Week3_Cassandra_Basics

Overview

- Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunable, and consistent database that abses its distribution design on Amazon's Dynamo and its data model on Google's Bigtable. Created at Facebook, it is now used at some of hte most popular sites on the Web.
- "Always available" type of services (Netflix, Spotify)
- Fast writes - capture all data
- Availability & scalability
- Peer to peer architecture

A reliable, performant, scalable database for data storage

- Not a drop-in replacement for a relational database
    - does not support joins
    - limited agfgragations support
    - limited support of transactions

For joins and aggragations: Cassandra + Spark

When to use?

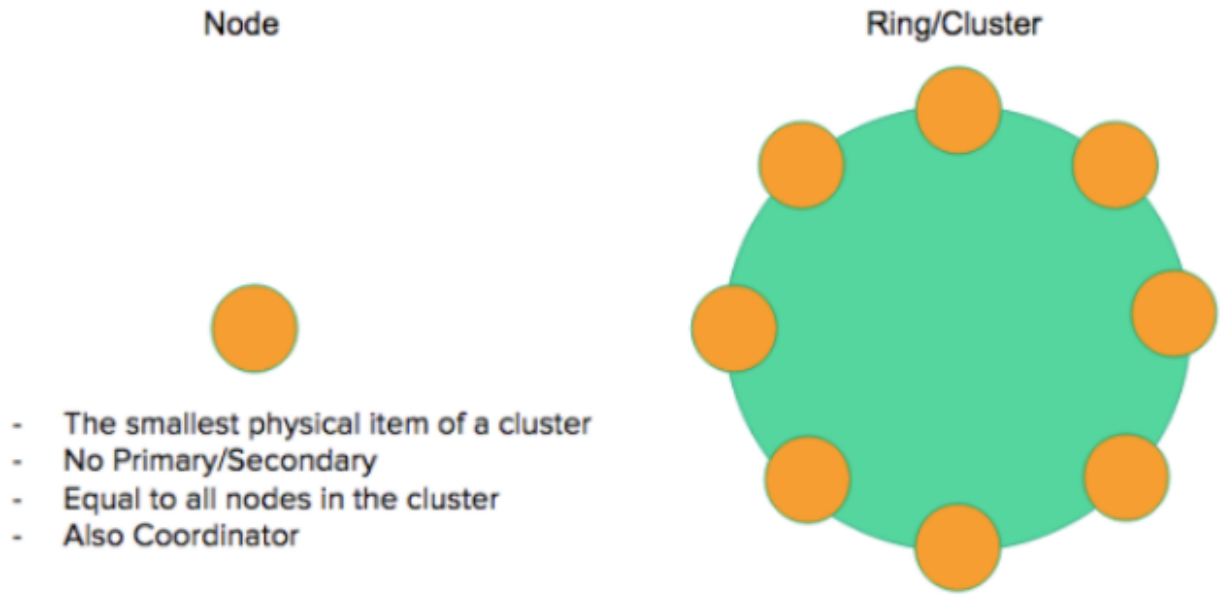
- When writes exceed read requests
    - Fore xample, storing all the clicks on your website or all the access attempts on your service
- When using append-like type of data
    - Not many updates or deletes
- When you can predefine your queries and your data access is by a known primary key
    - Data can be partitioned via a key that allows the database to be spread evenly across multiple nodes
- When there is no need for joins or aggregations
- Online services
    - Users authentication for access to services
    - Tracking users activitiy in the application
- eCommerce websites
    - Storing transactions

- website interactions (clicks) for prediction of customer behavior
- Status of orders/users transactions
- Users profiles and shopping history

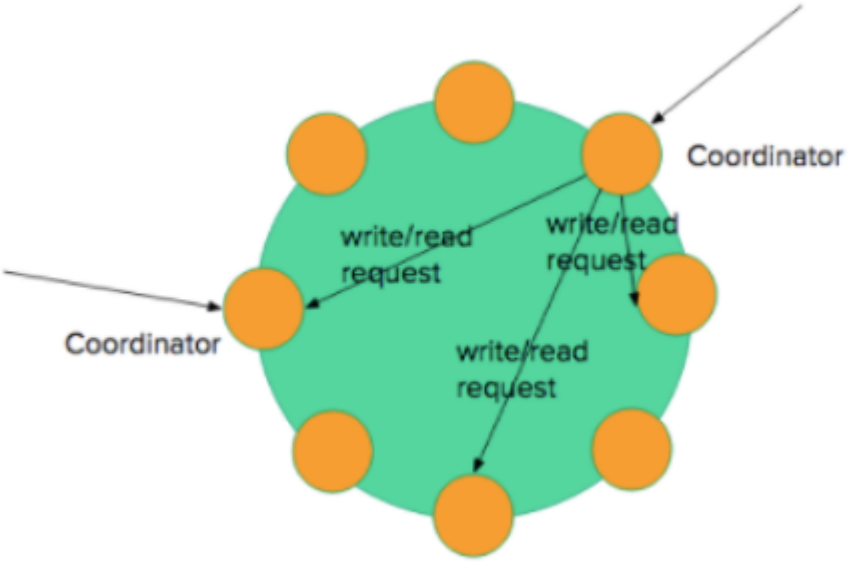
Architecture

The Apache Cassandra architecture is designed to provide scalability, availability, and reliability to store massive amounts of data

Cassandra is based on a distributed system architecture. In its simplest form, Cassandra can be installed on a single machine or container. A single Cassandra instance is called a node. Cassandra supports horizontal scalability achieved by adding more than one node as a part of a Cassandra cluster.



As well as being a distributed system, Cassandra is designed to be a peer-to-peer architecture, with each node connected to all other nodes. Each Cassandra node can perform all database operations and can serve client requests without the need for a primary node.

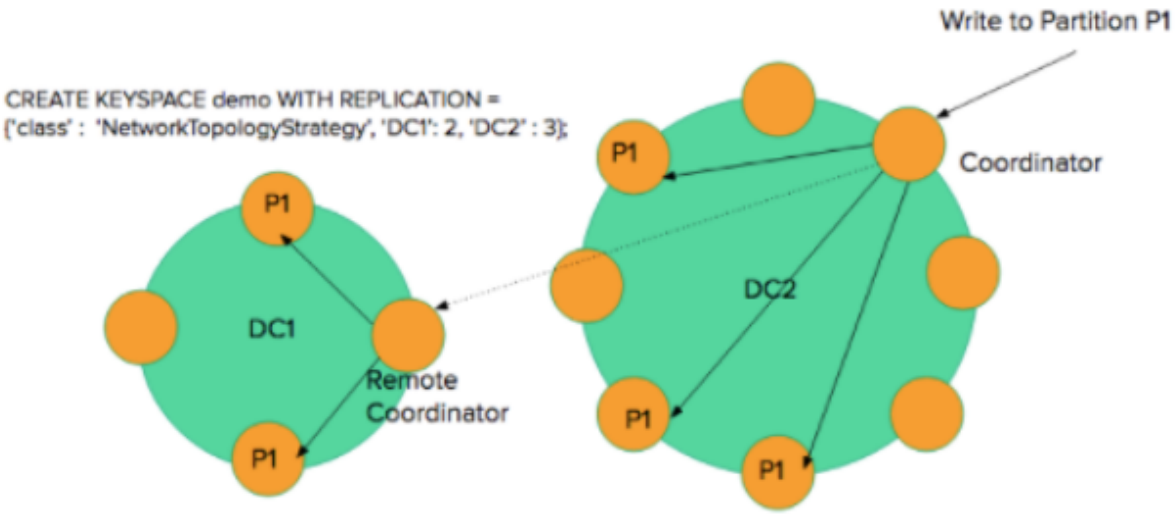


How do the nodes in this peer-to-peer architecture (no primary node) know to which node to route a request and if a certain node is down or up? Through Gossip.

Gossip is the protocol used by Cassandra nodes for peer-to-peer communication. The gossip protocol informs a node about the state of all other nodes. A node performs gossip communications with up to three other nodes every second. The gossip messages follow a specific format and use version numbers to make efficient communication, thus shortly each node can build the entire metadata of the cluster (which nodes are up/down, what are the tokens allocated to each node, etc..).

Multi Data Centers Deployment

A Cassandra cluster can be a single data center deployment (like in the above pics), but most of the time Cassandra clusters are deployed in multiple data centers. A multi data-center deployment looks like below – where you can see depicted a 12 nodes Cassandra cluster, topology wise installed in 2 datacenters. Since replication is being set at keyspace level, demo keyspace specifies a replication factor 5: 2 in data center 1 and 3 in data center 2.



Note: since a Cassandra node can be as well a coordinator of operations, in our example since the operation came in data center 2 the node receiving the operation becomes the coordinator of the operation, while a node in data center 1 will become the remote coordinator – taking care of the operation in only data center 1.

Components of a Cassandra Node

There are several components in Cassandra nodes that are involved in the write and read operations. Some of them are listed below:

Memtable

Memtables are in-memory structures where Cassandra buffers writes. In general, there is one active Memtable per table. Eventually, Memtables are flushed onto disk and become immutable SSTables.

This can be triggered in several ways:

The memory usage of the Memtables exceeds a configured threshold.

The CommitLog approaches its maximum size, and forces Memtable flushes in order to allow Commitlog segments to be freed.

When we set a time to flush per table.

CommitLog

Commitlogs are an append-only log of all mutations local to a Cassandra node. Any data written to Cassandra will first be written to a commit log before being written to a Memtable. This provides durability in the case of unexpected shutdown. On startup, any mutations in the commit log will be applied to Memtables.

SSTables

SSTables are the immutable data files that Cassandra uses for persisting data on disk. As SSTables are flushed to disk from Memtables or are streamed from other nodes, Cassandra triggers compactions which combine multiple SSTables into one. Once the new SSTable has been written, the old SSTables can be removed.

Each SSTable is comprised of multiple components stored in separate files, some of which are listed below:

Data.db: The actual data.

Index.db: An index from partition keys to positions in the Data.db file.

Summary.db: A sampling of (by default) every 128th entry in the Index.db file.

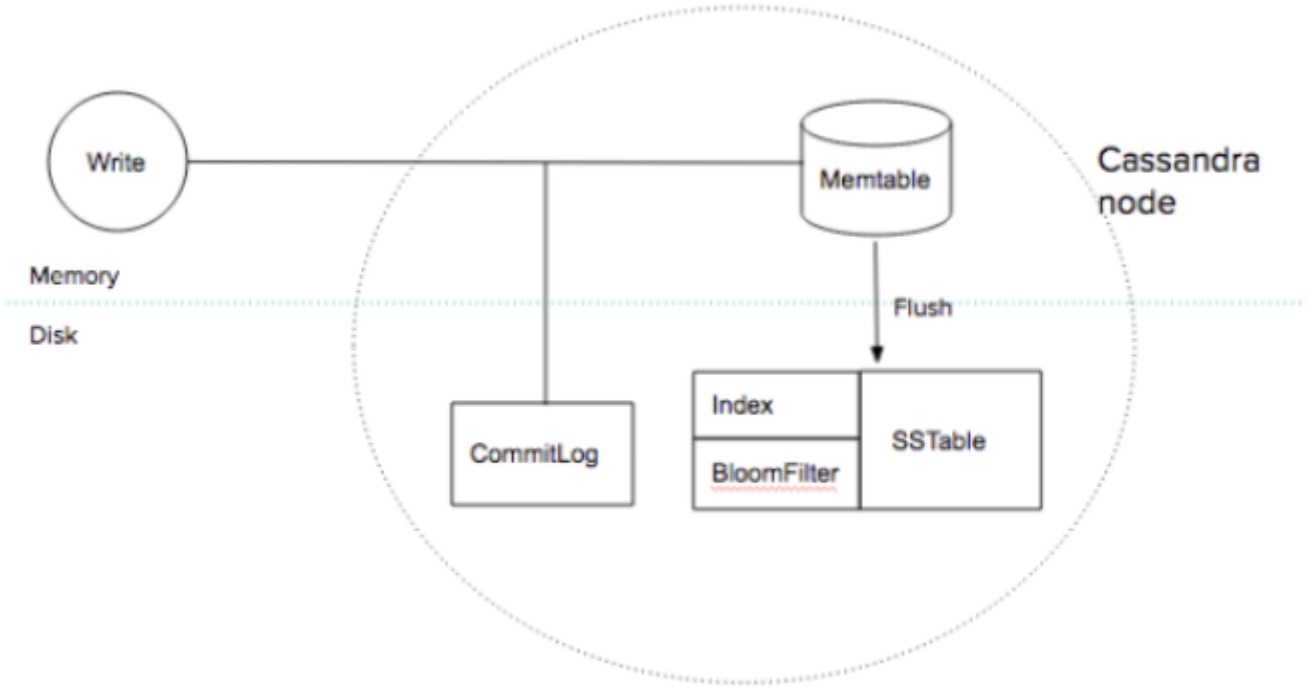
Filter.db: A Bloom Filter of the partition keys in the SSTable.

CompressionInfo.db: Metadata about the offsets and lengths of compression chunks in the Data.db file.

Write Process at Node Level

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending with a write of data to disk:

- Logging data in the commit log
- Writing data to the Memtable
- Flushing data from the Memtable
- Storing data on disk in SSTables

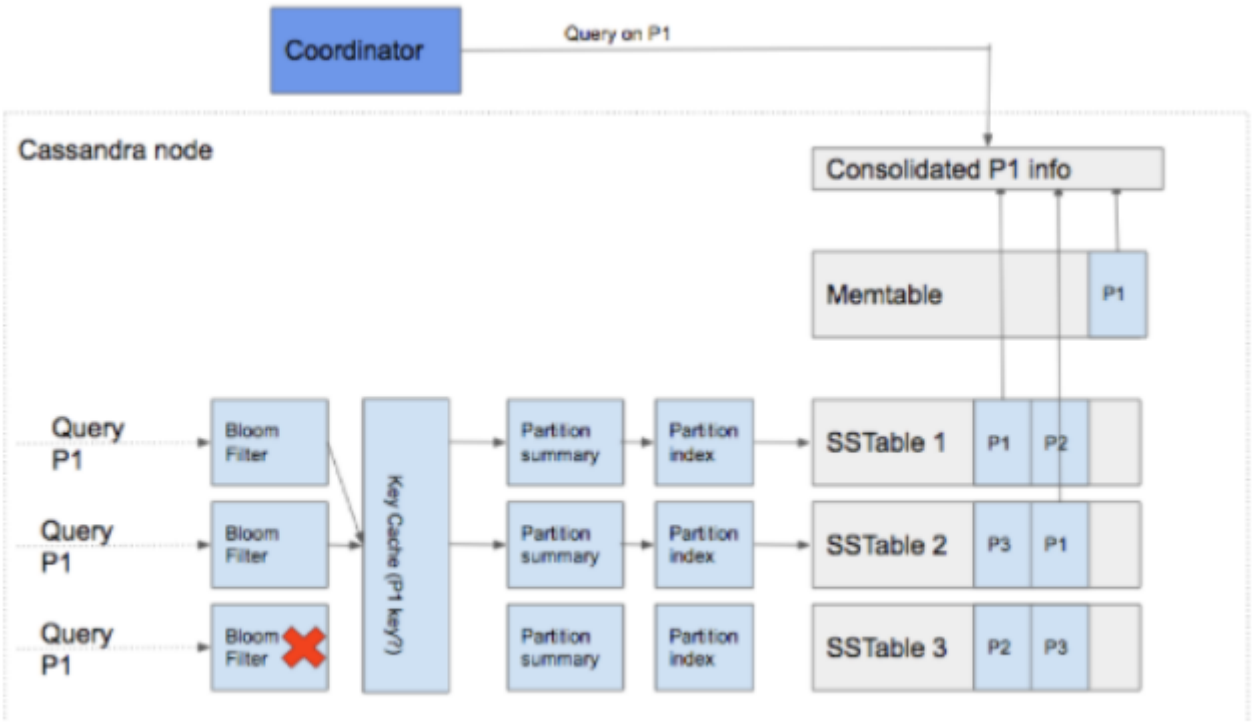


Read at node level

While writes in Cassandra are very simple and fast operations, done in memory, the read is a bit more complicated, since it needs to consolidate data from both memory (Memtable) and disk (SSTables). Since data on disk can be fragmented in several SSTables, the read process needs to identify which SSTables most likely contain info about the partitions we are querying - this selection is done by the Bloom Filter information. The steps are described below:

- Checks the Memtable
- Checks Bloom filter
- Checks partition key cache, if enabled
- If the partition is not in the cache, the partition summary is checked

- Then the partition index is accessed
- Locates the data on disk
- Fetches the data from the SSTable on disk
- Data is consolidated from Memtable and SSTables before being sent to coordinator



Key Features

- **Distributed and decentralized**
  - Cluster runs on multiple distributed machines
  - Users address the cluster in the same way: seamless to the number of nodes in the cluster
  - All nodes perform the same functions (server symmetry)
  - Peer-to-peer architecture
- **Always available with tunable consistency**
- Tunable consistency
  - per operation set consistency (read/write)
- CAP theorem: Cassandra favors availability over consistency
  - Tunable: Strong or eventual consistency
  - Consistency conflicts solved during reads
- **Fault tolerant**
  - Peer to peer architecture

    - Nodes temporary/permament failures are immediately recognized by the other nodes in the cluster
    - Nodes reconfigure the data distribution once nodes are taken out of the cluster
    - Failed erquests can be re-transmitted to other nodes
- **High write throughput**

At cluster level, writes can be distributed in parallel to all nodes holding replicas

- No reading before writing (by default)
- At node level
    - Writes are done in node memory and later flushed on disk
    - All disk write are sequential ones - append-like operations
- **Fast and linear scalability**
- Scales horizontally by adding nerw nodes in the cluster
- Performance increases linearly with the number of added nodes
- New nodes are automatically assigned tokens from existing nodes
- Adding and removing of nodes is done seamlessly
- **Multiple data center support**
- **SQL-like query language**

CQL is the data definition and manipulation language for Cassandra, an sql-like syntax

Cassandra Data Model

Cassandra has two logical entities: Tables and Keyspaces

- **Table**:
    - Logical entity that organizes data storage at cluster and node level (according to a declared schema);
    - Data is organized in tables containing rows of columns
    - Tables can be created, dropped, and altered at runtime without blocking updates and queries
    - To create a table, you must define a primary key and other data columns (regular columns)

```
CREATE TABLE intro_cassandra.groups(
groupid int,
group_name text STATIC,
username text,
age int,
PRIMARY KEY ((groupid), username));
```

You can see that the primary key is composed by two columns, where:
`groupid` = Partition key
`username` = clustering key

- The primary key is a subset of the declared columns
- It is mandatory to define a primary key, and you cannot change it once declared
- Two main roles:
  - Optimize read performance for table queries - query driven table design
  - Provide uniqueness to the entries
- Two components:
  - **Partititon key**:
    - Mandatory, it is responsible for data locality
  - **Clustering key(s)**:
    - Optional, stores data in acscending or descending order within the partition for the fast retrieval or similar values;
    - Can have single or multoiple columns
    - Completes the primary key in dynamic tables
      - Gives uniqueness to a primary key
      - Improves read query performance

When data is written to a table, it is grouped into partitions and distributed on cluster nodes - based on Partition key

Partition Key => Hash (token) => Node
Partition key determines data (partition) locality in cluster

- Two types:
  - Static table: PRIMARY KEY only
    `PRIMARY KEY (username)`
  - Dynamic table: PRIMARY KEY composed
    `PRIMARY KEY ((groupid), username)`
- **Keyspace**: Logical entity that contains one or more table;
  A replication and data center's distribution is defined at keyspace level;
  Recommended 1 keyspace/application

Basic Rules of Data Modeling

- Data Modeling - build a primary key that optimizes query execution time;
- Choose a partition Key - starts answering your query and spreads the data uniformly in the cluster;

- Minimize the number of partitions read in order to answer the query.
  Note: To further optmize your query model, order clustering keys according to the query.

Introduction to cqlsh

- CQL is the primary language for communication with Cassandra clusters
- Simple yet intuitive syntax (sql-like)
- CQL lacks grammar for relational features such as JOIN statements (if you need a join, then store the data already join)
- Different behavior of CQL commands vs SQL
- CQL keywords are case-insensitive
- Identifiers in CQL are case-insensitive unless enclosed in double quotation marks
- Names for identifiers created using uppercase are stored in lowercase
- Commented text (//) is ignored by CQL

```
CREATE KEYSPACE intro_cassandra WITH...

CREATE TABLE test () ...

INSERT INTO test () VALUES ()

SELECT * FROM test WHERE ..

UPDATE test SET age = 25 WHERE userid=30

DELETE FROM test WHERE userid=30

DROP TABLE test;

TRUNCATE TABLE test;
```

Running CQL Queries

- Run using Cassandra client drivers
  - Java, Python, Ruby, Node.js, PHP, Scala, ..
  - Default = open source Datastax Java Driver
- Run using cqlsh client
  - Python-based command line shell for interacting with Cassandra through CQL
  - Shipped with every Cassandra package
  - Connects to a single node (default node or one specified on the command line)

- Other CQL client editors are avaiable

Using `cqlsh`, you can:

- Create, alter, drop keyspaces
- Create, alter, drop, tables
- Insert, update, delete data
- Execute read queries

```
--help
--version
- u -user specifies username to authenticate to cassandra
-p -password specifies password to authenticate to cassandra
-k -keyspace specifies a keyspace
-f -file enables execution of commands from a given file
--request-timeout specifies the request timeout in seconds
```

cqlsh - special commands

`CAPTURE` - Captures the output of a command and adds it to a file
`CONSISTENCY` - Shows the current consistency level and sets a new one;

Sets the consistency level for the operations to follow. Consistency refers to the number of nodes (out of the total replicas) that should respond to a query (write/read) in order to consider the query successful.

`COPY` - Copies data to and from Cassandra

CQL shell command that imports and exports csv - not suitable for bulk loading

`DESCRIBE` - Describes the current cluster of Cassandra and its objects
`EXIT` - Terminates the cqlsh session
`PAGING` - Enables or disables paging of the query results
`TRACING` - Enables or disables request tracing

- Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault tolerant, and tunable and consistent database.
- Apache Cassandra is best used by "always available" type of applications that require a database that is always available.
- Data distribution and replication takes place in one or more data center clusters.
- Its distributed and decentralized architecture helps Cassandra be available, scalable, and fault tolerant.

- Cassandra stores data in tables.
- Tables are grouped in keyspaces.
- A clustering key specifies the order that the data is arranged inside the partition (ascending or descending).
- Dynamic tables partitions grow dynamically with the number of entries.
- CQL is the primary language for communicating with Apache Cassandra clusters.
- CQL queries can be run programmatically using a licensed Cassandra client driver, or they can be run on the Python-based CQL shell client provided with Cassandra.