Apache Cassandra

Topics

- Introduction
- Data Modeling
- Querying
- Administration
- DevOps

Introduction

- Overview
- Architecture
- Compare with Other Database

Apache Cassandra

- Distributed DBMS
  - Open-Source
  - Distributed
  - Decentralized
- Developed at Facebook
- Power the Facebook inbox search feature

Applications of Cassandra

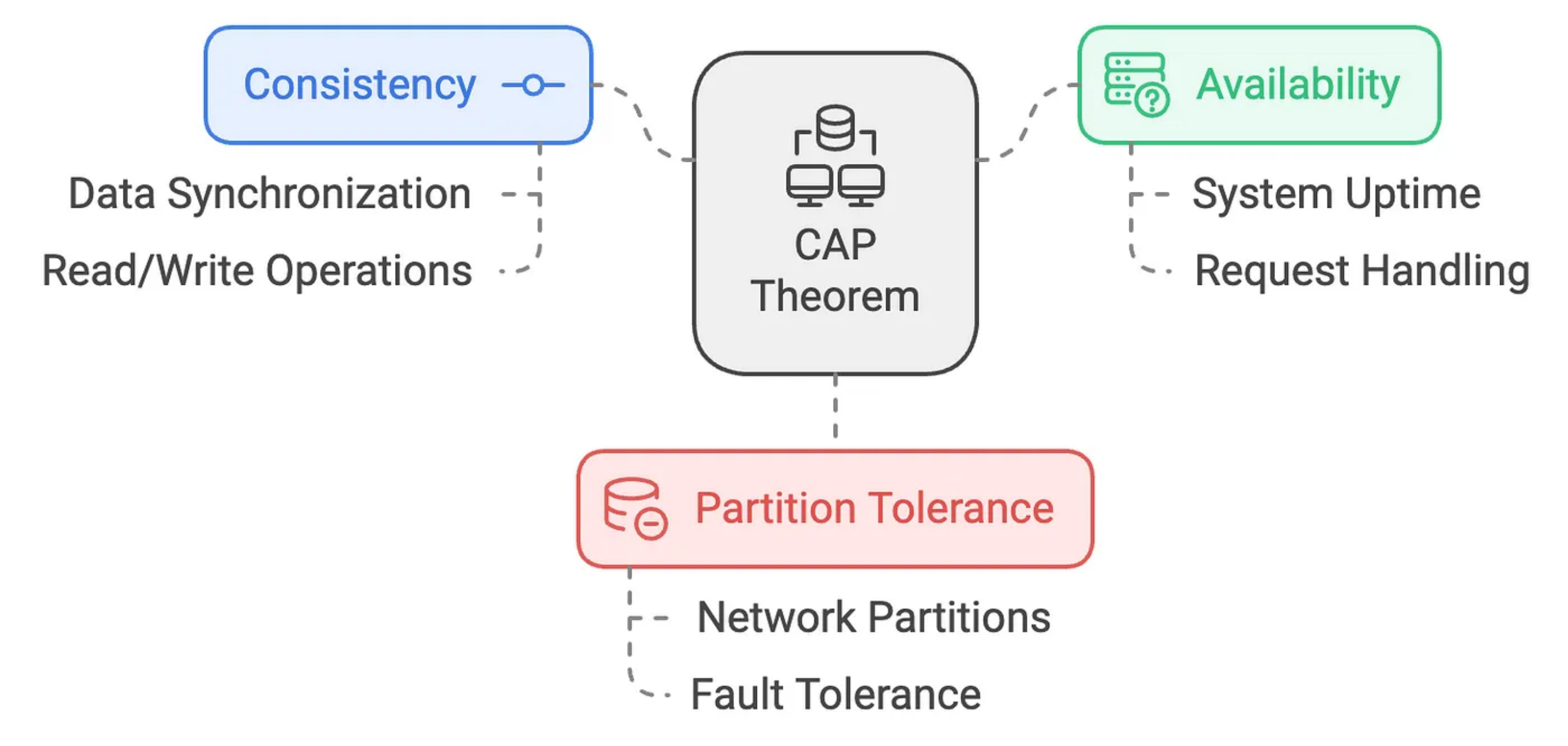
- Great Application where **Data** is collected at **High Speed** from **Different** kinds of **sources**
- Internet of Things
- Product & Retail apps
- Messaging
- Social Media Analytics
- Recommendation Engine

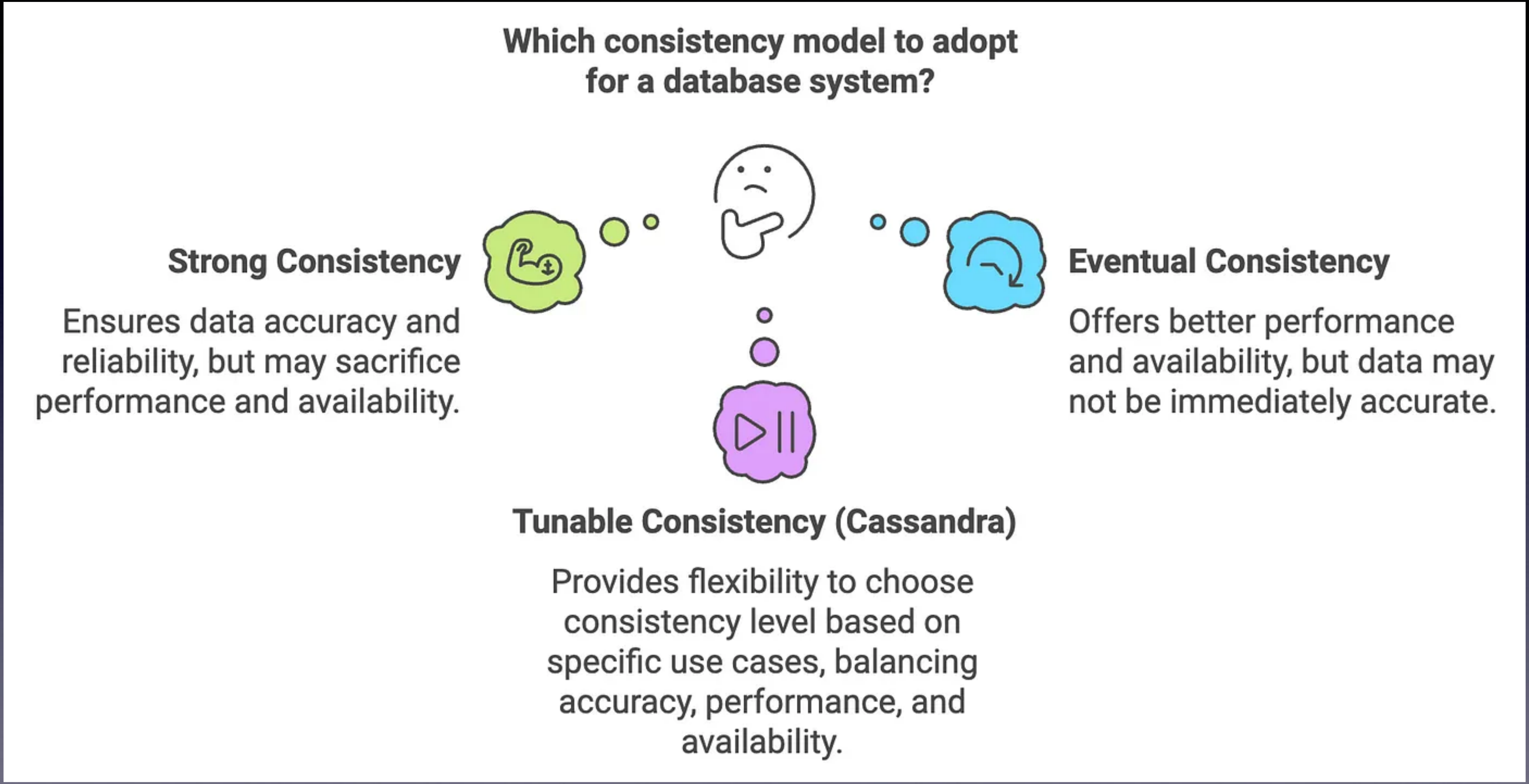
Traditional RDBMS vs Cassandra

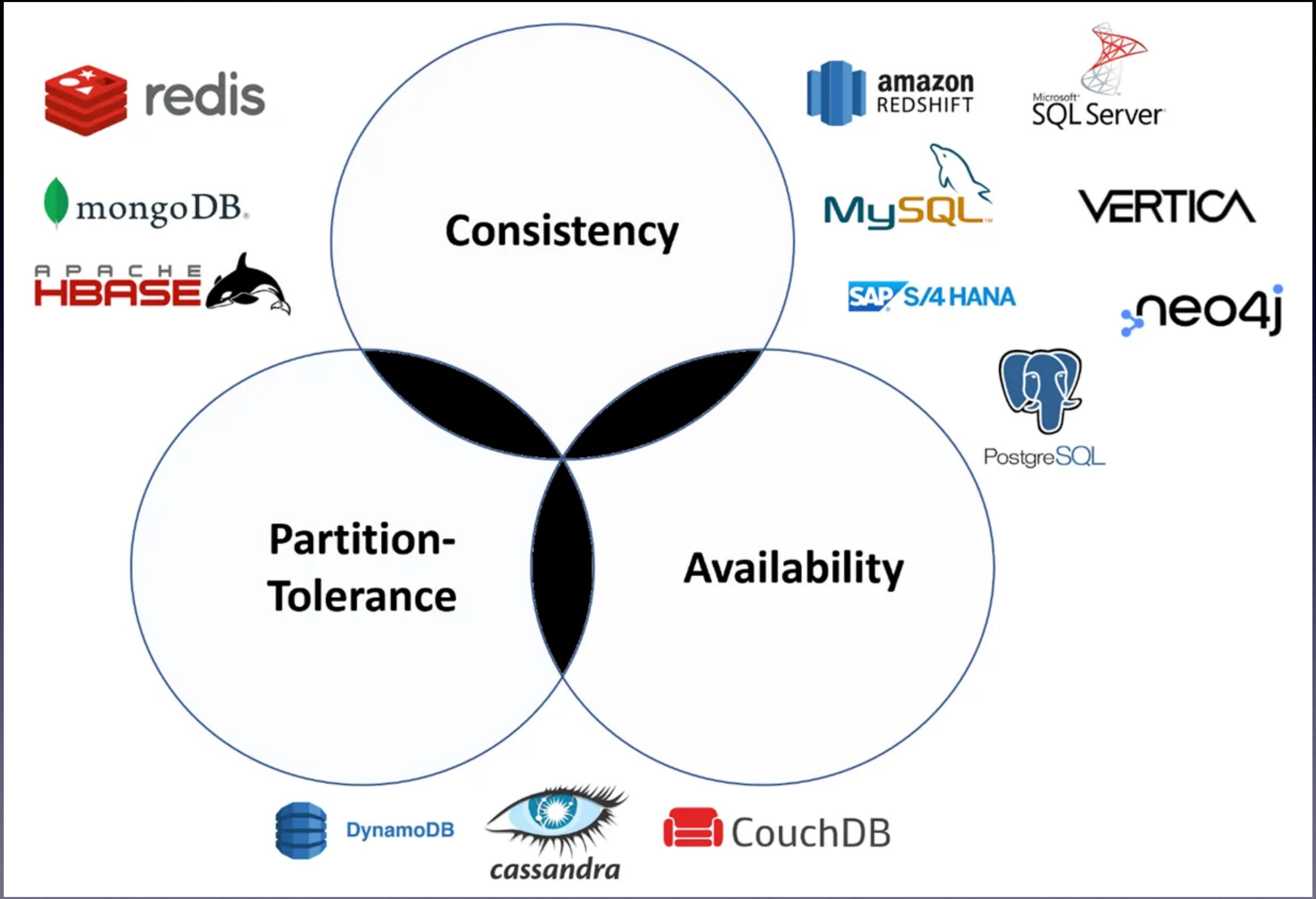
Feature	Traditional RDBMS	Cassandra
Data Model	Relational	NoSQL, flexible column families
Architecture	Centralized/Master-Slave	Distributed, Peer-to-peer
Scalability	Vertical, complex horizontal	Horizontal, easy to scale out
Consistency	Strong (ACID)	Tunable, often eventually consistent
Transactions	Complex, nested	Simple
Use Cases	Structured data, strong consistency	High availability, large datasets

The CAP Theorem

- **Consistency**
  - all nodes see the same data at the same time
- **Availability**
  - every request receives a response, even if some nodes are down
- **Partition Tolerance**
  - the system continues to function despite communication breaks between nodes


Consistency
Data Synchronization
Read/Write Operations
CAP Theorem
Availability
System Uptime
Request Handling
Partition Tolerance
Network Partitions
Fault Tolerance


Which consistency model to adopt for a database system?
Strong Consistency
Ensures data accuracy and reliability, but may sacrifice performance and availability.
Eventual Consistency
Offers better performance and availability, but data may not be immediately accurate.
Tunable Consistency (Cassandra)
Provides flexibility to choose consistency level based on specific use cases, balancing accuracy, performance, and availability.


redis
mongoDB
APACHE HBASE
Consistency
amazon REDSHIFT
Microsoft SQL Server
MySQL
VERTICA
SAP S/4 HANA
neo4j
PostgreSQL
Partition-Tolerance
Availability
DynamoDB
cassandra
CouchDB

CAP

- CA: The system prioritizes **consistency** and **availability** but doesn't handle network partitions well. In the case of a network partition, it might choose to become unavailable to maintain consistency.

- CP: The system prioritizes **consistency** and **partition tolerance** but sacrifices availability. In the face of a network partition, it will maintain consistency by refusing some requests.

- AP: The system prioritizes **availability** and **partition tolerance** but sacrifices strong consistency. It continues to operate and respond to requests even if it means returning stale data or data that is not consistent across all nodes.

BASE (NoSQL Philosophy)

- [BASICALLY AVAILABLE]
- System mostly works
- [SOFT STATE]
- May change over time
- [EVENTUAL CONSISTENCY]
- Gets fixed later

Key Terms in Cassandra

- Nodes
- Data Center / Cluster
- Commit Log
- SSTable
- MemTable
- Replication

Cassandra Node

- A node is a basic unit of Cassandra,
- It is a system that is part of a cluster.
- Node is the main area where the data is stored.
- The units of a node is represented as computer/server

Cassandra Data Center / Cluster

- A data center is a collection of Cassandra nodes.
- The data in a data center is stored in the form of a cluster
- The cluster is also referred to as a collection of nodes.

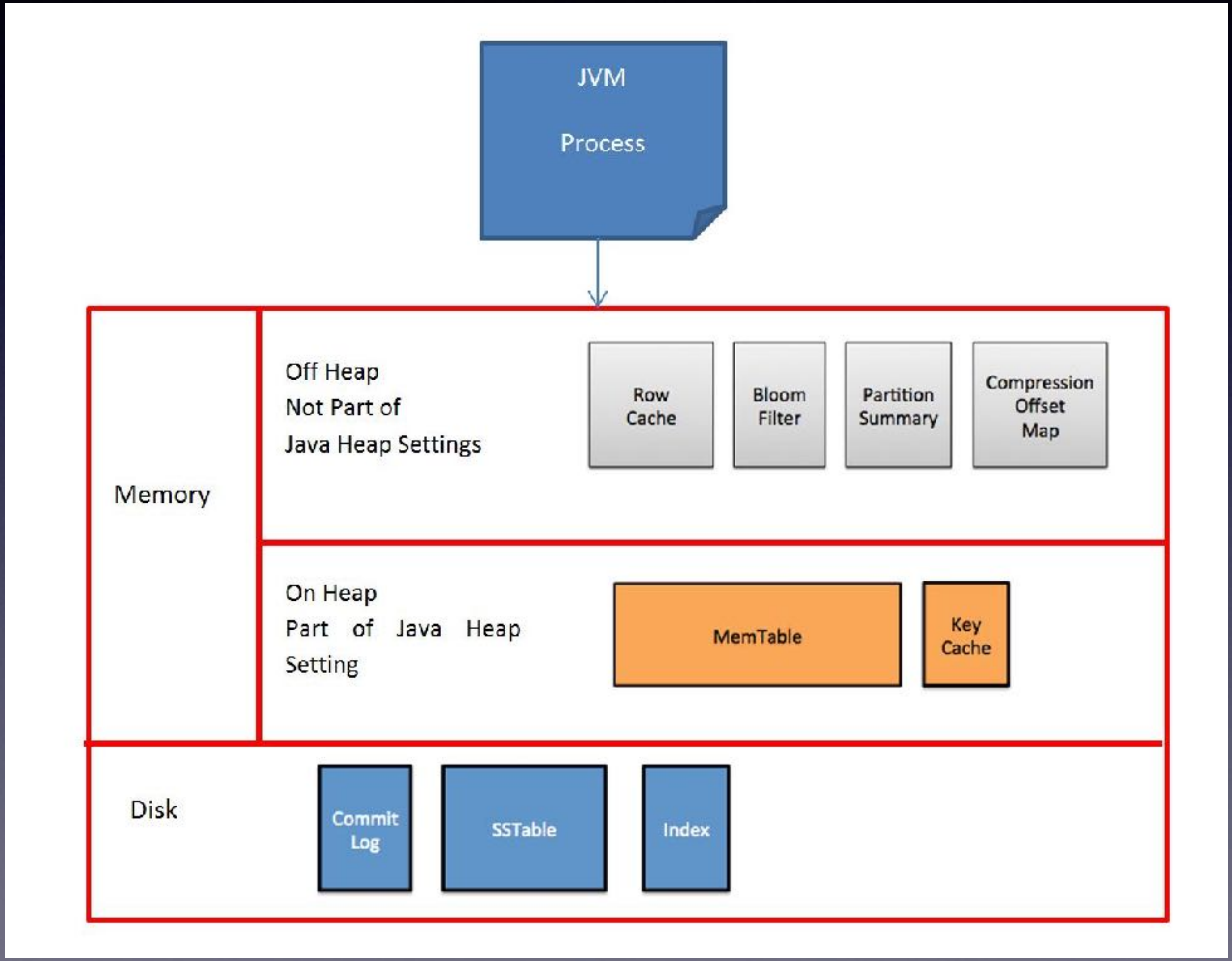
Cassandra MemTable

- MemTable is a location where data is written and stored temporarily.
- Data is written in memtable after the data is completed in the commit log.
- Memtable is a storage engine in Cassandra.
- Data in MemTable is classified into a key, and where the data is retrieved using the key as each column category has its own MemTable.
- When the write memory is full, it deletes the messages automatically.

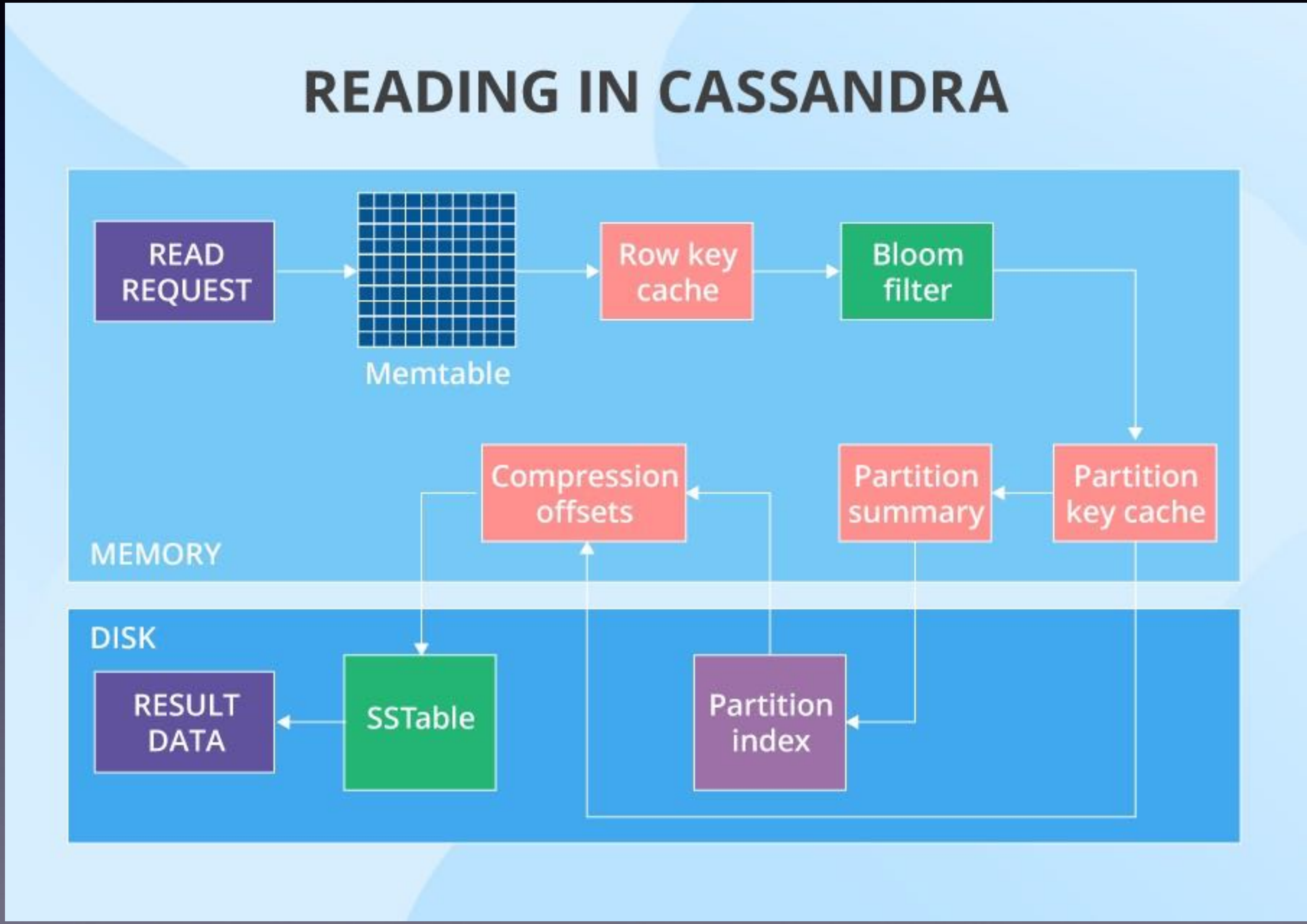
Cassandra SSTable

- SSTable also means 'Sorted String Table'.
- SSTable is a data file in Cassandra
- Its main function is to save data that is flushed from memtable.
- Unlike MemTable, SSTbale doesn't delete any data or lets any further addition once data is written.

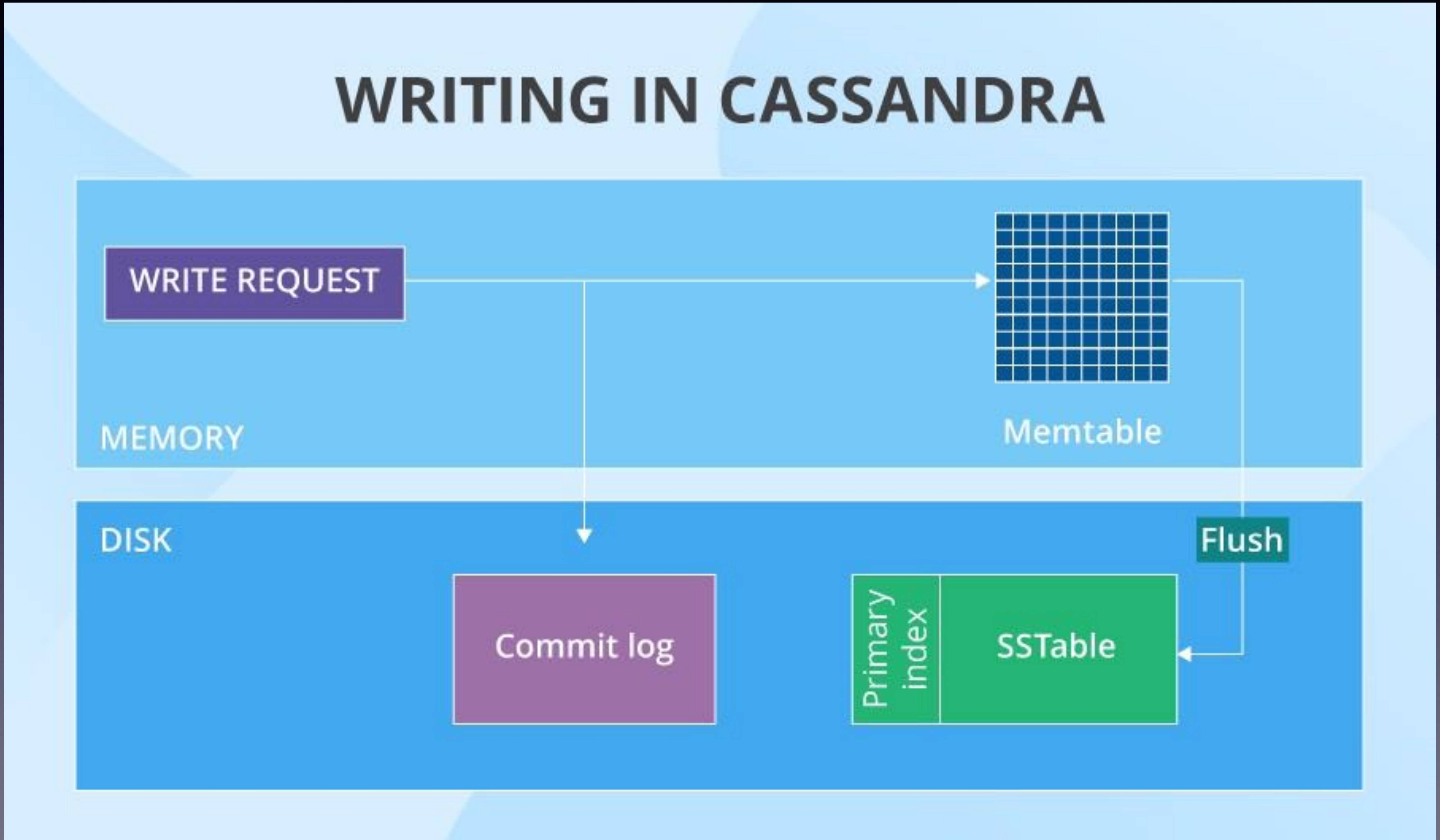
Architecture



Read Operation



Write Operation



Cassandra 5.0 Features

- Storage Attached Indexes (SAI): More flexible secondary indexing with better performance
- Vector search: Native support for AI/ML workloads with vector data type
- Unified Compaction Strategy (UCS): Adaptive compaction that optimizes automatically
- JDK 17 support: Up to 20% performance improvement from better memory management
- Trie-based storage: New memtable and SSTable formats for improved efficiency
- ACID transactions: Limited support for multi-partition transactions

Data Modeling

- Data Model
- Denormalization Strategies

What is Data Modeling

- Data modeling is the process of identifying entities and their relationships.
- In relational databases, data is placed in normalized tables with foreign keys used to reference related data in other tables.
- Queries that the application will make are driven by the structure of the tables and related data are queried as table joins.

What is Data Modeling

- In Cassandra, data modeling is Query-Driven.
- Queries are best designed to access a single table, which implies that all entities involved in a query must be in the same table to make data access (reads) very fast.
- Data is modeled to best suit a query or a set of queries. A table could have one or more entities as best suits a query. As entities do typically have relationships among them and queries could involve entities with relationships among them, a single entity may be included in multiple tables.

Query-Driven Modeling

- Unlike a relational database model in which queries make use of table joins to get data from multiple tables, joins are not supported in Cassandra so all required fields (columns) must be grouped together in a single table.

- Since each query is backed by a table, data is duplicated across multiple tables in a process known as denormalization.

- Data duplication and a high write throughput are used to achieve a high read performance.

Goals

- The choice of the **primary** key and **partition** key is important to distribute data evenly across the **cluster**.
- Keeping the number of partitions read for a query to a **minimum** is also important because different partitions could be located on different nodes and the coordinator would need to send a request to each node adding to the request overhead and latency.
- Even if the different partitions involved in a query are on the same node, fewer partitions make for a more efficient query.

Partitions

- Apache Cassandra is a distributed database that stores data across a cluster of nodes.
- A partition key is used to partition data among the nodes.
- Cassandra partitions data over the storage nodes using a variant of consistent hashing for data distribution.
- A partition key is generated from the first field (or group of fields) of a primary key.
- Data partitioned into hash tables using partition keys provides for rapid lookup.
- Fewer the partitions used for a query faster is the response time for the query.

Primary Key Components

- The primary key in Cassandra is always composed of:1.
  - **Partition** Key (mandatory)
    - Determines data distribution across nodes
  - **Clustering** Columns (optional)
    - Determines sorting within a partition

Partition Key

- The **first component** of the **primary** key
- Controls which node **stores** the data (via consistent hashing)
- All rows with the same partition key are stored together on the same node
- Data is spread across the cluster based on the partition key's hash value
- You must include the partition key in queries
- High cardinality partition keys distribute data more evenly

Partition Key

```sql
CREATE TABLE users (
    user_id UUID,
    name TEXT,
   email TEXT,
    PRIMARY KEY (user_id)  -- user_id is the partition key
);
```

Clustering Columns

- The **second** and **subsequent** components of the primary key
- Control the **sort order** of rows within a partition
- Determines the on-disk sort order of data within a partition
- Enable efficient **range queries** within a partition
- Contribute to row uniqueness within a partition

Clustering Columns

```
CREATE TABLE user_orders (
    user_id UUID,           -- Partition key
    order_date TIMESTAMP,   -- First clustering column
    order_id UUID,          -- Second clustering column
    amount DECIMAL,
    PRIMARY KEY (user_id, order_date, order_id)
) WITH CLUSTERING ORDER BY (order_date DESC);
```

Composite Partition Key

```
CREATE TABLE sensor_readings (
    sensor_type TEXT,
    sensor_id UUID,
    reading_time TIMESTAMP,
    value FLOAT,
    PRIMARY KEY ((sensor_type, sensor_id), reading_time)
);
```

Denormalizing a Relational Model

```sql
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    name VARCHAR,
    email VARCHAR,
    registration_date DATE
);
```

```sql
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR,
    price DECIMAL,
    category VARCHAR
);
```

```sql
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    user_id INT REFERENCES users(user_id),
    order_date TIMESTAMP,
    total DECIMAL
);
```

```sql
CREATE TABLE order_items (
    order_id INT REFERENCES orders(order_id),
    product_id INT REFERENCES products(product_id),
    quantity INT,
    item_price DECIMAL,
    PRIMARY KEY (order_id, product_id)
);
```

Orders by User (Query: "Get all orders for a user")

```sql
CREATE TABLE orders_by_user (
    user_id UUID,
    order_id UUID,
    order_date TIMESTAMP,
    total DECIMAL,
    -- Denormalized user data
    user_name TEXT,
    user_email TEXT,
    -- Order items as a collection
    items MAP<UUID, TEXT>, -- product_id -> "product_name:quantity:price"
    PRIMARY KEY (user_id, order_date, order_id)
) WITH CLUSTERING ORDER BY (order_date DESC);
```

Orders by Product (Query: "Get all orders containing a product")

```
CREATE TABLE orders_by_product (
    product_id UUID,
    order_date TIMESTAMP,
    order_id UUID,
    -- Denormalized product data
    product_name TEXT,
    product_price DECIMAL,
    -- Denormalized order/user data
    quantity INT,
    user_id UUID,
    user_name TEXT,
    PRIMARY KEY (product_id, order_date, order_id)
) WITH CLUSTERING ORDER BY (order_date DESC);
```

User Profiles (Single document-style record)

```
CREATE TABLE user_profiles (
    user_id UUID PRIMARY KEY,
    name TEXT,
    email TEXT,
    registration_date TIMESTAMP,
    -- Denormalized recent orders
    recent_orders LIST<TEXT> -- ["order_id:date:total", ...]
);
```

Product Catalog with Popularity Data

```sql
-- Table for product details (no counters allowed here)
CREATE TABLE product_catalog (
    product_id UUID PRIMARY KEY,
    name TEXT,
    price DECIMAL,
    category TEXT,
    last_ordered TIMESTAMP
);

-- Separate table for counters
CREATE TABLE product_sales (
    product_id UUID PRIMARY KEY,
    total_orders COUNTER
);
```

Key Denormalization Techniques Used

- Data Duplication: User/order/product info appears in multiple tables
- Collections: Using maps/lists to store related data together
- Query-First Design: Each table serves a specific query pattern
- Time-Ordered Data: Clustering by timestamp for time-series patterns
- Counter Columns: For aggregated data that changes frequently

Normalization vs Denormalization

Denormalization vs. Normalization

	Denormalization	Normalization
PROS	+ Faster data reads + Simpler queries advantageous to developers + Requires less compute on read operations + Makes data available quickly	+ Faster writes + No redundant data + Less database complexity + Data always consistent
CONS	– Slower writes – More database complexity – Potential for data inconsistency – Requires more storage, RAM	– Slower reads – Heavy querying can overwhelm, crash hardware – Table joins required since data isn't duplicated – Indexing not as efficient due to table joins

Handling Data Duplication

- Application-Managed Duplication
- Change Data Capture (CDC)
  - Triggers
  - Events

Denormalization Strategies

- Denormalization involves strategically duplicating data across multiple tables to optimize read performance.

- This is a core principle because Cassandra does not support joins or derived tables like relational databases, so denormalization is key for efficient query access.

- Instead of minimizing redundancy like in normalized relational databases, Cassandra prioritizes data access patterns by replicating data for faster retrieval.

Denormalization Strategies

Data Duplication

- Social media application might have tables for posts by **user**, posts by **topic**, and posts by **date**
- Video-Sharing application might store comments in both **comments_by_video** and **comments_by_user** tables.
- User profile data duplicated in tables organized by **user_id**, **email**, and **username**

Denormalization Strategies

Partition-Centric Design

- Co-Locate Related Data
  - Store data that's frequently accessed together in the same partition
- Partition Key Selection
  - Choose keys that align with your most common query patterns
- Example:
  - All order items stored together with the order header in a single partition

Denormalization Strategies

Composite Partition Keys

- Multi-column Partitioning:
  - Combine multiple fields to create optimal partitions
- Example
  - (country, city) as a partition key for location-based queries

Denormalization Strategies

Aggregation at Write Time

- Pre-Calculate Aggregates
    - Store summary data that would otherwise require expensive scans
- Example:
    - Maintain running totals or **counters** instead of calculating on read

Denormalization Strategies

Bucketing Strategy

- Control Partition Size:
  - Use time buckets or other ranges to prevent unbounded partition growth
- Example
  - (user_id, year_month) as partition key for user activity data

Querying

- Data Types
- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Dynamic Data Masking (DDM)
- CQL (Cassandra Query Language)
- Query Optimization
- Advanced CQL Features

Data Types

- Native Types
- Collection Types
- User-Defined Types
- Tuple Types
- Custom Types

Native types

- ASCII | BIGINT | BLOB | BOOLEAN | COUNTER | DATE
- DECIMAL | DOUBLE | DURATION | FLOAT | INET | INT
- SMALLINT | TEXT | TIME | TIMESTAMP | TIMEUUID | TINYINT
- UUID | VARCHAR | VARINT | VECTOR

Counter Data Type

- Specifically designed for counting purposes.
- It allows you to perform atomic increment/decrement operations in a distributed environment
- No Other Updates: Once created, counters can only be incremented or decremented (no other updates allowed)

Counter Data Type Operation

```
* Incrementing a counter

UPDATE page_views SET views = views + 1 WHERE page_id = 'home';

* Decrementing a counter

UPDATE page_views SET views = views - 1 WHERE page_id = 'home';

* Incrementing by a specific value

UPDATE page_views SET views = views + 5 WHERE page_id = 'home';
```

Important Considerations

- Counter Tables:
    - Can only contain counter columns (plus primary key columns)
    - Cannot have non-counter columns
- Performance:
    - Counter updates require read-before-write (more expensive than regular writes)
    - Not recommended for high-frequency updates
- Limitations:
    - Cannot set a counter to a specific value (only increment/decrement)
    - Cannot be used in conditional updates
    - Not idempotent (retries may cause multiple counts)
- Replication:
    - Counters use a special replication mechanism to handle concurrent updates

Counter Example Table

```
CREATE TABLE user_actions (
    user_id text,
    action_date date,
    likes counter,
    shares counter,
    PRIMARY KEY (user_id, action_date)
) WITH CLUSTERING ORDER BY (action_date DESC);
```

Administration

- Cluster Setup and Configuration
- Monitoring and Troubleshooting
- Backup and Recovery

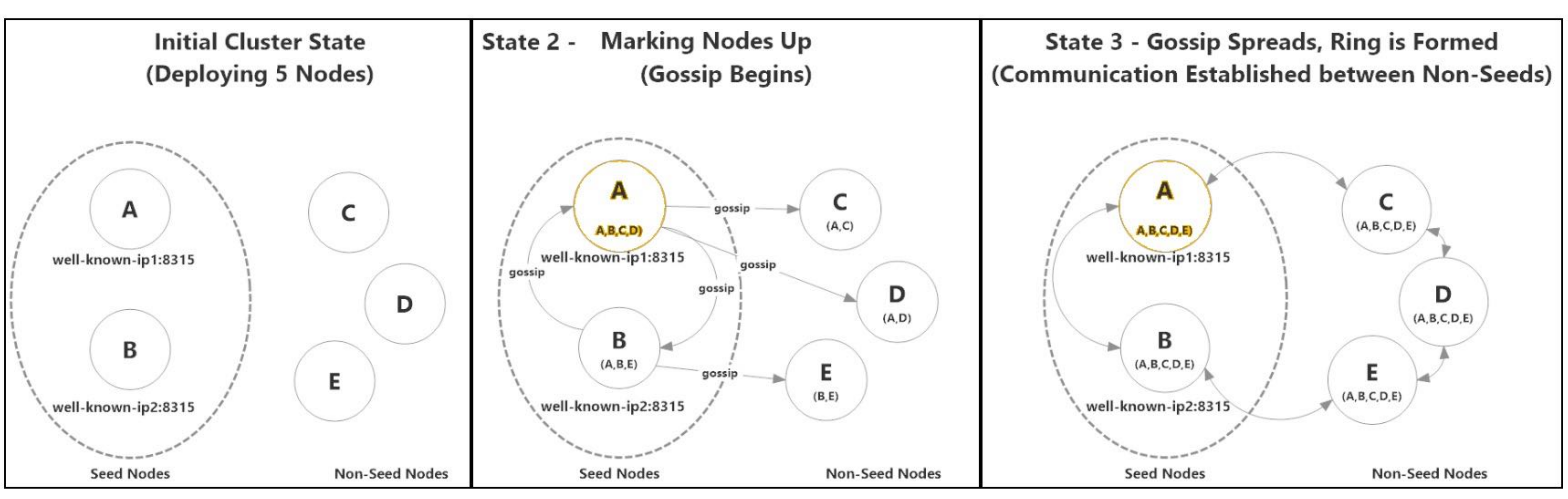
Development

- Client Drivers
- Data Consistency and Durability
- Performance Optimization

DevOps

- Integrating with DevOps Tools

High Availability



Direct vs Digest Requests

- Direct request involves the coordinator node directly contacting one replica for the data.

- Digest requests involve contacting multiple replicas and checking for data consistency by comparing digests (hashes) of the data

Direct vs Digest Requests

Feature	Direct Request	Digest Request
Primary Goal	Quickly retrieve data from one replica	Ensure data consistency across multiple replicas
Consistency	Limited consistency guarantees	Higher consistency guarantees
Performance	Potentially faster	Potentially slower due to additional network traffic and data comparison
Data Delivery	Full data retrieval	Digest (hash) of the data, not the full data
Consistency Verification	No direct verification	Uses digests to check for data consistency

Repair Request

- A repair request initiates the process of resolving data inconsistencies between replicas.
    - These inconsistencies can arise when nodes fail
    - When writes are not synchronized across all replicas.
- Repairs ensure data accuracy and consistency within the cluster, which is crucial for maintaining data integrity.

Read Repair vs Repair Request

- **Read Repair** is a process that occurs during a read operation to ensure data consistency across replicas if inconsistencies are detected.
- **Repair Request** is a broader term that refers to any request to reconcile data between replicas, whether it's during a read or as a separate maintenance task.

Repair Command

- ***nodetool repair***
  - Initiates an incremental repair.
- ***nodetool repair --full***
  - Initiates a full repair.
- ***nodetool repair [keyspace_name]***
  - Repairs a specific keyspace.
- ***nodetool repair [keyspace_name] [table1] [table2]:***
  - Repairs specific tables within a keyspace.