Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

«Исследование подходов к созданию высокопроизводительного, масштабируемого сервиса для дедупликации данных в хранилище»

по дисциплине «Системы анализа больших данных»

Выполнила студентка
гр. 5140904/30201 Ковалёва О. А.

Руководитель Ковалев А. Д.

«___»_____ 2024 г.

Санкт-Петербург
2024

Содержание

Введение

Современные тенденции в области хранения данных направлены на создание распределенных хранилищ, которые обеспечивают высокую надежность, хорошую масштабируемость, безопасность и производительность, а также позволяют эффективно хранить и обрабатывать колоссальный объем входных данных.

Одними из важнейших аспектов хранения данных являются стоимость хранения и простота управления данными. Для неструктурированных данных или данных, которые представляются в виде последовательности байт на диске, для оптимального хранения и улучшения управляемости используются подходы дедупликации. Дедупликация подразумевает физическое хранение на носителе информации (жестком диске) только уникальных блоков данных.

В рамках данной работы разрабатывается прототип решения по дедупликации данных на локальном хранилище.

Постановка задачи

Цель работы заключается в реализации системы оптимального хранения данных за счет использования подхода дедупликации данных и проведении тестирования для измерения производительности созданного прототипа.

Задачи:

- Изучить подходы к оптимизации хранения данных в традиционных базах данных;
- Выбрать стек технологий, необходимый для создания прототипа системы дедупликации;
- Разработать систему оптимального хранения данных на диске;
- Провести нагрузочное тестирование и установить зависимость скорости чтения/записи данных в локальное хранилище от размера сегмента;
- Установить оптимальный размер сегмента блока данных, при котором наблюдается:
    - максимальная скорость записи
    - максимальная скорость чтения
- Установить зависимость процента ошибки восстановления данных от использованной hash-функции.

Программная реализация

Стек технологий

Для разработки прототипа системы хранения используется Python версии 3.12.0, так как данный язык, благодаря обширному набору встроенных библиотек, хорошо подходит для создания различных программных решений. Для хранения хеш-значений сегментов данных в работе используется база данных PostgreSQL 16 [1] – одна из наиболее распространенных реляционных баз данных. Работа с PostgreSQL осуществляется посредством Python-библиотеки psycopg2 [2].

Структура программы

Главным классом программы является ***класс хранилища StorageSystem***, реализующий запись (дедупликацию), чтение (восстановление) данных и взаимодействие с базой данных.

StorageSystem содержит следующие поля:

- storage_size – размер файла для хранения сегментов данных;
- segment_size – размер сегмента чтения данных;
- id_size – размер сегмента данных, записываемых в дедуплицированный файл;
- hash_fun – функция хеширования;
- connection и cursor – объект, хранящий соединение с базой данных, и объект для взаимодействия с ней;
- cur_storage_file – текущий файл хранилища;
- cur_storage_pos – текущая позиция в файле хранилища, измеряемая в сегментах.

Параметры системы по умолчанию задаются глобальными переменными, все размеры задаются в байтах:

```
ID_SIZE = 3
SEGMENT_SIZE = 15
STORAGE_SIZE= 5000
HASH_FUN = MD5
```

Также система хранения может быть запущена с ручными настройками, прописанными при создании класса в функции main при установке флага manual.

Для ***хранения сегментов*** используется папка в проекте решения, содержащая файлы фиксированной длины (storage_size) с записанными сегментами данных. Новые файлы хранения генерируются при полном заполнении предыдущего файла. При записи нового файла система всегда открывает последний незаполненный файл, если такой имеется.

Таблица hash_table в базе данных содержит следующие поля:

- id – идентификатор записи, являющийся первичным ключом;
- hash – хеш-значение сегмента;
- file_name – название файла, в котором записан сегмент;
- position – позиция сегмента, относительно начала файла;
- rep_num – число повторений сегмента;
- cut – вспомогательное поле для сегментов неполной длины.

Сжатый файл представляет собой бинарный файл, содержащий последовательность идентификаторов записей таблицы хеш-значений, соответствующих исходным сегментам файла.

Помимо основных методов deduplicate_file(self, file) и duplicate_file(self, file_name) класс StorageSysytem содержит вспомогательные методы:

- create_table(self) – создание таблицы хеш-значений при ее отсутствии, вызывается при создании экземпляра класса;
- free_db(cursor, connection) – статик-метод для удаления таблицы и записанных в хранилище данных;
- generate_storage_file(self) – получение имени нового файла хранилища с использованием текущей даты и времени;
- find_latest_storage_file(self) – поиск в папке хранения сегментов последнего созданного файла;
- get_deduplicated_file_name(self, file), get_duplicated_file_name(file) – получение имени файла, хранящегося в системе и обратная операция при восстановлении файла;
- get_hash(self, bytes) – получение хеш-значения сегмента байт.

Алгоритм дедупликации файла

Полученный на вход файл считывается в цикле сегментами размера segment_size. Алгоритм обработки каждого сегмента:

1. Вычисляется хеш-значение сегмента.
2. Производится запрос в базу данных на получение записи с хеш-значением текущего сегмента.
3. Если записи с хеш-значением текущего сегмента в таблице нет, производится проверка допустимости записи в текущий файл хранилища. Если размер файла превышен, открывается новый файл хранения. Если размер сегмента оказывается меньше, чем размер сегмента чтения, сегмент дополняется до полной длины незначащими нулями. Сегмент записывается в файл хранения и выполняется запрос на добавление новой записи в таблицу. Иначе, если возвращается запись с соответствующим хеш-значением, – выполняется запрос на обновление числа повторений сегмента.
4. В сжатый файл записывается идентификатор только что записанного или повторно используемого сегмента.

```
def deduplicate_file(self, file):
    # check if file exists
    if not os.path.exists(DATA_FOLDER + "/" + file):
        print("ERROR: File doesn't exists.")
        exit(-1)

    print(f"Deduplicate file: {file}")
    deduplacated_file_name = self.get_deduplicated_file_name(file)
    with open(DATA_FOLDER + "/" + file, "rb") as input, \
         open(DEDUPLICATED_FOLDER + "/" + deduplacated_file_name, "wb") as output:
        seg = input.read(self.seg_size)

        while seg:
            hash = self.get_hash(seg)
            cut = '0'
            self.cursor.execute("SELECT * FROM hash_table WHERE hash = %s;", (hash, ))
            db_str = self.cursor.fetchone()

            if db_str is None: # does not exist in db
                # check if there is enough place in current file
                if self.cur_storage_pos * self.seg_size + self.seg_size > self.storage_size:
```

```
                    self.generate_storage_file()

                # write segment to storage
                storage_name = STORAGE_FOLDER + "/" + self.cur_storage_file
                with open(storage_name, "ab") as storage:
                    if len(seg) < self.seg_size:
                        cut = str(self.seg_size - len(seg))
                        zeros = bytes([0x00 for _ in range(self.seg_size - len(seg))])
                        seg = zeros + seg
                    storage.write(seg)

                # add hash to hash_table
                self.cursor.execute("""INSERT INTO hash_table (hash, file_name, position,
rep_num, cut) VALUES(%s,%s,%s,%s,%s) RETURNING id;""",
                            (hash, self.cur_storage_file, self.cur_storage_pos, 1, cut))
                self.cur_storage_pos += 1
                self.conn.commit()
                id = self.cursor.fetchone()[0]
            else: # increment repetition num
                id = db_str[0]
                rep_num = db_str[4]
                self.cursor.execute("""UPDATE hash_table SET rep_num = %s WHERE id =
%s;""", (rep_num + 1, id))
                self.conn.commit()

            output.write(id.to_bytes(self.id_size, byteorder='big'))
            seg = input.read(self.seg_size)

    return deduplacated_file_name
```

Алгоритм восстановления файла

Полученный на вход сжатый файл считывается в цикле сегментами размера id_size. Алгоритм обработки каждого сегмента:

1. Сегмент преобразуется в целое число – идентификатор.
2. Производится запрос в базу данных на получение записи с текущим идентификатором.
3. Из возвращенной записи получается имя файла и позиция сегмента в нем.
4. Чтение сегмента происходит со смещением указателя чтения в нужную позицию и пропуском незначащих нулей, если это необходимо.
5. Полученный сегмент записывается в восстановленный файл.

```
def duplicate_file(self, file_name):
        # check if file exists
        if not os.path.exists(DEDUPLICATED_FOLDER + "/" + file_name):
            print("ERROR: File doesn't exists.")
            exit(-1)
        # check if table exists
        self.cursor.execute("SELECT EXISTS(SELECT * from information_schema.tables where
table_name='hash_table')")
        if not self.cursor.fetchone()[0]:
            print("ERROR: Table doesn't exists.")
            exit(-1)

        print(f"Duplicate file: {file_name}")
        with open(DEDUPLICATED_FOLDER + "/" + file_name, "rb") as file, \
             open(DUPLICATED_FOLDER + "/" + self.get_duplicated_file_name(file_name),
"wb") as output:
            id_bytes = file.read(self.id_size)
            while id_bytes:
                id = int.from_bytes(id_bytes, byteorder='big')

                # find id in database
                self.cursor.execute("SELECT * FROM hash_table WHERE id = %s;", (id, ))
                db_str = self.cursor.fetchone()
                if db_str:
                    # read segment from storage
                    with open(STORAGE_FOLDER + f"/{db_str[2]}", "rb") as f:
                        f.seek(db_str[3] * self.seg_size + int(db_str[5]))
                        seg_bytes = f.read(self.seg_size)
                    output.write(seg_bytes)
                else:
                    print("ERROR: No such file in database.")
                    exit(-1)
                id_bytes = file.read(self.id_size)
```

Тестирование прототипа и результаты исследований

Для тестирования системы хранения используются изображения формата JPG и размера от 15 килобайт до 1 мегабайта, а также текстовый файл формата TXT и размера 2.6 мегабайт. Для запуска тестов необходимо установить соответствующие флаги в функции main: storage_size_test, seg_size_test, hash_fun_test.

Тесты строятся по общему шаблону:

1. Одна итерация теста – последовательная запись и чтение файла, выполняется для каждого значения параметра, зависимость от которого исследуется;
2. Для каждого значения параметра результаты фиксируются для каждого файла из списка тестируемых файлов;
3. При вычислении времени результаты тестирования усредняются по нескольким прогонам тестов.

Для проверки корректности восстановления реализована функция compare_files(file), которая проводит сравнение полученного файла с исходным на каждой итерации.

Исследование зависимостей от размера файлов хранения

Исследуем зависимость скорости чтения/записи данных от размера файлов хранения. Используем 10 изображений размером от 15 килобайт до 1 мегабайта. Зафиксируем параметры системы: id_size = 3, seg_size = 100, hash_fun = MD5. Будем использовать следующие значения размеров файлов: [100, 500, 1000, 2000, 5000, 10000, 15000, 25000]. Усредним значения на пяти прогонах.

Построим график зависимости чтения/записи от размера файлов хранения для каждого тестируемого файла (рис. 1 – рис. 2) и с усреднением по файлам (рис. 1* – рис. 2*).

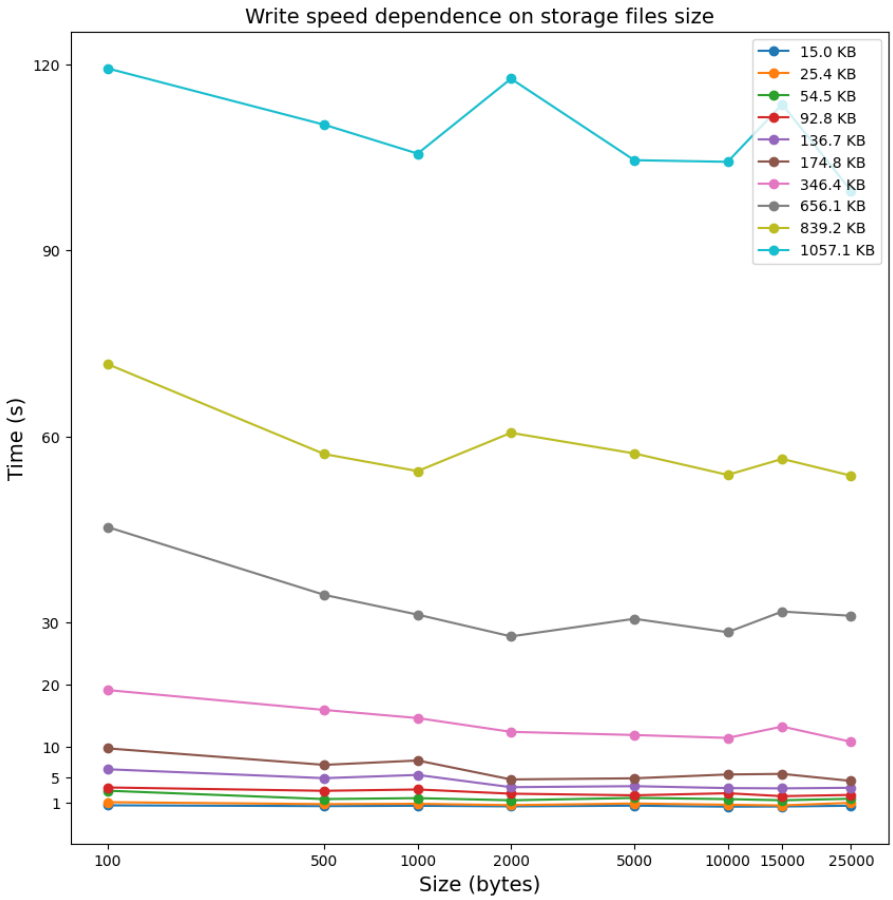

Рисунок 1. Зависимость скорости записи от размера файлов хранения

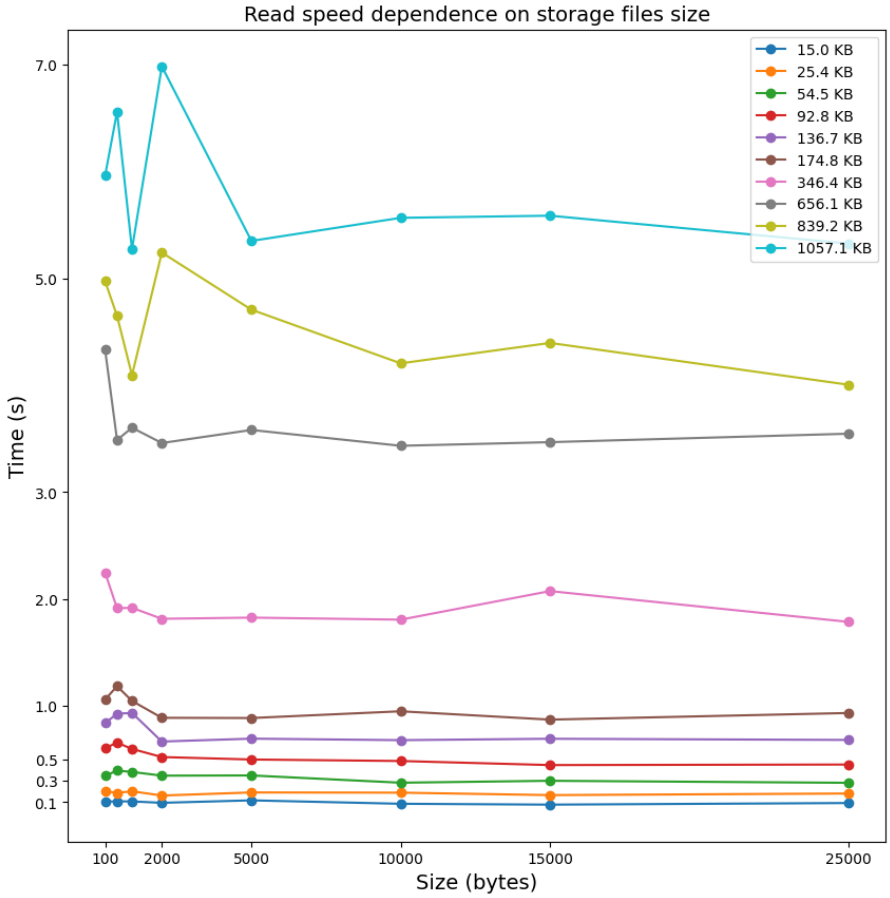

Рисунок 2. Зависимость скорости чтения от размера файлов хранения

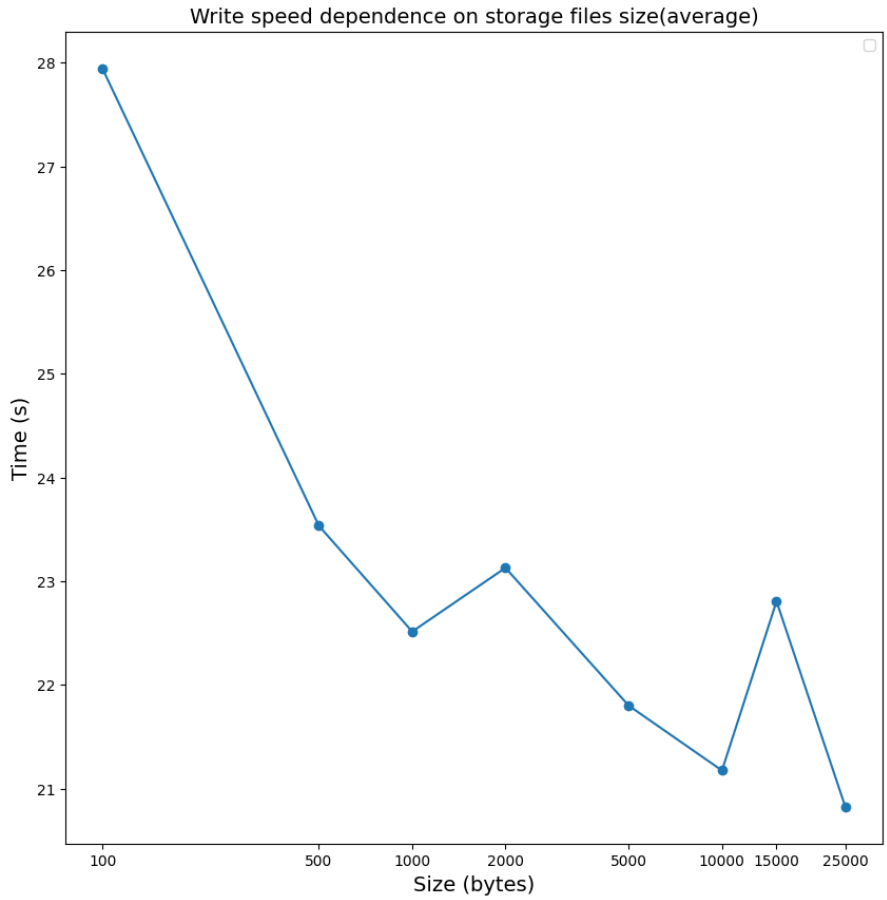

Рисунок 1*. Зависимость скорости записи от размера файлов хранения

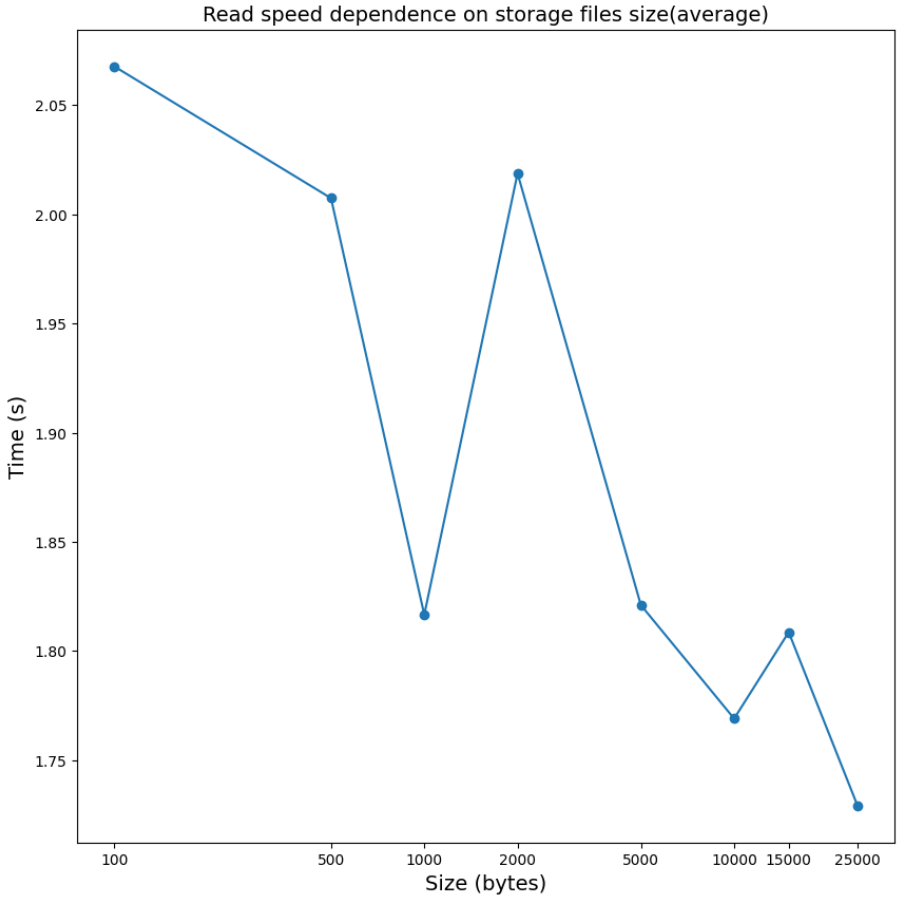

Рисунок 2*. Зависимость скорости чтения от размера файлов хранения

Заметим, что в основном лучшая скорость наблюдается на больших значениях размера, но некоторые минимумы наблюдаются и при меньших значениях. Полученные графики можно рассматривать как средство достижения оптимальной организации хранилища.

Зафиксируем значение параметра системы storage_size = 5000 для последующих тестов.

Исследование зависимостей от размера сегмента

Изучим зависимость скорости чтения/записи, процента переиспользуемых сегментов и коэффициента сжатия от размера сегмента. Используем для этого пять первых по величине изображений и самое большое, а также текстовый файл размера 2.6 мегабайт. Зафиксируем параметры системы: id_size = 3, storage_size = 5000, hash_fun = MD5. Будем использовать следующие значения размеров сегментов: [4, 10, 25, 50, 100, 250]. Усредним значения на 3 прогонах.

Процент повторно используемых сегментов – отношение числа незаписанных в хранилище сегментов файла к числу сегментов в файле. Коэффициент сжатия вычисляется по формуле:

$$coef = \frac{size(\text{сжатый файл}) + size(\text{записанных в хранилище сегментов})}{size(\text{исходный файл})}$$

По полученным графикам (рис. 3 – рис. 4) видим, что скорость чтения/записи уменьшается с увеличением размера сегмента. Для текстового файла время записи при сегменте в 4 байта меньше, чем при сегменте в 10 байт. Это может быть связано с переиспользованием большого числа фрагментов. Для подкрепления этого предположения посмотрим на графики процента переиспользуемых фрагментов и коэффициента сжатия (рис. 5 – рис. 6).

Из графиков видно, что на небольших изображениях процент переиспользуемых сегментов мал. Больше всего повторяющихся сегментов у файлов по 25 и 54 килобайт. У этих изображений (рис. 7) однотонный или почти однотонный фон. В сравнении с остальными файлами, эти картинки содержат наименьшее количество цветов.

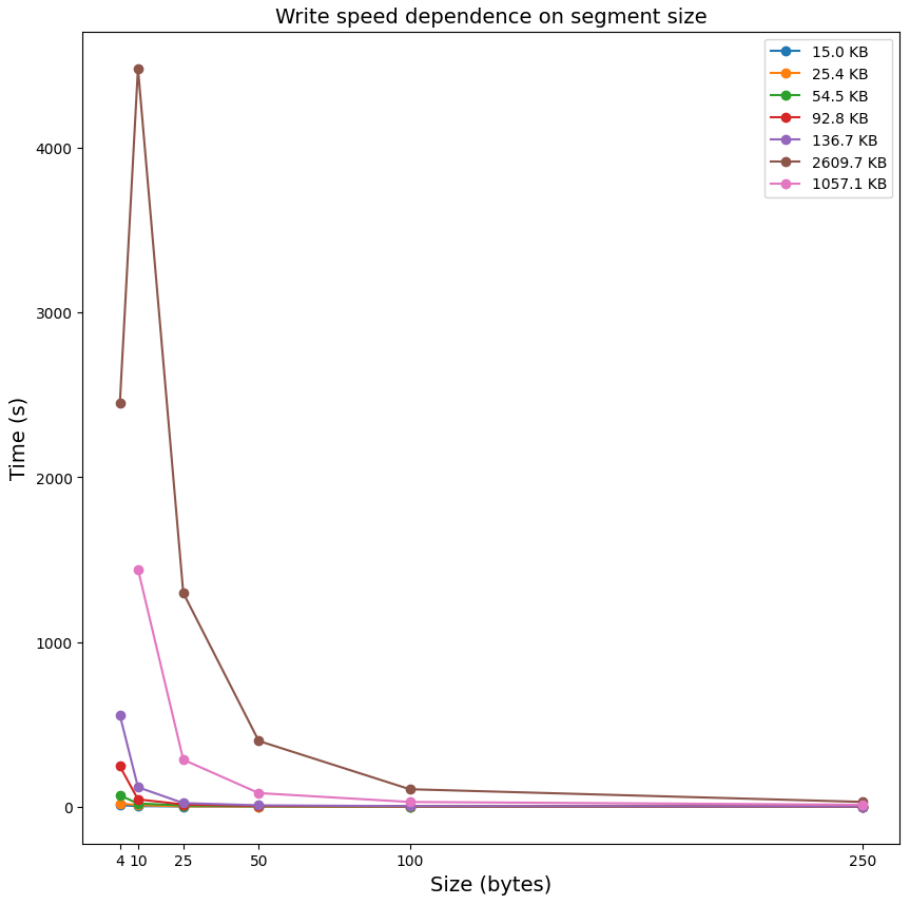

Рисунок 3. Зависимость скорости записи от размера сегмента

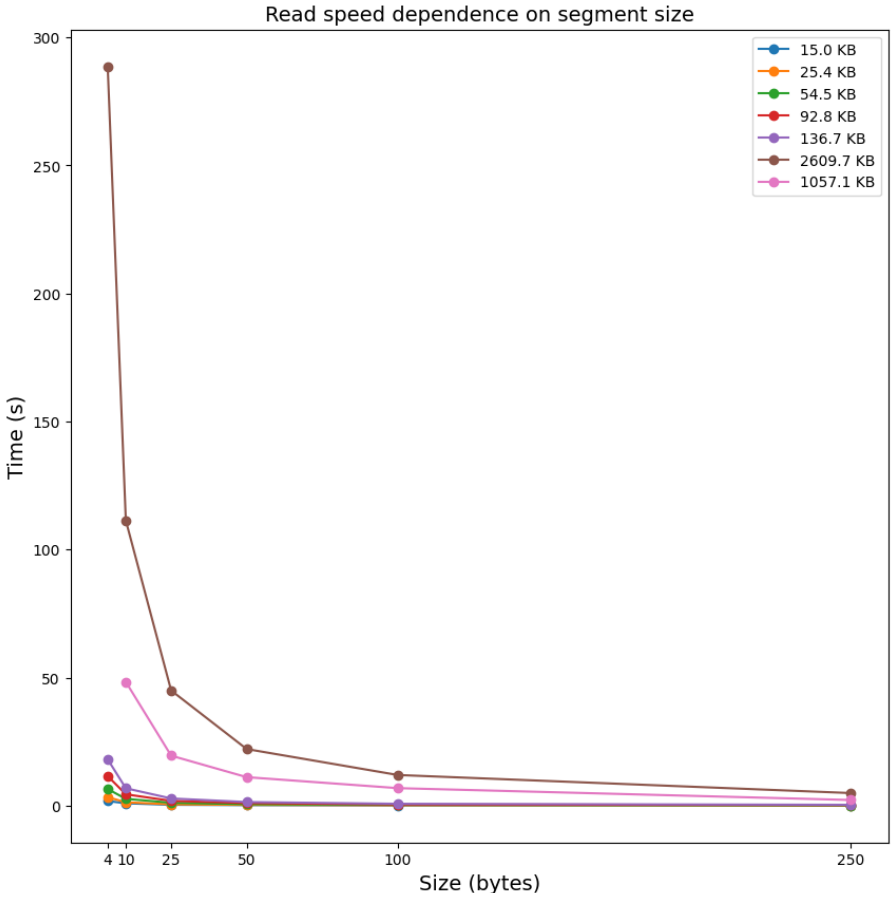

Рисунок 4. Зависимость скорости чтения от размера сегмента

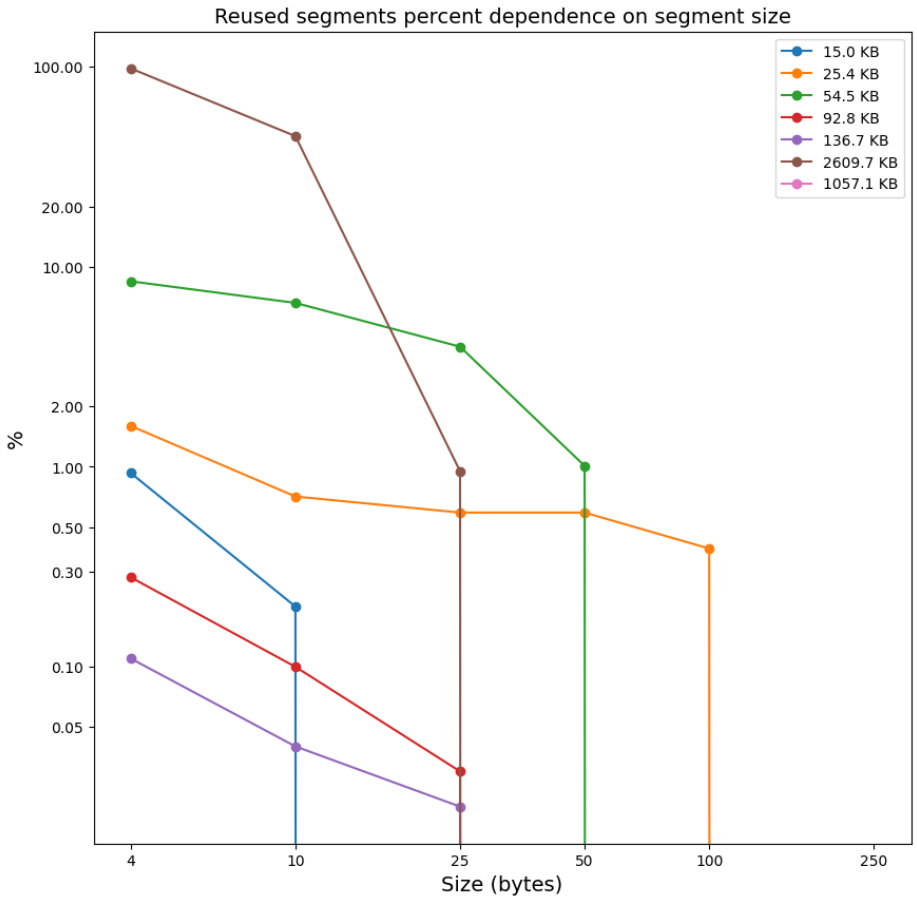

Рисунок 5. Зависимость процента переиспользованных сегментов от размера сегмента

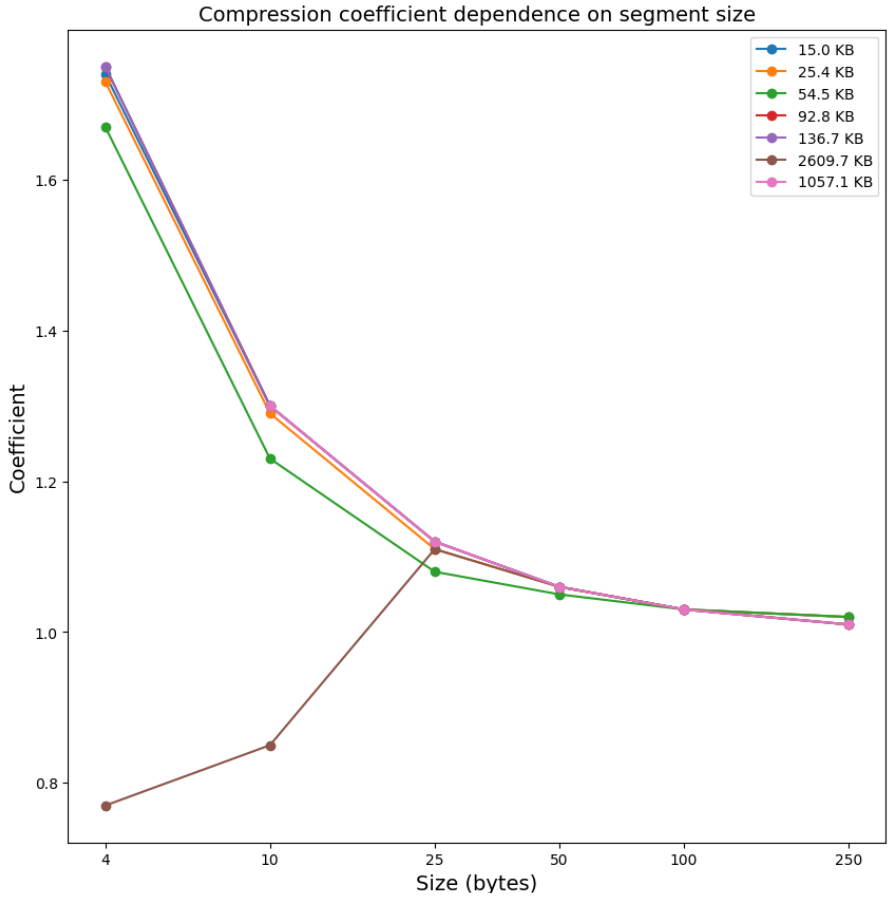

Рисунок 6. Зависимость коэффициента сжатия от размера сегмента

Рисунок 7. Изображения с наименьшим разнообразием цветов

Значение коэффициента сжатия больше единицы означает, что размер исходного файла увеличился за счет размера идентификаторов и малого переиспользования сегментов. Лучший результат наблюдается на текстовом файле. Несмотря на малый размер сегмента считывания, благодаря высокому проценту переиспользования (97.94%) скорость сжатия выше, чем на большем сегменте. Однако и на сегменте длиной 10 байт, процент переиспользованных сегментов достаточно высок – 44.92%. Коэффициенты сжатия для 4 байт и 10 байт – 0.77 и 0.85 соответственно. В остальных случаях сжатия файла не происходит.

Исследование зависимостей от хеш-функции

Изучим зависимость скорости чтения/записи от используемой хеш-функции. Используем для этого пять первых и три последних по величине изображения, а также текстовый файл размера 2.6 мегабайт. Зафиксируем параметры системы: id_size = 3, seg_size = 25, storage_size = 5000. Будем использовать следующие хеш-функции, соответствующие методам библиотеки hashlib:

```
MD5 = "MD5" # output - 16 bytes
SHA1 = "SHA128" # output - 20 bytes
SHA256 = "SHA256" # output - 32 bytes
SHA512 = "SHA512" # output - 64 bytes
SHA224 = "SHA3_224" # output - 28 bytes
```

Исходя из графиков (рис. 8 – рис. 9), можно сказать, что самыми быстрыми являются MD5 и SHA3_224.

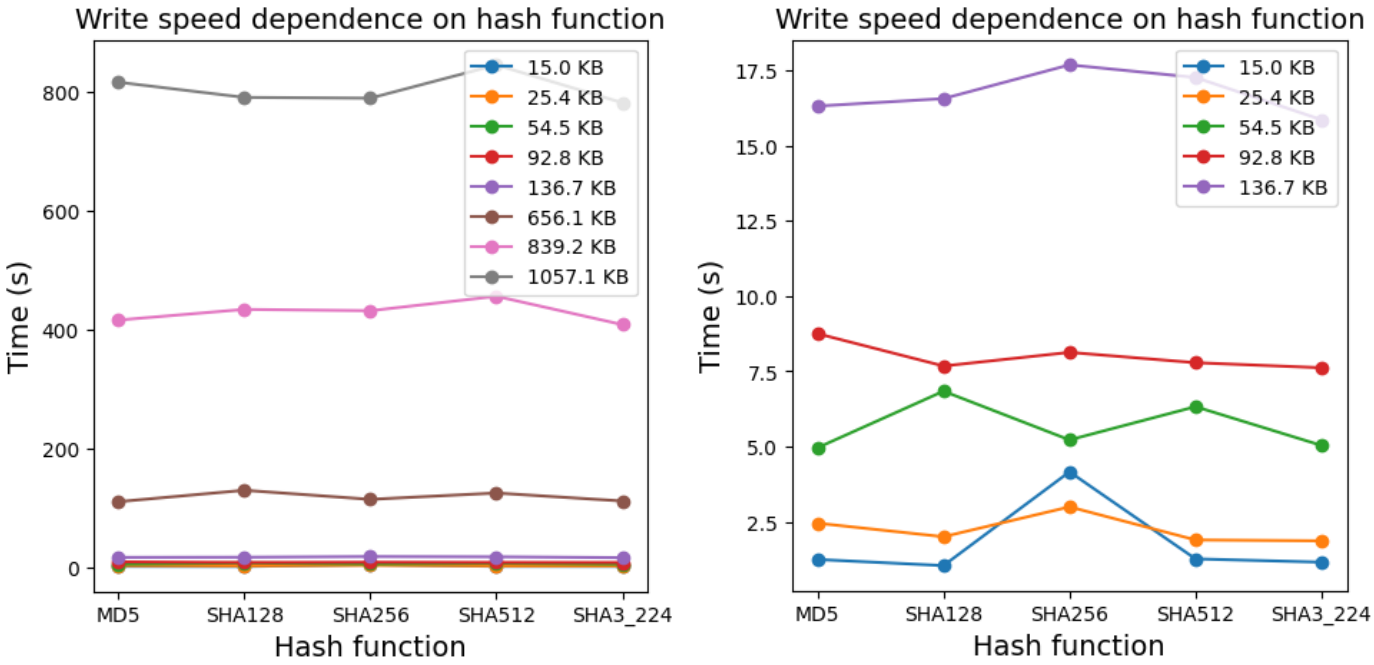

Рисунок 8. Зависимость скорости записи от функции хеширования

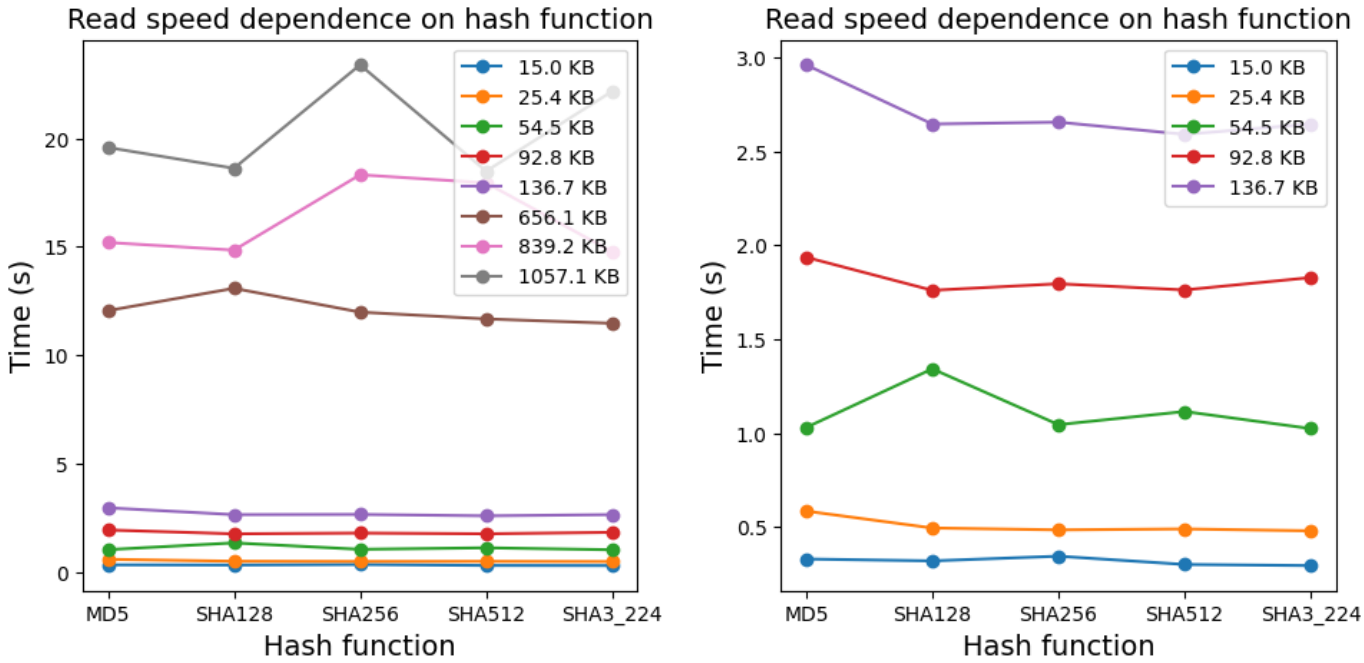

Рисунок 9. Зависимость скорости чтения от функции хеширования

Заключение

В результате проделанной работы был получен прототип системы хранения данных, использующей подход дедупликации, и проведено тестирование на небольших файлах для измерения производительности созданного прототипа. В ходе тестирования ошибок восстановления файлов не обнаружено.

По результатам тестирования можно заключить:

- Разработанная система хранения позволяет корректно сохранять и восстанавливать без потери сегментов файлы до нескольких мегабайт.
- С увеличением размера файла хранения прослеживается повышение скорости чтения/записи данных. Выбор оптимального значения этого параметра – сложная задача, так как оно может определяться многими факторами, например, свойствами используемых физических дисков.
- Максимальная скорость чтения и записи наблюдается при размере сегмента 250 байт, но при таком сегменте на тестовых данных не обнаружено ни одного повторяющегося сегмента.
- По полученным показателям дедупликации изображений на рис. 7 можно предположить, что система хранения может быть эффективна для сохранения изображений с малым количеством цветов.
- Среди тестовых данных лучшее поведение системы хранения наблюдается на текстовом файле. Размер других файлов при сохранении в хранилище увеличивается за счет дополнительной записи трехбайтных идентификаторов всех сегментов и высокого содержания уникальных сегментов.
- На тестовых данных без усреднения по нескольким прогонам тестов система хранения быстрее всего работает при хеш-функциях MD5 и SHA3_224.

Список источников

1. https://www.postgresql.org/docs/16/bookindex.html
2. https://www.psycopg.org/docs/