

Apache Cassandra™

WRITTEN BY BRIAN O'NEIL ARCHITECT, IRON MOUNTAIN
UPDATED BY MILAN MILOSEVIC LEAD DATA ENGINEER IN SMARTCAT

CONTENTS

- INTRODUCTION
- WHO IS USING CASSANDRA?
- DATA MODEL OVERVIEW
- SCHEMA
- TABLE
- ROWS
- COLUMNS
- CASSANDRA ARCHITECTURE
- PARTITIONING
- RANDOM PARTITIONER
- AND MORE...

INTRODUCTION

Apache Cassandra is a high-performance, extremely scalable, fault-tolerant (i.e., no single point of failure), distributed non-relational database solution. Cassandra combines all the benefits of Google Bigtable and Amazon Dynamo to handle the types of database management needs that traditional RDBMS vendors cannot support. DataStax is the leading worldwide commercial provider of Cassandra products, services, support, and training.

WHO IS USING CASSANDRA?

Cassandra is in use at [Apple](#) (75,000+ nodes), [Spotify](#) (3,000+ nodes), [eBay](#), [Capital One](#), [Macy's](#), [Bank of America](#), [Netflix](#), [Twitter](#), [Urban Airship](#), [Constant Contact](#), [Reddit](#), [Cisco](#), [OpenX](#), [Rackspace](#), [Ooyala](#), and [more companies](#) that have large active data sets. The largest known Cassandra cluster has more than 300 TB of data across more than 400 machines ([cassandra.apache.org](#)).

RDBMS VS. CASSANDRA

	CASSANDRA	RDBMS
Atomicity	Success or failure for inserts/deletes in a single partition (one or more rows in a single partition).	Enforced at every scope, at the cost of performance and scalability.
Sharding	Native share-nothing architecture, inherently partitioned by a configurable strategy.	Often forced when scaling, partitioned by key or function.

Consistency	No tunable consistency in the ACID sense. Can be tuned to provide more consistency or to provide more availability. The consistency is configured per request. Since Cassandra is a distributed database, traditional locking and transactions are not possible (there is, however, a concept of lightweight transaction that should be used very carefully).	Favors consistency over availability tunable via isolation levels.
-------------	---	--

TABLE CONTINUED ON PAGE THREE

DATASTAX®

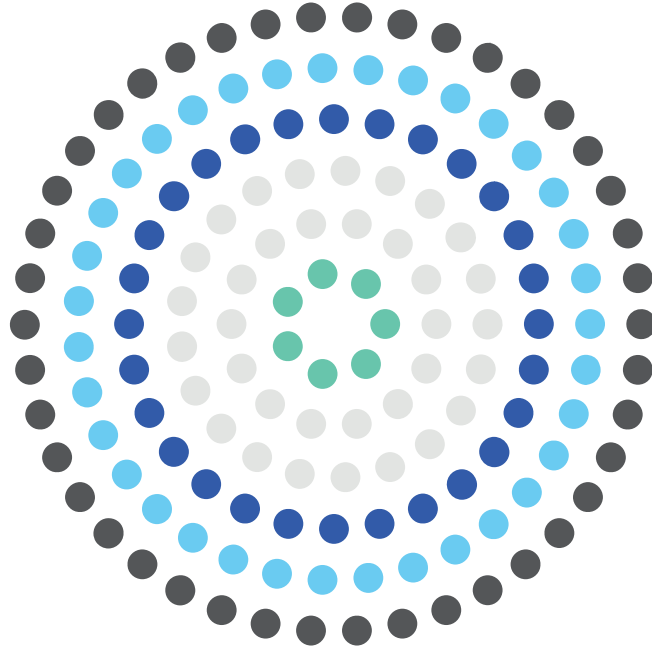
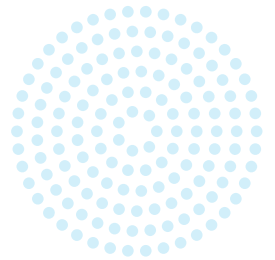
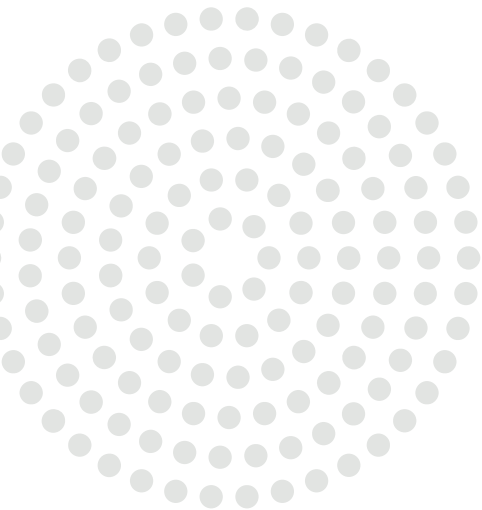
Relational Databases
Had Their Day.

Now and Tomorrow,
NoSQL Is the Way.

Explore NoSQL Cassandra

Relational Databases Had Their Day.

Now and Tomorrow, NoSQL Is the Way.



Explore NoSQL Cassandra

Durability	Writes are durable to a replica node, being recorded in memory and the commit log before acknowledged. In the event of a crash, the commit log replays on restart to recover any lost writes before data is flushed to disk.	Typically, data is written to a single master node, sometimes configured with synchronous replication at the cost of performance and cumbersome data restoration.
Multi-Datcenter Replication	Native and out-of-the-box capabilities for data replication over lower bandwidth, higher latency, less reliable connections.	Typically, only limited long-distance replication to read-only slaves receiving asynchronous updates.
Security	Coarse-grained and primitive, but authorization, authentication, roles, and data encryption are provided out-of-the-box.	Fine-grained access control to objects.

DATA MODEL OVERVIEW

Cassandra has a tabular schema comprising keyspaces, tables, partitions, rows, and columns.

Note that, since Cassandra 3.x terminology is altered due to changes in the storage engine, a “column family” is now a table and a “row” is now a partition.

	DEFINITION	RDBMS ANALOGY	OBJECT EQUIVALENT
Schema/ Keyspace	A collection of tables.	Schema/ Database	Set
Table/ Column Family	A set of partitions.	Table	Map
Partition	A set of rows that share the same partition key.	N/A	
Row	An ordered (inside of a partition) set of columns.	Row	OrderedMap
Column	A key/value pair and timestamp.	Column (Name, Value)	(Key, Value, Timestamp)

SCHEMA

The keyspace is akin to a database or schema in RDBMS, contains a set of tables, and is used for replication. A keyspace is also the

unit for Cassandra's access control mechanism. When enabled, users must authenticate to access and manipulate data in a schema or table.

TABLE

A table, previously known as a column family, is a map of rows. Similar to RDBMS, a table is defined by a primary key. The primary key consists of a partition key and clustering columns. The partition key defines data locality in the cluster, and the data with the same partition key will be stored together on a single node. The clustering columns define how the data will be ordered on the disk within a partition. The client application provides rows that conform to the schema. Each row has the same fixed subset of columns.

As values for these properties, Cassandra provides the following CQL data types for columns ([DataStax documentation](#)):

TYPE	PURPOSE	STORAGE
ascii	Efficient storage for simple ASCII strings.	Arbitrary number of ASCII bytes (i.e., values are 0-127).
boolean	True or false.	Single byte.
blob	Arbitrary byte content.	Arbitrary number of bytes.
counter	Used for counters, which are cluster-wide incrementing values.	8 bytes.
timestamp	Stores time in milliseconds.	8 bytes.
time	Value is encoded as a 64-bit signed integer representing the number of nanoseconds since midnight. Values can be represented as strings, such as 13:30:54.234.	64-bit signed integer.
date	Value is a date with no corresponding time value; Cassandra encodes date as a 32-bit integer representing days since epoch (January 1, 1970). Dates can be represented in queries and inserts as a string, such as 2015-05-03 (yyyy-mm-dd).	32-bit integers.

TABLE CONTINUED ON NEXT PAGE

decimal	Stores BigDecimals.	4 bytes to store the scale, plus an arbitrary number of bytes to store the value.
double	Stores Doubles.	8 bytes.
float	Stores Floats.	4 bytes.
tinyint	Stores 1-byte integer.	1 byte.
smallint	Stores 2-byte integer.	2 bytes.
int	Stores 4-byte integer.	4 bytes.
varint	Stores variable precision integer.	An arbitrary number of bytes used to store the value.
bigint	Stores Longs.	8 bytes.
text, varchar	Stores text as UTF-8.	UTF-8.
timeuuid	Version 1 UUID only	16 bytes.
uuid	Suitable for UUID storage.	16 bytes.
frozen	A frozen value serializes multiple components into a single value. Non-frozen types allow updates to individual fields. Cassandra treats the value of a frozen type as a blob. The entire value must be overwritten.	N/A
inet	IP address string in IPv4 or IPv6 format, used by the python-cql driver and CQL native protocols.	N/A
list	A collection of one or more ordered elements: [literal, literal, literal].	N/A
map	A JSON-style array of literals: { literal : literal, literal : literal ... }.	N/A

set	A collection of one or more elements: { literal, literal, literal }.	N/A
tuple	A group of 2-3 fields.	N/A

ROWS

Cassandra 3.x supports tables defined with composite primary keys. The first part of the primary key is a partition key. Remaining columns are clustering columns and define the order of the data on the disk.

For example, let's say there is a table called `users_by_location` with the following primary key:

```
((country, town), birth_year, user_id)
```

In that case, the `(country, town)` pair is a partition key (a composite one). All the users with the same `(country, town)` values will be stored together on a single node and replicated together based on the replication factor. The rows within the partition will be ordered by `birth_year` and then by `user_id`. The `user_id` column provides uniqueness for the primary key.

If the partition key is not separated by parentheses, then the first column in the primary key is considered a partition key.

For example, if the primary key is defined by `(country, town, birth_year, user_id)`, then `country` would be the partition key and `town` would be a clustering column.

COLUMNS

A column is a triplet: key, value, and timestamp. The validation and comparator on the column family define how Cassandra sorts and stores the bytes in column keys.

The timestamp portion of the column is used to sequence mutations. The timestamp is defined and specified by the client. Newer versions of Cassandra drivers provide this functionality out of the box. Client application servers should have synchronized clocks.

Columns may optionally have a time-to-live (TTL), after which Cassandra asynchronously deletes them. Note that TTLs are defined per cell, so each cell in the row has an independent time-to-live and is handled by the Cassandra independently.

HOW DATA IS STORED ON DISK

Using the `sstabledump` tool, you can inspect how the data is stored on the disk. This is very important if you want to develop intuition

about data modeling, reads, and writes in Cassandra. Here is an example from a [DataStax blog post](#). Given the table defined by:

```
CREATE TABLE IF NOT EXISTS symbol_history (
  symbol    text,
  year      int,
  month     int,
  day       int,
  volume    bigint,
  close     double,
  open      double,
  low       double,
  high      double,
  idx       text static,
  PRIMARY KEY ((symbol, year), month, day)
) with CLUSTERING ORDER BY (month desc, day desc);
```

The data (when deserialized into JSON using the `sstabledump` tool) is stored on the disk in this form:

```
[
  {
    "partition": {
      "key": [ "CORP", "2016" ],
      "position": 0
    },
    "rows": [
      {
        "type": "static_block",
        "position": 48,
        "cells": [
          { "name": "idx", "value": "NASDAQ",
            "tstamp": 1457484225583260, "ttl": 604800, "expires_at": 1458089025, "expired": false }
        ]
      },
      {
        "type": "row",
        "position": 48,
        "clustering": [ "1", "5" ],
        "deletion_info": { "deletion_time": 1457484273784615, "tstamp": 1457484273 }
      },
      {
        "type": "row",
        "position": 66,
        "clustering": [ "1", "4" ],
        "liveness_info": { "tstamp": 1457484225586933, "ttl": 604800, "expires_at": 1458089025, "expired": false },
        "cells": [
          { "name": "close", "value": "8.54" },
          { "name": "high", "value": "8.65" },
          { "name": "low", "value": "8.2" },
          { "name": "open", "value": "8.2" },
          { "name": "volume", "value": "1054342" }
        ]
      },
      {
        "type": "row",
        "position": 131,
```

```
        "clustering": [ "1", "1" ],
        "liveness_info": { "tstamp": 1457484225583260, "ttl": 604800, "expires_at": 1458089025, "expired": false },
        "cells": [
          { "name": "close", "value": "8.2" },
          { "name": "high", "deletion_time": 1457484267368678, "tstamp": 1457484267, "value": "8.02" },
          { "name": "open", "value": "9.33" },
          { "name": "volume", "value": "1055334" }
        ]
      }
    ],
    {
      "partition": {
        "key": [ "CORP", "2015" ],
        "position": 194
      },
      "rows": [
        {
          "type": "static_block",
          "position": 239,
          "cells": [
            { "name": "idx", "value": "NYSE", "tstamp": 1457484225578370, "ttl": 604800, "expires_at": 1458089025, "expired": false }
          ]
        },
        {
          "type": "row",
          "position": 239,
          "clustering": [ "12", "31" ],
          "liveness_info": { "tstamp": 1457484225578370, "ttl": 604800, "expires_at": 1458089025, "expired": false },
          "cells": [
            { "name": "close", "value": "9.33" },
            { "name": "high", "value": "9.57" },
            { "name": "low", "value": "9.21" },
            { "name": "open", "value": "9.55" },
            { "name": "volume", "value": "1054342" }
          ]
        }
      ]
    }
  ]
}
```

CASSANDRA ARCHITECTURE

Cassandra uses a masterless ring architecture. The ring represents a cyclic range of token values (i.e., the token space). Since Cassandra 2.0, each node is responsible for a number of small token ranges defined by the `num_tokens` property in `cassandra.yml`.

PARTITIONING

Keys are mapped into the token space by a partitioner. The important distinction between the partitioners is order preservation (OP). Users can define their own partitioners by implementing IPartitioner, or they can use one of the native partitioners.

	CASSANDRA	RDBMS	OP
Murmur3Partitioner	MurmurHash	64-bit hash value	No
RandomPartitioner	MD5	BigInteger	No
BytesOrderPartitioner	Identity	Bytes	Yes

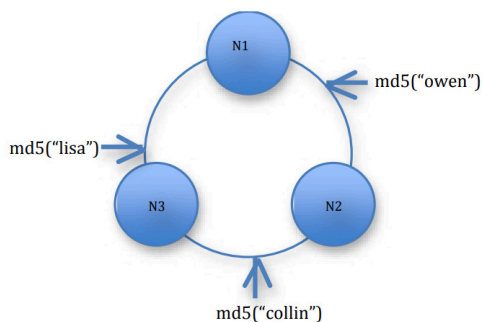
The following examples illustrate this point.

RANDOM PARTITIONER

Since the Random Partitioner uses an MD5 hash function to map keys into tokens, on average those keys will evenly distribute across the cluster. The row key determines the node placement.

ROW KEY			
lisa	state: CA	graduated: 2008	gender: F
owen	state: TX	gender: M	
collin	state: UT	gender: M	

This may result in the following ring formation, where "collin," "owen," and "lisa" are row keys.



With Cassandra's storage model, where each node owns the preceding token space, this results in the following storage allocation based on the tokens:

ROW KEY	MD5 HASH	NODE
collin	CC982736AD62AB	3
owen	9567238FF72635	2
lisa	001AB62DE123FF	1

Notice that the keys are not in order. With RandomPartitioner, the

keys are evenly distributed across the ring using hashes, but you sacrifice order, which means any range query needs to query all nodes in the ring.

MURMUR3 PARTITIONER

Murmur3 partitioner is the default partitioner since Cassandra 1.2. The Murmur3Partitioner provides faster hashing and improved performance than the RandomPartitioner. The Murmur3Partitioner can be used with vnodes.

ORDER PRESERVING PARTITIONERS (OPP)

The Order Preserving Partitioners preserve the order of the row keys as they are mapped into the token space.

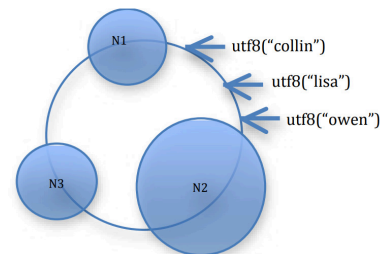
In our example, since:

```
"collin" < "lisa" < "owen"
```

then,

```
token("collin") < token("lisa") < token("owen")
```

With OPP, range queries are simplified, and a query may not need to consult each node in the ring. This seems like an advantage, but it comes at a price. Since the partitioner is preserving order, the ring may become unbalanced unless the row keys are naturally distributed across the token space, as illustrated below:



To manually balance the cluster, you can set the initial token for each node in the Cassandra configuration.

Hot tip: If possible, it is best to design your data model to use Murmur3 Partitioner to take advantage of the automatic load balancing and decreased administrative overhead of manually managing token assignment.

REPLICATION

Cassandra provides continuous, high availability and fault tolerance through data replication. The replication uses the ring to determine nodes used for replication. Replication is configured on the keyspace level. Each keyspace has an independent replication factor, *n*. When writing information, the data is written to the target node as determined by the partitioner and *n-1* subsequent nodes along the ring.

There are two replication strategies: SimpleStrategy and NetworkTopologyStrategy.

SIMPLESTRATEGY

The SimpleStrategy is the default strategy and blindly writes the data to subsequent nodes along the ring. This strategy is NOT RECOMMENDED for a production environment.

In the previous example with a replication factor of 2, this would result in the following storage allocation:

ROW KEY	REPLICA 1 (AS DETERMINED BY PARTITIONER)	REPLICA 2 (FOUND BY TRAVERSING THE RING)
collin	3	1
owen	2	3
lisa	1	2

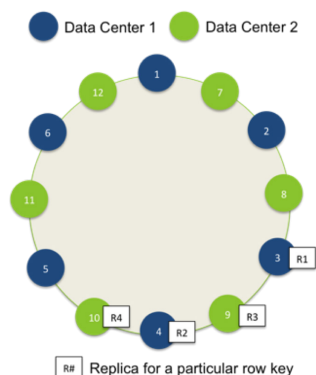
NETWORKTOPOLOGYSTRATEGY

The NetworkTopologyStrategy is useful when deploying to multiple data centers. It ensures data is replicated across data centers.

Effectively, the NetworkTopologyStrategy executes the SimpleStrategy independently for each data center, spreading replicas across distant racks. Cassandra writes a copy in each data center as determined by the partitioner.

Data is written simultaneously along the ring to subsequent nodes within that data center with preference for nodes in different racks to offer resilience to hardware failure. All nodes are peers and data files can be loaded through any node in the cluster, eliminating the single point of failure inherent in master-slave architecture and making Cassandra fully fault-tolerant and highly available.

See the following ring and deployment topology:



With blue nodes deployed to one data center (DC1), green nodes deployed to another data center (DC2), and a replication factor of

two per each data center, one row will be replicated twice in Data Center 1 (R1, R2) and twice in Data Center 2 (R3, R4).

NOTE: Cassandra attempts to write data simultaneously to all target nodes, then waits for confirmation from the relevant number of nodes needed to satisfy the specified consistency level.

CONSISTENCY LEVELS

One of the unique characteristics of Cassandra that sets it apart from other databases is its approach to consistency. Clients can specify the consistency level on both read and write operations, trading off between high availability, consistency, and performance.

WRITE

LEVEL	EXPECTATION
ANY	The write was written in at least one node's commit log. Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and continuous availability.
LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.
ONE	A write is successfully acknowledged by at least one replica (in any DC).
TWO	A write is successfully acknowledged by at least two replicas.
THREE	A write is successfully acknowledged by at least three replicas.
QUORUM	A write is successfully acknowledged by at least $n/2+1$ replicas, where n is the replication factor.
LOCAL_QUORUM	A write is successfully acknowledged by at least $n/2+1$ replicas within the local data center.
EACH_QUORUM	A write is successfully acknowledged by at least $n/2+1$ replicas within each data center.
ALL	A write is successfully acknowledged by all n replicas. This is useful when absolute read consistency and/or fault tolerance are necessary (e.g., online disaster recovery).

READ

LEVEL	EXPECTATION
ONE	Returns a response from the closest replica, as determined by the <code>snitch</code> .
TWO	Returns the most recent data from two of the closest replicas.
THREE	Returns the most recent data from three of the closest replicas.
QUORUM	Returns the record after a quorum ($n/2 + 1$) of replicas from all data centers that responded.
LOCAL_QUORUM	Returns the record after a quorum of replicas in the current data center, as the coordinator has reported. Avoids latency of communication among data centers.
EACH_QUORUM	Not supported for reads.
ALL	The client receives the most current data once all replicas have responded.

NETWORK TOPOLOGY

As input into the replication strategy and to efficiently route communication, Cassandra uses a *snitch* to determine the data center and rack of the nodes in the cluster. A snitch is a component that detects and informs Cassandra about the network topology of the deployment.

The snitch dictates what is used in the strategy options to identify replication groups when configuring replication for a keyspace.

The following table shows the snitches provided by Cassandra and what you should use in your keyspace configuration for each snitch:

SNITCH	SPECIFY
SimpleSnitch	Specify only the replication factor in your strategy options.
PropertyFileSnitch	Specify the data center names from your properties file in the keyspace strategy options.
GossipingPropertyFileSnitch	Returns the most recent data from three of the closest replicas.

TABLE CONTINUED IN NEXT COLUMN

RackInferringSnitch	Specify the second octet of the IPv4 address in your keyspace strategy options.
EC2Snitch	Specify the region name in the keyspace strategy options and <code>dc_suffix</code> in <code>cassandra-rackdc.properties</code> .
Ec2MultiRegionSnitch	Specify the region name in the keyspace strategy options and <code>dc_suffix</code> in <code>cassandra-rackdc.properties</code> .
GoogleCloudSnitch	Specify the region name in the keyspace strategy options.

SIMPLESNITCH

The SimpleSnitch provides Cassandra no information regarding racks or datacenters. It is the default setting and is useful for simple deployments where all servers are collocated. It is not recommended for a production environment, as it does not provide failure tolerance.

PROPERTYFILESNITCH

The PropertyFileSnitch allows users to be explicit about their network topology. The user specifies the topology in a properties file, `cassandra-topology.properties`. The file specifies which nodes belong to which racks and data centers. Below is an example property file for our sample cluster:

```
# DC1
192.168.0.1=DC1:RAC1
192.168.0.2=DC1:RAC1
192.168.0.3=DC1:RAC2

# DC2
192.168.1.4=DC2:RAC3
192.168.1.5=DC2:RAC3
192.168.1.6=DC2:RAC4

# Default for nodes
default=DC3:RAC5
```

GOSSIPINGPROPERTYFILESNITCH

This snitch is recommended for production. It uses rack and datacenter information for the local node defined in the `cassandra-rackdc.properties` file and propagates this information to other nodes via gossip.

Unlike PropertyFileSnitch, which contains topology for the entire cluster on every node, GossipingPropertyFileSnitch contains DC

and rack information only for the local node. Each node describes and gossips its location to other nodes.

Example contents of the `cassandra-rackdc.properties` files:

```
dc=DC1
rack=RACK1
RackInferringSnitch
```

THE RACKINFERRINGSNITCH

The `RackInferringSnitch` infers network topology by convention.

From the IPv4 address (e.g., 9.100.47.75), the snitch uses the following convention to identify the datacenter and rack:

OCTET	EXAMPLE	INDICATES
1	9	Nothing
2	100	Data-center
3	47	Rack
4	75	Node

EC2SNITCH

The `EC2Snitch` is useful for deployments to Amazon's EC2. It uses Amazon's API to examine the regions to which nodes are deployed. It then treats each region as a separate datacenter.

EC2MULTIREGIONSNITCH

Use this snitch for deployments on Amazon EC2 where the cluster spans multiple regions. This snitch treats datacenters and availability zones as racks within a datacenter and uses public IPs as `broadcast_address` to allow cross-region connectivity. Cassandra nodes in one EC2 region can bind to nodes in another region, thus enabling multi-datacenter support.

Hot tip: Pay attention to which snitch you are using and which file you are using to define the topology. Some of the snitches use `cassandra-topology.properties` file and the other, newer ones use `cassandra-rackdc.properties` file.

QUERYING/INDEXING

Cassandra provides simple primitives. Its simplicity allows it to scale linearly with continuous, high availability and very little performance degradation. That simplicity allows for extremely fast read and write operations for specific keys, but servicing more sophisticated queries that span keys requires pre-planning.

Using the primitives that Cassandra provides, you can construct indexes that support exactly the query patterns of your application. Note, however, that queries may not perform well without properly designing your schema.

SECONDARY INDEXES

To satisfy simple query patterns, Cassandra provides a native indexing capability called Secondary Indexes. A column family may have multiple secondary indexes. A secondary index is hash-based and uses specific columns to provide a reverse lookup mechanism from a specific column value to the relevant row keys.

Under the hood, Cassandra maintains hidden column families that store the index. The strength of Secondary Indexes is allowing queries by value. Secondary indexes are built in the background automatically without blocking reads or writes. To create a Secondary Index using CQL is straightforward.

For example, you can define a table of data about movie fans, and then create a secondary index of states where they live:

```
CREATE TABLE fans ( watcherID uuid, favorite_actor
text, address text, zip int, state text PRIMARY KEY
(watcherID) );

CREATE INDEX watcher_state ON fans (state);
```

Hot tip: Try to avoid indexes whenever possible. It is (almost) always a better idea to denormalize data and create a separate table that satisfies a particular query than it is to create an index.

RANGE QUERIES

It is important to consider partitioning when designing your schema to support range queries.

RANGE QUERIES WITH ORDER PRESERVATION

Since order is preserved, order preserving partitioners better supports range queries across a range of rows. Cassandra only needs to retrieve data from the subset of nodes responsible for that range. For example, if we are querying against a column family keyed by phone number and we want to find all phone numbers between that begin with 215-555, we could create a range query with start key `215-555-0000` and end key `215-555-9999`.

To service this request with `OrderPreservingPartitioning`, it's possible for Cassandra to compute the two relevant tokens: `token(215-555-0000)` and `token(215-555-9999)`.

Then satisfying that querying simply means consulting nodes responsible for that token range and retrieving the rows/tokens in that range.

Hot tip: Try to avoid queries with multiple partitions whenever possible. The data should be partitioned based on the access patterns, so it is a good idea to group the data in a single partition (or several) if such queries exist. If you have too many range queries that cannot be satisfied by looking into several partitions, you may want to rethink whether Cassandra is the best solution for your use case.

RANGE QUERIES WITH RANDOM PARTITIONING

The RandomPartitioner provides no guarantees of any kind between keys and tokens. In fact, ideally row keys are distributed around the token ring evenly. Thus, the corresponding tokens for a start key and end key are not useful when trying to retrieve the relevant rows from tokens in the ring with the RandomPartitioner. Consequently, Cassandra must consult all nodes to retrieve the result. Fortunately, there are well-known design patterns to accommodate range queries. These are described below.

INDEX PATTERNS

There are a few design patterns to implement indexes. Each services different query patterns. The patterns leverage the fact that Cassandra columns are always stored in sorted order and all columns for a single row reside on a single host.

INVERTED INDEXES

First, let's consider the *inverted index* pattern. In an inverted index, columns in one row become row keys in another. Consider the following data set, in which users IDs are row keys:

TABLE: USERS			
PARTITION KEY	ROWS/COLUMNS		
BONE42	{ name : "Brian" }	{ zip: 15283 }	{ dob : 09/19/1982 }
LKEL76	{ name : "Lisa" }	{ zip: 98612 }	{ dob : 07/23/1993 }
COW89	{ name : "Dennis" }	{ zip: 98612 }	{ dob : 12/25/2004 }

Without indexes, searching for users in a specific zip code would mean scanning our Users column family row-by-row to find the users in the relevant zip code. Obviously, this does not perform well.

To remedy the situation, we can create a table that represents the query we want to perform, inverting rows and columns. This would result in the following table:

TABLE: USERS			
PARTITION KEY	ROWS/COLUMNS		
98612	{ user_id : LKEL76 }		
	{ user_id : COW89 }		
15283	{ user_id : BONE42 }		

Since each partition is stored on a single machine, Cassandra can quickly return all user IDs within a single zip code by returning all rows within a single partition. Cassandra simply goes to a single host based on *partition key (zip code)* and returns the contents of that single partition.

TIME SERIES DATA

When working with time series data, consider partitioning data by time unit (hourly, daily, weekly, and so on), depending on the rate of events. That way, all the events in a single period (e.g., one hour) are grouped together and can be fetched and/or filtered based on the clustering columns. `TimeWindowCompactionStrategy` is specifically designed to work with time series data and is recommended in this scenario. The `TimeWindowCompactionStrategy` compacts the all the SSTables in a single partition per time unit. This allows for extremely fast reads of the data in a single time unit because it guarantees that only one SSTable will be read.

DENORMALIZATION

Finally, it is worth noting that each of the indexing strategies as presented would require two steps to service a query if the request requires the actual column data (e.g., user name). The first step would retrieve the keys out of the index. The second step would fetch each relevant column by row key.

We can skip the second step if we denormalize the data. In Cassandra, denormalization is the norm. If we duplicate the data, the index becomes a true materialized view that is custom tailored to the exact query we need to support.

INSERTING/UPDATING/DELETING

Everything in Cassandra is an insert, typically referred to as a mutation. Since Cassandra is effectively a key-value store, operations are simply mutations of a key/value pairs.

HINTED HANDOFF

Similar to ReadRepair, hinted handoff is a background process that ensures data integrity and eventual consistency. If a replica is down

in the cluster, the remaining nodes will collect and temporarily store the data that was intended to be stored on the downed node. If the downed node comes back online soon enough (configured by `max_hint_window_in_ms` option in `cassandra.yml`), other nodes will "hand off" the data to it. This way Cassandra smooths out short network or other outages out of the box.

OPERATIONS AND MAINTENANCE

Cassandra provides tools for operations and maintenance. Some of the maintenance is mandatory because of Cassandra's eventually consistent architecture. Other facilities are useful to support alerting and statistics gathering.

Use `nodetool` to manage Cassandra. Datastax provides a reference card on `nodetool`, which is available here: https://docs.datastax.com/en/dse/6.7/dse-admin/datastax_enterprise/tools/nodetool/toolsNodetool.html.

NODETOOL REPAIR

Cassandra keeps record of deleted values for some time to support the eventual consistency of distributed deletes. These values are called tombstones. Tombstones are purged after some time (`GCGraceSeconds`, which defaults to 10 days). Since tombstones prevent improper data propagation in the cluster, you will want to ensure that you have consistency before they get purged.

To ensure consistency, run:

```
>$CASSANDRA_HOME/bin/nodetool repair
```

The repair command replicates any updates missed due to downtime or loss of connectivity. This command ensures consistency across the cluster and obviates the tombstones. You will want to do this periodically on each node in the cluster (within the window before tombstone purge).

The repair process is greatly simplified by using a tool called [Cassandra Reaper](#) (originally developed and open sourced by Spotify but taken over and improved by The Last Pickle).

MONITORING

Cassandra has support for monitoring via JMX, but the simplest way to monitor the Cassandra node is by using OpsCenter, which is designed to manage and monitor Cassandra database clusters. There is a free community edition as well as an enterprise edition that provides management of Apache SOLR and Hadoop.

Simply download `mx4j` and execute the following:

```
cp $MX4J_HOME/lib/mx4j-tools.jar $CASSANDRA_HOME/lib
```

The following are key attributes to track per column family:

ATTRIBUTE	PROVIDES
Read Count	Frequency of reads against the column family.
Read Latency	Latency of reads against the column family.
Write Count	Frequency of writes against the column family.
Write Latency	Latency of writes against the column family.
Pending Tasks	Queue of pending tasks, informative to know if tasks are queuing.

An open-source alternative to OpsCenter can be achieved by combining a few tools:

1. [cassandra-diagnostics](#) or [Jolokia](#) for exposing/shipping JMX metrics.
2. [Grafana](#) or [Prometheus](#) for displaying the metrics.

BACKUP

OpsCenter facilitates backing up data by providing snapshots of the data. A snapshot creates a new hardlink to every live SSTable. Cassandra also provides online backup facilities using `nodetool`. To take a snapshot of the data on the cluster, invoke:

```
$CASSANDRA_HOME/bin/nodetool snapshot
```

This will create a snapshot directory in each keyspace data directory. Restoring the snapshot is then a matter of shutting down the node, deleting the commitlogs and the data files in the keyspace, and copying the snapshot files back into the keyspace directory.

CLIENT LIBRARIES

Cassandra has a very active community developing libraries in different languages.

JAVA

CLIENT	DESCRIPTION
DataStax Java driver for Apache Cassandra	Industry standard for Java driver: github.com/datastax/java-driver
Astyanax	Inspired by Hector, Astyanax is a client library developed by the folks at Netflix: github.com/Netflix/astyanax

C#

CLIENT	DESCRIPTION
DataStax Enterprise	<p>This driver is built on top of DataStax C# driver for Apache Cassandra:</p> <p>https://docs.datastax.com/en/developer/csharp-driver-dse/2.9/</p>

C/C++

CLIENT	DESCRIPTION
DataStax Enterprise	<p>This driver builds on the DataStax C/C++ driver for Apache Cassandra and includes specific features for DSE:</p> <p>https://docs.datastax.com/en/developer/cpp-driver-dse/1.10/</p>

CQL

CLIENT	DESCRIPTION
CQL	<p>Cassandra provides an SQL-like query language called the Cassandra Query Language (CQL). The CQL shell allows you to interact with Cassandra as if it were a SQL database. Start the shell with:</p> <pre>>\$CASSANDRA_HOME/bin/cqlsh</pre> <p>Datastax has a reference card for CQL available here:</p> <p>https://www.datastax.com/sites/default/files/content/technical-guide/2019-09/cqltop10.final_.pdf</p>

PYTHON

CLIENT	DESCRIPTION
Pycassa	<p>Pycassa is the most well-known Python library for Cassandra:</p> <p>github.com/pycassa/pycassa</p>

PHP CQL

CLIENT	DESCRIPTION
Cassandra-PDO	<p>A CQL (Cassandra Query Language) driver for PHP:</p> <p>code.google.com/a/apache-extras.org/p/cassandra-pdo</p>

RUBY

CLIENT	DESCRIPTION
Ruby Gem	<p>Ruby has support for Cassandra via a gem:</p> <p>rubygems.org/gems/cassandra</p>

REST

CLIENT	DESCRIPTION
Virgil	<p>Virgil is a Java-based REST client for Cassandra:</p> <p>github.com/hmsonline/virgil</p>

COMMAND LINE INTERFACE (CLI)

Cassandra also provides a Command Line Interface (CLI), through which you can perform all schema-related changes. It also allows you to manipulate data.

Datastax provides reference cards for both CQL and nodetool, available here: <https://docs.datastax.com/en/dse/6.7/cql/cql/cqlQuickReference.html>.



Written by **Milan Milosevic**, Lead Data Engineer, SmartCat

Milan Milosevic is Lead Data Engineer in SmartCat, where he leads the team of data engineers and data scientists implementing end-to-end machine learning and data-intensive solutions. He is also responsible for designing, implementing, and automating highly available, scalable, distributed cloud architectures (AWS, Apache Cassandra, Ansible, CloudFormation, Terraform, MongoDB). His primary focus is on Apache Cassandra and monitoring, performance tuning, and automation around it.



DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensus, and Sauce Labs.

Devada, Inc.
 600 Park Offices Drive
 Suite 150
 Research Triangle Park, NC 27709

888.678.0399 919.678.0300

Copyright © 2019 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.