Адрианов Н.М.
Иванов А.Б.

Алгоритмы и структуры данных

Хеш-таблицы.
Распределенные хеш-таблицы

Ассоциативный массив (Map, Dictionary)

Абстрактная структура данных - набор пар <ключ, значение> (ключ уникальный).

Операции (интерфейс):

- Вставка
- Поиск по ключу
- Удаление по ключу

Хеш-таблица (Hash Table)

Структура данных, реализующая ассоциативный массив.

Соглашение по сложности (контракт):

- Вставка: $O(1)^*$
- Поиск по ключу: $O(1)^*$
- Удаление по ключу: $O(1)^*$

Память - $O(n)$

Применение:

- Анализ потока данных (уникальные пользователи, etc)
- Сетевое оборудование (firewall, балансировщик, IPS)
- 3-SUM (vs Subset sum)
- ...

Хеш-функция

$$h : U \to \{0, \ldots, m-1\}$$

Коллизия:

$$x, y \in U : h(x) = h(y)$$

Идеальное хэширование (perfect hash)

Хеш-функция

$$h : U \to \{0, \ldots, m-1\}$$

Коллизия:

$$x, y \in U : h(x) = h(y)$$

Парадокс дней рождения. В группе, состоящей из 23 или более человек, вероятность совпадения дней рождения (число и месяц) хотя бы у двух людей превышает 50%.

U – группа людей, $h : U \to \{1, \ldots 365\}$.

`https://en.wikipedia.org/wiki/Birthday_problem`
Количество n необходимое для получения коллизий с вероятностью p:

$$n(p, m) \approx \sqrt{2m \cdot \ln\left(\frac{1}{1-p}\right)}$$

Пример реализации в Java (сейчас)

```
/** Cache the hash code for the string */
private int hash; // Default to 0

public int hashCode(char[] value) {
    int h = hash;
    if (h == 0 && value.length > 0) {
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + value[i];
        }
        hash = h;
    }
    return h;
}
```

Пример реализации в Java: почему 31?

Joshua Bloch “Effective Java”

- `Always override hashcode when you override equals`

The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimization automatically.

Пример реализации в Java (давно)

```
public int hashCode() {
    int h = 0;
    int off = offset;
    char val[] = value;
    int len = count;

    if (len < 16) {
        for (int i = len ; i > 0; i--) {
            h = (h * 37) + val[off++];
        }
    } else {
        // only sample some characters
        int skip = len / 8;
        for (int i = len ; i > 0; i -= skip, off += skip) {
            h = (h * 39) + val[off];
        }
    }
    return h;
}
```

Реализация: метод цепочек (chaining)

- Массив размера m
- В каждой ячейке храним список (key, value)

```
buckets = array of length m
function find(key):
  h = hash(key)
  node = buckets[h] // linked list
  while node != null:
    if node.key == key:
      return node.value
    node = node.next
  return null
end
```

Реализация: Открытая адресация

- Массив размера m
- В каждой ячейке храним (key, value) (не список!)
- Приведена одна из возможных реализаций открытой адресации (линейное исследование)
  probing = опробывание, исследование

```
buckets = array of length m
function find(key):
  h = hash(key)
  node = buckets[h] // array element
  if node.key == key:
    return node.value
  i = (h + 1) % m
  while node != null && i != h:
    if node.key == key:
      return node.data
  return null
end
```

Варианты открытой адресации

- Линейное исследование:

$$h(i, x) = h_1(x) + i \pmod m$$

- Квадратичное исследование:

$$h(i, x) = h_1(x) + c_2 i^2 + c_1 i \pmod m$$

Например:

$$h(i, x) = h_1(x) + 1 + 2 + \ldots + i = h_1(x) + i(i + 1)/2$$

- Двойное хэширование:

$$h(i, x) = h_1(x) + i h_2(x) \pmod m$$

- "Робин Гуд"("Robin Hood")
- "Кукушка"("Cuckoo")

Сложность: метод цепочек

Утверждение

Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна $\Theta(1+\alpha)$, *где* α *- коэффициент заполнения таблицы.*
($\alpha = \frac{n}{m}$, *где* m *- количество ячеек)*

Сложность: открытая адресация

Утверждение

Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна $O(\frac{1}{1-\alpha})$, *где* $\alpha < 1$ *- коэффициент заполнения таблицы.* ($\alpha = \frac{n}{m}$, *где* m *- количество ячеек*)

Сложность: открытая адресация

Утверждение

Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна $O(\frac{1}{1-\alpha})$, *где* $\alpha < 1$ *- коэффициент заполнения таблицы.* ($\alpha = \frac{n}{m}$, *где* m *- количество ячеек)*

Если X =количество проверок при неуспешном поиске, то

$$E[X] = \sum_{i=1}^{\infty} P\{X \geq i\} \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

т.к.

$$\sum_{i=0}^{\infty} iP\{X = i\} = \sum_{i=0}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\}) = \sum_{i=0}^{\infty} P\{X \geq i\}$$

Открытая адресация VS метод цепочек

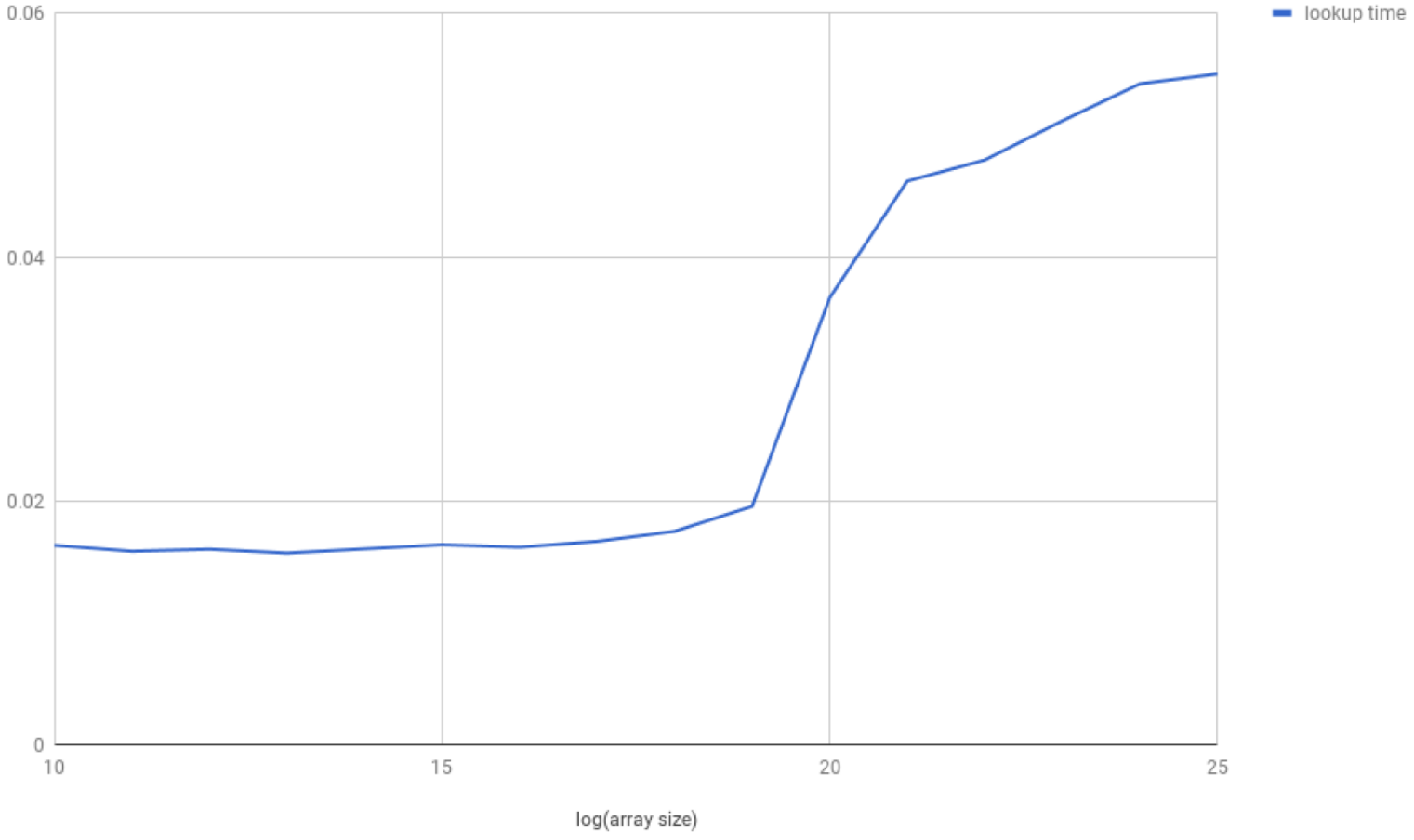
Недостатки:

- Сложность реализации (относительная)
- Сложность удаления
- Большая чувствительность к коэффициенту заполнения

Преимущества:

- Локальность!!!

Локальность



Распределенные хэш-таблицы

Объект $x \in U$
Ячейки (сервера): $S = \{S_1, \dots S_n\}$

Необходимо построить хеш-функцию

$$H : U \to S$$

Свойства:

- Равномерное распределение
- Минимальная перестройка при изменении S (добавлении/удалении элемента)

Применение

- Распределение нагрузки
- Распределённые хеш таблицы
- Распределённые БД

Консистентное хеширование (Consistent Hashing/Hash Ring)

- Представляем диапазон $[1 \dots 2^{32}]$ как окружность S^1
- Для каждого S_i генерим m случайных точек $s_{ij} \in S^1$
- Хеш функция $h : U \to S^1$
- Если ближайшая по часовой стрелке к $h(x)$ – точка $s_{kj} \in S^1$, то полагаем $H(x) = S_k$.

Rendezvous хеширование

- Хеш-функция

$$h : U \times S \to [1 \dots 2^{32}]$$

- Для $x \in U$ выбираем ячейку S_k, на которой хеш-функция принимает максимальное значение:

$$H(x) = \underset{s}{\operatorname{argmax}}\ h(x, s)$$

Другими словами

$$H(x) = k, \text{ если } h(x, S_k) \geq h(x, S_i)\ \forall i$$

Фильтр Блума (Bloom filter)

Вероятностая структура данных (множество). Операции:

- Вставка
- Поиск по ключу (бинарный результат)

vs Хеш-таблица:

- $\oplus$ память и скорость!!!
- $\ominus$ нет удалений
- $\ominus$ нельзя хранить значение
- $\ominus$ вероятность ложноположительного поиска

Фильтр Блума - конструкция

- Битовый массив B длины n (например, $n = 8|S|$)
- k хеш функций $h_1, \ldots, h_k$, $h_i : U \to \{1, \ldots, n\}$

Вставка ключа x: $B[h_i(x)] = 1$ для всех $i = 1, \ldots, k$

Поиск ключа x: "да" $\Leftrightarrow$ $B[h_i(x)] = 1$ для всех $i = 1, \ldots, k$

Фильтр Блума - анализ

В предположении равномерности и независимости хеш функций h_i: вероятность того, что конкретный бит установлен в 1 после вставки всего S, равна

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

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

Фильтр Блума - анализ

В предположении равномерности и независимости хеш функций h_i: вероятность того, что конкретный бит установлен в 1 после вставки всего S, равна

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

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

$$n/|S| = 8, \quad k = 2 \quad \longrightarrow \quad < 5\%$$

Фильтр Блума - анализ

В предположении равномерности и независимости хеш функций h_i: вероятность того, что конкретный бит установлен в 1 после вставки всего S, равна

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

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

$$n/|S| = 8, \quad k = 2 \quad \longrightarrow \quad < 5\%$$

$$n/|S| = 9, \quad k = 6 \quad \longrightarrow \quad \approx 2\%$$

Фильтр Блума - анализ

В предположении равномерности и независимости хеш функций h_i: вероятность того, что конкретный бит установлен в 1 после вставки всего S, равна

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

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

$$n/|S| = 8, \quad k = 2 \quad \longrightarrow \quad < 5\%$$

$$n/|S| = 9, \quad k = 6 \quad \longrightarrow \quad \approx 2\%$$

$$n/|S| = 16, \quad k = 11 \quad \longrightarrow \quad < 0.05\%$$

Фильтр Блума - анализ

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

Фильтр Блума - анализ

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

При фиксированном $n/|S|$ как функция от k достигает минимума при

$$k^* = \frac{n}{|S|} \ln 2$$

Фильтр Блума - анализ

Вероятность ложноположительного поиска:

$$\left(1 - e^{-\frac{k|S|}{n}}\right)^k$$

При фиксированном $n/|S|$ как функция от k достигает минимума при

$$k^* = \frac{n}{|S|} \ln 2$$

$$n/|S| = 9, \quad k^* \approx 6.24, \quad k = 6 \quad \longrightarrow \quad \approx 2\%$$

$$n/|S| = 16, \quad k^* \approx 11.09, \quad k = 11 \quad \longrightarrow \quad < 0.05\%$$

Фильтр Блума - использование

`https://en.wikipedia.org/wiki/Bloom_filter`

Akamai Technologies (CDN) - to prevent "one-hit-wonders"from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.

Google Bigtable, Apache HBase and Apache Cassandra and PostgreSQL - to reduce the disk lookups for non-existent rows or columns.

Google Chrome web browser - to identify malicious URLs

Microsoft Bing (search engine) uses multi-level hierarchical Bloom filters for its search index, BitFunnel. Bloom filters provided lower cost than the previous Bing index, which was