BIGDATA

Лекция 4

ПЛАН

- Protobuf, Avro
- Parquet

BASE64

- Позволяет представлять бинарные данные в виде текста
- Оптимальнее, чем прямая печать цифр
- Режем последовательность битов на кусочки по 6
- Каждый кусочек имеет 64 разных значения
- Представим как буквы латиницы + 12 других знаков

ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ

- С JSON/CSV: представим вектор чисел
- С protobuf: представим всю запись
- Чтобы split легко находил границы

Пример описания

```
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  string number = 1;
  PhoneType type = 2;
}
```

Продолжаем

```
message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
  repeated PhoneNumber phones = 4;
  google.protobuf.Timestamp last_updated = 5;
}
```

Что с этим делаем

```
$ protoc person.proto --python_out=src

# породит что-то вроде src/person_pb2.py
```

Питончик

```python
import person_pb2

p = person_pb2.Person()
p.name = "vasya"
ph = person_pb2.PhoneNumber()
ph.number = '12345678'
p.phones.append(ph)

p.SerializeToString() # в bytes на самом деле
```

PROTOBUF: ТИПЫ ДАННЫХ

- double/float - представляется "как есть"
- int32/int64 - представляется как VarInt
- Влезающие в 7 бит займут байт
- В 14 бит - 2 байта и т.д.

PROTOBUF: ТИПЫ ДАННЫХ

- Отрицательные числа занимают много бит
- И это проблема для int32/int64
- Ее решает пара sint32/sint64
- Модуль сдвигается на бит влево, младший бит - признак отрицательности

PROTOBUF: ТИПЫ ДАННЫХ

- uint32/uint64 - беззнаковый varint
- fixed32/fixed64 - просто целое
- Эффективно при доминирующих больших значениях
- Примеры: хеши, длинные идентификаторы

PROTOBUF: ТИПЫ ДАННЫХ

- bool - один байт
- string - строка UTF-8
- bytes - байтовая последовательность

PROTOBUF: ТИПЫ ДАННЫХ

- Можно задать map как тип поля с типами-параметрами (примитивного типа)
- Поля примитивного типы могут быть optional/required/repeated
- map напрямую не может, но можно завернуть в message
- Есть конструкция oneof - полезно для комбайнера

PROTOBUF: ХРАНЕНИЕ ДАННЫХ

- Перед значением поля хранится числовой идентификатор
- Задается в исходнике
- При глубокой вложенности номеров полей становится больше

PROTOBUF: ХРАНЕНИЕ ДАННЫХ

- В начале поля повторяющегося типа хранится его длина
- Можно "перепрыгнуть" неинтересное repeated-поле
- Или один из элементов
- Сложно одним махом прыгнуть на 1000 элементов вперед
- Или прыгнуть на 100Mb и найти границу элементов

AVRO

- Тоже внеязыковое описание структур
- Структура описывается на JSON-схеме
- Порождение кода возможно как вариант
- Основной вариант - чтение/запись по приложенной схеме

Пример описания

```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number", "type": ["int", "null"]},
     {"name": "favorite_color",
      "type": ["string", "null"]}
 ]
}
```

Пример кода

```
1 import avro.schema
2 from avro.datafile import DataFileReader, DataFileWriter
3 from avro.io import DatumReader, DatumWriter
4
5 # читаем схему
6 schema = avro.schema.parse(open("user.avsc", "rb").read())
```

Пример кода

```
# записываем данные

with DataFileWriter(open("users.avro", "wb"),
                                DatumWriter(), schema) as writer:
    writer.append({"name": "Alyssa",
                   "favorite_number": 0x555})
    writer.append({"name": "Ben",
                   "favorite_number": 0x5678,
                   "favorite_color": "red"})
```

Пример кода

```
1 # читаем данные
2
3 with DataFileReader(open("users.avro", "rb"),
4                      DatumReader()) as reader:
5     for user in reader:
6         print(user)
```

ТИПЫ ДАННЫХ

- null - нужен в явном виде
- boolean, представлен байтом
- int/long - как sint32/64 в protobuf
- float/double - как есть
- bytes/string

ТИПЫ ДАННЫХ

- fixed - фиксированный размер в байтах (md5 и т.п.)
- Есть вариативный тип
- Частный случай - необязательное значение
- Стоит 1 байт на поле
- Можно потерять выигрыш от отсутствия номера поля

ЛОГИЧЕСКИЕ ТИПЫ

- Можем взять за основу один из базовых типов
- И интерпретировать его как более высокоуровневый тип
- Например: bytes можно интерпретировать как Decimal
- string - как uuid
- int - как дату

ЛОГИЧЕСКИЕ ТИПЫ

- И как время в разной точности
- И как timestamp (абстрактный или локальный)
- Облегчает процесс борьбы за экономию памяти

СОСТАВНЫЕ ТИПЫ

- Массивы: `{ "type": "array", "items" : "string", }`
- Словари - аналогично
- Ключ в словаре - только строка
- Запись - видели пример

ПРЕДСТАВЛЕНИЕ

- protobuf фокусируется на представлении структуры
- Относительно небольшой, уж точно влезающей в память
- Avro готов целиком представлять гигантскую таблицу
- И хранить массивы/словари, не влезающие в память

ПРЕДСТАВЛЕНИЕ

- Схема может быть положена в начало файла
- За ней пойдут данные
- Теги полей не хранятся
- Схема определяет перечень полей
- Нет издержек на вложенные записи

ПРЕДСТАВЛЕНИЕ

- Но возможны издержки на необязательные поля
- Они представляются как вариантый тип: int/null и т.п.
- Требуют байта на тег выбора типа

ПРЕДСТАВЛЕНИЕ

- Массивы хранятся "кусками"
- Для каждого знаем его размер
- Выбирается так, чтобы влезал в память
- Можем независимо прочитать фрагмент

ПРЕДСТАВЛЕНИЕ

- Можем широким шагом пройти по HDFS-файлу
- И поделить массив верхнего уровня примерно по границам блоков
- Или передавать куски map-задачам
- Аналогично для словарей

ВЕРТИКАЛЬНАЯ ОРГАНИЗАЦИЯ

- (Не путаем с колоночной)
- Классическая схема хранения таблиц отталкивается от строки
- Данные одной строки лежат рядом
- Запрос находит нужные строки и забирает из них нужные поля

+/- ГОРИЗОНТАЛЬНОГО ХРАНЕНИЯ

- +: соответствует интуиции
- +: удобно обрабатывать запросы с LIMIT :small:
- +: часто интерсуют разные поля одной записи и их взаимосвязи
- Но возможны и минусы

+/- ГОРИЗОНТАЛЬНОГО ХРАНЕНИЯ

- Полей в записи может быть много
- Нужна небольшая часть
- Перемещаемся по большому объему данных
- Собирая небольшую их долю

+/- ГОРИЗОНТАЛЬНОГО ХРАНЕНИЯ

- Представим, что одно поле - это рост человека
- Другое - месячный доход
- Еще одно - количество посещений страницы
- У всех свои диапазоны, шаблоны разреженности

+/- ГОРИЗОНТАЛЬНОГО ХРАНЕНИЯ

- В горизонтальном срезе они могут быть разношерстными и случайными
- А в вертикальном - очень даже регулярными
- Иногда монотонно растущими или повторяющимися
- Хотелось бы это использовать

РАДИКАЛЬНОЕ РЕШЕНИЕ

- А давайте "транспонируем"
- Будем последовательно хранить данные одной колонки
- А потом - другой
- Первый это вопрос - а как добавлять ?

РАДИКАЛЬНОЕ РЕШЕНИЕ

- Вопрос не самый трудный - можно держать по файлу на колонку
- И собрать агрегации по нескольким полям - тоже не самое сложное
- Сложнее - выполнить WHERE с участием двух полей
- Подойдем менее радикально

ПРОМЕЖУТОЧНОЕ РЕШЕНИЕ

- Логическую таблицу нарежем на кусочки
- Размер кусочков - соразмерно HDFS-блоку
- А внутри кусочка - "транспонируем" данные

ЧАСТНЫЙ СЛУЧАЙ ПАРТИЦИОНИРОВАНИЯ

- Есть общий шаблон проектирования баз данных: партиционирование
- На примере классического SQL: давайте создадим несколько таблиц одной структуры
- И будем считать их частью одной большой логической таблицы
- При добавлении выбираем таблицу по какому-то полю или по времени добавления

ЧАСТНЫЙ СЛУЧАЙ ПАРТИЦИОНИРОВАНИЯ

- Удобно массированно удалять (при правильно подобранном критерии)
- Удобно управлять индексами
- Может помогать в поиске

ЧАСТНЫЙ СЛУЧАЙ ПАРТИЦИОНИРОВАНИЯ

- Вернемся к нашему случаю
- В каком-то смысле - это вариант партиционирования
- Есть сложности с одной большой вертикальной таблицей - сделаем их несколько поменьше