


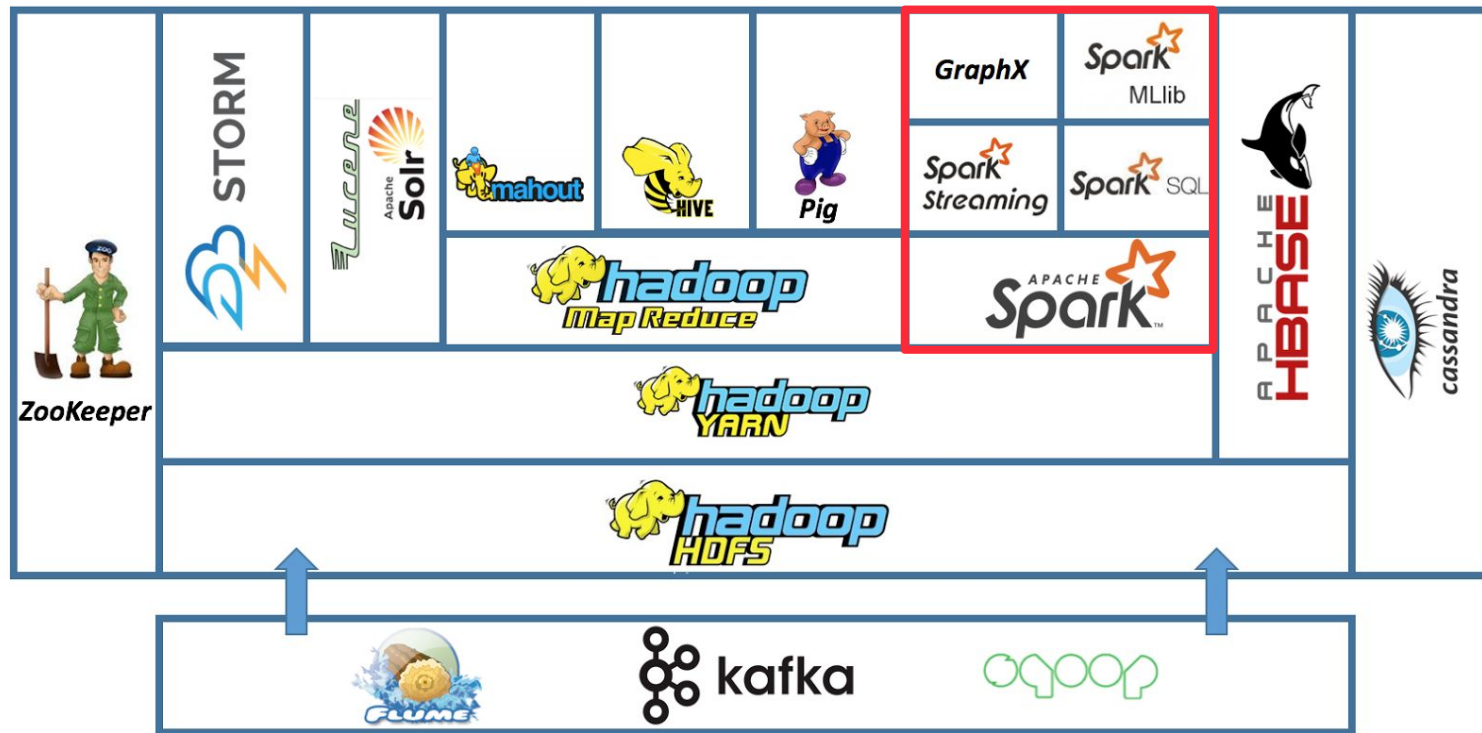
# Approximate алгоритмы для больших данных

Андрей Кузнецов  
26.11.2022

# Структура курса

1. Введение в Большие Данные
2. Hadoop экосистема и MapReduce
3. SQL поверх больших данных
4. Инструменты визуализации при работе с Большими Данными
5. Введение в Scala
6. Устройство и API Spark
7. Approximate алгоритмы для больших данных 
8. Поточковая обработка данных (Kafka, Spark Streaming, Flink)
9. Основы распределённой СУБД Apache Cassandra

# Hadoop ecosystem



# План лекции

1. Хэш и примеры применений
  - а. Расчет статистик
  - б. Извлечение признаков
  - с. Разделение на группы
  - д. Фильтрация
2. Поиск ближайших соседей
  - а. Примерный подсчет расстояния
  - б. Примерный поиск ближайших соседей



Apprrox methods

Зачем примерные методы?

**Считать в лоб - дорого**

# Зачем примерные методы?

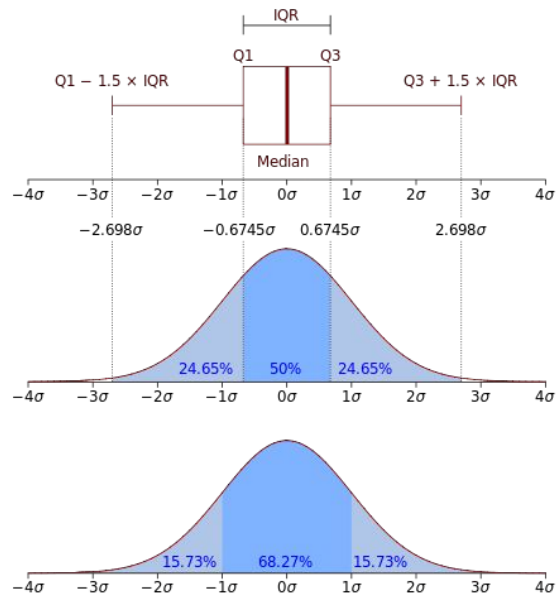
**Задача:** посчитать статистики большого распределенного датасета

Считать в лоб - дорого

**Решение:**

`df.stat.approxQuantile("x", Array(0.5), 0.25)`

This method implements a variation of the Greenwald-Khanna algorithm



# Хэш

Хэш функция переводит вход (число / строку / whatever) в целочисленный номер корзины (bucket number)

Хороший хэш должен равномерно распределять по корзинам ключи.

Хэш от одного ключа всегда попадает в одну корзину

**Пример:** остаток от деления на простое число для целочисленных ключей

**Пример:** конвертируем символ для строки в Int (Unicode / ASCII), суммируем.



# Hashing trick

Реальные данные могут оказаться гораздо более динамичными, и мы не всегда можем рассчитывать, что категориальные признаки не будут принимать новых значений. Все это сильно затрудняет использование уже обученных моделей на новых данных. Кроме того, **LabelEncoder** подразумевает предварительный анализ всей выборки и хранение построенных отображений в памяти, что затрудняет работу в режиме больших данных.

## Аналог One-Hot-encoding

1. Рассмотрим фичу с именем “country”
2. Создаем хэш-таблицу с весами линейной модели
3. Пусть для текущей строки ее значение “russia”
4. Берем хэш от “country\_russia” и достаем из хэш-таблицы соответствующий вес

## HashingTF в Spark

1. Вместо сохранения конкретных слов, берем хэши от них
2. Сохраняем это в хэш-таблицу (хэш слова -> встречаемость)

## A/B/n-тесты

1. Берем id пользователя – превращаем в строку
2. Конкатенируем с солью (например, название текущего теста)
3. Берем от этого хэш (MurMurHash3 – хороший вариант)
4. Выделяем остаток от деления значения хэша на большое простое число

# Bloom filter

## **Задача**

У нас есть очень большое множество

Хотим проверить входят ли какие-то элементы в него

## **Подход в лоб**

Делаем хэш-таблицу по данным

Смотрим, что хэш для нового элемента уже существует

Быстро (лукап по хэш-таблице – константа), но требует много места

## **Вероятностный подход - Блум Фильтр**

# Bloom filter

Имеет две операции:

1. добавление в множество
2. проверка на отсутствие

Состоит из  $n$  бит (сначала все 0) и  $k$  хэш-функций, которые кладут в один из  $n$  бит

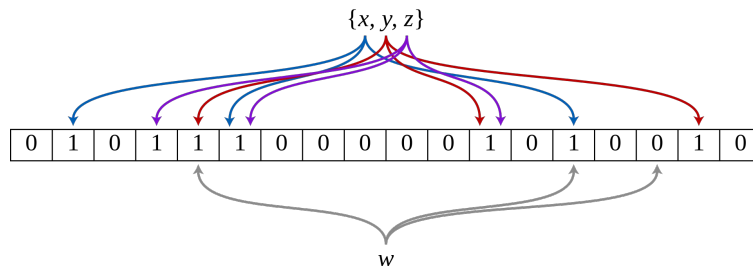
## Добавление

1. Считаем  $k$  хэш-функций от элемента
2. Меняем 0 на 1 (или оставляем 1) в корзинках, которые вернули хэши

## Поиск

1. Считаем  $k$  хэш-функций от элемента
2. Если какой-то бит равен 0, то элемента нет в множестве

Но могут быть коллизии!



# Bloom filter

## Но могут быть коллизии!

Давайте посчитаем вероятность False Positive через параметры  $n$ ,  $k$  и количества элементов в множестве ( $m$ )

1. Каждая хэш-функция попадает равномерно в одну из  $n$  корзинок, тогда вероятность промаха по конкретной корзине  $1 - \frac{1}{n}$
2. Тогда вероятность, что все хэш-функции промахнутся мимо этого бита равна  $\left(1 - \frac{1}{n}\right)^k$
3. Всего  $m$  элементов, поэтому вероятность, что в заполненном фильтре пустой бит равна  $\left(1 - \frac{1}{n}\right)^{km}$
4. Вероятность того, что все хэши выставлены для нового элемента

$$\left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k \approx \left(1 - e^{-\frac{km}{n}}\right)^k$$

# Bloom filter

Вероятность коллизии  $p = (1 - e^{-\frac{km}{n}})^K$  минимальна при  $k = \frac{n}{m} \ln 2$

Подставив  $k$  в формулу  $p$  получим  $\ln p = -\frac{n}{m} (\ln 2)^2$

Отсюда

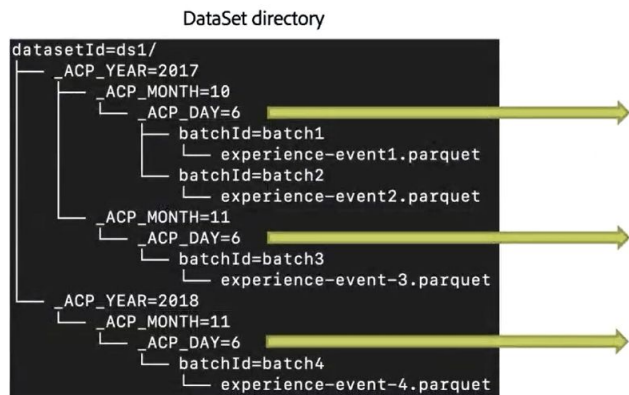
$$n = -\frac{m \ln p}{(\ln 2)^2} \approx -1.44 m \log_2 p$$
$$k = -\log_2 p$$

Если у нас  $m = 10^6$  элементов и хотим ошибаться не чаще  $p = 0.01$

Количество требуемых бит  $n \approx 10^7, k \approx 7$

Обычная хэш-таблица при хэше в 32 бита –  $3.2 * 10^7$

# Bloom filter. Parquet



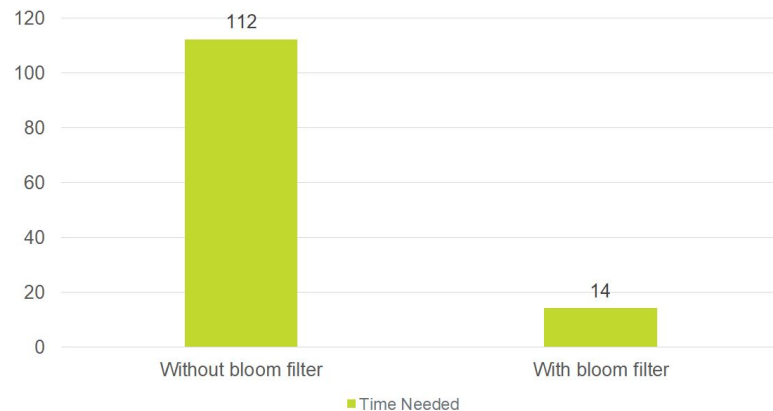
Bloom Filter

Bloom Filter 1

Bloom Filter 2

Bloom Filter 3

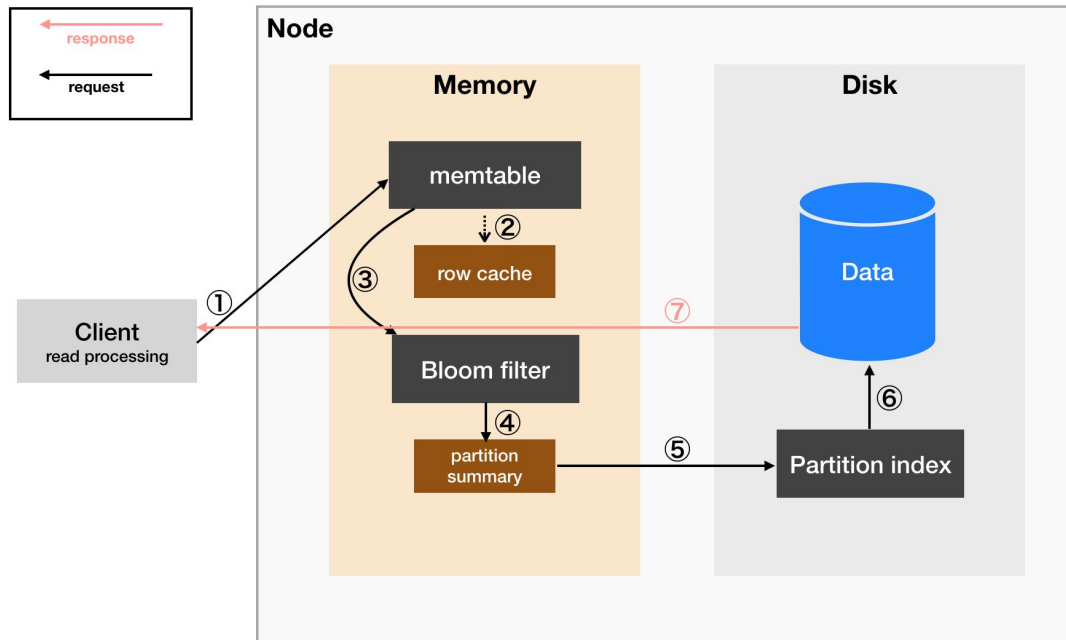
Cost to Scan Dataset 100 times (hours)



<https://medium.com/@songkunjump/parquet-bloom-filter-with-spark-495a5f019c6c>

<https://blog.developer.adobe.com/search-optimization-for-large-data-sets-for-gdpr-7c2f52d4ea1f>

# Bloom filter. Cassandra





# Count-min sketch

## **Задача**

Имеем нагруженный поток данных

Хотим посчитать как часто конкретный элемент встречается в потоке

## **Подход в лоб**

Делаем хэш-таблицу элемент – встречаемость

Смотрим по хэшу встречаемость элемента

Быстро (лукап по хэш-таблице – константа), но требует много места

# Count-min sketch

## Операции

- Инкремент счетчика для элемента
- Извлечение счетчика для элемента

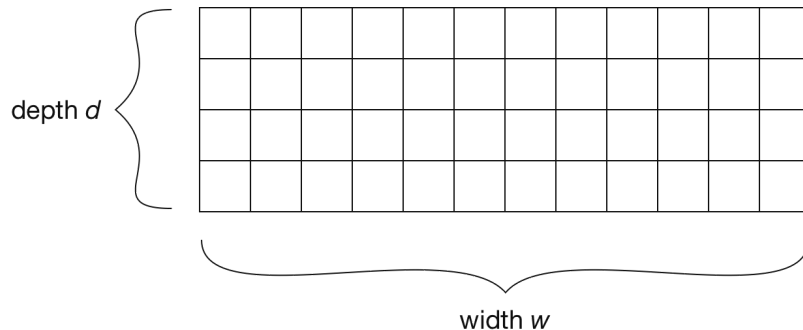
Состоит из  $d$  разных хэш-функций с  $w$  корзинами

## Добавление

1. Считаем  $d$  хэш-функций от элемента
2. Добавляем 1 в соответствующие ячейки

## Извлечение

1. Считаем  $d$  хэш-функций от элемента
2. Берем минимум из полученных значений в ячейках





Поиск похожих

# Поиск похожих

Поиск наиболее похожих элементов (документов, товаров, пользователей...) согласно заданной метрике близости или расстояния (Жаккард, Косинус, Евклид,...)

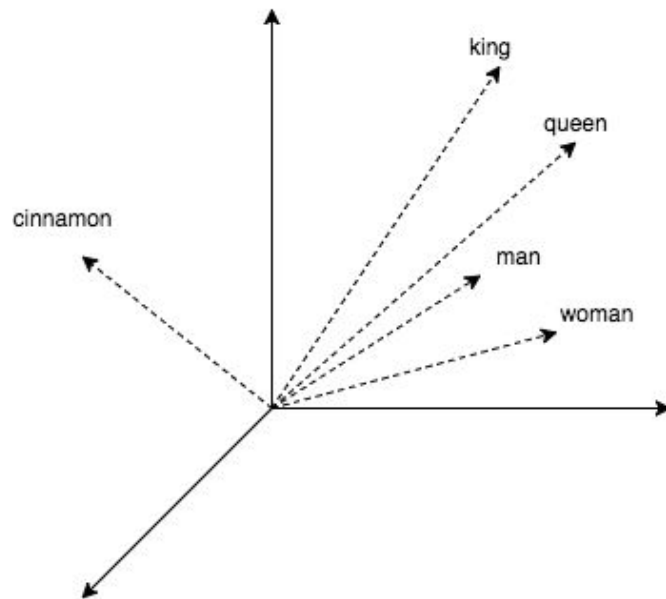
Классические примеры: k-ближайших соседей, ранжирование (рексистемы, чат-боты, распознавание лиц, ), дедупликация документов

**Расстояние** – переводит пару элементов некоторого пространства в вещественное число

Должно удовлетворять условиям:

1. Расстояние неотрицательно
2. Расстояние равно нулю только тогда, когда совпадают элементы
3. Расстояние симметрично

**Примеры:** Jaccard, Euclidian, Cosine



# Min-hash

Позволяет быстро посчитать меру Жаккарда

<i>Element</i>	$S_1$	$S_2$	$S_3$	$S_4$
<i>a</i>	1	0	0	1
<i>b</i>	0	0	1	0
<i>c</i>	0	1	0	1
<i>d</i>	1	0	1	1
<i>e</i>	0	0	1	0

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Предположим, у нас есть: два множества  $A$ ,  $B$  и хэш-функция  $h$ , которая умеет считать хэши для элементов этих множеств. Далее, определим функцию  $h_{\min}(S)$ , которая вычисляет функцию  $h$  для всех членов какого-либо множества  $S$  и возвращает наименьшее её значение. Теперь, начнём вычислять  $h_{\min}(A)$  и  $h_{\min}(B)$  для различных пар множеств, вопрос: чему равна вероятность того, что  $h_{\min}(A) = h_{\min}(B)$ ?

Если задуматься, эта вероятность должна быть пропорциональна размеру пересечения множеств — при отсутствии общих членов, она стремится к нулю, и к единице, когда множества равны, в промежуточных случаях она где-то посередине. Ничего не напоминает? Ага, всё верно, это и есть  $J(A, B)$  — коэффициент Жаккара!

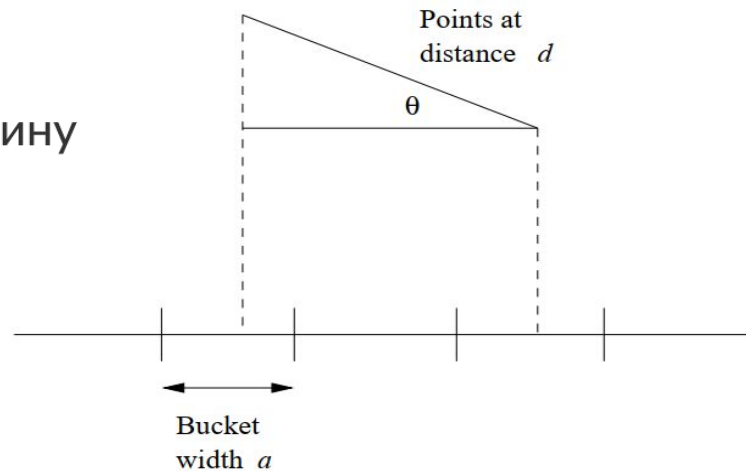
# Bucketed random projection (Euclidean)

1. Проводим прямую
2. Делим на корзинки равной длины
3. Смотрим попали ли две точки в одну корзину

Для проекции используем хэш-функцию

$$h(x) = \left\lfloor \frac{v \cdot x}{a} \right\rfloor$$

$v$  – случайный единичный вектор



# Annoy

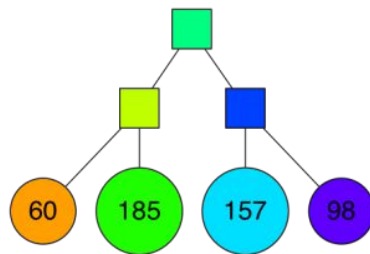
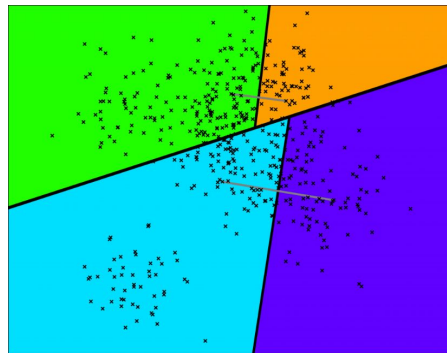
Выберем две случайные точки и проведем к ним нормаль

Повторим для подплоскости, пока в листе останется не больше заданного  $K$  элементов

Сохраним индекс в бинарное дерево

Но одно дерево слишком неточное

Соберем лес!



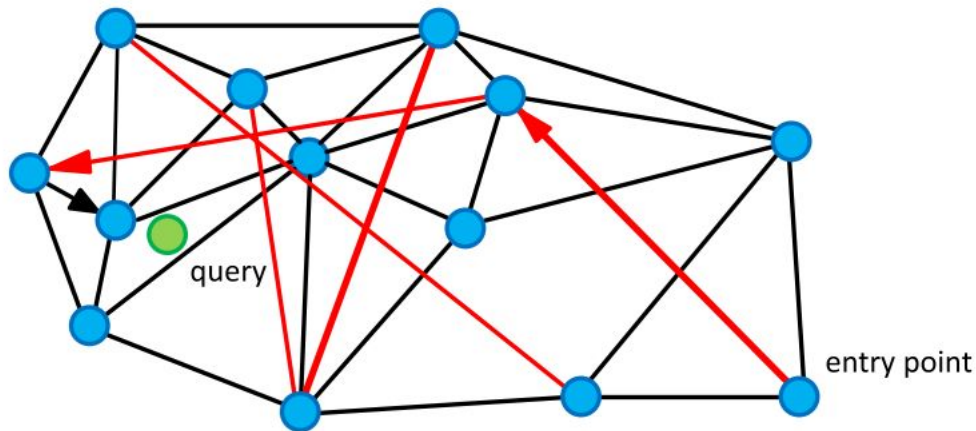
# Navigable Small World (NSW)

Строим Small World граф по данным

При запросе попадаем на случайную вершину

Идем в ближайшую к запросу соседнюю вершину

Повторяем, пока не окажемся в ближайшей точке.



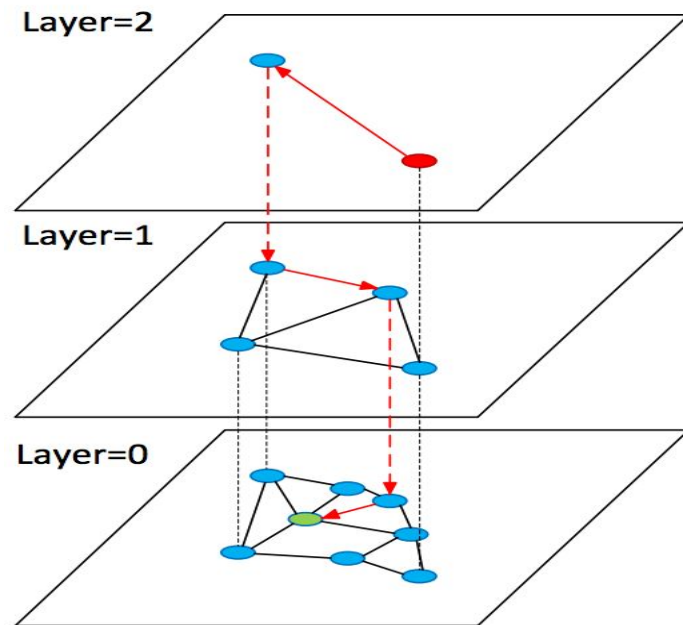


# Navigable Small World (NSW)

Теперь разделим граф на слои

Все точки со слоя  $n$  переходят на слой  $n + 1$

1. Выбираем случайную точку на верхнем слое
2. Используем механизм NSW
3. Как только нашли ближайшую точку на слое, спускаемся ниже



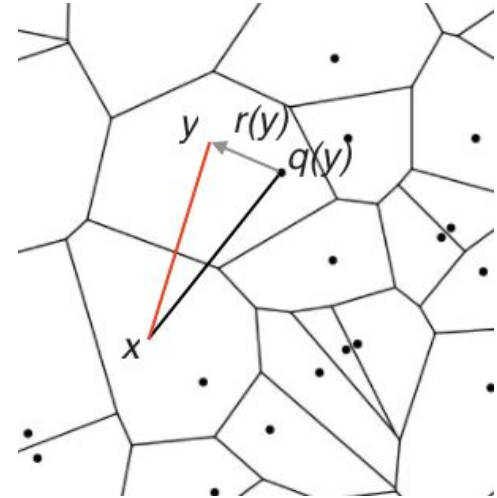
# Facebook AI research Similarity Search (FAISS)

FAISS состоит из трех основных частей

1. Asymmetric distance computation (ADC)
2. Inverted file (IVF)
3. Product quantization (PQ)

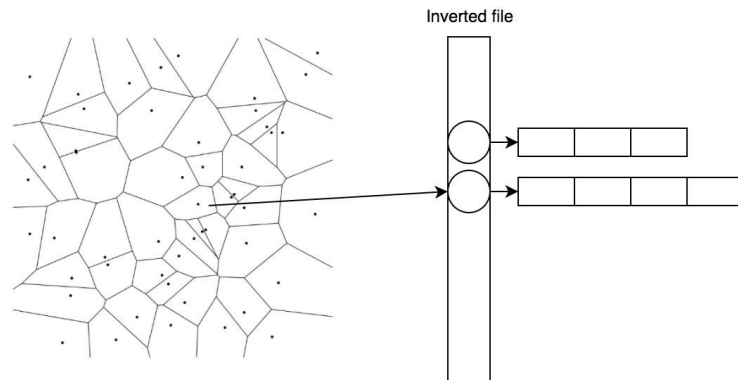
# FAISS. ADC

1. Воспользуемся идеей корзинок и разделим наше пространство на кластеры с помощью K-Means
2. Для репрезентации корзины возьмем вектор центрального элемента
3. При запросе находим ближайший центр
4. Достаем элементы кластера через IVF



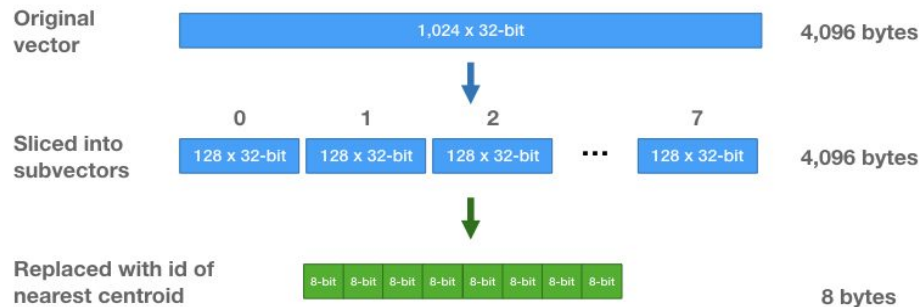
# FAISS IVF

Для центров кластера просто храним  
список элементов в нем



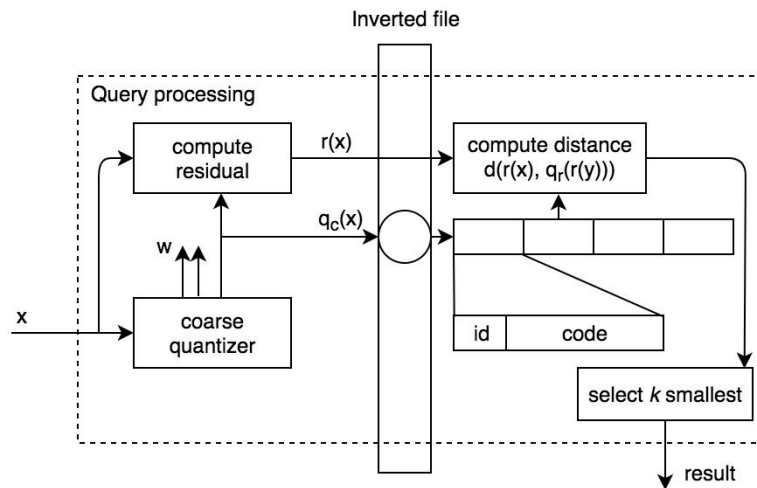
# FAISS: PQ

1. Вычитаем элементы центроида из вектора
2. Делим полученный вектор на корзинки
3. Каждую из частей кластеризуем и заменяем индексом центра



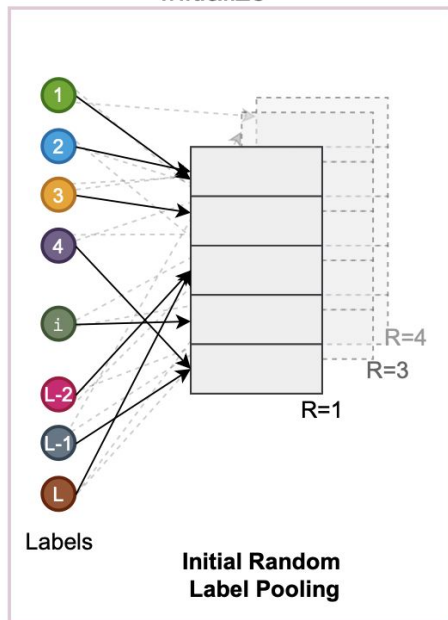
# FAISS: Поиск

1. При запросе находим несколько ближайших центров кластеров
2. Достаем элементы кластера через IVF
3. Остаток вектора от запроса до центра кластера кодируется через PQ
4. Расстояние от запроса до элемента определяется, как сумма расстояний от центров кластеров между всеми корзинками
5. Достаем ближайших

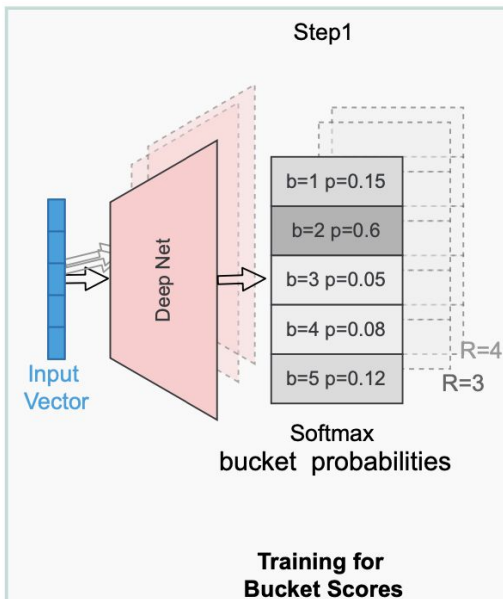


# BLISS

## Initialize

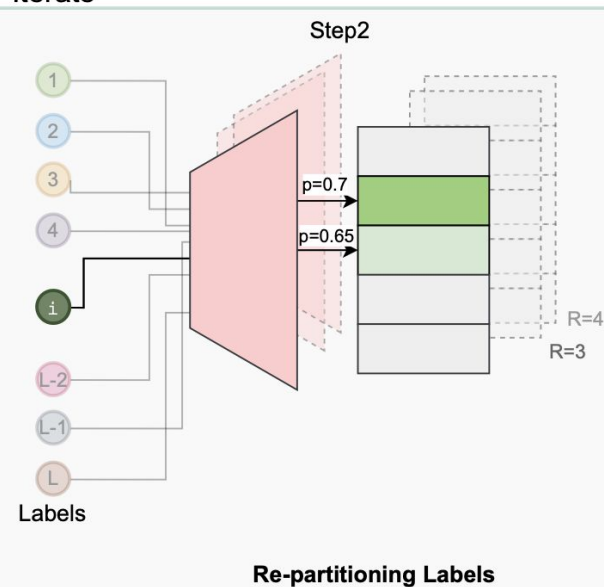


## Step1

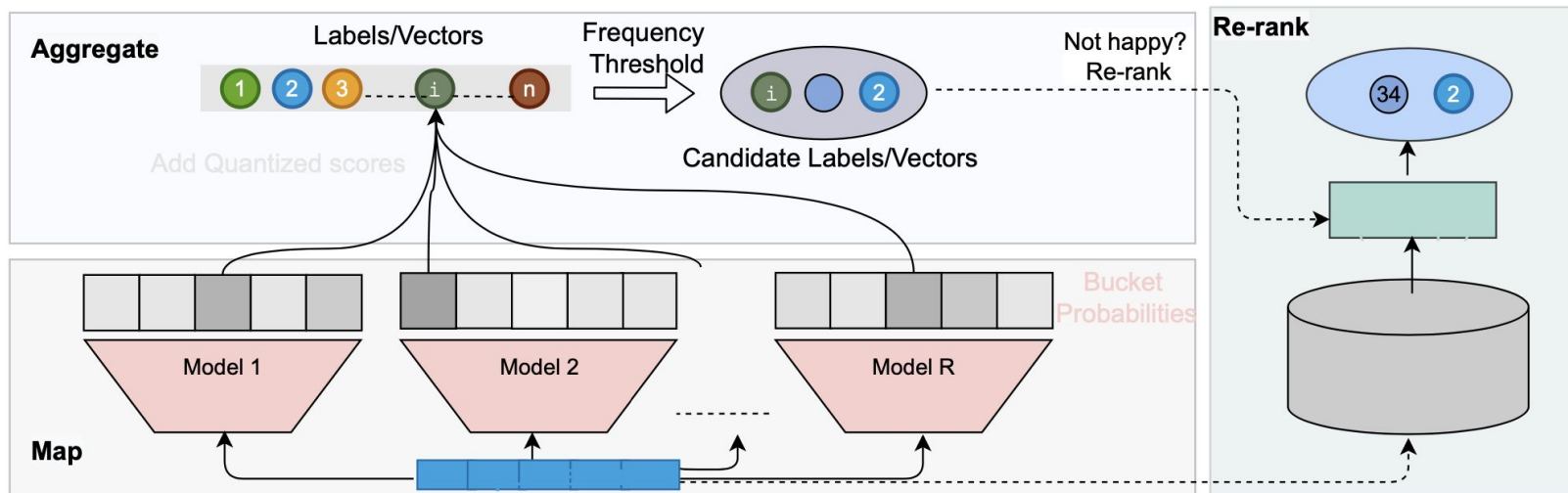


## Iterate

### Step2



# BLISS





# BLISS

	QPS				Recall10@10				Index size (construction time)			
Dataset	BLISS	BLISS <sup>+</sup>	HNSW	FAISS	BLISS	BLISS <sup>+</sup>	HNSW	FAISS	BLISS	BLISS <sup>+</sup>	HNSW	FAISS
Deep1B	249	<b>400</b>	277	222	<b>0.9183</b>	0.81	0.8825	<b>0.919</b>	<b>15.5GB</b>	<b>137MB</b>	437GB	97GB
	384	<b>1724</b>	476	<b>769</b>	<b>0.8828</b>	0.6	0.846	0.7887	(1hr)	(1.2hrs)	(9hrs)	(>5days)
BIGANN	121	344	<b>909</b>	243	0.8443	0.658	0.8734	<b>0.8764</b>	<b>15.5GB</b>	<b>137MB</b>	557GB	127GB
	344	<b>909</b>	625	526	<b>0.792</b>	0.516	0.76	0.7495	(1hr)	(1.1hrs)	(10hrs)	(>5days)
Yandex TI	<b>110</b>	<b>384</b>	15	4	<b>0.568</b>	0.434	0.566	0.4919	<b>15.5GB</b>	<b>137MB</b>	826GB	194GB
	<b>1631</b>	<b>1470</b>	102	18	<b>0.4544</b>	<b>0.3053</b>	0.2629	0.272	(1hr)	(1.3hrs)	(16hrs)	(>5days)
MSSpaceV	220	270	322	<b>613</b>	0.8328	0.73	0.830	0.843	<b>15.5GB</b>	<b>137MB</b>	452GB	101GB
	416	<b>1333</b>	1075	1216	0.7879	0.6705	0.784	0.7852	(1hr)	(1.1hrs)	(9hrs)	(>5days)

**Table 2: Query Per Sec, Recall10@10, Index size and construction time number on 4 Large scale data [2] [17] [26] [37] against the popular baselines FAISS[18] and HNSW [24]. FAISS took a long time for indexing, it couldn't finish the construction for Yandex and MSSpaceV in the given time constraint.**

# Qdrant

Dataset: deep-image-96-angular ▾ Search threads: 100 ▾ Plot values: ● RPS | ○ Latency | ○ p95 latency | ○ Index time



Qdrant (read: quadrant) is a vector similarity search engine. It provides a production-ready service with a convenient API to store, search, and manage points - vectors with an additional payload.

Qdrant is tailored to extended filtering support. It makes it useful for all sorts of neural network or semantic-based matching, faceted search, and other applications.

# Lecture summary

Хэши позволяют многое ускорить:

1. Подготовка фичей
2. Фильтрация данных
3. Подсчет статистик
4. Оценка похожести
5. Поиск ближайших

Но нужно помнить, что они вносят ошибку

# Recommended literature

