بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

جلسه‌ی پانزدهم

پارتیشنینگ – شروع ترنزکشن

جلسه گذشته

ریپلیکیشن

Version Vector – Multiple replica

- Imagine you have N replicas: Replica A, Replica B, Replica C, …
- Each replica keeps track of a local integer counter.
- For example:
  - *Replica A has version counter vA.*
  - *Replica B has version counter vB.*
  - *Replica C has version counter vC.*

Version Vector – Multiple replica

- The version vector is a collection of each replica's version counters
- instead of a single integer, you have a small list/tuple of integer counters—one counter for each replica.
- Example: if you have three replicas, your version vector might look like [vA=5, vB=3, vC=7].

جلسه‌ی جدید

پارتیشنینگ

Partitioning

- Why?
- Partition / shard / region / vnode

Partitioning

- Partitioning is breaking up the data so that each piece of data is part of one partition, and the partitions are spread across multiple servers
- The main motivation for partitioning is scalability
- Partitioning can be used with replication, and the partitioning strategy is largely independent of the replication methodology, so the two mostly can be considered separately

Partitioning

- چه چیزهایی برامون مهمه؟

Partitioning on K/V store

- Key
- Value
- Queries on Key

Partitioning on K/V store

- Partitioning by Key range

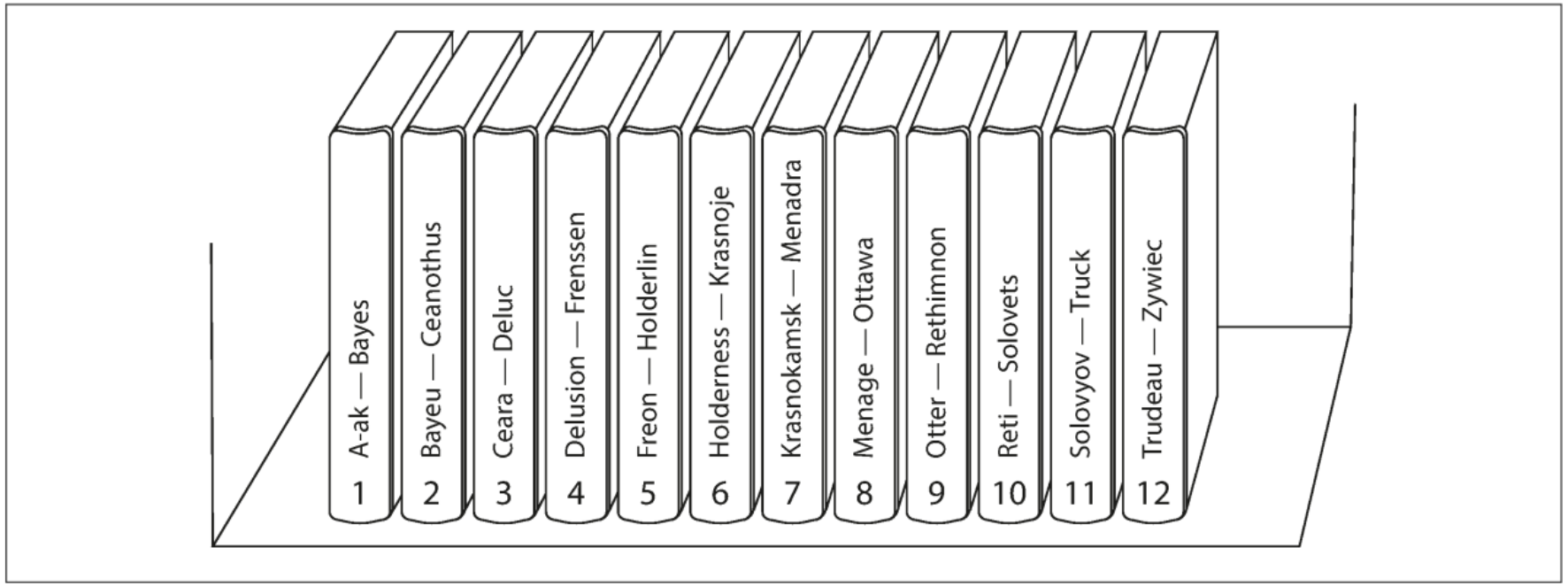


Figure 6-2. A print encyclopedia is partitioned by key range.

Partitioning on K/V store

- Partitioning by Key range
  - *Should ranges evenly spaced?*
  - *How to choose boundaries?*
  - *Data can be stored in sorted order, making range queries efficient*

- Beware that range partitioning can easily lead to hot spots

Partitioning on K/V store

- Example: an application that store sensors
  - *Key: timestamp*
  - *Value: metric name, measurement*
- Hot Spots!
  - *How to fix?*

Partitioning on K/V store

- Partitioning by Hash key

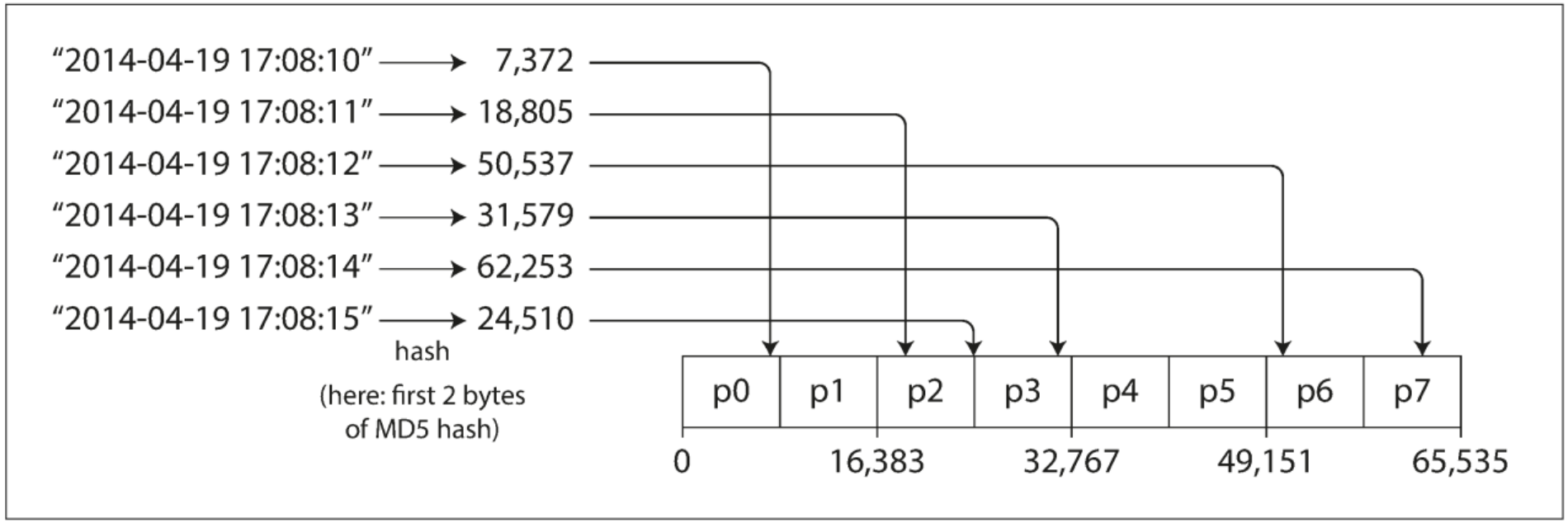


Figure 6-3. Partitioning by hash of key.

Partitioning on K/V store

- Partitioning by Hash key
  - *What properties a hash function need?*
    - Should they be cryptographically strong?
    - Is java HashCode is good?
  - *Assign continuous ranges of hash values to partitions*
    - Why not hash mode n? (explain later)

Partitioning on K/V store

- Partitioning by Hash key
  - *What about range queries?*
    - Range queries are now difficult
    - Cassandra, compound primary key,
      - *A primary key: col1, col2, col3, …*
      - *col1 used for partitioning*
      - *Col2, col3, … have local index*

Skewed Workloads and Relieving Hot Spots

- Hashing will generally even out requests across keys, but that doesn't guarantee no hot spots
- Example?
- If you have an extreme number of requests to the same key, you still have a hot spot
  - *How to fix?*

Partitioning and Secondary Indexes

- Why secondary indexes?
- Example:
  - *Divar - Cars*
  - *Primary key: post id*
  - *Secondary index: car color*

Partitioning secondary indexes by document

- This is a local index
- Each partition maintains its own secondary indexes for keys on that partition
- Writes are easy because all secondary indexes that need updating are on the same partition as the the key/document
- Reads, however, must be sent to every partition, called scatter/gather
  - *Because you must synchronously wait for every partition to respond, scatter/gather is prone to tail latency amplification*

Partitioning secondary indexes by document

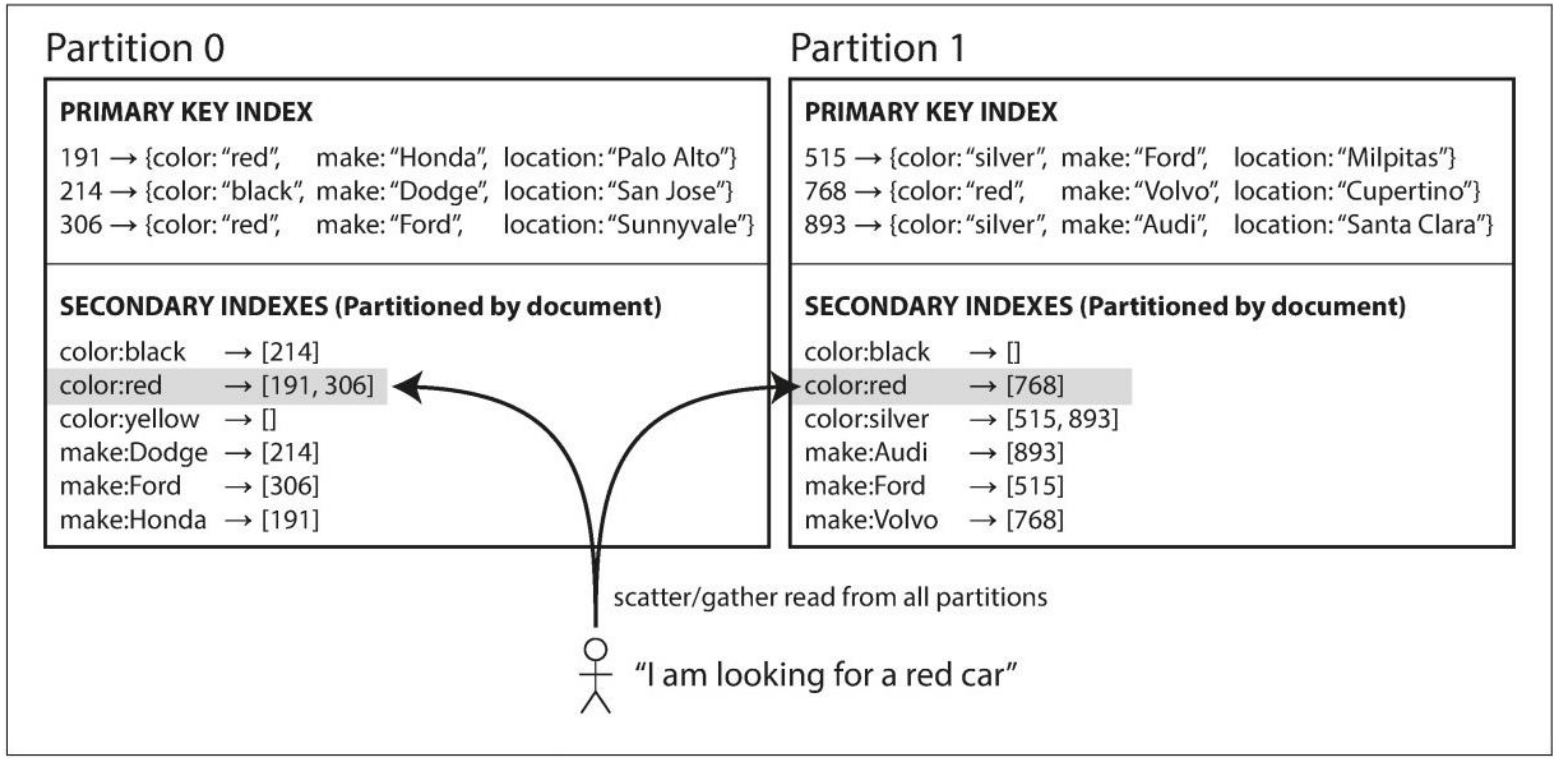


Figure 6-4. Partitioning secondary indexes by document.

Partitioning secondary indexes by term

- This is a global index
- Typically term-partitioned secondary indexes use ranges of values (terms), not hashed values.
  - *Why?*
- Reads using a secondary index access usually only require one/few partitions
- Writes, however, are more complex because other partitions may need to be updated for each secondary index that exists

Partitioning secondary indexes by term

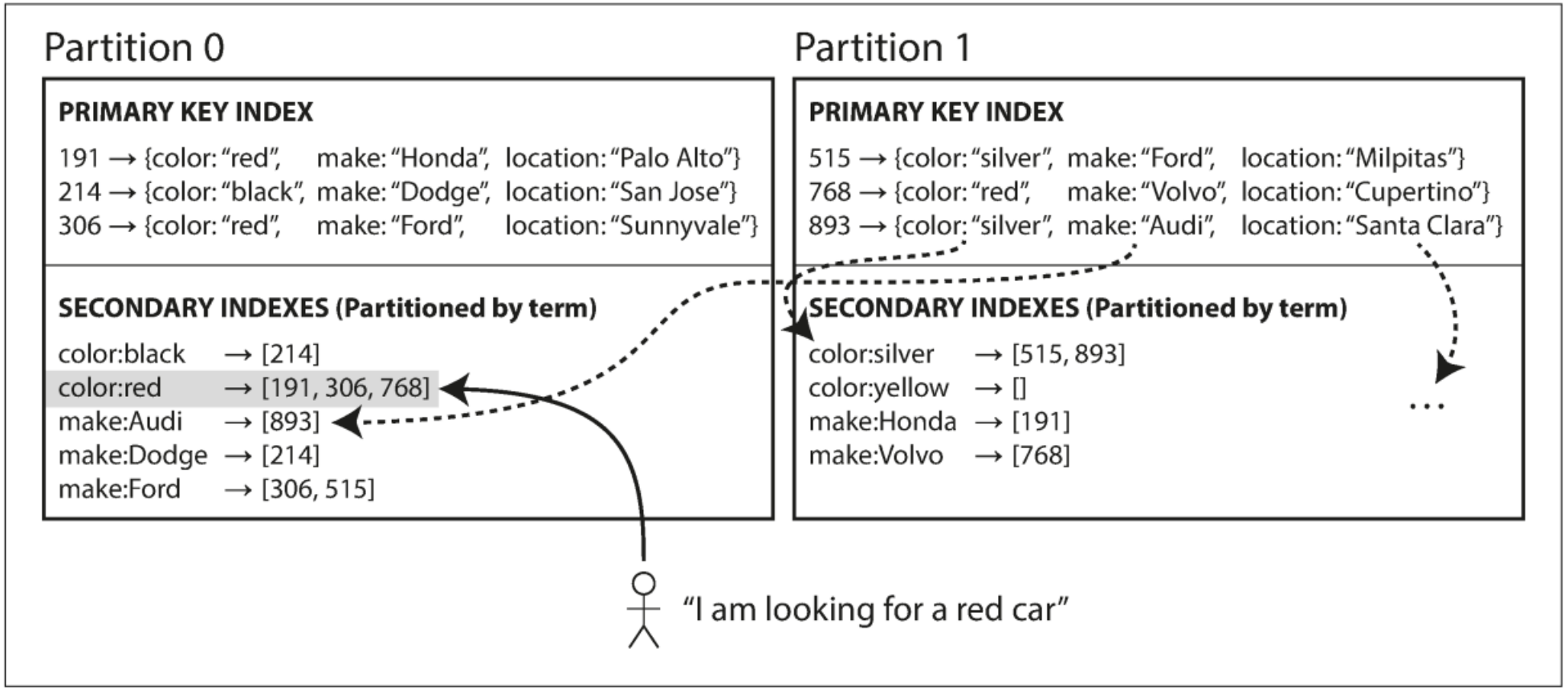


Figure 6-5. Partitioning secondary indexes by term.

Rebalancing Partitions

- What is rebalancing?
- Why it is needed?
- Over time:
  - *The query throughput increases, so you want to add more CPUs to handle the load*
  - *The dataset size increases, so you want to add more disks and RAM to store it*
  - *A machine fails, and other machines need to take over the failed machine's responsibilities.*

Rebalancing Partitions

- Adding hardware or a node failure requires shifting requests from one node to another
- Goals:
  - *While rebalancing, reads and writes still supported*
  - *Afterward, load is shared fairly between nodes*
  - *Try to minimize the amount of data moved during rebalancing*

Strategies for Rebalancing

- hash mod N??
- Example:
  - *Hash = 537, n = 10 -> partition assign to node 7*
  - *If n change to 11*
  - *About 90.9% of keys should move!*

Strategies for Rebalancing

- Fixed number of partitions
  - *Create partitions more than nodes*
  - *Example:*
    - 10 nodes but 1000 partitions
    - What happen if new node added?
    - What happened if a node removed?

Strategies for Rebalancing

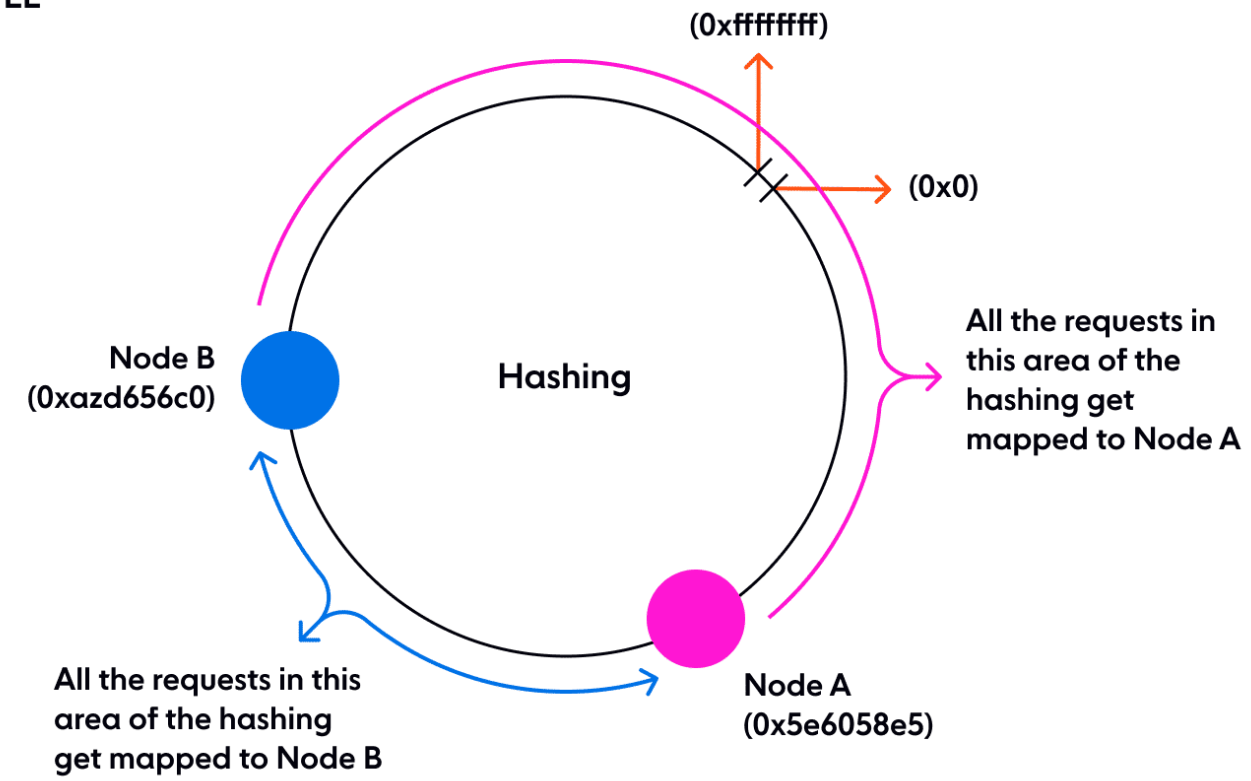
- Fixed number of partitions
  - *Consistent Hashing*

Consistent Hashing

- Two hash functions, one for keys and one for nodes
- A ring

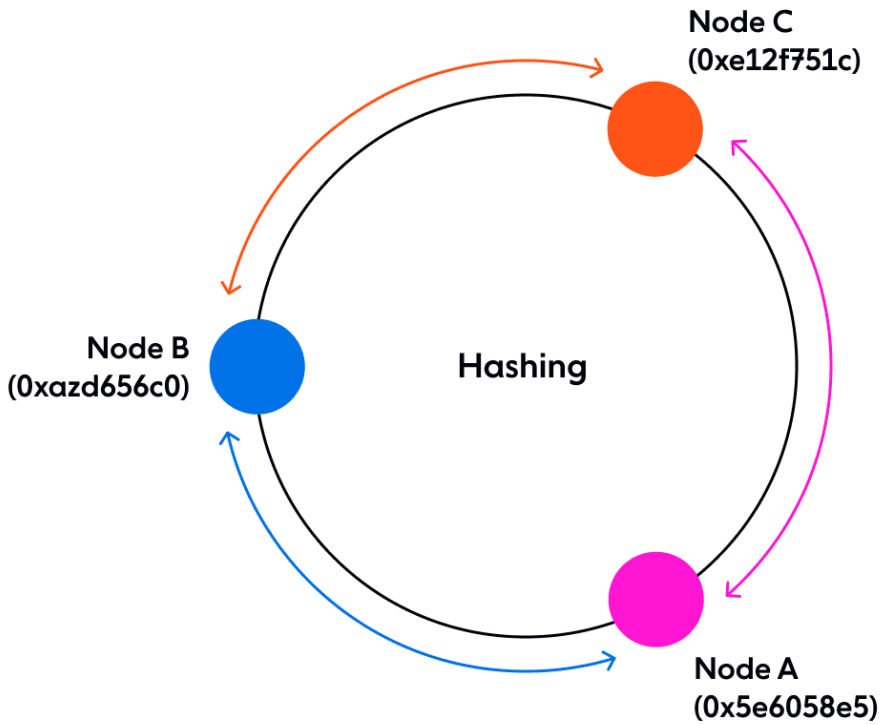
Consistent Hashing

WORKING EXAMPLE



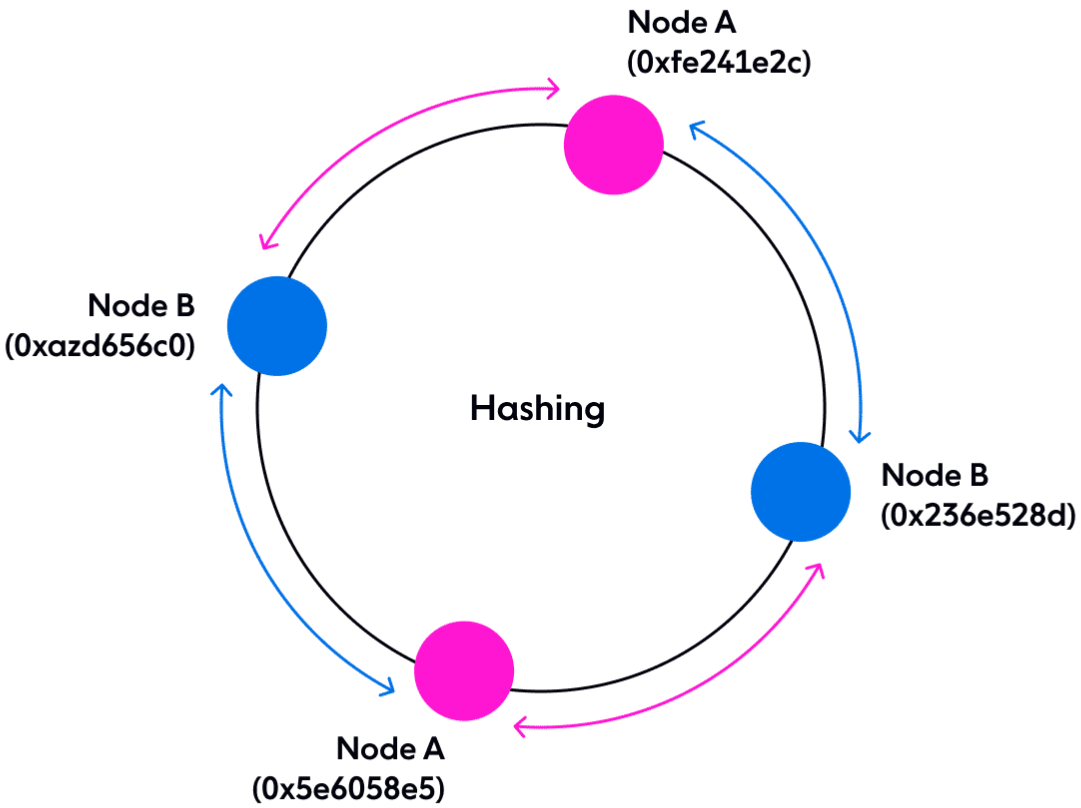
Consistent Hashing

NODE C
ADDED



Consistent Hashing

MULTIPLE
RANDOM
HASHES



Strategies for Rebalancing

- Dynamic partitioning
  - *For key range partitioning*
  - *Dynamic partitioning splits a partition when it exceeds a certain size, and merges a partition with another when it drops below another size threshold*
  - *The number of partitions grows with the size of the data*
  - *You may want to initialize an empty database with one partition per node*

Strategies for Rebalancing

- Partitioning proportionally to nodes
  - *For example: Each node has 200 partitions*
  - *When a node is added, it picks a number of existing partitions to split, and takes half of each*

Automatic or manual rebalancing

Request Routing

- How does a client know where to send each request?
- Service Discovery
- Solutions:
  - *Client sends to any node, and the node behind the scenes forwards the request to the proper partition when needed*
  - *Client sends to an intermediate routing tier, and that tier forwards the request*
  - *Clients keep track of partitioning and go directly to correct node*

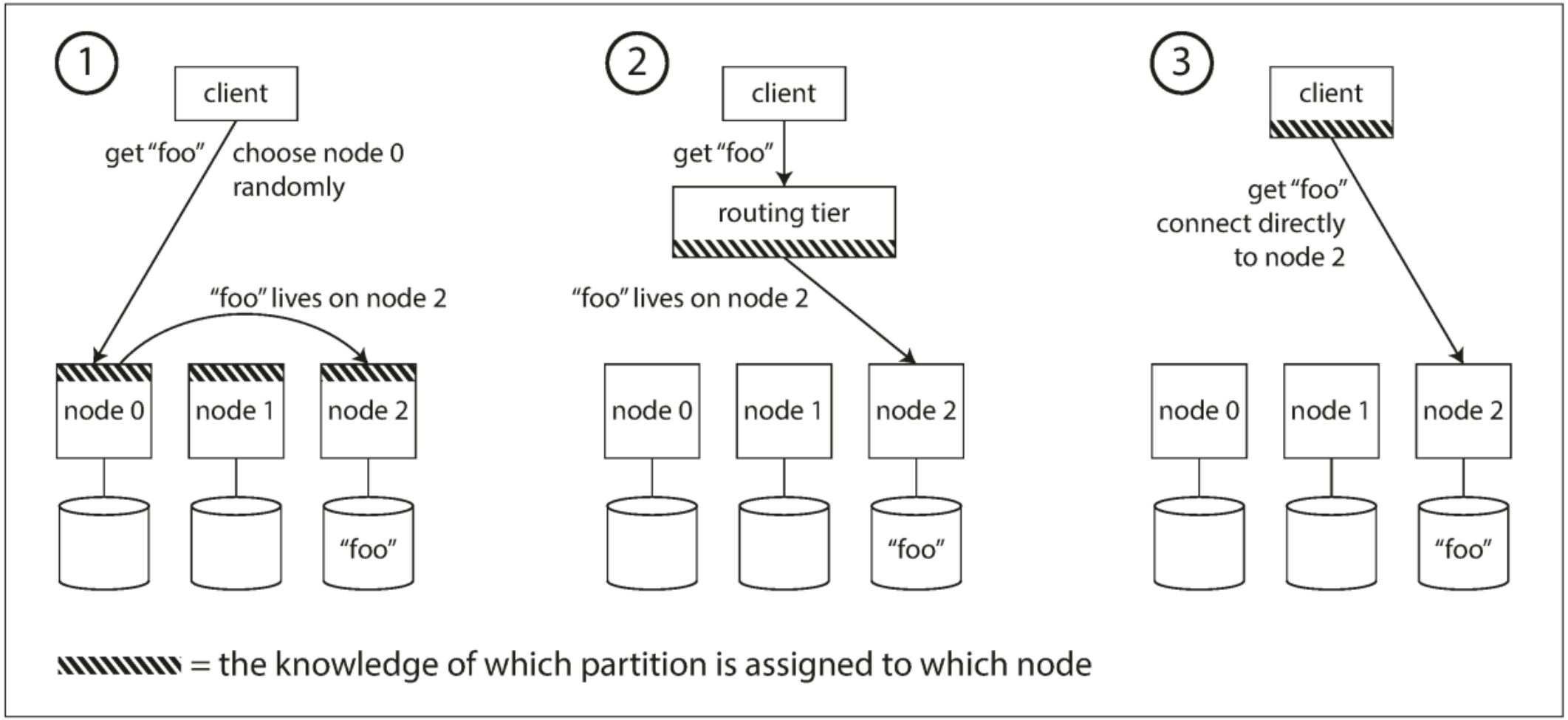
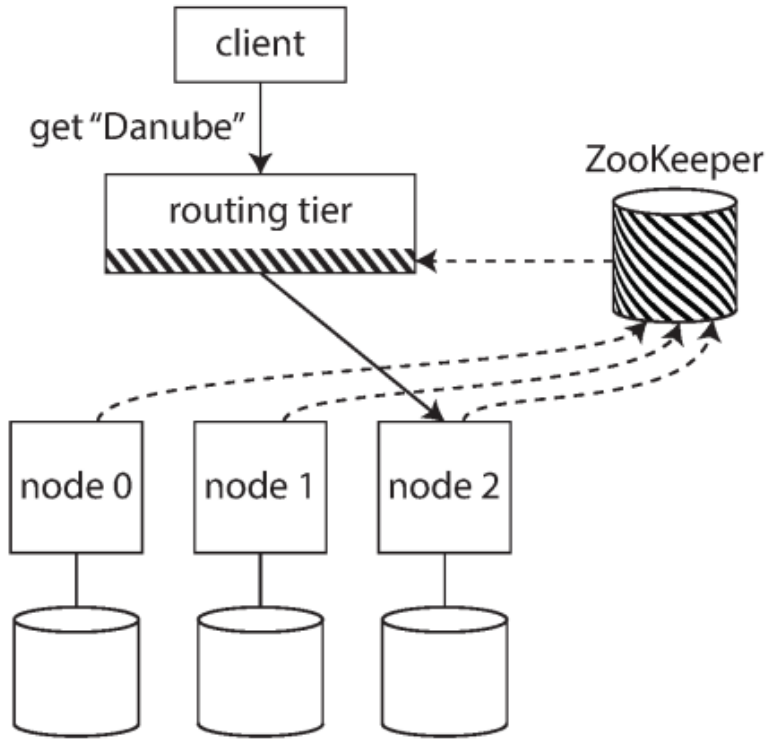


Figure 6-7. Three different ways of routing a request to the right node.



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

= the knowledge of which partition is assigned to which node

Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

Request Routing

- There will be nodes/tier/clients that need to keep track of partition changes!
- Consensus in discussed in chapter 9, and can be hard
- Many systems use an external service, such as Zookeeper, to keep track of cluster/partition metadata
- Cassandra and Riak use a *gossip protocol* to communicate changes.

Parallel Query Execution

- For analytic use cases, often want large amounts of data (from multiple keys). Most NoSQL implementations only support simple reads and maybe scatter/gather.
- Some relational products support massively parallel processing (MPP) where queries with complex joins, grouping, aggregation, and filtering can be coordinated by multiple nodes in parallel. Typically an optimizer coordinates all of the activity.
- Details will be discussed in chapter 10.

TRANSACTIONS

the harsh reality of data systems

- transactions have been the mechanism of choice for simplifying these issues.
- Purpose: simplify the programming model for applications accessing a database

Transaction

- A transaction is a series of operations grouped together as a unit, for which the system provides the certain set of guarantees mentioned above

- Although the concept of transactions applies to distributed databases, the examples in this chapter are all framed from a single-node perspective for the sake of introducing the topic with simpler examples

ACID Safety guarantees

- The safety guarantees provided by transactions are often characterized by the well-known acronym ACID, which stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**

ACID Safety guarantees

- Atomicity:
  - *Different from the atomic in concurrency*
  - *all writes will either succeed or all will fail. Upon error, any partial writes need to be undone.*

ACID Safety guarantees

- Consistency:
  - *in this context, used to mean DB will always be in a consistent state.*
  - Users probably interpret this as saying invariant properties will always remain true
  - But preserving invariants is really about how the application blocks operations in transactions
  - So consistency is really an application property, and shouldn't be part of this set of database guarantees

Consistency

- In Chapter 5 we discussed replica consistency and the issue of eventual consistency that arises in asynchronously replicated systems
- Consistent hashing is an approach to partitioning that some systems use for rebalancing
- In the CAP theorem (see Chapter 9), the word consistency is used to mean linearizability (see “Linearizability” on page 324).
- In the context of ACID, consistency refers to an application-specific notion of the database being in a “good state.

Consistency

- Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application.

ACID Safety guarantees

- Isolation:
  - *definition is that concurrent transactions won't interfere with each other at all, but that's rarely fully true*
  - Not interfering at all would be *serializability*, or acting as if the transactions had run one at a time (in some order), even though they may have actually had some or all parts running concurrently.
  - This is costly, so most systems don't provide full serializability
  - Most of the rest of this chapter is about isolation

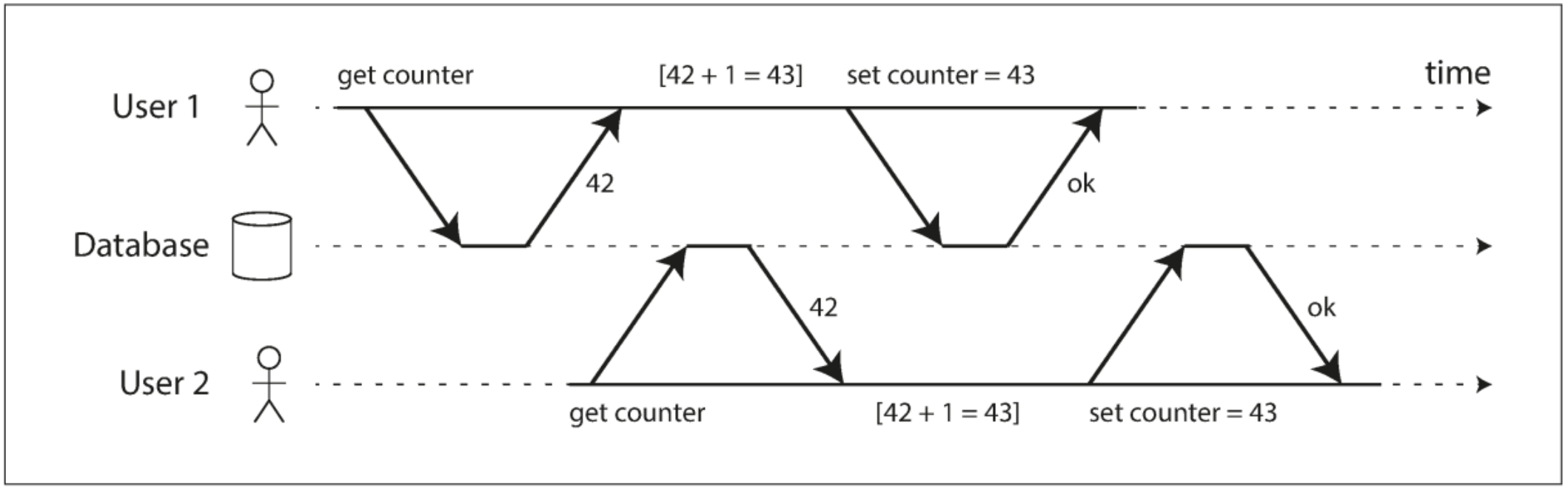


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

ACID Safety guarantees

- Durability:
  - *once a transaction is reported success, any data written won't be lost*
  - In practice, this is pretty intuitive, e.g. the data has been written to disk
  - Technically, (multiple) disks can fail, and other storage anomalies mean that data written to disk isn't fully 100.0% safe

Weak Isolation Levels

- Isolation levels weaker than serializable have been implemented to provide better scalability
- Because most systems do not provide serializability
- Rather than blindly relying on tools, we need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them.

Read Committed

- **dirty read:**
  - *Imagine a transaction has written some data to the database*
  - *but the transaction has not yet committed or aborted.*
  - *Can another transaction see that uncommitted data?*
  - *If yes, that is called a dirty read*

Read Committed

- **dirty read**

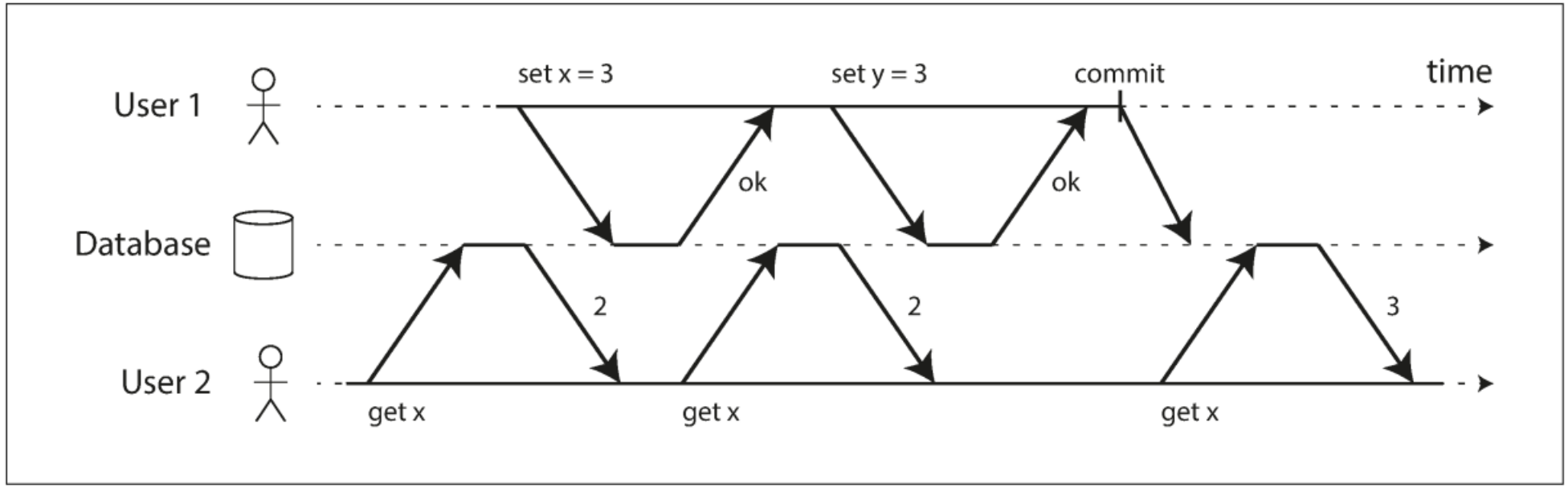


Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

Read Committed

- **dirty write:**
  - *Imagine two transactions concurrently try to update the same object in a database.*
  - *assume that the later write overwrites the earlier write.*
  - *However, what happens if the earlier write is part of a transaction that has not yet committed so the later write overwrites an uncommitted value?*
  - *This is called a dirty write*

Read Committed

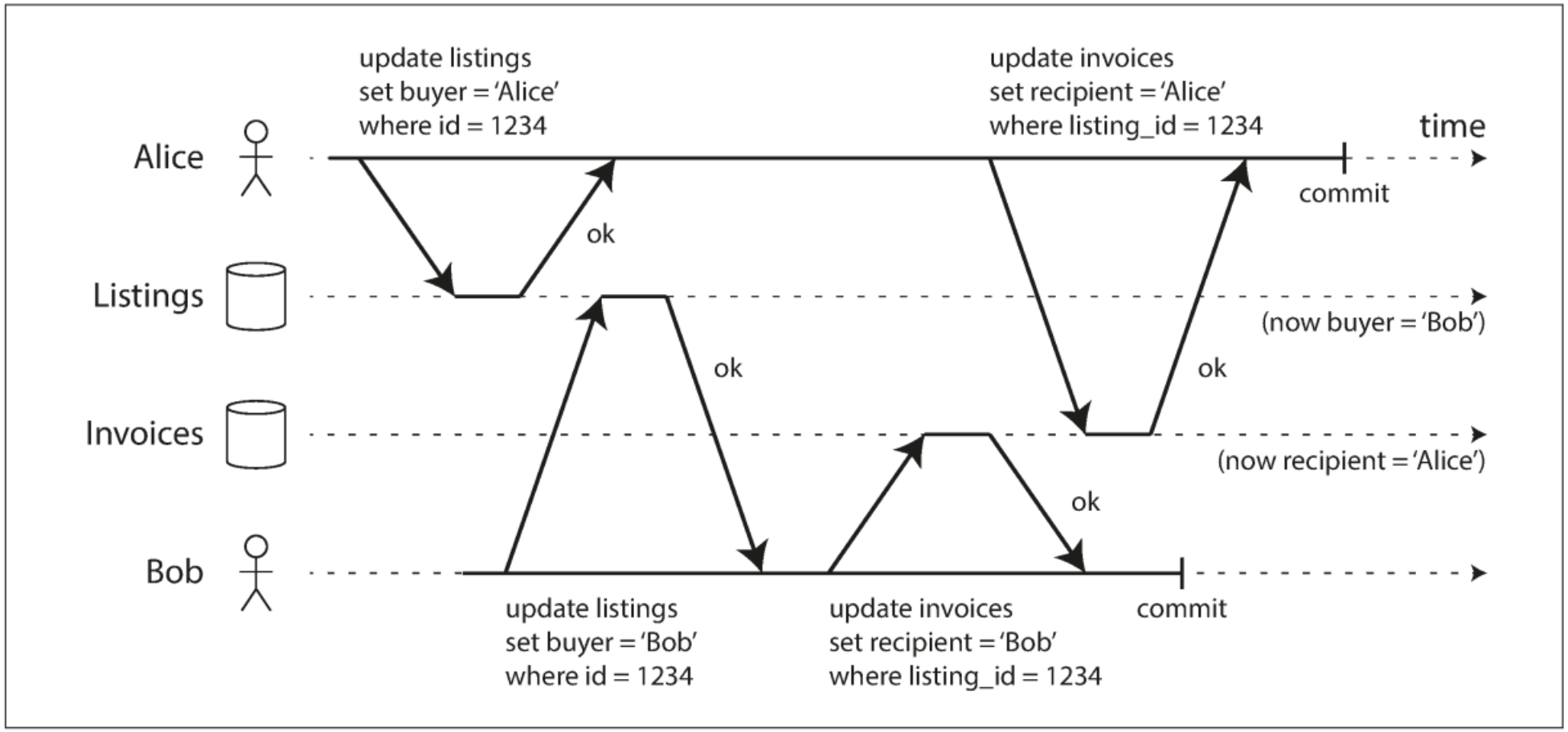


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Read Committed

- When reading from the database, you will only see data that has been committed
  - *no dirty reads*
- When writing to the database, you will only overwrite data that has been committed
  - *no dirty writes*

Read Committed

- How to implement it

Snapshot Isolation / Repeatable Read

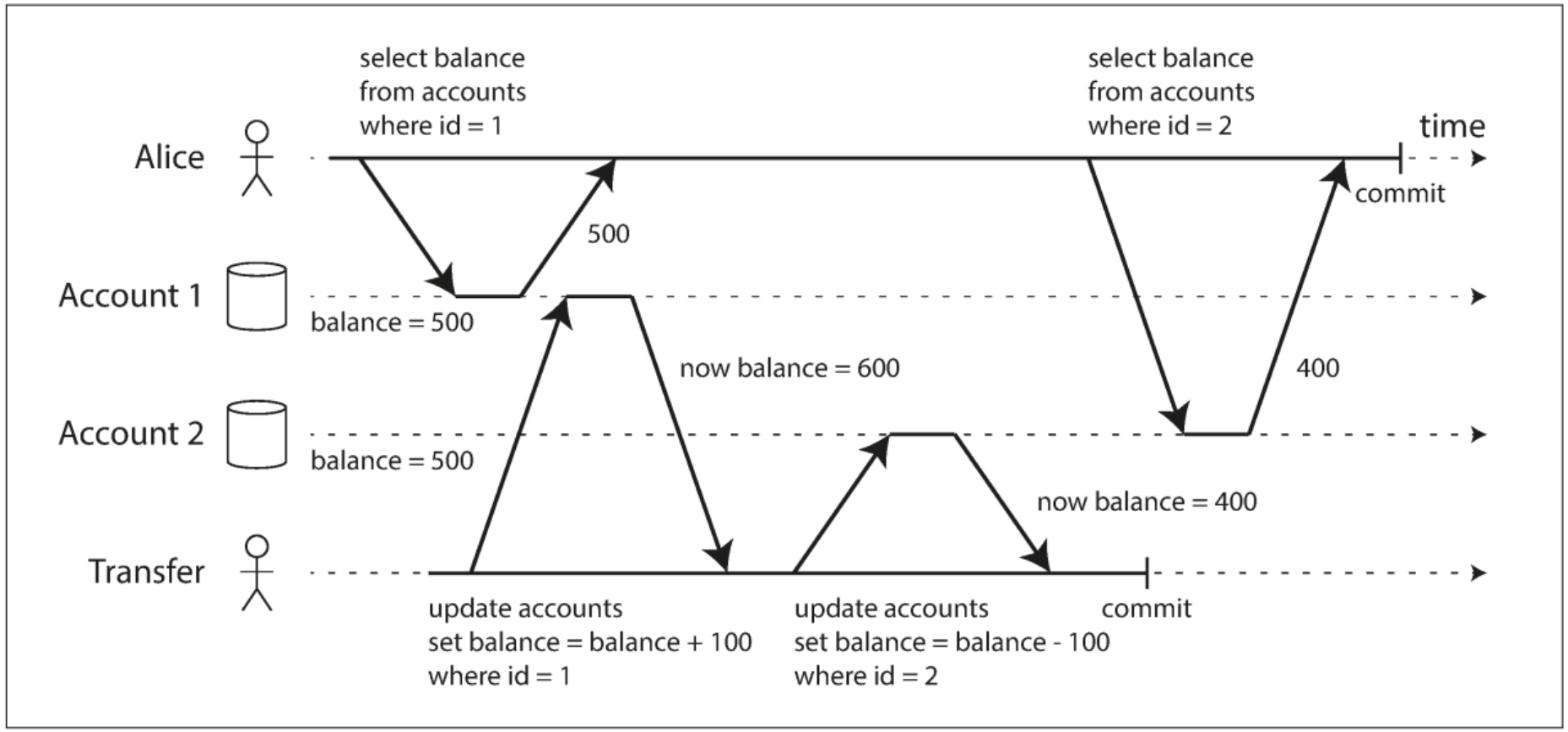


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

Snapshot Isolation / Repeatable Read

- Backups
- Analytic queries and integrity checks

Snapshot Isolation / Repeatable Read

- Solution:
  - *Multi version concurrency control (MVCC).*

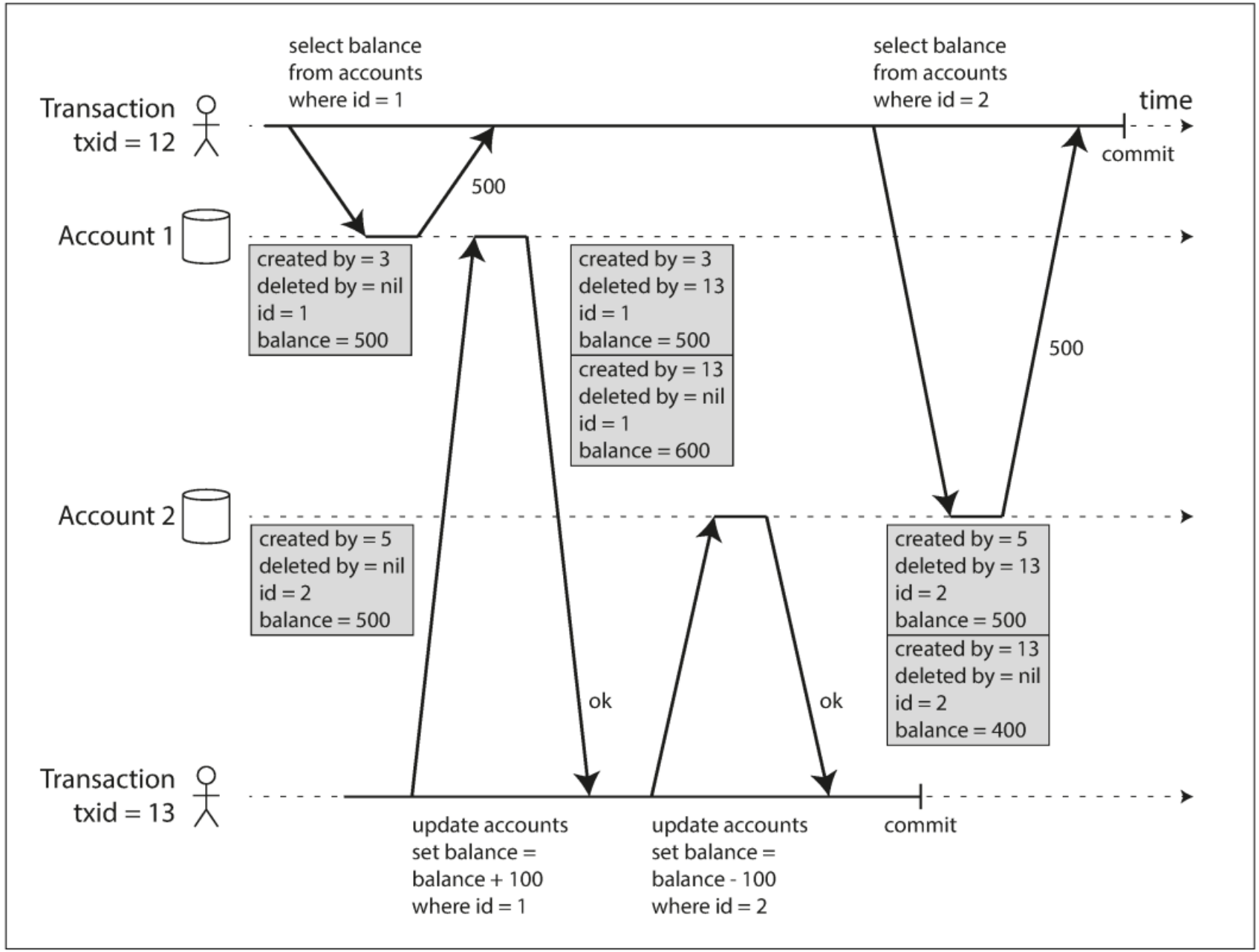


Figure 7-7. Implementing snapshot isolation using multi-version objects.

MVCC

- Visibility rules for observing a consistent snapshot
  - *At the start of each transaction, the database makes a list of all the other transactions that are in progress (not yet committed or aborted) at that time. Any writes that those transactions have made are ignored, even if the transactions subsequently commit.*
  - *Any writes made by aborted transactions are ignored.*
  - *Any writes made by transactions with a later transaction ID (i.e., which started after the current transaction started) are ignored, regardless of whether those transactions have committed.*
  - *All other writes are visible to the application's queries*

MVCC

- Indexes and snapshot isolation

- Snapshot isolation naming confusion
  - *In Oracle it is called serializable*
  - *in PostgreSQL and MySQL it is called repeatable read*

جلسه‌ی بعد