





C++ZeroCost
Conf 



C++ zero-cost abstractions на примере хеш-таблиц в ClickHouse



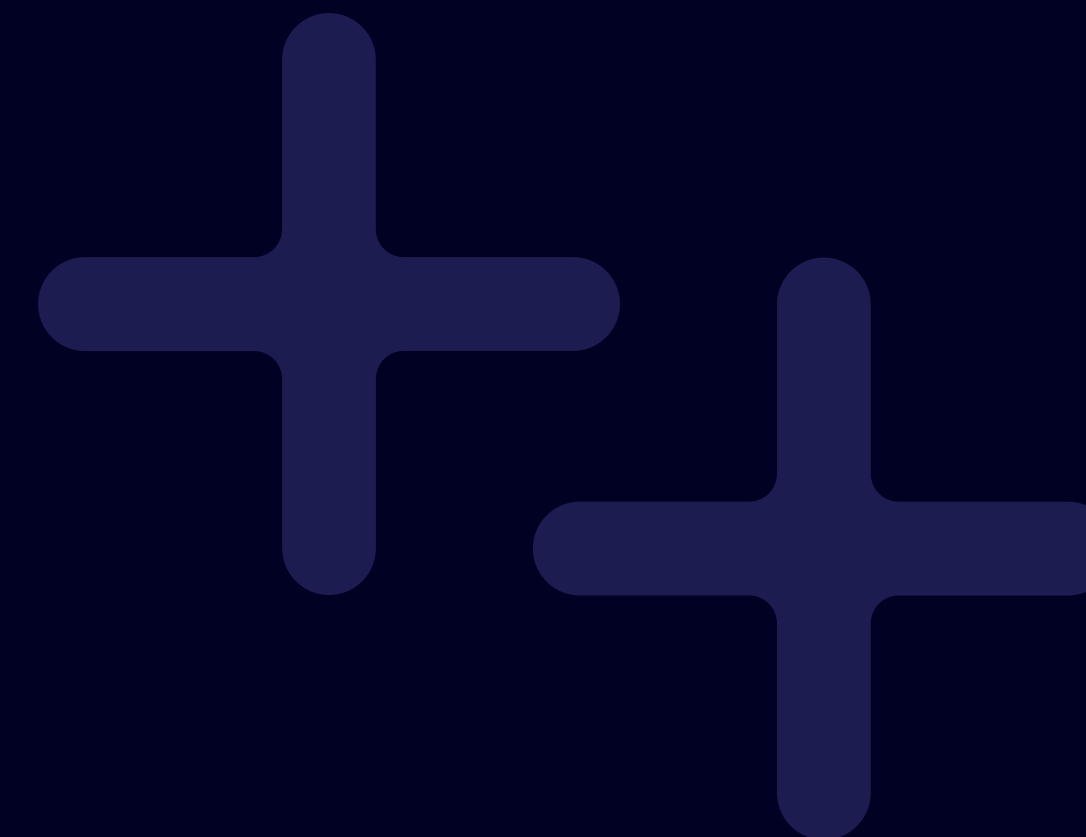
Максим Кита
Старший разработчик ClickHouse



Yandex for `developers` 

Содержание

- 01 Введение в хеш-таблицы
- 02 Основные вопросы дизайна
- 03 Бенчмарки
- 04 C++ дизайн хэш-таблицы



01

Введение в хеш-таблицы

Хеш-таблицы в ClickHouse

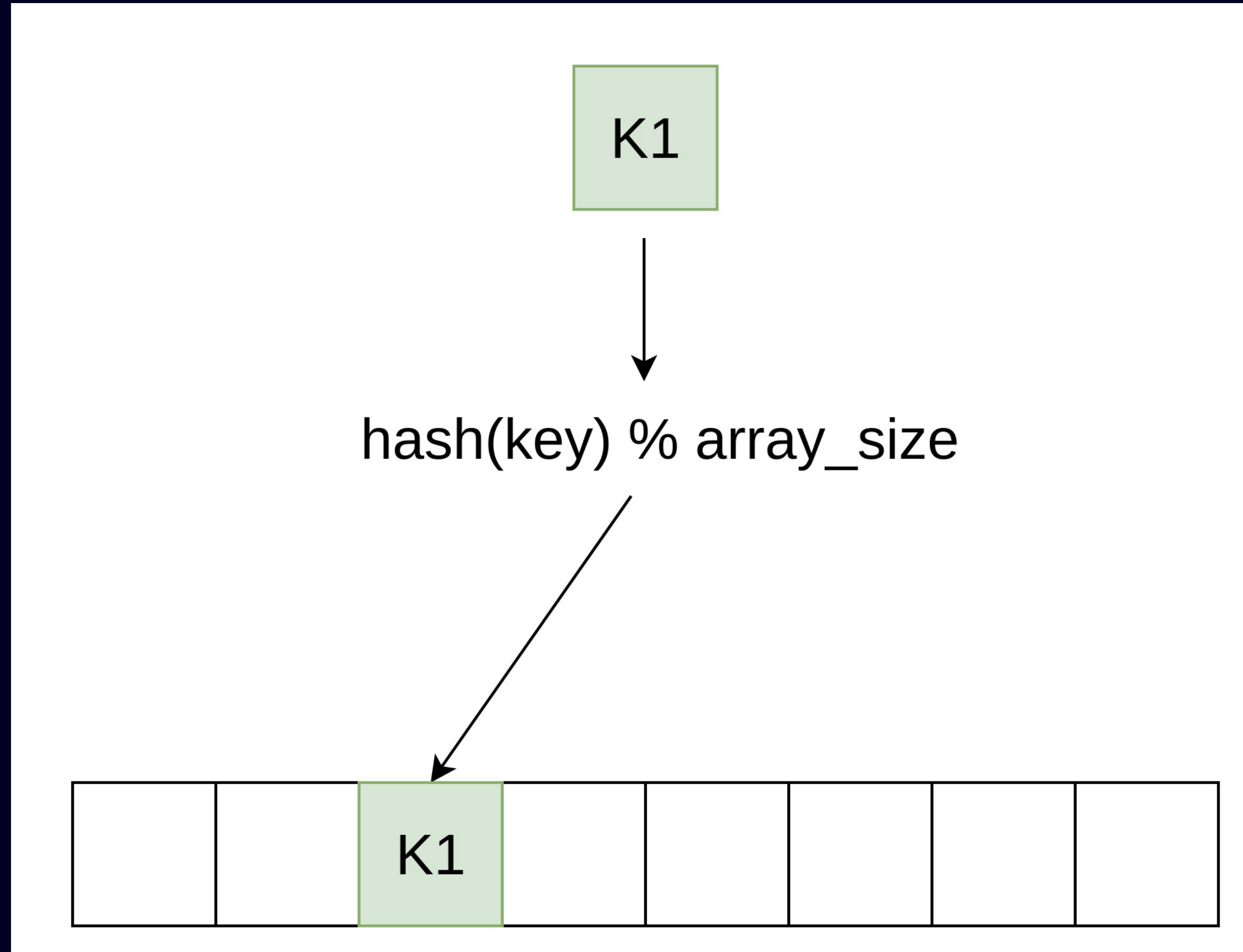
1. GROUP BY
2. JOIN
3. SELECT DISTINCT

Хеш-таблица

Основные методы

1. lookup $O(1)$ average
2. insert $O(1)$ average
3. erase $O(1)$ average (Не очень важен для наших сценариев)

Хеш-таблица



02

Дизайн хеш-таблицы

Составляющие хеш-таблицы

1. Хеш-функция
2. Способ разрешения коллизий
3. Ресайз
4. Способ размещения ячеек в памяти

Выбор хеш-функции

1. Не использовать identity-функцию для целочисленных типов
2. Не использовать хэш-функции для строк (CityHash) для целочисленных типов
3. Не использовать криптографические хэш-функции, если вас не атакуют. Например, вычисление SipHash ~980 MB/s. CityHash ~9 GB/s.
4. Не использовать устаревшие хэш-функции. FNV1a

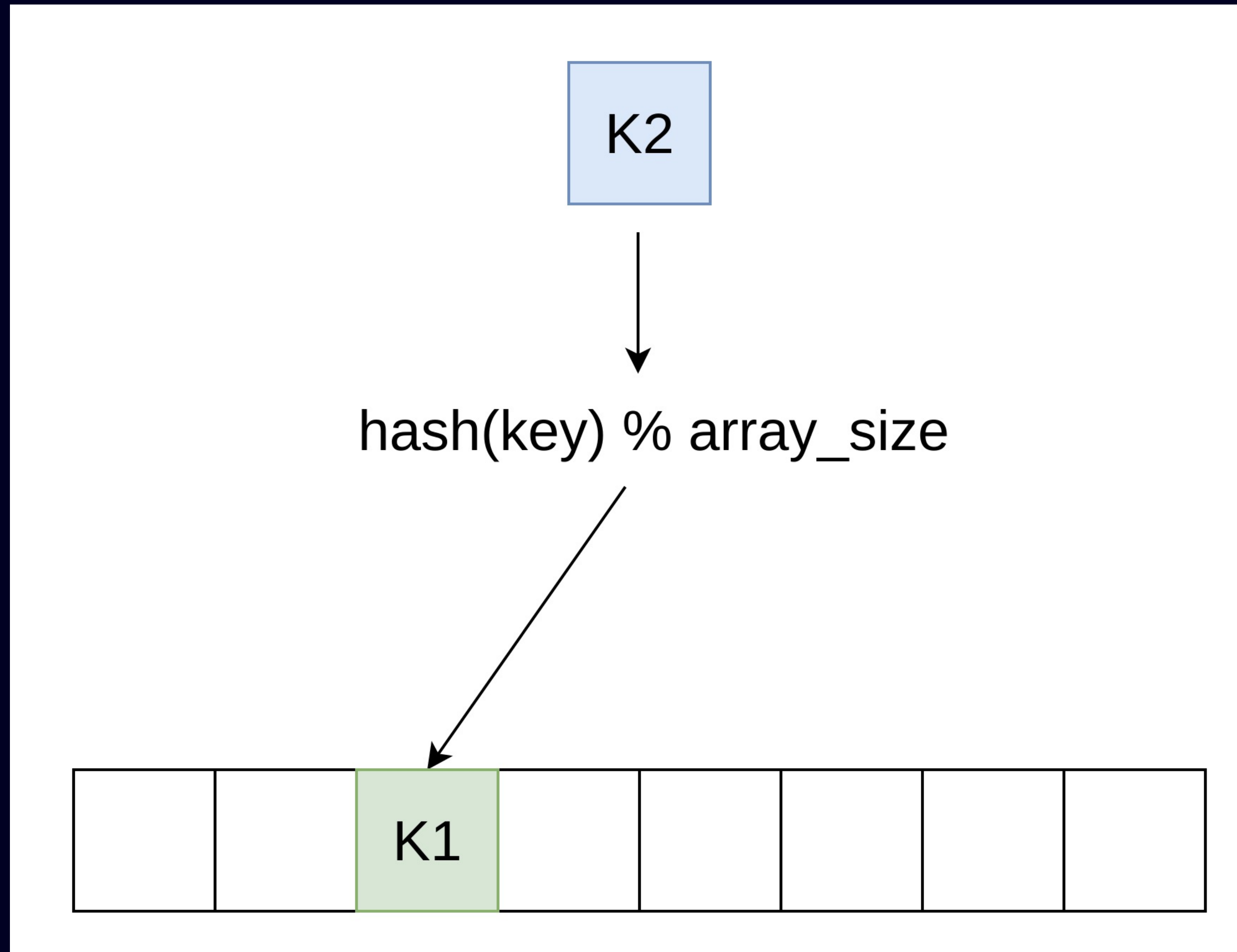
<https://github.com/rurban/smhasher>

Выбор хеш-функции

По умолчанию в ClickHouse плохие хэш-функции

1. CRC32-C для целочисленных типов. Одна инструкция (на самом деле две) процессора latency 3 такта
2. Специальная хэш-функция для строк. Стандартно можно использовать CityHash, xxHash, wyhash

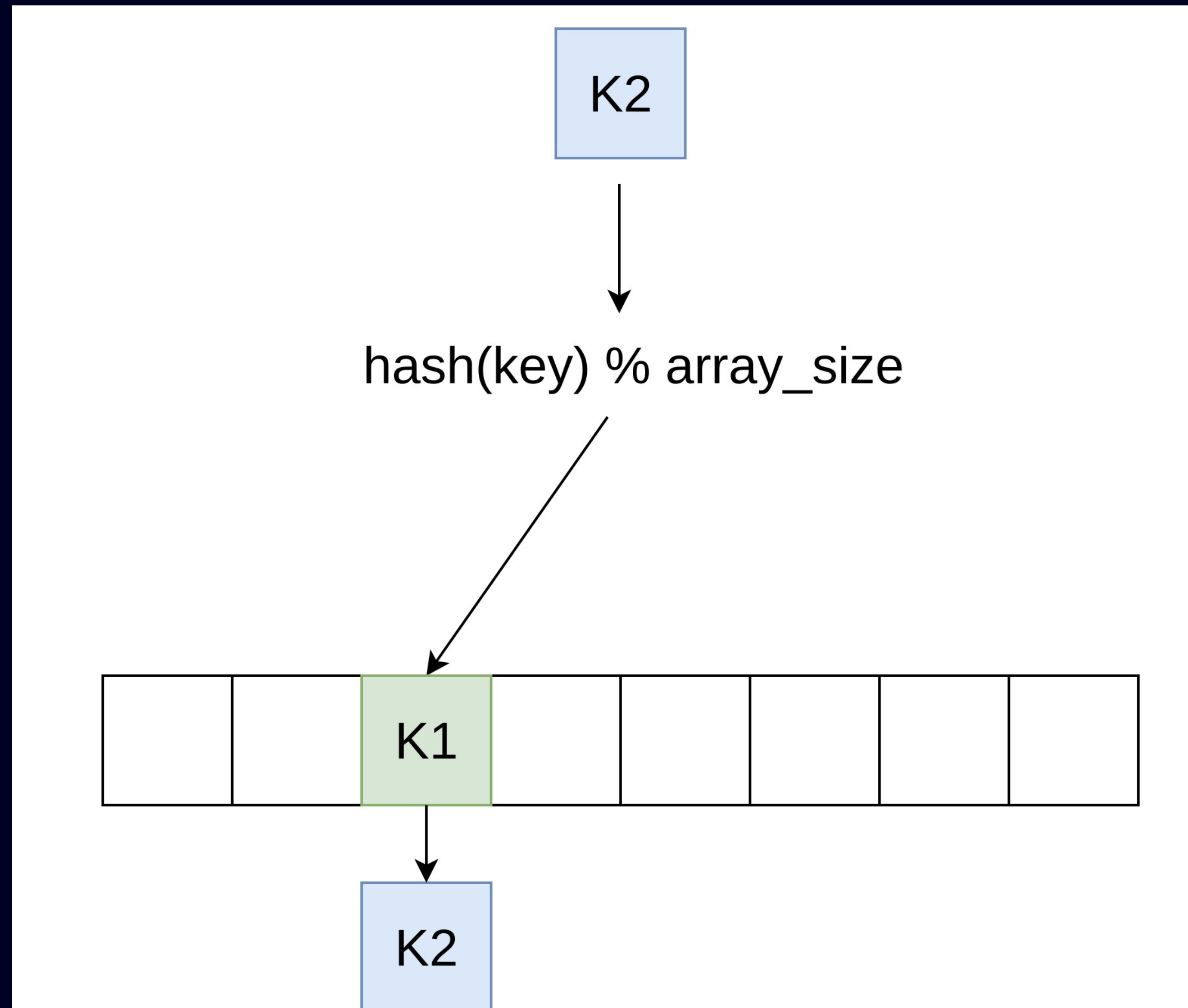
Разрешения коллизий



Разрешения коллизий

1. Метод цепочек (Chaining)
2. Открытая адресация (Open Addressing)
3. Хорошие в теории (Cuckoo hashing, Hopscotch hashing, 2-choice hashing). Обычно либо тяжело реализуемые, либо медленные за счет дополнительных фетчей из памяти

Метод цепочек

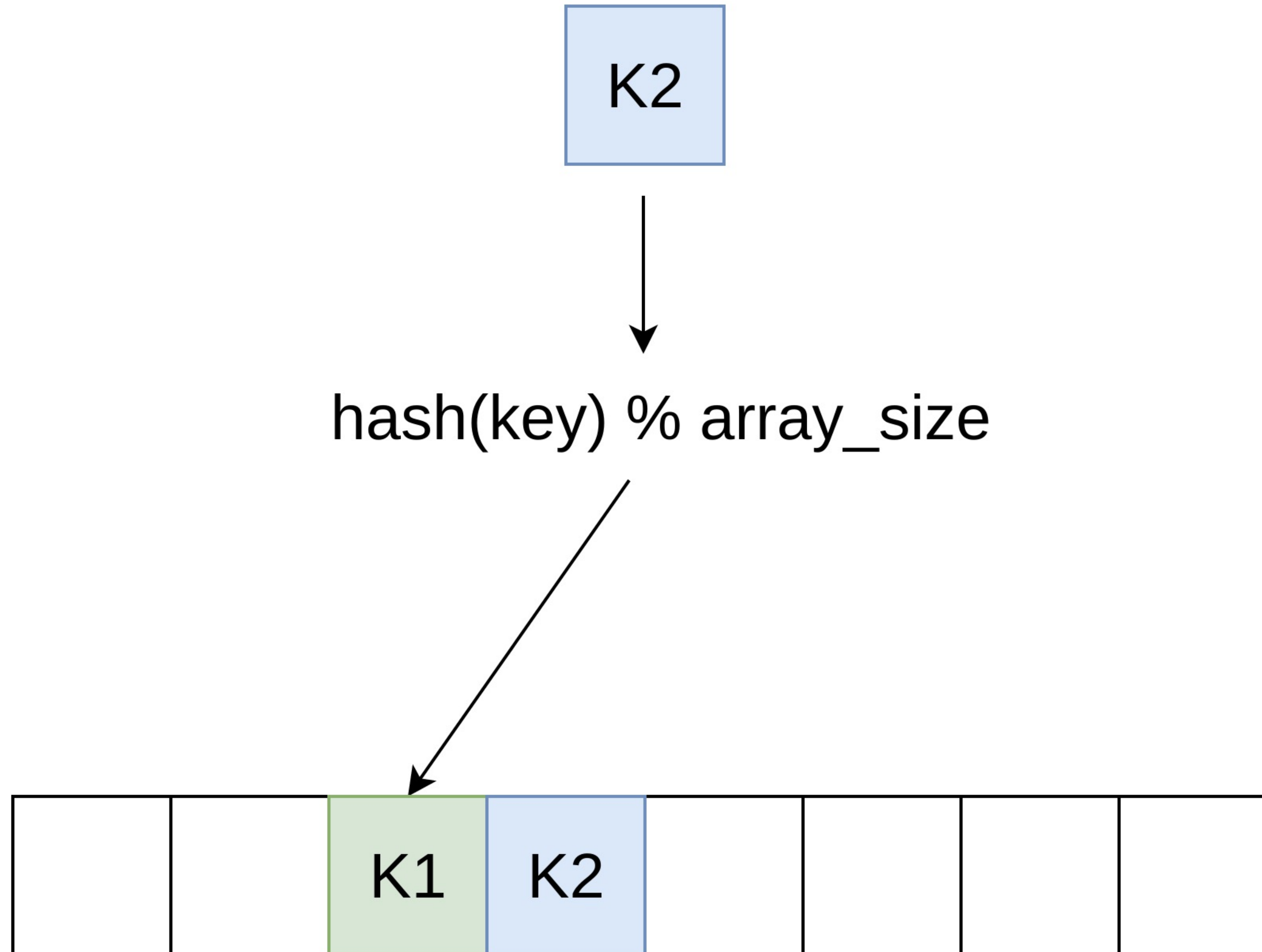


Метод цепочек

Пример: `std::unordered_map`

1. Стабильность указателей на ключ, значение
2. Возможность хранить большие объекты, непереключаемые объекты
3. Хорошо работает с плохой хэш-функцией, высоким load factor
4. Очень сильно тормозит. Нагружает аллокатор (даже просто вызов функции дорого для hot path)

Открытая адресация



Открытая адресация

Линейный пробы (Linear probing). Пример ClickHouse HashMap.

Квадратичные пробы (Quadratic probing). Пример: Google DenseHashMap.

1. Хорошая кэш-локальность.
2. Нужно аккуратно выбирать хэш-функцию.
3. Нельзя хранить большие объекты. Сериализуем в арену и храним указатели на них.

Ресайз

1. По степеням двойки. Быстрое деление по модулю

`size_t place = hash & (size - 1)`

2. На размер простого числа близкого к степени двойки. Медленное деление даже с `constant switch`, `libdivide` но есть ещё `fastrange`

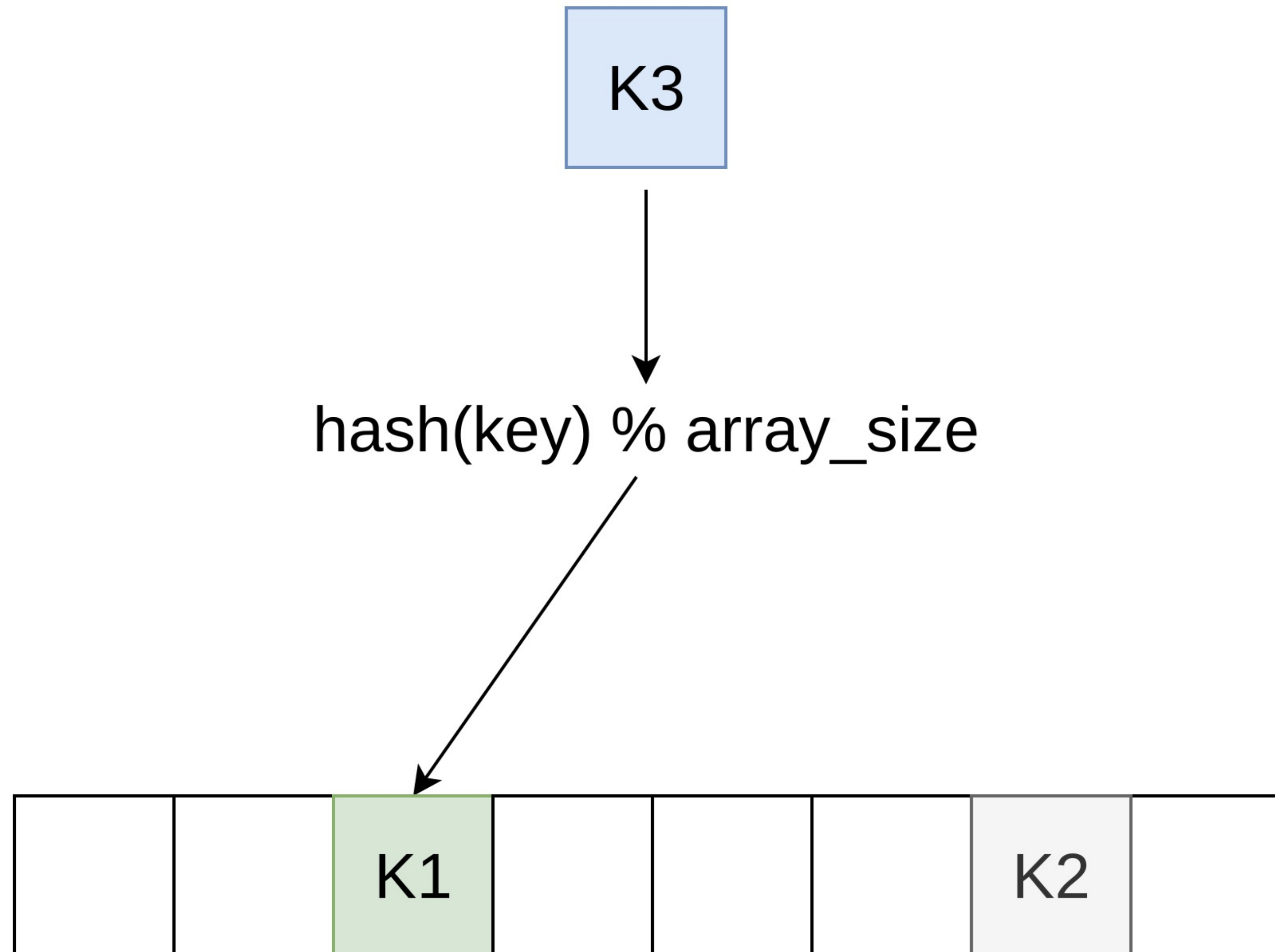
Выбор load factor

0.5 хороший вариант для линейных проб с шагом 1

ClickHouse HashMap, Google DenseHashMap использует 0.5

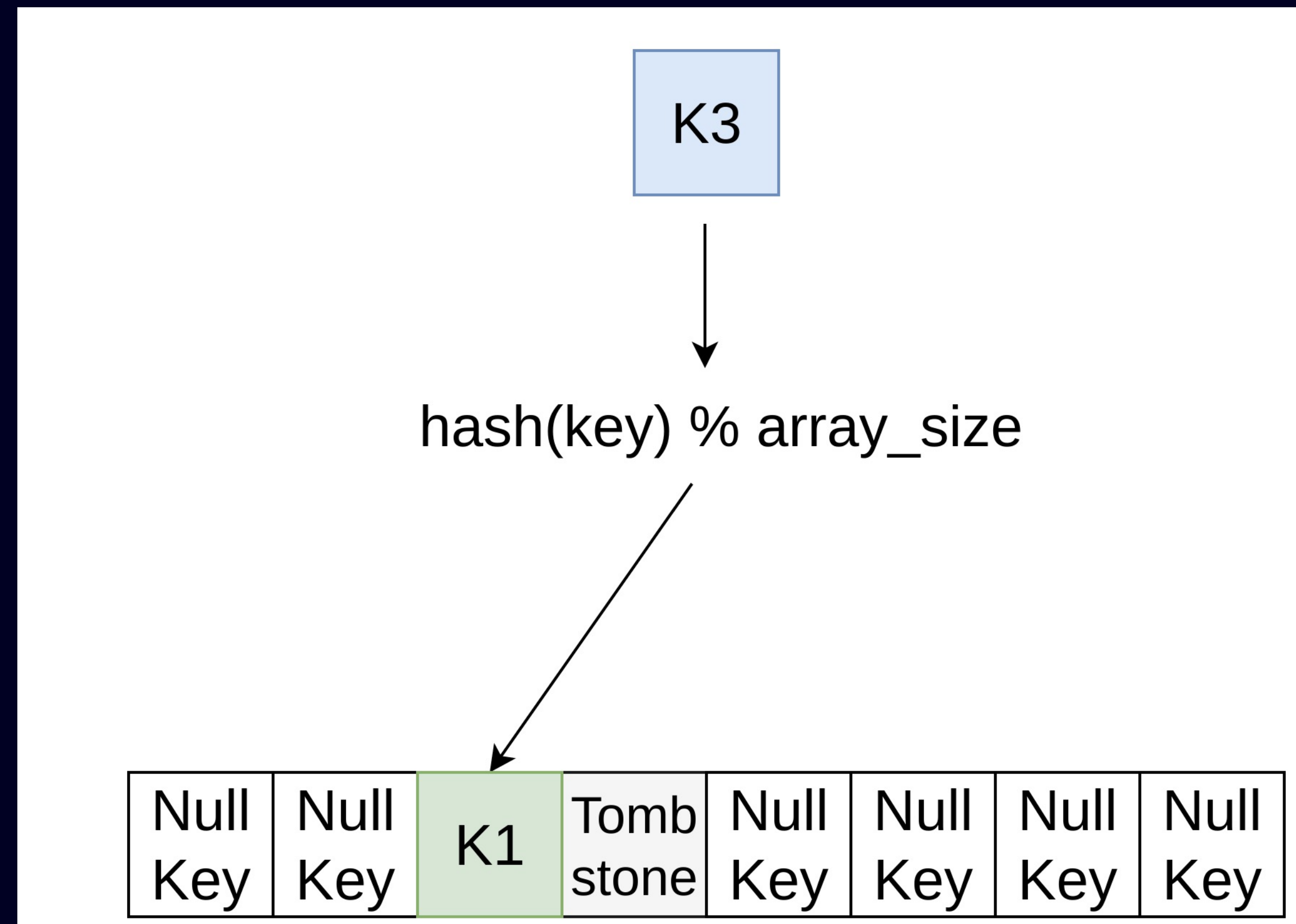
Abseil HashMap использует 0.875

Способ размещения в памяти



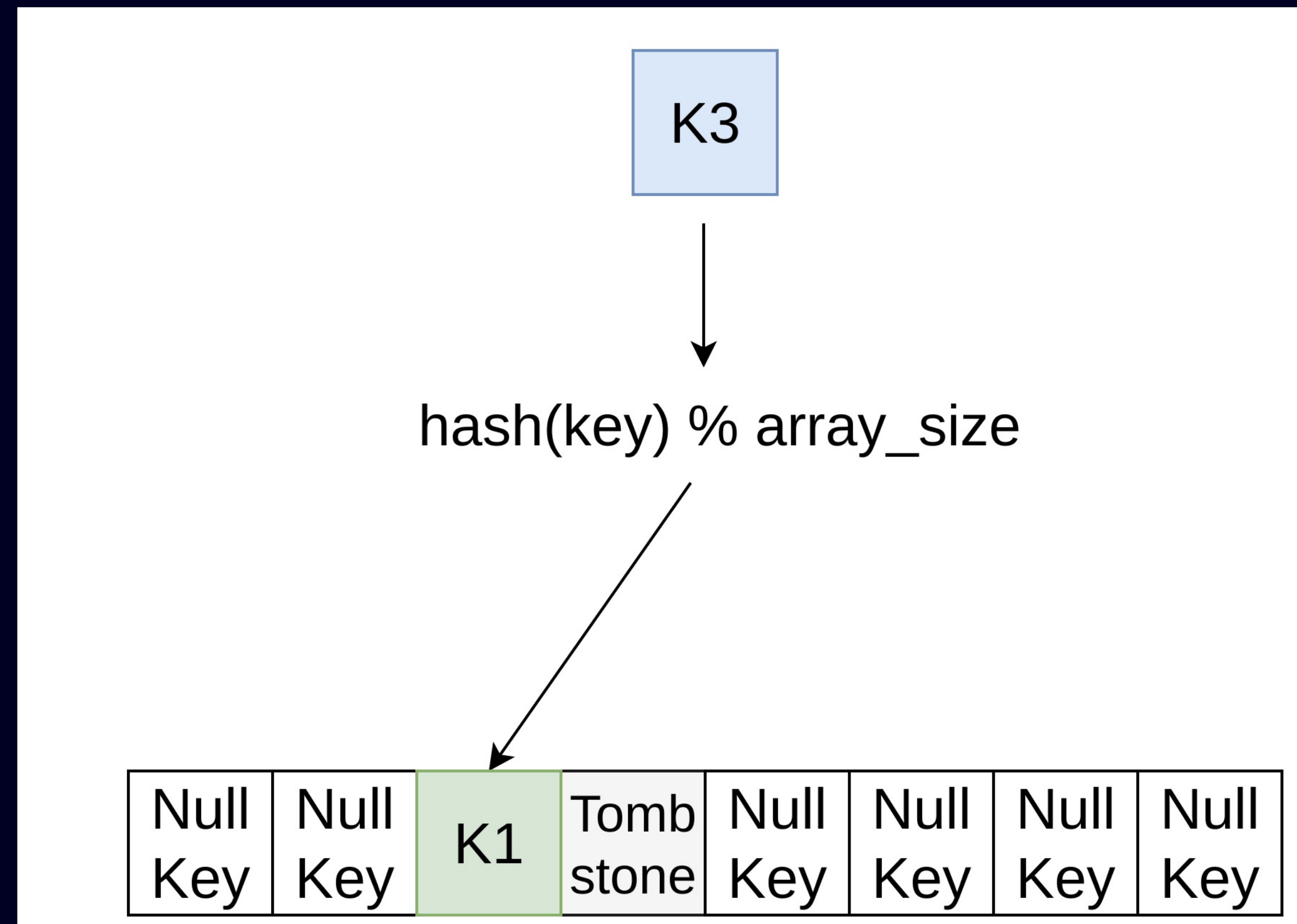
Способ размещения в памяти

Просить клиента выбрать ключи для пустого значения и удаленного



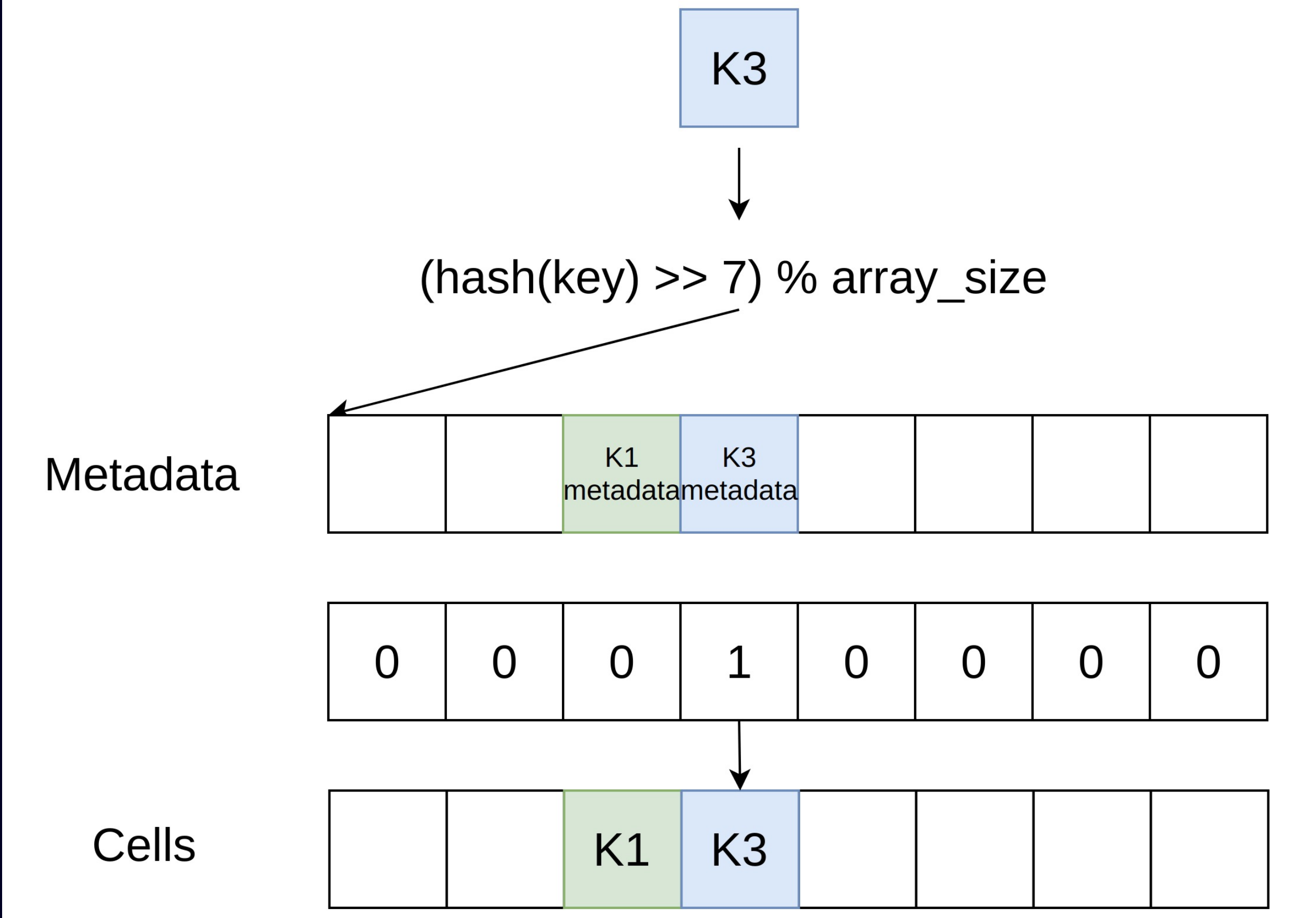
Способ размещения в памяти

Отдельно обрабатывать пустое значение и не хранить его в хэш-таблице.



Способ размещения в памяти

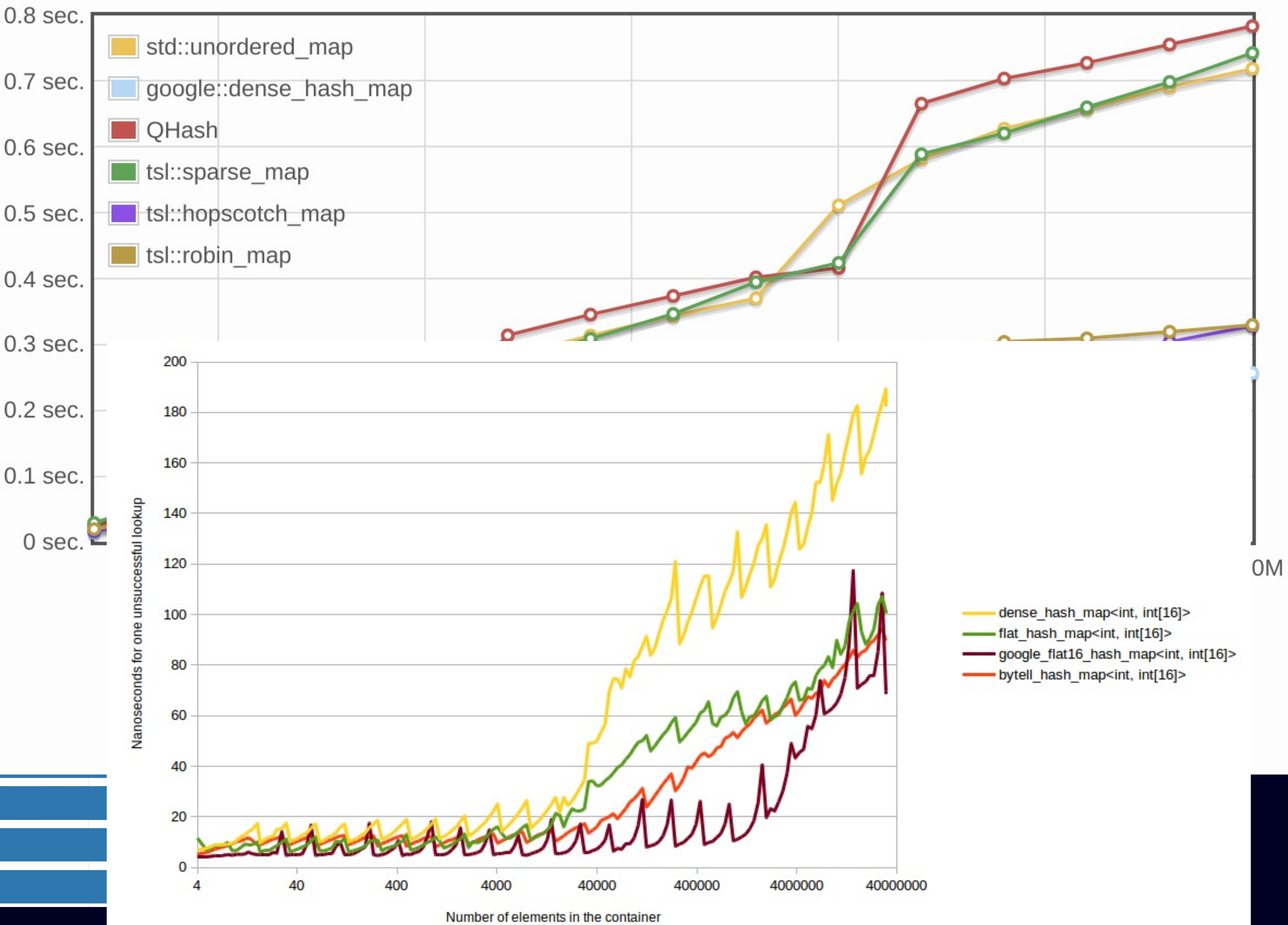
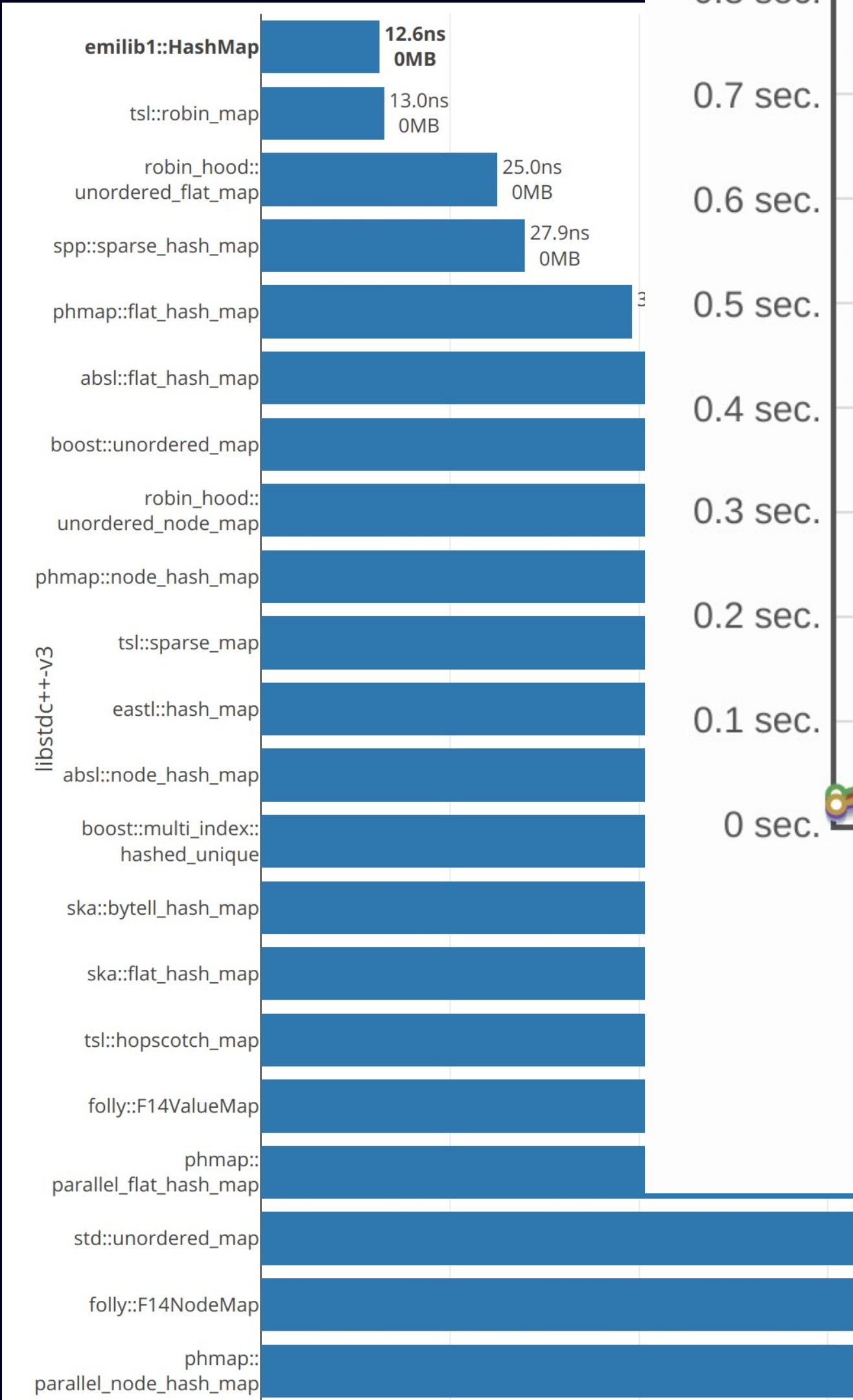
Сжатое хранения метадаты и данных.



03

Бенчмарки

Бенчмарки



Как не надо делать бенчмарки

1. Тестировать хеш-таблицы на случайных целочисленных значениях
2. Тестировать хеш-таблицы без учета максимального load factor, memory consumption
3. Тестировать и не показывать код бенчмарка

Как надо делать бенчмарки

На реальных сценариях и на реальных данных. В ClickHouse реальный сценарий - агрегация данных.

Датасет данных Яндекс.Метрики.

```
wget https://datasets.clickhouse.tech/hits/partitions/hits_100m_obfuscated_v1.tar.xz
```

Бенчмарки

WatchID почти все значения уникальные. Размер хеш-таблицы 20714865 элементов. Это ~600 MB, не влазит в LL-кэши.

Хеш-таблица	Время
ClickHouse HashMap	7.366 сек.
Google DenseMap	10.089 сек.
Abseil HashMap	9.011 сек.
std::unordered_map	44.758 сек.

Бенчмарки

```
perf stat -e cache-misses:u ./integer_hash_tables_and_hashes
```

Хеш-таблица	Cache misses
ClickHouse HashMap	329,664,616
Google DenseMap	383,350,820
Abseil HashMap	415,869,669
std::unordered_map	1,939,811,017

Бенчмарки

Latency Comparison Numbers

L1 cache reference	0.5	ns	
Branch mispredict	5	ns	
L2 cache reference	7	ns	14x L1 cache
Mutex lock/unlock	25	ns	
Main memory reference	100	ns	20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000	ns	3 us
Send 1K bytes over 1 Gbps network	10,000	ns	10 us
Read 4K randomly from SSD*	150,000	ns	150 us ~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250 us
Round trip within same datacenter	500,000	ns	500 us

<http://norvig.com/21-days.html#answers>

Бенчмарки

RegionID часто повторяющиеся значения. Размер хэш таблицы 9040 элементов. Влазит в LL кэши.

Хеш-таблица	Время
ClickHouse HashMap	0.201 сек.
Google DenseMap	0.261 сек.
Abseil HashMap	0.307 сек.
std::unordered_map	0.466 сек.

04

C++ дизайн хеш-таблицы

C++ дизайн хэш-таблицы

1. Hash
2. Allocator
3. Cell
4. Grower (интерфейс для ресайза)
5. HashTable

C++ дизайн хэш-таблицы

Hash

Такой же как `std::hash`.

```
template <typename T>
struct Hash
{
    size_t operator() (T key) const
    {
        return DefaultHash<T>(key);
    }
};
```

С++ дизайн хэш-таблицы

Hash

Такой же как `std::hash`.

```
template <typename T>
struct Hash
{
    size_t operator() (T key) const
    {
        return DefaultHash<T>(key);
    }
};
```

C++ дизайн хэш-таблицы

Allocator

Использует `mmap`, `mremap` для больших кусков памяти.

```
class Allocator
{
    void * alloc(size_t size, size_t alignment);
    void free(void * buf, size_t size);
    void * realloc(void * buf, size_t old_size, size_t new_size);
};
```

Есть аллокатор, изначально выделяющий память на стеке:

```
AllocatorWithStackMemory<HashTableAllocator, initial_bytes>
```

C++ дизайн хэш-таблицы

Cell

```
template <typename Key, typename Mapped, typename HashTableState>
struct HashTableCell
{
    ...

    void setHash(size_t hash_value);

    size_t getHash(const Hash & hash) const;

    bool isZero(const State & state);

    void setZero();

    ...
};
```

C++ дизайн хэш-таблицы

Grower

```
struct HashTableGrower
{
    size_t place(size_t x) const;
    size_t next(size_t pos) const;
    bool willNextElementOverflow() const;
    void increaseSize();
};
```

C++ дизайн хэш-таблицы

HashTable

```
template
<
    typename Key,
    typename Cell,
    typename Hash,
    typename Grower,
    typename Allocator
>
class HashTable :
    protected Hash,
    protected Allocator,
    protected Cell::State;
    protected ZeroValueStorage<Cell::need_zero_value_storage, Cell>
```


C++ дизайн хэш-таблицы

ZeroValueStorage

```
template <bool need_zero_value_storage, typename Cell>
struct ZeroValueStorage;

template <typename Cell>
struct ZeroValueStorage<true, Cell>
{
    ...
};

template <typename Cell>
struct ZeroValueStorage<false, Cell>
{
    ...
};
```

C++ дизайн хэш-таблицы

Возможность передавать кастомный Grower.

1. Передаем Grower с фиксированным размером, без ресайза и разрешения цепочек коллизий получаем Lookup-таблицу
2. Передаем Grower с шагом разрешения коллизий не 1

C++ дизайн хэш-таблицы

Возможность хранить состояние в ячейке.

Сохранять хеш. Используется для строковых хэш-таблиц.

```
struct HashMapCellWithSavedHash : public HashMapCell
{
    size_t saved_hash;

    void setHash(size_t hash_value) { saved_hash = hash_value; }

    size_t getHash(const Hash &) const { return saved_hash; }
};
```

С++ дизайн хэш-таблицы

Быстроочищаемая хеш-таблица

```
struct FixedClearableHashMapCell
{
    struct ClearableHashSetState
    {
        UInt32 version = 1;
    };

    using State = ClearableHashSetState;
    UInt32 version = 1;

    bool isZero(const State & st) const { return version != st.version; }

    void setZero() { version = 0; }
};
```

C++ дизайн хэш-таблицы

LRUCache

```
struct LRUHashMapCell
{
    static bool constexpr need_to_notify_cell_during_move = true;
    static void move(LRUHashMapCell * old_loc, LRUHashMapCell * new_loc);
    LRUHashMapCell * next = nullptr;
    LRUHashMapCell * prev = nullptr;
};
```

C++ дизайн хэш-таблицы

LRUCache. Используем `boost::intrusive::list`.

```
using LRUList = boost::intrusive::list
    <
        Cell,
        boost::intrusive::value_traits<LRUHashMapCellIntrusiveValueTraits>,
        boost::intrusive::constant_time_size<false>
    >;

LRUList lru_list;
```

https://www.boost.org/doc/libs/1_76_0/doc/html/intrusive.html

Специализированные хэш-таблицы

SmallTable

Состоит из массива на некоторое небольшое количество элементов.
Помещается в L1-кэш.

```
template <typename Key, typename Cell, size_t capacity>
class SmallTable : protected Cell::State
{
    size_t m_size = 0;
    Cell buf[capacity];
    ...
}
```


Специализированные хэш-таблицы

StringHashTable

Состоит из 4 хэш-таблиц:

1. Для строк размером 0-8 байт
2. Для строк размером 9-16 байт
3. Для строк размером 17-24 байта
4. Для строк размером больше 24 байт

https://www.researchgate.net/publication/339879042_SAHA_A_String_Adaptive_Hash_Table_for_Analytical_Databases

Специализированные хэш-таблицы

TwoLevelHashTable

Состоит из 256 хэш-таблиц.

На вставке ключа мы вычисляем индекс хэш-таблицы в которую будем вставлять ключ.

```
size_t getBucketFromHash(size_t hash_value)
{
    return (hash_value >> (32 - BITS_FOR_BUCKET)) & MAX_BUCKET;
}
```

Заключение

Мы написали фреймворк для хэш-таблиц под свой сценарий агрегации данных.

<https://github.com/ClickHouse/ClickHouse/blob/master/src/Common/HashTable/HashTable.h>

https://github.com/ClickHouse/ClickHouse/blob/master/src/Common/examples/integer_hash_tables_benchmark.cpp



C++ Zero Cost
Conf ^{*/}



Спасибо за внимание

Максим Кита
Старший разработчик ClickHouse

maksim-kita@yandex-team.ru

[@kitaisreal](#)



Yandex for **developers** ^{*/}>