

# MML minor #8

Примеры работы с большими данными

# Онлайн-обучение

Онлайн-обучение – данные поступают последовательно, улучшаем модель после каждого нового примера

Когда используется:

- Невозможно обучаться на всей выборке
- Вся выборка не помещается в память
- Нужно быстро адаптироваться к новым зависимостям в данных

# На примере линейной регрессии

- Обучение на всей выборке:

$$w^* = (X^T X)^{-1} X^T Y$$

- Можно обновлять веса рекурсивно:

$$w_0 = \mathbf{0} \in \mathbb{R}^d$$

$$\Gamma_0 = I \in \mathbb{R}^{d \times d}$$

$$\begin{aligned}\Gamma_i &= \Gamma_{i-1} - \frac{\Gamma_{i-1} x_i x_i^T \Gamma_{i-1}}{1 + x_i^T \Gamma_{i-1} x_i} \\ w_i &= w_{i-1} - \Gamma_i x_i (x_i^T w_{i-1} - y_i)\end{aligned}$$

- Стохастический градиентный спуск (SGD):

$$w_i = w_{i-1} - \gamma_i x_i (x_i^T w_{i-1} - y_i)$$



Похожи

# Vowpal Wabbit (VW)

- Открытая реализация алгоритмов онлайн-обучения
- Не только линейные модели (парные взаимодействия, простые нейросети)
- Много функций потерь
- Обычная строка текста – это валидное описание объекта. Применяет **хэширование** признаков.
  - 1 | The dog ate my homework
- Эффективно масштабируется на 1000 машин при помощи AllReduce.
- Может работать в режиме с обратной связью (Contextual Bandit)
- Может работать в режиме активного обучения
- ...

# Хэширование признаков

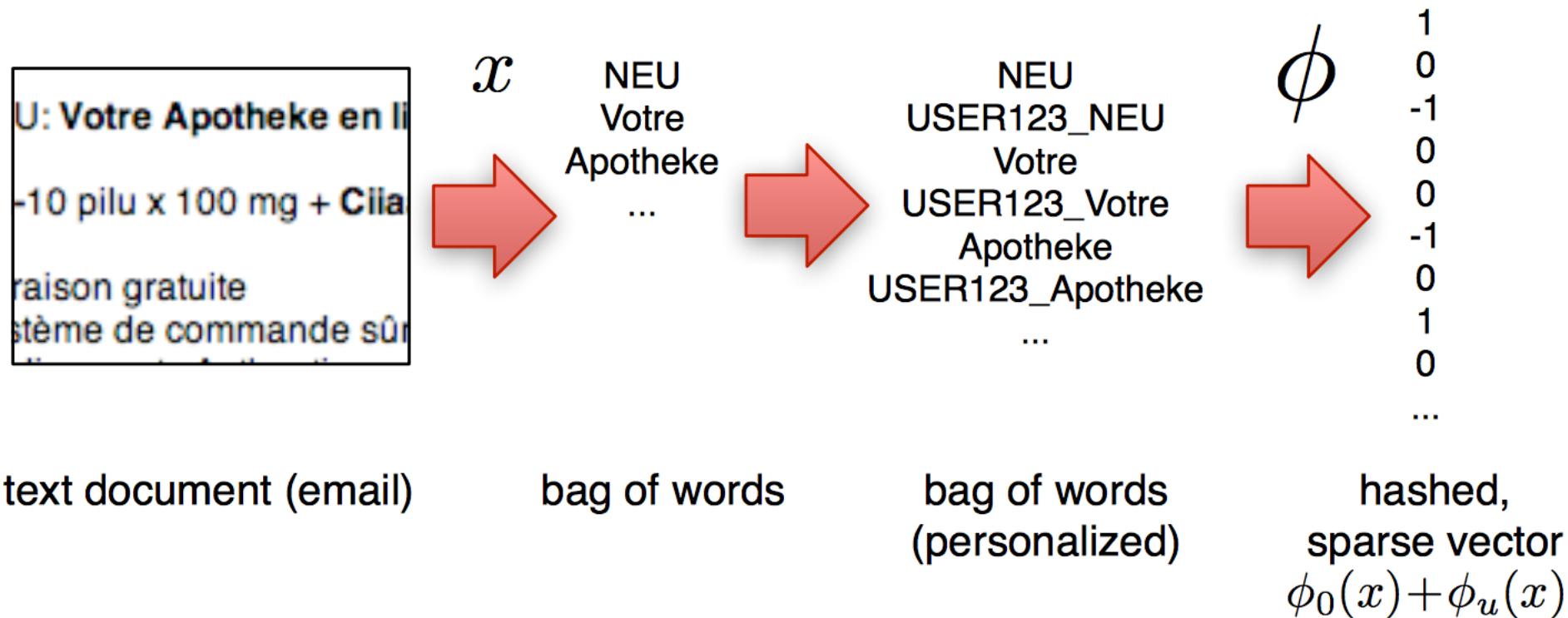
- Когда требуется one-hot кодирование признаков:
  - Категориальные признаки
  - Слова текста в модели мешка слов
- Храним “word” → index
  - Словарь должен быть общим для всех машин
  - Может не поместиться в RAM
- Считаем “word” → hash(“word”)  $\text{hash}(s) = s[0] + s[1]p + s[2]p^2 + \dots + s[n]p^n$ 
  - Большое число корзинок хэш-функции ( $\sim 2^{24}$ ), качество растет по  $\log(\text{hash bits})$
  - Значительно быстрее, не занимает памяти и легко распараллеливается

Пример хэш-функции  
для строк ( $s$  – строка,  $p$  –  
фиксированное простое  
число,  $s[i]$  – код  
символа строки)



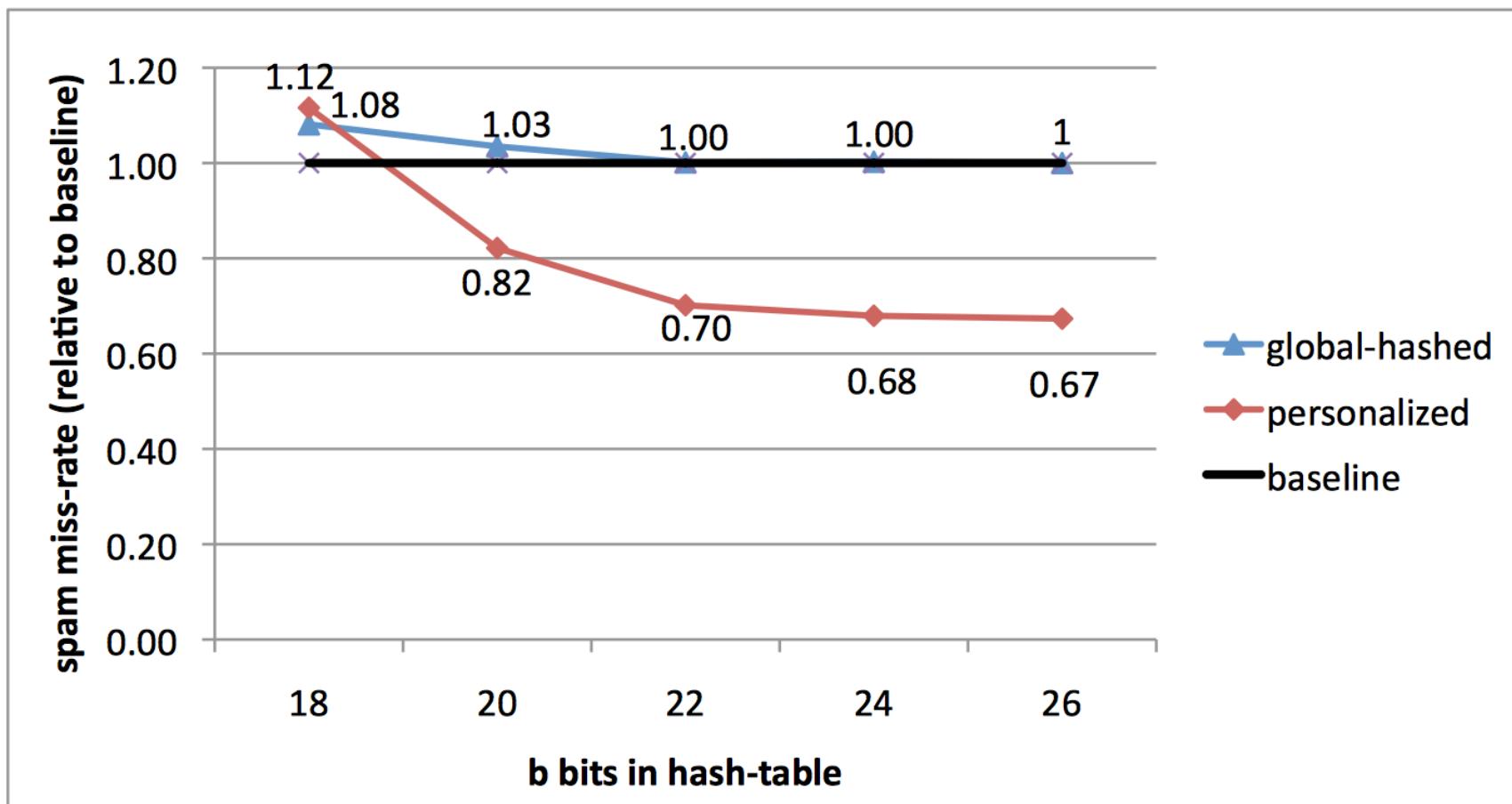
# Хэширование в детекции спама

- 0.4 млн пользователей, 3.2 млн писем, 40 млн слов
- 16 трлн пар (пользователь, слово) – поможет только хэширование



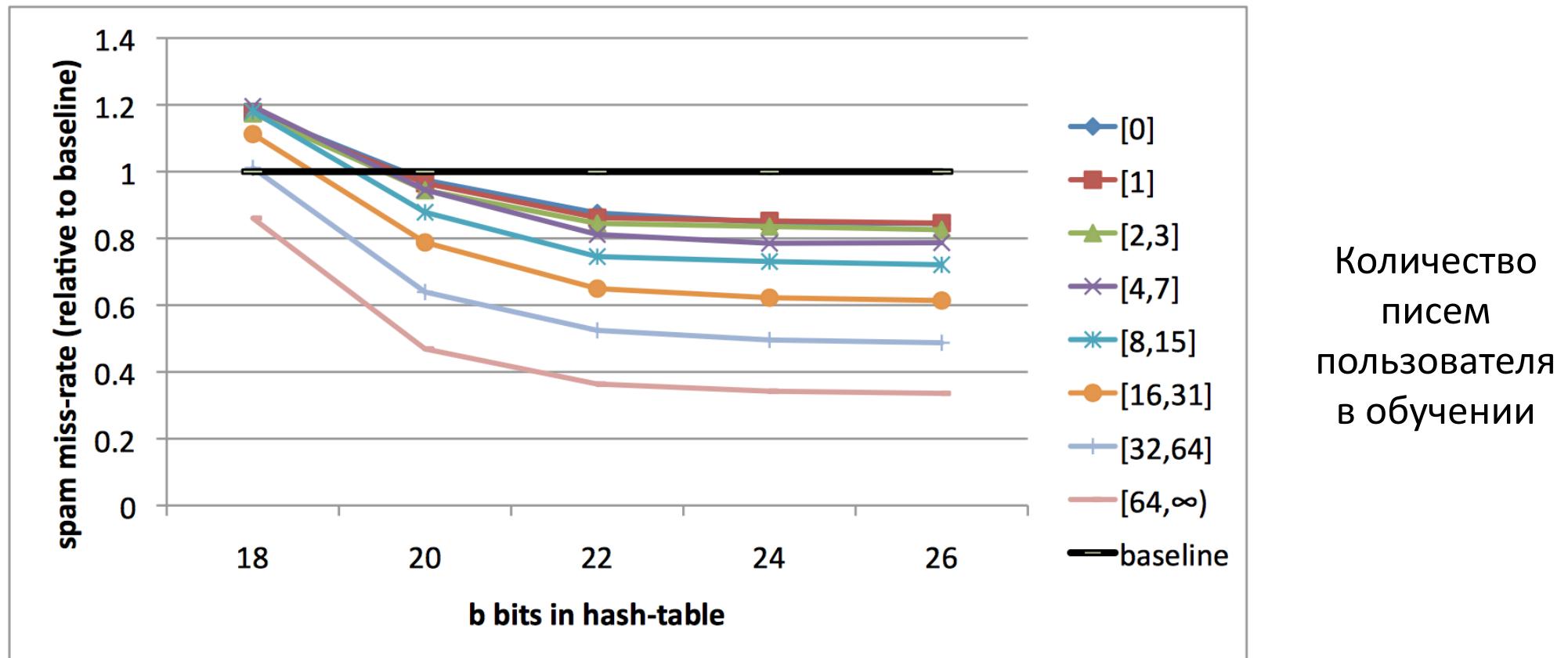
# Хэширование в детекции спама

- Хэшированные **16 трлн признаков** дают существенный прирост качества
- Хэширование перестает влиять на качество **не персональной модели**



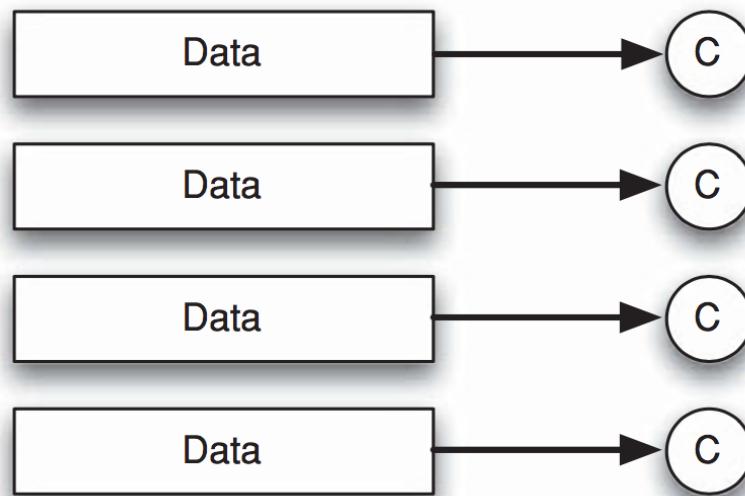
# Хэширование в детекции спама

- Хорошо работает даже на пользователях, которых не было в обучении!
- Гипотеза: все «локальные» зависимости были учтены новыми признаками, «глобальные» зависимости стали более универсальными

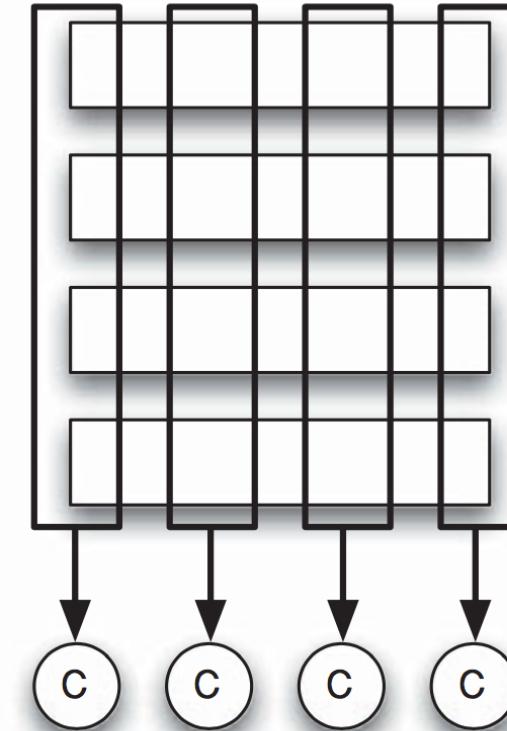


# Как можно распараллелить работу VW

Матрица «объекты-признаки»



Делим объекты

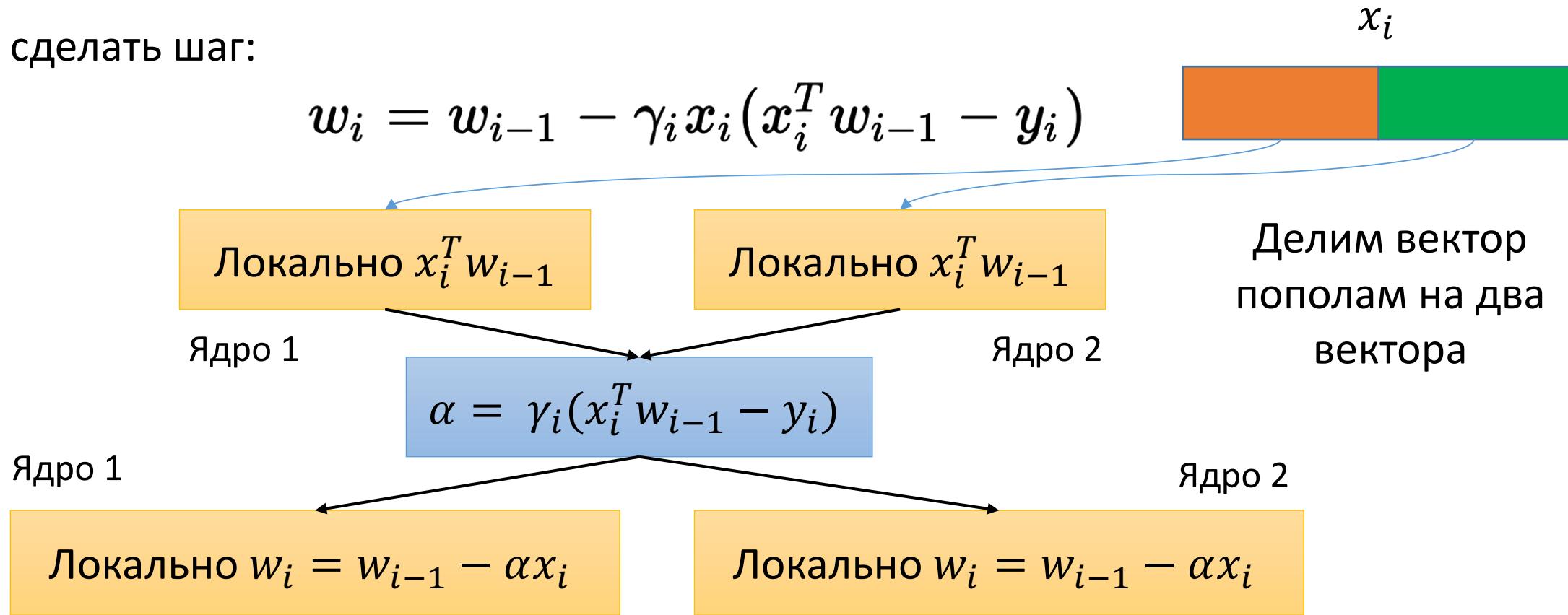


Делим признаки

# Делим признаки на многоядерной машине

- Нужно сделать шаг:

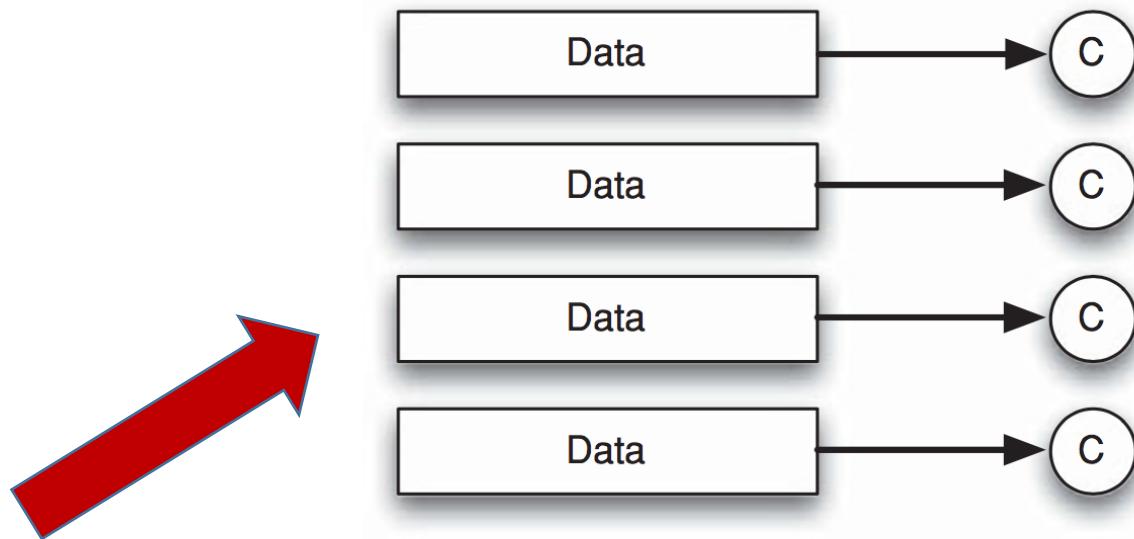
$$w_i = w_{i-1} - \gamma_i x_i (x_i^T w_{i-1} - y_i)$$



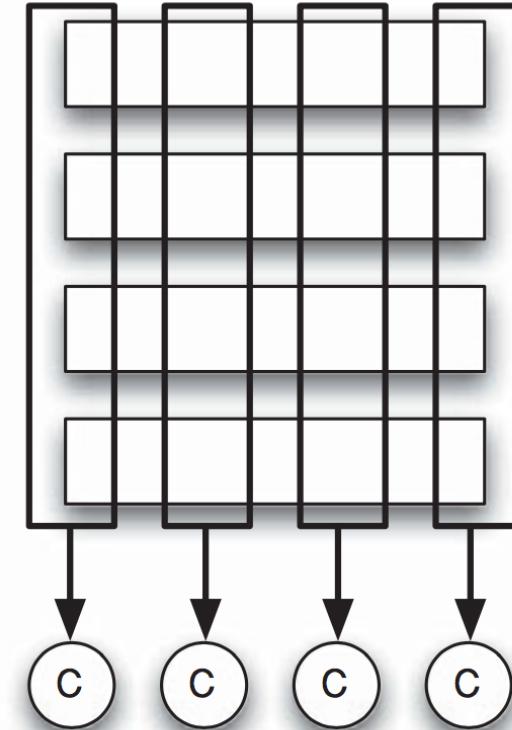
- Результат аналогичен однопоточному варианту
- На 4 ядрах дает ускорение в 3 раза, дальше слабо масштабируется

# Надо делить объекты на много машин!

Матрица «объекты-признаки»



Делим объекты



Делим признаки

# Поможет Hadoop

- **Hadoop** – проект Apache для распределенных вычислений
- **Hadoop YARN** – планировщик задач и система для управления ресурсами кластера
- **Hadoop MapReduce** – система для вычислений в парадигме Map-Reduce
- **HDFS** – распределенная файловая система Hadoop

## Особенности HDFS:

- Файлы хранятся **блоками** на разных машинах
- Каждый **блок дублируется** на нескольких разных машинах (replication factor)
- Информация о соответствии **путь файла → его блоки** хранится на специальной машине **Name Node** в RAM

# Парадигма Map-Reduce на примере Word Count

## Шаг Map:

$(K_1, V_1) \rightarrow \text{List}(K_2, V_2)$

#строки, “Deer Bear”  $\rightarrow [(\text{“Deer”, 1}), (\text{“Bear”, 1})]$

## Шаг Shuffle (или Sort):

Shuffle делит данные по  $\text{hash}(\text{key}) \% N$  на  $N$  частей

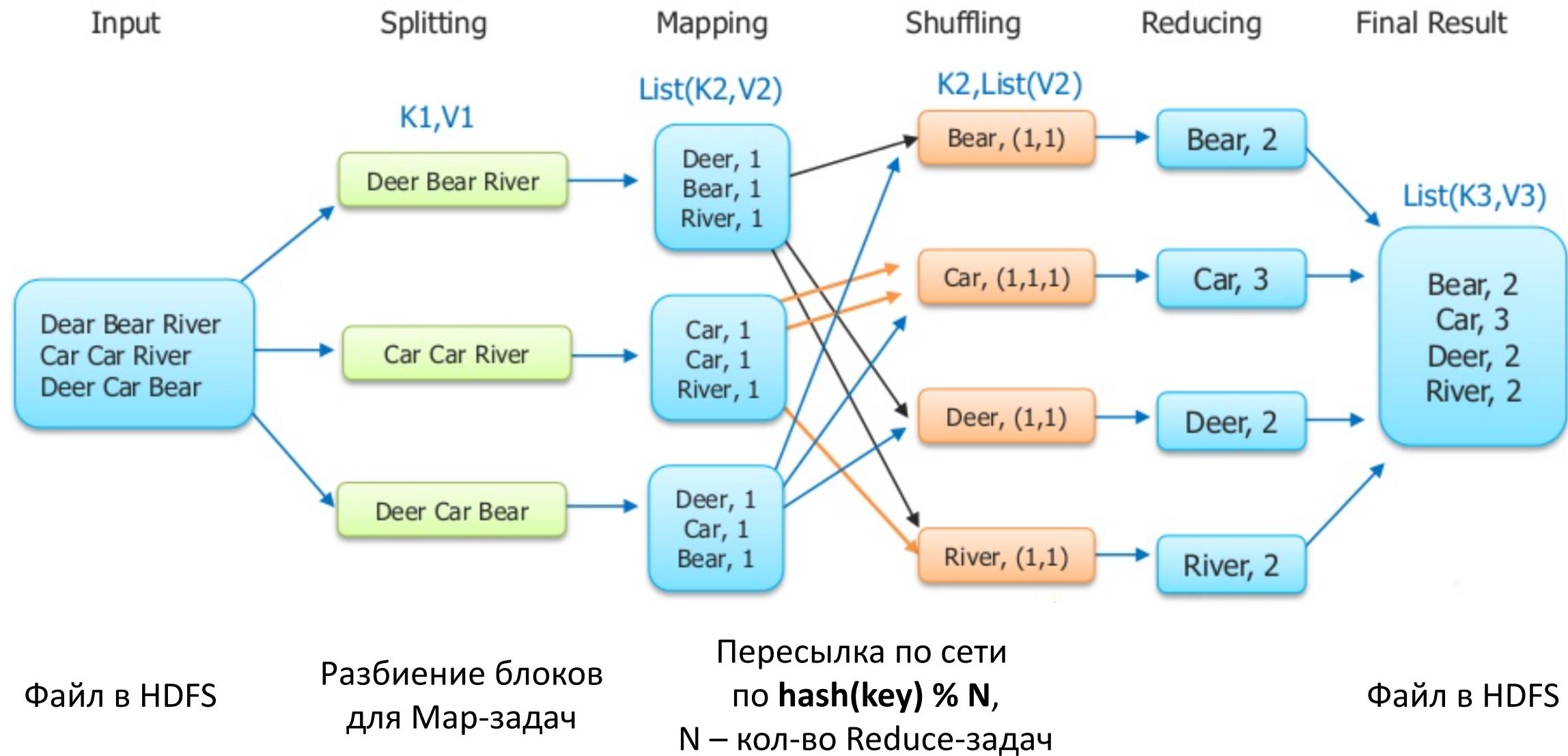
Sort сортирует данные по  $\text{key}$  и делит на  $N$  частей (по границам  $\text{key}$ )

## Шаг Reduce:

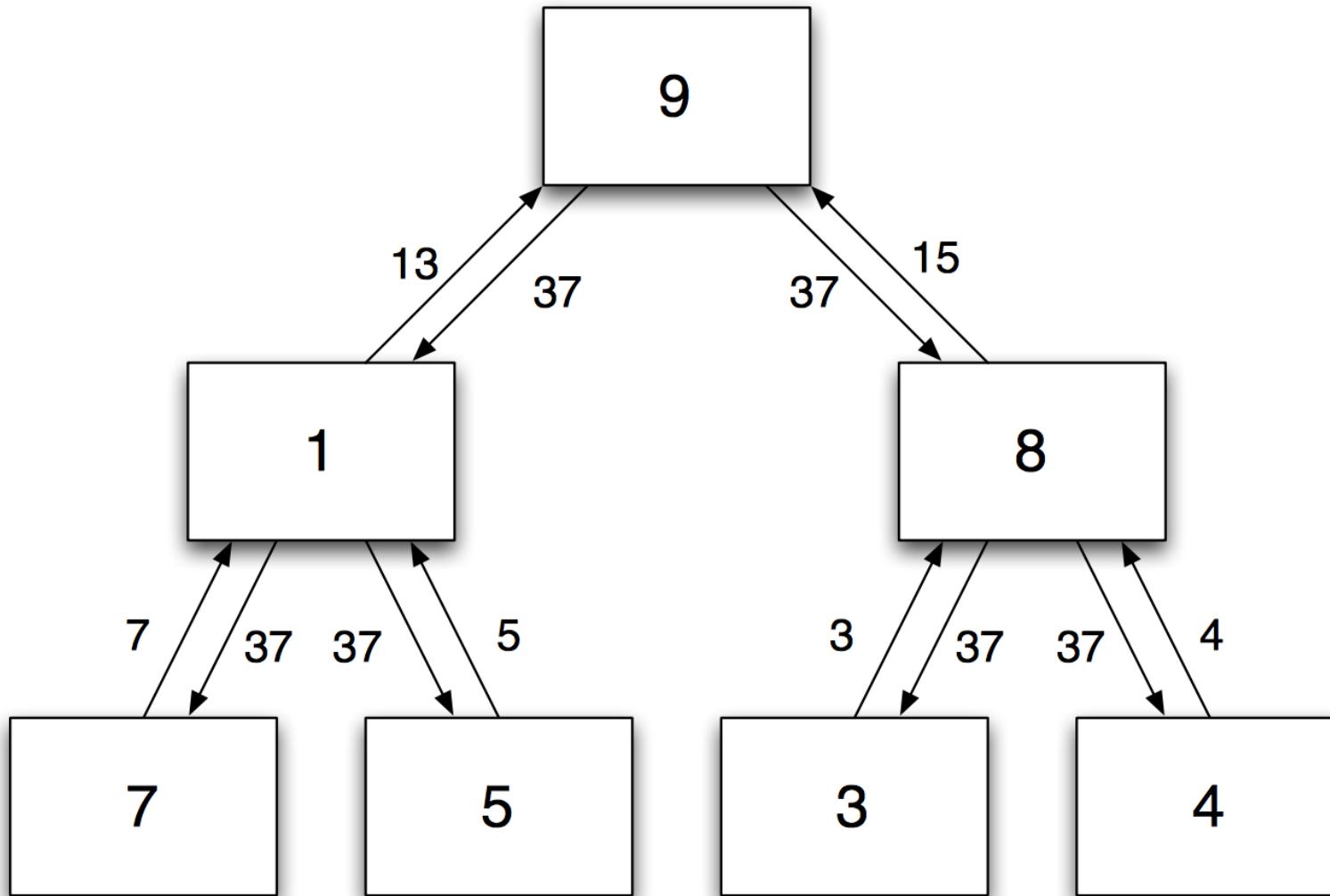
$(K_1, (V_1, V_2, \dots)) \rightarrow \text{List}(K_3, V_3)$

“Bear”, (1, 1)  $\rightarrow [(\text{“Bear”, 2})]$

# Парадигма Map-Reduce на примере Word Count

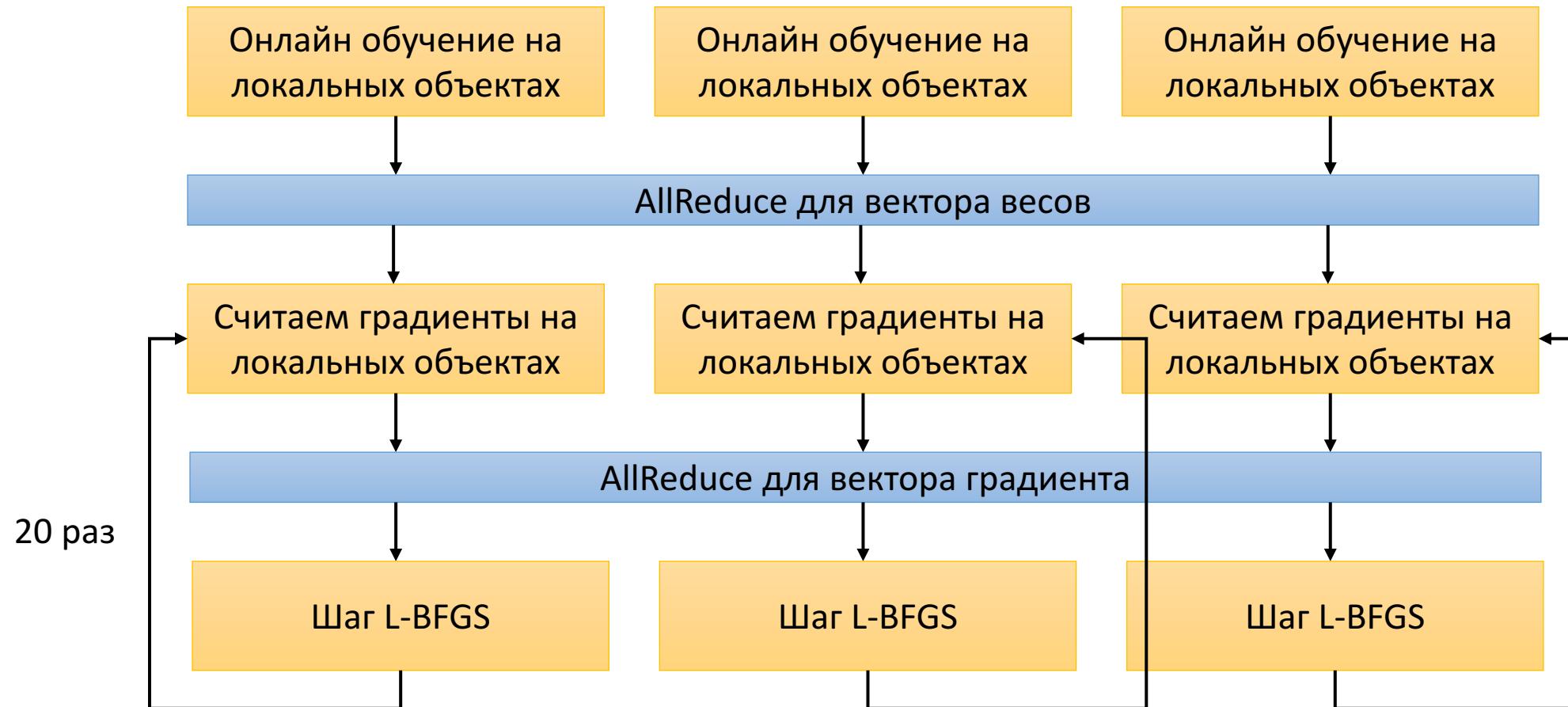


# AllReduce в VW (Опционально)



- Каждая машина считает вектор (градиент или веса) по своим объектам
- На каждой машине хочется получить сумму этих векторов от всех машин

# Гибридная схема работы VW (Опционально)



# Задача предсказания кликов по рекламе

- **2.1T** разреженных признаков
- **17B** примеров
- **16M** параметров (24 hash bits)
- **1K** машин
- Оптимальный линейный классификатор за **70 минут**

|       | 1%     | 10%    | 100%   |
|-------|--------|--------|--------|
| auROC | 0.8178 | 0.8301 | 0.8344 |
| auPRC | 0.4505 | 0.4753 | 0.4856 |
| NLL   | 0.2654 | 0.2582 | 0.2554 |

Сэмплирование ухудшает качество –  
нужно учиться на всех данных!

# Apache Spark



- Фреймворк для выполнения распределенных задач
- API на нескольких языках: Scala, Java, Python
- Будем рассматривать на примере Python API – PySpark
- Включает полезные модули:
  - MLlib: алгоритмы машинного обучения
  - Spark SQL: выполнение SQL запросов на кластере
  - Алгоритмы на графах, потоковая обработка данных, ...

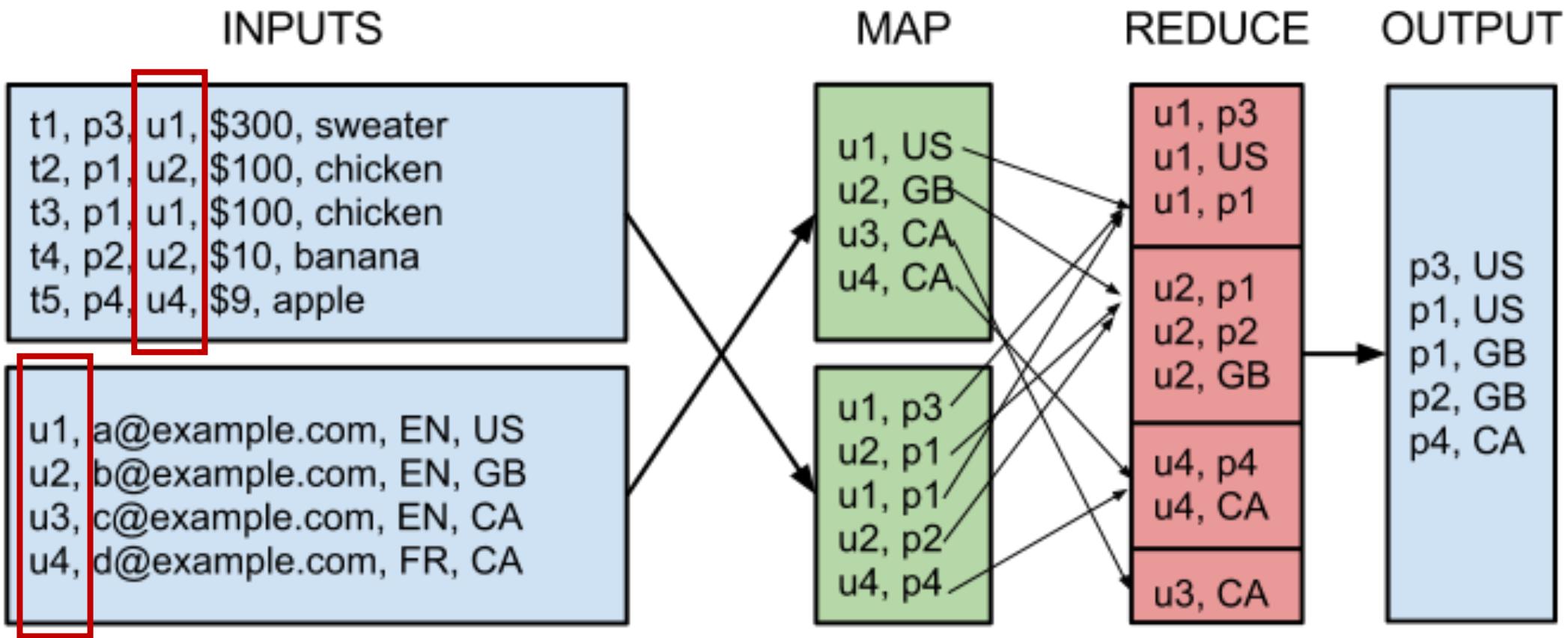
# Join на SQL

- Таблица **a** – покупки пользователей (**user**, **product**, ...)
- Таблица **b** – информация о пользователях (**user**, **state**, ...)
- Хотим получить покупки продуктов по штатам
- Нужно сделать join по **user**

```
select
    a.product,
    b.state
from
    a join b on a.user = b.user
```

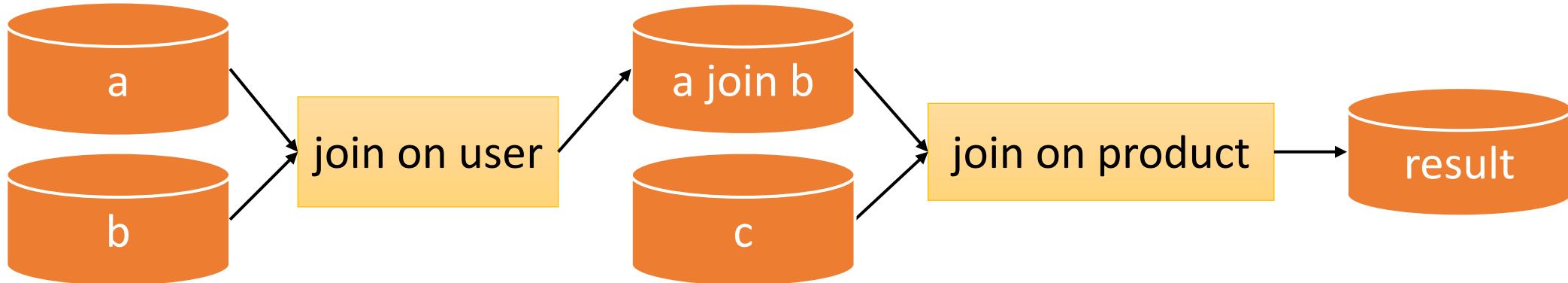
# Join на MapReduce

- Этот же запрос на MapReduce:



# MapReduce: еще один join

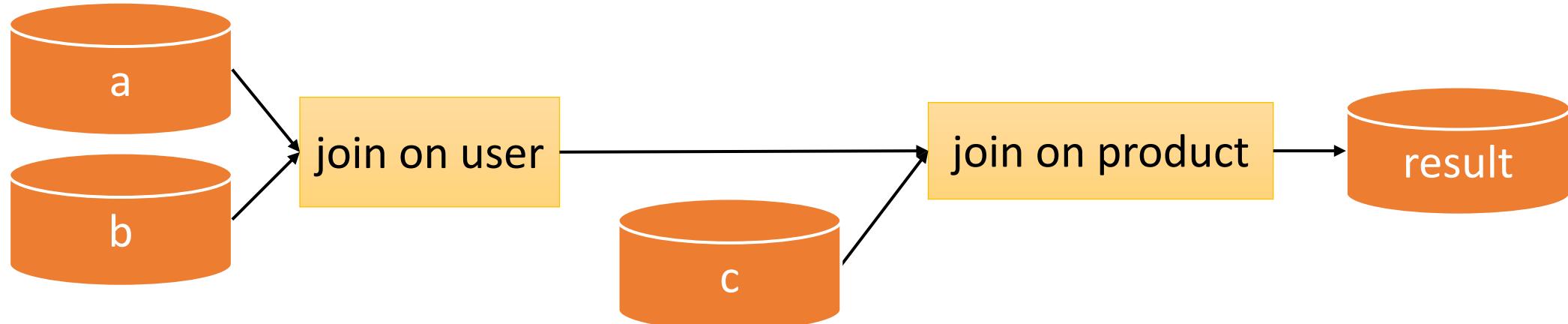
- В таблице **с** лежит информация о продуктах (**product**, **type**, ...)
- Решение на **Hadoop MapReduce**:



- Промежуточные результаты пишутся на диск и читаются с диска
- Это медленно и не всегда необходимо!

# Spark: еще один join

- Решение на **Spark**:



- Результаты вычислений передаются по возможности в памяти **без сохранения на диск**
- Промежуточные результаты можно **закэшировать** в памяти и выполнять над ними несколько операций (если хватает суммарной памяти машин)
- Вычисления можно представить в виде графа (**DAG** – направленный граф без циклов)

# RDD в Spark

- **RDD** (resilient distributed dataset) – восстанавливаемый распределенный набор данных
  - Набор данных хранится распределенно на разных машинах
  - Потерянные части могут быть восстановлены (известна цепочка вычислений – DAG)
- Как сделать RDD?
  - Из файла (например, в HDFS)
  - **Распараллелив Python коллекцию** (список, итератор, ...)
  - **Трансформацией** из другого RDD

# Операции над RDD

- Программа на Spark пишется в терминах операций над RDD
- **Трансформации:**
  - RDD → RDD
  - Ленивые – не вычисляются сразу
  - Примеры: `map`, `reduceByKey`, `join`
- **Действия:**
  - Приводят к вычислению RDD
  - Примеры: `saveAsTextFile`, `collect`, `count`
- **Другие операции:**
  - `persist`, `cache` – помечают RDD для сохранения в памяти/на диске при первом их вычислении

# Пример трансформации

```
rdd = (sc
        .parallelize([1, 2, 3, 4])
        .map(lambda x: x * 2))
print rdd
print rdd.collect()
```

```
PythonRDD[17] at RDD at PythonRDD.scala:48
[2, 4, 6, 8]
```

# Пример трансформации

```
rdd = (sc
        .parallelize([1, 2, 3, 4])
        .flatMap(lambda x: [x, x * 2]))
print rdd
print rdd.collect()
```

```
PythonRDD[19] at RDD at PythonRDD.scala:48
[1, 2, 2, 4, 3, 6, 4, 8]
```

# Пример действия

```
rdd = (sc
        .parallelize(np.random.random((1000,))))
        .flatMap(lambda x: [x, x * 2]))
print rdd
print rdd.takeOrdered(2, lambda x: -x)
```

```
PythonRDD[29] at RDD at PythonRDD.scala:48
[1.9987386963918603, 1.997388155520317]
```

# MapReduce как две операции в Spark

```
rdd = (  
    sc  
.parallelize(["this is text", "text too"])  
.flatMap(lambda x: [(w, 1) for w in x.split()])  
.reduceByKey(lambda a, b: a + b))  
  
print rdd  
print rdd.collect()
```

```
PythonRDD[61] at RDD at PythonRDD.scala:48  
[( 'text', 2 ), ( 'too', 1 ), ( 'is', 1 ), ( 'this', 1 )]
```

# DataFrame API



- Работает с DataFrame (таблицы)
- Поддерживает SQL запросы
- Умеет конвертировать из/в Pandas DataFrame
- Вычисления производятся на Java (без Python)
  - Для этого нужно использовать базовые типы и коллекции в колонках

# DataFrame API пример

```
import pandas as pd
from pyspark.sql import SparkSession

ss = (SparkSession
      .builder
      .appName("spark sql example")
      .getOrCreate())
```

```
df = pd.DataFrame(
    [[ "cat", [1, 1]], [ "cat", [2]], [ "dog", [1]]],
    columns=[ "name", "cnt"])
```

# DataFrame API пример

```
df
```

|          | <b>name</b> | <b>cnt</b> |
|----------|-------------|------------|
| <b>0</b> | cat         | [1, 1]     |
| <b>1</b> | cat         | [2]        |
| <b>2</b> | dog         | [1]        |

```
sdf = ss.createDataFrame(df)
```

```
sdf.printSchema()
```

```
root
```

```
|-- name: string (nullable = true)
|-- cnt: array (nullable = true)           ← Native Java Types
|   |-- element: long (containsNull = true)
```

# DataFrame API пример

```
sdf.registerTempTable("animals")
```

```
ss.sql("""  
select name, sum(cnt) as sum  
from  
    (select name, explode(cnt) as cnt  
     from animals)  
group by name  
""").toPandas()
```

← Выполняется на кластере

|          | <b>name</b> | <b>sum</b> |
|----------|-------------|------------|
| <b>0</b> | dog         | 1          |
| <b>1</b> | cat         | 4          |

# Рекомендательные системы

Вам также могут понравиться



Tefal GV7485

от 13 790 руб.

0 отзывов 6 предложений

Утюг

- с парогенератором
- мощность 2000 Вт
- мощный паровой удар
- мощность подачи пара до 120 г/мин



Цены



Philips GC 1026

от 1 448 руб.

0 отзывов 94 предложения

Утюг

- мощность 2000 Вт
- мощность подачи пара до 25 г/мин
- вес 0.9 кг
- паровой удар



Цены



Tefal FV2352E0

от 1 800 руб.

1 отзыв 9 предложений

Утюг

- мощность 2000 Вт
- мощность подачи пара до 30 г/мин
- вес 1.4 кг
- паровой удар



Цены



Sinbo SSI-2872

от 900 руб.

1 отзыв 94 предложения

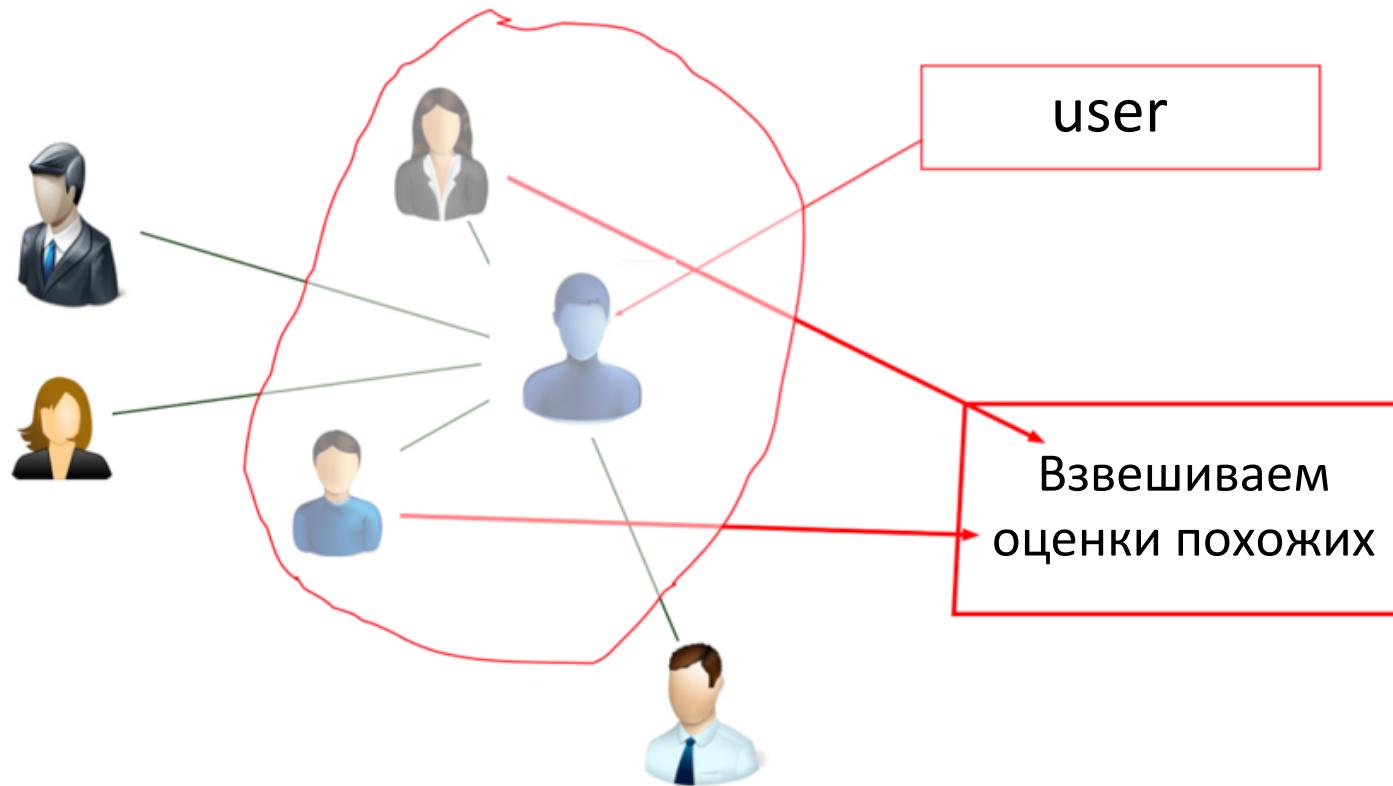
Утюг

- мощность 2000 Вт
- керамическая подошва
- паровой удар
- вертикальное отпаривание



Цены

# User-based CF



# User-based CF

- Как считать похожесть пользователей?
- Корреляция оценок!

# Пример User-based CF



# Пример User-based CF



# Пример User-based CF



# Приимер User-based CF



# Пример User-based CF



# User-based CF

- Минусы user-based:
- У большинства пользователей не так много оценок, что приводит к неуверенной оценке похожести пользователей.
- Оценки конкретного пользователя меняются во времени, поэтому при добавлении хотя бы одной новой оценки его похожесть на других пользователей может сильно измениться.

# User-based CF

- $n$  пользователей,  $m$  товаров
- $r_{ui}$  – оценка пользователя и товару  $i$
- $r_u$  – средняя оценка пользователя и
- Похожесть пользователей  $u$  и  $v$ :

Пустые ячейки остаются пустыми (нулевыми) после вычитания среднего!

$$\text{corr}(u, v) = \frac{\sum_{i=1}^m (r_{ui} - r_u)(r_{vi} - r_v)}{\sqrt{\sum_{i=1}^m (r_{ui} - r_u)^2} \sqrt{\sum_{i=1}^m (r_{vi} - r_v)^2}}$$



- Предсказание:

$$\widehat{r}_{ui} = r_u + \frac{\sum_{v=1}^n \text{corr}(u, v)(r_{vi} - r_v)}{\sum_{v=1}^n |\text{corr}(u, v)|}$$

# Item-based CF



Рекомендуем похожие товары к товарам из истории пользователя

# Item-based CF

- Похожие формулы
- Более увереные оценки похожести товаров
- Медленнее устаревают похожести товаров
- Как работает:
  - Посчитали заранее Item-Item похожести
  - Приходит пользователь, делает новый клик
  - По его кликам можно найти похожие товары в real-time

# Параллелим Item-based CF

- $n$  пользователей,  $m$  товаров,  $n \sim m \sim 10^6$
- Нужно посчитать заранее Item-Item похожести:

$$\text{corr}(i, j) = \frac{\sum_{u=1}^n (r_{ui} - r_u)(r_{uj} - r_u)}{\sqrt{\sum_{u=1}^n (r_{ui} - r_u)^2 \sum_{u=1}^n (r_{uj} - r_u)^2}}$$

Сумма только по пользователям,  
которые смотрели  $i$  и  $j$  фильм



adjusted  
cosine  
similarity

- Наивный подход –  $O(m^2n)$
- Матрица оценок сильно разрежена, можно лучше?
- Вклад в похожесть двух товаров ненулевой только от пользователей, которые оценили оба этих товара

# Шаг Мар

|  |   |   |   |   |   |   |
|--|---|---|---|---|---|---|
|  |   |   |   |   |   |   |
|  | 2 |   |   | 4 | 5 |   |
|  | 5 |   | 4 |   |   | 1 |
|  |   |   | 5 |   | 2 |   |
|  |   | 1 |   | 5 |   | 4 |
|  |   |   | 4 |   |   | 2 |
|  | 4 | 5 |   | 1 |   |   |

$$(1, 3) \rightarrow 5 * 4$$

$$(1, 6) \rightarrow 5 * 1$$

$$(3, 6) \rightarrow 4 * 1$$

# Как посчитать все числители на Spark

```
def emit_pairs(x):
    user, items = x
    items = sorted(items)
    if len(items) < 300:
        for i in range(len(items) - 1):
            for j in range(i + 1, len(items)):
                yield (
                    (items[i][0], items[j][0]),
                    items[i][1] * items[j][1]
                )
    dot_product = (
        ratings
        .map(lambda x: (x.user, (x.product, x.rating)))
        .groupByKey()
        .flatMap(emit_pairs)
        .reduceByKey(lambda x, y: x + y)
    )
```

$$corr(i, j) = \frac{\sum_{u=1}^n (r_{ui} - r_u)(r_{uj} - r_u)}{\sqrt{\dots}}$$

# Решение на Spark SQL

**SELECT**

```
ic1.itemId,  
ic2.itemId AS jointItemId,  
SUM(ic1.val * ic2.val) AS cosine  
FROM ic AS ic1  
    JOIN ic AS ic2 ON ic1.clientId = ic2.clientId  
WHERE ic1.itemId < ic2.itemId  
GROUP BY ic1.itemId, ic2.itemId;
```

# Резюме

- Линейные модели можно учить на огромных выборках (спам, реклама)
- Хэширование позволяет учесть большое количество исходных признаков
- Для работы с большими данными удобна парадигма Map-Reduce
- SQL запросы бывают полезны и их можно превратить в Map-Reduce операции
- На Map-Reduce можно считать много всего, например, похожести для рекомендательной системы

# Ссылки

- A Reliable Effective Terascale Linear Learning System  
<https://arxiv.org/pdf/1110.4198.pdf>
- <http://civr.cs.nyu.edu/diglib/lsm/lsm/lecture01-online-linear.pdf>
- <http://mlwave.com/predicting-click-through-rates-with-online-machine-learning/>
- <http://spark.apache.org/docs/latest/programming-guide.html>
- <https://www.kaggle.com/tkm2261/outbrain-click-prediction/bigquery-is-cool-lb-0-63692>