
OTUS
ОНЛАЙН-ОБРАЗОВАНИЕ

Фильтр Блума

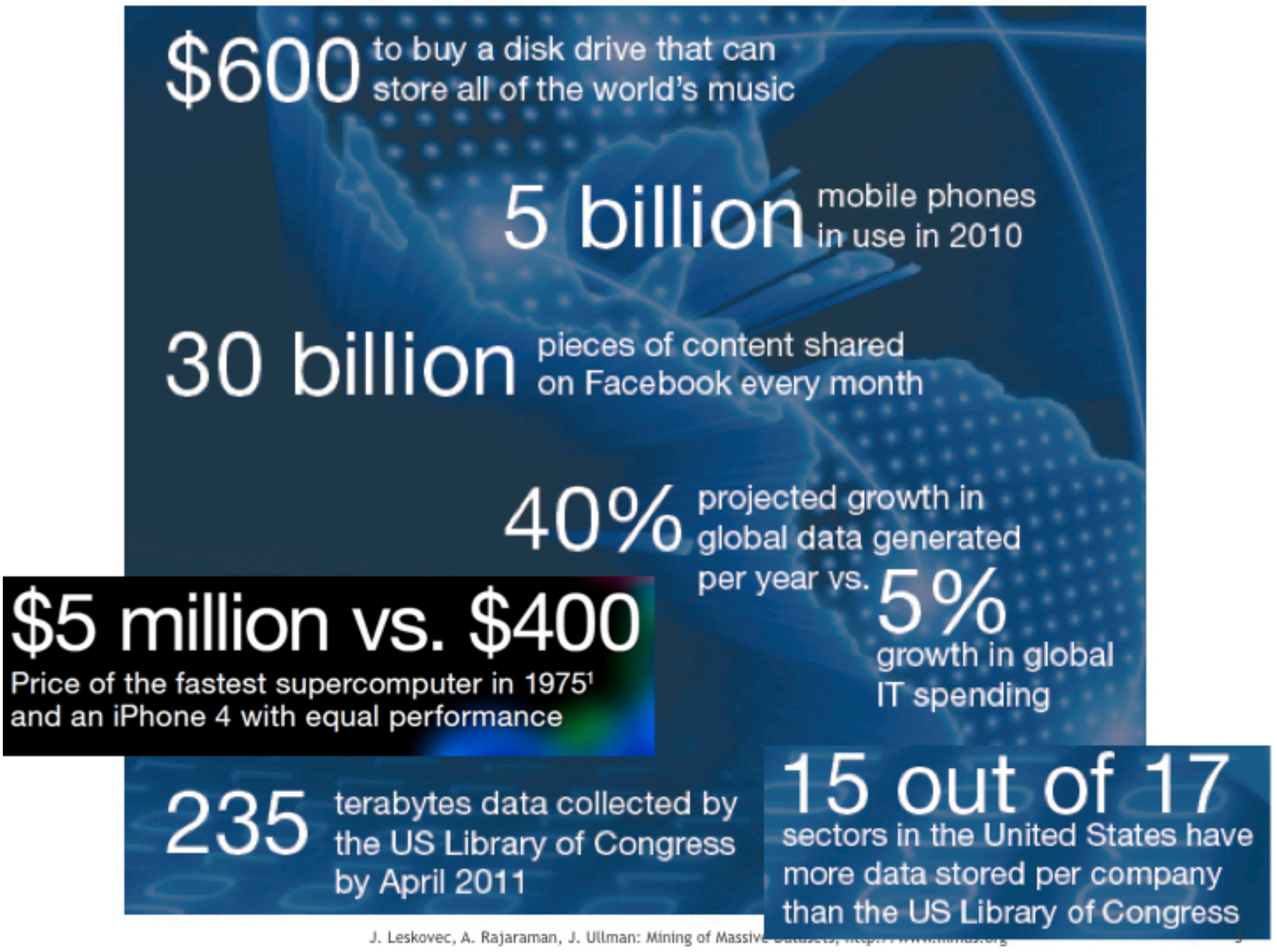
Как он устроен и его использование

Михаил Горшков
разработчик

ВКЛЮЧИТЬ ЗАПИСЬ!!!

Заполните опросник перед началом занятия:
https://docs.google.com/forms/d/e/1FAIpQLSedzn_AdN6MF1zTTD9oBArcT9dEZJMopXeBtUXhKvJ5v68wrw
(https://docs.google.com/forms/d/e/1FAIpQLSedzn_AdN6MF1zTTD9oBArcT9dEZJMopXeBtUXhKvJ5v68wrw

Data contains value and knowledge



Ближайшие 3 вебинара - про работу с "большими данными".

Тема бигдата пользуется в наши дни особой популярностью.

Рассмотрим и алгоритмы, и структуры данных для работы с большими данными.

"Большие данные" как правило не помещаются в оперативную память. Это требует переосмысления даже таких базовых алгоритмов, как подсчет числа уникальных элементов и принадлежность элемента множеству.

Коснемся небольшого числа тем. Будем рассматривать:

- Задача принадлежности элемента множеству - *Bloom Filter*;

- Поиск похожих данных - *MinHash*, *SimHash*;
- Подсчет количества вхождений элемента в множество - *Count Min Sketch*;
- Сколько уникальных элементов содержит множество - *HyperLogLog*.

Эти алгоритмы и структуры данных вероятностные. Почему вероятностные? Что это значит?

Что такое "вероятностные алгоритмы"?

Perfect is the enemy of good. Voltair

Что делают с большими данными?

- Hadoop, Spark;
- вероятностные алгоритмы и структуры данных.

Пример. Подсчет числа "хитов" разных видео (NetFlix, YouTube).

unique id -> view count

- делаем хэш-таблицу;
- требуемое место - O(n);
- 4 млрд видео, каждое видео имеет 32-битный уникальный id (это минимум), тогда нужно 16 Гб только чтобы хранить id! Затем счетчик—32-битное целое - еще 16 Гб. Итого 32 Гб! Не помещается в RAM.

Можно использовать Spark, но есть подход лучше - вероятностный. Задача решается алгоритмом *HyperLogLog*.

Ввиду наличия большого к-ва данных иногда приходится идти на компромиссы—результат вместо точного получается с некоей вероятностью. Точное решение некоторых задач является невозможным или нерациональным: слишком дорогим, требующим слишком много ресурсов и т.п. Немного жертвуя точностью можно получить огромный выигрыш в производительности и/или ресурсах. Для повседневных задач могут быть избыточны, хорошо подходит для больших данных.

Вероятностный алгоритм - не значит, что алгоритм не рабочий или работает через раз.

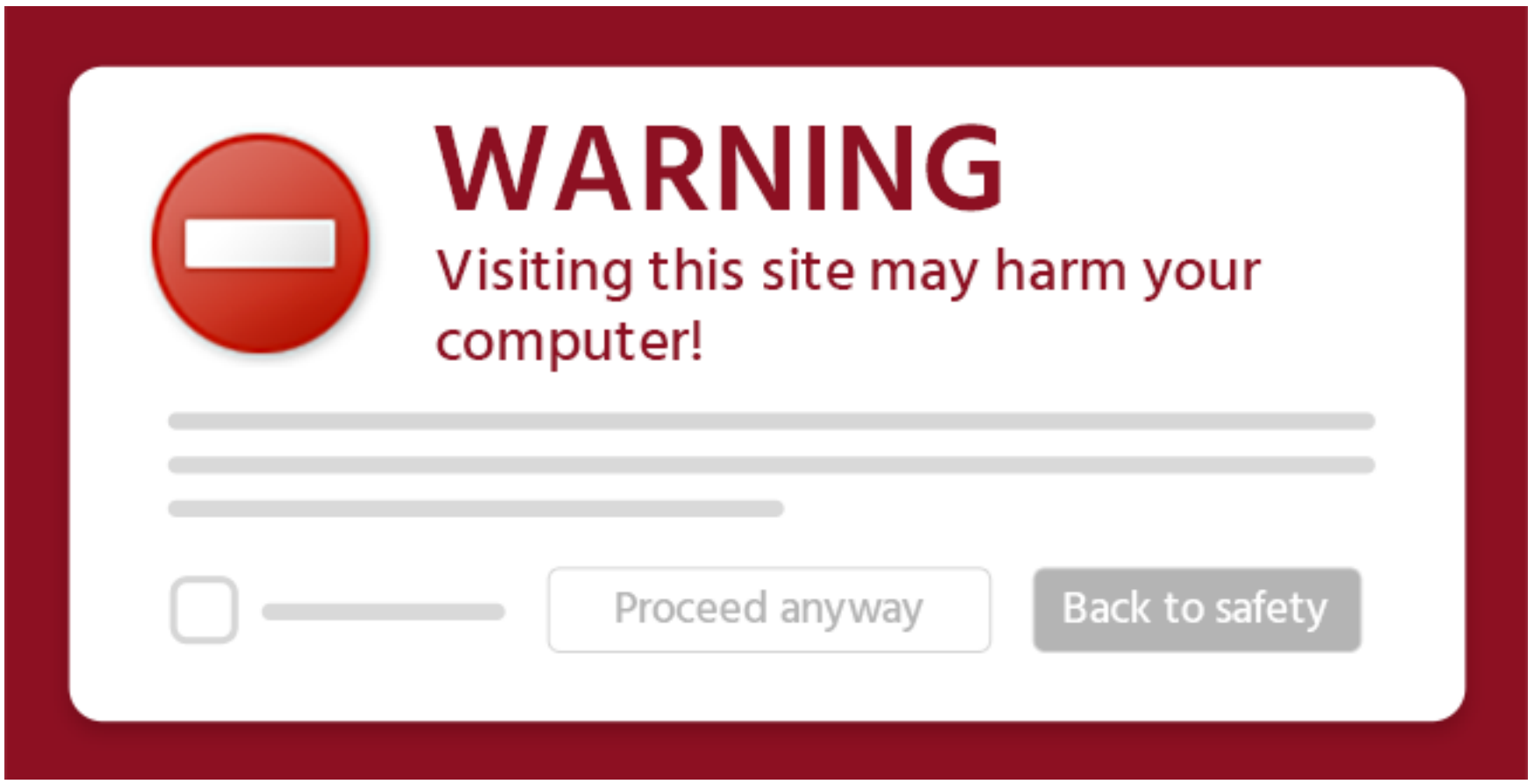
Алгоритм:

- всегда работает;
- результат может вычисляться с погрешностью, но всегда имеется ее точная оценка.

Рассмотрим этот подход на примере другой задачи.

Фильтр Блума

Постановка задачи



Вы - разработчик веб-браузера. Пользователи заходят на разные сайты. Вам нужно предупреждать пользователя о том, что URL, на который он зашел - вредоносный.

Пример URL: http://domain.com/page (http://domain.com/page).

Количество доменов - 330 млн. Возможны несколько URL на одном домене. Как это сделать? Ваши варианты. Каковы проблемы, с которыми вам придется столкнуться при решении этой задачи?

Требования к структуре данных

Основное назначение - определять, есть ли заданный объект в заданном множестве.

API структуры данных:

- добавить объект: операция **add**;
- определить, содержит ли она объект: операция **has**;
- удалить объект: операция **delete**.

Варианты решения:

- Хранить все URL в массиве (сортированном) и искать URL бинарным поиском. Например, у нас 1 млн зловредных доменов, и на каждый нужно 50 байт. Итого нужно 50 Мб. Вставка - сложность $O(N)$ $O(N)$. Поиск занимает $O(log(N))$ $O(log(N))$. Место - $O(N)$ $O(N)$;
- Бинарное дерево поиска. Вставка - $O(log(N))$ $O(log(N))$. Поиск $O(log(N))$ $O(log(N))$. Место - $O(N)$ $O(N)$;
- Хэш-таблица. Вставка - худшее время $O(N)$ $O(N)$, поиск - худшее время $O(N)$ $O(N)$. Место - $O(N)$ $O(N)$. Идеальное хэширование? Не подходит, потому множество ключей не является заранее заданным и статическим.

Контейнер	Вставка (avg)	Вставка (worst)	Поиск (avg)	Поиск (worst)	Место
Сортированный массив	$O(N)$ $O(N)$	$O(N)$ $O(N)$	$O(log(N))$ $O(log(N))$	$O(log(N))$ $O(log(N))$	$O(N)$ $O(N)$
Дерево	$O(log(N))$ $O(log(N))$	$O(N)$ $O(N)$	$O(log(N))$ $O(log(N))$	$O(log(N))$ $O(log(N))$	$O(N)$ $O(N)$
Хэш-таблица	$O(1)$ $O(1)$	$O(N)$ $O(N)$	$O(1)$ $O(1)$	$O(N)$ $O(N)$	$O(N)$ $O(N)$

Проблема со всеми реализациями - требуется $O(N)$ $O(N)$ места (надо хранить все URL-ы) и поиск в худшем случае $O(N)$ $O(N)$.

Готовы пожертвовать точностью

НО!!!

- готовы ошибаться, но с заранее известным и небольшим % ошибок;

Почему заранее известным? Если процент не известен, придется очень часто перепроверять.

Попробуем массив

Начнем с варианта простого массива, но будем хранить там хэшированные URL-ы (можно отсортировать его по значению хэш-функции для log-поиска).

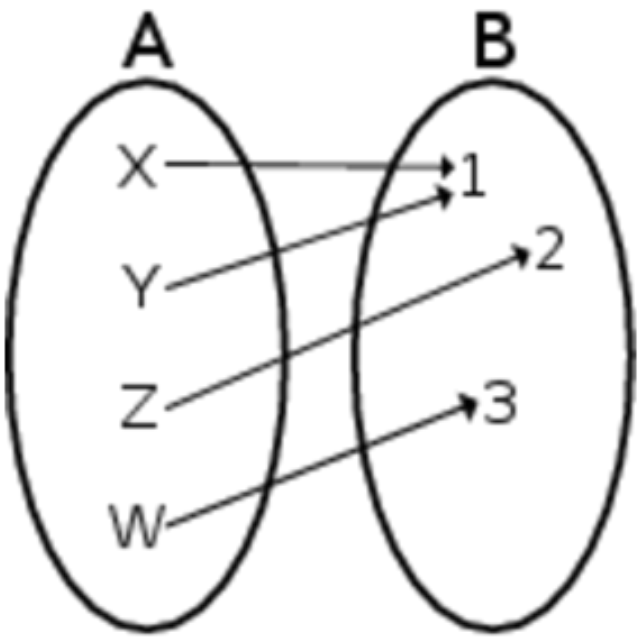
- Для каждого URL U подсчитаем хэш-функцию $h(U_i)$. Пусть результат хэш-функции будет всегда состоять из m бит.
- Будем хранить хэшированные URL-ы в обычном массиве.
- Массив будет представлять собой множество S_{hashed}: $(h(U_0), h(U_1), \dots h(U_{N-1}))$
- Размер массива $m * N$, но коэффициент уже будет меньше, чем для просто массива URL-ов.
- Чтобы проверить, является ли URL вредоносным, посчитаем от него хэш-функцию $h(U)$ и поищем результат в массиве. Если есть, значит URL, возможно, вредоносный. Если нет, то точно не вредоносный.

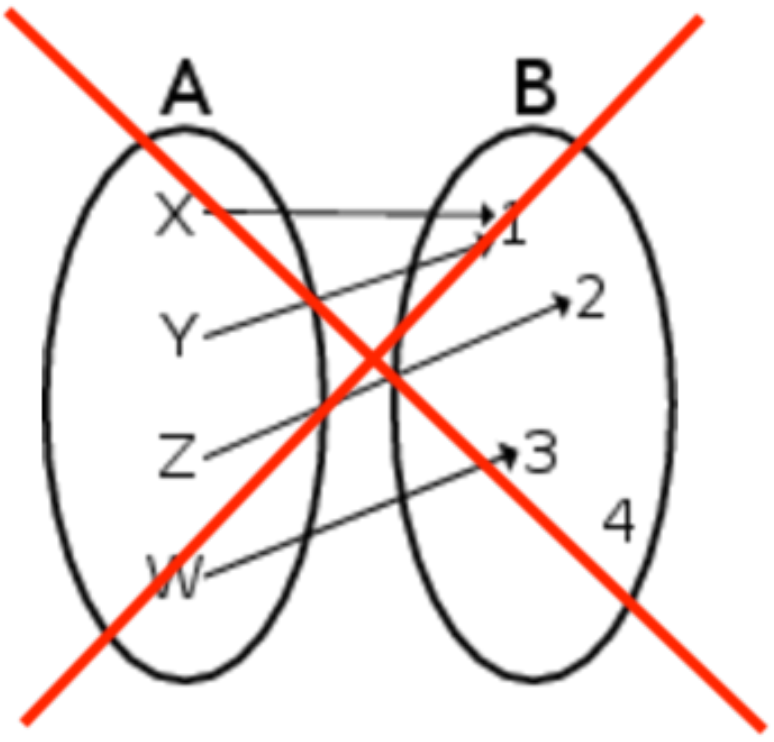
Вопрос 1: почему "возможно"?

Вопрос 2: бывают ли false negatives в данном случае?

Хэш - сюръективная функция

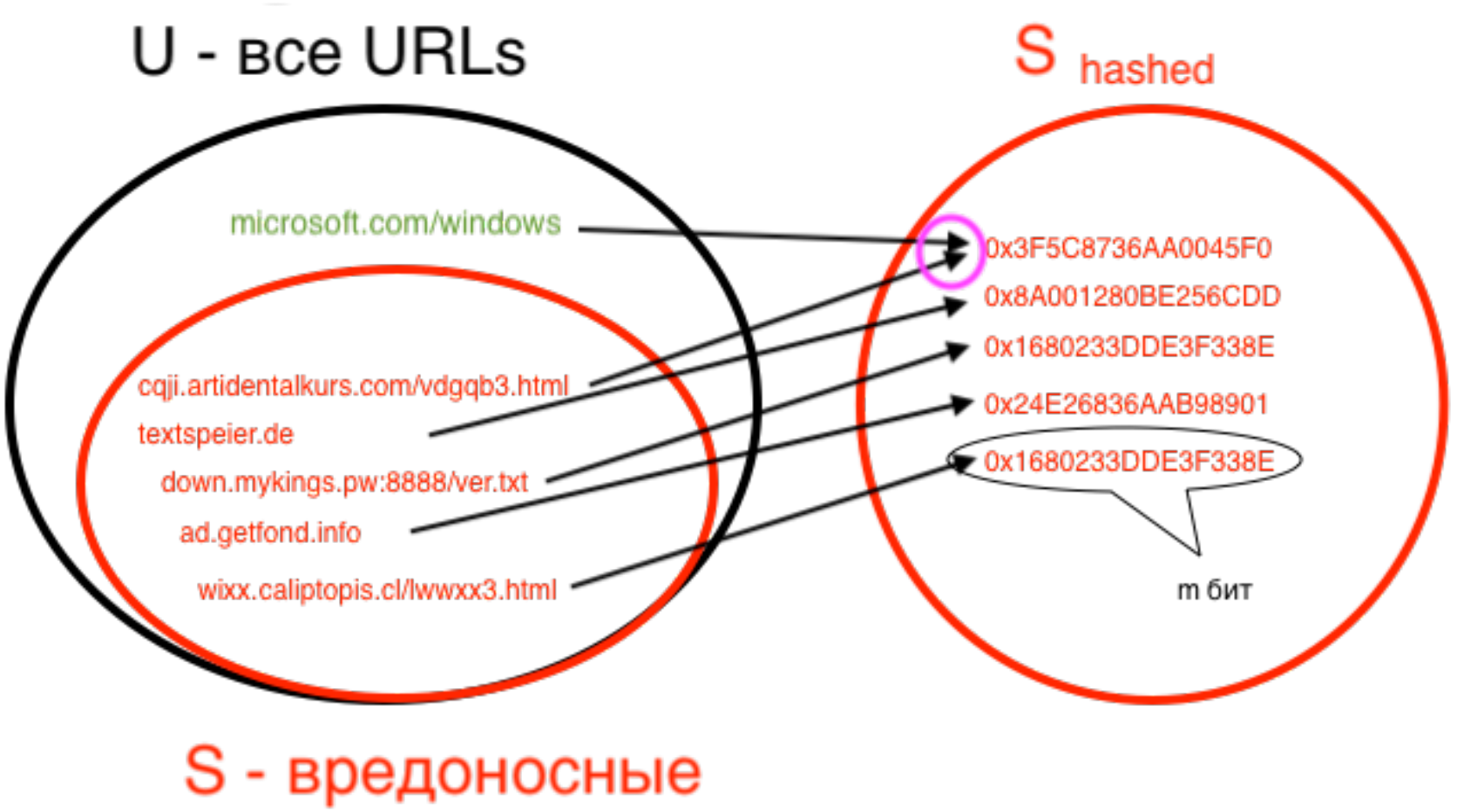
Сюръекция - это функция f: $A \to B$ такая что для каждого элемента b в множестве B найдется как минимум один элемент a в множестве A такой что $f(a) = b$.





Поэтому не может быть false negatives.

Почему могут быть false positives?



- Опасность вернуть неверный результат появляется в случае коллизии, то есть если $h(U) = h(U_i)$ $h(U) = h(U_i)$, но сам $U \neq U_i$ $U \neq U_i$.
- Чем больше m m, тем меньше вероятность коллизии, следовательно меньше вероятность ложного срабатывания (false positive).
- Мы можем сделать сколь угодно малой вероятность ложного срабатывания за счет увеличения m m.

Небольшой экскурс в теорию вероятностей

Элементарным исходом (или элементарным событием) называют любой простейший (т.е. неделимый в рамках данного опыта) исход опыта.
Пример. Выпадение герба после подбрасывания монеты.

Множество всех элементарных исходов будем называть **пространством элементарных исходов**.
Пример. Шесть граней игральной кости.

Любой набор элементарных исходов называют **событием**.
Пример. Выпадение четного числа на игральной кости

Вероятность события будем представлять числом pp, которое принимает значение от 0 до 1.

Некоторые свойства:

- $p(x) = 1$ $p(x) = 1$ - означает, что мы уверены, что событие точно произойдет;
- $p(y) = 0 p(y) = 0$ - означает, что мы точно уверены, что событие не произойдет;
- $\sum_x p(x) = 1$ $\sum_x p(x) = 1$ - сумма вероятностей по пространству элементарных исходов равна 1;
- если pp - вероятность того, что событие произойдет, то вероятность того, что событие не произойдет, равна 1 - pp.
- вероятность совместного исхода двух событий $p(x, y) p(x, y) = p(x \ and \ y \, p(x \ and \, y)$ (например бросаем кость и одновременно выпадет четное и простое число);
- если $p(x, y) = p(x)p(y)$ $p(x, y) = p(x)p(y)$, то события называются независимыми.

Вопрос: независимы ли события "выпадает четное число на кости" и "выпадает простое число на кости"?

$p_{even} = \frac{1}{2} p_{even} = \frac{1}{2}$, $p_{prime} = \frac{1}{2} p_{prime} = \frac{1}{2}$;

Какова вероятность совместного исхода двух этих событий?

Ваши варианты в чат*

$p_{even \ and \ prime} = ? p_{even \ and \ prime} = \ ?$

Почему вероятность false positive может быть сколь угодно малой?

Пусть P - вероятность верного определения, что URL U НЕ находится в нашей структуре данных S. Это означает, что нет false positive для некоторого конкретного URL.

То есть, с вероятностью P для всех $0 \leq i \leq N - 1$

$h(U_i) \neq h(U)$

Тогда $p = 1$ - P - вероятность ложного срабатывания.

У нас в хэше m бит, всего возможны 2^m значений хэша.

Вероятность, что $h(U_i) = h(U)$ для конкретного i при хорошо выбранной хэш-функции (значения равномерно распределены) равна $\frac{1}{2^m}$.

Вероятность, что $h(U_i) \neq h(U)$ для любого i равна $1 - \frac{1}{2^m}$, и все эти события независимы для всех N хэшей.

Тогда $\mathbf{p = 1 - (1 - \frac{1}{2^m})^N}$.

То есть, чтобы обеспечить сколь угодно малую наперед заданную вероятность ложного срабатывания p, необходимо правильно подобрать m.

$p \to 0$ при $m \to \infty$.

Сколько нужно памяти на всю структуру данных, чтобы обеспечить заданную вероятность ложного срабатывания?

Обозначим к-во памяти в битах на всю структуру данных через M. Найдем M как функцию от p и N.

$$M = Nm$$

бит.

$$m = \frac{M}{N}$$

Подставляя в предыдущее уравнение, получаем

$$p = 1 - (1 - \frac{1}{2^{\frac{M}{N}}})^N$$

$$p = 1 - (1 - \frac{1}{2^{\frac{M}{N}}})^N$$

Вычисляем M M:

$$(1 - \frac{1}{2^{\frac{M}{N}}})^N = 1 - p$$

$$(1 - \frac{1}{2^{\frac{M}{N}}})^N = 1 - p$$

$$1 - \frac{1}{2^{\frac{M}{N}}} = (1 - p)^{\frac{1}{N}}$$

$$1 - \frac{1}{2^{\frac{M}{N}}} = (1 - p)^{\frac{1}{N}}$$

$$\frac{1}{2^{\frac{M}{N}}} = 1 - (1 - p)^{\frac{1}{N}}$$

$$\frac{1}{2^{\frac{M}{N}}} = 1 - (1 - p)^{\frac{1}{N}}$$

$$2^{\frac{M}{N}} = \frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

$$2^{\frac{M}{N}} = \frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

$$\frac{M}{N} = log_2 \frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

$$\frac{M}{N} = log_2 \frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

$$M = Nlog_2 \frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

$$M = Nlog_2 \frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

Столько памяти (в битах) нужно на всю структуру данных.

Если задаться достаточной малой вероятностью pp, например, 1%, то можно доказать, что

$$M \approx Nlog_2 \frac{N}{p}$$

$$M \approx Nlog_2 \frac{N}{p}$$

$$((1 - x)^{\frac{1}{N}} \approx 1 - \frac{x}{N}, x \to 0)$$

$$((1-x)^{\frac{1}{N}} \approx 1 - \frac{x}{N}, x \to 0)$$

На каждый хэш требуется $log_2 \frac{N}{p} log_2 \frac{N}{p}$ бит.

Итоги варианта с хэшированием

По памяти лучше, чем просто хранить все URL, но имеются false positives

Как бороться с false positives?

Ваши варианты в чат

Как бороться с false positives?

- Хранить отдельно список URL, для которых известно, что наша хэш-функция выдает false positives. В случае положительного ответа дополнительно проверяем этот ответ по списку false positives. Если он там есть, то возвращаем false вместо true.
- Все положительные ответы проверять альтернативным способом (например, проверять их с помощью внешнего сервиса).

Улучшим вариант с хэш-таблицей: используем битовый массив

Предположим, нам нужно каким-то образом хранить подмножество S чисел от 0 до 999. Например, *{2, 35, 878, 999}*.

Варианты те же: массив, дерево, хэш-таблица, но можно еще битовый массив.

Заведем битовый массив из 1000 бит. Каждый бит с номером i - это признак, есть ли число i в нашем подмножестве. Если все биты нулевые, подмножество пусто. Если все биты установлены в 1 - то это все числа от 0 до 999.

Добавление, и удаление числа из множества тривиальны: нужно просто выставить/убрать соответствующий бит.

Для проверки принадлежности числа X множеству нужно проверить, установлен ли бит с номером X.

Независимо от размера подмножества размер нашего массива всегда 1000 бит. Если бы мы хранили, скажем, наше подмножество как массив 32-битных целых, понадобилось бы $32 * S$ бит, где S - размер подмассива.

Если S маленькое, выгоднее хранить как массив 32-битных целых. Если большое, то выгоднее использовать битовый массив.

Комбинируем битовый массив с хэш-функцией

Нам нужен был битовый массив фактически для нумерации элементов, но сами элементы могут быть любыми объектами. Однако их количество должно быть невелико.

Чтобы нумеровать элементы множества, можем использовать хэш-функцию.

Пусть снова $h(x)$ $h(x)$ - хэш-функция, возвращающая m m бит.

Пусть имеем множество $S = x_0, x_1, \ldots, x_{N-1}$ $S = x_0, x_1, \ldots, x_{N-1}$.

Представим его как битовый массив из 2^m 2^m элементов.

Рассматриваем $h(x_i)$ $h(x_i)$ как число из диапазона $0, 1, 2^{m-1}$ $0, 1, 2^{m-1}$. Для каждого x_i x_i выставим в 1 элемент с номером $h(x_i)$ $h(x_i)$ массива.

Можем добавить x в множество, установив соответствующий бит (с номером $h(x)$ $h(x)$). Можем удалить из множества, сняв бит. Можем проверить принадлежность элемента множеству, посчитав хэш и проверив бит.

Проблема опять та же - false positives. То есть, $h(x) = h(x_i)$ $h(x) = h(x_i)$ для какого-то i, но $x \neq x_i$ $x \neq x_i$. Как мы выяснили ранее, вероятность false positive

$$p = 1 - (1 - \frac{1}{2^m})^N$$

$$p = 1 - (1 - \frac{1}{2^m})^N$$

Размер всей структуры данных в битах $M = 2^m$ $M = 2^m$ (ранее было $M = Nm$ $M = Nm$).

Подставляем $M = 2^m$ $M = 2^m$ в предыдущее уравнение

$$p = 1 - (1 - \frac{1}{M})^N$$

$$p = 1 - (1 - \frac{1}{M})^N$$

Аналогично выведенному ранее

$$M = \frac{1}{1 - p^{\frac{1}{N}}}$$

$$M = \frac{1}{1 - p^{\frac{1}{N}}}$$

и

$$M \approx \frac{N}{p}$$

$$M \approx \frac{N}{p}$$

Нам требуется небольшие значения p p, следовательно M M будет большим.

Надо минимизировать M.

В предыдущей схеме было $Nlog_2\frac{N}{p}$ $Nlog_2\frac{N}{p}$.

Сравним два подхода

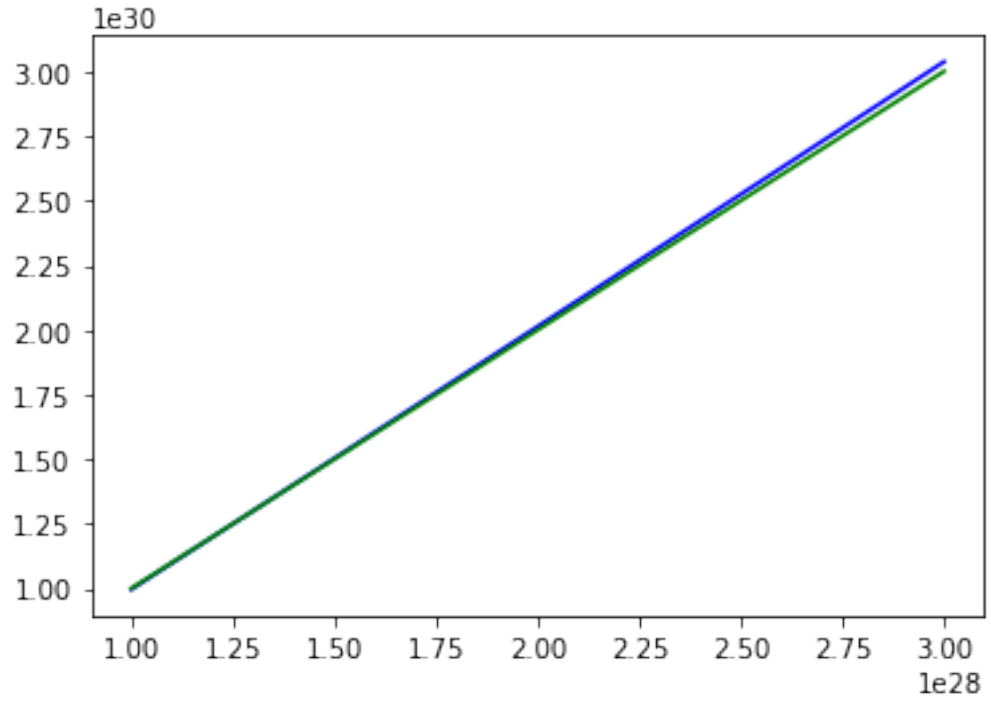
При N был коэффициент $log_2\frac{N}{p}log_2\frac{N}{p}$, стал $\frac{1}{p}\frac{1}{p}$. Чем меньше коэффициент, тем лучше, потому что меньше M.

Если N - очень большое, то текущий подход лучше. Например, если p = 0.01, то текущий подход лучше, если $N > 1.28 * 10^{28}$

Схема	M
Массив хэшированных URL	$Nlog_2\frac{N}{p}$ $Nlog_2\frac{N}{p}$
Битовый массив с хэш-функцией	$\frac{N}{p}\frac{N}{p}$

In [27]:

```
## Графики

from numpy import *
import math
import matplotlib.pyplot as plt

p = 0.01
N = linspace(1*(10 ** 28), 3*(10 ** 28), 10)
HashedUrls = N * log2(N / p)
BitArray = N / p

plt.plot(N, HashedUrls, 'b')
plt.plot(N, BitArray, 'g')

plt.show()
```



Почему второй подход не так эффективен и что с этим делать?

Вопрос к аудитории: почему второй подход оказался не таким эффективным?

Первая схема использует mN mN бит памяти, вторая - 2^m 2^m бит. Вторая схема использует экспоненциально больше памяти при фиксированном N N. При этом вероятность ложного срабатывания одинакова.

На первый взгляд, вторая идея неудачна. Но ее можно использовать в несколько ином ключе.

Множественная фильтрация

Назовем хеширование в битовый массив "фильтром". Используем фильтрацию несколько раз.

Используем k k различных независимых друг от друга хэш-функций: $h_0, \ldots, h_{k-1}$ $h_0, \ldots, h_{k-1}$. Каждая хэширует элемент нашего множества в m m-битный результат. Наша структура данных будет состоять из k k битовых массивов, каждый содержит 2^m 2^m бит.

Общий размер структуры данных $M = k2^m$ $M = k2^m$ бит.

Как работает API нашей структуры данных?

- **add**. Считаем $h_0(x)$ $h_0(x)$, записываем 1 в $h_0(x)$ $h_0(x)$-й бит 0-го массива. Считаем $h_1(x)$ $h_1(x)$, записываем 1 в $h_1(x)$ $h_1(x)$-й бит 1-го массива и т.д.
- **has**. Посчитаем хэши от целевых данных $h_i(x)$ $h_i(x)$. Проверим, установлен ли бит с номером $h_i(x)$ $h_i(x)$ в i-м фильтре.

В случае, если каждый фильтр показывает отсутствие бита, x x отсутствует в нашем контейнере. Если бит где-то установлен, значит данные имеются в нашем контейнере.

- **delete**.

Вопрос к аудитории: как сделать delete?

Посчитаем вероятность false positive в этом случае.

$p_i = 1 - (1 - \frac{1}{2^m})^N$ $p_i = 1 - (1 - \frac{1}{2^m})^N$ для каждого фильтра.

Поскольку фильтры независимы, общая вероятность false positive

$$p = (1 - (1 - \frac{1}{2^m})^N)^k$$

$$p = (1 - (1 - \frac{1}{2^m})^N)^k$$

Общий размер структуры данных $M = k2^m$ $M = k2^m$ бит.

Подставляя в предыдущее уравнение, получаем, что

$$M = \frac{k}{1 - (1 - p^{\frac{1}{k}})^{\frac{1}{N}}}$$

$$M = \frac{k}{1 - (1 - p^{\frac{1}{k}})^{\frac{1}{N}}}$$

$$p^{\frac{1}{k}} << 1 => M \approx \frac{kN}{p^{\frac{1}{k}}}$$

$$p^{\frac{1}{k}} << 1 => M \approx \frac{kN}{p^{\frac{1}{k}}}$$

Стратегия более эффективна по памяти. Если $k\,k = 2$, $M \approx k\sqrt{p}\,M \approx k\sqrt{p}$.

Число ошибок экспоненциально уменьшается с увеличением $k\,k$.

В случае схемы с одним фильтром к-во ошибок линейно уменьшается с ростом к-ва бит. Ранее выведенная формула $p = 1 - (1 - \frac{1}{2^m})^N$ $p = 1 - (1 - \frac{1}{2^m})^N$.

Если $\frac{1}{2^m}\frac{1}{2^m}$ мало, то вероятность ошибки обратно пропорциональна $N\,N$.

Перекрывающиеся фильтры

Вместо $k\,k$ массивов по $2^m\,2^m$ бит будем использовать один массив на $2^m\,2^m$ бит. Как работают наши операции теперь?

- **add**. Считаем $h_0(x)\,h_0(x)$, записываем 1 в $h_0(x)\,h_0(x)$-й бит массива. Считаем $h_1(x)\,h_1(x)$, записываем 1 в $h_1(x)\,h_1(x)$-й бит массива и т.д.
- **has**. Посчитаем хэши от целевых данных $h_i(x)\,h_i(x)$. Проверим, установлены ли биты с номерами $h_i(x)\,h_i(x)$ в массиве.

В случае, если все биты сняты, $x\,x$ отсутствует в нашем контейнере. Если бит где-то установлен, значит данные имеются в нашем контейнере.

- **delete**. Посчитать хэши, снять соответствующие биты.

Вопрос к аудитории: почему это не будет работать?

Какова вероятность false positive?

Предположим, что $x\,x$ отсутствует в контейнере. Ложное срабатывание происходит, когда $h_0(x) = h_{i_0}(x_{j_0})\,h_0(x) = h_{i_0}(x_{j_0})$ для некоторых $i_0\;i_0$ и $j_0 j_0$, $h_1(x) = h_{i_1}(x_{j_1})\,h_1(x) = h_{i_1}(x_{j_1})$ для некоторых $i_1\;i_1$ и j_1 и т.д. для остальных хэш-функций $h_2 \ldots h_{k-1}$.

Это независимые события, следовательно вероятность их совместного появления - это вероятность их произведения.

Интуитивно понятно, что каждое индивидуальное событие имеет одну и ту же вероятность, поэтому найдем вероятность для h_0. То есть, найдем вероятность события $h_0(x) = h_{i_0}(x_{j_0})$ для некоторых i_0 и j_0 и возведем ее в k-ую степень, чтобы получить общую вероятность.

Вероятность того, что $h_0(x) = h_{i_0}(x_{j_0})$ для некоторых i_0 и j_0 равна 1 - вероятность того, что $h_0(x) \neq h_{i_0}(x_{j_0})$ для всех i_0 и j_0. Это независимые события, вероятность каждого - $1 - \frac{1}{2^m}$, следовательно

$p(h_0(x) = h_{i_0}(x_{j_0}))$ для некоторых i_0 и $j_0 = 1 - (1 - \frac{1}{2^m})^{kN}$, так как есть kN пар различных значений (i_0, j_0). Следовательно,

$$p = (1 - (1 - \frac{1}{2^m})^{kN})^k$$

$2^m = M$, следовательно,

$$p = (1 - (1 - \frac{1}{M})^{kN})^k$$

$$M = \frac{1}{1 - (1 - p^{\frac{1}{k}})^{\frac{1}{kN}}}$$

Похоже на результат для повторяющихся фильтров.

Так как $p^{\frac{1}{k}} << 1$,

$$M \approx \frac{kN}{p^{\frac{1}{k}}}$$

Можно доказать, что

$$\frac{1}{1 - (1 - p^{\frac{1}{k}})^{\frac{1}{kN}}} < \frac{k}{1 - (1 - p^{\frac{1}{k}})^{\frac{1}{N}}}$$

Перекрывающиеся фильтры лучше, чем множественная фильтрация и намного лучше, чем одиночный фильтр ($1.28 * 10^{28}$).

Схема	M
Массив хэшированных URL	$Nlog_2 \frac{N}{p}$
Битовый массив с хэш-функцией	$\frac{N}{p}$
Перекрывающиеся фильтры	$\frac{kN}{p^{\frac{1}{k}}}$

Фильтр Блума

То же самое, что перекрывающиеся фильтры, но хэшируем в массив из M бит (а не в 2^m бит).

Аналогично предыдущим выкладкам

$$p = (1 - (1 - \frac{1}{M})^{kN})^k$$

и

$$M = \frac{1}{1 - (1 - p^{\frac{1}{k}})^{\frac{1}{kN}}}$$

Какое значение k выбрать, чтобы минимизировать к-во бит M для данной вероятности p и размера N?

Не будем это доказывать, но

$$k \approx \frac{M}{N} ln2$$

минимизирует p. ln - натуральный логарифм.

После подстановки в предыдущее равенство получаем

$$M = -\frac{N log_2 p}{ln2}$$

Это лучший показатель для M из тех вариантов структур данных, что мы рассмотрели.

В частности,

$$m = \frac{1}{ln2} log_2 \frac{1}{p} <= log_2 \frac{N}{p}$$

(столько было для массива хэшей)

Bloom Filter требует $\frac{1}{ln2} log_2 \frac{1}{p} \approx 1.44 * log_2 \frac{1}{p}$ бит на элемент.

Доказано, что любая структура, поддерживающая операции add и $test$, потребует как минимум $log_2 \frac{1}{p}$ бит на элемент.

Bloom Filter близок к оптимальному.

N заранее не известно

Как правило, N не известно заранее, поэтому выбираем максимальный размер множества, который мы хотим представить - n.

$$M = -\frac{nlog_2 p}{ln2}$$

и

$$k = ln\frac{1}{p}$$

Подводим итог по Bloom Filter

Пусть хотим Bloom Filter с емкостью n и вероятностью p для false positives.

Тогда выбираем $k = ln\frac{1}{p}$ независимых хэш-функций h_0, h_1,...,h_{k-1}.

Каждая хэш-функция имеет диапазон 0 - $M - 1$, где M - общее число бит в фильтре.

$$M = -\frac{nlog_2 p}{ln2}$$

Нумеруем биты от 0 до $M - 1$.

При добавлении элемента x устанавливаем биты $h_0(x)$, $h_1(x)$,...,$h_{M-1}(x)$ в фильтре.

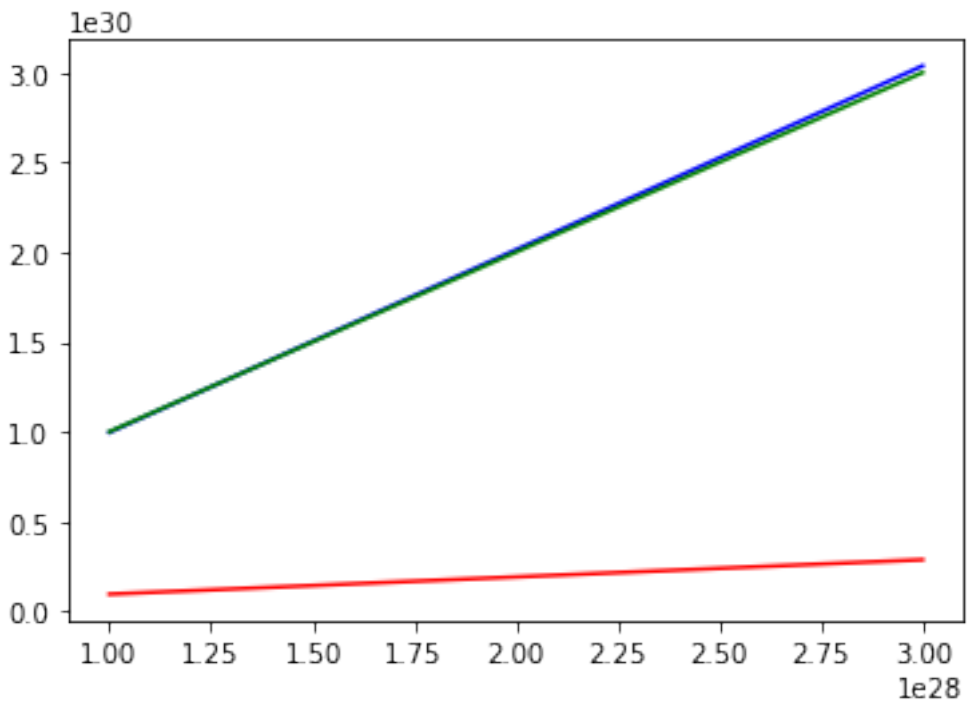
Чтобы проверить наличие элемента проверяем, что все биты $h_0(x)$, $h_1(x)$,...,$h_{M-1}(x)$ установлены.

Итоговая таблица по вычислению размера M.

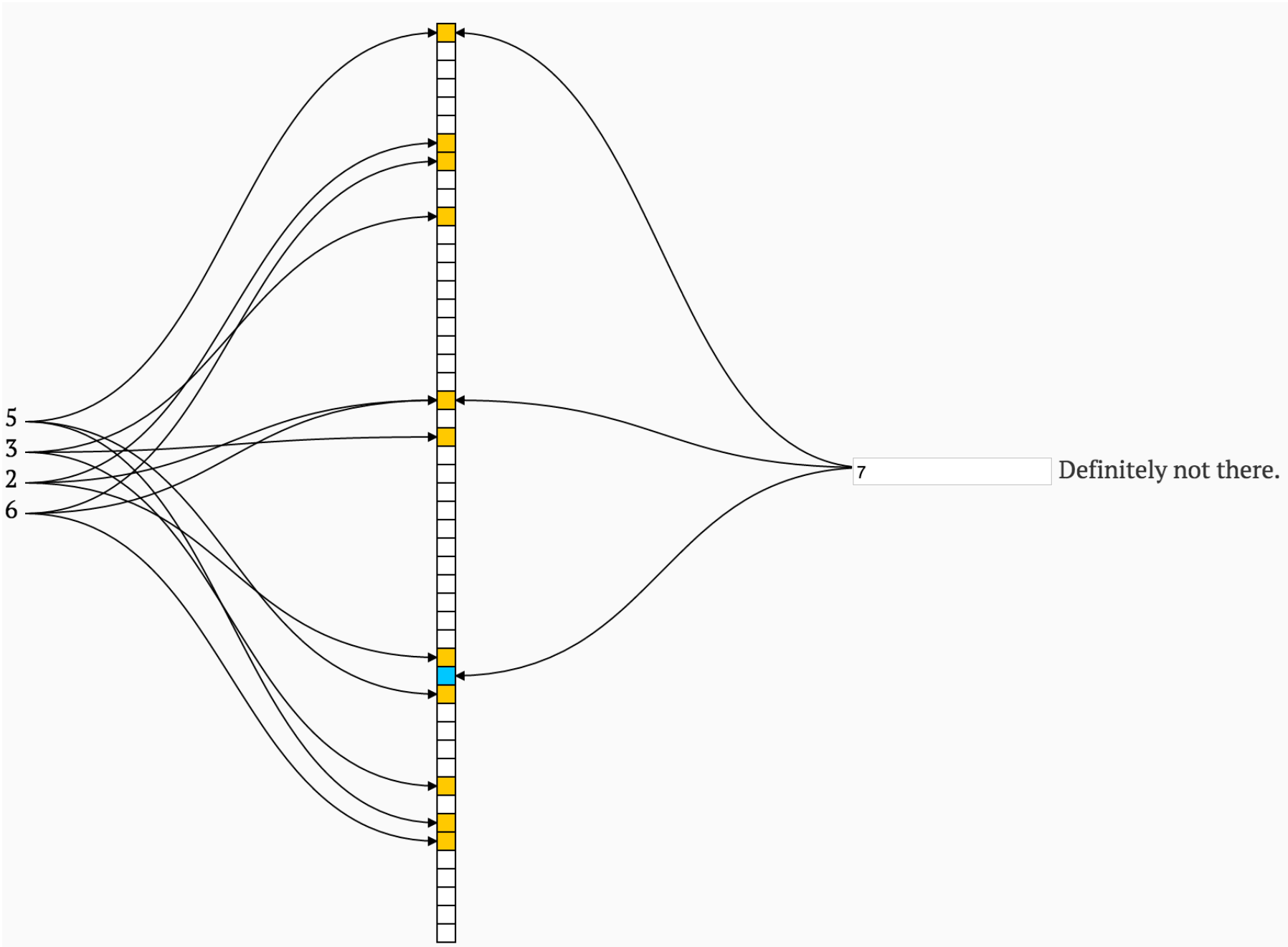
Схема	M
Массив хэшированных URL	$Nlog_2\frac{N}{p}$
Битовый массив с хэш-функцией	$\frac{N}{p}$
Перекрывающиеся фильтры	$\frac{kN}{p^{\frac{1}{k}}}$
Bloom Filter	$-\frac{nlog_2 p}{ln2}$

In [36]:

```
## Графики

from numpy import *
import math
import matplotlib.pyplot as plt

p = 0.01
N = linspace(1*(10 ** 28), 3*(10 ** 28), 10)
HashedUrls = N * log2(N / p)
BitArray = N / p
Bloom = -N / log(2) * log2(p)

plt.plot(N, HashedUrls, 'b')
plt.plot(N, BitArray, 'g')
plt.plot(N, Bloom, 'r')

plt.show()
```

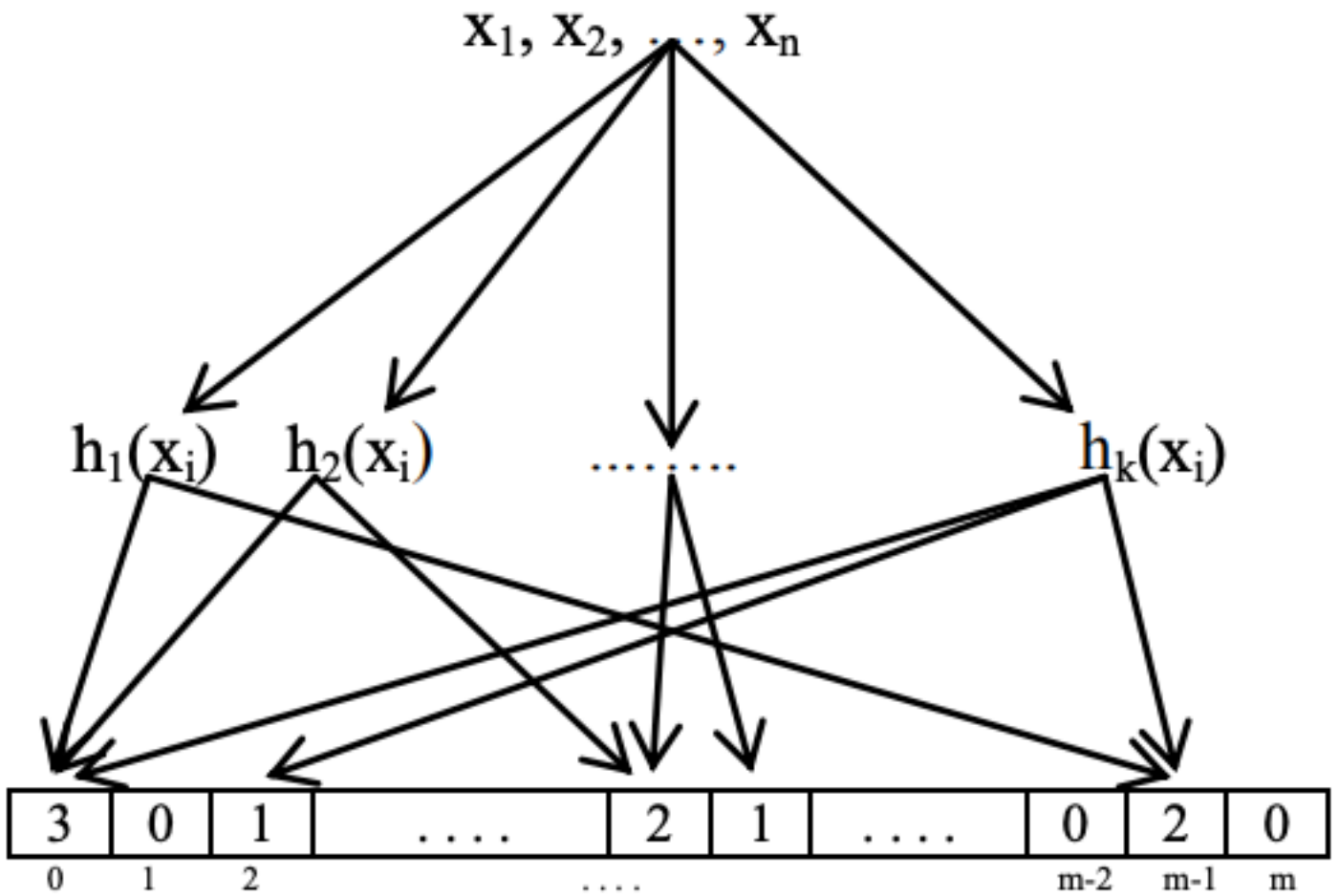


Как работает Bloom Filter



Как быть с удалением элементов?

Использовать подсчитывающий фильтр Блума. В нем вместо 1-битных булевских значений используются n-битные целые числа. Это приводит к тому, что фильтр занимает гораздо больше места, но взамен мы имеем счетчики для вставок элементов и можем удалить элементы из фильтра.



Интерактивное демо

http://jasondavies.com/bloomfilter/ (http://jasondavies.com/bloomfilter/)

Калькулятор параметров фильтра Блума

https://hur.st/bloomfilter/ (https://hur.st/bloomfilter/)

Примеры использования фильтра Блума

In [9]:

```
# Демо кода фильтра Блума
# (только для демонстрации, в продакшне не использовать!!!)
class Bloom:
    """ Bloom Filter Demo """
    def __init__(self, m, k, hash_fun):
        """
        m, size of the vector
        k, number of hash functions to compute
        hash_fun, hash function to use
        """
        self.m = m
        # initialize the vector
        # (attention: a real implementation should use an actual bit-array)
        self.vector = [0]*m
        self.k = k
        self.hash_fun = hash_fun

    def calcHash(self, key, i):
        """ 'i' works as a seed """
        return self.hash_fun(key + str(i))

    def insert(self, key):
        """ insert the key to the database """
        for i in range(self.k):
            self.vector[self.calcHash(key, i) % self.m] = 1

    def contains(self, key):
        """ check if key is contained in the database
        using the filter mechanism """
        for i in range(self.k):
            if self.vector[self.calcHash(key, i) % self.m] == 0:
                return False # the key doesn't exist
        return True # the key can be in the data set
```

In [8]:

```
# Используем фильтр
import hashlib

def hash_f(x):
    # we'll use sha256 just for this example
    h = hashlib.sha256(x.encode())
    return int(h.hexdigest(),base=16)

bloom = Bloom(100, 10, hash_f)

words_present = ['abound','abounds','abundance','abundant','accessable',
                 'bloom','blossom','bolster','bonny','bonus','bonuses',
                 'coherent','cohesive','colorful','comely','comfort',
                 'gems','generosity','generous','generously','genial']

for item in words_present:
    bloom.insert(item)

# words not added, but we'll check them anyway
```

```
words_absent = ['bluff','cheater','hate','war','humanity',
                'racism','hurt','nuke','gloomy','facebook',
                'geeksforgeeks','twitter']

test_words = words_present + words_absent
n_total_positives = 0
n_false_positives = 0
for word in test_words:
    if bloom.contains(word):
        n_total_positives += 1
        if word in words_absent:
            n_false_positives += 1
            print("'{}' is a false positive!".format(word))
        else:
            print("'{}' is present (not a false positive)!".format(word))
    else:
        print("'{}' is definitely not present!".format(word))

print("Rate of false positives: {}%".format(n_false_positives / n_total_positives
```

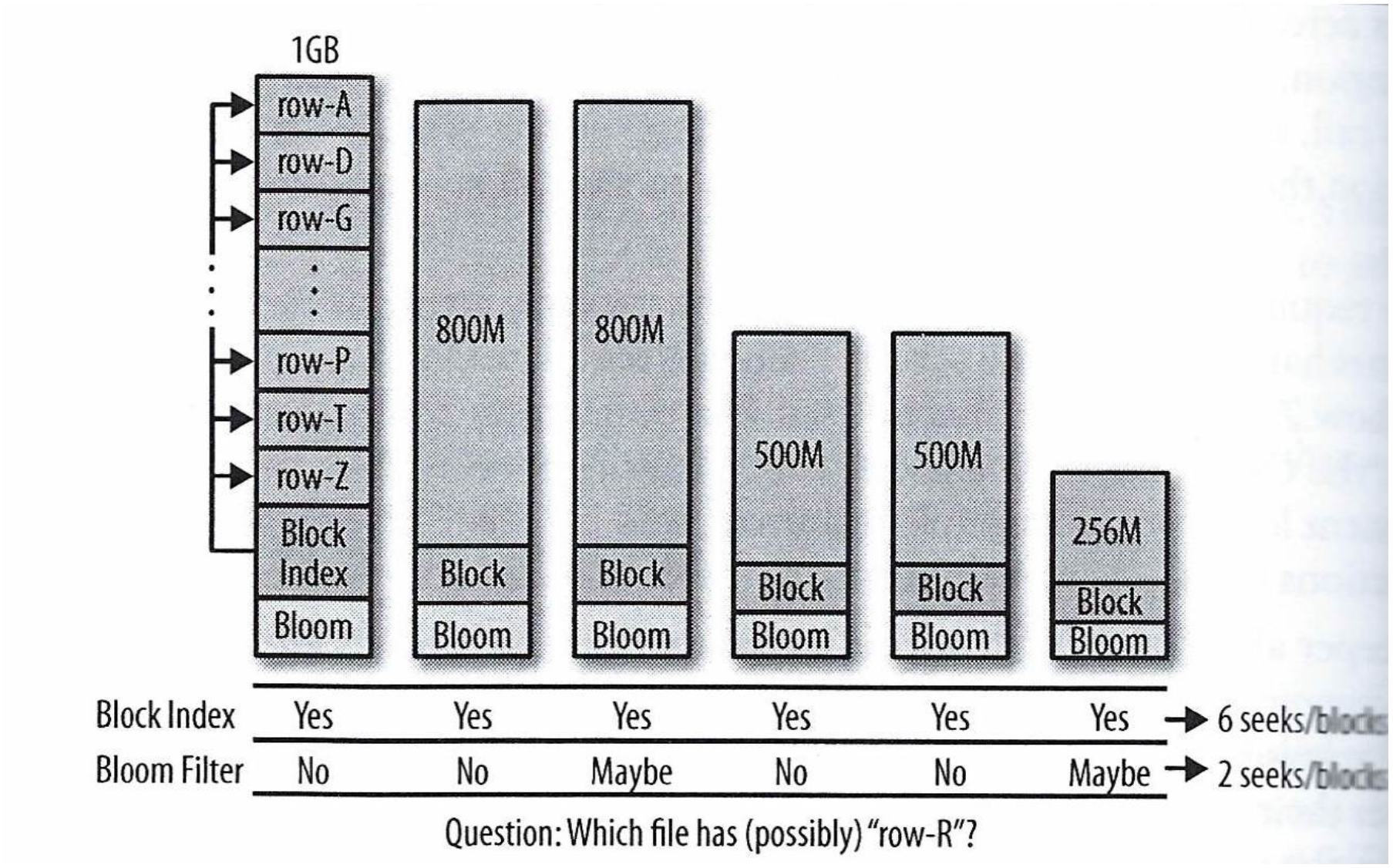
```
'abound' is present (not a false positive)!
'abounds' is present (not a false positive)!
'abundance' is present (not a false positive)!
'abundant' is present (not a false positive)!
'accessable' is present (not a false positive)!
'bloom' is present (not a false positive)!
'blossom' is present (not a false positive)!
'bolster' is present (not a false positive)!
'bonny' is present (not a false positive)!
'bonus' is present (not a false positive)!
'bonuses' is present (not a false positive)!
'coherent' is present (not a false positive)!
'cohesive' is present (not a false positive)!
'colorful' is present (not a false positive)!
'comely' is present (not a false positive)!
'comfort' is present (not a false positive)!
'gems' is present (not a false positive)!
'generosity' is present (not a false positive)!
'generous' is present (not a false positive)!
'generously' is present (not a false positive)!
'genial' is present (not a false positive)!
'bluff' is definitely not present!
'cheater' is definitely not present!
'hate' is definitely not present!
'war' is a false positive!
'humanity' is definitely not present!
'racism' is a false positive!
'hurt' is a false positive!
'nuke' is a false positive!
'gloomy' is definitely not present!
'facebook' is definitely not present!
'geeksforgeeks' is definitely not present!
'twitter' is definitely not present!
Rate of false positives: 16.0%
```

Применение фильтра Блума

- Bitcoin используем фильтры Блума для ускорения синхронизации кошельков и улучшения их безопасности;
- Google Chrome использует фильтр Блума для проверки вредоносности URL - как в нашем примере;
- Google BigTable и Apache HBase и Apache Cassandra используют фильтру Блума для уменьшения к-ва обращений к диску, чтобы не читать несуществующие записи-колонки;
- Squid Web Proxy Cache использует фильтры Блума;
- фильтр Блума используется для классификации ДНК;
- еще примеры применения на https://en.wikipedia.org/wiki/Bloom_filter (https://en.wikipedia.org/wiki/Bloom_filter).

Применение фильтра Блума в Apache HBase

Apache HBase - это колоночная СУБД. Данные хранятся в массиве записей вида "ключ+колонка+значение". Записи хранятся в store files. Каждый файл 1 Гб к примеру.



Цель: при поиске ключа уменьшить к-во операций ввода-вывода.

Store files состоят из блоков. Каждый блок 64 кбайта. Индекс находится в каждом блоке и хранит стартовый ключ блока. Всего 16384 блока в store file. То есть, всего проиндексировано 16384 ключей. Пусть ячейка занимает 200 байт. Тогда всего в store file хранится 5 млн ячеек. Ищем какой-то ключ. Вероятность, что он попадет не точно в ключ, а между стартовыми ключами двух блоков, очень велика. При обычном поиске ключа без Блум-фильтра находим блок, в котором находится данный ключ, а дальше начинаем full scan этого блока, чтобы найти данный ключ.

Как применяется Bloom Filter?

Bloom filter дает возможности сразу же узнать точно тот факт, что ключа в блоке нет. Но может дать и false positive. Данных может не быть, а фильтр скажет, что данные есть. Процент false positive обычно около 1%. Если вернули true, то скачаем и проверим весь блок. Бонус - число загрузок блока существенно уменьшится, что имеет большое значения для высоконагруженных систем.

Минусы: Надо обновлять Bloom filter при обновлении записей Добавляется оверхэд в 1 байт на 1 запись в фильтре. Если store file имеет размер 1Гб, как в нашем примере, то добавляя ключи в фильтр размера 20 байт (это полностью координаты имеющихся ячеек) имеем 51 Мбайт оверхеда. Если только row level, тогда примерно 100 кб на 1 Гб файл.

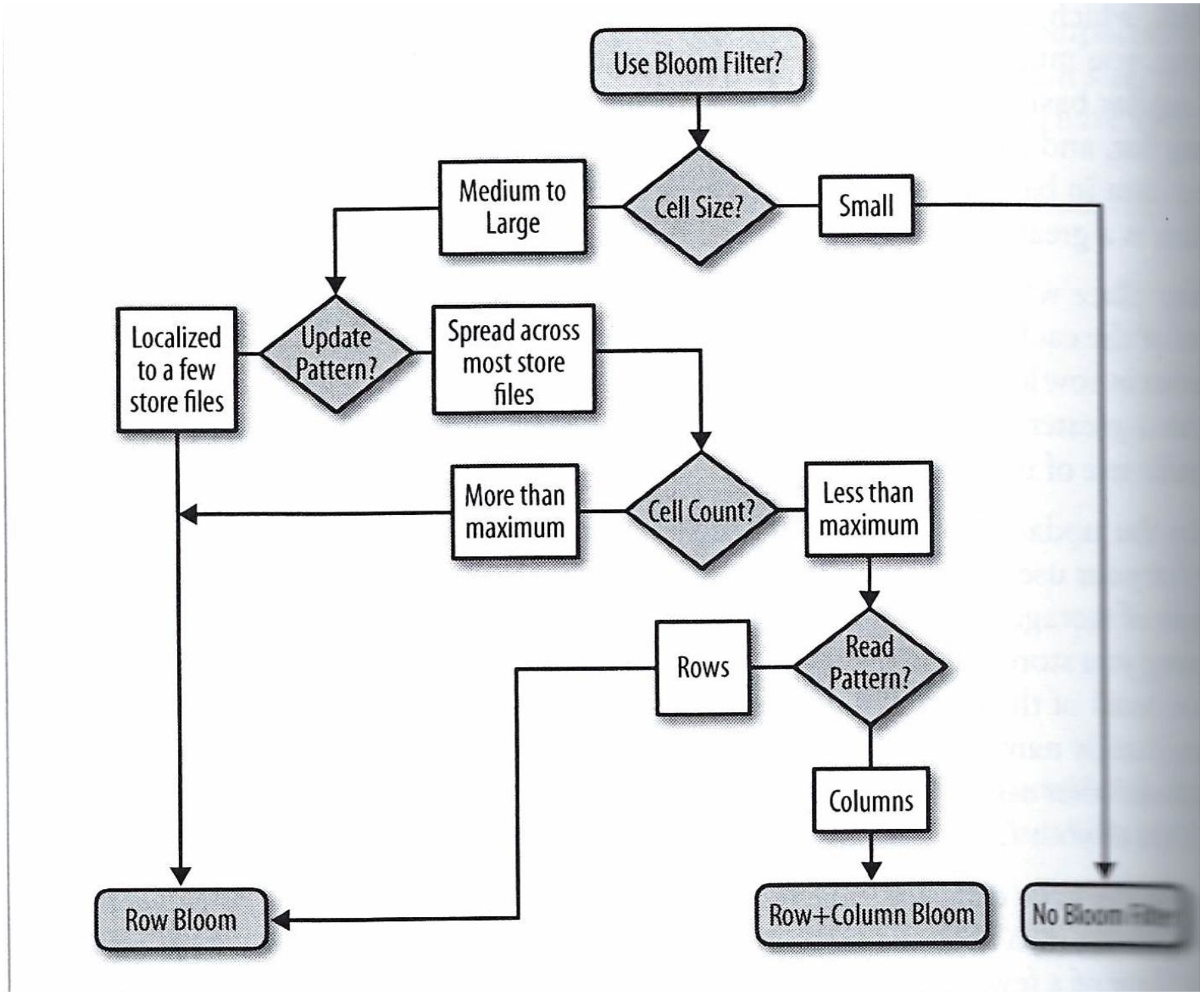
Можно использовать фильтр только для строк, можно строки + колонки. Если ищем только строки, фильтр для колонок не имеет смысла. Если колонки, то имеет смысл.

Параметры

io.hfile.bloom.enabled = Kill switch in case something goes wrong. Default = True

io.hfile.bloom.error.rate = average false positive rate. Default = 1%.

Схема использования



In [ ]:

```
>java
// Фрагменты кода фильтра Блума в HBase

public class ByteBloomFilter {
    public ByteBloomFilter(int maxKeys, double errorRate, int hashType,
        int foldFactor) throws IllegalArgumentException {
        this(hashType);

        long bitSize = computeBitSize(maxKeys, errorRate);
```

```
        hashCount = optimalFunctionCount(maxKeys, bitSize);
        this.maxKeys = maxKeys;

        // increase byteSize so folding is possible
        byteSize = computeFoldableByteSize(bitSize, foldFactor);

        sanityCheck();
    }

    public void add(byte [] buf) {
        add(buf, 0, buf.length);
    }

    public void add(byte [] buf, int offset, int len) {
        /*
         * For faster hashing, use combinatorial generation
         * http://www.eecs.harvard.edu/~kirsch/pubs/bbbf/esa06.pdf
         */
        int hash1 = this.hash.hash(buf, offset, len, 0);
        int hash2 = this.hash.hash(buf, offset, len, hash1);

        for (int i = 0; i < this.hashCount; i++) {
            long hashLoc = Math.abs((hash1 + i * hash2) % (this.byteSize * 8));
            set(hashLoc);
        }

        ++this.keyCount;
    }
```

```
    public static boolean contains(byte[] buf, int offset, int length,
        ByteBuffer bloomBuf, Hash hash, int hashCount) {

        int hash1 = hash.hash(buf, offset, length, 0);
        int hash2 = hash.hash(buf, offset, length, hash1);
        int bloomBitSize = bloomSize << 3;

        int compositeHash = hash1;
        for (int i = 0; i < hashCount; i++) {
            int hashLoc = Math.abs(compositeHash % bloomBitSize);
            compositeHash += hash2;
            if (!get(hashLoc)) {
                return false;
            }
        }
        return true;
    }
}

//Тест:
public void testBasicBloom() throws Exception {
    ByteBloomFilter bf1 = new ByteBloomFilter(1000, (float)0.01, Hash.MURMUR_HASH
    ByteBloomFilter bf2 = new ByteBloomFilter(1000, (float)0.01, Hash.MURMUR_HASH
    bf1.allocBloom();
    bf2.allocBloom();

    // test 1: verify no fundamental false negatives or positives
    byte[] key1 = {1,2,3,4,5,6,7,8,9};
    byte[] key2 = {1,2,3,4,5,6,7,8,7};
```

```
    bf1.add(key1);
    bf2.add(key2);

    assertTrue(bf1.contains(key1));
    assertFalse(bf1.contains(key2));
    assertFalse(bf2.contains(key1));
    assertTrue(bf2.contains(key2));
}
```

Домашнее задание

Будет после 3-его вебинара по вероятностным алгоритмам и структурам данных

Заполните опросник в конце занятия:

https://docs.google.com/forms/d/e/1FAIpQLSdbKCxxgd0O6G3E-dDwG0QIhFIAwvEvaTa759KNh22j83IROg/viewform (https://docs.google.com/forms/d/e/1FAIpQLSdbKCxxgd0O6G3E-dDwG0QIhFIAwvEvaTa759KNh22j83IROg/viewform)

Литература

Bloom Filter, оригинальная статья

Bloom, Burton H. (1970), "Space/Time Trade-offs in Hash Coding with Allowable Errors"

Wikipedia

https://en.wikipedia.org/wiki/Bloom_filter (https://en.wikipedia.org/wiki/Bloom_filter)

Интерактивное демо

http://jasondavies.com/bloomfilter/ (http://jasondavies.com/bloomfilter/)

Калькулятор расчета параметров Bloom Filter

https://hur.st/bloomfilter/ (https://hur.st/bloomfilter/)

Data mining: Bloom Filter, other algos & datastructures

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org (http://www.mmds.org)

Спасибо
за внимание!


OTUS
ОНЛАЙН-ОБРАЗОВАНИЕ

Фильтр Блума

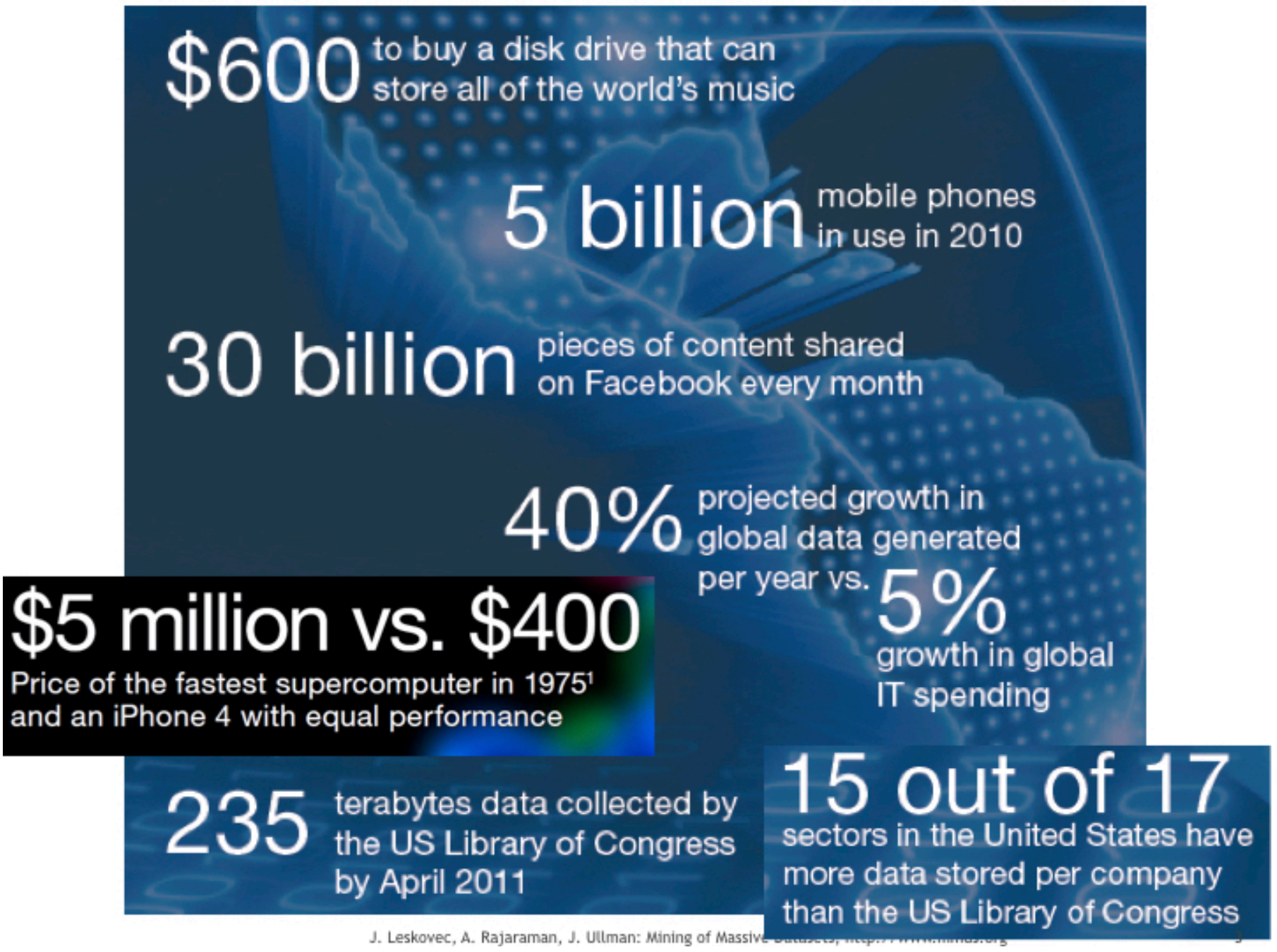
Как он устроен и его использование

Михаил Горшков
разработчик

ВКЛЮЧИТЬ ЗАПИСЬ!!!

Заполните опросник перед началом занятия:
https://docs.google.com/forms/d/e/1FAIpQLSedzn_AdN6MF1zTTD9oBArcT9dEZJMopXeBtUXhKvJ5v68wrw
(https://docs.google.com/forms/d/e/1FAIpQLSedzn_AdN6MF1zTTD9oBArcT9dEZJMopXeBtUXhKvJ5v68wrw

Data contains value and knowledge



Ближайшие 3 вебинара - про работу с "большими данными".

Тема бигдата пользуется в наши дни особой популярностью.

Рассмотрим и алгоритмы, и структуры данных для работы с большими данными.

"Большие данные" как правило не помещаются в оперативную память. Это требует переосмысления даже таких базовых алгоритмов, как подсчет числа уникальных элементов и принадлежность элемента множеству.

Коснемся небольшого числа тем. Будем рассматривать:

- Задача принадлежности элемента множеству - *Bloom Filter*;

- Поиск похожих данных - *MinHash*, *SimHash*;
- Подсчет количества вхождений элемента в множество - *Count Min Sketch*;
- Сколько уникальных элементов содержит множество - *HyperLogLog*.

Эти алгоритмы и структуры данных вероятностные. Почему вероятностные? Что это значит?

Что такое "вероятностные алгоритмы"?

Perfect is the enemy of good. Voltair

Что делают с большими данными?

- Hadoop, Spark;
- вероятностные алгоритмы и структуры данных.

Пример. Подсчет числа "хитов" разных видео (NetFlix, YouTube).

unique id -> view count

- делаем хэш-таблицу;
- требуемое место - O(n);
- 4 млрд видео, каждое видео имеет 32-битный уникальный id (это минимум), тогда нужно 16 Гб только чтобы хранить id! Затем счетчик—32-битное целое - еще 16 Гб. Итого 32 Гб! Не помещается в RAM.

Можно использовать Spark, но есть подход лучше - вероятностный. Задача решается алгоритмом *HyperLogLog*.

Ввиду наличия большого к-ва данных иногда приходится идти на компромиссы—результат вместо точного получается с некоей вероятностью. Точное решение некоторых задач является невозможным или нерациональным: слишком дорогим, требующим слишком много ресурсов и т.п. Немного жертвуя точностью можно получить огромный выигрыш в производительности и/или ресурсах. Для повседневных задач могут быть избыточны, хорошо подходит для больших данных.

Вероятностный алгоритм - не значит, что алгоритм не рабочий или работает через раз.

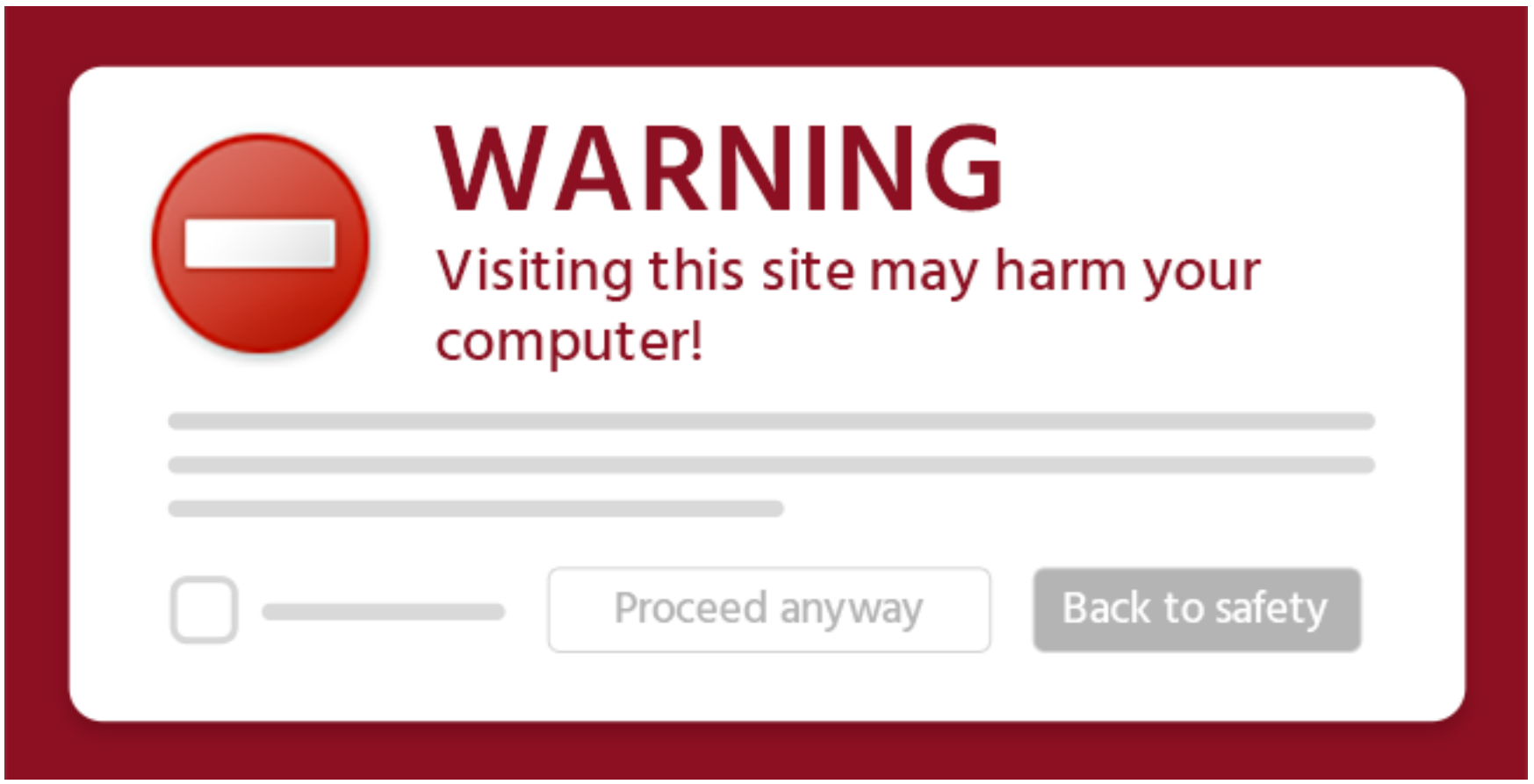
Алгоритм:

- всегда работает;
- результат может вычисляться с погрешностью, но всегда имеется ее точная оценка.

Рассмотрим этот подход на примере другой задачи.

Фильтр Блума

Постановка задачи



Вы - разработчик веб-браузера. Пользователи заходят на разные сайты. Вам нужно предупреждать пользователя о том, что URL, на который он зашел - вредоносный.

Пример URL: http://domain.com/page (http://domain.com/page).

Количество доменов - 330 млн. Возможны несколько URL на одном домене. Как это сделать? Ваши варианты. Каковы проблемы, с которыми вам придется столкнуться при решении этой задачи?

Требования к структуре данных

Основное назначение - определять, есть ли заданный объект в заданном множестве.

API структуры данных:

- добавить объект: операция **add**;
- определить, содержит ли она объект: операция **has**;
- удалить объект: операция **delete**.

Варианты решения:

- Хранить все URL в массиве (сортированном) и искать URL бинарным поиском. Например, у нас 1 млн зловредных доменов, и на каждый нужно 50 байт. Итого нужно 50 Мб. Вставка - сложность O(N). Поиск занимает O(log(N)). Место - O(N);
- Бинарное дерево поиска. Вставка - O(log(N)). Поиск O(log(N)). Место - O(N);
- Хэш-таблица. Вставка - худшее время O(N), поиск - худшее время O(N). Место - O(N). Идеальное хэширование? Не подходит, потому множество ключей не является заранее заданным и статическим.

Контейнер	Вставка (avg)	Вставка (worst)	Поиск (avg)	Поиск (worst)	Место
Сортированный массив	O(N)	O(N)	O(log(N))	O(log(N))	O(N)
Дерево	O(log(N))	O(N)	O(log(N))	O(log(N))	O(N)
Хэш-таблица	O(1)	O(N)	O(1)	O(N)	O(N)

Проблема со всеми реализациями - требуется O(N) места (надо хранить все URL-ы) и поиск в худшем случае O(N).

Готовы пожертвовать точностью

НО!!!

- готовы ошибаться, но с заранее известным и небольшим % ошибок;

Почему заранее известным? Если процент не известен, придется очень часто перепроверять.

Попробуем массив

Начнем с варианта простого массива, но будем хранить там хэшированные URL-ы (можно отсортировать его по значению хэш-функции для log-поиска).

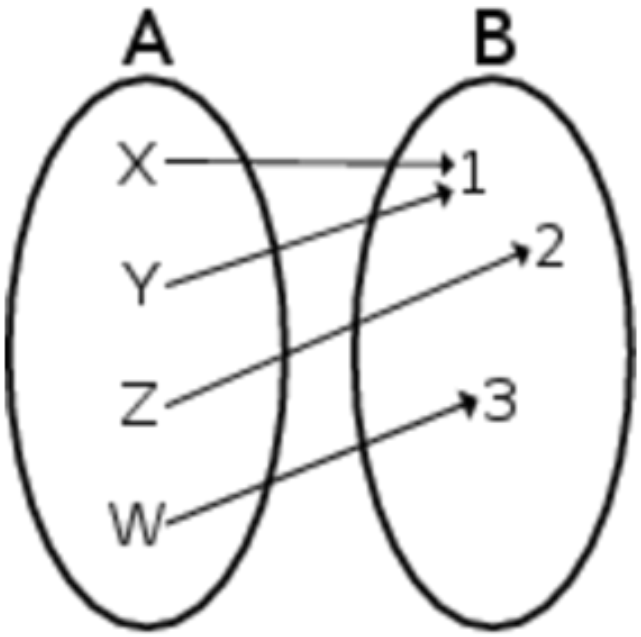
- Для каждого URL U подсчитаем хэш-функцию h(U_i). Пусть результат хэш-функции будет всегда состоять из m бит.
- Будем хранить хэшированные URL-ы в обычном массиве.
- Массив будет представлять собой множество S_{hashed}: (h(U_0), h(U_1),...h(U_{N-1}))
- Размер массива m * N, но коэффициент уже будет меньше, чем для просто массива URL-ов.
- Чтобы проверить, является ли URL вредоносным, посчитаем от него хэш-функцию h(U) и поищем результат в массиве. Если есть, значит URL, возможно, вредоносный. Если нет, то точно не вредоносный.

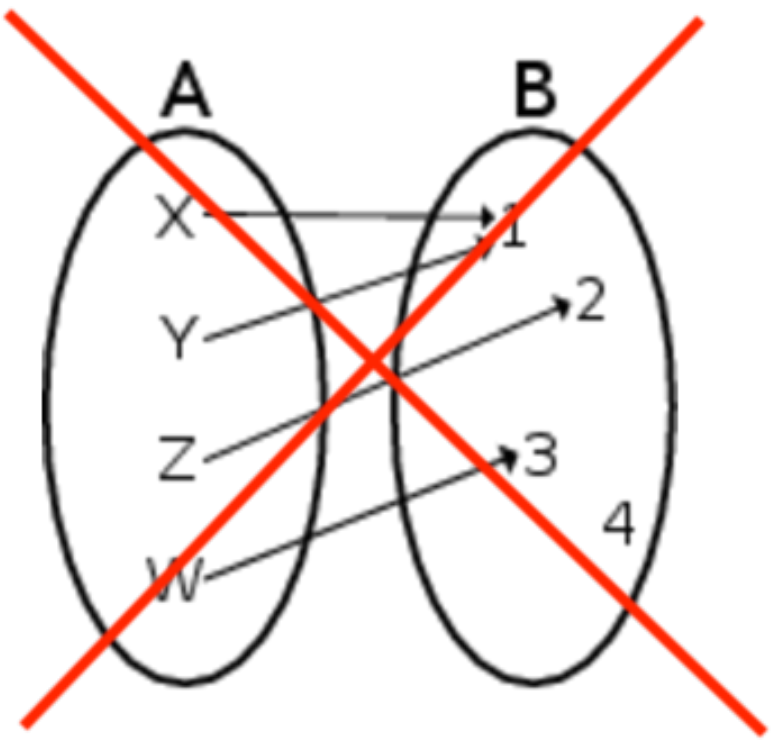
Вопрос 1: почему "возможно"?

Вопрос 2: бывают ли false negatives в данном случае?

Хэш - сюръективная функция

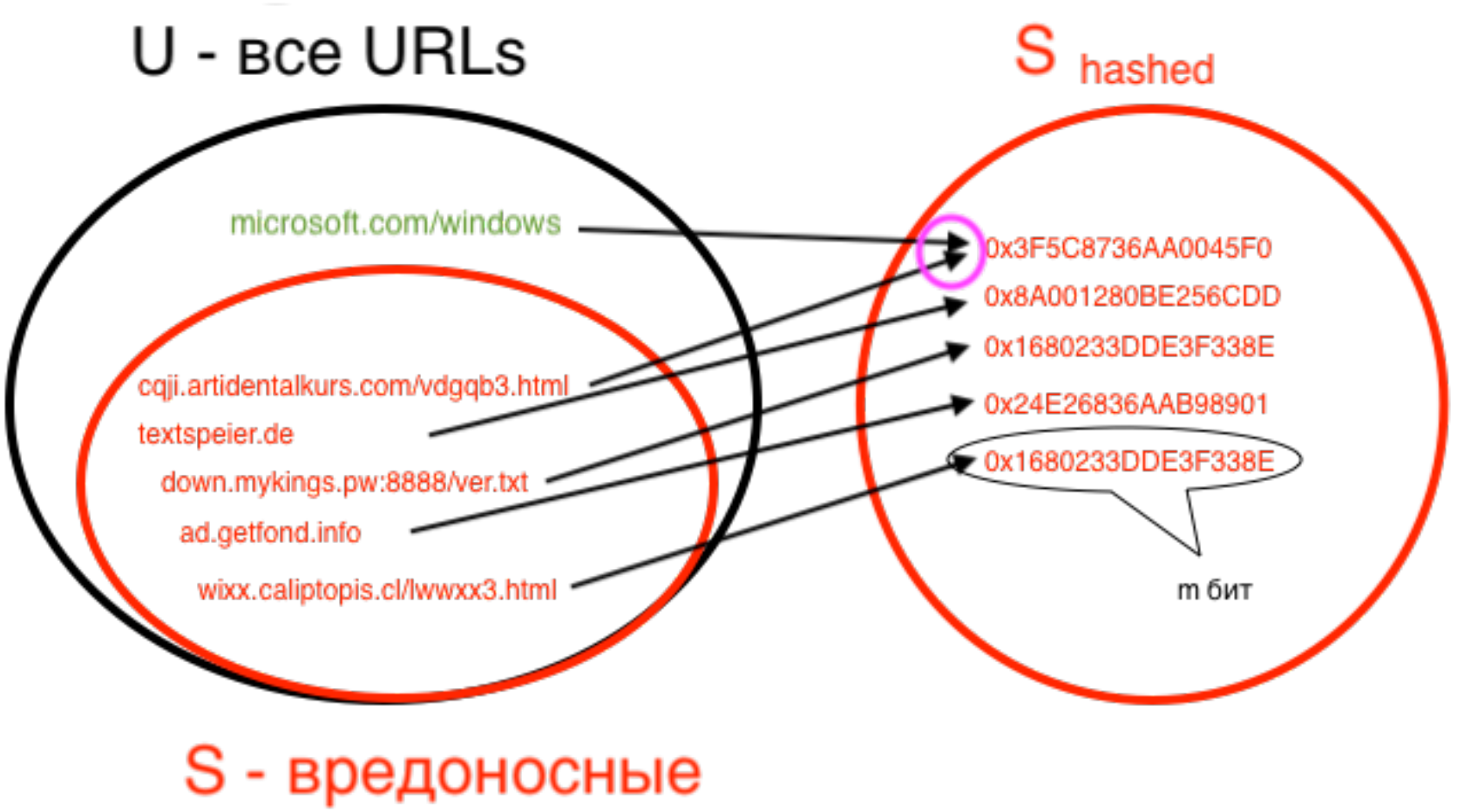
Сюръекция - это функция f : A → B такая что для каждого элемента b в множестве B найдется как минимум один элемент a в множестве A такой что f(a)=b.





Поэтому не может быть false negatives.

Почему могут быть false positives?



- Опасность вернуть неверный результат появляется в случае коллизии, то есть если $h(U) = h(U_i)$, но сам $U \neq U_i$.
- Чем больше m, тем меньше вероятность коллизии, следовательно меньше вероятность ложного срабатывания (false positive).
- Мы можем сделать сколь угодно малой вероятность ложного срабатывания за счет увеличения m.

Небольшой экскурс в теорию вероятностей

Элементарным исходом (или элементарным событием) называют любой простейший (т.е. неделимый в рамках данного опыта) исход опыта.
Пример. Выпадение герба после подбрасывания монеты.

Множество всех элементарных исходов будем называть **пространством элементарных исходов**.
Пример. Шесть граней игральной кости.

Любой набор элементарных исходов называют **событием**.
Пример. Выпадение четного числа на игральной кости

Вероятность события будем представлять числом p, которое принимает значение от 0 до 1.

Некоторые свойства:

- $p(x)=1$ - означает, что мы уверены, что событие точно произойдет;
- $p(y)=0$ - означает, что мы точно уверены, что событие не произойдет;
- $\sum_x p(x) = 1$ - сумма вероятностей по пространству элементарных исходов равна 1;
- если p - вероятность того, что событие произойдет, то вероятность того, что событие не произойдет, равна 1 - p.
- вероятность совместного исхода двух событий $p(x, y) = p(x\ and \ y)$ (например бросаем кость и одновременно выпадет четное и простое число);
- если $p(x, y) = p(x)\ p(y)$, то события называются независимыми.

Вопрос: независимы ли события "выпадает четное число на кости" и "выпадает простое число на кости"?

$p_{even} = \frac{1}{2}$, $p_{prime} = \frac{1}{2}$;

Какова вероятность совместного исхода двух этих событий?

Ваши варианты в чат*

$p_{even\ and\ prime} = ?$

Почему вероятность false positive может быть сколь угодно малой?

Пусть P - вероятность верного определения, что URL U НЕ находится в нашей структуре данных S. Это означает, что нет false positive для некоторого конкретного URL.

То есть, с вероятностью P для всех $0 \leq i \leq N - 1$

$h(U_i) \neq h(U)$

Тогда $p = 1 - P$ - вероятность ложного срабатывания.

У нас в хэше m бит, всего возможны 2^m значений хэша.

Вероятность, что $h(U_i) = h(U)$ для конкретного i при хорошо выбранной хэш-функции (значения равномерно распределены) равна $\frac{1}{2^m}$.

Вероятность, что $h(U_i) \neq h(U)$ для любого i равна $1 - \frac{1}{2^m}$, и все эти события независимы для всех N хэшей.

Тогда $\mathbf{p = 1 - (1 - \frac{1}{2^m})^N}$.

То есть, чтобы обеспечить сколь угодно малую наперед заданную вероятность ложного срабатывания p, необходимо правильно подобрать m.

$p \to 0$ при $m \to \infty$.

Сколько нужно памяти на всю структуру данных, чтобы обеспечить заданную вероятность ложного срабатывания?

Обозначим к-во памяти в битах на всю структуру данных через M. Найдем M как функцию от p и N.

$M = Nm$ бит.

$$m = \frac{M}{N}$$

Подставляя в предыдущее уравнение, получаем

$$p = 1 - (1 - \frac{1}{{2}^\frac{M}{N}})^N$$

Вычисляем M:

$$(1 - \frac{1}{{2}^\frac{M}{N}})^N = 1 - p$$

$$1 - \frac{1}{{2}^\frac{M}{N}} = (1 - p)^\frac{1}{N}$$

$$\frac{1}{2^\frac{M}{N}} = 1 - (1 - p)^\frac{1}{N}$$

$$2^\frac{M}{N} = \frac{1}{1 - (1 - p)^\frac{1}{N}}$$

$$\frac{M}{N} = log_2\frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

$$M = N log_2\frac{1}{1 - (1 - p)^{\frac{1}{N}}}$$

Столько памяти (в битах) нужно на всю структуру данных.

Если задаться достаточной малой вероятностью p, например, 1%, то можно доказать, что

$$M \approx N log_2 \frac{N}{p}$$

$$((1–x)^\frac{1}{N} \approx 1 - \frac{x}{N}, x \to 0)$$

На каждый хэш требуется $log_2\frac{N}{p}$ бит.

Итоги варианта с хэшированием

По памяти лучше, чем просто хранить все URL, но имеются false positives

Как бороться с false positives?

Ваши варианты в чат

Как бороться с false positives?

- Хранить отдельно список URL, для которых известно, что наша хэш-функция выдает false positives. В случае положительного ответа дополнительно проверяем этот ответ по списку false positives. Если он там есть, то возвращаем false вместо true.
- Все положительные ответы проверять альтернативным способом (например, проверять их с помощью внешнего сервиса).

Улучшим вариант с хэш-таблицей: используем битовый массив

Предположим, нам нужно каким-то образом хранить подмножество S чисел от 0 до 999. Например, *{2, 35, 878, 999}*.

Варианты те же: массив, дерево, хэш-таблица, но можно еще битовый массив.

Заведем битовый массив из 1000 бит. Каждый бит с номером i - это признак, есть ли число i в нашем подмножестве. Если все биты нулевые, подмножество пусто. Если все биты установлены в 1 - то это все числа от 0 до 999.

Добавление, и удаление числа из множества тривиальны: нужно просто выставить/убрать соответствующий бит.

Для проверки принадлежности числа X множеству нужно проверить, установлен ли бит с номером X.

Независимо от размера подмножества размер нашего массива всегда 1000 бит. Если бы мы хранили, скажем, наше подмножество как массив 32-битных целых, понадобилось бы 32 * S бит, где S - размер подмассива.

Если S маленькое, выгоднее хранить как массив 32-битных целых. Если большое, то выгоднее использовать битовый массив.

Комбинируем битовый массив с хэш-функцией

Нам нужен был битовый массив фактически для нумерации элементов, но сами элементы могут быть любыми объектами. Однако их количество должно быть невелико.

Чтобы нумеровать элементы множества, можем использовать хэш-функцию.

Пусть снова $h(x)$ - хэш-функция, возвращающая m бит.

Пусть имеем множество $S = \{x_0, x_1, ..., x_{N - 1}\}$.

Представим его как битовый массив из 2^m элементов.

Рассматриваем $h(x_i)$ как число из диапазона $\{0, 1, 2^{m - 1}\}$. Для каждого x_i выставим в 1 элемент с номером $h(x_i)$ массива.

Можем добавить x в множество, установив соответствующий бит (с номером $h(x)$). Можем удалить из множества, сняв бит. Можем проверить принадлежность элемента множеству, посчитав хэш и проверив бит.

Проблема опять та же - false positives. То есть, $h(x) = h(x_i)$ для какого-то i, но $x \neq x_i$. Как мы выяснили ранее, вероятность false positive

$$p = 1 - (1 - \frac{1}{2^m})^N$$

Размер всей структуры данных в битах $M = 2^m$ (ранее было $M = Nm$).

Подставляем $M = 2^m$ в предыдущее уравнение

$$p = 1 - (1 - \frac{1}{M})^N$$

Аналогично выведенному ранее

$$M = \frac{1}{1 - p ^ {\frac{1}{N}}}$$

и

$$M \approx \frac{N}{p}$$

Нам требуется небольшие значения p, следовательно M будет большим.

Надо минимизировать M.

В предыдущей схеме было $N log_2\frac{N}{p}$.

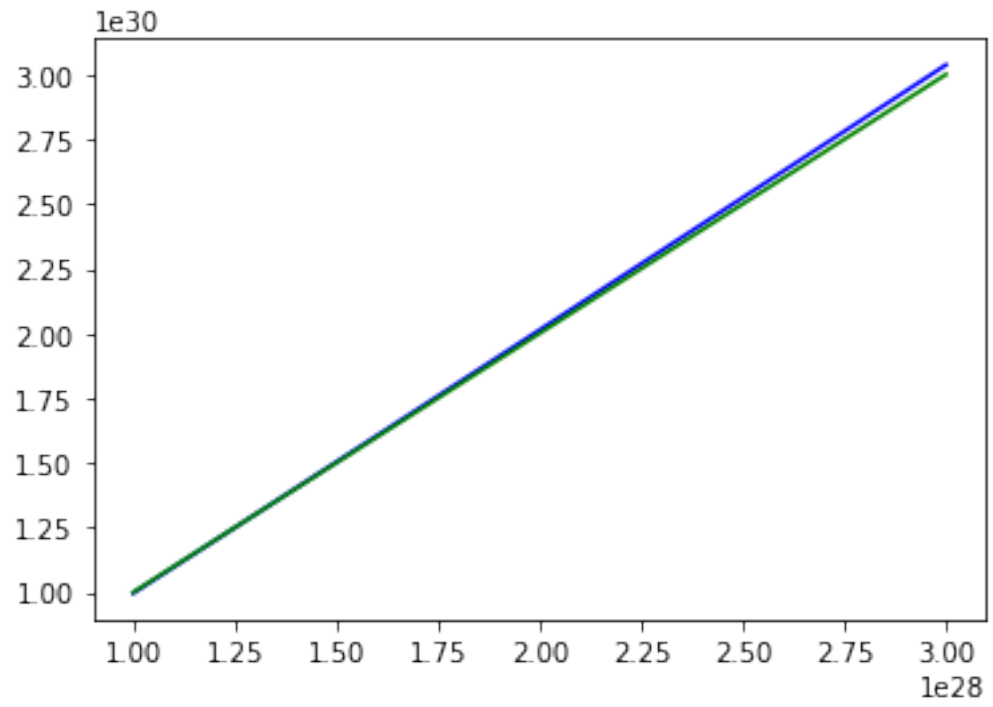
Сравним два подхода

При N был коэффициент $log_2\frac{N}{p}$, стал $\frac{1}{p}$. Чем меньше коэффициент, тем лучше, потому что меньше M.

Если N - очень большое, то текущий подход лучше. Например, если p = 0.01, то текущий подход лучше, если $N > 1.28 * 10^{28}$

Схема	M
Массив хэшированных URL	$Nlog_2\frac{N}{p}$
Битовый массив с хэш-функцией	$\frac{N}{p}$

In [27]:



Почему второй подход не так эффективен и что с этим делать?

Вопрос к аудитории: почему второй подход оказался не таким эффективным?

Первая схема использует mN бит памяти, вторая - 2^m бит. Вторая схема использует экспоненциально больше памяти при фиксированном N. При этом вероятность ложного срабатывания одинакова.

На первый взгляд, вторая идея неудачна. Но ее можно использовать в несколько ином ключе.

Множественная фильтрация

Назовем хеширование в битовый массив "фильтром". Используем фильтрацию несколько раз.

Используем k различных независимых друг от друга хэш-функций: $h_0, ..., h_{k-1}$. Каждая хэширует элемент нашего множества в m-битный результат. Наша структура данных будет состоять из k битовых массивов, каждый содержит 2^m бит.

Общий размер структуры данных $M = k2^m$ бит.

Как работает API нашей структуры данных?

- **add**. Считаем $h_0(x)$, записываем 1 в $h_0(x)$-й бит 0-го массива. Считаем $h_1(x)$, записываем 1 в $h_1(x)$-й бит 1-го массива и т.д.
- **has**. Посчитаем хэши от целевых данных $h_i(x)$. Проверим, установлен ли бит с номером $h_i(x)$ в i-м фильтре.

В случае, если каждый фильтр показывает отсутствие бита, x отсутствует в нашем контейнере. Если бит где-то установлен, значит данные имеются в нашем контейнере.

- **delete**.

Вопрос к аудитории: как сделать delete?

Посчитаем вероятность false positive в этом случае.

$p_i = 1 - (1 - \frac{1}{2^m})^N$ для каждого фильтра.

Поскольку фильтры независимы, общая вероятность false positive

$p = (1 - (1 - \frac{1}{2^m})^N) ^ k$

Общий размер структуры данных $M = k2^m$ бит.

Подставляя в предыдущее уравнение, получаем, что

$M = \frac{k}{1 - (1 - p^\frac{1}{k}) ^ \frac{1}{N}}$

$p ^ \frac{1}{k} << 1 => M \approx \frac {k {N}}{{p} ^ \frac{1}{k}}$

Стратегия более эффективна по памяти. Если $k = 2$, $M \approx k \sqrt{p}$.

Число ошибок экспоненциально уменьшается с увеличением k.

В случае схемы с одним фильтром к-во ошибок линейно уменьшается с ростом к-ва бит. Ранее выведенная формула $p = 1 - (1 - \frac{1}{2^m})^N$.

Если $\frac{1}{2^m}$ мало, то вероятность ошибки обратно пропорциональна N.

Перекрывающиеся фильтры

Вместо k массивов по 2^m бит будем использовать один массив на 2^m бит. Как работают наши операции теперь?

- **add**. Считаем h_0(x), записываем 1 в h_0(x)-й бит массива. Считаем h_1(x), записываем 1 в h_1(x)-й бит массива и т.д.
- **has**. Посчитаем хэши от целевых данных h_i(x). Проверим, установлены ли биты с номерами h_i(x) в массиве.

В случае, если все биты сняты, x отсутствует в нашем контейнере. Если бит где-то установлен, значит данные имеются в нашем контейнере.

- **delete**. Посчитать хэши, снять соответствующие биты.

Вопрос к аудитории: почему это не будет работать?

Какова вероятность false positive?

Предположим, что x отсутствует в контейнере. Ложное срабатывание происходит, когда h_0(x) = h_{i_0}(x_{j_0}) для некоторых i_0 и j_0, h_1(x) = h_{i_1}(x_{j_1}) для некоторых i_1 и j_1 и т.д. для остальных хэш-функций h_2...h_{k-1}.

Это независимые события, следовательно вероятность их совместного появления - это вероятность их произведения.

Интуитивно понятно, что каждое индивидуальное событие имеет одну и ту же вероятность, поэтому найдем вероятность для h_0. То есть, найдем вероятность события h_0(x) = h_{i_0}(x_{j_0}) для некоторых i_0 и j_0 и возведем ее в k-ую степень, чтобы получить общую вероятность.

Вероятность того, что h_0(x) = h_{i_0}(x_{j_0}) для некоторых i_0 и j_0 равна 1 - вероятность того, что h_0(x) \neq h_{i_0}(x_{j_0}) для всех i_0 и j_0. Это независимые события, вероятность каждого - 1 - \frac{1}{2^m}, следовательно

p(h_0(x) = h_{i_0}(x_{j_0})) для некоторых i_0 и j_0 = 1 - (1 - \frac{1}{2 ^m}) ^ {kN}, так как есть kN пар различных значений (i_0, j_0). Следовательно,

p = (1 - (1 - \frac{1}{2^m}) ^ {kN}) ^k

2^m = M, следовательно,

p = (1 - (1 - \frac{1}{M}) ^ {kN}) ^k

M = \frac{1}{1 - (1 - p^\frac{1}{k}) ^ \frac{1}{kN}}

Похоже на результат для повторяющихся фильтров.

Так как p^{\frac{1}{k}} << 1,

M \approx \frac{kN}{p^\frac{1}{k}}

Можно доказать, что

\frac{1}{1 - (1 - p^\frac{1}{k}) ^ \frac{1}{kN}} < \frac{k}{1 - (1 - p^\frac{1}{k}) ^ \frac{1}{N}}

Перекрывающиеся фильтры лучше, чем множественная фильтрация и намного лучше, чем одиночный фильтр (1.28*10^{28}).

Схема	M
Массив хэшированных URL	Nlog_2\frac{N}{p}
Битовый массив с хэш-функцией	\frac{N}{p}
Перекрывающиеся фильтры	\frac{kN}{p^\frac{1}{k}}

Фильтр Блума

То же самое, что перекрывающиеся фильтры, но хэшируем в массив из M бит (а не в 2^m бит).

Аналогично предыдущим выкладкам

p = (1 - (1 - \frac{1}{M}) ^ {kN})^k

и

M = \frac{1}{1 - (1 - p^\frac{1}{k}) ^ \frac{1}{kN}}

Какое значение k выбрать, чтобы минимизировать к-во бит M для данной вероятности p и размера N?

Не будем это доказывать, но

k \approx \frac{M}{N} ln 2

минимизирует p. ln - натуральный логарифм.

После подстановки в предыдущее равенство получаем

M = -\frac{N log_2 p}{ln 2}

Это лучший показатель для M из тех вариантов структур данных, что мы рассмотрели.

В частности,

m = \frac{1}{ln 2} log_2\frac{1}{p} <= log_2 \frac{N}{p} (столько было для массива хэшей)

Bloom Filter требует \frac{1}{ln 2} log_2\frac{1}{p} \approx 1.44 * log_2\frac{1}{p} бит на элемент.

Доказано, что любая структура, поддерживающая операции add и test, потребует как минимум log_2\frac{1}{p} бит на элемент.

Bloom Filter близок к оптимальному.

N заранее не известно

Как правило, N не известно заранее, поэтому выбираем максимальный размер множества, который мы хотим представить - n.

$$M = -\frac{n \log_2 p}{\ln 2}$$

и

$$k = \ln\frac{1}{p}$$

Подводим итог по Bloom Filter

Пусть хотим Bloom Filter с емкостью n и вероятностью p для false positives.

Тогда выбираем $k = \ln\frac{1}{p}$ независимых хэш-функций $h_0, h_1,...,h_{k-1}$.

Каждая хэш-функция имеет диапазон 0 - $M - 1$, где M - общее число бит в фильтре.

$$M = -\frac{n \log_2 p}{\ln 2}$$

Нумеруем биты от 0 до $M-1$.

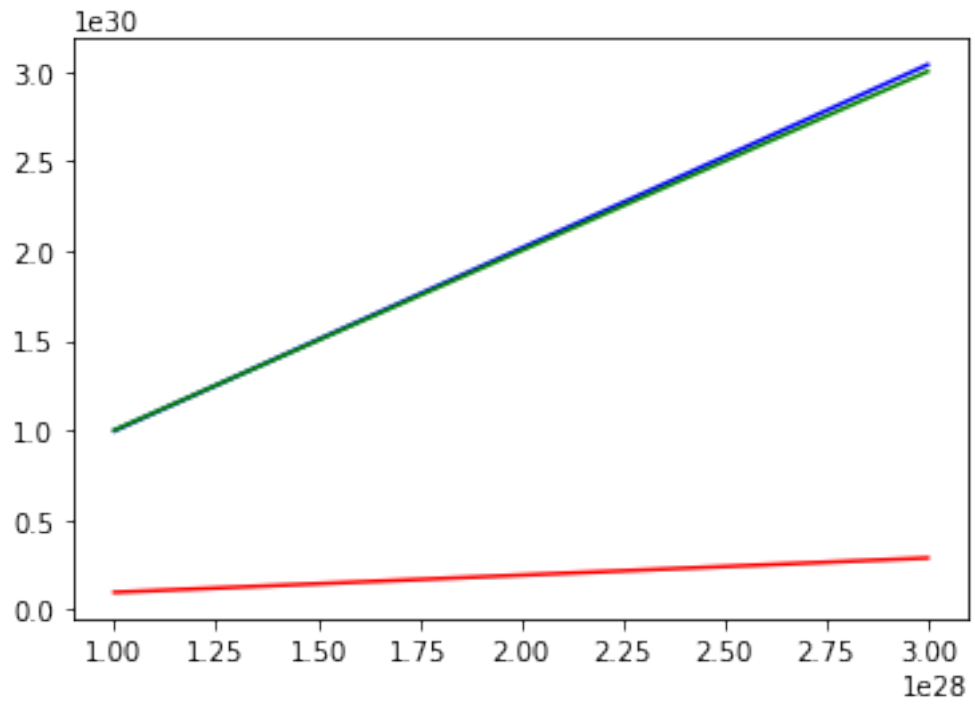
При добавлении элемента x устанавливаем биты $h_0(x), h_1(x),...,h_{M-1}(x)$ в фильтре.

Чтобы проверить наличие элемента проверяем, что все биты $h_0(x), h_1(x),...,h_{M-1}(x)$ установлены.

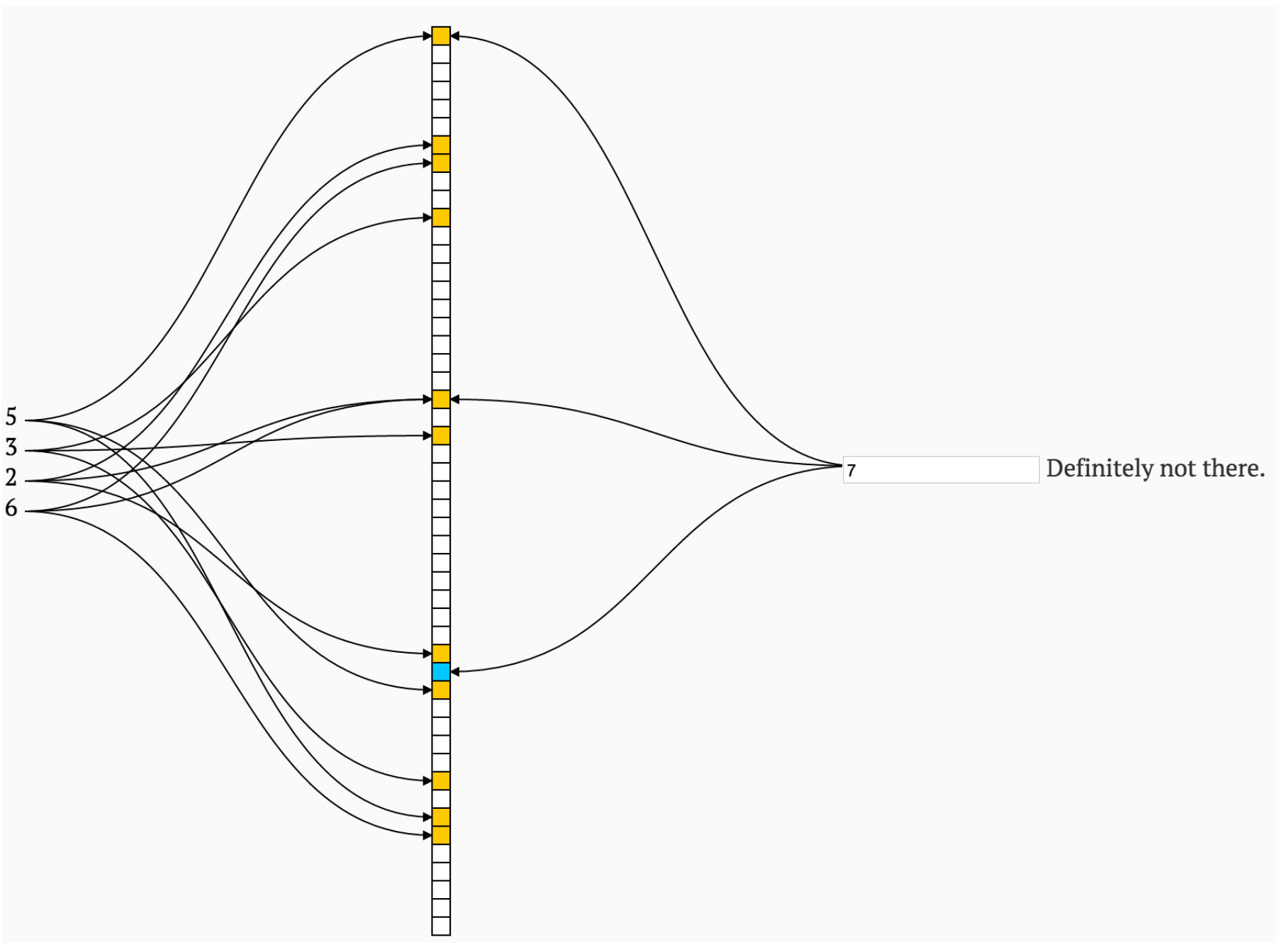
Итоговая таблица по вычислению размера M.

Схема	M
Массив хэшированных URL	$N\log_2\frac{N}{p}$
Битовый массив с хэш-функцией	$\frac{N}{p}$
Перекрывающиеся фильтры	$\frac{kN}{p^\frac{1}{k}}$
Bloom Filter	$-\frac{n \log_2 p}{\ln 2}$

In [36]:

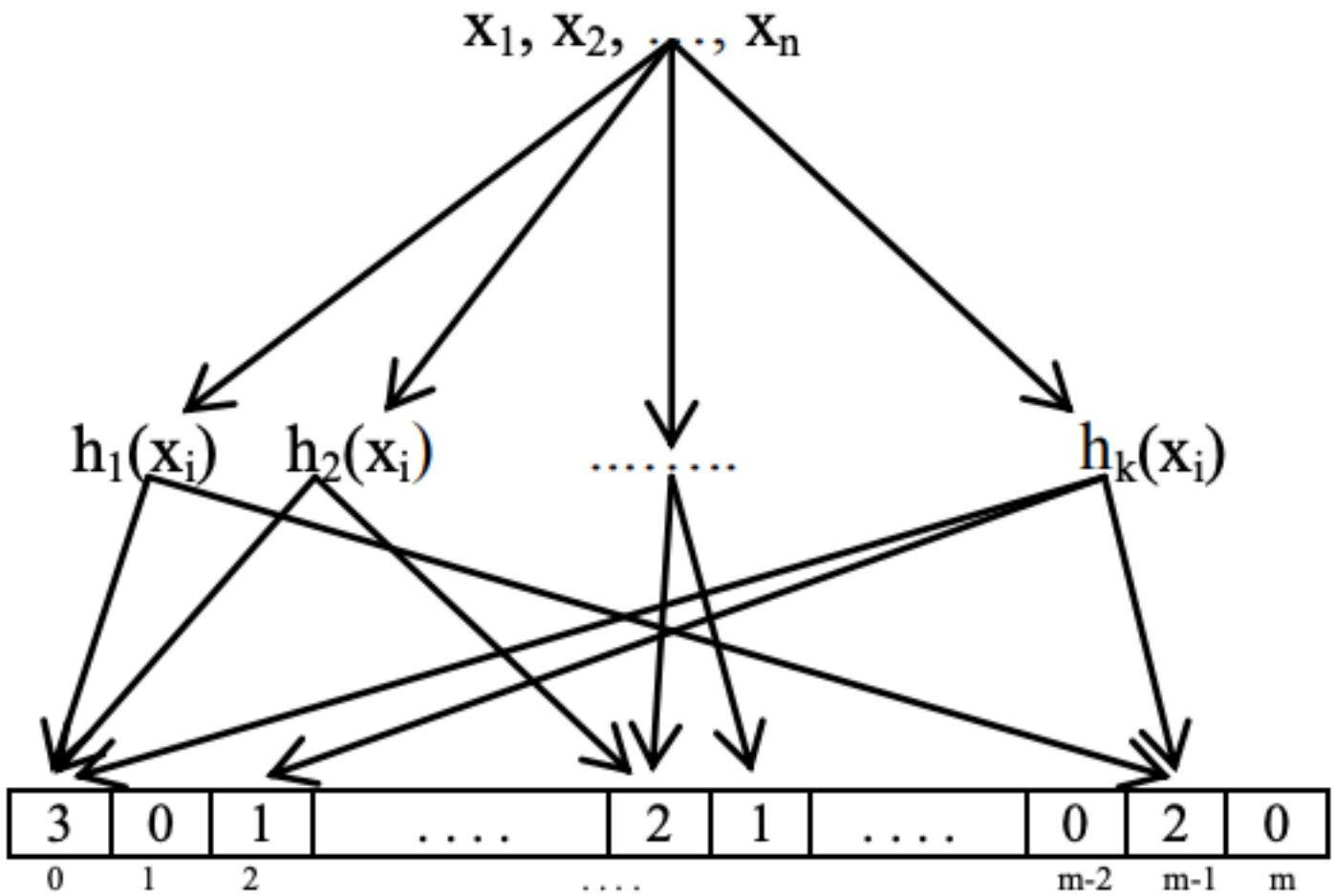


Как работает Bloom Filter



Как быть с удалением элементов?

Использовать подсчитывающий фильтр Блума. В нем вместо 1-битных булевских значений используются n-битные целые числа. Это приводит к тому, что фильтр занимает гораздо больше места, но взамен мы имеем счетчики для вставок элементов и можем удалить элементы из фильтра.



Интерактивное демо

http://jasondavies.com/bloomfilter/ (http://jasondavies.com/bloomfilter/)

Калькулятор параметров фильтра Блума

https://hur.st/bloomfilter/ (https://hur.st/bloomfilter/)

Примеры использования фильтра Блума

In [9]:

In [8]:

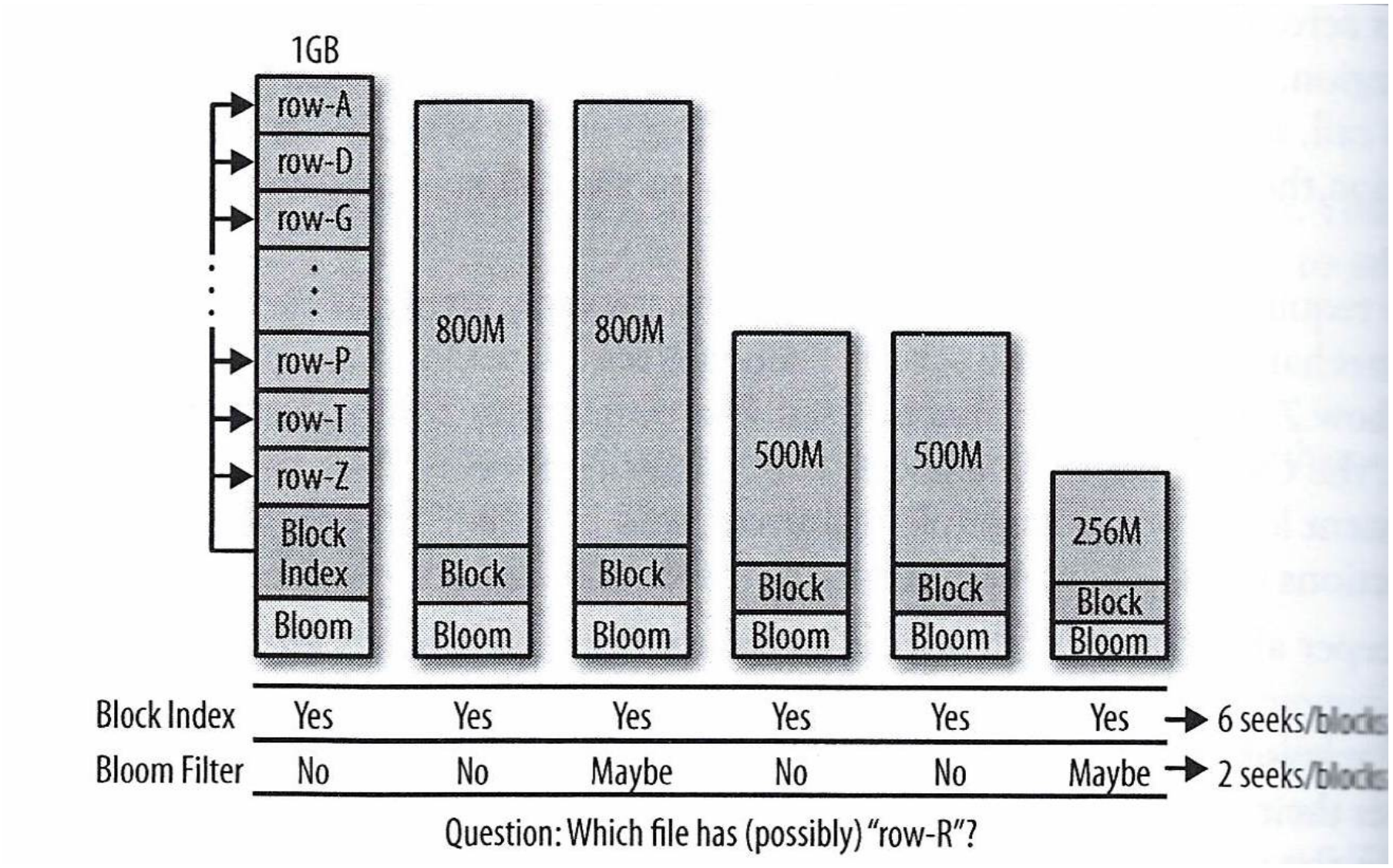
```
'abound' is present (not a false positive)!
'abounds' is present (not a false positive)!
'abundance' is present (not a false positive)!
'abundant' is present (not a false positive)!
'accessable' is present (not a false positive)!
'bloom' is present (not a false positive)!
'blossom' is present (not a false positive)!
'bolster' is present (not a false positive)!
'bonny' is present (not a false positive)!
'bonus' is present (not a false positive)!
'bonuses' is present (not a false positive)!
'coherent' is present (not a false positive)!
'cohesive' is present (not a false positive)!
'colorful' is present (not a false positive)!
'comely' is present (not a false positive)!
'comfort' is present (not a false positive)!
'gems' is present (not a false positive)!
'generosity' is present (not a false positive)!
'generous' is present (not a false positive)!
```

Применение фильтра Блума

- Bitcoin используем фильтры Блума для ускорения синхронизации кошельков и улучшения их безопасности;
- Google Chrome использует фильтр Блума для проверки вредоносности URL - как в нашем примере;
- Google BigTable и Apache HBase и Apache Cassandra используют фильтру Блума для уменьшения к-ва обращений к диску, чтобы не читать несуществующие записи-колонки;
- Squid Web Proxy Cache использует фильтры Блума;
- фильтр Блума используется для классификации ДНК;
- еще примеры применения на https://en.wikipedia.org/wiki/Bloom_filter (https://en.wikipedia.org/wiki/Bloom_filter).

Применение фильтра Блума в Apache HBase

Apache HBase - это колоночная СУБД. Данные хранятся в массиве записей вида "ключ+колонка+значение". Записи хранятся в store files. Каждый файл 1 Гб к примеру.



Цель: при поиске ключа уменьшить к-во операций ввода-вывода.

Store files состоят из блоков. Каждый блок 64 кбайта. Индекс находится в каждом блоке и хранит стартовый ключ блока. Всего 16384 блока в store file. То есть, всего проиндексировано 16384 ключей. Пусть ячейка занимает 200 байт. Тогда всего в store file хранится 5 млн ячеек. Ищем какой-то ключ. Вероятность, что он попадет не точно в ключ, а между стартовыми ключами двух блоков, очень велика. При обычном поиске ключа без Блум-фильтра находим блок, в котором находится данный ключ, а дальше начинаем full scan этого блока, чтобы найти данный ключ.

Как применяется Bloom Filter?

Bloom filter дает возможности сразу же узнать точно тот факт, что ключа в блоке нет. Но может дать и false positive. Данных может не быть, а фильтр скажет, что данные есть. Процент false positive обычно около 1%. Если вернули true, то скачаем и проверим весь блок. Бонус - число загрузок блока существенно уменьшится, что имеет большое значения для высоконагруженных систем.

Минусы: Надо обновлять Bloom filter при обновлении записей Добавляется оверхэд в 1 байт на 1 запись в фильтре. Если store file имеет размер 1Гб, как в нашем примере, то добавляя ключи в фильтр размера 20 байт (это полностью координаты имеющихся ячеек) имеем 51 Мбайт оверхеда. Если только row level, тогда примерно 100 кб на 1 Гб файл.

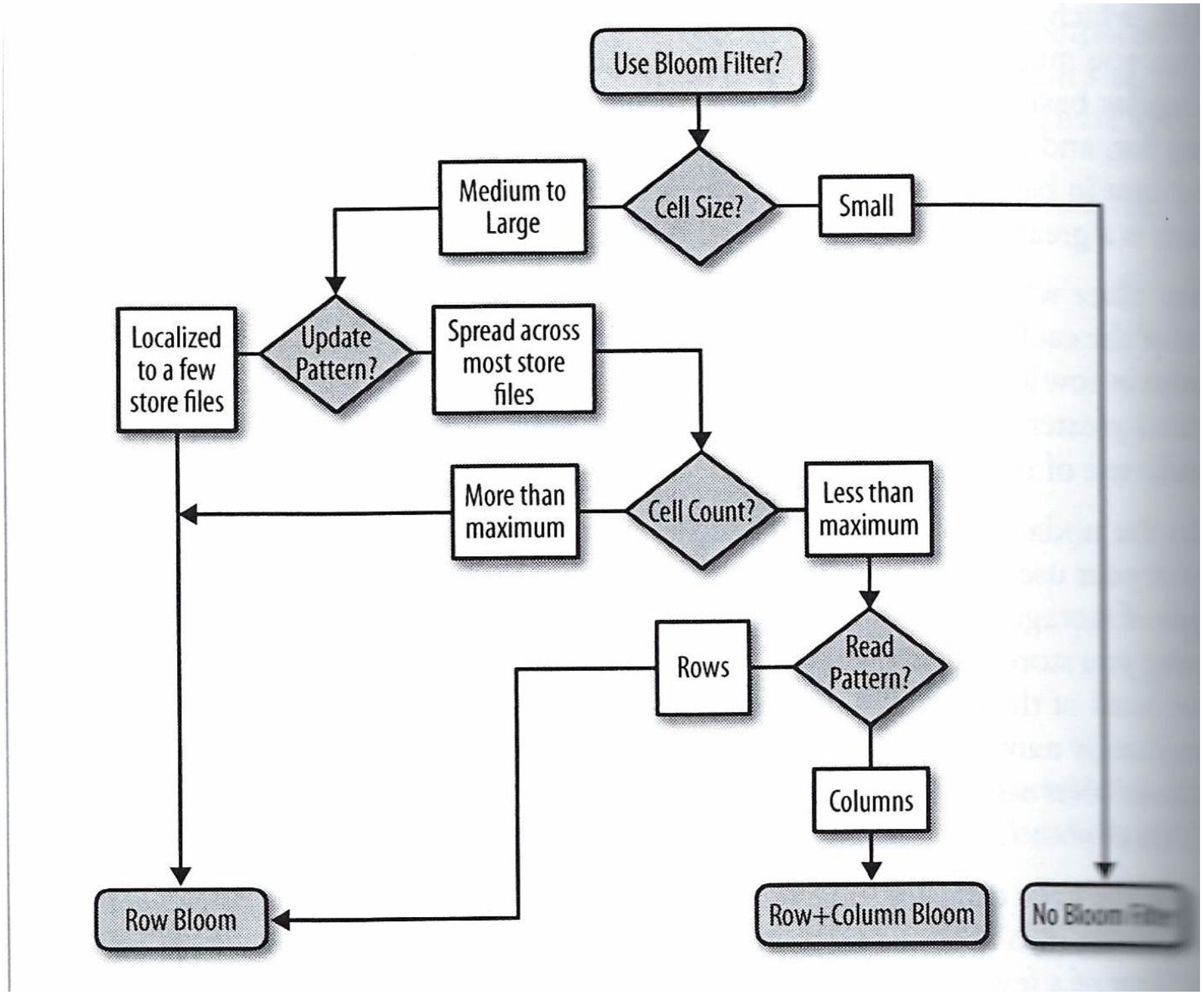
Можно использовать фильтр только для строк, можно строки + колонки. Если ищем только строки, фильтр для колонок не имеет смысла. Если колонки, то имеет смысл.

Параметры

io.hfile.bloom.enabled = Kill switch in case something goes wrong. Default = True

io.hfile.bloom.error.rate = average false positive rate. Default = 1%.

Схема использования



In [ ]:

Домашнее задание

Будет после 3-его вебинара по вероятностным алгоритмам и структурам данных

Заполните опросник в конце занятия:

https://docs.google.com/forms/d/e/1FAIpQLSdbKCxxgd0O6G3E-dDwG0QIhFIAwvEvaTa759KNh22j83IROg/viewform (https://docs.google.com/forms/d/e/1FAIpQLSdbKCxxgd0O6G3E-dDwG0QIhFIAwvEvaTa759KNh22j83IROg/viewform)

Литература

Bloom Filter, оригинальная статья

Bloom, Burton H. (1970), "Space/Time Trade-offs in Hash Coding with Allowable Errors"

Wikipedia

https://en.wikipedia.org/wiki/Bloom_filter (https://en.wikipedia.org/wiki/Bloom_filter)

Интерактивное демо

http://jasondavies.com/bloomfilter/ (http://jasondavies.com/bloomfilter/)

Калькулятор расчета параметров Bloom Filter

https://hur.st/bloomfilter/ (https://hur.st/bloomfilter/)

Data mining: Bloom Filter, other algos & datastructures

J. Leskovec, A. Rajaraman, J. Ullman: Mining of Massive Datasets, http://www.mmds.org (http://www.mmds.org)

Спасибо
за внимание!