UNIT-5 CHAPTER-1

AGGREGATE DATA MODELS

Definitions:

Data Model vs. Storage Model

- A data model is how we perceive and manipulate data in a database.
- It describes how users interact with data.
- A storage model is how the database stores and manipulates data internally.
- Ideally, users don't need to know the storage model, but in reality, some understanding helps with performance.

Meaning of "Data Model"

- Often refers to the specific data structure in an application (e.g., an entity-relationship diagram showing customers, orders, and products).
- In this book, "data model" refers to how the database organizes data, also called a metamodel.

Relational Data Model (Dominant for Decades)

- Best visualized as tables, like a spreadsheet.
- Each table has rows (each row represents an entity).
- Columns describe entity attributes, each holding a single value.
- A column may reference another row in the same or a different table, forming relationships.
- Formal terms:
    - Tables → Relations
    - Rows → Tuples

NoSQL Databases

- One of the most obvious shifts with NoSQL is a move away from the relational model.
- Each NoSQL solution has its own data model.
- NoSQL data models fall into four main categories:
    1. Key-Value Stores
    2. Document Stores
    3. Column-Family Stores
    4. Graph Databases

Aggregate Orientation in NoSQL

- Key-Value, Document, and Column-Family models share a key characteristic: ***aggregate*** orientation.

- This affects how data is organized and structured in NoSQL databases.

Aggregates:

Relational Model and Its Limitations

- In the **relational model**, data is stored as **tuples (rows)** in tables.
- A **tuple** is a simple structure that captures a set of values.
- **Limitations of tuples:**
  - You **cannot nest** one tuple inside another.
  - You **cannot store lists** or other tuples inside a single tuple.
- Because of this, all operations in relational databases work with and return **tuples only**.

What is Aggregate Orientation?

- Unlike relational databases, some databases **group related data together** in a more complex structure.
- These databases allow **nesting**, where a record can contain lists or other records inside it.
- This concept is called **aggregate orientation**.
- **Key-value, document, and column-family databases** use this approach.
- There is no standard term for this complex data structure, but in this book, it is called an **"aggregate."**

Origin of the Term "Aggregate"

- The term comes from **Domain-Driven Design (DDD)** (by Eric Evans).
- In DDD, an **aggregate** is a **collection of related objects** that we treat as a **single unit**.
- Aggregates help in:
  1. **Data manipulation** – We update and retrieve aggregates as a whole.
  2. **Ensuring consistency** – Changes within an aggregate remain consistent.

Advantages of Aggregates in NoSQL Databases

1. **Atomic Operations** – Updates are made to the entire aggregate at once.
2. **Easier Cluster Management** – Aggregates help in:
   - **Replication** (copying data across servers).
   - **Sharding** (distributing data across multiple servers).
3. **Better for Developers** – Application programmers find aggregates easier to work with since they match how applications naturally structure data.

Example of Relations and Aggregates

To better understand **relations vs. aggregates**, let's take an example.

Scenario: E-Commerce Website

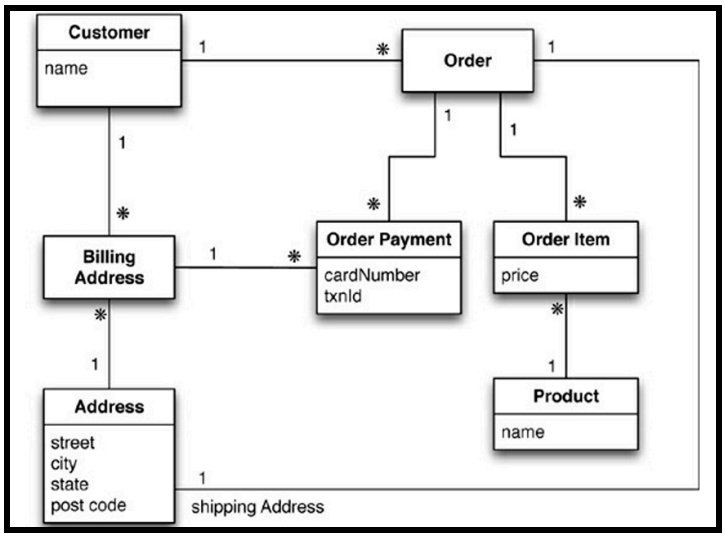
We are building an **e-commerce website** where customers buy products online.
We need to store data about:

- **Users**
- **Product catalog**
- **Orders**
- **Shipping addresses**
- **Billing addresses**
- **Payment details**

We will compare **relational databases** and **NoSQL databases** by modeling this data in both ways.

Relational Model Approach

- In a **relational database**, we organize data into **separate tables** (relations).
- Each table stores **specific types of data**.
- For example:



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

Figure 2.2. Typical data using RDBMS data model

Relationships are used to connect data.

- Example: The **Orders** table will have a **User ID** column linking to the **Users** table.
- The **Shipping Address** table will link to both **Orders** and **Users**.

Pros of Relational Model:

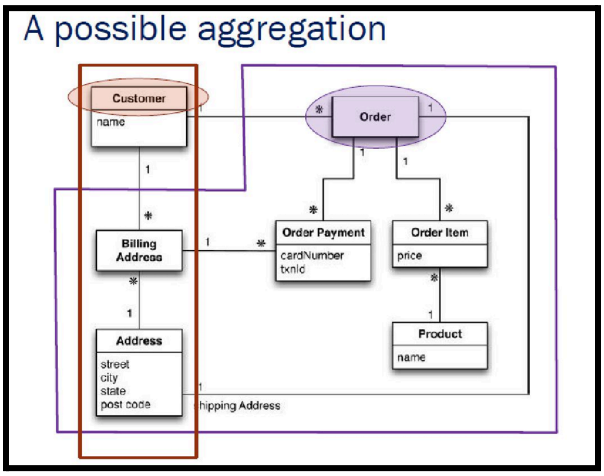
✔ Data is **structured and normalized**, reducing redundancy.
✔ **ACID transactions** ensure data integrity.

Cons of Relational Model:

❌ **Complex joins** are needed to fetch related data (e.g., getting a user's order along with shipping details).
❌ **Scalability is harder** because queries involve multiple tables.

NoSQL (Aggregate) Model Approach

- In a **NoSQL database**, we structure data as **aggregates** instead of separate tables.
- An **aggregate** is a collection of related data stored as a single unit.
- In an **aggregate-oriented model (NoSQL)**, we **combine related data** into **single units (aggregates)**.
- **Example: A customer's order and all related data (items, payments, shipping, etc.) can be stored together.**
- This removes the need for **complex joins**, improving **read performance** and **scalability**.



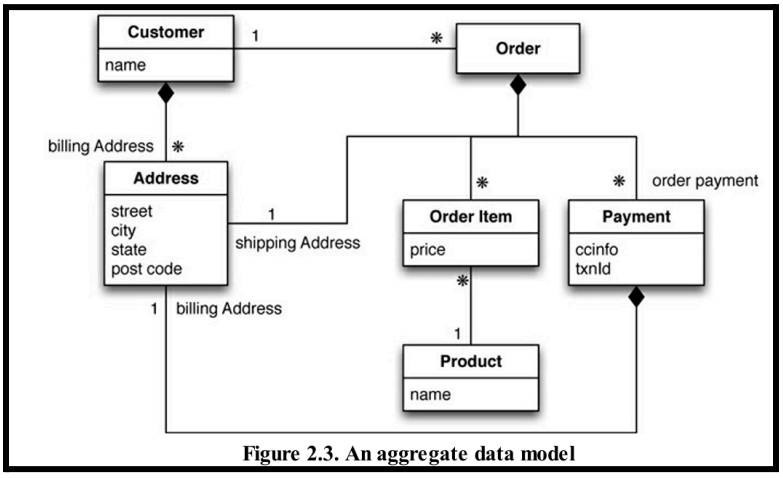


Figure 2.3. An aggregate data model

- **Black-Diamond Composition Marker (UML Notation)**
  - The black-diamond in the diagram represents composition (strong ownership).
  - This means the order cannot exist without its items, payments, and addresses—they are tightly bound together.
- **Two Main Aggregates in the Model:**
  - Customer Aggregate:
    - Contains a list of billing addresses.
  - Order Aggregate:
    - Contains a list of order items, a shipping address, and payments.
    - Each payment also contains a billing address.
  - The **Customer and Order** are stored **separately as aggregates** instead of being linked. **Why?**
    - We don't always need **customer details** when working with orders.
    - Orders can be processed independently from customer updates.
    - **When a user views an order**, they need:
    - Order details
    - Order items
    - Shipping address
    - Payment status
    - **When a user updates their profile**, it affects:
    - Customer name
    - Billing addresses

    - Since **Orders and Customers are accessed separately**, it makes sense to **keep them as separate aggregates**.
- The provided JSON structures represent **two separate aggregates: Customer and Order**. Each aggregate groups related data together, reducing the need for joins and optimizing read operations:

```
// 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"
      }
    }
  ]
}
```

- **Separate Aggregates for Customers and Orders**

Customers change over time, but orders must stay the same for historical accuracy.
Orders reference customers only by ID (`customerId`) instead of embedding full customer details.

- **Data Duplication for Faster Reads**

Billing addresses are duplicated in both:
1.The Customer aggregate (latest billing address).
2.The Order aggregate (billing address used for that order).
Product details (name, price) are stored directly in the order to avoid costly joins with a Product table.

- **No Joins, Just IDs Between Aggregates**

Instead of using joins (like in SQL databases), we use `customerId` in the Order to link it to the Customer aggregate.
This keeps reads fast, as an order does not need to fetch full customer details every time.

Why is this a good aggregate design?

- Orders remain unchanged after creation.
- Customers can update their billing address without affecting past orders.
- Products in orders don't need real-time updates, avoiding costly joins.
- Data duplication is intentional for performance and scalability.

Pros of NoSQL Model:

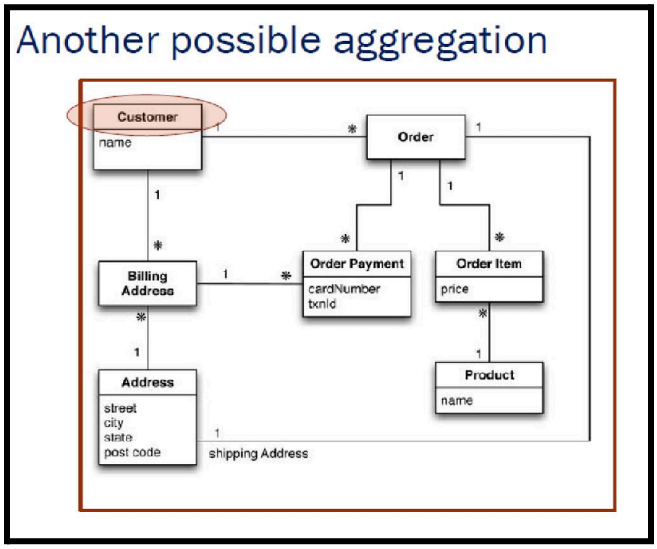
✔ **Faster reads** – No need for complex joins; all related data is stored together.
✔ **Easier scalability** – Data can be distributed across multiple servers easily.

Cons of NoSQL Model:

❌ **Redundancy** – The same information (e.g., shipping address) may be stored multiple times.
❌ **Consistency issues** – Updating data across different aggregates can be challenging.

Why Use the Aggregate Model?

1. **Faster Queries** – Since an **entire order and its related data are stored together**, fetching data is much faster than querying multiple relational tables.
2. **Better Performance at Scale** – Since we avoid **expensive JOIN operations**, NoSQL databases can handle **large-scale applications** more efficiently.
3. **Simplified Application Code** – Instead of writing **complex SQL joins**, application logic becomes easier to manage.
4. **Efficient Data Distribution** – Aggregates make it easier to **replicate** and **shard data** across multiple servers.



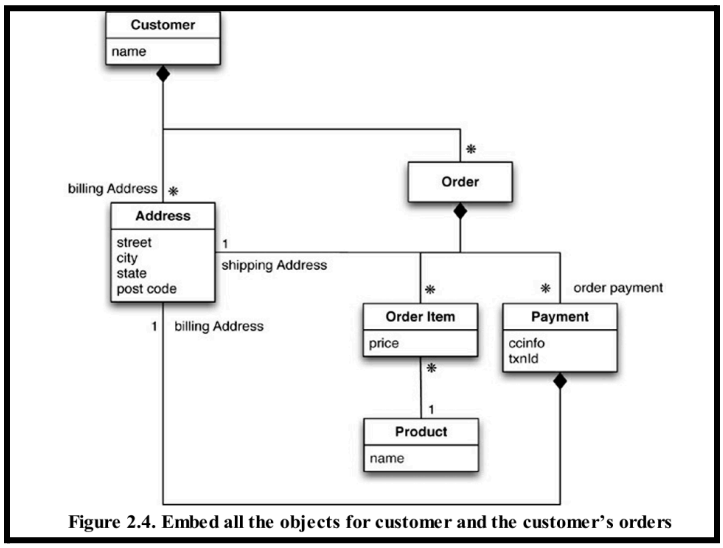


Figure 2.4. Embed all the objects for customer and the customer's orders

The diagram represents **another possible aggregation approach** for an **e-commerce system**. The diagram represents an **embedded aggregation model**, where all objects related to a **customer and their orders** are stored **inside the same aggregate**. This means that instead of storing orders separately and linking them with a `customerId`, **each customer's orders are embedded within their document**.

Customer Aggregate (Top-Level Object)

- Contains **customer details** such as `name`.
- **Billing addresses** are embedded within the customer aggregate.

Orders Embedded Within Customer

- Each customer contains a **list of orders**.
- Each **order** contains:
    - **Order Items**

- **Shipping Address**
- **Order Payments**

Order Items Contain Product Details

- Instead of referencing a separate **Product table**, the order stores:
  - `productId`
  - `productName`
  - `price`

Payments Embedded Inside Order

- **Payments include billing addresses**, ensuring consistency.
- Transactions (`txnId`) and card details (`ccinfo`) are **stored within the order itself**.

```
// 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"}
          }
        ]
      }
    ]
  }
}
```

Consequences of Aggregate Orientation:

Understanding Aggregate Orientation vs. Relational Mapping

- **Relational databases** use **tables with foreign keys** to capture relationships between different entities (e.g., orders, payments, addresses).
- However, **relational databases do not recognize aggregate entities**—they treat all relationships equally.
- In contrast, **aggregate-oriented databases (NoSQL)** group related data into **self-contained aggregates** (like storing an order along with its items, shipping details, and payment together).
- This **aggregate structure** is important because it affects **how the data is stored, queried, and distributed** in NoSQL systems.

Aggregate-Ignorant vs. Aggregate-Aware Databases

- **Relational databases have no concept of aggregate within their data model, so we call them aggregate-ignorant.**
- **Relational databases** are **aggregate-ignorant** because they treat **all relationships the same**.
- **Graph databases** are also **aggregate-ignorant** since they allow flexible connections between data without enforcing an aggregate boundary.
- **NoSQL aggregate-oriented databases** explicitly define aggregates, helping with **scalability, query efficiency, and distributed storage**.
- An aggregate-ignorant model allows you to easily look at the data in different ways, so it is a better choice when you don't have a primary structure for manipulating your data.

When are Aggregate-Ignorant Databases Useful?

- Some queries **don't fit well into aggregate models**.
- **Example:** If a retailer wants to analyze product sales over the past few months,
    - In an **aggregate model**, the sales data is buried inside thousands of order aggregates, making it **hard to query**.
    - A **relational model** (or an aggregate-ignorant approach) allows easier querying of **product sales across multiple orders**.

✔ **Aggregates help optimize queries for specific use cases (like fetching an order quickly).**
❌ **They can be inefficient for analytical queries that require scanning across many aggregates**

Aggregates and Distributed Databases (NoSQL Clusters)

- One major reason for using aggregates in NoSQL databases is efficient scaling across clusters.
- Why?
    - In a distributed system, data is spread across multiple servers (nodes).
    - If related data (e.g., orders and payments) is stored separately across multiple nodes, fetching a complete order requires querying multiple servers.
    - Aggregates minimize node queries by storing related data together on the same node, making data retrieval much faster.

✅ **By explicitly defining aggregates, NoSQL databases can store related data on the same server, reducing query time.**

Impact of Aggregates on Transactions (ACID vs. NoSQL)

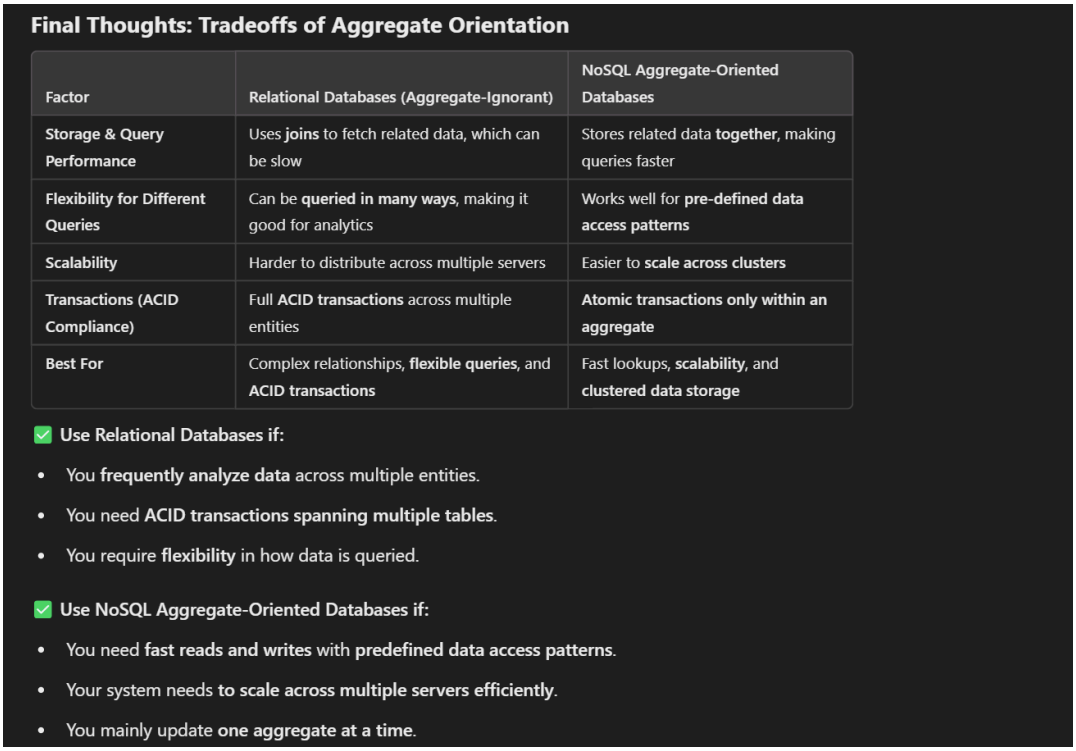
- Relational databases fully support ACID transactions:
  - **Atomic** – Either all updates succeed or none do.
  - **Consistent** – Data remains valid before and after transactions.
  - **Isolated** – Transactions do not interfere with each other.
  - **Durable** – Once committed, data remains safe.
- In relational databases, ACID transactions can span multiple tables, meaning you can update customers, orders, and payments at the same time atomically.

What About NoSQL Databases?

- NoSQL aggregate-oriented databases do not support ACID transactions across multiple aggregates.
- Instead, they ensure atomicity only within a single aggregate.
- If an application needs to update multiple aggregates together, it must handle consistency manually in the application code.

Why is This Not a Big Issue?

- Most operations naturally fit within a single aggregate.
- For example, updating an order (items, payment, shipping) can be done atomically inside one aggregate.
- However, if we need to update both an order and a customer's billing address together, we cannot do this atomically in an aggregate-based NoSQL model.



Final Thoughts: Tradeoffs of Aggregate Orientation

Factor	Relational Databases (Aggregate-Ignorant)	NoSQL Aggregate-Oriented Databases
Storage & Query Performance	Uses **joins** to fetch related data, which can be slow	Stores related data **together**, making queries faster
Flexibility for Different Queries	Can be **queried in many ways**, making it good for analytics	Works well for **pre-defined data access patterns**
Scalability	Harder to distribute across multiple servers	Easier to **scale across clusters**
Transactions (ACID Compliance)	Full **ACID transactions** across multiple entities	**Atomic transactions only within an aggregate**
Best For	Complex relationships, **flexible queries**, and **ACID transactions**	Fast lookups, **scalability**, and **clustered data storage**

✅ **Use Relational Databases if:**

- You **frequently analyze data** across multiple entities.
- You need **ACID transactions spanning multiple tables**.
- You require **flexibility** in how data is queried.

✅ **Use NoSQL Aggregate-Oriented Databases if:**

- You need **fast reads and writes** with **predefined data access patterns**.
- Your system needs **to scale across multiple servers efficiently**.
- You mainly update **one aggregate at a time**.

Key-Value and Document Data Models:

- Key-value and document databases are strongly aggregate-oriented.
- These databases are primarily constructed through aggregates.
- Both of these types of databases consist of lots of aggregates with each aggregate having a key or ID that's used to get at the data.
- Each **aggregate** (a self-contained unit of data) is assigned a **key or ID**, which is used to access it.

Key Differences Between Key-Value and Document Databases

Key-Value Databases: Aggregates Are "Opaque" (Unstructured)

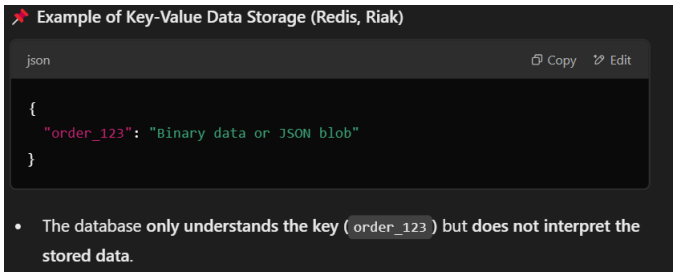
- In **key-value databases**, the database **does not understand** what is inside an aggregate—it simply treats it as a **big, unstructured blob of data**.
- **Example:**
  - Imagine a **locker system** where each locker has a **key**.
  - The database does **not care what's inside the locker**—it just knows how to open it using the key.

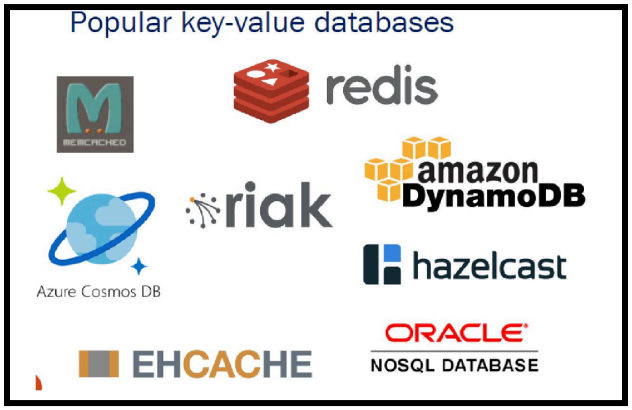
Advantages of Key-Value Databases:

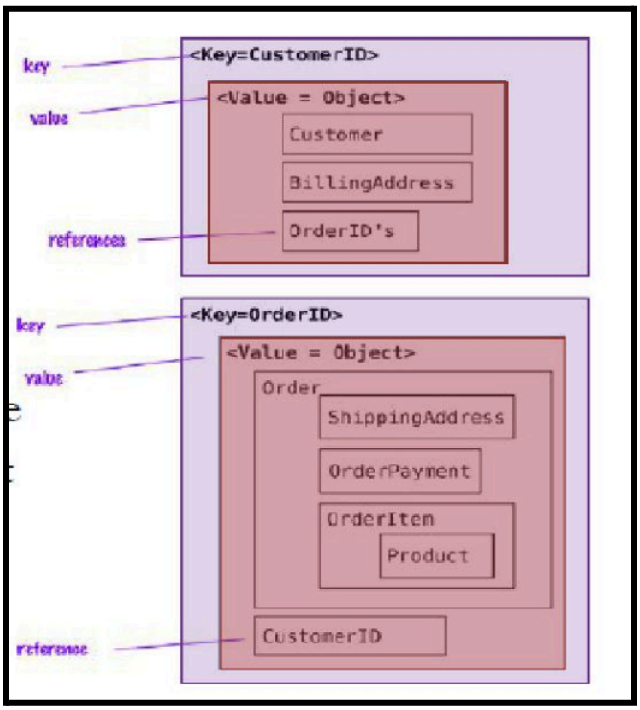
- **Can store anything** without restrictions (text, images, files, serialized objects).
- **Very fast lookups** since data is accessed directly by key.
- **Minimal database overhead**, making them highly scalable.

Disadvantages:

- No ability to **search inside** the data (only exact key lookups are possible).
- No built-in indexing for efficient queries.







Document Databases: Aggregates Are "Structured" (Readable & Searchable)

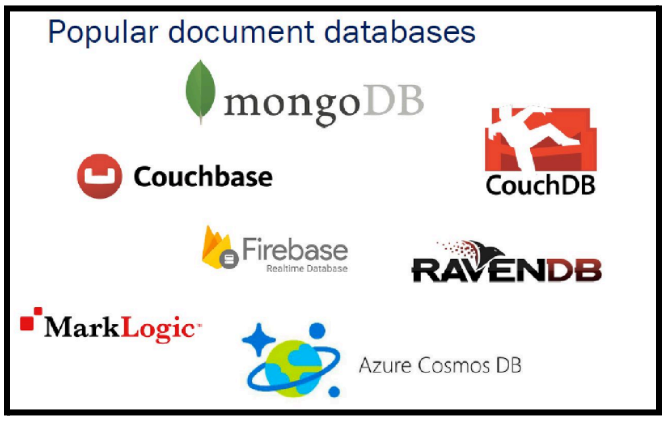
- In **document databases**, the database **understands the structure** inside an aggregate, allowing it to **search, index, and filter data**.
- **Example:**
    - Instead of a **locker system**, imagine a **library catalog** where books are indexed **by title, author, and category**.
    - You can find books **by keyword, genre, or author**—not just by ID.

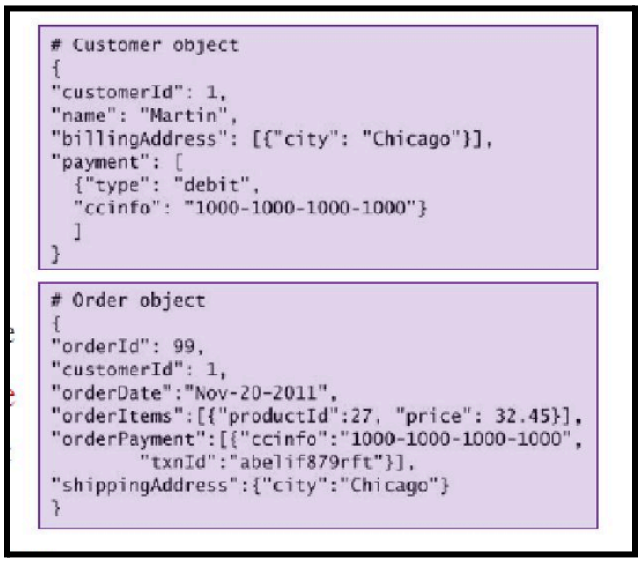
Advantages of Document Databases:

- Allows **querying based on fields inside the document** (not just by key).
- Supports **partial retrieval**, so you don't need to load the entire document.
- Can **create indexes** for efficient searching.

Disadvantages:

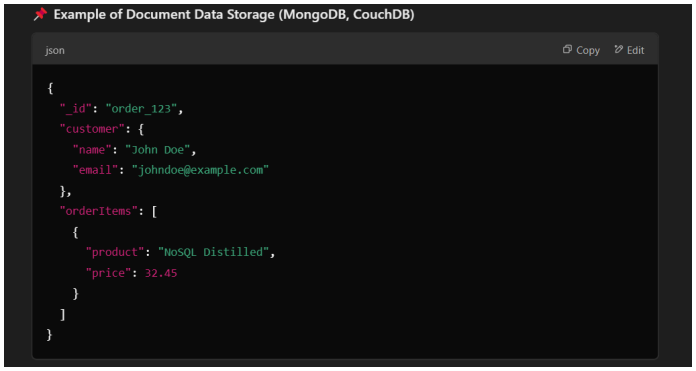
- More overhead than key-value stores (since the database must process document structure).
- Some constraints on data types and structure.
- The database **understands** the fields (`customer.name`, `orderItems.product`), allowing **queries on any field** instead of just `_id`.





```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}
```

```
# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```



Example of Document Data Storage (MongoDB, CouchDB)

json Copy Edit

```
{
  "_id": "order_123",
  "customer": {
    "name": "John Doe",
    "email": "johndoe@example.com"
  },
  "orderItems": [
    {
      "product": "NoSQL Distilled",
      "price": 32.45
    }
  ]
}
```

Key Differences Between Key-Value and Document Databases

Feature	Key-Value Databases	Document Databases
Data Structure	Unstructured ("opaque" blob)	Structured (JSON, BSON, XML)
Querying	Only by key	Query by any field inside the document
Indexing	No built-in indexing	Supports indexing for fast searches
Flexibility	Store anything without restrictions	Some constraints on data structure
Performance	Faster for simple lookups	Slightly slower but more powerful queries
Best For	Caching, session storage, simple lookups	Complex queries, hierarchical data

Blurry Line Between Key-Value and Document Databases:

- In **real-world systems, the distinction is not always clear**.
- Many **key-value databases allow some structure** in data storage, while some **document databases allow key-based lookups** like a key-value store.

Examples of Hybrid Features:

1. **Riak (Key-Value Database)**
   - Allows **metadata tagging** for indexing.
   - Supports **searching within JSON/XML aggregates** (Solr-like search).
2. **Redis (Key-Value Store)**
   - Supports **lists, sets, and sorted sets** instead of just raw blobs.
3. **MongoDB (Document Database)**
   - Can perform **exact key lookups** (similar to a key-value store).
   - Can **store large binary files** (like key-value stores).

DISTRIBUTION MODELS

Why NoSQL Uses Distribution Models?

- The **main reason** for using NoSQL databases is their ability to run on **large clusters of servers**.
- As **data volumes grow**, it becomes **expensive and difficult** to scale up by simply **buying a bigger server**.
- Instead, a better approach is to **scale out** by **distributing the database across multiple servers (cluster)**.

How does NoSQL achieve this?

- **NoSQL databases distribute data across multiple nodes** (servers) instead of storing everything in one place.
- **Aggregate-oriented NoSQL models** (like key-value and document stores) work well with distribution because **aggregates can be stored independently on different servers**.

Why Do We Need Distribution Models?

The right distribution model allows NoSQL databases to:
✅ **Handle large amounts of data efficiently.**
✅ **Support high read and write traffic.**
✅ **Improve availability**, so the system **still works even if some servers fail**.

- ❖ **But there is a tradeoff:**
  - Running databases over **multiple servers (a cluster) introduces complexity**.
  - We only **use distributed models when the benefits outweigh the added complexity**.
- ❖ **Two Main Paths to Distribute Data**
  - ➢ NoSQL databases distribute data in two primary ways:
    Replication → Copies the **same data** to multiple servers.
    Sharding → Stores **different parts of the data** on different servers.
- **Replication and sharding can be used together**.
- **Replication improves availability**, while **sharding helps distribute large data efficiently**.

Replication

- **Replication** means creating **multiple copies of the same data** across different servers.
- This ensures that **if one server fails, another can take over**.

Types of Replication:

1. **Master-Slave Replication**
2. **Peer-to-Peer Replication**

Master-Slave Replication

- ❖ **How It Works:**
  - **One server acts as the "master"**, handling all writes (inserts, updates, deletes).
  - **Multiple "slave" servers** copy data from the master and handle only **read operations**.
- ❖ **Advantages:**
  - **Good for read-heavy applications** (because reads are distributed across multiple slaves).
  - **Ensures consistency** since all writes go to the master first.
- ❖ **Disadvantages:**
  - **If the master fails, the system stops accepting writes** until a new master is assigned.
  - **Slaves may have outdated data** if there's a delay in copying updates from the master.
- ❖ **Example:**

- **Logging systems** → All logs are written to the master, but multiple read replicas handle analytics queries.

Peer-to-Peer Replication

- ❖ **How It Works:**
  - Instead of having a single master, **all nodes (servers) can accept both reads and writes**.
  - Data is **replicated between all nodes**, so updates made on one node are copied to the others.
- ❖ **Advantages:**
  - **No single point of failure** → If one node goes down, others can continue processing requests.
  - **Faster writes** since multiple nodes accept updates.
- ❖ **Disadvantages:**
  - **Conflict resolution is needed** → If two nodes update the same data at the same time, the system must decide which version to keep.
  - **More complex synchronization** between nodes.
- ❖ **Example:**
  - **Multi-region applications** → A banking system where users in different locations update their data on separate servers, which later sync.

Sharding (Distributing Different Data Across Servers)

- **Sharding** (also called **horizontal partitioning**) means **splitting data across multiple servers**.
- Each server **stores only a portion of the total dataset**.
- Unlike replication (where all servers have the same data), **each shard contains different data**.
- ❖ **How It Works:**
  - **Example: A database of users with 10 million records.**
    - Instead of keeping all 10 million users on **one server**, we **split** them into **10 shards**, each storing **1 million users**.
    - Queries are directed to the **right shard based on the user's ID**.
- ❖ **Advantages:**
  - **Massively scalable** → Can handle huge amounts of data by adding more servers.
  - **Faster queries** → Since each shard stores only part of the data, searches are quicker.
- ❖ **Disadvantages:**
  - **More complex query logic** → The system must know which shard contains the data it needs.
  - **Shards may become unbalanced** → If some shards hold more data than others, performance issues arise.

Example:

- **Large social media platforms (e.g., Facebook, Twitter)** → User data is spread across multiple shards to handle billions of profiles efficiently.

Combining Replication and Sharding

Why Use Both?

- Many NoSQL databases **use both replication and sharding together** to get **the best of both worlds**.
- **Example setup:**
    - **Sharding is used to split data** across multiple servers.
    - **Each shard is then replicated** to multiple nodes for **high availability**.

Advantages:

- **Scalable** (Sharding) → Spreads the load across many servers.
- **Fault-tolerant** (Replication) → Keeps copies of data for backup and failover.

Real-World Example:

- **Amazon DynamoDB**
    - **Shards user data** across many servers.
    - Each shard is **replicated to prevent data loss**.

Progression of Complexity in Distribution Models

Single-Server Database → All data is stored on **one server** (easy but not scalable).
Master-Slave Replication → Read performance improves, but **single point of failure** exists.
Sharding → Data is distributed, but **queries become more complex**.
Peer-to-Peer Replication → Eliminates single points of failure but **increases complexity**.

SHARDING

What is Sharding?

- **Sharding** is a technique used in databases to distribute data across multiple servers.
- Instead of **storing all data on a single server**, we **divide it into multiple parts** and store each part on a **different server**.
- This is also known as **horizontal scalability**, meaning we increase performance by **adding more servers instead of upgrading a single powerful one**.
- **Each shard (server) handles only a portion of the data**, which **reduces the load on each individual server** and improves performance.

Example:

- Suppose we have **a database with 10 million users**.

- Instead of keeping **all 10 million records on one server**, we can divide them into **10 shards**, with each storing **1 million users**.
- If a user logs in, their request goes directly to the shard that contains their data, making the query **much faster** than searching through all 10 million records.

How Sharding Works

- Each **server (shard) stores a different portion of the data**.
- Each shard handles **its own reads and writes**, so the workload is distributed.
- If the database runs on **10 servers**, then each server **only handles 10% of the traffic**, making the system **efficient and scalable**.

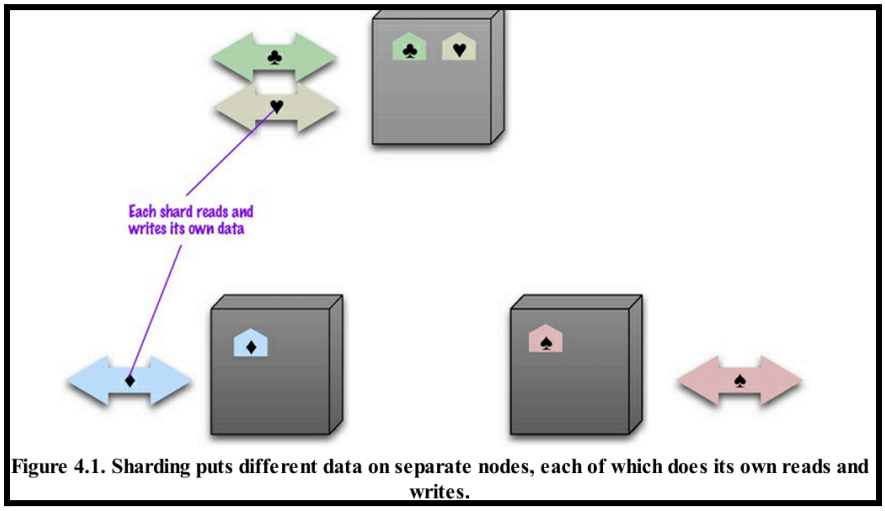


Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

The Ideal Case for Sharding

- Ideally, **each user would only need to access one shard** to get all their data.
- This would mean that **users are evenly distributed across all shards**, and no single shard is overloaded.
- In this perfect scenario:
  - Each query **goes to only one shard**, so it's very fast.
  - The workload is **balanced across all servers** (e.g., if there are 10 servers, each handles 10% of the load).

Reality Check: The Ideal Case is Rare

- In real-world scenarios, users **don't always access different parts of the dataset evenly**.
- Some shards may get **more traffic than others**, leading to **imbalanced workloads**.
- To get close to the ideal case, we need **smart strategies** for **grouping and distributing data**.

How to Organize Data for Efficient Sharding/ Improve Sharding:

To improve performance, we need to **carefully decide how to distribute data across shards**.

1. **Store Related Data Together (Clumping Data in One Shard)**
    - A **single user's data should be stored on one shard**, so that their queries do not need to access multiple servers.
    - **This is where aggregate orientation helps**:
        - Aggregates are designed to **group commonly accessed data together**, making them **perfect units for sharding**.
        - Example: **A customer's orders, payment details, and shipping information** should be stored **together on the same shard**.
2. **Use Location-Based Sharding (Geographical Data Placement)**
    - If users **from a certain location access data frequently**, it makes sense to store their data **close to where it's being accessed**.
    - **Example:**
        - Orders for **customers in New York** can be stored in a **data center on the East Coast**.
        - Orders for **customers in Los Angeles** can be stored in a **West Coast data center**.
    - **This reduces network latency** and improves **response times** for users.
3. **Keep the Load Balanced Across Shards**
    - Shards should be **evenly distributed**, meaning **each shard should handle a similar amount of requests**.
    - If some shards contain **more frequently accessed data**, they will become overloaded.
    - **Example:**
        - If a database stores **product purchases**, some products might be **much more popular than others**.
        - A shard storing **high-demand products** will get **more traffic than other shards**, causing performance issues.
    - **Solution:**
        - Dynamically **rebalance shards** by **moving some high-traffic data to less active shards**.
        - Identify patterns, like **certain data being accessed more on specific days of the week**, and adjust shard placement accordingly.
4. **Store Data That's Read in Sequence Together**
    - If **data is often accessed in a specific order**, keeping related records together can **improve query performance**.
    - **Example:** Google's Bigtable sorts web addresses by **reversed domain names** (`com.amazon`, `com.google`) so that data for **multiple related pages** can be accessed together efficiently.

How Sharding Has Been Done Historically:

- In the past, **sharding was handled manually** in the **application code**.
- Example:
    - Customers **with last names A-D** were stored in **Shard 1**.
    - Customers **E-G** were stored in **Shard 2**, and so on.

Problems with Manual Sharding:

- **Developers had to write code** to ensure that **queries were sent to the right shard**.
- **Rebalancing shards** (moving data to new shards) was difficult and required **changing the application code**.

Modern NoSQL Databases Offer Auto-Sharding

- Many NoSQL databases now **automatically manage sharding**.
- **The database itself decides** where to store new data and ensures that queries go to the correct shard.
- **No need for manual sharding logic in application code**.

Examples of Auto-Sharding in NoSQL Databases:

- **MongoDB** → Automatically distributes documents across shards.
- **Cassandra** → Uses **consistent hashing** to distribute data dynamically.

Why is Sharding Essential for Write Performance?

- **Replication improves read performance** by **copying data to multiple servers**, but it **does not help with write-heavy applications**.
- **Sharding improves both read AND write performance** because:
    - Each shard **handles only a fraction of the writes**.
    - Instead of all updates going to **one massive database**, they are **spread across multiple servers**.

Example:

- A **social media platform** where millions of users post updates.
- If **all writes go to a single server**, it will quickly **become overloaded**.
- **Sharding distributes the writes** across multiple servers, preventing bottlenecks.

Does Sharding Improve System Resilience?

No, Not By Itself

- **Sharding alone does NOT improve fault tolerance**.
- If **one shard fails, all data on that shard becomes unavailable**.
- Unlike replication (where multiple copies exist), **sharding spreads unique data across different nodes**.

How Can We Make Sharding More Resilient?

- ✅ **Combine Sharding with Replication**
    - **Each shard is replicated** across multiple nodes.

    - If one shard fails, the **replica can take over**, preventing data loss.

Challenges of Moving from a Single Server to Sharded Architecture

- Some databases are **built for sharding from the start**, but others start as **single-server solutions** and later introduce sharding.
- **Problem:**
    - If sharding is added **too late**, **migrating data to new shards** can **overload the database**.
    - If not planned properly, **turning on sharding in production can cause major downtime**.

Best Practice:

- **Plan for sharding early, even if you don't need it yet.**
- Start with a **single-server setup**, but be ready to **enable sharding before the system reaches its limits**.

Example of Sharding Failure:

- A team **waited too long to enable sharding**, and when they finally turned it on, the database **became unavailable** because it was too busy moving data between shards.

MASTER-SLAVE REPLICATION

What is Master-Slave Replication?

- **Master-Slave Replication** is a technique used in databases to **copy data across multiple servers**.
- **One server (the master) is responsible for handling all updates (writes, deletes, and modifications)**, while **other servers (slaves) only copy and store data from the master**.
- The **slaves continuously synchronize with the master**, meaning they receive updates to **ensure they always have the latest version of the data**.
- The **main purpose of this setup is to scale read operations** since slaves can handle read queries, reducing the load on the master.

How Master-Slave Replication Works

- The **master node** processes **all write operations**.
- The **slave nodes** continuously receive copies of data from the master.
- **Read queries are directed to the slaves**, while the master handles only **write operations**.

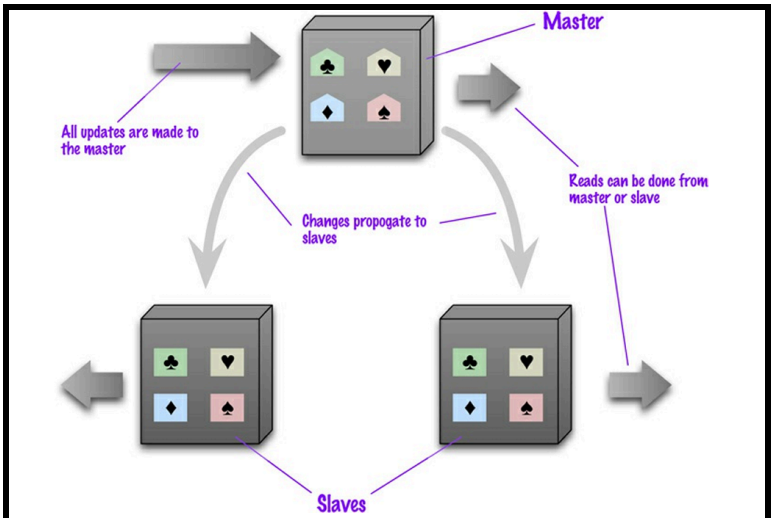


Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

When is Master-Slave Replication Useful?

Master-slave replication is **most useful in read-heavy applications**, where **most database operations involve retrieving data (reads) rather than updating it (writes).**

Scaling Read Requests

- If an application has **millions of users reading data simultaneously**, a **single database server will struggle to handle all those read queries**.
- By **adding multiple slave nodes**, we can distribute read requests across many servers, reducing the load on the master.
- **Example:**
    - **A news website** where thousands of users read articles simultaneously.
    - Instead of overloading one server, **the articles are copied to multiple slaves**, allowing more users to read them **without slowing down the system**.

Not Ideal for Write-Heavy Applications

- **Problem:** The **master node** still has to handle **all write requests**.
- If an application has **millions of users constantly writing (updating, inserting, or deleting data),** the master can become **overloaded and slow**.
- **Example:**
    - A **social media platform** where users **constantly post updates and comments**.
    - Since the **master handles all writes**, it might **struggle to process updates quickly**.

Helps with Read Resilience

- If the **master node fails**, the **slaves can still handle read operations**.
- This is **useful for applications where reading is more important than writing**.

- **Example:**
    - A **product catalog for an e-commerce website**.
    - Even if the master goes down, users can **still browse products and view prices** because **read queries are handled by the slaves**.

What Happens If the Master Fails?

Temporary Problem:

- If the master **goes offline**, **no new updates (writes) can happen**.
- However, the **slaves can still handle read requests**, meaning users **can still access data**, just not update it.

Solution: Appointing a New Master

- A **slave can be promoted to become the new master**, allowing writes to continue.
- This process is known as **failover** and can be **done manually or automatically**.

Manual vs. Automatic Master Selection

Manual Master Selection

- When configuring a database cluster, **one node is assigned as the master manually**.
- If the master fails, **a database administrator (DBA) must manually assign a new master**.

✅ **Simple and predictable**
❌ **Causes downtime** while waiting for a human to assign a new master

Example:

- A **banking system** where strict control is needed before changing the master node.

Automatic Master Election

- In this approach, **the database cluster automatically elects a new master** when the current master fails.
- Nodes **communicate with each other** to determine which one should take over.

✅ **Minimizes downtime** because the transition happens instantly
❌ **More complex to configure**

Example:

- A **real-time chat application** where downtime must be avoided at all costs.

Ensuring Read Resilience

To ensure **that the system can still function even if the master fails**, applications need to:
1. **Separate Read and Write Paths**

- Applications should use **different connections for reads and writes**.
- This ensures that if **writes fail due to a master failure, reads can still continue on the slaves**.

2. **Use Database Connection Pools that Support Failover**

- Some database connection libraries **do not support separate read and write connections**, which can lead to unexpected failures.
- **Solution:** Choose database drivers that allow **failover mechanisms**.

3. **Test Read Resilience**

- Developers should **disable the master node** in a test environment and check if reads still work correctly.
- This ensures that the **application doesn't crash when the master goes down**.

Example of Read Resilience in Action:

- **A stock market tracking website**
    - Users continuously read **stock prices** from the database.
    - If the master fails, the **slaves continue providing stock data**, ensuring **zero downtime** for users.

The Dark Side of Replication: Data Inconsistency

Replication has **many benefits**, but it also introduces **the risk of inconsistency**.

What is Data Inconsistency?

- Since **slaves receive updates from the master asynchronously**, they may **not always have the latest data**.
- If a user writes data to the master, **it might take some time** for that update to be copied to the slaves.
- During this delay, **different users querying different slaves may see different versions of the data**.

Worst-Case Scenario:

- A user **writes data to the master**, then **immediately reads from a slave**—but the update hasn't reached the slave yet.
- This is known as **"Read-After-Write Inconsistency"**, where a user **cannot read their own recent update**.

How to Reduce Inconsistency?

1. **Read from the Master for Critical Data**

- If a query **must always return the latest data**, direct it to the master instead of a slave.

2. **Use Read-Your-Own-Write Techniques**

- Some databases allow a user's reads **to be directed to the master for a short period after they perform a write**.

3. **Tunable Consistency in NoSQL Databases**

- Some NoSQL databases **allow developers to configure how strict or relaxed consistency should be**.
- Example: **Cassandra's "QUORUM" consistency level** ensures that **a write is acknowledged only when a majority of nodes have received it**.

Example of Data Inconsistency Issue:

- **A user sends a message on a chat app**.
- They immediately check the chat history, but their message **doesn't appear because they are reading from a slave** that hasn't been updated yet.
- After a few seconds, the message finally appears.

PEER-TO-PEER REPLICATION

What is Peer-to-Peer Replication?

- **Peer-to-Peer (P2P) Replication** is a method of **replicating data across multiple servers** where **all nodes are equal**—there is **no master** server.
- This means **any node can accept read and write operations**, and losing a node **does not stop the system from functioning**.

How is This Different from Master-Slave Replication?

- In **Master-Slave Replication**, **only the master can process writes**.
    - **Problem:** The **master becomes a bottleneck**.
    - If the master **fails, writes cannot happen** until a new master is assigned.
- In **Peer-to-Peer Replication**, **all nodes can accept writes**.
    - **Advantage:** The system is **more scalable and resilient** since **any node can handle read and write operations**.
- Instead of **one master**, **all nodes are treated equally**.
- Each node **synchronizes with other nodes**, ensuring data is **replicated across the system**.

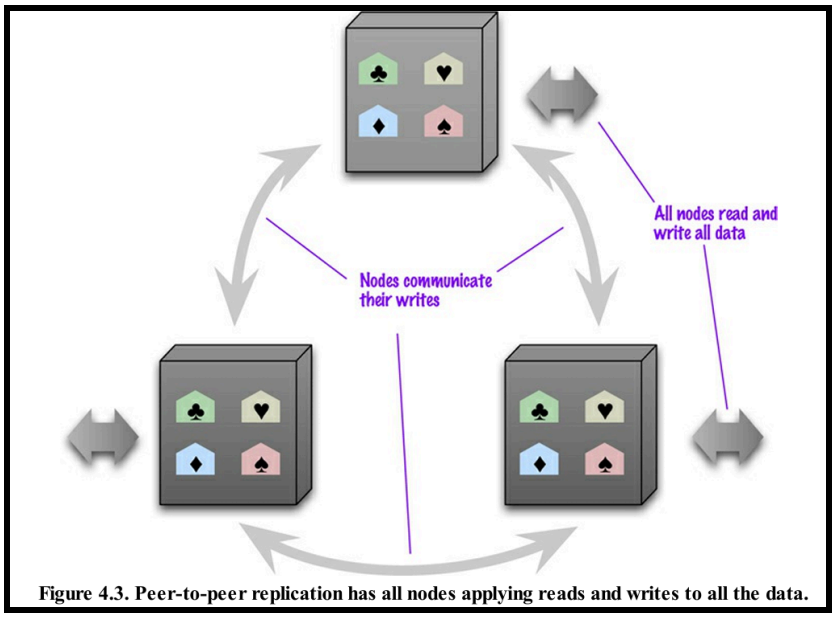


Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

Advantages of Peer-to-Peer Replication

1. **No Single Point of Failure**

- Since **all nodes are equal**, the failure of **one node does not affect the system**.
- Even if a server goes offline, **other nodes can still accept read and write requests**.

Example:

- A **banking system** where users deposit money into their accounts.
- If a **server handling transactions fails**, other servers **continue processing deposits without interruption**.

2. **Improved Write Scalability**

- Since **any node can accept writes**, the system can **handle more write operations** by adding more nodes.
- **This solves the problem in Master-Slave Replication**, where the **master becomes a bottleneck for writes**.

Example:

- A **social media platform** where millions of users post updates every second.
- Instead of **one master handling all posts**, multiple servers **share the write load**, making the system faster.

3. **Easier to Scale**

- If the application **needs more capacity**, new nodes can be **added easily**.

- More servers mean **more power to handle both reads and writes**.

Example:

- A **video streaming service** that grows in popularity.
- The company can **add more servers to handle more users** without restructuring the database.

Challenges of Peer-to-Peer Replication

Data Consistency Issues (Write Conflicts)

- **The biggest problem** with Peer-to-Peer Replication is **data inconsistency**.
- Since **multiple nodes accept writes**, there is a risk that **two people update the same record at the same time on different nodes**.
- **This leads to conflicting data, known as a write-write conflict**.

Example:

- Imagine **two users update their profile picture at the same time**.
    - **User 1 updates it on Server A.**
    - **User 2 updates it on Server B.**
- Now, both **servers have different profile pictures for the same user—which one is correct?**

Why is this different from Master-Slave Replication?

- In Master-Slave Replication, **all writes go to the master**, so there is **no chance of conflicting updates**.
- In Peer-to-Peer, since **any node can accept writes, conflicts can happen**.

How to Handle Write Conflicts?

There are two broad ways to handle **write conflicts**:

Option 1: Synchronous Coordination (Strong Consistency)

- When a node **receives a write**, it first **synchronizes with other nodes** to **ensure there are no conflicts** before committing the update.
- This provides **the same level of consistency as a master**, but **slows down the system** due to **network communication** between nodes.

✅ **Advantage:** Ensures **data consistency** (no conflicting writes).
❌ **Disadvantage:** Slower writes because nodes **must communicate before committing an update**.

Example:

- A **financial system** where transactions **must always be accurate**.

- Before confirming a bank transfer, the system **ensures all nodes agree on the transaction details**.

Option 2: Allow Inconsistent Writes and Merge Later (Eventual Consistency)

- Instead of **blocking writes until all nodes agree**, we **allow writes to happen freely**, even if conflicts occur.
- Later, we use **policies to merge conflicting writes**.

✅ **Advantage:** Much **faster performance** because nodes **do not need to wait for confirmation**.
❌ **Disadvantage:** Requires **custom logic** to merge conflicts, and **data might be temporarily inconsistent**.

Example:

- A **collaborative document editing tool** like Google Docs.
- If two people **edit the same sentence on different nodes**, the system **automatically merges their changes** or shows **both versions with a conflict marker**.

Consistency vs. Availability Tradeoff

Peer-to-Peer Replication forces a tradeoff between consistency and availability.

Approach	Consistency Level	Availability	Performance
Synchronous Coordination (Strong Consistency)	High (No conflicts)	Lower (Slower writes)	Slower
Allow Inconsistent Writes (Eventual Consistency)	Lower (Conflicts can occur)	High (Always available)	Faster

📌 Which One to Choose?

- If data accuracy is critical (e.g., financial transactions), use strong consistency.
- If high availability is more important (e.g., social media posts, messages), allow eventual consistency.

COMBINING SHARDING AND REPLICATION

Why Combine Sharding and Replication?

- **Sharding and replication are two powerful techniques**, but they **solve different problems**.
- **Sharding** helps with **scalability** by **splitting data across multiple servers**.
- **Replication** improves **availability and fault tolerance** by **copying data across multiple servers**.

- **When used together**, they provide **both scalability and reliability**, making the database **highly available and fast**.

How Does This Work?

- Each **piece of data belongs to only one master** (sharded across different nodes).
- Each **shard is then replicated** so that multiple copies exist.
- **If a node fails**, the replica takes over, preventing data loss.
- **Data is first divided into shards** (different subsets stored on different nodes).
- Each **shard is then replicated** so that multiple copies exist.
- This means that **even if one node fails, another copy of the data is still available**.

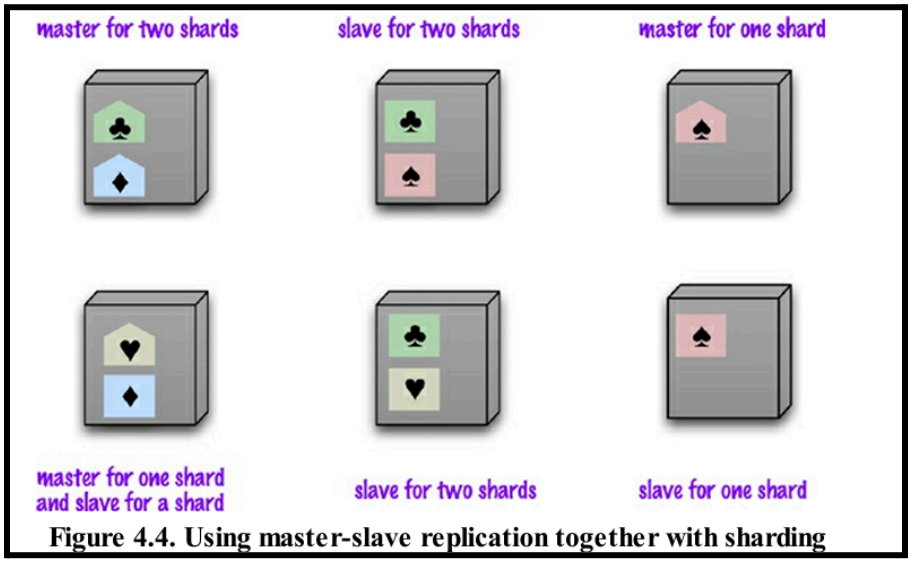


Figure 4.4. Using master-slave replication together with sharding

Using Master-Slave Replication with Sharding

- **Master-Slave Replication** can be **combined with sharding** so that:
  - **Each shard has its own master** that handles all **writes** for that shard.
  - **Slave nodes** replicate the shard and handle **reads**.
- This allows for **fast read performance**, while **writes are still handled by one master per shard**.

Example:

- Suppose we have a **database with 1 billion customer orders**.
- We **shard the database** into **10 shards**, each containing **100 million orders**.
- Each shard has:
  - **1 master (handles writes)**
  - **Multiple slave replicas (handle reads)**
- **If one master fails**, a **replica can be promoted to master**, ensuring **high availability**.

✅ **Advantages:**

- **Read scalability** → Reads are distributed across multiple slave nodes.
- **Data availability** → If a master fails, a slave can take over.
- **Efficient writes** → Each shard's master handles only **a portion** of the write load.

❌ **Disadvantages:**

- **The master of each shard is still a bottleneck** for writes.
- **Shards must be carefully balanced** to avoid some becoming overloaded.

Using Peer-to-Peer Replication with Sharding

- **Peer-to-Peer Replication** can also be combined with **sharding**, and this approach is **common in column-family databases** like **Cassandra**.
- **Each shard is distributed across multiple nodes**, and **all nodes can accept writes** (no master).
- A common strategy is to have **a replication factor of 3**, meaning each shard is copied to **3 different nodes**.

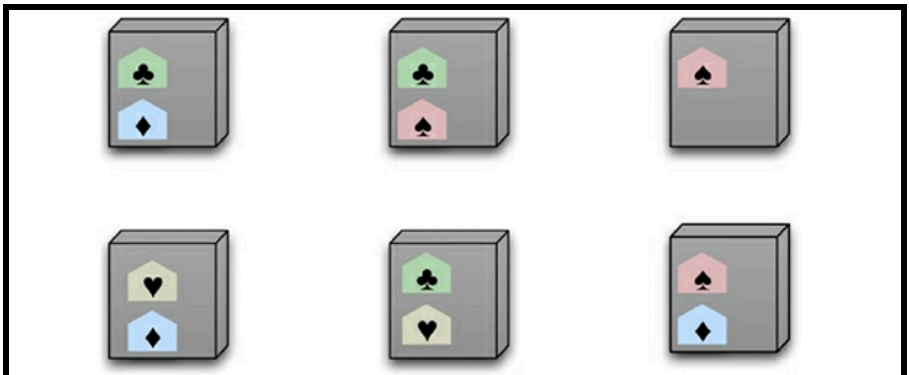


Figure 4.5. Using peer-to-peer replication together with sharding

Example:

- Suppose we have a **database with 1 billion user profiles**.
- We **shard the database** into **10 shards**, each containing **100 million users**.
- Each **shard is then replicated** across **3 different nodes**.
- If **one node fails**, the database still has **two copies of that shard**, ensuring **high availability**.

✅ **Advantages:**

- **No master bottleneck** → Any node can accept writes.
- **High availability** → Data is copied across multiple nodes, reducing the impact of node failures.
- **Easier scaling** → More nodes can be added anytime.

❌ **Disadvantages:**

- **More complex conflict resolution** since multiple nodes can write at the same time.
- **Increased network traffic** to keep replicas synchronized.

Comparison of Sharding + Master-Slave vs. Sharding + Peer-to-Peer

Feature	Sharding + Master-Slave Replication	Sharding + Peer-to-Peer Replication
Write Handling	Writes go to **one master per shard**	Writes can go to **any node**
Read Handling	Reads go to **slave nodes**	Reads can go to **any node**
Fault Tolerance	If master fails, **a slave takes over**	If one node fails, **other nodes handle requests**
Scalability	**Easier to implement** but **limited by master bottleneck**	**More scalable** since any node can write
Consistency	**Stronger consistency** (writes go through one master)	**Eventual consistency** (conflicts can occur)
Best For	Read-heavy applications **with controlled writes**	Large-scale distributed applications **with high availability needs**

When to Use Which Approach?

Use Sharding + Master-Slave Replication When:
✅ The application is **read-heavy**, meaning **most queries are for retrieving data** rather than updating it.
✅ You need **strong consistency** and cannot afford conflicting writes.
✅ You want a **simpler design** with **a single master per shard handling writes**.

Example:

- **E-commerce websites** where users browse products **(read-heavy)**, but orders must be **accurate**.

Use Sharding + Peer-to-Peer Replication When:
✅ The application needs **high write scalability**—many users write data at the same time.
✅ High **fault tolerance** is required—data should still be available even if multiple nodes fail.
✅ The application can **tolerate eventual consistency** (some temporary data conflicts).

Example:

- **Social media platforms** where millions of users post and update data simultaneously.

Handling Node Failures in Sharding + Replication

Since **sharding distributes data**, the failure of a node **only affects the data stored on that node**.

Solution:

- **Replication ensures that each shard is copied to multiple nodes.**
- If a node **fails**, the database **rebuilds the lost shard from its replicas**.

✅ **Best Practice:** Start with **a replication factor of 3** (each shard exists on 3 different nodes).
