Университет ИТМО

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.01 Информатика и вычислительная техника

Лабораторная работа №1

по дисциплине «Низкоуровневое программирование»
Вариант №2

Выполнил:
Студент группы P33302
Иванов Н.Д.

Преподаватель:
Кореньков Юрий Дмитриевич

г. Санкт-Петербург
2023

Содержание

Цели

Создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное обновление) информации общим объёмом от 10GB соответствующего варианту вида.

Задачи

1. Спроектировать структуры данных для представления информации в оперативной памяти
    a. Для порции данных, состоящий из элементов определённого рода (см форму данных), поддержать тривиальные значения по меньшей мере следующих типов: четырёхбайтовые целые числа и числа с плавающей точкой, текстовые строки произвольной длины, булевские значения
    b. Для информации о запросе
2. Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним:
    a. Операции над схемой данных (создание и удаление элементов схемы)
    b. Базовые операции над элементами данных в соответствии с текущим состоянием схемы (над узлами или записями заданного вида)
        i. Вставка элемента данных
        ii. Перечисление элементов данных
        iii. Обновление элемента данных
        iv. Удаление элемента данных
3. Используя в сигнатурах только структуры данных из п.1, реализовать публичный интерфейс со следующими операциями над файлом данных:
    a. Добавление, удаление и получение информации об элементах схемы данных, размещаемых в файле данных, на уровне, соответствующем виду узлов или записей
    b. Добавление нового элемента данных определённого вида
    c. Выборка набора элементов данных с учётом заданных условий и отношений со смежными элементами данных (по свойствам/полями/атрибутам и логическим связям соответственно)
    d. Обновление элементов данных, соответствующих заданным условиям
    e. Удаление элементов данных, соответствующих заданным условиям
4. Реализовать тестовую программу для демонстрации работоспособности решения
    a. Параметры для всех операций задаются посредством формирования соответствующих структур данных

b. Показать, что при выполнении операций, результат выполнения которых не отражает отношения между элементами данных, потребление оперативной памяти стремится к O(1) независимо от общего объёма фактического затрагиваемых данных
c. Показать, что операция вставки выполняется за O(1) независимо от размера данных, представленных в файле
d. Показать, что операция выборки без учёта отношений (но с опциональными условиями) выполняется за O(n), где n – количество представленных элементов данных выбираемого вида
e. Показать, что операции обновления и удаления элемента данных выполняются не более чем за O(n*m) > t -> O(n+m), где n – количество представленных элементов данных обрабатываемого вида, m – количество фактически затронутых элементов данных
f. Показать, что размер файла данных всегда пропорционален количеству фактически размещённых элементов данных
g. Показать работоспособность решения под управлением ОС семейств Windows и *NIX

5. Результаты тестирования по п.4 представить в составе отчёта, при этом:
   a. В части 3 привести описание структур данных, разработанных в соответствии с п.1
   b. В части 4 описать решение, реализованное в соответствии с пп.2-3
   c. В часть 5 включить графики на основе тестов, демонстрирующие амортизированные показатели ресурсоёмкости по п. 4

Описание работы

Структуры для хранения информации

Начнем с простого, есть enum DataType, который хранит поддерживаемые типы данных.
Далее идет структура FieldValue, она хранит в себе указатель общего типа на данные, на которые ссылается непосредственно поле и размер в байтах этого поля.
Ниже идет структура под названием LinkNext, она де факто хранит в себе метаданные, которые позволяют склеить 2 структуры данных EntityRecord
Ну и непосредственно главная структура - EntityRecord, она представляет собой запись сущности в реляционной базе данных, с точки зрения программы она хранит в себе указатель на 1-й элемент массива из структур FieldValue и ссылку на следующуя часть записи

```
enum DataType {
    INT,
    DOUBLE,
    STRING,
    BOOL
};


typedef struct {
    void *data;
    uint64_t dataSize;
} FieldValue;


typedef struct {
    uint64_t blockOffset;
    uint16_t offsetInBlock;
    uint8_t fieldNumber;
    uint64_t positionInField;
    uint16_t idPosition;
} LinkNext;

typedef struct {
    FieldValue *fields;
    LinkNext *linkNext;
} EntityRecord;
```

Структуры для работы с файлом

Файл поделен на блоки фиксированного размера (по 8 кб каждый). В блоке есть заголовок (HeaderSection), который хранит в себе информацию о номере текущего блока в рамках единой таблицы, отступы от конца заголовка с которого начинается и заканчивается свободное место (startEmptySpaceOffset и endEmptySpaceOffset соответственно). Также в заголовке хранится количество сущностей, которое находится в этом блоке (recordsNumber).

```
typedef struct {
   uint16_t pageNumber;
   uint16_t startEmptySpaceOffset;
   uint16_t endEmptySpaceOffset;
   uint8_t recordsNumber;
} HeaderSection;
```

Есть раздел, который называется SpecialDataSection (он содержит отступы от начала файла к предыдущему блоку и следующему).

```
typedef struct {
   uint64_t previousBlockOffset;
   uint64_t nextBlockOffset;
} SpecialDataSection;
```

В блоке для каждой записи (EntityRecord) хранится RecordId с информацией о длинне сущности и ее отступе от начала блока. (Вообще структура хранения данных в блоке скопирована со статьи про postgres, с одного конца заполняются RecordId, с другого EntityRecord,)

```
typedef struct {
   uint16_t offset;
   uint64_t length;
} RecordId;
```

Есть также структуры, которые представляют данные о таблице:
NameTypeBlock, который хранит в себе название поля и тип этого поля.

```
typedef struct {
   char fieldName[MAX_LENGTH_FIELD_NAME];
   enum DataType dataType;
} NameTypeBlock;
```

TableOffsetBlock, который хранит поле isActive (оно необходимо, чтобы можно было перезаписывать TableOffsetBlock в файлах, при удалении таблицы оно становится false тем самым указывает что таблицы больше не существует и на это место можно записать TableOffsetBlock для другой таблицы). Хранит имя таблицы(tableName) и массив NameTypeBlock, который представляет собой метаданные - список полей и типов данных для каждого поля в рамках таблицы, чтобы не дублировать эту информацию в таблице. Содержит еще количество полей и отступы от начала файла на 1-й и последний блоки, относящиеся к данной таблицы (необходимо для поиска данных по таблице, а последний блок нужен для вставки данных в конец)

```
typedef struct {
   bool isActive;
   char tableName[MAX_LENGTH_TABLE_NAME];
   NameTypeBlock nameTypeBlock[MAX_FIELDS];
```

```
    uint8_t fieldsNumber;
    uint64_t firsTableBlockOffset;
    uint64_t lastTableBLockOffset;
} TableOffsetBlock;
```

DefineTablesBlock создается 1 на файл, он располагается в начале в нем хранятся все метаданные о созданных таблицах необходимы для дальнейшей работы, а именно количество таблиц, массив всех tableOffsetBlock-ов и отступ от начала файла где начинается пустое место (emptySpaceOffset необходим для того чтобы обрезать файл по нему и для того чтобы быстро аллоцировать новый блок)

```
typedef struct {
    uint32_t countTables;
    TableOffsetBlock tableOffsetBlock[MAX_TABLES];
    uint64_t emptySpaceOffset;
} DefineTablesBlock;
```

Iterator - необходим для того, чтобы выполнять перечисление данных - это дает оптимальное потребление оперативной памяти и позволяет доставать объекты по их необходимости
Он содержит в себе указатель на массив структур predicate и количество элементов в этом массиве (поскольку мы выбираем элементы из таблицы с учетом условий Iterator должен о них знать, чтобы уметь выполнить метод hasNext()), текущую позицию в блоке(currentPositionInBlock, поскольку для каждой EntityRecord есть свой RecordId по которым мы итерируемся, каждый раз сдвигаясь на следующий RecordId мы читаем по нему EntityRecord из блока), отступ от начала файла(blockOffset), количество полей в EntityRecord и указатель на массив структур NameTypeBlock (это необходимо для того чтобы правильно прочитать данные из файла, нам нужно знать что хранится в сущности)

```
typedef struct {
    Predicate *predicate;
    uint8_t predicateNumber;
    uint16_t currentPositionInBlock;
    uint64_t blockOffset;
    uint8_t fieldsNumber;
    NameTypeBlock *nameTypeBlock;
} Iterator;
```

Основные методы взаимодействия с базой данных

```
void insertRecordIntoTable(
        FILE *file, EntityRecord *entityRecord,
        const char *tableName
);
```

Метод необходимый для вставки данных в таблицу, принимает указатель на файл в котором содержится таблица, указатель на сущность, и имя таблицы в которую требуется вставить элемент.

```
Iterator *readEntityRecordWithCondition(
        FILE *file,
        const char *tableName,
        Predicate *predicate,
        uint8_t predicateNumber
);
```

Метод необходимый для перечисления данных, с учетом фильтров, которые представляют собой массив predicate-ов, если требуется перечислить все элементы, то

необходимо сделать так, чтобы условие предикатов всегда выдавало бы true, самый банальный метод как это можно сделать - это передать в качестве predicateNumber - 0, тогда вообще никакие условия учитываться не будут.

```
void deleteRecordFromTable(
        FILE *file,
        const char *tableName,
        Predicate *predicate,
        uint8_t predicateNumber
);
```

Метод необходимый для удаления данных, с учетом фильтров, которые представляют собой массив predicate-ов. Принимает указатель на файл, имя таблицы, массив предикатов и их количество.

```
void updateRecordsFromTable(
        FILE *file,
        const char *tableName,
        Predicate *predicate,
        uint8_t predicateNumber,
        EntityRecord *entityRecord
);
```

Принимает то же самое, что и delete и указатель на сущность, которая де факто представляет собой массив обновленных полей.

Вспомогательные методы

```
EntityRecord *readRecord(
        FILE *file,
        uint16_t idPosition,
        uint64_t offset,
        uint16_t fieldsNumber
);
```

Метод необходимый непосредственно для считывания сущности по метаданным, принимает offset - абсолютный отступ от начала файла до начала блока, в котором требуется прочитать сущность, количество полей, которые надо считать, и позиция id-шника в рамках блока (напоминаю структура с id-шниками взята из postgres-a).

```
void insertRecord(
        FILE *file,
        EntityRecord *entityRecord,
        TableOffsetBlock *tableOffsetBlock
);
```

Метод для вставки сущности, принимает указатель на сущность и указатель на tableOffsetBlock, в котором хранится метаинформация о таблице, которая помогает быстро вставить сущность в эту таблицу

Публичный интерфейс для взаимодействия с базой данных

```
void writeEmptyTablesBlock(FILE *file);

DefineTablesBlock *readTablesBlock(FILE *file);

uint32_t readTablesCount(FILE *file);

uint64_t readEmptySpaceOffset(FILE *file);
```

```
void writeTableCount(FILE *file, uint32_t tablesCount);

void writeEmptySpaceOffset(FILE *file, uint64_t offset);

TableOffsetBlock *readTableOffsetBlock(FILE *file, uint16_t tablePosition);

uint64_t findOffsetForTableOffsetBlock(FILE *file);

void writeTableOffsetBlock(FILE *file, TableOffsetBlock *tableOffsetBlock);

void insertRecord(FILE *file, EntityRecord *entityRecord, TableOffsetBlock *tableOffsetBlock);

EntityRecord *readRecord(FILE *file, uint16_t idPosition, uint64_t offset, uint16_t
fieldsNumber);

void insertRecordIntoTable(FILE *file, EntityRecord *entityRecord, const char *tableName);

Iterator *readEntityRecordWithCondition(FILE *file, const char *tableName, Predicate
*predicate,
                                        uint8_t predicateNumber);

void deleteRecordFromTable(FILE *file, const char *tableName, Predicate *predicate,
                           uint8_t predicateNumber);

void rebuildArrayOfRecordIds(unsigned char *buffer, RecordId *recordIdArray, uint8_t
recordsNumber,
                             uint16_t positionToDelete, uint64_t deletedRecordLength);

void updateRecordFromTable(FILE *file, const char *tableName, Predicate *predicate,
                           uint8_t predicateNumber, EntityRecord *entityRecord);

TableOffsetBlock *findTableOffsetBlock(FILE *file, const char *tableName);

void optimiseSpaceInFile(FILE *file);

void deleteTable(const char *tableName, FILE *file);

FieldValue **separateString(FieldValue *fieldValue, uint32_t capacity);

FieldValue *concatenateFieldValues(FieldValue *fieldValue1, FieldValue *fieldValue2);

EntityRecord **separateEntityRecord(EntityRecord *entityRecord, int64_t capacity,
                                    uint8_t fieldsNumber, NameTypeBlock *nameTypeBlock);

EntityRecord *compoundEntityRecords(EntityRecord *entityRecord1, EntityRecord *entityRecord2,
uint8_t fieldsNumber);
```

Описание основных операций интерфейс для взаимодействия с базой данных

Работа с таблицами:
В начале работы нашей системы необходимо вызвать функцию writeEmptyTablesBlock(), эта функция помимо создания метаинформации в начале файла создает еще метатаблицу, которая хранит в себе блоки, которые стали пустыми в результате удаления других таблиц или удаления сущностей из этих блоков, дабы их переиспользовать потом.

Для выполнения любой операции необходимо прежде всего найти соответствующий tableOffsetBlock, который хранится в начале файла, поиск этой структуры данных происходит по имени таблицы, так как по сути это и есть высокоуровневая информация, которую мы ожидаем от пользователя. После нахождения этого блока в зависимости от операции мы уже итерируемся по блокам, которые принадлежат таблице.

При удалении таблицы, tableOffsetBlock.isActive = false и этот блок становится можно использовать для создания новой таблицы. Также все блоки принадлежавшие таблице

удаляются и отступы до этих блоков записываются в мета таблицу, чтобы их можно было впоследствии переиспользовать

Операция вставки происходит следующим образом:

Если элемент влезает в текущий блок, то он вставляется, если он превышает оставшееся место в блоке, то происходит разделение сущности в том случае если осталось какое-то разумное место, от 200 байт, разделение сущности происходит либо по полю, то есть часть полей хранится в старом блоке, часть хранится в следующем, либо происходит по строке, строка делится на 2, вся вспомогательная информация для того чтобы собрать сущность обратно хранится в структуре данных linkNext - это своего рода ссылка на оставшуюся часть сущности, там хранится то на каком поле произошло разделение, ссылка в виде отступа для блока и позиции id-шника в этом блоке для оставшейся сущности.

Операция удаления происходит в два этапа:

1. Создается итератор, который в себе хранит условия и де факто осуществляет поиск элемента проходящего под условия, после чего останавливается как только находит элемент.
2. Элемент удаляется беря всю необходимую метаинформацию из итератора. Происходит полная перестройка блока, в котором хранилась данная сущность, перестройка id-шников, которые хранятся в конце блока, в итоге блок выглядит так, будто в нем этой сущности и не было (ее наличие никак не повлияло на другие). Если в блоке остается 0 элементов, то мы добавляем этот блок в метатаблицу для того, чтобы при необходимости аллоцировать новый блок сначала обратиться в метатаблицу

Операция обновления состоит из двух операций:

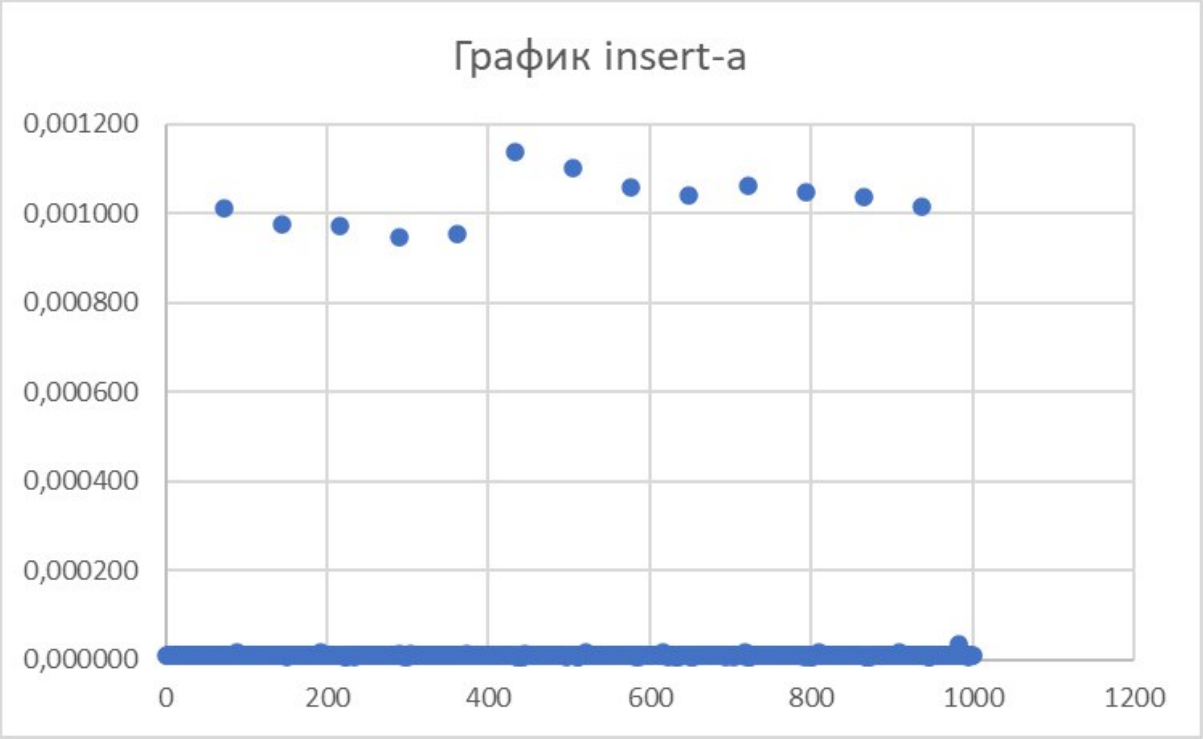
1. Удаление элемента
2. Создание нового

Выборка элементов происходит при помощи итератора. При вызове метода has_next() происходит поиск необходимого элемента и сохранение его в памяти и счетчик итератора увеличивается. Таким образом потребление оперативной памяти имеет асимптотику O(1), так как в любой момент времени храним только один элемент и вспомогательные данные, которые не меняются, это позволяет вытаскивать по 1-му элементу и хранить по сути структуру данных, которая в любой момент может выдать элементы в соответствии с набором предикатов.

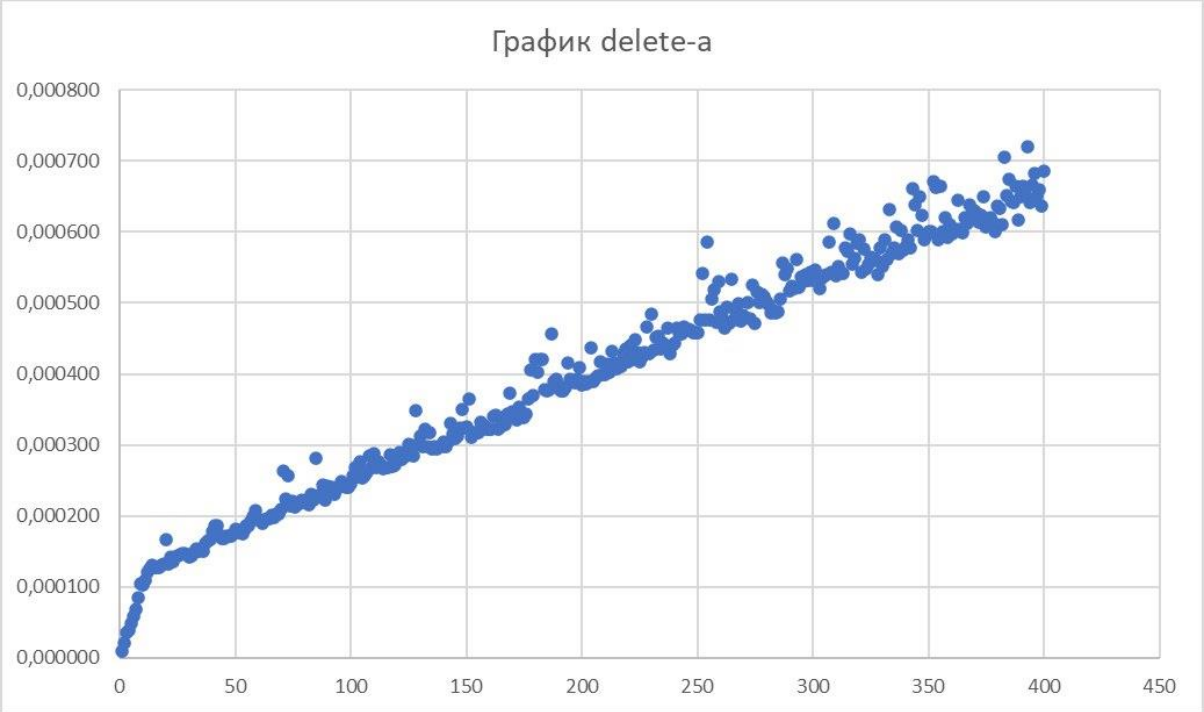
Амортизированные показатели ресурсоемкости

Как проводились тесты:
Были созданы функции, которые проверяют performance основных функций и записывали данные в соответствующий файл (каждая функция в свой файл). Далее данные копировались в excel и строились точечные диаграммы по этим данным.

Вставка элементов: по вертикали – время выполнения вставки для 1-го элемента, по горизонтали количество элементов в файле, которой находится на момент вставки. Есть выбросы связанные с аллоцированием нового блока, это доказывает их периодичность, так как раз в сколько-то инсертов аллоцируется новый блок. Исходя из графика видно, что операция insert выполняется за O(1), то есть не зависит от количества данных в таблице.



Удаление элементов. по вертикали время в миллисекундах, по горизонтали - количество сущностей в таблице на момент операции удаления. Исходя из графика видно, что операция delete выполняется за O(n + m), где n - количество сущностей в файле на момент удаления, m - количество удаленных сущностей



Обновление элементов (по вертикали время, по горизонтали количество сущностей в таблице в которой происходит обновление). Приведено 2 графика для 200 и 400 сущностей, видно, что операция обновления выполняется за O(n + m) где n - количество данных в таблице на момент обновления, m - количество затронутых сущностей

Вставка 200 элементов, удаление 100. (По вертикали время для вставки 200 элементов, по горизонтали порядковый номер итерации)`

Сценарий теста (обсуждался на паре):
1. Инсерт 500 сущностей
2. Delete 400 сущностей (по 1-й сущности вызовом верхнеуровневого метода удалять)
3. Замерить время для для каждых 100 вставок и каждых 100 delete-ов и построить графики

График insert-а (по вертикали время для 100 штук, по горизонтали итерация замера)

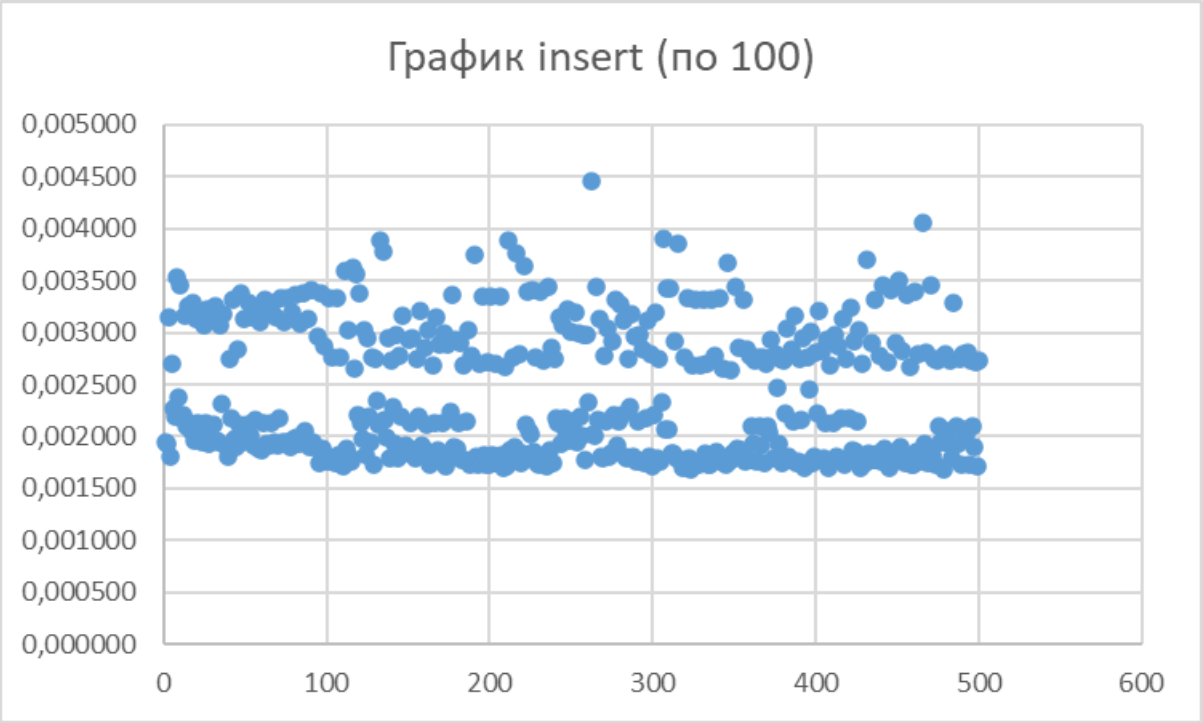
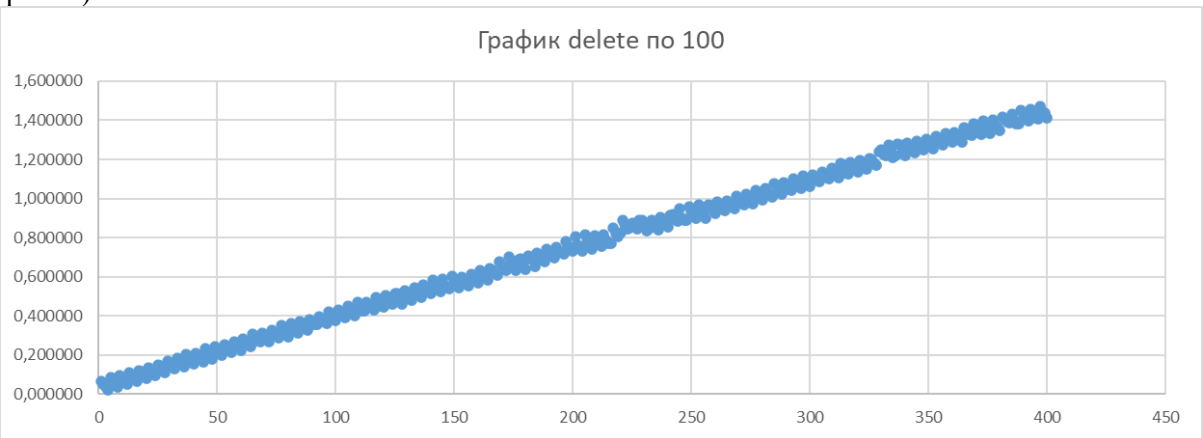


График delete-а (по вертикали время для 100 штук, по горизонтали итерация замера). Видно, что время для delete-а растет линейно (O(n) где n - это количество данных в файле)



Зависимость размера файла от количества элементов (file.bin)

123392256 байт (0.1149 гб) - 1 000 000 сущностей

```
~/CLionProjects/lab1/test 〉main ?5  ls -l
total 129584
-rw-r--r--. 1 iwaa0303 iwaa0303   9296664 Nov 20 23:18 data.bin
-rw-r--r--. 1 iwaa0303 iwaa0303 123392256 Nov 20 23:46 file.bin
-rw-r--r--. 1 iwaa0303 iwaa0303       882 Oct 29 22:04 local.c
-rw-r--r--. 1 iwaa0303 iwaa0303       246 Oct 20 22:46 local.h

~/CLionProjects/lab1/test 〉main ?5
```

3432616304 (3,1968) байт > 28 000 000 сущностей

```
~/CLionProjects/lab1/test 〉main ?5  ls -l
total 3361248
-rw-r--r--. 1 iwaa0303 iwaa0303    9296664 Nov 20 23:18 data.bin
-rw-r--r--. 1 iwaa0303 iwaa0303 3432616304 Nov 20 23:43 file.bin
-rw-r--r--. 1 iwaa0303 iwaa0303        882 Oct 29 22:04 local.c
-rw-r--r--. 1 iwaa0303 iwaa0303        246 Oct 20 22:46 local.h

~/CLionProjects/lab1/test 〉main ?5
```

Результаты

- В ходе выполнения работы была изучена организация данных в postgres по страницам, и попытка реализовать подобное в более упрощенном виде.
- Пришло осознание того, что необходимо брать из имеющихся решений, а что нет (потому что как говорили в начале курса, не стоит брать под копирку все что читаете, поскольку перед нами не стоят те ограничения и задачи, что стояли перед разработчиками того же postgres-а)
- Были продуманы структуры данных необходимые для представления вспомогательных сущностей
- Была разработана архитектура базы данных, определены способы взаимодействия с файлом.
- Были реализованы основные методы работы с данными
- Создан интерфейс для взаимодействия

Выводы

Если хорошо продумать архитектуру разрабатываемой системы в начале, то это поможет в будущем избежать таких проблем как дублирование кода, плохую переносимость на другие платформы. По моему мнению концепция предложенная для решения этой задачи в начале курса хорошо помогла впоследствии при решении проблем, с которыми пришлось столкнуться в процессе реализации задачи, а именно разделить все на блоки, дабы лучше ориентироваться в файле и обкладывать его метаданными для поиска необходимых сущностей и организации их в хранилище.