

NoSQL

Apache Cassandra
Continued...



Apache Cassandra



This Photo by Unknown Author is licensed under [CC BY-SA](#)

- **Distributed** NoSQL database management system
- Data model is based around optimizing for queries
- Stores data across a **cluster** of nodes
- CQL (Cassandra Query Language)

Apache Cassandra



This Photo by Unknown Author is licensed under [CC BY-SA](#)

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

Wide Column Store

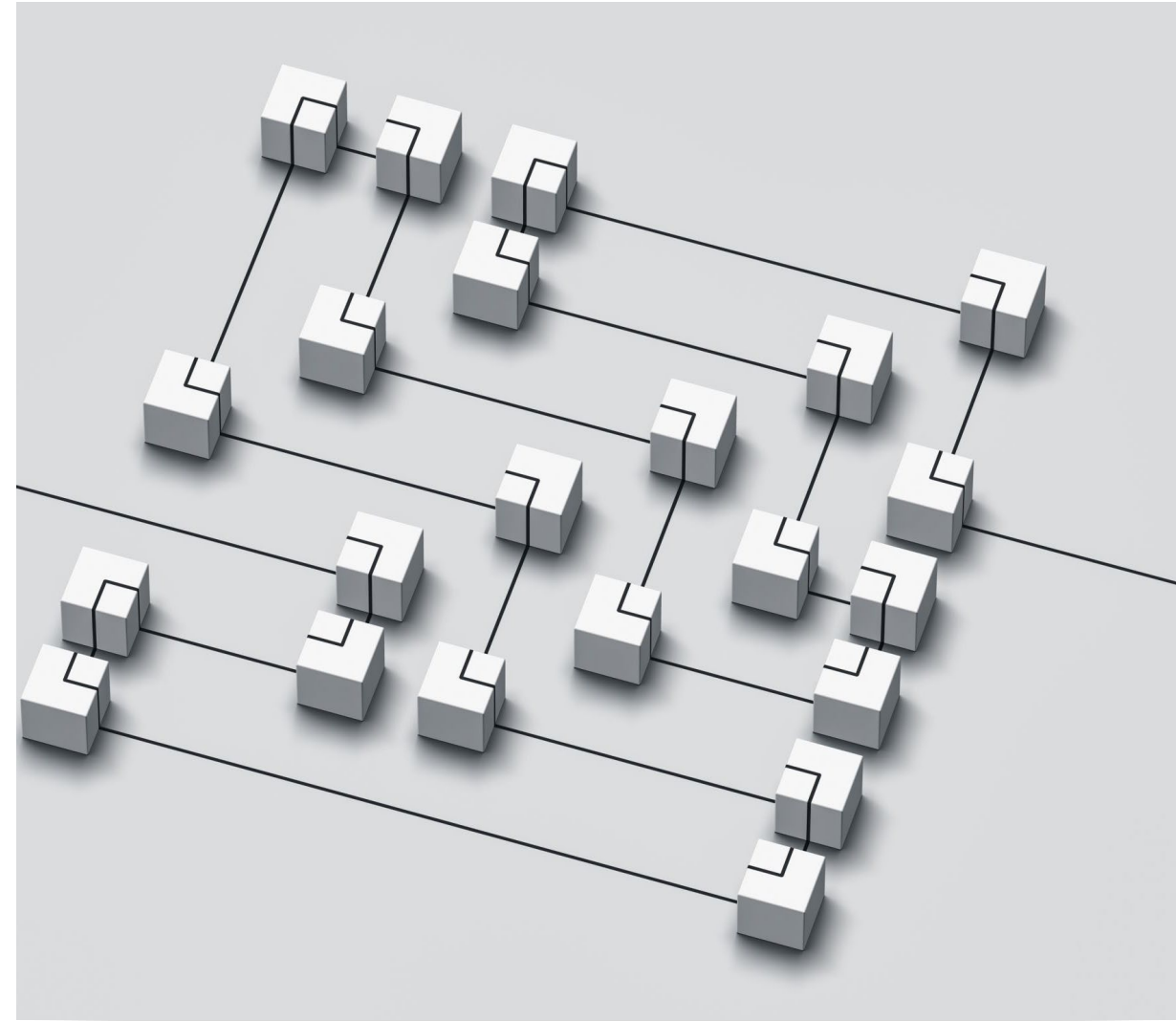
- Table = Column family
 - Table is a collection of similar rows (not necessarily identical)
- Row is a collection of columns
 - Should encompass a group of data that is accessed together
 - Associated with a unique row key
- Column
 - Column consists of a column name and column value
 - (and possibly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

Wide Column Stores

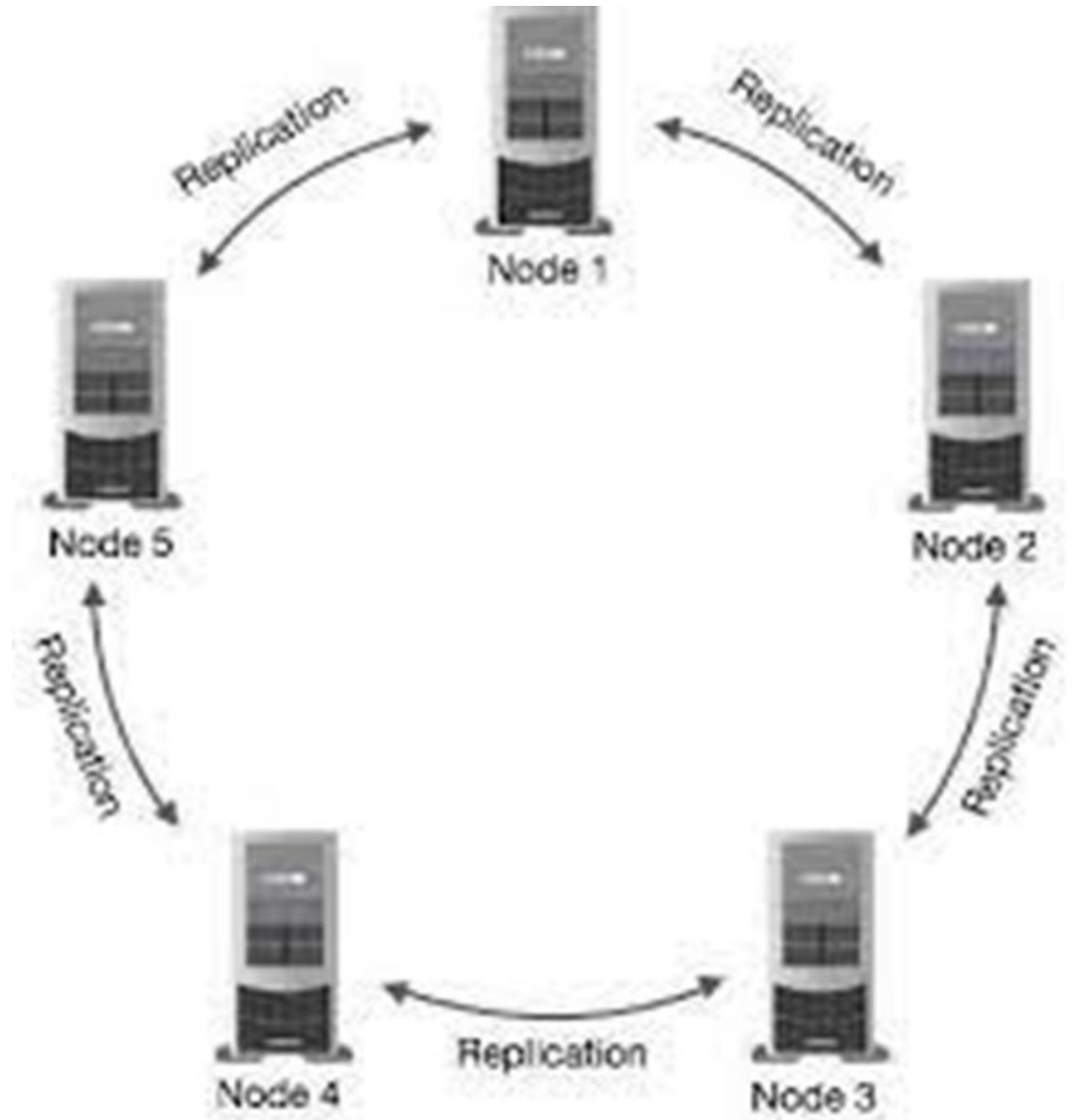
- Query patterns
 - Create, update or remove a row within a given column family
 - Select rows according to a row key or simple conditions
- Suitable use cases
 - Event logging, content management systems, blogs, ...
 - i.e. for structured flat data with similar schema
- When not to use
 - ACID transactions are required
 - Complex queries: aggregation (SUM, AVG, ...), joining, ...
 - Early prototypes: i.e. when database design may change

BASE

- **Basically Available**
 - 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.
- **Soft 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.
- **Eventually 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).



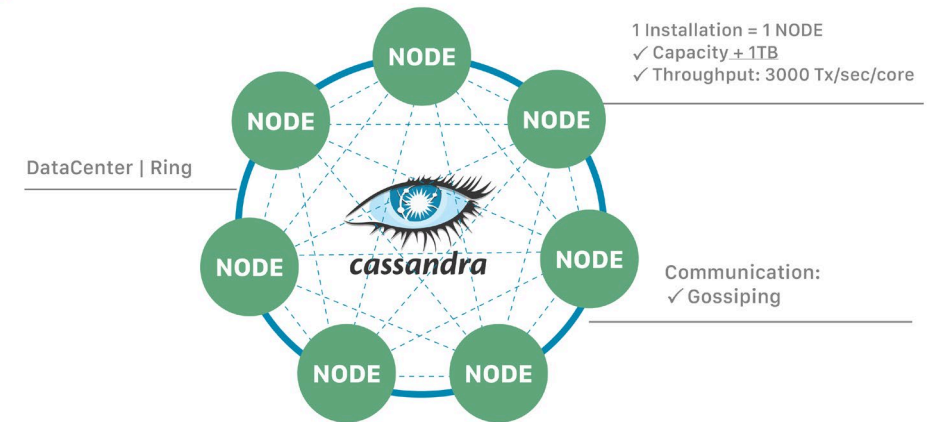
Cassandra Architecture

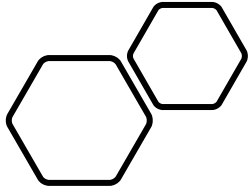


Apache Cassandra

- Uses a ring-type architecture
 - The smallest logical unit is a node.
- Uses the partitioning of data for optimizing queries.
 - Every piece of data has a partition key.
 - The partition key for every row is hashed.
 - This means we get a unique token for every piece of data.
 - For each node, there is an assigned range of tokens.
 - Therefore data with the same token is stored on the same node.

ApacheCassandra™ = NoSQL Distributed Database

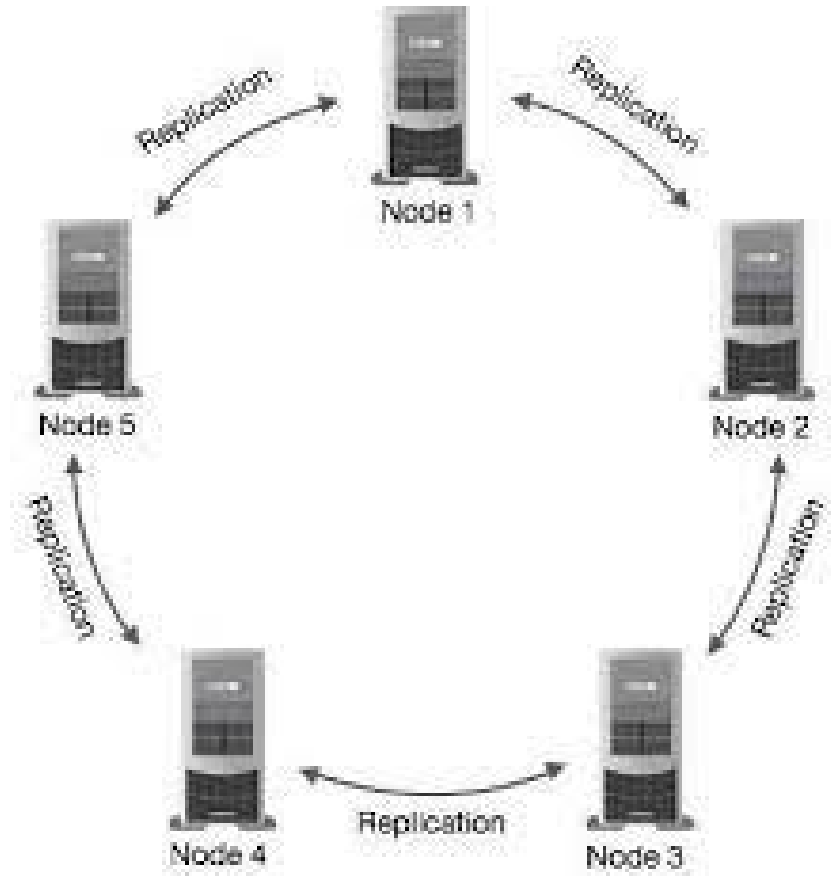




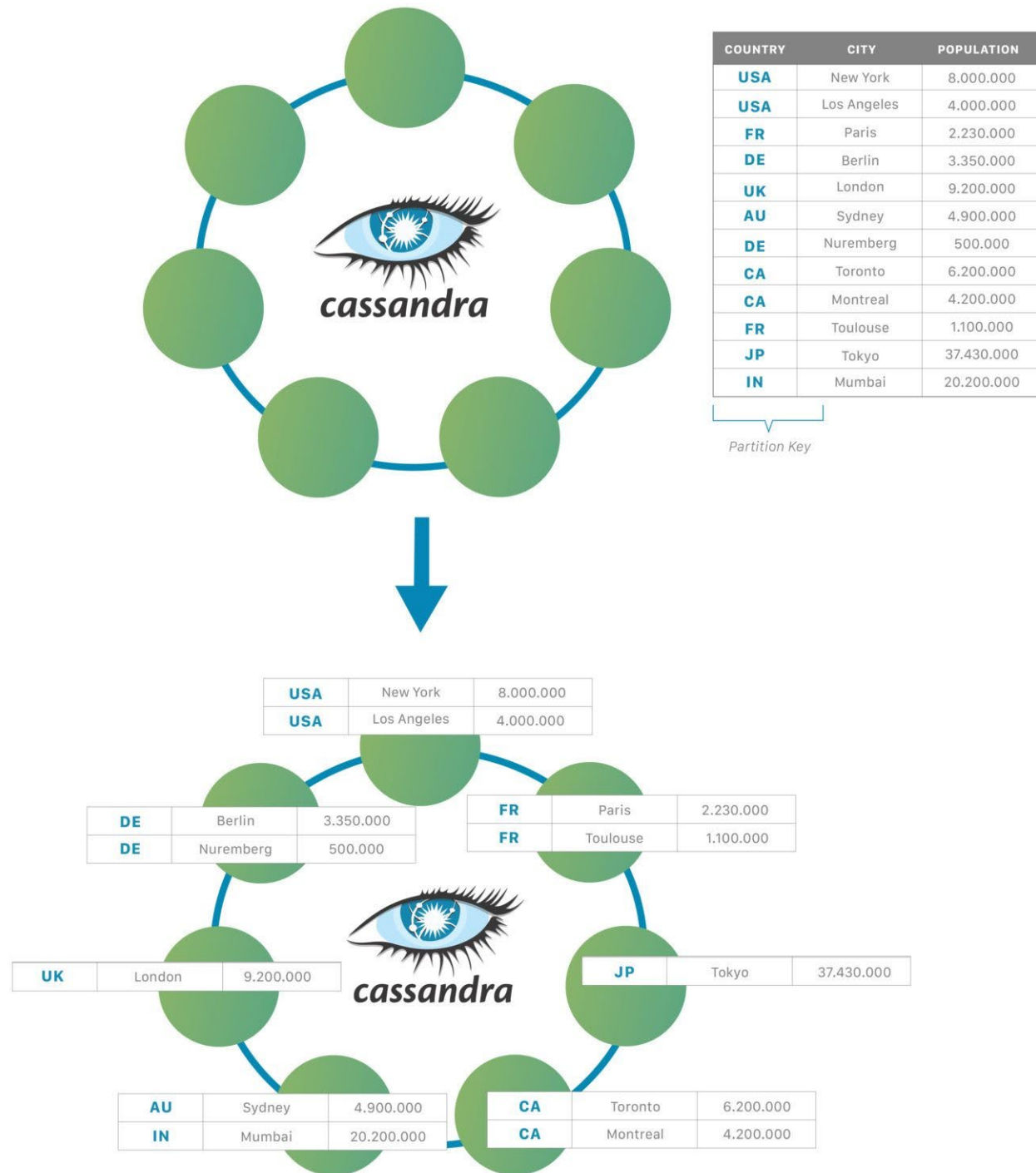
Cassandra Architecture

- NODE

- A Cassandra node is a place where data is stored.
- Each node is a fully functional machine
 - Can be a physical server or an EC2 instance, or a virtual machine.
- Each node is running a single instance of Cassandra.
- Each node connects with other nodes in the cluster through the high internal network.
- Nodes are organized with ring network topology
 - Every node is independent and has the same role in the ring.

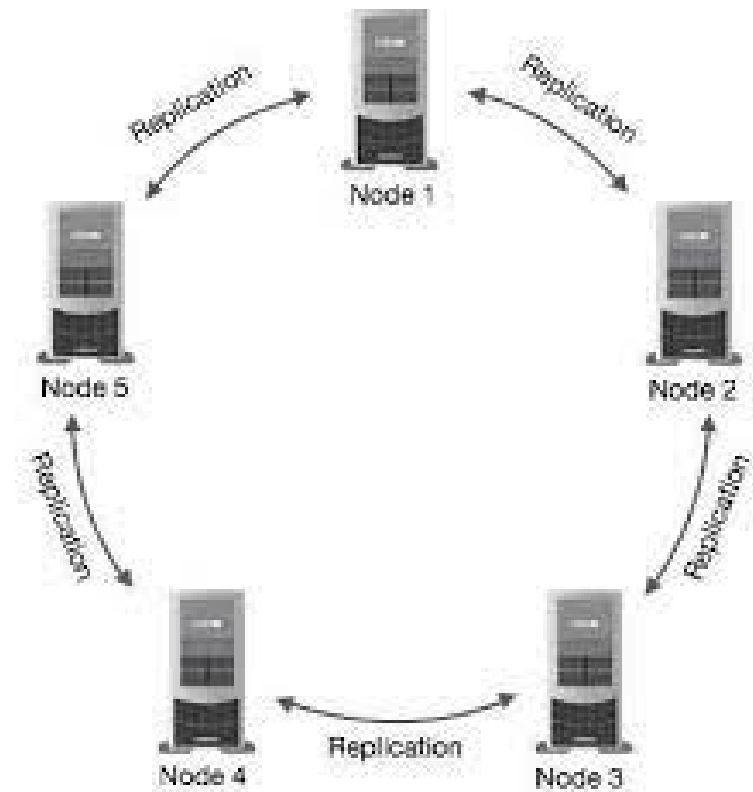


Cassandra



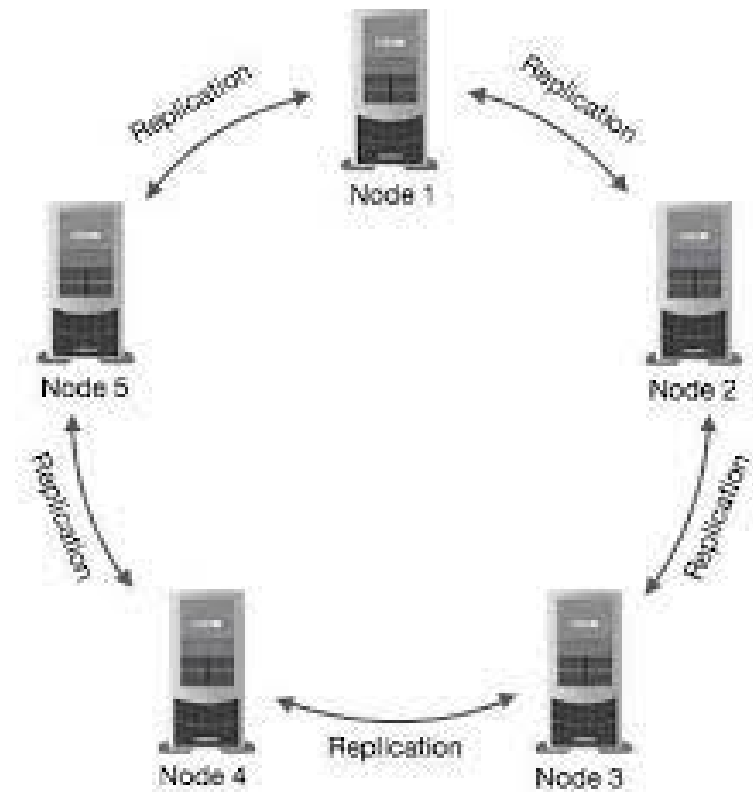
Cassandra Architecture - NODE

- Nodes are arranged in a peer-to-peer structure
- Each node in a cluster can accept read and write requests.
 - It doesn't matter where the data is actually located in the cluster – each read will always get the newest version of data.



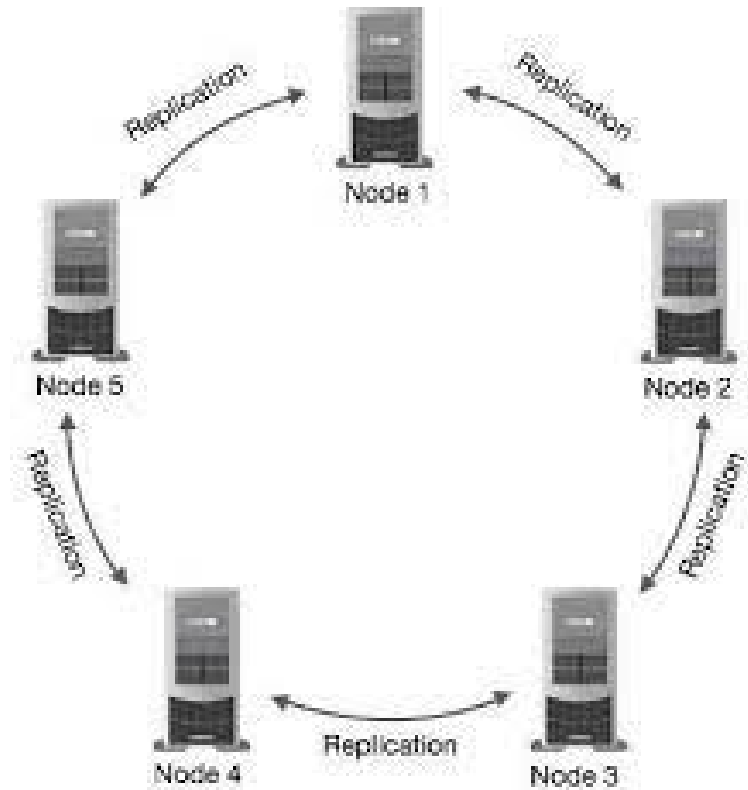
Cassandra Architecture - SERVER

- Machine on which Cassandra is installed
- Each node is therefore technically a server



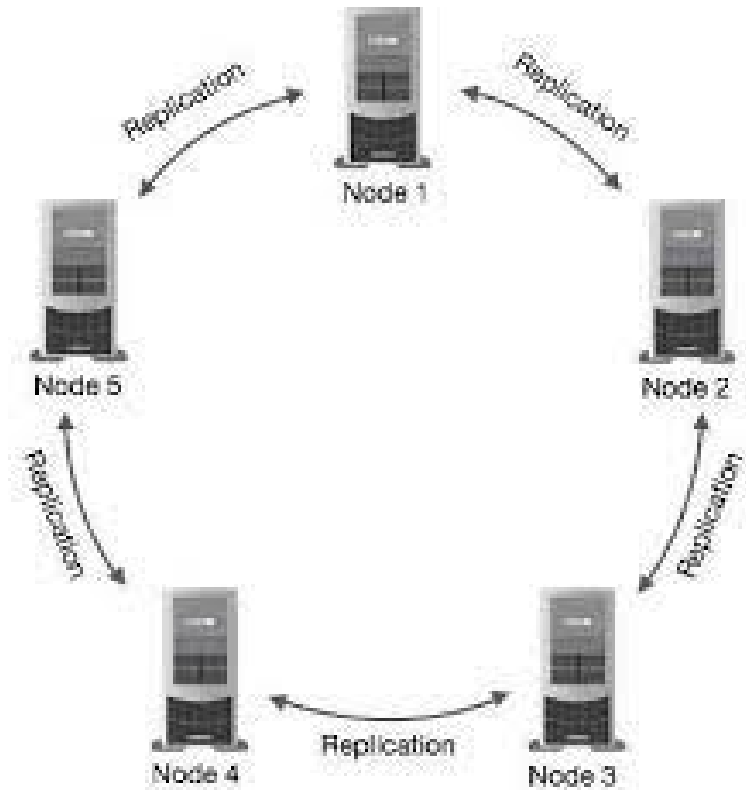
Cassandra Architecture - RACK

- Logical grouping of nodes within the ring
 - A rack is a collection of servers
- Used to ensure replicas are distributed among different logical groupings
- A rack can send operations not only to just one node.
- Multiple nodes, each on a separate rack, can provide greater fault tolerance and availability



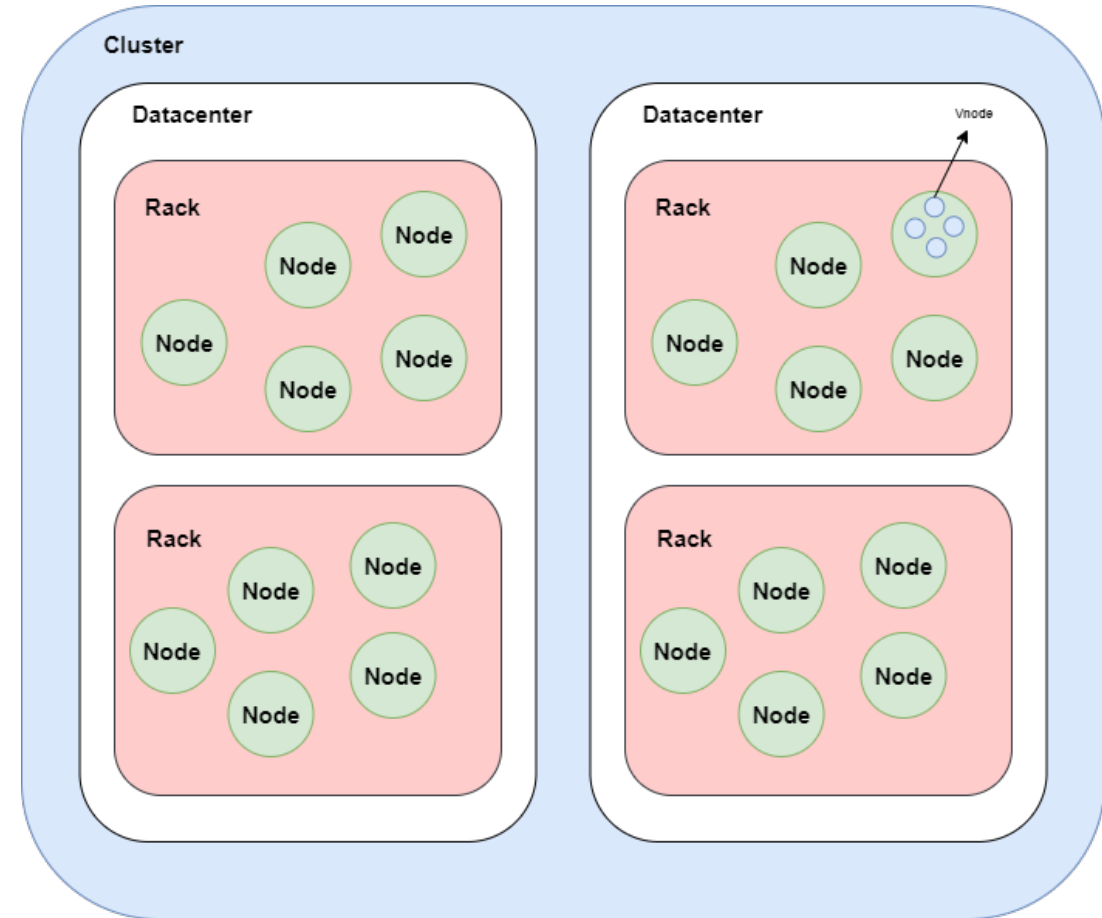
Cassandra Architecture - DATACENTER

- Logical set of racks
 - Should contain at least one rack.
- Can be described as a group of nodes related and configured within a cluster for replication purposes.
- Helps to reduce latency, prevent transactions from impact by other workloads and related effects.
- The replication factor can also be set up to write to multiple data centers.
 - This allows greater flexibility in architectural design and organization



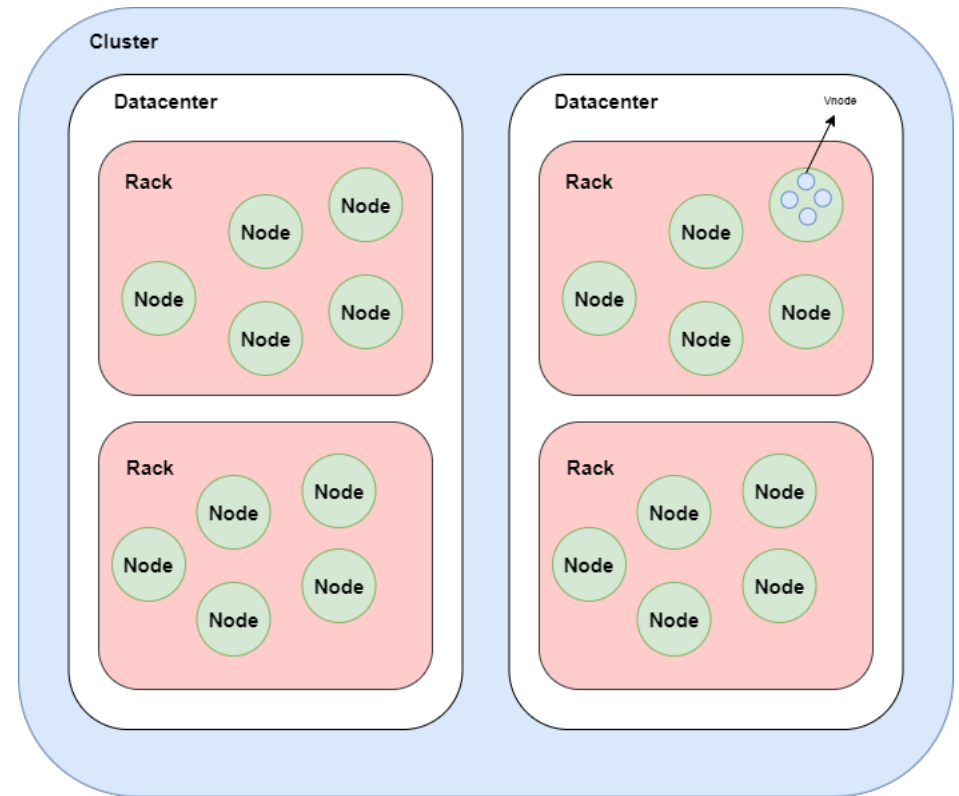
Cassandra Architecture - CLUSTER

- Contains one or more data centers
- Outermost container in the data base



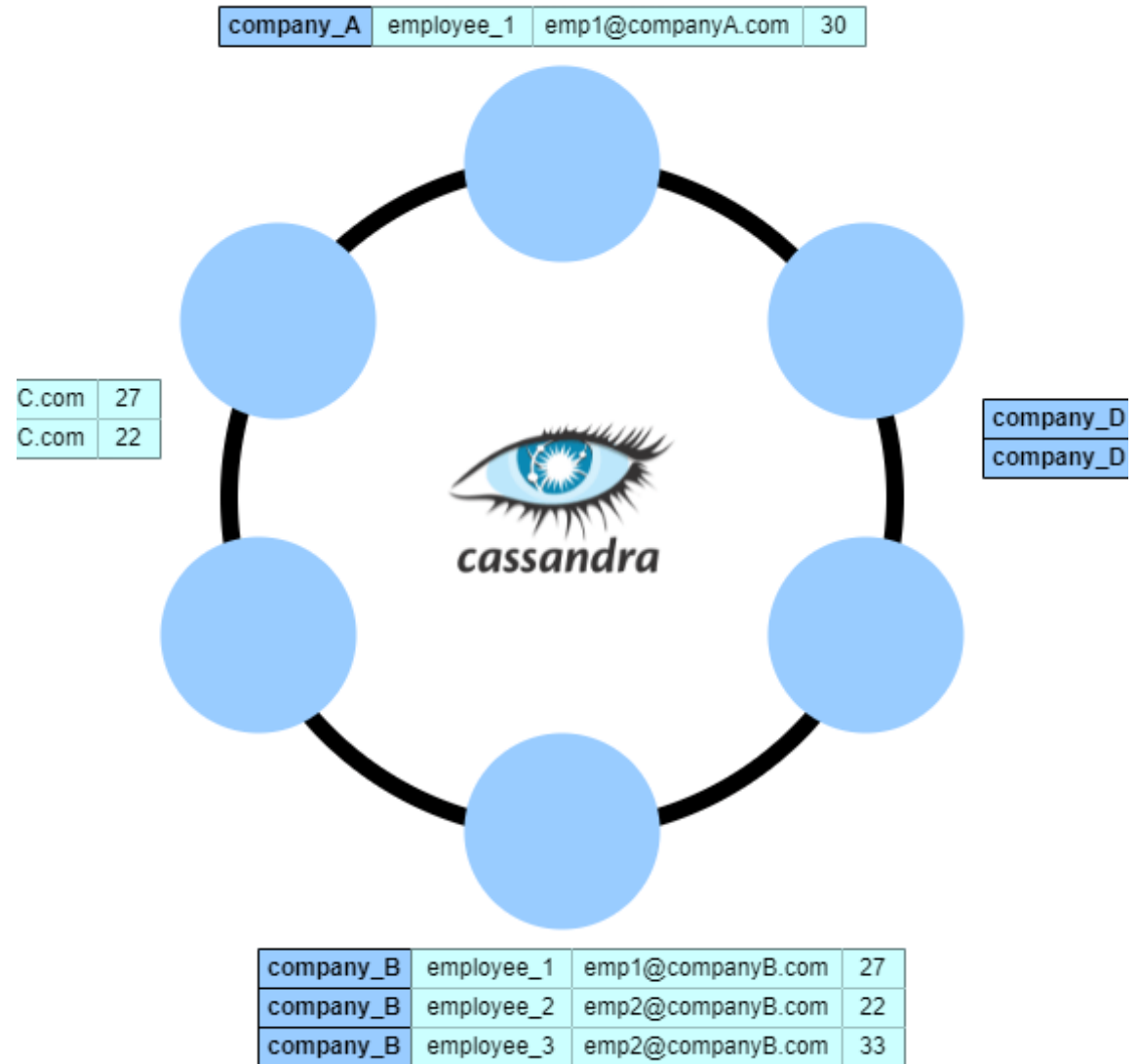
Cassandra Architecture

- A **database** contains one or more **clusters**
- A **cluster** contains one or more **data centers**
- A **data center** contains one or more **racks**
- A **rack** contains one or more **nodes**
- Each **node** is running an **instance of Cassandra**



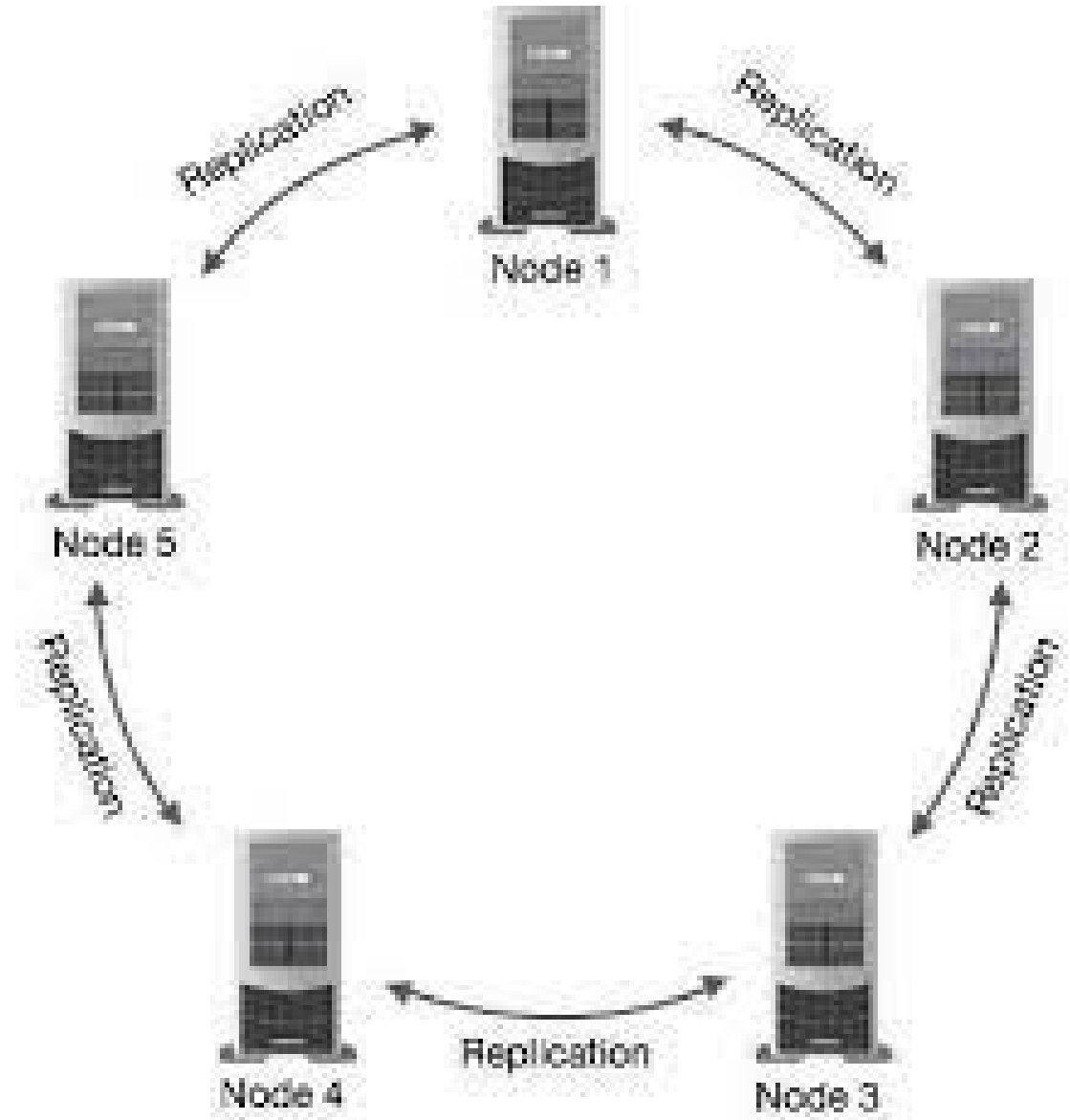
Cassandra Architecture

- Cassandra offers linear scalability and performance directly proportional to the number of nodes available.



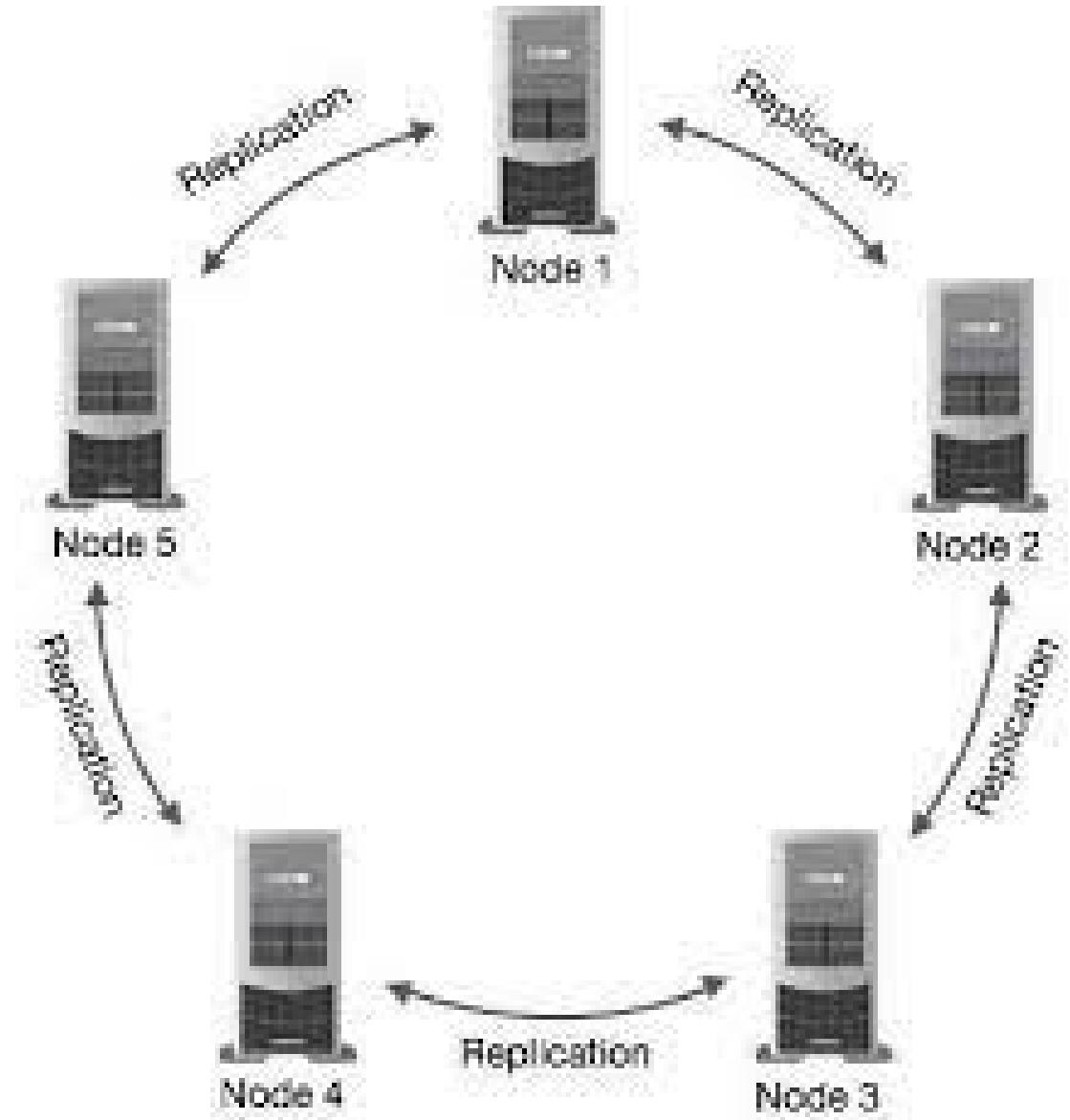
Cassandra Architecture

- Commit log:
 - The commit log is a crash-recovery mechanism.
 - Every write operation is written to the commit log.



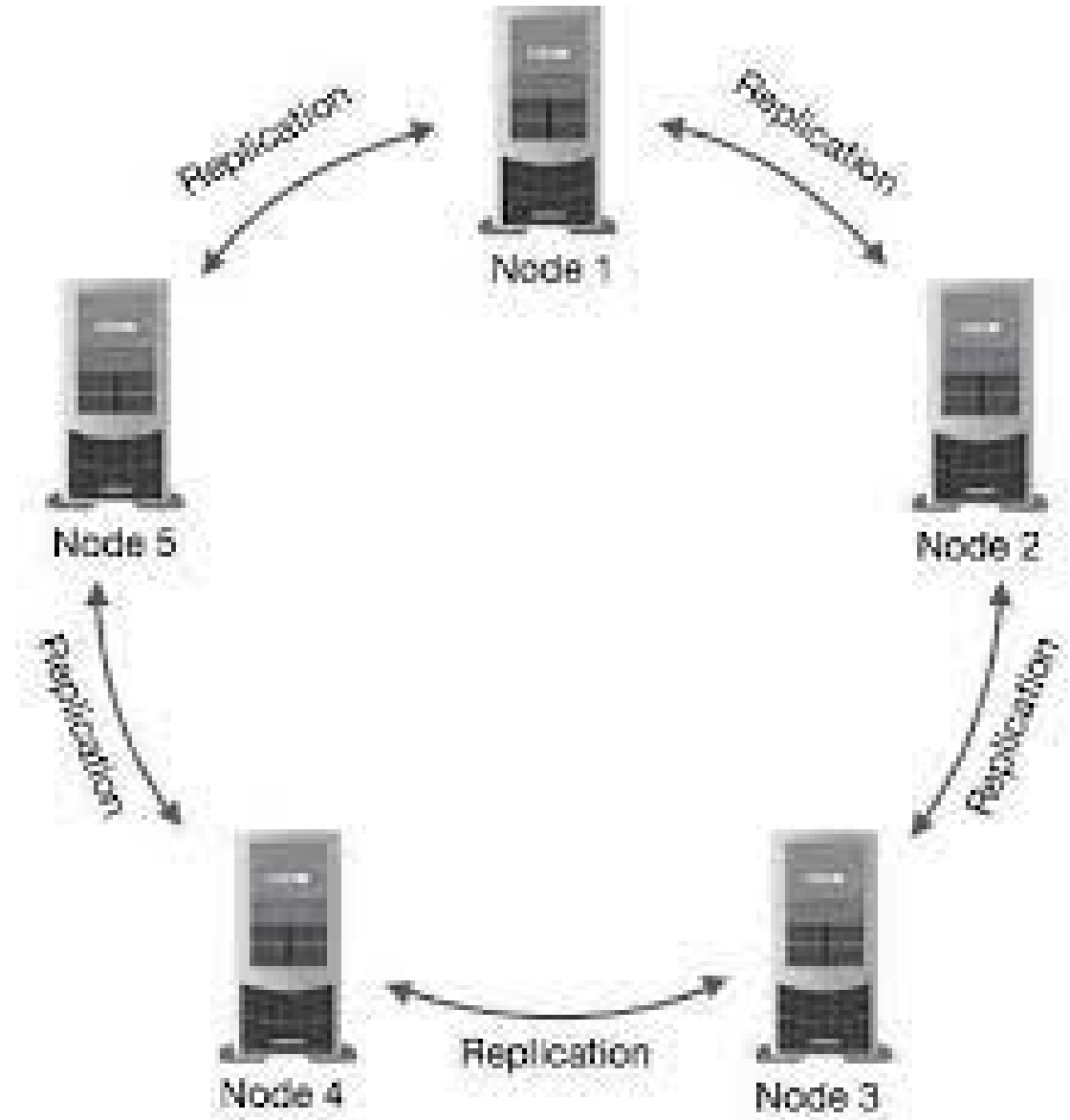
Cassandra Architecture

- 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.



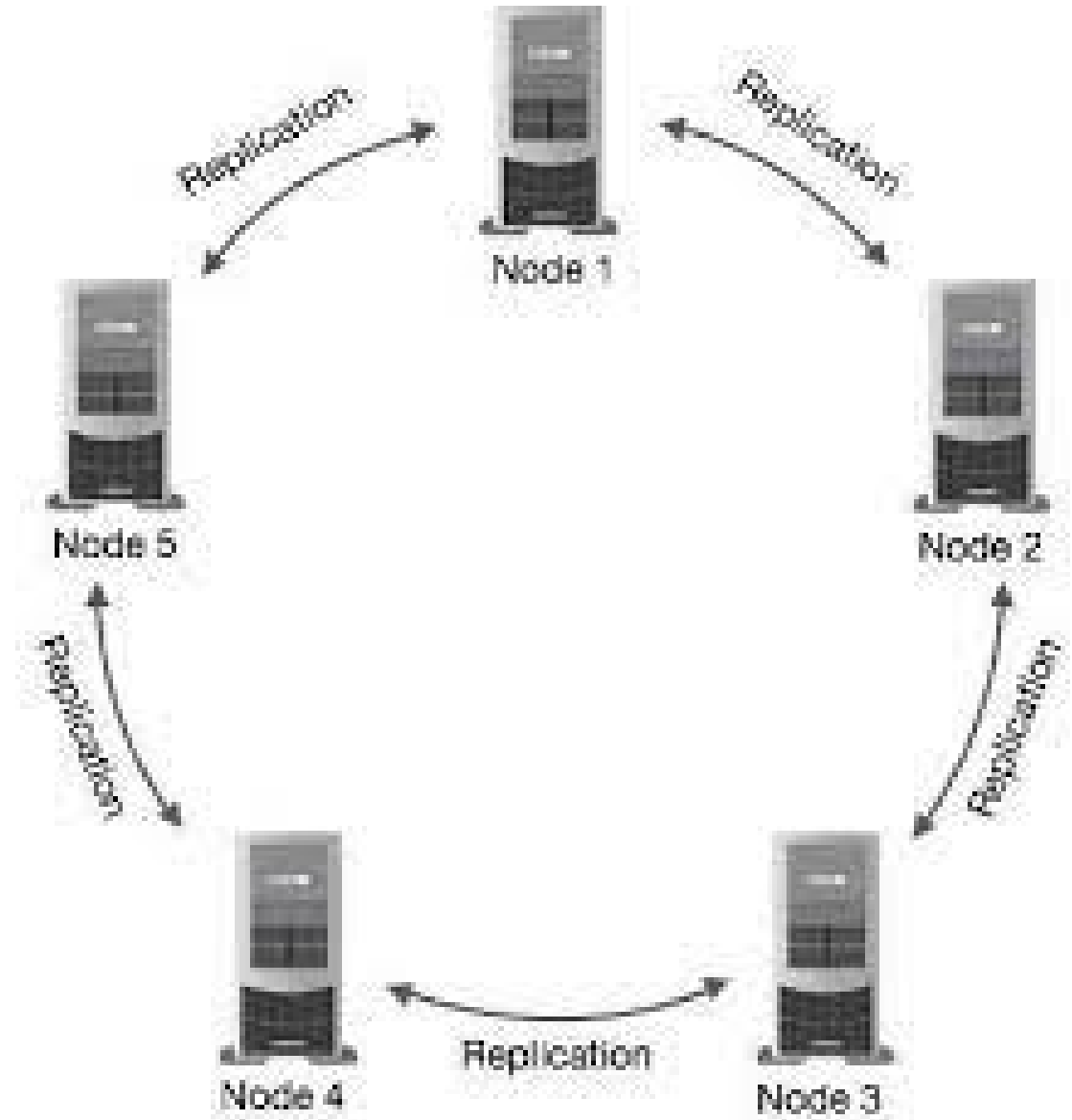
Cassandra Architecture

- 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.



Replication and Partitioning



Replication

- Some systems can not allow for data loss or interruption in data delivery.
 - For example, in the case of hardware problems, or links can be down at any time during the data process.
- The solution is to provide a backup when the problem has occurred.
- Cassandra stores data replicas on multiple nodes to ensure reliability and fault tolerance.

Replication Factor

- Determines the number of replicas and their location by the replication factor and replication strategy.
- The replication factor is the total number of replicas across the cluster.
 - A replication factor of N will store each data row on N different nodes in the cluster.
 - If we set this factor to one, it means that only one copy of each row exists in a cluster and so on.
- We can set this factor on the datacenter level and on rack level.

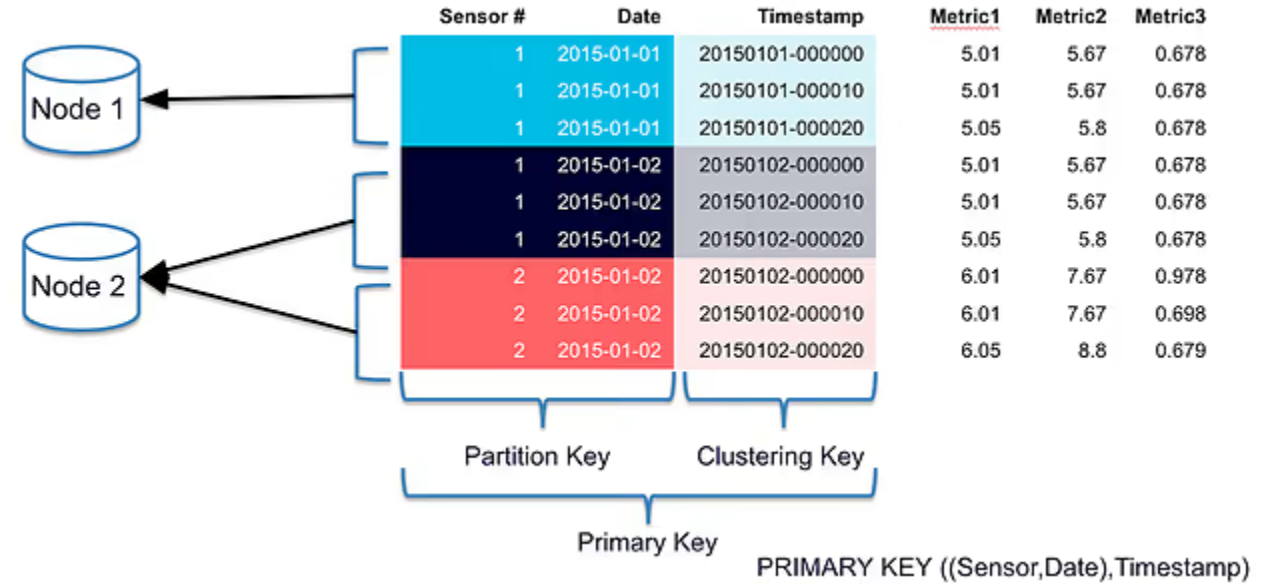
Replication Strategy

- Controls how the replicas are chosen.
- SimpleStrategy
 - Unaware of the logical division of nodes for data centers and racks.
 - 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
- NetworkTopologyStrategy
 - More complicated
 - Both racks aware and data center aware
 - Allows us to define how many replicas would be placed in different datacenters
 - Also tries to avoid situations when two replicas are placed on the same rack.

Partitioning

- Stores data with tunable consistency in partitions across a cluster
- Each partition represents a set of rows
- The partitioning algorithm is set at cluster level
- Partition key is set at table level

Partitioning



Partitioning

- A primary key represents a unique data partition and data arrangement within a partition.
 - The optional clustering columns handle the data arrangement part.
- A unique partition key represents a set of rows in a table which are managed within a server (including all servers managing its replicas).
- Primary Key = Partition Key + [Clustering Columns]

Partitioning

```
CREATE TABLE server_logs (  
    log_hour timestamp,  
    log_level text,  
    message text,  
    server text,  
    PRIMARY KEY (log_hour, log_level)  
)
```

Partition key: log_hour

Clustering column: log_level

Partitions created: all rows sharing a log_hour are a single partition

Partitioning

```
CREATE TABLE server_logs(  
    log_hour timestamp,  
    log_level text,  
    message text,  
    server text,  
    PRIMARY KEY ((log_hour, server))  
)
```

Partition key: log_hour and server

Clustering column: none

Partitions created: all rows sharing a log_hour for a particular server are a single partition

Partitioning

```
CREATE TABLE server_logs(  
    log_hour timestamp,  
    log_level text,  
    message text,  
    server text,  
    PRIMARY KEY ((log_hour,  
server), log_level)  
    )WITH CLUSTERING ORDER BY (column3  
DESC) ;
```

Partition key: log_hour and server

Clustering column: log_level

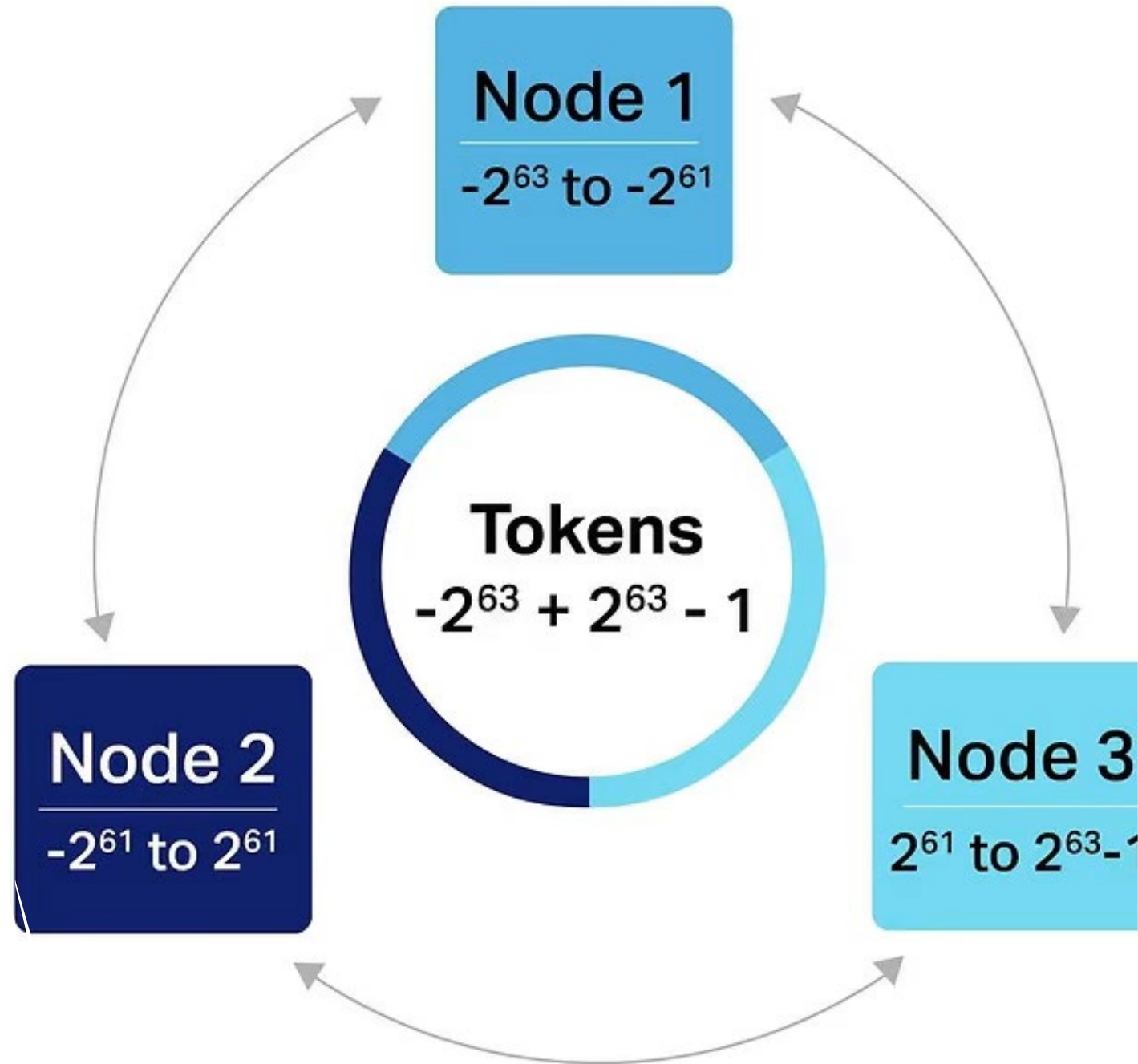
Partitions created: all rows sharing a log_hour for a particular server are in a single partition, rows are arranged in descending order of 'log_level' within the partition

Partitioning

- Read and write operations are performed using a partition key on a table.
- Cassandra uses 'tokens' (a long value out of range -2^{63} to $+2^{63} - 1$) for data distribution and indexing.
- The tokens are mapped to the partition keys using a 'partitioner'.
- Cassandra uses a partition key to determine which node store data on and where to find data when it's needed

Partitioner

- The partitioner applies a partitioning function to convert any given partition key to a token.
- Each node in a Cassandra cluster owns a set of data partitions using this token mechanism.
- The data is then indexed on each node with the help of the partition key.



Partitioning

- A partition key for a table should be designed to satisfy its access pattern and with the ideal amount of data to fit into partitions.
- A partition key should not allow 'unbounded partitions'.
 - An unbounded partition grows indefinitely in size as time passes.
 - In the server_logs table example, if the server column is used as a partition key it will create unbounded partitions as logs for a server will increase with time.
 - The time attribute of log_hour puts a bound on each partition to accommodate an hour worth of data

Partitioning

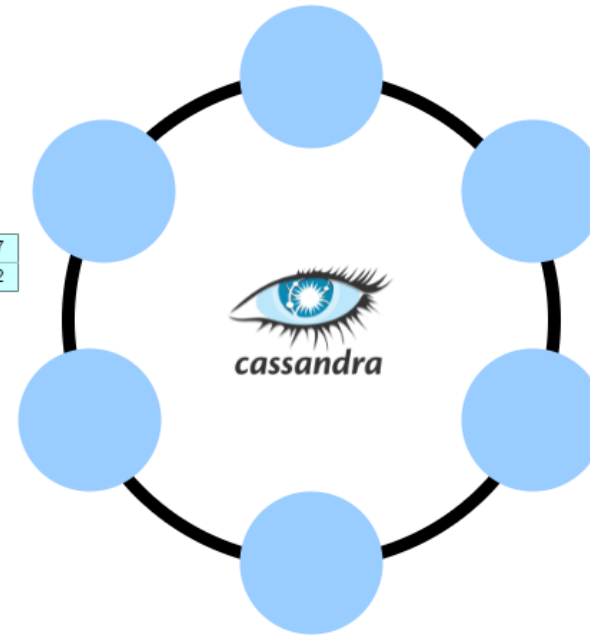
- A partition key should not create partition skew, in order to avoid uneven partitions and hotspots.
 - A partition skew is a condition in which there is more data assigned to a partition as compared to other partitions and the partition grows indefinitely over time.
 - In the `server_logs` table example, if the partition key is `server` and if one server generates way more logs than other servers, it will create a skew.
 - Can be avoided by introducing some other attribute from the table in the partition key so that all partitions get even data.
 - If you can't use a real attribute to remove skew create a dummy column and include in to the partition key.
 - This dummy column will then distinguish partitions and it can be controlled from an application without disturbing the data semantics.

Example

- The company table is split into partitions using the partition key `company_name` and distributed across the nodes.
- Notice that Cassandra groups the rows with the same `company_name` value and stores them on the same physical partition on the disk.
- Therefore, we can read all the data for a given company with minimal I/O cost.

company_C	employee_1	emp1@companyC.com	27
company_C	employee_2	emp2@companyC.com	22

company_A	employee_1	emp1@companyA.com	30
-----------	------------	-------------------	----



company_D	employee_1	emp1@companyD.com	60
company_D	employee_2	emp2@companyD.com	48

company_B	employee_1	emp1@companyB.com	27
company_B	employee_2	emp2@companyB.com	22
company_B	employee_3	emp2@companyB.com	33

```
CREATE TABLE company (  
    company_name text,  
    employee_name text,  
    employee_email text,  
    employee_age int,  
    PRIMARY KEY ((company_name), employee_email)  
);
```

Data Latency

- Data is distributed across nodes
 - These may be on different machines in different geo locations
- It takes time to retrieve the data
- Creates latency
 - Time lag between us (or, more often, our applications and databases making that request) asking for data and when we're actually able to get it.
- This is something we need to work on to try to improve performance

Why partition?

- Improve scalability.
 - When you scale up a single database system, it will eventually reach a physical hardware limit
 - If you divide data across multiple partitions, each hosted on a separate server, you can scale out the system almost indefinitely.
- Improve performance.
 - Data access operations on each partition take place over a smaller volume of data.
 - Partitioning can make your system more efficient.
 - Operations that affect more than one partition can run in parallel.
- Improve security.
 - In some cases, you can separate sensitive and nonsensitive data into different partitions and apply different security controls to the sensitive data.



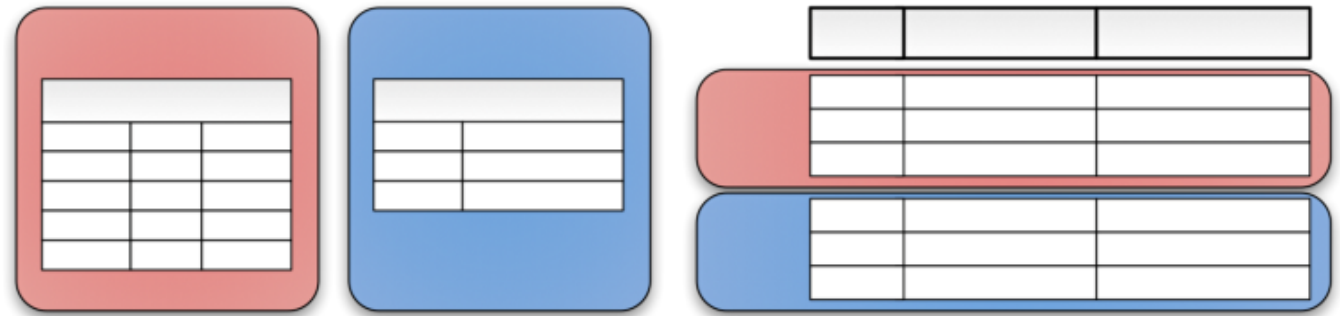
Why partition?

- Provide operational flexibility.
 - You can define different strategies for management, monitoring, backup and restore, and other administrative tasks based on the importance of the data in each partition.
- Improve availability.
 - Separating data across multiple servers avoids a single point of failure.
 - If one instance fails, only the data in that partition is unavailable - operations on other partitions can continue



Vertical v Horizontal Partitioning

- Vertical Partitioning stores tables and/or columns in a separate database or tables.
- Horizontal Partitioning (sharding) stores rows of a table in multiple database clusters.
- When do we decide ?
 - Physical Design

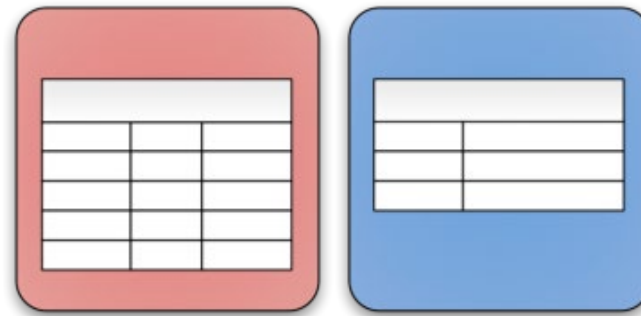


Vertical

Horizontal

Vertical v Horizontal Partitioning

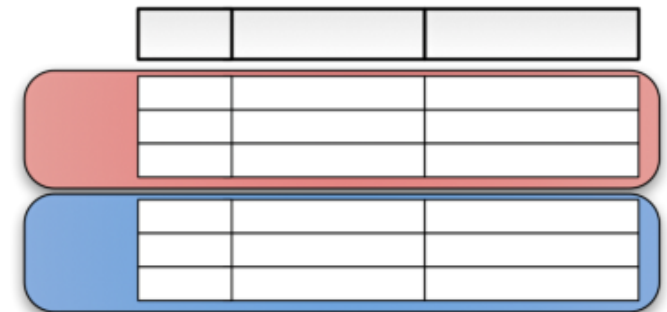
- Vertical partitioning allows a table to be partitioned into disjoint sets of columns.
- Each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use.
 - For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.
- Can significantly impact the performance of the workload i.e., queries and updates, by reducing cost of accessing and processing data.



Vertical

Vertical v Horizontal Partitioning

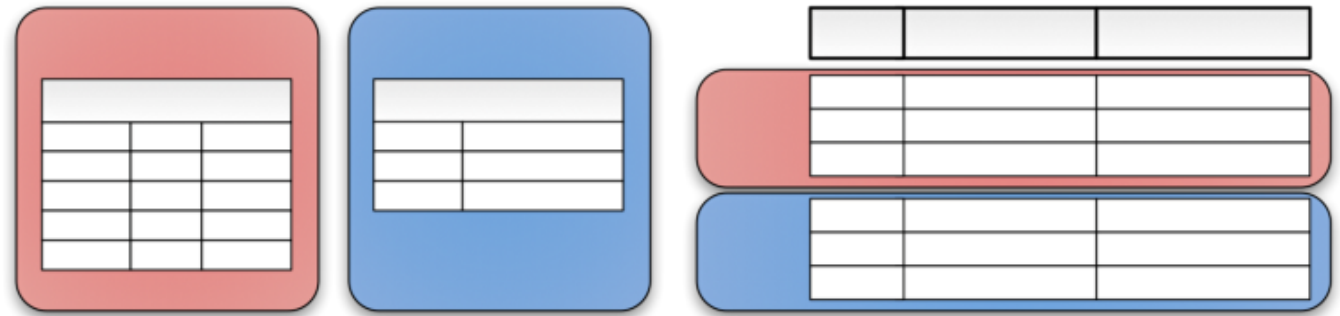
- Horizontal partitioning
- Each partition is a separate data store, but all partitions have the same schema.
- Each partition is known as a shard and holds a specific subset of the data, such as all the orders for a specific set of customers.
- Allows tables, indexes and materialized views to be partitioned into disjoint sets of rows that are physically stored and accessed separately.
- Can significantly impact the performance of the workload i.e., queries and updates, by reducing cost of accessing and processing data.



Horizontal

Vertical v Horizontal Partitioning

- Can be done together
- You might divide data into shards and then use vertical partitioning to further subdivide the data in each shard

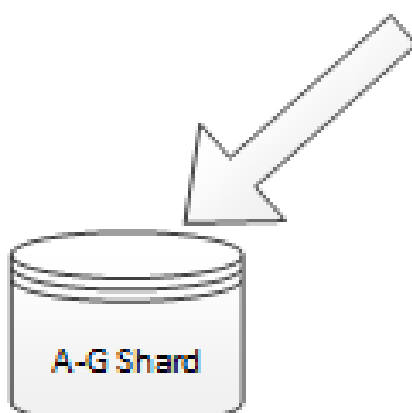


Vertical

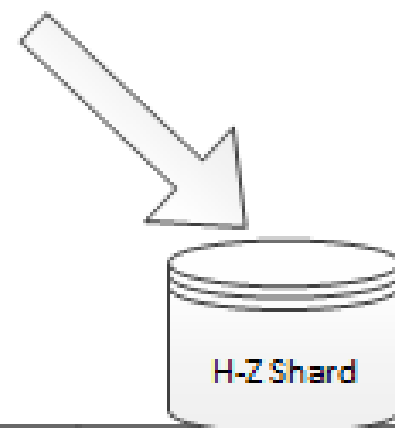
Horizontal

Horizontal Partitioning

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013



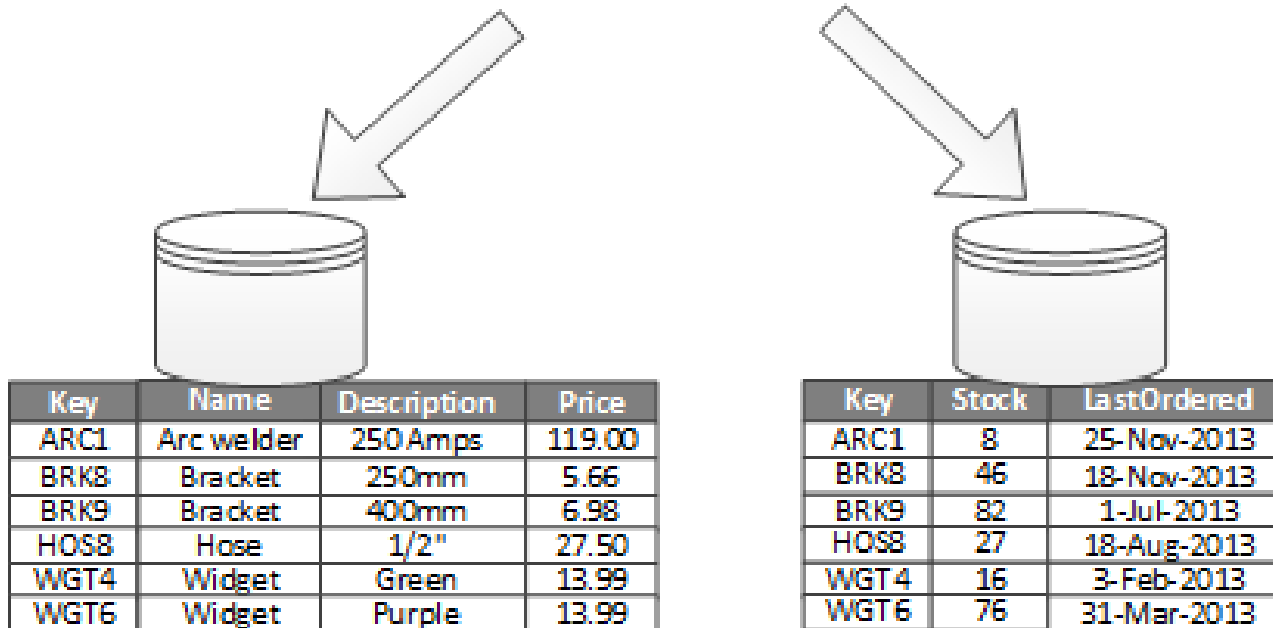
Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013



Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Vertical Partitioning

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013





Partitioning and Replication

- Combining partitioning and replication is common pattern
 - Splitting the data into separate partitions and then creating multiple replicas for each partition.
 - The replicas for each partition coordinate with each other
- Improves fault tolerance
- Improves scalability
- Improves availability

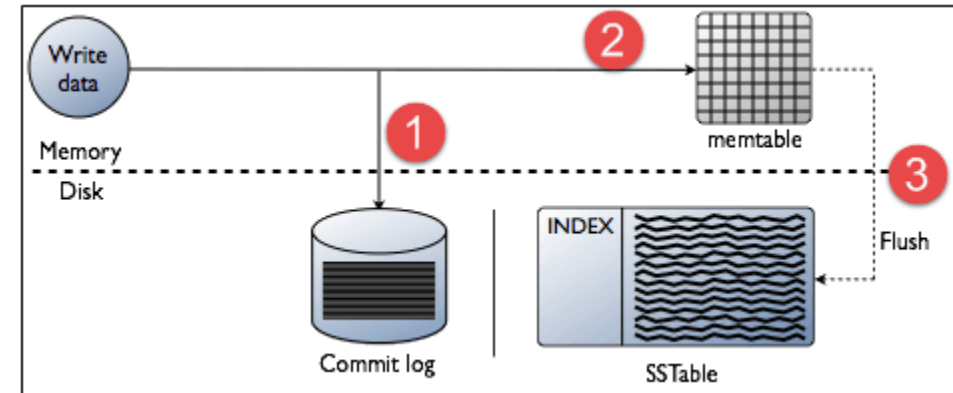
Cassandra Read and Write

Write

- Starts with the immediate logging of a write
- Ending with a write of data to disk
- Stages:
 - Logging data in the commit log
 - Writing data to the memtable
 - Flushing data from the memtable
 - Storing data on disk in SSTables

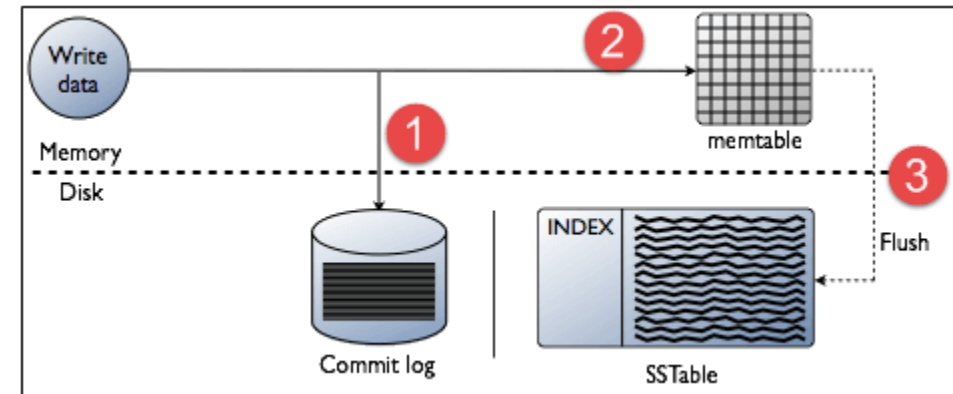
Write Operations

- Every write activity of nodes is captured by the commit logs written in the nodes.
- Data will also be captured and stored in the mem-table.



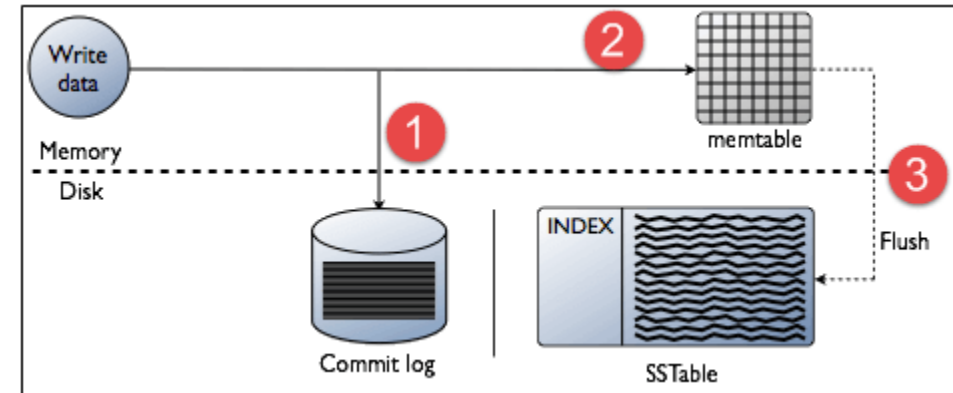
Write Operations

- All writes in Cassandra are **durable**
- All writes to a replica node are recorded both in memory and in a commit log on disk before they are acknowledged as a success.
- Writes to the commit log survive permanently even if power fails on a node.



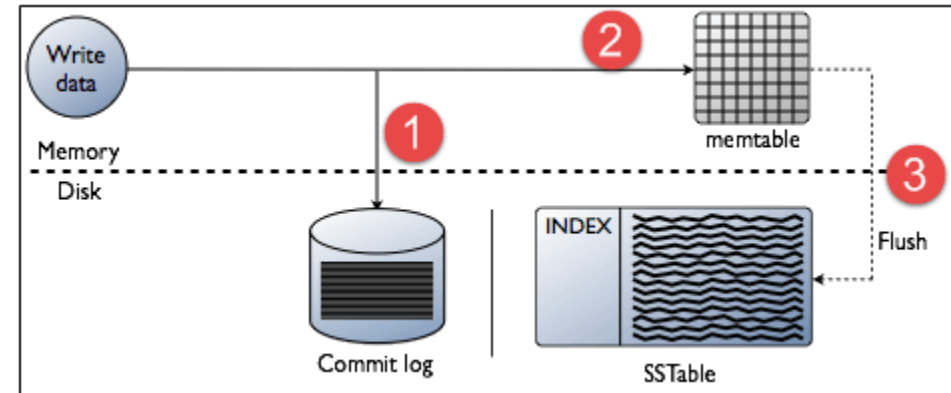
Write Operations

- Write durability can be controlled when you set up a keyspace
- You can instruct Cassandra whether or not to use the commit log
 - The default is that write durability is set to TRUE



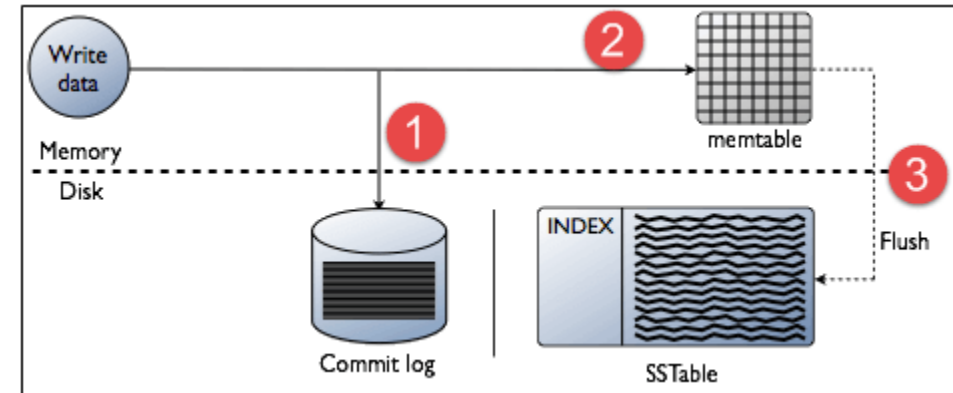
Write Operations

- When the memtable is full Cassandra writes the data to disk in SSTables, in the memtable-sorted order.
- A partition index is also created on the disk that maps the tokens to a location on disk.
- Data in the commit log is purged after its corresponding data in the memtable is flushed to an SSTable on disk



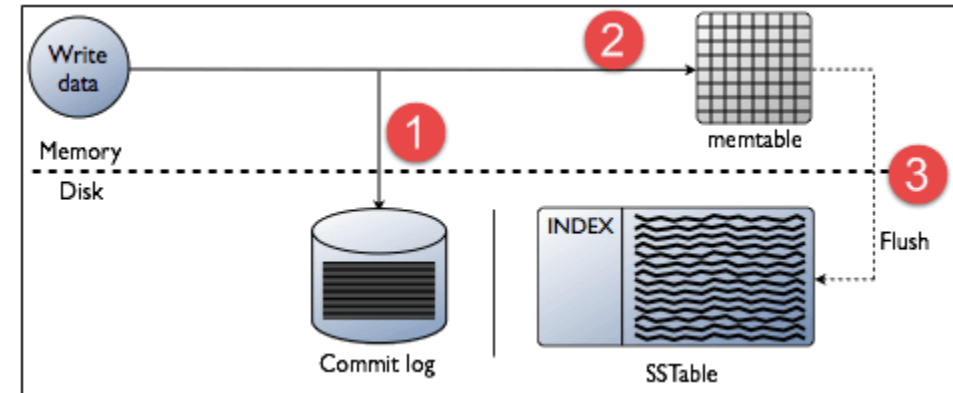
Write Operations

- When the memtable content exceeds the configurable threshold or the commitlog space exceeds its total allocation, the memtable is put in a queue that is flushed to disk.
- If the data to be flushed exceeds the a set threshold, Cassandra blocks writes until the next flush succeeds to ensure not data is lost.



Write Operations

- Memtables and SSTables are maintained per table.
- The commit log is shared among tables.
- SSTables are immutable, not written to again after the memtable is flushed.
- Therefore a partition is typically stored across multiple SSTable files



Write Operations

For each SSTable Cassandra creates:

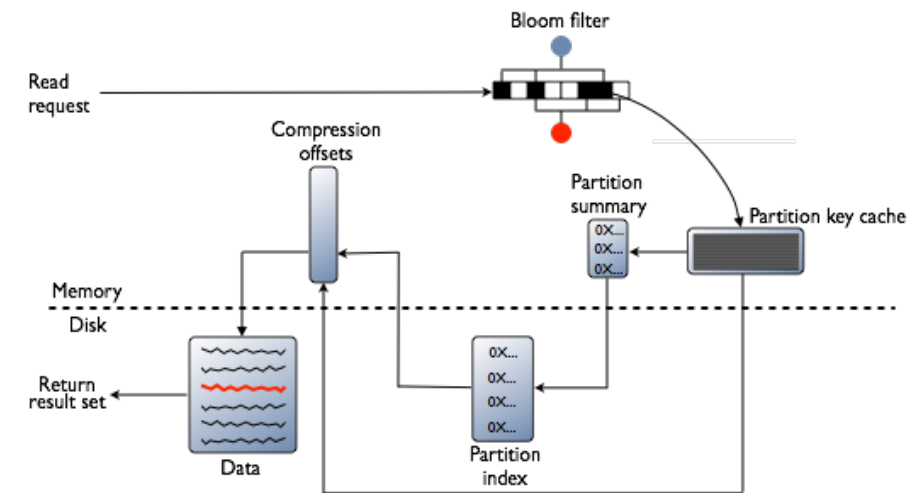
- Data (Data.db)
 - the SSTable data
- Primary Index (Index.db)
 - Index of the row keys with pointers to their positions in the data file
- Bloom filter (Filter.db)
 - A structure stored in memory that checks if row data exists in the memtable before accessing SSTables on disk
- SSTable Index Summary (SUMMARY.db)
 - A sample of the partition index stored in memory
- SSTable Table of Contents (TOC.txt)
 - A file that stores the list of all components for the SSTable TOC
- Secondary Index (SI_*.db)
 - Built-in secondary index. Multiple SIs may exist per SSTable
- Compression Information (CompressionInfo.db)
 - A file holding information about uncompressed data length, chunk offsets and other compression information
- Statistics (Statistics.db)
 - Statistical metadata about the content of the SSTable
- Digest (Digest.crc32, Digest.adler32, Digest.sha1)
 - A file holding adler32 checksum of the data file – used to verify the integrity of data during transmission
- CRC (CRC.db)
 - A file holding the CRC32 for chunks in an uncompressed file.

SSTables

- Data files are stored in a data directory set up at install
- For each keyspace, a directory within the data directory stores each table.
 - E.g. `/data/data/ks1/cf1-5be396077b811e3a3ab9dc4b9ac088d/la-1-big-Data.db` represents a data file
 - `ks1` represents the keyspace name to distinguish the keyspace for streaming or bulk loading data.
 - A hexadecimal string, `5be396077b811e3a3ab9dc4b9ac088d`, is appended to table names to represent unique table IDs.

Read

- Cassandra must combine results from the active memtable and potentially multiple SSTables.
- Process starts with memtables and ends with SSTables:
 - Check the memtable
 - Check row cache, if enabled
 - Check Bloom filter
 - Check partition key cache, if enabled
 - if a partition key is found in the partition key cache
 - Goes directly to the compression offset map
 - If not checks the partition summary
 - The partition index is accessed
 - Data on disk is located using the compression offset map
 - Data is fetched from the SSTable on disk

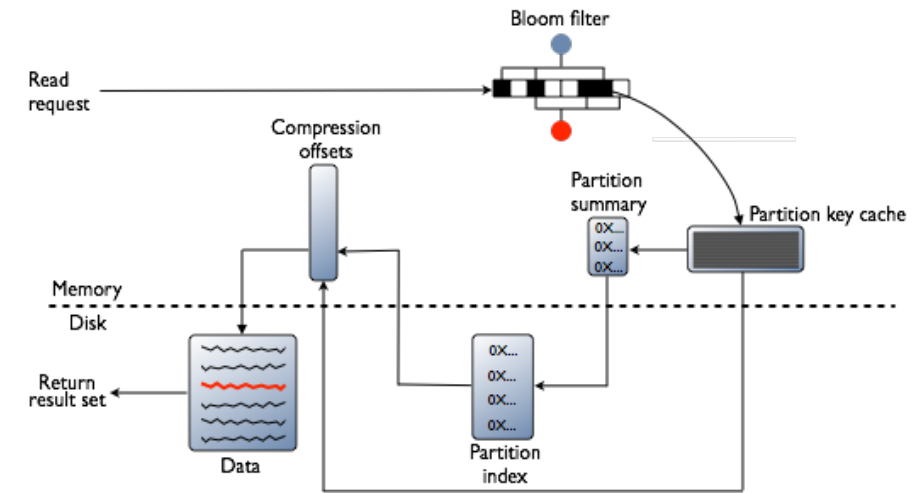


Read

- Partition Key Cache
 - Stores a cache of the partition index in off-heap memory.
 - Uses a small, configurable amount of memory, and each "hit" saves one seek during the read operation.
 - If a partition key is found in the key cache can go directly to the compression offset map to find the compressed block on disk that has the data.
- Partition Summary
 - Stores a sampling of the partition index.
 - A partition index contains all partition keys, whereas a partition summary samples every X keys, and maps the location of every Xth key's location in the index file.
 - After finding the range of possible partition key values, the partition index is searched
- Partition Index Cache
 - Stores an index of all partition keys mapped to their offset
 - If the partition summary has been checked for a range of partition keys, the search passes to the partition index to seek the location of the desired partition key.
 - A single seek and sequential read of the columns over the passed-in range is performed.
 - Using the information found, the partition index now goes to the compression offset map to find the compressed block on disk that has the data.
- Compression Offset Map
 - Stores pointers to the exact location on disk that the desired partition data will be found
 - The desired compressed partition data is fetched from the correct SSTable(s) once the compression offset map identifies the disk location.

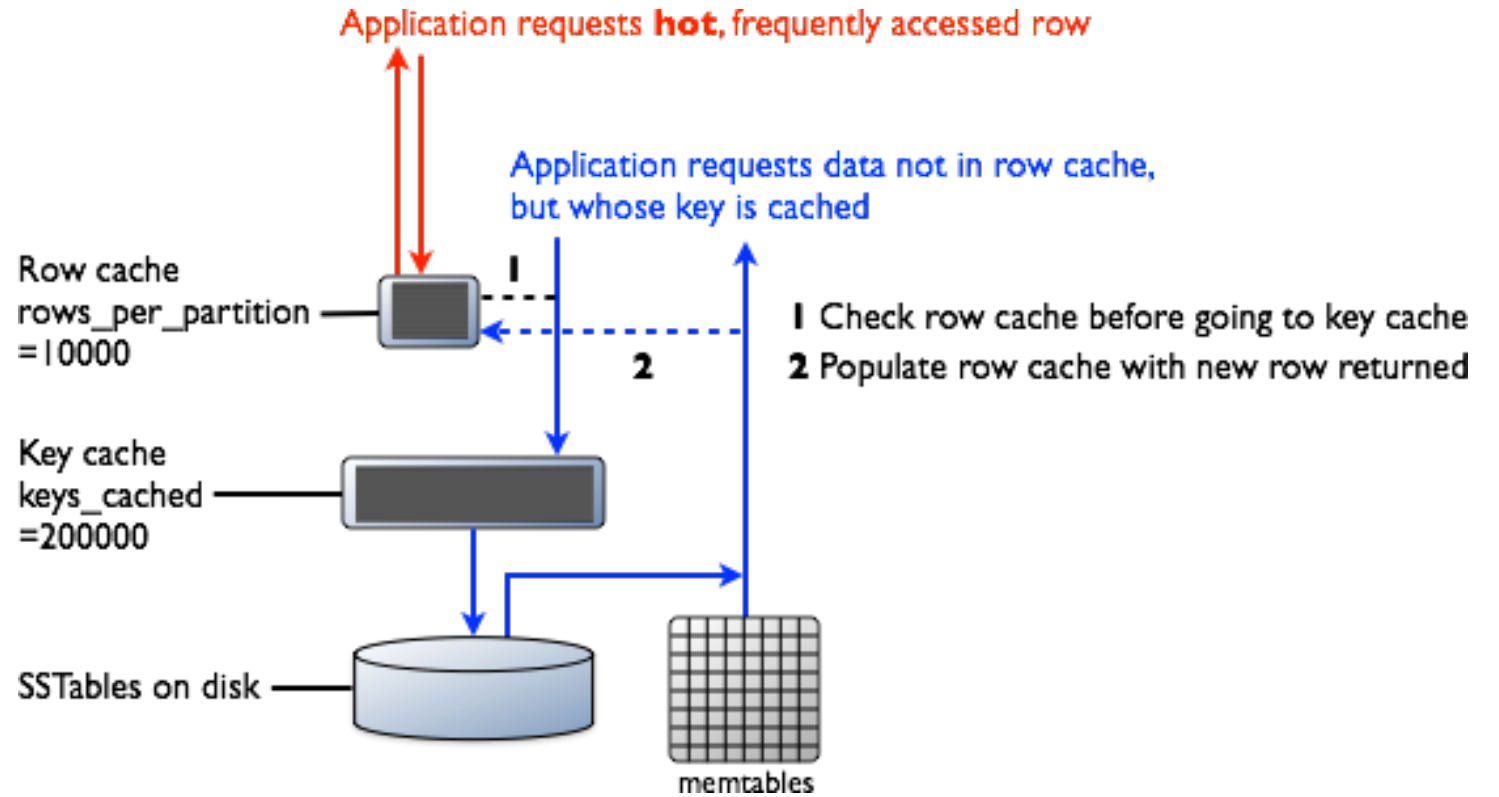
Read

- If the memtable has the desired partition data, then the data is read and then merged with the data from the SSTables.
- Reads are fastest when the most in-demand data fits into memory.



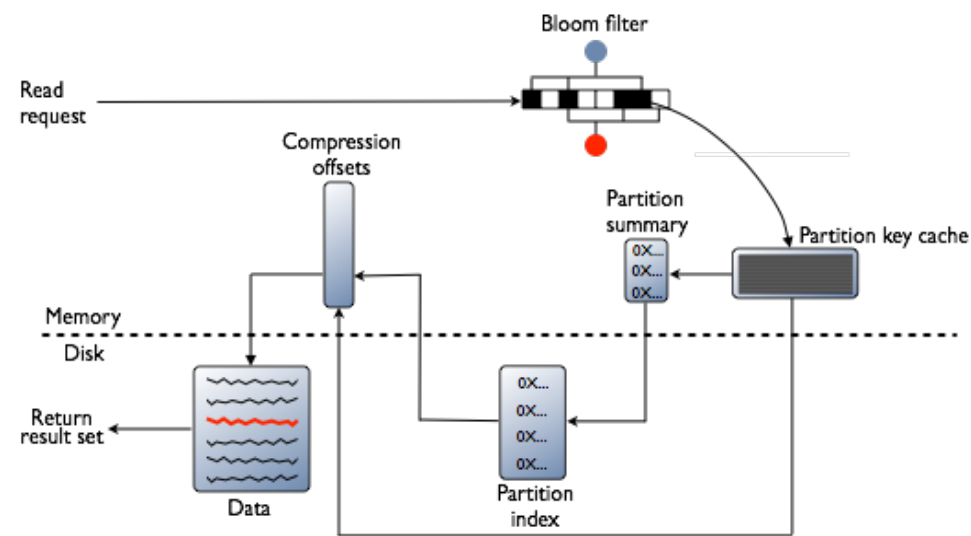
Read

- Row Cache
 - Can provide some improvement for very read-intensive operations, where read operations are 95% of the load.
- Stores a subset of the partition data stored on disk in the SSTables in memory



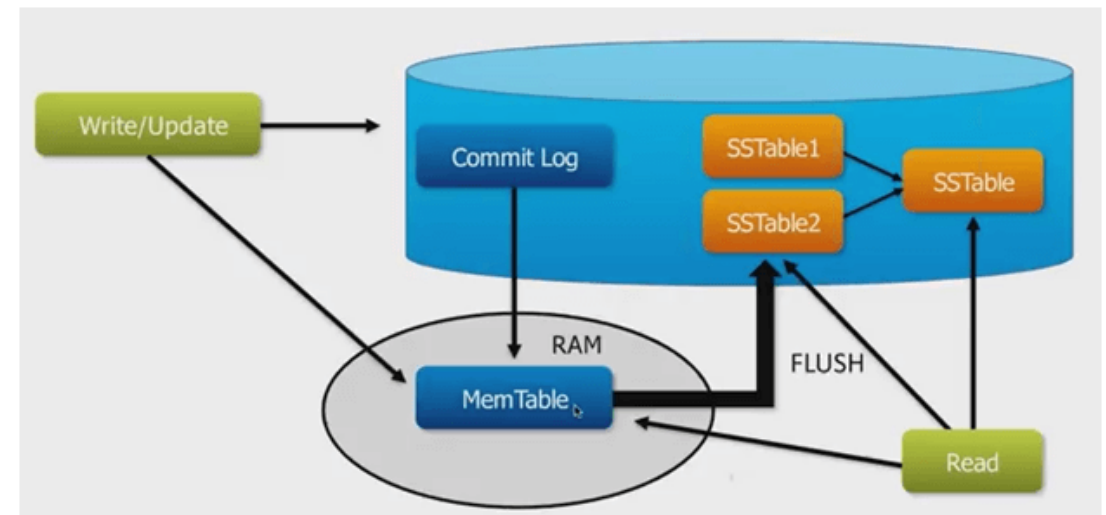
Read

- Bloom Filter
- Cassandra checks the Bloom filter to discover which SSTables are likely to have the request partition data.
- Each SSTable has a Bloom filter associated with it.
 - Can establish that a SSTable does not contain certain partition data.
 - Can also find the likelihood that partition data is stored in a SSTable.
 - Speeds up the process of partition key lookup by narrowing the pool of keys.
- Because the Bloom filter is a probabilistic function, it can result in false positives.
 - Not all SSTables identified by the Bloom filter will have data
 - If the Bloom filter does not rule out an SSTable, Cassandra checks the partition key cache



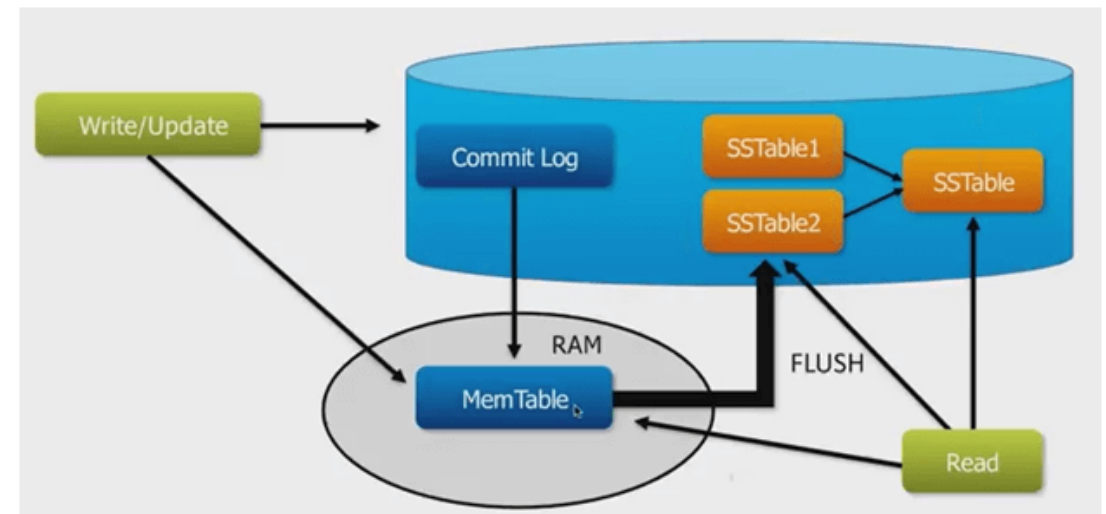
Read Operations – Direct Request

- 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.



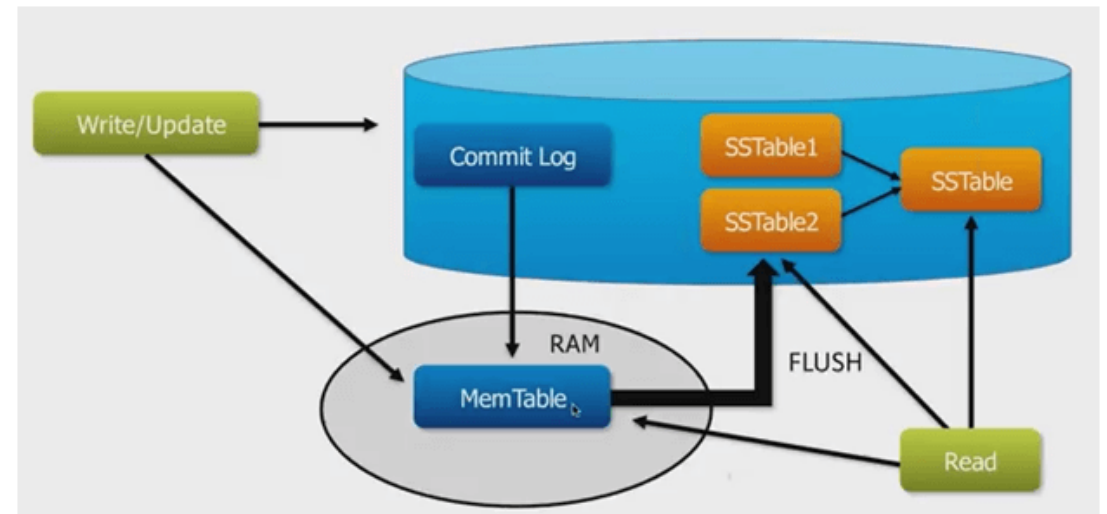
Read Operations – Digest Request

- The coordinator first contacts the replicas specified by the consistency level.
- The coordinator sends these requests to the replicas that currently respond the fastest.
- The contacted nodes respond with a digest of the requested data; if multiple nodes are contacted, the rows from each replica are compared in memory for consistency.
- If they are not consistent, the replica having the most recent data (based on the timestamp) is used by the coordinator to forward the result back to the client.
- To ensure that all replicas have the most recent version of the data, read repair is carried out to update out-of-date replicas.



Read Operations – Read/Repair Request

- Process of repairing replicas during the read process
- Happens in background
- Read repair requests ensure that the requested row is made consistent on all replicas involved in a read query.



Cassandra Indexes

Cassandra Indexing

- Provides a means to access data in Cassandra using attributes other than the partition key.
- Improves speed and efficiency of data lookup when matching a given condition.
- The index indexes column values in a separate, hidden table from the one that contains the values being indexed.
- Indexes can be used for collections, collection columns, and any other columns except counter columns and static columns.

Cassandra Indexing

- Suppose we have the following table:

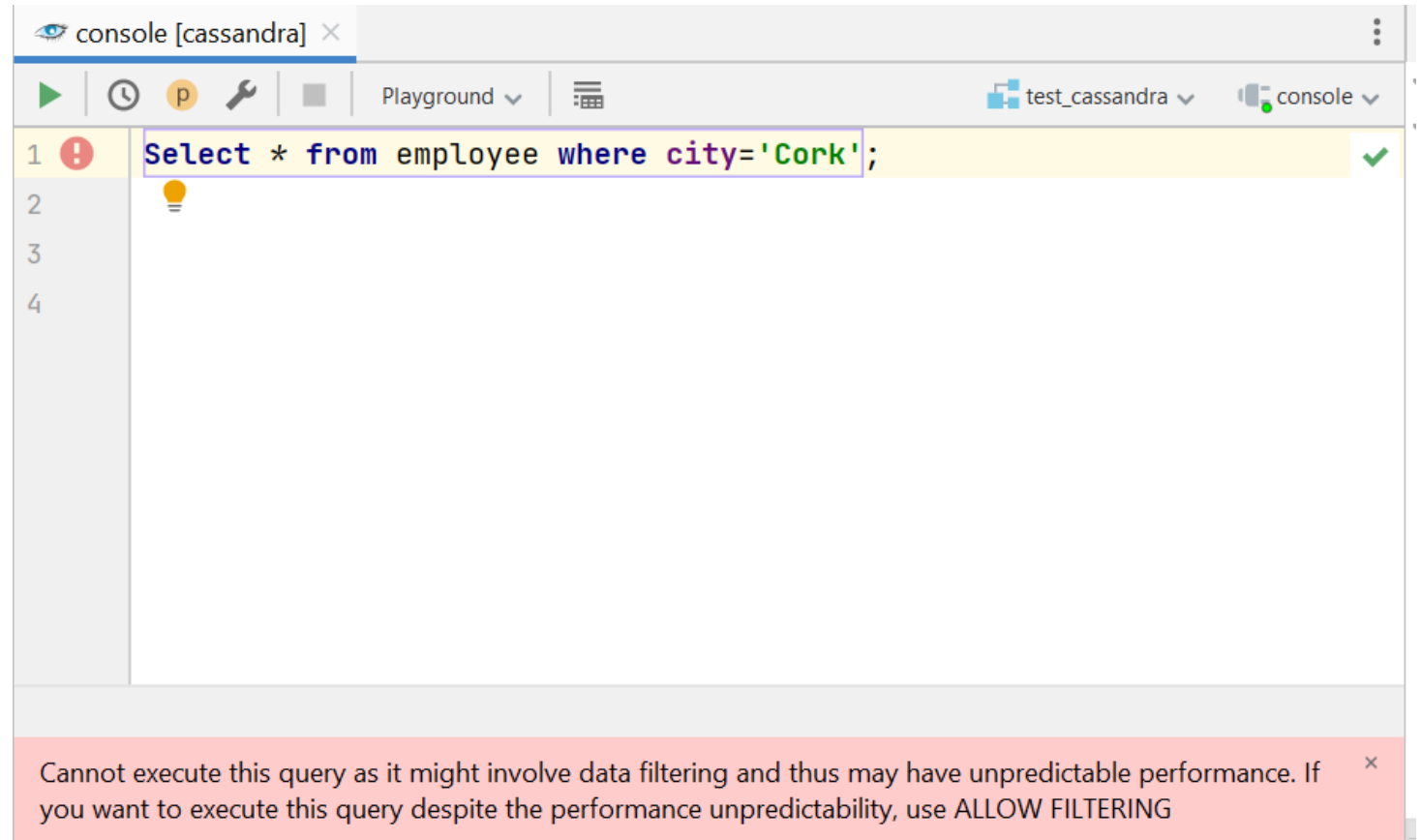
```
CREATE TABLE employee (  
  emp_id int,  
  name text ,  
  city text,  
  PRIMARY KEY ((emp_id), name)  
);
```

Primary
Key

Clustering
Key

Querying on a non-primary key

- We cannot query a column that's not part of the primary key



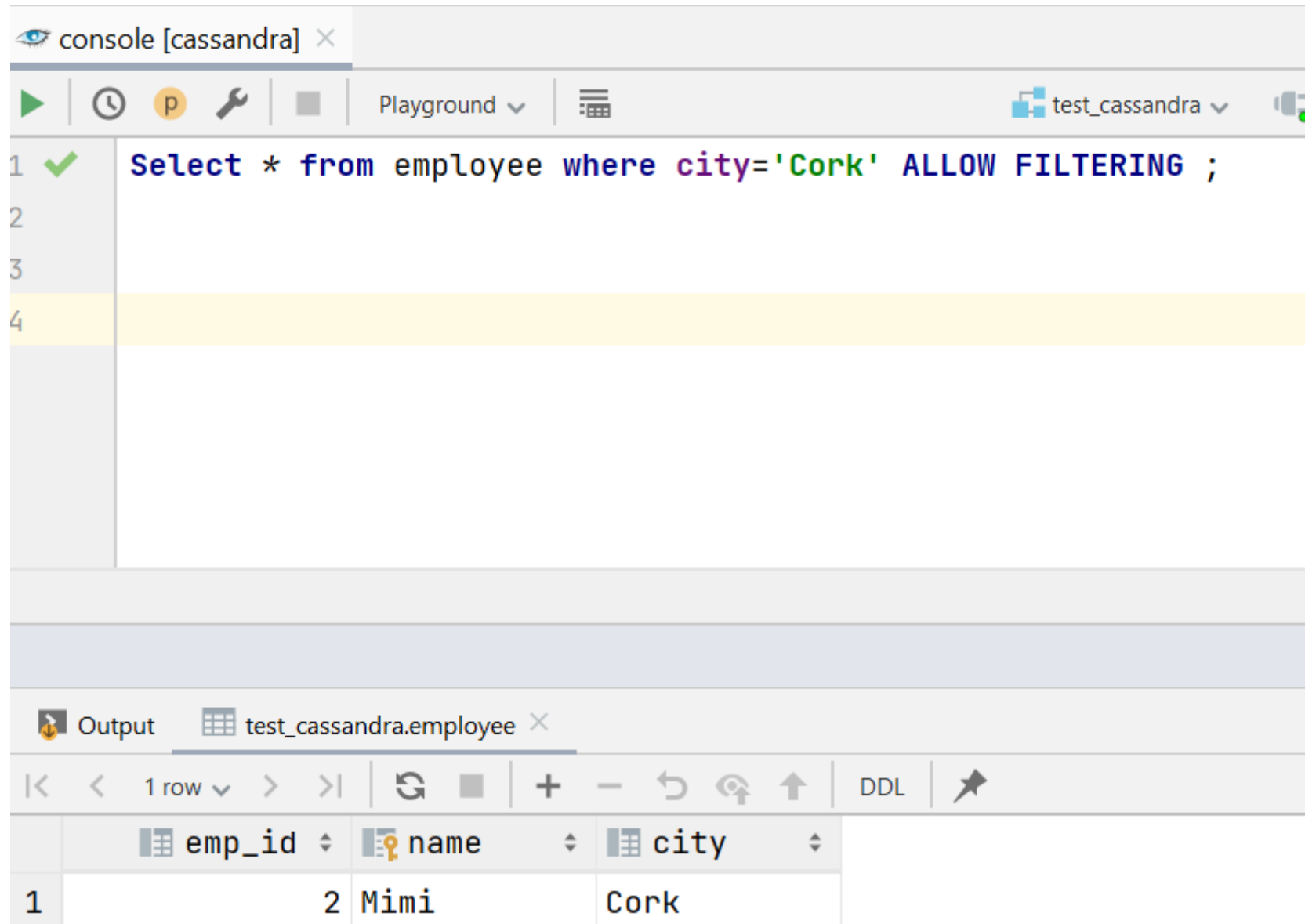
The screenshot shows a web-based interface for a Cassandra console. At the top, there's a tab labeled 'console [cassandra]'. Below the tab, there's a toolbar with icons for play, stop, refresh, and a dropdown menu labeled 'Playground'. To the right of the toolbar, there are two dropdown menus: 'test_cassandra' and 'console'. The main area is a text editor with a line number column on the left (1, 2, 3, 4). Line 1 contains the query: `Select * from employee where city='Cork';`. A red exclamation mark icon is next to line 1, and a green checkmark icon is at the end of the line. A yellow lightbulb icon is positioned below the query. At the bottom of the console, there's a red error message box that reads: 'Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING'.

```
1 ! Select * from employee where city='Cork'; ✓
2
3
4
```

Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING

Querying on a non-primary key

- Unless we use the ALLOW FILTERING clause



The screenshot shows a web-based interface for interacting with a Cassandra database. At the top, a tab labeled 'console [cassandra]' is active. Below the tab is a toolbar with icons for running, saving, and other actions. The main area contains a SQL query: `Select * from employee where city='Cork' ALLOW FILTERING ;`. The query is highlighted in yellow. Below the query, there is a section labeled 'Output' with a sub-tab 'test_cassandra.employee'. This section displays the results of the query in a table format. The table has three columns: 'emp_id', 'name', and 'city'. The first row shows '1' for emp_id, 'Mimi' for name, and 'Cork' for city.

```
Select * from employee where city='Cork' ALLOW FILTERING ;
```

	emp_id	name	city
1	2	Mimi	Cork



Querying on a non-primary key

Even if we technically can, we should not use it in production because ALLOW FILTERING is expensive and time-consuming.

This is because, in the background, it starts full-table scans across all nodes in the cluster to fetch the results, which has a negative impact on performance.

Querying on a non-primary key

- An acceptable use case is when we need to do a lot of filtering on a single partition.
- In this case, Cassandra still performs a table scan, but we can limit it to a single node
- E.g.

```
Select * from employee where name='Mimi' ALLOW FILTERING;
```

- Because name is a clustering column it can be used as a condition
 - Cassandra uses it to identify the node that holds all the company data.
 - Consequently, it performs a table scan just on the table data on that specific node.
-

Secondary Indexes

- Solve the need for querying columns that are not part of the primary key.

Regular Secondary Indexes




Regular Secondary Index

- Most basic index that can be defined for executing queries on non-primary key columns.

```
CREATE INDEX IF NOT EXISTS ON employee (city);
```

- A query using `city` can now run without any errors or need to use `allow filtering`:

```
SELECT * FROM employee WHERE city='Cork';
```

	 emp_id	 city	 name
1	2	Cork	Mimi

Regular Secondary Index

- Cassandra creates a hidden table for storing the index data in the background
- Cassandra doesn't distribute the hidden index table using the cluster-wide partitioner.
- The index data is co-located with the source data on the same nodes.
 - Why?
 - Reduces latency
 - Also because index is updated locally avoids losing updates due to connectivity
- Therefore, when executing a search query using the secondary index, Cassandra reads the indexed data from every node and collects all the results.
- If our cluster has many nodes, this can lead to increased data transfer and high latency.

Regular Secondary Index

- When data is written to (insert into) a table with a secondary index attached,
 - Cassandra writes to both the index and the base Memtable.
 - Both are flushed to the SSTables simultaneously.
 - The index data will have a separate lifecycle than the source data.
- When data is read based on the secondary index (select from)
 - Cassandra first retrieves the primary keys for all matching rows in the index,
 - It then uses them to fetch all the data from the source table.

SSTable-Attached Secondary Index (SASI)

SSTable-Attached Secondary Index (SASI)

- Binds the SSTable lifecycle to the index
- Can be created on any non-collection column

```
CREATE CUSTOM INDEX IF NOT EXISTS  
    employee_by_city ON employee (city) USING  
    'org.apache.cassandra.index.sasi.SASIIndex';
```

- Performing in-memory indexing followed by flushing the index with the SSTable to disk reduces disk usage and saves CPU cycles.
- The advantages of SASI are the tokenized text search, fast range scans, and in-memory indexing.
- A disadvantage is that it generates big index files, especially when enabling text tokenization.

SSTable-Attached Secondary Index (SASI) PREFIX

- SASI implements three types of indexes, PREFIX, CONTAINS, and SPARSE.
- PREFIX is the default mode.

```
CREATE CUSTOM INDEX nm_prefix ON employee(name) USING  
    'org.apache.cassandra.index.sasi.SASIIndex';
```

- Queries can find exact matches for values in name.
- Queries can find matches for values in name based on partial matches.
- The use of LIKE specifies that the match is looking for a word that starts with the letter "M". The % after the letter "M" will match any characters can return a matching value.

```
SELECT * FROM employee WHERE name LIKE 'M%';
```

SSTable-Attached Secondary Index (SASI) PREFIX

Many queries will fail with this index:

```
SELECT * FROM employee WHERE name = 'MIMI';
```

```
SELECT * FROM employee WHERE name LIKE 'm%';
```

```
SELECT * FROM employee WHERE name LIKE '%m%';
```

```
SELECT * FROM employee WHERE name LIKE '%m%' ALLOW FILTERING;
```

Fail due to case sensitivity

```
SELECT * FROM employee WHERE name LIKE '%M%';
```

```
SELECT * FROM employee WHERE name = '%M';
```

```
SELECT * FROM employee WHERE name = '%M%';
```

Fail because of PREFIX mode – only a trailing % will work

SSTable-Attached Secondary Index (SASI) CONTAINS

- CONTAINS facilitates pattern matching for partial patterns given:

```
CREATE CUSTOM INDEX nm_contains ON employee (name)
    USING 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = { 'mode': 'CONTAINS' };
```

- Queries can find exact matches for values in name.
- **NOTE:** for queries on CONTAINS indexing, the ALLOW FILTERING phrase must be included, although Cassandra will not actually filter.

```
SELECT * FROM employee WHERE name LIKE 'M%' ALLOW FILTERING;
```

SSTable-Attached Secondary Index (SASI) CONTAINS




- Queries can find matches for values in name based on partial matches.
- The use of LIKE specifies that the match is looking for a word that contains the letter "M".
- The % before and after the letter "M" will match any characters and return a matching value.

```
SELECT * FROM employee WHERE name LIKE '%M%' ALLOW  
FILTERING;
```

SSTable-Attached Secondary Index (SASI) CONTAINS

- More sensitive than the prefix index:

```
SELECT * FROM employee WHERE name LIKE '%n%' ALLOW  
FILTERING;
```

	 emp_id	 name	 city
1	1	John	Dublin
2	3	Annie	Wexford

SSTable-Attached Secondary Index (SASI) CONTAINS

- Many will still fail

```
SELECT * FROM employee WHERE name = 'MariAnne'  
ALLOW FILTERING;
```

```
SELECT * FROM employee WHERE name LIKE '%m%';
```

Fail due to case sensitivity

```
SELECT * FROM employee WHERE name LIKE '%A';
```

```
SELECT * FROM employee WHERE name LIKE 'n%';
```

Fail due to placement of the %

SSTable-Attached Secondary Index (SASI) Additional Options

CONTAINS and PREFIX can be created with additional options to handle case sensitivity
The option `case_sensitive` set to `false` makes the indexing case insensitive.

```
CREATE CUSTOM INDEX IF NOT EXISTS  
employee_by_city ON employee (city) USING  
'org.apache.cassandra.index.sasi.SASIIndex'  
WITH OPTIONS = {  
  'mode': 'CONTAINS',  
  'analyzer_class':  
'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',  
  'case_sensitive': 'false'};
```

SSTable-Attached Secondary Index (SASI) Additional Options

If the analyzer_class used is the non-tokenizing analyzer it does not perform analysis on the text in the specified column:

```
CREATE CUSTOM INDEX IF NOT EXISTS  
employee_by_city ON employee (city) USING  
'org.apache.cassandra.index.sasi.SASIIndex'  
WITH OPTIONS = {  
  'mode': 'CONTAINS',  
  'analyzer_class':  
  'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',  
  'case_sensitive': 'false'};
```

This means that this query will now work:

```
SELECT * FROM employee WHERE name LIKE '%m%';
```

SSTable-Attached Secondary Index (SASI) SPARSE

- Can improve performance of querying large, dense number ranges like timestamps for data inserted every millisecond
- Best choice when the data is numeric
 - and millions of columns values with a small number of partition keys characterize the data
 - and range queries will be performed against the index
- For numeric data that does not meet this criteria, PREFIX is the best choice.

Storage-Attached Secondary Index (SAI)

Storage-Attached Indexing (SAI)

- Custom index
- One or more can be defined on any column and then used for range queries (numeric only), CONTAINS semantics, and filter queries.
- SAI stores individual index files for each column and contains a pointer to the offset of the source data in the SSTable.
 - Once data is inserted into an indexed column, it will be written first to memory.
 - Whenever Cassandra flushes data from memory to disk, it writes the index along with the data table.

Storage-Attached Indexing (SAI)

- This approach improves throughput by 43% and latency by 230% over regular secondary indexes by reducing the overhead for writing.
- Compared to SASI and secondary indexes, it uses significantly less disk space for indexing, has fewer failure points, and comes with a more simplified architecture.
- We won't be investigating this.

Consistency

Consistency Levels

- Consistency indicates how recent and in-sync all replicas of a row of data are.
- With the replication of data across the distributed system, achieving data consistency is a very complicated task.
- Cassandra prefers availability over consistency.
- It doesn't optimize for consistency.
- Instead, it provides the flexibility to tune the consistency depending on your use case.
- In most use cases, Cassandra relies on eventual consistency.

Consistency Levels - Write

- The consistency level specifies how many replica nodes must acknowledge back before the coordinator successfully reports back to the client.
- Can be set to a particular number, LOCAL or QUORUM
- Example
 - Consistency level ONE
 - means it needs acknowledgment from only one replica node.
 - The consistency level of LOCAL or QUORUM
 - means it needs acknowledgment from 51% or a majority of replica nodes across all datacenters.

Consistency Levels - Write

- The number of nodes that acknowledge (for a given consistency level) and the number of nodes storing replicas (for a given RF) are mostly different.
 - For example, with the consistency level ONE and RF = 3, even though only one replica node acknowledges back for a successful write operation, Cassandra asynchronously replicates the data to 2 other nodes in the background.

Consistency Levels - Read

- Specifies how many replica nodes must respond with the latest consistent data before the coordinator successfully sends the data back to the client.
- Example
 - The consistency level ONE
 - Means only one replica node returns the data.
 - The consistency level ALL or QUORUM
 - Means 51% or a majority of replica nodes across all datacenters responds.
 - Then the coordinator returns the data to the client.
 - In the case of multiple data centers, the latency of inter-data center communication results in a slow read.

Eventual Consistency V Strong Consistency

- Cassandra defaults to Eventual consistency
 - the value for a specific data item will, given enough time without updates, be consistent across all nodes
- Cassandra can achieve Strong Consistency
 - If $W + R > RF$, where R – read CL replica count, W – write CL replica count, RF – replication factor.
 - Example
 - If $RF = 3$, $W = \text{QUORUM}$ or LOCAL_QUORUM , $R = \text{QUORUM}$ or LOCAL_QUORUM , then $W(2) + R(2) > RF(3)$
 - The write operation makes sure two replicas have the latest data.
 - Then the read operation also makes sure it receives the data successfully only if at least two replicas respond with consistent latest data.

Cassandra Data Types

Data types

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

Data types

- `ascii`: strings, US-ASCII character string.
- `bigint`: integers, 64-bit signed long.
- `blob`: blobs, contains arbitrary bytes (no validation) and expressed as hexadecimal.
- `boolean`: booleans, either true or false.
- `decimal`: integers and floats, variable-precision decimal.
- `double`: double, 64-bit IEEE-754 floating-point.
- `float`: integers and floats, 32-bit IEEE-754 floating-point.
- `inet`: strings but an IP address string in IPv4 or IPv6 format.
- `int`: integers, a 32-bit signed integer.

Data types

- text: text, string.
- timestamp: integers and strings, includes as a date plus time and is encoded as 8 bytes since epoch.
- uuid: uuids, a UUID in standard UUID format.
- timeuuid: timeuuid, a type 1 UUID only (CQL 3).
- varchar: varchar, encoded string.
- varint: variant, an arbitrary-precision integer.

Data types

- counter: integers, a distributed counter value (64-bit long).
- A special column used to store a number that this changed increments.
- Restriction on the counter column:
- Counter column cannot index, delete or re-add a counter column.
- All non-counter columns in the table must be defined as a part of the primary key.
- To load data in a counter column or to increase or decrease the value of the counter, use the update command.

Cassandra Collections

- When a user has multiple values against one field in a relational database, it's common to store them in a separate table.
 - E..g. if a user has numerous orders, contact information, or postal addresses in an online retail situation we need to apply joins between two tables to retrieve all the data in this case.
- Joins are not allowed in Cassandra but Cassandra provides a way to group and store data together in a column using collection types.

Cassandra Collections

- Map
 - A sorted set of key-value pairs, where keys are unique and the map is sorted by its keys.
- Sets
 - A sorted collection of unique values.
- Lists:
 - A sorted collection of non-unique values where elements are ordered by their position in the list.

Cassandra Collections - List

- There is one rule for the list data type.
 - The order of the elements cannot be changed.
- After storing the values in the list, the elements get a particular index.
 - The values can be retrieved through these indexes.

```
CREATE TABLE <table name>(
column1 PRIMARY KEY,
column2 list <data type>,
column3 list <data type>,.....);

INSERT INTO <table name>(column1, column2,
column3,...)
VALUES('R1value1',['R1value1','R1value2','R1value
3'...]['R1value11','R1value12','R1value13'...]....);
```


Cassandra Collections - List

```
CREATE TABLE employee(  
  ENum int,  
  NAME text,  
  PRIMARY KEY(ENum),  
  EMAIL LIST<text>  
);
```

```
INSERT INTO college_student (ENum,  
NAME, EMAIL)
```

```
VALUES(001,'Ayush',['ayush@gmail.com',  
'AY@mail.com']));
```

```
INSERT INTO college_student (ENum,  
NAME, EMAIL)
```

```
VALUES(002,'Aarav',['aarav@ymail.com',  
'AR@mail.com']));
```

```
INSERT INTO college_student (ENum,  
NAME, EMAIL)
```

```
VALUES(003,'Kabir',p'kabir@hotmail.com']));
```

```
Update college_student set email = email +  
['data science'] where Name='Ayush';
```

Cassandra Collections - Set

- The elements in the set returns in a sorted order after execution.

```
CREATE TABLE<table name> (  
column1 PRIMARY KEY,  
column2 set <data type>,  
column3 set <data type>.....);
```

```
INSERT INTO <table name>(column1, column2, column3...)  
VALUES('R1value',{'R1value1', 'R1value2',...},{ 'R1value11', 'R1value12',...}....);
```

Cassandra Collections - Set

```
CREATE TABLE college_student(  
  SNum int,  
  NAME text,  
  BRANCH SET<text>,  
  PRIMARY KEY(ID));
```

```
INSERT INTO college student (SNum, NAME, BRANCH)  
VALUES(001,'Ayush',{ 'electrical engineering', 'computer  
science'});
```

```
INSERT INTO college student (SNum, NAME, BRANCH)  
VALUES(002,'Aarav',{'Computer engineering'});
```

```
INSERT INTO college student (SNum, NAME, BRANCH)  
VALUES(003,'Kabir',{'Applied Physics'});
```

```
Update college_student set branch = branch +  
{ 'computer science' } where Name='Aarav';
```

Cassandra Collections - Map

- Stores a Key+Value Pair

```
CREATE TABLE<table name> (  
column1 PRIMARY KEY,  
column2 map <type, data type>,  
column3 map <type, data type>.....);
```

```
INSERT INTO <table name>(column1, column2, column3...)  
VALUES('R1value',{'R1value1':'R1value1',R1value2:'R1value01',...},{  
'R1value11':'R1value011','R1value12':'R1value012',...}....);
```

Cassandra Collections - Map

```
CREATE TABLE student(  
  ENum int,  
  SUBJECT MAP(text,text),  
  PRIMARY KEY(EN));
```

```
INSERT INTO student(ENum, SUBJECT)  
VALUES(001,{'physics':'mathematics', 'operating system': 'linux'})  
  
INSERT INTO student(ENum, SUBJECT)  
VALUES(002,{'operating system':'windows'})  
  
INSERT INTO student(ENum, SUBJECT)  
VALUES(003,{'power system':'machines'}
```

```
Update college_student set subject['power system'] = 'devices'  
where EN=003;
```

```
Update college_student set subject= system +  
{'pyhsics:mathematics'} where EN=003;
```