**Федеральное государственное автономное образовательное**
**учреждение высшего образования**
**«Национальный исследовательский университет**
**«Высшая школа экономики»**

**Факультет компьютерных наук**
**Основная образовательная программа**
**Прикладная математика и информатика**

# КУРСОВАЯ РАБОТА
## Программный проект на тему
## Взаимная интеграция аллокатора и кеша

**Выполнил студент группы 176 3 курса,**
**Кот Михаил Евгеньевич**

**Руководитель КР:**
**руководитель разработки ClickHouse, Миловидов Алексей Николаевич**

**Москва 2020**

# Оглавление

# Аннотация

## Русский

В курсовой работе представлен интерфейс кеширующего аллокатора для некоторых кешей ClickHouse. Интерфейс использует технику манипулирования большими объемами памяти внутри, сокращая время на взаимодействие с операционной системой по вопросам выделения и освобождения памяти. В работе также приведено описание интеграции аллокатора в кеши.

## Английский

This coursework presents an interface combining a mmap-backed reference-counting memory allocator with a domain-specific cache storage. The interface manages large data chunks to reduce time needed to interact with the OS. The work also contains the description of the process of integrating the allocator with various ClickHouse caches.

## Ключевые слова

c++17, mmap-backed allocator, domain-specific cache, clickhouse.

# Введение

## Постановка задачи

Задачу курсовой работы можно переформулировать следующим образом: есть программа, которая использует некоторый функционал по выделению/освобождению памяти, в этой программе поддерживается кеш некоторой структуры данных.
Предположим для простоты, что аллокатор стандартный (stdlib malloc).
Предположим также, что кеш может содержать не более M элементов (ограничение задается при старте программы в каком-то конфигурационном файле), и в текущий момент все M элементов заняты. Приходит запрос на добавление еще одного (или нескольких) элементов. Старые элементы деаллоцируются, новые добавляются на освободившееся пространство.

1. Можно ли сделать так, чтобы структуры, которые были деаллоцированы, как-то сохранялись в памяти (при соблюдении определенных условий), и были доступны для чтения?
2. Можно ли избавиться от хранения кеша аллокатора как набора байт в каком-то месте?

Решение этой задачи заключается в совмещении т.н. domain-specific кеша вышеуказанных данных (т.е. кеша, реализуемого непосредственно разработчиком программы) и кеша свободных блоков аллокатора. Для этих целей создается новый класс, который одновременно занимается и выделением/освобождением памяти под какие-то объекты, и поддерживанием кеша этих данных.

## Цель работы

- Разработать интерфейс, отвечающий постановке задачи.
- Интегрировать его в некоторые кеши ClickHouse.
- Добиться большего быстродействия кешей по сравнению с предыдущей реализацией.

## Актуальность, значимость и новизна

Предполагается, что реализация интерфейса позволит ускорить обработку запросов ClickHouse. Новизна работы обуславливается тем, что раньше попытки разработать такой интерфейс не доводились до конца.

## Корректность результатов

Корректность результатов подтверждается результатами интеграционных и функциональных тестов на GitHub и внутренних площадках Яндекса. Улучшение быстродействия подтверждается результатами тестов производительности. Сводные результаты в виде таблиц приведены в пункте "Результаты тестирования".

## Структура работы

В разделе "Обзор литературы" приводятся различные кеширующие аллокаторы, которые тем или иным образом повлияли на разработку, с указанием их краткого механизма работы и описанием того, что было взято для реализации в интерфейс.
В разделе "Вводные определения" даются определения и примеры (в том числе, специфичные для C++), которые будут использованы позже.
В разделе "Краткий обзор ClickHouse и ее архитектуры применительно к задаче" рассматривается релевантная часть кода, отвечающая за обработку запросов и кеширование данных.
В разделе "Реализация аллокатора" рассматриваются публичный и приватный интерфейсы получившегося класса, алгоритмы и структуры данных (включая собственные), которые были использованы. В разделе также рассматриваются различные ошибки, которые были найдены в процессе разработки (включая ошибки в компиляторах). Отдельная часть раздела посвящена вопросу синхронизации при использовании аллокатора в многопоточном контексте (с описанием возможных ошибок и примеров их исправления).

В разделе "Интеграция аллокатора в ClickHouse" описаны изменения интерфейсов, необходимые для поддержки аллокатора в некоторых кешах данных. В разделе также рассматривается изменение внутренних контейнеров для адаптации к использованию этого аллокатора.
В разделе "Результаты тестирования" приведены численные метрики, рассчитанные по результатам внедрения аллокатора в ClickHouse.
В разделе "Заключение" приведены временные затраты на разработку и выводы, обобщающие тему этой курсовой работы
В разделе "Источники" приведены ссылки на используемые источники.

# Обзор литературы

Стоит понимать, что деталей реализации вышеуказанного класса может быть огромное количество, поэтому нельзя найти в точности подходящее решение (для рассмотрения в списке альтернатив). Далее я рассмотрю 5 примеров кеширующих аллокаторов (и один пример внутренней реализации кеша) с той оговоркой, что они

1. Не позволяют получить доступ к памяти, которая считается деаллоцированной, но на самом деле хранится в программе и может быть ей затребована (без необходимости вовлекать в этот процесс ядро операционной системы).
2. Рассматривают память как массив байт, а не массив структур (с соответствующей невозможностью проинтерпретировать эти данные и отдать их внешнему наблюдателю); не распространяется на LRUCache.

Рассматриваемые примеры касаются аллокаторов, работающих в пространстве пользователя, так как целевая программа (ClickHouse) тоже будет работать в пространстве пользователя. от не-ядерных).

## glibc malloc (ptmalloc)

Если чуть точнее, стандартный для C и C++ malloc -- это ptmalloc (ptread malloc), который отпочковался (forked) от dlmalloc.
Использует разделение по размерам запрашиваемых регионов памяти:

3. Маленькие (до 256 бит) аллокации обрабатываются выделением памяти ближайшей степенью двойки,
4. Средние (257 бит до порогового значения mmap (то есть, те, для которых mmap был бы неоптимален с точки зрения расходуемого пространства)) обрабатываются через trie (поиском значения, а затем смещением границы кучи с помощью системного вызова sbrk),
5. Большие аллоцируются непосредственно mmap-ом.

Кеширование свободных регионов памяти делается через двусвязный список чанков (заголовок кеширующей структуры, за которым следуют данные, длина вычисляется с помощью указательной арифметики путем вычитания адресов двух соседних заголовков).
Неудобство в том, что нельзя выделять память страницами, не во всех реализациях есть поддержка Hugepages.
Из этого аллокатора я взял алгоритм выделения больших участков памяти. По постановке задачи не предполагается наличие регионов размером меньшим, чем пороговое значение mmap, аллокацию через trie и степень двойки можно убрать.

## jemalloc (Jason Evans malloc)

Стоит отметить, что в изначальной задаче предполагается, что аллокатор будет локальным для процесса (но может быть совместно использован некоторыми потоками; этому посвящен, например, раздел Конкурентная инициализация), этот же аллокатор рассматривается для использования в многопоточных, и более -- многопроцессорных системах.
Основной упор в нем делается на раздельные арены (хранилища памяти, выделенной ядром операционной системы), но интересен этот источник подходом к кешированию -- в отличие от glibc, добавлены несколько функций, включая mallctlnametomib(), которые должны, цитата, "предотвращать повторяющиеся запросы частей (одного размера памяти, имеется в виду)". Для этого поддерживается некоторая база данных из хеш-таблицы и красно-черного дерева.
Из этого аллокатора я взял подход по использованию красно-черного дерева для хранения регионов памяти, упорядоченных по размеру (где одному ключу-размеру может соответствовать несколько регионов).

## tcmalloc (Thread-caching malloc, Google)

Этот аллокатор наиболее похож на то, что я делаю (с точки зрения изначальной постановки задачи), хотя реализация у него значительно более сложна. Сделан похоже на jemalloc, только вместо хеш-таблицы использует логарифмические корзины по размерам выделяемой памяти (2 - 2^10, 2^10 + 1, ...,) и по типам размеров (меньше границы mmap, меньше границы hugepages, hugepages), к каждой корзине прилагается своя кеширующая структура (не буду описывать, потому что их там действительно много).
Из этого аллокатора я не взял ничего. В процессе обсуждения интерфейса была идея использовать логарифмические корзины, но предварительные расчеты показали, что оптимальность линейного поиска по корзине в среднем будет ниже логарифмического поиска по красно-черному дереву, к тому же, ухудшится локальность данных, что приведет к еще большему понижению производительности.

## Hoard malloc

В отличие от mimalloc и tcmalloc, использует так называемую концепцию суперблоков (как в файловой системе), для каждого суперблока используется как бы свой аллокатор.
Не очень интересен для конкретной задачи, потому что объем аллокаций будет явно меньше, чем предполагается в hoard malloc, причем, фрагментации можно не бояться, так как известна верхняя граница и знаменатель (минимальный размер) блока.
Из этого аллокатора я тоже ничего не взял по результатам рассмотрения.

## mimalloc (Microsoft)

Считается одним из самых простых аллокаторов. Принципиально интересных идей две -- постраничное кеширование (с точки зрения страниц памяти) и т.н. "ранний сброс страницы" -- уведомление операционной системы о том, что страница не используется или в ближайшее время может стать неиспользуемой.
Из этого аллокатора я взял использование флага MMAP_POPULATE (описан позже).

## LRUCache (Внутрений кеш в Clickhouse)

Стоит заметить, что это не аллокатор. Однако, именно эта структура использовалась в качестве родителя в кешах, поэтому я посчитал ее интересной для рассмотрения.
Общая структура такова: есть

1. std::unordered_map<Key, {std::shared_ptr<Value>, size, iter}, хранящая собственно пары ключ-значение. Значения выделены где-то в сегменте данных, нас это не интересует.
2. Односвязный список ключей.
3. Хеш-таблица запросов на добавление.

Во избежание дублирования описания алгоритмов скажу, что мой класс сделан по подобию этого, из LRUCache взяты запросы на добавление, алгоритм параллельной инициализации, и многое другое (описано ниже в соответствующих разделах).

# Вводные определения.

## Аллокатор, аллокация, деаллокация.

В общем случае -- некоторый класс или набор функций, который занимается выделением памяти (аллокацией) на стеке или в сегменте данных процесса, ее освобождением (деаллокацией), а так же хранит некоторые метаданные о выделенных регионах для повышения быстродействия программы. Точный интерфейс аллокатора не определен и, как мы далее увидим, может быть совершенно разным.

## Фабрика, фабричный метод

Шаблон проектирования, при котором некоторый объект-фабрика может создавать объекты произвольного типа, не зная о них практически ничего. Далее в работе под фабричным методом будет подразумеваться метод, создающий, удаляющий или модифицирующий некоторый объект без вызова его конструкторов или деструкторов.
Пример фабричного метода:

```
struct Obj
{
  static Obj* create() { return new Obj(); }
}
```

## Вариативный шаблон (вариадик, variadic template)

Шаблон в C++, принимающий произвольное число аргументов.
В общем случае (на примере вариадика для функции) определяется как

```
template <class FooT types...>
void foo(FooT&& ... types);
```

где FooT... -- множество типов аргументов в порядке их передачи, ...types -- множество передаваемых аргументов. Стоит заметить, что вариативный шаблон может быть пустым. В таком случае sizeof...(FooT) == 0, функция не принимает никаких аргументов.

## Функтор

Обобщение функции. Любой объект, у которого определен/перегружен operator()(), в любой форме (не принимающий аргументов, принимающий один или несколько аргументов, принимающий вариативный шаблон с переменным числом аргументов).
Пример функтора:

```
struct Functor
{
   int operator()() const {return 42; }
}
```

## Заголовочный файл

Стоит заметить, что в этой работе я использую понятие "Заголовочный файл" в несколько нестандартном смысле. Обычно заголовочный файл или h-файл -- код, содержащий декларацию класса и его методов. Реализация же методов находится в другом файле (обычно с расширением

cpp). Это сделано для логического отделения интерфейса от его реализации и ускорения компиляции.
Я же использую это значение в смысле "декларация части какого-то интерфейса". Фактически, весь код аллокатора находится в одном файле IGrabberAllocator.h, но для удобства читателя в разных разделах имеет смысл приводить его части.

## Мьютекс

От английского mutual exclusion -- взаимное исключение. Некоторый объект-точка синхронизации, позволяющая среди нескольких потоков только одному за раз получить некоторый доступ к критической секции (строго говоря, это не всегда так, но shared_mutex в моем коде не используется). Устанавливает глобальный порядок на множестве путей исполнения программы.
Пример мьютекса -- std::mutex из стандартной библиотеки.

## Состояние гонки (data race, race, рейс)

Состояние в процессе выполнения программы, когда два или более потоков могут получить одновременный доступ на чтение/запись (или любую комбинацию чтений/записей) какой-либо части. Таким образом, результат выполнения этой части не предопределен и зависит только от переупорядочивания выполнения потоков планировщиком потоков. Устраняется мьютексом.
Пример: два потока std::thread, пытающихся модифицировать не-атомарную переменную.

## Дедлок (deadlock, взаимная блокировка).

Состояние, при котором два или несколько потоков (во время исполнения программы) при попытке доступа к каким-либо мьютексам взаимно заблокировались и не могут продолжать работу дальше.
Пример: Пусть есть два потока и два мьютекса. Рассмотрим следующее возможное исполнение программы.

- 1. Поток 1 блокирует мьютекс 1
- 2. Планировщик потока передает управление потоку 2.
- 3. Поток 2 блокирует мьютекс 2 и хочет заблокировать мьютекс 1 (но не может).
- 4. По истечению кванта времени планировщик передает управление потоку 1.
- 5. Поток 1 пытается блокировать мьютекс 2 (но не может).
- 6. Дальнейшее исполнение программы невозможно.

## Конкурентная инициализация

То же самое, что и параллельная инициализация -- действие, когда несколько потоков пытаются получить доступ к секции. Порядок доступа устанавливается некоторым мьютексом.

## Краткий обзор ClickHouse и ее архитектуры применительно к задаче

ClickHouse - столбцовая система управления базами данных (СУБД) для онлайн обработки аналитических запросов (OLAP). Основной упор делается на скорость обработки запросов. Кратко рассмотрим релевантную часть исполнения программы, ответственную за обработку запросов:

- Серверу приходит запрос по одному из интерфейсов (http, tcp, odbc).
- Перед непосредственно посылкой запрос парсится, в зависимости от интерфейса, есть разные реализации: TCPHandler, HTTPHandler, и так далее.
- Вызывается метод executeQuery, строящий конвейер по тексту запроса.
  - Запрос парсится.
  - Преобразовывается из строки в AST
  - Для конкретного запроса достается конвейер из Interpeters.
  - Вызывается execute, возвращающий конвейер (на примере SELECT -- InterpreterSelectQuery).
- Конвейер постепенно обходит AST.
- В какой-то момент происходит чтение из таблицы. Чтение из таблицы, в свою очередь, возвращает часть конвейера, из которого можно читать.
- находится процессор, читающий из MergeTree (определенный движок). Его задача - реализовать метод generate, который отдает части столбцов с данными.
- MergeTreeRangeReader (используемый внутри generate )решает, какие засечки и в каком количестве нужно прочитать.
- MergeTreeRangeReader в свою очередь использует интерфейс IMergeTreeReader, который может быть реализоват через MergeTreeReaderWide

# Реализация аллокатора

## Шаблонные параметры

Правильный выбор шаблонных параметров позволяет избежать громоздкости публичного интерфейса и вынести часть вычислений на этап компиляции. Были выбраны 8 основных шаблонных параметров, из которых 6 -- шаблонные параметры без ограничения типа (template type parameters) и 2 -- шаблонные параметры с определением типа (template non-type arguments). Изначально предполагалось добавить концепты вместо первых шести параметров, но в процессе разработки пришлось отказаться от этой идеи. Более подробно в разделе [Процесс разработки/ концепты].

Так выглядит шаблонный интерфейс класса:

```
template <
    class TKey,
    class TValue,
    class KeyHash = std::hash<TKey>,
    class SizeFunction = ga::Runtime,
    class InitFunction = ga::Runtime,
    class ASLRFunction = AllocatorsASLR,

    size_t MinChunkSize = MMAP_THRESHOLD,
    size_t ValueAlignment = ga::defaultValueAlignment<TValue>>
class IGrabberAllocator;
```

Рассмотрим каждый из параметров.

### `TKey`

Ключ, по которому будет адресоваться значение. Уникален для значения. Пример типа ключа -- UInt128.

Необходимые условия:

- Публичный конструктор копирования (для инициализации в RegionMetadata::init_key, подробнее в разделе [Используемые структуры данных / Внутренние / RegionMetadata]).
- Публичный и определенный operator<(const TKey& other) (для добавления в free_regions и used_regions)

Неформально: малый размер. Вместе с экземпляром класса TValue объект данного класса будет храниться в RegionMetadata, следовательно, должен быть удобен для передачи, копирования и хранения.

### `TValue`

Значение, хранимое в контейнере. Предполагается, что объект будет реализовывать шаблон проектирования "динамический контейнер", то есть, подобно std::vector, хранить внутри некоторое количество "легковесных" данных (вроде пары указателей и size_t), указывающих на память, аллоцированную не на стеке.

Пример типа значения -- DB::PODArray.

Необходимые условия (помимо вышеуказанного):

- Возможность передать аллокатор в качестве шаблонного параметра. Необходимо для подмены указателей на данные в сегменте данных. Более подробно -- в разделе [Используемые алгоритмы/Подмена аллокатора].
- Публичный конструктор копирования -- необходим для инициализации функции. Подробнее в разделах [Публичный интерфейс/ init_func] и [Используемые алгоритмы/ инициализация значения].

### TKeyHash

Хеш-функция на объектах класса TKey. Необходима для адресации запросов на добавления элемента (подробнее в разделе [Приватный интерфейс/ insertion_attempts]).
По умолчанию используется std::hash<TKey>, если такая специализация шаблона определена (в противном случае будет ошибка компиляции).
Пример типа -- DB::UInt128TrivialHash.

### SizeFunction

Функтор без аргументов, позволяющий вычислить верхнюю границу на предполагаемый объем данных, хранимых в сегменте данных.
Есть две возможных ситуации:

- Каждый объект класса TValue выделяет в сегменте данных блок одного и того же размера, известного на этапе компиляции. В таком случае, пользователь может задать этот шаблонный параметр, что позволит компилятору соптимизировать его вызовы в коде (разумеется, для этого перегруженный operator() должен быть помечен атрибутом constexpr или consteval)

  Пример: Объект класса TValue = Foo хранит в себе указатель char * и выделяет в сегменте данных 1024 байта. В таком случае, можно написать следующий функтор для SizeFunction:

  ```
  template<class TValue>    struct FooSizeFunction    {
          consteval size_t operator()() const noexcept       {          return 1024;       }
  }
  ```

  Заметим, что передаваемых аргументов нет. Это сделано потому, что от размера объекта размер выделяемой памяти не зависит, и это можно явно указать компилятору для дальнейшей оптимизации.
- Объекты класса TValue могут выделять в сегменте данных блоки разного размера. В таком случае, пользователь устанавливает параметр SizeFunction в ga::Runtime.

ga::Runtime -- специальная структура-заглушка, указывающая, что целевой функтор (в нашем случае, SizeFunction) будет передан в качестве аргумента в функцию getOrSet (подробнее в разделе [Публичный интерфейс/ getOrSet]).

### InitFunction

Функтор, который инициализирует временное значение, от которого позже будет сконструирован объект класса TValue, и подменяет указатели на полученные.
Вызывается с параметром void * heap_storage, который передается туда из DB::RegionMetadata (подробнее в разделе [Используемые алгоритмы/ Инициализация]) и [Публичный интерфейс/getOrSet].
Как и в предыдущем шаблонном параметре, если пользователь не намерен выполнять какую-то дополнительную работу до/при инициализации, он может определить функтор и передать его как шаблонный параметр.
Пример:
Для некоторого TValue = Foo определяем следующий функтор:

```
struct FooInitFunction { SomeTempValue operator(void * heap_storage) { return {heap_storage}; } }
```

Заметим, что объект, возвращаемый этим функтором, не обязательно должен каким-либо образом относиться к TValue (совпадать/быть родственным в какой-то иерархии наследования), если у

TValue определен const lvalue& конструктор копирования от возвращаемого объекта. Про мотивацию подробнее в разделе [Публичный интерфейс/init_func].

В противном случае, пользователь может также указать InitFunction = ga::Runtime и передавать экземпляр функтора как аргумент в функцию getOrSet.

### ASLRFunction

Этот шаблонный параметр определяет функтор, занимающийся дополнительной рандомизацией адресного пространства (ASLR -- address space layout organization). Рандомизация адресного пространства позволяет дополнительно защитить приложение от ошибок вроде return to libc. Ядро Linux по умолчанию выполняет ее при запуске процесса, однако, можно дополнительно выполнять рандомизацию по адресам выделяемых блоков данных (с помощью mmap).
Второе возможное использование этой структуры -- упрощение отладки приложения. Можно генерировать адреса в узком диапазоне (или, например, помещать эти адреса далеко от адресов, выдаваемых стандартным системным аллокатором), чтобы получать больше ошибок, связанных с конфликтным доступом к памяти (memory stomping bugs).
Функтор вызывается с генератором случайных чисел pcg64 и возвращает void *, указывающий на некоторый адрес в памяти, на котором в качестве address_hint можно будет вызвать mmap.
По умолчанию используется AllocatorsASLR, привожу его в качестве примера.

```
struct AllocatorsASLR
{
#if ALLOCATOR_ASLR == 1
   [[nodiscard, gnu::const]] void * operator()(pcg64& rng = thread_local_rng) const noexcept
   {
      return reinterpret_cast<void *>(
            std::uniform_int_distribution<size_t>(
               0x100000000000UL,
               0x700000000000UL)(rng));
   }
#else
   [[nodiscard, gnu::const]] constexpr void * operator()(pcg64&) const noexcept
   {
      return nullptr;
   }

   [[nodiscard, gnu::const]] constexpr void * operator()() const noexcept
   {
      return nullptr;
   }

#endif
};
```

Макрос ALLOCATORS_ASLR устанавливается в 1 в режимах сборки с отладочной информацией (Debug, RelWithDebInfo).

### MinChunkSize

Этот шаблонный параметр с указанием типа size_t используется в качестве нижней границы на размер региона памяти. Поскольку производительность mmap на небольших участках (по размеру) памяти ниже, чем у обычных аллокаторов (из обзора литературы, например), предполагается, что параметр будет установлен в сравнительно большое значение (50-70МБ).
По умолчанию параметр равен константе MMAP_THRESHOLD, о котором я сейчас расскажу.

В ClickHouse есть несколько аллокаторов (мой интерфейс, Allocator, Memory и так далее). Поскольку все они используют один и тот же системный вызов, возникла потребность где-то определить константу (в зависимости от которой некоторые другие аллокаторы могут использовать другой метод аллокации, например).
В режиме с сохранением отладочной информации эта константа устанавливается в 4096 байт, поскольку малый размер региона вкупе с ASLR позволит получить больше ошибок конфликтующего доступа к памяти.
В режиме "production ready" константа установлена в 75МБ, что примерно соответствует порогу, начиная с которого mmap выигрывает у аллокаторов стандартной библиотеки по скорости.
Константа определена в заголовочном файле allocatorsCommon.h.

#### `ValueAlignment`

Этот шаблонный параметр, также с указанием типа size_t, отвечает за минимальное выравнивание объекта типа TValue при размещении его в RegionMetadata. По умолчанию берется константа времени компиляции alignof(TValue).

## Публичный интерфейс

Примечание: Этот раздел описывает интерфейс со стороны пользователя. Алгоритмы, используемые в функциях интерфейса,
подробно описываются в разделе [Используемые алгоритмы].
Публичный интерфейс класса (в формате заголовочного файла) приведен ниже:

```
class IGrabberAllocator : private boost::noncopyable
{
public:
    using Key = TKey;
    using Value = TValue;
    using ValuePtr = std::shared_ptr<Value>;
    using GetOrSetRet = std::pair<ValuePtr, bool>;

    constexpr IGrabberAllocator(size_t max_cache_size_);
    ~IGrabberAllocator() noexcept;

    constexpr size_t getSizeInUse() const noexcept;
    constexpr size_t getUsedRegionsCount() const noexcept;

    void shrinkToFit(bool clear_stats = true);
    void reset();

    [[nodiscard]] ga::Stats getStats() const noexcept;

    inline ValuePtr get(const Key& key);

    template<class S = SizeFunction, class I = InitFunction>
    inline typename std::enable_if<!std::is_same_v<S, ga::Runtime> &&
                                   !std::is_same_v<I, ga::Runtime>,
        GetOrSetRet>::type getOrSet(const Key & key);

    template<class Init, class S = SizeFunction, class I = InitFunction>
    inline typename std::enable_if<!std::is_same_v<S, ga::Runtime> &&
                                    std::is_same_v<I, ga::Runtime>,
```

```
        GetOrSetRet>::type getOrSet(const Key & key, Init && init_func);

    template<class Size, class S = SizeFunction, class I = InitFunction>
    inline typename std::enable_if<std::is_same_v<S, ga::Runtime> &&
                                   !std::is_same_v<I, ga::Runtime>,
        GetOrSetRet>::type getOrSet(const Key & key, Size  && size_func);

    template <class Init, class Size>
    inline GetOrSetRet getOrSet(const Key & key, Size && get_size, Init && initialize);
}
```

### Конструктор

Инициализирует максимальный размер кеша и всех связанных с ним структур. За размер кеша считается сумма размеров всех регионов памяти, обрабатываемых в MemoryChunk (то есть, сумма полей size у объектов данных классов). Метаданные, такие как размеры TKey и TValue, при расчете размера кеша не учитываются -- предполагается, что они будут сравнительно малы).
При максимальном размере кеша меньше, чем минимальный размер региона памяти (шаблонный параметр MinChunkSize), выбрасывается исключение с кодом ошибки DB::ErrorCodes::BAD_ARGUMENTS.

### Деструктор

Занимается очисткой интрузивных контейнеров (подробнее в разделе [Используемые структуры данных / Внешние / Интурузивные контейнеры]).

### getSizeInUse

Получает константу типа size_t, отвечающую за суммарный размер регионов памяти, который а) инициализирован; б) используется (в терминах c++ это означает, что вне аллокатора существует хотя бы один объект типа ValuePtr в инициализированном состоянии, то есть, содержащий внутри не nullptr).
Функция используется в классе AsyncronousMetrics, который занимается подсчетом различных метрик во время работы приложения.

### getUsedRegionsCount

Получает константу типа size_t, отвечающую за количество пар TKey - TValue, которые используются (это означает, что с ними связан используемый регион).
Функция используется аналогично предыдущей.

### shrinkToFit

Функция реализована сообразно функциям-членам контейнеров из стандартной библиотеки.
Пример: у std::vector функция shrink_to_fit освобождает аллоцированную, но не занятую память, приводя параметр
capacity (аллоцированная память, в которую можно размещать элементы без реаллокации контейнера) к параметру size
(память, занятая элементами).
При вызове она очищает все неиспользуемые регион (то есть, те, на которые нет ни одной ссылки снаружи аллокатора), все свободные регионы (то есть, те, которые были выделены, но не проинициализированы), и, при необходимости, всю статистику (за это отвечает параметр clear_stats). Используемые регионы памяти, равно как и связанные с ними пары ключа-значение, остаются незменными.
В качестве одного из примеров использования функции можно привести метод класса Context::dropAllCaches(), который при окончании обработки запроса и выдачи данных (подробнее в

разделе [ClickHouse/ Обработка запросов]) сбрасывает все доступные кеши данных (в т.ч. кеш засечек и кеш несжатых данных).

### reset

Выполняет полную инвалидацию кеша. Выполняет shrinkToFit, затем помечает все используемые регионы (в метаданных) как "удаленные" (disposed). Первый запрос на чтение или запись в таком регионе приведет к его удалению.
Примечание: стоит заметить, что в интерфейсе ClickHouse нужды в методе reset нет, и для коррекной работы достаточно
shrinkToFit. Функциональность reset, описанная выше, предполагается опциональной для реализации; во всех остальных
случах (в текущей реализации, в частности) reset() просто вызывает shrinkToFit().

### getStats

Функция возвращает статистику метрик аллокатора. Статистика определена в файле IGrabberAllocator_fwd.h, заголовочный файл приведен ниже.

```
struct Stats
{
    size_t chunks_size {0};
    size_t chunks{0};

    size_t allocated_size {0};
    size_t used_size {0};
    size_t initialized_size {0};

    size_t regions {0};
    size_t free_regions {0};
    size_t used_regions {0};
    size_t unused_regions {0};

    size_t hits {0};
    size_t concurrent_hits {0};
    size_t misses {0};

    size_t allocations {0};
    size_t allocated_bytes {0};
    size_t evictions {0};
    size_t evicted_bytes {0};
    size_t secondary_evictions {0};

    template <class Ostream>
    void print(Ostream& out_stream) const noexcept;
```

Функция-член print выполняет текстовый вывод в поток содержимого объекта данного класса.

#### *chunks_size*

Суммарный размер аллоцированных регионов памяти.

#### *chunks*

Количество объектов класса MemoryChunk, то есть, количество аллоцированных регионом памяти.

*allocated_size*
Суммарный размер аллоцированной памяти. Включает в себя chunks_size и размеры метаданных.

*used_size*
Суммарный размер памяти, которая в данный момент используется (это означает, что на каждый RegionMetadata, соответствующий участку памяти, есть хотя бы одна ссылка).

*initialized_size*
Суммарный размер памяти, который был инициализирован (то есть, в нем были размещены какие-то данные).

*regions*
Суммарное количество регионов (пар TKey - TValue).

*free_regions*
Количество неинициализированных регионов.

*used_regions*
Количество регионов, на которых есть хотя бы одна ссылка.

*unused_regions*
Количество регионов, на которых нет ни одной ссылки.

*hits*
Количество запросов ключа, которые завершились успехом (значение было либо найдено в кеше, либо получено из другого потока).

*concurrent_hits*
Количество запросов ключа, которые завершились получением значения из другого потока.

*misses*
Количество запросов ключа, завершившихся неудачей (значение не было ни найдено, ни получено).

*allocations*
Количество выделений памяти с помощью mmap.

*allocated_bytes*
Сумма количества байт, выделенных за время жизни программы.

*evictions*
Количество вытеснений неиспользуемых данных из кеша (подробнее в разделе [Используемые алгоритмы/Вытеснение данных]).

*evicted_bytes*
Сумма количества вытесненных байт.

*secondary_evictions*
Количество вытеснений, для которых было недостаточно объединение региона с левым или правым соседом.

**get**
Функция пытается получить из кеша элемент, соответствующий переданному ключу. Если ключ не найден, функция создает запрос на получение ключа (подробнее в разделе [Используемые алгоритмы/конкурентная инициализация]). Если же и в этом случае ключ оказывается не найден, функция возвращает nullptr.

### getOrSet

Функция сначала пытается получить значение из кеша. Если функция get вернула nullptr, производится аллокация нужного места (вызов функтора get_size), затем размещение значения в кеш (вызов функтора initialize), затем инициализированное значение возвращается с обновлением ссылок и контейнеров.
Для тех случаев, когда один из шаблонных параметров SizeFunction, InitFunction установлен не в значение ga::Runtime, предусмотрены перегрузки функции, принимающие меньшее число аргументов. Это сделано с помощью механизма SFINAE, позволяющего "отключить" перегрузки в тех случаях, когда они должны быть недоступны.

## Приватный интерфейс

Приватный интерфейс класса даже в виде заголовочного файла достаточно велик, чтобы приводить его за раз (110 строк кода), рассмотрим его по логическим частям.

### Статические члены и глобальный мьютекс

```
static constexpr const size_t page_size = 4096;

mutable std::mutex mutex;

static constexpr auto ASLR = ASLRFunction();
static constexpr auto getSize = SizeFunction();
static constexpr auto initValue = InitFunction();

Logger& log;
```

*page_size*

Размер страницы памяти, используется для округления вверх размера выделяемой памяти, полученного из функтора get_size.

*mutex*

Глобальный мьютекс. Защищает доступ к интрузивным контейнерам (кроме used_regions), а так же контейнерам value_to_region и chunks.

*ASLR, getSize, initValue*

Экземпляры функторов, заданных через шаблонные параметры. В случае, если шаблонные параметры были установлены в ga::Runtime, инстанциируется эта структура (пустая, таким образом, расход памяти составляет примерно 3 байта в зависимости от архитектуры).

*log*

Объект библиотеки POCO, используемой ClickHouse. Выводит в стандартный поток вывода сообщение о невозможности аллокатора выделить память под какой-либо объект.

### Параметры, используемые для статистики

```
const size_t max_cache_size;

size_t total_chunks_size {0};
size_t total_allocated_size {0};

std::atomic_size_t total_size_in_use {0};

std::atomic_size_t total_size_currently_initialized {0};
```

```
std::atomic_size_t hits {0};
std::atomic_size_t concurrent_hits {0};
std::atomic_size_t misses {0};

size_t allocations {0};
size_t allocated_bytes {0};
size_t evictions {0};
size_t evicted_bytes {0};
size_t secondary_evictions {0};
```

Эти параметры накапливают статистику, которую можно получить в методе getStats. Атомарность некоторых переменных обуславливается возможными гонками при их обновлении и рассматривается в разделе Используемые алгоритмы.

### Структуры, используемые для хранения метаданных

Типы и описания их работы находятся в разделе [Используемые структуры данных/Внешние/Интрузивные контейнеры].

```
TRegionsList all_regions;

TFreeRegionsMap free_regions;

TUsedRegionsMap used_regions;
mutable std::mutex used_regions_mutex;

TUnusedRegionsList unused_regions;
```

*all_regions*

Хранит в себе (в терминах интрузивного контейнера) все объекты класса RegionMetadata, созданные в программе. В некотором смысле, ответственен за их удаление, поскольку в деструкторе классе именно к all_regions применяется метод erase_and_dispose.

*free_regions, used_regions, unused_regions*

Хранят в себе свободные (но не инициализированные), используемые и неиспользуемые (но инициализированные) контейнеры соответственно. Для used_regions нужен свой мьютекс из-за поиска в getImpl, подробнее в разделе [Используемые алгоритмы / getImpl].

### Структуры, используемые для хранения регионов и запросов на добавление

```
std::unordered_map<const Value*, RegionMetadata*> value_to_region;

std::list<MemoryChunk> chunks;

std::unordered_map<Key, std::shared_ptr<InsertionAttempt>, KeyHash> insertion_attempts;
std::mutex attempts_mutex;
```

*value_to_region*

Используется для того, чтобы в деструкторе ValuePtr найти RegionMetadata, соответствующий удаляемому значению, и корректно обновить глобальное состояние.

*chunks*

Хранит список регионов памяти (на части которых ссылаются объекты класса RegionMetadata).

*insertion_attempts*

Хранит список запросов на добавление значения для ключа (было описано в публичном интерфейсе

функции get). Для контейнера необходим отдельный мьютекс для упрощения синхронизации глобального состояния.

### Регион памяти, метаданные, запрос на добавление

```
struct InsertionAttempt;
struct InsertionAttemptDisposer

struct MemoryChunk;
struct RegionMetadata;

static constexpr auto region_metadata_disposer = [](RegionMetadata * ptr) { ptr->destroy(); };
```

Соответствующие структуры будут подробно описаны в разделе [Используемые структуры данных/Собственные]. region_metadata_disposer -- функтор, передаваемый в интрузивные контейнеры с тем, чтобы удалить объект из памяти при его удалении из интрузивного контейнера.

### Конкурентное создание и удаление ссылок на значение, реализация get

```
  template <bool MayBeInUnused>
  void onSharedValueCreate(RegionMetadata& metadata) noexcept;

  void onValueDelete(Value * value) noexcept;

  inline ValuePtr getImpl(const Key& key, InsertionAttemptDisposer& disposer, InsertionAttempt *& attempt);
```

*onSharedValueCreate*

Функция вызывается при создании новой ссылки на объект класса TValue. Занимается обновлением ссылок на объект. Шаблонный параметр MayBeInUnused обусловлен вызовом функции из двух мест (getOrSet, getImpl) и отвечает за то, будет ли объект удаляться из контейнера unused_regions.

*onValueDelete*

Функция вызывается в деструкторе ValuePtr, позволяя корректно обновить глобальное состояние при разрушении одной ссылки на объект. В некотором смысле, обратная по алгоритму onSharedValueCreate.

*getImpl*

Общая часть кода для get и getOrSet. Выполняет поиск значения в кеше и размещения запроса на добавление при необходимости.

### Компараторы для контейнеров, хранящих метаданные.

```
struct RegionCompareBySize
{
  constexpr bool operator() (const RegionMetadata & a, const RegionMetadata & b) const noexcept { return a.size < b.size; }
  constexpr bool operator() (const RegionMetadata & a, size_t size) const noexcept { return a.size < size; }
  constexpr bool operator() (size_t size, const RegionMetadata & b) const noexcept { return size < b.size; }
};

struct RegionCompareByKey
{
  constexpr bool operator() (const RegionMetadata & a, const RegionMetadata & b) const noexcept {
```

```
return a.key() < b.key(); }
  constexpr bool operator() (const RegionMetadata & a, Key key) const noexcept { return a.key() <
key; }
  constexpr bool operator() (Key key, const RegionMetadata & b) const noexcept { return key <
b.key(); }
};
```

Интрузивные контейнеры для поиска позволяют передать произвольный функтор, который будет заниматься сравнением элементов. Для красно-черного дерева (free_regions, used_regions) нужны два компаратора (первый для определения региона минимального размера, который подойдет для запроса на аллокацию, второй -- для поиска значения по ключу в getImpl).

### Методы по аллокации и вытеснению памяти.

```
inline RegionMetadata * allocate(size_t size);
constexpr RegionMetadata * allocateFromFreeRegion(RegionMetadata & free_region, size_t size);
constexpr RegionMetadata * addNewChunk(size_t size);

RegionMetadata * evict(size_t requested) noexcept;

constexpr void freeAndCoalesce(RegionMetadata & region) noexcept;
};
```

*allocate*

Функция использует нижеуказанные для того, чтобы создать новый регион памяти размера не меньше size, который был бы доступен для инициализация.

*allocateFromFreeRegion*

Если в аллокаторе существует некий свободный регион с размером не меньше size, можно использовать его или его часть для аллокации. Функция пытается найти такой регион.

*addNewChunk*

Функция выделяет новую область памяти с помощью mmap в том случае, когда существующего размера недостаточно.

*evict*

Функция занимается вытеснением неиспользуемых (unreferenced) регионов так, чтобы в результате образовался регион размера не меньше size.

*freeAndCoalesce*

Функция пытается объединить регион с левым и правым соседом (в терминах односвязного списка). Это нужно для тех ситуаций, когда размера текущего региона недостаточно, но потенциально сумма его с соседями даст нужный size.

### Используемые алгоритмы

*getImpl (параллельная инициализация)*

Ключевые моменты алгоритма:

- Ищет регион, если регион найдет, вызывает onSharedValueCreate и возвращает его.
- Если регион не найден, под мьютексом создает новый запроса на добавление.
- Пытается захватить мьютекс региона. Возможно к этому моменту кто-то из других потоков уже проинициализирует значение.

### *onSharedValueCreate*

Функция вызывается при создании новой ссылки на существующий регион. Алгоритм работы:

- Захватить уникальный доступ к мьютексу метаданных региона (причины рассмотрены в RegionMetadata).
- Обновить количество ссылок (refcount). Если это не первая ссылка, закончить работу.
- Вернуть уникальный доступ к мьютексу метаданных.
- Захватить уникальный доступ к глобальному мьютексу.
- При необходимости стереть регион из unused_regions.
- Добавить регион в used_regions.
- Вернуть доступ к глобальному мьютексу.
- Атомарно обновить количество используемых ссылок в регионе MemoryChunk (самая строгая модель синхронизации).
- Атомарно обновить total_size_in_use.

Примечания:

- Мьютекс региона метаданных был выбран по той причине, что операция декремента и сравнения с константой не может быть сделана атомарной, а колчиество ссылок нужно было атомарно обновлять с доступом в дальнейшную секцию.
- В шаге (5) проверка на наличие региона в unused_regions не всегда производится. Это связано с тем, что одно из мест вызова функции -- getImpl, где регион найден в used_regions. Из инвариантов очевидно, что в таком случае находиться в unused_regions регион не может, поэтому проверка отключается шаблонным параметром.
- Модели синхронизации для шагов (8) и (9) были выбраны такими потому, что на этапе 8 функция уже не захватывает ни один мьютекс.

Если посмотреть на цепочку захватов-освобождений мьютексов, получим два изолированных вызова. Таким образом, эта функция не может привести к дедлоку или гонке (поскольку любое переупорядочивание не приведет к захвату более одного конкурирующего мьютекса).

### *onValueDelete*

Функция вызывается при разрушении объекта класса ValuePtr. Алгоритм работы:

- Захватить доступ к глобальному мьютексу.
- Выполнить поиск переданного аргумента в контейнере value_to_region.
- Если результат -- "не найдено", программа будет аварийно завершена по нарушению BOOST_ASSERT, так как это свидетельствует о нарушении инвариантов -- хотя бы одна ссылка на значение-аргумент есть, а в контейнере используемых регионов этого значения нет.
- Захватить доступ в мьютексу метаданных региона.
- Обновить количество ссылок (уменьшить на 1). Если результат -- не 0, выйти из функции
- Вернуть доступ к мьютексу метаданных региона.
- Стереть ключ-аргумент из value_to_region (поскольку текущий регион становится неиспользуемым).
- Добавить регион в unused_regions.
- Вернуть доступ к глобальному мьютексу.
- Атомарно обновить количество используемых ссылок в регионе MemoryChunk (самая строгая модель синхронизации).
- Захватить доступ к мьютексу used_regions.
- Добавить метаданные в used_regions.
- Вернуть доступ к мьютексу used_regions.

Примечания:

- В шаге (3) был выбран BOOST_ASSERT, а не исключение по той причине, что эта ситуация в идеальном исполнении программы является невозможной (а исключения могут отражать только реальные возможные ситуации, вродео ошибки выделения памяти или какой-либо другой). Следовательно, такое нарушение инвариантов свидетельствует об ошибке в самом интерфейсе. Кроме того, в режиме сборки Release, в отличие от исключений, BOOST_ASSERT убирается, не оказывая замедляющего влияния на код.
- Модель синхронизации для (10) была выбрала из-за отсутствия захвата глобального мьютекса.

Если посмотреть на цепочку захватов-освобождений мьютексов, получим цепь mutex -> metadata.mutex, что потенциально может привести к дедлоку при некорректной реализации других методов.

### *allocate*

- Функция занимается вызовом других для аллокации региона. Алгоритм работы:
- Захватить глобальный мьютекс.
- Округлить вверх размер до размера страницы.
- Поискать свободный регион по размеру. Если найден, вызвать allocateFromFreeRegion.
- Если общего размера аллоцированной памяти недостаточно, создать новый чанк и аллоцировать на нем.
- В противном случае, в цикле вытеснять регионы до тех пор, пока не будет найден нужный размер или не достигнут предел кеша.

### *allocateFromFreeRegion*

Функция создает новый регион, оккупирующий часть данного. Алгоритм работы:

- Если размер региона совпдает с целевым, вернуть регион, обновив его в контейнерах.
- Создать новый регион.
- Удалить изначальный регион из free_regions.
- Обновить новый размер изначального региона, и его указатель (смещается на размер).
- Добавить старый и новый регион в free_regions.
- Добавить новый регион в all_regions.

Функция требует у вызывающей стороны синхронизацию глобального мьютекса.

### *addNewChunk*

Функция добавляет новый чанк и регион, ссылающися на него. Алгоритм работы:

- Создать новый чанк и добавить его в chunks.
- Обновить размер
- Попытаться создать новый объект метаданных для региона. При исключении удалить чанк и прокинуть исключение наверх.
- Обновить указатели метаданных
- Добавить метаданные в соответствующие контейнеры.

Функция требует у вызывающей стороны синхронизацию глобального мьютекса.

### *evict*

Функция изымает (вытесняет, evicts) из кеша неиспользуемые регионы с тем условием, чтобы образовался свободный регион размера не меньше заданного. Алгоритм работы:

- Найти первый неиспользуемый регион. Если такого нет, вернуть nullptr.
- Войти в вечный цикл.

- Стереть регион из unused_regions.
- Под мьютексом стереть регион из used_regions.
- Вызывать на регионе freeAndCoaliesce.
- Если новый размер региона больше запрашиваемого, вернуть указатель.
- Инкрементировать итератор (взято новый регион из unused_regions).
- Если новый итератор указывает на регион другого чанка или на конец, вернуть указатель
- Иначе обновить secondary_evictions и продолжить.

Функция потенциально может привести к гонкам т.к дополнительно (кроме блокировок вызывающей стороны) на время захватывает used_regions_mutex.

### *freeAndCoalesce*

Функция добавляет регион в free_regions, возможно, сливая его с левым и/или правым соседом.
Алгоритм работы:

- Найти указатель на положение региона в all_regions.
- Определить специальную лямбду-disposer, которая будет, принимая на вход другой регион, удалять его, схлопывая с изначальным.
- Определить левого соседа изначального региона. Если он существует, слить с изначальным.
- Определить правого соседа изначальног региона. Если он существует, слить с изначальным.
- Добавить изначальный регион в free_regions.
- Функция модифицирует all_regions и free_regions без блокировки, но вызывается исключительно из другой функции. Вопросы синхронизации переадресовываются вызывающей стороне.

### *Подмена аллокатора*

Предполагается, что объекты класса TValue будут ссылаться внутри себя на область памяти, которая была выделена каким-то MemoryChunkом, но как этого добиться?
Из требований на TValue можно заметить, что у него должен быть шаблонный параметр, отвечающий за некоторый аллокатор. Обычно динамические контейнеры при получении запроса на добавление элемента вызывают метод allocate(size_t), возвращающий им указатель на регион памяти, и сохраняют его внутри.
Вспомним, что в функции getOrSet при вызове функтора initialize функтор вызывается с параметром void * heap_storage. Это и есть тот указатель на область памяти, в которой объекту класса TValue предполагается размещать свои данные.
Но как передать в наш аллокатор это указатель, если функция allocate обычно принимает только размер? Здесь есть два подхода:

- Полиморфный аллокатор. Этот вариант не рассматривается из-за своей скорости (на каждый вызов функции нужно выполнять развиртуализацию указателя на таблицу диспетчеризации). Заключается он в том, что в конкретный класс-наследник аллокатора в качестве параметра передается heap_storage при конструировании объекта, от которого позже будет инициализирован TValue.
- Результат возврата initialize, отличный от TValue. Ранее отмечалось, что тип объекта, возвращаемый функцией initialize, не обязан быть связан с TValue до тех пор, пока TValue от этого объекта можно сконструировать (конструктор копирования от const lvalue&). Это сделано намеренно. Предполагается, что функция вернет некоторый объект, который передаст TValue heap_storage и закончит свой жизненный цикл.

При второй реализации возникает другая проблема. Что делать при удалении объекта класса TValue? Очищать память не нужно, поскольку она будет автоматически очищена при вызове unmmap в деструкторе объекта MemoryChunk. Решение -- сделать в нашем аллокаторе метод free, который не будет делать ничего.

### Используемые Linux API

В целом, у процессов, которые хотят получить от операционной системы дополнительную память, не так много вариантов -- в стандарте POSIX прописаны две возможные опции:

- Системные вызовы brk и sbrk, отвечающие за смещение границы сегмента данных.
- Системный вызов mmap, отвечающий за выделение региона заданного размера по произвольному адресу.

brk не подходит для больших размеров, поэтому мы его и не рассматриваем. Из интересного для mmap можно рассмотреть флаг MMAP_POPULATE. Этот флаг доступен на новых версиях ядра Linux и позволяет выполнить предварительное чтение страниц памяти.
Обычный алгоритм следующий: 1. Ядро выделяет процессу указатель на несуществующие страницы памяти. 2. При первой попытке чтения или записи возникает ошибка отсутствия страницы (page fault error), происходит выделение новой страницы памяти и доставка ее процессу. Если же сначала прочитать все страницы памяти, то при последующих обращения ошибки отсутствия страницы не будет, таким образом, мы переносим возможные задержки со всего времени выполнения на время аллокации региона.

### Используемые структуры данных

#### *Собственные*

MemoryChunk
Самый низкоуровневый примитив, с которым нужно иметь дело -- это класс-обертка над некоторым непрерывным участком виртуальных адресов. Поскольку наш аллокатор оперирует mmap как средством выделения памяти, разумно реализовать этот класс сообразно std::span. Заголовочный файл приведен ниже:

```
struct MemoryChunk : private boost::noncopyable
{
    void * ptr;
    size_t size;

    size_t used_refcount;

    constexpr MemoryChunk(size_t size_, void * address_hint);
    constexpr MemoryChunk(MemoryChunk && other) noexcept;

    ~MemoryChunk();
}
```

Члены ptr и size отвечают за хранение начального адреса участка памяти и его длину соответственно.
Член used_refcount хранит количество экземпляров класса RegionMetadata, ссылающихся на данный объект. Необходимость его наличия была обоснована в разделе [Публичный интерфейс/shrinkToFit].
Класс реализует идиому RAII (Resource acquisition is initialization, обращение к ресурсу и есть его инициализация), выделяя память с помощью mmap в конструкторе и освобождая ее в деструкторе. Сообразно принятой в ClickHouse практике написания кода, при возникновении ошибки при выделении памяти (в конструкторе -- ненулевой код возврата функции mmap) или ее освобождении (в деструкторе -- ненулевой код возврата функции munmap) создается объект класса DB::Exception с кодом ошибки DB::ErrorCodes::CANNOT_ALLOCATE_MEMORY или DB::ErrorCodes::CANNOT_MUNMAP соответственно.
Операции выделения и освобождения памяти отслеживаются классом CurrentMemoryTracker.
Параметр address_hint в конструкторе отвечает за передачу функции mmap подсказки для адреса,

генерируемого функцией ASLR. Подробно об этом было рассказано в разделе [Шаблонные параметры/ASLR].
Предполагается, что адрес, подаваемый в конструкторе, выровнен по ValueAlignment байт.

### RegionMetadata

Класс, относящийся к RegionMetadata как многие-к-одному (many-to-one). Это означает, что несколько объектов класса RegionMetadata могут ссылаться на один MemoryChunk. Класс отвечает за хранение пар TKey - TValue во всех интрузивных контейнерах. Его заголовочный файл приведен ниже.

```
struct RegionMetadata :
    public TUnusedRegionHook,
    public TAllRegionsHook,
    public TFreeRegionHook,
    public TUsedRegionHook
{
    std::aligned_storage_t<sizeof(Key), alignof(Key)> key_storage;
    std::aligned_storage_t<sizeof(Value), alignof(Value)> value_storage;

    std::mutex mutex;

    bool key_initialized{false};
    bool value_initialized{false};

    union {
        void * ptr {nullptr};
        char * char_ptr;
    };

    size_t size {0};
    size_t refcount {0};

    MemoryChunk * chunk {nullptr};

    [[nodiscard]] static RegionMetadata * create();

    constexpr void destroy() noexcept;

    constexpr void init_key(const Key& key);

    template <class Init>
    constexpr void init_value(Init&& init_func);

    constexpr const Key& key() const noexcept;

    constexpr Value * value() noexcept;

    [[nodiscard, gnu::pure]] constexpr bool operator< (const RegionMetadata & other) const noexcept;
    [[nodiscard, gnu::pure]] constexpr bool isFree() const noexcept;

private:
```

```
    RegionMetadata() {}
    ~RegionMetadata() = default;
};
```

### Конструктор и деструктор

Заметим, что и конструктор, и деструктор класса сделаны приватными. Предполагается, что класс будет сконструирован с помощью вызова "фабричного" метода create, а удален вызовом destroy, следовательно, в обычных конструкторах и деструкторах нет надобности (фабричное конструирование -- необходимость для объекта, который помещается в интрузивные контейнеры). Родительские классы для данного рассмотрены в разделе [Используемые структуры данных/Внешние/Интрузивные контейнеры].

### key_storage, value_storage.

Главная проблема для классов TKey и TValue -- отсутствие необходимости быть конструируемыми без аргументов. Можно было бы это потребовать, но тогда потенциальный спектр возможных типов был бы сильно ограничен (например, у DB::PODArray конструктора по умолчанию нет в силу интерфейса). Использовать идиому отложенной инициализации (deferred initialization) с помощью std::unique_ptr также не представляется возможным, поскольку объекты этих двух классов надо хранить именно тут, а не в произвольном месте сегмента данных.
Решение -- выделять хранилище достаточного размера (с учетом выравнивания) и конструировать объекты по адресу этих хранилищ. Для определения того, инициализирован ли ключ или значение, есть члены key_initialized и value_initialized. Использовать std::optional мне также показалось нецелесообразным из-за излишней громоздкости интерфейса.

### mutex, refcount

Защищает член refcount (количество активных ссылок на значение, соответствующее ключу key). При удалении значения предполагается, что только один из потоков может войти в критическую область. Следовательно, нужно озаботиться синхронизацией. Поскольку выполняемая операция (декремент и сравнение с нулем) не может быть сделана атомарной, был использован дополнительный мьютекс.

### ptr, char_ptr, size

Ссылается на какую-то область MemoryChunka размера size, где объект value будет размещать свои данные. Проблема в том, что указатель может указывать не на начало MemoryChunk, поэтому одного указателя void *недостаточно (в системе типов C этот указатель соответствует указателю на любой тип, следовательно, указательная арифметика для него неприменима). Для смещения указателя (в методе allocateFromNewRegion) и нужен char_ptr.

### create, destroy

Первая функция выполняет выделение в сегменте данных объекта данного класса. Вторая корректно разрушает (путем вызова соответствующих деструкторов) key и value, если они были проинициализированы, и удаляет сам объект.

### init_key, init_value

Функции занимаются инициализацией ключа и значения соответственно. Для инициализации используется метод стандатной библиотеки std::construct_at, которому передается указатель на начало хранилища, проинтерпретированный как указатель на объект класса ключа и значения соответственно. Для подавления оптимизации работы с константными указателями (в связи с изменением в C++17 работы с памятью) используется специально предназначенный для этого std::launder. Необходимость шаблонного параметра в init_value обусловлена тем, что туда из getOrSet передается функтор initialize.

*key, value*

Получают указатель и ссылку на значение и ключ соответственно. Работают аналогично предыдущей функции (реинтерпретация указателя на хранилище и std::launder). Замечу, что вызов этих функций на неинициализированных хранилищах является неопределенным поведением (undefined behaviour). Таким образом, удостовериться, что хранилища проинициализированы -- задача вызывающей стороны.

### ValuePtr

Сам по себе ValuePtr -- всего лишь умный разделяемый указатель, параметризованный TValue, интереса не представляет. Стоит лишь заметить, что, как и другие умные указатели, он предоставляет возможность указать свой объект-Deleter в конструкторе, о нем мы и поговорим. При уничтожении какого-то ValuePtr нужно корректно обновить глобальное состояние аллокатора -- уменьшить refcount в метаданных, возможно, переместить объект из одного контейнера в другой. Это делает функция onValueDelete, но как сделать из нее объект-Deleter, и что это такое?
В деструкторе std::shared_ptr вызывается некоторый объект, ответственный за удаление содержимого, по умолчанию это std::default_delete, который обычно выполняет delete ptr на хранимом указателе. Но есть возможность передать в конструктор класса свой объект, который будет использован при удалении. Сам объект копируется, место под него выделяется некоторым фабричным методам, но не будем углубляться в стандартную библиотеку.
На помощь приходит std::bind_front, выполняющий каррирование функции (превращение n-арной функции в n-1-арную, то есть, принимающую на один (первый) аргумент меньше). Любую функцию класса (в частности, onValueDelete) можно представить как свободную, принимающую первым дополнительным аргументом указатель на объект этого класса.
К сожалению, в libc++ (библиотека, поставляемая с компилятором llvm clang++) эта функция пока недоступна, поэтому вопользуемся ее предшественником -- фунцией std::bind. Механика ее работы такова -- она конструирует специальный объект-байндер, которому в конструктор передаются нужные нам аргументы (указатель this, в данном случае). У полученного объекта определен operator() на оставшихся аргументах. Этот байндер и будет передан умному указателю в качестве параметра Deleter.

### InsertionAttempt

Представляет запрос на добавление значения для некоторого ключа в кеш. Позволяет конкурентно (параллельно) инициализировать значения для ключей из нескольких потоков, обеспечивая, таким образом, лучший результат в соотношении hits/misses (количетсво случаев, когда запись была найдена в кеше, по отношению к количеству случаев, когда запись в кеше не была найдена). Заголовочный файл приведен ниже.

```
struct InsertionAttempt
{
   constexpr InsertionAttempt(IGrabberAllocator & alloc_) noexcept;

   IGrabberAllocator & alloc;

   std::mutex mutex;
   bool is_disposed{false};

   size_t refcount {0};

   ValuePtr value;
};
```

6. mutex защищает остальные параметры (кроме alloc),

7. is_disposed отвечает за то, есть ли в этом запросе значение,
8. value хранит предполагаемое значение,
9. refcount отвечает за количество объектов класса InsertionAttemptDiposer, которые хотят данный объект удалить.

### InsertionAttemptDisposer

Отвечает за удаление какого-то объекта класса InsertionAttempt. Удаление, так же как и инициализация значения в кеше, может производиться конкурентно. В случае одновременного доступа несколькими потоками к этому объекту за удаление отвечает первый. Если же во время удаления произошло какое-то исключение, то ответственнен второй (и так далее, вплоть до последнего). Порядок доступа определяется порядком доступа к мьютексу объекта хранимого InsertionAttempt. Заголовочный файл приведен ниже.

```
struct InsertionAttemptDisposer
{
   constexpr InsertionAttemptDisposer() noexcept = default;

   const Key * key {nullptr};
   bool attempt_disposed{false};

   std::shared_ptr<InsertionAttempt> attempt;

   inline void acquire(const Key * key_, const std::shared_ptr<InsertionAttempt> & attempt_) noexcept;

   inline void dispose() noexcept;

   ~InsertionAttemptDisposer() noexcept;
};
```

10. acquire захватывает ответственность за удаление какого-то InsertionAttempt, увеличивая количество ссылок на него>
11. dispose (вызванный явно) удаляет attempt из insertion_attempts и обновляет его состояние.
12. Деструктор при наличии неудаленного attempt обновляет его ссылочность, при необходимости (при нулевой ссылке) вызывая dispose.

### Внутренние

Под внутренними контейнерами я подразумеваю те, которые определены в ClickHouse (но не в библиотеках, которые ClickHouse может использовать). В процессе работы я использовал два из них (в основном), рассмотрим их.

#### *PODArray*

Класс представляет динамический контейнер какого-то типа данных. Оптимизирован под IBO (initial buffer optimization), когда при инициализации контейнера часть данных может храниться в предварительно выделенном на стеке (как часть объекта) хранилище, таким образом, не аллоцируя память в сегменте данных. Контейнер позволяет параметризовать себя аллокатором, и, что наиболее ценно, передавать туда некоторые параметры. Этим я возпользуюсь при реализации аллокатора в разделе (Интеграция/Кеш засечек/fakepodalloc).
В остальном, контейнер реализован сообразно другим динамическим.
Memory
Еще один динамический контейнер, также позволяет передать параметры аллокатору, отличается более простым интерфейсом и тем, что не пред-инициализирует память нулями с помощью memset в конструкторе (в отличие от PODArray).

*Внешние*

Интрузивные контейнеры

Основное быстродействие аллокатора достигается именно за счет интрузивных контейнеров, о которых я сейчас поговорю.

В общем случае, интрузивный контейнер хранит не сам объект, а указатель на него, обеспечивая возможность (в комбинации с другими контейнерами) эффективно обращаться с данными (например, комбинация set + list позволяет эффективно и быстро искать как сам элемент, так и его соседей).

В связи с этим, возникает несколько вопросов:

- Как удалять и создавать объект? Предполагается, что пользователь заранее определит методы так называемого "фабричного" создания и удаления (объекта из памяти при удалении из контейнера).
- В каком порядке добавлять и удалять элементы из контейнеров? Порядок добавления не имеет значения, порядок удаления -- тоже. Стоит лишь учитывать, что при удалении элемента из последнего контейнера нужно также удалить его из памяти, используя специальную функцию-deleter (для избежания утечек памяти).

В своей работе я использовал библиотеку boost::intrusive, рассмотрим конкретные особенности реализации.

- Для начала, нужно определиться, какой объект мы будем хранить в контейнерах. Наиболее логичным претендентом кажется RegionMetadata, поскольку он хранит всю необходимую информацию.
- Затем нужно определить специальные структуры-теги. Поскольку boost::intrusive позволяет добавить объект в несколько контейнеров одного типа (например, в два boost::intrusive::list -- односвязный список), нужно уметь отличать состояние объекта в каждом из этих контейнеров. Создадим 4 тега соответственно:

```
struct UnusedRegionTag;
struct RegionTag;
struct FreeRegionTag;
struct UsedRegionTag;
```

Структуры используются лишь для определения конкретного контейнера, поэтому достаточно сделать их неполными (incomplete type).

- Затем нужно определить специальные хуки. Хук -- это некоторая структура, хранящая информацию о связи контейнера и элемента, хранится обычно в элементе. В этом еще одно отличие интрузивных контейнеров от обычных -- элементы обычных контейнеров ничего не знают о самих контейнерах.

  В библиотеке есть много параметров для хуков, нас интересуют только два.
  - `constant_time_size`. Поскольку предполагается, что определение размера контейнера будет производиться часто,
    разумно сдлеать его константным. Достигается это в реализации хранением в самом хуке размера контейнера, куда он
    помещен. Дополнительный `std::size_t` в каждом `RegionMetadata` кажется небольшой платой за быстродействие,
    поэтому передадим этот шаблонный параметр в определение хука.
  - `link_mode`. Этот параметр отвечает за то, что будет происходить с элементом при его удалении из контейнера.
    1. `normal_link`. Состояние элемента при удалении из контейнера не определено.
    2. `safe_link`. Используется по умолчанию. Предоставляет следующие инварианты:
    1. Конструктор хука устаналивает некоторое известное состояние
    2. При добавлении элемента в контейнер проверяется это состояние и обновляется так,

чтобы элемент "знал",
что он в контейнере.

3. При удалении элемента из контейнера, аналогично, устанавливается некоторое состояние.

4. Деструктор тоже проверяет это состояние.

Как следствие, на объекте можно за константное время получить информацию, есть ли он в каком-то контейнере
(с помощью функции `is_linked()`)

- `auto_unlink`. Позволяет удалить элемент из контейнера, не обращаясь к самому контейнеру, удаляет элемент из
всех контейнеров при вызове деструктора элемента. Для нас неприемлемо, так как не позволяет определять
размер контейнера за константу.

Таким образом, хуки для всех контейнеров будут выглядеть следующим образом:
```cpp
using TUnusedRegionHook  =
boost::intrusive::list_base_hook<boost::intrusive::tag<UnusedRegionTag>>;
using TAllRegionsHook    =
boost::intrusive::list_base_hook<boost::intrusive::tag<RegionTag>>;
using TFreeRegionHook    = boost::intrusive::set_base_hook<
boost::intrusive::tag<FreeRegionTag>>;
using TUsedRegionHook    =
boost::intrusive::set_base_hook<boost::intrusive::tag<UsedRegionTag>>;
```

4. Последний шаг -- определение типов самих контейнеров.
    1. Нам нужно уметь быстро искать ключи, поэтому для used_regions возьмем boost::intrusive::set, представляющий красно-черное дерево ключей.
    2. Для поиска региона, подходяшего по размеру (нужно в evict), для free_regions возьмем boost::intrusive_multiset, представляющий красно-черное дерево списка регионов, упорядоченных по размеру.
    3. Для поиска соседей региона, с которыми его можно будет слить, нужен односвязный список, возьмем boost::intrusive::list.
    4. Для хранения списка всех регионов есть только одно ограничение -- на быструю вставку и удаление, также возьмем list.

Результирующие типы контейнеров приведены ниже:
```
using TUnusedRegionsList = boost::intrusive::list<RegionMetadata,
   boost::intrusive::base_hook<TUnusedRegionHook>,
   boost::intrusive::constant_time_size<true>>;
using TRegionsList = boost::intrusive::list<RegionMetadata,
   boost::intrusive::base_hook<TAllRegionsHook>,
   boost::intrusive::constant_time_size<true>>;
using TFreeRegionsMap = boost::intrusive::multiset<RegionMetadata,
   boost::intrusive::compare<RegionCompareBySize>,
   boost::intrusive::base_hook<TFreeRegionHook>,
   boost::intrusive::constant_time_size<true>>;
using TUsedRegionsMap = boost::intrusive::set<RegionMetadata,
   boost::intrusive::compare<RegionCompareByKey>,
```

```
    boost::intrusive::base_hook<TUsedRegionHook>,
    boost::intrusive::constant_time_size<true>>;
```

### libc++

Стоит заметить, что ClickHouse компонуется с библиотекой libc++ (входит в пакет clang), а не libstdc++ (входит в пакет gcc), впрочем, разницы для пользователя нет.

std::unordered_map

Хеш-таблица с адресацией методом цепочек. Используется в value_to_region для быстрого определения региона, соответствующего какому-то значению, и в insertion_attempts для определения запроса на добавление, соответствующего региону.

std::list

Односвязный список. Используется в chunks (потому что необходимо быстрое добавление и удаление).

## Процесс разработки

### *Почему убрали концепты*

Изначально, предполагалось, что шаблонные параметры аллокатора будут использовать концепты для упрощения разработки. Это также позволило бы упростить код -- для сравнения, привожу шаблонный интерфейс аллокатора и некоторые из определенных концептов.

```
template <class Hash, class Value>
concept GAKeyHash = requires(const Value& v) {
    { Hash().operator()(v) } -> std::convertible_to<std::size_t>;
};

template <class F, class Value>
concept GASizeFunction = std::is_same<F, ga::runtime> || requires(const Value& v) {
    {F().operator()(v)} -> std::convertible_to<std::size_t>;
};

template <class F, class Value>
concept GAInitFunction = std::is_same<F, ga::runtime> || requires(void * address_hint) {
    F(address_hint, Value*);
};

template <class T> concept GAAslrFunction = requires(const pcg64& rng) {
    {T().operator()(rng)} -> std::is_same<void *>;
};

template <
 GAKey TKey,
 GAValue TValue,

 GAKeyHash KeyHash = std::hash<Key>,

 GASizeFunction SizeFunction = ga::runtime,
 GAInitFunction InitFunction = ga::runtime,
 GAAslrFunction ASLRFunction = ga::DefaultASLR,
```

К сожалению, от этой идеи пришлось отказаться из-за ошибки в компиляторе g++ https://gcc.gnu.org/bugzilla/show_bug.cgi?id=91867

При компиляции одной из зависимых библиотек re2 с флагом -fconcepts компилятор выдавал

ошибку парсера internal compiler error in type_unification_real на коде
```
for (bool e : {true, false})
```

### Изменение интерфейса из-за дедлока

Изначально, в интерфейсе предполагалось наличие только одного мьютекса (не во внутренних структурах) -- глобального. Он бы защищал все контейнеры разом и обеспечивал бы достаточную синхронизацию. Эта идея оказалась неудачно по причине медлительности, но самое главное -- по причине дедлока, который я сейчас опишу.
Предположим, что у нас есть два потока, пытающихся инициализировать один и тот же ключ.
1. Поток 1 вызывает getOrSet.
2. Внутри вызывается getImpl.
3. Поток выходит из getImpl, не найдя значения.
4. Планировщик передает управление потоку 2.
5. Поток 2 вызывает getImpl в getOrSet.
6. Планировщик передает управление потоку 1.
7. Поток 1 пытается захватить мьютекс для функции allocate (но не может).
8. По истечении кванта времени плнировщик передает управление потоку 2.
9. Поток 2 не может продолжить работу (вызвав функцию onSharedValue), потому что желаемый мьютекс занят.
10. Дедлок
Ситуация масштабируется на любое число потоков.

## Глава 4: Интеграция аллокатора в ClickHouse

### Кеш засечек

#### *Определение*

Кеш засечек (Cache of marks, mark cache) локален для каждого файла, хранит в себе массив засечек. Засечка -- это пара size_t "смещение в сжатом файле" и "смещение в несжатом файле". При загрузке сжатого файла кеширование засечек помогает определить (при запросе данных), откуда брать данные для разжатия.

#### *Связанные структуры данных*

- MarkInCompressedFile -- Структура, непосредственно идентифицирующая засечку (пара size_t, отвечающих за смещение от начала файла для сжатой и несжатой версии соответственно).
- MarksInCompressedFile : PODArray<MarkInCompressedFile> -- массив засечек для файла с каким-то именем. Не параметризуются аллокатором (т.е. место под экземпляры этого класса выделяется где-то в куче).
- CacheMarksInCompressedFile : PODArray<MarkInCompressedFile, FakePODAllocatorForIG -- массив засечек для файла, специализированный для использования с IGrabberAllocator. Параметризуется специальным аллокатором, см. ниже.
- MergeTreeMarksLoader -- структура, отвечающая за загрузку засечек для определенного файла и их хранение.

#### *Изменение интерфейсов для реализации задачи*

В интерфейсе изначально предполагалось несколько настроек, нас интересуют две из них: - MarkCache * mark_cache, передаваемый экземпляру MergeTreeMarksLoader. Если указатель нулевой, предполагается, что кеш выключен. - bool save_new_marks_in_cache, отвечающий за настройку вымывания кеша. Если он установлен в false, кеш может использоваться, но только в режиме чтения (сделано для того, чтобы не вымывать кеш новыми записями).
Для реализации была добавлена структура CacheMarksInCompressedFile, отвечающая за хранение MarkInCompressedFile в аллокаторе.

Поскольку структуры MarksInCompressedFile и CacheMarksInCompressedFile несовместимы, в интерфейс MergeTreeMarksLoader добавляется std::unique_ptr<MarksInCompressedFile> marks_non_cache.
Если кеш выключен, то создается новый экземпляр MarksIncompressedFile, хранящийся в marks_non_cache, все дальнейшие обращения происходят к нему.
Если кеш выключен, возможны два варианта:

- Значение по данному ключу (вычисляется как хеш от имени файла и пути к нему) не было найдено в кеше, и save_new_marks_in_cache = false. Алгоритм такой же, как и с выключенным кешом.
- Значение по данному ключу было найдено в кеше, или сохранение новых засечек включено. Тогда результат IGrabberAllocator::getOrSet() записывается в MarkCache::ValuePtr marks_cache, все дальнейшие обращения идут к нему.

### *Изменение интерфейса PODArray*

В PODArray были изменены сигнатуры функций alloc_for_num_elements и конструкторов копирования так, чтобы у пользователя появилась возможность передавать туда параметры аллокатора через вариативный шаблон TAllocatorParams&& ...params.
Для того, чтобы в случае единичного параметра аллокатора эффективно отличать конструкторы PODArray::PODArray(size_t n, const T& x) и PODArray::PODArray(size_t n, TAllocParam param) при совпадении типов T и TAllocParam был добавлен фиктивный элемент struct alloc_tag_t alloc_tag.
Таким образом, конструктор, принимающий на вход параметры аллокатора, стал PODArray::PODArray(size_t n, alloc_tag_t, TAllocatorParams&& ..params).
Был добавлен конструктор копирования от PODArray, параметризованного другим аллокатором (необходимо на момент инициализации PODArray<FakePodAllocatorForIG> от PODArray<Allocator<false>>).

### *FakePODAllocatorForIG*

Как и другие контейнеры, PODArray позволяет параметризовать себя аллокатором (по умолчанию используется Allocator<false>).
alloc()
Рассмотрим сначала последовательность действий при вызове IGrabberAllocator::getOrSet для Value : PODArray, подразумевая, что PODArray параметризован аллокатором класса TAlloc:

- Предположим, что значение для данного **Key key** не было найдено в кеше.
- Предположим также, что при попытке захвата мьютекса ни один из потоков, использующих этот аллокатор, кроме текущего, не добавил значение для данного **key** в кеш.
- Вдобавок, места в аллокаторе достаточно для размещения текущего элемента. Пропустим все этапы до вызова init_func.
- Вызывается init_func(void * heap_storage_start). Напомним, что эта функция выполняет два действия:
- Создание некоторого временного объекта **T**, от которого как от const T& t (мотивация приведена в разделе Публичный интерфейс:init_func) инициализируется **Value value**, хранящееся в RegionMetadata.
- Добавление в **T** необходимой информации (обычно достаточно heap_storage_start) так, чтобы объект Value "понимал", где ему размещать данные.
- Функция возвращает некоторый временный объект (предположим, что он кроме состояния хранит сам void *  heap_storage_start).

Дальнейший алгоритм специфичен для PODArray, но легко адаптируется для других типов данных:

- Вызывается конструктор Value::Value(size_t, TAllocatorParams...), в нашем случае, Value(size_t, void *).

- Вызывается конструктор родительского класса PODArray::PODArray(size_t, TAllocatorParams... )
- Вызывается функция аллокации PODArray::alloc_for_num_bytes(size_t, TAllocatorParams ... ). аллокатора).
- Внутри последней функции происходит присваивание c_start = TAlloc::alloc(size_t, TAllocatorParams...).

По алгоритму, описанному выше, функция TAlloc::alloc должна вернуть указатель на место, выделенное под хранение. Это в точности void * heap_storage_start, записанный во временное значение.
free()
При уничтожении объекта класса PODArray в деструкторе происходит вызов функции free(), которая в свою очередь вызывает TAlloc::free(char * storage, size_t size). Предполагается, что память, выделенная IGrabberAllocator::MemoryChunk-ом, будет очищена с помощью munmap в деструкторе, в нашем аллокаторе не требуется ничего делать.
realloc()
При вызове конструктора копирования PODArray::PODArray(const PODArray<OtherAlloc>, alloc_tag_t, TAllocatorParams...) по цепочке вызываются следующие функции:

- PODArray::insert(It, It, TAllocatorParams ...)
- PODArray::insertPrepare(It, It, TAllocatorParams...)
- PODArray::reserve(size_t, TAllocatorParams...)
- PODArray::realloc(size_t, TAllocatorParams...)
- TAlloc::realloc(char * size_t, size_t, TAllocatorParams...)

Поскольку на момент вызова последней функции место под хранение уже выделено, аналогично пункту alloc(), достаточно просто возвратить пришедший в качестве настроек аллокатора void * heap_address_storage.
stack_threshold().
PODArray позволяет использовать стековые аллокаторы, поэтому для интерфейса требуется реализация функции, позволяющей определить, выделено ли место на стеке или в куче. В соответствии с требованиями всегда возвращаем из нее 0 (выделено не на стеке).
Выводы
Три вышеуказанные функции (с некоторыми изменениями) реализованы в классе FakePODAllocatorForIG, которым параметризуется CacheMarksInCompressedFile.

### Кеш несжатых данных

#### *Определение*

Кеш несжатых данных -- это кеш массивов данных, полученных в результате разжатия данных, полученных из какого-то хранилища.

#### *Связанные структуры данных*

- CachedCompressedReadBuffer -- Класс, занимающийся загрузкой несжатых данных. Хранит указатель на элемент, полученный из кеша. Локален для файла.
- UncompressedCache -- специализация IGrabberAllocator для кеша несжатых данных.
- UncompressedCell -- временная структура, передающая элементу кеша heap_storage.
- CacheUncompressedCell -- Элемент кеша, UncompressedCell с подмененным аллокатором у члена data.

#### *Изменение интерфейсов для реализации задачи*

Нас интересует только метод CachedCompressedReadBuffer::nextImpl, загружающий блок несжатых данных для этого файла. Алгоритм после изменения таков:

13. Попытаться получить элемент из кеша с помощью функции get. Если элемент найден, перейти к шагу 7
14. Инициализировать файл и установить его позицию для вычитывания блока.
15. Аллоцировать на стеке UncompressedCell cell.
16. Загрузить в cell информацию о размере сжатого и несжатого блока данных.
17. Определить финальный размер элемента кеша с учетом того, что он может быть нулевым.
18. Проинициализировать значение следующим образом:
    1. В лямбда-функторе initialize аллоцировать еще одну ячейку UncompressedCell other.
    2. Передать ей heap_storage.
    3. Если размер cell был больше нуля, выполнить загрузку и декомпрессию данных в other с помощью функции decompress
    4. Вернуть временную структуру из функтора, от который будет инициализировано значение CacheUncompressedCell.
19. Создать буфер с разжатыми данными.

### *Изменения интерфейса Memory*

В класс Memory было внесено три изменения:

20. Добавлен конструктор копирования const lvalue& от другого объекта Memory, возможно, с другим аллокатором, принимающий также вариативный шаблон параметров для аллокатора.
21. Методы resize() и alloc() были изменены так, чтобы при необходимости им также можно было передать параметры аллокатора.

### *FakeMemoryAllocatorForIG*

В отличие от FakePodAllocatorForIG, у текущего аллокатора предполагается несколько иной интерфейс, рассмотрим его.

22. Вызывается конструктор Memory::Memory(const Memory<OtherAlloc>&, TAllocatorParam...).
23. Вызывается функция alloc, куда передается размер и наш параметр heap_storage.
24. Функция должна просто возвратить переданный ей параметр.

Метод free, как и в предыдущем реализованном аллокаторе, не делает ничего.
Иногда может вызываться метод realloc -- предполагается, что он будет выделять место где-то в другой области памяти и копировать данные туда, возвращая указатель на новое начало данных. По постановке задачи предполагается, что реаллокаций в процесе хранения произойти не может (это undefined behaviour), поэтому просто возвращаем переданный параметр аллокатора (тот самый heap_storage).

## Заключение

В процессе разработки (240 часов чистого времени) было выявлено очень много различных интересных ошибок. Сам опыт тоже оказался чрезвычайно полезным. Надеюсь, что мой интерфейс пригодится в ClickHouse.