
# Apache Cassandra:
## Core Concepts, Skills, and Tools

### Introducing the Cassandra Data Model and Cassandra Query Language
Exercise Workbook

Artem Chebotko
Leo Schuman
October, 2014

# Exercise 1: Model sample data as column families

**In this exercise, you will:**

- Explore sample user data
- Organize data into column families

## Steps

### Explore sample user data

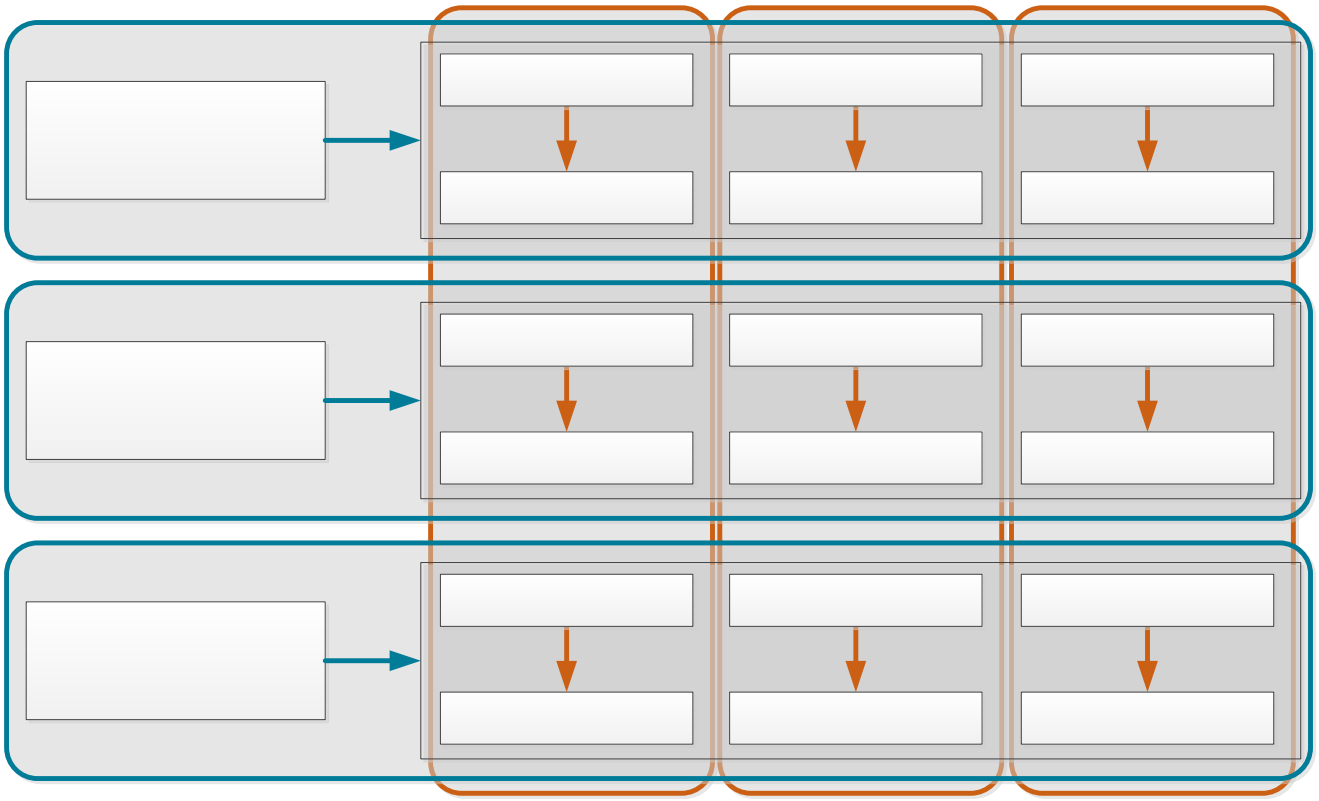
1. Shown below, understand the tabular data that describes users.

| Name | DOB | Email | Join Date |
|---|---|---|---|
| John | 12/01/1986 | john@data.org | 03/08/2014 |
| Mary | 12/01/1986 | mary@data.org | 03/08/2014 |
| John | 02/18/1979 | john@data.edu | 01/01/2013 |

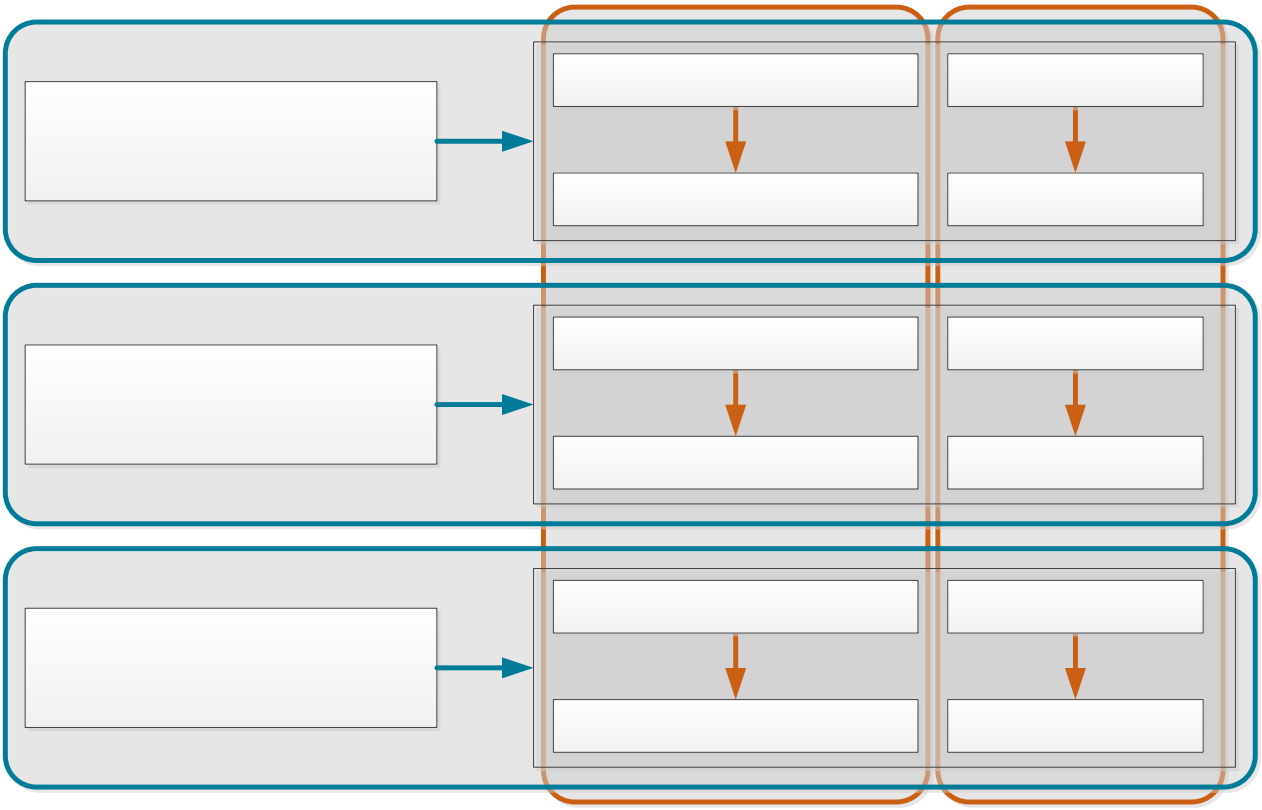
2. In the space below or a separate sheet, list the fields above which uniquely identify a user.

## Organize data into column families

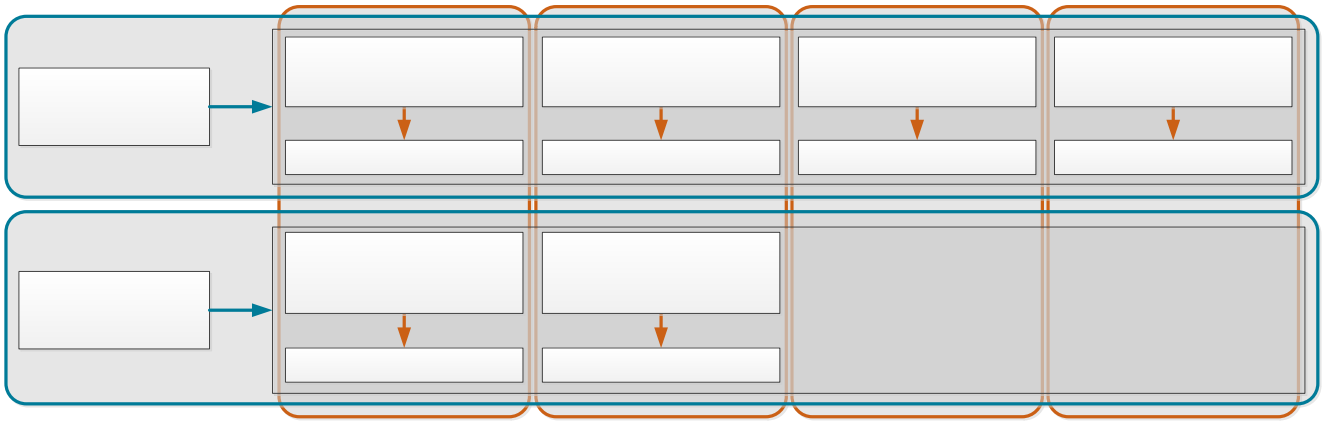
3. In the template below or a separate sheet, fill in blanks to show how a column family can store user data in rows, where each row is identified by an email.



4. In the template below or a separate sheet, fill in blanks to show how a column family can store user data in rows, where each row is identified by a name and a date of birth.



5. In the template below or a separate sheet, fill in blanks to show how a column family can store user data in rows, where each row is identified by a join date.



END OF EXERCISE

# Exercise 2: Represent column families as tables

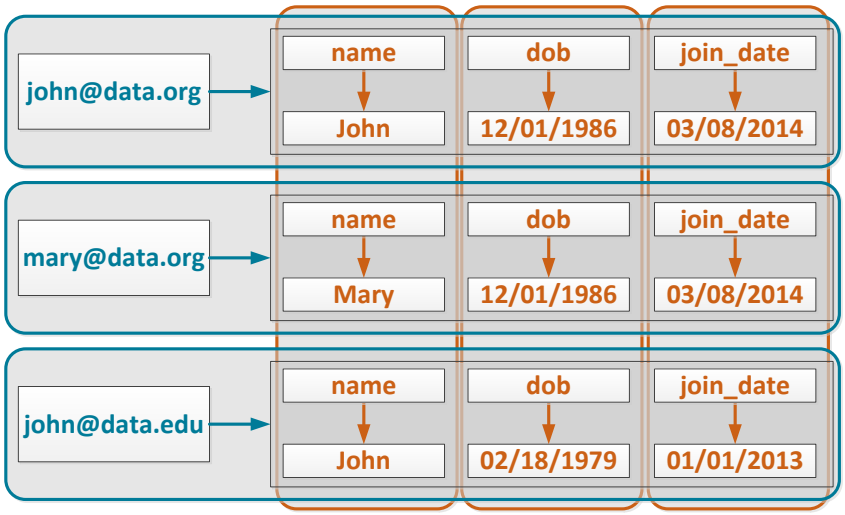
**In this exercise, you will:**

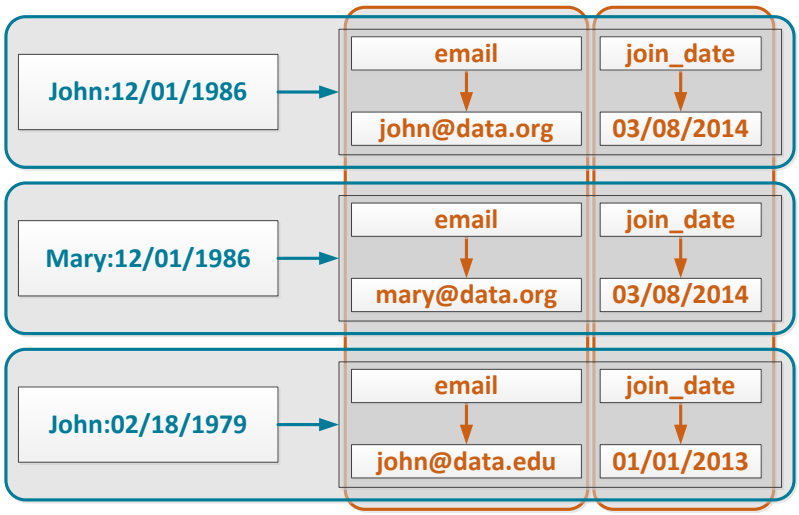
- Explore sample column families
- Represent column families as tables

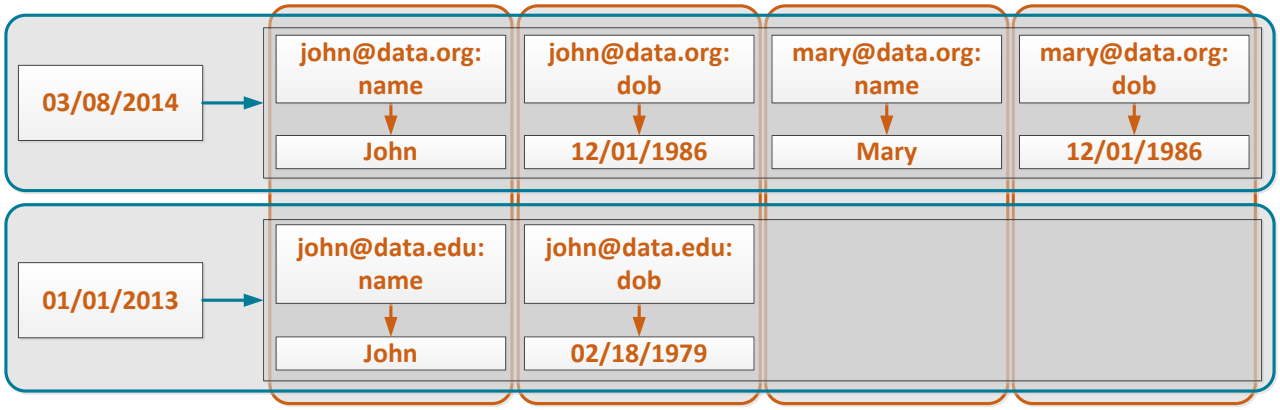
## Steps

### Explore sample column families

1. Shown below, consider sample column families that describe users.






2. In the space below or a separate sheet, list a row key (simple or composite) and column keys (simple or composite) for each column family.

## Represent column families as tables

3. In the templates below or a separate sheet, show how user data can be stored in tables.

users_by_email

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

users_by_name_and_dob

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

users_by_join_date

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

4. In the space below or a separate sheet, list a partition key (simple or composite), clustering columns, and a primary key for each table.

END OF EXERCISE

# Demo 3: How to launch and use *cqlsh*

**In this demo, we will:**

- Show how to start *cqlsh*
- Show how to execute a CQL script using *cqlsh*
- Show how to explore a database schema and instance

## Steps

### Show how to start *cqlsh*

1. In a terminal window, start *cqlsh*.

```
ccm node1 cqlsh
```

*This demonstration can be run in either the cascor cluster or tarball installation.*

### Show how to execute a CQL script using *cqlsh*

2. In *cqlsh*, using the *SOURCE* command, execute script *userdb.cql* for this demo, located at *~/cascor/intro-dm-cql/demo-3*. The script content is shown below.

```
SOURCE '~/cascor/intro-dm-cql/demo-3/userdb.cql';
```

```
CREATE KEYSPACE userdb
WITH replication = {'class': 'SimpleStrategy',
'replication_factor' : 1};

USE userdb;

CREATE TABLE users_by_email (
  name VARCHAR,
  dob TIMESTAMP,
  email VARCHAR,
  join_date TIMESTAMP,
  PRIMARY KEY (email)
);
```

```
CREATE TABLE users_by_name_and_dob (
  name VARCHAR,
  dob TIMESTAMP,
  email VARCHAR,
  join_date TIMESTAMP,
  PRIMARY KEY ((name,dob))
);

CREATE TABLE users_by_join_date (
  name VARCHAR,
  dob TIMESTAMP,
  email VARCHAR,
  join_date TIMESTAMP,
  PRIMARY KEY (join_date, email)
);

INSERT INTO users_by_email (name, dob, email, join_date)
VALUES ('John', '1986-12-01', 'john@data.org', '2014-03-
08');
INSERT INTO users_by_email (name, dob, email, join_date)
VALUES ('Mary', '1986-12-01', 'mary@data.org', '2014-03-
08');
INSERT INTO users_by_email (name, dob, email, join_date)
VALUES ('John', '1979-02-18', 'john@data.edu', '2013-01-
01');

INSERT INTO users_by_name_and_dob (name, dob, email,
join_date) VALUES ('John', '1986-12-01', 'john@data.org',
'2014-03-08');
INSERT INTO users_by_name_and_dob (name, dob, email,
join_date) VALUES ('Mary', '1986-12-01', 'mary@data.org',
'2014-03-08');
INSERT INTO users_by_name_and_dob (name, dob, email,
join_date) VALUES ('John', '1979-02-18', 'john@data.edu',
'2013-01-01');

INSERT INTO users_by_join_date (name, dob, email,
join_date) VALUES ('John', '1986-12-01', 'john@data.org',
'2014-03-08');
INSERT INTO users_by_join_date (name, dob, email,
join_date) VALUES ('Mary', '1986-12-01', 'mary@data.org',
'2014-03-08');
INSERT INTO users_by_join_date (name, dob, email,
join_date) VALUES ('John', '1979-02-18', 'john@data.edu',
'2013-01-01');
```

## Show how to explore a database schema and instance

3. In *cqlsh*, run *USE* and *DESCRIBE KEYSPACE* commands for *userdb*.

4. In *cqlsh*, run *DESCRIBE TABLES* and *DESCRIBE TABLE* commands for each table.

5. In *cqlsh*, run *SELECT* statement to retrieve all rows from each table.

6. In *cqlsh*, run *EXIT* command.

END OF DEMO

# Exercise 4: Create a keyspace and tables using *cqlsh*

**In this exercise, you will:**

- Create a keyspace
- Create tables in a keyspace
- Populate tables from CSV files
- Execute simple queries

## Steps

### Create a keyspace

1. In the terminal window, navigate to the *cascor/intro-dm-cql* directory.

```
cd ~/cascor/intro-dm-cql
```

2. From the *cascor/intro-dm-cql* directory, start up *cqlsh*.

```
ccm node1 cqlsh
```

3. In *cqlsh*, create a new keyspace named *musicdb* for a music database, with the simple replication strategy, and a replication factor of 3.

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

4. In *cqlsh*, use the DESCRIBE command to display information about the keyspace.

```
DESCRIBE KEYSPACE musicdb
```

## Create tables in a keyspace

5. In *cqlsh*, use the USE command to set the *musicdb* keyspace as the current default.

```
USE musicdb;
```

6. In *cqlsh*, create the tables for the *musicdb* keyspace by executing the CQL script.

```
SOURCE '~/cascor/intro-dm-cql/exercise-4/musicdb.cql'
```

7. Shown below, study and understand the table definitions that were created from the CQL script.

```
CREATE TABLE performer (
  name VARCHAR,
  type VARCHAR,
  country VARCHAR,
  style VARCHAR,
  founded INT,
  born INT,
  died INT,
  PRIMARY KEY (name)
);

CREATE TABLE performers_by_style (
  style VARCHAR,
  name VARCHAR,
  PRIMARY KEY (style, name)
);
```

```
CREATE TABLE album (
  title VARCHAR,
  year INT,
  performer VARCHAR,
  genre VARCHAR,
  tracks MAP<INT,VARCHAR>,
  PRIMARY KEY ((title, year))
);

CREATE TABLE albums_by_performer (
  performer VARCHAR,
  year INT,
  title VARCHAR,
  genre VARCHAR,
  PRIMARY KEY (performer, year, title)
) WITH CLUSTERING ORDER BY (year DESC, title ASC);

CREATE TABLE albums_by_genre (
  genre VARCHAR,
  performer VARCHAR,
  year INT,
  title VARCHAR,
  PRIMARY KEY (genre, performer, year, title)
) WITH CLUSTERING ORDER BY (performer ASC, year DESC,
title ASC);

CREATE TABLE albums_by_track (
  track_title VARCHAR,
  performer VARCHAR,
  year INT,
  album_title VARCHAR,
  PRIMARY KEY (track_title, performer, year, album_title)
) WITH CLUSTERING ORDER BY (performer ASC, year DESC,
album_title ASC);

CREATE TABLE tracks_by_album (
  album_title VARCHAR,
  year INT,
  performer VARCHAR STATIC,
  genre VARCHAR STATIC,
  number INT,
  track_title VARCHAR,
  PRIMARY KEY ((album_title, year), number)
);
```

8. In *cqlsh*, display the names of all tables in the *musicdb* keyspace.

```
DESCRIBE TABLES
```

## Populate tables from CSV files

9. In *cqlsh*, run the commands below one at a time to import data into the tables from CSV files.

*You may run into an intermittent bug that causes the COPY command to abort or to crash cqlsh, as reported in the JIRA ticket CASSANDRA-8351. This may show errors such as:*

line contains NULL byte
Aborting import at record #0. Previously-inserted values still present.

Segmentation fault (core dumped)

*Retry the previous command as necessary until it succeeds.*

```
COPY performer
     (name, type, country, style, founded, born, died)
FROM '~/cascor/intro-dm-cql/exercise-4/performer.csv'
WITH HEADER = 'true';

COPY performers_by_style (style, name)
FROM '~/cascor/intro-dm-cql/exercise-4/performers_by_style.csv'
WITH HEADER = 'true';

COPY album (title, year, performer, genre, tracks)
FROM '~/cascor/intro-dm-cql/exercise-4/album.csv'
WITH HEADER = 'true';

COPY albums_by_performer (performer, year, title, genre)
FROM '~/cascor/intro-dm-cql/exercise-4/albums_by_performer.csv'
WITH HEADER = 'true';

COPY albums_by_genre (genre, performer, year, title)
FROM '~/cascor/intro-dm-cql/exercise-4/albums_by_genre.csv'
WITH HEADER = 'true';

COPY albums_by_track (track_title, performer, year, album_title)
FROM '~/cascor/intro-dm-cql/exercise-4/albums_by_track.csv'
WITH HEADER = 'true';

COPY tracks_by_album
     (album_title, year, performer, genre, number, track_title)
FROM '~/cascor/intro-dm-cql/exercise-4/tracks_by_album.csv'
WITH HEADER = 'true';
```

## Execute simple queries

10. In *cqlsh*, test the following queries.

```
SELECT * FROM performer WHERE name = 'The Beatles';
```

```
SELECT * FROM performer WHERE name = 'John Lennon';
```

*Notice the differences in data that is stored for the band and the artist.*

11. In *cqlsh*, test the following queries.

```
SELECT *
FROM tracks_by_album
WHERE album_title = 'Revolver' AND year = 1966;
```

```
SELECT *
FROM album
WHERE title = 'Revolver' AND year = 1966;
```

*Notice how the same data is organized differently in these tables.*

END OF EXERCISE

# Exercise 5: Create tables using UUID, TIMEUUID, and COUNTER columns

**In this exercise, you will:**

- Select an existing keyspace
- Create additional tables with UUID, TIMEUUID, and COUNTER columns

## Steps

### Select an existing keyspace

1. In *cqlsh*, use the USE command to set the *musicdb* keyspace as the current default.

```
USE musicdb;
```

*Note that you may also use the keyspace name as a table name prefix. The USE command, however, adds convenience by setting a default keyspace so that no prefix is required.*

### Create additional tables with UUID, TIMEUUID, and COUNTER columns

2. In *cqlsh*, create table *user*.

```
CREATE TABLE user (
  id UUID,
  name VARCHAR,
  PRIMARY KEY (id)
);
```

3. In *cqlsh*, create table *track_ratings_by_user*.

```
CREATE TABLE track_ratings_by_user (
  user UUID,
  activity TIMEUUID,
  rating INT,
  album_title VARCHAR,
  album_year INT,
  track_title VARCHAR,
  PRIMARY KEY (user, activity)
) WITH CLUSTERING ORDER BY (activity DESC);
```

*Note that the rating activities for a user will be sorted in the descending order of their timestamps (most recent first) extracted from the TIMEUUID column.*

4. In *cqlsh*, create table *ratings_by_track*.

```
CREATE TABLE ratings_by_track (
  album_title VARCHAR,
  album_year INT,
  track_title VARCHAR,
  num_ratings COUNTER,
  sum_ratings COUNTER,
  PRIMARY KEY (album_title, album_year, track_title)
);
```

*Note that all non-counter columns are part of the primary key.*
*Also, note that an average rating can be computed as*
*avg_rating = sum_ratings / num_ratings.*

*We will populate these tables in a different exercise.*

END OF EXERCISE

# Exercise 6: Add user-defined type, alter tables, add collection column, and add secondary indexes

**In this exercise, you will:**

- Create a user-defined type
- Add and drop columns to an existing table
- Add secondary indexes to an existing table

## Steps

### Create a user-defined type

1. In *cqlsh*, set the *musicdb* keyspace as the current default.

```
USE musicdb;
```

2. In *cqlsh*, display information about available user-defined types in *musicdb*.

```
DESCRIBE TYPES
```

3. In *cqlsh*, create a new user-defined type called *track*.

```
CREATE TYPE track (
  album_title VARCHAR,
  album_year INT,
  track_title VARCHAR
);
```

4. In *cqlsh*, display information about types.

```
DESCRIBE TYPES
```

5. In *cqlsh*, display information about the user-defined type *track*.

```
DESCRIBE TYPE track
```

## Add columns to an existing table

6. In *cqlsh*, drop the columns *album_title*, *album_year*, and *track_title* for the table *track_ratings_by_user*.

```
ALTER TABLE track_ratings_by_user DROP album_title;
ALTER TABLE track_ratings_by_user DROP album_year;
ALTER TABLE track_ratings_by_user DROP track_title;
```

7. In *cqlsh*, add a new column called *song* to *the track_ratings_by_user* table with the user-defined type *track*.

```
ALTER TABLE track_ratings_by_user ADD song frozen<track>;
```

8. In *cqlsh*, display information about the table *track_ratings_by_user*.

```
DESCRIBE TABLE track_ratings_by_user
```

9. In cqlsh, display information about table *user*.

```
DESCRIBE TABLE user
```

10. In *cqlsh*, add two new columns to table *user*.

```
ALTER TABLE user ADD email VARCHAR;
```

```
ALTER TABLE user ADD preferences SET<VARCHAR>;
```

11. In *cqlsh*, display information about table *user*.

```
DESCRIBE TABLE user
```

## Add secondary indexes to an existing table

12. In *cqlsh*, create a secondary index on the *preferences* column in the table *user*.

```
CREATE INDEX user_preferences_key ON user (preferences);
```

13. In *cqlsh*, display information about table *user*.

```
DESCRIBE TABLE user
```

14. In *cqlsh*, display information about table *performer*.

```
DESCRIBE TABLE performer
```

15. In *cqlsh*, create two secondary indexes for table *performer*.

```
CREATE INDEX performer_country_key ON performer (country);
```

```
CREATE INDEX performer_style_key ON performer (style);
```

16. In *cqlsh*, display information about table *performer*.

```
DESCRIBE TABLE performer
```

17. In *cqlsh*, test the following query.

```
SELECT name
FROM performer
WHERE country = 'Iceland' AND style = 'Rock'
ALLOW FILTERING;
```

END OF EXERCISE

# Demo 7: How to launch and use DevCenter

**In this demo, we will:**

- Show how to start DevCenter
- Show how to connect to a Cassandra cluster and explore database objects
- Show how to create, edit, and execute a new CQL script

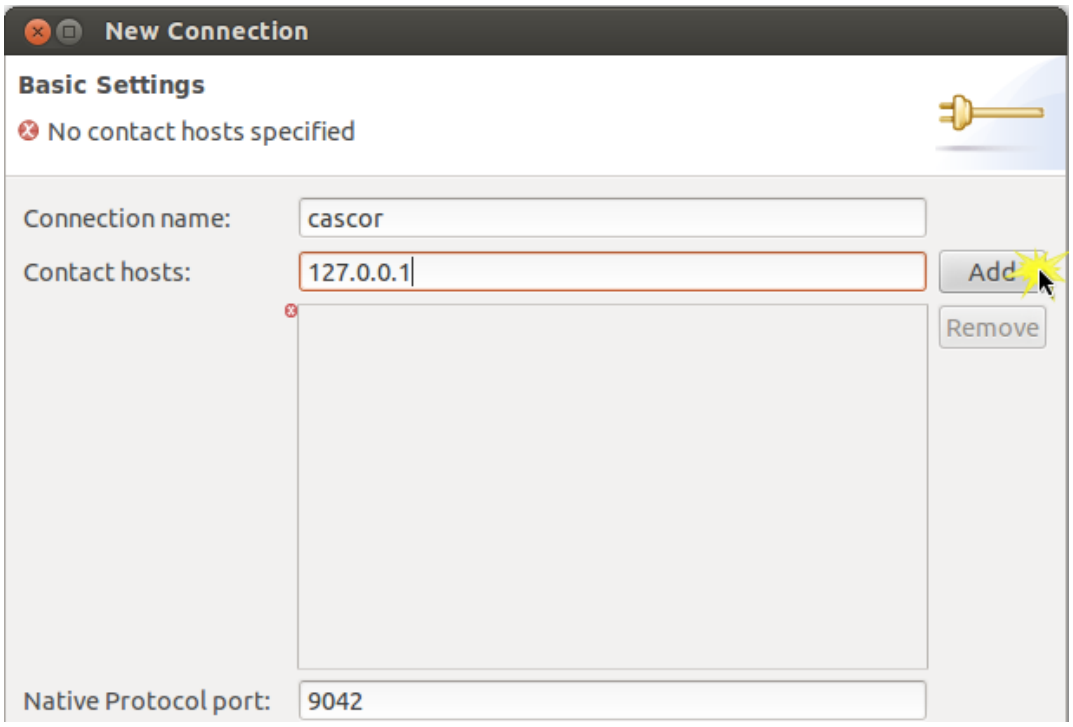
## Steps

### Show how to start DevCenter

1. In a terminal window, start *DevCenter*.

### Show how to connect to a Cassandra cluster

2. In *DevCenter*, using the *Connection Manager*, connect to the Cassandra cluster.



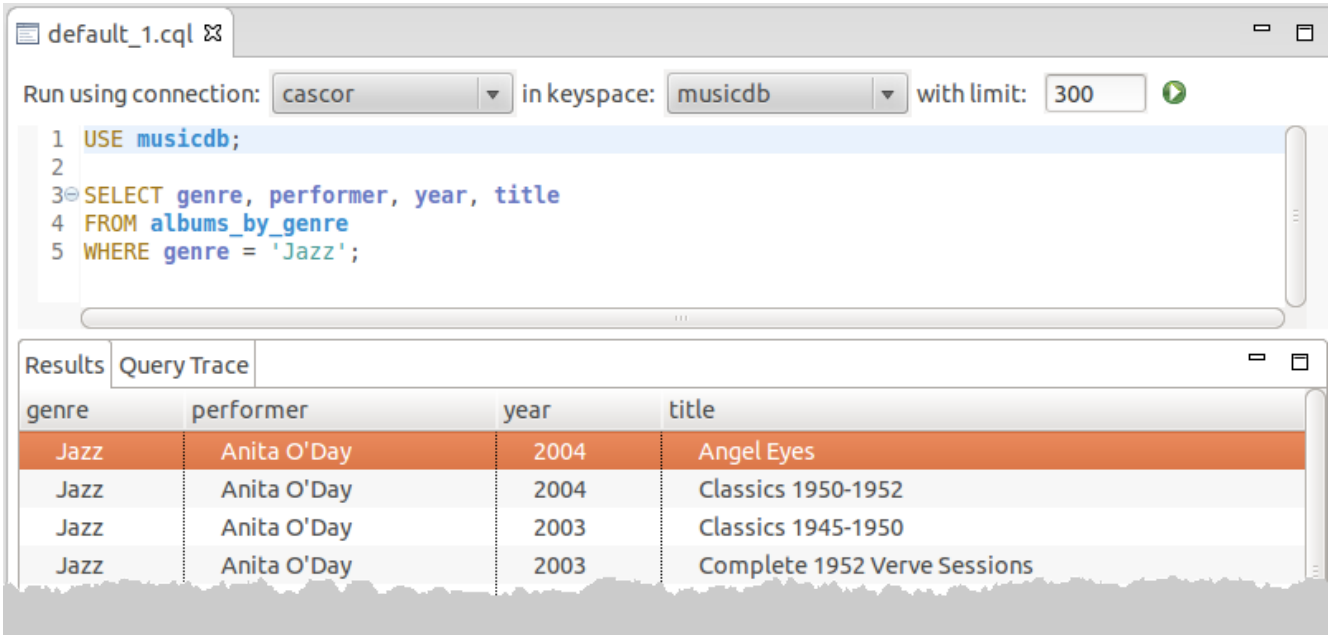
## Show how to explore existing database objects

3. In *DevCenter*, using the *Schema Explorer*, explore database objects in the *musicdb* keyspace.

## Show how to create, edit, and execute a new CQL script

4. In *DevCenter*, create a new CQL script, add *USE* and *SELECT* statements, and execute the script.

```
USE musicdb;
SELECT genre, performer, year, title FROM albums_by_genre
WHERE genre = 'Jazz';
```




*Useful DevCenter shortcuts:*
*<Ctrl>+<s> to save a current CQL script*
*<Alt>+<f>, then <.> to open an existing CQL script*
*<Ctrl>+<space> to bring up the autocomplete menu*
*<Shift>+<Ctrl>+<l> to open the key assist / shortcut menu*
*<Alt>+<F11> to execute the current CQL script*

END OF DEMO

# Exercise 8: Inserting and updating values using *DevCenter*

**In this exercise, you will:**

- Insert rows into tables
- Update rows in tables

## Steps

### Insert rows into tables

1. In *DevCenter*, create a new script with the following content.

```
USE musicdb;

INSERT INTO user (id, name, email) VALUES (12345678-abcd-
abcd-abcd-abcd12345678, 'John', 'john@datastax.com');

INSERT INTO user (id, name, email) VALUES (87654321-abcd-
abcd-abcd-abcd87654321, 'Mary', 'mary@datastax.com');

INSERT INTO user (id, name, email) VALUES (77777777-beef-
beef-beef-beef77777777, 'Joe', 'joe@datastax.com');
```

*Note that more readable UUIDs were selected on purpose.*

2. In *DevCenter*, execute the above script exactly one time.

*Make a note of the execution time for the three insert operations. You'll be comparing them with some other inserts in just a few steps.*

3. In *DevCenter*, create a new script with the query and execute it.

```
USE musicdb;

INSERT INTO user (id, name, email) VALUES (00000000-aaaa-
aaaa-aaaa-aaaa00000000, 'Ron', 'ron@datastax.com')
IF NOT EXISTS;

INSERT INTO user (id, name, email) VALUES (12345678-abcd-
abcd-abcd-abcd12345678, 'Steve', 'steve@datastax.com')
IF NOT EXISTS;
```

*Do these two INSERTs execute successfully? What is the value of the* [applied] *column? How long did these INSERT operations take to execute compared to the earlier ones?*

4. In *DevCenter*, create a new script with the query and execute it.

```
USE musicdb;

SELECT * FROM user;
```

### Update rows in tables

5. In *DevCenter*, create a new script with the following content and execute it.

```
USE musicdb;

SELECT * FROM performer WHERE name = 'The Beatles';
```

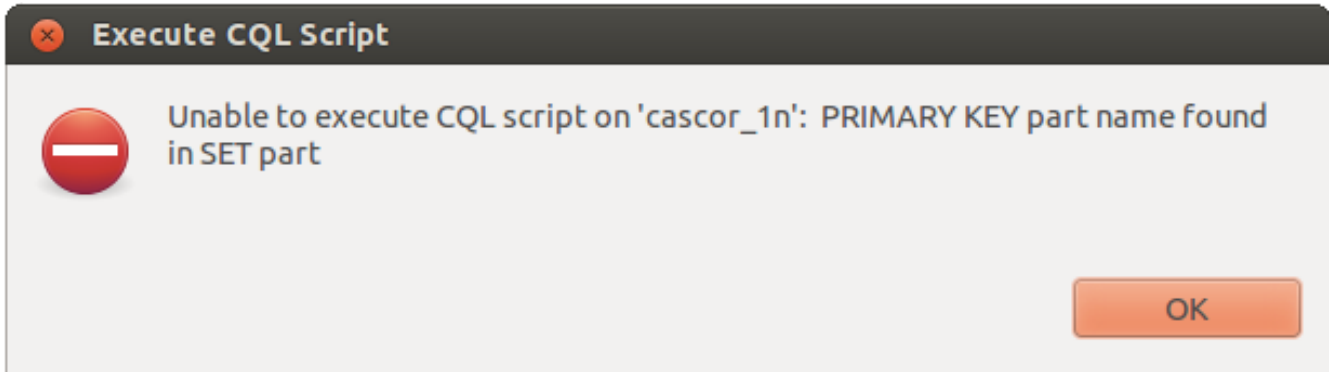
*Notice the year the band was founded.*

6. In *DevCenter*, modify the script to add the following statements and execute it.

```
USE musicdb;

UPDATE performer
SET founded = 1960
WHERE name = 'The Beatles';

SELECT * FROM performer WHERE name = 'The Beatles';
```

7. In *DevCenter*, modify the script with the following *UPDATE* statement and execute it.

```
USE musicdb;

UPDATE performer
SET name = 'Beatles'
WHERE name = 'The Beatles';

SELECT * FROM performer WHERE name = 'Beatles';
```




*Values in primary key columns cannot be updated. If such values change, an old row should be deleted and a new one should be inserted.*

8. In *DevCenter*, modify the script with the following *UPDATE* statement and execute it.

```
USE musicdb;

UPDATE performer
SET founded = 1957
WHERE name = 'The Beatles'
IF type = 'band';

SELECT * FROM performer WHERE name = 'The Beatles';
```

*The UPDATE operation is only executed if the value for the* type *column is the same. What is the value of the returned* [applied] *column for this update?*

9. In *DevCenter*, modify the script with the following *UPDATE* statement and execute it.

```
USE musicdb;

UPDATE performer
SET born = 1960
WHERE name = 'The Beatles'
IF type = 'artist';

SELECT * FROM performer WHERE name = 'The Beatles';
```

*Why did this update fail? What are the returned values for* the [applied] *and* type *column?*

END OF EXERCISE

# Exercise 9: Manipulate values in counter, collection and UDT columns

**In this exercise, you will:**

- Update a counter column
- Update a collection column
- Update a UDT column

## Steps

### Update a counter column

1. In *cqlsh*, select the *musicdb* keyspace to work with.

```
USE musicdb;
```

2. Display information about table *ratings_by_track*.

```
DESCRIBE TABLE ratings_by_track;
```

3. Update the counter columns.

```
UPDATE ratings_by_track
SET num_ratings = num_ratings + 1,
    sum_ratings = sum_ratings + 5
WHERE album_title = 'Revolver' AND
      album_year = 1966 AND
      track_title = 'Yellow Submarine';
```

4. Run the following query and observe the results.

```
SELECT * FROM ratings_by_track;
```

5. Update the counter columns again.

```
UPDATE ratings_by_track
SET num_ratings = num_ratings + 1,
    sum_ratings = sum_ratings + 4
WHERE album_title = 'Revolver' AND
      album_year = 1966 AND
      track_title = 'Yellow Submarine';
```

6. Run the following query and observe the results.

```
SELECT * FROM ratings_by_track;
```

### Update a collection column

7. In *cqlsh*, display information about table *user*.

```
DESCRIBE TABLE user;
```

8. Run the following query and observe the results.

```
SELECT * FROM user;
```

9. Update the *set* column for a specified user.

```
UPDATE user SET preferences = preferences +
                              {'Rock', 'Classic'}
WHERE id = 12345678-abcd-abcd-abcd-abcd12345678;
```

10. Run the following query and observe the results.

```
SELECT * FROM user;
```

11. Update the *set* column for a specified user.

```
UPDATE user SET preferences = preferences +
                              {'Rock', 'Jazz'}
WHERE id = 12345678-abcd-abcd-abcd-abcd12345678;
```

12. In *cqlsh*, test the following query and observe the results.

```
SELECT * FROM user;
```

*Notice how the set values are ordered. Duplicate values have been eliminated.*

## Update a UDT column

13. Display information about table *track_ratings_by_user*.

```
DESCRIBE TABLE track_ratings_by_user;
```

14. In *cqlsh*, insert a new rating for a specified user.

```
INSERT INTO track_ratings_by_user
(user, activity, rating, song)
VALUES(12345678-abcd-abcd-abcd-abcd12345678, 1234abcd-
1234-1134-1234-abcd1234abcd, 8, {album_title: 'What A
Wonderful World', album_year: 1968, track_title: 'Boogie
After Midnight'});
```

15. In *cqlsh*, test the following query and observe the results.

```
SELECT * FROM track_ratings_by_user;
```

16. In *cqlsh*, update the *track_title* to a new specified value.

```
UPDATE track_ratings_by_user SET song =
{track_title: 'Mousetrap'}
WHERE user = 12345678-abcd-abcd-abcd-abcd12345678 AND
activity = 1234abcd-1234-1134-1234-abcd1234abcd;
```

17. In *cqlsh*, test the following query and observe the results.

```
SELECT * FROM track_ratings_by_user;
```

*Although the intention was just to update the track title, what happened to the album title and the album year?*

18. In *cqlsh*, update the *song* column to a new specified value.

```
UPDATE track_ratings_by_user SET song =
{album_title: 'What A Wonderful World', album_year: 1968,
track_title: 'Mousetrap' }
WHERE user = 12345678-abcd-abcd-abcd-abcd12345678 AND
activity = 1234abcd-1234-1134-1234-abcd1234abcd;
```

19. In *cqlsh*, test the following query and observe the results.

```
SELECT * FROM track_ratings_by_user;
```

*Since the UDT column is frozen, all of the fields for the* song *column must be updated together.*

END OF EXERCISE

# Exercise 10: Explore equality and range search in queries

**In this exercise, you will:**

- Understand and execute a set of queries

## Steps

### Understand and execute a set of queries

1. In *DevCenter*, run the following query.

```
SELECT * FROM albums_by_track
WHERE track_title IN ('Yesterday', 'Tomorrow');
```

2. Run the following query.

```
SELECT * FROM tracks_by_album
WHERE album_title = '20 Greatest Hits' AND year = 1982 AND
      number > 8 AND number < 12;
```

3. Run the following query.

```
SELECT * FROM albums_by_performer
WHERE performer = 'The Beatles';
```

4. Run the following query.

```
SELECT * FROM albums_by_performer
WHERE performer = 'The Beatles' AND
      year >= 1960 AND year <= 1980;
```

5. Run the following query.

```
SELECT * FROM albums_by_performer
WHERE performer = 'The Beatles' AND
      year >= 1960 AND year <= 1980
ORDER BY year ASC;
```

6. Run the following query.

```
SELECT * FROM albums_by_performer
WHERE year >= 1960 AND year <= 1980
LIMIT 50 ALLOW FILTERING;
```

7. Run the following query.

```
SELECT * FROM performer
WHERE style = 'Rock';
```

8. Run the following query.

```
SELECT * FROM performer
WHERE style = 'Rock' AND country >= 'U'
LIMIT 1000
ALLOW FILTERING;
```

9. Run the following query.

```
SELECT * FROM user
WHERE preferences CONTAINS 'Rock';
```

END OF EXERCISE

# Demo 11: Explore queries with various predicates (optional)

**In this demo, we will:**

- Showcase equality queries
- Showcase range queries

## Steps

### Showcase equality queries

1. In *DevCenter*, design and execute different equality queries over an existing table (below is an example).

```
SELECT * FROM albums_by_genre;

SELECT * FROM albums_by_genre
WHERE genre = 'Rock';

SELECT * FROM albums_by_genre
WHERE genre IN ('Country', 'Rock');

SELECT * FROM albums_by_genre
WHERE genre = 'Rock' AND performer = 'The Beatles';

SELECT * FROM albums_by_genre
WHERE genre = 'Rock' AND
      performer = 'The Beatles' AND
      year = 1964;

SELECT * FROM albums_by_genre
WHERE genre = 'Rock' AND
      performer = 'The Beatles' AND
      year = 1964 AND
      title = 'Beatles For Sale';
```

## Showcase range queries

2. In *DevCenter*, design and execute different range queries over an existing table (below is an example).

```
SELECT * FROM albums_by_genre
WHERE genre = 'Rock' AND performer > 'T';

SELECT * FROM albums_by_genre
WHERE genre IN ('Country', 'Rock') AND
      performer > 'T' AND performer <= 'U2';

SELECT * FROM albums_by_genre
WHERE genre = 'Rock' AND
      performer = 'The Beatles' AND
      year < 1980;

SELECT * FROM albums_by_genre
WHERE genre = 'Rock' AND
      performer = 'The Beatles' AND
      year = 1964 AND
      title < 'C';
```

END OF DEMO

# Demo 12: Explore online resources on CQL and data modeling

**In this demo, we will:**

- Showcase online resources on CQL and data modeling
- Showcase educational video presentations on CQL and data modeling

## Steps

### Showcase online resources on CQL and data modeling

1. In a web browser, go to http://www.datastax.com/documentation and explore existing documentation on data modeling and CQL. Point out CQL features that are not covered in this module.

### Showcase educational video presentations on CQL and data modeling

2. In a web browser, go to http://planetcassandra.org/summit-presentations/ and explore existing summit presentations, community webinars, meetup presentations, podcasts, and other available resources.

END OF DEMO