

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

Ростов-на-Дону

2025



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 c.

3. Герман, О. Н. Теоретико-числовые методы в криптографии / О.Н. Герман, Ю.В. Нестеренко. – М.: Академия, 2012. – 272 c.

|  |
| --- |
| **Содержание пояснительной записки** |
| Введение:  Описывается появление самых первых алгоритмов сжатия, а также актуальность использования этих методов. |
| Разделы основной части: |
| 1. В разделе "Обзор gzip" дается полное описание утилиты, включая основные принципы, алгоритмы, механизмы сжатия, а также применение и особенности использования. 2. В разделе “Обзор pigz” дается полное описание алгоритма, включая основные принципы, механизмы сжатия, а также применение и особенности использования. 3. В разделе “Обзор Fastlz” дается полное описание алгоритмов, включая основные принципы, механизмы сжатия, а также применение и особенности использования. 4. 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. 5. 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](#_Toc194402333)

[1 Обзор утилиты gzip 6](#_Toc194402334)

[1.1 Принцип работы и описание 6](#_Toc194402335)

[1.2 Достоинства и недостатки 7](#_Toc194402336)

[2. Обзор алгоритма FastLZ 8](#_Toc194402337)

[2.1 Принцип работы и описание 8](#_Toc194402338)

[2.2 Описание работы FastLZ Level 1 8](#_Toc194402341)

[2.3 Описание работы FastLZ Level 2 11](#_Toc194402342)

[3. Обзор утилиты Pigz 13](#_Toc194402343)

[3.1 Принцип работы и описание 13](#_Toc194402344)

[3.2 Достоинства и недостатки 14](#_Toc194402345)

[4 Программная реализация архиватора данных 16](#_Toc194402346)

[4.1 Выбор языка программирования, инструментов и архитектуры 16](#_Toc194402347)

[4.2 Основные этапы работы программы 17](#_Toc194402348)

[4.3 Реализация алгоритма компрессии и декомпрессии gzip 18](#_Toc194402349)

[4.4 Реализация алгоритма компрессии и декомпрессии pgzip 19](#_Toc194402350)

[4.5 Реализация алгоритма FastLZ 23](#_Toc194402351)

[5 Сравнительный анализ алгоритмов 28](#_Toc194402352)

[5.1 Сравнение эффективности сжатия и разжатия Gzip, Pigz, FastLZ 28](#_Toc194402353)

[Заключение 30](#_Toc194402354)

[Перечень используемых информационных ресурсов 31](#_Toc194402355)

[Приложение А Листинг кода 32](#_Toc194402356)

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: дамп базы данных, документы и т. п.

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 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.

В рамках данной работы ставятся следующие цели:

1. Изучить теоретические основы алгоритмов fastlz, gzip, pgzip включая их математические принципы и особенности функционирования.
2. Разработать программное обеспечение для архиватора, включая алгоритмы сжатия, разжатия данных.
3. Провести сравнительный анализ эффективности на основе результатов программной реализации.
4. Выявить преимущества и недостатки каждого из алгоритмов с целью определения их пригодности для конкретных задач в различных областях применения.

1 Обзор утилиты gzip

1.1 Принцип работы и описание

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. 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.

Говоря проще, gzip – это практическое применение алгоритма Deflate. Опишем его процесс работы. 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.

1.2 Достоинства и недостатки

Достоинства:

* + - 1. Эффективность сжатия: gunzip обеспечивает достаточно высокую степень сжатия.

1. 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.
2. 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.
3. 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:

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

2. Обзор алгоритма FastLZ

2.1 Принцип работы и описание

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.

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.



2.2 Описание работы FastLZ Level 1

У данного уровня есть некоторые определенные параметры, которые его отличают от Level 2. Распишу процесс работы кодирования ниже:

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

Рассмотрим теперь процесс декодирования. Процесс декомпрессии осуществляется «на лету» — по мере чтения сжатого потока декомпрессор последовательно обрабатывает каждую инструкцию, восстанавливая исходный (несжатый) поток данных. Рассмотрим ключевые этапы работы декомпрессора на основе представленного кода:

* 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).
  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). Дальше производится копирование блока.

2.3 Описание работы FastLZ Level 2

FastLZ Level 2 представляет собой оптимизированную версию для Level 1. Основные отличия заключаются в поддержке более длинных совпадений и расширенном диапазоне смещений, что достигается за счет дополнительных проверок и использовании гамма – кодирования для кодирования длины совпадения. За счет этого компрессия становится выше. Распишем 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.
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.

Рассмотрим теперь процесс декомпрессии (распаковки) также по этапам:

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.

3. Обзор утилиты Pigz

****3.1 Принцип работы и описание****

**Pigz** - (Parallel Implementation of GZip) – это параллельная реализация популярного формата сжатия gzip, разработанная для использования всех доступных ядер процессора. 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 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.

Алгоритм PIGZ организован вокруг параллельного сжатия файла по блокам. Рассмотрим основные этапы работы, как это реализовано.

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 модифицкации, дополнительные флаги и идентификатор операционной системы.
6. Финализация и формирование трейлера: после записи всех сжатых блоков 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.

3.2 Достоинства и недостатки

Достоинства:

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.
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:

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.
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, pigz 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.

4 Программная реализация архиватора данных

4.1 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 моего архиватора большое количество сторонних библиотек. Рассмотрим каждую библиотеку и дадим ей описание.

Для управления зависимости 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 я использовал библиотеку 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.

В моем случае мой 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 мой софт изначально работает синхронно.

Последнее, что я использовал – это 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 я использовал 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.

4.2 Основные этапы работы программы

Из-за того, что мы пишем в архитектурном стиле Domain-Driven Design, появляется множество прослоек для работы, которые обеспечивают слабую связанность архитектуры. Для пояснения работы программы приведу ниже поток выполнения программы:

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

4.3 Реализация алгоритма компрессии и декомпрессии gzip

В моем случае, для максимальной производительности я использовал готовый модуль в Python. В моем случае оставалось написать только прослойку для работы с этим модулем. Для этого я определил 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 в принципе очень похожи на друг друга, мы открываем 2 потока на файлы, считываем и кодируем или декодируем. В случае полностью удачных операций в логгер передается сообщение об удачном кодировании и декодировании.



Рисунок 1 – Класс для работы с gzip

Произведем теперь ручное тестирование gzip, насколько все получилось. В качестве примера буду использовать роман “Война и мир”.

4.4 Реализация алгоритма компрессии и декомпрессии pgzip

Как и в прошлом случае я написал класс – прослойку – PigzCompressor, который представлен на рисунке 2. Он служит для имплементации интерфейса Compressor.



Рисунок 2 – Класс для работы с pigz

Распишу ниже алгоритм работы моего класса PigzFile, который представлен на рисунке 3.

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:
   1. Вызывается для каждого блока в пуле потоков.
   2. Определяет, является ли текущий блок последним (сравнивая номер чанка с номером последнего блока).
   3. Вызывает метод \_compress\_chunk для выполнения сжатия.
10. \_compress\_chunk:
    1. Создается объект компрессора (zlib.compressobj) с использованием заданного уровня сжатия и параметров DEFLATE (например, отрицательное значение wbits для отсутствия заголовков).
    2. Выполняется сжатие блока посредством метода compress().
    3. Для последнего блока применяется flush(zlib.Z\_FINISH) для завершения сжатия, для остальных – flush(zlib.Z\_SYNC\_FLUSH).
    4. 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:
    1. Метод \_setup\_output\_file формирует имя выходного файла (добавляя расширение ".gz") и открывает его для записи.
    2. Вызывается метод \_write\_output\_header, который записывает gzip-заголовок. Заголовок включает:
       1. Магические числа (ID).
       2. Метод сжатия (DEFLATE).
       3. Флаги (например, наличие имени исходного файла).
       4. Время модификации (MTIME).
       5. Дополнительные флаги (XFL) и идентификатор операционной системы (OS).
13. Запись сжатых блоков:
    1. Метод \_write\_file работает в отдельном потоке.
    2. Из приоритетной очереди извлекаются сжатые блоки по порядку (номер чанка).
    3. Перед записью каждого блока вычисляется контрольная сумма (CRC32) исходного блока через метод calculate\_chunk\_check.
    4. Блоки записываются в выходной файл.
    5. После записи последнего блока производится вызов финализирующих операций.
14. Запись трейлера:
    1. Метод write\_file\_trailer дописывает в конец файла трейлер, который содержит:
       1. Итоговую контрольную сумму (CRC32).
       2. Общий размер исходного файла (ISIZE) (с учетом модуля 2^32).
    2. Эти данные необходимы для проверки целостности при последующей декомпрессии.
15. Очистка ресурсов:
    1. Метод clean\_up закрывает выходной файл, вызывает метод \_close\_workers для завершения работы пула потоков и освобождает остальные ресурсы, использованные для параллельной обработки.



Рисунок 3 – Реализация класса для pigz

4.5 Реализация алгоритма FastLZ

Для начала я описал 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. Данный класс представлен на рисунке 4.

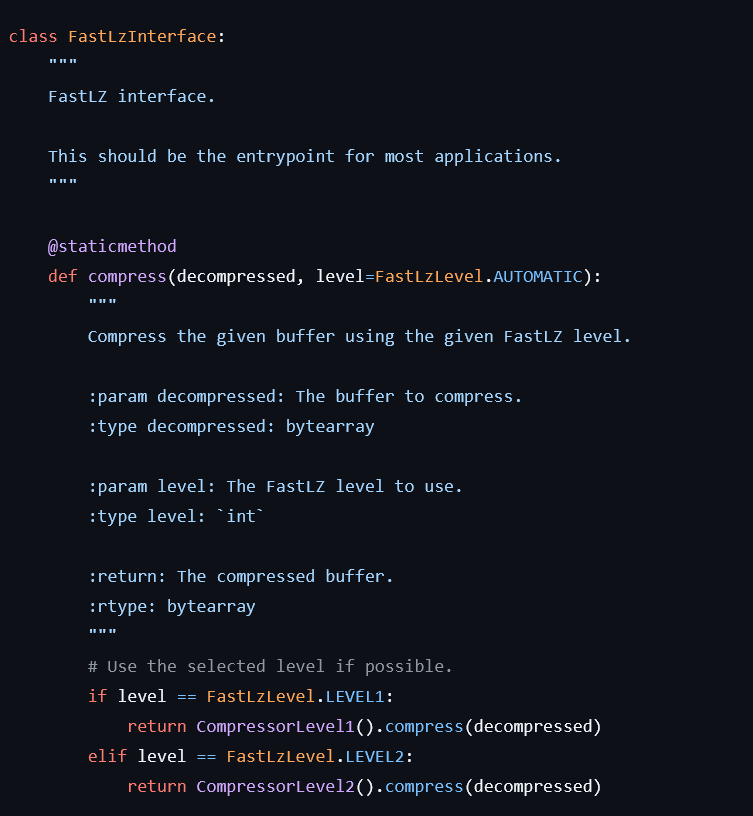


Рисунок 4 – класс FastLZInterface

Перейдем теперь к логике компрессоров различных уровней, как и что они работают. Начнем обзор с CompressorLevel1. Здесь у нас есть единственный метод, который осуществляет компрессию. Его 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:
   1. source\_bound — максимальная позиция, до которой можно безопасно читать 4 байта (исключаются последние 4 байта, чтобы избежать выхода за пределы буфера).
   2. 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:
    1. Чтение 4 байтов: считываются 4 байта с текущей позиции, затем оставляются только 3 младших байта (с помощью побитового И с 0xFFFFFF).
    2. Вычисление хеша: на основе 3-байтовой последовательности вычисляется хеш-значение с помощью функции calculate\_hash\_value.
    3. Получение предыдущей позиции: по найденному хешу из хеш-таблицы извлекается ранее сохранённая позиция (если такая есть).
    4. Обновление хеш-таблицы: текущая позиция записывается в хеш-таблицу по вычисленному индексу, чтобы использовать её для будущих сравнений.
    5. 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).
    6. 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:
    1. Корректировка позиции: поскольку в процессе поиска позиция увеличилась, её уменьшают на 1, чтобы учесть, что совпадение обнаружено на предыдущем байте.
    2. Эмитирование литералов: если между началом итерации и текущей позицией есть несжатые данные, они отправляются на кодирование как литералы с помощью функции emit\_literal\_instructions.
    3. 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).
    4. Кодирование совпадения: функция \_emit\_match\_instructions генерирует инструкции для совпадения.
    5. Если длина совпадения меньше 7 байт, используется короткая инструкция (short match).
    6. 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.
    7. 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).
    8. 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.

5 Сравнительный анализ алгоритмов

5.1 Сравнение эффективности сжатия и разжатия Gzip, Pigz, FastLZ

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

|  |  |  |  |
| --- | --- | --- | --- |
| Название алгоритма | Gzip | Pigz | FastLZ |
| Скорость сжатия | 100–200 МБ/c | 400–600 МБ/c | 300–500 МБ/c |
| Скорость распаковки | 200–300 МБ/c | 200–300 МБ/c | 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 |

Таблица 1 – сравнение производительности эталонных реализаций алгоримтов

Алгоритм FastLZ ориентирован на скорость и использует относительно простую схему поиска совпадений в данных. 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 объединяет алгоритм LZ77 и кодирование Хаффмана. В среднем алгоритм работает за *O(n),* однако дополнительные этапы (например, построение деревьев Хаффмана) увеличивают константные коэффициенты. Это делает gzip медленнее FastLZ, особенно на больших объемах данных.

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).* В сумме это даёт аналогичную сложность, но за счет параллелизма общее время обработки существенно сокращается на многоядерных системах.

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. Изучена актуальная литература по технологиям сжатия данных, базам данных, облачному хранению и современным архитектурным подходам.
2. Проведен анализ алгоритмов сжатия (FastLZ, gunzip, pgzip) с оценкой их вычислительной сложности, эффективности и особенностей реализации.
3. Проанализированы возможности применения PostgreSQL для хранения метаданных и обеспечения надежного управления данными.
4. Изучена интеграция с minio для реализации функциональности облачного хранения.
5. Разработана архитектура программного средства на основе принципов DDD, что обеспечило модульность, масштабируемость и удобство сопровождения проекта.
6. Реализованы и протестированы ключевые компоненты архиватора, продемонстрировавшие высокую скорость обработки и надежность при сжатии и распаковке данных.

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.

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 c.
3. Горев, А И; Симаков А А Обеспечение Информационной Безопасности / А Горев А И; Симаков А. – Москва: Мир, 2005. – 844 c.
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.

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)  
@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: 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:  
 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))  
 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  
 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.  
 """  
  
 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:  
 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)

"""  
Functions and classes to speed up compression of files by utilizing  
multiple cores on a system.  
"""  
  
import gzip  
import logging  
import os  
import sys  
import time  
import zlib  
from multiprocessing.dummy import Pool  
from pathlib import Path  
from queue import PriorityQueue  
from threading import (  
 Lock,  
 Thread,  
)  
  
from typing\_extensions import override  
  
from app.domain.entities.file\_objects import (  
 CompressedFileObjectEntity,  
 FileObjectEntity,  
)  
from app.domain.values.backup import CompressionType  
from app.infrastructure.compressors.base import Compressor  
from app.infrastructure.compressors.gunzip import CHUNK\_SIZE  
  
CPU\_COUNT = os.cpu\_count()  
DEFAULT\_BLOCK\_SIZE\_KB = 128  
  
logger = logging.getLogger(\_\_name\_\_)  
  
# 1 is fastest but worst, 9 is slowest but best  
GZIP\_COMPRESS\_OPTIONS = list(range(1, 9 + 1))  
\_COMPRESS\_LEVEL\_BEST = max(GZIP\_COMPRESS\_OPTIONS)  
  
# FLG bits  
FTEXT = 0x1  
FHCRC = 0x2  
FEXTRA = 0x4  
FNAME = 0x8  
FCOMMENT = 0x10  
  
  
class PigzFile: # pylint: disable=too-many-instance-attributes  
 """Class to implement Pigz functionality in Python"""  
  
 def \_\_init\_\_(  
 self,  
 compression\_target,  
 compresslevel=\_COMPRESS\_LEVEL\_BEST,  
 blocksize=DEFAULT\_BLOCK\_SIZE\_KB,  
 workers=CPU\_COUNT,  
 ):  
 """  
 Take in a file or directory and gzip using multiple system cores.  
 """  
 self.compression\_target = Path(compression\_target)  
 self.compression\_level = compresslevel  
 self.blocksize = blocksize \* 1000  
 self.workers = workers  
  
 self.output\_file = None  
 self.output\_filename = None  
  
 # This is how we know if we're done reading, compressing, & writing the file  
 self.\_last\_chunk = -1  
 self.\_last\_chunk\_lock = Lock()  
 # This is calculated as data is written out  
 self.checksum = 0  
 # This is calculated as data is read in  
 self.input\_size = 0  
  
 self.chunk\_queue = PriorityQueue()  
  
 if Path(compression\_target).is\_dir():  
 raise NotImplementedError  
 if not Path(compression\_target).exists():  
 raise FileNotFoundError  
  
 # Setup the system threads for compression  
 self.pool = Pool(processes=self.workers)  
 # Setup read thread  
 self.read\_thread = Thread(target=self.\_read\_file)  
 # Setup write thread  
 self.write\_thread = Thread(target=self.\_write\_file)  
  
 def process\_compression\_target(self):  
 """  
 Setup output file.  
 Start read and write threads.  
 Join to write thread.  
 """  
 self.\_setup\_output\_file()  
  
 # Start the write thread first so it's ready to accept data  
 self.write\_thread.start()  
 # Start the read thread  
 self.read\_thread.start()  
  
 # Block until writing is complete  
 # This prevents us from returning prior to the work being done  
 self.write\_thread.join()  
  
 def \_set\_output\_filename(self):  
 """  
 Set the output filename based on the input filename  
 """  
 base = Path(self.compression\_target).name  
 self.output\_filename = base + ".gz"  
  
 def \_write\_output\_header(self):  
 """  
 Write gzip header to file  
 See RFC documentation: http://www.zlib.org/rfc-gzip.html#header-trailer  
 """  
 self.\_write\_header\_id()  
 self.\_write\_header\_cm()  
  
 # We must first figure out if we can write out the filename before writing FLG  
 fname = self.\_determine\_fname(self.compression\_target)  
 flags = 0x0  
 if fname:  
 flags = flags | FNAME  
  
 self.\_write\_header\_flg(flags)  
  
 self.\_write\_header\_mtime()  
 self.\_write\_header\_xfl()  
 self.\_write\_header\_os()  
  
 # After this point, content of flags (FLG) determines what (if anything)  
 # we write to header  
 if flags & FNAME:  
 # Write the FNAME  
 self.output\_file.write(fname)  
  
 def \_write\_header\_id(self):  
 """  
 Write ID (IDentification) ID1, then ID2 to file header  
 These denote the file as being gzip format  
 """  
 self.output\_file.write((0x1F).to\_bytes(1, sys.byteorder))  
 self.output\_file.write((0x8B).to\_bytes(1, sys.byteorder))  
  
 def \_write\_header\_cm(self):  
 """Write the CM (compression method) to file header"""  
 self.output\_file.write((8).to\_bytes(1, sys.byteorder))  
  
 def \_write\_header\_flg(self, flags):  
 """Write FLG (FLaGs)"""  
 self.output\_file.write((flags).to\_bytes(1, sys.byteorder))  
  
 def \_write\_header\_mtime(self):  
 """Write MTIME (Modification time)"""  
 mtime = self.\_determine\_mtime()  
 self.output\_file.write((mtime).to\_bytes(4, sys.byteorder))  
  
 def \_write\_header\_xfl(self):  
 """Write XFL (eXtra FLags)"""  
 extra\_flags = self.\_determine\_extra\_flags(self.compression\_level)  
 self.output\_file.write((extra\_flags).to\_bytes(1, sys.byteorder))  
  
 def \_write\_header\_os(self):  
 """Write OS"""  
 os\_number = self.\_determine\_operating\_system()  
 self.output\_file.write((os\_number).to\_bytes(1, sys.byteorder))  
  
 def \_setup\_output\_file(self):  
 """  
 Setup the output file  
 """  
 self.\_set\_output\_filename()  
 full\_path = Path(self.compression\_target.parent, self.output\_filename)  
 self.output\_file = Path.open(full\_path, "wb")  
 self.\_write\_output\_header()  
  
 def \_determine\_mtime(self):  
 """  
 Determine MTIME to write out in Unix format (seconds since Unix epoch).  
 From http://www.zlib.org/rfc-gzip.html#header-trailer:  
 If the compressed data did not come from a file, MTIME is set to the time at  
 which compression started.  
 MTIME = 0 means no time stamp is available.  
 """  
 try:  
 return int(Path.stat(self.compression\_target).st\_mtime)  
 except Exception: # pylint: disable=broad-except  
 return int(time.time())  
  
 @staticmethod  
 def \_determine\_extra\_flags(compression\_level):  
 """  
 Determine the XFL or eXtra FLags value based on compression level.  
 Note this is copied from the pigz implementation.  
 """  
 return 2 if compression\_level >= 9 else 4 if compression\_level == 1 else 0  
  
 @staticmethod  
 def \_determine\_operating\_system():  
 """  
 Return appropriate number based on OS format.  
 0 - FAT filesystem (MS-DOS, OS/2, NT/Win32)  
 1 - Amiga  
 2 - VMS (or OpenVMS)  
 3 - Unix  
 4 - VM/CMS  
 5 - Atari TOS  
 6 - HPFS filesystem (OS/2, NT)  
 7 - Macintosh  
 8 - Z-System  
 9 - CP/M  
 10 - TOPS-20  
 11 - NTFS filesystem (NT)  
 12 - QDOS  
 13 - Acorn RISCOS  
 255 - unknown  
 """  
 if sys.platform.startswith(("freebsd", "linux", "aix", "darwin")):  
 return 3  
 if sys.platform.startswith("win32"):  
 return 0  
  
 return 255  
  
 @staticmethod  
 def \_determine\_fname(input\_filename):  
 """  
 Determine the FNAME (filename) of the source file to the output  
 """  
 try:  
 # RFC 1952 requires the FNAME field to be Latin-1. Do not  
 # include filenames that cannot be represented that way.  
 fname = Path(input\_filename).name  
 if not isinstance(fname, bytes):  
 fname = fname.encode("latin-1")  
 if fname.endswith(b".gz"):  
 fname = fname[:-3]  
 # Terminate with zero byte  
 fname += b"\0"  
 except UnicodeEncodeError:  
 fname = b""  
  
 return fname  
  
 def \_read\_file(self):  
 """  
 Read {filename} in {blocksize} chunks.  
 This method is run on the read thread.  
 """  
 # Initialize this to 0 so our increment sets first chunk to 1  
 chunk\_num = 0  
 with Path.open(self.compression\_target, "rb") as input\_file:  
 while True:  
 chunk = input\_file.read(self.blocksize)  
 # Break out of the loop if we didn't read anything  
 if not chunk:  
 with self.\_last\_chunk\_lock:  
 self.\_last\_chunk = chunk\_num  
 break  
  
 self.input\_size += len(chunk)  
 chunk\_num += 1  
 # Apply this chunk to the pool  
 self.pool.apply\_async(self.\_process\_chunk, (chunk\_num, chunk))  
  
 def \_process\_chunk(self, chunk\_num: int, chunk: bytes):  
 """  
 Overall method to handle the chunk and pass it back to the write thread.  
 This method is run on the pool.  
 """  
 with self.\_last\_chunk\_lock:  
 last\_chunk = chunk\_num == self.\_last\_chunk  
 compressed\_chunk = self.\_compress\_chunk(chunk, last\_chunk)  
 self.chunk\_queue.put((chunk\_num, chunk, compressed\_chunk))  
  
 def \_compress\_chunk(self, chunk: bytes, is\_last\_chunk: bool):  
 """  
 Compress the chunk.  
 """  
 compressor = zlib.compressobj(  
 level=self.compression\_level,  
 method=zlib.DEFLATED,  
 wbits=-zlib.MAX\_WBITS,  
 memLevel=zlib.DEF\_MEM\_LEVEL,  
 strategy=zlib.Z\_DEFAULT\_STRATEGY,  
 )  
 compressed\_data = compressor.compress(chunk)  
 if is\_last\_chunk:  
 compressed\_data += compressor.flush(zlib.Z\_FINISH)  
 else:  
 compressed\_data += compressor.flush(zlib.Z\_SYNC\_FLUSH)  
  
 return compressed\_data  
  
 def \_write\_file(self):  
 """  
 Write compressed data to disk.  
 Read chunks off of the priority queue.  
 Priority is the chunk number, so we can keep track of which chunk to get next.  
 This is run from the write thread.  
 """  
 next\_chunk\_num = 1  
 while True:  
 if not self.chunk\_queue.empty():  
 chunk\_num, chunk, compressed\_chunk = self.chunk\_queue.get()  
  
 if chunk\_num != next\_chunk\_num:  
 # If this isn't the next chunk we're looking for,  
 # place it back on the queue and sleep  
 self.chunk\_queue.put((chunk\_num, chunk, compressed\_chunk))  
 time.sleep(0.5)  
 else:  
 # Calculate running checksum  
 self.calculate\_chunk\_check(chunk)  
 # Write chunk to file, advance next chunk we're looking for  
 self.output\_file.write(compressed\_chunk)  
 # If this was the last chunk,  
 # we can break the loop and close the file  
 if chunk\_num == self.\_last\_chunk:  
 break  
 next\_chunk\_num += 1  
 else:  
 # If the queue is empty, we're likely waiting for data.  
 time.sleep(0.5)  
 # Loop breaks out if we've received the final chunk  
 self.clean\_up()  
  
 def calculate\_chunk\_check(self, chunk: bytes):  
 """  
 Calculate the check value for the chunk.  
 """  
 self.checksum = zlib.crc32(chunk, self.checksum)  
  
 def clean\_up(self):  
 """  
 Close the output file.  
 Clean up the processing pool.  
 """  
 self.write\_file\_trailer()  
  
 # Flush internal buffers  
 self.output\_file.flush()  
 self.output\_file.close()  
  
 self.\_close\_workers()  
  
 def write\_file\_trailer(self):  
 """  
 Write the trailer for the compressed data.  
 """  
 # Write CRC32  
 self.output\_file.write((self.checksum).to\_bytes(4, sys.byteorder))  
 # Write ISIZE (Input SIZE)  
 # This contains the size of the original (uncompressed) input data modulo 2^32.  
 self.output\_file.write((self.input\_size & 0xFFFFFFFF).to\_bytes(4, sys.byteorder))  
  
 def \_close\_workers(self):  
 """  
 Close compression thread pool.  
 """  
 self.pool.close()  
 self.pool.join()  
  
  
def compress\_file(  
 source\_file,  
 compresslevel=\_COMPRESS\_LEVEL\_BEST,  
 blocksize=DEFAULT\_BLOCK\_SIZE\_KB,  
 workers=CPU\_COUNT,  
):  
 """Helper function to call underlying class and compression method"""  
 pigz\_file = PigzFile(source\_file, compresslevel, blocksize, workers)  
 pigz\_file.process\_compression\_target()  
  
  
class PigzCompressor(Compressor):  
 @override  
 def compress(self, backup: FileObjectEntity) -> CompressedFileObjectEntity:  
 compress\_file(backup.file\_path)  
  
 logger.debug(f"Compressed to {backup.file\_path}")  
  
 return CompressedFileObjectEntity(  
 backup.file\_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)