



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
(ДГТУ)**

Факультет «Информатика и вычислительная техника»

Кафедра «Кибербезопасность информационных систем»

КУРСОВАЯ РАБОТА

Тема: «ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АРХИВАТОРА ДАННЫХ»

Дисциплина: «Теория информации»

Специальность: 10.05.01 Компьютерная безопасность

Специализация: Математические методы защиты информации

Обозначение курсовой работы ТИ.990000.000 Группа ВКБ32

Обучающийся _____ Д. П. Ковалев
подпись, дата

Курсовая работа защищена с оценкой _____

Руководитель работы _____ ст. преподаватель, И. А. Алферова
подпись, дата



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
(ДГТУ)**

Факультет «Информатика и вычислительная техника»

Кафедра «Кибербезопасность информационных систем»

ЗАДАНИЕ

на выполнение курсовой работы

Тема «ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АРХИВАТОРА ДАННЫХ»

Дисциплина: Теория информации

Обучающийся: Ковалев Данил Петрович

Обозначение курсовой работы ТИ.990000.000 Группа: ВКБ32

Срок представления работы к защите «__» _____ 20__ г.

Исходные данные для курсовой работы:

1. Задание на выполнение курсовой работы
2. Горев, А И; Симаков А.А Обеспечение Информационной Безопасности / А Горев А.И; Симаков А. – Москва: Мир, 2005. – 844 с.

Содержание пояснительной записки

Введение:

Описывается появление самых первых алгоритмов сжатия, а также актуальность использования этих методов.

Разделы основной части:

1. В разделе "Анализ алгоритмов сжатия" дается полное описание fastlz, gunzip, pgzip, включая основные принципы, алгоритмы, механизмы сжатия, а также применение и особенности использования.
2. В разделе "Программная реализация архиватора" дается обоснование выбора языка программирования и среды разработки, выбора базы данных, S3 хранилища, архитектуры приложения, описываются основные методы и классы программы, а также показывается, как выглядит программное средство.
3. В разделе "Работоспособность программного средства" представлены примеры работы программы, сравнение объемов файлов до сжатия и после.
4. В разделе "Сравнительный анализ алгоритмов сжатия" проводится сравнительный анализ данных алгоритмов, демонстрируется их работоспособность, входные и выходные данные, а также рассматриваются преимущества и недостатки каждого алгоритма.

Заключение:

В рамках данной курсовой работы было разработано программное средство – архиватор.

Руководитель работы

подпись, дата

И. А. Алферова

Задание принял к исполнению

подпись, дата

Д. П. Ковалев

Содержание

Введение	5
1. Анализ алгоритмов сжатия	6
1.1. Утилита gzip	6
1.2. Алгоритм FastLZ	8
1.2.1. Описание работы FastLZ Level 1	8
1.2.2. Описание работы FastLZ Level 2	10
1.3. Утилита Pigz	12
2. Программная реализация архиватора данных	16
2.1. Выбор языка программирования, инструментов и архитектуры	16
2.2. Основные этапы работы программы	17
2.3. Реализация алгоритмов компрессии и декомпрессии	18
3. Работоспособность программного средства	24
4. Сравнительный анализ алгоритмов	25
Заключение	26
Перечень используемых информационных ресурсов	27
Приложение А Листинг кода	28

					ТИ.990000.000 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата				
Разраб.		Ковалев Д.П.			Программная реализация архиватора данных Пояснительная записка	Лит.	Лист	Листов
Провер.		Алферова И.А.					4	36
						ДГТУ Кафедра КБИС		
Н.контр.								
Утв.		Сафарьян О.А.						

Введение

В современном информационном обществе объемы данных растут с каждым днем, что делает эффективное управление и хранение информации одной из ключевых задач. Архивирование данных представляет собой важный инструмент, позволяющий уменьшить занимаемое пространство и упростить процесс передачи информации. В этом контексте разработка архиватора с интерфейсом командой строки (CLI) становится актуальной задачей, позволяющей таким специалистам, как DevOps эффективно сжимать такие данные: дампы базы данных, документы и т. п.

Объектом курсовой работы являются алгоритмы сжатия: FastLZ, Pigz, Gunzip.

Предметом данной работы является разработка программного средства – архиватор.

Целью данной курсовой работы является создание программного обеспечения для архивирования данных, которое будет реализовывать алгоритмы сжатия и обеспечивать удобный интерфейс для взаимодействия с потенциальным инженером.

В рамках данной работы ставятся следующие цели:

1. Провести анализ алгоритмов fastlz, gzip, pgzip включая их математические принципы и особенности функционирования.
2. Разработать программное обеспечение для архиватора, включая алгоритмы сжатия, разжатия данных.
3. Проверить разработанное программное средство на работоспособность.
4. Выявить преимущества и недостатки каждого из алгоритмов с целью определения их пригодности для конкретных задач в различных областях применения.

В первой части будет представлен обзор теоретических основ сжатия информации, включая алгоритмы: FastLZ, gzip, pgzip. Далее будет описан процесс разработки архиватора, его архитектура, включая реализацию алгоритмов сжатия и распаковки. Также будет обоснован выбор базы данных для тестирования приложения, выбор S3 хранилища, а также технология Docker.

					ТИ.990000.000 ПЗ	Лист
						5
Изм.	Лист	№ докум.	Подпись			

1. Анализ алгоритмов сжатия

1.1. Утилита gzip

gzip (GNU zip) – это утилита для сжатия и восстановления данных, использующая алгоритм Deflate. Она широко применяется для сжатия интернет – трафика и является стандартом для сжатия данных в ряде UNIX – систем [1]. Как уже упоминалось, gzip основан на алгоритме Deflate – это алгоритм сжатия без потерь, который сочетает в себе методы LZ77 и кодирования Хаффмана. Утилита была разработана Жан – Лу Гайи и Марком Адлером, первая версия 0.1 была выпущена 31 октября 1992 года, а версия 1.0 – в феврале 1993 года.

Другими словами, gzip – это практическое применение алгоритма Deflate. Процесс сжатия начинается с анализа входных данных для выявления повторяющихся последовательностей. Алгоритм LZ77 заменяет эти повторяющиеся фрагменты ссылками на их предыдущие вхождения, что позволяет значительно сократить объем данных. Каждая ссылка состоит из пары значений: смещение (offset) и длины (length), которые указывают на позицию и длину повторяющейся последовательности в обработанных данных.

После применения алгоритма LZ77, полученные данные передаются на этап кодирования Хаффмана. Этот метод создает переменные длины кода для символов, основываясь на их частоте появления: более частые символы получают более короткие коды, а реже встречающиеся – более длинные. Это позволяет дополнительно уменьшить размер данных.

При распаковке данных процесс происходит в обратном порядке. Сначала декодируются коды Хаффмана, восстанавливая последовательности, а затем алгоритм LZ77 использует ссылки для восстановления оригинальных данных. Таким образом, алгоритм Deflate обеспечивает эффективное сжатие и восстановление данных, что делает его идеальным для использования в утилите gzip.

					ТИ.990000.000 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись			6

Достоинства:

1. Эффективность сжатия: gunzip обеспечивает достаточно высокую степень сжатия.
2. Скорость работы: Утилита демонстрирует хорошую производительность как при сжатии, так и при распаковке данных. Алгоритм Deflate оптимизирован для быстрого выполнения операций, что делает gunzip подходящим для использования в реальном времени.
3. Отсутствие потерь: gunzip использует алгоритм сжатия без потерь, что означает, что оригинальные данные могут быть полностью восстановлены после распаковки. Это критически важно для многих приложений, где сохранение целостности данных является приоритетом.
4. Широкая поддержка: gunzip мало того, что является стандартной утилитой в Unix – подобных системах, но он также поддерживается в большинстве браузеров. Данная утилита активно применяется в Web серверах по типу Nginx.

Недостатки:

5. Неэффективность при малых файлах: при сжатии небольших файлов эффективность gunzip может быть ниже, чем при работе с большими объемами данных. Это связано с накладными расходами на обработку и метаданные.
6. Отсутствие встроенной защиты: gunzip не предоставляет никаких средств для шифрования или защиты данных, что может быть проблемой в ситуациях, где требуется безопасность информации.

					ТИ.990000.000 ПЗ	Лист
						7
Изм.	Лист	№ докум.	Подпись			

1.2. Алгоритм FastLZ

FastLZ — это высокоскоростной алгоритм сжатия данных без потерь, ориентированный на минимальные задержки и высокую производительность. Он был создан Амитэбом Мукерджи (Amitabh Mukherjee) и впервые представлен в 2005 году. Алгоритм распространяется под лицензией MIT, что делает его свободным для использования в открытых и коммерческих проектах [2].

FastLZ основан на модификации алгоритма LZ77, но с упрощениями, которые ускоряют обработку данных. Вот ключевые этапы его работы:

1. Деление на блоки: входной буфер делится на независимые блоки. Каждый блок обрабатывается отдельно.
2. Поиск повторяющихся последовательностей: алгоритм ищет совпадения в скользящем окне с размером от 8 до 32 КБ. Для сохранения и поиска обнаружений используется хэш — таблица, которая хранит ранее встреченные 3-байтовые последовательности.
3. При обнаружении повторяющихся последовательностей генерируется ссылка в формате (offset, length), где offset — расстояние от текущей позиции до начала встреченной последовательности. Максимальное расстояние зависит от уровня FastLZ.
4. Если совпадение не найдено или между совпадениями имеются данные, они копируются как литералы. Каждая инструкция литералов кодирует от 1 до 32 байт.

1.2.1. Описание работы FastLZ Level 1

У данного уровня есть некоторые определенные параметры, которые его отличают от Level 2. Процесс работы кодирования можно представить в виде трех этапов:

1. Инициализация: Исходный буфер разделяется на блоки до 65535 байт. В данной конфигурации обычно используется хэш-таблица размером 16384

					ТИ.990000.000 ПЗ	Лист
						8
Изм.	Лист	№ докум.	Подпись			

символа элементов, но это зависит от реализации. Смещения – 8192 байт. Максимальная длина совпадения – 264 байта.

2. Поиск совпадений: обработка начинается с позиции, смещенной на 2 байта от начала, чтобы избежать ложного самосравнения первых 3 байтов. На каждой итерации считывается 3 байта, для которых вычисляется хэш – значение. Если в хэш – таблице по этому индексу уже сохранена позиция, происходит сравнение текущей последовательности с сохраненной для определения длины совпадения (минимум 3 байта).
3. Генерация инструкций сжатия: если между текущей позицией и позицией начала совпадения есть данные, они сначала кодируются как литералы. При обнаружении совпадения выбирается тип инструкции: `short match` – используется для совпадений длиной от 3 до 8 байт. При этом старших 3 бита первого байта опкода задают длину совпадения, а оставшиеся биты вместе со вторым байтом содержат смещение. `Long match` – применяется, если длина совпадения превышает 8 байт. Здесь первый байт начинается с шаблона “111”, второй байт кодирует длину совпадения (со смещением) и третий байт – младшие 8 бит смещения. Если совпадение слишком длинное, инструкция разбивается на несколько частей.

Декодирование. Процесс декомпрессии осуществляется «на лету» — по мере чтения сжатого потока декомпрессор последовательно обрабатывает каждую инструкцию, восстанавливая исходный (несжатый) поток данных. К ключевым этапам работы декомпрессора можно отнести:

1. Инициализация: Декомпрессор задаёт начальные позиции для чтения из входного буфера (`source_position`) и записи в выходной буфер (`destination_position`). Первым шагом считывается инструкция, при этом извлекаются последние 5 бит первого байта. Эти биты определяют тип инструкции: если значение равно или превышает 32, инструкция интерпретируется как `literal run`. Если значение равно или превышает 32, инструкция интерпретируется как `match` (инструкция совпадения).

					ТИ.990000.000 ПЗ	Лист
						9
Изм.	Лист	№ докум.	Подпись			

2. Обработка литералов: при получении инструкции литералов значение, полученное из 5 бит, интерпретируется как длина литералов минус 1 (то есть значение 0 означает, что нужно скопировать 1 байт).
3. Обработка совпадений: если инструкция является match – инструкцией, декомпрессор выполняет следующие шаги. Из старших бит первого байта (полученных путем побитового сдвига на 5 позиций) вычисляется базовая длина совпадения. При этом к значению производится корректировка (вычитается 1), так как минимальная длина совпадения равна 3 байтам.
4. Извлечение смещения: пять наименее значащих бит первого байта, сдвинутые влево на 8 бит, образуют старшую часть смещения. Далее декомпрессор считывает следующий байт, содержащий младшую часть смещения, и уменьшает вычисленное значение на единицу (так как смещение 0 не имеет смысла, оно трактуется как ссылка на последний байт выходного буфера). Таким образом, смещение определяется как число байт, на которое нужно вернуться в уже восстановленном выходном буфере для копирования совпадающей последовательности.
5. Обработка длинных совпадений: если базовая длина совпадения равна 6 (что соответствует коду “7” в 3 – х старших битах, т. е. сигнализирует о длинном совпадении), декомпрессор считывает дополнительный байт, который прибавляется к базовой длине для получения окончательной длины совпадения.
6. Восстановление данных: после определения длины совпадения (с добавлением базового значения 3) и вычисления смещения, декомпрессор определяет индекс в выходном буфере минус смещение (с поправкой на единицу). Далее производится копирование блока [5].

1.2.2. Описание работы FastLZ Level 2

FastLZ Level 2 представляет собой оптимизированную версию для Level 1. Основные отличия заключаются в поддержке более длинных совпадений и

					ТИ.990000.000 ПЗ	Лист
						10
Изм.	Лист	№ докум.	Подпись			

расширенном диапазоне смещений, что достигается за счет дополнительных проверок и использовании гамма – кодирования для кодирования длины совпадения. За счет этого компрессия становится выше. Этапы работы:

1. Инициализация: входной буфер разбивается на блоки, аналогично уровню создаётся хэш – таблица с размером до 16384 записей. Для предотвращения ложных совпадений первые два байта пропускаются, поскольку минимальное совпадение должно иметь длину не менее 3 байт.
2. На каждой итерации считываются 3 байта, для которых вычисляется хэш. Хэш-таблица хранит позиции ранее встреченных последовательностей, что позволяет быстро найти потенциальное совпадение. Если найденная позиция удовлетворяет условию совпадения (минимум 3 байта), дополнительно проверяется, что при больших смещениях первые 5–6 байт совпадают. Это обеспечивает корректность при использовании расширенного кодирования для больших смещений.
3. Эмитирование литералов: если между текущей позицией и обнаруженным совпадением имеются данные, они сначала кодируются как литералы. Литералы группируются в блоки от 1 до 32 байт [3].
4. Кодирование: если смещение превышает 8191 байт, то смещение уменьшается на 8191 байт. В short match инструкции в первые 5 битах опкода записывается максимальное значение (31), что сигнализирует о расширенном формате. Затем добавляется байт со значением 255, после чего записываются два байта, кодирующие оставшееся смещение (16 битное значение). Аналогично для long match инструкций, после установки первых битов опкода (с шаблоном 111 и значением 31 в 5 битах) длина совпадения кодируется с помощью гамма – кода, а затем дополнительно указывается байт со значением 255 и два байта для расширенного смещения.
5. Маркировка уровня: после завершения кодирования блока в первый байт результата добавляется специальный маркер, который устанавливается путем установки одного бита (сдвиг 1 в 5-м бите). Это позволяет декомпрессору определить, что сжатие выполнено по алгоритму Level 2.

					ТИ.990000.000 ПЗ	Лист
						11
Изм.	Лист	№ докум.	Подпись			

Процесс декомпрессии (распаковки) также можно представить в виде этапов:

1. Считывание первого байта: сначала читается первый байт, из которого определяется тип инструкции. Первые 3 бита помогают отличить литералы от инструкций совпадения, а наличие установленного бита указывает на использование Level 2.
2. Если инструкция указывает на литералы, декомпрессор копирует указанное число байт напрямую в выходной буфер.
3. Если инструкция представляет собой совпадение: из опкода извлекается длина совпадения и часть смещения. В случае длинных совпадений есть значение длины равно максимальному для короткого совпадения, считываются дополнительные байты с гамма – кодированием для получения итоговой длины.
4. При обнаружении расширенного смещения (значения 255 в дополнительном байте) декомпрессор считывает ещё два байта для восстановления полного 16 – битного смещения. После вычисления смещения и длины, декомпрессор копирует указанное число байт из уже восстановленных данных, используя функцию, которая корректно обрабатывает перекрывающиеся участки.

Таким образом, FastLZ Level 2 достигает лучшего коэффициента сжатия за счет более гибкого кодирования совпадений и поддержки расширенного диапазона смещений. При этом, несмотря на дополнительную сложность, алгоритм продолжает работать “на лету”, динамически обновляя хеш – таблицу и формируя инструкции сжатия по мере обхода входного буфера.

1.3. Утилита Pigz

Pigz - (Parallel Implementation of GZip) – это параллельная реализация популярного формата сжатия gzip, разработанная для использования всех доступных ядер процессорах. Изначально идея pigz принадлежит Марку Адлеру, одному из создателей библиотеки zlib, что позволило использовать алгоритм DEFLATE (на котором основан gzip) в многопоточном режиме. pigz разработан с учётом

					ТИ.990000.000 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись			12

современных требований к скорости и масштабируемости, особенно при работе с большими файлами, и распространяется под открытой лицензией, что делает его популярным как в открытых, так и в коммерческих проектах [4].

Алгоритм PIGZ организован вокруг параллельного сжатия файла по блокам.

Этапы работы алгоритма:

1. Подготовка и инициализация: задается уровень сжатия (от 1 до 9, где 1 – самый быстрый, но с худшей степенью сжатия, а 9 – самый эффективный, но медленный), размер блока (по умолчанию 128 КБ) и количество рабочих потоков (обычно равное количеству ядер процессора)
2. Создание потоков: для обработки файла используются два потока: одна чтения исходного файла и распределения данных по блокам, другая – для записи сжатых блоков в выходной файл. Помимо этого, создается пул рабочих потоков, который параллельно обрабатывает каждый блок.
3. Чтение файла и разделение по блоки: далее мы читаем файл порциями. Каждой порции присваивается уникальный номер (номер чанка), что обеспечивает возможность последующей сортировки. Прочитанные блоки передаются в пул потоков, где мы уже обрабатываем данные.
4. Параллельное сжатие блоков: для каждого блока происходит обработка с помощью алгоритма DEFLATE. Каждый блок обрабатывается независимо, что позволяет задействовать все доступные ядра процессора.
5. Запись сжатых данных: сжатые блоки, помеченные их уникальными номерами (чанками), помещаются в приоритетную очередь. Это обеспечивает восстановление исходного порядка блоков независимо от того, в каком порядке они завершают обработку. Затем данные извлекаются из очереди и записываются в выходной файл. Перед началом записи формируется gzip – заголовок, включающий магические числа, информацию о методе сжатия, флаги (например, наличие имени исходного файла), время модификации, дополнительные флаги и идентификатор операционной системы.
6. Финализация и формирование трейлера: после записи всех сжатых блоков

					ТИ.990000.000 ПЗ	Лист
						13
Изм.	Лист	№ докум.	Подпись			

утилита завершает работу следующим образом: вычисляется и записывается контрольная сумма (CRC32) исходных данных, записывается размер исходного файла (ISIZE) в трейлер. Эти данные позволяют при декомпрессии проверить целостность и корректность восстановленного файла.

Таким образом, pigz использует параллельную обработку, что существенно ускоряет сжатие больших файлов, сохраняя при этом полную совместимость с форматом gzip.

Достоинства утилиты:

1. Повышенная скорость сжатия: благодаря использованию параллельной обработки, pigz существенно ускоряет сжатие больших файлов за счет распределения работы между всеми доступными ядрами процессора.
2. Масштабируемость и эффективное использование ресурсов: многопоточная архитектура позволяет оптимально использовать современные многоядерные системы, что особенно актуально при обработке объемных данных в серверных и высоконагруженных средах [4].
3. Полная совместимость с gzip: выходной файл, созданный pigz, полностью соответствует стандарту gzip, что обеспечивает возможность его распаковки любыми стандартными утилитами, такими как gunzip, 7z.
4. Отсутствие потерь: как и gzip, pgzip реализует алгоритм сжатия без потерь, что гарантирует восстановление исходных данных без каких-либо изменений.

Недостатки утилиты:

1. Накладные расходы при работе с малыми файлами: для небольших файлов затраты на управление потоками и параллельную обработку могут снизить общую эффективность, делая pigz менее оптимальным по сравнению с однопоточной реализацией gzip.
2. Зависимость от аппаратного обеспечения: эффективность pigz напрямую зависит от наличия нескольких ядер процессора. На системах с одним ядром его преимущества не проявляются.

					ТИ.990000.000 ПЗ	Лист
						14
Изм.	Лист	№ докум.	Подпись			

3. Усложненное управление ресурсами: многопоточная архитектура требует дополнительной синхронизации и контроля за порядком обработки блоков, что может усложнять отладку и настройку при возникновении непредвиденных ситуаций.
4. Потенциальное увеличение потребления памяти: параллельное сжатие и хранение промежуточных блоков может приводить к большему потреблению памяти, что следует учитывать при работе с большими файлами или на системах с ограниченными ресурсами.

Таким образом, *rigz* является отличным решением для ускорения сжатия больших файлов на многоядерных системах, однако его преимущества могут нивелироваться при работе с небольшими данными или на однопроцессорных системах.

					ТИ.990000.000 ПЗ	Лист
						15
Изм.	Лист	№ докум.	Подпись			

2. Программная реализация архиватора данных

2.1. Выбор языка программирования, инструментов и архитектуры

Для реализации архиватора вполне целесообразно будет обратить внимание на Python. Это высокоуровневый, динамически типизированный язык программирования с высоким уровнем абстракции, обладающий лаконичным синтаксисом и имеющий обширную стандартную библиотеку функций. В отличие от C++ и C#, Python обеспечивает более удобную работу со структурами данных (например, возможность динамического изменения списков) и возможность приведения переменной к новому типу (например, получение целого числа из строки в двоичном виде). Также несомненным плюсом является форматирование строк и возможность работы с байтовыми строками. Таким образом, Python является оптимальным выбором для реализации поставленной задачи.

Для реализации инфраструктуры архиватора используется большое количество сторонних библиотек.

Для управления зависимостями использовалась технология poetry, которая является оберткой над стандартным модулем pip. Она умеет автоматически настраивать зависимости для проекта так, чтобы не было конфликтов. Благодаря данному пакетному менеджеру можно гибко настраивать различные зависимости, включая тестирование, форматирование кода и т. п. В моем случае с помощью poetry я настраивал ruff, isort.

Для создания CLI интерфейса была выбрана библиотека click, которая позволяет достаточно удобно создавать команды для ввода в терминал. Она является оберткой над встроенным модулем в Python – argparse, но в отличие от нее предоставляет более насыщенный инструментарий, позволяя писать код в более декларативном стиле за счет использования паттерна декоратор. Так как мой продукт больше нацелен на DevOps специалистов, то им будет проще и рациональнее использовать именно CLI интерфейс, а не GUI.

					ТИ.990000.000 ПЗ	Лист
						16
Изм.	Лист	№ докум.	Подпись			

Реализуемый архиватор позволяет загружать данные на удаленное S3 хранилище. Для взаимодействия с ним я использовал библиотеку boto3, которая предоставляет API для работы с Amazon AWS. Данная библиотека немного посредственна с точки зрения типизации и скорости, но толковых аналогов нет, которые делали это лучше, исключая асинхронные библиотеки. Но асинхронные библиотеки теряют свой смысл, так как софт изначально работает синхронно.

Также при реализации архиватора использовался IoC Container - dishka. IoC-контейнер (Inversion of Control container) — это инструмент в программных фреймворках, который реализует принцип инверсии управления (IoC) и автоматизирует создание, конфигурацию и управление объектами (бинами) в приложении. Он отвечает за внедрение зависимостей (Dependency Injection, DI), устраняя необходимость явного создания объектов в коде и снижая связанность компонентов. Данная библиотека является одним из лучших решений для Python. Она с легкостью интегрируется с различными фреймворками и библиотеками.

При выборе архитектуры использовался подход Domain-Driven Design (DDD) — это подход к проектированию программного обеспечения, который акцентирует внимание на сложных бизнес-доменах и взаимодействии между техническими и бизнес-экспертами. Основная идея DDD заключается в том, чтобы создать модель, которая точно отражает бизнес-логику и процессы, что позволяет разработчикам и бизнес-аналитикам работать более эффективно и согласованно.

2.2. Основные этапы работы программы

В связи с использованием подхода Domain-Driven Design, появляется множество прослоек для работы, которые обеспечивают слабую связанность архитектуры. Для пояснения работы программы можно привести поток выполнения программы:

1. Ввод данных в терминал, где пользователь прописывает путь до команды, а также ее аргументы.

					ТИ.990000.000 ПЗ	Лист
						17
Изм.	Лист	№ докум.	Подпись			

2. После успешного ввода аргументов в терминал создается DTO для передачи в информации в шину сообщений.
3. Шина сообщений передает DTO нужному обработчику команд, который будет оперировать сервисами.
4. Создается экземпляр класса сервиса, который будет оперировать всеми классами, реализующими логику компрессии.
5. Далее автоматически определяется нужный класс для компрессии с помощью паттерна – фабричный метод.
6. Начинается логика работы класса.

Таким образом, обеспечивается чистота кода с точки зрения принципов SOLID. При использовании подобной архитектуры мы сможем спокойно изменять большинство функционала, не переписывая весь исходный код.

2.3. Реализация алгоритмов компрессии и декомпрессии

Gzip. Для максимальной производительности использовался готовый модуль в Python. Необходимо было написать «прослойку» для работы с этим модулем. Для этого был определен класс, который имплементирует интерфейс Compressor.

В данном классе я переопределил два метода – compress, decompress. В данном случае у меня методы ожидают доменные сущности – FileObjectEntity, CompressedFileObject. Методы по принципу работы похожи на друг друга, открывается 2 потока на файлы, считываются и кодируются или декодируются. В случае полностью удачных операций в логгер передается сообщение об удачном кодировании и декодировании.

PgZip. Как и в предыдущем случае был реализован класс – прослойка – PigzCompressor. Он служит для имплементации интерфейса Compressor.

Алгоритм работы класса PigzFile можно представить следующим образом:

					ТИ.990000.000 ПЗ	Лист
						18
Изм.	Лист	№ докум.	Подпись			

1. Инициализация и настройка параметров в конструкторе:
2. Чтение файла и разделение на блоки: метод `_read_file` открывает исходный файл в бинарном режиме.
3. Файл считывается порциями заданного размера (`blocksize`).
4. Каждой порции присваивается уникальный номер (номер чанка), что обеспечивает последующую корректную сортировку.
5. Обновляется общий размер считанных данных (`input_size`).
6. Если достигнут конец файла, фиксируется номер последнего чанка.
7. Каждая считанная порция передается в пул потоков для обработки через вызов метода `_process_chunk`.
8. Параллельное сжатие блоков:
9. `_process_chunk`:
 - a. Вызывается для каждого блока в пуле потоков.
 - b. Определяет, является ли текущий блок последним (сравнивая номер чанка с номером последнего блока).
 - c. Вызывает метод `_compress_chunk` для выполнения сжатия.
10. `_compress_chunk`:
 - a. Создается объект компрессора (`zlib.compressobj`) с использованием заданного уровня сжатия и параметров DEFLATE (например, отрицательное значение `wbits` для отсутствия заголовков).
 - b. Выполняется сжатие блока посредством метода `compress()`.
 - c. Для последнего блока применяется `flush(zlib.Z_FINISH)` для завершения сжатия, для остальных – `flush(zlib.Z_SYNC_FLUSH)`.
 - d. Результатом является сжатый блок, который вместе с его номером помещается в приоритетную очередь.
11. Настройка и запись сжатых данных в итоговый файл:
12. Настройка выходного файла:
 - a. Метод `_setup_output_file` формирует имя выходного файла (добавляя расширение ".gz") и открывает его для записи.

- b. Вызывается метод `_write_output_header`, который записывает gzip-заголовок. Заголовок включает:
 - i. Магические числа (ID).
 - ii. Метод сжатия (DEFLATE).
 - iii. Флаги (например, наличие имени исходного файла).
 - iv. Время модификации (MTIME).
 - v. Дополнительные флаги (XFL) и идентификатор операционной системы (OS).

13. Запись сжатых блоков:

- a. Метод `_write_file` работает в отдельном потоке.
- b. Из приоритетной очереди извлекаются сжатые блоки по порядку (номер чанка).
- c. Перед записью каждого блока вычисляется контрольная сумма (CRC32) исходного блока через метод `calculate_chunk_check`.
- d. Блоки записываются в выходной файл.
- e. После записи последнего блока производится вызов финализирующих операций.

14. Запись трейлера:

- a. Метод `write_file_trailer` дописывает в конец файла трейлер, который содержит:
 - i. Итоговую контрольную сумму (CRC32).
 - ii. Общий размер исходного файла (ISIZE) (с учетом модуля 2^{32}).
- b. Эти данные необходимы для проверки целостности при последующей декомпрессии.

15. Очистка ресурсов:

- a. Метод `clean_up` закрывает выходной файл, вызывает метод `_close_workers` для завершения работы пула потоков и освобождает остальные ресурсы, использованные для параллельной обработки.

FastLZ. Для начала был описан класс FastLZInterface, который будет инкапсулировать логику выбора уровня сжатия. Он выбирает уровень сжатия в зависимости от того какое количество битов будет передано для компрессии данных.

Перейдем к CompressorLevel1. Здесь есть единственный метод, который осуществляет компрессию. Алгоритм представлен ниже:

1. Инициализация исходных параметров:
2. Установка позиции в исходном буфере: Начальная позиция (source_position) устанавливается в 0, а общая длина буфера определяется как source_length = len(source).
3. Определение границ буфера. Вычисляются две границы:
 - a. source_bound — максимальная позиция, до которой можно безопасно читать 4 байта (исключаются последние 4 байта, чтобы избежать выхода за пределы буфера).
 - b. source_limit — позиция, после которой начинается финальная обработка литералов (оставшиеся до 12 байт).
4. Инициализация буфера для сжатых данных: Создаётся пустой выходной буфер (destination), а текущая позиция в нём инициализируется нулём.
5. Создание хеш-таблицы: хеш-таблица размером HASH_TABLE_SIZE инициализируется нулями. Она будет использоваться для быстрого поиска совпадений 3-байтовых последовательностей.
6. Начало обработки входного буфера
7. Сдвиг позиции: перед началом основной обработки позиция смещается на 2 байта. Это нужно для того, чтобы первые 3 байта не сравнивались сами с собой, так как минимальная длина совпадения равна 3 байтам.
8. Установка опорной точки: переменная iteration_start_position сохраняет позицию начала текущего блока, который будет либо закодирован как литералы, либо будет продолжением совпадения.

9. Основной цикл поиска совпадений. Пока позиция не достигла `source_limit`, выполняется следующий алгоритм:

10. Поиск совпадения:

- a. Чтение 4 байтов: считываются 4 байта с текущей позиции, затем оставляются только 3 младших байта (с помощью побитового И с `0xFFFFFFFF`).
- b. Вычисление хеша: на основе 3-байтовой последовательности вычисляется хеш-значение с помощью функции `calculate_hash_value`.
- c. Получение предыдущей позиции: по найденному хешу из хеш-таблицы извлекается ранее сохранённая позиция (если такая есть).
- d. Обновление хеш-таблицы: текущая позиция записывается в хеш-таблицу по вычисленному индексу, чтобы использовать её для будущих сравнений.
- e. Определение смещения: вычисляется текущая разница между текущей позицией и сохранённой позицией (`offset`). Если смещение меньше максимального значения (`MATCH_OFFSET_MAX_LEVEL1`), считываются 3 байта из найденной позиции для сравнения. Если смещение слишком велико, используется фиксированное значение (`0x1000000`).
- f. Проверка совпадения: если позиция не превысила `source_limit`, происходит переход к следующей позиции и сравнение: если текущая 3-байтовая последовательность совпадает с полученным значением, поиск прекращается.

11. Обработка найденного совпадения:

- a. Корректировка позиции: поскольку в процессе поиска позиция увеличилась, её уменьшают на 1, чтобы учесть, что совпадение обнаружено на предыдущем байте.
- b. Эмитирование литералов: если между началом итерации и текущей позицией есть несжатые данные, они отправляются на кодирование как литералы с помощью функции `emit_literal_instructions`.

					ТИ.990000.000 ПЗ	Лист
						22
Изм.	Лист	№ докум.	Подпись			

- c. Определение длины совпадения: после обнаружения совпадения происходит сравнение последовательностей, начиная с 3-го байта после найденного совпадения, до первого отличия. Это позволяет определить фактическую длину совпадения (минимум 3 байта).
 - d. Кодирование совпадения: функция `_emit_match_instructions` генерирует инструкции для совпадения.
 - e. Если длина совпадения меньше 7 байт, используется короткая инструкция (`short match`).
 - f. Если длина равна или больше 7 байт, применяется длинная инструкция (`long match`). При этом в инструкции кодируются: Длина совпадения (с поправкой, так как минимальное совпадение — 3 байта). Смещение, которое указывает, на сколько байт назад нужно обратиться в уже сжатых данных для копирования совпадающей последовательности.
 - g. Обновление позиции: Текущая позиция смещается на длину совпадения, после чего обновляются значения хешей для новых последовательностей (обработка двух следующих позиций с использованием 3 младших и 3 старших байт соответственно).
 - h. Сброс опорной точки: в конце итерации значение `iteration_start_position` обновляется на текущую позицию, чтобы следующая серия литералов начиналась именно с этой точки.
12. Обработка оставшихся данных:
13. Когда цикл завершается (достигается `source_limit`), оставшиеся в конце исходного буфера данные (литералы) кодируются с помощью функции `emit_literal_instructions` и добавляются в выходной буфер.
14. Возврат результата:
15. По завершении обработки метод `compress` возвращает итоговый сжатый буфер, содержащий комбинацию литералов и инструкций совпадения.

3. Работоспособность программного средства

Для запуска приложения необходимо преждевременно установить Docker. При помощи данной технологии производится развертывание базы данных PostgreSQL. После этого остается установить библиотеки для Python, используя команду `poetry install`. Теперь приложение готово к использованию.

Для сжатия любого файла можно использовать следующую команду: `poetry run python app/application/cli/compressing/handlers.py compress <path> -t <algorithm_name>`. При удачном выполнении команды создается файл в папке «tmp» с нужным префиксом. На рисунке 1 представлен пример сжатия бэкапа базы данных с использованием gunzip.

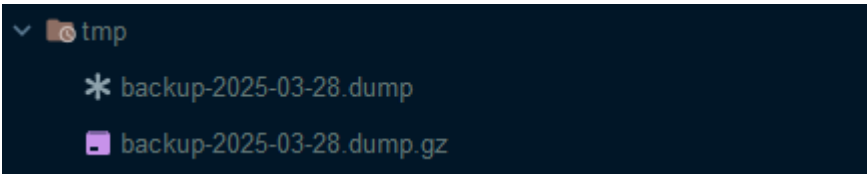


Рисунок 1 – Результат выполнения операции по сжатию данных

Изначальный размер был 1 Кб, в результате после сжатия размер стал 1 Кб. Здесь сжатие не совсем было эффективно, так как база данных не была заполнена.

Для тестирования эффективности будет использоваться рассказ «Война и мир» в формате txt. Результат сравнения размеров можно увидеть на рисунке 2.




 war_and_piece.txt	30.03.2025 10:14	Текстовый докум...	2 845 Кб
 war_and_piece.txt.fastlz	01.04.2025 18:13	Файл "FASTLZ"	1 460 Кб
 war_and_piece.txt.gz	01.04.2025 13:17	Сжатая архивная ...	801 Кб

Рисунок 2 – Сравнение эффективности алгоритмов сжатия

На рисунке 2 можно увидеть, что размер до сжатия составлял 2845 Кб, после сжатия алгоритмом FastLZ размер стал 1460 Кб, сжатие происходит в 1.94 раза. Gunzip сжал до 801 Кб, он сжал в 3.55 раза.

4. Сравнительный анализ алгоритмов

Алгоритмы Gzip, Pigz и FastLZ ориентированы на разные сценарии использования, что отражается на их производительности, степени сжатия и требованиях к ресурсам. В таблице 1 представлено сравнение производительности эталонных реализаций на Си.

Таблица 1 – сравнение производительности эталонных реализаций алгоритмов

Название алгоритма	Gzip	Pigz	FastLZ
Скорость сжатия	100–200 МБ/с	400–600 МБ/с	300–500 МБ/с
Скорость распаковки	200–300 МБ/с	200–300 МБ/с	500–700 МБ/с
Степень сжатия	50–70 % (высокая)	50–70 % (высокая)	30–50 % (ниже среднего)
Использование памяти	100–500 МБ	200–800 МБ	10–20 МБ

Алгоритм FastLZ ориентирован на скорость. В среднем его асимптотическая сложность составляет $O(n)$, где n — размер входного потока, с очень низкими константами, что обеспечивает высокую скорость работы.

Алгоритм gzip основан на методе DEFLATE, который в среднем алгоритм работает за $O(n)$, однако дополнительные этапы (например, построение деревьев Хаффмана) увеличивают константные коэффициенты.

Pgzip представляет собой параллельную реализацию gzip, что позволяет обрабатывать данные по частям (чанкам) с использованием нескольких потоков. С точки зрения алгоритмической сложности, каждый отдельный чанк обрабатывается по схеме gzip, то есть $O(chunk_size)$.

Заключение

В рамках данной работы было разработано программное средство – архиватор, реализующий функции сжатия и распаковки файлов с использованием современных технологий и алгоритмов, таких как PostgreSQL, FastLZ, gunzip, minio, fastlz, pgzip и принципов архитектурного подхода DDD.

В ходе работы были решены следующие задачи:

1. Проведен анализ алгоритмов сжатия FastLZ, gunzip, pgzip, включая их математические принципы и особенности функционирования.
2. Разработано программное для архиватора, включая алгоритмы сжатия, разжатия данных.
3. Проведен сравнительный анализ эффективности на основе результатов программной реализации.
4. Выявлены преимущества и недостатки каждого из алгоритмов с целью определения их пригодности для конкретных задач в различных областях применения.

Итогом данной курсовой работы стало создание рабочего и эффективного программного средства, способного удовлетворять современные требования к сжатию и хранению информации. Проведенное исследование и сравнительный анализ алгоритмов позволили не только выбрать оптимальные решения для конкретных задач, но и заложить основу для дальнейшего развития проекта и его интеграции с другими системами. Достигнутые результаты свидетельствуют о практической значимости выполненной работы и перспективности выбранного направления исследований.

					ТИ.990000.000 ПЗ	Лист
						26
Изм.	Лист	№ докум.	Подпись			

Перечень используемых информационных ресурсов

1. Ватолин Д., Ратушняк А., Смирнов М., Юкин В. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео. - М.: ДИАЛОГ-МИФИ, 2002. - 384 с.
2. Березкин Е. Ф. Основы теории информации и кодирования. Учебное пособие для вузов. – Москва.: ЛАНЬ, 2021. – 321 с.
3. Горев, А И; Симаков А А Обеспечение Информационной Безопасности / А Горев А И; Симаков А. – Москва: Мир, 2005. – 844 с.
4. Деундяк В.М., Маевский А.Э., Могилевская Н.С. Методы помехоустойчивой защиты данных: учебник / В.М. Деундяк, А.Э Маевский, Н.С. Могилевская. – Ростов н/Д: Издательство Южного федерального университета, 2014. – 309 с.
5. Алгоритм FastLZ // GitHub.com: сайт. URL: <https://github.com/ariya/FastLZ> (дата обращения: 15.04.2025)

					ТИ.990000.000 ПЗ	Лист
						27
Изм.	Лист	№ докум.	Подпись			

Приложение А Листинг кода

```
from pathlib import Path
from typing import TYPE_CHECKING

import click
from dishka import FromDishka
from dishka.integrations.click import setup_dishka

from app.application.cli.const import BACKUP_DIRECTORY_PATH
from app.infrastructure.uow.compression import CompressionUnitOfWork
from app.logic.bootstrap import Bootstrap
from app.logic.commands.compression import (
    CompressFileCommand,
    DecompressFileCommand,
)
from app.logic.container import container
from app.settings.logger.config import setup_logging

if TYPE_CHECKING:
    from app.logic.message_bus import MessageBus

@click.group()
@click.pass_context
def cli(context: click.Context):
    setup_logging()
    BACKUP_DIRECTORY_PATH.mkdir(exist_ok=True)
    setup_dishka(container=container, context=context,
    auto_inject=True)

@cli.command("compress")
@click.argument("src_file_path", type=click.Path(exists=True,
dir_okay=False, path_type=Path), required=True)
```

					ТИ.990000.000 ПЗ	Лист
						28
Изм.	Лист	№ докум.	Подпись			

```

@click.option(
    "-t",
    "--type_of_compression",
    type=click.Choice(["gzip", "pigz", "fastlz"]),
    default="gzip",
    help="Compression type",
)

def compress(
    src_file_path: Path, type_of_compression: str, bootstrap:
FromDishka[Bootstrap[CompressionUnitOfWork]]
) -> None:
    """
    Compress any file that user gives
    :param src_file_path: path to file to compress
    :param type_of_compression: compression type
    :param bootstrap: Bootstrap
    :return: Returns nothing
    """

    message_bus: MessageBus = bootstrap.get_messagebus()
    message_bus.handle(CompressFileCommand(src_file_path,
compress_type=type_of_compression))
    click.echo("Compress complete.")

@cli.command("decompress")
@click.argument("src_file_path", type=click.Path(exists=True,
dir_okay=False, path_type=Path), required=True)
@click.option(
    "-t",
    "--type_of_compression",
    type=click.Choice(["gzip", "pigz", "fastlz"]),
    default="gzip",
    help="Compression type",
)

def decompress(
    src_file_path: Path, type_of_compression: str, bootstrap:

```

					ТИ.990000.000 ПЗ	Лист
						29
Изм.	Лист	№ докум.	Подпись			

```

FromDishka[Bootstrap[CompressionUnitOfWork]]
) -> None:
    """
    Decompress any file that user gives
    :param src_file_path: File to decompress
    :param type_of_compression: Type of compression
    :param bootstrap: Bootstrap
    :return: Nothing
    """

    message_bus: MessageBus = bootstrap.get_messagebus()
    message_bus.handle(DecompressFileCommand(src_file_path,
compress_type=type_of_compression))
    click.echo("Decompress complete.")

if __name__ == "__main__":
    cli()

from typing import TYPE_CHECKING

import click
from dishka import FromDishka
from dishka.integrations.click import setup_dishka

from app.application.cli.const import BACKUP_DIRECTORY_PATH
from app.infrastructure.uow.compression import CompressionUnitOfWork
from app.logic.bootstrap import Bootstrap
from app.logic.commands.database import (
    CreateDatabaseBackupCommand,
    ListAllDatabasesCommand,
)
from app.logic.container import container
from app.settings.logger.config import setup_logging

if TYPE_CHECKING:

```

					ТИ.990000.000 ПЗ	Лист
						30
Изм.	Лист	№ докум.	Подпись			

```

    from app.logic.message_bus import MessageBus
@click.group()
@click.pass_context
def cli(context: click.Context):
    setup_logging()
    BACKUP_DIRECTORY_PATH.mkdir(exist_ok=True)
    setup_dishka(container=container, context=context,
auto_inject=True)

@cli.command("list")
def list_all_databases(bootstrap:
FromDishka[Bootstrap[CompressionUnitOfWork]]) -> None:
    """
    Command to list all databases.
    :return: nothing
    """
    message_bus: MessageBus = bootstrap.get_messagebus()
    message_bus.handle(ListAllDatabasesCommand())
    click.echo("Successfully listed all databases")

@cli.command("backup")
@click.argument("database_name", type=click.STRING, required=True)
def backup_database(database_name: str, bootstrap:
FromDishka[Bootstrap[CompressionUnitOfWork]]) -> None:
    """
    Command to back up a database.
    :param database_name: name of the existing database
    :param bootstrap: bootstrap instance
    :return: nothing
    """
    message_bus: MessageBus = bootstrap.get_messagebus()

    message_bus.handle(CreateDatabaseBackupCommand(database_name=database_
name))

```

					ТИ.990000.000 ПЗ	Лист
						31
Изм.	Лист	№ докум.	Подпись			

```

click.echo(f"Successfully backup the database {database_name}")

if __name__ == "__main__":
    cli()

from dataclasses import dataclass
from pathlib import Path

from app.domain.entities.base import BaseEntity
from app.domain.values.backup import CompressionType
from app.domain.values.file_objects import (
    PermissionsOfFile,
    SizeOfFile,
    TypeOfFile,
)

@dataclass(eq=False)
class FileObjectEntity(BaseEntity):
    """
    Entity that describes a file object.
    This entity has only one field: file_path.
    """
    file_path: Path

@dataclass(eq=False)
class CompressedFileObjectEntity(BaseEntity):
    """
    Entity that describes file which was compressed.
    There are several fields:
    - file_path: path to file.
    - compression_type: type of compressed file.
    """
    file_path: Path

```

					ТИ.990000.000 ПЗ	Лист
						32
Изм.	Лист	№ докум.	Подпись			


```

        compression_type: CompressionType
@dataclass(eq=False)
class FileStatistic(BaseEntity):
    """
    Entity that describes file statistics.
    There are several fields:
    - name: Name of file.
    - size: Size of file.
    - type_of_file: Type of file. It can be only directory or usual
file.
    - extension: Extension of file. If it is directory or usual file.
    - permissions: Permissions of file.
    """
    name: str
    size: SizeOfFile
    type_of_file: TypeOfFile
    extension: str
    permissions: PermissionsOfFile

from abc import ABC
from dataclasses import (
    dataclass,
    field,
)
from datetime import (
    UTC,
    datetime,
)
from typing import Any
from uuid import uuid4

@dataclass(eq=False)
class BaseEntity(ABC):
    """
    Base entity, from which any domain model should be inherited.

```

					ТИ.990000.000 ПЗ	Лист
						33
Изм.	Лист	№ докум.	Подпись			

```

"""

    oid: str = field(default_factory=lambda: str(uuid4())),
kw_only=True)

    created_at: datetime = field(default_factory=lambda:
datetime.now(UTC), kw_only=True)

    updated_at: datetime = field(default_factory=lambda:
datetime.now(UTC), kw_only=True)

    async def to_dict(
        self, exclude: set[str] | None = None, include: dict[str, Any]
| None = None
    ) -> dict[str, Any]:
        """
        Create a dictionary representation of the entity.

        exclude: set of model fields, which should be excluded from
dictionary representation.

        include: set of model fields, which should be included into
dictionary representation.
        """

        data: dict[str, Any] = vars(self)

        # For sqlalchemy
        data.pop("_sa_instance_state", None)

        # Handle exclude set
        if exclude:
            for key in exclude:
                data.pop(key, None)

        # Handle include dictionary
        if include:
            data.update(include)

```

```

        return data

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, BaseEntity):
            raise NotImplementedError
        return self.oid == other.oid

    def __hash__(self) -> int:
        return hash(self.oid)

import gzip
import logging
from pathlib import Path
from typing import (
    Final,
)

from typing_extensions import override

from app.application.cli.const import BACKUP_DIRECTORY_PATH
from app.domain.entities.file_objects import (
    CompressedFileObjectEntity,
    FileObjectEntity,
)
from app.domain.values.backup import CompressionType
from app.infrastructure.compressors.base import Compressor

logger = logging.getLogger(__name__)

CHUNK_SIZE: Final[int] = 64 * 1024 # 64KB

class GunZipCompressor(Compressor):
    @override
    def compress(self, backup: FileObjectEntity) ->
    CompressedFileObjectEntity:

```

					ТИ.990000.000 ПЗ	Лист
Изм.	Лист	№ докум.	Подпись			35

```

        source_path: Path = backup.file_path
        dest_path: Path = BACKUP_DIRECTORY_PATH / (source_path.name +
".gz")

        with Path.open(source_path, "rb") as f_in,
gzip.open(dest_path, "wb") as f_out:
            while chunk := f_in.read(CHUNK_SIZE):
                f_out.write(chunk)

        logger.debug(f"Compressed {source_path} to {dest_path}")

        return CompressedFileObjectEntity(file_path=dest_path,
compression_type=CompressionType("gzip"))

    @override
    def decompress(self, backup: CompressedFileObjectEntity) ->
FileObjectEntity:
        source_path: Path = backup.file_path
        dest_path: Path = source_path.with_suffix("")

        with gzip.open(source_path, "rb") as f_in,
Path.open(dest_path, "wb") as f_out:
            while chunk := f_in.read(CHUNK_SIZE):
                f_out.write(chunk)

        logger.debug(f"Decompressed {source_path} to {dest_path}")

        return FileObjectEntity(file_path=dest_path)

```