

Если лекция про

# Распределённые вычисления

была так хороша, то где же сиквел?

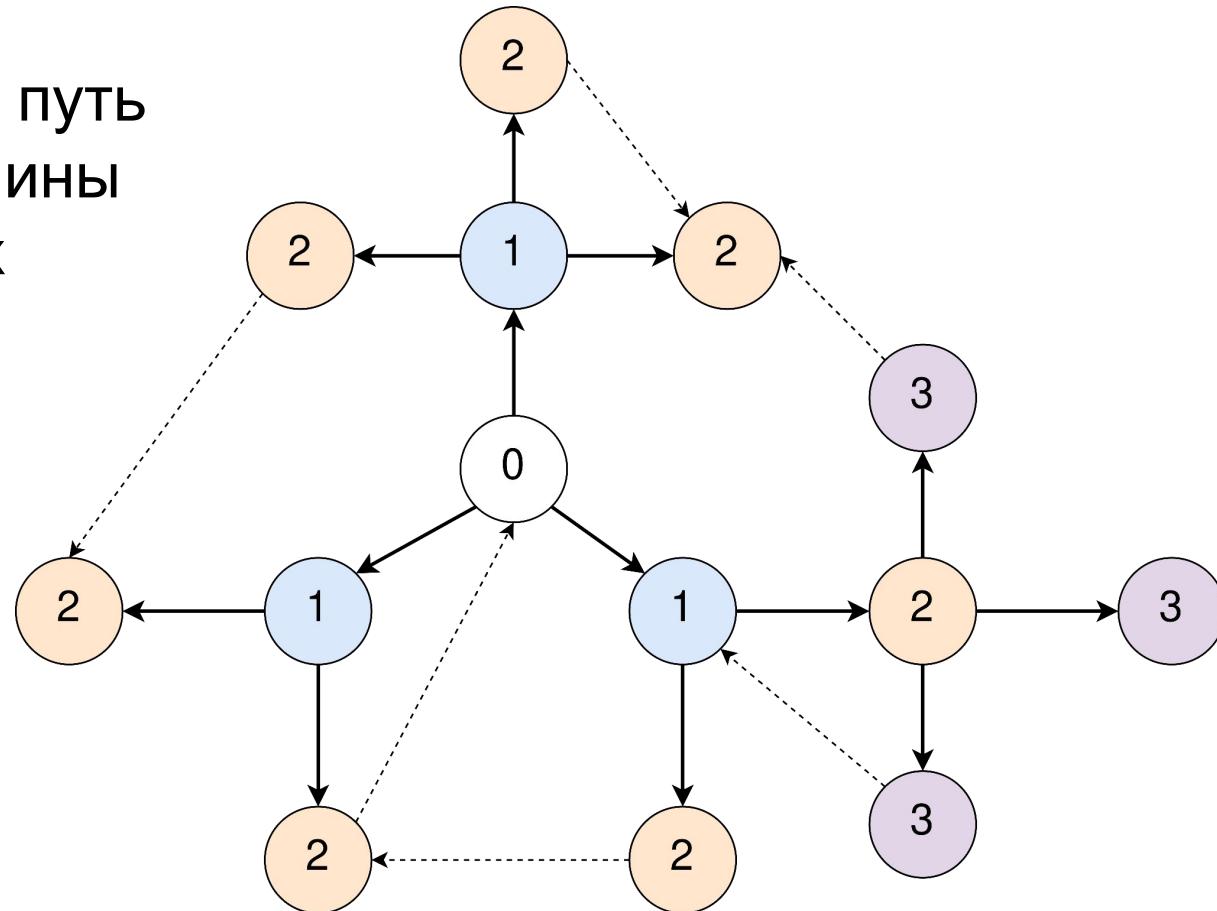


Илья Кокорин

[kokorin.ilya.1998@gmail.com](mailto:kokorin.ilya.1998@gmail.com)

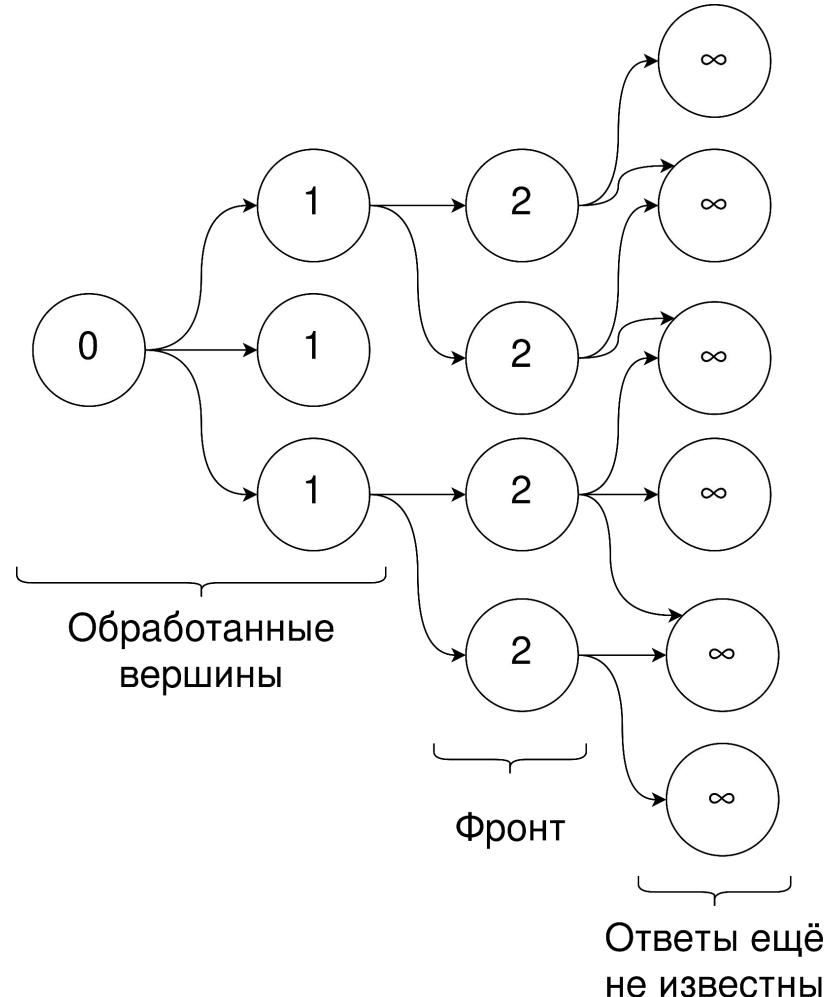
# BFS

- Ищем кратчайший путь от стартовой вершины до всех остальных
- Обходим сначала **все** вершины, расстояние до которых равно 1
- Потом 2
- Потом 3
- ...



# BFS: три типа вершин

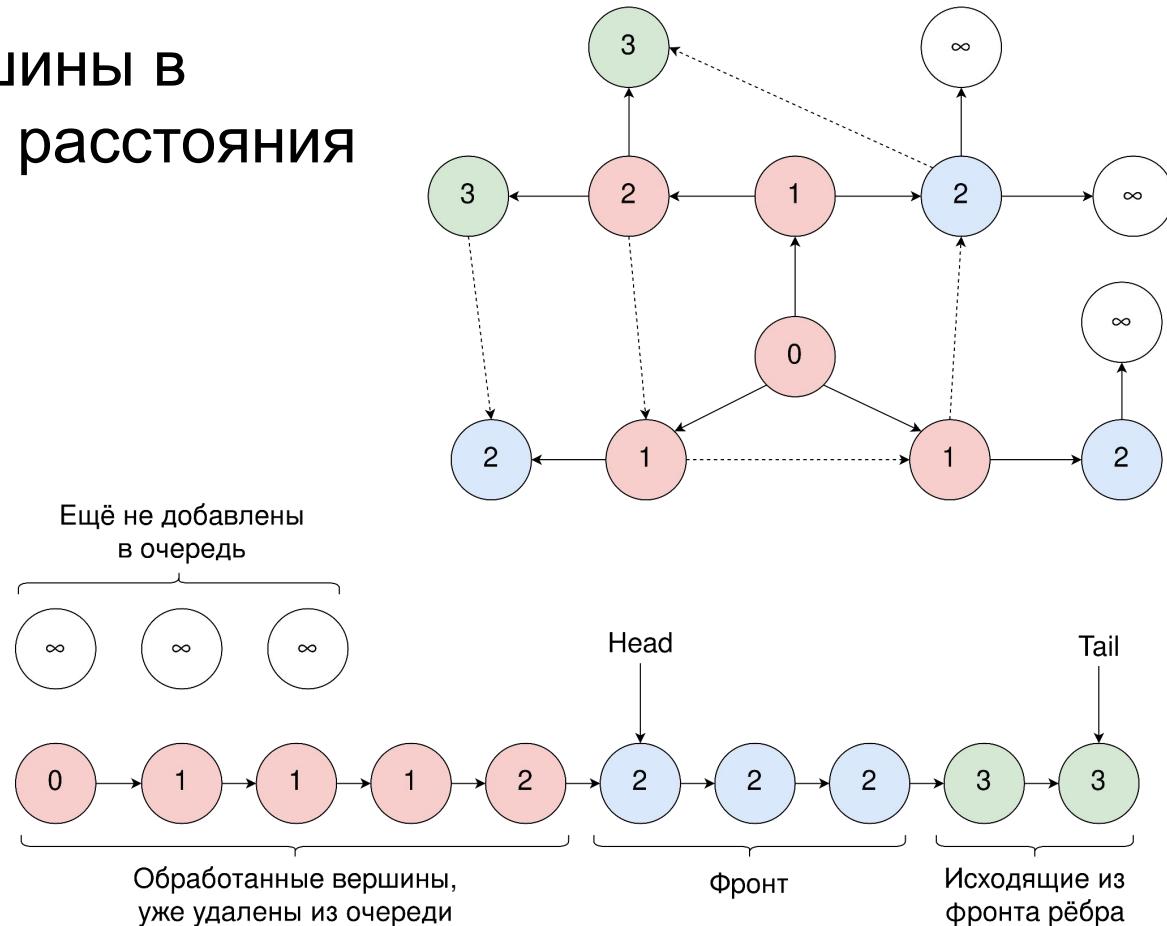
- Расстояние известно, все исходящие рёбра обработаны
  - Знаем расстояния до всех соседей
- Расстояние известно, не все исходящие рёбра обработаны
  - Wave front
- Расстояние неизвестно



# BFS: последовательный алгоритм

- Обрабатываем вершины в порядке увеличения расстояния

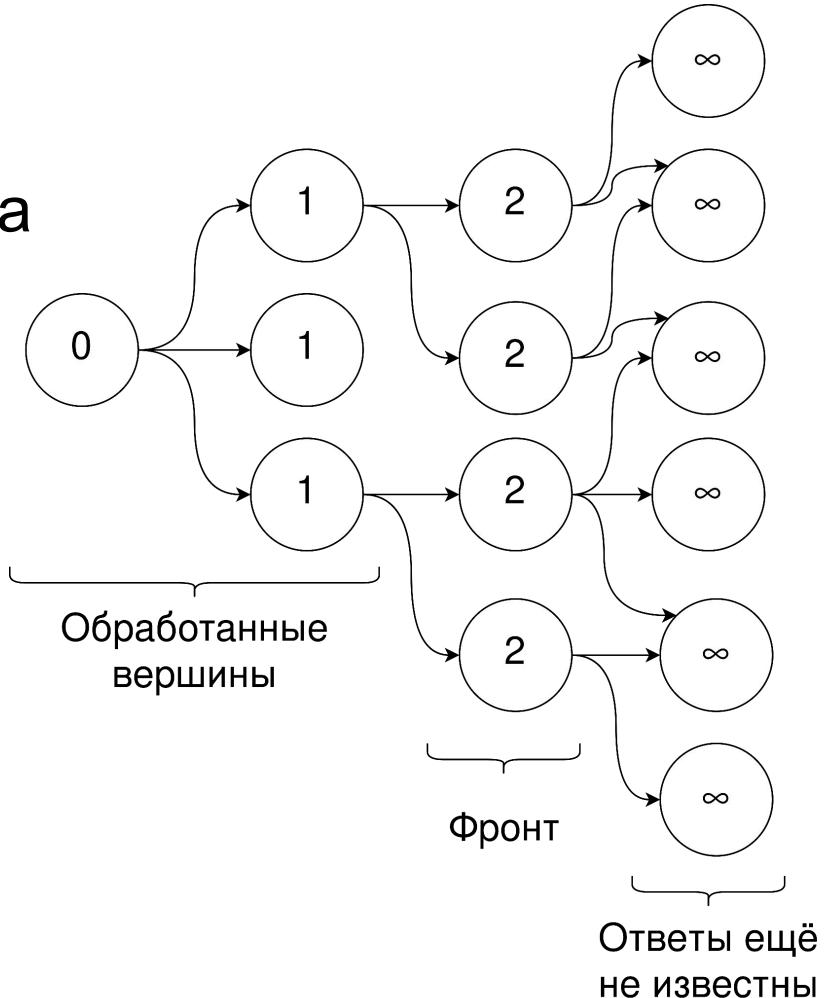
```
1 dist[1 ... | V |] ← ∞  
2 dist[s] = 0  
3 queue = new Queue()  
4 queue.push(s)  
5 while not queue.empty():  
6     v = queue.pop()  
7     assert dist[v] ≠ ∞  
8     for (v, u) in E:  
9         if dist[u] ≠ ∞:  
10             dist[u] = dist[v] + 1  
11             queue.push(u)
```



# Параллельный BFS: идея

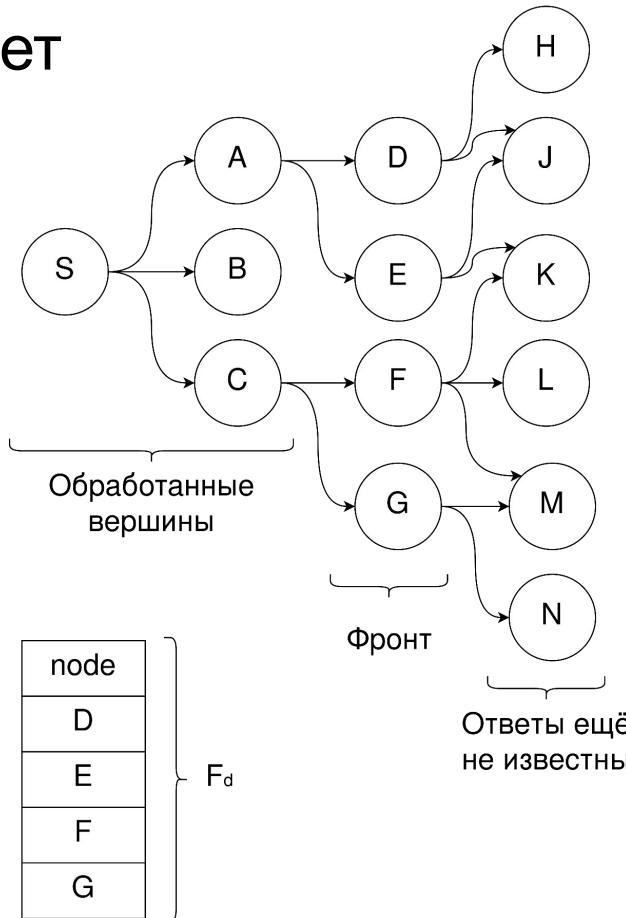
- Параллельно обрабатываем все исходящие из фронта рёбра

```
1  $F_{d+1} := \emptyset$ 
2 pfor  $v$  in  $F_d$ :
3   pfor  $(v, u) \in E$ :
4     if  $\text{dist}[u] = \infty$ :
5        $\text{dist}[u] \leftarrow d + 1$ 
6        $F_{d+1} \leftarrow F_{d+1} \cup u$ 
```



# Распределённый BFS: множества

- $R_d(\text{node}, \text{dist})$  — ответ известен
  - $\text{dist} \leq d$
- $F_d$  — фронт
  - $\text{dist} = d$
- $U_d$  — ответ неизвестен
  - $R_d \cup U_d = V$

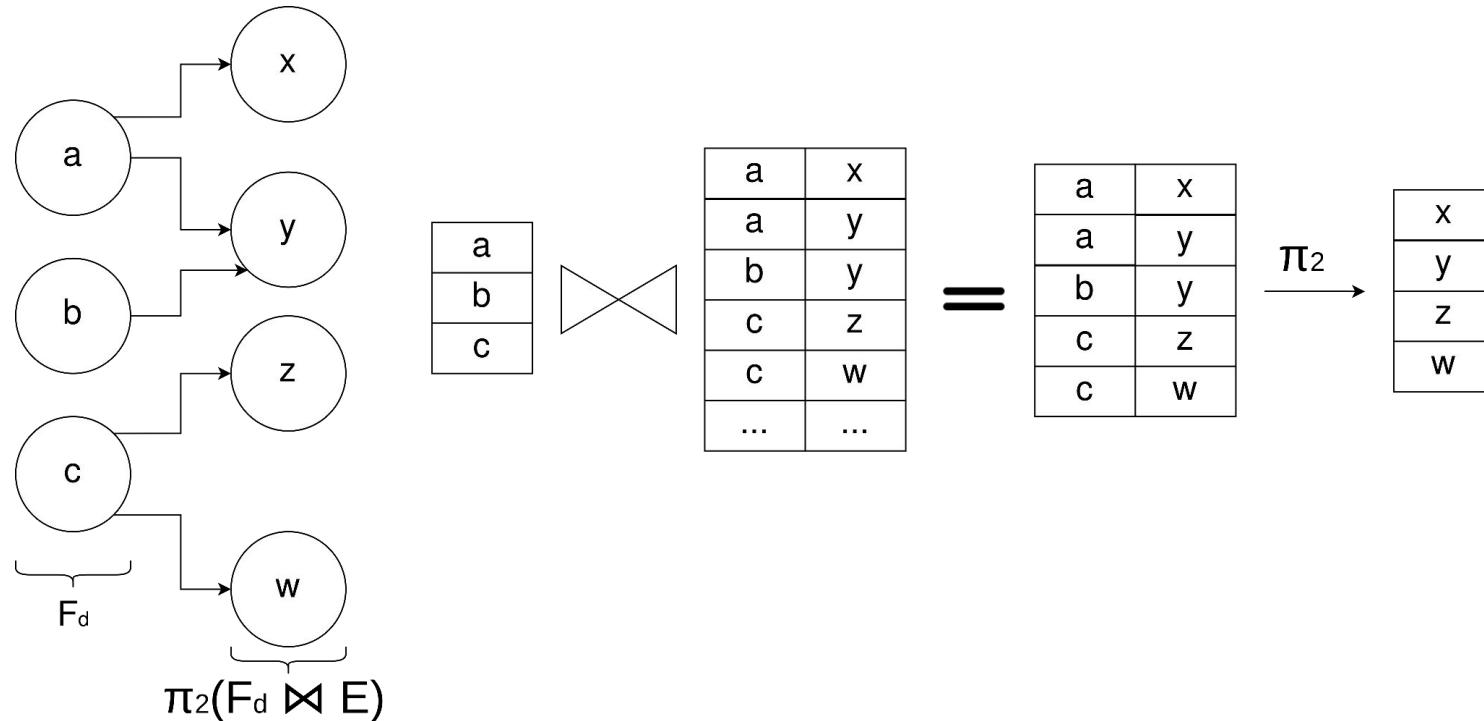


node	dist
S	0
A	1
B	1
C	1
D	2
E	2
F	2
G	2

node
H
J
K
L
M
N

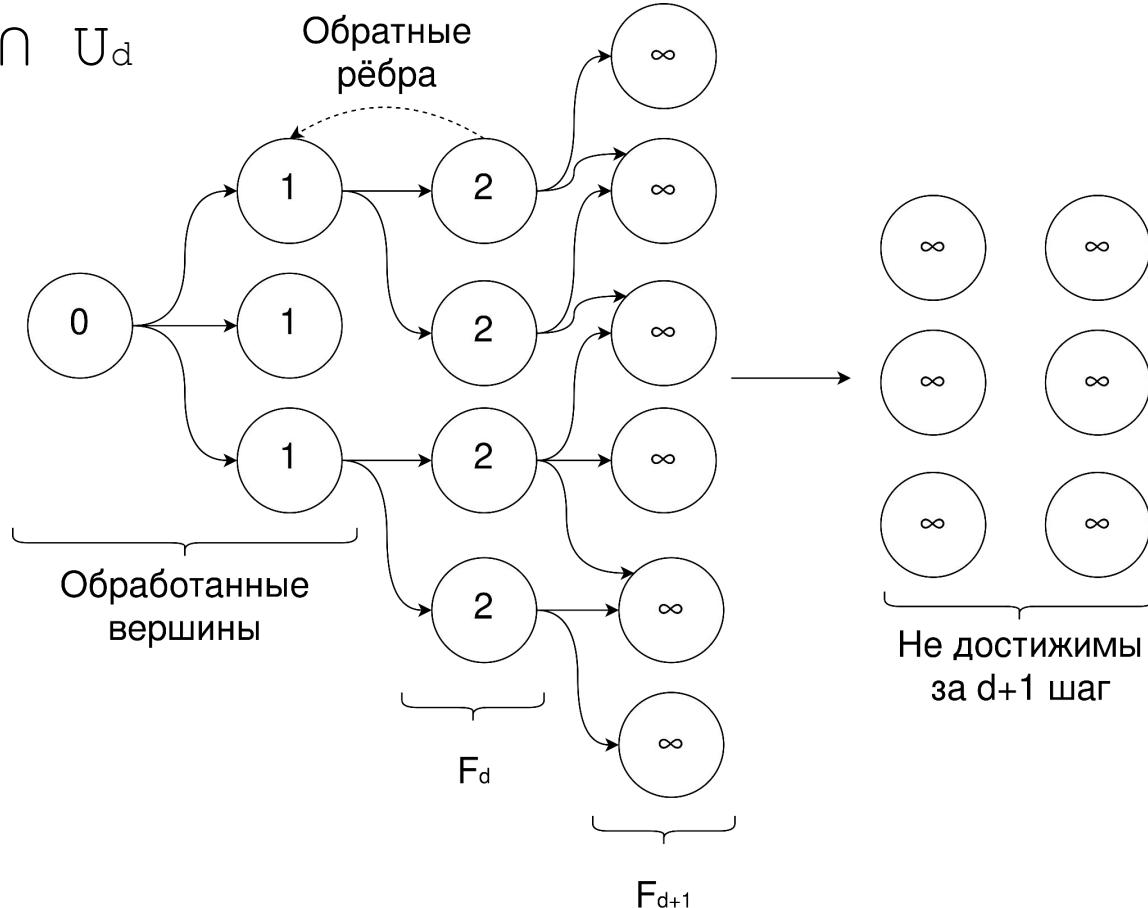
# Распределённый BFS: переход

- Вершины  $\pi_2(F_d \bowtie E)$  достижимы из фронта
- Не забываем выкинуть дубликаты



# Распределённый BFS: переход

- $F_{d+1} = \pi_2(F_d \bowtie E) \cap U_d$
- Чтобы не добавлять во фронт обработанные ранее вершины
- Достижимые по обратным рёбрам в дереве обхода

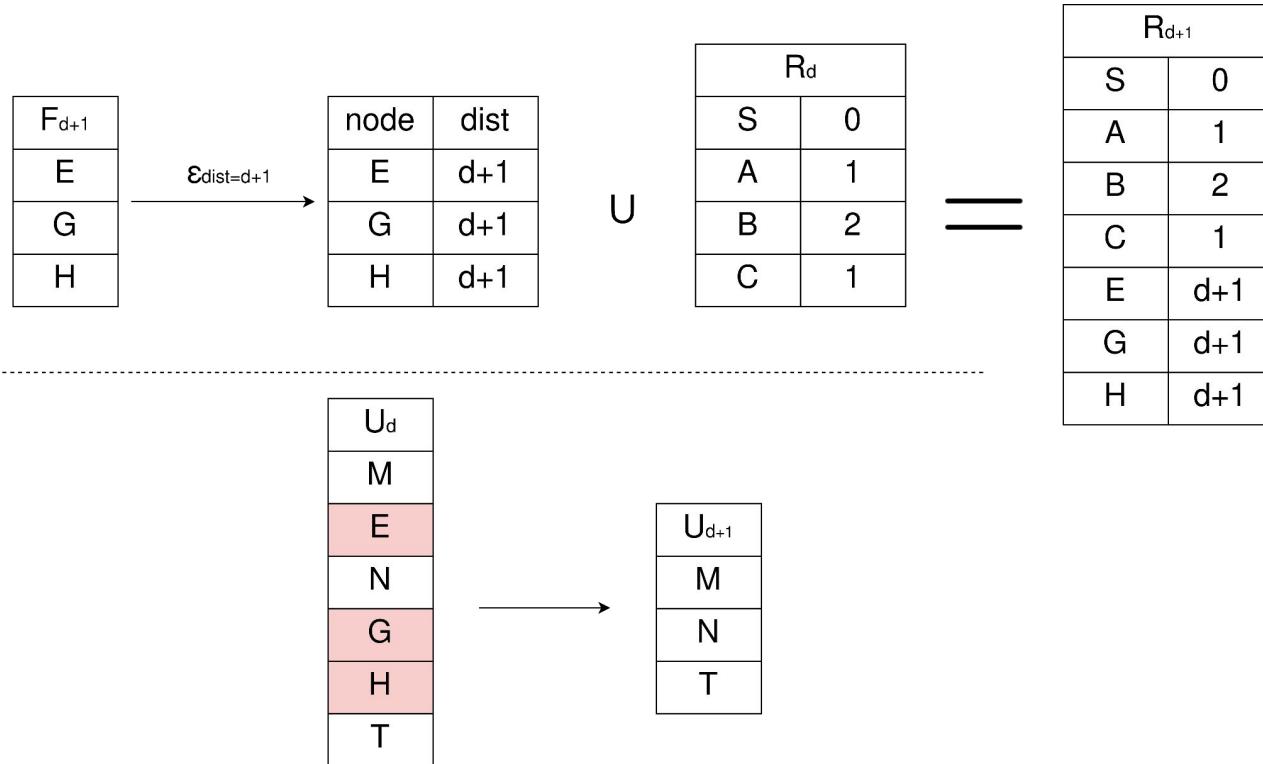


# Распределённый BFS: переход

- $R_{d+1} = R_d \cup \varepsilon_{dist=d+1}(F_{d+1})$

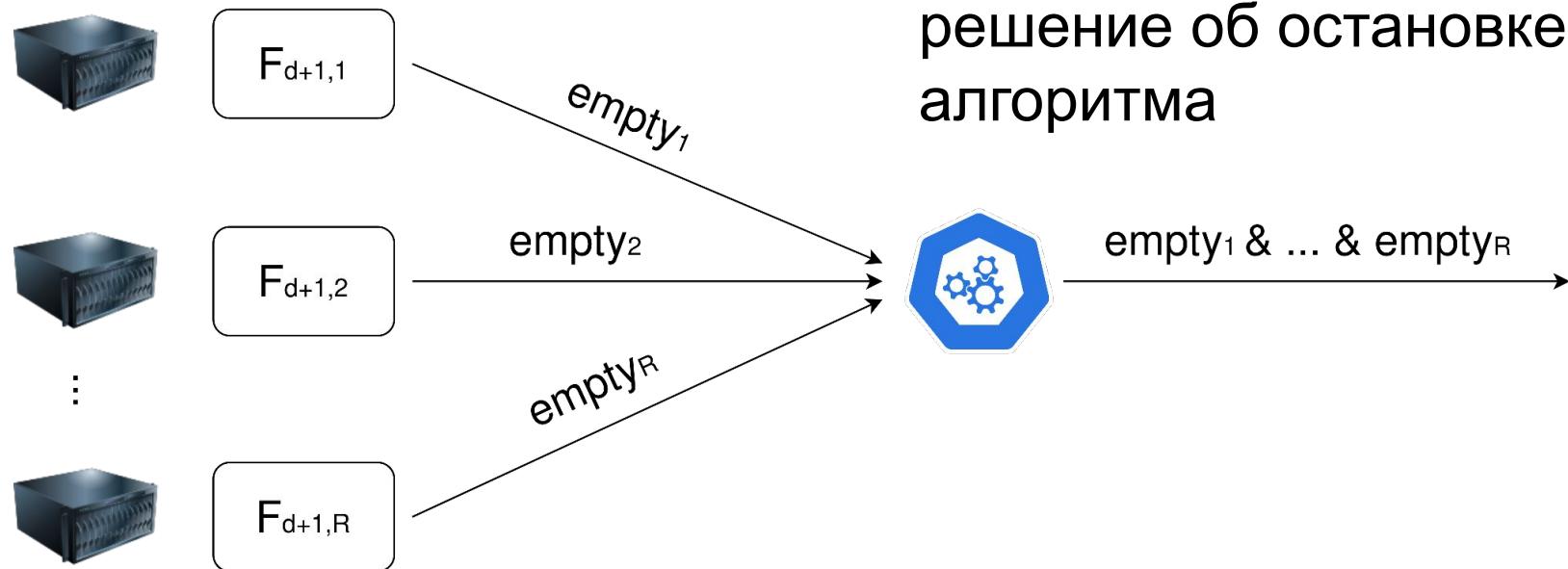
- Ответ стал известен

- $U_{d+1} = U_d \setminus F_{d+1}$



# Распределённый BFS: остановка

- Останавливаемся когда  $F_{d+1} = \emptyset$
- Каждый редьюсер сообщает координатору, пуста ли его партиция

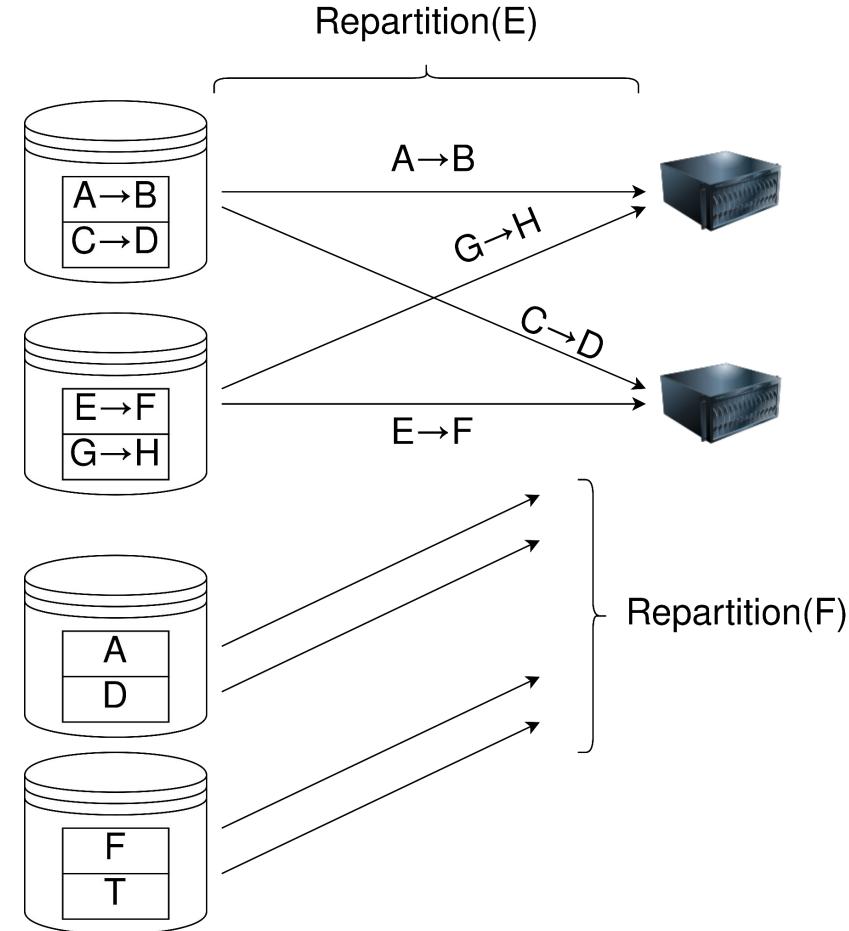


# Распределённый BFS: алгоритм

```
1 fun BFS(s, V, E):
2     R := {(s, 0)}
3     U := V \ {s}
4     F := {s}
5     d := 0
6     while true:
7         assert U ∪ π2(R) = V
8         N := F ⋈ E
9         F ← π2(N) ∩ U
10        if F = ∅:
11            return R, U
12        R ← R ∪ εdist=d+1(F)
13        U ← U \ F
14        d ← d + 1
```

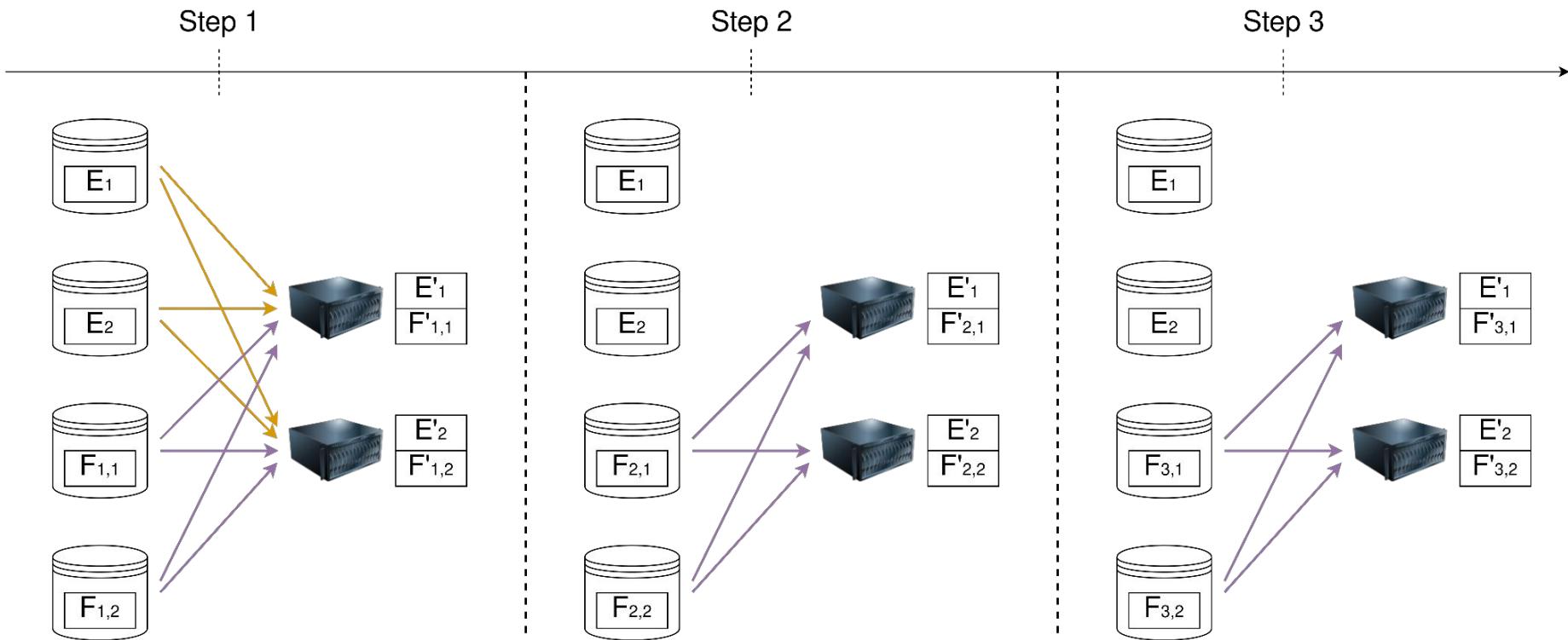
# Распределённый BFS: оптимизация

- Каждый раз, считая  $F_d \bowtie E$ , вынуждены делать repartition обеих частей выражения
- $F_d$  новый на каждом шаге
  - Придётся на каждом шаге делать repartition
- $E$  инвариантно
  - Хотим сделать repartition всего один раз



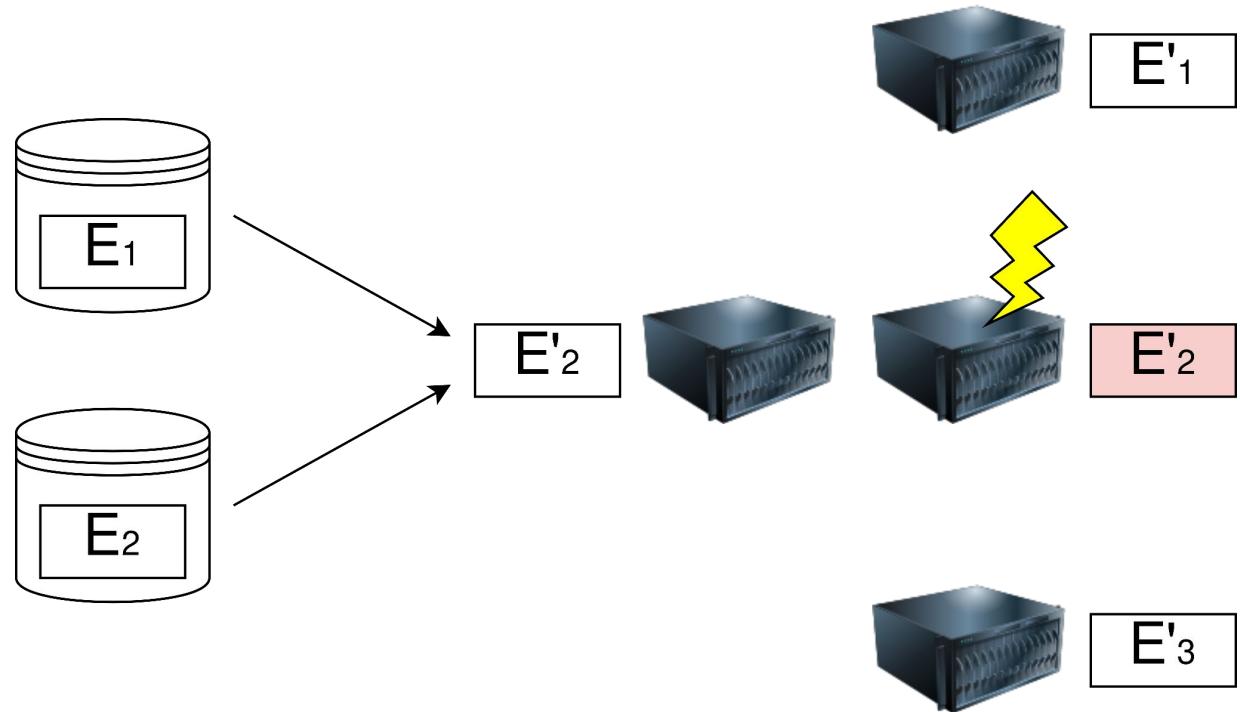
# Распределённый BFS: оптимизация

- Кешируем результаты repartition E на первом шаге



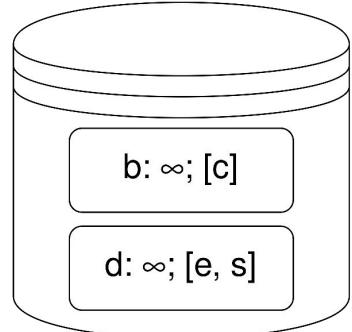
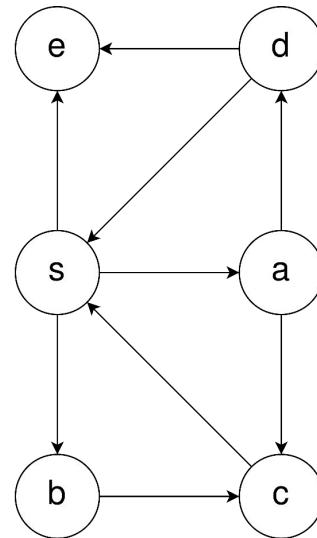
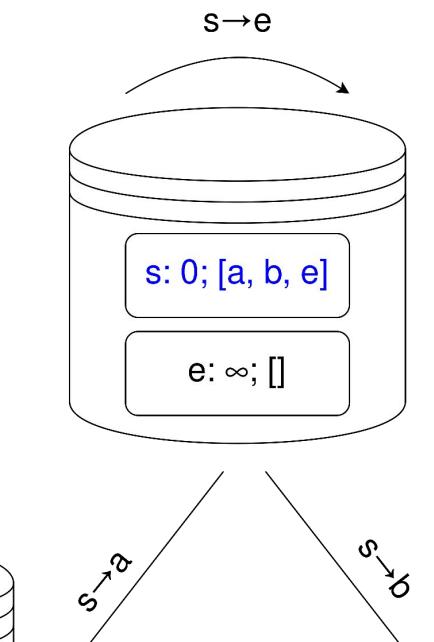
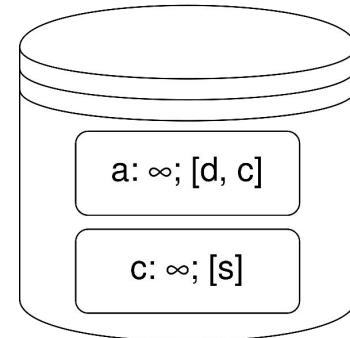
# Распределённый BFS: сбои

- При сбое кеширующего редьюсера пересчитываем потерянную партицию



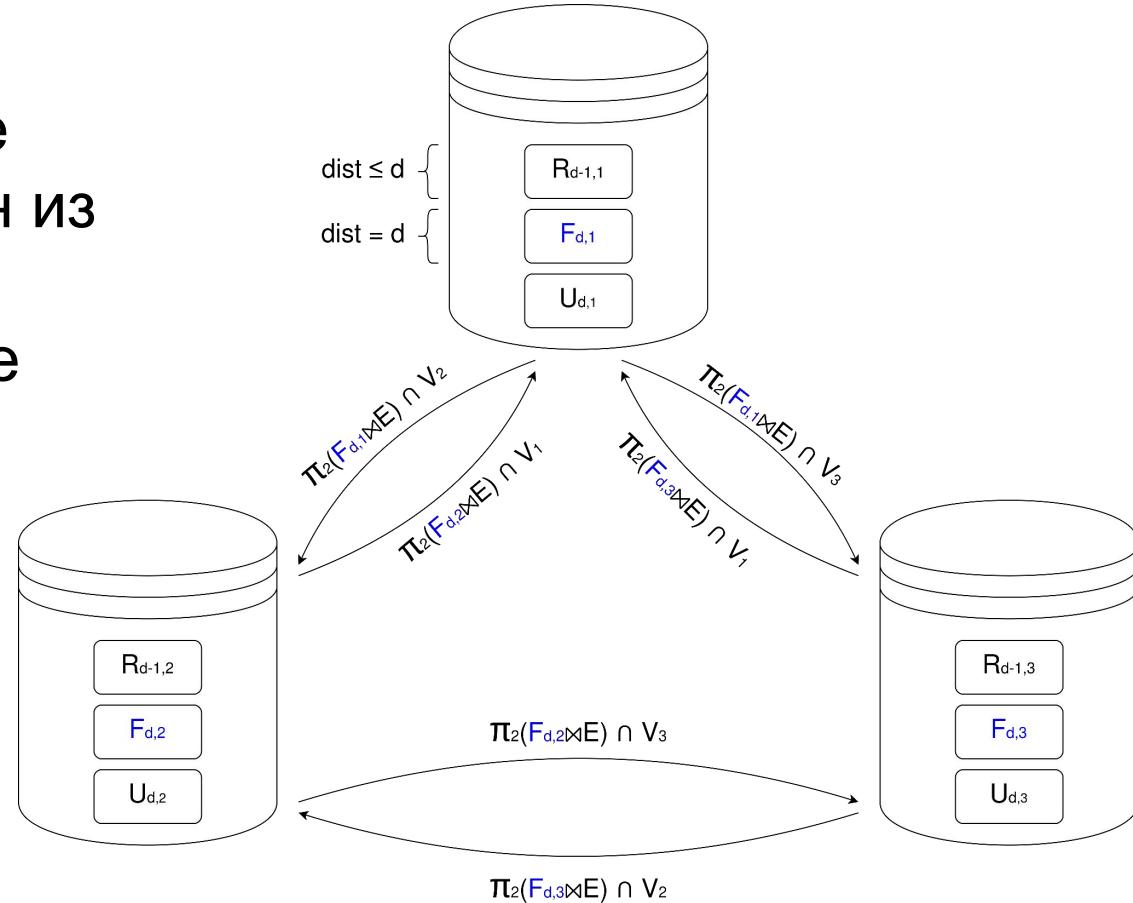
# BFS за один Repartition

- Пусть граф шардирован по кластеру
- Сервер  $S$  хранит вершину  $v$  и все исходящие из неё рёбра
- На каждом шаге посылаем сообщение всем соседям вершин из фронта



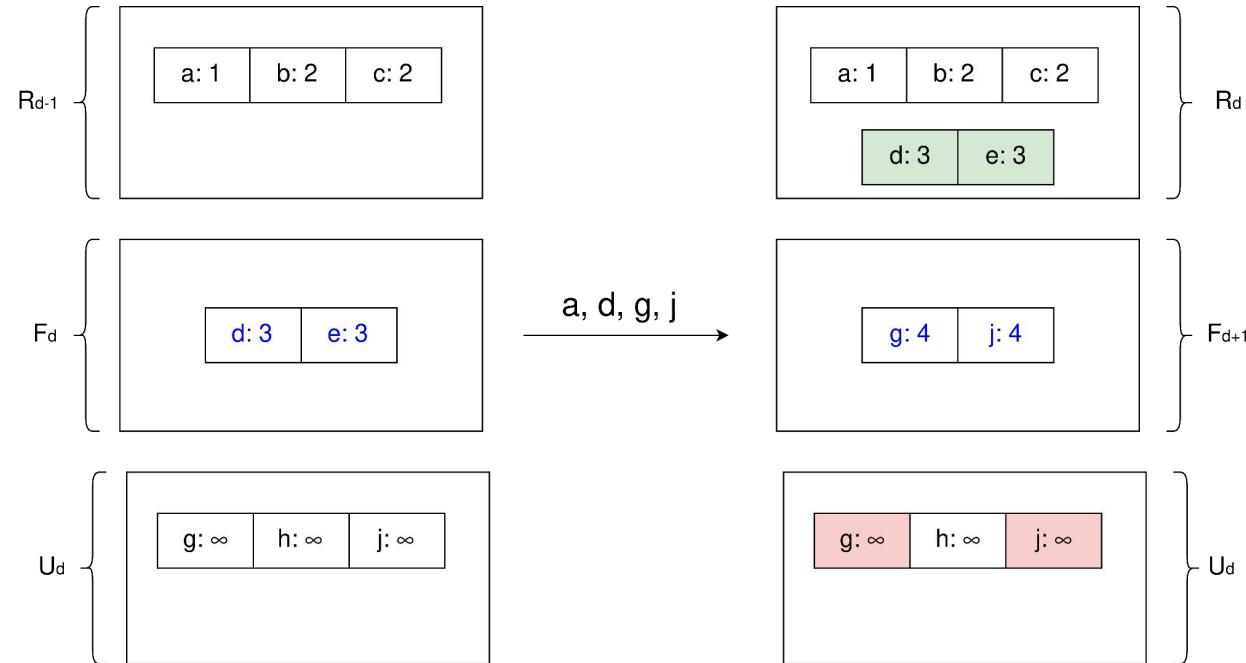
# BFS за один Repartition

- На каждом шаге посылаем сообщение всем соседям вершин из фронта
- Посылаем сообщение серверам, хранящим соседние вершины



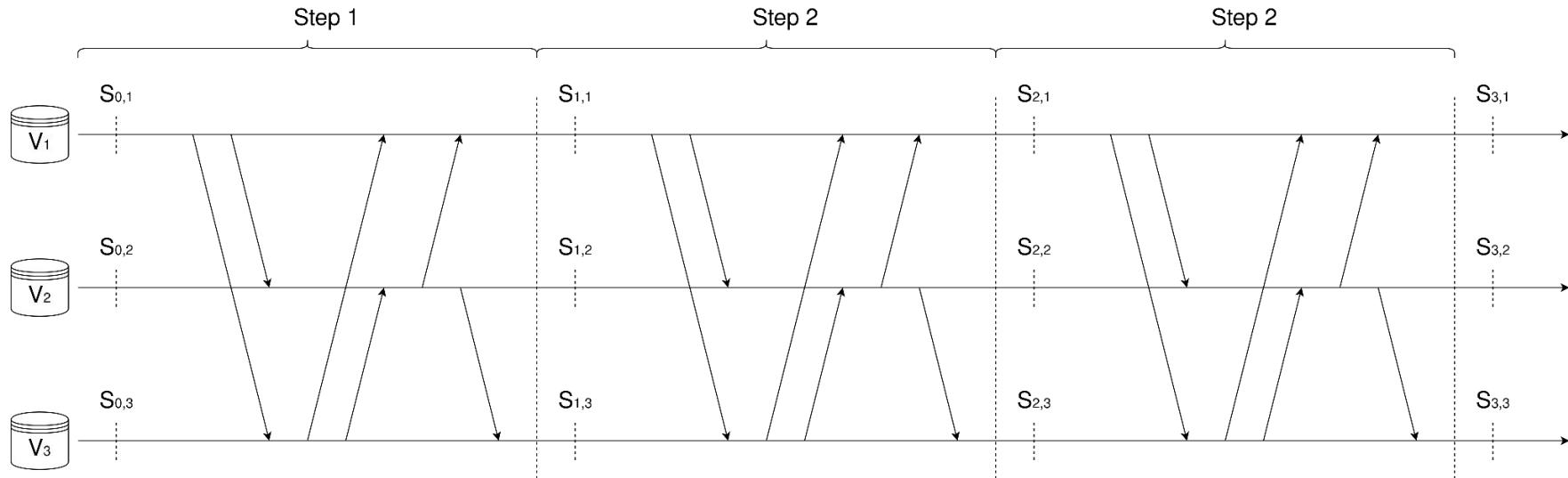
# BFS за один Repartition: обновление состояния

- После получения сообщений от **всех** серверов обновляем состояние
- Переносим вершины из фронта с множеством с известным ответом
- Вершины, для которых стал известен ответ, переносим во фронт



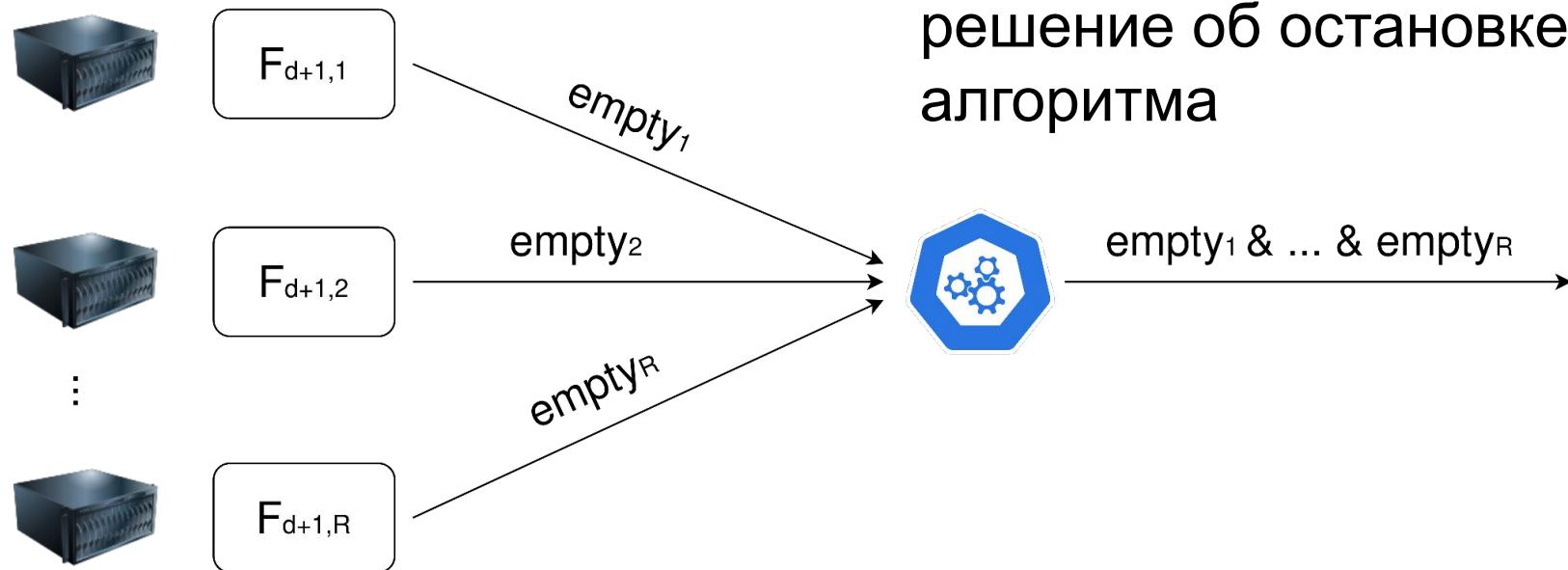
# Pregel Computational Model

- На каждом шаге каждый сервер посылает сообщения
  - В BFS посылаем соседям текущего фронта
- Получает сообщения от остальных серверов
- Обновляет локальное состояние



# Pregel BFS: остановка

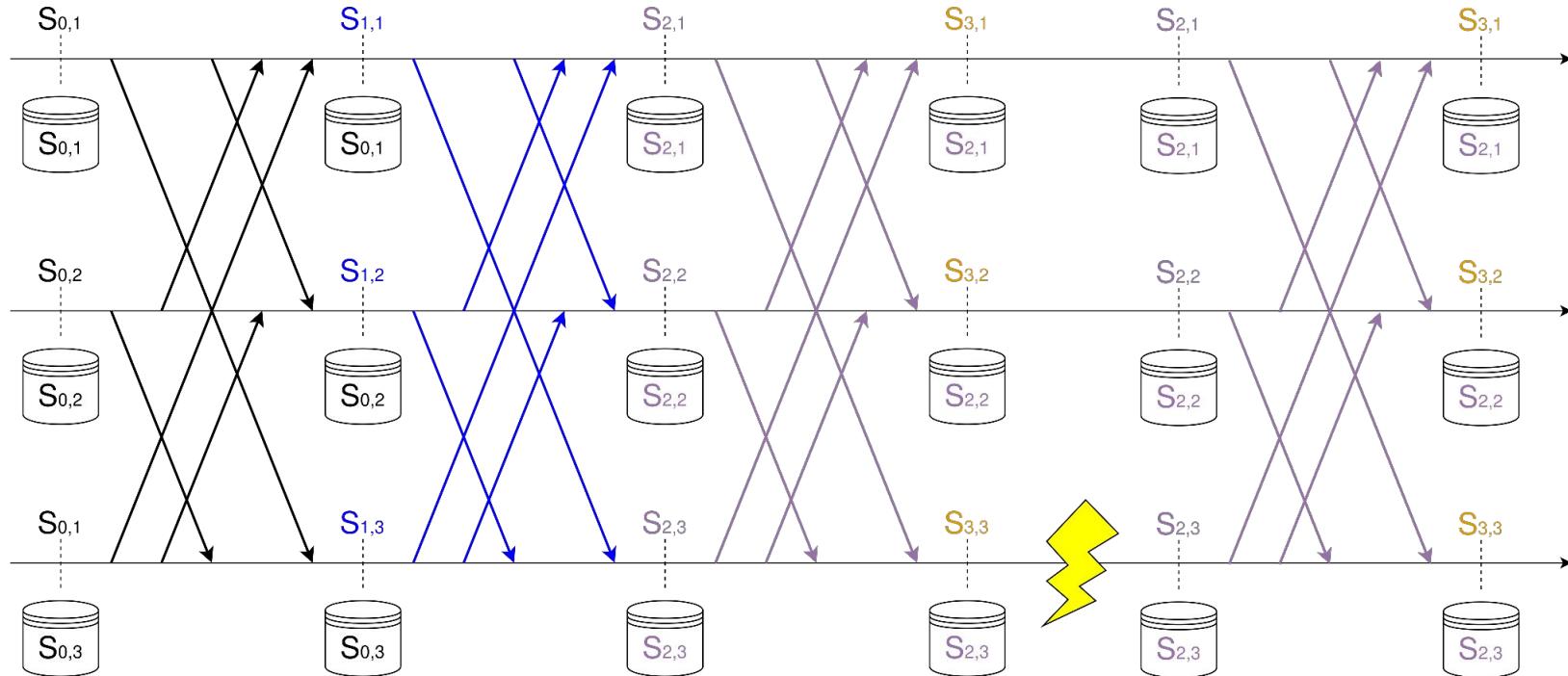
- Останавливаемся когда  $F_{d+1} = \emptyset$
- Каждый редьюсер сообщает координатору, готов ли он остановиться



- Координатор принимает решение об остановке алгоритма

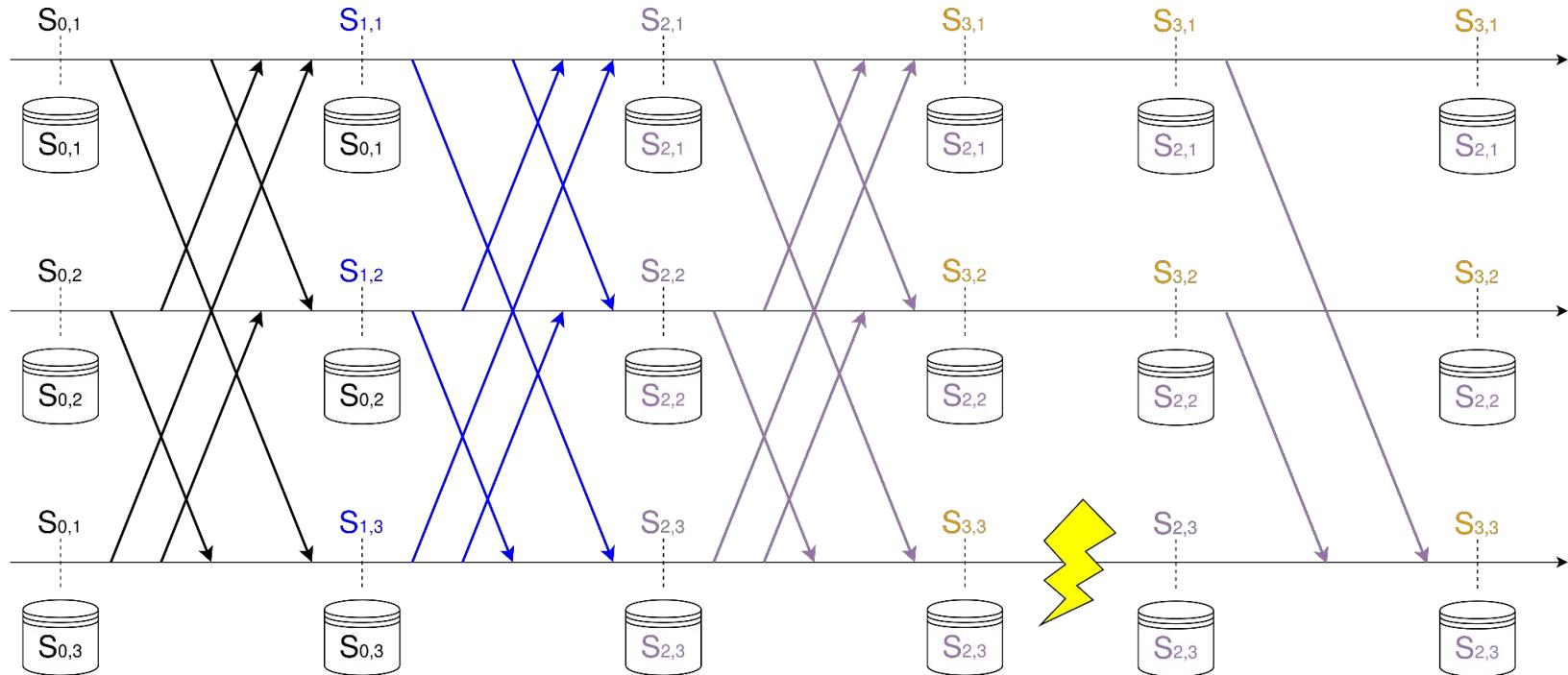
# Pregel: обработка сбоев

- Периодически делаем снимки состояния
- В случае сбоя откатываем **всех** на последний снимок



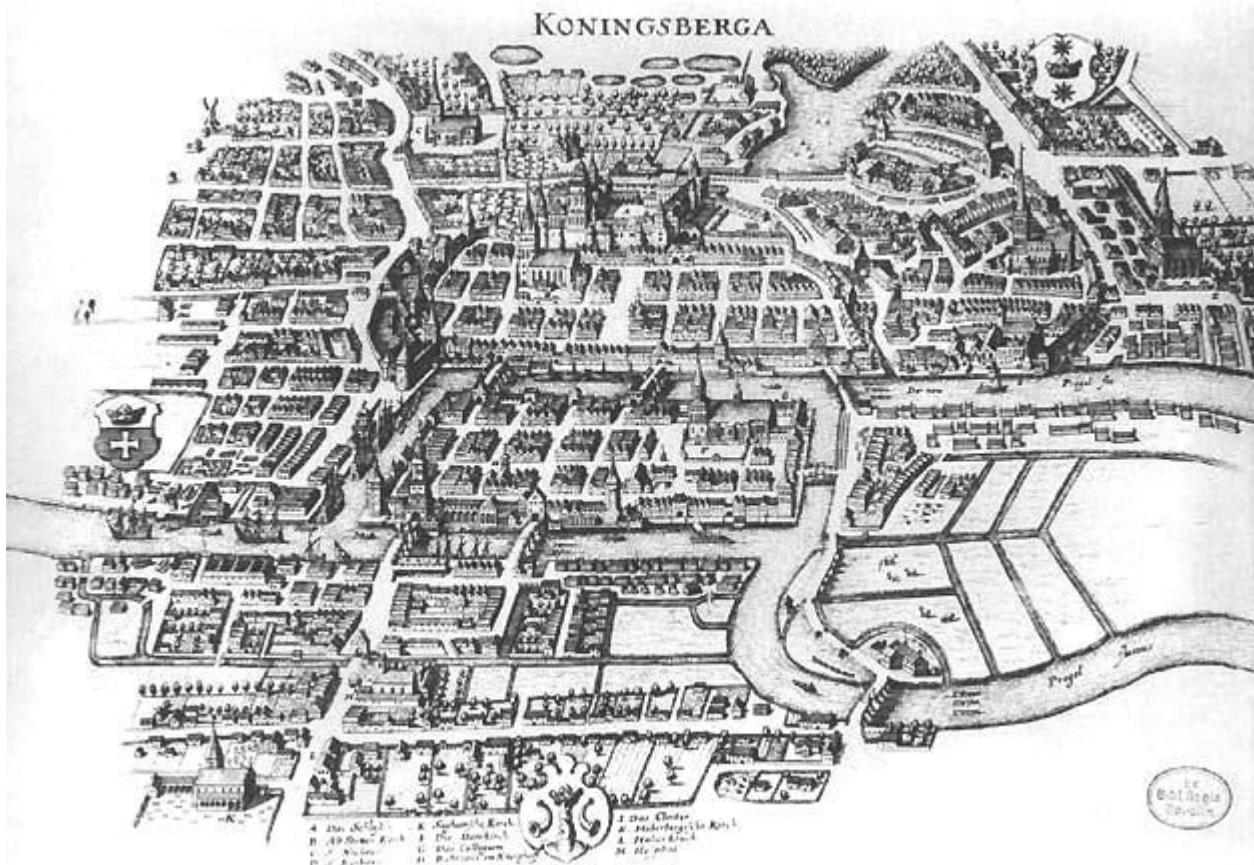
# Pregel: обработка сбоев

- Чтобы не повторять старые шаги, можно просто репленировать сообщения из прошлых шагов на сбойный сервер



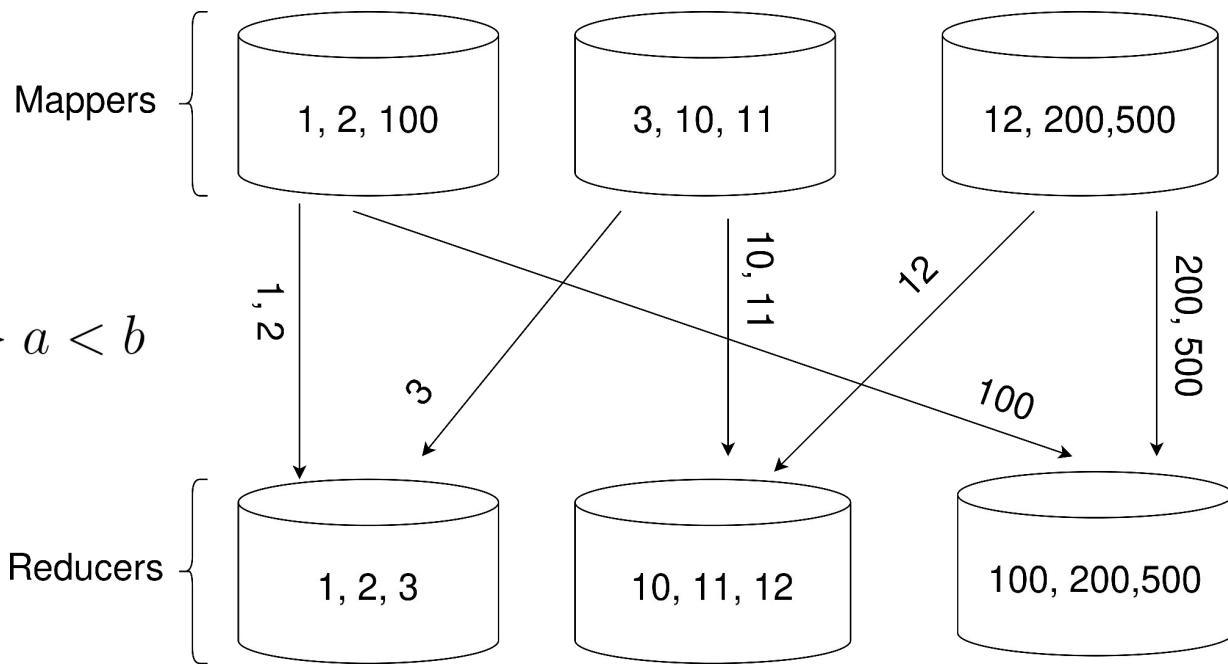
# Pregel: минутка истории

- Задача о мостах Кёнигсберга
  - Леонард Эйлер, 1736
  - Стала началом современной теории графов



# Распределённая сортировка: постановка задачи

- До: ~~карты~~ ключи расположены в ~~другом~~ произвольном порядке по узлам
  - Вы их в киосках что ли заряжаете?



# Распределённая сортировка: алгоритм

- Будем выбирать редьюсера в зависимости от интервала, в который попадает ключ
- Очевидно, получаем решение задачи
- Но как выбрать границы интервалов?

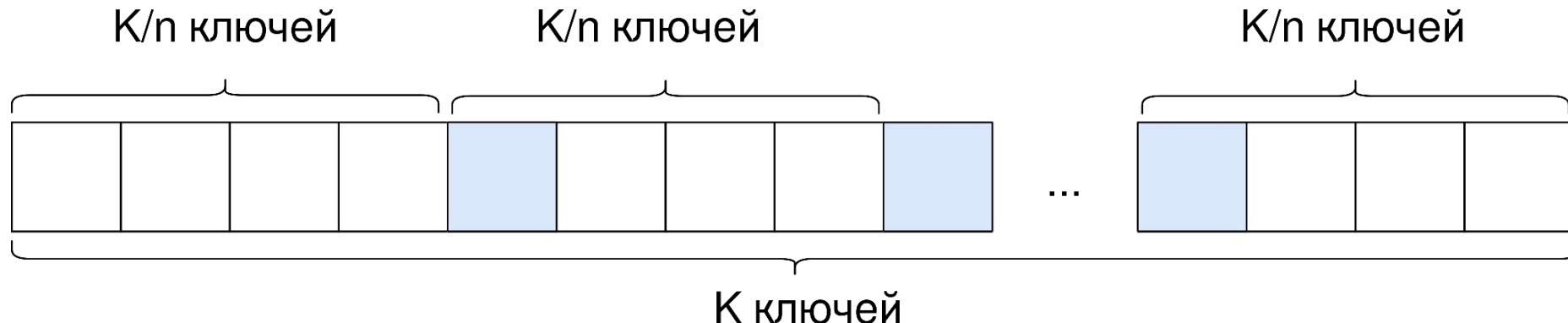
$$R(x) = \begin{cases} R_1 & x < a_1 \\ R_2 & a_1 \leq x < a_2 \\ R_3 & a_2 \leq x < a_3 \\ \dots \\ R_{n-1} & a_{n-2} \leq x < a_{n-1} \\ R_n & x \geq a_{n-1} \end{cases}$$

# Распределённая сортировка: выбор границ

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

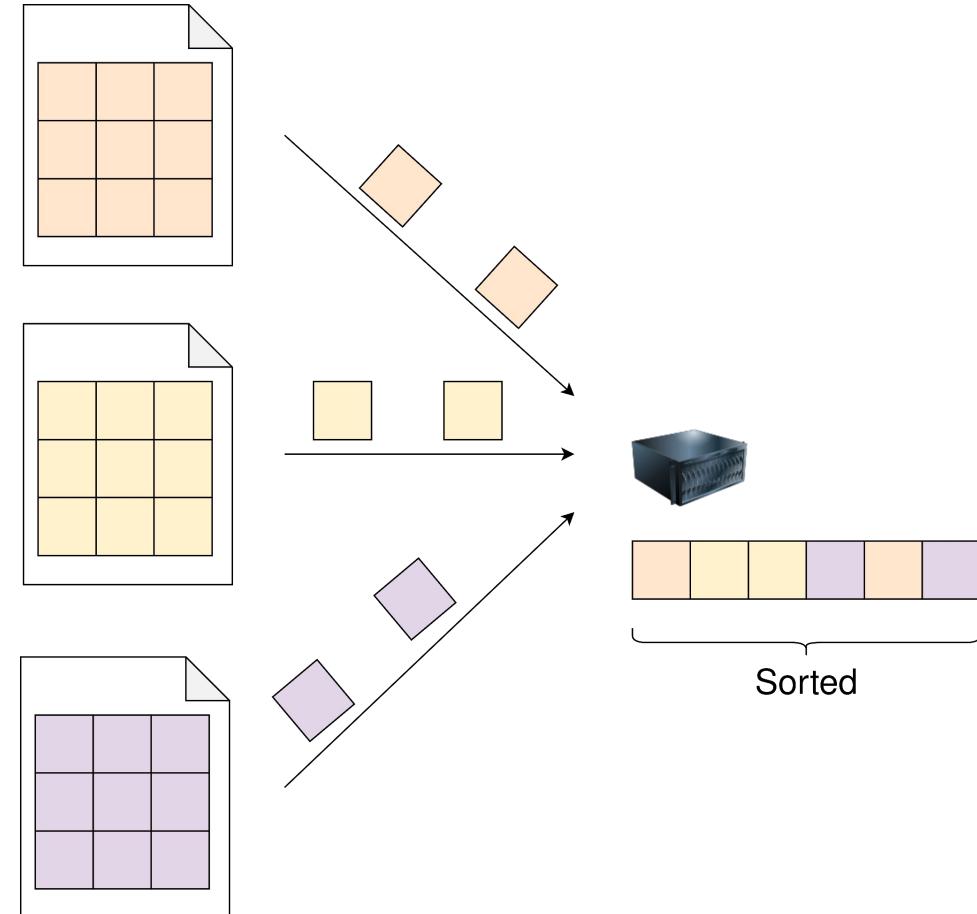
$$\mathbb{P}(x < a_1) = \mathbb{P}(a_2 \leq x < a_3) = \dots = \mathbb{P}(a_{n-2} \leq x < a_{n-1}) = \mathbb{P}(x \geq a_{n-1})$$

- Если бы у нас был отсортированный массив ключей, границы выбирались бы тривиально



# Распределённая сортировка: выбор границ

- Случайным образом выберем небольшое число ключей с каждого маппера
- Перенесём их на один сервер
- Выберем границы

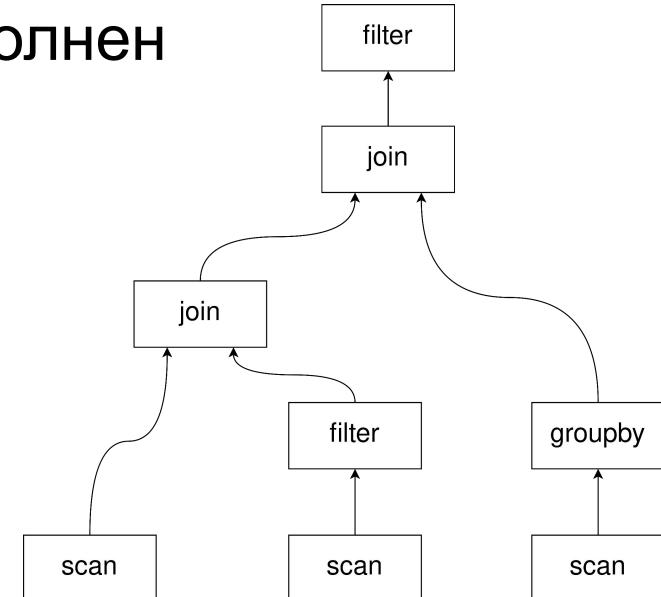


# Высокоуровневые языки обработки данных

- Каждый оператор может быть выполнен MapReduce-like алгоритмом

```
good_urls = FILTER urls BY pagerank > 0.2;  
groups = GROUP good_urls BY category;  
big_groups = FILTER groups BY COUNT(good_urls)>106;  
output = FOREACH big_groups GENERATE  
    category, AVG(good_urls.pagerank);
```

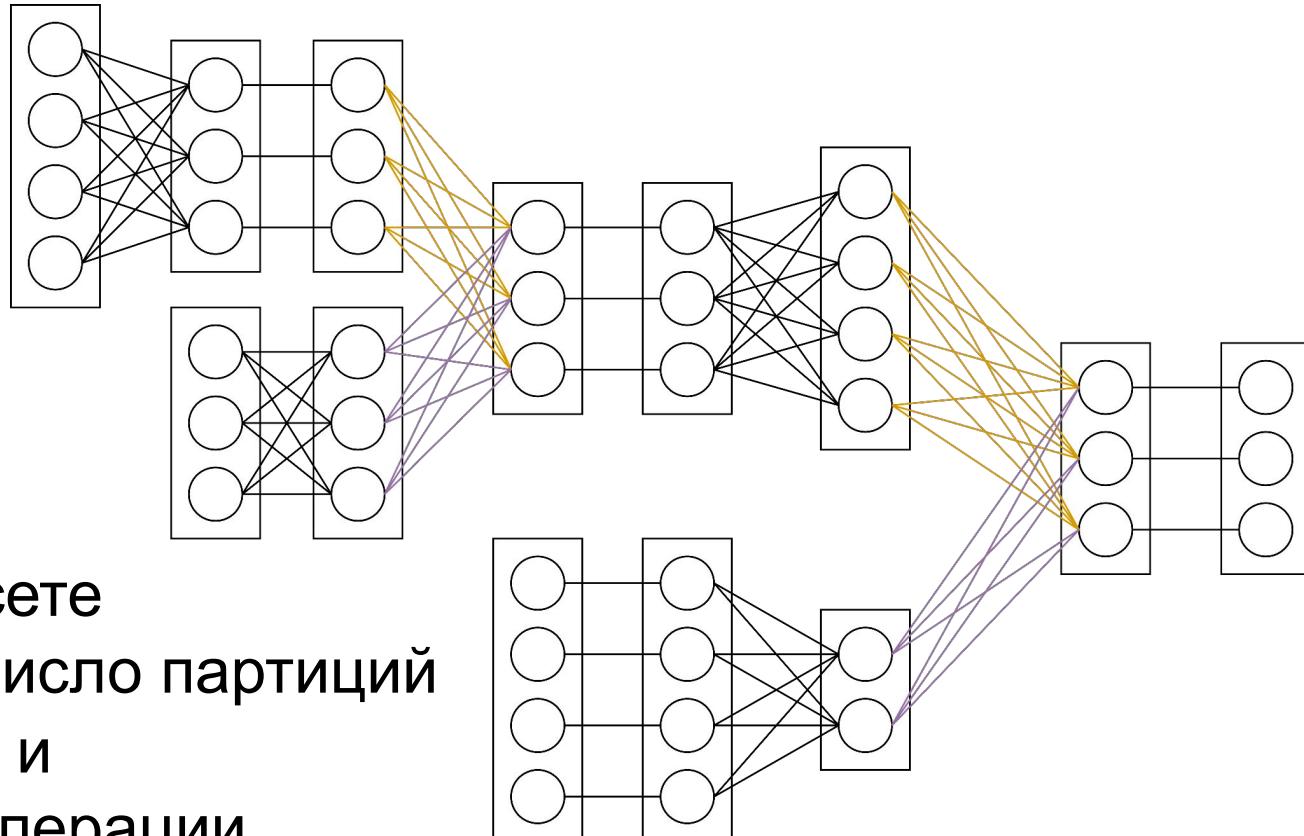
```
SELECT DISTINCT Students.StudentId  
FROM Students EXCEPT ALL (  
    SELECT DISTINCT Students.StudentId  
    FROM Students  
        NATURAL JOIN Marks  
        NATURAL JOIN Plan  
    WHERE Plan.LecturerId = ?  
);
```



```
text.map(normalize_string)  
.filter(lambda line: line != '')  
.flatMap(lambda line: line.split())  
.map(normalize_word)  
.filter(lambda word: len(word) > 5)  
.groupByKey(lambda word: word)  
.mapValues(len)
```

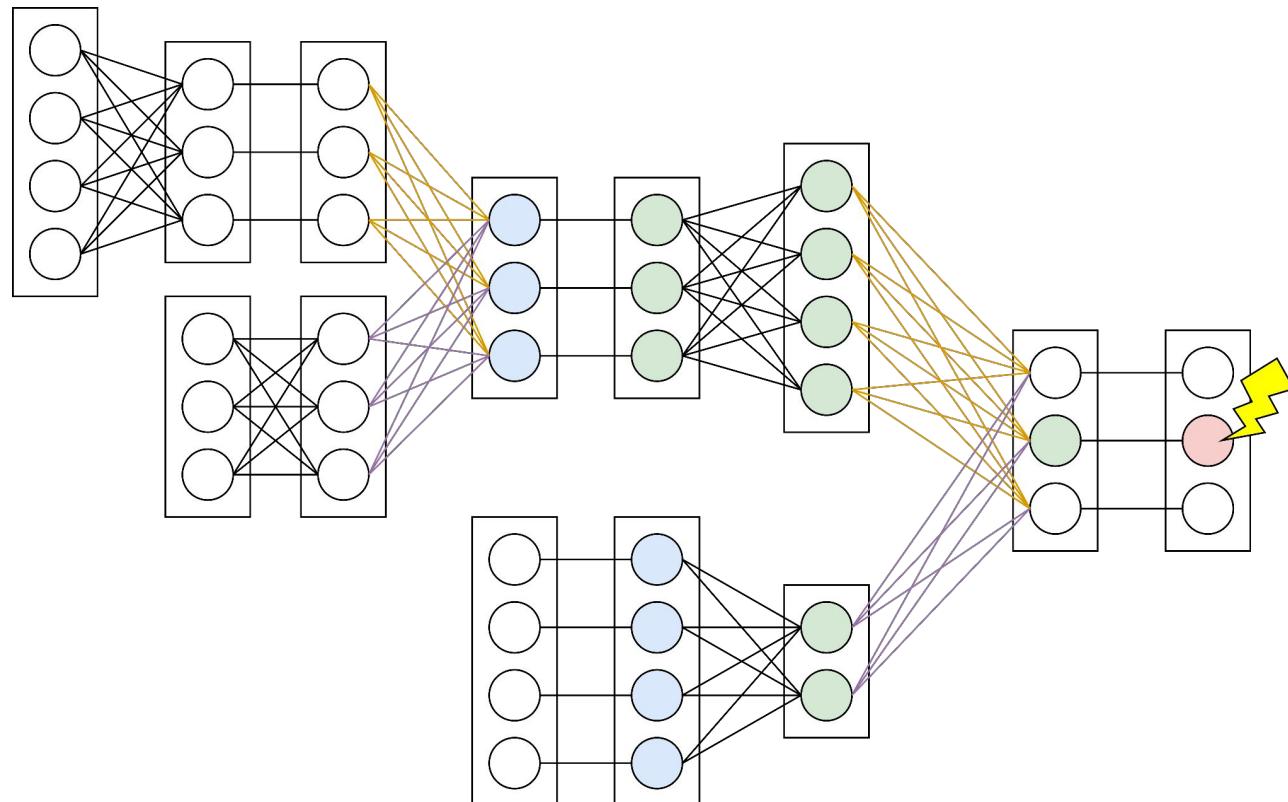
# Высокоуровневые языки обработки данных

- Строим граф задач
- Каждая задача — построение датасета из нескольких других
- В каждом датасете произвольное число партиций
- Есть one-to-one и many-to-many операции



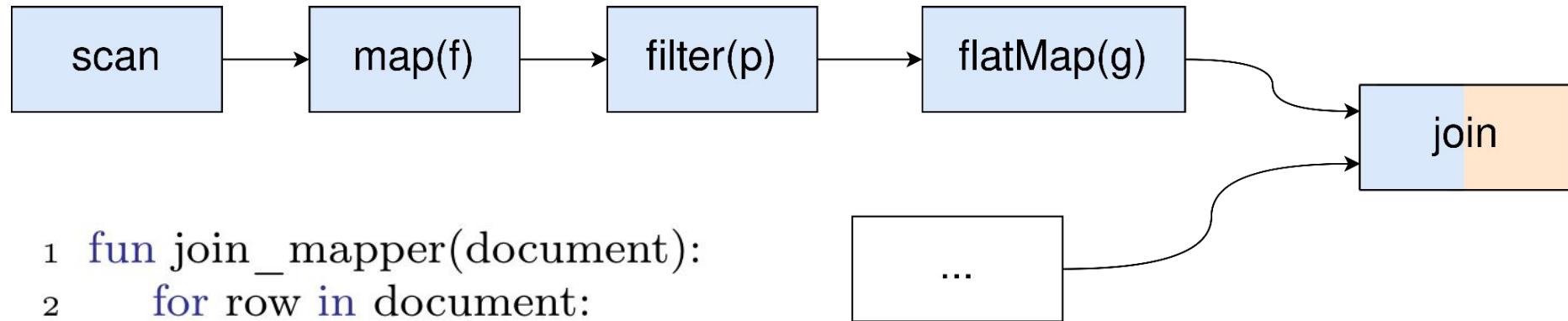
# Высокоуровневые языки обработки данных

- При сбое пересчитываем потерянную партицию
- И зависимости,  
если нужно
- В том числе  
транзитивные
- Кэшированные  
партиции не  
пересчитываем



# Высокоуровневые языки обработки данных

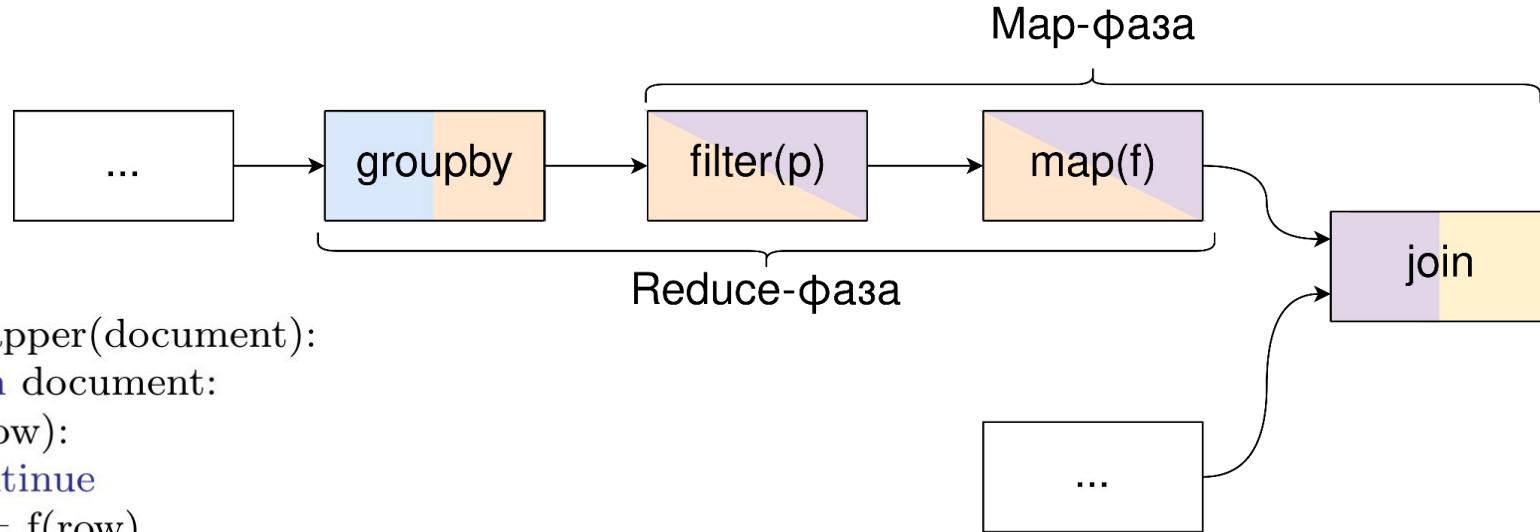
- Все операции, предшествующие первой сложной операции, уходят в тар-фазу этой операции



```
1 fun join_mapper(document):  
2     for row in document:  
3         mapped := f(row)  
4         if p(mapped):  
5             for (a, b) in g(mapped):  
6                 yield b, (a, Left)
```

# Высокоуровневые языки обработки данных

- Операции между двумя сложными операциями поместить либо в reduce-фазу первой, либо в map-фазу второй



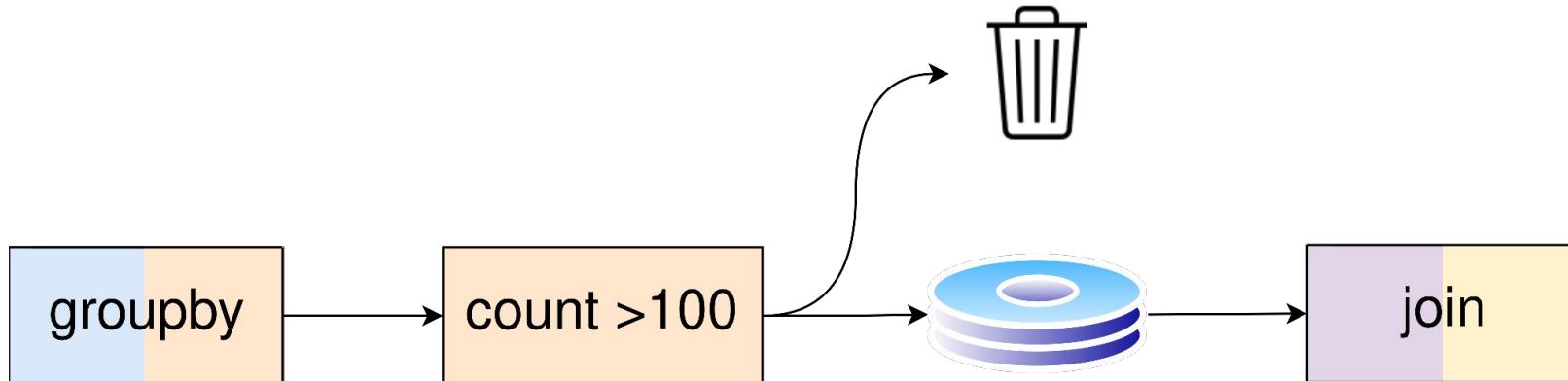
```
1 fun join_mapper(document):  
2     for row in document:  
3         if !p(row):  
4             continue  
5         a, b := f(row)  
6         yield b, (a, Left)
```

```
1 fun groupby_reducer(key, values):  
2     if !p(key, values):  
3         return  
4     yield f(key, values)
```

# Высокоуровневые языки обработки данных

- Лучше выполнить в редьюсере, если удастся выкинуть часть данных
- Не записывая их на диск
- Актуально для компиляции в Hadoop MapReduce

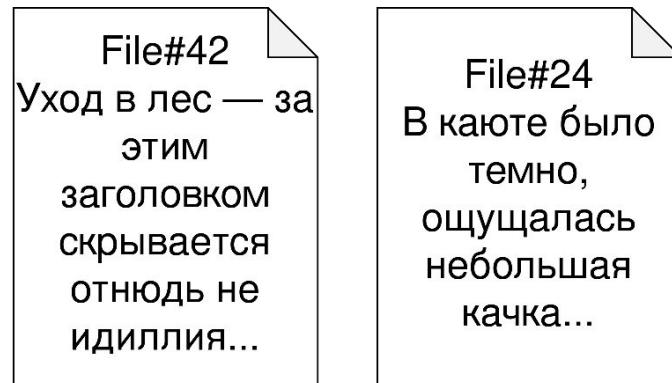
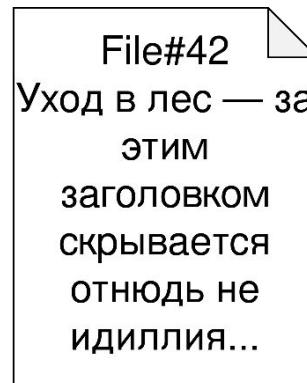
```
SELECT Users.*  
FROM Users INNER JOIN (  
    SELECT Events.user_id  
    FROM Events  
    GROUP BY Events.user_id  
    HAVING count(*) > 100  
) ON Users.id = Events.user_id
```



# Высокоуровневые языки обработки данных

- Имеется корпус текстов
- Каждый файл разбит на части и сжат
- Нужно объединить части, разжать файл
- Посчитать, в скольких файлах встречается каждое слово

file_id	data
42	0xAB...
24	0xFF...
42	0x0A...
24	0xEF...



word	count
общество	2
Луций	1
...	...

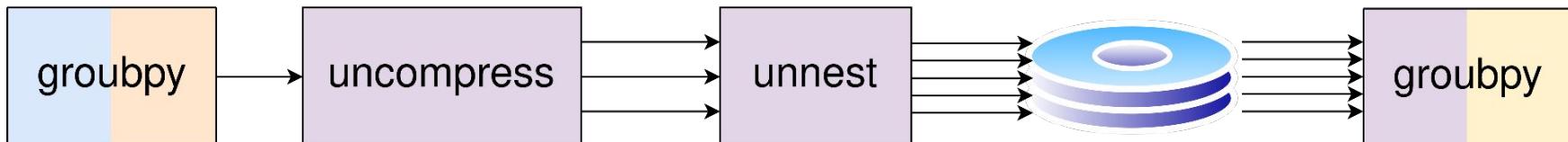
# Высокоуровневые языки обработки данных

- Имеется корпус текстов
- Каждый файл разбит на части и сжат
- Нужно объединить части, разжать файл
- Посчитать, в скольких файлах встречается каждое слово

```
WITH Files AS (
    SELECT FileParts.file_id,
           merge_uncompress(FileParts.data) AS text
    FROM Files
   GROUP BY FileParts.file_id
)
SELECT Words.word, count(DISTINCT Words.file_id)
  FROM (
    SELECT Files.file_id,
           unnest(words(Files.text)) AS word
      FROM Files
)
 AS Words
 GROUP BY Words.word;
```

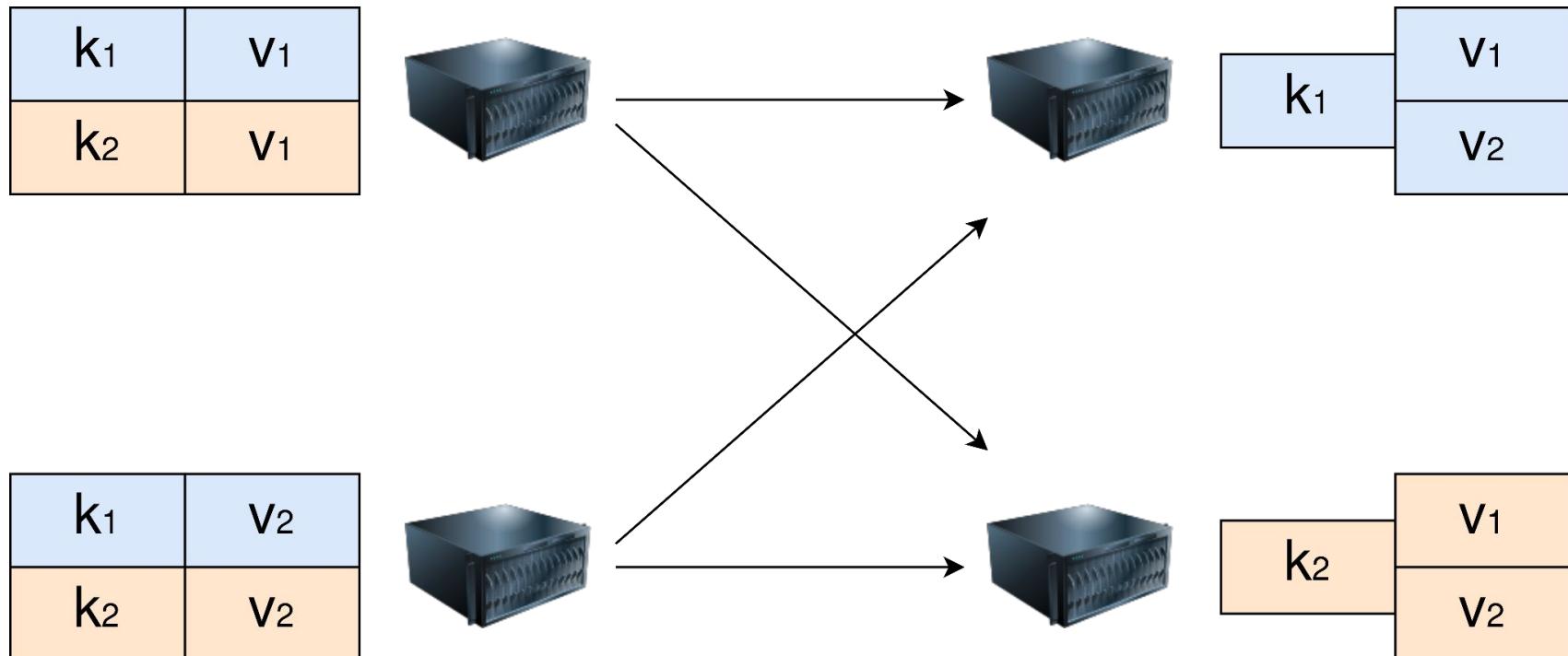
# Высокоуровневые языки обработки данных

- `uncompress` и `unnest` не уменьшают размера данных, как фильтрация
- А увеличивают
- Не хотим записывать их результаты на диск



# Коллокация данных

- В общем случае для GROUP BY и JOIN необходимо провести решардинг

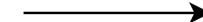


# Коллокация данных

- Можно сделать локально на каждой машине, если все значения для одного ключа *коллоцированы*
- Аналогично с JOIN
- Соединяемые таблицы должны быть шардированы по ключу соединения



$k_1$	$v_1$
$k_1$	$v_2$



$k_1$	$v_1$
$k_1$	$v_2$



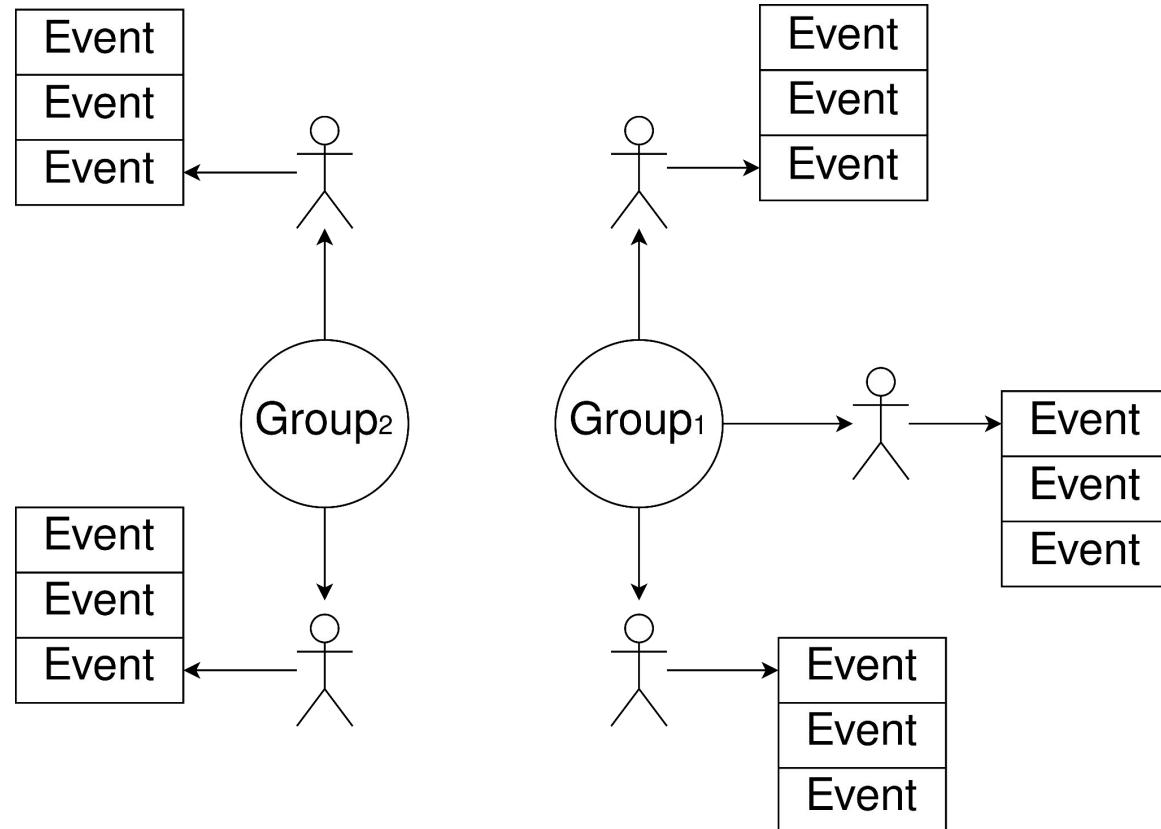
$k_2$	$v_1$
$k_2$	$v_2$



$k_2$	$v_1$
$k_2$	$v_2$

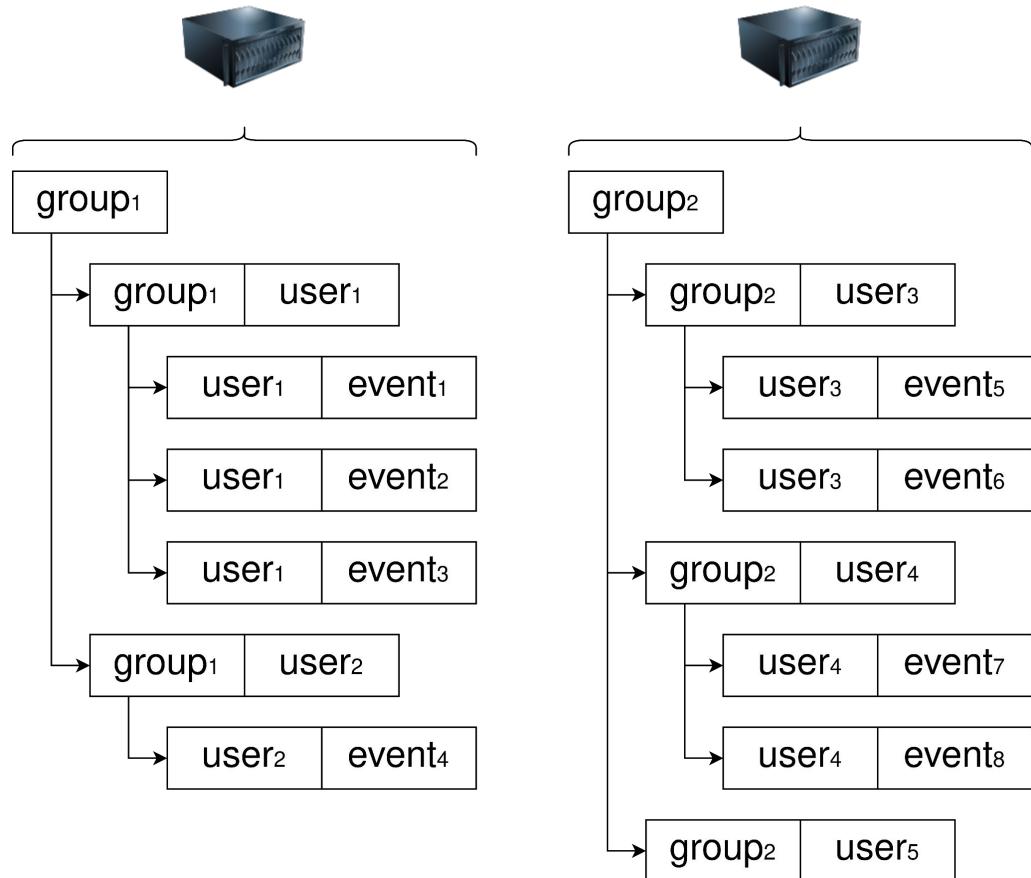
# Коллокация данных: иерархическая схема

- Существует множество групп
- В каждой группе есть множество пользователей
- Каждый пользователь находится строго в одной группе
- У каждого пользователя есть множество событий



# Коллокация данных: иерархическая схема

- Группа вместе со всеми своими подчинёнными записями хранится на одном шарде



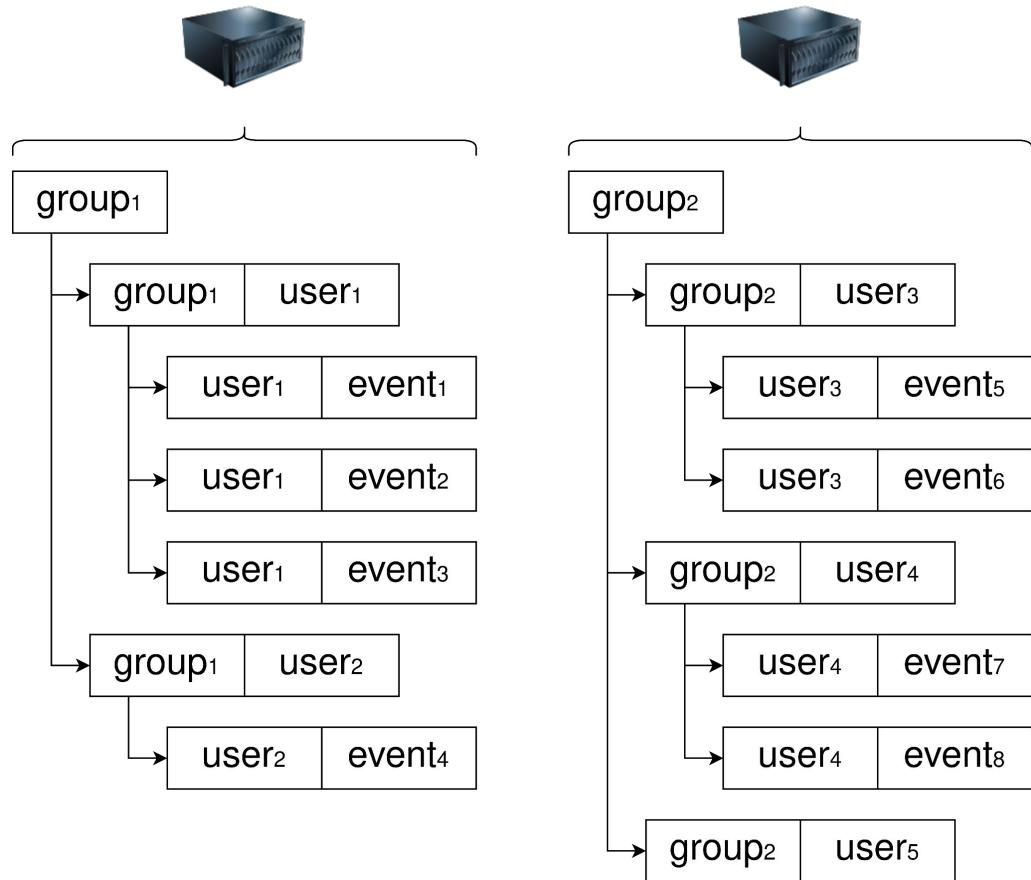
# Коллокация данных: иерархическая схема

- Ускоряем запросы  
вида

```
SELECT Groups.*, Users.*  
FROM Groups  
    INNER JOIN Users  
        ON Groups.id = Users.group_id;
```

```
SELECT Users.*, Events.*  
FROM Users  
    INNER JOIN Events  
        ON Users.id = Events.user_id;
```

```
SELECT Groups.*, Users.*, Events.*  
FROM Groups  
    INNER JOIN Users  
        ON Groups.id = Users.group_id  
    INNER JOIN Events  
        ON Users.id = Events.user_id;
```



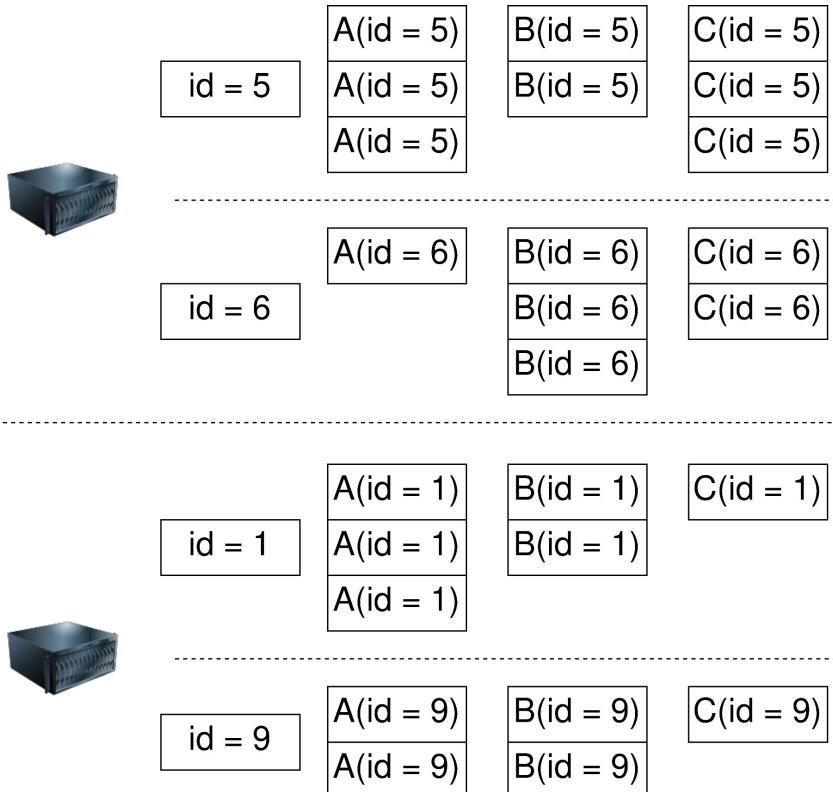
# Коллокация данных

- После одной операции данные могут быть коллоцированы таким образом, что вторая операция может быть исполнена локально
  - После соединения данные будут шардированы по терму
    - Группировка по терму происходит локально
- ```
SELECT TF_IDF.term, avg(TF_IDF.tf_idf)
FROM (
    SELECT TF.term, TF.doc,
           TF.tf * IDF.idf AS tf_idf
    FROM TF INNER JOIN IDF
    ON TF.term = IDF.term
) AS TF_IDF
GROUP BY TF_IDF.term
```

# Соединения трёх и более таблиц

- Если происходят по одному и тому же ключу, могут быть исполнены за одну операцию

```
SELECT A.*, B.*, C.*  
FROM A INNER JOIN B ON A.id = B.id  
      INNER JOIN C ON A.id = C.id;
```



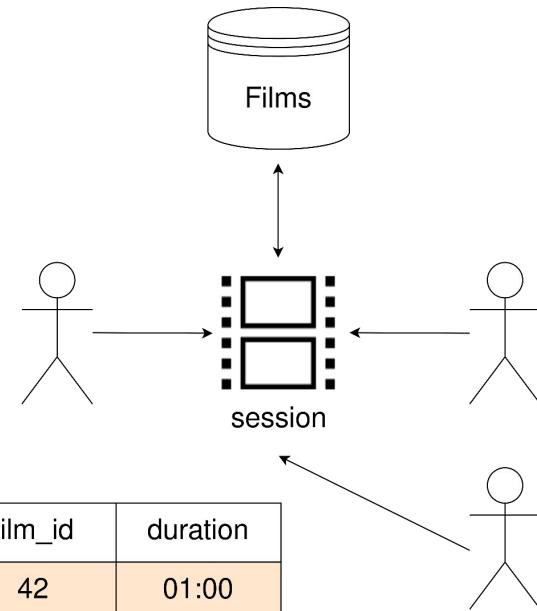
- После распределения по редьюсерам каждый элемент в группе находится в тройке с каждым другим

# Прочие оптимизации

- Участники сессии получают одинаковый видеопоток
- Для каждой сессии известна продолжительность каждого просмотренного фильма
- И список участников

| user_id | session_id |
|---------|------------|
| 1       | 0xFF...    |
| 2       | 0xFF...    |
| 3       | 0x00...    |
| 1       | 0x00...    |
| 4       | 0xAA...    |

| session_id | film_id | duration |
|------------|---------|----------|
| 0xFF...    | 42      | 01:00    |
| 0xAA...    | 26      | 01:00    |
| 0xAA...    | 24      | 02:00    |
| 0x00...    | 14      | 00:30    |
| 0xFF...    | 24      | 02:00    |
| 0xAA...    | 25      | 01:30    |



## Прочие оптимизации

- Считаем общую длительность просмотра для каждого участника

```
SELECT UserFilms.user_id, sum(UserFilms.duration)
FROM (
    SELECT Users.user_id, Users.session_id,
           Films.film_id, Films.duration
    FROM Users INNER JOIN Films
        ON Users.session_id = Films.session_id
) AS UserFilms
GROUP BY UserFilms.user_id;
```

# Прочие оптимизации

- Считаем общую длительность просмотра для каждого участника
- Приходится материализовать очень большую таблицу
- Размера потенциально  $|Users| \times |Films|$

| user_id | session_id |
|---------|------------|
| 1       | 0xFF...    |
| 2       | 0xFF...    |
| 3       | 0x00...    |
| 1       | 0x00...    |
| 4       | 0xAA...    |



| session_id | film_id | duration |
|------------|---------|----------|
| 0xFF...    | 42      | 01:00    |
| 0xAA...    | 26      | 01:00    |
| 0xAA...    | 24      | 02:00    |
| 0x00...    | 14      | 00:30    |
| 0xFF...    | 24      | 02:00    |
| 0xAA...    | 25      | 01:30    |



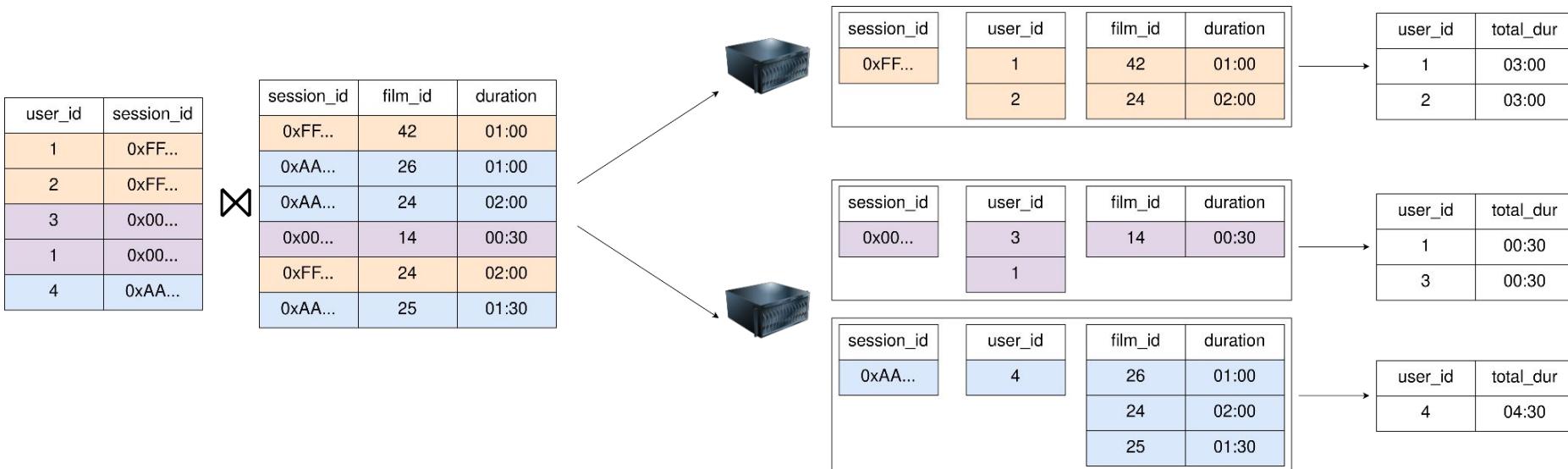
| user_id | session_id | film_id | duration |
|---------|------------|---------|----------|
| 1       | 0xFF...    | 42      | 01:00    |
| 1       | 0xFF...    | 24      | 02:00    |
| 2       | 0xFF...    | 42      | 01:00    |
| 2       | 0xFF...    | 24      | 02:00    |
| 3       | 0x00...    | 14      | 00:30    |
| 1       | 0x00...    | 14      | 00:30    |
| 4       | 0xAA...    | 26      | 01:00    |
| 4       | 0xAA...    | 24      | 02:00    |
| 4       | 0xAA...    | 25      | 01:30    |



| user_id | total_dur |
|---------|-----------|
| 1       | 03:30     |
| 2       | 03:00     |
| 3       | 00:30     |
| 4       | 04:30     |

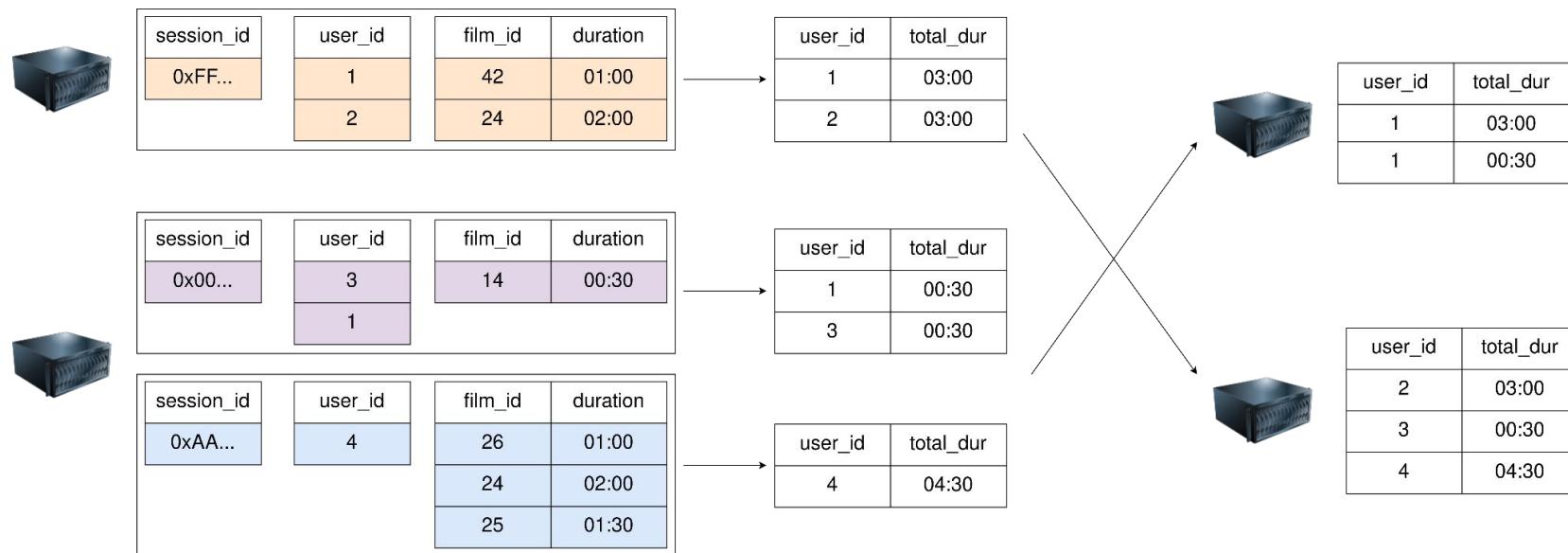
# Прочие оптимизации

- Избегаем материализации промежуточной таблицы
- В ходе первой операции подсчитываем суммарное время просмотра для каждого пользователя в каждой сессии
- Размер не превосходит  $|Users|$



# Прочие оптимизации

- Избегаем материализации промежуточной таблице
- В ходе второй операции проводим repartition по user\_id
- Подсчитываем финальную агрегацию
- Используем combiner



# Высокоуровневые языки обработки данных

- Целый зоопарк
- Буквально

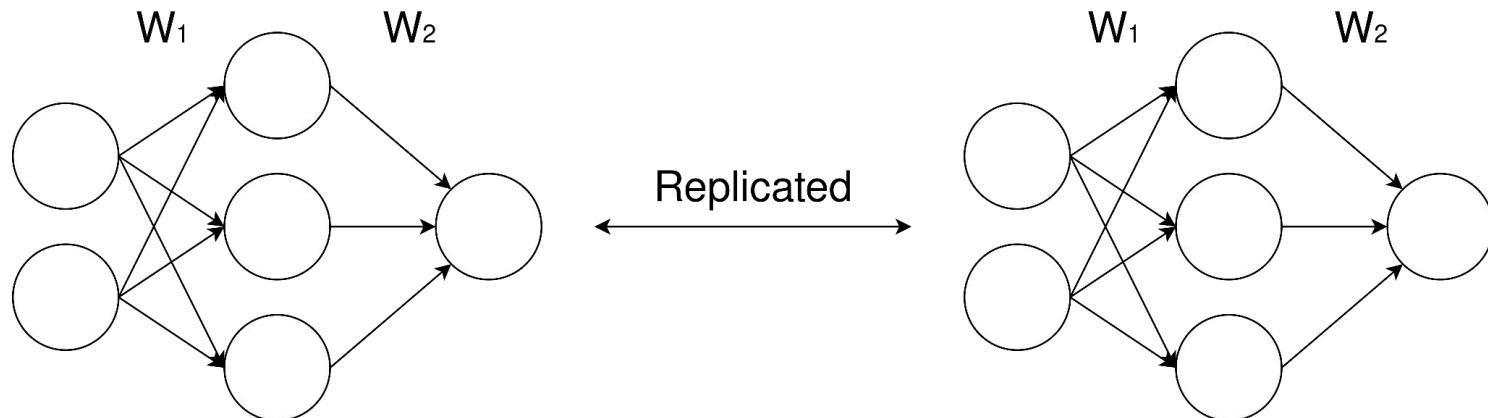


Apache Pig



# Распределённое машинное обучение

- Датасет шардируем
- Веса модели реплицируем



# Разбиение градиента

- Разбиваем градиент по шардированному датасету
- $D_i$  — разбиение  $D$

$$\bigcup_{i=1}^S D_i = D ; \forall i \neq j : D_i \cap D_j = \emptyset$$

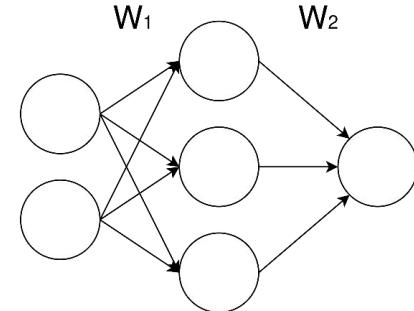
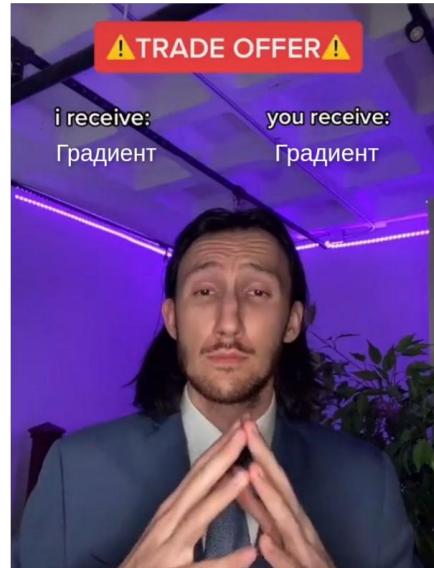
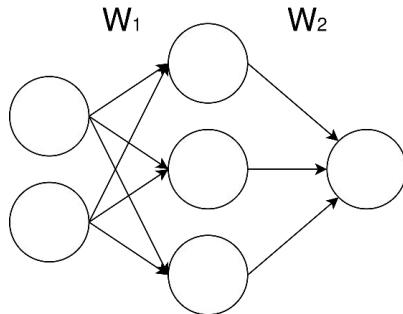
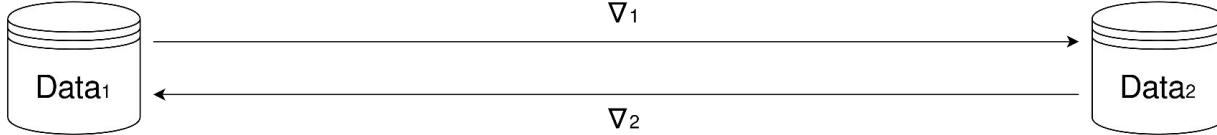
$$\mathbb{L}_{D_i} = \sum_{x,y \in D_i} L(x, y, \hat{y})$$

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

$$\frac{\partial \mathbb{L}}{\partial W} = \frac{\sum_i \mathbb{L}_{D_i}}{\partial W} = \sum_i \frac{\partial \mathbb{L}_{D_i}}{\partial W} = \sum_i \nabla_i$$

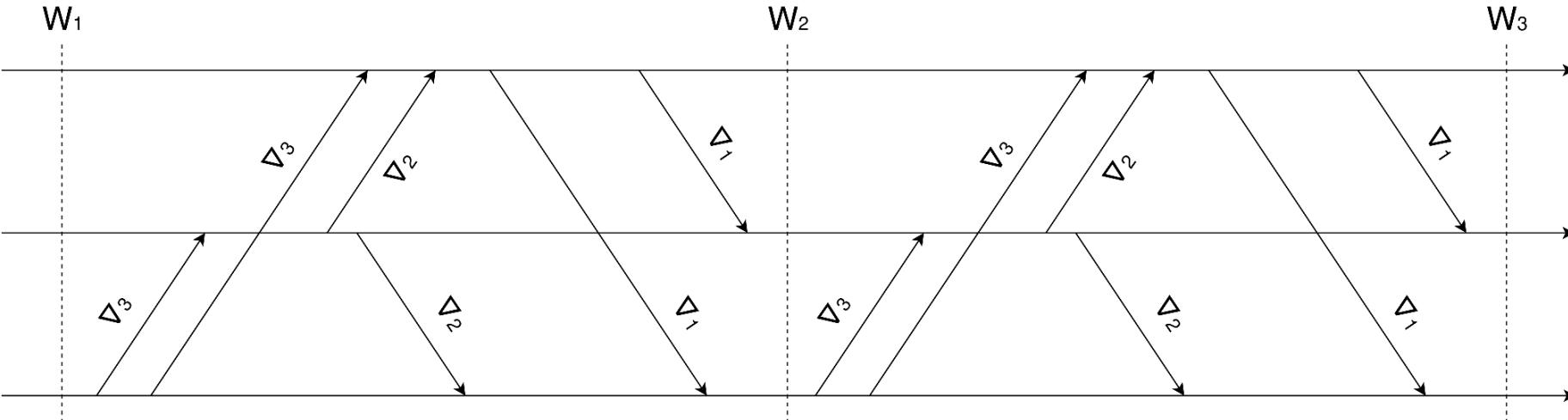
# Распределённое машинное обучение

- Сервера обмениваются локальными градиентами
- Считают общий градиент



# Распределённое машинное обучение

- Каждый участник рассыпает свой градиент всем остальным
- Дожидается градиентов от всех остальных
- $W \leftarrow W - \eta (\nabla_1 + \nabla_2 + \dots + \nabla_n)$
- После этого веса снова одинаковые



# Распределённый ML: масштабирование

- Что происходит при увеличении числа процессов?
- Всего система рассыпает  $4 \cdot |P| \cdot (|P| - 1) \cdot |W|$  байт
- Каждый процесс тратит  $\frac{|D|}{|P|} \cdot |W|$  времени на вычисления
  - Размер датасета уменьшается пропорционально числу процессов
  - Не точная оценка, оцениваем только порядок
- И  $|P| \cdot |W|$  времени на пересылку градиента
  - Время пересылки начинает доминировать в алгоритме

# Распределённый ML: 1-bit quantization

- Заменяем положительные числа на среднее среди положительных
- Отрицательные — на среднее среди отрицательных
- Пересылаем только знак числа и два средних
- Позволяет приблизительно передать направление движения

|   |    |   |   |    |
|---|----|---|---|----|
| 5 | -1 | 1 | 3 | -3 |
|---|----|---|---|----|

$$\begin{array}{l} E_+ = 3 \\ E_- = -2 \end{array}$$

|   |    |   |   |   |   |   |
|---|----|---|---|---|---|---|
| 3 | -2 | + | - | + | + | - |
|---|----|---|---|---|---|---|

|   |    |   |   |    |
|---|----|---|---|----|
| 3 | -2 | 3 | 3 | -2 |
|---|----|---|---|----|

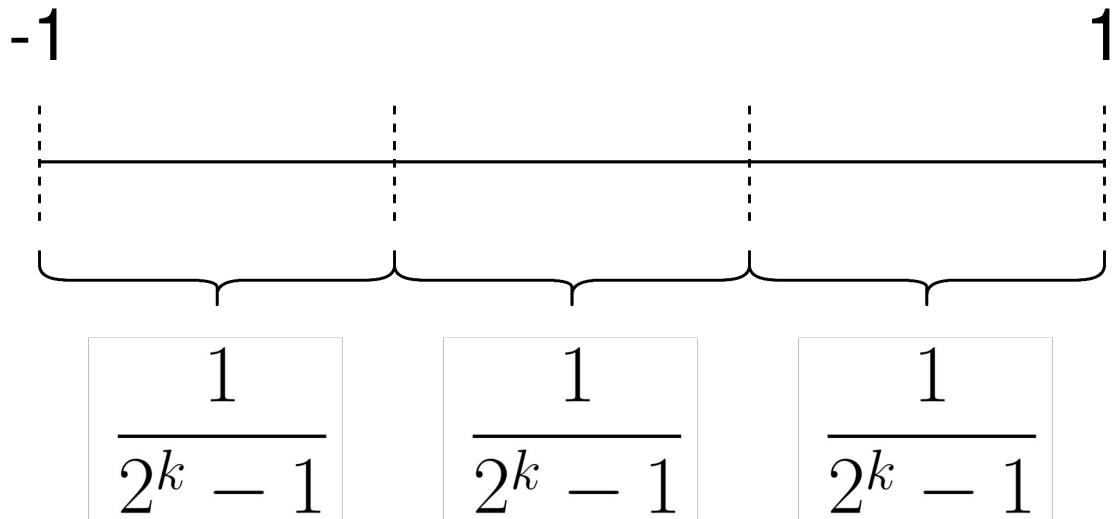
# Распределённый ML: Stochastic quantization

- Будем кодировать нормированный градиент
- Константу нормировки будем пересылать, чтобы на принимающей стороне восстановить исходный градиент

$$\hat{W}_i = \frac{W_i}{\sqrt{\sum_{j=1}^{|W|} W_j^2}} = \frac{W_i}{\sqrt{W_i^2 + \sum_{1 \leq j \leq |W|, j \neq i} W_j^2}} \in [-1, 1]$$

# Распределённый ML: Stochastic quantization

- Выберем количество бит  $k$ , которым будем кодировать каждую компоненту градиента
- На отрезке  $[-1; 1]$  расположим равномерно  $2^k$  чисел
  - Будем ими кодировать компоненты градиента
- Пересылаем  $k|W| + 32$  бита

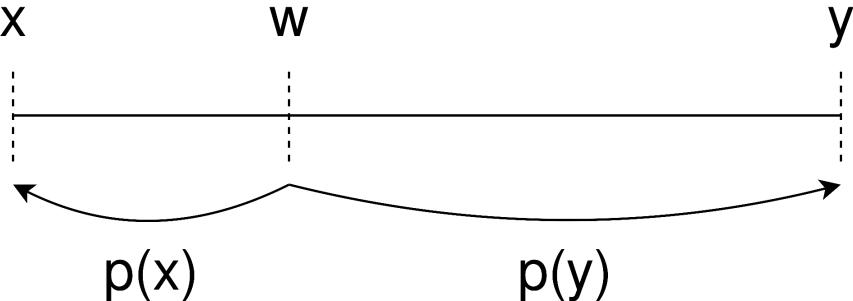


# Распределённый ML: Stochastic quantization

- Для каждой компоненты градиента  $w$  ищем ближайшие представимые числа справа и слева
- Заменим  $w$  на одно из них

$$\mathbb{P}(y) = \frac{w - x}{y - x}$$

$$\mathbb{P}(x) = \frac{y - w}{y - x}$$



- Несмешённое преобразование

$$\mathbb{P}(x) + \mathbb{P}(y) = \frac{y - w}{y - x} + \frac{w - x}{y - x} = 1$$

$$\mathbb{E}(w) = x \cdot \mathbb{P}(x) + y \cdot \mathbb{P}(y) = x \cdot \frac{y - w}{y - x} + y \cdot \frac{w - x}{y - x} = \frac{yx - wx +yw - yx}{y - x} = w$$

# Распределённый ML: Sparsification

- Оставляем только  $K$  самых больших по модулю компонентов
  - $K = 3$
  - $\nabla = [-2, 3, 10, 1, -100, 15, -1]$
  - $\nabla' = [0, 0, 10, 0, -100, 15, 0]$
  - $\text{serialize}(\nabla') = [2 : 10, 4 : -100, 5 : 15]$
- Пересылаем только ненулевые компоненты
  - Пересылаем  $32 * K + K * \log |W|$  бит

# Распределённый ML: Error correction

- Учитываем на  $(i + 1)$ -ом шаге ошибку с  $i$ -ого шага
- Ошибку определяем как разность между истинным градиентом и полученным в результате преобразования
- Каждый процесс делает независимо **только для локального градиента**
- После коррекции преобразует градиент

$$\nabla_i = \left( \nabla_i^1, \nabla_i^2, \dots, \nabla_i^{|W|} \right)$$

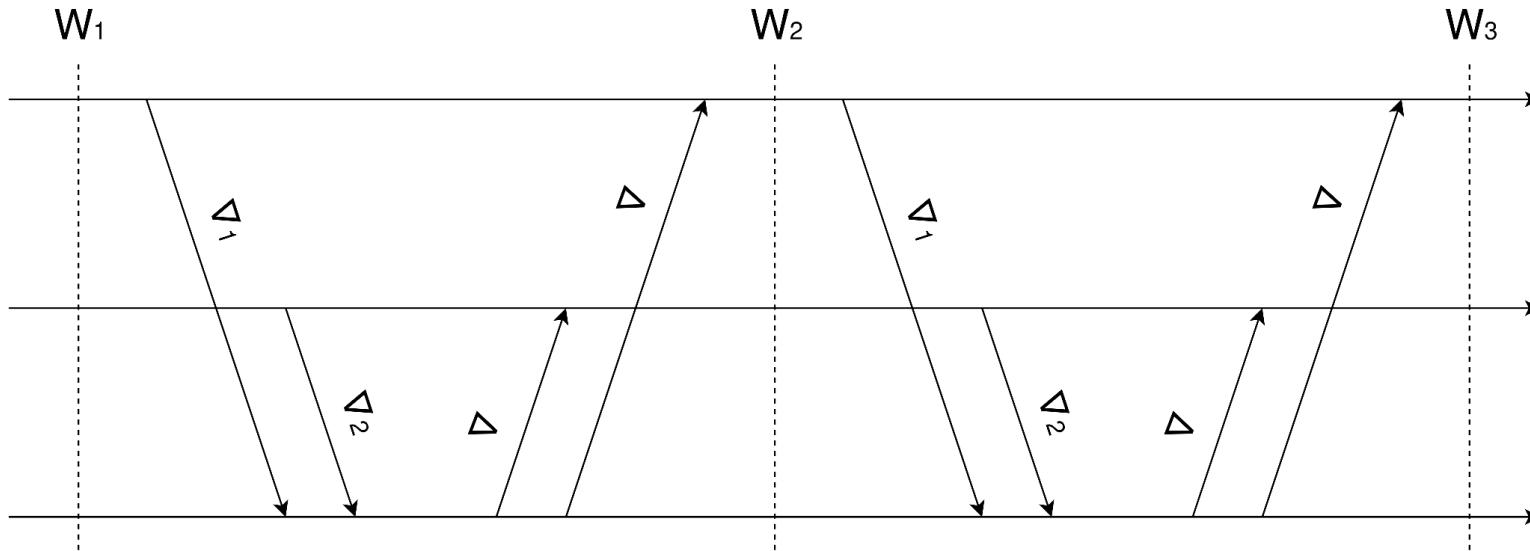
$$\widetilde{\nabla}_i = \left( \widetilde{\nabla}_i^1, \widetilde{\nabla}_i^2, \dots, \widetilde{\nabla}_i^{|W|} \right)$$

$$e_i = \nabla_i - \widetilde{\nabla}_i = \left( \nabla_i^1 - \widetilde{\nabla}_i^1, \nabla_i^2 - \widetilde{\nabla}_i^2, \dots, \nabla_i^{|W|} - \widetilde{\nabla}_i^{|W|} \right)$$

$$corr(\nabla_{i+1}) = \nabla_i + e_i$$

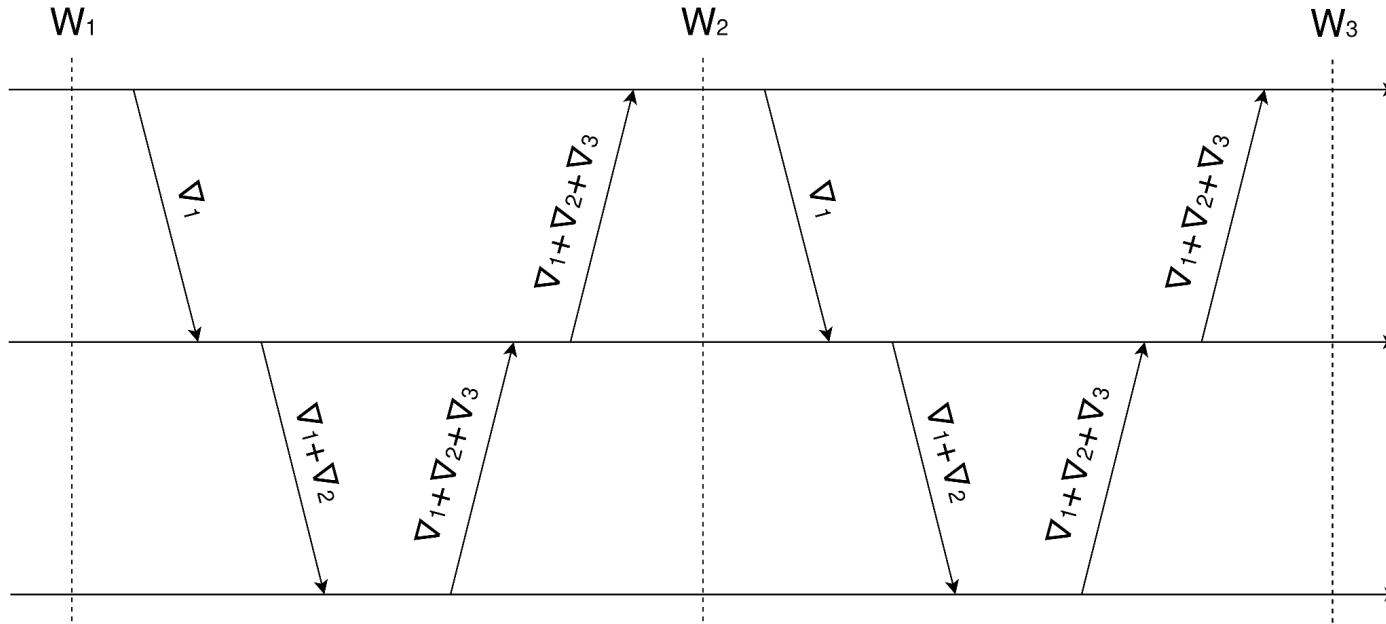
# Распределённый ML: пересылка через мастера

- Посыпаем градиенты мастеру, мастер считает сумму и возвращает её всем процессам
- $O(|W| * |P|)$  байт пересылаем
- Совместимо** со сжатием



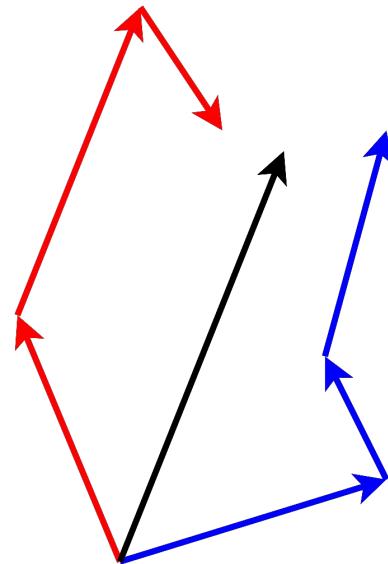
# Распределённый ML: пересылка по кругу

- Децентрализованное решение с равномерным распределением нагрузки
- Время ожидания полного градиента больше 1 RTT



# Распределённый ML: усреднение весов

- Каждый процесс независимо учит модель на своих данных
- Раз в K ходов усредняем веса
- Совместимо со схемами пересылки
- **Не совместимо со сжатием**



- Траектория параметров модели на узле 2
- Траектория параметров модели на узле 1
- Траектория параметров модели после усреднения

## Что почитать: распределённые вычисления

- *Bu Y. et al.* HaLoop: Efficient iterative data processing on large clusters
- *Ekanayake J. et al.* Twister: a runtime for iterative mapreduce
- *Malewicz G. et al.* Pregel: a system for large-scale graph processing
- *Zaharia M. et al.* Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing
- *Olston C. et al.* Pig latin: a not-so-foreign language for data processing
- *Zaharia M. et al.* Spark: Cluster computing with working sets
- *Anand R., Jeffrey David U.* Mining of massive datasets
  - [youtube](#)

# Что почитать: распределённый SQL

- [Станислав Лукьянов. Почему распределенный SQL сложнее, чем кажется](#)
- *Armbrust M. et al.* Relational data processing in Spark
- *Thusoo A. et al.* Hive: a petabyte scale data warehouse
- *Thusoo A. et al.* Hive: a warehousing solution over a map-reduce framework
- *Bittorf M. et al.* Impala: A modern SQL engine for hadoop
- *Shute J. et al.* F1-the fault-tolerant distributed rdbms supporting google's ad business
- *Shute J. et al.* F1: A distributed SQL database that scales
- *Samwel B. et al.* F1 query: Declarative querying at scale

## Что почитать & посмотреть: распределённый ML

- *Alistarh D. et al.* QSGD: Communication-efficient SGD via gradient quantization and encoding
- *Alistarh D. et al.* Communication-efficient stochastic gradient descent, with applications to neural networks
- *Shi S. et al.* Understanding top-k sparsification in distributed deep learning
- *Shi S. et al.* A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks
- *Dan Alistarh. Distributed and concurrent optimization for machine learning*
- [github.com/horovod/horovod](https://github.com/horovod/horovod)

# Thanks for your attention

