Apache Cassandra : Indexes

Indexing concepts

Indexing concepts

The data stored in CQL tables can be queried by a variety of methods. The main method uses the partition key defined for a table, and is called primary indexing. Often, however, a query must use another column of a table to select the rows desired, and secondary indexing is required. Secondary indexing use fast, efficient lookup of data that matches a given condition. After any index is created, data can be queried using that index.

Apache Cassandra has the following types of indexing available:

Indexing type	Versions
Primary indexing	All
Storage-attached indexing (SAI)	5.0
Secondary indexing (2i)	All

Primary indexing

The primary index is the partition key in **Apache Cassandra**. The storage engine of **Apache Cassandra** uses the partition key to store rows of data, and the most efficient and fast lookup of data matches the partition key.

Storage-attached indexing (SAI)

SAI uses indexes for non-partition columns, and attaches the indexing information to the SSTables that store the rows of data. The indexes are located on the same node as the SSTable, and are updated when the SSTable is updated. SAI is the most appropriate indexing method for most use cases.

Secondary indexing (2i)

Secondary indexing is the original built-in indexing written for **Apache Cassandra**. These indexes are all local indexes, stored in a hidden table on each node of a **Apache Cassandra** cluster, separate from the table that contains the values being indexed. The index must be read from the node. This indexing method is only recommended when used in conjunction with a partition key.

CREATE INDEX

CREATE INDEX

Define a new index on a single column of a table. If the column already contains data, it is indexed during the execution of this statement. After an index has been created, it is automatically updated when data in the column changes. **Apache Cassandra** supports creating an index on most columns, including the partition and cluster columns of a PRIMARY KEY, collections, and static columns. Indexing via this `CREATE INDEX` command can impact performance. Before creating an index, be aware of when and **when not to create an index**.

Use **CREATE CUSTOM INDEX** for Storage-Attached Indexes (SAI).

Restriction: Indexing counter columns is not supported. For maps, index the key, value, or entries.

Synopsis

```
CREATE INDEX [ IF NOT EXISTS ] <index_name>
  ON [<keyspace_name>.]<table_name>
  ([ ( KEYS | FULL ) ] <column_name>)
  (ENTRIES <column_name>) ;
```

▼ *Syntax legend*

include:cassandra:partial$cql-syntax-legend.adoc[]

index_name

Optional identifier for index. If no name is specified, DataStax Enterprise names the index: `<table_name>_<column_name>_idx`. Enclose in quotes to use special characters or preserve capitalization.

Examples

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 IF NOT EXISTS cycling.rank_by_year_and_name (
  race_year int,
  race_name text,
  cyclist_name text,
```

```
  rank int,
  PRIMARY KEY ((race_year, race_name), rank)
);
```

```
CREATE INDEX IF NOT EXISTS rank_idx
ON cycling.rank_by_year_and_name (rank);
```

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 teams to the `cyclist_career_teams` table to index the data in the teams set.

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (
  id UUID PRIMARY KEY,
  lastname text,
  teams set<text>
);
```

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

Creating an index on map keys

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. Assume a cyclist table contains this map data:

```
{'nation':'CANADA' }
```

The map key is located to the left of the colon, and the map value is located to the right of the colon.

To index map keys, use the `KEYS` keyword and map name in nested parentheses:

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

To query the table, you can use **CONTAINS KEY** in `WHERE` clauses.

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

The example returns cyclist teams that have an entry for the year 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)
```

Creating an index on map entries

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, use the `ENTRIES` keyword and map name in nested parentheses:

```
CREATE INDEX IF NOT EXISTS blist_idx
ON cycling.birthday_list ( ENTRIES(blist) );
----
```

To query the map entries in the table, use a `WHERE` clause with the map name and a value.

```
SELECT *
FROM cycling.birthday_list
WHERE blist[ 'age' ] = '23';
```

The example finds cyclists who are the same age.

```
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)
```

Use the same index to 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)
```

Creating an index on map values

To create an index on map values, use the `VALUES` keyword and map name in nested parentheses:

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

To query the table, use a `WHERE` clause with the map name and the value it contains.

```
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)
```

Creating an index on the full content of a frozen collection

You can create an index on a full `FROZEN` collection. A `FULL` index can be created on a set, list, or map column of a table that doesn't have an existing index.

Create an index on the full content of a `FROZEN list`. The table in this example stores the number of Pro wins, Grand Tour races, and Classic races that a cyclist has competed in.

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

To index collection entries, use the `FULL` keyword and collection name in nested parentheses. For example, index the frozen list `rnumbers`.

```
CREATE INDEX IF NOT EXISTS rnumbers_idx
ON cycling.race_starts ( FULL(rnumbers) );
```

To query the table, use a `WHERE` clause with the collection name and values:

```
SELECT *
FROM cycling.race_starts
WHERE rnumbers = [39, 7, 14];
```

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

(1 rows)
```

CREATE CUSTOM INDEX

CREATE CUSTOM INDEX

Cassandra 5.0 is the only supported database currently.

Creates a Storage-Attached Indexing (SAI) index. You can create multiple secondary indexes on the same database table, with each SAI index based on any column in the table. All column date types except the following are supported for SAI indexes:

- `counter`
- geospatial types: `PointType`, `LineStringType`, `PolygonType`
- non-frozen user-defined type (UDT)

One exception

You cannot define an SAI index based on the partition key when it's comprised of only one column. If you attempt to create an SAI index in this case, SAI issues an error message.

However,you can define an SAI index on one of the columns in a table's composite partition key, i.e., a partition key comprised of multiple columns. If you need to query based on one of those columns, an SAI index is a helpful option. In fact, you can define an SAI index on each column in a composite partition key, if needed.

Defining one or more SAI indexes based on any column in a database table (with the rules noted above) subsequently gives you the ability to run performant queries that use the indexed column to filter results.

See the **SAI section**.

Synopsis

```
CREATE [CUSTOM] INDEX [ IF NOT EXISTS ] [ <index_name> ]
  ON [ <keyspace_name>.]<table_name> (<column_name>)
    | [ (KEYS(<map_name>)) ]
    | [ (VALUES(<map_name>)) ]
    | [ (ENTRIES(<map_name>)) ]
  USING 'sai'
  [ WITH OPTIONS = { <option_map> } ] ;
```

▼ *Syntax legend*

Table 1. Legend

Syntax conventions	Description
UPPERCASE	Literal keyword.
Lowercase	Not literal.
`< >`	Variable value. Replace with a user-defined value.
`[]`	Optional. Square brackets (`[]`) surround optional command arguments. Do not type the square brackets.
`( )`	Group. Parentheses (`( )`) identify a group to choose from. Do not type the parentheses.
`|`	Or. A vertical bar (`|`) separates alternative elements. Type any one of the elements. Do not type the vertical bar.
`...`	Repeatable. An ellipsis (`...`) indicates that you can repeat the syntax element as often as required.
`'<Literal string>'`	Single quotation (`'`) marks must surround literal strings in CQL statements. Use single quotation marks to preserve upper case.
`{ <key> : <value> }`	Map collection. Braces (`{ }`) enclose map collections or key value pairs. A colon separates the key and the value.
`<datatype2`	Set, list, map, or tuple. Angle brackets (`< >`) enclose data types in a set, list, map, or tuple. Separate the data types with a comma.
`<cql_statement>;`	End CQL statement. A semicolon (`;`) terminates all CQL statements.
`[--]`	Separate the command line options from the command arguments with two hyphens (`--`). This syntax is useful when arguments might be mistaken for command line options.
`' <<schema\> ... </schema\>> '`	Search CQL only: Single quotation marks (`'`) surround an entire XML schema declaration.
`@<xml_entity>='<xml_entity_type>'`	Search CQL only: Identify the entity and literal value to overwrite the XML element in the schema and solrConfig files.

index_name

Optional identifier for index. If no name is specified, the default is used,

`<table_name>_<column_name>_idx`. Enclose in quotes to use special characters or to preserve capitalization.

column_name

The name of the table column on which the SAI index is being defined. SAI allows only alphanumeric characters and underscores in names. SAI returns `InvalidRequestException` if you try to define an index on a column name that contains other characters, and does not create the index.

map_name

Used with **collections**, identifier of the `map_name` specified in `CREATE TABLE ... map(<map_name>)`. The regular column syntax applies for collection types `list` and `set`.

option_map

Define options in JSON simple format.

Option	Description
`case_sensitive`	Ignore case in matching string values. Default: `true`.
`normalize`	When set to `true`, perform Unicode normalization on indexed strings. SAI supports Normalization Form C (NFC) Unicode. When set to `true`, SAI normalizes the different versions of a given Unicode character to a single version, retaining all the marks and symbols in the index. For example, SAI would change the character Å (U+212B) to Å (U+00C5). When implementations keep strings in a normalized form, equivalent strings have a unique binary representation. See Unicode Standard Annex #15, Unicode Normalization Forms. Default: `false`.
`ascii`	When set to `true`, SAI converts alphabetic, numeric, and symbolic characters that are not in the Basic Latin Unicode block (the first 127 ASCII characters) to the ASCII equivalent, if one exists. For example, this option changes à to a. Default: `false`.

Query operators

SAI supports the following query operators for tables with SAI indexes:

- Numerics: `=`, `<`, `>`, `⇐`, `>=`, `AND`, `OR`, `IN`
- Strings: `=`, `CONTAINS`, `CONTAINS KEY`, `AND`, `OR`, `IN`

SAI does not supports the following query operators for tables with SAI indexes:

- Strings or Numerics: `LIKE`

Examples

These examples define SAI indexes for the `cycling.cyclist_semi_pro` table, which is demonstrated in the **SAI quickstart**.

```
CREATE INDEX lastname_sai_idx ON cycling.cyclist_semi_pro (lastname)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'false', 'normalize': 'true', 'ascii': 'true'};

CREATE INDEX age_sai_idx ON cycling.cyclist_semi_pro (age)
USING 'sai';

CREATE INDEX country_sai_idx ON cycling.cyclist_semi_pro (country)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'false', 'normalize': 'true', 'ascii': 'true'};

CREATE INDEX registration_sai_idx ON cycling.cyclist_semi_pro (registration)
USING 'sai';
```

For sample queries that find data in `cycling.cyclist_semi_pro` via these sample SAI indexes, see **Submit CQL queries**. Also refer **Examine SAI column index and query rules**.

SAI collection map examples with keys, values, and entries

The following examples demonstrate using collection maps of multiple types (`keys`, `values`, `entries`) in SAI indexes. For related information, see **Creating collections** and **Using map type**.

Also refer to the SAI collection examples of type **list and set** in this topic.

First, create the keyspace:

```
CREATE KEYSPACE demo3 WITH REPLICATION =
       {'class': 'SimpleStrategy', 'replication_factor': '1'};
```

Next, use the keyspace:

```
USE demo3;
```

Create an `audit` table, with a collection map named `text_map`:

```
CREATE TABLE audit ( id int PRIMARY KEY , text_map map<text, text>);
```

Create multiple SAI indexes on the same `map` column, each using `KEYS`, `VALUES`, and `ENTRIES`.

NOTE	Creating multiple SAI indexes with different map types **on the same column** requires Cassandra 5.0 or later.

```
CREATE INDEX ON audit (KEYS(text_map)) USING 'sai';
CREATE INDEX ON audit (VALUES(text_map)) USING 'sai';
CREATE INDEX ON audit (ENTRIES(text_map)) USING 'sai';
```

Insert some data:

```
INSERT INTO audit (id, text_map) values (1, {'Carlos':'Perotti', 'Marcel':'Silva'});
INSERT INTO audit (id, text_map) values (2, {'Giovani':'Pasi', 'Frances':'Giardello'});
INSERT INTO audit (id, text_map) values (3, {'Mark':'Pastore', 'Irene':'Cantona'});
```

Query all data:

Query

```
SELECT * FROM audit;
```

Result

```
 id | text_map
----+-------------------------------------------
  1 | {'Carlos': 'Perotti', 'Marcel': 'Silva'}
  2 | {'Frances': 'Giardello', 'Giovani': 'Pasi'}
  3 | {'Irene': 'Cantona', 'Mark': 'Pastore'}

(3 rows)
```

Query using the SAI index for specific entries in the `map` column:

Query

```
SELECT * FROM audit WHERE text_map['Irene'] = 'Cantona' AND text_map['Mark'] =
'Pastore';
```

Result

```
 id | text_map
----+-----------------------------------------
  3 | {'Irene': 'Cantona', 'Mark': 'Pastore'}

(1 rows)
```

Query using the SAI index for specific keys in the `map` column using `CONTAINS KEY`:

Query

```
SELECT * FROM audit WHERE text_map CONTAINS KEY 'Giovani';
```

Result

```
 id | text_map
----+-----------------------------------------------
  2 | {'Frances': 'Giardello', 'Giovani': 'Pasi'}

(1 rows)
```

Query using the SAI index for specific values in the `map` column with `CONTAINS`:

Query

```
SELECT * FROM audit WHERE text_map CONTAINS 'Silva';
```

Result

```
 id | text_map
----+-----------------------------------------
  1 | {'Carlos': 'Perotti', 'Marcel': 'Silva'}

(1 rows)
```

Remember that in CQL queries using SAI indexes, the `CONTAINS` clauses are supported with, and specific to:

- SAI **collection maps** with `keys`, `values`, and `entries`

- SAI **collections** with `list` and `set` types

SAI collection examples with list and set types

These examples demonstrate using collections with the `list` and `set` types in SAI indexes. For related information, see:

- **Creating collections**
- **Using list type**
- **Using set type**

```
CREATE KEYSPACE IF NOT EXISTS demo3 WITH REPLICATION =
       {'class': 'SimpleStrategy', 'replication_factor': '1'};
```

```
USE demo3;
```

Using the list type

Create a `calendar` table with a collection of type `list`.

```
CREATE TABLE calendar (key int PRIMARY KEY, years list<int>);
```

Create an SAI index using the collection's `years` column.

```
CREATE INDEX ON calendar(years) USING 'sai';
```

Insert some random `int` list data for `years`, just for demo purposes.

TIP

Notice the `INSERT` command's square brackets syntax for list values.

```
INSERT INTO calendar (key, years) VALUES (0, [1990,1996]);
INSERT INTO calendar (key, years) VALUES (1, [2000,2010]);
INSERT INTO calendar (key, years) VALUES (2, [2001,1990]);
```

Query with `CONTAINS` example:

Query

```
SELECT * FROM calendar WHERE years CONTAINS 1990;
```

Result

```
 key | years
-----+--------------
   0 | [1990, 1996]
   2 | [2001, 1990]
(2 rows)
```

This example created the `calendar` table with `years list<int>`. Of course, you could have created the table with `years list<text>`, for example, inserted 'string' values, and queried on the strings.

Using the set type

Now create a `calendar2` table with a collection of type `set`.

```
CREATE TABLE calendar2 (key int PRIMARY KEY, years set<int>);
```

Create an SAI index using the collection's `years` column — this time for the `calendar2` table.

```
CREATE INDEX ON calendar2(years) USING 'sai';
```

Insert some random `int` set data for `years`, again just for demo purposes.

TIP

Notice the `INSERT` command's curly braces syntax for set values.

```
INSERT INTO calendar2 (key, years) VALUES (0, {1990,1996});
INSERT INTO calendar2 (key, years) VALUES (1, {2000,2010});
INSERT INTO calendar2 (key, years) VALUES (2, {2001,1990,2020});
```

Query with `CONTAINS` example from the list:

Query

```
SELECT * FROM calendar2  WHERE years CONTAINS 1990;
```

Result

```
 key | years
-----+--------------------
   0 |       {1990, 1996}
   2 | {1990, 2001, 2020}

(2 rows)
```

DROP INDEX

DROP INDEX

Removes an existing index. The default index name is `table_name_column_name_idx`.

Synopsis

```
DROP INDEX [ IF EXISTS ] [<keyspace_name>.]<index_name> ;
```

▼ *Syntax legend*

include:cassandra:partial$cql-syntax-legend.adoc[]

Example

Drop the index `rank_idx` from the `cycling.rank_by_year_and_name` table.

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