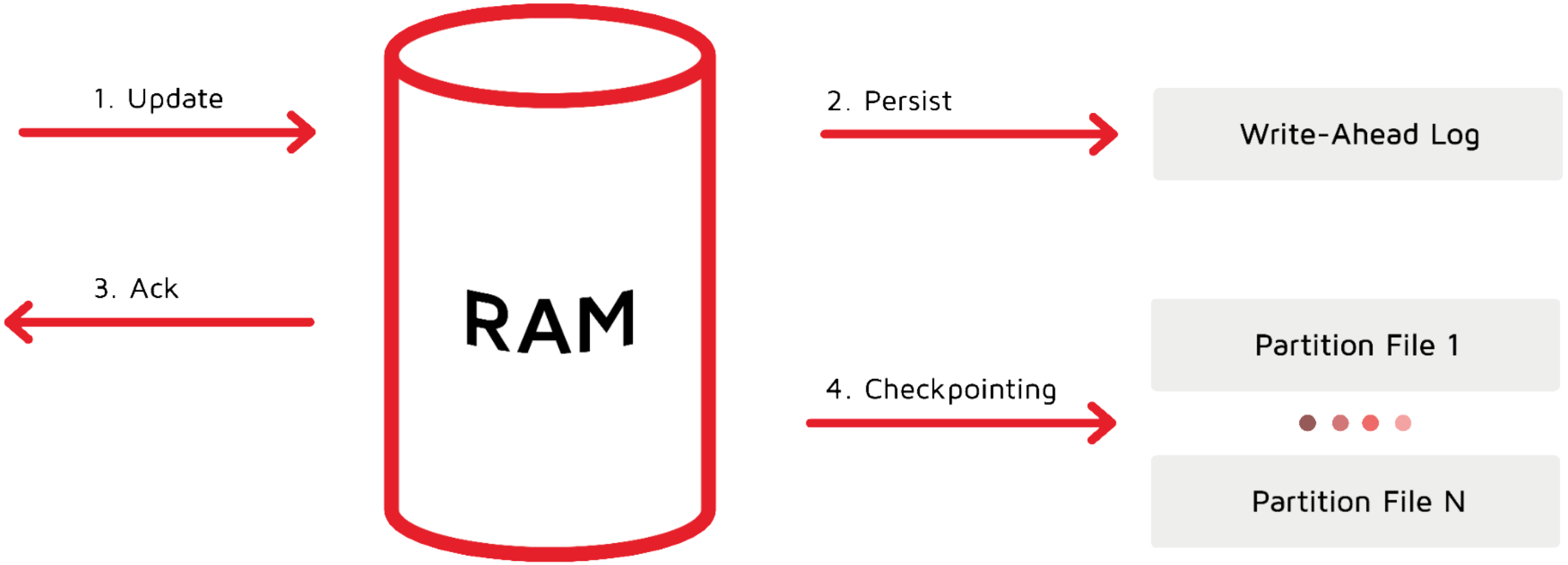
```
new DataStorageConfiguration()
    .setDefaultDataRegionConfiguration(
        new DataRegionConfiguration()
          .setMaxSize(45L * 1024 * 1024 * 1024)
          .setPersistenceEnabled(true))
    .setDataRegionConfigurations(
       new DataRegionConfiguration()
          .setName("hot")
          .setMaxSize(15L * 1024 * 1024 * 1024));
```

Use case: hot and cold data

- cacheCcfg.setDataRegionName(“hot”);



Ignite Persistence: high-level architecture



Disk-based consistency

Crash recovery in persistent mode is guaranteed due to keeping WAL

- dataStorageCfg.setWalMode(mode);

Disk-based consistency

Crash recovery in persistent mode is guaranteed due to keeping WAL

- dataStorageCfg.setWalMode(mode);
- BACKGROUND
  - Updates are logged to WAL file asynchronously

Disk-based consistency

Crash recovery in persistent mode is guaranteed due to keeping WAL

- dataStorageCfg.setWalMode(mode);
- BACKGROUND
  - Updates are logged to WAL file asynchronously
- LOG_ONLY
  - Updates are logged to WAL file synchronously

Disk-based consistency

Crash recovery in persistent mode is guaranteed due to keeping WAL

- dataStorageCfg.setWalMode(mode);
- BACKGROUND
  - Updates are logged to WAL file asynchronously
- LOG_ONLY
  - Updates are logged to WAL file synchronously
- FSYNC
  - Updates are logged to WAL file and synced with storage device synchronously via `fsync` syscall

Disk-based consistency: protection from lost updates

	BACKGROUND	LOG_ONLY (default)	FSYNC
Ignite process crash	x	✔	✔
Power loss / OS crash	x	x	✔

Disk-based consistency: protection from lost updates

	BACKGROUND	LOG_ONLY (default)	FSYNC
Ignite process crash	x	✔	✔
Power loss / OS crash	x	x	✔

LOG_ONLY is enough for keeping data safe in practice
If one node crashes, consistency will be recovered through rebalance

Summary: keywords

- Data replication modes PARTITIONED, REPLICATED, setBackups
- Data sync guarantees PRIMARY_SYNC, FULL_SYNC, readFromBackup
- Data consistency ATOMIC, TRANSACTIONAL
- Data storage
  - In-memory / disk Baseline Topology
  - Capacity DataStorageConfiguration
  - Disk-based consistency WAL mode

Thanks for your attention!
Questions?

e-mail: irakov@gridgain.com

public list for discussions: user@ignite.apache.org