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

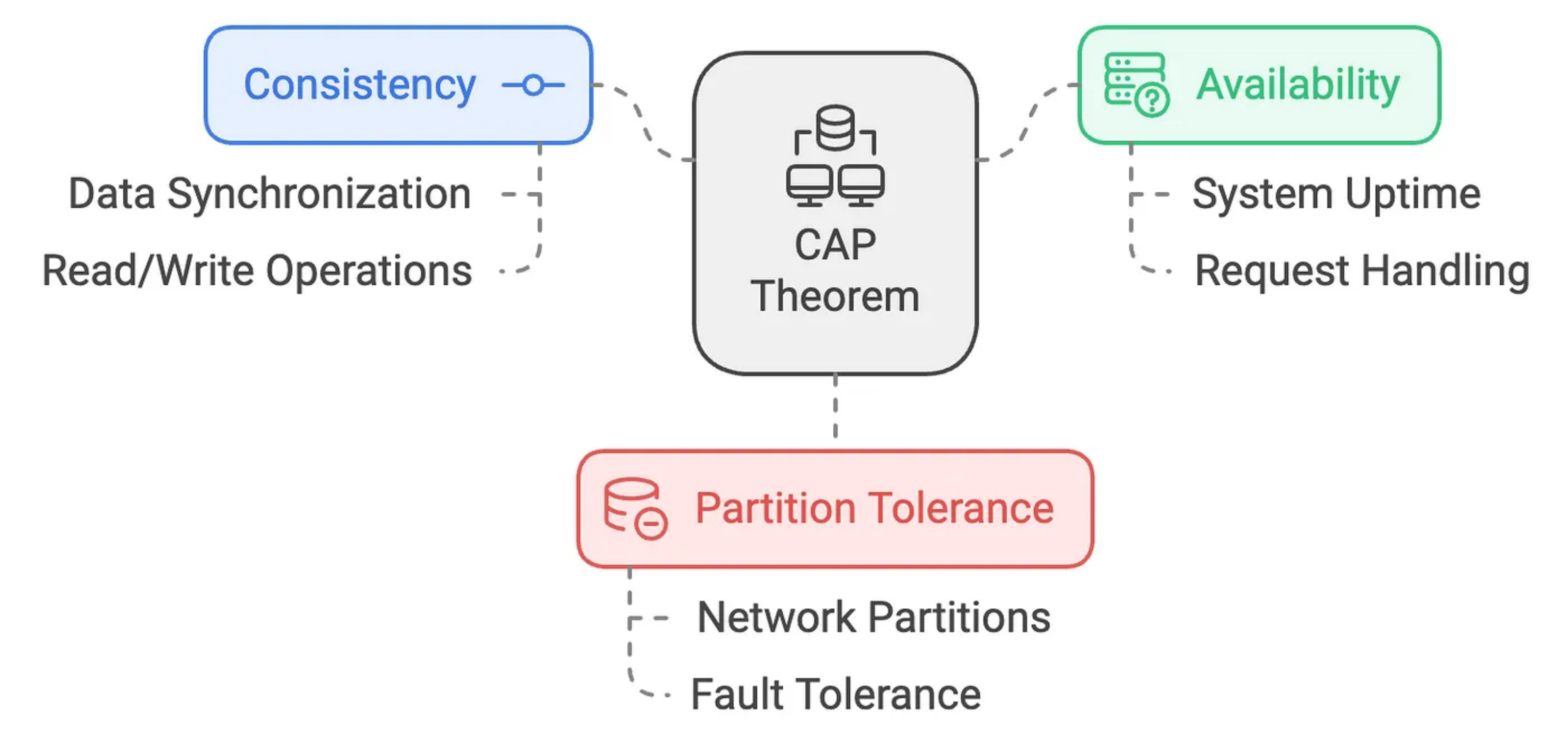
- Great Application where **Data** is collected at **High Speed** from **Different** kinds of **sources**
- Internet of Things
- Product & Retail apps
- Messaging
- Social Media Analytics
- Recommendation Engine

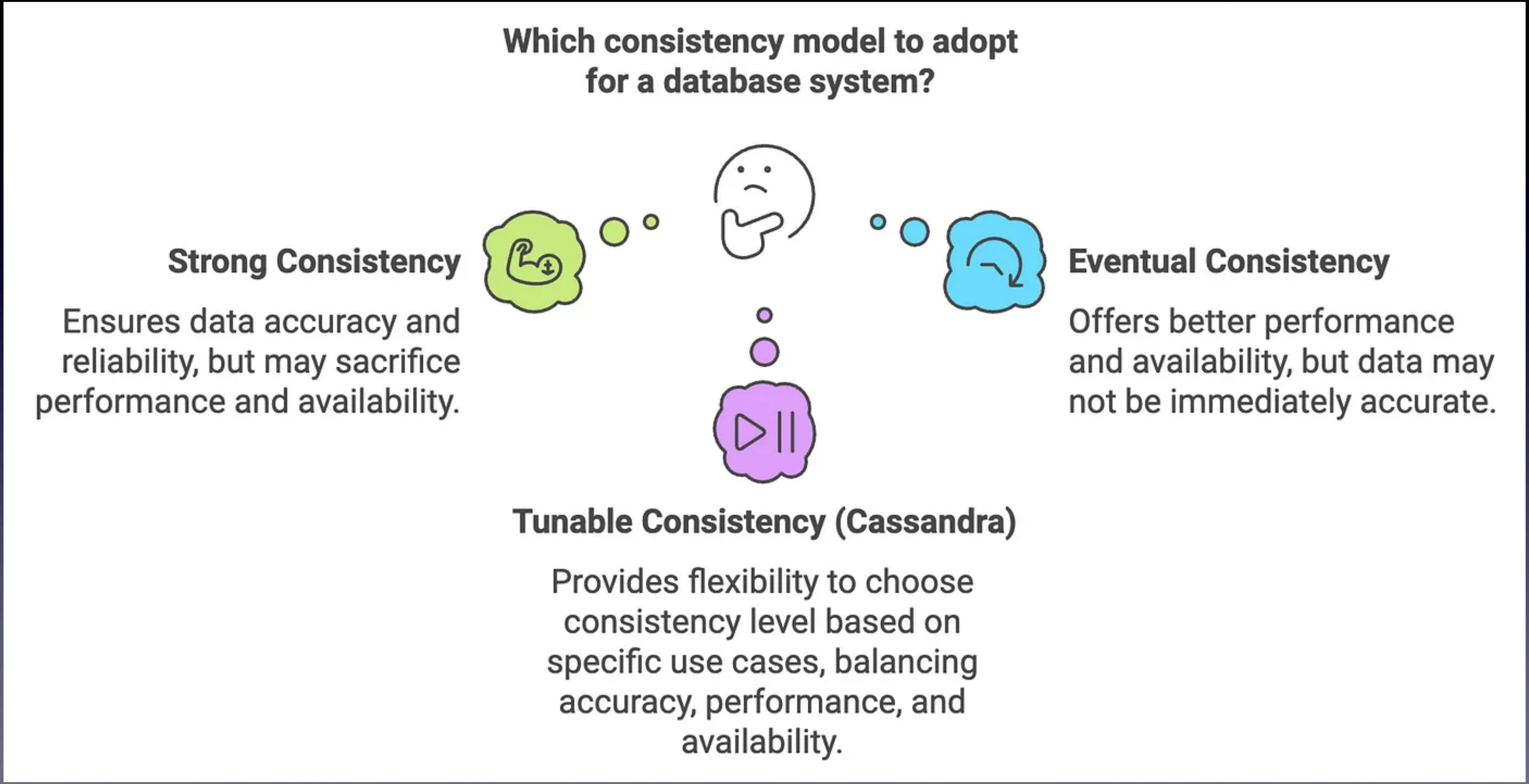
Traditional RDBMS vs Cassandra

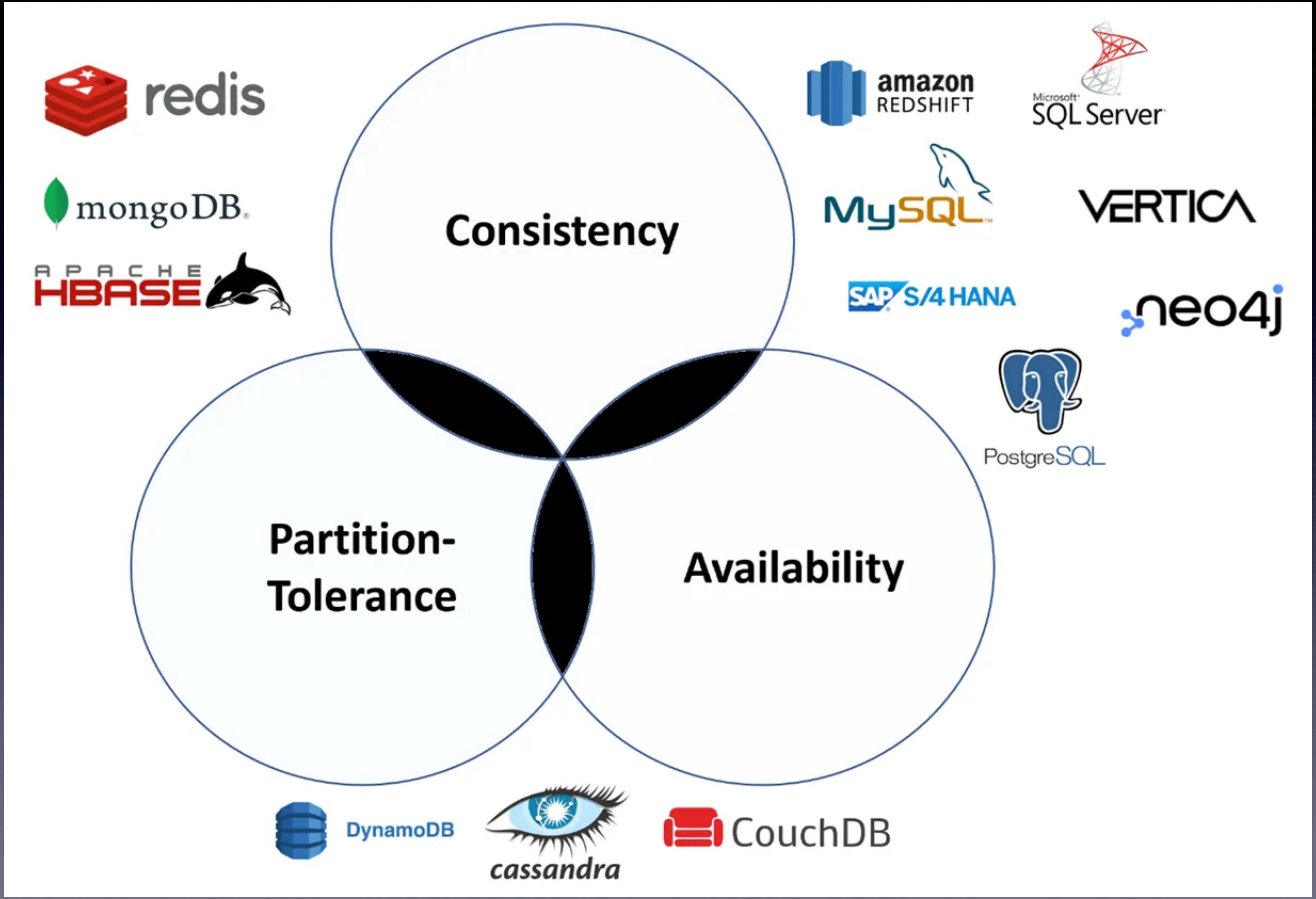
Feature	Traditional RDBMS	Cassandra
Data Model	Relational	NoSQL, flexible column families
Architecture	Centralized/Master-Slave	Distributed, Peer-to-peer
Scalability	Vertical, complex horizontal	Horizontal, easy to scale out
Consistency	Strong (ACID)	Tunable, often eventually consistent
Transactions	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.

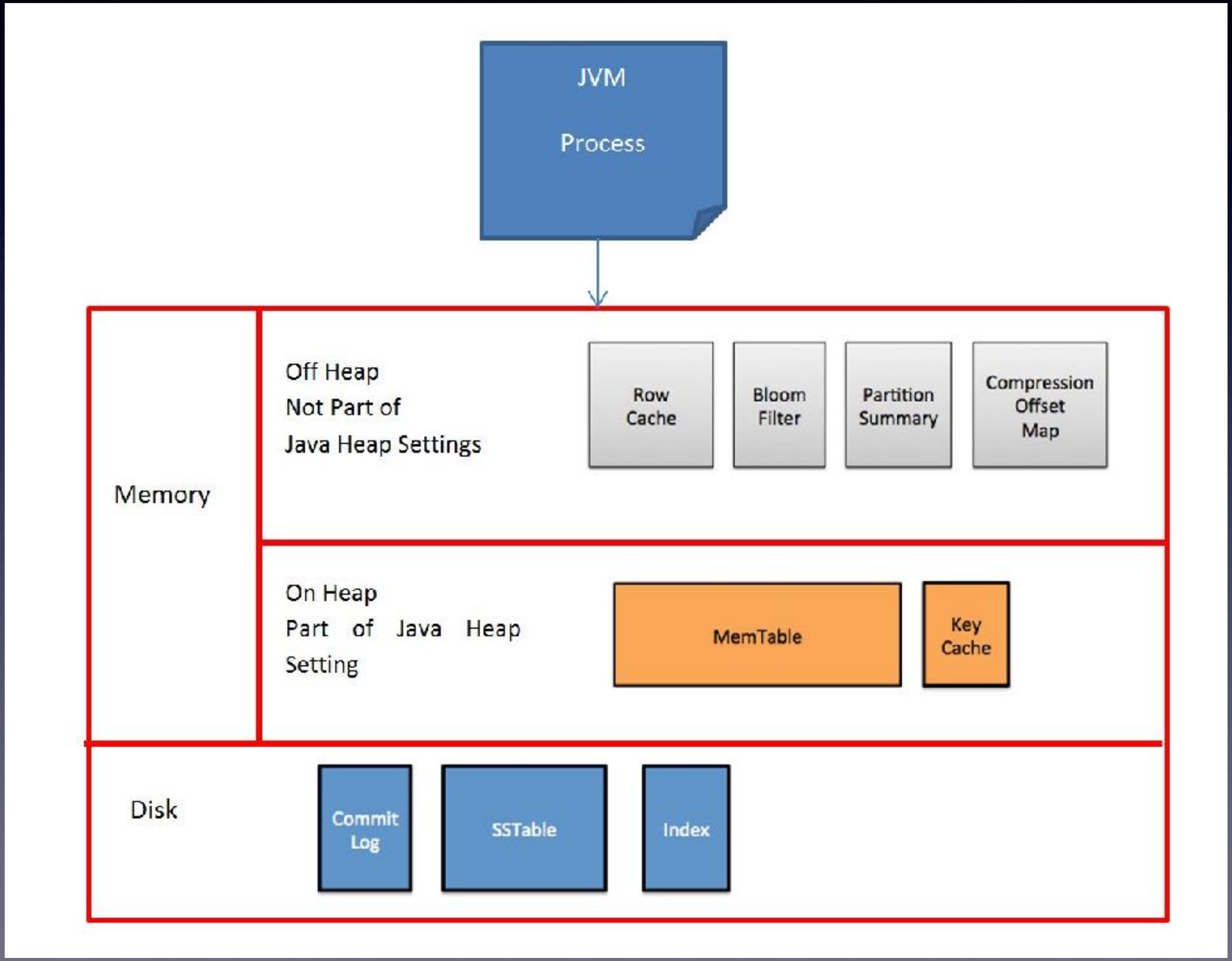
Cassandra MemTable

- 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.

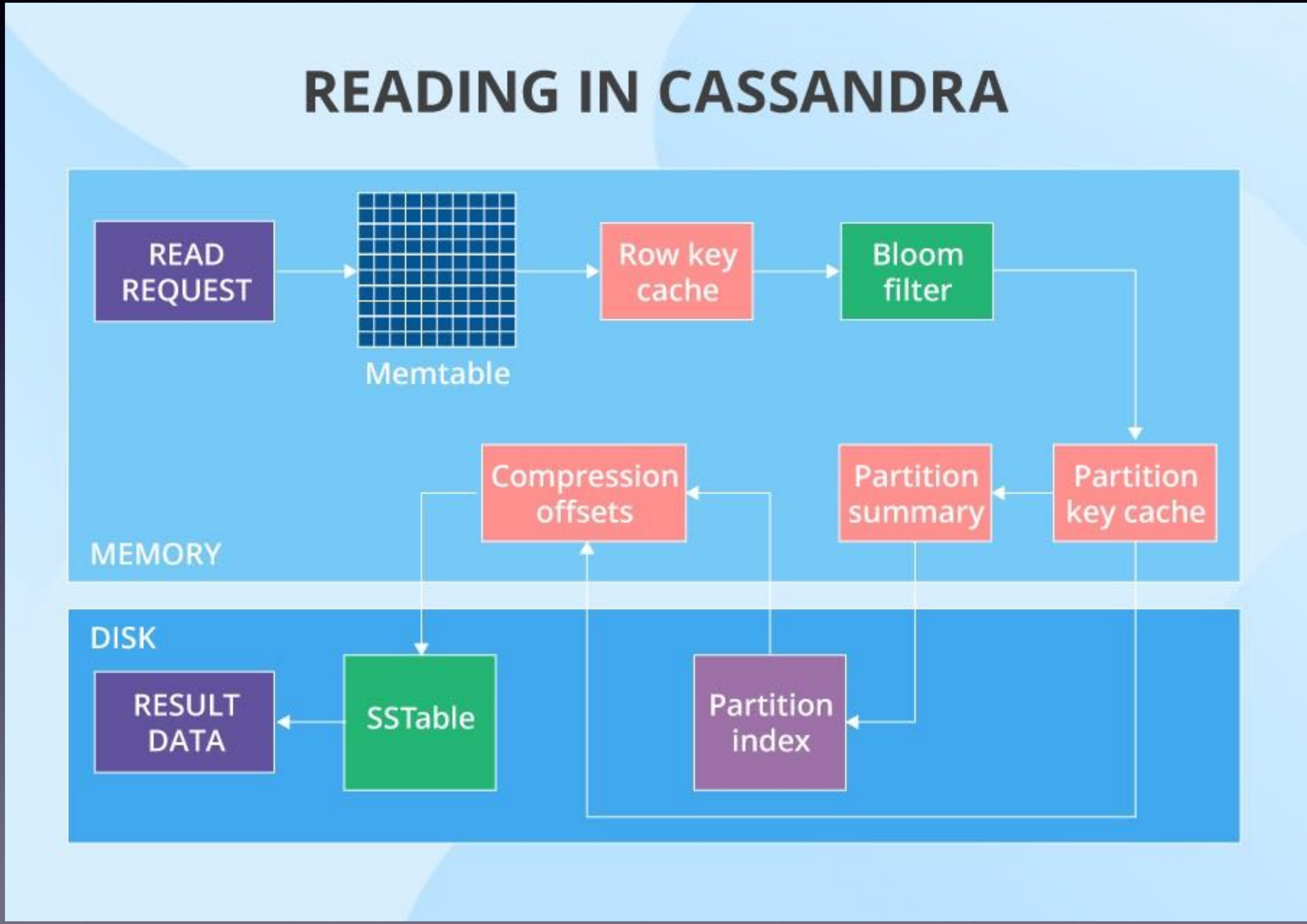
Cassandra SSTable

- 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.

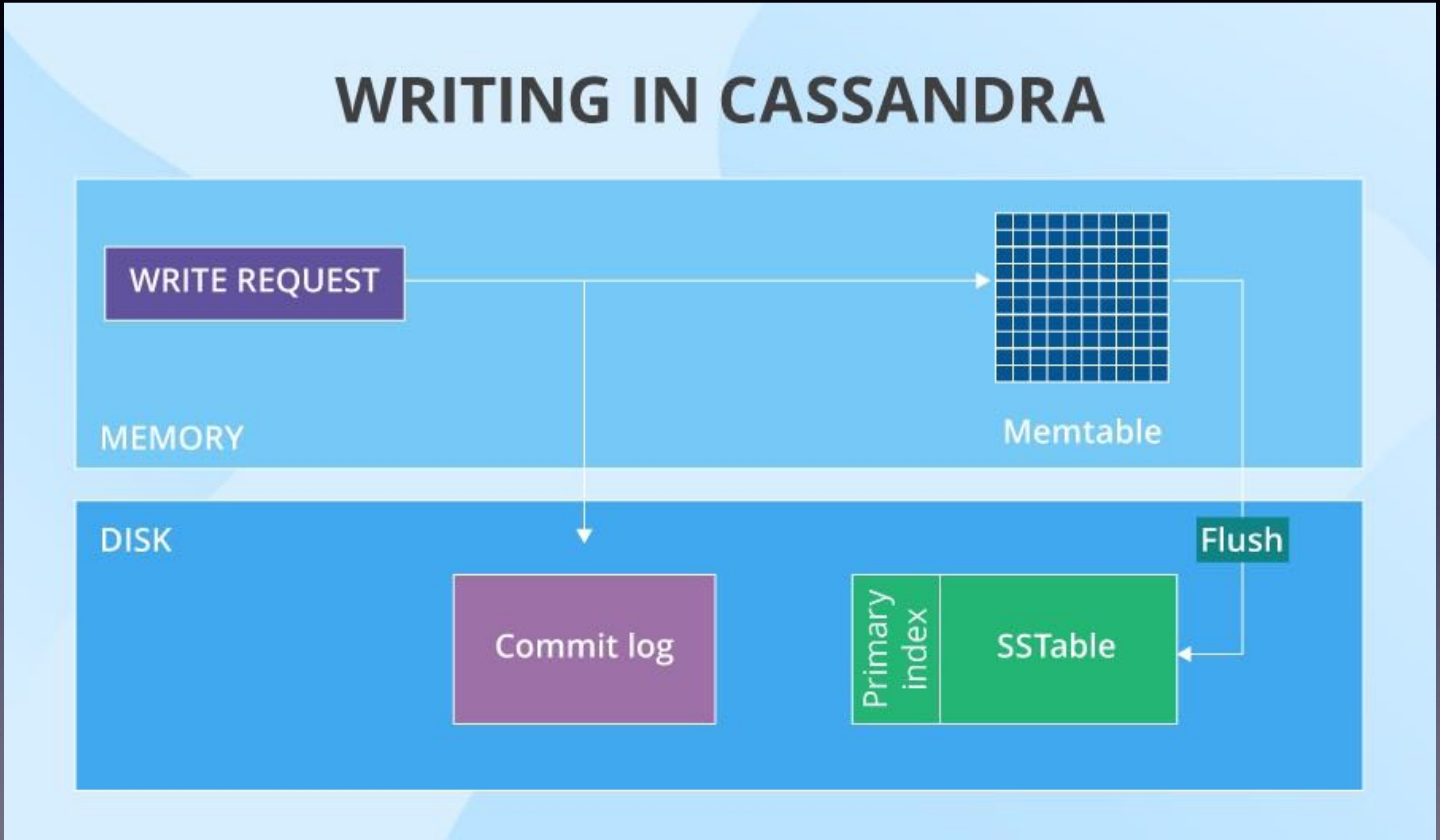
Architecture



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

```
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")

```
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);
```

Duration

- Store Periods of time with nanosecond precision.
- It represents a time duration with months, days, and nanoseconds components.
- Specified using ISO 8601 format
    - P[n]Y[n]M[n]DT[n]H[n]M[n]S
- Cannot be Part of a Primary Key
- Measuring time intervals
- Storing Service-Level Agreement (SLA) durations
- Representing Timeouts or Delays
- Tracking process execution times

Scenario: Order Delivery Tracking

```
CREATE TABLE order_delivery_metrics (
    order_id UUID PRIMARY KEY,
    customer_id UUID,
    order_date TIMESTAMP,

    -- Durations
    estimated_delivery_time DURATION,
    actual_processing_time DURATION,

    -- Other relevant fields
    shipping_method TEXT,
    delivery_region TEXT
);
```

```sql
-- Standard delivery (3 business days)
INSERT INTO order_delivery_metrics (order_id, customer_id, order_date, estimated_delivery_time, shipping_method)
VALUES (
    uuid(),
    uuid(),
    '2023-11-15 14:30:00',
    3d,  -- 3 days
    'standard'
);

-- Express delivery (12 hours)
INSERT INTO order_delivery_metrics (order_id, customer_id, order_date, estimated_delivery_time, shipping_method)
VALUES (
    uuid(),
    uuid(),
    '2023-11-15 09:15:00',
    12h,  -- 12 hours
    'express'
);

-- International order (7 days + 12 hours)
INSERT INTO order_delivery_metrics (order_id, customer_id, order_date, estimated_delivery_time, shipping_method)
VALUES (
    uuid(),
    uuid(),
    '2023-11-15 10:00:00',
    P7DT12H,  -- 7 days and 12 hours (ISO 8601 format)
    'international'
);
```

```sql
-- Order processed in 2 hours 45 minutes
UPDATE order_delivery_metrics
SET actual_processing_time = 2h45m
WHERE order_id = [order_id];

-- Order processed in 1 day, 3 hours, 20 minutes
UPDATE order_delivery_metrics
SET actual_processing_time = P1DT3H20M
WHERE order_id = [order_id];
```

Find Orders That Missed Delivery SLA

```
SELECT order_id, customer_id
FROM order_delivery_metrics
WHERE actual_processing_time >
estimated_delivery_time;
```

Calculate Average Processing Time By Shipping Method

```
SELECT shipping_method,
avg(actual_processing_time) as avg_time
FROM order_delivery_metrics
GROUP BY shipping_method;
```

Find Express Deliveries that Took Longer Than 8 Hours

```
SELECT order_id
FROM order_delivery_metrics
WHERE shipping_method = 'express'
AND actual_processing_time > 8h;
```

Collection Data Types

- Cassandra provides three collection data types that allow you to store multiple values in a single column
    - Sets
    - Lists
    - Maps
- These are useful for denormalized data models where you want to group related information together.

SET (Unordered Unique Values)

```
CREATE TABLE user_profiles (
    user_id UUID PRIMARY KEY,
    email TEXT,
    phone_numbers SET<TEXT>,  -- Stores unique
phone numbers
    tags SET<TEXT>            -- Stores unique tags
);
```

```
-- Add elements
UPDATE user_profiles
SET phone_numbers =
phone_numbers + {'+1-555-1234'}
WHERE user_id = ?;
```

```
-- Remove elements
UPDATE user_profiles
SET phone_numbers = phone_numbers -
{'+1-555-1234'}
WHERE user_id = ?;
```

```
-- Replace entire set
UPDATE user_profiles
SET phone_numbers = {'+1-555-5678',
'+44-555-1234'}
WHERE user_id = ?;
```

LIST (Ordered with Duplicates)

```sql
CREATE TABLE playlists (
    playlist_id UUID PRIMARY KEY,
    title TEXT,
    songs LIST<TEXT>);
```

```sql
-- Prepend to list
UPDATE playlists
SET songs = ['Bohemian Rhapsody'] +
songs
WHERE playlist_id = ?;
```

```sql
-- Append to list
UPDATE playlists
SET songs = songs + ['Stairway to Heaven']
WHERE playlist_id = ?;
```

```sql
-- Set element at position (0-based)
UPDATE playlists
SET songs[2] = 'Hotel California'
WHERE playlist_id = ?;
```

```sql
-- Remove by value (all occurrences)
UPDATE playlists
SET songs = songs - ['Hey Jude']
WHERE playlist_id = ?;
```

MAP (Key-Value Pairs)

```
CREATE TABLE product_inventory (
    product_id UUID PRIMARY KEY,
    name TEXT,
    prices MAP<TEXT, DECIMAL>,
    attributes MAP<TEXT, TEXT>
);
```

- Store Key-Value Pairs
- Keys must be unique
- Both keys and values have data types

Map Statements

```sql
CREATE TABLE product_inventory (
    product_id UUID PRIMARY KEY,
    name TEXT,
    prices MAP<TEXT, DECIMAL>,
    attributes MAP<TEXT, TEXT>
);
```

```sql
-- Remove by key
UPDATE product_inventory
SET prices = prices - {'EUR'}
WHERE product_id = ?;
```

```sql
-- Add/update a key-value pair
UPDATE product_inventory
SET prices = prices + {'USD': 19.99}
WHERE product_id = ?;
```

```sql
-- Update specific key
UPDATE product_inventory
SET prices['USD'] = 24.99
WHERE product_id = ?;
```

VECTOR

- It is used to store vectors, which are **arrays of fixed length**, for vector search.
- It's a new data type introduced in Cassandra 5.0 and is essential for storing and searching high-dimensional vectors, like those used in AI and machine learning.

Vector Features

- **Fixed Length**: Vectors have a predefined dimension, meaning the number of elements in the array is set at creation.
- **Non-Null Elements**: All elements within a vector must be non-null.
- **Flattened Array**: Vectors are flattened, meaning they are stored as a single array rather than a nested structure.
- **Maximum Dimension**: Vectors are limited to a maximum dimension of 8K (2^13) items.
- **Vector Search**: The VECTOR data type is designed for vector search, enabling efficient retrieval of data based on semantic similarity.

Benefits of Vector Search

- **Semantic Similarity**: Vector search allows you to find data that is semantically similar based on the relationships encoded in the vectors.

- **AI Applications**: It's particularly useful for storing and searching data related to machine learning, image recognition, and natural language processing.

- **Efficient Retrieval**: When combined with storage-attached indexing (SAI), vector search can significantly improve the speed and efficiency of searching large datasets.

What is Vector Search?

- Vector search is a cutting-edge approach to searching and retrieving data that leverages the power of vector similarity calculations.
- Unlike traditional **keyword-based search**, which matches documents based on the occurrence of specific terms, vector search focuses on the **semantic meaning and similarity** of data points.
- By representing data as vectors in a high-dimensional space, vector search enables more accurate and intuitive search results.
- “Man bites dog” and “dog bites man” include the same words but have opposite semantics.
- “Tourism numbers are collapsing” and “Travel industry fears Covid-19 crisis will cause more companies to enter bankruptcy” have very similar meanings but different word choices and specificity.
- “I need a new phone” and “My old device is broken” have related meanings but no common words.

UDT

- Allow you to define custom data structures, grouping related fields into a single logical unit, Similar to struct in C or class in Java.

- They improve schema clarity, flexibility and reduce data duplication.

- Schema Enforcement – Fields must match defined types (unlike JSON blobs).

UDT Sample

```
CREATE TYPE phone (
    country_code int,
    number text,
);
```

```
CREATE TYPE address (
    street text,
    city text,
    zip text,
    phones map<text, phone>
);
```

```
CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, frozen<address>>
);
```

UDT Operation

```
INSERT INTO user (name, addresses)
   VALUES ('z3 Pr3z1den7', {
     'home' : {
        street: '1600 Pennsylvania Ave NW',
        city: 'Washington',
        zip: '20500',
        phones: { 'cell' : { country_code: 1, number: '202 456-1111' },
                  'landline' : { country_code: 1, number: '...' } }
     },
     'work' : {
        street: '1600 Pennsylvania Ave NW',
        city: 'Washington',
        zip: '20500',
        phones: { 'fax' : { country_code: 1, number: '...' } }
     }
   }
);
```

Tuple

- It is a fixed-length, ordered collection of elements where each position has a specific data type.

- Tuples are anonymous (unnamed) and immutable (cannot be updated partially).

Tuple Sample

```sql
-- Create a table with a tuple
CREATE TABLE weather (
    station_id text PRIMARY KEY,
    last_reading TUPLE<float, timestamp, text>  -- (temp, time, condition)
);

-- Insert data
INSERT INTO weather (station_id, last_reading)
VALUES ('STN_001', (72.3, toTimestamp(now()), 'sunny'));

-- Query specific elements
SELECT
    station_id,
    last_reading[0] AS temperature,
    last_reading[2] AS weather
FROM weather;
```

frozen Keyword

- A column whose type is a frozen collection (set, map, list or UDT) can only have its value **replaced as a whole**.
- We **can't add, update, or delete individual elements** from the collection as we can in non-frozen collection types.
- Can be useful, when we want to protect collections against single-value updates.
- We can use a frozen collection as the primary key in a table.

frozen Operations

```
CREATE TABLE users
(
    id          uuid PRIMARY KEY,
    ip_numbers frozen<set<inet>>,
);
```

```
INSERT INTO users (id, ip_numbers)
VALUES (6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47,
{'10.10.11.1', '10.10.10.1', '10.10.12.1'});
```

frozen Operations

```
UPDATE users
SET ip_numbers = ip_numbers + {'10.10.14.1'}
WHERE id = 6ab09bec-e68e-48d9-
a5f8-97e6fb4c9b47;
```

```
InvalidRequest: Error from server: code=2200
[Invalid query]
message="Invalid operation (ip_numbers =
ip_numbers + {'10.10.14.1'}) for frozen
collection column ip_numbers"
```

frozen on UDT

- frozen on a UDT treats the value like a blob.
- This blob is obtained by serializing our UDT to a single value.
- We can’t update parts of a user-defined type value.
- We have to overwrite the entire value.

frozen on Tuples

- Tuple is always frozen.
- We don’t have to mark tuples with the frozen keyword.
- It is not possible to update only some elements of a tuple.
- We have to overwrite the entire value.

When to Use frozen

- When you need to **nest** collections
- When you want to enforce **atomic updates** of the entire structure
- When using collections in **primary keys**
- When storing collections within UDTs

frozen Performance Considerations

- Reads: Frozen values are read as a single unit, which can be more efficient
- Writes: Entire structure must be rewritten for any change
- Storage: Frozen values typically use slightly more storage space

Data Definition

- CQL stores data in **Tables**, whose schema defines the layout of the data in the table.
- Tables are located in **Keyspaces**.
- A keyspace defines options that apply to all the keyspace's tables.
- The **replication** strategy is an important keyspace option, as is the replication factor.
- A good general rule is **one keyspace per application**.
- It is common for a cluster to define **only one keyspace for an active application.**

KeySpace & Table Name Grammar

- keyspace_name::= name
- table_name::= [keyspace_name '.'] name
- name::= unquoted_name | quoted_name
- unquoted_name::= re('[a-zA-Z_0-9]\{1, 48}')
- quoted_name::= '"' unquoted_name '"'

Keyspace name

- Both keyspace and table name should be comprised of only alphanumeric characters, cannot be empty and are limited in size to 48 characters (that limit exists mostly to avoid filenames (which may include the keyspace and table name) to go over the limits of certain file systems).

- By default, keyspace and table names are case-insensitive (myTable is equivalent to mytable) but case sensitivity can be forced by using double-quotes ("myTable" is different from mytable).

- Further, a table is always part of a keyspace and a table name can be provided fully-qualified by the keyspace it is part of. If is is not fully-qualified, the table is assumed to be in the current keyspace

Replication Strategy

- **SimpleStrategy**
    - Defines a replication factor for data to be spread across the **entire cluster**.
- **NetworkTopologyStrategy**
    - Sets the replication factor independently for **each data-center**

Basic Syntax

```
CREATE KEYSPACE keyspace_name
WITH REPLICATION = {
    'class':
'ReplicationStrategyClass',
    'replication_factor': number |
'datacenter1': number, 'datacenter2':
number, ...
};
```

Changing Replication

```
ALTER KEYSPACE my_keyspace
   WITH REPLICATION = {
       'class':
'NetworkTopologyStrategy',
       'datacenter1': 4,
       'datacenter2': 2
   };
```

auto-expansion

- When later altering keyspaces and changing the replication_factor, auto-expansion will only add new datacenters for safety, it will not alter existing datacenters or remove any, even if they are no longer in the cluster.

```
CREATE KEYSPACE testA
    WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' :
3};

DESCRIBE KEYSPACE testA;


CREATE KEYSPACE testA
    WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3', 'DC2':
'2'}
    AND durable_writes = true;


CREATE KEYSPACE testB
   WITH replication = {'class': 'NetworkTopologyStrategy', 'replication_factor' :
3, 'DC2': 0};

DESCRIBE KEYSPACE testB;

CREATE KEYSPACE testB
    WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3'}
    AND durable_writes = true;
```

Benefits of Higher Replication Factor

- Increased Availability
- Improved Read Performance
- Better Fault Tolerance
- Geographic Distribution

Costs of Higher Replication Factor

- Increased Storage Requirements
- Write Performance Impact
- Network Overhead
- Compaction Overhead

Recommended Replication Factors

- Development / Testing
    - RF=1 (single copy)
- Single Production DC
    - RF=3 (standard for most production deployments)
- Multi-DC Production
    - RF=3 per DC (e.g., 'dc1':3, ‘dc2':3)'
- Critical Systems
    - RF=5 for ultra-high availability

CREATE KEYSPACE

```
create_keyspace_statement::= CREATE KEYSPACE [ IF NOT
EXISTS ] keyspace_name
    WITH options
```

```
CREATE KEYSPACE rayan
   WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 3};
```

```
CREATE KEYSPACE rayan
   WITH replication = {'class':
'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}
   AND durable_writes = false;
```

Durable Writes in Cassandra

- Durable writes control whether the commit log is used for writes to the keyspace (default: true).
- The commit log provides crash recovery by persisting writes before they're applied to memtables.
-

Effects of Durable Writes Enabled (by default)

- All writes go to both commit log and memtable
- Guarantees data isn't lost if a node crashes
- Adds slight write latency (typically 10-20%)
- Recommended for all production systems

Effects of Durable Writes Disabled

- Writes only go to memtable (no commit log)
- Better write performance (no disk I/O for commit log)
- Risk of data loss if node crashes before memtable is flushed to SSTable
- Only suitable for temporary data or test environments

Safe Scripting Tips

- Attempting to create a keyspace that already exists will return an error unless the **IF NOT EXISTS** option is used.
- If it is used, the statement will be a no-op if the keyspace already exists.

Transient Replication

- Introduced in Cassandra 4.0
- Provides a more efficient way to handle temporary copies of data while maintaining fault tolerance.
- Don't permanently store the data
- Only keep the data temporarily (until it's streamed to permanent replicas)
- Act as temporary stand-ins for failed nodes

How It Works

- When writing data, it's sent to both full and transient replicas
- Transient replicas hold the data only until the system confirms it's properly stored on the full replicas
- Once the data is safely on the full replicas, it can be discarded from transient replicas

```
CREATE KEYSPACE my_keyspace
WITH REPLICATION = {
    'class': 'NetworkTopologyStrategy',
    'dc1': '3/1'  // 3 total replicas with 1 transient
};
```

```
Here, "3/1" means:
- 3 total replicas
- 1 of which is transient
- 2 are full replicas
```

Benefits

- Reduced Storage Requirements
  - Fewer permanent copies needed
  - Transient replicas don't add to long-term storage needs
- Maintained Availability
  - Still provides temporary redundancy during failures
  - Can satisfy consistency level requirements during outages
- Improved Performance
  - Less data to compact and maintain long-term
  - Faster bootstrap of new nodes
- Flexible Topology
  - Can use cheaper hardware for transient replicas
  - Good for temporary capacity expansion

Use Cases

- Temporary Node Failures
  - When a node goes down, transient replicas can temporarily take its place
- Maintenance Windows
  - During rolling upgrades or maintenance
- Cost Optimization
  - When you want redundancy but want to minimize storage costs
- Cloud Deployments
  - Where temporary resources are easily provisioned

Limitations

- Not for Permanent Storage
  - Transient replicas shouldn't be your only copies
  - Must maintain sufficient full replicas
- Cassandra 4.0+
  - Only available in newer Cassandra versions
- Consistency Level Considerations
  - Need to adjust consistency levels appropriately
  - QUORUM calculations include both full and transient replicas

Administration

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

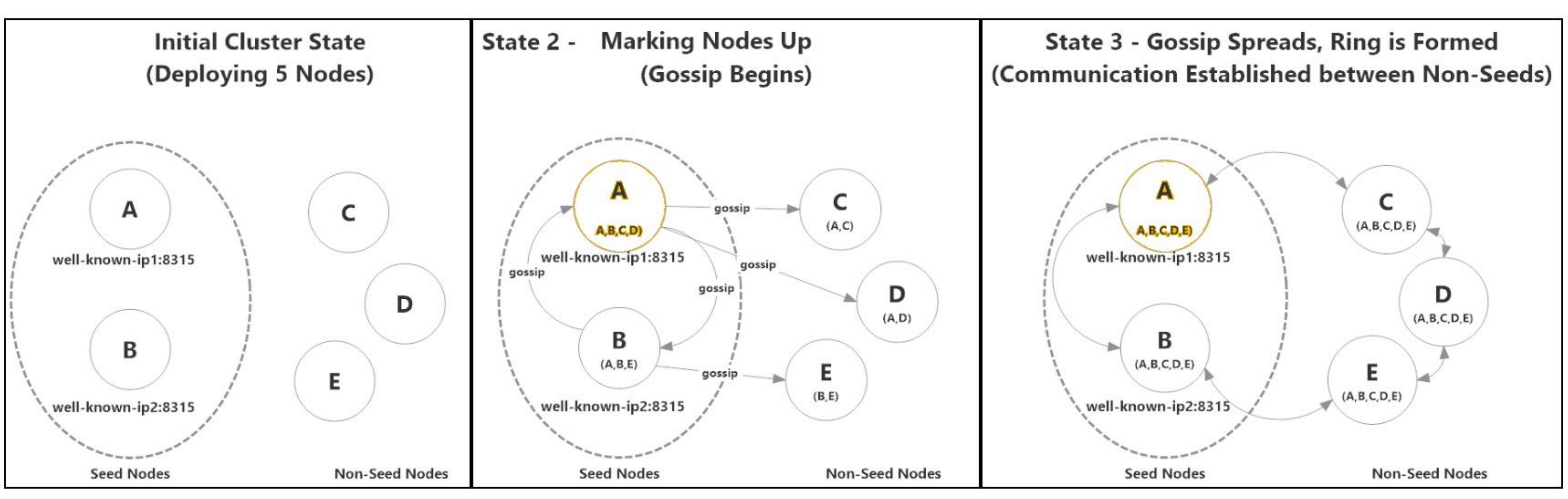
Development

- 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.