Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

ГРУППОВАЯ КУРСОВАЯ РАБОТА

Программный проект на тему

«Словари полигонов и geospatial структур»

Выполнили студенты группы 175, 3 курса,
Кваша Антон Михайлович
Петуховский Артур Михайлович
Чулков Андрей Сергеевич

Научный руководитель:
Руководитель группы разработки ClickHouse,
Миловидов Алексей Николаевич

Москва 2020

Содержание

1 Введение 6

1.1 Краткое описание работы 6

1.2 Постановка задачи 7

1.3 Актуальность 7

1.4 Цели и задачи 7

1.5 Полученные результаты 9

1.6 Структура работы 9

2 Описание предметной области 10

2.1 Используемые термины 10

2.2 ClickHouse 11

2.3 Словари в ClickHouse 11

2.4 Словари полигонов в ClickHouse 12

2.5 Запросы к словарю полигонов 12

2.6 Формальная постановка задачи 13

3 Обзор существующих решений 14

3.1 Характеристика релевантных работ 14

3.2 Классификация рассматриваемых работ и решений 15

3.3 Алгоритмы и реализации pointInPolygon 15

3.4 Алгоритмы из Polygon Gems 15

3.5 Реализация pointInPolygon в ClickHouse 17

3.6 Алгоритм Point Location 17

3.7 Наивная реализация алгоритма Point-in-Polygon Search 19

3.8 Реализация алгоритма Point-in-Polygon Search в платформе Vertica 20

3.9 Реализация RapidPolygonLookup 20

3.10 Выводы 21

4 Интерфейс словаря полигонов **22**
4.1 Реализация интерфейса словаря полигонов 22
4.2 Реализация наивного алгоритма 24

5 Алгоритм SlabsPolygonIndex **24**
5.1 Ускорение теста четности, используя идею из Point Location . . 24
5.2 Построение разбиения на плиты для теста четности 27
5.3 Решение Point-in-Polygon Search на основе ускоренного теста четности 29
5.4 Улучшение потребления памяти за счет использования дерева отрезков 32
5.5 Оптимизации без улучшения асимптотики 35
5.6 Реализация в ClickHouse 36

6 Алгоритм Grid **37**
6.1 Идея использования алгоритма сетки для ускорения поиска . . . 37
6.2 Интерфейс и реализация сетчатого алгоритма 37
6.3 Оптимизации сетчатого алгоритма 39
6.4 Реализация индекса POLYGON_INDEX_EACH 39
6.5 Реализация индекса POLYGON_INDEX_CELL 40

7 Тестирование и эксперименты **40**
7.1 Подготовка данных 40
7.2 Тестирование корректности 42
7.3 Тестирование производительности 43
7.3.1 Проведение экспериментов 43
7.3.2 Изменение детализации данных для ускорения поиска . 45

8 Использование разработанного функционала **45**
8.1 Пользовательская документация 45
8.2 Пример использования 45

9 Заключение 49

Аннотация

В данной работе рассматривается одна из задач работы с геоданными: определение района для большого количества геолокаций. Мы расскажем о решении обобщенного варианта данной задачи, в котором заданные регионы могут пересекаться, и последующем его внедрении в аналитическую систему управления базами данных ClickHouse. Хоть ClickHouse и не является geospatial базой данных, данный функционал позволит выполнять широкий пласт запросов на географических и геометрических данных. Например, с большого количества устройств могут поступать данные вместе с географическими координатами устройств, и быстро агрегировать такие данные по стране, городу или району может быть полезно. Мы расскажем об использованных нами алгоритмах для индексирования и поиска, а также про наш опыт добавления такого функционала в аналитическую базу ClickHouse. В процессе работы нам удалось реализовать несколько различных алгоритмов, которые последовательно улучшались с помощью экспериментов на реальных данных. В результате была реализована поддержка словарей с полигонами, в которых предоставленный список полигонов индексируется для быстрого поиска.

Abstract

This work stems from a classic geoprocessing task — given a large number of geolocations and a partition of the world into disjoint regions, you need to determine the region in which a query point lies. We will describe a solution for a generalized version of this problem, which allows for intersecting regions, and its following implementation as part of the open-source ClickHouse database management system. Even though ClickHouse is not a geospatial database, this functionality will allow for a wide range of geographical and geometrical queries. For example, if there is a large amount of data coming from many different devices along with their geographical coordinates aggregating this data by country, city or region could be quite useful. We will discuss the algorithms used for indexing the data and performing queries, as well

as our experience of contributing this functionality to ClickHouse. We implemented several different algorithms in the process of our work, which were consistently improved by performing real-data experiments. As a result, support for polygon dictionaries capable of indexing the provided polygons and achieving efficient querying times was implemented.

Ключевые слова

локализация точек, поиск точки в многоугольнике, геопространственные индексы, аналитические СУБД, ClickHouse, геометрические алгоритмы, бинарный поиск, дерево отрезков.

1 Введение

1.1 Краткое описание работы

Данная работа представляет собой проект по расширению возможностей популярной аналитической базы данных ClickHouse [1]. ClickHouse является столбцовой системой управления базами данных, разработанной в компании Яндекс, ориентированной на обработку аналитических запросов. В рамках данной работы необходимо реализовать конечный функционал, который будет доступен для использования всеми желающими.

База данных ClickHouse не предназначена для работы с геоданными [2], но тем не менее, предоставляет несколько функций для работы с ними. Одна из таких функций – pointInPolygon, позволяет проверить принадлежность точки к заранее заданному полигону[1]. Эта функциональность нужна, например, для агрегации мобильных устройств по координатам, для улучшения рекламной выдачи. В рамках данной работы необходимо реализовать похожий, но более общий функционал, который ClickHouse пока что не поддерживает. Требуется эффективно решать задачу поиска полигона, в которой попадает заданная точка, в словаре полигонов. В качестве примера можно рассмотреть список из всех стран, или областей одной страны, или районов одного города. В результате работы должен получиться функционал по построению словаря полигонов из такого списка, в котором поддерживается быстрый поиск страны (или области, или района) по заданной точке.

В рамках этой курсовой работы планируется разработка структуры данных, позволяющей построить некоторый индекс на наборе полигонов, который позволит эффективно разрешать запросы нахождения полигона, содержащего данную точку. Реализовать эту структуру планируется в рамках интерфейса внешних словарей, доступного в ClickHouse, используя комбинацию различных подходов, популярных в геопространнственном анализе.

[1] Термины точка, полигон и остальные описываются в следующей главе

1.2 Постановка задачи

Описание задачи легко сформулировать на практическом примере: по сохраненному разбиению России на районы требуется выполнять запросы локализации района содержащего координаты пользователя, получаемые, например, с GPS в реальном времени. В данной работе рассматривается обобщение этой задачи, в котором рассматриваемые районы могут состоять из нескольких несвязных областей, пересекаться и иметь самопересечения.

1.3 Актуальность

Данная работа предоставит текущим пользователям аналитической СУБД ClickHouse больше возможностей по работе с геоданными, в частности, с самым частым примером геоданных — собираемыми геолокациями, для их последующей классификации между разными областями и проведения аналитической деятельности. В текущей версии ClickHouse доступна функция pointInPolygon, которая умеет проверять принадлежность точки к полигону с фиксированными координатами, что не позволяет использовать ее для нужной классификации, по причинам неэффективности и необходимости подготовки запросов на каждую интересующую область.

Также данная работа добавляет новый функционал, который поможет привлечь новых пользователей СУБД ClickHouse, и использовать его в качестве варианта решения возникающих у них задач по обработке большого количество геолокаций.

1.4 Цели и задачи

Целью работы является добавление поддержки в ClickHouse словарей полигонов, которые предоставляют возможность эффективного определения полигона, в который попадает заданная точка. В качестве интерфейса для использования этого функционала будет использоваться интерфейс словарей [3] – од-

ной из существенных возможностей СУБД ClickHouse. Также, добавленный исходный код должен позволять с небольшим трудом добавлять новые подходы и алгоритмы.

Перед нами были поставлены следующие задачи:

- Проанализировать существующие подходы к схожим задачам, которые можно перенести на данную. **Этой задачей занимался Артур Петуховский.**

- Реализовать интерфейс словарей полигонов, позволяющий загружать данные из разных, в особенности внешних, источников для дальнейшего эффективного выполнения вышеописанных запросов. Важно, чтобы приведение данных к требуемому для загрузки формату выполнялось довольно просто. В частности, требуется поддерживать популярные форматы представления геометрических данных. **Этой задачей занимался Андрей Чулков**.

- Реализовать несколько различных алгоритмов, индексирующих имеющийся набор полигонов таким образом, чтобы итоговая структура не занимала слишком большое количество оперативной памяти и позволяла при этом эффективный поиск полигона, содержащего данную точку. **Этой задачей занимались Артур Петуховский и Андрей Чулков.**

- Используя данные с OpenStreetMaps, подготовить тесты, основанные на реальных данных, проверяющие корректность и быстродействие реализованного алгоритма. **Этой задачей занимался Антон Кваша.**

- Провести сравнительный анализ реализованных подходов на реальных географических данных. **Этой задачей занимался Антон Кваша.**

- Написать документацию реализованного функционала на английском и русском языках, содержащую актуальный пример использования словаря с реальными внешними данными достаточного размера. **Этой задачей**

занимался Антон Кваша. Переводом на английский язык занимался Андрей Чулков.

1.5 Полученные результаты

В рамках данной работы мы внедрили в ClickHouse поддержку словарей полигонов, а также реализовали несколько эффективных алгоритмов индексирования и поиска. Для реализованных возможностей были добавлены тесты корректности и производительности, написана пользовательская документация. Также данный функционал был успешно добавлен в основную кодовую базу ClickHouse, что делает его использование доступным для всех желающих.

1.6 Структура работы

В главе 2 описываются существующие элементы базы данных ClickHouse, а также описываются термины, необходимые для понимания содержимого данной работы.

В главе 3 производится обзор существующих решений и алгоритмов, которые могут быть полезны и использованы в дальнейшей работе.

В главе 4 описана реализация интерфейса словарей полигонов. **Этой главой занимался Андрей Чулков.** Описанная в этой главе работа была выполнена на языке C++ в формате пулл-реквеста[2] в репозиторий ClickHouse.

В главе 5 описана одна из реализаций алгоритма Indexed Point-in-Polygon Search. **Этой главой занимался Артур Петуховский.**

В главе 6 описана другая реализация алгоритма Indexed Point-in-Polygon Search и её комбинирование с алгоритмом из главы 5. **Этой главой занимался Андрей Чулков.**

В главе 7 описываются различные подходы к тестированию и проведению экспериментов. **Этой главой занимался Антон Кваша.**

[2] Ссылка на пулл-реквест: https://github.com/ClickHouse/ClickHouse/pull/8436

В главе 8 содержится пользовательская документация, а также пример использования словаря полигонов. **Этой главой занимался Антон Кваша.**

Описанная в главах 5-8 работа была выполнена на языке C++ в формате пулл-реквеста[3] в репозиторий ClickHouse.

В заключительной главе 9 делаются выводы касательно результатов данной работы.

2 Описание предметной области

2.1 Используемые термины

- Точка (англ. «point») — объект на плоскости, имеющий две координаты, каждая из которых является вещественным числом. В рамках данной работы в большинстве случаев в качестве точки мы будем рассматривать геопозицию, которая также имеет две координаты и задает какое-то место.

- Кольцо (англ. «ring») — упорядоченный набор вершин, проведя ребра через которые получается замкнутый цикл без самопересечений. Представляет связную область.

- Полигон (англ. «polygon) — внешнее кольцо, из которого вырезаны ноль или более не пересекающихся колец, вложенных во внешнее. Представляет связную область с дырками.

- Мультиполигон (англ. «multipolygon») — набор не пересекающихся полигонов. Дает возможность представить несвязную область, что часто встречается в геопространственных данных.

- Словарь полигонов (англ. «polygon dictionary») — структура, построенная на наборе полигонов, позволяющая по точке-запросу находить содержащий её полигон.

[3] Ссылка на пулл-реквест: https://github.com/ClickHouse/ClickHouse/pull/9278

- Индекс (англ. «index») — структура данных, которая требует дополнительной памяти и времени на ее построение, но позволяет ускорить последующие запросы. Получается в процессе индексирования (англ. «indexing»).

- Геопространственная база данных (англ. «geospatial database») — база данных, оптимизированная для работы с геометрическими и географическими сущностями.

2.2 ClickHouse

ClickHouse [1] является аналитической системой управления базами данных, разрабатываемой компанией Яндекс уже более 5 лет. С 2016 года исходный код проекта доступен по модели open-source на GitHub [4]. На момент написания данной курсовой работы основная разработка происходит по этому же адресу, с использованием возможностей GitHub и системы контроля версий git. Основным языком, используемым для разработки ClickHouse, является C++.

2.3 Словари в ClickHouse

Словари [5] в ClickHouse представляют из себя простое отображение из ключа в значение, или другими словами, поиск значения по ключу. Их удобно использовать в виде справочников для различных запросов.

ClickHouse поддерживает внешние словари. При создании таких словарей указывается источник, данные из которого периодически обновляются и динамически подгружаются в оперативную память, полностью или частично.

В интерфейсе словарей описана функция dictGet [3], использование которой интересует нас в первую очередь. Она используется для получения значения по ключу, и может использоваться для обращения к словарю полигонов.

У внешнего словаря доступно несколько настроек:

1. name — Идентификатор, под которым словарь будет доступен для использования. Для названия можно использовать символы латинского ал-

фавита, цифры, знак дефиса и нижние подчеркивания.

2. source — Источник словаря.

3. layout — Размещение словаря в памяти.

4. structure — Структура словаря. Ключ и атрибуты, которые можно получить по ключу.

5. lifetime — Периодичность обновления словарей.

2.4 Словари полигонов в ClickHouse

Словарь полигонов будет представлять новый вид размещения в памяти для внешних словарей. В качестве источника словаря будет передаваться расположение данных, предназначенного для последующей загрузки и построения индекса. В качестве layout будет передаваться название одной из нескольких предложенных реализаций, которые могут также принимать некоторые параметры для конфигурации. Поддержка частичного обновления данных в рамках данной работы не планируется.

2.5 Запросы к словарю полигонов

Запросом к словарю полигонов называется один вызов функции dictGet, которому на вход передается точка. Для измерения производительности запросов в дальнейшем будет использоваться метрика количества отработанных запросов за какой-то промежуток времени. Для большей достоверности, данная метрика будет получаться при выполнении всех запросов на одном процессорном ядре. Это замечание является важным, по причине того, что при поступлении большого количества запросов в ClickHouse, эти запросы автоматически разбиваются на блоки и обрабатываются параллельно.

2.6 Формальная постановка задачи

Необходимо решить задачу Indexed Point-in-Polygon Search, которая заключается в быстром ответе на запросы нахождения покрывающего точку полигона, по предоставленной точке и заранее известному набору полигонов.

В качестве ее решения необходимо реализовать интерфейс словаря полигонов, и на основе этого интерфейса реализовать несколько различных алгоритмов словаря полигонов, выбор из которых будет доступен с помощью опции конфигурации внешнего словаря layout.

Каждая такая реализация будет состоять из двух частей:

- Индексирование. В процессе индексирования происходит построение структуры по заданному набору полигонов. Эта операция вызывается ровно один раз для одного экземпляра структуры, и на ее время работы и количество используемой памяти не накладываются жесткие требования.

- Выполнение одного запроса. Используя заранее построенную структуру, ответить на запрос поиска покрывающего полигона по переданной в запросе точке. Эта операция вызывается большое количество раз, и от нее необходима максимальная эффективность.

Сравнительную характеристику требуемых реализаций можно задать в порядке следующего приоритета:

1. Время выполнения запросов.

2. Количество постоянно потребляемой структурой индекса памяти.

3. Время построения индекса.

Также для демонстрации результата нужно подготовить набор данных, и продемонстрировать быструю работу эффективного алгоритма поиска полигонов, по сравнению с наивной реализацией. Один из допустимых наборов данных — OpenStreetMap, который содержит свободно-распространяемые данные про административные границы различных городов, стран и районов.

3 Обзор существующих решений

3.1 Характеристика релевантных работ

Специализированных решений данной задачи в такой постановке не так много. Большинство продуктов, поддерживающих работу с геопространственными данными реализуют индексирование сложных объектив не самих объектов, а наименьших содержащих их прямоугольников. Например, так происходит в PostGIS, одном из самых популярных решений для работы с геометрическими данными [6]. Как правило, для индексирования используются квадро- [7], R- [8] или kd-деревья [9], позволяющие сузить число перебираемых полигонов, для которых уже используется существенно более медленная проверка принадлежности [10] [11]. Найти решения с открытой документацией внутреннего алгоритма, в которых построенный индекс используется не только для оптимизации поиска полигонов-кандидатов, но и запросов принадлежности к ним, на момент написания данной работы не получилось.

Направления работы по этой теме можно разделить на несколько категорий. К первой категории можно отнести оптимизации алгоритма проверки принадлежности точки полигону (pointInPolygon). Ко второй категории можно отнести задачу построения некоторого индекса на полигонах, позволяющего сузить число перебираемых полигонов-ответов, для которых выполняется проверка принадлежности. Отдельно интерес представляют алгоритмы планарной локализации точек, которые накладывают дополнительное условие того, что полигоны не имеют в себе дырок и не пересекаются друг с другом.

Готовые решения поставленной задачи в основном представляют из себя комбинацию или использование различных подходов из этих категорий и будут описаны в процессе обзора.

3.2 Классификация рассматриваемых работ и решений

- Алгоритмы и реализации pointInPolygon
- Алгоритм проверки вхождения точки в выпуклый полигон
- Алгоритм Point Location
- Примеры систем, с доступным функционалом Point-in-Polygon Search
- Алгоритмы и реализации Point-in-Polygon Search

3.3 Алгоритмы и реализации pointInPolygon

Алгоритм pointInPolygon является основополагающим в решении исходной задачи по нескольким причинам. Во-первых, исходная задача вырождается в этот алгоритм, если заданный набор из полигонов содержит единственный полигон. Во-вторых, само решение задачи и результат выполнения запроса result для множества полигонов P выражается через алгоритм pointInPolygon следующим образом:

$$\text{result} \in \{p \in P : \text{pointInPolygon}(\text{point}, p)\}$$

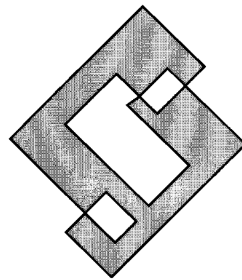
3.4 Алгоритмы из Polygon Gems

Рассмотрим алгоритмы и примеры реализаций pointInPolygon. Много информации про них доступно в статье [12] из книги Polygon Gems IV.

В этой статье полигон определяется как упорядоченный набор вершин, проведя ребра через которые получается замкнутый цикл. Также выделяется два важных типа полигонов: невыпуклые и выпуклые. Выпуклые полигоны обладают полезными свойствами, которые позволяют решать заданные задачи быстрее в частном случае.

Одно из определений принадлежности точки полигону называется «тест четности». Оно говорит, что точка лежит внутри полигона, если луч направ-

Рис. 3.1: Пример самопересечений



ленный в любую сторону пересекает ребра нечетное количество раз. В таком случае некоторые полигоны с самопересечениями (см. Рис. 3.1) не содержат внутри себя все покрываемые точки. Это соответствует нашему определению полигона — непокрытые области соответствуют вырезаемым из него дыркам.

В случае, когда такие точки нужно считать принадлежащими полигону, используется определение через «количество оборотов», которое считается как количество полных оборотов вокруг точки, при обходе точек в порядке полигона. При количестве оборотов равном нулю, точка считается не принадлежащей полигону.

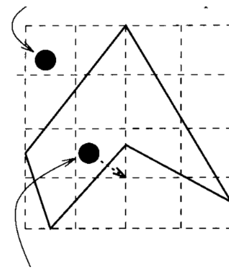
Важная оптимизация, также описанная в данной статье, называется на английском «bounding box». Ее суть заключается в подсчете ограничивающих координат во все четыре стороны и использованию их для быстрой проверки на непринадлежность точки полигону. Из-за её эффективности и простоты она часто используется перед вызовом любого другого алгоритма.

Реализация алгоритма, основанная на «тесте четности», является самой быстрой, среди реализаций без предварительной обработки. Она заключается в сдвиге точки на центр оси координат, и проведении луча согласно одной из главных осей. После этого, проверка на пересечение происходит как сравнение знаков координат точек ребра, что является очень быстрой операцией. Еще одним преимуществом данного алгоритма является возможность пропуска заведомо не пересекающих луч ребер, так как важно только количество пересечений.

Реализация, использующая «количество оборотов», не является такой быст-

рой, из-за необходимости производить сложные математические операции для расчета углов, или определения четверти осей координат.

Рис. 3.2: Пример сетки



Один из примеров более быстрого алгоритма использует предварительное построение сетки (см. Рис. 3.2) для ускорения работы. Для каждой клетки сетки она может быть полностью покрыта или не покрыта полигоном. В таком случае ответ на запрос сразу известен. Недостатком данного алгоритма является дополнительное потребление памяти.

Выпуклые полигоны обладают свойством, что любая прямая не пересекается с ними более чем два раза. Из-за этого свойства, алгоритм с «тестом четности» может быть ускорен до проверки не более чем двух возможных пересечений.

3.5 Реализация pointInPolygon в ClickHouse

База данных ClickHouse содержит встроенную реализацию функции pointInPolygon [13]. На момент написания данной работы, в ClickHouse используется несколько алгоритмов, включая построение сетки, оптимизацию «bounding box» и «тест четности».

3.6 Алгоритм Point Location

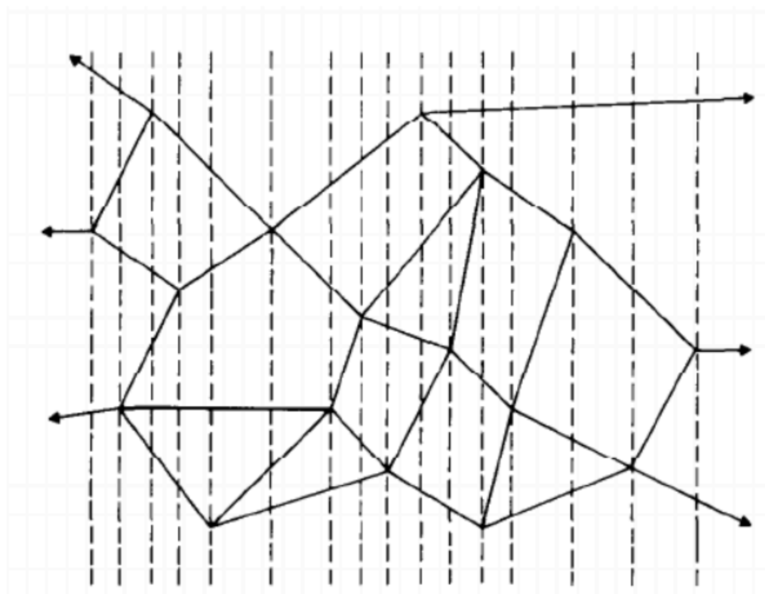
Существует задача Point Location, являющаяся частным случаем Point-in-Polygon Search, но при этом содержащая идеи, которые могут быть полезны в решении исходной задачи.

В этой задаче плоскость разбивается на полигоны, которые пересекаются только по сторонам. То есть, отличия от исходной задачи заключаются в отсутствии пересечений.

У такой задачи известно достаточно эффективное решение [14], которое требует $O(n)$ памяти, и способно отвечать на запросы за время $O(\log n)$, где n равно количеству вершин.

Данное решение работает следующим образом. Для каждой точки проводится вертикальная прямая, которая разделяет всю плоскость на вертикальные «плиты» (см. Рис. 3.3). После проведения таких прямых можно заметить, что каждая плита пересекает некоторое количество ребер. Важное свойство такого пересечения заключается в том, что ребро не может пересекать плиту частично. Так как ребра не пересекаются между собой (что следует из постановки задачи), то для каждой плиты можно упорядочить все пересекаемые ребра сверху вниз.

Рис. 3.3: Разбиение на вертикальные плиты



При наличии такого построенного разбиения на плиты, найти ответ на запрос можно с помощью двух бинарных поисков, за время $O(\log n)$. С помощью первого бинарного поиска можно найти вертикальную плиту, в которую попадает точка. Вторым бинарным поиском можно найти ближайшие к точке ребра, одно из которых будет принадлежать искомому полигону.

Для эффективного построения такого индекса, авторы используют алгоритм сканирующей прямой в комбинации с персистентными бинарными деревьями поиска. Такое построение требует $O(n \log n)$ времени.

Данный алгоритм является крайне эффективным, но его главный недостаток заключается в том, что он подходит только для решения частного случая исходной задачи.

3.7 Наивная реализация алгоритма Point-in-Polygon Search

Существует наивная реализация решения данной задачи, которая заключается в проверке попадания точки в каждый из заданных полигонов. Для данной реализации можно предложить оценку времени обработки одного запроса равную $O(n \cdot T(m))$, где $T(m)$ – время, необходимое на проверку принадлежности точки полигону, состоящему из m вершин.

Существует реализация поиска, использующая k-мерные деревья [11]. Данная реализация использует эврвистический подход. В этом подходе для каждого полигона эффективно вычисляется центроид, и происходит поиск ближайшей к точки с помощью k-мерных деревьев. В этих ближайших полигонах запускается аналог наивной реализации, то есть для ближайших полигонов запускается проверка принадлежности точки. У данного алгоритма нету оценки сверху на время выполнения, и в этом заключается один из его недостатков.

Также существуют реализации, использующие алгоритм R-дерева. Эти реализации похожи на предыдущие реализации с использованием К-мерных деревьев. В начале алгоритма производится поиск интересующих полигонов по их наборам минимальных и максимальных координат, убирая из рассмотрения все точно не включающие точку полигоны. Лучше всего работает на полигонах, которые почти не пересекаются. Среднее время работы данного алгоритма $O(\log N \cdot T(M))$. У данного алгоритма нету оценки сверху на время выполнения, которая была бы лучше наивного алгоритма, и в этом его главный недостаток.

3.8 Реализация алгоритма Point-in-Polygon Search в платформе Vertica

Платформа Vertica содержит функционал по решению задачи Indexed Point-in-Polygon Search. Этот функционал доступен в виде использования функций `STV_Create_Index` для построения индекса, и `STV_Intersect`[15] для выполнения запросов.

У этой реализации несколько недостатков. Главным недостатком является отсутствие открытого исходного кода и описания используемых алгоритмов. Вторым недостатком является отсутствие поддержки некоторых видов полигонов, таких как полигонов с дырками.

3.9 Реализация RapidPolygonLookup

Данная реализация[11] представляет собой оптимизацию «наивного» алгоритма, в котором для каждой точки запускается pointInPolygon для всех заданных полигонов. Время работы такого алгоритма на одном запросе составляет $O(M)$, где M —суммарное количество точек во всех полигонах. Данный «наивный» алгоритм является слишком неэффективным, и не подходит для использования даже при среднем количестве полигонов.

Несмотря на неэффективность данного алгоритма, к нему возможно применить несколько оптимизаций, улучшающих его быстродействие. Первой оптимизацией является использование более эффективного алгоритма pointInPolygon. Второй оптимизацией является эвристический подход, который заключается в эффективном сокращении списка рассматриваемых полигонов, с последующим запуском «наивного» алгоритма на сокращенном списке.

Данная статья основывается на второй оптимизации. В этом подходе для каждого полигона эффективно вычисляется центроид, и происходит поиск ближайшей точки с помощью kd-деревьев. Для этих ближайших полигонов запускается аналог «наивного» алгоритма.

У данного алгоритма нет оценки сверху на время выполнения, и в этом заключается один из его недостатков. Также, алгоритм полагается на эффективную реализацию pointInPolygon, и требует хотя бы одного ее запуска на искомом полигоне. Это значит, что алгоритм не способен работать быстрее функции pointInPolygon, которая может быть неэффективна на полигонах большого размера.

3.10 Выводы

Рассмотренные алгоритмы могут быть использованы для реализации оптимальной проверки pointInPolygon, а также содержат полезные идеи, которые могут быть использованы для решения поставленной задачи Indexed Point-in-Polygon Search. Готового исходного кода решающего поставленную задачу, который подходит для бесшовной интеграции в исходный код ClickHouse найдено не было.

В процессе курсовой работы планируется экспериментирование с разными алгоритмами, в поисках самого эффективного. В том числе, планируется объединить несколько из рассмотренных подходов, для достижения наилучших результатов.

В дальнейшей работе планируется использование теста четности, как самого эффективного варианта для основы pointInPolygon. Для быстрой предварительной проверки будет использоваться подход «bounding box». Для оптимизации количества рассматриваемых полигонов планируется использование алгоритма сетки. Также планируется использование идей из алгоритма Point Location, на основе которых планируется реализовать похожий алгоритм, но уже для решения нашей задачи.

4 Интерфейс словаря полигонов

4.1 Реализация интерфейса словаря полигонов

В качестве интерфейса для реализации поставленной задачи внутри кодовой базы ClickHouse были выбраны «внешние словари» [16]. В рамках этого интерфейса пользователю доступны несколько видов словарей, которые можно специальным образом конфигурировать. В частности, есть возможность указать источник данных на диске, из которых будет строиться заданный словарь. Построенные словари хранятся в оперативной памяти и предоставляют пользователю функции поиска по ним. Этот интерфейс наиболее соответствует ожидаемому функционалу.

В рамках работы был реализован новый вид словарей — словари полигонов. В основе всего лежит общий интерфейс IPolygonDictionary, который отвечает за загрузку и сохранение данных в некоторое внутреннее представление. Этот интерфейс реализует требуемые пользовательские функции для работы со внешними словарями [3], используя абстрактный метод для поиска полигона в этом внутреннем представлении. Наследникии этого интерфейса представляют конкретные реализации исследуемого алгоритма и реализуют этот абстрактный метод, имея возможность построить какую-то вспомогательную структуру в своем конструкторе на основе сохраненных в интерфейсе данных. Таким образом, добавление новых вариаций исследуемого алгоритма теперь не требует знания специфики работы внешних словарей в ClickHouse — достаточно реализовть аналоги класса, приведенного ниже.

```
class PolygonDictionaryImplementation : public IPolygonDictionary
{
public:
    PolygonDictionaryImplementation(...);
private:
    bool find(const Point & point, size_t & id) const override;
```

```
};
```

В конструкторе доступен массив `polygon`, содержащий список полигонов, хранящихся в словаре. С его помощью можно построить и сохранить в памяти любой индекс. Метод `find` использует этот индекс, чтобы найти в словаре точку `point`. Если она была найдена, то возвращается положительный результат и в параметр `id` записывается индекс найденного полигона в массиве `polygons`.

Ключевым моментом в реализации вышеописанного интерфейса был выбор формата читаемых и сохраняемых данных. Хранить данные было решено с помощью библиотеки Boost Geometry [17], предоставляющей большие возможности для работы с геометрическими примитивами. Одним из главных применений рассматриваемой задачи является работа с географическими данными, в которых могут встречаться острова, анклавы и т.д. Поэтому наиболее правильным является чтение данных в виде мультиполигонов. Этот формат популярен во многих продуктах для работы с географическими данными, например в PostGIS [18] и в спецификации GeoJson [19]. После чтения данные сохраняются в виде списка отдельных полигонов с дырками, отсортированных по площади, для каждого из которых хранится индекс мультиполигона, к которому он принадлежит. Наследники интерфейса должны реализовывать функцию поиска по точке таким образом, чтобы она возвращала первый найденный объект в порядке вышеописанной сортировки. Это разрешает конфликты принадлежности и полезно в том случае, когда объекты могут быть вложены друг в друга, как это часто бывает с административными границами на карте.

Для входных данных было предоставлено два различных формата, из которых может выбирать пользователь. В более простом варианте элементы словаря задаются как массив точек, то есть состоят из одного кольца. В более сложном варианте данные задаются как массив полигонов, соответствующий уже мультиполигону. Полигон в свою очередь задается как массив из одного или нескольких колец. Первое кольцо задаёт внешнюю границу полигона, а дальнейшие описывают вырезанные из него дырки и не должны пересекаться. Каж-

дое кольцо задается как массив своих точек. Таким образом, мультиполигон задается трехмерным массивом точек. Точка здесь и выше может задаваться как кортеж или массив своих координат.

4.2 Реализация наивного алгоритма

Наивная реализация не строит никакой дополнительной структуры при создании. Для каждой точки-запроса выполняется проход по всем сохранным полигонам с дырками и проверка принадлежности к каждому из них с помощью линйеного алгоритма, реализованного в Boost Geometry. При первой положительной проверке выдается ответ на запрос. Чтобы использовать эту реализацию алгоритма, в конфигурации словаря нужно указать тип размещения в памяти POLYGON.

5 Алгоритм SlabsPolygonIndex

В этой главе описывается алгоритм SlabsPolygonIndex, который является решением задачи Indexed Point-in-Polygon Search. Помимо самого алгоритма, описываются промежуточные алгоритмы и различные его оптимизации.

5.1 Ускорение теста четности, используя идею из Point Location

Рассмотрим алгоритм, использующий тест четности и идеи из Point Location. Основа этого алгоритма заключается в оптимизации теста четности, при направлении луча строго вниз, то есть в сторону уменьшения второй координаты. В тесте четности считается количество пересечений, которые совершает со всеми ребрами полигона луч, направленный в одну из сторон. Точка считается лежащей внутри полигона, если количество пересечений является нечетным. В самой простой реализации теста четности производится итерация по всем ребрам полигона, и для каждого ребра отдельно проверяется пересечение с ним. В случае луча, направленного строго вниз, проверка пересечения происходит в

два условия: проверка на то, что граница текущего ребра по первой координате включает в себя точку, то есть вертикальная прямая из точки пересекает ребро, и то, что точка находится выше прямой, образованной ребром.

Псевдокод описанного выше теста четности:

```
bool isInside(const std::vector<Edge>& edges, const Point& pt) {
    bool parity = false;
    for (const auto& edge : edges) {
        if (pt.x < edge.l.x || pt.x >= edge.r.x) {
            continue;
        }

        double edge_y = edge.l.y +
              (edge.r.y - edge.l.y)
            / (edge.r.x - edge.l.x)
            * (pt.x - edge.l.x);
        if (pt.y >= edge_y) {
            parity ^= true;
        }
    }

    return parity;
}
```

Используем идею из Point Location, которая заключается в проведении вертикальных прямых для всех точек (см. Рис. 3.3). После такого проведения, вся плоскость разбивается на так называемые плиты (на англ. «slab»). Важным свойством такого разбиения на плиты является то, что для каждой плиты любое ребро либо полностью ее покрывает, либо совсем не пересекается с данной плитой. Можно заметить, что это условие на покрытие или отсутствие пересечения совпадает с первой проверкой на пересечение в тесте четности с лучами на-

правленными строго вниз. А именно, если ребро полностью покрывает данную плиту, то проверка выполняется, а если полностью не покрывает — то проверка не выполняется.

Исходя из этого, можно предложить алгоритм, который для каждой плиты сохраняет все ребра, которые ее покрывают. При наличии таких сохраненных данных, алгоритм проверки с помощью теста четности может просматривать гораздо меньшее количество ребер. А именно, по точке находится плита в которую она попадает, и для каждого ребра из этой плиты проверяется второе условие, то что точка находится выше чем прямая, состоящая из этого ребра. Полученное число пересечений не отличается от полученного при проходе всех ребер числа. Визуально это можно представить так: проводится вертикальная прямая, и среди всех пересекающих эту прямую ребер считается количество тех, которые расположены ниже точки. Для такого алгоритма можно написать следующий псевдокод, который проверяет вхождение для построенной плиты:

```
bool isInsideSlab(const std::vector<Edge>& slab, const Point& pt) {
    bool parity = false;
    for (const auto& edge : slab) {
        double edge_y = edge.l.y +
              (edge.r.y - edge.l.y)
            / (edge.r.x - edge.l.x)
            * (pt.x - edge.l.x);
        if (pt.y >= edge_y) {
            parity ^= true;
        }
    }

    return parity;
}
```

Значительное ускорение запроса, по сравнению с тестом честности `isInside`,

достигается за счет уменьшения количества рассматриваемых ребер. Лучше всего такое уменьшение работает на выпуклых полигонах. Если представить выпуклый полигон, то можно понять, что любая вертикальная прямая не пересекает такой полигон более чем два раза, то есть не пересекается более чем с двумя ребрами. А это значит, что после нахождения нужной плиты, нужно будет проверить положение точки относительно ребра не более чем два раза. Асимптотическая сложность такой проверки $O(1)$, при известной плите.

Заметим, что если проводить вертикальные прямые не через все точки, а через какой-то другой набор координат, получится другое разбиение на плиты. Такое разбиение на плиты может не обладать свойством, при котором первое условие можно будет опустить, но все равно может уменьшать количество рассматриваемых ребер и быть полезным в задаче. Такой подход в данной работе дальше не рассматривается.

Плюс этого алгоритма заключается в ускорении запросов, по сравнению с обычным тестом четности.

Минус этого алгоритма заключается в необходимости препроцессинга полигонов, для построения всех плит.

5.2 Построение разбиения на плиты для теста четности

Можно описать алгоритм, который после построения будет явно хранить все плиты. Данный алгоритм является алгоритмом построения индекса для одного полигона, и включает в себя следующие шаги:

- Итерируемся по всем ребрам полигона, и складываем все ребра в отдельный массив
- Итерируемся по всем ребрам, и складываем первые координаты точек с двух концов ребра в отдельный массив
- Сортируем массив с первыми координатами точек, и затем с помощью одного прохода по массиву удаляем дубликаты координат

- Заводим массив под все плиты, его размер будет на единицу меньше массива с координатами

- Итерируемся по всем ребрам, и для каждого ребра с помощью алгоритма бинарного поиска находим позицию координат в массиве, в которых он начинается и заканчивается. После этого нужно проитерироваться по позициям всех плит, которые покрывает ребро, и добавить в соответствующие плиты текущее ребро.

Псевдокод данного алгоритма:

```
void buildSlabs(
    const Polygon& polygon,
    std::vector<std::vector<Edge>>& slabs,
    std::vector<double>& borders
) {
    const auto& edges = allEdges(polygon);
    const auto& points = allPoints(edges);
    borders = allCoordinatesX(points);
    borders = unique(sorted(borders));

    size_t n = borders.size();
    slabs.resize(n - 1);

    for (const auto& edge : edges) {
        size_t l = binarySearch(borders, edge.l.x);
        size_t r = binarySearch(borders, edge.r.x);
        for (size_t i = l; i < r; i++) {
            slabs[i].push_back(edge);
        }
    }
}
```

Пусть n — количество различных координат, m — количество ребер, T — суммарный размер всех плит после выполнения алгоритма. Данный алгоритм будет работать за $O(m \log n + T)$.

Для самой реализации проверки вхождения, нужно найти нужную плиту бинарным поиском, а затем использовать функцию `isInsideSlab`, псевдокод которой находится выше в тексте. Пусть C — количество ребер в плите. Данный алгоритм проверки вхождения будет работать за $O(\log n + C)$.

Минус этого алгоритма заключается в том, что он строит индекс только для одного полигона, и соответственно оптимизирует проверку вхождения тоже для одного полигона. Дальше в работе будет описываться модификация этого алгоритма, которая позволяет строить индекс сразу для нескольких полигонов.

5.3 Решение Point-in-Polygon Search на основе ускоренного теста четности

Заметим, что разбиение на плиты, описанное в оптимизации теста четности, можно точно так же строить на любом количестве полигонов одновременно. Для этого у всех этих полигонов берутся точки, и через все эти точки проводятся вертикальные прямые. В случае, если тест четности выполняется сразу для нескольких полигонов, то для однозначности можно выбирать полигон с меньшим идентификатором, который является целым числом. Построение такого индекса почти ничем не отличается от построения разбиения на плиты из предыдущего раздела. Единственное отличие заключается в том, что при построении необходимо доставать ребра из всех предоставленных полигонов.

Также можно описать алгоритм ответа на запрос. Данный алгоритм включает следующие шаги:

- Координаты всех плит были записаны на этапе построения индекса, поэтому можно найди нужную плиту, то есть плиту в которую попадает точка, за $O(\log n)$, используя для этого алгоритм бинарного поиска.

- После нахождения нужной плиты, нужно проитерироваться по всем ребрам, покрывающим данную плиту, и проверить положение точки относительно этих ребер. Если точка находится выше данного ребра, то нужно записать идентификатор полигона этого ребра в массив, для того чтобы потом провести с этими идентификаторами тест четности.
- Для проведения теста четности нужно найти минимальный идентификатор, который встречается нечетное количество раз. Для этого можно отсортировать все записанные идентификаторы, и итерируясь найти нужный минимальный, который встречается нечетное количество раз.

Псевдокод данного алгоритма:

```
bool SlabsPolygonIndex::find(const Point& pt, size_t& idx) {
    if (pt.x < this->borders[0] || pt.x >= this->borders.back()) {
        return false;
    }

    size_t slabIndex = binarySearchSlab(this->borders, pt.x);
    const std::vector<Edge>& slab = this->slabs[slabIndex];

    return findPolygonInSlab(slab, pt, idx);
}

bool findPolygonInSlab(
    const std::vector<Edge>& slab,
    const Point& pt,
    size_t& idx
) {
    std::vector<size_t> idxs;

    for (const auto& edge : slab) {
```

```
        double edge_y = edge.l.y +
              (edge.r.y - edge.l.y)
            / (edge.r.x - edge.l.x)
            * (pt.x - edge.l.x);
        if (pt.y >= edge_y) {
            idxs.push_back(edge.polygon_id);
        }
    }

    return findInsideIndex(idxs, idx);
}

bool findInsideIndex(conts std::vector<size_t>& idxs, size_t& res) {
    sort(idxs.begin(), idxs.end());
    for (size_t i = 0; i < idxs.size(); i += 2) {
        if (i + 1 == idxs.size() || idxs[i] != idxs[i + 1]) {
            res = idxs[i];
            return true;
        }
    }

    return false;
}
```

Данный псевдокод состоит из трех функций, каждая из которой является одним этапом алгоритма:

- `SlabsPolygonIndex::find` — производит поиск плиты за $O(\log n)$
- `findPolygonInSlab` — проверяет пересечение луча и ребер из плиты, оставляет индексы полигонов для всех пересечений, работает за $O(C)$

- `findInsideIndex` — ищет полигон с нечетным количеством пересечений, работает за $O(C \log C)$

Суммарная сложность такого алгоритма для поиска полигона составляет $O(\log n + C \log C)$.

Главный недостаток этого подхода находится на этапе построения индекса. В особых случаях, построение может занимать большое количество памяти и времени.

Например, рассмотрим случай, в котором исходные полигоны расположены друг над другом. Для простоты предположим, что каждый полигон является четырехугольником, самая левая и самая правая точка всех полигонов совпадает, а всего полигонов m штук. В случае, когда каждый полигон содержит две уникальные координаты, можно оценить количество различных координат как n, где $n = 2m+2$. Давайте оценим суммарное количество ребер во всех плитах, и обозначим его как T. В этом случае, каждый полигон будет добавлять суммарно $2(n-1)$ ребер во все плиты, и $T = 2m(n-1) = 2m(2m+1) = \Theta(m^2)$, что является квадратичной оценкой на память и время, которая зависит от количества полигонов.

5.4 Улучшение потребления памяти за счет использования дерева отрезков

Главный недостаток предыдущего алгоритма заключается в возможном неограниченно большом суммарном количестве ребер, положенных во все плиты. Обозначим это число за T. В самом худшем случае $T = \Theta(n^2)$, где n — количество ребер. В таком случае будет расходоваться очень большое количество памяти, и существует способ уменьшить количество используемой памяти до $O(n \log n)$.

Рассмотрим одно ребро относительно плит. Каждое ребро записывается в какой-то отрезок, состоящий из плит. Используя алгоритм дерева отрезков, можно записать каждое ребро не более чем в $O(\log n)$ вершин. Для этого, при построении индекса:

- Нужно инициализировать дерево отрезков, хранящее ребра соответствующие плитам
- Для любого отрезка из плит можно найти $O(\log n)$ вершин в дереве отрезков, в которые нужно записать эти плиты

Рассмотрим псевдокод алгоритма дерева отрезков:

```
void SegmentTree::init(size_t n) {
    this->n = n;
    this->tree = std::vector<std::vector<Edge>>(2 * n);
}

void SegmentTree::add(size_t l, size_t r, const Edge& edge) {
    for (l += n, r += n; l < r; l /= 2, r /= 2) {
        if (l & 1) tree[l++].push_back(edge);
        if (r & 1) tree[--r].push_back(edge);
    }
}

std::vector<Edge> SegmentTree::find(size_t pos) {
    std::vector<Edge> res;
    pos += n;
    do {
        res.append(tree[pos]);
        pos /= 2;
    } while (pos != 0);
    return res;
}
```

Данная реализация не использует рекурсию, и за счет этого может работать быстрее, потому что не тратит время на рекурсивные вызовы функции.

Идея алгоритма дерева отрезков заключается в том, что каждой вершине сопоставляется какой-то отрезок из массива. В данной реализации алгоритма, длина массива обозначается как n, и само дерево отрезков состоит из $2n - 1$ вершин и $\log n$ уровней. Самый нижний уровень состоит из n вершин, и каждая вершина из этого уровня называется листом. Каждая вершина нижнего уровня однозначно соответствует одной позиции массива. На уровне на один выше располагаются $\frac{n}{2}$ вершин, на котором одной вершине соответствуют две соседних позиции из массива. Для последующих уровней количество вершин уменьшается вдвое, по аналогии. Между вершинами разных уровней существует связь сын-родитель. Удобным свойством этой связи является отношение $p = \lfloor \frac{x}{2} \rfloor$, где p - вершина родитель, а x - вершина сын. То есть, индекс вершины родителя получается с помощью целочисленного деления на два. Данная операция зачастую оптимизируется компиляторами до операции побитового сдвига на один вправо, из-за ее эффективности по сравнению с операциями деления в общем случае.

Используя устройство дерева отрезков, опишем реализацию добавления элемента на полуинтервале позиций $[l'; r')$ в массиве.

- индексы массива l' и r' преобразуются в индексы вершин, по выражению $l = l' + n, r = r' + n$
- происходит итерация от нижнего уровня к верхнему, пока полуинтервал содержит хотя бы одну вершину
- на итерации происходит проверка левой и правой включенной вершины, и в нее добавляется элемент, если родительская вершина ей не соответствует
- вершины полуинтервала заменяются на родительские, которым соответствует такой же полуинтервал на массиве

А для реализации получения всех элементов для данной позиции массива, достаточно просто собрать все элементы из всех родительских вершин, начиная

с листовой. Только в этих вершинах будут содержаться нужные элементы.

При использовании такого подхода, используемое время и память, на построение индекса из плит, сокращается до $O(m \log n)$, где n — количество плит, и одновременно размер массива в дереве отрезков, а m — суммарное количество ребер во всех полигонах.

Теперь рассмотрим обработку запроса при использовании дерева отрезков. Асимптотика такого алгоритма остается такой же, но в самом алгоритме есть небольшие изменения. Для итерирования по всем ребрам для данной плиты нужно сделать запрос в дереве отрезков, который можно совершить за $O(\log n)$.

Данный алгоритм в некоторых случаях может обрабатывать запросы немного медленнее, из-за того что нужно совершить подъем в дереве отрезков, но этот недостаток нивелируется значительной экономией памяти.

Плюсом данного алгоритма является скорость построения и объем занимаемой памяти, по сравнению с предыдущими версиями.

5.5 Оптимизации без улучшения асимптотики

В этом разделе описываются оптимизации времени и памяти, которые не влияют на асимптотику алгоритмов.

Одной из оптимизаций является уменьшение количества арифметических операций в функции `findPolygonInSlab`. В ней происходит вычисление следующей функции:

$$f(x) = ly + (ry - ly)/(rx - lx) \cdot (x - lx)$$

В этом выражении lx, ly, rx, ry это координаты левой и правой точек ребра, а x это координата точки. Данная функция является линейной, а это значит ее можно привести к виду $f(x) = kx + b$.

$$k = (ry - ly)/(rx - lx)$$

$$b = ly - k \cdot lx$$

Оптимизация заключается в предпросчете данных параметров на этапе построения индекса. Во время запросов используются эти числа, и за счет этого получается уменьшение арифметических операций и ускорение запросов, за счет незначительного замедления построения.

Следующая оптимизация связана с предыдущей, и заключается в оптимизации структуры. Фактически, теперь во время запроса для каждого ребра используются только три поля:

- `k`, тип `double`, размер 64 бита
- `b`, тип `double`, размер 64 бита
- `polygon_id`, тип `size_t`, размер 64 бита

Изначальная структура содержит еще 4 числа типа `double`, которые хранят координаты точек ребра. Если создать отдельную структуру, то можно сэкономить 256 бит на одно ребро, и тем самым уменьшить количество памяти на ребро более чем в два раза.

Следующая оптимизация заключается в использовании 32-битного типа `float`, вместо 64-битного типа `double`. Количество памяти при использовании такого типа сокращается на 64 бита для одного ребра. Недостаток такого типа заключается в меньшей точности. Несмотря на это, точности этого типа хватает для задания географических координат с точностью до двух метров.

5.6 Реализация в ClickHouse

Алгоритм реализован на языке C++ в виде класса SlabsPolygonIndex. Включен в кодовую базу ClickHouse и доступен для переиспользования.

6 Алгоритм Grid

6.1 Идея использования алгоритма сетки для ускорения поиска

Для того, чтобы сузить число объектов, для которых вызывается проверка на принадлежность можно реализовать структуру и алгоритм, в чем-то напоминающий квадродерево [7]:

1. Для набора полигонов с дырками (объектов) считается минимальный содержащий их прямоугольник.

2. Этот прямоугольник делится на 4^2 равных частей (ячеек), для каждой из которых вычисляются объекты, которые она пересекает.

3. Алгоритм продолжается рекурсивно, останавливаясь, когда прямоугольник пересекает малое число объектов, или была достигнута слишком большая глубина.

4. В листовых ячейках сохраняется некоторая информация о пересекающих её полигонах. В самом простом варианте этой информацией является список их индексов.

5. В дальнейшем, при выполнения запроса, рекурсивно находится наименьшая по площади ячейка, содержащая данную точку, после чего сохраненная в ней информация используется для ответа на запрос.

6.2 Интерфейс и реализация сетчатого алгоритма

Для реализации вышеописанного алгоритма было широко использованы парадигмы наследования и виртуальных методов. Интерфейс ячеек описывается следующим упрощенным шаблонным виртуальным классом:

```
template <class ReturnCell>
class ICell
{
public:
    virtual const ReturnCell * find(Coord x, Coord y) const = 0;
};
```

В шаблонный параметр передается класс, реализующий листовую ячейку, что позволяет использовать разные вариации этого алгоритма для различных реализаций словаря. Этот интерфейс удобен тем, что позволяет пользоваться различными ячейками однообразным образом. Листовые ячейки в этом методе возвращают указатель на себя. Промежуточные ячейки сопоставляют точку правильной ячейке следующей глубины и вызывают у неё метод `find`.

Сама сетка строится с помощью класса `GridRoot`, также унаследованного от `ICell`. В конструкторе он принимает список полигонов, а также два параметра для построения сетки — максимальную глубину рекурсии и число пересечений, при достижении которого рекурсия останавливается. В конфигурации словарей эти параметры называются MAX_DEPTH и MIN_INTERSECTIONS соответственно. Далее простым перебором координат находится минимальный прямоугольник, содержащий все полигоны, после чего на нем запускается алгоритм построения сетки.

Алгоритм построения соответствует описанию в разделе выше и реализован с помощью рекурсивного метода `makeCell`. Для проверки пересечений используются алгоритмы из библиотеки Boost Geometry. Важным аспектом является то, что полигоны обрабатываются и сохраняются в ячейках в том же относительном порядке, в котором они были в переданном в конструктор массиве.

Для использования получившегося индекса нужно вызвать у построенного класса `GridRoot` метод `find`, передав ему координаты точки-запроса. Возвращаемая в результате ячейка позволяет тем или иным образом, в зависимости от реализации, найти первый полигон в переданном при построении порядке, содержащий данную точку.

В рамках работы было реализовано две вариации листовых ячеек. В первой, `FinalCell`, просто сохраняются индексы пересекающих её полигонов. Во второй, `FinalCellWithSlabs`, вычисляется пересечение этих полигонов с границами ячейки, и на этих пересечениях строится индекс из главы 5.

6.3 Оптимизации сетчатого алгоритма

Вызовы `makeCell` для каждой из четырех «строчек» сетки на глубине рекурсии один и два выполняются в новом потоке. Таким образом, построение этой структуры может выполняться параллельно с использованием вплоть до 16 потоков. Это позволяет существенно снизить время построения такой структуры для больших наборов данных.

Также, очень существенные результаты дала оптимизация случаев, когда ячейка целиком лежит в некотором полигоне. Для обоих типов листовых ячеек сохраняется индекс первого из таких полигонов, или факт его отсутствия. В первом случае все полигоны с большим индексом не сохраняются и не обрабатываются в этой ячейке. Это корректно, так как точка, лежащая в некоторой ячейке, заведомо принадлежит полигону, содержащему данную ячейку. Мы же ищем лишь первый полигон в известном заранее порядке.

Оптимизация из предыдущего абзаца также была использована для более эффективных затрат памяти. При подсчете количества полигонов, пересекающих ячейку, игнорируются полигоны, содержащие её целиком.

6.4 Реализация индекса POLYGON_INDEX_EACH

При создании эта реализация словаря строит структуру, описанную в предыдущей секции, с использованием листовой клетки `FinalCell`. Также строится структура из главы 5 для каждого полигона по-отдельности. При выполнении запроса, извлекается ячейка сетки, содержащая данную точку, после чего кандидаты, сохраненные в этой ячейке, проверяются с помощью построенных для них структур `SlabsPolygonIndex`.

6.5 Реализация индекса POLYGON_INDEX_CELL

При создании эта реализация словаря строит структуру, описанную в предыдущей секции, с использованием листовой клетки `FinalCellWithSlabs`. При выполнении запроса, извлекается ячейка сетки, содержащая данную точку, после чего к выполняется запрос поиска в сохраненной в этой ячейке структуре `SlabsPolygonIndex`.

7 Тестирование и эксперименты

7.1 Подготовка данных

Одной из целей нашего проекта является тестирование реализованных алгоритмов на карте реального мира. Разумеется, нам были необходимы данные для тестирования корректности реализаций, то есть было необходимо получить информацию о границах всех мировых субъектов с помощью сторонних ресурсов. Для этого использовались данные с OpenStreetMap [20] — веб-картографического проекта по созданию географической карты мира. С помощью данного ресурса мною были получены мировые границы, после чего их необходимо было преобразовать в подходящий для ClickHouse формат входных данных таблиц. Для этого мною были изучены разные способы и форматы получения данных с OpenStreetMap, после чего был выбран наиболее удобный для работы формат GeoJson [21]. Мною были реализованы инструменты для конвертации полученных данных в формат JsonEachRow – один из форматов, допустимых в ClickHouse, имеющий сходство с известным форматом json. После этого необходимо было определить, как именно планируется хранить данные, чтобы их было удобно использовать при создании словарей. Необходимо было сделать 3 принципиально различных типа датасетов:

- Несколько небольших наборов данных для проверки корректности реализаций.

- Датасет большего размера для тестирования производительности реализаций.

- Датасет, содержащий информацию о всем мире, чтобы пользователь ClickHouse мог самостоятельно выбрать необходимые ему объекты.

К хранению всех типов было несколько возможных подходов:

- В момент инициализации словарей обращаться к API OpenStreetMap, после чего используя реализованные мною инструменты получать данные в подходящем формате и сразу строить на их основе словари. Данный подход не требовал свободного места на диске для хранения данных и подходил для обоих типов датасетов. К сожалению, данный подход оказался крайне сложным технически, и обращение к внешнему ресурсу в ClickHouse в данный момент является отдельной задачей, поэтому от данного варианта было решено отказаться.

- Хранить все заранее подготовленные данные мировых границ прямо на диске, после чего строить словарь, указав путь к файлу. Очевидная проблема такого подхода —объем расходуемой памяти, размер мировых данных превышает 90GB. Поэтому данный вариант рассматривался только для первого типа датасета, проверяющего корректность алгоритмов, то есть относительного небольшого датасета в несколько мегабайт. После обсуждения с руководителем проекта, было решено остановиться на данном варианте и хранить данные в сжатом виде, разархивируя их при инициализации словарей, еще больше экономя потребление памяти. Позднее было принято решение воспользоваться данным способом и для хранения второго датасета, так как в сжатом виде он занимал достаточно мало места на диске.

- Хранить все заранее подготовленные данные на статических серверах ClickHouse, после чего обращаться к ним, как к части ClickHouse. Преимущество данного вариант состоит в доступности данных для пользо-

вателей —любой пользователь ClickHouse может скачать и использовать их. Этот способ был выбран для хранения мировых данных.

7.2 Тестирование корректности

Для проверки корректности реализованных алгоритмов, а так же определения наиболее эффективного алгоритма необходимо подготовить тесты. Изначально тестирование проводилось на простейших геометрических фигурах, таких как треугольник и квадрат. Однако для полноценного тестирования необходим был большой датасет данных. Затем был выбран небольшой датасет – в рамках одной страны, для тестирования алгоритмов на данных большего размера. На этом датасете была проверена работоспособность каждого алгоритма – одной из реализаций алгоритма поиска был линейный поиск, то есть заведомо верное решение. Мною был реализованы тесты, которые запускали все существующие решения на этих данных, после чего результаты работы алгоритмов сравнивались между собой. Каждый тест это набор SQL-инструкций, который имеет следующую структуру:

- Инициализацию словаря полигонов, используя передаваемый датасет.
- Инициализация таблицы, содержащий набор точек, местоположение которых хочет определить пользователь. На простейшем датасете значения точек инициализировались вручную, на всех остальных точки случайно генерировались внутри области датасета.
- Выполнение запроса поиска к словарю.

С помощью этого были найдены несколько критических ошибок реализаций, которые не были выявлены на простейших геометрических фигурах. После того, как была проверена корректность алгоритмов, было необходимо убедиться в оптимальности представленных реализаций. Именно поэтому был необходим следующий этап тестирования – экспериментирование с тонкостями реализации алгоритмов для получения максимальной производительности.

7.3 Тестирование производительности

Часть предложенных в рамках данного проекта алгоритмов не могут быть улучшены асимптотически, однако за счет подбора необходимых констант или глубин построения можно добиться прироста к скорости и снижения потребления памяти. Для этого необходимо было создать удобный способ отслеживать потребляемую память или затраченную на какую либо операцию времени. В ClickHouse существует набор инструментов для этого, например опция `--time` в запросах, отображающая реальное и процессорное время исполнения. Моей задачей было изучить существующие в ClickHouse инструкции для этого, дополнить необходимые, но отсутствующие параметры стандартным утилитами Unix, после чего воспользоваться ими в bash-скриптах, которые помимо запуска тестов отслеживают потребляемые в это время ресурсы. После этого несложно было выделить ряд наиболее ресурсозатратных мест, на которые и была направлена дальнейшая работа.

7.3.1 Проведение экспериментов

После подготовки инструментов для мониторинга, был проведен ряд экспериментов с различными версиями алгоритмов. Одним из самых первых улучшений был объем потребляемой памяти, который был улучшен в несколько раз после смены способа хранения полосок на дерево отрезков. После этого были рассмотрены несколько вариаций алгоритмов, которые как нам казалось могут улучшить результаты, однако замеры показали, что прироста производительности практически не было. Когда были попробованы и протестированы все оригинальные подходы, было принято решение сосредоточится на оптимизации тех, что показали себя наиболее хорошо. В данных подходах так же было несколько мест, с которыми имело смысл проводить эксперименты: изменять глубину сетки при построении, хранить сами объекты вместо индексов, пробовать совмещать наиболее удачные подходы – все это могло привести к улучшению, и большинство из данных идей улучшили как потребление памяти, так и

производительность.

Приведем результаты некоторых экспериментов с параметрами. Ниже приведена зависимость времени построения, работы (в секундах) и памяти (в гигабайтах) от параметров алгоритма MAX_DEPTH и MIN_INTERSECTION. В качестве данных для эксперимента выступали данные о границах субъектов России, количество запросов к словарю составляло 1 миллион.

Реализация POLYGON_INDEX_CELL				
MAX_DEPTH	MIN_INTER-SECTION	Время построения	Время работы	Потребление памяти
0	1	8.944	7.381	2.94
1	1	6.432	3.212	2.45
2	1	3.907	1.711	2.17
3	1	4.95	1.301	1.87
4	1	11.5	1.122	1.88
5	1	66	1.106	2.07
6	1	610	1.115	2.7
6	3	147	1.095	2.63

Реализация POLYGON_INDEX_EACH				
MAX_DEPTH	MIN_INTER-SECTION	Время построения	Время работы	Потребление памяти
2	1	7.55	15.2	2.2
3	1	8.423	3.134	2.19
4	1	12.8	1.7	2.18
5	1	48.3	1.324	2.15
6	3	75	1.372	2.1

На момент сбора этих данных уже использовалась оптимизация памяти с использованием дерева отрезков. Без данной оптимизации расход памяти был больше в 10 раз, и проведение экспериментов было нерелевантным.

7.3.2 Изменение детализации данных для ускорения поиска

Все данные разбиты на несколько административных уровней, из-за чего можно конфигурировать детализируемость набора данных для улучшения производительности. К примеру, если необходимо найти точку с точностью до города, нет никакой необходимости загружать данные районов, можно ограничиться лишь соответствующим административным уровнем. Разумеется, такой подход крайне увеличивает производительность как построения словаря, так и ответа на запросы.

8 Использование разработанного функционала

8.1 Пользовательская документация

Была написана пользовательская документация, освещающая основные аспекты словарей полигонов и реализованных в них алгоритмов. В ней приведены примеры использования нового функционала. Документация была написана на двух языках - русском и английском.

8.2 Пример использования

Приведем пример использования реализованных функций. Одна из возможных конфигураций словаря:

```
<dictionary>
    <structure>
        <key>
            <name>key</name>
            <type>Array(Array(Array(Array(Float64))))</type>
        </key>
        <attribute>
            <name>name</name>
            <type>String</type>
```

```
            <null_value></null_value>
        </attribute>
        <attribute>
            <name>value</name>
            <type>UInt64</type>
            <null_value>0</null_value>
        </attribute>
    </structure>
    <layout>
        <polygon />
    </layout>
</dictionary>
```

Можно задавать конфигурацию с помощью DDL-запросов, соответствующий примеру выше запрос:

```
CREATE DICTIONARY polygon_dict_name (
    key Array(Array(Array(Array(Float64)))),
    name String,
    value UInt64
)
PRIMARY KEY key
LAYOUT(POLYGON())
...
```

Все атрибуты являются опциональными, при создании словаря ключ должен иметь один из двух типов:

- Простой полигон. Представляет из себя массив точек.
- Мультиполигон. Представляет из себя массив полигонов. Каждый полигон задается двумерным массивом точек — первый элемент этого массива задает внешнюю границу полигона, последующие элементы могут задавать дырки, вырезаемые из него.

Точка может задаваться массивом или кортежем из своих координат. В данный момент поддерживаются только двумерные точки.

Пользователь может загружать свои собственные данные во всех поддерживаемых ClickHouse форматах.

В данном примере в качестве данных полигонов может выступать следующий файл формата JsonEachRow:

```
{"key" : [[[[0, 0], [1, 0], [0, 1]]]],
 "name" : "Moscow",
 "value" : "1"}
{"key" : [[[[0, 0], [2, 0], [0, 2]]]],
 "name" : "Russia",
 "value" : "2"}
...
```

В данный момент доступно 3 типа хранения данных в памяти:

- POLYGON, описанный в разделе 4.2
- POLYGON_EACH_INDEX, описанный в разделе 6.4
- POLYGON_INDEX_CELL, описанный в разделе 6.5

После инициализации, запросы к словарю осуществляются с помощью стандартных функций для работы со внешними словарями. Важным отличием является то, что здесь ключами будут являются точки, для которых хочется найти содержащий их полигон. Пример работы со словарем, определенным выше:

```
CREATE TABLE points (
    x Float64,
    y Float64
)
... //points initialization
SELECT 'dictGet', 'polygon_dict_name' AS dict_name, tuple(x, y) AS key,
dictGet(dict_name, 'name', key),
dictGet(dict_name, 'value', key)
FROM points ORDER BY x, y;
```

В результате исполнения последней команды для каждой точки в таблице points будет найден полигон минимальной площади, содержащий данную точку, и выведены запрошенные аттрибуты.

Пример вывода данной команды:

```
dictGet polygon_dict_name (1.1,0) Russia 2
dictGet polygon_dict_name (0,0.5) Moscow 1
...
```

Таким образом, полный пример использования словаря полигонов может выглядеть следующим образом:

```
CREATE DATABASE example Engine = Ordinary;
CREATE TABLE example.points
(
   x Float64,
   y Float64
) ENGINE = Memory;
//Filling points table

CREATE TABLE example.polygons_array
(
   key Array(Array(Array(Array(Float64)))),
   name String,
   value UInt64
) ENGINE = Memory;
//Filling polygon table

CREATE DICTIONARY example.dict_array
(
key Array(Array(Array(Array(Float64)))),
name String DEFAULT 'qqq',
value UInt64 DEFAULT 101
)
PRIMARY KEY key
SOURCE(CLICKHOUSE(HOST 'localhost'
```

```
                PORT 9000
                USER 'default'
                TABLE 'polygons_array'
                PASSWORD ''
                DB 'example')
    )
LIFETIME(MIN 1 MAX 10)
LAYOUT(POLYGON_INDEX_CELL());

SELECT 'dictGet', 'example.dict_array' AS dict_name, tuple(x, y) AS key,
dictGet(dict_name, 'value', key)
FROM example.points ORDER by x, y;
```

9 Заключение

В результате работы в ClickHouse были добавлены новые словари, поддерживающие географические координаты, которые могут быть использованы для решения других связанных с геолокацией задач. Также мы реализовали несколько алгоритмов, решающих поставленную задачу поиска полигона по данной точке. В реализованных алгоритмах представлены асимптотически оптимальные реализации, которые после этого были оптимизированы с помощью подбора оптимальных параметров, основанных на результатах проводимых бенчмарков. Были написаны тесты, проверяющие корректность и эффективность реализаций всех представленных подходов. На их основании мы можем сделать вывод, что ожидаемая производительность в создании словарей и ответах на запросы была достигнута. Все новые функции были описаны в документации ClickHouse.

В качестве перспектив дальнейшей работы можно выделить оптимизацию существующих алгоритмов, а также изучение и реализацию новых подходов.

Список литературы

1. ClickHouse. — URL: https://clickhouse.tech/ (дата обращения: 03.05.2020).

2. Обзор – Документация ClickHouse. — URL: https://clickhouse.tech/docs/ru/ (дата обращения: 10.02.20).

3. Функции для работы с внешними словарями – Документация ClickHouse. — URL: https://clickhouse.tech/docs/ru/query_language/functions/ext_dict_functions/ (дата обращения: 10.02.20).

4. Репозиторий ClickHouse. — URL: https://github.com/ClickHouse/ClickHouse (дата обращения: 27.05.2020).

5. Словари – Документация ClickHouse. — URL: https://clickhouse.tech/docs/ru/sql-reference/dictionaries/ (дата обращения: 15.05.20).

6. 15. Spatial Indexing — Introduction to PostGIS. — 2012. — URL: https://postgis.net/workshops/postgis-intro/indexing.html (дата обращения: 13.02.2020).

7. Finkel Raphael, Bentley Jon. Quad Trees: A Data Structure for Retrieval on Composite Keys. // Acta Inf. — 1974. — 03. — Т. 4. — С. 1–9.

8. Guttman Antonin. R-Trees: A Dynamic Index Structure for Spatial Searching // Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. — SIGMOD ’84. — New York, NY, USA : Association for Computing Machinery, 1984. — С. 47–57. — Режим доступа: https://doi.org/10.1145/602259.602266.

9. Bentley Jon Louis. Multidimensional Binary Search Trees Used for Associative Searching // Commun. ACM. — 1975. — Сент. — Т. 18, № 9. — С. 509–517. — Режим доступа: https://doi.org/10.1145/361002.361007.

10. polygon-lookup. — 2019. — URL: https://github.com/pelias/polygon-lookup (дата обращения: 13.02.2020).

11. Loecher Markus, Kumar Madhav. RapidPolygonLookup : An R package for polygon lookup using kd trees. — 2014.

12. Graphics Gems IV / Под ред. Paul S. Heckbert. — USA : Academic Press Professional, Inc., 1994. — C. 24–46. — ISBN: 0123361559.

13. Функции для работы с географическими координатами – Документация ClickHouse. — URL: https://clickhouse.tech/docs/ru/sql-reference/functions/geo/#pointinpolygon (дата обращения: 03.05.2020).

14. Sarnak Neil, Tarjan Robert E. Planar Point Location Using Persistent Search Trees // Commun. ACM. — 1986. — Июль. — Т. 29, № 7. — С. 669–679. — Режим доступа: https://doi.org/10.1145/6138.6151.

15. Vertica. STV_Intersect Scalar Function. — URL: https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/SQLReferenceManual/Functions/Geospatial/STV_IntersectScalarFunction.htm (дата обращения: 03.05.2020).

16. Внешние словари – Документация ClickHouse. — URL: https://clickhouse.tech/docs/ru/sql-reference/dictionaries/external-dictionaries/external-dicts/ (дата обращения: 10.12.19).

17. Chapter 1. Geometry - 1.73.0. — URL: https://www.boost.org/doc/libs/1_73_0/libs/geometry/doc/html/index.html (дата обращения: 15.05.20).

18. 9. Geometries – Introduction to PostGIS. — URL: http://postgis.net/workshops/postgis-intro/geometries.html#collections (дата обращения: 27.05.2020).

19. RFC 7496 – The GeoJSON Format. — URL: https://tools.ietf.org/html/rfc7946#section-3.1.7 (дата обращения: 27.05.2020).

20. OpenStreetMap. — URL: https://www.openstreetmap.org/ (дата обращения: 03.05.2020).

21. GeoJson. — URL: https://geojson.org/ (дата обращения: 03.05.2020).