BIGDATA

Лекция 2

ПЛАН

- Концепция MapReduce
- Схема реализации
- Базовые алгоритмы
- Вспомогательные структуры

ПРИМЕР

- Обходим интернет, собираем тексты
- Каждый текст храним в условной строке HDFS-файла
- С какими-то метаданными
- Что-то типа csv-файла

ПРИМЕР

- Хотим как-то обработать
- Например, подсчитать количество слов в текстах
- Для каждого слова - свое число

ПРИМЕР

- Файл читать умеем (в том числе HDFS-файл)
- Обработать CSV - тоже
- И слова выделить
- Через Counter прогнать
- Слов будет - до миллиона, не больше

ПРИМЕР

- Все выглядит подъемно
- Но есть нюанс
- Все данные поедут туда, где работает наш код
- Что немного обидно
- А по месту жительства данных есть свои процессоры

СТРУКТУРА ДАННЫХ И ВЫЧИСЛЕНИЙ

- Исходим из табличной модели
- Много записей ("вытянуто в высоту")
- Записи "маленькие"
- Уж точно влезают в HDFS-блок
- По-хорошему - сотни или тысячи

СТРУКТУРА ДАННЫХ И ВЫЧИСЛЕНИЙ

- Запись легко влезает в память (в отличие от таблицы в целом)
- Устройство записи - произвольно
- Может быть CSV-строка
- Или protobuf, avro т.п.
- Или json

СТРУКТУРА ДАННЫХ И ВЫЧИСЛЕНИЙ

- Вычисления отталкиваются от записей
- WordCount: для начала режем текст на слова
- Какие-то записи могут вообще нас не интересовать
- Часто можем решить это, глядя на запись
- (Не всегда: при отборе по критерию "количество слов выше среднего" среднее не знаем локально)

СТРУКТУРА ДАННЫХ И ВЫЧИСЛЕНИЙ

- Первая фаза - преобразование каждой записи
- Возможно, с фильтрацией
- Фильтрацию можно считать частным случаем
- Назовем эту фазу map

СВЕРТКА

- Просто набора преобразованных записей обычно недостаточно
- Обычно надо сделать что-то еще
- И это что-то можно обобщить
- Оттолкнемся от понятия map

СВЕРТКА

- map применяет функцию к каждой записи
- А мы хотим посчитать функцию от многих записей
- Допустим, что функцию от таблицы можно вычислять инкрементально
- Очень часто так и есть: сумма какого-то поля по всех записях, максимум, top 10

СВЕРТКА

- К каждой записи применим некую функцию
- А значение - "передадим дальше"
- У функции два параметра: запись и накопленное значение
- При первом вызове - какое-то базовое, "ноль" в смысле нашей общей функции

СВЕРТКА В WORDCOUNT

- Начальное значение - пустой словарь
- Преобразованная запись - словарь, описывающий количества слов в документе
- Значение - обновленный словарь
- (Это не финальная версия)

ВЕРНЕМСЯ В HDFS

- Каждый файл хранится на разных узлах
- Запись точно хранится на одном узле
- Можем на каждом узле параллельно запустить код
- В нем будет универсальная часть (независимая от конкретной задачи)
- И часть, зависящая от конкретной задачи

ВЕРНЕМСЯ В HDFS

- Универсальная часть делит блок на записи
- Но критерий разделения конфигурируется для задачи
- (По переводу строки, по фиксированной длине и т.п.)
- Разбираемся с записями, по которым прошла граница блока

ВЕРНЕМСЯ В HDFS

- Для каждой записи вызывается логика map
- Ее надо написать для конкретной задачи
- Технически это реализация абстрактного класса на Java
- Или реализация функции на Python - если у нас есть такая надстройка

ВЕРНЕМСЯ В HDFS

- Логика map вызывается по 1 разу на запись
- С записью приезжает некий ключ
- Во многих задачах он не нужен и игнорируется
- Но может содержать знание об источнике записи

ВЕРНЕМСЯ В HDFS

- Пример ключа: смещение в HDFS файле (по умолчанию)
- Или номер блока
- Или имя файла, если на входе несколько файлов
- (Актуально для реализации JOIN)
- Логику назначения ключа можно написать для конкретной задачи

REDUCE: БАЗОВЫЙ И УТОЧНЕННЫЙ

- Базовая версия свертки предполагает сбор всех данных от map
- В одной точке
- Иногда это выглядит неизбежным (max, sum) - но такое впечатление бывает обманчивым
- А иногда сходу видно, почему этого можно избежать

REDUCE: БАЗОВЫЙ И УТОЧНЕННЫЙ

- WordCount: зачем тащить словари-счетчики со всех документов в один узел ?
- Счетчики для каждого слова независимы
- А давайте сделаем несколько точек свертки
- Каждая будет отвечать за свертку для группы слов

СВЕРТКА ПО КЛЮЧАМ

- Для свертки тоже нужен ключ
- И он имеет "системное" значение (в отличие от ключа для map)
- Но мы пока еще в WordCount порождаем словарь и про ключи на выходе с map речь не шла
- Надо как-то "распилить" этот словарь на пары ключ-значение

УТОЧНЯЕМ MAP

- Логика распиливания вытекает из логики конкретной задачи
- Не можем выполнять универсально
- Обобщим интерфейс map
- Разрешим порождать много значений
- Каждое - со своим ключом

УТОЧНЯЕМ MAP

- Базовый map превращается в порождение одноэлементного набора
- Отфильтровывание - в порождение пустого набора
- Если хотим глобальную свертку - порождаем всегда один ключ

ВОЗВРАЩАЕМСЯ В REDUCE

- На каждом узле с данными map порождаем набор пар ключ-значение
- И есть сколько-то машинок, готовых исполнять reduce
- Конфигурация по умолчанию распределяет данные по хешу от ключа
- По остатку от деления на количество машинок (которое определяется конфигурированием)

ВОЗВРАЩАЕМСЯ В REDUCE

- Каждый map-узел считает хеш от ключа и раскладывает данные по "корзинкам"
- И в итоге отправляются на нужную машинку
- Там они уже группируются по точному значению ключа
- И потенциально от каждой map-машины - свой поток пар
- И для каждого ключа собираем записи от всех

ВОЗВРАЩАЕМСЯ В REDUCE

- Настает звездный час reduce-логики
- Реализуем метод/функцию на Java/Python
- На входе - НЕ список !!!
- Для Python - скорее ленивый генератор
- И не надо его превращать в список не глядя

ВОЗВРАЩАЕМСЯ В REDUCE

- Реализация reduce-части - вычитывание входных записей
- И агрегация
- В идеале - с константым расходом памяти
- Ну ладно - можно логарифмическим

Пример

```
1  public class TokenCounterMapper
2     extends Mapper<Object, Text, Text, IntWritable>{
3
4     private final static IntWritable one =
5          new IntWritable(1);
6     private Text word = new Text();
7
8     public void map(Object key, Text value,
9                     Context context)
10       throws IOException, InterruptedException {
11
12
13       // ..................
```

Пример

```
1       // ..................
2
3       // разбиваем на слова
4       StringTokenizer itr =
5            new StringTokenizer(value.toString());
6
7       while (itr.hasMoreTokens()) {
8         word.set(itr.nextToken());
9         // ключ - слова, 1 - значение
10        context.write(word, one); // тут можно улучшить
11      }
12      // map-ключ не использовали
13    }
14  }
```

Пример

```
1  public class IntSumReducer<Key>
2         extends Reducer<Key,IntWritable,
3                         Key,IntWritable> {
4    private IntWritable result = new IntWritable();
5
6    public void reduce(Key key,
7                       Iterable<IntWritable> values,
8                       Context context
9    ) throws IOException, InterruptedException {
10     // ..............
```

Пример

```
 1      // ..............
 2      int sum = 0;
 3      // итерируемся, но не материализуем
 4      // одномоментно в памяти
 5      for (IntWritable val : values) {
 6        sum += val.get();
 7      }
 8      result.set(sum);
 9      context.write(key, result);
10    }
11  }
```

Python (mrjob)

```python
from mrjob.job import MRJob
class MRWordCount(MRJob):
    def mapper(self, _, line):
        for word in line.split():
                yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()
```

КРУПНЫМ ПЛАНОМ

- Данные режутся на записи
- Запись передается в map-часть алгоритма
- Может передаваться ключ с данными о происхождении записи
- map-часть порождает набор пар ключ-значение
- Этот ключ используется для группировки

КРУПНЫМ ПЛАНОМ

- reduce-часть алгоритма запускается для каждого уникального ключа по 1 разу
- На входе - итератор
- Порядок данных - произвольный
- Данные перебираются и агрегируются
- Результат агрегации отдается системе как пара ключ-значение и оседает в выходном файле

ПРИМЕРЫ

- Глобальные простые агрегации: sum/max/count
- Агрегации с группировкой (аналог GROUP BY в SQL)
- Дедупликация
- Можем даже по какому-то критерию выбирать элемент

БЕЗ REDUCE

- "SELECT", "FILTER", "SELECT" + "FILTER"
- Reduce-фаза не нужна
- Но можем сделать вырожденный reduce и получить сортировку по ключу свертки
- Или прямо отказаться от reduce (без сортировки, но быстрее)

АГРЕГАЦИИ

- Есть узкое место
- Рассмотрим вариант глобального max/sum/count
- Или с разбиением на малое число больших групп
- От каждой записи летит чиселка в reduce-машинку
- А надо ли ?

АГРЕГАЦИИ

- Общее в них - ассоциативность
- map-узлов обычно сильно меньше, чем записей
- Хотелось бы уметь частичные свертки делать на них же
- И отсылать на финальный reduce сильно меньше данных

COMBINER

- Такой механизм есть, называется Combiner
- "Локальный" reducer
- В нашем случае он повторяем логику общего reduce
- Но не обязан - и иногда это бывает полезно

TOP 10

- Для каждого покупателя хотим найти 10 самых дорогих чеков
- Без комбайнера - ключом будет id покупателя, значением - сумма чека
- reduce будет отбирать лучших (heap)
- Можно завести комбайнер

TOP 10

- Первоначальная версия reduce переезжает в combine
- В reduce начинаем принимать списки в качестве значений
- combine и reduce разошлись в поведении
- Но надо еще и map подкорректировать

TOP 10

- В силу нюансов реализации не всегда удобно все выходы map пропускать через combiner
- MR старается это сделать, но не гарантирует
- В reduce может приехать запись с map, минуя combiner
- Вариант 1: сделать данные с combine отличимыми от map и учитывать в логике reduce
- Вариант 2: сделать данные с map частным случаем данных с combine

СРЕДНЕЕ

- Лучше с комбайнером
- Передаем локальное среднее и количество
- Не забываем вариант "мимо комбайнера"
- С медианой посложнее

ПРО HAVING

- Аналог "HAVING" сделать несложно
- И даже если считаем одну агрегацию, а фильтруем по другой
- Например: самый дорогой чек, для покупателей, у которых средний чек выше фиксированного значения
- Сложнее, когда начинаются аналоги подзапросов

ДОПОЛНИТЕЛЬНЫЕ БОНУСЫ

- Максимально используется параллелизм
- Если данные реплицируются - выбирается одна из реплик
- Единица межузлового параллелизма - блок
- Если реплика выходит из строя - система узнает и перезапустит на другой
- И с reduce - тоже

НЮАНСЫ РЕАЛИЗАЦИИ

- Пишем код, который оформляет MapReduce-задачу (Job)
- И код самой задачи
- Они могут находиться в одном файле
- Но сам код задач будет работать на других машинах
- И print будет печататься на них

НЮАНСЫ РЕАЛИЗАЦИИ

- Мы можем завести глобальную или статическую переменную
- Например, считать количество вызовов map
- Но надо понимать, что это будет работать на разных машинках и в разных процессах
- И каждый счетчик посчитает часть от общего
- И динамика может меняться от запуска к запуску

НЮАНСЫ РЕАЛИЗАЦИИ

- Грубая ошибка - пытаться в алгоритме ориентироваться на глобальное состояние
- На худой конец можно сохранять что-то промежуточное в SQL-базе, kv-хранилище, hdfs-файле
- (Но не в обычном файле на локальном диске)

НЮАНСЫ РЕАЛИЗАЦИИ

- Но надо держать в голове сценарии аварий, рестартов
- Лучший вариант - "приняли от системы, обработали, записали в систему"
- Не закладываться на то, что не гарантируется

НЕСКОЛЬКО ЗАДАЧ

- Задача может не влезать в схему map + reduce
- Например, надо посчитать среднее число покупок для людей
- А потом узнать день недели, в который больше всего было потрачено теми, кто покупает выше среднего
- Организуем цепочку задач

ПАРТИЦИОНИРОВАНИЕ

- Можем настраивать распределение reduce-узлов
- Например, чтобы бороться с перекосом нагрузки из-за выбросов