Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б10-мм

Сравнение алгоритмов дедупликации при помощи инструмента на основе ZboxFS

Пилецкий Олег Антонович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент, к. ф.-м. н. В. И. Гориховский

Санкт-Петербург
2024

Оглавление

Введение

На данный момент существует множество систем хранения и обработки данных, и объемы информации, которые через них проходят, огромны. Одной из новых таких систем с открытым исходным кодом является ZboxFS [13], написанная на языке Rust. Она встраивается в приложение и предоставляет зашифрованную виртуальную файловую систему. Ее целью является безопасное и надежное хранение данных.

Чтобы уменьшить объем хранимых данных и, соответственно, стоимость обслуживания этих систем, используются различные методы. Одним из таких является дедупликация, основанная на том, что в хранилищах содержится довольно большой объем повторяющихся данных. Она состоит из нескольких этапов, а именно из разбиения массива данных на непрерывные отрезки данных, называемые чанками, вычисления их отпечатка (fingerprint), индексирования чанков при помощи этих отпечатков, в процессе которого повторяющиеся чанки удаляются, и их последующего сохранения на диск.

В основном, самый затратный по времени и нагрузке на центральный процессор этап — чанкирование, поскольку необходимо прочесть весь файл побайтово, совершая в процессе некоторые вычисления, благодаря чему можно идентифицировать чанки. Этих алгоритмов существует довольно большое количество, и многие из них были придуманы последние несколько лет, из-за чего нет их сравнения между собой. В самих же статьях отдельные сравнения проводились без привязки к файловой системе, и сами алгоритмы в статьях не были реализованы на Rust. Большинство из них относится к семейству Content Defined Chunking алгоритмов, которые разделяют файл на чанки в зависимости от его содержимого.

В ZboxFS используется один из самых медленных алгоритмов разбиения на чанки, что влияет на его производительность, а также на объем памяти, занимаемый им при использовании. Необходимо встроить в ZboxFS другие алгоритмы чанкинга, позволяющие добиться более высокой степени сжатия данных и не теряющие в скорости, не только

сами по себе, но и при взаимодействии со всей системой, и дать пользователю право выбора необходимого алгоритма для разных сценариев использования.

1. Постановка задачи

Целью работы является разработка системы сравнения алгоритмов чанкинга, используя ZboxFS, и ее использование для определения наиболее эффективных алгоритмов. Для её выполнения были поставлены следующие задачи:

1. выполнить обзор алгоритмов Content Defined Chunking;
2. реализовать избранные алгоритмы на Rust;
3. изучить возможность встраивания алгоритмов чанкинга в ZboxFS;
4. встроить в ZboxFS трейт для возможности менять используемый алгоритм чанкинга в ходе работы программы;
5. провести первичное исследование эффективности алгоритмов в ZboxFS;

2. Обзор алгоритмов дедупликации

Дедупликация - способ сжатия массива данных, исключающий повторяющиеся блоки информации. В данной работе рассматриваются алгоритмы, разбивающие массив данных на отдельные непрерывные отрезки, называемые чанками. Их можно разделить на два типа: Fixed Size Chunking и Content Defined Chunking [5].

1. Fixed Size Chunking — простой в реализации базовый алгоритм [6], разделяющий массив данных на одинаковые по размеру отрезки. Из-за своей простоты работает очень быстро, но достигает очень малого коэффициента дедупликации. Кроме того, если в середину этого массива данных добавить другие данные так, чтобы их размер не был кратен размеру чанка, то все последующие чанки сместятся, и выигрыша в объеме хранимых данных не будет.

2. Content Defined Chunking — семейство алгоритмов, определяющее границы чанков в зависимости от самих данных, что позволяет добиться более высокого коэффициента дедупликации, создавая чанки разных размеров, но эти алгоритмы работают медленнее, чем FSC.

Рассматривались, в основном, алгоритмы из статей, опубликованных за последние 5 лет.

В большинстве алгоритмов используется вычисление скользящего хэша по скользящему окну. Он заключается в том, что берется небольшой массив заданного размера, и заполняется из потока данных. По нему определенным образом считается хэш, и, чтобы перейти к следующему окну, необходимо лишь вычесть значение, соответствующее выпадающему из окна элемента, и прибавить новое. Если хэш соответствует некоторому условию, обычно это равенство "И" с некоторой маской, тогда правая граница окна становится следующей границей чанка.

Также, чтобы чанки не оказались слишком большими, граница чанка ставится, если его размер превысит максимальный, обычно это 64 КБ.

2.1. RabinCDC

RabinCDC [8] — самый медленный алгоритм чанкинга из рассмотренных. В текущей версии ZboxFS для чанкинга применяется он. Для работы он использует скользящее окно, вычисляющее полиномиальный хэш. Он достаточно прост в реализации, но из его недостатков можно отметить большую стоимость вычисления хэша, а также то, что коэффициент дедупликации страдает из-за неоднородности чанков.

2.2. Gear-based CDC (2014)

Gear-based CDC [2] предъявляет другой подход к вычислению хэша скользящего окна, использующий одну операцию битового сдвига, одну операцию обращения по индексу в наполненном случайными числами массиве, состоящем из 256 элементов, и одно сложение. Это позволяет достигнуть трехкратного прироста производительности, практически не теряя в коэффициенте дедупликации. Этот подход допускает различные модификации и оптимизации.

2.3. Leap-based CDC (2015)

Leap-based CDC [7] показывает еще один подход к генерации чанков, также дающий прирост в производительности, однако его сложнее модифицировать, чем GearCDC. Для каждого байта проверяется некоторое число "окон" одинакового размера, находящихся перед ним, начало которых сдвинуто друг относительно друга на один байт. Если все окна "подходят" - то чанк отделяется, если нет - можно пропустить столько байт, сколько окон было проверено. Проверка окна занимает 5 обращений к матрице и 4 XOR операции.

2.4. RapidCDC (2019)

RapidCDC [9] использует то, что похожие чанки с высокой вероятностью будут находиться рядом друг с другом. Вместе с хэшем чанка

записывается размер следующего чанка, что служит подсказкой для обнаружения наиболее вероятной его границы, чтобы избежать использования скользящего окна, просматривающего каждый байт, на повторяющихся данных. Другими словами, этот алгоритм старается находить повторяющиеся группы чанков, находя только лишь первый при помощи скользящего окна. Для определения границы чанка предлагаются несколько критериев.

Если он не находит ни одного такого повторяющегося участка, его производительность падает лишь на 10% по сравнению с GearCDC.

Как и у других алгоритмов, использующих память, у него есть недостаток - информацию необходимо хранить так, чтобы ее можно было переиспользовать для различных файлов, а в ZboxFS чанкинг происходит лишь в пределах одного файла.

2.5. FastCDC (2020)

FastCDC [3] — улучшение GearCDC, а именно, следующих его недостатков

- Маленькое окно (13 байтов у GearCDC и 48 байтов у RabinCDC), из-за чего страдает коэффициент дедупликации, поскольку граница чанка зависит от меньшего числа байт;
- Несмотря на то, что процесс вычисления хэша происходит намного быстрее по сравнению с RabinCDC, проверка хэша занимает, согласно статье, до 60% времени выполнения алгоритма.

В статье предлагается увеличить окно путем дополнения его нулями и упростить условие на конец чанка, что приведет к росту производительности. Чтобы добиться более высокого коэффициента дедупликации, используется подход нормализованного чанкинга, при котором для чанков длины меньше 8КБ условие на конец чанка становится сложнее, а при длине больше - наоборот. Это достигается сменой масок в зависимости от того, сколько байт было рассмотрено. Тем самым количество

чанков маленькой длины становится сильно меньше, и это почти не влияет на производительность.

В качестве еще одной оптимизации предлагается проверка двух байтов за одну итерацию, что, согласно статье, ускоряет работу алгоритма примерно на 40%.

2.6. QuickCDC (2021)

QuickCDC [12] работает похоже на RapidCDC, но не требует того, чтобы повторяющиеся чанки шли подряд, в отличие от него. Может работать быстрее обычного CDC на изолированных повторных чанках, а RapidCDC - нет. Согласно статье, он в 11.4 раза быстрее. Алгоритм состоит из трех оптимизаций обычного CDC:

1. Находить повторяющиеся чанки при помощи первых и последних трех байтов, храня эти данные в таблицах вместе с размером чанка;

2. Если чанк не в таблице, то нужно пропустить столько байт, сколько может занимать чанк минимального размера;

3. Менять битовую маску, чтобы корректировать размеры чанка, как в FastCDC.

Возможно, достигает такого прироста производительности лишь на очень больших датасетах, где действительно скипаются чанки.

2.7. SuperCDC (2022)

SuperCDC [10] совмещает идеи, взятые из FastCDC и RapidCDC:

1. Совмещение эффективных вычислений и "истории", чтобы находить повторяющиеся чанки — как можно большее уменьшение оверхеда на вычисления;

2. Эффективная по памяти структура для трекинга и хранения "истории". Хранится только информация из последнего бекапа. Вместо фингерпринта чанка в качестве ключа в этой структуре выступает gear hash последнего окна чанка;

3. Установка трех различных масок, чтобы нормализовать распределение чанков, включая одну дополнительную, чтобы, если никакие не сработали и чанк достиг максимального размера, в качестве границы чанкинга использовалась та, которая, возможно, была получена с ней;

4. Упрощенные вычисления на чанках длины меньше минимальной.

Согласно статье, он в 4.94 раза быстрее, чем RapidCDC, уменьшает лишний объем занимаемой памяти на 99.58% по сравнению с ним. В 14.21 раз быстрее, чем FastCDC.

2.8. UltraCDC (2022)

UltraCDC [11] работает примерно с такой же скоростью, как FastCDC, но не теряет в производительности на строках с низкой энтропией, так как находит их сразу же.

Этот алгоритм определяет окно размером 8 байт и паттерн для вычисления расстояния Хэмминга между ними. Окно идет по потоку данных и вычисляет это расстояние. Когда оно удовлетворяет заданному условию, на правом конце окна устанавливается конец чанка. Чтобы не было оверхеда на вычисления, этому процессу задаются оптимизации, с помощью которых значение вычисляется за одну операцию сложения и просмотра массива. Маленький размер окна приводит к высокой стабильности чанкинга.

Чтобы не терять в deduplication ratio при слишком маленьких/больших чанках, используется подход с несколькими масками, как в FastCDC.

Чтобы находить строки с низкой энтропией (повторяющиеся символы или паттерны), определяются два буфферных окна с обеих сторон скользящего окна. Если они совпадают - окно пропускается сразу же.

Как только число таких прыжков достигает некоторого значения - чанк обрезается.

2.9. Double Sliding Window (2023)

Алгоритм Double Sliding Window [4] был создан, чтобы хранить данные, связанные с исследованиями океана. Сравнивается только с RabinCDC и достигает максимум 50%-го прироста производительности, что сильно меньше, чем у других рассмотренных алгоритмов.

2.10. Speculative Jump (2023)

Speculative Jump [1] использует две маски как условие для отделения чанка:

- Одна маска определяет границу чанка, вторая - необходимость ”прыгнуть”;
- Чтобы не пересчитывать хэш для каждого байта дважды, вторая маска ”встраивается” в первую, то есть если не выполнено условие для первой маски, то и для второй тоже.

Из-за того, что для проверки соответствия окна маске требуется меньше операций, чем для проверки соответствия окна псевдослучайному условию в LeapCDC, он достигает 57%-го прироста производительности по сравнению с ним. Также, он примерно в 2 раза быстрее, чем FastCDC.

Он может быть встроен в такие алгоритмы как RabinCDC, GearCDC и RapidCDC, увеличивая их производительность и почти не влияя на коэффициент дедупликации.

2.11. Выводы

По результатам обзора из рассмотренных алгоритмов следующие были выбраны для реализации и сравнения как самые используемые и

эффективные, и содержащие отличные друг от друга идеи для разбиения на чанки: FastCDC, Leap-based CDC, SuperCDC, UltraCDC. Также была попытка встроить QuickCDC и Speculative Jump в существующие алгоритмы, но в результате они не привели к росту производительности.

Алгоритмы Gear-based CDC и RapidCDC не были реализованы, поскольку FastCDC и SuperCDC являются их улучшениями.

Алгоритм Double Sliding Window не был реализован, поскольку достигает слишком маленькой скорости по сравнению с другими алгоритмами.

3. Применение ZboxFS для сравнения алгоритмов дедупликации

ZboxFS — это встроенная в приложение файловая система с открытым исходным кодом, написанная на языке Rust, инкапсулирующая файлы и каталоги в зашифрованное хранилище. Поддерживаются различные виды хранилищ, такие как хранилище в оперативной памяти, на диске, в SQLite, в облачном хранилище Zbox. Благодаря дедупликации файлов и данных достигается более экономное использование памяти. Имеются биндинги для других языков, включая Javascript, Java, C и C++.

Используемый алгоритм дедупликации — RabinCDC, один из первых алгоритмов чанкинга. Разбиение на чанки и последующая их обработка в ZboxFS происходит в процессе записи любого файла, однако она способна находить и заменять повторяющиеся чанки только в пределах одного файла. В системе поддерживается версионирование файлов, где может использоваться дедупликация.

Дедупликация данных в ZboxFS происходит в процессе записи данных в файл, и для этого используется структура Chunker. Она принимает на вход структуру, которая должна реализовывать трейты (аналог интерфейса в Rust) Write и Seek, куда будут передаваться чанки. Это позволяет использовать чанкер в различных ситуациях, в частности, можно тестировать различные реализации алгоритмов чанкинга, используемые в системе, при помощи вспомогательной структуры, которая хранит начальную позицию и длину каждого обработанного чанка. Благодаря этому можно узнать коэффициент дедупликации и скорость работы алгоритма.

Для того, чтобы сравнивать то, как различные алгоритмы влияют на производительность всей системы, сделана возможность менять их при создании репозитория или отдельного файла. Поскольку в языке Rust нет перегрузки методов, для создания сложной структуры с множеством параметров вместо конструкторов, как в других языках, принято создавать отдельную вспомогательную структуру, содержа-

щую поля исходной с отсутствующими значениями, либо значениями по умолчанию, которые затем можно менять, вызывая нужные методы. После того, как все необходимые значения были выставлены, вызывается метод, который возвращает экземпляр нужной структуры с этими значениями.

Чтобы использовать алгоритм чанкинга, отличающийся от используемого по умолчанию, применяется метод chunking_algorithm, который принимает один из возможных вариантов перечисления, обозначающих алгоритм. Каждый файл при сохранении в систему проходит процесс разбиения на чанки, но чтобы сократить используемое место на диске, не записывая дублирующиеся чанки, при создании репозитория или при создании каждого файла нужно устанавливать параметр dedup_chunk в true. По умолчанию он false.

4. Реализация

Разбиение на чанки происходит в процессе записи файла в систему, для этого в Zbox используется структура Chunker, в которой содержится поле dst — destination writer. Она реализует несколько трейтов, самым важным из которых является трейт Write. Он содержит два обязательных для реализации метода:

- write, который принимает буфер с данными и производит его запись. В данном случае в нем происходит разбиение на чанки и отправка их в destination writer. Этот метод возвращает тип Result<usize>, возвращающий либо число записанных во внутренний буфер байт, либо ошибку ввода/вывода. Если он вернет Ok(0), то запись будет считаться завершенной.

- flush, который записывает оставшиеся во внутреннем буфере данные. В данном случае он необходим, чтобы, если граница чанка не была обнаружена, но конец файла был достигнут, эти данные не пропали.

В исходной версии ZboxFS алгоритм RabinCDC был реализован в методе write, и его внутренние переменные хранились в структуре Chunker. Кроме того, в этой структуре хранились переменные, относившиеся к внутреннему буферу. В итоговой версии необходимые для его работы поля и методы были выделены в отдельную структуру ChunkerBuf. Чтобы с этой структурой можно было работать, как с содержащимся внутри ее буфером, для нее были реализованы трейты Deref и DerefMut, позволяющие напрямую обращаться к лежащему внутри массиву, а также Index и IndexMut, чтобы к его элементам можно было обращаться по индексу.

Вручную были реализованы алгоритмы LeapCDC, UltraCDC, SuperCDC. Была взята реализация FastCDC из стороннего пакета fastcdc-rs. Также на crates.io есть пакет quickcdc, однако его невозможно использовать в Zbox. Ручная реализация также показала уменьшение производительности по сравнению с исходным алгоритмом, куда он встраивался.

Чтобы сделать поддержку различных чанкеров, был создан трейт Chunking, содержащий метод next_write_range. Он принимает переменную типа ChunkerBuf, поскольку буфер не может храниться в каждой из отдельных реализаций из-за правил заимствования языка Rust, и возвращает Option<Range<usize>>. Этот тип - перечисление, содержащее либо Some(range), если чанкеру хватило данных в буфере и он нашел границу чанка, либо если размер чанка оказался больше максимального или равен ему. None возвращается, когда граница чанка не была достигнута из-за недостатка данных в буфере. Этот трейт был написан для всех реализаций алгоритмов.

В языке Rust полиморфизм достигается при помощи trait-объектов, но их размер при компиляции неизвестен, поэтому они не могут быть помещены на стек и должны быть обернуты в умный указатель. Были выбраны Arc, atomic reference counter, позволяющий нескольким объектам потокобезопасно ссылаться на один и тот же, но не изменять лежащие внутри него данные, и RwLock, read-write lock, позволяющий иметь либо несколько reader, либо одного writer, также потокобезопасный. В ZboxFS поддерживается одновременное чтение одного файла несколькими потоками, что возможно благодаря тому, что все ссылки на объекты имеют тип Arc<RwLock<T>>, поэтому было принято решение обернуть каждую реализацию чанкера в этот же тип. Для удобства для него был создан алиас ChunkerRef, который раскрывается в Arc<RwLock<dyn Chunking>>.

Чтобы это все работало, структуры, реализующие трейт Chunking, должны также поддерживать трейты Send и Sync, позволяющие, соответственно, отправлять объект в другой поток и использовать его из разных потоков. Они реализуются автоматически, если все поля в структуре их поддерживают, и для большинства стандартных типов это ограничение выполнено.

В итоге, структура Chunker содержит три переменных:

- dst, имеющая генерик-тип W, который должен реализовывать трейты Write и Seek;

- buffer, имеющий тип ChunkerBuf, в который производится запись данных, поступающих в трейтовом методе write и над которым затем проводят работу алгоритмы чанкинга;
- chunker, имеющий тип ChunkerRef и выполняющий работу над буфером;

Чтобы можно было удобно обращаться к различным алгоритмам чанкинга, было создано перечисление ChunkingAlgorithm, содержащее на данный момент 5 вариантов, соответствующих реализованным алгоритмам, используемое при создании экземпляра структуры Chunker.

5. Эксперимент

Необходимо было протестировать как работу самих алгоритмов чанкинга, так и то, как ведет себя система при использовании различных алгоритмов. Проводился базовый эксперимент.

5.1. Условия эксперимента

Эксперимент проводился на системе Manjaro Linux 23.1.0, процессор: Intel Core i5-11300H. Версия компилятора rustc: 1.75.0. Бенчмаркинг проводился при помощи стороннего пакета criterion, запуск бенчмарков производится при помощи команды cargo bench. Код бенчмарков можно найти в файле perf.rs по ссылке.

Были выбраны датасеты с повторяющимися данными, чтобы можно было сравнить то, как хорошо различные алгоритмы чанкинга справляются с дедупликацией:

- Несколько копий исходного кода ядра Linux версий с 6.0 по 6.3;
- Enron (набор электронных писем);
- Образ Ubuntu 22.04.

Чтобы проверить коэффициент дедупликации при работе отдельного алгоритма, используется структура, которая записывает в себя полученные чанки. Тестирование проводилось в модуле Chunker, недоступном извне исходного кода ZboxFS, поэтому код тестов содержится в подмодуле в файле chunker.rs.

Чтобы протестировать скорость работы и пропускную способность ZboxFS, были выбраны тестовые сценарии записи одного файла в память, чтения одного файла из памяти, и копирования файла внутри zbox-репозитория.

ZboxFS поддерживает несколько видов хранилищ, основные из них - хранилище в файловой системе (file) и хранилище в оперативной памяти (mem). Измерение скорости работы проводилось на них обоих.

5.2. Метрики

- **Коэффициент дедупликации**, показывающий, сколько повторяющихся данных смог найти и устранить алгоритм. Считается как размер данных до / размер данных после;
- **Скорость работы**;
- **Средний размер чанка**;

5.3. Результаты

Рассматривая работу алгоритмов в отрыве от файловой системы, можно видеть, что реализованные алгоритмы значительно превосходят оригинальный в пропускной способности, однако коэффициент дедупликации высок лишь на наборе данных с несколькими версиями ядра Linux, и оригинальный превосходят лишь алгоритмы Leap-based CDC и UltraCDC. Второй из них позиционируется как подходящий для массивов данных с низкой энтропией. По скорости лучше всех себя показывает SuperCDC, однако она сильно зависит от массива данных, в этом плане UltraCDC и FastCDC более стабильны.

При применении Leap-based CDC к набору данных Enron среднеквадратичное отклонение довольно высоко. Вероятно, это происходит из-за того, что этот набор данных содержит большое число очень маленьких файлов, и, кроме того, таблица, необходимая для определения границы чанка при работе алгоритма, каждый раз генерируется заново.

При работе файловой системы не наблюдается такого же прироста производительности, как при процессе простого разбиения файла на чанки, при использовании других алгоритмов.

1. Для операции записи лучше всего себя показывает Leap-based CDC.
2. При чтении на датасете Linux выигрывают другие алгоритмы.
3. На других наборах данных оригинальный алгоритм показывает лучшие результаты.

	Linux	Ubuntu	Enron
Rabin	0.71	0.945	0.980
Leap	0.632	0.936	0.975
Fast	0.950	0.998	0.996
Super	0.875	0.977	0.975
Ultra	0.649	0.985	0.996

Таблица 1: Коэффициент дедупликации (меньше — лучше)

	Linux	Ubuntu	Enron
Rabin	285.2 ± 0.3	421 ± 0.8	305 ± 0.8
Leap	1789 ± 87	1806 ± 6	3183 ± 725
Fast	2802 ± 9	2869 ± 24	2763 ± 18
Super	4034 ± 8	5485 ± 28	3725 ± 32
Ultra	1025 ± 8	1022 ± 17	1030 ± 12

Таблица 2: Пропускная способность алгоритмов дедупликации, МБ/с

4. Копирование быстрее всего происходит при использовании RabinCDC, вероятно, из-за наибольшего размера чанка, и, возможно, напрямую зависит от него.

	Linux	Ubuntu	Enron
Rabin	40.5	32	38.2
Leap	21	20.3	17.9
Fast	5.1	4.9	5.2
Super	6.6	5.5	6.9
Ultra	14.7	11.3	20.7

Таблица 3: Средний размер чанков, КБ

Условия	Rabin	Leap	Fast	Super	Ultra
Linux, file	316 ± 1	316 ± 1	252.5 ± 0.1	258.8 ± 0.3	264.3 ± 0.7
Ubuntu, file	297 ± 0.3	332.5 ± 0.4	309.7 ± 0.3	299.3 ± 0.6	277.1 ± 0.2
Enron, file	284 ± 0.3	354.6 ± 3	310 ± 1	310.7 ± 1	290 ± 1
Linux, mem	287 ± 0.6	329 ± 1	260.4 ± 0.1	268.7 ± 0.1	273.4 ± 0.2
Ubuntu, mem	308 ± 0.3	346.8 ± 0.4	323 ± 1	312.1 ± 0.5	286.8 ± 0.3
Enron, mem	295 ± 0.1	367.2 ± 4	322.9 ± 0.5	320.9 ± 0.3	299.5 ± 0.5

Таблица 4: Пропускная способность операции записи в ZboxFS, МБ/с

Условия	Rabin	Leap	Fast	Super	Ultra
Linux, file	335 ± 11	404 ± 7	445 ± 24	341 ± 20	411 ± 15
Ubuntu, file	907 ± 1	878 ± 0.8	942 ± 1	920 ± 1	894 ± 2
Enron, file	906.4 ± 1	897.1 ± 1	920.2 ± 6.2	921.5 ± 1	907 ± 1
Linux, mem	615 ± 1	953 ± 2	942 ± 2	904 ± 1	890 ± 1
Ubuntu, mem	1002 ± 2	979 ± 1	987 ± 1	969 ± 1	940 ± 1
Enron, mem	993.7 ± 3	979.2 ± 3	924.5 ± 1	945 ± 2	957 ± 1

Таблица 5: Пропускная способность операции чтения в ZboxFS, МБ/с

Условия	Rabin	Leap	Fast	Super	Ultra
Linux, file	59.3 ± 0.2	36 ± 0.5	16.1 ± 0.1	19.2 ± 0.1	27.5 ± 0.1
Ubuntu, file	50.3 ± 0.1	36.3 ± 0.1	15.9 ± 0.1	12.5 ± 0.1	22.3 ± 0.1
Enron, file	56.7 ± 0.2	33.6 ± 0.6	17.5 ± 0.5	19.6 ± 0.1	36.4 ± 0.1
Linux, mem	61.4 ± 0.1	36.8 ± 0.2	17 ± 0.1	19.4 ± 0.1	27.8 ± 0.1
Ubuntu, mem	52.6 ± 0.1	37.8 ± 0.1	16.5 ± 0.1	12.9 ± 0.1	23.1 ± 0.1
Enron, mem	59 ± 0.1	33.9 ± 0.4	17.5 ± 0.5	19.6 ± 0.5	37.6 ± 0.1

Таблица 6: Пропускная способность операции копирования в ZboxFS, ГБ/с

Заключение

В течение семестра были изучены, реализованы и встроены в ZboxFS некоторые алгоритмы Content Defined Chunking, включая самые новые, было проведено их сравнение с ее помощью.

- был выполнен обзор алгоритмов Content Defined Chunking;
- избранные алгоритмы были реализованы на Rust;
- эти алгоритмы были встроены в ZboxFS;
- в ZboxFS был встроен трейт, позволяющий менять используемый алгоритм чанкинга в ходе работы программы;
- было проведено первичное исследование эффективности алгоритмов;

Благодаря достаточно гибкой архитектуре ZboxFS было возможно модифицировать компонент, лежащий глубоко в иерархии структур, и добавить новую функциональность, доступную конечному пользователю таким же образом, как и другие похожие функциональные возможности. Это позволяет проводить количественное сравнение алгоритмов Content Defined Chunking как самих по себе, так и в "боевых" условиях, интегрируя их в файловую систему Zbox. Рассмотренные алгоритмы позволяют уменьшить используемое место на диске, однако каждый из них подходит для разных сценариев и при чтении/записи работает поразному, поэтому при открытии файла можно выбрать необходимый алгоритм в зависимости от ситуации.

В весеннем семестре планируется продолжать работу над инструментом, рассмотреть другие тестовые сценарии, реализовать больше алгоритмов и оптимизировать уже реализованные.

Код реализации можно найти в репозитории по ссылке, ветка: dynchunkers.

Список литературы

[1] Accelerating Content-Defined Chunking for Data Deduplication Based on Speculative Jump / Xiaozhong Jin, Haikun Liu, Chencheng Ye et al. // IEEE Transactions on Parallel and Distributed Systems. — 2023. — Vol. 34, no. 9. — P. 2568–2579.

[2] Ddelta: A deduplication-inspired fast delta compression approach / Wen Xia, Hong Jiang, Dan Feng et al. // Performance Evaluation. — 2014. — Vol. 79. — P. 258–272. — Special Issue: Performance 2014. URL: https://www.sciencedirect.com/science/article/pii/S0166531614000790.

[3] The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems / Wen Xia, Xiangyu Zou, Hong Jiang et al. // IEEE Transactions on Parallel and Distributed Systems. — 2020. — Vol. 31, no. 9. — P. 2017–2031.

[4] Double Sliding Window Chunking Algorithm for Data Deduplication in Ocean Observation / Shuai Guo, Xiaodong Mao, Meng Sun, Shuang Wang // IEEE Access. — 2023. — Vol. 11. — P. 70470–70481.

[5] Feng Dan. Chunking Algorithms // Data Deduplication for High Performance Storage System. — Singapore : Springer Nature Singapore, 2022. — P. 25–51. — ISBN: 978-981-19-0112-6. — URL: https://doi.org/10.1007/978-981-19-0112-6_3.

[6] Krishnaprasad P.K., Narayamparambil Biju Abraham. A Proposal for Improving Data Deduplication with Dual Side Fixed Size Chunking Algorithm // 2013 Third International Conference on Advances in Computing and Communications. — 2013. — P. 13–16.

[7] Leap-based Content Defined Chunking — Theory and Implementation / Chuanshuai Yu, Chengwei Zhang, Yiping Mao, Fulu Li // 2015 31st Symposium on Mass Storage Systems and Technologies (MSST). — 2015. — P. 1–12.

[8] Muthitacharoen Athicha, Chen Benjie, Mazières David. A Low-Bandwidth Network File System. — Vol. 35. — 2001. — 12. — P. 174–187.

[9] Ni Fan, Jiang Song. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based Deduplication Systems // Proceedings of the ACM Symposium on Cloud Computing. — SoCC '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 220–232. — URL: https://doi.org/10.1145/3357223.3362731.

[10] SuperCDC: A Hybrid Design of High-Performance Content-Defined Chunking for Fast Deduplication / Binzhaoshuo Wan, Lifeng Pu, Xiangyu Zou et al. // 2022 IEEE 40th International Conference on Computer Design (ICCD). — 2022. — P. 170–178.

[11] UltraCDC:A Fast and Stable Content-Defined Chunking Algorithm for Deduplication-based Backup Storage Systems / Peng Zhou, Zhenyu Wang, Wen Xia, Haotong Zhang // 2022 IEEE International Performance, Computing, and Communications Conference (IPCCC). — 2022. — P. 298–304.

[12] Xu Zhen, Zhang Wenbo. QuickCDC: A Quick Content Defined Chunking Algorithm Based on Jumping and Dynamically Adjusting Mask Bits // 2021 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom). — 2021. — P. 288–299.

[13] ZboxFS — A zero-details, privacy-focused in-app file system. — 2019. — URL: https://zbox.io/fs/.