Storage-Attached Indexing (SAI) Overview

Storage-Attached Indexing (SAI) Overview

Storage-Attached Indexing (SAI) lets you create one or multiple secondary indexes on the same database table, with each SAI index based on any single column.

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

SAI concepts

Use Storage-Attached Indexing (SAI) to create multiple secondary indexes on the same table.

SAI quickstart

Follow the steps to get started quickly with SAI.

SAI FAQs

Frequently asked questions about SAI.

Working with SAI

Create, check, alter, drop, and query SAI.

SAI operations

Configuring and monitoring SAI indexes.

Querying with SAI

Understand the columns on which you can define SAI indexes and run queries.

Reference: CREATE CUSTOM INDEX, **DROP INDEX**

Storage-attached indexing (SAI) concepts

Storage-attached indexing (SAI) concepts

Storage-Attached Indexing (SAI) is a highly-scalable, globally-distributed index for **Cassandra** databases.

The main advantage of SAI over existing indexes for **Apache Cassandra** are:

- enables vector search for AI applications
- shares common index data across multiple indexes on same table
- alleviates write-time scalability issues
- significantly reduced disk usage
- great numeric range performance
- zero copy streaming of indexes

In fact, SAI provides the most indexing functionality available for **Apache Cassandra**. SAI adds column-level indexes to any CQL table column of almost any CQL data type.

SAI enables queries that filter based on:

- vector embeddings
- AND/OR logic for numeric and text types
- IN logic (use an array of values) for numeric and text types
- numeric range
- non-variable length numeric types
- text type equality
- CONTAINs logic (for collections)
- tokenized data
- row-aware query path
- case sensitivity (optional)
- unicode normalization (optional)

Advantages

Defining one or more SAI indexes based on any column in a database table subsequently gives you the ability to run performant queries that specify the indexed column. Especially compared to relational

databases and complex indexing schemes, SAI makes you more efficient by accelerating your path to developing apps.

SAI is deeply integrated with the storage engine of Cassandra. The SAI functionality indexes the in-memory memtables and the on-disk SSTables as they are written, and resolves the differences between those indexes at read time. Consequently, the design of SAI has very little operational complexity on top of the core database. From snapshot creation, to schema management, to data expiration, SAI integrates tightly with the capabilities and mechanisms that the core database already provides.

SAI is also fully compatible with zero-copy streaming (ZCS). Thus, when you bootstrap or decommission nodes in a cluster, the indexes are fully streamed with the SSTables and not serialized or rebuilt on the receiving node's end.

At its core, SAI is a filtering engine, and simplifies data modeling and client applications that would otherwise rely heavily on maintaining multiple query-specific tables.

Performance

SAI outperforms any other indexing method available for **Apache Cassandra**.

SAI provides more functionality than secondary indexing (2i), using a fraction of the disk space, and reducing the total cost of ownership (TCO) for disk, infrastructure, and operations. For read path performance, SAI performs at least as well as the other indexing methods for throughput and latency.

SAI write path and read path

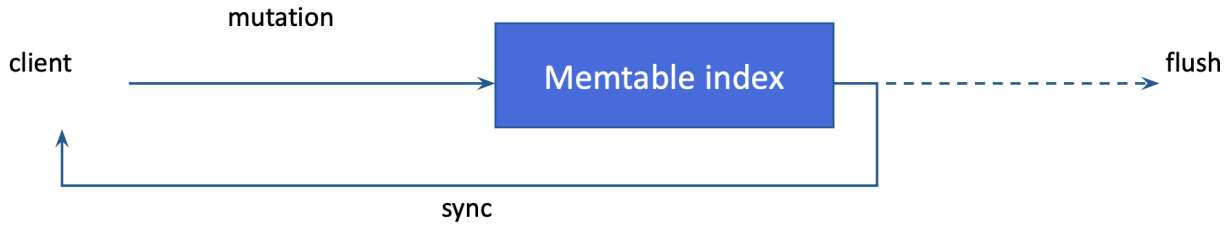
SAI is deeply integrated with the storage engine of the underlying database. SAI does not abstractly index tables. Instead, SAI indexes **Memtables** and Sorted String Tables **(SSTables)** as they are written, resolving the differences between those indexes at read time. Each Memtable is an in-memory data structure that is specific to a particular database table. A Memtable resembles a write-back cache. Each SSTable is an immutable data file to which the database writes Memtables periodically. SSTables are stored on disk sequentially and maintained for each database table.

This topic discusses the details of the SAI read and write paths, examining the SAI indexing lifecycle.

SAI write path

An SAI index can be created either before any data is written to a CQL table, or after data has been written. As a refresher, data written to a CQL table will first be written to a Memtable, and then to an SSTable once the data is flushed from the Memtable. After an SAI index is created, SAI is notified of all mutations against the current Memtable. Like any other data, SAI updates the indexes for inserts and updates, which **Apache Cassandra** treats in exactly the same way. SAI also supports partition deletions, range tombstones, and row removals. If a delete operation is executed in a query, SAI

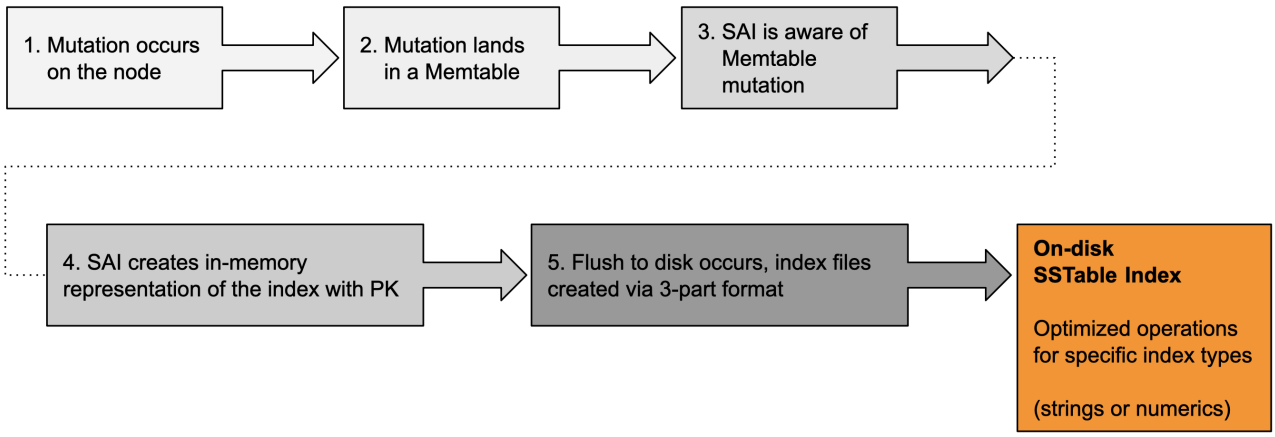
handles index changes in a post-filtering step. As a result, SAI imposes no special penalties when indexing frequently deleted columns.



If an insert or update contains valid content for the indexed column, the content is added to a **Memtable index**, and the primary key of the updated row is associated with the indexed value. SAI calculates an estimate of the incremental heap consumption of the new entry. This estimate counts against the heap usage of the underlying Memtable. This feature also means that as more columns are indexed on a table, the Memtable flush rate will increase, and the size of flushed SSTables will decrease. The number of total writes and the estimated heap usage of all live Memtable indexes are exposed as metrics. See **SAI metrics**.

Memtable flush

SAI flushes Memtable index contents directly to disk when the flush threshold is reached, rather than creating an additional in-memory representation. This is possible because Memtable indexes are sorted by term/value and then by primary key. When the flush occurs, SAI writes a new SSTable index for each indexed column, as the SSTable is being written.



The flush operation is a two-phase process. In the first phase, rows are written to the new SSTable. For each row, a row ID is generated and three index components are created. The components are:

- On-disk mappings of row IDs to their corresponding token values — SAI supports the Murmur3Partitioner
- SSTable partition offsets
- A temporary mapping of primary keys to their row IDs, which is used in a subsequent phase

The contents of the first and second component files are compressed arrays whose ordinals correspond to row IDs.

In the second phase, after all rows are written to the new SSTable and the shared SSTable-level index components have been completed, SAI begins its indexing work on each indexed column. Specifically in the second phase, SAI iterates over the Memtable index to produce pairs of terms and their token-sorted row IDs. This iterator translates primary keys to row IDs using the temporary mapping structure built in the first phase. The terms iterator (with postings) is then passed to separate writing components based on whether each indexed element is for a string or numeric column.

In the string case, the SAI index writer iterates over each term, first writing its postings to disk, and then recording the offset of those postings in the postings file as the payload of a new entry (for the term itself) in an on-disk, byte-ordered trie. In the numeric case, SAI separates the writing of a numeric index into two steps:

The terms are passed to a balanced kd-tree writer, which writes the kd-tree to disk. As the leaf blocks of the tree are written, their postings are recorded temporarily in memory. Those temporary postings are then used to build postings on-disk, at the leaves, and at various levels of the index nodes.

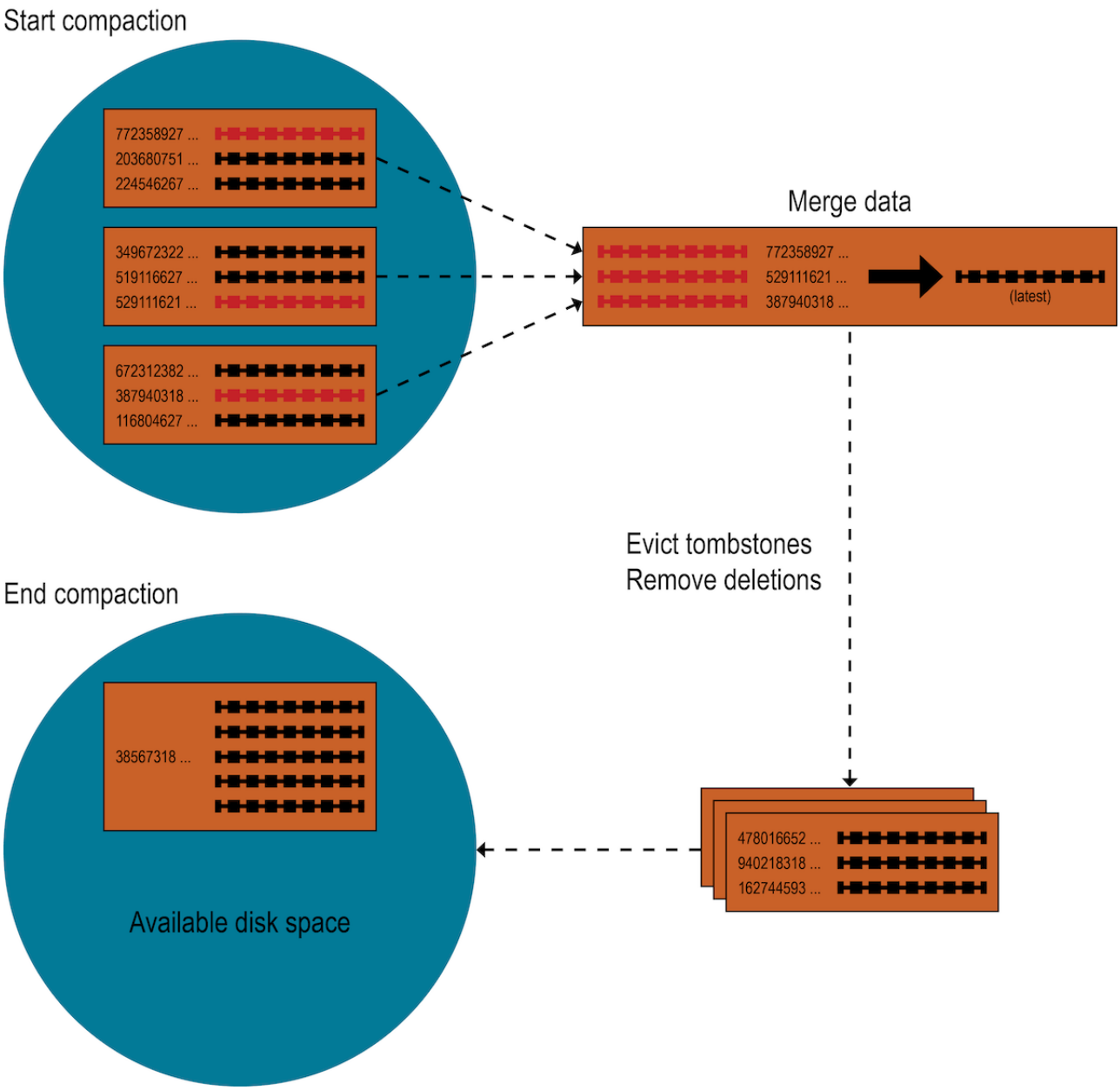
When a column index flush completes, a special empty marker file is flagged to indicate success. This procedure is used on startup and incremental rebuild operations to differentiate case where:

- SSTable index files are missing for a column.
- There is no indexable data — such as when an SSTable only contains tombstones. (A tombstone is a marker in a row that indicates a column was deleted. During compaction, marked columns are deleted.)

SAI then increments a counter on the number of Memtable indexes flushed for the column, and adds to a histogram the number of cells flushed per second.

When compaction is triggered

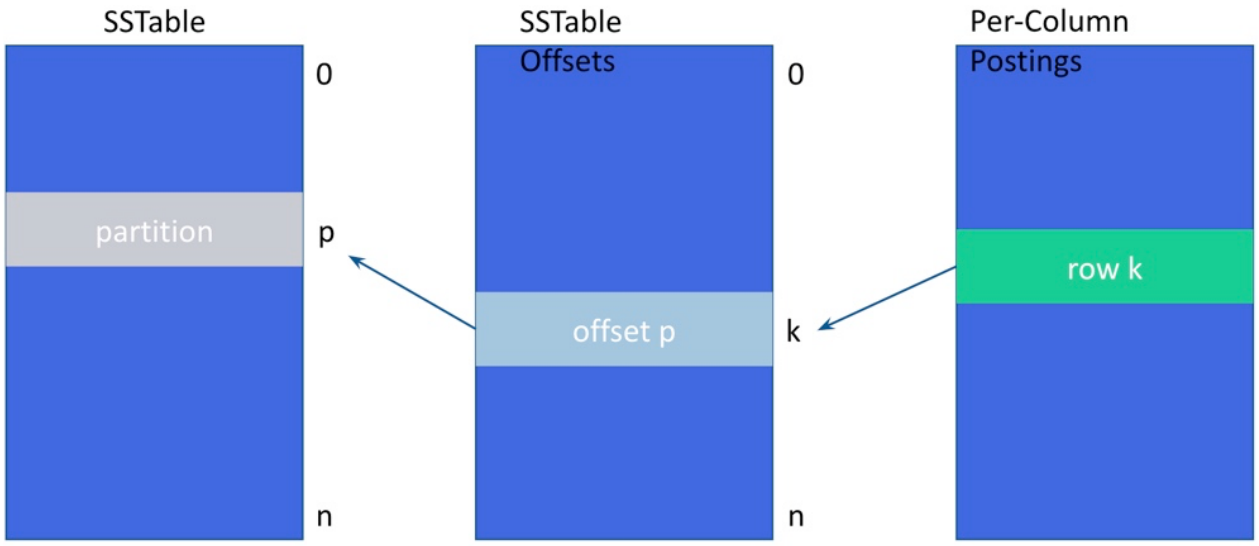
Recall that **Apache Cassandra** uses compaction to merge SSTables. Compaction collects all versions of each unique row and assembles one complete row, using the most up-to-date version (by timestamp) of each of the row's columns from the SSTables. The merge process is performant, because rows are sorted by partition key within each SSTable, and the merge process does not use random I/O. The new versions of each row is written to a new SSTable. The old versions, along with any rows that are ready for deletion, are left in the old SSTables, and are deleted when any pending reads are completed.



For SAI, when compaction is triggered, each index group creates an SSTable Flush Observer to coordinate the process of writing all attached column indexes from the newly compacted data in their respective SSTable writer instances. Unlike Memtable flush (where indexed terms are already sorted), when iterating merged data during compaction, SAI buffers indexed values and their row ids, which are added in token order.

To avoid issues such as exhausting available heap resources, SAI uses an accumulated segment buffer, which is flushed to disk synchronously by using a proprietary calculation. Then each segment records a segment row ID **offset** and only stores the segment row ID (SSTable row ID minus segment row ID offset). SAI flushes segments into the same file synchronously to avoid the cost of rewriting all segments and to reduce the cost of partition-restricted queries and paging range queries, as it reduces the search space.

The on-disk layout from the per-column indexed posting, to the SSTable offset, to the SSTable partition:



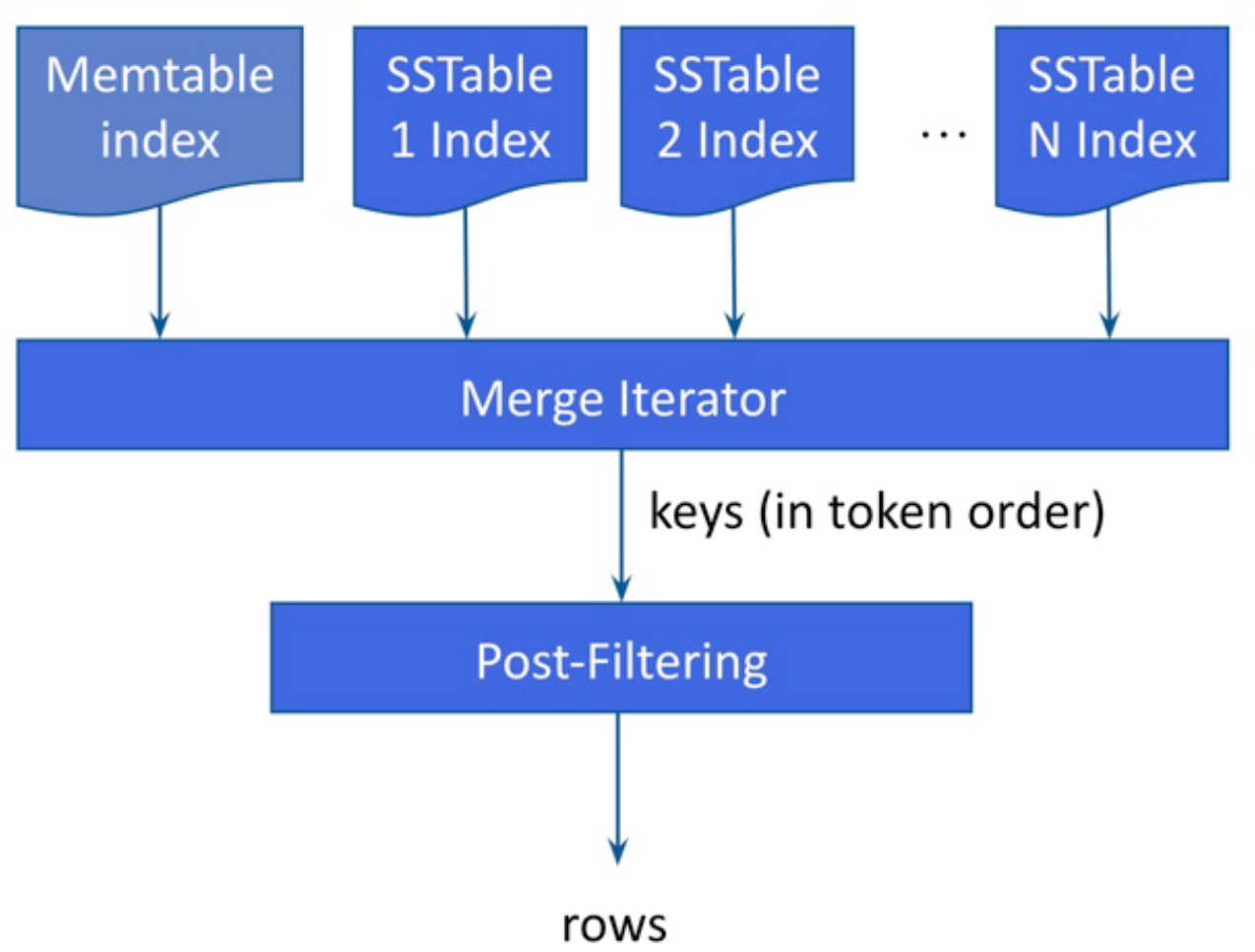
The actual segment flushing process is very similar to a Memtable flush. However, buffered terms are sorted before they can be written with their postings to their respective type-specific on-disk structures. At the end of compaction for a given index, a special empty marker file is flagged to indicate success, and the number of segments is recorded in SAI metrics. See **Global indexing metrics**.

When the entire compaction task finishes, SAI receives an SSTable List Changed Notification that contains the SSTables added and removed during the transaction. SSTable Context Manager and Index View Manager are responsible for replacing old SSTable indexes with new ones atomically. At this point, new SSTable indexes are available for queries.

SAI read path

This section explains how index queries are processed by the SAI coordinator and executed by replicas. Unlike legacy secondary indexes, where at most one column index will be used per query, SAI implements a `Query Plan` that makes it possible to use all available column indexes in a single query.

The overall flow of a SAI read operation is as follows:

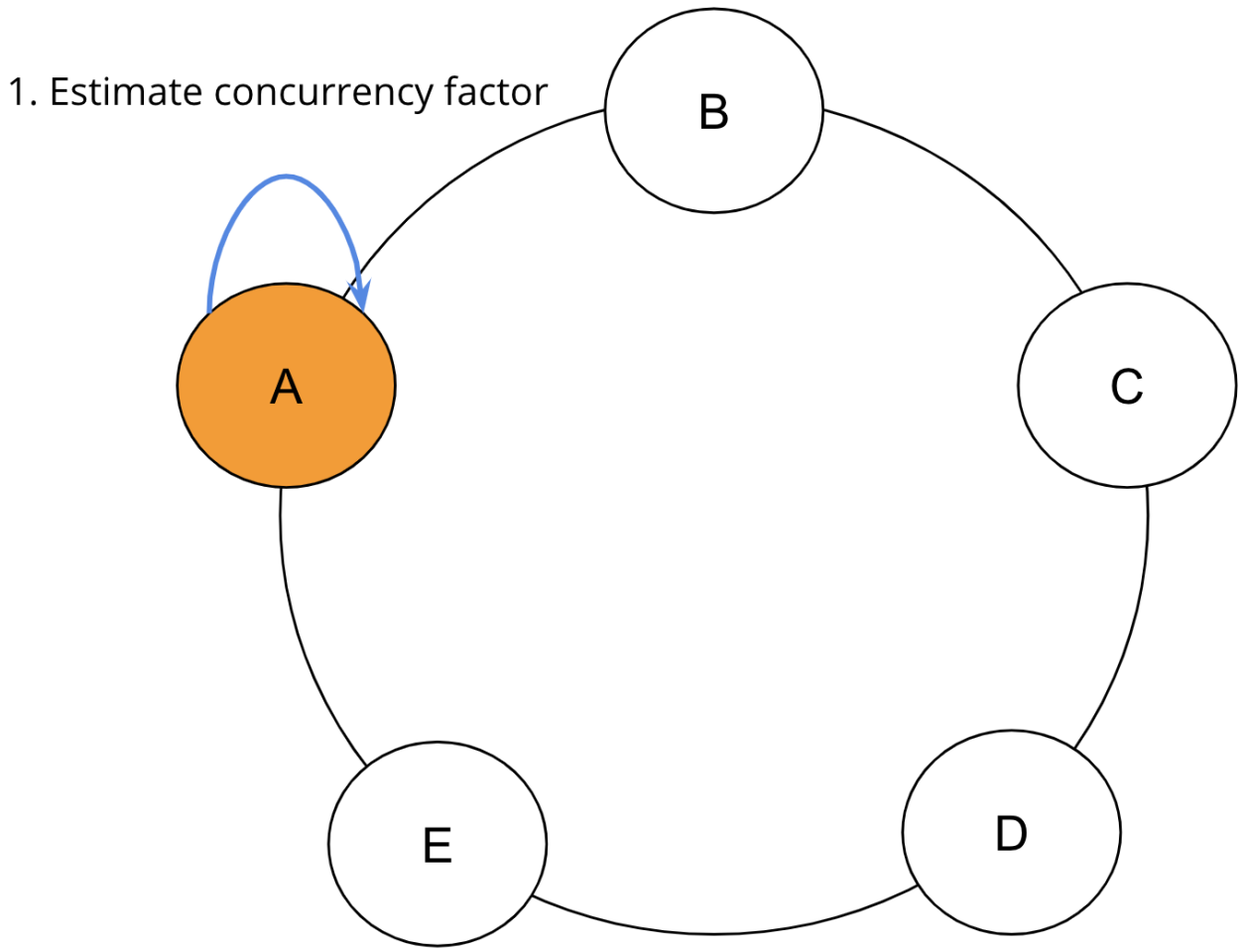


Index selection and Coordinator processing

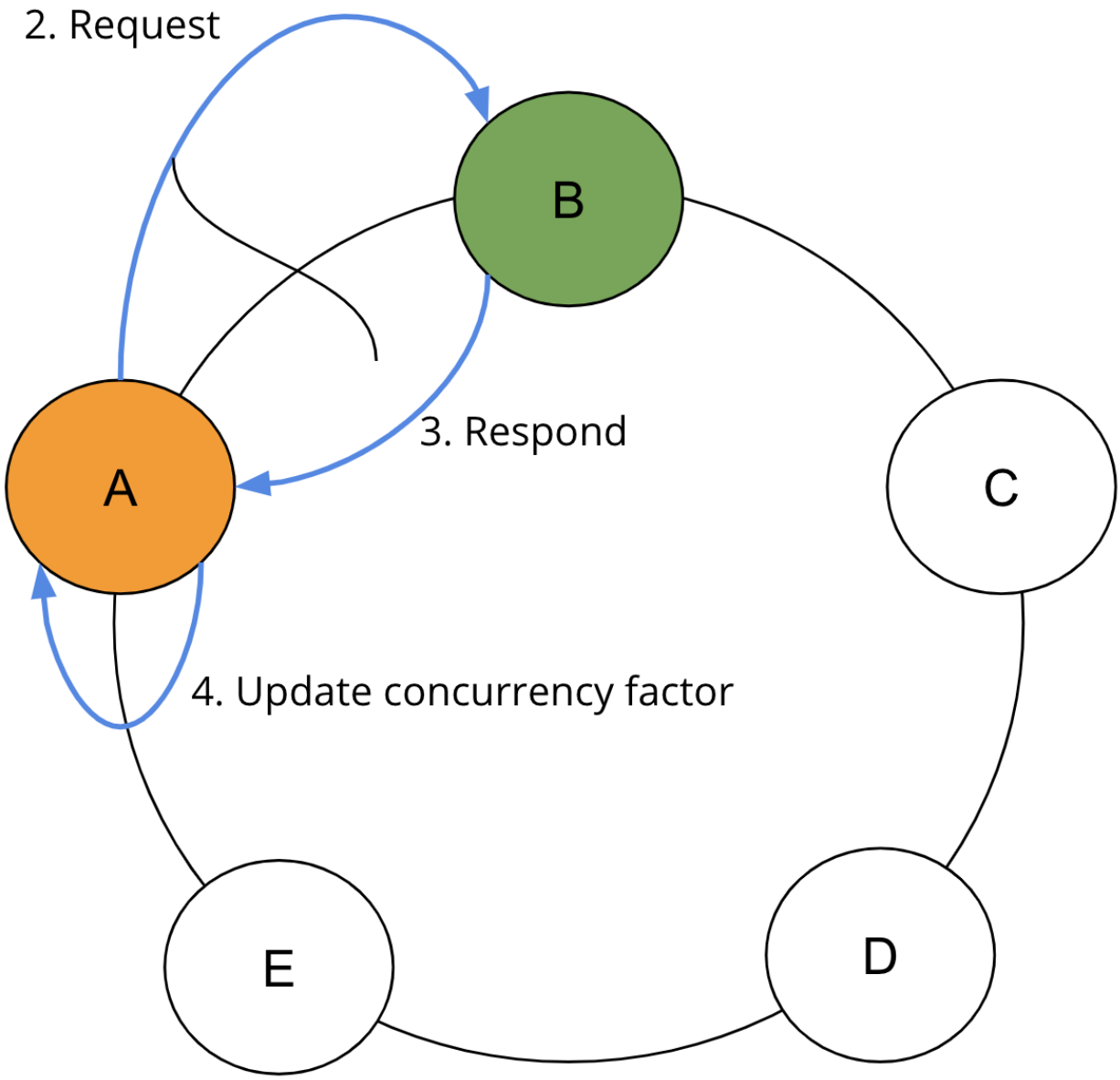
When presented with a query, the first action the SAI Coordinator performs, to take advantage of one or more indexes, is to identify the most selective index. That most selective index is the index that will most aggressively narrow the filtering space and the number of ultimate results by returning the lowest value from an estimated results row calculation. If multiple SAI indexes are present (that is, where each SAI index is based on a different column, but the query involves more than one column), it does not matter which SAI index is selected first.

Once the best index for the read operation is selected, the index is embedded with the read command, which enters the distributed range read apparatus. A distributed range read queries the **Apache Cassandra** cluster in one or more rounds in token order. The SAI Coordinator estimates the **concurrency factor**, the number of rows per range based on local data and the query limit to determine the number of ranges to contact. For each round, a concurrency factor determines how many replicas will be queried in parallel.

Before the first round commences, SAI estimates the initial concurrency factor via a proprietary calculation, shown here as Step 1.

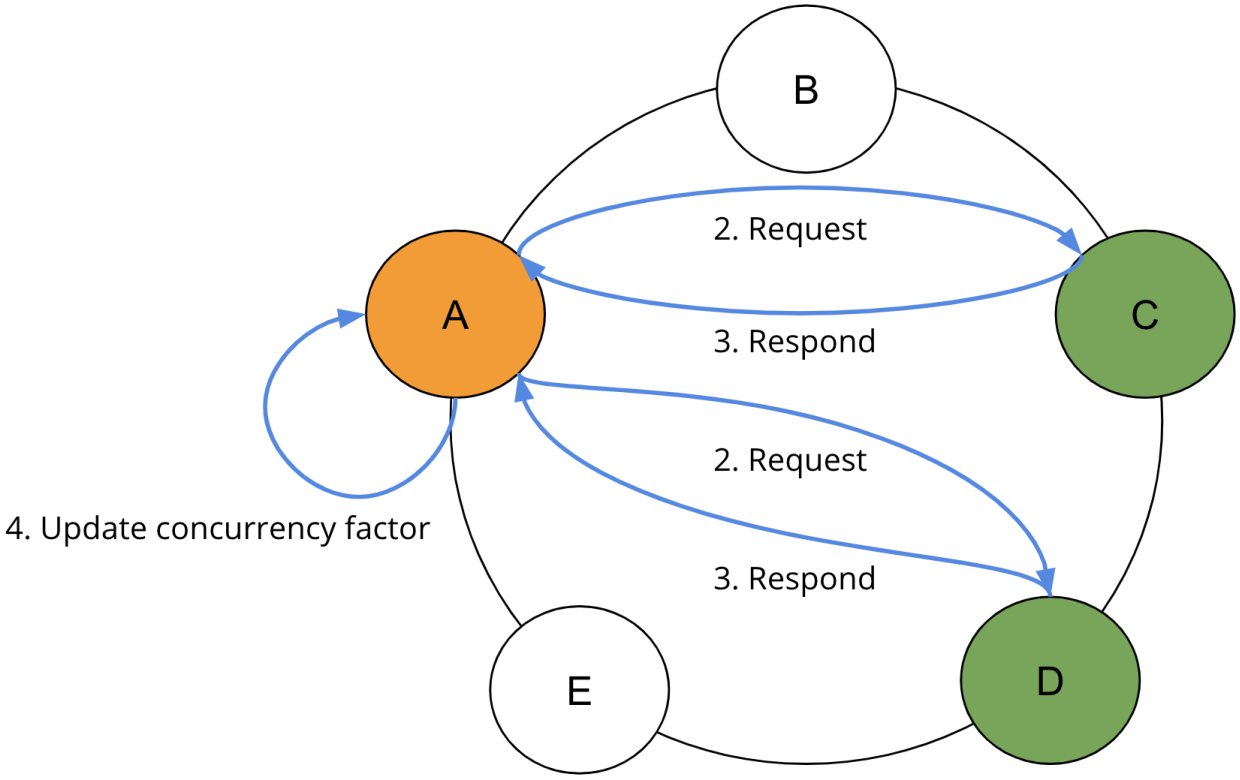


Once the initial concurrency factor established, the range read begins.



In Step 2, the SAI Coordinator sends requests to the required ranges in parallel based on the Concurrency factor. In Step 3, the SAI Coordinator waits for the responses from the requested replicas. And in Step 4, SAI Coordinator collects the results and recomputes the concurrency factor based on returned rows and query limit.

At the completion of each round, if the limit has not been reached, the concurrency factor is adjusted to take into account the shape of the results already read. If no results are returned from the first round, the concurrency factor is immediately increased to the minimum calculation of remaining token ranges and the maximum calculation of the concurrency factor. If results are returned, the concurrency factor is updated. SAI repeats steps 2, 3, and 4 until the query limit is satisfied.



To avoid querying replicas with failed indexes, each node propagates its own local index status to peers via gossip. At the coordinator, read requests will filter replicas that contain non-queryable indexes used in the request. In most cases, the second round of replica queries should return all the necessary results. Further rounds may be necessary if the distribution of results around the replicas is extremely imbalanced.

A closer look: replica query planning and view acquisition

Once a replica receives a token range read request from the SAI Coordinator, the local index query begins. SAI implements an index searcher interface via a Query Plan that makes it possible to access all available SAI column indexes in a single query.

The Query Plan performs an analysis of the expressions passed to it via the read command. SAI determines which indexes should be used to satisfy the query clauses on the given columns. Once column expressions are paired with indexes, a view of the active SSTable indexes for each column index is acquired by a Query Controller. In order to avoid compaction removing index files used by in-flight queries, before reading any index files, the Query Controller tries to acquire references to the SSTables for index files that intersect with the query's token range, and releases them when the read request completes.

At this point, a Token Flow is created to stream matches from each column index. Those flows, along with the Boolean logic that determines how they are merged, is wrapped up in an Operation, which is returned to the Query Plan component.

Role of the SAI Token Flow framework

The SAI query engine revolves around a Token Flow framework that defines how SAI asynchronously iterates over, skips into, and merges streams of matching partitions from both individual SSTable indexes and entire column indexes. SAI uses a token to describe a container for partition matches within a Cassandra ring token.

Iteration is the simplest of the three operations. Specifically, the iteration of postings involves sequential disk access — via the chunk cache — to row IDs, which are used to lookup ring token and partition key offset information.

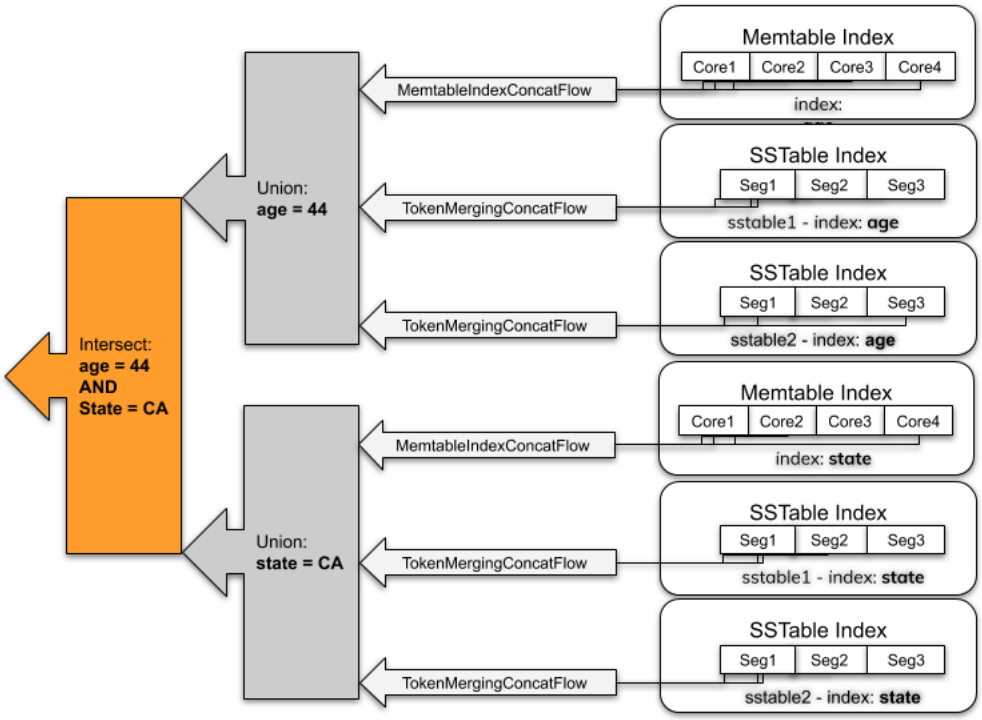
Token skipping is used to skip unmatched tokens when continuing from the previous paging boundary, or when a larger token is found during token intersection.

Match streaming and post filtering example

Consider an example with an individual column index (such as `age = 44`), the flow produced is the union of all Memtable indexes and all SSTable indexes.

- SAI iterates over each Memtable index "lazily" (that is, one at a time) in token order through its individual token range-partitioned instances. This feature reduces the overhead that would occur otherwise from unnecessary searches of data toward the end of the ring.
- On-disk index: SAI returns the union of all matching SSTable index. Within one SSTable index, there can be multiple segments because of the memory limit during compaction. Similar to the Memtable index, SAI lazily searches segments in token sorted order.

When there are multiple indexed expressions in the query (such as `WHERE age=44 AND state='CA'`) connected with `AND` query operator, the results of indexed expressions are intersected, which returns partition keys that match all indexed expressions.



After the index search, SAI exposes a flow of partition keys. For every single partition key, SAI executes a single partition read operation, which returns the rows in the given partition. As rows are materialized, SAI uses a filter tree to apply another round of filtering. SAI performs this subsequent filtering step to address the following:

- Partition granularity: SAI keeps track of partition offsets. In the case of a wide partition schema, not all rows in the partition will match the index expressions.
- Tombstones: SAI does not index tombstones. It's possible that an indexed row has been shadowed by newly added tombstones.
- Non-indexed expressions: Operations may include non-index expressions for which there are no index structures.

What's next?

See the blog, Better Cassandra Indexes for a Better Data Model: Introducing Storage-Attached Indexing.

Storage-attached Indexing (SAI) Quickstart

Storage-attached Indexing (SAI) Quickstart

To get started with Storage-Attached Indexing (SAI), we'll do the following steps:

- Create a keyspace.
- Create a table.
- Create an **index using SAI**.
- Add data.
- Create and run a query using SAI.

The examples in this quickstart topic show SAI indexes with non-partition key columns.

Create a keyspace

In `cqlsh`, define the `cycling` keyspace to try the commands in a test environment:

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

Create a database table

Using `cqlsh` or the CQL Console, create a `cyclist_semi_pro` database **table** in the `cycling` keyspace or the keyspace name of your choice:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_semi_pro (
  id int,
  firstname text,
  lastname text,
  age int,
  affiliation text,
  country text,
  registration date,
  PRIMARY KEY (id));
```

Create SAI indexes on the database table

To test a non-trivial query, you'll need some SAI indexes. Use **CREATE CUSTOM INDEX** commands to create SAI indexes on a few non-primary-key columns in the `cyclist_semi_pro` table:

```
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';
```

Let's take a look at the description of the table and its indexes:

Query

```
DESCRIBE TABLE cycling.cyclist_semi_pro;
```

Result

```
CREATE TABLE cycling.cyclist_semi_pro (
    id int PRIMARY KEY,
    affiliation text,
    age int,
    country text,
    firstname text,
    lastname text,
    registration date
) WITH additional_write_policy = '99PERCENTILE'
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    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.LZ4Compressor'}
```

```
    AND crc_check_chance = 1.0
    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 nodesync = {'enabled': 'true', 'incremental': 'true'}
    AND read_repair = 'BLOCKING'
    AND speculative_retry = '99PERCENTILE';
CREATE INDEX registration_sai_idx ON cycling.cyclist_semi_pro (registration) USING
'sai';
CREATE INDEX country_sai_idx ON cycling.cyclist_semi_pro (country) USING 'sai'
WITH OPTIONS = {'normalize': 'true', 'case_sensitive': 'false', 'ascii': 'true'};
CREATE INDEX age_sai_idx ON cycling.cyclist_semi_pro (age) USING 'sai';
CREATE INDEX lastname_sai_idx ON cycling.cyclist_semi_pro (lastname) USING 'sai'
WITH OPTIONS = {'normalize': 'true', 'case_sensitive': 'false', 'ascii': 'true'};
```

Add data to your table

Use CQLSH `INSERT` commands to add some data to the `cyclist_semi_pro` database table:

```
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (1, 'Carlos', 'Perotti', 22, 'Recco Club', 'ITA', '2020-01-12');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (2, 'Giovani', 'Pasi', 19, 'Venezia Velocità', 'ITA', '2016-05-15');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (3, 'Frances', 'Giardello', 24, 'Menaggio Campioni', 'ITA', '2018-
07-29');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (4, 'Mark', 'Pastore', 19, 'Portofino Ciclisti', 'ITA', '2017-06-
16');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (5, 'Irene', 'Cantona', 24, 'Como Velocità', 'ITA', '2012-07-22');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (6, 'Hugo', 'Herrera', 23, 'Bellagio Ciclisti', 'ITA', '2004-02-
12');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (7, 'Marcel', 'Silva', 21, 'Paris Cyclistes', 'FRA', '2018-04-28');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (8, 'Theo', 'Bernat', 19, 'Nice Cavaliers', 'FRA', '2007-05-15');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (9, 'Richie', 'Draxler', 24, 'Normandy Club', 'FRA', '2011-02-26');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (10, 'Agnes', 'Cavani', 22, 'Chamonix Hauteurs', 'FRA', '2020-01-
02');
```

```
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (11, 'Pablo', 'Verratti', 19, 'Chamonix Hauteurs', 'FRA', '2006-05-
15');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (12, 'Charles', 'Eppinger', 24, 'Chamonix Hauteurs', 'FRA', '2018-
07-29');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (13, 'Stanley', 'Trout', 30, 'Bolder Boulder', 'USA', '2016-02-12');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (14, 'Juan', 'Perez', 31, 'Rutgers Alumni Riders', 'USA', '2017-06-
16');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (15, 'Thomas', 'Fulton', 27, 'Exeter Academy', 'USA', '2012-12-15');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (16, 'Jenny', 'Hamler', 28, 'CU Alums Crankworkz', 'USA', '2012-07-
22');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (17, 'Alice', 'McCaffrey', 26, 'Pennan Power', 'GBR', '2020-02-12');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (18, 'Nicholas', 'Burrow', 26, 'Aberdeen Association', 'GBR', '2016-
02-12');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (19, 'Tyler', 'Higgins', 24, 'Highclere Agents', 'GBR', '2019-07-
31');
INSERT INTO cycling.cyclist_semi_pro (id, firstname, lastname, age, affiliation, country,
registration) VALUES (20, 'Leslie', 'Boyd', 18, 'London Cyclists', 'GBR', '2012-12-15');
```

Adding data in this quickstart topic shows simple `INSERT` commands. To load databases with many rows, consider using DataStax Bulk Loader for Apache Cassandra.

Try out CQL queries

Use the CQLSH `SELECT` command to submit queries.

TIP The supported query operators for tables with SAI indexes: * Numerics: `=`, `<`, `>`, `<=`, `>=`, `AND`, `OR`, `IN` * Strings: `=`, `CONTAINS`, `CONTAINS KEY`, `AND`, `OR`, `IN` * Strings or Numerics: `LIKE`

Find a specific semi-pro cyclist:

Query

```
SELECT * FROM cycling.cyclist_semi_pro WHERE lastname = 'Eppinger';
```

Result

```
id | affiliation       | age | country | firstname | lastname | registration
----+-------------------+-----+---------+-----------+----------+--------------
 12 | Chamonix Hauteurs |  24 |     FRA |   Charles | Eppinger |   2018-07-29

(1 rows)
```

Find semi-pro cyclists whose age is less than or equal to 23:

Query

```
SELECT * FROM cycling.cyclist_semi_pro WHERE age <= 23;
```

Result

```
id | affiliation        | age | country | firstname | lastname | registration
----+--------------------+-----+---------+-----------+----------+--------------
 10 |  Chamonix Hauteurs |  22 |     FRA |     Agnes |   Cavani |   2020-01-02
 11 |  Chamonix Hauteurs |  19 |     FRA |     Pablo | Verratti |   2006-05-15
  1 |         Recco Club |  22 |     ITA |    Carlos |  Perotti |   2020-01-12
  8 |     Nice Cavaliers |  19 |     FRA |      Theo |   Bernat |   2007-05-15
  2 |   Venezia Velocità |  19 |     ITA |   Giovani |     Pasi |   2016-05-15
  4 | Portofino Ciclisti |  19 |     ITA |      Mark |  Pastore |   2017-06-16
 20 |    London Cyclists |  18 |     GBR |    Leslie |     Boyd |   2012-12-15
  7 |    Paris Cyclistes |  21 |     FRA |    Marcel |    Silva |   2018-04-28
  6 |  Bellagio Ciclisti |  23 |     ITA |      Hugo |  Herrera |   2004-02-12

(9 rows)
```

Find semi-pro cyclists from Great Britain:

Query

```
SELECT * FROM cycling.cyclist_semi_pro WHERE country = 'GBR';
```

Result

```
id | affiliation          | age | country | firstname | lastname  | registration
----+----------------------+-----+---------+-----------+-----------+--------------
 19 |     Highclere Agents |  24 |     GBR |     Tyler |   Higgins |   2019-07-31
 18 | Aberdeen Association |  26 |     GBR |  Nicholas |    Burrow |   2016-02-12
```

```
 20 |      London Cyclists |  18 |     GBR |    Leslie |      Boyd |   2012-12-15
 17 |         Pennan Power |  26 |     GBR |     Alice | McCaffrey |   2020-02-12

(4 rows)
```

Find semi-pro cyclists who registered between a given date range:

Query

```
SELECT * FROM cycling.cyclist_semi_pro WHERE registration > '2010-01-01' AND
registration < '2015-12-31' LIMIT 10;
```

Result

```
id | affiliation         | age | country | firstname | lastname | registration
----+---------------------+-----+---------+-----------+----------+--------------
  5 |       Como Velocità |  24 |     ITA |     Irene |  Cantona |   2012-07-22
 16 | CU Alums Crankworkz |  28 |     USA |     Jenny |   Hamler |   2012-07-22
 15 |      Exeter Academy |  27 |     USA |    Thomas |   Fulton |   2012-12-15
 20 |     London Cyclists |  18 |     GBR |    Leslie |     Boyd |   2012-12-15
  9 |       Normandy Club |  24 |     FRA |    Richie |  Draxler |   2011-02-26

(5 rows)
```

TIP For query examples with `CONTAINS` clauses that take advantage of SAI collection maps, lists, and sets, be sure to see **SAI collection map examples with keys, values, and entries**.

Removing an SAI index

To remove an SAI index, use `DROP INDEX`.

Example:

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

Resources

SAI overview

SAI FAQ

SAI FAQ

Use this FAQ to find answers to common questions and get help with Storage-Attached Indexing (SAI).

What is SAI?

Storage-Attached Indexing (SAI) is a highly-scalable, globally-distributed index for the **Cassandra** database. SAI combines:

- the storage-attached architecture of open source SSTable Attached Secondary Indexes (SASI)
- a number of highly optimized on-disk index structures

Which databases are supported?

Cassandra 5.0 is the only supported database currently.

After creating your database, a keyspace, and one or more tables, use `CREATE INDEX ... USING 'sai'` to define one or more SAI indexes on the table. For Cassandra databases, use `cqlsh`. The same `CREATE INDEX ... USING 'sai'` command is available for both. See **SAI quickstart**.

What configuration settings should I use with SAI?

Compared with most indexing environments, SAI configuration and related settings are much simpler. Key points:

- Increase `--XX:MaxDirectMemorySize`, leaving approximately 15-20% of memory for the OS and other in-memory structures.
- In cassandra.yaml, explicitly set `file_cache_size_in_mb` to 75% of that value.
- Heavy mixed read/write workloads may want to:
  - Decrease `range_request_timeout_in_ms`
  - Increase `write_request_timeout_in_ms`
- If the `memtable_flush_writers` value is set too low, writes may stall. If this occurs in your environment, increase `memtable_flush_writers`.

Aside from memory, SAI uses the same tunable parameters for Cassandra, such as compaction throughput and compaction executors. This matters for write performance. For read performance, again, maximizing use of the Chunk Cache will benefit SAI reads because all on-disk index components

are accessed through this mechanism. Refer to **Configure SAI indexes**.

What computing challenges does SAI solve?

Oftentimes, developers ask: "How can I query additional fields outside of the **Apache Cassandra** partition key?"

SAI implements efficient indexes based on a table's columns, such as parts of a composite partition key. Before SAI, you could index clustering keys, but you could not index parts of a composite partition. The development of SAI was inspired by SASI to achieve the goal of efficient and simpler filtering via the creation of secondary indexes.

SAI also makes data modeling easier because you do not need to create custom tables just to cater to particular query patterns. You can create a table that is most natural for you, write to just that table, and query it any way you want.

What are the advantages of using SAI?

SAI makes it possible to define multiple indexes on the same database table. Each SAI index can be based on any column in the table. Exception: there is no need to define an SAI index based on the partition key when it's comprised of only one column; in this case, SAI issues an `invalid query` message. You can also define an SAI index using a single column in the table's composite partition key. A composite partition key means that the partition is based on two or more columns. In this case with an SAI index, you would specify just one of the columns that comprises the composite partition key.

For developers, SAI removes several previous pain points, including the need to duplicate denormalized data to query non-PrimaryKey columns.

For operators, SAI has several advantages, including the use of significantly less disk space for indexing; fewer failure points; easier uptime due to the simplified architecture of SAI; and fewer copies of data to secure.

Is SAI a complete search solution?

SAI is not an enterprise search engine. While it does provide some of the same functionality, SAI is not a complete replacement for text-based search. At its core, SAI is a filtering engine, and simplifies data modeling and client applications that would otherwise rely heavily on maintaining multiple query-specific tables.

How does schema management compare between SAI and text-based search?

SAI is an index, not a search engine. Unlike the text-based search, SAI has no need for schema management. SAI configuration is simpler and is tuned with existing database parameters, such as in cassandra.yaml. With SAI, there is no commit log to accept writes during bootstrap; SAI does not need to wait for bootstrap to read the database configuration. With SAI, schema/indexing options reside in the index metadata, which is handled by native database schema management.

How do I use SAI features?

Storage-Attached Indexing has queries are entirely CQL-based. The features, by design, are intentionally simple and easy to use.

At a high level, SAI indexes are:

- Created and dropped per column via CQL `CREATE INDEX ... USING 'sai'` commands and `DROP INDEX` commands. Start in **SAI quickstart**.
- Rebuilt and backed up via nodetool. See **nodetool**.
- Monitored via a combination of nodetool, CQL virtual tables, system metrics, and JMX. See **Monitor SAI indexes**.

On which column in a database table can I base an SAI index?

Define each SAI index on any table column. Exception: there is no need to define an SAI index based on the partition key when it's comprised of only one column; in this case, SAI issues an `invalid query` message.

You can also define an SAI index using a single column in the table's composite partition key. A composite partition key means that the partition is based on two or more columns. In this case with an SAI index, you would specify just one of the columns that comprises the composite partition key.

With collection maps, you can define one or more SAI indexes on the same column, specifying `keys`, `values`, and `entries` as map types. SAI also supports `list` and `set` collections.

TIP

In CQL queries of database tables with 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

When I DROP and recreate an SAI index on the same column, does that block any read operations? And is there a way to check the indexing status?

When you DROP / recreate an SAI index, you are not blocked from entering queries that do not use the index. However, you cannot use that SAI index (based on the same column) until it has finished building and is queryable. To determine the current state of a given index, query the system_views.indexes virtual table. Example:

```
SELECT is_queryable,is_building FROM system_views.indexes WHERE keyspace_name='keyspace'
       AND table_name='table' AND index_name='index';
```

See **Virtual tables** and **Virtual tables for SAI indexes and SSTables**.

What are the write and read paths used by SAI indexes?

SAI indexes Memtables and SSTables as they are written, resolving the differences between those indexes at read time. See **SAI write path and read path**.

What on-disk index formats does SAI support?

SAI supports two on-disk index formats, optimized for:

- Equality and non-exact matching on **strings**.
  - Strings are indexed on-disk using the trie data structure, in conjunction with postings (term/row pairs) lists. The trie is heap friendly, providing string prefix compression for terms, and can match any query that can be expressed as a deterministic finite automaton. The feature minimizes on-disk footprint and supports simple token skipping.
- Equality and range queries on **numeric and non-literal types**.
  - Numeric values and the other non-literal CQL types (timestamp, date, UUID) are indexed on-disk using k-dimensional tree, a balanced structure that provides fast lookups across one or more dimensions, and compression for both values and postings.

What is the disk footprint overhead for SAI indexes?

SAI requires significantly lower disk usage compared to other native or bolt-on Cassandra index solutions. SAI produces an additional 20-35% disk usage compared with **unindexed** data. The SAI disk usage is largely dependent on the underlying data model and the number of columns indexed.

What are the supported column data types for SAI indexing?

The supported types are: `ASCII`, `BIGINT`, `DATE`, `DECIMAL`, `DOUBLE`, `FLOAT`, `INET`, `INT`, `SMALLINT`, `TEXT`, `TIME`, `TIMESTAMP`, `TIMEUUID`, `TINYINT`, `UUID`, `VARCHAR`, `VARINT`.

NOTE

- `INET` support for IPv4 and IPv6 was new starting with Cassandra ???.
- The `DECIMAL` and `VARINT` support was new starting with Cassandra ???.
- SAI also supports collections — see the **next FAQ**.

Does SAI support indexes on a collection column?

Yes — SAI supports collections of type `map`, `list`, and `set`. See the following topics:

- **Using collections with SAI**

TIP

In CQL queries of database tables with 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

What are the supported query operators?

For queries on tables with SAI indexes:

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

The unsupported query operators are:

- Strings or Numerics: `LIKE`

Examples:

```
SELECT * FROM cycling.cyclist_semi_pro WHERE registration > '2010-01-01' AND registration
< '2015-12-31' LIMIT 10;
```

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

TIP For query examples with `CONTAINS` clauses that take advantage of SAI collection maps, lists, and sets, be sure to see **SAI collection map examples with keys, values, and entries**.

On the `CREATE INDEX` command for SAI, what options are available?

Use the `WITH OPTIONS` clause to indicate how SAI should handle case sensitivity and special characters in the index. For example, given a string column `lastname`:

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

TIP SAI has an `ascii` option. The default is `false`. 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`.

See **CREATE CUSTOM INDEX** and examples in the **SAI quickstart** topic.

Does SAI support composite indexing: meaning, a single index on multiple columns?

No. There is a 1-to-1 mapping of an SAI index to a column. However, you can create a separate index on each column in a given table. Also, SAI can use multiple defined indexes within a single read query.

How can I view SAI memory usage metrics?

The SAI memory footprint is divided between the JVM heap and the Chunk Cache. The heap stores memtable indexes, and the chunk cache stores recently accessed on-disk index components as well as other SSTable components. SAI provides metrics for both the heap and the chunk cache. For each index, SAI also provides metrics for determining the size in bytes of memory used by the on-disk data structure, as well as disk usage. Refer to **Index group metrics**. SAI also provides Table state metrics that give you visibility into the disk usage, the percentage of disk usage of the base table, the index builds in progress, and related metrics. See **Table state metrics**.

What is the performance impact of adding SAI columns to a read query? How many `AND` clauses can I add?

There is no limit on the number of index columns that can be used in a single query. The `sai_indexes_per_table_failure_threshold` setting in cassandra.yaml controls the maximum number of SAI indexes allowed in a single table (10, by default). However, querying against multiple indexed columns incurs a cost that is related to the increased number of index components processed. When evaluating multiple indexed columns in a query, SAI performs a workflow (1: Traverse. 2: Merged. 3: Intersect that ultimately coalesces data from multiple memtables and SSTables.

NOTE	In a query, AND queries will process up to two SAI indexes; if more than two SAI indexes are used by the query, this circumstance will result in SAI performing post-filtering on the remaining clauses.

For related information, see the **match streaming and post filtering** example.

Are SAI write operations asynchronous, or does SAI wait before acknowledging the write to the user?

The SAI write path is actually very simple. The indexes live with the data, both in memtables and SSTables. When a write is acknowledged to the client, the data is already indexed. This is a **synchronous** process. When the memtable is flushed, the indexes are flushed as well. See **SAI write path and read path**.

The on-disk index components are broken down into per-SSTable index files and per-column index files. The column indexes do not store the primary keys or tokens; instead, they store compressible row IDs. The per-SSTable index files link the row IDs from the column-indexes to their backing SSTables.

This SAI design allows all column indexes within a single SSTable to share per-SSTable index files, which further helps reduce the disk footprint.

What are the guidelines regarding column cardinality with SAI indexes?

Column cardinality can affect read performance when it comes to range queries among replicas. The number of rows matching a value of a high-cardinality column, such as credit card numbers, is more likely to be isolated on very few nodes (or even isolated to one node), while the rows matching a value on a low-cardinality column are more likely to reside on numerous nodes. If a query does not specify a partition key, the Cassandra coordinator scans the token ring and group token ranges by endpoints (nodes). The coordinator then concurrently execute read commands for all participating endpoints. In the worst case where the indexed column has very high cardinality, an entire cluster scan may be required before finding a match. With low-cardinality columns, be aware that if your `LIMIT` is higher than the number of values in your targeted column, Cassandra has to search all replicas again before determining that the `LIMIT` cannot be satisfied. In this case, Cassandra returns only the number of matching results.

What are the circumstances under which SAI applies post filtering?

SAI applies post-filtering in numerous scenarios. For example, consider a simple table, and an SAI index on just one of the two non-PK columns:

```
CREATE KEYSPACE test WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor':
1};
CREATE TABLE test.mytable (id int PRIMARY KEY,
       col1 text,
       col2 timestamp);
CREATE CUSTOM INDEX mytable_col1_idx ON test.mytable (col1) USING 'StorageAttachedIndex';
```

Given a query such as the following:

```
SELECT * FROM test.mytable WHERE col1 = 'hello world' and col2 < toTimestamp(now()) ALLOW
FILTERING;
```

For this query, Cassandra narrows down the search by the indexed column (`col1`) first, then applies post-filtering on `col2`. (Use the `ALLOW FILTERING` clause with caution.) In this scenario, no additional replica roundtrips are needed; the post filtering on `col2` is carried out on the replicas themselves.

Another case were post-filtering comes into play is when constructing a query that involves more than two SAI indexes. Refer to this **related FAQ** about `AND` queries.

Can I create an SAI index based on a static column?

Yes. For example, consider a `transaction_by_customer` table where you have a primary key `customer_id`, plus static columns to contain each customer's `address`, `phone_number`, and `date_of_birth`. Given a query like:

```
SELECT * from transaction_by_customer where customer_id = 'xyz123';
```

If there are 100,000 `transaction_by_customer` rows, because you defined those three static fields, this query runs against a table that uses significantly less disk space, as compared to an environment where writes had inserted the per-customer values (`address`, `phone_number`, `date_of_birth`) in every row.

NOTE	SAI delivers the option and advantage of creating indexes based on static columns, while also achieving the benefit of conserving table space.

For indexed strings, how does SAI handle Unicode characters in the column data?

When you create an SAI index based on a string column, set the `normalize` option to `true` if you want SAI to perform Unicode normalization on the column data. 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). See **CREATE CUSTOM INDEX**.

Can the index's column name have special characters?

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

What partitioner does SAI support?

SAI supports only the `Murmur3Partitioner`.

Working with Storage-attached indexing (SAI)

Working with Storage-attached indexing (SAI)

Prerequisites

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

Create SAI index

To create an SAI index, you must define the index name, table name, and column name for the column to be indexed.

To create a simple SAI index:

```
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 most SAI indexes, the column name is defined in the `CREATE INDEX` statement that also uses `USING 'sai'`. The SAI index options are defined in the `WITH OPTIONS` clause. The `case_sensitive` option is set to `false` to allow case-insensitive searches. The `normalize` option is set to `true` to allow searches to be normalized for Unicode characters. The `ascii_only` option is set to `true` to allow searches to be limited to ASCII characters.

The `map` collection data type is the one exception, as shown in the **example below**.

Partition key SAI error

SAI indexes cannot be created on the partition key, as a primary index already exists and is used for queries. If you attempt to create an SAI on the partition key column, an error will be returned:

CQL

```
CREATE INDEX ON demo2.person_id_name_primarykey (id)
  USING 'sai';
```

Result

```
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot
create secondary index on the only partition key column id"
```

`map` collection in SAI index

Map collections do have a different format than other SAI indexes:

```
// Create an index on a map key to find all cyclist/team combos for a year
// tag::keysidx[]
CREATE INDEX IF NOT EXISTS team_year_keys_idx
ON cycling.cyclist_teams ( KEYS (teams) );
// end::keysidx[]

// Create an index on a map key to find all cyclist/team combos for a year
// tag::valuesidx[]
CREATE INDEX IF NOT EXISTS team_year_values_idx
ON cycling.cyclist_teams ( VALUES (teams) );
// end::valuesidx[]

// Create an index on a map key to find all cyclist/team combos for a year
// tag::entriesidx[]
CREATE INDEX IF NOT EXISTS team_year_entries_idx
ON cycling.cyclist_teams ( ENTRIES (teams) );
// end::entriesidx[]
```

similarity-function for vector search

This example uses the following table:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
```

To check if `comment_vector` has a particular similarity function set, use the `similarity-function` option set to one of the supported similarity functions: DOT_PRODUCT, COSINE, or EUCLIDEAN. The default similarity function is COSINE.

This index creates an index on the `comment_vector` column with the similarity function set to DOT_PRODUCT:

```
CREATE INDEX sim_comments_idx
    ON cycling.comments_vs (comment_vector)
    USING 'sai'
    WITH OPTIONS = { 'similarity_function': 'DOT_PRODUCT'};
```

Other resources

See **CREATE CUSTOM INDEX** for more information about creating SAI indexes.

Alter SAI index

SAI indexes cannot be altered. If you need to modify an SAI index, you will need to drop the current index, create a new index, and rebuild the cycling.

1. Drop index

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

2. Create new index

   ```
   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';
```

Drop SAI index

SAI indexes can be dropped (deleted).

To drop an SAI index:

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

This command does not return a result.

Querying with SAI

The **SAI quickstart** focuses only on defining multiple indexes based on non-primary key columns (a very useful feature). Let's explore other options using some examples of how you can run queries on tables that have differently defined SAI indexes.

IMPORTANT SAI only supports `SELECT` queries, but not `UPDATE` or `DELETE` queries.

Vector search

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
```

```
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX IF NOT EXISTS ann_index
  ON cycling.comments_vs(comment_vector) USING 'sai';
```

Query vector data with CQL

To query data using Vector Search, use a `SELECT` query:

CQL

```
SELECT * FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.15, 0.1, 0.1, 0.35, 0.55]
    LIMIT 3;
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-----------------------------------------+-------------------------------+
-----------+----------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-04-01 14:33:02.160000+0000 | LATE
RIDERS SHOULD NOT DELAY THE START | [0.9, 0.54, 0.12, 0.1, 0.95] |      Alex |
616e77e0-22a2-11ee-b99d-1f350647414a
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-02-17 08:43:20.234000+0000 |
Glad you ran the race in the rain | [0.3, 0.34, 0.2, 0.78, 0.25] |       Amy |
6170c1d0-22a2-11ee-b99d-1f350647414a
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-04-01 13:43:08.030000+0000 |
Last climb was a killer |   [0.3, 0.75, 0.2, 0.2, 0.5] |       Amy | 62105d30-
22a2-11ee-b99d-1f350647414a
```

NOTE The limit has to be 1,000 or fewer.

Scrolling to the right on the results shows the comments from the table that most closely matched the embeddings used for the query.

Single index match on a column

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

The column `commenter` is not the partition key in this table, so an index is required to query on it.

Query for a match on that column:

Query

```
SELECT * FROM cycling.comments_vs
        WHERE commenter = 'Alex';
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-----------------------------------------+-----------------------------+
-----------+--------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-04-01 14:33:02.160000+0000 | LATE
RIDERS SHOULD NOT DELAY THE START | [0.9, 0.54, 0.12, 0.1, 0.95] |      Alex |
6d0cdaa0-272b-11ee-859f-b9098002fcac
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 |
```

```
Second rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex |
6d0b7b10-272b-11ee-859f-b9098002fcac
```

Failure with index

Note that a failure will occur if you try this query before creating the index:

Query

```
SELECT * FROM cycling.comments_vs
        WHERE commenter = 'Alex';
```

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"
```

Single index match on a column with options

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_cs_idx ON cycling.comments_vs (commenter)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'true', 'normalize': 'true', 'ascii': 'true'};
```

Case-sensitivty

The column `commenter` is not the partition key in this table, so an index is required to query on it. If we want to check `commenter` as a case-sensitive value, we can use the `case_sensitive` option set to `true`.

Note that no results are returned if you use an inappropriately case-sensitive value in the query:

Query

```
SELECT * FROM comments_vs WHERE commenter ='alex';
```

Result

```
id | created_at | comment | comment_vector | commenter | record_id
----+------------+---------+----------------+-----------+-----------

(0 rows)
```

When we switch the case of the cyclist's name to match the case in the index, the query succeeds:

Query

```
SELECT comment,commenter FROM comments_vs WHERE commenter ='Alex';
```

Result

```
comment                                | commenter
---------------------------------------+-----------
 LATE RIDERS SHOULD NOT DELAY THE START |      Alex
      Second rest stop was out of water |      Alex
(2 rows)
```

Index match on a composite partition key column

This example uses the following table and indexes:

```
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 race_name_idx
    ON cycling.rank_by_year_and_name (race_name)
    USING 'sai';
CREATE INDEX race_year_idx
    ON cycling.rank_by_year_and_name (race_year)
    USING 'sai';
```

Composite partition keys have a partition defined by multiple columns in a table. Normally, you would need to specify all the columns in the partition key to query the table with a `WHERE` clause. However, an SAI index makes it possible to define an index using a single column in the table's composite partition key. You can create an SAI index on each column in the composite partition key, if you need to query based on just one column.

SAI indexes also allow you to query tables without using the inefficient `ALLOW FILTERING` directive. The `ALLOW FILTERING` directive requires scanning all the partitions in a table, leading to poor performance.

The `race_year` and `race_name` columns comprise the composite partition key for the `cycling.rank_by_year_and_name` table.

Query for a match on the column `race_name`:

Query

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

Result

```
race_year | race_name                                  | rank | cyclist_name
-----------+--------------------------------------------+------+
----------------------
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |        Daniel
MARTIN
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 | Johan Esteban
CHAVES
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |      Benjamin
PRADES
      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
```

Query for a match on the column `race_year`:

Query

```
SELECT * FROM cycling.rank_by_year_and_name
        WHERE race_year = 2014;
```

Result

```
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
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |        Daniel
MARTIN
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 | Johan Esteban
CHAVES
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |      Benjamin
PRADES
```

Multiple indexes matched with AND

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
```

```
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

Several indexes are created for the table to demonstrate how to query for matches on more than one column.

Query for matches on more than one column, and both columns must match:

Query

```
SELECT * FROM cycling.comments_vs
    WHERE
    created_at='2017-03-21 21:11:09.999000+0000'
    AND commenter = 'Alex';
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-----------------------------------+-----------------------------+-----------+
--------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 | Second
rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex | 6d0b7b10-
272b-11ee-859f-b9098002fcac
```

Multiple indexes matched with OR

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
```

```
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

Several indexes are created for the table to demonstrate how to query for matches on more than one column.

Query for a match on either one column or the other:

Query

```
SELECT * FROM cycling.comments_vs
    WHERE
    created_at='2017-03-21 21:11:09.999000+0000'
    OR created_at='2017-03-22 01:16:59.001000+0000';
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-----------------------------------+-----------------------------+-----------+
-------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 | Second
rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex | 6d0b7b10-
272b-11ee-859f-b9098002fcac
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-03-22 01:16:59.001000+0000 |
Great snacks at all reststops |  [0.1, 0.4, 0.1, 0.52, 0.09] |       Amy |
6d0fc0d0-272b-11ee-859f-b9098002fcac
```

Multiple indexes matched with IN

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

Several indexes are created for the table to demonstrate how to query for matches on more than one column.

Query for match with column values in a list of values:

Query

```
SELECT * FROM cycling.comments_vs
    WHERE created_at IN
    ('2017-03-21 21:11:09.999000+0000'
    ,'2017-03-22 01:16:59.001000+0000');
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
---------------------------------+-----------------------------+-----------+
-------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 | Second
```

```
rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex | 6d0b7b10-
272b-11ee-859f-b9098002fcac
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-03-22 01:16:59.001000+0000 |
Great snacks at all reststops |  [0.1, 0.4, 0.1, 0.52, 0.09] |       Amy |
6d0fc0d0-272b-11ee-859f-b9098002fcac
```

User-defined type

SAI can index either a user-defined type (UDT) or a list of UDTs. This example shows how to index a list of UDTs.

This example uses the following user-defined type (UDT), table and index:

```
CREATE TYPE IF NOT EXISTS cycling.race (
  race_title text,
  race_date timestamp,
  race_time text
);
CREATE TABLE IF NOT EXISTS cycling.cyclist_races (
  id UUID PRIMARY KEY,
  lastname text,
  firstname text,
  races list<FROZEN <race>>
);
CREATE INDEX races_idx
    ON cycling.cyclist_races (races)
    USING 'sai';
```

An index is created on the list of UDTs column `races` in the `cycling.cyclist_races` table.

Query with `CONTAINS` from the list `races` column:

CQL

```
SELECT * FROM cycling.cyclist_races
    WHERE races CONTAINS {
       race_title:'Rabobank 7-Dorpenomloop Aalburg',
       race_date:'2015-05-09',
       race_time:'02:58:33'};
```

Result

```
id                                   | firstname | lastname | races
--------------------------------------+-----------+----------+
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
-------------------------------------------------------------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS | [{race_title:
'Rabobank 7-Dorpenomloop Aalburg', race_date: '2015-05-09 00:00:00.000000+0000',
race_time: '02:58:33'}, {race_title: 'Ronde van Gelderland', race_date: '2015-04-
19 00:00:00.000000+0000', race_time: '03:22:23'}]

(1 rows)
```

SAI indexing with collections

SAI supports collections of type `map`, `list`, and `set`. Collections allow you to group and store data together in a column.

In a relational database, a grouping such as a user's multiple email addresses is achieved via many-to-one joined relationship between (for example) a `user` table and an `email` table. **Apache 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.

TIP

In CQL queries of database tables with 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

Using the set type

This example uses the following table and index:

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

```
CREATE INDEX teams_idx
    ON cycling.cyclist_career_teams (teams)
    USING 'sai';
```

An index is created on the set column `teams` in the `cyclist_career_teams` table.

Query with `CONTAINS` from the set `teams` column:

CQL

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

Result

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

Using the list type

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.upcoming_calendar (
  year int,
  month int,
  events list<text>,
  PRIMARY KEY (year, month)
);
CREATE INDEX events_idx
    ON cycling.upcoming_calendar (events)
    USING 'sai';
```

An index is created on the list column `events` in the `upcoming_calendar` table.

Query with `CONTAINS` from the list `events` column:

CQL

```
SELECT * FROM cycling.upcoming_calendar
   WHERE events CONTAINS 'Criterium du Dauphine';
```

Result

```
year | month | events
------+-------+-------------------------------------------------
 2015 |     6 | ['Criterium du Dauphine', 'Tour de Sui\nsse']
```

A slightly more complex query selects rows that either contain a particular event or have a particular month date:

CQL

```
SELECT * FROM cycling.upcoming_calendar
    WHERE events CONTAINS 'Criterium du Dauphine'
          OR month = 7;
```

Result

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

Using the map type

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_teams (
  id uuid PRIMARY KEY,
  firstname text,
  lastname text,
  teams map<int, text>
);
CREATE INDEX IF NOT EXISTS team_year_keys_idx
ON cycling.cyclist_teams ( KEYS (teams) );
CREATE INDEX IF NOT EXISTS team_year_entries_idx
```

```
ON cycling.cyclist_teams ( ENTRIES (teams) );
CREATE INDEX IF NOT EXISTS team_year_values_idx
ON cycling.cyclist_teams ( VALUES (teams) );
```

Indexes created on the map column `teams` in the `cyclist_career_teams` table target the keys, values, and full entries of the column data.

Query with `KEYS` from the map `teams` column:

CQL

```
SELECT * FROM cyclist_teams WHERE teams CONTAINS KEY 2014;
```

Result

```
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 |
{2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
```

Query a value from the map `teams` column, noting that only the keyword `CONTAINS` is included:

CQL

```
SELECT * FROM cyclist_teams WHERE teams CONTAINS 'Team Garmin - Cervelo';
```

Result

```
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'}
```

Query entries from the map `teams` column, noting the difference in the `WHERE` clause:

CQL

```
SELECT * FROM cyclist_teams
WHERE
    teams[2014] = 'Boels:Dolmans Cycling Team'
    AND teams[2015] = 'Boels:Dolmans Cycling Team';
```

Result

```
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'}
```

This example looks for a row where two entries are present in the map `teams` column.

For more information, see:

- **CREATE CUSTOM INDEX**
- **Creating collections**
- **Using set type**
- **Using list type**
- **Using map type**

Querying with SAI

Querying with SAI

The **SAI quickstart** focuses only on defining multiple indexes based on non-primary key columns (a very useful feature). Let's explore other options using some examples of how you can run queries on tables that have differently defined SAI indexes.

IMPORTANT | SAI only supports `SELECT` queries, but not `UPDATE` or `DELETE` queries.

Vector search

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX IF NOT EXISTS ann_index
  ON cycling.comments_vs(comment_vector) USING 'sai';
```

Query vector data with CQL

To query data using Vector Search, use a `SELECT` query:

CQL

```
SELECT * FROM cycling.comments_vs
    ORDER BY comment_vector ANN OF [0.15, 0.1, 0.1, 0.35, 0.55]
    LIMIT 3;
```

Result

```
id                                   | created_at                      | comment
```

```
| comment_vector               | commenter | record_id
--------------------------------------+--------------------------------+
---------------------------------------+------------------------------+
-----------+--------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-04-01 14:33:02.160000+0000 | LATE
RIDERS SHOULD NOT DELAY THE START | [0.9, 0.54, 0.12, 0.1, 0.95] |      Alex |
616e77e0-22a2-11ee-b99d-1f350647414a
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-02-17 08:43:20.234000+0000 |
Glad you ran the race in the rain | [0.3, 0.34, 0.2, 0.78, 0.25] |       Amy |
6170c1d0-22a2-11ee-b99d-1f350647414a
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-04-01 13:43:08.030000+0000 |
Last climb was a killer |   [0.3, 0.75, 0.2, 0.2, 0.5] |       Amy | 62105d30-
22a2-11ee-b99d-1f350647414a
```

NOTE The limit has to be 1,000 or fewer.

Scrolling to the right on the results shows the comments from the table that most closely matched the embeddings used for the query.

Single index match on a column

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

The column `commenter` is not the partition key in this table, so an index is required to query on it.

Query for a match on that column:

Query

```
SELECT * FROM cycling.comments_vs
        WHERE commenter = 'Alex';
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-----------------------------------------+-----------------------------+
-----------+-----------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-04-01 14:33:02.160000+0000 | LATE
RIDERS SHOULD NOT DELAY THE START | [0.9, 0.54, 0.12, 0.1, 0.95] |      Alex |
6d0cdaa0-272b-11ee-859f-b9098002fcac
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 |
Second rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex |
6d0b7b10-272b-11ee-859f-b9098002fcac
```

Failure with index

Note that a failure will occur if you try this query before creating the index:

Query

```
SELECT * FROM cycling.comments_vs
        WHERE commenter = 'Alex';
```

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"
```

Single index match on a column with options

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_cs_idx ON cycling.comments_vs (commenter)
USING 'sai'
WITH OPTIONS = {'case_sensitive': 'true', 'normalize': 'true', 'ascii': 'true'};
```

Case-sensitivty

The column `commenter` is not the partition key in this table, so an index is required to query on it. If we want to check `commenter` as a case-sensitive value, we can use the `case_sensitive` option set to `true`.

Note that no results are returned if you use an inappropriately case-sensitive value in the query:

Query

```
SELECT * FROM comments_vs WHERE commenter ='alex';
```

Result

```
id | created_at | comment | comment_vector | commenter | record_id
----+------------+---------+----------------+-----------+-----------

(0 rows)
```

When we switch the case of the cyclist's name to match the case in the index, the query succeeds:

Query

```
SELECT comment,commenter FROM comments_vs WHERE commenter ='Alex';
```

Result

```
comment                                | commenter
---------------------------------------+-----------
 LATE RIDERS SHOULD NOT DELAY THE START |      Alex
      Second rest stop was out of water |      Alex
(2 rows)
```

Index match on a composite partition key column

This example uses the following table and indexes:

```
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 race_name_idx
    ON cycling.rank_by_year_and_name (race_name)
    USING 'sai';
CREATE INDEX race_year_idx
    ON cycling.rank_by_year_and_name (race_year)
    USING 'sai';
```

Composite partition keys have a partition defined by multiple columns in a table. Normally, you would need to specify all the columns in the partition key to query the table with a `WHERE` clause. However, an SAI index makes it possible to define an index using a single column in the table's composite partition key. You can create an SAI index on each column in the composite partition key, if you need to query based on just one column.

SAI indexes also allow you to query tables without using the inefficient `ALLOW FILTERING` directive. The `ALLOW FILTERING` directive requires scanning all the partitions in a table, leading to poor performance.

The `race_year` and `race_name` columns comprise the composite partition key for the `cycling.rank_by_year_and_name` table.

Query for a match on the column `race_name`:

Query

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

Result

```
race_year | race_name                                  | rank | cyclist_name
-----------+--------------------------------------------+------+
----------------------
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |        Daniel
MARTIN
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 | Johan Esteban
CHAVES
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |      Benjamin
PRADES
      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
```

Query for a match on the column `race_year`:

Query

```
SELECT * FROM cycling.rank_by_year_and_name
        WHERE race_year = 2014;
```

Result

```
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
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    1 |        Daniel
MARTIN
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    2 | Johan Esteban
CHAVES
      2014 | Tour of Japan - Stage 4 - Minami > Shinshu |    3 |      Benjamin
PRADES
```

Multiple indexes matched with AND

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

Several indexes are created for the table to demonstrate how to query for matches on more than one column.

Query for matches on more than one column, and both columns must match:

Query

```
SELECT * FROM cycling.comments_vs
    WHERE
    created_at='2017-03-21 21:11:09.999000+0000'
```

```
    AND commenter = 'Alex';
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-------------------------------------+-------------------------------+-----------+
---------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 | Second
rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex | 6d0b7b10-
272b-11ee-859f-b9098002fcac
```

Multiple indexes matched with OR

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
    USING 'sai';
```

Several indexes are created for the table to demonstrate how to query for matches on more than one column.

Query for a match on either one column or the other:

Query

```
SELECT * FROM cycling.comments_vs
    WHERE
    created_at='2017-03-21 21:11:09.999000+0000'
    OR created_at='2017-03-22 01:16:59.001000+0000';
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
----------------------------------+-------------------------------+-----------+
-------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 | Second
rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex | 6d0b7b10-
272b-11ee-859f-b9098002fcac
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-03-22 01:16:59.001000+0000 |
Great snacks at all reststops |  [0.1, 0.4, 0.1, 0.52, 0.09] |       Amy |
6d0fc0d0-272b-11ee-859f-b9098002fcac
```

Multiple indexes matched with IN

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.comments_vs (
  record_id timeuuid,
  id uuid,
  commenter text,
  comment text,
  comment_vector VECTOR <FLOAT, 5>,
  created_at timestamp,
  PRIMARY KEY (id, created_at)
)
WITH CLUSTERING ORDER BY (created_at DESC);
CREATE INDEX commenter_idx
    ON cycling.comments_vs (commenter)
    USING 'sai';
CREATE INDEX created_at_idx
    ON cycling.comments_vs (created_at)
    USING 'sai';
CREATE INDEX ann_index
    ON cycling.comments_vs (comment_vector)
```

```
    USING 'sai';
```

Several indexes are created for the table to demonstrate how to query for matches on more than one column.

Query for match with column values in a list of values:

Query

```
SELECT * FROM cycling.comments_vs
    WHERE created_at IN
    ('2017-03-21 21:11:09.999000+0000'
    ,'2017-03-22 01:16:59.001000+0000');
```

Result

```
id                                   | created_at                      | comment
| comment_vector               | commenter | record_id
--------------------------------------+---------------------------------+
-----------------------------------+-----------------------------+-----------+
--------------------------------------
 e7ae5cf3-d358-4d99-b900-85902fda9bb0 | 2017-03-21 21:11:09.999000+0000 | Second
rest stop was out of water | [0.99, 0.5, 0.99, 0.1, 0.34] |      Alex | 6d0b7b10-
272b-11ee-859f-b9098002fcac
 c7fceba0-c141-4207-9494-a29f9809de6f | 2017-03-22 01:16:59.001000+0000 |
Great snacks at all reststops |  [0.1, 0.4, 0.1, 0.52, 0.09] |       Amy |
6d0fc0d0-272b-11ee-859f-b9098002fcac
```

User-defined type

SAI can index either a user-defined type (UDT) or a list of UDTs. This example shows how to index a list of UDTs.

This example uses the following user-defined type (UDT), table and index:

```
CREATE TYPE IF NOT EXISTS cycling.race (
  race_title text,
  race_date timestamp,
  race_time text
);
CREATE TABLE IF NOT EXISTS cycling.cyclist_races (
  id UUID PRIMARY KEY,
```

```
  lastname text,
  firstname text,
  races list<FROZEN <race>>
);
CREATE INDEX races_idx
    ON cycling.cyclist_races (races)
    USING 'sai';
```

An index is created on the list of UDTs column `races` in the `cycling.cyclist_races` table.

Query with `CONTAINS` from the list `races` column:

CQL

```
SELECT * FROM cycling.cyclist_races
    WHERE races CONTAINS {
       race_title:'Rabobank 7-Dorpenomloop Aalburg',
       race_date:'2015-05-09',
       race_time:'02:58:33'};
```

Result

```
id                                   | firstname | lastname | races
--------------------------------------+-----------+----------+
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------------
 5b6962dd-3f90-4c93-8f61-eabfa4a803e2 |  Marianne |      VOS | [{race_title:
'Rabobank 7-Dorpenomloop Aalburg', race_date: '2015-05-09 00:00:00.000000+0000',
race_time: '02:58:33'}, {race_title: 'Ronde van Gelderland', race_date: '2015-04-
19 00:00:00.000000+0000', race_time: '03:22:23'}]

(1 rows)
```

SAI indexing with collections

SAI supports collections of type `map`, `list`, and `set`. Collections allow you to group and store data together in a column.

In a relational database, a grouping such as a user's multiple email addresses is achieved via many-to-one joined relationship between (for example) a `user` table and an `email` table. **Apache 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.

TIP

In CQL queries of database tables with 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

Using the set type

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (
  id UUID PRIMARY KEY,
  lastname text,
  teams set<text>
);
CREATE INDEX teams_idx
    ON cycling.cyclist_career_teams (teams)
    USING 'sai';
```

An index is created on the set column `teams` in the `cyclist_career_teams` table.

Query with `CONTAINS` from the set `teams` column:

CQL

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

Result

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

Using the list type

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.upcoming_calendar (
  year int,
  month int,
  events list<text>,
  PRIMARY KEY (year, month)
);
CREATE INDEX events_idx
    ON cycling.upcoming_calendar (events)
    USING 'sai';
```

An index is created on the list column `events` in the `upcoming_calendar` table.

Query with `CONTAINS` from the list `events` column:

CQL

```
SELECT * FROM cycling.upcoming_calendar
   WHERE events CONTAINS 'Criterium du Dauphine';
```

Result

```
year | month | events
------+-------+-----------------------------------------------
 2015 |     6 | ['Criterium du Dauphine', 'Tour de Sui\nsse']
```

A slightly more complex query selects rows that either contain a particular event or have a particular month date:

CQL

```
SELECT * FROM cycling.upcoming_calendar
    WHERE events CONTAINS 'Criterium du Dauphine'
          OR month = 7;
```

Result

```
year | month | events
```

```
------+-------+-------------------------------------------------
 2015 |     6 | ['Criterium du Dauphine', 'Tour de Sui\nsse']
 2015 |     7 |                            ['Tour de France']
```

Using the map type

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_teams (
  id uuid PRIMARY KEY,
  firstname text,
  lastname text,
  teams map<int, text>
);
CREATE INDEX IF NOT EXISTS team_year_keys_idx
ON cycling.cyclist_teams ( KEYS (teams) );
CREATE INDEX IF NOT EXISTS team_year_entries_idx
ON cycling.cyclist_teams ( ENTRIES (teams) );
CREATE INDEX IF NOT EXISTS team_year_values_idx
ON cycling.cyclist_teams ( VALUES (teams) );
```

Indexes created on the map column `teams` in the `cyclist_career_teams` table target the keys, values, and full entries of the column data.

Query with `KEYS` from the map `teams` column:

CQL

```
SELECT * FROM cyclist_teams WHERE teams CONTAINS KEY 2014;
```

Result

```
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 |
```

```
{2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
```

Query a value from the map `teams` column, noting that only the keyword `CONTAINS` is included:

CQL

```
SELECT * FROM cyclist_teams WHERE teams CONTAINS 'Team Garmin - Cervelo';
```

Result

```
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'}
```

Query entries from the map `teams` column, noting the difference in the `WHERE` clause:

CQL

```
SELECT * FROM cyclist_teams
WHERE
    teams[2014] = 'Boels:Dolmans Cycling Team'
    AND teams[2015] = 'Boels:Dolmans Cycling Team';
```

Result

```
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'}
```

This example looks for a row where two entries are present in the map `teams` column.

For more information, see:

- **CREATE CUSTOM INDEX**
- **Creating collections**
- **Using set type**
- **Using list type**
- **Using map type**

SAI indexing with collections

SAI indexing with collections

SAI supports collections of type `map`, `list`, and `set`. Collections allow you to group and store data together in a column.

In a relational database, a grouping such as a user's multiple email addresses is achieved via many-to-one joined relationship between (for example) a `user` table and an `email` table. **Apache 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.

TIP

In CQL queries of database tables with 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

Using the set type

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_career_teams (
  id UUID PRIMARY KEY,
  lastname text,
  teams set<text>
);
CREATE INDEX teams_idx
    ON cycling.cyclist_career_teams (teams)
    USING 'sai';
```

An index is created on the set column `teams` in the `cyclist_career_teams` table.

Query with `CONTAINS` from the set `teams` column:

CQL

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

Result

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

Using the list type

This example uses the following table and index:

```
CREATE TABLE IF NOT EXISTS cycling.upcoming_calendar (
  year int,
  month int,
  events list<text>,
  PRIMARY KEY (year, month)
);
CREATE INDEX events_idx
    ON cycling.upcoming_calendar (events)
    USING 'sai';
```

An index is created on the list column `events` in the `upcoming_calendar` table.

Query with `CONTAINS` from the list `events` column:

CQL

```
SELECT * FROM cycling.upcoming_calendar
   WHERE events CONTAINS 'Criterium du Dauphine';
```

Result

```
year | month | events
------+-------+-----------------------------------------------
 2015 |     6 | ['Criterium du Dauphine', 'Tour de Sui\nsse']
```

A slightly more complex query selects rows that either contain a particular event or have a particular month date:

CQL

```
SELECT * FROM cycling.upcoming_calendar
    WHERE events CONTAINS 'Criterium du Dauphine'
          OR month = 7;
```

Result

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

Using the map type

This example uses the following table and indexes:

```
CREATE TABLE IF NOT EXISTS cycling.cyclist_teams (
  id uuid PRIMARY KEY,
  firstname text,
  lastname text,
  teams map<int, text>
);
CREATE INDEX IF NOT EXISTS team_year_keys_idx
ON cycling.cyclist_teams ( KEYS (teams) );
CREATE INDEX IF NOT EXISTS team_year_entries_idx
ON cycling.cyclist_teams ( ENTRIES (teams) );
CREATE INDEX IF NOT EXISTS team_year_values_idx
ON cycling.cyclist_teams ( VALUES (teams) );
```

Indexes created on the map column `teams` in the `cyclist_career_teams` table target the keys, values, and full entries of the column data.

Query with `KEYS` from the map `teams` column:

CQL

```
SELECT * FROM cyclist_teams WHERE teams CONTAINS KEY 2014;
```

Result

```
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 |
{2014: 'Rabobank-Liv Woman Cycling Team', 2015: 'Rabobank-Liv Woman Cycling Team'}
```

Query a value from the map `teams` column, noting that only the keyword `CONTAINS` is included:

CQL

```
SELECT * FROM cyclist_teams WHERE teams CONTAINS 'Team Garmin - Cervelo';
```

Result

```
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'}
```

Query entries from the map `teams` column, noting the difference in the `WHERE` clause:

CQL

```
SELECT * FROM cyclist_teams
WHERE
    teams[2014] = 'Boels:Dolmans Cycling Team'
    AND teams[2015] = 'Boels:Dolmans Cycling Team';
```

Result

```
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'}
```

This example looks for a row where two entries are present in the map `teams` column.

For more information, see:

- **CREATE CUSTOM INDEX**
- **Creating collections**
- **Using set type**
- **Using list type**
- **Using map type**

SAI write path and read path

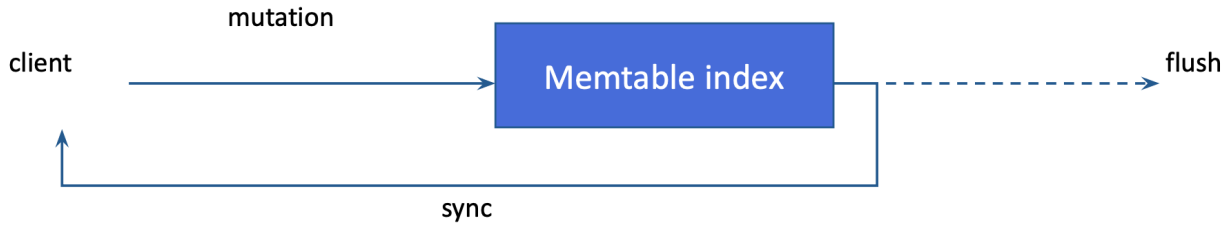
SAI write path and read path

SAI is deeply integrated with the storage engine of the underlying database. SAI does not abstractly index tables. Instead, SAI indexes **Memtables** and Sorted String Tables (**SSTables**) as they are written, resolving the differences between those indexes at read time. Each Memtable is an in-memory data structure that is specific to a particular database table. A Memtable resembles a write-back cache. Each SSTable is an immutable data file to which the database writes Memtables periodically. SSTables are stored on disk sequentially and maintained for each database table.

This topic discusses the details of the SAI read and write paths, examining the SAI indexing lifecycle.

SAI write path

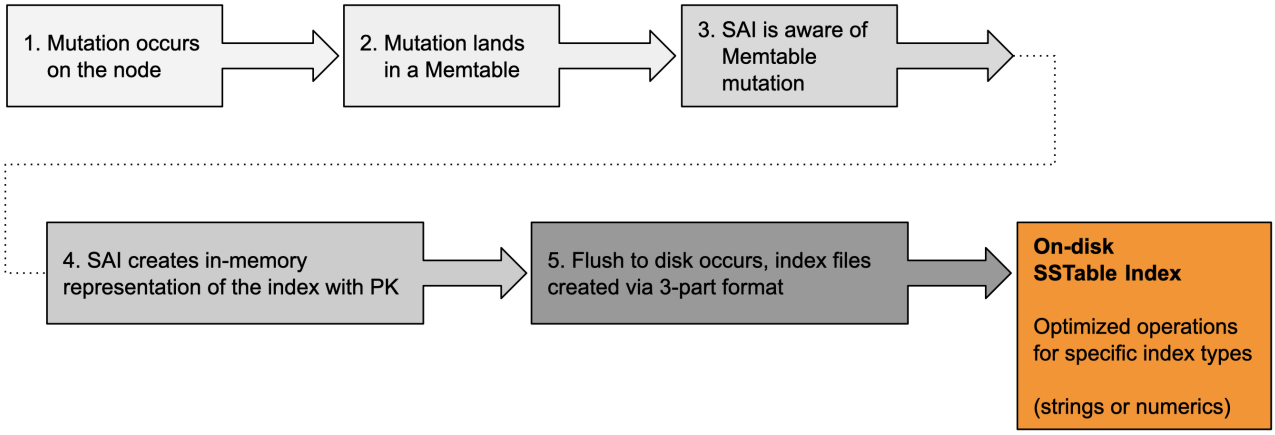
An SAI index can be created either before any data is written to a CQL table, or after data has been written. As a refresher, data written to a CQL table will first be written to a Memtable, and then to an SSTable once the data is flushed from the Memtable. After an SAI index is created, SAI is notified of all mutations against the current Memtable. Like any other data, SAI updates the indexes for inserts and updates, which **Apache Cassandra** treats in exactly the same way. SAI also supports partition deletions, range tombstones, and row removals. If a delete operation is executed in a query, SAI handles index changes in a post-filtering step. As a result, SAI imposes no special penalties when indexing frequently deleted columns.



If an insert or update contains valid content for the indexed column, the content is added to a **Memtable index**, and the primary key of the updated row is associated with the indexed value. SAI calculates an estimate of the incremental heap consumption of the new entry. This estimate counts against the heap usage of the underlying Memtable. This feature also means that as more columns are indexed on a table, the Memtable flush rate will increase, and the size of flushed SSTables will decrease. The number of total writes and the estimated heap usage of all live Memtable indexes are exposed as metrics. See **SAI metrics**.

Memtable flush

SAI flushes Memtable index contents directly to disk when the flush threshold is reached, rather than creating an additional in-memory representation. This is possible because Memtable indexes are sorted by term/value and then by primary key. When the flush occurs, SAI writes a new SSTable index for each indexed column, as the SSTable is being written.



The flush operation is a two-phase process. In the first phase, rows are written to the new SSTable. For each row, a row ID is generated and three index components are created. The components are:

- On-disk mappings of row IDs to their corresponding token values — SAI supports the Murmur3Partitioner
- SSTable partition offsets
- A temporary mapping of primary keys to their row IDs, which is used in a subsequent phase

The contents of the first and second component files are compressed arrays whose ordinals correspond to row IDs.

In the second phase, after all rows are written to the new SSTable and the shared SSTable-level index components have been completed, SAI begins its indexing work on each indexed column. Specifically in the second phase, SAI iterates over the Memtable index to produce pairs of terms and their token-sorted row IDs. This iterator translates primary keys to row IDs using the temporary mapping structure built in the first phase. The terms iterator (with postings) is then passed to separate writing components based on whether each indexed element is for a string or numeric column.

In the string case, the SAI index writer iterates over each term, first writing its postings to disk, and then recording the offset of those postings in the postings file as the payload of a new entry (for the term itself) in an on-disk, byte-ordered trie. In the numeric case, SAI separates the writing of a numeric index into two steps:

The terms are passed to a balanced kd-tree writer, which writes the kd-tree to disk. As the leaf blocks of the tree are written, their postings are recorded temporarily in memory. Those temporary postings are then used to build postings on-disk, at the leaves, and at various levels of the index nodes.

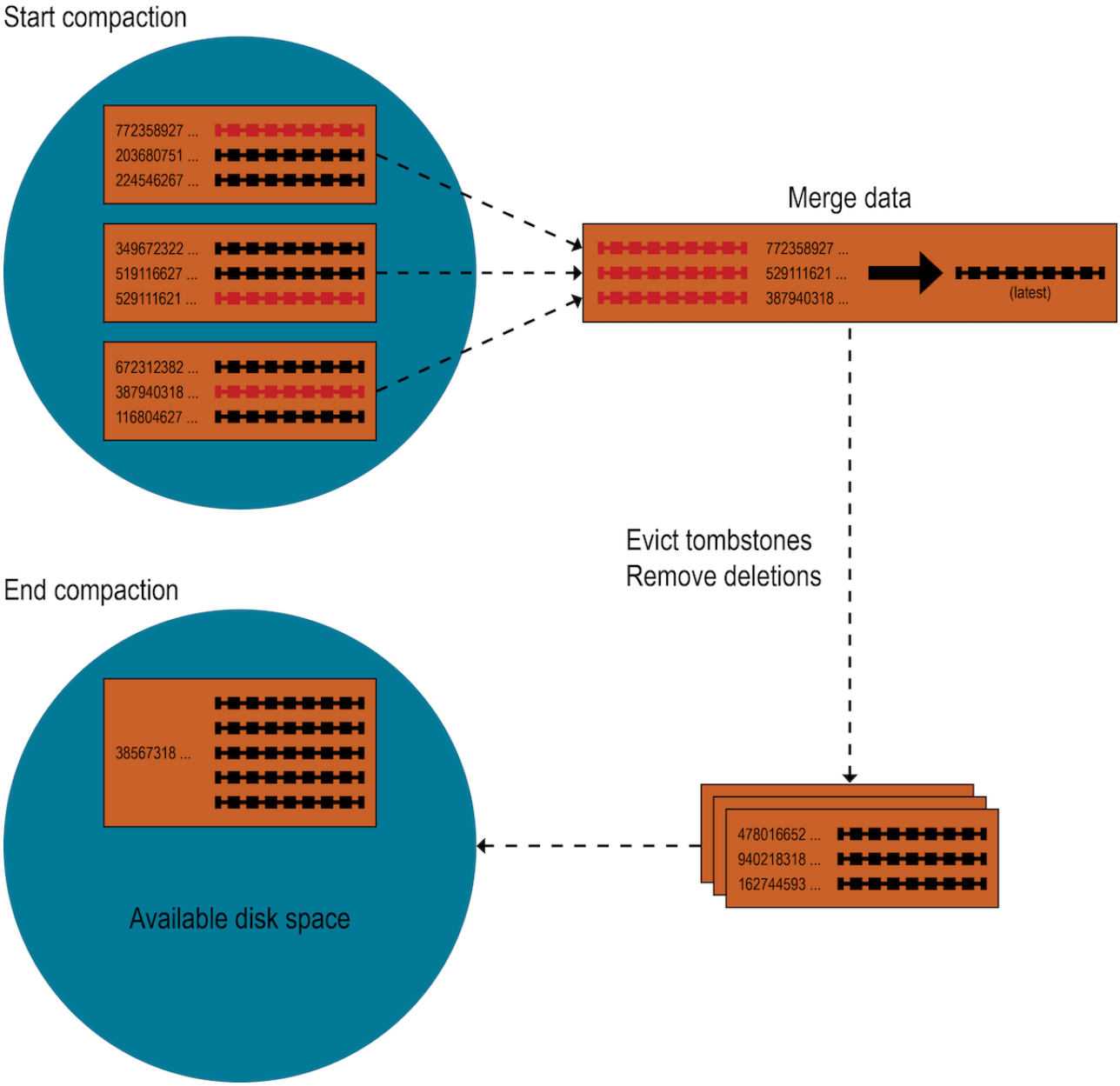
When a column index flush completes, a special empty marker file is flagged to indicate success. This procedure is used on startup and incremental rebuild operations to differentiate case where:

- SSTable index files are missing for a column.
- There is no indexable data — such as when an SSTable only contains tombstones. (A tombstone is a marker in a row that indicates a column was deleted. During compaction, marked columns are deleted.)

SAI then increments a counter on the number of Memtable indexes flushed for the column, and adds to a histogram the number of cells flushed per second.

When compaction is triggered

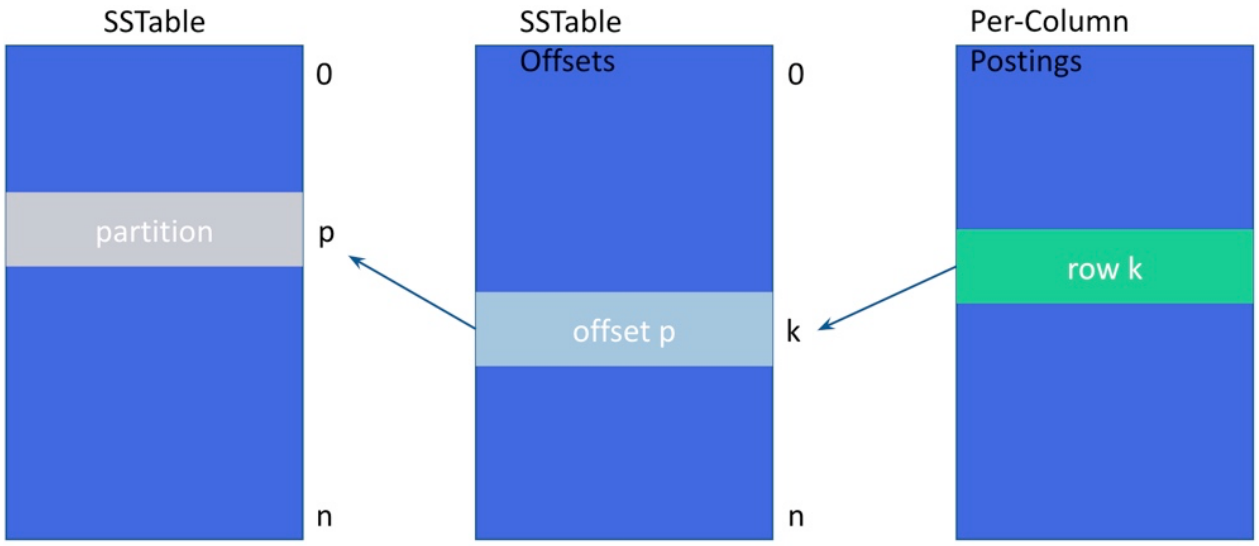
Recall that **Apache Cassandra** uses compaction to merge SSTables. Compaction collects all versions of each unique row and assembles one complete row, using the most up-to-date version (by timestamp) of each of the row's columns from the SSTables. The merge process is performant, because rows are sorted by partition key within each SSTable, and the merge process does not use random I/O. The new versions of each row is written to a new SSTable. The old versions, along with any rows that are ready for deletion, are left in the old SSTables, and are deleted when any pending reads are completed.



For SAI, when compaction is triggered, each index group creates an SSTable Flush Observer to coordinate the process of writing all attached column indexes from the newly compacted data in their respective SSTable writer instances. Unlike Memtable flush (where indexed terms are already sorted), when iterating merged data during compaction, SAI buffers indexed values and their row ids, which are added in token order.

To avoid issues such as exhausting available heap resources, SAI uses an accumulated segment buffer, which is flushed to disk synchronously by using a proprietary calculation. Then each segment records a segment row ID **offset** and only stores the segment row ID (SSTable row ID minus segment row ID offset). SAI flushes segments into the same file synchronously to avoid the cost of rewriting all segments and to reduce the cost of partition-restricted queries and paging range queries, as it reduces the search space.

The on-disk layout from the per-column indexed posting, to the SSTable offset, to the SSTable partition:



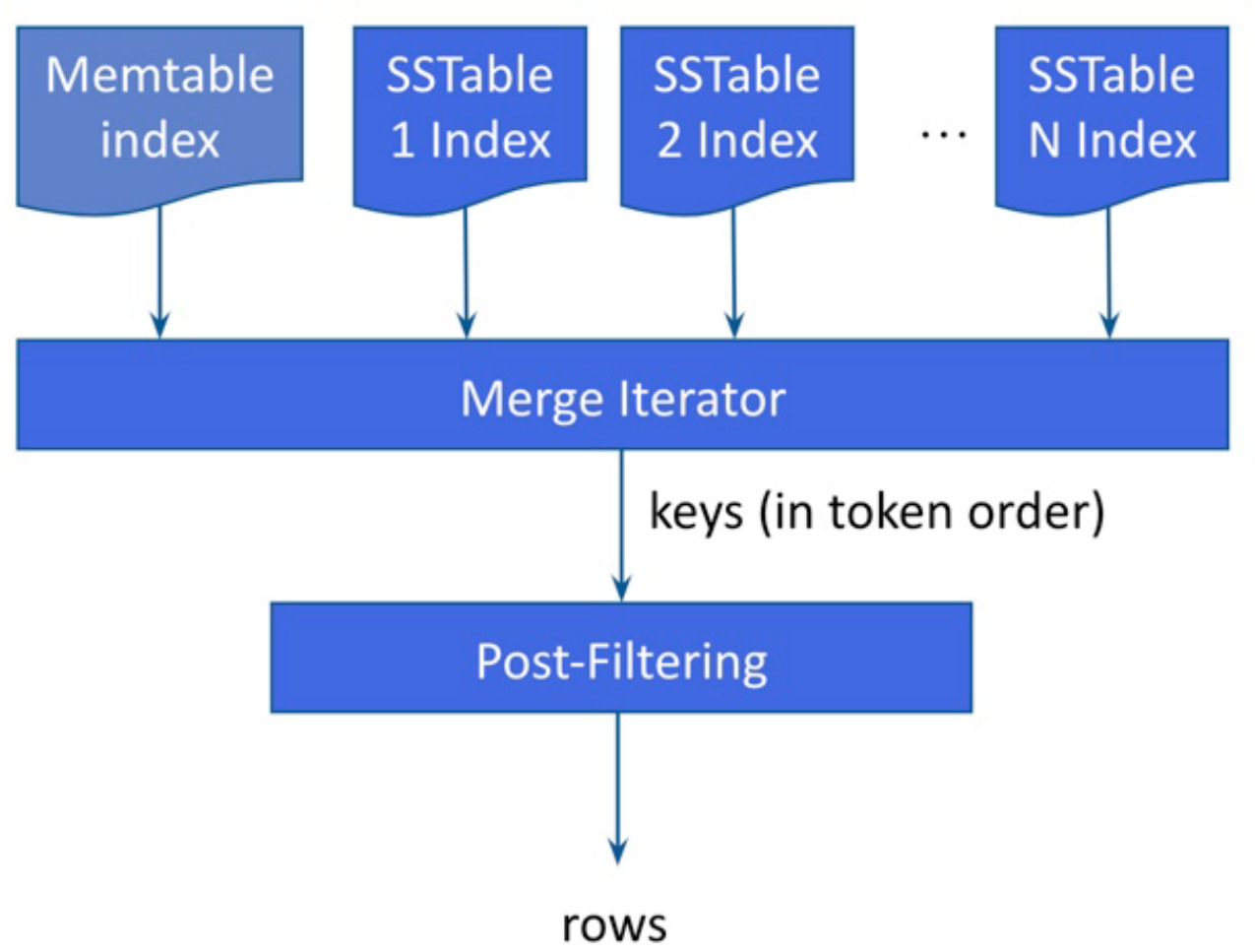
The actual segment flushing process is very similar to a Memtable flush. However, buffered terms are sorted before they can be written with their postings to their respective type-specific on-disk structures. At the end of compaction for a given index, a special empty marker file is flagged to indicate success, and the number of segments is recorded in SAI metrics. See **Global indexing metrics**.

When the entire compaction task finishes, SAI receives an SSTable List Changed Notification that contains the SSTables added and removed during the transaction. SSTable Context Manager and Index View Manager are responsible for replacing old SSTable indexes with new ones atomically. At this point, new SSTable indexes are available for queries.

SAI read path

This section explains how index queries are processed by the SAI coordinator and executed by replicas. Unlike legacy secondary indexes, where at most one column index will be used per query, SAI implements a `Query Plan` that makes it possible to use all available column indexes in a single query.

The overall flow of a SAI read operation is as follows:

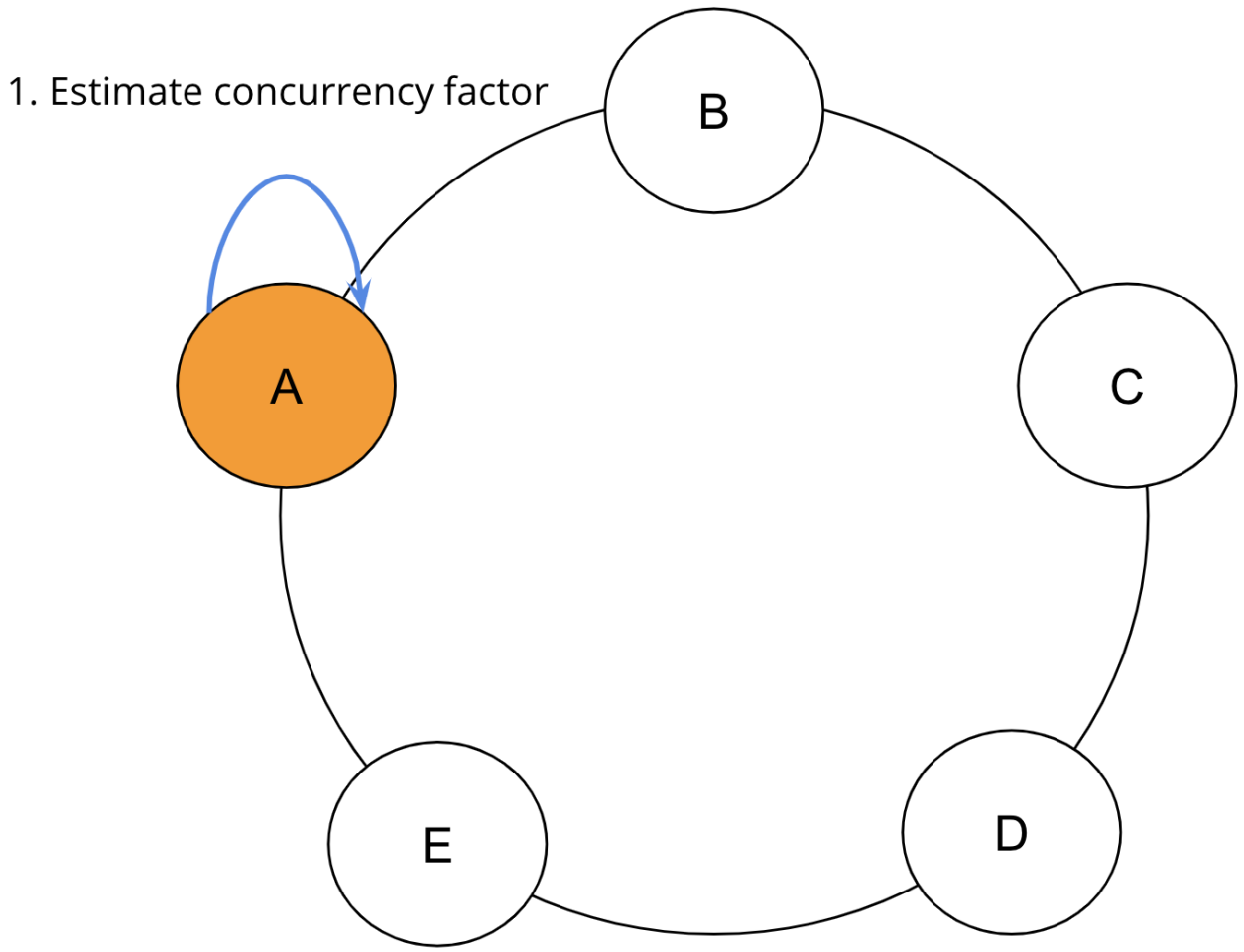


Index selection and Coordinator processing

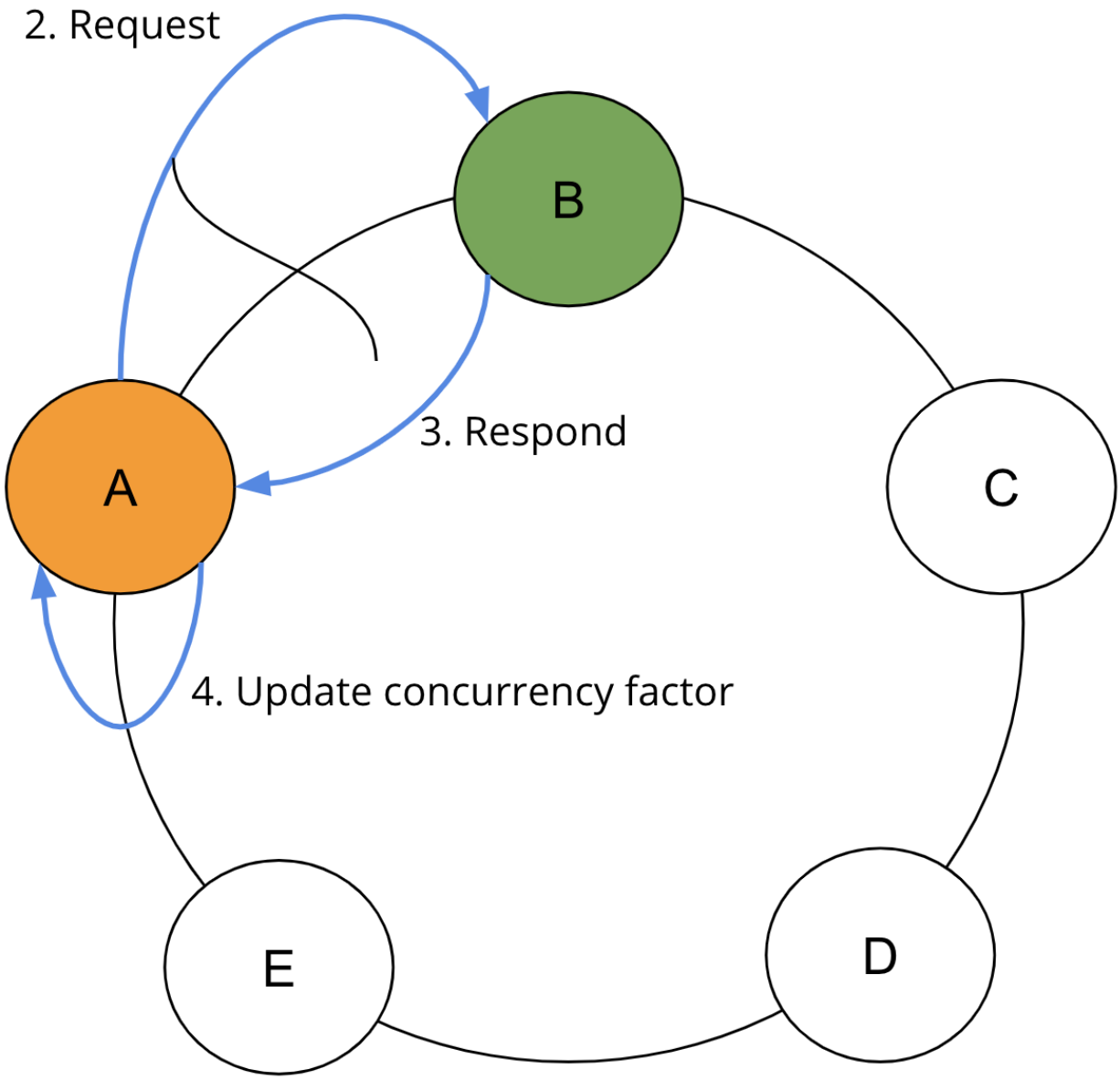
When presented with a query, the first action the SAI Coordinator performs, to take advantage of one or more indexes, is to identify the most selective index. That most selective index is the index that will most aggressively narrow the filtering space and the number of ultimate results by returning the lowest value from an estimated results row calculation. If multiple SAI indexes are present (that is, where each SAI index is based on a different column, but the query involves more than one column), it does not matter which SAI index is selected first.

Once the best index for the read operation is selected, the index is embedded with the read command, which enters the distributed range read apparatus. A distributed range read queries the **Apache Cassandra** cluster in one or more rounds in token order. The SAI Coordinator estimates the **concurrency factor**, the number of rows per range based on local data and the query limit to determine the number of ranges to contact. For each round, a concurrency factor determines how many replicas will be queried in parallel.

Before the first round commences, SAI estimates the initial concurrency factor via a proprietary calculation, shown here as Step 1.

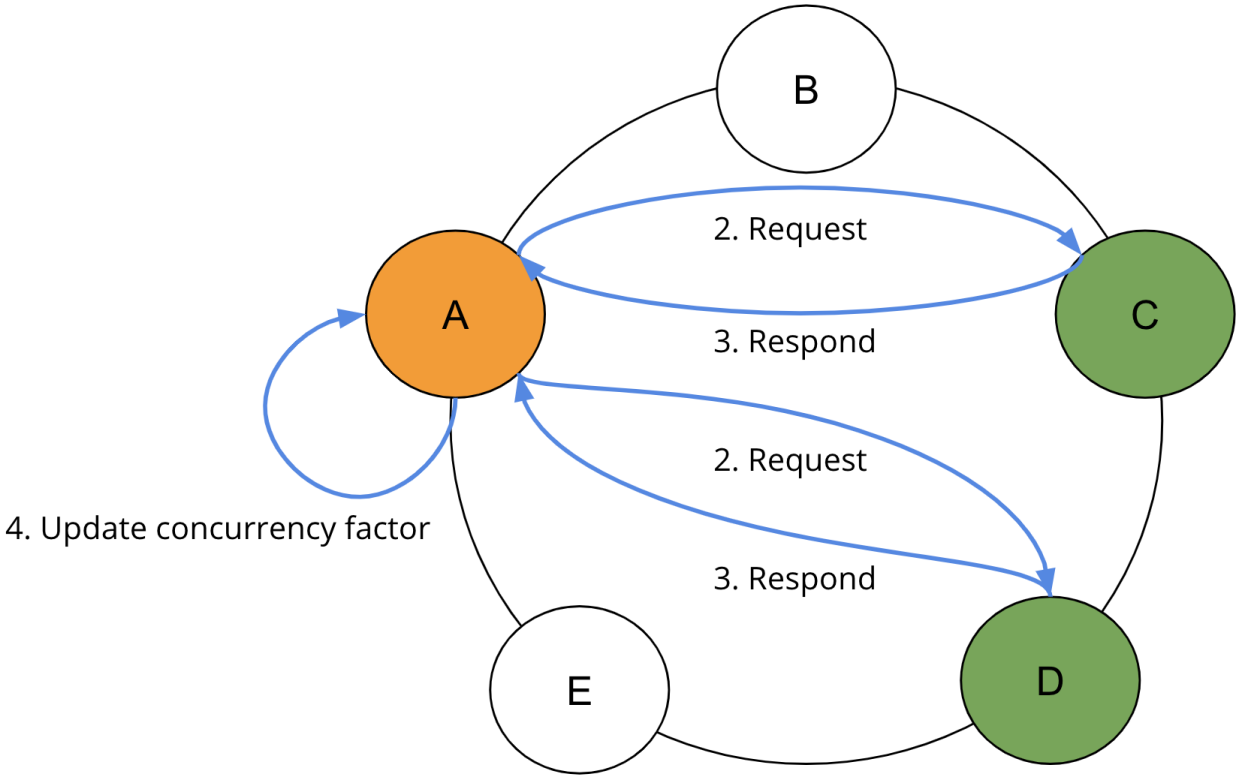


Once the initial concurrency factor established, the range read begins.



In Step 2, the SAI Coordinator sends requests to the required ranges in parallel based on the Concurrency factor. In Step 3, the SAI Coordinator waits for the responses from the requested replicas. And in Step 4, SAI Coordinator collects the results and recomputes the concurrency factor based on returned rows and query limit.

At the completion of each round, if the limit has not been reached, the concurrency factor is adjusted to take into account the shape of the results already read. If no results are returned from the first round, the concurrency factor is immediately increased to the minimum calculation of remaining token ranges and the maximum calculation of the concurrency factor. If results are returned, the concurrency factor is updated. SAI repeats steps 2, 3, and 4 until the query limit is satisfied.



To avoid querying replicas with failed indexes, each node propagates its own local index status to peers via gossip. At the coordinator, read requests will filter replicas that contain non-queryable indexes used in the request. In most cases, the second round of replica queries should return all the necessary results. Further rounds may be necessary if the distribution of results around the replicas is extremely imbalanced.

A closer look: replica query planning and view acquisition

Once a replica receives a token range read request from the SAI Coordinator, the local index query begins. SAI implements an index searcher interface via a Query Plan that makes it possible to access all available SAI column indexes in a single query.

The Query Plan performs an analysis of the expressions passed to it via the read command. SAI determines which indexes should be used to satisfy the query clauses on the given columns. Once column expressions are paired with indexes, a view of the active SSTable indexes for each column index is acquired by a Query Controller. In order to avoid compaction removing index files used by in-flight queries, before reading any index files, the Query Controller tries to acquire references to the SSTables for index files that intersect with the query's token range, and releases them when the read request completes.

At this point, a Token Flow is created to stream matches from each column index. Those flows, along with the Boolean logic that determines how they are merged, is wrapped up in an Operation, which is returned to the Query Plan component.

Role of the SAI Token Flow framework

The SAI query engine revolves around a Token Flow framework that defines how SAI asynchronously iterates over, skips into, and merges streams of matching partitions from both individual SSTable indexes and entire column indexes. SAI uses a token to describe a container for partition matches within a Cassandra ring token.

Iteration is the simplest of the three operations. Specifically, the iteration of postings involves sequential disk access — via the chunk cache — to row IDs, which are used to lookup ring token and partition key offset information.

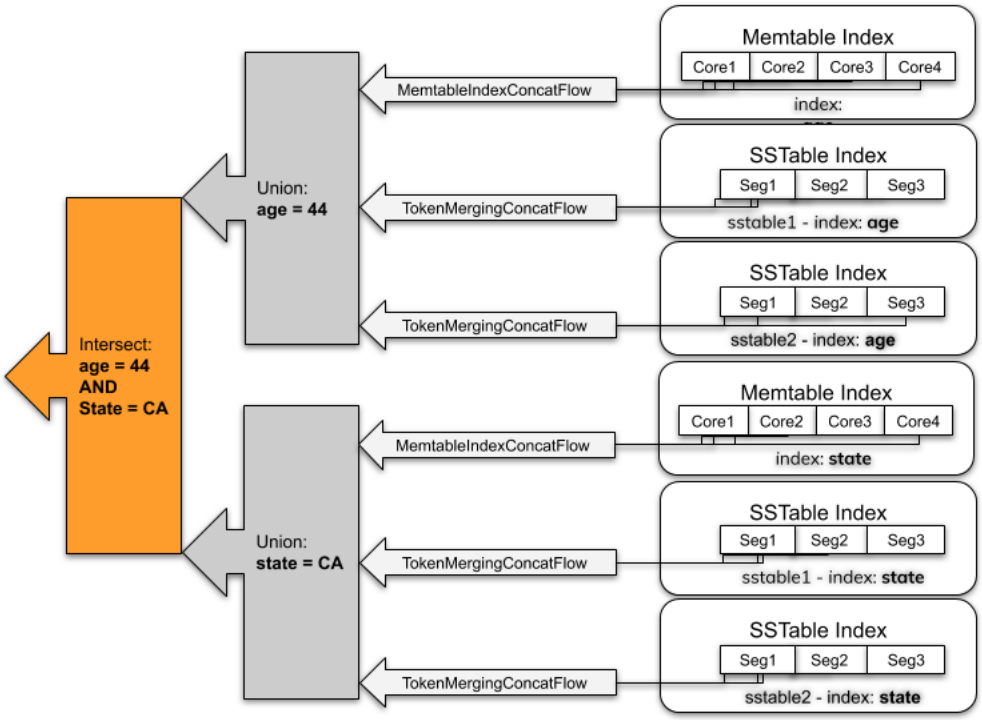
Token skipping is used to skip unmatched tokens when continuing from the previous paging boundary, or when a larger token is found during token intersection.

Match streaming and post filtering example

Consider an example with an individual column index (such as `age = 44`), the flow produced is the union of all Memtable indexes and all SSTable indexes.

- SAI iterates over each Memtable index "lazily" (that is, one at a time) in token order through its individual token range-partitioned instances. This feature reduces the overhead that would occur otherwise from unnecessary searches of data toward the end of the ring.
- On-disk index: SAI returns the union of all matching SSTable index. Within one SSTable index, there can be multiple segments because of the memory limit during compaction. Similar to the Memtable index, SAI lazily searches segments in token sorted order.

When there are multiple indexed expressions in the query (such as `WHERE age=44 AND state='CA'`) connected with `AND` query operator, the results of indexed expressions are intersected, which returns partition keys that match all indexed expressions.



After the index search, SAI exposes a flow of partition keys. For every single partition key, SAI executes a single partition read operation, which returns the rows in the given partition. As rows are materialized, SAI uses a filter tree to apply another round of filtering. SAI performs this subsequent filtering step to address the following:

- Partition granularity: SAI keeps track of partition offsets. In the case of a wide partition schema, not all rows in the partition will match the index expressions.
- Tombstones: SAI does not index tombstones. It's possible that an indexed row has been shadowed by newly added tombstones.
- Non-indexed expressions: Operations may include non-index expressions for which there are no index structures.

What's next?

See the blog, Better Cassandra Indexes for a Better Data Model: Introducing Storage-Attached Indexing.

SAI operations

SAI operations

- **Configuring SAI operations**
- **Monitoring SAI operations**

Configure SAI indexes

Configure SAI indexes

Configuring your **Apache Cassandra** environment for Storage-Attached Indexing (SAI) may require some customization of the `cassandra.yaml` file.

Compaction strategies

Read queries perform better with compaction strategies that produce fewer SSTables.

For most environments that include SAI indexes, using the default `SizeTieredCompactionStrategy` (STCS) is recommended. 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. For details, see **Configuring compaction**.

For time series data, an alternative is the `TimeWindowCompactionStrategy` (TWCS). TWCS compacts SSTables using a series of time windows. While in a time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS. At the end of the time window, all of these SSTables are compacted into a single SSTable. Then the next time window starts and the process repeats. The duration of the time window is the only setting required. See **TimeWindowCompactionStrategy**. For more information about TWCS, see **Time Window Compaction Strategy**.

In general, do not use `LeveledCompactionStrategy` (LCS) unless your index queries restrict the token range, either directly or by providing a restriction on the partition key. However, if you decide to use LCS, use the following guidelines:

- The `160` MB default for the `CREATE TABLE` command's `sstable_size_in_mb` option, described in this **topic**, may result in suboptimal performance for index queries that do not restrict on token range or partition key.
- While even higher values may be appropriate, depending on your hardware, we recommend at least doubling the default value of `sstable_size_in_mb`.

Example:

```
CREATE TABLE IF NOT EXISTS my_keyspace.my_table
.
.
.
   WITH compaction = {
     'class' : 'LeveledCompactionStrategy',
     'sstable_size_in_mb' : '320' };
```

After increasing the MB value, observe whether the query performance improves on tables with SAI indexes. To observe any performance deltas, per query, look at the `QueryLatency` and `SSTableIndexesHit`

data in the Cassandra query metrics.

Using a larger value reserves more disk space, because the SSTables are larger, and the ones destined for replacement will use more space while being compacted. However, the larger value results in having fewer SSTables, which lowers query latencies. Each SAI index should ultimately consume less space on disk because of better long-term compression with the larger indexes.

If query performance degrades on large (`sstable_max_size` ~2GB) SAI indexed SSTables when the workload is not dominated by reads but is experiencing increased write amplification, consider using Unified Compaction Strategy (UCS).

The `cassandra.yaml` options `sai_sstable_indexes_per_query_warn_threshold` (default: 32) and `sai_sstable_indexes_per_query_fail_threshold` (default: disabled) determine the number of SSTable indexes a SAI query touches before warning clients and failing queries respectively. When enabled, they can provide feedback for clients and protection for the database in the face of sub-optimal read queries.

About SAI encryption

With SAI indexes, its on-disk components are simply additional SSTable data. To protect sensitive user data, including any present in the table's partition key values, SAI will need to encrypt all parts of the index that contain user data,the trie index data for strings and the kd-tree data for numerics. By design, SAI does not encrypt non-user data such as postings metadata or SSTable-level offsets and tokens.

Monitor SAI indexes

Monitor SAI indexes

Both virtual tables and JMX-based metrics can be used to monitor the SAI indexes in your cluster.

SAI virtual tables

You can refer to data in the following **Apache Cassandra** virtual tables to determine the status of indexes created with SAI:

- `system_views.indexes` — provides information at the column index level, including the index name, number of indexed SSTables, disk usage, and index state. From the index state, the data reveals if the index is currently building, and whether the index can be queried.

TIP

When you `DROP` / recreate an SAI index, you are not blocked from entering queries that do not use the index. However, you cannot use that SAI index (based on the same column) until it has finished building and is queryable. To determine the current state of a given index, query the `system_views.indexes` virtual table. Example:

```
SELECT is_queryable,is_building FROM system_views.indexes WHERE
keyspace_name='keyspace'
       AND table_name='table' AND index_name='index';
```

- `system_views.sstable_indexes` — describes individual SSTable indexes, and includes information around disk size, min/max row ID, the min/max ring token, and the write-time version of the index.
- `system_views.sstable_index_segments` — describes the segments of the SSTable indexes. It exposes the segment row ID offset and most of the information in the SSTable-level virtual table, specifically at a segment granularity. For more details, refer to **Virtual tables for SAI indexes and SSTables**.

SAI tracing

SAI provides tracing capability just like other database components. Information is captured by the `system_traces` keyspace. You can enable tracing in CQLSH with `TRACING ON`, or in the Cassandra driver with `statement.enableTracing()`.

The number of rows filtered by a particular query will show up in the CQL query trace. Example:

```
Index query accessed memtable indexes, 2 SSTable indexes, and 2 segments, post-filtered
14 rows in 14 partitions, and took 88582 microseconds.
```

For details about tracing, refer to CQL **TRACING**.

SAI metrics

SAI provides a number of metrics to help you monitor the health of your indexes.

The categorized data:

- Global indexing metrics
- Table query metrics
- Per query metrics
- Column query metrics per index
- Range slice metrics

For example, you can use metrics to get the current count of total partition reads since the node started for `cycling.cyclist_semi_pro`. The keyspace and table were defined in **SAI quickstart**. This metric's `ObjectName`: `org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=cycling,table=cyclist_semi_pro,scope=TableQueryMetrics,name=TotalPartitionReads`.

The metrics are exposed via JMX, so any JMX-based tool can be used to monitor.

Global indexing metrics

```
ObjectName: org.apache.cassandra.metrics,type=StorageAttachedIndex,name=<metric>
```

The global indexing metrics for this node are:

- `ColumnIndexBuildsInProgress` — The number of individual on-disk column indexes currently being built.
- `SegmentBufferSpaceLimitBytes` — The limit on heap used to buffer SSTable index segments during compaction and index builds.

> **TIP** In cassandra.yaml, `segment_write_buffer_space_mb` limits the amount of heap used to build on-disk column indexes during compaction and initial builds. The default is 1024 MB.

For example, if there is only one column index building, SAI can buffer up to `segment_write_buffer_space_mb`. If there is one column index building per table across 8 compactors, each index will be eligible to flush once it reaches (`segment_write_buffer_space_mb / 8`) MBs.

- `SegmentBufferSpaceUsedBytes` — The heap currently being used to buffer SSTable index segments

during compaction and index builds.

NOTE	At any given time, the minimum size for a flushing segment, in bytes, is (`SegmentBufferSpaceLimitBytes / ColumnIndexBuildsInProgress`).

Index group metrics

```
ObjectName:
org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=<keyspace>,table=<table>,
scope=IndexGroupMetrics,name=<metric>
```

The index group metrics for the given keyspace and table:

- `DiskUsedBytes` — Size in bytes on disk for the given table's SAI indices.
- `IndexFileCacheBytes` — Size in bytes of memory used by the on-disk data structure of the per-column indices.
- `OpenIndexFiles` — Number of open index files for the given table's SAI indices.

Per query metrics

```
ObjectName:
org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=<keyspace>,table=<table>,
scope=PerQuery,name=<metric>
```

The per query metrics for the given keyspace and table include:

- `RowsFiltered` — A histogram of the number of rows post-filtered per query since the node started.
- `QueryLatency` — Overall query latency percentiles (in microseconds) and one/five/fifteen minute query throughput.
- `PartitionReads` — Histogram over the number of partitions read per query.
- `SSTableIndexesHit` — Histogram over the number of SSTable indexes read per query.
- `KDTreeChunkCacheLookups` — Histogram over the number of chunk cache lookups while reading kd-tree index files per query.
- `KDTreeChunkCacheMisses` — Histogram over the number of chunk cache misses while reading kd-tree index files per query.

Table query metrics

```
ObjectName:
org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=<keyspace>,table=<table>,
scope=TableQueryMetrics,name=<metric>
```

The table query metrics for the given keyspace and table:

- `TotalPartitionReads` — Total partition reads by all queries since the node started.
- `TotalQueriesCompleted` — Total number of successfully completed queries since the node started.
- `TotalQueryTimeouts` — Total number of timeouts from queries since the node started.
- `TotalRowsFiltered` — Total number of rows post-filtered by all queries since the node started.

Table state metrics

```
ObjectName:
org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=<keyspace>,table=<table>,
scope=TableStateMetrics,name=<metric>
```

The table state metrics for the given keyspace and table:

- `DiskPercentageOfBaseTable` — SAI size on Disk as a percentage of table size per table.
- `DiskUsedBytes` — Size on-disk in bytes of SAI indices per table.
- `TotalIndexBuildsInProgress` — Status of SAI indices per table currently in the `is_building` state.
- `TotalIndexCount` — Total number of SAI indices per table.
- `TotalQueryableIndexCount` — Status of SAI indices per table currently in the `is_querable` state.

Column query metrics for each numeric index

```
ObjectName:
org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=<keyspace>,table=<table>,
index=<index>,scope=ColumnQueryMetrics,name=<metric>
```

The column query metrics for the given keyspace, table, and index include:

- `KDTreeNiceTryLatency` — For numeric indexes, such as `age_sai_idx` in the **quickstart** examples, this metric may be used to present a histogram of the times spent waiting for chunk cache misses during kd-tree intersection (in microseconds) and one/five/fifteen minute chunk miss throughputs.

NOTE	The throughputs are zero if there are no cache misses during kd-tree intersection.

Column query metrics for each string index

```
ObjectName:
org.apache.cassandra.metrics:type=StorageAttachedIndex,keyspace=<keyspace>,table=<table>,
index=<index>,scope=ColumnQueryMetrics,name=<metric>
```

The column query metrics for the given keyspace, table, and index include:

- `TermsLookupLatency` — For string indexes, such as `country_sai_idx` in the **quickstart** examples, this metric shows terms lookup latency percentiles (in microseconds) per one/five/fifteen minute query throughput.

Range slice metrics

```
ObjectName:
org.apache.cassandra.metrics:type=ClientRequest,scope=RangeSlice,name=<metric>
```

The `RoundTripsPerReadHistogram` metric tracks the number of round-trip requests sent for range query commands from the coordinator. Fewer requests typically mean the server is operating more efficiently than ones requiring more requests to satisfy the same range queries.

Latency metric tracks the min, max, mean as well as a set of percentiles for range read requests latency. Timeouts metric tracks the number of timeouts for range read requests.

Virtual tables for SAI indexes and SSTables

Virtual tables for SAI indexes and SSTables

Storage Attached Indexing (SAI) provides CQL-based virtual tables that enable you to discover the current state of system metadata for SAI indices and associated SSTables. These virtual tables reside in the `system_views` keyspace.

For related information, see the **SAI information**.

system_views.indexes

The `system_views.indexes` virtual table contains stateful information about SAI indexes. This view provides information at the column index level, including the index name, number of indexed SSTables, disk usage, and index state. From the index state, the data reveals if the index is currently building, and whether the index can be queried.

Use CQL to view the table's description. Example:

```
DESCRIBE TABLE system_views.indexes;
```

```
/*
Warning: Table system_views.indexes is a virtual table and cannot be recreated with CQL.
Structure, for reference:
VIRTUAL TABLE system_views.indexes (
    keyspace_name text,
    index_name text,
    analyzer text,
    cell_count bigint,
    column_name text,
    indexed_sstable_count int,
    is_building boolean,
    is_queryable boolean,
    is_string boolean,
    per_column_disk_size bigint,
    per_table_disk_size bigint,
    table_name text,
    PRIMARY KEY (keyspace_name, index_name)
) WITH CLUSTERING ORDER BY (index_name ASC)
    AND comment = 'Storage-attached column index metadata';
*/
```

To view the current data, submit a query such as:

```
SELECT * FROM system_views.indexes;
```

```
 keyspace_name | index_name           | analyzer
| cell_count | column_name  | indexed_sstable_count | is_building | is_queryable |
is_string | per_column_disk_size | per_table_disk_size | table_name
---------------+----------------------+
-------------------------------------------------------------+------------+
-------------+----------------------+------------+-------------+-----------+
---------------------+---------------------+-----------------
       cycling |          age_sai_idx |
NoOpAnalyzer{} |          0 |          age |                     0 |       False |
True |     False |                     0 |                   0 | cyclist_semi_pro
       cycling |      country_sai_idx | NonTokenizingAnalyzer{caseSensitive=false,
normalized=true} |          0 |      country |                     0 |       False |
True |      True |                     0 |                   0 | cyclist_semi_pro
       cycling |     lastname_sai_idx | NonTokenizingAnalyzer{caseSensitive=false,
normalized=true} |          0 |     lastname |                     0 |       False |
True |      True |                     0 |                   0 | cyclist_semi_pro
       cycling | registration_sai_idx |
NoOpAnalyzer{} |          0 | registration |                     0 |       False |
True |     False |                     0 |                   0 | cyclist_semi_pro

(4 rows)
```

Table 1. system_views.indexes metadata

Column name	CQL type	Meaning
keyspace_name	text	The name of the keyspace to which the index belongs.
index_name	text	The name of the index.
analyzer	text	The toString representation of the analyzer used by the index.
cell_count	bigint	The number of indexed table cells, or the number of index value-key entries. This is the sum of the number of index entries in each SSTable.
column_name	text	The name of the indexed column.
indexed_sstable_count	int	The number of indexed SSTables. Note that SSTables without relevant data won't be indexed or counted here.
is_building	boolean	Whether there is a on going build for the index.

Column name	CQL type	Meaning
is_queryable	boolean	Whether it is possible to query the index. It won't be possible if the initial task build hasn't finished yet.
is_string	boolean	Whether the index is for a text field (ascii, text, or varchar).
per_column_disk_size	bigint	The on-disk size of the index components that are exclusive to the column, in bytes.
per_table_disk_size	bigint	The on-disk size of the index components that are shared with other SAI indexes for the same table, in bytes.
table_name	text	The name of the table to which the indexed column belongs.

system_views.sstable_indexes

The system_views.sstable_indexes virtual table has a row per SAI index and SSTable. This view describes individual SSTable indexes, and includes information around disk size, min/max row ID, the min/max ring token, and the write-time version of the index.

Use CQL to view the table's description. Example:

```
DESCRIBE TABLE system_views.sstable_indexes;
```

```
/*
Warning: Table system_views.sstable_indexes is a virtual table and cannot be recreated
with CQL.
Structure, for reference:
VIRTUAL TABLE system_views.sstable_indexes (
    keyspace_name text,
    index_name text,
    sstable_name text,
    cell_count bigint,
    column_name text,
    end_token text,
    format_version text,
    max_row_id bigint,
    min_row_id bigint,
    per_column_disk_size bigint,
    per_table_disk_size bigint,
    start_token text,
    table_name text,
```

```
    PRIMARY KEY (keyspace_name, index_name, sstable_name)
) WITH CLUSTERING ORDER BY (index_name ASC, sstable_name ASC)
    AND comment = 'SSTable index metadata';
*/
```

To view the current data, submit a query such as:

```
SELECT * FROM system_views.sstable_indexes;
```

Table 2. system_views.sstable_indexes metadata<

Column name	CQL type	Meaning
keyspace_name	text	The name of the keyspace to which the index belongs.
index_name	text	The name of the index.
sstable_name	text	The name of the SSTable.
cell_count	bigint	The number of indexed table cells, or the number of index value-key entries.
column_name	text	The name of the indexed column.
start_token	text	The start of the token range covered by the indexed SSTable.
end_token	text	The end of the token range covered by the indexed SSTable.
min_row_id	bigint	The minimum row id in the SSTable index.
max_row_id	bigint	The maximum row id in the SSTable index.
per_column_disk_size	bigint	The on-disk size of the SSTable index components that are exclusive to the column, in bytes.
per_table_disk_size	bigint	The on-disk size of the SSTable index components that are shared with other SAI indexes for the same table, in bytes.
table_name	text	The name of the table to which the indexed column belongs.

system_views.sstable_index_segments

The system_views.sstable_index_segments virtual table has a row per SAI index and SSTable segment. This view describes the segments of the SSTable indexes. It exposes the segment row ID offset and most of the information in the SSTable-level virtual table, specifically at a segment granularity.

Use CQL to view the table's description. Example:

```
DESCRIBE TABLE system_views.sstable_index_segments;
```

```
/*
Warning: Table system_views.sstable_index_segments is a virtual table and cannot be
recreated with CQL.
Structure, for reference:
VIRTUAL TABLE system_views.sstable_index_segments (
    keyspace_name text,
    index_name text,
    sstable_name text,
    segment_row_id_offset bigint,
    cell_count bigint,
    column_name text,
    component_metadata frozen<map<text, map<text, text>>>,
    end_token text,
    max_sstable_row_id bigint,
    max_term text,
    min_sstable_row_id bigint,
    min_term text,
    start_token text,
    table_name text,
    PRIMARY KEY (keyspace_name, index_name, sstable_name, segment_row_id_offset)
) WITH CLUSTERING ORDER BY (index_name ASC, sstable_name ASC, segment_row_id_offset ASC)
    AND comment = 'SSTable index segment metadata';
*/
```

To view the current data, submit a query such as:

```
SELECT * FROM system_views.sstable_index_segments;
```

Table 3. system_views.sstable_index_segments metadata

Column name	CQL type	Meaning
keyspace_name	text	The name of the keyspace to which the index belongs.
index_name	text	The name of the index.
sstable_name	text	The name of the SSTable.
segment_row_id_offset	bigint	The row id offset for the SSTable segment.
cell_count	bigint	The number of indexed segments, or the number of index segments value-key entries.
column_name	text	The name of the indexed column.

Column name	CQL type	Meaning
component_metadata	frozen<map<text, map<text, text>>>	The component metadata in the SSTable segment.
end_token	text	The end of the token range covered by the SSTable segment.
max_sstable_row_id	bigint	The maximum row id in the SSTable segment.
max_term	text	The maximum term in the SSTable segment.
min_sstable_row_id	bigint	The minimum row id in the SSTable segment.
min_term	text	The minimum term in the SSTable segment.
start_token	text	The start of the token range covered by the SSTable segment.
table_name	text	The name of the table to which the SSTable segment belongs.