
OTUS
ОНЛАЙН-ОБРАЗОВАНИЕ

OTUS

Алгоритмы HyperLogLog и Count-min Sketch

Михаил Горшков
разработчик

HyperLogLog & Count-Min Sketch

- работаем с мультисетами — множествами, где каждый уникальный элемент может появиться более, чем 1 раз:

{2, 15, 1, 1, 36, 2}

- актуально для больших данных;
- HyperLogLog может определить (приблизительно) сколько уникальных элементов имеет мультисет;
- Count-Min Sketch может определить (приблизительно), сколько раз встречается в мультисете каждый элемент;

Постановка задач

Решаем 2 задачи в "точном" виде.

Дан мультисет, нужно найти:

- количество уникальных элементов (англ. *cardinality*);
- сколько раз встречается каждый элемент.

Для нашего примера:

{2, 15, 1, 1, 36, 2}

ответы:

- 4 уникальных элемента;
- 2 - 2 раза, 15 - 1 раз, 1 - 2 раза, 36 - 1 раз.

Как решить задачу? Нужен алгоритм.

Ваши варианты

- использовать массив, отсортировать; (?)
- использовать вместо массива специализированные контейнеры.

Рассмотрим мультисеты в C++ (для остальных есть аналоги, но не в ядре языка).

In [ ]:

```
// C++
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class multiset;
```

std::multiset

std::multiset - контейнер, который хранит сортированный набор объектов типа Key. В отличие от std::set, допустимы одинаковые значения ключей. Обычно реализован как бинарное дерево поиска (RB-tree).

Задачи:

- как определить, сколько уникальных элементов содержит multiset, если он реализован как бинарное дерево поиска?
- как определить, сколько раз встречается в multiset-е каждый элемент?

Ваши варианты??

In [ ]:

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_multiset;
```

std::unordered_multiset

std::unordered_multiset - hash-based контейнер, который хранит набор объектов типа Key (возможно, не уникальных). Элементы помещаются в bucket-ы в зависимости от значения хэш-функции.

Задачи:

- как определить, сколько уникальных элементов содержит unordered_multiset?
- как определить, сколько раз встречается в unordered_multiset-е каждый элемент?

Ваши варианты??

HyperLogLog

Задача - подсчитать (приблизительно) число уникальных элементов в массиве

Зачем это нужно?

- обнаружение распространения вирусов, обнаружение сетевых атак (напр., DDOS - малое к-во уникальных и большое к-во одинаковых паттернов в трафике подозрительно);
- обнаружение спама - также большое к-во одинаковых паттернов подозрительно;
- data mining - нахождение к-ва одинаковых самплов данных;

Ограничения области применения

- хорошо (точно) работает на больших объемах данных;
- плохо на малых объемах данных (для них лучше применять традиционные алгоритмы).

Задача: необходимо подсчитать число уникальных пользователей, заходящих на ресурс в течение дня

Как это сделать? Ваши варианты

Использовать хэш-таблицу, добавляя пользователя в нее, когда он заходит на ресурс. Если пользователь входит с разных ip, девайсов, несколько раз, считаем это за 1 вход (пользователь должен залогиниться, учитываем login id). В конце дня считаем и сохраняем размер хэш-таблицы и записываем его в таблицу статистики. Очищаем хэш-таблицу, считаем следующий день и т.д.

Если у нас миллиарды пользователей каждый день, а мы храним user id как 32-битное целое число, понадобится 4 Гб, чтобы сохранить пользователей. Можно использовать битовую карту —каждый пользователь это бит—тогда все равно нужен 1 Гб памяти.

Хэш-таблица и битмап имеют требование месту O(n), место прямо пропорционально к-ву данных.

Другие метрики могут потребовать еще места.

Нас устроит не совсем точное значение, потому что предполагается, что к-во пользователей очень велико и оно постоянно увеличивается. Нужен +- порядок величины, но погрешность должна быть под контролем.

Вводная

Представим, что вы ведете базу входящих звонков. Стоит АОН, вы запоминаете номер в базе. Нужно оценить примерно к-во уникальных номеров телефонов, с которых принимались звонки.

Предполагаем, что цифры в телефонных номерах не зависят друг от друга и распределены равномерно и каждый номер состоит из одинакового к-ва цифр!!

Пример

Пришли звонки от:

```
095-17-29

127-13-25

095-17-29

326-82-98

485-23-19

095-17-29

...
```

Вопрос 1: сколько примерно уникальных номеров мы получим всего, если есть только один номер (095-17-29), который начинается ровно с одного нуля (а двух и более нулей нигде в начале номера нет)? А если какой-то номер начинается с двух нулей (и больше, чем двух в начале нет)? А если с трёх?

Объяснение.

Схлопываем одинаковые номера в один:

```
095-17-29

127-13-25

326-82-98

485-23-19

...
```

Очевидно, что номеров, начинающихся на один 0 - 1/10 часть от всех. На 00 - 1/100 часть и т.д.

Вопрос 2: нужно ли смотреть до конца все номера для общей оценки к-ва?

Вопрос 3: нужно ли заводить какие-то счетчики для оценки к-ва?

Не нужно, нужно смотреть только их начало - на длину начальной цепочки нулей. Просто хранить максимальную длину цепочки из нулей, и всё!! К-во элементов оценивается как 10^m 10^m, где m m - длина цепочки.

Алгоритм HyperLogLog

- не нужна хэш-таблица;
- исходные данные можно не хранить, а обрабатывать и сразу удалять: делается один проход по данным;
- легко параллелится;
- требование к занимаемому месту всего $log(log(N))$ $log(log(N))$, поэтому называется логлог;
- позволяет оценить к-во уникальных элементов по самой длинной цепочке нулей;
- прекрасные параметры по соотношению погрешность-длина датасета-занимаемое место: ошибка всего 2% при к-ве уникальных элементов более 10^9 10^9 с использованием всего лишь 1.5 кбайт памяти.

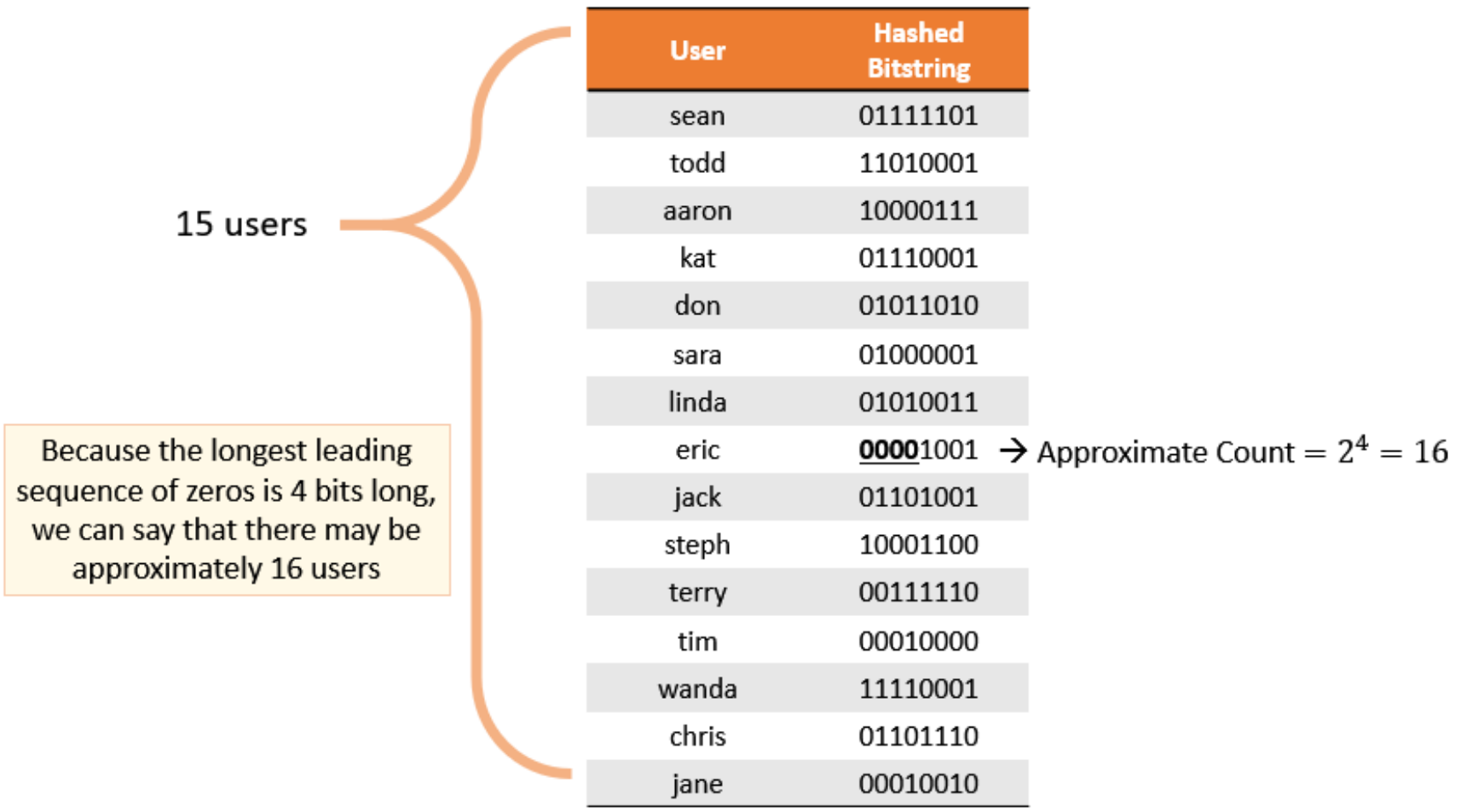
Подсчитаем число пользователей ресурса

Рассмотрим работу алгоритма на примере задачи с подсчетом числа пользователей.

- когда пользователь заходит на ресурс, алгоритм HyperLogLog хэширует id пользователя в битовую строку;
- при условии равномерного распределения мы можем сделать вывод, что примерно половина

битовых строк начинается с 0, а другая половина начинается с 1;

- аналогично, примерно $\frac{1}{4}\frac{1}{4}$ битовых строк начинается с 00 и примерно $\frac{1}{8}\frac{1}{8}$ с 000;
- обобщим: примерно 1 из каждых $2^z 2^z$ равномерно-распределенных битовых строк начинается с последовательности из zz нулей;
- предположим, что некоторый момент времени мы хэшируем id всех новых активных пользователей, и обнаруживаем, что максимальная последовательность ведущих нулей 0000;
- вероятность этого на любой случайной битовой строке $\frac{1}{16}\frac{1}{16}$;
- то есть нам нужно в среднем 16 случайных битовых строк, чтобы найти ту, которая начинается с 0000;
- выясняем, что у нас примерно 16 посетителей.



User	Hashed Bitstring
sean	01111101
todd	11010001
aaron	10000111
kat	01110001
don	01011010
sara	01000001
linda	01010011
eric	**0000**1001
jack	01101001
steph	10001100
terry	00111110
tim	00010000
wanda	11110001
chris	01101110
jane	00010010

Идея алгоритма:

- обозначим максимальное количество последовательных начальных нулей через n;
- тогда количество уникальных элементов в наборе можно оценить, как $2^n 2^n$;

То есть, если максимум один начальный бит равен нулю, тогда количество уникальных элементов, в среднем, равно 2; если максимум три начальных бита равны нулю, в среднем, мы можем ожидать 8 уникальных элементов и т.д.

Недостатки этого подхода:

- оценка слишком грубая - всегда $2^n 2^n$;
- если выбранная хэш-функция не дает равномерное распределение, алгоритм теряет точность, поскольку не выполняются базовые допущения, лежащие в его основе: если найдется один выброс высокочастотного элемент с префиксом 0, он может все испортить.

Как улучшить алгоритм

ваши предложения

- использовать хэш-функцию, дающую качественное равномерное распределение, например MurmurHash3.

Требование к HyperLogLog: "Предполагается, что хэш-функция работает таким образом, что результат близок к равномерному распределению, то есть биты хэша независимы и каждый бит имеет вероятность 0.5 (т.е. 0 и 1 равновероятны).".

- можно использовать несколько хэш-функций, считать максимум для каждой и усреднять его;
- в алгоритме используется другой подход - усреднение длин цепочек из нулей по подмножествам исходного мультисета (хэширование относительно дорого).

Описание алгоритма

- хэшируем данные;
- делим входящие хэши на 2 части - условно номер корзины и остальное, что ведет себя по вышеописанным законам как часть хэша;
- заводим счетчики (регистры) M, по числу корзин, инициализируем их нулями;
- разделяем хеши на m подгрупп (корзин) на основании их начальных битов (1-я часть хэша) и записываем в регистры максимальную длину цепочки начальных нулей из 2-й части - то же, что раньше;

01 111101 --> 1-я корзина, пока что самая длинная цепочка начальных нулей - 0;

11 010001 --> 3-я корзина, пока что самая длинная цепочка начальных нулей - 1;

10 000111 --> 2-я корзина, пока что самая длинная цепочка начальных нулей - 3;

01 110001 --> снова 1-я корзина, пока что самая длинная цепочка начальных нулей - 0;

01 011010 --> снова 1-я корзина, пока что самая длинная цепочка начальных нулей - 1;

01 000001 --> снова 1-я корзина, длина 5;

01 010011 --> снова 1-я, длина 5 остается;

00 001001 --> 0-я корзина, длина 2.

и т.д.

Результат по всему множеству:

M = {0, 2}, {1, 5}, {2, 3}, {3, 1}.

- вычисляем среднее гармоническое по корзинам - Z.

Что такое "среднее гармоническое"?

Вопрос: кто знает, что это такое и что оно дает?

Среднее гармоническое - это один из способов, которым можно вычислять «среднюю» величину некоторого набора чисел.

Пусть даны положительные числа $x_1, \ldots, x_n$ $x_1, \ldots, x_n$, тогда их средним гармоническим будет такое число H H, что

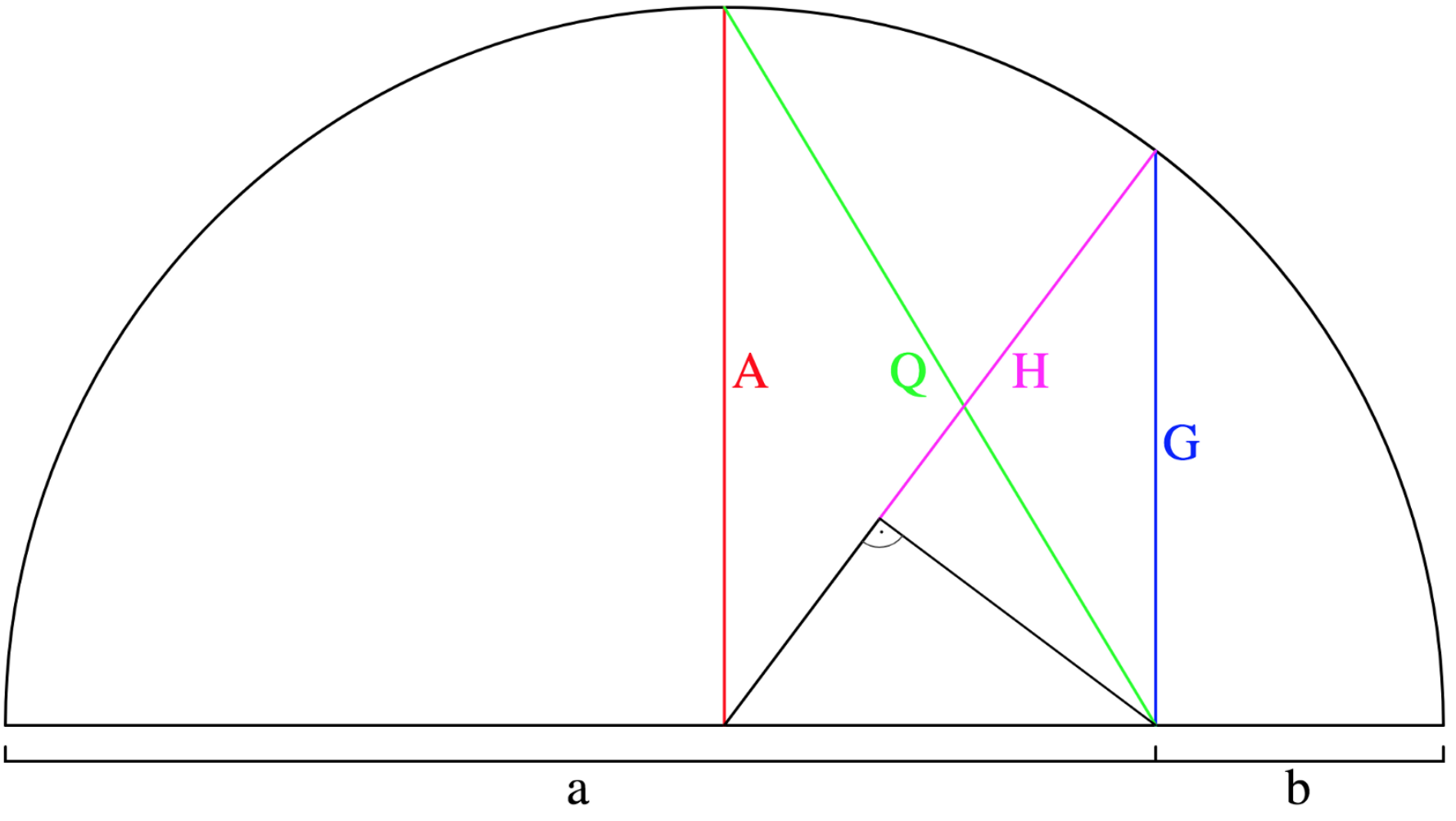
$$\frac{n}{H} = \frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}$$

$$\frac{n}{H} = \frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}$$

Формула:

$$H(x_1, \ldots, x_n) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}} = \frac{1}{\frac{1}{n}\sum_{i=1}^{n} \frac{1}{x_i}}$$

$$H(x_1, \ldots, x_n) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}} = \frac{1}{\frac{1}{n}\sum_{i=1}^{n} \frac{1}{x_i}}$$



- A - среднее арифметическое a a и b b;
- G - среднее геометрическое;
- Q - среднее квадратичное;
- H - среднее гармоническое.

Вычисляем среднее гармоническое Z (умножения на m нет)

$$Z = \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$$

$$Z = \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$$

Вычислим для нашего случая.

$$Z = (2^{-2} + 2^{-5} + 2^{-3} + 2^{-1})^{-1} = \left(\tfrac{1}{4} + \tfrac{1}{32} + \tfrac{1}{8} + \tfrac{1}{2}\right)^{-1} = \tfrac{32}{29} \approx 1.103$$

$$Z = (2^{-2} + 2^{-5} + 2^{-3} + 2^{-1})^{-1} = \left(\frac{1}{4} + \frac{1}{32} + \frac{1}{8} + \frac{1}{2}\right)^{-1} = \frac{32}{29} \approx 1.103$$

Если будет выброс в какой-то корзине, он усреднится и мало повлияет на результат.

Вычисляем оценку кардинальности мультисета E

$E = \alpha_m m^2 Z$ $E = \alpha_m m^2 Z$, где

$$\alpha_m = \left(m \int_0^{\infty} \left(log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

$$\alpha_m = \left(m \int_0^{\infty} \left(log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

Коэффициент $\alpha_m \alpha_m$ аппроксимируется следующим образом:

- для m = 16 $\alpha_m \alpha_m \approx \approx$ 0.673;
- для m = 32 $\alpha_m \alpha_m \approx \approx$ 0.697;
- для m = 64 $\alpha_m \alpha_m \approx \approx$ 0.709;

и т.д.

Вычислим для нашего случая.

$$E = 0.673 * 16 * 1.103 = 11.87$$ $E = 0.673 * 16 * 1.103 = 11.87$

Пример использования алгоритма (Java)

- добавить библиотеку hll

In [ ]:

```
<dependency>
    <groupId>net.agkn</groupId>
    <artifactId>hll</artifactId>
    <version>1.6.0</version>
</dependency>
```

- создать объект HLL, конструктор принимает два аргумента:
  - log2m - количество регистров, используемых HLL (то есть m)
  - regwidth - число бит, используемых каждым регистром

Чем большую точность мы хотим получить, тем в большее значение необходимо устанавливать эти параметры.

Например, нам нужно посчитать уникальные значения в датасете из 100 млн объектов.

Установим параметр log2m в 14, а regwidth в 5 - разумные значения для этого размера данных.

Перед вставкой в HLL каждый элемент необходимо захэшировать.

Для этого используем Hashing.murmur3_128() из библиотеки Guava (есть в депенденсах к hll). Этот хэш и точный, и быстрый.

In [ ]:

```
HashFunction hashFunction = Hashing.murmur3_128();
long numberOfElements = 100_000_000;
long toleratedDifference = 1_000_000;
HLL = new HLL(14, 5);
```

Эти параметры дадут нам процент ошибок ниже 1% (1000000 элементов).

Вставим эти 100 млн элементов:

In [ ]:

```
LongStream.range(0, numberOfElements).forEach(element -> {
    long hashedValue = hashFunction.newHasher().putLong(element).hash().asLong();
    hll.addRaw(hashedValue);
  }
);
```

Наконец, мы можем протестировать, что cardinality, которую вернул HLL, находится в пределах нашей желаемой погрешности:

In [ ]:

```
long cardinality = hll.cardinality();
assertThat(cardinality)
  .isCloseTo(numberOfElements, Offset.offset(toleratedDifference));
```

Можем вычислить, сколько памяти занимает наш HLL - примерно 81000 бит или 8100 байт. То есть оценка размера уникальных элементов из 100 млн множества использует всего 8100 байт памяти.

Если сравнить с обычной имплементацией, нам нужен будет сет из 100 млн длинных целых, что займет 100 000 000 * 8 байт = 800 000 000 байт.

Разница 8100 байт и 800 МБайт.

Чем больше батасет, тем больше разница.

Пример реализации алгоритма (python)

https://pypi.org/project/hyperloglog/ (https://pypi.org/project/hyperloglog/)

Посмотрим, если останется время.

Применение алгоритма

Используют:

- Redis Labs (для уменьшения размера storage в Redis): https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/ (https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/)
- Google (для подсчета числа запросов в день): https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf (https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf)
- Google (для подсчета числа результатов BigQuery): https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog (https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog)
- Elastic search (для подсчета количества уникальных значений в поле): https://www.elastic.co/blog/count-elasticsearch (https://www.elastic.co/blog/count-elasticsearch)
- AWS Redshift (для функции COUNT DISTINCT): https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html (https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html)
- Reddit (для подсчета числа комментариев к посту): https://redditblog.com/2017/05/24/view-counting-at-reddit/ (https://redditblog.com/2017/05/24/view-counting-at-reddit/)

Алгоритм Count-Min Sketch

Count-Min Sketch - семейство эффективных по памяти структур данных, которые позволяют оценить частоту появления конкретных элементов, найти top K частотных элементов, сделать выборку по диапазону элементов (найти сумму частот элементов внутри какого-либо диапазона).

Задача

Есть набор значений с дубликатами (мультисет). Например, у нас видеостриминговый сервис и нужно оценить частоту скачивания (число дубликатов во множестве) отдельных или всех видео.

{
"Hello by Adele": 2,300,000,000
"What are Blockchain Smart Contracts?": 32,000
"History of Japan": 39,000,000
}

Оценка для нечастотных значений может быть неточной, но популярные видео должны быть оценены более-менее точно.

Прямое решение

Будем хранить счетчики в хэш-таблице. Пусть у нас 4 млрд видео, каждое видео имеет 32-битный уникальный id (это минимум для адресации такого к-ва видео).

Тогда нам понадобится 16 Гб только чтобы хранить id каждого видео. Для счетчиков нужно еще 16 Гб. Итого 32 Гб. Уже проблема с размещением в RAM. А если хранить перекрестные ссылки на видео или метаинформацию (регион, язык, год, длительность, жанр) для продвинутой аналитики?

Нужен кластер Hadoop или Spark?

Не обязательно!!!

Sketches

Что такое Sketches ("наброски", "эскизы", "зарисовки", "отрывки")?

Это класс структур данных, которые могут "отображать на себя" огромные датасеты с сублогарифмическими или даже константными требованиями по месту.

MinHash и **SimHash**, которые мы проходили на прошлом занятии - это скетчи. Отображают большой документ в множество хэшей либо в единственный хэш.

Вопрос: требования по месту у них были какие?

Для скетчей не нужно хранить полные данные. Нужно использовать трюки для сжатия данных -> это приводит к неточным "зарисовкам".

Под скетчи можно резервировать сколько угодно места, но если мультисет вырастет, скетч скорее всего вернет неточные результаты.

Трейд-офф: размер vs точность.

Важно найти верный баланс между размером и точностью.

Как пример: картина ("Джоконда") и ее зарисовка. Чем больше усилий тратится на зарисовку, чем более она похожа на оригинал. Также требуется больше места для хранения. Как и в реальности сложную картину труднее воспроизвести, чем простую. Большой сет требует больше места для скетча, чем маленький.

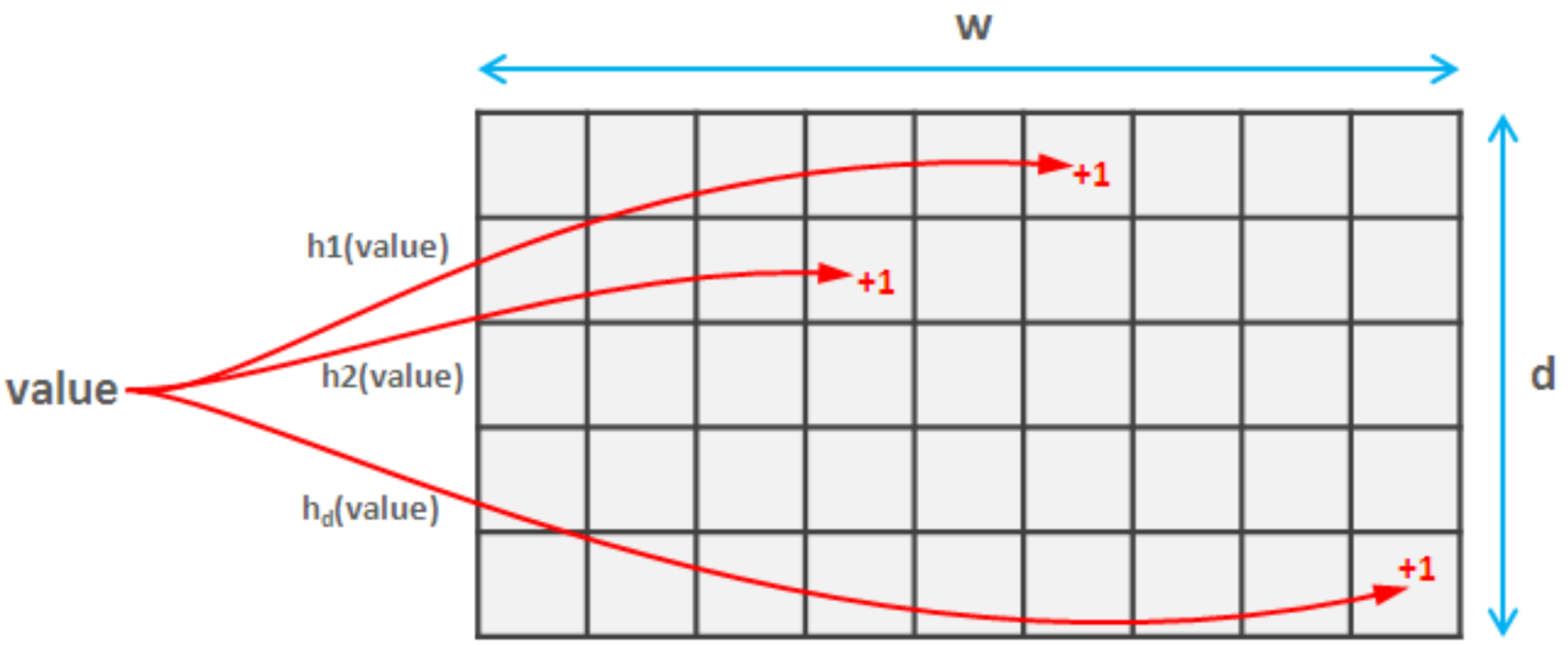
Задача уменьшения размерности в ML - это аналог скетчинга.

Нормально, когда приложение, работающее с большими данными, не требует большой точности.

Идея алгоритма

Count-Min sketch - это просто двумерный массив ($d\,d$ * $w\,w$) целочисленных счетчиков.

- когда приходит новое значение, оно мапится в одну позицию в каждую из $d\,d$ строк используя d d различных независимых хэш-функций;
- счетчики по каждой позиции инкрементируются.



Запрос значений счетчиков

Чтобы запросить значение счетчика (к-во просмотров по видео) вычисляем минимум из значений всех счетчиков по этому видео.

$count(x)\ count(x) = min(h_i(x))\ min(h_i(x))$, где $h_i\,h_i$ - i-тая хэш-функция

Почему берется минимум, а не максимум?

Ваши варианты?

Потому что ошибка у нас всегда в сторону увеличения (каждое вхождения значения увеличивает счетчик, но коллизии могут добавить дополнительный плюс).

Пример

Хэш-функции здесь по горизонтали.

Имеем 3-колоночный, 4-строчный Count-Min Sketch.

Самый первый пользователь вашей платформы хочет посмотреть видео "History of Japan".

Когда id этого видео хэшируется тремя хэшами, получаем 1, 2 и 1, соответственно мы должны увеличить счетчики в позициях (1,1), (2,2) и (1,3).

Hash 1	Hash 2	Hash 3
0	0	0
0	0	0
0	0	0
0	0	0

Hash 1	Hash 2	Hash 3
1	0	1
0	1	0
0	0	0
0	0	0

put(Japan)

hash(Japan) = (hash1(Japan), hash2(Japan), hash3(Japan)) = (1,2,1)

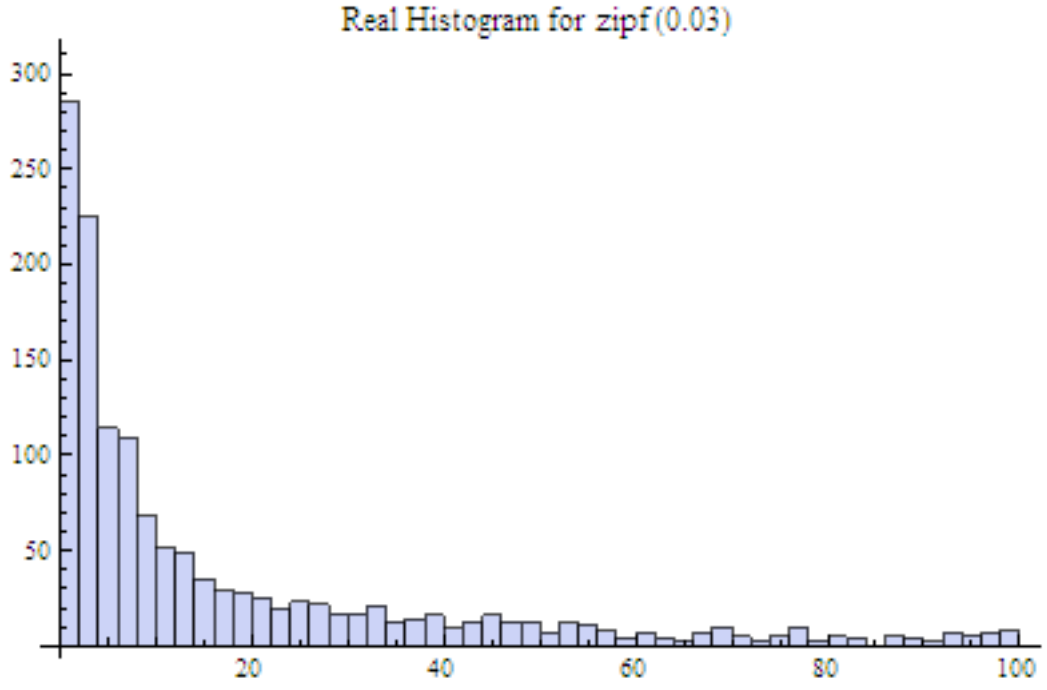
Оценка точности алгоритма

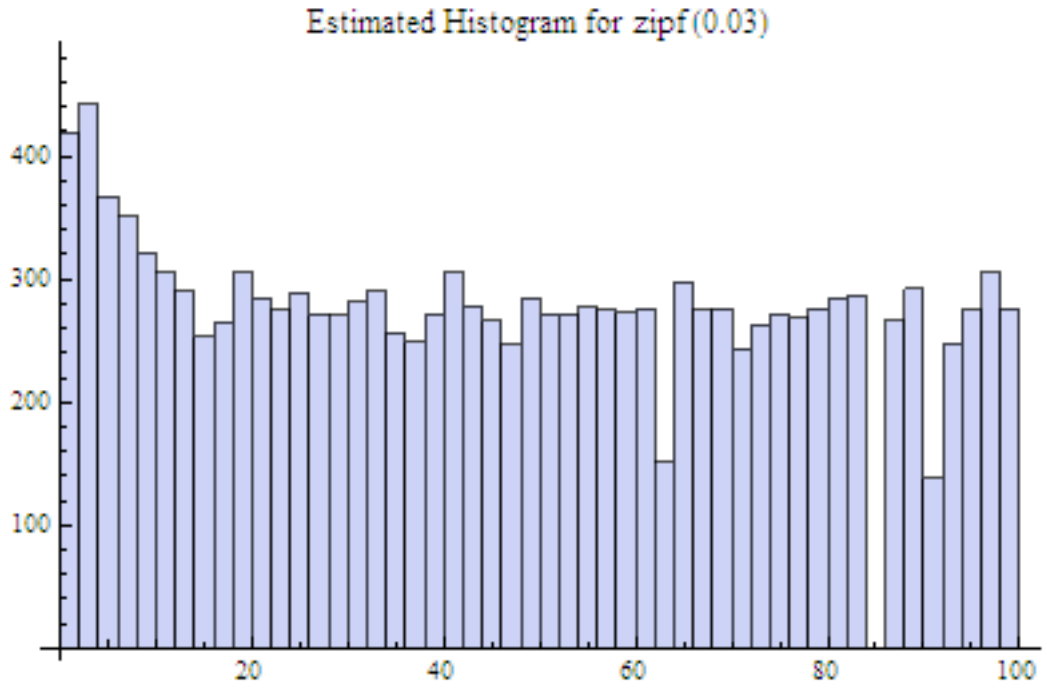
Точность алгоритма Count-Min sketch зависит от соотношения размера скетча и общего числа объектов. Это означает, что Count-Min sketch обеспечивает экономию памяти только для очень случайных данных, то есть вероятности событий (объектов) очень сильно скачут.

Две диаграммы показывают результаты Count-Min sketch размера 3×64, т.е. всего 192 счетчика.

В первом случае в sketch поданы относительно случайный набор данных в 10k элементов, среди них около 8500 уникальных значений (частоты элементов подчиняются распределению Ципфа, которое, например, соответствует распределению слов в текстах на естественных языках).

Настоящая гистограмма (показаны только самые частотные элементы, диаграмма имеет "длинный плоский хвост" справа, который отрезан на этой картинке) и гистограмма, восстановленная по скетчу показаны ниже:

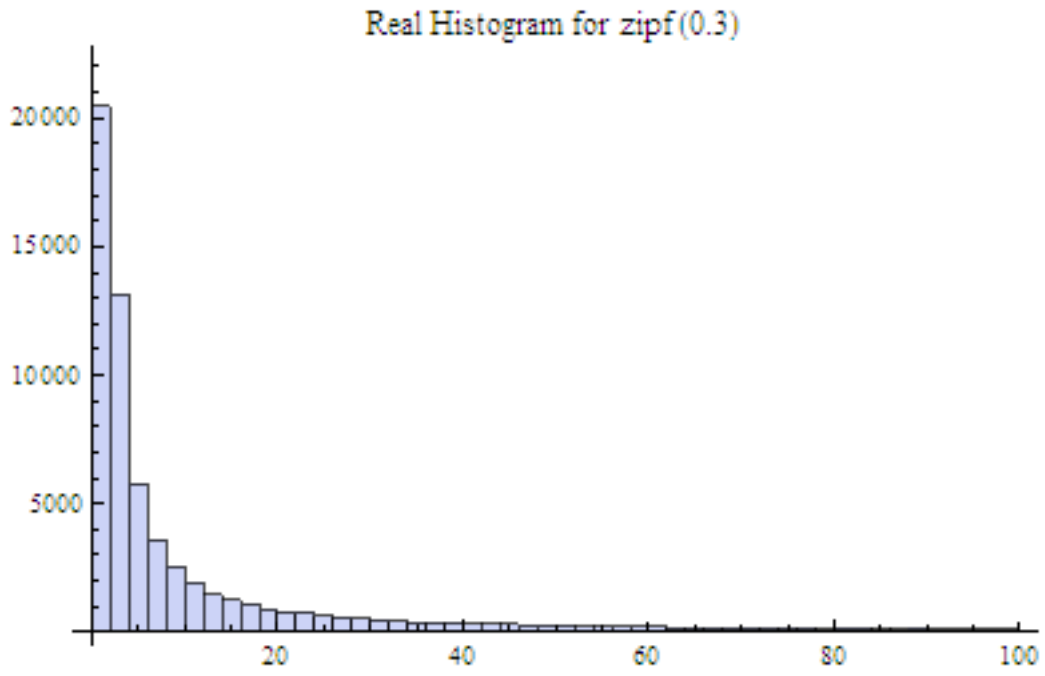


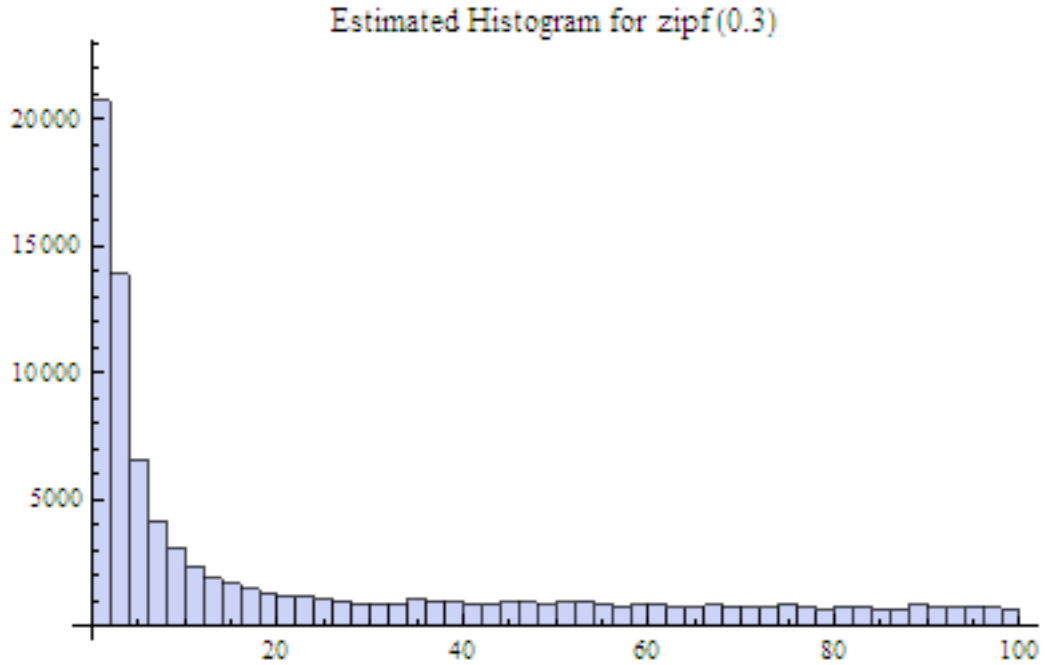


Очевидно, что Count-Min sketch не может отследить частоты 8500 элементов с использованием всего 192 счетчиков в случае малого отклонения частот, поэтому гистограмма получилась очень неточной.

Во втором случае в скетч были поданы относительно случайные данные в размере 80k элементов, среди которых также около 8500 уникальных.

Реальная гистограмма и гистограмма по скетчу показаны ниже:





Очевидно, что результат в данном случае более точный.

Оценка точности:

$estimation\ error\ \ \varepsilon \leq 2n/w$ $with\ probability\ \delta = 1 - (1/2)^d$	n – total count of registered events w – sketch width d – sketch height (aka depth)

Домашнее задание

- взять готовую реализацию одного из алгоритмов: **Bloom Filter**, **MinHash**, **SimHash**, **HyperLogLog** или **Count-Min Sketch** для вашего языка программирования для ее изучения;
- изучить код реализации;
- вариант повышенной сложности (опционально): самим реализовать этот алгоритм;
- найти большой датасет, подходящий для выбранного алгоритма;
- применить ее для решения практической задачи: например, определить, принаджежит ли элемент множеству, подсчитать число уникальных элементов в большом массиве данных или подсчитать числа вхождений каждого элемента в большой массив данных;
- оценить точность реализации (% ошибок, false positives и т.д.) с помощью тестов;
- выложить ваш код вместе с датасетом;
- (опционально): сделать вышеперечисленное для еще одного алгоритма из списка.
- (опционально): сравнить между собой "парные" алгоритмы, если вы выбрали их (напр, MinHash и SimHash).

Пример. Если вы выбрали MinHash и/или SimHash:

- разобрать пример реализации алгоритма MinHash по ссылке https://github.com/chrisjmccormick/MinHash (https://github.com/chrisjmccormick/MinHash) (можно также взять любую другую готовую реализацию алгоритма);
- (опционально) реализовать алгоритм MinHash на вашем языке программирования; если это python - попробовать улучшить пример по ссылке (отрефакторить, улучшить скорость работы, уменьшить требуемую память и т.д.);
- применить алгоритм для решения практической задачи: найти похожие объекты в большом датасете (1000 статей из интернета). Примеры статей есть в проекте https://github.com/chrisjmccormick/MinHash/tree/master/data (https://github.com/chrisjmccormick/MinHash/tree/master/data);
- (опционально) реализовать алгоритм SimHash на вашем языке программирования (как отправную точку можно взять реализацию https://github.com/seomoz/simhash-py (https://github.com/seomoz/simhash-py) и https://github.com/seomoz/simhash-cpp) (https://github.com/seomoz/simhash-cpp));
- (опционально) сравнить работу ваших алгоритмов MinHash и SimHash по скорости, точности работы и занимаемой памяти на любом датасете (1000 статей из интернета).

Литература и ссылки

HyperLogLog

Математика

HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm Philippe Flajolet, Éric Fusy, Olivier Gandouet, Frédéric Meunier http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf (http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf)

Об алгоритме и пример реализации на C++

https://habr.com/ru/post/119852/ (https://habr.com/ru/post/119852/)

Реализация

- Python https://pypi.org/project/hyperloglog/ (https://pypi.org/project/hyperloglog/)
- Java https://github.com/prasanthj/hyperloglog (https://github.com/prasanthj/hyperloglog)
- Java/Scala - сравнение реализаций - http://koff.io/posts/comparison-of-hll/ (http://koff.io/posts/comparison-of-hll/)
- C# (Microsoft): https://github.com/Microsoft/CardinalityEstimation (https://github.com/Microsoft/CardinalityEstimation)

Использование

- Redis Labs: https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/ (https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/)
- Google: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf (https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf), https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog (https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog)
- Elastic search: https://www.elastic.co/blog/count-elasticsearch (https://www.elastic.co/blog/count-elasticsearch)
- AWS Redshift: https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html (https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html)
- Reddit: https://redditblog.com/2017/05/24/view-counting-at-reddit/ (https://redditblog.com/2017/05/24/view-counting-at-reddit/)

Count-Min Sketch

Математика

An Improved Data Stream Summary: The Count-Min Sketch and its Applications Graham Cormode, S. Muthukrishnan http://www.eecs.harvard.edu/~michaelm/CS222/countmin.pdf (http://www.eecs.harvard.edu/~michaelm/CS222/countmin.pdf)

Реализация

- C, Python: https://github.com/barrust/count-min-sketch (https://github.com/barrust/count-min-sketch)

Обо всех рассмотренных структурах данных

https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/ (https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/)

Спасибо
за внимание!


OTUS
ОНЛАЙН-ОБРАЗОВАНИЕ

OTUS

Алгоритмы HyperLogLog и Count-min Sketch

Михаил Горшков
разработчик

HyperLogLog & Count-Min Sketch

- работаем с мультисетами — множествами, где каждый уникальный элемент может появиться более, чем 1 раз:

{2, 15, 1, 1, 36, 2}

- актуально для больших данных;
- HyperLogLog может определить (приблизительно) сколько уникальных элементов имеет мультисет;
- Count-Min Sketch может определить (приблизительно), сколько раз встречается в мультисете каждый элемент;

Постановка задач

Решаем 2 задачи в "точном" виде.

Дан мультисет, нужно найти:

- количество уникальных элементов (англ. *cardinality*);
- сколько раз встречается каждый элемент.

Для нашего примера:

{2, 15, 1, 1, 36, 2}

ответы:

- 4 уникальных элемента;
- 2 - 2 раза, 15 - 1 раз, 1 - 2 раза, 36 - 1 раз.

Как решить задачу? Нужен алгоритм.

Ваши варианты

- использовать массив, отсортировать; (?)
- использовать вместо массива специализированные контейнеры.

Рассмотрим мультисеты в C++ (для остальных есть аналоги, но не в ядре языка).

In [ ]:

std::multiset

std::multiset - контейнер, который хранит сортированный набор объектов типа Key. В отличие от std::set, допустимы одинаковые значения ключей. Обычно реализован как бинарное дерево поиска (RB-tree).

Задачи:

- как определить, сколько уникальных элементов содержит multiset, если он реализован как бинарное дерево поиска?
- как определить, сколько раз встречается в multiset-е каждый элемент?

Ваши варианты??

In [ ]:

std::unordered_multiset

std::unordered_multiset - hash-based контейнер, который хранит набор объектов типа Key (возможно, не уникальных). Элементы помещаются в bucket-ы в зависимости от значения хэш-функции.

Задачи:

- как определить, сколько уникальных элементов содержит unordered_multiset?
- как определить, сколько раз встречается в unordered_multiset-е каждый элемент?

Ваши варианты??

HyperLogLog

Задача - подсчитать (приблизительно) число уникальных элементов в массиве

Зачем это нужно?

- обнаружение распространения вирусов, обнаружение сетевых атак (напр., DDOS - малое к-во уникальных и большое к-во одинаковых паттернов в трафике подозрительно);
- обнаружение спама - также большое к-во одинаковых паттернов подозрительно;
- data mining - нахождение к-ва одинаковых самплов данных;

Ограничения области применения

- хорошо (точно) работает на больших объемах данных;
- плохо на малых объемах данных (для них лучше применять традиционные алгоритмы).

Задача: необходимо подсчитать число уникальных пользователей, заходящих на ресурс в течение дня

Как это сделать? Ваши варианты

Использовать хэш-таблицу, добавляя пользователя в нее, когда он заходит на ресурс. Если пользователь входит с разных ip, девайсов, несколько раз, считаем это за 1 вход (пользователь должен залогиниться, учитываем login id). В конце дня считаем и сохраняем размер хэш-таблицы и записываем его в таблицу статистики. Очищаем хэш-таблицу, считаем следующий день и т.д.

Если у нас миллиарды пользователей каждый день, а мы храним user id как 32-битное целое число, понадобится 4 Гб, чтобы сохранить пользователей. Можно использовать битовую карту —каждый пользователь это бит—тогда все равно нужен 1 Гб памяти.

Хэш-таблица и битмап имеют требование месту O(n), место прямо пропорционально к-ву данных.

Другие метрики могут потребовать еще места.

Нас устроит не совсем точное значение, потому что предполагается, что к-во пользователей очень велико и оно постоянно увеличивается. Нужен +- порядок величины, но погрешность должна быть под контролем.

Вводная

Представим, что вы ведете базу входящих звонков. Стоит АОН, вы запоминаете номер в базе. Нужно оценить примерно к-во уникальных номеров телефонов, с которых принимались звонки.

Предполагаем, что цифры в телефонных номерах не зависят друг от друга и распределены равномерно и каждый номер состоит из одинакового к-ва цифр!!

Пример

Пришли звонки от:

```
095-17-29

127-13-25

095-17-29

326-82-98

485-23-19

095-17-29

...
```

Вопрос 1: сколько примерно уникальных номеров мы получим всего, если есть только один номер (095-17-29), который начинается ровно с одного нуля (а двух и более нулей нигде в начале номера нет)? А если какой-то номер начинается с двух нулей (и больше, чем двух в начале нет)? А если с трёх?

Объяснение.

Схлопываем одинаковые номера в один:

```
095-17-29

127-13-25

326-82-98

485-23-19

...
```

Очевидно, что номеров, начинающихся на один 0 - 1/10 часть от всех. На 00 - 1/100 часть и т.д.

Вопрос 2: нужно ли смотреть до конца все номера для общей оценки к-ва?

Вопрос 3: нужно ли заводить какие-то счетчики для оценки к-ва?

Не нужно, нужно смотреть только их начало - на длину начальной цепочки нулей. Просто хранить максимальную длину цепочки из нулей, и всё!! К-во элементов оценивается как 10^m, где m - длина цепочки.

Алгоритм HyperLogLog

- не нужна хэш-таблица;
- исходные данные можно не хранить, а обрабатывать и сразу удалять: делается один проход по данным;
- легко параллелится;
- требование к занимаемому месту всего $log(log(N))$, поэтому называется логлог;
- позволяет оценить к-во уникальных элементов по самой длинной цепочке нулей;
- прекрасные параметры по соотношению погрешность-длина датасета-занимаемое место: ошибка всего 2% при к-ве уникальных элементов более 10^9 с использованием всего лишь 1.5 кбайт памяти.

Подсчитаем число пользователей ресурса

Рассмотрим работу алгоритма на примере задачи с подсчетом числа пользователей.

- когда пользователь заходит на ресурс, алгоритм HyperLogLog хэширует id пользователя в битовую строку;
- при условии равномерного распределения мы можем сделать вывод, что примерно половина

битовых строк начинается с 0, а другая половина начинается с 1;

- аналогично, примерно $\frac{1}{4}$ битовых строк начинается с 00 и примерно $\frac{1}{8}$ с 000;
- обобщим: примерно 1 из каждых 2^z равномерно-распределенных битовых строк начинается с последовательности из z нулей;
- предположим, что некоторый момент времени мы хэшируем id всех новых активных пользователей, и обнаруживаем, что максимальная последовательность ведущих нулей 0000;
- вероятность этого на любой случайной битовой строке $\frac{1}{16}$;
- то есть нам нужно в среднем 16 случайных битовых строк, чтобы найти ту, которая начинается с 0000;
- выясняем, что у нас примерно 16 посетителей.

15 users

Because the longest leading sequence of zeros is 4 bits long, we can say that there may be approximately 16 users

User	Hashed Bitstring
sean	01111101
todd	11010001
aaron	10000111
kat	01110001
don	01011010
sara	01000001
linda	01010011
eric	**0000**1001 → Approximate Count $= 2^4 = 16$
jack	01101001
steph	10001100
terry	00111110
tim	00010000
wanda	11110001
chris	01101110
jane	00010010

Идея алгоритма:

- обозначим максимальное количество последовательных начальных нулей через n;
- тогда количество уникальных элементов в наборе можно оценить, как 2^n;

То есть, если максимум один начальный бит равен нулю, тогда количество уникальных элементов, в среднем, равно 2; если максимум три начальных бита равны нулю, в среднем, мы можем ожидать 8 уникальных элементов и т.д.

Недостатки этого подхода:

- оценка слишком грубая - всегда 2^n;
- если выбранная хэш-функция не дает равномерное распределение, алгоритм теряет точность, поскольку не выполняются базовые допущения, лежащие в его основе: если найдется один выброс высокочастотного элемент с префиксом 0, он может все испортить.

Как улучшить алгоритм

ваши предложения

- использовать хэш-функцию, дающую качественное равномерное распределение, например MurmurHash3.

Требование к HyperLogLog: "Предполагается, что хэш-функция работает таким образом, что результат близок к равномерному распределению, то есть биты хэша независимы и каждый бит имеет вероятность 0.5 (т.е. 0 и 1 равновероятны).".

- можно использовать несколько хэш-функций, считать максимум для каждой и усреднять его;
- в алгоритме используется другой подход - усреднение длин цепочек из нулей по подмножествам исходного мультисета (хэширование относительно дорого).

Описание алгоритма

- хэшируем данные;
- делим входящие хэши на 2 части - условно номер корзины и остальное, что ведет себя по вышеописанным законам как часть хэша;
- заводим счетчики (регистры) M, по числу корзин, инициализируем их нулями;
- разделяем хеши на m подгрупп (корзин) на основании их начальных битов (1-я часть хэша) и записываем в регистры максимальную длину цепочки начальных нулей из 2-й части - то же, что раньше;

01 111101 --> 1-я корзина, пока что самая длинная цепочка начальных нулей - 0;

11 010001 --> 3-я корзина, пока что самая длинная цепочка начальных нулей - 1;

10 000111 --> 2-я корзина, пока что самая длинная цепочка начальных нулей - 3;

01 110001 --> снова 1-я корзина, пока что самая длинная цепочка начальных нулей - 0;

01 011010 --> снова 1-я корзина, пока что самая длинная цепочка начальных нулей - 1;

01 000001 --> снова 1-я корзина, длина 5;

01 010011 --> снова 1-я, длина 5 остается;

00 001001 --> 0-я корзина, длина 2.

и т.д.

Результат по всему множеству:

M = {0, 2}, {1, 5}, {2, 3}, {3, 1}.

- вычисляем среднее гармоническое по корзинам - Z.

Что такое "среднее гармоническое"?

Вопрос: кто знает, что это такое и что оно дает?

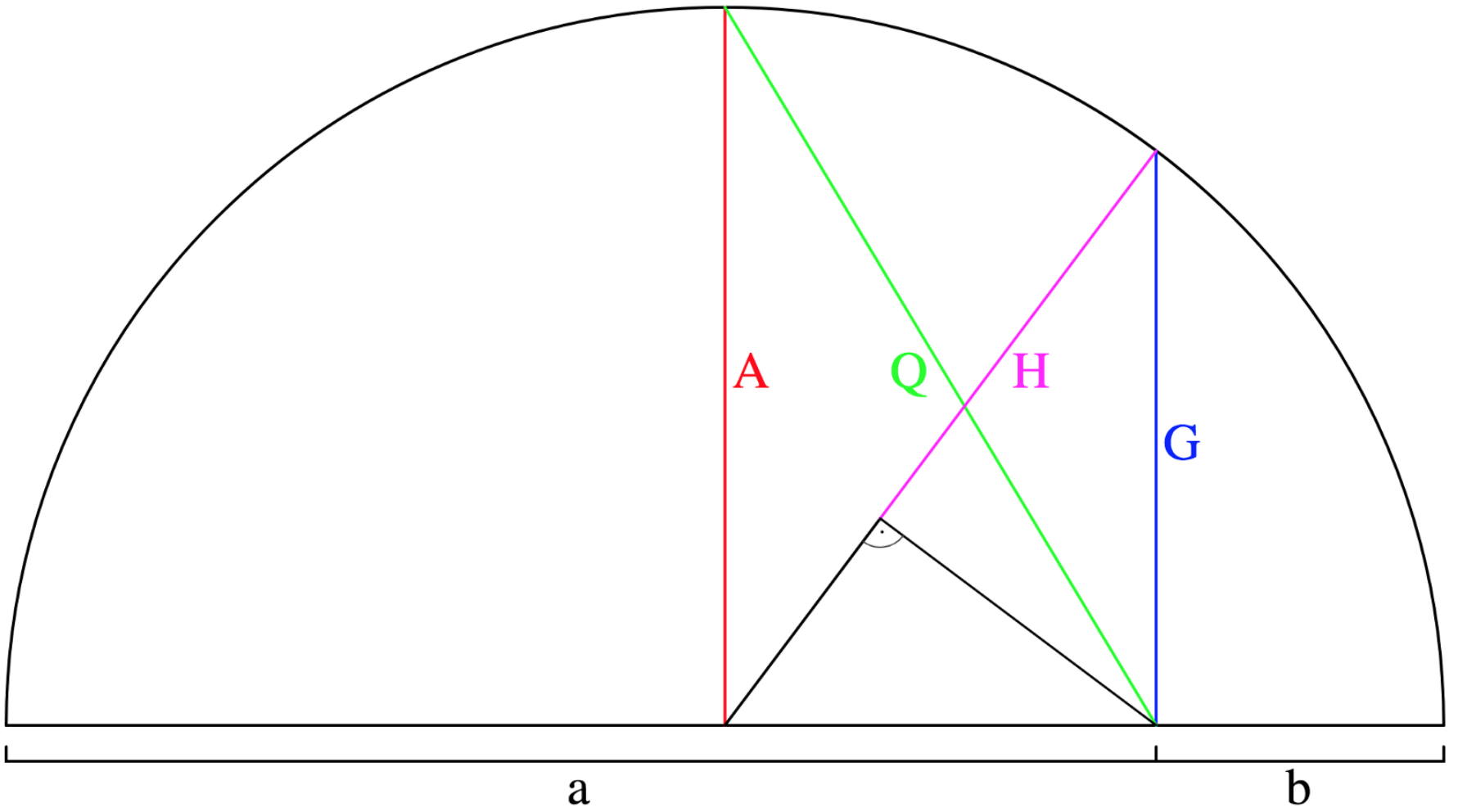
Среднее гармоническое - это один из способов, которым можно вычислять «среднюю» величину некоторого набора чисел.

Пусть даны положительные числа $x_1, \ldots, x_n$, тогда их средним гармоническим будет такое число H, что

$$\frac{n}{H} = \frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}$$

Формула:

$$H(x_1, \ldots, x_n) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}} = \frac{1}{\frac{1}{n}\sum_{i=1}^{n} \frac{1}{x_i}}$$



- A - среднее арифметическое a и b;
- G - среднее геометрическое;
- Q - среднее квадратичное;
- H - среднее гармоническое.

Вычисляем среднее гармоническое Z (умножения на m нет)

$$Z = \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1}$$

Вычислим для нашего случая.

$$Z = (2^{-2} + 2^{-5} + 2^{-3} + 2^{-1})^{-1} = \left(\frac{1}{4} + \frac{1}{32} + \frac{1}{8} + \frac{1}{2} \right)^{-1} = \frac{32}{29} \approx 1.103$$

Если будет выброс в какой-то корзине, он усреднится и мало повлияет на результат.

Вычисляем оценку кардинальности мультисета E

$E = \alpha_m m^2 Z$, где

$$\alpha_m = \left(m \int_0^\infty \left(log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

Коэффициент α_m аппроксимируется следующим образом:

- для m = 16 $\alpha_m \approx 0.673$;
- для m = 32 $\alpha_m \approx 0.697$;
- для m = 64 $\alpha_m \approx 0.709$;

и т.д.

Вычислим для нашего случая.

$E = 0.673 * 16 * 1.103 = 11.87$

Пример использования алгоритма (Java)

- добавить библиотеку hll

In [ ]:

- создать объект HLL, конструктор принимает два аргумента:
  - log2m - количество регистров, используемых HLL (то есть m)
  - regwidth - число бит, используемых каждым регистром

Чем большую точность мы хотим получить, тем в большее значение необходимо устанавливать эти параметры.

Например, нам нужно посчитать уникальные значения в датасете из 100 млн объектов.

Установим параметр log2m в 14, а regwidth в 5 - разумные значения для этого размера данных.

Перед вставкой в HLL каждый элемент необходимо захэшировать.

Для этого используем Hashing.murmur3_128() из библиотеки Guava (есть в депенденсах к hll). Этот хэш и точный, и быстрый.

In [ ]:

Эти параметры дадут нам процент ошибок ниже 1% (1000000 элементов).

Вставим эти 100 млн элементов:

In [ ]:

Наконец, мы можем протестировать, что cardinality, которую вернул HLL, находится в пределах нашей желаемой погрешности:

In [ ]:

Можем вычислить, сколько памяти занимает наш HLL - примерно 81000 бит или 8100 байт. То есть оценка размера уникальных элементов из 100 млн множества использует всего 8100 байт памяти.

Если сравнить с обычной имплементацией, нам нужен будет сет из 100 млн длинных целых, что займет 100 000 000 * 8 байт = 800 000 000 байт.

Разница 8100 байт и 800 МБайт.

Чем больше батасет, тем больше разница.

Пример реализации алгоритма (python)

https://pypi.org/project/hyperloglog/ (https://pypi.org/project/hyperloglog/)

Посмотрим, если останется время.

Применение алгоритма

Используют:

- Redis Labs (для уменьшения размера storage в Redis): https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/ (https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/)
- Google (для подсчета числа запросов в день): https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf (https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf)
- Google (для подсчета числа результатов BigQuery): https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog (https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog)
- Elastic search (для подсчета количества уникальных значений в поле): https://www.elastic.co/blog/count-elasticsearch (https://www.elastic.co/blog/count-elasticsearch)
- AWS Redshift (для функции COUNT DISTINCT): https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html (https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html)
- Reddit (для подсчета числа комментариев к посту): https://redditblog.com/2017/05/24/view-counting-at-reddit/ (https://redditblog.com/2017/05/24/view-counting-at-reddit/)

Алгоритм Count-Min Sketch

Count-Min Sketch - семейство эффективных по памяти структур данных, которые позволяют оценить частоту появления конкретных элементов, найти top K частотных элементов, сделать выборку по диапазону элементов (найти сумму частот элементов внутри какого-либо диапазона).

Задача

Есть набор значений с дубликатами (мультисет). Например, у нас видеостриминговый сервис и нужно оценить частоту скачивания (число дубликатов во множестве) отдельных или всех видео.

{
"Hello by Adele": 2,300,000,000
"What are Blockchain Smart Contracts?": 32,000
"History of Japan": 39,000,000
}

Оценка для нечастотных значений может быть неточной, но популярные видео должны быть оценены более-менее точно.

Прямое решение

Будем хранить счетчики в хэш-таблице. Пусть у нас 4 млрд видео, каждое видео имеет 32-битный уникальный id (это минимум для адресации такого к-ва видео).

Тогда нам понадобится 16 Гб только чтобы хранить id каждого видео. Для счетчиков нужно еще 16 Гб. Итого 32 Гб. Уже проблема с размещением в RAM. А если хранить перекрестные ссылки на видео или метаинформацию (регион, язык, год, длительность, жанр) для продвинутой аналитики?

Нужен кластер Hadoop или Spark?

Не обязательно!!!

Sketches

Что такое Sketches ("наброски", "эскизы", "зарисовки", "отрывки")?

Это класс структур данных, которые могут "отображать на себя" огромные датасеты с сублогарифмическими или даже константными требованиями по месту.

MinHash и **SimHash**, которые мы проходили на прошлом занятии - это скетчи. Отображают большой документ в множество хэшей либо в единственный хэш.

Вопрос: требования по месту у них были какие?

Для скетчей не нужно хранить полные данные. Нужно использовать трюки для сжатия данных -> это приводит к неточным "зарисовкам".

Под скетчи можно резервировать сколько угодно места, но если мультисет вырастет, скетч скорее всего вернет неточные результаты.

Трейд-офф: размер vs точность.

Важно найти верный баланс между размером и точностью.

Как пример: картина ("Джоконда") и ее зарисовка. Чем больше усилий тратится на зарисовку, чем более она похожа на оригинал. Также требуется больше места для хранения. Как и в реальности сложную картину труднее воспроизвести, чем простую. Большой сет требует больше места для скетча, чем маленький.

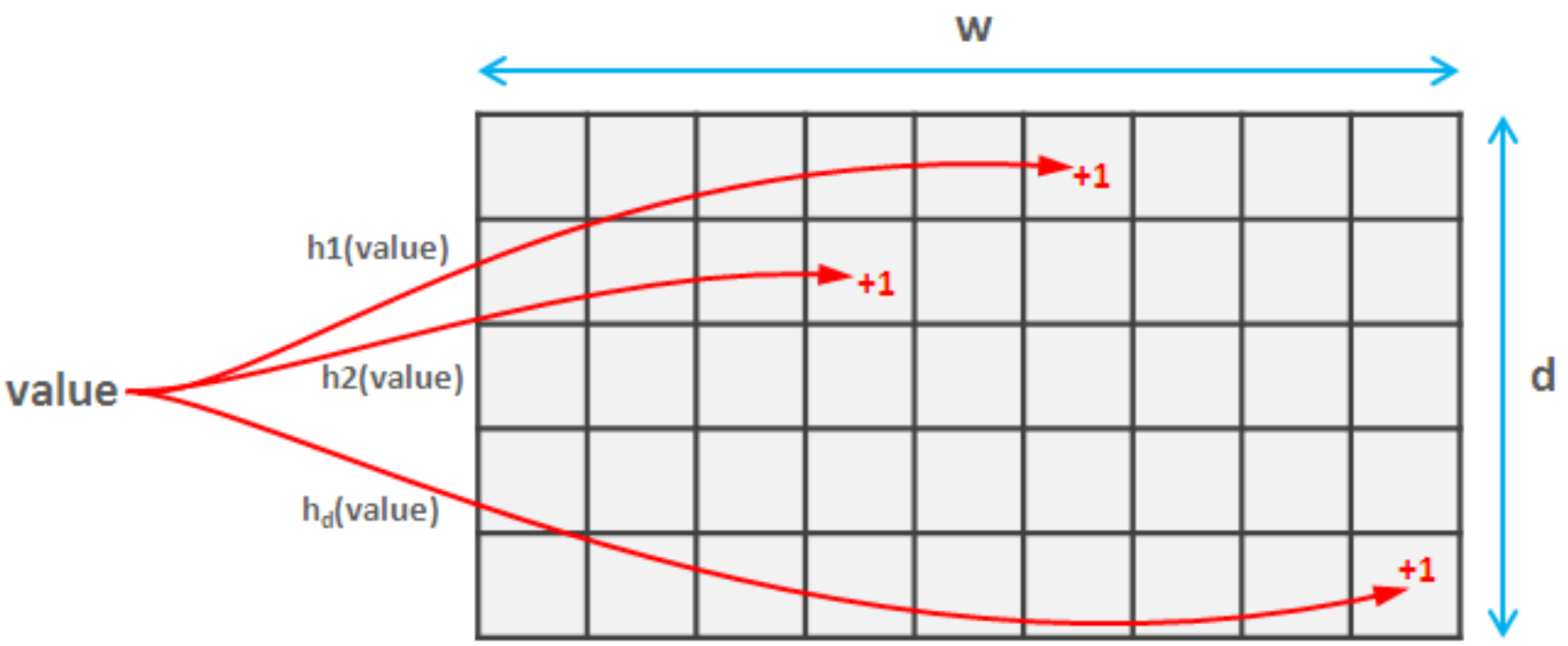
Задача уменьшения размерности в ML - это аналог скетчинга.

Нормально, когда приложение, работающее с большими данными, не требует большой точности.

Идея алгоритма

Count-Min sketch - это просто двумерный массив ($d * w$) целочисленных счетчиков.

- когда приходит новое значение, оно мапится в одну позицию в каждую из d строк используя d различных независимых хэш-функций;
- счетчики по каждой позиции инкрементируются.



Запрос значений счетчиков

Чтобы запросить значение счетчика (к-во просмотров по видео) вычисляем минимум из значений всех счетчиков по этому видео.

$count(x) = min(h_i(x))$, где h_i - i-тая хэш-функция

Почему берется минимум, а не максимум?

Ваши варианты?

Потому что ошибка у нас всегда в сторону увеличения (каждое вхождения значения увеличивает счетчик, но коллизии могут добавить дополнительный плюс).

Пример

Хэш-функции здесь по горизонтали.

Имеем 3-колоночный, 4-строчный Count-Min Sketch.

Самый первый пользователь вашей платформы хочет посмотреть видео "History of Japan".

Когда id этого видео хэшируется тремя хэшами, получаем 1, 2 и 1, соответственно мы должны увеличить счетчики в позициях (1,1), (2,2) и (1,3).

Hash 1	Hash 2	Hash 3
0	0	0
0	0	0
0	0	0
0	0	0

Hash 1	Hash 2	Hash 3
1	0	1
0	1	0
0	0	0
0	0	0

put(Japan)

hash(Japan) = (hash1(Japan), hash2(Japan), hash3(Japan)) = (1,2,1)

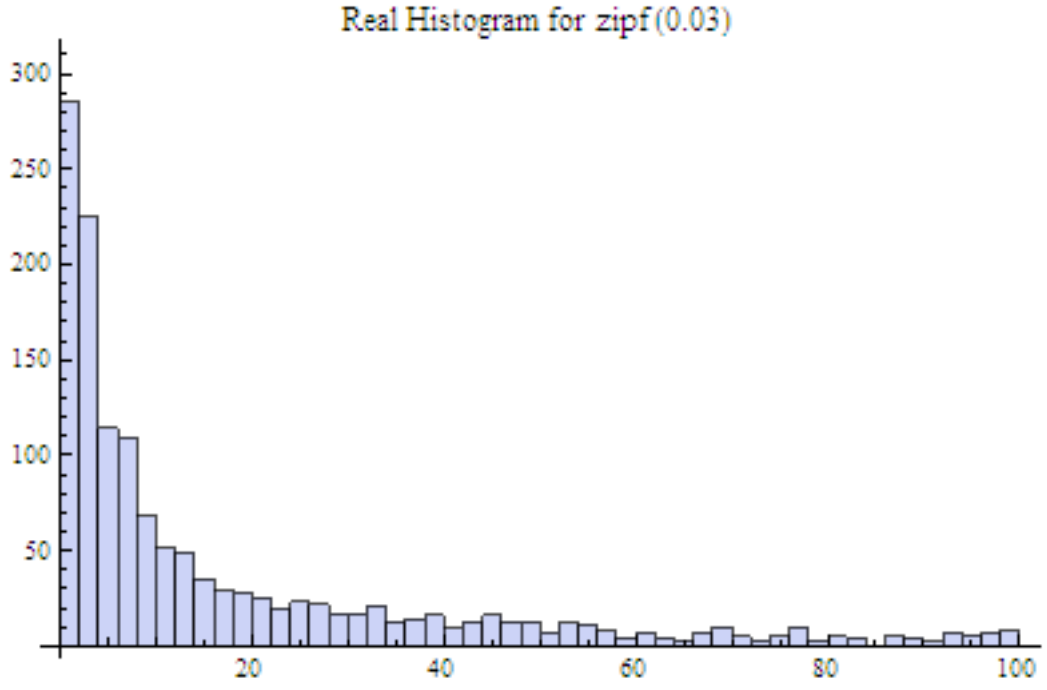
Оценка точности алгоритма

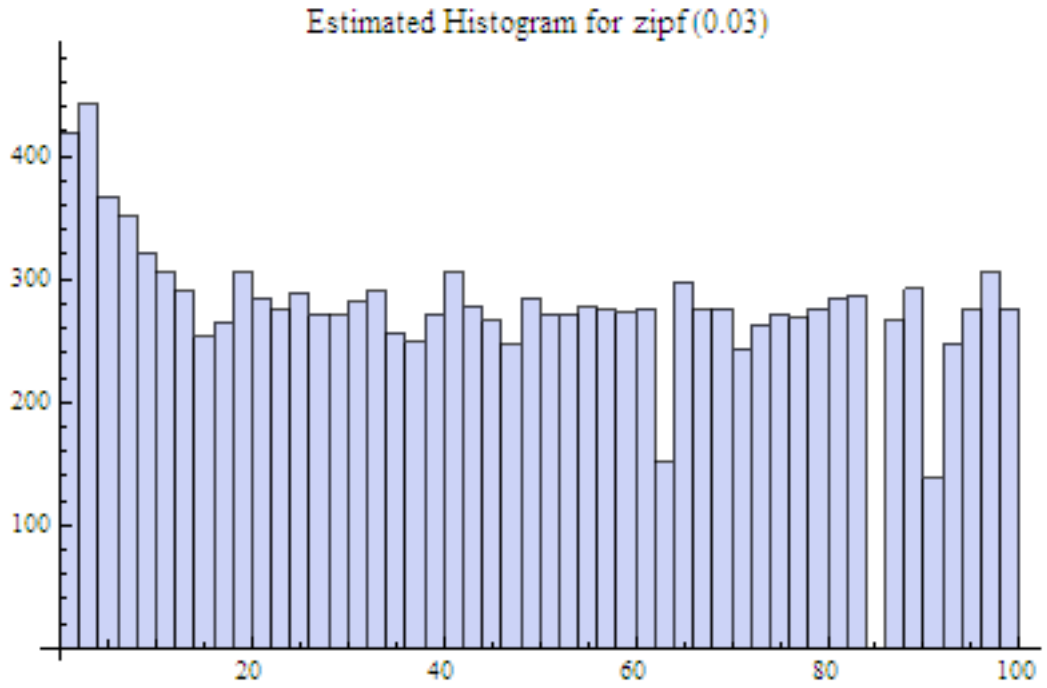
Точность алгоритма Count-Min sketch зависит от соотношения размера скетча и общего числа объектов. Это означает, что Count-Min sketch обеспечивает экономию памяти только для очень случайных данных, то есть вероятности событий (объектов) очень сильно скачут.

Две диаграммы показывают результаты Count-Min sketch размера 3×64, т.е. всего 192 счетчика.

В первом случае в sketch поданы относительно случайный набор данных в 10k элементов, среди них около 8500 уникальных значений (частоты элементов подчиняются распределению Ципфа, которое, например, соответствует распределению слов в текстах на естественных языках).

Настоящая гистограмма (показаны только самые частотные элементы, диаграмма имеет "длинный плоский хвост" справа, который отрезан на этой картинке) и гистограмма, восстановленная по скетчу показаны ниже:

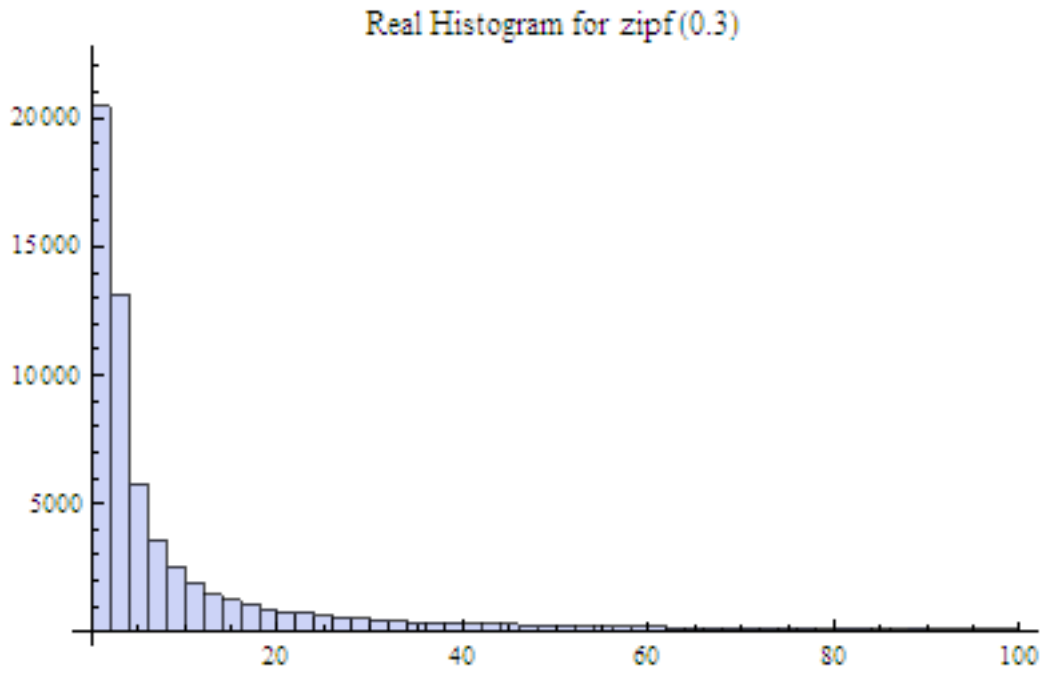


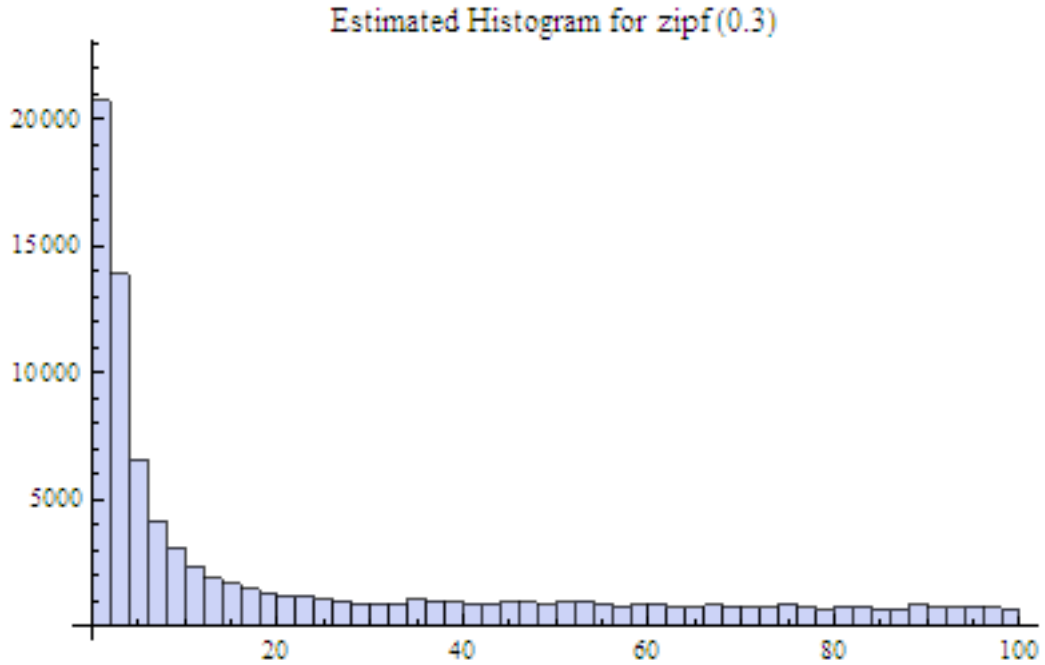


Очевидно, что Count-Min sketch не может отследить частоты 8500 элементов с использованием всего 192 счетчиков в случае малого отклонения частот, поэтому гистограмма получилась очень неточной.

Во втором случае в скетч были поданы относительно случайные данные в размере 80k элементов, среди которых также около 8500 уникальных.

Реальная гистограмма и гистограмма по скетчу показаны ниже:





Очевидно, что результат в данном случае более точный.

Оценка точности:

$estimation\ error\ \ \varepsilon \leq 2n/w$ $with\ probability\ \delta = 1 - (1/2)^d$	n – total count of registered events w – sketch width d – sketch height (aka depth)

Домашнее задание

- взять готовую реализацию одного из алгоритмов: **Bloom Filter**, **MinHash**, **SimHash**, **HyperLogLog** или **Count-Min Sketch** для вашего языка программирования для ее изучения;
- изучить код реализации;
- вариант повышенной сложности (опционально): самим реализовать этот алгоритм;
- найти большой датасет, подходящий для выбранного алгоритма;
- применить ее для решения практической задачи: например, определить, принаджежит ли элемент множеству, подсчитать число уникальных элементов в большом массиве данных или подсчитать числа вхождений каждого элемента в большой массив данных;
- оценить точность реализации (% ошибок, false positives и т.д.) с помощью тестов;
- выложить ваш код вместе с датасетом;
- (опционально): сделать вышеперечисленное для еще одного алгоритма из списка.
- (опционально): сравнить между собой "парные" алгоритмы, если вы выбрали их (напр, MinHash и SimHash).

Пример. Если вы выбрали MinHash и/или SimHash:

- разобрать пример реализации алгоритма MinHash по ссылке https://github.com/chrisjmccormick/MinHash (https://github.com/chrisjmccormick/MinHash) (можно также взять любую другую готовую реализацию алгоритма);
- (опционально) реализовать алгоритм MinHash на вашем языке программирования; если это python - попробовать улучшить пример по ссылке (отрефакторить, улучшить скорость работы, уменьшить требуемую память и т.д.);
- применить алгоритм для решения практической задачи: найти похожие объекты в большом датасете (1000 статей из интернета). Примеры статей есть в проекте https://github.com/chrisjmccormick/MinHash/tree/master/data (https://github.com/chrisjmccormick/MinHash/tree/master/data);
- (опционально) реализовать алгоритм SimHash на вашем языке программирования (как отправную точку можно взять реализацию https://github.com/seomoz/simhash-py (https://github.com/seomoz/simhash-py) и https://github.com/seomoz/simhash-cpp) (https://github.com/seomoz/simhash-cpp));
- (опционально) сравнить работу ваших алгоритмов MinHash и SimHash по скорости, точности работы и занимаемой памяти на любом датасете (1000 статей из интернета).

Литература и ссылки

HyperLogLog

Математика

HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm Philippe Flajolet, Éric Fusy, Olivier Gandouet, Frédéric Meunier http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf (http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf)

Об алгоритме и пример реализации на C++

https://habr.com/ru/post/119852/ (https://habr.com/ru/post/119852/)

Реализация

- Python https://pypi.org/project/hyperloglog/ (https://pypi.org/project/hyperloglog/)
- Java https://github.com/prasanthj/hyperloglog (https://github.com/prasanthj/hyperloglog)
- Java/Scala - сравнение реализаций - http://koff.io/posts/comparison-of-hll/ (http://koff.io/posts/comparison-of-hll/)
- C# (Microsoft): https://github.com/Microsoft/CardinalityEstimation (https://github.com/Microsoft/CardinalityEstimation)

Использование

- Redis Labs: https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/ (https://redislabs.com/blog/how-to-use-redis-at-least-x1000-more-efficiently/)
- Google: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf (https://storage.googleapis.com/pub-tools-public-publication-data/pdf/40671.pdf), https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog (https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog)
- Elastic search: https://www.elastic.co/blog/count-elasticsearch (https://www.elastic.co/blog/count-elasticsearch)
- AWS Redshift: https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html (https://docs.aws.amazon.com/redshift/latest/dg/r_COUNT.html)
- Reddit: https://redditblog.com/2017/05/24/view-counting-at-reddit/ (https://redditblog.com/2017/05/24/view-counting-at-reddit/)

Count-Min Sketch

Математика

An Improved Data Stream Summary: The Count-Min Sketch and its Applications Graham Cormode, S. Muthukrishnan http://www.eecs.harvard.edu/~michaelm/CS222/countmin.pdf (http://www.eecs.harvard.edu/~michaelm/CS222/countmin.pdf)

Реализация

- C, Python: https://github.com/barrust/count-min-sketch (https://github.com/barrust/count-min-sketch)

Обо всех рассмотренных структурах данных

https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/ (https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/)

Спасибо
за внимание!