Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«База данных "ключ-значение"с компрессией данных»

Студент _____ _____ _____
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель _____ _____
(Подпись, дата) (И.О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

1 Введение

Многие приложения генерируют большие объемы однотипных текстовых данных в процессе работы, что может привести к значительному расходу памяти. Для сжатия данных существуют алгоритмы компрессии, которые могут значительно уменьшать объем таких файлов. Использовать такой метод будет удобно, если реализовать компонент, представляющий собой базу данных, которая будет сжимать поступающие данные при помощи этого алгоритма. Одним из примеров подобной такой базы данных является хранение логов системы антиплагиата в формате JSON, который подходит для эффективного сжатия. В рамках выполения курсовой работы были изучены и проанализированы различные алгоритмы компрессии и разработана база данных с интерфейсом "ключ-значение"на языке C++, использующая алгоритм сжатия lzw.

2 Теоретические сведения

2.1 Алгоритм LZW

Алгоритм Лемпеля-Зива-Велча (LZW) - это универсальный алгоритм сжатия данных без потерь, созданный трио ученых: Авраамом Лемпелем, Яаковом Зивом и Терри Велчем. Он был опубликован в 1984 году [1] как улучшенная версия алгоритма LZ78, предложенного Лемпелем и Зивом в 1978 году. Алгоритм был разработан таким образом, чтобы его можно было легко реализовать как программно, так и аппаратно.

Алгоритм сжатия LZW работает следующим образом: символы входного потока последовательно проверяются на наличие соответствующих строк в таблице. Если строка уже существует, то алгоритм переходит к следующему символу. В противном случае, алгоритм добавляет код предыдущей найденной строки в выходной поток и добавляет новую строку в таблицу. Новые строки добавляются в таблицу по мере их появления и им соответствует уникальный код.

Для декодирования используется только закодированный текст, так как алгоритм может воссоздать таблицу преобразований по этому тексту. Каждый раз, когда генерируется новый код, новая строка добавляется в таблицу, но если строка уже известна, алгоритм выводит ее код без генерации нового. Таким образом, каждая строка хранится только в одном экземпляре с уникальным номером. При декодировании при получении нового кода генерируется новая строка, а при получении уже известного, строка извлекается из таблицы.

Алгоритм кодирования

1. Все возможные символы заносятся в словарь. Во входную фразу X заносится первый символ сообщения.
2. Считать очередной символ Y из сообщения.
3. Если Y — это символ конца сообщения, то выдать код для X, иначе:
   - — Если фраза XY уже имеется в словаре, то присвоить входной фразе значение XY и перейти к Шагу 2,
   - — Иначе выдать код для входной фразы X, добавить XY в словарь и присвоить входной фразе значение Y. Перейти к Шагу 2.

Алгоритм декодирования

1. Все возможные символы заносятся в словарь. Во входную фразу X заносится первый код декодируемого сообщения.
2. Считать очередной код Y из сообщения.
3. Если Y — это конец сообщения, то выдать символ, соответствующий коду X, иначе:
   — Если фразы под кодом XY нет в словаре, вывести фразу, соответствующую коду X, а фразу с кодом XY занести в словарь.
   — Иначе присвоить входной фразе код XY и перейти к Шагу 2 .

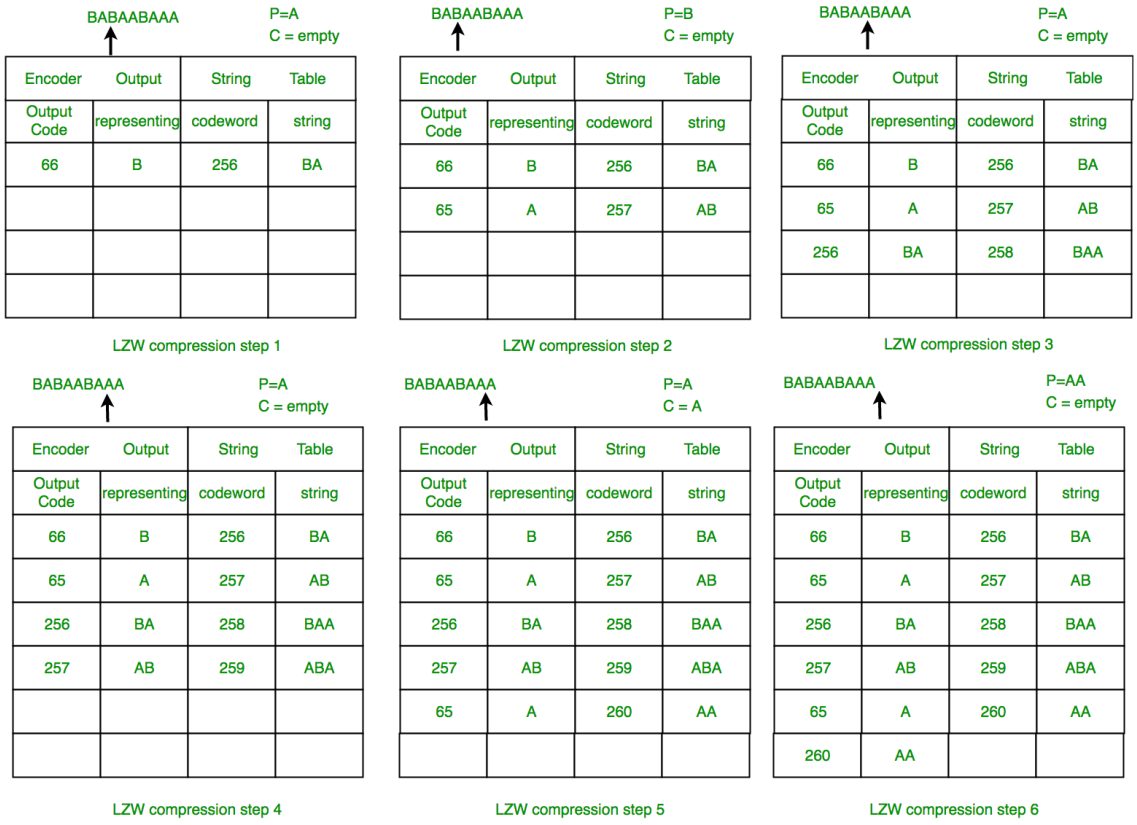

Рисунок 1 — Работа алгоритма LZW на примере строки $BABAABAAA$

2.2 B-дерево

B-дерево (англ. B-tree) — сильноветвящееся сбалансированное дерево поиска, позволяющее проводить поиск, добавление и удаление элементов за $O(logn)$. B-дерево с n узлами имеет высоту $O(logn)$. Количество детей узлов может быть от нескольких до тысяч (обычно степень ветвления B-дерева определяется харак-

теристиками устройства (дисков), на котором производится работа с деревом). B-деревья также могут использоваться для реализации многих операций над динамическими множествами за время $O(logn)$.

2.2.1 Назначение B-дерева

B-деревья находят свою область применения в файловых системах и иных энергонезависимых носителях информации с прямым доступом, а также в базах данных. B-деревья имеют схожую структуру с красно-чёрными деревьями (например, все B-деревья с n узлами имеют высоту $O(logn)$), однако они лучше минимизируют количество операций чтения-записи с диском.

2.2.2 Структура B-дерева

B-дерево является идеально сбалансированным, то есть глубина всех его листьев одинакова. B-дерево имеет следующие свойства (t — параметр дерева, называемый минимальной степенью B-дерева, не меньший 2.):

- Каждый узел, кроме корня, содержит не менее $t-1$ ключей, и каждый внутренний узел имеет по меньшей мере t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ.
- Каждый узел, кроме корня, содержит не более $2t-1$ ключей и не более чем $2t$ сыновей во внутренних узлах
- Корень содержит от 1 до $2t-1$ ключей, если дерево не пусто и от 2 до $2t$ детей при высоте большей 0 .
- Каждый узел дерева, кроме листьев, содержащий ключи $k1, ..., kn$, имеет n+1 сына. i-й сын содержит ключи из отрезка $[ki-1; ki]$, $k0 = -\infty, kn+1 = \infty$.
- Ключи в каждом узле упорядочены по неубыванию.
- Все листья находятся на одном уровне.

Структуры узла и дерева могут быть представлены следующим образом:

```
1  struct Node
2      bool leaf     // является ли узел листом
3      int  n        // количество ключей узла
4      int  key[]    // ключи узла
5      Node c[]      // указатели на детей узла
```

```
6
7  struct BTree
8     int  t        // минимальная степень дерева
9     Node root     // указатель на корень дерева
```

2.2.3 Высота B-дерева

Количество обращений к диску, необходимое для выполнения большинства операций с B-деревом, пропорционально его высоте. Для высоты B-дерева h выполняется формула:

$$h \leq log_t \frac{n+1}{2},$$

где $n \geq 1$ - количесвто узлов дерева, $t \geq 2$ - минимальная степень дерева.

2.2.4 B^+-дерево

Будем рассматривать модификацию B-дерева, называемую B^+-деревом.

В отличие от B-деревьев, где во всех вершинах хранятся ключи вместе с сопутствующей информацией, в B^+-деревьях вся информация хранится в листьях, в то время как во внутренних узлах хранятся только копии ключей. Таким образом удается получить максимально возможную степень ветвления во внутренних узлах. Кроме того, листовой узел может включать в себя указатель на следующий листовой узел для ускорения последовательного доступа, что решает одну из главных проблем B-деревьев.

Структуры узла и дерева могут быть представлены следующим образом:

```
1  struct Node
2     bool leaf        // является ли узел листом
3     int  key_num     // количество ключей узла
4     int  key[]       // ключи узла
5     Node parent      // указатель на отца
6     Node child[]     // указатели на детей узла
7     Info pointers[] // если лист - указатели на данные
8     Node left        // указатель на левого брата
9     Node right       // указатель на правого брата
10
```

```
11 struct BPlusTree
12     int  t          // минимальная степень дерева
13     Node root       // указатель на корень дерева
```

2.2.5 Высота B^+-дерева

Для высоты B^+-дерева h выполняется формула:

$$h \leq log_t \frac{n}{2} + 1,$$

где $n \geq 1$ - количесвто узлов дерева, $t \geq 2$ - минимальная степень дерева.

Как можно заметить, высота B^+-дерева не более чем на 1 отличается от высоты B-дерева, пожтому хранение информации только в листах почти не ухудшает эффективность дерева

2.2.6 Операции в B^+-дереве

B-деревья - это сбалансированные деревья, которые позволяют выполнять стандартные операции с временем выполнения, пропорциональным высоте. Основная цель алгоритмов B-дерева заключается в уменьшении количества операций ввода-вывода, что делает их особенно эффективными для работы с большими объемами информации и базами данных на дисках или других носителях информации.

Поиск ключа

— Реализуем вспомогательную функцию *find_leaf*, которая вернет лист с ключом, переданным ей:

1. Определим интервал и перейдем к следующему потомку.
2. Повторяем до тех пор, пока не достигнем листа.

— Находим нужный лист через *find_leaf* и ищем нужный ключ в нем.

Добавление ключа

1. Ищем лист, куда можно поместить ключ, и заносим его в список ключей.
2. Если узел заполнен, то мы должны разделить его на два. При этом мы предполагаем, что в дереве не может быть двух одинаковых ключей.

3. В результате выполнения операции *insert* будет возвращено значение, указывающее на то, был ли добавлен новый ключ.

Разбиение узла

1. Добавляем первые t ключей в первый подузел и последние t-1 ключей во второй подузел.
2. Если узел является листом, то оставшийся ключ добавляется в правое поддерево, а его копия становится разделительной точкой для двух новых поддеревьев в родительском узле.
3. Если родительский узел уже заполнен, мы не копируем ключ, а перемещаем его в родительский узел, так как это просто дубликат.
4. Повторяется до тех пор, пока мы не достигнем пустого узла или корня дерева.
5. Если мы достигли корня, то мы разбиваем его на два узла, что приводит к увеличению высоты дерева.

Так как минимальный ключ из второй половины всегда отправляется в родительскую вершину, каждый ключ, находящийся во внутренней вершине, является минимальным для правого поддерева этого ключа.

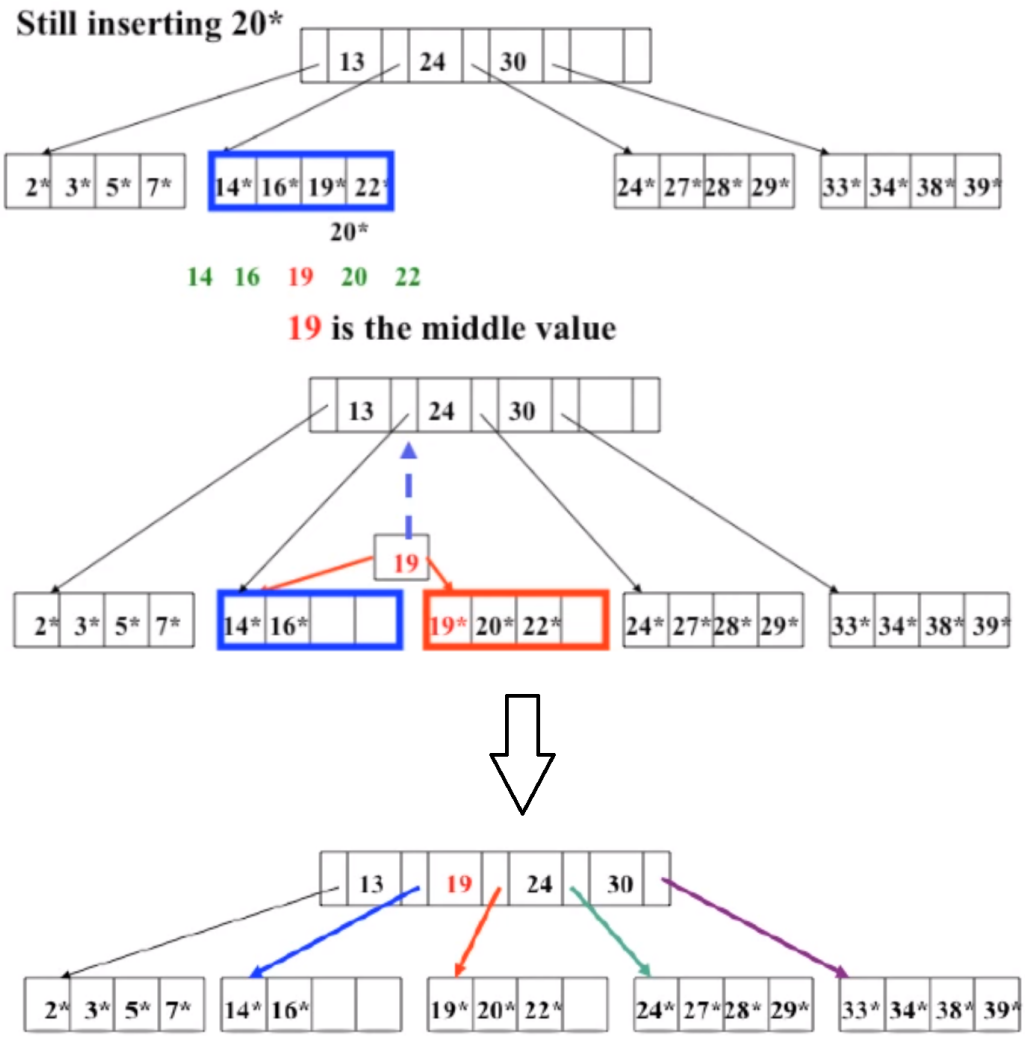

Рисунок 2 — Схема разбиения узла в B^+-дереве

Удаление

1. Ищем листовой узел, в котором находится необходимый ключ.
2. Если узел содержит не менее t-1 ключей, где t - это степень дерева, то удаление завершено.
3. Иначе необходимо выполнить попытку перераспределения элементов, то есть добавить в узел элемент из левого или правого брата (не забыв обновить информацию в родителе).
4. Если это невозможно, необходимо выполнить слияние с братом и удалить ключ, который указывает на удалённый узел. Объединение может распространяться на корень, тогда происходит уменьшение высоты дерева.

Так как мы считаем, что в дереве не может находиться два одинаковых ключа, то *delete* будет возвращать был ли удален ключ.

2.2.7 Представление B-дерева в памяти

Кроме оперативной памяти, в компьютере используется внешний носитель, как правило, представляющий собой магнитные диски (или твердотельный накопитель). Хотя диски существенно дешевле оперативной памяти и имеют высокую емкость, они гораздо медленнее оперативной памяти из-за механического построения считывания.

Для того чтобы снизить время ожидания, связанное с механическим перемещением, при обращении к диску выполняется обращение одновременно сразу к нескольким элементам, хранящимся на диске. Информация разделяется на несколько страниц одинакового размера, которые хранятся последовательно друг за другом в пределах одного цилиндра (набора дорожек на дисках на одном расстоянии от центра), и каждая операция чтения или записи работает сразу с несколькими страницами. Для типичного диска размер страницы варьируется от 2 до 16 КБайт. После того, как головка установлена на нужную дорожку, а диск поворачивается так, что головка становится на начало интересующей нас страницы, чтение и запись становятся полностью электронными процессами, не зависящими от поворота диска, и диск может быстро читать или писать крупные объёмы данных.

В типичном приложении с B-деревом, объём хранимой информации так велик, что вся она просто не может храниться в основной памяти единовременно. Алгоритмы B-дерева копируют выбранные страницы с диска в основную память по мере надобности и записывает обратно на диск страницы, которые были изменены. Алгоритмы B-дерева хранят лишь определённое количество страниц в основной памяти в любой момент времени; таким образом, объём основной памяти не ограничивает размер B-деревьев, которые можно создавать.

Система в состоянии поддерживать в процессе работы в оперативной памяти только ограниченное количество страниц. Мы будем считать, что страницы, которые более не используются, удаляются из оперативной памяти системой; наши алгоритмы работы с B-деревьями не будут заниматься этим самостоятельно. Поскольку в большинстве систем время выполнения алгоритма, работающего с B-деревьями, зависит в первую очередь от количества выполняемых операций чтения/записи с диском, желательно минимизировать их количество и за один раз считывать и записывать как можно больше информации. Таким образом, размер

узла B-дерева обычно соответствует дисковой странице. Количество потомков узла B-дерева, таким образом, ограничивается размером дисковой страницы. Для больших B-деревьев, хранящихся на диске, степень ветвления обычно находится между 50 и 2000, в зависимости от размера ключа относительно размера страницы. Большая степень ветвления резко снижает как высоту дерева, так и количество обращений к диску для поиска ключа. Например, если есть миллиард ключей, и $t = 1001$, то поиск ключа займёт две дисковые операции.

3 Практическая реализация

3.1 Язык реализации

В языка реализации был выбран C++, так как он он имеет слвместимость с языком C, что позволяет проще реализовать интерфейс прилодения в виде библиотеки на языке C. Кроме того, в языке C++ имеются встроенные структуры данных, такие как std::map, которые реализуют эффективный по времени доступ к данным в оперативной памяти. Так же выбранный язык позволяет реализовать подоход объектно-ориентированного программирования, что упрощает написание сложного приложения.

3.2 Обзор модулей

Для удобства разработки было реализовано несколько модулей, представленных в виде классов, каждый из которых выполняет свою задачу.

В реализации представлены следующие классы:

— **BTreePackedDb** — основной модуль базы данных, реализующий опреации вставки и чтения значений, а так же загрузки и созранинния данных,

— **Compressor** — класс, реализующий алгоритм сжатия LZW,

— **Dictionary** — класс, отвечающий за хранение табдицы для кодирования и декодирования,

— **BPlusIndex** — реализация B-дерева,

— **FileHandler** — интерфейс для взаимодействия с бинарными файлами.

Зависимость модулей друг от друга приведена в следующей диаграмме:

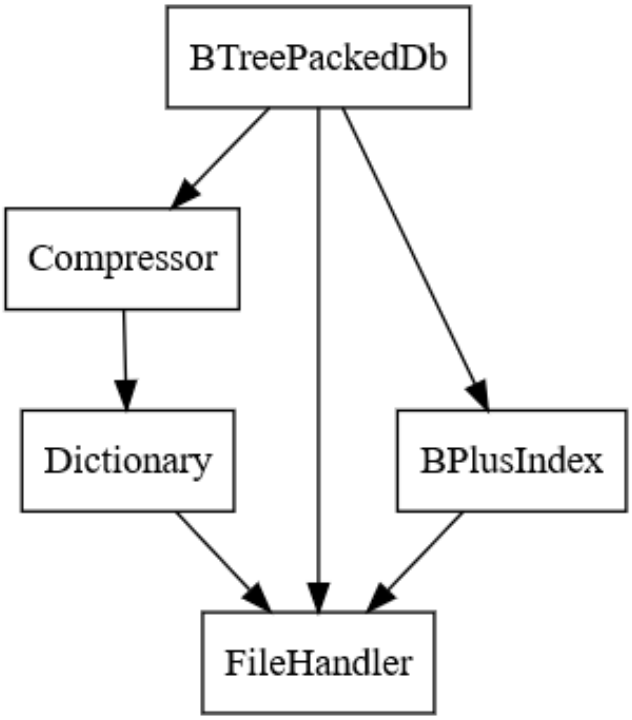

Рисунок 3 — Диаграмма зависимости классов

3.3 Взаимодействие с бинарными файлами

Для хранения фрагмента бинарного файла будем использовать структуру Fragment, содержащую начало и конец последовательности битов. Так же для данного класса предусмотрена сериализация в потоки вывода для удобства отладки.

```
1  struct Fragment
2  {
3          ull begin = 0;
4          ull end = 0;
5          Fragment() {}
6          Fragment(ull begin, ull end) : begin(begin), end(end) {}
7          friend ostream &operator<<(ostream &stream, const Fragment &frag);
8  };
```

3.3.1 Класс-обработчик бинарного файла

Для чтения и записи примитивов в файлы был реализован класс *FileHandler*, который реализует следующие методы:

— Загрузка файла, создание нового файла

— Запись массива целых беззнаковых чисел в файл
— Запись строки в файл
— Чтение массива целых беззнаковых чисел из файла
— Чтение строки из файла

Для чтения строк и массивов предусмотрено 2 режима:

— Чтение из произвольного места файла. В этом случае методу передаётся номер байта, начиная с которого производистя чтение.
— Последовательное чтение. Если метод чтения вызван без аргументов, чтение происходит с того места, на котором было закончено чтения в результате предыдущей операции.

Публичный интерфейс класса выглядит следующим образом:

```
class FileHandler
{
public:
        ~FileHandler();

        bool open(const string &filename);
        bool create(const string &filename);
        void close();
        Fragment write_vector(const vector<size_t> &vec);
        Fragment write_string(const string &str);

        vector<size_t> read_vector();
        vector<size_t> read_vector(ull pos);
        string read_string();
        string read_string(ull pos);

        void read_pos(ull pos);
        void clear();

        bool eof();
 ...
```

В приватной области класса определены вспомогательные функции для записи беззнакового целого и символа, а так же методы их чтения из файла.

```
...
private:
```

```
        string filename;
        ull write_pos = 0;
        fstream file;
        size_t write_uint(size_t val);
        size_t write_char(char c);
        size_t read_uint();
        char read_char();
};
```

3.3.2 Кодирование примитивов

Рассмотрим подробнее функцию записи беззнакового целого числа в файл.

Функция использует алгоритм для кодирования значений беззнаковых целых чисел, чтобы сохранить его эффективно и занимаемое пространство было минимально возможным. Для этого каждое число разбивается на группы по 7 бит, и каждая группа кодируется в один байт. Все байты, кроме последнего, имеют старший бит установленным в единицу, а у последнего байта старший бит устанавливается в ноль. Это означает, что номер последней группы может быть определен по количеству единиц в старших битах всех предыдущих байтов.

Функция начинает с проверки, является ли значение равным нулю. Если это так, то функция записывает один байт со значением 128 в файл. Это означает, что число равно нулю и занимает только один байт.

Затем функция использует цикл while для разбивки значения беззнакового целого числа val на группы по 7 бит и кодирования каждой группы в один байт. Она вычисляет остаток от деления значения val на 128 и записывает его в файл. Затем значение val делится на 128, чтобы перейти к следующей группе битов. Если значение $val/128$ равно нулю, то это последняя группа битов, и старший бит устанавливается в ноль.

Функция возвращает количество байтов, которые были записаны в файл.

```
size_t FileHandler::write_uint(size_t val)
{
        size_t counter = 0;
        if (val == 0)
        {
                unsigned char c = 128;
```

```
7                   file.write((const char *)&c, sizeof(char));
8                   return sizeof(unsigned char);
9           }
10          while (val)
11          {
12                  unsigned char c = val % 128;
13                  if (val / 128 == 0)
14                          c += 128;
15                  file.write((const char *)&c, sizeof(unsigned char));
16                  val /= 128;
17                  counter++;
18          }
19          return counter * sizeof(unsigned char);
20  }
```

Функция *write_vector()* записывает вектор беззнаковых целых чисел (vec) в файл, используя функцию write_uint(). Сначала в файл записываается длина вектора, а затем последовательно его элементы. В конце функция возвращает объект типа Fragment с координатами записанного фрагмента.

```
1   Fragment FileHandler::write_vector(const vector<size_t> &vec)
2   {
3           Fragment frag(write_pos, write_pos);
4           frag.end += write_uint(vec.size());
5           for (auto k : vec)
6                   frag.end += write_uint(k);
7           write_pos = frag.end;
8           return frag;
9   }
```

Функции записи символов типа *char* и строк имеют аналогичную структуру за тем исключением, что символ записывается как один байт.

3.3.3 Декодирование примитивов

Функция read_uint() читает из файла беззнаковое целое число, используя алгоритм декодирования, обратный write_uint(). Она считывает первый байт числа из файла, удаляет старший бит и добавляет его к накопителю acc, умножая

на текущую степень power. Затем она увеличивает значение power, умножая его на 128 и повторяет этот процесс для следующих байтов, пока старший бит последнего байта не будет равен 0. Функция возвращает значение декодированного беззнакового целого числа.

```
size_t FileHandler::read_uint()
{
        unsigned char c;
        size_t acc = 0;
        size_t power = 1;

        do
        {
                file.read((char *)(&c), sizeof(unsigned char));
                acc += (c & ~128) * power;
                power *= 128;
        } while (!(c & 128));

        return acc;
}
```

Функция чтения вектора сначала считывает длину входной последовательности, а затем соответсвующее количество эллементов.

```
vector<size_t> FileHandler::read_vector()
{
        vector<size_t> res;
        size_t length = read_uint();
        for (size_t i = 0; !eof() && i < length; i++)
                res.push_back(read_uint());
        return res;
}
```

Для работы с произвольным местом файла перед вызовом функции необходимо вызвать переход к заданной позициии:

```
void FileHandler::read_pos(ull pos)
{
        file.seekp(pos * sizeof(unsigned char), ios::beg);
}
```

3.4 Хранение таблицы кодирования

3.4.1 Формат хранения таблицы кодирования в файле

В алгоритме LZW каждой строке в таблие соответсвует свой порядковый номер, являющийся ключом. Для быстродействия работы программы таблица полностью загружается в оперативную память во время сессии. Это обусловлено тем, что к ней потребуется выполнять много обращений в процессе кодирования и декодирования.

Таким образом самый простой способ хранения таблицы LZW — последовательность подряд идущих строк в файле. При загрузке строки помещаются в структуру std::map, что позволяет ускорить доступ к ключам по строкам.

3.4.2 Интерфейс взаимодейсвия с файлом

Класс Dictionary предназначен для хранения таблицы, используемой в алгоритме LZW. Он содержит методы для открытия и создания файлов, добавления строк в таблицу, получения значений по ключу или строке, проверки наличия строки в таблице. Кроме того, класс реализует оператор доступа к элементам таблицы по ключу и по строке.

Закрытые данные класса включают объект FileHandler для работы с бинарными файлами, массив key_to_str для хранения строк по соответствующим ключам и словарь str_to_key для хранения ключей по соответствующим строкам.

```
1  class Dictionary
2  {
3  public:
4          bool open(const string &filename);
5          bool create(const string &filename);
6
7          void add(const string &value, bool add_to_file = 1);
8          string get_str(size_t key) const;
9          std::optional<size_t> get_key(const string &str) const;
10         bool contains(const string &str) const;
11
12         string operator[](size_t key) const;
13         size_t operator[](const string &str) const;
```

```
14
15  private:
16          FileHandler fh;
17          vector<string> key_to_str;
18          map<string, size_t> str_to_key;
19  };
```

При создании таблицы происходит проверка загруженной таблицы на пустоту. Если полученная таблица пуста, то её необходимо инициализировать, добавив ключи для всех однобайтовых символов.

```
1   bool Dictionary::open(const string &filename)
2   {
3           if (!fh.open(filename))
4                   return false;
5           fh.read_pos(0);
6           while (!fh.eof())
7                   add(fh.read_string(), 0);
8
9           fh.close();
10          if (!fh.open(filename))
11                  return false;
12
13          if (key_to_str.size() == 0)
14          {
15                  for (unsigned k = 1; k < 256; k++)
16                  {
17                          string str = "";
18                          str.push_back(k);
19                          add(str);
20                  }
21          }
22
23          return true;
24  }
```

3.5 Сжатие с использованием алгоритма LZW

Функция compress реализует алгоритм сжатия данных LZW, ей передаётся таблица замены последовательностей символов короткими кодами. Функция возвращает вектор целых чисел - компрессированный результат, где каждое число соответствует индексу значения из словаря. Алгоритм работает посимвольно, проверяет наличие последовательности символов в словаре и если такая последовательность найдена, то она добавляется к текущей строке acc, иначе значение текущей строки записывается в результат res, добавляется новая запись в словарь и текущая строка устанавливается в символ c. Алгоритм заканчивает работу, добавляя в результат последний элемент словаря, соответствующий последней строке acc.

```
vector<size_t> compressLZW(Dictionary &dict, const string &str)
{
        vector<size_t> res;
        string acc = "";
        for (char c : str)
        {
                if (dict.contains(acc + c))
                {
                        acc += c;
                }
                else
                {
                        res.push_back(dict[acc]);
                        dict.add(acc + c);
                        acc = c;
                }
        }
        res.push_back(dict[acc]);
        return res;
}
```

Функция decompress реализует расшировку упакованных данных. Она последовательно извлекает ключи из вектора и суммирует соответствующие подстроки в финальную строку.

```
string decompress(const Dictionary &dict, const vector<size_t>& arr)
{
        string res = "";
        for (auto key : arr)
        {
                res += dict[key];
        }
        return res;
}
```

3.6 Реализация B^+-дерева

Для реализации B^+-дерева был применен механизм шаблонов языка C++. Это позволило создать создать класс B^+-дерева не зависящим от типа ключа. К типу ключа предъявлятся лишь следующин требования: ключ должен быть фиксированного размера и для него должны быть определены операции сравнения.

Это также позволило упростить реализацию класса: размер экземпляра узла определяется на этапе компиляции, что позволяет манипулировать узлами без использования динамической памяти. Обратной стороной выбранного решения является невозможность изменить размер ключа и/или узла в процессе работы программы. Однако для экспериментального кода не прдназначеннного для промышленного использования такое решение приемлемо.

3.6.1 Класс B-дерева

Каждый узел B^+-дерева описывается шаблоном

```
template <typename K> struct BPlusRecord
{
    K key;
    size_t value;
};
template <typename K, int SIZE> class BPlusNode
{
    size_t count;
    size_t left_node_pos;
```

```
10      BPlusRecord<K> records[SIZE];
11  }
```

Таким образом размер каждого узла фиксирован и определяется произведением размера ключа K на количество ключей в узле $SIZE$. Размер узла для максимальной скорости работы можнет быть подобран эксперимательно, так, чтобы за одно обращение к внешнему носителю загружался один узел целиком.

Каждый экземпояр $BPlusNode$ может выполнять роль как внутреннего, так и листового узла. Это накладывает ограничение на тип данных, хранимый в листовом узле. Так как во внутреннем узле тип хранимых данных должен быть ссылкой на узлел-потомок (номер узла-потомка в файле), то и хранимые в листе данные тоже могут быть только числом. Типом данных для хранения был выбран $size\_t$. Это компромиссное решение: для номера узла в дереве 64-битный тип слишком велик, можно обойтись 32-битным, а для пользовательских данных, которые могут представлять собой например позицию в файле 64-битный тип слишком мал. Для решения этой проблемы улучшенная реализация могла бы использовать разные типы для корневого и листового узлов, возможно храня их в отдельных файлах.

Поле $left\_node\_pos$ используется только внутренними узлами. В нем хранится номер узла-потомка для ключей меньших, чем $records[0].key$. Таким образом внутренний узел с количеством ключей $count$ определяет $count^+1$ интервал и хранит ссылки на $count + 1$ потомков.

3.6.2 Формат хранения B^+-дерева в файле

B^+-дерево записывается в файл поблочно, блоками фиксированного размера. Каждый блок является экземпляром класса $BPlueNode$. Корневой узел всегда находится на нулевой позиции в файле. В начале работы корневой узел является также и листовым. Когда количество добавленных ключей превышает размер узла, узел разделяется на две части. Левая часть остается на старой позиции в файле, а правая добавляется к концу файла. Указатель на позицию добавленного узла добавляется к узлу-родителю. Если родитель также переполняется, процедура повторяется.

Дополнительно обрабатывается ситуация, когда переполняется корневой узел. В этом случае левая часть корневого узла также переносится в конец файлв, а

в нулевую позицию записывается корневой узел с двумя потомками - половинами быашего корневого узла.

3.6.3 Главный класс B^+-дерева

Основной класс индекса $BPlusIndex$ также является шаблоном от типа ключа и количества ключей.

```
template <typename K, int SIZE> class BPlusIndex
{
public:
    bool create(const string &filename);
    bool open(const string &filename)
    size_t get_nodes_count()
    size_t get_keys_count();
    optional<size_t> get(const K &key);
    void put(const K &key, size_t value);
}
```

Класс предоставляет методы для создания нового файла, открытия существующего файла и методы для добавления и получения значения тика $size_t$ по ключу K,

Класс не реализует кеширования узлов: при каждом обращении к узлу происходит обращение напосредственно к файлу. Такое решение не подходит для промышленного использования, но зато позволяет провести эксперименты по определению зависимости скорости работы от размера узла без дополнительных факторов.

3.6.4 Дополнительный класс для B^+-дерева со строковым ключом

Для удобства работы со строковыми ключами реализован шаблонный класс $BPlusIndexCharBuf$. Данный класс упрощает работу со строковыми ключами, позволяя вызывать методы get и put с параметром ключа $constchar*$. Шаблон принимает параметры $SIZE\_BUF$ - размер строкового ключа в индексе и $SIZE$ - количество ключей в узле.

```
template <int BUF_SIZE, int SIZE>
class BPlusIndexCharBuf : public BPlusIndex<BPlusCharBuf<BUF_SIZE>, SIZE> {};
```

$BPlusIndexCharBuf$ является пустым, так как возможность вызывать get и put с параметром ключа $constchar*$ фактически реализована в $BPlusCharBuf$, который является оберткой для $char[BUFF\_SIZE]$ с конструктором из $constchar*$ и операциями сравнения, необходимыми для $BPlusIndex$.

3.7 Структура базы данных

3.7.1 Формат хранения базы данных в файле

3.7.2 Класс базы данных в C++

Интерфейс на C++ фактически повторяет интерфейс класса $BPlusIndex$ с той лишь разницей, что типом данных для записи и извлечения является строка.

```
class BTreePackedDb
{
public:
    BTreePackedDb(const char *basename);
    bool create();
    bool open();
    void put(const string &key, const string &value);
    const char *get(const string &key);
private:
    ...
    BPlusIndexCharBuf<BTREE_KEY_SIZE, BTREE_NODE_SIZE> index;
    FileHandler data;
    Dictionary dictionary;

    string buf;
}
```

Параметры размера ключа $BTREE\_KEY\_SIZE$ и количества ключей в узле $BTREE\_NODE\_SIZE$ определяются на этапе компиляции.

База состоит из трех файлов: файла индекса ($index$, файла словаря $dictionary$ и файла упакованных строк $data$.

При добавлении записи:

— строка $value$ пакуется алгоритмом LZW и обновленный словарь записывается в $dictionary$

— упакованные данные пишутся в файлу строк $data$

— позиция в $data$ сохраняется в $index$ с ключом key

Извлечение происходит в обрастном порядке: в индексе находится позиция упакованных данных, данные извлекаются и распоковываются. Распаковка выполняется во внутренний буфер класса и метод get возвращает указатель на буфер. Указатель остается валидным до нового вызова get.

3.8 Интерфейс на языке C

Для использоваия базы из языка C предоставляется структура

```
1 struct btree_packed_db
2 {
3     void *data;
4 };
```

Данная структура содержит указатель на экземпляр $BTreePackedDb$ в динамической памяти. Для создания, ударения и работы с базой предоставляются функции, повторяющие методы класса $BTreePackedDb$.

```
1 struct btree_packed_db *btree_packed_db_open(const char *basename);
2 struct btree_packed_db *btree_packed_db_create(const char *basename);
3 void btree_packed_db_close(struct btree_packed_db *db);
4 void btree_packed_db_put(struct btree_packed_db *db, const char *key, const cha
5 const char *btree_packed_db_get(struct btree_packed_db *db, const char *key);
```

4 Тестирование

В проекте реализовны модульные и интеграционные тесты.

Модульные тесты находятся в директории $tests$ и могут быть запущены в автоматическом режиме с использованием тестирующей системы $CTest$, являющейся частью системы построения $CMake$

```
mkdir build
cd build
cmake ..
cmake --build . --target tests --config Release
ctest -C Release
```

Интеграционные тесты реализованы в виде утилит $pack$ и $unpack$ находящиеся в директории $examples$. Утилиты принимают три параметра: имя базы данных, текстовый файл с ключами, текстовый файл со значениями. Утилита $pack$ сканирует построчно файлы с ключами и значениями и добавляет пары ключ-значение в базу. Утилита $unpack$ сканирует файл с ключами и пишет в файл со значениями строки, извлеченные из базы.

Сценарий интеграционного теста следующий:

1. $packtestbasekeys.txtkeys.txt$ - ключ равен значению для удобства тестирования
2. $unpacktestbasekeys.txtkeys.out$ - файл keys.out должен содержать те же строки. что и исходный keys.txt

5 Анализ результатов

Был проведен эксперимент по измерению зависимости скорости записи и чтения B^+-дерева в зависимости от размера узла. Был использовал следующий скрипт для cmd.exe под Windows:

```
for /D %%i in (8, 80, 800, 8000, 80000) do call :test %1 %%i
goto :eof

:test
cmake ../.. -DBTREE_NODE_SIZE=%2 -DBTREE_KEY_SIZE=128
cmake --build ../.. --target examples --config Release
echo %2 >> result.txt
echo %time% >> result.txt
pack db %1 %1
echo %time% >> result.txt
unpack db %1 nul
echo %time% >> result.txt
exit /b
```

На каждом шагу библиотека перекомпилировалась с новым размером узла и фиксированным размером ключа. После компиляции запускалась упаковка 30 мегабайт уникальных ключей в базу и извлечение этих ключей из базы. Замерялось время, затраченное на запись и чтение всех ключей. Использовались модифицированные версии $pack$ и $unpack$ с отключенной упаковкой / распаковкой, проверялась только скорость работы двоичного дерева.

Результат измерения в таблице:

Размер узла	1K	10K	100K	1M	10M
Запись	26 с	25 с	115 с	1 с	1 с
Чтение	22 с	22 с	100 с	1 с	1 с

6 Перспективы дальнейшего развития

6.1 Кэширование записей

В текущей реализации чтение для получения данных из хранилища необходимо каждый раз заново читать данные с диска. Можно уменьшить количество обращений к диску, если реализовать кэш, то есть in-memory хранилище, в котором будет хранится ограниченное количество данных, и этот набор будет обновлятся при чтении или записи, вытесняя одни данные, и заменяя их другими. Такой подход имеет как преимущества, так и недостатки, к преимуществам можно отнести сокращение количества походов на диск при частом чтении одних и тех же записей, а к недостаткам - необходимость либо лишнего копирования, если данные в кэше будут сразу же записываться на физическое устройство, либо необходимость поддерживать состояние данных, когда они есть в оперативной памяти, но еще не записаны на диск. Интеграция данной функциональности требует дальнейшего анализа и тестирования.

6.2 Многопоточный режим работы

В настоящий момент база данных может работать только в однопоточном режиме, то есть ее методы могут быть вызваны только из одного и того же потока, попытка конкурентного доступа может обернутся гонкой условий, которая может привести к неконсистентному состоянию базы данных. Разумеется, можно добавить примитивы синхронизации, которые будут контролировать, чтобы в критические секции мог войти только один поток, но такой подход замедлит работу в одном потоке, и не позволит выиграть в производительности при работе в нескольких потоках. Если мы хотим обеспечить полноценную поддержку многопоточного чтения или записи, необходимо создание схемы параллельного доступа к данным.

ЗАКЛЮЧЕНИЕ

В рамках курсовой работы были изучены алгоритмы и проанализированны алгоритмы сжатия, такие как алгоритм Лемпеля — Зива — Уэлча, была изучена структура хранения пар ключ-значения в виде B^+-дерева. Была разработана база данных с интерфейсом "ключ-значение"на языке C++, использующая алгоритм сжатия LZW. Был разработан формат хранения базы данных на диске.

Разработка производилась с поддержкой модульного тестирвоания. К готовой реализации был создан программный интерфейс на языке C.

Реализация была проанализирована на быстродействие с использованием разных конфигураций хранения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Welch T. A. A technique for high-performance data compression // Computer. — 1984. — Т. 17, № 06. — С. 8—19.