Национальная научно-образовательная корпорация ИТМО
ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

Лабораторная работа №1

по дисциплине
«Низкоуровневое программирование»

Вариант №5

Выполнил: Лебедев
Вячеслав Владимирович
Группа № P33312
Проверил:
Кореньков Юрий Дмитриевич

Санкт-Петербург
2023

Цели

Создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное
обновление) информации общим объёмом от 10GB в виде реляционных таблиц.

Решение

Программа состоит из библиотеки и тестов. Библиотека реализует работу с файлом и предоставляет интерфейсы для взаимодействия.

Тесты проверяют корректность работы библиотеки и ее соответствие требования производительности, размера файла и потребления памяти.

Структура библиотеки

Библиотека состоит из 6 модулей:

1) allocator – отвечает за непосредственную работу с файлом (открытие/закрытие файла, создание/очищение файловых отображений), предоставляя единый интерфейс для взаимодействия с файлами в ОС Windows и Linux.
2) util – модуль, содержащий вспомогательные структуры, функции для работы с ними и определения: map (хэш-таблица), list (двухсвязный список), vector (динамический массив), а также макросы для обработки ошибок.
3) buffer – модуль, содержащий структуру байтового буфера и многочисленные функции работы с ним (чтение, запись, создание, удаление).
4) heap – модуль для работы со структурой heap (связным списком файловых страниц) для хранения записей фиксированного размера, а также функции для работы с ним: чтение, запись поверх списка, присоединение новых страниц, удаление старых.
5) pool – модуль для работы с пулом структур heap для хранения байтовых записей произвольного размера: в пуле хранятся ссылки на 57 heap, имеющие размеры, кратные степеням 2, от 64 до 2^{63} байт, а также функции, для: вставки записи, итерации и чтения по пулу записей, удаления записи.
6) database – основной модуль для работы с файловым хранилищем: создание и удаление таблиц, вставка/выборка/удаление/обновление записей в таблицах.

Структура файла

Весь файл делится на страницы размером 4096 байт.

В начале файла хранится заголовок со следующими полями:

1) magic – магическое число 0xABCD

2) free_pages_count – число свободных страниц, не использующихся в данный момент

3) free_pages_next – указатель на голову связного списка свободных страниц

4) entrypoint – указатель на первую страницу, содержащую полезные данные

В моем случае entrypoint – это указатель на пул байтовых записей, содержащий метаинформацию о таблицах (их название, схема, указатель на данные таблицы).

heap – структура, представляющая собой абстракцию для работы со связным списком страниц. Позволяет производить запись и чтение байтовых массивов фиксированного для данного heap размера.

Так как записи могут иметь размер, больший размера страницы, то реализация учитывает это при чтении и записи, производя запись по частям на разные страницы (чтение аналогично).

Обоснование

Причиной необходимости данной структуры является то, что в файле (монотонном массиве байт) нужно хранить записи разных типов, число которых меняется во времени. Метаинформация о таблицах, данные Таблицы №1 и Таблицы №2 – это записи разных типов. Если хранить такие записи друг за другом в файле, то их обход будет занимать время, пропорциональное числу записей всех типов. Поэтому хочется хранить записи разных типов отдельно друг от друга. Можно было бы реализовать связный список из записей одного типа, тогда обход занимал бы линейное относительно числа записей этого типа время, но, с другой стороны, нарушалась бы локальность – записи одного типа лучше хранить по соседству. Поэтому было принято решение хранить записи в виде связного списка, но не отдельных записей, а фиксированных страниц, размером 4096 байт. Это улучшает локальность, а также делает переиспользование освобожденной памяти проще – все страницы одинакового размера.

Теоретически хранить записи разного размера в heap можно, но в таком случае удаление записей внутри структуры приводит к проблеме дефрагментации и необходимости поиска решений для борьбы с ней, а также к проблемам переиспользования освобожденной памяти. Если хранить внутри heap записи одного размера, то «дыры» внутри структуры закрываются, переносом записей с конца. За счет этого все освобожденное после удаления пространство остается в конце и может быть легко переиспользовано.

pool – структура, представляющая собой пул из 57 heap для хранения записей разных размеров.

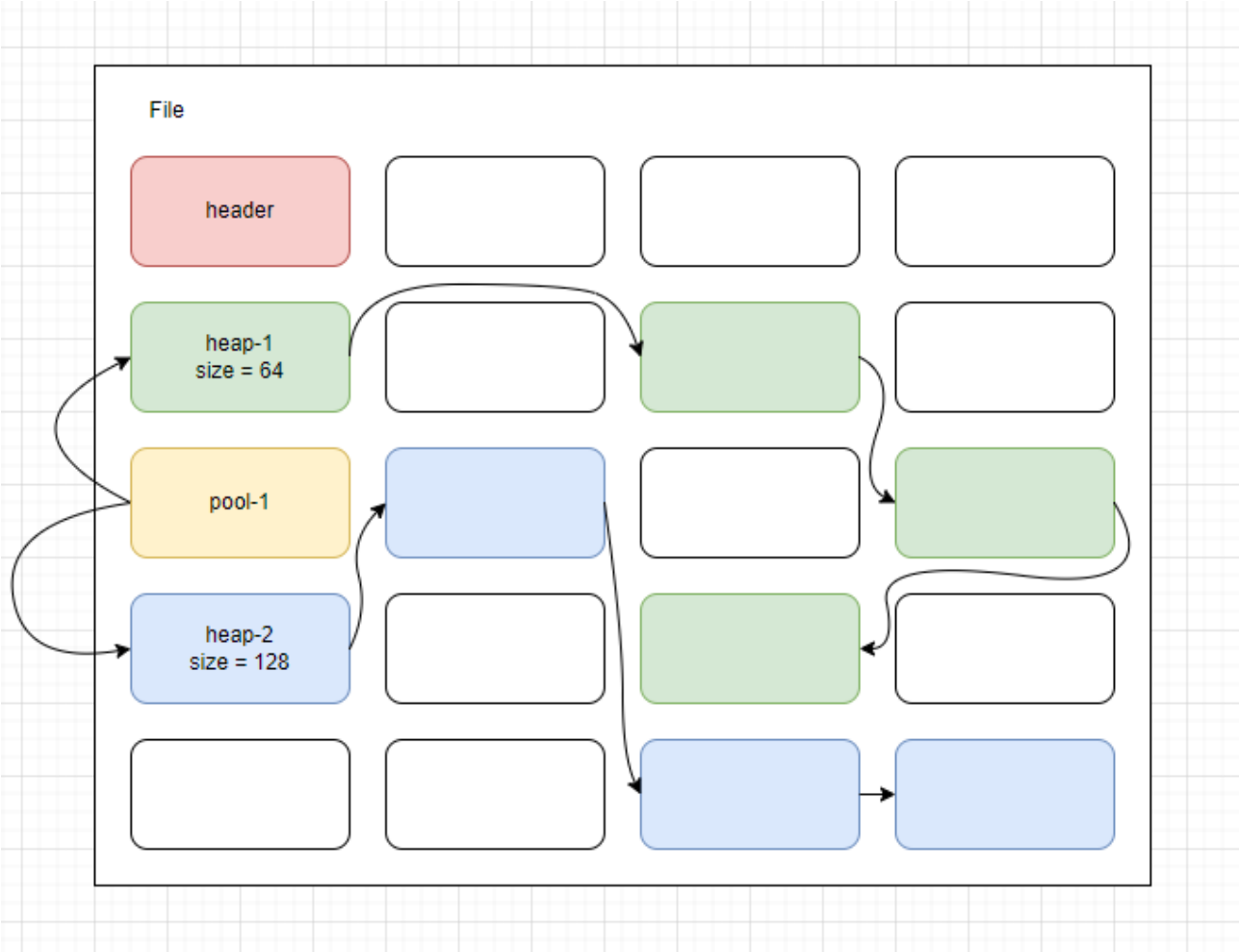
Обоснование

Записи (схемы таблиц или отдельные записи в самих таблицах) после сериализации могут иметь произвольный размер: число колонок в схеме таблицы произвольно, длина строк, используемых в названии таблицы, названиях ее колонок и записях внутри таблиц с колонкой типа «строка» также произвольны.

В связи с этим возникает потребность хранить байтовые записи произвольного размера, но heap предоставляет возможность хранить записи только фиксированного для данного heap размера.

Решением проблемы является пул, который хранит структуры heap, имеющие размер хранимых записей кратный степени 2. При необходимости сохранить запись размера N в пул, будет подобрана минимальная необходимая структура heap, способная вместить запись такого размера. В худшем случае, когда N = $2^k + 1$, эта запись попадет в heap размера 2^{k+1}, заняв в ~2 раза больше места, чем необходимо. Таким образом, для хранения произвольного множества записей в худшем случае будет потрачено в два раза больше памяти, чем необходимо. Это не идеально, но такой подход позволяет решить многие проблемы, в том числе дефрагментацию.

Схема внутреннего устройства файла



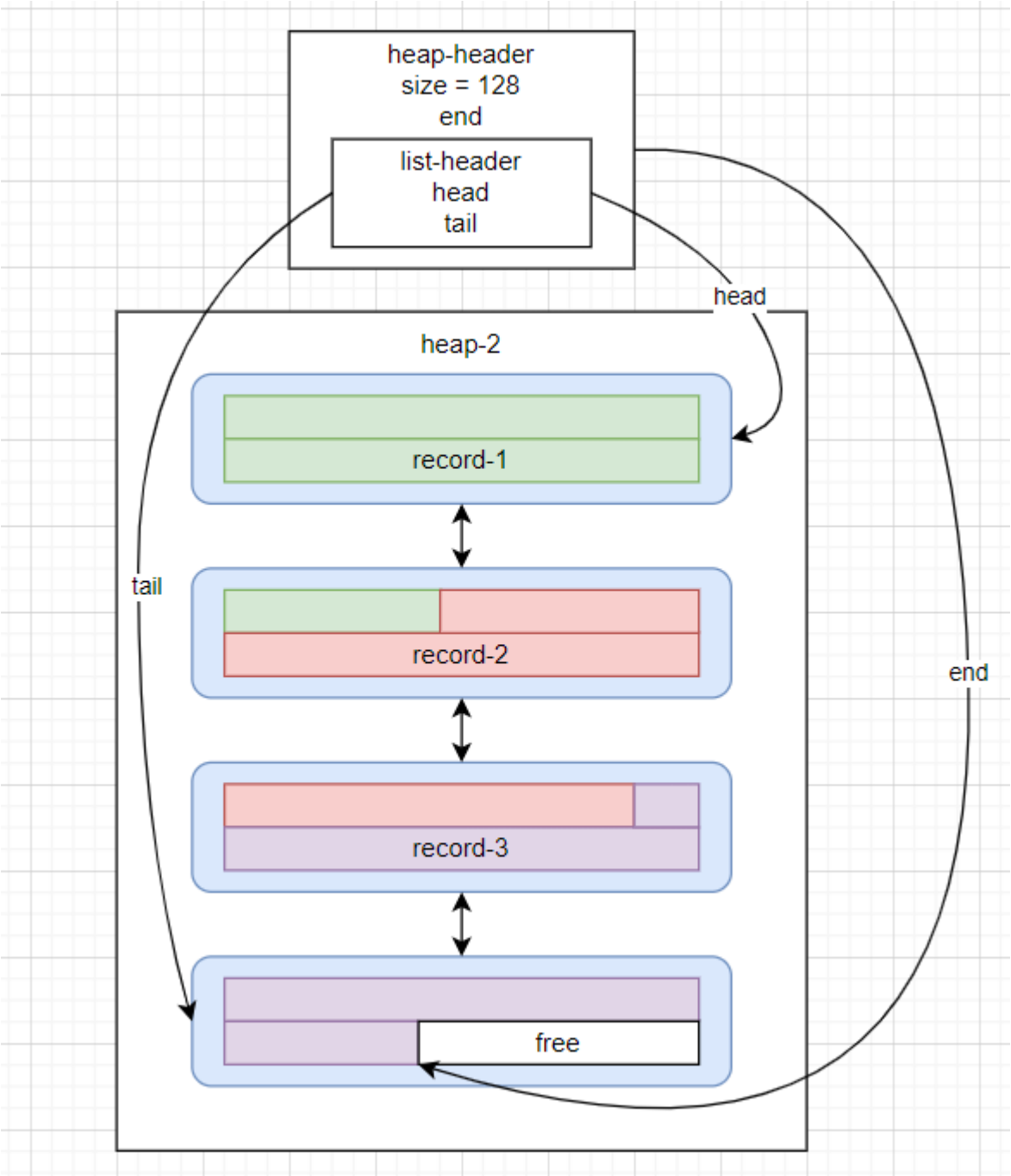
* остальные heap (размером больше 128 байт) опущены для схематичности

* белые страницы – свободные страницы, связи между связным списком свободных страниц опущены

* ссылки на входную страницу и голову связного списка свободных страниц из заголовка файла опущены

* заголовок и метаинформация heap лежит отдельно от самого списка с данными, в следующей схеме – подробнее

Более подробная схема heap



* для схематичности конкретное расположение страниц (узлов связного списка) внутри файла упрощено

head – указатель на первую страницу в списке

tail – указатель на последнюю страницу в списке

end – указатель на конец последней записи, куда впоследствии будут дописываться новые записи

Тесты

Для тестов будет использоваться следующая таблица

```
scheme_builder_t scheme_builder = scheme_builder_init( name: "test");
scheme_builder_add_column( builder: scheme_builder,  name: "int",  type: COLUMN_TYPE_INT);
scheme_builder_add_column( builder: scheme_builder,  name: "string",  type: COLUMN_TYPE_STRING);
scheme_builder_add_column( builder: scheme_builder,  name: "bool",  type: COLUMN_TYPE_BOOL);
scheme_builder_add_column( builder: scheme_builder,  name: "float",  type: COLUMN_TYPE_FLOAT);
database_create_table( database: db,  table_scheme: scheme_builder_build( builder: scheme_builder));
scheme_builder_free( builder: &scheme_builder);
```

Название таблицы – test, а также 4 колонки – по одному на каждый поддерживаемый тип данных.

Данные для вставки генерируются случайно, используя программу на языке Python:

```
def random_string(n: int) -> str:
    return ''.join(random.choice(string.digits + string.ascii_letters) for _ in range(n))


1 usage   new *
def random_int32() -> int:
    return random.randint(-2147483647,  b: 2147483646)


1 usage   new *
def random_bool() -> int:
    return random.choice([0, 1])


1 usage   new *
def random_float() -> float:
    return round(random.uniform(-1,  b: 1), 7)


1 usage   new *
def random_input() -> str:
    return f"{random_int32()} {random_string(32)} {random_bool()} {random_float()} "
```

Как несложно заметить, генерируемая строка имеет длину 32 символа и состоит из случайного набора цифр и латинских буквенных символов. Таким образом, общий размер записи для данной таблицы 42 байта: 33 байта (с учетом нуль-терминатора) на строку, 4 байта на целочисленный тип, 4 байта на тип с плавающий точкой, 1 байт на булев тип.

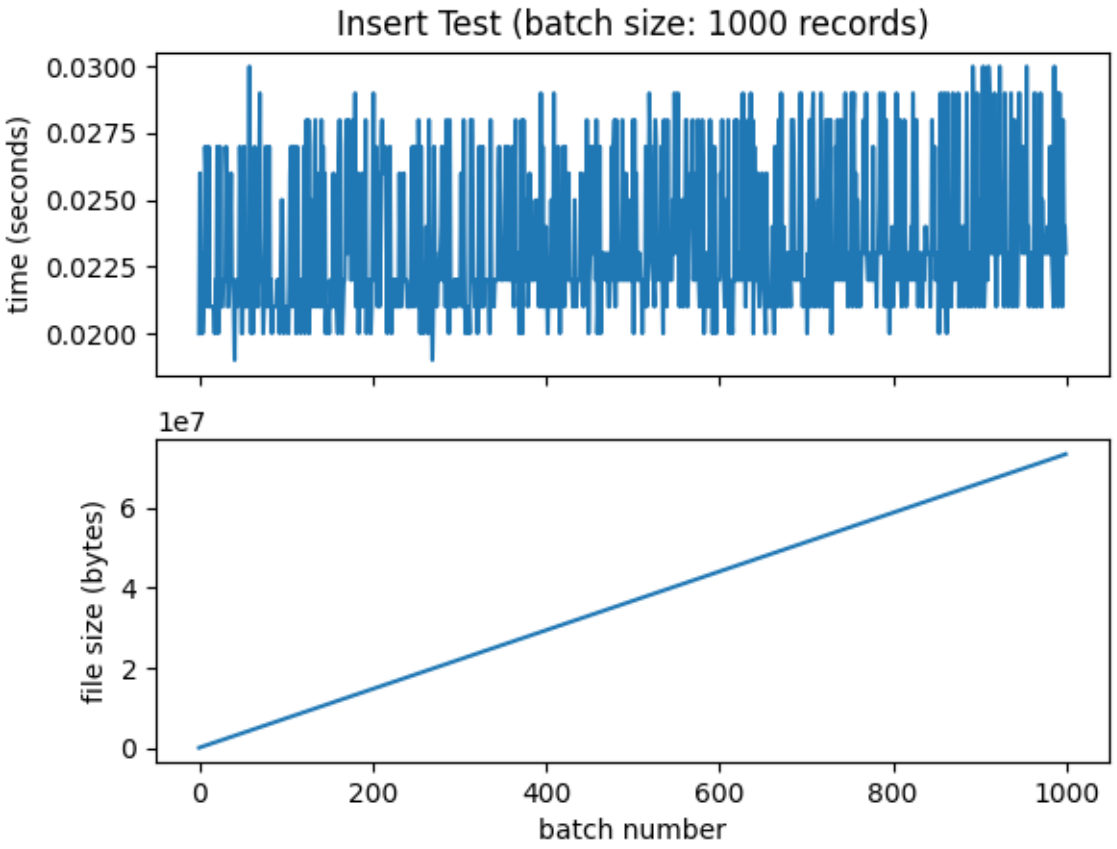
Вставка

Перед тестированием файл очищается, после происходит 1000 итераций тестов: в рамках теста происходит пакетная вставка 1000 записей. Записи генерируются тестирующей программой и передаются на стандартный ввод тестируемой программы.

Замер времени работы производится самой тестируемой программой, для того чтобы измерить непосредственно время работы библиотечной функции, осуществляющей вставку, и исключить время, не относящееся к ней напрямую (открытие и закрытие файла, меж процессное взаимодействие во время получения тестовых данных).

```
clock_t begin = clock();
database_insert( database: db,  name: "test", batch);
clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
printf( format: "%f", time_spent);
```

Результаты теста



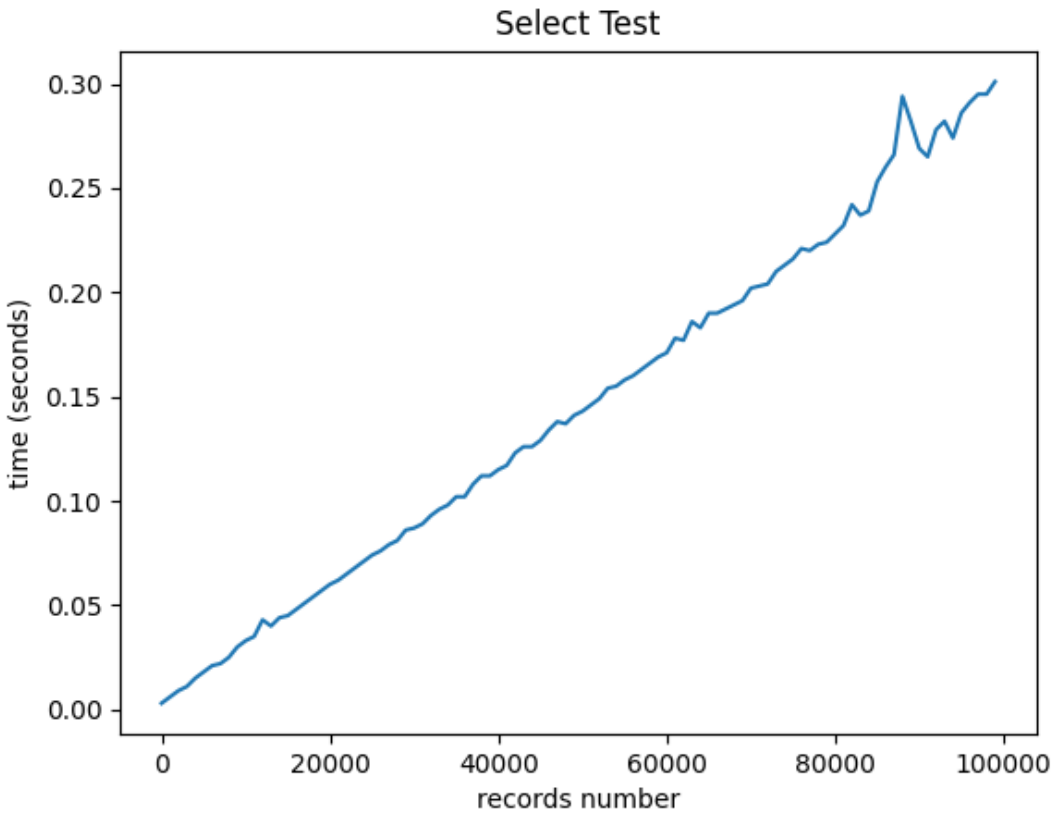
Как можно заметить, время вставки остается константным, а размер файла растет линейно. Итоговый размер файла 73Мб, число записей – 1 миллион. Таким образом, средний объем памяти на одну запись – 73 байт (чистый размер полезных данных – 42 байта).

Выборка

Перед тестированием файл очищается. На каждом тесте добавляется пакет из 1000 элементов, и производится выборка всех элементов таблицы. Всего производится 100 тестов.

Результаты тестирования

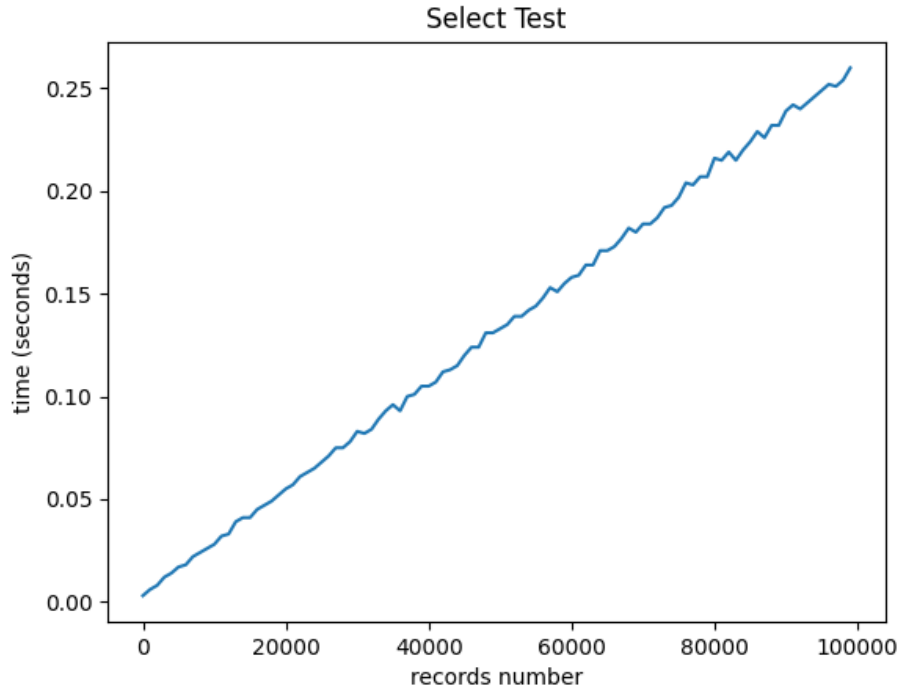
Выборка без условий



Выборка с условием

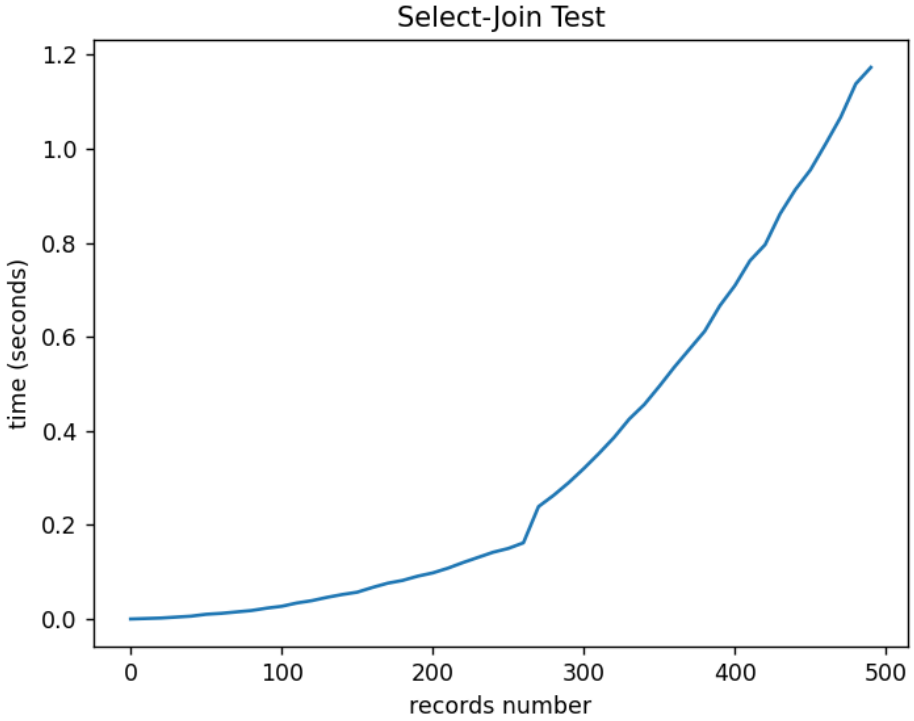
```
.where = where_condition_and(
        first: where_condition_compare(
                type: COMPARE_EQ,
                op1: operand_column( table: "test",  column: "bool"),
                op2: operand_literal_bool( value: true)
        ),
        second: where_condition_compare(
                type: COMPARE_GE,
                op1: operand_column( table: "test",  column: "float"),
                op2: operand_literal_float( value: 0.5f)
        )
),
```

* колонка «bool» должна иметь значение «true», а колонка «float» должна быть не меньше 0.5



Как можно заметить, время выборки с условием и без практически не отличаются, а также имеют линейную сложность от количества элементов таблицы.

Выборка с соединением (одинаковые таблицы, одинакового размера)



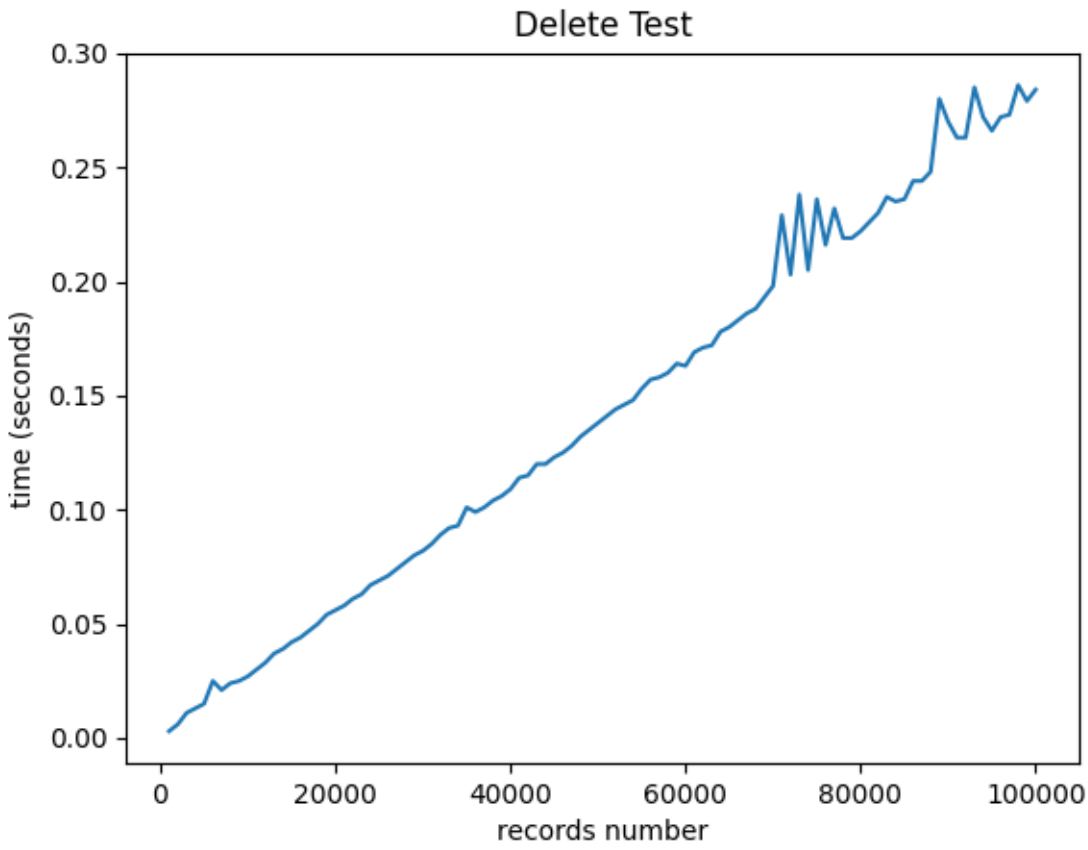
Удаление

В начале каждого теста происходит очищение файла, добавляются случайно сгенерированные записи в количестве $1000 \times i$, где i – номер теста и происходит удаление части из них по условию. Всего проводится 100 итераций тестов.

Удаление происходит по следующему условию:

```
.where = where_condition_compare(
        type: COMPARE_EQ,
        op1: operand_column( table: "test",  column: "bool"),
        op2: operand_literal_bool( value: true)
),
```

* колонка «bool» должна принимать значение «true»



Как можно заметить, скорость удаления по условию линейно зависит от числа записей в таблице.

Обновление

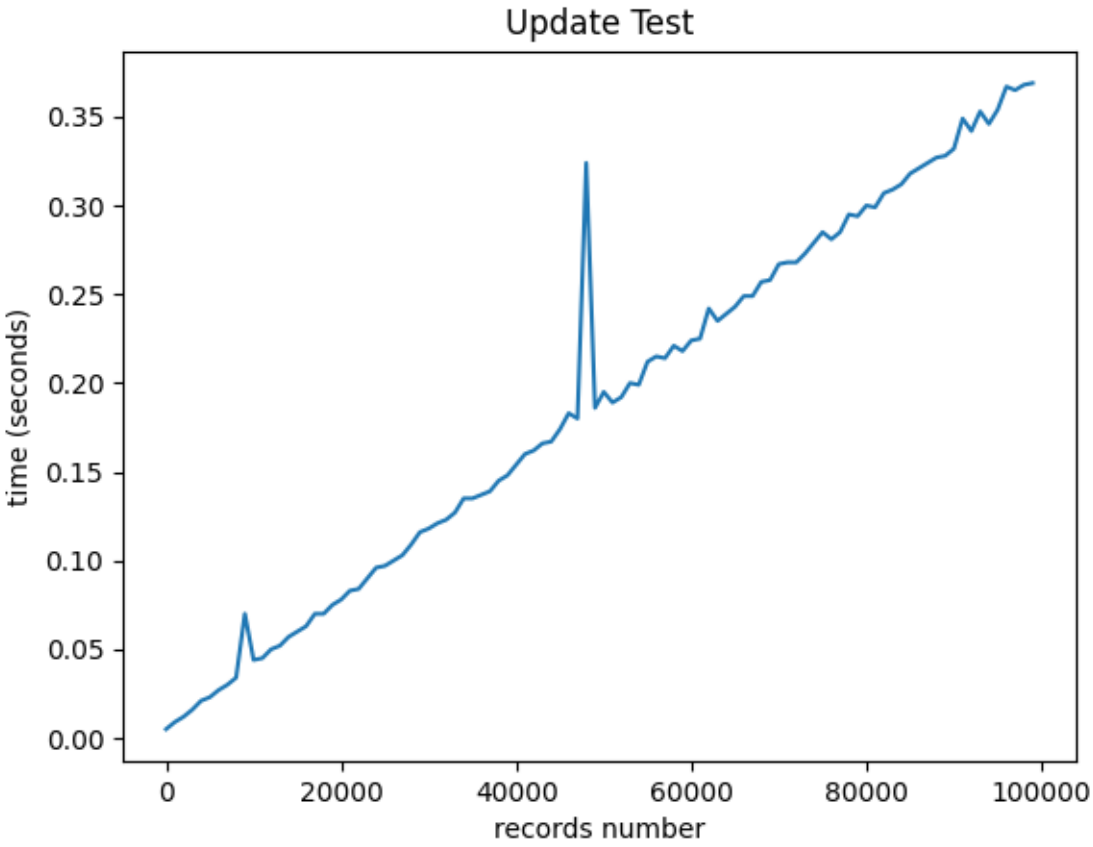
Перед тестированием файл очищается. На каждом тесте добавляется пакет из 1000 элементов, и производится операция обновления по условию. Всего производится 100 тестов.

Условие обновления:

```
.where = where_condition_and(
        first: where_condition_compare(
                type: COMPARE_EQ,
                op1: operand_column( table: "test",  column: "bool"),
                op2: operand_literal_bool( value: true)
        ),
        second: where_condition_compare(
                type: COMPARE_GE,
                op1: operand_column( table: "test",  column: "float"),
                op2: operand_literal_float( value: 0.5f)
        )
),
```

При обновлении будем менять значение столбца «bool» на «false»

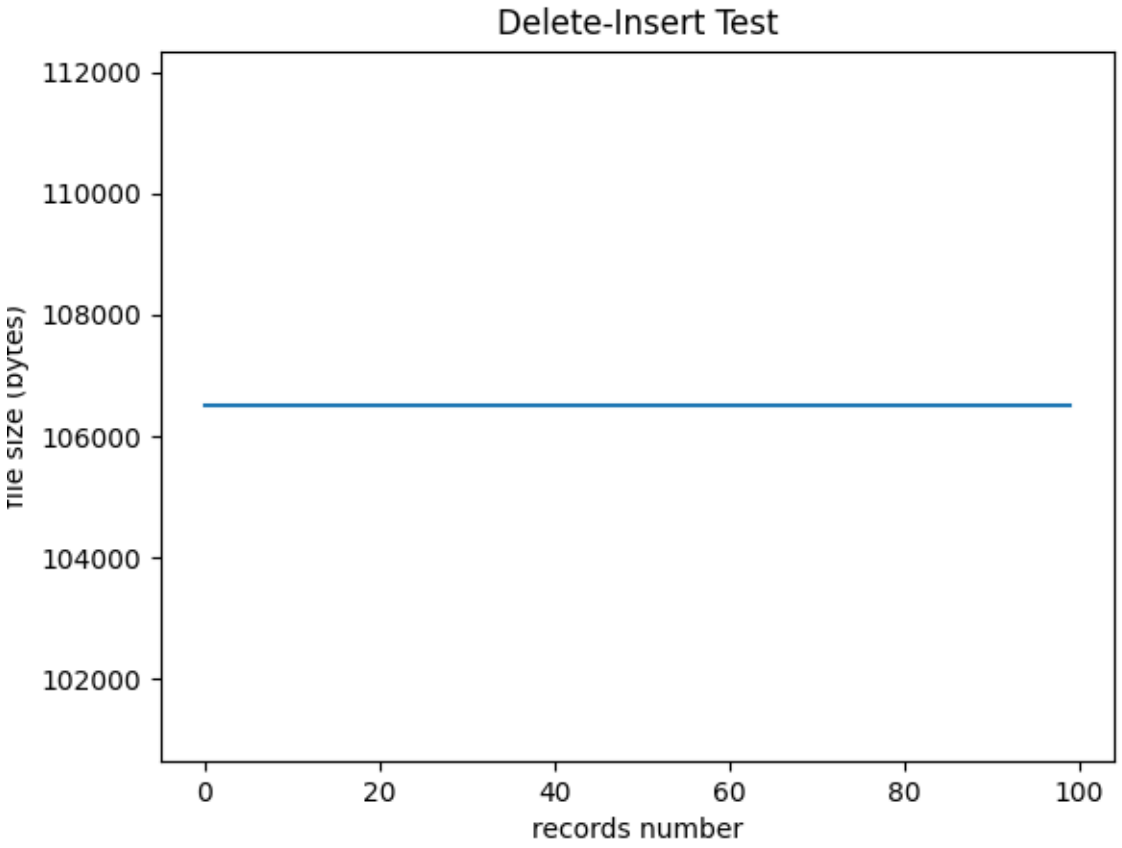
```
updater_builder_add(updater,
                    column_updater_of(
                            target: "bool",
                            new_value: column_bool( value: false)
                    )
);
```



Как можно заметить, скорость обновления линейно зависит от числа записей в таблице.

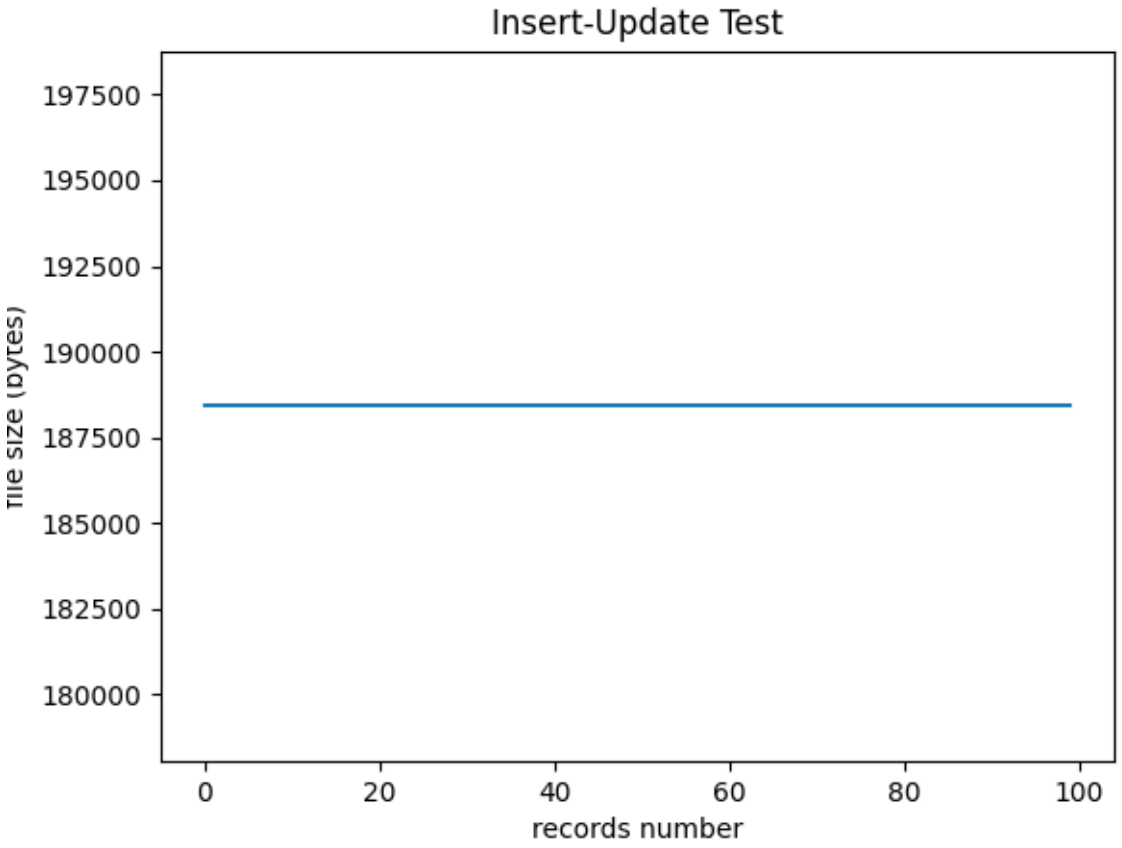
Для того, чтобы убедиться, что размер файла пропорционален количеству размещенных файлов, проведем следующее тестирование: в начале очистим файл, проведем 1000 тестов, в рамках которых будем добавлять 1000 записей и удалять их. Тогда, размер файла должен оставаться одинаковым – место освобожденных записей должны будут занять записи из следующего теста.

Условие удаления оставим тем же, но для того, чтобы добиться удаления 100% записей, будем генерировать их так, чтобы они удовлетворяли условию.



Как можно заметить, размер файла не меняется, а одна запись в среднем занимает 106 байт. Это происходит из-за меньшего числа записей, чем в предыдущих тестах, при фиксированном объеме метаинформации.

Проведем аналогичный тест с обновлением записей: очистим файл, произведем вставку 1000 записей и в цикле будем обновлять их 1000 раз.



Как можно заметить, объем файла увеличился, по сравнению с предыдущим тестом. Это объясняется тем, что в случае обновления обновленные записи добавляются в «конец таблицы», поэтому единовременное занимаемое пространство равно объему до обновления плюс объем новых значений обновленных записей.

Но тем не менее, объем файла остается неизменным на протяжении всех тестов.

Потребление оперативной памяти

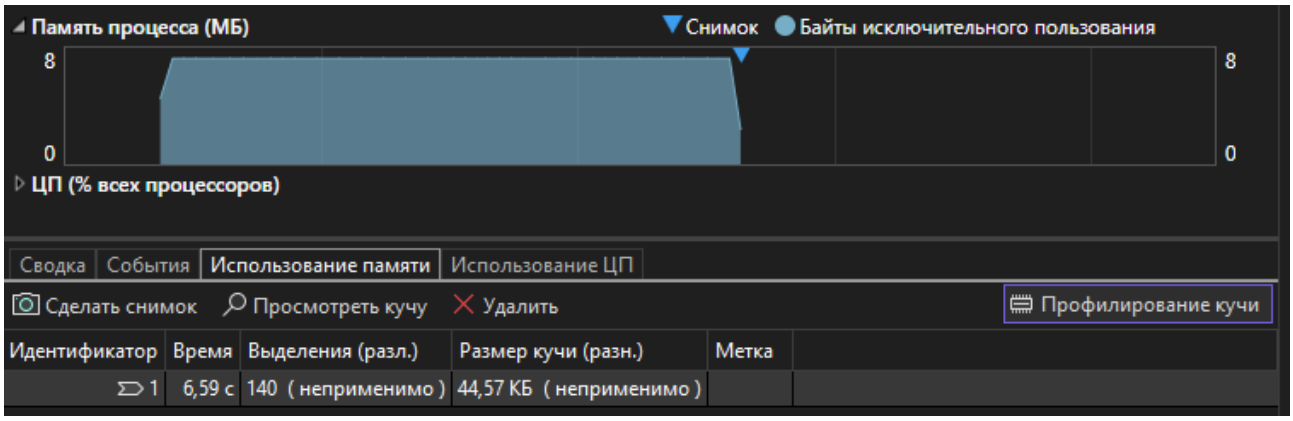
Единственным найденным способом построить график и измерить потребление, а также проверить программу на наличие утечек оказался профилировщик Visual Studio.

Для этого были запущены те же тестовые программы, но уже и использованием профилировщика. Будет 2 теста для каждой операции, в первом тесте 25000 записей, во втором 50000. За счет этого можно будет отследить, как влияет число записей на потребление оперативной памяти. В идеале провести больше, чем 2 теста, чтобы определить динамику, но это не представляется возможным в виду ручного тестирования при помощи профилировщика.

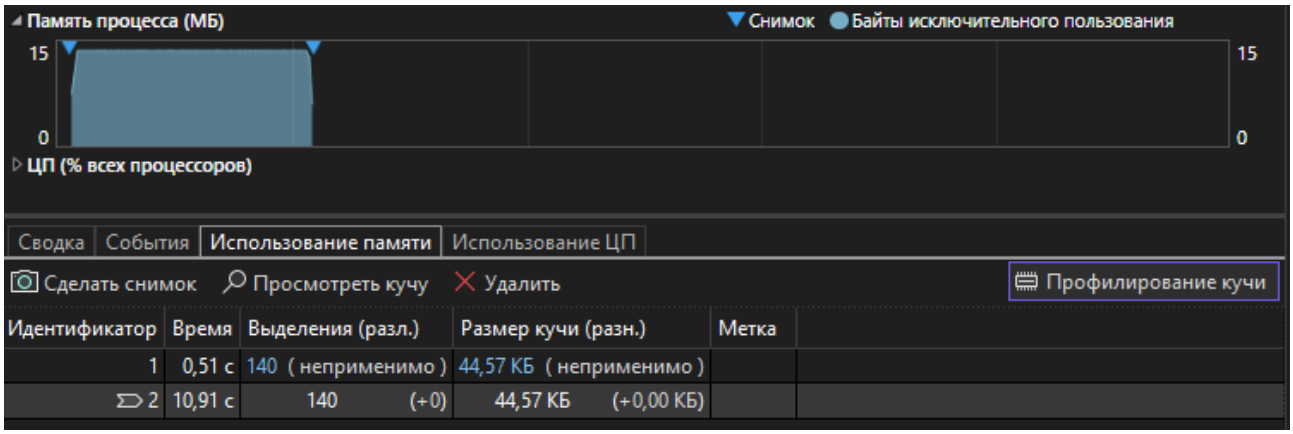
Вставка

В рамках тестирования производилась единоразовая пакетная вставка 25000 и 50000 записей.

25000



50000

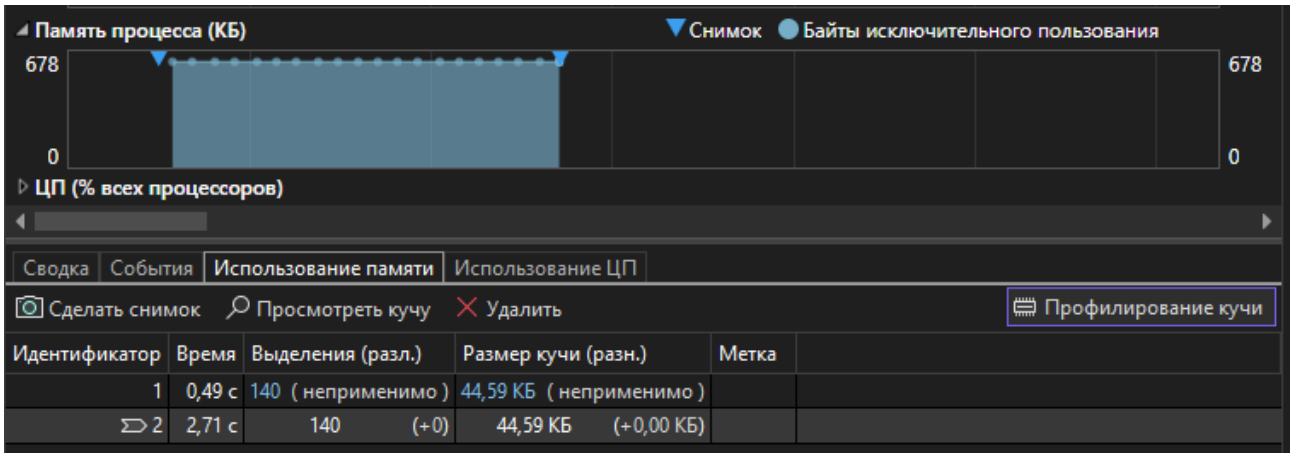


Были произведены снимки памяти перед выполнением программы и после ее завершения. Как можно заметить, число выделений не поменялось, что свидетельствует об отсутствии утечек. Стоит обратить внимание на объем потребляемой памяти – она отличается в тестах.

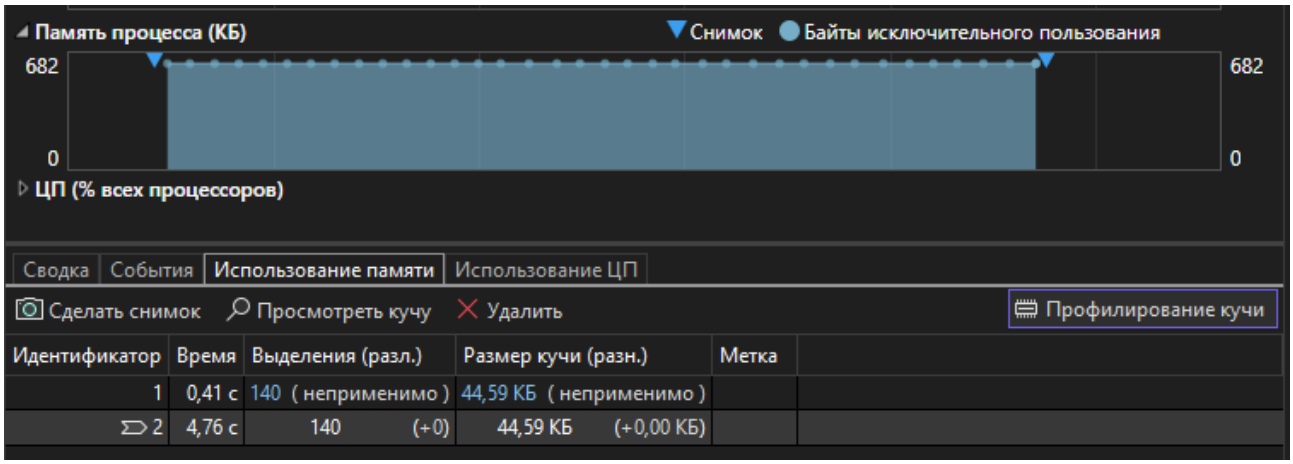
Рост в начале, спад в конце, а также достаточно большой объем занимаемой памяти объясняется спецификой пакетной вставки: в начале составляется пакет со всеми его записями, поэтому происходит рост и занимается 7.5 Мб и 15Мб памяти. В момент, когда пакет сформирован и начинается работа библиотеки – объем памяти почти не меняется, значит, библиотека потребляет небольшое константное количество памяти, которое не зависит от объема вставляемых данных. А также спад в конце объясняется освобождением памяти, выделенной под пакет.

Выборка

При тестировании будем извлекать записи, добавленные в предыдущем тесте.



50000

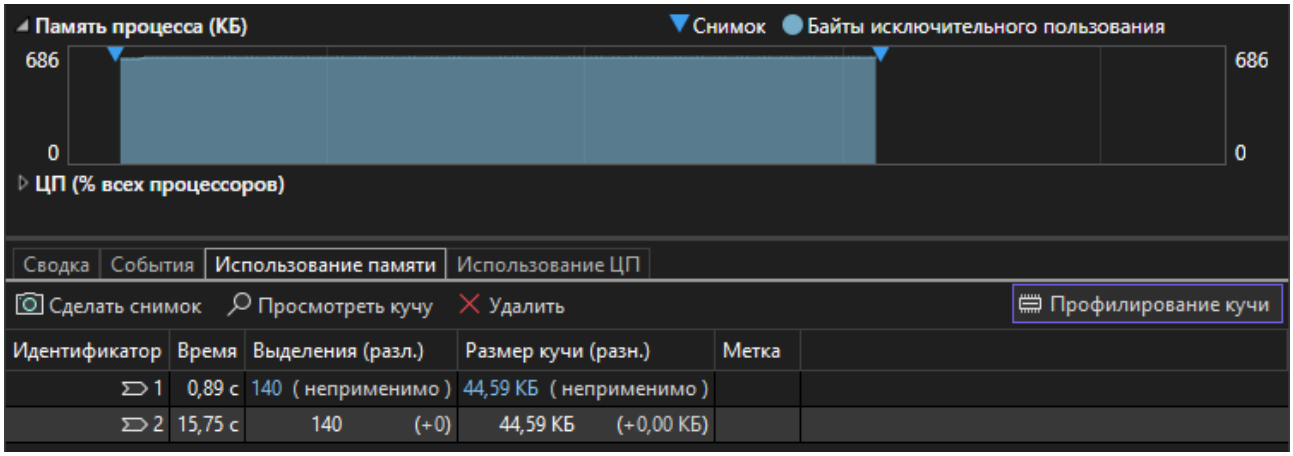


Как можно заметить, в обоих тестах отсутствуют утечки. Потребление памяти во втором тесте примерно равно потреблению в первом тесте.

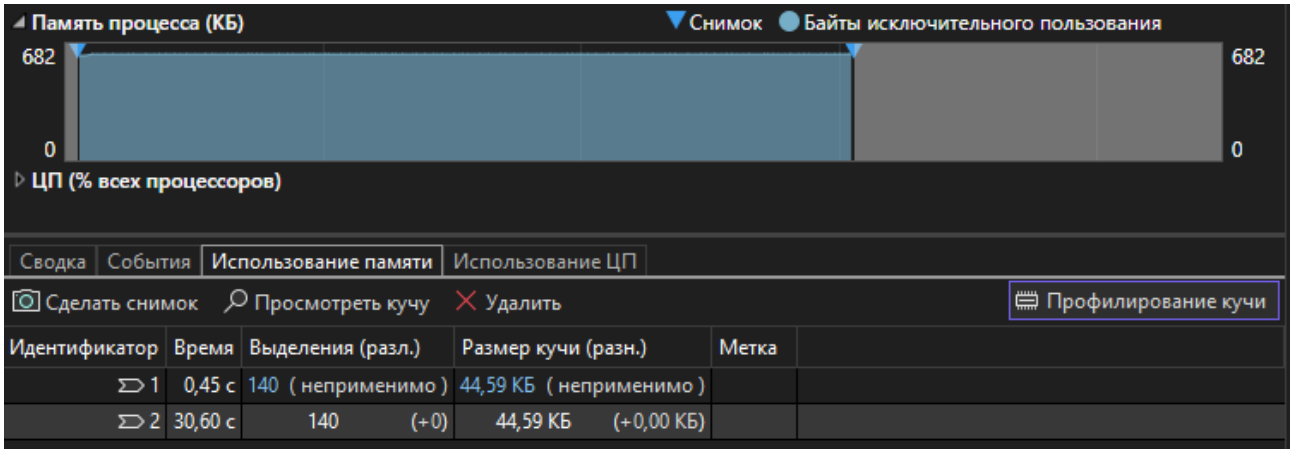
Обновление

Обновим записи, добавленные в предыдущих тестах.

25000



50000

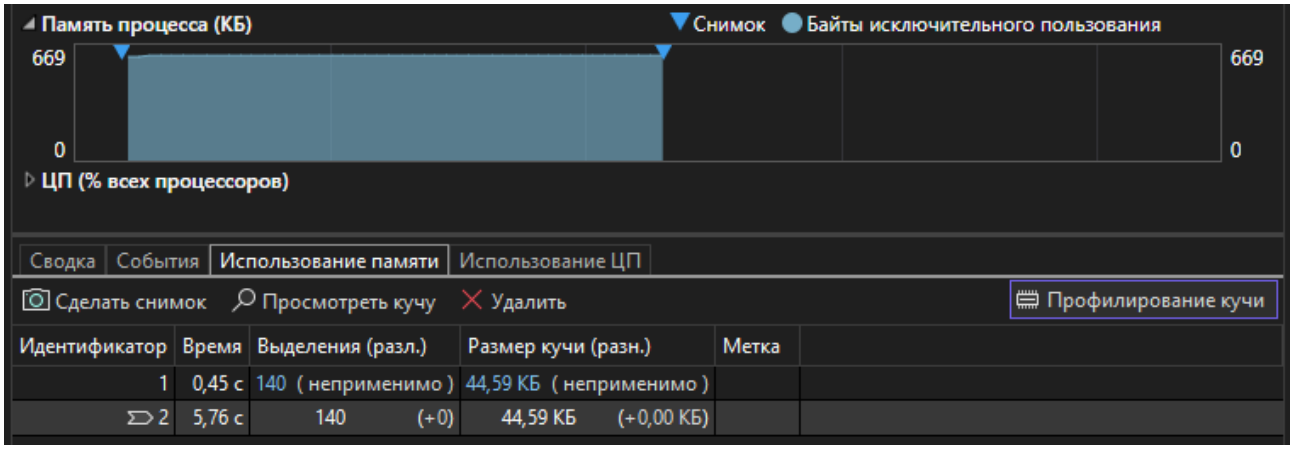


Утечки в обоих тестах отсутствуют, потребление памяти примерно одинаковое.

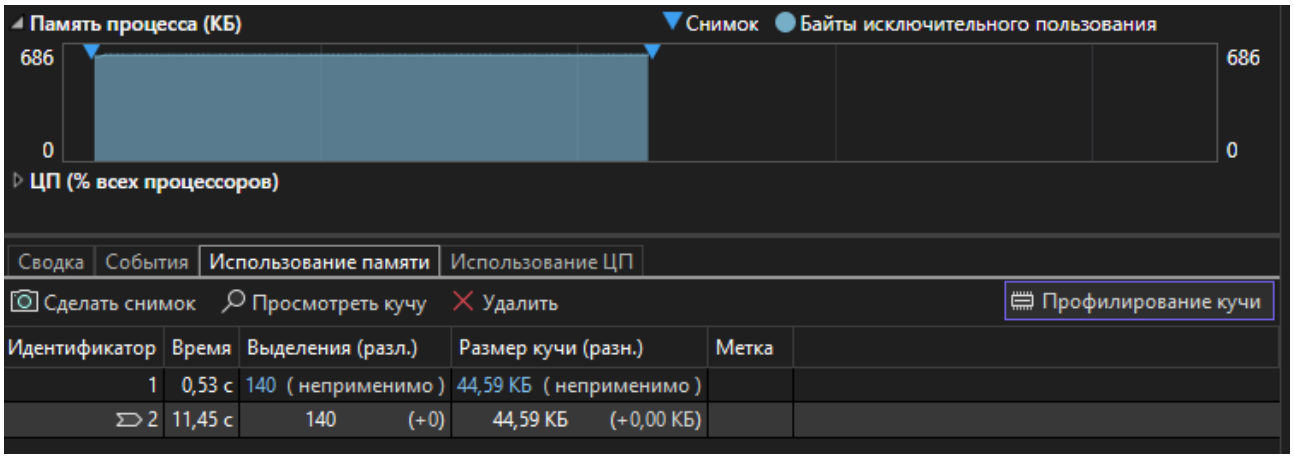
Удаление

Теперь удалим все записи, добавленные и обновленные в предыдущих тестах.

25000



50000



Утечки отсутствуют в обоих тестах, потребление памяти отличается в рамках погрешности.

Вывод

В результате выполнения лабораторной работы был создан модуль, реализующий хранение, доступ и модификацию данных внутри файла.

В ходе тестирования было установлено, что модуль соответствует заявленным требованиям:

- отсутствие утечек памяти, побайтового ввода-вывода;

- необходимая асимптотика операций над файлом;

- пропорциональность размера файла количеству фактически размещенных данных;

- работоспособность на ОС семейств Windows и *NIX.