

Введение в обработку больших данных

Илья Кокорин

kokorin.ilya.1998@gmail.com

При участии

Романа Елизарова

Виталия Аксёнова

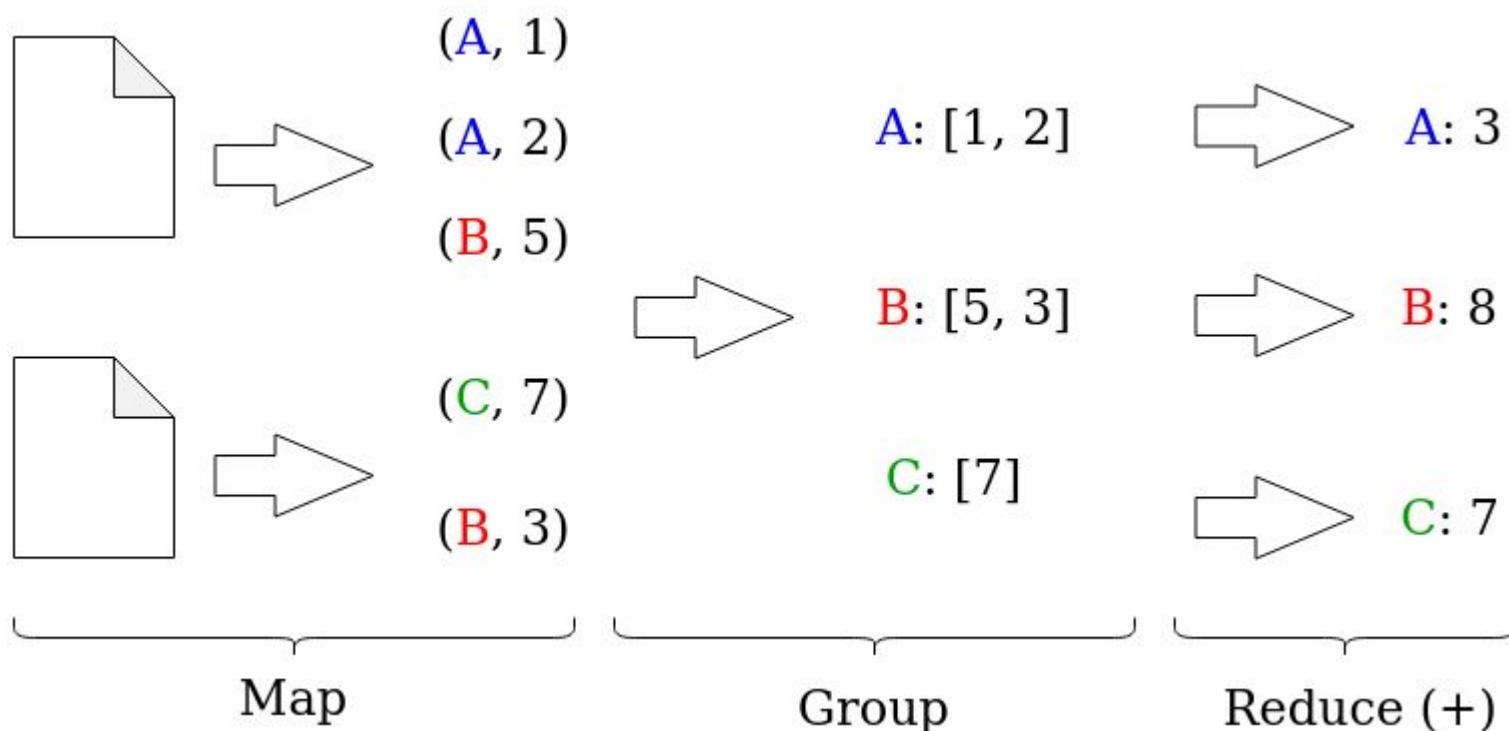
MapReduce: мотивация

- Встречаемость слов:
 - Даны коллекция текстовых документов
 - Нужно посчитать, сколько раз в коллекции встречается каждое слово
- Подсчёт tf-idf
- Посещаемость сайта:
 - Даны таблица посещений (URL, IP)
 - Нужно для каждого URL посчитать, сколько уникальных IP посещали эту страницу
- Исходные данные не влезают на одну машину
- Последовательно считать слишком долго
- Писать распределённый алгоритм с нуля - слишком сложно

MapReduce: модель

map : Document \rightarrow [(Key, Value)]

reduce : Key, [Value] \rightarrow Result



Подсчёт встречаемости слов: алгоритм

- Разбиваем документ на слова
- Для каждого слова выдаем пару (слово, 1)
 - Слово - ключ
 - 1 - значение
- За кадром происходит группировка по ключу
 - То есть по слову
- Складываем все единички чтобы узнать, сколько раз встретилось слово
 - Каждая единичка - одно использование слова

```
1 fun map(doc: Document):  
2     for word in words(doc):  
3         yield word, 1  
4  
5 fun reduce(word: String, ones: [Int]):  
6     return word, sum(ones)
```

Подсчёт встречаемости слов: пример

Хорошо, что нет Царя.
Хорошо, что нет России.
Хорошо, что Бога нет.

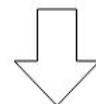
[хорошо: 1, что: 1, нет: 1, царь: 1,
хорошо: 1, что: 1, нет: 1, россия: 1,
хорошо: 1, что: 1, бог: 1, нет: 1]

Только желтая заря,
Только звезды ледяные,
Только миллионы лет.

[только: 1, жёлтая: 1, заря: 1,
только: 1, звёзда: 1, ледяная: 1,
только: 1, миллион: 1, год: 1]

Хорошо – что никого,
Хорошо – что ничего
...

[хорошо: 1, что: 1, никто: 1,
хорошо: 1, что: 1, ничто: 1]



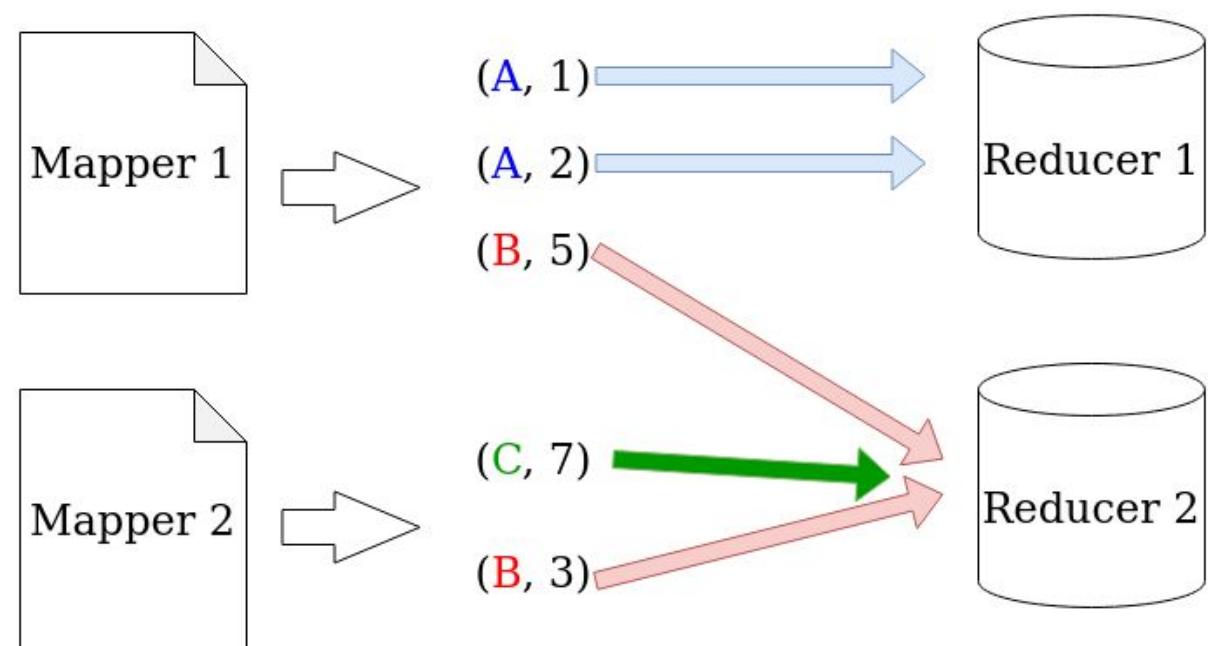
хорошо: [1, 1, 1, 1, 1] => 5;
что: [1, 1, 1, 1, 1] => 5;
нет: [1, 1, 1] => 3;
царь: [1] => 1;
россия: [1] => 1;
бог: [1] => 1;
только: [1, 1, 1] => 3;
жёлтая: [1] => 1;
заря: [1] => 1;
звезды: [1] => 1;
ледяная: [1] => 1;
миллион: [1] => 1;
год: [1] => 1;
никто: [1] => 1;
ничто: [1] => 1;

MapReduce: последовательная реализация

```
1 fun mapReduce(docs: [Document] ,  
2                 map: Document -> [(Key , Value)] ,  
3                 reduce: Key, [Value] -> Result) -> [Result]:  
4     kvs = {}  
5     for doc in docs:  
6         for key, value in map(doc):  
7             if key ∉ kvs:  
8                 kvs[key] = []  
9                 kvs[key].append(value)  
10    return kvs.map { key, values -> reduce(key, values) }
```

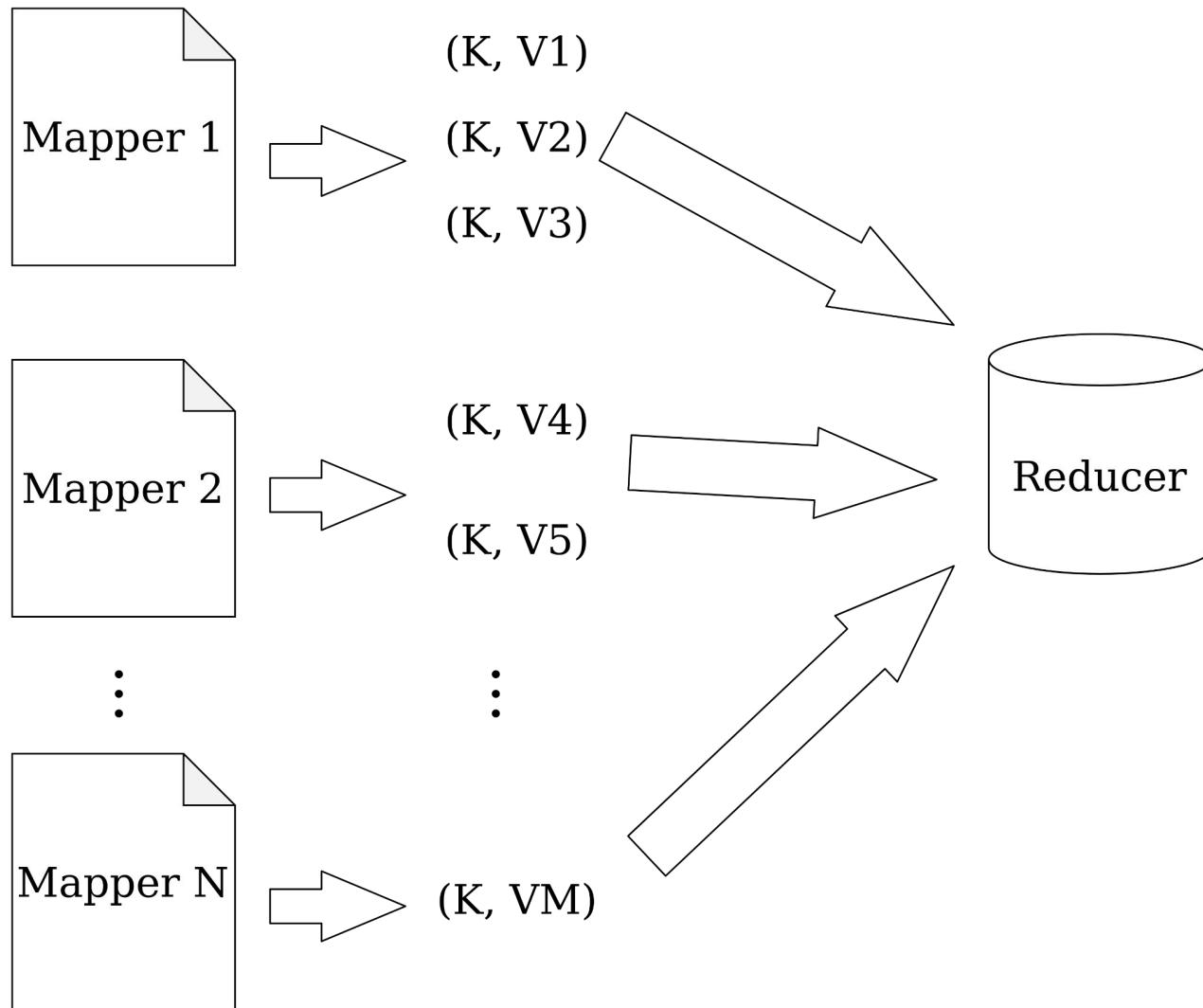
MapReduce: распределённая реализация

- В системе есть M узлов-мапперов
 - Параллельно выполняют операцию map для разных документов
- И R узлов-редьюсеров
 - Параллельно выполняют операцию reduce для разных ключей
- И один мастер для координации
- Все пары, соответствующие определённому ключу, должны попадать на одного редьюсера
 - $\text{hash}(\text{key}) \% R$



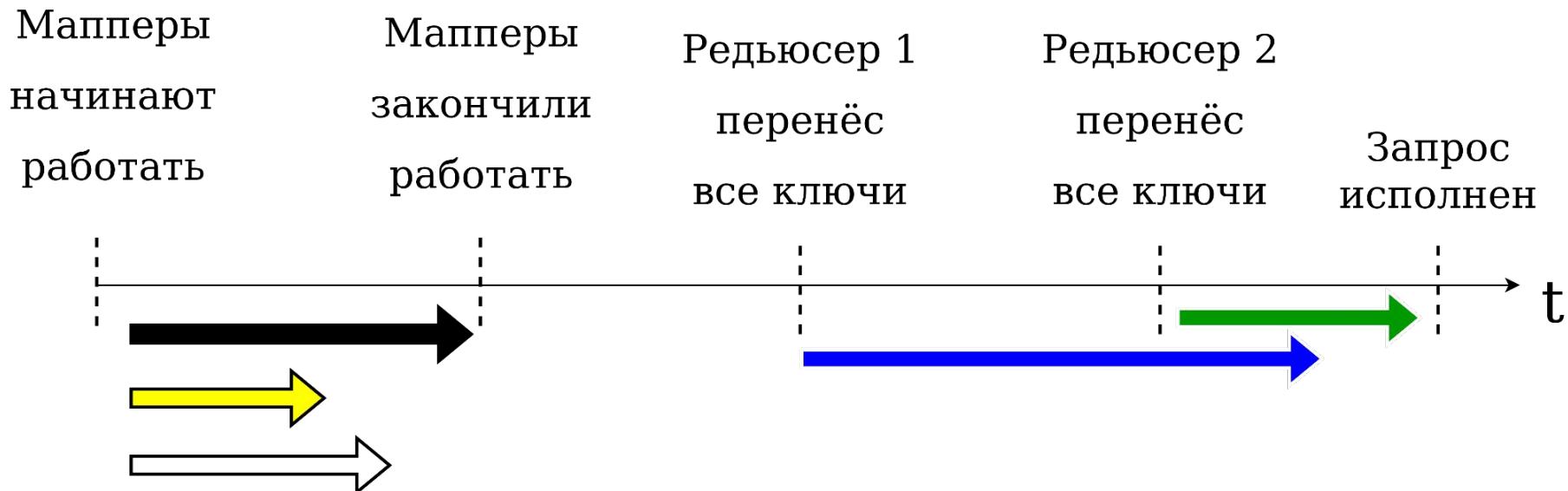
MapReduce: исполнение запроса

- Чтобы редьюсер начал работать, он должен собрать нужные ему ключи со всех мапперов



MapReduce: исполнение запроса

- Мапперы начинают работать
- **Все** мапперы закачивают работать
- Каждый из редьюсеров независимо переносит пары (ключ, значение) с мапперов
- Перенеся все пары, редьюсер начинает работать
- **Все** редьюсеры закачивают работать
- Запрос выполнен



MapReduce: каскады

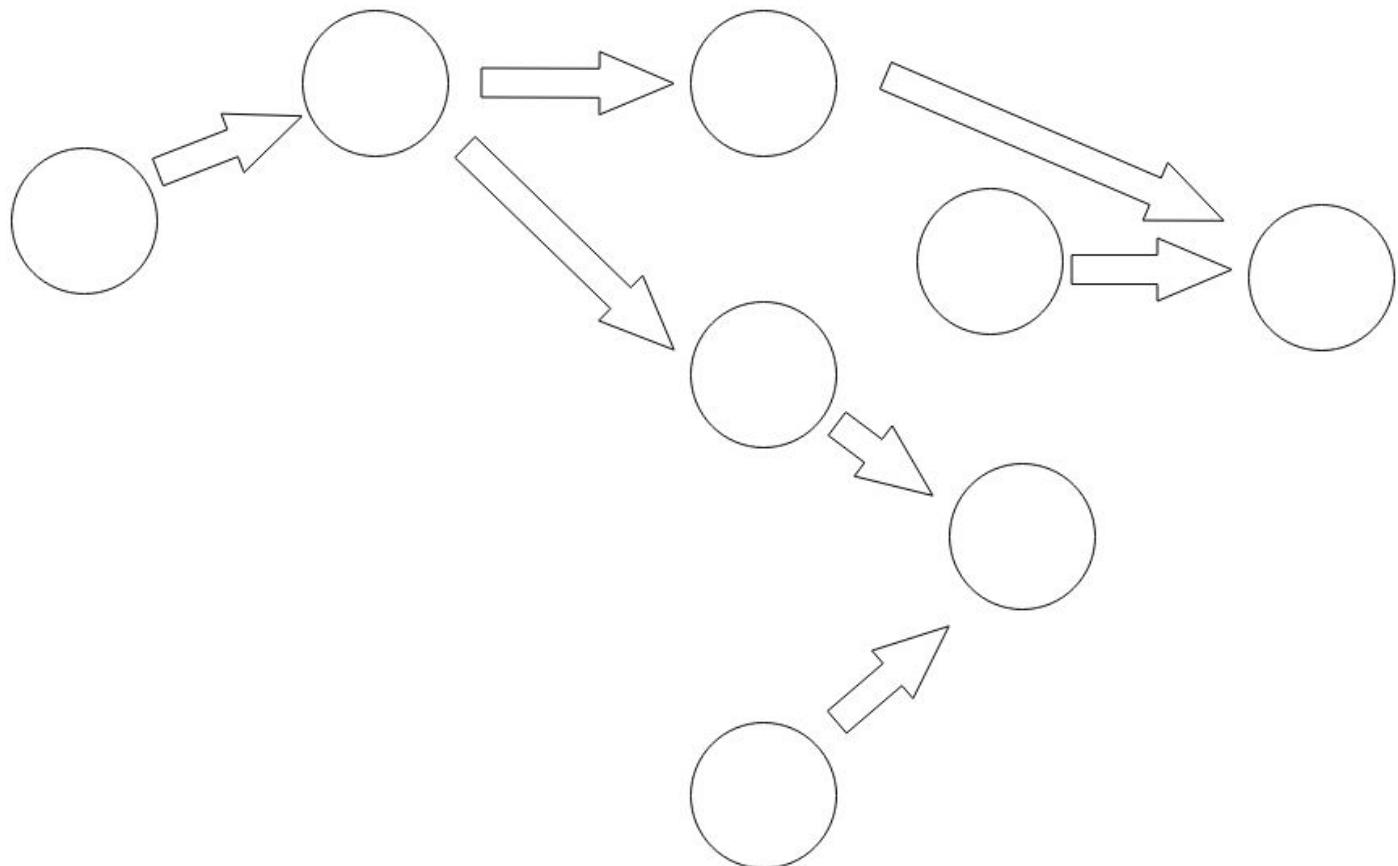
- Иногда задачу нельзя решить за один мапредьюс
- Представим задачу: для каждого интервала [0; 1000), [1000; 2000), [2000; 3000) и т.д. сказать, сколько слов имеет встречаемость в данном интервале



```
1 fun map(doc: [(String, Int)]) -> [(Int, String)]:  
2     for word, count in doc:  
3         yield [count / 1000], word  
4  
5 fun reduce(rangeNum: Int, words: [String]) -> (Range, Int):  
6     rangeBegin = rangeNum * 1000  
7     rangeEnd = rangeBegin + 1000  
8     return [rangeBegin; rangeEnd], distinct(words)
```

MapReduce: каскады

- В общем случае MapReduce задачи представляют из себя ориентированный ациклический граф
 - Топологически сортируем граф
 - Выполняем задачи в порядке топологической сортировки



Оптимизация MapReduce: Combiner

- Мар может выдать очень много пар, содержащих один и тот же ключ
 - Популярные слова при подсчёте слов
 - Приходится передавать много лишней информации по сети
- Применим локальную свёртку - превратим множество пар (я, 1) в одну пару (я, N)
 - Такая операция называется Combiner
 - Выполняется над данными только одного узла

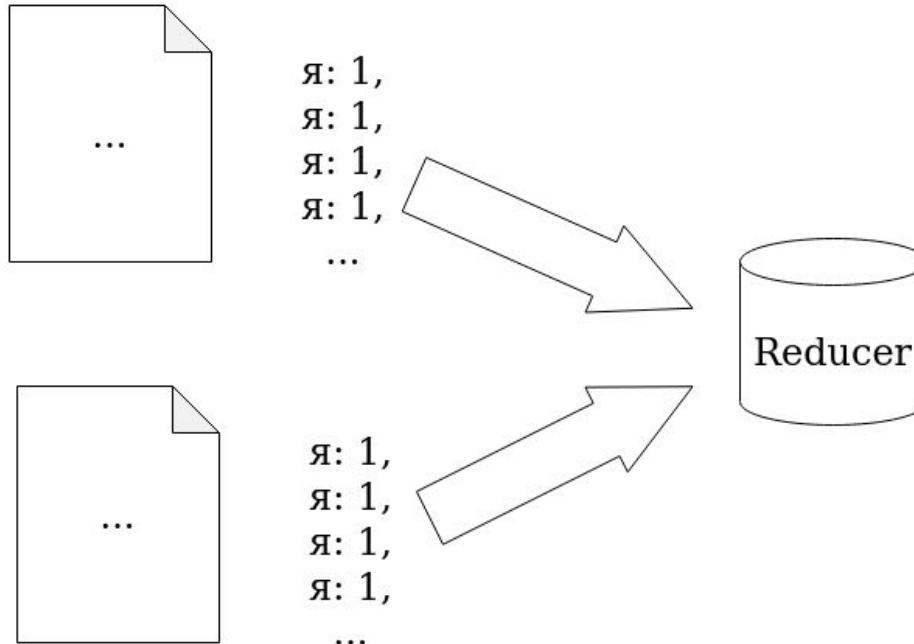
Я тега ега модо гадо.
Я тега ега могол гадо
дано. Я тега мого
нога ега
модо. я тега модо воро
нора мого. я нода поро
нега гено раты



я: 1,
я: 1,
я: 1,
я: 1,
...

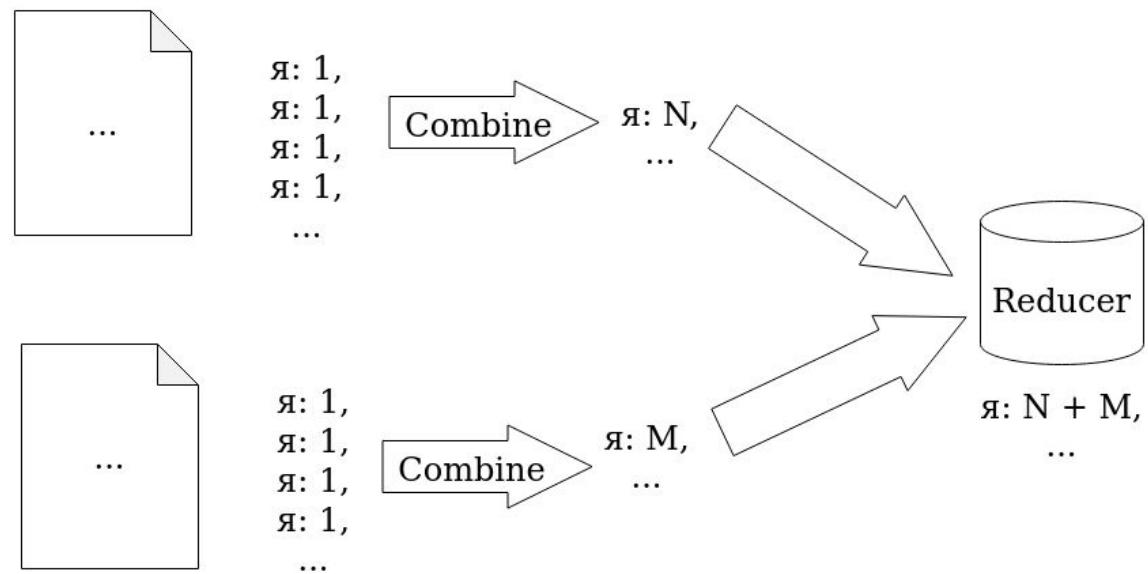
Очень много
одинаковых
слов

Combiner: применение



- Уменьшили объём передаваемых данных
- Получили тот же результат

- Чаще всего код combiner совпадает с кодом reducer
 - Но combiner выполняется локально

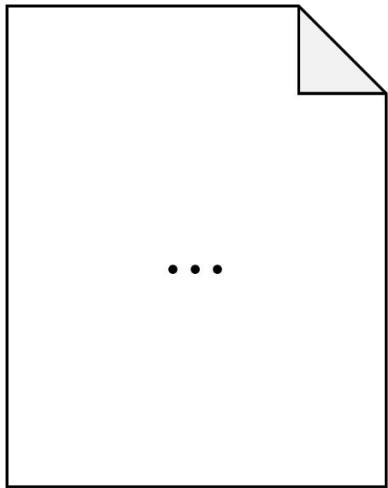


Combiner != Reducer: пример

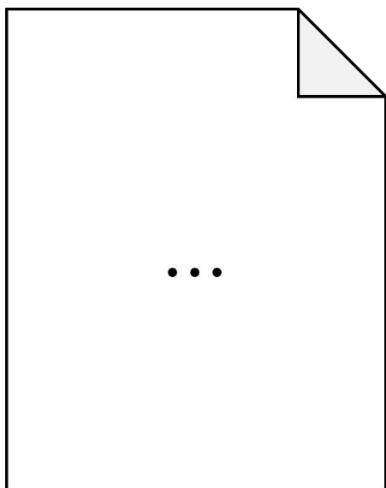
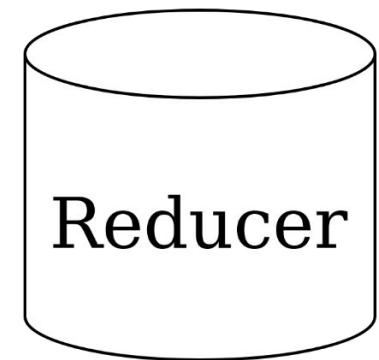
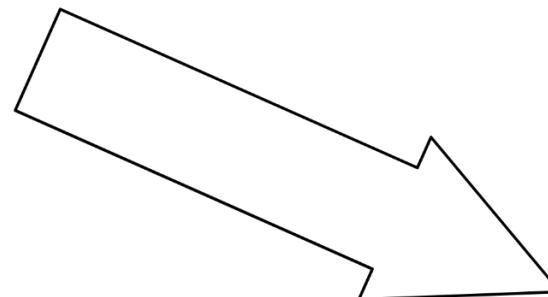
- Рассмотрим задачу: дана таблица вида (group: String, value: Float)
- Нужно найти среднее в каждой группе

```
1 fun map(doc: Doc) -> [(String, Float)]:  
2     for group, value in doc:  
3         yield group, value  
4  
5 fun reduce(group: String, values: [Float]) -> (String, Float):  
6     return group, sum(values) / values.size()
```

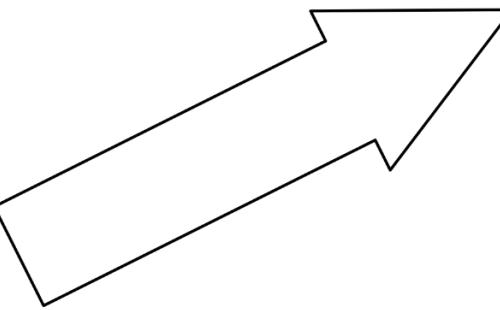
Необходимость Combiner



G: 2,
G: 5,
G: 1,
G: 7,
...

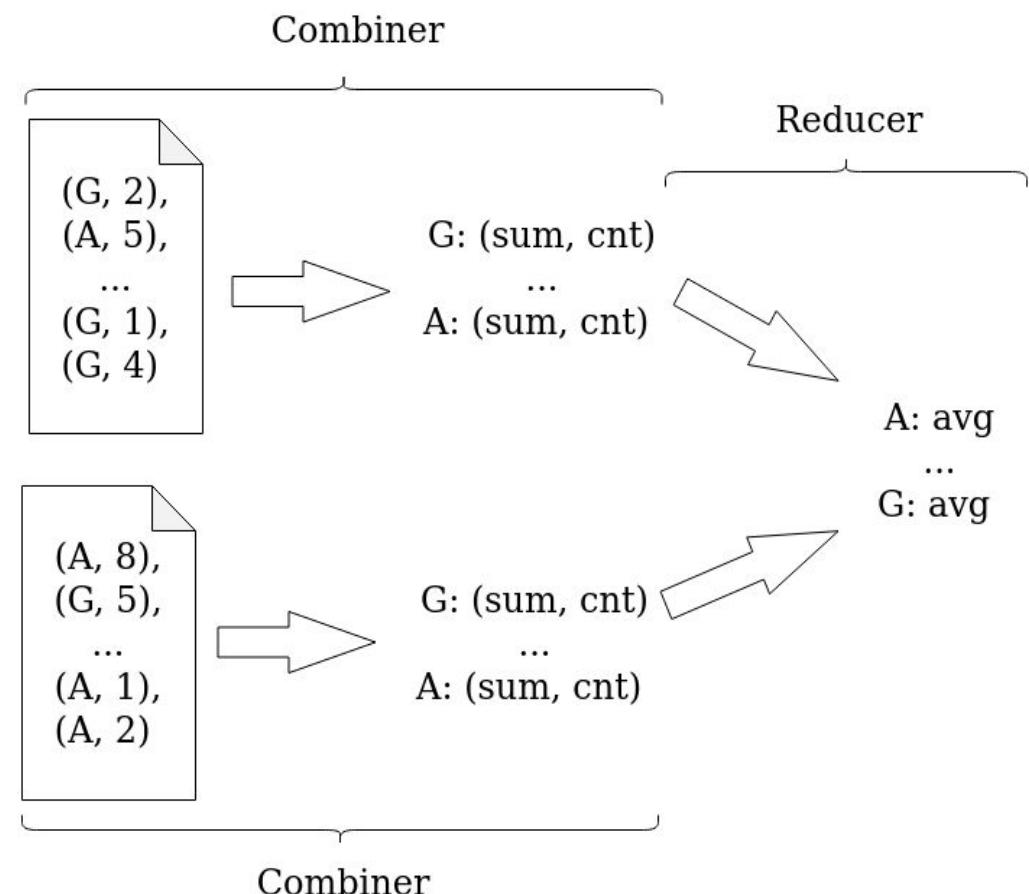


G: 1,
G: 3,
G: 3,
G: 7,
...



Combiner != Reducer: почему?

- Потому что среднее средних это не среднее!
 - $\text{average}(100, 100, 100, 1) = 75.25$
 - $\text{average}(\text{average}(100, 100, 100), \text{average}(1)) = \text{average}(100, 1) = 50.5$
 - $75.25 \neq 50.5$
- Combiner должен считать не среднее, а что-то другое
 - Сумму в каждой группе
 - И количество чисел в каждой группе



Решение проблемы

- Редьюсер считает сумму чисел для каждой группы
 - Сумма сумм, посчитанных мапперами
- И количество чисел в каждой группе
 - Сумма количеств, посчитанных мапперами

```
1 fun map(doc: Doc) -> [(String, Float)]:  
2     for group, value in doc:  
3         yield group, value  
4  
5 fun combine(group: String, values: [Float]):  
6     return group, sum(values), values.size()  
7  
8 fun reduce(group: String, values: [(Float, Int)]):  
9     total_sum = values.map { sum, _ -> sum }.sum()  
10    total_cnt = values.map { _, cnt -> cnt }.sum()  
11    return group, total_sum / total_cnt
```

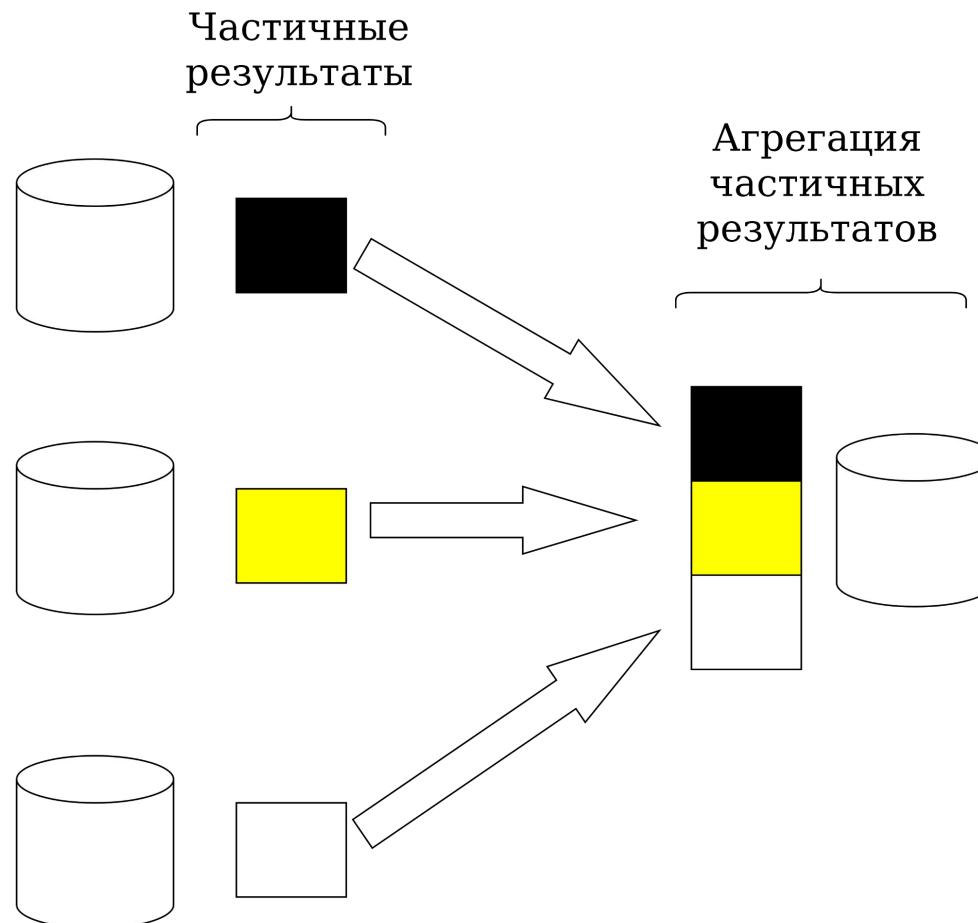
Реализации MapReduce

- Тысячи их
- Первая исторически была создана J.Dean et.al. в компании Google
- Самая популярная: Hadoop MapReduce
 - Open Source
 - Продукт фонда Apache
 - Огромное сообщество
 - Множество ПО в Hadoop-экосистеме
- Есть много других реализаций
 - Обычно они что-то оптимизируют
 - На распределённых системах поговорим о том, что можно оптимизировать в MapReduce



Философское отступление: почему MapReduce?

- Почему мы формулируем наши алгоритмы в парадигме MapReduce?
 - Не потому, что нам нравится какая-то из реализаций
- А потому, что это естественный способ формулировать распределённые алгоритмы
 - Сначала на каждом из узлов обработать свой кусочек данных
 - А потом агрегировать результаты



Философское отступление: почему MapReduce?

- Аналогичным образом параллельные алгоритмы для общей памяти мы формулируем в fork-join парадигме
- В распределённой системе мы не можем просто так сделать fork!
- Задача, исполняемая в fork-блоке должна работать параллельно
 - То есть на другой машине
- Но на другой машине лежат другие данные

```
1 fun calc_sum(arr, left, right):  
2     if left + 1 == right:  
3         return arr[left]  
4     mid = ⌊(left+right)/2⌋  
5     var a  
6     fork { a = calc_sum(arr, left, mid) }  
7     b = calc_sum(arr, mid, right)  
8     join  
9     return a + b
```

Map-only задачи: распределённый grep

- grep по большой коллекции документов
- Выполним grep локально на каждом мар-узле
- Это и есть результат: reduce просто не нужен

```
<html>
<a href = "site.com">
    </a>
<a href = "other.com">
    </a>
</html>
```

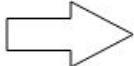
```
grep -Po '(?=<href=")[^"]*' file
```



```
site.com
other.com
```

```
<html>
<a href = "site.ru">
    </a>
</html>
```

```
grep -Po '(?=<href=")[^"]*' file
```



```
site.ru
```

Map

Итоговый
результат

Обратный индекс: структура данных

- Входные данные: корпус текстов
- Выходные данные: Мар<Слово, Список документов>
 - В которых это слово встречается
- Базовый компонент любой поисковой системы или системы обработки текста

Full-text Search 101: The inverted index

1	The old night keeper keeps the keep in the town.
2	In the big old house in the big old gown.
3	The house in the town had the big old keep.
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night.
6	And keeps in the dark and sleeps in the light.



6 documents to index

Example from:
Justin Zobel, Alistair Moffat,
Inverted files for text search engines,
ACM Computing Surveys (CSUR)
v.38 n.2, p.6-es, 2006

Term	Documents
and	<6>
big	<2> <3>
dark	<6>
did	<4>
gown	<2>
had	<3>
house	<2> <3>
in	<1> <2> <3> <5> <6>
keep	<1> <3> <5>
keeper	<1> <4> <5>
keeps	<1> <5> <6>
light	<6>
never	<4>
night	<1> <4> <5>
old	<1> <2> <3> <4>
sleep	<4>
sleeps	<6>
the	<1> <2> <3> <4> <5> <6>
town	<1> <3>
where	<4>

The index:

Dictionary and
posting lists

Обратный индекс: алгоритм

- Для каждого слова выдаём пару (слово, id документа)
- За кадром происходит группировка по словам
 - Теперь мы даже знаем, как именно!
- После чего в reduce-фазе оставим только уникальные id документов

```
1 fun map(doc: Document):  
2     for word in words(doc):  
3         yield word, doc.id  
4  
5 fun reduce(word: String, ids: [Int]):  
6     return word, set(ones)
```

Обратный индекс: пример

- Применим combiner = reducer

Хорошо, что нет Царя.
Хорошо, что нет России.
Хорошо, что Бога нет.

[хорошо: 1, что: 1, нет: 1, царь: 1,
хорошо: 1, что: 1, нет: 1, россия: 1,
хорошо: 1, что: 1, бог: 1, нет: 1]

Только желтая заря,
Только звезды ледяные,
Только миллионы лет.

[только: 2, жёлтая: 2, заря: 2,
только: 2, звёзда: 2, ледяная: 2,
только: 2, миллион: 2, год: 2]

Хорошо – что никого,
Хорошо – что ничего
...

[хорошо: 3, что: 3, никто: 3,
хорошо: 3, что: 3, ничто: 3]



хорошо: [1, 1, 1, 3, 3] => {1, 3}
что: [1, 1, 1, 3, 3] => {1, 3}
нет: [1, 1, 1] => {1}
царь: [1] => {1}
россия: [1] => {1}
бог: [1] => {1}
только: [2, 2, 2] => {2}
жёлтая: [2] => {2}
заря: [2] => {2}
звезды: [2] => {2}
ледяная: [2] => {2}
миллион: [2] => {2}
год: [2] => {2}
никто: [3] => {3}
ничто: [3] => {3}

Join: что это за операция?

- Таблица X состоит из двух столбцов: (A, B)
- Таблица Y состоит из двух столбцов: (B, C)
- Тогда Join(X, Y) состоит из столбцов (A, B, C) и определяется так:

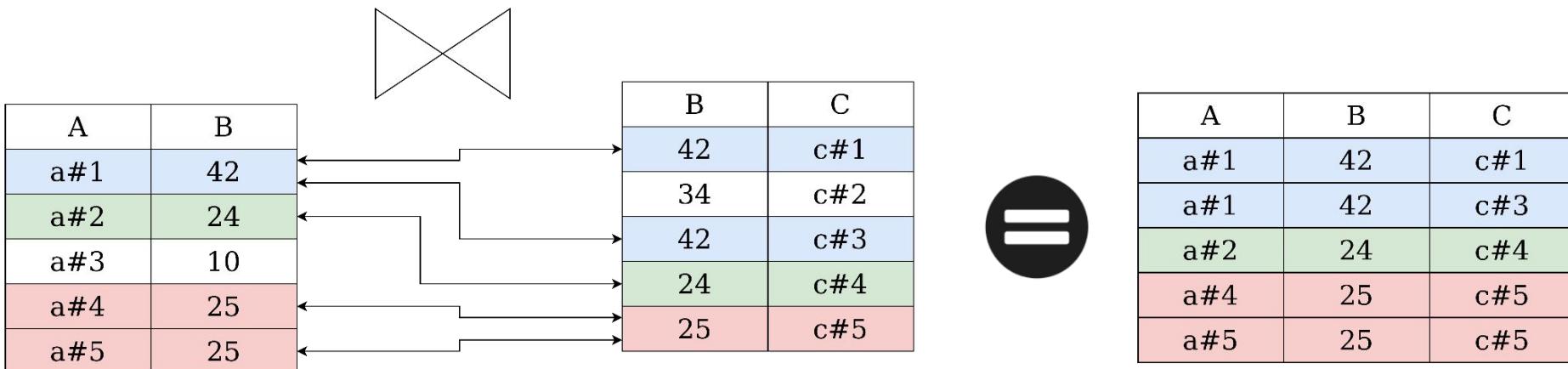
$$X \bowtie Y = \{a, b, c \mid (a, b) \in X; (b, c) \in Y\}$$

- Может быть посчитано следующим кодом:

```
1 fun X <math>\bowtie</math> Y:  
2     for a, b_1 in X:  
3         for b_2, c in Y:  
4             if b_1 == b_2:  
5                 yield a, b_1, c
```

Join: визуализируем

- Физический смысл: для каждой строки из левой таблицы ищем множество соответствующих строк из правой таблицы
 - Соответствие определяется по равенству столбца В
- Симметричная операция: $X \bowtie Y = Y \bowtie X$
- А, В, С могут быть не столбцами, а группами столбцов
 - Определение то же



Join: практический пример

- Пусть мы хотим классифицировать сообщения как спам/не спам
- Для каждого сообщения мы посчитали p_spam - вероятность того, что это сообщение спам
 - С помощью каких-то методов анализа текста
- Хотим повысить точность предсказания
- Вспомним, что у сообщения есть контекст!
 - Например, id отправителя

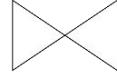
text	p_spam
...	...

text	p_spam	sender_id
...

Join: практический пример

- Пусть мы для каждого отправителя насчитали p_{spamer} - вероятность того, что этот пользователь спамер
- Сделаем join таблиц Senders и Messages
 - Теперь мы для каждой строки знаем вероятность того, что сообщение - спам
 - И того, что оно отправлено спамером
 - У нас есть уже две фичи вместо одной!

text	p_{spam}	sender_id
...



sender_id	p_{spamer}
...	...



text	p_{spam}	p_{spamer}	sender_id
...

- Join - мощный инструмент при анализе данных
- Join'ы бывают разные
 - Inner, Left Outer, Right Outer, Full Outer, Cross
 - Подробнее расскажет Георгий Александрович

Join: алгоритм для MapReduce

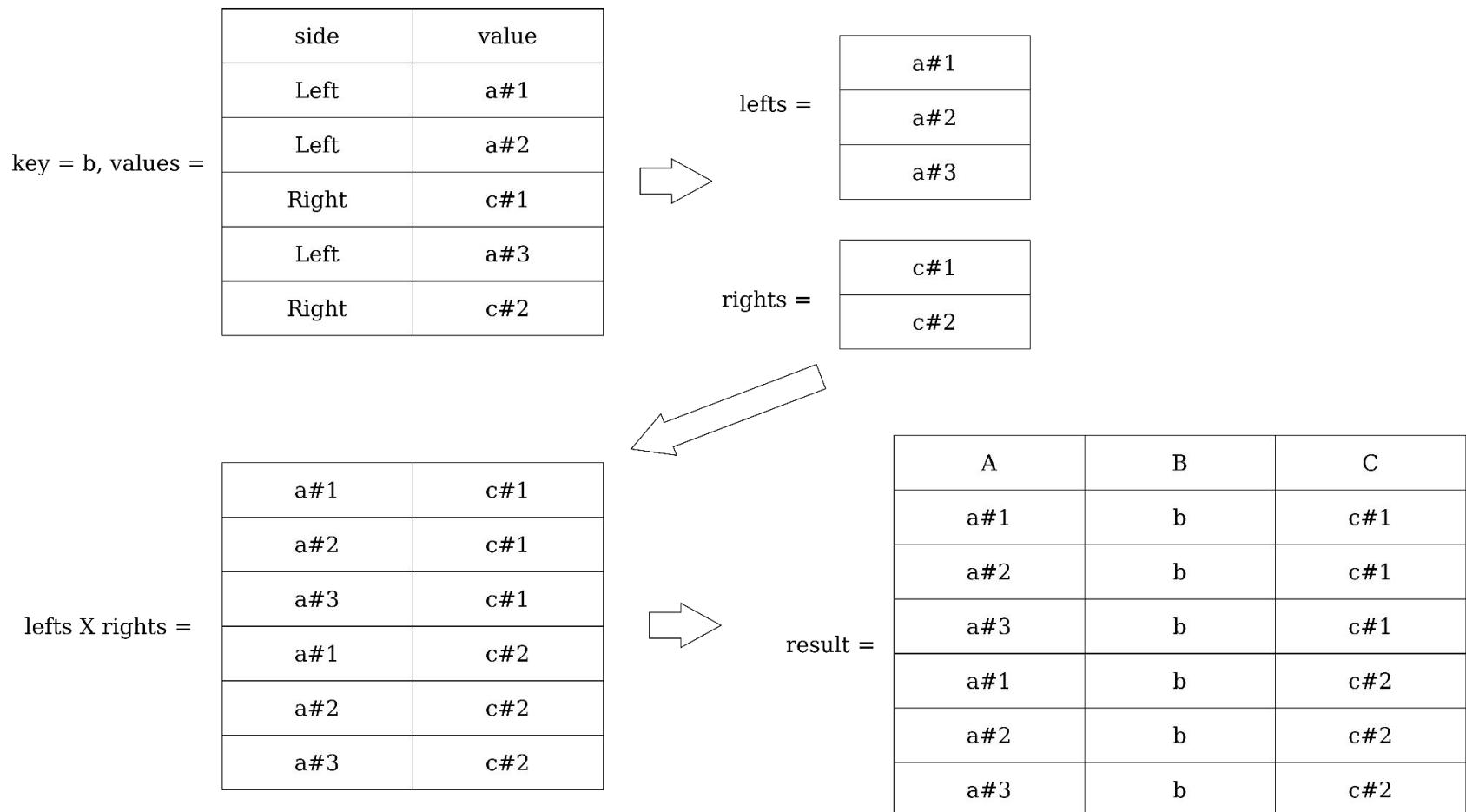
- Для каждой строки левой таблицы тар выдаст пару $\langle \text{key} = \text{Row.B}, \text{value} = (\text{Left}, \text{Row.A}) \rangle$
 - Аналогично для правой:
 $\langle \text{key} = \text{Row.B}, \text{value} = (\text{Right}, \text{Row.C}) \rangle$
- За кадром произойдёт группировка по ключу
 - То есть по значению столбца B
- После группировки ключу соответствует множество строк
 - “Левые” и “правые” строки в перемешку

key = b, values =

side	value
Left	a#1
Left	a#2
Right	c#1
Left	a#3
Right	c#2

Join: алгоритм для MapReduce

- Разделим строки на правые и левые
- Каждая левая строка входит в пару с каждой правой
 - Так как у них одно и то же значение столбца В
- Построим декартово произведение

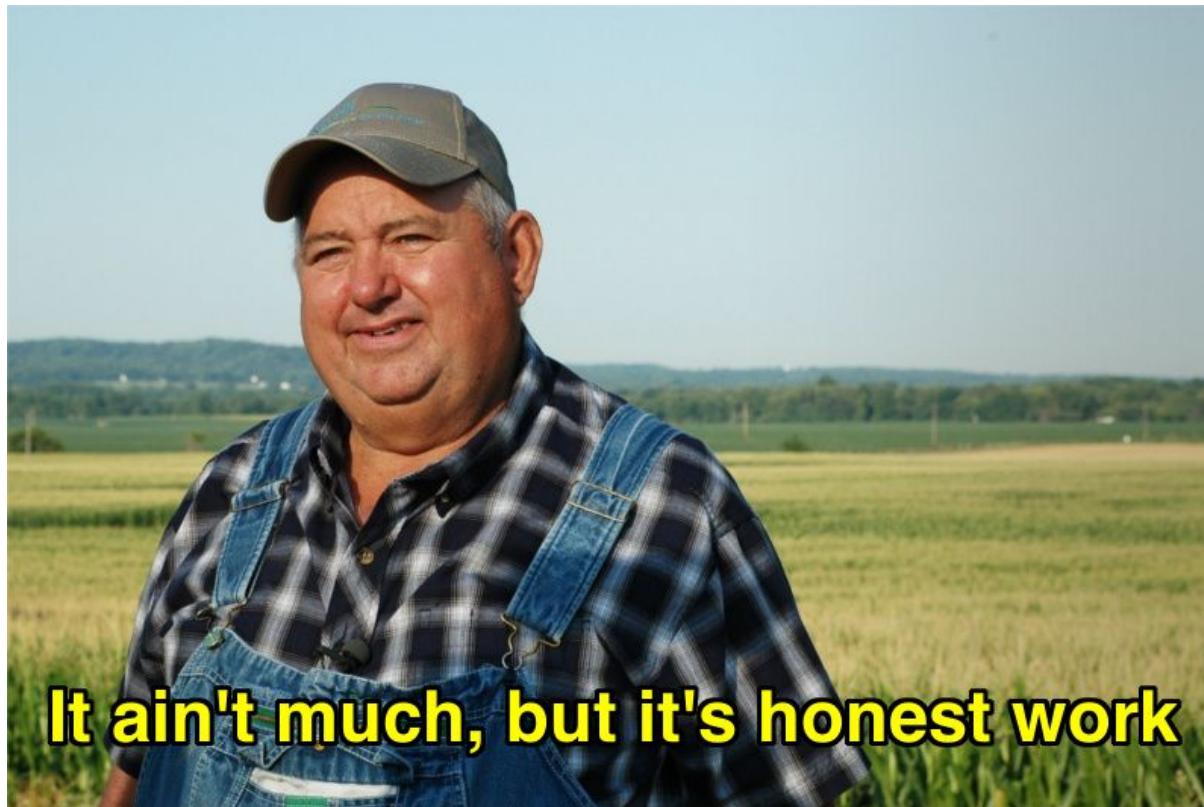


Join: алгоритм для MapReduce

```
1 fun map(table: Table):
2     if table.side = Left:
3         for a, b in table:
4             yield b, {side = Left, value = a}
5     else:
6         for b, c in table:
7             yield b, {side = Right, value = c}
8
9 fun reduce(key, values):
10    lefts = [it.value for it in values if it.side = Left]
11    rights = [it.value for it in values if it.side = Right]
12    for a in lefts:
13        for c in rights:
14            yield a, key, c
```

Join: анализ

- Наша реализация не самая эффективная
 - С точки зрения равномерности распределения нагрузки
 - На распределённых системах обсудим детальнее
- Пока нас устроит и так



tf-idf: вспоминаем былое

- Что такое tf-idf?
 - Вы должны это знать
- $tf(t, d)$ - количество повторений слова t в документе d , делённое на общее количество слов в документе d
- $idf(d)$ -
логарифм
единицы,
делённой на
долю
документов, в
которых
встречается
слово

$$tf(t, d) = \frac{|\{i : d_i = t\}|}{|d|}$$

$$idf(t) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

$$tf\text{-}idf(t, d) = tf(t, d) \cdot idf(t)$$

tf-idf: алгоритм для MapReduce

- Посчитаем $cnt_t_d(t, d) = |\{i : d_i = t\}|$

```
1 fun map(doc: Document):  
2     for word in words(doc):  
3         yield {key=(word, doc.id), value=1}  
4  
5 fun reduce({word, docId}, values):  
6     return word, docId, sum(values)
```

- На самом деле reduce не нужен
 - Достаточно combine с таким же кодом
 - Так как документ целиком лежит всего на единственном узле
 - Можно посчитать любым другим способом на одном узле
 - Главное чтобы в память влезло

tf-idf: алгоритм для MapReduce

- Посчитаем $doc_size(d) = |d|$

```
1 fun map(doc: Document):  
2     return doc.id, words(doc).size()
```

- reduce не нужен
 - Так как документ целиком лежит всего на единственном узле
 - Считаем любым способом

tf-idf: алгоритм для MapReduce

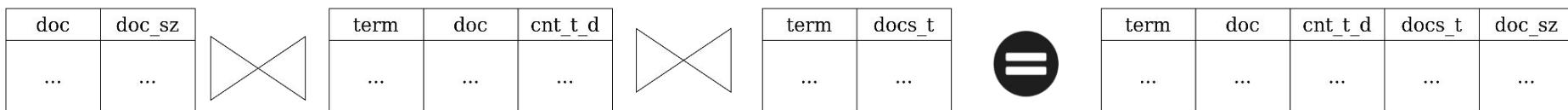
- Посчитаем $docs_t(t) = |\{d \in D : t \in d\}|$

```
1 fun map(doc: Document) -> [(String, Int)]:  
2     for word in words(doc):  
3         yield word, doc.id  
4  
5 fun reduce(word: String, docIds: [Int]):  
6     return word, set(docsCounts).size()
```

- Без редьюса не обойтись
 - Так как терм может лежать на нескольких узлах
- Можно применить combiner чтобы уменьшить количество передаваемых данных
 - Схлопнуть все пары (word=слово, docId=42) в одну

tf-idf: объединяем результаты

- Есть таблицы:
 - (doc, doc_size)
 - (term, doc, count_term_doc)
 - (term, docs_count_by_term)
 - Сделаем 2 соединения

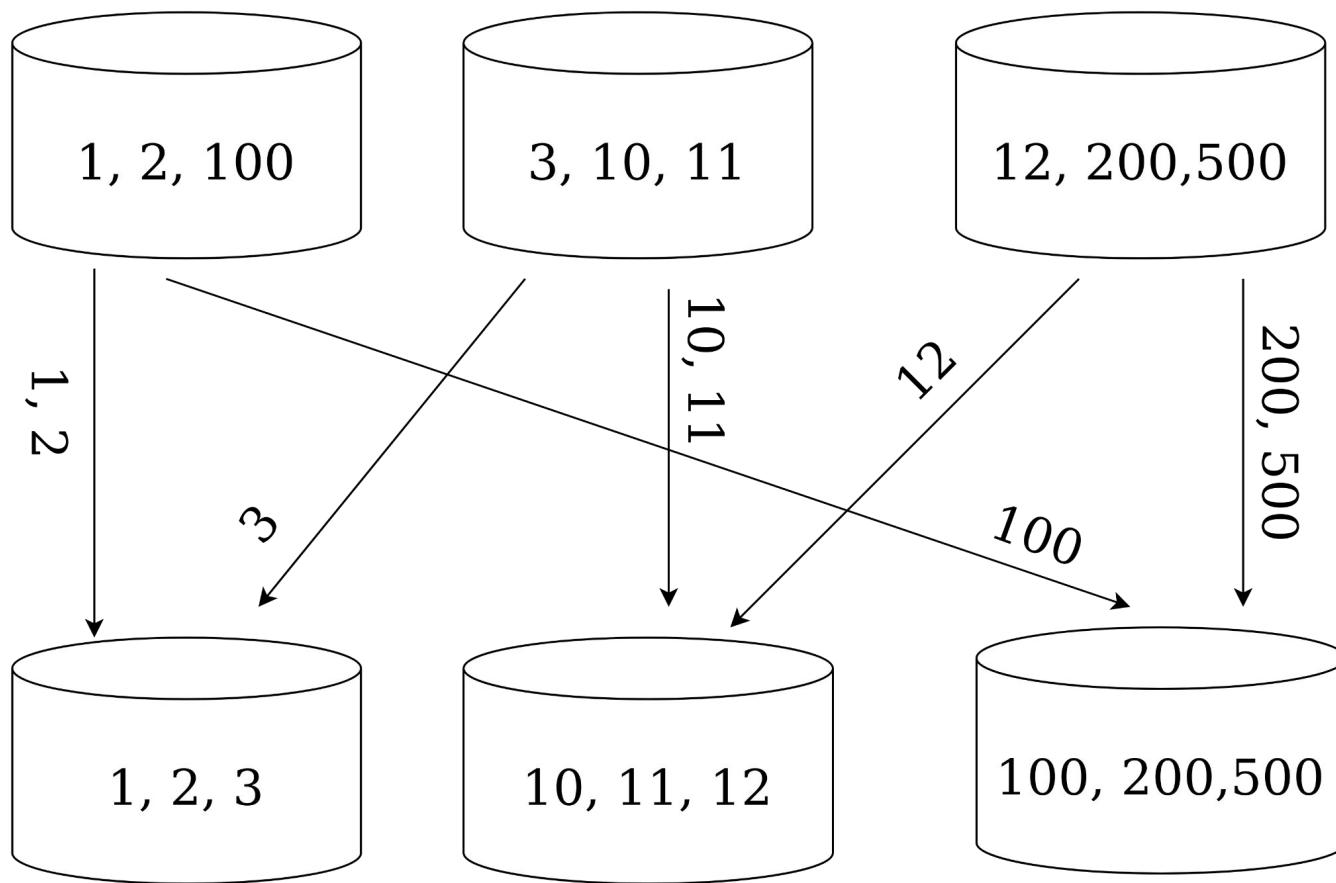


- Получили таблицу
(doc, term, cnt_term_doc, docs_cnt_by_term, doc_sz)
- Для каждой строки результирующей таблицы считаем

$$tf-idf(t, d) = \frac{count_term_doc}{doc_size} \cdot \log \frac{|D|}{docs_count_by_term}$$

Распределённая сортировка

- До: ~~карты~~ ключи расположены в ~~другом~~ произвольном порядке по узлам
 - Вы их в киосках что ли заряжаете?
- После: $\forall a \in D_i, b \in D_j, i \leq j : a \leq b$



Распределённая сортировка: алгоритм

- Будем выбирать редьюсера в зависимости от интервала, в который попадает ключ

$$a_1 < a_2 < a_3 < \dots < a_{n-1}$$

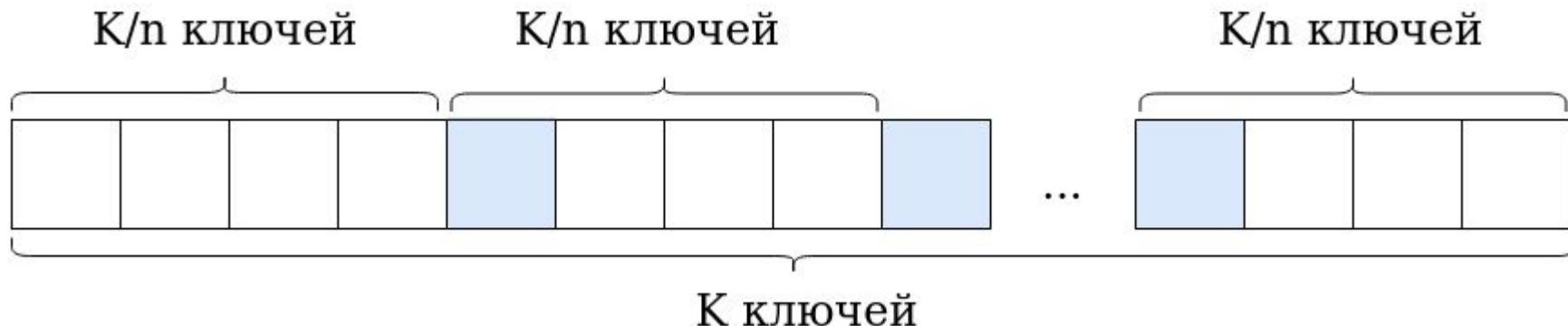
$$r(x) = \begin{cases} 1 & x < a_1 \\ 2 & a_1 \leq x < a_2 \\ 3 & a_2 \leq x < a_3 \\ \dots \\ n & x \geq a_{n-1} \end{cases}$$

Распределённая сортировка: определение границ

- Границы должны быть выбраны так, чтобы на каждого редьюсера пришлось примерно равное число ключей

$$\mathbb{P}(x < a_1) = \mathbb{P}(a_1 \leq x < a_2) = \mathbb{P}(a_2 \leq x < a_3) = \dots = \mathbb{P}(x \geq a_{n-1})$$

- Пусть у нас всего M ключей
- Выберем примерно K случайным образом
 - Можно ровно K , но это чуть сложнее
 - И не очень нужно



Определение границ

- Все ключи попадут на единственного редьюсера
 - Выберем достаточно маленькое K
- Редьюсер отсортирует ключи и выберет границы

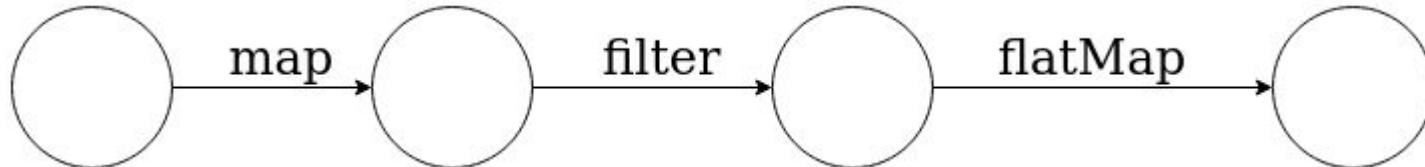
```
1 fun map(doc: [Key]) -> [(String, Key)]:  
2     for key in doc:  
3         if random.uniform(0, 1) <  $\frac{K}{M}$ :  
4             yield "key", key
```

Spark: мотивация

- Работать с данными в функциональном стиле удобно

```
1 result = datasource.fetch()  
2             .map(...)  
3             .filter(...)  
4             .flatMap(...)  
5             .collect(...)
```

- Для MapReduce получаем граф вычислений



- Каждое ребро преобразует набор входных файлов в набор выходных
 - Много ненужного дискового IO
 - Много ненужных вычислений

Ленивые вычисления: как делать не нужно

```
1 result = datasource.fetch()  
2             .map(...)  
3             .filter(...)  
4             .flatMap(...)  
5             .collect(...)
```

- На каждом шаге вычисляем результат

```
1 data = datasource.fetch()  
2  
3 mapped = []  
4 for elem in data:  
5     mapped.add(f(elem))  
6  
7 filtered = []  
8 for elem in mapped:  
9     if p(elem):  
10         filtered.add(elem)  
11  
12 result = []  
13 for elem in filtered:  
14     result.addAll(g(elem))
```

Ленивые вычисления: как делать нужно

```
1 result = datasource.fetch()  
2             .map(...)  
3             .filter(...)  
4             .flatMap(...)  
5             .collect(...)
```

- Считаем результат лениво

```
1 result = []  
2 for elem in datasource.fetch():  
3     mapped = f(elem)  
4     if p(mapped):  
5         result.addAll(g(mapped))
```

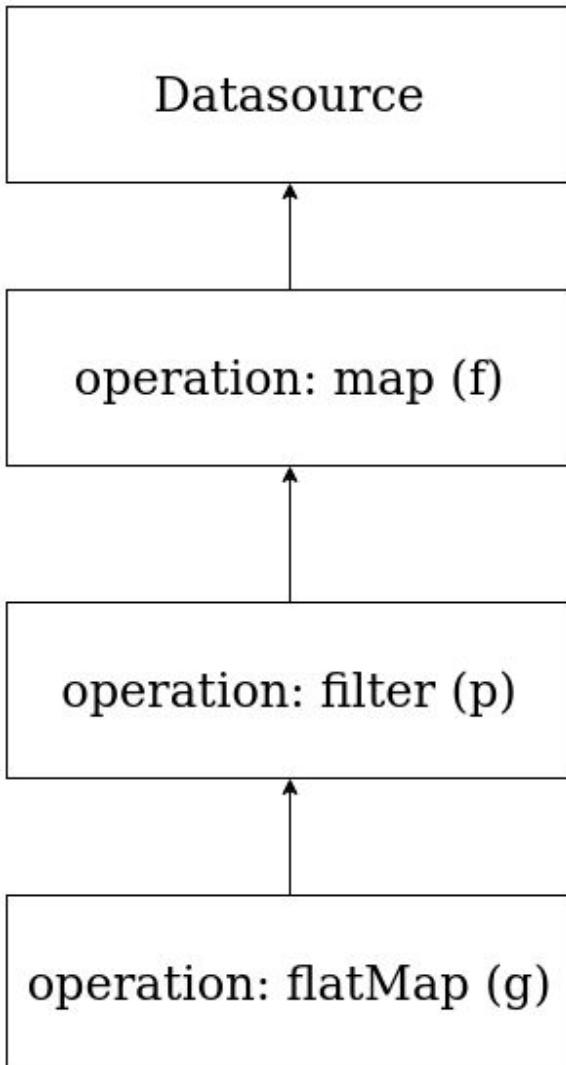
Знакомая концепция

- Java Streams
- C++20 Ranges
- Haskell Stream Fusion
- Python Generators
- ...



Реализация

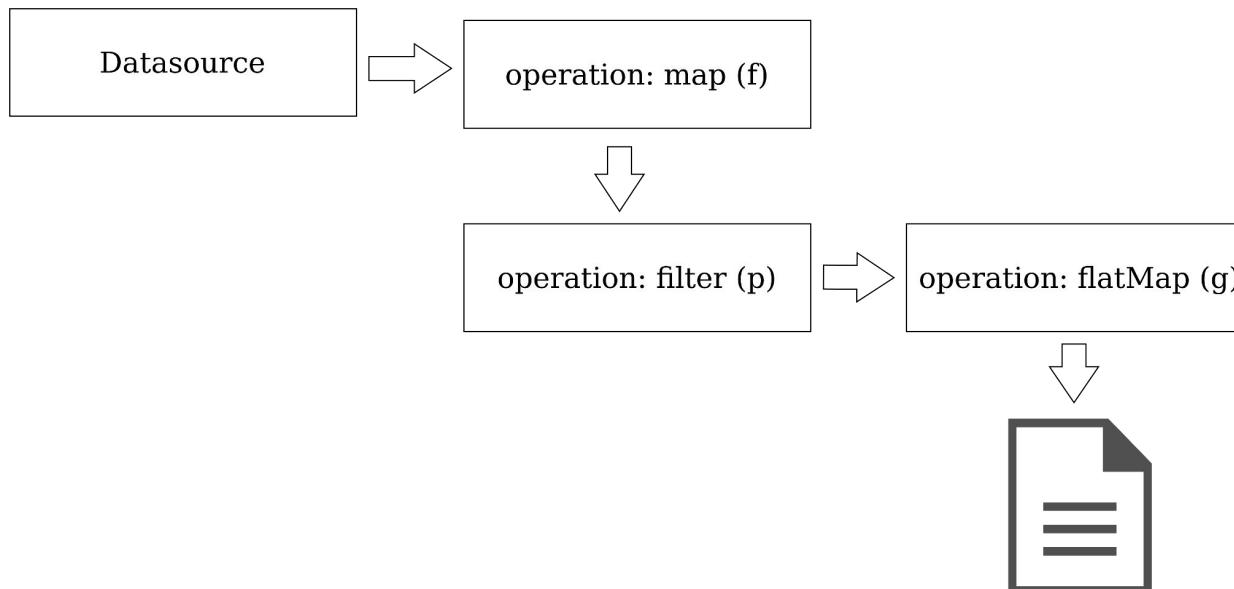
```
1 result = datasource.fetch()  
2 .map(...)  
3 .filter(...)  
4 .flatMap(...)  
5 .collect(...)
```



- Ленивый контейнер не хранит данные
 - Только операцию, с помощью которой он был получен
 - И **список ленивых контейнеров**, из которых он был получен

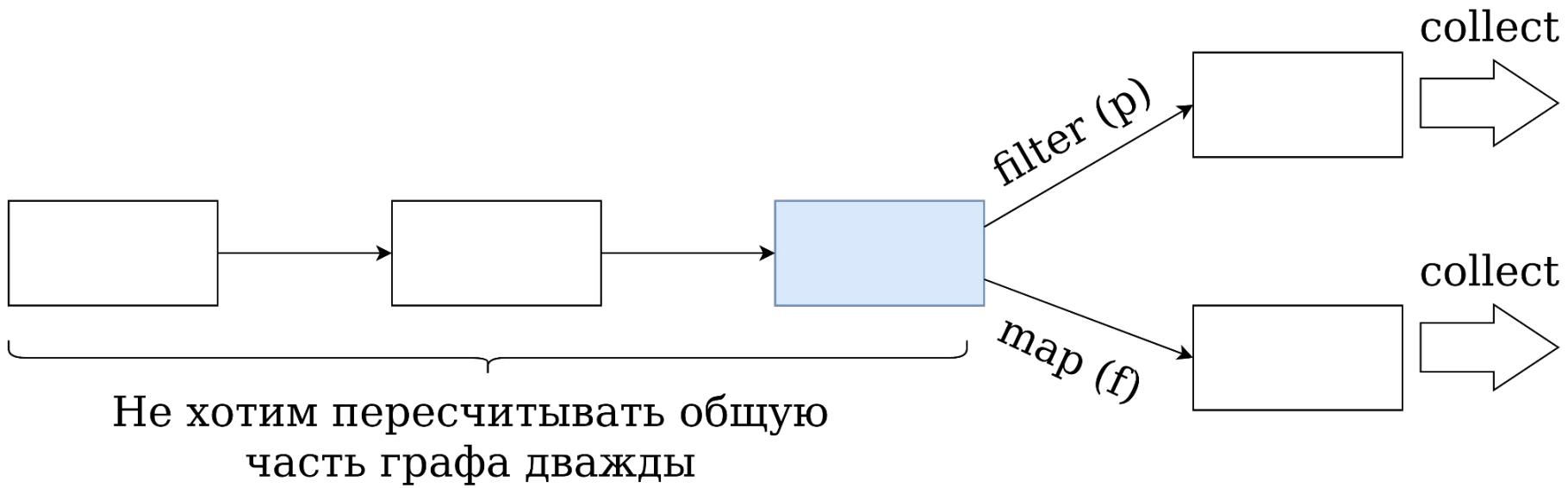
Когда происходит выполнение операций?

- Когда пользователь выполняет терминирующую операцию
 - `count()`
 - `collect()`
 - ...
- Строки исходного датасета начинают проходить через граф
 - К ним применяются операции



Материализация датасетов

- Иногда датасет нужно в явном виде посчитать и сохранить (“материализовать”)
 - На диске или в памяти
 - Например, если к нему планируется сделать несколько запросов, и пересчитывать весь датасет на каждый запрос не хочется
 - Некоторые операции (сортировка) этого требуют



А что ещё есть?

- Dryad
 - Строим ациклический граф вычислений
 - Вычисляем его на нескольких узлах
 - Очень похоже на Spark
 - Разработан Microsoft Research
 - На текущий момент мёртв
 - Последний коммит в 2014 году

А что ещё есть?

- Pig
 - Язык обработки данных
 - Основан не на функциональных комбинаторах, а на собственном синтаксисе
 - Программы на этом языке компилируются в цепочки MapReduce-задач
 - На текущий момент практически умер
 - Последний коммит полтора года назад
 - Последний релиз в 2017
 - Я видел живого человека, использующего Pig

```
A = LOAD 'data' AS (f1,f2,f3);
B = FOREACH A GENERATE f1 + 5;
C = FOREACH A generate f1 + f2;
```

```
salesinp = LOAD '/pig/data/salesdata' USING PigStorage(',') AS
    (product:chararray, year:int, region:chararray, state:chararray, city:chararray, sales:long);
cubedinp = CUBE salesinp BY CUBE(product,year);
result = FOREACH cubedinp GENERATE FLATTEN(group), SUM(cube.sales) AS totalsales;
```



А что ещё есть?

- Hive
 - Вытеснил Pig (более-менее активно развивается)
 - Компилируем в цепочки MapReduce-задач не какой-то новый язык обработки данных
 - А SQL
 - Его знают программисты, аналитики, ML-инженеры
 - Даже ваш домашний хомяк



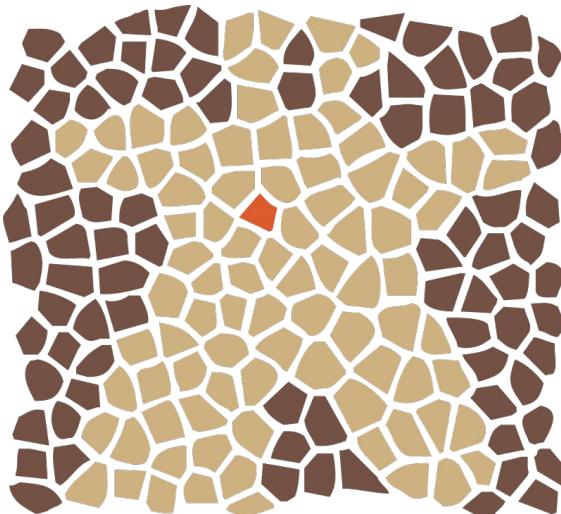
```
SELECT customer.customer_num,
       (SELECT SUM(ship_charge)
        FROM orders
        WHERE customer.customer_num = orders.customer_num
       ) AS total_ship_chg
  FROM customer
```

```
SELECT a.key, a.value
  FROM a
 WHERE a.key in
       (SELECT b.key
        FROM B);
```

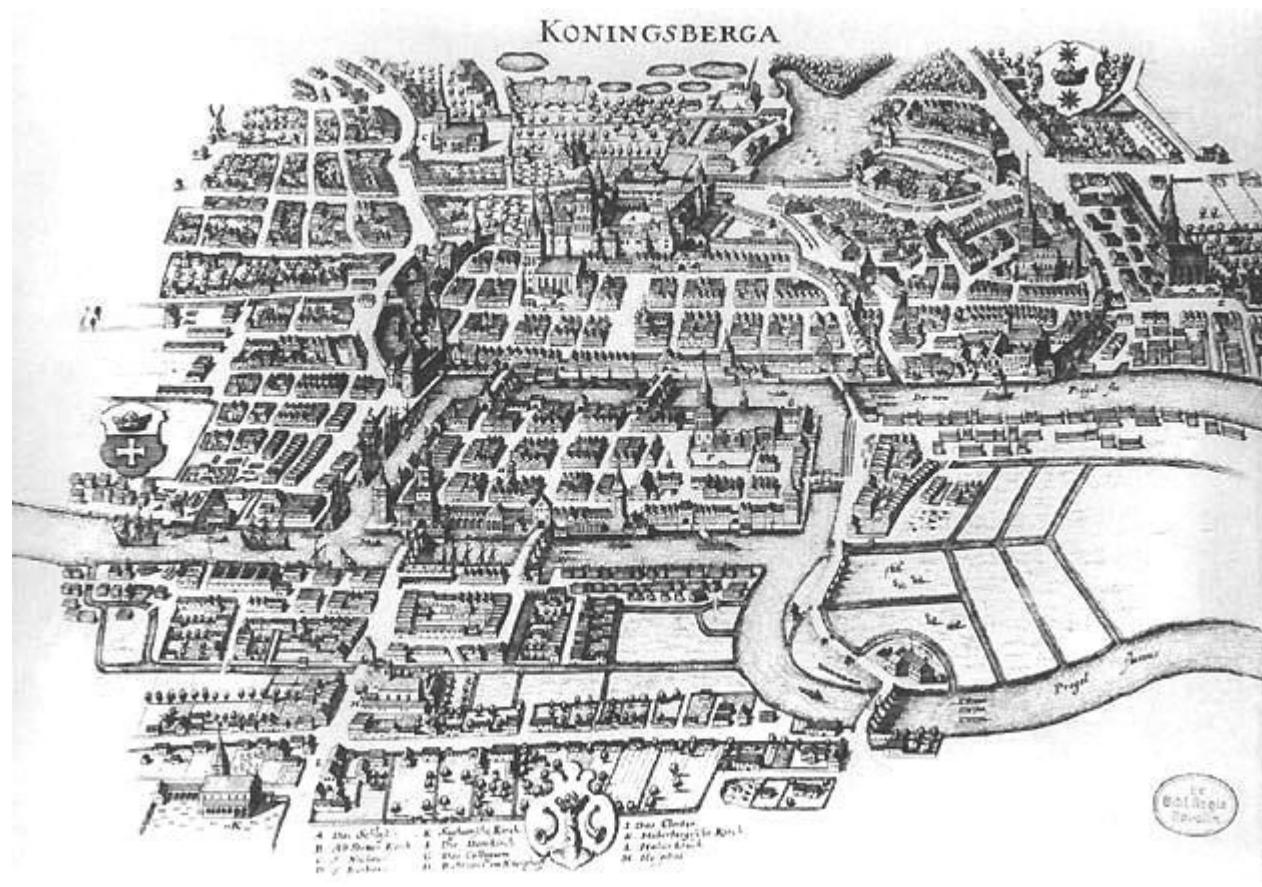
```
SELECT a.val1, a.val2, b.val, c.val
  FROM a
 JOIN b ON (a.key = b.key)
 LEFT OUTER JOIN c ON (a.key = c.key)
```

А что ещё есть?

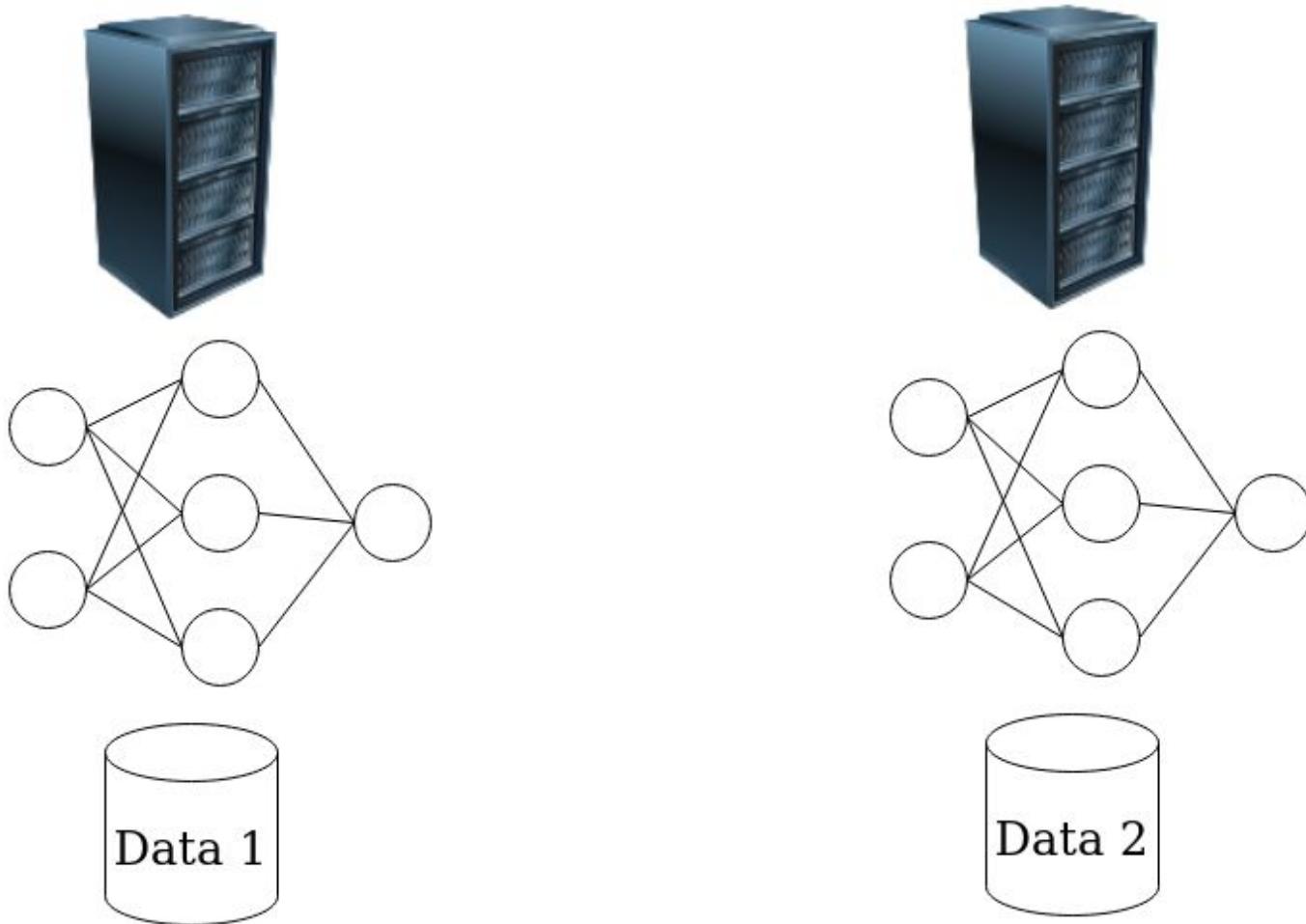
- Pregel
 - Распределённая обработка графов
 - Разработан в Google
 - Есть свободная альтернатива - Apache [Giraph](#)



А Р А С Н Е
GIRAPH



Распределённое машинное обучение: общий принцип



<https://2020.hydraconf.com/2020/msk/talks/1pztdbhwxwel2ojzdudehs/>

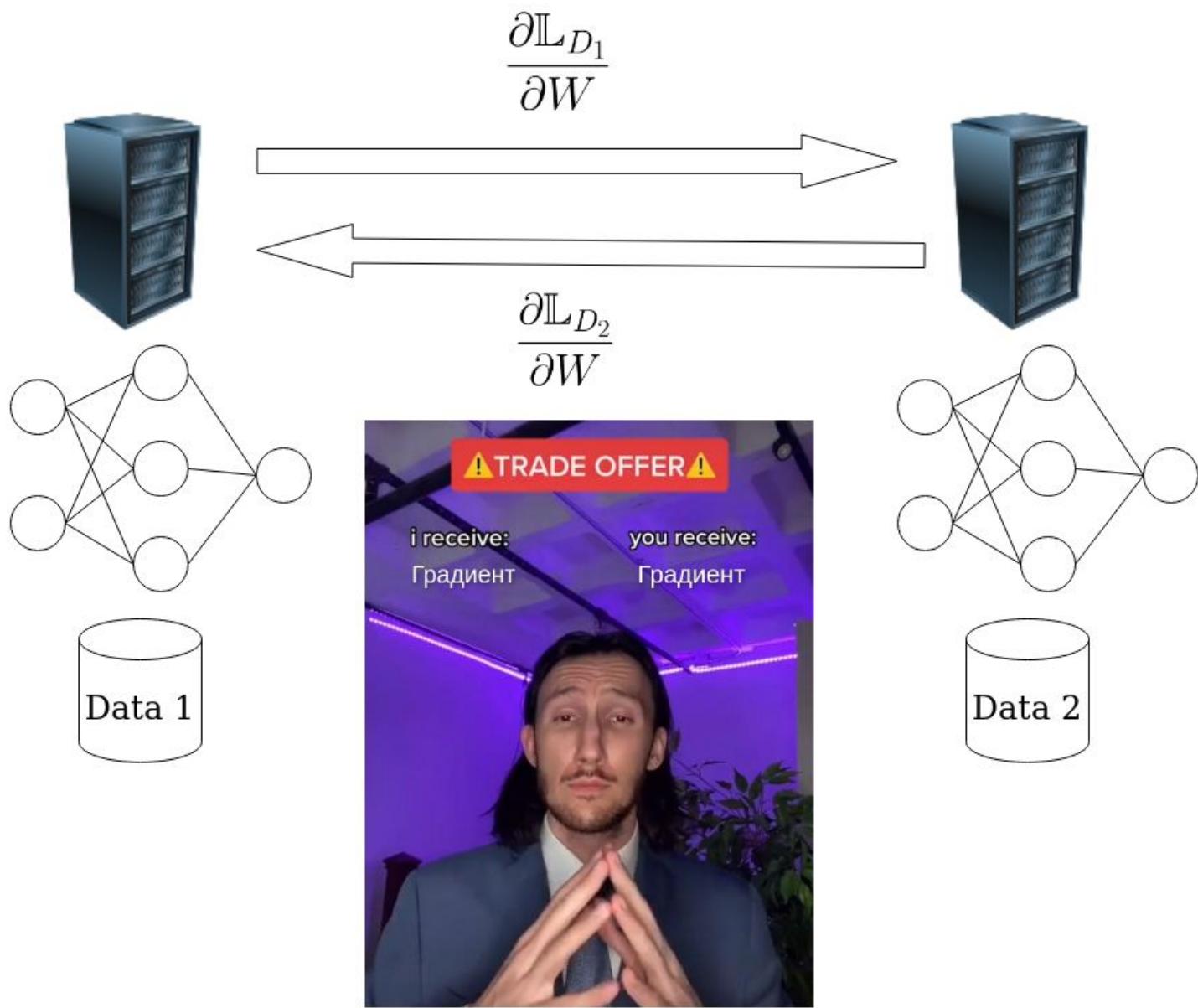
Распределённое машинное обучение: разбиение градиента

$$D_i \subset D : \mathbb{L}_{D_i} = \sum_{x,y \in D_i} L(x, y, \hat{y})$$

$$\mathbb{L} = \sum_{x,y \in D} L(x, y, \hat{y}) = \sum_i \sum_{x,y \in D_i} L(x, y, \hat{y}) = \sum_i \mathbb{L}_{D_i}$$

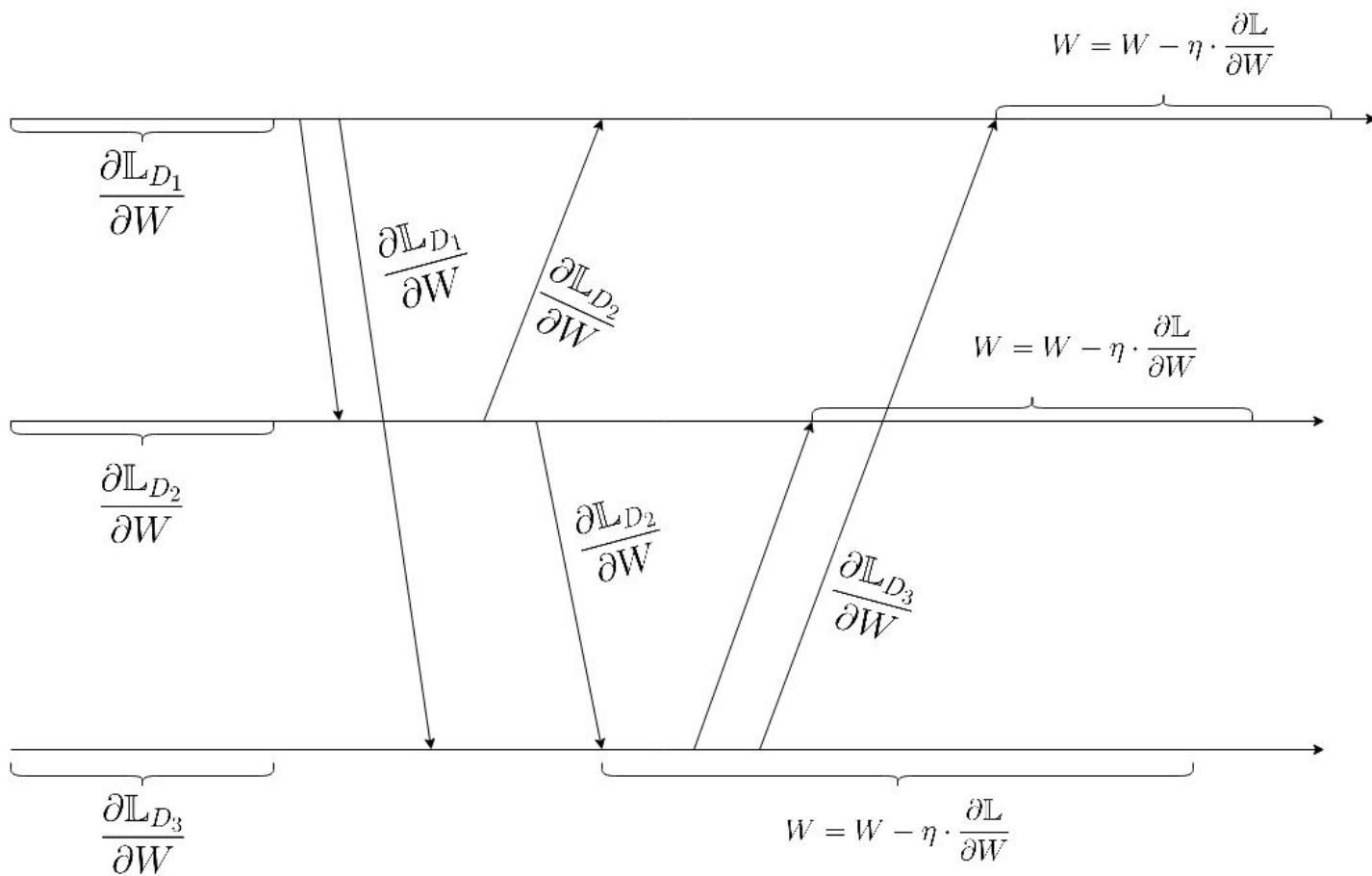
$$\frac{\partial \mathbb{L}}{\partial W} = \frac{\partial \left(\sum_i \mathbb{L}_{D_i} \right)}{\partial W} = \sum_i \frac{\partial \mathbb{L}_{D_i}}{\partial W}$$

Распределённое машинное обучение: алгоритм



Распределённое машинное обучение: алгоритм

- Каждый узел посылает каждому своему часть градиента
- После чего каждый узел вычисляет итоговый градиент
- И вычисляет новые параметры модели



Распределённое машинное обучение: масштабирование

- Если в системе N узлов - каждый узел должен послать свой градиент каждому
 - $N * (N - 1) * |W|$ байт требуется пересылать на каждый раунд
- Сколько работы должен сделать каждый узел?
 - Вычисление градиента и обновление весов:

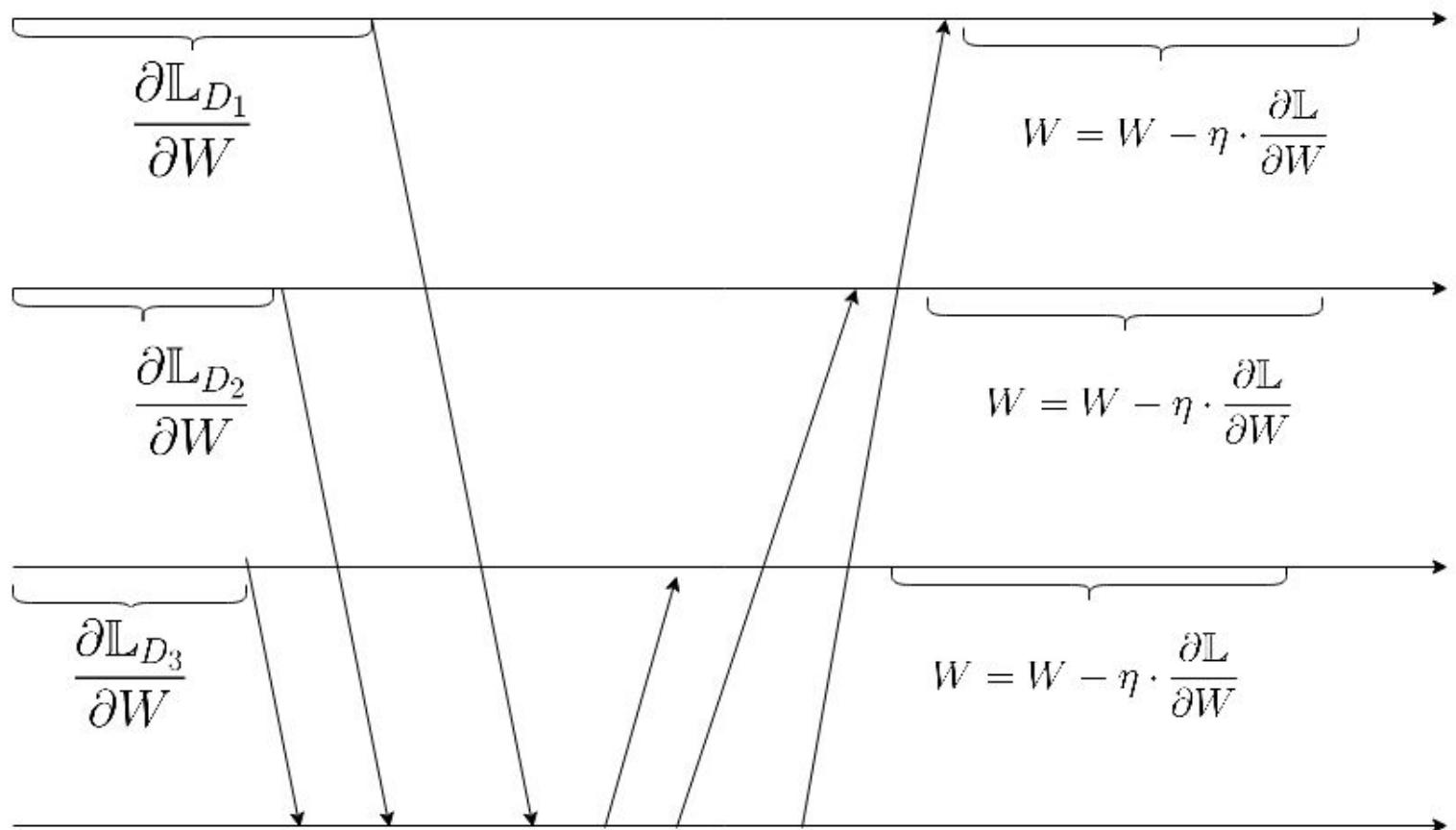
$$O\left(|W| \cdot \frac{|D|}{N}\right)$$

- Пересылка градиента:

$$O(|W| \cdot N)$$

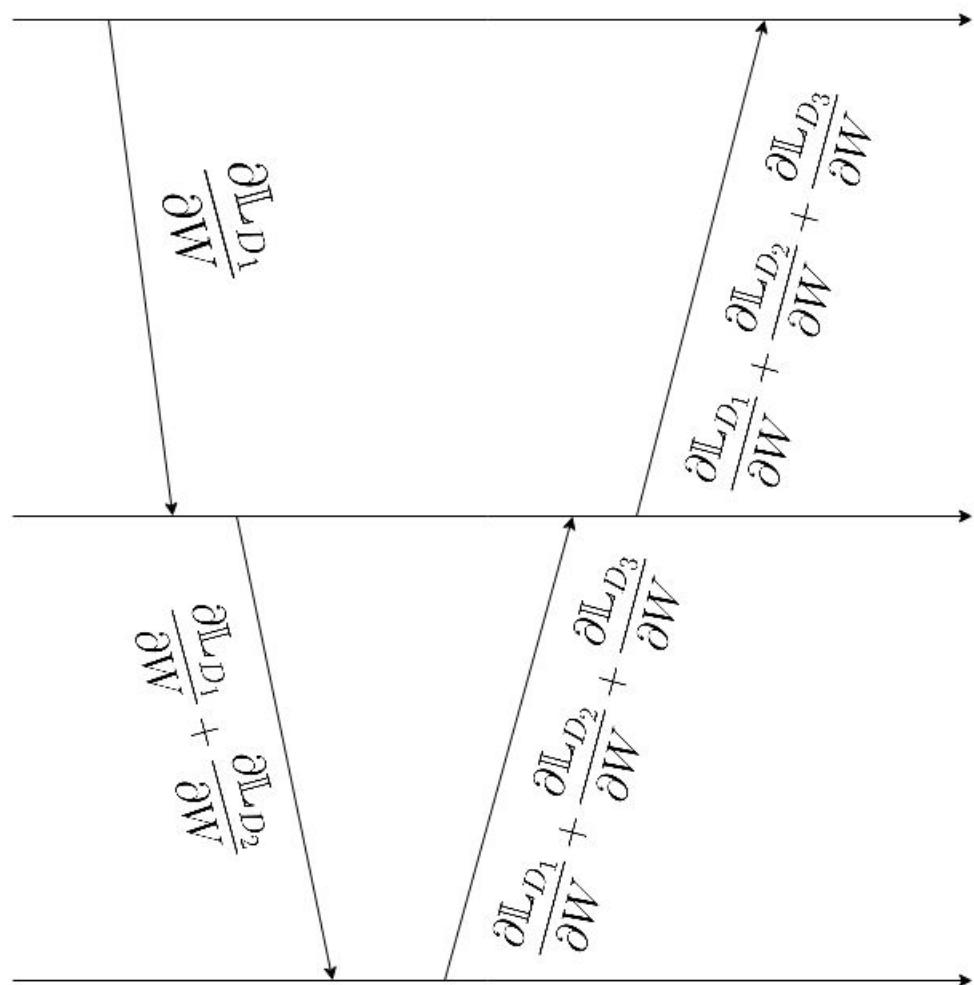
Схемы пересылки сообщений: пересылка через мастера

- Каждый узел (кроме мастера) пошлёт всего $O(|W|)$ бит
 - Вместо $O(|W| * N)$
 - В системе будет $2 * (N - 1)$ сообщений вместо $N * (N - 1)$



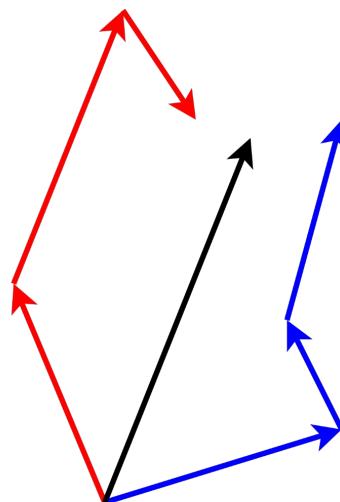
Схемы пересылки сообщений: пересылка по кругу

- Не нужен выделенный мастер
 - Децентрализованный протокол
 - Посыпаем $2 * (N - 1)$ сообщений



Обмен весами

- Каждый узел независимо учит сеть K ходов
 - Только на своих данных
- Потом все узлы обмениваются весами
 - А не градиентами
- И усредняют веса
- Посыпаем в K раз меньше данных



→ Траектория параметров модели на узле 2

→ Траектория параметров модели на узле 1

→ Траектория параметров модели после усреднения

Осталось за кадром (но будет рассмотрено на распределённых системах!)

- Тонкости реализации Map-Reduce
 - Когда и как пересылать данные
 - Как обрабатывать сбои узлов
- Другие оптимизации Map-Reduce
 - Оптимизация циклов
 - Кэширование многократно используемых результатов вычислений
 - Шардирование
- Распределённые алгоритмы на графах
- Оптимизация распределённых алгоритмов
- Компиляция высокоуровневых языков обработки данных в Map-Reduce
- Оптимизации распределённого градиентного спуска
- Графовые системы распределённых вычислений

Приходите на “Распределённые системы” в следующем семестре!

