NATIONAL RESEARCH UNIVERSITY

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Фильтры

Рамон Антонио Родригес Залепинос
arodriges@hse.ru

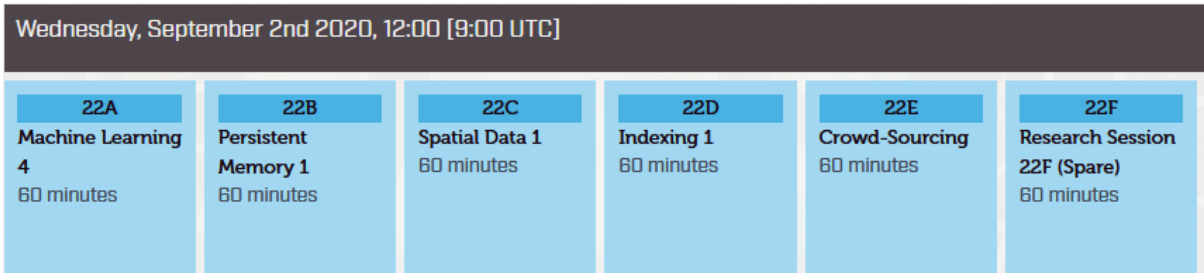
Структура модуля 2

	№		Дата	Тема лекции	№	Домашние задания
Модуль № 2	7	1	27 окт	Хеширование, хэш таблицы 2	2b	ДЗ-Каникулы
	8	2	03 ноя	Фильтры	3	Задание на C++
	9	3	10 ноя	СД для вторичной памяти	4	Задание на C++
	10	4	17 ноя	Пространственные СД	5	Задание на C++
	11	5	24 ноя	Параллельные СД	6	Задание на C++ и/или C# (зависит от выбранного уровня сложности)
	12	6	01 дек	Параллельные СД 2		
	13	?	08 дек	Деревья в оперативной памяти		Примерно за 2 недели заканчиваются ДЗ
	14	8	15 дек	Современные тренды		
				СЕССИЯ с 21.12.2020		

Highlights:

- лекции, семинары и ДЗ синхронизированы
- ? – возможно будет еще одна лекция, с другой темой
- некоторые представления об эффективности СД будут развеяны:
  - на семинаре вы собственноручно, на практике сравните производительность красно-черных деревьев и хэш таблиц (если не нужны next & prev)
- пространственные СД – обширный класс
  - в современном мире, около 80% всех данных содержат географическую привязку: ссылка 1 (Forbes), ссылка 2 (Carto)
  - отдельные секции на значимых конференциях (e.g., VLDB: https://vldb2020.org/program.html)

Требование к студентам на лекции: слушайте внимательно!



«Постановка задачи»

Входные данные

- Ключи $\boldsymbol{key} \in \boldsymbol{K}$

Цель – быстрая **проверка принадлежности ключа множеству И малый объем потребляемой памяти**

Структура данных

- «множество» $\boldsymbol{S}$

$$|\boldsymbol{S}| \ll |\boldsymbol{K}|$$

Поддерживаемые операции

- Нечеткая проверка $key \in S$
- Добавление key в S

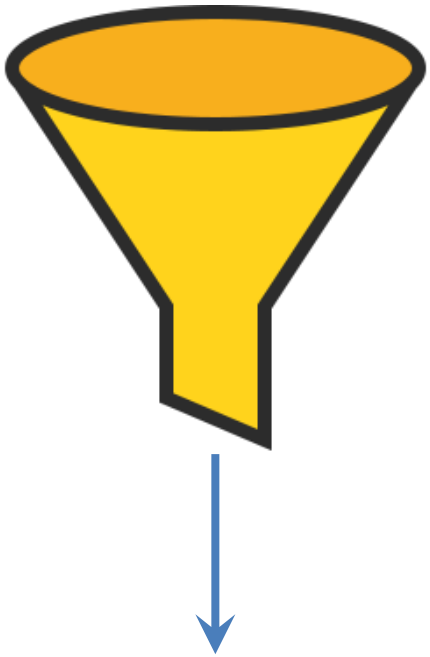
$$O(1)$$

Bloom Filter

Б. Блум, 1970 г.

- вероятностная структура данных
- добавление ключей к множеству S
- проверка принадлежности ключа key к множеству S

Ключ key принадлежит множеству S?



Точно НЕТ

Может быть ДА

FALSE positive

Устройство Bloom Filter

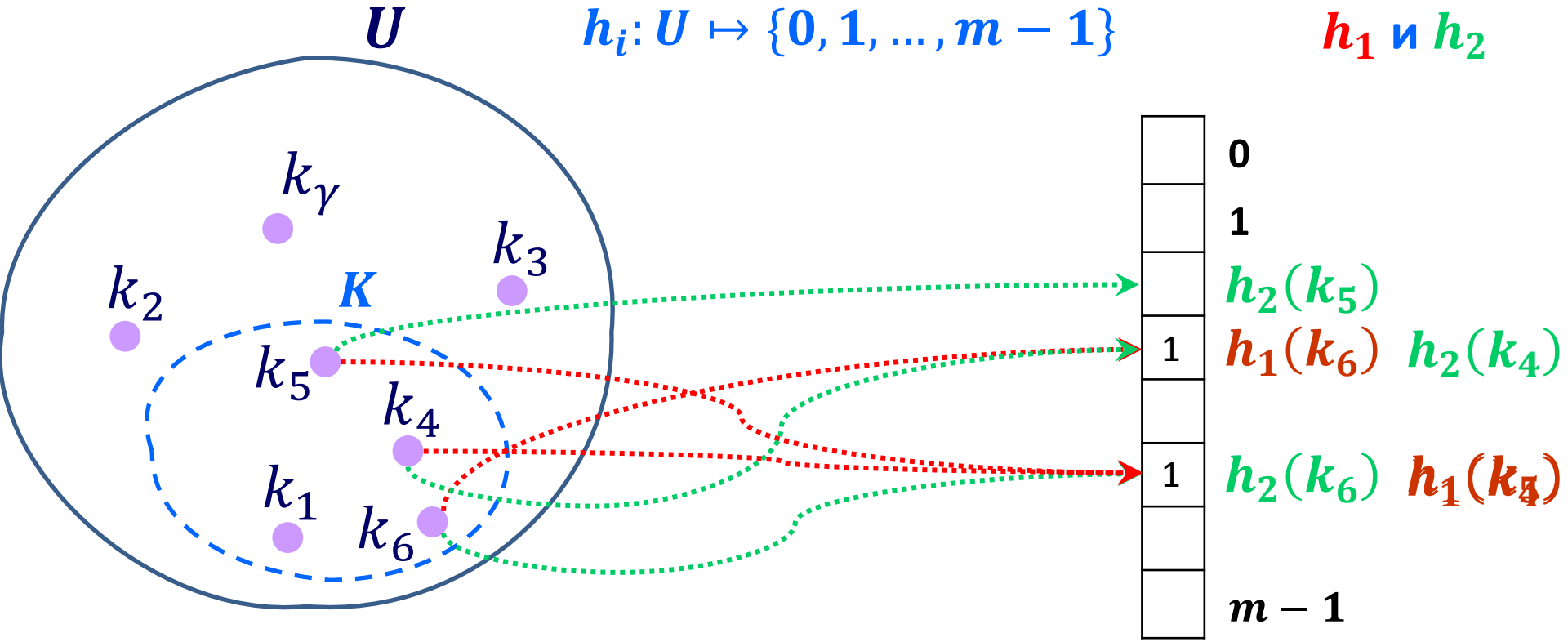
Входные данные

k **хеш-функций:** $h_1, \ldots, h_k$

$h_i: U \mapsto \{0, 1, \ldots, m-1\}$

Пусть $k = 2$**:**

h_1 **и** h_2



Битовый массив

Добавление в фильтр

- Фильтр[$h_1(key)$] = Фильтр[$h_2(key)$] = 1

Проверка принадлежности «может быть»

- Фильтр[$h_1(key)$] == Фильтр[$h_2(key)$] == 1

Размер $m \approx \Theta(|K|)$

Хеш-функции: сколько их нужно?

Оптимальное число

$$k = \frac{\mathrm{m}}{\mathrm{n}} \ln 2$$

Оценка вероятности False Positive

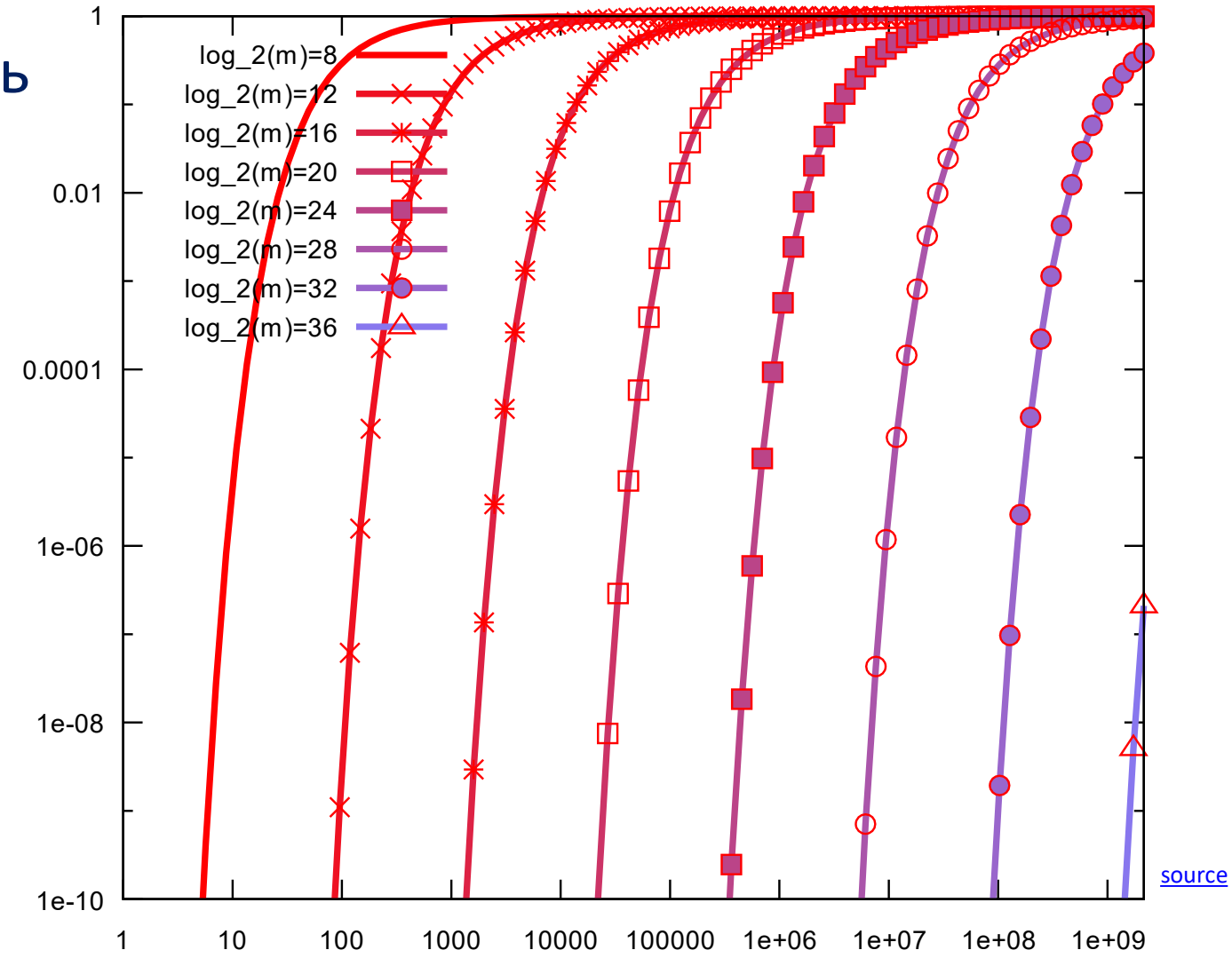
При оптимальном числе хеш-функций $k = (m/n)\ln 2$

Вероятность

Note: это битовый массив ➔
при $m = 2^{32}$ нужно
$2^{32}/2^3 = 2^{29}/$
$2^{20} = 2^9 = 512\text{МБ}$

Можно уверенно вставлять до 100 миллионов ключей

Тогда $k = \frac{2^{32}}{10^8} \times \ln 2$
$\approx 42.94\ldots \times 0.69\ldots$
≈ 30

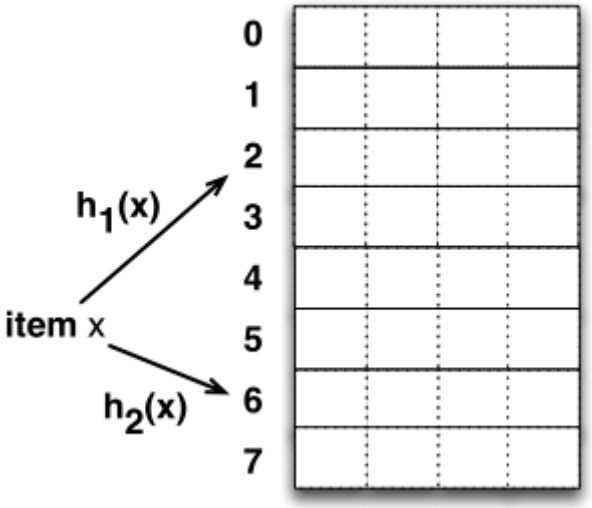


n число вставленных элементов

Кукушкин фильтр: вставка

- Хранит fingerprints (например, 7 бит) вместо ключей (нельзя восстановить хеш вставленного ключа при *relocation*)
- Каждая ячейка (bucket) состоит из b элементов (entry)

partial-key cuckoo hashing
вычисляет альтернативный индекс элемента на основе fingerprint



Bin Fan et al., Cuckoo Filter: Practically Better Than Bloom, 2014

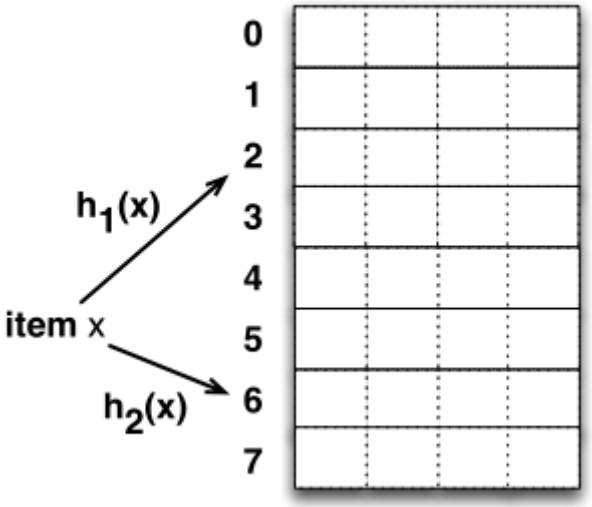
Algorithm 1: `Insert(x)`

```
f = fingerprint(x);
i1 = hash(x);
i2 = i1 ⊕ hash(f);
if bucket[i1] or bucket[i2] has an empty entry then
    add f to that bucket;
    return Done;
// must relocate existing items;
i = randomly pick i1 or i2;
for n = 0; n < MaxNumKicks; n++ do
    randomly select an entry e from bucket[i];
    swap f and the fingerprint stored in entry e;
    i = i ⊕ hash(f);
    if bucket[i] has an empty entry then
        add f to bucket[i];
        return Done;
// Hashtable is considered full;
return Failure;
```

Кукушкин фильтр: поиск и удаление

- Хранит fingerprints (например, 7 бит) вместо ключей (нельзя восстановить хеш вставленного ключа при *relocation*)
- Каждая ячейка (bucket) состоит из b элементов (entry)

Можно удалять только зная, что элемент был точно вставлен ранее



Bin Fan et al., Cuckoo Filter: Practically Better Than Bloom, 2014

Algorithm 2: `Lookup(x)`

f = fingerprint(x);
i_1 = hash(x);
$i_2 = i_1 \oplus$ hash(f);
if bucket[i_1] or bucket[i_2] has f **then**
    return `True`;
return `False`;

Algorithm 3: `Delete(x)`

f = fingerprint(x);
i_1 = hash(x);
$i_2 = i_1 \oplus$ hash(f);
if bucket[i_1] or bucket[i_2] has f **then**
    remove a copy of f from this bucket;
    return `True`;
return `False`;

NATIONAL RESEARCH UNIVERSITY

Благодарю за внимание!

Рамон Антонио Родригес Залепинос
arodriges@hse.ru