Лабораторная работа «Работа с колоночной NOSQL БД на примере CassandraDb»
по дисциплине «Постреляционные базы данных»
Цель работы:

1. Изучить модель представления данных и способы работы с колоночными БД NoSql.
2. Освоить методы создания колоночной БД и языки запросов к ним.
3. Получить навыки работы с колоночной БД CassandraDb.

Время выполнения:

Время выполнения лабораторной работы 4 часа.

Литература:

1. Фаулер, Мартин, Садаладж, Прамодкумар Дж. NoSQL: новая методология разработки нереляционных баз данных. : Пер. с англ. - М.: ООО "И.Д. Вильямс", 2013г.
2. Теория - https://habrahabr.ru/post/155115/
3. Документация - http://cassandra.apache.org/doc/latest/ ; Язык CQL - http://cassandra.apache.org/doc/latest/cql/index.html

Пункты задания для выполнения:

Задание 1. Создание БД (базовая часть)

1.1 Создать в среде CassandraDb свое пространство ключей. (см теорию [2], документацию [3]). Использование cqlsh командной строки, далее примеры команд на языке CQL:

CREATE KEYSPACE myTestKS WITH REPLICATION = { ‘class’ : ‘SimpleStrategy’, ‘replication_factor’ : 1 };

1.2. Использовать новое пространство ключей: Use myTestKS;

1.3. Определить семейство столбцов по теме своего ДЗ. Добавить в семейство столбцов строки с данными. Продемонстрировать (вывести на экран) содержимое БД. (примеры и пояснения по типам ключей см. https://habr.com/ru/post/203200/).

Задание 2. CRUD и работа с индексами (базовая часть)

2.1. Продемонстрировать добавление, изменение и удаление данных в БД, используя команды API и/или язык Cassandra Query Language. Примеры (см [3]):
INSERT INTO NerdMovies (movie, director, main_actor, year)

```
        VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005);

UPDATE NerdMovies  SET director   = 'Joss Whedon',
      main_actor = 'Nathan Fillion',
      year       = 2005
 WHERE movie = 'Serenity';

UPDATE UserActions
  SET total = 123
  WHERE user = B70DE1D0-9908-4AE3-BE34-5573E5B09F14
    AND action = 'click';

DELETE FROM NerdMovies WHERE movie = 'Serenity';

DELETE phone FROM Users
 WHERE userid IN (C73DE1D3-AF08-40F3-B124-3FF3E5109F22, B70DE1D0-9908-
4AE3-BE34-5573E5B09F14);
```

2.2. Определить для семейства столбцов индекс(ы). Выполнить запросы к с фильтрацией по ключам и индексам. Продемонстрировать работу allow filtering.

(см. примеры и описание https://habr.com/ru/post/205176/).

Задание 3. Запросы к БД. (базовая часть)

3.1. Выполнить запросы к базе данных с селекцией и проекцией.

(см. примеры и описание https://habr.com/ru/post/205176/)

3.2. Выполнить запрос с использованием агрегатных функций.

(см. документацию [3]).

3.3. Добавить строку с указанием TTL, продемонстрировать действие TTL (время существования значения в секундах). Примеры [см. 3]:

```
        INSERT INTO NerdMovies (movie, director, main_actor, year)
          VALUES ('Serenity', 'Joss Whedon', 'Nathan Fillion', 2005)
    USING TTL 86400;

UPDATE NerdMovies USING TTL 400
  SET director   = 'Joss Whedon'
 WHERE movie = 'Serenity';
```

Задание 4. Группировка и сортировка (Хорошо)

4.1. Выполнить запросы с группировкой и сортировкой данных.
(см https://howtoprogram.xyz/2017/02/18/using-group-apache-cassandara/ **;** документация [3]).

4.2. Создать еще одно семейство столбцов по теме ДЗ, определить для него кластерный и распределительный ключи. Выполнить запросы к с фильтрацией по ключам.
(см. примеры и описание https://habr.com/ru/post/205176/)

4.3. Продемонстрировать усечение таблицы и удаление таблицы/индекса
(команды **truncate**, **drop ;** документация [3]).

Задание 5. Дополнительные возможности (Отлично)

5.1. Создать материализованное представление. Продемонстрировать работу с ним.
(См. MATERIALIZED VIEW, документация в [3])

5.2. Продемонстрировать создание пакета запросов
(команда **begin Batch … apply batch** , см. https://habr.com/ru/post/204026/ и документацию [3]).

5.3. Продемонстрировать изменение и удаление данных в БД, используя условия
(IF для Update и Delete, документация [3]).

Оглавление

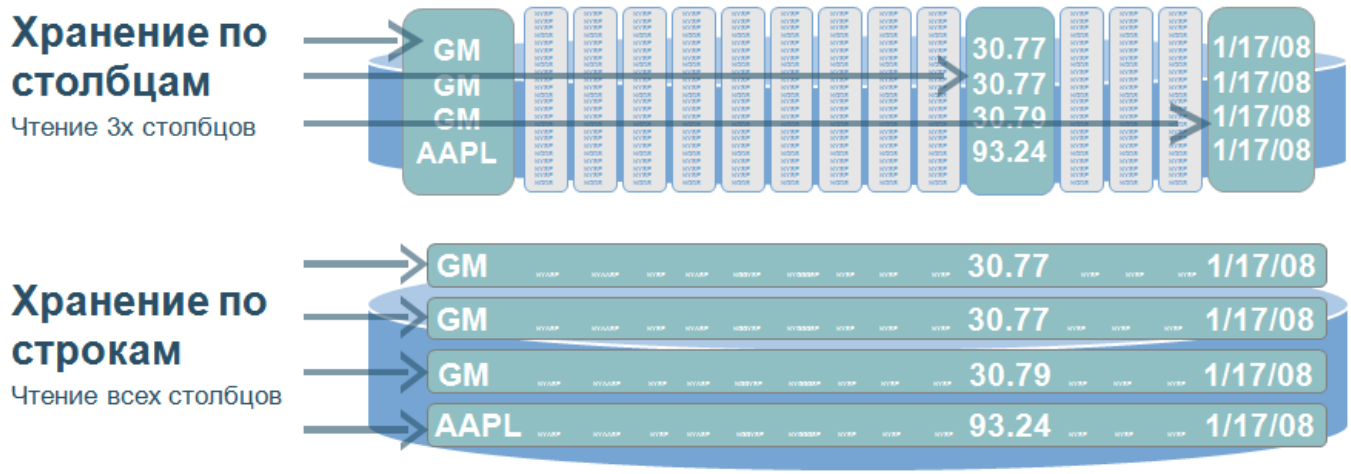
Теоретические сведения

Apache Cassandra – это нереляционная отказоустойчивая распределенная СУБД, рассчитанная на создание высокомасштабируемых и надёжных хранилищ огромных массивов данных, представленных в виде хэша. Проект был разработан на языке Java в корпорации Facebook в 2008 году, и передан фонду Apache Software Foundation в 2009. Эта СУБД относится к гибридным NoSQL-решениям, поскольку она сочетает модель хранения данных на базе семейства столбцов (ColumnFamily) с концепцией key-value (ключ-значение).

Основная идея колоночных СУБД — это хранение данных не по строкам, как это делают традиционные СУБД, а по колонкам. Это означает, что с точки зрения SQL-клиента данные представлены как обычно в виде таблиц, но физически эти таблицы являются совокупностью колонок, каждая из которых по сути представляет собой таблицу из одного поля. При этом физически на диске значения одного поля хранятся последовательно друг за другом — приблизительно так:

[A1, A2, A3], [B1, B2, B3], [C1, C2, C3] и т.д.

Такая организация данных приводит к тому, что при выполнении select в котором фигурируют только 3 поля из 50 полей таблицы, с диска **физически** будут прочитаны только 3 колонки. Это означает что нагрузка на канал ввода-вывода будет приблизительно в 50/3=17 раз меньше чем при выполнении такого же запроса в традиционной СУБД.



Модель данных Apache Cassandra

Модель данных Cassandra состоит из следующих элементов:

- **столбец или колонка (column)** – ячейка с данными, включающая 3 части – имя (column name) в виде массива байтов, метку времени (timestamp) и само

значение (value) также в виде байтового массива. С каждым значением связана **метка времени** — задаваемое пользователем 64-битное число, которое используется для разрешения конфликтов во время записи: чем оно больше, тем новее считается столбец. Это учитывается при удалении старых колонок.

- **строка или запись (row)** – именованная коллекция столбцов;
- **семейство столбцов (column family)** – именованная коллекция строк;
- **пространство ключей (keyspace)** – группа из нескольких семейств столбцов, собранных вместе. Оно логически группирует семейства столбцов и обеспечивает изолированные области имен.

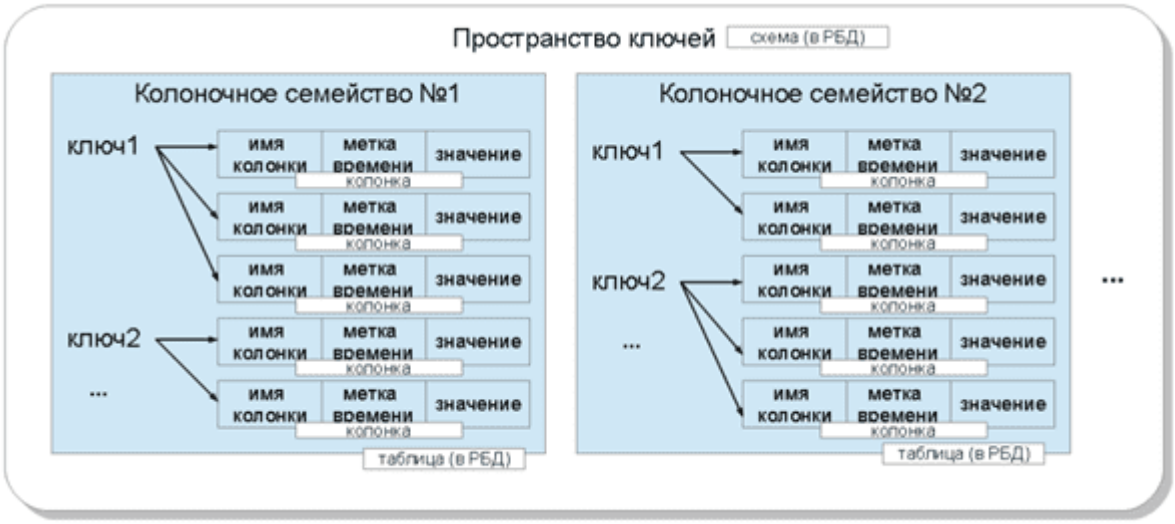
В сравнении с хранением данных в строках, как в большинстве реляционных баз данных, преимущества хранения в колонках заключаются в быстром поиске/доступе и агрегации данных. Реляционные базы данных хранят каждую строку как непрерывную запись на диске. Разные строки хранятся в разных местах на диске, в то время как колоночные базы данных хранят все ячейки, относящиеся к колонке, как непрерывную запись, что делает операции поиска/доступа быстрее.

Пример: получение списка заголовков нескольких миллионов статей будет трудоёмкой задачей при использовании реляционных баз данных, так как для извлечения заголовков придётся проходить по каждой записи. А можно получить все заголовки с помощью только одной операции доступа к диску.

В рассматриваемой СУБД доступны следующие **типы данных**:

- BytesType: любые байтовые строки (без валидации);
- AsciiType: ASCII строка;
- UTF8Type: UTF-8 строка;
- IntegerType: число с произвольным размером;
- Int32Type: 4-байтовое число;
- LongType: 8-байтовое число;
- UUIDType: UUID 1-ого или 4-ого типа;
- TimeUUIDType: UUID 1-ого типа;
- DateType: 8-байтовое значение метки времени;
- BooleanType: два значения: true = 1 или false = 0;
- FloatType: 4-байтовое число с плавающей запятой;
- DoubleType: 8-байтовое число с плавающей запятой;
- DecimalType: число с произвольным размером и плавающей запятой;
- CounterColumnType: 8-байтовый счётчик.

Пространство ключей соответствует понятию схемы базы данных (database schema) в реляционной модели, а находящиеся в нем семейства столбцов – реляционной таблице. Столбцы объединяются в записи при помощи ключа (row key) в виде массива байтов, по значению которого столбцы упорядочены в пределах одной записи. В отличие от реляционных СУБД, в NoSQL модели возможна ситуация, когда разные строки содержат разное число колонок или столбцы с такими же именами, как и в других записях.



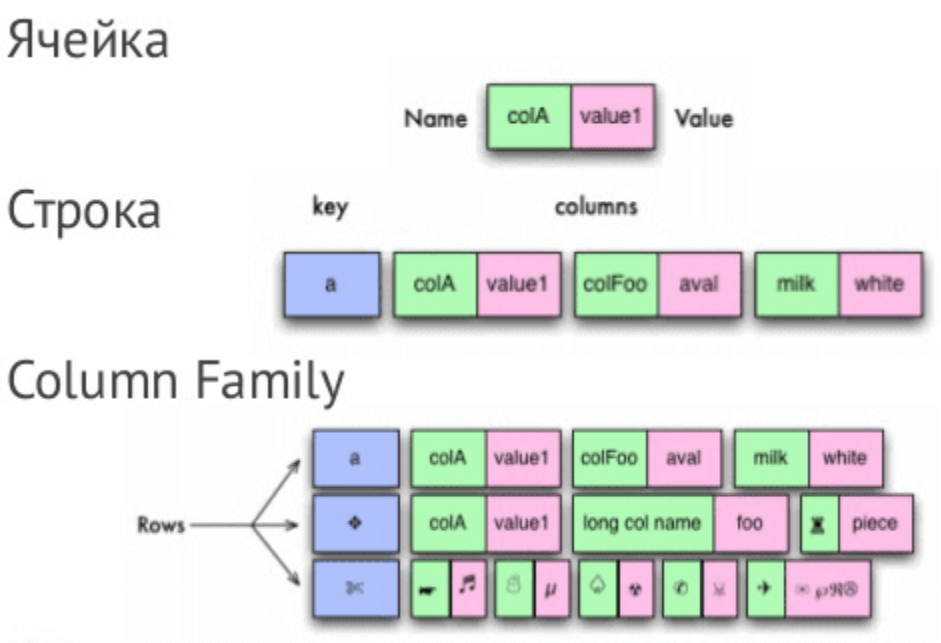
Соответствие понятий модели данных NoSQL-СУБД Cassandra реляционной парадигме

Можно сказать, конкретное значение, хранимое в Apache Cassandra, идентифицируется следующими привязками:

- к приложению или предметной области в пространстве ключей, что позволяет на одном кластере размещать данные разных приложений;
- к запросу в рамках семейства столбцов;
- к узлу кластера с помощью ключа, который определяет, на какие узлы попадут сохранённые колонки;
- к атрибуту в записи с помощью имени колонки, что позволяет в одной записи хранить несколько значений.

Структура данных при этом выглядит так:

- *Keyspace*
  - *ColumnFamily*
    - *Row*
      - *Key*
      - *Column*
        - *Name*
        - *Value*
      - *Column*
      - *…*
    - *…*



Модель данных Apache Cassandra

Установка для Ubuntu Linux

Cassandra — ресурсозатратная СУБД, поэтому рекомендуется имеет не менее 2 Гб оперативной памяти в системе.

Обновляем программные пакеты:

```
$ apt-get update
```

Для работы Cassandra нам предварительно необходимо установить JDK.

```
$ apt-get install default-jdk
```

Проверяем версию java

```
$ java -version
```

```
root@cs57300:~# java -version
openjdk version "1.8.0_111"
OpenJDK Runtime Environment (build 1.8.0_111-8u111-b14-2ubuntu0.16.04.2-b14)
OpenJDK 64-Bit Server VM (build 25.111-b14, mixed mode)
```

Переходим непосредственно к самой установки Cassandra. Мы будем устанавливать версию 3.9.

Выполняем в командной строке:

```
$ echo "deb http://debian.datastax.com/datastax-ddc 3.9 main" | sudo tee -a
/etc/apt/sources.list.d/cassandra.sources.list
```

Устанавливаем утилиту curl

```
$ apt-get install curl
```

Снова обновляем программные пакеты:

```
$ apt-get update
```

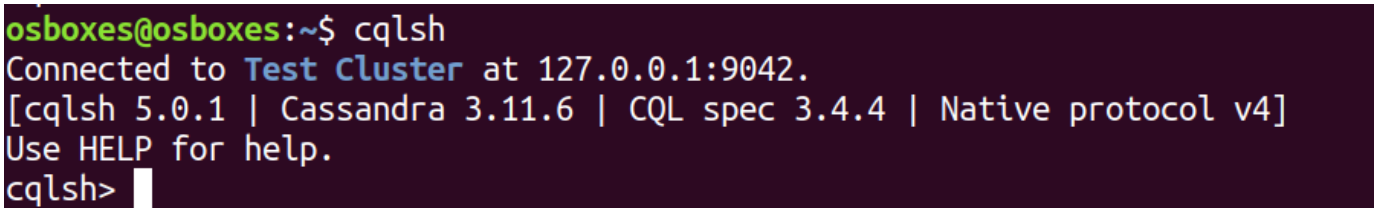
И устанавливаем Cassandra

```
$ apt-get install datastax-ddc
```

Проверяем статус

```
$ service cassandra status
```

Чтобы попасть в командную оболочку, необходимо в командной строке ввести cqlsh



Создаем пространство ключей:

CREATE KEYSPACE [IF NOT EXISTS] ***keyspace_name***

WITH REPLICATION = {

'class' : 'SimpleStrategy', 'replication_factor' : ***N*** }

| 'class' : 'NetworkTopologyStrategy',

'***dc1_name***' : ***N*** [, ...]

};

keyspace_name – название пространства ключей.

dc1_name – имя узла

REPLICATION = { *replication_map* }

Карта репликации определяет, сколько копий данных хранится в данном узле. Этот параметр влияет на согласованность, доступность и скорость запроса.

Класс стратегии репликации и настройки факторов

Класс	Фактор	Описание
'SimpleStrategy'	'replication_factor' : *N*	Для всего кластера будет использоваться один коэффициент репликации. Параметр N означает количество копий данных, должен быть целым числом.
'NetworkTopologyStrategy'	'*datacenter_name*' : *N*	Для каждого узла задается свой коэффициент репликации. Параметр N означает количество копий данных, должен быть целым числом.

Примеры задания топологий:

Простая топология:

'class' : 'SimpleStrategy', 'replication_factor' : 1

Сетевая топология:

'class' : 'NetworkTopologyStrategy',

‘London_dc’ : 1, ‘Moscow_dc’:2

Более подробно про параметры Вы можете прочитать в официальной документации [3]

Создадим пространство ключей test с стратегией репликации 'SimpleStrategy' и коэффициентом репликации 1.

```
cqlsh> CREATE KEYSPACE test WITH REPLICATION = { 'class' : 'SimpleStrategy', 're
plication_factor' : 1};
```

Перейдем в созданное пространство

```
cqlsh> use test
```

CREATE TABLE

Создадим семейство колонок:

Синтаксис создания:

```
CREATE TABLE ИМЯ_ТАБЛИЦЫ(
ИМЯ_КОЛОНКИ_1 ТИП_ДАННЫХ,
ИМЯ_КОЛОНКИ_2 ТИП_ДАННЫХ,
```

ИМЯ_КОЛОНКИ_3 ТИП_ДАННЫХ,
PRIMARY KEY (ИМЯ_КОЛОНКИ_1)
)
В CREATE TABLE должен быть обязательно объявлен PRIMARY KEY для индексации пространства колонок.
CREATE TABLE student (id uuid, citizenship text, first_name text, last_name text, age int, PRIMARY KEY (id));
В этом примере будет создано семейство колонок student с полями id типа uuid, citizenship, first_name, last_name типа text, age типа int.
В данном случае первичным ключом будет являться поле id.

Создадим таблицу test

```
> CREATE TABLE IF NOT EXISTS test (
 id timeuuid,
 title text,
 PRIMARY KEY (title, id)
) WITH default_time_to_live = 120 and CLUSTERING ORDER BY (id DESC);
```

Будет создано семейство колонок с полями id и title.

Первичный ключ состоит из распределительного и кластерного ключа. Распределительный ключ отвечает за распределение данных по узлам, а кластерный за группировку данных внутри узла. Ключи сами по себе могут быть составными.

В данном случае title – распределительный ключ, а id - кластерный

Этап проектирования первичного ключа является самым важным, так как от этого напрямую зависит вся эффективность системы.

Конструкция CLUSTERING ORDER BY определяет сортировку данных.

Для изменения структуры таблицы используется команда следующего вида:

ALTER TABLE <ИМЯ_ТАБЛИЦЫ> <ДАННЫЕ_ДЛЯ_ИЗМЕНЕНИЯ>

Изменим тип id в таблице test
ALTER TABLE test ALTER ID TYPE uuid;

Как видно, язык запросов очень похож на традиционный SQL.
Вставим в колонки значения:

INSERT INTO student (id, citizenship, first_name, last_name, age) VALUES (now(), 'Russia', 'Ivan', 'Ivanov', 25);
Выполнив select –запрос отобразим все значения.
SELECT * FROM tablename;

select * from student;

```
cqlsh:lab7> select * from student;

 id                                   | age | citizenship | first_name | last_name
--------------------------------------+-----+-------------+------------+-----------
 3a26e100-9aeb-11ea-b1d1-3148925e06e7 |  25 |      Russia |       Ivan |    Ivanov
 6d0317a0-9aec-11ea-b1d1-3148925e06e7 |  22 |      France |       test |    Petrov
 47957270-9aeb-11ea-b1d1-3148925e06e7 |  35 |      Russia |       Petr |    Petrov
```

Update и Delete

Операторы Update и Delete так же имеют традиционный вид.
Пример UPDATE:
UPDATE имя_таблицы SET название_колонки=значение WHERE условие
Обновим поля first_name и last_name и строки с заданным id:
UPDATE student SET first_name = 'Example', last_name='Example' WHERE id=54daf810-9aeb-11ea-b1d1-3148925e06e7

Пример DELETE:
DELETE FROM название_таблицы WHERE условие.
DELETE FROM student WHERE id=54daf810-9aeb-11ea-b1d1-3148925e06e7;
В данном случае происходит поиск по ключу.

Создание индекса

В SELECT запросах Cassandra Query Language (**CQL**) отсутствуют привычные нам SQL операции JOIN, GROUP BY. А операция WHERE сильно урезана. В SQL вы можете фильтровать по любой колонке, тогда как в CQL только по распределительным ключам (*partition key*), кластерным ключам (*clustering columns*) и вторичным индексам.

В Cassandra можно создавать вторичные INDEX-ы у любой колонки наподобие SQL индексов. Фактически же, вторичные индексы Cassandra — это скрытая дополнительная таблица, поэтому производительность WHERE запросов по ним хуже запросов по ключевым колонкам.

СУБД не дает возможности по умолчанию искать по неключевым колонкам, для которых не создан индекс.

Запрос SELECT * FROM student WHERE citizenship=’Russia’ выдаст ошибку:

```
... ;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this q
uery as it might involve data filtering and thus may have unpredictable performance. If you
 want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
```

Решением этой проблемы является создание вторичного индекса или использование конструкции allow filtering.

Рассмотрим создание индекса:

Вы можете создать индекс в Cassandra с помощью команды **CREATE INDEX.** Его синтаксис выглядит следующим образом:

```
CREATE INDEX <identifier> ON <tablename>
```

Создадим индекс по колонке citizenship.

CREATE INDEX citizenshipIndex ON student (citizenship);

После этого можно выполнять запрос с условием на эту колонку.

SELECT * FROM student WHERE citizenship='Russia';

```
id                                   | age | citizenship | first_name | last_name
--------------------------------------+-----+-------------+------------+-----------
3a26e100-9aeb-11ea-b1d1-3148925e06e7 |  25 |      Russia |       Ivan |    Ivanov
47957270-9aeb-11ea-b1d1-3148925e06e7 |  35 |      Russia |       Petr |    Petrov
```

ALLOW FILTERING

Если попытаться выполнить запрос SELECT * FROM student WHERE citizenship='Russia' без использования индекса, возникнет ошибка.

```
... ;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this q
uery as it might involve data filtering and thus may have unpredictable performance. If you
 want to execute this query despite the performance unpredictability, use ALLOW FILTERING"
```

Cassandra знает, что она не сможет выполнить запрос эффективным способом. Поэтому он предупреждает вас: «Будьте осторожны. Выполнение этого запроса как такового может быть не очень хорошей идеей, поскольку оно может использовать много ваших вычислительных ресурсов ».

Единственный способ, которым Cassandra может выполнить этот запрос, - это извлечь все строки из таблицы, а затем отфильтровать те, у которых есть запрошенное значение для столбца citizenship.

Если ваша таблица содержит, например, 1 миллион строк, и 95% из них имеют запрошенное значение для столбца citizenship, запрос все равно будет относительно эффективным, и вам следует использовать ALLOW FILTERING.

С другой стороны, если ваша таблица содержит 1 миллион строк и только 2 строки содержат запрошенное значение для столбца citizenship, ваш запрос крайне неэффективен. Cassandra будет бессмысленно загружать 999, 998 строк. Если запрос часто используется, возможно, лучше добавить индекс в столбец citizenship.

К сожалению, у Cassandra нет возможности провести различие между двумя вышеупомянутыми случаями, поскольку они зависят от распределения данных таблицы. Поэтому Cassandra предупреждает вас и полагается на вас, чтобы сделать правильный выбор.

Пример:
SELECT * FROM student WHERE last_name=’Ivanov’ ALLOW FILTERING;
В данном запросе будут отобраны все студенты с фамилией Иванов.

```
cqlsh:lab7> SELECT * FROM student WHERE last_name='Ivanov' ALLOW FILTERING;

 id                                   | age | citizenship | first_name | last_name
--------------------------------------+-----+-------------+------------+-----------
 3a26e100-9aeb-11ea-b1d1-3148925e06e7 |  25 |      Russia |       Ivan |    Ivanov
```

Ограничения WHERE для операторов SELECT

Ограничения распределительных ключей

Столбцы распределительного ключа поддерживают только два оператора: = и IN.
К примеру, создадим таблицу с составным распределительным ключом:

CREATE TABLE numberOfRequests (cluster text, date text, time text, numberOfRequests int, PRIMARY KEY ((cluster, date), time))
Будет создана таблица с полями cluster, date, time типа text, numberOfRequests типа int и распределительным и кластерным ключами (cluster, date) и time.

В условиях на распределительный ключ вы можете использовать операторы = и IN.
Например:
SELECT * FROM numberOfRequests WHERE cluster IN ('cluster1', 'cluster2') AND date = '2015-05-06';

Запрос:

SELECT * FROM numberOfRequests WHERE cluster = 'cluster1' and time = '12: 00 ';

будет отклонен, поскольку столбец даты не используется. Причина в том, что Cassandra нужны все столбцы ключа раздела, чтобы иметь возможность вычислять хеш, который позволит находить узлы, содержащие раздел. Если для ключей раздела не указаны ограничения, но некоторые из них указаны для ключей кластеризации, Cassandra потребует, чтобы в запрос был добавлен параметр ALLOW FILTERING.

Ограничения кластерных ключей

Кластерные ключи поддерживают операторы =, IN,>,> =, <=, <, CONTAINS и CONTAINS KEY в ограничениях на один столбец и операторы =, IN,>,> =, <= и <в ограничениях на несколько столбцов.
Использование кластерных ключей в запросе:

Предположим, существует таблица

CREATE TABLE numberOfRequests (cluster text, date text, datacenter text, hour int, minute int, numberOfRequests int, PRIMARY KEY ((cluster, date), datacenter, hour, minute))
Будет создана таблица с полями cluster, date, datacenter типа text, hour, minute, numberOfRequests типа int и распределительным и кластерным ключами (cluster, date) и datacenter, hour, minute.

Вы можете видеть, что для эффективного извлечения данных без вторичного индекса вам нужно знать все ключевые кластерные ключи для выбора. Итак, если вы выполните:

SELECT * FROM numberOfRequests WHERE cluster = ‘cluster1’ AND date = ‘2015-06-05’ AND datacenter = 'US_WEST_COAST' AND hour = 14 AND minute = 00;

Cassandra найдет данные эффективно, но если вы выполните:

SELECT * FROM numberOfRequests, WHERE cluster = ‘cluster1’ AND date = ‘2015-06-05’ AND hour = 14 AND minute = 00; (в запросе отсутствует условие на кластерный ключ datacenter)

Cassandra отклонит запрос, поскольку ей придется сканировать весь раздел, чтобы найти запрошенные данные, что неэффективно.

Более подробную информацию Вы можете прочитать в официальной документации [5]

Запросы с использованием агрегатных функций

В Cassandra стандартные агрегатные функции min, max, avg, sum и count являются встроенными функциями.
Функция min находит минимальное значение, max находит максимальное значение, avg находит среднее значение, sum – суммирует значения в указанной колонке и функция count определяет количество значений в колонке.
Воспользуемся функцией min() для вычисления минимального возраста.
SELECT min(age) AS min_age FROM student WHERE citizenship='Russia';

Ключевое слово TTL

Ключевое слово TTL позволяет указать, сколько будет существовать запись (созданная, или измененная)
INSERT и UPDATE поддерживают TTL. Время задается в секундах.
Например, запрос INSERT INTO student (id, citizenship, first_name, last_name, age) VALUES (now(), 'Test', 'Test', 'Test', 34) USING TTL 15; создаст запись, которая будет существовать 15 секунд. Через указанное время она исчезнет.
Поведение аналогично для UPDATE. После истечения указанного времени измененное значение заменится на null.

Более подробно про TTL Вы можете прочитать в официальной документации [4]

Пример:
UPDATE student USING TTL 15 SET age=1 WHERE id=54daf810-9aeb-11ea-b1d1-3148925e06e7.
Видно, что значение изменилось:

id	age	citizenship	first_name	last_name
3a26e100-9aeb-11ea-b1d1-3148925e06e7	1	Russia	Ivan	Ivanov
6d0317a0-9aec-11ea-b1d1-3148925e06e7	22	France	test	Petrov
47957270-9aeb-11ea-b1d1-3148925e06e7	35	Russia	Petr	Petrov

Через 15 секунд значения возраста изменится на null:

id	age	citizenship	first_name	last_name
3a26e100-9aeb-11ea-b1d1-3148925e06e7	null	Russia	Ivan	Ivanov
6d0317a0-9aec-11ea-b1d1-3148925e06e7	22	France	test	Petrov
47957270-9aeb-11ea-b1d1-3148925e06e7	35	Russia	Petr	Petrov

Группировка и сортировка данных

Как и в SQL, группировка и сортировка осуществляется с помощью ключевых слов GROUP BY и ORDER BY.

GROUP BY используется для группировки данных, ORDER BY для их сортировки.
Отсортируем курсы по цене:
SELECT * FROM course WHERE title='test3' ORDER BY price;

```
cqlsh:lab7> SELECT * FROM course WHERE title='test3' ORDER BY price;

 title | price | description
-------+-------+-------------
 test3 |  1000 |       test3
 test3 |  2000 |       test3
 test3 |  4000 |       test3
```

Выберем список курсов и для каждого из них укажем максимальную стоимость:
SELECT title, MAX(price) FROM course GROUP BY title;

```
cqlsh:lab7> select title, max(price) from course group by title;

 title | system.max(price)
-------+-------------------
 test3 |              4000
 test2 |              1000
 test1 |              1000
 test5 |              2001

(4 rows)
```

Вы можете удалить таблицу с помощью команды TRUNCATE. Когда вы усекаете таблицу, все строки таблицы удаляются навсегда. Ниже приведен синтаксис этой команды.
TRUNCATE <имя таблицы>
Например, TRUNCATE course;

Удалить индекс можно следующей командой:
DROP INDEX indexname;

Материализованное представление

В Cassandra все операции записи данных - это всегда операции перезаписи, то есть, если в колоночную семью приходит колонка с таким же ключом и именем, которые уже существуют, и метка времени больше, чем та которая сохранена, то значение

перезаписывается. Записанные значения никогда не меняются, просто приходят более новые колонки с новыми значениями.

Запись в Cassandra работает с большей скоростью, чем чтение. Это меняет подход, который применяется при проектировании. Если рассматривать Cassandra с точки зрения проектирования модели данных, то проще представить колоночное семейство не как таблицу, а как *материализованное представление* (materialized view) — структуру, которая представляет данные некоторого сложного запроса, но хранит их на диске. Вместо того, чтобы пытаться как-либо скомпоновать данные при помощи запросов, лучше постараться сохранить в колоночное семейство все, что может понадобиться для этого запроса. То есть, подходить необходимо не со стороны отношений между сущностями или связями между объектами, а со стороны запросов: какие поля требуются выбрать; в каком порядке должны идти записи; какие данные, связанные с основными, должны запрашиваться совместно — всё это должно уже быть сохранено в колоночное семейство.

Более подробно про параметры материализованного представления Вы можете прочитать в официальной документации [4]

Синтаксис создания материализованного представления:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] view_name AS
                    select_statement
                    PRIMARY KEY '(' primary_key ')'
```

view_name – название материализованного представления
select_statement – запрос, на основе которого создается материализованное представление
primary_key – первичный ключ для материализованного представления
Ограничения для материализованных представлений:
Все первичные ключи исходной таблицы включаются в первичный ключ материализованного представления.
Только один новый столбец может быть добавлен к первичному ключу материализованного представления.
Создадим материализованное представление на основе запроса из семейства колонок course:
CREATE MATERIALIZED VIEW course_price AS SELECT * FROM course WHERE title IS NOT NULL AND price IS NOT NULL PRIMARY KEY(title,price) ORDER BY (price desc);
PRIMARY KEY указывает, какие атрибуты из course использовать в качестве ключевых.

Будут выбраны записи, в которых title и price не NULL, они будут отсортированы по полю price.
Теперь оно сохранено на диск и мы можем к нему обратиться:
SELECT * FROM course_price;

```
cqlsh:lab7> SELECT * FROM course_price;

 title | price | description
-------+-------+-------------
 test3 |  4000 |       test3
 test3 |  2000 |       test3
 test3 |  1000 |       test3
 test2 |  1000 |       test2
 test1 |  1000 |       test1
 test5 |  2000 | conditional
```

BATCH

Объединяет несколько операторов языка изменения данных (DML) (таких как INSERT, UPDATE и DELETE) для достижения атомарности и изоляции. Позволяет выполнять запросы «пачками».

Синтаксис использования ключевого слова BATCH следующий:

BEGIN [(**UNLOGGED** | **COUNTER**)] **BATCH**

[**USING TIMESTAMP** [***epoch_microseconds***]]

dml_statement [**USING TIMESTAMP** [***epoch_microseconds***]] ;

[***dml_statement*** [USING TIMESTAMP [***epoch_microseconds***]] [; ...]]

APPLY BATCH ;

dml_statement – dml оператор

BATCH может содержать данные dml операторы:

INSERT

UPDATE

DELETE

Параметры:

UNLOGGED | COUNTER:

Если UNLOGGED не указан, будет создан журнал операций. Если задействовано несколько разделов, пакеты регистрируются по умолчанию. Зарегистрированная партия гарантирует, что все или ни одна из пакетных операций будут выполнены успешно (атомарность). Ведение журнала снижает производительность, журнал записывается на два других узла.

Опция COUNTER используется для пакетных обновлений типа счетчик.

USING TIMESTAMP:

Устанавливает время записи для транзакций, выполненных в BATCH.

Ограничение: USING TIMESTAMP не поддерживает LWT (облегченные транзакции), такие как операторы DML, которые имеют предложение IF NOT EXISTS.

Пример:
BEGIN BATCH INSERT INTO course (title, price, description) VALUES ('test5', 2000, 'test5'); UPDATE course SET description='update1' WHERE title='test5' AND price=2000; APPLY BATCH;
Данный запрос вставит значения в поля title, price, description, а после этого обновит поле description.

Условия IF для Update и Delete

Ключевое слово UPDATE работает следующим образом: если значения, удовлетворяющего условию не существует, оно будет создано.

Так, следующий запрос создаст новую запись:
UPDATE course SET description='conditional' WHERE title='test5'
AND price=2001;

```
cqlsh:lab7> select * from course;

 title | price | description
-------+-------+-------------
 test3 |  1000 |       test3
 test3 |  2000 |       test3
 test3 |  4000 |       test3
 test2 |  1000 |       test2
 test1 |  1000 |       test1
 test5 |  2000 | conditional
 test5 |  2001 | conditional
```

Чтобы избежать такого поведения, можно использовать IF EXISTS
UPDATE course SET description='conditional' WHERE title='test6'
AND price=2000 IF EXISTS;

```
 [applied]
-----------
      True
```

Значения будут обновлены. В случае отсутствия данной записи запрос вернет false и новая запись не будет создана.
Так же данная конструкция позволяет выиграть в производительности, если записей не существует.

DELETE так же можно использовать с IF

DELETE FROM course WHERE title='test5'
AND price=2000 IF EXISTS;

В IF может содержаться любое условие, например:
DELETE FROM employees WHERE department_id = 2 AND employee_id = 1 IF name = 'Joe';

Вопросы для самопроверки:

1. Особенности, модель данных и функциональные возможности колоночных СУБД?
2. Что такое пространство ключей, семейство столбцов, строка, столбец и суперстолбец?
3. Что такое ключи и индексы? Как их объявлять и использовать?
4. Функциональные возможности и языки запросов для СУБД Cassandra?
5. Типы данных и TTL в СУБД Cassandra?
6. Возможности и конструкции для определения данных, работы с данными и запросами в СУБД Cassandra?
7. Технология репликации и фрагментации в СУБД Cassandra? Как организован и работает кластер?
8. Уровни согласованности данных в СУБД Cassandra?
9. Восстановление данных в СУБД Cassandra?
10. Запись на диск и уплотнение данных в СУБД Cassandra?
11. Поддержка транзакций в СУБД Cassandra?
12. Что такое материализованное представление и как с ним работать?

Литература:

1. Фаулер, Мартин, Садаладж, Прамодкумар Дж. NoSQL: новая методология разработки нереляционных баз данных. : Пер. с англ. - М.: ООО "И.Д. Вильямс", 2013г.
2. Materialized View - https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlCreateMaterializedView.html (дата обращения 01.06.20)
3. CREATE KEYSPACE - https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlCreateKeyspace.html (дата обращения 01.06.20)
4. Time-to-live - https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useTTL.html (дата обращения 01.06.20)
5. A deep look at the CQL WHERE clause - https://www.datastax.com/blog/2015/06/deep-look-cql-where-clause (дата обращения 01.06.20)