

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б10-мм

Реализация хранилища чанков для файловой системы ChunkFS на основе B+ дерева

Закарлюка Иван Владимирович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Гориховский В. И.

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	4
2. Описание решения	5
2.1. Последовательная версия	5
2.2. Асинхронная версия	5
2.3. Интеграция в ChunkFS	8
3. Тестирование	10
3.1. Функциональное тестирование	10
3.2. Производительность в ChunkFS	10
3.3. Производительность хранилища	10
3.4. Масштабируемость	11
3.5. Тестовая среда	12
Заключение	13
Список литературы	14

Введение

Для систематического и стандартизированного сравнения реализаций алгоритмов дедупликации на языке Rust была разработана модельная универсальная файловая система ChunkFS[6]. В основе работы алгоритмов дедупликации после разбиения исходных данных на чанки лежит необходимость их сравнения по хешу и сохранения только уникальных экземпляров. Естественным образом возникает потребность в структуре данных, позволяющей эффективно добавлять, искать и извлекать чанки по их уникальному хешу. Такой структурой является key-value хранилище — абстрактный тип данных, предназначенный для хранения значений и обеспечения быстрого доступа к ним по уникальным ключам.

Существующие готовые решения key-value хранилищ не могут быть напрямую интегрированы в ChunkFS без возникновения существенных накладных расходов. Кроме того, специализированная реализация хранилища, ориентированная на работу с чанками, позволяет применить специфические оптимизации, недоступные в универсальных системах.

В предыдущей работе[5] был проведен обзор существующих решений, обоснован выбор B+ дерева в качестве основы для key-value хранилища и реализован его прототип на языке Rust. Конечной целью является создание надежного асинхронного key-value хранилища, обеспечивающего гарантии ACID и оптимизированного для работы с чанками.

В рамках данной работы были реализованы, протестированы и интегрированы в ChunkFS надежные последовательная и асинхронная версии key-value хранилища.

1. Постановка задачи

Целью работы является создание надежного асинхронного хранилища чанков на основе B+ дерева на языке Rust и его последующая интеграция в ChunkFS. Для достижения этой цели были поставлены следующие задачи:

1. Разработать стабильную последовательную версию key-value хранилища на основе B+ дерева, учитывая опыт реализации прототипа; провести её тестирование и интегрировать в ChunkFS
2. Разработать стабильную асинхронную версию key-value хранилища; провести её тестирование и интегрировать в ChunkFS
3. Провести сравнительный анализ производительности последовательной и асинхронной версий хранилища; выполнить тесты производительности обеих версий хранилища; провести бенчмарки ChunkFS с интегрированными хранилищами

2. Описание решения

2.1. Последовательная версия

В ходе предыдущей работы был реализован прототип B+ дерева. В данной работе, используя полученный опыт, была разработана надежная последовательная версия хранилища на основе B+ дерева.

Хранилище использует структуры `ChunkHandler` для указателей на данные, содержащие:

- Путь к файлу с чанком
- Смещение чанка в файле
- Размер чанка

Структура `ChunkHandler` предоставляет метод чтения, загружающий соответствующий чанк из файла.

Процесс вставки чанка:

1. Создание экземпляра `ChunkHandler`
2. Запись чанка в файл
3. Вставка пары `<хэш чанка, ChunkHandler>` в дерево
4. При достижении файлом предельного размера (задается параметром) система создает новый файл для последующих чанков

Интерфейс `Database`, предоставляемый `ChunkFS`, был реализован для последовательной версии хранилища.

2.2. Асинхронная версия

Для асинхронной реализации функции `get` и `insert` были переписаны в итеративной форме. Для решения проблемы конкурентного доступа рассматривались подходы:

- Lock-free алгоритмы [3, 1]

- Грубые блокировки (coarse-grained locking)
- Тонкие блокировки (fine-grained locking), в частности latch crabbing[2] (lock-coupling)

Выбор latch crabbing обоснован следующими причинами:

- Эффективные lock-free алгоритмы сложны в реализации
- Алгоритмы основанные на грубых блокировках всегда блокируют все дерево на любую операцию. Из-за этого этот метод крайне прост в реализации и создает меньше накладных расходов, ведь нужно захватывать и освобождать гоораздо меньше блокировок, однако уже при небольшом количество вставок дерево начинает работать практически синхронно
- Latch crabbing(сравнительно с эффективными lock-free алгоритмами) прост в реализации, но все ещё является очень эффективным

Для обеспечения конкурентного доступа к B+ дереву в многопоточной среде был реализован алгоритм latch crabbing (также известный как lock coupling) [2]. Ниже представлено описание алгоритма.

Описание алгоритма latch crabbing

Введем несколько обозначений, которые используем при описании алгоритма.

- Блокировка на запись — эксклюзивная блокировка, гарантирует, что параллельно с ней не удерживается ни одна блокировка на этот же ресурс.
- Блокировка на чтение — блокировка, которая гарантирует только то, что параллельно не захвачена блокировка на запись для этого же ресурса.
- Безопасный узел — узел B+ дерева, в котором находится менее 2^* (степень ветвления дерева - 1) узлов. В этот узел можно вставить ещё один ключ, не разделяя сам узел на два.

Рассмотрим операции поиска (*get*) и вставки (*insert*) отдельно.

- Операция чтения: Обозначим текущий узел как *Current*.
 1. Захват блокировки на чтение корневого узла, $Current = \text{корневой узел}$
 2. Пока *Current* не является листом:
 - (a) Определить дочерний узел, соответствующий искомому ключу
 - (b) Если такого узла нет, значит ключа нет в дереве, чтение заканчивается
 - (c) Захват блокировки на чтение для этого дочернего узла. Обозначим его за *Next*
 - (d) Освобождается блокировка *Current*
 - (e) $Current = Next$
 3. Поиск ключа в листе *Current*
 4. Освобождение блокировки *Current*
- Операция записи:
 1. Захват блокировки на запись корневого узла, $Current = \text{корневой узел}$
 2. Создание стека узлов с захваченной блокировкой, обозначим его за *guards*. В *guards* добавляется *Current*
 3. Пока *Current* не является листом:
 - (a) Определить дочерний узел, соответствующий ключу
 - (b) Захватить блокировку на запись для этого дочернего узла, обозначим узел как *Next*
 - (c) Если *Next* является безопасным, то освободить все блокировки узлов в *guards*
 - (d) Поместить *Next* в *guards*
 - (e) $Current = Next$

4. Выполнить вставку в лист *Current*
5. Если *Current* оказался переполнен, то разделить его и соответственно изменить его прямого предка. Если прямой предок тоже оказывается переполнен, то рекурсивно подняться и изменить/разделить все нужны узлы
6. Освободить все блокировки в *guards*

Узким местом такого алгоритма является то, что любой поток при вставке сначала получает блокировку на запись корневого узла. Получается, что никакие две вставки не могут начаться параллельно. Одним из вариантов решения этой проблемы является "оптимистичный" latch crabbing:

1. Предполагается что нужный лист безопасен
2. При спуске к листу удерживается не более двух блокировок в один момент(подобно операции чтения), блокировки на чтение
3. Захватывается блокировка на запись на нужный лист
4. Проверяется безопасность листа. Если он безопасен, то происходит вставка и операция завершается. Иначе блокировки снимаются и вызывается обычная вставка

Поскольку лист будет оказываться небезопасным очень редко (примерно каждые t раз, где t — степень ветвления дерева), то в большинстве случаев успешно произойдёт оптимистичная вставка.

2.3. Интеграция в ChunkFS

Интеграция требовала реализации интерфейса Database, который предполагает синхронные методы. Это потребовало создания синхронной обертки для асинхронного хранилища. Реализация обертки:

- При инициализации получает аргументы для инициализации хранилища и асинхронный рантайм из фреймворка Tokio[4]

- При вставке запускается асинхронная функция `insert` самого хранилища в рантайме Tokio
- При чтении используется блокирующий вызов для функции чтения самого хранилища, ведь интерфейс Database требует возвращаемого значения, что заметно ухудшает производительность

Возможна ситуация, когда чтение начинается до завершения ранее запущенной вставки того же ключа. Для решения реализован механизм отслеживания:

- При инициализации обертки создается хэш-множество ключей с незавершенными вставками
- Ключ добавляется в хэш-множество при вызове `insert`
- Ключ удаляется из хэш-множества после фактической записи
- Если чтение пытается найти ключ, который находится в хэш-множестве, то поток ожидает, пока ключ не удалится из хэш-множества

3. Тестирование

3.1. Функциональное тестирование

Асинхронная версия хранилища прошла юнит-тестирование. Для подтверждения потокобезопасности и сохранения консистентности выполнены стресс-тесты:

- 10^6 параллельных операций вставки
- 10^6 параллельных операций чтения

Во всех тестах дерево сохраняло согласованное состояние после параллельных операций.

3.2. Производительность в ChunkFS

Бенчмарки ChunkFS на ядре Linux(1.5GB) в качестве данных с разными алгоритмами чанкирования показали следующие результаты:

Алгоритм чанкирования	Версия	Чтение (с)	Запись (с)
RabinCDC (390 000 записей)	Асинхронная	3.291 ± 1.463	2.552 ± 0.031
	Синхронная	1.451 ± 0.016	6.107 ± 1.215
SeqCDC (710 000 записей)	Асинхронная	5.408 ± 1.338	2.051 ± 0.166
	Синхронная	2.768 ± 0.027	6.484 ± 1.372

Таблица 1: Производительность ChunkFS с разными версиями хранилища

3.3. Производительность хранилища

Тесты изолированного хранилища (10^6 операций):

Версия	Чтение (мс)	Вставка (мс)
Асинхронная	0.286 ± 0.015	2.691 ± 0.041
Синхронная	0.577 ± 0.002	4.116 ± 0.013

Таблица 2: Производительность операций хранилища

3.4. Масштабируемость

Исследование эффективности параллелизации:

- **Чтение:** Среднее время 100 операций в дереве с 10^6 записей
- **Вставка:** Среднее время 1000 операций в изначально пустом дереве (итого 10^6 записей)

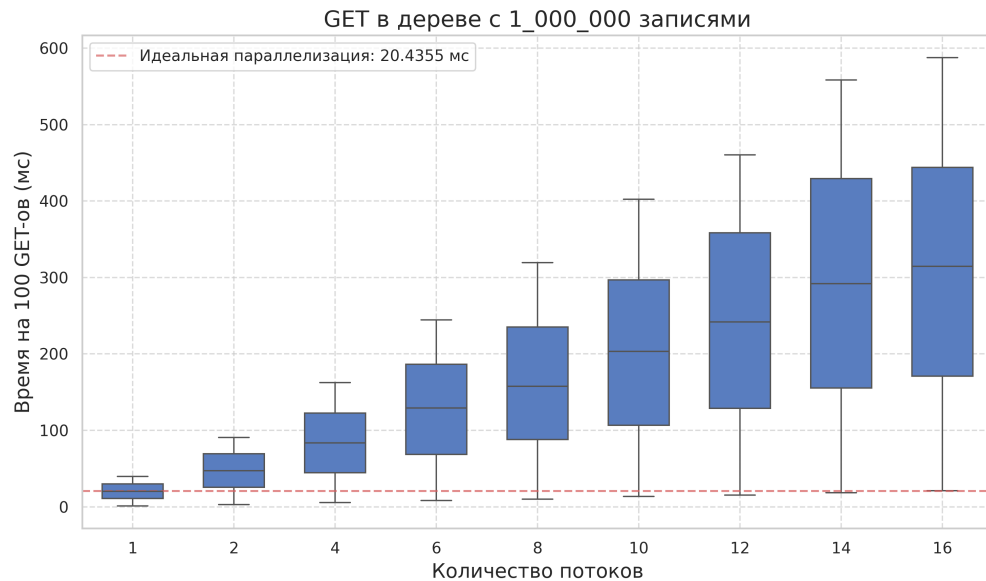


Рис. 1: Масштабируемость операций чтения

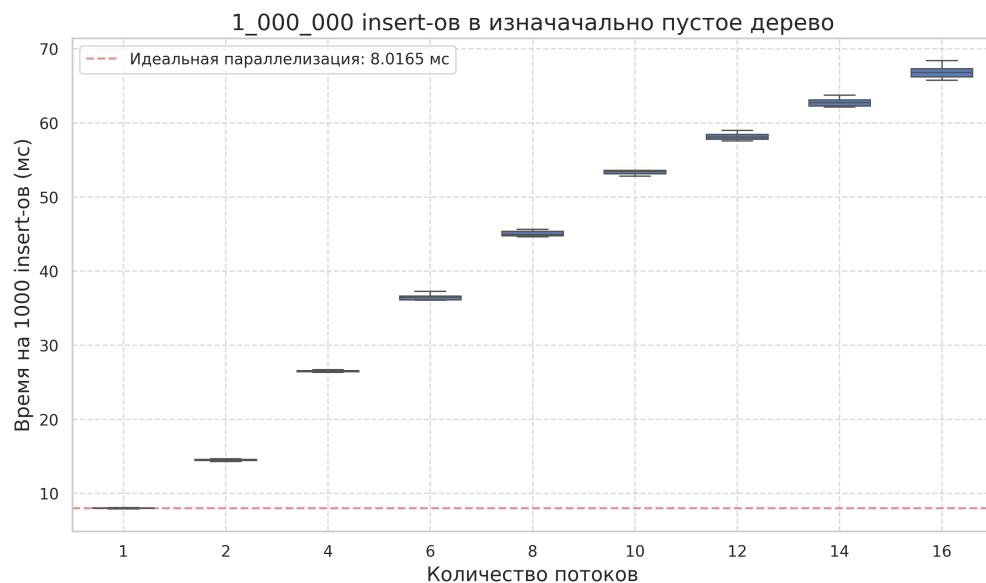


Рис. 2: Масштабируемость операций вставки

3.5. Тестовая среда

Все эксперименты проводились на следующей конфигурации:

- **ОС:** NixOS Linux
- **Процессор:** AMD Ryzen 7 6800H (16 потоков) @ 4.785 GHz
- **Память:** 16 GB RAM

Заключение

В ходе данной работы были выполнены следующие задачи:

- Была реализована стабильная последовательная версия key-value хранилища на основе B+ дерева. Система прошла комплексное тестирование и успешно интегрирована в ChunkFS, подтвердив корректность работы в интеграционных тестах
- Была реализована асинхронная версия хранилища на основе B+ дерева с применением алгоритма latch crabbing для параллельного доступа. Решение протестировано на потокобезопасность и интегрировано в ChunkFS
- Были произведены тесты производительности с использованиями как синхронной, так и асинхронной версий хранилища

Возможное дальнейшее развитие проекта:

- Полная реализация гарантий ACID
 - Текущая версия обеспечивает базовые атомарность, изоляцию и согласованность
 - Требуется добавление механизмов восстановления после сбоев
- Оптимизация интеграции в ChunkFS
- Специфические оптимизации для чанков

Исходный код реализованного хранилища доступен в репозитории

¹.

¹<https://github.com/kamenkremen/BPlusTree>

Список литературы

- [1] II Edward R. Jorgensen. Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree Approaches for Self-Balancing B-Tree. — URL: <https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=4733&context=thesesdissertations> (дата обращения: 20 июня 2025 г.).
- [2] Pavlo Andy. Index Locking Latching. — URL: <https://15721.courses.cs.cmu.edu/spring2017/slides/06-latching.pdf> (дата обращения: 20 июня 2025 г.).
- [3] Sewall Jason. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. — URL: https://www.researchgate.net/publication/220538843_PALM_Parallel_Architecture-Friendly_Latch-Free_Modifications_to_B_Trees_on_Many-Core_Processors (дата обращения: 20 июня 2025 г.).
- [4] Tokio Runtime Documentation. — URL: <https://docs.rs/tokio/latest/tokio/runtime/index.html> (дата обращения: 20 июня 2025 г.).
- [5] Закарлюка Иван. Реализация хранилища чанков для файловой системы ChunkFS на основе B+ дерева. — URL: https://se.math.spbu.ru/thesis/texts/Zakarljuka_Ivan_Vladimirovich_Autumn_practice_2nd_year_2024_text.pdf (дата обращения: 20 июня 2025 г.).
- [6] Пилецкий Олег. Разработка инструмента для сравнения алгоритмов дедупликации. — URL: https://se.math.spbu.ru/thesis/texts/Piletskij_Oleg_Antonovich_Spring_practice_3rd_year_2024_text.pdf (дата обращения: 20 июня 2025 г.).