UNIVERSITY OF CENTRAL MISSOURI

DSA 5600 – NoSQL Database Systems
Section 3: Apache Cassandra

Instructor: Yutong Zhao

What is Cassandra?

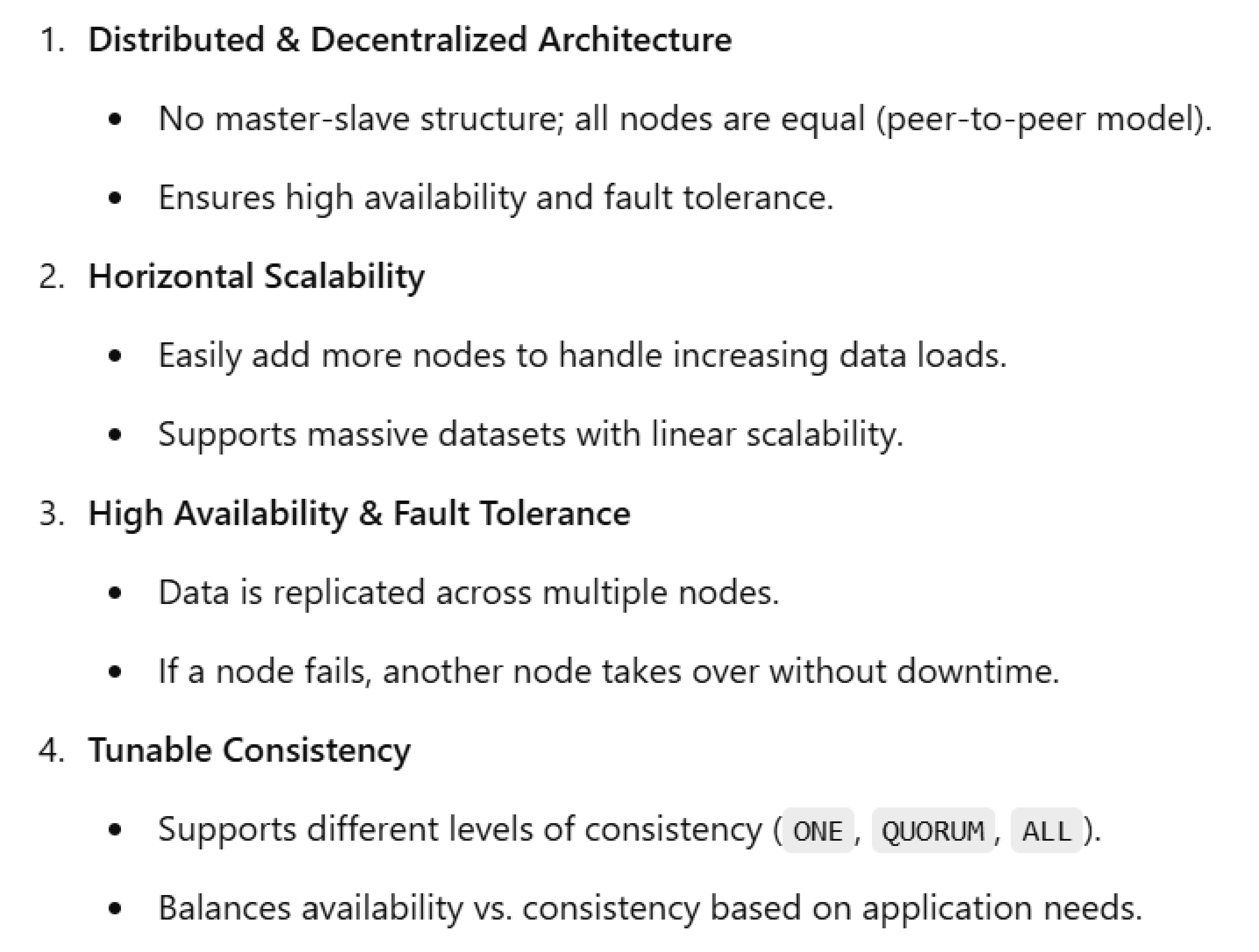
- **Apache Cassandra** is a **distributed, NoSQL database** designed for handling large volumes of structured data across many servers without a single point of failure.

- Originally developed at **Facebook** to power their inbox search, later open-sourced and adopted by the **Apache Software Foundation**.

- **Key Characteristics:**
    - **Scalability:** Easily scales horizontally by adding more nodes.
    - **High Availability:** No single point of failure; data is replicated across multiple nodes.
    - **NoSQL Model:** Uses a schema-less, column-family-based storage model.
    - **Optimized for Write-Intensive Workloads:** Handles high-speed inserts and updates efficiently.
    - **Eventual Consistency:** Ensures availability over strict consistency (CAP theorem).

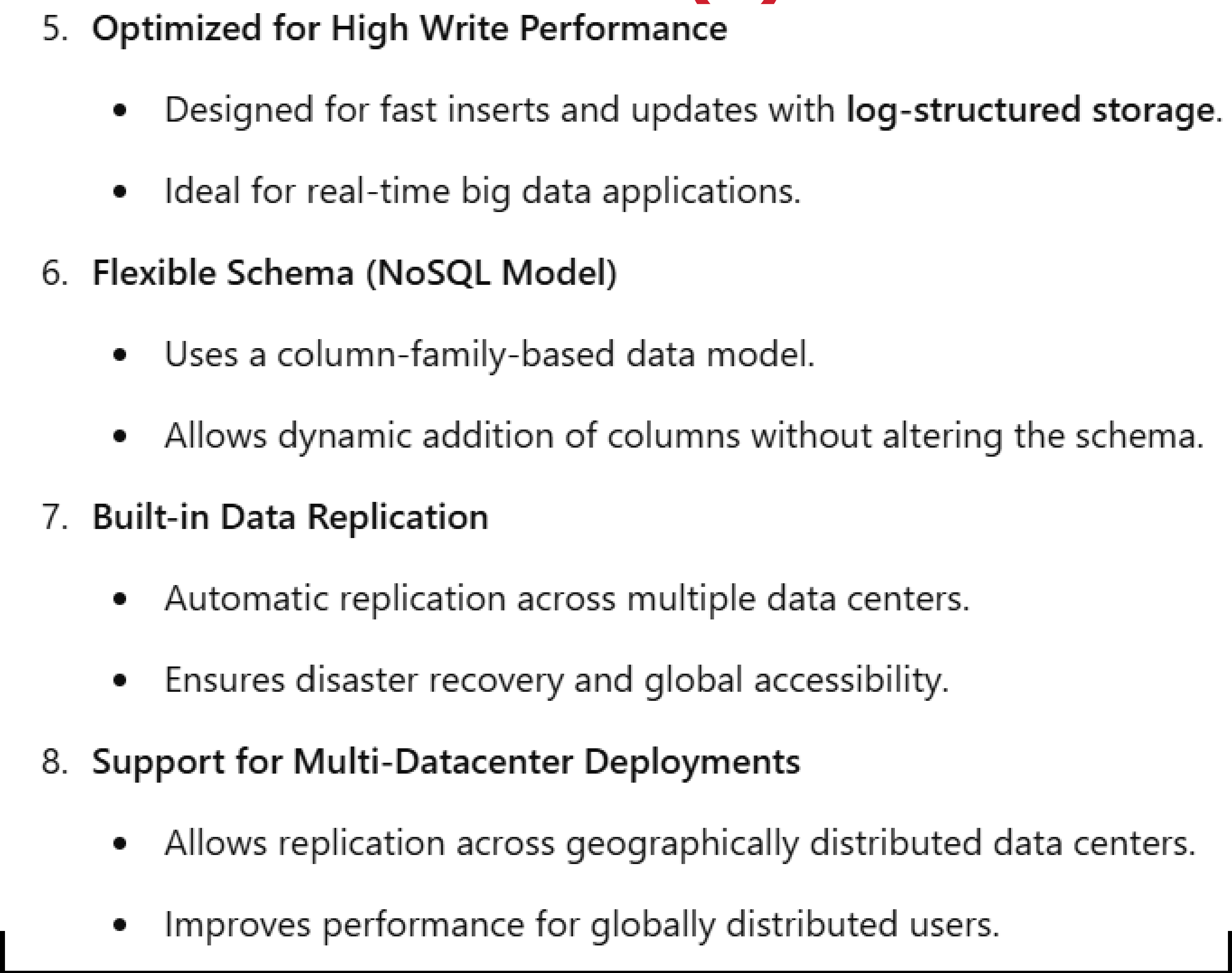
- **Used By:** Netflix, Twitter, eBay, Uber, and many more for high-performance, globally distributed applications.

Key Features of Cassandra (1)



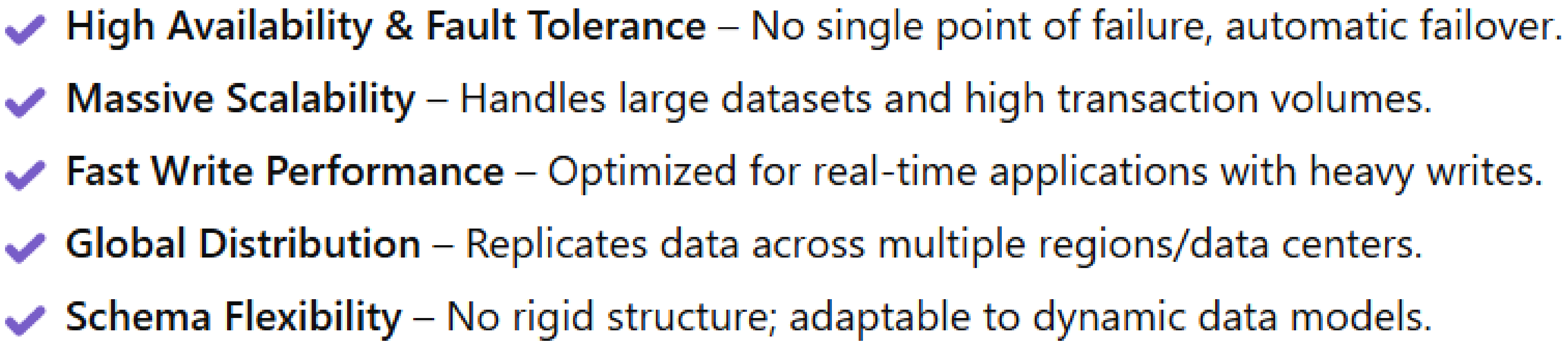
1. **Distributed & Decentralized Architecture**
   - No master-slave structure; all nodes are equal (peer-to-peer model).
   - Ensures high availability and fault tolerance.
2. **Horizontal Scalability**
   - Easily add more nodes to handle increasing data loads.
   - Supports massive datasets with linear scalability.
3. **High Availability & Fault Tolerance**
   - Data is replicated across multiple nodes.
   - If a node fails, another node takes over without downtime.
4. **Tunable Consistency**
   - Supports different levels of consistency (`ONE`, `QUORUM`, `ALL`).
   - Balances availability vs. consistency based on application needs.

Key Features of Cassandra (2)



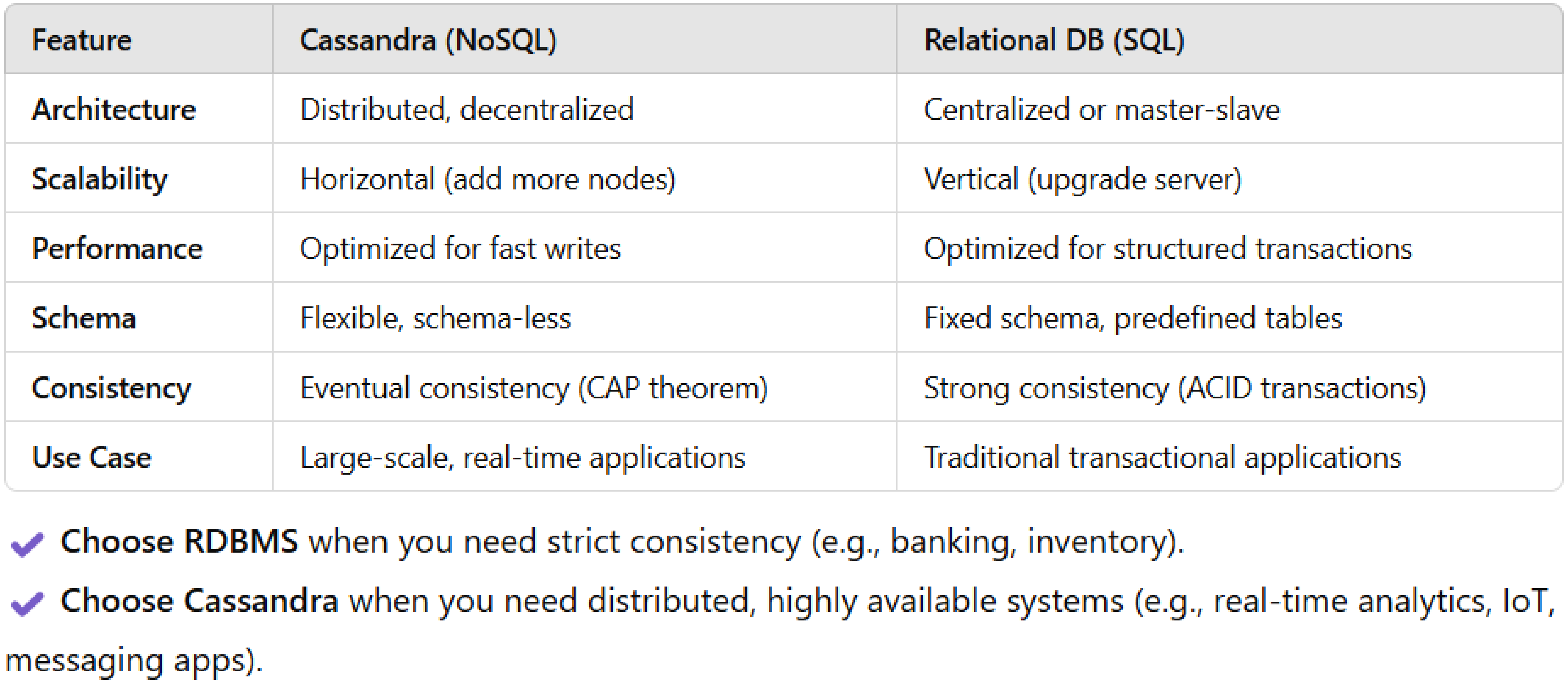
5. **Optimized for High Write Performance**
   - Designed for fast inserts and updates with **log-structured storage**.
   - Ideal for real-time big data applications.
6. **Flexible Schema (NoSQL Model)**
   - Uses a column-family-based data model.
   - Allows dynamic addition of columns without altering the schema.
7. **Built-in Data Replication**
   - Automatic replication across multiple data centers.
   - Ensures disaster recovery and global accessibility.
8. **Support for Multi-Datacenter Deployments**
   - Allows replication across geographically distributed data centers.
   - Improves performance for globally distributed users.

When to use Cassandra?



- ✓ **High Availability & Fault Tolerance** – No single point of failure, automatic failover.
- ✓ **Massive Scalability** – Handles large datasets and high transaction volumes.
- ✓ **Fast Write Performance** – Optimized for real-time applications with heavy writes.
- ✓ **Global Distribution** – Replicates data across multiple regions/data centers.
- ✓ **Schema Flexibility** – No rigid structure; adaptable to dynamic data models.

Cassandra vs. Relational Database (RDBMS)

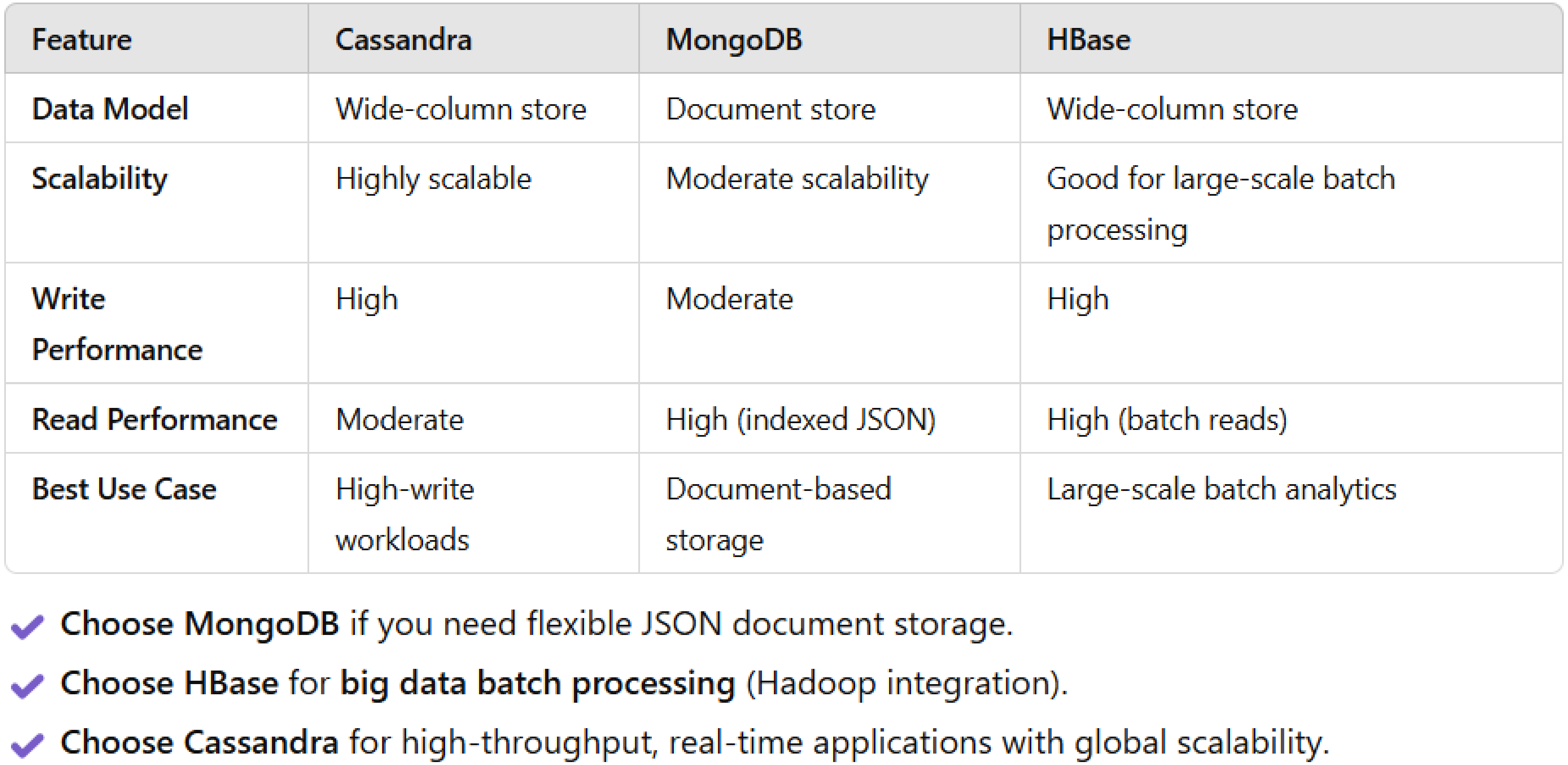


Feature	Cassandra (NoSQL)	Relational DB (SQL)
Architecture	Distributed, decentralized	Centralized or master-slave
Scalability	Horizontal (add more nodes)	Vertical (upgrade server)
Performance	Optimized for fast writes	Optimized for structured transactions
Schema	Flexible, schema-less	Fixed schema, predefined tables
Consistency	Eventual consistency (CAP theorem)	Strong consistency (ACID transactions)
Use Case	Large-scale, real-time applications	Traditional transactional applications

✓ **Choose RDBMS** when you need strict consistency (e.g., banking, inventory).

✓ **Choose Cassandra** when you need distributed, highly available systems (e.g., real-time analytics, IoT, messaging apps).

Cassandra vs. NoSQL Database



Feature	Cassandra	MongoDB	HBase
Data Model	Wide-column store	Document store	Wide-column store
Scalability	Highly scalable	Moderate scalability	Good for large-scale batch processing
Write Performance	High	Moderate	High
Read Performance	Moderate	High (indexed JSON)	High (batch reads)
Best Use Case	High-write workloads	Document-based storage	Large-scale batch analytics

✓ **Choose MongoDB** if you need flexible JSON document storage.

✓ **Choose HBase** for **big data batch processing** (Hadoop integration).

✓ **Choose Cassandra** for high-throughput, real-time applications with global scalability.

Architecture Overview: Nodes, Partitions, Replication

🌍 Cassandra's Distributed Architecture

✅ **Peer-to-Peer Model** – No master-slave setup; all nodes are equal.

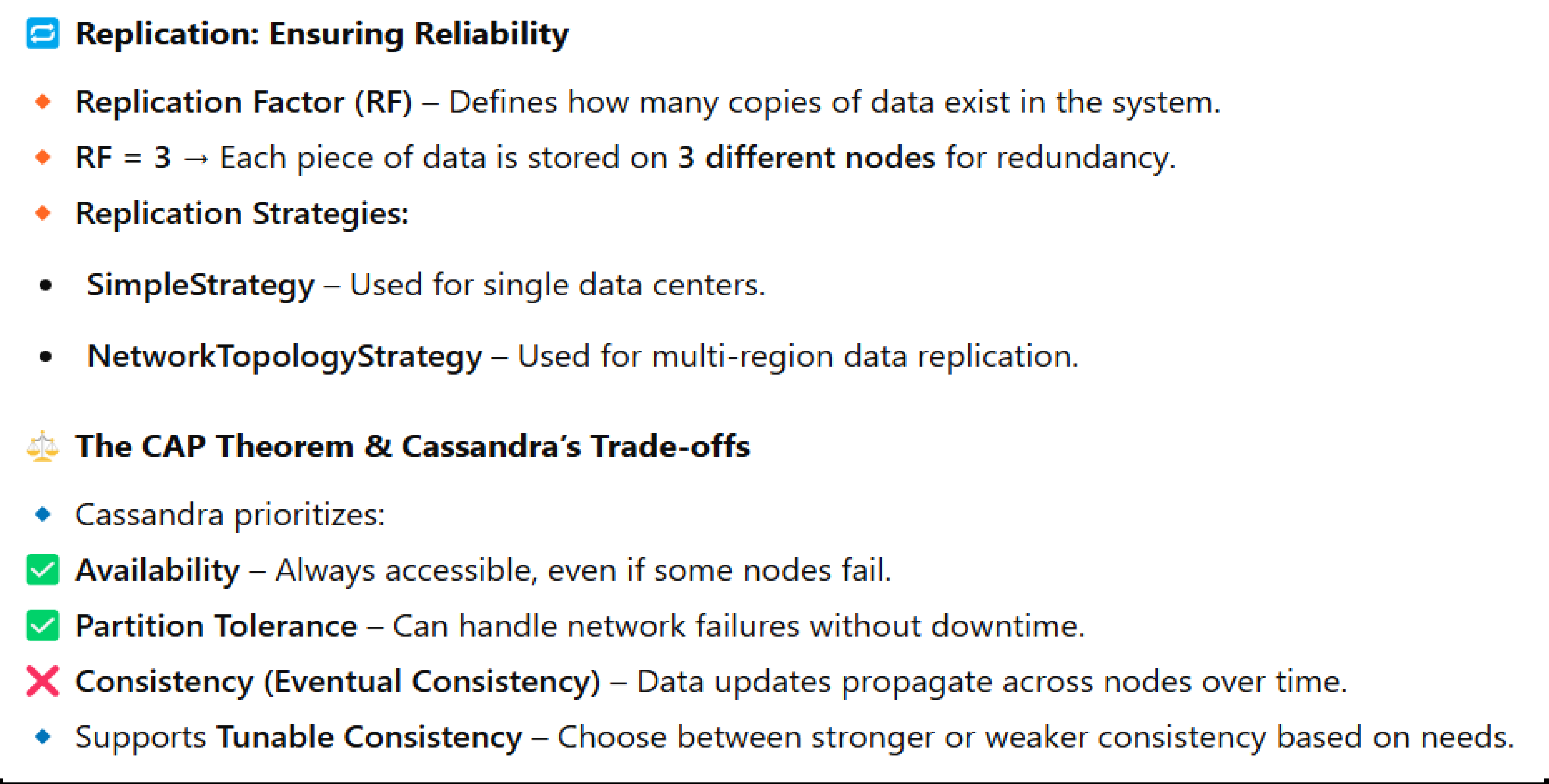
✅ **High Availability & Fault Tolerance** – Data is automatically distributed across nodes.

✅ **Ring-Based Architecture** – Nodes are logically arranged in a circle to balance load.

🖥️ Nodes & Partitions: How Cassandra Stores Data

- 🔹 **Node** – A single machine in the Cassandra cluster.
- 🔹 **Partitioning** – Data is divided into **partitions** using a **Partition Key**, ensuring even distribution.
- 🔹 **Data Centers & Clusters** – Nodes are grouped into **data centers** for geographical redundancy.

Architecture Overview: Nodes, Partitions, Replication



🔁 Replication: Ensuring Reliability

🔸 **Replication Factor (RF)** – Defines how many copies of data exist in the system.

🔸 **RF = 3** → Each piece of data is stored on **3 different nodes** for redundancy.

🔸 **Replication Strategies:**

- **SimpleStrategy** – Used for single data centers.
- **NetworkTopologyStrategy** – Used for multi-region data replication.

⚖️ The CAP Theorem & Cassandra's Trade-offs

🔹 Cassandra prioritizes:

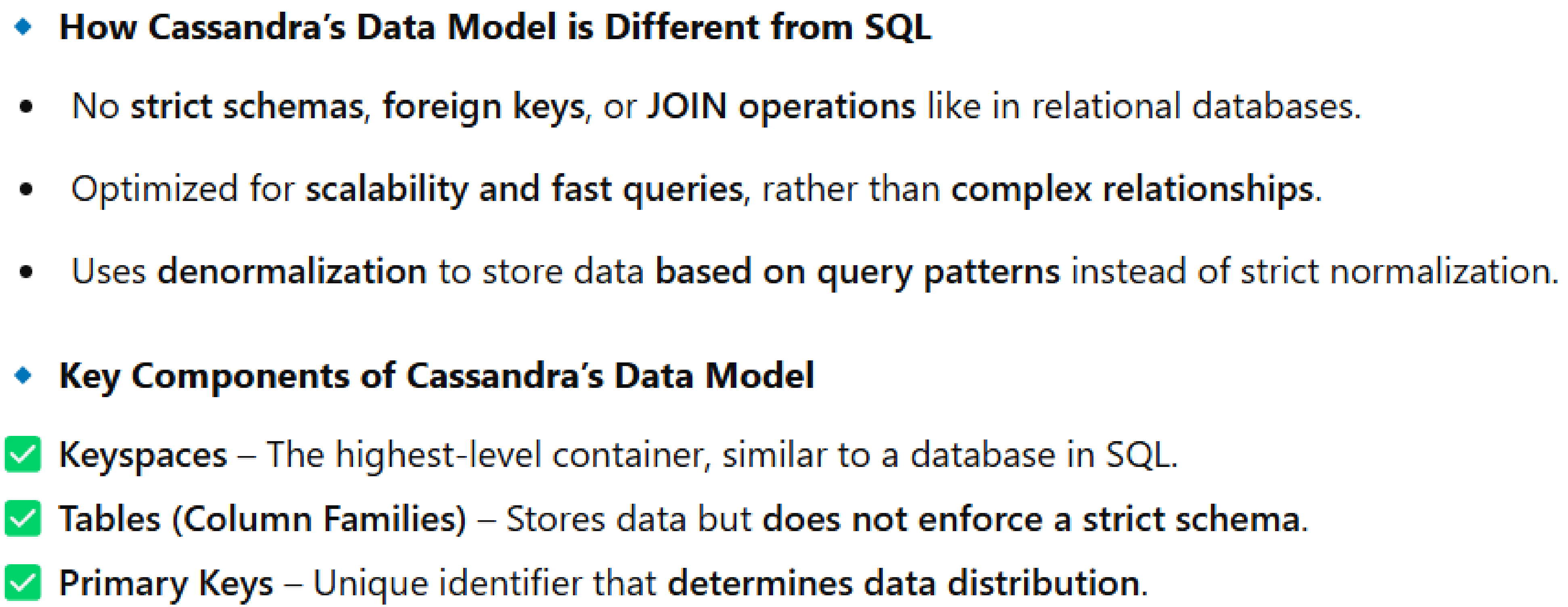
✅ **Availability** – Always accessible, even if some nodes fail.

✅ **Partition Tolerance** – Can handle network failures without downtime.

❌ **Consistency (Eventual Consistency)** – Data updates propagate across nodes over time.

🔹 Supports **Tunable Consistency** – Choose between stronger or weaker consistency based on needs.

Cassandra’s Data Model



- **How Cassandra’s Data Model is Different from SQL**
  - No **strict schemas**, **foreign keys**, or **JOIN operations** like in relational databases.
  - Optimized for **scalability and fast queries**, rather than **complex relationships**.
  - Uses **denormalization** to store data **based on query patterns** instead of strict normalization.
- **Key Components of Cassandra’s Data Model**

✅ **Keyspaces** – The highest-level container, similar to a database in SQL.

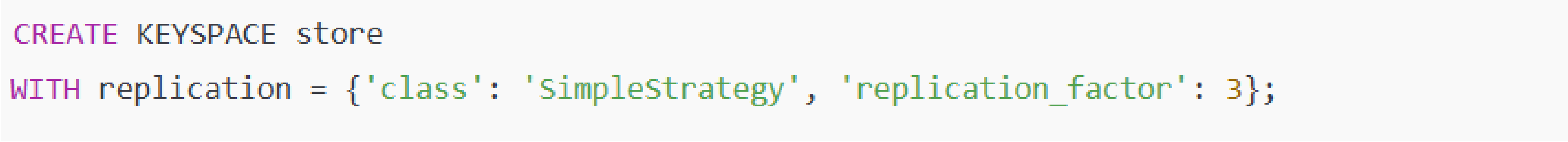
✅ **Tables (Column Families)** – Stores data but **does not enforce a strict schema**.

✅ **Primary Keys** – Unique identifier that **determines data distribution**.

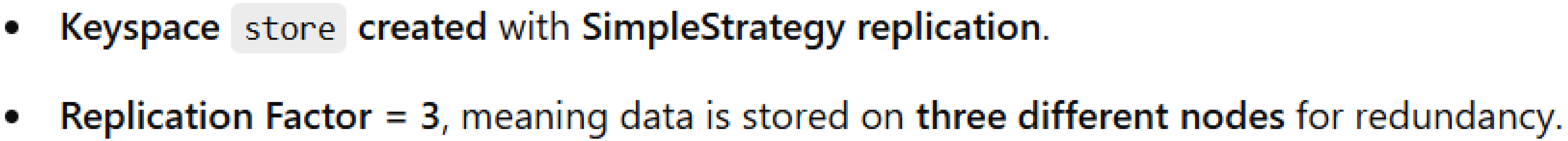
Key Features of Cassandra

- **Partition Keys: Data Distribution in Cassandra**
  - The **Partition Key** decides **which node stores the data**.
  - Helps distribute data **evenly across the cluster**.
  - Ensures **fast and scalable queries** by keeping related data together.
- **Clustering Columns: Organizing Data Within Partitions**
  - Defines **how data is sorted inside a partition**.
  - Enables efficient **ordering and filtering** within a partition.
- **Denormalization: Why Cassandra Avoids Joins**
  - Data is **duplicated across tables** to **avoid costly JOIN operations**.
  - Ensures **faster queries** by storing **pre-aggregated** data.

Creating a Keyspace

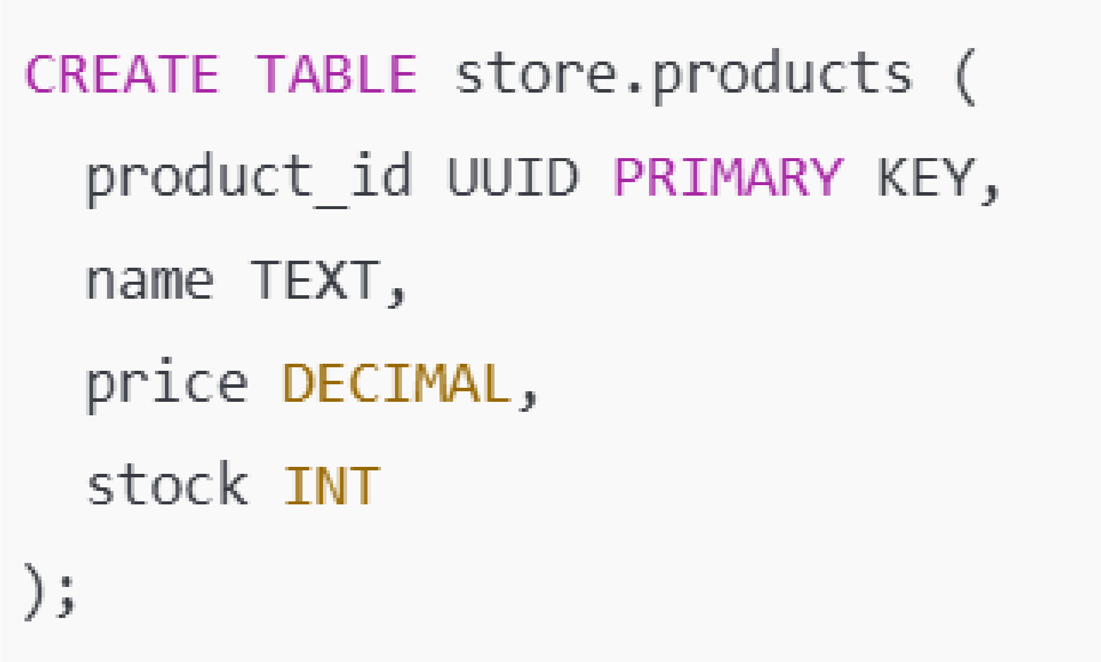


```
CREATE KEYSPACE store
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
```



- **Keyspace** `store` **created** with **SimpleStrategy replication**.
- **Replication Factor = 3**, meaning data is stored on **three different nodes** for redundancy.

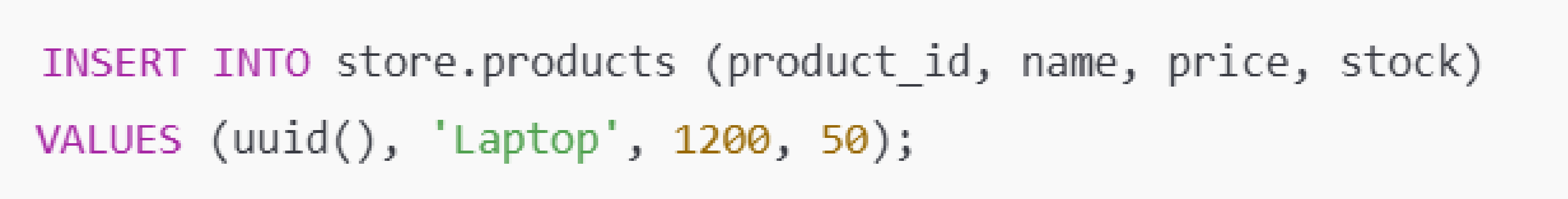
Creating a Table with Primary and Clustering Keys



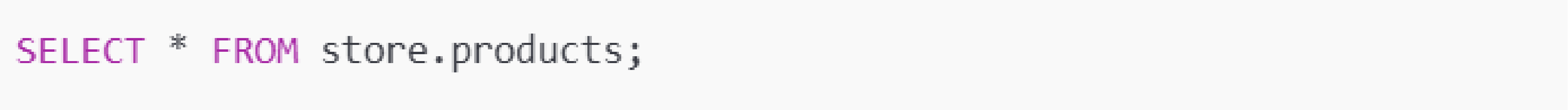
```
CREATE TABLE store.products (
  product_id UUID PRIMARY KEY,
  name TEXT,
  price DECIMAL,
  stock INT
);
```

- *product_id* is the Partition Key, which decides which node stores the data.
- The table is schema-flexible, allowing new columns to be added dynamically.

Inserting and Querying Data

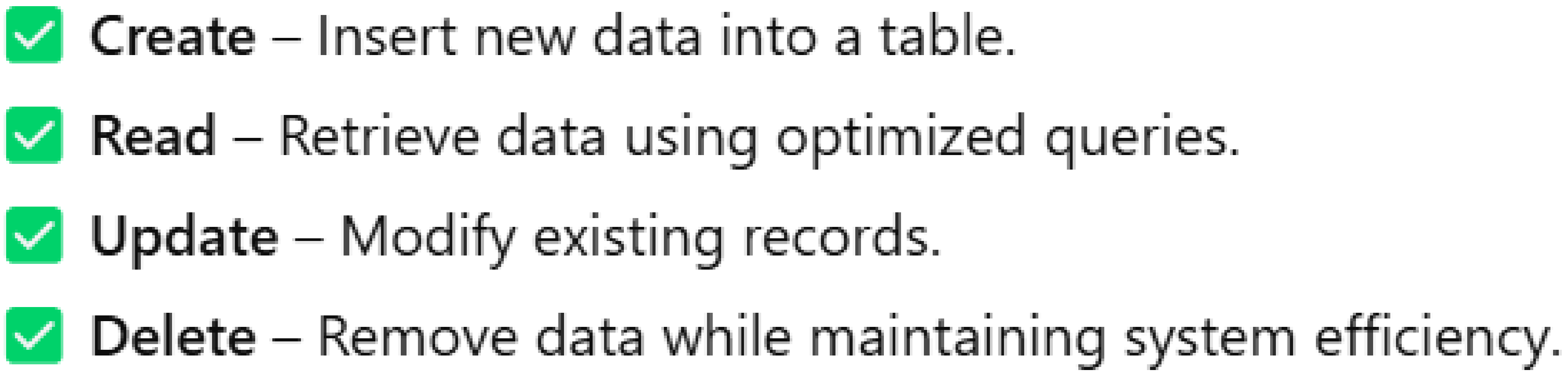


```
INSERT INTO store.products (product_id, name, price, stock)
VALUES (uuid(), 'Laptop', 1200, 50);
```



```
SELECT * FROM store.products;
```

CRUD Operations in Cassandra



✅ **Create** – Insert new data into a table.

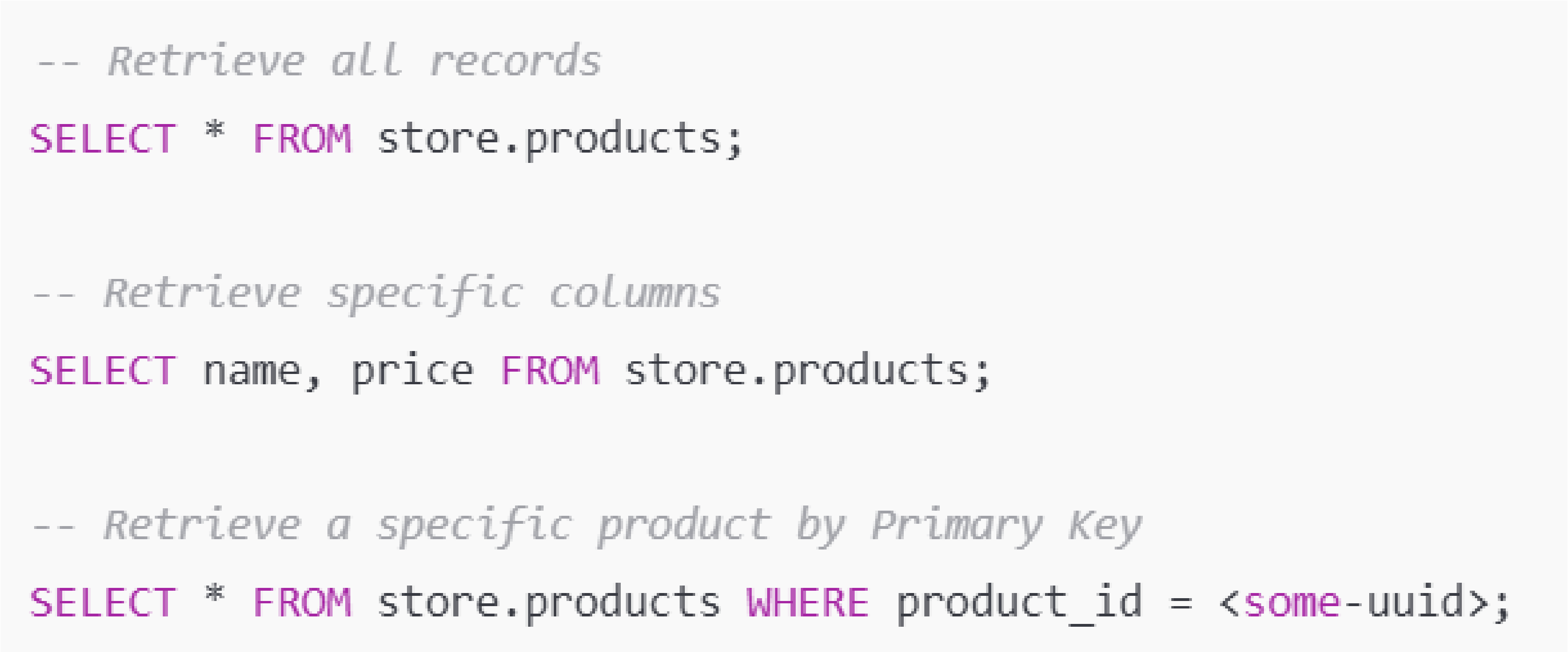
✅ **Read** – Retrieve data using optimized queries.

✅ **Update** – Modify existing records.

✅ **Delete** – Remove data while maintaining system efficiency.

Query Data with CQL

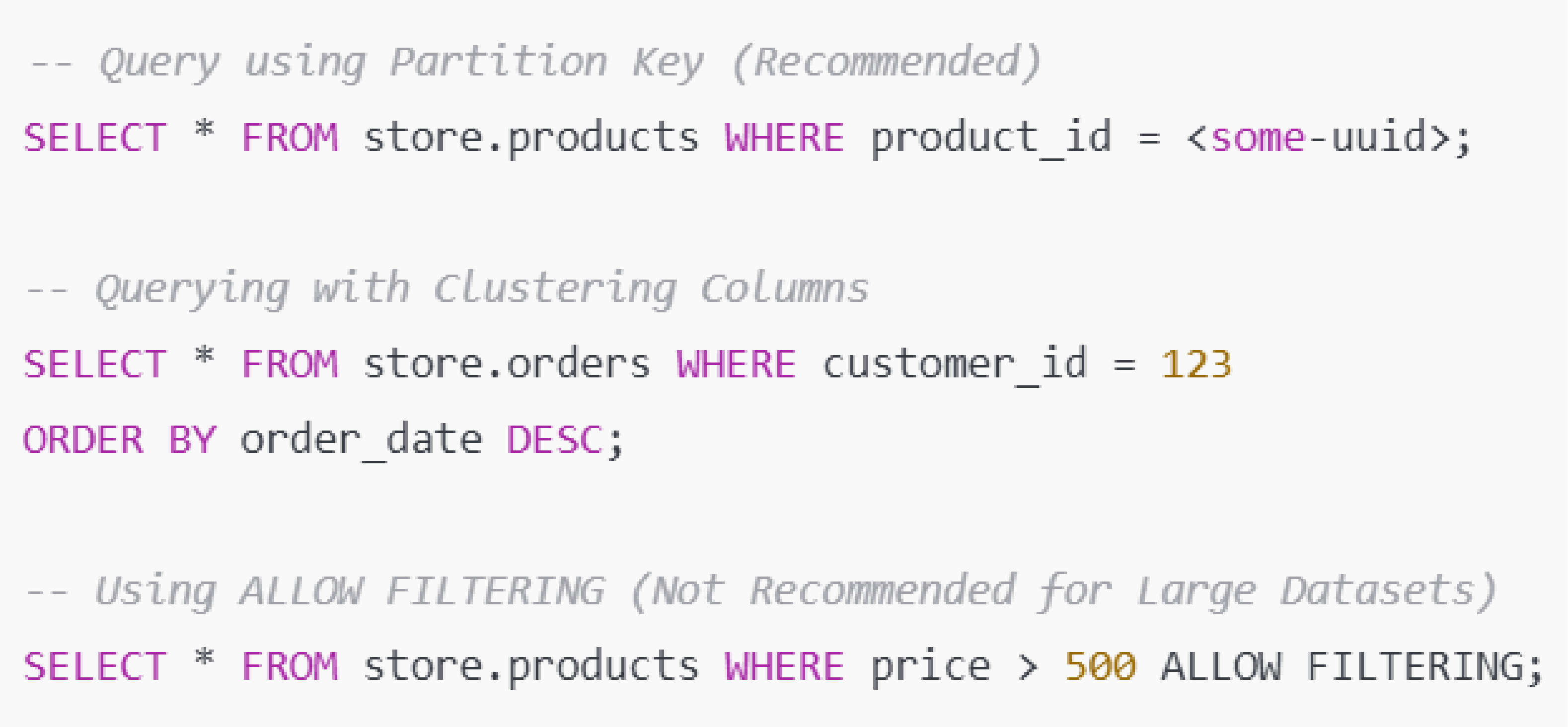
- **Basic Querying with SELECT:**



```
-- Retrieve all records
SELECT * FROM store.products;

-- Retrieve specific columns
SELECT name, price FROM store.products;

-- Retrieve a specific product by Primary Key
SELECT * FROM store.products WHERE product_id = <some-uuid>;
```

Query Data with CQL

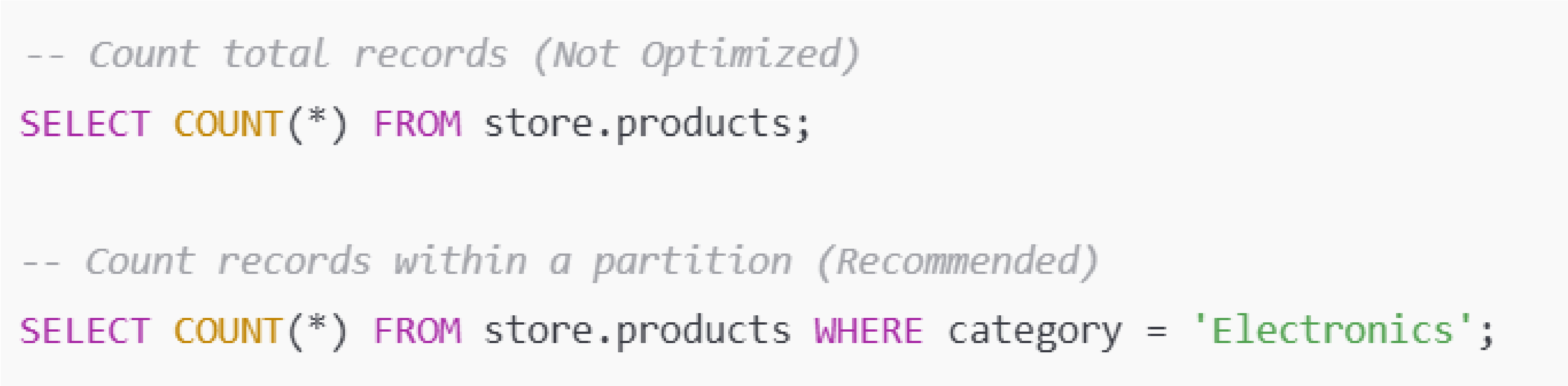
- **Filtering Data:**



```
-- Query using Partition Key (Recommended)
SELECT * FROM store.products WHERE product_id = <some-uuid>;

-- Querying with Clustering Columns
SELECT * FROM store.orders WHERE customer_id = 123
ORDER BY order_date DESC;

-- Using ALLOW FILTERING (Not Recommended for Large Datasets)
SELECT * FROM store.products WHERE price > 500 ALLOW FILTERING;
```

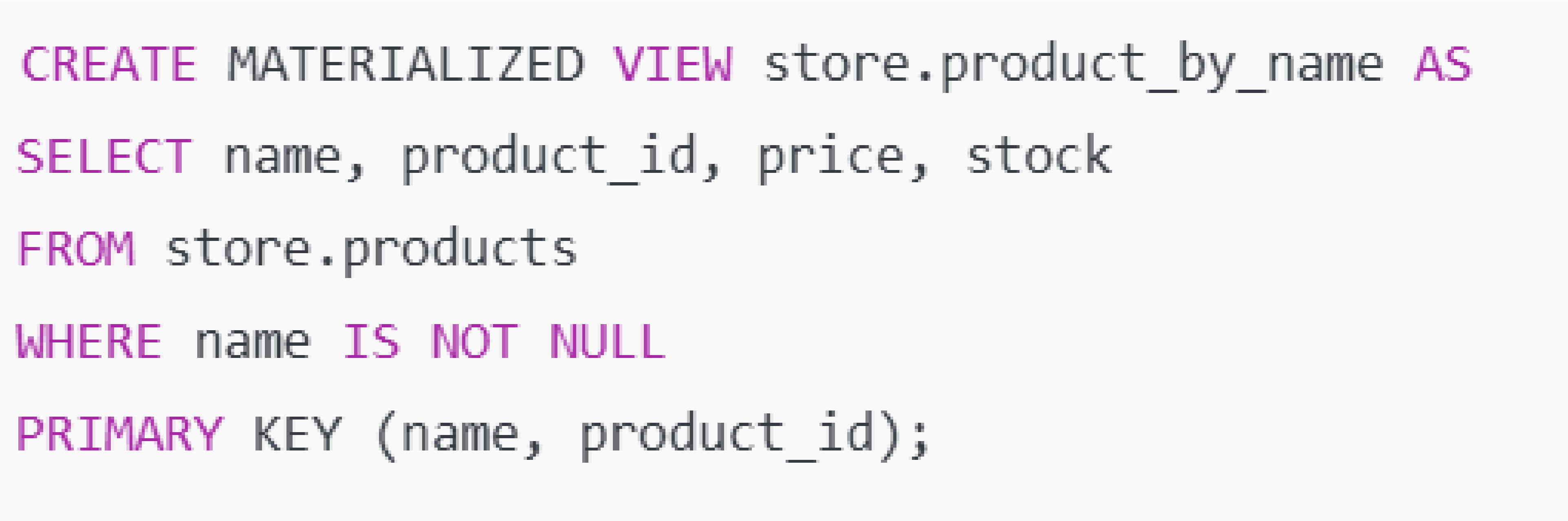
Query Data with CQL

- **Aggregation & Counting Records:**



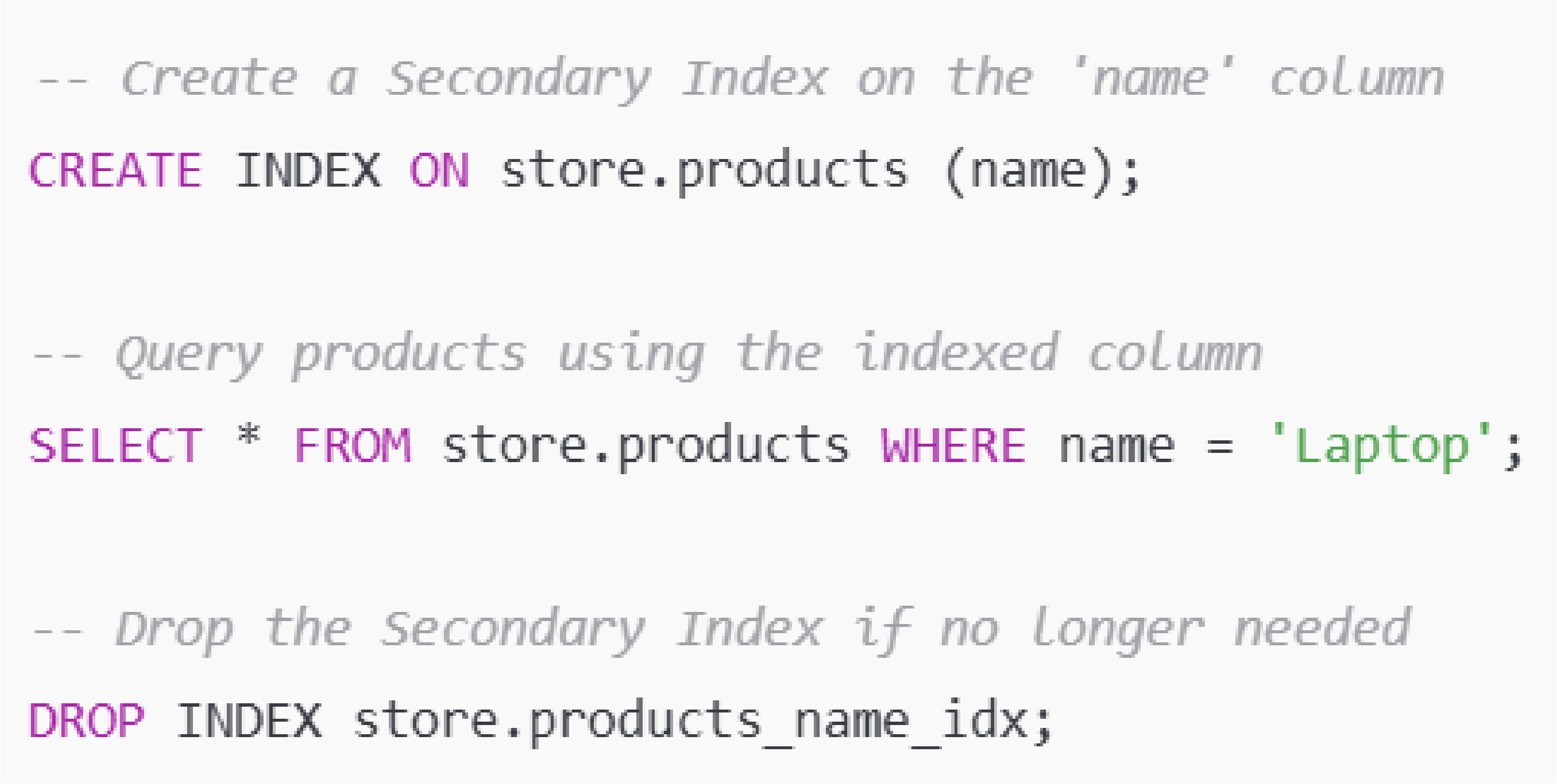
```
-- Count total records (Not Optimized)
SELECT COUNT(*) FROM store.products;

-- Count records within a partition (Recommended)
SELECT COUNT(*) FROM store.products WHERE category = 'Electronics';
```

Materialized View



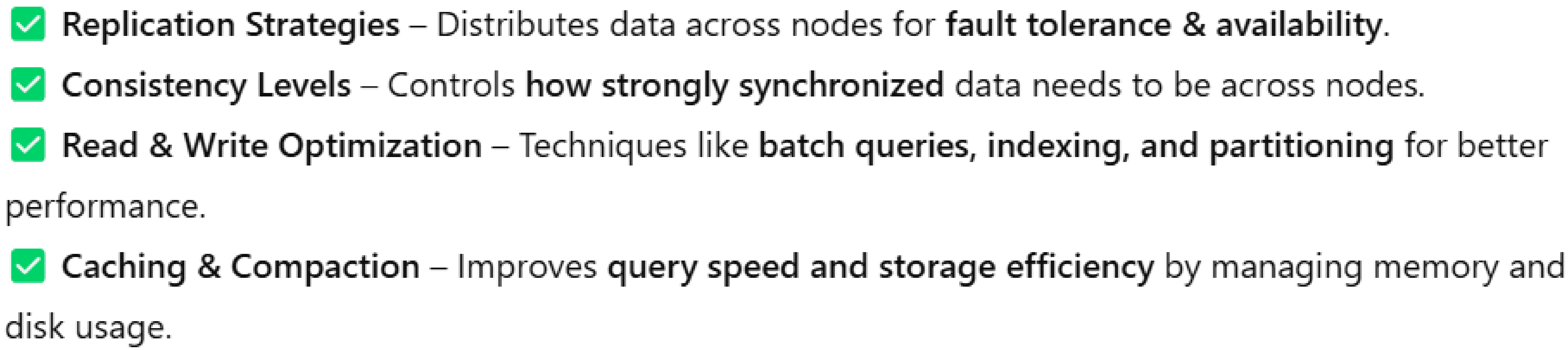
```
CREATE MATERIALIZED VIEW store.product_by_name AS
SELECT name, product_id, price, stock
FROM store.products
WHERE name IS NOT NULL
PRIMARY KEY (name, product_id);
```

Secondary Index



```sql
-- Create a Secondary Index on the 'name' column
CREATE INDEX ON store.products (name);

-- Query products using the indexed column
SELECT * FROM store.products WHERE name = 'Laptop';

-- Drop the Secondary Index if no longer needed
DROP INDEX store.products_name_idx;
```

Performance Optimization & Data Replication



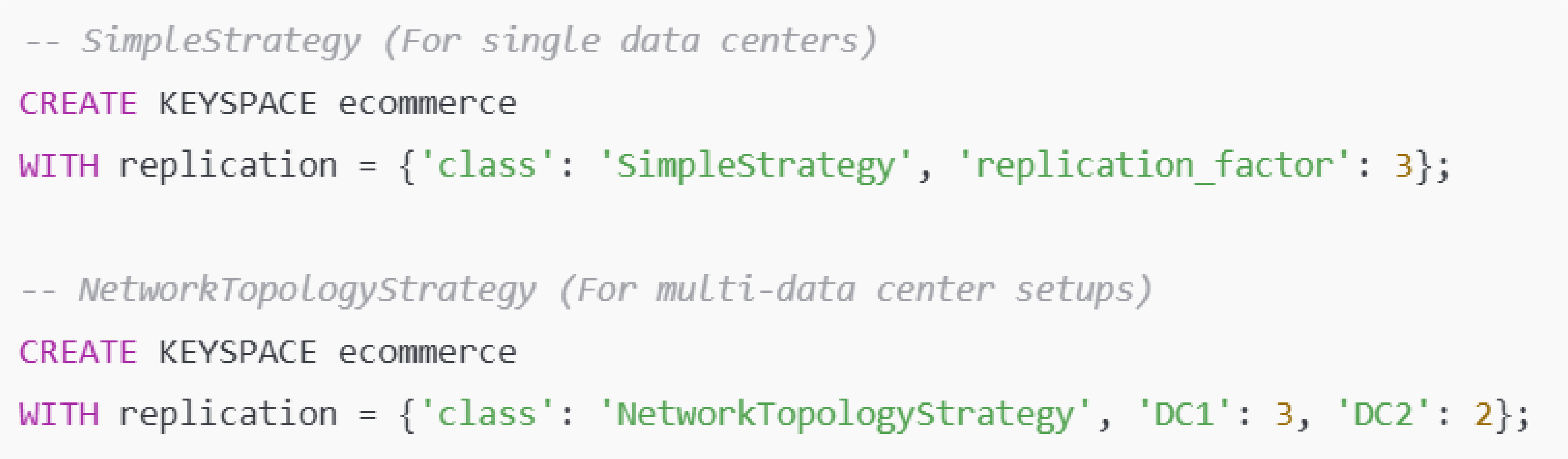
✅ **Replication Strategies** – Distributes data across nodes for **fault tolerance & availability**.

✅ **Consistency Levels** – Controls **how strongly synchronized** data needs to be across nodes.

✅ **Read & Write Optimization** – Techniques like **batch queries, indexing, and partitioning** for better performance.

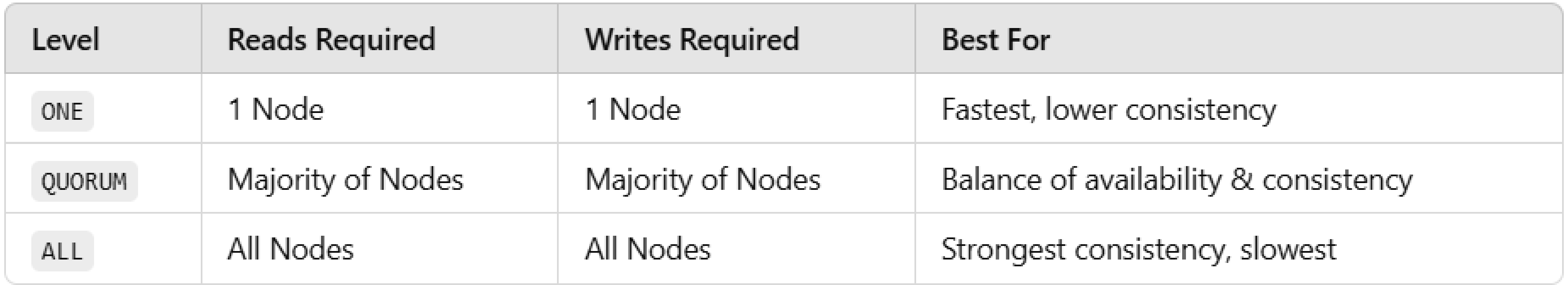
✅ **Caching & Compaction** – Improves **query speed and storage efficiency** by managing memory and disk usage.

Replication Strategies in Cassandra

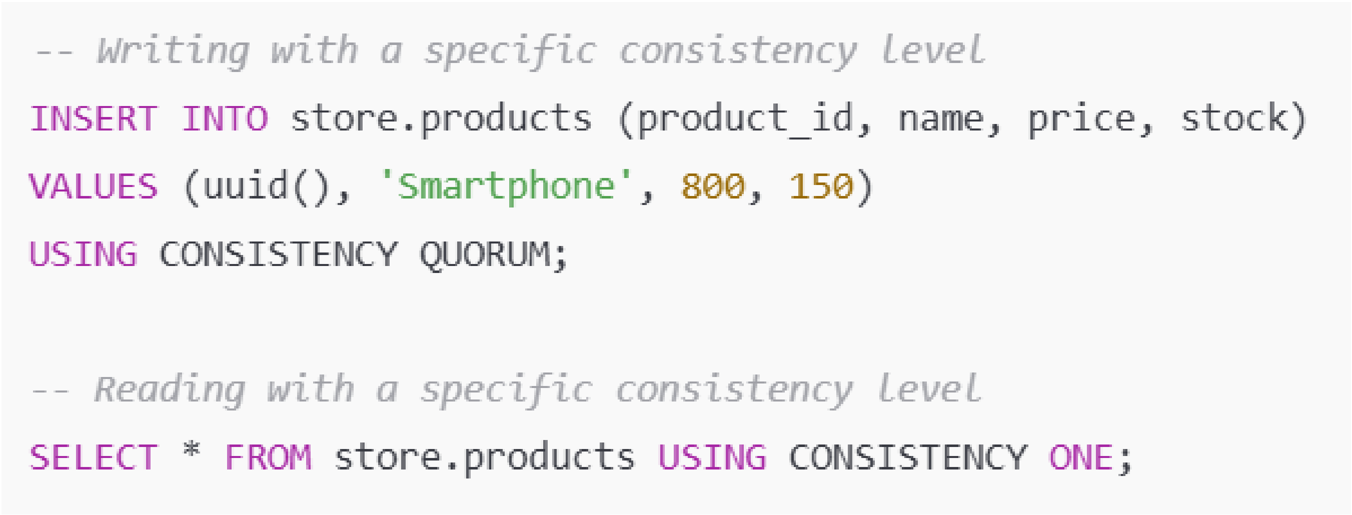


```
-- SimpleStrategy (For single data centers)
CREATE KEYSPACE ecommerce
WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};

-- NetworkTopologyStrategy (For multi-data center setups)
CREATE KEYSPACE ecommerce
WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': 3, 'DC2': 2};
```

Understanding Consistency Levels

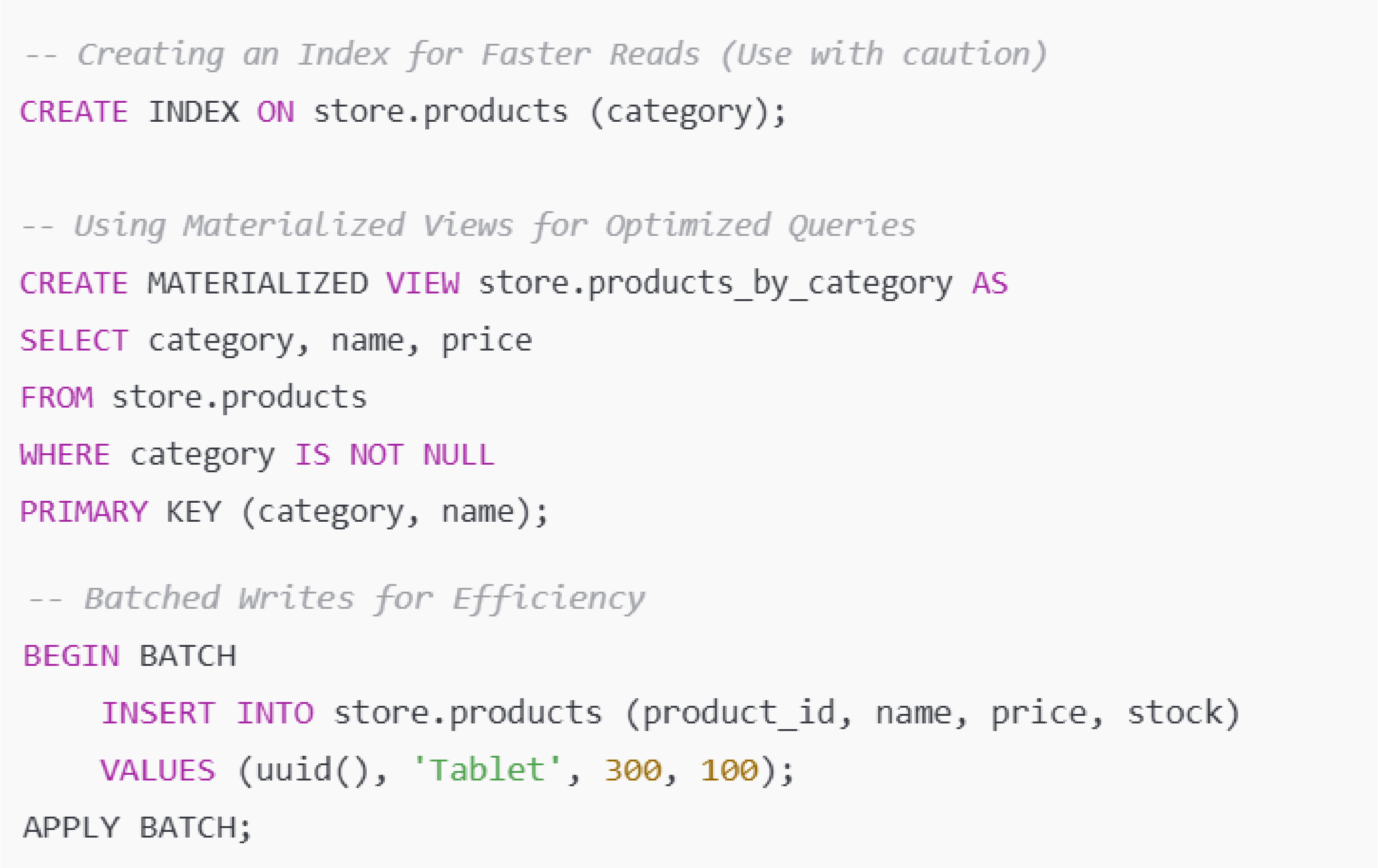


Level	Reads Required	Writes Required	Best For
`ONE`	1 Node	1 Node	Fastest, lower consistency
`QUORUM`	Majority of Nodes	Majority of Nodes	Balance of availability & consistency
`ALL`	All Nodes	All Nodes	Strongest consistency, slowest



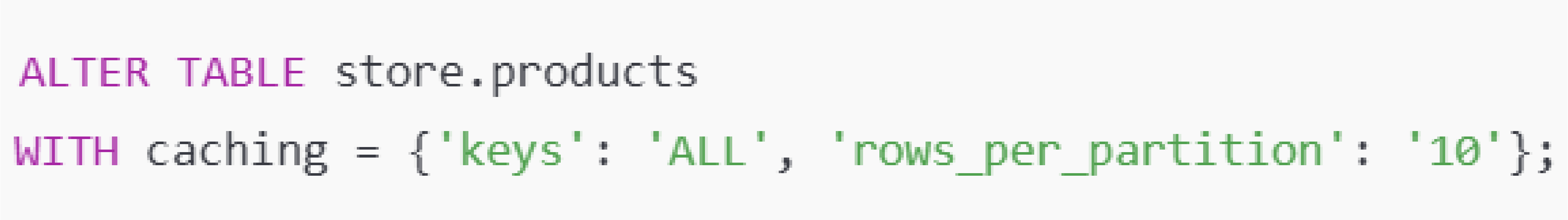
```
-- Writing with a specific consistency level
INSERT INTO store.products (product_id, name, price, stock)
VALUES (uuid(), 'Smartphone', 800, 150)
USING CONSISTENCY QUORUM;

-- Reading with a specific consistency level
SELECT * FROM store.products USING CONSISTENCY ONE;
```

Tuning Read & Write Performance

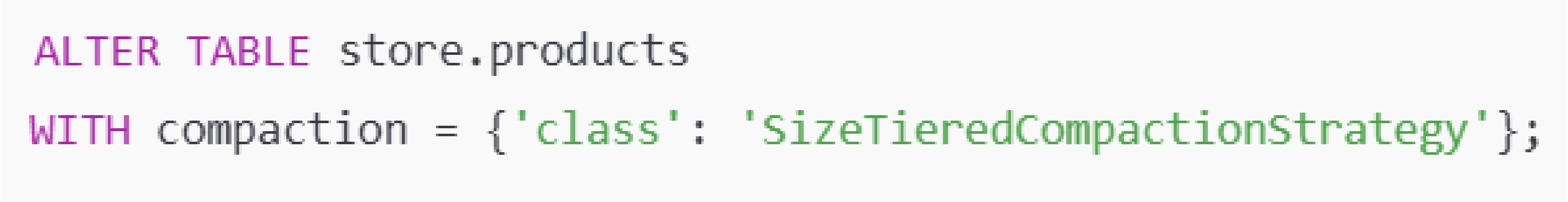


```
-- Creating an Index for Faster Reads (Use with caution)
CREATE INDEX ON store.products (category);


-- Using Materialized Views for Optimized Queries
CREATE MATERIALIZED VIEW store.products_by_category AS
SELECT category, name, price
FROM store.products
WHERE category IS NOT NULL
PRIMARY KEY (category, name);

-- Batched Writes for Efficiency
BEGIN BATCH
    INSERT INTO store.products (product_id, name, price, stock)
    VALUES (uuid(), 'Tablet', 300, 100);
APPLY BATCH;
```

Using Caching & Compaction Strategies

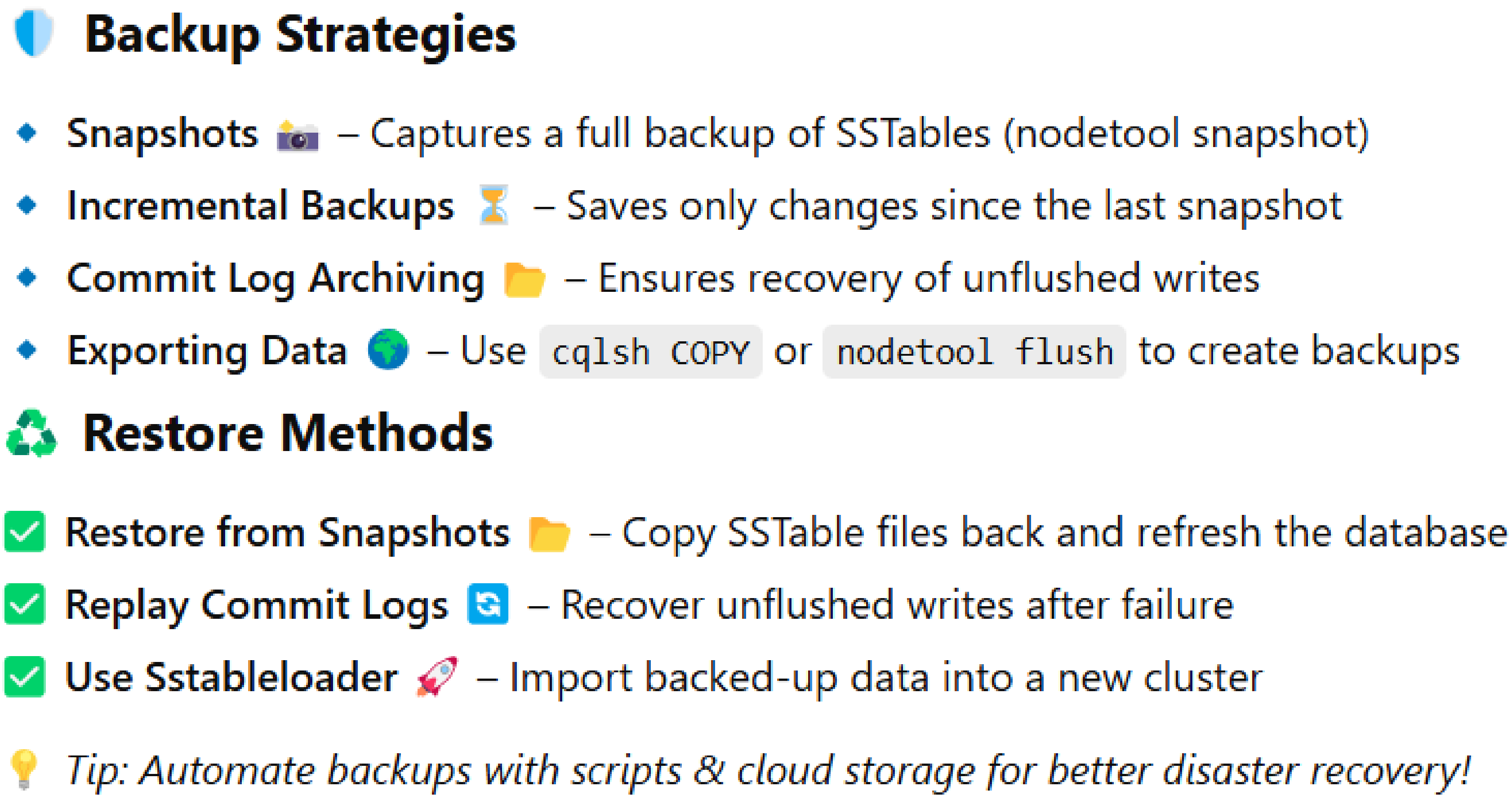


```
ALTER TABLE store.products
WITH caching = {'keys': 'ALL', 'rows_per_partition': '10'};
```



```
 ALTER TABLE store.products
WITH compaction = {'class': 'SizeTieredCompactionStrategy'};
```

Backup and Restore in Cassandra



🛡️ Backup Strategies

- 🔹 **Snapshots** 📷 – Captures a full backup of SSTables (nodetool snapshot)
- 🔹 **Incremental Backups** ⏳ – Saves only changes since the last snapshot
- 🔹 **Commit Log Archiving** 📁 – Ensures recovery of unflushed writes
- 🔹 **Exporting Data** 🌍 – Use `cqlsh COPY` or `nodetool flush` to create backups

♻️ Restore Methods

✅ **Restore from Snapshots** 📂 – Copy SSTable files back and refresh the database

✅ **Replay Commit Logs** 🔄 – Recover unflushed writes after failure

✅ **Use Sstableloader** 🚀 – Import backed-up data into a new cluster

💡 *Tip: Automate backups with scripts & cloud storage for better disaster recovery!*

Security Features in Cassandra

🔐 Authentication & Authorization

- 🔹 User Authentication 👤 – Verify identities before accessing Cassandra
- 🔹 Role-Based Access Control (RBAC) 🎭 – Assign roles & permissions
- 🔹 LDAP & Kerberos Support 🔑 – Integrate with enterprise security

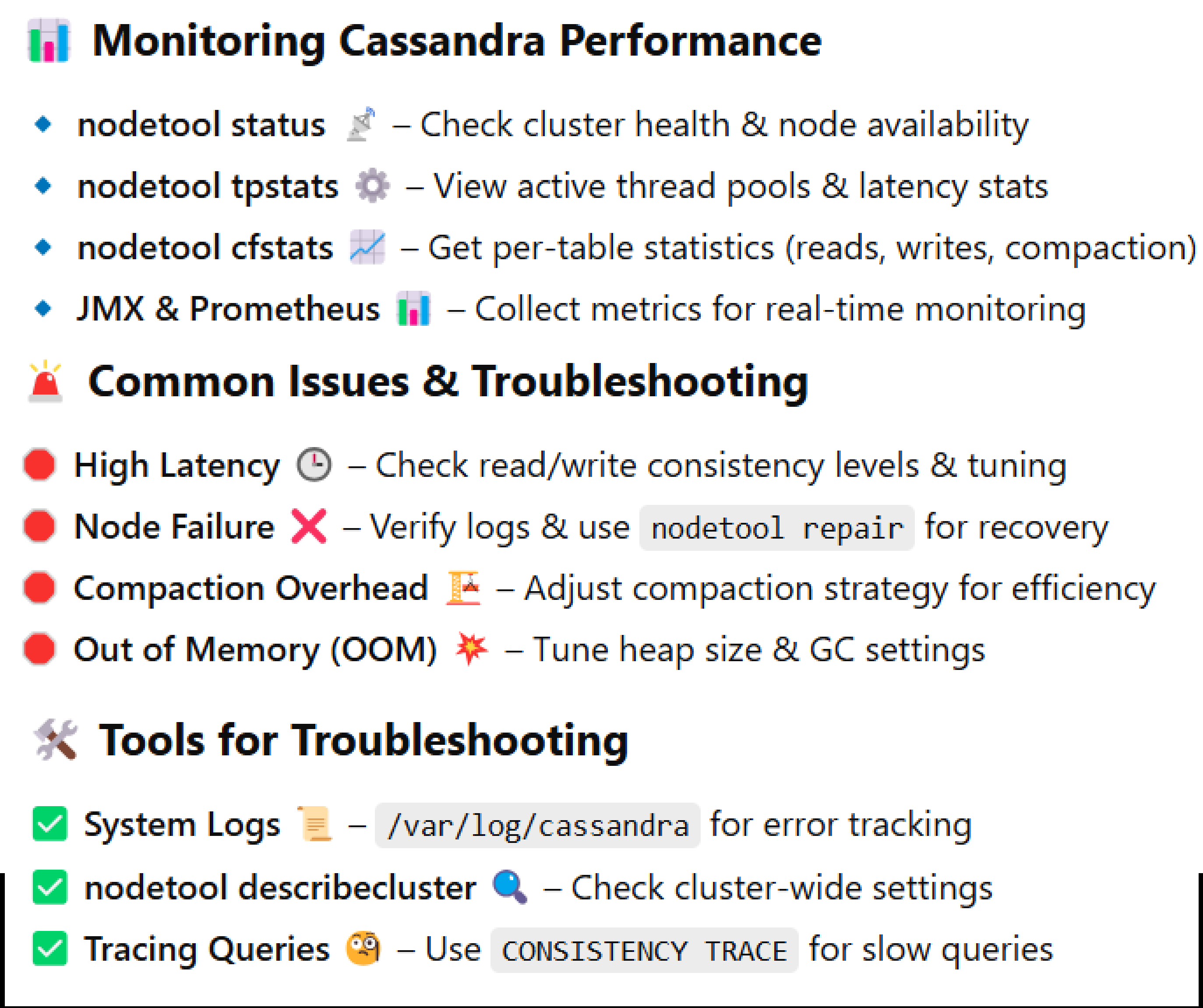
🛡️ Data Encryption

🔒 Client-to-Node Encryption 🔄 – Protects data in transit

🔒 Node-to-Node Encryption 🔗 – Secures inter-node communication

🔒 At-Rest Encryption 🗄️ – Safeguards stored data using secure keys

Monitoring and Troubleshooting Cassandra



📊 Monitoring Cassandra Performance

- 🔹 **nodetool status** 📡 – Check cluster health & node availability
- 🔹 **nodetool tpstats** ⚙️ – View active thread pools & latency stats
- 🔹 **nodetool cfstats** 📈 – Get per-table statistics (reads, writes, compaction)
- 🔹 **JMX & Prometheus** 📊 – Collect metrics for real-time monitoring

🚨 Common Issues & Troubleshooting

🔴 **High Latency** 🕒 – Check read/write consistency levels & tuning

🔴 **Node Failure** ❌ – Verify logs & use `nodetool repair` for recovery

🔴 **Compaction Overhead** 🏗️ – Adjust compaction strategy for efficiency

🔴 **Out of Memory (OOM)** 💥 – Tune heap size & GC settings

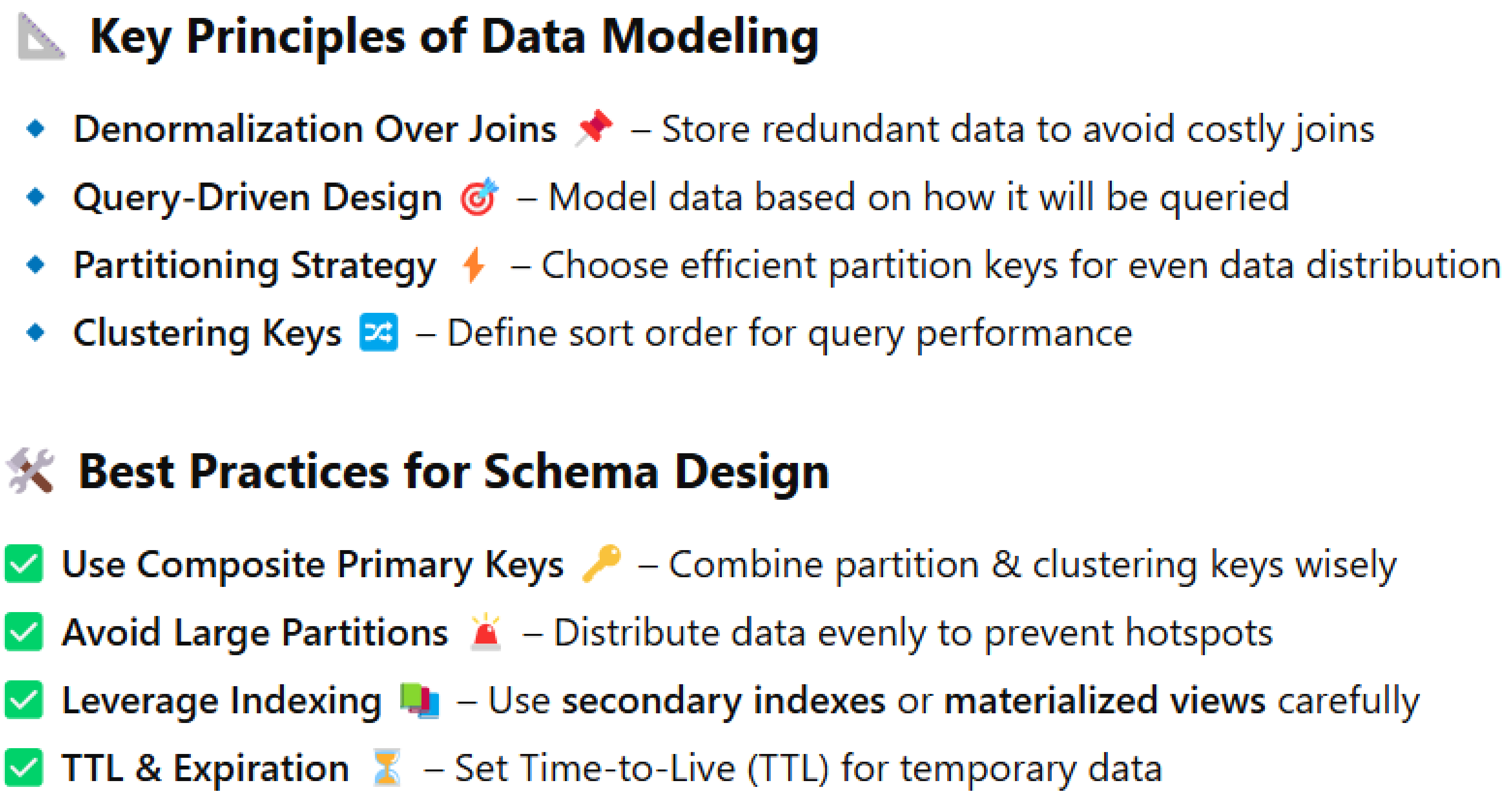
🛠️ Tools for Troubleshooting

✅ **System Logs** 📜 – `/var/log/cassandra` for error tracking

✅ **nodetool describecluster** 🔍 – Check cluster-wide settings

✅ **Tracing Queries** 🧐 – Use `CONSISTENCY TRACE` for slow queries

Advanced Data Modeling in Cassandra



📐 Key Principles of Data Modeling

- 🔹 **Denormalization Over Joins** 📌 – Store redundant data to avoid costly joins
- 🔹 **Query-Driven Design** 🎯 – Model data based on how it will be queried
- 🔹 **Partitioning Strategy** ⚡ – Choose efficient partition keys for even data distribution
- 🔹 **Clustering Keys** 🔀 – Define sort order for query performance

🛠️ Best Practices for Schema Design

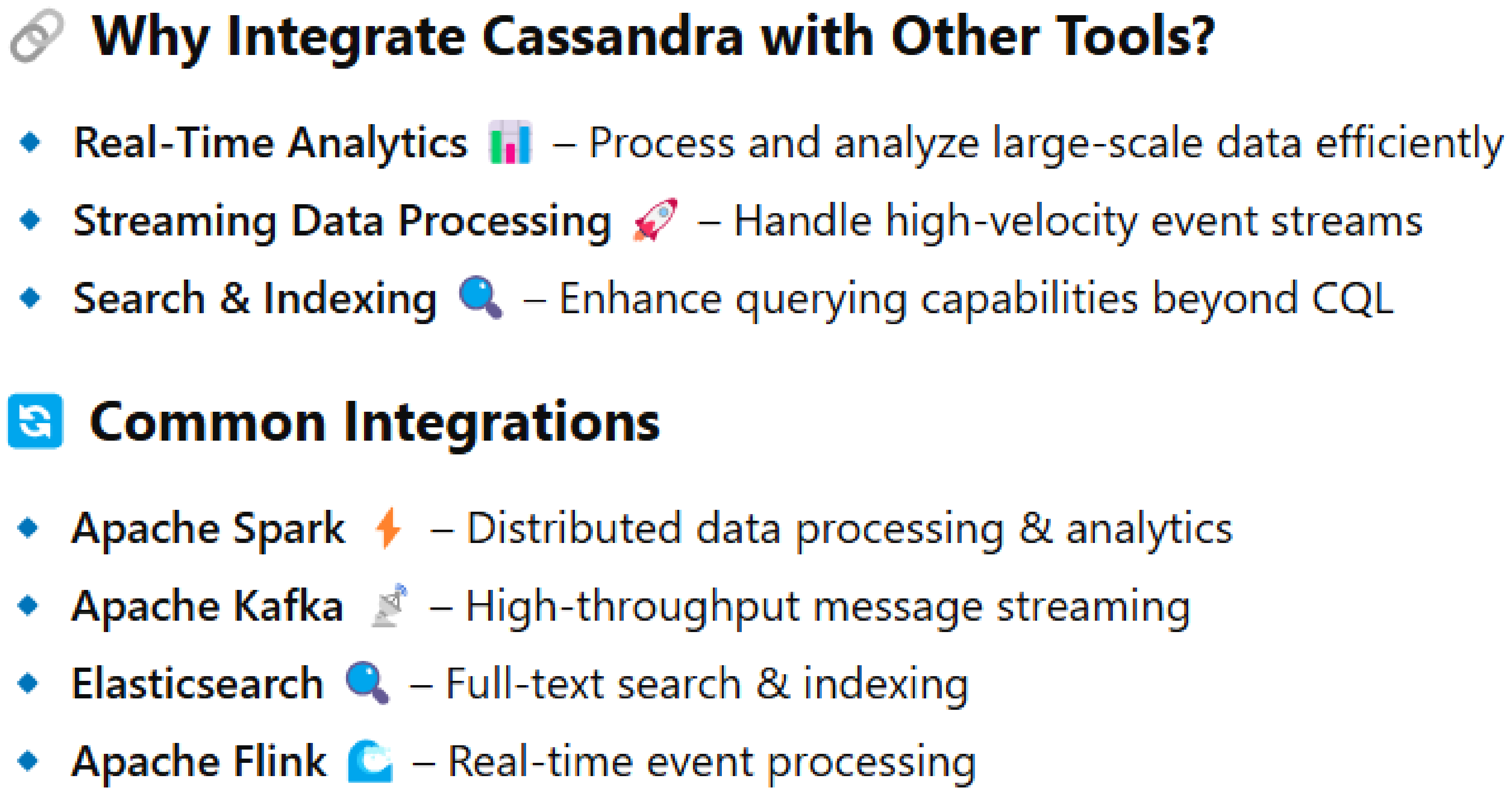
✅ **Use Composite Primary Keys** 🔑 – Combine partition & clustering keys wisely

✅ **Avoid Large Partitions** 🚨 – Distribute data evenly to prevent hotspots

✅ **Leverage Indexing** 📚 – Use **secondary indexes** or **materialized views** carefully

✅ **TTL & Expiration** ⌛ – Set Time-to-Live (TTL) for temporary data

Integration with Other Tools



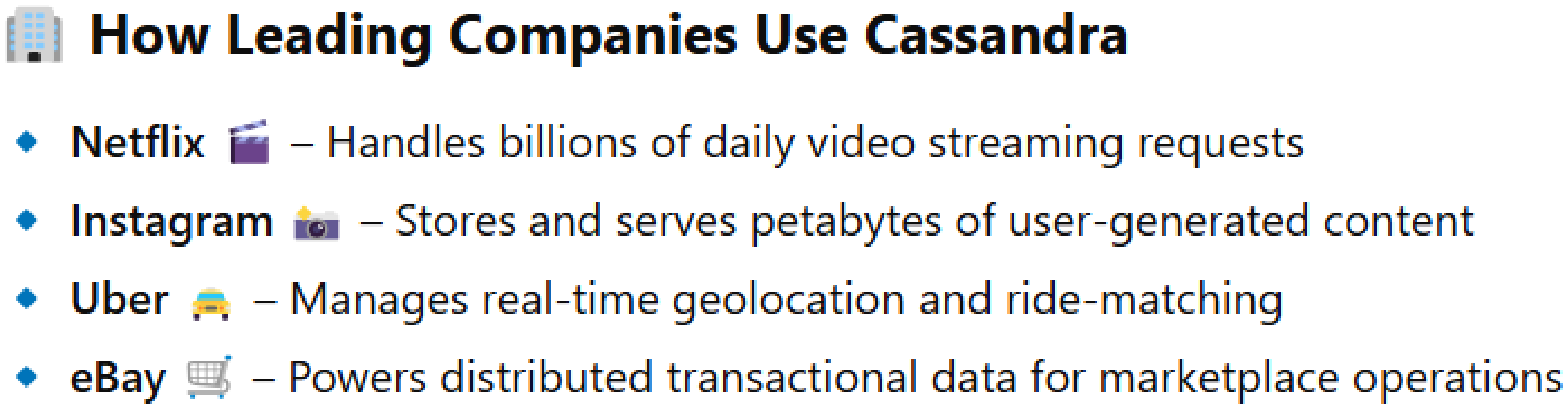
🔗 Why Integrate Cassandra with Other Tools?

- **Real-Time Analytics** 📊 – Process and analyze large-scale data efficiently
- **Streaming Data Processing** 🚀 – Handle high-velocity event streams
- **Search & Indexing** 🔍 – Enhance querying capabilities beyond CQL

🔄 Common Integrations

- **Apache Spark** ⚡ – Distributed data processing & analytics
- **Apache Kafka** 📡 – High-throughput message streaming
- **Elasticsearch** 🔍 – Full-text search & indexing
- **Apache Flink** 🌊 – Real-time event processing

Case Studies of Cassandra in Production



🏢 How Leading Companies Use Cassandra

- 🔹 **Netflix** 🎬 – Handles billions of daily video streaming requests
- 🔹 **Instagram** 📸 – Stores and serves petabytes of user-generated content
- 🔹 **Uber** 🚕 – Manages real-time geolocation and ride-matching
- 🔹 **eBay** 🛒 – Powers distributed transactional data for marketplace operations

Thank
You
Merci
Gracias
TerimaKasih
Hvala
Arigatô
KamSahHamnida
Spacibo
Danke
Mèsi
Xièxie
Grazie
Chokrane
Kiitos
Dziakuju
Takk
aDank
BarakAllahuFiik
ThintKo
Buznyg
TapadhLeat
Köszönöm
Murakoze
WaadMahadsantahay
Grazzi
Blagodaram
Rahmat
Matondo
Mercé
Taiku
Dakujem
Gràcies
Faleminderit
Efharisto
NajisTuke
Grandmerce
Tanemirt
Tenki
Aabhar
Dhanyavaad
Mauruuru
Barkal
GratiasAgimus
Ačiū
TänanVäga
Dankie
Motashakkeram
Tak
Sagolun
GratiasAgo
Chnorakaloutioun
KhobChaiDeu
Gratzias
Gracie
KopKhunKha
CámOn
Multumesc
Saha
Salamat
Obrigado
Bayarlalaa
Nirringrazzjak
Mamnun
aNKie