Университет ИТМО
Факультет ПИиКТ

Низкоуровневое программирование

Лабораторная работа №1

Работу выполнил:
Абузов Ярослав

Группа:
P33302

Вариант:
Документное дерево

Преподаватель:
Кореньков Ю. Д.

Санкт-Петербург
2022

1) Цель

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

2) Задачи

- Спроектировать структуры данных для представления информации в оперативной памяти для порции данных и для информации о запросе

- Спроектировать устройство файла и способы реализации базовых операций, способы переиспользования пространства

- Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним: операции над схемой данных, вставка элемента данных, перечисление элементов данных, обновление элемента данных, удаление элемента данных

- Используя в сигнатурах только структуры данных из п.1, реализовать публичный интерфейс со следующими операциями над файлом данных: добавление, удаление и получение информации об элементах схемы данных; добавление нового элемента данных; выборка набора элементов данных с учётом заданных условий и отношений со смежными элементами данных; обновление и удаление элементов данных, соответствующих заданным условиям

- Реализовать тестовую программу для демонстрации работоспособности решения

- Построить графики с полученными тестовыми данными

3) Описание работы

Модуль состоит из 5 частей (директория zgdb):

1. data – компонент отвечает за определение основных структур для представления данных, для их обработки (список результатов), для их перечисления (итераторы) + в этом же компоненты определны структуры для формирования запросов поиска

2. format – компонент отвечает за инициализацию, создание и закрытие файла базы данных

3. index – компонент отвечает за работу с индексами, а также со «списком» свободных блоков (см. аспекты реализации)

4. schema – компонент отвечает за работу со схемой для документа

5. zdbc.c – основной компонент, в нём представлены функции для конечного пользователя для работы с базой

В директории tests предствлены тесты. Те, что в директориях - использовались во время разработки (половина уже не актуальна) для проверки соответствующей имени части программы. finalTest – итоговый тест для замера производительности (описан в разделе с графиками).

Описание функций публичного интерфейса для работы с данными:

1) *zgdbFile* init(const char* path);*
   bool finish(zgdbFile file);*

Функции для открытия (или создания, если файла нет) файла базы данных.

2) *createStatus createDocument(zgdbFile* file, const char* name,*
   documentSchema schema, document parent);*

Функция для создания документа с определенным именем и схемой и привязки его к родителю. Перед созданием необходимо получить родительский документ.

3) *updateElementStatus updateElement(zgdbFile* file, document doc, char* key, char* input);*

Функция для обновления свойства (или элемента, или аттрибута, это одно и тоже). Необходимо указать имя свойства и новое значение. Если элемента с таким именем не существует или тип невозможно преобразовать к целевому у элемента, то будет возвращена соответствующая ошибка.

4) *void deleteDocument(zgdbFile* file, document doc);*

Функция для удаления документа и все его потомков.

5) *void printDocumentElements(zgdbFile* file, document document);*

Фунукция для вывода в консоль всех элементов и их значений из конкретного документа.

6) *resultList join(zgdbFile* file, document parent);*

Функция для поиска всех детей документа.

7) *findIfResult findIfFromRoot(zgdbFile* file, path p);*

Функция поиска по заданным условиям. Возвращает список документов или список элементов. Информация о том, что было возвращено, хранится в возвращаемой структуре.

8) *void forEachDocument(zgdbFile* file, void (* consumer)(document, zgdbFile*), document start);*

Аналог цикла foreach для документов. Принимает некоторую функцию-consumer, которая будет применена к каждому документу.

9) *document getRootDocument(zgdbFile* file);*

Получить корневой документ.

Описание функций публичного интерфейса для работы со схемой:

1) *documentSchema initSchema(size_t capacity);*
void destroySchema(documentSchema schema);*

Функции для создания и удаления схемы. Схема – список элементов документа и их начальных значений.

2) *addStatus addIntToSchema(documentSchema* schema, char* key, int32_t initValue);*
addStatus addDoubleToSchema(documentSchema schema, char* key, double initValue);*
addStatus addBooleanToSchema(documentSchema schema, char* key, uint8_t initValue);*
addStatus addTextToSchema(documentSchema schema, char* key, char* initValue);*

Функции для расширения схемы. Если элемент с таким именем существует или количество элемнтов превышено максимальное количество, то будет возвращена соответствующая ошибка.

Описание функций для работы с итераторами:

1) *documentIterator createDocIterator(zgdbFile* file, uint64_t order, uint64_t orderParent, uint64_t startDepth);*
void destroyDocIterator(documentIterator iterator);*

Функции для создания и удаления итератора документов. Итератор будет ходить по всем документам данного уровня, пока не обойдет все документы на данном уровне глубины, и затем спустится ниже и снова пойдет обход уровня. Итератор работает итеративно (да, звучит не очень), а не рекурсивно, что позволяет не потреблять огромное количество памяти (по сути потребление O(1) т.к. за раз загружается один документ).

2) *bool hasNextDoc(documentIterator* iterator);*
document nextDoc(zgdbFile file, documentIterator* iterator, uint64_t* depth);*

Функции для проверки наличия и для получения следующего документа. Здесь также можно получить текущую глубину погружения (начальная задается при создании итератора).

3) *elementIterator createElIterator(zgdbFile* file, document* doc);*
void destroyElIterator(elementIterator iterator);*

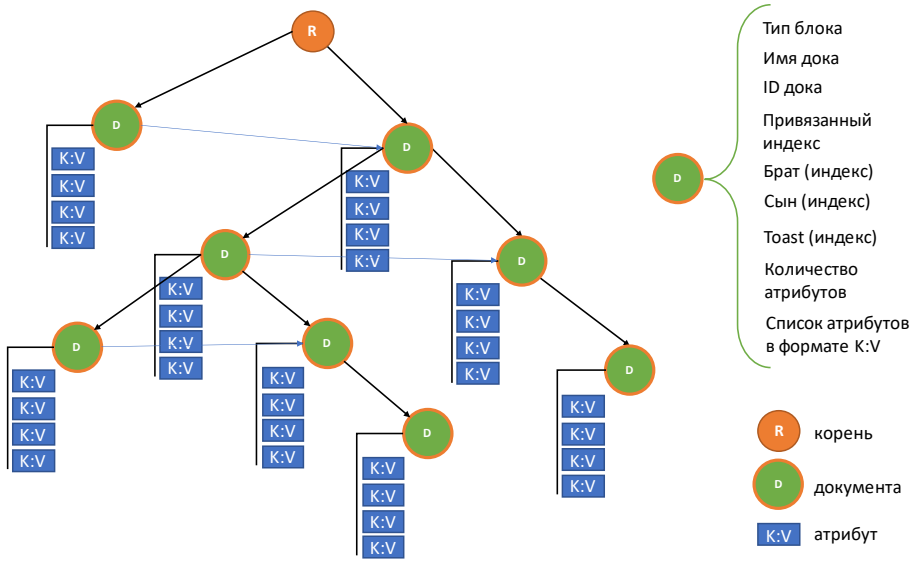
Функции для создания и удаления итератора элементов. Итератор читает данные в буфер и при запросе следующего «собирает» элемент, определяет его позицию в файле (необходимо для, например, обновления) и возвращает эту информацию. Если следующий элемент за пределами буфера, то буфер обновляется. Размер буфера фиксирован и содержит только некоторое количество элементов (т.е. не все элементы, что позволяет итератору работать с константным потреблением памяти).

4) *bool hasNextEl(elementIterator* iterator);*
elementEntry nextEl(zgdbFile file, elementIterator* iterator, bool reqData);*

Функции для проверки наличия и для получения следующего элемента. Можно указать, нужно ли содержимое элемента (если нет, то вернется только позиция в файле и сам элемент без значения), это может быть необходимо, например, во время обновления и т.д.

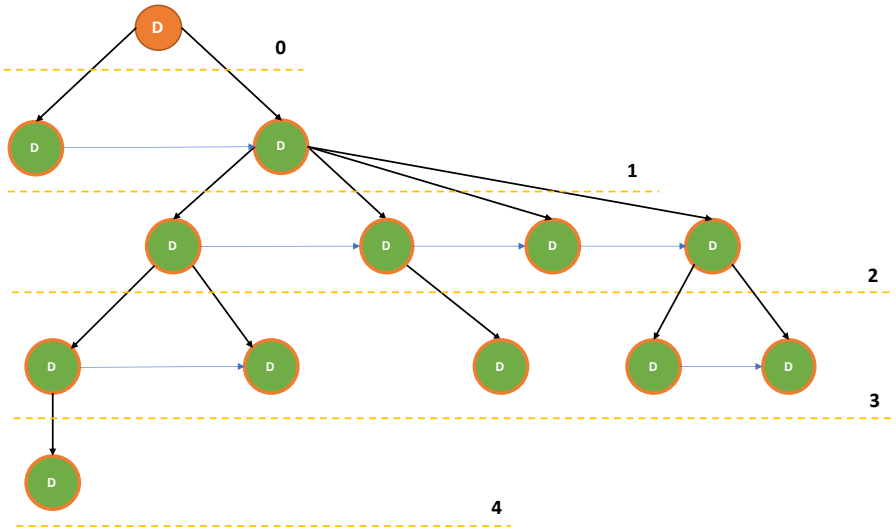
4) Аспекты реализации

Абстрактное представление базы данных:

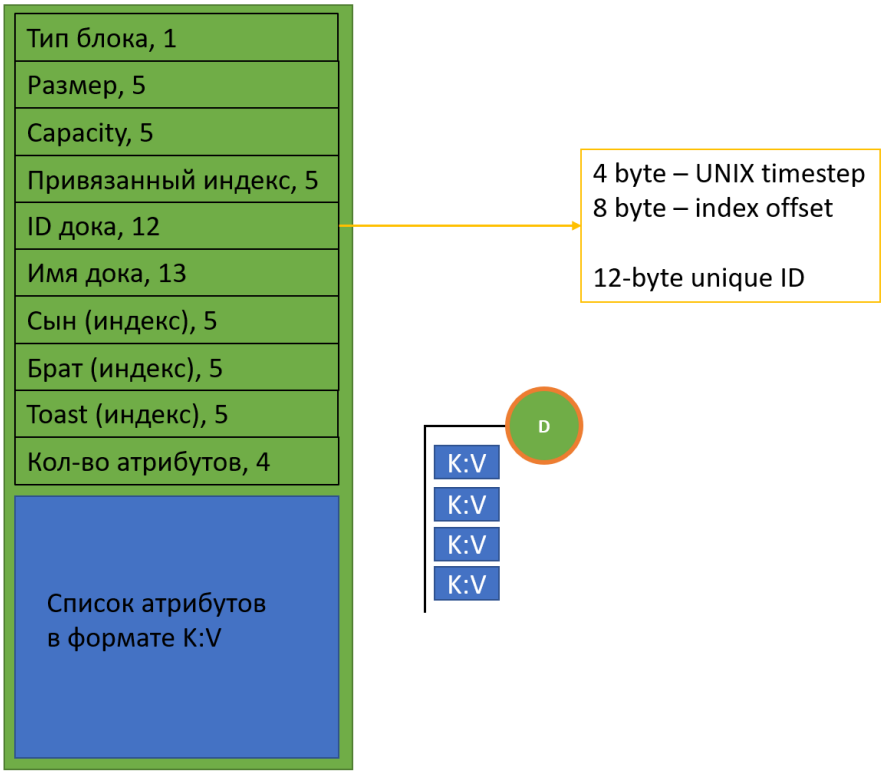


База данных представляет собой дерево общего вида, в узлах которого находятся документы. Документы имеют имя, уникальный id и список элементов (свойств или атрибутов, это одно и тоже). Список элементов предствлен в виде пары ключ-значение. Существует 4 типа элементов: целое 32-битное число, логическое значение, дробное 64-битное число, строка переменной длины.

Корневой элемент создается при первом создании базы и не может быть удален или изменён. Каждый документ может иметь дочерние документы. У каждого документа есть внутренние поля – ссылка на первого сына и ссылка на брата (об организации ссылок и связей ниже). По сути – потомки одного документа представлены связным списком. В runtime еще можно получить ссылку на родителя и информацию о том, является ли документ корневым. В самом документе в файле информация о родителе не хранится т.к. это бы создало проблемы со связностью. Также есть внутренее поле – ссылка на первый toast блок, это необходимо для хранения строк, об этом ниже. (Господи, зачем я сделал эти toast блоки, можно было просто инлайнить в док и переносить его при увеличении длины; крик души)



Устройство документа:



На рисунке представлено устроство документа и его layout в файле. В начале идёт заголовок документа с необходимыми полями (число справа от названия поля – его размер в байтах). Из интересного – тип блока нужен для различия документа и toast при их переносе в другое место (об этом ниже), capacity показывает фактически занимаемое пространство блоком, size показывает текущий размер блока.

Пару слов о том, как генерируется уникальный id – первые 4 байта id это время в форме UNIX timestamp, а след. 8 байт это смещение до документа от начала файла (эта информация используется в индексах, об этом позже). Соответсвенно, если в одно время создать несколько докуентов, то их смещение будет по определению разным. Если создать позже еще несолкьо документов, то у них по сравнению с предыдущими будет другое время создания, а если смещение и совпадет с прошлыми (из-за переиспользования пространства), то проблем это не создаст из-за другого времени создания.

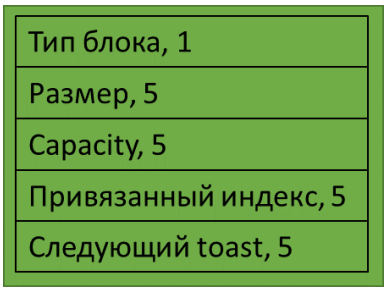
Элементы идут «в ряд» после заголовка документа. Элементы задаются в формате тип-ключ-значение.

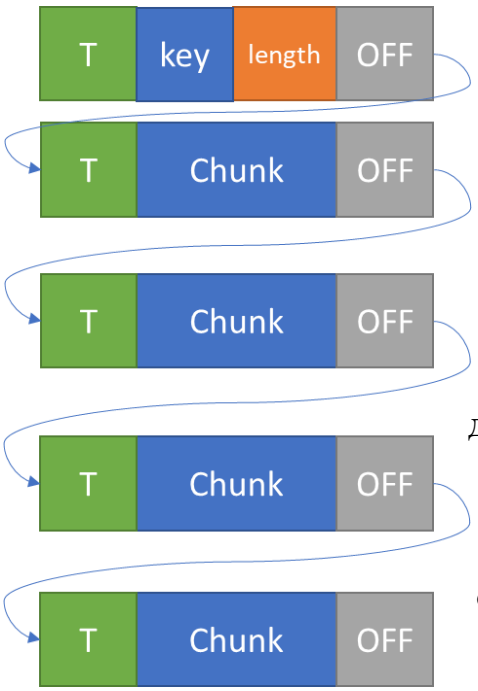
Исключение из этого правила – строчки, у них вместо value хранится мета-информация, об этом ниже. Тип – двухбайтовое число, ключ ограничен 13 байтами (обусловлено средним размером слова в английском), размер значения зависит от типа.

Принцип работы со строчками:

Строка бьется на чанки фиксированного размера, затем эти чанки сохраняются в специальных блоках – toast. Каждый чанк содержит адрес следующего чанка в формате toast-смещение. Toast блоки в свою очередь хранят информауию о максимальном размере блока (capacity), о текущем использовании пространства в количестве живых чанков и ссылку на следующий toast блок. Внутри документа хранится длина строки и смещение внутри привязанного к документу toast.

Адресация работает так: имеем ссылку на toast, если это текущий, то ничего не делаем, если же нет – переходим в toast по ссылке. Далее с помощью значения смещения находим внутри toast необходимый чанк. Таким образом и совершаем проход и сбор/обновление строки.



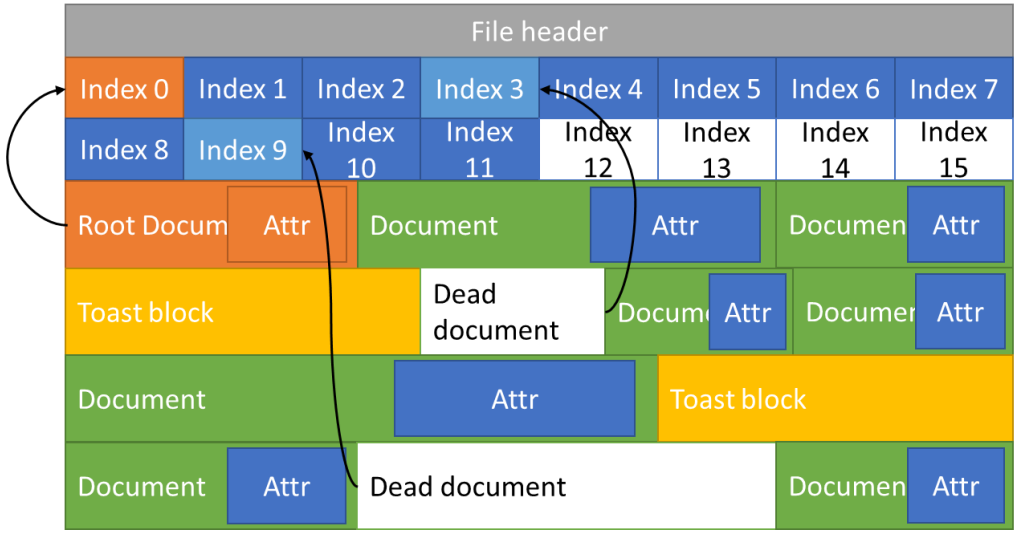


Алгоритм обновления строчек:

- Если длина такая же, то просто перезапись всех связанных чанков на новое содержимое.
- Если длина меньше, то перезаписываем все чанки до необходимой длины и в последний чанк ставим смещение 0 и текущую ссылку на toast (это все есть признак последнего чанка, хотя его можно и посчитать исходя из длины).
- Если длина больше, то записываем все доступные чанки до упора и если в текущем toast есть место, то создаем новые чанки в нём. Если места в текщем недостаточно, то создаём новый с необходимым количества места и пишем оствшиеся чанки в него.

P.S. Да, можно было сделать просто inline в документ и перемещать его, если места не хватает, но мне казалосб это странным вариантом, но уже поздно…

Адресация документов внутри файла и организация связи родитель-ребёнок:



Вся адресация строится на т.н. индексах. Они расположены сразу после заголовка и перед всеми документами. Они содержат информацию о состоянии (флаг) пространства документа и смещение до этого документа. Нумерация индексов сквозная и начинается с 0. Нулевой индекс всегда привязан к корневому документу. В документах есть информация о привязанном к ним индексе (двухсторонняя связь для удобства). Кстати, если брата или ребенка нет, то значение 0 (т.к. этот индекс только у корня).

У индекса есть три возможных флага (один байт):

- NEW – индекс ни к чему не привязан и ни разу не использовался, смещение 0, значение по умолчанию после создания индекса
- ALIVE – индекс привязан к существующему документу, смещение до привязанного документа, занчение после привязки индекса к документу
- DEAD – индекс привязан к удаленному документу, смещение до привязанного документа, может быть переиспользован (об этом ниже)

Соответсвенно, все «ссылки» являеются номерами индексов и что не делай с самим документом или toast – перемещай или удаляй, индексы свое положение не меняют и, соответственно, по ним всегда можно найти необходимый блок. Если необходимо переместить документ или toast (зачем это может быть нужно ниже), то достаточно обновить информацию о смещении в индексе и все зависящие от этой информации сущности всегда будут иметь актульную информацию, что очень сильно упрощает многие операции и в целом работу с данными.

Основной минус такой системы – необходимость расширения индексов, иначе свободные индексы просто закончатся. Когда необходимо расширить индексы, происходит следующие операции:

- Подсчёт необходимого места для новых индексов
- Начиная с первого после индексов документа, перемещаем документы в конец или в свободное место в файле (и обновляем инормацию об их смещении в индексах), пока не получим необходимое свободное место. Если получили больше, то не проблема – дополнительное пространство также будет исопльзовано для индексов.
- Далее создаем пачку индексов от конца последнего индекса в файле.
- Profit, можно работать дальше.

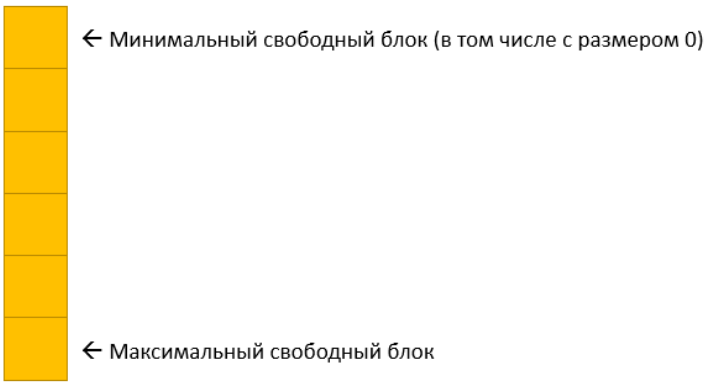
Эту проблему можно решить, создавая индексы в файле в любом месте, но для этого необходимо немного изменить орагнизацию файла и скорее всего сделать ее блочной, в которой один блок это по сути то, что есть сейчас у меня. Но это уже на будущее.

Заголовок файла:

Тип файла, 4
Версия, 1
Количество индексов, 5
Пространство между блоком и индексом, 1
Размер файла, 8
Количество документов, 8

Ничего интересного из себя не представляет, хранит служебную информацию. Отмечу только пространство между первым блоком и последним индексом – необходимо для корректоного опредления первого докуенмта после всех индексов. Вызвано это тем, что при доавблении индексов может остаться пространство меньшее, чем размер индека.

Переиспользование пространства при добавлении документа:



Для реализации переиспользования пространства внутри файла используется следующая структура – отсортированный связный список, который в хвосте хранит максимальный свободный блок и номер его индекса, а в голове все новые индексы (с флагом NEW) или минимальный свободный блок и номер его индекса.

Таким образом, при добавлении проверяем – есть ли в хвосте свободный блок, размер которого больше или равен размеру нового документа. Если это так, то переиспользуем это пространство для нового документа. Иначе же пишем новый документ в конец, перед этим взяв из головы свободный индекс.

При удалении блока добавляем в эту структуру информацию о его размере и привязанном к нему индексе.

Создание нового документа:

Документ создается их схемы, где указаны его элементы (их имена, типы и начальные значения). Схема не определяет тип документа и явно не хранится, но может быть восстановлена (но зачем?). При создании мы находим подходящий индекс (см. пункт выше про переиспользование) и пишем по соотв. смещению заголовок документа, а за ним все элементы. Индекс помечаем как ALIVE.

Удаление документа:

Обращаемся к родителю (эта информация доступна в runtime, искать родителя дополнительно не нужно) и отвязываем данный документ от него. Если он является первым ребенком, то заменяем ссылку на первого ребенка на брата удаляемого документа. Если документ не является первым ребенком, то заменяем в предыдущем ребенке ссылку на его брата на брата удаляемого документа.

Затем удаляем всех потомков (и потомков потомков и т.д., т.е. все поддерево потомков) удаляемого документа, если они есть. Далее удаляем цепочку toast блоков, если они есть. И наконец удаляем сам документ.

При самом удалении происходит очень простая операция – индекс помечается как DEAD (что можно скзать моментально) и необходимая информация падает в список свободных блоков.

Обновление документа:

Ничего необычного – создается итератор элементов по целевому документу и, проходя по всем элементам, находим нужный и обновляем его. Примитивы обновляются полной перезаписью знчения, про строчки сказано выше. Если нужный элемент не найден или обнолвяемое значение не соответствует типу элемента, то будет возвращена ошибка.

Поиск документа:

Самое интересное. Для поиска используется итератор документов, которому можно задать начальный документ. Затем по условиям будут найдены необходимые документы или элементы (да, так тоже можно). Запрос строится по подобию языка XPath (здесь мы его назовем ZPath, потому что мой ник Zavar30). Т.е. зпрос состоит пути до документа/элемента. А сам путь состоит из шагов. У шагов есть имя (документа или элемента) и может быть цепочка предикатов. Контекст – текуший набор документов, относительного которых нужно искать.

Шаги бывают двух видов по контексту:

1) Абсолютный – ищет из текущего контекста строго в пределах одного уровня глубины
2) Относительный – ищет из текущего контекста по всему поддереву данного контекста.

Шаги бывают двух видов по типу данных:

1) Документный шаг – ищет в рамках документов, поддерживает предикаты, может быть промежуточным шагом
2) Элементный шаг – ищет в элементах текущего контекста (т.е. во всех документах текущего контекста), не поддерживает предикаты и не

может быть промежуточным шагом. По сути нужен для того, чтобы уже на отсортированных документах вернуть результат сразу в виде элемента, а не в виде документа.

Предикаты поддерживают следующие связки:

1) Логическое И
2) Логическое ИЛИ.

Предикаты могут быть двух видов:

1) Предикат по номеру документа – выбирает n-ый документ из контекста
2) По значению элемента документа – указывается ключ элемента, оператор сравнения и значение.

Операторы сравнения:

1) =
2) !=
3) >
4) <
5) >=
6) <=
7) contains (только для строк)

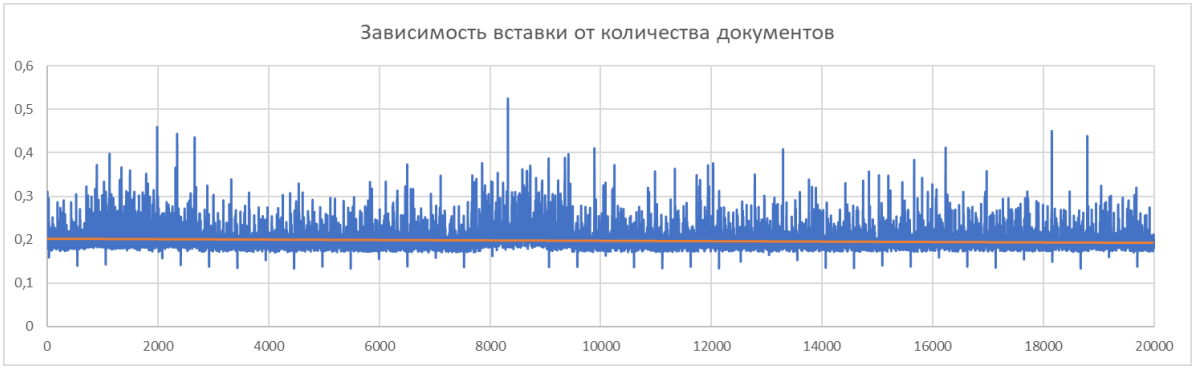
Для строк и логических типов можно применить операции = и !=.

Join:

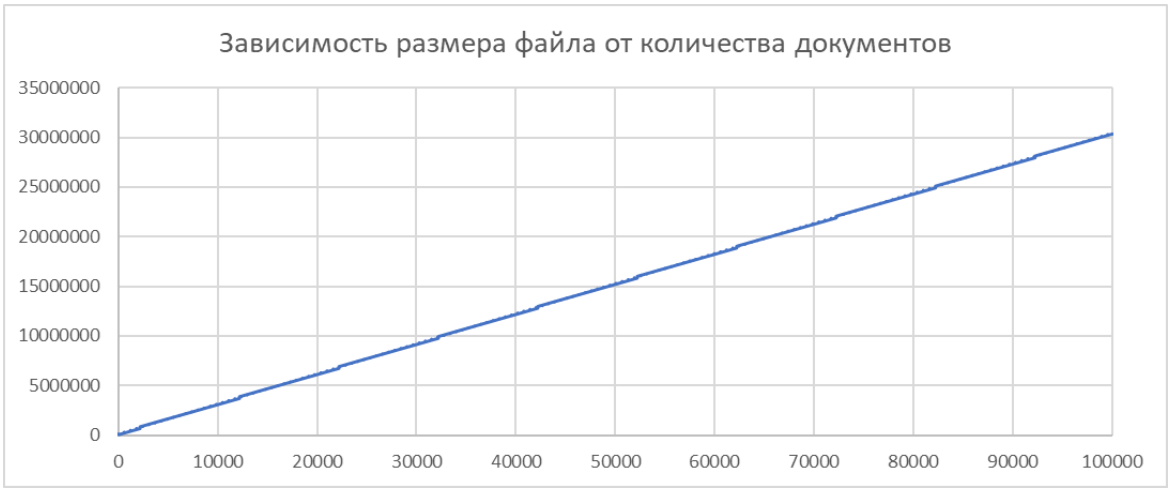
Получает список всех детей, просто идёт начиная с первого ребенка до конца списка детей (т.е. пока не встретит 0 в ссылке). Надо будет добавить сюда возможность установить условия на детей.

5) Полученные результаты

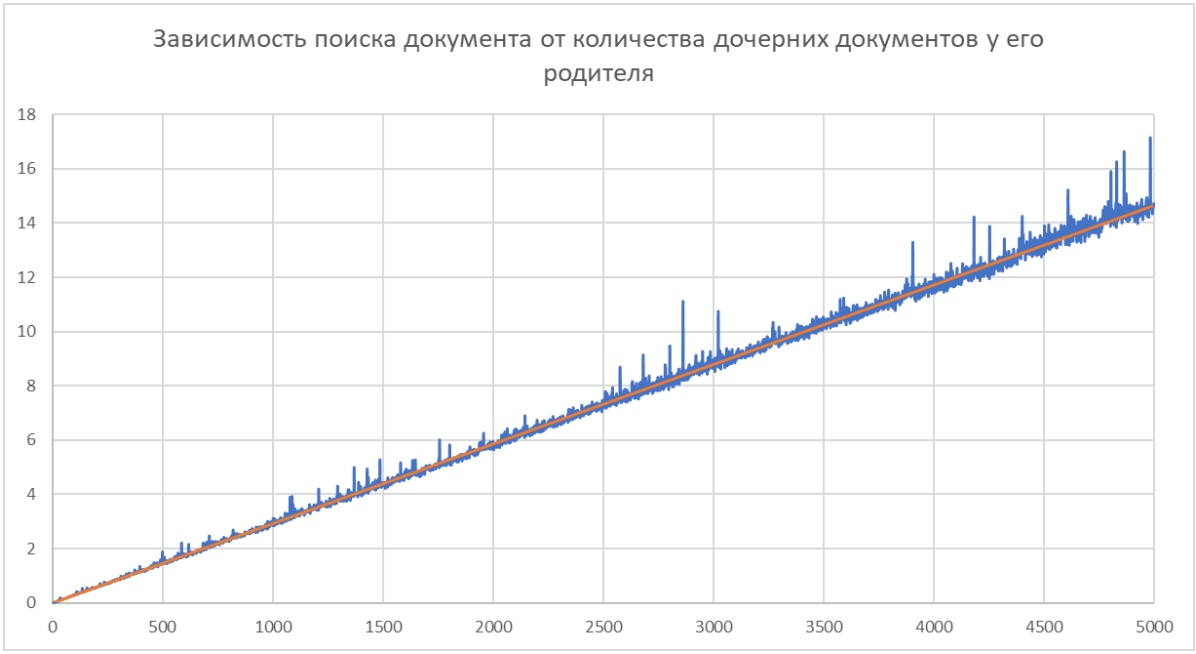
Создан модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное обновление) информации общим объёмом от 10GB документного дерева. Реализованы необходимые структуры для представления данных и реализованы операции для работы с этими структурами. Измерена производительность и построены графики:



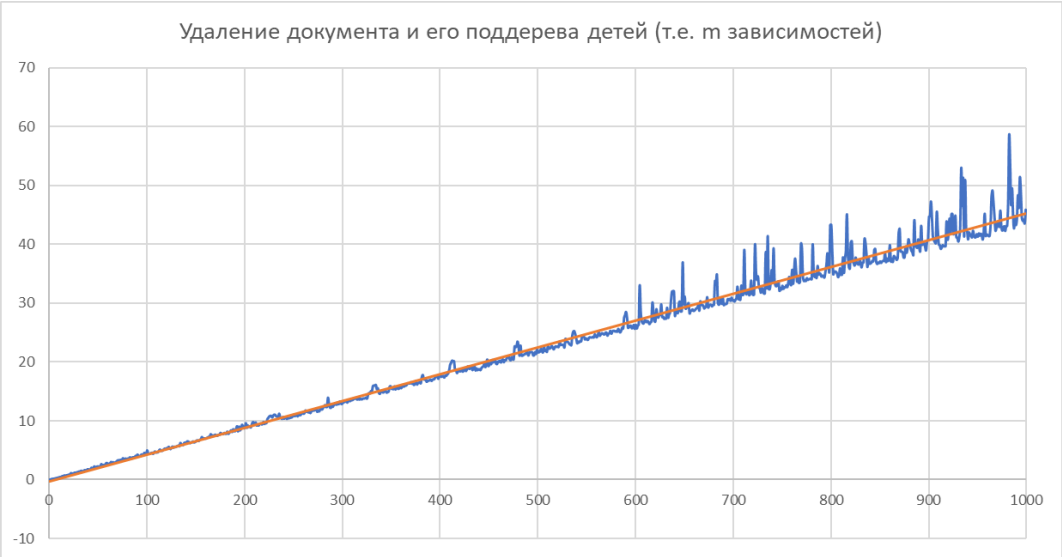
По оси X – количество документов, по Y – время в мс. *Как в требованиях – O(1).*



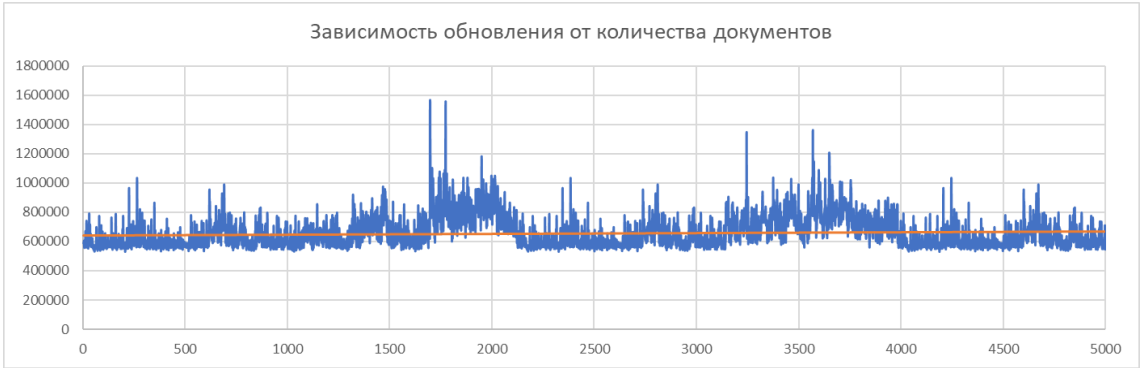
По оси X – количество документов, по Y – размер файла в байтах. *Как в требованиях – всегда пропорционален.*



По оси X – количество дочерних документов, по Y – время в мс. *Как в требованиях – O(n).*



По оси X – количество дочерних документов (m штук), по Y – время в мс. Для удаления документа его необходима сначала найти, что делаем за O(n), согласно предыдущему графику. А для самого удаления имеем O(m). *Поэтому итоговая сложность соответствует требованиям – O(n*m) > t –> O(n+m).*



По оси X – количество документов, по Y – время в мс. Обновление происходит за O(1), что связано с принципами организации адресации и с работой итератора элементов, что неудивительно (именно так и планировалось). *Поэтому итоговая сложность O(n) т. к. сначала нужно найти документ, что не превышает требования O(n*m) > t –> O(n+m).*

По оси X – количество документов, по Y – память. Во время вставки потреблените памяти не зависит от количества документов, что *соответствует требованиям в O(1).*

6) Выводы

В результате лабораторной работы мною был разработан и реализован на языке Си модуль, который позволяет хранить в файле базу данных в виде документного дерева и который поддерживает различные операции над документами. Работа заняла много времени т. к. необходимо было все продумать, а потом только писать. В процессе крафта архитектуры я изучил много статей, порылся во многих исходниках или описаниях БД (SQLite, MongoDB, PostgreSQL) и узнал много нового не только для создания данной работы. Наконец-то написал полноценную программу на Си (до этого были очень простые вещи или без особо вникания в нюансы т. к. этого не требовалось). Работа понравилась, хоть было и сложно, но считаю, что она заставляет хорошо подумать и развивает навыки проектирования программных систем.