VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Cassandra CQL Queries

Lecturer* : *Dr. Pavle Mogin

SWEN 432
Advanced Database Design and Implementation

Plan for CQL Queries

- The Syntax of the `SELECT` Statement
  - The Syntax of the `select_expression`
  - The Syntax of the `relation` in the WHERE Clause
- Simple `SELECT` expressions
- Filtering Data using `WHERE` Clause
- Restrictions on Conditions in the `WHERE` Clause
- Using Indexes
- Filtering Collections
- Querying Tables with Columns of the `counter` Type
- Keyspace Design Heuristics

  - ***Readings:*** Have a look *at Useful Links at the Course Home Page*

The Syntax of CQL `SELECT` Statement

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation AND relation ...
ORDER BY (clustering_column (ASC | DESC)...)
LIMIT n
ALLOW FILTERING
```

The Syntax of `select_expression`

`select_expression` is:

`selection_list | DISTINCT selection_list | (COUNT(*|1))`

`selection_list` is one of:

- A list of partition keys (used with `DISTINCT`)
- `selector AS alias, selector AS alias, ...|*`
  - `alias` is an alias for a column name
- `A selector` is:

  `column_name|( WRITETIME (column_name) )|`
  `( TTL (column_name) ) | (function (selector, selector, ...) )`
  - function is a timeuuid function, a token function, or a blob conversion function

The Syntax of `relation`

- A `relation` is:

  `column_name op term |`

  `column_name IN ( term, ( term ... ) ) |`

  `TOKEN (column_name, ...) op ( term )`
- An `op` is:

  `= | < | > | <= | > | = | CONTAINS | CONTAINS KEY`
- A term is:
  - A constant: string, number, uuid, boolean, hex
  - A bind marker (`?`)
  - A function
  - A collection : `set: {literal, ...}, list: [literal, ... ], map: {literal: literal, ...}`

Using CQL `SELECT` Statement

- The CQL SELECT statement works very similar to the SQL SELECT statement with a few exceptions
  - The main exception is that it retrieves one or more records from a single table only
  - The others will be discussed through examples
- Simple examples

```
use blogs;
SELECT * FROM users;
SELECT COUNT(*) FROM blog_entries;
SELECT DISTINCT user_name, body
FROM blog_entries;
```

  - The last `SELECT` returns only one row for each distinct `user_name` that is the partition key

More Simple `SELECT` Examples

```
ALTER TABLE users ADD created_at timeuuid;
```

```
INSERT INTO users (user_name, name,
created_at)  VALUES ('jbond', 'James',
now());
// now() returns a unique timeuuid in ms
```

```
SELECT user_name, name, dateOf(created_at)
AS creation_date FROM users;
//dateOf() returns the timestamp part of an
//timeuuid
```

Filtering Data Using `WHERE`

- In the `WHERE` clause, columns should be referred using the actual names, not aliases
- Each column in the `WHERE` clause has:
  - Either to belong to the partition key or
  - To be indexed using `CREATE INDEX`
- A `WHERE` clause relation has to be build by putting:
  - The name of the column to the left of an operator and
  - The column value to the right of the operator

```
SELECT * FROM users
WHERE user_name = 'jbond';
```

  - Note, `user_name` is the partitioning column

Restrictions on Conditions (1)

- Cassandra does not support non-equal conditional operations on the partition key
  - The token function should be used for range queries on the partition key

```
SELECT * FROM users WHERE
TOKEN(user_name) >= TOKEN('asmith') AND
TOKEN(user_name) =< TOKEN('jbond');
```

Restriction on The Use of Conditions (2)

- The `IN` condition is allowed on the last column of the partition key only if the query contains equality conditions on all preceding columns of the key
- Assume, the `blog_entries` table has been defined in the following way:

```
CREATE TABLE blog_entries (user_id text,
date int, body text, no int,
PRIMARY KEY ((user_name, date), no));
```

```
SELECT * FROM blog_entries WHERE
user_name = 'jbond' AND date IN
(20150810, 20150811, 20150812);
```

The `ALLOW FILTERING` Clause

- When one attempts a potentially expensive query, such as searching a range of rows, Cassandra prompts the following 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.
```

- In such cases, imposing a limit using the `LIMIT n` clause is also recommended to reduce memory used

Restriction on The Use of Conditions (3)

- Cassandra supports greater-than and less-than comparisons on a clustering column, but for a given partition key, the conditions on the clustering column are restricted to the filters that allow Cassandra to select a contiguous ordering of rows
- Assume `blog_entries` primary key is `(user_name, date)`

```
SELECT * FROM blog_entries WHERE
user_name = 'jbond' AND
date >= 20150810 AND date < 20160410
ALLOW FILTERING;
```

Secondary Indexes

- Building and using a secondary index gives best results if the filtering column has many duplicate values
- Cassandra would refuse to execute a query asking for users from a certain city unless `ALLOW FILTERING` has been defined

```
CREATE INDEX city_index on users(city);
```

```
SELECT user_name, name FROM users
WHERE city = 'London' LIMIT 10;
```

- If a secondary index on the filtering field has been defined, the `ALLOW FILTERING` clause is not needed
- If a secondary index on the filtering field has not been defined, the `ALLOW FILTERING` clause is needed

Filtering Collections

- Cassandra retrieves the collection in its entirety
  - Assume `subscribers` column of the `subscribed_to` table is of the type set

```
SELECT user_name, subscribers
FROM subscribed_to;
```

- A table containing a collection column can be indexed on the collection column
- The CONTAINS condition in the WHERE clause can be used to filter the data for a particular value in the collection

```
CREATE INDEX ON subscribed_to
(subscribers);
```

```
SELECT user_name FROM subscribed_to
WHERE subscribers CONTAINS 'canslow';
```

Filtering a `map` Collection Column

- Assume now, `subscribers` column of the `subscribed_to` table is of the type map
  - This is an index on map values:

```
CREATE INDEX my_map_values_index ON
subscribed_to (subscribers);
```

```
SELECT user_name FROM subscribed_to WHERE
subscribers CONTAINS 'London';
```

- Indexes on keys and values of a map can't co-exist

```
DROP INDEX my_map_values_index;
```

```
CREATE INDEX my_map_keys_index ON
subscribed_to (KEYS(subscribers));
```

```
SELECT user_name FROM subscribed_to
WHERE subscribers CONTAINS KEY 'canslow';
```

Tables With `counter` Data Type

- Extending `blogs` key space by a table for counting daily and total number of visits to users' blogs

```
CREATE TABLE visits (user_name text, date
int, total_count counter static,
daily_count counter, PRIMARY KEY(
user_name, date);
```

```
UPDATE visits SET dayly_count =
daily_count + 1, total_count = total_count
+ 1 WHERE user_name = 'jbond' AND date =
20160317;
```

```
SELECT DISTINCT total_count FROM visits
WHERE user_name = 'jbond';
```

`COUNT(*)` versus `counter` Tables

- Assume the `blog_entries` table has the primary key `((user_name, date), no)`
- To find the total number of blogs of the user `jbond`, the following query has to search through several partitions

```
SELECT COUNT(*) FROM blog_entries WHERE
user_name = 'jbond';
```

- A query to a table having a counter column containing the number of blogs for each user, would require accessing just one partition

Queries to `blog` Keyspace

- Each table in a Cassandra keyspace is aimed for a few specific queries
- In the `blog keyspace`:
  - The table `users` is aimed for answering queries about bloggers data,
  - The table `blog_entries` is aimed for retrieving blogs of a user,
  - The table `subscribes_to` is aimed for answering the question: who are subscribers to a user's blog
  - The table `visits` is aimed for retrieving counts of blogs made by a user
- Each table represents a materialized view for answering at least one query
  - Loosely speaking the answers for all anticipated queries are already physically stored

Database Design Principle Guidelines

- Each table of a keyspace has a self-contained set of columns intended to support one (or possibly more) specific queries
  - Since joins are not supported, for each query there has to exist at least one table containing all data needed to satisfy the query
  - Additionally, data representing the query condition need to belong either to partition key or to be indexed, otherwise `ALLOW FILTERING` clause has to be used
- The statements above are the main guidelines for designing tables implying that the starting point in designing tables represents the identification of user queries

Formalized Database Design Guidelines

- Let *q* be a query having:
  - The set of column names *C*(*s*) in its `SELECT` clause and
  - The set of column names *C*(*w*) in its `WHERE` clause
- Let *t* be a table having:
  - The set of column names *C*(*t*),
  - The set of partition key column names *C*(*k*), and
  - The set of indexed column names *C*(*i*)
- If

$$(C(t) \supseteq (C(s) \setminus \{\text{COUNT}(*)\})) \wedge ((C(k) \cup C(i)) \supseteq C(w))$$

  then the query *q* can be efficiently satisfied by table *t*
- If

$$(\exists a \in C(t) \cap C(w)) (a \notin (C(k) \cup C(i)))$$

  the clause `ALLOW FILTERING` has to be used

Other Design Considerations

- The primary key values have to be unique
- The primary key can be a superkey
  - The primary key can contain logically redundant columns
    - To avoid creating secondary indexes
    - To use the same table for answering more than one query efficiently
- Consider queries q_1 and q_2, having

$$(C(w_1) \subseteq C(w_2))$$

- Both queries can be efficiently satisfied by a table t having

$$((C(k) \cup C(i)) \supseteq C(w_2)) \wedge (C(t) \supseteq (C(s_1) \cup C(s_2)))$$

Summary

- Syntax of the CQL `SELECT` statement:

```
SELECT select_expression
FROM keyspace_name.table_name
WHERE relation AND relation ...
ORDER BY (clustering_column (ASC | DESC)...)
LIMIT n
ALLOW FILTERING
```

- The `relation` clause should contain:
  - Either primary key and indexed columns, or
  - The `ALLOW FILTERING` clause has to be used
- Since joins are not supported, tables are designed to contain all data needed to answer a set of queries