Secondary indexes (2i) overview

Secondary indexes (2i) overview

Secondary indexes (2i) allow you create one or multiple secondary indexes on the same database table, with each 2i based on any column.

Exception: There is no need to define an 2i based on a single-column partition key.

Concepts

How 2i works and when to use this type of indexing.

Working with 2i

Create, check, alter, drop, and query secondary indexes (2i).

2i operations

Rebuilding and maintaining 2i indexes.

Reference: CREATE INDEX, **DROP INDEX**

When to use an index

When to use an index

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 on average is required 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.

If secondary indexes are required, based on one or more table columns other than its partition key, use Storage-Attached Indexing (SAI). For details, see **CREATE CUSTOM INDEX**.

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.
- Do not add a secondary index and a search index to the same table.

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 incurs 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 recording 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 built-in index. For columns containing unique data, it is sometimes better for performance 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

The database 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. When looking for a row in a large partition, narrow the search to avoid query performance degradation.

Secondary index (2i) concepts

Secondary index (2i) concepts

Secondary indexing (2i) is a globally-distributed index for **Cassandra** databases.

2i is the original secondary indexing of **Apache Cassandra**, but today, it suffers from poor performance and latency. If you are building new indexing, use **SAI indexing** instead.

2i adds column-level indexes to any CQL table column of any CQL data type, except for a counter column. However, the indexes are locally built on each **Apache Cassandra** node in a cluster, so using 2i for queries results in poor performance. A number of techniques exist for guarding against the undesirable scenario where data might be incorrectly retrieved during a query based on stale values in an index.

There are distinct conditions about **when and when not to use a 2i index**.

Secondary Indexes

Secondary Indexes

CQL supports creating secondary indexes on tables, allowing queries on the table to use those indexes. A secondary index is identified by a name defined by:

```
index_name::= re('[a-zA-Z_0-9]+')
```

CREATE INDEX

The `CREATE INDEX` statement is used to create a new secondary index for a given (existing) column in a given table. A name for the index itself can be specified before the `ON` keyword, if desired.

```
create_index_statement::= CREATE [ CUSTOM ] INDEX [ IF NOT EXISTS ] [ index_name ]
    ON table_name '(' index_identifier ')'
    [ USING index_type [ WITH OPTIONS = map_literal ] ]
index_identifier::= column_name
    | ( KEYS | VALUES | ENTRIES | FULL ) '(' column_name ')'
index_type::= 'sai' | 'legacy_local_table' | fully_qualified_class_name
```

If data already exists for the column, it will be indexed asynchronously. After the index is created, new data for the column is indexed automatically at insertion time. Attempting to create an already existing index will return an error unless the `IF NOT EXISTS` option is used. If it is used, the statement will be a no-op if the index already exists.

Examples:

```
CREATE INDEX userIndex ON NerdMovies (user);
CREATE INDEX ON Mutants (abilityId);
CREATE INDEX ON users (KEYS(favs));
CREATE INDEX ON users (age) USING 'sai';
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/'};
```

Index Types

The `USING` keyword optionally specifies an index type. There are two built-in types:

- legacy_local_table - (default) legacy secondary index, implemented as a hidden local table
- sai - "storage-attched" index, implemented via optimized SSTable/Memtable-attached indexes

To create a custom index, a fully qualified class name must be specified.

Indexes on Map Keys

When creating an index on a `maps <maps>`, you may index either the keys or the values. If the column identifier is placed within the `keys()` function, the index will be on the map keys, allowing you to use `CONTAINS KEY` in `WHERE` clauses. Otherwise, the index will be on the map values.

DROP INDEX

Dropping a secondary index uses the `DROP INDEX` statement:

```
drop_index_statement::= DROP INDEX [ IF EXISTS ] index_name
```

The `DROP INDEX` statement is used to drop an existing secondary index. The argument of the statement is the index name, which may optionally specify the keyspace of the index.

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.

Working with secondary indexing (2i)

Working with secondary indexing (2i)

Prerequisites

- **Keyspace created**
- **Table created**

Create a secondary index (2i)

Create indexes on one or more columns after defining a table. Secondary indexes created with 2i can be used to query a table using a column other than the table's partition key.

In a production environment, certain columns might not be good choices, depending on their **cardinality**.

IMPORTANT	Do not add an storage-attached index (SAI) to the same table. See the difference between these index types in the **overview**.

Create simple 2i

Create simple 2i indexes on a table to see how indexing works. Start by creating a table, `cycling.alt_stats`, that yields statistics about cyclists:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_alt_stats (
  id UUID PRIMARY KEY,
  lastname text,
  birthday date,
  nationality text,
  weight float,
  w_units text,
  height float,
  first_race date,
  last_race date
);
```

Now create indexes on the columns `birthday` and `nationality`:

```
CREATE INDEX IF NOT EXISTS birthday_idx
ON cycling.cyclist_alt_stats (birthday);
CREATE INDEX IF NOT EXISTS nationality_idx
```

```
ON cycling.cyclist_alt_stats (nationality);
```

The following query attempts to retrieve the cyclists with a specified `birthday` and `nationality`. The query returns an error:

CQL

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

Result

```
InvalidRequest: Error from server: code=2200 [Invalid query]
message="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"
```

The indexes have been created on appropriate low-cardinality columns, but the previous 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, the database requires the `ALLOW FILTERING` directive. The error is not due to multiple indexes, but the lack of a partition key definition in the query.

CQL

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

Result

```
id                                   | birthday   | first_race | height |
last_race  | lastname | nationality | w_units | weight
--------------------------------------+------------+------------+--------+
------------+----------+-------------+---------+--------
 e0953617-07eb-4c82-8f91-3b2757981625 | 1982-01-29 | 1998-02-15 |   1.78 | 2017-
```

```
04-16 |    BRUTT |      Russia |      kg |     68

(1 rows)
```

Thus, one of the difficulties of using 2is is illustrated. **SAI** is almost always a better option.

Create a 2i on a collection column

Collections can be indexed and queried to find a collection containing a particular value. Sets and lists are indexed a bit 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.

- 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 IF NOT EXISTS teams_idx
ON cycling.cyclist_career_teams (teams);
```

```
SELECT *
FROM cycling.cyclist_career_teams
WHERE teams CONTAINS 'Rabobank-Liv Giant';
```

```
id                                   | lastname | teams
--------------------------------------+----------+
------------------------------------------------------------------------------
-----------------------
 1c9ebc13-1eab-4ad5-be87-dce433216d40 |    BRAND | {'AA Drink - Leontien.nl',
'Leontien.nl', 'Rabobank-Liv Giant', 'Rabobank-Li
v Woman Cycling Team'}

(1 rows)
```

- 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 IF NOT EXISTS team_year_keys_idx
ON cycling.cyclist_teams ( KEYS (teams) );
```

```
SELECT *
FROM cycling.cyclist_teams
WHERE teams CONTAINS KEY 2015;
```

```
id                                   | firstname | lastname   | teams
--------------------------------------+-----------+------------+
-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------
 cb07baad-eac8-4f65-b28a-bddc06a0de23 | Elizabeth | ARMITSTEAD | {2011: 'Team Garmin -
Cervelo', 2012: 'AA Drink - Leontien.nl', 2013: 'Boels:Dolmans Cycling Team', 2014:
'Boels:Dolmans Cycling Team', 2015: 'Boels:Dolmans Cycling Team'}
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |        VOS |
{2015: 'Rabobank-Liv Woman Cycling Team'}

(2 rows)
```

- 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 IF NOT EXISTS cycling.birthday_list (
  cyclist_name text PRIMARY KEY,
  blist map<text, text>
);
```

```
CREATE INDEX IF NOT EXISTS 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'}
```

```
(2 rows)
```

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

```
SELECT *
FROM cycling.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'}

(2 rows)
```

- Create an index on the map values and find cyclists who have a particular value found in the specified map.

```
CREATE TABLE IF NOT EXISTS cycling.birthday_list (
  cyclist_name text PRIMARY KEY,
  blist map<text, text>
);
```

```
CREATE INDEX IF NOT EXISTS blist_values_idx
ON cycling.birthday_list ( VALUES(blist) );
```

```
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'}

(2 rows)
```

- 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 IF NOT EXISTS cycling.race_starts (
  cyclist_name text PRIMARY KEY,
  rnumbers FROZEN<LIST<int>>
);
```

```
CREATE INDEX IF NOT EXISTS 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]

(1 rows)
```

Check secondary index (2i) existence

Verify that an index exists:

CQL

```
DESCRIBE TABLE cycling.birthday_list;
```

Result

```
TBD
```

Alter a secondary index (2i)

Secondary indexes cannot be altered. If you wish to change a 2i, you need to **drop the index** and **create a new one**.

Drop a secondary index (2i)

Drop a 2i:

```
DROP INDEX IF EXISTS cycling.teams_idx;
```

Querying using secondary indexes (2i)

Rebuilding and maintaining secondary indexes (2i)

Rebuilding and maintaining secondary indexes (2i)

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.