Cassandra Learning (Cassandra 2.0)

Wednesday, March 05, 2014 16:36

An overview of Cassandra's structure.

Cassandra is designed to handle big data workloads across multiple nodes with no single point of failure. Its architecture is based on the understanding that system and hardware failures can and do occur. Cassandra addresses the problem of failures by employing a peer-to-peer distributed system where all nodes are the same and data is distributed among all nodes in the cluster. Each node exchanges information across the cluster every second. A commit log on each node captures write activity to ensure data durability. Data is also written to an in-memory structure, called a memtable, and then written to a data file called an SSTable on disk once the memory structure is full. All writes are automatically partitioned and replicated throughout the cluster. Using a process called compaction Cassandra periodically consolidates SSTables, discards tombstones (an indicator that a column was deleted), and regenerates the index in the SSTable.

Cassandra is a row-oriented database. Cassandra's architecture allows any authorized user to connect to any node in any data center and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL. From the CQL perspective the database consists of tables. Typically, a cluster has one keyspace per application. Developers can access CQL through cqlsh as well as via drivers for application languages.

Client read or write requests can go to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured. For more information, see Client requests.

Key components for configuring Cassandra

• **Gossip**: A peer-to-peer communication protocol to discover and share location and state information about the other nodes in a Cassandra cluster. Gossip information is also persisted locally by each node to use immediately when a node restarts. You may want to purge gossip history on node restart for various reasons, such as when the node's IP addresses has changed.

a. **Purging gossip state on a node**

Gossip information is persisted locally by each node to use immediately on node restart without having to wait for gossip communications. You may want to clear gossip history on node restart in certain cases, such as when node IP addresses have changed.

Procedure

1. Edit the cassandra-env.sh file:
   - Packaged installs: /usr/share/cassandra
   - Tarball installs: install_location/conf
2. Add the following line to the cassandra-env.sh file: -Dcassandra.load_ring_state=false

• **Partitioner**: A partitioner determines how to distribute the data across the nodes in the cluster. Choosing a partitioner determines which node to place the first copy of data on. You must set the partitioner type and assign the node a num_tokens value for each node. If not using virtual nodes (vnodes), use the initial_token setting instead.

• **Replica placement strategy**: Cassandra stores copies (replicas) of data on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines which nodes to place replicas on. The first replica of data is simply the first copy; it is not unique in any sense. When you create a keyspace, you must define the replica placement strategy and the number of replicas you want.

• **Snitch**: A snitch defines the topology information that the replication strategy uses to place replicas and route requests efficiently. You need to configure a snitch when you create a cluster. The snitch is responsible for knowing the location of nodes within your network topology and distributing replicas by grouping machines into data centers and racks.

• **The cassandra.yaml file** is the main configuration file for Cassandra. In this file, you set the initialization properties for a cluster, caching parameters for tables, properties for tuning and resource utilization, timeout settings, client connections, backups, and security.

Virtual nodes

Overview of virtual nodes (vnodes).

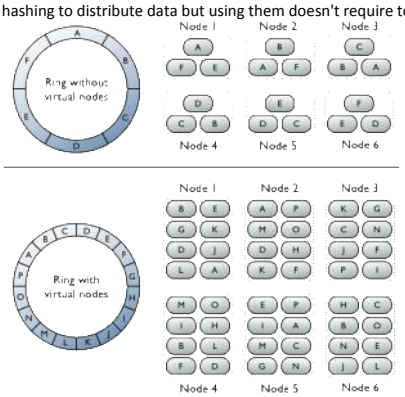
Vnodes simplify many tasks in Cassandra:

- You no longer have to calculate and assign tokens to each node.
- Rebalancing a cluster is no longer necessary when adding or removing nodes. When a node joins the cluster, it assumes responsibility for an even portion of data from the other nodes in the cluster. If a node fails, the load is spread evenly across other nodes in the cluster.
- Rebuilding a dead node is faster because it involves every other node in the cluster and because data is sent to the replacement node incrementally instead of waiting until the end of the validation phase.
- Improves the use of heterogeneous machines in a cluster. You can assign a proportional number of vnodes to smaller and larger machines.

How data is distributed across a cluster (using virtual nodes)

Prior to version 1.2, you had to calculate and assign a single token to each node in a cluster. Each token determined the node's position in the ring and its portion of data according to its hash value. Although the design of consistent hashing used prior to version 1.2 (compared to other distribution designs), allowed moving a single node's worth of data when adding or removing nodes from the cluster, it still required substantial effort to do so.

Starting in version 1.2, Cassandra changes this paradigm from one token and range per node to many tokens per node. The new paradigm is called virtual nodes (vnodes). Vnodes allow each node to own a large number of small partition ranges distributed throughout the cluster. Vnodes also use consistent hashing to distribute data but using them doesn't require token generation and assignment.



The top portion of the graphic shows a cluster without vnodes. In this paradigm, each node is assigned a single token that represents a location in the ring. Each node stores data determined by mapping the partition key to a token value within a range from the previous node to its assigned value. Each node also contains copies of each row from other nodes in the cluster. For example, range E replicates to nodes 5, 6, and 1. Notice that a node owns exactly one contiguous partition range in the ring space.

The bottom portion of the graphic shows a ring with vnodes. Within a cluster, virtual nodes are randomly selected and non-contiguous. The placement of a row is determined by the hash of the partition key within many smaller partition ranges belonging to each node.

Partitioners

A partitioner determines how data is distributed across the nodes in the cluster (including replicas). Basically, a partitioner is a hash function for computing the token (it's hash) of a partition key. Each row of data is uniquely identified by a partition key and distributed across the cluster by the value of the token.

Both the Murmur3Partitioner and RandomPartitioner use tokens to help assign equal portions of data to each node and evenly distribute data from all the tables throughout the ring or other grouping, such as a keyspace. This is true even if the tables use different partition keys, such as usernames or timestamps. Moreover, the read and write requests to the cluster are also evenly distributed and load balancing is simplified because each part of the hash range receives an equal number of rows on average.

Cassandra offers the following partitioners:

• **Murmur3Partitioner** (default):

The Murmur3Partitioner provides faster hashing and improved performance than the previous default partitioner (RandomPartitioner). You can only use Murmur3Partitioner for new clusters; you cannot change the partitioner in existing clusters. If you are switching to the 1.2 cassandra.yaml be sure to change the partitioner setting to match the previous partitioner. The Murmur3Partitioner uses the MurmurHash function. This hashing function creates a 64-bit hash value of the partition key. The possible range of hash values is from -263 to +263. When using the Murmur3Partitioner, you can page through all rows using the token function in a CQL query.

- **RandomPartitioner**:
  The default partitioner prior to Cassandra 1.2. Although no longer the default partitioner, you can use the RandomPartitioner in version 1.2, even when using virtual nodes (vnodes). However, if you don't use vnodes, you must calculate the tokens, as described in Generating tokens. The RandomPartition distributes data evenly across the nodes using an MD5 hash value of the row key. The possible range of hash values is from 0 to 2127 -1. When using the RandomPartitioner, you can page through all rows using the token function in a CQL query.
- **ByteOrderedPartitioner**:
  Cassandra provides the ByteOrderedPartitioner for ordered partitioning. This partitioner orders rows lexically by key bytes. You calculate tokens by looking at the actual values of your partition key data and using a hexadecimal representation of the leading character(s) in a key. For example, if you wanted to partition rows alphabetically, you could assign an A token using its hexadecimal representation of 41. Using the ordered partitioner allows ordered scans by primary key. This means you can scan rows as though you were moving a cursor through a traditional index. For example, if your application has user names as the partition key, you can scan rows for users whose names fall between Jake and Joe. This type of query is not possible using randomly partitioned partition keys because the keys are stored in the order of their MD5 hash (not sequentially). Although having the capability to do range scans on rows sounds like a desirable feature of ordered partitioners, there are ways to achieve the same functionality using table indexes.

The Murmur3Partitioner is the default partitioning strategy for new Cassandra clusters and the right choice for new clusters in almost all cases. Set the partitioner in the cassandra.yaml file:
- **Murmur3Partitioner**: org.apache.cassandra.dht.Murmur3Partitioner
- **RandomPartitioner**: org.apache.cassandra.dht.RandomPartitioner
- **ByteOrderedPartitioner**: org.apache.cassandra.dht.ByteOrderedPartitioner

Data replication

Cassandra stores replicas on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines the nodes where replicas are placed.

The total number of replicas across the cluster is referred to as the replication factor. A replication factor of 1 means that there is only one copy of each row on one node. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica. As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes later. When replication factor exceeds the number of nodes, writes are rejected, but reads are served as long as the desired consistency level can be met.

Two replication strategies are available:

- **SimpleStrategy:**
  Use only for a single data center. SimpleStrategy places the first replica on a node determined by the partitioner. Additional replicas are placed on the next nodes clockwise in the ring without considering topology (rack or data center location).
- **NetworkTopologyStrategy:**
  Use NetworkTopologyStrategy when you have (or plan to have) your cluster deployed across multiple data centers. This strategy specify how many replicas you want in each data center. NetworkTopologyStrategy places replicas in the same data center by walking the ring clockwise until reaching the first node in another rack. NetworkTopologyStrategy attempts to place replicas on distinct racks because nodes in the same rack (or similar physical grouping) often fail at the same time due to power, cooling, or network issues.

  When deciding how many replicas to configure in each data center, the two primary considerations are (1) being able to satisfy reads locally, without incurring cross data-center latency, and (2) failure scenarios. The two most common ways to configure multiple data center clusters are:
  - Two replicas in each data center: This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of ONE.
  - Three replicas in each data center: This configuration tolerates either the failure of a one node per replication group at a strong consistency level of LOCAL_QUORUM or multiple node failures per data center using consistency level ONE.

  Asymmetrical replication groupings are also possible. For example, you can have three replicas per data center to serve real-time application requests and use a single replica for running analytics.

Snitches

A snitch determines which data centers and racks are written to and read from.

Snitches inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into data centers and racks. All nodes must have exactly the same snitch configuration. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).
Note: If you change the snitch after data is inserted into the cluster, you must run a full repair, since the snitch affects where replicas are placed.

Dynamic snitching
Monitors the performance of reads from the various replicas and chooses the best replica based on this history. By default, all snitches also use a dynamic snitch layer that monitors read latency and, when possible, routes requests away from poorly-performing nodes. The dynamic snitch is enabled by default and is recommended for use in most deployments. For information on how this works, see Dynamic snitching in Cassandra: past, present, and future. Configure dynamic snitch thresholds for each node in the cassandra.yaml configuration file.

SimpleSnitch
The SimpleSnitch (the default) does not recognize data center or rack information. Use it for single-data center deployments (or single-zone in public clouds). Using a SimpleSnitch, you define the keyspace to use SimpleStrategy and specify a replication factor.

RackInferringSnitch
The RackInferringSnitch determines the location of nodes by rack and data center, which are assumed to correspond to the 3rd and 2nd octet of the node's IP address, respectively. Use this snitch as an example of writing a custom Snitch class.

PropertyFileSnitch
Determines the location of nodes by rack and data center. This snitch uses a user-defined description of the network details located in the cassandra-topology.properties file. Use this snitch when your node IPs are not uniform or if you have complex replication grouping requirements. When using this snitch, you can define your data center names to be whatever you want. Make sure that the data center names you define correlate to the name of your data centers in your keyspace definition. Every node in the cluster should be described in the cassandra-topology.properties file, and this file should be exactly the same on every node in the cluster. The location of the cassandra-topology.properties file depends on the type of installation; see Install locations.

If you had non-uniform IPs and two physical data centers with two racks in each, and a third logical data center for replicating analytics data, the cassandra-topology.properties file might look like this:

```
# Data Center One
175.56.12.105 =DC1:RAC1
175.50.13.200 =DC1:RAC1
175.54.35.197 =DC1:RAC1
120.53.24.101 =DC1:RAC2
120.55.16.200 =DC1:RAC2
120.57.102.103 =DC1:RAC2
# Data Center Two
110.56.12.120 =DC2:RAC1
110.50.13.201 =DC2:RAC1
110.54.35.184 =DC2:RAC1
50.33.23.120 =DC2:RAC2
50.45.14.220 =DC2:RAC2
50.17.10.203 =DC2:RAC2
# Analytics Replication Group
172.106.12.120 =DC3:RAC1
172.106.12.121 =DC3:RAC1
172.106.12.122 =DC3:RAC1
# default for unknown nodes
default =DC3:RAC1
```

GossipingPropertyFileSnitch
The GossipingPropertyFileSnitch defines a local node's data center and rack; it uses gossip for propagating this information to other nodes. The conf/cassandra-rackdc.properties file defines the default data center and rack used by this snitch:
dc =DC1
rack =RAC1
The location of the conf directory depends on the type of installation; see Install locations. To migrate from the PropertyFileSnitch to the GossipingPropertyFileSnitch, update one node at a time to allow gossip time to propagate. The PropertyFileSnitch is used as a fallback when cassandra-topologies.properties is present.

EC2Snitch
Use the EC2Snitch for simple cluster deployments on Amazon EC2 where all nodes in the cluster are within a single region. The region is treated as the data center and the availability zones are treated as racks within the data center. For example, if a node is in us-east-1a, us-east is the data center name and 1a is the rack location. Because private IPs are used, this snitch does not work across multiple regions.

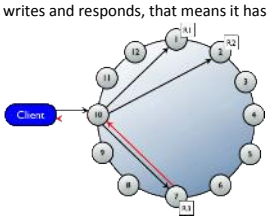
EC2MultiRegionSnitch
Use the EC2MultiRegionSnitch for deployments on Amazon EC2 where the cluster spans multiple regions. As with the EC2Snitch, regions are treated as data centers and availability zones are treated as racks within a data center. For example, if a node is in us-east-1a, us-east is the data center name and 1a is the rack location.

Client requests

1. **Write request**
   The coordinator sends a write request to all replicas that own the row being written. As long as all replica nodes are up and available, they will get the write regardless of the consistency level specified by the client. The write consistency level determines how many replica nodes must respond with a success acknowledgment in order for the write to be considered successful. Success means that the data was written to the commit log and the memtable as described in About writes.

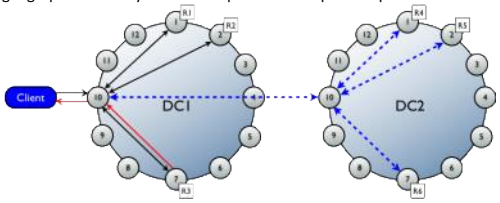
   For example, in a single data center 10 node cluster with a replication factor of 3, an incoming write will go to all 3 nodes that own the requested row. If the write consistency level specified by the client is ONE, the first node to complete the write responds back to the coordinator, which then proxies the success message back to the client. A consistency level of ONE means that it is possible that 2 of the 3 replicas could miss the write if they happened to be down at the time the request was made. If a replica misses a write, Cassandra will make the row consistent later using one of its built-in repair mechanisms: hinted handoff, read repair, or anti-entropy node repair.

   That node forwards the write to all replicas of that row. It will respond back to the client once it receives a write acknowledgment from the number of nodes specified by the consistency level. when a node writes and responds, that means it has written to the commit log and puts the mutation into a memtable.



2. **Multiple data center write requests**
   In multiple data center deployments, Cassandra optimizes write performance by choosing one coordinator node in each remote data center to handle the requests to replicas within that data center. The coordinator node contacted by the client application only needs to forward the write request to one node in each remote data center.

   If using a consistency level of ONE or LOCAL_QUORUM, only the nodes in the same data center as the coordinator node must respond to the client request in order for the request to succeed. This way, geographical latency does not impact client request response times.



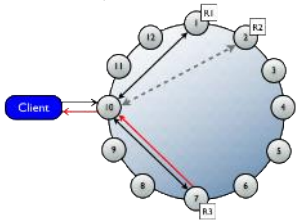
3. **Read requests**
   There are two types of read requests that a coordinator can send to a replica:
   - A direct read request
   - A background read repair request

   The number of replicas contacted by a direct read request is determined by the consistency level specified by the client. Background read repair requests are sent to any additional replicas that did not receive a direct request. Read repair requests ensure that the requested row is made consistent on all replicas.

   Thus, the coordinator first contacts the replicas specified by the consistency level. The coordinator sends these requests to the replicas that are currently responding the fastest. The nodes contacted respond with the requested data; if multiple nodes are contacted, the rows from each replica are compared in memory to see if they are consistent. If they are not, then the replica that has 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 frequently-read data, the coordinator also contacts and compares the data from all the remaining replicas that own the row in the background. If the replicas are inconsistent, the coordinator issues writes to the out-of-date replicas to update the row to the most recent values. This process is known as read repair. Read repair can be configured per table (using read_repair_chance), and is enabled by default.

   For example, in a cluster with a replication factor of 3, and a read consistency level of QUORUM, 2 of the 3 replicas for the given row are contacted to fulfill the read request. Supposing the contacted replicas had different versions of the row, the replica with the most recent version would return the requested data. In the background, the third replica is checked for consistency with the first two, and if needed, the most recent replica issues a write to the out-of-date replicas.



Configuration

1. **The cassandra.yaml configuration file**
   The configuration properties are grouped into the following sections:
   - **Initialization properties** : Controls how a node is configured within a cluster, including inter-node communication, data partitioning, and replica placement.
   - **Global row and key caches properties** : Caching parameters for tables.
   - **Performance tuning properties** : Tuning performance and system resource utilization, including memory, disk I/O, CPU, reads, and writes.
   - **Binary and RPC protocol timeout properties** : Timeout settings for the binary protocol.
   - **Remote procedure call tuning (RPC) properties** : Settings for configuring and tuning RPCs (client connections).
   - **Fault detection properties** : Settings to handle poorly performing or failing nodes.

- **Automatic backup properties** : Automatic backup settings.
- **Security properties** : Server and client security settings.

Details: http://www.datastax.com/documentation/cassandra/1.2/cassandra/configuration/configCassandra_yaml_r.html

Security

1. **Securing Cassandra**
2. **SSL encryption**
3. **Internal authentication**
4. **Internal authorization**
5. **Configuring firewall port access**

   Which ports to open when nodes are protected by a firewall.

   If you have a firewall running on the nodes in your Cassandra cluster, you must open up the following ports to allow communication between the nodes, including certain Cassandra ports. If this isn't done, when you start Cassandra on a node, the node acts as a standalone database server rather than joining the database cluster.

Public ports	
Port number	**Description**
22	SSH port
8888	OpsCenter website. The opscenterd daemon listens on this port for HTTP requests coming directly from the browser.
Cassandra inter-node ports	
Port number	**Description**
7000	Cassandra inter-node cluster communication.
7001	Cassandra SSL inter-node cluster communication.
7199	Cassandra JMX monitoring port.
Cassandra client ports	
Port number	**Description**
9042	Cassandra client port.
9160	Cassandra client port (Thrift).
Cassandra OpsCenter ports	
Port number	**Description**
61620	OpsCenter monitoring port. The opscenterd daemon listens on this port for TCP traffic coming from the agent.
61621	OpsCenter agent port. The agents listen on this port for SSL traffic initiated by OpsCenter.