Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
Высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана

Курсовая работа по курсу «Базы данных»

Разработка пространственно-временной базы данных с использованием средств параллельного программирования CUDA

Студент группы ИУ9-61
_____ А. Ю. Борнев
«___» _____ 2016 г.

Руководитель курсовой работы
_____ И. Э. Вишняков
«___» _____ 2016 г.

Москва
2016

Оглавление

Введение

В курсовой работе ставится задача реализовать свою пространственно-временную базу данных, используя средства параллельного программирования CUDA. Для ускорения запросов к базе данных был выбран параллельный алгоритм построения пространственных индексов HLBVH2 из статьи [1]. Изначально при построении алгоритма HLBVH2 авторы предполагали его использование для построения реалистичных изображений методом трассировки лучей. В данной работе на его основе разработаны алгоритмически эффективные операции для вставки в базу данных, поиска k ближайших соседей для данной точки, поиска точек, удалённых от данной ломаной на заданное расстояние, и поиска точек, лежащих в данном полигоне. Также разработан прототип базы данных, предоставляющий данные возможности.

1 Пространственные и временные базы данных

В различных областях необходимо поддерживать геометрические, географические или пространственные данные. В таких областях применяются пространственные базы данных. Они предоставляют дополнительные возможности для обработки пространственных объектов по сравнению с обычными базами данных. Такие БД имеют дополнительные пространственные типы, такие как точка, ломаная, полигон. Они представляют фундаментальные абстракции для моделирования геометрических структур в пространстве. Точка представляет собой объект, для которого важна только позиция в пространстве, но не его форма. Например, город может быть представлен точкой на карте большого масштаба. Ломаная – это базовая абстракция, которая позволяет представить объект «соединённый в пространстве» (дороги, реки, телефонные кабели, и т.д.). Полигон может иметь отверстия и представлять собой страны, моря и другие объекты. Также в таких базах данных имеются специальные операции над пространственными типами [2].

Одним из важных аспектов пространственных баз является построение индексов для пространственных данных. Индексы позволяют повысить производительность выполнения запросов к базе данных. Основная идея состоит в сокращении количества обробатываемых объектов при выполнении запроса. В пространственных базах данных для этого используются алгоритмы, основанные на иерархии ограничивающих оболочек или R-дереве, то есть дереве, где узлами являются ограничивающие оболочки исходных объектов, и каждый родительский узел является объемлющей оболочкой дочерних узлов [3]. Другой тип алгоритмов предполагает перевод пространственных координат в одномерные с последующим расположением в B-дереве [2]. Для перевода координат используются специальные коды Мортона или их аналоги. Коды Мортона позволяют добиться локальности данных в пространстве, так как кодам с одинаковым префиксом соответствуют объекты, расположенные рядом друг с другом в пространстве. Обратное утверждение верно не всегда.

Временной базой данных называется база, которая использует временную модель данных [4]. Моделью данных называется набор структур и средства для их обработки и обновления. Вот уже много лет нет единого мнения, какие пространства времени нужно использовать: время транзакции *(transaction time)*,

действительное время *(valid time)*, время принятия решений *(decision time)* и т.д. Также неясно, какие отметки времени необходимо использовать: точки *(points)*, отрезки (периоды), множество точек, и т.д. Можно сделать вывод, что время – это мультипространственная величина, поэтому во временных базах данных необходимо поддерживать несколько временных пространств. Наиболее важными являются время транзакции и действительное время. Временем транзакции называется временная отметка, когда запись была внесена в базу данных. Действительное время – это время, когда событие произойдёт (возможно в будущем, настоящем или прошлом). Временная модель данных может поддерживать ни одного, одно, два или больше временных пространств. Можно выделить следующие временные модели:

1. Модель типа снимок (snapshot data model) – представляет одиночный снимок базы данных.
2. Модель с действительным временем – поддерживает только действительное время.
3. Модель с временем транзакции – поддерживает только время транзакции.
4. Битемпоральная модель данных – поддерживает и действительное время, и время транзакции.

В данной работе было решено разработать базу данных, сочетающую некоторые возможности временной базы данных и пространственной базы данных. Такая база данных с некоторыми ограничениями может называться пространственно-временной базой данных. Обычно от таких баз требуют рномещё поддержку непрерывного обновления данных. Эта возможность является крайне сложной в реализации и реализована не будет.

2 Разработка алгоритмов для пространственно-временной базы данных

Разрабатываемая база данных позволяет создавать таблицы с элементами вида ключ-значение, в которых ключ состоит из пространственной и временной части. Пространственная часть – это либо полигон, либо точка, либо ломаная в двумерном пространстве. Временной частью является либо интервал действительного времени *(valid time)*, либо время транзакции *(transaction time)*, либо и то и другое *(bitemporal time)*. Столбцы таблицы могут иметь следующие типы: строка, целое число, действительное число, временная метка. Также в базе данных обеспечивается уникальность ключа. Разрабатываемая база данных должна поддерживать следующие операции:

1. Поиск k ближайших соседей заданной точки среди множества точек.
2. Поиск точек, принадлежащих заданному полигону.
3. Поиск точек, удалённых на расстояние не более R от заданной прямой.
4. Вставке множества строк в таблицу.
5. Удаление, выборка и изменение строк в таблице по предикату.

Основной подзадачей данной курсовой работы является изучение структуры данных HLBVH2 [1], и способов её применения в пространственно-временных базах данных. В разрабатываемой базе данных эта структура используется для ускорения пространственных запросов. Она представляет собой иерархию ограничивающих оболочек, организованную в двоичное бинарное дерево, которая строится за линейное число операций. В базе данных структура данных HLBVH2 используется в следующих алгоритмах:

1. Поиск k ближайших соседей заданной точки среди множества точек.
2. В качестве фильтра при поиске точек, принадлежащих заданному полигону.
3. В качестве фильтра при поиске точек, удалённых на расстояние не более R от заданной прямой.
4. При вставке множества строк в таблицу.

Перейдём к описанию алгоритма построения HLBVH2. На вход алгоритму поступает массив четырёхмерных ограничивающих оболочек $arrayAABB$ и размер листового элемента $leafSize$. В данном алгоритме оболочки выравнены по осям координат и задаются двумя точками min и max. Первым шагом

для каждой из оболочек высчитывается центр и кодируется с помощью кода Мортона. Результат записывается в массив $codes$ в виде пары $(key, value)$, где key – код Мортона, $value$ – позиция в массиве $codes$. Затем выполняется числовая сортировка массива $codes$ по первому элементу пары. Следующим шагом необходимо построить структуру будущего дерева. Для этого заводится две очереди $queue[2]$, элементами которых являются тройки целых чисел, три массива $nodeoffset$, $links$, $ranges$, которые после выполнения будут кодировать структуру дерева. Первая компонента элемента $queue$ представляет номер узла дерева, вторая и третья компоненты кодируют левую и правую границу (не включительно) в отсортированном массиве кодов. Изначально в очередь помещается тройка $(0, 0, b)$, где b – размер массива кодов. Каждый шаг алгоритма строит структуру дерева уровень за уровнем. Для отслеживания количества уровней используется счётчик $treeLVL$ и массив $nodeoffset$, который позволяет определить число узлов до i-ого уровня включительно. Пошаговая схема алгоритма построения структуры дерева заключается в следующем:

1. $nodeoffset[0] = 0$ – количество узлов до i-ого уровня включительно, $treeLVL = 0$ – текущий уровень дерева.
2. $nodeIter = 0$ – номер узла, $ch = 0$ – выбранная очередь.
    2.1. Пока очередь не пуста, извлекается элемент (n, l, r) из $queue[ch]$.
    2.2. Если $r - l > leafSize$, то узел n не листовой. Вычисляется $N = codes[l].keyxorcodes[r].key$.
        2.2.1. Если N нуль, то это означает, что коды на участке $[l, r)$ полностью совпадают. Поэтому разбиение производится по середине массива: целочисленная переменная m устанавливается равной $(r + l)/2$.
        2.2.2. Если N не нуль, то необходимо найти секущую плоскость. Для этого выполняется поиск позиции p первого единичного бита, начиная со старших разрядов. Затем нужно найти такое число $l < m < r$, что код $codes[m]$ в позиции p имеет единичный бит, а $codes[m - 1]$ имеет нулевой бит. Это выполняется с помощью бинарного поиска по массиву кодов.

2.3. $range[n] = (l, r)$ – означает, что узел n «охватывает» элементы с l-ого до $(r-1)$-ого массива $codes$.

2.4. Если узел n листовой, то в $links[n]$ записывается константа $LEAF$.

2.5. Если узел n не листовой, то в $links[n]$ записывается $nodeIter$. В очередь $queue[1 - ch]$ помещается элементы $(NodeIter, left, m)$, $(NodeIter + 1, m, right)$. $NodeIter = NodeIter + 2$.

3. $ch = 1 - ch$, $nodeoffet[treeLVL] = nodeIter$, $treeLVL = treeLVL + 1$.
4. Если очередь $queue[ch]$ пустая, то алгоритм заканчивает работу. Иначе выполняется переход на шаг 2.1.

После того как построена структура дерева, выполняется построение иерархии ограничивающих оболочек снизу вверх, начиная с предпоследнего уровня.

Алгоритм:

1. $i = treeLVL - 2$ – предпоследний уровень дерева.
2. Пока $i >= 0$:
   2.1. $left = nodeoffset[i]$, $right = nodeoffset[i + 1]$.
   2.2. Для каждого узла j, где j изменяется от $left$ до $right - 1$:
      2.2.1. Если узел j листовой, то $(l, r) = range[j]$. Считается объединение оболочек $AABB[codes[k].value]$, где k изменяется от l до $r - 1$.
      2.2.2. Если узел j не листовой, то считается объединение оболочек двух его детей: $AABB[links[j]]$, $AABB[links[j + 1]]$.
      2.2.3. В обоих случаях объединение записывается в $treeAABB[j]$.
   2.3. $i = i - 1$.

Теперь перейдём к рассмотрению алгоритма поиска k ближайших соседей точки среди множества заданных точек $Bset$, $|Bset| \geq k$. В этом алгоритме используются определения $mindist$, $minmaxdist$ и основные идеи из статьи [5]. Приведём необходимые сведения здесь:

***Определение* 1.** *Пусть $p = (p_1, \ldots, p_n)$ – точка в n-мерном пространстве, R – некоторая оболочка, заданная двумя точками $S = (s_1, \ldots, s_n)$, $T = (t_1, \ldots, t_n)$*

и $\forall i : s_i \leq t_i$. *Тогда определим две величины:*

$$mindist(p, R) = \sum_{1}^{n} |p_i - r_i|, \text{ где } r_i = \begin{cases} s_i, \text{ если } p_i < s_i \\ t_i, \text{ если } p_i > t_i \\ p_i, \text{ иначе} \end{cases}$$

$$minmaxdist(p, R) = \min_{1 \leq k \leq n} \left(|p_k - rm_k| + \sum_{\substack{i \neq k \\ 1 \leq i \leq k}} |p_i - rM_i| \right),$$

$$\text{где } rm_k = \begin{cases} s_k, \text{ если } p_k \leq \dfrac{s_i + t_i}{2} \\ t_i, \text{ иначе} \end{cases} \quad rM_i = \begin{cases} s_k, \text{ если } p_k \geq \dfrac{s_i + t_i}{2} \\ t_i, \text{ иначе} \end{cases}$$

Также заметим, что для $mindist$, $minmaxdist$ из определения 1 выполняется следующая лемма:

***Лемма* 1.** *Для любой оболочки* t *и точки* p *существует такая точка* $e \in t$, *что* $mindist(p, t) \leq dist(e,p) \leq minmaxdist(p,t)$, *где* $dist(e,p)$ – *евклидово расстояние.*

Тогда можно описать алгоритм поиска *k* ближайших соседей для данной точки следующим образом:

1. По множеству $Bset$ строится HLBVH2.
2. Выделяется очередь с приоритетами pr размера k. Элемент очереди представляет собой пару $(key,value)$, где key – расстояние от точки p до точки с номером $value$ в исходном множестве $Bset$. Создаётся пустой стек st, на вершину стека кладётся вершина дерева HLBVH2.
3. Пока стек не пуст:
    - 3.1. Извлекается верхний элемент стека: $pos = st.pop()$.
    - 3.2. Если $mindist$ от оболочки $treeAABB[pos]$ до точки p больше вершины очереди $pr.top()$, то дальнейшие шаги пропускаются.
    - 3.3. Если узел pos есть в pr, то удалить его и внести пару с ключом бесконечность в pr.
    - 3.4. Если узел pos листовой, то для каждой точки q, которая содержится в узле (переход от узла к точкам выполняется с помощью массивов $ranges$ и $codes$), если $dist(q, p) > pr.top().key$, то вы-

полняется извлечение максимума из очереди pr и вставка пары $(dist(p,q), id_q)$ в очередь, где id_q – индекс точки q в HLBVH2.

3.5. Если узел pos не листовой, то необходимо выбрать, какое поддерево посещать первым. Ясно, что лучше посещать ближайшее поддерево. Поэтому в стек st сначала кладётся индекс удалённого потомка, а затем ближайшего. Расстояние считается на основе $mindist$. Также считается $minmaxdist$ до каждого из потомков. И если для потомка выполняется $minmaxdist < pr.top().key$, то он содержит точку, которая является кандидатом на ближайшую к p (лемма 1). Поэтому выполняется извлечение максимума из очереди pr и вставка $(minmaxdist, h)$, где h – потомок pos.

В разрабатываемой базе данных обеспечивается уникальность вставляемых ключей, поэтому приходится использовать сложный алгоритм для вставки множества строк $rows$ в таблицу. При том, что наивный алгоритм требует $O(N^2)$ операций, рассматриваемый алгоритм требует примерно $O(N \log N)$ операций в худшем случае (N – сумма числа строк в таблице и числа вставляемых строк). Рассмотрим данный алгоритм. Для каждой вставляемой строки считается её ограничивающая оболочка. По этой оболочке в HLBVH2 (построенной по оболочкам строк таблицы) выполняется поиск всех оболочек, которые полностью совпадают с ней. На основе этого формируется множество строк set из таблицы, которые могут иметь одинаковые ключи со вставляемыми строками. После этого производится сортировка слиянием по ключу множества $set \cup rows$. Затем за один проход выполняется проверка, что i-й элемент имеет ключ, отличный от $(i+1)$-го. В этом случае вставляемые ключи являются уникальными. Итоговая сложность $O(n \log n)$, где n – размер $set \cup rows$.

Рассмотрим алгоритм поиска всех точек, принадлежащих полигону, с использованием уже построенной HLBVH2 по исходной таблице. Первым шагом выполнятся поиск и формирование множества точек-кандидатов, которые лежат в ограничивающей оболочке. Затем для каждой точки из множества кандидатов считается её *порядок относительно кривой*[1], задающей границу полигона [6]. Если этот порядок равен нулю, то точка не принадлежит полигону, иначе принадлежит. Аналогично строится алгоритм поиска точек, удалённых от за-

[1]Порядок точки относительно замкнутой кривой на плоскости – это целое число, представляющее число полных оборотов, которое делает кривая вокруг заданной точки против часовой стрелки.

данной кривой не больше чем на r. Также как и в случае с полигоном, сначала формируется множество кандидатов, которые затем тестируются с использованием кривой. Для каждой точки кривой p и точки из множества кандидатов выполняется проверка на принадлежность окружности с центром в точке p и радиусом r. Также каждое ребро e заключается в прямоугольник со сторонами, равными длине ребра e и удвоенному радиусу r. После этого точки из множества кандидатов проверяются на принадлежность этому прямоугольнику.

3 Реализация базы данных и её API

Реализация базы данных была выполнена на языке C++ с использованием средств параллельного программирования CUDA. Использовались следующие сторонние библиотеки:

1. `moderngpu` – для алгоритма сортировки слиянием на GPU [7].
2. `cub` – для числовой сортировки и префиксного сканирования на GPU [8].
3. `thrust` – аналог `std::vector` на GPU [9].
4. `glew/glfw` – для визуализации [10, 11].

Для обработки ошибок в базе данных используется специальный класс `Result<T,E>`, аналогичный используемому в языке Rust. Данный тип позволяет вернуть объект типа `T`, если функция завершается без ошибок, и объект типа `E`, если функция завершается с ошибкой. При реализации класса `Result<T,E>` использовались новейшие средства языка C++, такие как move-семантика. Объекты переносятся внутрь класса `Result` и также переносятся при их извлечении. Также введён макрос `TRY`, принимающий объект типа `Result<T,E>` и позволяющий выполнить ранний возврат из функции, если переданный объект содержит ошибку. Данный макрос можно использовать только в функциях, которые возвращают объект типа `Result<U,V>`, где `V` совпадает с `E`. Если же переданный объект не содержит ошибку, то `TRY` вернёт его.

Перейдём к описанию типов ключей и столбцов в базе данных. В листинге 1 приводятся возможные типы пространственной и временной частей ключа. Пространственный тип `POLYGON` представляет собой конечную последователь-

Листинг 1 — Типы пространственной и временной частей ключа

```
enum class SpatialType {
    POLYGON,
    LINE,
    POINT,
    UNKNOWN
};

enum class TemporalType {
    TRANSACTION_TIME,
    VALID_TIME,
    BITEMPORAL_TIME,
    UNKNOWN
};
```

ность двумерных точек, задающих полигон, ориентированный против часо-

вой стрелки. Пространственный тип POINT представляет собой точку в двумерном пространстве. Пространственный тип LINE представляет собой конечную последовательность точек. Пространственный тип UNKNOWN означает, что пространственный ключ не задан. Строки с такими ключами в базу данных вставлены быть не могут. Временной тип TRANSACTION_TIME означает, что в ключ добавляется время внесения строки в таблицу. Временной тип VALID_TIME означает, что в ключ добавляется отрезок действительного времени. Временной тип BITEMPORAL_TIME означает, что временная часть состоит из времени транзакции и отрезка действительного времени. Также как и у пространственного типа UNKNOWN означает, что временной ключ не задан. В листинге 2 приводятся типы для столбцов таблицы. Тип данных STRING является ASCIIZ-строкой

Листинг 2 — Типы данных столбцов

```
enum class Type {
    STRING,
    INT,
    REAL,
    DATE_TYPE,
    SET,
    UNKNOWN
};
```

фиксированного размера (256 символов включая нуль-терминатор), тип данных INT является 64-разрядным знаковым числом, тип данных REAL является 64-разрядным числом с плавающей точкой. Тип данных DATA_TYPE представляет собой временную метку, с точностью до микросекунд и способен представлять диапазон от 50000 лет до нашей эры, до 50000 нашей эры включительно, при этом для кодирования временной метки используется единственное 64 битное число. Тип данных SET используется только в таблицах специального типа TempTable (см. раздел 3.2). Он позволяет иметь в таблице множество ссылок на строки другой таблицы. Тип UNKNOWN означает, что тип столбца не задан.

База данных (класс DataBase) реализована с использованием паттерна *singleton*, это означает, что имеется только один представитель данного класса. Для получения данного представителя используется метод `static DataBase &DataBase::getInstance()`. Далее в тексте будут описываться методы класса DataBase без привязки к конкретному объекту.

Базу данных возможно сохранять и загружать с диска. Функция `Result<void, Error<std::string>> saveOnDisk(std::string path)`

класса DataBase принимает полный путь (путь и название базы с расширением) path и выполняет сохранение базы данных на диск. Функция Result<void, Error<std::string>> loadFromDisk(std::string path) класса DataBase принимают путь path до файла, из которого будет произведена загрузка базы данных. Для обеспечения целостности данных в базе при сохранении внутрь файла записывается контрольная сумма SHA512, а при загрузке происходит проверка файла базы на соответствие этой контрольной сумме. Обе функции возвращают Result<void, Error<std::string>>, то есть если функция завершится без ошибок, то Result будет содержать void, иначе объект класса Error<std::string>, который хранит название исходного файла, номер строки, название функции и описание ошибки.

3.1 Создание таблицы и её заполнение

Опишем теперь классы, необходимые для создания таблицы. Каждая таблица в базе данных состоит из двух частей. Одна часть хранится на CPU и содержит описание таблицы TableDescription, другая хранится на GPU и представляет собой описание таблицы и данных. Дублирование описания таблицы обусловлено необходимостью знать типы данных и на GPU (например, при применении предикатов), и на CPU (например, при выводе таблицы, получении данных, вставке).

Класс AttributeDescription представляет собой описание атрибута. Он позволяет задавать имя атрибута (столбца) и его тип. Класс имеет следующие поля:

1. std::string name – задаёт имя столбца.
2. Type type – задаёт тип столбца (см. листинг 2).

Класс TableDescription представляет собой описание таблицы. Он позволяет задавать имя таблицы и вместе с классом AttributeDescription позволяет дать полное описание всех столбцов таблицы. Данный класс реализует следующие методы:

1. Result<void, Error<std::string>> setName(std::string newName) – устанавливает имя таблицы newName.
2. Result<void, Error<std::string>> setSpatialKey(std::string const keyName, SpatialType keyType) – устанавливает имя пространственного ключа newName и его тип keyType.

3. Result<void, Error<std::string>> setTemporalKey(std::string const keyName, TemporalType keyType) – устанавливает имя временного ключа newName и его тип keyType.
4. Result<void, Error<std::string>> addColumn(AttributeDescription col) – добавляет столбец col в таблицу.
5. Result<void, Error<std::string>> delColumn(std::string colName) – удаляет столбец из таблицы по имени (colName).

После того, как задано описание таблицы TableDescription, таблицу можно создать методом класса DataBase: Result<void,Error<std::string>> createTable(TableDescription table). Важно, что созданная таблица не может иметь тип столбца SET. Для удаления таблицы можно воспользоваться методом Result<void, Error<std::string>> dropTable(std::string tableName), где параметром является имя таблицы tableName.

Перейдём теперь к заполнению созданной таблицы. Класс Attribute хранит информацию о значении определённой ячейки в строке. Класс реализует следующие методы:

1. Result<void, Error<std::string>> setName(std::string name) – позволяет задать имя атрибута (должно совпадать с именем столбца).
2. std::string getName() const – позволяет получить имя атрибута.
3. bool isNull() const – позволяет проверить является ли NULL значением атрибута.
4. Type getType() const – позволяет получить тип атрибута.
5. Result<std::string, Error<std::string>> getString() const – если тип атрибута STRING, то возвращается строка, иначе сообщение об ошибке.
6. Result<int64_t, Error<std::string>> getInt() const – если тип атрибута INT, то возвращается int_64t, иначе сообщение об ошибке.
7. Result<double, Error<std::string>> getReal() const – если тип атрибута REAL, то возвращается double, иначе сообщение об ошибке.
8. Result<std::unique_ptr<TempTable>, Error<std::string>> getSet() const – если тип атрибута SET, то возвращает указатель на TempTable, который содержит данное множество строк, иначе сообщение об ошибке.

9. `template<typename T> Result<void, Error<std::string>> setValue(T val = T())` – позволяет устанавливать значение для атрибута и автоматически выводить тип. Для этого для данной функции написана частичная реализация многих примитивных типов языка C++. Возвращает void в случае успеха, иначе `Error<std::string>` с описанием ошибки.
10. `Result<void, Errorstd::string> setNullValue(Type t)` – позволяет установить значение атрибута `NULL` типа `t`.
11. `bool operator<(Attribute const &b) const` – сравнение в лексикографическом порядке.
12. `bool operator==(Attribute const &b) const` – два атрибута равны тогда и только тогда, когда равны их имена, так как имена используются для идентификации столбцов.

Класс `SpatialKey` представляет собой пространственную часть ключа. Он имеет следующие поля:

1. `SpatialType type` – тип пространственного ключа.
2. `std::string name` – имя пространственного ключа.
3. `std::vector<float2> points` – последовательность точек, которая представляет пространственный ключ данного типа.

Класс `SpatialKey` реализует единственный метод `isValid()`, который позволяет проверить: выполняются ли все необходимые ограничения на данный ключ. Опишем данные ограничения:

1. Если типом пространственного ключа является `POINT`, то `points.size()` равен единице и точка является подмножеством декартового произведения $[-180; 180] \times [-90; 90]$.
2. Если типом пространственного ключа является `LINE`, то `points.size()` больше либо равно двум и каждая точка является подмножеством декартового произведения $[-180; 180] \times [-90; 90]$.
3. Если типом пространственного ключа является `POLYGON`, то `points.size()` больше либо равно трёх, каждая точка является подмножеством декартового произведения $[-180; 180] \times [-90; 90]$, полигон, построенный по данным точкам, ориентирован против часовой стрелки, и любые три последовательные точки не лежат на одной прямой.

Класс TemportalKey представляет собой временную часть ключа. Он имеет следующие поля:

1. std::string name – имя пространственного ключа.
2. TemporalType type – тип пространственного ключа.
3. Date validTimeS – начало отрезка действительного времени.
4. Date validTimeE – конец отрезка действительного времени.
5. Date transactionTime – время транзакции (устанавливается автоматически базой данных).

Также как и класс SpatialKey, данный класс реализует единственный метод isValid(). Метод возвращает true, если выполнено следующие ограничение: если ключ имеет тип BITEMPORAL_TIME или VALID_TIME, то validTimeS меньше или равно validTimeE.

Наконец тип Row, который представляет собой строку в базе данных, имеет следующие публичные поля и методы:

1. SpatialKey spatialKey – пространственный ключ.
2. TemportalKey temporalKey – временной ключ.
3. Result<void, Error<std::string>> addAttribute(Attribute const& atr) – данный метод принимает описание атрибута (atr) и добавляет его в строку. Для добавленных атрибутов гарантируется уникальность по имени.
4. Result<void, Error<std::string>> delAttribute(std::string const &atr) – данный метод выполняет удаление атрибута по имени.
5. uint getAttributeSize() – возвращает количество атрибутов в строке.
6. Result<Attribute, Error<std::string>> getAttribute(uint id) – в случае корректной работы возвращает ячейку с индексом id, иначе описание ошибки.
7. void clearAttributes() – выполняет очистку атрибутов в строке.

После того, как сформирован класс Row, вставку в таблицу можно выполнить двумя методами класса DataBase:

1. Result<void, Error<std::string>> insertRow(std::string tableName, Row &row) – данный метод выполняет вставку строки row в таблицу с именем tableName. Вставка выполняется с помощью полного сканирования (линейное время), то есть для каждой строки происходит сравнение с данной на совпадение ключей.

2. `Result<void, Error<std::string>> insertRow(std::string tableName, std::vector<Row> &rows)` – данный метод использует алгоритм вставки множества строк в таблицу из раздела 3. Выполняется данный алгоритм за квазилинейное время.

3.2 Запросы к базе данных и специальные таблицы

Помимо основного типа таблиц, существует специальный тип таблиц `TempTable`. Данный тип имеет следующие методы и ограничения:

1. `SpatialType getSpatialKeyType() const` – возвращает тип пространственной части ключа.
2. `TemporalType getTemporalType() const` – возвращает тип временной части ключа.
3. `bool isValid() const` – возвращает `true`, если приватное поле `bool valid` имеет истинное значение. Для всех созданных базой данных таблиц значение `valid` по умолчанию равно `true`.
4. `TempTable(TempTable const &table) = delete` – запрет конструктора копирования.
5. `void operator=(TempTable const &t) = delete` – запрет присваивания.
6. `void operator=(TempTable &&t) = delete` – запрет конструктора переноса.
7. `TempTable(TempTable &&table) = delete` – запрет переносящего присваивания.

Данный класс позволяют формировать выборки из строк таблиц. Такие таблицы создаются из основных таблиц с помощью метода класса `DataBase`: `Result<std::unique_ptr<TempTable>, Error<std::string>> selectTable(std::string tableName)`. Данный метод принимает в качестве аргумента имя основной таблицы `tableName` и возвращает уникальный указатель на таблицу типа `TempTable`. Данные таблицы не могут быть скопированы или перемещены (move-семантика), но использование класса `unique_ptr` позволяет выполнять перемещение. При создании таблицы данным методом происходит копирование всех строк исходной таблицы. В дальнейшем, при формировании множеств типа `SET` на основе `TempTable`-таблиц, `TempTable`-таблицы организуются в иерархию и копирования строк не происходит. При удалении родительской `TempTable`-таблицы (на основе которой построено множество

SET) происходит инвалидация (поле valid становится равным false) и всех дочерних таблиц. Рассмотрим операции, которые можно проводить только над TempTable-таблицами:

1. Если две TempTable-таблицы a и b имеют пространственный тип ключа POINT, и задано число k, такое что число строк в таблице b меньше либо равно k, то можно использовать метод Result<std::unique_ptr<TempTable>, Error<std::string>> pointxpointKnearestNeighbor(std::unique_ptr<TempTable> const &a, std::unique_ptr<TempTable> &b, uint k) класса DataBase и получить новую Temptable-таблицу, где строками являются строки таблицы a, для каждой из которых добавлен дополнительный столбец типа SET. Этот столбец представляет собой множество ссылок на строки таблицы b размера k, таких что точки из множества SET являются k ближайшими соседями для точки из строки a (учитывается только пространственная часть ключа). При реализации данного метода используется параллельная по строкам таблицы a реализация алгоритма поиска k ближайших соседей точки среди множества заданных точек (раздел 3).
2. Если TempTable-таблица a имеет пространственный тип ключа POLYGON, а TempTable-таблица имеет пространственный тип ключа POINT, то можно использовать метод Result<std::unique_ptr<TempTable>, Error<std::string>> polygonxpointPointsInPolygon(std::unique_ptr<TempTable> const &a, std::unique_ptr<TempTable> &b) класса DataBase и получить новую TempTable-таблицу, где строками являются строки таблицы a, для каждой из которых добавлен дополнительный столбец типа SET. Этот столбец представляет собой множество ссылок на строки таблицы b, таких что точка из таблицы b лежит внутри полигона из таблицы a. При реализации данного метода используется параллельная по строкам таблицы a версия алгоритма поиска всех точек, принадлежащих полигону (раздел 3).
3. Если TempTable-таблица a имеет пространственный тип ключа LINE, а TempTable-таблица b имеет пространственный тип ключа POINT, и задано число R > 0, то можно использовать ме-

тод Result<std::unique_ptr<TempTable>, Error<std::string>> linexpointPointsInBufferLine(std::unique_ptr<TempTable> const &a, std::unique_ptr<TempTable> &b, float radius) класса DataBase и получить новую TempTable-таблицу, где строками являются строки таблицы a, для каждой из которых добавлен дополнительный столбец типа SET. Этот столбец представляет собой множество ссылок на строки таблицы b, таких что точка из таблицы b лежит внутри полигона, образованного с помощью отступа на расстояние R от линии из таблицы a. При реализации данного метода используется параллельная по строкам таблицы a версия алгоритма поиска точек удалённых от заданной кривой не больше чем на R (раздел 3).

Теперь рассмотрим операции, которые применимы как к TempTable-таблицам, так и к основным таблицам:

1. Функция Result<void, Error<std::string>> update(std::string tableName, std::set<Attribute> const &atrSet, Predicate p) класса DataBase выполняет обновление строки в таблице. Она принимает название таблицы tableName, список атрибутов для изменения atrSet, и предикат p, который позволяет определить обновлять ли атрибуты для конкретной строки. Каждый атрибут из atrSet должен иметь имя атрибута и тип как в строке таблицы, это используется для идентификации обновляемого атрибута. Важно заметить, что нельзя обновлять атрибуты типа SET. Аналогичная функция для TempTable – функция Result<void, Error<std::string>> update(std::unique_ptr<TempTable> &t, std::set<Attribute> const &atrSet, Predicate p). Первым параметром функция принимает ссылку на уникальный указатель на TempTable, остальные параметры и принцип работы аналогичный описанному ранее.
2. Функция Result<void, Error<std::string>> dropRow(std::string tableName, Predicate p) принимает название таблицы tableName и предикат p. Предикат выполняется для всех строк таблицы, и если для строки он возвращает истинное значение, то строка удаляется из таблицы. Почти аналогичная функция для TempTable-таблиц – Result<std::unique_ptr<TempTable>, Error<std::string>> filter(std::unique_ptr<TempTable> &t, Predicate p). Пер-

вым аргументом она принимает ссылку на уникальный указатель на TempTable, вторым аргументом принимается предикат p. В отличии от функции dropRow, данная функция возвращает уникальный указатель на новую TempTable-таблицу, которая содержит ссылки на те строки, от которых предикат p вернул ложное значение.

3. Функция Result<void, Error<std::string>> showTable(std::string tableName) принимает название таблицы tableName и выполняет отображение таблицы в консоли. Аналогичная функция для TempTable-таблиц – Result<void, Error<std::string>> showTable(std::unique_ptr<TempTable> const &t). Первым аргументом она принимает ссылку на уникальный указатель на TempTable, остальные параметры и принцип работы аналогичны описанному ранее.
4. Функция Result<void, Error<std::string>> showTableHeader(std::string tableName) принимает название таблицы tableName и выполняет отображение названий ключей и их типов, названий атрибутов и их типов. Аналогичная функция для TempTable-таблиц: Result<void, Error<std::string>> showTableHeader(std::unique_ptr<TempTable> const &t). Первым аргументом она принимает ссылку на уникальный указатель на TempTable, остальные параметры и принцип работы аналогичны описанному ранее.
5. Функция Result<std::vector<Row>, Error<std::string>> selectRow(std::string tableName, Predicate p) принимает название таблицы tableName и предикат p и возвращает вектор строк, удовлетворяющих предикату p. Аналогичная функция для TempTable-таблиц: Result<std::vector<Row>, Error<std::string>> selectRow(std::unique_ptr<TempTable> &table, Predicate p). Первым аргументом она принимает ссылку на уникальный указатель на TempTable, остальные параметры и принцип работы аналогичны описанному ранее.

3.3 Написание предикатов для базы данных

Изначально в базе данных объявлен один предикат SELECT_ALL_ROWS, который для каждой строки таблицы возвращает true. Этот предикат позволяет, на-

пример, выполнить выборку сразу всех строк с помощью функции `selectRow`. Также база данных поддерживает написание собственных предикатов, для этого необходимо:

1. Создать новый заголовочный файл и подключить заголовочный файл `filter.h`.
2. Создать новый файл исходных текстов CUDA.
3. В заголовочном файле объявить предикат с помощью макроса `FILTER_H(имя_предиката)`.
4. В файле исходных текстов написать предикат, используя в качестве сигнатуры функции макрос `FILTER_CU(имя_предиката)`.
5. При написании предиката для обращения к строке в базе данных, для которой выполняется данный предикат, можно использовать константный объект `row` типа `CRow`.
6. Для передачи предиката в функцию используется функциональный вызов вида `имя_предиката()`.

Также с помощью макроса `FILTER_CU_FUNC(name)` можно объявлять функции, которые можно вызвать из предиката.

Теперь перейдём к описанию класса `CRow` и всех необходимых классов для работы со строками базы данных в фильтрах. При описании конструкторы классов будут опускаться, так как они используются только самой базой данных для создания этих объектов, также все приватные части классов описаны не будут. Из-за того что фильтры выполняются на стороне GPU, API доступа к строками БД для них отличается от API на CPU. Это сделано, во-первых, из-за невозможности использования классов типа `vector` и `string` на GPU, во-вторых, для увеличения скорости работы. Класс `CRow` реализует следующий набор методов:

1. `SpatialType getSpatialKeyType() const` – возвращает пространственный тип ключа.
2. `TemporalType getTemporalKeyType() const` – возвращает временной тип ключа.
3. `char const *getSpatialKeyName() const` – возвращает название пространственной части ключа.
4. `char const *getTemporalKeyName() const` – возвращает название временной части ключа.

5. `uint getColumnSize() const` – возвращает количество столбцов в таблице.
6. `Type getColumnType(uint id) const` – принимает номер столбца и возвращает его тип (см. листинг 2).
7. `bool getColumnIsNull(uint id) const` – принимает номер столбца и возвращает `true`, если в строке на этой позиции установлено значение `NULL`.
8. `char const *getColumnName(uint id) const` – принимает номер столбца и возвращает название столбца.
9. `int64_t getColumnINT(uint id) const` – принимает номер столбца. Если типом столбца является `INT`, то функция возвращает значение типа `int64_t`, иначе неопределённое поведение.
10. `char const *getColumnSTRING(uint id) const` – принимает номер столбца. Если типом столбца является `STRING`, то функция возвращает значение типа `char const*`, иначе неопределённое поведение.
11. `double getColumnREAL(uint id) const` – принимает номер столбца. Если типом столбца является `INT`, то функция возвращает значение типа `double`, иначе неопределённое поведение.
12. `CGpuSet getColumnSET(uint id) const` – принимает номер столбца. Если столбец имеет тип `SET`, то функция возвращает значение типа `CGpuSet`, иначе неопределённое поведение.
13. `Date getKeyValidTimeStart() const` – возвращает начало отрезка действительного времени, если типом временной части ключа является `VALID_TIME` или `BITEMPORAL_TIME`, иначе неопределённое поведение.
14. `Date getKeyValidTimeEnd() const` – возвращает конец отрезка действительного времени, если типом временной части ключа является `VALID_TIME` или `BITEMPORAL_TIME`, иначе неопределённое поведение.
15. `Date getKeyTransactionTime() const` – возвращает время транзакции, если типом временной части ключа является `TRANSACTION_TIME` или `BITEMPORAL_TIME`, иначе неопределённое поведение.
16. `CGpuPoint getKeyGpuPoint() const` – возвращает объект типа `CGpuPoint`, если пространственная часть ключа имеет тип `POINT`, иначе неопределённое поведение.

17. `CGpuPolygon getKeyGpuPolygon() const` – возвращает объект типа `CGpuPolygon`, если пространственная часть ключа имеет тип `POLYGON`, иначе неопределённое поведение.
18. `CGpuLine getKeyGpuLine() const` – возвращает объект типа `CGpuLine`, если пространственная часть ключа имеет тип `LINE`, иначе неопределённое поведение.

Класс `CGpuPoint` представляет собой одну точку и имеет единственный метод `float2 getPoint() const`, который возвращает точку типа `float2`. Класс `CGpuLine` представляет собой последовательность двумерных точек, формирующих ломанную. Данный класс реализует следующие методы:

1. `float2 getPoint(uint id) const` – возвращает `id`-ую точку последовательности.
2. `uint getPointNum() const` – возвращает количество точек в последовательности.

Класс `CGpuPolygon` представляет собой последовательность двумерных точек, которые формируют полигон в двумерном пространстве, ориентированный против часовой стрелки.

1. `float2 getPoint(uint id) const` – возвращает `id`-ую точку последовательности.
2. `uint getPointNum() const` – возвращает количество точек в последовательности.

Класс `CGpuSet` представляет собой множество ссылок на строки *какой-то* таблицы. Данный класс реализует следующие методы:

1. `CRow getRow(uint id) const` – принимает номер строки и возвращает объект типа `CRow`, который позволяет работать с данной строкой.
2. `uint getRowNum() const` – возвращает количество строк в множестве.

4 Тестирование функциональности базы данных

В данном разделе будет приведено несколько примеров задач, которые можно решить с помощью разработанной базы данных. Рассмотрим задачу об офисах оператора сотовой связи «ТЕЛЕ–3». Пусть компания «ТЕЛЕ–3» хочет узнать, какие абоненты были в определённом офисе в заданный промежуток времени. Для решения данной задачи создадим таблицу абонентов, где в качестве пространственной части ключа используется объект типа `POINT`, кодирующий зафиксированное, например с помощью GPS, положение абонента в пространстве. Также будем считать, что координаты автоматически переводятся из географической системы координат в двумерные декартовы координаты. В качестве временной части ключа выступает объект типа `VALID_TIME`. Он кодирует промежуток времени, в течении которого координаты абонента не менялись. Для упрощения будем считать, что у абонентов, помимо ключа, заданы только фамилия и имя. Также создадим таблицу офисов компании, где пространственным ключом является объект типа `POLYGON`, кодирующий границу офиса. В столбцах этой таблицы записаны название офиса и адрес типа `STRING`. В листинге 3 приводится код, выполняющий создание необходимых таблиц. Для генерации ключей таблицы «Абоненты ТЕЛЕ–3» используются

Листинг 3 — Создание таблиц для задачи с офисами «Теле–3».

```
1  DataBase &db = DataBase::getInstance();
2  TableDescription abonents, offices;
3  abonents.setName("Абоненты ТЕЛЕ–3");
4  abonents.setSpatialKey("Позиция", SpatialType::POINT);
5  abonents.setTemporalKey("Время", TemporalType::VALID_TIME);
6  AttributeDescription name, secondName;
7  name.name = "Имя"; name.type = Type::STRING;
8  secondName.name = "Фамилия"; secondName.type = Type::STRING;
9  abonents.addColumn(name); abonents.addColumn(secondName);
10 db.createTable(abonents);
11 offices.setName("Офисы ТЕЛЕ–3");
12 offices.setSpatialKey("Граница офиса", SpatialType::POLYGON);
13 offices.setTemporalKey("Время внесения в таблицу", TemporalType::
       TRANSACTION_TIME);
14 AttributeDescription street, officeName;
15 street.name = "Адрес"; street.type = Type::STRING;
16 officeName.name = "Название офиса"; officeName.type = Type::STRING;
17 offices.addColumn(street); offices.addColumn(officeName);
18 db.createTable(offices);
19 /*... заполнение таблицы ...*/
20 db.showTable("Абоненты ТЕЛЕ–3"); // консольный вывод 1
21 db.showTable("Офисы ТЕЛЕ–3"); // консольный вывод 2
```

точки $t \in [-1;1] \times [-1;1]$ и случайные промежутки времени с 2012 по 2016 годы включительно. В консольных выводах 1 и 2 показаны таблицы «Абоненты ТЕЛЕ–3» и «Офисы ТЕЛЕ–3». Пусть теперь компания «ТЕЛЕ–3» хочет полу-

```
Table "Абоненты ТЕЛЕ-3" [{
 0:  Key {[Позиция : Point : {0,680375, -0,211234}], [Время : Valid Time {2015/12/01 17:20:05:148160 - 2015/12/01 17:20:53:54464}]}
     Value {[Имя : STRING : "Ануфрий"] , [Фамилия : STRING : "Гришин"] }
 1:  Key {[Позиция : Point : {-0,329554, 0,536459}], [Время : Valid Time {2013/05/22 11:51:55:869440 - 2013/05/22 11:52:29:107712}]}
     Value {[Имя : STRING : "Лазарь"] , [Фамилия : STRING : "Нестеров"] }
 2:  Key {[Позиция : Point : {-0,270431, 0,026802}], [Время : Valid Time {2016/10/05 17:22:08:361984 - 2016/10/05 17:23:03:333632}]}
     Value {[Имя : STRING : "Елисей"] , [Фамилия : STRING : "Попов"] }
 3:  Key {[Позиция : Point : {-0,716795, 0,213938}], [Время : Valid Time {2012/01/30 18:44:50:872576 - 2012/01/30 18:45:05:445888}]}
     Value {[Имя : STRING : "Томас"] , [Фамилия : STRING : "Симонов"] }
 4:  Key {[Позиция : Point : {-0,686642, -0,198111}], [Время : Valid Time {2012/08/25 03:03:05:437696 - 2012/08/25 03:03:11:966208}]}
     Value {[Имя : STRING : "Абрам"] , [Фамилия : STRING : "Давыдов"] }
 5:  Key {[Позиция : Point : {0,025865, 0,678224}], [Время : Valid Time {2015/01/24 07:01:52:966656 - 2015/01/24 07:02:10:728448}]}
     Value {[Имя : STRING : "Елизар"] , [Фамилия : STRING : "Лукин"] }
 6:  Key {[Позиция : Point : {-0,012834, 0,945550}], [Время : Valid Time {2013/06/18 10:16:33:466624 - 2013/06/18 10:17:19:748096}]}
     Value {[Имя : STRING : "Карл"] , [Фамилия : STRING : "Самсонов"] }
 7:  Key {[Позиция : Point : {-0,199543, 0,783059}], [Время : Valid Time {2013/06/01 14:47:05:938176 - 2013/06/01 14:47:27:85568}]}
     Value {[Имя : STRING : "Варлаам"] , [Фамилия : STRING : "Фокин"] }
 8:  Key {[Позиция : Point : {-0,860489, 0,898654}], [Время : Valid Time {2014/08/18 23:50:38:819584 - 2014/08/18 23:50:43:982848}]}
     Value {[Имя : STRING : "Сильвестр"] , [Фамилия : STRING : "Одинцов"] }
 9:  Key {[Позиция : Point : {0,780465, -0,302214}], [Время : Valid Time {2012/04/27 05:47:02:666240 - 2012/04/27 05:47:03:867648}]}
     Value {[Имя : STRING : "Лукьян"] , [Фамилия : STRING : "Белоусов"] }
10:  Key {[Позиция : Point : {-0,523440, 0,941268}], [Время : Valid Time {2016/07/06 08:01:10:582272 - 2016/07/06 08:02:01:637376}]}
     Value {[Имя : STRING : "Рауф"] , [Фамилия : STRING : "Мамонтов"] }
11:  Key {[Позиция : Point : {-0,249586, 0,520498}], [Время : Valid Time {2014/07/25 09:39:02:328576 - 2014/07/25 09:39:42:392064}]}
     Value {[Имя : STRING : "Камиль"] , [Фамилия : STRING : "Артемьев"] }
12:  Key {[Позиция : Point : {-0,124725, 0,863670}], [Время : Valid Time {2016/08/27 14:08:52:464896 - 2016/08/27 14:09:35:721984}]}
     Value {[Имя : STRING : "Протас"] , [Фамилия : STRING : "Романов"] }
13:  Key {[Позиция : Point : {0,279958, -0,291903}], [Время : Valid Time {2015/06/10 17:20:46:465536 - 2015/06/10 17:20:56:423936}]}
     Value {[Имя : STRING : "Максимилиан"] , [Фамилия : STRING : "Трофимов"] }
14:  Key {[Позиция : Point : {0,658402, -0,339326}], [Время : Valid Time {2013/02/22 07:47:46:910208 - 2013/02/22 07:48:40:512512}]}
     Value {[Имя : STRING : "Никон"] , [Фамилия : STRING : "Панфилов"] }
15:  Key {[Позиция : Point : {0,912937, 0,177280}], [Время : Valid Time {2015/04/15 21:28:03:91200 - 2015/04/15 21:28:54:611712}]}
     Value {[Имя : STRING : "Малик"] , [Фамилия : STRING : "Фомин"] }
16:  Key {[Позиция : Point : {-0,203127, 0,629534}], [Время : Valid Time {2015/06/04 01:36:50:26240 - 2015/06/04 01:37:44:684544}]}
     Value {[Имя : STRING : "Кузьма"] , [Фамилия : STRING : "Гущин"] }
17:  Key {[Позиция : Point : {0,900505, 0,840257}], [Время : Valid Time {2012/09/26 18:35:46:777088 - 2012/09/26 18:36:39:640832}]}
     Value {[Имя : STRING : "Евстрат"] , [Фамилия : STRING : "Копылов"] }
18:  Key {[Позиция : Point : {0,239193, -0,437881}], [Время : Valid Time {2015/12/07 00:37:11:977984 - 2015/12/07 00:37:30:425344}]}
     Value {[Имя : STRING : "Май"] , [Фамилия : STRING : "Данилов"] }
19:  Key {[Позиция : Point : {-0,624934, -0,447531}], [Время : Valid Time {2014/10/13 14:56:44:817920 - 2014/10/13 14:57:09:807872}]}
     Value {[Имя : STRING : "Сухраб"] , [Фамилия : STRING : "Федотов"] }
20:  Key {[Позиция : Point : {-0,793658, -0,747849}], [Время : Valid Time {2014/06/24 04:13:53:154304 - 2014/06/24 04:14:38:782720}]}
     Value {[Имя : STRING : "Авксентий"] , [Фамилия : STRING : "Харитонов"] }
21:  Key {[Позиция : Point : {0,368890, -0,233623}], [Время : Valid Time {2015/10/01 19:57:13:148928 - 2015/10/01 19:57:35:268608}]}
     Value {[Имя : STRING : "Помпей"] , [Фамилия : STRING : "Денисов"] }
22:  Key {[Позиция : Point : {0,168977, -0,511174}], [Время : Valid Time {2012/10/05 09:59:15:337216 - 2012/10/05 09:59:59:266048}]}
     Value {[Имя : STRING : "Урман"] , [Фамилия : STRING : "Сергеев"] }
23:  Key {[Позиция : Point : {-0,671796, 0,490143}], [Время : Valid Time {2012/05/16 03:58:57:708032 - 2012/05/16 03:59:54:714112}]}
     Value {[Имя : STRING : "Цезарь"] , [Фамилия : STRING : "Логинов"] }
24:  Key {[Позиция : Point : {-0,647579, -0,519875}], [Время : Valid Time {2015/12/28 13:50:56:322304 - 2015/12/28 13:51:40:281600}]}
     Value {[Имя : STRING : "Дорофей"] , [Фамилия : STRING : "Шилов"] }
25:  Key {[Позиция : Point : {0,278917, 0,519470}], [Время : Valid Time {2012/06/19 18:55:55:64064 - 2012/06/19 18:56:03:158272}]}
     Value {[Имя : STRING : "Касьян"] , [Фамилия : STRING : "Бирюков"] }
26:  Key {[Позиция : Point : {-0,860187, -0,590690}], [Время : Valid Time {2014/04/23 00:21:53:702144 - 2014/04/23 00:22:42:882816}]}
     Value {[Имя : STRING : "Илиан"] , [Фамилия : STRING : "Савельев"] }
27:  Key {[Позиция : Point : {-0,896122, -0,684386}], [Время : Valid Time {2016/12/31 23:43:05:177088 - 2016/12/31 23:43:17:436672}]}
     Value {[Имя : STRING : "Арслан"] , [Фамилия : STRING : "Веселов"] }
28:  Key {[Позиция : Point : {0,995598, -0,891885}], [Время : Valid Time {2016/05/09 11:25:55:191040 - 2016/05/09 11:25:59:530752}]}
     Value {[Имя : STRING : "Ярополк"] , [Фамилия : STRING : "Фомин"] }
29:  Key {[Позиция : Point : {0,187784, -0,639255}], [Время : Valid Time {2012/10/25 00:59:23:928576 - 2012/10/25 00:59:47:429888}]}
     Value {[Имя : STRING : "Антон"] , [Фамилия : STRING : "Соловьёв"] }
}]
```

Консольный вывод 1: Абоненты «ТЕЛЕ–3».

```
Table "Офисы ТЕЛЕ-3" [{
0:  Key {[Граница офиса : Polygon : { {-1,000000, 0,000000} {0,000000, 0,000000} {0,000000, 0,800000} {-0,300000, 0,800000}
         {-0,300000, 0,500000} {-0,700000, 0,500000} {-0,700000, 0,800000} {-1,000000, 0,800000} }],
         [Transaction Time : {2016/08/09 18:37:28:449763}]}
    Value {[Адрес : STRING : "Улица "Никанор""], [Название офиса : STRING : "Офис имени "Маслов""] }
1:  Key {[Граница офиса : Polygon : { {1,000000, -0,100000} {0,000000, -0,600000} {-1,000000, -0,100000} }],
         [Transaction Time : {2016/08/09 18:37:28:449863}]}
    Value {[Адрес : STRING : "Улица "Израиль""], [Название офиса : STRING : "Офис имени "Красильников""] }
}]
```

Консольный вывод 2: Таблица «Офисы ТЕЛЕ–3».

чить выборку из базы для всех абонентов, которые были в первом офисе после

2013 года. Для этого напишем фильтр, который удалит все лишние строки в таблице «Абоненты ТЕЛЕ–3»:

```
FILTER_CU(tester){
    Date validE = row.getKeyValidTimeEnd();
    return (validE.getYear() < 2013);
}
```

В консольном выводе 3 приводится результат работы фильтра.

```
TempTable "Абоненты ТЕЛЕ-3" [{
 0:  Key {[Позиция : Point : {0,680375, -0,211234}], [Время : Valid Time {2015/12/01 17:20:05:148160 - 2015/12/01 17:20:53:54464}]}
     Value {[Имя : STRING : "Ануфрий"] , [Фамилия : STRING : "Гришин"] }
 1:  Key {[Позиция : Point : {-0,329554, 0,536459}], [Время : Valid Time {2013/05/22 11:51:55:869440 - 2013/05/22 11:52:29:107712}]}
     Value {[Имя : STRING : "Лазарь"] , [Фамилия : STRING : "Нестеров"] }
 2:  Key {[Позиция : Point : {-0,270431, 0,026802}], [Время : Valid Time {2016/10/05 17:22:08:361984 - 2016/10/05 17:23:03:333632}]}
     Value {[Имя : STRING : "Елисей"] , [Фамилия : STRING : "Попов"] }
 3:  Key {[Позиция : Point : {0,025865, 0,678224}], [Время : Valid Time {2015/01/24 07:01:52:966656 - 2015/01/24 07:02:10:728448}]}
     Value {[Имя : STRING : "Елизар"] , [Фамилия : STRING : "Лукин"] }
 4:  Key {[Позиция : Point : {-0,012834, 0,945550}], [Время : Valid Time {2013/06/18 10:16:33:466624 - 2013/06/18 10:17:19:748096}]}
     Value {[Имя : STRING : "Карл"] , [Фамилия : STRING : "Самсонов"] }
 5:  Key {[Позиция : Point : {-0,199543, 0,783059}], [Время : Valid Time {2013/06/01 14:47:05:938176 - 2013/06/01 14:47:27:85568}]}
     Value {[Имя : STRING : "Варлаам"] , [Фамилия : STRING : "Фокин"] }
 6:  Key {[Позиция : Point : {-0,860489, 0,898654}], [Время : Valid Time {2014/08/18 23:50:38:819584 - 2014/08/18 23:50:43:982848}]}
     Value {[Имя : STRING : "Сильвестр"] , [Фамилия : STRING : "Одинцов"] }
 7:  Key {[Позиция : Point : {-0,523440, 0,941268}], [Время : Valid Time {2016/07/06 08:01:10:582272 - 2016/07/06 08:02:01:637376}]}
     Value {[Имя : STRING : "Рауф"] , [Фамилия : STRING : "Мамонтов"] }
 8:  Key {[Позиция : Point : {-0,249586, 0,520498}], [Время : Valid Time {2014/07/25 09:39:02:328576 - 2014/07/25 09:39:42:392064}]}
     Value {[Имя : STRING : "Камиль"] , [Фамилия : STRING : "Артемьев"] }
 9:  Key {[Позиция : Point : {-0,124725, 0,863670}], [Время : Valid Time {2016/08/27 14:08:52:464896 - 2016/08/27 14:09:35:721984}]}
     Value {[Имя : STRING : "Протас"] , [Фамилия : STRING : "Романов"] }
10:  Key {[Позиция : Point : {0,279958, -0,291903}], [Время : Valid Time {2015/06/10 17:20:46:465536 - 2015/06/10 17:20:56:423936}]}
     Value {[Имя : STRING : "Максимилиан"] , [Фамилия : STRING : "Трофимов"] }
11:  Key {[Позиция : Point : {0,658402, -0,339326}], [Время : Valid Time {2013/02/22 07:47:46:910208 - 2013/02/22 07:48:40:512512}]}
     Value {[Имя : STRING : "Никон"] , [Фамилия : STRING : "Панфилов"] }
12:  Key {[Позиция : Point : {0,912937, 0,177280}], [Время : Valid Time {2015/04/15 21:28:03:91200 - 2015/04/15 21:28:54:611712}]}
     Value {[Имя : STRING : "Малик"] , [Фамилия : STRING : "Фомин"] }
13:  Key {[Позиция : Point : {-0,203127, 0,629534}], [Время : Valid Time {2015/06/04 01:36:50:26240 - 2015/06/04 01:37:44:684544}]}
     Value {[Имя : STRING : "Кузьма"] , [Фамилия : STRING : "Гущин"] }
14:  Key {[Позиция : Point : {0,239193, -0,437881}], [Время : Valid Time {2015/12/07 00:37:11:977984 - 2015/12/07 00:37:30:425344}]}
     Value {[Имя : STRING : "Май"] , [Фамилия : STRING : "Данилов"] }
15:  Key {[Позиция : Point : {-0,624934, -0,447531}], [Время : Valid Time {2014/10/13 14:56:44:817920 - 2014/10/13 14:57:09:807872}]}
     Value {[Имя : STRING : "Сухраб"] , [Фамилия : STRING : "Федотов"] }
16:  Key {[Позиция : Point : {-0,793658, -0,747849}], [Время : Valid Time {2014/06/24 04:13:53:154304 - 2014/06/24 04:14:38:782720}]}
     Value {[Имя : STRING : "Аксентий"] , [Фамилия : STRING : "Харитонов"] }
17:  Key {[Позиция : Point : {0,368890, -0,233623}], [Время : Valid Time {2015/10/01 19:57:13:148928 - 2015/10/01 19:57:35:268608}]}
     Value {[Имя : STRING : "Помпей"] , [Фамилия : STRING : "Денисов"] }
18:  Key {[Позиция : Point : {-0,647579, -0,519875}], [Время : Valid Time {2015/12/28 13:50:56:322304 - 2015/12/28 13:51:40:281600}]}
     Value {[Имя : STRING : "Дорофей"] , [Фамилия : STRING : "Шилов"] }
19:  Key {[Позиция : Point : {-0,860187, -0,590690}], [Время : Valid Time {2014/04/23 00:21:53:702144 - 2014/04/23 00:22:42:882816}]}
     Value {[Имя : STRING : "Илиан"] , [Фамилия : STRING : "Савельев"] }
20:  Key {[Позиция : Point : {-0,896122, -0,684386}], [Время : Valid Time {2016/12/31 23:43:05:177088 - 2016/12/31 23:43:17:436672}]}
     Value {[Имя : STRING : "Арслан"] , [Фамилия : STRING : "Веселов"] }
21:  Key {[Позиция : Point : {0,995598, -0,891885}], [Время : Valid Time {2016/05/09 11:25:55:191040 - 2016/05/09 11:25:59:530752}]}
     Value {[Имя : STRING : "Ярополк"] , [Фамилия : STRING : "Фомин"] }
}]
```

Консольный вывод 3: Результат работы фильтра.

Теперь можно сформулировать алгоритм для получения списка всех абонентов, которые были в первом офисе:

1. Создается `TempTable officesTT` по таблице «Офисы ТЕЛЕ–3».
2. Создается `TempTable abonentsTT` по таблице «Абоненты ТЕЛЕ–3».
3. Производится фильтрация таблицы `abonentsTT` с использованием фильтра `tester`.
4. Для каждого офиса из `officesTT` добавляется множество абонентов из `abonentsTT`, которые были внутри данного офиса (результат приведён в консольном выводе 4).

5. Производится выбор всех строк из таблицы `officesTT`, а затем из первой строки выбирается сформированное множество абонентов (результат приведён в консольном выводе 5).

В листинге 4 приводится код, выполняющий данную последовательность действий. На рисунке 1 представлена визуализация офисов и абонентов до и после применения фильтрации. Офисы представлены двумя геометрическими фигурами, а абоненты точками. На рисунке 2 приводится визуализация конечного результата.

Листинг 4 — Решение задачи.

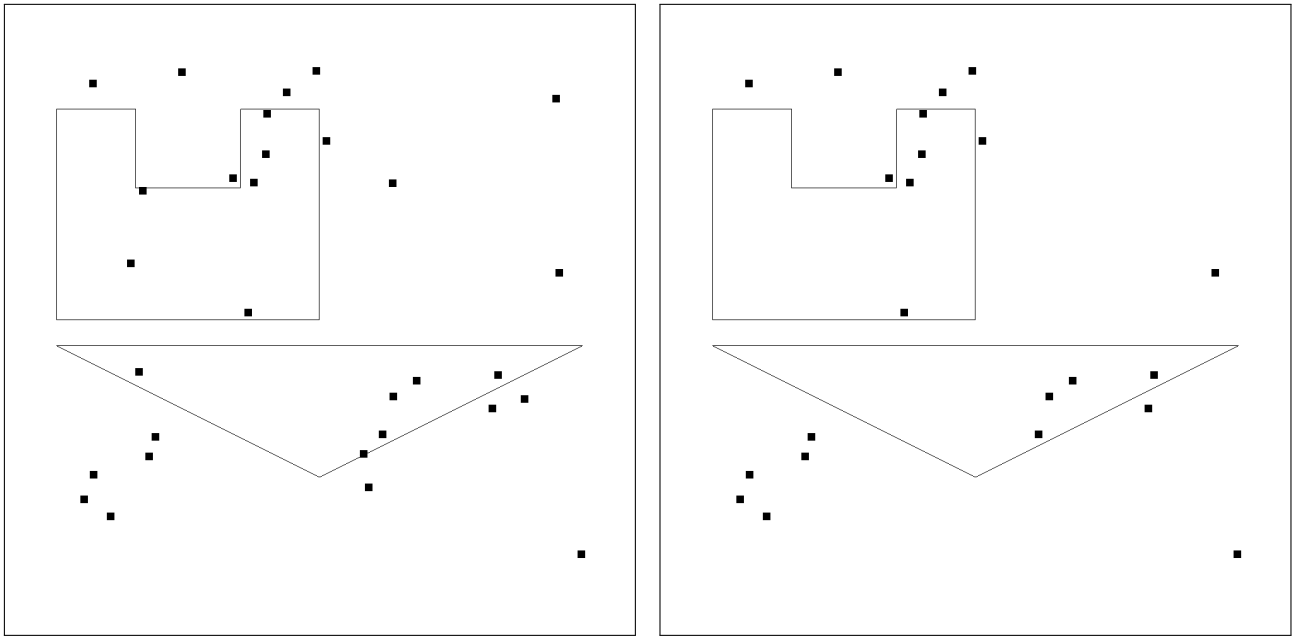
```
1 std::unique_ptr<TempTable> officesTT =db.selectTable("Офисы ТЕЛЕ-3").
      unwrap();//мы не хотим обрабатывать ошибки, поэтому unwrap()
2 std::unique_ptr<TempTable> abonentsTT = db.selectTable("Абоненты ТЕЛЕ-3"
      ).unwrap();
3 std::unique_ptr<TempTable> abonentsTT_filtered = db.filter(abonentsTT,
      tester());
4 std::unique_ptr<TempTable> output = db.polygonxpointPointsInPolygon(
      officesTT, abonentsTT_filtered).unwrap();
5 db.showTable(output);//консольный вывод 4
6 std::vector<Row> output_rows = db.selectRow(output, SELECT_ALL_ROWS());
7 db.showTable(output_rows[0].getAttribute(2).unwrap().getSet().unwrap());
      //консольный вывод 5
```

```
TempTable "Points inside Polygon operation result" [{
0:  Key {[Граница офиса : Polygon : { {-1,000000, 0,000000} {0,000000, 0,000000} {0,000000, 0,800000}
        {-0,300000, 0,800000} {-0,300000, 0,500000} {-0,700000, 0,500000} {-0,700000, 0,800000} {-1,000000, 0,800000} }],
        [Transaction Time : {2016/08/09 20:04:54:494987}]}
    Value {[Адрес : STRING : "Улица "Никанор""] , [Название офиса : STRING : "Офис имени "Маслов""] ,
          [Points inside Polygon operation result : SET :
          0:    Key {  [Позиция : Point : {-0,249586, 0,520498}],
                       [Время : Valid Time {2014/07/25 09:39:02:328576 - 2014/07/25 09:39:42:392064}]  }
                Value {  [Имя : STRING :   "Камиль"  ]   ,   [Фамилия : STRING :   "Артемьев"  ]   }
          1:    Key {  [Позиция : Point : {-0,203127, 0,629534}],
                       [Время : Valid Time {2015/06/04 01:36:50:26240 - 2015/06/04 01:37:44:684544}]  }
                Value {  [Имя : STRING :   "Кузьма"  ]   ,   [Фамилия : STRING :   "Гущин"  ]   }
          2:    Key {  [Позиция : Point : {-0,199543, 0,783059}],
                      [Время : Valid Time {2013/06/01 14:47:05:938176 - 2013/06/01 14:47:27:85568}]  }
                Value {  [Имя : STRING :   "Варлаам"  ]   ,   [Фамилия : STRING :   "Фокин"  ]   }
          3:    Key {  [Позиция : Point : {-0,270431, 0,026802}],
                      [Время : Valid Time {2016/10/05 17:22:08:361984 - 2016/10/05 17:23:03:333632}]  }
                Value {  [Имя : STRING :   "Елисей"  ]   ,   [Фамилия : STRING :   "Попов"  ]   }
] }
1:  Key {[Граница офиса : Polygon : { {1,000000, -0,100000} {0,000000, -0,600000} {-1,000000, -0,100000} }],
        [Transaction Time : {2016/08/09 20:04:54:495092}]}
    Value {[Адрес : STRING : "Улица "Израиль""] , [Название офиса : STRING : "Офис имени "Красильников""] ,
          [Points inside Polygon operation result : SET :

          0:    Key {  [Позиция : Point : {0,239193, -0,437881}],
                       [Время : Valid Time {2015/12/07 00:37:11:977984 - 2015/12/07 00:37:30:425344}]  }
                Value {  [Имя : STRING :   "Май"  ]   ,   [Фамилия : STRING :   "Данилов"  ]   }

          1:    Key {  [Позиция : Point : {0,279958, -0,291903}],
                       [Время : Valid Time {2015/06/10 17:20:46:465536 - 2015/06/10 17:20:56:423936}]  }
                Value {  [Имя : STRING :   "Максимилиан"  ]   ,   [Фамилия : STRING :   "Трофимов"  ]   }
          2:    Key {  [Позиция : Point : {0,368890, -0,233623}],
                       [Время : Valid Time {2015/10/01 19:57:13:148928 - 2015/10/01 19:57:35:268608}]  }
                Value {  [Имя : STRING :   "Помпей"  ]   ,   [Фамилия : STRING :   "Денисов"  ]   }
          3:    Key {  [Позиция : Point : {0,680375, -0,211234}],
                       [Время : Valid Time {2015/12/01 17:20:05:148160 - 2015/12/01 17:20:53:54464}]  }
                Value {  [Имя : STRING :   "Ануфрий"  ]   ,   [Фамилия : STRING :   "Гришин"  ]   }
] }
}]
```

Консольный вывод 4: Конечный результат.

```
TempTable ”Абоненты ТЕЛЕ-3” [{
0:  Key {[Позиция : Point : {-0,249586, 0,520498}], [Время : Valid Time {2014/07/25 09:39:02:328576 - 2014/07/25 09:39:42:392064}]}
    Value {[Имя : STRING : ”Камиль”] , [Фамилия : STRING : ”Артемьев”] }
1:  Key {[Позиция : Point : {-0,203127, 0,629534}], [Время : Valid Time {2015/06/04 01:36:50:26240 - 2015/06/04 01:37:44:684544}]}
    Value {[Имя : STRING : ”Кузьма”] , [Фамилия : STRING : ”Гущин”] }
2:  Key {[Позиция : Point : {-0,199543, 0,783059}], [Время : Valid Time {2013/06/01 14:47:05:938176 - 2013/06/01 14:47:27:85568}]}
    Value {[Имя : STRING : ”Варлаам”] , [Фамилия : STRING : ”Фокин”] }
3:  Key {[Позиция : Point : {-0,270431, 0,026802}], [Время : Valid Time {2016/10/05 17:22:08:361984 - 2016/10/05 17:23:03:333632}]}
    Value {[Имя : STRING : ”Елисей”] , [Фамилия : STRING : ”Попов”] }
}]
```

Консольный вывод 5: Конечный результат.



а) До фильтрации. б) После фильтрации.

Рисунок 1 — Применение фильтра *tester*.

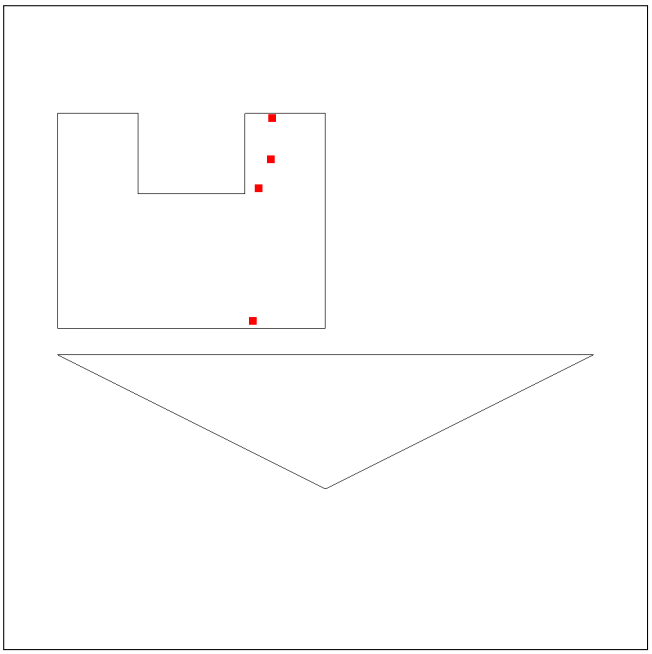


Рисунок 2 — Конечный результат.

Рассмотрим задачу о придорожных ресторанах и кафе. Пусть, для упрощения, первая таблица содержит только одну дорогу, а вторая таблица содержит различные строения. Будем считать придорожными все строения, которые удалены от данной дороги не более чем на пять метров (пусть один метр равняется `0.01f`). Требуется получить список всех придорожных ресторанов и кафе. Для решения задачи будем считать, что строения имеют пространственный тип ключа `POINT` и временной тип `TRANSACTION_TIME`, так как в данной задаче время никак не учитывается. Также строения имеют два атрибута: название и тип. Типом называем то, чем является данное строение (кафе, ресторан и т.д). Дорога задаётся ломаной линией (пространственный тип ключа `LINE`) и имеет атрибутом только название. Так же, как и в случае строений, временной тип равен `TRANSACTION_TIME`. В консольных выводах 6, 7 приводятся исходные таблицы.

```
Table "Дороги" [{
0:  Key {[Дорога : Line : { {0,100000; 0,100000}, {0,200000; 0,100000}, {0,100000; 0,200000}, {0,100000; 0,300000},
                            {0,300000; 0,400000}, {0,400000; 0,000000}, {0,500000; 0,000000}, {0,700000; 0,100000},
                            {0,500000; 0,400000}, {0,500000; 0,600000}, {0,400000; 0,600000}, {0,400000; 0,700000},
                            {0,800000; 0,700000}, {0,900000; 0,000000} }],
        [Transaction Time : {2016/08/18 20:23:15:421013}]}
    Value {[Название : STRING : "M4"]}
}]
```

Консольный вывод 6: Таблица «Дороги».

Для решения данной задачи, необходимо:

1. Найти все точки, удалённые от данной прямой на пять метров.
2. Произвести фильтрацию по атрибуту «тип».

В листинге 5 приводится решение данной задачи. Сначала производится создание двух `TempTable`-таблиц, затем с помощью функции `DataBase::linexpointPointsInBufferLine` происходит формирование `TempTable`-таблицы, которая содержит строку из таблицы «Дороги» с дополнительным атрибутом в виде множества строений, удалённых от данной дороги не более чем на пять метров. Далее происходит получение этого множества строений и отбрасывание лишних строк с помощью предиката из листинга 6. В консольном выводе 8 приводится конечный результат. На рисунке 3 приводится визуализация дороги и строений. Дорога представлена как ломаная, выделенная синим цветом. Вокруг дороги сформирован контур зелёного цвета, такой что точки, которые окажутся в нём, удалены от дороги не более, чем на пять метров. Красным выделены точки, которые попадают в

```
Table "Строения" [{
 0:  Key {[Позиция : Point : {0,498761; 0,033992}], [Transaction Time : {2016/08/18 20:25:45:209685}]}
     Value {[Название : STRING : "Улица "Мартин""], [Тип : STRING : "Стадион"]}
 1:  Key {[Позиция : Point : {0,603745; 0,006660}], [Transaction Time : {2016/08/18 20:25:45:209922}]}
     Value {[Название : STRING : "Улица "Добрыня""], [Тип : STRING : "Кафе"]}
 2:  Key {[Позиция : Point : {0,548661; 0,679430}], [Transaction Time : {2016/08/18 20:25:45:210012}]}
     Value {[Название : STRING : "Улица "Нифонт""], [Тип : STRING : "Кинотеатр"]}
 3:  Key {[Позиция : Point : {0,113908; 0,176936}], [Transaction Time : {2016/08/18 20:25:45:210094}]}
     Value {[Название : STRING : "Улица "Рихард""], [Тип : STRING : "Памятник"]}
 4:  Key {[Позиция : Point : {0,569275; 0,740459}], [Transaction Time : {2016/08/18 20:25:45:210158}]}
     Value {[Название : STRING : "Улица "Леон""], [Тип : STRING : "Отель"]}
 5:  Key {[Позиция : Point : {0,425463; 0,889715}], [Transaction Time : {2016/08/18 20:25:45:210239}]}
     Value {[Название : STRING : "Улица "Аникита""], [Тип : STRING : "Отель"]}
 6:  Key {[Позиция : Point : {0,622414; 0,590444}], [Transaction Time : {2016/08/18 20:25:45:210303}]}
     Value {[Название : STRING : "Улица "Серапион""], [Тип : STRING : "Отель"]}
 7:  Key {[Позиция : Point : {0,812990; 0,432279}], [Transaction Time : {2016/08/18 20:25:45:210365}]}
     Value {[Название : STRING : "Улица "Иннокентий""], [Тип : STRING : "Отель"]}
 8:  Key {[Позиция : Point : {0,986619; 0,311751}], [Transaction Time : {2016/08/18 20:25:45:210426}]}
     Value {[Название : STRING : "Улица "Рустам""], [Тип : STRING : "Ресторан"]}
 9:  Key {[Позиция : Point : {0,895191; 0,590364}], [Transaction Time : {2016/08/18 20:25:45:210509}]}
     Value {[Название : STRING : "Улица "Конрад""], [Тип : STRING : "Ресторан"]}
10:  Key {[Позиция : Point : {0,843541; 0,443852}], [Transaction Time : {2016/08/18 20:25:45:210572}]}
     Value {[Название : STRING : "Улица "Радомир""], [Тип : STRING : "Шаурма"]}
11:  Key {[Позиция : Point : {0,524824; 0,957449}], [Transaction Time : {2016/08/18 20:25:45:210633}]}
     Value {[Название : STRING : "Улица "Кирилл""], [Тип : STRING : "Церковь "Ктулху""]}
12:  Key {[Позиция : Point : {0,985158; 0,094099}], [Transaction Time : {2016/08/18 20:25:45:210696}]}
     Value {[Название : STRING : "Улица "Таврион""], [Тип : STRING : "Памятник"]}
13:  Key {[Позиция : Point : {0,084629; 0,410621}], [Transaction Time : {2016/08/18 20:25:45:210757}]}
     Value {[Название : STRING : "Улица "Мисаил""], [Тип : STRING : "Кинотеатр"]}
14:  Key {[Позиция : Point : {0,593411; 0,707043}], [Transaction Time : {2016/08/18 20:25:45:210818}]}
     Value {[Название : STRING : "Улица "Маркелл""], [Тип : STRING : "Кафе"]}
15:  Key {[Позиция : Point : {0,990508; 0,406400}], [Transaction Time : {2016/08/18 20:25:45:210881}]}
     Value {[Название : STRING : "Улица "Зафар""], [Тип : STRING : "Церковь "Ктулху""]}
16:  Key {[Позиция : Point : {0,835276; 0,977127}], [Transaction Time : {2016/08/18 20:25:45:210942}]}
     Value {[Название : STRING : "Улица "Эльдар""], [Тип : STRING : "Шаурма"]}
17:  Key {[Позиция : Point : {0,535143; 0,730468}], [Transaction Time : {2016/08/18 20:25:45:211024}]}
     Value {[Название : STRING : "Улица "Боримир""], [Тип : STRING : "Кафе"]}
18:  Key {[Позиция : Point : {0,111978; 0,378684}], [Transaction Time : {2016/08/18 20:25:45:211099}]}
     Value {[Название : STRING : "Улица "Вениамин""], [Тип : STRING : "Ресторан"]}
19:  Key {[Позиция : Point : {0,310029; 0,636802}], [Transaction Time : {2016/08/18 20:25:45:211163}]}
     Value {[Название : STRING : "Улица "Эмиль""], [Тип : STRING : "Церковь "Ктулху""]}
20:  Key {[Позиция : Point : {0,200736; 0,295187}], [Transaction Time : {2016/08/18 20:25:45:211224}]}
     Value {[Название : STRING : "Улица "Годфрид""], [Тип : STRING : "Церковь "Ктулху""]}
21:  Key {[Позиция : Point : {0,109493; 0,285365}], [Transaction Time : {2016/08/18 20:25:45:211288}]}
     Value {[Название : STRING : "Улица "Градимир""], [Тип : STRING : "Памятник"]}
22:  Key {[Позиция : Point : {0,131329; 0,702904}], [Transaction Time : {2016/08/18 20:25:45:211350}]}
     Value {[Название : STRING : "Улица "Ульманас""], [Тип : STRING : "Кинотеатр"]}
23:  Key {[Позиция : Point : {0,737324; 0,121837}], [Transaction Time : {2016/08/18 20:25:45:211422}]}
     Value {[Название : STRING : "Улица "Бруно""], [Тип : STRING : "Кинотеатр"]}
24:  Key {[Позиция : Point : {0,864016; 0,572601}], [Transaction Time : {2016/08/18 20:25:45:211494}]}
     Value {[Название : STRING : "Улица "Добрыня""], [Тип : STRING : "Ресторан"]}
25:  Key {[Позиция : Point : {0,503740; 0,399159}], [Transaction Time : {2016/08/18 20:25:45:211554}]}
     Value {[Название : STRING : "Улица "Лазарь""], [Тип : STRING : "Стадион"]}
26:  Key {[Позиция : Point : {0,272496; 0,615718}], [Transaction Time : {2016/08/18 20:25:45:211614}]}
     Value {[Название : STRING : "Улица "Федор""], [Тип : STRING : "Ресторан"]}
27:  Key {[Позиция : Point : {0,148018; 0,582525}], [Transaction Time : {2016/08/18 20:25:45:211678}]}
     Value {[Название : STRING : "Улица "Автандил""], [Тип : STRING : "Памятник"]}
28:  Key {[Позиция : Point : {0,698137; 0,348754}], [Transaction Time : {2016/08/18 20:25:45:211737}]}
     Value {[Название : STRING : "Улица "Игорь""], [Тип : STRING : "Церковь "Ктулху""]}
29:  Key {[Позиция : Point : {0,290392; 0,807630}], [Transaction Time : {2016/08/18 20:25:45:211800}]}
     Value {[Название : STRING : "Улица "Ермолай""], [Тип : STRING : "Шаурма"]}
30:  Key {[Позиция : Point : {0,758258; 0,421722}], [Transaction Time : {2016/08/18 20:25:45:211862}]}
     Value {[Название : STRING : "Улица "Жан""], [Тип : STRING : "Шаурма"]}
31:  Key {[Позиция : Point : {0,447293; 0,495582}], [Transaction Time : {2016/08/18 20:25:45:211923}]}
     Value {[Название : STRING : "Улица "Евстигней""], [Тип : STRING : "Кинотеатр"]}
32:  Key {[Позиция : Point : {0,308976; 0,311309}], [Transaction Time : {2016/08/18 20:25:45:211985}]}
     Value {[Название : STRING : "Улица "Назарий""], [Тип : STRING : "Церковь "Ктулху""]}
33:  Key {[Позиция : Point : {0,848639; 0,812716}], [Transaction Time : {2016/08/18 20:25:45:212065}]}
     Value {[Название : STRING : "Улица "Кронид""], [Тип : STRING : "Отель"]}
34:  Key {[Позиция : Point : {0,602288; 0,121135}], [Transaction Time : {2016/08/18 20:25:45:212137}]}
     Value {[Название : STRING : "Улица "Джамал""], [Тип : STRING : "Церковь "Ктулху""]}
}]
```

Консольный вывод 7: Таблица «Строения».

данный контур, но имеют тип, отличный от ресторанов и кафе. Фиолетовым выделены искомые строения.

Листинг 5 — Решение задачи про придорожные кафе и рестораны.

```
auto buildingTT = db.selectTable("Строения").unwrap();
auto roadTT = db.selectTable("Дороги").unwrap();
double radious = 0.05;
auto outTT = db.linexpointPointsInBufferLine(roadTT, buildingTT, radious
    ).unwrap();
std::vector<Row> linexpointRows = db.selectRow(outTT, SELECT_ALL_ROWS())
    .unwrap();
auto set = linexpointRows[0].getAttribute(1).unwrap().getSet().unwrap();
auto setfiltered = db.filter(set, roadFilter()).unwrap();
db.showTable(setfiltered);
```

Листинг 6 — Предикат оставляющий только кафе и рестораны.

```
bool FILTER_CU_FUNC(filter_strcmp)(char const *str1, char const *str2) {
    for (; *str1 && *str2 && *str1 == *str2; ++str1, ++str2) {}
    return *str2 == *str1;
}

FILTER_CU(roadFilter) {
    return !(filter_strcmp(row.getColumnSTRING(1), "Кафе")
             || filter_strcmp(row.getColumnSTRING(1), "Ресторан"));
}
```

```
TempTable "Строения" [{
0:  Key {[Позиция : Point : {0,593411; 0,707043}], [Transaction Time : {2016/08/18 20:25:45:210818}]}
       Value {[Название : STRING : "Улица "Маркелл""], [Тип : STRING : "Кафе"]}
1:  Key {[Позиция : Point : {0,864016; 0,572601}], [Transaction Time : {2016/08/18 20:25:45:211494}]}
       Value {[Название : STRING : "Улица "Добрыня""], [Тип : STRING : "Ресторан"]}
2:  Key {[Позиция : Point : {0,603745; 0,006660}], [Transaction Time : {2016/08/18 20:25:45:209922}]}
       Value {[Название : STRING : "Улица "Добрыня""], [Тип : STRING : "Кафе"]}
3:  Key {[Позиция : Point : {0,535143; 0,730468}], [Transaction Time : {2016/08/18 20:25:45:211024}]}
       Value {[Название : STRING : "Улица "Боримир""], [Тип : STRING : "Кафе"]}
}]
```

Консольный вывод 8: Придорожные кафе и рестораны.

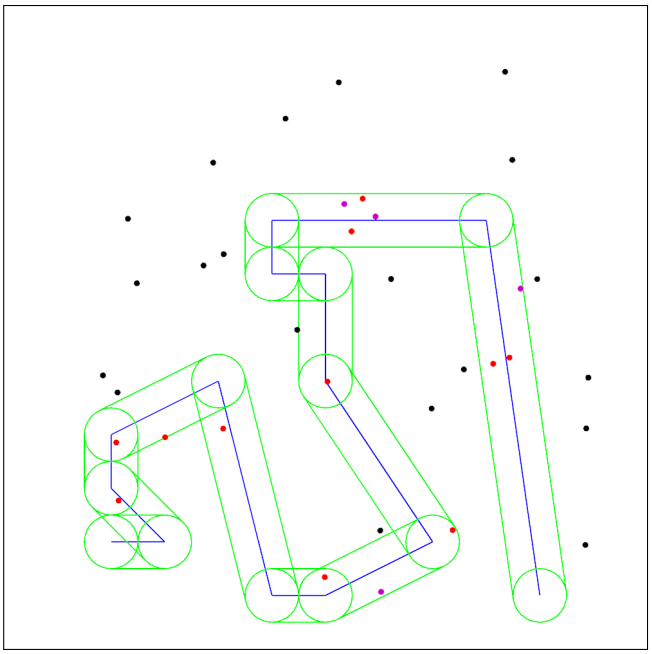


Рисунок 3 — Визуализация задачи про придорожные кафе и рестораны.

5 Тестирование ускоряющей структуры HLBVH2

В данном разделе проводится тестирование реализации HLBVH2. В таблице 1 приводятся системные параметры компьютера, на котором производилось тестирование.

Таблица 1 — Системные параметры.

ОС	GNU/Linux
Ядро	4.6.0 с патчами Gentoo
Дистрибутив	Gentoo
CPU	AMD FX-8150 Eight-Core
Частота, ГГц	3.6
RAM, ГБайт	8
GPU	GeForce GTX 580
VRAM, ГБайт	3
Шина VRAM, бит	512
Ядер CUDA	512

Целью тестирования является подтверждение теоретических оценок на скорость поиска и построения ускоряющей структуры HLBVH2. Так как при её построении используется только эвристический метод на основе кодов Мортона, то бинарное дерево получается несбалансированным, то есть если объекты в пространстве сильно пересекаются, то сложность операции поиска может возрасти до линейной. В лучшем случае это логарифмическая сложность от входных данных. Операция построения не должна зависеть от пересечения объектов в пространстве и должна иметь линейную сложность. Поэтому было решено произвести три теста с четырёхмерными точками, с четырёхмерными оболочками произвольного размера и с четырёхмерными оболочками фиксированного размера (ограничивающая оболочка задавалась двумя точками, первая генерировалась случайным образом, вторая получалась из первой с помощью смешения по каждой координате на одну десятую от максимального размера измерения) для каждой операции.

В таблице 2 приводится результат замеров скорости построения. Из неё видно, что скорости построения почти не отличается, несмотря на достаточно сильные пересечения второго типа объектов и отсутствие пересечений у первого типа. Для четырёхмерных точек методом наименьших квадратов была полу-

Таблица 2 — Замеры времени построения.

Количество объектов	Время построения, мс		
	Точки	Оболочки	Фикс. оболочки
500000	19	19	20
1000000	37	37	38
1500000	55	53	55
2000000	72	71	72
2500000	89	87	90
3000000	106	105	108
3500000	123	123	122
4000000	140	139	139
4500000	157	155	158
5000000	175	173	177
5500000	191	190	192
6000000	209	207	210
6500000	226	225	226
7000000	247	242	244
7500000	265	258	261
8000000	279	275	279
8500000	297	294	295
9000000	315	311	314
9500000	331	327	331
10000000	350	343	348

чена линейная функция $y = 0.000034x + 5.294737$, аппроксимирующая время построения объектов в зависимости от количества входных данных x. Данная функция подтверждает теоретическую оценку линейности числа операций при построении HLBVH2.

В виду параллельной архитектуры CUDA, тестирование поиска с использованием одного потока не является интересным. Поэтому для операции поиска запускалось 50 тысяч потоков, каждый из которых должен был найти свой объект. В таблице приводится суммарное время, которое потребовалось всем потокам. Как и ожидалось, для точек наиболее подходящими функциями стали $y = 1/(-0.000001x + 28.907776)$ и $y = 0.028931ln(x) - 0.379641$. Для оболочек фиксированного размера $y = 0.000045x^{0.748215}$, $y = 0.000001x + 0.705159$. А для оболочек произвольного размера наилучшее приближение дала линейная функция $y = 0.000466x + 16.708135$.

Таблица 3 — Замеры времени поиска.

Количество объектов	Время поиска, мс		
	Точки	Оболочки	Фикс. оболочки
500000	0.034	242.318	0.923
1000000	0.036	479.650	1.453
1500000	0.050	715.339	1.754
2000000	0.037	949.849	2.201
2500000	0.038	1183.929	2.633
3000000	0.041	1417.654	3.036
3500000	0.050	1651.327	3.434
4000000	0.052	1885.207	3.777
4500000	0.060	2118.706	4.138
5000000	0.053	2351.854	4.512
5500000	0.055	2584.414	4.883
6000000	0.057	2817.718	5.230
6500000	0.041	3050.580	5.636
7000000	0.053	3283.302	6.017
7500000	0.041	3514.895	6.372
8000000	0.055	3747.201	6.736
8500000	0.040	3980.177	7.106
9000000	0.043	4213.231	7.448
9500000	0.048	4444.680	7.806
10000000	0.043	4676.656	8.199

Заключение

В данной работе была разработана база данных с использованием технологии параллельного программирования CUDA. Был применён алгоритм построения HLBVH2, который позволил получить алгоритмически эффективные операции вставки в базу данных, поиска k ближайших соседей к заданной точке и другие. В дальнейшем можно наращивать возможности данной базы, в частности, добавить дополнительные функции для работы с контурами и ломаными, а также ввести дополнительные типы данных. Исходные коды работы выложены на GitHub (https://github.com/sargarass/spatial-database-coursework/).

Список литературы

[1] Garanzha K., Pantaleoni J., McAllister D. Simpler and faster hlbvh with work queues. — 2011. — URL: http://dl.acm.org/citation.cfm?id=2018333.

[2] Dignös A. Spatial databases. — 2014. — URL: http://www.inf.unibz.it/dis/teaching/TSDB/sl06.pdf.

[3] Guttman A. R-trees. a dynamic index structure for spatial search. — 1984. — URL: http://www-db.deis.unibo.it/courses/SI-LS/papers/Gut84.pdf.

[4] Dignös A. Abstract and concrete temporal data models. — 2014. — URL: http://www.inf.unibz.it/dis/teaching/TSDB/sl03.pdf.

[5] Roussopoulos N., Kelley S., Vincent F. Nearest neighbor queries. — 1995. — URL: http://postgis.refractions.net/support/nearestneighbor.pdf.

[6] Sunday D. Inclusion of a point in a polygon. — URL: http://geomalgorithms.com/a03-_inclusion.html.

[7] Modern gpu. — URL: https://nvlabs.github.io/moderngpu/.

[8] Cub library. — URL: https://nvlabs.github.io/cub/.

[9] Thrust library. — URL: https://thrust.github.io/.

[10] The opengl extension wrangler library. — URL: http://glew.sourceforge.net/.

[11] Glfw library. — URL: http://www.glfw.org/.