CQL for Apache Cassandra™ 2.2 & later

Documentation

August 11, 2016

Contents

Introduction to Cassandra Query Language

About this document

Welcome to the CQL documentation provided by DataStax. To ensure that you get the best experience in using this document, take a moment to look at the Tips for using DataStax documentation.

The landing pages provide information about supported platforms, product compatibility, planning and testing cluster deployments, recommended production settings, troubleshooting, third-party software, resources for additional information, administrator and developer topics, and earlier documentation.

Overview of the Cassandra Query Language

Cassandra Query Language (CQL) is a query language for the Cassandra database. This release of CQL works with Cassandra 3.0 for Linux and Windows and Cassandra 3.x for Linux and Windows.

The Cassandra Query Language (CQL) is the primary language for communicating with the Cassandra database. The most basic way to interact with Cassandra is using the CQL shell, cqlsh. Using cqlsh, you can create keyspaces and tables, insert and query tables, plus much more. If you prefer a graphical tool, you can use DataStax DevCenter. For production, DataStax supplies a number of drivers so that CQL statements can be passed from client to cluster and back. Other administrative tasks can be accomplished using OpsCenter.

Important: This document assumes you are familiar with the Cassandra 2.2 documentation, Cassandra 3.0 documentation or Cassandra 3.x documentation.

Table: CQL for Cassandra 2.2 and later features

New CQL features	• JSON support for CQL3 • User Defined Functions (UDFs) • User Defined Aggregates (UDAs) • Role Based Access Control (RBAC) • Native Protocol v.4 • In Cassandra 3.0 and later, Materialized Views • Addition of CLEAR command for cqlsh • In Cassandra 3.4 and later, SSTable Attached Secondary Indexes (SASI) have been introduced that improve on the existing secondary index implementation with superior performance for queries that previously required the use of `ALLOW FILTERING`.
Improved CQL features	• Additional COPY command options • New WITH ID option with `CREATE TABLE` command • Support IN clause on any partition key column or clustering column • Accept Dollar Quoted Strings • Allow Mixing Token and Partition Key Restrictions • Support Indexing Key/Value Entries on Map Collections • Date data type added and improved time/date conversion functions • Tinyint and smallint data types added • Change to CREATE TABLE syntax for compression options • In Cassandra 3.4 and later, static columns can be indexed. • In Cassandra 3.6 and later, clustering columns can be used in WHERE clause without secondary index.

	• In Cassandra 3.6 and later, individual subfields of a user-defined type (UDT) can be updated and deleted if non-collection fields are used. • In Cassandra 3.6 and later, a query can be limited to return results from each partition, such as a "Top 3" listing using PER PARTITION LIMIT.
Removed CQL features	• Removal of CQL2 • Removal of cassandra-cli
Native protocol	• The Native Protocol has been updated to version 4, with implications for CQL use in the DataStax drivers.

CQL data modeling

Note: DataStax Academy provides a course in Cassandra data modeling. This course presents techniques using the Chebotko method for translating a real-world domain model into a running Cassandra schema.

Data modeling concepts

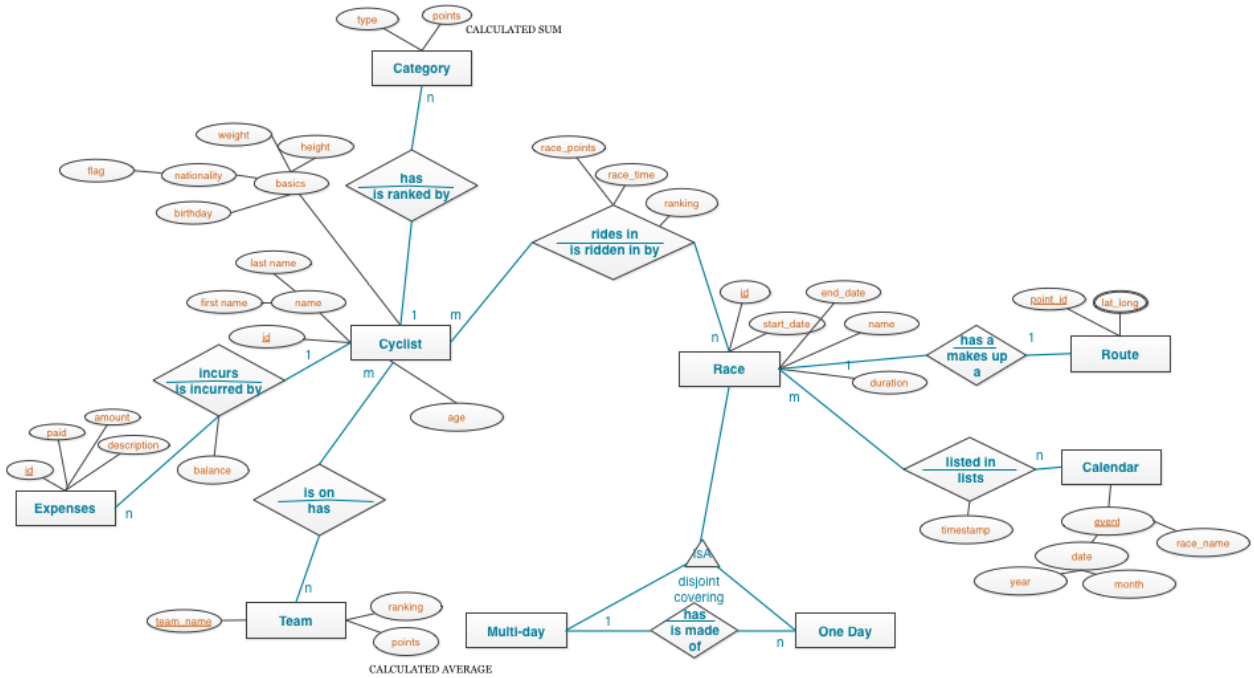
Data modeling is a process that involves identifying the entities (items to be stored) and the relationships between entities. To create your data model, identify the patterns used to access data and the types of queries to be performed. These two ideas inform the organization and structure of the data, and the design and creation of the database's tables. Indexing the data can lead to either performance or degradation of queries, so understanding indexing is an important step in the data modeling process.

Data modeling in Cassandra uses a query-driven approach, in which specific queries are the key to organizing the data. Queries are the result of selecting data from a table; schema is the definition of how data in the table is arranged. Cassandra's database design is based on the requirement for fast reads and writes, so the better the schema design, the faster data is written and retrieved.

In contrast, relational databases normalize data based on the tables and relationships designed, and then writes the queries that will be made. Data modeling in relational databases is table-driven, and any relationships between tables are expressed as table joins in queries.

Cassandra's data model is a partitioned row store with tunable consistency. Tunable consistency means for any given read or write operation, the client application decides how consistent the requested data must be. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns can be indexed separately from the primary key. Because Cassandra is a distributed database, efficiency is gained for reads and writes when data is grouped together on nodes by partition. The fewer partitions that must be queried to get an answer to a question, the faster the response. Tuning the consistency level is another factor in latency, but is not part of the data modeling process.

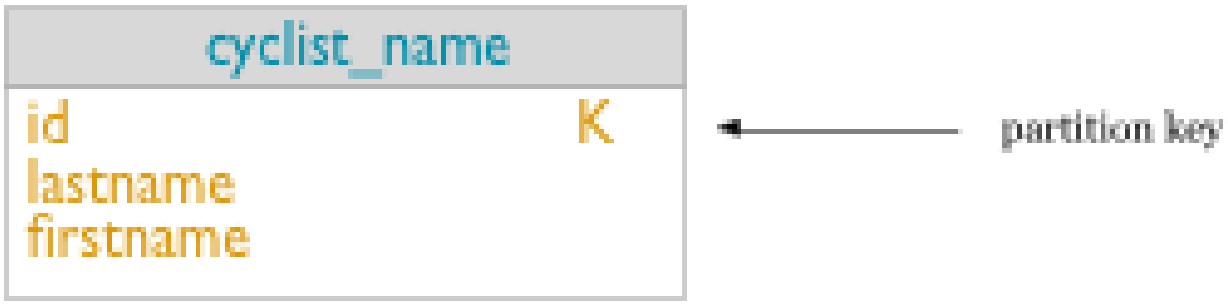
Cassandra data modeling focuses on the queries. Throughout this topic, the example of Pro Cycling statistics demonstrates how to model the Cassandra table schema for specific queries. The conceptual model for this data model shows the entities and relationships.



The entities and their relationships are considered during table design. Queries are best designed to access a single table, so all entities involved in a relationship that a query encompasses must be in the table. Some tables will involve a single entity and its attributes, like the first example shown below. Others will involve more than one entity and its attributes, such as the second example. Including all data in a single Cassandra table contrasts with a relational database approach, where the data would be stored in two or more tables and foreign keys would be used to relate the data between the tables. Because Cassandra uses this single table-single query approach, queries can perform faster.

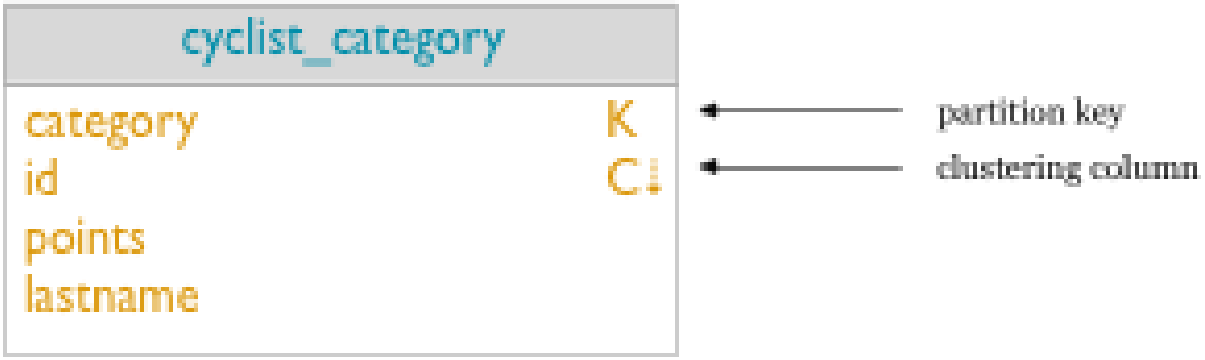
One basic query (Q1) for Pro Cycling statistics is a list of cyclists, including each cyclist's id, firstname, and lastname. To uniquely identify a cyclist in the table, an id using UUID is used. For a simple query to list all cyclists a table that includes all the columns identified and a partition key (K) of id is created. The diagram below shows a portion of the logical model for the Pro Cycling data model.

Figure: Query 1: Find a cyclist's name with a specified id



A related query (Q2) searches for all cyclists by a particular race category. For Cassandra, this query is more efficient if a table is created that groups all cyclists by category. Some of the same columns are required (id, lastname), but now the primary key of the table includes category as the partition key (K), and groups within the partition by the id (C). This choice ensures that unique records for each cyclist are created.

Figure: Query 2: Find cyclists given a specified category



These are two simple queries; more examples will be shown to illustrate data modeling using CQL.

Notice that the main principle in designing the table is not the relationship of the table to other tables, as it is in relational database modeling. Data in Cassandra is often arranged as one query per table, and data is repeated amongst many tables, a process known as denormalization. Relational databases instead normalize data, removing as much duplication as possible. The relationship of the entities is important, because the order in which data is stored in Cassandra can greatly affect the ease and speed of data retrieval. The schema design captures much of the relationship between entities by including related attributes in the same table. Client-side joins in application code is used only when table schema cannot capture the complexity of the relationships.

Data modeling analysis

You've created a conceptual model of the entities and their relationships. From the conceptual model, you've used the expected queries to create table schema. The last step in data model involves completing an analysis of the logical design to discover modifications that might be needed. These modifications can arise from understanding partition size limitations, cost of data consistency, and performance costs due to a number of design choices still to be made.

For efficient operation, partitions must be sized within certain limits. Two measures of partition size are the number of values in a partition and the partition size on disk. The maximum number of rows per partition is not theoretically limited, although practical limits can be found with experimentation. Sizing the disk space is more complex, and involves the number of rows and the number of columns, primary key columns and static columns in each table. Each application will have different efficiency parameters, but a good rule of thumb is to keep the maximum number of values below 100,000 items and the disk size under 100MB.

Data redundancy must be considered as well. Two redundancies that are a consequence of Cassandra's distributed design are duplicate data in tables and multiple partition replicates.

Data is generally duplicated in multiple tables, resulting in performance latency during writes and requires more disk space. Consider storing a cyclist's name and id in more than one data, along with other items like race categories, finished races, and cyclist statistics. Storing the name and id in multiple tables results in linear duplication, with two values stored in each table. Table design must take into account the possibility of higher order duplication, such as unlimited keywords stored in a large number of rows. A case of *n* keywords stored in *m* rows is not a good table design. You should rethink the table schema for better design, still keeping the query foremost.

Cassandra replicates partition data based on the replication factor, using more disk space. Replication is a necessary aspect of distributed databases and sizing disk storage correctly is important.

Application-side joins can be a performance killer. In general, you should analyze your queries that require joins and consider pre-computing and storing the join results in an additional table. In Cassandra, the goal is to use one table per query for performant behavior. Lightweight transactions (LWT) can also affect

performance. Consider whether or not the queries using LWT are necessary and remove the requirement if it is not strictly needed.

Using CQL

CQL provides an API to Cassandra that is simpler than the Thrift API. The Thrift API and legacy versions of CQL expose the internal storage structure of Cassandra. CQL adds an abstraction layer that hides implementation details of this structure and provides native syntaxes for collections and other common encodings.

Accessing CQL

Common ways to access CQL are:

- Start `cqlsh`, the Python-based command-line client, on the command line of a Cassandra node.
- Use `DataStax DevCenter`, a graphical user interface.
- For developing applications, you can use one of the official DataStax C#, Java, or Python open-source drivers.
- Use the set_cql_version Thrift method for programmatic access.

This document presents examples using `cqlsh`.

Starting the CQL shell (cqlsh)

Starting cqlsh on Linux and Mac OS X

This procedure briefly describes how to start `cqlsh` on Linux and Mac OS X. The `cqlsh` command is covered in detail later.

Procedure

1. Navigate to the Cassandra installation directory.
2. Start `cqlsh` on the Mac OSX, for example.

   ```
   $ bin/cqlsh
   ```

   If you use security features, provide a user name and password.
3. Print the help menu for `cqlsh`.

   ```
   $ bin/cqlsh --help
   ```
4. Optionally, specify the IP address and port to start `cqlsh` on a different node.

   ```
   $ bin/cqlsh 1.2.3.4 9042
   ```

   Note: You can use tab completion to see hints about how to complete a `cqlsh` command. Some platforms, such as Mac OSX, do not ship with tab completion installed. You can use easy_install to install tab completion capabilities on Mac OSX:

   ```
   $ easy_install readline
   ```

Starting cqlsh on Windows

This procedure describes how to start `cqlsh` on Windows. The P command is covered in detail later.

Procedure

You can start cqlsh in two ways:

- From the **Start** menu:
  a) Navigate to **Start** > **Programs** > **DataStax Distribution of Apache Cassandra**.
  b) If using Cassandra 3.0+, click **DataStax Distribution of Apache Cassandra** > **Cassandra CQL Shell**
  c) If using Cassandra 2.2, click **DataStax Community Edition** > **Cassandra CQL Shell**.

  The `cqlsh` prompt appears: `cqlsh>`
- From the Command Prompt:
  a) Open the Command Prompt.
  b) Navigate to `Cassandra bin directory`:

Cassandra 3.0+:

```
C:\> cd C:"Program Files\DataStax-DDC\apache-cassandra\bin"
```

Cassandra 2.2:

```
C:\> cd C:"Program Files\DataStax Community\apache-cassandra\bin"
```

c) Enter `cqlsh`.

The `cqlsh` prompt appears: `cqlsh>`

To get the help menu for `cqlsh`:

```
C:\> cqlsh --help
```

To start `cqlsh` on a different node, add the IP address and port:

```
C:\> cqlsh 1.2.3.4 9042
```

Note: You can use tab completion to see hints about how to complete a `cqlsh` command.

To install tab completion capabilities on Windows, you can use `pip install pyreadline`.

Creating and updating a keyspace

Creating a keyspace is the CQL counterpart to creating an SQL database, but a little different. The Cassandra keyspace is a namespace that defines how data is replicated on nodes. Typically, a cluster has one keyspace per application. Replication is controlled on a per-keyspace basis, so data that has different replication requirements typically resides in different keyspaces. Keyspaces are not designed to be used as a significant map layer within the data model. Keyspaces are designed to control data replication for a set of tables.

When you create a keyspace, specify a strategy class for replicating keyspaces. Using the `SimpleStrategy` class is fine for evaluating Cassandra. For production use or for use with mixed workloads, use the `NetworkTopologyStrategy` class.

To use `NetworkTopologyStrategy` for evaluation purposes using, for example, a single node cluster, the default data center name is used. To use `NetworkTopologyStrategy` for production use, you need to change the default snitch, `SimpleSnitch`, to a network-aware snitch, define one or more data center names in the snitch properties file, and use the data center name(s) to define the keyspace; see

Snitch. For example, if the cluster uses the PropertyFileSnitch, create the keyspace using the user-defined data center and rack names in the `cassandra-topologies.properties` file. If the cluster uses the Ec2Snitch, create the keyspace using EC2 data center and rack names. If the cluster uses the GoogleCloudSnitch, create the keyspace using GoogleCloud data center and rack names.

If you fail to change the default snitch and use `NetworkTopologyStrategy`, Cassandra will fail to complete any write request, such as inserting data into a table, and log this error message:

```
Unable to complete request: one or more nodes were unavailable.
```

Note: You cannot insert data into a table in keyspace that uses `NetworkTopologyStrategy` unless you define the data center names in the snitch properties file or you use a single data center named `datacenter1`.

Example of creating a keyspace

To query Cassandra, create and use a keyspace. Choose an arbitrary data center name and register the name in the properties file of the snitch. Alternatively, in a cluster in a single data center, use the default data center name, for example, datacenter1, and skip registering the name in the properties file.

Procedure

1. Determine the default data center name, if using NetworkTopologyStrategy, using `nodetool status`.

   ```
   $ bin/nodetool status
   ```

   The output is:

   ```
   Datacenter: datacenter1
   =======================
   Status=Up/Down
   |/ State=Normal/Leaving/Joining/Moving
   --  Address    Load       Tokens  Owns (effective)  Host ID      Rack
   UN  127.0.0.1  41.62 KB   256     100.0%            75dcca8f...  rack1
   ```

2. Create a keyspace.

   ```
   cqlsh> CREATE KEYSPACE IF NOT EXISTS cycling WITH REPLICATION =
    { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 };
   ```

3. Use the keyspace.

   ```
   cqlsh> USE cycling;
   ```

Updating the replication factor

Increasing the replication factor increases the total number of copies of keyspace data stored in a Cassandra cluster. for more information about replication, see Data replication.

When you change the replication factor of a keyspace, you affect each node that the keyspaces replicates to (or no longer replicates to). Follow this procedure to prepare all affected nodes for this change.

Procedure

1. Update a keyspace in the cluster and change its replication strategy options.

```
cqlsh> ALTER KEYSPACE system_auth WITH REPLICATION =
  {'class' : 'NetworkTopologyStrategy', 'dc1' : 3, 'dc2' : 2};
```

   Or if using SimpleStrategy:

```
cqlsh> ALTER KEYSPACE "Excalibur" WITH REPLICATION =
  { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };
```

2. On each affected node, run the `nodetool repair` command.
3. Wait until repair completes on a node, then move to the next node.

   For more about replication strategy options, see Changing keyspace replication strategy

What to do next

Changing the replication factor of the system_auth keyspace

If you are using security features, it is particularly important to increase the replication factor of the system_auth keyspace from the default (1) because you will not be able to log into the cluster if the node with the lone replica goes down. It is recommended to set the replication factor for the system_auth keyspace equal to the number of nodes in each data center.

Restricting replication for a keyspace

The example above shows how to configure a keyspace to create different numbers of replicas on different data centers. In some cases, you may want to prevent the keyspace from sending replicas to particular data centers — or restrict a keyspace to just one data center.

To do this, use `ALTER KEYSPACE` to configure the keyspace to use NetworkTopologyStrategy, as shown above. You can prevent the keyspace from sending replicas to a specific datacenter by setting its replication factor to `0` (zero). For example:

```
cqlsh> ALTER KEYSPACE keyspace1 WITH REPLICATION =
  {'class' : 'NetworkTopologyStrategy', 'dc1' : 0, 'dc2' : 3, 'dc3' : 0 };
```

This command configures `keyspace1` to create replicas only on `dc2`. The data centers `dc1` and `dc3` receive no replicas from tables in `keyspace1`.

Creating a table

In CQL, data is stored in tables containing rows of columns.

Creating a table

In CQL, data is stored in tables containing rows of columns, similar to SQL definitions.

Note: The concept of rows and columns in the internal implementation of Cassandra are **not** the same. For more information, see A thrift to CQL3 upgrade guide or CQL3 for Cassandra experts.

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. Add the optional `WITH` clause and keyword arguments to configure table properties (caching, compaction, etc.). See the Setting a table property page for details.

Create schema using cqlsh

Create table schema using `cqlsh`. Cassandra does not support dynamic schema generation — collision can occur if multiple clients attempt to generate tables simultaneously. To recover from collisions, follow the instructions in schema collision fix.

Primary Key

A primary key identifies the location and order of stored data. The primary key is defined when the table is created and cannot be altered. If you must change the primary key, create a new table schema and write the existing data to the new table. See ALTER TABLE for details on altering a table after creation.

Cassandra is a partition row store. The first element of the primary key, the partition key, specifies which node will hold a particular table row. At the minimum, the primary key must consist of a partition key. You can define a compound partition key to split a data set so that related data is stored on separate partitions. A compound primary key includes clustering columns which order the data on a partition.

Note: In Cassandra3.0 and earlier, you cannot insert any value larger than 64K bytes into a clustering column.

The definition of a table's primary key is critical in Cassandra. Carefully model how data in a table will be inserted and retrieved before choosing which columns to define in the primary key. The size of the partitions, the order of the data within partitions, the distribution of the partitions among the nodes of the cluster — you must consider all of these when selecting the table's primary key.

Table characteristics

The name of a table can be a string of alphanumeric characters and underscores, but it must begin with a letter. Tips for the table name:

- To specify the keyspace that contains the table, put the keyspace name followed by a period before the table name: *keyspace_name*.*table_name*. This allows you to create a new table in a keyspace that is different from the one set for the current session (by the USE command, for example).
- To create a table in the current keyspace, just use the new table name.

Column characteristics

CQL supports several column types. You assign a data type to each column when you create a table. The table definition defines (non-collection) columns in a comma-delimited list of name and type pairs. The following example illustrates three data types, UUID, text and timestamp:

```
CREATE TABLE cycling.cyclist_alt_stats ( id UUID PRIMARY KEY, lastname text,
 birthday timestamp, nationality text, weight text, height text );
```

CQL supports the following collection column types: map, set, and list. A collection column is defined using the collection type, followed by another type, such as int or text, in angle brackets. The collrection column definition is included in the column list as described above. The following example illustrates each collection type, but is not designed for an actual query:

```
CREATE TABLE cycling.whimsey ( id UUID PRIMARY KEY, lastname text,
 cyclist_teams set<text>, events list<text>, teams map<int,text> );
```

Collection types cannot be nested. Collections can include frozen data types. For examples and usage, see Collection type on page 92

A column of type tuple holds a fixed-length set of typed positional fields. Use a tuple as an alternative to a user-defined type. A tuple can accommodate many fields (32768) — although it would not be a good idea to usse this many. A typical tuple holds 2 to 5 fields. Specify a tuple in a table definition, using angle

brackets; within thesse, use a comma-delimited list to define each component type. Tuples can be nested. The following example illustrates a tuple type composed of a text field and a nested tuple of two float fields:

```
CREATE TABLE cycling.route (race_id int, race_name text, point_id int,
 lat_long tuple<text, tuple<float,float>>, PRIMARY KEY (race_id, point_id));
```

Note: Cassandra no longer requires the use of frozen for tuples:

```
frozen <tuple <int, tuple<text, double>>>
```

For more information, see "Tuple type".

Create a User-defined type (UDTs) as a data type of several fields, using CREATE TYPE. It is best to create a UDT for use with multiple table definitions. The user-defined column type (UDT) requires the frozen keyword. 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. The scope of a user-defined type is the keyspace in which you define it. Use dot notation to access a type from a keyspace outside its scope: keyspace name followed by a period followed the name of the type. An example is `test.myType` where `test` is the keyspace name and `myType` is the type name. Cassandra accesses the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, Cassandra accesses the type within the current keyspace. For examples and usage information, see "Using a user-defined type".

A counter is a special column used to store a number that is changed in increments. A counter can only be used in a dedicated table that includes a column of counter data type. For more examples and usage information, see "Using a counter".

Using the keyspace qualifier

Sometimes issuing a USE statement to select a keyspace is inconvenient. Connection pooling requires managing multiple keyspaces. To simplify tracking multiple keyspaces, use the keyspace qualifier instead of the USE statement. You can specify the keyspace using the keyspace qualifier in these statements:

- ALTER TABLE
- CREATE TABLE
- DELETE
- INSERT
- SELECT
- TRUNCATE
- UPDATE

Procedure

To specify a table when you are not in the keyspace that contains the table, use the name of the keyspace followed by a period, then the table name. For example, cycling.race_winners.

```
cqlsh> INSERT INTO cycling.race_winners ( race_name, race_position,
 cyclist_name ) VALUES (
  'National Championships South Africa WJ-ITT (CN)',
  1,
  {firstname:'Frances',lastname:'DU TOUT'}
);
```

Simple Primary Key

For a table with a simple primary key, Cassandra uses one column name as the partition key. The primary key consists of only the partition key in this case. Data stored with a simple primary key will be fast to insert and retrieve if many values for the column can distribute the partitions across many nodes.

Often, your first venture into using Cassandra involves tables with simple primary keys. Keep in mind that only the primary key can be specified when retrieving data from the table (unless you use secondary indexes). If an application needs a simple lookup table using a single unique identifier, then a simple primary key is the right choice. The table shown uses id as the primary key.

```
id                                   | firstname | lastname
--------------------------------------+-----------+-----------------
e7ae5cf3-d358-4d99-b900-85902fda9bb0 |      Alex |           FRAME
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |             VOS
220844bf-4860-49d6-9a4b-6b5d3a79cbfb |     Paolo |       TIRALONGO
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |    Steven |      KRUIKSWIJK
e7cd5752-bc0d-4157-a80f-7523add8dbcd |      Anna | VAN DER BREGGEN
```

If you have simple retrieval needs, use a simple primary key.

Using a simple primary key

Use a simple primary key to create a single column that you can use to query and return results. This example creates a cyclist_name table storing an ID number and a cyclist's first and last names in columns. The table uses a UUID as a primary key. This table can be queried to discover the name of a cyclist given their ID number.

A simple primary key table can be created in three different ways, as shown.

Procedure

- Create the table cyclist_name in the cycling keyspace, making id the primary key. Insert the `PRIMARY KEY` keywords after the column name in the `CREATE TABLE` definition. Before creating the table, set the keyspace with a `USE` statement.

```
cqlsh> USE cycling;
CREATE TABLE cyclist_name ( id UUID PRIMARY KEY, lastname text, firstname
 text );
```

- This same example can be written with the primary key identified at the end of the table definition. Insert the `PRIMARY KEY` keywords after the last column definition in the `CREATE TABLE` definition, followed by the column name of the key. The column name is enclosed in parentheses.

```
cqlsh> USE cycling;
CREATE TABLE cyclist_name ( id UUID, lastname text, firstname text,
 PRIMARY KEY (id) );
```

- The keyspace name can be used to identify the keyspace in the `CREATE TABLE` statement instead of the `USE` statement.

```
cqlsh> CREATE TABLE cycling.cyclist_name ( id UUID, lastname text,
 firstname text, PRIMARY KEY (id) );
```

Composite Partition Key

For a table with a composite partition key, Cassandra uses multiple columns as the partition key. These columns form logical sets inside a partition to facilitate retrieval. In contrast to a simple partition key, a composite partition key uses two or more columns to identify where data will reside. Composite partition keys are used when the data stored is too large to reside in a single partition. Using more than one column for the partition key breaks the data into chunks, or buckets. The data is still grouped, but in smaller chunks. This method can be effective if a Cassandra cluster experiences hotspotting, or congestion in writing data to one node repeatedly, because a partition is heavily writing. Cassandra is often used for time series data, and hotspotting can be a real issue. Breaking incoming data into buckets by year:month:day:hour, using four columns to route to a partition can decrease hotspots.

Data is retrieved using the partition key. Keep in mind that to retrieve data from the table, values for all columns defined in the partition key have to be supplied, if secondary indexes are not used. The table shown uses race_year and race_name in the primary key, as a composition partition key. To retrieve data, both parameters must be identified.

```
 race_year | race_name                                    | rank | cyclist_name
-----------+----------------------------------------------+------+----------------------
      2014 |                        4th Tour of Beijing |    1 |    Phillippe GILBERT
      2014 |                        4th Tour of Beijing |    2 |         Daniel MARTIN
      2014 |                        4th Tour of Beijing |    3 | Johan Esteban CHAVES
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |         Ilnur ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |      Carlos BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |      Benjamin PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |          Adam PHELAN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |         Thomas LEBAS
```

Cassandra stores an entire row of data on a node by partition key. If you have too much data in a partition and want to spread the data over multiple nodes, use a composite partition key.

Using a composite partition key

Use a composite partition key in your primary key to create a set of columns that you can use to distribute data across multiple partitions and to query and return sorted results. This example creates a rank_by_year_and_name table storing the ranking and name of cyclists who competed in races. The table uses race_year and race_name as the columns defining the composition partition key of the primary key. The query discovers the ranking of cyclists who competed in races by supplying year and race name values.

A composite partition key table can be created in two different ways, as shown.

Procedure

- Create the table rank_by_year_and_name in the cycling keyspace. Use race_year and race_name for the composite partition key. The table definition shown has an additional column rank used in the primary key. Before creating the table, set the keyspace with a USE statement. This example identifies the primary key at the end of the table definition. Note the double parentheses around the first two columns defined in the `PRIMARY KEY`.

```
cqlsh> USE cycling;
CREATE TABLE rank_by_year_and_name (
race_year int,
race_name text,
cyclist_name text,
rank int,
PRIMARY KEY ((race_year, race_name), rank)
```

```
);
```

- The keyspace name can be used to identify the keyspace in the `CREATE TABLE` statement instead of the `USE` statement.

```
cqlsh> CREATE TABLE cycling.rank_by_year_and_name (
race_year int,
race_name text,
cyclist_name text,
rank int,
PRIMARY KEY ((race_year, race_name), rank)
);
```

Compound Primary Key

For a table with a compound primary key, Cassandra uses a partition key that is either simple or composite. In addition, clustering column(s) are defined. Clustering is a storage engine process that sorts data within each partition based on the definition of the clustering columns. Normally, columns are sorted in ascending alphabetical order. Generally, a different grouping of data will benefit reads and writes better than this simplistic choice.

Remember that data is distributed throughout the Cassandra cluster. An application can experience high latency while retrieving data from a large partition if the entire partition must be read to gather a small amount of data. On a physical node, when rows for a partition key are stored in order based on the clustering columns, retrieval of rows is very efficient. Grouping data in tables using clustering columns is the equivalent of `JOIN`s in a relational database, but are much more performant because only one table is accessed. This table uses category as the partition key and points as the clustering column. Notice that for each category, the points are ordered in descending order.

category	points	id	lastname
One-day-races	367	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
One-day-races	198	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
Time-trial	182	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
Time-trial	3	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
Sprint	39	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
Sprint	0	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO
GC	1324	6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	KRUIJSWIJK
GC	1269	220844bf-4860-49d6-9a4b-6b5d3a79cbfb	TIRALONGO

Cassandra stores an entire row of data on a node by partition key and can order the data for retrieval with clustering columns.Retrieving data from a partition is more versatile with clustering columns. For the example shown, a query could retrieve all point values greater than 200 for the One-day-races. If you have more complex needs for querying, use a compound primary key.

Using a compound primary key

Use a compound primary key to create multiple columns that you can use to query and return sorted results. If our pro cycling example was designed in a relational database, you would create a cyclists table with a foreign key to the races. In Cassandra, you denormalize the data because joins are not performant in a distributed system. Later, other schema are shown that improve Cassandra performance. Collections and indexes are two data modeling methods. This example creates a cyclist_category table storing a cyclist's last name, ID, and points for each type of race category. The table uses category for the partition key and points for a single clustering column. This table can be queried to retrieve a list of cyclists and their points in a category, sorted by points.

A compound primary key table can be created in two different ways, as shown.

Procedure

- To create a table having a compound primary key, use two or more columns as the primary key. This example uses an additional clause `WITH CLUSTERING ORDER BY` to order the points in descending order. Ascending order is more efficient to store, but descending queries are faster due to the nature of the storage engine.

```
cqlsh> USE cycling;
CREATE TABLE cyclist_category (
category text,
points int,
id UUID,
lastname text,
PRIMARY KEY (category, points)
) WITH CLUSTERING ORDER BY (points DESC);
```

  Note: The combination of the category and points uniquely identifies a row in the cyclist_category table. More than one row with the same category can exist as long as the rows contain different pointsvalues.
- The keyspace name can be used to identify the keyspace in the `CREATE TABLE` statement instead of the `USE` statement.

```
cqlsh> CREATE TABLE cycling.cyclist_category (
category text,
points int,
id UUID,
lastname text,
PRIMARY KEY (category, points)
) WITH CLUSTERING ORDER BY (points DESC);
```

  Note: In both of these examples, points is defined as a clustering column. In Cassandra3.0 and earlier, you cannot insert any value larger than 64K bytes into a clustering column.

Creating a counter table

A counter is a special column used to store an integer that is changed in increments.

Counters are useful for many data models. Some examples:

- To keep track of the number of web page views received on a company website
- To keep track of the number of games played online or the number of players who have joined an online game

The table shown below uses id as the primary key and keeps track of the popularity of a cyclist based on thumbs up/thumbs down clicks in the popularity field of a counter table.

Tracking count in a distributed database presents an interesting challenge. In Cassandra, at any given moment, the counter value may be stored in the Memtable, commit log, and/or one or more SSTables. Replication between nodes can cause consistency issues in certain edge cases. Cassandra counters were

redesigned in Cassandra 2.1 to alleviate some of the difficulties. Read "What's New in Cassandra 2.1: Better Implementation of Counters" to discover the improvements made in the counters.

Because counters are implemented differently from other columns, counter columns can only be created in dedicated tables. A counter column must have the datatype counter data type. This data type cannot be assigned to a column that serves as the primary key or partition key. To implement a counter column, create a table that only includes:

- The primary key (can be one or more columns)
- The counter column

Many counter-related settings can be set in the `cassandra.yaml` file.

A counter column cannot be indexed or deleted.. To load data into a counter column, or to increase or decrease the value of the counter, use the `UPDATE` command. Cassandra rejects `USING TIMESTAMP` or `USING TTL` when updating a counter column.

To create a table having one or more counter columns, use this:

- Use CREATE TABLE to define the counter and non-counter columns. Use all non-counter columns as part of the PRIMARY KEY definition.

Using a counter

To load data into a counter column, or to increase or decrease the value of the counter, use the UPDATE command. Cassandra rejects USING TIMESTAMP or USING TTL in the command to update a counter column.

Procedure

- Create a table for the counter column.

```
cqlsh> USE cycling;
CREATE TABLE popular_count (
  id UUID PRIMARY KEY,
  popularity counter
  );
```

- Loading data into a counter column is different than other tables. The data is updated rather than inserted.

```
UPDATE cycling.popular_count
 SET popularity = popularity + 1
 WHERE id = 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

- Take a look at the counter value and note that popularity has a value of 1.

```
SELECT * FROM cycling.popular_count;
```

- Additional increments or decrements will change the value of the counter column.

Create table with COMPACT STORAGE

When you create a table using compound primary keys, for every piece of data stored, the column name needs to be stored along with it. Instead of each non-primary key column being stored such that each column corresponds to one column on disk, an entire row is stored in a single column on disk. If you need to conserve disk space, use the WITH COMPACT STORAGE directive that stores data in the legacy (Thrift) storage engine format.

```
CREATE TABLE sblocks (
```

```
  block_id uuid,
  subblock_id uuid,
  data blob,
  PRIMARY KEY (block_id, subblock_id)
)
WITH COMPACT STORAGE;
```

Using the compact storage directive prevents you from defining more than one column that is not part of a compound primary key. A compact table using a primary key that is not compound can have multiple columns that are not part of the primary key.

A compact table that uses a compound primary key must define at least one clustering column. Columns cannot be added nor removed after creation of a compact table. Unless you specify WITH COMPACT STORAGE, CQL creates a table with non-compact storage.

Collections and static columns cannot be used with COMPACT STORAGE tables.

Table schema collision fix

Dynamic schema creation or updates can cause schema collision resulting in errors.

Procedure

1. Run a rolling restart on all nodes to ensure schema matches. Run `nodetool describecluster` on all nodes. Check that there is only one schema version.
2. On each node, check the `data` directory, looking for two directories for the table in question. If there is only one directory, go on to the next node. If there are two or more directories, the old table directory before update and a new table directory for after the update, continue.
3. Identify which `cf_id` (column family ID) is the newest table ID in `system.schema_columnfamilies`. The column family ID is fifth column in the results.

   ```
   $ cqlsh -e "SELECT * FROM system.schema_column_families" |grep <tablename>
   ```

4. Move the data from the older table to the newer table's directory and remove the old directory. Repeat this step as necessary.
5. Run `nodetool refresh`.

Creating a materialized view

In Cassandra 3.0 and later, a materialized view is a table that is built from another table's data with a new primary key and new properties. In Cassandra, queries are optimized by primary key definition. Standard practice is to create the table for the query, and create a new table if a different query is needed. Until Cassandra 3.0, these additional tables had to be updated manually in the client application. A materialized view automatically receives the updates from its source table.

Secondary indexes are suited for low cardinality data. Queries of high cardinality columns on secondary indexes require Cassandra to access all nodes in a cluster, causing high read latency.

Materialized views are suited for high cardinality data. The data in a materialized view is arranged serially based on the view's primary key. Materialized views cause hotspots when low cardinality data is inserted.

Requirements for a materialized view:

- The columns of the source table's primary key must be part of the materialized view's primary key.
- Only one new column can be added to the materialized view's primary key. Static columns are not allowed.

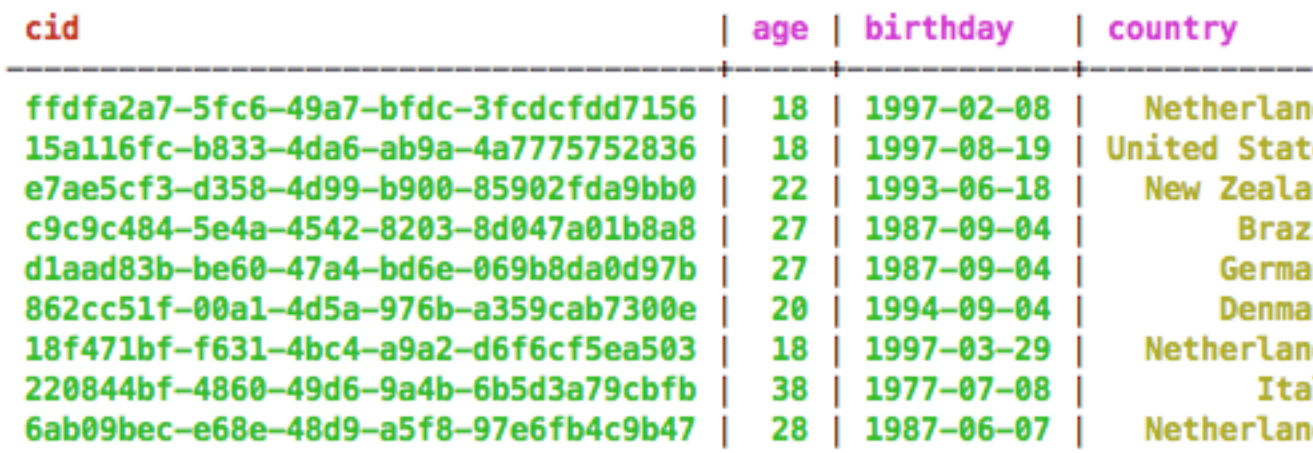
You can create a materialized view with its own WHERE conditions and its own properties.

Materialized view example

The following table is the original, or source, table for the materialized view examples in this section.

```
CREATE TABLE cyclist_mv (cid UUID PRIMARY KEY, name text, age int, birthday
 date, country text);
```

This table holds values for the name, age, birthday, and country affiliation of several



cid	age	birthday	country
ffdfa2a7-5fc6-49a7-bfdc-3fcdcfdd7156	18	1997-02-08	Netherlan
15a116fc-b833-4da6-ab9a-4a7775752836	18	1997-08-19	United Stat
e7ae5cf3-d358-4d99-b900-85902fda9bb0	22	1993-06-18	New Zeala
c9c9c484-5e4a-4542-8203-8d047a01b8a8	27	1987-09-04	Braz
d1aad83b-be60-47a4-bd6e-069b8da0d97b	27	1987-09-04	Germa
862cc51f-00a1-4d5a-976b-a359cab7300e	20	1994-09-04	Denma
18f471bf-f631-4bc4-a9a2-d6f6cf5ea503	18	1997-03-29	Netherlan
220844bf-4860-49d6-9a4b-6b5d3a79cbfb	38	1977-07-08	Ita
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47	28	1987-06-07	Netherlan

cyclists.

The `cyclist_mv` table can be the basis of a materialized view that uses age in the primary key.

```
CREATE MATERIALIZED VIEW cyclist_by_age
AS SELECT age, birthday, name, country
FROM cyclist_mv
WHERE age IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (age, cid);
```

This `CREATE MATERIALIZED VIEW` statement has several features:

- The `AS SELECT` phrase identifies the columns copied from the base table to the materialized view.
- The `FROM` phrase identifies the source table from which Cassandra will copy the data.
- The `WHERE` clause must include all primary key columns with the `IS NOT NULL` phrase so that only rows with data for all the primary key columns are copied to the materialized view.
- As with any table, the specification of the primary key columns. Since `cyclist_mv`, the source table, uses cid as its primary key, cid must be present in the materialized view's primary key.

  Note: In this materialized view, age is used as the primary key and cid is a clustering column. In Cassandra3.0 and earlier, you cannot use a column as a clustering column if any of its values is larger than 64K bytes.

Because the new materialized view is partitioned by age, it supports queries based on the cyclists' ages.

```
SELECT age, name, birthday FROM cyclist_by_age WHERE age = 18;
```

age	name	birthday
18	Adrien COSTA	1997-08-19
18	Bram WELTEN	1997-03-29
18	Pascal EENKHOORN	1997-02-08

Other materialized views, based on the same source table, can organize information by cyclists' birthdays or countries of origin.

```
CREATE MATERIALIZED VIEW cyclist_by_birthday
AS SELECT age, birthday, name, country
FROM cyclist_mv
WHERE birthday IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (birthday, cid);

CREATE MATERIALIZED VIEW cyclist_by_country
AS SELECT age,birthday, name, country
FROM cyclist_mv
WHERE country IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (country, cid);
```

The following queries use the new materialized views.

```
SELECT age, name, birthday FROM cyclist_by_country WHERE country =
 'Netherlands';
```

```
 age | name                | birthday
-----+---------------------+------------
  18 |        Bram WELTEN | 1997-03-29
  28 | Steven KRUIKSWIJK | 1987-06-07
  18 |  Pascal EENKHOORN | 1997-02-08
```

```
SELECT age, name, birthday FROM cyclist_by_birthday WHERE birthday =
 '1987-09-04';
```

```
 age | name              | birthday
-----+-------------------+------------
  27 | Cristian EGIDIO | 1987-09-04
  27 | Johannes HEIDER | 1987-09-04
```

When another INSERT is executed on `cyclist_mv`, Cassandra updates the source table and both of these materialized views. When data is deleted from `cyclist_mv`, Cassandra deletes the same data from any related materialized views.

Materialized views allow fast lookup of the data using the normal Cassandra read path. However, materialized views do not have the same write performance as normal table writes. Cassandra performs an additional read-before-write to update each materialized view. To complete an update, Cassandra preforms a data consistency check on each replica. A write to the source table incurs latency. The performance of deletes on the source table also suffers. If a delete on the source table affects two or more contiguous rows, this delete is tagged with one tombstone. But these same rows may not be contiguous in materialized views derived from the source table. If they are not, Cassandra creates multiple tombstones in the materialized views.

Cassandra can only write data directly to source tables, not to materialized views. Cassandra updates a materialized view asynchronously after inserting data into the source table, so the update of materialized view is delayed. Cassandra performs a read repair to a materialized view only after updating the source table.

For additional information on how materialized views work, go to New in Cassandra 3.0: Materialized Views and Cassandra Summit 2015 talk on Materialized Views.

Related information

Altering a materialized view on page 67

CREATE MATERIALIZED VIEW on page 153

ALTER MATERIALIZED VIEW on page 131

DROP MATERIALIZED VIEW on page 171

Creating advanced data types in tables

Data can be inserted into tables using more advanced types of data.

Creating collections

Cassandra provides collection types as a way to group and store data together in a column. For example, in a relational database a grouping such as a user's multiple email addresses is related with a many-to-one joined relationship between a user table and an email table. Cassandra avoids joins between two tables by storing the user's email addresses in a collection column in the user table. Each collection specifies the data type of the data held.

A collection is appropriate if the data for collection storage is limited. If the data has unbounded growth potential, like messages sent or sensor events registered every second, do not use collections. Instead, use a table with a compound primary key where data is stored in the clustering columns.

CQL contains these collection types:

- set
- list
- map

Observe the following limitations of collections:

- Never insert more than 2 billion items in a collection, as only that number can be queried.
- The maximum number of keys for a `map` collection is 65,535.
- The maximum size of an item in a `list` or a `map` collection is 2GB.
- The maximum size of an item in a `set` collection is 65,535 bytes.
- Keep collections small to prevent delays during querying.

  Collections cannot be "sliced"; Cassandra reads a collection in its entirety, impacting performance. Thus, collections should be much smaller than the maximum limits listed. The collection is not paged internally.
- Lists can incur a read-before-write operation for some insertions. Sets are preferred over lists whenever possible.

Note: The limits specified for collections are for non-frozen collections.

You can expire each element of a collection by setting an individual time-to-live (TTL) property.

Also see Using frozen in a collection.

Creating the set type

A set consists of a group of elements with unique values. Duplicate values will note be stored distinctly. The values of a set are stored unordered, but will return the elements in sorted order when queried. Use the set data type to store data that has a many-to-one relationship with another column. For example, in the example below, a set called teams stores all the teams that a cyclist has been a member of during their career.

Procedure

- Define teams in a table cyclist_career_teams. Each team listed in the set will have a textdata type.

```
cqlsh> CREATE TABLE cycling.cyclist_career_teams ( id UUID PRIMARY KEY,
 lastname text, teams set<text> );
```

```
 lastname        | teams
-----------------+-----------------------------------------------------------------------------------------------------
      ARMITSTEAD |                                {'AA Drink - Leontien.nl', 'Boels-Dolmans Cycling Team', 'Team Garmin - Cervelo'}
             VOS | {'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
           BRAND |   {'AA Drink - Leontien.nl', 'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
VAN DER BREGGEN |                 {'Rabobank-Liv Woman Cycling Team', 'Sengers Ladies Cycling Team', 'Team Flexpoint'}
```

Creating the list type

A list has a form much like a set, in that a list groups and stores values. Unlike a set, the values stored in a list do not need to be unique and can be duplicated. In addition, a list stores the elements in a particular order and may be inserted or retrieved according to an index value.

Use the list data type to store data that has a possible many-to-many relationship with another column. For example, in the example below, a list called events stores all the race events on an upcoming calendar. Each month/year pairing might have several events occurring, and the races are stored in a list. The list can be ordered so that the races appear in the order that they will take place, rather than alphabetical order.

Procedure

- Define events in a table upcoming_calendar. Each event listed in the list will have a textdata type.

```
cqlsh> CREATE TABLE cycling.upcoming_calendar ( year int, month int,
 events list<text>, PRIMARY KEY ( year, month) );
```

```
 year | month | events
------+-------+-----------------------------------------------------
 2015 |     6 | ['Criterium du Dauphine', 'Tour de Suisse']
 2015 |     7 |                                ['Tour de France']
```

Creating the map type

A map relates one item to another with a key-value pair. For each key, only one value may exist, and duplicates cannot be stored. Both the key and the value are designated with a data type.

.Using the map type, you can store timestamp-related information in user profiles. Each element of the map is internally stored as one Cassandra column that you can modify, replace, delete, and query. Each element can have an individual time-to-live and expire when the TTL ends.

Procedure

Define teams in a table cyclist_teams. Each team listed in the map will have an integer data type for the year a cyclist belonged to the team and a textdata type for the team name. The map collection is specified with a map column name and the pair of data types enclosed in angle brackets.

```
cqlsh> CREATE TABLE cycling.cyclist_teams ( id UUID PRIMARY KEY, lastname
 text, firstname text, teams map<int,text> );
```

```
lastname        | firstname | teams
----------------+-----------+-----------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------
      ARMITSTEAD | Elizabeth |          {2011: 'Team Garmin - Cervelo', 2012: 'AA Drink - Leontien.nl', 2013: 'Boels-Dolmans Cyclin
g Team', 2014: 'Boels-Dolmans Cycling Team', 2015: 'Boels-Dolmans Cycling Team'}
            VOS |  Marianne |               {2011: 'Nederland bloeit', 2012: 'Rabobank Women Team', 2013: 'Rabobank-Liv Giant', 2
014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
 VAN DER BREGGEN |      Anna | {2009: 'Team Flexpoint', 2012: 'Sengers Ladies Cycling Team', 2013: 'Sengers Ladies Cycling Team', 2
014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
```

Creating a table with a tuple

Tuples are a data type that allow two or more values to be stored together in a column. A user-defined type can be used, but for simple groupings, a tuple is a good choice.

Procedure

- Create a table cycling.route using a tuple to store each waypoint location name, latitude, and longitude.

```
cqlsh> CREATE TABLE cycling.route (race_id int, race_name text, point_id
 int, lat_long tuple<text, tuple<float,float>>, PRIMARY KEY (race_id,
 point_id));
```

- Create a table cycling.nation_rankusing a tuple to store the rank, cyclist name, and points total for a cyclist and the country name as the primary key.

```
CREATE TABLE cycling.nation_rank ( nation text PRIMARY KEY, info
 tuple<int,text,int> );
```

- The table cycling.nation_rank is keyed to the country as the primary key. It is possible to store the same data keyed to the rank. Create a table cycling.popular using a tuple to store the country name, cyclist name and points total for a cyclist and the rank as the primary key.

```
CREATE TABLE cycling.popular (rank int PRIMARY KEY, cinfo
 tuple<text,text,int> );
```

Creating a user-defined type (UDT)

In Cassandra 2.1 and later, user-defined types (UDTs) can attach multiple data fields, each named and typed, to a single column. The fields used to create a UDT may be any valid data type, including collections and other existing UDTs. Once created, UDTs may be used to define a column in a table.

Procedure

- Use the cycling keyspace.

```
cqlsh> USE cycling;
```

- Create a user-defined type named basic_info.

```
cqlsh> CREATE TYPE cycling.basic_info (
  birthday timestamp,
  nationality text,
  weight text,
  height text
);
```

- Create a table for storing cyclist data in columns of type basic_info. Use the frozen keyword in the definition of the user-defined type column. In Cassandra 3.6 and later, the frozen keyword is not required for UDTs that contain only non-collection fields.

  When using the frozen keyword, you cannot update parts of a user-defined type value. The entire value must be overwritten. Cassandra treats the value of a frozen, user-defined type like a blob.

```
cqlsh> CREATE TABLE cycling.cyclist_stats ( id uuid PRIMARY KEY, lastname
 text, basics FROZEN<basic_info>);
```

- A user-defined type can be nested in another column type. This example nests a UDT in a list.

```
CREATE TYPE cycling.race (race_title text, race_date timestamp, race_time
 text);
CREATE TABLE cycling.cyclist_races ( id UUID PRIMARY KEY, lastname text,
 firstname text, races list<FROZEN <race>> );
```

Creating functions

In Cassandra 3.0, users can create user-defined functions (UDFs) and user-defined aggregate functions (UDAs). Functions are used to manipulate stored data in queries. Retrieving results using standard aggregate functions are also available for queries.

Creating user-defined function (UDF)

Cassandra 2.2 and later allows users to define functions that can be applied to data stored in a table as part of a query result. The function must be created prior to its use in a SELECT statement. The function will be performed on each row of the table. To use user-defined functions with Java or Javascript in Cassandra 2.2 or Javascript in Cassandra 3.0, `enable_user_defined_functions` must be set `true` incassandra.yaml file setting to enable the functions; it is not required for Java in Cassandra 3.0. User-defined functions are defined within a keyspace; if no keyspace is defined, the current keyspace is used. User-defined functions are executed in a sandbox in Cassandra 3.0 and later. In Cassandra 2.2, there is no security manager to prevent execution of malicious code; see the cassandra.yaml file for more details.

By default, Cassandra 2.2 and later supports defining functions in `java` and `javascript`. Other scripting languages, such as `Python`, `Ruby`, and `Scala` can be added by adding a JAR to the classpath. Install the JAR file into `$CASSANDRA_HOME/lib/jsr223/[language]/[jar-name].jar` where language is 'jruby', 'jython', or 'scala'

Procedure

- Create a function, specifying the data type of the returned value, the language, and the actual code of the function to be performed. The following function, `fLog()`, computes the logarithmic value of each

input. It is a built-in `java` function and used to generate linear plots of non-linear data. For this example, it presents a simple math function to show the capabilities of user-defined functions.

```
cqlsh> CREATE OR REPLACE FUNCTION fLog (input double) CALLED
 ON NULL INPUT RETURNS double LANGUAGE java AS 'return
 Double.valueOf(Math.log(input.doubleValue()));';
```

Note:

- `CALLED ON NULL INPUT` ensures the function will always be executed.
- `RETURNS NULL ON NULL INPUT` ensures the function will always return `NULL` if any of the input arguments is `NULL`.
- `RETURNS` defines the data type of the value returned by the function.

- A function can be replaced with a different function if `OR REPLACE` is used as shown in the example above. Optionally, the `IF NOT EXISTS` keywords can be used to create the function only if another function with the same signature does not exist in the keyspace. `OR REPLACE` and `IF NOT EXISTS` cannot be used in the same command.

```
cqlsh> CREATE FUNCTION IF NOT EXISTS fLog (input double)
 CALLED ON NULL INPUT RETURNS double LANGUAGE java AS 'return
 Double.valueOf(Math.log(input.doubleValue()));';
```

Creating User-Defined Aggregate Function (UDA)

Cassandra 2.2 and later allows users to define aggregate functions that can be applied to data stored in a table as part of a query result. The aggregate function must be created prior to its use in a SELECT statement and the query must only include the aggregate function itself, but no columns. The state function is called once for each row, and the value returned by the state function becomes the new state. After all rows are processed, the optional final function is executed with the last state value as its argument. Aggregation is performed by the coordinator.

The example shown computes the team average for race time for all the cyclists stored in the table. The race time is computed in seconds.

Procedure

- Create a state function, as a user-defined function (UDF), if needed. This function adds all the race times together and counts the number of entries.

```
cqlsh> CREATE OR REPLACE FUNCTION avgState ( state tuple<int,bigint>, val
 int ) CALLED ON NULL INPUT RETURNS tuple<int,bigint> LANGUAGE java AS
  'if (val !=null) { state.setInt(0, state.getInt(0)+1); state.setLong(1,
 state.getLong(1)+val.intValue()); } return state;';
```

- Create a final function, as a user-defined function (UDF), if needed. This function computes the average of the values passed to it from the state function.

```
cqlsh> CREATE OR REPLACE FUNCTION avgFinal ( state tuple<int,bigint> )
 CALLED ON NULL INPUT RETURNS double LANGUAGE java AS
  'double r = 0; if (state.getInt(0) == 0) return null; r =
 state.getLong(1); r/= state.getInt(0); return Double.valueOf(r);';
```

- Create the aggregate function using these two functions, and add an `STYPE` to define the data type for the function. Different `STYPE`s will distinguish one function from another with the same name. An aggregate can be replaced with a different aggregate if `OR REPLACE` is used as shown in the examples above. Optionally, the `IF NOT EXISTS` keywords can be used to create the aggregate only if another

aggregate with the same signature does not exist in the keyspace. `OR REPLACE` and `IF NOT EXISTS` cannot be used in the same command.

```
cqlsh> CREATE AGGREGATE IF NOT EXISTS average ( int )
SFUNC avgState STYPE tuple<int,bigint> FINALFUNC avgFinal INITCOND (0,0);
```

What to do next

For more information on user-defined aggregates, see Cassandra Aggregates - min, max, avg, group by and A few more Cassandra aggregates.

Inserting and updating data

Data can be inserted into tables using the INSERT command. With Cassandra 3.0, JSON data can be inserted.

Inserting simple data into a table

In a production database, inserting columns and column values programmatically is more practical than using `cqlsh`, but often, testing queries using this SQL-like shell is very convenient.

Insertion, update, and deletion operations on rows sharing the same partition key for a table are performed atomically and in isolation.

Procedure

- To insert simple data into the table cycling.cyclist_name, use the `INSERT` command. This example inserts a single record into the table.

```
cqlsh> INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
 (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS','Marianne');
```

- You can insert complex string constants using double dollar signs to enclose a string with quotes, backslashes, or other characters that would normally need to be escaped.

```
cqlsh> INSERT INTO cycling.calendar (race_id, race_start_date,
 race_end_date, race_name) VALUES
  (201, '2015-02-18', '2015-02-22', $$Women's Tour of New Zealand$$);
```

Inserting and updating data into a set

If a table specifies a set to hold data, then either `INSERT` or `UPDATE` is used to enter data.

Procedure

- Insert data into the set, enclosing values in curly brackets.

  Set values must be unique, because no order is defined in a set internally.

```
cqlsh>INSERT INTO cycling.cyclist_career_teams (id,lastname,teams)
  VALUES (5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS',
  { 'Rabobank-Liv Woman Cycling Team','Rabobank-Liv Giant','Rabobank Women
 Team','Nederland bloeit' } );
```

- Add an element to a set using the UPDATE command and the addition (+) operator.

```
cqlsh> UPDATE cycling.cyclist_career_teams
  SET teams = teams + {'Team DSB - Ballast Nedam'} WHERE id =
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Remove an element from a set using the subtraction (-) operator.

```
cqlsh> UPDATE cycling.cyclist_career_teams
  SET teams = teams - {'WOMBATS - Womens Mountain Bike & Tea Society'}
 WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Remove all elements from a set by using the UPDATE or DELETE statement.

  A set, list, or map needs to have at least one element because an empty set, list, or map is stored as a null set.

```
cqlsh> UPDATE cyclist.cyclist_career_teams SET teams = {} WHERE id =
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

DELETE teams FROM cycling.cyclist_career_teams WHERE id =
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

  A query for the teams returns null.

```
cqlsh> SELECT id, teams FROM users WHERE id = 5b6962dd-3f90-4c93-8f61-
eabfa4a803e2;
```

```
 lastname | teams
----------+-------
      vos |  null
```

Inserting and updating data into a list

If a table specifies a list to hold data, then either INSERT or UPDATE is used to enter data.

Procedure

- Insert data into the list, enclosing values in square brackets.

```
INSERT INTO cycling.upcoming_calendar (year, month, events) VALUES (2015,
 06, ['Criterium du Dauphine','Tour de Suisse']);
```

- Use the UPDATE command to insert values into the list. Prepend an element to the list by enclosing it in square brackets and using the addition (+) operator.

```
cqlsh> UPDATE cycling.upcoming_calendar SET events = ['The Parx Casino
 Philly Cycling Classic'] + events WHERE year = 2015 AND month = 06;
```

- Append an element to the list by switching the order of the new element data and the list name in the UPDATE command.

```
cqlsh> UPDATE cycling.upcoming_calendar SET events = events + ['Tour de
 France Stage 10'] WHERE year = 2015 AND month = 06;
```

  These update operations are implemented internally without any read-before-write. Appending and prepending a new element to the list writes only the new element.
- Add an element at a particular position using the list index position in square brackets.

```
cqlsh> UPDATE cycling.upcoming_calendar SET events[2] = 'Vuelta Ciclista a
 Venezuela' WHERE year = 2015 AND month = 06;
```

  To add an element at a particular position, Cassandra reads the entire list, and then rewrites the part of the list that needs to be shifted to the new index positions. Consequently, adding an element at a particular position results in greater latency than appending or prefixing an element to a list.
- Remove an element from a list, use the DELETE command and the list index position in square brackets. For example, remove the event just placed in the list in the last step.

```
cqlsh> DELETE events[2] FROM cycling.upcoming_calendar WHERE year = 2015
 AND month = 06;
```

  The method of removing elements using an indexed position from a list requires an internal read. In addition, the client-side application could only discover the indexed position by reading the whole list and finding the values to remove, adding additional latency to the operation. If another thread or client prepends elements to the list before the operation is done, incorrect data will be removed.
- Remove all elements having a particular value using the UPDATE command, the subtraction operator (-), and the list value in square brackets.

```
cqlsh> UPDATE cycling.upcoming_calendar SET events = events - ['Tour de
 France Stage 10'] WHERE year = 2015 AND month = 06;
```

  Using the UPDATE command as shown in this example is recommended over the last example because it is safer and faster.

Inserting and updating data into a map

If a table specifies a map to hold data, then either INSERT or UPDATE is used to enter data.

Procedure

- Set or replace map data, using the INSERT or UPDATE command, and enclosing the integer and text values in a map collection with curly brackets, separated by a colon.

```
cqlsh> INSERT INTO cycling.cyclist_teams (id, lastname, firstname, teams)
VALUES (
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2,
  'VOS',
  'Marianne',
  {2015 : 'Rabobank-Liv Woman Cycling Team', 2014 : 'Rabobank-Liv Woman
 Cycling Team', 2013 : 'Rabobank-Liv Giant',
    2012 : 'Rabobank Women Team', 2011 : 'Nederland bloeit' });
```

  Note: Using INSERT in this manner will replace the entire map.

- Use the UPDATE command to insert values into the map. Append an element to the map by enclosing the key-value pair in curly brackets and using the addition (+) operator.

```
cqlsh> UPDATE cycling.cyclist_teams SET teams = teams + {2009 : 'DSB Bank
 - Nederland bloeit'} WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Set a specific element using the UPDATE command, enclosing the specific key of the element, an integer, in square brackets, and using the equals operator to map the value assigned to the key.

```
cqlsh> UPDATE cycling.cyclist_teams SET teams[2006] = 'Team DSB - Ballast
 Nedam' WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

- Delete an element from the map using the DELETE command and enclosing the specific key of the element in square brackets:

```
cqlsh> DELETE teams[2009] FROM cycling.cyclist_teams WHERE id=e7cd5752-
bc0d-4157-a80f-7523add8dbcd;
```

- Alternatively, remove all elements having a particular value using the UPDATE command, the subtraction operator (-), and the map key values in curly brackets.

```
cqlsh> UPDATE cycling.cyclist_teams SET teams = teams - {'2013','2014'}
 WHERE id=e7cd5752-bc0d-4157-a80f-7523add8dbcd;
```

Inserting tuple data into a table

Tuples are used to group small amounts of data together that are then stored in a column.

Procedure

- Insert data into the table cycling.route which has tuple data. The tuple is enclosed in parentheses. This tuple has a tuple nested inside; nested parentheses are required for the inner tuple, then the outer tuple.

```
cqlsh> INSERT INTO cycling.route (race_id, race_name, point_id, lat_long)
 VALUES (500, '47th Tour du Pays de Vaud', 2, ('Champagne', (46.833,
 6.65)));
```

- Insert data into the table cycling.nation_rank which has tuple data. The tuple is enclosed in parentheses. The tuple called info stores the rank, name, and point total of each cyclist.

```
cqlsh> INSERT INTO cycling.nation_rank (nation, info) VALUES ('Spain',
 (1,'Alejandro VALVERDE' , 9054));
```

- Insert data into the table popular which has tuple data. The tuple called cinfo stores the country name, cyclist name, and points total.

```
cqlsh> INSERT INTO cycling.popular (rank, cinfo) VALUES (4, ('Italy',
 'Fabio ARU', 163));
```

Inserting or updating data into a user-defined type (UDT)

If a table specifies a user-defined type (UDT) to hold data, then either INSERT or UPDATE is used to enter data.

Procedure

Inserting data into a UDT

- Set or replace user-defined type data, using the `INSERT` or `UPDATE` command, and enclosing the user-defined type with curly brackets, separating each key-value pair in the user-defined type by a colon.

```
cqlsh> INSERT INTO cycling.cyclist_stats (id, lastname, basics) VALUES (
  e7ae5cf3-d358-4d99-b900-85902fda9bb0,
  'FRAME',
  { birthday : '1993-06-18', nationality : 'New Zealand', weight : null,
 height : null }
);
```

  Note: Note the inclusion of null values for UDT elements that have no value. A value, whether null or otherwise, must be included for each element of the UDT.

- Data can be inserted into a UDT that is nested in another column type. For example, a list of races, where the race name, date, and time are defined in a UDT has elements enclosed in curly brackets that are in turn enclosed in square brackets.

```
cqlsh> INSERT INTO cycling.cyclist_races (id, lastname, firstname, races)
 VALUES (
  5b6962dd-3f90-4c93-8f61-eabfa4a803e2,
  'VOS',
  'Marianne',
  [ { race_title : 'Rabobank 7-Dorpenomloop Aalburg',race_date :
 '2015-05-09',race_time : '02:58:33' },
  { race_title : 'Ronde van Gelderland',race_date :
 '2015-04-19',race_time : '03:22:23' } ]
);
```

  Note: The UDT nested in the list is frozen, so the entire list will be read when querying the table.

Updating individual field data in a UDT

- In Cassandra 3.6 and later, user-defined types that include only non-collection fields can update individual field values. Update an individual field in user-defined type data using the `UPDATE` command. The desired key-value pair are defined in the command. In order to update, the UDT must be defined in the `CREATE TABLE` command as an unfrozen data type.

```
cqlsh> CREATE TABLE cycling.cyclist_stats ( id UUID, lastname text, basics
 basic_info, PRIMARY KEY (id) );
INSERT INTO cycling.cyclist_stats (id, lastname, basics)
    VALUES (220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO',
 { birthday:'1977-07-08',nationality:'Italy',weight:'63 kg',height:'1.78
 m' });
UPDATE cyclist_stats SET basics.birthday = '2000-12-12' WHERE id =
 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
```

  The UDT is defined in the table with `basics basic_info`. This example shows an inserted row, followed by an update that only updates the value of `birthday` inside the UDT `basics`.

```
cqlsh:cycling> SELECT * FROM cycling.cyclist_stats WHERE id =
 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;

 id                                   | basics
                                                                                |
 lastname
--------------------------------------
+--------------------------------------------------------------------------------
+-----------
```

```
220844bf-4860-49d6-9a4b-6b5d3a79cbfb | {birthday: '2000-12-12
08:00:00.000000+0000', nationality: 'Italy', weight: '63 kg', height: '1.78
m'} | TIRALONGO
```

The resulting change is evident, as is the unchanged values for nationality, weight, and height.

Note: UDTs with collection fields must be frozen in table creation, and individual field values cannot be updated.

Inserting JSON data into a table

In a production database, inserting columns and column values programmatically is more practical than using cqlsh, but often, testing queries using this SQL-like shell is very convenient. With Cassandra 2.2 and later, JSON data can be inserted. All values will be inserted as a string if they are not a number, but will be stored using the column data type. For example, the id below is inserted as a string, but is stored as a UUID. For more information, see What's New in Cassandra 2.2: JSON Support.

Procedure

- To insert JSON data, add `JSON` to the `INSERT` command.. Note the absence of the keyword `VALUES` and the list of columns that is present in other `INSERT` commands.

```
cqlsh> INSERT INTO cycling.cyclist_category JSON '{
  "category" : "GC",
  "points" : 780,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda",
  "lastname" : "SUTHERLAND" }';
```

- A null value will be entered if a defined column like lastname, is not inserted into a table using JSON format.

```
cqlsh> INSERT INTO cycling.cyclist_category JSON '{
  "category" : "Sprint",
  "points" : 700,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda"
}';
```

```
 category | points | id                                   | lastname
----------+--------+--------------------------------------+-----------
   Sprint |    700 | 829aa84a-4bba-411f-a4fb-38167a987cda |      null
       GC |   1269 | 220844bf-4860-49d6-9a4b-6b5d3a79cbfb |  TIRALONGO
       GC |    780 | 829aa84a-4bba-411f-a4fb-38167a987cda | SUTHERLAND
```

Using lightweight transactions

INSERT and UPDATE statements using the `IF` clause support lightweight transactions, also known as Compare and Set (CAS). A common use for lightweight transactions is an insertion operation that must be unique, such as a cyclist's identification. Lightweight transactions should not be used casually, as the latency of operations increases fourfold due to the due to the round-trips necessary between the CAS coordinators.

Cassandra 2.1.1 and later support non-equal conditions for lightweight transactions. You can use <, <=, >, >=, != and IN operators in WHERE clauses to query lightweight tables.

It is important to note that using `IF NOT EXISTS` on an `INSERT`, the timestamp will be designated by the lightweight transaction, and USING TIMESTAMP is prohibited.

Procedure

- Insert a new cyclist with their id.

```
cqlsh> INSERT INTO cycling.cyclist_name (id, lastname, firstname)
  VALUES (4647f6d3-7bd2-4085-8d6c-1229351b5498, 'KNETEMANN', 'Roxxane')
  IF NOT EXISTS;
```

- Perform a CAS operation against a row that does exist by adding the predicate for the operation at the end of the query. For example, reset Roxane Knetemann's firstname because of a spelling error.

```
cqlsh> UPDATE cycling.cyclist_name
  SET firstname = ‘Roxane’
  WHERE id = 4647f6d3-7bd2-4085-8d6c-1229351b5498
  IF firstname = ‘Roxxane’;
```

Expiring data with Time-To-Live

Data in a column, other than a counter column, can have an optional expiration period called TTL (time to live). The client request specifies a TTL value, defined in seconds, for the data. After the requested amount of time has expired, TTL data is no longer live and will not be included in results. TTL data is marked with a tombstone after on the next read on the read path and exists for a maximum of gc_grace_seconds. After data is marked with a tombstone, the data is automatically removed during the normal compaction and repair processes.

Use CQL to set the TTL for data.

If you want to change the TTL of expiring data, the data must be re-inserted with a new TTL. In Cassandra, the insertion of data is actually an insertion or update operation, depending on whether or not a previous version of the data exists.

TTL data has a precision of one second, as calculated on the server. Therefore, a very small TTL is not very useful. Moreover, the clocks on the servers must be synchronized; otherwise reduced precision will be observed because the expiration time is computed on the primary host that receives the initial insertion but is then interpreted by other hosts on the cluster.

Expiring data has an additional overhead of 8 bytes in memory and on disk (to record the TTL and expiration time) compared to standard data.

Expiring data with TTL example

Both the `INSERT` and `UPDATE` commands support setting a time for data in a column to expire. The expiration time (TTL) is set using CQL.

Procedure

- Use the `INSERT` command to set calendar listing in the calendar table to expire in 86400 seconds, or one day.

```
cqlsh> INSERT INTO cycling.calendar (race_id, race_name, race_start_date,
 race_end_date) VALUES (200, 'placeholder', '2015-05-27', '2015-05-27')
 USING TTL 86400;
```

- Extend the expiration period to three days by using the UPDATE command and change the race name.

```
cqlsh> UPDATE cycling.calendar USING TTL 259200
  SET race_name = 'Tour de France - Stage 12'
  WHERE race_id = 200 AND race_start_date = '2015-05-27' AND race_end_date
 = '2015-05-27';
```

Inserting data using COPY and a CSV file

In a production database, inserting columns and column values programmatically is more practical than using cqlsh, but often, testing queries using this SQL-like shell is very convenient. A comma-delimited file, or CSV file, is useful if several records need inserting. While not strictly an INSERT command, it is a common method for inserting data.

Procedure

1. Locate your CSV file and check options to use.

```
category|point|id|lastname
GC|1269|2003|TIRALONGO
One-day-races|367|2003|TIRALONGO
GC|1324|2004|KRUIJSWIJK
```

2. To insert the data, using the COPY command with CSV data.

```
$ COPY cycling.cyclist_catgory FROM 'cyclist_category.csv' WITH
 DELIMITER='|' AND HEADER=TRUE
```

Batching data insertion and updates

Batching is used to insert or update data in tables. Understanding the use of batching, if used, is crucial to performance.

Using and misusing batches

Batches are often mistakenly used in an attempt to optimize performance. Improved performance is not a reason to use batches. Batches place a burden on the coordinator for both logged and unlogged batches.

Batches are best used when a small number of tables must synchronize inserted or updated data. The number of partitions involved in a batch operation and thus potential for multi-node access, can increase the amount of time the operation takes to complete. Batch statements are logged, causing additional performance latency. For logged batches, the coordinator sends a batch log to two other nodes, so that if the coordinator fails, the batch will be retried by those nodes.

For unlogged batches, the coordinator manages all insert/update operations, causing that single node to do more work. If the partition keys for the operations are stored on more than one node, extra network hops occur.

Note: Unlogged batches are deprecated in Cassandra2.2+.

Batched statements can save network round-trips between the client and the server, and possibly between the server coordinator and the replicas. However, consider carefully before implementing batch operations, and decide if they are truly necessary. For information about the fastest way to load data, see "Cassandra: Batch loading without the Batch keyword."

Use of BATCH statement

Batch operations can be either beneficial or detrimental. Look at the examples below to see a good use of BATCH.

Procedure

- An example of a good batch that is logged. Note in the table definition for cyclist_expenses, the balance column is STATIC.

```
cqlsh> CREATE TABLE cycling.cyclist_expenses (
  cyclist_name text,
  balance float STATIC,
  expense_id int,
  amount float,
  description text,
  paid boolean,
  PRIMARY KEY (cyclist_name, expense_id)
);
```

- The first INSERT in the BATCH statement sets the balance to zero. The next two statements insert an expense and change the balance value. All the INSERT and UPDATE statements in this batch write to the same partition, keeping the latency of the write operation low.

```
cqlsh> BEGIN BATCH
  INSERT INTO cycling.cyclist_expenses (cyclist_name, balance) VALUES
 ('Vera ADRIAN', 0) IF NOT EXISTS;
  INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount,
 description, paid) VALUES ('Vera ADRIAN', 1, 7.95, 'Breakfast', false);
  APPLY BATCH;
```

  As explained in the BATCH statement reference, in Cassandra 2.0.6+ you can batch conditional updates. This example shows batching conditional updates combined with using static columns.

  Note: It would be reasonable to expect that an UPDATE to the balance could be included in this BATCH statement:

```
cqlsh> UPDATE cycling.cyclist_expenses SET balance = -7.95 WHERE
 cyclist_name = 'Vera ADRIAN' IF balance = 0;
```

  However, it is important to understand that all the statements processed in a BATCH statement timestamp the records with the same value. The operations may not perform in the order listed in the BATCH statement. The UPDATE might be processed BEFORE the first INSERT that sets the balance value to zero, allowing the conditional to be met.

  An acknowledgement of a batch statement is returned if the batch operation is successful.

```
 [applied]
-----------
      True
```

The resulting table will only have one record so far.

```
 cyclist_name | expense_id | balance | amount | description | paid
--------------+------------+---------+--------+-------------+-------
  Vera ADRIAN |          1 |       0 |   7.95 |   Breakfast | False
```

- The balance can be adjusted separately with an UPDATE statement. Now the balance will reflect that breakfast was unpaid.

```
cqlsh> UPDATE cycling.cyclist_expenses SET balance = -7.95 WHERE
 cyclist_name = 'Vera ADRIAN' IF balance = 0;
```

```
 cyclist_name | expense_id | balance | amount | description | paid
--------------+------------+---------+--------+-------------+-------
  Vera ADRIAN |          1 |   -7.95 |   7.95 |   Breakfast | False
```

- The table cyclist_expenses stores records about each purchase by a cyclist and includes the running balance of all the cyclist's purchases. Because the balance is static, all purchase records for a cyclist have the same running balance. This `BATCH` statement inserts expenses for two more meals changes the balance to reflect that breakfast and dinner were unpaid.

```
cqlsh> BEGIN BATCH
INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount,
 description, paid) VALUES ('Vera ADRIAN', 2, 13.44, 'Lunch', true);
INSERT INTO cycling.cyclist_expenses (cyclist_name, expense_id, amount,
 description, paid) VALUES ('Vera ADRIAN', 3, 25.00, 'Dinner', false);
UPDATE cycling.cyclist_expenses SET balance = -32.95 WHERE cyclist_name =
 'Vera ADRIAN' IF balance = -7.95;
APPLY BATCH;
```

```
 cyclist_name | expense_id | balance | amount | description | paid
--------------+------------+---------+--------+-------------+-------
  Vera ADRIAN |          1 |  -32.95 |   7.95 |   Breakfast | False
  Vera ADRIAN |          2 |  -32.95 |  13.44 |       Lunch |  True
  Vera ADRIAN |          3 |  -32.95 |     25 |      Dinner | False
```

- Finally, the cyclist pays off all outstanding bills and the balance of the account goes to zero.

```
caqlsh> BEGIN BATCH
UPDATE cycling.cyclist_expenses SET balance = 0 WHERE cyclist_name = 'Vera
 ADRIAN' IF balance = -32.95;
UPDATE cycling.cyclist_expenses SET paid = true WHERE cyclist_name = 'Vera
 ADRIAN' AND expense_id = 1 IF paid = false;
UPDATE cycling.cyclist_expenses SET paid = true WHERE cyclist_name = 'Vera
 ADRIAN' AND expense_id = 3 IF paid = false;
APPLY BATCH;
```

```
 cyclist_name | expense_id | balance | amount | description | paid
--------------+------------+---------+--------+-------------+------
  Vera ADRIAN |          1 |       0 |   7.95 |   Breakfast | True
  Vera ADRIAN |          2 |       0 |  13.44 |       Lunch | True
  Vera ADRIAN |          3 |       0 |     25 |      Dinner | True
```

Because the column is static, you can provide only the partition key when updating the data. To update a non-static column, you would also have to provide a clustering key. Using batched conditional updates, you can maintain a running balance. If the balance were stored in a separate table, maintaining a running balance would not be possible because a batch having conditional updates cannot span multiple partitions.

Misuse of BATCH statement

Misused, `BATCH` statements can cause many problems in a distributed database like Cassandra. Batch operations that involve multiple nodes are a definite anti-pattern. Keep in mind which partition data will be written to when grouping `INSERT` and `UPDATE` statements in a `BATCH` statement. Writing to several partitions might require interaction with several nodes in the cluster, causing a great deal of latency for the write operation.

Procedure

- This example shows an anti-pattern since the `BATCH` statement will write to several different partitions, given the partition key id.

```
cqlsh> BEGIN BATCH
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
 (6d5f1663-89c0-45fc-8cfd-60a373b01622,'HOSKINS', 'Melissa');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
 (38ab64b6-26cc-4de9-ab28-c257cf011659,'FERNANDES', 'Marcia');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
 (9011d3be-d35c-4a8d-83f7-a3c543789ee7,'NIEWIADOMA', 'Katarzyna');
INSERT INTO cycling.cyclist_name (id, lastname, firstname) VALUES
 (95addc4c-459e-4ed7-b4b5-472f19a67995,'ADRIAN', 'Vera');
APPLY BATCH;
```

Using unlogged batches

Unlogged BATCH statements require management by the coordinator. Best case scenario for using unlogged BATCH statements is when inserts will all take place on a single node.

Procedure

- An unlogged batch that writes to a single partition resolves to only one write internally, regardless of the number of writes, and is an acceptable use of batch. In this example, the partition key includes both date and time.

```
cqlsh> BEGIN UNLOGGED BATCH;
  INSERT INTO sensor_readings (date, time, reading) values
 (20140910,'2014-09-10T11:00:00.00+0000', 6335.2);
  INSERT INTO sensor_readings (date, time, reading) values
 (20140910,'2014-09-10T11:00:15.00+0000', 5222.2);
APPLY BATCH;
```

Querying tables

Data can be queried from tables using the SELECT command. With Cassandra 3.0, many new options are available, such as retrieving JSON data, using standard aggregate functions, and manipulating retrieved data with user-defined functions (UDFs) and user-defined aggregate functions (UDAs).

Retrieval and sorting results

Querying tables to select data is the reason data is stored in databases. Similar to SQL, CQL can SELECT data using simple or complex qualifiers. At its simplest, a query selects all data in a table. At its most complex, a query delineates which data to retrieve and display and even calculate new values based on user-defined functions. For SASI indexing, see queries in Using SASI.

Controlling the number of rows returned using PER PARTITION LIMIT

In Cassandra 3.6 and later, the `PER PARTITION LIMIT` option sets the maximum number of rows that the query returns from each partition. Create a table that will sort data into more than one partition.

```
CREATE TABLE cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank) );
```

After inserting data, the table holds:

```
race_year | race_name                                    | rank | cyclist_name
-----------+----------------------------------------------+------
+----------------------
      2014 |                       4th Tour of Beijing |    1 |
 Phillippe GILBERT
      2014 |                       4th Tour of Beijing |    2 |
 Daniel MARTIN
      2014 |                       4th Tour of Beijing |    3 | Johan
 Esteban CHAVES
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |
 Ilnur ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |      Carlos
 BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |
 Benjamin PRADES
```

```
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |
 Adam PHELAN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |
 Thomas LEBAS
```

Now, to get the top two racers in every race year and race name, use the following command with PER PARTITION LIMIT 2.

```
SELECT * FROM cycling.rank_by_year_and_name PER PARTITION LIMIT 2;
```

Output:

```
race_year | race_name                                    | rank | cyclist_name
-----------+----------------------------------------------+------
+-------------------
      2014 |                        4th Tour of Beijing |    1 | Phillippe
 GILBERT
      2014 |                        4th Tour of Beijing |    2 |     Daniel
 MARTIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |     Ilnur
 ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |   Carlos
 BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |   Benjamin
 PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |       Adam
 PHELAN
```

Procedure

- Use a simple SELECT query to display all data from a table.

```
cqlsh> SELECT * FROM cycling.cyclist_category;
```

```
 category | id | lastname | points
----------+----+----------+--------
   SPRINT |  1 |    TERRI |     34
   SPRINT |  2 |      JIM |    120
       GC |  1 |    TERRI |   1234
       GC |  2 |      JIM |   2234
```

- The example below illustrates how to create a query that uses category as a filter.

```
cqlsh> SELECT * FROM cycling.cyclist_category WHERE category = 'SPRINT';
```

```
 category | id | lastname | points
----------+----+----------+--------
   SPRINT |  1 |    TERRI |     34
   SPRINT |  2 |      JIM |    120
```

Note that Cassandra will reject this query if category is not a partition key or clustering column. Queries require a sequential retrieval across the entire cyclist_category table. In a distributed database like Cassandra, this is a crucial concept to grasp; scanning all data across all nodes is prohibitively slow and thus blocked from execution. The use of partition key and clustering columns in a `WHERE` clause must result in the selection of a contiguous set of rows.

Queries can filter using secondary indexes, discussed in the Indexing Tables section. A query based on lastname can result in satisfactory results if the lastname column is indexed.

- In Cassandra 3.6 and later, clustering columns can be defined in `WHERE` clauses if `ALLOW FILTERING` is also used even if a secondary index is not created. The table definition is given and then the `SELECT` command. Note that `race_start_date` is a clustering column that has no secondary index.

```
CREATE TABLE cycling.calendar (
    race_id int,
    race_name text, race_start_date timestamp, race_end_date timestamp,
 PRIMARY KEY (race_id, race_start_date, race_end_date));
SELECT * FROM cycling.calendar WHERE race_start_date='2015-06-13' ALLOW
 FILTERING;
```

```
 race_id | race_start_date                 | race_end_date                   | race_name
---------+---------------------------------+---------------------------------+---------------
     102 | 2015-06-13 07:00:00.000000+0000 | 2015-06-21 07:00:00.000000+0000 | Tour de Suisse
```

- You can also pick the columns to display instead of choosing all data.

```
cqlsh> SELECT category, points, lastname FROM cycling.cyclist_category;
```

```
 category | points | lastname
----------+--------+----------
   SPRINT |     34 |    TERRI
   SPRINT |    120 |      JIM
       GC |   1234 |    TERRI
       GC |   2234 |      JIM
```

- For a large table, limit the number of rows retrieved using `LIMIT`. The default limit is 10,000 rows. To sample data, pick a smaller number. To retrieve more than 10,000 rows set `LIMIT` to a large value.

```
cqlsh> SELECT * From cycling.cyclist_name LIMIT 3;
```

```
 id                                   | firstname | lastn
--------------------------------------+-----------+------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 |      Alex |     F
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |
 220844bf-4860-49d6-9a4b-6b5d3a79cbfb |     Paolo | TIRAL
```

- You can fine-tune the display order using the `ORDER BY` clause. The partition key must be defined in the `WHERE` clause and the `ORDER BY` clause defines the clustering column to use for ordering.

```
cqlsh> CREATE TABLE cycling.cyclist_cat_pts ( category text, points int,
 id UUID,lastname text, PRIMARY KEY (category, points) );
SELECT * FROM cycling.cyclist_cat_pts WHERE category = 'GC' ORDER BY
 points ASC;
```

```
 category | points | id
----------+--------+--------------------------------------
       GC |    780 | 829aa84a-4bba-411f-a4fb-38167a987cda
       GC |   1269 | 220844bf-4860-49d6-9a4b-6b5d3a79cbfb
```

- Tuples are retrieved in their entirety. This example uses AS to change the header of the tuple name.

```
cqlsh> SELECT race_name, point_id, lat_long AS CITY_LATITUDE_LONGITUDE
 FROM cycling.route;
```

```
 race_name                  | point_id | city_latitude_longitude
----------------------------+----------+---------------------------------
 47th Tour du Pays de Vaud |        1 |    ('Onnens', (46.8444, 6.6667))
 47th Tour du Pays de Vaud |        2 |    ('Champagne', (46.833, 6.65))
 47th Tour du Pays de Vaud |        3 |       ('Novalle', (46.833, 6.6))
 47th Tour du Pays de Vaud |        4 |     ('Vuiteboeuf', (46.8, 6.55))
 47th Tour du Pays de Vaud |        5 |   ('Baulmes', (46.7833, 6.5333))
 47th Tour du Pays de Vaud |        6 | ('Les Clées', (46.7222, 6.5222))
```

- In Cassandra 3.6 and later, the `PER PARTITION LIMIT` option sets the maximum number of rows that the query returns from each partition. This is interesting because it allows a query to select a "Top 3" selection if the partitions are separated correctly. Create a table that will sort data into more than one partition and insert some data:

```
CREATE TABLE cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank) );
```

```
race_year | race_name                                    | rank |
 cyclist_name
-----------+----------------------------------------------+------
+----------------------
```

```
      2014 |                          4th Tour of Beijing |    1 |
 Phillippe GILBERT
      2014 |                          4th Tour of Beijing |    2 |
 Daniel MARTIN
      2014 |                          4th Tour of Beijing |    3 | Johan
 Esteban CHAVES
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |
 Ilnur ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |
 Carlos BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |
 Benjamin PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |
 Adam PHELAN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |
 Thomas LEBAS
```

- Now use a `PER PARTITION LIMIT` to get the top two races for each race year and race name pair:

```
SELECT * FROM cycling.rank_by_year_and_name PER PARTITION LIMIT 2;
```

```
race_year | race_name                                    | rank |
 cyclist_name
-----------+----------------------------------------------+------
+------------------
      2014 |                          4th Tour of Beijing |    1 | Phillippe
 GILBERT
      2014 |                          4th Tour of Beijing |    2 |
 Daniel MARTIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |     Ilnur
 ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |   Carlos
 BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |
 Benjamin PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |
 Adam PHELAN
```

Retrieval using collections

Collections do not differ from other columns in retrieval. To query for a subset of the collection, a secondary index for the collection must be created.

Procedure

- Retrieve teams for a particular cyclist id from the set.

```
cqlsh> SELECT lastname, teams FROM cycling.cyclist_career_teams WHERE id =
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

To query a table containing a collection, Cassandra retrieves the collection in its entirety. Keep collections small enough to be manageable because the collection store in memory. Alternatively, construct a data model to replace a collection if it must accommodate large amounts of data.

Cassandra returns results in an order based on the type of the elements in the collection. For example, a set of text elements is returned in alphabetical order. If you want elements of the collection returned in insertion order, use a list.

```
 lastname | teams
----------+-----------------------------------------------------------------------------------------------------
      VOS | {'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabobank-Liv Woman Cycling Team'}
```

- Retrieve events stored in a list from the upcoming calendar for a particular year and month.

```
cqlsh> SELECT * FROM cycling.upcoming_calendar WHERE year=2015 AND
 month=06;
```

```
 year | month | events
------+-------+-----------------------------------------------------------------------------------------------------
 2015 |     6 | ['The Parx Casino Philly Cycling Classic', 'Criterium du Dauphine', 'Vuelta Ciclista a Venezuela']
```

Note: The order is not alphabetical, but rather in the order of insertion.

- Retrieve teams for a particular cyclist id from the map.

```
cqlsh> SELECT lastname, firstname, teams FROM cycling.cyclist_teams WHERE
 id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

The order of the map output depends on the key type of the map. In this case, the key is an integer type.

```
 lastname | firstname | teams
----------+-----------+-----------------------------------------------------------------------------------------------------
-----------------------------------
      VOS |  Marianne | {2011: 'Nederland bloeit', 2012: 'Rabobank Women Team', 2013: 'Rabobank-Liv Giant', 2014: 'Rabobank-Liv Wom
an Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
```

Retrieval using JSON

The SELECT command can be used to retrieve data from a table in JSON format. For more information, see What's New in Cassandra 2.2: JSON Support.

Procedure

Specify that the result should use the JSON format with the keyword JSON.

```
cqlsh> SELECT JSON month, year, events FROM cycling.upcoming_calendar;
```

```
 [json]
-----------------------------------------------------------------------------------------------------------------
-----------------------------------
 {"month": 6, "year": 2015, "events": ["The Parx Casino Philly Cycling Classic", "Criterium du Dauphi
ne", "Vuelta Ciclista a Venezuela"]}
                                                                                                {"month": 7, "year":
 2015, "events": ["Tour de France"]}
             {"month": 8, "year": 2015, "events": ["Clasica Ciclista San Sebastian", "Tour de Pologne
", "Eneco Tour", "Vuelta a Espana"]}
```

Retrieval using the IN keyword

The IN keyword can define a set of clustering columns to fetch together, supporting a "multi-get" of CQL rows. A single clustering column can be defined if all preceding columns are defined for either equality or group inclusion. Alternatively, several clustering columns may be defined to collect several rows, as long as all preceding columns are queried for equality or group inclusion. The defined clustering columns can also be queried for inequality.

Note that using both IN and ORDER BY will require turning off paging with the PAGING OFF command in `cqlsh`.

Procedure

- Turn off paging.

```
cqlsh> PAGING OFF
```

- Retrieve and sort results in descending order.

```
cqlsh> SELECT * FROM cycling.cyclist_cat_pts WHERE category IN ('Time-
trial', 'Sprint') ORDER BY id DESC;
```

```
 category   | id                                   | points | lastname
------------+--------------------------------------+--------+------------
     Sprint | 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |     39 | KRUIJSWIJK
 Time-trial | 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |      3 | KRUIJSWIJK
     Sprint | 220844bf-4860-49d6-9a4b-6b5d3a79cbfb |      0 |  TIRALONGO
 Time-trial | 220844bf-4860-49d6-9a4b-6b5d3a79cbfb |    182 |  TIRALONGO
```

- Alternatively, retrieve and sort results in ascending order.

  To retrieve results, use the SELECT command.

```
cqlsh> SELECT * FROM cycling.cyclist_cat_pts WHERE category IN ('Time-
trial', 'Sprint') ORDER BY id ASC;
```

```
 category   | id                                   | points | lastname
------------+--------------------------------------+--------+------------
 Time-trial | 220844bf-4860-49d6-9a4b-6b5d3a79cbfb |    182 |  TIRALONGO
     Sprint | 220844bf-4860-49d6-9a4b-6b5d3a79cbfb |      0 |  TIRALONGO
 Time-trial | 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |      3 | KRUIJSWIJK
     Sprint | 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |     39 | KRUIJSWIJK
```

- Retrieve rows using multiple clustering columns. This example searches the partition key race_ids for several races, but the partition key can also be composed as an equality for one value.

```
cqlsh> SELECT * FROM cycling.calendar WHERE race_id IN (100, 101, 102)
 AND (race_start_date, race_end_date) IN (('2015-05-09','2015-05-31'),
('2015-05-06', '2015-05-31'));
```

```
 race_id | race_start_date          | race_end_date            | race_name
---------+--------------------------+--------------------------+---------------
     100 | 2015-05-09 00:00:00-0700 | 2015-05-31 00:00:00-0700 | Giro d'Italia
```

- Retrieve several rows using multiple clustering columns and inequality.

```
cqlsh> SELECT * FROM cycling.calendar WHERE race_id IN (100, 101, 102) AND
 (race_start_date, race_end_date) >= ('2015-05-09','2015-05-24');
```

```
 race_id | race_start_date          | race_end_date            | race_name
---------+--------------------------+--------------------------+----------------------
     100 | 2015-05-09 00:00:00-0700 | 2015-05-31 00:00:00-0700 |         Giro d'Italia
     101 | 2015-06-07 00:00:00-0700 | 2015-06-14 00:00:00-0700 | Criterium du Dauphine
     102 | 2015-06-13 00:00:00-0700 | 2015-06-21 00:00:00-0700 |        Tour de Suisse
```

Retrieval by scanning a partition

Queries can scan a partition to retrieve a segment of stored data. The segment must be sequentially stored, so clustering columns can be used to define the slice of data selected.

Procedure

- Create a table race_times to hold the race times of various cyclists for various races.

```
CREATE TABLE cycling.race_times (race_name text, cyclist_name text,
 race_time text, PRIMARY KEY (race_name, race_time));
```

```
race_name                        | race_time | cyclist_name
---------------------------------+-----------+---------------
17th Santos Tour Down Under |  19:15:18 | Rohan DENNIS
17th Santos Tour Down Under |  19:15:20 | Richie PORTE
17th Santos Tour Down Under |  19:15:38 |  Cadel EVANS
17th Santos Tour Down Under |  19:15:40 | Tom DUMOULIN
```

- Scan the race times in the table to find a particular segment of data using a conditional operator.

```
SELECT * FROM cycling.race_times WHERE race_name = '17th Santos Tour Down
 Under' AND race_time >= '19:15:19' AND race_time <= '19:15:39' );
```

```
 race_name                        | race_time | cyclist_name
----------------------------------+-----------+----------------
 17th Santos Tour Down Under |  19:15:20 | Richie PORTE
 17th Santos Tour Down Under |  19:15:38 |  Cadel EVANS
```

Retrieval using standard aggregate functions

In Cassandra 2.2, the standard aggregate functions of `min`, `max`, `avg`, `sum`, and `count` are built-in functions.

Procedure

- A table cyclist_points records the race points for cyclists.

```
cqlsh> CREATE TABLE cycling.cyclist_points (id UUID, firstname text,
 lastname text, race_title text, race_points int, PRIMARY KEY (id,
 race_points );
```

```
id                                   | race_points | firs
--------------------------------------+-------------+----
e3b19ec4-774a-4d1c-9e5a-decec1e30aac |           6 |   G
e3b19ec4-774a-4d1c-9e5a-decec1e30aac |          75 |   G
e3b19ec4-774a-4d1c-9e5a-decec1e30aac |         120 |   G
```

- Calculate the standard aggregation function `sum` to find the sum of race points for a particular cyclist. The value of the aggregate will be returned.

```
cqlsh> SELECT sum(race_points) FROM cycling.cyclist_points WHERE
 id=e3b19ec4-774a-4d1c-9e5a-decec1e30aac;
```

```
 system.sum(race_points)
-------------------------
                     201
```

- Another standard aggregate function is `count`. A table country_flag records the country of each cyclist.

```
CREATE TABLE cycling.country_flag (country text, cyclist_name text, flag
 int STATIC, PRIMARY KEY (country, cyclist_name));
```

```
 country | cyclist_name | flag
---------+--------------+------
 Belgium |        Andre |    1
 Belgium |      Jacques |    1
  France |        Andre |    3
  France |       George |    3
```

- Calculate the standard aggregation function `count` to find the number of cyclists from Belgium. The value of the aggregate will be returned.

```
cqlsh> SELECT count(cyclist_name) FROM cycling.country_flag WHERE
 country='Belgium';
```

```
 system.count(cyclist_name)
----------------------------
                          2
```

Retrieval using a user-defined function (UDF)

The SELECT command can be used to retrieve data from a table while applying a user-defined function (UDF) to it.

Procedure

Use the user-defined function (UDF) fLog() created previously to retrieve data from a table cycling.cyclist_points.

```
cqlsh> SELECT id, lastname, fLog(race_points) FROM cycling.cyclist_points;
```

```
id                                   | lastname   | cycling.flog(race_points)
--------------------------------------+------------+---------------------------
220844bf-4860-49d6-9a4b-6b5d3a79cbfb | TIRALONGO  |                  0.693147
e3b19ec4-774a-4d1c-9e5a-decec1e30aac |  BRONZINI  |                   1.79176
e3b19ec4-774a-4d1c-9e5a-decec1e30aac |  BRONZINI  |                   4.31749
e3b19ec4-774a-4d1c-9e5a-decec1e30aac |  BRONZINI  |                   4.78749
```

Retrieval using user-defined aggregate (UDA) functions

Referring back to the user-defined aggregate average(), retrieve the average of the column cyclist_time_sec from a table.

Procedure

1. List all the data in the table.

```
cqlsh> SELECT * FROM cycling.team_average;
```

```
team_name                                | race_title
                | cyclist_name     | cyclist_time_sec
-----------------------------------------+-----------------------------------------------------------------------------------------------------------------
-------------+------------------+-----------------
                TWENTY16 presented by Sho-Air | Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe
> Lake Tahoe |  Lauren KOMANSKI |            11451
 UnitedHealthCare Pro Cycling Womens Team | Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe
> Lake Tahoe |    Hannah BARNES |            11490
 UnitedHealthCare Pro Cycling Womens Team | Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe
> Lake Tahoe |       Katie HALL |            11449
 UnitedHealthCare Pro Cycling Womens Team | Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe
> Lake Tahoe |  Linda VILLUMSEN |            11485
                           Velocio-SRAM | Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe
> Lake Tahoe | Alena AMIALIUSIK |            11451
                           Velocio-SRAM | Amgen Tour of California Women's Race presented by SRAM - Stage 1 - Lake Tahoe
> Lake Tahoe |    Trixi WORRACK |            11453
```

2. Apply the user-defined aggregate function `average()` to the cyclist_time_sec column.

```
cqlsh> SELECT average(cyclist_time_sec) FROM cycling.team_average
 WHERE team_name='UnitedHealthCare Pro Cycling Womens Team' AND
 race_title='Amgen Tour of California Women''s Race presented by SRAM -
 Stage 1 - Lake Tahoe > Lake Tahoe';
```

```
 cycling.average(cyclist_time_sec)
-----------------------------------

                          11474.66667
```

Querying a system table

The system keyspace includes a number of tables that contain details about your Cassandra database objects and cluster configuration.

Cassandra 2.2 populates these tables and others in the system keyspace.

Table: Columns in System Tables - Cassandra 2.2

Table name	Column name	Comment
local	"key", bootstrapped, broadcast_address,cluster_name,cql_version,data_center,gossip_generation,host_id,native_proto truncated_at map	Information on a node has about itself and data center, gossip
peers	peer, data_center, host_id,preferred_ip,rack, release_version, rpc_address, schema_version, tokens	Each node records what other nodes tell it about themselves over the gossip.
schema_aggregates	keyspace_name, aggregate_name, signature, argument_types, final_func,initcond,return_type,state_func,state_type	Information about user-defined aggregates
schema_columnfamilies	See comment.	Inspect schema_columnfamilies to get detailed information about specific tables.
schema_columns	keyspace_name, columnfamily_name, column_name, component_index, index_name, index_options, index_type, validator	Information on columns and column indexes. Used internally for compound primary keys.
schema_functions	keyspace_name, function_name, signature, argument_names,argument_types, body,called_on_null_input,language,return_type	Information on user-defined functions
schema_keyspaces	keyspace_name, durable_writes, strategy_class, strategy_options	Information on keyspace strategy class and replication factor
schema_usertypes	keyspace_name,type_name,field_names,field_types	Information about user-defined types

Cassandra 3.0 populates these tables and others in the system keyspace.

Table: Columns in System Tables - Cassandra 3.0

Table name	Column name	Comment
available_ranges	keyspace_name,ranges	
batches	id,mutations,version	
batchlog	id,data,version,written_at	
built_views	keyspace_name,view_name	Information on materialized views
compaction_history	id,bytes_in,bytes_out,columnfamily_name,compacted_at,keyspace_name,rows_merged	Information on keyspace history
hints	target_id,hint_id,message_version,mutation	
"IndexInfo"	table_name,index_name	Information on indexes

Table name	Column name	Comment
local	"key", bootstrapped, broadcast_address,cluster_name,cql_version,data_center,gossip_generation,host_id,listen_addre truncated_at map	Information on a node has about itself and data center, rack, and gossip
paxos	row_key,cf_id,in_progress_ballot,most_recent_commit,most_recent_commit_at,most_recent_com	Information on lightweight Paxos transactions
peers	peer, data_center, host_id,preferred_ip,rack, release_version, rpc_address, schema_version, tokens	Each node records what other nodes tell it about themselves over the gossip.
peer_events	peer,hints_dropped	
range_xfers	token_bytes,requested_at	
size_estimates	keyspace_name,table_name,range_start,range_end,mean_partition_size,partitions_count	Information about token ranges and partitions
sstable_activity	keyspace_name,columnfamily_name,generation,rate_120m,rate_15m	
views_builds_in_progress	keyspace_name,view_name,generation_number,last_token	

Cassandra 3.0 populates these tables in the system_schema keyspace.

Table: Columns in System_Schema Tables - Cassandra 3.0

Table name	Column name	Comment
aggregates	keyspace_name, aggregate_name, argument_types, final_func,initcond,return_type,state_func,state_type	Information about user-defined aggregates
columns	keyspace_name,table_name,column_name,clustering_order,column_name_bytes,kind,position,ty	Information about table columns
dropped_columns	keyspace_name,table_name,column_name,dropped_time,type	Information about dropped columns
functions	keyspace_name, function_name, argument_types,argument_names, body,called_on_null_input,language,return_type	Information on user-defined functions
indexes	keyspace_name,table_name,index_name,kind,options	Information about indexes
keyspaces	keyspace_name, durable_writes, replication	Information on keyspace durable writes and replication
tables	keyspace_name, table_name,bloom_filter_fp_chance,caching,comment,compaction,compression,crc_check_chanc	Information on columns and column indexes used internally for compound primary keys.
triggers	keyspace_name,table_name,trigger_name,options	Information on triggers
types	keyspace_name,type_name,field_names,field_types	Information about user-defined types
views	keyspace_name,view_name,base_table_id,base_table_name,bloom_filter_fp_chance,caching,co	Information about materialized views

Keyspace, table, and column information

An alternative to the `cqlsh` describe_* functions or using `DevCenter` to discover keyspace, table, and column information is querying system.schema_* table directly.

Procedure

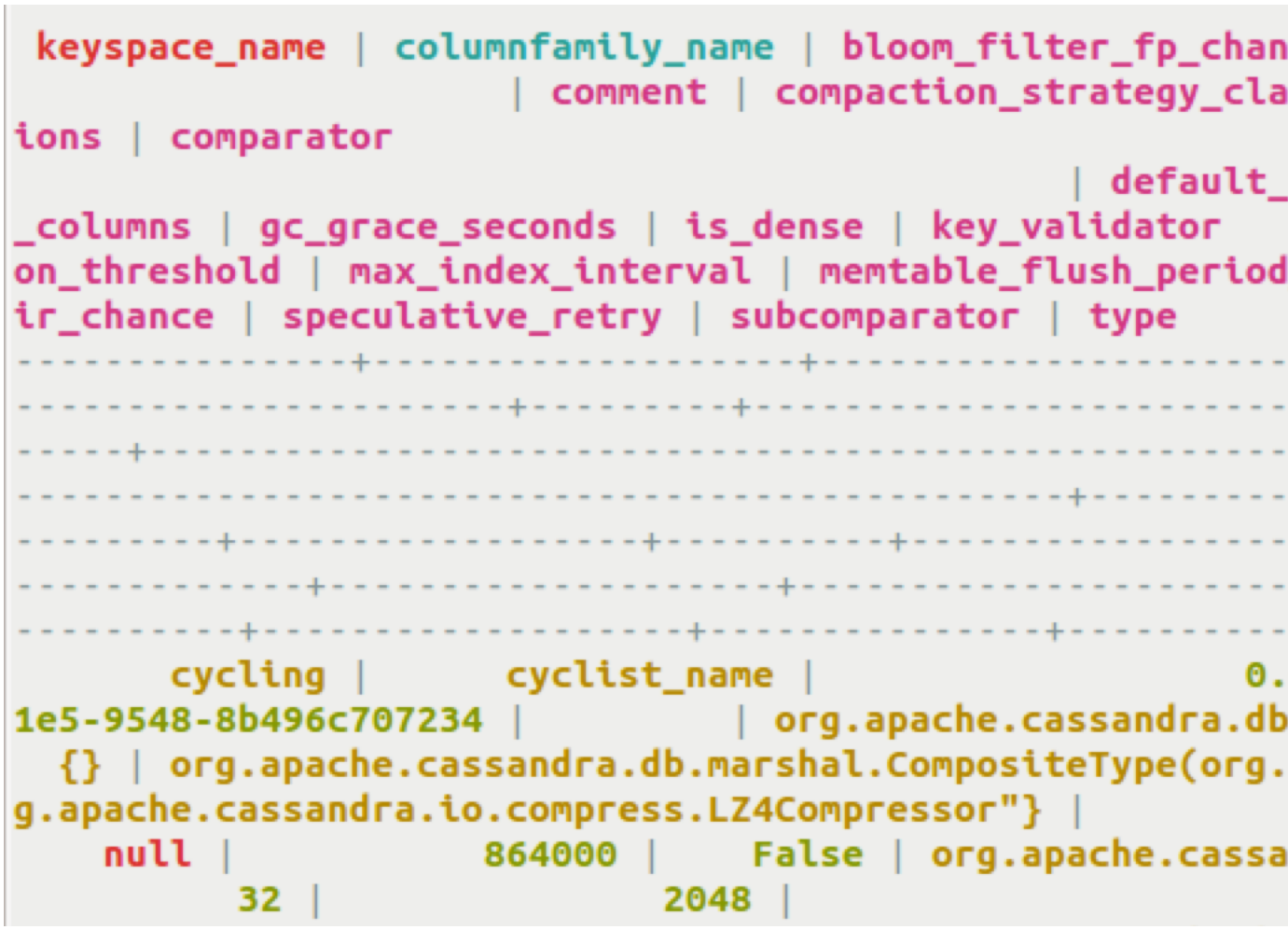
- Query the defined keyspaces using the SELECT statement.

```
cqlsh> SELECT * FROM system.schema_keyspaces;
```

```
keyspace_name      | durable_writes | strategy_class                                | strategy_options
-------------------+----------------+-----------------------------------------------+----------------------------
          cycling |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}
      system_auth |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}
system_distributed |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"3"}
           system |           True |  org.apache.cassandra.locator.LocalStrategy |                         {}
     system_traces |           True | org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"2"}
```

- Query schema_columnfamilies about a particular table.

```
cqlsh> SELECT * FROM system.schema_columnfamilies WHERE keyspace_name =
 'cycling' AND columnfamily_name = 'cyclist_name';
```



- Query schema_columns about the columns in a table.

```
cqlsh> SELECT * FROM system.schema_columns WHERE keyspace_name = 'cycling'
 AND columnfamily_name = 'cyclist_name';
```

```
 keyspace_name | columnfamily_name | column_name | compon
  | validator
---------------+-------------------+-------------+-------
--+----------------------------------------
       cycling |      cyclist_name |   firstname |
r | org.apache.cassandra.db.marshal.UTF8Type
       cycling |      cyclist_name |          id |
y | org.apache.cassandra.db.marshal.UUIDType
       cycling |      cyclist_name |    lastname |
r | org.apache.cassandra.db.marshal.UTF8Type
```

Cluster information

You can query system tables to get cluster topology information. Display the IP address of peer nodes, data center and rack names, token values, and other information. *"The Data Dictionary"* article describes querying system tables in detail.

Procedure

After setting up a cluster, query the peers and local tables.

```
SELECT * FROM system.peers;
```

Output from querying the peers table looks something like this:

```
   peer    | data_center | host_id     | preferred_ip | rack  |
 release_version | rpc_address | schema_version | tokens
----------+-------------+-------------+--------------+-------
+-----------------+-------------+----------------+-----------
 127.0.0.3 | datacenter1 | edda8d72... |         null | rack1 |
 2.1.0 | 127.0.0.3   | 59adb24e-f3... |  {3074...
 127.0.0.2 | datacenter1 | ef863afa... |         null | rack1 |
 2.1.0 | 127.0.0.2   | 3d19cd8f-c9... | {-3074...}

(2 rows)
```

Functions, aggregates, and user types

Currently, the system tables are the only method of displaying information about user-defined functions, aggregates, and user types.

Procedure

- Show all user-defined functions in the system.schema_functions table.

```
cqlsh> SELECT * FROM system.schema_functions;
```

```
 keyspace_name | function_name | signature                       | argument_names   | argument_types

             | body
              | called_on_null_input | language | return_type
---------------+---------------+---------------------------------+------------------+-----------------------------------
---------------------------------------------------------------------------------------------------------------------
---------------+-----------------------+----------+-----------------------------------------------------------------
---------------+-----------------------+----------+-----------------------------------------------------------------
-----------------------------------------
       cycling |      avgfinal |        ['tuple<int, bigint>'] |        ['state'] |
     ['org.apache.cassandra.db.marshal.TupleType(org.apache.cassandra.db.marshal.Int32Type,org.apache.cassandra.db.marshal
.LongType)'] | double r = 0; if (state.getInt(0) == 0) return null; r = state.getLong(1); r/= state.getInt(0); return Doubl
e.valueOf(r); |                  True |     java |
          org.apache.cassandra.db.marshal.DoubleType
       cycling |      avgstate | ['tuple<int, bigint>', 'int'] | ['state', 'val'] | ['org.apache.cassandra.db.marshal.Tuple
Type(org.apache.cassandra.db.marshal.Int32Type,org.apache.cassandra.db.marshal.LongType)', 'org.apache.cassandra.db.marshal
.Int32Type'] |  if (val !=null) { state.setInt(0, state.getInt(0)+1); state.setLong(1, state.getLong(1)+val.intValue()); }
return state; |                  True |     java | org.apache.cassandra.db.marshal.TupleType(org.apache.cassandra.db.marshal
.Int32Type,org.apache.cassandra.db.marshal.LongType)
       cycling |          flog |                    ['double'] |        ['input'] |
                                                                                  ['org.apache.cassandra.db.marshal.
DoubleType'] |                                                                  return Double.valueOf(Math.log(input.dou
bleValue())); |                  True |     java |
          org.apache.cassandra.db.marshal.DoubleType
       cycling |          fsin |                    ['double'] |        ['input'] |
                                                                                  ['org.apache.cassandra.db.marshal.
DoubleType'] |                                                                 return Double.valueOf(Math.sin(input.doub
leValue()));  |                  True |     java |
          org.apache.cassandra.db.marshal.DoubleType
```

- Show all user-defined aggregates in the system.schema_aggregates table.

```
cqlsh> SELECT * FROM system.schema_aggregates;
```

```
 keyspace_name | aggregate_name | signature | argument_types                                | final_func | initcond
                        | return_type                                | state_func | state_type
---------------+----------------+-----------+----------------------------------------------+------------+-----------------
-----------------------+--------------------------------------------+------------+-----------------------------------
-------------------------------------------------------------------------------
       cycling |        average |   ['int'] | ['org.apache.cassandra.db.marshal.Int32Type'] |   avgfinal | 0x00000004000000
00000000080000000000000000 | org.apache.cassandra.db.marshal.DoubleType |   avgstate | org.apache.cassandra.db.marshal.Tupl
eType(org.apache.cassandra.db.marshal.Int32Type,org.apache.cassandra.db.marshal.LongType)
```

- Show all user-defined types in the system.schema_usertypes table.

```
cqlsh> SELECT * FROM system.schema_usertypes;
```

```
 keyspace_name | type_name  | field_names                                                    | field_types
---------------+------------+----------------------------------------------------------------+-----------------------------
-----------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------
       cycling | basic_info | ['birthday', 'lastname', 'nationality', 'weight', 'height'] | ['org.apache.cassandra.db.marsh
al.TimestampType', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cass
andra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']
```

Indexing tables

Data can be queried from tables using indexes, once created.

Indexing

An index provides a means to access data in Cassandra using attributes other than the partition key. The benefit is fast, efficient lookup of data matching a given condition. The index indexes column values in a separate, hidden table from the one that contains the values being indexed. Cassandra has a number of techniques for guarding against the undesirable scenario where data might be incorrectly retrieved during a query involving indexes on the basis of stale values in the index.

Indexes can be used for collections, collection columns, static columns, and any other columns except counter columns.

In Cassandra 3.4 and later, SSTable Attached Secondary Indexes (SASI) have been introduced.

When to use an index

Cassandra's built-in indexes are best on a table having many rows that contain the indexed value. The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index. For example, suppose you had a races table with a billion entries for cyclists in hundreds of races and wanted to look up rank by the cyclist. Many cyclists' ranks will share the same column value for race year. The race_year column is a good candidate for an index.

In Cassandra 3.4 and later, a new implementation of secondary indexes, SSTable Attached Secondary Indexes (SASI) have greatly improved the performance of secondary indexes. If secondary indexes are required, SASI should be considered.

When *not* to use an index

Do not use an index in these situations:

- On high-cardinality columns for a query of a huge volume of records for a small number of results. See Problems using a high-cardinality column index below.
- In tables that use a counter column.
- On a frequently updated or deleted column. See Problems using an index on a frequently updated or deleted column below.
- To look for a row in a large partition unless narrowly queried. See Problems using an index to look for a row in a large partition unless narrowly queried below.

Problems using a high-cardinality column index

If you create an index on a high-cardinality column, which has many distinct values, a query between the fields will incur many seeks for very few results. In the table with a billion songs, looking up songs by writer (a value that is typically unique for each song) instead of by their artist, is likely to be very inefficient. It would probably be more efficient to manually maintain the table as a form of an index instead of using the Cassandra built-in index. For columns containing unique data, it is sometimes fine performance-wise to use an index for convenience, as long as the query volume to the table having an indexed column is moderate and not under constant load.

Conversely, creating an index on an extremely low-cardinality column, such as a boolean column, does not make sense. Each value in the index becomes a single row in the index, resulting in a huge row for all the false values, for example. Indexing a multitude of indexed columns having foo = true and foo = false is not useful.

Problems using an index on a frequently updated or deleted column

Cassandra stores tombstones in the index until the tombstone limit reaches 100K cells. After exceeding the tombstone limit, the query that uses the indexed value will fail.

Problems using an index to look for a row in a large partition unless narrowly queried

A query on an indexed column in a large cluster typically requires collating responses from multiple data partitions. The query response slows down as more machines are added to the cluster. You can avoid a performance hit when looking for a row in a large partition by narrowing the search.

Using a secondary index

Using CQL, you can create an index on a column after defining a table. In Cassandra 2.1 and later, you can index a collection column. In Cassandra 3.4 and later, static columns can be indexed. Secondary indexes are used to query a table using a column that is not normally queryable.

Secondary indexes are tricky to use and can impact performance greatly. The index table is stored on each node in a cluster, so a query involving a secondary index can rapidly become a performance nightmare

if multiple nodes are accessed. A general rule of thumb is to index a column with low cardinality of few values. Before creating an index, be aware of when and when not to create an index.

In Cassandra 3.4 and later, a new implementation of secondary indexes, SSTable Attached Secondary Indexes (SASI) have greatly improved the performance of secondary indexes and should be used, if possible.

Procedure

- The table rank_by_year_and_name can yield the rank of cyclists for races.

```
cqlsh> CREATE TABLE cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank)
);
```

- Both race_year and race_name must be specified as these columns comprise the partition key.

```
cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015
 AND race_name='Tour of Japan - Stage 4 - Minami > Shinshu';
```

```
 race_year | race_name                                        | rank | cyclist_name
-----------+--------------------------------------------------+------+-----------------
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 | Benjamin PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |     Adam PHELAN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |    Thomas LEBAS
```

- A logical query to try is a listing of the rankings for a particular year. Because the table has a composite partition key, this query will fail if only the first column is used in the conditional operator.

```
cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;
```

```
cqlsh:cycling> SELECT * from rank_by_year_and_name where race_year=2015;
InvalidRequest: code=2200 [Invalid query] message="Partition key parts: race_name must be restricted as other parts are"
```

- An index is created for the race year, and the query will succeed. An index name is optional and must be unique within a keyspace. If you do not provide a name, Cassandra will assign a name like race_year_idx.

```
cqlsh> CREATE INDEX ryear ON cycling.rank_by_year_and_name (race_year);
SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;
```

```
 race_year | race_name                                        | rank | cyclist_name
-----------+--------------------------------------------------+------+-----------------
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |   Ilnur ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 | Carlos BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 | Benjamin PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |     Adam PHELAN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |    Thomas LEBAS
```

- A clustering column can also be used to create an index. An index is created on rank, and used in a query.

```
cqlsh> CREATE INDEX rrank ON cycling.rank_by_year_and_name (rank);
SELECT * FROM cycling.rank_by_year_and_name WHERE rank = 1;
```

```
 race_year | race_name                                        | rank | cyclist_name
-----------+--------------------------------------------------+------+--------------------
      2014 |                          4th Tour of Beijing |    1 | Phillippe GILBERT
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |      Ilnur ZAKARIN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |   Benjamin PRADES
```

Using multiple indexes

Indexes can be created on multiple columns and used in queries. The general rule about cardinality applies to all columns indexed. In a real-world situation, certain columns might not be good choices, depending on their cardinality.

Procedure

- The table cycling.alt_stats can yield the statistics about cyclists.

```
cqlsh> CREATE TABLE cycling.cyclist_alt_stats ( id UUID PRIMARY KEY,
 lastname text, birthday timestamp, nationality text, weight text, height
 text );
```

- Create indexes on the columns birthday and nationality.

```
cqlsh> CREATE INDEX birthday_idx ON cycling.cyclist_alt_stats
 ( birthday );
CREATE INDEX nationality_idx ON cycling.cyclist_alt_stats ( nationality );
```

- Query for all the cyclists with a particular birthday from a certain country.

```
cqlsh> SELECT * FROM cycling.cyclist_alt_stats WHERE birthday =
 '1982-01-29' AND nationality = 'Russia';
```

```
cqlsh:cycling> SELECT * FROM cycling.cyclist_alt_stats WHERE birthday = '1982-01-29' AND nationality = 'Russia';
InvalidRequest: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus ma
y have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FIL
TERING"
```

- The indexes have been created on appropriate low cardinality columns, but the query still fails. Why? The answer lies with the partition key, which has not been defined. When you attempt a potentially expensive query, such as searching a range of rows, Cassandra requires the ALLOW FILTERING directive. The error is not due to multiple indexes, but the lack of a partition key definition in the query.

```
cqlsh> SELECT * FROM cycling.cyclist_alt_stats WHERE birthday =
 '1990-05-27' AND nationality = 'Portugal' ALLOW FILTERING
```

```
 id                                   | birthday                | height | lastname | nationality | weight
--------------------------------------+-------------------------+--------+----------+-------------+--------
 1ba0417d-62da-4103-b710-de6fb227db6f | 1990-05-27 00:00:00-0700 |   null | PAULINHO |    Portugal |   null
```

Indexing a collection

Collections can be indexed and queried to find a collection containing a particular value. Sets and lists are indexed slightly differently from maps, given the key-value nature of maps.

Sets and lists can index all values found by indexing the collection column. Maps can index a map key, map value, or map entry using the methods shown below. Multiple indexes can be created on the same

map column in a table, so that map keys, values, or entries can be queried. In addition, frozen collections can be indexed using FULL to index the full content of a frozen collection.

Note: All the cautions about using secondary indexes apply to indexing collections.

Procedure

- For set and list collections, create an index on the column name. Create an index on a set to find all the cyclists that have been on a particular team.

```
CREATE INDEX team_idx ON cycling.cyclist_career_teams ( teams );
SELECT * FROM cycling.cyclist_career_teams WHERE teams CONTAINS 'Nederland
 bloeit';
```

```
id                                   | lastname | teams
--------------------------------------+----------+-----------------------------------------------------------------------------
--------------------------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |      VOS | {'Nederland bloeit', 'Rabobank Women Team', 'Rabobank-Liv Giant', 'Rabob
ank-Liv Woman Cycling Team'}
```

- For map collections, create an index on the map key, map value, or map entry. Create an index on a map key to find all cyclist/team combinations for a particular year.

```
CREATE INDEX team_year_idx ON cycling.cyclist_teams ( KEYS (teams) );
SELECT * From cycling.cyclist_teams WHERE teams CONTAINS KEY 2015;
```

```
 id                                   | firstname | lastn
--------------------------------------+-----------+------
-----------------------------------------------------------
-----
 cb07baad-eac8-4f65-b28a-bddc06a0de23 | Elizabeth |
ink - Leontien.nl', 2013: 'Boels-Dolmans Cycling Team', 2
am'}
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |
bank Women Team', 2013: 'Rabobank-Liv Giant', 2014: 'Rabo
am'}
 e7cd5752-bc0d-4157-a80f-7523add8dbcd |      Anna | VAN D
g Team', 2013: 'Sengers Ladies Cycling Team', 2014: 'Rabo
am'}
```

- Create an index on the map entries and find cyclists who are the same age. An index using ENTRIES is only valid for maps.

```
CREATE TABLE cycling.birthday_list (cyclist_name text PRIMARY KEY, blist
 map<text,text>);
CREATE INDEX blist_idx ON cycling.birthday_list (ENTRIES(blist));
SELECT * FROM cycling.birthday_list WHERE blist['age'] = '23';
```

```
 cyclist_name     | blist
------------------+-----------------------------------------------------------------
   Claudio HEINEN | {'age': '23', 'bday': '27/07/1992', 'nation': 'GERMANY'}
 Laurence BOURQUE |  {'age': '23', 'bday': '27/07/1992', 'nation': 'CANADA'}
```

- Using the same index, find cyclists from the same country.

```
SELECT * FROM cyclist.birthday_list WHERE blist['nation'] = 'NETHERLANDS';
```

```
 cyclist_name  | blist
---------------+-----------------------------------------------------------------
 Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
   Toine POELS | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
```

- Create an index on the map values and find cyclists who have a particular value found in the specified map. An index using VALUES is only valid for maps.

```
CREATE TABLE cycling.birthday_list (cyclist_name text PRIMARY KEY, blist
      map<text,text>;
);CREATE INDEX blist_idx ON cycling.birthday_list (VALUES(blist));
SELECT * FROM cycling.birthday_list CONTAINS 'NETHERLANDS';
```

```
lorina@cqlsh:cycling> SELECT * FROM cycling.birthday_list WHERE blist CONTAINS 'NETHERLANDS';

 cyclist_name  | blist
---------------+-----------------------------------------------------------------
 Luc HAGENAARS | {'age': '28', 'bday': '27/07/1987', 'nation': 'NETHERLANDS'}
   Toine POELS | {'age': '52', 'bday': '27/07/1963', 'nation': 'NETHERLANDS'}
```

- Create an index on the full content of a FROZEN map. The table in this example stores the number of Pro wins, Grand Tour races, and Classic races that a cyclist has competed in. The SELECT statement finds any cyclist who has 39 Pro race wins, 7 Grand Tour starts, and 14 Classic starts.

```
CREATE TABLE cycling.race_starts (cyclist_name text PRIMARY KEY, rnumbers
 FROZEN<LIST<int>>);
CREATE INDEX rnumbers_idx ON cycling.race_starts (FULL(rnumbers));
SELECT * FROM cycling.race_starts WHERE rnumbers = [39,7,14];
```

```
 cyclist_name    | rnumbers
-----------------+--------------
 John DEGENKOLB | [39, 7, 14]
```

Indexing with SSTable attached secondary indexes (SASI)

In Cassandra 3.4 and later, SSTable Attached Secondary Indexes (SASI) have been introduced that improve on the existing secondary index implementation with superior performance for queries that previously required the use of `ALLOW FILTERING`. SASI is significantly less resource intensive, using less memory, disk, and CPU. It enables querying with prefix and contains on strings, similar to the SQL implementation of `LIKE = "foo%"` or `LIKE = "%foo%"`, as shown in SELECT. It also supports SPARSE indexing to improve performance of querying large, dense number ranges such as time series data.

SASI takes advantage of Cassandra's write-once immutable ordered data model to build indexes when data is flushed from the memtable to disk. The SASI index data structures are built in memory as the SSTable is written and flushed to disk as sequential writes before the SSTable writing completes. One index file is written for each indexed column.

SASI supports all queries already supported by CQL, and supports the `LIKE` operator using `PREFIX`, `CONTAINS`, and `SPARSE`. If `ALLOW FILTERING` is used, SASI also supports queries with multiple predicates using `AND`. With SASI, the performance pitfalls of using filtering are not realized because the filtering is not performed even if `ALLOW FILTERING` is used.

SASI is implemented using memory mapped B+ trees, an efficient data structure for indexes. B+ trees allow range queries to perform quickly. SASI generates an index for each SSTable. Some key features that arise from this design are:

- SASI can reference offsets in the data file, skipping the Bloom filter and partition indexes to go directory to where data is stored.
- When SSTables are compacted, new indexes are generated automatically.

Currently, SASI does not support collections. Regular secondary indexes can be built for collections. Static columns are supported in Cassandra 3.6 and later.

Using a SSTable Attached Secondary Index (SASI)

In Cassandra 3.4 and later, a new implementation of secondary indexes, SSTable Attached Secondary Indexes (SASI), have greatly improved the performance of secondary indexes and should be used, if possible.

Using CQL, SSTable attached secondary indexes (SASI) can be created on a non-collection column defined in a table. Secondary indexes are used to query a table that uses a column that is not normally queryable, such as a non primary key column. SASI implements three types of indexes, `PREFIX`, `CONTAINS`, and `SPARSE`.

Procedure

PREFIX index

- Create an index `fn_prefix` for the table `cyclist_name` on the column `firstname`. `PREFIX` is the default mode, so it does not need to be specified.

```
CREATE TABLE cycling.cyclist_name (
  id UUID PRIMARY KEY,
  lastname text,
  firstname text
);

CREATE CUSTOM INDEX  fn_prefix ON cyclist_name (firstname) USING
 'org.apache.cassandra.index.sasi.SASIIndex';
```

Figure: SELECT * FROM cycling.cyclist_name;

```
id                                   | firstname | lastname
-------------------------------------+-----------+-----------------
fb372533-eb95-4bb4-8685-6ef61e994caa |   Michael |        MATTHEWS
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |            VOS
220844bf-4860-49d6-9a4b-6b5d3a79cbfb |     Paolo |       TIRALONGO
6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47 |    Steven |      KRUIKSWIJK
e7cd5752-bc0d-4157-a80f-7523add8dbcd |      Anna | VAN DER BREGGEN
```

- Queries can find exact matches for values in `firstname`. Note that indexing is used for this query, as the primary key `id` is not specified.

```
SELECT * FROM cyclist_name WHERE firstname = 'Marianne';
```

```
 id                                   | firstname | lastname
--------------------------------------+-----------+----------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

- Queries can find matches for values in `firstname` 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. Note that indexing is used for this query, as the primary key `id` is not specified.

```
SELECT * FROM cyclist_name WHERE firstname LIKE 'M%';
```

```
 id                                   | firstname | lastname
--------------------------------------+-----------+----------
 fb372533-eb95-4bb4-8685-6ef61e994caa |   Michael | MATTHEWS
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

- Many queries will fail to find matches based on the partial string. All the of the following queries will fail.

```
SELECT * FROM cyclist_name WHERE firstname = 'MARIANNE';
SELECT * FROM cyclist_name WHERE firstname LIKE 'm%';
SELECT * FROM cyclist_name WHERE firstname LIKE '%m%';
SELECT * FROM cyclist_name WHERE firstname LIKE '%m%' ALLOW FILTERING;
SELECT * FROM cyclist_name WHERE firstname LIKE '%M%';
SELECT * FROM cyclist_name WHERE firstname = 'M%';
SELECT * FROM cyclist_name WHERE firstname = '%M';
SELECT * FROM cyclist_name WHERE firstname = '%M%';
SELECT * FROM cyclist_name WHERE firstname = 'm%';
```

The first four queries fail because of case sensitivity. "MARIANNE" is all uppercase, whereas the stored value is not. The next three use a lowercase "m". The placement of the `%` are critical; since the index specifies the `PREFIX` mode, only a trailing `%` will yield results when coupled with `LIKE`. The queries with equalities fail unless the exact match is designated.

CONTAINS index

- Create an index `fn_suffix` for the table `cyclist_name` on the column `firstname`. `CONTAINS` is the specified mode, so that pattern matching for partial patterns given, not just in the prefix.

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

- Queries can find exact matches for values in `firstname`. Note that indexing is used for this query, as the primary key `id` is not specified. For queries on `CONTAINS` indexing, the `ALLOW FILTERING` phrase must be included, although Cassandra will not actually filter.

```
SELECT * FROM cyclist_name WHERE firstname = 'Marianne' ALLOW FILTERING;
```

```
 id                                   | firstname | lastname
--------------------------------------+-----------+----------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

This query returns the same results as a query using `PREFIX` indexing that does an exact match using a slightly modified query.

- Queries can find matches for values in `firstname` 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 can return a matching value. Note that indexing is used for this query, as the primary key `id` is not specified.

```
SELECT * FROM cyclist_name WHERE firstname LIKE '%M%';
```

```
 id                                   | firstname | lastname
--------------------------------------+-----------+----------
 fb372533-eb95-4bb4-8685-6ef61e994caa |   Michael | MATTHEWS
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

  Again, the same results are returned as for the `PREFIX` indexing, using a slightly modified query.
- The `CONTAINS` indexing has a more versatile matching algorithm than `PREFIX`. Look at the examples below to see what results from variations of the last search.

```
SELECT * FROM cyclist_name WHERE firstname LIKE '%arianne';
SELECT * FROM cyclist_name WHERE firstname LIKE '%arian%';
```

```
 id                                   | firstname | lastname
--------------------------------------+-----------+----------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

  Each query matches the pattern, either the final characters of the column value as in `%arianne` or the characters bracketed by `%` such as `%arian%`.
- With `CONTAINS` indexing, even inequality pattern matching is possible. Note again the use of the `ALLOW FILTERING` phrase that required but causes no latency in the query response.

```
SELECT * FROM cyclist_name WHERE firstname > 'Mar' ALLOW FILTERING;
```

```
 id                                   | firstname | lastname
--------------------------------------+-----------+----------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

  The only row matching the conditions returns the same value as the last query.
- Like with `PREFIX` indexing, many queries will fail to find matches based on the partial string. All the of the following queries will fail.

```
SELECT * FROM cyclist_name WHERE firstname = 'Marianne';
SELECT * FROM cyclist_name WHERE firstname = 'MariAnne' ALLOW FILTERING;
SELECT * FROM cyclist_name WHERE firstname LIKE '%m%';
SELECT * FROM cyclist_name WHERE firstname LIKE 'M%';
SELECT * FROM cyclist_name WHERE firstname LIKE '%M';
SELECT * FROM cyclist_name WHERE firstname LIKE 'm%';
```

  The first query fails due to the absence of the `ALLOW FILTERING` phrase. The next two queries fail because of case sensitivity. "MariAnne" has one uppercase letter, whereas the stored value does not. The last three fail due to placement of the `%`.
- Either the `PREFIX` index or the `CONTAINS` index can be created with case sensitivity by adding an analyzer class and case_sensitive option.

```
CREATE CUSTOM INDEX fn_suffix_allcase ON cyclist_name (firstname) USING
 'org.apache.cassandra.index.sasi.SASIIndex'
```

```
WITH OPTIONS = {
'mode': 'CONTAINS',
'analyzer_class':
 'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
'case_sensitive': 'false'
};
```

The `analyzer_class` used here is the non-tokenizing analyzer that does not perform analysis on the text in the specified column. The option `case_sensitive` is set to `false` to make the indexing case insensitive.

- With the addition of the analyzer class and option, the following query now also works, using a lowercase "m".

```
SELECT * FROM cyclist_name WHERE firstname LIKE '%m%';
```

```
id                                   | firstname | lastname
--------------------------------------+-----------+----------
fb372533-eb95-4bb4-8685-6ef61e994caa |   Michael | MATTHEWS
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS
```

- If queries are narrowed with an indexed column value, non-indexed columns can be specified. Compound queries can also be created with multiple indexed columns. This example alters the table to add a column `age` that is not indexed before performing the query.

```
ALTER TABLE cyclist_name ADD age int;
UPDATE cyclist_name SET age=23 WHERE id=5b6962dd-3f90-4c93-8f61-
eabfa4a803e2;
INSERT INTO cyclist_name (id,age,firstname,lastname) VALUES
 (8566eb59-07df-43b1-a21b-666a3c08c08a,18,'Marianne','DAAE');
SELECT * FROM cyclist_name WHERE firstname='Marianne' and age > 20 allow
 filtering;
```

```
id                                   | age | firstname | lastname
--------------------------------------+-----+-----------+----------
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  23 |  Marianne |      VOS
```

SPARSE index

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

```
CREATE CUSTOM INDEX fn_contains ON cyclist_name (age) USING
 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = { 'mode': 'SPARSE' };
```

Use SPARSE indexing for data that is sparse (every term/column value has less than 5 matching keys). Indexing the `created_at` field in time series data (where there is typically few matching rows/events per `created_at` timestamp) is a good use case. SPARSE indexing is primarily an optimization for range queries, especially large ranges that span large timespans.

- To illustrate the use of the SPARSE index, create a table and insert some time series data:

```
CREATE TABLE cycling.comments (commenter text, created_at timestamp,
 comment text, PRIMARY KEY (commenter));
INSERT INTO cycling.comments (commenter, comment, created_at) VALUES
 ('John', 'Fantastic race!', '2013-01-01 00:05:01.500');
INSERT INTO cycling.comments (commenter, comment, created_at) VALUES
 ('Jane', 'What a finish', '2013-01-01 00:05:01.400');
INSERT INTO cycling.comments (commenter, comment, created_at) VALUES
 ('Mary', 'Hated to see the race end.', '2013-01-01 00:05:01.300');
INSERT INTO cycling.comments (commenter, comment, created_at) VALUES
 ('Jonnie', 'Thankfully, it is over.', '2013-01-01 00:05:01.600');
```

```
 commenter | comment                      | created_at
-----------+------------------------------+---------------------------------
      John |            Fantastic race! | 2013-01-01 08:05:01.500000+0000
      Mary | Hated to see the race end. | 2013-01-01 08:05:01.300000+0000
      Jane |              What a finish | 2013-01-01 08:05:01.400000+0000
    Jonnie |    Thankfully, it is over. | 2013-01-01 08:05:01.600000+0000
```

- Find all the comments made before the timestamp `2013-01-01 00:05:01.500`.

```
SELECT * FROM cycling.comments WHERE created_at < '2013-01-01
 00:05:01.500';
```

```
 commenter | comment                      | created_at
-----------+------------------------------+---------------------------------
      Mary | Hated to see the race end. | 2013-01-01 08:05:01.300000+0000
      Jane |              What a finish | 2013-01-01 08:05:01.400000+0000
```

This query returns all the results where `created_at` is found to be less than the timestamp supplied. The inequalities >=, > and <= are all valid operators.

SPARSE indexing is used only for numeric data, so `LIKE` queries do not apply.

Using analyzers

- Analyzers can be specified that will analyze the text in the specified column. The `NonTokenizingAnalyzer` is used for cases where the text is not analyzed, but case normalization or sensitivity is required. The `StandardAnalyzer` is used for analysis that involves stemming, case normalization, case sensitivity, skipping common words like "and" and "the", and localization of the language used to complete the analysis. Altering the table again to add a lengthier text column provides a window into the analysis.

```
ALTER TABLE cyclist_name ADD comments text;
UPDATE cyclist_name SET comments ='Rides hard, gets along with others, a
 real winner' WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa;
UPDATE cyclist_name SET comments ='Rides fast, does not get along with
 others, a real dude' WHERE id = 5b6962dd-3f90-4c93-8f61-eabfa4a803e2;

CREATE CUSTOM INDEX stdanalyzer_idx ON cyclist_name (comments) USING
 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
'mode': 'CONTAINS',
'analyzer_class':
 'org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer',
'analyzed': 'true',
'tokenization_skip_stop_words': 'and, the, or',
'tokenization_enable_stemming': 'true',
'tokenization_normalize_lowercase': 'true',
'tokenization_locale': 'en'
```

```
};
```

- This query will search for the presence of a designated string, using the analyzed text to return a result.

```
SELECT * FROM cyclist_name WHERE comments LIKE 'ride';
```

```
id                                   | age | comments                                                  |
firstname | lastname
--------------------------------------+-----+-----------------------------------------------------------+
----------+----------
fb372533-eb95-4bb4-8685-6ef61e994caa |  18 |       Rides hard, gets along with others, a real winner |
  Michael | MATTHEWS
5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  23 | Rides fast, does not get along with others, a real dude |
 Marianne |      VOS
```

This query returns all the results where `ride` is found either as an exact word or as a stem for another word - `rides` in this case.

Building and maintaining indexes

An advantage of indexes is the operational ease of populating and maintaining the index. Indexes are built in the background automatically, without blocking reads or writes. Client-maintained *tables as indexes* must be created manually; for example, if the artists column had been indexed by creating a table such as songs_by_artist, your client application would have to populate the table with data from the songs table.

To perform a hot rebuild of an index, use the `nodetool rebuild_index` command.

Altering a table

Tables can be changed with the `ALTER` command.

Altering columns in a table

The `ALTER TABLE` command can be used to add new columns to a table and to alter the column type of an existing column.

Procedure

- Add a age column of type int to the table cycling.cyclist_alt_stats.

```
cqlsh> ALTER TABLE cycling.cyclist_alt_stats ADD age int;
```

This creates the column metadata and adds the column to the table schema, and sets the value to NULL for all rows.

```
id                                   | age  | birthday                | height | lastname  | nationality | weight
-------------------------------------+------+-------------------------+--------+-----------+-------------+--------
e0953617-07eb-4c82-8f91-3b2757981625 | null | 1982-01-29 00:00:00-0800 | 1.78 m |     BRUTT |      Russia |  68 kg
a9e96714-2dd0-41f9-8bd0-557196a44ecf | null | 1986-04-21 00:00:00-0800 | 1.88 m |  ISAYCHEV |      Russia |  80 kg
ed584e99-80f7-4b13-9a90-9dc5571e6821 | null | 1989-07-05 00:00:00-0700 | 1.69 m | TSATEVICH |      Russia |  64 kg
078654a6-42fa-4142-ae43-cebdc67bd902 | null | 1981-01-14 00:00:00-0800 | 1.82 m |   LAGUTIN |      Russia |  63 kg
1ba0417d-62da-4103-b710-de6fb227db6f | null | 1990-05-27 00:00:00-0700 |   null |  PAULINHO |    Portugal |   null
d74d6e70-7484-4df5-8551-f5090c37f617 | null | 1991-08-25 00:00:00-0700 | 1.75 m |     GRMAY |    Ethiopia |  63 kg
c09e9451-50da-483d-8108-e6bea2e827b3 | null | 1981-03-29 00:00:00-0800 | 1.78 m | VEIKKANEN |     Finland |  66 kg
823ec386-2a46-45c9-be41-2425a4b7658e | null | 1985-01-09 00:00:00-0800 | 1.84 m |    BELKOV |      Russia |  71 kg
f1deff54-7d96-4981-b14a-b70be4da82d2 | null | 1987-03-07 00:00:00-0800 |   null | TLEUBAYEV |  Kazakhstan |   null
4ceb495c-55ab-4f71-83b9-81117252bb13 | null | 1990-05-27 00:00:00-0700 |   null |     DUVAL |      France |   null
```

- Add a column favorite_color of varchar, and then change the data type of the same column to text.

```
cqlsh> ALTER TABLE cycling.cyclist_alt_stats ADD favorite_color varchar;
```

```
ALTER TABLE cycling.cyclist_alt_stats ALTER favorite_color TYPE text;
```

Note: There are limitations on altering the data type of a column. The two data types, the original and the one changing to, must be compatible.

Altering a table to add a collection

The ALTER TABLE command can be used to add new collection columns to a table and to alter the column type of an existing column.

Procedure

- Alter the table cycling.upcoming_calendar to add a collection map description that can store a description for each race listed.

```
cqlsh> ALTER TABLE cycling.upcoming_calendar ADD description
 map<text,text>;
```

- After updating cycling.upcoming_calendar table to insert some data, description can be displayed.

```
cqlsh> UPDATE cycling.upcoming_calendar
SET description = description + {'Criterium du Dauphine' : "Easy race',
 'Tour du Suisse' : 'Hard uphill race'} WHERE year = 2015 AND month = 6;
```

```
year | month | description                                                                                | events
------+-------+--------------------------------------------------------------------------------------------+----------------------------
----------------------------------------------------
2015 |     6 | {'Criterium du Dauphine': 'Easy race', 'Tour de Suisse': 'Hard uphill race'} |
           ['Criterium du Dauphine', 'Tour de Suisse']
2015 |     7 |                                                                                       null |
                                    ['Tour de France']
2015 |     8 |                                                                                       null | ['Clasica Ciclista San Sebas
tian', 'Tour de Pologne', 'Eneco Tour', 'Vuelta a Espana']
```

Altering the data type of a column

Using ALTER TABLE, you can change the data type of a column after it is defined or added to a table.

Procedure

Change the `favorite_color` column to store as `text` instead of `varchar` by changing the data type of the column.

```
ALTER TABLE cycling.cyclist_alt_stats ADD favorite_color varchar;
ALTER TABLE cycling.cyclist_alt_stats ALTER favorite_color TYPE text;
```

Only newly inserted values will be created with the new type. However, the data type before must be compatible with the new data type specified.

Altering the table properties

Using ALTER TABLE, you can change the table properties of a table.

Procedure

Alter a table to change the caching properties.

```
cqlsh> ALTER TABLE cycling.race_winners WITH caching = {'keys' : 'NONE',
 'rows_per_partition' : '15' };
```

Altering a materialized view

In Cassandra 3.0 and later, a materialized view has table properties like its source tables. Use the `ALTER MATERIALIZED VIEW` command alter the view's properties. Specify updated properties and values in a `WITH` clause. Materialized views do not perform repair, so properties regarding repair are invalid.

Procedure

- Alter a materialized view to change the caching properties.

```
cqlsh> ALTER MATERIALIZED VIEW cycling.cyclist_by_birthday WITH caching =
 {'keys' : 'NONE', 'rows_per_partition' : '15' };
```

Related information

Creating a materialized view on page 21

CREATE MATERIALIZED VIEW on page 153

ALTER MATERIALIZED VIEW on page 131

DROP MATERIALIZED VIEW on page 171

Altering a user-defined type

The `ALTER TYPE` command can be used to add new columns to a user-defined type and to alter the data type of an existing column in a user-defined type.

Procedure

- Add a middlename column of type text to the user-defined type cycling.fullname.

```
cqlsh> ALTER TYPE cycling.fullname ADD middlename text;
```

This creates the column metadata and adds the column to the type schema. To verify, display the table ssystem.schema_usertypes.

```
 keyspace_name | type_name  | field_names                                                    | field_types
---------------+------------+----------------------------------------------------------------+-----------------------------
-------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------
       cycling | basic_info | ['birthday', 'lastname', 'nationality', 'weight', 'height'] | ['org.apache.cassandra.db.marsh
al.TimestampType', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cass
andra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']
       cycling |   fullname |                    ['firstname', 'lastname', 'middlename'] |
                                                                  ['org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cass
andra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']
```

- A column can be renamed in either ALTER TABLE or ALTER TYPE. In ALTER TABLE, only primary key columns may be renamed.

```
cqlsh> ALTER TYPE cycling.fullname RENAME middlename TO middleinitial;
```

```
 keyspace_name | type_name  | field_names                                                                | field_types
---------------+------------+----------------------------------------------------------------------------+----------------------------
---------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------
       cycling | basic_info | ['birthday', 'lastname', 'nationality', 'weight', 'height'] | ['org.apache.cassandra.db.marsh
al.TimestampType', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cass
andra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']
       cycling |   fullname |                 ['firstname', 'lastname', 'middleinitial'] |
                                                                                  ['org.apache.cassandra.db.marshal.UTF8Type', 'org.apache.cass
andra.db.marshal.UTF8Type', 'org.apache.cassandra.db.marshal.UTF8Type']
```

Removing a keyspace, schema, or data

To remove data, you can set column values for automatic removal using the TTL (time-to-expire) table attribute. You can also drop a table or keyspace, and delete keyspace column metadata.

Dropping a keyspace, table or materialized view

You drop a table or keyspace using the DROP command.

Procedure

- Drop the test keyspace.

```
cqlsh> DROP KEYSPACE test;
```

- Drop the cycling.last_3_days table.

```
cqlsh> DROP TABLE cycling.last_3_days;
```

- Drop the cycling.cyclist_by_age materialized view in Cassandra 3.0 and later.

```
cqlsh> DROP MATERIALIZED VIEW cycling.cyclist_by_age;
```

Deleting columns and rows

CQL provides the DELETE command to delete a column or row. Deleted values are removed completely by the first compaction following deletion.

Procedure

1. Deletes the values of the column lastname from the table cyclist_name.

```
cqlsh> DELETE lastname FROM cycling.cyclist_name WHERE id = c7fceba0-
c141-4207-9494-a29f9809de6f;
```

2. Delete entire row for a particular race from the table calendar.

```
cqlsh> DELETE FROM cycling.calendar WHERE race_id = 200;
```

Dropping a user-defined function (UDF)

You drop a user-defined function (UDF) using the DROP command.

Procedure

Drop the fLog() function. The conditional option `IF EXISTS` can be included.

```
cqlsh> DROP FUNCTION [IF EXISTS] fLog;
```

Securing a table

Data in CQL tables is secured using either user or role-based security commands.

cassandra.yaml settings must be changed in order to use authentication and authorization. Change the following settings:

```
authenicator: PasswordAuthenticator
authorizer: CassandraAuthorizer
```

Database users

User-based access control enables authorization management on a per-user basis.

Notice: Creating users is supported for backwards compatibility. Authentication and authorization for Cassandra 2.2 and later are based on roles, and role-based commands should be used.

Procedure

- Create a user with a password. `IF NOT EXISTS` is included to ensure a previous user definition is not overwritten.

  ```
  cqlsh> CREATE USER IF NOT EXISTS sandy WITH PASSWORD 'Ride2Win@'
   NOSUPERUSER;
  ```

- Create a user with SUPERUSER privileges. SUPERUSER grants the ability to create users and roles unconditionally.

  ```
  cqlsh> CREATE USER chuck WITH PASSWORD 'Always1st$' SUPERUSER;
  ```

  Note: `WITH PASSWORD` implicitly specifies `LOGIN`.

- Alter a user to change options. A role with SUPERUSER status can alter the SUPERUSER status of another user, but not the user currently held. To modify properties of a user, the user must have permission.

  ```
  cqlsh> ALTER USER sandy SUPERUSER;
  ```

- List the users.

  ```
  cqlsh> LIST USERS;
  ```

```
 name      | super
-----------+-------
 cassandra |  True
     chuck |  True
     sandy | False
  sysadmin |  True
```

- Drop user that is not a current user. User must be a SUPERUSER.

```
DROP USER IF EXISTS chuck;
```

Database roles

Roles-based access control is available in Cassandra 2.2 and later. Roles enable authorization management on a larger scale than security per user can provide. A role is created and may be granted to other roles. Hierarchical sets of permissions can be created. For more information, see Role Based Access Control in Cassandra.

Procedure

- Create a role with a password. `IF NOT EXISTS` is included to ensure a previous role definition is not overwritten.

```
cqlsh> CREATE ROLE IF NOT EXISTS team_manager WITH PASSWORD =
 'RockIt4Us!';
```

- Create a role with `LOGIN` and `SUPERUSER` privileges. `LOGIN` allows a client to identify as this role when connecting. `SUPERUSER` grants the ability to create roles unconditionally if the role has `CREATE` permissions.

```
cqlsh> CREATE ROLE sys_admin WITH PASSWORD = 'IcanDoIt4ll' AND LOGIN =
 true AND SUPERUSER = true;
```

- Alter a role to change options. A role with SUPERUSER status can alter the SUPERUSER status of another role, but not the role currently held. PASSWORD, LOGIN, and SUPERUSER can be modified with ALTER ROLE. To modify properties of a role, the user must have ALTER permission.

```
cqlsh> ALTER ROLE sys_admin WITH PASSWORD = 'All4one1forAll' AND SUPERUSER
 = false;
```

- Grant a role to a user or a role. To execute GRANT and REVOKE statements requires AUTHORIZE permission on the role being granted/revoked.

```
cqlsh> GRANT sys_admin TO team_manager;
GRANT team_manager TO sandy;
```

- List roles of a user.

```
cqlsh> LIST ROLES;
LIST ROLES OF sandy;
```

Note: NORECURSIVE is an option to discover all roles directly granted to a user. Without NORECURSIVE, transitively acquired roles are also listed.

```
 role          | super | login | options
---------------+-------+-------+---------
    cassandra |  True |  True |      {}
     sysadmin |  True |  True |      {}
 team_manager |  True | False |      {}
```

- Revoke role that was previously granted to a user or a role. Any permission that derives from the role is revoked.

```
cqlsh> REVOKE sys_admin FROM team_manager;
REVOKE team_manager FROM sandy;
```

- Drop role that is not a current role. User must be a SUPERUSER.

```
DROP ROLE IF EXISTS sys_admin;
```

Database Permissions

Authentication and authorization should be set based on roles, rather than users. Authentication and authorization for Cassandra 2.2 and later are based on roles, and user commands are included only for legacy backwards compatibility.

Roles may be granted to other roles to create hierarchical permissions structures; in these hierarchies, permissions and SUPERUSER status are inherited, but the LOGIN privilege is not.

Permissions can be granted at any level of the database hierarchy and flow downwards. Keyspaces and tables are hierarchical as follows: ALL KEYSPACES > KEYSPACE > TABLE. Functions are hierarchical in the following manner: ALL FUNCTIONS > KEYSPACE > FUNCTION. ROLES can also be hierarchical and encompass other ROLES. Permissions can be granted on:

- CREATE - keyspace, table, function, role, index
- ALTER - keyspace, table, function, role

- DROP - keyspace, table, function, role, index
- SELECT - keyspace, table
- MODIFY - INSERT, UPDATE, DELETE, TRUNCATE - keyspace, table
- AUTHORIZE - GRANT PERMISSION, REVOKE PERMISSION - keyspace, table, function, role
- DESCRIBE - LIST ROLES
- EXECUTE - SELECT, INSERT, UPDATE - functions

Note: Index must additionally have ALTER permission on the base table in order to CREATE or DROP.

The permissions are extensive with many variations. A few examples are described below.

Procedure

- The first line grants anyone with the team_manager role the ability to `INSERT, UPDATE, DELETE, and TRUNCATE` any table in the keyspace cycling. The second line grants anyone with the sys_admin role the ability to view all roles in the database.

```
GRANT MODIFY ON KEYSPACE cycling TO team_manager;
GRANT DESCRIBE ON ALL ROLES TO sys_admin;
```

- The first line revokes `SELECT` in all keyspaces for anyone with the team_manager role. The second line prevents the team_manager role from executing the named function `fLog()`.

```
REVOKE SELECT ON ALL KEYSPACES FROM team_manager;
REVOKE EXECUTE ON FUNCTION cycling.fLog(double) FROM team_manager;
```

- All permissions can be listed, for either all keyspaces or a single keyspace.

```
LIST ALL PERMISSIONS OF sandy;
LIST ALL PERMISSIONS ON cycling.cyclist_name OF chuck;
```

```
role      | username | resource                | permission
----------+----------+-------------------------+------------
sysadmin | sysadmin | <keyspace cycling> |     MODIFY
sysadmin | sysadmin | <keyspace cycling> |  AUTHORIZE
```

- Grant permission to drop all functions, including aggregate in the current keyspace.

```
GRANT DROP ON ALL FUNCTIONS IN KEYSPACE TO coach;
```

What to do next

See the Apache Cassandra CQL document for more details.

Tracing consistency changes

In a distributed system such as Cassandra, the most recent value of data is not necessarily on every node all the time. The client application configures the consistency level per request to manage response time versus data accuracy. By tracing activity on a five-node cluster, this tutorial shows the difference between these consistency levels and the number of replicas that participate to satisfy a request:

- ONE

  Returns data from the nearest replica.
- QUORUM

Returns the most recent data from the majority of replicas.

- ALL

  Returns the most recent data from all replicas.

Follow instructions to setup five nodes on your local computer, trace reads at different consistency levels, and then compare the results.

Setup to trace consistency changes

To setup five nodes on your local computer, trace reads at different consistency levels, and then compare the results. This example uses ccm, a tool for running multiple nodes of Cassandra on a local computer.

Procedure

1. Get the ccm library of scripts from github.

   You will use this library in subsequent steps to perform the following actions:

   - Download Apache Cassandra source code.
   - Create and launch an Apache Cassandra cluster on a single computer.

   Refer to the ccm README for prerequisites.
2. Optional: For Mac computers, set up loopback aliases. All other platforms, skip this step.

   ```
   $ sudo ifconfig lo0 alias 127.0.0.2 up
   $ sudo ifconfig lo0 alias 127.0.0.3 up
   $ sudo ifconfig lo0 alias 127.0.0.4 up
   $ sudo ifconfig lo0 alias 127.0.0.5 up
   ```
3. Download Apache Cassandra source code into the /.ccm/repository.
4. Start the ccm cluster named `trace_consistency` using Cassandra version 3.0.5. The source code to run the cluster will automatically download and compile.

   ```
   $ ccm create trace_consistency -v 3.0.5
   ```

   ```
   Current cluster is now: trace_consistency
   ```
5. Use the following commands to populate and check the cluster:

   ```
   $ ccm populate -n 5
   $ ccm start
   ```
6. Check that the cluster is up:

   ```
   $ ccm node1 ring
   ```

   The output shows the status of all five nodes.
7. Connect cqlsh to the first node in the ring.

   ```
   $ ccm node1 cqlsh
   ```

Related information

Cassandra 2.0 cassandra.yaml
Cassandra 2.1 cassandra.yaml
Cassandra 2.2 cassandra.yaml
Cassandra 3.0 cassandra.yaml

Trace reads at different consistency levels

After performing the setup steps, run and trace queries that read data at different consistency levels. The tracing output shows that using three replicas on a five-node cluster, a consistency level of ONE processes responses from one of three replicas, QUORUM from two of three replicas, and ALL from three of three replicas.

Procedure

1. On the cqlsh command line, create a keyspace that specifies using three replicas for data distribution in the cluster.

```
cqlsh> CREATE KEYSPACE cycling_alt WITH replication =
 {'class':'SimpleStrategy', 'replication_factor':3};
```

2. Create a table, and insert some values:

```
cqlsh> USE cycling_alt;
cqlsh> CREATE TABLE cycling_alt.tester ( id int PRIMARY KEY, col1 int,
 col2 int );
cqlsh> INSERT INTO cycling_alt.tester (id, col1, col2) VALUES (0, 0, 0);
```

3. Turn on tracing and use the CONSISTENCY command to check that the consistency level is ONE, the default.

```
cqlsh> TRACING on;
cqlsh> CONSISTENCY;
```

The output should be:

```
Current consistency level is 1.
```

4. Query the table to read the value of the primary key.

```
cqlsh> SELECT * FROM cycling_alt.tester WHERE id = 0;
```

The output includes tracing information:

```
  id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)

Tracing session: 65bd3150-0109-11e6-8b46-15359862861c

 activity
    | timestamp                  | source    | source_elapsed
-------------------------------------------------------------------------------
+----------------------------+-----------+----------------
                                                      Execute CQL3
 query | 2016-04-12 16:50:55.461000 | 127.0.0.1 |              0
 Parsing SELECT * FROM cycling_alt.tester WHERE id = 0; [SharedPool-
Worker-1] | 2016-04-12 16:50:55.462000 | 127.0.0.1 |            276
                                    Preparing statement [SharedPool-
Worker-1] | 2016-04-12 16:50:55.462000 | 127.0.0.1 |            509
              Executing single-partition query on tester [SharedPool-
Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 |           1019
```

```
                              Acquiring sstable references [SharedPool-
Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 |           1106
                                Merging memtable contents [SharedPool-
Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 |           1159
                          Read 1 live and 0 tombstone cells [SharedPool-
Worker-2] | 2016-04-12 16:50:55.463000 | 127.0.0.1 |           1372
                                                          Request
 complete | 2016-04-12 16:50:55.462714 | 127.0.0.1 |           1714
```

The tracing results list all the actions taken to complete the SELECT statement.

5. Change the consistency level to QUORUM to trace what happens during a read with a QUORUM consistency level.

```
cqlsh> CONSISTENCY quorum;
cqlsh> SELECT * FROM cycling_alt.tester WHERE id = 0;
```

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)

Tracing session: 5e3601f0-0109-11e6-8b46-15359862861c

 activity
    | timestamp                  | source    | source_elapsed
-------------------------------------------------------------------------------
+----------------------------+-----------+----------------
                                                      Execute CQL3
 query | 2016-04-12 16:50:42.831000 | 127.0.0.1 |              0
 Parsing SELECT * FROM cycling_alt.tester WHERE id = 0; [SharedPool-
Worker-1] | 2016-04-12 16:50:42.831000 | 127.0.0.1 |            259
                                  Preparing statement [SharedPool-
Worker-1] | 2016-04-12 16:50:42.831000 | 127.0.0.1 |            557
             Executing single-partition query on tester [SharedPool-
Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |           1076
                              Acquiring sstable references [SharedPool-
Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |           1182
                                Merging memtable contents [SharedPool-
Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |           1268
                          Read 1 live and 0 tombstone cells [SharedPool-
Worker-3] | 2016-04-12 16:50:42.832000 | 127.0.0.1 |           1632
                                                          Request
 complete | 2016-04-12 16:50:42.832887 | 127.0.0.1 |           1887
```

6. Change the consistency level to ALL and run the SELECT statement again.

```
cqlsh> CONSISTENCY ALL;
cqlsh> SELECT * FROM cycling_alt.tester WHERE id = 0;
```

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)

Tracing session: 6c4678b0-0109-11e6-8b46-15359862861c

 activity
    | timestamp                  | source    | source_elapsed
```

```
-------------------------------------------------------------------------------
+----------------------------+-----------+----------------
                                                    Execute CQL3
 query | 2016-04-12 16:51:06.427000 | 127.0.0.1 |              0
 Parsing SELECT * FROM cycling_alt.tester WHERE id = 0; [SharedPool-
Worker-1] | 2016-04-12 16:51:06.427000 | 127.0.0.1 |            324
                                  Preparing statement [SharedPool-
Worker-1] | 2016-04-12 16:51:06.427000 | 127.0.0.1 |            524
                                 Read-repair DC_LOCAL [SharedPool-
Worker-1] | 2016-04-12 16:51:06.428000 | 127.0.0.1 |           1016
             Executing single-partition query on tester [SharedPool-
Worker-3] | 2016-04-12 16:51:06.428000 | 127.0.0.1 |           1793
                           Acquiring sstable references [SharedPool-
Worker-3] | 2016-04-12 16:51:06.429000 | 127.0.0.1 |           1886
                            Merging memtable contents [SharedPool-
Worker-3] | 2016-04-12 16:51:06.429000 | 127.0.0.1 |           1951
                      Read 1 live and 0 tombstone cells [SharedPool-
Worker-3] | 2016-04-12 16:51:06.429000 | 127.0.0.1 |           2176
                                                   Request
 complete | 2016-04-12 16:51:06.429391 | 127.0.0.1 |           2391
```

How consistency affects performance

Changing the consistency level can affect read performance. The tracing output shows that as you change the consistency level from ONE to QUORUM to ALL, performance degrades in from 1714 to 1887 to 2391 microseconds, respectively. If you follow the steps in this tutorial, it is not guaranteed that you will see the same trend because querying a one-row table is a degenerate case, used for example purposes. The difference between QUORUM and ALL is slight in this case, so depending on conditions in the cluster, performance using ALL might be faster than QUORUM.

Under the following conditions, performance using ALL is worse than QUORUM:

- The data consists of thousands of rows or more.
- One node is slower than others.
- A particularly slow node was not selected to be part of the quorum.

Tracing queries on large datasets

You can use probabilistic tracing on databases having at least ten rows, but this capability is intended for tracing through much more data. After configuring probabilistic tracing using the `nodetool settraceprobability` command, you query the system_traces keyspace.

```
SELECT * FROM system_traces.events;
```

Displaying rows from an unordered partitioner with the TOKEN function

The ByteOrdered partitioner arranges tokens the same way as key values, but the RandomPartitioner and Murmur3Partitioner distribute tokens in a completely unordered manner. When using the RandomPartitioner or Murmur3Partitioner, Cassandra rows are ordered by the hash of their partition key, or for one partition queries, rows are ordered by their clustering key. Hence, the order of rows is not meaningful, because of the hashes generated.

To order the rows for display when using RandomPartitioner or Murmur3Partitioner, the token function may be used. However, ordering with the `TOKEN` function does not always provide the expected results. Use the `TOKEN` function to express a conditional relation on a partition key column. In this case, the query returns rows based on the token of the partition key rather than on the value.

The `TOKEN` function can also be used to select a range of partitions for a ByteOrderedPartitioner. Using the `TOKEN` function with ByteOrderedPartitioner will generally yield expected results.

The type of the arguments to the `TOKEN` function depends on the type of the columns used as the argument of the function. The return type depends on the partitioner in use:

- Murmur3Partitioner, bigint
- RandomPartitioner, varint
- ByteOrderedPartitioner, blob

Procedure

- Select data based on a range of tokens of a particular column value.

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) > TOKEN('2015-05-24');
```

```
year                     | rank | cyclist_name      | race_name
-------------------------+------+-------------------+-----------------------------------------------------------
2015-05-26 00:00:00-0700 |    1 |       Mikel Landa |                                Giro d'Italia Stage 16
2015-05-26 00:00:00-0700 |    2 | Steven Kruijswijk |                                Giro d'Italia Stage 16
2015-05-26 00:00:00-0700 |    3 |  Alberto Contador |                                Giro d'Italia Stage 16
2015-05-25 00:00:00-0700 |    1 |    Matthew Busche | National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700 |    2 |    Joe Dombrowski | National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700 |    3 |      Kiel Reijnen | National Championships United States - Road Race (NC)
```

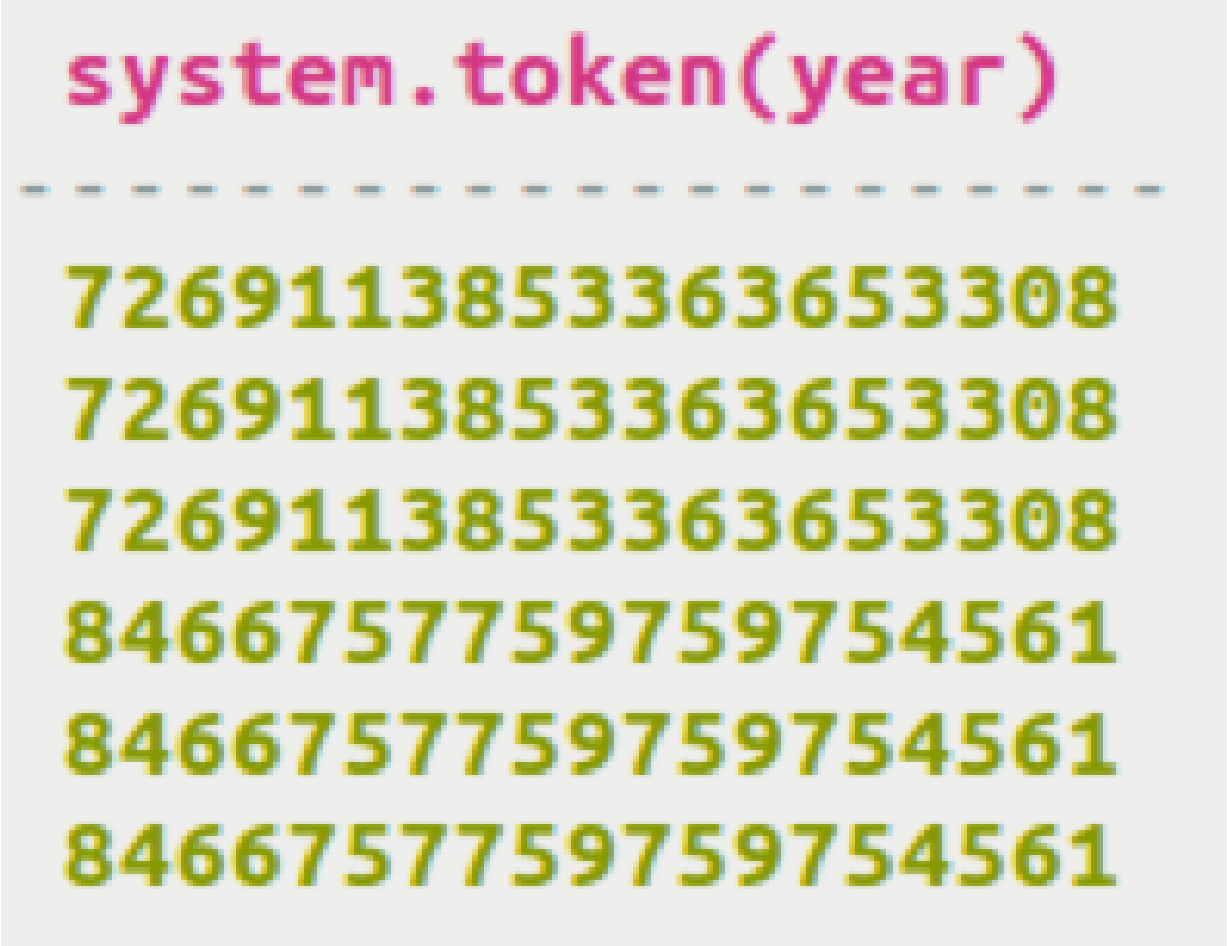
- The results will not always be consistent with expectations, because the token function actually queries directly using tokens. Underneath, the token function uses token-based comparisons and does not convert year to token (not year > '2015-05-26').

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) > TOKEN('2015-05-26');
```

```
year                     | rank | cyclist_name   | race_name
-------------------------+------+----------------+-----------------------------------------------------------
2015-05-25 00:00:00-0700 |    1 | Matthew Busche | National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700 |    2 | Joe Dombrowski | National Championships United States - Road Race (NC)
2015-05-25 00:00:00-0700 |    3 |   Kiel Reijnen | National Championships United States - Road Race (NC)
```

- Display the tokens for all values of the column year.

```
SELECT TOKEN(year) FROM cycling.last_3_days;
```



- Tokens and partition keys can be mixed in conditional statements. The results will not always be straightforward, but they are not unexpected if you understand what the `TOKEN` function does.

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) < TOKEN('2015-05-26')
 AND year IN ('2015-05-24','2015-05-25');
```

```
 year | rank | cyclist_name | race_name
------+------+--------------+-----------
```

Determining time-to-live (TTL) for a column

To set the TTL for data, use the `USING TTL` keywords. The `TTL` function may be used to retrieve the `TTL` information.

The `USING TTL` keywords can be used to insert data into a table for a specific duration of time. To determine the current time-to-live for a record, use the `TTL` function.

Procedure

- Insert data into the table cycling.calendar and use the `USING TTL` clause to set the expiration period to 86400 seconds.

```
INSERT INTO cycling.calendar (race_id, race_name, race_start_date,
 race_end_date) VALUES (200, 'placeholder','2015-05-27', '2015-05-27')
 USING TTL 86400;
```

- Issue a SELECT statement to determine how much longer the data has to live.

```
SELECT TTL (race_name) from cycling.calendar WHERE race_id = 200;
```

```
 ttl(race_name)
----------------
          86371
```

  If you repeat this step after some time, the time-to-live value will decrease.
- The time-to-live value can also be updated with the USING TTL keywords in an UPDATE command.

```
UPDATE cycling.calendar USING TTL 300 SET race_name = 'dummy' WHERE
 race_id = 200 AND race_start_date = '2015-05-27' AND race_end_date =
 '2015-05-27';
```

```
 ttl(race_name)
----------------
             295
```

Determining the date/time of a write

A table contains a timestamp representing the date/time that a write occurred to a column. Using the WRITETIME function in a SELECT statement returns the date/time that the column was written to the database. The output of the function is microseconds except in the case of Cassandra 2.1 counter columns. Counter column writetime is milliseconds. This procedure continues the example from the previous procedure and calls the WRITETIME function to retrieve the date/time of the writes to the columns.

Procedure

- Retrieve the date/time that the value Paolo was written to the firstname column in the table cyclist_points. Use the WRITETIME function in a SELECT statement, followed by the name of a column in parentheses:

```
SELECT WRITETIME (firstname) FROM cycling.cyclist_points WHERE
 id=220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
```

```
 writetime(firstname)
----------------------
     1435108325093225
```

Note: The writetime output in microseconds converts to Wed, 24 Jun 2015 01:12:05 GMT.

Legacy tables

Legacy tables must be handled differently from currently built CQL tables.

Working with legacy applications

Internally, CQL does not change the row and column mapping from the Thrift API mapping. CQL and Thrift use the same storage engine. CQL supports the same query-driven, denormalized data modeling principles as Thrift. Existing applications do not have to be upgraded to CQL. The CQL abstraction layer makes CQL easier to use for new applications. For an in-depth comparison of Thrift and CQL, see "A Thrift to CQL Upgrade Guide" and CQL for Cassandra experts.

Creating a legacy table

You can create legacy (Thrift/CLI-compatible) tables in CQL using the COMPACT STORAGE directive. The compact storage directive used with the CREATE TABLE command provides backward compatibility with older Cassandra applications; new applications should generally avoid it.

Compact storage stores an entire row in a single column on disk instead of storing each non-primary key column in a column that corresponds to one column on disk. Using compact storage prevents you from adding new columns that are not part of the PRIMARY KEY.

Querying a legacy table

Using CQL, you can query a legacy table. A legacy table managed in CQL includes an implicit WITH COMPACT STORAGE directive.

Using a music service example, select all the columns in the playlists table that was created in CQL. This output appears:

```
[default@music ] GET playlists  [62c36092-82a1-3a00-93d1-46196ee77204 ];
                 =>  ( column =7db1a490-5878-11e2-bcfd-0800200c9a66:,value =,
 timestamp =1357602286168000 )
                 =>  ( column =7db1a490-5878-11e2-bcfd-0800200c9a66:album,
                   value =4e6f204f6e65205269646573206666f722046726565,
 timestamp =1357602286168000 )
                 .
                 .
                 .
                 =>  ( column =a3e64f8f-bd44-4f28-b8d9-6938726e34d4:title,
                   value =4c61204772616e6765,  timestamp =1357599350478000 )
                 Returned 16 results.
```

The output of cell values is unreadable because GET returns the values in byte format.

Using a CQL legacy table query

Using CQL, you can query a legacy table. A legacy table managed in CQL includes an implicit WITH COMPACT STORAGE directive. When you use CQL to query legacy tables with no column names defined for data within a partition, Cassandra generates the names (column1 and value1) for the data. Using the RENAME clause, you can change the default column name to a more meaningful name.

```
ALTER TABLE users RENAME userid to user_id;
```

CQL supports dynamic tables created in the Thrift API, CLI, and earlier CQL versions. For example, a dynamic table is represented and queried like this:

```
CREATE TABLE clicks (
  userid uuid,
  url text,
  timestamp date,
  PRIMARY KEY  (userid, url ) ) WITH COMPACT STORAGE;

INSERT INTO clicks (userid, url,timestamp) VALUES
 (148e9150-1dd2-11b2-0000-242d50cf1fff,'http://google.com', '2016-02-03');

SELECT url, timestamp FROM clicks WHERE  userid =
 148e9150-1dd2-11b2-0000-242d50cf1fff;

SELECT timestamp FROM clicks WHERE  userid =
 148e9150-1dd2-11b2-0000-242d50cf1fff AND url = 'http://google.com';

SELECT timestamp FROM clicks WHERE  userid =
 148e9150-1dd2-11b2-0000-242d50cf1fff AND url > 'http://google.com';
```

CQL reference

Introduction

All of the commands included in the CQL language are available on the `cqlsh` command line. There are a group of commands that are available on the command line, but are not support by the CQL language. These commands are called `cqlsh` commands. You can run `cqlsh` commands from the command line only. You can run CQL commands in a number of ways.

This reference covers CQL and `cqlsh` based on the CQL specification 3.3.

CQL lexical structure

CQL input consists of statements. Like SQL, statements change data, look up data, store data, or change the way data is stored. Statements end in a semicolon (;).

For example, the following is valid CQL syntax:

```
SELECT * FROM MyTable;

UPDATE MyTable
  SET SomeColumn = 'SomeValue'
  WHERE columnName = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

This is a sequence of two CQL statements. This example shows one statement per line, although a statement can usefully be split across lines as well.

Uppercase and lowercase

Identifiers created using CQL are case-insensitive unless enclosed in double quotation marks. If you enter names for these objects using any uppercase letters, Cassandra stores the names in lowercase. You can force the case by using double quotation marks. For example:

```
CREATE TABLE test (
  Foo int PRIMARY KEY,
  "Bar" int
);
```

The following table shows partial queries that work and do not work to return results from the test table:

Table: What Works and What Doesn't

Queries that Work	Queries that Don't Work
SELECT foo FROM . . .	SELECT "Foo" FROM . . .
SELECT Foo FROM . . .	SELECT "BAR" FROM . . .
SELECT FOO FROM . . .	SELECT bar FROM . . .
SELECT "Bar" FROM . . .	SELECT Bar FROM . . .

Queries that Work	Queries that Don't Work
SELECT "foo" FROM . . .	SELECT "bar" FROM . . .

SELECT "foo" FROM ... works because internally, Cassandra stores foo in lowercase. The double-quotation mark character can be used as an escape character for the double quotation mark.

Case sensitivity rules in earlier versions of CQL apply when handling legacy tables.

CQL keywords are case-insensitive. For example, the keywords SELECT and select are equivalent. This document shows keywords in uppercase.

Valid characters for data types and keyspace/table/column names

This is not true when the names are double-quoted. We only enforce this for keyspaces and tables because the names are used in filenames.

Keyspace and table names created using CQL can only contain alphanumeric and underscore characters because these identifiers are used in Cassandra filenames. Other identifiers, such as columns, user-defined data type names and field names, user-defined function names, and user-defined aggregate names created using CQL can only contain alphanumeric and underscore characters if not enclosed in double quotation marks. If double quotation marks are used, any characters are valid for all identifiers except keyspace and table names. If you enter names for these objects using anything other than alphanumeric characters or underscores, Cassandra will issue an invalid syntax message and fail to create the object.

Table: What Works and What Doesn't

Creations that Work	Creations that Don't Work
CREATE TABLE foo ...	CREATE TABLE foo!$% ...
CREATE TABLE foo_bar ...	CREATE TABLE foo[]"90 ...
CREATE TABLE foo ("what#*&" text, ...)	CREATE TABLE foo (what#*& text, ...)
ALTER TABLE foo5 ...	ALTER TABLE "foo5$$"...
CREATE FUNCTION "foo5$$$^%" ...	CREATE FUNCTION foo5$$...
CREATE AGGREGATE "foo5!@#" ...	CREATE AGGREGATE foo5$$
CREATE TYPE foo5 ("bar#"9 text, ...	CREATE TYPE foo5 (bar#9 text ...

Escaping characters

Column names that contain characters that CQL cannot parse need to be enclosed in double quotation marks in CQL.

Dates, IP addresses, and strings need to be enclosed in single quotation marks. To use a single quotation mark itself in a string literal, escape it using a single quotation mark.

```
cqlsh> INSERT INTO cycling.calendar (race_id, race_start_date,
 race_end_date, race_name) VALUES
  (201, '2015-02-18', '2015-02-22', 'Women''s Tour of New Zealand');
```

An alternative is to use dollar-quoted strings. Dollar-quoted string constants can be used to create functions, insert data, and select data when complex quoting is needed. Use double dollar signs to enclose the desired string.

```
cqlsh> INSERT INTO cycling.calendar (race_id, race_start_date,
 race_end_date, race_name) VALUES
  (201, '2015-02-18', '2015-02-22', $$Women's Tour of New Zealand$$);
```

Valid literals

Valid literal consist of these kinds of values:

- blob

  hexadecimal defined as 0[xX](hex)+
- boolean

  true or false, case-insensitive, not enclosed in quotation marks
- numeric constant

  A numeric constant can consist of integers 0-9 and a minus sign prefix. A numeric constant can also be float. A float can be a series of one or more decimal digits, followed by a period, ., and one or more decimal digits. There is no optional + sign. The forms .42 and 42 are unacceptable. You can use leading or trailing zeros before and after decimal points. For example, 0.42 and 42.0. A float constant, expressed in E notation, consists of the characters in this regular expression:

  ```
  '-'?[0-9]+('.'[0-9]*)?([eE][+-]?[0-9+])?
  ```

  NaN and Infinity are floats.
- identifier

  Names of tables, columns, and other objects are identifiers. Identifiers fall into two categories based on whether or not the identifier is used in a Cassandra filename. Keyspace and table names are used to create data files, and must adhere to a sequence of letters, digits, or underscore. All other identifier, such as column and user-defined function names may be a sequence of letters, digits or underscore if not enclosed in double quotation marks. If an identifier is not a keyspace or table name, any characters are allowed if enclosed in double quotation marks.
- integer

  An optional minus sign, -, followed by one or more digits.
- string literal

  Characters enclosed in single quotation marks. To use a single quotation mark itself in a string literal, escape it using a single quotation mark. For example, use '' to make dog possessive: dog''s.
- uuid

  32 hex digits, 0-9 or a-f, which are case-insensitive, separated by dashes, -, after the 8th, 12th, 16th, and 20th digits. For example: 01234567-0123-0123-0123-0123456789ab
- timeuuid

  Uses the time in 100 nanosecond intervals since 00:00:00.00 UTC (60 bits), a clock sequence number for prevention of duplicates (14 bits), plus the IEEE 801 MAC address (48 bits) to generate a unique identifier. For example: d2177dd0-eaa2-11de-a572-001b779c76e3
- whitespace

  Separates terms and used inside string literals, but otherwise CQL ignores whitespace.

Exponential notation

Cassandra supports exponential notation. This example shows exponential notation in the output from a cqlsh command.

```
CREATE TABLE test(
  id varchar PRIMARY KEY,
  value_double double,
  value_float float
);

INSERT INTO test (id, value_float, value_double)
  VALUES ('test1', -2.6034345E+38, -2.6034345E+38);

SELECT * FROM test;
```

```
 id    | value_double | value_float
-------+--------------+-------------
 test1 |  -2.6034e+38 | -2.6034e+38
```

CQL code comments

You can use the following notation to include comments in CQL code:

- Double hyphen

  ```
  -- Single-line comment
  ```

- Double forward slash

  ```
  //Single-line comment
  ```

- Forward slash asterisk

  ```
  /* Multi-line comment */
  ```

CQL Keywords

This table lists keywords and whether or not the words are reserved. A reserved keyword cannot be used as an identifier unless you enclose the word in double quotation marks. Non-reserved keywords have a specific meaning in certain context but can be used as an identifier outside this context.

Table: Keywords

Keyword	Reserved
ADD	yes
AGGREGATE	yes
ALL	no
ALLOW	yes
ALTER	yes
AND	yes
ANY	yes

Keyword	Reserved
APPLY	yes
AS	no
ASC	yes
ASCII	no
AUTHORIZE	yes
BATCH	yes
BEGIN	yes
BIGINT	no
BLOB	no
BOOLEAN	no
BY	yes
CLUSTERING	no
COLUMNFAMILY	yes
COMPACT	no
CONSISTENCY	no
COUNT	no
COUNTER	no
CREATE	yes
CUSTOM	no
DECIMAL	no
DELETE	yes
DESC	yes
DISTINCT	no
DOUBLE	no
DROP	yes
EACH_QUORUM	yes
ENTRIES	yes
EXISTS	no
FILTERING	no
FLOAT	no
FROM	yes
FROZEN	no
FULL	yes
GRANT	yes

Keyword	Reserved
IF	yes
IN	yes
INDEX	yes
INET	yes
INFINITY	yes
INSERT	yes
INT	no
INTO	yes
KEY	no
KEYSPACE	yes
KEYSPACES	yes
LEVEL	no
LIMIT	yes
LIST	no
LOCAL_ONE	yes
LOCAL_QUORUM	yes
MAP	no
MATERIALIZED	yes
MODIFY	yes
NAN	yes
NORECURSIVE	yes
NOSUPERUSER	no
NOT	yes
OF	yes
ON	yes
ONE	yes
ORDER	yes
PARTITION	yes
PASSWORD	yes
PER	yes
PERMISSION	no
PERMISSIONS	no
PRIMARY	yes
QUORUM	yes

Keyword	Reserved
RENAME	yes
REVOKE	yes
SCHEMA	yes
SELECT	yes
SET	yes
STATIC	no
STORAGE	no
SUPERUSER	no
TABLE	yes
TEXT	no
TIME	yes
TIMESTAMP	no
TIMEUUID	no
THREE	yes
TO	yes
TOKEN	yes
TRUNCATE	yes
TTL	no
TUPLE	no
TWO	yes
TYPE	no
UNLOGGED	yes
UPDATE	yes
USE	yes
USER	no
USERS	no
USING	yes
UUID	no
VALUES	no
VARCHAR	no
VARINT	no
VIEW	yes
WHERE	yes
WITH	yes

Keyword	Reserved
WRITETIME	no

CQL data types

CQL defines built-in data types for columns. The counter type is unique.

Table: CQL Data Types

CQL Type	Constants	Description
ascii	strings	US-ASCII character string
bigint	integers	64-bit signed long
blob	blobs	Arbitrary bytes (no validation), expressed as hexadecimal
boolean	booleans	true or false
counter	integers	Distributed counter value (64-bit long)
date	strings	Date string, such as 2015-05-03
decimal	integers, floats	Variable-precision decimal Java type **Note:** When dealing with currency, it is a best practice to have a currency class that serializes to and from an int or use the Decimal form.
double	integers, floats	64-bit IEEE-754 floating point Java type
float	integers, floats	32-bit IEEE-754 floating point Java type
frozen	user-defined types, collections, tuples	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. **Note:** Cassandra no longer requires the use of frozen for tuples: `frozen <tuple <int, tuple<text, double>>>`
inet	strings	IP address string in IPv4 or IPv6 format, used by the python-cql driver and CQL native protocols
int	integers	32-bit signed integer
list	n/a	A collection of one or more ordered elements: [literal, literal, literal]

CQL Type	Constants	Description
map	n/a	A JSON-style array of literals: { literal : literal, literal : literal ... }
set	n/a	A collection of one or more elements: { literal, literal, literal }
smallint	integers	2 byte integer
text	strings	UTF-8 encoded string
time	strings	Time string, such as 13:30:54.234
timestamp	integers, strings	Date plus time, encoded as 8 bytes since epoch
timeuuid	uuids	Version 1 UUID only
tinyint	integers	1 byte integer
tuple	n/a	Cassandra 2.1 and later. A group of 2-3 fields.
uuid	uuids	A UUID in standard UUID format
varchar	strings	UTF-8 encoded string
varint	integers	Arbitrary-precision integer Java type

In addition to the CQL types listed in this table, you can use a string containing the name of a JAVA class (a sub-class of AbstractType loadable by Cassandra) as a CQL type. The class name should either be fully qualified or relative to the org.apache.cassandra.db.marshal package.

Enclose ASCII text, timestamp, and inet values in single quotation marks. Enclose names of a keyspace, table, or column in double quotation marks.

Java types

The Java types, from which most CQL types are derived, are obvious to Java programmers. The derivation of the following types, however, might not be obvious:

Table: Derivation of selective CQL types

CQL type	Java type
decimal	java.math.BigDecimal
float	java.lang.Float
double	java.lang.Double
varint	java.math.BigInteger

CQL type compatibility

CQL data types have strict requirements for conversion compatibility. The following table shows the allowed alterations for data types:

Data type may be altered to:	Data type
ascii, bigint, boolean, decimal, double, float, inet, int, timestamp, timeuuid, uuid, varchar, varint	blob

Data type may be altered to:	Data type
int	varint
text	varchar
timeuuid	uuid
varchar	text

Clustering columns have even stricter requirements, because clustering columns mandate the order in which data is written to disk. The following table shows the allow alterations for data types used in clustering columns:

Data type may be altered to:	Data type
int	varint
text	varchar
varchar	text

Blob type

The Cassandra blob data type represents a constant hexadecimal number defined as 0[xX](hex)+ where hex is an hexadecimal character, such as [0-9a-fA-F]. For example, 0xcafe. The maximum theoretical size for a blob is 2GB. The practical limit on blob size, however, is less than 1 MB, ideally even smaller. A blob type is suitable for storing a small image or short string.

Blob conversion functions

These functions convert the native types into binary data (blob):

- typeAsBlob(value)
- blobAsType(value)

For every native, nonblob data type supported by CQL, the typeAsBlob function takes a argument of that data type and returns it as a blob. Conversely, the blobAsType function takes a 64-bit blob argument and converts it to a value of the specified data type, if possible.

This example shows how to use bigintAsBlob:

```
CREATE TABLE bios ( user_name varchar PRIMARY KEY,
   bio blob
 );

INSERT INTO bios (user_name, bio) VALUES ('fred', bigintAsBlob(3));

SELECT * FROM bios;

 user_name | bio
-----------+--------------------
      fred | 0x0000000000000003
```

This example shows how to use blobAsBigInt.

```
ALTER TABLE bios ADD id bigint;

INSERT INTO bios (user_name, id) VALUES ('fred',
 blobAsBigint(0x0000000000000003));
```

```
SELECT * FROM bios;

 user_name | bio                | id
-----------+--------------------+----
      fred | 0x0000000000000003 |  3
```

Collection type

A collection column is declared using the collection type, followed by another type, such as int or text, in angle brackets. For example, you can create a table having a list of textual elements, a list of integers, or a list of some other element types.

```
list<text>
list<int>
```

Collection types cannot be nested, but frozen collection types can be nested inside frozen or non-frozen collections. For example, you may define a list within a list, provided the inner list is frozen:

```
list<frozen <list<int>>>
```

Indexes may be created on a collection column of any type.

Using frozen in a collection

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.

Note: Frozen collections can be used for primary key columns. Non-frozen collections cannot be used for primary key columns.

```
column_name <collection_type><cql_type, frozen<column_name>>
```

For example:

```
CREATE TABLE mykeyspace.users (
  id uuid PRIMARY KEY,
  name frozen <fullname>,
  direct_reports set<frozen <fullname>>,     // a collection set
  addresses map<text, frozen <address>>     // a collection map
  score set<frozen <set<int>>>              // a set with a nested frozen
 set
);
```

Counter type

A counter column value is a 64-bit signed integer. You cannot set the value of a counter, which supports two operations: increment and decrement.

Use counter types as described in the "Using a counter" section. Do not assign this type to a column that serves as the primary key or partition key. Also, do not use the counter type in a table that contains anything other than counter types and the primary key. To generate sequential numbers for surrogate keys, use the timeuuid type instead of the counter type. You cannot create an index on a counter column or set data in a counter column to expire using the Time-To-Live (TTL) property.

UUID and timeuuid types

The UUID (universally unique id) comparator type is used to avoid collisions in column names. Alternatively, you can use the timeuuid.

Timeuuid types can be entered as integers for CQL input. A value of the timeuuid type is a Type 1 UUID. A Version 1 UUID includes the time of its generation and are sorted by timestamp, making them ideal for use in applications requiring conflict-free timestamps. For example, you can use this type to identify a column (such as a blog entry) by its timestamp and allow multiple clients to write to the same partition key simultaneously. Collisions that would potentially overwrite data that was not intended to be overwritten cannot occur.

A valid timeuuid conforms to the timeuuid format shown in valid literals.

uuid and Timeuuid functions

Cassandra 2.0.7 and later includes the uuid() function. This function takes no parameters and generates a random Type 4 UUID suitable for use in `INSERT` or `SET` statements.

Several timeuuid functions are designed for use with the timeuuid type:

- `dateOf()`

  Used in a `SELECT` clause, this function extracts the timestamp of a timeuuid column in a result set. This function returns the extracted timestamp as a date. Use `unixTimestampOf()` to get a raw timestamp.
- `now()`

  In the coordinator node, generates a new unique timeuuid in milliseconds when the statement is executed. The timestamp portion of the timeuuid conforms to the UTC (Universal Time) standard. This method is useful for inserting values. The value returned by `now()` is guaranteed to be unique.
- `minTimeuuid()` and `maxTimeuuid()`

  Returns a UUID-like result given a conditional time component as an argument. For example:

  ```
  SELECT * FROM myTable
     WHERE t > maxTimeuuid('2013-01-01 00:05+0000')
     AND t < minTimeuuid('2013-02-02 10:00+0000')
  ```

  The min/maxTimeuuid example selects all rows where the timeuuid column, t, is strictly later than 2013-01-01 00:05+0000 but strictly earlier than 2013-02-02 10:00+0000. The t >= maxTimeuuid('2013-01-01 00:05+0000') does not select a timeuuid generated exactly at 2013-01-01 00:05+0000 and is essentially equivalent to t > maxTimeuuid('2013-01-01 00:05+0000').

  The values returned by `minTimeuuid` and `maxTimeuuid` functions are not true UUIDs in that the values do not conform to the Time-Based UUID generation process specified by the RFC 4122. The results of these functions are deterministic, unlike the `now()` function.
- `unixTimestampOf()`

  Used in a `SELECT` clause, this functions extracts the timestamp in milliseconds of a timeuuid column in a result set. Returns the value as a raw, 64-bit integer timestamp.

Cassandra 2.2 and later support some additional timeuuid and timestamp functions to manipulate dates. The functions can be used in `INSERT`, `UPDATE`, and `SELECT` statements.

- `toDate(timeuuid)`

  Converts timeuuid to date in YYYY-MM-DD format.
- `toTimestamp(timeuuid)`

  Converts timeuuid to timestamp format.
- `toUnixTimestamp(timeuuid)`

Converts timeuuid to UNIX timestamp format.

- `toDate(timestamp)`

  Converts timestamp to date in YYYY-MM-DD format.
- `toUnixTimestamp(timestamp)`

  Converts timestamp to UNIX timestamp format.
- `toTimestamp(date)`

  Converts date to timestamp format.
- `toUnixTimestamp(date)`

  Converts date to UNIX timestamp format.

An example of the new functions creates a table and inserts various time-related values:

```
CREATE TABLE sample_times (a int, b timestamp, c timeuuid, d bigint, PRIMARY
 KEY (a,b,c,d));
INSERT INTO sample_times (a,b,c,d) VALUES (1, toUnixTimestamp(now()),
 50554d6e-29bb-11e5-b345-feff819cdc9f, toTimestamp(now()));
```

```
a | b                         | c                                    | d
---+---------------------------+--------------------------------------+---------------
1 | 2015-07-13 17:13:37-0700 | 50554d6e-29bb-11e5-b345-feff819cdc9f | 1436832817476
```

Select data and convert it to a new format:

```
SELECT toDate(c) FROM sample_times;
```

```
 a | b                         | system.todate(c) | system.todate(d)
---+---------------------------+------------------+------------------
 1 | 2015-07-13 17:13:37-0700 |       2015-07-14 |       2015-07-14
```

Timestamp type

Values for the timestamp type are encoded as 64-bit signed integers representing a number of milliseconds since the standard base time known as the epoch: January 1 1970 at 00:00:00 GMT. A timestamp type can be entered as an integer for CQL input, or as a string literal in any of the following ISO 8601 formats:

```
yyyy-mm-dd HH:mm
yyyy-mm-dd HH:mm:ss
yyyy-mm-dd HH:mmZ
yyyy-mm-dd HH:mm:ssZ
yyyy-mm-dd'T'HH:mm
yyyy-mm-dd'T'HH:mmZ
yyyy-mm-dd'T'HH:mm:ss
yyyy-mm-dd'T'HH:mm:ssZ
yyyy-mm-dd'T'HH:mm:ss.ffffffZ
yyyy-mm-dd
yyyy-mm-ddZ
```

where Z is the RFC-822 4-digit time zone, expressing the time zone's difference from UTC. For example, for the date and time of Jan 2, 2003, at 04:05:00 AM, GMT:

```
2011-02-03 04:05+0000
2011-02-03 04:05:00+0000
```

```
2011-02-03T04:05+0000
2011-02-03T04:05:00+0000
```

In Cassandra 3.4 and later, timestamps are displayed in `cqlsh` in sub-second precision by default, as shown below. Applications reading a timestamp may use the sub-second portion of the timestamp, as Cassandra stored millisecond-precision timestamps in all versions.

```
%Y-%m-%d %H:%M:%S.%f%z
```

If no time zone is specified, the time zone of the Cassandra coordinator node handing the write request is used. For accuracy, DataStax recommends specifying the time zone rather than relying on the time zone configured on the Cassandra nodes.

If you only want to capture date values, you can also omit the time of day. For example:

```
2011-02-03
2011-02-03+0000
```

In this case, the time of day defaults to 00:00:00 in the specified or default time zone.

Timestamp output appears in the following format by default in `cqlsh`:

```
yyyy-mm-dd HH:mm:ssZ
```

You can change the format by setting the `time_format` property in the `[ui]` section of the `cqlshrc` file.

```
[ui]
time_format = %Y-%m-%d %H:%M
```

Tuple type

The tuple data type holds fixed-length sets of typed positional fields. Use a tuple as an alternative to a user-defined type. A tuple can accommodate many fields (32768), more than can be prudently used. Typically, create a tuple with a few fields.

In the table creation statement, use angle brackets and a comma delimiter to declare the tuple component types. Surround tuple values in parentheses to insert the values into a table, as shown in this example.

```
CREATE TABLE collect_things (
  k int PRIMARY KEY,
  v tuple<int, text, float>
);

INSERT INTO collect_things (k, v) VALUES(0, (3, 'bar', 2.1));

SELECT * FROM collect_things;

 k | v
---+-----------------
 0 | (3, 'bar', 2.1)
```

Note: Cassandra no longer requires the use of frozen for tuples:

```
frozen <tuple <int, tuple<text, double>>>
```

You can filter a selection using a tuple.

```
CREATE INDEX on collect_things (v);

SELECT * FROM collect_things WHERE v = (3, 'bar', 2.1);
```

```
 k | v
---+-----------------
 0 | (3, 'bar', 2.1)
```

You can nest tuples as shown in the following example:

```
CREATE TABLE nested (k int PRIMARY KEY, t tuple <int, tuple<text, double>>);

INSERT INTO nested (k, t) VALUES (0, (3, ('foo', 3.4)));
```

User-defined type

A user-defined type facilitates handling multiple fields of related information in a table. Applications that required multiple tables can be simplified to use fewer tables by using a user-defined type to represent the related fields of information instead of storing the information in a separate table. The address type example demonstrates how to use a user-defined type.

You can create, alter, and drop a user-defined type using these commands:

- CREATE TYPE
- ALTER TYPE
- DROP TYPE

The cqlsh utility includes these commands for describing a user-defined type or listing all user-defined types:

- DESCRIBE TYPE
- DESCRIBE TYPES

The scope of a user-defined type is the keyspace in which you define it. Use dot notation to access a type from a keyspace outside its scope: keyspace name followed by a period followed the name of the type. An example is `test.myType` where `test` is the keyspace name and `myType` is the type name. Cassandra accesses the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, Cassandra accesses the type within the current keyspace.

CQL keyspace and table properties

The CQL WITH clause specifies keyspace and table properties in these CQL commands:

- ALTER KEYSPACE
- ALTER TABLE
- CREATE KEYSPACE
- CREATE TABLE

CQL keyspace properties

CQL supports setting the following keyspace properties in addition to naming data centers.

- **replication**

  `replication` is a keyspace property that must be set for each keyspace created. Two sub-properties must be set to configure this setting:

  - class

    The name of the replication strategy: SimpleStrategy or NetworkTopologyStrategy. The replication factor can be set independently for each data center when using NetworkTopologyStrategy.

SimpleStrategy should be used for evaluation only. NetworkTopologyStrategy is recommended for production deployments that may require future expansion and multiple data centers.

- replication_factor

  The replication factor value is the total number of replicas across the cluster. This setting is used with SimpleStrategy, as shown in CREATE KEYSPACE examples.

- <data center>

  This value is the number of replicas on each node in each data center. The <data center> sub-property is used with NetworkTopologyStrategy, as shown in CREATE KEYSPACE examples. Multiple data centers may be configured.

- **durable_writes**

  This property can be set to either true or false to turn on or off durable writes for a keyspace.

Table properties

CQL supports Cassandra table properties, such as comments and compaction options, listed in the following table.

In CQL commands, such as CREATE TABLE, add properties and their values in either the name-value pair or collection map format:

```
name = value [ AND name = value ] [ ... ]
```

```
{ name : value, name : value [, name : value ] [ ... ] }
```

Enclose strings in single quotation marks.

See CREATE TABLE for examples.

Table: CQL properties

CQL property	Default	Description
bloom_filter_fp_chance	For `SizeTieredCompactionStrategy`: `0.01`, for `DateTieredCompactionStrategy`: `0.1`, for `LeveledCompactionStrategy`: `0.1`	Desired false-positive probability for SSTable Bloom filters. See Bloom filter below.
caching	Cassandra 2.1: ALL for keys NONE for rows_per_partition Cassandra 2.0.x: keys_only	Optimizes the use of cache memory without manual tuning. See caching below.
comment	N/A	A human readable comment describing the table. See comments below.
compaction	`SizeTieredCompactionStrategy`	Sets the compaction strategy for the table. See compaction below.
compression	`LZ4Compressor`	The compression algorithm. Valid values are `LZ4Compressor`), `SnappyCompressor`, and `DeflateCompressor`. See compression below.

CQL property	Default	Description
dclocal_read_repair_chance	In Cassandra 2.1, Cassandra 2.0.9 and later: `0.1`, in Cassandra 2.0.8 and earlier: `0.0`	The probability that a successful read operation triggers a read repair. Unlike the repair controlled by read_repair_chance, this repair is limited to replicas in the same DC as the coordinator. The value must be between `0` and `1`. For details see read repairs below.
default_time_to_live	`0`	The default expiration time in seconds for a table. Drops table contents without creating tombstones. Use in MapReduce scenarios when you have no control of TTL.
gc_grace_seconds	`864000` [10 days]	The number of seconds after data is marked with a tombstone (deletion marker) before it is eligible for garbage-collection. Cassandra will not execute hints or batched mutations on a tombstoned record within its gc_grace_period. The default value allows a great deal of time for Cassandra to maximize consistency prior to deletion. For details about decreasing this value, see garbage collection below.
memtable_flush_period_in_ms	`0`	The number of milliseconds before Cassandra flushes memtables associated with this table.
min_index_interval		The minimum gap between index entries in the index summary. A lower min_index_interval means the index summary contains more entries from the index, which allows Cassandra to search fewer index entries to execute a read. A larger index summary may also use more memory. The value for min_index_interval is the densest possible sampling of the index.
max_index_interval		If the total memory usage of all index summaries reaches this value, Cassandra decreases the index summaries for the coldest SSTables to the maximum set by max_index_interval. The max_index_interval is the sparsest possible sampling in relation to memory pressure.
read_repair_chance	`0.0`	The probability that a successful read operation will trigger a read repair.of read repairs being invoked. Unlike the repair controlled by dc_local_read_repair_chance, this repair is not limited to replicas in the same DC as the coordinator. The value must be between `0` and `1`. For details see read repairs below.
speculative_retry	99percentile	Overrides normal read timeout when read_repair_chance is not 1.0, sending another request to read. See speculative retry below.

Bloom filter

The Bloom filter sets the false-positive probability for SSTable Bloom filters. When a client requests data, Cassandra uses the Bloom filter to check if the row exists before doing disk I/O. Bloom filter property value ranges from 0 to 1.0. Lower Bloom filter property probabilities result in larger Bloom filters that use more memory. The effects of the minimum and maximum values:

- `0`: Enables the unmodified, effectively the largest possible, Bloom filter.
- `1.0`: Disables the Bloom filter.

Recommended setting: `0.1`. A higher value yields diminishing returns.

caching

Caching optimizes the use of cache memory by a table without manual tuning. Cassandra weighs the cached data by size and access frequency. Coordinate this setting with the global caching properties in the `cassandra.yaml` file. See Cassandra 3.0 documentation.

Configure the cache by creating a property map of values for the caching property. Options:

- keys: `ALL` or `NONE`
- rows_per_partition: number of CQL rows (N), `NONE`, or `ALL`

According to the rows_per_partition value, Cassandra caches only the first N rows in a partition, as determined by the clustering order.

For example:

```
CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (species, block_id)
) WITH caching = { 'keys' : 'NONE', 'rows_per_partition' : '120' };
```

comments

Use comments to document CQL statements in your application code. Single line comments can begin with a double dash (--) or a double slash (//) and extend to the end of the line. Enclose multi-line comments in /* and */ characters.

compaction

The compaction property defines the compaction strategy class for this table. Choose the compaction strategy that best fits your data and environment.

Note: For more guidance, see the When to Use Leveled Compaction, Leveled Compaction in Apache Cassandra blog.

- `SizeTieredCompactionStrategy (STCS)`: The default compaction strategy. This strategy triggers a minor compaction when there are a number of similar sized SSTables on disk as configured by the table subproperty min_threshold. A minor compaction does not involve all the tables in a keyspace. See STCS compaction subproperties.
- `DateTieredCompactionStrategy (DTCS)`: DateTieredCompactionStrategy stores data written within a certain period of time in the same SSTable. For example, Cassandra stores your last hour of data in one SSTable *time window*, and the next 4 hours of data in another time window, and so on. Cassandra performs compaction when the number of SSTables in those windows reaches min_threshold (4 by default). The most common queries for time series workloads retrieve the last hour/day/month of data. Cassandra can limit SSTables returned to those having the relevant data. Cassandra can then drop the SSTable without doing any compaction. Also see DTCS compaction subproperties.
- `LeveledCompactionStrategy (LCS)`: Creates SSTables that are grouped into levels. Within each level, SSTables are guaranteed to be non-overlapping. LCS uses STCS for the first level, called *level 0* (`L0`). Each level beyond L0 (L1, L2 and so on) is 10 times as large as the previous level. Disk I/O is more uniform and predictable on higher than on lower levels as SSTables are continuously being compacted into progressively larger levels. At each level, row keys are merged into non-overlapping SSTables. This can improve performance for reads, because Cassandra can determine which SSTables in each level to check for the existence of row key data. This compaction strategy is modeled after Google's LevelDB implementation. See LCS compaction subproperties.

Hybrid (leveled and size-tiered) compaction improvements to the leveled compaction strategy reduce the performance overhead on read operations when compaction cannot keep pace with write-heavy workload. If Cassandra cannot keep pace with the workload when using the LCS, the compaction strategy switches to STCS until Cassandra catches up. For this reason, it is a best practice to configure the max_threshold subproperty for a table to use when the switch occurs.

You can specify a custom strategy. Use the full class name as a string constant.

garbage collection

The default value or gr_grace_seconds is 864000 seconds (10 days). In a single-node cluster, this property can safely be set to zero. This value can also be reduced for tables whose data will not be explicitly deleted — for example, tables containing only data with TTL set, or tables with default_time_to_live set. However, if you lower the gr_grace_seconds value, consider its interaction with these operations:

- **hint replays** — When a node goes down and then comes back up, other nodes replay write operations that Cassandra has queued up for that node while it was unresponsive (called hints). Cassandra does not replay a hint until gc_grace_seconds after it was created. Lowering the table's gc_grace_seconds increases the chance that Cassandra replays a hint for a record after that record has been garbage-collected, restoring deleted data. The number of hints collected for an unresponsive node is controlled for the node by the max_hint_window_in_ms setting in the cassandra.yaml file.
- **batch replays** — Like hint queues, batch operations store database mutations that are replayed in sequence. As with hints, Cassandra does not replay a batched mutation until gc_grace_seconds after it was created. If your application uses batch operations, consider the possibility that decreasing gc_grace_seconds increases the chance that a batched write operation may restore deleted data. The batchlog_replay_throttle_in_kb and concurrent_batchlog_writes properties in the cassandra.yaml file give some control of the batch replay process. The most important factors, however, are the size and scope of the batches you use.

For more details about deletes, see How is data deleted?.

compression

Configure compression when creating or altering a table by setting compaction property to `LZ4Compressor`, `SnappyCompressor`, or `DeflateCompressor`. To disable compression, use an empty string (" "), as shown in the example of how to use subproperties. Choosing the right compressor depends on your requirements for space savings over read performance. LZ4 is fastest to decompress, followed by Snappy, then by Deflate. Compression effectiveness is inversely correlated with decompression speed. The extra compression from Deflate or Snappy does not offset the decreased performance for general-purpose workloads, but these compresssors may work for archival data. You can also implement a custom compression class using the `org.apache.cassandra.io.compress.ICompressor` interface. To specify a custom compressor class, add the full class name enclosed in single quotation marks as the value for the compaction property. You can configure compression further using Cassandra's compression subproperties.

read repairs

Cassandra performs read repair whenever a read reveals inconsistencies among replicas. You can also configure Cassandra to perform read repair after a completely consistent read. Cassandra compares and coordinates all replicas, even those that were not accessed in the successful read. The probability that a consistent read of a table triggers a read repair is set by dclocal_read_repair_chance and read_repair_chance. The first of these properties sets the probability for a read repair that is confined to the same datacenter as the coordinator node. The second property sets the probability for a read repair across all datacenters that contain matching replicas. This cross-datacenter operation is much more resource-intensive than the local operation.

Recommendations: if the table is for time series data, both properties can be set to `0` (zero). For other tables, the more performant strategy is to set dc_local_read_repair_chance to `0.1` and read_repair_chance to `0`. If you want to use read_repair_chance, set this property to `0.1`.

speculative retry

Use the speculative retry property to configure rapid read protection. In a normal read, Cassandra sends data requests to just enough replica nodes to satisfy the consistency level. In rapid read protection, Cassandra sends out extra read requests to other replicas, even after the consistency level has been met. The speculative retry property specifies the trigger for these extra read requests.

- `ALWAYS`: Send extra read requests to all other replicas after every read.
- **`X`**`percentile`: Cassandra constantly tracks each table's typical read latency (in milliseconds). If you set speculative retry to **`X`**`percentile`, Cassandra sends redundant read requests if the coordinator has not received a response after **X** percent of the table's typical latency time.
- **`N`**`ms`: Send extra read requests to all other replicas if the coordinator node has not received any responses within `N` milliseconds.
- `NONE`: Do not send extra read requests after any read.

For example:

```
ALTER TABLE users WITH speculative_retry = '10ms';
```

Or:

```
ALTER TABLE users WITH speculative_retry = '99percentile';
```

Related information

Cassandra 2.1 cassandra.yaml
Cassandra 2.2 cassandra.yaml
Cassandra 3.0 cassandra.yaml

Compaction subproperties

Using CQL, you can configure a table to use SizeTieredCompactionStrategy (STCS), DateTieredCompactionStrategy (DTCS), or LeveledCompactionStrategy (LCS). You can specify a compaction strategy for a new table using the CREATE TABLE command, or change or reconfigure an existing table's strategy using ALTER TABLE. To configure the compaction strategy, construct a map of the compaction property and the relevant subproperties:

Table: CQL compaction subproperties for STCS

Compaction Subproperties	Default	Description
bucket_high	1.5	Size-tiered compaction merges sets of SSTables that are approximately the same size. Casssandra compares each SSTable size to the average of all SSTable sizes on the node. It merges SSTAbles whose sizes in KB are within [average-size × bucket_low] and [average-size × bucket_high].
bucket_low	0.5	See above.
enabled	true	Enables background compaction. See Enabling and disabling background compaction.
log_all	false	Activates advanced logging for the entire cluster.
max_threshold	32	The maximum number of SSTables to allow in a minor compaction.

Compaction Subproperties	Default	Description
min_threshold	4	The minimum number of SSTables to trigger a minor compaction.
min_sstable_size	50MB	STCS groups SSTables into buckets. The bucketing process groups SSTables that differ in size by less than 50%. This bucketing process is too fine grained for small SSTables. If your SSTables are small, use min_sstable_size to define a size threshold (in bytes) below which all SSTables belong to one unique bucket.
only_purge_repaired_tombstones	false	In Cassandra 3.0 and later: *true* allows purging tombstones only from repaired SSTables. The purpose is to prevent data from resurrecting if repair is not run within gc_grace_seconds. If you do not run repair for a long time, Cassandra keeps all tombstones — this may cause problems.
tombstone_compaction_interval	86400 (one day)	The minimum number of seconds after which an SSTable is created before Cassandra considers the SSTable for tombstone compaction. An SSTable is eligible for tombstone compaction if the table exceeds the tombstone_threshold ratio.
tombstone_threshold	0.2	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, Cassandra starts compaction on that table alone, to purge the tombstones.
unchecked_tombstone_compaction	false	*True* allows Cassandra to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, Cassandra checks an SSTable to make sure it is safe to drop tombstones.

Table: CQL Compaction subproperties for DTCS

Compaction Subproperties	Default	Description
base_time_seconds	3600 (1 hour)	The size of the first time window.
enabled	true	Enables background compaction. See Enabling and disabling background compaction.
log_all	false	*True* activates advanced logging for the entire cluster.
max_sstable_age_days	1000	*True* stops compacting SSTables only having data older than these specified days. Fractional days can be set. This parameter is deprecated in Casandra3.2.
max_window_size_seconds	864000 (one day)	The maximum window size in seconds.
max_threshold	32	The maximum number of SSTables allowed in a minor compaction.
min_threshold	4	The minimum number of SSTables that trigger a minor compaction.

Compaction Subproperties	Default	Description
timestamp_resolution	MICROSECONDS	Set to *MICROSECONDS* or *MILLISECONDS*, to match the timestamp unit of the data you insert
tombstone_compaction_interval	864000(1 day)	The minimum number of seconds after an SSTable is created before Cassandra considers the SSTable for tombstone compaction. Tombstone compaction is triggered if the number of garbage-collectable tombstones in the SSTable is greater than tombstone_threshold.
tombstone_threshold	0.2	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, Cassandra starts compaction on that table alone, to purge the tombstones.
unchecked_tombstone_compaction	false	*True* allows Cassandra to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, Cassandra checks an SSTable to make sure it is safe to drop tombstones.

Table: CQL compaction Subproperties for LCS

Compaction Subproperties	Default	Description
enabled	true	Enables background compaction. See cold_reads_to_omit below.
log_all	false	*True* activates advanced logging for the entire cluster.
sstable_size_in_mb	160MB	The target size for SSTables that use the Leveled Compaction Strategy. Although SSTable sizes should be less or equal to sstable_size_in_mb, it is possible tthat compaction may produce a larger SSTable during compaction. This occurs when data for a given partition key is exceptionally large. Cassandra does not splie the data into two SSTables.
tombstone_compaction_interval	864000 (one day)	The minimum number of seconds after an SSTable is created before Cassandra considers the SSTable for tombstone compaction. Cassandra begins tombstone compaction SSTable's tombstone_threshold exceeds value of the following property.
tombstone_threshold	0.2	The ratio of garbage-collectable tombstones to all contained columns. If the ratio exceeds this limit, Cassandra starts compaction on that table alone, to purge the tombstones.
unchecked_tombstone_compaction	false	*True* allows Cassandra to run tombstone compaction without pre-checking which tables are eligible for this operation. Even without this pre-check, Cassandra checks an SSTable to make sure it is safe to drop tombstones.

Enabling and disabling background compaction

The following example sets the *enable* property to disable background compaction:

```
ALTER TABLE mytable WITH COMPACTION = {'class':
 'SizeTieredCompactionStrategy', 'enabled': 'false' }
```

Disabling background compaction can be harmful: without it, Cassandra does not regain disk space, and may allow zombies to propagate. Although compaction uses I/O, it is better to leave it enabled in most cases.

Enabling extended compaction logging

You can configure Casandra to collect in-depth information about compaction activity on a node, and write it to a dedicated log file. To enable extended compaction logging, all `log-all : true` to the configuration map for any table.

Important: If you enable extended logging for any table on any node, Cassandra enables it for all tables on all nodes in the cluster.

When extended compaction is enabled, Cassandra creates a file named `compaction-%d.log` (where `%d` is a sequential number) in `$CASSANDRA_HOME/logs`.

The compaction logging service logs detailed information about four types of compaction events:

- `type:enable`

  Lists SSTables that have been flushed previouly

  ```
  {"type":"enable","keyspace":"test","table":"t","time":1470071098866,"strategies":
    [
      {"strategyId":"0","type":"LeveledCompactionStrategy","tables":
  [],"repaired":true,"folders":
        ["/home/carl/oss/cassandra/bin/../data/data"]},
      {"strategyId":"1","type":"LeveledCompactionStrategy","tables":
  [],"repaired":false,"folders":
        ["/home/carl/oss/cassandra/bin/../data/data"]
      }
    ]
  }
  ```

- `type: flush`

  Logs a flush event from a memtable to an SSTable on disk, including the CompactionStrategy for each table.

  ```
  {"type":"flush","keyspace":"test","table":"t","time":1470083335639,"tables":
    [
      {"strategyId":"1","table":
        {"generation":1,"version":"mb","size":106846362,"details":

   {"level":0,"min_token":"-9221834874718566760","max_token":"92213696997139245178"}
        }
      }
    ]
  }
  ```

- `type: compaction`

  Logs a compaction event.

  ```
  {"type":"compaction","keyspace":"test","table":"t","time":1470083660267,"start":"1470
    [
      {"strategyId":"1","table":
  ```

```
      {"generation":1372,"version":"mb","size":1064979,"details":

 {"level":1,"min_token":"7199305267944662291","max_token":"7323434447996777057"}
      }
    }
  ],"output":
  [
    {"strategyId":"1","table":
      {"generation":1404,"version":"mb","size":1064306,"details":

 {"level":2,"min_token":"7199305267944662291","max_token":"7323434447996777057"}
      }
    }
  ]
}
```

- `type: pending`

  Lists the number of pending tasks for a compaction strategy

```
{"type":"pending","keyspace":"test","table":"t","time":1470083447967,"strategyId":"1"
```

Compression subproperties

Using CQL, you can configure compression for a table by constructing a map of the compaction property and the following subproperties:

Table: CQL Compression Subproperties

Compression Subproperties	Default	Description
class	`LZ4Compressor`	The compression algorithm to use. Valid values are `LZ4Compressor`, `SnappyCompressor`, and `DeflateCompressor`. See sstable_compression below.
chunk_length_kb	64KB	On disk, SSTables are compressed by block to allow random reads. This subproperty of compression defines the size (in KB) of the block. Values larger than the default value might improve the compression rate, but increases the minimum size of data to be read from disk when a read occurs. The default value is a good middle-ground for compressing tables. Adjust compression size to account for read/write access patterns (how much data is typically requested at once) and the average size of rows in the table.
crc_check_chance	1.0	When compression is enabled, each compressed block includes a checksum of that block for the purpose of detecting disk bitrot and avoiding the propagation of corruption to other replica. This option defines the probability with which those checksums are checked during read. By default they are always checked. Set to 0 to disable checksum checking and to 0.5, for instance, to check them on every other read.

sstable_compression

The compression algorithm to use. Valid values are `LZ4Compressor` `SnappyCompressor`, and `DeflateCompressor`. Use an empty string ('') to disable compression:

```
ALTER TABLE mytable WITH COMPRESSION = {'sstable_compression': ''};
```

Choosing the right compressor depends on your requirements for space savings over read performance. LZ4 is fastest to decompress, followed by Snappy, then by Deflate. Compression effectiveness is inversely correlated with decompression speed. The extra compression from Deflate or Snappy is not enough to make up for the decreased performance for general-purpose workloads, but for archival data they may be worth considering. Developers can also implement custom compression classes using the `org.apache.cassandra.io.compress.ICompressor` interface. Specify the full class name as a "string constant".

Functions

CQL supports several functions that transform one or more column values into a new value. In addition, users can define functions and aggregates. The native Cassandra functions are:

- Blob conversion functions
- UUID and Timeuuid functions
- Token function
- WRITETIME function
- TTL function
- Standard aggregate functions such as MIN(), MAX(), SUM(), and AVG().
- TOKEN function

CQL limits

Observe the following upper limits:

- Cells in a partition: 2B (2^{31}); single column value size: 2 GB (1 MB is recommended)
- Clustering column value, length of: 65535 (2^{16}-1)
- Key length: 65535 (2^{16}-1)
- Table / CF name length: 48 characters
- Keyspace name length: 48 characters
- Query parameters in a query: 65535 (2^{16}-1)
- Statements in a batch: 65535 (2^{16}-1)
- Fields in a tuple: 32768 (2^{15}) (just a few fields, such as 2-10, are recommended)
- Collection (List): collection size: 2B (2^{31}); values size: 65535 (2^{16}-1) (Cassandra 2.1 and later, using native protocol v3)
- Collection (Set): collection size: 2B (2^{31}); values size: 65535 (2^{16}-1) (Cassandra 2.1 and later, using native protocol v3)
- Collection (Map): collection size: 2B (2^{31}); number of keys: 65535 (2^{16}-1); values size: 65535 (2^{16}-1) (Cassandra 2.1 and later, using native protocol v3)
- Blob size: 2 GB (less than 1 MB is recommended)

Note: The limits specified for collections are for non-frozen collections.

cqlsh commands

Important: The commands listed on this page, and described in this section, work only within the cqlsh shell. These commands are not accessible from drivers.

cqlsh

Start the CQL interactive terminal.

Synopsis

```
cqlsh [options] [host [port]]
```

```
python cqlsh.py [options] [host [port]]
```

Description

The Cassandra installation includes the cqlsh utility, a python-based command line client for executing Cassandra Query Language (CQL) commands. The cqlsh command is used on the Linux or Windows command line to start the cqlsh utility. On Windows, the keyword *python* is used if the PATH environment variable does not point to the python installation.

You can use cqlsh to execute CQL commands interactively. cqlsh supports tab completion. You can also execute cqlsh commands, such as TRACE.

Requirements

The cqlsh utility uses the native protocol and the Datastax python driver. The default cqlsh listen port is 9042.

For more information about configuration, see the Cassandra 3.0 cassandra.yaml or Cassandra 2.2 cassandra.yaml file.

Options

Table: Options

Short	Long	Description
	`--version`	Show program's version number and exit.
`-h`	`--help`	Show help message and exit.
`-C`	`--color`	Always use color output.
	`--no-color`	Never use color output.
`-browser=BROWSER`		Browser used to display CQL help. BROWSER can be one of the supported browsers in: https://docs.python.org/2/library/webbrowser.html. The browser path may be followed by %s. For example, "/usr/bin /google-chrome-stable %s".
	`--ssl`	Use SSL.
`-u USERNAME`	`--username=USERNAME`	Authenticate as a user.

Short	Long	Description
`-p PASSWORD`	`--password=PASSWORD`	Authenticate using a password.
`-k KEYSPACE`	`--keyspace=KEYSPACE`	Authenticate to the given keyspace. Equivalent to issuing a USE *keyspace* command immediately after starting cqlsh.
`-f FILE`	`--file=FILE`	Execute commands from FILE, then exit.
	`--debug`	Show additional debugging information.
	`--encoding=ENCODING`	Specify a non-default encoding for output. If you are experiencing problems with unicode characters, using utf8 may fix the problem. (Default from system preferences: UTF-8)
	`--cqlshrc=CQLSHRC`	Specify an alternative cqlshrc file location.
	`--cqlversion=CQLVERSION`	Specify a particular CQL version. Default is shown on starting cqlsh.
`-e EXECUTE`	`--execute=EXECUTE`	Execute the statement and exit. Useful for saving CQL output to a file.
	`--connect-timeout=CONNECT_TIMEOUT`	Specify the connection timeout in seconds (default: 5 seconds).
	`--request-timeout=REQUEST_TIMEOUT`	Specify the request timeout in seconds (default: 10 seconds).
`-t`	`--tty`	Force tty mode (command prompt).

Using CQL commands

On startup, cqlsh shows the name of the cluster, IP address, and the port used for connection to the cqlsh utility. The cqlsh prompt initially is cqlsh>. After you specify a keyspace to use, the prompt includes the name of the keyspace. For example:

```
bin/cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.3.0 | CQL spec 3.4.0 | Native protocol v4]
Use HELP for help.
cqlsh> USE testcql;
cqlsh:testcql>
```

At the cqlsh prompt, type CQL commands. Use a semicolon to terminate a command. A new line does not terminate a command, so commands can be spread over several lines for clarity.

```
cqlsh> USE demo_cl;
cqlsh:demo_cl> SELECT * FROM demo_table
           ... WHERE id = 0;
```

If a command is sent and executed successfully, results are sent to standard output.

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)
```

The lexical structure of commands, covered earlier in this reference, includes how upper- and lower-case literals are treated in commands, when to use quotation marks in strings, and how to enter exponential notation.

Saving CQL output in a file

Using the -e option to the `cqlsh` command followed by a CQL statement, enclosed in quotation marks, accepts and executes the CQL statement. For example, to save the output of a SELECT statement to `myoutput.txt`:

```
cqlsh -e "SELECT * FROM mytable" > myoutput.txt
```

Using files as input

To execute CQL commands in a file, use the -f option and the path to the file on the operating system command line. Or, after starting cqlsh, use the `SOURCE` command and the path to the file on the cqlsh command line.

The cqlsh environment variables

The default cqlsh host and listen port may be overridden by setting the CQLSH_HOST and CQLSH_PORT environment variables. Set the CQLSH_HOST to a host name or IP address. When configured, cqlsh uses the variables instead of the default values of localhost and port 9042. A host and port number given on the command line takes precedence over configured variables.

Related information

Cassandra 3.0 cassandra.yaml

Cassandra 3.x cassandra.yaml

Creating and using the cqlshrc file

The `cqlshrc` file can pass default configuration information to cqlsh. Use the `cqlshrc` file to configure SSL encryption instead of overriding the SSL_CERTFILE environmental variables repeatedly. To help you create this file, Cassandra includes a cqlshrc.sample file. Copy this file to the hidden `.cassandra` directory in your user home folder and rename it to `cqlshrc`.

cqlshrc options

Configure the `cqlshrc` file using these options:

- [authentication] options
- [ui] options
- [cql] option
- [connection] options
- [csv] options
- [tracing] options
- [ssl] options
- [certfiles] options
- Common COPY TO and COPY FROM options
- COPY TO options
- COPY FROM options
- COPY options

[authentication] options

Note: Cassandra internal authentication must be configured before users can use the authentication options. For more information, see *Using cqlsh with SSL encryption* for your version:

- Apache Cassandra 3.0
- DataStax Distribution of Apache Cassandra 3.x
- Apache Cassandra 3.0 for Windows
- DataStax Distribution of Apache Cassandra™ 3.x for Windows

username

Authenticate as user.

password

Authenticate using password.

keyspace

Use the given keyspace. Equivalent to issuing a USE keyspace command immediately after starting cqlsh.

[ui] options

color

Always use color output.

datetimeformat

Configure the format of timestamps using Python strftime syntax.

float_precision, double_precision

Sets the number of digits displayed after the decimal point for single and double precision numbers.

Note: Increasing this to large numbers can result in unusual values.

completekey

Set the key for automatic completion of a cqlsh shell entry. Default is the tab key.

encoding

The encoding used for characters. The default is UTF8.

browser

Sets the browser for cqlsh help. If the value is not specified, cqlsh uses the default browser. Available browsers are those supported by the Python webbrowser module. For example, to use Google Chrome:

- Mac OSX: `browser = open -a /Applications/Google\ Chrome.app %s`
- Linux: `browser = /usr/bin/google-chrome-stable %s`
- Windows: `browser = C:\"Program Files (x86)"\Google\Chrome\Application\chrome.exe %s`

This setting can be overridden with the `--browser` command line option.

[cql] option

version

Sets which version of CQL to use. This should rarely be used.

[connection] options

hostname

The host for the cqlsh connection.

port

The connection port. Default: `9042` (native protocol).

port

Always connect using SSL. Default: `false`.

timeout

Configures timeout in seconds for opening new connections.

request_timeout

Configures the request timeout in seconds for executing queries. Set to `None` or the number of seconds of inactivity.

[csv] option

field_size_limit

Configures the cqlsh field size. Set to a particular field size, such as `field_size_limit = 1000000000`.

[tracing] option

max_trace_wait

The maximum number of seconds to wait for a trace to complete.

[ssl] options

certfile

The path to the cassandra certificate. See *Using cqlsh with SSL encryption* in the Cassandra documentation (links above).

validate

Optional. Default: `true`.

userkey

Must be provided when `require_client_auth=true` in cassandra.yaml.

usercert

Must be provided when `require_client_auth=true` in cassandra.yaml.

[certfiles] options

Overrides default `certfiles` in [ssl] section.

Common COPY TO and COPY FROM options

Also see the Common COPY options for TO and FROM table.

nullval

The string placeholder for null values.

header

For COPY TO, controls whether the first line in the CSV output file contains the column names.

For COPY FROM, specifies whether the first line in the CSV file contains column names.

decimalsep

Set a separator for decimal values.

thousandssep

Set a separator for thousands digit groups. Default: empty string.

boolstyle

Set a representation for boolean values for True and False. The values are case insensitive. Example: yes,no or 1,0.

thousandssep

Set the number of worker processes. Maximum value is 16.

maxattempts

Set the maximum number of attempts for errors.

reportfrequency

Set the frequency with which status is displayed, in seconds.

ratefile

Specify a file for printing output statistics.

COPY TO options

Also see the COPY TO table.

maxrequests
Set the maximum number of requests each worker process can work on in parallel.

pagesize
Set the page size for fetching results.

pagetimeout
Set the page timeout for fetching results.

begintoken
Set the minimum token string for exporting data.

endtoken
Set the maximum token string for exporting data.

maxoutputsize
Set the maximum size of the output file, measured in number of lines. When set, the output file is split into segment when the value is exceeded. "-1" sets no maximum.

encoding
The encoding used for characters. The default is UTF8.

COPY FROM options

Also see the COPY FROM table.

ingestrate
Set an approximate ingest rate in rows per second. Must be set to a greater value than chunk size.

maxrows
Set the maximum number of rows. "-1" sets no maximum.

skiprows
The number of rows to skip.

skipcols
Set a comma-separated list of column names to skip.

maxparseerrors
Set the maximum global number of parsing errors. "-1" sets no maximum.

maxinserterrors
Set the maximum global number of insert errors. "-1" sets no maximum.

errfile
Set a file to store all rows that are not imported. If no value is set, the information is stored in import_ks_table.err where <ks> is the keyspace and <table> is the table name.

maxbatchsize
Set the maximum size of an import batch.

minbatchsize
Set the minimum size of an import batch.

5

chunksize
Set the size of chunks passed to worker processes.

COPY options

Also see the COPY table.

chunksize
Set the size of chunks passed to worker processes.

ingestrate
Set an approximate ingest rate in rows per second. Must be set to a greater value than chunk size.

pagetimeout
Set the page timeout for fetching results.

CAPTURE

Captures command output and appends it to a file.

Synopsis

```
CAPTURE ('<file>' | OFF )
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

To start capturing the output of a query, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to $HOME.

Examples

```
CAPTURE '~/mydir/myfile.txt'
```

Output is not shown on the console while it is captured. Only query result output is captured. Errors and output from cqlsh-only commands still appear.

To stop capturing output and return to normal display of output, use CAPTURE OFF.

To determine the current capture state, use CAPTURE with no arguments.

CLEAR

Clears cqlsh console screen.

Synopsis

```
CLEAR | CLS
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Clears the cqlsh console screen. A CTRL+L will also clear the cqlsh console screen.

CONSISTENCY

Shows the current consistency level, or given a level, sets it.

Synopsis

```
CONSISTENCY level
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

Providing an argument to the `CONSISTENCY` command overrides the default consistency level of `ONE`, and configures the consistency level for future requests. Valid values are: `ANY`, `ONE`, `TWO`, `THREE`, `QUORUM`, `ALL`, `LOCAL_QUORUM`, `EACH_QUORUM`, `SERIAL` and `LOCAL_SERIAL`.

Running the command without an argument shows the current consistency level.

Examples

Use `CONSISTENCY` without arguments to discover the current consistency level.

```
cqlsh> CONSISTENCY
```

If you haven't changed the default, the output of the CONSISTENCY command with no arguments is:

```
Current consistency level is ONE.
```

Use `CONSISTENCY` with an argument to set the consistency level to a new value.

```
cqlsh> CONSISTENCY QUORUM
```

```
Consistency level set to QUORUM.
```

The consistency level can be set to `SERIAL` or `LOCAL_SERIAL`, but only `SELECT` queries are supported. While the consistency level is set to either of these options, `INSERT` and `UPDATE` commands will fail. See `SERIAL CONSISTENCY`SERIAL CONSISTENCY on page 124 to set the serial consistency level.

Note: Consistency level refers to the consistency level for all non-lightweight transaction reads and writes. Serial consistency level refers to the consistency level of lightweight transaction reads and writes where `IF EXISTS` and `IF NOT EXISTS` are used.

```
cqlsh> CONSISTENCY SERIAL
```

```
Consistency level set to SERIAL.
```

Attempt to INSERT some data.

```
cqlsh> INSERT INTO bar (foo) VALUES ('foo');
```

```
InvalidRequest: code=2200 [Invalid query] message="You must use conditional
 updates for serializable writes"
```

```
cqlsh> INSERT INTO bar (foo) VALUES ('foo') IF NOT EXISTS;
```

```
InvalidRequest: code=2200 [Invalid query] message="SERIAL is not supported
 as conditional update commit consistency. Use ANY if you mean
"make sure it is accepted by I don't care how many replicas commit it for
 non-SERIAL reads""
```

COPY

Imports and exports CSV (comma-separated values) data to and from Cassandra.

Synopsis

```
COPY table_name ( column, ...)
FROM ( 'file_name1', 'file_name2', ... | STDIN )
WITH option = 'value' AND ...

COPY table_name ( column , ... )
TO ( 'file_name1', 'file_name2', ... | STDOUT )
WITH option = 'value' AND ...
```

Note: COPY uses an argument of one or more comma-separated filenames or python glob expressions.

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

Using the COPY options in a WITH clause, you can change the CSV format. These tables describe the available options:

Table: Common COPY options for TO and FROM

COPY Options	Default Value	Use To
DELIMITER	comma (,)	Set the character that separates fields having newline characters in the file.
QUOTE	quotation mark (")	Set the character that encloses field values.
ESCAPE	backslash (\)	Set the character that escapes literal uses of the QUOTE character.
HEADER	false	Set true to indicate that first row of the file is a header.

COPY Options	Default Value	Use To
NULL	an empty string	Represents the absence of a value.
DATETIMEFORMAT	'%Y-%m-%d %H:%M:%S%z'	Set the time format for reading or writing CSV time data. The timestamp uses the strftime format. If not set, the default value is set to the time_format value in the cqlshrc file.
MAXATTEMPTS	5	Set the maximum number of attempts for errors.
REPORTFREQUENCY	0.25	Set the frequency with which status is displayed, in seconds.
DECIMALSEP	period (.)	Set a separator for decimal values.
THOUSANDSSEP	None	Set a separator for thousands digit groups.
BOOLSTYLE	True, False	Set a representation for boolean values for True and False. The values are case insensitive. Example: yes,no or 1,0.
NUMPROCESSES	Number of cores - 1	Set the number of worker processes. Maximum value is 16.
CONFIGFILE	None	Specify a configuration file with the same format as the cqlshrc file to set WITH options. The following sections can be specified: [copy], [copy-to], [copy-from], [copy:ks.table], [copy-to:ks.table], [copy-from:ks.table], where <ks> is the keyspace name and <table> is the tablename. Options are read from these sections, in the order specified above. Command line options always override options in configuration files. Depending on the COPY direction, only the relevant copy-from or copy-to sections are used. If no configuration file is specified, the cqlshrc file is searched instead.
RATEFILE	None	Specify a file for printing output statistics.

Table: COPY FROM options

COPY Options	Default Value	Use To
CHUNKSIZE	1,000	Set the size of chunks passed to worker processes.
INGESTRATE	100,000	Set an approximate ingest rate in rows per second. Must be set to a greater value than chunk size.
MAXBATCHSIZE	20	Set the maximum size of an import batch.
MINBATCHSIZE	2	Set the minimum size of an import batch.
MAXROWS	-1	Set the maximum number of rows. "-1" sets no maximum.
SKIPROWS	0	The number of rows to skip.
SKIPCOLS	None	Set a comma-separated list of column names to skip.
MAXPARSEERRORS	-1	Set the maximum global number of parsing errors. "-1" sets no maximum.

COPY Options	Default Value	Use To
MAXINSERTERRORS	-1	Set the maximum global number of insert errors. "-1" sets no maximum.
ERRFILE	None	Set a file to store all rows that are not imported. If no value is set, the information is stored in import_ks_table.err where <ks> is the keyspace and <table> is the table name.
TTL	3600	Set the time to live in seconds. By default, data will not expire.

Table: COPY TO options

COPY Options	Default Value	Use To
ENCODING	UTF8	Set the COPY TO command to output unicode strings.
PAGESIZE	1,000	Set the page size for fetching results.
PAGETIMEOUT	10	Set the page timeout for fetching results.
BEGINTOKEN	None	Set the minimum token string for exporting data.
ENDTOKEN	None	Set the maximum token string for exporting data.
MAXREQUESTS	6	Set the maximum number of requests each worker process can work on in parallel.
MAXOUTPUTSIZE	-1	Set the maximum size of the output file, measured in number of lines. When set, the output file is split into segment when the value is exceeded. "-1" sets no maximum.

The `ENCODING` option is available only for the `COPY TO` command. This table shows that, by default, Cassandra expects the CSV data to consist of fields separated by commas (,), records separated by line separators (a newline, \r\n), and field values enclosed in double-quotation marks (""). Also, to avoid ambiguity, escape a literal double-quotation mark using a backslash inside a string enclosed in double-quotation marks ("\""). By default, Cassandra does not expect the CSV file to have a header record on the first line that consists of the column names. `COPY TO` includes the header in the output if `HEADER=TRUE`. COPY FROM ignores the first line if `HEADER=TRUE`.

COPY FROM a CSV file

By default, when you use the `COPY FROM` command, Cassandra expects every row in the CSV input to contain the same number of columns. The number of columns in the CSV input is the same as the number of columns in the Cassandra table metadata. Cassandra assigns fields in the respective order. To apply your input data to a particular set of columns, specify the column names in parentheses after the table name.

`COPY FROM` loads rows from a CVS file in a parallel non-deterministic order. Empty data for a column is assumed by default as NULL value and will override a value; if the data is overwritten, unexpected results can occur. Data can be successfully loaded but with non-deterministic random results if there is more than one row in the CSV file with the same primary key. Having more than one row with the same primary key is not explicitly checked, so unintended results can occur.

COPY FROM is intended for importing small datasets (a few million rows or less) into Cassandra. For importing larger datasets, use the Cassandra bulk loader.

COPY TO a CSV file

For example, assume you have the following table in CQL:

```
cqlsh> SELECT * FROM test.airplanes;
```

```
 name          | mach | manufacturer | year
---------------+------+--------------+------
 P38-Lightning |  0.7 |     Lockheed | 1937
```

After inserting data into the table, you can copy the data to a CSV file in another order by specifying the column names in parentheses after the table name:

```
COPY airplanes
(name, mach, year, manufacturer)
 TO 'temp.csv'
```

Specifying the source or destination files

Specify the source file of the CSV input or the destination file of the CSV output by a file path. Alternatively, you can use the STDIN or STDOUT keywords to import from standard input and export to standard output. When using stdin, signal the end of the CSV data with a backslash and period ("\.") on a separate line. If the data is being imported into a table that already contains data, COPY FROM does not truncate the table beforehand. You can copy only a partial set of columns. Specify the entire set or a subset of column names in parentheses after the table name in the order you want to import or export them. By default, when you use the COPY TO command, Cassandra copies data to the CSV file in the order defined in the Cassandra table metadata. You can also omit listing the column names when you want to import or export all the columns in the order they appear in the source table or CSV file.

Roundtrip copying of a simple table

Copy a table to a CSV file.

1. Using CQL, create a table named airplanes and copy it to a CSV file.

```
CREATE KEYSPACE test
  WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
 'datacenter1' : 1 };

USE test;

CREATE TABLE airplanes (
  name text PRIMARY KEY,
  manufacturer ascii,
  year int,
  mach float
);

INSERT INTO airplanes
  (name, manufacturer, year, mach)
  VALUES ('P38-Lightning', 'Lockheed', 1937, 0.7);

COPY airplanes (name, manufacturer, year, mach) TO 'temp.csv';
```

```
1 rows exported in 0.004 seconds.
```

2. Clear the data from the airplanes table and import the data from the temp.csv file.

```
TRUNCATE airplanes;

COPY airplanes (name, manufacturer, year, mach) FROM 'temp.csv';
```

```
1 rows imported in 0.087 seconds.
```

Copy data from standard input to a table.

1. Enter data directly during an interactive cqlsh session, using the COPY command defaults.

```
TRUNCATE airplanes;

COPY airplanes (name, manufacturer, year, mach) FROM STDIN;
```

The output is:

```
[Use \. on a line by itself to end input]
[copy]
```

2. At the [copy] prompt, enter the following data:

```
'F-14D Super Tomcat', Grumman, 1987, 2.34
'MiG-23 Flogger', Russian-made, 1964, 2.35
'Su-27 Flanker', U.S.S.R., 1981, 2.35
\.
```

3. Query the airplanes table to see data imported from STDIN:

```
SELECT * FROM airplanes;
```

Output is:

```
 name               | manufacturer | year | mach
--------------------+------+--------------+-------------
 F-14D Super Tomcat |      Grumman | 1987 | 2.35
      P38-Lightning |     Lockheed | 1937 | 0.7
      Su-27 Flanker |     U.S.S.R. | 1981 | 2.35
```

Copying collections

Cassandra supports round-trip copying of collections to and from CSV files. To perform this example, download the sample code now.

1. Unzip the downloaded file named `cql_collections.zip`.
2. Copy/paste all the CQL commands from the `cql_collections.txt` file to the cqlsh command line.
3. Take a look at the contents of the songs table. The table contains a map of venues, a list of reviews, and a set of tags.

```
cqlsh> SELECT * FROM music.songs;
```

```
 id         |album|artist|data|reviews            |tags             |
title|venue
------------+-----+------+----+-------------------+-----------------+-----
+-----------------------------------------------------------------------
 7db1a490...| null| null |null|['hot dance music']|         {'rock'}|
 null|{'2013-09-22...': 'The Fillmore', '2013-10-01...': 'The Apple
 Barrel'}
```

```
 a3e64f8f...| null| null |null|                null|{'1973', 'blues'}|
 null|null
 8a172618...| null| null |null|                null|'2007', 'covers'}|
 null|null
```

4. Copy the music.songs table to a CSV file named `songs-20140603.csv`.

```
cqlsh> COPY music.songs to 'songs-20140603.csv';
```

```
3 rows exported in 0.006 seconds.
```

5. Check that the copy operation worked.

```
cqlsh> exit;

$ cat songs-20140603.csv
7db1a490...,,,,['hot dance music'],{'rock'},,"{'2013-09-22...': 'The
 Fillmore', '2013-10-01....': 'The Apple Barrel'}"
a3e64f8f...,,,,,"{'1973', 'blues'}",,
8a172618...,,,,,"{'2007', 'covers'}",,
```

6. Start cqlsh again, and create a table definition that matches the data in the `songs-204140603` file.

```
cqlsh> CREATE TABLE music.imported_songs (
        id uuid PRIMARY KEY,
        album text,
        artist text,
        data blob,
        reviews list<text>,
        tags set<text>,
        title text,
        venue map<timestamp, text>
       );
```

7. Copy the data from the CSV file into the imported_songs table.

```
cqlsh> COPY music.imported_songs from 'songs-20140603.csv';
```

```
3 rows imported in 0.004 seconds.
```

DESCRIBE

Provides information about the connected Cassandra cluster, or about the objects stored in the cluster.

Synopsis

```
DESCRIBE FULL ( CLUSTER | SCHEMA )
| KEYSPACES
| ( KEYSPACE keyspace_name )
| TABLES
| ( TABLE table_name )
| TYPES
| ( TYPE user_defined_type )
| FUNCTIONS
| ( FUNCTION user_defined_function )
| AGGREGATES
| ( AGGREGATE user_defined_aggregate )
| INDEX
| ( INDEX index_name )
| MATERIALIZED VIEW
```

```
| ( MATERIALIZED VIEW view_name )
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

The DESCRIBE or DESC command outputs information about the connected Cassandra cluster, or about the objects stored on in the cluster. To query the system tables directly, use SELECT.

In Linux the keyspace and table name arguments are case-sensitive and need to match the upper or lowercase names stored internally. In Windows, the keyspace and table name arguments are not case-sensitive. Use the DESCRIBE commands to list objects by their internal names. Use DESCRIBE FULL SCHEMA if you need the schema of system_* keyspaces.

DESCRIBE functions in the following ways:

DESCRIBE commands	Example	Description
DESCRIBE CLUSTER	DESCRIBE CLUSTER;	Output information about the connected Cassandra cluster. Cluster name, partitioner, and snitch are output. For non-system keyspace, the endpoint-range ownership information is also shown.
DESCRIBE KEYSPACES	DESCRIBE KEYSPACES;	Output a list of all keyspace names.
DESCRIBE KEYSPACE <keyspace_name>	DESCRIBE KEYSPACE cycling;	Output the CQL command that could be used to recreate the given keyspace, and the objects in it, such as the tables, types and functions.
DESCRIBE [FULL] SCHEMA	DESCRIBE SCHEMA;	Output the CQL command that could be used to recreate the entire non-system schema. Use the FULL option to also include system keyspaces.
DESCRIBE TABLES	DESCRIBE TABLES;	Output a list of all tables in the current keyspace, or in all keyspaces if there is not current keyspace.
DESCRIBE TABLE <keyspace>.<table_name>	DESCRIBE TABLE upcoming_calendar;	Output the CQL command that could be used to recreate the given table.
DESCRIBE INDEX <keyspace>.<index_name>	DESCRIBE INDEX team_entry;	Output the CQL command that could be used to recreate the given index.
DESCRIBE TYPES	DESCRIBE TYPES;	Output list of all user-defined types in the current keyspace, or in all keyspaces if there is no current keyspace.
DESCRIBE TYPE <keyspace>.<type_name>	DESCRIBE TYPE basic_info;	Output the CQL command that can be used to recreate the given user-defined type.
DESCRIBE FUNCTIONS	DESCRIBE FUNCTIONS;	Output names of all user-defined functions in the given keyspace, or in all keyspaces if there is no current keyspace.

DESCRIBE commands	Example	Description
DESCRIBE FUNCTION <keyspace_name>.<function_name>	DESCRIBE FUNCTION cycling.myFunction;	Output the CQL command that could be used to recreate the given user-defined function.
DESCRIBE AGGREGATES	DESCRIBE AGGREGATES;	Output a list of all user-defined aggregates in the given keyspace, or in all keyspaces if there is no current keyspace.
DESCRIBE AGGREGATE <keyspace_name>.<aggregate_name>	DESCRIBE AGGREGATE cycling.myAggregate;	Output the CQL command that could be used to recreate the given user-defined aggregate.
DESCRIBE MATERIALIZED VIEW <view_name>	DESCRIBE MATERIALIZED VIEW cyclistsByAge;	Output the CQL command that could be used to recreate the given materialized view.
DESCRIBE <objectname>	DESCRIBE	Output CQL commands that could be used to recreate the entire object schema, where object can be either a keyspace or a table or an index or a materialized view (in this order).

EXPAND

Formats the output of a query vertically.

Synopsis

```
EXPAND ( ON | OFF )
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

This command lists the contents of each row of a table vertically, providing a more convenient way to read long rows of data than the default horizontal format. You scroll down to see more of the row instead of scrolling to the right. Each column name appears on a separate line in column one and the values appear in column two.

Sample output of EXPAND ON is:

```
cqlsh:my_ks> EXPAND ON
             Now printing expanded output
cqlsh:my_ks> SELECT * FROM users;
```

```
@ Row 1
------------+-----------------------------------------------
 userid     | samwise
 emails     | {'samwise@gmail.com', 's@gamgee.com'}
 first_name | Samwise
 last_name  | Gamgee
```

```
 todo       | {'2013-09-22 12:01:00-0700': 'plant trees'}
 top_scores | null

@ Row 2
------------+--------------------------------------------
 userid     | frodo
 emails     | {'baggins@gmail.com', 'f@baggins.com'}
 first_name | Frodo
 last_name  | Baggins
 todo       | {'2012-10-02 12:00:00-0700': 'throw my precious into mount
 doom'}
 top_scores | null

(2 rows)
```

EXIT

Terminates cqlsh.

Synopsis

```
EXIT | QUIT
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

LOGIN

Synopsis

```
cqlsh> LOGIN user_name password
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Use this command to change login information without requiring `cqlsh` to restart. Login using a specified username. If the password is specified, it will be used. Otherwise, you will be prompted to enter the password.

Examples

Login as the user cutie with the password patootie.

```
LOGIN cutie patootie
```

PAGING

Enables or disables query paging.

Synopsis

```
PAGING ( ON | OFF )
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

In Cassandra 2.1.1, you can use query paging in cqlsh. Paging provides the output of queries in 100-line chunks followed by the *more* prompt. To get the next chunk, press the space bar. Turning paging on enables query paging for all further queries. Using no ON or OFF argument with the command shows the current query paging status.

SERIAL CONSISTENCY

Sets the current serial consistency level that will be used for lightweight transactions. The serial consistency level is different from the consistency level for non-lightweight transaction commands.

Synopsis

```
SERIAL CONSISTENCY level
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

Providing an argument to the `SERIAL CONSISTENCY` command sets the serial consistency level. Valid values are: `SERIAL` and `LOCAL_SERIAL`.

Examples

Use `SERIAL CONSISTENCY` without an argument to see the serial consistency level.

```
cqlsh> SERIAL CONSISTENCY
```

```
Current serial consistency level set to SERIAL.
```

Set the serial consistency level with a value.

```
cqlsh> SERIAL CONSISTENCY LOCAL_SERIAL
```

```
Serial consistency level set to LOCAL_SERIAL.
```

Attempt to INSERT some data.

```
cqlsh> INSERT INTO bar (foo) VALUES ('foo') IF NOT EXISTS;
```

```
[APPLIED] | foo
------+----
FALSE | foo
```

A value of `FALSE` is returned if the value of foo already exists.

SHOW

Shows the Cassandra version, host, or tracing information for the current cqlsh client session.

Synopsis

```
SHOW VERSION
| HOST
| SESSION tracing_session_id
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

A SHOW command displays this information about the current cqlsh client session:

- The version and build number of the connected Cassandra instance, as well as the CQL specification version for cqlsh and the native protocol version used by the connected Cassandra instance.
- The host information of the Cassandra node that the cqlsh session is currently connected to.
- Tracing information for the current cqlsh session.

The SHOW SESSION command retrieves tracing session information, which is available for 24 hours. After that time, the tracing information time-to-live expires.

These examples show how to use the commands.

```
cqlsh:my_ks> SHOW version
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.3 | Native protocol v3]
```

```
cqlsh:my_ks> SHOW host
Connected to Test Cluster at 127.0.0.1:9042.

cqlsh:my_ks> SHOW SESSION d0321c90-508e-11e3-8c7b-73ded3cb6170
```

Sample output of SHOW SESSION is:

```
Tracing session: d0321c90-508e-11e3-8c7b-73ded3cb6170

 activity
                                                                                | timestamp
 | source    | source_elapsed
-----------------------------------------------------------------------------------------------
+-------------+-----------+----------------

                                            execute_cql3_query | 12:19:52,372
 | 127.0.0.1 |              0
 Parsing CREATE TABLE emp (\n  empID int,\n  deptID int,\n  first_name
 varchar,\n  last_name varchar,\n  PRIMARY KEY (empID, deptID)\n); |
 12:19:52,372 | 127.0.0.1 |            153

                                              Request complete | 12:19:52,372
 | 127.0.0.1 |            650
. . .
```

SOURCE

Executes a file containing CQL statements.

Synopsis

```
SOURCE 'file'
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

To execute the contents of a file, specify the path of the file relative to the current directory. Enclose the file name in single quotation marks. The shorthand notation in this example is supported for referring to $HOME:

Examples

```
SOURCE '~/mydir/myfile.txt'
```

The output for each statement, if there is any, appears in turn, including any error messages. Errors do not abort execution of the file.

Alternatively, use the --file option to execute a file while starting CQL.

TRACING

Enables or disables request tracing.

Synopsis

```
TRACING ( ON | OFF )
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

Description

To turn tracing read/write requests on or off, use the TRACING command. After turning on tracing, database activity creates output that may help you understand Cassandra internal operations and troubleshoot performance problems. For example, using the tracing tutorial you can see how different consistency levels affect operations. Some tracing messages refer to internals of the database that can be provided those troubleshooting Cassandra issues.

For 24 hours, Cassandra saves the tracing information in sessions and events tables in the `system_traces` keyspace, which you query when using probabilistic tracing. For information about probabilistic tracing, see Cassandra 3.0 documentation or Cassandra 2.2 documentation.

```
CREATE TABLE system_traces.sessions (
    session_id uuid PRIMARY KEY,
    client inet,
    command text,
    coordinator inet,
    duration int,
    parameters map<text, text>,
    request text,
    started_at timestamp)

CREATE TABLE system_traces.events (
    session_id uuid,
    event_id timeuuid,
    activity text,
    source inet,
    source_elapsed int,
    thread text,
    PRIMARY KEY (session_id, event_id)
```

The source_elapsed column stores the elapsed time in microseconds before the event occurred on the source node.

To keep tracing information, copy the data in sessions and event tables to another location. Alternatively, use the tracing session id to retrieve session information using SHOW SESSION. Tracing session information expires after one day.

Tracing a write request

This example shows tracing activity on a 3-node cluster created by ccm on Mac OSX. Using a keyspace that has a replication factor of 3 and an employee table similar to the one in "Using a compound primary key on page 18," the tracing shows that the coordinator performs the following actions:

- Identifies the target nodes for replication of the row.
- Writes the row to the commitlog and memtable.
- Confirms completion of the request.

```
TRACING ON

INSERT INTO emp (empID, deptID, first_name, last_name)
  VALUES (104, 15, 'jane', 'smith');
```

Cassandra provides a description of each step it takes to satisfy the request, the names of nodes that are affected, the time for each step, and the total time for the request.

```
Tracing session: 740b9c10-3506-11e2-0000-fe8ebeead9ff

 activity                                  | timestamp    | source    |
 source_elapsed
-------------------------------------------+--------------+-----------
+----------------
                    execute_cql3_query | 16:41:00,754 | 127.0.0.1 |
   0
                     Parsing statement | 16:41:00,754 | 127.0.0.1 |
  48
                    Preparing statement | 16:41:00,755 | 127.0.0.1 |
 658
   Determining replicas for mutation | 16:41:00,755 | 127.0.0.1 |
 979
    Message received from /127.0.0.1 | 16:41:00,756 | 127.0.0.3 |
  37
      Acquiring switchLock read lock | 16:41:00,756 | 127.0.0.1 |
 1848
       Sending message to /127.0.0.3 | 16:41:00,756 | 127.0.0.1 |
 1853
              Appending to commitlog | 16:41:00,756 | 127.0.0.1 |
 1891
       Sending message to /127.0.0.2 | 16:41:00,756 | 127.0.0.1 |
 1911
              Adding to emp memtable | 16:41:00,756 | 127.0.0.1 |
 1997
      Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.3 |
 395
    Message received from /127.0.0.1 | 16:41:00,757 | 127.0.0.2 |
  42
              Appending to commitlog | 16:41:00,757 | 127.0.0.3 |
 432
      Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.2 |
 168
              Adding to emp memtable | 16:41:00,757 | 127.0.0.3 |
 522
              Appending to commitlog | 16:41:00,757 | 127.0.0.2 |
 211
              Adding to emp memtable | 16:41:00,757 | 127.0.0.2 |
 359
    Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |
 1282
    Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |
 1024
       Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |
 1469
       Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |
 1179
    Message received from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |
 10966
```

```
    Message received from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |
 10966
 Processing response from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |
 11063
 Processing response from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |
 11066
                    Request complete | 16:41:00,765 | 127.0.0.1 |
 11139
```

Tracing a sequential scan

Due to the log structured design of Cassandra, a single row is spread across multiple SSTables. Reading one row involves reading pieces from multiple SSTables, as shown by this trace of a request to read the employee table, which was pre-loaded with 10 rows of data.

```
SELECT * FROM emp;
```

Output is:

```
 empid | deptid | first_name | last_name
-------+--------+------------+-----------
   110 |     16 |      naoko |     murai
   105 |     15 |       john |     smith
   111 |     15 |       jane |     thath
   113 |     15 |       lisa |     amato
   112 |     20 |       mike |     burns
   107 |     15 |     sukhit |       ran
   108 |     16 |        tom |     brown
   109 |     18 |        ann |     green
   104 |     15 |       jane |     smith
   106 |     15 |        bob |     jones
```

The tracing output of this read request looks something like this (a few rows have been truncated to fit on this page):

```
Tracing session: bf5163e0-350f-11e2-0000-fe8ebeead9ff

 activity                                                  | timestamp    | source
 | source_elapsed
-----------------------------------------------------------+--------------+-----------
+----------------
                                 execute_cql3_query | 17:47:32,511 | 127.0.0.1
 |              0
                                  Parsing statement | 17:47:32,511 | 127.0.0.1
 |             47
                                Preparing statement | 17:47:32,511 | 127.0.0.1
 |            249
                     Determining replicas to query | 17:47:32,511 | 127.0.0.1
 |            383
                     Sending message to /127.0.0.2 | 17:47:32,512 | 127.0.0.1
 |            883
                   Message received from /127.0.0.1 | 17:47:32,512 | 127.0.0.2
 |             33
  Executing seq scan across 0 sstables for . . . | 17:47:32,513 | 127.0.0.2
 |            670
                   Read 1 live cells and 0 tombstoned | 17:47:32,513 | 127.0.0.2
 |            964
                   Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2
 |           1268
```

```
               Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2
 |           1502
               Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2
 |           1673
                      Scanned 4 rows and matched 4 | 17:47:32,514 | 127.0.0.2
 |           1721
                  Enqueuing response to /127.0.0.1 | 17:47:32,514 | 127.0.0.2
 |           1742
                     Sending message to /127.0.0.1 | 17:47:32,514 | 127.0.0.2
 |           1852
                  Message received from /127.0.0.2 | 17:47:32,515 | 127.0.0.1
 |           3776
                 Processing response from /127.0.0.2 | 17:47:32,515 | 127.0.0.1
 |           3900
                     Sending message to /127.0.0.2 | 17:47:32,665 | 127.0.0.1
 |         153535
                  Message received from /127.0.0.1 | 17:47:32,665 | 127.0.0.2
 |             44
 Executing seq scan across 0 sstables for . . . | 17:47:32,666 | 127.0.0.2
 |           1068
               Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2
 |           1454
               Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2
 |           1640
                      Scanned 2 rows and matched 2 | 17:47:32,667 | 127.0.0.2
 |           1694
                  Enqueuing response to /127.0.0.1 | 17:47:32,667 | 127.0.0.2
 |           1722
                     Sending message to /127.0.0.1 | 17:47:32,667 | 127.0.0.2
 |           1825
                  Message received from /127.0.0.2 | 17:47:32,668 | 127.0.0.1
 |         156454
                 Processing response from /127.0.0.2 | 17:47:32,668 | 127.0.0.1
 |         156610
 Executing seq scan across 0 sstables for . . . | 17:47:32,669 | 127.0.0.1
 |         157387
               Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
 |         157729
               Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
 |         157904
               Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
 |         158054
               Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1
 |         158217
                      Scanned 4 rows and matched 4 | 17:47:32,669 | 127.0.0.1
 |         158270
                                  Request complete | 17:47:32,670 | 127.0.0.1
 |         159525
```

The sequential scan across the cluster shows:

- The first scan found 4 rows on node 2.
- The second scan found 2 more rows found on node 2.
- The third scan found the 4 rows on node 1.

Related information

Cassandra 3.0 probabilistic tracing

Cassandra 2.2 probabilistic tracing

CQL commands

ALTER KEYSPACE

Change property values of a keyspace.

Synopsis

```
ALTER ( KEYSPACE | SCHEMA ) keyspace_name
WITH REPLICATION = map
| ( WITH DURABLE_WRITES = ( true | false ))
AND ( DURABLE_WRITES = ( true | false))
```

map is a map collection, a JSON-style array of literals:

{ literal : literal *, literal : literal, ...* }

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

ALTER KEYSPACE changes the map that defines the replica placement strategy and/or the DURABLE_WRITES value. You can also use the alias ALTER SCHEMA. Use these properties and values to construct the map. To set the replica placement strategy, construct a map of properties and values, as shown in the table of map properties on the CREATE KEYSPACE reference page. CQL property map keys must be lower case. For example, class and replication_factor are correct.

You cannot change the name of the keyspace.

Example

Change the definition of the mykeyspace to use the NetworkTopologyStrategy in a single data center. Use the default data center name in Cassandra and a replication factor of 3.

```
ALTER KEYSPACE "Excalibur" WITH REPLICATION =
  { 'class' : 'NetworkTopologyStrategy', 'datacenter1' : 3 };
```

ALTER MATERIALIZED VIEW

Modify the properties of a materialized view in Cassandra 3.0 and later.

Synopsis

```
ALTER MATERIALIZED VIEW [ keyspace_name. ] view_name
( WITH property [ AND property ] [ . . . ] ]
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

Use `ALTER MATERIALIZED VIEW` to change view properties. The statement returns no results.

Parameters

keyspace_name

To alter a materialized view in a keyspace other than the current keyspace, put the keyspace name in front of the name of the materialized view , followed by a period.

view_name

The name of the new materialized view.

property

A CQL table property and value, as in the following examples:

```
compaction = { 'class' : 'LeveledCompactionStrategy' }

read_repair_chance = 1.0

speculative_retry = '10ms'
```

Examples

Modifying table properties

For an overview of properties that apply to materialized views, see Table properties on page 97.

- To change the table properties established during creation of the materialized view, use the WITH keyword followed by a n expression that includes the property name and value.
- To change multiple properties, use AND:

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age WITH comment = 'A most
 excellent and useful view'
  AND bloom_filter_fp_chance = 0.02;
```

- If the value for a property is a text string, enclose it in single quotation marks.

Modifying compression and compaction

Use a property map to specify new properties for compression or compaction.

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age WITH compression =
  { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 };
```

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age
  WITH compaction = {'class': 'SizeTieredCompactionStrategy',
 'cold_reads_to_omit': 0.05};
```

Note: Cassandra 3.0 and later no longer support the `cold_reads_to_omit` property for SizeTieredCompactionStrategy.

Changing caching

You can create and change caching properties using a property map.

This example changes the keys property to `NONE` (the default is `ALL`) and changes the rows_per_partition property to `15`.

```
ALTER MATERIALIZED VIEW cycling.cyclist_by_age WITH caching = { 'keys' :
 'NONE', 'rows_per_partition' : '15' };
```

Viewing current materialized view properties

Use DESCRIBE MATERIALIZED VIEW to see all current properties.

```
DESCRIBE MATERIALIZED VIEW cycling.cyclist_by_age

CREATE MATERIALIZED VIEW cycling.cyclist_by_age AS
    SELECT age, cid, birthday, country, name
    FROM cycling.cyclist_mv
    WHERE age IS NOT NULL AND cid IS NOT NULL
    PRIMARY KEY (age, cid)
    WITH CLUSTERING ORDER BY (cid ASC)
    AND bloom_filter_fp_chance = 0.02
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = 'A most excellent and useful view'
    AND compaction = {'class':
 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class':
 'org.apache.cassandra.io.compress.DeflateCompressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

Related information

Creating a materialized view on page 21

CREATE MATERIALIZED VIEW on page 153

DROP MATERIALIZED VIEW on page 171

ALTER ROLE

Alter existing role options.

Synopsis

```
ALTER ROLE role_name
WITH PASSWORD = [ 'password'
```

```
[AND | WITH] LOGIN = true | false ]
[AND | WITH] SUPERUSER = true | false ]
[ OPTIONS = <map_literal> ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

The role used to alter roles must have ALTER permission, either directly or on ALL ROLES. To alter a superuser role, a superuser role must be used. To prevent disabling all superusers, superusers cannot change their own superuser status. Ordinary roles can change only their own password.

Enclose the role name in single quotation marks if it contains non-alphanumeric characters. Enclose the password in single quotation marks.

Examples

```
ALTER ROLE coach WITH PASSWORD 'bestTeam';
```

ALTER TABLE

Modify the column metadata of a table.

You can also use the alias **ALTER COLUMNFAMILY**.

Synopsis

```
ALTER TABLE keyspace_name. table_name instruction;
```

Parameters

instruction is:

```
( TYPE   cql_type
| ADD column_name cql_type
| DROP column_name
|  RENAME column_name TO column_name
|  WITH property [ AND property ] . . . } )
```

cql_type is compatible with the original type and is a CQL type, not a collection or counter. Exception: You cannot change the type of an existing column to a map or collection, but when you ADD a new column, you can define it as a map or collection, If the table is a counter, you can ADD a column of type counter.

property is a CQL table property and value, such as `speculative_retry = '10ms'`. Enclose the value for a string property in single quotation marks.

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Use `ALTER TABLE` to manipulate the table metadata. Do this to change the datatype of a columns, add new columns, drop existing columns, and change table properties. The command returns no results.

Start the command with the keywords `ALTER TABLE`, followed by the table name, followed by the instruction: ALTER. ADD, DROP, RENAME, or WITH. See the following sections for the information each instruction requires.

You can qualify table name by prepending the name of its keyspace. For example, to specify the `teams` table in the `cycling` keyspace:

```
ALTER TABLE cycling.teams ALTER ID TYPE uuid;
```

Changing the type of a column

You can only change the datatype of a column when the original datatype of the column is compatible with the type you are changing to. Examples:

- You can change a column of type ascii to type text
- You cannot change a textor varchar column to type ascii because not every UTF8 string is type ascii.
- You can convert a text column to type blob
- You cannot change a blob column to type text because not every blob is a UTF string.

More examples: change this type of the `bio` column in the `users` table from ascii to text, and then from text to blob.

```
CREATE TABLE users (
  user_name varchar PRIMARY KEY,
  bio ascii,
  );
ALTER TABLE users ALTER bio TYPE text;
ALTER TABLE users ALTER bio TYPE blob;
```

You can only change the datatype of a column if the column already exists in current rows. When a column's datatype changes, the bytes stored in values for that column remain unchanged. If existing data cannot be deserialized to conform to the new datatype, your CQL driver or interface returns errors.

The following datatype changes are not supported:

- Changing the type of a clustering column
- Changing the type of a column on which an index is defined

Altering the type of a column after inserting data can confuse CQL drivers/tools if the new type is incompatible with the data just inserted.

Adding a column

To add a column (other than a column of a collection type) to a table, use `ALTER TABLE` and the `ADD` instruction as follows:

```
ALTER TABLE cycling.cyclist_races ADD firstname text;
```

To add a column of a collection type:

```
ALTER TABLE cycling.upcoming_calendar ADD events list<text>;
```

This operation does not validate the existing data.

You cannot use the `ADD` command to add:

- A column with the same name as an existing column
- A static column

Dropping a column

To remove (*drop*) a column from the table, use `ALTER TABLE` and the `DROP` instruction.

```
ALTER TABLE cycling.basic_info DROP birth_year;
```

`ALTER DROP` removes the column from the table definition, removes data included in that column, and eventually reclaims the space formerly used by the column. The column becomes unavailable for queries immediately after it is dropped. The actual data removal occurs during compaction. The column data is not included in SSTables in the future. To force the removal of dropped columns before compaction occurs, use the ALTER TABLE command to update the metadata, followed by nodetool upgradesstables to register the drop.

More tips about dropping columns:

- If you drop a column then re-add it, a SELECT on this column does not return the values written before the column was dropped.
- Do not re-add a dropped column to a table if it contains timestamps that are generated by the client, not by Cassandra's write time facility.
- You cannot drop columns from tables defined with the COMPACT STORAGE option.

Renaming a column

The main purpose of `RENAME` is to change the names of CQL-generated primary key and column names that are missing from a legacy table. The following restrictions apply to the `RENAME` operation:

- You can only rename clustering columns, which are part of the primary key.
- You cannot rename the partition key.
- You can index a renamed column.
- You cannot rename a column if an index has been created on it.
- You cannot rename a static column (since you cannot use a static column in the table's primary key).

Modifying table properties

To change the table storage properties set when the table was created, use one of the following formats:

- Use `ALTER TABLE` and a `WITH` instruction that introduces the property name and value
- Use `ALTER TABLE WITH` and a property map, as shown in the next section on compression and compaction.

This example uses the `WITH` instruction to modify the read_repair_chance property, which configures read repair for tables that use for a non-quorum consistency.

This example also shows how to change multiple properties using AND:

```
ALTER TABLE cyclist_mv
  WITH comment = 'ID, name, birthdate and country'
  AND read_repair_chance = 0.2;
```

Enclose a text property value in single quotation marks. You cannot modify properties of a table that uses compact storage.

Modifying compression and compaction

Use a property map to alter a table's compression or compaction setting.

```
ALTER TABLE cycling_comments WITH compression =
  { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 };

ALTER TABLE mykeyspace.mytable
  WITH compaction = {'class': 'SizeTieredCompactionStrategy',
 'cold_reads_to_omit': 0.05};
```

To change the values of the caching property: for example, change the keys option from ALL, the default, to NONE and change the rows_per_partition to 15.

CAUTION: If you change the compaction strategy of a table with existing data, Cassandra rewrites all existing SSTables according to the new strategy. This can take hours, which can be a major problem for a production system. For strategies to minimize this disruption, see How to change Cassandra compaction strategy on a production cluster and Impact of Changing Compaction Strategy.

Changing caching

Create and change the caching options using a property map.

```
ALTER TABLE users WITH caching = { 'keys' : 'NONE', 'rows_per_partition' :
 '15' };
```

Next, change just the rows_per_partition to 25.

```
//Cassandra 2.1 only

ALTER TABLE users WITH caching = { 'rows_per_partition' : '25' };
```

Finally, use DESCRIBE to view the table definition.

```
//Cassandra 2.1 only

DESCRIBE TABLE users;

CREATE TABLE mykeyspace.users (
    user_name text PRIMARY KEY,
    bio blob
) WITH bloom_filter_fp_chance = 0.01
    AND caching = '{"keys":"NONE", "rows_per_partition":"25"}'
    AND comment = ''
    AND compaction = {'min_threshold': '4', 'class':
 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
 'max_threshold': '32'}
    AND compression = {'sstable_compression':
 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
```

```
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.1
    AND speculative_retry = '99.0PERCENTILE';
```

In Cassandra 2.0.x, you alter the caching options using the WITH directive.

```
//Cassandra 2.0.x only
      ALTER TABLE users WITH caching = "keys_only";
```

Important: Use row caching in Cassandra 2.0.x with caution.

ALTER TYPE

Modify a user-defined type. Cassandra 2.1 and later.

Synopsis

```
ALTER TYPE field_name instruction
```

instruction is:

```
ALTER field_name TYPE new_type
| ( ADD field_name new_type )
| ( RENAME field_name TO field_name )
   ( AND field_name TO field_name ) ...
```

field_name is an arbitrary identifier for the field.

new_type is an identifier other than the reserved type names.

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

ALTER TYPE can change a user-defined type in the following ways:

- Change the type of an existing field.
- Append a new field to an existing type.
- Rename a field defined by the type.

First, after the ALTER TYPE keywords, specify the name of the user-defined type to be changed, followed by the type of change: ALTER, ADD, or RENAME. Next, provide the rest of the needed information, as explained in the following sections.

Changing the type of a field

To change the type of a field, the field must already exist in type definition and its type should be compatible with the new typeCQL type compatibility on page 90. Consider the compatibility list before changing a data type. It is best to careful choose the data type for each column at the time of table creation.

This example shows changing the type of the model field from ascii to text and then to blob.

```
CREATE TYPE version (
  model ascii,
  version_number int
);
```

```
ALTER TYPE version ALTER model TYPE text;
ALTER TYPE version ALTER model TYPE blob;
```

Clustering column data types are very restricted in the options for alteration, because clustering column data is used to order rows in tables. Indexed columns cannot be altered.

Adding a field to a type

To add a new field to a type, use ALTER TYPE and the ADD keyword.

```
ALTER TYPE version ADD release_date timestamp;
```

To add a collection map field called point_release in this example that represents the release date and decimal designator, use this syntax:

```
ALTER TYPE version ADD point_release map<timestamp, decimal>;
```

Renaming a field of a type

To change the name of a field of a user-defined type, use the RENAME old_name TO new_name syntax. You can't use different keyspaces prefixes for the old and new names. Make multiple changes to field names of a type by appending AND old_name TO new_name to the command.

```
ALTER TYPE version RENAME model TO sku;
ALTER TYPE version RENAME sku TO model AND version_number TO num
```

ALTER USER

Alter existing user options.

Notice: `ALTER USER` is supported for backwards compatibility. Authentication and authorization for Cassandra 2.2 and later are based on `ROLES`, and `ALTER ROLE` should be used.

Synopsis

```
ALTER USER user_name
WITH PASSWORD 'password'
[ SUPERUSER | NOSUPERUSER ]
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements

Markup	Indicates
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

Superusers can change a user's password or superuser status. To prevent disabling all superusers, superusers cannot change their own superuser status. Ordinary users can change only their own password. Enclose the user name in single quotation marks if it contains non-alphanumeric characters. Enclose the password in single quotation marks. See CREATE ROLE for more information about SUPERUSER and NOSUPERUSER.

Examples

Alter a user's password:

```
ALTER USER moss WITH PASSWORD 'bestReceiver';
```

Alter a user to make that a superuser:

```
ALTER USER moss SUPERUSER;
```

BATCH

Write multiple DML statements.

Synopsis

Cassandra 2.1 and later:

```
BEGIN [ UNLOGGED ] BATCH
  [ USING TIMESTAMP timestamp ]
  dml_statement;
  [ dml_statement; ]
  [ ... ]
APPLY BATCH;
```

Cassandra 2.0.x:

```
BEGIN [ { UNLOGGED | COUNTER } ] BATCH
   [ USING TIMESTAMP timestamp ]
 dml_statement;
  [ dml_statement; ]
  [ ... ]
APPLY BATCH;
```

dml_statement can be statement using one of the following CQL commands:

- **INSERT**
- **UPDATE**
- **DELETE**

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

A BATCH statement combines multiple data modification language (DML) statements (INSERT, UPDATE, DELETE) into a single logical operation, and can set a client-supplied timestamp for all columns written by the statements in the batch. Batching multiple statements saves network exchanges between client and server and between server coordinator and replicas. Cassandra spreads requests across nearby nodes as much as possible to optimize performance. Batch statements have an uneven workload, and may access many nodes or a single node repeatedly.

Using a batch statement is not a good way to optimize performance — for details, see Using and misusing batches. For a discussion about the fastest way to load data, see Cassandra: Batch loading without the Batch keyword. Instead, using a batch is a good way to synchronize data to tables.

Batches are logged by default. Running a batch with logging enabled ensures that if any of the batch succeeds, all of it will. Cassandra first writes the serialized batch to the batchlog system table that consumes the serialized batch as blob data. After Cassandra has successfully written and persisted (or hinted) the rows in the batch, it removes the batchlog data. There is a performance penalty associated with the batchlog, as it is written to two other nodes. If you do not want to incur this penalty, run the batch operation without using the batchlog table by using the `UNLOGGED` keyword.

Although a logged batch enforces atomicity (that is, it guarantees if all DML statements in the batch succeed or none do), Cassandra does no other transactional enforcement at the batch level. For example, there is no batch isolation. Clients are able to read the first updated rows from the batch, while other rows are still being updated on the server. Within a batch, however, transactional row updates within a partition key are isolated: clients cannot read a partial update.

Statement order does not matter within a batch. Cassandra applies the same timestamp to all rows. Use client-supplied timestamps to achieve a particular order.

Using a timestamp

BATCH allows you to set a client-supplied timestamp, an integer, in the USING TIMESTAMP clause. If you add USING TIMESTAMP but do not supply an integer value, Cassandra applies the time of the insertion (in microseconds).

The timestamp applies to all statements in the batch. You can apply a timestamp by adding a USING clause to any of the DML statements in the batch.

For example, specify a timestamp in an INSERT statement:

```
cqlsh> BEGIN BATCH
  INSERT INTO purchases (user, balance) VALUES ('user1', -8) USING TIMESTAMP
 19998889022757000;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 1, 8, 'burrito', false);
```

```
APPLY BATCH;
```

Verify that balance column has the client-provided timestamp:

```
cqlsh> SELECT balance, WRITETIME(balance) FROM PURCHASES;
```

```
 balance | writetime_balance
---------+-------------------
      -8 | 19998889022757000
```

Warning:

If any DML statement in the batch uses compare-and-set (CAS) logic, Cassandra cannot set a timestamp. Example: a batch containing the following statement will not run:

```
cqlsh> INSERT INTO users (id, lastname) VALUES (999, 'Sparrow')  IF NOT
 EXISTS
```

Batching conditional updates

You can batch conditional updates introduced as lightweight transactions. However, a batch containing conditional updates can only operate within a single partition, because the underlying Paxos implementation only works at partition-level granularity. If one statement in a batch is a conditional update, the conditional logic must return true, or the entire batch fails. If the batch contains two or more conditional updates, all the conditions must return true, or the entire batch fails. This example shows batching of conditional updates:

The statements for inserting values into purchase records use the IF conditional clause.

```
cqlsh> BEGIN BATCH
  INSERT INTO purchases (user, balance) VALUES ('user1', -8) IF NOT EXISTS;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;

cqlsh> BEGIN BATCH
  UPDATE purchases SET balance = -208 WHERE user='user1' IF balance = -8;
  INSERT INTO purchases (user, expense_id, amount, description, paid)
    VALUES ('user1', 2, 200, 'hotel room', false);
APPLY BATCH;
```

A continuation of this example shows how to use a static column with conditional updates in batch.

Batching counter updates

A batch of counters should use the `UNLOGGED` option because, unlike other writes in Cassandra, a counter update is not an idempotent operation.

In Cassandra 2.1 and later, use `BEGIN COUNTER BATCH` in a batch statement for batched counter updates.

Cassandra 2.1 Example

```
cqlsh> BEGIN UNLOGGED BATCH
  UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
  UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

Cassandra 2.0 Example

```
cqlsh> BEGIN COUNTER BATCH
  UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
  UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

CREATE AGGREGATE

Synopsis

```
CREATE [OR REPLACE] AGGREGATE [IF NOT EXISTS] <keyspace>.aggregate-name
 ( <arg-name> <arg-type> )
SFUNC <state-function-name>
STYPE <type>
FINALFUNC <final-function-name>
INITCOND ( <value>,<value> )
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

A user-defined aggregate function can be created using user-defined functions. Returns a single value that is the aggregate value over all rows returned in query.

Examples

Create an aggregate that calculates a user-defined average.

```
cqlsh> CREATE OR REPLACE AGGREGATE IF NOT EXISTS average ( int ) SFUNC
 avgState STYPE tuple<int,bigint> FINALFUNC avgFinal INITCOND (0,0);
```

CREATE CUSTOM INDEX (SASI)

In Cassandra 3.4 and later, a new type of index, the SSTable Attached Secondary Index (SASI). SASI indexing and querying improves on the existing secondary index implementation with superior performance for queries that previously required the use of `ALLOW FILTERING`. SASI uses significantly using fewer memory, disk, and CPU resources. It enables querying with PREFIX and CONTAINS on strings, similar to the SQL implementation of `LIKE = "foo*"` or `LIKE = "*foo*"`.

A SASI index can be created on a single column of a table. For more information about SASI, see Using SASI.

Synopsis

```
CREATE CUSTOM INDEX IF NOT EXISTS index_name
ON keyspace_name.table_name  ( column_name )
```

```
USING 'org.apache.cassandra.index.sasi.SASIIndex' (WITH OPTIONS = map)
```

index_name is an identifier, enclosed or not enclosed in double quotation marks, excluding reserved words.

`WITH OPTIONS` identifies the characteristics of the SASI index.

KEYWORD	USE
mode	PREFIX, CONTAINS, or SPARSE. Default: PREFIX
analyzed	true or false. Indicates whether literal column is analyzed using the specified analyzer.
analyzer_class	Specifies an analyzer class. Two classes are available: `org.apache.cassandra.index.sasi.analyzer.NonToke` and `org.apache.cassandra.index.sasi.analyzer.Standar`
is_literal	Designates a column as literal.
max_compaction_flush_memory_in_mb	

OPTION	ANALYZER	USE
tokenization_enable_stemming	Standard tokenizer	Enables stemming: indexing and querying recognize that "stems", "stemmer", "stemming", "stemmed" as based on "stem". Default is to ignore stemming, setting: `false`.
tokenization_skip_stop_words	Standard tokenizer	Indexing ignores "stop words" like "and" and "the". Default is to ignore stop terms, setting: `true`.
tokenization_locale	Standard tokenizer	Default: English, setting: `en`. List of localization codes
tokenization_normalize_lowercase	Standard tokenizer	Index all strings as lowercase. Default is to be case-sensitive, setting `false`.
tokenization_normalize_uppercase	Standard tokenizer	Index all strings as uppercase. Default: `false`.
normalize_lowercase	Non-tokenizing	Index all strings as lowercase. Default: `false`.
normalize_uppercase	Non-tokenizing	Index all strings as uppercase. Default: `false`.
case_sensitive	Non-tokenizing	Ignore case in matching. Default is case-sensitive indexing, setting: `true`.

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional

- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

`CREATE CUSTOM INDEX` creates a new SASI index on the specified table for column specified after the ON keyword. You can optionally define a name for the index before ON. Add the column name in parentheses after ON. It is not necessary for the column to exist on any current rows when the index is created, but the column and its data type must be defined when the table is created, or added afterward by altering the table.

You can specify the name of the table's keyspace before the table name, with period between them. If you do, Cassandra creates the index on a table in the specified keyspace, without changing the session's current keyspace; otherwise, Cassandra creates the index on the table within the current keyspace.

If you try to create a custom index that already exists, the command returns an error message. Exception: if you use the IF NOT EXISTS option, the command fails and outputs feedback to standard output.

Creating a SASI PREFIX index on a column

Define a table and then create an SASI index on the column `firstname`:

```
CREATE TABLE cycling.cyclist_name (
  id UUID PRIMARY KEY,
  lastname text,
  firstname text
);

CREATE CUSTOM INDEX  fn_prefix ON cyclist_name (firstname) USING
 'org.apache.cassandra.index.sasi.SASIIndex';
```

The SASI mode `PREFIX` is the default, and does not need to be specified.

Creating a SASI CONTAINS index on a column

Define a table and then create an SASI index on the column `firstname`:

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

The SASI mode `CONTAINS` must be specified.

Creating a SASI SPARSE index on a column

Define a table and then create an SASI index on the column `age`:

```
CREATE CUSTOM INDEX  fn_contains ON cyclist_name (age) USING
 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = { 'mode': 'SPARSE' };
```

The SASI mode `SPARSE` must be specified. This mode is used for dense number columns that store timestamps or millisecond sensor readings.

Creating a SASI PREFIX index on a column using the non-tokenizing analyzer

Define a table, then create an SASI index on the column age:

```
CREATE CUSTOM INDEX  fn_contains ON cyclist_name (age) USING
 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
'analyzer_class':
 'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
'case_sensitive': 'false'};
```

Using the non-tokenizing analyzer is a method to specify case sensitivity or character case normalization without analyzing the specified column.

Creating a SASI analyzing index on a column

Define a table and then create an SASI index on the column age:

```
CREATE CUSTOM INDEX stdanalyzer_idx ON cyclist_name (comments) USING
 'org.apache.cassandra.index.sasi.SASIIndex'
WITH OPTIONS = {
'mode': 'CONTAINS',
'analyzer_class':
 'org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer',
'analyzed': 'true',
'tokenization_skip_stop_words': 'and, the, or',
'tokenization_enable_stemming': 'true',
'tokenization_normalize_lowercase': 'true',
'tokenization_locale': 'en'
};
```

Specifying an analyzer allows

- Analyzing and indexing text column data
- Using word stemming for indexing
- Specifying words that can be skipped
- Applying localization based on a specified language
- Case normalization, like the non-tokening analyzer

CREATE INDEX

Define a new index on a single column of a table. To create indexes using SASI, see CREATE CUSTOM INDEX.

Synopsis

```
CREATE CUSTOM INDEX IF NOT EXISTS index_name
ON keyspace_name.table_name ( KEYS ( column_name ) )
(USING class_name) (WITH OPTIONS = map)
```

Restrictions: *Using class_name* is allowed only if CUSTOM is used and class_name is a string literal containing a java class name.

index_name is an identifier, enclosed or not enclosed in double quotation marks, excluding reserved words.

map is a map collection, a JSON-style array of literals:

```
{ literal : literal, literal : literal ... }
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

CREATE INDEX creates a new index on the given table for the named column. Attempting to create an already existing index will return an error unless the IF NOT EXISTS option is used. If you use the option, the statement will be a no-op if the index already exists. Optionally, specify a name for the index itself before the ON keyword. Enclose a single column name in parentheses. It is not necessary for the column to exist on any current rows. The column and its data type must be specified when the table is created, or added afterward by altering the table.

You can use dot notation to specify a keyspace for the table: keyspace name followed by a period followed the name of the table. Cassandra creates the table in the specified keyspace, but does not change the current keyspace; otherwise, if you do not use a keyspace name, Cassandra creates the index for the table within the current keyspace.

If data already exists for the column, Cassandra indexes the data during the execution of this statement. After the index is created, Cassandra indexes new data for the column automatically when new data is inserted.

Cassandra supports creating an index on most columns, excluding counter columns but including a clustering column of a compound primary key or on the partition (primary) key itself. Cassandra 2.1 and later supports creating an index on a collection or the key of a collection map. Cassandra rejects an attempt to create an index on both the collection key and value. In Cassandra 3.4 and later, static columns can be indexed.

Indexing can impact performance greatly. Before creating an index, be aware of when and when not to create an index.

Cassandra supports creating a custom index, which is primarily for internal use, and options that apply to the custom index. For example:

```
CREATE CUSTOM INDEX ON users (email) USING 'path.to.the.IndexClass';
CREATE CUSTOM INDEX ON users (email) USING 'path.to.the.IndexClass' WITH
 OPTIONS = {'storage': '/mnt/ssd/indexes/'};
```

In Cassandra 3.4 or later, a new custom SASI index has been added that has many advantages.

Creating an index on a column

Define a table and then create an index on two of its columns:

```
CREATE TABLE myschema.users (
   userID uuid,
   fname text,
   lname text,
   email text,
   address text,
   zip int,
```

```
   state text,
   PRIMARY KEY (userID)
 );

CREATE INDEX user_state
   ON myschema.users (state);

CREATE INDEX ON myschema.users (zip);
```

Creating an index on a clustering column

Define a table having a composite partition key, and then create an index on a clustering column.

```
CREATE TABLE mykeyspace.users (
   userID uuid,
   fname text,
   lname text,
   email text,
   address text,
   zip int,
   state text,
  PRIMARY KEY ((userID, fname), state)
);

CREATE INDEX ON mykeyspace.users (state);
```

Creating an index on a set or list collection

Create an index on a set or list collection column as you would any other column. Enclose the name of the collection column in parentheses at the end of the CREATE INDEX statement. For example, add a collection of phone numbers to the users table to index the data in the phones set.

```
ALTER TABLE users ADD phones set<text>;
CREATE INDEX ON users (phones);
```

If the collection is a map, Cassandra can create an index on map values. Assume the users table contains this map data from the example of a todo map:

```
{'2014-10-2 12:10' : 'die' }
```

The map key, the timestamp, is located to the left of the colon, and the map value is located to the right of the colon, 'die'. Indexes can be created on both map keys and map entries .

Creating an index on map keys

In Cassandra 2.1 and later, you can create an index on map collection keys. If an index of the map values of the collection exists, drop that index before creating an index on the map collection keys.

To index map keys, you use the KEYS keyword and map name in nested parentheses. For example, index the collection keys, the timestamps, in the todo map in the users table:

```
CREATE INDEX todo_dates ON users (KEYS(todo));
```

To query the table, you can use CONTAINS KEYin `WHERE` clauses.

Creating an index on the map entries

In Cassandra 2.2 and later, you can create an index on map entries. An `ENTRIES` index can be created only on a map column of a table that doesn't have an existing index.

To index collection entries, you use the ENTRIES keyword and map name in nested parentheses. For example, index the collection entries in a list in a race table:

```
CREATE INDEX entries_idx ON race (ENTRIES(race_wins));
```

To query the table, you can use a `WHERE` clause.

Creating an index on a full collection

In Cassandra 2.2 and later, you can create an index on a full `FROZEN` collection. An `FULL` index can be created on a set, list, or map column of a table that doesn't have an existing index.

To index collection entries, you use the FULL keyword and collection name in nested parentheses. For example, index the list rnumbers.

```
CREATE INDEX rnumbers_idx ON cycling.race_starts (FULL(rnumbers));
```

To query the table, you can use a `WHERE` clause.

CREATE FUNCTION

Synopsis

```
CREATE [OR REPLACE] FUNCTION [IF NOT EXISTS] <keyspace>.function-name
 ( <arg-name> <arg-type> )
(CALLED | RETURNS NULL) ON NULL INPUT
RETURNS <type>
LANGUAGE <language>
AS <body>
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Create or replace a user-defined function (UDF) that defines a function using Java or Javascript. By default, UDFs include Java generic methods where possible.

Examples

Compute logarithm of an input value. `CALLED ON NULL INPUT` ensures that the function will always be executed.

```
CREATE OR REPLACE FUNCTION fLog (input double) CALLED
 ON NULL INPUT RETURNS double LANGUAGE java AS 'return
 Double.valueOf(Math.log(input.doubleValue()));';
```

Compute logarithm of an input value. Return `NULL` if the input argument is `NULL`.

```
CREATE OR REPLACE FUNCTION fLog (input double) RETURNS
 NULL ON NULL INPUT RETURNS double LANGUAGE java AS 'return
 Double.valueOf(Math.log(input.doubleValue()));';
```

CREATE KEYSPACE

Define a new keyspace and its replica placement strategy.

Synopsis

```
CREATE ( KEYSPACE | SCHEMA ) IF NOT EXISTS keyspace_name
WITH REPLICATION = map
AND DURABLE_WRITES = ( true | false )
```

map is a map collection, a JSON-style array of literals:

```
{ literal : literal, literal : literal ... }
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

CREATE KEYSPACE creates a top-level namespace and sets the keyspace name, replica placement strategy class, replication factor, and DURABLE_WRITES options for the keyspace. For information about the replica placement strategy, see Cassandra 2.2 replica placement strategy or Cassandra 2.1 replica placement strategy.

When you configure NetworkTopologyStrategy as the replication strategy, you set up one or more virtual data centers. Alternatively, you use the default data center. Use the same names for data centers as those used by the snitch. For information about the snitch, see Cassandra 2.2 snitch documentation or Cassandra 2.1 snitch documentation.

You assign different nodes, depending on the type of workload, to separate data centers. For example, assign Hadoop nodes to one data center and Cassandra real-time nodes to another. Segregating workloads ensures that only one type of workload is active per data center. The segregation prevents incompatibility problems between workloads, such as different batch requirements that affect performance.

A map of properties and values defines the two different types of keyspaces:

```
{ 'class' : 'SimpleStrategy', 'replication_factor' : <integer> };
```

```
{ 'class' : 'NetworkTopologyStrategy'[, '<data center>' : <integer>, '<data
 center>' : <integer>] . . . };
```

Table: Table of map properties and values

Property	Value	Value Description
'class'	'SimpleStrategy' or 'NetworkTopologyStrategy'	Required. The name of the replica placement strategy class for the new keyspace.
'replication_factor'	<number of replicas>	Required if class is SimpleStrategy; otherwise, not used. The number of replicas of data on multiple nodes.
'<first data center>'	<number of replicas>	Required if class is NetworkTopologyStrategy and you provide the name of the first data center. This value is the number of replicas of data on each node in the first data center. Example
'<next data center>'	<number of replicas>	Required if class is NetworkTopologyStrategy and you provide the name of the second data center. The value is the number of replicas of data on each node in the data center.
. . .	. . .	More replication factors for optional named data centers.

CQL property map keys must be lower case. For example, class and replication_factor are correct. Keyspace names are 48 or fewer alpha-numeric characters and underscores, the first of which is an alpha character. Keyspace names are case-insensitive. To make a name case-sensitive, enclose it in double quotation marks.

You can use the alias CREATE SCHEMA instead of CREATE KEYSPACE. Attempting to create an already existing keyspace will return an error unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the keyspace already exists.

Example of setting the SimpleStrategy class

To construct the CREATE KEYSPACE statement, first declare the name of the keyspace, followed by the WITH REPLICATION keywords and the equals symbol. The name of the keyspace is case insensitive unless enclosed in double quotation marks. Next, to create a keyspace that is not optimized for multiple data centers, use SimpleStrategy for the class value in the map. Set replication_factor properties, separated by a colon and enclosed in curly brackets. For example:

```
CREATE KEYSPACE Excelsior
  WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' :
 3 };
```

Using SimpleStrategy is fine for evaluating Cassandra. For production use or for use with mixed workloads, use NetworkTopologyStrategy.

Example of setting the NetworkToplogyStrategy class

Using NetworkTopologyStrategy is also fine for evaluating Cassandra. To use NetworkTopologyStrategy for evaluation purposes using, for example, a single node cluster, specify the default data center name of the cluster. To determine the default data center name, use `nodetool status`.

```
$ nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns    Host ID
    Rack
UN  127.0.0.1  46.59 KB   256     100.0%  dd867d15-6536-4922-b574-
e22e75e46432  rack1
```

Cassandra uses datacenter1 as the default data center name. Create a keyspace named NTSkeyspace on a single node cluster, for example:

```
CREATE KEYSPACE NTSkeyspace WITH REPLICATION = { 'class' :
 'NetworkTopologyStrategy', 'datacenter1' : 1 };
```

To use NetworkTopologyStrategy with data centers in a production environment, you need to change the default snitch, SimpleSnitch, to a network-aware snitch, define one or more data center names in the snitch properties file, and use those data center name(s) to define the keyspace; otherwise, Cassandra will fail to find a node, to complete a write request, such as inserting data into a table.

After configuring Cassandra to use a network-aware snitch, such as the PropertyFileSnitch, you define data center and rack names in the `cassandra-topology.properties` file.

Construct the CREATE KEYSPACE statement using NetworkTopologyStrategy for the class value in the map. Set one or more key-value pairs consisting of the data center name and number of replicas per data center, separated by a colon and enclosed in curly brackets. For example:

```
CREATE KEYSPACE "Excalibur"
  WITH REPLICATION = {'class' : 'NetworkTopologyStrategy', 'dc1' : 3,
 'dc2' : 2};
```

This example sets three replicas for a data center named dc1 and two replicas for a data center named dc2. The data center name you use depends on the cluster-configured snitch you are using. There is a correlation between the data center name defined in the map and the data center name as recognized by the snitch you are using. The `nodetool status` command prints out data center names and rack locations of your nodes if you are not sure what they are.

Note: For more about replication strategy options, see Changing keyspace replication strategy

Setting DURABLE_WRITES

You can set the DURABLE_WRITES option after the map specification of the CREATE KEYSPACE command. When set to false, data written to the keyspace bypasses the commit log. Be careful using this option because you risk losing data. Do not set this attribute on a keyspace using the SimpleStrategy.

```
CREATE KEYSPACE Risky
  WITH REPLICATION = { 'class' : 'NetworkTopologyStrategy',
  'datacenter1' : 3 } AND DURABLE_WRITES = false;
```

Checking created keyspaces

Check that the keyspaces were created:

```
SELECT * FROM system.schema_keyspaces;
```

```
keyspace_name  | durable_writes | strategy_class
           | strategy_options
---------------+----------------
+-----------------------------------------------------
+---------------------------
     excelsior |           True |
 org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"3"}
     Excalibur |           True |
 org.apache.cassandra.locator.NetworkTopologyStrategy |
 {"dc2":"2","dc1":"3"}
         risky |          False |
 org.apache.cassandra.locator.NetworkTopologyStrategy |
 {"datacenter1":"1"}
        system |           True |
 org.apache.cassandra.locator.LocalStrategy |                         {}
 system_traces |           True |
 org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}

(5 rows)
```

Cassandra converted the excelsior keyspace to lowercase because quotation marks were not used to create the keyspace and retained the initial capital letter for the Excalibur because quotation marks were used.

For more information about creating and configuring keyspaces, see Updating the replication factor on page 12

Related information

Cassandra 2.2 replication strategy
Cassandra 2.1 replication strategy
Cassandra 2.2 snitch configuration
Cassandra 2.1 snitch configuration
Cassandra 2.2 property file snitch
Cassandra 2.1 property file snitch

CREATE MATERIALIZED VIEW

Create a materialized view in Cassandra 3.0 and later.

Synopsis

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] [ keyspace_name. ] view_name
AS SELECT column_name [ , column_name ] [ . . . ]
FROM [ keyspace_name. ] table_name
WHERE source_pk_column_name IS NOT NULL [ AND source_pk_column_name IS NOT
 NULL [ . . . ] ]
[ AND relation ] [ AND [ relation ] [ . . . ]
PRIMARY KEY source_pk_column_name , [  source_pk_column_name  [ . . . ] ]
 [ column_name ]
WITH { property [ AND property ] [ . . . ] | option [ AND option ]
 [ . . . ] }
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

In Cassandra 3.0 and later, `CREATE MATERIALIZED VIEW` creates a new materialized view from a specified source table with specific properties.

Requirements for the materialized view:

- The columns of the source table's primary key must be part of the materialized view's primary key.
- Only one new column can be added to the materialized view's primary key. Static columns are not allowed.

Parameters

CREATE MATERIALIZED VIEW clause

IF NOT EXISTS

Cassandra checks on whether a table or materialized view with the same name already exists. If this expression returns false, the operation fails. Optional.

keyspace_name

To create a materialized view in a keyspace other than the current keyspace, put the keyspace name in front of the materialized view name, followed by a period.

view_name

The name of the new materialized view.

AS SELECT clause

column_name

In the SELECT clause, the name of each column in the materialized view. Each column must exist in the source table.

You cannot add any static columns in the source table to the materialized view. If added, Cassandra creates the materialized view without the static columns.

FROM clause

keyspace_name

The keyspace name that qualifies the name of the source table, followed by a period. Optional.

table_name

The name of the source table.

WHERE clause

source_pk_column_name

One of the source table's primary key columns. The NOT NULL entry ensures that Cassadra only populates the materialized view with rows from the source table that have non-null values for each primary key column.

relation

Other relations that target the specific data needed. See the relation section of the CQL SELECT documentation.

PRIMARY KEY clause

source_pk_column_name

The materialized view's primary key must contain each column in the primary key of the source table.

column_name

In addition to the primary key columns, you can add one other source table column to the materialized view's primary key. Static columns from the xorce table are not allowed.

WITH clause

Specify other table properties in the same way as you can for source tables.

To change an existing materialized view, use ALTER MATERIALIZED VIEW on page 131.

Example

Creates the materialized view `cyclist_by_age` based on the source table `cyclist_mv`. The `WHERE` clause ensures that only rows whose `age` and `cid` columns are non-NULL are added to the materialized view.

```
CREATE MATERIALIZED VIEW cyclist_by_age
AS SELECT age, birthday, name, country
FROM cyclist_mv
WHERE age IS NOT NULL AND cid IS NOT NULL
PRIMARY KEY (age, cid);
```

Related information

Creating a materialized view on page 21

ALTER MATERIALIZED VIEW on page 131

DROP MATERIALIZED VIEW on page 171

CREATE TABLE

Define a new table.

Synopsis

```
CREATE TABLE IF NOT EXISTS <keyspace_name>.table_name
( column_definition(, column_definition, ...))
WITH property (AND property ...) | option (AND option ...)
```

column_definition is:

```
column_name cql_type STATIC PRIMARY KEY
| column_name <tuple<tuple_type> tuple<tuple_type>... > PRIMARY KEY
| column_name frozen<user-defined_type> PRIMARY KEY
| column_name user-defined_type PRIMARY KEY
| column_name frozen<collection_type <collection_type>>... PRIMARY KEY
| column_name collection_type<collection_type,
 frozen<collection_type>>... PRIMARY KEY
| ( PRIMARY KEY ( partition_key ) )
```

Note: Frozen collections can be used for primary key columns. Non-frozen collections cannot be used for primary key columns.

Restrictions:

- There should always be exactly one primary key definition.
- cql_type of the primary key must be a CQL data type or a user-defined type.
- cql_type of a collection uses this syntax:

```
LIST<cql_type>
| SET<cql_type>
| MAP<cql_type, cql_type>
```

PRIMARY KEY is:

```
column_name
| ( column_name1, column_name2, column_name3 ...)
| ((column_name4, column_name5), column_name6, column_name7 ...)
```

column_name1 is the partition key.

column_name2, column_name3 ... are clustering columns.

column_name4, column_name5 are partitioning keys.

column_name6, column_name7 ... are clustering columns.

option is:

```
COMPACT STORAGE
| ID <id>
| ( CLUSTERING ORDER BY (clustering_column ( ASC) | DESC ), ...) )
```

property is a CQL table property, enclosed in single quotation marks in the case of strings, or one of these directives:

```
WITH compaction = { 'class' : 'LeveledCompactionStrategy' }
AND read_repair_chance = 1.0
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

The `CREATE TABLE` command creates a new table under the current keyspace.

The `IF NOT EXISTS` keywords may be used in creating a table. Attempting to create an existing table returns an error unless the `IF NOT EXISTS` option is used. If the option is used, the statement if a no-op if the table already exists.

A static column can store the same data in multiple clustered rows of a partition, and then retrieve that data with a single `SELECT` statement.

You can add a counter column to a counter table.

Defining a column

You assign a type to columns during table creation. Column types, other than collection-type columns, are specified as a parenthesized, comma-separated list of column name and type pairs.

This example shows how to create a table that includes collection columns and a tuple.

```
CREATE TABLE users (
  userid text PRIMARY KEY,
  first_name text,
  last_name text,
  emails set<text>,
  top_scores list<int>,
  todo map<timestamp, text>,
  tuple<int, text,text>
);
```

See "Creating a user-defined type" for information on creating UDTs. This example shows the use of a UDT in a table.

```
CREATE TYPE fullname ( firstname text, lastname text );
CREATE TABLE users (
  userid text PRIMARY KEY,
  username FROZEN<fullname>,
  emails set<text>,
  top_scores list<int>,
  todo map<timestamp, text>,
  tuple<int, text,text>
);
```

In Cassandra 3.6 and later, UDTs can be created unfrozen if only non-collection fields are used in the user-defined type creation. If the table is created with an unfrozen UDT, then individual field values can be updated and deleted.

Using compact storage

The `WITH COMPACT STORAGE` keywords are used to store data in the legacy (Thrift) storage engine format to conserve disk space. However, for Cassandra 3.0 and later, the storage engine is much more efficient at storing data, and compact storage is not very necessary.

```
CREATE TABLE sblocks (
  block_id uuid,
  subblock_id uuid,
  data blob,
  PRIMARY KEY (block_id, subblock_id)
)
WITH COMPACT STORAGE;
```

Creating a table WITH ID

If a table is accidentally dropped with DROP TABLE, this option can be used to recreate the table and run a commitlog replay to retrieve the data.

```
CREATE TABLE users (
  userid text PRIMARY KEY,
  emails set<text>
) WITH ID='5a1c395e-b41f-11e5-9f22-ba0be0483c18';
```

Defining a primary key column

The only schema information that must be defined for a table is the primary key and its associated data type. Unlike earlier versions, CQL does not require a column in the table that is not part of the primary key. A primary key can have any number (1 or more) of component columns.

If the primary key consists of only one column, you can use the keywords, PRIMARY KEY, after the column definition:

```
CREATE TABLE users (
  user_name varchar PRIMARY KEY,
  password varchar,
  gender varchar,
  session_token varchar,
  state varchar,
  birth_year bigint
);
```

Alternatively, you can declare the primary key consisting of only one column in the same way as you declare a compound primary key. Do not use a counter column for a key.

Setting a table property

Using the optional WITH clause and keyword arguments, you can configure caching, compaction, and a number of other operations that Cassandra performs on new table. You can use the WITH clause to specify the properties of tables listed in CQL table properties, including caching, table comments, compression, and compaction. Format the property as either a string or a map. Enclose a string property in single quotation marks. For example, to embed a comment in a table, you format the comment as a string property:

```
CREATE TABLE MonkeyTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
)
WITH comment='Important biological records'
AND read_repair_chance = 1.0;
```

To configure compression and compaction, you use property maps:

```
CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH compression =
    { 'sstable_compression' : 'DeflateCompressor', 'chunk_length_kb' : 64 }
  AND compaction =
    { 'class' : 'SizeTieredCompactionStrategy', 'min_threshold' : 6 };
```

To specify using compact storage or clustering order use the WITH clause.

To configure caching in Cassandra 2.1, you also use a property map.

```
// Cassandra 2.1
```

```
CREATE TABLE DogTypes (
             ...   block_id uuid,
             ...   species text,
             ...   alias text,
             ...   population varint,
             ...   PRIMARY KEY (block_id)
             ... ) WITH caching = '{ 'keys' : 'NONE', 'rows_per_partition' :
 '120' }';
```

To configure caching in Cassandra 2.0.x, you do not use a property map. Simply set the caching property to a value:

```
// Cassandra 2.0.x only

CREATE TABLE DogTypes (
  block_id uuid,
  species text,
  alias text,
  population varint,
  PRIMARY KEY (block_id)
) WITH caching = 'keys_only';
```

Important: In Cassandra 2.0.x, use row caching with caution.

Using a compound primary key

A compound primary key consists of more than one column and treats the first column declared in a definition as the partition key. To create a compound primary key, use the keywords, PRIMARY KEY, followed by the comma-separated list of column names enclosed in parentheses.

```
CREATE TABLE emp (
  empID int,
  deptID int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (empID, deptID)
);
```

Using a composite partition key

A composite partition key is a partition key consisting of multiple columns. You use an extra set of parentheses to enclose columns that make up the composite partition key. The columns within the primary key definition but outside the nested parentheses are clustering columns. These columns form logical sets inside a partition to facilitate retrieval.

```
CREATE TABLE Cats (
  block_id uuid,
  breed text,
  color text,
  short_hair boolean,
  PRIMARY KEY ((block_id, breed), color, short_hair)
);
```

For example, the composite partition key consists of block_id and breed. The clustering columns, color and short_hair, determine the clustering order of the data. Generally, Cassandra will store columns having the same block_id but a different breed on different nodes, and columns having the same block_id and breed on the same node.

Using clustering order

You can order query results to make use of the on-disk sorting of columns. You can order results in ascending or descending order. The ascending order will be more efficient than descending. If you need results in descending order, you can specify a clustering order to store columns on disk in the reverse order of the default. Descending queries will then be faster than ascending ones.

The following example shows a table definition that changes the clustering order to descending by insertion time.

```
create table timeseries (
  event_type text,
  insertion_time timestamp,
  event blob,
  PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

Sharing a static column

In a table that uses clustering columns, non-clustering columns can be declared static in the table definition. Static columns are only static within a given partition.

```
CREATE TABLE t (
  k text,
  s text STATIC,
  i int,
  PRIMARY KEY (k, i)
);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m shared', 0);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m still shared', 1);
SELECT * FROM t;
```

Output is:

```
 k |                  s | i
---------------------------
k  | "I'm still shared" | 0
k  | "I'm still shared" | 1
```

Restrictions

- A table that does not define any clustering columns cannot have a static column. The table having no clustering columns has a one-row partition in which every column is inherently static.
- A table defined with the COMPACT STORAGE directive cannot have a static column.
- A column designated to be the partition key cannot be static.

You can batch conditional updates to a static column.

In Cassandra 2.0.9 and later, you can use the DISTINCT keywordto select static columns. In this case, Cassandra retrieves only the beginning (static column) of the partition.

CREATE TRIGGER

Registers a trigger on a table.

Synopsis

```
CREATE TRIGGER IF NOT EXISTS trigger_name ON table_name
```

```
USING 'java_class'
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

The implementation of triggers includes the capability to register a trigger on a table using the familiar CREATE TRIGGER syntax. The Trigger API is semi-private and subject to change.

```
CREATE TRIGGER myTrigger
  ON myTable
  USING 'org.apache.cassandra.triggers.AuditTrigger'
```

In Cassandra 2.1 and later, you need to enclose trigger names that use uppercase characters in double quotation marks. The logic comprising the trigger can be written in any Java (JVM) language and exists outside the database. The Java class in this example that implements the trigger is named `org.apache.cassandra.triggers` and defined in an Apache repository. The trigger defined on a table fires before a requested DML statement occurs to ensures the atomicity of the transaction.

Place the custom trigger code (JAR) in the `triggers` directory on every node. The custom JAR loads at startup. The location of triggers directory depends on the installation:

- Cassandra 2.0.x tarball: *`install_location`*`/lib/triggers`
- Cassandra 2.1.x tarball: *install_location*/conf/triggers
- Datastax Enterprise 4.5 and later: Installer-No Services and tarball: *`install_location`*`/resources/cassandra/conf/triggers`
- Datastax Enterprise 4.5 and later: Installer-Services and packages: `/etc/dse/cassandra/triggers`

Cassandra 2.1.1 and later supports lightweight transactions for creating a trigger. Attempting to create an existing trigger returns an error unless the IF NOT EXISTS option is used. If the option is used, the statement is a no-op if the table already exists.

CREATE TYPE

Create a user-defined type in Cassandra 2.1 and later.

Synopsis

```
CREATE TYPE IF NOT EXISTS keyspace.type_name
( field, field, ...)
```

type_name is a type identifier other than the type names.

field is:

```
field_name type
```

field_name is an arbitrary identifier for the field.

type is a CQL collection or non-collection type other than a counter type.

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

A user-defined type is one or more typed fields. A user-defined type facilitates handling multiple fields of related information, such as address information: street, city, and postal code. Attempting to create an already existing type will return an error unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the type already exists.

To create a user-defined type, use the CREATE TYPE command followed by the name of the type and a list of fields delimited by commas and enclosed in parentheses.

Choose a name for the user-defined type other than reserved type names, such as:

- byte
- smallint
- complex
- enum
- date
- interval
- macaddr
- bitstring

If you are in the system keyspace, which is the keyspace when you launch cqlsh, you need to specify a keyspace for the type. You can use dot notation to specify a keyspace for the type: keyspace name followed by a period followed the name of the type. Cassandra creates the type in the specified keyspace, but does not change the current keyspace; otherwise, if you do not specify a keyspace, Cassandra creates the type within the current keyspace.

Example

This example creates a user-defined type `cycling.basic_info` that consists of personal data about an individual cyclist.

```
cqlsh> CREATE TYPE cycling.basic_info (
  birthday timestamp,
  nationality text,
  weight text,
  height text
);
```

After defining the UDT, you can create a table using the UDT. CQL collection columns and other columns support the use of user-defined types, as shown in Using CQL examples.

CREATE ROLE

Create a new role.

Synopsis

```
CREATE ROLE [ IF NOT EXISTS ] role_name
WITH PASSWORD = [ 'password'
[AND | WITH] LOGIN = true | false ]
[AND | WITH] SUPERUSER = true | false ]
[ OPTIONS = <map_literal> ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

CREATE ROLE defines a new database role. By default, roles do not have superuser status or login privileges. Use SUPERUSER or LOGIN to assign these privileges to a role. A superuser or a user with CREATE permissions can issue CREATE ROLE requests. Use NOSUPERUSER to revoke superuser status from a role. The user *cassandra* exists as a default superuser; after creating a non-default superuser role, alter the *cassandra* user to a non-superuser.

Roles accounts are used for logging in under internal authentication and authorization. If internal authentication is not enabled, or login is not authorized, WITH PASSWORD is not necessary.

Enclose the role name in single quotation marks if it contains non-alphanumeric characters. If a role exists, it cannot be recreated. To change the superuser status or password, use ALTER ROLE.

Creating internal roles

You need to use the WITH PASSWORD clause when creating a role for internal authentication. Enclose the password in single quotation marks.

```
CREATE ROLE coach WITH PASSWORD = 'All4One2day!' AND LOGIN = true AND
 SUPERUSER = true;
```

If internal authentication has not been set up, you do not need the WITH PASSWORD clause:

```
CREATE ROLE manager NOSUPERUSER;
```

Creating a role conditionally

In Cassandra 2.2 and later, you can test that the role does not exist before attempting to create one. Attempting to create an existing role results in an invalid query condition unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the role exists.

```
cqlsh> CREATE ROLE IF NOT EXISTS new_role WITH PASSWORD = 'password';
```

Using a map to create role

A map may be used to create a role assigning custom options.

```
cqlsh> CREATE ROLE manager WITH OPTIONS = { 'custom_option1' :
 'option1_value', 'custom_option2' : 99 };
```

CREATE USER

Create a new user.

Notice: `CREATE USER` is supported for backwards compatibility. Authentication and authorization for Cassandra 2.2 and later are based on `ROLES`, and `CREATE ROLE` should be used.

Synopsis

```
CREATE USER [ IF NOT EXISTS ] user_name
WITH PASSWORD 'password'
[ SUPERUSER | NOSUPERUSER ]
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

`CREATE USER` defines a new database user account. By default users accounts do not have superuser status. Only a superuser can issue `CREATE USER` requests. See CREATE ROLE for more information about `SUPERUSER` and `NOSUPERUSER`.

User accounts are required for logging in under internal authentication and authorization.

Enclose the user name in single quotation marks if it contains non-alphanumeric characters. You cannot recreate an existing user. To change the superuser status or password, use ALTER USER.

Creating internal user accounts

Use `WITH PASSWORD` to create a user account for internal authentication. Enclose the password in single quotation marks.

```
CREATE USER spillman WITH PASSWORD 'Niner27';
CREATE USER akers WITH PASSWORD 'Niner2' SUPERUSER;
CREATE USER boone WITH PASSWORD 'Niner75' NOSUPERUSER;
```

If internal authentication has not been set up, `WITH PASSWORD` is not required.

```
CREATE USER test NOSUPERUSER;
```

Creating a user account conditionally

In Cassandra 2.0.9 and later, you can test that the user does not have an account before attempting to create one. Attempting to create an existing user results in an invalid query condition unless the IF NOT EXISTS option is used. If the option is used, the statement will be a no-op if the user exists.

```
$ bin/cqlsh -u cassandra -p cassandra
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.3.0 | Native protocol v3]
Use HELP for help.

cqlsh> CREATE USER newuser WITH PASSWORD 'password';

cqlsh> CREATE USER newuser WITH PASSWORD 'password';
code=2200 [Invalid query] message="User newuser already exists"

cqlsh> CREATE USER IF NOT EXISTS newuser WITH PASSWORD 'password';
cqlsh>
```

DELETE

Removes entire rows or one or more columns from one or more rows.

Synopsis

```
DELETE [ column_name [ , column_name ] [ . . . ] | column_name [ term ] ]
FROM [ keyspace_name. ] table_name
[ USING TIMESTAMP timestamp_value ]
WHERE row_specification
[ { IF EXISTS | IF condition [ AND condition ] [ . . . ] } ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

`DELETE` removes one or more columns from one or more rows in a table, or removes the entire rows. Cassandra applies selections within the same partition key atomically and in isolation.

When a column is deleted, it is not removed from disk immediately. The deleted column is marked with a tombstone and then removed after the configured grace period has expired. The optional timestamp defines the new tombstone record.

Parameters

column_name

The name of the column to be deleted, or a list of column names

term

If column_name refers to a collection (a list or map), the parameter in parentheses indicates the term in the collection to be deleted. For each collection type, use term to specify:

list	index of the element in the list (the first item in the list has index 0, etc.)
map	the key of the element to be deleted

keyspace_name

The name of the keyspace containing the table from which data will be deleted. Not needed if the keyspace has been set for the session with the `USE` command.

table_name

The name of the table from which data will be deleted

timestamp_value

If a TIMESTAMP is specified, the command only deletes elements older than the timestamp_value (which match the WHERE and optional IF conditions).

row_specification

The `WHERE` clause must identify the row or rows to be deleted by primary key:

- To specify one row, use *`primary_key_name = primary_key_value`*. If the primary key is a combination of elements, follow this with `AND` *`primary_key_name = primary_key_value ...`*. In Cassandra 2.2, the WHERE clause must specify a value for every component of the primary key. In Cassandra 3.0 and later, range deletion may be done using inequality operators for clustering columns in the primary key.
- To specify more than one row, use *`primary_key_name`* `IN (` *`primary_key_value, primary_key_value`* `…` `)` As in the previous option, each *`primary_key_name`* and *`primary_key_value`* must match the composition of the primary key.

Note: To delete a static column, only the partition key must be specified.

IF EXISTS / IF condition

An IF clause can limit the command's action on rows that match the WHERE clause. Two options for IF:

- Use IF EXISTS to make the `DELETE` fail if there are no rows that match the `WHERE` conditions.
- Use IF to specify one or more conditions that must test true for the values in the specified row or rows.

If an IF is used, the command returns a result to standard output. See Conditionally deleting columns for examples.

IF NOT EXISTS condition

Use IF NOT EXISTS to make the `DELETE` fail if there are rows that match the `WHERE` conditions. An IF NOT EXISTS clause can limit the command's action on rows that do not match the WHERE clause. Two options for IF:

- .
- Use IF to specify one or more conditions that must test true for the values in the specified row or rows.

If an IF is used, the command returns a result to standard output. See Conditionally deleting columns for examples.

Deleting columns or a row

Delete specific columns by listing them after the DELETE command, separated by commas.

```
DELETE firstname, lastname FROM cycling.cyclist_name WHERE firstname =
 'Alex';
```

When no column are listed after `DELETE`, command deletes the entire row or rows specified in the WHERE clause.

```
DELETE FROM cycling.cyclist_name WHERE firstname IN ('Alex', 'Marianne');
```

Specifying the table

The table name follows the list of column names and the keyword FROM, preceded by the keyspace name if necessary.

Conditionally deleting columns

In Cassandra 2.0.7 and later, you can conditionally delete columns using IF or IF EXISTS. Deleting a column is similar to making an insert or update conditionally.

Add IF EXISTS to the command to ensure that the operation is not performed if the specified row does not exist:

```
DELETE id FROM cyclist_id WHERE lastname = 'WELTEN' and firstname = 'Bram'
 IF EXISTS;
```

Without IF EXISTS, the command proceeds with no standard output. If IF EXISTS returns true (if a row with this primary key does exist), standard output displays a table like the following:

```
 [applied]
-----------
      True
```

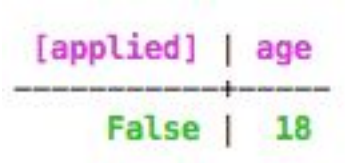
If no such row exists, however, the conditions returns FALSE and the command fails. In this case, standard output looks like:

```
 [applied]
-----------
     False
```

Use IF condition to apply tests to one or more column values in the selected row:

```
DELETE id FROM cyclist_id WHERE lastname = 'WELTEN' AND firstname = 'Bram'
 IF age = 2000;
```

If all the conditions return TRUE, standard output is the same as if IF EXISTS returned true (see above). If any of the conditions fails, standard output displays `False` in the `[applied]` column and also displays information about the condition that failed:



Conditional deletions incur a non-negligible performance cost and should be used sparingly.

Deleting old data using TIMESTAMP

The TIMESTAMP is an integer representing microseconds. You can identify the column for deletion using TIMESTAMP.

```
DELETE firstname, lastname
  FROM cycling.cyclist_name
  USING TIMESTAMP 1318452291034
  WHERE lastname = 'VOS';
```

Deleting more than one row

The WHERE clause specifies which row or rows to delete from the table.

```
DELETE FROM cycling.cyclist_name WHERE id = 6ab09bec-e68e-48d9-
a5f8-97e6fb4c9b47;
```

To delete more than one row, use the keyword IN and supply a list of values in parentheses, separated by commas:

```
DELETE FROM cycling.cyclist_name WHERE firstname IN ('Alex', 'Marianne');
```

In Cassandra 2.0 and later, CQL supports an empty list of values in the IN clause, useful in Java Driver applications.

Deleting from a collection set, list or map

To delete an element from a map that is stored as one column in a row, specify the *column_name* followed by the key of the element in square brackets:

```
DELETE sponsorship ['sponsor_name'] FROM cycling.races WHERE race_name =
 'Criterium du Dauphine';
```

To delete an element from a list, specify the *column_name* followed by the list index position in square brackets:

```
DELETE categories[3] FROM cycling.cyclist_history WHERE lastname =
 'TIRALONGO';
```

To delete all elements from a set, specify the *column_name* by itself:

```
DELETE sponsorship FROM cycling.races WHERE race_name = 'Criterium du
 Dauphine';
```

DROP AGGREGATE

Synopsis

```
DROP AGGREGATE IF EXISTS keyspace_name.aggregate_name
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Drop a user-defined aggregate.

Examples

Text

```
cqlsh> DROP AGGREGATE [IF EXISTS] myAverage;
```

DROP FUNCTION

Synopsis

```
DROP FUNCTION IF EXISTS keyspace_name.function_name
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Drop a user-defined function from a keyspace.

Examples

Text

```
cqlsh> DROP FUNCTION [IF EXISTS] fLog;
```

DROP INDEX

Drop the named index.

Synopsis

```
DROP INDEX IF EXISTS keyspace.index_name
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

A DROP INDEX statement removes an existing index. If the index was not given a name during creation, the index name is <table_name>_<column_name>_idx. If the index does not exists, the statement will return an error, unless IF EXISTS is used in which case the operation is a no-op. You can use dot notation to specify a keyspace for the index you want to drop: keyspace name followed by a period followed the name of the index. Cassandra drops the index in the specified keyspace, but does not change the current keyspace; otherwise, if you do not use a keyspace name, Cassandra drops the index for the table within the current keyspace.

Example

```
DROP INDEX user_state;

DROP INDEX users_zip_idx;

DROP INDEX myschema.users_state;
```

DROP KEYSPACE

Remove the keyspace.

Synopsis

```
DROP { KEYSPACE | SCHEMA } [ IF EXISTS ] keyspace_name
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used

Markup	Indicates
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

A DROP KEYSPACE statement results in the immediate, irreversible removal of a keyspace, including all tables and data contained in the keyspace. You can also use the alias DROP SCHEMA. If the keyspace does not exists, the statement will return an error unless IF EXISTS is used, in which case the operation is a no-op.

Cassandra takes a snapshot of the keyspace before dropping it. In Cassandra 2.0.4 and earlier, the user was responsible for removing the snapshot manually.

Example

```
DROP KEYSPACE MyTwitterClone;
```

DROP MATERIALIZED VIEW

Remove the named materialized view in Cassandra 3.0 and later.

Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS ] [ keyspace_name. ] view_name
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

In Cassandra 3.0 and later, `DROP MATERIALIZED VIEW` causes the immediate, irreversible removal of a materialized view, including all data it contains. This operation has no effect on the source table.

Note: Before you can drop a table, you must drop all materialized views derived from it.

Parameters

IF EXISTS

Cassandra checks on whether the specified materialized view exists. If the materialized view does not exist, the operation fails. Optional.

keyspace_name

To create a materialized view in a keyspace other than the current keyspace, put the keyspace name in front of the materialized view name, followed by a period.

view_name

The name of the materialized view to drop

Example

```
cqlsh> DROP MATERIALIZED VIEW cycling.cyclist_by_age;
```

Related information

Creating a materialized view on page 21

CREATE MATERIALIZED VIEW on page 153

ALTER MATERIALIZED VIEW on page 131

DROP ROLE

Remove a role.

Synopsis

```
DROP ROLE [ IF EXISTS ] role_name
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

DROP ROLE removes an existing user. In Cassandra 2.2 and later, you can test that the role exists. Attempting to drop a role that does not exist results in an invalid query condition unless the IF EXISTS option is used. If the option is used, the statement will be a no-op if the role does not exist.

The role used to drop roles must have DROP permission, either directly or on ALL ROLES. To drop a superuser role, a superuser role must be used. Roles cannot drop a role that is the currently logged in role.

Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

Examples

```
DROP ROLE IF EXISTS coach;
```

DROP TABLE

Remove the named table.

Synopsis

```
DROP TABLE IF EXISTS keyspace_name.table_name
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

A `DROP TABLE` statement results in the immediate, irreversible removal of a table, including all data contained in the table. You can also use the alias `DROP COLUMNFAMILY`.

In Cassandra 3.0 and later, you cannot drop a table until you have dropped any materialized views based on that table. If you try to drop a table with existing materialized views, Cassandra returns an error message that lists the materialized views to drop.

Example

```
DROP TABLE worldSeriesAttendees;
```

DROP TRIGGER

Removes registration of a trigger.

Synopsis

```
DROP TRIGGER IF EXISTS trigger_name ON table_name
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

The DROP TRIGGER statement removes the registration of a trigger created using CREATE TRIGGER. Cassandra 2.1.1 and later supports the IF EXISTS syntax for dropping a trigger. Cassandra checks for the existence of the trigger before dropping it.

The Trigger API is semi-private and subject to change.

DROP TYPE

Drop a user-defined type. Cassandra 2.1 and later.

Synopsis

```
DROP TYPE IF EXISTS type_name
```

type_name is the name of a user-defined type.

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

This statement immediately and irreversibly removes a type. To drop a type, use ALTER TYPE and the DROP keyword as shown in the following example. Attempting to drop a type that does not exist will return an error unless the IF EXISTS option is used. If the option is used, the statement will be a no-op if the type already exists. Dropping a user-defined type that is in use by a table or another type is not allowed.

```
DROP TYPE version;
```

DROP USER

Remove a user.

Notice: `DROP USER` is supported for backwards compatibility. Authentication and authorization for Cassandra 2.2 and later are based on `ROLES`, and `DROP ROLE` should be used.

Synopsis

```
DROP USER [ IF EXISTS ] user_name
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

`DROP USER` removes an existing user. Attempting to drop a user that does not exist results in an invalid query condition unless the `IF EXISTS` option is used. If the option is used, the statement will be a no-op if the user does not exist. A user must have appropriate permission to issue a DROP USER statement. Users cannot drop themselves.

Enclose the user name in single quotation marks only if it contains non-alphanumeric characters.

Examples

Drop a user if the user exists:

```
DROP USER IF EXISTS boone;
```

Drop a user:

```
DROP USER montana;
```

GRANT

Provide access to database objects.

Synopsis

```
GRANT [ ALL ]
CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | EXECUTE
ON [ ALL KEYSPACES | KEYSPACE keyspace_name |
TABLE table_name |
ALL ROLES ROLE role_name |
ALL FUNCTIONS [ IN KEYSPACE keyspace_name ] | FUNCTION function_name ]
TO role_name
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

Permissions to access all keyspaces, a named keyspace, or a table can be granted to a role. Enclose the role name in single quotation marks if it contains non-alphanumeric characters.

This table lists the permissions needed to use CQL statements:

Table: CQL Permissions

Permission	CQL Statement
ALL	All statements
ALTER	ALTER KEYSPACE, ALTER TABLE, CREATE INDEX, DROP INDEX, ALTER ROLE
AUTHORIZE	GRANT, REVOKE
CREATE	CREATE KEYSPACE, CREATE TABLE, CREATE FUNCTION, CREATE AGGREGATE, CREATE INDEX, CREATE ROLE
DROP	DROP KEYSPACE, DROP TABLE, DROP FUNCTION, DROP AGGREGATE, DROP INDEX, DROP ROLE
MODIFY	INSERT, DELETE, UPDATE, TRUNCATE
SELECT	SELECT

To perform `SELECT` queries on a table, `SELECT` permission on the table, on its parent keyspace, or on `ALL KEYSPACES` is required. To perform `CREATE TABLE`, `CREATE` permission on its parent keyspace or `ALL KEYSPACES` is required. A role with `SUPERUSER` status or `AUTHORIZE` permission on a resource (or one of its parents in the hierarchy) must also have the permission in question to `GRANT` or `REVOKE` that permission for a role. `GRANT`, `REVOKE` and `LIST` permissions check for the existence of the table and keyspace before execution. `GRANT` and `REVOKE` check that the user exists.

Examples

Give the role coach permission to perform `SELECT` queries on all tables in all keyspaces:

```
GRANT SELECT ON ALL KEYSPACES TO coach;
```

Give the role manager permission to perform `INSERT`, `UPDATE`, `DELETE` and `TRUNCATE` queries on all tables in the field keyspace.

```
GRANT MODIFY ON KEYSPACE field TO manager;
```

Give the role coach permission to perform `ALTER KEYSPACE` queries on the cycling keyspace, and also `ALTER TABLE`, `CREATE INDEX` and `DROP INDEX` queries on all tables in cycling keyspace:

```
GRANT ALTER ON KEYSPACE cycling TO coach;
```

Give the role coach permission to run all types of queries on cycling.name table.

```
GRANT ALL PERMISSIONS ON cycling.name TO coach;
```

Grant access to a keyspace to a role.

```
GRANT ALL ON KEYSPACE keyspace_name TO role_name;
```

INSERT

Add or update columns.

Synopsis

```
INSERT INTO [ keyspace_name. ] table_name ( column_name [ , column_name ]
  [ . . .  ] )
VALUES ( column_value [ , column_value [ . . . ] )
[ IF NOT EXISTS ]
[ USING { TTL time_value | TIMESTAMP timestamp_value } ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

INSERT writes a row to a Cassandra table atomically and in isolation. The command must specify the row by primary key. If the table already contains a row with the same primary key, the command writes (or overwrites) its values to the existing row. (Exception: if the command includes IF NOT EXISTS, and the row already exists, the command fails.)

Note: After the partition key, any column that is part of the primary key is used as a clustering key. In Cassandra3.0 and earlier, you cannot insert any value larger than 64K bytes into a clustering column.

INSERT does not return any result unless it includes IF NOT EXISTS. The command requires a value for each component of the primary key, but not for any of the row's other columns. Any column which is defined for the row but not populated with a value takes up no space on disk.

Internally, INSERT and UPDATE preform the same operation. A key difference: UPDATE supports counters, INSERT does not.

Parameters

keyspace_name

The name of the keyspace containing the table into which data will be inserted. Not needed if the keyspace has been set for the session with the USE command.

table_name

The name of the table into which data will be inserted

column_name

A list of one or more columns into which data will be inserted

column_value

For each column listed in the INSERT INTO list, the VALUES list must contain a corresponding value. The values must occur in the same order as the column names.

A column_value can be one of:

- a literal

- a set:

```
{ literal [, ... ] }
```

  (note the use of curly brackets)
- a list:

```
[ literal [, ... ] ]
```

  (note the use of square brackets)
- a map collection, a JSON-style array of literals:

```
{ literal : literal [, ... ] }
```

time_value

The value for TTL is a number of seconds. Column values in a command marked with TTL are automatically marked as deleted (with a tombstone) after the specified amount of time has expired. The TTL applies to the marked column values, not the column itself. Any subsequent update of the column resets the TTL to the TTL specified in the update. By default, values never expire. You cannot set a time_value for data in a counter column.

timestamp_value

If the optional TIMESTAMP parameter is used, the inserted column is marked with its value – a timestamp in microseconds. If a TIMESTAMP value is not set, Cassandra uses the time (in microseconds) that the write occurred to the column.

Specifying TTL and TIMESTAMP

- Time-to-live (TTL) in seconds
- Timestamp in microseconds

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
  VALUES (6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47, 'KRUIKSWIJK','Steven')
  USING TTL 86400;
```

```
INSERT INTO cycling.cyclist_name (id, lastname, firstname)
VALUES (220844bf-4860-49d6-9a4b-6b5d3a79cbfb, 'TIRALONGO','Paolo')
USING TIMESTAMP 123456789;
```

Note: IF NOT EXISTS and USING TIMESTAMP cannot be used in the same `INSERT`.

Using a collection set or map

To insert data into a collection, enclose values in curly brackets. Set values must be unique. Example: insert a list of categories for a cyclist.

```
INSERT INTO cycling.cyclist_categories (id,lastname,categories)
  VALUES('6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47', 'KRUIJSWIJK', {'GC', 'Time-
trial', 'Sprint'});
```

Insert a map named `teams` that lists two recent team memberships for the user *VOS*.

```
INSERT INTO cycling.cyclist_teams (id,lastname,teams)
  VALUES(5b6962dd-3f90-4c93-8f61-eabfa4a803e2, 'VOS', {2015 : 'Rabobank-Liv
 Woman Cycling Team', 2014 : 'Rabobank-Liv Woman Cycling Team' });
```

The size of one item in a collection is limited to 64K.

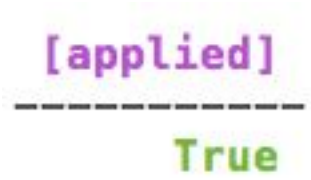
To insert data into a collection column of a user-defined type, enclose components of the type in parentheses within the curly brackets, as shown in "Using a user-defined type."

Inserting a row only if it does not already exist

Add IF NOT EXISTS to the command to ensure that the operation is not performed if a row with the same primary key already exists:

```
INSERT INTO cycling.cyclist_name (id,lastname,firstname)
  VALUES('c4b65263-fe58-83e8-f0e1c13d518f', 'RATTO', 'Risselda') IF NOT
 EXISTS;
```

Without IF NOT EXISTS, the command proceeds with no standard output. If IF NOT EXISTS returns true (if there is no row with this primary key), standard output displays a table like the following:



If, however, the row does already exist, the command fails, and standard out displays a table with the value `false` in the `[applied]` column, and the values that were not inserted, as in the following example:

Note: Using IF NOT EXISTS incurs a performance hit associated with using Paxos internally. For information about Paxos, see Cassandra 3.0 documentation.

Related information

Cassandra 2.1 tunable consistency

Cassandra 2.0 tunable consistency

Example of inserting data into playlists

The "Example of a music service" section described the playlists table. This example shows how to insert data into that table.

Procedure

Use the INSERT command to insert UUIDs for the compound primary keys, title, artist, and album data of the playslists table.

```
INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 1,
  a3e64f8f-bd44-4f28-b8d9-6938726e34d4, 'La Grange', 'ZZ Top', 'Tres
 Hombres');

INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 2,
```

```
  8a172618-b121-4136-bb10-f665cfc469eb, 'Moving in Stereo', 'Fu Manchu',
 'We Must Obey');

INSERT INTO playlists (id, song_order, song_id, title, artist, album)
  VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 3,
  2b09185b-fb5a-4734-9b56-49077de9edbf, 'Outside Woman Blues', 'Back Door
 Slam', 'Roll Away');
```

LIST PERMISSIONS

List permissions granted to a role.

Synopsis

```
LIST [ ALL PERMISSIONS] |
[ CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | EXECUTE
ON [ ALL KEYSPACES | KEYSPACE keyspace_name |
TABLE table_name |
ALL ROLES ROLE role_name |
ALL FUNCTIONS [ IN KEYSPACE keyspace_name ] | FUNCTION function_name ] ]
PERMISSION(S)
OF role_name [ NORECURSIVE ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

Permissions checks are recursive. If you omit the NORECURSIVE specifier, permission on the requests resource and its parents in the hierarchy are shown. The LIST command requires DESCRIBE permission.

- Omitting the resource name (ALL KEYSPACES, keyspace, or table), lists permissions on all tables and all keyspaces.
- Omitting the role name lists permissions of all roles. You need to be a superuser to list permissions of all roles. If you are not, you must add

  ```
  OF <my_rolename>
  ```

- Enclose the role name in single quotation marks only if it contains non-alphanumeric characters.

After creating roles and granting the permissions in the GRANT examples, you can list permissions that roles have on resources and their parents.

Example

List all permissions given to coach:

```
LIST ALL PERMISSIONS OF coach;
```

Output is:

```
 rolename | resource           | permission
----------+--------------------+------------
    coach | <keyspace field>   |     MODIFY
```

List permissions given to all the roles:

```
LIST ALL PERMISSIONS;
```

Output is:

```
 rolename | resource             | permission
----------+----------------------+------------
    coach |     <keyspace field> |     MODIFY
    manager | <keyspace cyclist> |      ALTER
    manager | <table cyclist.name> |     CREATE
    manager | <table cyclist.name> |      ALTER
    manager | <table cyclist.name> |       DROP
    manager | <table cyclist.name> |     SELECT
    manager | <table cyclist.name> |     MODIFY
    manager | <table cyclist.name> |  AUTHORIZE
    coach |      <all keyspaces> |     SELECT
```

List all permissions on the cyclist.name table:

```
LIST ALL PERMISSIONS ON cyclist.name;
```

Output is:

```
 username | resource             | permission
----------+----------------------+------------
    manager | <table cyclist.name> |     CREATE
    manager | <table cyclist.name> |      ALTER
    manager | <table cyclist.name> |       DROP
    manager | <table cyclist.name> |     SELECT
    manager | <table cyclist.name> |     MODIFY
    manager | <table cyclist.name> |  AUTHORIZE
    coach |      <all keyspaces> |     SELECT
```

List all permissions on the cyclist.name table and its parents:

Output is:

```
LIST ALL PERMISSIONS ON cyclist.name NORECURSIVE;
```

```
 username | resource             | permission
----------+----------------------+------------
    manager | <table cyclist.name> |     CREATE
    manager | <table cyclist.name> |      ALTER
    manager | <table cyclist.name> |       DROP
    manager | <table cyclist.name> |     SELECT
    manager | <table cyclist.name> |     MODIFY
    manager | <table cyclist.name> |  AUTHORIZE
```

LIST ROLES

List existing roles

Synopsis

```
LIST ROLES [ OF  'role_name'] [ NORECURSIVE ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

The `LIST ROLES` command will display users, their superuser status and login privilege. A user must have `DESCRIBE` permission to generate a list.

Example

Show all the roles that exist. Sufficient privileges are required to show this information.

```
LIST ROLES;
```

will return the output:

```
 role          | super | login | options
---------------+-------+-------+---------
     cassandra |  True |  True |      {}
      sysadmin |  True |  True |      {}
 team_manager |  True | False |      {}
```

Show the roles for a particular role. Sufficient privileges are required to show this information.

```
LIST ROLES OF @manager@;
```

LIST USERS

List existing users and their superuser status.

Notice: `LIST USERS` is supported for backwards compatibility. Authentication and authorization for Cassandra 2.2 and later are based on `ROLES`, and `LIST ROLES` should be used.

Synopsis

```
LIST USERS
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

Assuming you use internal authentication, created users with the command CREATE USER, and have not yet changed the default user, the following example shows the output of `LIST USERS`.

Example

List the current users:

```
LIST USERS;
```

Output is:

```
 name      | super
-----------+-------
 cassandra |  True
     boone | False
     akers |  True
  spillman | False
```

LOGIN

Synopsis

```
cqlsh> LOGIN user_name password
```

In the synopsis section of each statement, formatting has the following meaning:

- Uppercase means literal
- Lowercase means not literal

- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

Use this command to change login information without requiring `cqlsh` to restart. Login using a specified username. If the password is specified, it will be used. Otherwise, you will be prompted to enter the password.

Examples

Login as the user cutie with the password patootie.

```
LOGIN cutie patootie
```

REVOKE

Deny access to database objects.

Synopsis

```
REVOKE [ ALL ]
CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | EXECUTE
ON [ ALL KEYSPACES | KEYSPACE keyspace_name |
TABLE table_name |
ALL ROLES ROLE role_name |
ALL FUNCTIONS [ IN KEYSPACE keyspace_name ] | FUNCTION function_name ]
FROM role_name
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
Italics	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

Permissions to access all keyspaces, a named keyspace, or a table can be revoked from a role. Enclose the role name in single quotation marks if it contains non-alphanumeric characters.

The table in GRANT lists the permissions needed to use CQL statements:

Example

```
REVOKE SELECT ON cycling.name FROM manager;
```

The role manager can no longer perform SELECT queries on the cycling.name table. Exceptions: Because of inheritance, the user can perform SELECT queries on cycling.name if one of these conditions is met:

- The user is a superuser.
- The user has SELECT on ALL KEYSPACES permissions.
- The user has SELECT on the cycling keyspace.

```
REVOKE ALTER ON ALL ROLES FROM coach;
```

The role coach can no longer perform GRANT, ALTER or REVOKE commands on all roles.

SELECT

Retrieve data from a Cassandra table. For retrieving data using a SSTable Attached Secondary Index, see Using SASI.

Synopsis

```
SELECT select_expression
FROM [ keyspace_name. ] table_name
[ WHERE relation [ AND relation ] [ . . . ] ]
[ ORDER BY clustering_column [ { ASC | DESC } ]
[ LIMIT n ]
[ ALLOW FILTERING ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

The SELECT statement returns one or more rows from a Cassandra table. The WHERE clause introduces matching conditions that specify the rows to be returned. ORDER BY tells Cassandra how to sort these rows.

Parameters

select_expression

A specifies the rows to be returned.

```
* | DISTINCT partition_key | selection_list
```

Options:

*	Selects all columns.
`DISTINCT partition_key`	Returns a list all the different values that occur in the column. Only works with the partition key of the table.

selection_list

A list of values to be retrieved. Options:

```
column_name | selector [ AS alias ] [, …]
```

A *column_name* must match a column in the table.

alias

If you specify a literal value as an *alias* using the AS keyword , it replaces the column name or selector code in the headings of the output.

selector

Any native Cassandra function or aggregate, user-defined function, or user-defined aggregate.

function

Examples of native functions: standard aggregate functions, , timeuuid function, a token function, or a blob conversion function.

relation

A logical expression. Cassandra returns only those rows that return true for each relation. A relation can consist of:

```
column_name operator term
| ( column_name [ , column_name . . . ) operator term-tuple
| column_name IN ( term , term [ ,  term ] . . . ]
| ( column_name, column_name [ , column_name ] . . . ) IN term-tuple [ , term-
tuple ] . . . )
| TOKEN ( column_name ) operator TOKEN ( column_name )
```

operator

The logical symbol that specifies the relationship between the two sides of the relation. Casandra supports the following operators:

= | < | > | <= | >= | `CONTAINS` | `CONTAINS KEY`

term

- a constant: string, number, uuid, boolean, hex
- a function
- a collection:
  - set

    ```
    { literal, literal, ... }
    ```

term-tuple

```
( term, term, ... )
```

TOKEN

A Cassandra token

Specifying columns

The columns referenced in the SELECT clause must exist in the target table.

Columns in big data applications duplicate values. Use the DISTINCT keyword to return only distinct (different) values of partition keys.

You can program Cassandra to perform calculations on returned values by specifying functions that operate on the columns being returned. For details, see Retrieving aggregate values

Using a column alias

When your selection list includes functions or other complex expressions, use aliases to make the output more readable. This example applies aliases to the `dateOf(created_at)` and `blobAsText(content)` functions:

```
SELECT event_id,
  dateOf(created_at) AS creation_date,
  blobAsText(content) AS content
  FROM timeline;
```

The output labels these columns with more understandable names:

```
event_id                | creation_date            | content
------------------------+--------------------------+----------------
 550e8400-e29b-41d4-a716 | 2013-07-26 10:44:33+0200 | Some stuff
```

Specifying the source table using FROM

The `FROM` clause specifies the table to query. You may want to precede the table name with the name of the keyspace followed by a period (.). If you do not specify a keyspace, Cassandra queries the current keyspace.

The following example SELECT statement returns the number of rows in the `IndexInfo` table in the `system` keyspace:

```
SELECT COUNT(*) FROM system.IndexInfo;
```

Controlling the number of rows returned using LIMIT

The `LIMIT` option sets the maximum number of rows that the query returns:

```
SELECT lastname FROM cycling.cyclist_name LIMIT 50000;
```

Even if the query matches 105,291 rows, Cassandra only returns the first 50,000.

The `cqlsh` shell has a default row limit of 10,000. The Cassandra server and native protocol do not limit the number of returned rows, but they apply a timeout to prevent malformed queries from causing system instability.

Controlling the number of rows returned using PER PARTITION LIMIT

In Cassandra 3.6 and later, the `PER PARTITION LIMIT` option sets the maximum number of rows that the query returns from each partition. For example, create a table that will sort data into more than one partition.

```
CREATE TABLE cycling.rank_by_year_and_name (
```

```
  race_year int,
  race_name text,
  cyclist_name text,
  rank int,
  PRIMARY KEY ((race_year, race_name), rank) );
```

After inserting data, the table holds:

```
race_year | race_name                                    | rank | cyclist_name
-----------+----------------------------------------------+------
+----------------------
      2014 |                         4th Tour of Beijing |    1 |
 Phillippe GILBERT
      2014 |                         4th Tour of Beijing |    2 |
 Daniel MARTIN
      2014 |                         4th Tour of Beijing |    3 | Johan
 Esteban CHAVES
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |
 Ilnur ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |      Carlos
 BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |
 Benjamin PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |
 Adam PHELAN
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |
 Thomas LEBAS
```

To get the top two racers in every race year and race name, use the `SELECT` statement with `PER PARTITION LIMIT 2`.

```
SELECT * FROM cycling.rank_by_year_and_name PER PARTITION LIMIT 2;
```

Output:

```
race_year | race_name                                    | rank | cyclist_name
-----------+----------------------------------------------+------
+-------------------
      2014 |                         4th Tour of Beijing |    1 | Phillippe
 GILBERT
      2014 |                         4th Tour of Beijing |    2 |     Daniel
 MARTIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    1 |     Ilnur
 ZAKARIN
      2015 |   Giro d'Italia - Stage 11 - Forli > Imola |    2 |   Carlos
 BETANCUR
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |   Benjamin
 PRADES
      2015 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 |       Adam
 PHELAN
```

Filtering data using WHERE

The `WHERE` clause introduces one or more relations that filter the rows returned by `SELECT`.

The column specifications

The column specification of the relation must be one of the following:

- One or more members of the partition key of the table
- A clustering column, only if the relation is preceded by other relations that specify all columns in the partition key

- A column that is indexed using CREATE INDEX.

Note:

In the `WHERE` clause, refer to a column using the actual name, not an alias.

Filtering on the partition key

For example, the following table definition defines `id` as the table's partition key:

```
CREATE TABLE cycling.cyclist_career_teams ( id UUID PRIMARY KEY, lastname
 text, teams set<text> );
```

In this example, the SELECT statement includes in the partition key, so the WHERE clause can use the `id` column:

```

SELECT id, lastname, teams FROM cycling.cyclist_career_teams WHERE
 id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

Restriction: a relation that references the partition key can only use an equality operator — = or IN. For more details about the IN operator, see Examples below.

Filtering on a clustering column

You can use a relation on a clustering column only if it is preceded by relations that reference all the elements of the partition key.

Example:

```
CREATE TABLE cycling.cyclist_points (id UUID, firstname text, lastname text,
 race_title text, race_points int, PRIMARY KEY (id, race_points ));

SELECT sum(race_points) FROM cycling.cyclist_points WHERE
 id=e3b19ec4-774a-4d1c-9e5a-decec1e30aac AND race_points > 7;
```

Output:

```


 system.sum(race_points)
--------------------------
                      195

(1 rows)
```

In Cassandra 3.6 and later, it is possible to filter only on a non-indexed cluster column if `ALLOW FILTERING` is included. The table definition is included in this example to show that `race_start_date` is a clustering column without a secondary index.

Example:

```
CREATE TABLE cycling.calendar (
    race_id int,
    race_name text, race_start_date timestamp, race_end_date timestamp,
 PRIMARY KEY (race_id, race_start_date, race_end_date));

SELECT * FROM cycling.calendar WHERE race_start_date='2015-06-13' ALLOW
 FILTERING;
```

Output:

```
race_id | race_start_date                 | race_end_date
 | race_name
---------+---------------------------------
+---------------------------------+----------------
     102 | 2015-06-13 07:00:00.000000+0000 | 2015-06-13 07:00:00.000000+0000
 | Tour de Suisse
     103 | 2015-06-13 07:00:00.000000+0000 | 2015-06-17 07:00:00.000000+0000
 | Tour de France
```

It is possible to combine the partition key and a clustering column in a single relation. For details, see Comparing clustering columns.

Filtering on indexed columns

A WHERE clause in a SELECT on an indexed table must include at least one equality relation to the indexed column. For details, see Indexing a column.

Using the IN operator

Use `IN`, an equals condition operator, to list multiple possible values for a column. Thjs example selects two columns, `first_name` and `last_name`, from three rows having employee ids (primary key) 105, 107, or 104:

```
SELECT first_name, last_name FROM emp WHERE empID IN (105, 107, 104);
```

The list can consist of a range of column values separated by commas.

Using IN to filter on a compound or composite primary key

Use an `IN` condition on the last column of the partition key only when it is preceded by equality conditions for all preceding columns of the partition key. For example:

```
CREATE TABLE parts (part_type text, part_name text, part_num int, part_year
 text, serial_num text, PRIMARY KEY ((part_type, part_name), part_num,
 part_year));

SELECT * FROM parts WHERE part_type='alloy' AND part_name='hubcap' AND
 part_num=1249 AND part_year IN ('2010', '2015');
```

when using `IN`, you can omit the equality test for clustering columns other than the last. But this usage may require the use of `ALLOW FILTERING`, so its performance can be unpredictable. For example:

```
SELECT * FROM parts WHERE part_num=123456 AND part_year IN ('2010', '2015')
 ALLOW FILTERING;
```

CQL supports an empty list of values in the `IN` clause, useful in Java Driver applications when passing empty arrays as arguments for the `IN` clause.

When *not* to use IN

Under most conditions, using `IN` in relations on the partition key is not recommended. To process a list of values, the SELECT may have to query many nodes, which degrades performance. For example, consider a single local datacenter cluster with 30 nodes, a replication factor of 3, and a consistency level of `LOCAL_QUORUM`. A query on a single partition key query goes out to two nodes. But if the SELECT uses the `IN` condition, the operation can involve more nodes — up to 20, depending on where the keys fall in the token range.

Using `IN` for clustering columns is safer. See Cassandra Query Patterns: Not using the “in” query for multiple partitions for additional logic about using `IN`.

Filtering on collections

You can query a table containing a collection to retrieve the collection in its entirety. You can also index the collection column, and then use the `CONTAINS` condition in the `WHERE` clause to filter the data for a particular value in the collection, or `CONTAINS KEY` filter by key. This example features a collection of tags in the playlists table. When the tags are indexed. When you have indexed the tags, you can filter on 'blues' in the tags set.

```
SELECT album, tags FROM playlists WHERE tags CONTAINS 'blues';
```

```
album                | tags
---------------------+-----------------------
Tres Hombres         |     {"1973", "blues"}
```

After indexing the music venue map, you can filter on map values, such as 'The Fillmore':

```
SELECT * FROM playlists WHERE venue CONTAINS 'The Fillmore';
```

After indexing the collection keys in the venues map, you can filter on map keys.

```
SELECT * FROM playlists WHERE venue CONTAINS KEY '2014-09-22 22:00:00-0700';
```

Filtering a map's entries

Follow this example to query a table containing a collection to retrieve rows based on map entries. (This method only works for maps.)

```
CREATE INDEX blist_idx ON cycling.birthday_list (ENTRIES(blist));
```

This query finds all cyclists who are 23 years old based on their entry in the blist map of the table birthday_list.

```
SELECT * FROM cycling.birthday_list WHERE blist['age'] = '23';
```

Filtering a full frozen collection

This example presents a query on a table containing a `FROZEN` collection (set, list, or map). The query retrieves rows that fully match the collection's values.

```
CREATE INDEX rnumbers_idx ON cycling.race_starts (FULL(rnumbers));
```

The following SELECT finds any cyclist who has 39 Pro wins, 7 Grand Tour starts, and 14 Classic starts in a frozen list.

```
SELECT * FROM cycling.race_starts WHERE rnumbers = [39,7,14];
```

Range relations

The TOKEN function may be used for range queries on the partition key.

Cassandra supports greater-than and less-than comparisons, but for a given partition key, the conditions on the clustering column are restricted to the filters that allow Cassandra to select a contiguous set of rows.

For example:

```
CREATE TABLE ruling_stewards (
  steward_name text,
  king text,
  reign_start int,
  event text,
  PRIMARY KEY (steward_name, king, reign_start)
```

```
);
```

This query constructs a filter that selects data about stewards whose reign started by 2450 and ended before 2500. If `king` were not a component of the primary key, you would need to create an index on king to use this query:

```
SELECT * FROM ruling_stewards
  WHERE king = 'Brego'
  AND reign_start >= 2450
  AND reign_start < 2500 ALLOW FILTERING;
```

The output:

```
 steward_name | king  | reign_start | event
--------------+-------+-------------+--------------------
      Boromir | Brego |        2477 |   Attacks continue
       Cirion | Brego |        2489 | Defeat of Balchoth

(2 rows)
```

To allow Cassandra to select a contiguous set of rows, the WHERE clause must apply an equality condition to the `king` component of the primary key. The ALLOW FILTERING clause is also required. `ALLOW FILTERING` provides the capability to query the clustering columns using any condition when the query is not part of a production application.

CAUTION:

Only use `ALLOW FILTERING` for development! When you attempt a potentially expensive query, such as searching a range of rows, Cassandra displays this message:

```
Bad Request: 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.
```

To run this type of query, use the `ALLOW FILTERING` clause, and restrict the output to n rows using the `LIMIT n` clause. This is recommended to reduce memory used. For example:

```
Select * FROM ruling_stewards
  WHERE king = 'none'
  AND reign_start >= 1500
  AND reign_start < 3000 LIMIT 10 ALLOW FILTERING;
```

CAUTION: Using `LIMIT` does not prevent all problems caused by `ALLOW FILTERING`. In this example, if there are no entries without a value for *king*, the SELECT scans the entire table, no matter what `LIMIT` is.

Comparing clustering columns

The partition key and clustering columns can be grouped and compared to values for scanning a partition. For example:

```
SELECT * FROM ruling_stewards WHERE (steward_name, king) = ('Boromir',
 'Brego');
```

The syntax used in the `WHERE` clause compares records of `steward_name` and `king` as a tuple against the `Boromir, Brego` tuple.

Using compound primary keys and sorting results

`ORDER BY` clauses can only work on a single column. That column must be the second column in a compound `PRIMARY KEY`. This also applies to tables with more than two column components in the

primary key. Ordering can be done in ascending or descending order using the ASC or DESC keywords (default is ascending).

In the ORDER BY clause, refer to a column using the actual name, not an alias.

For example, set up the playlists table (which uses a compound primary key), insert the example data, and use this query to get information about a particular playlist, ordered by song_order. You do not need to include the ORDER BY column in the select expression.

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
  ORDER BY song_order DESC LIMIT 50;
```

Output:

```
id          | song_order | album                | artist         | song_id     | title
------------+------------+----------------------+----------------+-------------+---------------------
62c36092... |          4 | No One Rides for Free |      Fu Manchu | 7db1a490... |            Ojo Rojo
62c36092... |          3 |            Roll Away | Back Door Slam | 2b09185b... | Outside Woman Blues
62c36092... |          2 |         We Must Obey |      Fu Manchu | 8a172618... |    Moving in Stereo
62c36092... |          1 |         Tres Hombres |         ZZ Top | a3e63f8f... |           La Grange
```

Or, create an index on playlist artists, and use this query to get titles of Fu Manchu songs on the playlist:

```
CREATE INDEX ON playlists(artist)

SELECT album, title FROM playlists WHERE artist = 'Fu Manchu';
```

Output:

```
album                  | title
-----------------------+-------------------
We Must Obey           | Moving in Stereo
No One Riddes for Free | Ojo Rojo
```

Displaying rows from an unordered partitioner with the TOKEN function

Use the TOKEN function to display rows based on the partition key hash value of the clustering key, for a single partition table. Selecting a slice using TOKEN values will only work with clusters that use the ByteOrderedPartitioner .

For example, create this table:

```
CREATE TABLE cycling.last_3_days (
  race_name text,
  year timestamp,
  rank int,
  cyclist_name text,
  PRIMARY KEY (year, rank, cyclist_name));
```

After inserting data, SELECT using the TOKEN function to find the data using the partition key.

```
SELECT * FROM cycling.last_3_days WHERE TOKEN(year) < TOKEN('2015-05-26')
 AND year IN ('2015-05-24','2015-05-25');
```

Computing aggregates

Cassandra provides standard built-in functions that return aggregate values to SELECT statements.

Using COUNT() to get the non-NULL value count for a column

A SELECT expression using COUNT(column_name) returns the number of non-NULL values in a column.

For example, count the number of last names in the `cyclist_name` table:

```
SELECT COUNT(lastname) FROM cycling.cyclist_name;
```

Getting the number of matching rows and aggregate values with COUNT()

A `SELECT` expression using `COUNT(*)` returns the number of rows that matched the query. You can use `COUNT(1)` to get the same result. `COUNT(*)` or `COUNT(1)` can be used in conjunction with other aggregate functions or columns.

This example returns the number of rows in the users table:

```
SELECT COUNT(*) FROM users;
```

This example counts the number of rows and calculates the maximum value for *points* in the users table:

```
SELECT name, max(points), COUNT(*) FROM users;
```

Getting maximum and minimum values in a column

A `SELECT` expression using `MAX(`*`column_name`*`)` returns the maximum value in a column. If the column's datatype is numeric (`bigint`, `decimal`, `double`, `float`, `int`, `smallint`), this is the highest value.

```
SELECT MAX(points) FROM cycling.cyclist_category;
```

Output:

```
 system.max(points)
--------------------
               1324
```

`MIN` returns the minimum value. If the query includes a `WHERE` clause, `MAX` or `MIN` returns the largest or smallest value from the rows that satisfy the `WHERE` condition.

```
SELECT category, MIN(points) FROM cycling.cyclist_category WHERE category =
 'GC';
```

Output:

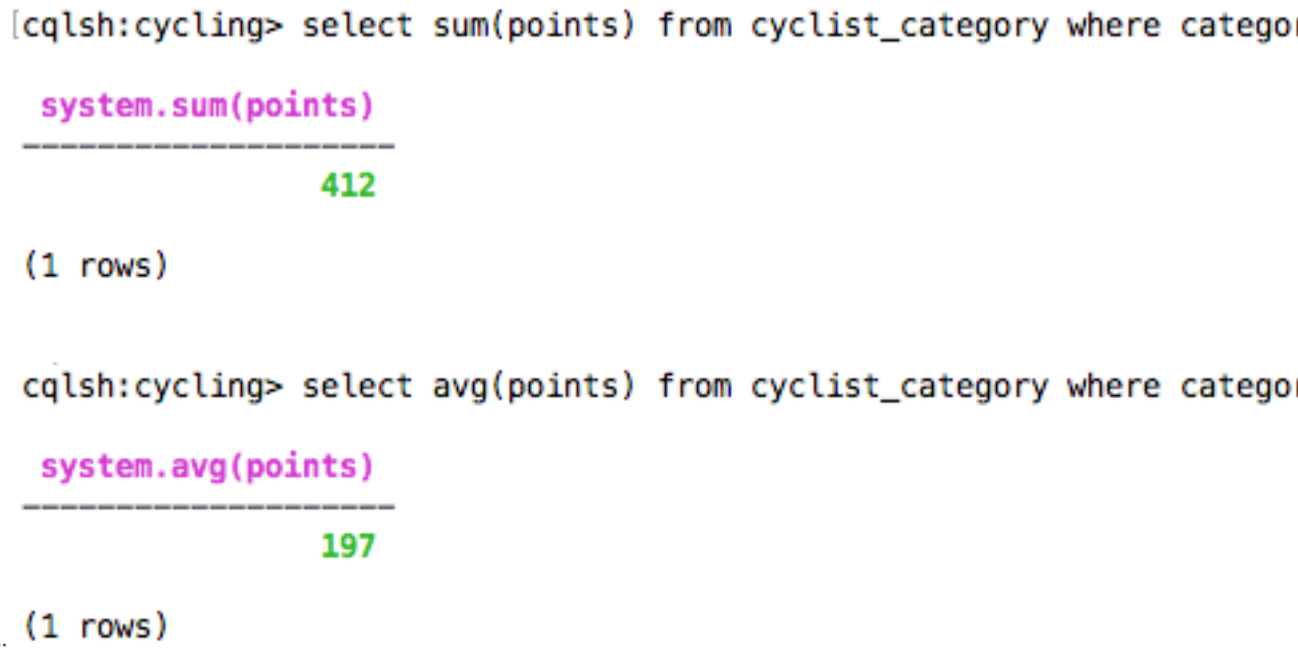
```
 category | system.min(points)
----------+--------------------
       GC |               1269
```

Note: If the column referenced by `MAX` or `MIN` has an `ascii` or `text` datatype, these functions return the last or first item in an alphabetic sort of the column values. If the specified column has datatype `date` or `timestamp`, these functions return the most recent or least recent times/dates. If the specified column has `null` values, the `MIN` function ignores it.

Note: Cassandra does not return a null value as the `MIN`.

Getting the sum or average of a column of numbers

Cassandra computes the sum or average of all values in a column when SUM or AVG is used in the SELECT



statement:

Note: If any of the rows returned has a null value for the column referenced for AVG aggregation, Cassandra includes that row in the row count, but uses a zero value to calculate the average.

Note: The sum and avg functions do not work with `text`, `uuid` or `date` fields.

Retrieving the date/time a write occurred

The WRITETIME function applied to a column returns the date/time in microseconds at which the column was written to the database.

R\For example, to retrieve the date/time that a write occurred to the *first_name* column of the user whose last name is Jones:

```
SELECT WRITETIME (first_name) FROM users WHERE last_name = 'Jones';

 writetime(first_name)
-----------------------
 1353010594789000
```

The WRITETIME output in microseconds converts to November 15, 2012 at 12:16:34 GMT-8.

Retrieving the time-to-live of a column

The time-to-live (TTL) value of a cell is the number of seconds before the cell is marked with a tombstone. You can set the TTL for a single cell, a column, or a column family. For example:

```
INSERT INTO cycling.calendar (race_id, race_name, race_start_date,
 race_end_date) VALUES (200, 'placeholder', '2015-05-27', '2015-05-27')
 USING TTL;
UPDATE cycling.calendar USING TTL 300 SET race_name = 'dummy' WHERE race_id
 = 200 AND race_start_date = '2015-05-27' AND race_end_date = '2015-05-27';
```

After inserting the TTL, you can use SELECT statement to check its current value:

```
SELECT TTL(race_name) FROM cycling.calendar WHERE race_id=200;
```

Output:

```
 ttl(race_name)
----------------
            276

(1 rows)
```

TRUNCATE

Remove all data from a table.

Synopsis

```
TRUNCATE [ TABLE ] [ keyspace_name.table_name ]
```

Markup	Indicates
`BOLD UPPERCASE`	Literal String
`Italics`	String to be replaced with a specific value for this operation
`bold lowercase`	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

The `TRUNCATE` statement removes all data from the specified table immediately and irreversibly, and removes all data from any materialized views derived from that table.

Examples

To remove all data from a table without dropping the table:

1. If necessary, use the cqlsh CONSISTENCY on page 114 command to set the consistency level to `ALL`.
2. Use nodetool status or some other tool to make sure all nodes are up and receiving connections.
3. Use `TRUNCATE` or `TRUNCATE TABLE`, followed by the table name. For example:

```
TRUNCATE user_activity;
```

```
TRUNCATE TABLE menu_item;
```

Note: TRUNCATE sends a JMX command to all nodes, telling them to delete SSTables that hold the data from the specified table. If any of these nodes is down or doesn't respond, the command fails and outputs a message like the following:

```
cqlsh:mykeyspace> truncate menu_item;
Unable to complete request: one or more nodes were unavailable.
```

UPDATE

Update columns in a row.

Synopsis

```
UPDATE [ keyspace_name. ] table_name
[ USING { TTL time_value | TIMESTAMP timestamp_value } ]
SET assignment [, assignment ] [ . . . ]
WHERE row_specification
[ { IF EXISTS | condition [ AND condition ] [ . . . ] } ]
```

Markup	Indicates
BOLD UPPERCASE	Literal String
Italics	String to be replaced with a specific value for this operation
bold lowercase	Parameter with its own set of elements
Square brackets []	Optional element or set of elements
Elements within brackets separated by pipe/vertical line { | }	Set of options, only one can be used
Ellipsis (. . .)	Preceding value can be repeated
Semi-colon (;)	Terminates the command

Description

UPDATE writes one or more column values to a row in a Cassandra table. If the specified row does not exist, the command creates it. All UPDATEs within the same partition key are applied atomically and in isolation.

The USING clause can add a timestamp or a 'time to live' value to the row. You cannot specify these options on counter columns.

Assign new values to the row's columns in the SET clause. UPDATE cannot update the values of a row's primary key fields. To update a counter column value in a counter table, specify the increment or decrement to the counter column.

Note: Unlike the INSERT command, the UPDATE command supports counters. Otherwise, the UPDATE and INSERT operations are identical.

The WHERE clause specifies the row or rows to be updated. To specify a row,the WHERE clause must specify a value for each column of its primary key. You can use the IN to specify more than one column value, but only for the last column of the primary key.

The IF EXISTS or IF options make the UPDATE into a lightweight transaction:

```
UPDATE cycling.cyclist_name
SET comments ='='Rides hard, gets along with others, a real winner'
```

```
WHERE id = fb372533-eb95-4bb4-8685-6ef61e994caa IF EXISTS;
```

Use the IF keyword followed by a condition to be met for the update to succeed. Using an IF condition incurs a performance hit associated with using Paxos internally to support linearizable consistency.

The UPDATE command does not return any result unless it includes IF EXISTS.

Parameters

keyspace_name

The name of the keyspace containing the table to be updated. Not needed if the keyspace has been set for the session with the USE command.

table_name

The name of the table to be updated

time_value

The value for TTL is a number of seconds. Column values in a command marked with TTL are automatically marked as deleted (with a tombstone) after the specified amount of time. The TTL applies to the marked column values, not the column itself. Any subsequent update of the column resets the value to the TTL specified in the update. By default, values never expire. You cannot set a time_value for data in a counter column.

timestamp_value

If TIMESTAMP is used, the inserted column is marked with its value – a timestamp in microseconds. If a TIMESTAMP value is not set, Cassandra uses the time (in microseconds) that the update occurred to the column.

assignment

Asigns a value to an existing element. Can be one of:

```
column_name = column_value
set_or_list_item = set_or_list_item ( + | - ) ...
map_name = map_name ( + | - ) ...
map_name = map_name ( + | - ) { map_key : map_value, ... }
column_name [ term ] = value
counter_column_name = counter_column_name ( + | - ) integer
```

- set is:

  ```
  { literal, literal, . . . }
  ```

  (note the use of curly brackets)
- A list is:

  ```
  [ literal, literal ]
  ```

  (note the use of square brackets)
- A map is:

  ```
  { literal : literal, literal : literal, . . . }
  ```
- A term is:

  ```
  [ list_index_position | [ key_value ]
  ```

row_specification

The WHERE clause must identify the row or rows to be updated by primary key:

- To specify one row, use *primary_key_name* = *primary_key_value*. If the primary key is a combination of elements, follow this with AND *primary_key_name* = *primary_key_value* .... The WHERE clause must specify a value for every component of the primary key.
- To specify more than one row, use *primary_key_name* IN (*primary_key_value*, *primary_key_value* …) As in the previous option, each *primary_key_name* and *primary_key_value* must match the composition of the primary key.

Note: To update a static column, only the partition key must be specified.

IF EXISTS / IF condition

An IF clause can limit the command's action on rows that match the WHERE clause. Two options for IF:

- Use IF EXISTS to make the UPDATE fail if there are no rows that match the WHERE conditions.
- Use IF to specify one or more conditions that must test true for the values in the specified row or rows.

If an IF is used, the command returns a result to standard output. See Conditionally updating columns for examples.

Examples: updating a column

Update a column in several rows at once:

```
UPDATE users
  SET state = 'TX'
  WHERE user_uuid
  IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
    06a8913c-c0d6-477c-937d-6c1b69a95d43,
    bc108776-7cb5-477f-917d-869c12dfffa8);
```

CQL supports an empty list of values in the IN clause, useful in Java Driver applications when passing empty arrays as arguments for the IN clause.

Update several columns in a single row:

```
UPDATE cycling.cyclist_teams
  SET firstname = 'Marianne,
  lastname = 'VOS'
  WHERE id = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

Update a row in a table with a complex primary key:

You need to specify all keys in a table having compound and clustering columns. For example, update the value of a column in a table having a compound primary key, userid and url:

```
UPDATE excelsior.clicks USING TTL 432000
  SET user_name = 'bob'
  WHERE userid=cfd66ccc-d857-4e90-b1e5-df98a3d40cd6 AND
    url='http://google.com';
UPDATE Movies SET col1 = val1, col2 = val2 WHERE movieID = key1;
UPDATE Movies SET col3 = val3 WHERE movieID IN (key1, key2, key3);
UPDATE Movies SET col4 = 22 WHERE movieID = key4;
```

Note: In Cassandra3.0 and earlier, you cannot insert any value larger than 64K bytes into a clustering column.

Updating a counter column

To update a counter column value in a counter table, specify the increment or decrement to the current value of the counter column.

```
UPDATE cycling.popular_count SET popularity = popularity + 2 WHERE id =
 6ab09bec-e68e-48d9-a5f8-97e6fb4c9b47;
```

To use a lightweight transaction on a counter column to ensure accuracy, put one or more counter updates in the batch statement.

Updating a collection set

To add an element to a set, use the `UPDATE` command and the addition (+) operator:

```
UPDATE cycling.cyclist_career_teams
SET teams = teams + {'Team DSB - Ballast Nedam'} WHERE
 id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

To remove an element from a set, use the subtraction (-) operator:

```
UPDATE cycling.cyclist_career_teams
SET teams = teams - {'DSB Bank - Nederland bloeit'} WHERE
 id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

To remove all elements from a set:

```
UPDATE cycling.cyclist_career_teams
SET teams = {} WHERE id=5b6962dd-3f90-4c93-8f61-eabfa4a803e2;
```

Updating a collection map

To set or replace map data, enclose the values in map collection syntax: strings in curly brackets, separated by a colon.

```
UPDATE cycling.upcoming_calendar
SET description = description + {'Criterium du Dauphine' : 'Easy race'}
 WHERE year = 2015;
```

To update or set a specific element - for example, update a map named `events` to add a new race to the calendar:

```
UPDATE cycling.upcoming_calendar
SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2016 AND month =
 06;
```

To set the a TTL for each map element:

```
UPDATE cycling.upcoming_calendar USING TTL <ttl_value>
SET events[2] = 'Vuelta Ciclista a Venezuela' WHERE year = 2016 AND month =
 06;
```

In Cassandra 2.1.1 and later, you can update the map by adding one or more elements separated by commas:

```
UPDATE cycling.upcoming_calendar
SET description = description + {'Criterium du Dauphine' : 'Easy race',
 'Tour du Suisse' : 'Hard uphill race'}
WHERE year = 2015 AND month = 6;
```

Remove elements from a map in the same way using - instead of +.

Caution: about updating sets and maps

CQL supports alternate methods for updating sets and maps. These alternatives may seem to accomplish the same tasks, but Cassandra handles them differently in important ways.

For example: CQL provides a straightforward method for creating a new row containing a collection map:

```
UPDATE cycling.upcoming_calendar
SET description =
{'Criterium du Dauphine' : 'Easy race',
 'Tour du Suisse' : 'Hard uphill race'}
WHERE year = 2015 AND month = 6;
```

The easiest way to add a new entry to the map is to use the + operator as described above:

```
UPDATE cycling.upcoming_calendar
SET description = description + { 'Tour de France' : 'Very competitive'}
WHERE year = 2015 AND month = 6;
```

You may, however, try to add the new entry with a command that overwrites the first two and adds the new one:

```
UPDATE cycling.upcoming_calendar
SET description =
{'Criterium du Dauphine' : 'Easy race',
 'Tour du Suisse' : 'Hard uphill race',
 'Tour de France' : 'Very competitive'}
WHERE year = 2015 AND month = 6;
```

These two statements seem to do the same thing. But behind the scenes, Cassandra process the second statement by deleting the entire collection and replacing it with a new collection containing three entries. This creates tombstones for the deleted entries, even though they are identical to the entries in the new map collection. If your code updates all map collections this way, it generates many tombstones, which may ultimately slow the system down.

The examples above use map collections, but the same caution applies to updating collection sets.

Using a collection list

To insert values into the list:

```
UPDATE cycling.upcoming_calendar
SET events = ['Criterium du Dauphine','Tour de Suisse'];
```

To prepend an element to the list, enclose it in square brackets and use the addition (+) operator:

```
UPDATE cycling.upcoming_calendar
SET events = [ 'Tour de France' ] + events WHERE year=2015 AND month=06;
```

To append an element to the list, switch the order of the new element data and the list name:

```
UPDATE cycling.upcoming_calendar
SET events = events + [ 'Tour de France' ] WHERE year=2017 AND month=05;
```

To add an element at a particular position, use the list index position in square brackets:

```
UPDATE cycling.upcoming_calendar
SET events[4] = 'Tour de France' WHERE year=2016 AND month=07;
```

To remove all elements having a particular value, use the subtraction operator (-) and put the list value in square brackets:

```
UPDATE cycling.upcoming_calendar
SET events = events - ['Criterium du Dauphine'] WHERE year=2016 AND
 month=07;
```

To update data in a collection column of a user-defined type, enclose components of the type in parentheses within the curly brackets, as shown in "Using a user-defined type."

CAUTION: The Java List Index is not thread safe. The set or map collection types are safer for updates.

Caution: about updating collections

Updating a UDT with non-collection fields

In Cassandra 3.6 and later, to change the value of an individual field value in a user-defined type with non-collection fields, use the UPDATE command:

```
UPDATE cyclist_stats SET basics.birthday = '2000-12-12' WHERE id =
 220844bf-4860-49d6-9a4b-6b5d3a79cbfb;
```

Conditionally updating columns

In Cassandra 2.0.7 and later, you can conditionally update columns using IF or IF EXISTS.

Add IF EXISTS to the command to ensure that the operation is not performed if the specified row does not exist:

```
UPDATE cycling.cyclist_id SET age = 28 WHERE lastname = 'WELTEN' and
 firstname = 'Bram' IF EXISTS;
```

Without IF EXISTS, the command proceeds with no standard output. If IF EXISTS returns true (if a row with this primary key does exist), standard output displays a table like the following:

```
 [applied]
-----------
      True
```

If no such row exists, however, the condition returns FALSE and the command fails. In this case, standard output looks like:

```
 [applied]
-----------
     False
```

Use IF condition to apply tests to one or more column values in the selected row:

```
UPDATE cyclist_id SET id = 15a116fc-b833-4da6-ab9a-4a3775750239 where
 lastname = 'WELTEN' and firstname = 'Bram' IF age = 18;
```

If all the conditions return TRUE, standard output is the same as if IF EXISTS returned true (see above). If any of the conditions fails, standard output displays `False` in the `[applied]` column and also displays information about the condition that failed:

```
 [applied] | age
-----------+-----
     False |  18
```

Conditional updates are examples of "lightweight transactions." They incur a non-negligible performance cost and should be used sparingly.

USE

Connect the client session to a keyspace.

Synopsis

```
USE keyspace_name
```

- Uppercase means literal
- Lowercase means not literal
- Italics mean optional
- The pipe (|) symbol means OR or AND/OR
- Ellipsis (...) means repeatable
- Orange (and) means not literal, indicates scope

A semicolon that terminates CQL statements is not included in the synopsis.

Description

A USE statement identifies the keyspace that contains the tables to query for the current client session. All subsequent operations on tables and indexes are in the context of the named keyspace, unless otherwise specified or until the client connection is terminated or another USE statement is issued.

To use a case-sensitive keyspace, enclose the keyspace name in double quotation marks.

Example

```
USE PortfolioDemo;
```

Continuing with the example of checking created keyspaces:

```
USE "Excalibur";
```