Fundamentals of NoSQL

CS185C: Introduction to NoSQL Databases

Instructor: Dr. Kim

Reference

- NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Oct 12, 2009, by Pramod J. Sadalage and Martin Fowler

What is wrong with relational databases?

- Nothing if they are used for right problems.
- The relational data model is intended to be useful and applicable to certain problems.
- A better question is “What problem do you have?”
    - You want to ensure that your solution matches the problem that you have.
    - For example, relational databases may not be a solution to deal with the explosion of big data from the Web, in particular social network.

The value of relational databases

- Rigid Schema
  - Structured data
  - Application logic becomes simple.
  - SQL can describe a table oriented operation at high level.
- SQL
  - Feature-rich and uses a simple, declarative syntax
  - Powerful – DML, DDL, DCL, and aggregations
  - Easy to use – basic syntax can be learned quickly
  - It is a standard - easy integration of a RDBMS with a wide variety of systems
- Transaction - ACID properties
  - Atomicity – all or nothing, no partial failure
  - Consistency– data moves from one correct state to another correct state
  - Isolation – concurrent transactions will not become entangled with each other
  - Durability – once a transaction has succeeded, the changes will not be lost.

Atomicity

To transfer money from 123 to 456.

```
UPDATE Accounts
SET balance = balance – 100
WHERE acctNo = 123;

UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;
```

Atomicity requires that both of the steps, or neither, be completed: all or none

Consistency in the context of ACID

Any data written to the database must be valid according to all defined rules, including but not limited to constraints and triggers, and any combination thereof.

Examples:

- Columns only store values of a particular type (int columns store only integers, etc…)
- Primary keys and unique keys are unique
- Check constraints are satisfied
- Foreign key (a.k.a referential integrity) constraints are satisfied
- In an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

Note that the notion of constancy in CAP theorem is different from that in ACID.

Isolation

- Deals with behavior of a transaction with respect to other concurrent transactions.
- Ensures **serializability** of concurrent execution of transactions - Operations may be interleaved, but execution must be equivalent to *some* serial order of all transactions.
- Providing isolation is the main goal of concurrency control

Isolation

```
SELECT seatNo
FROM Flights
WHERE fltNo = 123 AND
fltDate = DATE ‘2013-12-25’
AND seatStatus  = ‘available’;

UPDATE Flights
SET seatStatus = ‘occupied’
WHERE fltNo = 123 AND
fltDate = DATE ‘2013-12-25’
AND seatNO  = ‘22A’;
```

User 1 finds seat(22A) empty

User 2 finds seat empty

User 1 sets seat 22A occupied

User2 sets seat 22A occupied

time

Isolation

```
Lock flights;
SELECT @seat = min(seatNo)
FROM flights
WHERE fltNo = 123 AND
fltDate = DATE ‘2013-12-25’
AND seatStatus  = ‘available’;

UPDATE Flights
SET seatStatus = ‘occupied’
WHERE fltNo = 123 AND
fltDate = DATE ‘2013-12-25’
AND seatNO  = @seat;
Unlock flight;
```

User 1 finds seat empty
and sets seat 22A occupied

↓

User 2 finds
the seat 22A occupied

OR

User 2 finds seat empty
and sets seat 22A occupied

↓

User 1 finds
the seat 22A occupied

Durability

- Guarantees that transactions that have committed will survive any subsequent mal-functions.
- Example: If a flight booking reports that a seat has successfully been booked, then the seat will remain booked even if the system crashes.
- Write-Ahead Transaction Log: First write changes to a transaction log and then write the changes to the database.

Limitations of RDBMS

- Scalability problems
  - Transactions become difficult under heavy load.
  - Distributed relational databases (horizontal scaling of relational databases) account for distributed transactions which uses 2PC (two phase commit)
  - 2PC blocks introducing higher latency during partial failure.
- Impedance Mismatch
  - Difference between the relational model and your domain objects.
  - To create a properly normalized schema, you need joins.
  - ORM (e.g. Hibernate) needs extended memory requirements and increasingly unwieldy mapping code.

Impedance Mismatch

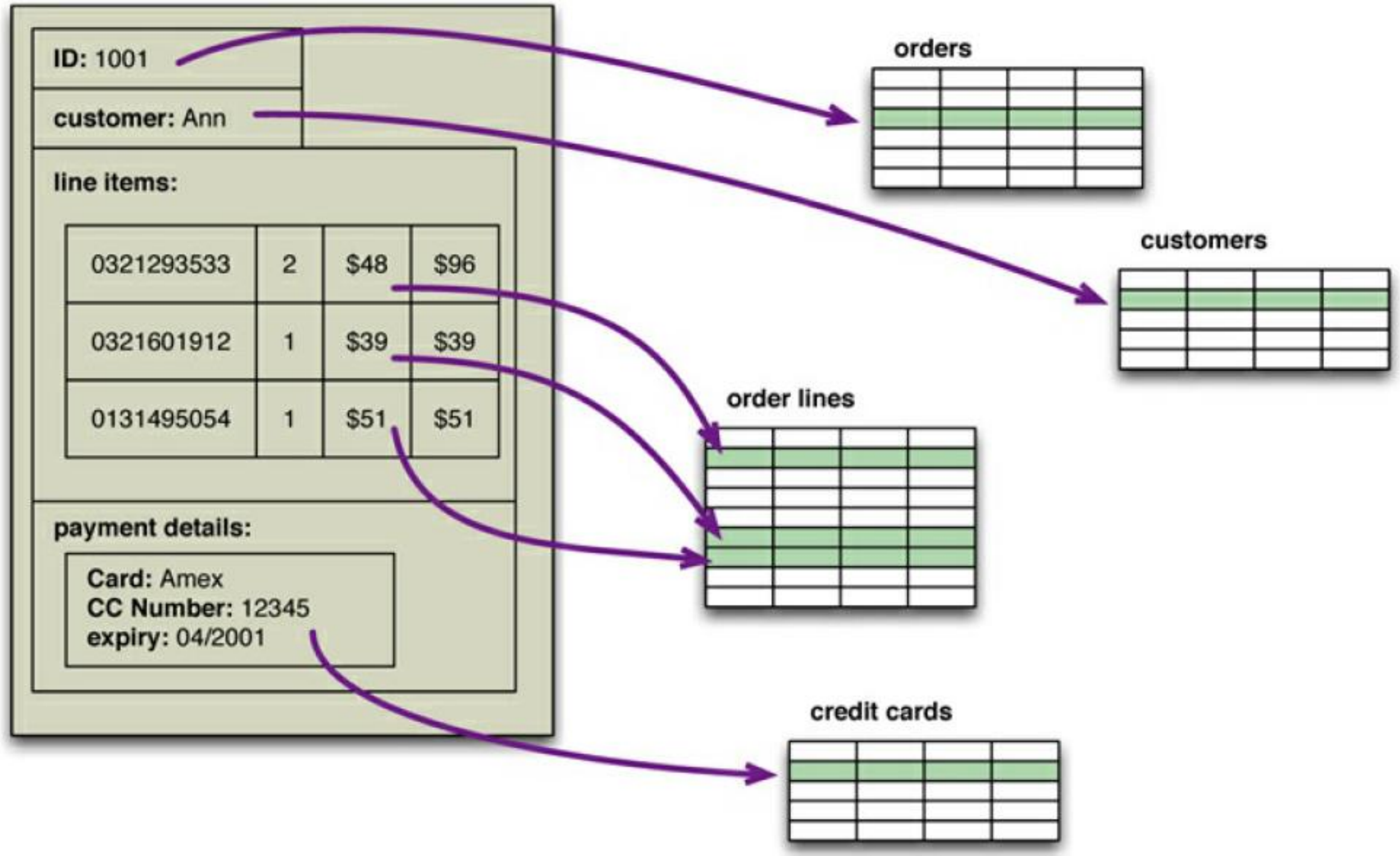


Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

Characteristics of NoSQL Systems

- Characteristic related to distributed databases and distributed systems
  1. Scalability
  2. Availability, Replication and Eventual consistency,
  3. Replication Models
  4. Sharding
  5. High-Performance Data Access
- Characteristics related to data models and query languages

  6. Not Requiring a Schema

  7. Less powerful Query Languages

  8. Versioning

http://www.slideshare.net/ateeqateeq/nosql-databases-62629001

1. Scalability

- Horizontal Scalability (employed by NoSQL systems)
  The distributed database system is expanded by adding more nodes for data storage and processing
- Vertical Scalability
  Expanding the storage and computing power of existing nodes

2. Availability, Replication and Eventual Consistency

- Data is replicated over multiple nodes in a transparent manner to support fault-tolerant and high availability mechanism.
- With replication, writes must be applied to every copy of the replicated data with potential inconsistency among reads.
- Eventual consistency: All updates will propagate through all of the replicas in a distributed system, but this may take some time. Eventually, all replicas will be consistent.

3. Replication Models

- Mater-slave replication

A master handles the master copy. All writes are applied to the master copy while slaves synchronize with the master and may handle reads.

- Peer-to-Peer replication

All the replicas have equal weight and they can all accept reads and writes.

4. Sharding

- Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
- A shard is a horizontal partition of data in a database. Each individual partition is referred to as a shard. Each shard is held on a separate database server instance, to spread load – Wikipedia
- Sharding improves both read and write performance.

5. High-Performance Data Access

- Hashing (as compared to sequential access)
  - A value can be found by hashing the partition key.
- Range partitioning (as compared to hash partitioning):
  - Selects a partition by determining if the partition key is inside a certain range. For example, a partition hold the data whose partition keys (e.g. zip codes) are in the rage of 7000 and 8000.
  - applications that requires range queries, range partition is preferred.

6. Not Requiring Schema

- It is not required to have a schema in most of the NoSQL systems
- Freedom and flexibility to store data
- Can deal with semi-structured and self describing data (e.g. JSON and XML)
- A schema-less database shifts the schema into the application code that accesses it. In application codes, there should present a set of logics to identify the structure of data to manipulate.

7. Less Powerful Query Languages

- SQL offers a rich set of DDL, DML, and DCL
- NoSQL systems provide a high-level query language, but queries themselves in NoSQL are not as powerful as SQL query language. For example, the joins need to be implemented in the application programs.

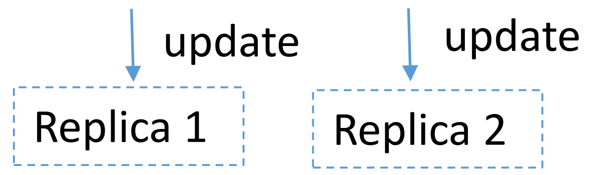
8. Versioning

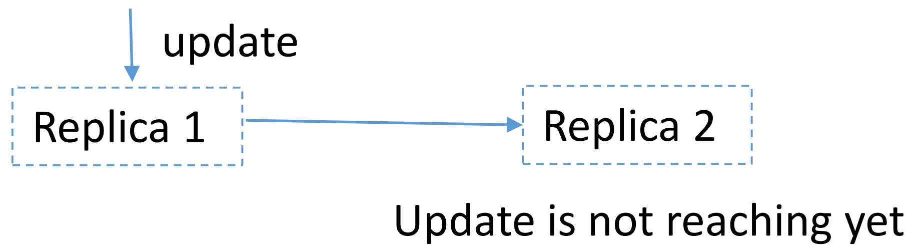
- Concurrency control methods: lock vs. versioning
- Version stamp (a.k.a vector clock, version vectors)
- Basic version stamp works well for a single server or a master-slave replication
- Example: conditional-update using etags of HTTP protocol.
  - An etag is a version stamp for a resource.
  - For each GET request for a resource, the server response with the etag of the resource in the header.
  - To update a resource, submit the etag from the last GET for the resource.
  - When the update request arrives, the server compares the etag from the client and that in the server. The update request is refused when they mismatch.

Version stamp for peer-to-peer replication

- A special form of version stamp, called vector stamp, is used in peer to peer NoSQL system

Example: scenarios resulting in inconsistency





- The vector stamp is a set of counters, one for each node.

Example: With nodes blue, green, and black in a cluster, each node holds a vector stamp like this [blue:43, green:54, black12].

Version stamp for peer-to-peer replication

How vector stamps are assigned?

1. Each time a node has an internal update, it updates its own counter.

Example: An update in the green node changes the vector of the green node to [blue:43, green:55, black 12].

2. When this update is sent to other nodes to synchronize other replicas, the version stamp of the update goes with it.

3. When a update is received, the receiving node updates its vector stamp by taking the maximum of each component of the received vector stamp and its current vector stamp.

Vector stamp for peer-to-peer replication

- ta < tb If and only if they meet two conditions:
  - they are not equal timestamps (there exists i, ta[i] != tb[i]) and
  - each ta[i] is less than or equal to tb[i] (for all i, ta[i] <= tb[i])
- One can tell if one version stamp is newer than another.
  - e.g. [blue:1, green:2, black:5] is newer than [blue:1, green:1, black 5]
- A write-write conflict presents if the order can't be determined.
  - e.g. [blue:1, green:2, black:5] v.s. [blue:2, green:1, black 5],
- In a case of missing values in a vector, the missing value is treated as 0. - This allows you to easily add a new node.
  - e.g. [blue: 6, black:2] would be treated as [blue:6, green:0, black:2]

Main NoSQL data models

Key-Value stores

Data model: A global collection of key-value pairs

Example: Amazon DynamoDB, Riak, Voldemort, and in-memory variants: MemcacheD, Redis

Column-family stores (a.f.k. wide-column stores)

Data model: A tabular model where each row can have an individual configuration of columns.

Example: Google’s Bigtable, Cassandra, Apache Hadoop’s HBase

Document stores

Data model: Collections of documents often stored in a format such as JSON, XML, or YAML.

Example: MongoDB, CouchDB

Graph-based

Data model: A network of nodes and edges that connect the nodes. Both nodes and edges can have properties in a form of key-value pairs

Example: Neo4J

Aggregate Data Models

- Data model: the model by which the database organizes data.
- NoSQL data model: key-value, document, column-family, and graph.
- Key-value, document and column-family – aggregate orientation

Aggregate

- A collection of related objects to be treated as a unit.
- A unit for data manipulation and management of consistency.
- Update aggregates with atomic operations and communicate with our data storage in terms of aggregates.
- A natural unit for replication and sharding – easier for database to handle operating on a cluster.
- Easier for application programmers to work with – impedance mismatch problem is resolved.

A relational data model for e-commerce web site

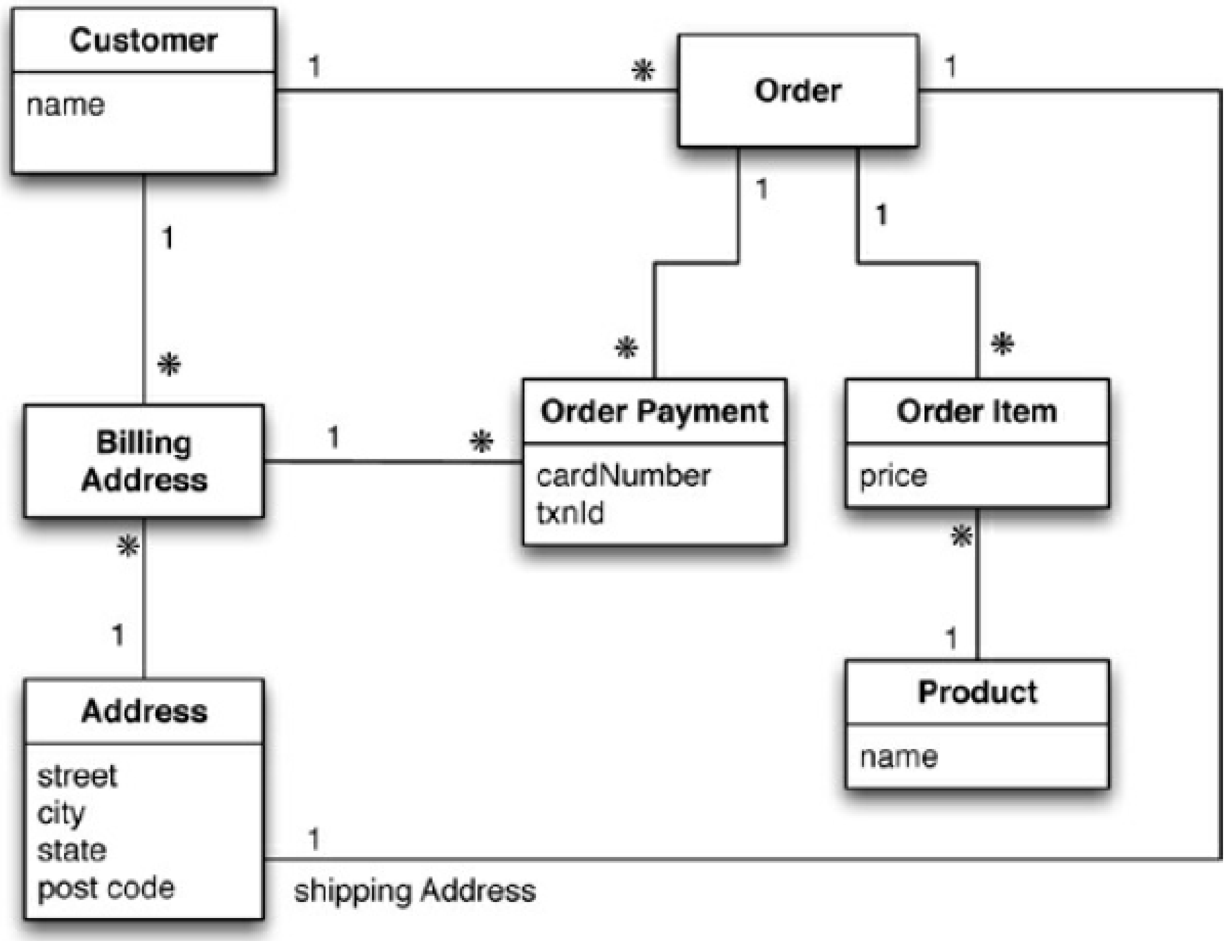
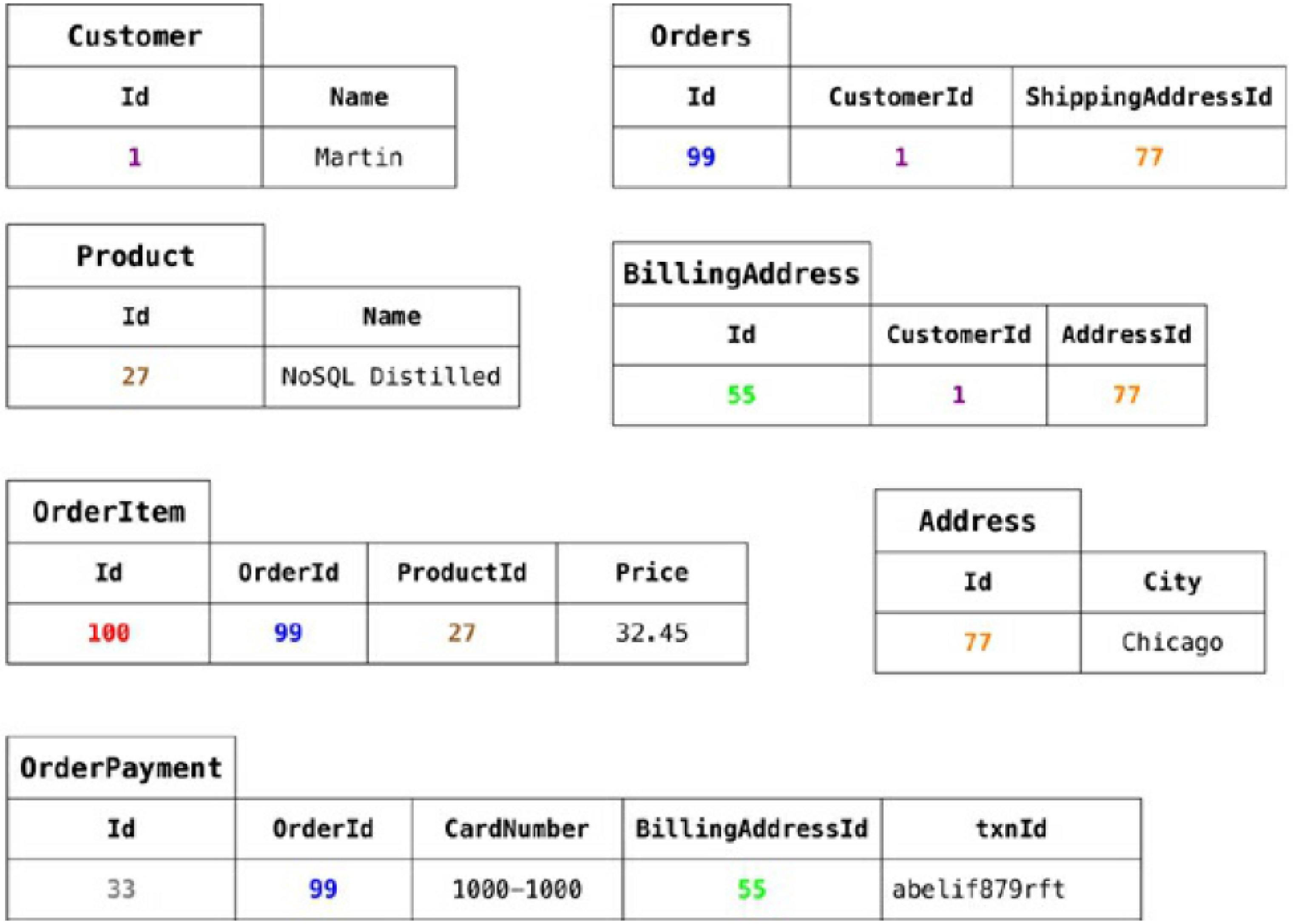


Figure 2.1. Data model oriented around a relational database using UML notation

Data representation in relational data models



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

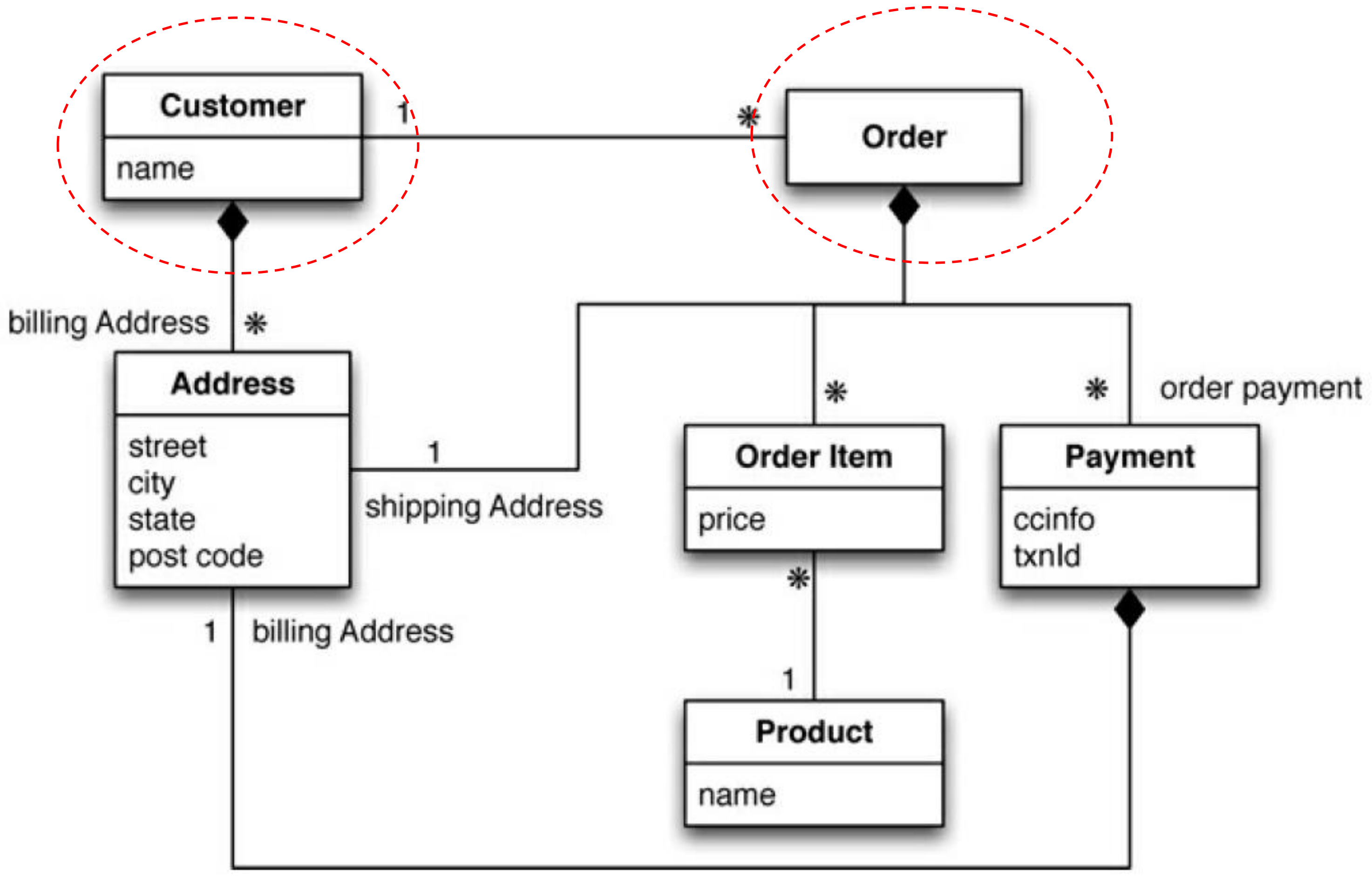
OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Normalized data

Referential Integrity

An aggregate data model and its data representation



```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Denormalized data: a single logical address record is copied three times.

An alternative aggregate data model and its data representation

Where to draw an aggregate boundary depends on the use cases of an application
minimize the number of aggregates to access during a data interaction.

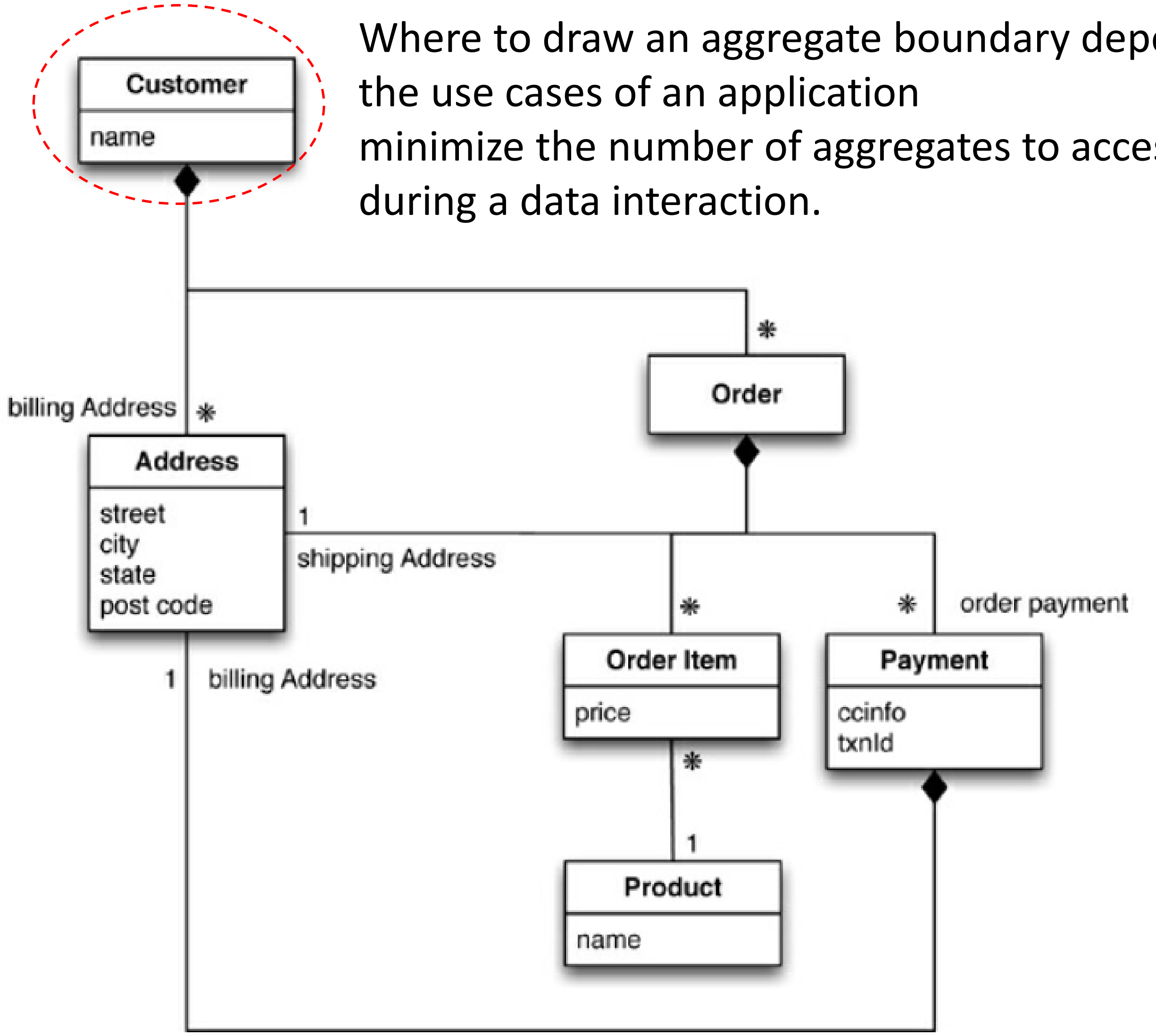


Figure 2.4. Embed all the objects for customer and the customer's orders

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
```

Relational vs. Aggregate data model

- Relational data model
  - Easily look at the data in different ways
  - ACID – real point is atomicity
  - The database cannot use a knowledge of aggregate structure to help to store and distribute the data.
- Aggregate data model
  - Helps greatly with running on a cluster – by explicitly including aggregates, the database knows which bits of data will be manipulated together, and thus should live on the same node.
  - Atomic manipulation of a single aggregate at a time.
  - Atomic manipulation of multiple aggregate has to be managed in the application code.

Key-Value and Document databases

- Key-value and document databases are strongly aggregate-oriented.
- In a key-value store, the aggregate is opaque to the database
  - The value is a big blob of mostly meaningless bits.
  - The aggregate can store whatever you like.
  - The value of an aggregate can be accessed as a whole based on its key.
- In a document store, the database is aware of the internal structure of aggregate.
  - More flexible access to an aggregate
    - We can retrieve part of the aggregate rather than the whole thing
    - The database can create indexes based on the contents of the aggregate.
  - The database imposes limits on what we can place in it, defining allowable structures and types.

Column-family databases

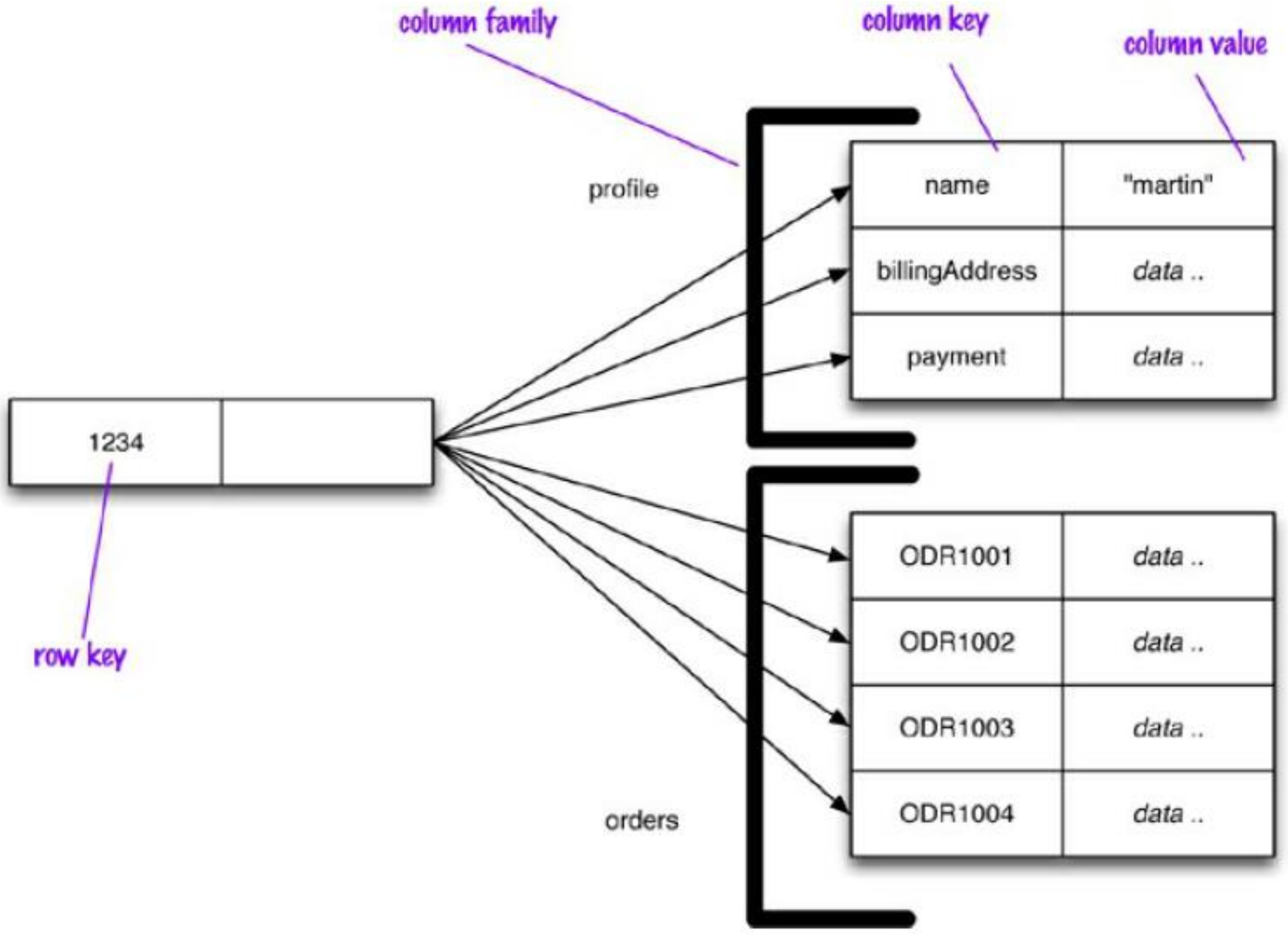


Figure 2.5. Representing customer information in a column-family structure

- The structure is a two-level map:

Row key → column key → column values

- The column acts as unit for access.
- Access control and both disk and memory accounting are performed as the column-family level.
- These terms are used in Google Bigtable and HBase. Cassandra uses these terms differently.

Graph databases (Aggregate Ingnorant)

- Aggregate databases are not ideal for capturing any data consisting of complex relationships such as social network.
- Relational databases can implement relationships using foreign keys – joins are required to navigate around but expensive.
- Graph data model
    - Stores small records with complex interconnection using nodes connected by edges.
    - Traveling along the relationship is cheap – the databases shift most of the work of navigating relationships from query time to insert time.
    - Emphasis on relationship
        - Most likely run on a single server rather than clusters
        - ACID transactions are used to cover multiple nodes and edges to maintain consistency.

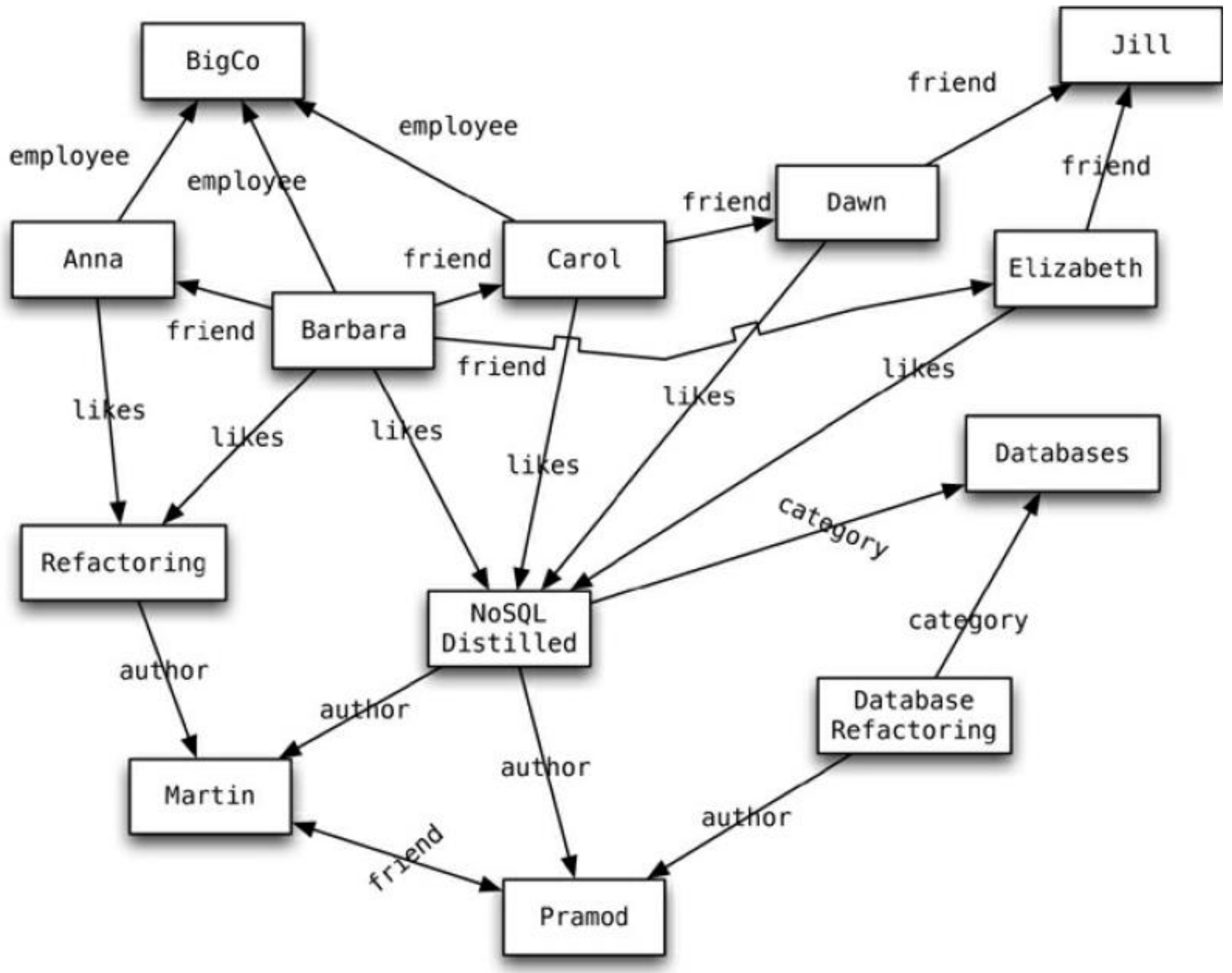


Figure 3.1. An example graph structure

Distribution Models

- Data distribution methods: sharding and replication
  - Sharding puts different data on different nodes
  - Replication takes the same data and copies it over multiple nodes
    - - Master-slave replication
    - - Peer-to-peer replication
  - You can use either or both of sharding and replication.

Sharding

- Putting different parts of the data onto different servers to support horizontal scalability

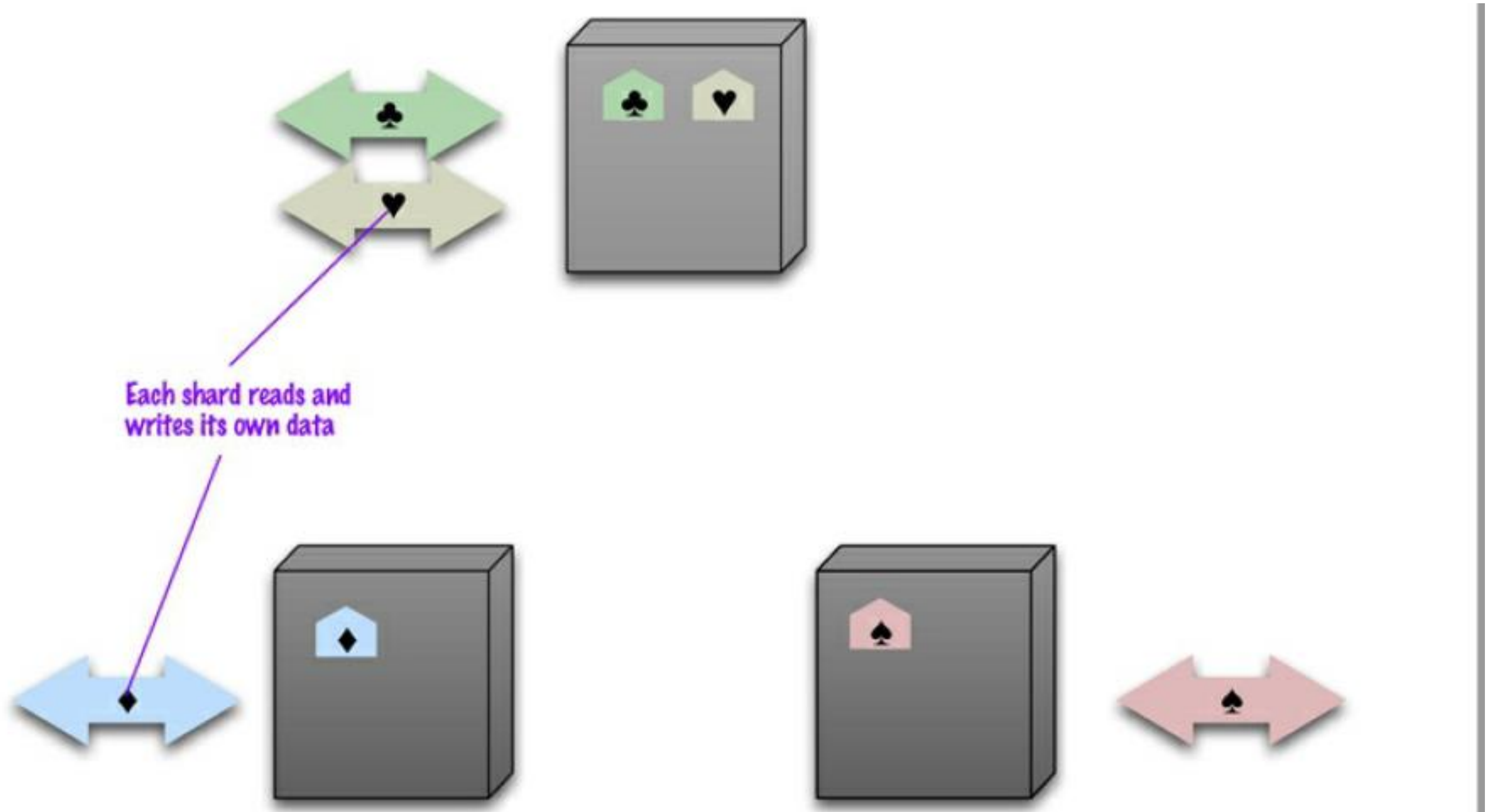


Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

Sharding

- Store data which are accessed together on the same node and assign the data to a node for the best data access
  - Use aggregates as an unit of distribution
  - If most accesses of certain aggregates are based on a physical location, place the data close to where it is being accessed
- Keep the load even
  - Distribute aggregate across the nodes in a way that they all get equal amount of the load.
- Auto-sharding: the database takes on the responsibility of allocating data to shards and ensuring the data access goes to the right shard.
- Improves both read and write performance

Master-Slave Replication

- One designated master node (or primary) is the authoritative source for the data and is responsible for processing any updates to that data. The other nodes are slaves (or secondaries).

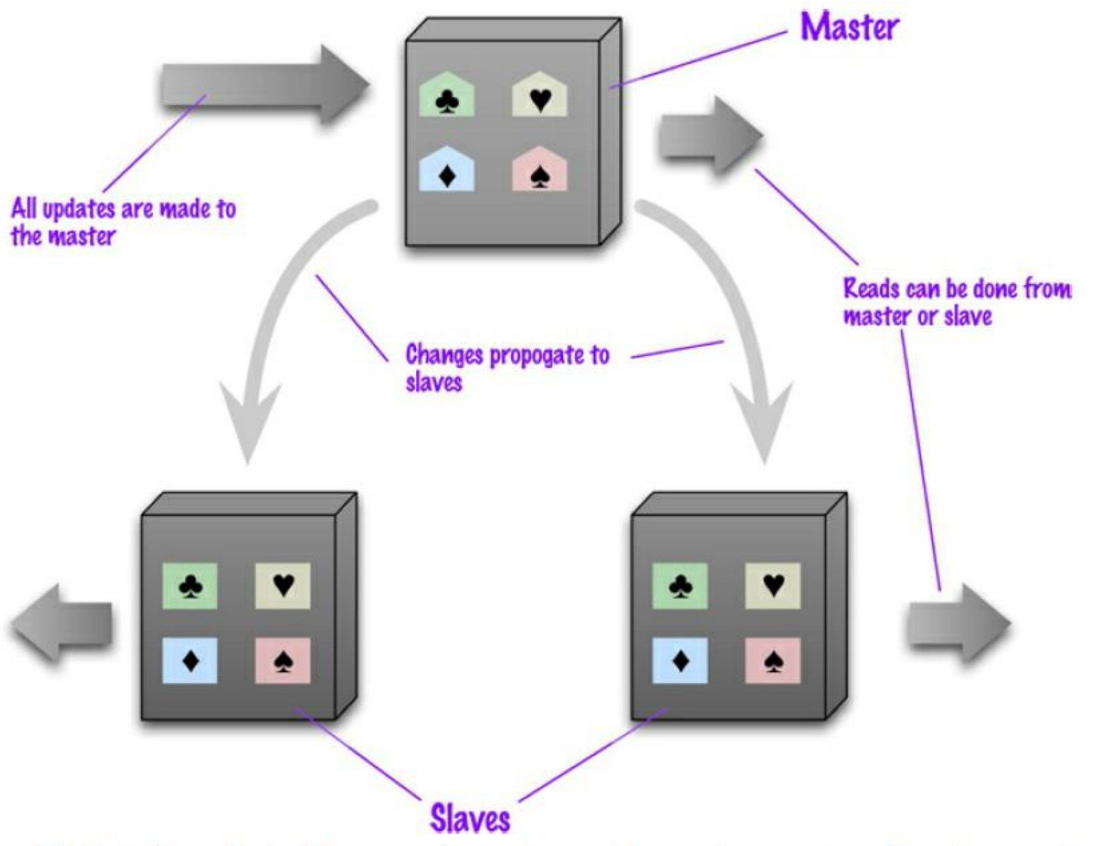

Master
All updates are made to the master
Reads can be done from master or slave
Changes propogate to slaves
Slaves

Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

Master-slave replication

- Good for a read-intensive datasets: horizontally scale read can be achieved by adding more slave nodes and ensuring all read requests are routed to the slaves.
- Not good for datasets with heavy write traffic: the master is solely in charge of updating data and passing the updates to slaves - the master is a bottleneck and a single point of failure.
- Read resilience: Although the master is down, slaves can still handle read requests.
- Slaves can serve as hot backup – a slave can be appointed as new mater quickly.
- Complication - consistency

Peer-to-Peer Replication

All the replicas have equal weight and they can all accept reads and writes.

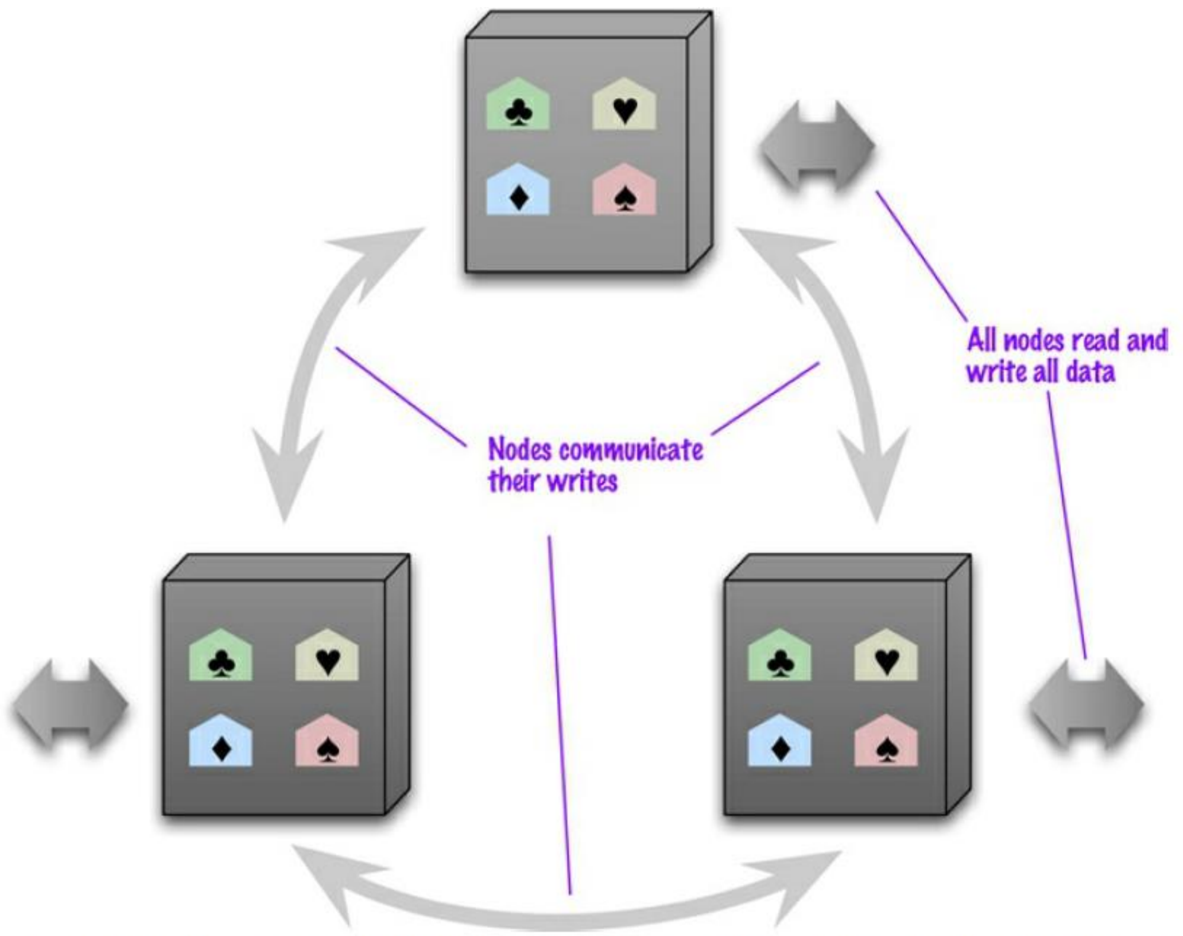

All nodes read and write all data
Nodes communicate their writes

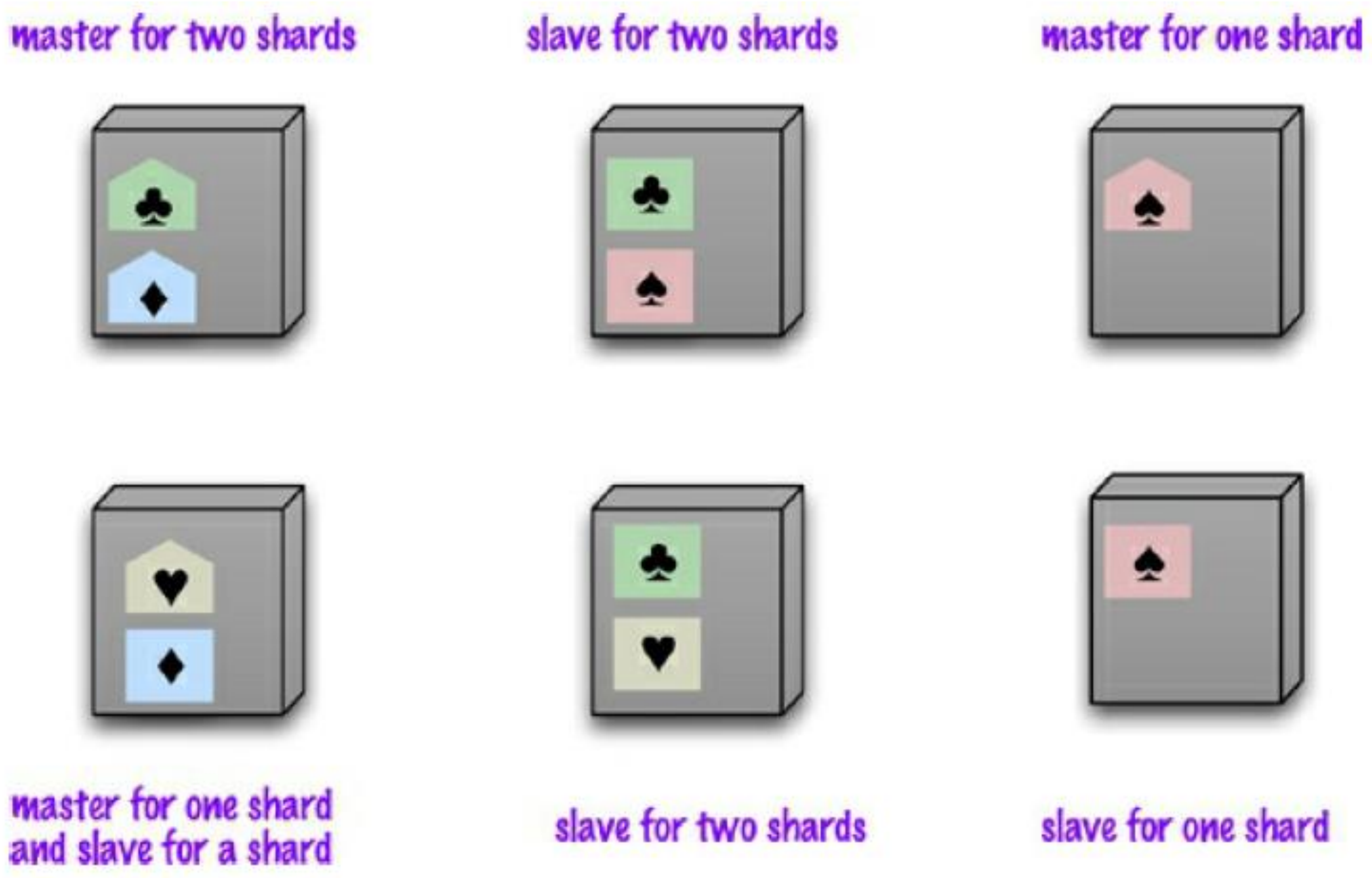
Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

Peer-to-Peer Replication

- Ride over node failures without losing access to data
- Easily adding nodes to improve performance
- Complication – consistency
  - Concurrent writes update the same record (two replicas representing the same data) at the same time
- How to deal with write inconsistency
  - For a write request, the replicas coordinate to ensure to avoid a conflict at the cost of network traffic.
  - Cope with an inconsistent write.

Master-slave replication + Sharding

Multiple masters but each data item only has a single master.



Peer-to-peer replication (RF = 3) + Sharding

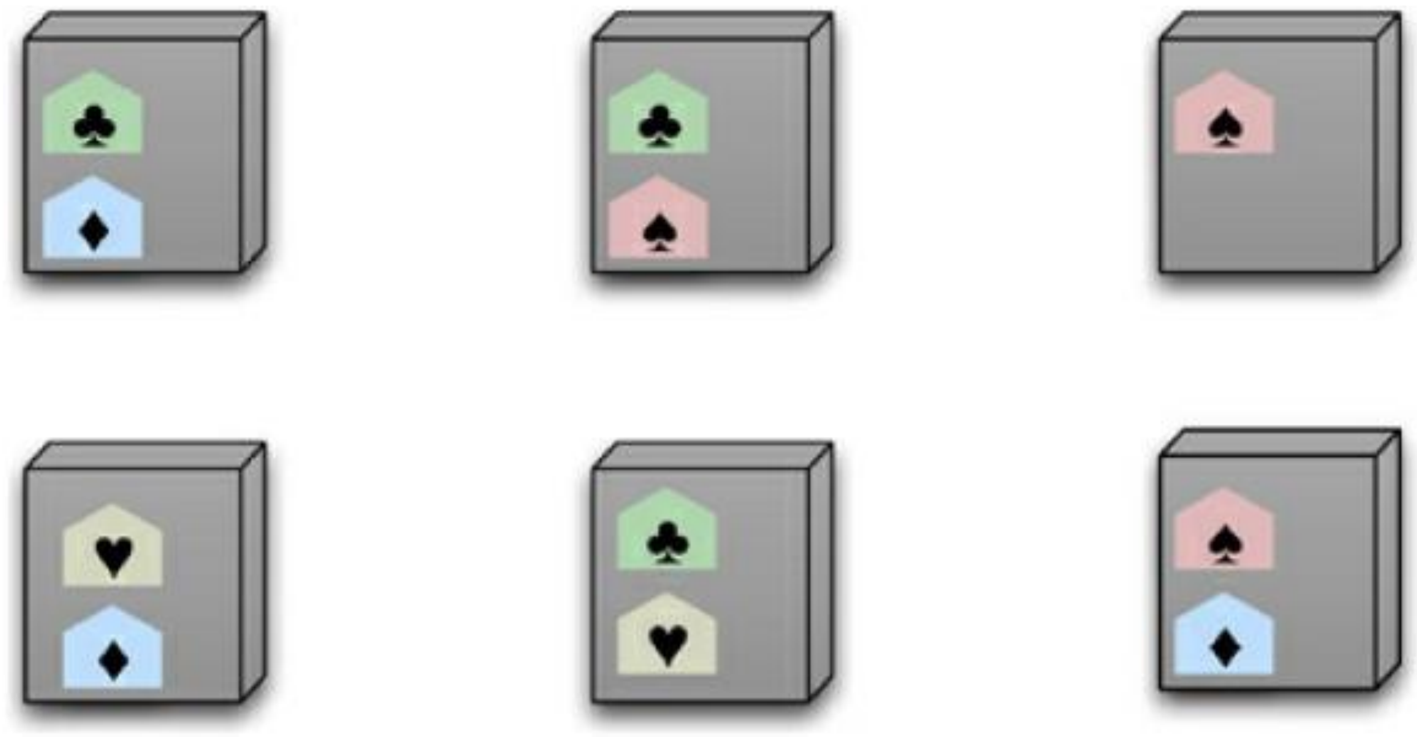


Figure 4.5. Using peer-to-peer replication together with sharding

Update Consistency

- Write-write conflict: two people updates the same data item at the same time.
- Without concurrency control, there may present a lost update.

A	B
Read X	Read X
Write X	
	Write X

A’s update will be lost.

- How to maintain consistency?
  - Pessimistic approach: preventing conflicts from occurring – e.g. using locks
  - Optimistic: lets conflicts occur, but detects them and takes action to sort them out e.g. conditional update or saving both updates and merge them somehow
  - Trade off – safety (avoiding conflicts) and liveness (responding quickly to clients)
- With replication, there present more write-write conflict.
  - Master-slave replication < Peer-to-Peer replication

Read Consistency

- Read-write conflict in logical consistency: ensuring different but logically related items make sense together.

Without Replication

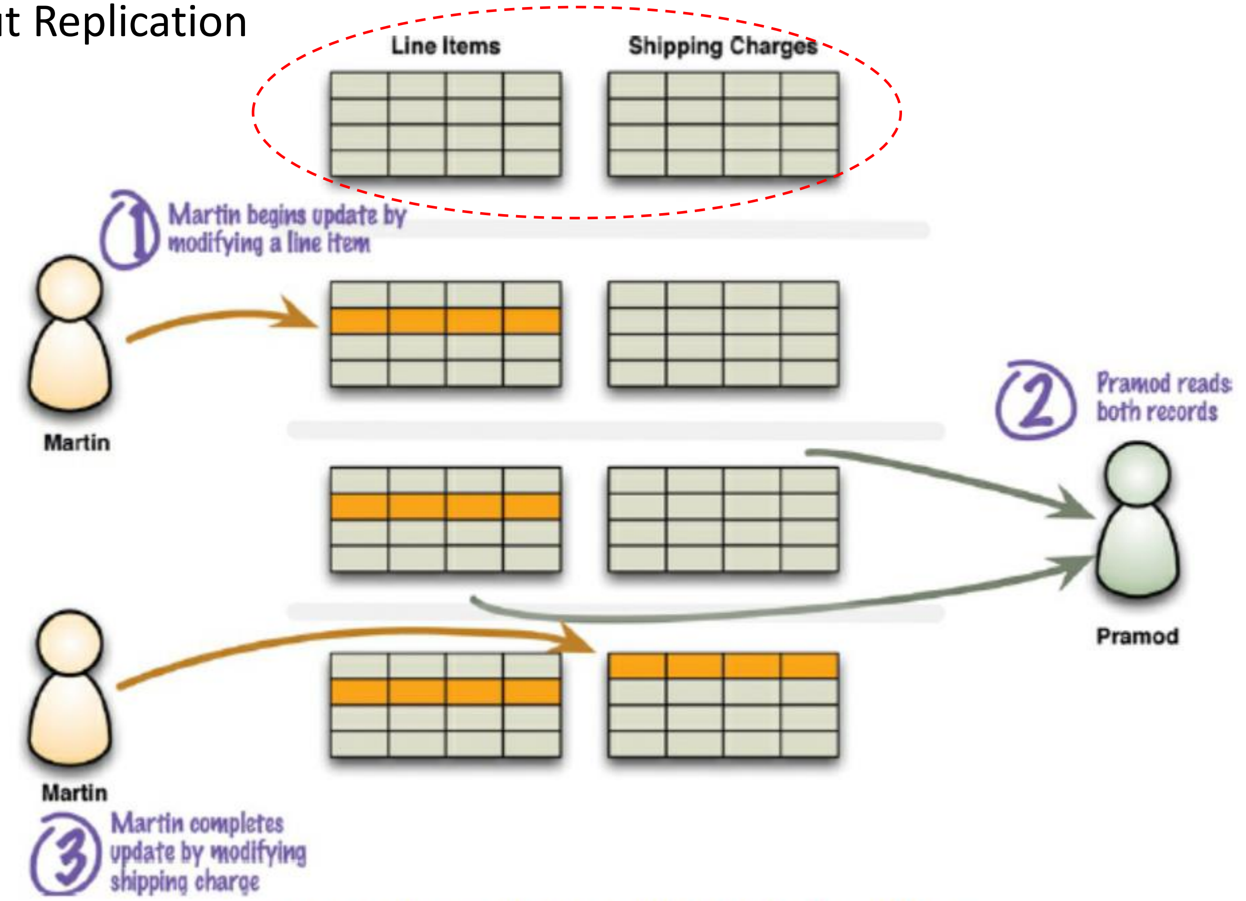

Line Items
Shipping Charges
1 Martin begins update by modifying a line item
Martin
2 Pramod reads both records
Pramod
Martin
3 Martin completes update by modifying shipping charge

Figure 5.1. A read-write conflict in logical consistency

- In RDBMS, put two updates of M in a transaction, to sure that P will either read both data items before the update or both after the update.
- In NoSQL,
  - Graph databases support ACID transactions.
  - Others support atomic updates, but only within a single aggregate.

Read Consistency

- Replication consistency: ensuring that the same data item has the same value when read from different replicas.

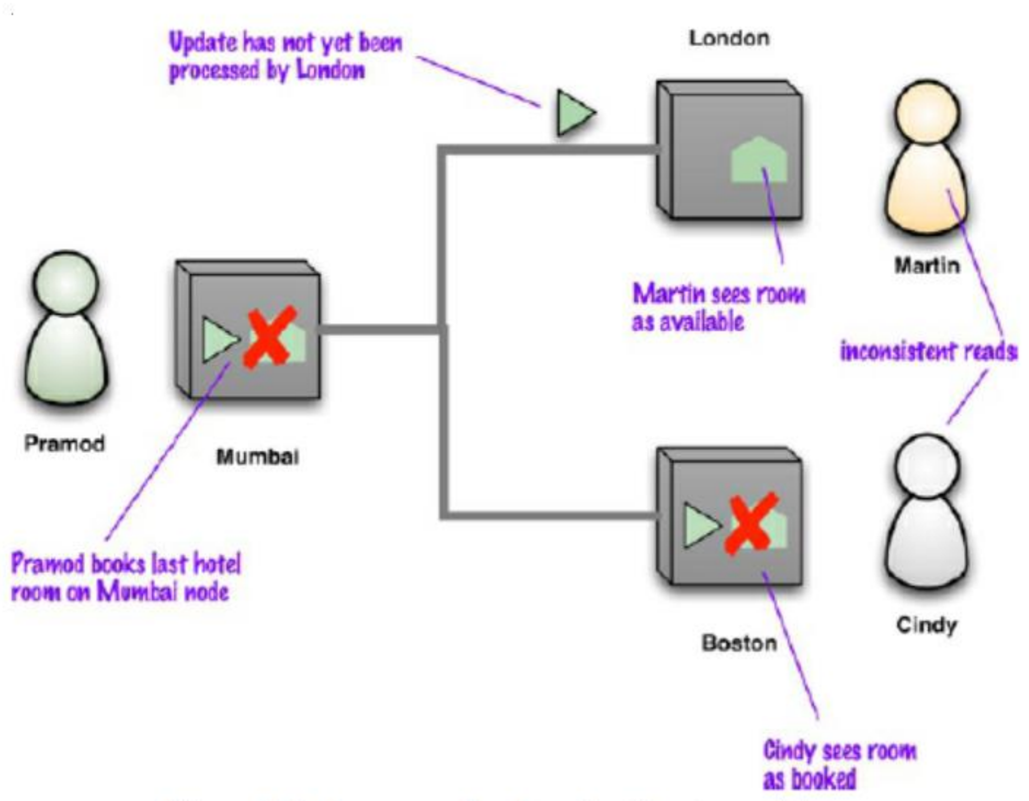

Update has not yet been processed by London
London
Martin
Martin sees room as available
inconsistent reads
Pramod
Mumbai
Pramod books last hotel room on Mumbai node
Boston
Cindy
Cindy sees room as booked

Figure 5.2. An example of replication inconsistency

Eventual consistency:
at any time nodes may have replication inconsistencies but, if there are no further updates, eventually all nodes will be updated to the same value.

Relaxing consistency

- Different isolation levels in relational database systems
- CAP theorem in NoSQL database systems.

Brewer’s CAP Theorem

- Within a large-scale distributed data system, there are three requirements to consider: Consistency, Availability, and Partition tolerance. The theorem states that you can strongly support only two of the three in any given system.
  - Consistency: consistent reads and writes so that concurrent operations see the same valid and consistent data state.
  - Availability: If you can talk to a node (non-failing node) in the cluster, it can read and write data.
  - Partition tolerance: The system is functional in spite of network partition.
- A distributed data system cannot give up the partition tolerance property, leaving us only two real options to compromise on: availability and consistency

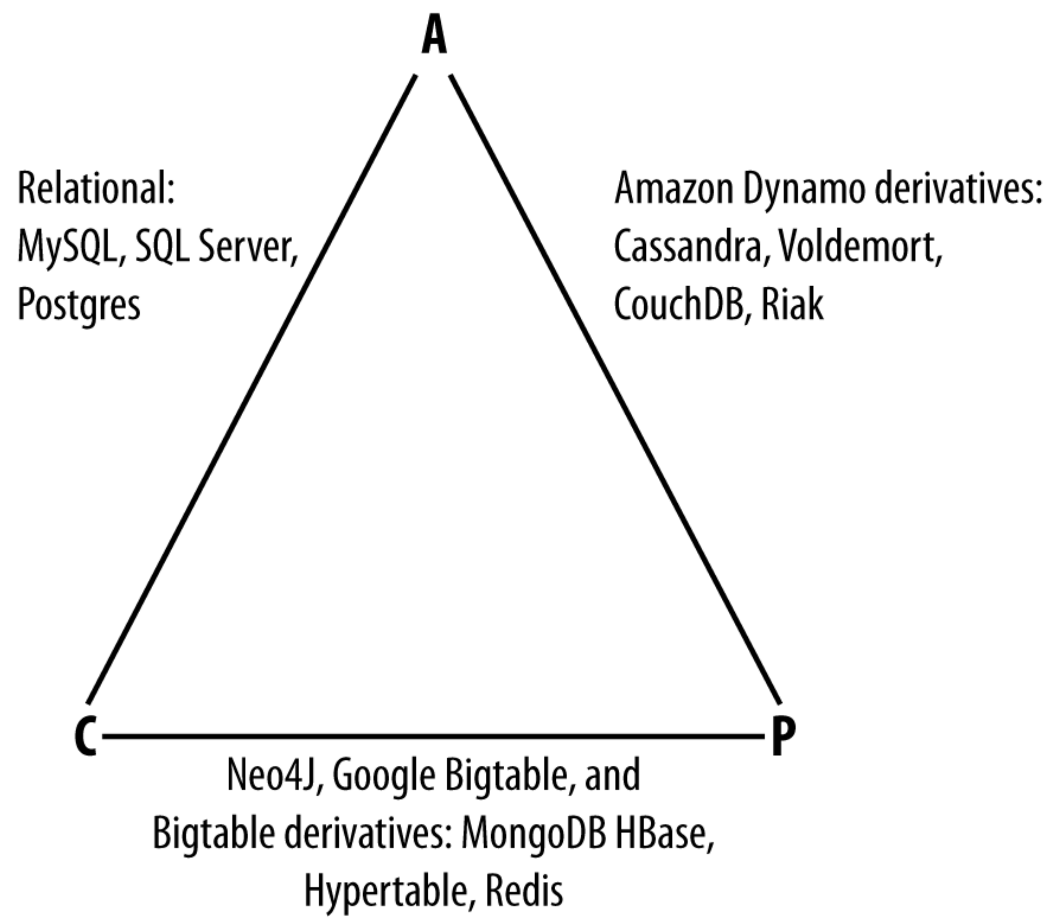


Figure 2-2. Where different databases appear on the CAP continuum

- It is not strictly precise.
- Not binary decision
- Trade off a little consistency to get some availability
- Practically, trade off between consistency and latency.

CA Systems

- A single server system
  - Single machine can’t partition – no worry about partition tolerance
  - Only one node – if it is up, it is available
  - Most of relational database systems
- A distributed CA system
  - Mostly likely it adopts two-phase locking and two-phase commit for distributed transaction to support consistency resulting a longer latency

- CP
  - Example - Master-slave replication
  - All writes are done at the master node. There may be a short read inconsistency window at slave nodes.
  - If the master goes down, a non-failing slave node is consider to be unavailable because one can talk to the slave node but cannot write on it.
- AP
  - Example: Peer-to-peer replication
  - A node can take writes and reads - update and replication consistencies present.
  - How to resolve inconsistencies depends on the application.

    Example from Amazon Dynamo paper: You are always allowed to write, even if network failures mean you end up with multiple shopping carts. The check out process can merge them into a single card and return it.

Note: Consistency in CAP

- Consistency in the context of CAP alludes to atomicity and isolation.
- In ACID, consistency means that data that does not satisfy predefined constraints is not persisted. That is not the same as the consistency in CAP.