SASI Index

`SASIIndex`, or `SASI` for short, is an implementation of Cassandra's `Index` interface that can be used as an alternative to the existing implementations. SASI's indexing and querying improves on existing implementations by tailoring it specifically to Cassandra's needs. SASI has superior performance in cases where queries would previously require filtering. In achieving this performance, SASI aims to be significantly less resource intensive than existing implementations, in memory, disk, and CPU usage. In addition, SASI supports prefix and contains queries on strings (similar to SQL's `LIKE = "foo*"` or `LIKE = "foo"'`).

The following goes on describe how to get up and running with SASI, demonstrates usage with examples, and provides some details on its implementation.

Using SASI

The examples below walk through creating a table and indexes on its columns, and performing queries on some inserted data.

The examples below assume the `demo` keyspace has been created and is in use.

```
cqlsh> CREATE KEYSPACE demo WITH replication = {
   ... 'class': 'SimpleStrategy',
   ... 'replication_factor': '1'
   ... };
cqlsh> USE demo;
```

All examples are performed on the `sasi` table:

```
cqlsh:demo> CREATE TABLE sasi (id uuid, first_name text, last_name text,
        ... age int, height int, created_at bigint, primary key (id));
```

Creating Indexes

To create SASI indexes use CQLs `CREATE CUSTOM INDEX` statement:

```
cqlsh:demo> CREATE CUSTOM INDEX ON sasi (first_name) USING
'org.apache.cassandra.index.sasi.SASIIndex'
        ... WITH OPTIONS = {
        ... 'analyzer_class':
        ...   'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer',
        ... 'case_sensitive': 'false'
        ... };
```

```
cqlsh:demo> CREATE CUSTOM INDEX ON sasi (last_name) USING
'org.apache.cassandra.index.sasi.SASIIndex'
        ... WITH OPTIONS = {'mode': 'CONTAINS'};

cqlsh:demo> CREATE CUSTOM INDEX ON sasi (age) USING
'org.apache.cassandra.index.sasi.SASIIndex';

cqlsh:demo> CREATE CUSTOM INDEX ON sasi (created_at) USING
'org.apache.cassandra.index.sasi.SASIIndex'
        ...  WITH OPTIONS = {'mode': 'SPARSE'};
```

The indexes created have some options specified that customize their behaviour and potentially performance. The index on `first_name` is case-insensitive. The analyzers are discussed more in a subsequent example. The `NonTokenizingAnalyzer` performs no analysis on the text. Each index has a mode: `PREFIX`, `CONTAINS`, or `SPARSE`, the first being the default. The `last_name` index is created with the mode `CONTAINS` which matches terms on suffixes instead of prefix only. Examples of this are available below and more detail can be found in the section on **OnDiskIndex**.The `created_at` column is created with its mode set to `SPARSE`, which is meant to improve performance of querying large, dense number ranges like timestamps for data inserted every millisecond. Details of the `SPARSE` implementation can also be found in the section on the **OnDiskIndex**. The `age` index is created with the default `PREFIX` mode and no case-sensitivity or text analysis options are specified since the field is numeric.

After inserting the following data and performing a `nodetool flush`, SASI performing index flushes to disk can be seen in Cassandra’s logs – although the direct call to flush is not required (see **IndexMemtable** for more details).

```
cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (556ebd54-cbe5-4b75-9aae-bf2a31a24500, 'Pavel', 'Yaskevich', 27, 181,
1442959315018);

cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (5770382a-c56f-4f3f-b755-450e24d55217, 'Jordan', 'West', 26, 173,
1442959315019);

cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (96053844-45c3-4f15-b1b7-b02c441d3ee1, 'Mikhail', 'Stepura', 36, 173,
1442959315020);

cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (f5dfcabe-de96-4148-9b80-a1c41ed276b4, 'Michael', 'Kjellman', 26, 180,
1442959315021);

cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (2970da43-e070-41a8-8bcb-35df7a0e608a, 'Johnny', 'Zhang', 32, 175,
1442959315022);
```

```
cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (6b757016-631d-4fdb-ac62-40b127ccfbc7, 'Jason', 'Brown', 40, 182,
1442959315023);

cqlsh:demo> INSERT INTO sasi (id, first_name, last_name, age, height, created_at)
        ... VALUES (8f909e8a-008e-49dd-8d43-1b0df348ed44, 'Vijay', 'Parthasarathy', 34,
183, 1442959315024);

cqlsh:demo> SELECT first_name, last_name, age, height, created_at FROM sasi;

 first_name | last_name     | age | height | created_at
------------+---------------+-----+--------+---------------
    Michael |      Kjellman |  26 |    180 | 1442959315021
    Mikhail |       Stepura |  36 |    173 | 1442959315020
      Jason |         Brown |  40 |    182 | 1442959315023
      Pavel |     Yaskevich |  27 |    181 | 1442959315018
      Vijay | Parthasarathy |  34 |    183 | 1442959315024
     Jordan |          West |  26 |    173 | 1442959315019
     Johnny |         Zhang |  32 |    175 | 1442959315022

(7 rows)
```

Equality & Prefix Queries

SASI supports all queries already supported by CQL, including LIKE statement for PREFIX, CONTAINS and SUFFIX searches.

```
cqlsh:demo> SELECT first_name, last_name, age, height, created_at FROM sasi
        ... WHERE first_name = 'Pavel';

  first_name | last_name | age | height | created_at
-------------+-----------+-----+--------+---------------
       Pavel | Yaskevich |  27 |    181 | 1442959315018

(1 rows)
```

```
cqlsh:demo> SELECT first_name, last_name, age, height, created_at FROM sasi
       ... WHERE first_name = 'pavel';

  first_name | last_name | age | height | created_at
-------------+-----------+-----+--------+---------------
       Pavel | Yaskevich |  27 |    181 | 1442959315018

(1 rows)
```

```
cqlsh:demo> SELECT first_name, last_name, age, height, created_at FROM sasi
        ... WHERE first_name LIKE 'M%';

 first_name | last_name | age | height | created_at
------------+-----------+-----+--------+---------------
    Michael |  Kjellman |  26 |    180 | 1442959315021
    Mikhail |   Stepura |  36 |    173 | 1442959315020

(2 rows)
```

Of course, the case of the query does not matter for the `first_name` column because of the options provided at index creation time.

```
cqlsh:demo> SELECT first_name, last_name, age, height, created_at FROM sasi
        ... WHERE first_name LIKE 'm%';

 first_name | last_name | age | height | created_at
------------+-----------+-----+--------+---------------
    Michael |  Kjellman |  26 |    180 | 1442959315021
    Mikhail |   Stepura |  36 |    173 | 1442959315020

(2 rows)
```

Compound Queries

SASI supports queries with multiple predicates, however, due to the nature of the default indexing implementation, CQL requires the user to specify `ALLOW FILTERING` to opt-in to the potential performance pitfalls of such a query. With SASI, while the requirement to include `ALLOW FILTERING` remains, to reduce modifications to the grammar, the performance pitfalls do not exist because filtering is not performed. Details on how SASI joins data from multiple predicates is available below in the **Implementation Details** section.

```
cqlsh:demo> SELECT first_name, last_name, age, height, created_at FROM sasi
        ... WHERE first_name LIKE 'M%' and age < 30 ALLOW FILTERING;

 first_name | last_name | age | height | created_at
------------+-----------+-----+--------+---------------
    Michael |  Kjellman |  26 |    180 | 1442959315021

(1 rows)
```

Suffix Queries

The next example demonstrates `CONTAINS` mode on the `last_name` column. By using this mode, predicates can search for any strings containing the search string as a sub-string. In this case the strings containing `a'' or` an''.

```
cqlsh:demo> SELECT * FROM sasi WHERE last_name LIKE '%a%';

 id                                   | age | created_at    | first_name | height |
last_name
--------------------------------------+-----+---------------+------------+--------+
---------------
 f5dfcabe-de96-4148-9b80-a1c41ed276b4 |  26 | 1442959315021 |    Michael |    180 |
Kjellman
 96053844-45c3-4f15-b1b7-b02c441d3ee1 |  36 | 1442959315020 |    Mikhail |    173 |
Stepura
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | 1442959315018 |      Pavel |    181 |
Yaskevich
 8f909e8a-008e-49dd-8d43-1b0df348ed44 |  34 | 1442959315024 |      Vijay |    183 |
Parthasarathy
 2970da43-e070-41a8-8bcb-35df7a0e608a |  32 | 1442959315022 |     Johnny |    175 |
Zhang

(5 rows)

cqlsh:demo> SELECT * FROM sasi WHERE last_name LIKE '%an%';

 id                                   | age | created_at    | first_name | height |
last_name
--------------------------------------+-----+---------------+------------+--------+
-----------
 f5dfcabe-de96-4148-9b80-a1c41ed276b4 |  26 | 1442959315021 |    Michael |    180 |
Kjellman
 2970da43-e070-41a8-8bcb-35df7a0e608a |  32 | 1442959315022 |     Johnny |    175 |
Zhang

(2 rows)
```

Expressions on Non-Indexed Columns

SASI also supports filtering on non-indexed columns like `height`. The expression can only narrow down an existing query using `AND`.

```
cqlsh:demo> SELECT * FROM sasi WHERE last_name LIKE '%a%' AND height >= 175 ALLOW
FILTERING;
```

```
 id                                   | age | created_at    | first_name | height |
last_name
--------------------------------------+-----+---------------+------------+--------+
---------------
 f5dfcabe-de96-4148-9b80-a1c41ed276b4 |  26 | 1442959315021 |    Michael |    180 |
Kjellman
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | 1442959315018 |      Pavel |    181 |
Yaskevich
 8f909e8a-008e-49dd-8d43-1b0df348ed44 |  34 | 1442959315024 |      Vijay |    183 |
Parthasarathy
 2970da43-e070-41a8-8bcb-35df7a0e608a |  32 | 1442959315022 |     Johnny |    175 |
Zhang

(4 rows)
```

Delimiter based Tokenization Analysis

A simple text analysis provided is delimiter based tokenization. This provides an alternative to indexing collections, as delimiter separated text can be indexed without the overhead of `CONTAINS` mode nor using `PREFIX` or `SUFFIX` queries.

```
cqlsh:demo> ALTER TABLE sasi ADD aliases text;
cqlsh:demo> CREATE CUSTOM INDEX on sasi (aliases) USING
'org.apache.cassandra.index.sasi.SASIIndex'
        ... WITH OPTIONS = {
        ... 'analyzer_class':
'org.apache.cassandra.index.sasi.analyzer.DelimiterAnalyzer',
        ... 'delimiter': ',',
        ... 'mode': 'prefix',
        ... 'analyzed': 'true'};
cqlsh:demo> UPDATE sasi SET aliases = 'Mike,Mick,Mikey,Mickey' WHERE id = f5dfcabe-de96-
4148-9b80-a1c41ed276b4;
cqlsh:demo> SELECT * FROM sasi WHERE aliases LIKE 'Mikey' ALLOW FILTERING;

 id                                   | age | aliases                | created_at    |
first_name | height | last_name
--------------------------------------+-----+------------------------+---------------+
------------+--------+-----------
 f5dfcabe-de96-4148-9b80-a1c41ed276b4 |  26 | Mike,Mick,Mikey,Mickey | 1442959315021 |
Michael |    180 |  Kjellman
```

Text Analysis (Tokenization and Stemming)

Lastly, to demonstrate text analysis an additional column is needed on the table. Its definition, index, and statements to update rows are shown below.

```
cqlsh:demo> ALTER TABLE sasi ADD bio text;
cqlsh:demo> CREATE CUSTOM INDEX ON sasi (bio) USING
'org.apache.cassandra.index.sasi.SASIIndex'
        ... WITH OPTIONS = {
        ... 'analyzer_class':
'org.apache.cassandra.index.sasi.analyzer.StandardAnalyzer',
        ... 'tokenization_enable_stemming': 'true',
        ... 'analyzed': 'true',
        ... 'tokenization_normalize_lowercase': 'true',
        ... 'tokenization_locale': 'en'
        ... };
cqlsh:demo> UPDATE sasi SET bio = 'Software Engineer, who likes distributed systems,
doesnt like to argue.' WHERE id = 5770382a-c56f-4f3f-b755-450e24d55217;
cqlsh:demo> UPDATE sasi SET bio = 'Software Engineer, works on the freight distribution
at nights and likes arguing' WHERE id = 556ebd54-cbe5-4b75-9aae-bf2a31a24500;
cqlsh:demo> SELECT * FROM sasi;

 id                                   | age | bio
| created_at    | first_name | height | last_name
--------------------------------------+-----+
-----------------------------------------------------------------------------------+
---------------+------------+--------+---------------
 f5dfcabe-de96-4148-9b80-a1c41ed276b4 |  26 |
null | 1442959315021 |    Michael |    180 |      Kjellman
 96053844-45c3-4f15-b1b7-b02c441d3ee1 |  36 |
null | 1442959315020 |    Mikhail |    173 |       Stepura
 6b757016-631d-4fdb-ac62-40b127ccfbc7 |  40 |
null | 1442959315023 |      Jason |    182 |         Brown
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | Software Engineer, works on the freight
distribution at nights and likes arguing | 1442959315018 |      Pavel |    181 |
Yaskevich
 8f909e8a-008e-49dd-8d43-1b0df348ed44 |  34 |
null | 1442959315024 |      Vijay |    183 | Parthasarathy
 5770382a-c56f-4f3f-b755-450e24d55217 |  26 |          Software Engineer, who likes
distributed systems, doesnt like to argue. | 1442959315019 |     Jordan |    173 |
West
 2970da43-e070-41a8-8bcb-35df7a0e608a |  32 |
null | 1442959315022 |     Johnny |    175 |         Zhang

(7 rows)
```

Index terms and query search strings are stemmed for the `bio` column because it was configured to use the `StandardAnalyzer` and `analyzed` is set to `true`. The `tokenization_normalize_lowercase` is similar to the `case_sensitive` property but for the `StandardAnalyzer`. These query demonstrates the stemming applied by StandardAnalyzer.

```
cqlsh:demo> SELECT * FROM sasi WHERE bio LIKE 'distributing';

 id                                   | age | bio
| created_at    | first_name | height | last_name
--------------------------------------+-----+
----------------------------------------------------------------------------------+
---------------+------------+--------+-----------
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | Software Engineer, works on the freight
distribution at nights and likes arguing | 1442959315018 |      Pavel |    181 |
Yaskevich
 5770382a-c56f-4f3f-b755-450e24d55217 |  26 |          Software Engineer, who likes
distributed systems, doesnt like to argue. | 1442959315019 |     Jordan |    173 |
West

(2 rows)

cqlsh:demo> SELECT * FROM sasi WHERE bio LIKE 'they argued';

 id                                   | age | bio
| created_at    | first_name | height | last_name
--------------------------------------+-----+
----------------------------------------------------------------------------------+
---------------+------------+--------+-----------
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | Software Engineer, works on the freight
distribution at nights and likes arguing | 1442959315018 |      Pavel |    181 |
Yaskevich
 5770382a-c56f-4f3f-b755-450e24d55217 |  26 |          Software Engineer, who likes
distributed systems, doesnt like to argue. | 1442959315019 |     Jordan |    173 |
West

(2 rows)

cqlsh:demo> SELECT * FROM sasi WHERE bio LIKE 'working at the company';

 id                                   | age | bio
| created_at    | first_name | height | last_name
--------------------------------------+-----+
----------------------------------------------------------------------------------+
---------------+------------+--------+-----------
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | Software Engineer, works on the freight
distribution at nights and likes arguing | 1442959315018 |      Pavel |    181 |
Yaskevich

(1 rows)
```

```
cqlsh:demo> SELECT * FROM sasi WHERE bio LIKE 'soft eng';

 id                                   | age | bio
| created_at    | first_name | height | last_name
--------------------------------------+-----+
--------------------------------------------------------------------------------+
---------------+------------+--------+-----------
 556ebd54-cbe5-4b75-9aae-bf2a31a24500 |  27 | Software Engineer, works on the freight
distribution at nights and likes arguing | 1442959315018 |      Pavel |    181 |
Yaskevich
 5770382a-c56f-4f3f-b755-450e24d55217 |  26 |          Software Engineer, who likes
distributed systems, doesnt like to argue. | 1442959315019 |     Jordan |    173 |
West

(2 rows)
```

Implementation Details

While SASI, at the surface, is simply an implementation of the `Index` interface, at its core there are several data structures and algorithms used to satisfy it. These are described here. Additionally, the changes internal to Cassandra to support SASI's integration are described.

The `Index` interface divides responsibility of the implementer into two parts: Indexing and Querying. Further, Cassandra makes it possible to divide those responsibilities into the memory and disk components. SASI takes advantage of Cassandra's write-once, immutable, ordered data model to build indexes along with the flushing of the memtable to disk – this is the origin of the name ``SSTable Attached Secondary Index''.

The SASI index data structures are built in memory as the SSTable is being written and they are flushed to disk before the writing of the SSTable completes. The writing of each index file only requires sequential writes to disk. In some cases, partial flushes are performed, and later stitched back together, to reduce memory usage. These data structures are optimized for this use case.

Taking advantage of Cassandra's ordered data model, at query time, candidate indexes are narrowed down for searching, minimizing the amount of work done. Searching is then performed using an efficient method that streams data off disk as needed.

Indexing

Per SSTable, SASI writes an index file for each indexed column. The data for these files is built in memory using the `OnDiskIndexBuilder`. Once flushed to disk, the data is read using the `OnDiskIndex` class. These are composed of bytes representing indexed terms, organized for efficient writing or searching respectively. The keys and values they hold represent tokens and positions in an SSTable and these are stored per-indexed term in `TokenTreeBuilder`s for writing, and `TokenTree`s for querying. These index files are memory mapped after being written to disk, for quicker access. For indexing data in the memtable, SASI uses its IndexMemtable class.

OnDiskIndex(Builder)

Each OnDiskIndex is an instance of a modified Suffix Array data structure. The OnDiskIndex is comprised of page-size blocks of sorted terms and pointers to the terms' associated data, as well as the data itself, stored also in one or more page-sized blocks. The OnDiskIndex is structured as a tree of arrays, where each level describes the terms in the level below, the final level being the terms themselves. The PointerLevel`s and their `PointerBlock`s contain terms and pointers to other blocks that end with those terms. The `DataLevel, the final level, and its DataBlock`s contain terms and point to the data itself, contained in `TokenTrees.

The terms written to the OnDiskIndex vary depending on its mode'': either PREFIX, CONTAINS, or SPARSE. In the PREFIX and SPARSE cases, terms' exact values are written exactly once per OnDiskIndex. For example, when using a PREFIX index with terms Jason, Jordan, Pavel, all three will be included in the index. A CONTAINS index writes additional terms for each suffix of each term recursively. Continuing with the example, a CONTAINS index storing the previous terms would also store ason, ordan, avel, son, rdan, vel, etc. This allows for queries on the suffix of strings. The SPARSE mode differs from PREFIX in that for every 64 blocks of terms a TokenTree is built merging all the TokenTrees for each term into a single one. This copy of the data is used for efficient iteration of large ranges of e.g. timestamps. The index mode'' is configurable per column at index creation time.

TokenTree(Builder)

The TokenTree is an implementation of the well-known B+-tree that has been modified to optimize for its use-case. In particular, it has been optimized to associate tokens, longs, with a set of positions in an SSTable, also longs. Allowing the set of long values accommodates the possibility of a hash collision in the token, but the data structure is optimized for the unlikely possibility of such a collision.

To optimize for its write-once environment the TokenTreeBuilder completely loads its interior nodes as the tree is built and it uses the well-known algorithm optimized for bulk-loading the data structure.

TokenTrees provide the means to iterate over tokens, and file positions, that match a given term, and to skip forward in that iteration, an operation used heavily at query time.

IndexMemtable

The IndexMemtable handles indexing the in-memory data held in the memtable. The IndexMemtable in turn manages either a TrieMemIndex or a SkipListMemIndex per-column. The choice of which index type is used is data dependent. The TrieMemIndex is used for literal types. AsciiType and UTF8Type are literal types by default but any column can be configured as a literal type using the is_literal option at index creation time. For non-literal types the SkipListMemIndex is used. The TrieMemIndex is an implementation that can efficiently support prefix queries on character-like data. The SkipListMemIndex, conversely, is better suited for other Cassandra data types like numbers.

The TrieMemIndex is built using either the ConcurrentRadixTree or ConcurrentSuffixTree from the com.goooglecode.concurrenttrees package. The choice between the two is made based on the indexing mode, `PREFIX` or other modes, and `CONTAINS` mode, respectively.

The `SkipListMemIndex` is built on top of `java.util.concurrent.ConcurrentSkipListSet`.

Querying

Responsible for converting the internal `IndexExpression` representation into SASI's `Operation` and `Expression` trees, optimizing the trees to reduce the amount of work done, and driving the query itself, the `QueryPlan` is the work horse of SASI's querying implementation. To efficiently perform union and intersection operations, SASI provides several iterators similar to Cassandra's `MergeIterator`, but tailored specifically for SASI's use while including more features. The `RangeUnionIterator`, like its name suggests, performs set unions over sets of tokens/keys matching the query, only reading as much data as it needs from each set to satisfy the query. The `RangeIntersectionIterator`, similar to its counterpart, performs set intersections over its data.

QueryPlan

The `QueryPlan` instantiated per search query is at the core of SASI's querying implementation. Its work can be divided in two stages: analysis and execution.

During the analysis phase, `QueryPlan` converts from Cassandra's internal representation of ``IndexExpression`s, which has also been modified to support encoding queries that contain ORs and groupings of expressions using parentheses (see the Cassandra Internal Changes section below for more details). This process produces a tree of `Operation``s, which in turn may contain `Expression`s, all of which provide an alternative, more efficient, representation of the query.

During execution, the `QueryPlan` uses the `DecoratedKey`-generating iterator created from the `Operation` tree. These keys are read from disk and a final check to ensure they satisfy the query is made, once again using the `Operation` tree. At the point the desired amount of matching data has been found, or there is no more matching data, the result set is returned to the coordinator through the existing internal components.

The number of queries (total/failed/timed-out), and their latencies, are maintined per-table/column family.

SASI also supports concurrently iterating terms for the same index across SSTables. The concurrency factor is controlled by the `cassandra.search_concurrency_factor` system property. The default is `1`.

QueryController

Each `QueryPlan` references a `QueryController` used throughout the execution phase. The `QueryController` has two responsibilities: to manage and ensure the proper cleanup of resources (indexes), and to strictly enforce the time bound per query, specified by the user via the range slice timeout. All indexes are accessed via the `QueryController` so that they can be safely released by it later. The `QueryController`'s `checkpoint` function is called in specific places in the execution path to ensure the time-bound is enforced.

QueryPlan Optimizations

While in the analysis phase, the `QueryPlan` performs several potential optimizations to the query. The goal of these optimizations is to reduce the amount of work performed during the execution phase.

The simplest optimization performed is compacting multiple expressions joined by logical intersections (`AND`) into a single `Operation` with three or more `Expression`s. For example, the query `WHERE age < 100 AND fname = 'p*' AND first_name != 'pa*' AND age > 21` would, without modification, have the following tree:

```
                      ┌───────────────┐
             ┌──────────────── │  AND   │─────────────┐
             │         └───────────────┘       │
            ▼                       ▼
          ┌───────────────┐              ┌──────────────────────┐
    ┌────────── │  AND   │ ──────────┐        │ age < 100  │
    │      └───────────────┘      │        └──────────────────────┘
   ▼                   ▼
┌─────────────────────┐           ┌───────────────┐
│ fname=p*  │         ┌── │  AND   │ ──────┐
└─────────────────────┘         │  └───────────────┘    │
                   ▼             ▼
                ┌─────────────────────┐   ┌─────────────────────┐
                │fname!=pa* │   │ age > 21  │
                └─────────────────────┘   └─────────────────────┘
```

`QueryPlan` will remove the redundant right branch whose root is the final `AND` and has leaves `fname != pa*` and `age > 21`. These `Expression`s will be compacted into the parent `AND`, a safe operation due to `AND` being associative and commutative. The resulting tree looks like the following:

```
                              ┌───────────────┐
                     ┌──────────────── │  AND   │ ────────────┐
                     │         └───────────────┘       │
                    ▼                       ▼
                  ┌───────────────┐              ┌──────────────────────┐
      ┌────────────────────── │  AND   │ ─────────────────┐     │age < 100  │
      │            └───────────────┘         │      └─────────────────────┘
     ▼                │            ▼
┌─────────────────────┐           │       ┌──────────────────────┐
│ fname=p*  │          ▼       │ age > 21  │
└─────────────────────┘     ┌─────────────────────┐  └──────────────────────┘
                │fname!=pa* │
                └─────────────────────┘
```

When excluding results from the result set, using `!=`, the `QueryPlan` determines the best method for handling it. For range queries, for example, it may be optimal to divide the range into multiple parts with a hole for the exclusion. For string queries, such as this one, it is more optimal, however, to simply note which data to skip, or exclude, while scanning the index. Following this optimization the tree looks like this:

```
                           ┌───────────────┐
                  ┌─────────────────│  AND   │──────────────┐
                  │         └───────────────┘       │
                 ▯                       ▯
               ┌───────────────┐              ┌──────────────────────┐
       ┌───────────────│  AND   │─────────────────┐     │ age < 100  │
       │        └───────────────┘         │     └──────────────────────┘
      ▯                        ▯
┌─────────────────────────────────────┐           ┌─────────────────────┐
│     fname=p*      │          │ age > 21  │
│ exclusions=[pa*]  │          └─────────────────────┘
└─────────────────────────────────────┘
```

The last type of optimization applied, for this query, is to merge range expressions across branches of the tree – without modifying the meaning of the query, of course. In this case, because the query contains all AND`s the `age expressions can be collapsed. Along with this optimization, the initial collapsing of unneeded `AND`s can also be applied once more to result in this final tree using to execute the query:

```
                 ┌───────────────┐
          ┌─────────────│  AND   │────────────────┐
          │       └───────────────┘        │
         ▯                      ▯
┌─────────────────────────────────────┐      ┌─────────────────────────────────┐
│     fname=p*      │     │ 21 < age < 100   │
│ exclusions=[pa*]  │     └──────────────────────────────────┘
└─────────────────────────────────────┘
```

Operations and Expressions

As discussed, the QueryPlan optimizes a tree represented by Operations as interior nodes, and Expressions as leaves. The Operation class, more specifically, can have zero, one, or two Operations as children and an unlimited number of expressions. The iterators used to perform the queries, discussed below in the `Range(Union|Intersection)Iterator'' section, implement the necessary logic to merge results transparently regardless of the `Operations children.

Besides participating in the optimizations performed by the QueryPlan, Operation is also responsible for taking a row that has been returned by the query and performing a final validation that it in fact does match. This satisfiesBy operation is performed recursively from the root of the Operation tree for a given query. These checks are performed directly on the data in a given row. For more details on how `satisfiesBy` works, see the documentation in the code.

Range(Union|Intersection)Iterator

The abstract `RangeIterator` class provides a unified interface over the two main operations performed by SASI at various layers in the execution path: set intersection and union. These operations are performed in a iterated, or ``streaming'', fashion to prevent unneeded reads of elements from either set. In both the intersection and union cases the algorithms take advantage of the data being pre-sorted using the same sort order, e.g. term or token order.

The `RangeUnionIterator` performs the ``Merge-Join'' portion of the Sort-Merge-Join algorithm, with the properties of an outer-join, or union. It is implemented with several optimizations to improve its performance over a large number of iterators – sets to union. Specifically, the iterator exploits the likely case of the data having many sub-groups of overlapping ranges and the unlikely case that all ranges will overlap each other. For more details see the javadoc.

The `RangeIntersectionIterator` itself is not a subclass of `RangeIterator`. It is a container for several classes, one of which, `AbstractIntersectionIterator`, sub-classes `RangeIterator`. SASI supports two methods of performing the intersection operation, and the ability to be adaptive in choosing between them based on some properties of the data.

`BounceIntersectionIterator`, and the `BOUNCE` strategy, works like the `RangeUnionIterator` in that it performs a ``Merge-Join'', however, its nature is similar to a inner-join, where like values are merged by a data-specific merge function (e.g. merging two tokens in a list to lookup in a SSTable later). See the javadoc for more details on its implementation.

`LookupIntersectionIterator`, and the `LOOKUP` strategy, performs a different operation, more similar to a lookup in an associative data structure, or ``hash lookup'' in database terminology. Once again, details on the implementation can be found in the javadoc.

The choice between the two iterators, or the `ADAPTIVE` strategy, is based upon the ratio of data set sizes of the minimum and maximum range of the sets being intersected. If the number of the elements in minimum range divided by the number of elements is the maximum range is less than or equal to `0.01`, then the `ADAPTIVE` strategy chooses the `LookupIntersectionIterator`, otherwise the `BounceIntersectionIterator` is chosen.

The SASIIndex Class

The above components are glued together by the `SASIIndex` class which implements `Index`, and is instantiated per-table containing SASI indexes. It manages all indexes for a table via the `sasi.conf.DataTracker` and `sasi.conf.view.View` components, controls writing of all indexes for an SSTable via its `PerSSTableIndexWriter`, and initiates searches with `Searcher`. These classes glue the previously mentioned indexing components together with Cassandra's SSTable life-cycle ensuring indexes are not only written when Memtable's flush, but also as SSTable's are compacted. For querying, the `Searcher` does little but defer to `QueryPlan` and update e.g. latency metrics exposed by

SASI.

Cassandra Internal Changes

To support the above changes and integrate them into Cassandra a few minor internal changes were made to Cassandra itself. These are described here.

SSTable Write Life-cycle Notifications

The `SSTableFlushObserver` is an observer pattern-like interface, whose sub-classes can register to be notified about events in the life-cycle of writing out a SSTable. Sub-classes can be notified when a flush begins and ends, as well as when each next row is about to be written, and each next column. SASI's `PerSSTableIndexWriter`, discussed above, is the only current subclass.

Limitations and Caveats

The following are items that can be addressed in future updates but are not available in this repository or are not currently implemented.

- The cluster must be configured to use a partitioner that produces ``LongToken`s, e.g. `Murmur3Partitioner``. Other existing partitioners which don't produce LongToken e.g. `ByteOrderedPartitioner` and `RandomPartitioner` will not work with SASI.
- Not Equals and OR support have been removed in this release while changes are made to Cassandra itself to support them.

Contributors

- Pavel Yaskevich
- Jordan West
- Michael Kjellman
- Jason Brown
- Mikhail Stepura