Яндекс 

Map-Reduce и стриминг в Python

3 ноября 2020 г.

Максим Ахмедов[1]

[1] Разрабатываю MR в YT

План на сегодня

- Map-Reduce
- Модель данных «распределённая система»
- Стриминг в обычной жизни
- Модель данных «стриминг»

Map-Reduce

Map-Reduce

- Вычислительная парадигма, придуманная в Google
- Представлена широкой публике в статье 2004 году (J. Dean)
- В 2006 году появился свой прототип в Яндексе
- В 2007 году появился Apache Hadoop — первая открытая реализация
- Используется и по сей день в хоть сколько-то больших вычислительных кластерах

Табличная модель вычислений

- Графы, деревья, другие содержательные структуры данных — слишком сложно для внешней памяти распределенных систем
- Таблицы со строками — просто и удобно
- `table := sequence of rows`
- `row := dictionary : column name → field`

Пример

- Визуально:

name	age	height
Max	24	183
Peter	30	162
Maria	20	200

- Логически:

```
[
    {"name": "Max", "age": 24, "height": 183},
    {"name": "Peter", "age": 30, "height": 162},
    {"name": "Maria", "age" 20, "height": 200},
]
```

Табличная модель вычислений

- Физически таблица может находиться на одной машине, а может быть нарезана на произвольное число кусков, находящихся на разных машинах.
- Замечательная черта парадигмы Map-Reduce: она хорошо масштабируется в распределённых системах, позволяя мыслить таблицами и не задумываться о физическом аспекте хранения.

Другой пример

- Дневной лог запросов к поиску Яндекса:

ts	query_text	device
14211307	”Где скачать Death Stranding?”	”PC”
14211396	”Где растёт кокос”	”Android”
14211402	”СКАЧАТЬ LITTLE BIG”	”Android”

- Размер таблицы с подобным дневным логом продакшн-сервиса может достигать петабайт данных.

Word Count

- Классическая задача, на которой можно проиллюстрировать парадигму Map-Reduce, это Word Count.
- Дан текст, представленный в виде набора строковых значений в столбце (потенциально большой) таблицы.
- Посчитать количество вхождений каждого слова в текст.
- Иными словами, сформировать таблицу из пар $(word_i, count_i)$.

Map

- Первая операция в нашей модели — $\texttt{Map}_M(T)$, которая применяет маппер M к таблице T.
- Маппер — правило, по которому одну строку таблицы можно преобразовать в одну или несколько (возможно ноль) строк новой таблицы.
- $M : \texttt{row} \to \texttt{row*}$.
- Операция $\texttt{Map}_M(T)$ применяет M ко всем строкам входной таблицы и составляет выходную таблицу из объединения результатов.
- $\texttt{Map}_M : table \to table$.

Подготовительные мапперы

- Примеры мапперов:
  - Project: оставить одно и то же подмножество столбцов в каждой строке.
  - StripPunctuation: избавить строковое значение от символов пунктуации в конкретном столбце.
  - ToLowercase: привести к нижнему регистру строковые значения в конкретном столбце.
- Все эти мапперы биективны, то есть они переводят строку таблицы ровно в одну строку таблицы.
- Подействуем по очереди мап-операциями с описанными мапперами на таблицу из предыдущего примера.

Word Count, шаг 1

› T_0:

ts	query_text	device
14211307	”Где скачать Death Stranding?”	”PC”
14211396	”Где растёт кокос”	”Android”
14211402	”СКАЧАТЬ LITTLE BIG!”	”Android”

› $T_1 = \mathtt{Map}_{Project}(T_0)$ (проецируем на колонку query_text):

query_text
”Где скачать Death Stranding?”
”Где растёт кокос”
”СКАЧАТЬ LITTLE BIG!”

Word Count, шаг 2

› T_1:

query_text
”Где скачать Death Stranding?”
”Где растёт кокос”
”СКАЧАТЬ LITTLE BIG!”

› $T_2 = \texttt{Map}_{StripPunctuation}(T_1)$:

query_text
”Где скачать Death Stranding”
”Где растёт кокос”
”СКАЧАТЬ LITTLE BIG”

Word Count, шаг 3

- T_2:

query_text
”Где скачать Death Stranding”
”Где растёт кокос”
”СКАЧАТЬ LITTLE BIG”

- $T_3 = \texttt{Map}_{ToLowercase}(T_2)$:

query_text
”где скачать death stranding”
”где растёт кокос”
”скачать little big”

Маппер Split

- Split: взять строковое значение из строки и породить по одной строке на каждое слово в этом строковом значении
- Уже не биективный маппер.

Word Count, шаг 4

› T_3:

query_text
”где скачать death stranding”
”где растёт кокос”
”скачать little big”

› $T_4 = \mathtt{Map}_{Split}(T_3)$:

word
”где”
”скачать”
”death”
”stranding”
”где”
”растёт”
”кокос”
”скачать”
”little”
”big”

Reduce

- Вторая операция модели — Reduce.
- $\texttt{Reduce}_{R,cols}(T)$ логически состоит из двух шагов:
  - разбить строки таблицы по группам согласно кортежу значений в наборе колонок $cols$;
  - применить редьюсер R к каждой подобной группе.
- Таким образом, R должен быть функцией $R : \texttt{row*} \to \texttt{row*}$.
- $\texttt{Reduce}_{R,cols} : \texttt{table} \to \texttt{table}$.
- $cols$ называется **ключом** редьюса.

Считающий редьюсер

- `Count`: простейший пример редьюсера, считающего количество строк в каждой группе.
- Формально, пусть `rows` — это группа строк с одинаковым значением `row["word"]`. Тогда:

$$\texttt{Count(rows)} = [\{\text{"}word\text{"} : rows[0][\text{"}word\text{"}], \text{"}count\text{"} : len(rows)\}]$$

- Это агрегирующий редьюсер, он принимает пачку строк и возвращает одну агрегированную строку.

Word Count, шаг 5

- T_4:

word
”где”
”скачать”
”death”
”stranding”
”где”
”растёт”
”кокос”
”скачать”
”little”
”big”

- $T_5 = \text{Reduce}_{Count,\{\text{"word"}\}}(T_4)$:

word	count
”где”	2
”death”	1
”скачать”	2
”stranding”	1
”растёт”	1
”кокос”	1
”little”	1
”big”	1

Парадигма Map-Reduce

- Резюмируем.
- Мы сформулировали две логические операции:
  - $\texttt{Map}_M : \texttt{table} \to \texttt{table}$
  - $\texttt{Reduce}_{R,cols} : \texttt{table} \to \texttt{table}$
- Простая и понятная абстрактная модель
- Чем же она хороша?

Модель данных «распределённая система»

Хранение таблиц

- setup1:
  - есть одна машина с $HDD = 4TiB$ и $CPU = 20cores$
  - таблица размером $1TiB$ хранится на HDD;
  - маппер M применяется к таблице в 10 потоков
- setup2:
  - есть 10 машин в той же конфигурации
  - таблица размером $1TiB$ разбита на 10 кусков по $100GiB$ на HDD на каждой машине
  - маппер M применяется к каждому куску на своей машине независимо в один поток
- **Вопрос**: какой setup отработает быстрее?

Хранение таблиц

- setup1:
  - есть одна машина с $HDD = 4TiB$ и $CPU = 20cores$
  - таблица размером $1TiB$ хранится на HDD;
  - маппер M применяется к таблице в 10 потоков
- setup2:
  - есть 10 машин в той же конфигурации
  - таблица размером $1TiB$ разбита на 10 кусков по $100GiB$ на HDD на каждой машине
  - маппер M применяется к каждому куску на своей машине независимо в один поток
- **Вопрос**: какой setup отработает быстрее?
- **Ответ**: setup2, т.к. IO сильно медленнее CPU

Распределённый Map

- Устроен предельно легко
- Независимо обрабатываем порции таблицы на каждой машине в отдельности

Распределённый Reduce

- Всё сложнее
- Группа строк с одинаковым значением ключа может быть разбросана по произвольному набор машин
- **Вопрос**: что же делать?

Сортировка

- Расширим нашу модель третьей операцией: `Sort`
- $\texttt{Sort}_{cols} : \texttt{table} \to \texttt{table}$
- После сортировки строки с одним значением ключа *cols* окажутся рядом.

Sorted Reduce

- Получаем первый вариант стратегии для Reduce: SortedReduce.
- SortedReduce реализует $\texttt{Reduce}_{R,cols}(T)$ при выполнении **предусловия**: таблица T сортирована по набору колонок $cols$.
- Типичный паттерн действий: `Map`, `Sort`, `Reduce`.

Shuffle

- Опишем альтернативный путь
- Вместо сортировки часто используется примитив `Shuffle` (иногда называется партицирование)
- `Shuffle` раскладывает строки по корзинам согласно некоторой функции от ключа *key*, типично — *hash*(*key*) mod *bucketCount*
- Каждая корзина целиком живёт на одной машине, значит одинаковые ключи снова оказались «рядышком»
- Обрабатываем каждую корзину независимо

Распределённые Sort и Shuffle

- **Вопрос**: как устроены?
- Подумайте сами на досуге :)

Где мы сталкиваемся со стримингом?

Shell

- Утилиты командной строки, превращающие вход (в stdin) в выход (в stdout):

`cat`	`rev`	`sort`
`sed -s 's/foo/bar/g'`	`grep 'foo'`	`tac`
`wc -l`	`tail -n 10`	`shuf`
`cut -f2`	`head -n 10`	
`md5sum`	`uniq -c`	
`head -c 1024`		
`tail -c 1024`		
`gzip -f`		

- **Вопрос:** по какому принципу сгруппированы утилиты?

Shell

› Утилиты командной строки, превращающие вход (в stdin) в выход (в stdout):

`cat`	`rev`	`sort`
`sed -s 's/foo/bar/g'`	`grep 'foo'`	`tac`
`wc -l`	`tail -n 10`	`shuf`
`cut -f2`	`head -n 10`	
`md5sum`	`uniq -c`	
`head -c 1024`		
`tail -c 1024`		
`gzip -f`		
$\mathcal{O}(1)$ state	$\mathcal{O}(\textit{line length})$ state	arbitrarily large state

› **Вопрос:** по какому принципу сгруппированы утилиты?

Pipe’ы

- Программы можно сцеплять друг за дружкой посредством пайпов:
- `cut -f2 | sed -s 's/foo/bar/g' | md5sum`
- Если все программы являются потоковыми, то комбинация через пайпы — тоже!
- Ядро ОС будет перекладывать небольшие кусочки данных из выхода одной программы во вход следующей, следя, чтобы в буферах пайпов не накапливалось слишком много необработанных данных
- Простейший пример парадигмы потоковой обработки aka «streaming».

Итераторы и генераторы в Python

- Основные примитивы для потока данных в Python — итераторы и генераторы.
- `range(n)` — поток чисел до *n*
- `open("file.txt", "r")` — поток строк из файла `file.txt`
- `(x * x for x in iterable)` — generator expression
- Предыдущее — частный случай генератора (функции с yield'ами).

range

- **Вопрос**: что содержится в состоянии у `range(n)`?

range

- **Вопрос**: что содержится в состоянии у `range(n)`?
- Несколько интов: текущее число, граница, шаг.
- $\mathcal{O}(1)$ state.

open

- **Вопрос**: что содержится в состоянии у `open("file.txt", "r")`?

open

- **Вопрос**: что содержится в состоянии у `open("file.txt", "r")`?
- Номер файлового дескриптора (идентификатор открытого файла в ядре ОС)
- Буфер чтения (порядка 64 KiB, чтобы не делать random read на каждую новую строку)

Материализация

- Рассмотрим функцию:

```
def filter_even(seq):
    result = []
    for item in seq:
        if item % 2 == 0:
            result.append(item)
    return result
```

- Пусть она используется следующим образом:

```
for even_item in filter_even(seq):
    print(even_item)
```

- **Вопрос**: какое у неё потребление памяти?

Материализация

- Рассмотрим функцию:

```
def filter_even(seq):
    result = []
    for item in seq:
        if item % 2 == 0:
            result.append(item)
    return result
```

- Пусть она используется следующим образом:

```
for even_item in filter_even(seq):
    print(even_item)
```

- **Вопрос**: какое у неё потребление памяти?
- Линейное, она материализует ответ в список.

Генератор

- Рассмотрим альтернативную реализацию:

```
def xfilter_even(seq):
    result = []
    for item in seq:
        if item % 2 == 0:
            yield item
```

- **Вопрос:** а у неё какое потребление памяти?

Устройство генератора

- Можно думать про генератор как про функцию, ”замороженную” в процессе исполнения.

```
gen = xfilter_even(seq)
for i in gen:
    print i
```

- Хронология одной итерации:
  - `for` зовёт `next(gen)`
  - функция генератора «просыпается»
  - функция дорабатывает до следующего `yield smth`
  - функция «засыпает»
  - `smth` возвращается из `next(gen)`

Состояние генератора

- Резюмируем, из чего состоит объект генератора:
  - служебная структура, указывающая на байт-код функции-генератора в интерпретаторе
  - позиция в коде, на которой «спит» функция
  - состояния локальных переменных в момент «засыпания» функции
- В примере выше — $\mathcal{O}(1)$ state!
- Эквивалентная форма записи: `(item for item in seq if item % 2 == 0)`.

Эффективность

- **Вопрос:** какая из реализация `filter_even` и `xfilter_even` эффективнее?

Эффективность

- **Вопрос:** какая из реализация `filter_even` и `xfilter_even` эффективнее?
- `xfilter_even`:
  - constant memory;
  - context switch per `next`;
  - context switches may be bottleneck;
  - low memory is good for Python interpreter.
- `filter_even`:
  - linear memory;
  - no context switches;
  - trivial implementation;
  - on large input may slow down interpreter or lead to OOM.
- **Вопрос:** что же выбрать?
- Подумайте, как взять лучшее от обоих подходов.

Модель данных «стриминг»

Табличный поток данных

- Рассмотрим модель данных, схожую с использовавшейся в MR
- `stream := iterator of rows`.
- `row` — как и раньше, словарь.
- Рассматриваемые нами операции будут теперь иметь сигнатуру `operation : stream → stream`.
- Критерий хорошести операции: она работает потоковым образом, то есть обладает небольшим (ограниченным) state'ом.
- Потоковая операция может «бесконечно» сидеть на входном потоке, порождая выходной, и не ООМиться.
- Поток данных можно мыслить как «бесконечную вниз» таблицу.

Streaming Map

- $\texttt{Map}_M : \texttt{stream} \to \texttt{stream}$.
- Устроен очень просто.
  - вытаскиваем из потока строку;
  - применяем к ней M;
  - кладём результат в выходной поток.

Streaming Reduce

- $\texttt{Reduce}_{R,cols} : \texttt{stream} \to \texttt{stream}$.
- Как и в MR, тут всё сложнее.
- Потребуем выполнения двух предусловий:
  - входной поток отсортирован;
  - размер каждой группы редьюса небольшой.
- Тогда понятно, как делать потоковый $\widehat{\text{Reduce}}$.
  - накапливаем группу, пока ключ совпадает с предыдущим;
  - когда ключ переключился, вызываем R от накопленной группы
  - кладём результат в выходной поток.

Join

- Обсудим небольшое (и очень удобное) расширение нашей модели операций.
- В этом месте я устал техать таблички, поэтому пойдём к доске :)

Join

- Подчеркнём один момент — формально говоря, Join не является потоковой операцией согласно предыдущему определению, так как он обрабатывает два потока одновременно.
- Join'ы слишком полезные, чтобы их выкинуть, поэтому придётся исправить определение :)
- `operation : stream* → stream`
- **Вопрос**: а можно ли сказать, что потоковая операция порождает больше одного потока?

Tee

- Рассмотрим классическую операцию, которая порождает два потока — это Tee.
- Tee : `stream` → `stream` × `stream`.
- Логически устроена так: любая запись из входного потока перекладывается в оба выходных потока.
- **Вопрос**: является ли эта операция потоковой?

Tee

- Рассмотрим классическую операцию, которая порождает два потока — это Tee.
- Tee : `stream` → `stream` × `stream`.
- Логически устроена так: любая запись из входного потока перекладывается в оба выходных потока.
- Вопрос: является ли эта операция потоковой?
- Нет. Один выходной поток может дальше обрабатываться сильно медленнее, чем другой. В таком случае Tee должен помнить материализованное «окно» между позициями двух потоков.

Sort

- Как и в MR, остаётся интересный и нераскрытый **вопрос**: как может быть устроен потоковый `Sort`?

Sort

- Как и в MR, остаётся интересный и нераскрытый **вопрос**: как может быть устроен потоковый `Sort`?
- Честно — никак, это невозможно :)
- В промышленности стриминг используется на временных окнах ограниченного размера, скажем, 15 секунд.
- Соответственно, поток данных становится «локально сортированным».
- Применения `Reduce` и `Join` к этому обычно готовы, так как их смысл зачастую это «агрегация» (подсчёт суммарной статистики, разбиение на субпотоки по корзинам), которую можно делать порциями произвольного размера.

Лекция всё!

Спасибо за внимание и удачи с домашкой! :)