CQL for Cassandra 2.x

ドキュメント

2016年03月16日

目次

はじめに

このドキュメントでは、CQL for Cassandra 2.1および2.0.xについて説明します。Cassandra 2.1ドキュメントとCassandra 2.0ドキュメントは、このドキュメントを補完するもので、これらのドキュメントのいずれかを理解していることを前提にしています。

Cassandra 2.1の機能

Cassandra 2.1には、以下の新しいCQL機能が含まれています。

- ネストされたユーザー定義の型
- Cassandraがコミット・ログを再生するときに正しいカウントを維持するための改善されたカウンター・カラム
- 構成可能なカウンター・キャッシュ
- コレクションに対するインデックスのサポート。クエリー結果をフィルターするためのマップ・キーの使用を含みます。
- ミリ秒精度のタイムスタンプ
- 位置指定された型付きのフィールドを要素とする固定要素数の集合を保持する新しいタプル型

cqlshユーティリティーも、以下の点が改善されています。

- オペレーティング・システムのコマンドラインからCQL文を受け付けて実行する機能
- DESCRIBEコマンドを使用して型の定義を表示するためのサポート

DataStax Java Driver 2.0.0は、制限付きでCassandra 2.1をサポートします。このバージョンのドライバーは、新しい機能との互換性はありません。

Cassandra 2.0.xの機能

Cassandra 2.0.xの主な機能は以下のとおりです。

- INSERTおよびUPDATE文でIFキーワードを使用する軽量トランザクション
- テーブル、キースペース、またはインデックスが存在するかどうかの条件テストを実行することによるアプリケーション・エラーの防止。

  DROP KEYSPACEまたはCREATE TABLEのようなDROP文またはCREATE文にIF EXISTSまたはIF NOT EXISTSを含めるだけです。
- データベース・クラスターの内側または外側で実行されるイベントを発行するトリガーの初期サポート。
- 以前のリリースで除外されたALTER TABLE DROPコマンド。
- SELECT文でのカラム別名。RDBMS SQLの別名に似ています。
- パーティション・キー、クラスター化カラムを含め、複合プライマリ・キーの任意の部分に対するインデックス作成。

DataStaxドライバーはCassandra 2.0をサポートしています。

CQL for Cassandra 2.0では、スーパー・カラムが廃止されています。Cassandraでは引き続き、スーパー・カラムをクエリーするアプリをサポートしており、スーパー・カラムをその場でCQLコンストラクトと結果に読み替えます。

CQL Cassandra 2.0のcqlshコマンドには、以下のような変更があります。

- ASSUMEコマンドは廃止されました。

  ASSUMEの代わりにblobAsTypeおよびtypeAsBlob変換関数を使用してください。
- COPYコマンドは、コレクションに対してサポートされるようになりました。

Cassandra 2.0に含まれるCQLに、以下のCQLテーブル属性が追加されました。

- default_time_to_live
- memtable_flush_period_in_ms
- populate_io_cache_on_flush
- speculative_retry

CQLデータ・モデリング

データ・モデリング例

Cassandraのデータ・モデルは、整合性が調整可能なパーティション分割される行ストアです。行はテーブルにまとめられます。テーブルのプライマリ・キーの最初の要素はパーティション・キーです。パーティション内では、各行がキーに対応する残りのカラムに応じてクラスター化されます。他のカラムには、プライマリ・キーとは別にインデックスを付けることができます。テーブルは実行時に作成、削除、変更を行うことができます。更新やクエリーを妨げることはありません。

音楽サービスの例は、Cassandraデータのモデル化に複合キー、クラスター化カラム、およびコレクションを使用する方法を示しています。

音楽サービスの例

このソーシャル音楽サービスの例では、タイトル(title)、アルバム(album)、アーティスト(artist)の各カラムを持つ楽曲(songs)テーブルが必要です。さらに、実際のオーディオ・ファイルそのもののデータ(data)カラムが必要です。このテーブルはプライマリ・キーとしてUUIDを使用しています。

```
CREATE TABLE songs (
id uuid PRIMARY KEY,
title text,
album text,
artist text,
data blob
 );
```

リレーショナル・データベースでは、楽曲(songs)に対する外部キーを持つプレイリスト(playlists)テーブルを作成しますが、Cassandra では、分散システムでの結合は効率的ではないのでデータを非正規化します。後でテーブルを結合して楽曲にタグ付するのと同じ目的のためにコレクションを使用する方法を説明します。プレイリスト・データを表現するため、以下のようなテーブルを作成します。

```
CREATE TABLE playlists (
id uuid,
song_order int,
song_id uuid,
title text,
album text,
artist text,
PRIMARY KEY  (id, song_order ) );
```

プレイリスト・テーブル内でのidとsong_orderの組み合わせは、プレイリスト・テーブル内の行を一意に識別します。同じidを持つ複数の行を、異なるsong_order値を含んでいる行の数だけ持つことができます。

注： UUIDは、データの順次処理、すなわち複数のマシンにわたる同期化を自動的にインクリメントするのに便利です。単純化のため、この例ではint song_orderを使用します。

データの例をプレイリストに挿入した後で全データを選択すると、出力は以下のようになります。

```
SELECT * FROM playlists;
```

```
 id          | song_order | album         | artist          | song_id     | title
-------------+------------+---------------+-----------------+-------------+----------------------
 62c36092... |          1 | Tres Hombres |          ZZ Top | a3e64f8f... |            La Grange
 62c36092... |          2 | We Must Obey |       Fu Manchu | 8a172618... |    Moving in Stereo
 62c36092... |          3 | Roll Away    | Back Door Slam | 2b09185b... | Outside Woman Blues
```

次の例は、フィルターとしてアーティストを使用したクエリーを作成する方法を示しています。まず、プレイリスト・テーブルに少しデータを追加して、後でコレクションの例が興味深くなるようにします。

```
INSERT INTO playlists (id, song_order, song_id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 4,
7db1a490-5878-11e2-bcfd-0800200c9a66,
'Ojo Rojo', 'Fu Manchu', 'No One Rides for Free');
```

これまでに指定されたスキーマでは、アーティストをフィルターとして含むクエリーは、プレイリストのデータセット全体を対象とした順次スキャンが必要になります。Cassandraではそのようなクエリーは拒否されます。先にアーティストのインデックスを作成しておけば、Cassandraは効率的に対象のレコードを取り出します。

```
CREATE INDEX ON playlists( artist );
```

たとえば、Fu Manchuの楽曲を探すためにプレイリストにクエリーしてみます。

```
SELECT album, title FROM playlists WHERE artist = 'Fu Manchu';
```

出力は以下のようになります。

```
 album                | title
----------------------+-----------------
 We Must Obey         | Moving in Stereo
 No One Rides for Free | Ojo Rojo
```

複合キーとクラスター化

複合プライマリ・キーにはパーティション・キーが含まれます。パーティション・キーによって、データの格納先となるノードが決まります。複合プライマリ・キーにはまた、1つまたは複数の追加カラムが含まれます。それらのカラムによって、パーティションごとのクラスター化が決まります。Cassandraは、プライマリ・キー定義内の最初のカラム名をパーティション・キーとして使用します。たとえば、プレイリスト・テーブルで、idはパーティション・キーです。残りのカラム、すなわちプライマリ・キー定義内でパーティション・キーでないカラムは、クラスター化カラムです。プレイリスト・テーブルで、song_order はクラスター化カラムです。各パーティションのデータは、残りのカラム、すなわちプライマリ・キー定義内でパーティション・キーでないカラムに基づいてクラスター化されています。物理ノード上では、パーティション・キーに属する行がクラスター化カラムに基づく順番で格納されていると、行の取得が非常に効率的になります。たとえば、プレイリスト・テーブル内のidはパーティション・キーなので、プレイリストの楽曲はすべて、残るsong_orderカラムの順番でクラスター化されています。

テーブルの同じパーティション・キーを共有する行の挿入、更新、削除操作は、アトミックかつ個別に行われます。

ディスク上にある1つの連続したデータ・セットにクエリーしてプレイリストの楽曲を取得できます。

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
ORDER BY song_order DESC LIMIT 50;
```

出力は以下のようになります。

```
 id          | song_order | album                  | artist          | song_id     | title
-------------+------------+------------------------+-----------------+-------------+----------------
 62c36092... |          4 | No One Rides for Free |       Fu Manchu | 7db1a490... |            Ojo
 62c36092... |          3 |             Roll Away | Back Door Slam | 2b09185b... | Outside Woman
 62c36092... |          2 |          We Must Obey |       Fu Manchu | 8a172618... |    Moving in S
 62c36092... |          1 |          Tres Hombres |          ZZ Top | a3e64f8f... |             La G
```

Cassandraは、データの行全体をパーティション・キーに従ってノードに格納します。パーティションにデータが多すぎて、データを複数のノードに分散させたい場合は、複合パーティション・キーを使ってください。

コレクション・カラム

CQLでは、以下のコレクション型を導入しています。

- セット
- リスト
- マップ

リレーショナル・データベースでユーザーが複数の電子メールアドレスを持てるようにするには、ユーザー・テーブルと多対一（結合）関係を持つemail_addressesテーブルを作成します。CQLでは、カラムをコレクションとして定義することによって、従来の複数の電子メールアドレスのユースケースだけでなく他のユースケースにも対応しています。複数電子メールアドレスの問題を解決するためにセット・コレクション型を使用することは、便利で直観的です。

コレクション型の他の使用方法は、音楽サービスの例で示しています。

テーブルへのコレクションの追加

音楽サービスの例には、楽曲にタグを付ける機能が含まれています。リレーショナルという観点から見ると、ストレージ・エンジンの各行をパーティションとして考えることができます。その中に（オブジェクト）行がクラスター化されています。楽曲にタグを付けるには、コレクション・セットを使用します。コレクション・セットを、CREATE TABLEまたはALTER TABLE文を使用して宣言します。プレイリスト・テーブルはすでに存在するので、そのテーブルを変更してコレクション・セット（tags）を追加してください。

```
ALTER TABLE playlists ADD tags set<text>;
```

コレクションの更新

プレイリスト・テーブルを更新してタグ・データを挿入します。

```
UPDATE playlists SET tags = tags + {'2007'}
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 2;
UPDATE playlists SET tags = tags + {'covers'}
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 2;
UPDATE playlists SET tags = tags + {'1973'}
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 1;
UPDATE playlists SET tags = tags + {'blues'}
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 1;
UPDATE playlists SET tags = tags + {'rock'}
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 4;
```

音楽レビュー（reviews）リストおよびライブのスケジュール（マップ・コレクション）をテーブルに追加できます。

```
ALTER TABLE playlists ADD reviews list<text>;
ALTER TABLE playlists ADD venue map<timestamp, text>;
```

セット、リスト、またはマップの要素はそれぞれ内部的には1つのCassandra カラムとして格納されます。セットを更新するには、UPDATE コマンドと加算(+)演算子を使用して追加するか、または減算(-)演算子を使用して要素を削除します。たとえば、セットを更新するには:

```
UPDATE playlists
SET tags = tags + {'punk rock'}
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND song_order = 4;
```

リストを更新するには、中かっこの代わりに大かっこを使う類似の構文を使用します。

```
UPDATE playlists
SET reviews = reviews + [ 'best lyrics' ]
WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 and song_order = 4;
```

マップを更新するには、INSERT を使用してマップ・コレクション内のデータを指定します。

```
INSERT INTO playlists (id, song_order, venue)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 4,
{ '2013-9-22 22:00'  :'The Fillmore',
'2013-10-1 21:00' :'The Apple Barrel'});

INSERT INTO playlists (id, song_order, venue)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 3,
{'2014-1-22 22:00:00' :'Cactus Cafe',
'2014-01-12 20:00' :'Mohawk'});
```

マップにデータを挿入すると、マップ全体が置き換えられます。

この時点でプレイリスト・テーブルからのデータをすべて選択すると、出力は次のようになります。

```
 id     |song_order|album                |artist        |reviews         |song_
--------+----------+---------------------+--------------+----------------+-----
 62c...|         1|          Tres Hombres|         ZZ Top|            null|a3e..
 62c...|         2|          We Must Obey|      Fu Manchu|            null|8a1..
 62c...|         3|             Roll Away|Back Door Slam|            null|2b0..
 62c...|         4|No One Rides for Free|      Fu Manchu|["best lyrics"]|7db..

g_id|tags                  |title              |venue
----+----------------------+-------------------+--------------------------------
... |    {"1973", "blues"}|          La Grange|null
... |   {"2007", "covers"}|   Moving in Stereo|null
... |                 null|Outside Woman Blues|{"2014-01-12..."Cactus Cafe"}
... |{"punk rock", "rock"}|           Ojo Rojo|{"2013-09-22..."The Apple Barrel"}
```

コレクションのインデックス作成

Cassandra 2.1以降では、コレクションにインデックスを付け、特定の値を含んでいるコレクションを見つけるためにデータベースにクエリーすることができます。音楽サービスの例で続けると、Fillmoreでデビューし、bluesタグが付けられた楽曲を探したいとします。tagsセットとvenueマップにインデックスを付けます。tagsセットとvenueマップ内の値をクエリーする方法については、次のセクションで示します。

```
CREATE INDEX ON playlists (tags);
CREATE INDEX mymapvalues ON playlists (venue);
```

mymapvaluesのようなインデックス名の指定は、任意です。

コレクション・マップ・キーのインデックス作成

最後の例では、venueマップ・カラム名を使用してvenueコレクション値のインデックスを作成しています。マップ・コレクション・キーのインデックスを作成することもできます。マップ・キーは、JSON形式配列のコロンの左側にあるリテラルです。マップ値は、コロンの右側のリテラルです。

```
{ literal : literal, literal : literal ... }
```

たとえば、venueマップ内のコレクション・キーはタイムスタンプです。venueマップ内のコレクション値は'The Fillmore'と'Apple Barrel'です。

```
 artist    | venue
-----------+-----------------------------------------------------------------------------------------------------
 Fu Manchu | {'2013-09-22 12:01:00-0700': 'The Fillmore', '2013-10-01 18:00:00-0700': 'The Apple Barrel'}
```

マップのキーとマップの値に対するインデックスは共存できません。たとえば、mymapindexを作成した場合は、それを削除してから、KEYSキーワードとマップ名（ネストされた丸かっこの中に入れる）を使用して、マップ・キーのインデックスを作成する必要があります。

```
DROP INDEX mymapvalues;
CREATE INDEX mymapkeys ON playlists (KEYS(venue));
```

コレクション内のデータのフィルタリング

tagsセット・コレクションにデータを追加した後にtagsセットを検索すると、タグの集合が返されます。

```
SELECT album, tags FROM playlists;
```

```
 album                    | tags
--------------------------+-----------------------
 Tres Hombres             |      {'1973', 'blues'}
 We Must Obey             |     {'2007', 'covers'}
 Roll Away                |                   null
 No One Rides for Free    | {'punk rock', 'rock'}
```

コレクションのインデックスを作成してあることを前提に、セット・コレクションの値を使用してデータをフィルターするには、SELECT文にコレクション・カラム名を含めます。たとえば、"blues"など、特定のタグを含んでいる行を、WHERE句のCONTAINS 条件を使用して見つけます。

```
SELECT album, tags FROM playlists WHERE tags CONTAINS 'blues';
```

出力は、bluesタグを含んでいるプレイリスト・テーブルからの行で

```
 album                    | tags
--------------------------+-----------------------
 Tres Hombres             |      {'1973', 'blues'}
```

す。

マップ値またはマップ・キーでフィルター

2種類のマップ・コレクション・インデックスを作成できます。マップ値のインデックスとマップ・キーのインデックスです。これらの2種類のインデックスは同じコレクションに共存できません。マップ値のインデックスが作成されているこ

とを前提に、venueマップ内の値をWHERE句のCONTAINS条件を使用してデータをフィルターします。文は、セットまたはリスト内のデータをフィルターするため使用した文と同じです。

```
SELECT artist, venue FROM playlists WHERE venue CONTAINS 'The Fillmore';
```

出力は、要求したFillmoreでデビューした楽曲のデータです。

マップ・キーのインデックスが作成されていることを前提に、venueマップ内のキーを使用してデータをフィルターします。

```
SELECT album, venue FROM playlists WHERE venue CONTAINS KEY '2013-09-22
 22:00:00-0700';
```

どのような場合にコレクションを使用するのか

少数のデータを格納または非正規化したいときにコレクションを使用します。コレクションの各項目の値は64Kに制限されます。他の制限事項も適用されます。コレクションは、ユーザーの電話番号や電子メールに付けられるラベルなどのデータを格納する場合に有効です。格納する必要があるデータが無限に成長する可能性がある場合、たとえばユーザーが送信したすべてのメッセージやセンサーが検知したすべてのイベントを格納するような場合は、コレクションを使用しないでください。代わりに、複合プライマリ・キーを持つテーブルを使用してクラスター化カラムにデータを格納してください。

インデックスの作成

インデックスは、パーティション・キー以外の属性を使用してCassandra内のデータにアクセスする手段を提供します。利点は、与えられた条件に一致するデータをすばやく効率的に検索できる点です。インデックス機能は、インデックスの作成対象となる値が含まれているテーブルとは別の隠れたテーブルにカラム値のインデックスを作成します。Cassandraには、インデックスを利用したクエリー時に、インデックスの古くなったデータに起因してデータが誤って取り出されるという望ましくない状況を防ぐためのいくつかの手法があります。

前述のように、Cassandra 2.1以降では、コレクション・カラムにインデックスを付けることができます。

どのような場合にインデックスを使用するのか

Cassandraのビルトイン・インデックスは、インデックス付けされた同じ値を含んでいる行を多数持つテーブルに最適です。特定のカラムに存在する固有値が多いほど、インデックスをクエリーして維持するためのオーバーヘッドが大きくなります。たとえば、10億の楽曲が入っているプレイリストがあり、アーティストで楽曲を検索したいとします。多くの楽曲は、アーティストのカラム値が同じです。アーティスト・カラムは、インデックスの良い候補となります。

どのような場合にインデックスを使用しないか

以下のような場合はインデックスを使用しないでください。

- カーディナリティが高いカラム。少数の結果を求めて膨大な量のレコードをクエリーすることになるからです ... 詳細
- カウンター・カラムを使用するテーブル
- 頻繁に更新または削除されるカラム... 詳細
- クエリーの範囲を狭めずに大きなパーティション内で行を検索する場合... 詳細

カーディナリティが高いカラムのインデックスを使用する場合の問題点

多くの固有値を持つカーディナリティが高いカラムのインデックスを作成した場合、フィールドを対象にクエリーを実行すると、少数の結果を得るのにも多数のシークが伴います。10億の楽曲が入っているテーブルでは、アーティス

トではなく作曲家（一般的には楽曲ごとに固有の値）で楽曲を検索すると、非常に効率が悪くなる可能性があります。Cassandraのビルトイン・インデックスを使用するのではなく、インデックスの形式として手動でテーブルを維持する方が恐らくより効率的です。固有のデータを含んでいるカラムでは、インデックス付きカラムを持つテーブルに対するクエリーの量がさほど多くなく、常に負荷がかかる状態にならない限り、便宜上インデックスを使用することはパフォーマンスの点では問題ありません。

逆に、ブーリアン・カラムなどの極端にカーディナリティの低いカラムのインデックスを作成するのは無意味です。インデックスの各値がインデックス内で1行になり、たとえばすべてのfalse値に対する巨大な行になります。foo = trueとfoo = falseを持つ多数のインデックス対象カラムのインデックスを作成するのは有益ではありません。

頻繁に更新または削除されるカラムでインデックスを使用する場合の問題点

Cassandraはトゥームストーンを、限界の100Kセルに達するまでインデックス内に格納します。トゥームストーンの限界を超えると、インデックス対象の値を使用しているクエリーは失敗します。

クエリーの範囲を狭めずに大きなパーティション内の行を検索するためにインデックスを使用する問題

大きなクラスター内のインデックス付きカラムへのクエリーは、通常、複数のデータ・パーティションから応答を照合する必要があります。クエリー応答は、クラスターに加わるマシンが多くなるほど遅くなります。次のセクションに示すように、大きなパーティション内で行を探すときに検索を狭めるとパフォーマンスへの影響を回避できます。

インデックスの使用

CQLを使用して、テーブルを定義した後にカラムのインデックスを作成することができます。Cassandra 2.1以降では、コレクション・カラムのインデックスを作成することができます。音楽サービスの例では、プレイリストのアーティスト・カラムのインデックスを作成する方法と、特定のアーティストの楽曲を取得するためにCassandraにクエリーする方法を示しています。

```
CREATE INDEX artist_names ON playlists( artist );
```

インデックス名の指定は任意です。名前を指定しなかった場合、Cassandraはartist_idxのような名前を割り当てます。名前を指定する場合は、たとえばartist_namesなど、名前はキースペース内で固有である必要があります。アーティスト・カラムのインデックスを作成し、プレイリスト・テーブルに値を挿入した後では、直接、アーティスト名、たとえばFu Manchuなど名前でCassandraにクエリーしたときに効率が高くなります。

```
SELECT * FROM playlists WHERE artist = 'Fu Manchu';
```

前述のように、大きなパーティションで行を探すときは、検索の範囲を狭めます。このクエリーは、ほとんどデータを使用していない不自然な例ですが、1つのidまで検索を狭めています。

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204 AND
 artist = 'Fu Manchu';
```

出力は以下のとおりです。

```
 id          | song_order | album                  | artist    | song_id     | title
-------------+------------+------------------------+-----------+-------------+------------------
 62c36092... |          2 |          We Must Obey | Fu Manchu | 8a172618... | Moving in Stereo
 62c36092... |          4 | No One Rides for Free | Fu Manchu | 7db1a490... |         Ojo Rojo
```

複数のインデックスの使用

たとえば、プレイリスト・テーブルのアルバム・カラムとタイトル・カラムの両方にインデックスを作成でき、WHERE句の複数の条件を使用して結果をフィルターするとします。実際には、後述するように、カラムのカーディナリティに依存するので、これらのカラムは良い選択肢ではないかもしれません。

```
CREATE INDEX album_name ON playlists ( album );
```

```
CREATE INDEX title_name ON playlists ( title );
```

```
SELECT * FROM playlists
WHERE album = 'Roll Away' AND title = 'Outside Woman Blues'
ALLOW FILTERING ;
```

WHERE句の条件に複数のデータが一致する場合、Cassandraは効率を上げるために、最も発生頻度の低い条件を選択して処理します。たとえば、Blind Joe ReynoldsのバージョンとCreamのバージョンの"Outside Woman Blues"のデータがプレイリスト・テーブルに挿入されているとします。データベース内で"Outside Woman Blues"という名前の楽曲の数より"Roll Away"という名前のアルバムの数が少ない場合は、Cassandraはアルバム名を最初にクエリーします。行の範囲を検索するなど潜在的に高負荷のクエリーを実行しようとする場合は、CassandraはALLOW FILTERINGディレクティブを必要とします。

インデックスの作成と維持

インデックスの利点は、データの挿入と維持の運用上の容易さです。インデックスはバックグラウンドで自動的に作成されるので、読み取りや書き込み操作を妨げることはありません。クライアントが維持するインデックスとしてのテーブルは手動で作成する必要があります。たとえば、songs_by_artistなどのテーブルを作成することでアーティスト・カラムにインデックス付けした場合、クライアント・アプリケーションは、楽曲テーブルのデータをそのテーブルにも登録しなければなりません。

インデックスを作成し直すには、`nodetool rebuild_index`コマンドを使用します。

レガシー・アプリケーションでの作業

CQLは、内部で、行とカラムのマッピングをThrift APIマッピングから変えません。CQLとThriftは同じストレージ・エンジンを使用します。CQLでは、Thriftと同じクエリー主導の非正規化されたデータ・モデリング原理をサポートしています。既存のアプリケーションはCQLにアップグレードする必要はありません。CQLの抽象化層は、新しいアプリケーションに対してCQLを使いやすくします。ThriftとCQLの詳細な比較については、"A Thrift to CQL Upgrade Guide"およびCQL for Cassandra expertsを参照してください。

レガシー・テーブルの作成

COMPACT STORAGEディレクティブを使用して、CQLでレガシー(Thrift/CLI互換)テーブルを作成できます。CREATE TABLEコマンドと一緒にCOMPACT STORAGEディレクティブを使用すると、古いCassandraアプリケーションとの後方互換性が得られます。新しいアプリケーションでは通常は使用を避けるべきです。

COMPACT STORAGEでは、各非プライマリ・キー・カラムを、ディスク上にある1つのカラムに対応するカラムに格納するのではなく、行全体を、ディスク上にある1つのカラムに格納します。COMPACT STORAGEを使用すると、プライマリ・キーに属さない新しいカラムの追加が妨げられます。

CQLクエリーの使用

CQLを使用して、レガシー・テーブルにクエリーできます。CQLで管理されるレガシー・テーブルには、暗黙のWITH COMPACT STORAGEディレクティブが含まれています。パーティション内のデータにカラム名が定義されていないレガシー・テーブルにクエリーするためCQLを使用すると、Cassandraはデータの名前(column1、value1など)を生成します。RENAME句を使用すると、デフォルトのカラム名を、より意味のある名前に変更できます。

```
ALTER TABLE users RENAME userid to user_id;
```

CQLは、Thrift API、CLI、以前のCQLバージョンで作成された動的テーブルをサポートしています。たとえば、動的テーブルは以下のように表現され、クエリーされます。

```
CREATE TABLE clicks (
userid uuid,
url text,
timestamp date
PRIMARY KEY  (userid, url ) ) WITH COMPACT STORAGE;

SELECT url, timestamp
FROM clicks
WHERE  userid = 148e9150-1dd2-11b2-0000-242d50cf1fff;

SELECT timestamp
FROM clicks
WHERE  userid = 148e9150-1dd2-11b2-0000-242d50cf1fff
AND  url = 'http://google.com';
```

これらのクエリーでは、等価条件のみが有効です。

CQLの使用

CQLは、新しいアプリケーション向けに、Cassandraに対するThrift APIよりもシンプルなAPIを提供します。Thrift APIと旧バージョンのCQLは、Cassandraの内部ストレージ構造を露出させます。CQLは、この構造の実装の詳細を隠す抽象化層を追加し、コレクションやその他の一般的なコード化に対するネイティブな構文を備えています。

CQLのアクセス

CQLへの一般的なアクセス方法は以下のとおりです。

- Cassandraノードのコマンドラインで、Pythonベースのコマンドライン・クライアントであるcqlshを起動します。
- グラフィカル・ユーザー・インターフェイスであるDataStax DevCenterを使用します。
- アプリケーション開発には、公式のDataStax C#、Java、またはPythonオープンソース・ドライバーを使用できます。
- プログラムによるアクセスにはset_cql_version Thriftメソッドを使用します。

このドキュメントでは、cqlshを使用した例を示しています。

cqlshの起動

Linuxでのcqlshの起動

このタスクについて

この手順では、Linuxでのcqlshの起動方法について簡単に説明します。cqlshコマンドについては、後で詳しく説明します。

手順

1. Cassandraインストール・ディレクトリーに移動します。
2. たとえば、Mac OSXでcqlshを起動します。

   ```
   bin/cqlsh
   ```

   セキュリティ機能を使用している場合は、ユーザー名とパスワードを指定します。

3. 任意で、別のノードでcqlshを起動する場合には、IPアドレスとポートを指定します。

```
bin/cqlsh 1.2.3.4 9042
```

Windowsでのcqlshの起動

このタスクについて

この手順では、Windowsでのcqlshの起動方法について簡単に説明します。cqlshコマンドについては、後で詳しく説明します。

手順

1. コマンド・プロンプトを開きます。
2. Cassandraのbinディレクトリーに移動します。
3. cqlshを起動するコマンドを入力します。

```
python cqlsh
```

任意で、別のノードでcqlshを起動する場合には、IPアドレスとポートを指定します。

```
python cqlsh 1.2.3.4 9160
```

タブ補完機能の使用

cqlshコマンドを補完する方法に関するヒントを見るには、タブ補完を使用します。Mac OSXなどの一部のプラットフォームでは、タブ補完がインストールされた状態で出荷されていません。Mac OSXにタブ補完機能をインストールするには、easy_installを使用します。

```
easy_install readline
```

キースペースの作成と更新

キースペースの作成は、SQLデータベースの作成に相当しますが、多少異なります。Cassandraキースペースは、データがノードでどのようにレプリケートされるかを定義する名前空間です。通常、クラスターには、アプリケーションごとに1つのキースペースがあります。レプリケーションはキースペース単位で管理されるため、異なるレプリケーション要件を持つデータは通常、異なるキースペースに存在します。キースペースは、データ・モデル内で重要なマップ層として使用することを目的としていません。テーブル・セットのデータ・レプリケーションを管理することを目的としています。

キースペースを作成する場合は、キースペースをレプリケーションするためのストラテジ・クラスを指定します。Cassandraの評価には、SimpleStrategyクラスを使用すると良いでしょう。実稼働環境で使用する場合、または混在ワークロードで使用する場合は、NetworkTopologyStrategyクラスを使用します。

1つのノード・クラスターなどを使用した評価のためにNetworkTopologyStrategyを使用するには、デフォルトのデータ・センター名を指定します。デフォルトのデータ・センター名を調べるには、`nodetool status`コマンドを使用します。たとえば、以下のLinuxのインストール・ディレクトリーの場合、

```
bin/nodetool status
```

出力は以下のとおりです。

```
Datacenter:datacenter1
=======================
Status=Up/Down
```

```
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens  Owns (effective)  Host ID
                Rack
UN  127.0.0.1  41.62 KB   256     100.0%            75dcca8f-3a90-4aa3-
a2f9-3e162baa4990  rack1
```

実稼働環境でNetworkTopologyStrategyを使用するには、デフォルトのスニッチであるSimpleSnitchをネットワーク対応スニッチに変更し、スニッチ・プロパティ・ファイルで1つ以上のデータ・センター名を定義して、データ・センター名を使用してキースペースを定義する必要があります。これを行わないと、テーブルへのデータの挿入などの書き込み要求を完了できず、以下のエラー・メッセージがログに記録されます。

```
Unable to complete request:one or more nodes were unavailable.
```

スニッチ・プロパティ・ファイルでデータ・センター名を定義するか、またはdatacenter1という名前の1つのデータ・センターを使用しない限り、NetworkTopologyStrategyを使用するキースペース内のテーブルにデータを挿入できません。

キースペース作成の例

このタスクについて

Cassandraにクエリーするには、最初にキースペースを作成して使用します。任意のデータ・センター名を選択して、その名前をスニッチのプロパティ・ファイルに登録できます。あるいは、1つのデータ・センターのクラスターを使用する場合は、OS Cassandraでデフォルトのデータ・センター名（datacenter1など）を使用し、プロパティ・ファイルへの名前の登録をスキップします。

手順

1. キースペースを作成します。

   ```
   cqlsh> CREATE KEYSPACE demodb
   WITH REPLICATION = { 'class' :'NetworkTopologyStrategy', 'datacenter1' :
    3 };
   ```

2. キースペースを使用します。

   ```
   USE demodb;
   ```

レプリケーション係数の更新

このタスクについて

レプリケーション係数を大きくすると、Cassandraクラスターに格納されているキースペース・データのコピーの総数が増加します。セキュリティ機能を使用する場合は、system_authキースペースのレプリケーション係数をデフォルトの1よりも大きくすることが特に重要になります。なぜなら、唯一のレプリカを含んでいるノードが停止するとクラスターにログインできなくなるからです。

手順

1. クラスター内のキースペースを更新し、そのレプリケーション・ストラテジ・オプションを変更します。

   ```
   ALTER KEYSPACE system_auth WITH REPLICATION =
   { 'class' :'NetworkTopologyStrategy', 'dc1' :3, 'dc2' : 2};
   ```

   または、SimpleStrategyを使用する場合は以下のようになります。

   ```
   ALTER KEYSPACE "Excalibur" WITH REPLICATION =
   ```

```
{ 'class' :'SimpleStrategy', 'replication_factor' : 3 };
```

2. 影響を受ける各ノードで、nodetool repairコマンドを実行します。
3. ノードでのリペアが完了するまで待ち、次のノードに移動します。

テーブルの作成

このタスクについて

テーブルは、単一および複合プライマリ・キーを持つことができます。単一プライマリ・キーを持つテーブルを作成するには、PRIMARY KEYキーワードの後に、キーの名前を丸かっこで囲んで指定します。

手順

1. 前回の例のキースペースを作成して使用します(まだ行っていない場合)。
2. demodbキースペースに、ユーザー名(user_name)をプライマリー・キーにしてユーザー(users)テーブルを作成します。

```
CREATE TABLE users (
user_name varchar,
password varchar,
gender varchar,
session_token varchar,
state varchar,
birth_year bigint,
PRIMARY KEY (user_name));
```

複合プライマリ・キーの使用

このタスクについて

ソートされた結果を返すようにクエリーできるカラムを作成するには、複合プライマリ・キーを使用します。

手順

複合プライマリ・キーを持つテーブルを作成するには、複数のカラムをプライマリ・キーとして使用します。

```
CREATE TABLE emp (
empID int,
deptID int,
first_name varchar,
last_name varchar,
PRIMARY KEY (empID, deptID));
```

この例では、複合プライマリ・キーはempIDとdeptIDカラムで構成されています。empIDは、クラスターを構成するさまざまなノードにデータを分散させるためのパーティション・キーの役割をします。プライマリ・キーの他のコンポーネントであるdeptIDは、クラスター化メカニズムとして動作し、データを昇順でディスク上に格納します(Microsoft SQL Serverのクラスター化インデックスに似ています)。

テーブルへのデータの挿入

このタスクについて

実稼働データベースでのカラムとカラム値の挿入は、cqlshを使用するよりプログラムで行った方が実用的です。しかし、多くの場合、SQLに似たこのシェルを使用してクエリーをテストできると非常に便利です。

手順

INSERTコマンドを使用して、Jane Smithの社員データを挿入します。

```
INSERT INTO emp (empID, deptID, first_name, last_name)
VALUES (104, 15, 'jane', 'smith');
```

ユーザー定義型の使用

このタスクについて

Cassandra 2.1以降では、ユーザー定義型を作成して、複数のデータ・フィールドをカラムに付随させることができます。以下の例は、これらのタスクを実現する方法を示します。

- ユーザー定義型として住所（address）とフルネーム（fullname）を作成します。
- address型とfullname型のマップ・コレクションとセット・コレクションのカラムをそれぞれ定義するテーブルを作成します。
- 同様に、fullname型の名前（name）カラムをテーブルに含めます。
- addressカラムに番地、都市、郵便番号を挿入します。
- nameカラムに名と姓を挿入します。
- ユーザー定義型のカラムでデータをフィルターします。
- セット・コレクションに直属の部下の氏名を挿入します。

手順

1. キースペースを作成します。

```
CREATE KEYSPACE mykeyspace WITH REPLICATION =
 { 'class' :'NetworkTopologyStrategy', 'datacenter1' : 1 };
```

2. addressという名前のユーザー定義型を作成します。

```
CREATE TYPE mykeyspace.address (
street text,
city text,
zip_code int,
phones set<text>
);
```

3. ユーザーの氏名のユーザー定義型を作成します。

```
CREATE TYPE mykeyspace.fullname (
firstname text,
lastname text
);
```

4. ユーザー・データをfullnameおよびaddress型のカラムに格納するテーブルを作成します。ユーザー定義型カラムの定義にfrozenキーワードを使用します。

```
CREATE TABLE mykeyspace.users (
```

```
id uuid PRIMARY KEY,
name frozen <fullname>,
direct_reports set<frozen <fullname>>,      // コレクション・セット
addresses map<text, frozen <address>>      // コレクション・マップ
);
```

5. ユーザーの氏名をfullnameカラムに挿入します。

```
INSERT INTO mykeyspace.users (id, name) VALUES
 (62c36092-82a1-3a00-93d1-46196ee77204, {firstname:'Marie-Claude',
 lastname:'Josset'});
```

6. homeとラベル付けされた住所をテーブルに挿入します。

```
UPDATE mykeyspace.users SET addresses = addresses + {'home':{ street:'191
 Rue St. Charles', city:'Paris', zip_code:75015, phones:{'33 6 78 90 12
 34'}}} WHERE id=62c36092-82a1-3a00-93d1-46196ee77204;
```

7. ユーザーの氏名を取得します。

```
SELECT name FROM mykeyspace.users WHERE
 id=62c36092-82a1-3a00-93d1-46196ee77204;
```

ユーザー定義型のカラム名を使用すると、すべてのフィールド（firstnameとlastname）が取得されます。

```
name
------------------------------------------------
{firstname:'Marie-Claude', lastname:'Josset'}
```

ドット表記を使用すると、ユーザー定義型のカラムのコンポーネントを取得できます。以下の例では姓のみを取得しています。

```
SELECT name.lastname FROM mykeyspace.users WHERE
 id=62c36092-82a1-3a00-93d1-46196ee77204;

name.lastname
---------------
Josset
```

8. ユーザー定義型のカラムでデータをフィルターします。インデックスを作成し、条件付きクエリーを実行します。Cassandra 2.1.xでは、WHERE句内でnameカラムのコンポーネントをすべてリストする必要があります。

```
CREATE INDEX on mykeyspace.users (name);

SELECT id FROM mykeyspace.users WHERE name = {firstname:'Marie-Claude',
 lastname:'Josset'};
```

出力は以下のとおりです。

```
 id
--------------------------------------
62c36092-82a1-3a00-93d1-46196ee77204
```

9. Marie-Claudeの管理下にある人の氏名を挿入します。UPDATEコマンドを使用します。INSERTコマンドを使用して、別のマネージャーの管理下にある人の氏名を挿入します。

frozenキーワードを使用していると、ユーザー定義型の値を部分的に更新することはできません。値全体を上書きしなければなりません。Cassandraでは、ユーザー定義型であるfrozenの値はBLOBのように扱われます。

```
UPDATE mykeyspace.users SET direct_reports = { ( 'Naoko', 'Murai'),
 ( 'Sompom', 'Peh') } WHERE id=62c36092-82a1-3a00-93d1-46196ee77204;

INSERT INTO mykeyspace.users (id, direct_reports) VALUES
 ( 7db1a490-5878-11e2-bcfd-0800200c9a66, { ('Jeiranan', 'Thongnopneua') } );
```

10. Marie-Claudeの直属の部下をテーブルにクエリーします。

```
SELECT direct_reports FROM mykeyspace.users;
```

```
direct_reports
-------------------------------------------------------------------------------
{{firstname:'Jeiranan', lastname:'Thongnopneua'}}
{{firstname:'Naoko', lastname:'Murai'}, {firstname:'Sompom',
 lastname:'Peh'}}
```

システム・テーブルへのクエリー

systemキースペースには、Cassandraデータベース・オブジェクトやクラスター構成に関する詳細が含まれている多くのテーブルがあります。

Cassandraは、これらのテーブルおよびsystemキースペースの他のテーブルにデータを追加します。

表 **1 :** システム・テーブル内のカラム

テーブル名	カラム名	コメント
schema_keyspaces	keyspace_name、durable_writes、strategy_class、strategy_options	なし
local	"key", bootstrapped, cluster_name, cql_version, data_center, gossip_generation, native_protocol_version,partitioner, rack, release_version, ring_id, schema_version, thrift_version, tokens set, truncated at map	ノードが持つノード自体に関する情報とゴシップのスーパーセット。
peers	peer、data_center、rack、release_version、ring_id、rpc_address、schema_version、tokens set	各ノードはゴシップを通して他のノードが自身について示したことを記録します。
schema_columns	keyspace_name、columnfamily_name、column_name、component_index、index_name、index_options、index_type、validator	複合コラムとともに内部で使用されます。
schema_columnfamilies	コメントを参照。	特定のテーブルに関する詳細を参照する場合にschema_columnfamiliesを調べます。

キースペース、テーブル、およびカラムの情報

このタスクについて

Thrift APIのdescribe_keyspaces関数に代わる手段として、system.schema_keyspacesに直接クエリーするという方法があります。また、テーブルに関する情報を取得するにはsystem.schema_columnfamiliesにクエリーし、カラム・メタデータに関する情報を取得するにはsystem.schema_columnsにクエリーします。

手順

SELECT文を使用して、定義されているキースペースをクエリーします。

```
SELECT * from system.schema_keyspaces;
```

cqlshの出力には、定義されているキースペースに関する情報が含まれます。

```
 keyspace | durable_writes | name     | strategy_class | strategy_options
```

```
----------+----------------+---------+----------------
+---------------------------
history |           True | history | SimpleStrategy |
 {"replication_factor":"1"}
ks_info |           True | ks_info | SimpleStrategy |
 {"replication_factor":"1"}

(2 rows)
```

クラスター情報

このタスクについて

システム・テーブルにクエリーすることで、クラスターのトポロジー情報を取得することができます。ピア・ノードのIPアドレス、データ・センターとラック名、トークン値、およびその他の情報を取得できます。「*The Data Dictionary*」の記事では、システム・テーブルのクエリーについて詳しく説明しています。

手順

Mac OSXでccmを使用して3ノード・クラスターを設定したら、peersおよびローカル・テーブルにクエリーします。

```
USE system;
SELECT * FROM peers;
```

peersテーブルへのクエリーによる出力は、以下のように表示されます。

```
   peer     | data_center | host_id      | preferred_ip | rack  |
 release_version | rpc_address | schema_version | tokens
-----------+-------------+-------------+--------------+-------
+-----------------+-------------+----------------+-----------
127.0.0.3 | datacenter1 | edda8d72...|          null | rack1 |
 2.1.0 | 127.0.0.3    | 59adb24e-f3... |  {3074...
127.0.0.2 | datacenter1 | ef863afa...|          null | rack1 |
 2.1.0 | 127.0.0.2    | 3d19cd8f-c9... | {-3074...}

(2 rows)
```

結果の取得とソート

このタスクについて

結果を取得するには、SELECTコマンドを使用します。

```
SELECT * FROM users WHERE first_name = 'jane' and last_name='smith';
```

SQLクエリーと同様に、Cassandra 2.0.xでは、以下の例に示すように、WHERE句、ORDER BY句の順に使用して、結果を取得してソートします。Cassandra 2.1でこのクエリーを実行するには、ドライバー・レベルでページングを無効にする必要があります。Javaでは、フェッチサイズをInteger.MAX_VALUEに設定します。

手順

1. 結果を取得し降順でソートします。

```
cqlsh:demodb> SELECT * FROM emp WHERE empID IN (130,104) ORDER BY deptID
 DESC;
```

```
 empid | deptid | first_name | last_name
```

```
-------+--------+------------+-----------
104 |     15 |       jane |     smith
130 |      5 |     sughit |     singh

(2 rows)
```

2. 結果を取得し昇順でソートします。

```
cqlsh:demodb> SELECT * FROM emp where empID IN (130,104) ORDER BY deptID
 ASC;
```

```
 empid | deptid | first_name | last_name

-------+--------+------------+-----------
130 |      5 |     sughit |     singh
104 |     15 |       jane |     smith
```

音楽サービスの例は、複合プライマリ・キーを使用した結果の取得およびソート方法を示しています。

パーティションの行のスライス取得

Cassandra 2.0.6以降では、テーブルに複数のクラスター化カラムがある場合、パーティションの行のスライスを取得する新しい構文を使用できます。条件演算子を使用して、クラスター化キーのグループを特定の値と比較できます。以下に例を示します。

```
CREATE TABLE timeline (
day text,
hour int,
min int,
sec int,
value text,
PRIMARY KEY (day, hour, min, sec)
   );

INSERT INTO timeline (day, hour, min, sec, value)
VALUES ('12 Jan 2014', 3, 43, 12, 'event1');
INSERT INTO timeline (day, hour, min, sec, value)
VALUES ('12 Jan 2014', 3, 52, 58, 'event2');
INSERT INTO timeline (day, hour, min, sec, value)
VALUES ('12 Jan 2014', 4, 37, 01, 'event3');
INSERT INTO timeline (day, hour, min, sec, value)
VALUES ('12 Jan 2014', 4, 37, 41, 'event3');
INSERT INTO timeline (day, hour, min, sec, value)
VALUES ('12 Jan 2014', 6, 00, 34, 'event4');

SELECT * FROM timeline;
```

```
day          | hour | min | sec | value
   -------------+------+-----+-----+--------
12 Jan 2014 |    3 |  43 |  12 | event1
12 Jan 2014 |    3 |  52 |  58 | event2
12 Jan 2014 |    4 |  37 |   1 | event3
12 Jan 2014 |    4 |  37 |  41 | event3
12 Jan 2014 |    6 |   0 |  34 | event4
```

2014年1月12日の3:50:00～4:37:30のイベントを取得するには、以下のように新しい構文を使用します。

```
SELECT * FROM timeline WHERE day='12 Jan 2014'
AND (hour, min) >= (3, 50)
AND (hour, min, sec) <= (4, 37, 30);
```

```
day          | hour | min | sec | value
   -------------+------+-----+-----+--------
12 Jan 2014 |    3 |  52 |  58 | event2
12 Jan 2014 |    4 |  37 |   1 | event3
```

この例の新しい構文では、条件演算子を使用して、hour、min、secなどのクラスター化キーのグループを特定の値と比較しています。

WHERE句では、連続するクラスター化カラムを使用する必要があります。この順序はテーブル定義のカラムの順序と一致していなければなりません。例:

```
CREATE TABLE no_column_skipping
(a int, b int, c int, d int, e int,
PRIMARY KEY (a, b, c, d))
```

以下のWHERE句は機能しません。

```
SELECT ...WHERE a=0 AND (b, d) > (1, 2)
```

以下のWHERE句は機能します。

```
SELECT ...WHERE a=0 AND (b, c) > (1, 2)
```

静的(STATIC)カラムに対する条件付き更新のバッチ処理

BATCH文リファレンスで説明しているように、Cassandra 2.0.6以降では、条件付き更新をバッチ処理できます。以下の例は、静的カラム(これもCassandra 2.0.6で導入されました)の使用を組み合わせた条件付き更新のバッチ処理を示しています。この例では、ユーザーごとに、各購入に関するレコードを格納し、ユーザーの全購入の現時点までの残高を含めます。

```
CREATE TABLE purchases (
user text,
balance int static,
expense_id int,
amount int,
description text,
paid boolean,
PRIMARY KEY (user, expense_id)
  );
```

balanceが静的(STATIC)であるため、ユーザーのすべての購入レコードが同じ残高になります。

購入レコードに値を挿入するための文でIF条件句を使用しています。

```
BEGIN BATCH
INSERT INTO purchases (user, balance) VALUES ('user1', -8) IF NOT EXISTS;
INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;

BEGIN BATCH
UPDATE purchases SET balance = -208 WHERE user='user1' IF balance = -8;
INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 2, 200, 'hotel room', false);
APPLY BATCH;
```

カラムが静的(STATIC)であるため、データの更新時はパーティション・キーのみを指定できます。静的ではないカラムを更新する場合は、クラスター化キーも指定する必要があります。バッチ処理される条件付き更新を使用すると、

現時点までの残高を維持できます。条件付き更新があるバッチは複数のパーティションにまたがることができないため、残高を別個のテーブルに格納すると現時点までの残高を維持できなくなります。

```
SELECT * FROM purchases;
```

この時点の出力は以下のとおりです。

```
 user  | expense_id | balance | amount | description | paid
-------+------------+---------+--------+-------------+-------
user1 |          1 |    -208 |      8 |     burrito | False
user1 |          2 |    -208 |    200 |  hotel room | False
```

次に、条件付きバッチを使用して、残高をクリアするようにレコードを更新することもできます。

```
BEGIN BATCH
UPDATE purchases SET balance=-200 WHERE user='user1' IF balance=-208;
UPDATE purchases SET paid=true WHERE user='user1' AND expense_id=1 IF
 paid=false;
APPLY BATCH;

SELECT * FROM purchases;
```

```
user  | expense_id | balance | amount | description | paid
-------+------------+---------+--------+-------------+-------
user1 |          1 |    -200 |      8 |     burrito | True
user1 |          2 |    -200 |    200 |  hotel room | False
```

バッチの使用と誤使用

バッチは、パフォーマンスを最適化することを意図して誤って使用されることがよくあります。ログを記録しないバッチは、コーディネーターが挿入を管理する必要があります。このために、コーディネーター・ノードに大きな負荷がかかる場合があります。他のノードがパーティション・キーを持っている場合、コーディネーター・ノードはネットワークを経由する必要があるため、結果として受け渡しが非効率になります。ログを記録しないバッチは、同じパーティション・キーに対して更新を行う場合に使用してください。

たとえば、(date, timestamp)のプライマリ・キーを使用したとすると、ログを記録しない以下のバッチは、書き込みの数に関係なく、内部的に1つの書き込みに読み替えられます（すべてが同じ日付値を持っている場合）。

```
BEGIN UNLOGGED BATCH;
INSERT INTO sensor_readings (date, timestamp, reading) values
 (20140910,'2014-09-10T11:00:00.00+0000', 6335.2);
INSERT INTO sensor_readings (date, timestamp, reading) values
 (20140910,'2014-09-10T11:00:15.00+0000', 5222.2);
APPLY BATCH;
```

コーディネーター・ノードは、テーブル間の整合性を維持しながらログを記録するバッチ処理の場合も、負担が大きくなることがあります。たとえば、コーディネーター・ノードは、バッチを受け取るとバッチのログを他の2つのノードに送信します。コーディネーターに障害が発生した場合は、他方のノードがバッチを再試行します。クラスター全体が影響を受けます。以下の例に示すように、ログを記録するバッチは、テーブルを同期化する場合に使用します。

```
BEGIN BATCH;
UPDATE songs SET tags = tags + {'2007'}
WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;
UPDATE songs SET tags = tags + {'covers'}
WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;
APPLY BATCH;
```

データを読み込む最も速い方法については、「Cassandra: Batch loading without the Batch keyword」（英語）を参照してください。

キースペース修飾子の使用

このタスクについて

キースペースを選択するためにUSE文を発行することは不便な場合があります。たとえば、接続プールを使用する場合は、複数のキースペースを裁く必要があります。複数のキースペースの追跡を簡略化するには、USE文の代わりにキースペース修飾子を使用します。以下の文でキースペース修飾子を使用してキースペースを指定できます。

- ALTER TABLE
- CREATE TABLE
- DELETE
- INSERT
- SELECT
- TRUNCATE
- UPDATE

手順

操作対象のテーブルが含まれていないキースペースにいるときに対象テーブルを指定するには、キースペースの名前に続き、ピリオド、テーブル名の順に指定します。以下にMusic.playlistsの例を示します。

```
INSERT INTO Music.playlists (id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 'La Grange', 'ZZ Top', 'Tres
 Hombres');
```

テーブルへのカラムの追加

このタスクについて

ALTER TABLEコマンドは、新しいカラムをテーブルに追加します。

手順

varcharデータ型のcoupon_codeカラムをusersテーブルに追加します。

```
cqlsh:demodb> ALTER TABLE users ADD coupon_code varchar;
```

これは、カラム・メタデータを作成し、カラムをテーブル・スキーマに追加しますが、既存の行を更新しません。

カラムの期限切れについて

カラムのデータには、TTL（Time to live）という有効期限を任意で持たせることができます。クライアント要求でデータに秒単位で定義したTTL値を指定します。要求された時間が経過すると、TTLデータにトゥームストーンのマークが付きます。データにトゥームストーンのマークが付くと、このデータはgc_grace_secondsおよびリペア・プロセスで定義された通常のコンパクション中に自動的に削除されます。

CQLを使用することで、データのTTLを設定できます。

期限切れになるデータのTTLを変更する場合は、TTLを変更したデータを再度挿入する必要があります。Cassandraでのデータの挿入とは、以前のデータが存在するかどうかに応じて、挿入または更新の操作になります。

TTLデータは、サーバーで計算される秒単位の精度があります。したがって、あまり小さいTTLは意味がない場合があります。さらに、サーバーのクロックを同期させる必要があります。有効期限は、最初の挿入を受け取るプライマリ・ホストで計算されますが、その後はクラスター上の他のホストにより解釈されるため、クロックを同期させないと、精度が低下する可能性があります。

期限切れになるデータには、標準データと比較して8バイトのオーバーヘッドがメモリー内とディスク上に追加されます（TTLと有効期限の記録のため）。

カラムの**time-to-live**の特定

このタスクについて

この手順では、テーブルを作成し、データを2つのカラムに挿入して、カラムへの書き込み日時を取得するTTL関数を呼び出します。

手順

1. excelsiorキースペースにclicksという名前のユーザー・テーブルを作成します。

```
CREATE TABLE excelsior.clicks (
userid uuid,
url text,
date timestamp,  //次節で説明するWRITETIMEとは無関係
name text,
PRIMARY KEY (userid, url)
);
```

2. yyyy-mm-dd形式の日付などのデータをテーブルに挿入し、そのデータの有効期限を1日（86400秒）に設定します。有効期限はUSING TTL句を使用して設定します。

```
INSERT INTO excelsior.clicks (
userid, url, date, name)
VALUES (
3715e600-2eb0-11e2-81c1-0800200c9a66,
'http://apache.org',
'2013-10-09', 'Mary')
USING TTL 86400;
```

3. しばらく待った後にSELECT文を発行し、手順2で入力したデータの期限がこの後どのくらい残っているかを確認します。

```
SELECT TTL (name) from excelsior.clicks
WHERE url = 'http://apache.org' ALLOW FILTERING;
```

たとえば、以下のように85908秒と出力されます。

```
 ttl(name)
-----------
 85908

(1 rows)
```

データの削除

データを削除するには、TTL（time-to-expire）テーブル属性を使用して、カラム値を自動削除に設定します。

テーブルまたはキースペースを削除したり、キースペース・カラム・メタデータを削除したりすることもできます。

カラムの期限切れ

このタスクについて

INSERTおよびUPDATEコマンドは、カラム内のデータが期限切れになる時間の設定をサポートしています。有効期限（TTL）はCQLを使用して設定します。

手順

1. INSERTコマンドを使用して、86400秒、つまり1日で期限切れになるようにusersテーブルのpasswordカラムを設定します。

```
cqlsh:demodb> INSERT INTO users
(user_name, password)
VALUES ('cbrown', 'ch@ngem4a') USING TTL 86400;
```

2. UPDATEコマンドを使用して、有効期限を5日間に延長します。

```
cqlsh:demodb> UPDATE users USING TTL 432000 SET password = 'ch@ngem4a'
WHERE user_name = 'cbrown';
```

テーブルまたはキースペースの削除

このタスクについて

DROPコマンドを使用してテーブルまたはキースペースを削除します。

手順

1. usersテーブルを削除します。

```
cqlsh:demodb> DROP TABLE users;
```

2. demodbキースペースを削除します。

```
cqlsh:demodb> DROP KEYSPACE demodb;
```

カラムと行の削除

このタスクについて

CQLには、カラムまたは行を削除するためのDELETEコマンドが用意されています。削除した値は、削除の後の最初のコンパクションによって完全に除去されます。

手順

1. ユーザーjsmithのセッション・トークン・カラムを削除します。

```
cqlsh:demodb> DELETE session_token FROM users where pk = 'jsmith';
```

2. jsmithの行全体を削除します。

```
cqlsh:demodb> DELETE FROM users where pk = 'jsmith';
```

書き込み日時の特定

このタスクについて

テーブルには、カラムへの書き込みが発生した日時を示すタイムスタンプが含まれています。SELECT文でWRITETIME関数を使用すると、カラムがデータベースに書き込まれた日時がミリ秒単位で返されます。この手順では、前の手順の例を引き続き使用し、カラムへの書き込み日時を取得するWRITETIME関数を呼び出します。

手順

1. テーブルに追加データを挿入します。

```
INSERT INTO excelsior.clicks (
userid, url, date, name)
VALUES (
cfd66ccc-d857-4e90-b1e5-df98a3d40cd6,
'http://google.com',
'2013-10-11', 'Bob'
);
```

2. 値Maryがapache.orgデータのnameカラムに書き込まれた日時を取得します。SELECT文で、WRITETIME関数に続き、カラムの名前をかっこで囲んで指定します。

```
SELECT WRITETIME (name) FROM excelsior.clicks
WHERE url = 'http://apache.org' ALLOW FILTERING;
```

マイクロ秒単位のWRITETIMEの出力を変換すると、「Sun, 14 Jul 2013 21:57:58 GMT」または「2:57 pm Pacific time」になります。

```
writetime(name)
------------------
1373839078327001
```

3. google.comデータのdateカラムへの前回の書き込み日時を取得します。

```
SELECT WRITETIME (date) FROM excelsior.clicks
WHERE url = 'http://google.com' ALLOW FILTERING;
```

マイクロ秒単位のWRITETIMEの出力を変換すると、「Sun, 14 Jul 2013 22:03:15 GMT」または「3:03 pm Pacific time」になります。

```
writetime(date)
------------------
1373839395324001

(1 rows)
```

カラムのデータ型の変更

このタスクについて

ALTER TABLEを使用すると、カラムを定義した後、またはカラムをテーブルに追加した後に、そのカラムのデータ型を変更できます。

手順

カラムのデータ型を変更することで、クーポン・コードをテキストではなく整数として格納するようにcoupon_codeカラムを変更します。

```
cqlsh:demodb> ALTER TABLE users ALTER coupon_code TYPE int;
```

既存のクーポン・コードではなく、新たに挿入された値のみが新しい型に対して検証されます。

コレクションの使用

Cassandraには、複数の電子メール・アドレスを扱う機能をテーブルに組み込むなどの作業を改善する手段を提供するコレクション型が含まれています。以下のコレクションの制限事項を確認してください。

- コレクション内の項目の最大サイズは64Kです。
- Cassandraではコレクション全体が読み出されるため、クエリー中の遅延を防ぐためにコレクションは小さくしておきましょう。コレクションは内部でページングされません。

  前に説明したように、コレクションは少量のデータのみを格納することを目的として設計されています。
- コレクションには64Kを超える項目を挿入しないでください。

  64Kを超える項目をコレクションに挿入すると、項目のうち64Kだけがクエリー可能になり、残りのデータは失われます。

time-to-live（TTL）プロパティを個別に設定することで、コレクションの各要素の有効期限を設定できます。

セット型の使用

このタスクについて

セットは、クエリー時にソートされた順序で返される要素のグループを格納します。セット型のカラムは、順序指定されていない固有の値で構成されています。セット・データ型を使用すると、新しい電子メール・アドレスの追加に先だって読み取りを必要としない直観的な方法で、複数電子メールの問題を解決できます。

手順

1. emailsというセットをusersテーブルに定義し、複数の電子メール・アドレスを格納できるようにします。

   ```
   CREATE TABLE users (
   user_id text PRIMARY KEY,
   first_name text,
   last_name text,
   emails set<text>
   );
   ```

2. 値を中かっこで囲んでデータをセットに挿入します。

   セットの各値は一意である必要があります。

```
INSERT INTO users (user_id, first_name, last_name, emails)
VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com', 'baggins@gmail.com'});
```

3. UPDATEコマンドと加算(+)演算子を使用して、1つの要素をセットに追加します。

```
UPDATE users
SET emails = emails + {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

4. frodoの電子メール・アドレスをセットから取得します。

```
SELECT user_id, emails FROM users WHERE user_id = 'frodo';
```

コレクションを含むテーブルにクエリーすると、コレクション全体が取得されます。したがって、コレクションを管理可能な大きさになるように小さくしておくか、または大量のデータを収容できるコレクションに代わるデータ・モデルを構築してください。

Cassandraは、コレクション内の要素の種類に基づいた順序で結果を返します。たとえば、テキスト要素のセットはアルファベット順に返されます。コレクションの要素が挿入順に返されるようにするには、リストを使用します。

```
user_id | emails
---------
+-----------------------------------------------------------------
frodo   | {"baggins@caramail.com","f@baggins.com","fb@friendsofmordor.org"}
```

5. 減算(-)演算子を使用して、セットから1つの要素を削除します。

```
UPDATE users
SET emails = emails - {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

6. UPDATEまたはDELETE文を使用して、セットからすべての要素を削除します。

セット、リスト、またはマップには、少なくとも1つの要素を持たせる必要があります。要素がないと、セットとNull値が区別されません。

```
UPDATE users SET emails = {} WHERE user_id = 'frodo';

DELETE emails FROM users WHERE user_id = 'frodo';
```

電子メールを検索するクエリーがNullを返します。

```
SELECT user_id, emails FROM users WHERE user_id = 'frodo';
```

```
user_id | emails
---------+-------------------------------------------------
frodo   | null

(1 rows)
```

リスト型の使用

このタスクについて

要素の順序が重要であり、それが要素のタイプによって決まる自然な順序ではない場合は、リストを使用します。また、同じ値を何度も格納する必要がある場合もリストを使用します。セットの値は(値がテキストの場合)アルファベット順で返されるのに対して、リスト値はリスト内のインデックス値に基づいて返されます。

リスト型を使用して、usersテーブルのユーザーごとに優先順位のリストを追加し、上位x位までのユーザーについてデータベースにクエリーできます。

手順

1. リスト型のカラムtop_placesをusersテーブルに追加することで、リスト宣言をテーブルに追加します。

```
ALTER TABLE users ADD top_places list<text>;
```

2. UPDATEコマンドを使用して、値をリストに挿入します。

```
UPDATE users
SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';
```

3. 要素を大かっこで囲み、加算(+)演算子を使用して、リストの先頭に1つの要素を追加します。

```
UPDATE users
SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';
```

4. UPDATEコマンドの中で、新しい要素とリスト名の順序を入れ替えて、リストに1つの要素を追加します。

```
UPDATE users
SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';
```

これらの更新操作は、内部的には、書き込みに先立つ読み取りのない操作として実装されています。リストの先頭および末尾に新しい要素を追加すると、新しい要素のみが書き込まれます。

5. 大かっこ内にリスト・インデックス位置を指定して、特定の位置に1つの要素を追加します。

```
UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';
```

要素を特定の位置に追加する場合は、リスト全体が読み取られた後に、更新された要素のみが書き込まれます。その結果として、特定の位置への要素の追加は、リストの末尾または先頭に要素を追加する場合に比べてレイテンシーが長くなります。

6. DELETEコマンドと、大かっこ内にリスト・インデックス位置を指定して、リストから要素を削除します。たとえば、mordorを削除しても、shire、rivendell、およびriddermarkはそのまま残ります。

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';
```

7. UPDATEコマンド、減算演算子(-)、大かっこで囲んだリスト値を使用して、特定の値を持つすべての要素を削除します。たとえば、riddermarkを削除します。

```
UPDATE users
SET top_places = top_places - ['riddermark'] WHERE user_id = 'frodo';
```

前に示した、リストから要素を削除するインデックスを指定する方法には、内部的には読み取りが必要です。ここで示すUPDATEコマンドの使用法は、クライアント側でこの操作をエミュレートして、リスト全体を読み込み、削除する値を含む要素のインデックスを見つけたら、そのインデックスの要素を削除するという方法よりも推奨されます。そのようなエミュレーションは、スレッド・セーフではないからです。読み取りと書き込みをしている間に、別のスレッド/クライアントがリストの先頭に要素を追加すると、誤った要素が削除されてしまいます。ここに示したUPDATEコマンドを使用すれば、このような問題は発生しません。

8. 上位のリストについてデータベースにクエリーします。

```
SELECT user_id, top_places FROM users WHERE user_id = 'frodo';
```

以下の結果が得られます。

```
 user_id | top_places
---------+----------------------------
frodo | ['the shire', 'rivendell']
```

マップ型の使用

このタスクについて

名前からわかるように、マップとは、あるものを別のものにマッピングします。マップは名前であり、型を持つ値のペアです。マップ型を使用して、タイムスタンプに関連付けられた情報をユーザーのプロファイルに格納できます。マップの各要素は、変更、置換、削除、およびクエリーできる1つのCassandraカラムとして内部で格納されます。各要素に個別のtime-to-live（TTL）を持たせて、TTLが終了するとその要素も期限切れとなるように設定できます。

手順

1. todoリストを既存のusersテーブル内の全ユーザー・プロファイルに追加します。これを行うには、CREATE TABLEまたはALTER文を使用し、マップ・コレクションを指定して、データ型のペアを山かっこで囲みます。

```
ALTER TABLE users ADD todo map<timestamp, text>;
```

2. マップ・データを設定または置き換えます。これを行うには、INSERTまたはUPDATEコマンドを使用し、マップ・コレクションのタイムスタンプおよびテキスト値をコロンで区切ったものを中かっこで囲みます。

```
UPDATE users
SET todo =
{ '2012-9-24' :'enter mordor',
'2014-10-2 12:00' :'throw ring into mount doom' }
WHERE userid = 'frodo';
```

3. 特定の要素を設定します。これを行うには、UPDATEコマンドを使用し、要素のキーであるタイムスタンプを大かっこで囲み、等価演算子を使用して値をそのタイムスタンプにマップします。

```
UPDATE users SET todo['2014-10-2 12:00'] = 'throw my precious into mount
 doom'
WHERE userid = 'frodo';
```

4. INSERTを使用して、マップ・コレクションのデータを指定します。

```
INSERT INTO users (userid, todo) VALUES ('frodo', { '2013-9-22
 12:01' :'birthday wishes to Bilbo', '2013-10-1 18:00':'Check into Inn of
 Pracing Pony'}) ;
```

このデータをマップに挿入すると、マップ全体が置換されます。

5. DELETEコマンドを使用し、要素のキーであるタイムスタンプを大かっこで囲むことで、マップから要素を削除します。

```
DELETE todo['2012-9-24'] FROM users WHERE userid = 'frodo';
```

6. todoマップを取得します。

```
SELECT user_id, todo FROM users WHERE userid = 'frodo';
```

マップの出力の順序は、マップの型によって異なります。

7. todoリスト要素がタイムスタンプの日付に期限切れとなるようにするためのTTLを計算し、要素が期限切れになるように設定します。

```
UPDATE users USING TTL <算出したTTL>
SET todo['2012-10-1'] = 'find water' WHERE userid = 'frodo';
```

カラムのインデックスの作成

このタスクについて

cqlshを使用して、カラム値にインデックスを作成できます。Cassandra 2.1以降では、コレクション・カラムのインデックスを作成できます。インデックスの作成はパフォーマンスに大きな影響を与える可能性があります。インデックスを作成する前に、インデックスを作成すべき場合とすべきでない場合について把握しておいてください。

手順

1. usersテーブルのstateおよびbirth_yearカラムにインデックスを作成します。

```
cqlsh:demodb> CREATE INDEX state_key ON users (state);
cqlsh:demodb> CREATE INDEX birth_year_key ON users (birth_year);
```

2. インデックスを作成したカラムにクエリーします。

```
cqlsh:demodb> SELECT * FROM users
WHERE gender = 'f' AND
state = 'TX' AND
birth_year > 1968
ALLOW FILTERING;
```

軽量トランザクションの使用

このタスクについて

IF句を使用するINSERTおよびUPDATE文は、Compare and Set（CAS）とも呼ばれる軽量トランザクションをサポートしています。

手順

1. 新しいユーザーを登録します。

```
INSERT INTO users (login, email, name, login_count)
VALUES ('jdoe', 'jdoe@abc.com', 'Jane Doe', 1)
IF NOT EXISTS
```

2. クエリーの最後に操作の述語を追加することで、存在する行に対してCAS操作を実行します。たとえば、Jane Doeのパスワードをリセットします。

```
UPDATE users
SET email = ‘janedoe@abc.com’
WHERE login = 'jdoe'
IF email = ‘jdoe@abc.com’
```

順序指定されていないパーティショナー結果全体のページング

RandomPartitionerまたはMurmur3Partitionerを使用する場合、Cassandraの行はそれらの値のハッシュで順序指定されるため、行の順序は意味がありません。CQLを使用すると、以下の例に示すように、TOKEN関数を使用することでランダム・パーティショナーまたはmurmur3パーティショナーを使用する場合であっても行をページングできます。

```
SELECT * FROM test WHERE token(k) > token(42);
```

ByteOrderedパーティショナーは、トークンをキー値と同じ方法で配置しますが、RandomPartitionerおよびMurmur3Partitionerは、順不同でトークンを配置します。TOKEN関数を使用すると、このような順序指定されていないパーティショナー結果をページングできます。TOKEN関数により、トークンを直接使用して結果にクエリーできます。実際は、TOKEN関数はトークンベースの比較を行い、キーをトークンに変換するわけではありません（すなわち、k > 42の比較ではありません）。

TOKEN関数を使用して、パーティション・キー・カラムを対象に条件付き関係を表すことができます。この場合、クエリーは、値ではなく、パーティション・キーのトークンに基づいて行を返します。

カウンターの使用

このタスクについて

カウンターは、インクリメントによって変化する数値の格納に使用される特別なカラムです。たとえば、ページが表示された回数のカウントにカウンター・カラムを使用することがあります。

Cassandra 2.1のカウンター・カラムは、カウンターの実装を向上させ、カウンターを調整するためのいくつかの構成オプションを備えています。Cassandra 2.1以降では、カウンターの書き込みの完了までのコーディネーターの待ち時間、メモリー内のカウンター・キャッシュのサイズ、カウンター・キャッシュ・キーを保存するまでのCassandraの待ち時間、保存するキーの数、およびconcurrent_counter_writesを構成できます。これらのオプションは、`cassandra.yaml`ファイルに設定します。Cassandra 2.0.xカウンター実装で使用されるreplicate_on_writeテーブル・プロパティは、Cassandra 2.1から除去されました。

カウンターは、専用テーブルにのみ定義し、カウンター・データ型を使用してください。カウンター・カラムのインデックス作成、削除、または再追加はできません。

データをカウンター・カラムにロードしたり、またはカウンターの値を増減したりするには、UPDATEコマンドを使用します。カウンター・カラムを更新するコマンドでは、USING TIMESTAMPまたはUSING TTLは拒否されます。

手順

1. 1つのデータ・センター、つまり1つのノード・クラスター用にLinuxでキースペースを作成します。`nodetool status`コマンドの出力からデフォルトのデータ・センター名（以下の例ではdatacenter1）を使用します。

```
CREATE KEYSPACE counterks WITH REPLICATION =
{ 'class' :'NetworkTopologyStrategy', 'datacenter1' : 1 };
```

2. カウンター・カラムのテーブルを作成します。

```
CREATE TABLE counterks.page_view_counts
(counter_value counter,
url_name varchar,
page_name varchar,
PRIMARY KEY (url_name, page_name)
);
```

3. データをカウンター・カラムにロードします。

```
UPDATE counterks.page_view_counts
SET counter_value = counter_value + 1
WHERE url_name='www.datastax.com' AND page_name='home';
```

4. カウンター値を見てみましょう。

```
SELECT * FROM counterks.page_view_counts;
```

出力は以下のとおりです。

```
 url_name          | page_name | counter_value
-------------------+-----------+---------------
www.datastax.com |      home |             1

(1 rows)
```

5. カウンターの値を大きくします。

```
UPDATE counterks.page_view_counts
SET counter_value = counter_value + 2
WHERE url_name='www.datastax.com' AND page_name='home';
```

6. カウンター値を見てみましょう。

```
 url_name          | page_name | counter_value
-------------------+-----------+---------------
www.datastax.com |      home |             3
```

整合性の変化のトレース

Cassandraなどの分散システムでは、データの最新値が必ずしもすべてのノードに存在するわけではありません。クライアント・アプリケーションは、応答時間とデータの正確さとの兼ね合いを管理するために要求ごとに整合性レベルを構成します。以下のチュートリアルでは、5ノード・クラスターでの動作をトレースすることで、これらの整合性レベルの違いと、要求に応えるために関与するレプリカの数を示します。

- ONE

  最も近いレプリカからデータを返します。
- QUORUM

  過半数のレプリカから最新のデータを返します。
- ALL

  すべてのレプリカから最新のデータを返します。

以下の手順に従ってローカル・コンピューターに5つのノードをセットアップし、異なる整合性レベルで読み取りをトレースして、結果を比較してみましょう。

整合性の変化をトレースするためのセットアップ

このタスクについて

ローカル・コンピューターに5つのノードをセットアップし、異なる整合性レベルで読み取りをトレースして、結果を比較するには、以下のようにします。

手順

1. githubからccmスクリプト・ライブラリを取得します。

   このライブラリを使用して、以下のアクションを実行します。

   - Apache Cassandraソース・コードのダウンロード。
   - 1台のコンピューター上でのApache Cassandraクラスターの作成と起動。

   前提条件については、ccmのREADMEを参照してください。
2. 以下のコマンドをコマンドラインで入力し、ローカルIPでエイリアスをセットアップします。

```
$ sudo ifconfig lo0 alias 127.0.0.2 up
```

```
$ sudo ifconfig lo0 alias 127.0.0.3 up
$ sudo ifconfig lo0 alias 127.0.0.4 up
$ sudo ifconfig lo0 alias 127.0.0.5 up
```

3. Apache Cassandraソース・コード(以下の例ではバージョン2.0.6)を/.ccm/repositoryにダウンロードし、trace_consistencyという名前のccmクラスターを起動します。

```
$ ccm create trace_consistency -v 2.0.6
```

```
Downloading http://archive.apache.org/dist/cassandra/2.0.6/
apache-cassandra-2.0.6-src.tar.gz to /var/folders/9k/
ywsprd8n14s7hzb5qnztgb5h0000gn/T/ccm-lnfEFA.tar.gz (10.830MB)
  11356108  [100.00%]
Extracting /var/folders/9k/ywsprd8n14s7hzb5qnztgb5h0000gn/T/ccm-
lnfEFA.tar.gz as version 2.0.6 ...
Compiling Cassandra 2.0.6 ...
Current cluster is now:trace_consistency
```

4. cassandra.yamlを開き、num_tokensを0に設定することで、/.ccm/repository/2.0.6/confディレクトリーで、vnodesの使用を禁止します。

   構成の詳細については、Cassandra 2.1 cassandra.yamlまたはCassandra 2.0 cassandra.yamlファイルを参照してください。

5. 以下のコマンドを使用して、構成ファイルを更新します。

```
ccm updateconf
```

6. 以下のコマンドを使用して、クラスターを生成して確認します。

```
$ ccm populate -n 5
$ ccm start
```

7. クラスターが起動していることを確認します。

```
$ ccm node1 ring
```

   出力には5つすべてのノードのステータスが示されます。

関連情報
Cassandra 2.0 cassandra.yaml
Cassandra 2.1 cassandra.yaml

異なる整合性レベルでの読み取りのトレース

このタスクについて

セットアップ手順を実行したら、データを異なる整合性レベルで読み取るクエリーを実行してトレースします。トレーシング出力は、5ノード・クラスターで3つのレプリカを使用すると、整合性レベルONEは3つのレプリカのうち1つからの応答を処理し、QUORUMは3つのレプリカのうち2つからの応答を処理し、ALLは3つのレプリカすべてからの応答を処理することを示します。

手順

1. cqlshをリング内の最初のノードに接続します。

```
$ ccm node1 cqlsh
```

2. cqlshコマンドラインで、クラスター内のデータ分散に3つのレプリカの使用を指定するキースペースを作成します。

```
cqlsh> CREATE KEYSPACE demo_cl WITH replication = {'class':'SimpleStrategy',
 'replication_factor':3};
```

3. 3つのレプリカを持つキースペースで、テーブルを作成し、いくつかの値を挿入します。

```
cqlsh> USE demo_cl;
```

```
cqlsh:demo_cl> CREATE TABLE demo_table ( id int PRIMARY KEY, col1 int, col2
 int );
cqlsh:demo_cl> INSERT INTO demo_table (id, col1, col2) VALUES (0, 0, 0);
```

4. トレーシングをオンにし、CONSISTENCYコマンドを使用して、整合性レベルがONE（デフォルト）であることを確認します。

```
cqlsh:demo_cl> TRACING on;
cqlsh:demo_cl> CONSISTENCY;
```

出力は以下のとおりです。

```
Current consistency level is ONE.
```

5. テーブルにクエリーして、プライマリ・キーの値を読み取ります。

```
cqlsh:demo_cl> SELECT * FROM demo_table WHERE id = 0;
```

出力にはトレーシング情報が含まれます。

```
id | col1 | col2
---+------+------
 0 |    0 |    0

(1 rows)

Tracing session:230a0d10-5576-11e3-add5-a180b5ee3938

activity                                                                  |
 timestamp    | source    | source_elapsed
---------------------------------------------------------------------------
+-------------+-----------+----------------
execute_cql3_query | 12:13:49,513 | 127.0.0.1 |              0
Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000; | 12:13:49,514 |
 127.0.0.1 |            540
Preparing statement | 12:13:49,515 | 127.0.0.1 |           1206
Sending message to /127.0.0.3 | 12:13:49,515 | 127.0.0.1 |           1768
Sending message to /127.0.0.4 | 12:13:49,515 | 127.0.0.1 |           1830
Message received from /127.0.0.1 | 12:13:49,517 | 127.0.0.3 |             61
Message received from /127.0.0.1 | 12:13:49,517 | 127.0.0.4 |             66
Executing single-partition query on demo_table | 12:13:49,518 | 127.0.0.3 |
           793
Executing single-partition query on demo_table | 12:13:49,518 | 127.0.0.4 |
           914
Acquiring sstable references | 12:13:49,518 | 127.0.0.3 |            823
Acquiring sstable references | 12:13:49,518 | 127.0.0.4 |            975
Merging memtable tombstones | 12:13:49,518 | 127.0.0.3 |            977
Merging memtable tombstones | 12:13:49,518 | 127.0.0.4 |           1125
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 12:13:49,518 | 127.0.0.3 |           1059
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 12:13:49,518 | 127.0.0.4 |           1203
Merging data from memtables and 0 sstables | 12:13:49,518 | 127.0.0.3 |
      1082
Merging data from memtables and 0 sstables | 12:13:49,518 | 127.0.0.4 |
      1225
Read 1 live and 0 tombstoned cells | 12:13:49,518 | 127.0.0.3 |
 1273
Enqueuing response to /127.0.0.1 | 12:13:49,519 | 127.0.0.3 |           1485
Read 1 live and 0 tombstoned cells | 12:13:49,519 | 127.0.0.4 |
 1406
Sending message to /127.0.0.1 | 12:13:49,519 | 127.0.0.3 |           1711
Enqueuing response to /127.0.0.1 | 12:13:49,519 | 127.0.0.4 |           1628
Sending message to /127.0.0.1 | 12:13:49,519 | 127.0.0.4 |           1899
Message received from /127.0.0.3 | 12:13:49,520 | 127.0.0.1 |           7003
```

```
Message received from /127.0.0.4 | 12:13:49,520 | 127.0.0.1 |           7008
Processing response from /127.0.0.3 | 12:13:49,521 | 127.0.0.1 |
 7379
Processing response from /127.0.0.4 | 12:13:49,521 | 127.0.0.1 |
 7413
Request complete | 12:13:49,521 | 127.0.0.1 |           8088
```

6. 整合性レベルをQUORUMに変更して、SELECT文を再実行します。

```
cqlsh:demo_cl> CONSISTENCY quorum;
cqlsh:demo_cl> SELECT * FROM demo_table WHERE id = 0;
```

```
. . .
Tracing session:298af000-5576-11e3-add5-a180b5ee3938

execute_cql3_query | 12:19:12,038 | 127.0.0.1 |              0
Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000; | 12:19:12,038 |
 127.0.0.1 |             88
Preparing statement | 12:19:12,038 | 127.0.0.1 |            408
Sending message to /127.0.0.3 | 12:19:12,039 | 127.0.0.1 |           1210
Sending message to /127.0.0.4 | 12:19:12,039 | 127.0.0.1 |           1230
Message received from /127.0.0.1 | 12:19:12,040 | 127.0.0.3 |             66
Message received from /127.0.0.1 | 12:19:12,040 | 127.0.0.4 |             53
Executing single-partition query on demo_table | 12:19:12,040 | 127.0.0.3 |
           760
Executing single-partition query on demo_table | 12:19:12,040 | 127.0.0.4 |
           745
Acquiring sstable references | 12:19:12,040 | 127.0.0.3 |            790
Acquiring sstable references | 12:19:12,040 | 127.0.0.4 |            774
Merging memtable tombstones | 12:19:12,041 | 127.0.0.3 |            875
Merging memtable tombstones | 12:19:12,041 | 127.0.0.4 |            895
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 12:19:12,041 | 127.0.0.3 |            979
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 12:19:12,041 | 127.0.0.4 |           1022
Merging data from memtables and 0 sstables | 12:19:12,041 | 127.0.0.3 |
      1017
Merging data from memtables and 0 sstables | 12:19:12,041 | 127.0.0.4 |
      1068
Read 1 live and 0 tombstoned cells | 12:19:12,041 | 127.0.0.3 |
 1187
Read 1 live and 0 tombstoned cells | 12:19:12,041 | 127.0.0.4 |
 1231
Enqueuing response to /127.0.0.1 | 12:19:12,041 | 127.0.0.3 |           1468
Enqueuing response to /127.0.0.1 | 12:19:12,041 | 127.0.0.4 |           1714
Sending message to /127.0.0.1 | 12:19:12,042 | 127.0.0.3 |           2025
Sending message to /127.0.0.1 | 12:19:12,042 | 127.0.0.4 |           2253
Message received from /127.0.0.3 | 12:19:12,042 | 127.0.0.1 |           4326
Message received from /127.0.0.4 | 12:19:12,042 | 127.0.0.1 |           4343
Processing response from /127.0.0.3 | 12:19:12,043 | 127.0.0.1 |
 4519
Processing response from /127.0.0.4 | 12:19:12,043 | 127.0.0.1 |
 4537
Request complete | 12:19:12,043 | 127.0.0.1 |           5168
```

7. 整合性レベルをALLに変更して、SELECT文を再実行します。

```
cqlsh:demo_cl> CONSISTENCY ALL;
cqlsh:demo_cl> SELECT * FROM demo_table WHERE id = 0;
```

```
. . .
Tracing session:32733600-5576-11e3-add5-a180b5ee3938

activity                                                                  |
 timestamp    | source    | source_elapsed
```

```
-------------------------------------------------------------------------------
+--------------+-----------+----------------
execute_cql3_query | 12:20:37,404 | 127.0.0.1 |              0
Parsing SELECT * FROM demo_table WHERE id = 0 LIMIT 10000; | 12:20:37,404 |
 127.0.0.1 |             88
Preparing statement | 12:20:37,405 | 127.0.0.1 |            418
Sending message to /127.0.0.3 | 12:20:37,406 | 127.0.0.1 |           2055
Sending message to /127.0.0.4 | 12:20:37,406 | 127.0.0.1 |           2076
Sending message to /127.0.0.5 | 12:20:37,406 | 127.0.0.1 |           2101
Message received from /127.0.0.1 | 12:20:37,407 | 127.0.0.3 |             46
Message received from /127.0.0.1 | 12:20:37,407 | 127.0.0.5 |             57
Executing single-partition query on demo_table | 12:20:37,407 | 127.0.0.3 |
          554
Executing single-partition query on demo_table | 12:20:37,407 | 127.0.0.5 |
          547
Acquiring sstable references | 12:20:37,407 | 127.0.0.3 |            575
Acquiring sstable references | 12:20:37,407 | 127.0.0.5 |            572
Merging memtable tombstones | 12:20:37,407 | 127.0.0.3 |           634
Merging memtable tombstones | 12:20:37,407 | 127.0.0.5 |           645
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 12:20:37,407 | 127.0.0.3 |            719
Skipped 0/0 non-slice-intersecting sstables, included 0 due to tombstones |
 12:20:37,407 | 127.0.0.5 |            745
Merging data from memtables and 0 sstables | 12:20:37,407 | 127.0.0.3 |
       757
Merging data from memtables and 0 sstables | 12:20:37,407 | 127.0.0.5 |
       778
Read 1 live and 0 tombstoned cells | 12:20:37,408 | 127.0.0.3 |
 886
Read 1 live and 0 tombstoned cells | 12:20:37,408 | 127.0.0.5 |
 960
Enqueuing response to /127.0.0.1 | 12:20:37,408 | 127.0.0.3 |           1148
Enqueuing response to /127.0.0.1 | 12:20:37,408 | 127.0.0.5 |           1318
Sending message to /127.0.0.1 | 12:20:37,408 | 127.0.0.3 |           1352
Sending message to /127.0.0.1 | 12:20:37,408 | 127.0.0.5 |           1702
Enqueuing response to /127.0.0.1 | 12:20:37,409 | 127.0.0.4 |           2530
Message received from /127.0.0.3 | 12:20:37,409 | 127.0.0.1 |           4449
Message received from /127.0.0.5 | 12:20:37,409 | 127.0.0.1 |           4609
Sending message to /127.0.0.1 | 12:20:37,410 | 127.0.0.4 |           3274
Processing response from /127.0.0.3 | 12:20:37,410 | 127.0.0.1 |
 5326
Processing response from /127.0.0.5 | 12:20:37,410 | 127.0.0.1 |
 5411
Message received from /127.0.0.4 | 12:20:37,410 | 127.0.0.1 |           6311
Processing response from /127.0.0.4 | 12:20:37,411 | 127.0.0.1 |
 6933
Request complete | 12:20:37,411 | 127.0.0.1 |           7658
```

整合性がパフォーマンスに及ぼす影響

このタスクについて

整合性レベルを変更すると、読み取りのパフォーマンスに影響を与える場合があります。トレーシング出力は、整合性レベルをONEからQUORUM、次にALLに変更すると、パフォーマンスが2585から2998、次に5219マイクロ秒へとそれぞれ低下することを示します。1行のテーブルのクエリーは、例示のために使用した望ましくないケースであるため、このチュートリアルの手順に従っても、同じ傾向が見られるとは限りません。QUORUMとALLの違いは、この場合はごくわずかですが、クラスターの条件によっては、ALLを使用した場合のパフォーマンスはQUORUMより速くなる場合があります。

以下の条件下では、ALLを使用した場合のパフォーマンスはQUORUMより悪くなります。

- データが何千もの行で構成されている。
- 1つのノードが他のノードより遅い。
- 特に遅いノードがquorumの一員として選択されなかった。

大きなデータセットでのクエリーのトレース

10行以上存在するデータベースを対象に確率的トレースを使用できますが、この機能はこれ以上の多くのデータをトレースすることを目的としています。nodetoolユーティリティを使用して確率的トレースを構成した後に、system_tracesキースペースにクエリーします。

```
SELECT * FROM system_traces.events;
```

CQLリファレンス

はじめに

CQL言語に含まれているコマンドは、すべてcqlshコマンドラインで使用できます。コマンドラインでは使用できるけれども、CQL言語ではサポートされていないコマンドもあります。これらのコマンドは、cqlshコマンドと呼ばれます。cqlshコマンドは、コマンドラインからのみ実行できます。CQLコマンドを実行するには、いくつかの方法があります。

このリファレンスでは、CQL仕様3.1.0～3.1.6に基づくCQLとcqlshについて説明します。CQL 2は廃止されており、Cassandra 3.0で除外される予定です。

CQLの語彙構造

CQL入力は文で構成されます。SQLと同様、文によってデータを変更し、データを検索し、データを格納し、データの格納方法を変更します。文はセミコロン(;)で終了します。

たとえば、以下は有効なCQL構文です。

```
SELECT * FROM MyTable;

UPDATE MyTable
SET SomeColumn = 'SomeValue'
WHERE columnName = B70DE1D0-9908-4AE3-BE34-5573E5B09F14;
```

ここでは、2つのCQL文が続いています。1つの文を複数の行に分割すると便利な場合もありますが、この例では1つの行が1つの文を示しています。

大文字と小文字

CQLで作成したキースペース、カラム、テーブル名は、二重引用符で囲んでいない限り大文字と小文字が区別されません。これらのオブジェクトの名前を大文字で入力すると、Cassandraでは小文字で格納されます。二重引用符を使用すると、大文字と小文字が区別されるようになります。例:

```
CREATE TABLE test (
Foo int PRIMARY KEY,
"Bar" int
);
```

以下の表に、testテーブルから結果が返されるクエリーと返されないクエリーの一部を示します。

表 **2 :** 有効なクエリーと無効なクエリー

有効なクエリー	無効なクエリー
SELECT foo FROM . . .	SELECT "Foo" FROM . . .
SELECT Foo FROM . . .	SELECT "BAR" FROM . . .
SELECT FOO FROM . . .	SELECT bar FROM . . .
SELECT "foo" FROM . . .	
SELECT "Bar" FROM . . .	

SELECT "foo" FROM ...は、内部で、Cassandraがfooを小文字で格納するので有効です。二重引用符文字は、二重引用符のエスケープ文字として使用することができます。

以前のCQLバージョンでの大文字と小文字の区別の規則は、レガシー・テーブルを扱う場合に適用されます。

CQLキーワードでは、大文字と小文字が区別されません。たとえば、SELECTおよびselectキーワードは同じになります。このドキュメントでは、キーワードを大文字で示しています。

文字のエスケープ

CQLでは、CQLが解析できない文字が含まれているカラム名を二重引用符で囲む必要があります。

日付、IPアドレス、および文字列は単一引用符で囲む必要があります。文字列リテラルで単一引用符自体を使用するには、単一引用符をもう1つ続けてエスケープしてください。

有効なリテラル

有効なリテラルは以下の種類の値で構成されます。

- blob

  0[xX](hex)+で定義される16進数です
- boolean

  trueまたはfalseで、大文字／小文字は区別されず、引用符で囲まれていません
- 数値定数

  数値定数は、整数0～9およびマイナス記号(-)のプレフィックスで構成されます。数値定数は浮動小数点型にもなります。浮動小数点型は、1つまたは複数の10進数の桁、その後に小数点(.)、さらに1つまたは複数の10進数の桁が続きます。任意指定のプラス記号(+)はありません。.42や42のような表記は許されません。小数点の前後に先頭または末尾の0を使用することができます。たとえば、0.42や42.0などです。浮動小数点型定数はE表記で表され、以下の正規表現の文字から成ります。

  ```
  '-'?[0-9]+('.'[0-9]*)?([eE][+-]?[0-9+])?
  ```

  NaNおよび無限大は浮動小数点型です。
- 識別子

  英字に、任意の順で数字、アンダースコア、英字が続きます。テーブル、カラム、他のオブジェクトの名前は識別子で、二重引用符で囲まれています。
- 整数

  任意指定のマイナス記号(-)の後に、1つまたは複数の数字が続きます。

- 文字列リテラル

  単一引用符で囲まれた文字群。文字列リテラルで単一引用符自体を使用するには、単一引用符をもう1つ続けてエスケープしてください。たとえば、犬の所有形を表すには、次のように「''」を使用します：dog''s。
- uuid

  32桁の16進数で、0～9またはa～fを使用し、大文字と小文字は区別されず、8番目、12番目、16番目、20番目の数字がダッシュ（-）で区切られてます。例：01234567-0123-0123-0123-0123456789ab
- timeuuid

  00:00:00.00 UTC（60ビット）から100ナノ秒単位で間隔を取る時間、重複防止のためのクロック・シーケンス番号（14ビット）、およびIEEE 801 MACアドレス（48ビット）を使用して、一意の識別子を生成します。例：d2177dd0-eaa2-11de-a572-001b779c76e3
- 空白文字

  用語を区切ったり文字列リテラル内で使用されたりしますが、それ以外の場合、CQLは空白文字を無視します。

指数表記

Cassandraは指数表記をサポートしています。この例は、cqlshコマンドからの出力内の指数表記を示しています。

```
CREATE TABLE test(
id varchar PRIMARY KEY,
value_double double,
value_float float
);

INSERT INTO test (id, value_float, value_double)
VALUES ('test1', -2.6034345E+38, -2.6034345E+38);

SELECT * FROM test;
```

```

id    | value_double | value_float
-------+--------------+-------------
test1 |  -2.6034e+38 | -2.6034e+38

(1 rows)
```

CQLコードのコメント

以下の表記を使用して、CQLコードにコメントを含めることができます。

- 二重ハイフン

  ```
  --  単一行のコメント
  ```
- 二重スラッシュ

  ```
  //単一行のコメント
  ```
- スラッシュとアスタリスク

  ```
  /*  複数行にわたるコメント  */
  ```

CQLキーワード

この表には、キーワードとそのキーワードが予約されているかどうかを示します。予約済みのキーワードは、二重引用符で囲まない限り、識別子として使用することはできません。予約されていないキーワードは所定の文脈においては特定の意味を持ちますが、その文脈の外では識別子として使用することができます。

表 **3 :** キーワード

キーワード	予約済み
ADD	はい
ALL	いいえ
ALLOW	はい
ALTER	はい
AND	はい
ANY	はい
APPLY	はい
AS	いいえ
ASC	はい
ASCII	いいえ
AUTHORIZE	はい
BATCH	はい
BEGIN	はい
BIGINT	いいえ
BLOB	いいえ
BOOLEAN	いいえ
BY	はい
CLUSTERING	いいえ
COLUMNFAMILY	はい
COMPACT	いいえ
CONSISTENCY	いいえ
COUNT	いいえ
COUNTER	いいえ
CREATE	はい
CUSTOM	いいえ
DECIMAL	いいえ
DELETE	はい
DESC	はい
DISTINCT	いいえ
DOUBLE	いいえ
DROP	はい

キーワード	予約済み
EACH_QUORUM	はい
EXISTS	いいえ
FILTERING	いいえ
FLOAT	いいえ
FROM	はい
frozen	はい
GRANT	はい
IF	はい
IN	はい
INDEX	はい
INET	はい
INFINITY	はい
INSERT	はい
INT	いいえ
INTO	はい
KEY	いいえ
KEYSPACE	はい
KEYSPACES	はい
LEVEL	いいえ
LIMIT	はい
LIST	いいえ
LOCAL_ONE	はい
LOCAL_QUORUM	はい
MAP	いいえ
MODIFY	はい
NAN	はい
NORECURSIVE	はい
NOSUPERUSER	いいえ
NOT	はい
OF	はい
ON	はい

キーワード	予約済み
ONE	はい
ORDER	はい
PASSWORD	はい
PERMISSION	いいえ
PERMISSIONS	いいえ
PRIMARY	はい
QUORUM	はい
RENAME	はい
REVOKE	はい
SCHEMA	はい
SELECT	はい
SET	はい
STATIC	いいえ
STORAGE	いいえ
SUPERUSER	いいえ
TABLE	はい
TEXT	いいえ
TIMESTAMP	いいえ
TIMEUUID	いいえ
THREE	はい
TO	はい
TOKEN	はい
TRUNCATE	はい
TTL	いいえ
TWO	はい
TYPE	いいえ
UNLOGGED	はい
UPDATE	はい
USE	はい
USER	いいえ
USERS	いいえ

キーワード	予約済み
USING	はい
UUID	いいえ
VALUES	いいえ
VARCHAR	いいえ
VARINT	いいえ
WHERE	はい
WITH	はい
WRITETIME	いいえ

CQLのデータ型

CQLでは、カラムに対して組み込みのデータ型が定義されています。counter型は独特です。

表 4 : CQLのデータ型

CQLの型	定数	説明
ascii	文字列	US-ASCII文字列
bigint	整数	64ビット符号付きlong
blob	BLOB	任意数のバイト(検証なし)で、16進数で表示される
boolean	ブーリアン	trueまたはfalse
counter	整数	分散されたカウンター値(64ビットlong)
decimal	整数、浮動小数点数	可変精度の10進数 Javaの型
double	整数	64ビットIEEE-754の浮動小数点数 Javaの型
float	整数、浮動小数点数	32ビットIEEE-754の浮動小数点数 Javaの型
inet	文字列	IPv4形式またはIPv6形式のIPアドレス文字列で、python-cqlドライバーとCQLネイティブ・プロトコルで使用される
int	整数	32ビット符号付き整数

CQLの型	定数	説明
list	なし	1つ以上の順序が指定された要素のコレクション
map	なし	リテラルのJSON形式の配列 : { literal :literal, literal :literal ... }
set	なし	1つ以上の要素のコレクション
text	文字列	UTF-8エンコード文字列
timestamp	整数、文字列	8バイトでエンコードされた、エポック以降の日付および時刻
timeuuid	uuid	タイプ1 UUIDのみ
tuple	なし	Cassandra 2.1以降。2〜3フィールドで構成されるグループ。
uuid	uuid	標準UUID形式のUUID
varchar	文字列	UTF-8エンコード文字列
varint	整数	任意精度の整数 Javaの型

この表に示したCQLの型に加えて、JAVAクラス（Cassandraでロード可能なAbstractTypeのサブクラス）の名前を含んでいる文字列をCQLの型として使用できます。このクラス名は、完全修飾名か、またはorg.apache.cassandra.db.marshalパッケージを基準にした相対名である必要があります。

ASCII text、timestamp、およびinetの値は単一引用符で囲みます。キースペース、テーブル、またはカラムの名前は二重引用符で囲みます。

Javaの型

CQLのほとんどの型はJavaの型から派生しており、Javaプログラマーには明白です。しかし、以下の型の派生元は、必ずしも明白ではありません。

表 **5 : CQL**の特定の型の派生元

CQLの型	Javaの型
decimal	java.math.BigDecimal
float	java.lang.Float
double	java.lang.Double
varint	java.math.BigInteger

BLOB型

CassandraのBLOBデータ型は、0[xX](hex)+で定義される16進数の定数を表します。ここで、hexは16進数[0-9a-fA-F]です。たとえば、0xcafeとなります。

BLOB変換関数

これらの関数は、ネイティブの型をバイナリー・データ(BLOB)に変換します。

- typeAsBlob(type)
- blobAsType

CQLでサポートされている、BLOB以外のすべてのネイティブの型に対して、typeAsBlob関数はtype型の引数を受け取り、それをBLOBとして返します。これとは逆に、blobAsType関数は、64ビットのBLOB引数を受け取り、それをbigint値に変換します。

例:

```
CREATE TABLE bios ( user_name varchar PRIMARY KEY,
bio blob
 );

INSERT INTO bios (user_name, bio) VALUES ('fred', bigintAsBlob(3));

SELECT * FROM bios;

user_name | bio
-----------+--------------------
fred | 0x0000000000000003

(1 rows)

ALTER TABLE bios ADD id bigint;

INSERT INTO bios (user_name, id) VALUES ('fred',
 blobAsBigint(0x0000000000000003));

SELECT * FROM bios;

user_name | bio                | id
-----------+--------------------+----
fred | 0x0000000000000003 |  3
```

コレクション型

コレクション・カラムはコレクション型を使用して宣言します。コレクション型の後ろに、intやtextなどの型を<>で囲みます。たとえば、テキスト要素のリスト、整数のリスト、またはその他の要素の型のリストを持つテーブルを作成できます。

```
list<text>
list<int>
```

現在、コレクション型をネストすることはできません。たとえば、以下のように、リストの中にリストを定義することはできません。

```
list<list<int>>      \\ネストできない
```

現在は、マップ、セット、またはリストの型のカラムにインデックスを作成することはできません。

counter型

カウンター・カラムの値は64ビットの符号付き整数です。カウンターの値は、2つの操作(インクリメントとデクリメント)をサポートするもので、設定できません。

「カウンターの使用」セクションで説明されているように、counter型を使用してください。この型を、プライマリ・キーまたはパーティション・キーとして使用されるカラムに割り当てないでください。また、counter型を、counter型とプライ

マリ・キー以外のものを含んでいるテーブル内で使用しないでください。サロゲート・キー用に連続番号を生成するには、counter型ではなくtimeuuid型を使用してください。カウンター・カラムではインデックスを作成できません。

UUIDおよびtimeuuid型

UUID（universally unique id）コンパレーター型は、カラム名の衝突を防ぐために使用されます。代わりに、timeuuidを使用することもできます。

timeuuid型はCQL入力の整数として入力できます。timeuuidの値はタイプ1のUUIDです。タイプ1のUUIDには生成時間が含まれ、タイムスタンプでソートされるので、競合のないタイムスタンプを必要とするアプリケーションで使用するのに理想的です。たとえば、このタイプを使用すると、タイムスタンプでカラム（ブログエントリなど）を特定したり、複数のクライアントが同じパーティション・キーに同時に書き込んだりすることができます。上書きを意図していないデータが上書きされるような衝突は発生しません。

有効なtimeuuidは、「有効なリテラル」に示すtimeuud形式に従います。

uuidおよびtimeuuid関数

Cassandra 2.0.7以降には、uuid()関数が含まれています。この関数ではパラメーターが不要で、INSERTまたはSET文での使用に適したランダムなタイプ4のUUIDを生成します。

いくつかのtimeuuid関数は、timeuuid型と併用するために設計されています。

- dateOf()

  SELECT句で使用するこの関数は、結果セットに含まれているtimeuuidカラムのタイムスタンプを抽出します。この関数は抽出されたタイムスタンプを日付として返します。生のタイムスタンプを取得するには、unixTimestampOf()を使用します。
- now()

  文が実行されたときに、新しい一意のtimeuuidをミリ秒単位で生成します。timeuuidのタイムスタンプの部分は、UTC（協定世界時）標準に一致します。この方法は値の挿入に役立ちます。now()で返される値は必ず一意になります。
- minTimeuuid()およびmaxTimeuuid()

  条件用の時間の構成要素を引数として受け取ってUUID相当の結果を返します。例:

  ```
  SELECT * FROM myTable
  WHERE t > maxTimeuuid('2013-01-01 00:05+0000')
  AND t < minTimeuuid('2013-02-02 10:00+0000')
  ```
- unixTimestampOf()

  SELECT句で使用するこの関数は、結果セットに含まれているtimeuuidカラムのタイムスタンプをミリ秒単位で抽出します。値を、生の64ビット整数タイムスタンプとして返します。

前述のminTimeuuid/maxTimeuuidの例では、timeuuidカラムのtが厳密に2013-01-01 00:05+0000以降で、かつ、2013-02-02 10:00+0000以前である行をすべて選択します。t >= maxTimeuuid('2013-01-01 00:05+0000')では、2013-01-01 00:05+0000ちょうどに生成されたtimeuuidは選択されず、基本的にt > maxTimeuuid('2013-01-01 00:05+0000')と等しくなります。

minTimeuuidおよびmaxTimeuuid関数で返される値は、RFC 4122に規定されている時間ベースのUUID生成プロセスに従っていないため、真正のUUIDではありません。これらの関数の結果は、now関数とは異なり決定的です。

タイムスタンプ型

タイムスタンプ型の値は64ビットの符号付き整数としてエンコードされ、エポックとして知られる標準基準時間（1970年1月1日、GMT時間0時0分0秒）からのミリ秒数を表します。タイムスタンプ型は、CQL用に整数として、または、以下のいずれかのISO 8601形式の文字列リテラルとして指定できます。

```
yyyy-mm-dd HH:mm
yyyy-mm-dd HH:mm:ss
yyyy-mm-dd HH:mmZ
yyyy-mm-dd HH:mm:ssZ
yyyy-mm-dd'T'HH:mm
yyyy-mm-dd'T'HH:mmZ
yyyy-mm-dd'T'HH:mm:ss
yyyy-mm-dd'T'HH:mm:ssZ
yyyy-mm-dd
yyyy-mm-ddZ
```

ここでZは、RFC-822の4桁のタイム・ゾーンで、UTCからの時差を表しています。たとえば、2011年2月3日、GMT時間午前4時5分0秒の場合、以下のようになります。

```
2011-02-03 04:05+0000
2011-02-03 04:05:00+0000
2011-02-03T04:05+0000
2011-02-03T04:05:00+0000
```

タイム・ゾーンが指定されていない場合、書き込み要求を扱うCassandraコーディネーター・ノードのタイム・ゾーンが使用されます。正確さを期すため、DataStaxではCassandraノードに設定されているタイム・ゾーンに依存するのではなく、タイム・ゾーンを指定することを推奨しています。

日付の値だけを捕らえたい場合、時刻を省略することもできます。例:

```
2011-02-03
2011-02-03+0000
```

この場合、時間のデフォルトは、指定のタイム・ゾーンまたはデフォルトのタイム・ゾーンで00:00:00です。

タイムスタンプ出力は、デフォルトで以下の形式に従い表示されます。

```
yyyy-mm-dd HH:mm:ssZ
```

cqlshrcファイルの[ui]セクションにあるtime_formatプロパティを設定することで、形式を変更できます。

タプル型

Cassandra 2.1には、位置指定された型付きのフィールドを要素とする固定要素数の集合を保持するタプル型が導入されています。タプルは、ユーザー定義型の代わりに使用できる便利な手段です。たとえば、コレクションとしてグループ化するフィールド群がある場合、プロトタイプ・フェーズでタプルを使用します。タプルには多くのフィールドを含めることができますが、使用できる数（32768）より控えめにしてください。一般的には、2、3のフィールドのみを含むタプルを作成します。プロトタイプ・フェーズ後に、さらにフィールドが必要になった場合は、ユーザー定義の型を使用します。

テーブル作成文では、カラムをfrozenタプル型として宣言し、山かっこ区切り文字のコンマを使用して、タプルの構成要素の型を宣言します。タプル値をかっこで囲んで、以下の例のようにテーブルに値を挿入します。

```
CREATE TABLE collect_things (
k int PRIMARY KEY,
v frozen <tuple<int, text, float>>
);

INSERT INTO collect_things (k, v) VALUES(0, (3, 'bar', 2.1));

SELECT * FROM collect_things;
```

```
k | v
---+-----------------
0 | (3, 'bar', 2.1)
```

タプルを使用して検索をフィルターできます。

```
CREATE INDEX on collect_things (v);

SELECT * FROM collect_things WHERE v = (3, 'bar', 2.1);

k | v
---+-----------------
0 | (3, 'bar', 2.1)
```

以下の例に示すように、タプルをネストできます。

```
create table nested (k int PRIMARY KEY, t frozen <tuple <int, tuple<text,
 double>>>);

INSERT INTO nested (k, t) VALUES (0, (3, ('foo', 3.4)));
```

ユーザー定義の型

Cassandra 2.1ではユーザー定義の型をサポートしています。ユーザー定義の型を使用すると、テーブル内で関連する情報を持つ複数のフィールドを扱うのが容易になります。情報を別のテーブルに格納するのではなく、ユーザー定義の型を使用して関連する情報のフィールドを表すことで、複数のテーブルを必要とするアプリケーションを、より少ないテーブルを使うようにして簡素化できます。住所の型の例では、ユーザー定義の型の使用方法を示します。

以下のコマンドを使用して、ユーザー定義の型の作成、変更、削除ができます。

- CREATE TYPE
- ALTER TYPE
- DROP TYPE

cqlshユーティリティには、ユーザー定義の型の仕様を表示したり、すべてのユーザー定義の型を一覧表示するための、以下のコマンドが含まれています。

- DESCRIBE TYPE
- DESCRIBE TYPES

ユーザー定義の型は、それを定義したキースペース内で有効です。その範囲外のキースペースから型にアクセスするには、ドット表記を使用します。キースペース名の後にピリオドを付け、その後に型の名前を記述します。Cassandraは、指定されたキースペース内でその型にアクセスしますが、現在のキースペースは変更しません。キースペースを指定しないと、Cassandraは現在のキースペース内のその型にアクセスします。

CQLのキースペースとテーブル・プロパティ

CQLのWITH句は、以下のCQLコマンドでキースペースとテーブル・プロパティを指定します。

- ALTER KEYSPACE
- ALTER TABLE
- CREATE KEYSPACE
- CREATE TABLE

CQLのキースペース・プロパティ

CQLでは、データ・センターの命名に加えて、以下のキースペース・プロパティの設定をサポートしています。

- class

  レプリケーション・ストラテジの名前：SimpleStrategyまたはNetworkTopologyStrategy。各データ・センターのレプリケーション係数を個別に設定します。
- replication_factor

  replication_factorプロパティは、CREATE KEYSPACE の例に示すように、SimpleStrategyを指定している場合にのみ使用します。レプリケーション係数値は、クラスター全体でのレプリカの総数です。

実稼働環境で使用する場合、または混在ワークロードで使用する場合は、NetworkTopologyStrategyを使用してキースペースを作成します。SimpleStrategyは評価目的に適しています。将来拡張が必要になった場合に複数のデータ・センターに容易に拡張できるため、ほとんどの展開にNetworkTopologyStrategyを推奨します。

キースペースの作成や変更時に、durable writesプロパティを構成することもできます。

テーブルのプロパティ

CQLは、Cassandraのテーブル・プロパティ（commentやcompactionオプションなど。以下の表を参照）をサポートしています。

CREATE TABLEのようなCQLコマンドでは、プロパティを名前と値の組またはコレクション・マップ形式のいずれかでフォーマットします。名前と値の組プロパティ構文は、以下のとおりです。

```
name = value AND name = value
```

compactionおよびcompressionプロパティで使用されるコレクション・マップ形式は以下のとおりです。

```
{ name :value, name :value, name :value ... }
```

文字列プロパティは単一引用符で囲みます。

例として、CREATE TABLEを参照してください。

表 **6 : CQL**プロパティ

CQLプロパティ	説明	デフォルト
bloom_filter_fp_chance	SSTableブルーム・フィルターの望ましい偽陽性確率。詳細. . .	0.01（`SizeTieredCompactionStrategy`の場合）0.1（`LeveledCompactionStrategy`の場合）
caching	キャッシュ・メモリーの使用を手動調整なしに最適化します。.. . .詳細	Cassandra 2.1: keysの場合ALL rows_per_partitionの場合NONE Cassandra 2.0.x:keys_only
comment	人間が読める、テーブルを説明するコメント。. . .詳細	なし
compaction	テーブルのコンパクション・ストラテジを設定します。. . .詳細	`SizeTieredCompactionStrategy`
compression	使用する圧縮アルゴリズム。有効な値は、`LZ4Compressor`、`SnappyCompressor`、	`LZ4Compressor`

CQLプロパティ	説明	デフォルト
	およびDeflateCompressorです。. . .詳細	
dclocal_read_repair_chance	現在のデータ・センターですべてのレプリカを対象にリード・リペアが呼び出される確率を指定します。	0.1（Cassandra 2.1、Cassandra 2.0.9以降）0.0（Cassandra 2.0.8以前）
default_time_to_live	テーブルのデフォルトの有効期限（秒単位）。TTLの制御がないときに、MapReduce/Hiveシナリオで使用されます。	0
gc_grace_seconds	トゥームストーン（削除マーカー）のガーベジ・コレクションを実行するまでの待機時間を指定します。デフォルト値は、削除前に整合性が満たされるように十分な時間が設定されています。多くの展開でこの間隔は短縮でき、また、シングル・ノード・クラスターでは安全に0に設定できます。	864000 [10 days]
min_index_interval、max_index_interval（Cassandra 2.1.x）またはindex_interval（Cassandra 2.0.x）	パーティション・インデックスのエントリーのサンプリングを制御するため、これらのプロパティを変更してパーティション・サマリーのサンプル頻度を設定します。. . .詳細	min_index_interval 128およびmax_index_interval 2048、またはindex_interval 128
memtable_flush_period_in_ms	指定した時間（ミリ秒単位）の経過後、memtableのフラッシュを強制的に実行します。	0
populate_io_cache_on_flush (Cassandra 2.0.xのみ)	新しくフラッシュされた、またはコンパクトされたSSTableをOSのページ・キャッシュに追加します。この結果、空き領域を作るために他のキャッシュされているデータの領域が明け渡される場合があります。テーブルの全データをメモリーに収める必要がある場合に有効にします。cassandra.yamlファイルのグローバルなcompaction_preheat_key_cacheオプションを設定することもできます。	false
read_repair_chance	非クォーラムの整合性のために構成されたクラスターの読み取り時にリード・リペアを呼び出す基準を	0.0（Cassandra 2.1、Cassandra 2.0.9以降）0.1（Cassandra 2.0.8以前）

CQLプロパティ	説明	デフォルト
	指定します。値は0～1とする必要があります。	
replicate_on_write（Cassandra 2.0.xのみ）	Cassandra 2.1で削除されました。カウンター・テーブルのみに適用します。trueに設定すると、クライアントに指定された書き込み要求の整合性レベルにかかわらず、影響を受けるすべてのレプリカへの書き込みが複製されます。カウンター・テーブルの場合、必ずtrueに設定してください。	true
speculative_retry	read_repair_chanceが1.0ではない場合に通常の読み取りタイムアウトをオーバーライドして、別の読み取り要求を送信します。詳細 . . .	99パーセンタイル（Cassandra 2.0.2以降）

ブルーム・フィルター

ブルーム・フィルターのプロパティは、SSTableブルーム・フィルターの望ましい偽陽性確率です。データが要求されると、ディスクI/Oが実行される前に、ブルーム・フィルターによって行が存在するかどうかがチェックされます。ブルーム・フィルターのプロパティ値は0～1.0です。最小値と最大値の効果は以下のとおりです。

0 変更のない、実質的に最大限のブルーム・フィルターを有効にします。

1.0 ブルーム・フィルターを無効にします。

推奨される設定値は0.1です。値が高いほど効果が小さくなります。

キャッシング

キャッシュ・メモリーの使用を手動調整なしに最適化します。テーブルを作ったり変更したりするときに、テーブル・プロパティを設定してキャッシングを構成します。Cassandraはキャッシュされたデータにサイズとアクセス頻度で重みを付けます。キャッシング・テーブル・プロパティを設定してから、`cassandra.yaml`ファイルのグローバル・キャッシング・プロパティを設定します。グローバル・キャッシング・プロパティについては、Cassandra 2.1ドキュメントまたはCassandra 2.0ドキュメントを参照してください。

Cassandra 2.1

キャッシング・プロパティの値のプロパティ・マップを作成することによってキャッシュを設定します。オプションは以下のとおりです。

- keys：ALLまたはNONE
- rows_per_partition：CQL行の数、NONEまたはALL

例：

```
CREATE TABLE DogTypes (
block_id uuid,
species text,
alias text,
population varint,
PRIMARY KEY (block_id)
) WITH caching = { 'keys' :'NONE', 'rows_per_partition' :'120' };
```

Cassandra 2.0

以下のキャッシング・プロパティ・オプションのいずれかを使用してキャッシュを設定します。

- all
- keys_only
- rows_only
- none

オプションを使用してキーまたは行のキャッシュを指定したり、キーと行のキャッシュの両方を指定したりできます。例:

```
// Cassandra 2.0.xのみ

CREATE TABLE DogTypes (
block_id uuid,
species text,
alias text,
population varint,
PRIMARY KEY (block_id)
) WITH caching = 'keys_only';
```

重要： Cassandra 2.0.xでは、注意して行キャッシングを使用してください。

コメント

コメントは、アプリケーション・コードのCQLステートメントに注釈を付けるために使用できます。1行のコメントはダブル・ダッシュ(--)またはダブル・スラッシュ(//)で始まり、行の終わりまで続きます。複数行のコメントは/*と*/の記号の間に挟みます。

コンパクション

コンパクション・プロパティは使用するコンパクション・ストラテジ・クラスを定義します。サポートされているクラスは以下のとおりです。

- `SizeTieredCompactionStrategy`:デフォルトのコンパクション・ストラテジ。このストラテジは、ディスク上に、min_thresholdテーブル・サブプロパティで構成されているサイズに似たサイズのSSTableがいくつかあるときに、マイナーなコンパクションを発動します。マイナーなコンパクションでは、キースペースのすべてのテーブルが関与するわけではありません。
- `LeveledCompactionStrategy`:レベル化コンパクション・ストラテジでは、いくつかのレベルにグループ化される、固定の、比較的小さなサイズ(デフォルトでは5MB)のSSTableが作成されます。各レベル内で、SSTableがオーバーラップすることはありません。各レベル(L0、L1、L2など)は、1つ下のレベルに比べると10倍大きくなっています。SSTableは、継続的に、次第により大きなレベルへとコンパクトされていくため、レベルが低い場合よりもレベルが高い方が、ディスクI/Oが平準化され予測しやすくなります。レベルごとに、行キーはオーバーラップしないSSTableにマージされます。Cassandraが、各レベルのどのSSTableで行キー・データの有無を調べるかを特定できるため、読み取りのパフォーマンスが向上します。このコンパクション・ストラテジは、Googleのleveldb実装をモデルにしています。詳細は、「When to Use Leveled Compaction」、「Leveled Compaction in Apache Cassandra」、「コンパクション・サブプロパティ」を参照してください。

書き込みが多いワークロードにコンパクションが追いつかない場合、レベル化コンパクション・ストラテジを向上するハイブリッド(レベル化およびサイズ階層化)コンパクションにより、読み取り操作のパフォーマンス・オーバーヘッドが低減します。LeveledCompactionStrategyを使用している場合、Cassandraがワークロードに追いつかないと、Cassandraが追いつくまでSizeTieredCompactionに切り替わります。このため、切り替えが発生したときに使用するように、テーブルのmax_thresholdサブプロパティを設定するのがベスト・プラクティスです。

カスタム・ストラテジを指定することができます。文字列定数として完全なクラス名を使用します。

圧縮

圧縮を設定するには、テーブルの作成や変更の際に、LZ4Compressor、SnappyCompressor、またはDeflateCompressorプロパティを選択して使用します。圧縮を無効にするには、空の文字列('')を使用します。サブプロパティの使い方の例を参照してください。適切な圧縮の選択は、要件として読み取りパフォーマンスよりも領域の節約が重要であるかどうかにより異なります。解凍速度が最も速いのはLZ4で、次にSnappy、Deflateの順になります。圧縮効果は解凍速度に反比例します。DeflateまたはSnappyによる高い圧縮率は、一般用途のワークロードでのパフォーマンス低下と引き換えにするほどではありませんが、アーカイブ用データでは検討に値する場合があります。また、開発者は、org.apache.cassandra.io.compress.ICompressorインターフェイスを使用してカスタム圧縮クラスを実装することもできます。完全なクラス名を単一引用符で囲んで指定します。また、圧縮サブプロパティを使用します。

min_index_intervalおよびmax_index_interval

index_interval(Cassandra 2.0.x)プロパティまたはmin_index_intervalおよびmax_index_interval(Cassandra 2.1)プロパティは、プライマリ行インデックスのエントリのサンプリングを制御し、インデックス間隔を変えることでパーティション・サマリーのサンプリング頻度を設定します。インデックス間隔を変えると、SSTableは新しい情報を使ってディスクに書き込まれます。間隔は、サンプルを取得するたびにスキップするインデックス・エントリの数を示します。デフォルトで、Cassandraは128の行キーごとに1つの行キーをサンプリングします。間隔が大きくなるほどサンプリングは少なくなり、有効性が低くなります。サンプリングが多いほど、インデックスの有効性は高くなりますが、メモリー使用量が大きくなります。Cassandra 2.0.xでは、一般的に、メモリー使用量とパフォーマンスの最良のトレード・オフは、大きなテーブル・キー・キャッシュと組み合わせた場合に128～512の値になります。しかし、行が小さい場合(1 OSページに多数)、サンプル・サイズを大きくした方がよいでしょう。こうすると、多くの場合パフォーマンスを損なわずにメモリー使用量が小さくなります。大きな行では、サンプル・サイズを小さくすると、読み取りパフォーマンスが向上する場合があります。

Cassandra 2.1では、index_intervalの名前がmin_index_intervalおよびmax_index_intervalに置き換えられています。max_index_intervalは、デフォルトで2048です。デフォルト値に達するのは、SSTableの読み取り頻度が低く、インデックス・サマリー・メモリ・プールがいっぱいの場合だけです。以前のリリースからアップグレードすると、Cassandraではmin_index_intervalとして古いindex_interval値が使用されます。

推測的再試行

read_repair_chanceが1.0ではない場合に通常の読み取りタイムアウトをオーバーライドして、読み取り要求をもう1回送信させるには、以下の値のいずれかを選び、プロパティを使用してテーブルを作成または変更します。

- ALWAYS:すべてのレプリカ読み取りを再試行します。
- Xpercentile:スループットおよびレイテンシーへの影響に基づいて読み取りを再試行します。
- Yms:指定されたミリ秒後に、読み取りを再試行します。
- NONE:読み取りを再試行しません。

推測的再試行プロパティを使用すると、高速読み取り保護をCassandra 2.0.2以降で設定できます。数ミリ秒が経過してから、または一般的な読み取りレイテンシーのパーセンタイルに達してから要求を再試行するには、このプロパティを使用します。これはテーブルごとに追跡されます。例:

```
  ALTER TABLE users WITH speculative_retry = '10ms';
```

または:

```
ALTER TABLE users WITH speculative_retry = '99percentile';
```

関連情報
Cassandra 2.0 cassandra.yaml
Cassandra 2.1 cassandra.yaml

コンパクション・サブプロパティ

CQLを使用して、コンパクション・プロパティのマップと以下のサブプロパティを作成することにより、テーブルのSizeTieredCompactionStrategy（STCS）またはLeveledCompactionStrategy（LCS）コンパクションを構成できます。

表 **7 : CQL**コンパクション・サブプロパティ

コンパクション・サブプロパティ	説明	デフォルト	サポートされているストラテジ
bucket_high	サイズ階層化コンパクションでは、SSTableのサイズとデフォルトのbucket_low値およびデフォルトbucket_high値の相違が50%以下の場合に同じバケット内にあると見なされます。[平均サイズ × bucket_low, 平均サイズ × bucket_high]	1.5	STCS
bucket_low	上記と同じ	0.5	STCS
cold_reads_to_omit	無視されたSSTableが考慮する読み取り数/秒の最大パーセント。値の推奨範囲は、0.0～1.0です。... 詳細	0.05	STCS
enabled	バックグラウンド・コンパクションを有効にします。...詳細	true	すべて
max_threshold	STCSでは、マイナー・コンパクションで許可するSSTableの最大数を設定します。LCSでは、L0が遅れたとき、つまり、L0が累積されMAX_COMPACTING_L0 SSTableを超えたときにL0に適用されます。	32	STCS
min_threshold	STCSでは、マイナー・コンパクションを発動するSSTableの最小数を設定します。	4	STCS
min_sstable_size	STCSは、コンパクション対象のSSTableをバケットにグループ化します。バケット・プロセスは、サ	50MB	STCS

コンパクション・サブプロパティ	説明	デフォルト	サポートされているストラテジ
	イズの違いが50%未満のSSTableをグループ化します。その結果、小規模なSSTableの場合に、粒度が細かすぎるバケット・プロセスになってしまいます。SSTableが小規模な場合は、一定値未満のSSTableはすべて同一のバケットに分類されるように、min_sstable_sizeを使用して、サイズのしきい値（単位はバイト）を定義してください。		
sstable_size_in_mb	レベル化コンパクション・ストラテジを使用するSSTableのターゲット・サイズ。SSTableのサイズは、sstable_size_in_mb以下である必要がありますが、コンパクション中にそれを超える場合があります。これは、パーティション・キーのデータが非常に大きい場合に発生します。データは2つのSSTableに分割されません。	160MB	LCS
tombstone_compaction_interval	SSTable作成後、トゥームストーン・コンパクション対象としてSSTableを考慮するまで待つ最小時間。トゥームストーン・コンパクションは、SSTableに、tombstone_thresholdよりも多くのガーベジ・コレクション対象にできるトゥームストーンがある場合に発動されるコンパクションです。	1 day	すべて
tombstone_threshold	ガベージ・コレクションの対象にできるトゥームストーンと、含まれているすべてのカラムとの比率。SSTableがこれを超えた場合、トゥームストーンの	0.2	すべて

コンパクション・サブプロパティ	説明	デフォルト	サポートされているストラテジ
	パージを目的として、(他のSSTableなしで)コンパクションが発動されます。		
unchecked_tombstone_compaction	trueの場合は、通常のトゥームストーン・コンパクションよりも積極的になります。成功の可能性をチェックしないで、SSTableトゥームストーン・コンパクションが1回実行されます。	false	すべて

cold_reads_to_omit

無視されたSSTableが考慮する読み取り数/秒の最大パーセント。値の推奨範囲は、0.0～1.0です。Cassandra 2.1以降では、最も読み取り頻度の低い5%のSSTableを無視します。Cassandra 2.0.3以降では、デフォルトのcold_reads_to_omitが(0.0)で、すべてのSSTableがコンパクト化されます。

cold_reads_to_omitプロパティ値を大きくしてテーブルごとにパフォーマンスを調整できます。値が1.0の場合、コンパクションは完全に無効になります。「Optimizations around Cold SSTables」ブログには、読み取り頻度の低いSSTableのコンパクト化を回避するためにこのプロパティを使用してパフォーマンスを調整する方法が詳しく記載されています。cold_reads_to_omitを構成するには、`ALTER TABLE`コマンドを使用します。

バックグラウンド・コンパクションを有効および無効にする

enabledプロパティを使用してバックグラウンド・コンパクションを無効にする例を以下に示します。

```
ALTER TABLE mytable WITH COMPACTION = {'class':'SizeTieredCompactionStrategy',
 'enabled':'false'}
```

圧縮サブプロパティ

CQLを使用して、コンパクション・プロパティのマップと以下のサブプロパティを作成することにより、テーブルの圧縮を構成できます。

表 **8 : CQL**圧縮サブプロパティ

圧縮サブプロパティ	説明	デフォルト
sstable_compression	使用する圧縮アルゴリズム。有効な値は、`LZ4Compressor`、`SnappyCompressor`、および`DeflateCompressor`です。. . .詳細	`SnappyCompressor`
chunk_length_kb	ディスク上では、SSTableは、ランダム読み取りを可能にするためにブロック単位で圧縮されます。この圧縮サブプロパティは、そのブロックのサイズ(KB)を定義します。値をデフォルト値より大きくすると、圧縮率が上	64KB

圧縮サブプロパティ	説明	デフォルト
	がりますが、実際に読み取るときに読み取るデータの最小サイズが大きくなります。テーブルの圧縮には、デフォルト値が適切です。読み取り/書き込みアクセス・パターン(一度に要求される標準的なデータ量)とテーブル行の標準的なサイズに合わせて圧縮サイズを調整します。	
crc_check_chance	圧縮が有効にされている場合、圧縮されている各ブロックには、ディスクのデータ劣化を検知し、破損が他のレプリカに伝播するのを防止するために、そのブロックのチェックサムが含まれます。このオプションは、読み取り時にそれらのチェックサムをチェックする割合を定義します。デフォルトでは、常にチェックされます。チェックサムのチェックを無効にするには、0に設定します。たとえば、2回に1回チェックするには、0.5に設定します。	1.0

sstable_compression

使用する圧縮アルゴリズム。有効な値は、`LZ4Compressor`、`SnappyCompressor`、および`DeflateCompressor`です。圧縮を無効にするには、以下のように空の文字列('')を使用します。

```
ALTER TABLE mytable WITH COMPRESSION = {'sstable_compression':''};
```

適切な圧縮の選択は、要件として読み取りパフォーマンスよりも領域の節約が重要であるかどうかにより異なります。解凍速度が最も速いのはLZ4で、次にSnappy、Deflateの順になります。圧縮効果は解凍速度に反比例します。DeflateまたはSnappyによる高い圧縮率は、一般用途のワークロードでのパフォーマンス低下と引き換えにするほどではありませんが、アーカイブ用データでは検討に値する場合があります。また、開発者は、`org.apache.cassandra.io.compress.ICompressor`インターフェイスを使用してカスタム圧縮クラスを実装することもできます。完全なクラス名を「文字列定数」として指定します。

関数

CQLでは、1つ以上のカラム値を新しい値に変換するいくつかの関数がサポートされています。集約関数はサポートされていません。

- BLOB変換関数
- timeuuid関数
- token関数
- WRITETIME関数
- TTL関数

特定のパーティション・キーのトークンを計算するには、token関数を使用します。token関数の正確なシグネチャーは、クラスターが使用するテーブルとパーティショナーによって異なります。token関数への引数の型は、パーティション・キー・カラムの型によって異なります。戻りの型は、使用されるパーティショナーによって以下のように異なります。

- Murmur3Partitioner、bigint
- RandomPartitioner、varint
- ByteOrderedPartitioner、blob

たとえば、デフォルトのMurmur3Partitionerを使用するクラスターでは、このテーブルのパーティション・キーのトークンを計算するtoken関数は、text型の1つの引数を取ります。パーティション・キーはユーザーIDです。クラスター化カラムがないため、このパーティション・キーはプライマリ・キーと同じで、戻りの型はbigintです。

```
CREATE TABLE users {
    userid text PRIMARY KEY,
    username text,
    ...
}
```

使用するパーティショナーにかかわらず、Cassandraでは、パーティション・キーにおいて不等号条件演算子をサポートしていません。パーティション・キーでの範囲のクエリーにはtoken関数を使用してください。

cqlshコマンド

cqlsh

構文

```
$ cqlsh [options] [host [port]]
```

```
$ python cqlsh [options] [host [port]]
```

説明

Cassandraインストールには、Cassandraクエリ言語（CQL）コマンドを実行するためのpythonベースのコマンドライン・クライアントである、cqlshユーティリティが含まれています。cqlshコマンドは、LinuxまたはWindowsのコマンドラインでcqlshユーティリティを起動するために使用します。Windowsでは、キーワード*python*を使用します。

cqlshを使用することで、CQLコマンドを対話型で実行できます。cqlshはタブ補完をサポートしています。また、TRACEなどのcqlshコマンドを実行することもできます。

要件

Cassandra 2.1のcqlshユーティリティは、ネイティブ・プロトコルを使用しています。Datastax pythonドライバーを使用するCassandra 2.1のデフォルトのcqlshリッスン・ポートは9042です。

Cassandra 2.0のcqlshユーティリティは、Thriftトランスポートを使用しています。Cassandra 2.0.xのデフォルトのcqlshリッスン・ポートは9160です。Cassandra 2.0.x以前のバージョンのデフォルトでは、cassandra.yamlファイルでstart_rpcをtrueに構成することでThriftを有効にします。cqlshユーティリティはThrift RPCサービスを使用しています。また、Thriftポートを介したアクセスを可能にするファイアウォール構成も必要になることがあります。

構成の詳細については、Cassandra 2.1 cassandra.yamlまたはCassandra 2.0 cassandra.yamlファイルを参照してください。

オプション

-C、--color
常に色出力を使用します。

--debug
詳細なデバッグ情報を表示します。

-e cql_statement、--execute cql_statement
Cassandra 2.1以降で、CQLコマンドを受け付けて実行します。ファイルへのCQL出力の保存に便利です。

-f file_name、--file=file_name
file_nameからコマンドを実行して終了します。

-h、--help
これらのオプションに関するオンライン・ヘルプを表示して終了します。

-k keyspace_name
指定のキースペースを使用します。cqlshの起動直後にUSE *keyspace*コマンドを発行することと同じです。

--no-color
色出力を使用しません。

-p password
パスワードを使用して認証を行います。デフォルトはcassandraです。

-t transport_factory_name、--transport=transport_factory_name
提供されたThriftトランスポート・ファクトリ機能を使用します。

-u user_name
user_nameに指定されたユーザーとして認証します。デフォルトはcassandraです。

--version
cqlshのバージョンを表示します。

cqlshrcオプション

ホーム・ディレクトリーの`.cassandra`隠しディレクトリーに存在する`cqlshrc`ファイルを作成できます。`cqlshrc`ファイルを構成するには、ファイルの[authenication]、[ui]、または[ssl]セクションにこれらのオプションを設定します。

[ui]オプションは以下のとおりです。

color
常に色出力を使用します。

completekey
このキーをcqlshシェル・エントリの自動補完に使用します。デフォルトはTabキーです。

float_precision
この小数点以下の桁数の精度を使用します。デフォルトは5です。

time_format
タイムスタンプ型のデータベース・オブジェクトの出力形式を構成します。たとえば、yyyy-mm-dd HH:mm:ssZという形式の場合、生成されるタイムスタンプは2014-01-01 12:00:00GMTです。デフォルトは'%Y-%m-%d %H:%M:%S%z'です。

[authentication]オプションは以下のとおりです。

keyspace

指定のキースペースを使用します。cqlshの起動直後にUSE keyspaceコマンドを発行することと同じです。

password

パスワードを使用して認証を行います。

username

指定のユーザーとして認証します。

[ssl]オプションについては、Cassandraのドキュメントで説明しています。

CQLコマンドの使用

起動時、cqlshによって、cqlshユーティリティへの接続に使用されるクラスターの名前、IPアドレス、およびポートが表示されます。cqlshの最初のプロンプトはcqlsh>です。使用するキースペースを指定すると、このプロンプトにキースペースの名前が付加されます。例:

```
$ cqlsh 1.2.3.4 9042 -u jdoe -p mypassword
Connected to trace_consistency at 1.2.3.4:9042.
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.2 | Native protocol v3]
Use HELP for help.
cqlsh>USE mykeyspace;
cqlsh:mykeyspace>
```

cqlshプロンプトでCQLコマンドを入力します。コマンドの終了にはセミコロンを使用します。改行はコマンドの終了を意味しないため、わかりやすくするためにコマンドを複数行にわたって記述することができます。

```
cqlsh> USE demo_cl;
cqlsh:demo_cl> SELECT * FROM demo_table
... WHERE id = 0;
```

コマンドを正常に送信して実行すると、結果が標準出力に送信されます。

```
 id | col1 | col2
----+------+------
  0 |    0 |    0

(1 rows)
```

このリファレンスの前半で説明したコマンドの語彙構造では、コマンドでの大文字および小文字リテラルの扱い、文字列で引用符を使用する場合、および指数表記の入力方法について説明しています。

ファイルへのCQL出力の保存

cqlshコマンドに対して-eオプションに続けてCQL文を引用符で囲んで指定すると、CQL文が受け付けられて実行されます。たとえば、SELECT文の出力をmyoutput.txtに保存する場合は以下のようにします。

```
$ cqlsh -e "SELECT * FROM mytable" > myoutput.txt
```

ファイルを入力として使用

ファイル内のCQLコマンドを実行するには、オペレーティング・システムのコマンドラインで、-fオプションとそのファイルへのパスを指定します。または、cqlshを起動した後、cqlshコマンドラインでSOURCEコマンドとファイルへのパスを指定します。

cqlshrcファイルの作成と使用

cqlshrcファイルが存在する場合は、そのファイルによってデフォルトの構成情報をcqlshに渡すことができます。サンプル・ファイルは以下のようになります。

```
; Sample ~/.cassandra/cqlshrc file.
[authentication]
username = fred
password = !!bang!!
```

Cassandraをインストールすると、confディレクトリーにcqlshrc.sampleファイルが含まれます。Windowsの場合、コマンド・プロンプトでこのファイルを作成するには、cqlshrc.sampleファイルをconfディレクトリーから、自分のユーザー・ホーム・フォルダーである.cassandra隠しフォルダーにコピーし、cqlshrcという名前に変更します。

SSL_CERTFILE環境変数を繰り返しオーバーライドしなくても、cqlshrcファイルを使用してSSL暗号化を構成できます。ユーザーが認証オプションを使用する前に、Cassandra内部認証を構成する必要があります。

関連情報
Cassandra 2.0 cassandra.yaml
Cassandra 2.1 cassandra.yaml

CAPTURE

構文

```
CAPTURE( '<file>' | OFF )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

クエリー出力のキャプチャーを開始するには、現在のディレクトリーからファイルまでの相対パスを指定します。ファイル名は単一引用符で囲んでください。この例に示す略記法は、$HOMEへの参照としてサポートされています。

例

```
CAPTURE '~/mydir/myfile.txt'
```

出力のキャプチャー中、出力はコンソールに表示されません。クエリー結果出力だけがキャプチャーされます。cqlsh専用のコマンドからのエラーと出力は表示されます。

キャプチャーを停止して、出力を通常の表示に戻すには、CAPTURE OFFを使用します。

現在のキャプチャー状態を確認するには、CAPTUREを引数なしで使用します。

CONSISTENCY

構文

```
CONSISTENCY level
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

説明

CONSISTENCYコマンドに引数を与えることにより、デフォルトの整合性レベルONEを変更して、将来の要求に備えて整合性レベルを構成します。有効な値は、ANY、ONE、TWO、THREE、QUORUM、ALL、LOCAL_QUORUM、EACH_QUORUMです。

引数なしの場合は、現在の整合性レベルが表示されます。

例

```
CONSISTENCY
```

デフォルトを変更していない場合、引数なしのCONSISTENCYコマンドの出力は、以下のようになります。

```
Current consistency level is ONE.
```

COPY

構文

```
COPY table_name ( column, ...)
FROM( 'file_name' | STDIN )
WITH option = 'value' AND ...

COPY table_name ( column , ... )
TO( 'file_name' | STDOUT )
WITH option = 'value' AND ...
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

説明

WITH句の中でCOPYのオプションを使用して、CSV形式を変更できます。以下の表でこれらのオプションを説明します。

表 9 : COPYのオプション

COPYのオプション	デフォルト値	使用目的
DELIMITER	コンマ(,)	ファイル内で改行文字を持つフィールドを区切る文字を設定します。
QUOTE	引用符(")	フィールド値を囲む文字を設定します。
ESCAPE	バックスラッシュ(\)	QUOTE文字のリテラル使用をエスケープする文字を設定します。
HEADER	false	ファイルの1行目がヘッダーであることを示すには、trueに設定します。
ENCODING	UTF8	unicode文字列を出力するようにCOPY TOコマンドを設定します。
NULL	空の文字列	値が存在しないことを表します。

COPY FROMコマンドではENCODINGオプションを使用できません。この表に示すように、デフォルトでは、Cassandraは、CSVデータがコンマ(,)で区切られたフィールド、行区切り記号(改行、\r\n)で区切られたレコード、および二重引用符("")で囲まれたフィールド値で構成されることを想定しています。また、あいまいさを避けるために、二重引用符("\"")で囲まれた文字列内で、バックスラッシュを使用してリテラル二重引用符をエスケープします。デフォルトでは、Cassandraは、CSVファイルの1行目に、カラム名で構成されるヘッダー・レコードがあることを想定していません。COPY TOコマンドで、HEADER=trueの場合は、出力にヘッダーが含まれます。COPY FROMコマンドでは、HEADER=trueの場合、1行目が無視されます。

カウンター・テーブルとの間でのデータのコピーはできません。

CSVファイルからのコピー

COPY FROMコマンドを使用する場合、デフォルトでは、Cassandraは、CSV入力の各行のカラム数が同じであることを想定しています。CSV入力のカラム数はCassandraテーブル・メタデータでのカラム数と同じです。Cassandraは対応する順序でフィールドを割り当てます。入力データを特定のカラム・セットに適用するには、テーブル名の後にカラム名を丸かっこで囲んで指定します。

COPY FROMコマンドは小規模のデータセット(数百万行以下)をCassandraにインポートするためのものです。大規模なデータセットのインポートの場合は、Cassandraバルク・ローダーを使用してください。

CSVファイルへのコピー

たとえば、CQLに以下のテーブルがあると仮定します。:

```
cqlsh> SELECT * FROM test.airplanes;
```

```
name           | mach | manufacturer | year
---------------+------+--------------+------
P38-Lightning |  0.7 |     Lockheed | 1937

(1 rows)
```

テーブルにデータを挿入した後、テーブル名の後にカラム名を丸かっこで囲んで指定することにより、そのデータを別の順序でCSVファイルにコピーできます。

```
COPY airplanes
(name, mach, year, manufacturer)
TO 'temp.csv'
```

コピー元ファイルとコピー先ファイルの指定

CSV入力のコピー元ファイルまたはCSV出力のコピー先ファイルをファイル・パスで指定します。あるいは、STDINキーワードを使用して標準入力からインポートするか、STDOUTキーワードを使用して標準出力にエクスポートできます。STDINを使用する場合は、独立の行でバックスラッシュとピリオド("\.")を付けてCSVデータの末尾を示します。すでにデータを含んでいるテーブルにデータをインポートする場合、COPY FROMでは、事前にテーブルが切り詰められません。カラムのセットを部分的にコピーすることができます。テーブル名の後に、インポートまたはエクスポートする順序で、カラム名の全セットまたはサブセットを丸かっこで囲んで指定します。COPY TOコマンドを使用する場合、デフォルトでは、Cassandraは、Cassandraテーブル・メタデータに定義された順序でデータをCSVファイルにコピーします。コピー元のテーブルまたはコピー元のCSVファイルに表示される順序ですべてのカラムをインポートまたはエクスポートする場合は、カラム名のリストを省略することもできます。

簡単なテーブルの往復コピー

テーブルをCSVファイルにコピーします。

1. CQLを使用して、airplanesという名前のテーブルを作成し、それをCSVファイルにコピーします。

```
CREATE KEYSPACE test
WITH REPLICATION = { 'class' :'NetworkTopologyStrategy', 'datacenter1' :
 1 };

USE test;

CREATE TABLE airplanes (
name text PRIMARY KEY,
manufacturer ascii,
year int,
mach float
);

INSERT INTO airplanes
(name, manufacturer, year, mach)
VALUES ('P38-Lightning', 'Lockheed', 1937, '.7');

COPY airplanes (name, manufacturer, year, mach) TO 'temp.csv'

1 rows exported in 0.004 seconds.
```

2. このデータをairplanesテーブルから消去し、このデータをtemp.csvファイルからインポートします。

```
TRUNCATE airplanes;

COPY airplanes (name, manufacturer, year, mach) FROM 'temp.csv';

1 rows imported in 0.087 seconds.
```

標準入力からテーブルにデータをコピーします。

1. COPYコマンドのデフォルト設定を使用して、対話型のcqlshセッション中にデータを直接入力します。

```
COPY airplanes (name, manufacturer, year, mach) FROM STDIN
```

2. [copy]プロンプトの場所に、以下のデータを入力します。

```
"F-14D Super Tomcat", Grumman,"1987", "2.34"
"MiG-23 Flogger", Russian-made, "1964", "2.35"
"Su-27 Flanker", U.S.S.R.,"1981", "2.35"
\.
```

3. airplanesテーブルをクエリーして、STDINからインポートされたデータを表示します。

```
SELECT * FROM airplanes;
```

出力は以下のとおりです。

```
 name                | manufacturer | year | mach
---------------------+--------------+------+-------------
F-14D Super Tomcat |      Grumman | 1987 | 2.35
P38-Lightning |     Lockheed | 1937 | 0.7
Su-27 Flanker |     U.S.S.R. | 1981 | 2.35
MiG-23 Flogger | Russian-made | 1967 | 2.35

(4 rows)
```

コレクションのコピー

Cassandraでは、CSVファイルへの(またはCSVファイルからの)コレクションのコピーをサポートしています。この例を実行するには、今すぐサンプルコードをダウンロードしてください。

1. ダウンロードしたcql_collections.zipファイルを解凍します。
2. すべてのCQLコマンドをcql_collections.txtファイルからコピーしてcqlshコマンドラインにペーストします。
3. songsという名前のテーブルの内容を見てみましょう。このテーブルには、会場のマップ(venue)、レビューのリスト(reviews)、タグのセット(tags)が含まれています。

```
cqlsh> SELECT * FROM music.songs;
```

```
id          |album|artist|data|reviews             |tags              |title|
venue
------------+-----+------+----+--------------------+------------------+-----
+-------------------------------------------------------------------------
7db1a490...| null| null |null|['hot dance music']|          {'rock'}| null|
{'2013-09-22...':'The Fillmore', '2013-10-01...':'The Apple Barrel'}
a3e64f8f...| null| null |null|                null|{'1973', 'blues'}| null|
null
8a172618...| null| null |null|                null|'2007', 'covers'}| null|
null

(3 rows)
```

4. music.songsテーブルをsongs-20140603.csvという名前のCSVファイルにコピーします。

```
cqlsh> COPY music.songs to 'songs-20140603.csv'
```

```
3 rows exported in 0.020 seconds.
```

5. コピー操作が完了したことを確認します。

```
cqlsh> exit;

$ cat songs-20140603.csv
7db1a490...,,,,,['hot dance music'],{'rock'},,"{'2013-09-22...':'The
 Fillmore', '2013-10-01....':'The Apple Barrel'}"
a3e64f8f...,,,,,,"{'1973', 'blues'}",,
8a172618...,,,,,,"{'2007', 'covers'}",,
```

6. 再度cqlshを起動し、songs-204140603ファイル内のデータと一致するテーブル定義を作成します。

```
cqlsh> CREATE TABLE music.imported_songs (
id uuid PRIMARY KEY,
album text,
artist text,
data blob,
reviews list<text<,
tags set<text<,
title text,
venue map<timestamp, text<
       );
```

7. データをCSVファイルからimported_songsテーブルにコピーします。

```
cqlsh> COPY music.imported_songs from 'songs-20140603.csv'

3 rows imported in 0.074 seconds.
```

DESCRIBE

構文

```
DESCRIBE FULL( CLUSTER | SCHEMA )
| KEYSPACES
| ( KEYSPACE keyspace_name )
| TABLES
| ( TABLE table_name )
| TYPES
| ( TYPES user_defined_type )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

説明

DESCRIBEまたはDESCコマンドは、接続されているCassandraクラスター、またはそこに格納されているデータに関する情報を出力します。システム・テーブルに直接クエリーを実行するには、SELECTを使用します。

キースペースおよびテーブル名の引数は大文字と小文字が区別されるため、内部に格納されている名前の大文字または小文字と一致する必要があります。オブジェクトをその内部名ごとにリストする場合はDESCRIBEコマンドを使用します。system_*キースペースのスキーマが必要な場合は、DESCRIBE FULL SCHEMAを使用します。

DESCRIBEは以下のように機能します。

- DESCRIBE CLUSTER

  出力は、接続されているCassandraクラスターに関する情報です(クラスター名、使用中のパーティショナーやスニッチなど)。非システム・キーペースに接続している場合は、このコマンドによってCassandraリングに関するエンドポイント範囲のオーナーシップ情報も表示されます。
- DESCRIBE KEYSPACES

  出力は、すべてのキースペース名のリストです。

- DESCRIBE KEYSPACE *keyspace_name*

  出力は、指定されたキースペースの再作成に使用できるCQLコマンドと、そのキースペースにあるテーブルのリストです。場合によっては、CQLインターフェイスが成熟するにつれ、CQLで表現できないキースペースに関するメタデータが発生する可能性があります。このようなメタデータは表示されません。

  システム以外のキースペースを使用する場合は、<keyspace_name>引数を省略できます。その場合は、現在のキースペースが記述されます。
- DESCRIBE <FULL> SCHEMA

  出力は、ユーザーが作成したスキーマ全体の再作成に使用できるCQLコマンドのリストです。キースペースkごとにDESCRIBE KEYSPACE <k>を呼び出したかのように機能します。システムキースペースを含める場合は、DESCRIBE FULL SCHEMAを使用します。
- DESCRIBE TABLES

  出力は、現在のキースペースにある全テーブルの名前のリスト、または現在のキースペースがない場合は全キースペースにある全テーブルの名前のリストです。
- DESCRIBE TABLE table_name

  出力は、指定のテーブルの再作成に使用できるCQLコマンドのリストです。場合によっては、表現できないテーブル・メタデータがあることがあります。これは表示されません。
- DESCRIBE TYPE type_name

  出力は、type_nameに指定された型のコンポーネントのリストです。
- DESCRIBE TYPES

  出力は、ユーザー定義の型のリストです。

例

```
DESCRIBE CLUSTER;

DESCRIBE KEYSPACES;

DESCRIBE KEYSPACE PortfolioDemo;

DESCRIBE TABLES;

DESCRIBE TABLE Stocks;
```

EXPAND

構文

```
EXPAND( ON | OFF )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

説明

このコマンドは、データの長い行をデフォルトの横方向の書式よりも読みやすくなるように、テーブルの各行を縦方向にリストします。行の続きを見るには、右方向ではなく下方向にスクロールします。カラム1の個別の行に各カラム名が表示され、カラム2に値が表示されます。

EXPAND ONのサンプル出力は以下のとおりです。

```
cqlsh:my_ks> EXPAND ON
Now printing expanded output
cqlsh:my_ks> SELECT * FROM users;


@ Row 1
------------+-----------------------------------------------
userid     | samwise
emails     | {'samwise@gmail.com', 's@gamgee.com'}
first_name | Samwise
last_name  | Gamgee
todo       | {'2013-09-22 12:01:00-0700':'plant trees'}
top_scores | null

@ Row 2
------------+-----------------------------------------------
userid     | frodo
emails     | {'baggins@gmail.com', 'f@baggins.com'}
first_name | Frodo
last_name  | Baggins
todo       | {'2012-10-02 12:00:00-0700':'throw my precious into mount doom'}
top_scores | null

(2 rows)
```

EXIT

構文

```
EXIT | QUIT
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

SHOW

構文

```
SHOW VERSION
| HOST
| SESSION tracing_session_id
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

説明

SHOWコマンドでは、現在のcqlshクライアント・セッションに関する以下の情報が表示されます。

- 接続されているCassandraインスタンスのバージョンおよびビルド番号、cqlshのCQLモード、および接続されているCassandraインスタンスによって使用されるネイティブ・プロトコル・バージョン。
- cqlshセッションが現在接続されているCassandraノードのホスト情報。
- 現在のcqlshセッションのトレース情報。

SHOW SESSIONコマンドは、24時間利用できるトレース・セッション情報を取得します。24時間を経過すると、トレース情報のTime To Liveが期限切れになります。

以下の例は、このコマンドの使用方法を示しています。

```
cqlsh:my_ks> SHOW version
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.2 | Native protocol v3]

cqlsh:my_ks> SHOW host
Connected to Test Cluster at 127.0.0.1:9042.

cqlsh:my_ks> SHOW SESSION d0321c90-508e-11e3-8c7b-73ded3cb6170
```

SHOW SESSIONの出力例は以下のとおりです。

```
Tracing session:d0321c90-508e-11e3-8c7b-73ded3cb6170

activity
                                                                                | timestamp    |
 source    | source_elapsed
-----------------------------------------------------------------------------------------------
+--------------+-----------+----------------
execute_cql3_query | 12:19:52,372 | 127.0.0.1 |              0
Parsing CREATE TABLE emp (\n  empID int,\n  deptID int,\n  first_name varchar,
\n  last_name varchar,\n  PRIMARY KEY (empID, deptID)\n); | 12:19:52,372 |
 127.0.0.1 |            153
Request complete | 12:19:52,372 | 127.0.0.1 |           650
. . .
```

SOURCE

構文

```
SOURCE 'file'
```

構文の凡例

- 大文字はリテラルを意味する

- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

ファイルの内容を実行するには、現在のディレクトリからのファイルの相対パスを指定します。ファイル名は単一引用符で囲んでください。この例に示す略記法は、$HOMEへの参照としてサポートされています。

例

```
SOURCE '~/mydir/myfile.txt'
```

各文の出力は、エラー・メッセージがある場合はそれも含めて順番に表示されます。エラーが発生してもファイルの実行は中止されません。

あるいは、CQLの開始中にファイルを実行するには--fileオプションを使用してください。

TRACING

構文

```
TRACING( ON | OFF )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

説明

読み取り/書き込み要求のトレーシングをオンまたはオフにするには、TRACINGコマンドを使用します。トレーシングをオンにすると、データベース操作があると、Cassandraの内部動作の把握およびパフォーマンス問題のトラブルシュートに役立つ出力が生成されます。たとえば、トレーシング・チュートリアルを使用して、異なる整合性のレベルに応じた操作を確認できます。

Cassandraはトレーシング情報をsystem_tracesキースペースにあるセッションやイベント・テーブルに24時間のあいだ保存して、確率的トレーシングを行う場合にクエリーの対象にできます。確率的トレーシングの詳細については、Cassandra 2.1ドキュメントまたはCassandra 2.0ドキュメントを参照してください。

```
CREATE TABLE sessions (
session_id uuid PRIMARY KEY,
coordinator inet,
duration int,
parameters map<text, text>,
request text,
started_at timestamp
```

```
);

CREATE TABLE events (
session_id uuid,
event_id timeuuid,
activity text,
source inet,
source_elapsed int,
thread text,
PRIMARY KEY (session_id, event_id)
);
```

source_elapsedカラムには、ソース・ノードでイベントが発生するまでの経過時間がミリ秒単位で格納されます。

トレーシング情報を保持するには、セッションおよびイベント・テーブルのデータを他の場所にコピーします。または、トレーシング・セッションIDを指定して、SHOW SESSIONを使用してセッション情報を取得します。トレーシング・セッション情報は1日で有効期限が切れます。

書き込み要求のトレーシング

この例では、ccmによってMac OSX上に作成された3ノード・クラスター上のトレーシング・アクティビティを示します。レプリケーション係数3を持つキースペースと、「複合プライマリ・キーの使用（17ページ）」のemployeeテーブルと同様のテーブルを使用して、トレーシングによって、coordinatorが以下のアクションを実行することを示します。

- 行のレプリケーション先となるノードを特定する。
- 行をcommitlogおよびmemtableに書き込む。
- 要求の完了を確認する。

```
TRACING ON

INSERT INTO emp (empID, deptID, first_name, last_name)
VALUES (104, 15, 'jane', 'smith');
```

Cassandraによって、要求を満たすための各ステップ、影響を受けるノードの名前、各ステップの時間、要求にかかる全時間の詳細が示されます。

```
 Tracing session:740b9c10-3506-11e2-0000-fe8ebeead9ff

activity                                  | timestamp    | source    |
 source_elapsed
-------------------------------------------+--------------+-----------
+----------------
execute_cql3_query | 16:41:00,754 | 127.0.0.1 |              0
Parsing statement | 16:41:00,754 | 127.0.0.1 |             48
Preparing statement | 16:41:00,755 | 127.0.0.1 |            658
Determining replicas for mutation | 16:41:00,755 | 127.0.0.1 |            979
Message received from /127.0.0.1 | 16:41:00,756 | 127.0.0.3 |             37
Acquiring switchLock read lock | 16:41:00,756 | 127.0.0.1 |           1848
Sending message to /127.0.0.3 | 16:41:00,756 | 127.0.0.1 |           1853
Appending to commitlog | 16:41:00,756 | 127.0.0.1 |            1891
Sending message to /127.0.0.2 | 16:41:00,756 | 127.0.0.1 |           1911
Adding to emp memtable | 16:41:00,756 | 127.0.0.1 |            1997
Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.3 |            395
Message received from /127.0.0.1 | 16:41:00,757 | 127.0.0.2 |             42
Appending to commitlog | 16:41:00,757 | 127.0.0.3 |             432
Acquiring switchLock read lock | 16:41:00,757 | 127.0.0.2 |            168
Adding to emp memtable | 16:41:00,757 | 127.0.0.3 |             522
Appending to commitlog | 16:41:00,757 | 127.0.0.2 |             211
Adding to emp memtable | 16:41:00,757 | 127.0.0.2 |             359
Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |           1282
Enqueuing response to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |           1024
Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.3 |           1469
```

```
Sending message to /127.0.0.1 | 16:41:00,758 | 127.0.0.2 |           1179
Message received from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |          10966
Message received from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |          10966
Processing response from /127.0.0.2 | 16:41:00,765 | 127.0.0.1 |
 11063
Processing response from /127.0.0.3 | 16:41:00,765 | 127.0.0.1 |
 11066
Request complete | 16:41:00,765 | 127.0.0.1 |          11139
```

連続スキャンのトレーシング

Cassandraのログ構造のため、1つの行が複数のSSTableに分散することがあります。10行のデータをあらかじめ読み込んでおいたemployeeテーブルを対象とした読み取り要求のトレースに示すように、1つの行を読み取るのに複数のSSTableからそれぞれ一部を読み取ることになります。

```
SELECT * FROM emp;
```

出力は以下のとおりです。

```
empid | deptid | first_name | last_name
------+--------+------------+-----------
110 |     16 |      naoko |     murai
105 |     15 |       john |     smith
111 |     15 |       jane |     thath
113 |     15 |       lisa |     amato
112 |     20 |       mike |     burns
107 |     15 |     sukhit |       ran
108 |     16 |        tom |     brown
109 |     18 |        ann |     green
104 |     15 |       jane |     smith
106 |     15 |        bob |     jones

(10 rows)
```

この読み込み要求のトレーシング出力は、以下のようになります（ページに収めるため、数行を省略しました）。

```
Tracing session:bf5163e0-350f-11e2-0000-fe8ebeead9ff

activity                                                | timestamp    | source    |
 source_elapsed
--------------------------------------------------------+--------------+-----------
+----------------
execute_cql3_query | 17:47:32,511 | 127.0.0.1 |              0
Parsing statement | 17:47:32,511 | 127.0.0.1 |             47
Preparing statement | 17:47:32,511 | 127.0.0.1 |            249
Determining replicas to query | 17:47:32,511 | 127.0.0.1 |            383
Sending message to /127.0.0.2 | 17:47:32,512 | 127.0.0.1 |            883
Message received from /127.0.0.1 | 17:47:32,512 | 127.0.0.2 |             33
Executing seq scan across 0 sstables for . . . | 17:47:32,513 | 127.0.0.2 |
         670
Read 1 live cells and 0 tombstoned | 17:47:32,513 | 127.0.0.2 |            964
Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2 |           1268
Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2 |           1502
Read 1 live cells and 0 tombstoned | 17:47:32,514 | 127.0.0.2 |           1673
Scanned 4 rows and matched 4 | 17:47:32,514 | 127.0.0.2 |           1721
Enqueuing response to /127.0.0.1 | 17:47:32,514 | 127.0.0.2 |           1742
Sending message to /127.0.0.1 | 17:47:32,514 | 127.0.0.2 |           1852
Message received from /127.0.0.2 | 17:47:32,515 | 127.0.0.1 |           3776
```

```
Processing response from /127.0.0.2 | 17:47:32,515 | 127.0.0.1 |
 3900
Sending message to /127.0.0.2 | 17:47:32,665 | 127.0.0.1 |         153535
Message received from /127.0.0.1 | 17:47:32,665 | 127.0.0.2 |             44
Executing seq scan across 0 sstables for . . . | 17:47:32,666 | 127.0.0.2 |
        1068
Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2 |           1454
Read 1 live cells and 0 tombstoned | 17:47:32,667 | 127.0.0.2 |           1640
Scanned 2 rows and matched 2 | 17:47:32,667 | 127.0.0.2 |           1694
Enqueuing response to /127.0.0.1 | 17:47:32,667 | 127.0.0.2 |           1722
Sending message to /127.0.0.1 | 17:47:32,667 | 127.0.0.2 |           1825
Message received from /127.0.0.2 | 17:47:32,668 | 127.0.0.1 |         156454
Processing response from /127.0.0.2 | 17:47:32,668 | 127.0.0.1 |
 156610
Executing seq scan across 0 sstables for . . . | 17:47:32,669 | 127.0.0.1 |
      157387
Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         157729
Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         157904
Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         158054
Read 1 live cells and 0 tombstoned | 17:47:32,669 | 127.0.0.1 |         158217
Scanned 4 rows and matched 4 | 17:47:32,669 | 127.0.0.1 |         158270
Request complete | 17:47:32,670 | 127.0.0.1 |         159525
```

クラスター全体の連続スキャンから、以下のことがわかります。

- 1回目のスキャンでは、ノード2で4行が検出された。
- 2回目のスキャンでは、ノード2でさらに2行が検出された。
- 3回目のスキャンでは、ノード1で4行が検出された。

関連情報

Cassandra 2.1確率的トレーシング

Cassandra 2.0確率的トレーシング

CQLコマンド

ALTER KEYSPACE

構文

```
ALTER( KEYSPACE | SCHEMA) keyspace_name
WITH REPLICATION = map
| ( WITH DURABLE_WRITES =( true | false ))
AND( DURABLE_WRITES =( true | false))
```

mapはマップ・コレクションで、JSON形式のリテラルの配列です。

{ literal :literal , *literal :literal, ...* }

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する

- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

ALTER KEYSPACEは、レプリカ配置ストラテジやDURABLE_WRITES値を定義するマップを変更します。エイリアスであるALTER SCHEMAも使用できます。これらのプロパティと値を使用してマップを作成します。レプリカ配置ストラテジを設定するには、CREATE KEYSPACEリファレンス・ページのマップ・プロパティの表に示されているように、プロパティと値のマップを作成します。CQLプロパティ・マップ・キーは小文字である必要があります。たとえば、classとreplication_factorは正しい表記です。

キースペースの名前を変更することはできません。

例

NetworkTopologyStrategyを1つのデータ・センターで使用するように、mykeyspaceの定義を変更します。Cassandraのデフォルトのデータ・センター名およびレプリケーション係数3を使用します。

```
ALTER KEYSPACE "Excalibur" WITH REPLICATION =
{ 'class' :'NetworkTopologyStrategy', 'datacenter1' : 3 };
```

ALTER TABLE

構文

```
ALTER TABLE keyspace_name.table_name instruction
```

instructionは以下の通りです。

```
ALTER column_name TYPE cql_type
| ( ADD column_name cql_type )
| ( DROP column_name )
| ( RENAME column_name TO column_name )
| ( WITH property AND property ... )
```

cql_typeは元の型と互換性があり、コレクションまたはカウンター以外のCQL型です。例外：ADDはコレクション型をサポートします。また、テーブルがカウンターの場合は、カウンター型もサポートします。

propertyは、speculative_retry = '10ms'のようなCQLテーブル・プロパティおよび値です。文字列プロパティは単一引用符で囲んでください。

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

ALTER TABLEはテーブル・メタデータを操作します。カラムのデータ・ストレージ型を変更したり、新しいカラムを追加したり、既存のカラムを削除したり、テーブル・プロパティを変更したりできます。結果は返されません。エイリアスであるALTER COLUMNFAMILYも使用できます。

最初に、変更するテーブルの名前をALTER TABLEキーワードの後に指定し、その後に変更の種類（ALTER、ADD、DROP、RENAME、またはWITH）を指定します。次に、以下のセクションで説明するように、必要な残りの情報を指定します。

テーブル名は、キースペースで修飾できます。たとえば、monstersキースペースのaddamsFamilyテーブルを変更するには、以下のように記述します。

```
ALTER TABLE monsters.addamsFamily ALTER lastKnownLocation TYPE uuid;
```

カラムの型の変更

カラムのストレージ型を変更するには、元の型が変更後の型と互換性がある必要があります。たとえば、usersテーブルのbioカラムの型をASCIIからテキストに、さらにテキストからBLOBに変更するには、以下のように記述します。

```
CREATE TABLE users (
user_name varchar PRIMARY KEY,
bio ascii,
  );
ALTER TABLE users ALTER bio TYPE text;
ALTER TABLE users ALTER bio TYPE blob;
```

カラムはすでに現在の行に存在する必要があります。そのカラムの値に格納されているバイトは変更されません。既存のデータが新しい型に従ってデシリアライズできない場合は、CQLドライバまたはインターフェイスがエラーを報告する場合があります。

カラム型への以下のような変更は許可されません。

- クラスタ化カラムの型を変更する
- インデックスが定義されているカラムを変更する

データを挿入した後でカラムの型を変更すると、新しい型がデータと互換性がない場合にCQLドライバー/ツールで混乱が生じることがあります。

カラムの追加

コレクション型以外のカラムをテーブルに追加するには、ALTER TABLEとADDキーワードを以下のように使用します。

```
ALTER TABLE addamsFamily ADD gravesite varchar;
```

コレクション型のカラムを追加するには、以下のように使用します。

```
ALTER TABLE users ADD top_places list<text>;
```

既存データは検証されません。

テーブルへの以下のような追加は許可されません。

- 既存のカラムと同じ名前のカラムの追加
- 静的（STATIC）カラム

カラムの削除

テーブルからカラムを削除するには、ALTER TABLEとDROPキーワードを使用します。カラムを削除すると、そのカラムはテーブルから削除されます。

```
ALTER TABLE addamsFamily DROP gender;
```

ALTER DROPは、テーブル定義からカラムを削除し、カラムに対応するデータを削除し、最終的には、そのカラムが使用していたスペースを使用可能にします。削除されたカラムは、すぐにクエリーできなくなります。実際にデータが削除されるのは、コンパクションのときです。データはその後、SSTableに含まれません。削除されたカラムの除去をコンパクションの前に強制的に実行するには、nodetool upgradesstablesコマンドの次にALTER TABLE文を記述します。これにより、テーブルのメタデータが更新されて削除が登録されます。

削除したカラムを再度追加した後、クエリーを実行しても、カラムが削除する前に書き込まれた値は返されません。削除したカラムをテーブルに再度追加する場合、クライアント指定タイムスタンプは使用しないでください。これは、Cassandraが生成した書き込み時間ではありません。

COMPACT STORAGEオプションで定義されたテーブルからはカラムを削除できません。

カラムの名前変更

RENAME句の主な目的は、CQLが生成したプライマリ・キーの名前と、レガシー・テーブルに見つからないカラム名を変更することです。プライマリ・キーのカラム名は変更できます。インデックスが付いたカラムまたは静的（STATIC）カラムの名前は変更できません。これは、Cassandra 2.0.6以降でサポートされています。

テーブル・プロパティの変更

テーブル作成時に確立されたテーブル・ストレージ・プロパティを変更するには、以下のいずれかの形式でテーブルを変更します。

- ALTER TABLE、およびプロパティ名と値を含むWITHディレクティブ
- ALTER TABLE、および次のセクションで示すプロパティ・マップ

WITHディレクティブを使用すると、たとえば、read_repair_chanceプロパティを変更できます。これにより、非クォーラムの整合性を維持するように構成されたクラスターの読み取り時に読み取りリペアを呼び出すための基準が構成されます。

複数のプロパティを変更するには、以下の例のようにANDを使用します。

```
ALTER TABLE addamsFamily
WITH comment = 'A most excellent and useful table'
AND read_repair_chance = 0.2;
```

テキスト・プロパティ値は単一引用符で囲んでください。コンパクト・ストレージを持つテーブルのプロパティは変更できません。

圧縮とコンパクションの変更

圧縮またはコンパクションのオプションを変更するには、プロパティ・マップを使用します。

```
ALTER TABLE addamsFamily WITH compression =
{ 'sstable_compression' :'DeflateCompressor', 'chunk_length_kb' : 64 };

ALTER TABLE mykeyspace.mytable
WITH compaction = {'class':'SizeTieredCompactionStrategy',
 'cold_reads_to_omit': 0.05};
```

キャッシング・プロパティの値を変更します。たとえば、keysオプションをALL（デフォルト）からNONEに変更して、rows_per_partitionを15に変更します。

キャッシングの変更

Cassandra 2.1では、プロパティ・マップを使用してキャッシング・オプションを作成および変更します。

```
//Cassandra 2.1のみ

ALTER TABLE users WITH caching = { 'keys' :'NONE',
 'rows_per_partition' :'15' };
```

次に、rows_per_partitionだけを25に変更します。

```
//Cassandra 2.1のみ

ALTER TABLE users WITH caching = { 'rows_per_partition' :'25' };
```

最後に、テーブル定義を見てみましょう。

```
//Cassandra 2.1のみ

DESCRIBE TABLE users;

CREATE TABLE mykeyspace.users (
user_name text PRIMARY KEY,
bio blob
) WITH bloom_filter_fp_chance = 0.01
AND caching = '{"keys":"NONE", "rows_per_partition":"25"}'
AND comment = ''
AND compaction = {'min_threshold':'4',
 'class':'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
 'max_threshold':'32'}
AND compression =
 {'sstable_compression':'org.apache.cassandra.io.compress.LZ4Compressor'}
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 read_repair_chance = 0.1
AND speculative_retry = '99.0PERCENTILE';
```

Cassandra 2.0.xでは、WITHディレクティブを使用してキャッシング・オプションを変更します。

```
//Cassandra 2.0.xのみ
ALTER TABLE users WITH caching = "keys_only;
```

重要： Cassandra 2.0.xでは、注意して行キャッシングを使用してください。

ALTER TYPE

構文

```
ALTER TYPE name instruction
```

instructionは以下の通りです。

```
ALTER field_name TYPE new_type
| ( ADD field_name new_type )
| ( RENAME field_name TO field_name )
( AND field_name TO field_name ) ...
```

nameはユーザー定義の型の識別子です。

field_nameはそのフィールドに対する任意の識別子です。

new_typeは、予約されている型の名前以外の型の識別子です。

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

ALTER TYPEにより、以下の方法で、ユーザー定義の型を変更できます。

- 既存のフィールドの型を変更する。
- 既存の型の後に新しいフィールドを追加する。
- 型で定義されたフィールドの名前を変更する。
- 同じキースペース内で別の名前を使用してユーザー定義の型の名前を変更する。

まず、ALTER TYPEキーワードの後に、変更するユーザー定義の型の名前を指定し、その後に変更の種類(ALTER、ADD、またはRENAME)を入力します。次に、以下のセクションで説明するように、必要な残りの情報を指定します。

フィールドの型の変更

フィールドの型を変更するには、そのカラムにデータがない場合を除き、そのフィールドが型の定義内にすでに存在し、その型が新しい型と互換性があることが必要です。以下の例では、modelフィールドの型をasciiからtextに変更し、さらにBLOBに変更します。

```
CREATE TYPE version (
model ascii,
version_number int
);
```

```
ALTER TYPE version ALTER model TYPE text;
ALTER TYPE version ALTER model TYPE blob;
```

以下のカラムの型は変更できません。

- クラスター化カラム。これを変更するとディスク上で行の順序付け処理が発生するため
- インデックス付きカラム

フィールドへの型の追加

型に新しいフィールドを追加するには、ALTER TYPEとADDキーワードを使用します。

```
ALTER TYPE version ADD release_date timestamp;
```

以下の例でpoint_releaseという名前のコレクション・マップ・フィールド(リリース日と10進数指定子を表す)を追加するには、以下の構文を使用します:

```
ALTER TYPE version ADD point_release map<timestamp, decimal>;
```

型のフィールドの名前変更

ユーザー定義の型のフィールド名を変更するには、RENAME old_name TO new_name構文を使用します。古い名前と新しい名前に対して、異なるキースペース・プレフィックスを使用します。型のフィールド名に対して複数の変更を行うには、そのコマンドの後にAND old_name TO new_nameを追加します。

```
ALTER TYPE version RENAME model TO sku;
ALTER TYPE version RENAME sku TO model AND version_number TO num
```

ユーザー定義の型の名前変更

型の名前を変更するには、以下の例に示すように、RENAMEを使用します。

```
ALTER TYPE version RENAME TO major_release;
```

ALTER USER

構文

```
ALTER USER user_name
WITH PASSWORD 'password'( NOSUPERUSER | SUPERUSER )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

スーパーユーザーは、ユーザーのパスワードまたはスーパーユーザー・ステータスを変更できます。すべてのスーパーユーザーが無効にされるのを防ぐために、スーパーユーザーは自分のスーパーユーザー・ステータスを変更することはできません。一般ユーザーが変更できるのは、自分のパスワードのみです。ユーザー名に英数字以外の文字が含まれている場合は、ユーザー名を単一引用符で囲んでください。パスワードは単一引用符で囲んでください。

例

```
ALTER USER moss WITH PASSWORD 'bestReceiver';
```

BATCH

構文

Cassandra 2.1以降

```
BEGIN UNLOGGED  BATCH
USING TIMESTAMP timestamp
dml_statement;
dml_statement;
  ...
APPLY BATCH;
```

Cassandra 2.0.x

```
BEGIN( UNLOGGED | COUNTER) BATCH
USING TIMESTAMP timestamp
dml_statement;
dml_statement;
  ...
APPLY BATCH;
```

dml_statementは以下の通りです。

- INSERT
- UPDATE
- DELETE

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

BATCH文は、複数のデータ操作言語(DML)文(INSERT、UPDATE、DELETE)を1つの論理演算にまとめた上で、各文によって書き込まれたすべてのカラムに対しクライアント指定タイムスタンプを一括して設定します。複数の文をバッチ化することで、クライアント/サーバーとサーバー・コーディネーター/レプリカ間のネットワーク交換を削減できます。しかし、Cassandraは分散型なので、パフォーマンスを最適化するために、できるだけ多く要求を近くのノードに分散させてください。「バッチの使用と誤使用」セクションで説明されているように、パフォーマンスを最適化することを目的としてバッチを使用しても、通常は成功しません。データを読み込む最も速い方法については、「Cassandra: Batch loading without the Batch keyword」(英語)を参照してください。

バッチは、デフォルトではアトミックです。Cassandraバッチ処理の場合、アトミックとは、バッチの一部が成功すれば、バッチ全体が成功することを意味します。アトミック性を実現するために、Cassandraは、まず、シリアライズされたバッチを、そのバッチをBLOBデータとして消費するバッチログ・システム・テーブルに書き込みます。バッチ内の各行が正しく書き込まれ、継続性が維持(ヒント)されたときにバッチログ・データが削除されます。アトミック性には、パフォーマンス上のペナルティがあります。このペナルティを発生させたくない場合は、Cassandraがバッチログ・システムに書き込まないように、UNLOGGEDオプションを使用します:BEGIN UNLOGGED BATCH。

アトミック・バッチは、バッチの一部が成功するとバッチ全体が成功することを保証しますが、その他のトランザクション制御はバッチ・レベルで実行しません。たとえば、バッチ隔離は行いません。クライアントは、サーバーで他の行がまだ更新中であっても、バッチによって最初に更新された行を読み取ることができます。ただし、パーティション・キー内でのトランザクショナルな行更新は隔離されているので、クライアントは、部分的な更新を読み取ることはできません。

バッチ内の文の順序は関係ありません。Cassandraは、同じタイムスタンプを使用してすべての行を適用します。特定の順序で実行するには、クライアント指定タイムスタンプを使用する必要があります。

タイムスタンプの使用

BATCH文では、1つの例外を除いて、USING句にクライアント指定タイムスタンプ（整数）を設定できます。バッチ内のDML文に、以下のような比較して設定する（compare-and-set：CAS）文が含まれている場合は、タイムスタンプを使用しないでください。

```
INSERT INTO users (id, lastname) VALUES (999, 'Sparrow')  IF NOT EXISTS
```

タイムスタンプは、バッチ内のすべての文に適用されます。指定しないと、挿入時の時刻（単位はマイクロ秒）が使用されます。USING句を使用してタイムスタンプを指定しない場合は、BATCH文内の個々のDML文にタイムスタンプを指定できます。

たとえば、INSERT文にタイムスタンプを指定します。

```
BEGIN BATCH
INSERT INTO purchases (user, balance) VALUES ('user1', -8) USING TIMESTAMP
 19998889022757000;
INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;
```

balanceカラムにクライアント指定タイムスタンプが入っていることを確認します。

```
SELECT balance, WRITETIME(balance) FROM PURCHASES;
```

```
 balance | writetime_balance
---------+-------------------
      -8 | 19998889022757000
```

条件付き更新のバッチ処理

Cassandra 2.0.6以降では、Cassandra 2.0で軽量トランザクションとして導入された条件付き更新をバッチ処理できます。基礎となっているPaxos実装は、パーティションの粒度で動作するので、バッチに含めることができるのは、同じパーティションへの更新のみです。条件付き更新と無条件更新を1つのグループにまとめることはできますが、バッチ内に1つでも条件付き更新の文があれば、バッチ内のすべての条件が適用されるかのように、バッチ全体が1つのPaxosプロポーザルを使用してコミットされます。以下の例では、条件付き更新のバッチ処理を示します。

購入レコードに値を挿入するための文でIF条件句を使用しています。

```
BEGIN BATCH
INSERT INTO purchases (user, balance) VALUES ('user1', -8) IF NOT EXISTS;
INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 1, 8, 'burrito', false);
APPLY BATCH;

BEGIN BATCH
UPDATE purchases SET balance = -208 WHERE user='user1' IF balance = -8;
INSERT INTO purchases (user, expense_id, amount, description, paid)
VALUES ('user1', 2, 200, 'hotel room', false);
APPLY BATCH;
```

この例の続きに、バッチ内で静的（STATIC）カラムを条件付き更新と一緒に使用する方法が示されています。

カウンター更新のバッチ処理

Cassandra 2.1以降では、カウンター更新は、Cassandraの他の書き込みとは異なりべき等演算ではないので、カウンターのバッチにUNLOGGEDを使用する必要があります。

Cassandra 2.0では、バッチ方式カウンター更新のために、バッチ文にBEGIN COUNTER BATCHを使用します。

Cassandra 2.1の例

```
BEGIN UNLOGGED BATCH
UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

Cassandra 2.0の例

```
BEGIN COUNTER BATCH
UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
UPDATE AdminActionCounts SET total = total + 2 WHERE keyalias = 701;
APPLY BATCH;
```

CREATE INDEX

構文

```
CREATE CUSTOM INDEX IF NOT EXISTS index_name
ON keyspace_name.table_name ( KEYS ( column_name ) )
(USING class_name) (WITH OPTIONS = map)
```

制限事項: *USING class_name*は、CUSTOMを使用しており、かつclass_nameがjavaクラス名を含む文字列リテラルである場合のみ許可されます。

index_nameは、予約語以外の識別子であり、二重引用符で囲む場合と囲まない場合があります。

mapはマップ・コレクションで、JSON形式のリテラルの配列です。

```
{ literal :literal, literal :literal ... }
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

CREATE INDEXは、指定されたテーブルに、指定されたカラムの新しいインデックスを作成します。すでに存在しているインデックスを作成しようとすると、IF NOT EXISTSオプションを使用していない限りエラーが返されます。このオプションを使用すると、すでにそのインデックスが存在していれば、この文は何もしません。任意で、ONキーワードの前にインデックスに付ける名前を指定できます。1つのカラム名を丸かっこで囲んで指定します。カラムは現在ある行に存在する必要はありません。テーブルを作成するとき、または後でテーブルを変更して追加するときに、カラムとそのデータ型を指定する必要があります。

ドット表記を使用して、そのテーブルのキースペースを指定できます。キースペース名の後にピリオドを付け、その後にテーブルの名前を記述します。Cassandraは、指定されたキースペースにそのテーブルを作成しますが、現在のキースペースは変更しません。キースペース名を使用しない場合は、現在のキースペース内にそのテーブルのインデックスを作成します。

カラムにデータがすでに存在している場合、Cassandraはこの文の実行中にデータのインデックスを作成します。インデックスを作成した後、Cassandraは、新しいデータの挿入時にカラムの新しいデータのインデックスを自動的に作成します。

Cassandraでは、複合プライマリ・キーのクラスター化カラムを含むほとんどのカラム、またはパーティション（プライマリ）キー自身を対象にインデックスを作成できます。Cassandra 2.1以降では、コレクション、およびコレクション・マップのキーを対象にインデックスを作成できます。コレクションのキーと値の両方を対象にインデックスを作成しようとすると拒否されます。

インデックスの作成はパフォーマンスに大きな影響を与える可能性があります。インデックスを作成する前に、インデックスを作成すべき場合とすべきでない場合について把握しておいてください。

カウンター・カラムのインデックスは作成できません。

Cassandraでは、主に内部使用を目的としたカスタム・インデックスの作成と、カスタム・インデックスに適用するオプションの指定ができます。例:

```
CREATE CUSTOM INDEX ON users (email) USING 'path.to.the.IndexClass';
CREATE CUSTOM INDEX ON users (email) USING 'path.to.the.IndexClass' WITH
 OPTIONS = {'storage':'/mnt/ssd/indexes/'};
```

カラムを対象としたインデックスの作成

テーブルを定義し、その2つのカラムにインデックスを作成します。

```
CREATE TABLE myschema.users (
userID uuid,
fname text,
lname text,
email text,
address text,
zip int,
state text,
PRIMARY KEY (userID)
 );

CREATE INDEX user_state
ON myschema.users (state);

CREATE INDEX ON myschema.users (zip);
```

クラスター化カラムを対象としたインデックスの作成

複合パーティション・キーを持つテーブルを定義し、クラスター化カラムにインデックスを作成します。

```


CREATE TABLE mykeyspace.users (
userID uuid,
fname text,
lname text,
email text,
address text,
zip int,
state text,
PRIMARY KEY ((userID, fname), state)
);

CREATE INDEX ON mykeyspace.users (state);
```

コレクションでのインデックスの作成

Cassandra 2.1以降では、他のカラムと同じように、コレクション・カラムにインデックスを作成します。CREATE INDEX文の末尾でコレクション・カラムの名前を丸かっこで囲んで指定します。たとえば、電話番号のコレクションをユーザー・テーブルに追加して、phonesセットのデータのインデックスを作成します。

```
ALTER TABLE users ADD phones set<text>;
CREATE INDEX ON users (phones);
```

コレクションがマップの場合、Cassandraは、マップ値にインデックスを作成します。usersテーブルに、todoマップの例にある以下のマップ・データが含まれていると仮定します。

```
{'2014-10-2 12:10' :'die' }
```

マップ値はコロンの右側にあります。ここでは'die'です。マップ・キー、つまりタイムスタンプは、コロンの左側にあります。わずかに異なる構文を使用して、インデックスをマップ・キーに作成することもできます。

マップ・キーを対象としたインデックスの作成

Cassandra 2.1以降では、マップ・コレクション・キーにインデックスを作成できます。

マップ・キーのインデックスを作成するには、ネストした丸かっこの中にKEYSキーワードとマップ名を入れます。たとえば、usersテーブル内のtodoマップに、コレクション・キー、つまりタイムスタンプのインデックスを作成します。

```
CREATE INDEX todo_dates ON users (KEYS(todo));
```

テーブルをクエリーするには、WHERE句でCONTAINS KEYを使用します。

CREATE KEYSPACE

構文

```
CREATE( KEYSPACE | SCHEMA) IF NOT EXISTS keyspace_name
WITH REPLICATION = map
AND DURABLE_WRITES =( true | false )
```

mapはマップ・コレクションで、JSON形式のリテラルの配列です。

```
{ literal :literal, literal :literal ... }
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

CREATE KEYSPACEは、最上位レベルの名前空間を作成し、キースペース名、レプリカ配置ストラテジ・クラス、レプリケーション係数、およびキースペースのDURABLE_WRITESオプションを設定します。レプリカ配置ストラテジの詳細については、「Cassandra 2.1のレプリカ配置ストラテジ」または「Cassandra 2.0のレプリカ配置ストラテジ」を参照してください。

NetworkTopologyStrategyをレプリケーション・ストラテジとして構成する場合は、1つ以上の仮想データ・センターを設定します。あるいは、デフォルトのデータ・センターを使用します。データ・センターにはスニッチで使用されている名前と同じ名前を使用します。スニッチの詳細については、Cassandra 2.1のスニッチのドキュメントまたはCassandra 2.0のスニッチのドキュメントを参照してください。

ワークロードの種類に応じて、異なるノードを別々のデータ・センターに割り当てます。たとえば、Hadoopノードを1つのデータ・センターに割り当て、Cassandraリアルタイム・ノードを別のデータ・センターに割り当てます。ワークロードを分離することで、データ・センターごとに1種類のワークロードだけがアクティブになります。分離によって、パフォーマンスに影響を与えるさまざまなバッチ要件など、ワークロード間の干渉の問題を防止できます。

以下のプロパティおよび値のマップは、2種類のキースペースを定義します。

```
{ 'class' :'SimpleStrategy', 'replication_factor' :<integer> };

{ 'class' :'NetworkTopologyStrategy'[, '<data center>' :<integer>, '<data
 center>' :<integer>] . . . };
```

表 **10 :** マップ・プロパティと値の表

プロパティ	値	値の説明
'class'	'SimpleStrategy'または'NetworkTopologyStrategy'	必須。新しいキースペースのレプリカ配置ストラテジ・クラスの名前です。
'replication_factor'	<レプリカ数>	クラスがSimpleStrategyの場合は必須ですが、それ以外の場合は使用しません。複数のノードにあるデータのレプリカの数です。
'<最初のデータ・センター名>'	<レプリカ数>	クラスがNetworkTopologyStrategyであり、最初のデータ・センターの名前を指定した場合は必須です。この値は、最初のデータ・センターの各ノードにあるデータのレプリカの数です。例
'<次のデータ・センター名>'	<レプリカ数>	クラスがNetworkTopologyStrategyであり、2番目のデータ・センターの名前を指定した場合は必須です。この値は、データ・センターの各ノードにあるデータのレプリカの数です。
. . .	. . .	任意で、指定のデータ・センターのレプリケーション係数を複数指定できます。

CQLプロパティ・マップ・キーは小文字である必要があります。たとえば、classとreplication_factorは正しい表記です。キースペース名は32文字以下の英数字とアンダースコアであり、名前の先頭は英字です。キースペース名は大文字と小文字が区別されません。名前の大文字と小文字を区別させるには、名前を二重引用符で囲みます。

CREATE KEYSPACEの代わりに、エイリアスであるCREATE SCHEMAを使用できます。すでに存在しているキースペースを作成しようとすると、IF NOT EXISTSオプションを使用していない限りエラーが返されます。このオプションを使用すると、すでにそのキースペースが存在していれば、この文は何もしません。

SimpleStrategyクラスの設定の例

CREATE KEYSPACE文を作成するには、まず、キースペースの名前の後にWITH REPLICATIONキーワードと等号を付けて宣言します。二重引用符で囲まれていないキースペースの名前は、大文字と小文字が区別されません。次に、複数のデータ・センターに最適化されていないキースペースを作成するには、マップのclassの値にSimpleStrategyを指定します。replication_factorプロパティをコロンで区切り、中かっこで囲んで設定します。例:

```
CREATE KEYSPACE Excelsior
WITH REPLICATION = { 'class' :'SimpleStrategy', 'replication_factor' : 3 };
```

Cassandraを評価するに当たってはSimpleStrategyでも十分です。実稼働環境で使用する場合、または混在ワークロードで使用する場合は、NetworkTopologyStrategyを使用します。

NetworkToplogyStrategyクラスの設定の例

NetworkTopologyStrategyもCassandraの評価に役立ちます。単一ノードのクラスターなどを使用した評価を目的としてNetworkTopologyStrategyを使用するには、クラスターのデフォルトのデータ・センター名を指定します。デフォルトのデータ・センター名を調べるには、nodetool statusを使用します。

```
$ nodetool status
Datacenter:datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens  Owns    Host ID
  Rack
UN  127.0.0.1  46.59 KB   256     100.0%  dd867d15-6536-4922-b574-e22e75e46432
  rack1
```

Cassandraは、デフォルトのデータ・センター名としてdatacenter1を使用します。単一ノードのクラスターにNTSkeyspaceという名前のキースペースを作成する例を以下に示します。

```
CREATE KEYSPACE NTSkeyspace WITH REPLICATION =
 { 'class' :'NetworkTopologyStrategy', 'datacenter1' : 1 };
```

実稼働環境でデータ・センターにNetworkTopologyStrategyを使用するには、デフォルトのスニッチであるSimpleSnitchをネットワーク対応スニッチに変更して、スニッチ・プロパティ・ファイルで1つ以上のデータ・センター名を定義したら、これらのデータ・センター名を使用してキースペースを定義する必要があります。これを行わないと、Cassandraは、テーブルへのデータの挿入などの書き込み要求を完了するためにノードを検索できません。

PropertyFileSnitchなどのネットワーク対応スニッチを使用するようにCassandraを構成したら、cassandra-topology.propertiesファイルでデータ・センターとラックの名前を定義します。

マップのclassの値にNetworkTopologyStrategyを使用するCREATE KEYSPACE文を作成します。データ・センター名とデータ・センターあたりのレプリカの数で構成される1つ以上のキーと値のペアを設定します。これはコロンで区切り、中かっこで囲んで設定します。例:

```
CREATE KEYSPACE "Excalibur"
WITH REPLICATION = {'class' :'NetworkTopologyStrategy', 'dc1' :3, 'dc2' : 2};
```

この例では、dc1という名前のデータ・センターの3つのレプリカと、dc2という名前のデータ・センターの2つのレプリカを設定しています。使用するデータ・センター名は、使用しているクラスター構成されたスニッチによって異なります。マップで定義されたデータ・センター名と、使用しているスニッチによって認識されたデータ・センター名の間には相関関係があります。データ・センター名とノードのラック位置が不明な場合は、nodetool statusコマンドでそれらを出力します。

DURABLE_WRITESの設定

CREATE KEYSPACEコマンドのマップ指定の後にDURABLE_WRITESオプションを設定できます。falseに設定すると、キースペースに書き込まれたデータはコミット・ログに記録されません。データを喪失する危険性があるため、このオプションを使用する際は気を付けてください。SimpleStrategyを使用して、キースペースでこの属性を設定しないでください。

```
CREATE KEYSPACE Risky
WITH REPLICATION = { 'class' :'NetworkTopologyStrategy',
'datacenter1' :3 } AND durable_writes = false;
```

作成したキースペースの確認

キースペースが作成されたことを確認します。

```
SELECT * FROM system.schema_keyspaces;
```

```


keyspace_name   | durable_writes | strategy_class
          | strategy_options
---------------+----------------
+-------------------------------------------------------
+---------------------------
excelsior |           True |
 org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"3"}
Excalibur |           True |
 org.apache.cassandra.locator.NetworkTopologyStrategy |
 {"dc2":"2","dc1":"3"}
risky |          False | org.apache.cassandra.locator.NetworkTopologyStrategy
 |        {"datacenter1":"1"}
system |           True |           org.apache.cassandra.locator.LocalStrategy
 |                        {}
system_traces |           True |
 org.apache.cassandra.locator.SimpleStrategy | {"replication_factor":"1"}

(5 rows)
```

Cassandraは、キースペースの作成に引用符が使用されていなかったためexcelsiorキースペースを小文字に変換し、Excaliburについては引用符が使用されていたための先頭の大文字を維持しました。

関連情報

Cassandra 2.1レプリケーション・ストラテジ

Cassandra 2.0レプリケーション・ストラテジ

Cassandra 2.1スニッチ構成

Cassandra 2.0スニッチ構成

Cassandra 2.1プロパティ・ファイル・スニッチ

Cassandra 2.0プロパティ・ファイル・スニッチ

CREATE TABLE

構文

```
CREATE TABLE IF NOT EXISTS keyspace_name.table_name
( column_definition, column_definition, ...)
WITH property AND property ...
```

以下はCassandra 2.1 column_definitionです。

```
column_name cql_type STATIC PRIMARY KEY
| column_name frozen<tuple<tuple_type> tuple<tuple_type>...> PRIMARY KEY
| column_name frozen<user-defined_type> PRIMARY KEY
| ( PRIMARY KEY ( partition_key ) )
```

以下はCassandra 2.0.x column_definitionです。

```
column_name cql_type STATIC PRIMARY KEY
| ( PRIMARY KEY ( partition_key ) )
```

制限事項:

- プライマリ・キー定義は常に1つだけ存在する必要があります。
- プライマリ・キーのcql_typeは、CQLデータ型またはユーザー定義の型でなければなりません。
- コレクションのcql_typeは、以下の構文を使用します。

  ```
  LIST<cql_type>
  | SET<cql_type>
  | MAP<cql_type, cql_type>
  ```
- Cassandra 2.1の場合のみ、タプル型およびユーザー定義の型には、*frozen*キーワードとその後に山かっこで囲んだ型が必要です。

以下はPRIMARY KEYです。

```
column_name
| ( column_name1, column_name2, column_name3 ...)
| ((column_name4, column_name5), column_name6, column_name7 ...)
```

column_name1はパーティション・キーです。

column_name2, column_name3 ...はクラスター化カラムです。

column_name4, column_name5はパーティション・キーです。

column_name6, column_name7 ...はクラスター化カラムです。

propertyは、文字列の場合は単一引用符で囲まれているCQLテーブル・プロパティか、または以下のいずれかのディレクティブです。

```
COMPACT STORAGE
| ( CLUSTERING ORDER BY (clustering_column( ASC) | DESC ), ...) )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

CREATE TABLEは、現在のキースペースの下に新しいテーブルを作成します。エイリアスであるCREATE COLUMNFAMILYを使用することもできます。

既存のテーブルを作成しようとすると、IF NOT EXISTSオプションを使用していない限りエラーが返されます。このオプションを使用すると、すでにそのテーブルが存在していれば、この文は何もしません。

有効なテーブル名は、英数字とアンダースコアで構成され、英字で始まる文字列です。ドット表記を使用して、そのテーブルのキースペースを指定できます。キースペース名の後にピリオドを付け、その後にテーブルの名前を記述します。これにより、指定したキースペース内にそのテーブルが作成されますが、現在のキースペースは変更されません。キースペース名を指定しなければ、現在のキースペース内にテーブルが作成されます。

Cassandra 2.0.6以降では、静的（STATIC）カラムを使用して、同じデータをパーティションのクラスター化された複数の行に格納し、1つのSELECT文でそのデータを取得できます。

Cassandra 2.1で改善されたカウンター・カラムをテーブルに追加できます。

カラムの定義

テーブル作成の際にカラムに型を割り当てます。コレクション型カラムを除くカラム型は、カラム名と型のペアを、丸かっこで囲んだコンマ区切りリストとして指定します。

以下の例は、コレクション型カラムのマップ、セット、リストを含むテーブルの作成方法を示しています。

```
CREATE TABLE users (
userid text PRIMARY KEY,
first_name text,
last_name text,
emails set<text>,
top_scores list<int>,
todo map<timestamp, text>
);
```

ユーザー定義またはタプル型のカラムの定義

将来の機能をサポートするために、ユーザー定義またはタプル型のカラムの定義には*frozen*キーワードが必要です。Cassandraは、複数のコンポーネントを持つfrozen値を1つの値にシリアライズします。例と使用方法については、「ユーザー定義の型の使用」と「タプル型」を参照してください。

結果の並べ替え

クエリー結果に順序を指定することで、ディスク上でのカラムのソート順を利用できます。結果を昇順または降順に並べることができます。昇順の方が降順より効率的になります。結果を降順にする必要がある場合は、デフォルトの逆の順序でカラムをディスクに格納するようにクラスター化順序を指定できます。そうした場合、クエリーを降順にすると昇順より高速になります。

以下の例は、クラスター化順序を挿入時間の降順に変更するテーブル定義を示しています。

```
CREATE TABLE timeseries (
event_type text,
insertion_time timestamp,
event blob,
PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

コンパクト・ストレージの使用

複合プライマリ・キーを使用してテーブルを作成する場合は、格納するデータごとにカラム名も一緒に格納する必要があります。各カラムがディスク上の1つのカラムに対応するように各非プライマリ・キー・カラムを格納するのではなく、行全体をディスク上の1つのカラムに格納します。ディスク領域を節約する必要がある場合は、データを従来の（Thrift）ストレージ・エンジン形式で保存するWITH COMPACT STORAGEディレクティブを使用します。

```
CREATE TABLE sblocks (
block_id uuid,
subblock_id uuid,
```

```
data blob,
PRIMARY KEY (block_id, subblock_id)
)
WITH COMPACT STORAGE;
```

コンパクト・ストレージ・ディレクティブを使用することで、複合プライマリ・キーに属さないカラムを複数定義することができなくなります。複合ではないプライマリ・キーを使用する圧縮テーブルは、プライマリ・キーに属さないカラムを複数持つことができます。

複合プライマリ・キーを使用する圧縮テーブルは、クラスター化カラムを少なくとも1つ定義する必要があります。圧縮テーブルの作成後は、カラムを追加または削除することはできません。WITH COMPACT STORAGEを指定しないと、CQLでは非コンパクト・ストレージでテーブルを作成します。

テーブル・プロパティの設定

任意指定のWITH句とキーワード引数を使用して、Cassandraが新しいテーブルで実行するキャッシング、コンパクション、および他の多くの操作を構成できます。WITH句を使用して、テーブル・プロパティの設定（93ページ）の一覧に示したテーブルのプロパティを指定します。文字列プロパティは単一引用符で囲んでください。

プライマリ・キー・カラムの定義

テーブルについて定義する必要がある最低限のスキーマ情報は、プライマリ・キーとそれに関連付けられるデータ型です。以前のバージョンとは異なり、CQLではプライマリ・キーに属さないカラムがテーブルに必要ありません。プライマリ・キーには、任意の数（1つ以上）のコンポーネント・カラムを含めることができます。

プライマリ・キーが1つのカラムだけで構成される場合は、カラム定義の後にキーワードPRIMARY KEYを使用できます。

```
CREATE TABLE users (
user_name varchar PRIMARY KEY,
password varchar,
gender varchar,
session_token varchar,
state varchar,
birth_year bigint
);
```

あるいは、1つのカラムだけで構成されるプライマリ・キーを、複合プライマリ・キーと同じ方法で宣言することもできます。キーにはカウンター・カラムを使用しないでください。

テーブル・プロパティの設定

任意指定のWITH句とキーワード引数を使用して、Cassandraが新しいテーブルで実行するキャッシング、コンパクション、および他の多くの操作を構成できます。WITH句を使用して、キャッシング、テーブル・コメント、圧縮、コンパクションなど、CQLテーブル・プロパティの一覧に示したテーブルのプロパティを指定できます。プロパティを文字列またはマップのいずれかとして設定します。文字列プロパティは単一引用符で囲んでください。たとえば、テーブルにコメントを埋め込むには、コメントを文字列プロパティとしてフォーマットします。

```
CREATE TABLE MonkeyTypes (
block_id uuid,
species text,
alias text,
population varint,
PRIMARY KEY (block_id)
)
WITH comment='Important biological records'
AND read_repair_chance = 1.0;
```

圧縮およびコンパクションを構成するには、プロパティ・マップを使用します。

```
CREATE TABLE DogTypes (
block_id uuid,
species text,
alias text,
population varint,
PRIMARY KEY (block_id)
) WITH compression =
{ 'sstable_compression' :'DeflateCompressor', 'chunk_length_kb' : 64 }
AND compaction =
{ 'class' :'SizeTieredCompactionStrategy', 'min_threshold' : 6 };
```

圧縮ストレージまたはクラスター化順序の使用を指定するには、WITH句を使用します。

Cassandra 2.1でキャッシングを構成する場合も、プロパティ・マップを使用します。

```
// Cassandra 2.1

CREATE TABLE DogTypes (
...   block_id uuid,
...   species text,
...   alias text,
...   population varint,
...   PRIMARY KEY (block_id)
... ) WITH caching = '{ 'keys' :'NONE', 'rows_per_partition' :'120' }';
```

Cassandra 2.0.xでキャッシングを構成するには、プロパティ・マップを使用しません。cachingプロパティに値を設定するだけです。

```
// Cassandra 2.0.xのみ

CREATE TABLE DogTypes (
block_id uuid,
species text,
alias text,
population varint,
PRIMARY KEY (block_id)
) WITH caching = 'keys_only';
```

重要： Cassandra 2.0.xでは、注意して行キャッシングを使用してください。

複合プライマリ・キーの使用

音楽サービスの例に示すように、複合プライマリ・キーは複数のカラムで構成され、定義で宣言されている最初のカラムをパーティション・キーとして扱います。複合プライマリ・キーを作成するには、キーワードPRIMARY KEYの後に、コンマで区切られたカラム名のリストを指定して丸かっこで囲みます。

```
CREATE TABLE emp (
empID int,
deptID int,
first_name varchar,
last_name varchar,
PRIMARY KEY (empID, deptID)
);
```

複合パーティション・キーの使用

複合パーティション・キーは、複数のカラムで構成されるパーティション・キーです。複合パーティション・キーを構成するカラムをもう1組の丸かっこで囲みます。プライマリ・キー定義に含まれているけれども、ネストされた丸かっこの

外側にあるカラムがクラスター化カラムです。これらのカラムは、取得を容易にするためにパーティション内で論理集合を形成します。

```
CREATE TABLE Cats (
block_id uuid,
breed text,
color text,
short_hair boolean,
PRIMARY KEY ((block_id, breed), color, short_hair)
);
```

たとえば、この複合パーティション・キーはblock_idとbreedで構成されています。クラスター化カラムcolorおよびshort_hairによって、データのクラスター化順序が決まります。一般に、Cassandraでは同じblock_idで異なるbreedを持つカラムを異なるノードに格納し、同じblock_idとbreedを持つカラムを同じノードに格納します。

クラスター化順序の使用

クエリー結果に順序を指定することで、ディスク上でのカラムのソート順を利用できます。結果を昇順または降順に並べることができます。昇順の方が降順より効率的になります。結果を降順にする必要がある場合は、デフォルトの逆の順序でカラムをディスクに格納するようにクラスター化順序を指定できます。そうした場合、クエリーを降順にすると昇順より高速になります。

以下の例は、クラスター化順序を挿入時間の降順に変更するテーブル定義を示しています。

```
create table timeseries (
event_type text,
insertion_time timestamp,
event blob,
PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

静的(**STATIC**)カラムの共有

クラスター化カラムを使用するテーブルでは、テーブル定義で非クラスター化カラムを静的(STATIC)なカラムとして宣言できます。静的カラムは特定のパーティション内だけで静的になります。

```
CREATE TABLE t (
k text,
s text STATIC,
i int,
PRIMARY KEY (k, i)
);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m shared', 0);
INSERT INTO t (k, s, i) VALUES ('k', 'I''m still shared', 1);
SELECT * FROM t;
```

出力は以下のとおりです。

```
 k |                   s | i
---------------------------
k  | "I'm still shared" | 0
k  | "I'm still shared" | 1
```

制限事項

- クラスター化カラムが定義されていないテーブルに静的カラムを含めることはできません。クラスター化カラムがないテーブルには、すべてのカラムが本質的に静的である1行パーティションで構成されます。
- COMPACT STORAGEディレクティブを使用して定義されたテーブルに静的カラムを含めることはできません。
- パーティション・キーとして指定されるカラムを静的カラムにすることはできません。

静的カラムに対する条件付き更新をバッチ処理できます。

Cassandra 2.0.9以降では、DISTINCTキーワードを使用して静的カラムを検索できます。この場合は、パーティションの最初（静的カラム）だけが取得されます。

CREATE TRIGGER

構文

```
CREATE TRIGGER trigger_name ON table_name
USING 'java_class'
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ（|）記号はORまたはAND/ORを意味する
- 省略記号（...）は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

トリガーの実装には、なじみのあるCREATE TRIGGER構文を使用してトリガーをテーブルに登録する機能が含まれます。この実装は実験的に行われています。

```
CREATE TRIGGER myTrigger
ON myTable
USING 'org.apache.cassandra.triggers.InvertedIndex'
```

Cassandra 2.1の場合、大文字を使ったトリガー名を単一引用符で囲む必要があります。トリガーを構成する実際のロジックは、どのJava（JVM）言語でも記述でき、データベースの外に存在します。トリガーを実装するこの例のJavaクラスは、「org.apache.cassandra.triggers」という名前で、Apacheリポジトリ内で定義されています。トリガーのコードをCassandraインストール・ディレクトリーのlib/triggersサブディレクトリーに置くと、クラスターの起動時に読み込まれ、クラスターに参加する各ノードに存在するようになります。テーブルに定義されたトリガーは要求されたDML文が実行される前に作動し、トランザクションのアトミック性を保証します。

CREATE TYPE

構文

```
CREATE TYPE IF NOT EXISTS keyspace.type_name
( field, field, ...)
```

type_nameは、予約されている型の名前以外の型識別子です。

fieldは、以下の通りです。

```
field_name type
```

field_nameはそのフィールドに対する任意の識別子です。

typeはカウンター型以外の、CQLコレクション型または非コレクション型です。

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

ユーザー定義の型は、型を持つ1つ以上のフィールドです。ユーザー定義の型を使用すると、住所情報(番地、都市、郵便番号)など互いに関連する情報を表す複数のフィールドの処理が容易になります。すでに存在している型を作成しようとすると、IF NOT EXISTSオプションを使用していない限りエラーが返されます。このオプションを使用すると、すでにその型が存在していれば、この文は何もしません。

ユーザー定義の型を作成するには、CREATE TYPEコマンドを使用して、その後に型の名前とフィールドのリストを、コンマで区切って丸かっこで囲んで指定します。

ユーザー定義の型の名前は、以下のような、予約されている型の名前以外の名前を選択してください。

- byte
- smallint
- complex
- enum
- date
- interval
- macaddr
- bitstring

ドット表記を使用して、その型のキースペースを指定できます。キースペース名の後にピリオドを付け、その後に型の名前を記述します。Cassandraは、指定されたキースペース内にその型を作成しますが、現在のキースペースは変更しません。キースペースを別途指定しなかった場合、Cassandraは現在のキースペース内にその型を作成します。

例

この例では、住所や電話番号の情報で構成されるaddressというユーザー定義の型を作成します。

```
CREATE TYPE address (
street text,
city text,
zip_code int,
phones set<text>
)
```

address型を定義した後、その型のカラムを持つテーブルを作成できます。CQLのコレクション・カラムとその他のカラムでは、「ユーザー定義の型の使用」に示すように、ユーザー定義の型の使用がサポートされています。

CREATE USER

構文

```
CREATE USER IF NOT EXISTS user_name WITH PASSWORD 'password'
( NOSUPERUSER | SUPERUSER )
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

CREATE USERは、新しいデータベース・ユーザー・アカウントを定義します。デフォルトでは、ユーザー・アカウントにはsuperuserステータスはありません。CREATE USER要求を発行できるのはスーパーユーザーだけです。

ユーザー・アカウントは、内部認証および権限でログインする場合に必要です。

ユーザー名に英数字以外の文字が含まれている場合は、ユーザー名を単一引用符で囲んでください。既存のユーザーを再作成することはできません。スーパーユーザーのステータスまたはパスワードを変更するには、ALTER USERを使用します。

内部ユーザー・アカウントの作成

内部認証用のユーザー・アカウントを作成する場合は、WITH PASSWORD句を使用する必要があります。パスワードは単一引用符で囲んでください。

```
CREATE USER spillman WITH PASSWORD 'Niner27';
CREATE USER akers WITH PASSWORD 'Niner2' SUPERUSER;
CREATE USER boone WITH PASSWORD 'Niner75' NOSUPERUSER;
```

内部認証を設定していない場合は、以下のようにWITH PASSWORD句を使用する必要はありません。

```
CREATE USER test NOSUPERUSER;
```

ユーザー・アカウントの条件付き作成

Cassandra 2.0.9以降では、ユーザー・アカウントを作成する前に、ユーザーがアカウントを持っているかどうかをテストできます。IF NOT EXISTSオプションを使用しないで既存のユーザーを作成しようとすると、無効なクエリー条件になります。このオプションを使用すると、ユーザーが存在していれば、この文は何もしません。

```
$ bin/cqlsh -u cassandra -p cassandra
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.0 | CQL spec 3.2.0 | Native protocol v3]
Use HELP for help.

cqlsh> CREATE USER newuser WITH PASSWORD 'password';

cqlsh> CREATE USER newuser WITH PASSWORD 'password';
code=2200 [Invalid query] message="User newuser already exists"

cqlsh> CREATE USER IF NOT EXISTS newuser WITH PASSWORD 'password';
cqlsh>
```

DELETE

構文

```
DELETE column_name, ...| ( column_name term )
FROM keyspace_name.table_name
USING TIMESTAMP integer
WHERE row_specification
( IF ( EXISTS |( condition(  AND condition ) . . .  ) ) )
```

termは以下のとおりです。

```
[ list_position ] | key_value
```

row_specificationは以下のいずれかです。

```
primary_key_name = key_value
primary_key_name IN ( key_value, key_value, ...)(
```

conditionは以下のとおりです。

```
column_name = key_value
| column_name [list_position] = key_value
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

DELETE文は、1つ以上のカラムをテーブル内の1つ以上の行から削除するか、またはカラムが指定されていない場合は、行全体を削除します。Cassandraでは、同じパーティション・キー内の選択をアトミックかつ個別に適用します。

カラムまたは行の削除

DELETEキーワードの後に、カラム名をコンマで区切ってリストすることもできます。

```
DELETE col1, col2, col3 FROM Planeteers WHERE userID = 'Captain';
```

カラム名を指定しないと、WHERE句で指定した行全体が削除されます。

```
DELETE FROM MastersOfTheUniverse WHERE mastersID IN ('Man-At-Arms', 'Teela');
```

カラムを削除しても、ディスクからはすぐに削除されません。削除されたカラムはトゥームストーンでマークされ、構成した猶予期間が経過すると削除されます。任意指定のタイムスタンプは、新しいトゥームストーン・レコードを定義します。

カラムの条件付き削除

Cassandra 2.0.7以降では、IFまたはIF EXISTSを使用してカラムを条件付きで削除できます。カラムの削除は、条件付きで挿入または更新することに似ています。条件付き削除は、多大なパフォーマンス・コストが発生するので、慎重に使用する必要があります。

テーブルの指定

テーブル名は、カラム名のリストとキーワードFROMの後に指定します。

古いデータの削除

タイムスタンプを使用して、削除するカラムを識別できます。

```
DELETE email, phone
FROM users
USING TIMESTAMP 1318452291034
WHERE user_name = 'jsmith';
```

TIMESTAMP入力はマイクロ秒を表す整数です。WHERE句では、テーブルから削除する行を指定します。

```
DELETE col1 FROM SomeTable WHERE userID = 'some_key_value';
```

以下の形式では、IN表記と、かっこで囲まれたコンマ区切りのキー名のリストを使用して、キー名のリストを指定しています。

```
DELETE col1 FROM SomeTable WHERE userID IN (key1, key2);
DELETE phone FROM users WHERE user_name IN ('jdoe', 'jsmith');
```

Cassandra 2.0以降、CQLでは、IN句に値の空のリストを使用できるようになりました。Javaドライバー・アプリケーションで空の配列をIN句の引数として渡す場合に役立ちます。

コレクション・セット、リスト、またはマップの使用

マップから要素を削除するには、DELETEコマンドを使用し、要素のキーを大かっこで囲んで指定します。

```
DELETE todo ['2012-9-24'] FROM users WHERE user_id = 'frodo';
```

リストから要素を削除するには、DELETEコマンドを使用し、リストのインデックス位置を大かっこで囲んで指定します。

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';
```

セットからすべての要素を削除するには、DELETE文を使用します。

```
DELETE emails FROM users WHERE user_id = 'frodo';
```

DROP INDEX

構文

```
DROP INDEX IF EXISTS keyspace.index_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

DROP INDEX文は既存のインデックスを削除します。インデックスの作成時に名前が付けられていない場合は、インデックス名は<テーブル名>_<カラム名>_idxです。インデックスが存在しない場合、IF EXISTSが使用されていない場合は、この文はエラーを返しますが、使用されている場合は何もしません。キースペース名の後にピリオドとインデックス名を続けて指定するドット表記を使用して、削除するインデックスのキースペースを指定できます。Cassandraは指定されたキースペースのインデックスを削除しますが、現在のキースペースは変更しません。キースペースを指定しなかった場合は、Cassandra は現在のキースペース内のテーブルのインデックスを削除します。

例

```
DROP INDEX user_state;

DROP INDEX users_zip_idx;

DROP INDEX myschema.users_state;
```

DROP KEYSPACE

構文

```
DROP( KEYSPACE | SCHEMA) IF EXISTS keyspace_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

DROP KEYSPACE文は、キースペース内のすべてのテーブルおよびデータを含むキースペースを即時に削除します。この操作は元に戻せません。エイリアスであるDROP SCHEMAを使用することもできます。キースペースが存在しない場合、IF EXISTSが使用されていない限り、この文はエラーを返しますが、使用されていれば何もしません。

Cassandraはキースペースを削除する前にスナップショットを作成します。Cassandra 2.0.4以前では、ユーザーがスナップショットを手作業で削除する必要がありました。

例

```
DROP KEYSPACE MyTwitterClone;
```

DROP TABLE

構文

```
DROP TABLE IF EXISTS keyspace_name.table_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

DROP TABLE文は、テーブル内のすべてのデータを含むテーブルを即時に削除します。この操作は元に戻せません。エイリアスであるDROP COLUMNFAMILYを使用することもできます。

例

```
DROP TABLE worldSeriesAttendees;
```

DROP TRIGGER

構文

```
DROP TRIGGER trigger_name ON table_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

実験的なDROP TRIGGER文はCREATE TRIGGERを使用して作成されたトリガーの登録を削除します。

DROP TYPE

構文

```
DROP TYPE IF EXISTS type_name
```

type_nameはユーザー定義の型の名前です。

構文の凡例

- 大文字はリテラルを意味する

- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

この文は型を即時に削除し、元に戻せません。型を削除するには、以下の例で示すように、ALTER TYPEおよびDROPキーワードを使用します。存在しない型を削除しようとすると、IF EXISTSオプションを使用していない限りエラーが返されます。このオプションを使用すると、すでにその型が存在していれば、この文は何もしません。テーブルまたは別の型によって使用されているユーザー定義の型は削除できません。

```
DROP TYPE version;
```

DROP USER

構文

```
DROP USER IF EXISTS user_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

DROP USERは既存のユーザーを削除します。Cassandra 2.0.9以降では、ユーザーが存在することをテストできます。存在しないユーザーを削除しようとすると、IF EXISTSオプションを使用していない限り、無効なクエリー条件になります。このオプションを使用すると、ユーザーが存在しなければ、この文は何もしません。DROP USER文を発行するには、スーパーユーザーとしてログインする必要があります。ユーザーは自分自身を削除することはできません。

ユーザー名に英数字以外の文字が含まれている場合のみ、ユーザー名を単一引用符で囲みます。

GRANT

構文

```
GRANT permission_name PERMISSION
| ( GRANT ALL PERMISSIONS) ON resource TO user_name
```

permission_nameは以下のいずれかです。

- ALL
- ALTER

- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resourceは以下のいずれかです。

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name*.table_name

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

すべてのキースペース、指定のキースペース、またはテーブルにアクセスするパーミッションをユーザーに付与することができます。ユーザー名に英数字以外の文字が含まれている場合は、ユーザー名を単一引用符で囲んでください。

以下の表に、CQL文を使用するときに必要なパーミッションをリストします。

表 **11 : CQL**パーミッション

パーミッション	**CQL**文
ALL	すべての文
ALTER	ALTER KEYSPACE、ALTER TABLE、CREATE INDEX、DROP INDEX
AUTHORIZE	GRANT、REVOKE
CREATE	CREATE KEYSPACE、CREATE TABLE
DROP	DROP KEYSPACE、DROP TABLE
MODIFY	INSERT、DELETE、UPDATE、TRUNCATE
SELECT	SELECT

テーブルを対象にSELECTクエリーを実行できるようにするには、テーブル、親キースペース、または全キースペース(ALL KEYSPACE)を対象とするSELECTパーミッションを持つ必要があります。CREATE TABLEを可能にするには、親キースペースまたはALL KEYSPACESを対象とするCREATEパーミッションが必要です。ユーザーを対象にパーミッションをGRANT(付与)またはREVOKE(取り消し)するには、スーパーユーザーであるか、リソース(または階層内のいずれかの親)を対象としたAUTHORIZEパーミッションに加えて、対象となるパーミッションを持っている

必要があります。GRANT、REVOKE、およびLISTパーミッションは、実行の前に、テーブルとキースペースが存在することをチェックします。GRANTとREVOKEはユーザーが存在することをチェックします。

例

すべてのキースペースのすべてのテーブルを対象としたSELECTクエリーを実行するパーミッションをspillmanに付与します。

```
GRANT SELECT ON ALL KEYSPACES TO spillman;
```

fieldキースペースのすべてのテーブルを対象としたINSERT、UPDATE、DELETE、およびTRUNCATEクエリーを実行するパーミッションをakersに付与します。

```
GRANT MODIFY ON KEYSPACE field TO akers;
```

forty9ersキースペースを対象としたALTER KEYSPACEクエリーの実行と、forty9ersキースペースのすべてのテーブルを対象としたALTER TABLE、CREATE INDEX、およびDROP INDEXクエリーの実行のパーミッションをbooneに付与します。

```
GRANT ALTER ON KEYSPACE forty9ers TO boone;
```

ravens.playsテーブルですべての種類のクエリーを実行するパーミッションをbooneに付与します。

```
GRANT ALL PERMISSIONS ON ravens.plays TO boone;
```

ほかの誰もALL KEYSPACESアクセス権を持っていないものとして、ただ1人のユーザーにキースペースへのアクセス権を付与します。

```
GRANT ALL ON KEYSPACE keyspace_name TO user_name;
```

INSERT

構文

```
INSERT INTO keyspace_name.table_name
( column_name, column_name...)
VALUES ( value, value ...) IF NOT EXISTS
USING option AND option
```

column_nameはカラムまたはコレクション名です。

valueは以下のいずれかです。

- リテラル
- セット

  ```
  { literal, literal, . . . }
  ```
- リスト

  ```
  [ literal, literal, . . . ]
  ```
- マップ・コレクション、つまりJSON形式のリテラル配列

  ```
  { literal :literal, literal :literal, . . . }
  ```

optionは以下のいずれかです。

- TIMESTAMP microseconds
- TTL seconds

構文の凡例

- 大文字はリテラルを意味する

- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

INSERTは、アトミックかつ個別に、Cassandraテーブルのレコードに1つ以上のカラムを書き込みます。結果は返されません。キーを構成するカラムを除き、すべてのカラムを指定する必要はありません。指定しなかったカラムは、ディスク上のスペースを占有しません。

カラムが存在する場合は更新されます。行が存在していなければ行が作成されます。行が存在しない場合のみ挿入を行うには、IF NOT EXISTSを使用します。IF NOT EXISTSを使用すると、内部的なPaxosの使用に関連するパフォーマンスの低下を招きます。Paxosの詳細については、Cassandra 2.1 のドキュメントまたはCassandra 2.0のドキュメントを参照してください。

テーブル名は、キースペースで修飾できます。INSERTはカウンターをサポートしませんが、UPDATEはサポートします。内部的には、挿入操作と更新操作は同じです。

TIMESTAMPとTTLの指定

- Time-to-live(TTL)は秒単位です
- Timestampはミリ秒単位です

```
INSERT INTO Hollywood.NerdMovies (user_uuid, fan)
VALUES (cfd66ccc-d857-4e90-b1e5-df98a3d40cd6, 'johndoe')
USING TTL 86400;
```

TTLの指定は秒単位です。要求された時間が経過すると、TTLカラムの値に自動的に削除済みのマーク(トゥームストーン)が付きます。TTLは、カラム自体にではなく、挿入された値に期限を設定します。カラムのその後の更新で、TTLは更新で指定されたTTLにリセットされます。デフォルトでは、値は期限切れになりません。

TIMESTAMPの指定はミリ秒単位です。指定がなかった場合、カラムに対して発生した書き込みの時刻(ミリ秒単位)が使用されます。

コレクション・セットまたはマップの使用

コレクションにデータを挿入するには、値を中かっこで囲みます。セットの各値は一意である必要があります。例:

```
INSERT INTO users (userid, first_name, last_name, emails)
VALUES('frodo', 'Frodo', 'Baggins', {'f@baggins.com', 'baggins@gmail.com'});
```

todoという名前を付けたマップを挿入して、ユーザーのfrodoに10月2日に'die'というリマインダーを挿入します。

```
INSERT INTO users (userid, todo )
VALUES('frodo', {'2014-10-2 12:10' :'die' } );
```

コレクションの各項目の値は64Kに制限されます。

ユーザー定義の型のコレクション・カラムにデータを挿入するには、「ユーザー定義の型」で示すように、ユーザー定義の型の構成要素を丸かっこで囲み中かっこ内に置きます。

関連情報

Cassandra 2.1の調節可能な整合性

Cassandra 2.0の調節可能な整合性

プレイリストへのデータ挿入の例

このタスクについて

「音楽サービスの例」セクションでは、プレイリスト・テーブルについて説明しました。この例では、そのテーブルにデータを挿入する方法を示します。

手順

INSERTコマンドを使用して、playlistsテーブルに複合プライマリ・キーのUUID、タイトル、アーティスト、およびアルバム・データを挿入します。

```
INSERT INTO playlists (id, song_order, song_id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 1,
a3e64f8f-bd44-4f28-b8d9-6938726e34d4, 'La Grange', 'ZZ Top', 'Tres
 Hombres');

INSERT INTO playlists (id, song_order, song_id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 2,
8a172618-b121-4136-bb10-f665cfc469eb, 'Moving in Stereo', 'Fu Manchu', 'We
 Must Obey');

INSERT INTO playlists (id, song_order, song_id, title, artist, album)
VALUES (62c36092-82a1-3a00-93d1-46196ee77204, 3,
2b09185b-fb5a-4734-9b56-49077de9edbf, 'Outside Woman Blues', 'Back Door
 Slam', 'Roll Away');
```

LIST PERMISSIONS

構文

```
LIST permission_name PERMISSION
| ( LIST ALL PERMISSIONS )
ON resource OF user_name
NORECURSIVE
```

permission_nameは以下のいずれかです。

- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resourceは以下のいずれかです。

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name*.table_name

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する

- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

パーミッション・チェックは再帰的です。NORECURSIVE指定子を省略した場合は、要求対象のリソースおよび階層のその親に対するパーミッションが示されます。

- リソース名(ALL KEYSPACES、キースペース、またはテーブル)を省略すると、すべてのテーブルおよびすべてのキースペースのパーミッションがリストされます。
- ユーザー名を省略すると、すべてのユーザーのパーミッションがリストされます。すべてのユーザーのパーミッションをリストするには、スーパーユーザーである必要があります。スーパーユーザーでない場合は、以下を追加する必要があります。

```
OF <自分のユーザー名>
```

- NORECURSIVE指定子を省略すると、リソースおよびその親リソースに対するパーミッションがリストされます。
- ユーザー名に英数字以外の文字が含まれている場合のみ、ユーザー名を単一引用符で囲みます。

GRANTの例で、ユーザーを作成し、パーミッションを付与した後、リソースおよびその親に対してユーザーが持っているパーミッションをリストできます。

例

「例」の例を完了したと想定して、akersに付与されたすべてのパーミッションをリストします。

```
LIST ALL PERMISSIONS OF akers;
```

出力は以下のとおりです。

```
 username | resource            | permission
----------+---------------------+------------
akers | <keyspace field>    |     MODIFY
```

すべてのユーザーに付与されたパーミッションをリストします。

```
LIST ALL PERMISSIONS;
```

出力は以下のとおりです。

```
 username | resource              | permission
----------+-----------------------+------------
akers |      <keyspace field> |     MODIFY
boone | <keyspace forty9ers> |      ALTER
boone | <table ravens.plays> |     CREATE
boone | <table ravens.plays> |      ALTER
boone | <table ravens.plays> |       DROP
boone | <table ravens.plays> |     SELECT
boone | <table ravens.plays> |     MODIFY
boone | <table ravens.plays> |  AUTHORIZE
spillman |       <all keyspaces> |     SELECT
```

playsテーブルのすべてのパーミッションをリストします。

```
LIST ALL PERMISSIONS ON ravens.plays;
```

出力は以下のとおりです。

```
 username | resource              | permission
----------+-----------------------+------------
```

```
boone | <table ravens.plays> |     CREATE
boone | <table ravens.plays> |      ALTER
boone | <table ravens.plays> |       DROP
boone | <table ravens.plays> |     SELECT
boone | <table ravens.plays> |     MODIFY
boone | <table ravens.plays> |  AUTHORIZE
spillman |      <all keyspaces> |     SELECT
```

ravens.playsテーブルとその親に対するすべてのパーミッションをリストします。

出力は以下のとおりです。

```
LIST ALL PERMISSIONS ON ravens.plays NORECURSIVE;
```

```
 username | resource              | permission
----------+-----------------------+------------
boone | <table ravens.plays> |     CREATE
boone | <table ravens.plays> |      ALTER
boone | <table ravens.plays> |       DROP
boone | <table ravens.plays> |     SELECT
boone | <table ravens.plays> |     MODIFY
boone | <table ravens.plays> |  AUTHORIZE
```

LIST USERS

構文

```
LIST USERS
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

内部認証を使用し、CREATE USERの例でユーザーを作成し、デフォルトのユーザーをまだ変更していないとすると、以下の例はLIST USERSの出力を示します。

例

LIST USERS;

出力は以下のとおりです。

```
 name      | super
-----------+-------
cassandra |  True
boone | False
akers |  True
spillman | False
```

REVOKE

構文

```
REVOKE( permission_name PERMISSION )
| ( REVOKE ALL PERMISSIONS )
ON resource FROM user_name
```

permission_nameは以下のいずれかです。

- ALL
- ALTER
- AUTHORIZE
- CREATE
- DROP
- MODIFY
- SELECT

resourceは以下のいずれかです。

- ALL KEYSPACES
- KEYSPACE keyspace_name
- TABLE *keyspace_name*.table_name

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

すべてのキースペース、指定のキースペース、または1つのテーブルにアクセスするためのユーザーのパーミッションを取り消すことができます。ユーザー名に英数字以外の文字が含まれている場合は、ユーザー名を単一引用符で囲んでください。

GRANTの項に示した表に、CQL文を使用するのに必要なパーミッションがリストされています。

例

```
REVOKE SELECT ON ravens.plays FROM boone;
```

ユーザーbooneは、ravens.playsテーブルを対象にSELECTクエリーを実行できなくなります。例外：継承に従い、以下のいずれかの条件を満たす場合、ユーザーはravens.playsでSELECTクエリーを実行できます。

- ユーザーがスーパーユーザーである。
- ユーザーにALL KEYSPACESを対象するSELECTパーミッションがある。
- ユーザーにravensキースペースを対象とするSELECTパーミッションがある。

SELECT

構文

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

select_expression（select式）は以下のとおりです。

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

selection_listは以下のいずれかです。

- パーティション・キーのリスト（DISTINCTとともに使用します）
- `selector AS alias, selector AS alias, ...| *`

  aliasはカラム名の別名です。

selectorは以下のとおりです。

```
column name
| ( WRITETIME (column_name) )
| ( TTL (column_name) )
| (function (selector , selector, ...) )
```

functionは、timeuuid関数、token関数、またはBLOB変換関数です。

relationは以下のとおりです。

```
column_name op term
| ( column_name, column_name, ...) op term-tuple
| column_name IN( term, ( term ... ) )
| ( column_name, column_name, ...) IN( term-tuple, ( term-tuple ... ) )
| TOKEN (column_name, ...) op( term )
```

opは、=、<、>、<=、>、=、CONTAINS、CONTAINS KEYのいずれかです。

term-tuple（Cassandra 2.1以降）は以下のとおりです。

```
( term, term, ... )
```

termは以下のとおりです。

- 定数：文字列、数値、uuid、ブーリアン、16進数
- バインド・マーカー（?）
- 関数
- セット：

  ```
  { literal, literal, ... }
  ```
- リスト：

  ```
  [ literal, literal, ... ]
  ```
- マップ：

  ```
  { literal :literal, literal :literal, ... }
  ```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

SELECT文は、Cassandraテーブルから1つ以上のレコードを読み取ります。SELECT文への入力はselect式です。select文の出力はselect式によって異なります。

表 **12 : Select**式の出力

Select式	出力
カラムのリストのカラム	キー値と一連のカラムを持つ行
COUNT集計関数	結果セットの行数の値があるカラムを持つ1つの行
DISTINCTパーティション・キー・リスト	他のカラム値とは異なるカラムの値
WRITETIME関数	カラムへの書き込みが発生した日時
TTL関数	カラムの残りのTime To Live

カラムの指定

SELECT式では、どのカラムが結果に表示されるかを指定します。アスタリスクを使用すると、すべてのカラムが選択されます。

```
SELECT * from People;
```

ビッグ・データ・アプリケーション内のカラムの値は重複します。パーティション・キーの固有の(異なる)値だけを返すには、DISTINCTキーワードを使用してください。

返された行のカウント

COUNT(*)を使用するSELECT式は、クエリーに一致した行数を返します。COUNT(1)を使用しても同じ結果を得ることができます。

usersテーブルの行数をカウントします。

```
SELECT COUNT(*) FROM users;
```

カラム名の別名を使用する機能は、select式でカラムに対してdateOf(created_at)などの関数呼び出しを使用する場合に特に便利です。

```
SELECT event_id, dateOf(created_at), blobAsText(content) FROM timeline;
```

カラム別名の使用

Cassandra 2.0以降では、カラムに別名を定義できます。

```
SELECT event_id,
dateOf(created_at) AS creation_date,
blobAsText(content) AS content
FROM timeline;
```

出力では、カラムにわかりやすい名前が表示されます。

```
event_id                | creation_date             | content
------------------------+---------------------------+----------------
550e8400-e29b-41d4-a716 | 2013-07-26 10:44:33+0200 | Some stuff
```

LIMITを使用した返される行の数の指定

LIMITオプションを使用して、クエリーによって返される行の数を制限するように指定できます。

```
SELECT COUNT(*) FROM big_table LIMIT 50000;
SELECT COUNT(*) FROM big_table LIMIT 200000;
```

データベースに105,291行あった場合、上記の文の出力はそれぞれ50000行と105,291行になります。cqlshシェルには、デフォルトで10,000行という上限があります。Cassandraサーバーおよびネイティブ・プロトコルでは、誤った形式のクエリーを実行してシステムが不安定になるのを防ぐためにタイムアウトによってクエリーの実行が停止されますが、返される行の数は制限されていません。

FROMを使用したテーブルの指定

FROM句ではクエリー対象のテーブルを指定します。任意で、テーブルのキースペースを指定した後に、ピリオドとテーブル名を指定します。キースペースを指定しなかった場合は、現在のキースペースが使用されます。

たとえば、systemキースペースのIndexInfoテーブルの行数をカウントします。

```
SELECT COUNT(*) FROM system."IndexInfo";
```

WHEREを使用したデータのフィルター

WHERE句ではクエリー対象の行を指定します。WHERE句は、プライマリ・キーの要素であるか、またはインデックスが作成されているカラムに対する条件で構成されています。Cassandraでは、WHERE句でのCONTAINS、CONTAINS KEY、IN、=、>、>=、<、または<=という条件演算子をサポートしていますが、特定の条件下ではすべてがサポートされるとは限りません。WHERE句では、別名ではなく実際の名前を使用してカラムを参照してください。

WHERE句でプライマリ・キーを使用することにより、Cassandraは、該当するデータが含まれている特定のノードを知ることができます。等値条件演算子=とINの使用は制限されていません。演算子の左側の項はカラム名に、右側の項はフィルター対象のカラム値にする必要があります。他の条件演算子には制限があります。

- パーティション・キーでの等値条件以外の条件に基づくフィルターは、パーティショナーの順序が指定されている場合にのみサポートされます。
- WHERE句には>または<比較を含めることができますが、特定のパーティション・キーの場合、クラスタ化カラムに対する条件は、Cassandraで連続した順序の行を選択できるフィルターに制限されます。

例:

```
CREATE TABLE ruling_stewards (
steward_name text,
king text,
reign_start int,
event text,
PRIMARY KEY (steward_name, king, reign_start)
);
```

次のクエリーは、reign（治世）が2450までに始まり2500より前に終わったstewards（執政）に関するデータを選択するフィルターを構築します。king（王）がプライマリ・キーのコンポーネントではなかった場合、以下のクエリーを使用するためにkingにインデックスを作成する必要があります。

```
SELECT * FROM ruling_stewards
WHERE king = 'Brego'
AND reign_start >= 2450
AND reign_start < 2500 ALLOW FILTERING;
```

出力は以下のとおりです。

```
 steward_name | king  | reign_start | event
--------------+-------+-------------+--------------------
Boromir | Brego |        2477 |   Attacks continue
Cirion | Brego |        2489 | Defeat of Balchoth

(2 rows)
```

Cassandraで連続した順序の行を選択できるようにするには、等価条件を使用してフィルターにプライマリ・キーのkingコンポーネントを含める必要があります。ALLOW FILTERING句も必要です。

クラスター化カラムの比較

Cassandra 2.0.6以降では、パーティション・キーとクラスター化カラムをグループ化し、タプルと値を比較することで、パーティションの行スライスを取得できます。例：

```
SELECT * FROM ruling_stewards WHERE (steward_name, king) = ('Boromir',
 'Brego');
```

WHERE句で使用される構文は、タプルとしてのsteward_nameとkingのレコードを、Boromir、Bregoタプルと比較します。

INフィルター条件の使用

カラムの複数の候補値を指定するには、WHERE句で等価条件演算子INを使用します。たとえば、従業員ID（プライマリ・キー）が199、200、または207である3つの行から、NameとOccupationという2つのカラムを選択します。

```
SELECT Name, Occupation FROM People WHERE empID IN (199, 200, 207);
```

IN条件テストの値はコンマで区切られたリストとして設定してください。リストは特定の範囲のカラム値で構成できます。

CQLでは、IN句内に値の空のリストを使用できるため、Javaドライバー・アプリケーションで空の配列を引数としてIN句に渡す場合に役立ちます。

INを使用しない場合

インデックスを使用しない場合の推奨事項は、WHERE句でINを使用するときにも当てはまります。ほとんどの条件下では、WHERE句にINを使用しないことを推奨します。INを使用すると、通常は多くのノードを対象にクエリーを実行する必要が生じるため、パフォーマンスが低下する可能性があります。たとえば、ノードが30、レプリケーション係数が3、および整合性レベルがLOCAL_QUORUMの1つのローカル・データ・センター・クラスターでは、キーを1つ指定したクエリーは2つのノードに送られますが、クエリーでIN条件を使用すると、キーがトークンの範囲のどこに入るかによってクエリー対象のノード数が増え、最大で20になる可能性があります。

ALLOW FILTERING句

特定範囲の行を検索するなど潜在的に高負荷のクエリーを実行しようとすると、以下のプロンプトが表示されます。

```
Bad Request:Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance.If you want
```

```
to execute this query despite the performance unpredictability,
use ALLOW FILTERING.
```

このようなクエリーを実行するには、ALLOW FILTERING句を使用してください。メモリの使用量を減らすために、LIMIT n句を使用して制限することを推奨します。例:

```
Select * FROM ruling_stewards
WHERE king = 'none'
AND reign_start >= 1500
AND reign_start < 3000 LIMIT 10 ALLOW FILTERING;
```

ただし、LIMITを使用しても最悪の状況をまぬがれるわけではありません。たとえば、kingのないエントリーがなかった場合は、LIMITにかかわらずリスト全体をスキャンする必要があります。

データに関して収集する統計データを増やしていけば、おそらくALLOW FILTERINGはそれほど厳しいものではなくなります。たとえば、エントリーの90%にkingがないと知っていたら、そのようなエントリーを10個見つけるのは比較的負荷が低いことがわかります。

順序付けられていない結果全体のページング

TOKEN関数は、クエリー対象のパーティション・キー・カラムに対して条件演算子とともに使用できます。その場合、クエリーは、パーティション・キーの値ではなく、そのトークンに基づいて行を検索します。キーのトークンは、使用しているパーティショナーによって異なります。RandomPartitionerとともに使用すると有意義な順序が得られません。

たとえば、以下のテーブルを定義したとします。

```
CREATE TABLE periods (
period_name text,
event_name text,
event_date timestamp,
weak_race text,
strong_race text,
PRIMARY KEY (period_name, event_name, event_date)
);
```

データの挿入後に、以下のクエリーではパーティション・キーを使用してデータを検索するためにTOKEN関数を使用します。

```
SELECT * FROM periods
WHERE TOKEN(period_name) > TOKEN('Third Age')
AND TOKEN(period_name) < TOKEN('Fourth Age');
```

複合プライマリ・キーの使用と結果のソート

ORDER BY句では1つのカラムだけを選択できます。そのカラムは、複合プライマリ・キーの2番目のカラムである必要があります。これは、プライマリ・キーに複数のカラム・コンポーネントがあるテーブルにも当てはまります。順序指定は昇順または降順で行うことができ、デフォルトは昇順です。ASCまたはDESCキーワードを使用して指定します。

ORDER BY句では、別名ではなく実際の名前を使用してカラムを参照してください。

たとえば、複合プライマリ・キーを使用するplaylistsテーブルを設定し、サンプル・データを挿入します。以下のクエリーを使用してsong_orderで順序が指定された特定のプレイリストに関する情報を取得します。select式にORDER BYカラムを含める必要はありません。

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
ORDER BY song_order DESC LIMIT 50;
```

出力は以下のとおりです。

```
 id          | song_order | album                | artist         | song_id     | title
-------------+------------+----------------------+----------------+-------------+---------------------
 62c36092... |          4 | No One Rides for Free |      Fu Manchu | 7db1a490... |            Ojo Rojo
 62c36092... |          3 |            Roll Away | Back Door Slam | 2b09185b... | Outside Woman Blues
 62c36092... |          2 |         We Must Obey |      Fu Manchu | 8a172618... |    Moving in Stereo
 62c36092... |          1 |         Tres Hombres |         ZZ Top | a3e64f8f... |           La Grange
```

または、プレイリストのアーティストにインデックスを作成し、以下のクエリーを使用してそのプレイリストでFu Manchuの曲のタイトルを取得します。

```
CREATE INDEX ON playlists(artist)

SELECT album, title FROM playlists WHERE artist = 'Fu Manchu';
```

出力は以下のとおりです。

```
 album                | title
----------------------+-----------------
 We Must Obey         | Moving in Stereo
 No One Rides for Free | Ojo Rojo
```

コレクション・セット、リスト、またはマップのフィルター

コレクション全体を取得するためにコレクションが含まれているテーブルを対象にクエリーを実行できます。コレクション・カラムを対象にインデックスを作成してから、WHERE句にCONTAINS条件を指定してコレクションに属する特定の値のデータをフィルターできます。音楽サービスの例を続けると、playlistsテーブルにタグのコレクションを追加し、タグ・データを追加し、タグにインデックスを作成した後に、そのタグ・セットで'blues'をフィルターできます。

```
SELECT album, tags FROM playlists WHERE tags CONTAINS 'blues';
```

```
 album               | tags
---------------------+----------------------
 Tres Hombres        | {'1973', 'blues'}
```

音楽会場（venue）マップを対象にインデックスを作成したら、'The Fillmore'などのマップ値でフィルターできます。

```
SELECT * FROM playlists WHERE venue CONTAINS 'The Fillmore';
```

venueマップのコレクション・キーを対象にインデックスを作成したら、マップ・キーでフィルターできます。

```
SELECT * FROM playlists WHERE venue CONTAINS KEY '2013-09-22 22:00:00-0700';
```

書き込みが発生した日時の取得

WRITETIMEの後にカラム名を丸かっこで囲んで指定すると、カラムがデータベースに書き込まれた日時がマイクロ秒単位で返されます。

名前がJonesというユーザーのfirst_nameカラムに書き込みが発生した日時を取得します。

```
SELECT WRITETIME (first_name) FROM users WHERE last_name = 'Jones';

writetime(first_name)
-----------------------
 1353010594789000

(1 rows)
```

マイクロ秒単位のWRITETIME出力を変換すると、「November 15, 2012 at 12:16:34 GMT-8」になります。

TRUNCATE

構文

```
TRUNCATE keyspace_name.table_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

TRUNCATE文は、指定されたテーブル内のすべてのデータを即時に不可逆的に削除します。

例

```
TRUNCATE user_activity;
```

UPDATE

構文

```
UPDATE keyspace_name.table_name
USING option AND option
SET assignment, assignment, ...
WHERE row_specification
IF column_name = literal AND column_name = literal . . .
```

optionは以下のいずれかです。

- TIMESTAMP microseconds
- TTL seconds

assignmentは以下のいずれかになります。

```
column_name = value
set_or_list_item = set_or_list_item ( + | - ) ...
map_name = map_name ( + | - ) ...
column_name [ term ] = value
counter_column_name = counter_column_name( + | - ) integer
```

setは以下のとおりです。

```
{ literal, literal, . . . }
```

listは以下のとおりです。

```
[ literal, literal ]
```

mapは以下のとおりです。

```
{ literal :literal, literal :literal, . . . }
```

termは以下のとおりです。

```
[ list_index_position | [ key_value ]
```

row_specificationは以下のとおりです。

```
primary key name = key_value
primary key name IN (key_value ,...)
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する
- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

UPDATEは、任意の行の1つまたは複数のカラム値をCassandraテーブルに書き込みます。結果は返されません。文はUPDATEキーワードで始まり、その後にCassandraテーブル名が続きます。

行は、これまでに存在していなければ作成され、存在していれば更新されます。WHERE句内に、パーティション・キーを構成するすべてのカラムを含めることで、更新対象の行を指定します。IN関係はパーティション・キーの最後のカラムでのみサポートされます。UPDATE SET操作は、プライマリ・キー・フィールドでは有効ではありません。SETを使用して他のカラム値を指定します。複数のカラムを更新するには、名前と値のペアをコンマで区切ります。

以下のようにUPDATEを使用すると、軽量トランザクションを実行できます。

```
UPDATE customer_account

SET customer_email='laurass@gmail.com'

IF customer_email='lauras@gmail.com';
```

IFキーワードに続いて、更新が成功するために満たすべき条件を指定します。IF条件を使用すると、直列化による整合性をサポートするPaxosを内部で使うことに関連した、パフォーマンスへの影響が発生します。UPDATE文では、同じパーティション・キー内のすべての更新がアトミックに、独立して適用されます。

カウンター・テーブル内のカウンター・カラム値を更新するには、カウンター・カラムの現在値に対するインクリメントまたはデクリメントを指定します。INSERTコマンドとは異なり、UPDATEコマンドはカウンターをサポートします。それ以外は、更新と挿入の操作は内部で同一になります。

```
UPDATE UserActionCounts SET total = total + 2 WHERE keyalias = 523;
```

UPDATE文では、カウンター・カラムではないカラムを対象に以下のオプションを指定できます。

- TTL seconds
- TIMESTAMP microseconds

TTLの指定は秒単位です。要求された時間が経過すると、TTLカラムの値に自動的に削除済みのマーク(トゥームストーン)が付きます。TTLは、カラム自体にではなく、挿入された値に期限を設定します。カラムのその後の更新で、TTLは更新で指定されたTTLにリセットされます。デフォルトでは、値は期限切れになりません。

TIMESTAMP入力はマイクロ秒を表す整数です。指定がなかった場合、カラムに対して発生した書き込みの時刻(ミリ秒単位)が使用されます。各更新文には、WHERE句を使用して指定した正確なプライマリ・キーのセットが必要です。複合カラムおよびクラスタリング・カラムを持っているテーブル内のすべてのキーを指定する必要があります。たと

えば、useridとurlからなる複合プライマリ・キー持っているテーブル内のカラム値を更新するには、以下のようにします。

```
UPDATE excelsior.clicks USING TTL 432000
SET user_name = 'bob'
WHERE userid=cfd66ccc-d857-4e90-b1e5-df98a3d40cd6 AND
url='http://google.com';
UPDATE Movies SET col1 = val1, col2 = val2 WHERE movieID = key1;
UPDATE Movies SET col3 = val3 WHERE movieID IN (key1, key2, key3);
UPDATE Movies SET col4 = 22 WHERE movieID = key4;
```

CQLでは、IN句内に値の空のリストを使用できるため、Javaドライバー・アプリケーションで空の配列を引数としてIN句に渡す場合に役立ちます。

カラム更新の例

一度に複数行のカラムを更新するには、以下のようにします。

```
UPDATE users
SET state = 'TX'
WHERE user_uuid
IN (88b8fd18-b1ed-4e96-bf79-4280797cba80,
06a8913c-c0d6-477c-937d-6c1b69a95d43,
bc108776-7cb5-477f-917d-869c12dfffa8);
```

1行内の複数のカラムを更新するには、以下のようにします。

```
UPDATE users
SET name = 'John Smith',
email = 'jsmith@cassie.com'
WHERE user_uuid = 88b8fd18-b1ed-4e96-bf79-4280797cba80;
```

カウンター・カラムの更新

数値を加算または減算する数式を代入することで、カウンター・カラムの値を任意の数値で増減することができます。カウンター・カラム値を更新するには、以下の例に示す構文を使用します。

```
UPDATE counterks.page_view_counts
SET counter_value = counter_value + 2
WHERE url_name='www.datastax.com' AND page_name='home';
```

正確を期すため、軽量トランザクションをカウンター・カラムを対象に使うには、1つまたは複数のカウンター更新をバッチ文に含めます。

コレクション・セットの使用

1つの要素をセットに追加するには、UPDATEコマンドと加算(+)演算子を併用します。

```
UPDATE users
SET emails = emails + {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

セットから1つの要素を削除するには、減算(-)演算子を使用します。

```
UPDATE users
SET emails = emails - {'fb@friendsofmordor.org'} WHERE user_id = 'frodo';
```

セットからすべての要素を削除するには、次のようなUPDATE文を使用します。

```
UPDATE users SET emails = {} WHERE user_id = 'frodo';
```

コレクション・マップの使用

マップ・データを設定または置き換えるには、UPDATEコマンドを使用します。たとえば、マップ・コレクション構文内のタイムスタンプおよびテキスト値を、コロンで区切って中かっこで囲みます。

```
UPDATE users
SET todo =
{ '2012-9-24' :'enter mordor',
'2012-10-2 12:00' :'throw ring into mount doom' }
WHERE user_id = 'frodo';
```

また、UPDATEコマンドを使用して、特定の要素を更新または設定できます。たとえば、以下のようにして、10月2日にユーザーfrodoに対して'die'というリマインダーを挿入するよう、todoというマップを更新します。

```
UPDATE users SET todo['2014-10-2 12:10'] = 'die'
WHERE user_id = 'frodo';
```

各マップ要素にTTLを設定できます。

```
UPDATE users USING TTL <TTL値>
SET todo['2012-10-1'] = 'find water' WHERE user_id = 'frodo';
```

コレクション・リストの使用

リストに値を挿入するには、以下のようにします。

```
UPDATE users
SET top_places = [ 'rivendell', 'rohan' ] WHERE user_id = 'frodo';
```

リストの先頭に1つの要素を追加するには、要素を大かっこで囲み、加算(+)演算子を使用します。

```
UPDATE users
SET top_places = [ 'the shire' ] + top_places WHERE user_id = 'frodo';
```

リストに1つの要素を追加するには、UPDATEコマンド内の新しい要素データとリスト名の順序を入れ替えます。

```
UPDATE users
SET top_places = top_places + [ 'mordor' ] WHERE user_id = 'frodo';
```

特定の位置に1つの要素を追加するには、大かっこで囲んだリスト・インデックス位置を使用します。

```
UPDATE users SET top_places[2] = 'riddermark' WHERE user_id = 'frodo';
```

特定の値を持つすべての要素を削除するには、UPDATEコマンド、減算演算子(-)、大かっこで囲んだリスト値を使用します。

```
UPDATE users
SET top_places = top_places - ['riddermark'] WHERE user_id = 'frodo';
```

ユーザー定義型のコレクション・カラムのデータを更新するには、「ユーザー定義の型」で示すように、そのユーザー定義型の構成要素を丸かっこで囲み中かっこ内に置きます。

USE

構文

```
USE keyspace_name
```

構文の凡例

- 大文字はリテラルを意味する
- 小文字は、リテラルでないことを意味する
- イタリック体は指定が任意であることを意味する

- パイプ(|)記号はORまたはAND/ORを意味する
- 省略記号(...)は繰り返し可能を意味する
- 範囲記号「(」および「)」はリテラルではなく、範囲を示す

この構文には、CQL文の終了となるセミコロンは含まれていません。

説明

USE文により、現在のクライアント・セッションのクエリーを行う対象となるテーブルを含んでいるキースペースを指定します。テーブルやインデックスを対象とした以降の操作はすべて、特に指定されない限り、クライアント接続が終了するか、他のUSE文が発行されるまで、指定されたキースペースのコンテキスト内で行われます。

大文字と小文字が区別されるキースペースを使用するには、キースペース名を二重引用符で囲みます。

例

```
USE PortfolioDemo;
```

作成したキースペースのチェックの例が続きます。

```
USE "Excalibur";
```