САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ

Мишенин А.Н., Блеканов И.С., Стученков А.Б.

Распределенные приложения и обработка данных

Apache ZooKeeper. Apache Hive.

Учебно-методическое пособие

Санкт-Петербург
2022

УДК 025.4.03 : 004.738.5
ББК 32.971.353 : 32.973

А.Н. Мишенин, И.С. Блеканов, А.Б. Стученков
Распределенные приложения и обработка данных. Apache ZooKeeper. Apache Hive.
Учебно-методическое пособие – СПб., 2022. – 44.

Учебно-методическое пособие разработано на основе опыта чтения курсов лекций и проведения практических занятий по учебным дисциплинам «Hadoop» и «Теоря и практика больших данных» на факультете Прикладной Математики – Процессов Управления Санкт-Петербургского Государственного Университета. Данная часть посвящена знакомству с распределенным координационным сервисом Apache ZooKeeper и системой для управления и обработки данных Apache Hive.

Содержание

Введение 3

Apache Zookeeper 4

Архитектура Apache Zookeeper . . . 4
ZNode . . . 5
Сессия . . . 8
Watches . . . 9
Запуск песочницы . . . 11
Работа с клиентом . . . 12
Создание ZNodes . . . 12
Наблюдения . . . 14
Транзакции . . . 14
Блокировки . . . 15
Election . . . 16
Очереди . . . 17
Счетчики . . . 18

Apache Hive 20

Архитектура . . . 21
Запуск песочницы . . . 25
Клиент для Python . . . 25
Типы данных . . . 27
Численные типа . . . 27
Date/Time . . . 27
Строковые типы . . . 27
Логические и двоичные типы . . . 28
Структурные типы . . . 28
Создание таблиц и баз данных . . . 28

Partitioning 33
Skewed tables 34
Bucketed table 35
Загрузка и изменение данных 35
Views 36
Индексы 37
Joins 37
UDF 38

Задачи **40**
Задача №1 40
Задача №2 41
Задача №3 41

Список литературы **44**

Введение

В данном методическом пособии изложен ориентированный на практику обзор ряда программных продуктов, активно использующихся в области обработки больших данных и написания распределённых приложений. Знакомство с основными идеями, которые заложены в архитектуре, значительно расширит кругозор и поможет лучше ориентироваться в принципах построения сложных распределенных приложений, нацеленных на обработку данных.

Для успешного освоения курса необходимо:

- владеть языком программирования Python
- уметь работать в командной оболочке
- знать язык SQL
- знать устройство `Hadoop`, а именно `HDFS` и `MapReduce`
- иметь представление об синхронизации процессов, устройстве распределенных систем.

Apache Zookeeper

`Apache Zookeeper` [4] - это распределенный координационный сервис для распределенных приложений, предназначенный для управления конфигурациями, синхронизацией, обмена вспомогательными данными между узлами. `ZooKeeper` можно рассматривать как некоторый аналог примитивов синхронизации, которые используются при написании многопоточных приложений.

Позволяет распределенным приложениям координироваться друг с другом, образуя некоторую иерархическую древовидную структуру, наподобие обычной файловой системе, к каждому узлу которой можно присвоить некоторое значение или семантику. Данные хранятся в памяти, что позволяет достигать высокой производительности и минимизировать задержки.

Основные особенности реализации:

- упор на высокой производительности и масштабируемости
- надежность и отказоустойчивость
- гарантируется строгая последовательность операций записи и чтения, что позволяет реализовать сложные алгоритмы синхронизации на стороне клиента.

Архитектура Apache Zookeeper

`ZooKeeper` - распределенная система, которая может функционировать на наборе серверов, называемом ансамблем. Эти сервера поддерживают связь друг с другом таким образом, чтобы поддерживать в памяти текущее состояние хранилища, логи транзакций и

снимки состояния хранилища, которые хранятся на диске. Кластер `ZooKeeper` может оставаться в работоспособном состоянии при отказе половины серверов.

Каждый клиент соединяется с одним сервером и поддерживает TCP соединение, по которому отправляются запросы, получаются ответы и уведомления о событиях, на которые подписан клиент. Также клиент получает информацию о работоспособности самого `ZooKeeper` (через так называемые *heartbeats*).

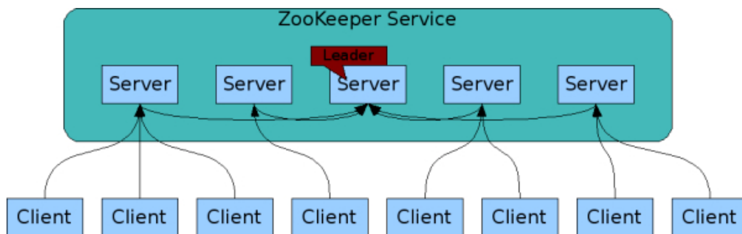

Рис. 1: Использование Zookeeper, источник: [4]

ZNode

Как говорилось выше, логически модель данных `ZooKeeper` похожа на файловую систему, точнее на распределенную файловую систему. Ключевое отличие - каждый узел файловой системы («файл» или «каталог» в классической файловой системе) может иметь какие-то ассоциированные с ним данные и дочерние узлы. Это можно обрисовать как файловую систему, где файлы и каталоги совмещены. Пути к узлам всегда записываются как абсолютные (в стиле `UNIX`), нормализованные, разделенные косой чертой. В качестве имени узлов может

быть использована любая последовательность символов `Unicode` со следующими ограничениями:

- символ **U+0000**, **U+0001**-**U+001F**, **U+D800**-**U+F8FF**, **U+FFF0**-**U+FFFF**, **U+007F**-**U+009F** не могут быть использованы
- символ «.» могут быть использованы в имени, но «.» и «..» не могут быть именем узла. Следующие пути некорректны: **/a/./c** или **/a/../c**
- слово «zookeeper» зарезервировано

Каждый узел в дереве `ZooKeeper` называется «znode». «Znode» имеет обязательные атрибуты, например число версий, параметры доступа, время создания. Номер версии совместно с временем создания позволяет держать кэш в валидном состоянии и координировать изменения. Каждый раз, когда значение, ассоциируемое с узлом, меняется, номер версии автоматически увеличивается. Когда клиент получает значение какого-то узла, он также получает номер версии. Когда клиент производит удаление или изменение данные, он может предоставить номер версии, которую он изменяет. Если реальные версии на сервере не совпадают, то изменение или удаление не состоится.

Клиенты могут запрашивать слежение на узлами. Каждое изменение узла вызывает событие, уведомление о котором посылается клиенту.

Важно заметить, что `ZooKeeper` не предназначен для хранения больших объемов данных, или вообще для хранения данных из бизнес-логики приложения. Данные, которые хранятся в `ZooKeeper` имеют другую природы - это координация между компонентами распределенного приложения, конфигурация, и т.д. Основная характеристика подобного рода данных - их относительно маленький

размер. Если же возникла необходимость хранить большой объем данные, то рекомендовано хранить в `ZooKeeper` льшь ссылку, а для самих данных использовать более подходящее хранилище (`HDFS`[5], `Amazon S3`, и т.д.)

Данные, хранящиеся в каждом узле, читаются и записываются атомарно. Каждый узел имеет список управления доступом (ACL), что позволяет огриничивать доступ какому-нибудь клиенту.

`ZooKeeper` может оперировать специальным типом узлов, которые называются эфемерными. Эти узлы существуют в течение жизни клиентской сессии, в рамках которой они были созданы. Когда сессия завершается, эфемерные узлы удаляются. Из-за подобного поведения эфемерные узлы не могут иметь дочерних узлов.

Также существует специальный тип узлов - контейнеры. Они имеют некоторые особенности поведения - когда всех их дочерние узлы удалены, `ZooKeeper` имеет права удалить такой узел. Помимо этого для некоторых узлов можно установить время жизни. Если в течение заданного времени они не изменялись, то `ZooKeeper` имеет право их удалить.

`ZooKeeper` оперирует временем временем несколькими способами:

1. `Zxid`. Каждый раз, когда в данных что-то изменяется, это изменение получает уникальный идентификатор - а `zxid` (ZooKeeper Transaction Id). Это счетчик, исли значение одного изменения меньше, чем другого, то гарантируется, что первое изменение произошло до второго.
2. Номер версии. Используется в качестве аттрибута узлов, каждый раз, когда значение узла меняется, его номер версии увеличивается

Как говорилось ранее, каждый узел хранит обязательные метаданные:

- `czxid`. `zxid` изменения, когда этот узел был создан
- `mzxid`. `zxid` последнего изменения узла
- `pzxid`. `zxid` последнего изменения дочернего узла
- `ctime`. Время после создания узла
- `mtime`. Время после последней модификации узла
- `version`. Номер версии узла
- `cversion`. Число изменений дочерних узлов
- `aversion`. Число изменений в списке управления доступом (ACL) узла
- `ephemeralOwner`. Идентификатор сессии, если это эфемерный узел
- `dataLength`. Размер данных, ассоциированных с этим узлом
- `numChildren`. Число дочерних узлов.

Сессия

Соединение клиента с сервером `ZooKeeper` называется сессией. Сессия может иметь определенные состояния:

- `CONNECTING` - клиенты пытается установить физическое соединение с одним из серверов`ZooKeeper`
- `CONNECTED` - соединение установлено, пройдена авторизация
- `CLOSE` - сессия закрыта (в ручную или по иной причине, например ошибка авторизации)

Для создания сессии клиенту предоставляется строка с разделенным запятой парами `host:port`, каждая пара соответствует какому-то серверу в кластере. Клиентская библиотека выбирает какой-то сервер

и пытается установить с ним соединение. Если соединение установить не удалось, клиент пробует следующий сервер. Каждая пара может иметь дополнительный параметр, который позволит клиенту работать с каким-то существующим узлом в качестве корня дерева.

Каждая сессия представляется 64 битным числом, который назначается клиенту. Если клиент вынужден соединиться с другим сервером `ZooKeeper`, то он передает существующий номер сессии. Помимо этого, из соображений безопасности, каждая сессия имеет какой-то секретный ключ, который используется, вместе с идентификатором.

Сессия существует, пока клиент шлет запросы. Если соединение не используется какое-то время, клиент обязан послать специальный запрос (`PING`), для того чтобы поддержать сессию в активном состоянии. Подобные запросы также помогают клиенту удостовериться, что выбранный сервер активен.

Watches

Все операции чтения на узлом (получить данные, получить список дочерних узлов, проверить, что узел существует) имеют возможность установить «наблюдение» (watch). Это единовременное уведомление, которые посылается клиенту, возникающее при изменении данных для которых клиент запросил данное уведомление. Тут есть три основных момента:

1. Уведомление является единовременным, оно посылается клиенту, когда данные изменяются. Например, если клиент запросил наблюдение за узлом **/znode** и этот узел изменился или удален, клиент получит уведомление. Если узел снова изменился, то уведомление получено не будет.

2. `ZooKeeper` гарантирует упорядоченность - клиент никогда не увидит изменение, для которого ему послано уведомление, перед тем, как это уведомление до него дойдет и будет обработано. Ключевой момент - все изменения, видимые разными клиентами, имеют определенный порядок.
3. Физически внутри сервера есть два различных списка спика наблюдения - за данными узла и за дочерними узлами.

Списки наблюдений поддерживаются локально тем сервером `ZooKeeper`, к которому клиент присоединен. Технически это позволяет дешево реализовывать этот механизм. Если клиент присоединяется к другому серверу, то его список наблюдений автоматически перерегистрируется, это происходит прозрачно для конечного пользователя. При этом есть один тонкий момент - если во время пересоединения узел создастся и потом удалится, то клиент не получит должного уведомления.

Можно установить наблюдения на три действия, которые получают состояние `ZooKeeper`: проверка существования узла (`exists`), получение данных узла (`getData`) и получение списка конечных узлов: (`getChildren`).

События могут быть следующих видов:

- `Created event` - создание узла
- `Deleted event` - удаление узла
- `Changed event` - изменение данных
- `Child event` - изменение в дочерних узлах

При реализации наблюдений, `ZooKeeper` гарантирует:

- наблюдения упорядочены по отношению к другим событиям. Клиентские библиотеки гарантируют, что события будут

обрабатываться в порядке из поступлениян order.
- клиент получит событие об изменении данных до того, как сможет получить новую версию данных
- `ZooKeeper` отправляет события в том порядке, в котором они произошли

При этом важно помнить:

- стандартные наблюдения срабатывают только один раз. Потому между получением уведомления о событии и постановкой нового наблюдения есть задержка, во время которой значение узла могло измениться
- клиент не может установить ещё одно наблюдение на объект, который он ставил наблюдение до этого. Событие будет передано один раз
- когда клиент отсоединяется от сервера, то до того как он соединится вновь события приходить не будут.

Запуск песочницы

Для удобства мы будем использовать контейнерезированный мини-кластер `ZooKeeper`, который используется в одном образе с распределённым брокером сообщений `Kafka` [2]. Некоторые версии Kafka использует `ZooKeeper` для своих нужд.

Запускаем его в `Docker` вместе с `Kafka`:

```
docker run --name kafka_sandbox -p 2181:2181 -p 9092:9092
    spotify/kafka
```

Этого достаточно для всех экспериментов в данной части. `ZooKeeper`

написан на Java, тем не менее есть клиентские библиотеки для многих языков. Мы будем использовать Kazoo[6] для Python.

Для установки можно использовать pip

```
pip install —U kazoo
```

Работа с клиентом

Подключаемся к ZooKeeper, в списке *hosts* можно указать несколько серверов:

```
1  from kazoo.client import KazooClient
2
3  zk = KazooClient(hosts='127.0.0.1:2181')
4  zk.start()
```

Создание ZNodes

Воспользуемся функцией для создания узла.

```
1  # рекурсивно создает пустые узлы
2  zk.ensure_path('/node/a')
3  # создает узел (если родительский узел существует)
4  # и устанавливает значение
5  zk.set('/node/a', b'hello')
```

```
ZnodeStat(
    czxid=29, mzxid=30,
    ctime=1636632798620,
    mtime=1636632798622,
    version=1, cversion=0,
    aversion=0, ephemeralOwner=0,
    dataLength=5, numChildren=0, pzxid=29
)
```

Можно создать ещё один узел:

```
1  zk.ensure_path('/database2/host')
2  zk.set('/database2/host', b'192.168.1.2')
```

```
ZnodeStat(
    czxid=32, mzxid=33,
    ctime=1636632800059,
    mtime=1636632800062, version=1,
    cversion=0, aversion=0, ephemeralOwner=0,
    dataLength=11, numChildren=0, pzxid=32
)
```

Получим значение узла:

```
1  zk.get('/database2/host')
```

```
(b'192.168.1.2',
 ZnodeStat(
     czxid=32, mzxid=33, ctime=1636632800059,
     mtime=1636632800062, version=1, cversion=0,
     aversion=0, ephemeralOwner=0, dataLength=11,
     numChildren=0, pzxid=32)
)
```

можно иначе, воспользовавшись функцией `create`

```
1  zk.create('/node/b', b'hello2')
```

```
'/node/b'
```

Наблюдения

В Python механизм наблюдений реализован с помощью функций обратного вызова:

```
def callback(p):
    print('Event detected: ', p)

zk.get('/node/a', callback)

# после выполнения операции будет выброшено событие
zk.set('/node/a', b'hello2')
```

```
Event detected:  WatchedEvent(type='CHANGED', state='
    CONNECTED', path='/node/a')
```

`set` возвращает информацию об измененном узле

```
ZnodeStat(
    czxid=29, mzxid=35,
    ctime=1636632798620,
    mtime=1636632803567,
    version=2, cversion=0,
    aversion=0, ephemeralOwner=0, dataLength=6,
    numChildren=0, pzxid=29
)
```

Транзакции

В kazoo реализован механизм транзакций - изменять данные можно в рамках одной транзакции. Если какие-то действия не будут выполнены до завершения транзакции, то все предыдущии операции отменяются.

```
with zk.transaction() as t:
    t.create('/node/c', b'c value')
    t.create('/node/d', b'd value')

# получаем значение узла
zk.get('/node/c')
```

```
(b'c value',
 ZnodeStat(
    czxid=36, mzxid=36,
    ctime=1636632805734,
    mtime=1636632805734,
    version=0, cversion=0,
    aversion=0, ephemeralOwner=0,
    dataLength=7, numChildren=0,
    pzxid=36)
)
```

Блокировки

В kazoo реализован механизм блокировок, на подобие хорошо известных примитивов синхронизации.

```python
import threading
import time

def thread_func(num):
    # блокировка привязывается к какому-нибудь узлу
    with zk.Lock('/node/a') as lock:
        # блокировка взята
        print('Thread #{} lock'.format(num))

        time.sleep(2)

        print('Thread #{} unlock'.format(num))

# создаем три потока
for i in range(3):
    threading.Thread(target=thread_func, args=(i,)).
        start()
```

```
Thread #0 lock
Thread #0 unlock
Thread #1 lock
Thread #1 unlock
Thread #2 lock
```

Election

Выборы - важных механизм, который позволяет выполнять определенные действия только одному узлу распределенного приложения. Как только по какой-т опричине узел стал недоступен, его место занимает кто-то другой.

```
1  def thread_func(num):
2      # объект выборов
3      election = zk.Election("/node")
4
5      # функция, которая вызовется при
6      # выигрыше "выборов"
7      def election_func():
8          print('Election won: {}'.format(num))
9
10     # участвуем в выборах
11     election.run(election_func)
12
13
14 for i in range(3):
15     # запускаем три потока
16     threading.Thread(target=thread_func, args=(i,)).
           start()
```

```
Election won: 0
Election won: 2
Election won: 1
Thread #2 unlock
```

Очереди

kazoo позволяет сопоставлять каждому узлу структуры данных, например - очередь с приоритетами, которая будет доступна всем клиентам.

```
# удаляем узел если он существует
if zk.exists('/queue'):
    zk.delete('/queue', recursive=True)

# создаем очередь
q = zk.Queue('/queue')
# добавляем элемент с приоритетом `10`
q.put(b'1', 10)
# добавляем элемент с приоритетом `0`
q.put(b'2', 0)
```

```
# получаем элемент с меньшим приоритетом
q.get()
```

```
b'2'
```

Счетчики

kazoo предоставляет распределенные счетчики, которые могут быть использованы для атомарного подсчета чего либо в рамках работы распределенного приложения

```
if zk.exists('/counter'):
    zk.delete('/counter', recursive=True)

# привязываем объект счетчика к узлу
counter = zk.Counter('/counter')
# увеличиваем на 20
counter += 20
# уменьшаем на 5
counter -= 5
# получаем значение
counter.value
```

```
15
```

Дальнейшее изменение счетчика:

```
1  counter += 100
```

Значение счетчика можно получить так:

```
1  zk.get('/counter')
```

```
(b'115',
 ZnodeStat(
     czxid=57, mzxid=60,
     ctime=1636632817262,
     mtime=1636632817398,
     version=3, cversion=0, aversion=0,
     ephemeralOwner=0, dataLength=3,
     numChildren=0, pzxid=57)
)
```

Apache Hive

`Apache Hive` [1] - это система для управления и обработки данных, которая работает поверх различных продуктов из экосистемы `Hadoop` (обычно `MapReduce`), упрощает обработку данных и предоставляет SQL-подобный язык запросов.

Появление `Hive` [1] в начала 2010-х мотивировано следующими соображениями:

- написание Hadoop-задач на Java и других языках нетривиально, доступно только профессиональным разработчикам, а не аналитикам данных, основным инструментом для которых является язык `SQL`
- стандартные реляционные СУБД изначально предназначались для других задач и не справлялись огромным объемом данных. При этом есть была необходимость использовать Hadoop для обработки такого объема данных
- нужен был инструмент для решения т.н. ETL задач (извлечение, преобразование, загрузка)
- появилась необходимость иметь определенную схему для данных, хранящихся на `HFDS`[5]
- при использовании `MapReduce` данные могли храниться в различных форматах (sequence file, csv, parquet, и так далее). Нужен механизм для унификации, чтобы конечных пользователь не думал о том, как работать с тем или иным форматом.

Архитектура

`Hive` обычно функционирует в рамках `Hadoop`-кластера и состоит из различных компонент (как физических, так и логических). Конечному пользователю предоставляется пользовательский интерфейс, интерфейс командной строки и API для доступа, например `JDBC`. Пользователь может использовать SQL-подобный язык запросов (`HiveQL`) для создания баз данных, таблиц, манипулирования с данными. Физически сами данные хранятся, чаще всего, в виде обычных файлов на `HDFS`. Последнее накладывает ограничения на сценарии использования `Hive`:

- операции изменения данных неэффективны
- индексы и `ACID`-транзакции имеют свои особенности (последние появились в поздних версиях и при реализации активно используется `Apache ZooKeeper` [4])
- сложно использовать в realtime сценариях
- предназначены для очень существенных объемов данных

Metastore

`Metastore` - это внутренняя база данных, которую `Hive` использует для хранения метаинформации. Содержит:

- данные о базах данных
- данные о таблицах, колонках
- информацию о том, где (например на `HDFS`) и в каких форматах хранятся таблицы

`Metastore` хранит метаданные для следующих объектов:

- `Database` (база данных) - это пространство имен для таблиц. По умолчанию, если пользователь во время выполнения запроса

не предоставил имя базы данных, используется база данных по умолчанию - `default`.
- `Table` (таблица) - содержит список колонок, владельца, спецификацию хранилища (например, путь к файлу в `HDFS`), информацию о формате данных. Все это должно быть предоставлено во время создания таблицы.
- `Partition` - данные таблицы могут быть разделены по какому-то критерию, каждая часть иметь собственный формат и хранится иначе. За хранение этой мета-информации отвечает `Partition`. Подобное разделение позволяет системе хранить данные более оптимальным образом, например все данные за определенную дату могу располагаться в отдельном каталоге, что ускоряет выполнение запросов, которые зависят от даты.
- `Buckets` - данные в рамках `Partition` могут быть, в свою очередь, разделены на `Buckets` по хэш-значению какого-то поля и физически хранится в разных файлах. Это нужно для ускорения некоторых аналитических запросов.

Физически в качестве хранилища может использоваться реляционная БД, в том числе встраиваемая. Основная мотивация для хранения информации в реляционной базе данных - легкость и универсальность доступа. Данные могли бы хранится на `HDFS`, но тут есть врожденные недостаток - высокие задержки и невозможность реализовать произвольный доступ к данным. База данных для хранения метаданных может быть удаленной или, как говорилось ранее, встраиваемой. В случае удаленной БД необходимо поддерживать работоспособность ещё одной системы, которую нужно администрировать, в том числе налаживать систему мониторинга.

Driver

`Driver`- компонент, который обслуживает жизненный цикл запроса к `Hive`. Этот комплемент реализует логику пользовательской сессии, предоставляет пользовательское API для выполнения запросов, в том числе с помощью интерфейсов `JDBC` и `ODBC` .

Query Compiler

`Query Compiler` - компонент, который обрабатывает запрос на `HiveSQL`, анализирует его, составляет план выполнения (с помощью данных, которые хранятся в `metastore`) и преобразует его в последовательность MapReduce-задач

Execution Engine

`Execution Engine` - компонент, который непосредственно запускает задачи, например с помощью *Hadoop MapReduce*. Подзадачи представляются в виде направленного графа без циклов, `Execution Engine` управляет зависимостями между подзадачами и выполняет их, используя необходимые компоненты.

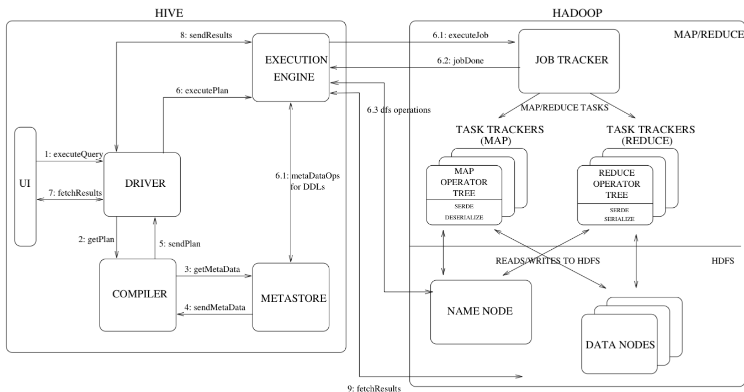

Рис. 2: Архитектура Hive, источник: [3]

Рисунок демонстрирует жизненный цикл запроса к системе. Пользовательских запрос из интерфейса попадает в `Driver` (шаг 1). `Driver` создает сессию для обработки запроса и посылает запрос `Query Compiler` для создания плана выполнения (шаг 2). `Query Compiler` получает необходимые метаданные из `metastore` (шаги 3 и 4). Эти метаданные используются для проверки типов. Полученный план (шаг 5) - это направленный граф без циклов, где каждая вершина - действие в виде задача MapReduce, операции над метаданными или операция с `HDFS`. `Execution Engine` запускает задачи (шаги 6, 6.1, 6.2 и 6.4). Далее конечный результат записывается на `HDFS` (шаги 7, 8 и 9).

Запуск песочницы

Для экспериментом можно использовать контейнеризированный Hadoop-кластер с Hive, который можно запустить следующим образом:

```
docker run -d --hostname=quickstart \
    --name=hadoop-sandbox \
    --privileged=true \
    -p 50070:50070 \
    -p 14000:14000 \
    -p 8088:8088 \
    -p 19888:19888 \
    -p 9083:9083 \
    -p 10000:10000 \
    -p 10002:10002  \
    bigtop/sandbox:1.2.1-ubuntu-16.04-hdfs_yarn_hive_pig
```

Клиент для Python

Для работы с Hive можно использовать клиент для языка Python, который называется PyHive[7]. Установить клиент можно с помощью pip

```
pip install PyHive[hive]
```

для работы может понадобиться библиотека libsasl2, на Debian-подобных системах её можно установить следующим образом:

```
sudo apt install libsasl2-dev
```

Теперь можно попробовать присоединиться к песочнице и создать таблицу:

```
from pyhive import hive

# присоединяемся к Hive
with hive.connect('localhost', username='root') as
     conn:
    # создаем таблицу
    with conn.cursor() as cursor:
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS person (
                          n STRING,
                          age INT)
            ''')
    # добавляем данные
    with conn.cursor() as cursor:
        cursor.execute('''
            INSERT INTO TABLE person VALUES ("John",
                25), ("Mike", 50), ("Mike", 30)
        ''')

    # выполняем SELECT
    with conn.cursor() as cursor:
        cursor.execute('''
            SELECT n, count(*) FROM person group by n
        ''')

        # печатаем результаты
        print(cursor.fetchall())
```

Результат ожидаем:

```
[('Mike', 2), ('John1', 1)]
```

Сразу можно заметить, что эти простые запросы работают очень долго, инициируется запуск `MarReduce` задач, что отличает `Hive` от классический реляционных СУБД.

Типы данных

`Hive` предоставляет большой набор типов данных, которые в определенном смысле расширяют возможности `SQL`

Численные типа

- `TINYINT` - 1 байтовое знаковое целое
- `SMALLINT` - 2 байтовое знаковое целое
- `INT`/`INTEGER` - 2 байтовое знаковое целое
- `BIGINT` - 2 байтовое знаковое целое
- `FLOAT` - 4 байтовое число с плавающей точкой
- `DOUBLE` - 8 байтовое число с плавающей точкой
- `DECIMAL` - десятичная дробь произвольной точности
- `NUMERIC` - аналогично `DECIMAL`

Date/Time

- `TIMESTAMP` - UNIX timestamp, может быть получено из строки
- `DATE` - задает конкретный день/месяц/годы в видео ГГГГ-ММ-ДД, например, *2021-01-01*.

- `INTERVAL` - задает временной интервал (число минут, секунд, дней и так далее)

Строковые типы

- `STRING` - строки произвольной длины

- `VARCHAR` - строки с заданной максимальной длинной
- `CHAR` - строки строго фиксированной длины

Логические и двоичные типы

- `BOOLEAN` - булевский тип
- `BINARY` - двоичные данные

Структурные типы

В `Hive` могут использоваться составные структуры данных:

- `arrays: ARRAY<data_type>` - массив
- `maps: MAP<primitive_type, data_type>` - словарь
- `structs: STRUCT<col_name : data_type [COMMENT col_comment ], ...>` - структура данных
- `union: UNIONTYPE<data_type, data_type, ...>` - объединение, все данные накладываются друг на друга и интерпретируются в зависимости от контекста

Создание таблиц и баз данных

Очень упрощенно, база данных представляет собой пространство имен с метаинформацией. Для создания баз данных используется следующий синтаксис:

```
CREATE DATABASE test_db;
```

База данных может быть удалена с помощью команды:

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [
    RESTRICT|CASCADE];
```

может быть изменена:

```
ALTER (DATABASE|SCHEMA) database_name SET
    DBPROPERTIES (property_name=property_value, ...);
```

может быть выбрана в качестве БД по умолчанию:

```
USE CIK;
SHOW DATABASES;
```

Таблицы в `Hive`:

- являются аналогами таблиц в реляционных БД
- каждая таблица обычно хранится в какой-то директории на `HDFS`
- каждая таблица имеет соответствующую метаинформацию в `Metastore`
- имя таблиц и колонок регистронезависимы, но свойства таблицы являются регистрозависимыми
- каждая таблица может иметь ассоциируемый `SerDe` (от англ. serialization/deserialization), расширяемый механизм, с помощью которого `Hive` понимает как преобразовывать данные, хранящиеся на `HDFS` в конкретные значения в соответствии со спецификацией таблицы.

Таблицы очень часто создаются из файлов на `HDFS` (или даже локальных). Допустим у нас есть файл *weather.csv*, который содержит данные о погоде в Санкт-Петербурге (поля: дата, температура, давление, и пр.)

	2006-01-01 00:00:00	0.7	761.6	762.2		83.0	Ветер, дующий с ...	2.0			100%.
1	2006-01-01 03:00:00	0.7	761.2	761.8		83.0	Ветер, дующий с ...	2.0			100%.
2	2006-01-01 06:00:00	0.9	761.4	762.0		85.0	Ветер, дующий с ...	2.0			100%.
3	2006-01-01 09:00:00	1.3	762.0	762.6		87.0	Ветер, дующий с ...	2.0			70 – 80%.
4	2006-01-01 12:00:00	1.8	763.0	763.6		80.0	Ветер, дующий с ...	1.0			70 – 80%.
5	2006-01-01 15:00:00	1.8	763.3	763.9		73.0	Ветер, дующий с ...	4.0			20–30%.
6	2006-01-01 18:00:00	0.9	764.3	764.9		70.0	Ветер, дующий с ...	2.0			70 – 80%.
7	2006-01-01 21:00:00	0.9	764.9	765.5		70.0	Ветер, дующий с ...	3.0			100%.
8	2006-01-02 00:00:00	-7.8	757.0	757.6		87.0	Ветер, дующий с ...	1.0			100%.
9	2006-01-02 03:00:00	-8.8	757.6	758.2		86.0	Штиль, безветрие	0.0			90 или более, но ...
10	2006-01-02 06:00:00	-13.9	758.3	758.8		83.0	Ветер, дующий с ...	1.0			Облаков нет.
11	2006-01-02 09:00:00	-16.4	758.6	759.2		85.0	Ветер, дующий с ...	0.0			Облаков нет.
12	2006-01-02 12:00:00	-14.6	758.6	759.2		80.0	Ветер, дующий с ...	1.0			100%.
13	2006-01-02 15:00:00	-13.0	757.7	758.3		75.0	Ветер, дующий с ...	1.0			100%.
14	2006-01-02 18:00:00	-12.6	756.7	757.3		75.0	Ветер, дующий с ...	1.0			100%.
15	2006-01-02 21:00:00	-13.9	756.2	756.8		75.0	Ветер, дующий с ...	2.0			100%.
16	2006-01-03 00:00:00	-10.4	758.6	759.2		85.0	Ветер, дующий с ...	1.0			100%.
17	2006-01-03 03:00:00	-11.3	759.4	760.0		90.0	Штиль, безветрие	0.0			100%.
18	2006-01-03 06:00:00	-10.2	760.1	760.7		90.0	Штиль, безветрие	0.0			100%.
19	2006-01-03 09:00:00	-8.9	761.0	761.6		89.0	Ветер, дующий с ...	1.0			100%.
20	2006-01-03 12:00:00	-7.4	761.7	762.3		82.0	Ветер, дующий с ...	1.0			40%.
21	2006-01-03 15:00:00	-4.2	761.6	762.2		70.0	Ветер, дующий с ...	2.0			Облаков нет.
22	2006-01-03 18:00:00	-3.5	761.3	761.9		61.0	Ветер, дующий с ...	1.0			20–30%.
23	2006-01-03 21:00:00	-6.1	761.6	762.2		68.0	Ветер, дующий с ...	3.0			Облаков нет.
24	2006-01-04 00:00:00	5.2	750.9	751.5		81.0	Ветер, дующий с ...	1.0			60%.

Рис. 3: Данные метеостанции

Создать таблицу и загрузить туда данные можно с помощью запроса:

```
1  CREATE TABLE weather (
2      dt TIMESTAMP,
3      t FLOAT,
4      po FLOAT,
5      p FLOAT,
6      u FLOAT,
7      vv FLOAT,
8      td float,
9      n STRING)
10 ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
11
12 LOAD DATA LOCAL INPATH 'weather.csv' INTO TABLE
       weather;
```

В данном случае мы указываем параметры SerDe - того как данные таблицы будут храниться на HDFS: это текстовый файл, поля которого разделены запятой.

Другой пример - данные президентских выборов 2018 года:

```sql
CREATE TABLE cik (
    region STRING,
    tik STRING,
    uik STRING,
    voters_total INT,
    total INT,
    total_ahead INT,
    total_inside INT,
    total_outside INT,
    total_removed INT,
    outside INT,
    inside INT,
    invalid INT,
    valid INT,
    lost INT,
    unkwn INT,

    baburin INT,
    grudinin INT,
    zhirinovsky INT,
    putin INT,
    sobchak INT,
    suraykin INT,
    titov INT,
    yavlinsky INT
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

LOAD DATA LOCAL INPATH 'cik.csv' INTO TABLE cik;
```

Другой пример - пусть у нас есть логи web-сервера, мы хотим создать таблицу на их основе. Мы можем просто указать формат файла с помощью регулярных выражений:

```
CREATE TABLE apachelog (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" =
    "([^]*) ([^]*) ([^]*)
    (-|\\[^\\]*\\])"
    ([^ \"]*|\"[^\"]*\")
    (-|[0-9]*)
    (-|[0-9]*)(?: ([^ \"]*|\".*\") ([^ \"]*|\".*\"))?
  "
)
STORED AS TEXTFILE;
```

Для нестандартного CSV:

```
CREATE TABLE my_table(a string, b string, ...)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
   "separatorChar" = "\t",
   "quoteChar"     = "'",
   "escapeChar"    = "\\"
)
```

Таблицы могут быть: - `INTERNAL` - данными и местом хранения на `HDFS` управляет `Hive` - `EXTERNAL` - данные хранятся в каком-то ином месте на `HDFS` (задается с помощью ключевых слов `EXTERNAL` и `LOCATION`) - `TEMPORARY` - существует только в рамках сессии, данные могут хранить-

ся в памяти

Partitioning

`Partitioning` - деление таблицы на пересекающиеся и независимые части. Осуществляется для оптимизации. Физически части таблицы хранятся в разных директориях.

Например, пусть у нас есть логи пользователей вэб-сайта - время посещения, идентификатор пользователя, адрес страницы, откуда он перешел, ip-адрес. Для популярных сайтов данные приходят в огромных объемах каждый день, потому их можно хранить в разделенном виде - данные для каждой даты и страны в отдельной директории. Для этого можно описать таблицу следующим образом:

```
1  CREATE TABLE page_view(
2       viewTime INT,
3       userid BIGINT,
4       page_url STRING,
5       referrer_url STRING,
6       ip STRING COMMENT 'IP Address of the User')
7   COMMENT 'This is the page view table'
8   PARTITIONED BY(dt STRING, country STRING)
9   STORED AS SEQUENCEFILE;
```

обратите внимание, что поля *dt* и *country* отсутствуют в декларации таблицы. В этом примере каждая часть будет хранится в поддиректории с именем вида

> dt=20210607,country=RU

Это позволяет анализировать отдельны части данных (для конкретной страны или за определенную дату) быстро и независимо.

Можно использовать динамическое партиционирование, в этом примере мы перераспределяем данные в выборов в таблицу, разделенную по региону.

```
SET hive.exec.dynamic.partition.mode=nonstrict;

CREATE TABLE cik_ext(
    uik string,
    valid INT,
    putin INT,
    grudinin INT)
 partitioned BY (region string);

INSERT overwrite TABLE cik_ext partition(region)
SELECT uik , valid, putin, grudinin, region
FROM cik;
```

Skewed tables

Если данные несбалансированы, мы можем настроить таблицу так, чтобы хранить строки с определенными значениями в отдельных директориях. Мы можем указать значения, которые появляются слишком часто, это в дальнейшем позволит `Hive` оптимизировать запросы.

```
CREATE TABLE skewed_table (
    col1 STRING,
    col2 INT,
    col3 STRING
  )
  SKEWED BY (col1, col2)
  ON (('s1',1), ('s3',3), ('s13',13), ('s78',78))
  STORED AS DIRECTORIES;
```

В данном случае строки с указанными значениями *col1* и *col2* будут храниться отдельно.

Bucketed table

Таблицы могут быть разделены на части (buckets), которые определяются по хэш-значению какого-то поля. Внутри частей данные могут быть отсортированы по другому полю. Это дает следующие возможности: - возможность более эффективной ститистической обработки (например сэмплирование) - возможность оптимизировать некоторые запросы - число частей фиксировано

Нужно иметь ввиду, что описание таблицы как bucketed влияют только на то, как данные интерпретируются со стороны Hive, а не как записываются. Таким образом, при добавления данных в эти таблицы нужно быть внимательным, и использовать в запросе соответстующие выражения *CLUSTER BY* и *SORT BY*.

```
CREATE TABLE page_view(
    viewTime INT,
    userid BIGINT,
    page_url STRING,
    referrer_url STRING,
    p STRING)
 PARTITIONED BY(dt STRING, country STRING)
 CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32
     BUCKETS
 STORED AS SEQUENCEFILE;
```

Загрузка и изменение данных

Для загрузски данных в какую-то определенную партицию можно использовать запросы вида:

```
LOAD DATA LOCAL INPATH '/page_views.csv'
OVERWRITE INTO TABLE page_view
PARTITION (dt='2017—05—05', country='RU');
```

Для вставки новых данных стандартная операция INSERT расширяется с помощью ключевого слова PARTITION:

```
CREATE TABLE pageviews (
    userid VARCHAR(64),
    link STRING,
    came_from STRING
  )
  PARTITIONED BY (datestamp STRING)
  CLUSTERED BY (userid) INTO 256 BUCKETS
  STORED AS ORC;

INSERT INTO TABLE pageviews PARTITION
         (datestamp = '2017-09-21')
  VALUES ('jsmith', 'mail.com', 'sports.com'),
         ('jdoe', 'mail.com', null);

INSERT INTO TABLE pageviews PARTITION (datestamp)
  VALUES ('tjohnson', 'sports.com', 'finance.com',
          '2017-09-23'),
         ('tlee', 'finance.com', null, '2017-09-21');
```

Можно использовать существующие таблицы:

```
FROM old_pageviews
  INSERT OVERWRITE TABLE pageviews_1
      PARTITION (datestamp = '2014-09-23')
      SELECT * WHERE came_from='google.com'
  INSERT OVERWRITE TABLE pageviews_2
      PARTITION (datestamp = '2014-09-23')
      SELECT * WHERE came_from='yandex.ru';
```

Views

Принципы применения View в Hive такое же, как и в обычном SQL.

- это чисто логические объекты и не имеют ассоциированного хранилища

- когда в запросах появляется ссылка на `View`, то оно вычисляется, то есть генерируется набор колонок
- схема `View` фиксируется в момент создания
- `View` доступны только для чтения.

Создать `View` можно следующим образом:

```
CREATE VIEW onion_referrers(url)
  AS
  SELECT DISTINCT referrer_url
  FROM page_view
  WHERE page_url='http://www.example.com';
```

Индексы

Как и в случае реляционных БД, индексы служат для ускорения доступа к данным по определенным предикатам, например:

```
SELECT * FROM wheather WHERE temp=10
```

без этого `Hive` будет вынужден просканировать всю таблицу. Цена - ресурсы на создание и хранение.

Joins

`Hive` поддерживает базовые операции `JOIN`, доступные в стандартах `SQL`, например:

- `INNER JOIN`
- `LEFT OUTER JOIN`
- `FULL OUTER JOIN`

- `RIGHT OUTER JOIN`
- `LEFT SEMI JOIN`
- …

При этом существуют особенности, связанные с оптимизацией выполнения подобных операций.

```
CREATE TABLE a (k1 string, v1 string);
CREATE TABLE b (k2 string, v2 string);

SELECT k1, v1, k2, v2
FROM a JOIN b ON k1 = k2;

SELECT a.* FROM a JOIN b ON (a.id = b.id
        AND a.department = b.department)
```

UDF

`UDF` - user defined function, функция в языке SQL, которая может определятся пользователем. Нативным способом функции имплементируются на языке Java, так как сам `Hive` написан на Java. Но можно использовать скриптовые языки, в частности Python.

Реализуем `UDF`, которая принимает в вход набор колонок, оставляя только последнее слово в каждой. Будем считать, что слова разделены пробелом.

Файл *our_udf.py*

```
import sys

# оставляем только последнее слово
for line in sys.stdin:
    for val in line.strip().split('\t'):
        print(val.split(' ')[-1])
```

Запустить UDF можно в консоли следующим образом:

```
1   ADD FILE our_udf.py;
2
3   SELECT TRANSFORM(region, uik) USING 'python3 out_udf.
        py' AS (h STRING) FROM cik;
```

Задачи

Задача №1

Имеется следующий код на языке Python. Одновременно запустите несколько экзепляров этого скрипта.

```
import time

from kazoo.client import KazooClient


def election_func():
    print("Election won")
    print("Busy...")

    time.sleep(20)

    print("Done")


def main():
    zk = KazooClient(hosts='127.0.0.1:2181')
    zk.start()

    election = zk.Election("/node")
    print("Waiting for election")

    election.run(election_func)


if __name__ == "__main__":
    main()
```

В каких сценариях подобный код может применяться? Какие недостатки и потенциальные ошибки вы видите?

Задача №2

В файле [8] находятся данные о погоде в Санкт-Петербурге с 2008 по 2016 год. Это CSV-файл из двух колонок - дата и средняя дневная температура в этот день. Выполните следующие упражнения:

1. Загрузите данные в таблицу `Hive`.
2. С помощью запросов на `HiveSQL`:

- определите год, когда в январе было наибольшее число дней с положительной температурой (t ≥ 0).
- определите в каком году было самое холодное лето (по средней температуре)?
- найдите день с самой большим перепадом температуры, если сравнивать со следующим днем.

Задача №3

В файле [9] находятся данные с избиательных участков президентских выборов 2018 года. Данные сохранены в формате:

```sql
CREATE TABLE cik (
    region STRING,
    tik STRING,
    uik STRING,
    voters_total INT,
    total INT,
    total_ahead INT,
    total_inside INT,
    total_outside INT,
    total_removed INT,
    outside INT,
    inside INT,
    invalid INT,
    valid INT,
    lost INT,
    unkwn INT,

    baburin INT,
    grudinin INT,
    zhirinovsky INT,
    putin INT,
    sobchak INT,
    suraykin INT,
    titov INT,
    yavlinsky INT
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
```

С помощью `HiveSQL` и Python:

1. Найдите явку (%) по всем рвывывегионам, результат отсортировать по убыванию.
2. Выберите произвольного кандидата и найдите тот избиратльный участок, на котором он получил наибольший результат (%, учитывать участки на которых проголосовало больше 300 человек).
3. Найдите регион, где разница между ТИК с наибольшей явкой и наименьшей максимальна.

4. Найдите дисперсию по явке для каждого региона (учитывать УИК).
5. Для каждого кандидата посчитайте таблицу: результат (%, округленный до целого) - количество УИК, на которых кандидат получил данный результат.

При выполнении упражнения используйте `partitions` и `buckets`.

Список литературы

1. Apache Hive [Электронный ресурс] / (01.10.2021). Режим доступа: https://hive.apache.org/.

2. Apache Kafka [Электронный ресурс] / (01.10.2021). Режим доступа: https://kafka.apache.org/.

3. Apache Wiki [Электронный ресурс] / (01.10.2021). Режим доступа: https://cwiki.apache.org/.

4. Apache Zookeeper [Электронный ресурс] / (01.10.2021). Режим доступа: https://zookeeper.apache.org/.

5. HDFS User Guide [Электронный ресурс] / (01.10.2021). Режим доступа: https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html/.

6. kazoo [Электронный ресурс] / (01.10.2021). Режим доступа: https://kazoo.readthedocs.io/en/latest//.

7. PyHive [Электронный ресурс] / (01.10.2021). Режим доступа: https://github.com/dropbox/PyHive.

8. Данные для задачи 2 [Электронный ресурс] / (01.10.2021). Режим доступа: https://drive.google.com/file/d/1UZgDGWnzHE7vgooanymzpJPbC6YssdP9/view?usp=sharing.

9. Данные для задачи 3 [Электронный ресурс] / (01.10.2021). Режим доступа: https://drive.google.com/file/d/1d_Iczml-3d1xOZQ03vCQMmMHVHlU_FDu/view?usp=sharing.