# NoSQL

Apache Cassandra

# Relational Model (SQL)

- Data model
  - Instance → database → table → row
- Query patterns
  - Selection based on complex conditions, projection, joins, aggregation, derivation of new values, recursive queries, …
- Query languages
  - Real-world: SQL (Structured Query Language)
  - Formal: Relational algebra, relational calculi (domain, tuple)
- Representatives
  - Oracle Database, Microsoft SQL Server, IBM DB2
  - MySQL, PostgreSQL

# Relational Model (SQL)

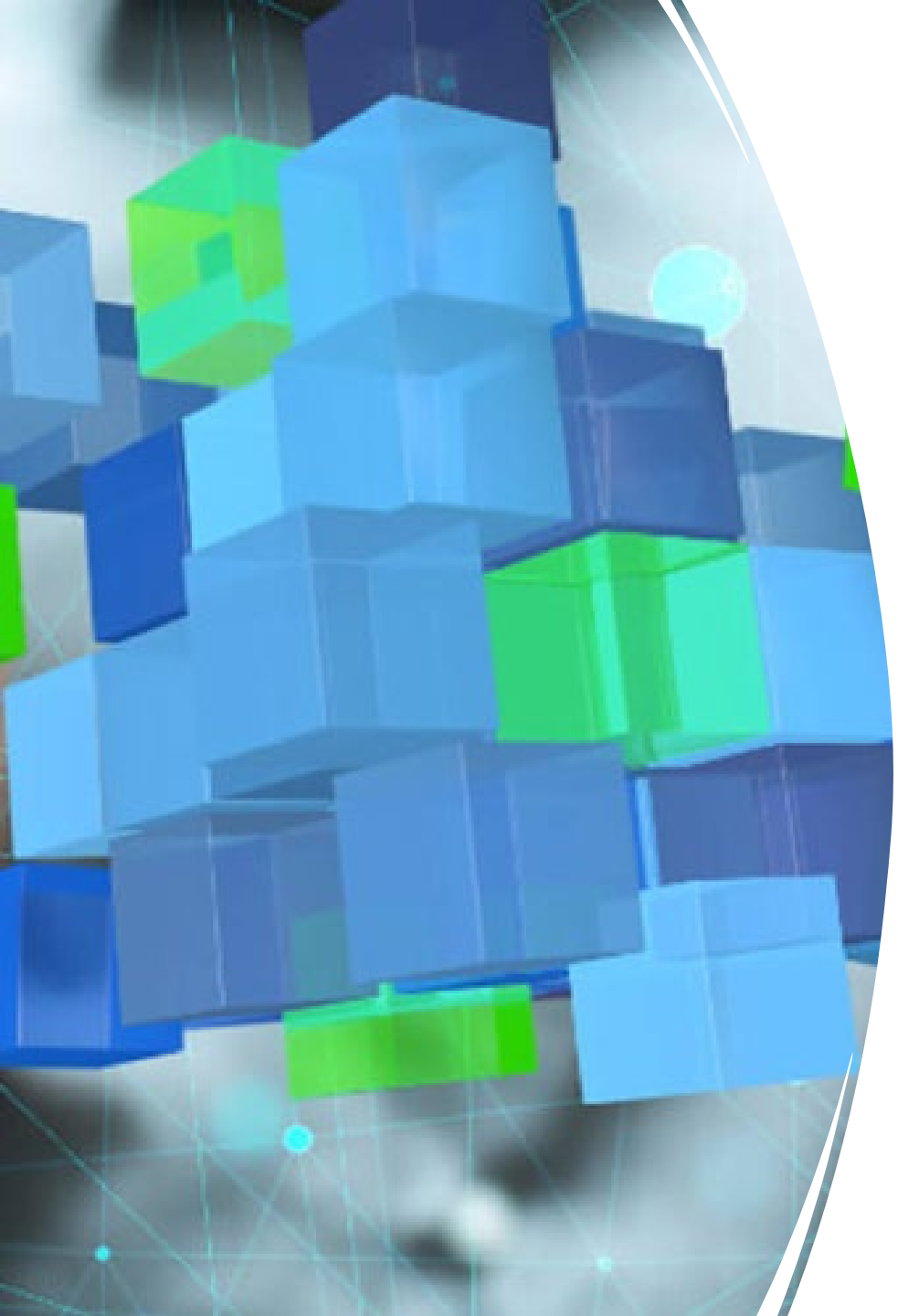
- Model
  - Functional dependencies
  - 1NF, 2NF, 3NF, BCNF (Boyce-Codd normal form)
- Objective
  - Normalization of database schema to BCNF or 3NF
  - Algorithms: decomposition or synthesis
- Motivation
  - Diminish data redundancy, prevent update anomalies
  - However:
    - Data is scattered into small pieces (high granularity), and so
    - these pieces have to be joined back together when querying

# Relational Model (SQL)

- Model
  - Transaction = flat sequence of database operations
  - (READ, WRITE, COMMIT, ABORT)
- Objectives
  - Enforcement of ACID properties
  - Efficient parallel / concurrent execution (slow hard drives, …)
- ACID properties
  - Atomicity – partial execution is not allowed (all or nothing)
  - Consistency – transactions turn one valid database state into another
  - Isolation – uncommitted effects are concealed among transactions
  - Durability – effects of committed transactions are permanent

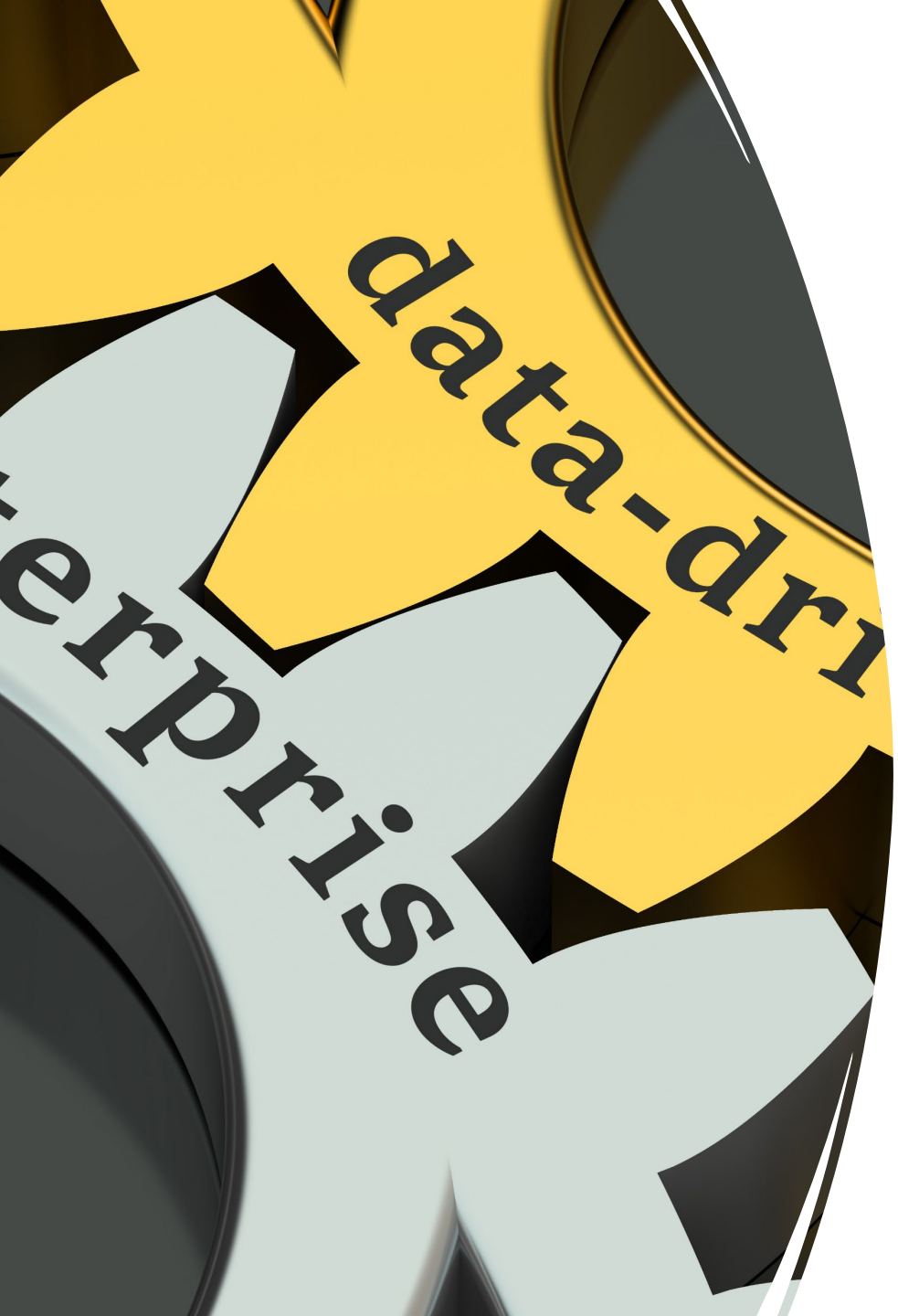
# Current Trends

- Everything is in cloud
  - DBaaS: Database as a Service
    - managed service offering that provides access to a database without requiring the setup of physical hardware, the installation of software or the need to configure the database
    - E.g. Oracle, MongoDB Atlas, Amazon DynamoDB
  - SaaS: Software as a Service
    - products offer both consumers and businesses cloud-based tools and applications for everyday use.
    - E.g. Dropbox
  - PaaS: Platform as a Service
    - products allow businesses and developers to host, build, and deploy consumer-facing apps.
    - E.g. Heroku
  - IaaS: Infrastructure as a Service
    - products allow organizations to manage their business resources — such as their network, servers, and data storage — on the cloud.
    - E.g. Google cloud



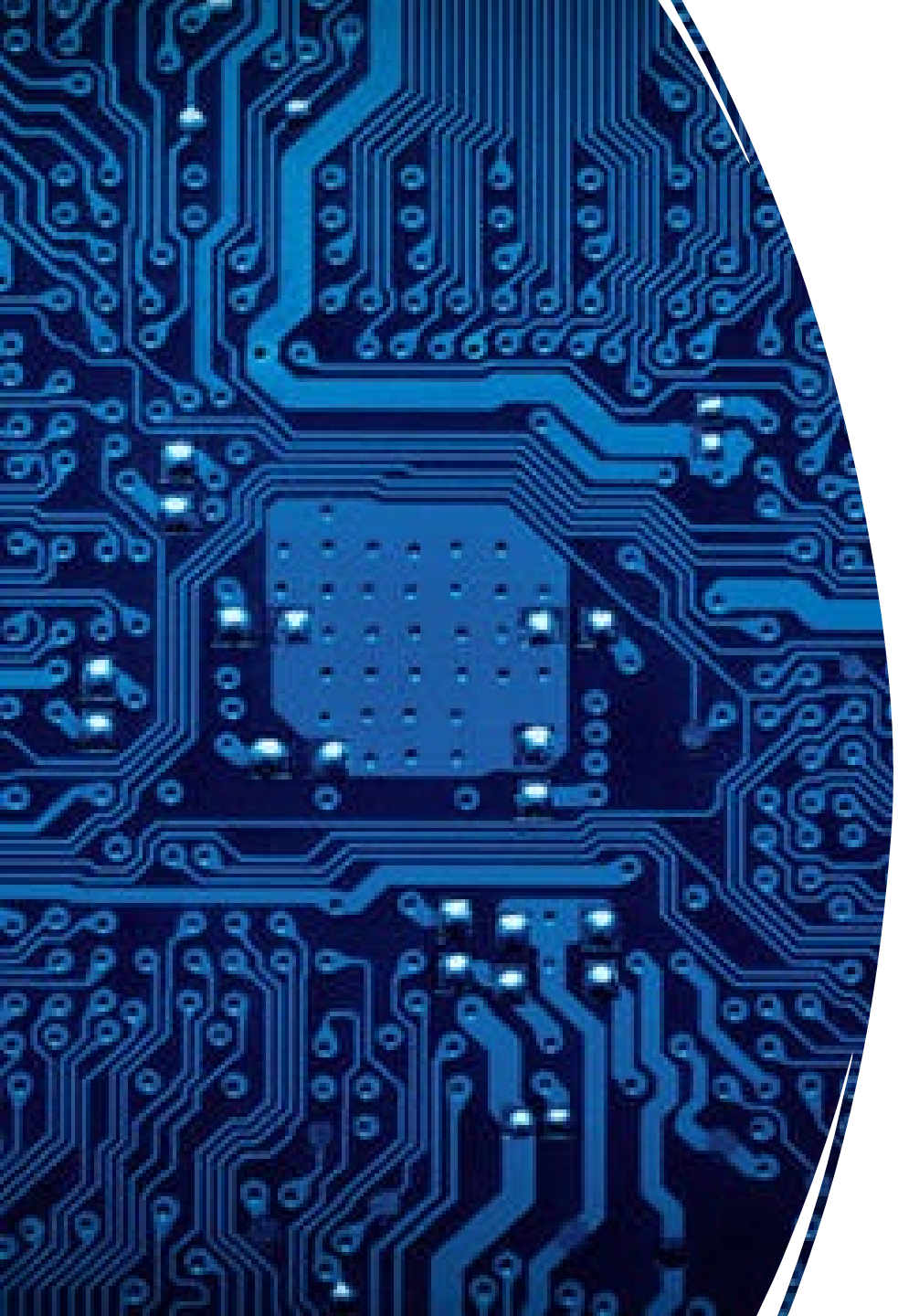
# Current Trends

- OLTP: Online Transaction Processing
  - class of software programs capable of supporting transaction-oriented applications
- OLAP: Online Analytical Processing
  - technology that organizes large business databases and supports complex analysis
- …but also…
- RTAP: Real-Time Analytic Processing
  - framework for developing high-performance distributed computing systems that analyze science data in real-time




# Current Trends

- Data assumptions
  - **Data format** is becoming unknown or inconsistent
  - Data updates are no longer frequent
  - Data is expected to be replaced
  - Linear growth -> **unpredictable exponential growth**
  - Strong **consistency** is no longer mission-critical
  - **Read requests** often prevail **write requests**

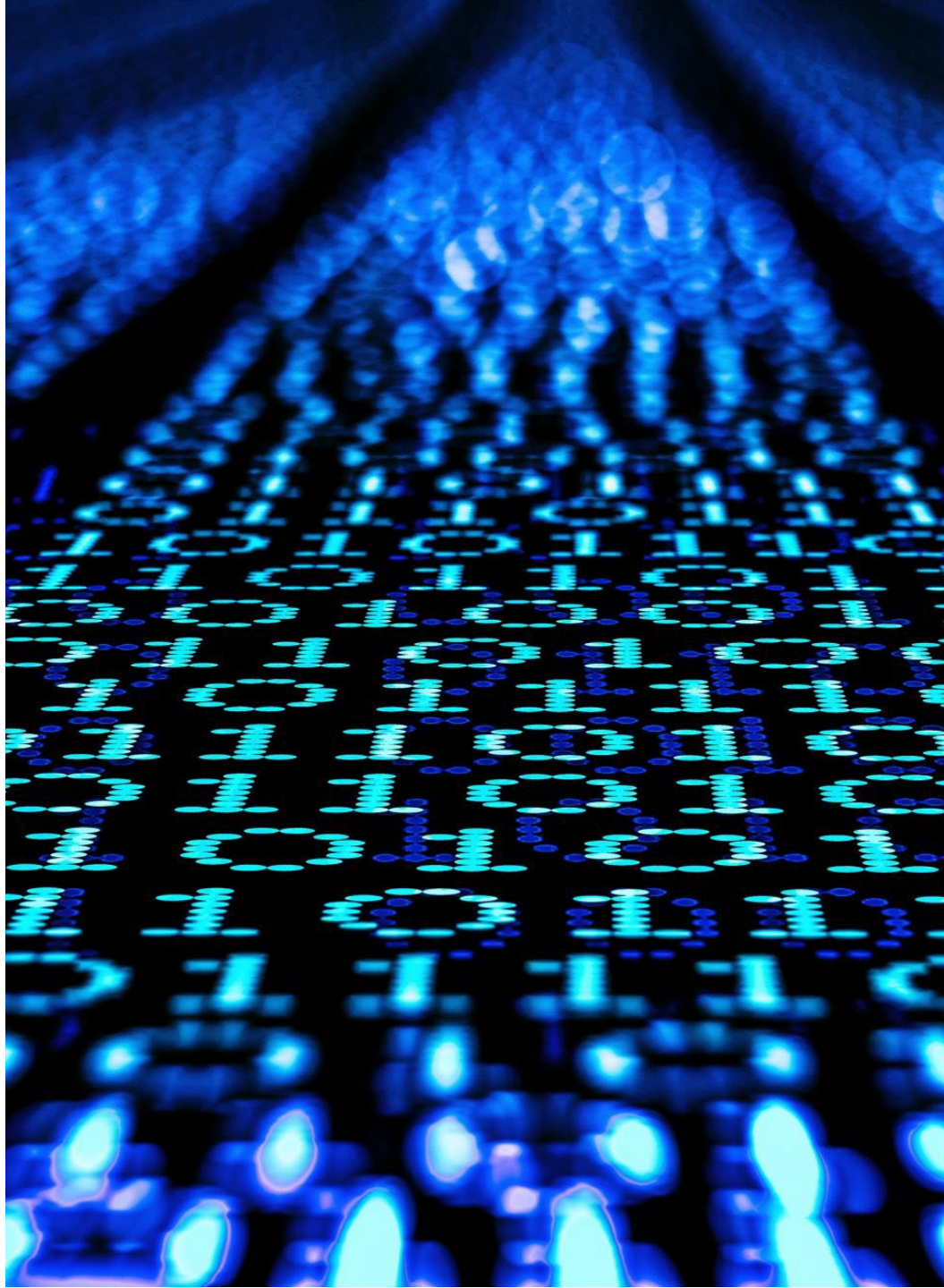


# Current Trends

- New approach is required
  - Relational databases simply do not follow the current trends
- Key technologies
  - Distributed file systems
  - NoSQL databases
  - Data warehouses
  - Data Lakes
  - Grid computing, cloud computing
  - Large scale machine learning

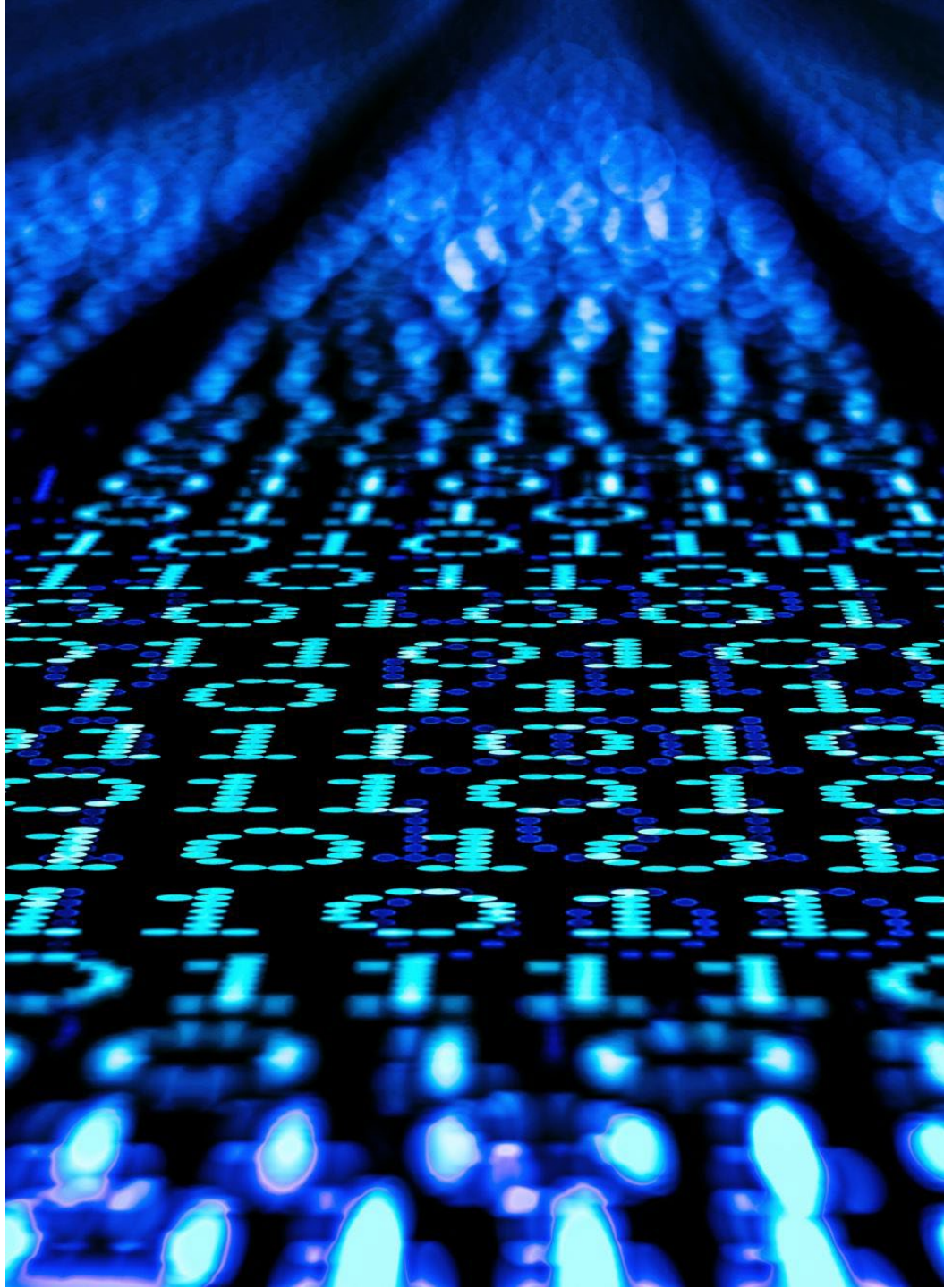
# NoSQL

- NoSQL – “not only SQL” (some say non SQL)
- 1998
  - First used for a relational database that omitted usage of SQL
- 2009
  - First used during a conference to advocate non-relational databases



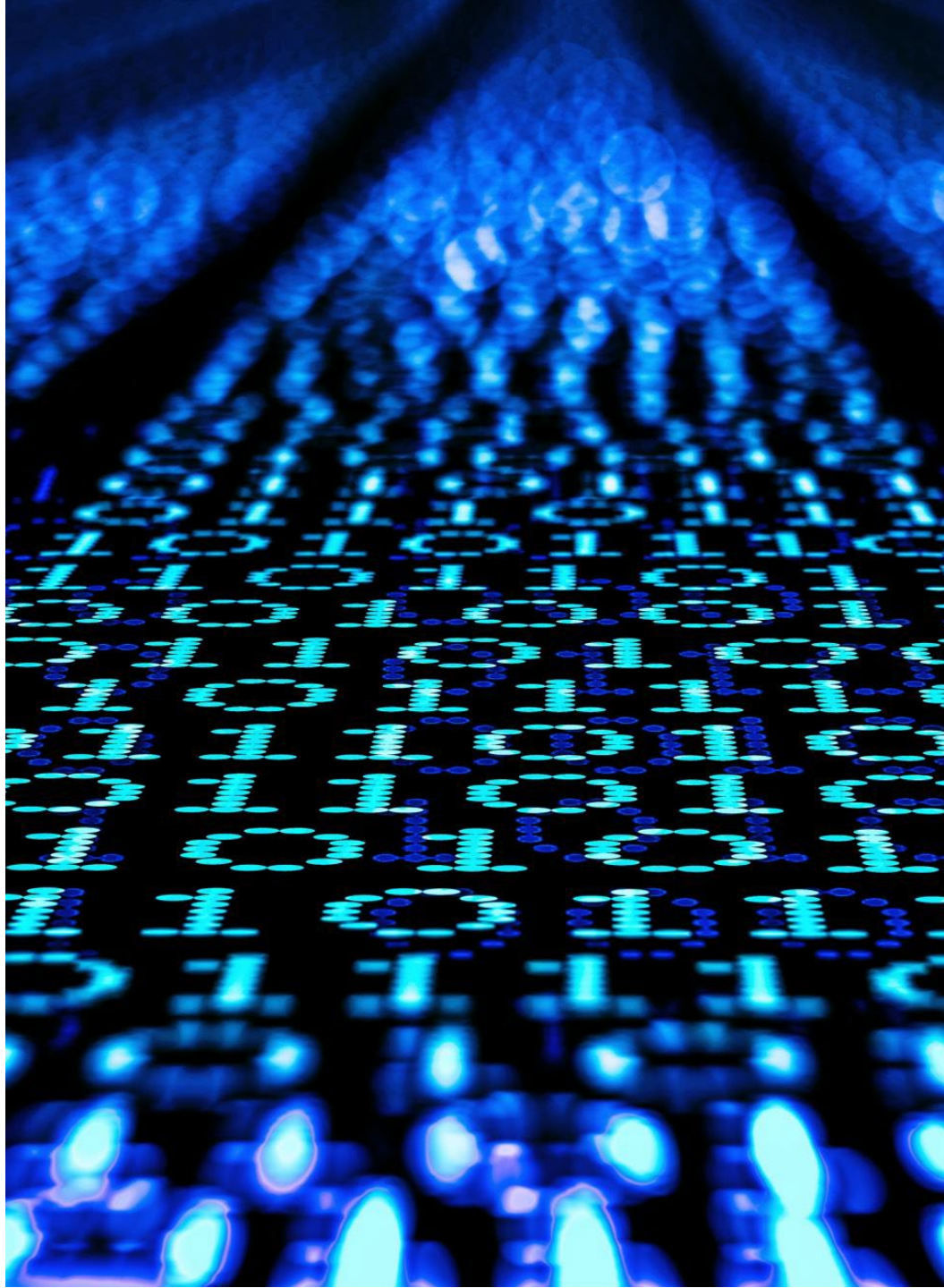
# NoSQL

- Non-tabular databases
- Store data differently than relational tables
- Optimized for developers' productivity
  - Developer time is major cost in software development



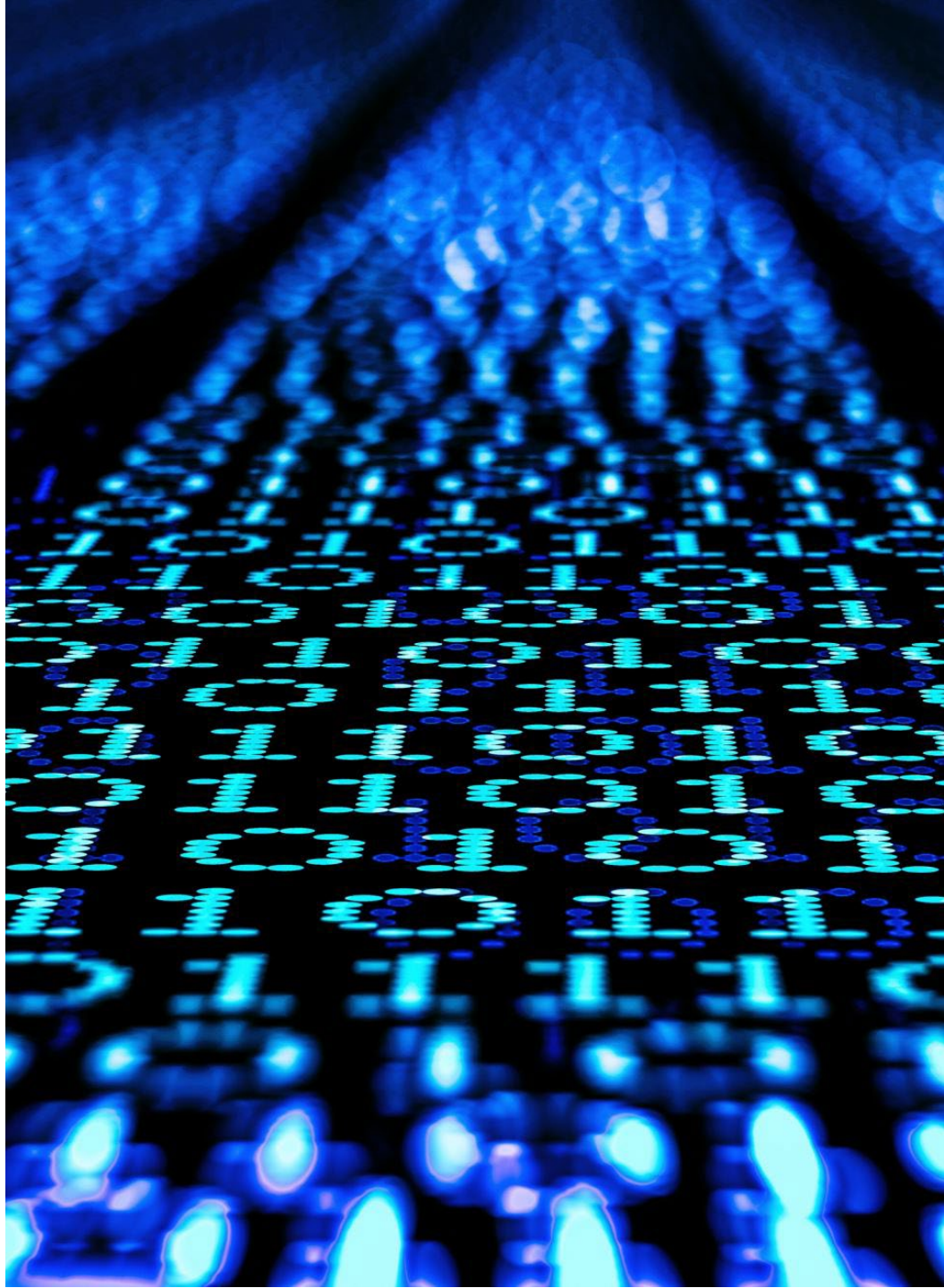
# NoSQL Movement

- The whole point of **seeking alternatives** is that you need to solve a problem that **relational databases are a bad fit for**



# NoSQL Databases

- Next generation databases mostly addressing some of the points: being **non-relational, distributed, open-source and horizontally scalable**.
- The original intention has been modern web-scale databases.
- Often more characteristics apply as**: schema-free, easy replication support, simple API, eventually consistent, a huge amount of data**
  - and more.



# Scalability

- Capability  of a system to handle a growing amount of work, or its potential to perform more total work in the same elapsed time when processing power is expanded to accommodate growth.
- A system is said to be scalable if it can increase its workload and throughput when additional resources are added.

# Scalability

- Vertical scaling, also known as scaling up, is the process of adding resources, such as memory or more powerful CPUs to an existing server.
- Removing memory or changing to a less powerful CPU is known as scaling down.

# Scalability

- Horizontal scaling, sometimes referred to as scaling out, is the process of adding more hardware to a system.
- This typically means adding nodes (new servers) to an existing system.
- Doing the opposite, that is removing hardware, is known as scaling in.

# Replication

- Process of copying data from a primary database to one or more replica databases in order to improve data accessibility and system fault-tolerance and reliability.
- Database replication is typically an ongoing process which occurs in real time as data is created, updated, or deleted in the primary database
  - but it can also occur as one-time or scheduled batch projects.

# Core Types of NoSQL

## Document databases

Store data in documents similar to JSON (JavaScript Object Notation) objects.

Each document contains pairs of fields and values.

- Values can be a variety of types including things like strings, numbers, booleans, arrays, or objects.

## Key-value databases

Simpler type of database where each item contains keys and values.

## Wide-column stores

Store data in tables, rows, and dynamic columns.

## Graph databases

Store data in nodes and edges.

Nodes typically store information about people, places, and things

Edges store information about the relationships between the nodes.

# Non-Core types of NoSQL

- Object databases
- Native XML databases
- RDF stores

# Key-Value Stores

- Data model
  - The most simple NoSQL database type
  - Works as a simple hash table (mapping)
- Key-value pairs
  - Key (id, identifier, primary key)
  - Value: binary object, black box for the database system
- Query patterns
  - Create, update or remove value for a given key
  - Get value for a given key
- Characteristics
  - Simple model => great performance, easily scaled, …
  - Simple model => not for complex queries nor complex data

# Key-Value Stores

- Suitable use cases
  - Session data, user profiles, user preferences, shopping carts, …
  - i.e. when values are only accessed via keys
- When not to use
  - Relationships among entities
  - Queries requiring access to the content of the value part
  - Set operations involving multiple key-value pairs
- Representatives
  - Redis, MemcachedDB, Riak KV, Hazelcast, Amazon
  - SimpleDB, Berkeley DB, Oracle NoSQL, Infinispan, LevelDB,
  - Ignite, Project Voldemort

# Document Stores

- Data model
  - Documents
  - Self-describing
  - Hierarchical tree structures (JSON, XML, …)
  - – Scalar values, maps, lists, sets, nested documents, …
  - Identified by a unique identifier (key, …)
  - Documents are organized into collections
- Query patterns
  - Create, update or remove a document
  - Retrieve documents according to complex query conditions
- Observation
  - Extended key-value stores where the value part is examinable

# Document Stores

- Suitable use cases
  - Event logging, content management systems, blogs, web analytics, e-commerce applications, …
  - I.e. for structured documents with similar schema
- When not to use
  - Set operations involving multiple documents
  - Design of document structure is constantly changing
    - i.e. when the required level of granularity would outbalance he advantages of aggregates
- Representatives
  - MongoDB, Couchbase, Amazon DynamoDB, CouchDB,
  - RethinkDB, RavenDB, Terrastore

# Wide Column Stores

- Data model
  - Column family (table)
    - Table is a collection of similar rows (not necessarily iden cal)
  - Row is a collection of columns
    - Should encompass a group of data that is accessed together
    - Associated with a unique row key
  - Column
    - Column consists of a column name and column value
      - (and possibly other metadata records)
    - Scalar values, but also flat sets, lists or maps may be allowed

# Wide Column Stores

- Query patterns
  - Create, update or remove a row within a given column family
  - Select rows according to a row key or simple conditions
- Warning
  - Wide column stores are not just a special kind of RDBMSs with a variable set of columns!

# Wide Column Stores

- Suitable use cases
  - Event logging, content management systems, blogs, …
  - i.e. for structured flat data with similar schema
- When not to use
  - ACID transactions are required
  - Complex queries: aggregation (SUM, AVG, …), joining, …
  - Early prototypes: i.e. when database design may change
- Representatives
  - Apache Cassandra, Apache HBase, Apache Accumulo,
  - Hypertable, Google Bigtable

# Graph

- Data model
  - Property graphs
  - Directed / undirected graphs, i.e. collections of …
    - – nodes (vertices) for real-world entities, and
    - – relationships (edges) between these nodes
- Both the nodes and relationships can be associated with additional properties
- Types of databases
  - Non-transactional = small number of very large graphs
  - Transactional = large number of small graphs

# Graph

- Query patterns
  - Create, update or remove a node / rela onship in a graph
  - Graph algorithms (shortest paths, spanning trees, …)
  - General graph traversals
  - Sub-graph queries or super-graph queries
  - Similarity based queries (approximate matching)
- Representatives
  - Neo4j, Titan, Apache Giraph, InfiniteGraph, FlockDB

# Graph

- Suitable use cases
  - Social networks, routing, dispatch, and location-based services, recommendation engines, chemical compounds, biological pathways, linguistic trees, …
  - i.e. simply for graph structures
- When not to use
  - Extensive batch operations are required
    - Multiple nodes / relationships are to be affected
  - Only too large graphs to be stored
    - Graph distribution is difficult or impossible at all

# Native XML

- Data model
  - XML documents
  - Tree structure with nested elements, attributes, and text values (beside other less important constructs)
  - Documents are organized into collections
- Query languages
  - XPath: XML Path Language (navigation)
  - XQuery: XML Query Language (querying)
  - XSLT: XSL Transformations (transformation)
- Representatives
  - Sedna, Tamino, BaseX, eXist-db

# Features of NoSQL

- Data model
  - Alternative to Traditional approach: relational model
  - (New) possibilities:
    - Key-value, document, wide column, graph
    - Object, XML, RDF, …
- Goal
  - Respect the real-world nature of data
    - (i.e. data structure and mutual relationships)

# Features of NoSQL

- Aggregate structure
  - Aggregate definition
  - Data unit with a complex structure
  - Collection of related data pieces we wish to treat as a unit (with respect to data manipulation and data consistency)
- Examples
  - Value part of key-value pairs in key-value stores
  - Document in document stores
  - Row of a column family in wide column stores

# Features of NoSQL

- Aggregate structure
  - Types of systems
    - Aggregate-ignorant: relational, graph
      - It is not a bad thing, it is a feature
    - Aggregate-oriented: key-value, document, wide column
- Design notes
  - No universal strategy how to draw aggregate boundaries
  - Atomicity of database operations: just a single aggregate at a time

# Features of NoSQL

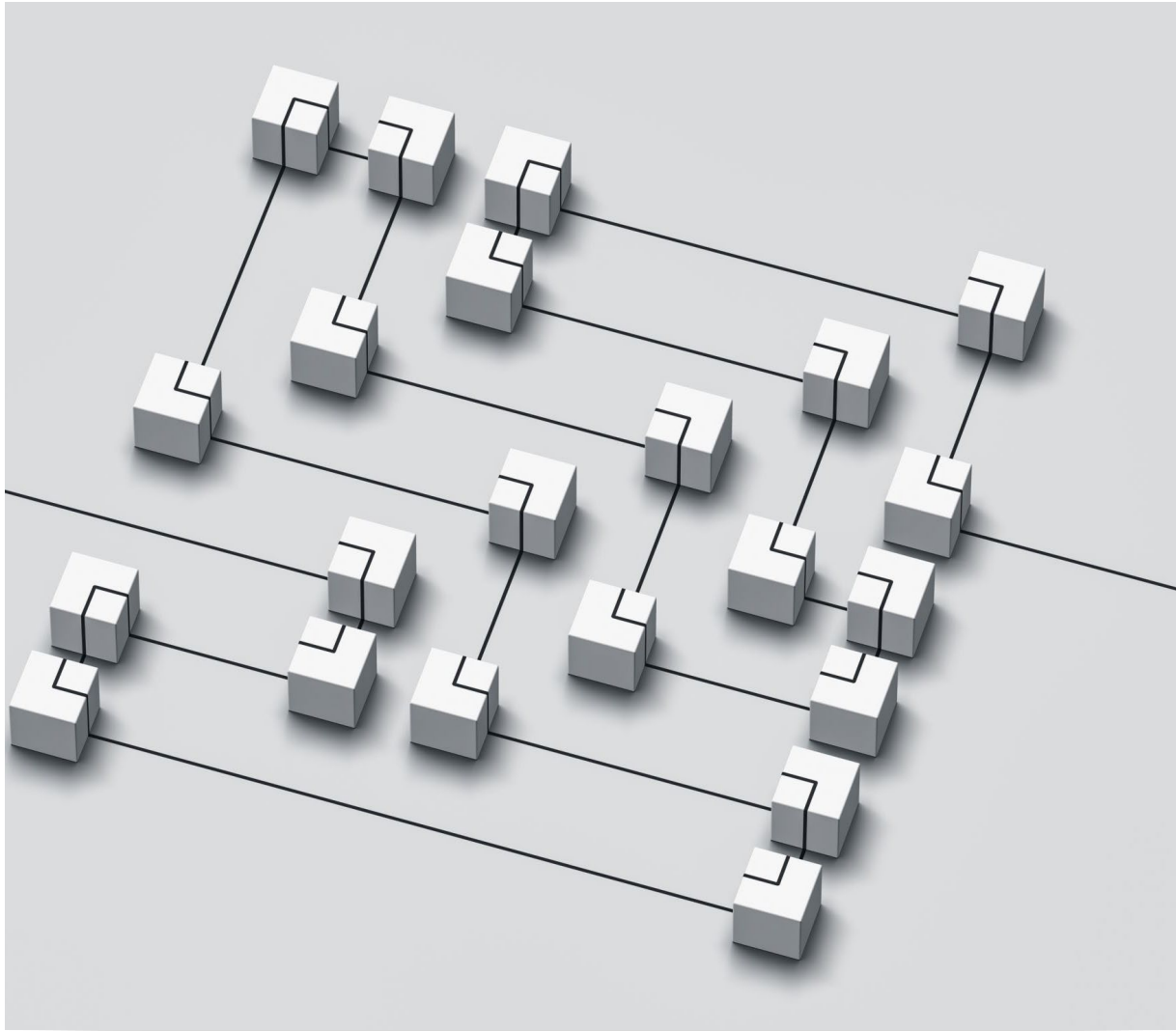
- Elastic scaling
  - Traditional approach: scaling-up
    - Buying bigger servers as database load increases
  - New approach: scaling-out
  - Distributing database data across multiple hosts
    - Graph databases (unfortunately): difficult or impossible at all
- Data distribution
  - Sharding
    - Particular ways  database data is split into separate groups
  - Replication
    - Maintaining several data copies (performance, recovery)

# Features of NoSQL

- Automated processes
  - Traditional approach
    - Expensive and highly trained database administrators
  - New approach: automatic recovery, distribution, tuning, …
- Relaxed consistency
  - Traditional approach
    - Strong consistency (ACID properties and transactions)
  - New approach
    - Eventual consistency only (BASE properties)
    - I.e. we have to make trade-offs because of the data distribution

# BASE

- **B**asically **A**vailable
  - Rather than enforcing immediate consistency, BASE-modelled NoSQL databases will ensure availability of data by spreading and replicating it across the nodes of the database cluster.
- **S**oft State
  - Due to the lack of immediate consistency, data values may change over time.
  - The BASE model breaks off with the concept of a database which enforces its own consistency, delegating that responsibility to developers.
- **E**ventually Consistent
  - The fact that BASE does not enforce immediate consistency does not mean that it never achieves it.
  - However, until it does, data reads are still possible (even though they might not reflect the reality).



# Features of NoSQL

- Schemalessness
- Relational databases
  - Database schema present and strictly enforced
- NoSQL databases
  - Relaxed schema or completely missing
  - Consequences: higher flexibility
    - Dealing with non-uniform data
    - Structural changes cause no overhead
  - However: there is (usually) an implicit schema
    - We must know the data structure at the application level anyway

# Features of NoSQL

- Open source
  - Often community and enterprise versions (with extended features or extent of support)
- Simple APIs
  - Often state-less application interfaces (HTTP)

# Current Advantages SQL databases

- Speed
  - Large amount of data is retrieved quickly and efficiently. Operations like Insertion, deletion, manipulation of data is also done in almost no time.
- No Coding Skills
  - For data retrieval, large number of lines of code is not required. All basic keywords such as SELECT, INSERT INTO, UPDATE, etc are used and also the syntactical rules are not complex in SQL, which makes it a user-friendly language.
- Standardized Language
  - Due to documentation and long establishment over years, it provides a uniform platform worldwide to all its users.
- Portable
  - It can be used in programs in PCs, server, laptops independent of any platform (Operating System, etc). Also, it can be embedded with other applications as per need/requirement/use.
- Interactive Language
  - Easy to learn and understand, answers to complex queries can be received in seconds
- Expertise
  - Widespread expertise and support available

# Current Challenges SQL databases

- Complex Interface
  - Since SQL has a complex structure, it becomes difficult for certain users to access it.
- Only partial control
  - Since there are certain hidden rules and conditions, the programmers who use SQL do not have power over the database.
- Expense
  - The expenses involved in SQL operation is too much, making it difficult for bringing vendor-in.

# Current Advantages NoSQL databases

- Scaling
  - Horizontal distribution of data among hosts
- Volume
  - High volumes of data that cannot be handled by RDBMS
- Administrators
  - No longer needed because of the automated maintenance
- Economics
  - Usage of cheap commodity servers, lower overall costs
- Flexibility
  - Relaxed or missing data schema, easier design changes

# Current Challenges NoSQL databases

- Maturity
  - Often still in pre-production phase with key features missing
- Support
  - Mostly open source, limited sources of credibility
- Administration
  - Sometimes relatively difficult to install and maintain
- Analytics
  - Missing support for business intelligence and ad-hoc querying
- Expertise
  - Still low number of NoSQL experts available in the market

# Is the reign of relational databases over?

## NO

- They are still suitable for most projects
- Familiarity, stability, feature set, available support, …

## However, we should also consider different database models and systems

- **Polyglot persistence** = usage of different data stores in different circumstances
- Choose the store that is appropriate for the data rather than make the data suitable for the storage mechanism




# Apache Cassandra

# Apache Cassandra



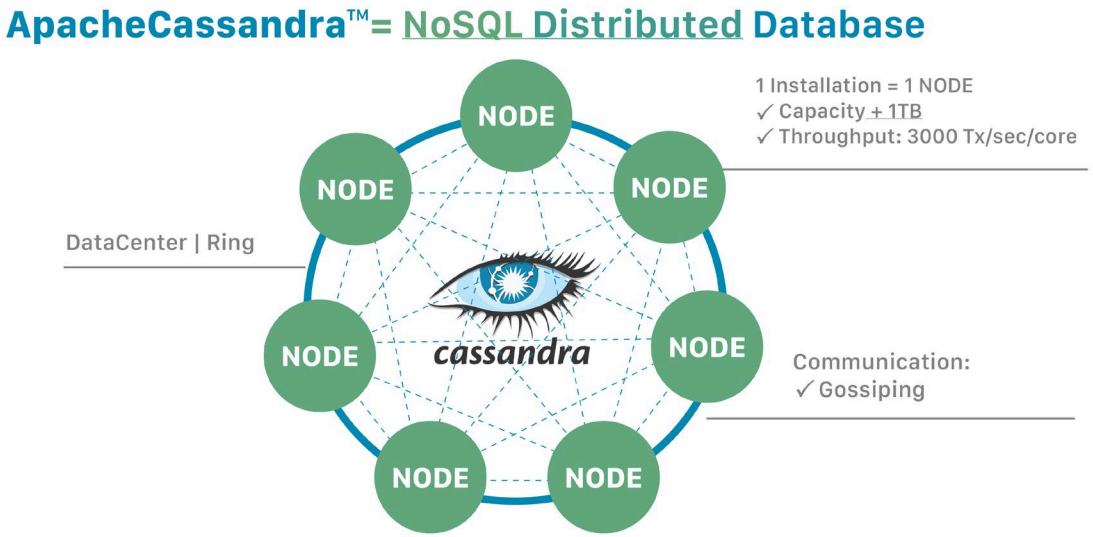

- **Distributed** NoSQL database management system
- Wide column store
  - Stores data in tables with rows and columns
- Data model is based around optimizing for queries
- Stores data across a **cluster** of nodes
- CQL (Cassandra Query Language) is used to query the data stored in tables

# Apache Cassandra

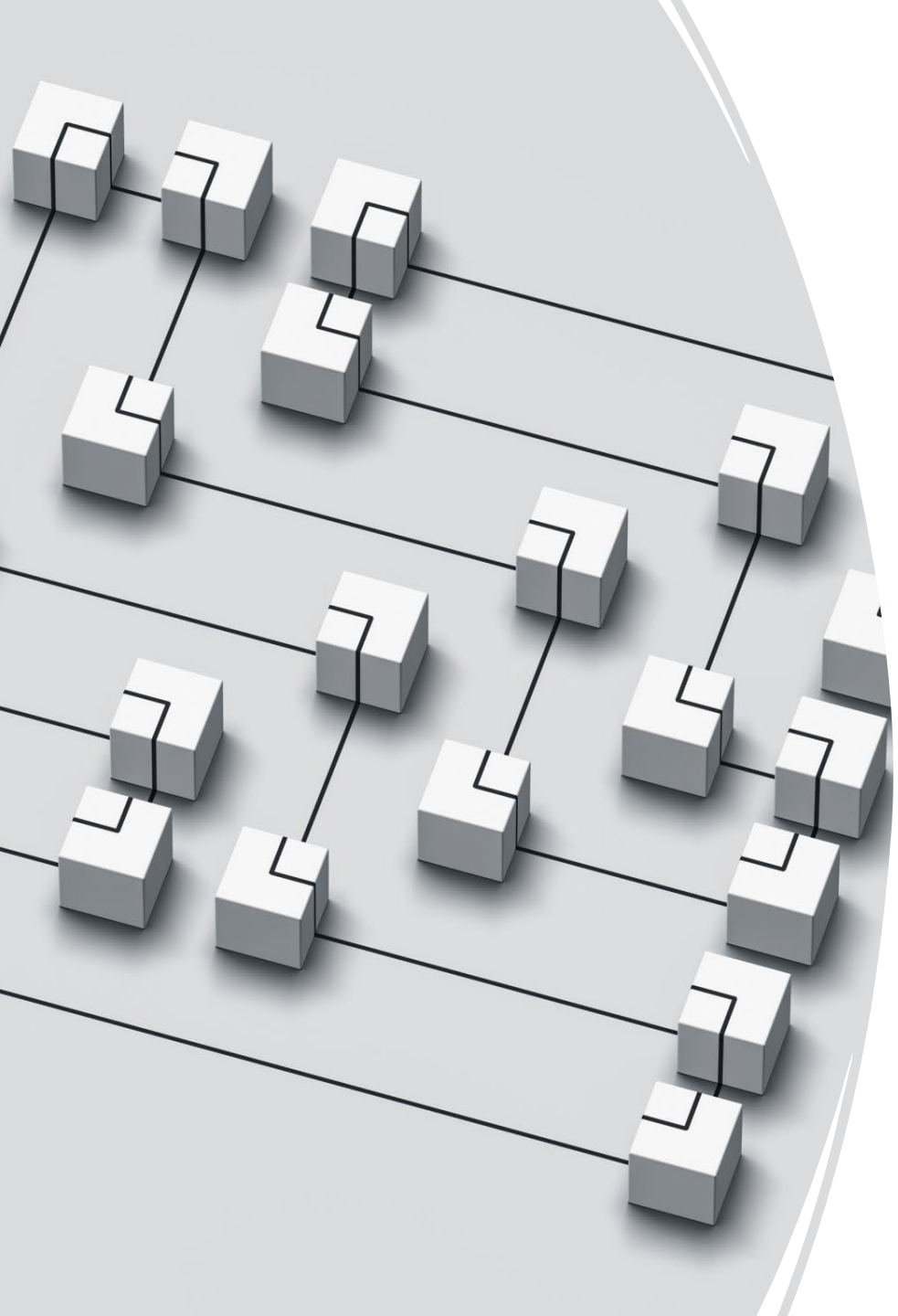



- Cassandra CQLsh stands for Cassandra CQL shell.
- Cassandra provides Cassandra query language shell (cqlsh) which facilitates users to communicate with it.

# Apache Cassandra




- Distributed
  - Cassandra can run on multiple machines while appearing to users as a unified whole.
- Nodes
  - Cassandra can (and usually does) have multiple nodes.
  - A node represents a single instance of Cassandra.
  - These nodes communicate with one another through a protocol called **gossip**, which is a process of computer peer-to-peer communication.
  - Cassandra also has a **masterless** architecture
    - Any node in the database can provide the exact same functionality as any other node
    - This contributes to Cassandra's robustness and resilience.
- Cluster
  - Multiple nodes can be organized logically into a cluster, or "ring".



# Query Driven Modelling

- The data access patterns and application queries determine the structure and organization of data which then used to design the database tables
- Joins are not supported
  - Therefore all required fields (columns) must be grouped together in a single table
- Each query is backed by a table
  - Data is duplicated across multiple tables using denormalization
  - This plus a high write throughput are used to achieve a high read performance

# Partitions

- Cassandra is a distributed database that stores data across a cluster of nodes.
- Cassandra partitions data over the storage nodes using a variant of consistent hashing for data distribution
  - Hashing is a technique used to map data
  - Given a key, a hash function generates a hash value (or simply a hash) that is stored in a hash table.

# Partitions

- A partition key is used to partition data among the nodes
- A partition key is generated from the first field of a primary key.
- This facilitates a faster  lookup.
  - The fewer partitions used for a query faster is the response time for the query.

# Cassandra Data model

- Cassandra data model consists of **keyspaces** at the highest level.
- Keyspaces are the containers of data
  - similar to the schema or database in a relational database.
- Typically, keyspaces contain many tables.

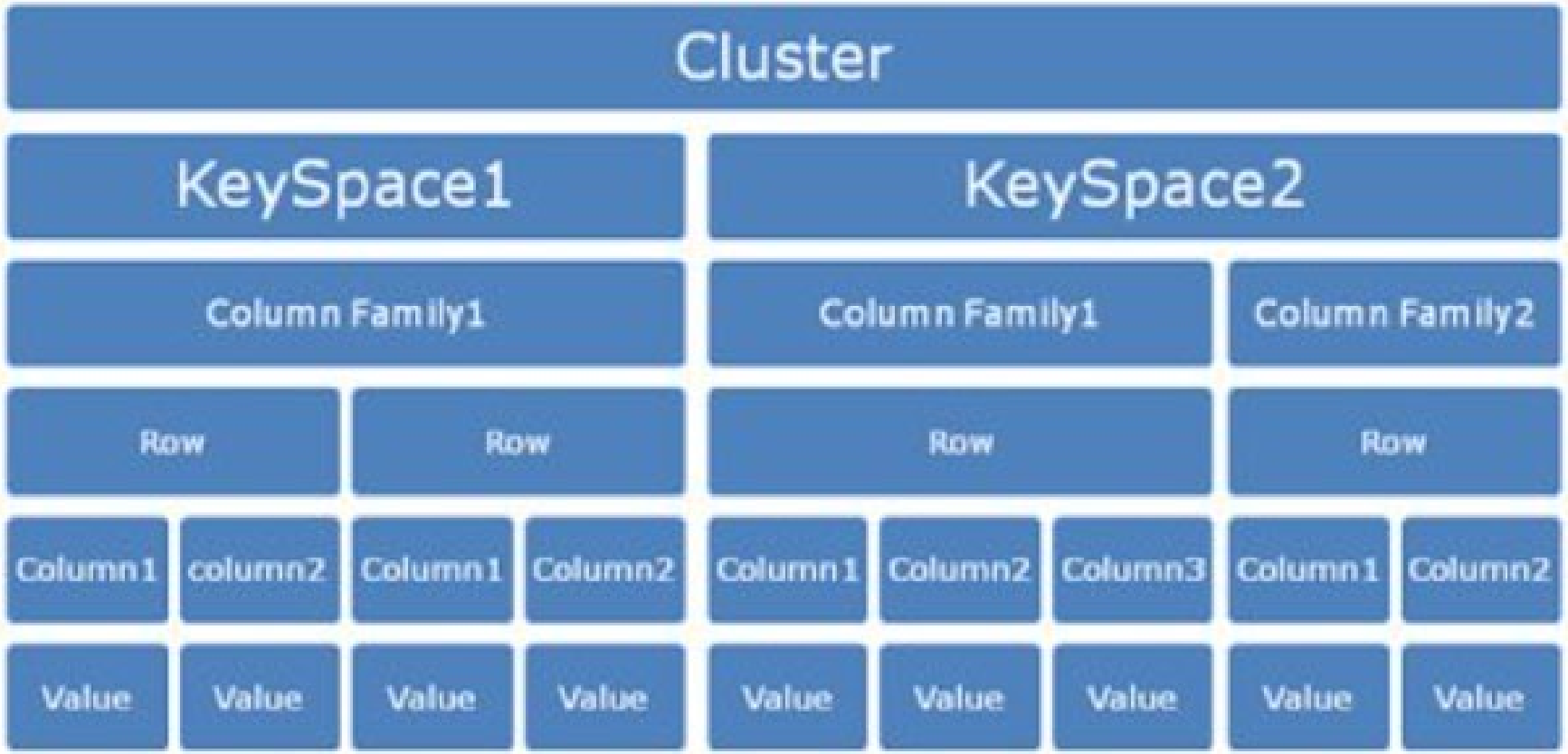
# Cassandra Data model

- The **tables** are defined within the keyspaces.
- Tables are also referred to as Column Families in the earlier versions of Cassandra.
- Tables contain
  - a **set of columns**
  - and a **primary key**
  - and they store data in a **set of rows**.

# Cassandra Data model

- **Columns** define the structure of data in a table.
- Each column has an associated type
  - E.g. integer, text, double, and Boolean.

# Cassandra Data Mode



# Cassandra Data Model - Rules

- Writes are not expensive
- Joins are not supported
- Group by, OR clause, aggregations are not supported
- You need to store your data in such a way that it should be completely retrievable.
- Need to keep the rules in mind when modelling data in Cassandra
  - Maximise the number of writes
  - Maximise data duplication

# Cassandra Data Model - Rules

- Need to keep the rules in mind when modelling data in Cassandra
  - Maximise the number of writes
    - Try to maximize your writes for better read performance and data availability.
    - There is a tradeoff between data write and data read.
    - Optimize you data read performance by maximizing the number of data writes.
  - Maximise data duplication
    - Data denormalization and data duplication are defacto of Cassandra.
    - Disk space is not more expensive than memory, CPU processing and IOs operation.
    - Cassandra is a distributed database, so data duplication provides instant data availability and no single point of failure.

# Cassandra Data Model - Goals

- Spread Data Evenly Around the Cluster
- You want an equal amount of data on each node of Cassandra Cluster.
- Data is spread to different nodes based on partition keys that is the first part of the primary key.
  - Try to choose integers as a primary key for spreading data evenly around the cluster.

# Cassandra Data Model - Goals

- Primary key - consists of one or more partition keys and zero or more clustering key components
- Partition key
  - Goal of a partition key is to distribute the data evenly across a cluster and query the data efficiently
  - A partition key is for data placement apart from uniquely identifying the data and is always the first value in the primary key definition.
- Clustering column – orders the data within a partition

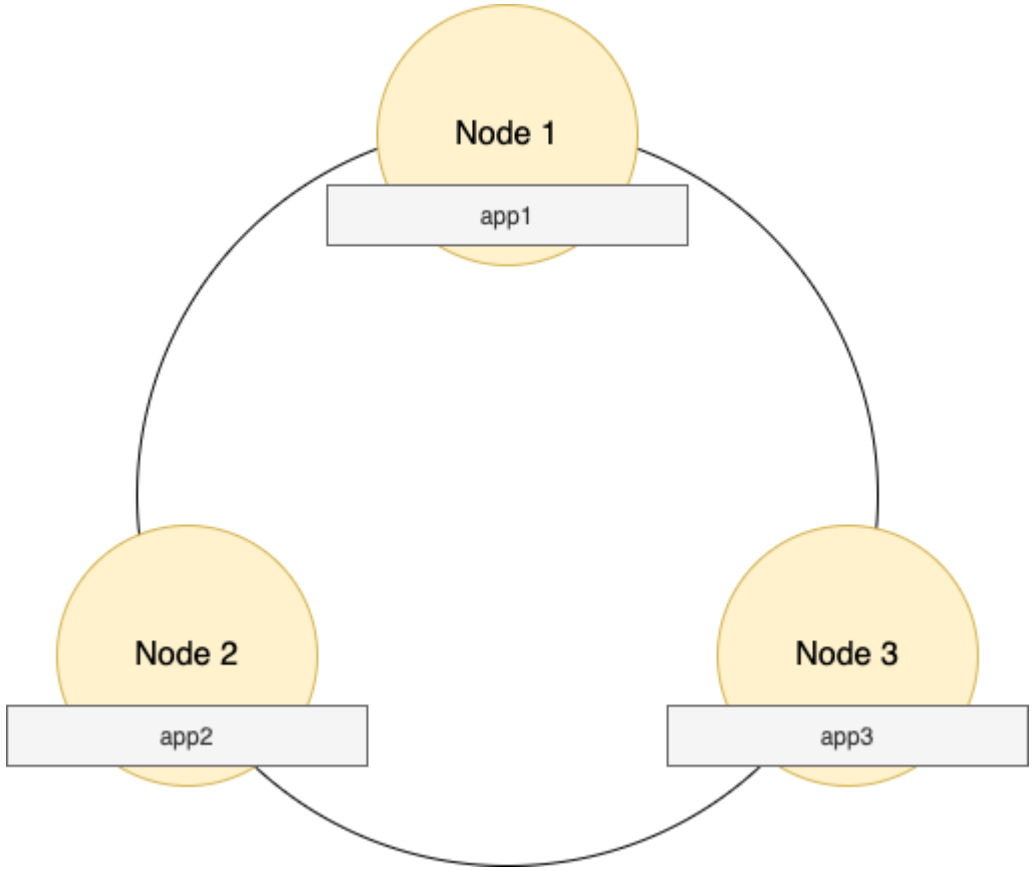
# Cassandra Data Model – Simple Example

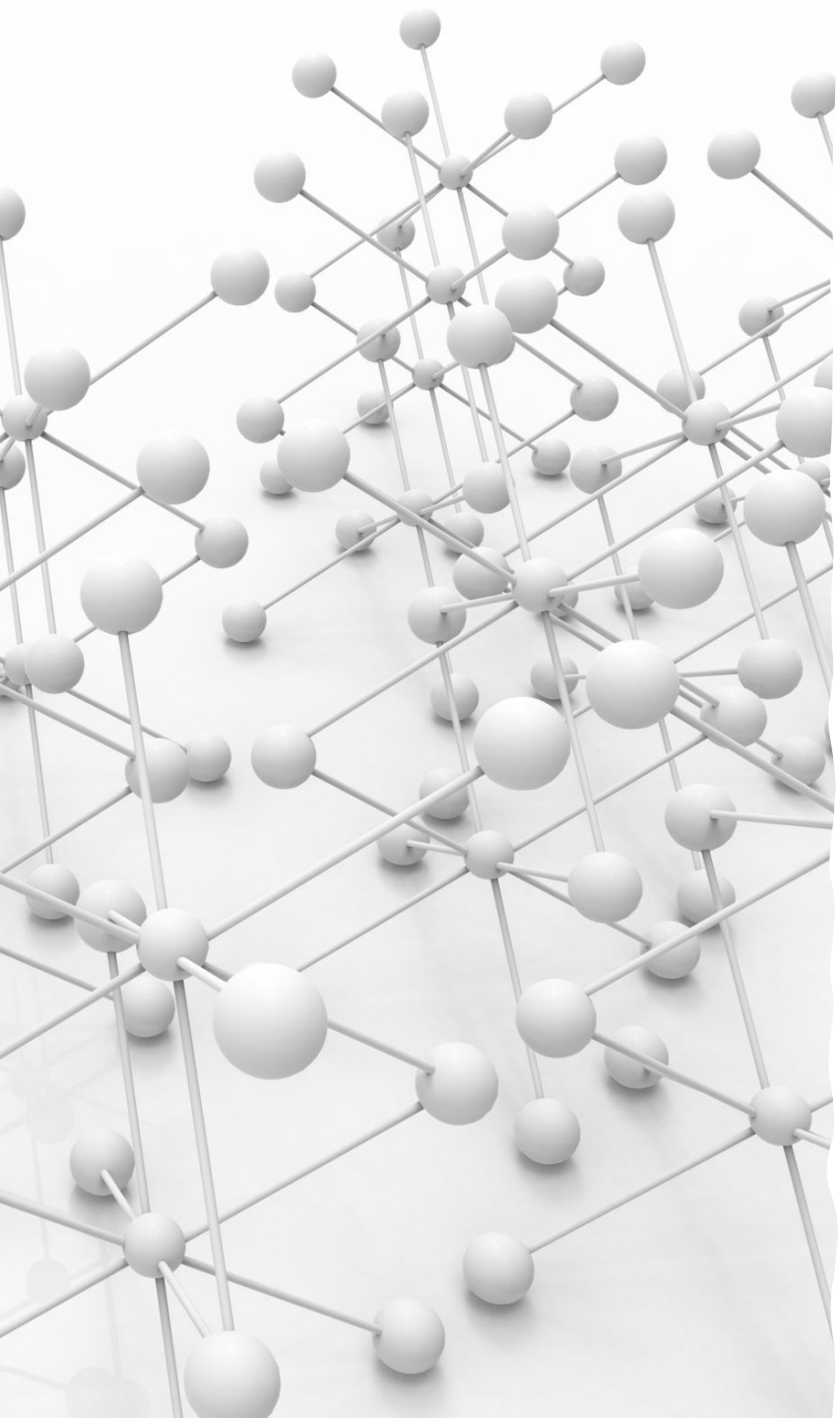
```
CREATE TABLE application_logs (
  id                    INT,
  app_name              VARCHAR,
  hostname              VARCHAR,
  log_datetime          TIMESTAMP,
  env                   VARCHAR,
  log_level             VARCHAR,
  log_message           TEXT,
  PRIMARY KEY (app_name)
);
```

| id | app_name | env | hostname | log_datetime | log_level | log_message |
|---|---|---|---|---|---|---|
| 1 | app1 | prod | host1 | 2021-08-15 04:05:00.000+0000 | INFO | app1 INFO message |
| 2 | app2 | dev | host2 | 2021-08-14 12:30:00.000+0000 | INFO | app2 log message |
| 3 | app3 | dev | host2 | 2021-08-13 16:45:50.000+0000 | WARN | app3 log message |

# Cassandra Data Model – Simple Example

- Cassandra uses a consistent hashing technique to generate the hash value of the partition key (app_name) and assign the row data to a partition range inside a node.
- Possible arrangement:
  - the hash values of app1, app2, and app3 resulted in each row being stored in three different nodes — Node1, Node2, and Node3, respectively.





# Cassandra Data model – Simple Example

- With a partition key in where clause, Cassandra uses the consistent hashing technique to identify the exact node and the exact partition range within a node in the cluster.
- As a result, the fetch data query is fast and efficient:

```
select * application_logs where
app_name = 'app1';
```

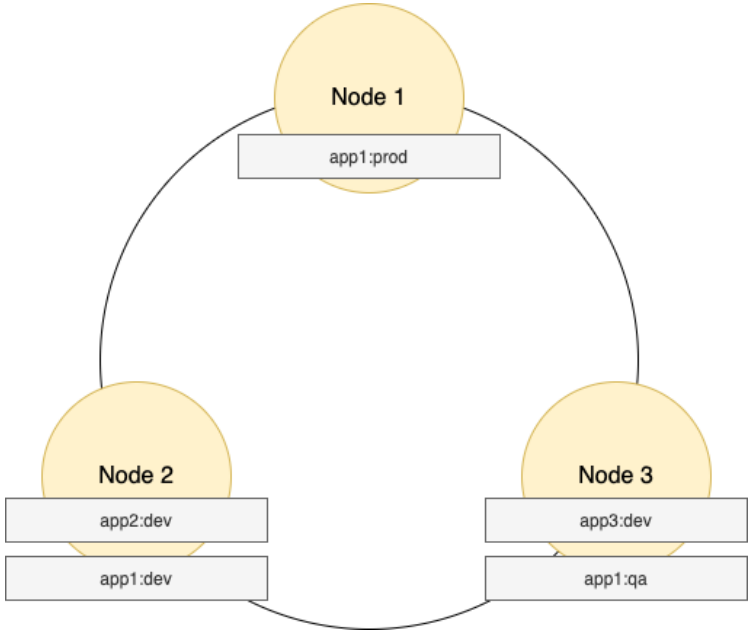
# Cassandra Data Model – Simple Example

- Composite Primary Key
  - Note in the above definition is the inner parenthesis around app_name and env primary key definition.
  - This inner parenthesis specifies that app_name and env are part of a partition key and are not clustering keys
  - If you just have a single parenthesis then app_name will be the partition key and env will be the clustering column

```
CREATE TABLE application_logs (
   id                              INT,
   app_name                        VARCHAR,
   hostname                        VARCHAR,
   log_datetime                    TIMESTAMP,
   env                             VARCHAR,
   log_level                       VARCHAR,
   log_message                     TEXT,
   PRIMARY KEY ((app_name, env))
);
```

# Cassandra Data Model – Simple Example

| id | app_name | env | hostname | log_datetime | log_level | log_message |
|---|---|---|---|---|---|---|
| 1 | app1 | prod | host1 | 2021-08-15 04:05:00.000+0000 | INFO | app1 INFO message |
| 2 | app2 | dev | host2 | 2021-08-14 12:30:00.000+0000 | INFO | app2 log message |
| 3 | app3 | dev | host2 | 2021-08-13 16:45:50.000+0000 | WARN | app3 log message |
| 4 | app1 | dev | host2 | 2021-08-10 11:35:50.000+0000 | INFO | app1 log message |
| 5 | app1 | qa | host2 | 2021-08-13 13:00:50.000+0000 | INFO | app1 log message |




The possible scenario where the hash value of app1:prod, app1:dev, app1:qa resulted in these three rows being stored in three separate nodes — Node1, Node2, and Node3, respectively.

All app1 logs from the prod environment go to Node1, while app1 logs from the dev environment go to Node2, and app1 logs from the qa environment go to Node3.

# Cassandra Data Model – Simple Example

- To efficiently retrieve data, the where clause in fetch query must contain all the composite partition keys in the same order as specified in the primary key definition:

```
select * application_logs where app_name = 'app1' and env = 'prod';
```

# Cassandra Data Model – Clustering Key

- Clustering is a storage engine process of sorting the data within a partition and is based on the columns defined as the clustering keys.
- Identification of the clustering key columns needs to be done upfront — that's because our selection of clustering key columns depends on how we want to use the data in our application.
- All the data within a partition is stored in continuous storage, sorted by clustering key columns.
- As a result, the retrieval of the desired sorted data is very efficient.

# Cassandra Data Model – Clustering Key

```
CREATE TABLE application_logs (
  id                    INT,
  app_name              VARCHAR,
  hostname              VARCHAR,
  log_datetime          TIMESTAMP,
  env                   VARCHAR,
  log_level             VARCHAR,
  log_message           TEXT,
  PRIMARY KEY ((app_name, env), hostname, log_datetime)
);
```

The hostname and the log_datetime are included as clustering key columns.
Assuming all the logs from app1 and prod environment are stored in Node1, the Cassandra storage engine lexically sorts those logs by the hostname and the log_datetime within the partition.

| id | app_name | env | hostname | log_datetime | log_level | log_message |
|---|---|---|---|---|---|---|
| 1 | app1 | prod | host1 | 2021-08-15 04:05:00.000+0000 | INFO | app1 INFO message |
| 2 | app1 | prod | host1 | 2021-08-15 07:15:00.000+0000 | DEBUG | app1 DEBUG message |
| 3 | app2 | dev | host2 | 2021-08-14 12:30:00.000+0000 | INFO | app2 log message |
| 4 | app3 | dev | host2 | 2021-08-13 16:45:50.000+0000 | WARN | app3 log message |
| 5 | app1 | dev | host2 | 2021-08-10 11:35:50.000+0000 | INFO | app1 log message |
| 6 | app1 | qa | host2 | 2021-08-13 13:00:50.000+0000 | INFO | app1 log message |

# Cassandra Data Model – Clustering Key

- By default, the Cassandra storage engine sorts the data in ascending order of clustering key columns
- But we can control the clustering columns' sort order by using WITH CLUSTERING ORDER BY clause in the table definition

```
CREATE TABLE application_logs (
  id                    INT,
  app_name              VARCHAR,
  hostname              VARCHAR,
  log_datetime          TIMESTAMP,
  env                   VARCHAR,
  log_level             VARCHAR,
  log_message           TEXT,
  PRIMARY KEY ((app_name,env), hostname, log_datetime)
)
WITH CLUSTERING ORDER BY (hostname ASC, log_datetime DESC);
```

Based on this definition the Cassandra storage engine will store all logs in the lexical ascending order of hostname, but in descending order of log_datetime within each hostname group.

# Cassandra Data Model – Simple Example

```
select * application_logs
where
app_name = 'app1' and env = 'prod'
and hostname = 'host1' and log_datetime > '2021-
08-13T00:00:00';
```

The *where* clause should contain the columns in the same order as defined in the primary key clause.

# Cassandra Data Model – Summary

Uses a partition key or a composite partition key to determine the placement of the data in a cluster.

The clustering key provides the sort order of the data stored within a partition.

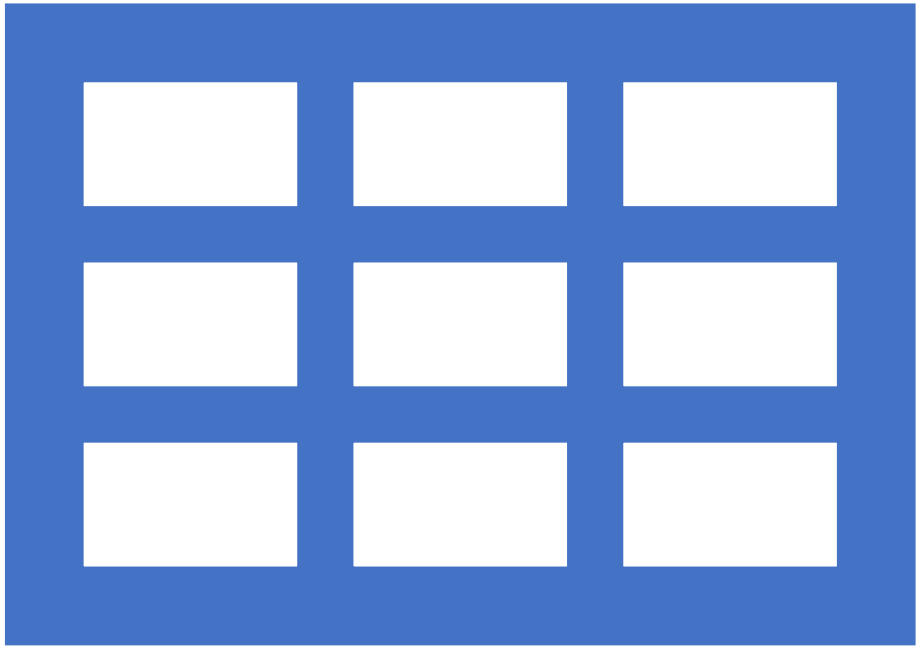
All of these keys also uniquely identify the data

# Creating a Keyspace

- Keyspace is an object that is used to hold column families, user defined types.
- A keyspace is like RDBMS database which contains column families, indexes, user defined types, data centre awareness, strategy used in keyspace, **replication factor**, etc

```
CREATE KEYSPACE <identifier> WITH <properties>
```

```
Create keyspace KeyspaceName with replicaton={'class':strategy name, 'replication_factor': No of replications on different nodes}
```



# Replication in Cassandra

- Cassandra stores data replicas on multiple nodes to ensure reliability and fault tolerance.
- The replication strategy for each  keyspace determines the nodes where replicas are placed.
- The total number of replicas for a keyspace across a Cassandra cluster is referred to as the keyspace's ***replication factor***.

# Keyspace

- Replication Factor:
  - The number of replicas of data placed on different nodes.
  - More than two replication factor are good to attain no single point of failure.
  - 3 is good replication factor

```
CREATE KEYSPACE example
WITH replication = {'class':'Simple
Strategy', 'replication_factor' : 3
};
```

# Keyspace

- Strategy:
  - There are two types of strategy declaration in Cassandra syntax:
  - Simple Strategy:
    - Used in the case of one data centre.
    - In this strategy, the first replica is placed on the selected node and the remaining nodes are placed in clockwise direction in the ring without considering rack or node location.
  - Network Topology Strategy:
    - Used in the case of more than one data centres. In this strategy, you have to provide replication factor for each data centre separately.

# Keyspace

- To use a keyspace  you have to use the USE command.

```
USE <identifier>
```

# Keyspace

```
CREATE TABLE tablename(
   column1 name datatype
PRIMARYKEY,
   column2 name data type,
   column3 name data type.
   )
```

Declaring primary keys:

- Primary key (ColumnName)
- Primary key(ColumnName1,ColumnName2 . . .)

# Creating Data

```
INSERT INTO <tablename>
(<column1 name>, <column2 name>....)
VALUES (<value1>, <value2>....)
USING <option>
```

**Example:**

```
INSERT INTO student (student_id, student_fees,
student_name)
VALUES(1,5000, 'Ajeet');
INSERT INTO student (student_id, student_fees,
student_name)
VALUES(2,3000, 'Kanchan');
INSERT INTO student (student_id, student_fees,
student_name)
VALUES(3, 2000, 'Shivani');
```

# Reading Data

```
SELECT FROM <tablename>
```

**Example:**

```
SELECT student_id, student_name
FROM student;
```

```
SELECT * FROM student WHERE
student_id=2;
```

# Updating Data

```
UPDATE <tablename>
SET <column name> = <new value>
<column name> = <value>....
WHERE <condition>

Update KeyspaceName.TableName
Set ColumnName1=new Column1Value,
      ColumnName2=new Column2Value,
      ColumnName3=new Column3Value,
       .
       .
       .
Where ColumnName=ColumnValue
```

- Example:

```
UPDATE student SET
student_fees=10000,student_name=
'Rahul'
WHERE student_id=2;
```

# Deleting Data

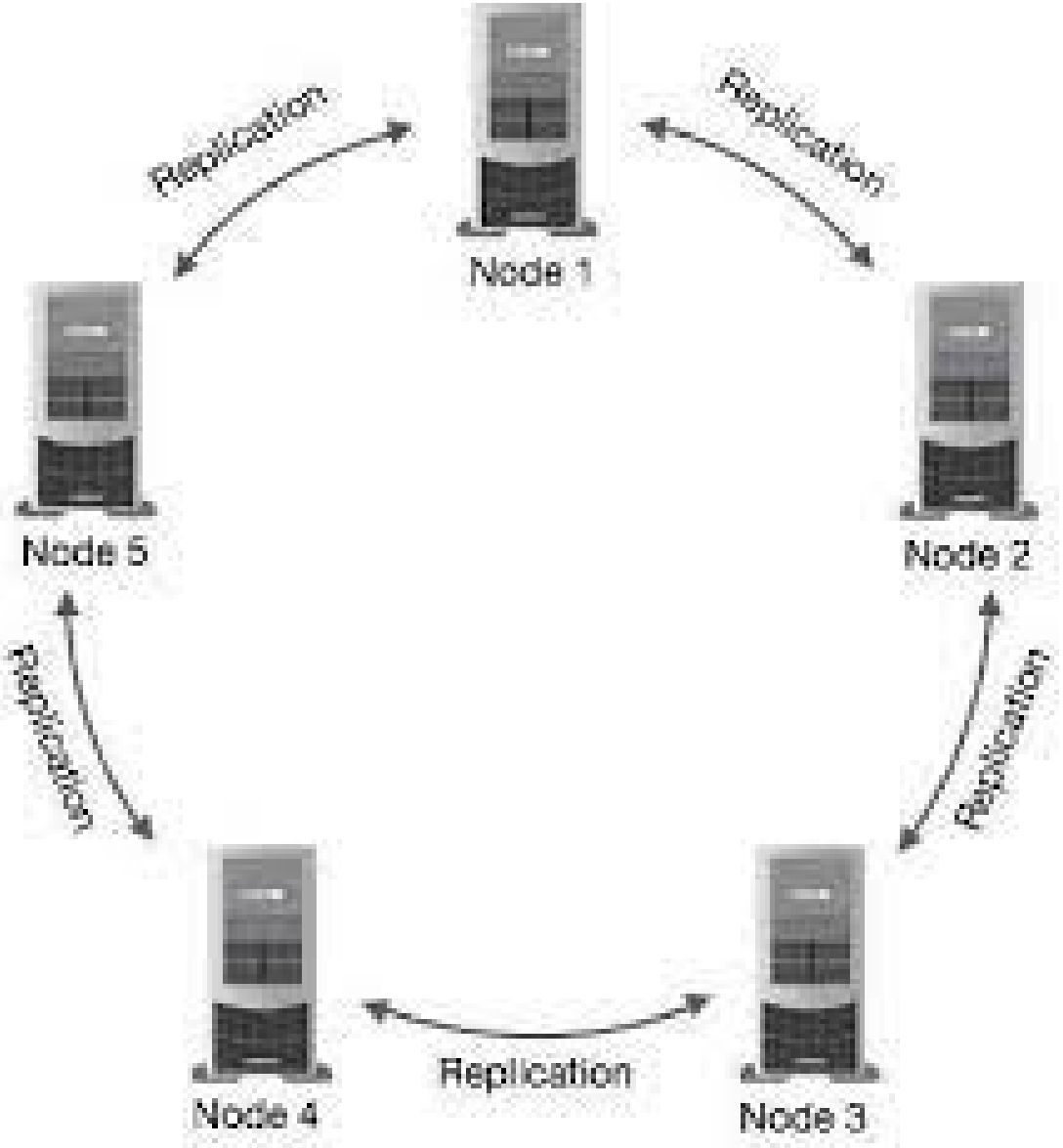
```
DELETE FROM <identifier> WHERE
<condition>;
```

**Example:**

```
DELETE FROM student WHERE
student_id=3;
```

# Cassandra Architecture

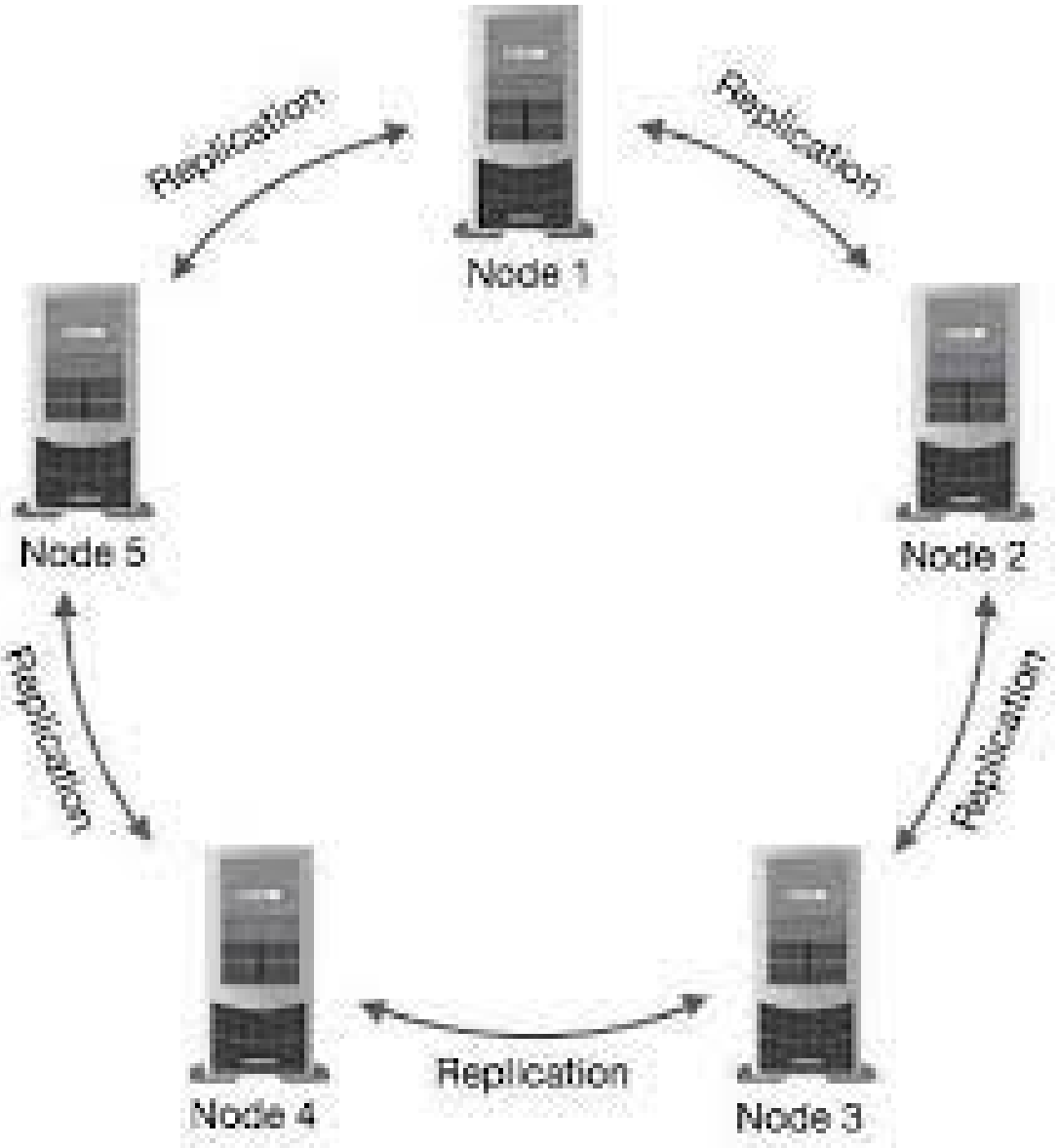
- Node:
  - A Cassandra node is a place where data is stored.
- Data centre:
  - Data centre is a collection of related nodes.
- Cluster:
  - A cluster is a component which contains one or more data centres.
- Commit log:
  - The commit log is a crash-recovery mechanism.
  - Every write operation is written to the commit log.



# Cassandra Architecture

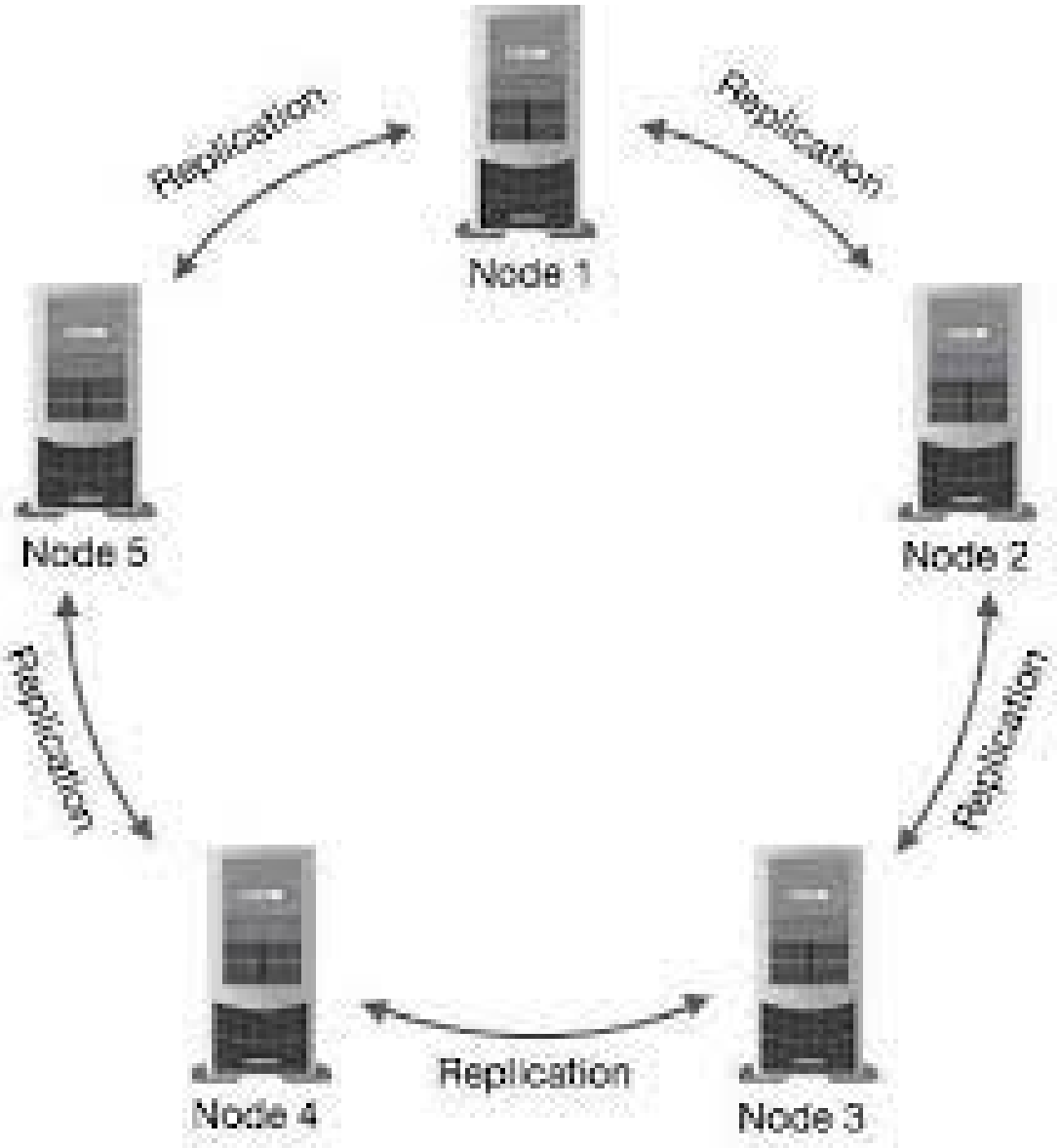
- Mem-table:
  - A mem-table is a memory-resident data structure.
  - After commit log, the data will be written to the mem-table.
  - Sometimes, for a single-column family, there will be multiple mem-tables.

- SSTable:
  - A disk file to which the data is flushed from the mem-table when its contents reach a threshold value.



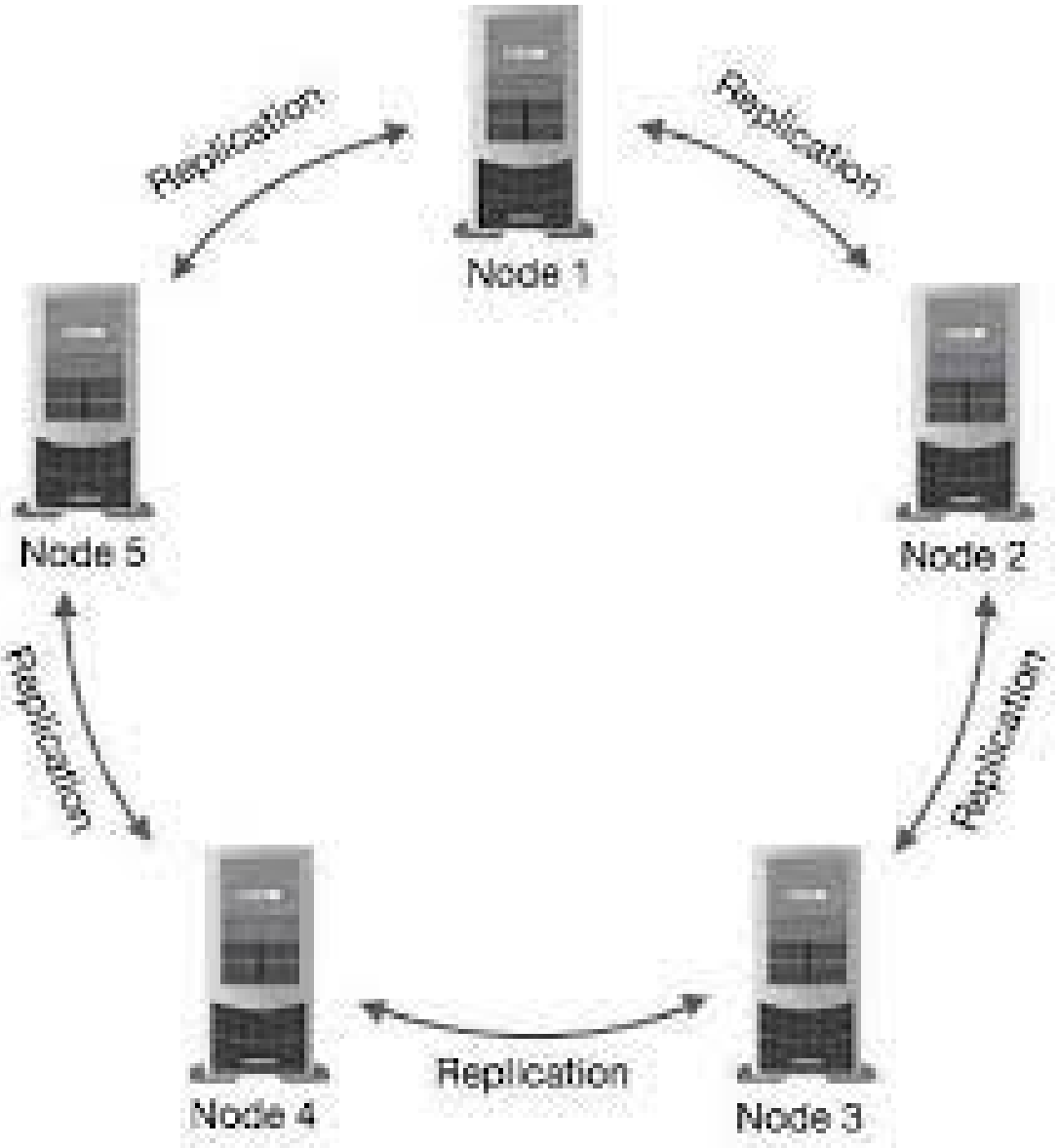
# Cassandra Architecture

- Bloom Filter:
  - Cassandra merges data on disk (in SSTables) with data in RAM (in memtables).
  - To avoid checking every SSTable data file for the partition being requested, Cassandra employs a data structure known as a bloom filter.



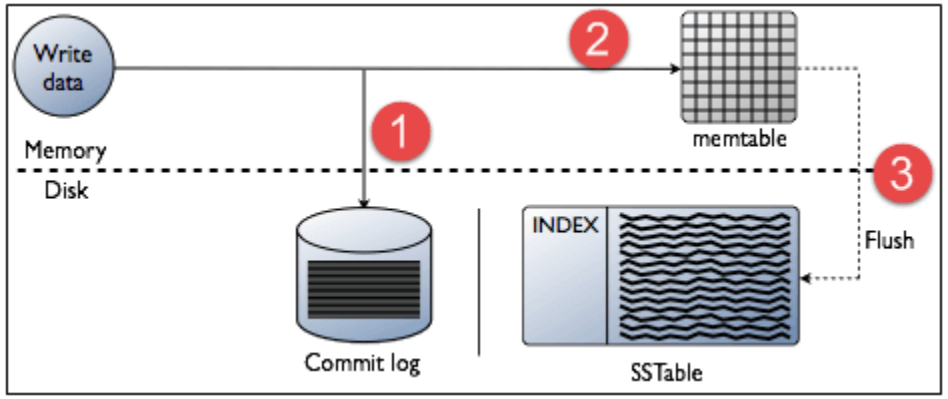
# Cassandra Architecture

- Bloom Filter:
  - A probabilistic data structure that allows Cassandra to determine one of two possible states: -
    - The data definitely does not exist in the given file,
    - or - The data probably exists in the given file.



# Write Operations

- Every write activity of nodes is captured by the commit logs written in the nodes.
- Later the data will be captured and stored in the mem-table.
- Whenever the mem-table is full, data will be written into the SStable data file.
- All writes are automatically partitioned and replicated throughout the cluster.
- Cassandra periodically consolidates the SSTables, discarding unnecessary data.



# Read Operations

- Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable which contains the required data.
- There are three types of read request that is sent to replicas by coordinators.
  - Direct request
  - Digest request
  - Read repair request

# Read Operations

- The coordinator sends direct request to one of the replicas.
- After that, the coordinator sends the digest request to the number of replicas specified by the consistency level and checks if the returned data is an updated data.
- After that, the coordinator sends digest request to all the remaining replicas.
- If any node gives out of date value, a background read repair request will update that data.
- This process is called read repair mechanism.