Большое домашнее задание 1

1 ноября 2024 г.

1 Общее описание задачи

В первом большом ДЗ от вас требуется реализовать собственную in-memory реляционную базу данных.

Чтобы не беспокоиться о сетевых взаимодействиях, правах доступа, пользовательском интерфейсе и прочих подобных вещах, оформить ее предлагается в виде библиотеки, которую можно подключить к произвольному проекту и использовать либо напрямую из пользовательского кода, либо через сторонние интерфейсы (по сети, из консоли или как-нибудь еще).

Что представляет из себя база данных? На самом деле, просто набор таблиц, состоящих из именованных колонок (как правило, их число фиксировано, либо меняется очень редко) и произвольного количества строк (вот они могут добавляться и удаляться часто). Можно думать об этом, как о множестве неких структур.

Естественно, просто хранить таблицы недостатчоно, требуется так же выполнять запросы к ним. Запросы как правило заключаются в добавлении/удалении строк, измененииии существующих и выборке по произвольным условиям. Естественно, критична производительность всего этого.

Требуются следующие возможности:

1. Создание объекта базы. Независимых объектов может быть много и с точки зрения пользователя они ничего не знают друг о друге.

2. Выполнение запросов (иначе какая это база данных). Поскольку предполагается, что запросы могут поступать динамически, требудется единый интерфейс, принимающий их в текстовом виде. Условный `db.execute(user_query)`. О языке запросов – в секции 3.

3. Получение результатов запросов. Если точнее, хочется иметь возможность проитерироваться по строкам результата и получить из них значения конкретных колонок по их именам.

4. Сохранение состоянии базы в файл и последующая его загрузка.

5. Естественно, критичны скорость работы и потребление памяти.

Ожидаемый интерфейс выглядит примерно так (но вы можете придумать лучше):

```
memdb::Database db;
db.load_from_file(std::ifstream("db.bin", ios::bin));
auto result = db.execute(query);
if (result.is_ok()) {
    for (auto &row : result) {
        int i = resp.get<int>("column1");
        std::string_view s = resp.get<std::string_view>("column2");
        do_something(i, s);
    }
} else {
    std::cerr << "Error: " << result.get_error() << "\n";
}
...
db.save_to_file(std::ofstream("db.bin", ios::bin));
```

2 Способ сдачи и требования к решению

Проверка проходит вручную. На проверку вы сдаете код с описанием его работы и инструкцией по компиляции и использованию. Конкретный способ сдачи (ссылка на репозиторий/zip архив/голосовое сообщение в телеграмме с надиктованным кодом/...) уточняйте у семинаристов.

Запрещено использование библиотек, которые уже реализуют требуемую функциональность или ее часть (например LLVM или sqlite). Служебные библиотеки (например ranges) или библиотеки для тестирования (boost::ut, gtest) – рарешены.

К реализации должны быть написаны тесты.

Проверяющий в праве задать любые уточняющие вопросы.

Не забывайте про хорошую декомпозицию кода.

3 Язык запросов

Можно было бы реализовать полноценный SQL, но мы возьмем собственный язык, похожий на него. Это во-первых, позволит проигнорировать несущественные вещи из SQL, а во-вторых даст возможность для расширений языка.

Общие правила:

- Ключевые слова (select, create и т.п.) – нечувствительны к регистру.
- Названия пользовательских сущностей (таблиц и колонок) – чувствительны.
- Трнбования к названиям – такие же, как в C++. Для простоты считаем, что все символы – ASCI.
- Пробельные символы (включая перевод строки) служат для разделения ключевых слов или параметров. Разделитель может содержать произвольное их количество (но, конечно, больше нуля).

3.1 Типы данных

Для простоты введем всего 4 типа данных:

1. `int32` – знаковое 32-битное целое число. В запросах записывается, как последовательность цифр, возможно начинающаяся со знака + или −.
2. `bool` – логическое значение (true или false). В запросах записывается одноименными ключевыми словами.
3. `string[X]` – строка длины не больше X. В запросах записывается внутри двойных ковычек (`"`). Может содержать экранированные символы, аналогично строковым литералам в C++ (например `"\x00\n\t"`).
4. `bytes[X]` – байтовая последовательность длины ровно X. В запросах описывается либо аналогично строке, либо в шестнадцатиричном виде: `0x000abeef`.

3.2 Выражения

Чтобы делать любые хоть немного сложные запросы надо уметь выполнять операции с данными. Мы ограничимся следующими:

- Арифметика – `+`, `-`, `*`, `/`, `%` – выполняется над числами, семантика аналогична C++.
- Сравнения – `<`, `=`, `>`, `<=`, `>=`, `!=` – выполняется над двумя значениями одинакового типа, семантика аналогична C++, кроме операции `=`, которая аналогична `==`. Числа сравниваются очевидным образом, строки и байтовые последовательности – лексикографически, логические значения – по правилу `true > false`.
- Логические операции – `&&`, `||`, `!`, `^^` – применяются к логическим значениям, аналогичны C++, кроме `^^`, который аналогичен `^`.

- `|s|` – применяется к строкам и байтовым последовательностям и означет длину строки/последовательности.
- `+` для строк – конкатенация
- Скобки – задают приоритет операций

Приоритеты операций аналогичны таковым в C++. Неявных преобразований типов нет, т.е. `42 || 0` или `true + false` – считаются невалидными выражениями.

3.3 Create table

Основная сущность базы данных – это таблица. Но чтобы с ней работать, надо ее сначала создать: `create table <имя таблицы> (описание колонок)`

Описание колонок – это последовательность разделенных запятой описаний следующего вида: `[{attributes}] <name>: <type> [= <value>]`, где

- `name` – имя колонки
- `type` – тип значений в колонке
- `value` – значение по-умалчанию для содержимого колонки, если оно есть
- `attributes` – последовательность аттрибутов колонки, перечисленных через запятую. Аттрибуты служат для более точного описания свойств колонки. Нужно реализовать хотя бы 3 аттрибута:
  - `unique` – означает, что значения в колонке должны быть уникальными
  - `autoincrement` – означет, что при добавлении новых строк значение в этой колонке в первый раз будет 0 и будет становиться на 1 больше в каждой новой строке. Применимо только к целочисленным колонкам.
  - `key` – то же, что `unique`, но автоматически построит индекс (о них ниже) по каждой колонке с этим аттрибутом.

Пример запроса: `create table users ({key, autoincrement} id : int32, {unique} login: string[32], password_hash: bytes[8], is_admin: bool = false)`

3.4 Insert

Служит для добавления новых строк в таблицу.

Синтаксис: `insert (<values>) to <table>`, где

- `table` – имя таблицы, куда надо вставить значения.
- `values` – одно из двух. Либо просто список значений через запятую, либо посдовательность записей вида `<name> = <value>`, разделенных затямыми, где `name` – имя колонки в таблице, а `value` – значение. В случае использования второго вида порядок колонок не важен. Колонки, которые имеют значение по умлочанию или аттрибут `autoincrement` – можно опустить. В первой форме записи для этого просто оставляют пустое место между запятыми (см. примеры).

Примеры:

- `insert (,"vasya", 0xdeadbeefdeadbeef) to users`
- `insert (login = "vasya", password_hash = 0xdeadbeefdeadbeef) to users` – эквивалентно предыдущему
- `insert (,"admin", 0x0000000000000000, true) to users`
- ```
  insert (
      is_admin = true,
      login = "admin",
      password_hash = 0x0000000000000000
  ) to users
  ```
  – эквивалентно предыдущему

3.5 Select

Запрос выборки данных. Берет таблицу, выбирает из нее данные, соответствующие условию и возвращает новую таблицу.

Синтаксис: `select <columns> from <table> where <condition>`, где

- `table` – либо имя таблицы, из которой надо получить значения, либо выражение `join` (о них ниже)
- `columns` – имена колонок через запятую в формате `<table>.<column>` (например `users.id`). Если в запросе используется только одна таблица то ее имя можно опустить и писать только имя колони.
- `condition` – логическое выражение над значениями колонок, по которому надо отфильтровать результаты.

Пример: `select id, login from users where is_admin || id < 10`
  ```

3.6 Update

Запрос обновления данных.

Синтаксис: `update <table> set <assignments> where <condition>`, где

- `table` – аналогично `select`
- `condition` – аналогично `select`
- `assignments` – набор имен колонок и их значений, синтаксис аналогичен второй форме `insert` за исключением того, что значения могут быть не константами, а выражениями от старых значений данной строки.

Примеры:

- `update users set is_admin = true where login = "vasya"`
- ```
  update users set login = login + "_deleted", is_admin = false where
      password_hash < 0x00000000ffffffff
  ```

3.7 Delete

Запрос удаления данных.

Синтаксис: `delete <table> where <condition>`, где

- `table` – имя таблицы, из которой надо удалить строки
- `condition` – аналогично `select`

Пример:

```
delete users where |login| % 2 = 0
```

3.8 Join

Естественно, делать запросы к отдельным таблицам – не особо полезно. Чтобы собирать данные из нескольких таблиц служит `join`.

Это не отдельная команда, а конструкция, которая может использоваться в запросах `select` и `update` вместо имени таблицы.

`join` делает очень простую вещь – он выбирает из первой и второй таблицы пары строк, подходящие под условие и составляет новую таблицу, имеющую колонки из обеих предыдущих и содержащую в соответствующих колонках данные из них.

Синтаксис: `<table1> join <table2> on <condition>`, где

- `table1` и `table2` – имена таблиц для объединения.
```

- `condition` – выражение, определяющее, какие пары записей брать.

Пример: пусть есть 2 таблицы users и posts:

id	login	is_admin
1	"vasya"	false
2	"petya"	false
3	"admin"	true

и

id	user_id	text
1	1	"A"
2	1	"B"
3	3	"C"

Тогда `select posts.id, users.login, posts.text from users join posts on users.id = posts.user_id where true`

Должен вернуть

id	login	text
1	"vasya"	"A"
2	"vasya"	"B"
3	"admin"	"C"

3.9 Create index

Запрос создания индекса. Индексы никак не влияют на результат запросов, но могут менять время исполнения. Индекс создается по несколким колонкам и позволяет базе в дальнейшем быстрее выполнять сравнения этих колонок. В нашей базе будет 2 вида индексов:

1. `ordered` – если такой был создан по колонке x то он позволяет в быстро делать выборку, сравнивая значения этой колонки с чем-то. Например `select ... where x > 10`. Естественно, ожидается, что это повлияет и на более сложные запросы с дополнительными условиями, например `select ... where x > 100 && x < 200 && y > x`.

2. `unordered` – создается по набору колонок и позволяет быстро отбирать строки, где весь этот набор равен чему-нибудь. Например, если создать такой индекс по колонкам x и y то ожидается, что запросы типа `select ... where x = 10 && y = 20` будут выполняться быстро. Аналогично предыдущему, условия могут быть и более сложными.

Мы будем считать, что индексы работают достаточно хорошо, если:

- для `ordered` – при наличии индекса по колонке `x` будет наблюдаться заметное ускорение при запросах с условиями вида `(...) && (x < 1000) && (x > 500)` (Естественно, порядок выражений может быть произвольным, вместо `<` и `>` могут идти другие сравнения, а вместо 1000 и 500 – другие значения, не обязательно числовые).
- для `unoordered` – при наличии индекса по колонкам `x, y, z` будет наблюдаться заметное ускорение при запросах с условиями вида `(...) && (x = 1) && (y = 2) && (z = 3) && (...)`. (Порядок выражений произвольный, значения могут быть другими).

Если в запросе можно применить несколько индексов то достатоно, чтобы был применен хотя бы один.

Hint: Возможно, тут пригодится STL...

Синтаксис: `create <index_type> index on <table> by <columns>`, где

- `index_type` – тип индекса
- `table` – имя таблицы
- `columns` – список колонок через запятую.

Примеры:

```
create ordered index on users by login
create unordered index on users by is_admin
```

4 Оценка

- 1 балл – за реализацию интерфейса базы (создание, метод для выполнения запросов, обработка ошибок, возможность итерации по результату запроса).
- 1 балл – `create table` + `insert`
- 1 балл – `select` без `join`
- 1 балл – `update` без `join`
- 1 балл – `delete`
- 1 балл – работающие индексы.
- 2 балла – `join`
- до 2 баллов – любые улучшения сверх описанного. Подойдут подзапросы, nullable колонки и 3 вида outer join, constexpr-запросы, ORM, structured binding на результат выборки, новые хитрые виды индексов и аттрибутов, особо хорошая оптимизация и все, что может впечетлить проверяющего

- За любой пункт оценка может быть снижена на усмотрение проверяющего за ошибки, недочеты или неполноту в реализации
- За любой пункт ошибка снижается в 2 раза за отсутствие тестов (и пропорционально за неполное тестирование, например протестированная наполовину функциональность принесет 0.75 от своих баллов).