

Распределённые вычисления

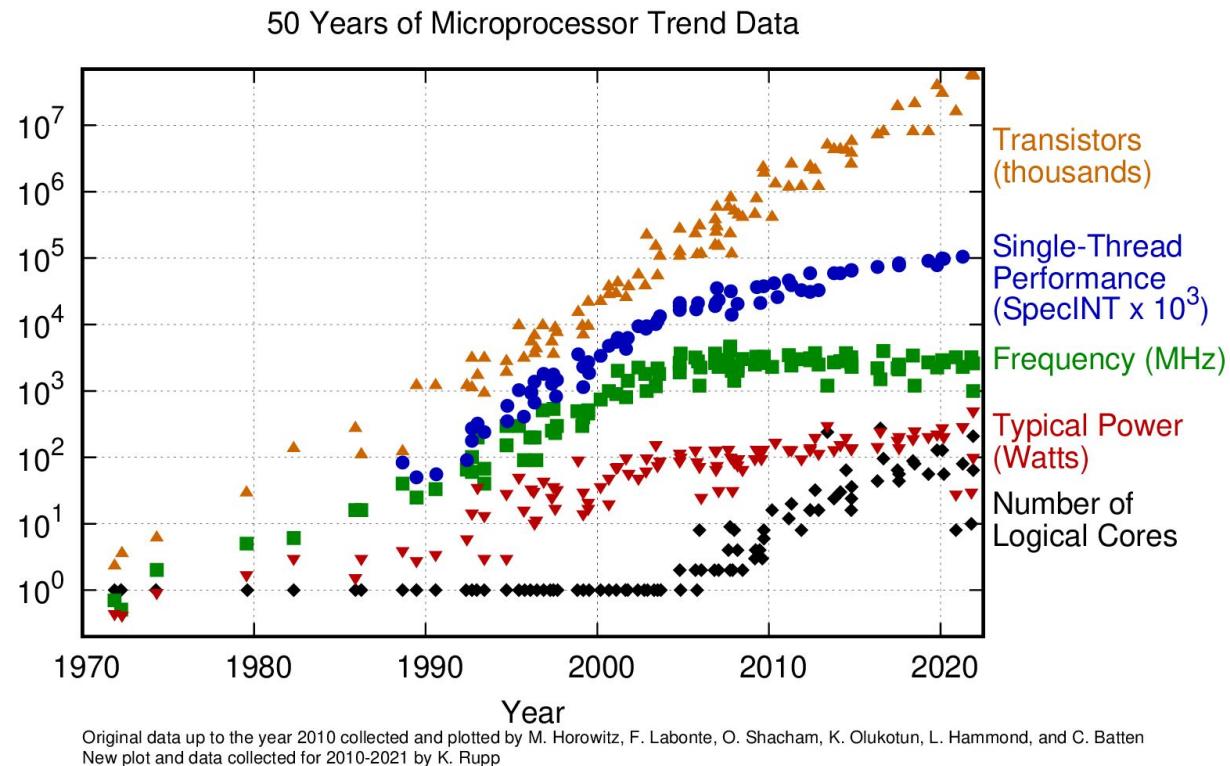


Илья Кокорин

kokorin.ilya.1998@gmail.com

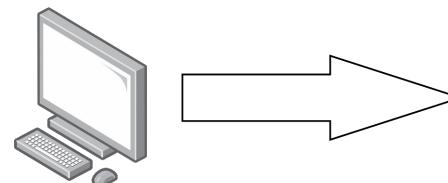
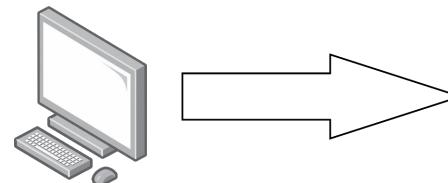
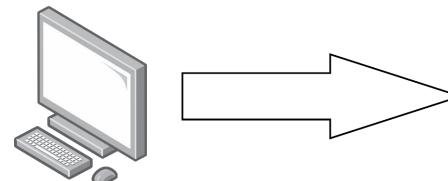
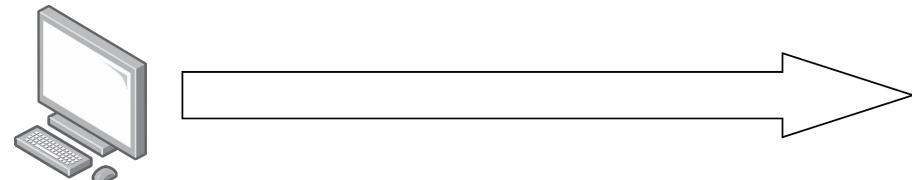
Мотивация: Free Lunch is Over

- Скорости
одного
компьютера
уже не хватает



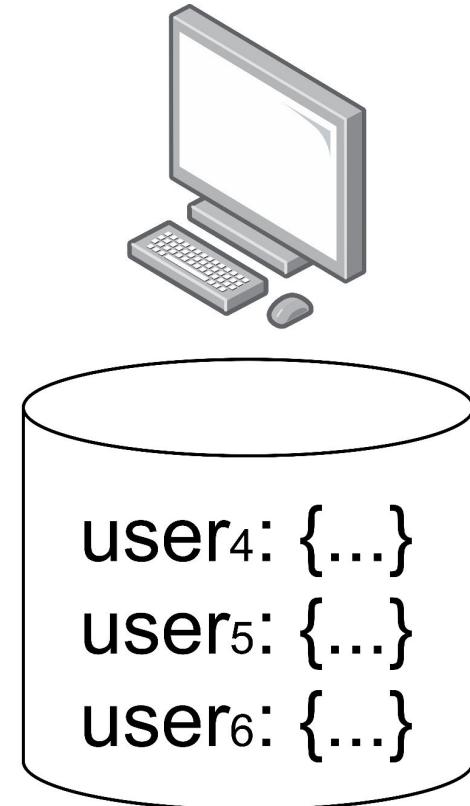
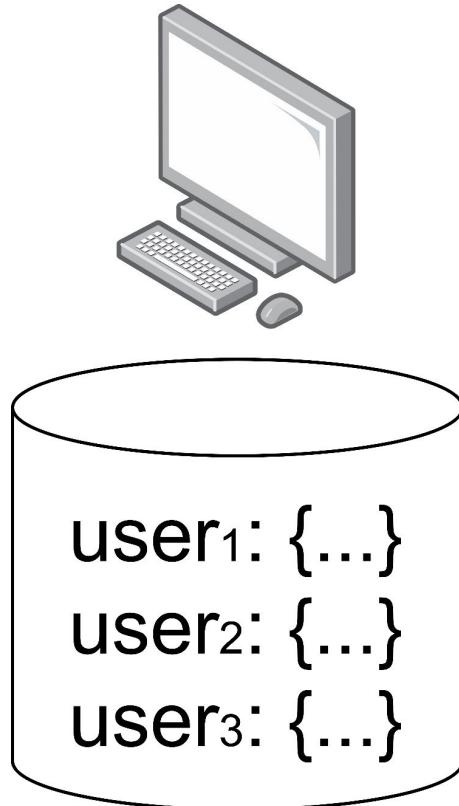
Мотивация: параллельные вычисления

- Разделим работу на несколько компьютеров и сделаем её быстрее



Мотивация: размер данных

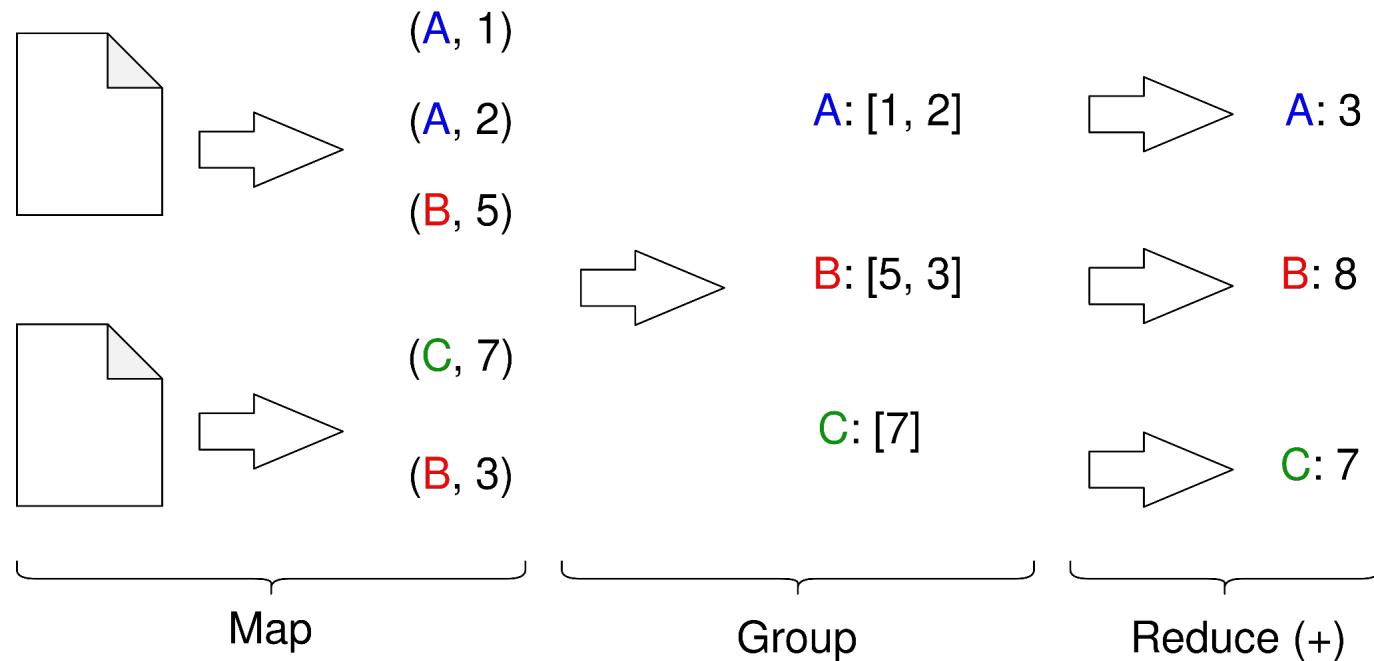
- Данные перестают влезать на один сервер
- Та самая Big Data



MapReduce: модель

- Для каждого документа выдаём набор пар $(key, value)$
- Группируем пары по ключу, получаем $(key, [values...])$
- К значениям

каждой
группы
применяем
функцию



Подсчёт встречаемости слов

- За кадром происходит группировка по слову
- Каждая единичка — одно вхождение слова в документ
- Количество единичек — суммарное количество вхождений слова

```
1 fun mapper(doc):  
2     for word in words(doc):  
3         yield < word, 1 >  
4  
5 fun reducer(word, ones):  
6     yield < 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;
звезда: [1] => 1;
ледяная: [1] => 1;
миллион: [1] => 1;
год: [1] => 1;
никто: [1] => 1;
ничто: [1] => 1;

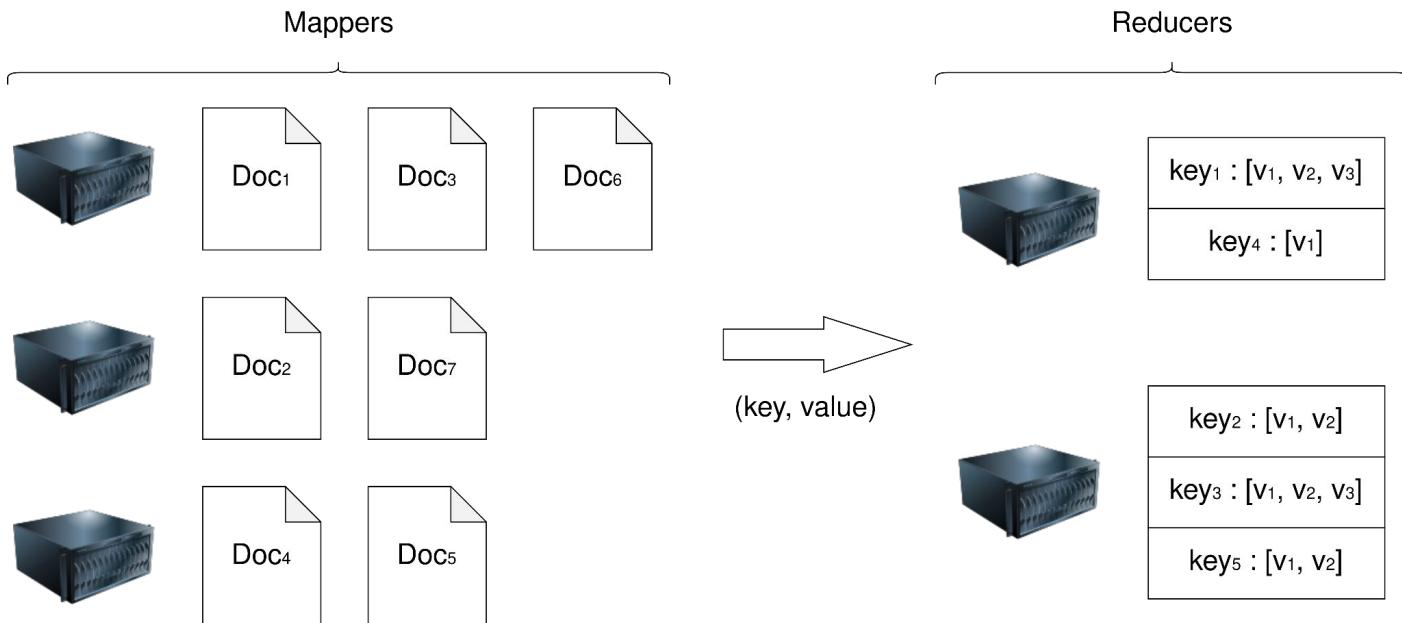
MapReduce: последовательная реализация

- Легко и просто

```
1 fun map_reduce(docs..., mapper, reducer):  
2     kvs = {}  
3     for doc in docs:  
4         for key, value in mapper(doc):  
5             if key ∉ kvs:  
6                 kvs[key] = []  
7                 kvs[key].append(value)  
8             result = []  
9             for key, values... in kvs:  
10                 result.append(reducer(key, values))  
11             return result
```

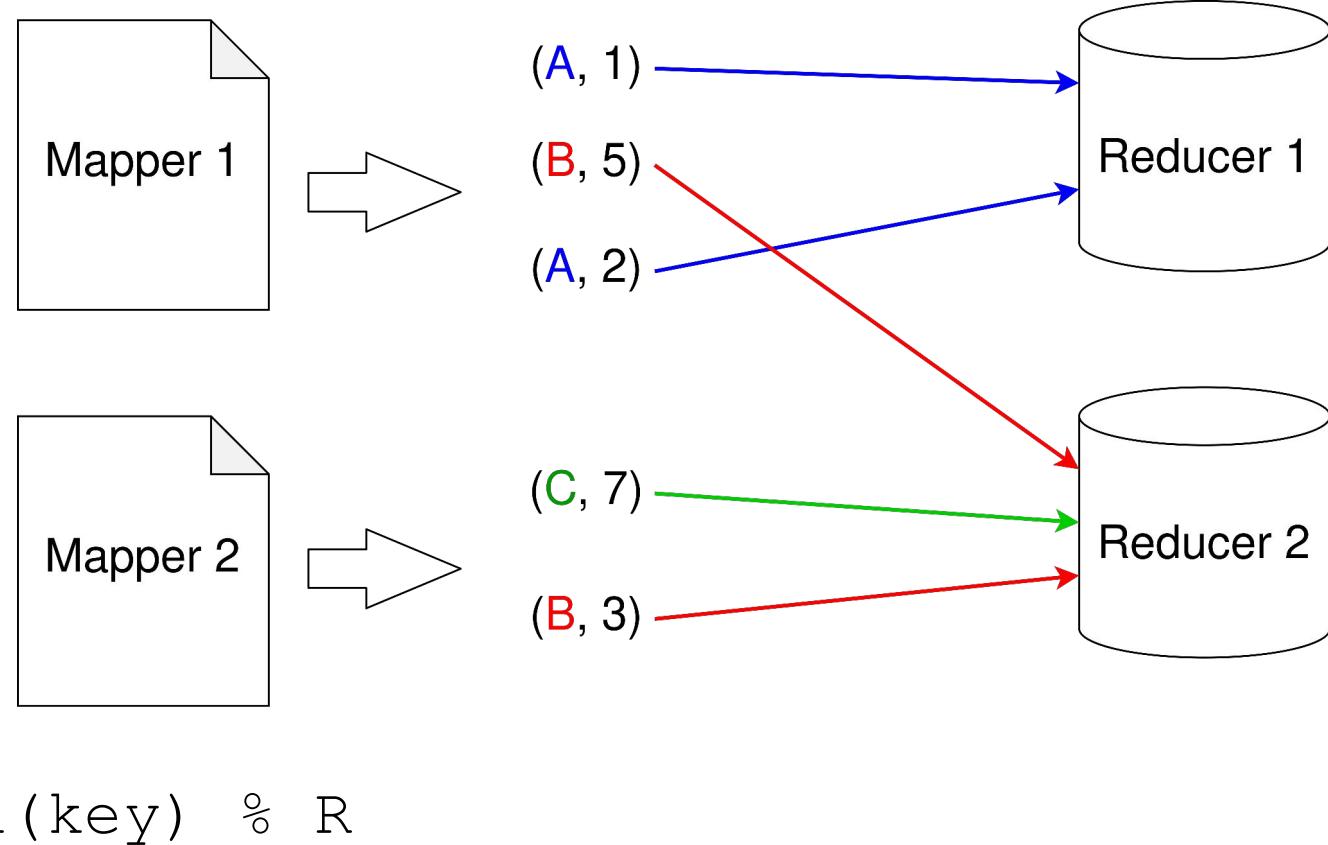
MapReduce: распределённая реализация

- В системе есть M процессов-мапперов
- Каждый документ обрабатывается одним из них
- И R процессов-редьюсеров
- reduce для каждого ключа считается одним из них



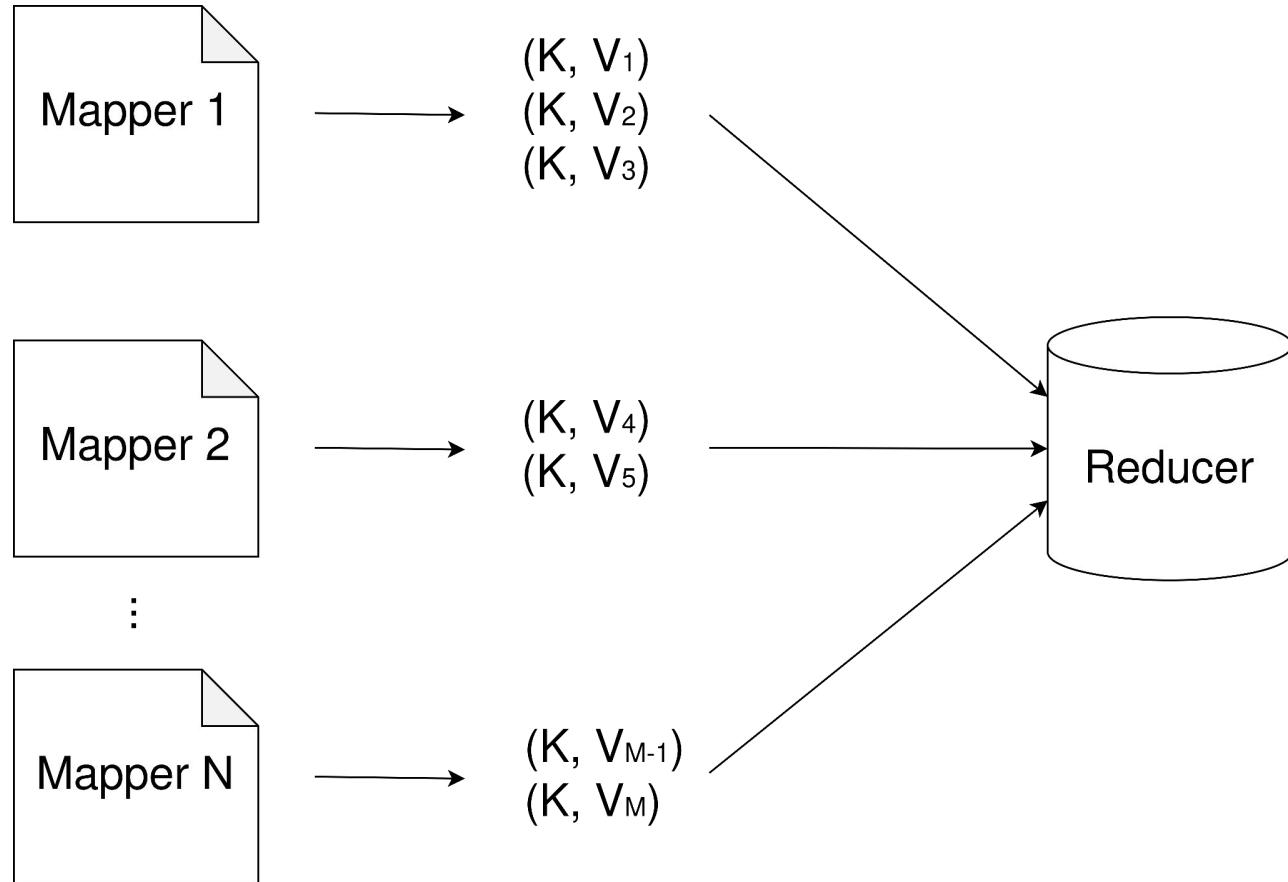
MapReduce: распределённая реализация

- Все значения для ключа K должны попадать на одного и того же редьюсера
- Однозначно выбираем редьюсера по ключу
- $r_id := \text{hash}(\text{key}) \% R$



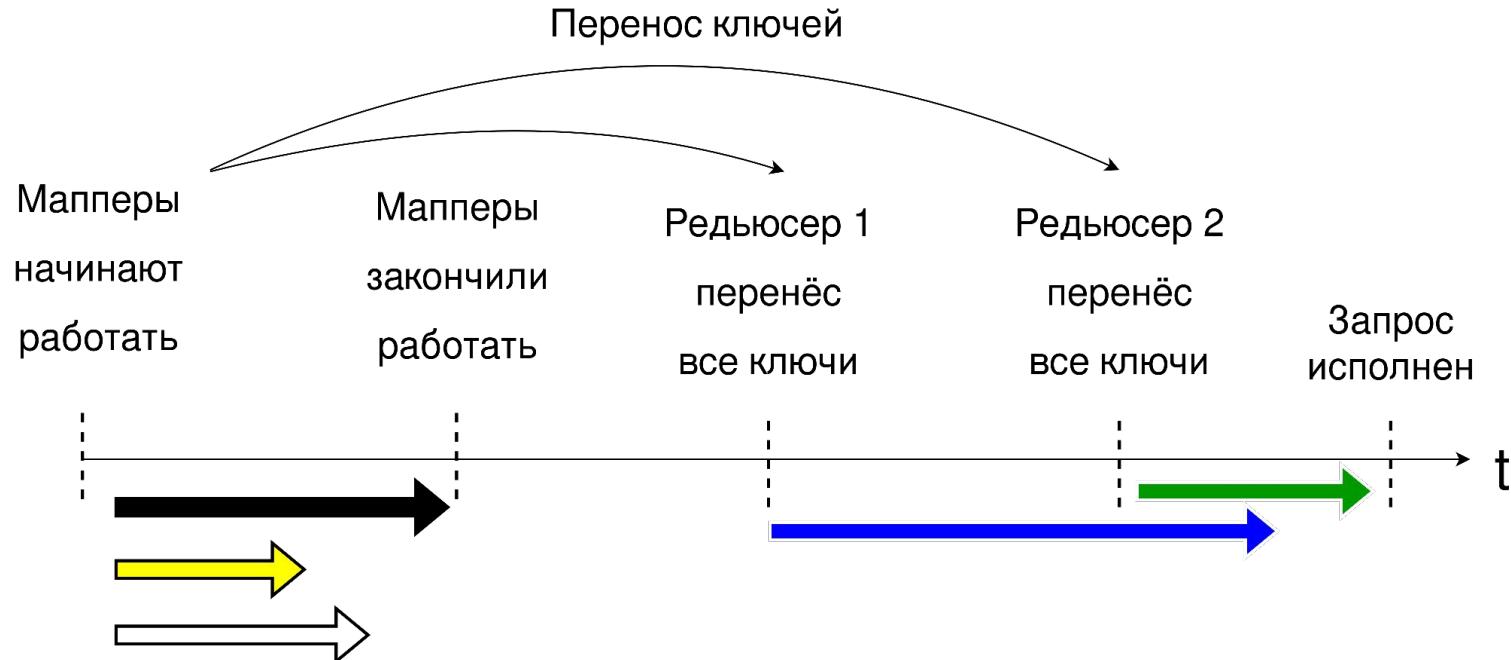
MapReduce: распределённая реализация

- Редьюсер
должен
собрать
значения со
всех мапперов
перед началом
работы



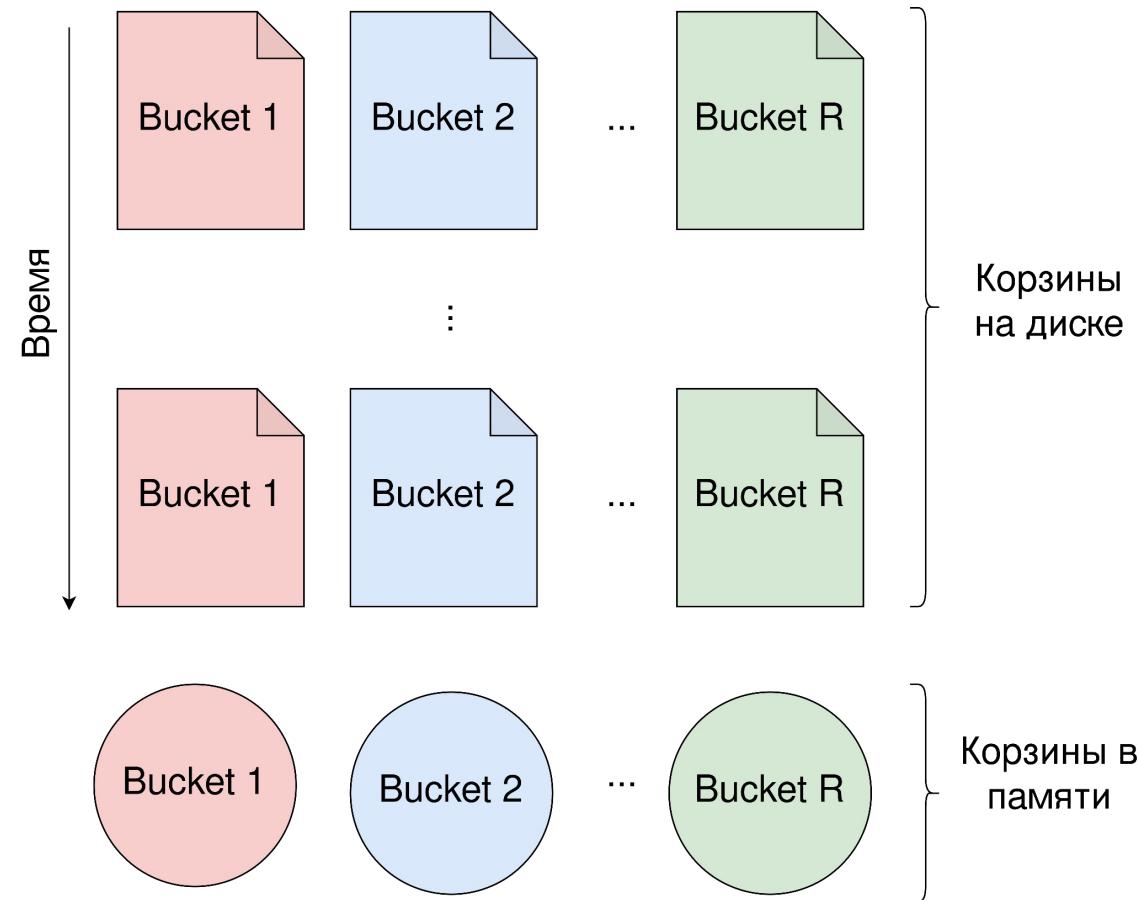
MapReduce: распределённая реализация

- Редьюсер может начать работать только после переноса всех ключей
- После конца работы всех мапперов



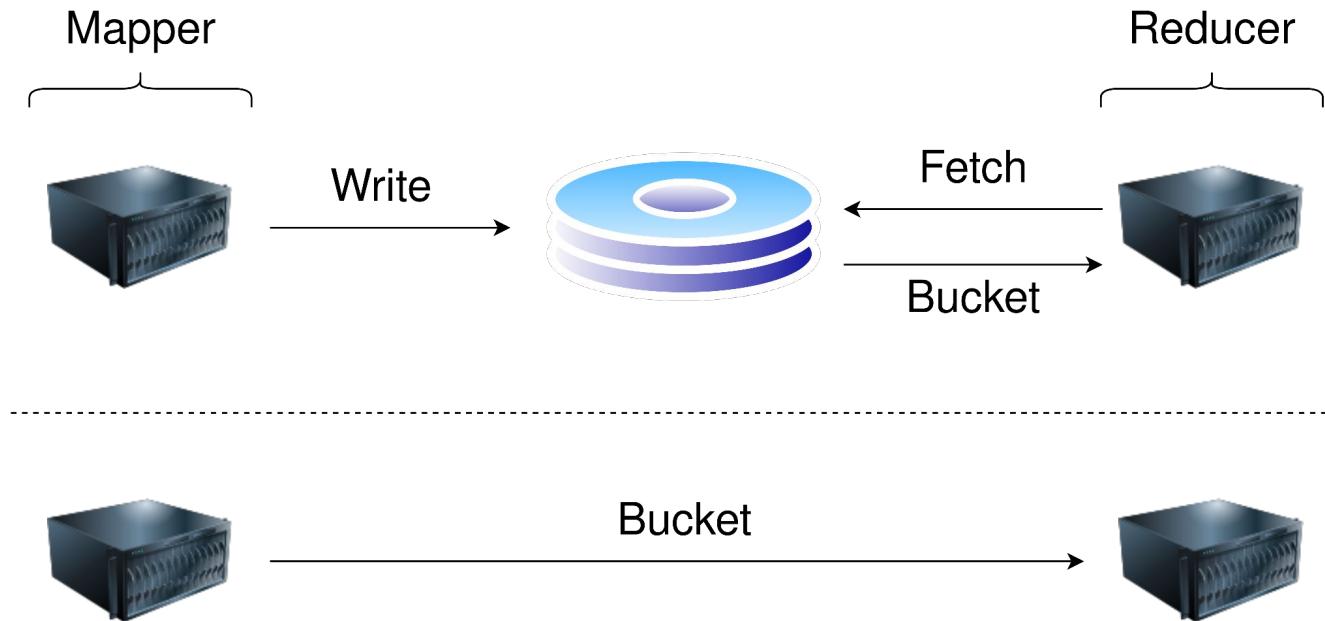
MapReduce: реализация мапперов

- Собираем пары (`key, value`) в R корзин
- i-ая корзина содержит пары, предназначенные i-ому редьюсеру
- Время от времени сортируем корзины и дампим на диск



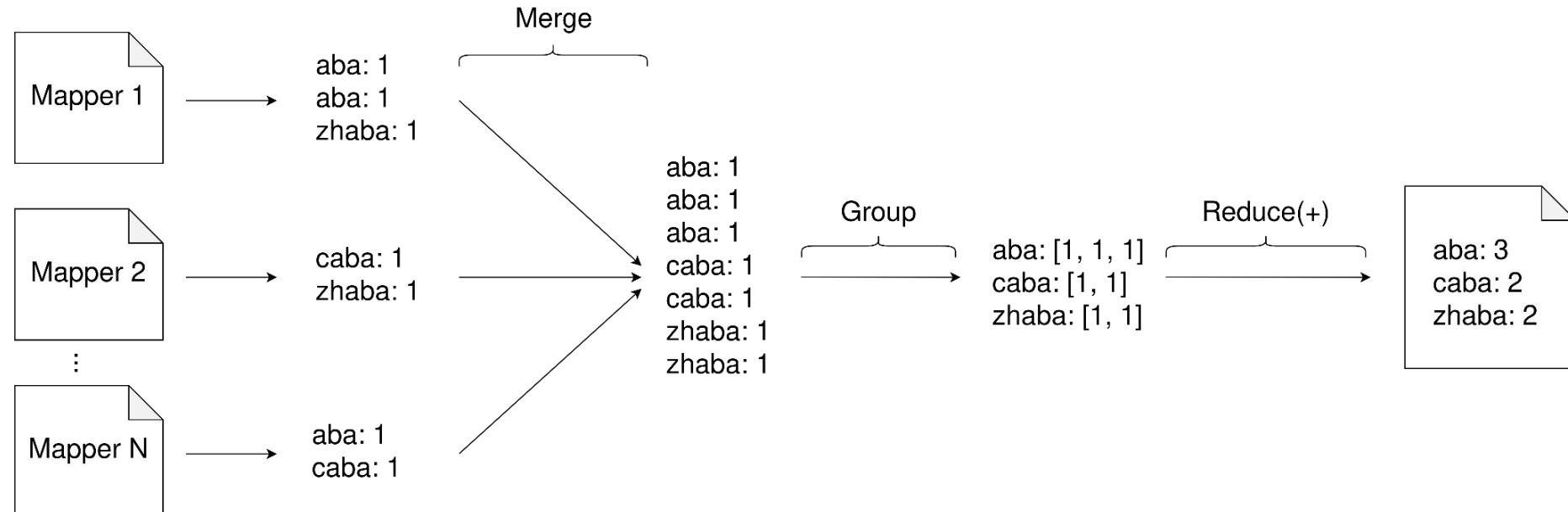
MapReduce: передача ключей

- Маппер может записать корзину на диск, а редьюсер придет за ними
- Редьюсер может забрать данные напрямую с маппера



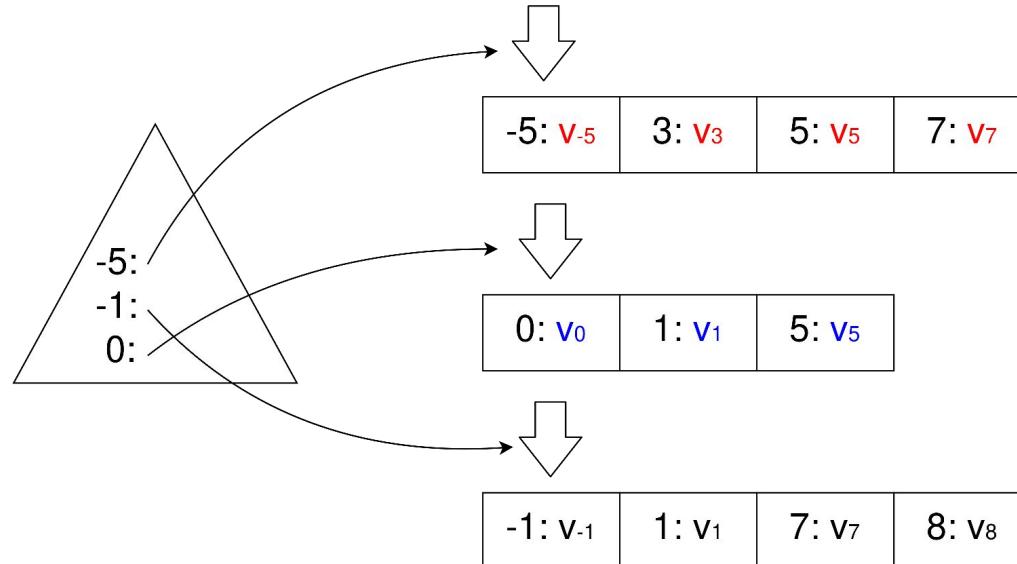
MapReduce: реализация редьюсеров

- Получаем і-ую корзину с **каждого** маппера
- Объединяем, группируем, сворачиваем
- Каждая корзина отсортирована



Объединение корзин: Heap merge

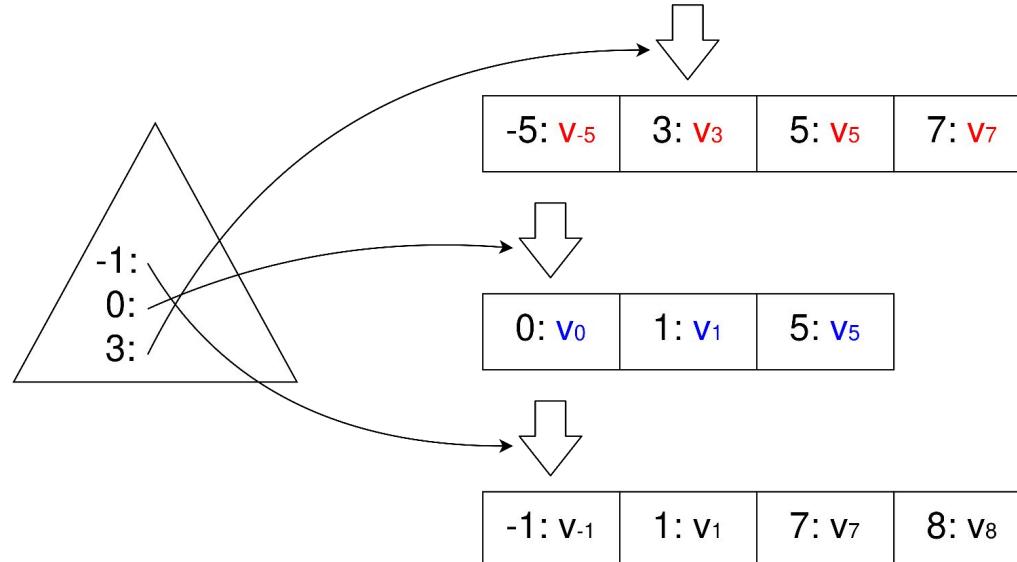
- Первый элемент каждой непустой корзины добавим в кучу
- Куча хранит текущий элемент вместе с указателем на его место в его корзине
- Умеет доставать минимальный элемент



result:

Объединение корзин: Heap merge

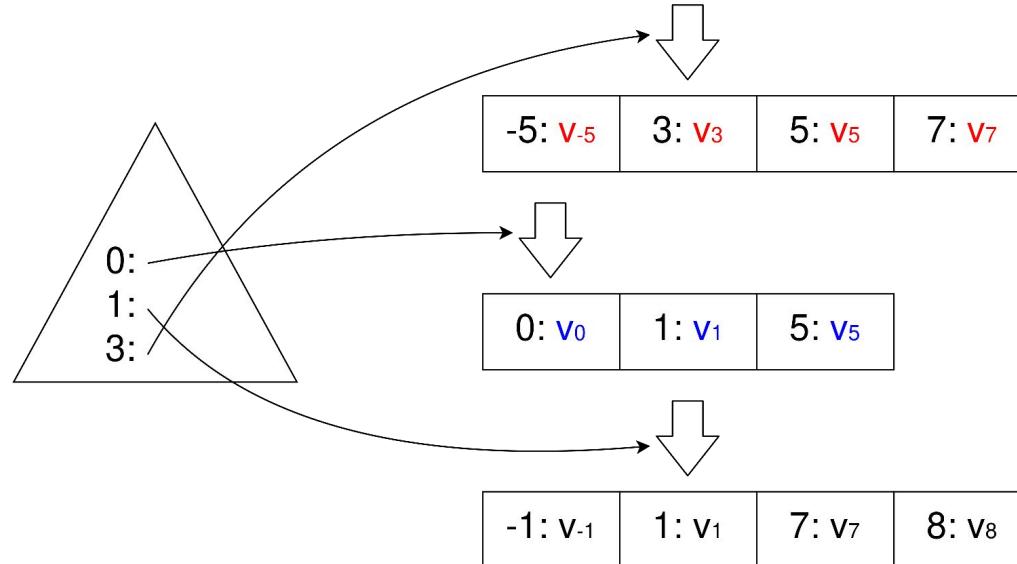
- На каждом шаге выбираем в куче минимальный элемент
- Это минимальный возможный элемент
- Добавляем его в ответ
- Передвигаем указатель вперёд



result: -5: v_{-5} []

Объединение корзин: Heap merge

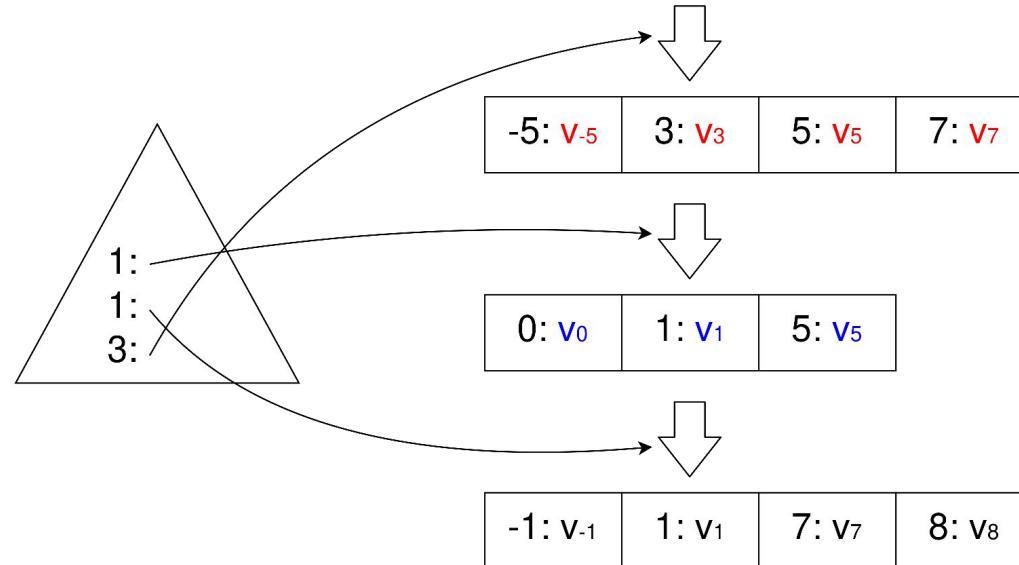
- На каждом шаге выбираем в куче минимальный элемент
- Это минимальный возможный элемент
- Добавляем его в ответ
- Передвигаем указатель вперёд



result: -5: v₋₅ -1: v₋₁

Объединение корзин: Heap merge

- Одному ключу может соответствовать несколько значений
- Это ожидаемое поведение
- В ответ берём все по очереди
- Порядок не важен

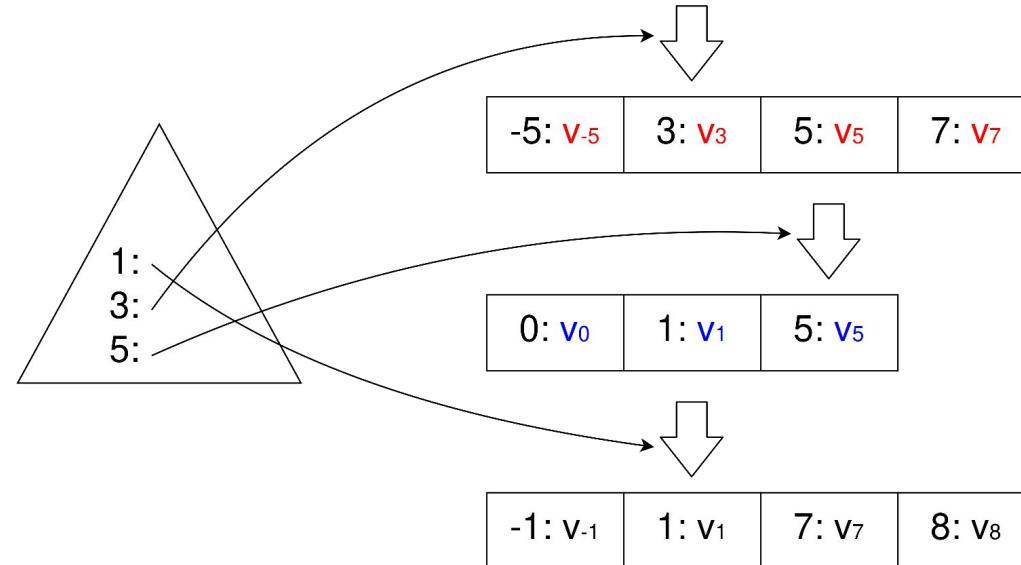


result:

-5: v_{-5}	-1: v_{-1}	0: v_0	
--------------	--------------	----------	--

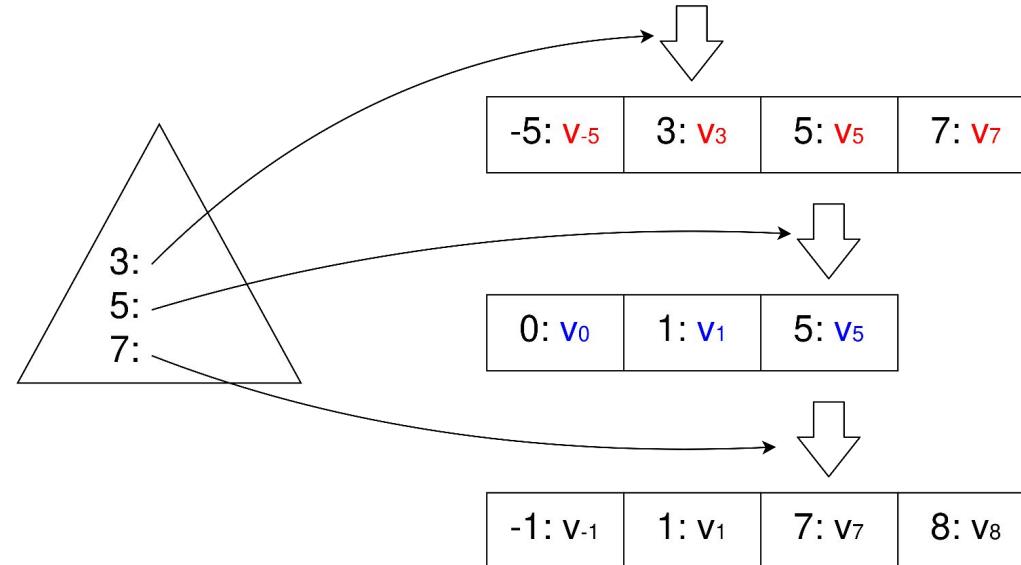
Объединение корзин: Heap merge

- Одному ключу может соответствовать несколько значений
- Это ожидаемое поведение
- В ответ берём все по очереди
- Порядок не важен



Объединение корзин: Heap merge

- Одному ключу может соответствовать несколько значений
- Это ожидаемое поведение
- В ответ берём все по очереди
- Порядок не важен

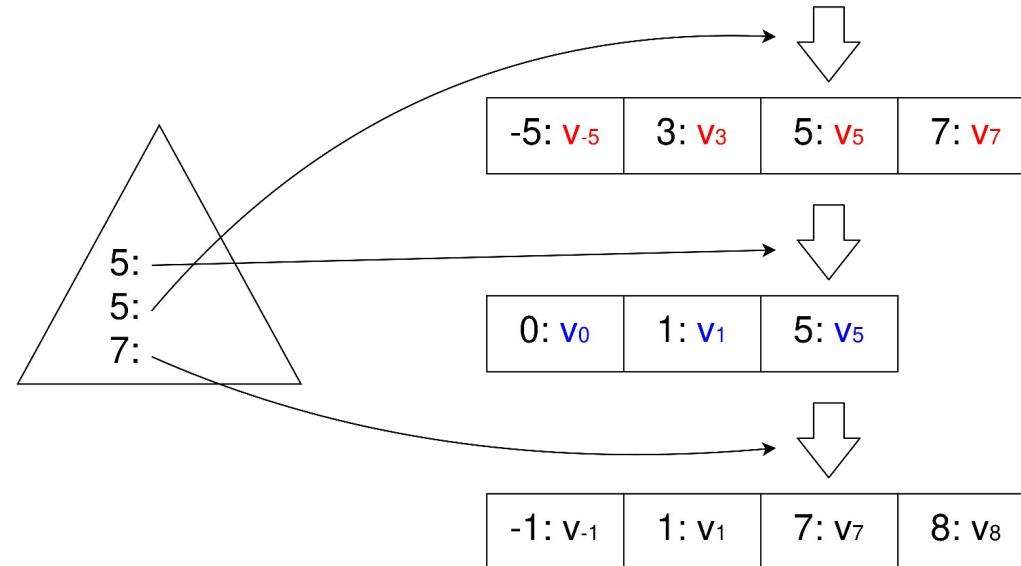


result:

-5: v_{-5}	-1: v_{-1}	0: v_0	1: v_1	1: v_1	
--------------	--------------	----------	----------	----------	--

Объединение корзин: Heap merge

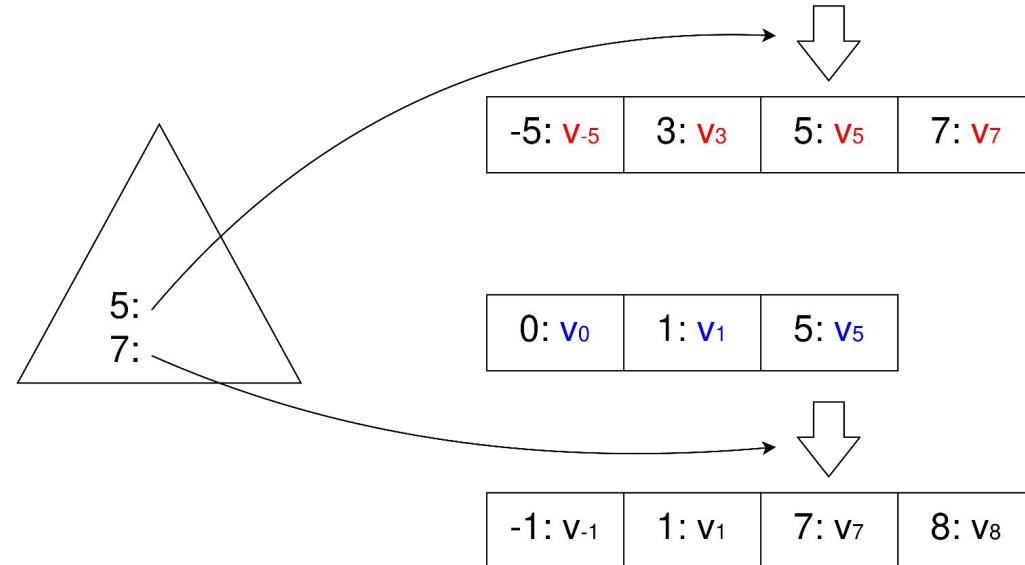
- Двигаем указатели, пока в корзинах есть элементы



result: $-5: v_{-5}$ $-1: v_{-1}$ $0: v_0$ $1: v_1$ $1: v_1$ $3: v_3$ \square

Объединение корзин: Heap merge

- Если указатель указывает на последний элемент в корзине, добавляем его в ответ
- Удаляем элемент из кучи
- Повторяем шаги, пока в куче есть элементы



result:

-5: v_{-5}	-1: v_{-1}	0: v_0	1: v_1	1: v_1	3: v_3	5: v_5	
--------------	--------------	----------	----------	----------	----------	----------	--

Распределённые вычисления: отказы

- В случае сбоя просто повторим задачу на другом сервере
- **Не всё вычисление, а одну**
Мар/Reduce стадию

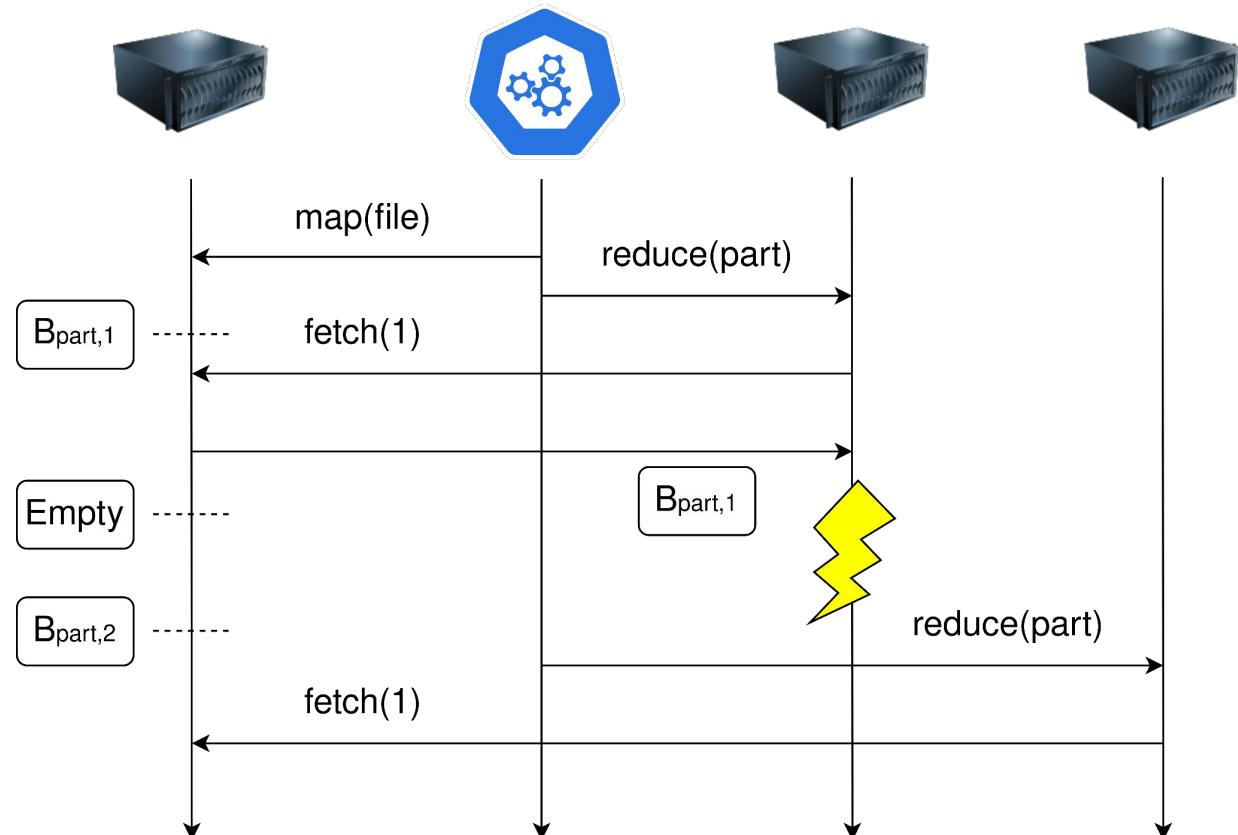


*Reassign
task*



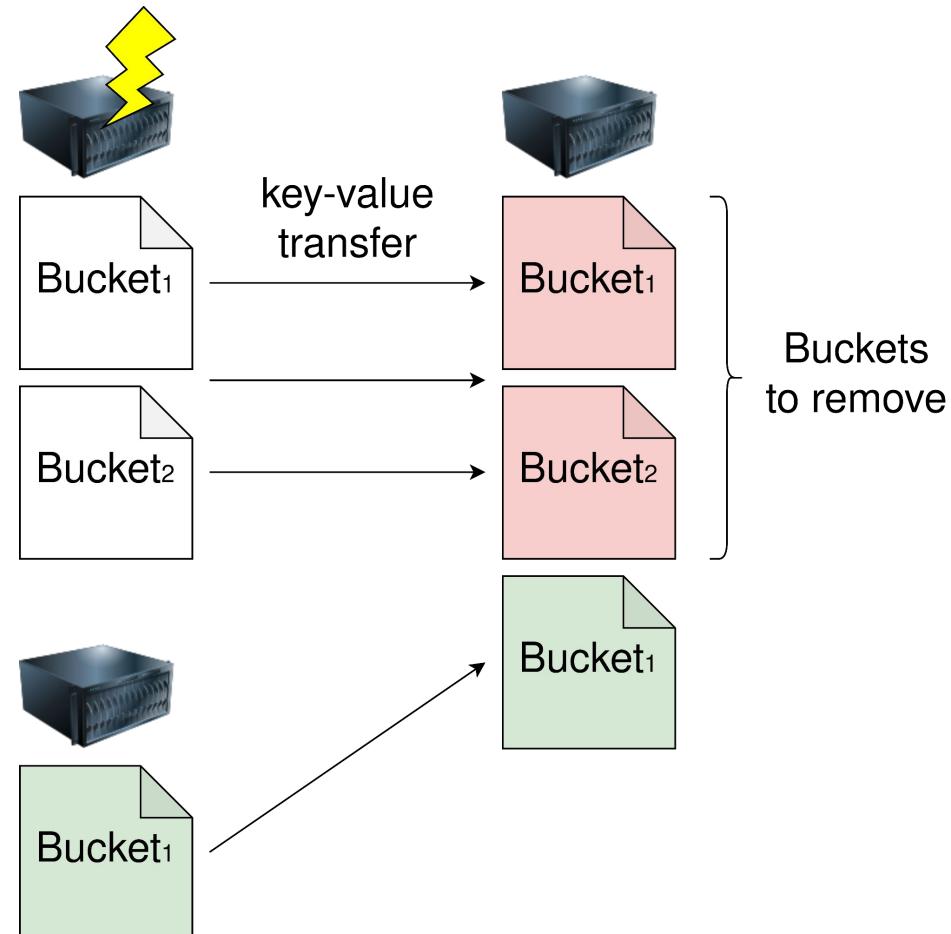
MapReduce: отказы редьюсеров

- Пусть мапперы отдают редьюсерам корзины, не сохраняя их на диске
- Сбой **одного** редьюсера может привести к перезапуску **всех** мапперов
- Кешируйте промежуточные результаты!



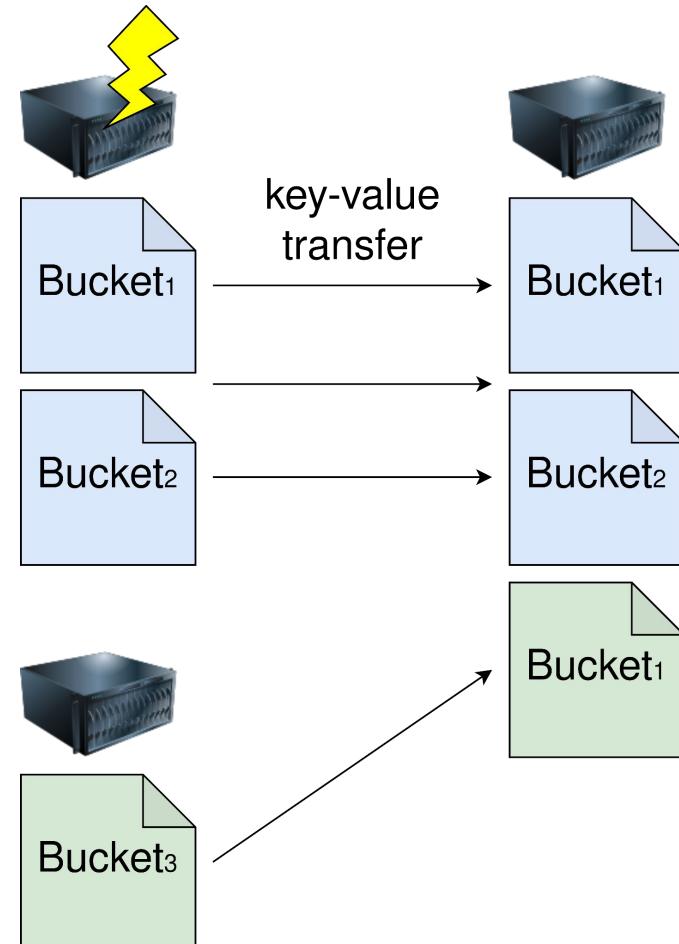
MapReduce: отказы мапперов

- Корзины, полученные от упавшего маппера, удаляются всеми редьюсерами
- Мар-задача перезапускается на другом сервере
- Редьюсеры начинают получать корзины с нового сервера



MapReduce: отказы мапперов

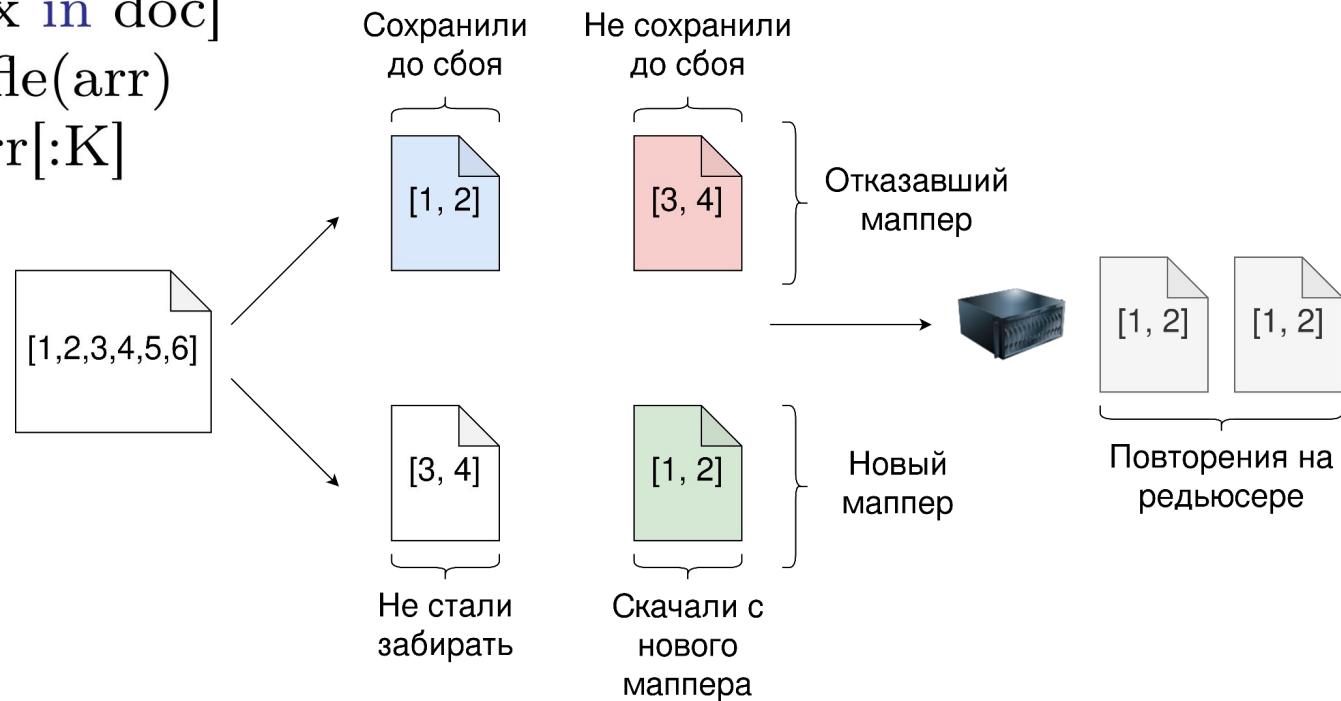
- Почему нельзя продолжить с того места, на котором закончили?



MapReduce: отказы мапперов

- Можем получить некорректный результат на редьюсере

```
1 fun mapper(doc):  
2     arr = [x for x in doc]  
3     random.shuffle(arr)  
4     yield from arr[:K]
```



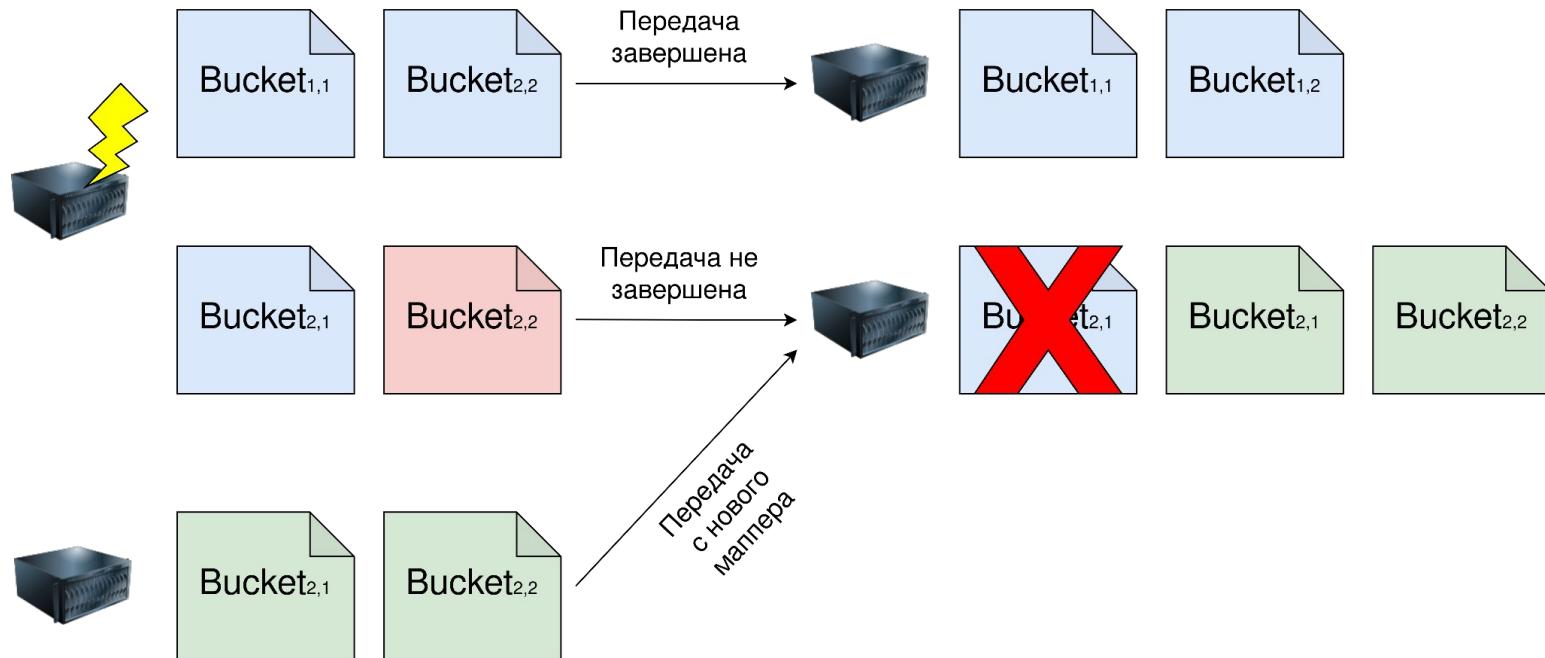
MapReduce: отказы мапперов

- Можем продолжить с того же места, на котором остановились, если маппер детерминирован



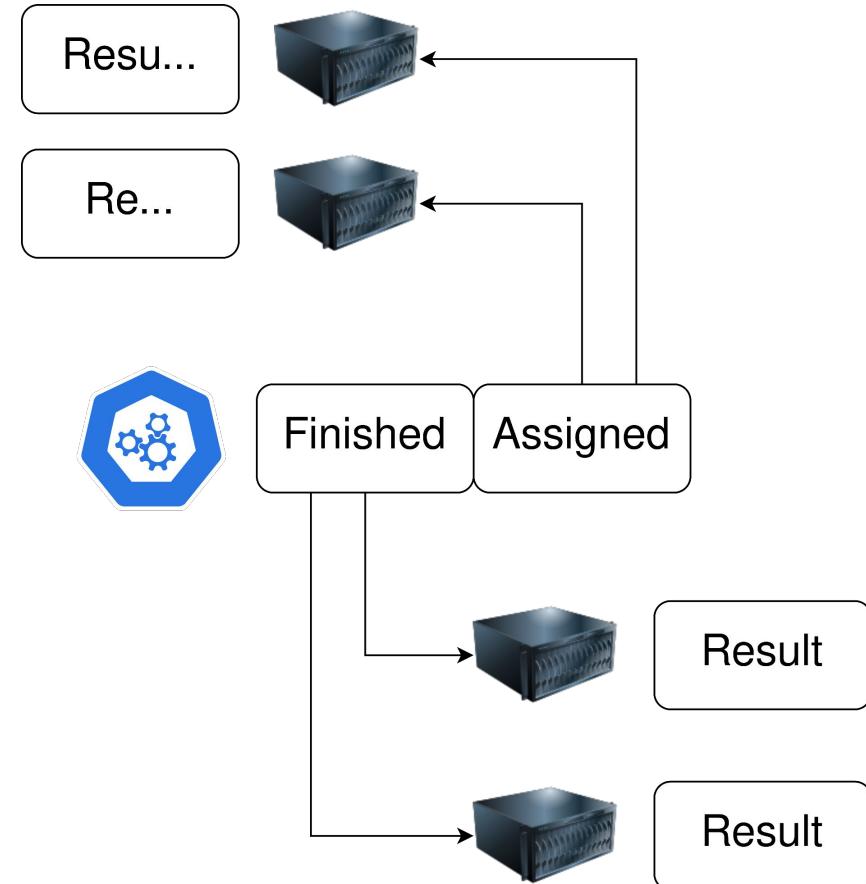
MapReduce: отказы мапперов

- Если передача с упавшего маппера на одного из редьюсеров завершена, этот редьюсер может не перекачивать корзины



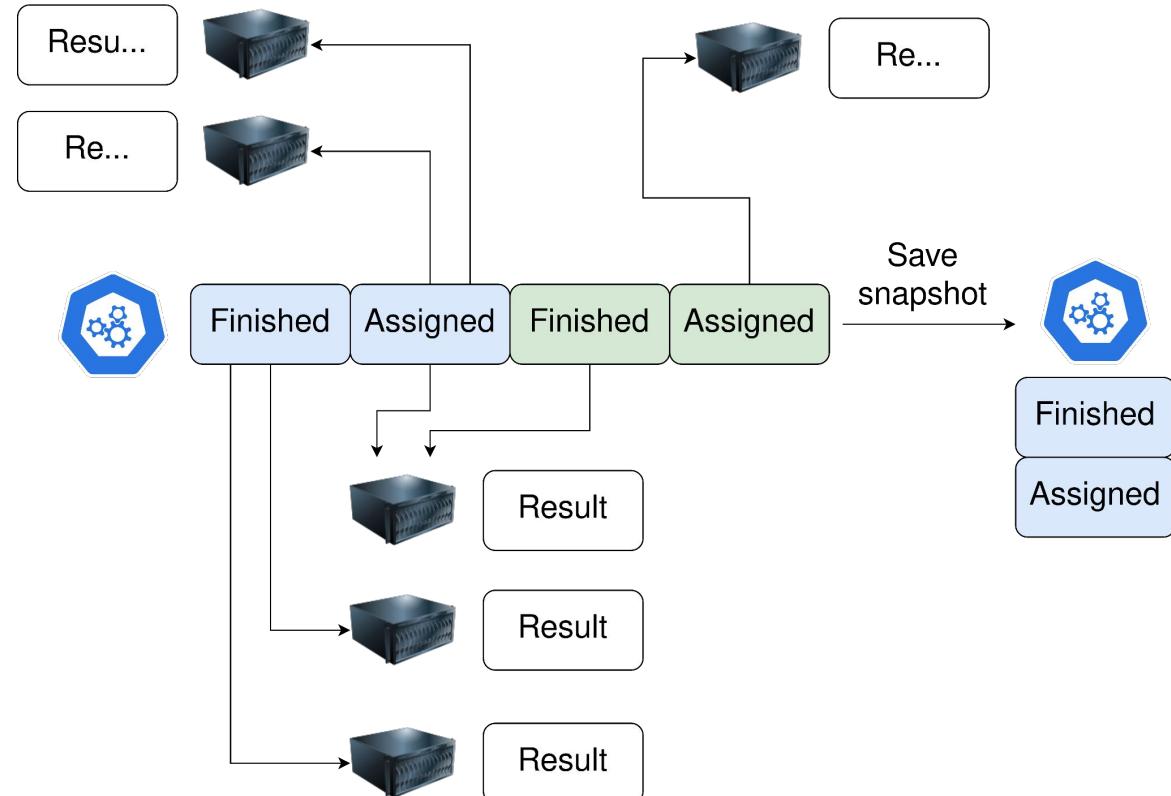
MapReduce: отказы мастера

- Хранит информацию обо всех задачах
- Исполненных и начатых
- Где лежат результаты
- Кто исполняет
- При сбое вынуждены начать вычисление с начала



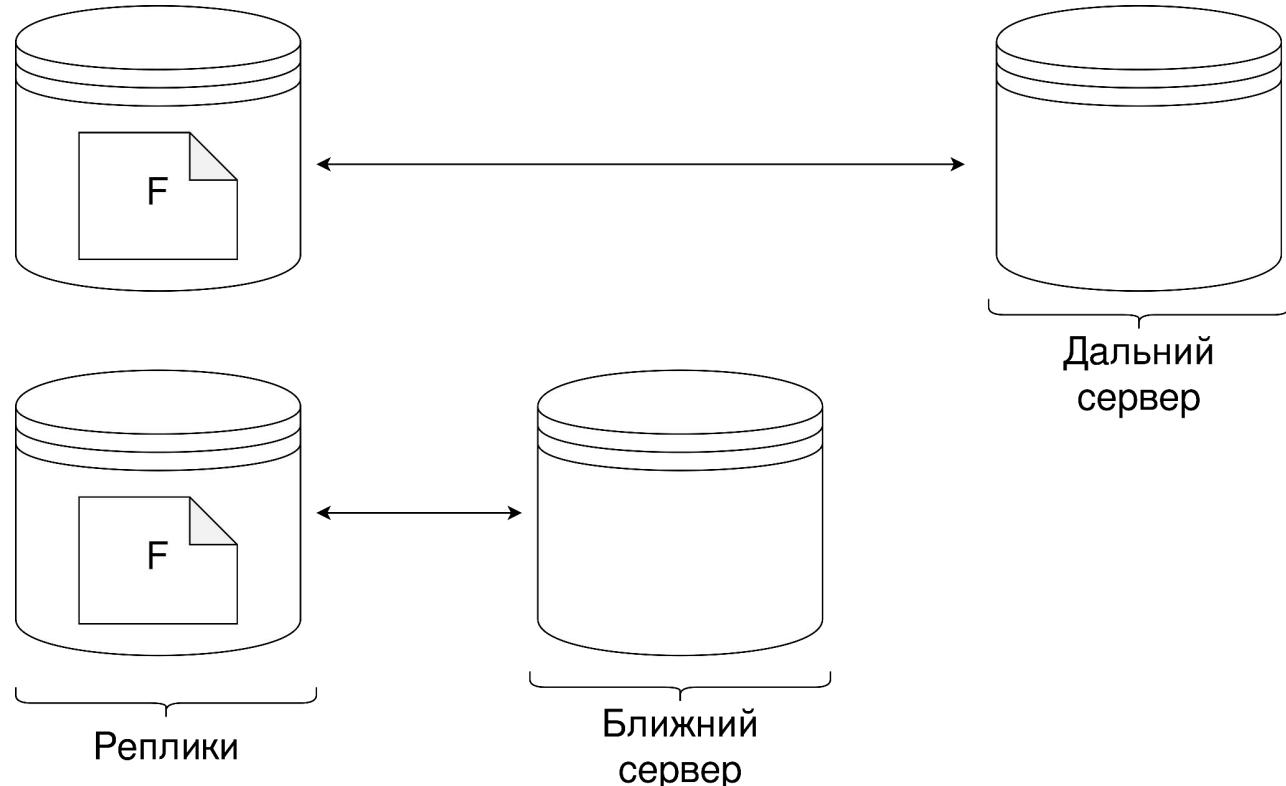
MapReduce: отказы мастера

- Периодически делаем снимки состояния мастера
- Реплицируем на standby-мастера
- В случае сбоя мастера вынуждены будете перезапустить часть задач
- И проверить, выполнение



MapReduce: локальность запуска

- Запустим Map(File) на одном из серверов, хранящих этот файл
- Или на одном из ближайших серверов
- Чтобы уменьшить затраты на передачу данных



MapReduce: избыточность

- Каждую задачу можно запускать в нескольких копиях
- Брать быстрейший результат
- Чтобы
тормозящие
сервера не
затормозили
всё вычисление
- Слишком
расточительно



Resu...₁



Result₂



Re...₁



Res...₂



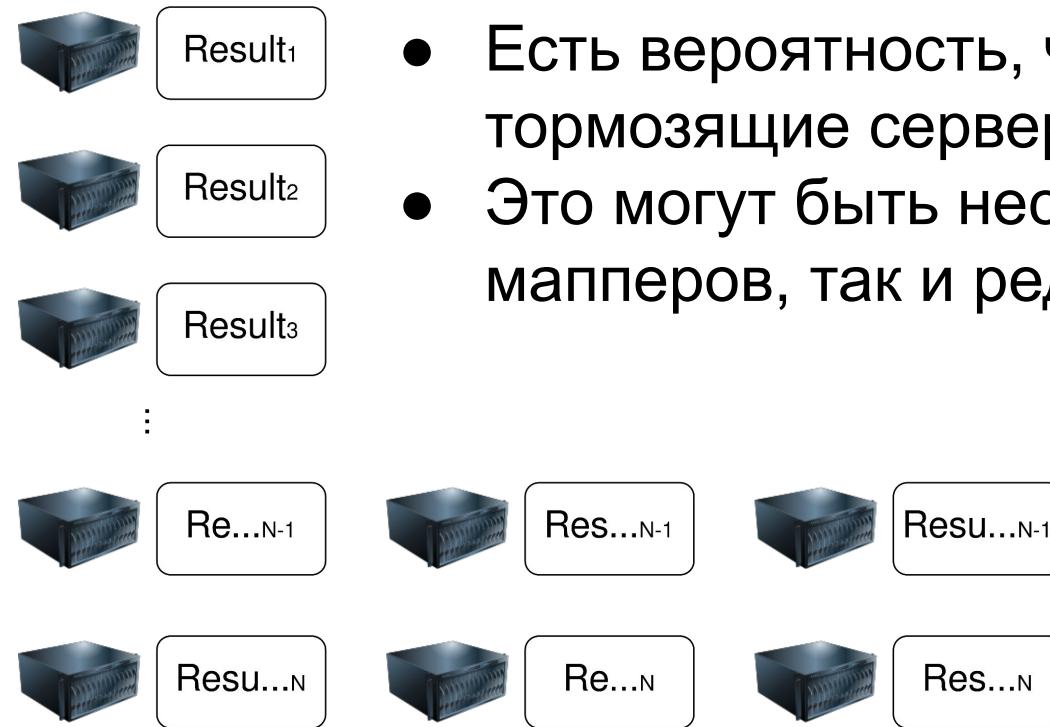
Result₁



Resu...₂

MapReduce: избыточность

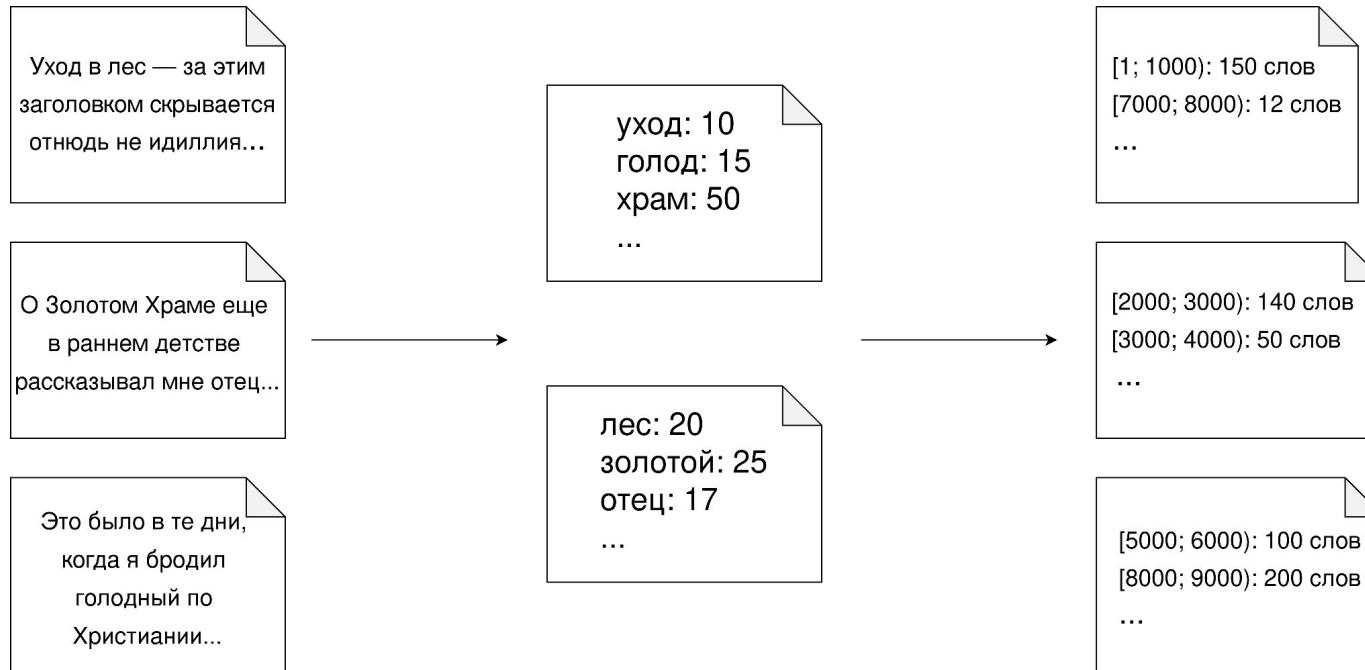
- Запустим несколько последних тормозящих задач в нескольких копиях



- Есть вероятность, что они просто попали на тормозящие сервера
- Это могут быть несколько последних как мапперов, так и редьюсеров

MapReduce: каскады

- Иногда задачу нельзя решить за один MapReduce
- Для каждого диапазона узнать, сколько слов имеет встречаемость в этом диапазоне



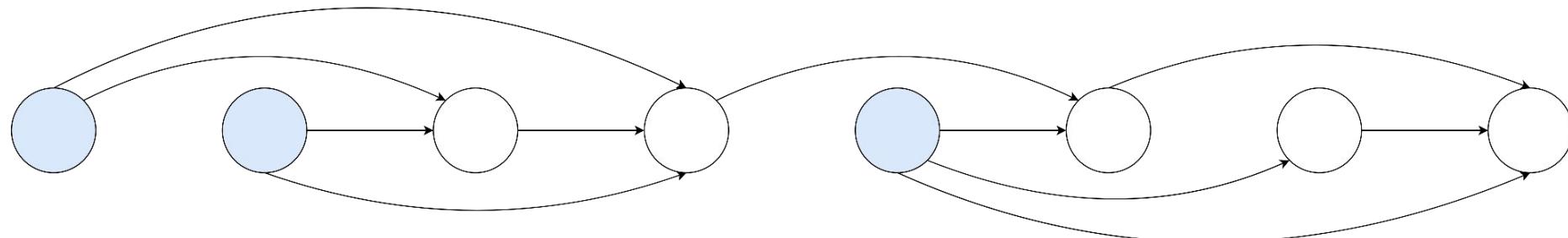
MapReduce: каскады задач

- Запускаем новое MapReduce вычисление на результате первого вычисления
- Почему этого достаточно?
- Почему ни одно слово не будет учтено дважды?

```
1 fun mapper(doc):  
2     for word, count in doc:  
3         yield ⌊  $\frac{count}{1000}$  ⌋, 1  
4  
5 fun reducer(range, ones):  
6     yield [range * 1000; (range + 1) * 1000), sum(ones)
```

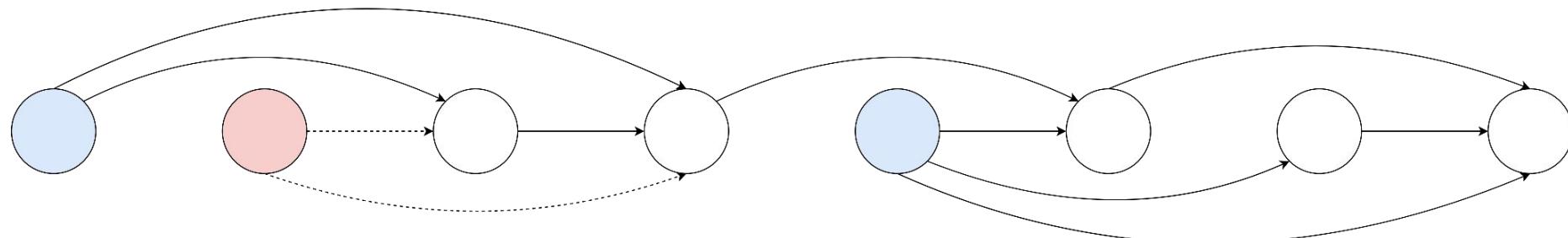
MapReduce: каскады задач

- В общем случае вычисление представляет из себя ациклический граф
- Топологически сортируем его
- Начинаем выполнение с истоков
 - В ациклическом графе всегда есть исток



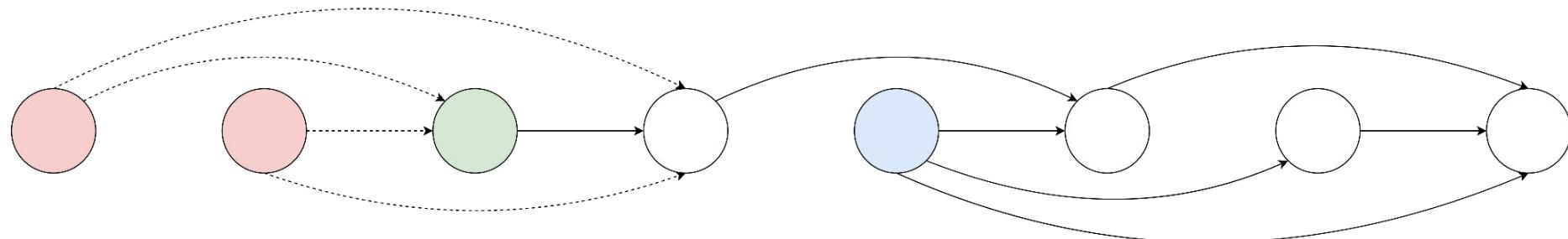
MapReduce: каскады задач

- После выполнения задачи удаляем соответствующее ей ребро из графа
- Помечаем исходящие из её рёбра как выполненные зависимости



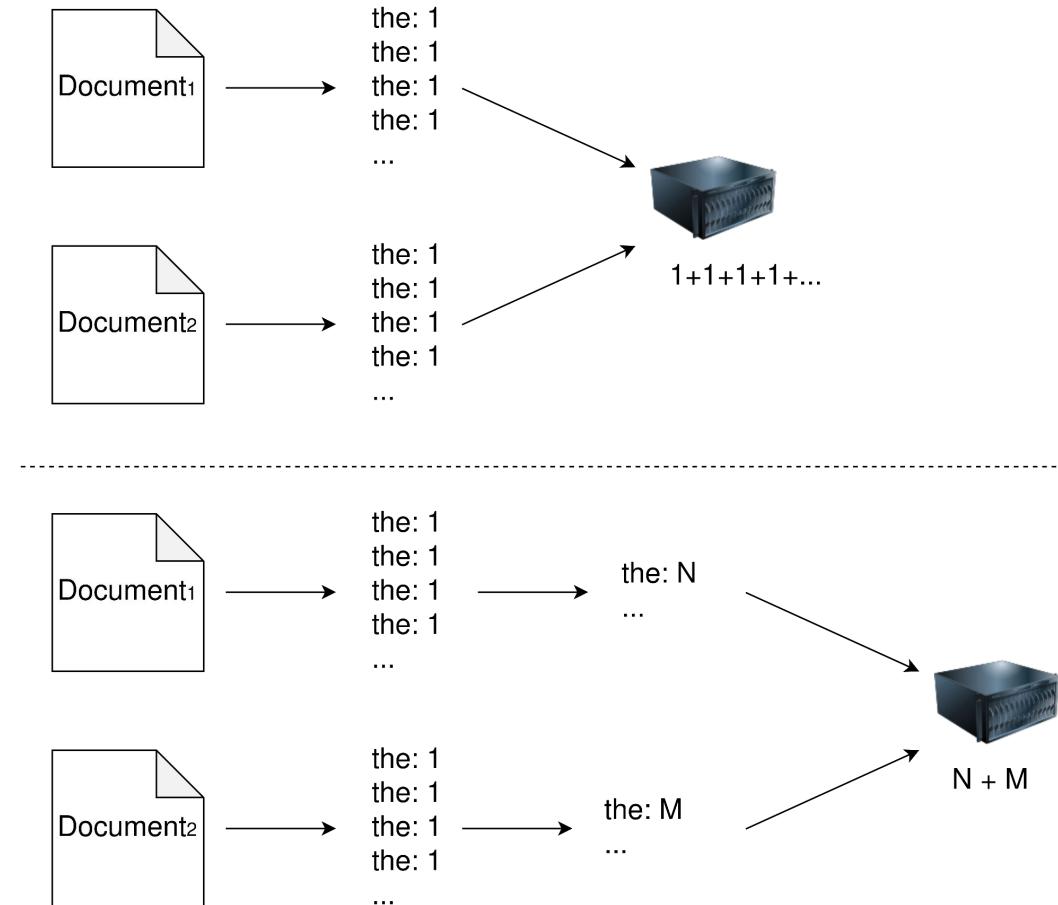
MapReduce: каскады задач

- После завершения задачи T для каждого исходящего из неё ребра $T \rightarrow V$ проверяем, можно ли выполнить задачу V
- Не осталось ли у неё невыполненных зависимостей



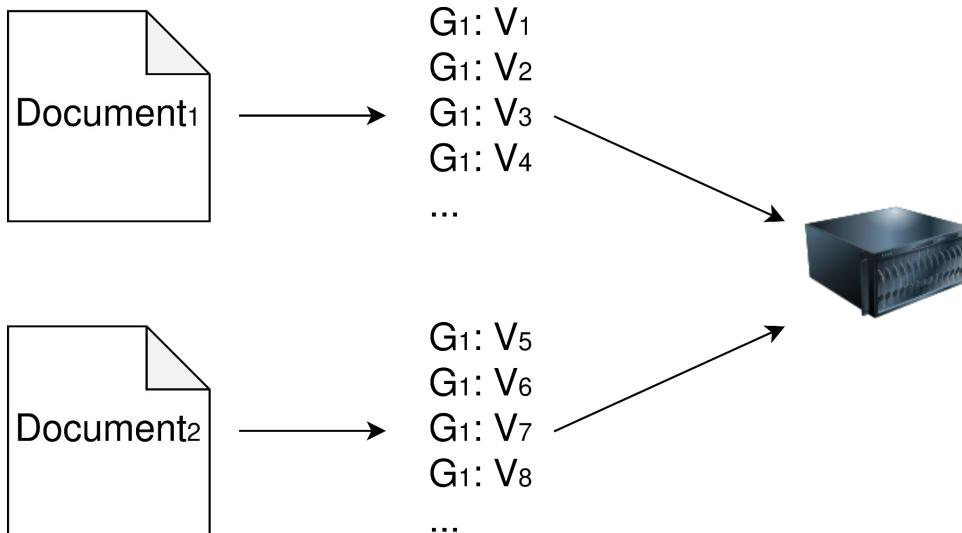
MapReduce: Combiner-оптимизация

- В задаче Word Count для часто используемых слов передаём слишком много пар (слово, 1) с каждого маппера
- Сделаем локальную свёртку перед передачей на редьюсер



Combiner ≠ Reducer: пример

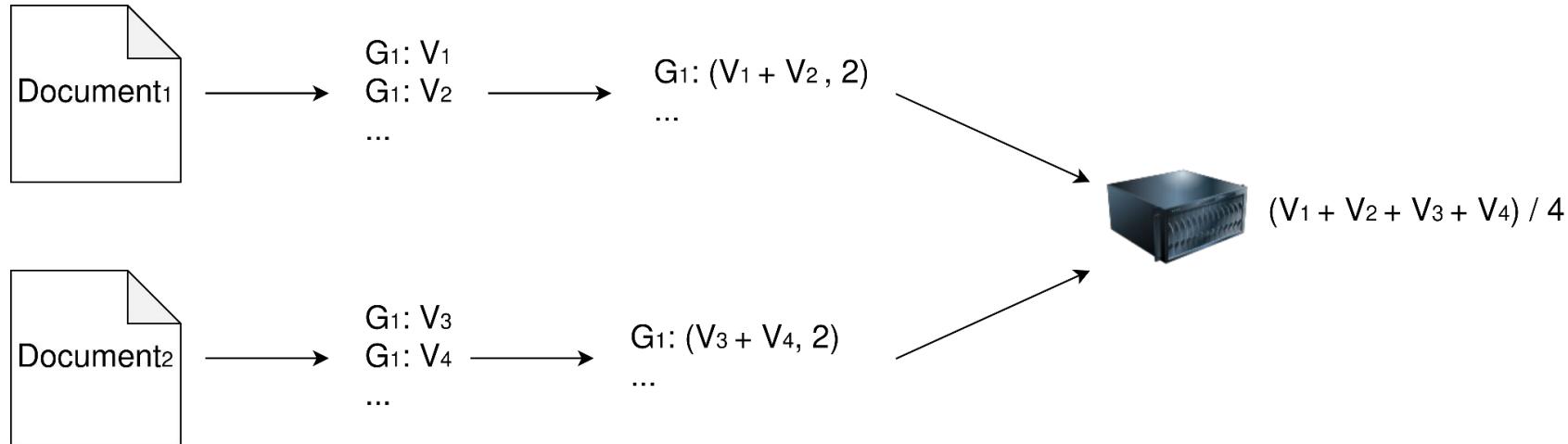
- Даны таблица пар (group, value)
- Найти среднее значение в каждой группе



```
1 fun mapper(doc):  
2     for group, value in doc:  
3         yield group, value  
4  
5 fun reducer(group, values):  
6     yield group,  $\frac{\text{sum(values)}}{\text{len(values)}}$ 
```

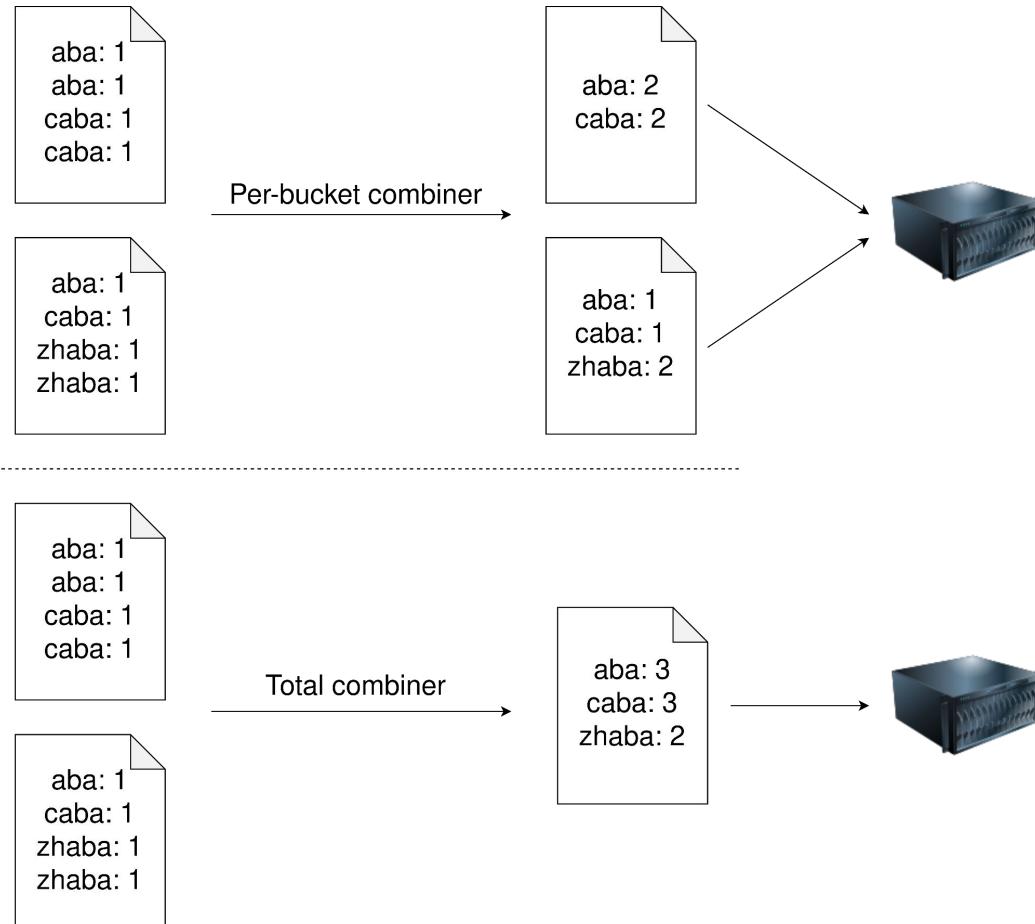
Combiner ≠ Reducer: пример

- Среднее средних это не среднее!
 - $\text{avg}(100, 100, 100, 1) = 75.25$
 - $\text{avg}(\text{avg}(100, 100, 100), \text{avg}(1)) = 50.5$
- Combiner будет считать не среднее, а локальную сумму и количество



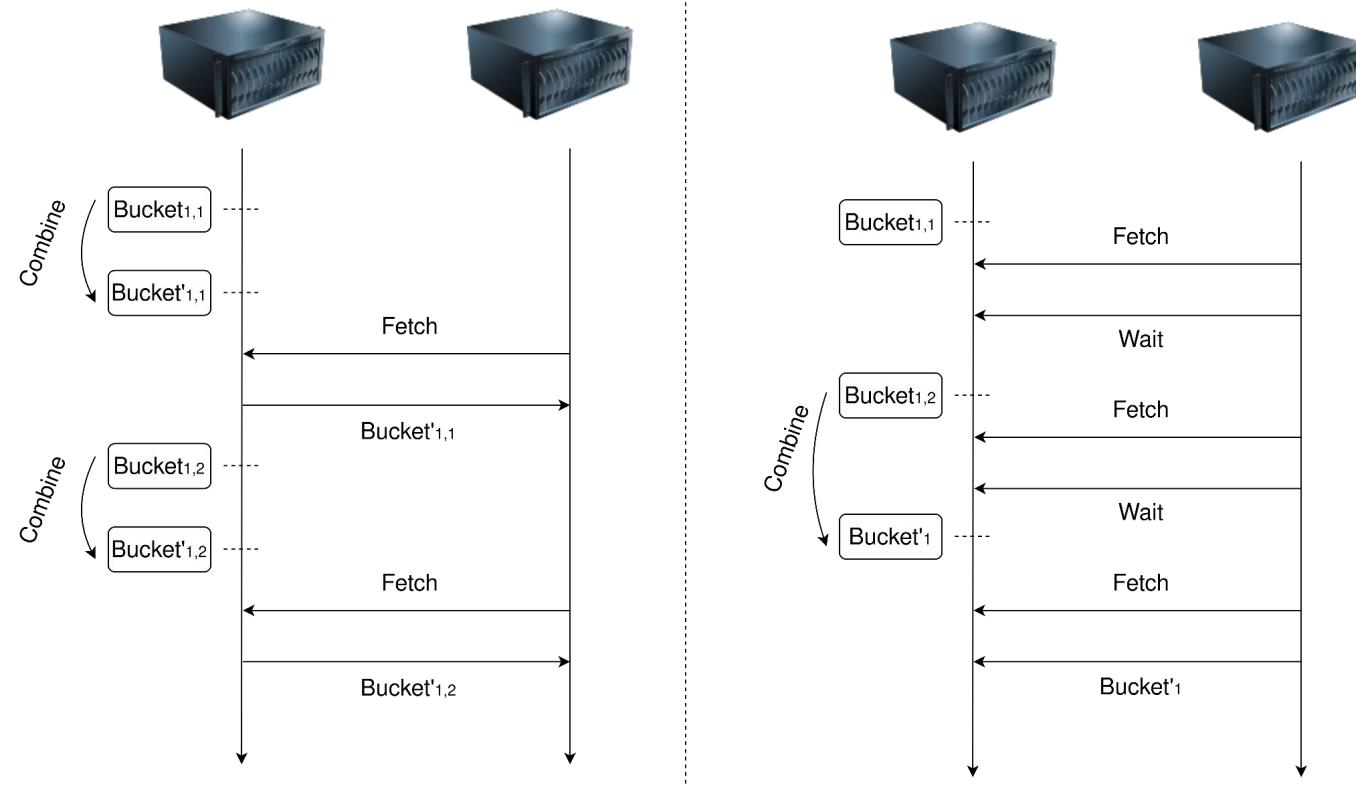
Combiner: когда применять?

- Можно перед сохранением корзины на диск
- А можно после сохранения всех локальных корзин сразу для всех данных
- Во втором случае перешлём меньше данных на редьюсеры



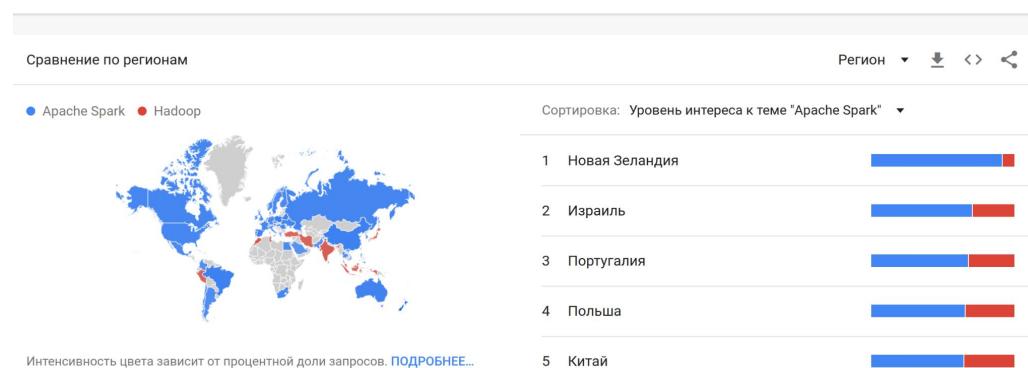
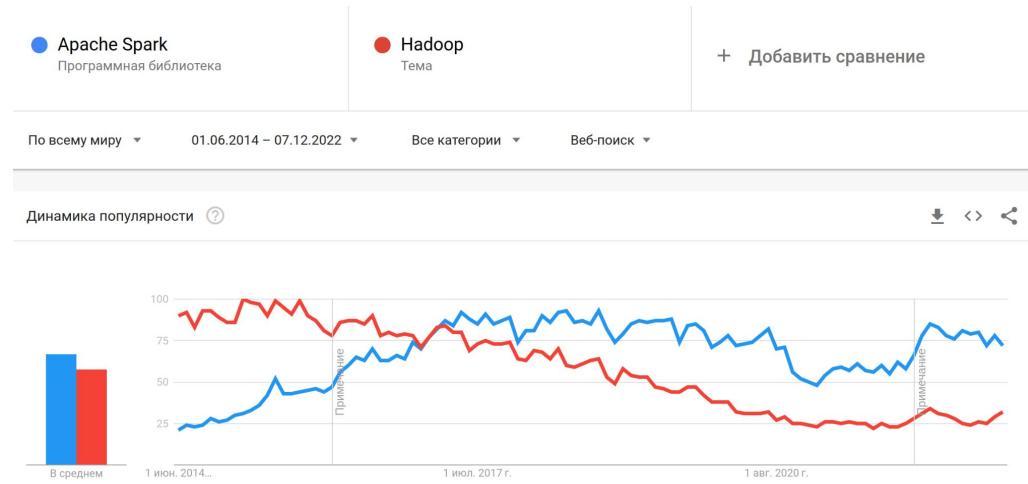
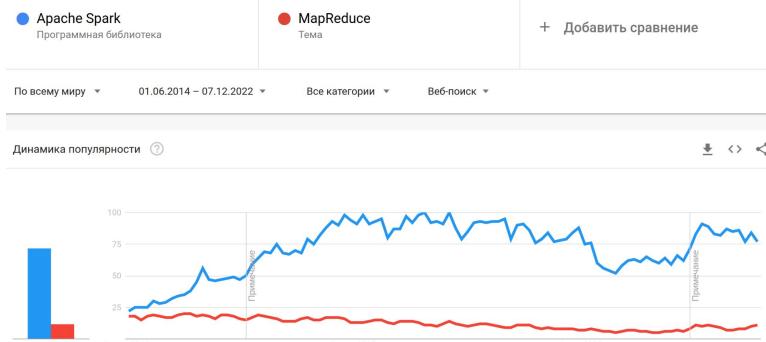
Combiner: когда применять?

- Во втором случае редьюсер может начать скачивать ключи только после завершения работы маппера



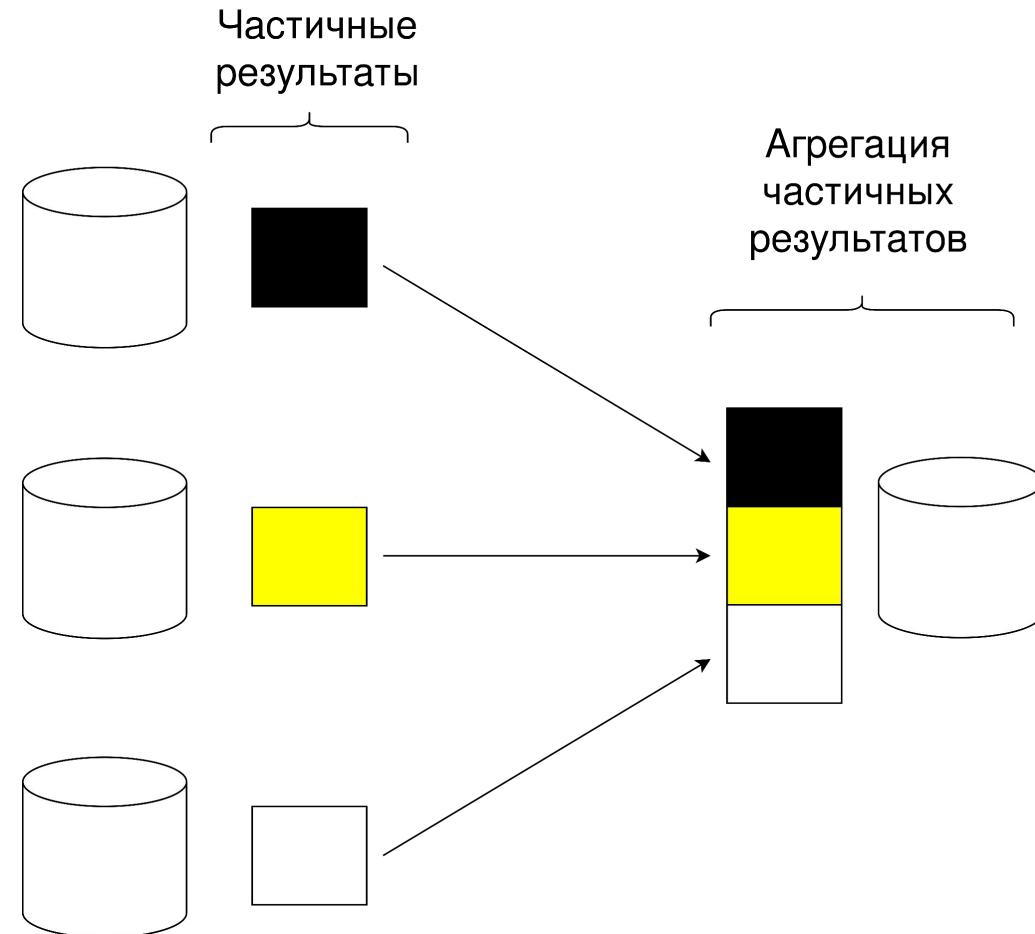
MapReduce не нужен?

- По данным Google Trends
- Аналогичные результаты по другим агрегаторам



MapReduce не нужен?

- MapReduce это естественный способ реализации распределённых алгоритмов
- Даже если технология называется по-другому



MapReduce не нужен?

- Аналогично, fork-join это естественный способ реализации параллельных алгоритмов в системах с общей памятью
- В распределённой системе невозможен
- Fork-подзадача должна запуститься 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 = ⌊  $\frac{left+right}{2}$  ⌋  
5     var a  
6     fork { a = calc_sum(arr, left, mid) }  
7     b = calc_sum(arr, mid, right)  
8     join  
9     return a + b
```

MapReduce не нужен?

- “*Gamma implements scalar aggregates by having each processor compute its piece of the result in parallel. The partial results are then sent to a single process which combines these partial results into the final answer. Aggregate functions are computed in two steps. First, each processor computes a piece of the result by calculating a value for each of the partitions. Next, the processors redistribute the partial results by hashing on the "group by" attribute. The result of this step is to collect the partial results for each partition at a single site so that the final result for each partition can be computed.*”
 - DeWitt et al. The Gamma database machine project, 1990

Мар-only задачи

- Отлично, если Reduce-фаза не нужна
- Обычно, в фильтрации данных и поиске

```
<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
```

```
other.ru
```

Map

Result

Полнотекстовый поиск: обратный индекс

- Входные данные:
корпус текстов

Full-text Search 101: The inverted index

User queries for “keeper”

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

- Итог: Map<Word, List<Docs>>
 - В которых это слово встречается

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 mapper(doc):  
2     for word in words(doc)  
3         yield word, doc.id  
4  
5 fun reducer(word, docs):  
6     yield word, unique(docs)
```

Обратный индекс: алгоритм

- Применим Combiner, совпадающий с редьюсером

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: операция

- Таблица X состоит из [групп] столбцов: (A, B)
- Таблица Y состоит из [групп] столбцов: (B, C)
- Тогда $X \bowtie Y$ состоит из [групп] столбцов (A, B, C):

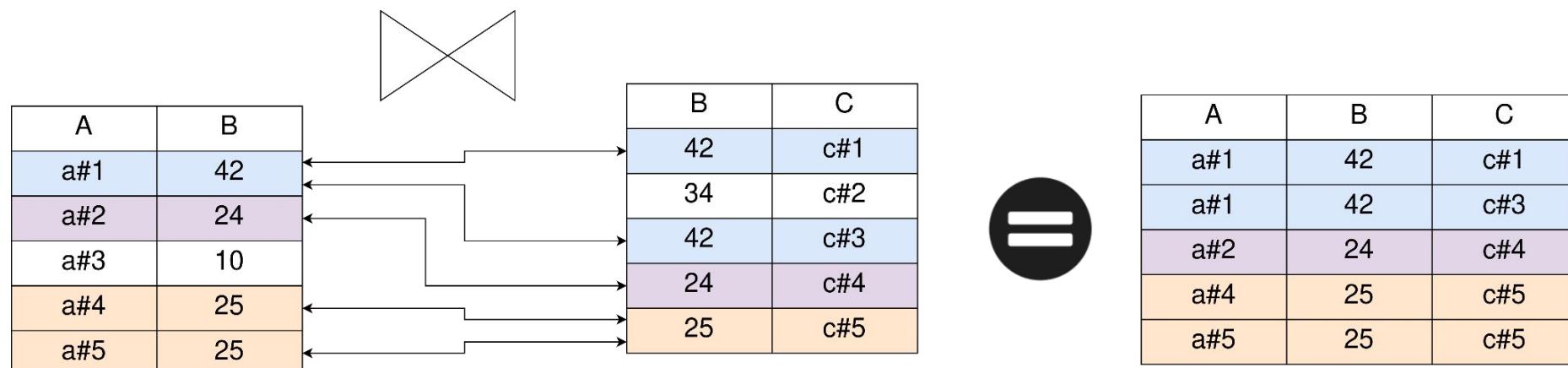
$$X \bowtie Y = \{a, b, c \mid (a, b) \in X; (b, c) \in Y\}$$

- Можно
посчитать
вложенными
циклами

```
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

- Для каждой строки из левой таблицы ищем множество соответствующих строк из правой таблицы
- Соответствие определяется по равенству столбца В
- Симметричная операция: $X \bowtie Y = Y \bowtie X$



Join: алгоритм для MapReduce

- Для каждой строки левой таблицы маппер выдаст пару
`<key=Row.B, value=(Left, Row.A)>`
- `<key=Row.B, value=(Right, Row.C)>` для правой
- За кадром произойдёт группировка по ключу
- После группировки ключу соответствует множество строк
 - “Левые” и “правые” строки 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

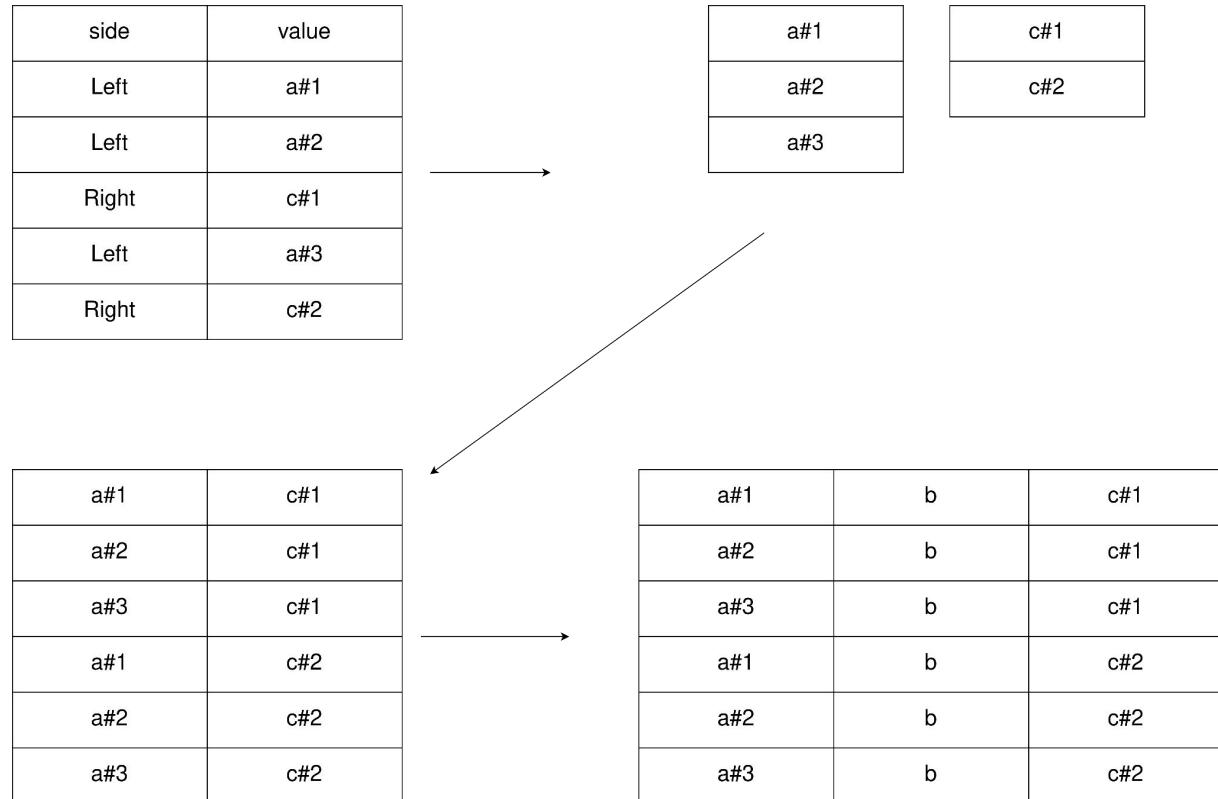
- Для каждой строки левой таблицы маппер выдаст пару
`<key=Row.B, value=(Left, Row.A)>`
- `<key=Row.B, value=(Right, Row.C)>` для правой
- За кадром произойдёт группировка по ключу
- После группировки ключу соответствует множество строк
 - “Левые” и “правые” строки перемешку

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 правая
входит в пару с
каждой левой
- Так как значение
столбца В
совпадает
- Строим
декартово
произведение



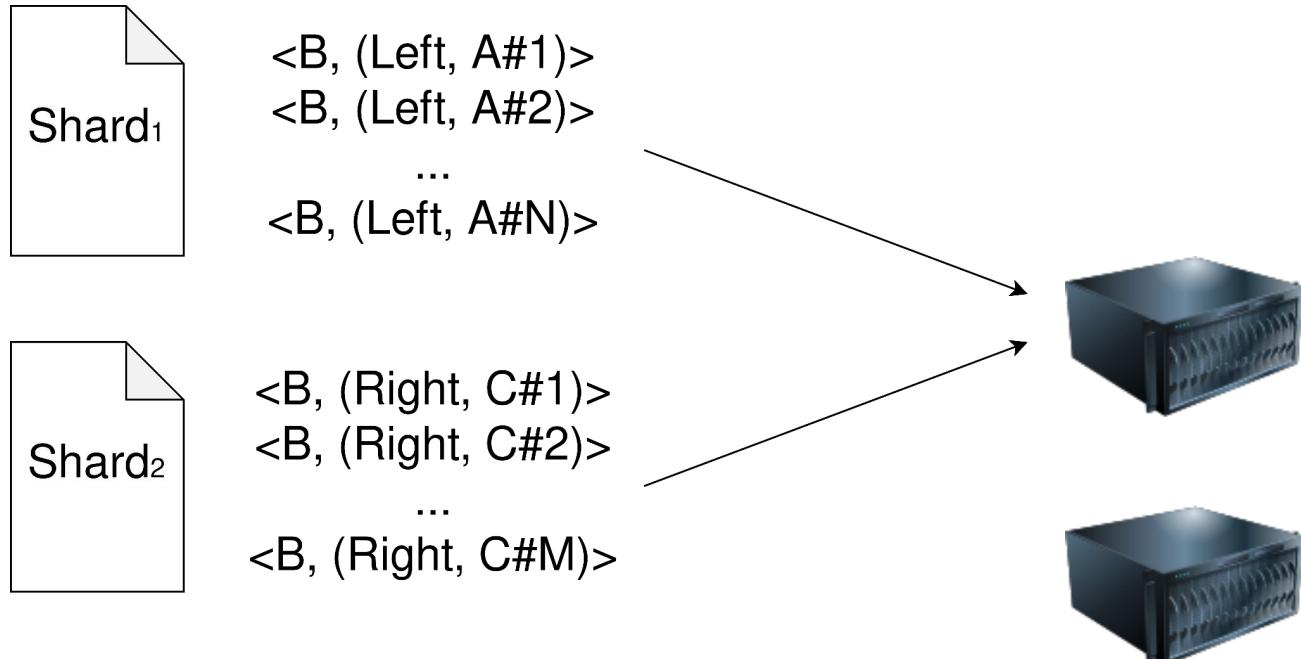
Join: алгоритм для MapReduce

- Разделим строки на правые и левые
- Каждая правая входит в пару с каждой левой
- Так как значение столбца В совпадает
- Строим декартово произведение

```
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: неравномерность нагрузки

- Combiner не поможет
- Reduce-фаза **увеличивает** размер данных
- Нужно научиться распределять популярные ключи по нескольким редьюсерам



Join: равномерное распределение нагрузки

- Разделим всё пространство пар с одним ключом (B) на k^*m **независимых ячеек** и расположим их в матрицу
- Каждая строка левой таблицы попадёт в каждую ячейку случайной строки

			...		
			...		
$\langle A, B \rangle$	$\langle A, B \rangle$	$\langle A, B \rangle$...	$\langle A, B \rangle$	$\langle A, B \rangle$
			...		
			...		

Join: равномерное распределение нагрузки

- Разделим всё пространство пар с одним ключом (B) на k^*m **независимых ячеек** и расположим их в матрицу
- Каждая строка правой таблицы попадёт в каждую ячейку случайного столбца

	<B, C>		...		
	<B, C>		...		
	<B, C>		...		
	<B, C>		...		
	<B, C>		...		

Join: равномерное распределение нагрузки

- Строка (A, B) из левой таблицы пересечётся со строкой (B, C) из правой таблицы **ровно** в одной ячейке

	$\langle B, C \rangle$...		
	$\langle B, C \rangle$...		
$\langle A, B \rangle$	$\langle A, B \rangle$ $\langle B, C \rangle$	$\langle A, B \rangle$...	$\langle A, B \rangle$	$\langle A, B \rangle$
			...		
	$\langle B, C \rangle$...		
	$\langle B, C \rangle$...		

Join: равномерное распределение нагрузки

- Строка (A_i, B_j) из левой таблицы пересечётся со строкой (B_k, C_l) из правой таблицы **ровно в одной ячейке**

$\langle A_1, B \rangle$ $\langle B, C_1 \rangle$	$\langle A_1, B \rangle$ $\langle B, C_2 \rangle$	$\langle A_1, B \rangle$ $\langle B, C_3 \rangle$...	$\langle A_1, B \rangle$ $\langle B, C_{m-1} \rangle$	$\langle A_1, B \rangle$ $\langle B, C_m \rangle$
$\langle A_2, B \rangle$ $\langle B, C_1 \rangle$	$\langle A_2, B \rangle$ $\langle B, C_2 \rangle$	$\langle A_2, B \rangle$ $\langle B, C_3 \rangle$...	$\langle A_2, B \rangle$ $\langle B, C_{m-1} \rangle$	$\langle A_2, B \rangle$ $\langle B, C_m \rangle$
$\langle A_3, B \rangle$ $\langle B, C_1 \rangle$	$\langle A_3, B \rangle$ $\langle B, C_2 \rangle$	$\langle A_3, B \rangle$ $\langle B, C_3 \rangle$...	$\langle A_3, B \rangle$ $\langle B, C_{m-1} \rangle$	$\langle A_3, B \rangle$ $\langle B, C_m \rangle$
$\langle A_{k-1}, B \rangle$ $\langle B, C_1 \rangle$	$\langle A_{k-1}, B \rangle$ $\langle B, C_2 \rangle$	$\langle A_{k-1}, B \rangle$ $\langle B, C_3 \rangle$...	$\langle A_{k-1}, B \rangle$ $\langle B, C_{m-1} \rangle$	$\langle A_{k-1}, B \rangle$ $\langle B, C_m \rangle$
$\langle A_k, B \rangle$ $\langle B, C_1 \rangle$	$\langle A_k, B \rangle$ $\langle B, C_2 \rangle$	$\langle A_k, B \rangle$ $\langle B, C_3 \rangle$...	$\langle A_k, B \rangle$ $\langle B, C_{m-1} \rangle$	$\langle A_k, B \rangle$ $\langle B, C_m \rangle$

Join: анализ производительности

- Пусть ключу b в левой таблице соответствует A_b строк, а в правой — C_b строк
- Каждая строка из левой таблицы обрабатывается m раз
- Из правой — k раз
- Вместо $A_b + C_b$ строк будет обработано $A_b \cdot m + C_b \cdot k$
- В каждую из независимых и равномерно нагруженных ячеек в среднем попадёт

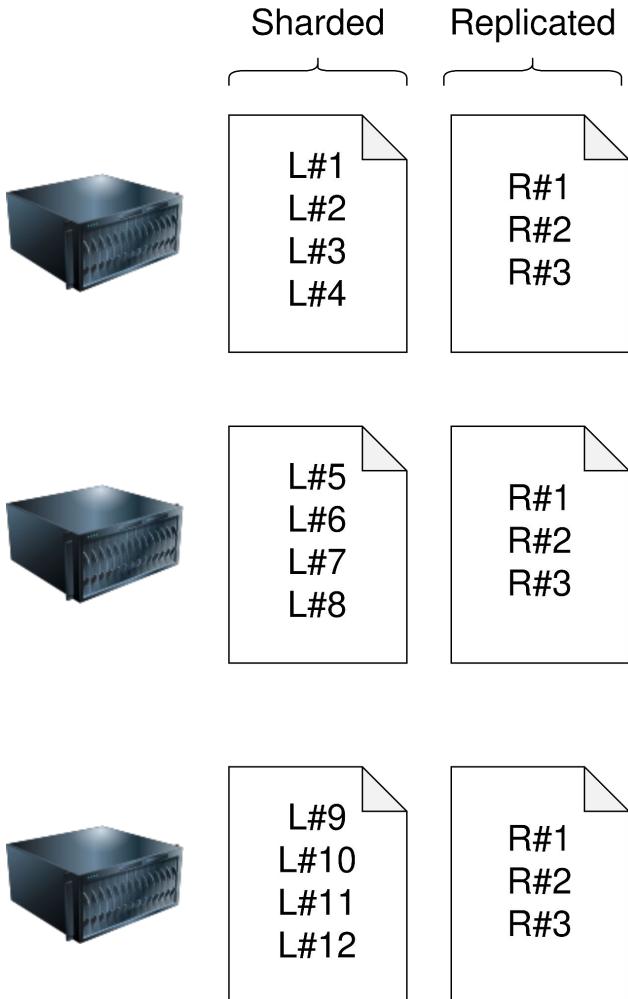
$$\frac{A_b \cdot m + C_b \cdot k}{k \cdot m} = \frac{A_b}{k} + \frac{C_b}{m}$$

строк вместо $A_b + C_b$

Map-only Join

- Меньшую таблицу (B, C) грузим на каждый маппер
- Строим индекс B → [Cs...]
- Большую таблицу (A, B) читаем на маппере построчно
- Редьюс не нужен

```
1 fun map(table, index):  
2     for ⟨a, b⟩ ← table:  
3         for c ← index.get(b):  
4             yield ⟨a, b, c⟩
```



Операции над множествами

- $A \cap B$
- $A \cup B$
- $A \setminus B$

```
1 fun mapper(doc):  
2     assert doc.side ∈ {Left, Right}  
3     for elem in doc:  
4         yield elem, doc.side
```

```
1 fun reduce_union(elem, sides):  
2     yield elem  
3  
4 fun reduce_intersection(elem, sides):  
5     if unique(sides) = {Left, Right}:  
6         yield elem  
7  
8 fun reduce_setminus(elem, sides):  
9     if unique(sides) = {Left}:  
10        yield elem
```

tf-idf: определение

- $tf(t, d)$ больше для слов, которые часто встречаются в документе d
 - Эти слова важны для документа d
 - $idf(t)$ меньше для слов, которые встречаются в большем числе документов
 - Это слова без смысловой нагрузки
- $$tf(t, d) = \frac{|\{i : d_i = t\}|}{|d|}$$
- $$idf(t) = \log \frac{|D|}{|d \in D : t \in d|}$$
- $$tfidf(t, d) = tf(t, d) \cdot idf(t)$$

tf-idf: подсчёт

- Считаем $cnt(t, d) = |\{i : d_i = t\}|$
- Если каждый документ лежит целиком на одном сервере, нам вообще не нужны распределённые вычисления



В каюте было темно, ощущалась небольшая качка, пол слегка подрагивал, как при легком землетрясении...



По правде сказать, я думаю, что я кончу гораздо хуже, чем начал. О, начал-то я совсем неплохо...



В каюте было темно, ощущалась небольшая качка, пол слегка подрагивал, как при легком землетрясении...



Когда Луций вошел во Дворец, сигналы отбоя в коридорах возвещали о конце боевой тревоги.

Документ целиком хранится на единственном сервере

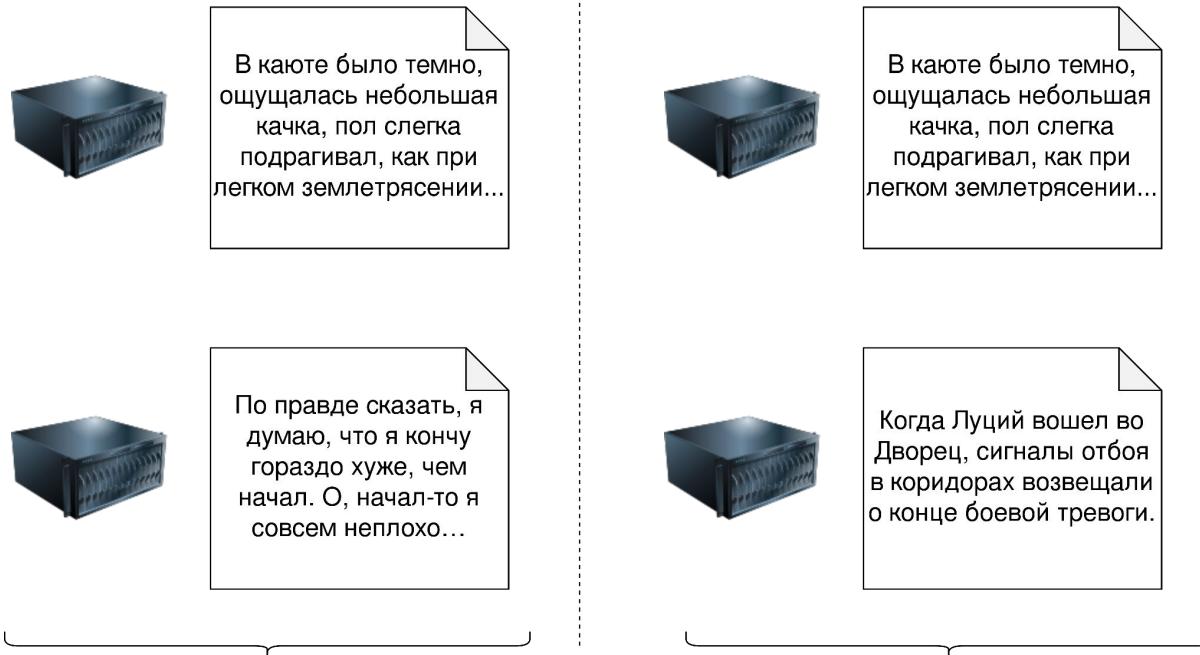
Документ может быть шардирован

tf-idf: подсчёт

- Считаем $cnt(t, d) = |\{i : d_i = t\}|$
- Иначе используем

```
1 fun mapper(doc):  
2     for word in words(doc):  
3         yield < doc.id, word >, 1  
4  
5 fun reducer(< doc_id, word >, ones):  
6     yield < doc_id, word >, sum(ones)
```

- Можем использовать combiner



tf-idf: подсчёт

- Аналогично $sz(d) = |d|$

- Если нужно, используем

```
1 fun mapper(doc):  
2     for word in words(doc):  
3         yield doc.id, word, 1  
4  
5 fun reducer(doc_id, ones):  
6     yield doc_id, sum(ones)
```

- Можем использовать combiner



В каюте было темно, ощущалась небольшая качка, пол слегка подрагивал, как при легком землетрясении...



По правде сказать, я думаю, что я кончу гораздо хуже, чем начал. О, начал-то я совсем неплохо...



В каюте было темно, ощущалась небольшая качка, пол слегка подрагивал, как при легком землетрясении...



Когда Луций вошел во Дворец, сигналы отбоя в коридорах возвещали о конце боевой тревоги.

{

Документ целиком хранится на единственном сервере

{

Документ может быть шардирован

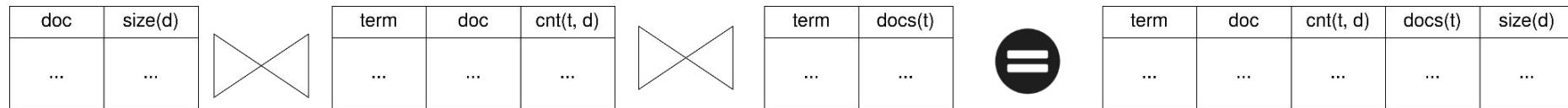
tf-idf: подсчёт

- Считаем $docs(t) = |d \in D : t \in d|$
- Без распределённых вычислений не обойтись
- Если документы не шардируются, используем unique только в коде combiner
- В коде редьюсера просто считаем количество

```
1 fun mapper(doc):  
2     for word in words(doc):  
3         yield word, doc.id  
4  
5 fun reducer(word, ids):  
6     yield word, unique(words).size()
```

tf-idf: подсчёт

- Есть таблицы:
 - DocSizes (doc, size)
 - DocTermCount (doc, term, count)
 - DocsByTermCount (term, count)
- Делаем два соединения



- Построчно считаем
$$tfidf(t, d) = \frac{cnt(t, d)}{size(d)} \cdot \log \frac{|D|}{docs(t)}$$
- В reduce-фазе последнего соединения

Что почитать

- *Dean J., Ghemawat S.* MapReduce: simplified data processing on large clusters
- *DeWitt D. J. et al.* The Gamma database machine project.
- *Lin J., Dyer C.* Data-intensive text processing with MapReduce
- *Anand R., Jeffrey David U.* Mining of massive datasets
 - [youtube.com](https://www.youtube.com)
- Александр Петров. Принципы работы с большими данными, парадигма MapReduce
- Александр Петров. Приемы и стратегии разработки MapReduce-приложений

Thanks for your attention

