**РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

**RT.KEYVALUE**

2021

ОГЛАВЛЕНИЕ

[1 ОПИСАНИЕ RT.KEYVALUE 5](#_Toc86147206)

[2 Язык запросов (CQL) 6](#_Toc86147207)

[2.1 Определения 6](#_Toc86147208)

[2.1.1 Условные обозначения 6](#_Toc86147209)

[2.1.2 Идентификаторы и ключевые слова 6](#_Toc86147210)

[2.1.3 Константы 8](#_Toc86147211)

[2.1.4 Термины (Terms) 9](#_Toc86147212)

[2.1.5 Комментарии 9](#_Toc86147213)

[2.1.6 Операторы 9](#_Toc86147214)

[2.1.7 Подготовленные операторы 11](#_Toc86147215)

[2.2 Типы данных 11](#_Toc86147216)

[2.2.1 Собственные типы (Native Types) 12](#_Toc86147217)

[2.2.2 Работа с метками времени (timestamps) 14](#_Toc86147218)

[2.2.3 Тип даты (Date type) 14](#_Toc86147219)

[2.2.4 Тип времени (Time type) 15](#_Toc86147220)

[2.2.5 Тип продолжительности (Duration Type) 15](#_Toc86147221)

[2.2.6 Коллекции (Collections) 16](#_Toc86147222)

[2.2.7 Типы, определяемые пользователем (User-defined Types (UDTs)) 20](#_Toc86147223)

[2.2.8 Кортежи (Tuples) 23](#_Toc86147224)

[2.2.9 Настраиваемые типы (Custom Types) 23](#_Toc86147225)

[2.3 Определение данных (Data Definition(DDL)) 24](#_Toc86147226)

[2.3.1 Общие определения 24](#_Toc86147227)

[2.3.2 Создание пространства ключей (CREATE KEYSPACE) 25](#_Toc86147228)

[2.3.3 USE 27](#_Toc86147229)

[2.3.4 ALTER KEYSPACE 27](#_Toc86147230)

[2.3.5 DROP KEYSPACE 28](#_Toc86147231)

[2.3.6 CREATE TABLE 28](#_Toc86147232)

[2.3.7 ALTER TABLE 42](#_Toc86147233)

[2.3.8 DROP TABLE 43](#_Toc86147234)

[2.3.9 TRUNCATE 44](#_Toc86147235)

[2.4 Манипулирование данными (Data Manipulation (DML)) 44](#_Toc86147236)

[2.4.1 SELECT 44](#_Toc86147237)

[2.4.2 INSERT 50](#_Toc86147238)

[2.4.3 UPDATE 51](#_Toc86147239)

[2.4.4 DELETE 53](#_Toc86147240)

[2.4.5 BATCH 54](#_Toc86147241)

[2.5 Арифметические операторы (Arithmetic Operators) 55](#_Toc86147242)

[2.5.1 Числовая арифметика (Number Arithmetic) 56](#_Toc86147243)

[2.5.2 Дата-время арифметика (Datetime Arithmetic) 56](#_Toc86147244)

[2.6 Вторичные индексы (Secondary Indexes) 56](#_Toc86147245)

[2.6.1 Создать индекс (CREATE INDEX) 56](#_Toc86147246)

[2.6.2 Удаление индекса (DROP INDEX) 57](#_Toc86147247)

[2.7 Материализованные представления (Materialized Views) 57](#_Toc86147248)

[2.7.1 Создать материализированное представление (CREATE MATERIALIZED VIEW) 58](#_Toc86147249)

[2.7.2 ALTER MATERIALIZED VIEW 60](#_Toc86147250)

[2.7.3 DROP MATERIALIZED VIEW 61](#_Toc86147251)

[2.8 Функции (Functions) 61](#_Toc86147252)

[2.8.1 Scalar - функции (functions) 62](#_Toc86147253)

[2.8.2 Aggregate - функции (functions) 70](#_Toc86147254)

[2.9 Поддержка JSON 74](#_Toc86147255)

[2.9.1 SELECT JSON 74](#_Toc86147256)

[2.9.2 INSERT JSON 75](#_Toc86147257)

[2.9.3 Кодирование JSON типов данных RT.KeyValue (JSON Encoding of RT.KeyValue Data Types) 75](#_Toc86147258)

[2.9.4 Функция fromJson() 77](#_Toc86147259)

[2.9.5 Функция toJson() 77](#_Toc86147260)

[2.10 Безопасность (Security) 78](#_Toc86147261)

[2.10.1 Роли базы данных (Database Roles) 78](#_Toc86147262)

[2.10.2 Пользователи (Users) 82](#_Toc86147263)

[2.10.3 Контроль данных (Data Control) 83](#_Toc86147264)

[2.11 Триггеры (Triggers) 90](#_Toc86147265)

[2.11.1 CREATE TRIGGER 90](#_Toc86147266)

[2.11.2 CREATE TRIGGER 90](#_Toc86147267)

[Приложение 1. КЛЮЧЕВЫЕ СЛОВА CQL 91](#_Toc86147268)

ОПИСАНИЕ RT.KEYVALUE

**RT.KeyValue** – это NoSQL распределенная база, обладающая высокой масштабируемостью и доступностью без ущерба для производительности. Линейная масштабируемость и проверенная отказоустойчивость на стандартном оборудовании или облачной инфраструктуре делают её идеальной платформой для критически важных данных.

Данный документ описывет возможности взаимодейтсвия Пользователя с RT.KeyValue при использованнии языка запросов CQL.

Язык запросов (CQL)

CQL представляет модель, аналогичную SQL. Данные хранятся в таблицах, содержащих строки столбцов. По этой причине использованные термины (таблицы, строки и столбцы) имеют то же определение, что и в SQL.

Определения

Условные обозначения

Чтобы помочь в указании синтаксиса CQL, в этоом документе будут использоваться следующие соглашения:

1. Языковые правила будут даны в неформальной вариантной нотации BNF. В частности, будут использованы квадратные скобки ([ item ]) для необязательных элементов, \* и + для повторяющихся элементов (где + означает хотя бы один).
2. В грамматике также будет использоваться следующее соглашение для удобства: нетерминальный термин будет написан в нижнем регистре (и будет ссылаться на его определение), в то время как терминальные ключевые слова будут предоставлены "все заглавными буквами". Однако обратите внимание, что ключевые слова являются identifiers (идентификаторами) и, таким образом, на практике нечувствительны к регистру. Также будет определа некоторая ранняя конструкция с помощью регулярного выражения, которое мы обозначим с помощью re(<some regular expression>) (<некоторое регулярное выражение>).
3. Грамматика предназначена для документирования и не учитывает некоторые незначительные детали. Например, запятая в определении последнего столбца в операторе CREATE TABLE необязательна, но поддерживается, если присутствует, даже если грамматика в этом документе предполагает иное. Кроме того, не все, что принято в грамматике, обязательно является правильным CQL.
4. Ссылки на ключевые слова или фрагменты кода CQL в текущем тексте будут отображаться fixed-width font (шрифтом фиксированной ширины).

Идентификаторы и ключевые слова

Язык CQL использует *идентификаторы* (или *имена*) для идентификации таблиц, столбцов и других объектов. Идентификатор - это токен, соответствующий регулярному выражению [a-zA-Z][a-zA-Z0-9\_]\*.

Ряд таких идентификаторов, как SELECT или WITH, являются *ключевыми словами*. Они имеют фиксированное значение для языка, и большинство из них зарезервированы. Список ключевых слов можно найти в **Приложении 1**.

Идентификаторы и ключевые слова (без кавычек) нечувствительны к регистру. Таким образом, SELECT совпадает с select или sElEcT, а myId совпадает с myid или MYID. Часто используемое соглашение (в частности, в примерах из этой документации) заключается в использовании верхнего регистра для ключевых слов и нижнего регистра для других идентификаторов.

Существует второй вид идентификаторов, называемых *идентификаторами в кавычках*, которые определяются заключением произвольной последовательности символов (непустых) в двойные кавычки ("). Цитированные идентификаторы никогда не являются ключевыми словами. Таким образом, "select" не является зарезервированным ключевым словом и может быть используется для ссылки на столбец (обратите внимание, что использовать это особенно не рекомендуется), тогда как select вызовет ошибку синтаксического анализа. Кроме того, в отличие от идентификаторов и ключевых слов без кавычек, идентификаторы в кавычках чувствительны к регистру ("My Quoted Id" отличается от "my quoted id"). Идентификатор в полностью строчных кавычках, который соответствует [a-zA-Z][a-zA-Z0-9\_]\*, однако, эквивалентен идентификатору без кавычек, полученному путем удаления двойных кавычек (так что "myid" эквивалентен в myid и myId, но отличается от myId). Внутри идентификатора, заключенного в кавычки, можно повторить символ двойной кавычки, чтобы избежать его, поэтому "foo "" bar" является допустимым идентификатором.

|  |
| --- |
| **Примечание.** |
| *Идентификатор в кавычках* может объявлять столбцы с произвольными именами, которые иногда могут конфликтовать с конкретными именами, используемыми сервером. Например, при использовании условного обновления сервер ответит набором результатов, содержащим специальный результат с именем "[applied]" "[применено]". Если вы объявили столбец с таким именем, это может сбить с толку некоторые инструменты, и этого следует избегать. Как правило, предпочтительнее использовать идентификаторы без кавычек, но если вы используете идентификаторы в кавычках, настоятельно рекомендуется избегать любых имен, заключенных в квадратные скобки (например, "[applied]"), и любых имен, которые выглядят как вызов функции (например, "f(x)"). |

Более формально:

|  |
| --- |
| identifier::= unquoted\_identifier | quoted\_identifier  unquoted\_identifier::= re('[a-zA-Z][link:[a-zA-Z0-9]]\*')  quoted\_identifier::= '"' (any character where " can appear if doubled)+ '"' |

Константы

CQL определяет следующие *константы*:

|  |
| --- |
| constant::= string | integer | float | boolean | uuid | blob | NULL  string::= ''' (any character where ' can appear if doubled)+ ''' : '$$' (any character other than '$$') '$$'  integer::= re('-?[0-9]+')  float::= re('-?[0-9]+(.[0-9]\*)?([eE][+-]?[0-9+])?') | NAN | INFINITY  boolean::= TRUE | FALSE  uuid::= hex\{8}-hex\{4}-hex\{4}-hex\{4}-hex\{12}  hex::= re("[0-9a-fA-F]")  blob::= '0' ('x' | 'X') hex+ |

Другими словами:

* Строковая константа - это произвольная последовательность символов, заключенная в одинарные кавычки ('). Можно включить одинарную кавычку, повторив ее, например 'It''s raining today'. Их не следует путать с цитируемыми идентификаторами, в которых используются двойные кавычки. В качестве альтернативы строку можно определить, заключив произвольную последовательность символов двумя символами доллара, и в этом случае можно использовать одинарные кавычки без экранирования (It's raining today). Эта последняя форма часто используется при определении **определяемых пользователем функций**, чтобы избежать необходимости экранировать символы одиночных кавычек в теле функции (поскольку они более вероятны, чем $$).
* Целочисленные (integer), плавающие (float) и логические (Boolean) константы определены должным образом. Однако обратите внимание, что float позволяет использовать специальные константы NaN и Infinity.
* CQL поддерживает UUID константы.
* Содержимое BLOB-объектов предоставляется в шестнадцатеричном формате с префиксом 0x.
* Специальная константа NULL означает отсутствие значения.

О том, как типизируются эти константы, см. В разделе **"Типы данных".**

Термины (Terms)

В CQL есть понятие терминов, обозначающих типы значений, поддерживаемых CQL. Термины определяются:

|  |
| --- |
| term::= constant | literal | function\_call | arithmetic\_operation | type\_hint | bind\_marker  literal::= collection\_literal | udt\_literal | tuple\_literal  function\_call::= identifier '(' [ term (',' term)\* ] ')'  arithmetic\_operation::= '-' term | term ('+' | '-' | '\*' | '/' | '%') term  type\_hint::= '(' cql\_type ')' term  bind\_marker::= '?' | ':' identifier |

Таким образом, термин является одним из:

* Константы.
* Литерал для коллекции, определяемого пользователем типа или кортежа.
* Вызов функции, либо встроенной функции, либо пользовательской функции.
* Арифметическая операция между терминами.
* Тип подсказки (hint).
* Маркер привязки, который обозначает переменную, которая должна быть привязана во время выполнения. См. Подробности в разделе о prepared-statements операторах. Маркер привязки может быть анонимным (?) Или именованным (:some\_name). Последняя форма обеспечивает более удобный способ ссылки на переменную для ее привязки и, как правило, предпочтительнее.

Комментарии

Комментарий в CQL - это строка, начинающаяся либо с двойного тире (--), либо с двойной косой черты (//).

Многострочные комментарии также поддерживаются заключением в / **и** / (но вложение не поддерживается).

|  |
| --- |
| -- This is a comment  // This is a comment too  /\* This is  a multi-line comment \*/ |

Операторы

CQL состоит из операторов, которые можно разделить на следующие категории:

* data-definition операторы определения данных, чтобы определять и изменять способ хранения данных (пространства ключей и таблицы).
* data-manipulation операторы управления данными для выбора вставки и удаления данных.
* secondary-indexes операторы вторичных индексов.
* materialized-views операторы утверждения материализованных просмотров.
* cql-roles операторы.
* cql-permissions операторы.
* User-Defined Functions (UDFs) операторы пользовательских функций (UDF).
* udts операторы.
* cql-triggers операторы.

Все операторы перечислены ниже и описаны в остальной части этой документации (см. Ссылки выше):

|  |
| --- |
| cql\_statement::= statement [ ';' ]  statement:=: ddl\_statement :  | dml\_statement  | secondary\_index\_statement  | materialized\_view\_statement  | role\_or\_permission\_statement  | udf\_statement  | udt\_statement  | trigger\_statement  ddl\_statement::= use\_statement  | create\_keyspace\_statement  | alter\_keyspace\_statement  | drop\_keyspace\_statement  | create\_table\_statement  | alter\_table\_statement  | drop\_table\_statement  | truncate\_statement  dml\_statement::= select\_statement  | insert\_statement  | update\_statement  | delete\_statement  | batch\_statement  secondary\_index\_statement::= create\_index\_statement  | drop\_index\_statement  materialized\_view\_statement::= create\_materialized\_view\_statement  | drop\_materialized\_view\_statement  role\_or\_permission\_statement::= create\_role\_statement  | alter\_role\_statement  | drop\_role\_statement  | grant\_role\_statement  | revoke\_role\_statement  | list\_roles\_statement  | grant\_permission\_statement  | revoke\_permission\_statement  | list\_permissions\_statement  | create\_user\_statement  | alter\_user\_statement  | drop\_user\_statement  | list\_users\_statement  udf\_statement::= create\_function\_statement  | drop\_function\_statement  | create\_aggregate\_statement  | drop\_aggregate\_statement  udt\_statement::= create\_type\_statement  | alter\_type\_statement  | drop\_type\_statement  trigger\_statement::= create\_trigger\_statement  | drop\_trigger\_statement |

Подготовленные операторы

CQL поддерживает подготовленные операторы (*prepared statements)*. Подготовленные операторы - это оптимизация, которая позволяет анализировать запрос только один раз, но выполнять его несколько раз с разными конкретными значениями.

Любой оператор, который использует хотя бы один маркер привязки (см. bind\_marker), должен быть подготовлен. После чего оператор может быть выполнен с заданными конкретными значениями для каждого его маркера. Точные сведения о том, как подготавливается и затем выполняется инструкция, зависят от используемого драйвера CQL, и вам следует обратиться к документации по вашему драйверу.

Типы данных

CQL - это типизированный язык, поддерживающий богатый набор типов данных, включая собственные типы (native types), типы коллекций (collection types), определяемые пользователем типы (user-defined types), типы кортежей (tuple types) и настраиваемые типы (custom types):

|  |
| --- |
| cql\_type::= native\_type| collection\_type| user\_defined\_type | tuple\_type | custom\_type |

Собственные типы (Native Types)

CQL поддерживает следующие собственные типы:

|  |
| --- |
| native\_type::= ASCII | BIGINT | BLOB | BOOLEAN | COUNTER | DATE  | DECIMAL | DOUBLE | DURATION | FLOAT | INET | INT |  SMALLINT | TEXT | TIME | TIMESTAMP | TIMEUUID | TINYINT |  UUID | VARCHAR | VARINT |

В **Таблице 1** представлена дополнительная информация о собственных типах данных и о том, какие типы констант поддерживает каждый тип:

**Таблица 1. Собственные типы и типы поддерживаемых констант.**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Поддерживаемые константы** | **Описание** |
| ascii | string | Строка символов ASCII |
| bigint | integer | 64-битная длинная подпись (signed long) |
| blob | blob | Произвольные байты (без проверки) |
| boolean | boolean | Либо true, либо false |
| counter | integer | Столбец счетчика (64-битное значение со знаком). См. Подробности на счетчиках (counters) |
| date | integer, string | Дата (без соответствующего значения времени). Подробности смотрите в датах (dates) ниже |
| decimal | integer, float | Десятичное число переменной точности |
| double | integer float | 64-битный IEEE-754 с плавающей запятой |
| duration | duration | Длительность с точностью до наносекунды. См. Подробности ниже (durations) |
| float | integer, float | 32-битный IEEE-754 с плавающей запятой |
| inet | string | IP-адрес: IPv4 (4 байта) или IPv6 (16 байтов). Обратите внимание, что нет константы inet, IP-адрес следует вводить в виде строк. |
| int | integer | 32-битное целое число со знаком |
| smallint | integer | 16-битное целое число со знаком |
| text | string | Строка в кодировке UTF8 |
| time | integer, string | Время (без соответствующего значения даты) с точностью до наносекунды. См. Подробности ниже (times). |
| timestamp | integer, string | Отметка времени (дата и время) с точностью до миллисекунды. См. Подробности ниже (timestamps) |
| timeuuid | uuid | UUID версии 1, обычно используемый как временная метка "без конфликтов". Также см. timeuuid-functions. |
| tinyint | integer | 8-битное целое число со знаком |
| uuid | uuid | UUID (любой версии) |
| varchar | string | Строка в кодировке UTF8 |
| varint | integer | Целое число произвольной точности |

Счетчики (Counters)

Тип счетчика (counter) используется для определения столбцов счетчика. Столбец счетчика - это столбец, значение которого представляет собой 64-битное целое число со знаком и для которого поддерживаются 2 операции: увеличение (incrementing) и уменьшение (decrementing). Обратите внимание, что значение счетчика не может быть установлено: счетчик не существует до первого увеличения / уменьшения, и это первое увеличение / уменьшение выполняется, как если бы предыдущее значение было 0.

Счетчики имеют ряд важных ограничений:

* Их нельзя использовать для столбцов, являющихся частью Первичного ключа (PRIMARY KEY) таблицы.
* Таблица, содержащая счетчик, может содержать только счетчики. Другими словами, либо все столбцы таблицы за пределами Первичного ключа (PRIMARY KEY) имеют тип счетчика (counter), либо ни один из них не имеет его.
* Счетчики не поддерживают истечение срока.
* Удаление счетчиков поддерживается, но сработает только при первом удалении счетчика. Другими словами, не следует повторно обновлять счетчик, который удалили.
* Обновления счетчика по своей природе не идемпотентны. Важным следствием является то, что в случае неожиданного сбоя обновления счетчика (тайм-аут или потеря соединения с узлом-координатором) клиент не имеет возможности узнать, было ли применено обновление или нет. В частности, повторное воспроизведение обновления может привести или не привести к перерасчету.

Работа с метками времени (timestamps)

Значения типа timestamp кодируются как 64-битовые целые числа со знаком, представляющие количество миллисекунд, прошедших с момента стандартного базового времени, известного как эпоха: 1 января 1970 года в 00:00:00 по Гринвичу.

Метки времени могут быть введены в CQL либо с использованием их значения в виде целого числа (integer), либо с использованием строки (string), представляющей дату ISO 8601. Например, все приведенные ниже значения являются допустимыми значениями метки времени для 2 марта 2011 г., 04:05:00 по Гринвичу:

* 1299038700000
* '2011-02-03 04:05+0000'
* '2011-02-03 04:05:00+0000'
* '2011-02-03 04:05:00.000+0000'
* '2011-02-03T04:05+0000'
* '2011-02-03T04:05:00+0000'
* '2011-02-03T04:05:00.000+0000'

Приведенный выше +0000 - это 4-значная спецификация часового пояса RFC 822; +0000 относится к GMT. Стандартное тихоокеанское время США: -0800. При желании часовой пояс может быть опущен ('2011-02-03 04:05:00'), и, если это так, дата будет интерпретироваться как находящаяся в часовом поясе, в котором настроен координирующий узел RT.KeyValue. Однако существуют трудности, связанные с тем, чтобы надо полагаться на ожидаемую конфигурацию часового пояса, поэтому рекомендуется всегда указывать часовой пояс для временных меток, когда это возможно.

Время дня также можно не указывать ('2011-02-03' или '2011-02-03+0000'), и в этом случае время дня по умолчанию будет 00:00:00 в указанном часовом поясе или часовом поясе по умолчанию. Однако, если важна только часть даты, рассмотрите возможность использования типа даты.

Тип даты (Date type)

Значения типа даты (date) кодируются как 32-битовые целые числа без знака, представляющие количество дней с "эпохой" в центре диапазона (2 ^ 31). Эпоха 1 января 1970 года.

Для отметок времени (timestamps) дату можно вводить либо как целое число (integer), либо в виде строки даты (string). В последнем случае формат должен быть yyyy-mm-dd (например, '2011-02-03').

Тип времени (Time type)

Значения типа time кодируются как 64-битовые целые числа со знаком, представляющие количество наносекунд, прошедших с полуночи.

Для отметок времени (timestamps) время может быть введено либо как целое число (integer), либо с использованием строки (string), представляющей время. В последнем случае формат должен быть чч: мм: сс hh:mm:ss[.fffffffff] (где субсекундная точность является необязательной и, если указана, может быть меньше наносекунды). Так, например, следующие допустимые входные данные на время:

* '08:12:54'
* '08:12:54.123'
* '08:12:54.123456'
* '08:12:54.123456789'

Тип продолжительности (Duration Type)

Значения типа продолжительности (duration) кодируются как 3 целых числа со знаком переменной длины. Первое целое число представляет количество месяцев, второе - количество дней, а третье - количество наносекунд. Это связано с тем, что количество дней в месяце может меняться, а в сутках может быть 23 или 25 часов в зависимости от летнего времени. Внутренне количество месяцев и дней декодируется как 32-битное целое число, тогда как количество наносекунд декодируется как 64-битное целое число.

Продолжительность можно ввести как:

1. единица количества ((quantity unit)+) + вроде 12h30m, где единица измерения может быть:

* y: лет (12 месяцев)
* mo: месяцы (1 месяц)
* w: недели (7 дней)
* d: дни (1 день)
* h: часы (3 600 000 000 000 наносекунд)
* m: минуты (60 000 000 000 наносекунд)
* s: секунды (1000000000 наносекунд)
* ms: миллисекунды (1000000 наносекунд)
* us или µs: микросекунды (1000 наносекунд)
* ns: наносекунды (1 наносекунда)

1. Формат ISO 8601: P[n]Y[n]M[n]DT[n]H[n]M[n]S or P[n]W
2. Альтернативный формат ISO 8601: P [ГГГГ] - [ММ] - [ДД] T [чч]: [мм]: [сс] (P[YYYY]-[MM]-[DD]T[hh]:[mm]:[ss])

Например:

|  |
| --- |
| INSERT INTO RiderResults (rider, race, result)  VALUES ('Christopher Froome', 'Tour de France', 89h4m48s);  INSERT INTO RiderResults (rider, race, result)  VALUES ('BARDET Romain', 'Tour de France', PT89H8M53S);  INSERT INTO RiderResults (rider, race, result)  VALUES ('QUINTANA Nairo', 'Tour de France', P0000-00-00T89:09:09); |

Столбцы продолжительности не могут использоваться в Первичном ключе (PRIMARY KEY) таблицы. Это ограничение связано с тем, что продолжительность не может быть заказана. Фактически невозможно узнать, 1mo превышает ли 29d без контекста даты.

Продолжительность 1d не равна 24h, поскольку тип продолжительности был создан для поддержки летнего времени.

Коллекции (Collections)

CQL поддерживает три типа коллекций: карты (maps), наборы (sets) и списки (lists). Типы этих коллекций определяются:

|  |
| --- |
| collection\_type::= MAP '<' cql\_type',' cql\_type'>'  | SET '<' cql\_type '>'  | LIST '<' cql\_type'>' |

и их значения могут быть введены с помощью литералов коллекции:

|  |
| --- |
| collection\_literal::= map\_literal | set\_literal | list\_literal  map\_literal::= '\{' [ term ':' term (',' term : term)\* ] '}'  set\_literal::= '\{' [ term (',' term)\* ] '}'  list\_literal::= '[' [ term (',' term)\* ] ']' |

Однако обратите внимание, что ни bind\_marker, ни NULL не поддерживаются внутри литералов коллекции.

Примечательные характеристики

Коллекции предназначены для хранения / денормализации относительно небольшого количества данных. Они хорошо подходят для таких вещей, как "номера телефонов данного пользователя", "метки, примененные к электронной почте" и т.д. Но когда, ожидается, что количество элементов будет неограниченным ("все сообщения, отправленные пользователем", "события, зарегистрированные датчиком" и т.д.), коллекции не подходят, и следует использовать конкретную таблицу (со столбцами кластеризации). Конкретно (незамороженные) коллекции имеют следующие примечательные характеристики и ограничения:

* Отдельные коллекции не индексируются внутри. Это означает, что даже для доступа к одному элементу коллекции необходимо прочитать коллекцию *while* (а чтение одной страницы не выгружается внутри).
* В то время как операции вставки в наборы и карты никогда не требуют внутреннего чтения перед записью, некоторые операции со списками делают это. Кроме того, некоторые операции со списками не являются идемпотентными по своей природе (подробности см. В разделе о списках ниже (lists)), что делает их повторные попытки в случае тайм-аута проблематичными. Поэтому рекомендуется по возможности предпочитать наборы спискам.

Обратите внимание, что, хотя некоторые из этих ограничений могут или не могут быть удалены / улучшены в будущем, использование (одной) коллекции для хранения больших объемов данных является анти-шаблоном.

Карты (Maps)

Карта (map) - это (отсортированный) набор пар ключ-значение, где ключи уникальны, а карта сортируется по своим ключам. Вы можете определить и вставить карту с помощью:

|  |
| --- |
| CREATE TABLE users (  id text PRIMARY KEY,  name text,  favs map<text, text> // A map of text keys, and text values  );  INSERT INTO users (id, name, favs)  VALUES ('jsmith', 'John Smith', { 'fruit' : 'Apple', 'band' : 'Beatles' });  // Replace the existing map entirely.  UPDATE users SET favs = { 'fruit' : 'Banana' } WHERE id = 'jsmith'; |

Кроме того, карты поддерживают:

* Обновление или вставка одного, или нескольких элементов:

|  |
| --- |
| UPDATE users SET favs['author'] = 'Ed Poe' WHERE id = 'jsmith';  UPDATE users SET favs = favs + { 'movie' : 'Cassablanca', 'band' : 'ZZ Top' } WHERE id = 'jsmith'; |

* Удаление одного или нескольких элементов (если элемент не существует, удаление не выполняется, но ошибка не возникает):

|  |
| --- |
| DELETE favs['author'] FROM users WHERE id = 'jsmith';  UPDATE users SET favs = favs - { 'movie', 'band'} WHERE id = 'jsmith'; |

Обратите внимание, что для удаления нескольких элементов на карте (map) вы удаляете из нее набор (set) ключей.

Наконец, TTL разрешены как для INSERT, так и для UPDATE, но в обоих случаях набор TTL применяется только к вновь вставленным / обновленным элементам.

Другими словами:

|  |
| --- |
| UPDATE users USING TTL 10 SET favs['color'] = 'green' WHERE id = 'jsmith'; |

будет применен TTL только к записи { 'color' : 'green' }, остальная часть карты останется неизменной.

Наборы (Sets)

Набор (set) - это (отсортированный) набор уникальных значений. Вы можете определить и вставить карту с помощью:

|  |
| --- |
| CREATE TABLE images (  name text PRIMARY KEY,  owner text,  tags set<text> // A set of text values  );  INSERT INTO images (name, owner, tags)  VALUES ('cat.jpg', 'jsmith', { 'pet', 'cute' });  // Replace the existing set entirely  UPDATE images SET tags = { 'kitten', 'cat', 'lol' } WHERE name = 'cat.jpg'; |

Далее, наборы поддерживают:

* Добавление одного или нескольких элементов (поскольку это набор, вставка уже существующего элемента не выполняется):

|  |
| --- |
| UPDATE images SET tags = tags + { 'gray', 'cuddly' } WHERE name = 'cat.jpg'; |

* Удаление одного или нескольких элементов (если элемент не существует, удаление не выполняется, но ошибка не возникает):

|  |
| --- |
| UPDATE images SET tags = tags - { 'cat' } WHERE name = 'cat.jpg'; |

Наконец, для наборов (sets), TTL применяются только к вновь вставленным значениям.

Списки (Lists)

|  |
| --- |
| **Примечание.** |
| Как упоминалось выше и дополнительно обсуждалось в конце этого раздела, списки имеют ограничения и особые соображения производительности, которые вы должны принять во внимание перед их использованием. В общем, если вы можете использовать набор вместо списка, всегда предпочитайте набор. |

Список (list) - это (отсортированный) набор неуникальных значений, элементы которого упорядочены по положению в списке. Вы можете определить и вставить список с помощью:

|  |
| --- |
| CREATE TABLE plays (  id text PRIMARY KEY,  game text,  players int,  scores list<int> // A list of integers  )  INSERT INTO plays (id, game, players, scores)  VALUES ('123-afde', 'quake', 3, [17, 4, 2]);  // Replace the existing list entirely  UPDATE plays SET scores = [ 3, 9, 4] WHERE id = '123-afde'; |

Далее, списки поддерживают:

* Операции добавления appending и prepending значений к списку:

|  |
| --- |
| UPDATE plays SET players = 5, scores = scores + [ 14, 21 ] WHERE id = '123-afde';  UPDATE plays SET players = 6, scores = [ 3 ] + scores WHERE id = '123-afde'; |

|  |
| --- |
| **Внимание.** |
| Операции append и prepend не идемпотентны по своей природе. Так, в частности, если для одной из этих операций установлен таймаут, то повторная попытка операции небезопасна и может (или не может) привести к добавлению значения дважды. |

* Установка значения в определенной позиции в списке, который имеет уже существующий элемент для этой позиции. Если в списке нет позиции, будет выдана ошибка:

|  |
| --- |
| UPDATE plays SET scores[1] = 7 WHERE id = '123-afde'; |

* Удаление элемента по его позиции в списке, у которого есть уже существующий элемент для этой позиции. Если в списке нет позиции, будет выдана ошибка. Далее, когда операция удаляет элемент из списка, размер списка будет уменьшаться на один элемент, сдвигая позиции всех следующих элементов на один вперед:

|  |
| --- |
| DELETE scores[1] FROM plays WHERE id = '123-afde'; |

* Удаление всех вхождений определенных значений в списке (если определенный элемент вообще не встречается в списке, он просто игнорируется, и ошибка не выдается):

|  |
| --- |
| UPDATE plays SET scores = scores - [ 12, 21 ] WHERE id = '123-afde'; |

|  |
| --- |
| **Внимание.** |
| Установка и удаление элемента по позиции и удаление вхождений определенных значений требуют внутреннего чтения перед записью. Эти операции будут выполняться медленно и потреблять больше ресурсов, чем обычные обновления (за исключением условной записи, которая имеет свои собственные затраты). |

Наконец, для списков (lists), TTL применяются только к вновь вставленным значениям.

Типы, определяемые пользователем (User-defined Types (UDTs))

CQL поддерживает определение определяемых пользователем типов (UDT). Такой тип может быть создан, изменен и удален с помощью create\_type\_statement, alter\_type\_statement и drop\_type\_statement, описанных ниже. Но после создания UDT просто упоминается по его имени:

|  |
| --- |
| user\_defined\_type::= udt\_name  udt\_name::= [ keyspace\_name '.' ] identifier |

Создание UDT

Создание нового пользовательского типа выполняется с помощью оператора CREATE TYPE, определяемого следующим образом:

|  |
| --- |
| create\_type\_statement::= CREATE TYPE [ IF NOT EXISTS ] udt\_name  '(' field\_definition ( ',' field\_definition)\* ')'  field\_definition::= identifier cql\_type |

UDT имеет имя (используется для объявленных столбцов этого типа) и представляет собой набор именованных и типизированных полей. Имя полей может быть любого типа, включая коллекции или другой UDT. Например:

|  |
| --- |
| CREATE TYPE phone (  country\_code int,  number text,  );  CREATE TYPE address (  street text,  city text,  zip text,  phones map<text, phone>  );  CREATE TABLE user (  name text PRIMARY KEY,  addresses map<text, frozen<address>>  ); |

Что нужно помнить о UDT:

* Попытка создать уже существующий тип приведет к ошибке, если не используется опция IF NOT EXISTS. Если используется, оператор не будет работать, если тип уже существует.
* Тип внутренне привязан к пространству ключей, в котором он создается, и может использоваться только в этом пространстве ключей. При создании, если имя типа предваряется именем пространства ключей, оно создается в этом пространстве ключей. В противном случае он создается в текущем пространстве ключей.
* Что касается RT.KeyValue, UDT необходимо заморозить в большинстве случаев, отсюда и frozen<address> в приведенном выше определении таблицы.

Литералы UDT

После создания определяемого пользователем типа можно ввести значение с помощью литерала UDT:

|  |
| --- |
| udt\_literal::= '{' identifier ':' term ( ',' identifier ':' term)\* '}' |

Другими словами, литерал UDT похож на литерал карты (map), но его ключи - это имена полей типа. Например, можно вставить в таблицу, определенную в предыдущем разделе, используя:

|  |
| --- |
| INSERT INTO user (name, addresses)  VALUES ('z3 Pr3z1den7', {  'home' : {  street: '1600 Pennsylvania Ave NW',  city: 'Washington',  zip: '20500',  phones: { 'cell' : { country\_code: 1, number: '202 456-1111' },  'landline' : { country\_code: 1, number: '...' } }  },  'work' : {  street: '1600 Pennsylvania Ave NW',  city: 'Washington',  zip: '20500',  phones: { 'fax' : { country\_code: 1, number: '...' } }  }  }  ); |

Чтобы быть действительным, литерал UDT может включать только поля, определенные типом, к которому он относится, но он может пропускать некоторые поля (для них будет установлено значение NULL).

Изменение UDT

Существующий определяемый пользователем тип (UDT) можно изменить с помощью оператора ALTER TYPE:

|  |
| --- |
| alter\_type\_statement::= ALTER TYPE udt\_name alter\_type\_modification  alter\_type\_modification::= ADD field\_definition  | RENAME identifier TO identifier( identifier TO identifier )\* |

Вы можете:

* Добавить новое поле к типу (ALTER TYPE address ADD country text). Это новое поле будет NULL для любых значений типа, созданного до добавления.
* Переименовать поля типа.

|  |
| --- |
| ALTER TYPE address RENAME zip TO zipcode; |

Удаление UDT

Вы можете удалить существующий пользовательский тип с помощью оператора DROP TYPE:

|  |
| --- |
| drop\_type\_statement::= DROP TYPE [ IF EXISTS ] udt\_name |

Удаление типа приводит к немедленному и необратимому удалению этого типа. Однако попытка удалить тип, который все еще используется другим типом, таблицей или функцией, приведет к ошибке.

Если удаляемый тип не существует, будет возвращена ошибка, если не используется IF EXISTS, и в этом случае операция не выполняется.

Кортежи (Tuples)

CQL также поддерживает кортежи и типы кортежей (где элементы могут быть разных типов). Функционально кортежи могут быть анонимными UDT с анонимными полями. Типы кортежей и литералы кортежей определяются:

|  |
| --- |
| tuple\_type::= TUPLE '<' cql\_type( ',' cql\_type)\* '>'  tuple\_literal::= '(' term( ',' term )\* ')' |

и могут быть созданы:

|  |
| --- |
| CREATE TABLE durations (  event text,  duration tuple<int, text>,  );  INSERT INTO durations (event, duration) VALUES ('ev1', (3, 'hours')); |

В отличие от других составных типов, таких как коллекции и UDT, кортеж всегда frozen <frozen> (без использования ключевого слова frozen), и невозможно обновить только некоторые элементы кортежа (без обновления всего кортежа). Кроме того, литерал кортежа всегда должен иметь то же количество значений, что и объявленный в типе, кортежем которого он является (некоторые из этих значений могут быть нулевыми, но они должны быть явно объявлены как таковые).

Настраиваемые типы (Custom Types)

|  |
| --- |
| **Примечание.** |
| Настраиваемые типы существуют в основном для целей обратной совместимости, и их использование не рекомендуется. Их использование сложное, неудобное для пользователя, а других предоставленных типов, особенно определяемых пользователем, почти всегда должно быть достаточно. |

Настраиваемый тип определяется:

|  |
| --- |
| custom\_type::= string |

Настраиваемый тип - это строка (string), содержащая имя класса Java, который расширяет класс AbstractType на стороне сервера и который может быть загружен RT.KeyValue (таким образом, он должен быть в CLASSPATH каждого узла, на котором запущена RT.KeyValue). Этот класс будет определять, какие значения допустимы для типа и как сортируется время при использовании для столбца кластеризации. Для любых других целей значение настраиваемого типа такое же, как у blob объекта, и, в частности, может быть введено с использованием синтаксиса blob литерала.

Определение данных (Data Definition(DDL))

CQL хранит данные в таблицах (*tables*), схема которых определяет расположение данных в таблице. Таблицы расположены в пространствах ключей (*keyspace*). Пространство ключей определяет параметры, которые применяются ко всем таблицам пространства ключей. Стратегия репликации (replication strategy) является важным параметром пространства ключей, как и фактор репликации. Хорошее общее правило - одно пространство ключей на приложение. Обычно кластер определяет только одно пространство ключей для активного приложения.

В этом разделе описаны операторы, используемые для создания, изменения и удаления этих пространств ключей и таблиц.

Общие определения

Имена пространств ключей и таблиц определяются следующей грамматикой:

|  |
| --- |
| keyspace\_name::= name  table\_name::= [keyspace\_name '.' ] name  name::= unquoted\_name | quoted\_name  unquoted\_name::= re('[a-zA-Z\_0-9]\{1, 48}')  quoted\_name::= '"' unquoted\_name '"' |

Как пространство ключей, так и имя таблицы должны состоять только из буквенно-цифровых символов, не могут быть пустыми и ограничены размером до 48 символов (этот предел существует в основном для того, чтобы имена файлов (которые могут включать пространство ключей и имя таблицы) не выходили за пределы определенного файла системы). По умолчанию в именах пространства ключей и таблиц регистр не учитывается (myTable эквивалентен mytable), но чувствительность к регистру может быть установлена с помощью двойных кавычек ("myTable" отличается от mytable).

Кроме того, таблица всегда является частью пространства ключей, и имя таблицы может быть полностью определено пространством ключей, частью которого она является. Если не указано полное значение, предполагается, что таблица находится в текущем пространстве ключей (см. раздел USE ниже).

Кроме того, допустимые имена столбцов определяются как:

|  |
| --- |
| column\_name::= identifier |

Мы также определяем понятие опций оператора для использования в следующем разделе:

|  |
| --- |
| options::= option ( AND option )\*  option::= identifier '=' ( identifier  | constant  | map\_literal ) |

Создание пространства ключей (CREATE KEYSPACE)

Пространство ключей создается с помощью оператора CREATE KEYSPACE:

|  |
| --- |
| create\_keyspace\_statement::= CREATE KEYSPACE [ IF NOT EXISTS ] keyspace\_name  WITH options |

Например:

|  |
| --- |
| CREATE KEYSPACE excelsior  WITH replication = {'class': 'SimpleStrategy', 'replication\_factor' : 3};  CREATE KEYSPACE excalibur  WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}  AND durable\_writes = false; |

Попытка создать пространство ключей, которое уже существует, вернет ошибку, если не будет использована опция IF NOT EXISTS. Если используется, оператор будет отключен, если пространство ключей уже существует.

Поддерживаемые варианты:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя** | **Вид** | **Обязательно** | **По умолчанию** | **Описание** |
| replication | map | да | n/a | Стратегия репликации и параметры, используемые для пространства ключей (см. Подробности ниже). |
| durable\_writes | simple | no | true | Следует ли использовать журнал фиксации для обновлений в этом пространстве ключей (отключите эту опцию на свой страх и риск!). |

Свойство репликации (replication) является обязательным и должно содержать подопцию "класс" ('class'), определяющую желаемый класс стратегии репликации. Остальные подпараметры зависят от того, какая стратегия репликации используется. По умолчанию RT.KeyValue поддерживает следующие значения "класса" ('class'), приведенные ниже.

SimpleStrategy

Простая стратегия (SimpleStrategy), определяющая коэффициент репликации для распределения данных по всему кластеру. Как правило, это не лучший выбор для производственной среды, поскольку он не учитывает макеты центра обработки данных и может привести к сильно различающейся задержке запроса. Для производства используйте NetworkTopologyStrategy. SimpleStrategy поддерживает единственный обязательный аргумент:

|  |  |  |  |
| --- | --- | --- | --- |
| **Подпараметр** | **Тип** | **С какого момента** | **Описание** |
| 'replication\_factor' | int | all | Количество реплик для хранения на диапазон |

NetworkTopologyStrategy

Готовая к работе стратегия репликации, которая устанавливает коэффициент репликации независимо для каждого центра обработки данных. Остальные подпараметры представляют собой пары "ключ-значение" (key-value), в которых ключ установлен на имя центра обработки данных, а его значение - на соответствующий коэффициент репликации. Параметры:

|  |  |  |  |
| --- | --- | --- | --- |
| **Подпараметр** | **Тип** | **Описание** | **Описание** |
| 'replication\_factor' | int | all | Количество реплик для хранения на диапазон |
| '<datacenter>' |  |  | Количество реплик для хранения на диапазон в предоставленном центре обработки данных. |

При последующем изменении пространств ключей и изменении replication\_factor автоматическое расширение только *добавит* новые центры обработки данных в целях безопасности, оно не изменит существующие центры обработки данных и не удалит их, даже если они больше не входят в кластер. Если вы хотите удалить центры обработки данных при установке параметра replication\_factor, явно обнулите центр обработки данных, для которого вы хотите иметь нулевые реплики.

Пример автоматического расширения центров обработки данных с двумя центрами обработки данных: DC1 и DC2:

|  |
| --- |
| CREATE KEYSPACE excalibur  WITH replication = {'class': 'NetworkTopologyStrategy', 'replication\_factor' : 3};  DESCRIBE KEYSPACE excalibur; |

Приведет к:

|  |
| --- |
| CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3', 'DC2': '3'} AND durable\_writes = true; |

Пример автоматического расширения и переопределения центра обработки данных:

|  |
| --- |
| CREATE KEYSPACE excalibur  WITH replication = {'class': 'NetworkTopologyStrategy', 'replication\_factor' : 3, 'DC2': 2};  DESCRIBE KEYSPACE excalibur; |

Приведет к:

|  |
| --- |
| CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3', 'DC2': '2'} AND durable\_writes = true; |

Пример, исключающий центр обработки данных при использовании replication\_factor:

|  |
| --- |
| CREATE KEYSPACE excalibur  WITH replication = {'class': 'NetworkTopologyStrategy', 'replication\_factor' : 3, 'DC2': 0};  DESCRIBE KEYSPACE excalibur; |

Приведет к:

|  |
| --- |
| CREATE KEYSPACE excalibur WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1': '3'} AND durable\_writes = true; |

Если временная репликация (transient replication) включена, временные реплики можно настроить как для SimpleStrategy, так и для NetworkTopologyStrategy путем определения факторов репликации в формате '<total\_replicas>/<transient\_replicas>'

Например, это пространство ключей будет иметь 3 реплики в DC1, 1 из которых временная, и 5 реплик в DC2, 2 из которых временные:

|  |
| --- |
| CREATE KEYSPACE some\_keyspace  WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : '3/1'', 'DC2' : '5/2'}; |

USE

Оператор USE изменяет текущее пространство ключей на указанное пространство ключей. Ряд объектов в CQL привязан к пространству ключей (таблицы, определяемые пользователем типы, функции и т. Д.), А текущее пространство ключей является пространством ключей по умолчанию, используемым, когда эти объекты упоминаются в запросе без полного имени (без префикса имени пространства ключей). Оператор USE определяет пространство ключей для использования в качестве аргумента:

|  |
| --- |
| use\_statement::= USE keyspace\_name |

Используя CQL:

|  |
| --- |
| USE excelsior; |

ALTER KEYSPACE

Оператор ALTER KEYSPACE изменяет параметры пространства ключей:

|  |
| --- |
| alter\_keyspace\_statement::= ALTER KEYSPACE keyspace\_name  WITH options |

Например:

|  |
| --- |
| ALTER KEYSPACE excelsior  WITH replication = {'class': 'SimpleStrategy', 'replication\_factor' : 4}; |

Поддерживаемые параметры такие же, как для создания пространства ключей.

DROP KEYSPACE

Удаление пространства ключей выполняется с помощью оператора DROP KEYSPACE:

|  |
| --- |
| drop\_keyspace\_statement::= DROP KEYSPACE [ IF EXISTS ] keyspace\_name |

Например:

|  |
| --- |
| DROP KEYSPACE excelsior; |

Удаление пространства ключей приводит к немедленному и необратимому удалению этого пространства ключей, включая все таблицы, пользовательские типы, пользовательские функции и все данные, содержащиеся в этих таблицах.

Если пространство ключей не существует, оператор вернет ошибку, если только не используется IF EXISTS, и в этом случае операция не выполняется.

CREATE TABLE

Для создания новой таблицы используется оператор CREATE TABLE:

|  |
| --- |
| create\_table\_statement::= CREATE TABLE [ IF NOT EXISTS ] table\_name '('  column\_definition ( ',' column\_definition )\*  [ ',' PRIMARY KEY '(' primary\_key ')' ]  ')' [ WITH table\_options ]  column\_definition::= column\_name cql\_type [ STATIC ] [ PRIMARY KEY]  primary\_key::= partition\_key [ ',' clustering\_columns ]  partition\_key::= column\_name | '(' column\_name ( ',' column\_name )\* ')'  clustering\_columns::= column\_name ( ',' column\_name )\*  table\_options:=: COMPACT STORAGE [ AND table\_options ]  | CLUSTERING ORDER BY '(' clustering\_order ')'  [ AND table\_options ] | options  clustering\_order::= column\_name (ASC | DESC) ( ',' column\_name (ASC | DESC) )\* |

Например, вот несколько операторов CQL для создания таблиц:

|  |
| --- |
| CREATE TABLE monkey\_species (  species text PRIMARY KEY,  common\_name text,  population varint,  average\_size int  ) WITH comment='Important biological records';  CREATE TABLE timeline (  userid uuid,  posted\_month int,  posted\_time uuid,  body text,  posted\_by text,  PRIMARY KEY (userid, posted\_month, posted\_time)  ) WITH compaction = { 'class' : 'LeveledCompactionStrategy' };  CREATE TABLE loads (  machine inet,  cpu int,  mtime timeuuid,  load float,  PRIMARY KEY ((machine, cpu), mtime)  ) WITH CLUSTERING ORDER BY (mtime DESC); |

Таблица CQL имеет имя и состоит из набора строк. Создание таблицы сводится к определению того, какие столбцы будут в каждой строке, какие из этих столбцов составляют первичный ключ (primary key), а также определенных опций для таблицы.

Попытка создать уже существующую таблицу вернет ошибку, если не будет использована директива IF NOT EXISTS. Если используется, оператор будет отключен, если таблица уже существует.

Определения столбцов (Column definitions)

Каждая строка в таблице CQL будет иметь предварительно определенные столбцы, определенные при создании таблицы. Столбцы можно добавить позже с помощью оператора alter.

column\_definition состоит из имени столбца и его типа, ограничивая значения, которые принимаются для этого столбца. Кроме того, определение столбца может иметь следующие модификаторы:

* STATIC: объявляет столбец как статический столбец (static column)
* PRIMARY KEY: объявляет столбец как единственный компонент первичного ключа таблицы

Статические столбцы (Static columns)

Некоторые столбцы могут быть объявлены как STATIC в определении таблицы. Статический столбец будет "общим" для всех строк, принадлежащих одному и тому же разделу (с одним и тем же ключом раздела (partition key)).

Например, код:

|  |
| --- |
| CREATE TABLE t (  pk int,  t int,  v text,  s text static,  PRIMARY KEY (pk, t)  );  INSERT INTO t (pk, t, v, s) VALUES (0, 0, 'val0', 'static0');  INSERT INTO t (pk, t, v, s) VALUES (0, 1, 'val1', 'static1');  SELECT \* FROM t; |

Результат:

|  |
| --- |
| pk | t | v | s  ----+---+--------+-----------  0 | 0 | 'val0' | 'static1'  0 | 1 | 'val1' | 'static1' |

Как можно видеть, значение s одинаково (static1) для обеих строк в разделе (ключ раздела - pk, и обе строки находятся в одном разделе): вторая вставка переопределяет значение для s.

Использование статических столбцов имеет следующие ограничения:

* Таблица без столбцов кластеризации не может иметь статических столбцов. В таблице без столбцов кластеризации каждый раздел имеет только одну строку, поэтому каждый столбец по своей природе статичен.
* Статическими могут быть только столбцы, не являющиеся первичными ключами.

Первичный ключ (Primary key)

Внутри таблицы строка однозначно идентифицируется своим PRIMARY KEY, и, следовательно, все таблицы **должны** определять один **ПЕРВИЧНЫЙ КЛЮЧ**. PRIMARY KEY состоит из одного или нескольких столбцов, определенных в таблице. Синтаксически первичный ключ определяется фразой PRIMARY KEY, за которой следует разделенный запятыми список имен столбцов в круглых скобках. Если первичный ключ имеет только один столбец, вы также можете добавить фразу PRIMARY KEY к этому столбцу в определении таблицы. Порядок столбцов в определении первичного ключа определяет ключ раздела и столбцы кластеризации.

Первичный ключ CQL состоит из двух частей:

1. **Ключ раздела (partition key)**

* Это первый компонент определения первичного ключа. Это может быть один столбец или, используя дополнительный набор скобок, может быть несколько столбцов. Таблица должна иметь хотя бы один ключ раздела, наименьшее возможное определение таблицы:

|  |
| --- |
| CREATE TABLE t (k text PRIMARY KEY); |

1. **Столбцы кластеризации (clustering columns)**

* Столбцы - это столбцы, следующие за ключом раздела в определении первичного ключа. Порядок этих столбцов определяет порядок кластеризации.

Вот некоторые примеры определения первичного ключа:

* PRIMARY KEY (a): a - это ключ одного раздела, столбцы кластеризации отсутствуют.
* PRIMARY KEY (a, b, c): a - ключ одного раздела, а b и c - столбцы кластеризации.
* PRIMARY KEY ((a, b), c): a и b составляют *составной* ключ раздела, а c - столбец кластеризации.

|  |
| --- |
| **Важно.** |
| Первичный ключ однозначно идентифицирует строку в таблице, как описано выше. Следствием этой уникальности является то, что если другая строка вставляется с использованием того же первичного ключа, то происходит UPSERT, и существующая строка с тем же первичным ключом заменяется. Столбцы, не являющиеся частью первичного ключа, не могут определять уникальность. |

Ключ раздела (partition key)

В таблице, CQL определяет понятие раздела, который определяет расположение данных в кластере RT.KeyValue. Раздел - это набор строк, которые имеют одно и то же значение для своего ключа раздела.

Обратите внимание, что если ключ раздела состоит из нескольких столбцов, то строки принадлежат одному разделу, если они имеют одинаковые значения для всех этих столбцов ключа раздела. Хэш вычисляется из столбцов ключа раздела, и это значение хеша определяет расположение раздела. Так, например, с учетом следующего определения и содержания таблицы:

|  |
| --- |
| CREATE TABLE t (  a int,  b int,  c int,  d int,  PRIMARY KEY ((a, b), c, d)  );  INSERT INTO t (a, b, c, d) VALUES (0,0,0,0);  INSERT INTO t (a, b, c, d) VALUES (0,0,1,1);  INSERT INTO t (a, b, c, d) VALUES (0,1,2,2);  INSERT INTO t (a, b, c, d) VALUES (0,1,3,3);  INSERT INTO t (a, b, c, d) VALUES (1,1,4,4);  SELECT \* FROM t; |

Приведет к:

|  |
| --- |
| a | b | c | d  ---+---+---+---  0 | 0 | 0 | 0  0 | 0 | 1 | 1  0 | 1 | 2 | 2  0 | 1 | 3 | 3  1 | 1 | 4 | 4  (5 rows) |

Строки 1 и 2 находятся в одном разделе, потому что оба столбца a и b равны нулю.

Строки 3 и 4 находятся в одном разделе, но в другом, потому что столбец a равен нулю, а столбец b равен 1 в обеих строках.

Строка 5 находится в третьем разделе, потому что оба столбца a и b равны 1.

Обратите внимание, что таблица всегда имеет ключ раздела, и что если таблица не имеет столбцов кластеризации (clustering columns), то каждый раздел этой таблицы имеет одну строку. потому что ключ раздела, составной или иной, определяет одно местоположение.

Наиболее важным свойством раздела является то, что все строки, принадлежащие одному разделу, гарантированно хранятся на одном и том же наборе узлов-реплик. Другими словами, ключ раздела таблицы определяет, какие строки будут локализованы на одном узле кластера. Локализация данных важна для эффективного извлечения данных, требуя, чтобы координатор RT.KeyValue контактировал с как можно меньшим количеством узлов. Однако у этой гарантии есть обратная сторона: все строки, имеющие общий ключ раздела, будут храниться на одном узле, создавая точку доступа как для чтения, так и для записи. При выборе первичного ключа, который группирует строки таблицы, помогает пакетным обновлениям и может гарантировать, что обновления являются атомарными и выполняются изолированно, размеры разделов должны быть "правильными, не слишком большими и не слишком маленькими".

Моделирование данных, которое учитывает шаблоны запросов и назначает первичные ключи на основе запросов, будет иметь наименьшую задержку при выборке данных.

Столбцы кластеризации (Clustering columns)

Столбцы кластеризации таблицы определяют порядок кластеризации для раздела этой таблицы. Для данного раздела (partition) все строки упорядочены в этом порядке кластеризации. Кластеризация столбцов также добавляет уникальности строке в таблице.

Например, учитывая:

|  |
| --- |
| CREATE TABLE t2 (  a int,  b int,  c int,  d int,  PRIMARY KEY (a, b, c)  );  INSERT INTO t2 (a, b, c, d) VALUES (0,0,0,0);  INSERT INTO t2 (a, b, c, d) VALUES (0,0,1,1);  INSERT INTO t2 (a, b, c, d) VALUES (0,1,2,2);  INSERT INTO t2 (a, b, c, d) VALUES (0,1,3,3);  INSERT INTO t2 (a, b, c, d) VALUES (1,1,4,4);  SELECT \* FROM t2; |

Приведет к:

|  |
| --- |
| a | b | c | d  ---+---+---+---  1 | 1 | 4 | 4  0 | 0 | 0 | 0  0 | 0 | 1 | 1  0 | 1 | 2 | 2  0 | 1 | 3 | 3  (5 rows) |

Строка 1 находится в одном разделе, а строки 2–5 - в другом. Порядок отображения также отличается.

При более внимательном рассмотрении четырех строк в одном разделе столбец кластеризации b определяет порядок, в котором эти строки отображаются. В то время как ключ раздела таблицы группирует строки на одном узле, столбцы кластеризации управляют тем, как эти строки хранятся на узле.

Такая сортировка позволяет очень эффективно извлекать диапазон строк в разделе:

|  |
| --- |
| SELECT \* FROM t2 WHERE a = 0 AND b > 0 and b <= 3; |

Приведет к:

|  |
| --- |
| a | b | c | d  ---+---+---+---  0 | 1 | 2 | 2  0 | 1 | 3 | 3  (2 rows) |

Параметры таблицы

Таблица CQL имеет ряд параметров, которые можно установить при создании (и, для большинства из них, изменить позже). Эти параметры указываются после ключевого слова WITH.

Один важный параметр, который нельзя изменить после создания, CLUSTERING ORDER BY, влияет на то, как можно выполнять запросы к таблице. Здесь стоит поговорить подробнее.

Порядок кластеризации (Clustering order)

Порядок кластеризации таблицы определяется столбцами кластеризации. По умолчанию порядок кластеризации, возрастающий для типов данных столбца кластеризации. Например, целые числа располагаются от 1, 2, … n, а текст - от A до Z.

Параметр таблицы CLUSTERING ORDER BY использует список столбцов кластеризации, разделенных запятыми, каждый из которых задается либо для ASC (для возрастающего порядка), либо для DESC (для \_descending order). По умолчанию для всех столбцов кластеризации используется возрастание, если не установлен параметр CLUSTERING ORDER BY. Этот параметр, в основном, является подсказкой для механизма хранения, который изменяет порядок, в котором он сохраняет строку. Остерегайтесь последствий установки этой опции:

* Он изменяет порядок возрастания результатов по умолчанию при запросе с помощью оператора SELECT без предложения ORDER BY.
* Он ограничивает использование предложения ORDER BY в операторах SELECT этой таблицы. Результаты можно упорядочить только с исходным или обратным порядком кластеризации. Предположим, вы создали таблицу с двумя столбцами кластеризации a и b, определенными WITH CLUSTERING ORDER BY (a DESC, b ASC). Запросы в таблице могут использовать ORDER BY (a DESC, b ASC) или ORDER BY (a ASC, b DESC). Смешанный порядок, например ORDER BY (a ASC, b ASC) или ORDER BY (a DESC, b DESC), не вернет ожидаемый порядок.
* Это влияет на производительность запросов. Запросы в обратном порядке кластеризации выполняются медленнее, чем в порядке возрастания по умолчанию. Если вы планируете выполнять запросы в основном в порядке убывания, объявите порядок кластеризации в схеме таблицы с помощью WITH CLUSTERING ORDER BY (). Эта оптимизация обычна для временных рядов, чтобы извлекать данные от самых новых до самых старых.

Другие параметры таблицы

В **Таблице 2.** приведены параметры, поддерживаемые таблицей:

**Таблица 2. Поддерживаемые параметры**

|  |  |  |  |
| --- | --- | --- | --- |
| **Параметр** | **Тип** | **По умолчанию** | **Описание** |
| comment | simple | none | Комментарий в свободной форме, удобочитаемый. |
| speculative\_retry | simple | 99PERCENTIL | Варианты спекулятивной повторной попытки. |
| cdc | boolean | false | Создайте журнал отслеживания измененных данных (CDC) в таблице. |
| additional\_write\_policy | simple | 99PERCENTIL | Такой же как speculative\_retry |
| gc\_grace\_seconds | simple | 864000 | Время ожидания перед сбором мусора надгробия (маркеры удаления) Time to wait before garbage collecting tombstones (deletion markers). |
| bloom\_filter\_fp\_chance | simple | 0.00075 | Целевая вероятность ложного срабатывания фильтров sstable bloom. Указанные фильтры Блума будут иметь размер, обеспечивающий предоставленную вероятность, таким образом, снижение этого значения повлияет на размер фильтров Блума в памяти и на диске. |
| default\_time\_to\_live | simple | 0 | Время истечения срока действия по умолчанию ("TTL") в секундах для таблицы. |
| compaction | map | see below | Варианты сжатия. |
| compression | map | see below | Варианты сжатия. |
| caching | map | see below | Варианты кеширования. |
| memtable\_flush\_period\_in\_ms | simple | 0 | Время (в мс) до того, как RT.KeyValue сбрасывает таблицы памяти на диск. |
| read\_repair | simple | BLOCKING | Устанавливает режим восстановления при чтении. |

Параметры спекулятивной повторной попытки (Speculative retry options)

По умолчанию координаторы чтения RT.KeyValue запрашивают столько реплик, сколько необходимо для соответствия уровням согласованности: один для уровня согласованности ONE, кворум для QUORUM и т. д. speculative\_retry определяет, когда координаторы могут запрашивать дополнительные реплики, полезное действие, когда реплики работают медленно или не отвечают. Спекулятивные попытки уменьшают задержку. Параметр speculative\_retry настраивает защиту от быстрого чтения, когда координатор отправляет больше запросов, чем необходимо для удовлетворения уровня согласованности.

|  |
| --- |
| **Важно.** |
| Частое чтение с дополнительных реплик может снизить производительность кластера. В случае сомнений оставьте значение по умолчанию 99PERCENTILE. |

Политика спекулятивных повторных попыток принимает в качестве параметра одну строку:

* NONE
* ALWAYS
* 99PERCENTILE (PERCENTILE)
* 50MS (CUSTOM)

Пример настройки спекулятивного повтора (speculative retry) устанавливает настраиваемое значение:

|  |
| --- |
| ALTER TABLE users WITH speculative\_retry = '10ms'; |

В этом примере для настройки используется процентиль:

|  |
| --- |
| ALTER TABLE users WITH speculative\_retry = '99PERCENTILE'; |

Настройка процентиля может иметь неприятные последствия. Если один хост становится недоступным, это может увеличить процентили. Значение p99 не будет спекулировать, как предполагалось, потому что значение указанного процентиля увеличилось слишком сильно. Если для уровня согласованности установлено значение ALL, все реплики опрашиваются независимо от настройки спекулятивного повтора.

RT.KeyValue поддерживает нечувствительность к регистру для значений спекулятивных повторных попыток. Например, присвоение значения none, None или NONE имеет тот же эффект.

В **Таблице 3**. Представлены дополнительно добавляющиеся значения:

**Таблица 3. Дополнительные значения**

|  |  |  |
| --- | --- | --- |
| **Формат** | **Пример** | **Описание** |
| XPERCENTILE | 90.5PERCENTILE | Координаторы записывают среднее время ответа для каждой таблицы для всех реплик. Если реплика занимает больше, чем X процентов среднего времени ответа этой таблицы, координатор запрашивает дополнительную реплику. X должен быть от 0 до 100. |
| XP | 90.5P | Такой же как XPERCENTILE |
| Yms | 25ms | Если реплике требуется более Y миллисекунд для ответа, координатор запрашивает дополнительную реплику. |
| MIN(XPERCENTILE,YMS) | MIN(99PERCENTILE,35MS) | Гибридная политика, которая использует либо указанный процентиль, либо фиксированные миллисекунды в зависимости от того, какое значение ниже во время расчета. Параметры: XPERCENTILE, XP или Yms. Этот параметр помогает защитить от одного медленного экземпляра. |
| MAX(XPERCENTILE,YMS) ALWAYS NEVER | MAX(90.5P,25ms) | Гибридная политика, которая использует либо указанный процентиль, либо фиксированные миллисекунды в зависимости от того, какое значение выше во время расчета. |

В RT.KeyValue присутствует поддержка гибридных спекулятивных повторных попыток MIN() и MAX() с сочетанием и соответствием MIN(), MAX(), MIN(), MIN() или MAX(), MAX(). Гибридный режим по-прежнему будет предполагать, если нормальное значение p99 для таблицы <50 мсек, минимального значения. Но если уровень p99 превышает максимальное значение, то это значение можно использовать. В гибридном значении одно значение должно быть фиксированным значением времени (мс), а другое - значением процентиля.

Чтобы проиллюстрировать варианты, все следующие примеры являются допустимыми:

|  |
| --- |
| min(99percentile,50ms)  max(99p,50MS)  MAX(99P,50ms)  MIN(99.9PERCENTILE,50ms)  max(90percentile,100MS)  MAX(100.0PERCENTILE,60ms) |

Параметр additional\_write\_policy указывает порог, при котором запись кворума будет обновлена для включения временных реплик.

Параметры уплотнения (Compaction options)

Параметры уплотнения (compaction) должны минимально определять подпараметр 'class', чтобы указать используемый класс стратегии уплотнения. Поддерживаемые классы:

* 'SizeTieredCompactionStrategy', STCS (по умолчанию)
* 'LeveledCompactionStrategy', LCS
* 'TimeWindowCompactionStrategy', TWCS

'DateTieredCompactionStrategy' также поддерживается, но не рекомендуется; Следует использовать 'TimeWindowCompactionStrategy'. Если требуется настраиваемая стратегия, укажите полное имя класса в виде строковой константы.

Все стратегии по умолчанию поддерживают ряд общих параметров, а также параметры, специфичные для выбранной стратегии.

Параметры сжатия (Compression options)

Параметры сжатия (compression) определяют, будут ли сжиматься SSTables таблицы и каким образом. Сжатие настраивается для каждой таблицы в качестве необязательного аргумента для CREATE TABLE или ALTER TABLE. В **Таблице 4.** Представлены варианты подпараметров сжатия.

**Таблица 4. Варианты подпараметров сжатия.**

|  |  |  |
| --- | --- | --- |
| **Подпараметр** | **По умолчанию** | **Описание** |
| class | LZ4Compressor | Используемый алгоритм сжатия. Компрессор по умолчанию: LZ4Compressor, SnappyCompressor, DeflateCompressor и ZstdCompressor. Используйте 'enabled' : false, чтобы отключить сжатие. Пользовательский компрессор может быть предоставлен путем указания полного имени класса в виде строковой константы. |
| enabled | true | Включение / отключение стабильного сжатия. Если для параметра enabled установлено значение false, никакие другие параметры указывать не нужно. |
| chunk\_length\_in\_kb | 64 | На диске SSTables сжимаются по блокам (для возможности произвольного чтения). Этот параметр определяет размер (в КБ) указанного блока. См. Примечание для получения дополнительной информации. |
| crc\_check\_chance | 1.0 | Определяет, насколько вероятно, что RT.KeyValue будет проверять контрольную сумму для каждого блока сжатия во время чтения. |
| compression\_level | 3 | Уровень сжатия. Применимо только для ZstdCompressor. Принимает значения от -131072 до 22. |

|  |
| --- |
| **Примечание.** |
| Большие значения могут улучшить степень сжатия, но увеличат минимальный размер данных, считываемых с диска для чтения. Значение по умолчанию является оптимальным значением для сжатия таблиц. Длина блока должна быть в степени 2 при вычислении номера блока по смещению несжатого файла. Размер блока можно регулировать на основе шаблонов доступа для чтения / записи, таких как:   * Сколько данных обычно запрашивается за один раз * Средний размер строк в таблице |

Например, чтобы создать таблицу с LZ4Compressor и chunk\_length\_in\_kb размером 4 КБ:

|  |
| --- |
| CREATE TABLE simple (  id int,  key text,  value text,  PRIMARY KEY (key, value)  ) WITH compression = {'class': 'LZ4Compressor', 'chunk\_length\_in\_kb': 4}; |

Параметры кеширования (Caching options)

Кэширование оптимизирует использование кэш-памяти таблицы. Кэшированные данные взвешиваются по размеру и частоте доступа. Параметры кэширования (caching) позволяют настроить как кеш ключей (key cache), так и кеш строк (row cache) для таблицы. Доступны следующие подпараметры: В **Таблице 5.** Представлены подпараметры сжатия.

**Таблица 5. Подпараметры сжатия**

|  |  |  |
| --- | --- | --- |
| **Параметр** | **По умолчанию** | **Описание** |
| keys | ALL | Кэшировать ли ключи (ключевой кеш) для этой таблицы. Допустимые значения: ALL и NONE. |
| rows\_per\_partition | NONE | Количество строк для кеширования на раздел (кеш строк). Если указано целое число n, первые n запрашиваемых строк раздела будут кэшированы. Допустимые значения: ALL, чтобы кэшировать все строки запрошенного раздела, или NONE, чтобы отключить кеширование строк. |

Например, чтобы создать таблицу с ключевым кешем и 10 кешированными строками на раздел:

|  |
| --- |
| CREATE TABLE simple (  id int,  key text,  value text,  PRIMARY KEY (key, value)  ) WITH caching = {'keys': 'ALL', 'rows\_per\_partition': 10}; |

Параметры read repair (Repair options)

Параметры read\_repair настраивают поведение восстановления при чтении, настраивая для различных характеристик производительности и согласованности.

В **Таблице 6.** Представлены значения read\_repair:

**Таблица 6. Значения** read\_repair

|  |  |  |
| --- | --- | --- |
| **Параметр** | **По умолчанию** | **Описание** |
| BLOCKING | yes | Если запускается восстановление при чтении, чтение блокирует записи, отправленные в другие реплики, до тех пор, пока записи не достигнут уровня согласованности. |
| NONE | no | Если установлено, координатор согласовывает любые различия между репликами, но не пытается их исправить. |

Поведение восстановления при чтении влияет на два свойства согласованности:

* Монотонное чтение кворума (Monotonic quorum reads): Монотонное чтение кворума препятствует тому, чтобы при некоторых обстоятельствах чтение возвращалось назад во времени. Когда монотонное чтение кворума не обеспечивается и запись не достигает кворума реплик, значения чтения могут быть видны при одном чтении, а затем исчезнуть при последующем чтении. BLOCKING обеспечивает такое поведение.
* Атомарность записи (write atomicity): атомарность записи предотвращает возвращение, при чтении, частично примененной записи. RT.KeyValue пытается обеспечить атомарность записи на уровне разделов, но поскольку при восстановлении чтения восстанавливаются только данные, охватываемые оператором SELECT, восстановление при чтении может нарушить атомарность записи, когда данные читаются на более детальном уровне, чем они записываются. Например, восстановление при чтении может нарушить атомарность записи, если вы записываете несколько строк в кластерный раздел в пакете, но затем выбираете одну строку, указав столбец кластеризации в операторе SELECT. NONE обеспечивает такое поведение.

Прочее

Добавление новых столбцов (см. ALTER TABLE) - это операция с постоянным временем. Таким образом, нет необходимости предвидеть будущее использование при первоначальном создании таблицы.

ALTER TABLE

Для изменения существующей таблицы используется оператор ALTER TABLE:

|  |
| --- |
| alter\_table\_statement::= ALTER TABLE table\_name alter\_table\_instruction  alter\_table\_instruction::= ADD column\_name cql\_type ( ',' column\_name cql\_type )\*  | DROP column\_name ( column\_name )\*  | WITH options |

Например:

|  |
| --- |
| ALTER TABLE addamsFamily ADD gravesite varchar;  ALTER TABLE addamsFamily  WITH comment = 'A most excellent and useful table'; |

Оператор ALTER TABLE может:

* ДОБАВИТЬ (ADD) новый столбец в таблицу. Первичный ключ таблицы нельзя изменить. Таким образом, новый столбец не может быть частью первичного ключа. Добавление столбца - это операция с постоянным временем, зависящая от количества данных в таблице.
* УДАЛИТЬ (DROP) столбец из таблицы. Эта команда удаляет столбец и все его содержимое. Имейте в виду, что, хотя столбец сразу становится недоступным, его содержимое лениво удаляется во время уплотнения. Из-за этого ленивого удаления команда является постоянной по времени операцией, зависящей от количества данных в таблице. Кроме того, важно знать, что после удаления столбца, столбец с тем же именем может быть повторно добавлен, если только удаленный столбец не был незамороженным столбцом, таким как коллекция.

|  |
| --- |
| **Внимание.** |
| Удаление столбца предполагает, что метки времени, используемые для значения этого столбца, являются "реальными" метками времени в микросекундах. Использование "реальных" временных меток в микросекундах осущетсвляется по умолчанию и **настоятельно** рекомендуется, но поскольку RT.KeyValue позволяет клиенту указывать любую временную метку для любой таблицы, теоретически возможно использовать другое соглашение. Пожалуйста, имейте в виду, что если вы сделаете это, удаление столбца будет выполнено неправильно. |

* Используйте WITH, чтобы изменить параметр таблицы. Поддерживаемые параметры такие же, как и при создании таблицы, за исключением CLUSTERING ORDER. Однако установка любых дополнительных параметров уплотнения (compaction) приведет к удалению **ВСЕХ** предыдущих параметров уплотнения (compaction), поэтому вам необходимо повторно указать все дополнительные параметры, которые вы хотите сохранить. То же верно и для подпараметров сжатия (compression).

DROP TABLE

Для удаления таблицы используется оператор DROP TABLE:

|  |
| --- |
| drop\_table\_statement::= DROP TABLE [ IF EXISTS ] table\_name |

Удаление таблицы приводит к немедленному и необратимому удалению таблицы, включая все данные, которые она содержит.

Если таблица не существует, оператор вернет ошибку, если только не используется IF EXISTS, когда операция не выполняется.

TRUNCATE

Таблица может быть усечена с помощью оператора TRUNCATE:

|  |
| --- |
| truncate\_statement::= TRUNCATE [ TABLE ] table\_name |

TRUNCATE TABLE foo - предпочтительный синтаксис для согласованности с другими операторами DDL. Однако таблицы - единственный объект, который в настоящее время можно усечь, и ключевое слово TABLE можно опустить.

При усечении таблицы безвозвратно удаляются все существующие данные из таблицы, но сама таблица не удаляется.

Манипулирование данными (Data Manipulation (DML))

В этом разделе описаны операторы, поддерживаемые CQL для вставки, обновления, удаления и запроса данных.

SELECT

Запрос данных из данных выполняется с помощью оператора SELECT:

|  |
| --- |
| select\_statement::= SELECT [ JSON | DISTINCT ] ( select\_clause | '\*' )  FROM `table\_name`  [ WHERE `where\_clause` ]  [ GROUP BY `group\_by\_clause` ]  [ ORDER BY `ordering\_clause` ]  [ PER PARTITION LIMIT (`integer` | `bind\_marker`) ]  [ LIMIT (`integer` | `bind\_marker`) ]  [ ALLOW FILTERING ]  select\_clause::= `selector` [ AS `identifier` ] ( ',' `selector` [ AS `identifier` ] )  selector::== `column\_name`  | `term`  | CAST '(' `selector` AS `cql\_type` ')'  | `function\_name` '(' [ `selector` ( ',' `selector` )\_ ] ')'  | COUNT '(' '\_' ')'  where\_clause::= `relation` ( AND `relation` )\*  relation::= column\_name operator term  '(' column\_name ( ',' column\_name )\* ')' operator tuple\_literal  TOKEN '(' column\_name# ( ',' column\_name )\* ')' operator term  operator::= '=' | '<' | '>' | '<=' | '>=' | '!=' | IN | CONTAINS | CONTAINS KEY  group\_by\_clause::= column\_name ( ',' column\_name )\*  ordering\_clause::= column\_name [ ASC | DESC ] ( ',' column\_name [ ASC | DESC ] )\* |

Например:

|  |
| --- |
| SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);  SELECT JSON name, occupation FROM users WHERE userid = 199;  SELECT name AS user\_name, occupation AS user\_occupation FROM users;  SELECT time, value  FROM events  WHERE event\_type = 'myEvent'  AND time > '2011-02-03'  AND time <= '2012-01-01'  SELECT COUNT (\*) AS user\_count FROM users; |

Оператор SELECT считывает один или несколько столбцов для одной или нескольких строк в таблице. Он возвращает набор результатов строк, соответствующих запросу, где каждая строка содержит значения для выбора, соответствующего запросу. Кроме того, к результату могут быть применены функции, включая агрегаты.

Оператор SELECT содержит как минимум предложение выбора и имя таблицы, в которой выполняется выбор. CQL **не** выполняет объединения или подзапросы, а оператор выбора применяется только к одной таблице. Оператор select также может иметь предложение where, которое может еще больше сузить результаты запроса. Дополнительные пункты могут упорядочивать или ограничивать результаты. Наконец, запросы, требующие полной кластерной фильтрации, могут добавлять ALLOW FILTERING к любому запросу.

Selection clause

select\_clause определяет, какие столбцы будут опрошены и возвращены в наборе результатов. В этом предложении также могут применяться преобразования для применения к результату перед возвратом. Предложение выбора состоит из списка определенных селекторов (*selectors*), разделенных запятыми, или, альтернативно, символом подстановки (\*) для выбора всех столбцов, определенных в таблице.

Selectors

Селектор может быть:

* Имя столбца выбранной таблицы, чтобы получить значения для этого столбца.
* Термин, который обычно используется вложенным в другие селекторы, такие как функции (если термин выбран напрямую, тогда в соответствующем столбце набора результатов будет просто значение этого термина для каждой возвращаемой строки).
* Приведение, которое позволяет преобразовать вложенный селектор в (совместимый) тип.
* Вызов функции, аргументы которой сами являются селектором. См. Раздел о функциях для получения более подробной информации.
* Специальный вызов COUNT(\*) функции COUNT, которая подсчитывает все ненулевые результаты.

Aliases

Каждому селектору верхнего уровня также можно присвоить алиас (используя AS). В этом случае имя соответствующего столбца в наборе результатов будет именем алиаса.

Например:

|  |
| --- |
| SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);  SELECT JSON name, occupation FROM users WHERE userid = 199;  SELECT name AS user\_name, occupation AS user\_occupation FROM users;  SELECT time, value  FROM events  WHERE event\_type = 'myEvent'  AND time > '2011-02-03'  AND time <= '2012-01-01'  SELECT COUNT (\*) AS user\_count FROM users; |

|  |
| --- |
| **Примечание.** |
| В настоящее время алиасы не распознаются в предложениях WHERE или ORDER BY в операторах. Вместо этого необходимо использовать исходное имя столбца. |

WRITETIME и функция TTL

Выбор (selection) поддерживает две специальные функции, которые больше нигде не разрешены: WRITETIME и TTL. Обе функции принимают только один аргумент - имя столбца. Эти функции извлекают метаданные, которая хранится внутри каждого столбца:

* WRITETIME хранит метку времени значения столбца.
* TTL сохраняет оставшееся время жизни (в секундах) для значения столбца, если для него установлен срок действия; в противном случае значение равно null.

The WHERE clause

WHERE clause указывает, какие строки запрашиваются. Он определяет отношение (relationship) для столбцов PRIMARY KEY или столбца, для которого определен вторичный индекс вместе с заданным значением.

Не все отношения разрешены в запросе. Например, для ключа раздела (partition key) допускается только равенство. IN clause считается равенством для одного или нескольких значений. TOKEN clause может использоваться для запроса неравенства ключей разделения. Ключ раздела должен быть указан перед столбцами кластеризации в WHERE clause. Отношение для кластеризации столбцов должно указывать непрерывный набор строк по порядку.

Например, учитывая:

|  |
| --- |
| CREATE TABLE posts (  userid text,  blog\_title text,  posted\_at timestamp,  entry\_title text,  content text,  category int,  PRIMARY KEY (userid, blog\_title, posted\_at)  ); |

Разрешен следующий запрос:

|  |
| --- |
| SELECT entry\_title, content FROM posts  WHERE userid = 'john doe'  AND blog\_title='John''s Blog'  AND posted\_at >= '2012-01-01' AND posted\_at < '2012-01-31'; |

Но следующий - нет, поскольку он не выбирает непрерывный набор строк (и мы предполагаем, что не установлены вторичные индексы):

|  |
| --- |
| // Needs a blog\_title to be set to select ranges of posted\_at  SELECT entry\_title, content FROM posts  WHERE userid = 'john doe'  AND posted\_at >= '2012-01-01' AND posted\_at < '2012-01-31'; |

При указании отношений, функцию TOKEN можно применить к столбцу PARTITION KEY для запроса. Строки будут выбираться на основе токена PARTITION\_KEY, а не значения.

|  |
| --- |
| **Важно.** |
| Маркер ключа (token of key) зависит от используемого разделителя, и, в частности, RandomPartitioner не даст значимого порядка. Также обратите внимание, что упорядочивающие разделители всегда упорядочивают значения токенов по байтам (так, даже если ключ раздела имеет тип int, в частности token(-1) > token(0)). |

Например:

|  |
| --- |
| SELECT \* FROM posts  WHERE token(userid) > token('tom') AND token(userid) < token('bob'); |

Отношение IN допускается только в последнем столбце ключа раздела или в последнем столбце полного первичного ключа.

Также возможно "сгруппировать" CLUSTERING COLUMNS вместе в отношении, используя нотацию кортежей.

Например:

|  |
| --- |
| SELECT \* FROM posts  WHERE userid = 'john doe'  AND (blog\_title, posted\_at) > ('John''s Blog', '2012-01-01'); |

Этот запрос вернет все строки, которые сортируются после строки с “John’s Blog” в качестве blog\_tile и "2012-01-01" для posted\_at в порядке кластеризации. В частности, строки, имеющие post\_at ⇐ '2012-01-01', будут возвращены, если их log\_title > 'John''s Blog'.

В данном примере это было бы не так:

|  |
| --- |
| SELECT \* FROM posts  WHERE userid = 'john doe'  AND blog\_title > 'John''s Blog'  AND posted\_at > '2012-01-01'; |

Обозначение кортежа также может использоваться для предложений IN в столбцах кластеризации:

|  |
| --- |
| SELECT \* FROM posts  WHERE userid = 'john doe'  AND (blog\_title, posted\_at) IN (('John''s Blog', '2012-01-01'), ('Extreme Chess', '2014-06-01')); |

Оператор CONTAINS может использоваться только для столбцов коллекции (списков, наборов и карт). В случае карт CONTAINS применяется к значениям карты. Оператор CONTAINS KEY может использоваться только в столбцах карты и применяется к ключам карты.

Grouping results

Параметр GROUP BY может объединить все выбранные строки, которые имеют одинаковые значения для набора столбцов, в одну строку.

Используя параметр GROUP BY, строки можно сгруппировать на уровне ключа раздела или столбца кластеризации. Следовательно, опция GROUP BY принимает только столбцы первичного ключа в определенном порядке в качестве аргументов. Если столбец первичного ключа ограничен ограничением равенства, он не включается в предложение GROUP BY.

Агрегатные функции производят отдельное значение для каждой группы. Если предложение GROUP BY не указано, агрегатные функции будут выдавать одно значение для всех строк.

Если столбец выбран без агрегатной функции, в операторе с GROUP BY будет возвращено первое обнаруженное значение в каждой группе.

Ordering results

Предложение ORDER BY выбирает порядок возвращаемых результатов. Аргументом является список имен столбцов и порядок каждого столбца (ASC для восходящего и DESC для нисходящего). Возможный порядок ограничивается порядком кластеризации, определенным в таблице:

* если таблица была определена без какого-либо конкретного CLUSTERING ORDER, то порядок определяется столбцами кластеризации или наоборот.
* в противном случае порядок определяется опцией CLUSTERING ORDER и обратным.

Limiting results

Параметр LIMIT оператора SELECT ограничивает количество строк, возвращаемых запросом. Параметр PER PARTITION LIMIT ограничивает количество строк, возвращаемых запросом для данного раздела. Оба типа ограничений могут использоваться в одном заявлении.

Allowing filtering

По умолчанию CQL разрешает только избранные запросы, которые не включают полное сканирование всех разделов. Если просканированы все разделы, то при возврате результатов может возникнуть значительная задержка, пропорциональная количеству данных в таблице. Параметр ALLOW FILTERING явно выполняет полное сканирование. Таким образом, производительность запроса может быть непредсказуемой.

Например, рассмотрим следующую таблицу профилей пользователей с указанием года рождения и страны проживания. Год рождения имеет вторичный индекс.

|  |
| --- |
| CREATE TABLE users (  username text PRIMARY KEY,  firstname text,  lastname text,  birth\_year int,  country text  );  CREATE INDEX ON users(birth\_year); |

Действительны следующие запросы:

|  |
| --- |
| // All users are returned  SELECT \* FROM users;  // All users with a particular birth year are returned  SELECT \* FROM users WHERE birth\_year = 1981; |

В обоих случаях производительность запроса пропорциональна количеству возвращаемых данных. Первый запрос возвращает все строки, поскольку выбраны все пользователи. Второй запрос возвращает только строки, определенные вторичным индексом, реализацией для каждого узла; результаты будут зависеть от количества узлов в кластере и косвенно пропорциональны количеству хранимых данных. Количество узлов всегда будет на несколько порядков меньше количества сохраненных профилей пользователей. Оба запроса могут возвращать очень большие наборы результатов, но добавление предложения LIMIT может уменьшить задержку.

Следующий запрос будет отклонен:

|  |
| --- |
| SELECT \* FROM users WHERE birth\_year = 1981 AND country = 'FR'; |

RT.KeyValue не может гарантировать, что большие объемы данных не придется сканировать, даже если результат будет небольшим. Если вы знаете, что набор данных невелик, а производительность будет приемлемой, добавьте ALLOW FILTERING, чтобы запрос мог выполняться:

|  |
| --- |
| SELECT \* FROM users WHERE birth\_year = 1981 AND country = 'FR' ALLOW FILTERING; |

INSERT

Вставка данных для строки выполняется с помощью оператора INSERT:

|  |
| --- |
| insert\_statement::= INSERT INTO table\_name ( names\_values | json\_clause )  [ IF NOT EXISTS ]  [ USING update\_parameter ( AND update\_parameter )\* ]  names\_values::= names VALUES tuple\_literal  json\_clause::= JSON string [ DEFAULT ( NULL | UNSET ) ]  names::= '(' column\_name ( ',' column\_name )\* ')' |

Например:

|  |
| --- |
| INSERT INTO NerdMovies (movie, director, main\_actor, year)  VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005)  USING TTL 86400;  INSERT INTO NerdMovies JSON '{"movie": "Serenity", "director": "Joss Whedon", "year": 2005}'; |

Оператор INSERT записывает один или несколько столбцов для данной строки в таблице. Поскольку строка идентифицируется своим PRIMARY KEY, необходимо указать хотя бы один столбец. Список вставляемых столбцов должен иметь синтаксис VALUES. При использовании синтаксиса JSON, VALUES указывать необязательно (см. раздел о поддержке JSON для получения более подробной информации). Все обновления для INSERT применяются атомарно и изолированно.

В отличие от SQL, INSERT по умолчанию не проверяет предшествующее существование строки. Строка создается, если ее раньше не было, и обновляется в противном случае. Кроме того, невозможно узнать, какое действие произошло.

Условие IF NOT EXISTS может ограничить вставку, если строка не существует. Однако обратите внимание, что использование IF NOT EXISTS повлечет за собой значительные затраты на производительность, поскольку используется Paxos, поэтому его следует использовать с осторожностью.

Пожалуйста, обратитесь к разделу UPDATE для получения информации о параметре update\_parameter. Также обратите внимание, что INSERT не поддерживает счетчики, а UPDATE поддерживает.

UPDATE

Обновление строки выполняется с помощью оператора UPDATE:

|  |
| --- |
| update\_statement ::= UPDATE table\_name  [ USING update\_parameter ( AND update\_parameter )\* ]  SET assignment( ',' assignment )\*  WHERE where\_clause  [ IF ( EXISTS | condition ( AND condition)\*) ]  update\_parameter ::= ( TIMESTAMP | TTL ) ( integer | bind\_marker )  assignment: simple\_selection'=' term  `| column\_name'=' column\_name ( '+' | '-' ) term  | column\_name'=' list\_literal'+' column\_name  simple\_selection ::= column\_name  | column\_name '[' term']'  | column\_name'.' field\_name  condition ::= `simple\_selection operator term |

Например:

|  |
| --- |
| UPDATE NerdMovies USING TTL 400  SET director = 'Joss Whedon',  main\_actor = 'Nathan Fillion',  year = 2005  WHERE movie = 'Serenity';  UPDATE UserActions  SET total = total + 2  WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14  AND action = 'click'; |

Оператор UPDATE записывает один или несколько столбцов для данной строки в таблице. Предложение WHERE используется для выбора строки для обновления и должно включать все столбцы PRIMARY KEY. Столбцы непервичного ключа устанавливаются с помощью ключевого слова SET. В операторе UPDATE все обновления в одном и том же ключе раздела применяются атомарно и изолированно.

В отличие от SQL, UPDATE по умолчанию не проверяет предшествующее существование строки. Строка создается, если ее раньше не было, и обновляется в противном случае. Кроме того, невозможно узнать, какое действие произошло.

Условие IF можно использовать, чтобы выбрать, обновляется ли строка или нет, если выполняется определенное условие. Однако, как и в случае с условием IF NOT EXISTS, могут возникнуть существенные затраты на производительность.

Относительно назначения SET:

* c = c + 3 будет увеличивать / уменьшать счетчики, единственная разрешенная операция. Имя столбца после знака '=' должно быть таким же, как имя столбца перед знаком '='. Увеличение / уменьшение разрешено только на счетчиках. См. Подробности в разделе о счетчиках (counters).
* id = id + <some-collection> и id[value1] = value2 предназначены для коллекций. См. Подробности в разделе о коллекциях (collections).
* id.field = 3 предназначен для установки значения поля в незамороженных пользовательских типах. См. Подробности в UDT.

Параметры UPDATE

Операторы UPDATE и INSERT поддерживают следующие параметры:

* TTL: указывает необязательное время жизни (Time To Live) (в секундах) для вставленных значений. Если установлено, вставленные значения автоматически удаляются из базы данных по истечении указанного времени. Обратите внимание, что TTL касается вставленных значений, а не самих столбцов. Это означает, что любое последующее обновление столбца также сбрасывает TTL (до любого значения TTL, указанного в этом обновлении). По умолчанию, значения никогда не истекают. Значение TTL равное 0 эквивалентно отсутствию TTL. Если таблица имеет значение default\_time\_to\_live, значение TTL, равное 0, удалит TTL для вставленных или обновленных значений. Значение TTL, равное null, эквивалентно вставке с TTL, равным 0.

Операторы UPDATE, INSERT, DELETE и BATCH поддерживают следующие параметры:

* TIMESTAMP: устанавливает отметку времени для операции. Если не указано иное, координатор будет использовать текущее время (в микросекундах) в начале выполнения оператора в качестве отметки времени. Обычно это подходящий вариант по умолчанию.

DELETE

Для удаления строк или частей строк используется оператор DELETE:

|  |
| --- |
| UPDATE NerdMovies USING TTL 400  SET director = 'Joss Whedon',  main\_actor = 'Nathan Fillion',  year = 2005  WHERE movie = 'Serenity';  UPDATE UserActions  SET total = total + 2  WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14  AND action = 'click'; |

Например:

|  |
| --- |
| DELETE FROM NerdMovies USING TIMESTAMP 1240003134  WHERE movie = 'Serenity';  DELETE phone FROM Users  WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D0-9908-4AE3-BE34-5573E5B09F14); |

Оператор DELETE удаляет столбцы и строки. Если имена столбцов указаны непосредственно после ключевого слова DELETE, из строки, указанной в предложении WHERE, удаляются только эти столбцы. В противном случае удаляются целые строки.

Предложение WHERE указывает, какие строки следует удалить. Несколько строк могут быть удалены одним оператором с помощью оператора IN. Диапазон строк можно удалить с помощью оператора неравенства (например, >=).

DELETE поддерживает параметр TIMESTAMP с той же семантикой, что и в обновлениях.

В операторе DELETE все удаления в одном и том же ключе раздела применяются атомарно и изолированно.

Операция DELETE может быть условной за счет использования предложения IF, аналогичного операторам UPDATE и INSERT. Однако, как и в случае с операторами INSERT и UPDATE, это повлечет за собой существенные затраты на производительность, поскольку используется Paxos, и его следует использовать с осторожностью.

BATCH

Несколько INSERT, UPDATE и DELETE могут быть выполнены в одном операторе, сгруппировав их с помощью оператора BATCH:

|  |
| --- |
| batch\_statement ::= BEGIN [ UNLOGGED | COUNTER ] BATCH  [ USING update\_parameter( AND update\_parameter)\* ]  modification\_statement ( ';' modification\_statement )\*  APPLY BATCH  modification\_statement ::= insert\_statement | update\_statement | delete\_statement |

Например:

|  |
| --- |
| BEGIN BATCH  INSERT INTO users (userid, password, name) VALUES ('user2', 'ch@ngem3b', 'second user');  UPDATE users SET password = 'ps22dhds' WHERE userid = 'user3';  INSERT INTO users (userid, password) VALUES ('user4', 'ch@ngem3c');  DELETE name FROM users WHERE userid = 'user1';  APPLY BATCH; |

Оператор BATCH группирует несколько операторов модификации (вставки/ обновления и удаления) в один оператор. Он служит нескольким целям:

* Он экономит сетевые циклы обмена между клиентом и сервером (а иногда и между координатором сервера и репликами) при пакетной обработке нескольких обновлений.
* Все обновления в BATCH, принадлежащем данному ключу раздела, выполняются изолированно.
* По умолчанию все операции в пакете выполняются как *зарегистрированные*, чтобы гарантировать, что все мутации в конечном итоге завершатся (или ни одна из них не завершится). См. UNLOGGED batches для получения более подробной информации.

Обратите внимание, что:

* Операторы BATCH могут содержать только операторы UPDATE, INSERT и DELETE (например, не другие пакеты).
* Пакеты (batches) не являются полным аналогом для транзакций SQL.
* Если метка времени не указана для каждой операции, то все операции будут применяться с одной и той же меткой времени (либо сгенерированной автоматически, либо с меткой времени, предоставленной на уровне пакета). Из-за процедуры разрешения конфликтов RT.KeyValue, в случае привязки меток времени операции могут применяться в порядке, отличном от порядка, в котором они перечислены в операторе BATCH. Чтобы принудительно упорядочить определенную операцию, вы должны указать временные метки для каждой операции.
* Пакет LOGGED для одного раздела будет преобразован в пакет UNLOGGED в качестве оптимизации.

UNLOGGED batches

По умолчанию RT.KeyValue использует журнал пакетов, чтобы гарантировать, что все операции в пакете в конечном итоге завершатся или ни одна из них не завершится (однако обратите внимание, что операции изолированы только в пределах одного раздела).

Если пакет занимает несколько разделов, это снижает производительность за атомарность пакета. Если вы не хотите подвергаться этому штрафу, вы можете сказать RT.KeyValue пропустить пакетный журнал с опцией UNLOGGED. Если используется опция UNLOGGED, неудачный пакет может оставить исправление только частично.

COUNTER batches

Используйте параметр COUNTER для пакетных обновлений счетчиков. В отличие от других обновлений в RT.KeyValue, обновления счетчиков не идемпотентны.

Арифметические операторы (Arithmetic Operators)

В **Таблице 7.** представлены поддерживаемые CQL арифметические параметры операторы.

**Таблица 7. Поддерживаемые арифметические операторы**

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| - (unary) | Отрицает операнд (Negates operand) |
| + | Добавление (Addition) |
| - | Вычитание (Subtraction) |
| \* | Умножение (Multiplication) |
| / | Деление (Division) |
| % | Возвращает остаток от деления (Returns the remaider of division) |

Числовая арифметика (Number Arithmetic)

Все арифметические операции поддерживаются с числовыми типами или счетчиками.

Тип возврата операции будет основан на типах операндов:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| left/right | tinyint | smallint | int | bigint | counter | float | double | varint | decimal |
| **tinyint** | tinyint | smallint | int | bigint | bigint | float | double | varint | decimal |
| **smallint** | smallint | smallint | int | bigint | bigint | float | double | varint | decimal |
| **int** | int | int | int | bigint | bigint | float | double | varint | decimal |
| **bigint** | bigint | bigint | bigint | bigint | bigint | double | double | varint | decimal |
| **counter** | bigint | bigint | bigint | bigint | bigint | double | double | varint | decimal |
| **float** | float | float | float | double | double | float | double | decimal | decimal |
| **double** | double | double | double | double | double | double | double | decimal | decimal |
| **varint** | varint | varint | varint | decimal | decimal | decimal | decimal | decimal | decimal |
| **decimal** | decimal | decimal | decimal | decimal | decimal | decimal | decimal | decimal | decimal |

Операторы \*, / и % имеют более высокий уровень приоритета, чем операторы + и -. Следовательно, они будут оценены раньше. Если два оператора в выражении имеют одинаковый уровень приоритета, они будут оцениваться слева направо в зависимости от их положения в выражении.

Дата-время арифметика (Datetime Arithmetic)

Продолжительность (duration) может быть добавлена (+) или вычтена (-) из отметки времени (timestamp) на дате (date), чтобы создать новую отметку времени или дату. Так, например:

|  |
| --- |
| SELECT \* FROM myTable WHERE t = '2017-01-01' - 2d; |

выберет все записи со значением t за последние 2 дня 2016 года.

Вторичные индексы (Secondary Indexes)

CQL поддерживает создание вторичных индексов для таблиц, позволяя запросам в таблице использовать эти индексы. Вторичный индекс идентифицируется именем, определяемым следующим образом:

|  |
| --- |
| index\_name::= re('[a-zA-Z\_0-9]+') |

Создать индекс (CREATE INDEX)

Для создания вторичного индекса в таблице используется оператор CREATE INDEX:

|  |
| --- |
| create\_index\_statement::= CREATE [ CUSTOM ] INDEX [ IF NOT EXISTS ] [ index\_name ]  ON table\_name '(' index\_identifier ')'  [ USING string [ WITH OPTIONS = map\_literal ] ]  index\_identifier::= column\_name  | ( KEYS | VALUES | ENTRIES | FULL ) '(' column\_name ')' |

Например:

|  |
| --- |
| CREATE INDEX userIndex ON NerdMovies (user);  CREATE INDEX ON Mutants (abilityId);  CREATE INDEX ON users (keys(favs));  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/'}; |

Оператор CREATE INDEX используется для создания нового (автоматического) вторичного индекса для данного (существующего) столбца в данной таблице. При желании перед ключевым словом ON можно указать имя самого индекса. Если данные для столбца уже существуют, они будут проиндексированы асинхронно. После создания индекса новые данные для столбца индексируются автоматически во время вставки.

Попытка создать уже существующий индекс вернет ошибку, если не используется опция IF NOT EXISTS. Если он используется, оператор не будет работать, если индекс уже существует.

Индексы на Ключах Карты (Indexes on Map Keys)

При создании индекса для карт (maps <maps>) вы можете индексировать либо ключи, либо значения. Если идентификатор столбца помещен в функцию keys(), индекс будет на ключах карты, что позволит вам использовать CONTAINS KEY в предложениях WHERE. В противном случае индекс будет на карте значений (map values).

Удаление индекса (DROP INDEX)

Для удаления вторичного индекса используется оператор DROP INDEX:

|  |
| --- |
| drop\_index\_statement::= DROP INDEX [ IF EXISTS ] index\_name |

Оператор DROP INDEX используется для удаления существующего вторичного индекса. Аргументом оператора является имя индекса, которое может дополнительно указывать пространство ключей индекса.

Если индекс не существует, оператор вернет ошибку, если только не используется IF EXISTS, и в этом случае операция не выполняется.

Материализованные представления (Materialized Views)

Имена материализованных представлений определяются:

|  |
| --- |
| view\_name::= re('[a-zA-Z\_0-9]+') |

Создать материализированное представление (CREATE MATERIALIZED VIEW)

Вы можете создать материализованное представление для таблицы с помощью оператора CREATE MATERIALIZED VIEW:

|  |
| --- |
| create\_materialized\_view\_statement::= CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] view\_name  AS select\_statement  PRIMARY KEY '(' primary\_key')'  WITH table\_options |

Например:

|  |
| --- |
| CREATE MATERIALIZED VIEW monkeySpecies\_by\_population AS  SELECT \* FROM monkeySpecies  WHERE population IS NOT NULL AND species IS NOT NULL  PRIMARY KEY (population, species)  WITH comment='Allow query by population instead of species'; |

Оператор CREATE MATERIALIZED VIEW создает новое материализованное представление. Каждое такое представление представляет собой набор строк, соответствующих строкам, которые присутствуют в базе или базовой таблице, указанной в операторе SELECT. Материализованное представление не может быть обновлено напрямую, но обновления базовой таблицы вызовут соответствующие обновления в представлении.

Создание материализованного представления состоит из 3 основных частей:

* Оператор выбора (select), ограничивающий данные, включенные в представление.
* Определение первичного ключа (primary kay) для представления.
* Параметры представления (view).

Попытка создать уже существующее материализованное представление вернет ошибку, если не используется опция IF NOT EXISTS. Если он используется, оператор не будет работать, если материализованное представление уже существует.

|  |
| --- |
| **Примечание.** |
| По умолчанию материализованные представления строятся в одном потоке. Первоначальную сборку можно распараллелить, увеличив количество потоков, заданных свойством concurrent\_materialized\_view\_builders в cassandra.yaml. Этим свойством также можно управлять во время выполнения, как с помощью команд JMX, так и setconcurrentviewbuilders и getconcurrentviewbuilders nodetool. |

Оператор выбора материализированного представления (MV select statement

Оператор select при создании материализованного представления определяет, какая из базовых таблиц включается в представление. Это утверждение ограничено несколькими способами:

1. выбор (selection) ограничен теми, которые выбирают только столбцы базовой таблицы. Другими словами, вы не можете использовать какие-либо функции (агрегированные или нет), приведение, термин и т.д. Алиасы также не поддерживаются. Однако вы можете использовать \* как ярлык для выбора всех столбцов. Кроме того, статические столбцы (static columns) нельзя включить в материализованное представление. Таким образом, команда SELECT \* недопустима, если в базовой таблице есть статические столбцы. Предложение WHERE имеет следующие ограничения:

* не может включать какой-либо bind\_marker
* не может иметь столбцов, которые не являются частью первичного ключа базовой таблицы, на которые не распространяется ограничение IS NOT NULL
* никакие другие ограничения не допускаются
* столбцы, которые являются частью первичного ключа представления, не могут иметь значение NULL, они всегда должны быть, по крайней мере, ограничены ограничением IS NOT NULL (или любым другим ограничением, но оно должно быть у них).

1. не может иметь условия упорядочивания, ограничения или внешней ссылки: cql / dml.adoc # allow-filtering[ALLOW FILTERING

Первичный ключ материализированного представления (MV primary key)

Представление должно иметь первичный ключ, и этот первичный ключ должен соответствовать следующим ограничениям:

* он должен содержать все столбцы первичного ключа базовой таблицы. Это гарантирует, что каждая строка представления соответствует ровно одной строке базовой таблицы.
* он может содержать только один столбец, который не является столбцом первичного ключа в базовой таблице.

Так, например, дайте следующее определение базовой таблицы:

|  |
| --- |
| CREATE TABLE t (  k int,  c1 int,  c2 int,  v1 int,  v2 int,  PRIMARY KEY (k, c1, c2)  ); |

то разрешены следующие определения представлений:

|  |
| --- |
| CREATE MATERIALIZED VIEW mv1 AS  SELECT \* FROM t  WHERE k IS NOT NULL AND c1 IS NOT NULL AND c2 IS NOT NULL  PRIMARY KEY (c1, k, c2);  CREATE MATERIALIZED VIEW mv1 AS  SELECT \* FROM t  WHERE k IS NOT NULL AND c1 IS NOT NULL AND c2 IS NOT NULL  PRIMARY KEY (v1, k, c1, c2); |

но недопустимы следующие:

|  |
| --- |
| // Error: cannot include both v1 and v2 in the primary key as both are not in the base table primary key  CREATE MATERIALIZED VIEW mv1 AS  SELECT \* FROM t  WHERE k IS NOT NULL AND c1 IS NOT NULL AND c2 IS NOT NULL AND v1 IS NOT NULL  PRIMARY KEY (v1, v2, k, c1, c2);  // Error: must include k in the primary as it's a base table primary key column  CREATE MATERIALIZED VIEW mv1 AS  SELECT \* FROM t  WHERE c1 IS NOT NULL AND c2 IS NOT NULL  PRIMARY KEY (c1, c2); |

Параметры материализированного представления (MV options)

Материализованное представление внутренне реализуется таблицей, и поэтому создание MV позволяет использовать те же параметры, что и создание таблицы (same options than creating a table <create-table-options>)

ALTER MATERIALIZED VIEW

После создания вы можете изменить параметры материализованного представления с помощью оператора ALTER MATERIALIZED VIEW:

|  |
| --- |
| alter\_materialized\_view\_statement::= ALTER MATERIALIZED VIEW view\_name WITH table\_options |

Параметры, которые можно обновить, такие же, как во время создания, и, следовательно, такие же, как для таблиц (same than for tables <create-table-options>).

DROP MATERIALIZED VIEW

Удаление материализованного представления с помощью оператора DROP MATERIALIZED VIEW:

|  |
| --- |
| drop\_materialized\_view\_statement::= DROP MATERIALIZED VIEW [ IF EXISTS ] view\_name; |

Если материализованное представление не существует, оператор вернет ошибку, если только не используется IF EXISTS, и в этом случае операция не выполняется.

Ограничения MV (MV Limitations)

|  |
| --- |
| **Примечание.** |
| Удаление столбцов, не выбранных в материализованном представлении (с помощью UPDATE base SET unselected\_column = null или DELETE unselected\_column FROM base), может затенять пропущенные обновления для других столбцов, полученных с помощью подсказок (hints) или исправлений (repiar). |

Функции (Functions)

CQL поддерживает 2 основные категории функций:

* скалярные функции (scalar functions), которые принимают ряд значений и производят вывод.
* агрегатные функции (aggregate functions), которые агрегируют несколько строк в результате оператора SELECT.

В обоих случаях CQL предоставляет ряд встроенных "жестко запрограммированных" функций, а также возможность создавать новые определяемые пользователем функции.

|  |
| --- |
| **Примечание.** |
| По умолчанию, использование пользовательских функций отключено по умолчанию из соображений безопасности (даже если оно включено, выполнение пользовательских функций изолировано, и "мошеннической" ("rogue") функции нельзя позволять делать зло, но идеальной песочницы нет, поэтому использование определяемых пользователем функций является подпиской). См. enable\_user\_defined\_functions в cassandra.yaml, чтобы включить их. |

Функция идентифицируется по имени:

|  |
| --- |
| function\_name ::= [ keyspace\_name'.' ] name |

Scalar - функции (functions)

Native - функции (functions)

Cast

Функция приведения (cast) может использоваться для преобразования одного собственного типа данных в другой.

В следующей **Таблице 8.** описаны преобразования, поддерживаемые функцией приведения (cast). RT.KeyValue будет молча игнорировать любое приведение, преобразующее тип данных в его собственный тип данных.

**Таблица 8. Преобразования, поддерживаемые функцией приведения (cast)**

|  |  |
| --- | --- |
| **From** | **To** |
| ascii | text, varchar |
| bigint | tinyint, smallint, int, float, double, decimal, varint, text,  varchar |
| boolean | text, varchar |
| counter | tinyint, smallint, int, bigint, float, double, decimal, varint,  text, varchar |
| date | timestamp |
| decimal | tinyint, smallint, int, bigint, float, double, varint, text,  varchar |
| double | tinyint, smallint, int, bigint, float, decimal, varint, text,  varchar |
| float | tinyint, smallint, int, bigint, double, decimal, varint, text,  varchar |
| inet | text, varchar |
| int | tinyint, smallint, bigint, float, double, decimal, varint, text,  varchar |
| smallint | tinyint, int, bigint, float, double, decimal, varint, text,  varchar |
| time | text, varchar |
| timestamp | date, text, varchar |
| timeuuid | timestamp, date, text, varchar |
| tinyint | tinyint, smallint, int, bigint, float, double, decimal, varint, text,  varchar |
| uuid | text, varchar |
| varint | tinyint, smallint, int, bigint, float, double, decimal, text,  varchar |

Преобразования строго зависят от семантики Java. Например, двойное значение 1 будет преобразовано в текстовое значение "1.0".

Например:

|  |
| --- |
| SELECT avg(cast(count as double)) FROM myTable |

Token

Функция token вычисляет токен для заданного ключа раздела. Точная подпись функции токена зависит от рассматриваемой таблицы и модуля разделения, используемого кластером.

Тип аргументов токена зависит от типа столбца ключа раздела. Возвращаемый тип зависит от определенного разделителя:

|  |  |
| --- | --- |
| **Разделитель (Partitioner)** | **Возвращаемый тип (Returned type)** |
| Murmur3Partitioner | bigint |
| RandomPartitioner | varint |
| ByteOrderedPartitioner | blob |

Например, рассмотрим следующую таблицу:

|  |
| --- |
| CREATE TABLE users (  userid text PRIMARY KEY,  username text,  ); |

В таблице используется Murmur3Partitioner по умолчанию. Функция token использует текст (text) с одним аргументом, потому что ключ раздела - это ИД пользователя (userid) текстового типа. Возвращаемый тип будет bigint.

Uuid

Функция uuid не принимает параметров и генерирует uuid случайного типа 4 (random type 4), подходящий для использования в операторах INSERT или UPDATE.

Timeuuid - функции

now

Функция uuid не принимает параметров и генерирует uuid случайного типа 4 (random type 4), подходящий для использования в операторах INSERT или UPDATE.

Функция now не принимает аргументов и генерирует на узле координатора новый уникальный timeuuid во время вызова функции. Обратите внимание, что этот метод полезен для вставки, но в основном бессмысленен в предложениях WHERE.

Например, запрос формы:

|  |
| --- |
| SELECT \* FROM myTable WHERE t = now(); |

не будет возвращать результат по умолчанию, поскольку значение, возвращаемое функцией now(), гарантированно будет уникальным.

currentTimeUUI - это алиас now.

minTimeuuid и maxTimeuuid

Функция minTimeuuid принимает значение timestamp t, либо метку времени, либо строку даты. Он возвращает поддельный (*fake*) timeuuid, соответствующий *наименьшему* возможному timeuuid для отметки времени t. maxTimeuuid работает аналогично, но возвращает *максимально* возможное значение timeuuid.

Например:

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

выберет все строки, в которых столбец timeuuid t находится позже '2013-01-01 00:05+0000' и раньше '2013-02-02 10:00+0000'. Предложение t >= maxTimeuuid('2013-01-01 00:05+0000') все равно не выберет timeuuid, созданный точно в '2013-01-01 00: 05 + 0000', и по существу эквивалентен t > maxTimeuuid('2013-01-01 00:05+0000').

|  |
| --- |
| **Примечание.** |
| Значения, генерируемые minTimeuuid и maxTimeuuid, называются поддельными (*fake*) UUID, поскольку они не учитывают процесс генерации UUID на основе времени, указанный в IETF RFC 4122. В частности, значение, возвращаемое этими двумя методами, не будет уникальным. Таким образом, используйте эти методы только для запросов, а не для вставки, чтобы предотвратить возможную перезапись данных. |

Datetime - функции

Получение текущей даты / времени

Следующие функции, представленные в **Таблице 9.**, могут использоваться для получения даты / времени в момент вызова функции.

**Таблица 9. Функции, используемые для получения даты / времени в момент вызова функции.**

|  |  |
| --- | --- |
| **Название функции** | **Выходной тип** |
| currentTimestamp | timestamp |
| currentDate | date |
| currentTime | time |
| currentTimeUUID | timeUUID |

Например, данные за последние два дня можно получить, используя:

|  |
| --- |
| SELECT \* FROM myTable WHERE date >= currentDate() - 2d; |

Функции преобразования времени (Time conversion functions)

В Таблице 10. предоставляется ряд функций для преобразования timeuuid, timestamp или date в другой собственный (native) тип.

**Таблица 10. Функции для преобразования** timeuuid**,** timestamp**,** date **в другой собственный тип.**

|  |  |  |
| --- | --- | --- |
| **Название функции** | **Входной тип** | **Описание** |
| toDate | timeuuid | Преобразует аргумент timeuuid в тип date |
| toDate | timestamp | Преобразует аргумент timestamp в тип date |
| toTimestamp | timeuuid | Преобразует аргумент timeuuid в тип timestamp |
| toTimestamp | date | Преобразует аргумент date в тип timestamp |
| toUnixTimestamp | timeuuid | Преобразует аргумент timeuuid в исходное значение bigInt |
| toUnixTimestamp | timestamp | Преобразует аргумент timestamp в исходное значение bigInt |
| toUnixTimestamp | date | Преобразует аргумент date в исходное значение bigInt |
| dateOf | timeuuid | Аналогично toTimestamp(timeuuid) (УСТАРЕЛО) |
| unixTimestampOf | timeuuid | Аналогично toUnixTimestamp(timeuuid) (УСТАРЕЛО) |

Функции преобразования BLOB-объектов (Blob conversion functions)

Предоставляется ряд функций для преобразования собственных типов в двоичные данные или большой двоичный объект (blob). Для каждого типа, поддерживаемого CQL, функция typeAsBlob принимает аргумент типа type и возвращает его как blob. И наоборот, функция blobAsType принимает 64-битный аргумент blob и преобразует его в значение bigint. Например, bigintAsBlob(3) возвращает 0x0000000000000003, а blobAsBigint(0x0000000000000003) возвращает 3.

Пользовательские функции (User-defined functions)

Пользовательские функции (UDFs) выполняют предоставленный пользователем код в RT.KeyValue. По умолчанию RT.KeyValue поддерживает определение функций на Java и JavaScript. Поддержка других языков сценариев, совместимых с JSR 223, таких как Python, Ruby и Scala, возможна путем добавления JAR в путь к классам.

UDF являются частью схемы RT.KeyValue и автоматически распространяются на все узлы в кластере. UDF могут быть *перегружены*, поэтому несколько UDF с разными типами аргументов могут иметь одно и то же имя функции.

Например:

|  |
| --- |
| CREATE FUNCTION sample ( arg int ) ...;  CREATE FUNCTION sample ( arg text ) ...; |

UDF подвержены всем обычным проблемам с выбранным языком программирования. Соответственно, реализации должны быть защищены от исключений нулевого указателя, недопустимых аргументов или любого другого потенциального источника исключений. Исключение во время выполнения функции приведет к сбою всего оператора. Допустимые запросы для использования UDF - это операторы SELECT, INSERT и UPDATE.

Сложные типы (complex types), такие как коллекции (collections), типы кортежей (tuple types) и определяемые пользователем типы (user-defined types), являются допустимыми типами аргументов и возвращаемых типов в UDF. Типы кортежей и определяемые пользователем типы используют функции преобразования драйвера Java DataStax.

Аргументы для функций могут быть литералами или терминами. Также можно использовать подготовленные заполнители (placeholders) для операторов.

Обратите внимание на использование синтаксиса с двойным знаком доллара для включения исходного кода UDF.

Например:

|  |
| --- |
| CREATE FUNCTION some\_function ( arg int )  RETURNS NULL ON NULL INPUT  RETURNS int  LANGUAGE java  AS $$ return arg; $$;  SELECT some\_function(column) FROM atable ...;  UPDATE atable SET col = some\_function(?) ...;  CREATE TYPE custom\_type (txt text, i int);  CREATE FUNCTION fct\_using\_udt ( udtarg frozen )  RETURNS NULL ON NULL INPUT  RETURNS text  LANGUAGE java  AS $$ return udtarg.getString("txt"); $$; |

Неявно доступное поле UDFContext (или привязка для UDF скриптов) предоставляет необходимые функции для создания новых значений UDT и кортежа:

|  |
| --- |
| CREATE TYPE custom\_type (txt text, i int);  CREATE FUNCTION fct\\_using\\_udt ( somearg int )  RETURNS NULL ON NULL INPUT  RETURNS custom\_type  LANGUAGE java  AS $$  UDTValue udt = udfContext.newReturnUDTValue();  udt.setString("txt", "some string");  udt.setInt("i", 42);  return udt;  $$; |

Определение интерфейса UDFContext можно найти в исходном коде RT.KeyValue для org.apache.cassandra.cql3.functions.UDFContext.

|  |
| --- |
| Cpublic interface UDFContext  {  UDTValue newArgUDTValue(String argName);  UDTValue newArgUDTValue(int argNum);  UDTValue newReturnUDTValue();  UDTValue newUDTValue(String udtName);  TupleValue newArgTupleValue(String argName);  TupleValue newArgTupleValue(int argNum);  TupleValue newReturnTupleValue();  TupleValue newTupleValue(String cqlDefinition);  } |

В UDF-функциях Java уже определен импорт для общих интерфейсов и классов. Этот импорт:

|  |
| --- |
| import java.nio.ByteBuffer;  import java.util.List;  import java.util.Map;  import java.util.Set;  import org.apache.cassandra.cql3.functions.UDFContext;  import com.datastax.driver.core.TypeCodec;  import com.datastax.driver.core.TupleValue;  import com.datastax.driver.core.UDTValue; |

Обратите внимание, что этот удобный импорт недоступен для сценариев UDF.

Оператор CREATE FUNCTION

Для создания новой пользовательской функции используется оператор CREATE FUNCTION:

|  |
| --- |
| create\_function\_statement::= CREATE [ OR REPLACE ] FUNCTION [ IF NOT EXISTS]  function\_name '(' arguments\_declaration ')'  [ CALLED | RETURNS NULL ] ON NULL INPUT  RETURNS cql\_type  LANGUAGE identifier  AS string arguments\_declaration: identifier cql\_type ( ',' identifier cql\_type )\* |

Например:

|  |
| --- |
| CREATE OR REPLACE FUNCTION somefunction(somearg int, anotherarg text, complexarg frozen<someUDT>, listarg list)  RETURNS NULL ON NULL INPUT  RETURNS text  LANGUAGE java  AS $$  // some Java code  $$;  CREATE FUNCTION IF NOT EXISTS akeyspace.fname(someArg int)  CALLED ON NULL INPUT  RETURNS text  LANGUAGE java  AS $$  // some Java code  $$; |

CREATE FUNCTION с необязательными ключевыми словами OR REPLACE создает либо функцию, либо заменяет существующую с такой же подписью. CREATE FUNCTION без OR REPLACE завершается ошибкой, если функция с такой же сигнатурой уже существует. Если используются необязательные ключевые слова IF NOT EXISTS, функция будет создана только в том случае, если другой функции с такой же сигнатурой не существует. OR REPLACE и IF NOT EXISTS не могут использоваться вместе.

Поведение для null входных значений должно быть определено для каждой функции:

* RETURNS NULL ON NULL INPUT объявляет, что функция всегда будет возвращать значение null, если какой-либо из входных аргументов имеет значение null.
* CALLED ON NULL INPUT объявляет, что функция будет выполняться всегда.

Function Signature

Сигнатуры(signatures) используются для различения отдельных функций. Сигнатура состоит из полного имени функции <keyspace>. <function\_name> и объединенного списка всех типов аргументов.

Обратите внимание, что имена пространств ключей, имена функций и типы аргументов подчиняются соглашениям об именах по умолчанию и правилам чувствительности к регистру.

Функции принадлежат пространству ключей; если пространство ключей не указано, используется текущее пространство ключей. Пользовательские функции не допускаются в системных пространствах ключей.

Оператор CREATE FUNCTION

Для удаления функции используется оператор DROP FUNCTION:

|  |
| --- |
| drop\_function\_statement::= DROP FUNCTION [ IF EXISTS ] function\_name [ '(' arguments\_signature ')' ]  arguments\_signature::= cql\_type ( ',' cql\_type )\* |

Например:

|  |
| --- |
| DROP FUNCTION myfunction;  DROP FUNCTION mykeyspace.afunction;  DROP FUNCTION afunction ( int );  DROP FUNCTION afunction ( text ); |

Вы должны указать типы аргументов функции, arguments\_signature, в команде drop, если есть несколько перегруженных функций с тем же именем, но разными подписями. DROP FUNCTION с необязательными ключевыми словами IF EXISTS удаляет функцию, если она существует, но не вызывает ошибку, если ее нет.

Aggregate - функции (functions)

Агрегатные функции работают с набором строк. Значения для каждой строки вводятся, чтобы вернуть одно значение для набора агрегированных строк.

Если обычные (normal) столбцы, скалярные функции (scalar functions), поля UDT, writetime или ttl выбраны вместе с агрегатными функциями, возвращаемые для них значения будут значениями первой строки, соответствующей запросу.

Native - aggregates

Count

Функция count может использоваться для подсчета строк, возвращаемых запросом.

|  |
| --- |
| SELECT COUNT (\*) FROM plays;  SELECT COUNT (1) FROM plays; |

count также может подсчитывать ненулевые значения данного столбца:

|  |
| --- |
| SELECT COUNT (scores) FROM plays; |

Max и Min

Функции max и min вычисляют максимальное и минимальное значение, возвращаемое запросом для данного столбца.

Например:

|  |
| --- |
| SELECT MIN (players), MAX (players) FROM plays WHERE game = 'quake'; |

Sum

Функция sum суммирует все значения, возвращаемые запросом для данного столбца.

Например:

|  |
| --- |
| SELECT SUM (players) FROM plays; |

Avg

Функция avg вычисляет среднее всех значений, возвращаемых запросом для данного столбца.

Например:

|  |
| --- |
| SELECT AVG (players) FROM plays; |

Определяемые пользователем агрегаты (User-Defined Aggregates (UDAs))

UDAs позволяют создавать собственные агрегатные функции. В операторе SELECT можно использовать определенные пользователем агрегаты.

Каждому агрегату требуется *начальное* состояние типа STYPE, определенное с помощью значения INITCOND`value (default value: `null). Первый аргумент функции состояния должен иметь тип STYPE. Остальные аргументы функции состояния должны соответствовать типам определяемых пользователем агрегированных аргументов. Функция состояния вызывается один раз для каждой строки, и значение, возвращаемое функцией состояния, становится новым состоянием. После обработки всех строк необязательная функция FINALFUNC выполняется со значением последнего состояния в качестве аргумента.

Значение STYPE является обязательным, чтобы различать возможно перегруженные версии состояния и/ или конечной функции, поскольку перегрузка может появиться после создания агрегата.

Полный рабочий пример для определяемых пользователем агрегатов (при условии, что пространство ключей было выбрано с помощью оператора USE):

|  |
| --- |
| CREATE OR REPLACE FUNCTION test.averageState(state tuple<int,bigint>, val int)  CALLED ON NULL INPUT  RETURNS tuple  LANGUAGE java  AS $$  if (val != null) {  state.setInt(0, state.getInt(0)+1);  state.setLong(1, state.getLong(1)+val.intValue());  }  return state;  $$;  CREATE OR REPLACE FUNCTION test.averageFinal (state tuple<int,bigint>)  CALLED ON NULL INPUT  RETURNS double  LANGUAGE java  AS $$  double r = 0;  if (state.getInt(0) == 0) return null;  r = state.getLong(1);  r /= state.getInt(0);  return Double.valueOf(r);  $$;  CREATE OR REPLACE AGGREGATE test.average(int)  SFUNC averageState  STYPE tuple  FINALFUNC averageFinal  INITCOND (0, 0);  CREATE TABLE test.atable (  pk int PRIMARY KEY,  val int  );  INSERT INTO test.atable (pk, val) VALUES (1,1);  INSERT INTO test.atable (pk, val) VALUES (2,2);  INSERT INTO test.atable (pk, val) VALUES (3,3);  INSERT INTO test.atable (pk, val) VALUES (4,4);  SELECT test.average(val) FROM atable; |

Оператор CREATE AGGREGATE

Для создания (или замены) пользовательской агрегатной функции используется оператор CREATE AGGREGATE:

|  |
| --- |
| create\_aggregate\_statement ::= CREATE [ OR REPLACE ] AGGREGATE [ IF NOT EXISTS ]  function\_name '(' arguments\_signature')'  SFUNC function\_name  STYPE cql\_type:  [ FINALFUNC function\_name]  [ INITCOND term ] |

Команда CREATE AGGREGATE с необязательными ключевыми словами OR REPLACE создает либо агрегат, либо заменяет существующий с такой же подписью. CREATE AGGREGATE без OR REPLACE завершается ошибкой, если агрегат с такой же подписью уже существует. Команда CREATE AGGREGATE с необязательными ключевыми словами IF NOT EXISTS создает агрегат, если он еще не существует. Фразы OR REPLACE и IF NOT EXISTS нельзя использовать вместе.

Значение STYPE определяет тип значения состояния и должно быть указано. Необязательный INITCOND определяет значение начального состояния для агрегата; значение по умолчанию - null. Ненулевой INITCOND должен быть указан для функций состояния, которые объявлены с RETURNS NULL ON NULL INPUT.

Значение SFUNC ссылается на существующую функцию для использования в качестве функции изменения состояния. Первый аргумент функции состояния должен иметь тип STYPE. Остальные аргументы функции состояния должны соответствовать типам определяемых пользователем агрегированных аргументов. Функция состояния вызывается один раз для каждой строки, и значение, возвращаемое функцией состояния, становится новым состоянием. Состояние не обновляется для функций состояния, объявленных с помощью RETURNS NULL ON NULL INPUT и вызванных с помощью null. После обработки всех строк необязательная функция FINALFUNC выполняется со значением последнего состояния в качестве аргумента. Он должен принимать только один аргумент с типом STYPE, но тип возвращаемого значения FINALFUNC может быть другого типа. Последняя функция, объявленная с RETURNS NULL ON NULL INPUT, означает, что возвращаемое значение агрегата будет null, если последнее состояние равно null.

Если FINALFUNC не определен, общий тип возвращаемого значения агрегатной функции - STYPE. Если FINALFUNC определен, это тип возвращаемого значения этой функции.

Оператор DROP AGGREGATE

Для удаления определяемой пользователем агрегатной функции используется оператор DROP AGGREGATE:

|  |
| --- |
| drop\_aggregate\_statement::= DROP AGGREGATE [ IF EXISTS ] function\_name[ '(' arguments\_signature ')' ] |

Например:

|  |
| --- |
| DROP AGGREGATE myAggregate;  DROP AGGREGATE myKeyspace.anAggregate;  DROP AGGREGATE someAggregate ( int );  DROP AGGREGATE someAggregate ( text ); |

Оператор DROP AGGREGATE удаляет агрегат, созданный с помощью CREATE CREATE AGGREGATE. Вы должны указать типы аргументов агрегата для удаления, если имеется несколько перегруженных агрегатов с тем же именем, но с другой подписью.

Команда DROP AGGREGATE с необязательными ключевыми словами IF EXISTS удаляет агрегат, если он существует, и ничего не делает, если функция с подписью не существует.

Поддержка JSON

RT.KeyValue представляет поддержку (support) JSON для операторов SELECT <select-statement> и INSERT <insert-statement>. Эта поддержка принципиально не меняет CQL API (например, схема все еще применяется). Он просто предоставляет удобный способ работы с документами JSON.

SELECT JSON

В операторах SELECT ключевое слово JSON используется для возврата каждой строки в виде единой карты в кодировке JSON. Остальная часть поведения оператора SELECT такая же.

Ключи карты результатов (result map keys) соответствуют именам столбцов в нормальном наборе результатов. Например, такой оператор, как SELECT JSON a, ttl(b) FROM …, приведет к карте с ключами "a" и "ttl(b)". Однако есть одно заметное исключение: для симметрии с поведением INSERT JSON, чувствительные к регистру имена столбцов с прописными буквами будут заключены в двойные кавычки. Например, SELECT JSON myColumn FROM …​ приведет к ключу карты "\"myColumn\"" с экранированными кавычками.

Значения карты будут представлениями в кодировке JSON (как описано ниже) значений набора результатов.

INSERT JSON

С помощью операторов INSERT можно использовать новое ключевое слово JSON для включения вставки карты в кодировке JSON в виде одной строки. Формат карты JSON обычно должен соответствовать формату, возвращаемому оператором SELECT JSON в той же таблице. В частности, имена столбцов с учетом регистра следует заключать в двойные кавычки. Например, чтобы вставить в таблицу с двумя столбцами с именами "myKey" и "значение" ("value"), вы должны сделать следующее:

|  |
| --- |
| INSERT INTO mytable JSON '{ "\"myKey\"": 0, "value": 0}'; |

По умолчанию (или если DEFAULT NULL явно используется), столбцу, отсутствующему на карте JSON, будет установлено значение NULL, что означает, что любое ранее существовавшее значение для этого столбца будет удалено (что приведет к созданию tombstone). В качестве альтернативы, если после значения используется директива DEFAULT UNSET, пропущенные значения столбца будут оставлены неустановленными, что означает, что ранее существовавшие значения для этого столбца будут сохранены.

Кодирование JSON типов данных RT.KeyValue (JSON Encoding of RT.KeyValue Data Types)

По возможности, RT.KeyValue будет представлять и принимать типы данных в их собственном представлении JSON. RT.KeyValue также будет принимать строковые представления, соответствующие литеральному формату CQL для всех типов с одним полем. Например, числа с плавающей запятой (floats), целые числа (ints), UUID и даты могут быть представлены строковыми литералами CQL. Однако составные типы, такие как коллекции, кортежи и определяемые пользователем типы, должны быть представлены собственными коллекциями JSON (карты и списки) или строковым представлением коллекции в кодировке JSON.

В следующей **Таблице 11.** описаны кодировки, которые RT.KeyValue примет в значениях INSERT JSON (и аргументы fromJson()), а также формат, который RT.KeyValue будет использовать при возврате данных для операторов SELECT JSON (и fromJson()).

**Таблица 11. Кодировки и форматы**

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Принятые форматы** | **Формат возврата** | **Примечание** |
| ascii | string | string | Использует escape-символ JSON \u |
| bigint | integer, string | integer | Строка должна быть допустимым 64-битным целым числом |
| blob | string | string | Строка должна быть 0x, за которой следует четное количество шестнадцатеричных цифр |
| boolean | boolean, string | boolean | Строка должна быть "true" или "false" |
| date | string | string | Дата в формате YYYY-MM-DD, часовой пояс UTC |
| decimal | integer, float, string | float | Может превышать 32- или 64-битную точность с плавающей запятой IEEE-754 в клиентском декодере. |
| double | integer, float, string | float | Строка должна быть действительным целым числом () или числом с плавающей запятой. |
| float | integer, float, string | float | Строка должна быть действительным целым числом или числом с плавающей запятой. |
| inet | string | string | IPv4 или IPv6 адрес |
| int | integer, string | integer | Строка должна быть действительным 32-битным целым числом |
| list | list, string | list | Использует собственное представление списка JSON |
| map | map, string | map | Использует собственное представление карты JSON |
| smallint | integer, string | integer | Строка должна быть допустимым 16-битным целым числом |
| set | list, string | list | Использует собственное представление списка JSON |
| text | string | string | Использует escape-символ JSON \u |
| time | string | string | Текущее время в формате HH-MM-SS[.fffffffff] |
| timestamp | integer, string | string | Отметка времени. Константа Strings позволяет вводить timestamps as dates <timestamps>. Возвращаются отметки даты в формате YYYY-MM-DD HH:MM:SS.SSS |
| timeuuid | string | string | UUID типа 1. См. раздел constant для формата UUID |
| tinyint | integer, string | integer | Строка должна быть допустимым 8-битным целым числом |
| tuple | list, string | list | Использует собственное представление списка JSON |
| UDT | map, string | map | Использует собственное представление карты JSON с именами полей в качестве ключей |
| uuid | string | string | См. constant для формата UUID |
| varchar | string | string | Использует escape-символ JSON \u |
| varint | integer, string | integer | Переменная длина; может переполнять 32- или 64-битные целые числа в декодере на стороне клиента |

Функция fromJson()

Функцию fromJson() можно использовать аналогично INSERT JSON, но для одного значения столбца. Его можно использовать только в предложении VALUES оператора INSERT или как одно из значений столбца в операторе UPDATE, DELETE или SELEC. Например, его нельзя использовать в предложении выбора оператора SELECT.

Функция toJson()

Функцию toJson() можно использовать аналогично SELECT JSON, но для одного значения столбца. Его можно использовать только в предложении выбора оператора SELECT.

Безопасность (Security)

Роли базы данных (Database Roles)

CQL использует роли базы данных для представления пользователей и группы пользователей. Синтаксически роль определяется:

|  |
| --- |
| role\_name ::= identifier | string |

CREATE ROLE

Создание роли использует оператор CREATE ROLE:

|  |
| --- |
| create\_role\_statement ::= CREATE ROLE [ IF NOT EXISTS ] role\_name  [ WITH role\_options# ]  role\_options ::= role\_option ( AND role\_option)\*  role\_option ::= PASSWORD '=' string  | LOGIN '=' boolean  | SUPERUSER '=' boolean  | OPTIONS '=' map\_literal  | ACCESS TO DATACENTERS set\_literal  | ACCESS TO ALL DATACENTERS |

Например:

|  |
| --- |
| CREATE ROLE new\_role;  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true;  CREATE ROLE bob WITH PASSWORD = 'password\_b' AND LOGIN = true AND SUPERUSER = true;  CREATE ROLE carlos WITH OPTIONS = { 'custom\_option1' : 'option1\_value', 'custom\_option2' : 99 };  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true AND ACCESS TO DATACENTERS {'DC1', 'DC3'};  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true AND ACCESS TO ALL DATACENTERS; |

По умолчанию роли не обладают привилегиями LOGIN или статусом SUPERUSER.

Разрешения (permissions) на ресурсы базы данных предоставляются ролям; типы ресурсов включают пространства ключей, таблицы, функции и сами роли. Роли могут быть предоставлены другим ролям для создания иерархических структур разрешений; в этих иерархиях разрешения и статус SUPERUSER наследуются, а привилегия LOGIN - нет.

Если у роли есть привилегия LOGIN, клиенты могут идентифицировать эту роль при подключении. На время этого соединения клиент получит все роли и привилегии, предоставленные этой роли.

Только клиент с разрешением CREATE для ресурса ролей базы данных может выдавать запросы CREATE ROLE, если только клиент не является SUPERUSER. Управление ролями в RT.KeyValue является подключаемым, и пользовательские реализации могут поддерживать только подмножество перечисленных опций.

Имена ролей следует заключать в кавычки, если они содержат не буквенно-цифровые символы.

Установка учетных данных для внутренней аутентификации

Используйте предложение WITH PASSWORD, чтобы установить пароль для внутренней аутентификации, заключив пароль в одинарные кавычки.

Если внутренняя аутентификация не была настроена или роль не имеет привилегий LOGIN, предложение WITH PASSWORD не требуется.

Ограничение подключений к определенным центрам обработки данных

Если network\_authorizer настроен, вы можете ограничить роли входа в систему определенными центрами обработки данных с помощью предложения ACCESS TO DATACENTERS, за которым следует набор литералов центров обработки данных, к которым пользователь может получить доступ. Отсутствие указания центров обработки данных неявно предоставляет доступ ко всем центрам обработки данных. Предложение ACCESS TO ALL DATACENTERS может использоваться для ясности, но функциональной разницы нет.

Условное создание роли

Попытка создать существующую роль приводит к недопустимому условию запроса, если не используется опция IF NOT EXISTS. Если опция используется и роль существует, оператор не работает:

|  |
| --- |
| CREATE ROLE other\_role;  CREATE ROLE IF NOT EXISTS other\_role; |

ALTER ROLE

Для изменения параметров роли используется оператор ALTER ROLE:

|  |
| --- |
| alter\_role\_statement ::= ALTER ROLE role\_name WITH role\_options |

Например:

|  |
| --- |
| ALTER ROLE bob WITH PASSWORD = 'PASSWORD\_B' AND SUPERUSER = false; |

Ограничение подключений к определенным центрам обработки данных

Если network\_authorizer настроен, вы можете ограничить роли входа в систему определенными центрами обработки данных с помощью предложения ACCESS TO DATACENTERS, за которым следует набор литералов центров обработки данных, к которым пользователь может получить доступ. Чтобы снять любые ограничения центра обработки данных, используйте предложение ACCESS TO ALL DATACENTERS.

Условия выполнения операторов ALTER ROLE:

* клиент должен иметь статус SUPERUSER, чтобы изменить статус SUPERUSER другой роли.
* клиент не может изменить статус SUPERUSER ни одной из ролей, которые он в настоящее время занимает.
* клиент может изменять только определенные свойства роли, которую он идентифицировал при входе в систему (например, PASSWORD).
* для изменения свойств роли, клиенту должно быть предоставлено разрешение ALTER permission <cql-permissions> для этой роли.

DROP ROLE

Для удаления роли используется оператор DROP ROLE:

|  |
| --- |
| drop\_role\_statement ::= DROP ROLE [ IF EXISTS ] role\_name |

DROP ROLE требует, чтобы клиент имел разрешение DROP permission <cql-permissions> для данной роли. Кроме того, клиент не может удалить (DROP) роль, которую он идентифицировал при входе в систему. Наконец, только клиент со статусом Суперпользователь (SUPERUSER) может DROP другую роль SUPERUSER.

Попытка отбросить несуществующую роль приводит к недопустимому условию запроса, если не используется опция IF EXISTS. Если опция используется, а роль не существует, оператор не работает.

|  |
| --- |
| **Примечание.** |
| DROP ROLE намеренно не прерывает никаких открытых пользовательских сеансов. В настоящее время подключенные сеансы останутся подключенными и сохранят возможность выполнять любые действия с базой данных, не требующие авторизации. Однако, если авторизация включена, разрешения удаленной роли также аннулируются в зависимости от параметров кэширования, настроенных в файле cassandra.yaml. Если отброшенная роль впоследствии будет воссоздана и ей будут предоставлены новые разрешения или роли, все еще подключенные клиентские сеансы получат вновь предоставленные разрешения и роли. |

GRANT ROLE

При предоставлении роли другому пользователю используется оператор GRANT ROLE:

|  |
| --- |
| grant\_role\_statement ::= GRANT role\_name TO role\_name |

Например:

|  |
| --- |
| GRANT report\_writer TO alice; |

Этот оператор предоставляет alice роль report\_writer. Все разрешения, предоставленные report\_writer, также получены alice.

Роли моделируются как ориентированный ациклический граф, поэтому циклические гранты не допускаются. Следующие примеры приводят к ошибочным состояниям:

|  |
| --- |
| GRANT role\_a TO role\_b;  GRANT role\_b TO role\_a;  GRANT role\_a TO role\_b;  GRANT role\_b TO role\_c;  GRANT role\_c TO role\_a; |

REVOKE ROLE

Для отзыва роли используется оператор REVOKE ROLE:

|  |
| --- |
| revoke\_role\_statement ::= REVOKE role\_name FROM role\_name |

Например:

|  |
| --- |
| REVOKE report\_writer FROM alice; |

Этот оператор отменяет роль report\_writer у alice. Любые разрешения, которые alice получила через роль report\_writer, также аннулируются.

LIST ROLE

Все известные роли (в системе или назначенные конкретной роли) можно перечислить с помощью оператора LIST ROLES:

|  |
| --- |
| list\_roles\_statement ::= LIST ROLES [ OF role\_name] [ NORECURSIVE ] |

Например:

|  |
| --- |
| LIST ROLES; |

возвращает все известные роли в системе, для этого требуется разрешение DESCRIBE для ресурса ролей базы данных.

В этом примере перечислены все роли, предоставленные alice, в том числе приобретенные транзитивно:

|  |
| --- |
| LIST ROLES OF alice; |

В этом примере перечислены все роли, напрямую предоставленные bob, без включения каких-либо транзитивно приобретенных:

|  |
| --- |
| LIST ROLES OF bob NORECURSIVE; |

Пользователи (Users)

До введения ролей аутентификация и авторизация основывались на концепции USER. Для обратной совместимости был сохранен унаследованный синтаксис, при этом операторы, ориентированные на USER, стали синонимами для эквивалентов, основанных на ROLE. Другими словами, создание / обновление пользователя - это просто другой синтаксис для создания / обновления роли.

CREATE USER

Для создания пользователя используется оператор CREATE USER:

|  |
| --- |
| create\_user\_statement ::= CREATE USER [ IF NOT EXISTS ] role\_name  [ WITH PASSWORD string ]  [ user\_option ]  user\_option: SUPERUSER | NOSUPERUSER |

Например:

|  |
| --- |
| CREATE USER alice WITH PASSWORD 'password\_a' SUPERUSER;  CREATE USER bob WITH PASSWORD 'password\_b' NOSUPERUSER; |

Команда CREATE USER эквивалентна CREATE ROLE, где параметр LOGIN имеет значение true. Итак, следующие пары операторов эквивалентны:

|  |
| --- |
| CREATE USER alice WITH PASSWORD 'password\_a' SUPERUSER;  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true AND SUPERUSER = true;  CREATE USER IF NOT EXISTS alice WITH PASSWORD 'password\_a' SUPERUSER;  CREATE ROLE IF NOT EXISTS alice WITH PASSWORD = 'password\_a' AND LOGIN = true AND SUPERUSER = true;  CREATE USER alice WITH PASSWORD 'password\_a' NOSUPERUSER;  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true AND SUPERUSER = false;  CREATE USER alice WITH PASSWORD 'password\_a' NOSUPERUSER;  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true;  CREATE USER alice WITH PASSWORD 'password\_a';  CREATE ROLE alice WITH PASSWORD = 'password\_a' AND LOGIN = true; |

ALTER USER

Для изменения параметров пользователя используется оператор ALTER USER:

|  |
| --- |
| alter\_user\_statement ::= ALTER USER role\_name [ WITH PASSWORD string] [ user\_option] |

Например:

|  |
| --- |
| ALTER USER alice WITH PASSWORD 'PASSWORD\_A';  ALTER USER bob SUPERUSER; |

DROP USER

Для удаления пользователя используется оператор DROP USER:

|  |
| --- |
| drop\_user\_statement ::= DROP USER [ IF EXISTS ] role\_name |

LIST USER

Существующие пользователи могут быть перечислены с помощью оператора LIST USERS:

|  |
| --- |
| list\_users\_statement::= LIST USERS |

Обратите внимание, что этот оператор эквивалентен LIST ROLES, но в вывод включаются только роли с привилегией LOGIN.

Контроль данных (Data Control)

Разрешения (Permissions)

Разрешения на ресурсы предоставляются ролям; В RT.KeyValue есть несколько различных типов ресурсов, и каждый тип моделируется иерархически:

* Иерархия ресурсов данных, пространств ключей и таблиц имеет структуру ALL KEYSPACES → KEYSPACE → TABLE.
* Ресурсы функций имеют структуру ALL FUNCTIONS → KEYSPACE → FUNCTION.
* Ресурсы, представляющие роли, имеют структуру ALL ROLES → ROLE
* Ресурсы, представляющие имена объектов JMX, которые сопоставляются с наборами компонентов MBeans / MXBean, имеют структуру ALL MBEANS → MBEAN

Разрешения могут быть предоставлены на любом уровне этих иерархий, и они идут вниз. Таким образом, предоставление разрешения для ресурса выше в цепочке автоматически предоставляет такое же разрешение для всех ресурсов ниже. Например, предоставление SELECT для KEYSPACE автоматически предоставляет его для всех TABLES в этом KEYSPACE. Аналогичным образом, предоставление разрешения для ALL FUNCTIONS предоставляет его для каждой определенной функции, независимо от того, в каком пространстве ключей она находится. Также возможно предоставить разрешения для всех функций, ограниченных определенным пространством ключей.

Изменения разрешений видны существующим клиентским сеансам; то есть нет необходимости повторно устанавливать соединения после изменения разрешений.

Полный набор доступных разрешений:

* CREATE
* ALTER
* DROP
* SELECT
* MODIFY
* AUTHORIZE
* DESCRIBE
* EXECUTE

Не все разрешения применимы к каждому типу ресурса. Например, EXECUTE имеет значение только в контексте функций или mbeans; предоставление EXECUTE для ресурса, представляющего таблицу, бессмысленно. Попытка GRANT (предоставить) разрешение для ресурса, к которому оно не может быть применено, приводит к ответу с ошибкой. В **Таблице 12**. показано, какие разрешения могут быть предоставлены для каких типов ресурсов и какие операторы включены этим разрешением.

**Таблица 12. Разрешения для типов ресурсов**.

|  |  |  |
| --- | --- | --- |
| **Разрешение** | **Ресурс** | **Операции** |
| CREATE | ALL KEYSPACES | CREATE KEYSPACE и CREATE TABLE в любом пространстве ключей |
| CREATE | KEYSPACE | CREATE TABLE в указанном пространстве ключей |
| CREATE | ALL FUNCTIONS | CREATE FUNCTION в любом пространстве ключей и CREATE AGGREGATE в любом пространстве ключей |
| CREATE | ALL FUNCTIONS IN KEYSPACE | CREATE FUNCTION и CREATE AGGREGATE в указанном пространстве ключей |
| CREATE | ALL ROLES | CREATE ROLE |
| ALTER | ALL KEYSPACES | ALTER KEYSPACE и ALTER TABLE в любом пространстве ключей |
| ALTER | KEYSPACE | ALTER KEYSPACE и ALTER TABLE в указанном пространстве ключей |
| ALTER | TABLE | ALTER TABLE |
| ALTER | ALL FUNCTIONS | CREATE FUNCTION и CREATE AGGREGATE: замена любых существующих |
| ALTER | ALL FUNCTIONS IN KEYSPACE | CREATE FUNCTION и CREATE AGGREGATE: замена существующих в указанном пространстве ключей |
| ALTER | FUNCTION | CREATE FUNCTION и CREATE AGGREGATE: замена существующих |
| ALTER | ALL ROLES | ALTER ROLE на любую роль |
| ALTER | ROLE | ALTER ROLE |
| DROP | ALL KEYSPACES | DROP KEYSPACE и DROP TABLE в любом пространстве ключей |
| DROP | KEYSPACE | DROP TABLE в указанном пространстве ключей |
| DROP | TABLE | DROP TABLE |
| DROP | ALL FUNCTIONS | DROP FUNCTION и DROP AGGREGATE в любом пространстве ключей |
| DROP | ALL FUNCTIONS IN KEYSPACE | DROP FUNCTION и DROP AGGREGATE в указанном пространстве ключей |
| DROP | FUNCTION | DROP FUNCTION |
| DROP | ALL ROLES | DROP ROLE для любой роли |
| DROP | ROLE | DROP ROLE |
| SELECT | ALL KEYSPACES | SELECT для любой таблицы |
| SELECT | KEYSPACE | SELECT для любой таблицы в указанном пространстве ключей |
| SELECT | TABLE | SELECT в указанной таблице |
| SELECT | ALL MBEANS | Вызов методов получения для любого mbean |
| SELECT | MBEANS | Вызов методов получения для любого mbean, соответствующего шаблону подстановки |
| MODIFY | ALL KEYSPACES | INSERT, UPDATE, DELETE и TRUNCATE для любой таблицы |
| MODIFY | KEYSPACE | INSERT, UPDATE, DELETE и TRUNCATE для любой таблицы в указанном пространстве ключей |
| MODIFY | TABLE | INSERT, UPDATE, DELETE и TRUNCATE в указанной таблице |
| MODIFY | ALL MBEANS | Вызов методов установки для любого mbean |
| MODIFY | MBEANS | Вызов методов установки для любого mbean, соответствующего шаблону подстановки |
| MODIFY | MBEAN | Вызов методов установки для именованного mbean |
| AUTHORIZE | ALL KEYSPACES | GRANT PERMISSION и REVOKE PERMISSION для любой таблицы |
| AUTHORIZE | KEYSPACE | GRANT PERMISSION и REVOKE PERMISSION для любой таблицы в указанном пространстве ключей |
| AUTHORIZE | TABLE | GRANT PERMISSION и REVOKE PERMISSION на указанную таблицу |
| AUTHORIZE | ALL FUNCTIONS | GRANT PERMISSION и REVOKE PERMISSION на любую функцию |
| AUTHORIZE | ALL FUNCTIONS IN KEYSPACE | GRANT PERMISSION и REVOKE PERMISSION в указанном пространстве ключей |
| AUTHORIZE | FUNCTION | GRANT PERMISSION и REVOKE PERMISSION на указанную функцию |
| AUTHORIZE | ALL MBEANS | GRANT PERMISSION и REVOKE PERMISSION на любые mbean |
| AUTHORIZE | MBEANS | GRANT PERMISSION и REVOKE PERMISSION для любого mbean, соответствующего шаблону подстановки |
| AUTHORIZE | MBEAN | GRANT PERMISSION и REVOKE PERMISSION на имя mbean |
| AUTHORIZE | ALL ROLES | GRANT ROLE и REVOKE ROLE для любой роли |
| AUTHORIZE | ROLES | GRANT ROLE и REVOKE ROLE для указанных ролей |
| DESCRIBE | ALL ROLES | LIST ROLES для всех ролей или только ролей, предоставленных другой указанной роли |
| DESCRIBE | ALL MBEANS | Получение метаданных о любом mbean из MBeanServer платформы |
| DESCRIBE | MBEANS | Получение метаданных о любых mbean, соответствующих шаблону с подстановочными знаками, из MBeanServer платформы |
| DESCRIBE | MBEAN | Получение метаданных об именованном mbean из MBeanServer платформы |
| EXECUTE | ALL FUNCTIONS | SELECT, INSERT и UPDATE с использованием любой функции и использование любой функции в CREATE AGGREGATE |
| EXECUTE | ALL FUNCTIONS IN KEYSPACE | SELECT, INSERT и UPDATE с использованием любой функции в указанном пространстве ключей и использование любой функции в пространстве ключей в CREATE AGGREGATE |
| EXECUTE | FUNCTION | SELECT, INSERT и UPDATE с использованием указанной функции и использование функции в CREATE AGGREGATE |
| EXECUTE | ALL MBEANS | Выполнять операции для любого mbean |
| EXECUTE | MBEANS | Выполнять операции для любого mbean, соответствующем подстановочному шаблону |
| EXECUTE | MBEAN | Выполнять операции с именем mbean |

GRANT PERMISSION

Для предоставления разрешения используется оператор GRANT PERMISSION:

|  |
| --- |
| grant\_permission\_statement ::= GRANT permissions ON resource TO role\_name  permissions ::= ALL [ PERMISSIONS ] | permission [ PERMISSION ]  permission ::= CREATE | ALTER | DROP | SELECT | MODIFY | AUTHORIZE | DESCRIBE | EXECUTE  resource ::= ALL KEYSPACES  | KEYSPACE keyspace\_name  | [ TABLE ] table\_name  | ALL ROLES  | ROLE role\_name  | ALL FUNCTIONS [ IN KEYSPACE keyspace\_name ]  | FUNCTION function\_name '(' [ cql\_type( ',' cql\_type )\* ] ')'  | ALL MBEANS  | ( MBEAN | MBEANS ) string |

Например:

|  |
| --- |
| GRANT SELECT ON ALL KEYSPACES TO data\_reader; |

В этом примере любому пользователю с ролью data\_reader предоставляется разрешение на выполнение операторов SELECT для любой таблицы во всех пространствах ключей:

|  |
| --- |
| GRANT MODIFY ON KEYSPACE keyspace1 TO data\_writer; |

Чтобы предоставить любому пользователю с ролью data\_writer разрешение выполнять запросы UPDATE, INSERT, UPDATE, DELETE и TRUNCATE для всех таблиц в пространстве ключей keyspace1:

|  |
| --- |
| GRANT DROP ON keyspace1.table1 TO schema\_owner; |

Чтобы предоставить любому пользователю с ролью schema\_owner права на удаление (DROP) определенного keyspace1.table1:

|  |
| --- |
| GRANT EXECUTE ON FUNCTION keyspace1.user\_function( int ) TO report\_writer; |

Эта команда предоставляет любому пользователю с ролью report\_writer разрешение на выполнение запросов SELECT, INSERT и UPDATE, которые используют функцию keyspace1.user\_function( int ):

|  |
| --- |
| GRANT DESCRIBE ON ALL ROLES TO role\_admin; |

Это дает любому пользователю с правами роли role\_admin просматривать любые и все роли в системе с помощью оператора LIST ROLES.

GRANT ALL

Когда используется форма GRANT ALL, соответствующий набор разрешений определяется автоматически на основе целевого ресурса.

Automatic Granting

Когда ресурс создается с помощью операторов CREATE KEYSPACE, CREATE TABLE, CREATE FUNCTION, CREATE AGGREGATE или CREATE ROLE, создателю (роль пользователя базы данных, который выдает этот оператор), автоматически предоставляются все применимые разрешения на новый ресурс.

REVOKE PERMISSION

Для отзыва разрешения у роли используется инструкция REVOKE PERMISSION:

|  |
| --- |
| revoke\_permission\_statement ::= REVOKE permissions ON resource FROM role\_name |

Например:

|  |
| --- |
| REVOKE SELECT ON ALL KEYSPACES FROM data\_reader;  REVOKE MODIFY ON KEYSPACE keyspace1 FROM data\_writer;  REVOKE DROP ON keyspace1.table1 FROM schema\_owner;  REVOKE EXECUTE ON FUNCTION keyspace1.user\_function( int ) FROM report\_writer;  REVOKE DESCRIBE ON ALL ROLES FROM role\_admin; |

Из-за того, что они используются в обычных операциях с драйверами, для некоторых таблиц нельзя отозвать разрешения SELECT. Следующие таблицы будут доступны всем авторизованным пользователям независимо от их роли:

|  |
| --- |
| \* `system\_schema.keyspaces`  \* `system\_schema.columns`  \* `system\_schema.tables`  \* `system.local`  \* `system.peers` |

LIST PERMISSIONS

Для вывода списка предоставленных разрешений используется оператор LIST PERMISSIONS:

|  |
| --- |
| list\_permissions\_statement ::= LIST permissions [ ON resource] [ OF role\_name[ NORECURSIVE ] ] |

Например:

|  |
| --- |
| LIST ALL PERMISSIONS OF alice; |

Показать все разрешения, предоставленные Алисе, в том числе полученные транзитивно от любых других ролей.

|  |
| --- |
| LIST ALL PERMISSIONS ON keyspace1.table1 OF bob; |

Показать все разрешения на keyspace1.table1, предоставленные bob, в том числе полученные транзитивно от любых других ролей. Это также включает любые разрешения выше по иерархии ресурсов, которые могут быть применены к keyspace1.table1. Например, если у bob есть разрешение ALTER на keyspace1, оно будет включено в результаты этого запроса.

Добавление переключателя NORECURSIVE ограничивает результаты только теми разрешениями, которые были напрямую предоставлены bob или одной из ролей `bob.

|  |
| --- |
| LIST SELECT PERMISSIONS OF carlos; |

Показать все разрешения, предоставленные carlos или carlos’s roles, ограниченные разрешениями `SELECT (limited to `SELECT) для любого ресурса.

Триггеры (Triggers)

Триггеры идентифицируются по имени, определяемому:

|  |
| --- |
| trigger\_name ::= identifier |

CREATE TRIGGER

Для создания нового триггера используется оператор CREATE TRIGGER:

|  |
| --- |
| create\_trigger\_statement ::= CREATE TRIGGER [ IF NOT EXISTS ] trigger\_name  ON table\_name  USING string |

Например:

|  |
| --- |
| CREATE TRIGGER myTrigger ON myTable USING 'org.apache.cassandra.triggers.InvertedIndex'; |

Фактическая логика, составляющая триггер, может быть написана на любом языке Java (JVM) и существовать вне базы данных. Вы помещаете код триггера в подкаталог lib/triggers каталога установки RT.KeyValue, он загружается во время запуска кластера и существует на каждом узле, который участвует в кластере. Триггер, определенный для таблицы, срабатывает до выполнения запрошенного оператора DML, что обеспечивает атомарность транзакции.

CREATE TRIGGER

Для удаления триггера используется оператор DROP TRIGGER:

|  |
| --- |
| drop\_trigger\_statement ::= DROP TRIGGER [ IF EXISTS ] trigger\_nameON table\_name |

Например:

|  |
| --- |
| DROP TRIGGER myTrigger ON myTable; |

1. КЛЮЧЕВЫЕ СЛОВА CQL

CQL различает зарезервированные и незарезервированные ключевые слова. Зарезервированные ключевые слова не могут использоваться в качестве идентификатора, они действительно зарезервированы для языка (но можно заключить зарезервированное ключевое слово в двойные кавычки, чтобы использовать его в качестве идентификатора). Однако незарезервированные ключевые слова имеют определенное значение только в определенном контексте, но в противном случае могут использоваться в качестве идентификатора. Единственная причина существования этих незарезервированных ключевых слов - удобство: некоторые ключевые слова не зарезервированы, когда синтаксическому анализатору всегда было легко решить, использовались ли они в качестве ключевых слов или нет.

|  |  |
| --- | --- |
| **Ключевое слово** | **Зарезервировано?** |
| ADD | да |
| AGGREGATE | нет |
| ALL | нет |
| ALLOW | да |
| ALTER | да |
| AND | да |
| APPLY | да |
| AS | нет |
| ASC | да |
| ASCII | нет |
| AUTHORIZE | да |
| BATCH | да |
| BEGIN | да |
| BIGINT | нет |
| BLOB | нет |
| BOOLEAN | нет |
| BY | да |
| CALLED | нет |
| CLUSTERING | нет |
| COLUMNFAMILY | да |
| COMPACT | нет |
| CONTAINS | нет |
| COUNT | нет |
| COUNTER | нет |
| CREATE | да |
| CUSTOM | нет |
| DATE | нет |
| DECIMAL | нет |
| DELETE | да |
| DESC | да |
| DESCRIBE | да |
| DISTINCT | нет |
| DOUBLE | нет |
| DROP | да |
| ENTRIES | да |
| EXECUTE | да |
| EXISTS | нет |
| FILTERING | нет |
| FINALFUNC | нет |
| FLOAT | нет |
| FROM | да |
| FROZEN | нет |
| FULL | да |
| FUNCTION | нет |
| FUNCTIONS | нет |
| GRANT | да |
| IF | да |
| IN | да |
| INDEX | да |
| INET | нет |
| INFINITY | да |
| INITCOND | нет |
| INPUT | нет |
| INSERT | да |
| INT | нет |
| INTO | да |
| JSON | нет |
| KEY | нет |
| KEYS | нет |
| KEYSPACE | да |
| KEYSPACES | нет |
| LANGUAGE | нет |
| LIMIT | да |
| LIST | нет |
| LOGIN | нет |
| MAP | нет |
| MODIFY | да |
| NAN | да |
| NOLOGIN | нет |
| NORECURSIVE | да |
| NOSUPERUSER | нет |
| NOT | да |
| NULL | да |
| OF | да |
| ON | да |
| OPTIONS | нет |
| OR | да |
| ORDER | да |
| PASSWORD | нет |
| PERMISSION | нет |
| PERMISSIONS | нет |
| PRIMARY | да |
| RENAME | да |
| REPLACE | да |
| RETURNS | нет |
| REVOKE | да |
| ROLE | нет |
| ROLES | нет |
| SCHEMA | да |
| SELECT | да |
| SET | да |
| SFUNC | нет |
| SMALLINT | нет |
| STATIC | нет |
| STORAGE | нет |
| STYPE | нет |
| SUPERUSER | нет |
| TABLE | да |
| TEXT | нет |
| TIME | нет |
| TIMESTAMP | нет |
| TIMEUUID | нет |
| TINYINT | нет |
| TO | да |
| TOKEN | да |
| TRIGGER | нет |
| TRUNCATE | да |
| TTL | нет |
| TUPLE | нет |
| TYPE | нет |
| UNLOGGED | да |
| UPDATE | да |
| USE | да |
| USER | нет |
| USERS | нет |
| USING | да |
| UUID | нет |
| VALUES | нет |
| VARCHAR | нет |
| VARINT | нет |
| WHERE | да |
| WITH | да |
| WRITETIME | нет |