Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Метод построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей»

Студент: ИУ7-83Б _____ М. Д. Маслова
(группа) (подпись, дата) (И. О. Фамилия)

Руководитель: _____ А. А. Оленев
(подпись, дата) (И. О. Фамилия)

Нормоконтролер: _____ Д. Ю. Мальцева
(подпись, дата) (И. О. Фамилия)

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка 79 с., 36 рис., 1 табл., 32 источн., 3 прил. ИНДЕКСЫ, ОБУЧЕННЫЕ ИНДЕКСЫ, ГЛУБОКИЕ НЕЙРОННЫЕ СЕТИ, РЕЛЯЦИОННАЯ БАЗА ДАННЫХ

В данной работе представлена разработка метода построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей.

В разделе 1 представлено описание построения индексов на основе базовых структур и применение методов машинного обучения к построению индексов. Проведено сравнение индексов на основе B-деревьев, хеш-таблиц и битовых карт с индексами на основе моделей машинного обучения. Описаны особенности индексов в реляционных базах данных и применение нейронных сетей к построению индексов.

В разделе 2 спроектирован метод построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей. Представлены схемы алгоритмов этапов метода и случаев его применения: поиска и вставки.

В разделе 3 спроектировано, разработано и протестировано программное обеспечение, реализующее метод.

В разделе 4 проведено исследование времени выполнения операций построения, поиска и вставки от количества ключей с использованием индекса, построенного разработанным методом, и классического индекса.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На протяжении последнего десятилетия происходит автоматизация все большего числа сфер человеческой деятельности [1], что приводит к росту числа данных. Так, по исследованию компании IDC (International Data Corporation), изучающей мировой рынок информационных технологий и тенденций развития технологий, объем данных к 2025 году составит около 175 зеттабайт, в то время как на год исследования их объем составлял 33 зеттабайта [2].

Для хранения накопленных данных используются базы данных (БД), доступ к ним обеспечивается системами управления базами данных (СУБД), обрабатывающими запросы на поиск, вставку, удаление или обновление. При больших объемах информации необходимы методы для уменьшения времени обработки запросов, одним из которых является построение индексов [3].

Базовые методы построения индексов основаны на таких структурах, как деревья поиска, хеш-таблицы и битовые карты [4]. Однако с ростом объема данных требуется модификация существующих или разработка новых структур для уменьшения времени поиска и затрат на перестроение индекса при изменении данных, а также сокращения дополнительно используемой памяти. Одним из решений являются обученные индексы (*learned indexes*) [5], которые представляют совокупность способов построения, основанных на использовании различных моделей машинного обучения от линейной регрессии до нейронных сетей, позволяющих учитывать, в отличие от индексов на базовых структурах, распределение индексируемых данных. На этой идее строится данная работа.

Целью данной работы является разработка метода построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей.

Для достижения поставленной цели требуется решить следующие задачи:

- рассмотреть и сравнить известные методы построения индексов;
- привести описание построения индексов с помощью нейронных сетей;
- разработать метод построения индексов в реляционной базе данных на основе глубоких нейронных сетей;
- разработать программное обеспечение, реализующее данный метод;
- провести исследование (по времени и памяти) операций поиска и вставки с использованием индекса, построенного разработанным методом, при различных объемах данных.

1 Аналитическая часть

1.1 Основные определения

Индекс — это некоторая структура, обеспечивающая быстрый поиск записей в базе данных [6]. Индекс определяет соответствие значения полей или набора полей — ключа поиска — конкретной записи с местоположением этой записи [7]. Это соответствие организуется с помощью индексных записей. Каждая из них соответствует записи в индексируемой наборе данных — наборе, по которому строится индекс — и содержит два поля: идентификатор записи или указатель на нее, а также значение индексированного поля в этой записи [8].

Индексы могут использоваться для поиска по конкретному значению или диапазону значений, а также для проверки существования элемента в наборе, однако обеспечение уменьшения времени доступа к записям в общем случае достигается за счет [7]:

— упорядочивания индексных записей по ключу поиска, что уменьшает количество записей, которые необходимо просмотреть;
— а также меньшего размера индекса по сравнению с индексируемой таблицей, сокращающего время чтения одного элемента.

В то же время индекс является структурой, которая строится в дополнение к существующим данным, то есть он занимает дополнительный объем памяти и должен соответствовать текущим данным. Последнее значит, что индекс необходимо изменять при вставке или удалении элементов, на что затрачивается время, поэтому индекс, ускоряя работу СУБД при доступе к данным, замедляет операции изменения исходного набора данных, что необходимо учитывать[9].

Таким образом, индекс может описываться [7]:

— типом доступа — поиск записей по атрибуту с конкретным значением, или со значением из указанного диапазона;
— временем доступа — время поиска записи или записей;
— временем вставки, включающее время поиска правильного места вставки, а также время для обновления индекса;
— временем удаления, аналогично вставке, включающее время на поиск удаляемого элемента и время для обновления индекса;
— дополнительной памятью, занимаемой индексной структурой.

1.2 Индексы в реляционных базах данных

Построение индексов для ускорения поиска данных используется в базах данных с различными моделями представления хранимой информации: в реляционных, документо- и объектно-ориентированных, в базах данных «ключ-значение» и др. По исследованию 2023 года [10] реляционные базы данных являются первыми по популярности, охватывая 72.7% рынка против 10.1% документоориентированных и 5.4% баз данных типа «ключ-значение», занимающих второе и третьи места соответственно. Таким образом, совершенствование индексов для реляционных баз данных является актуальной задачей.

Реляционные базы данных [11] основаны на реляционной модели, представляющей сущности и связи между ними в виде отношений — двумерных таблиц, каждая строка (кортеж) которых является записью, содержащей данные о конкретном объекте данной сущности, а столбцы — ее свойствами, называющиеся атрибутами. То есть реляционная база данных представляет собой совокупность связанных между собой отношений, каждое из которых содержит информацию о соответствующей сущности в структурированном виде.

Особенностями построения индексов в реляционных базах данных является то, что каждый индекс строится по определенной таблице, при этом каждая индексная запись содержит указатель на определенную строку таблицы, а также ключ поиска, в качестве которого могут быть выбраны:

- первичные ключи, однозначно идентифицирующие запись в таблице, для ускорения доступа к конкретным записям;
- внешние ключи, обеспечивающие связи между отношениями, для быстрого доступа к связанным записям в разных таблицах;
- другие атрибуты, по которым наиболее часто поступают запросы.

Остальные вышеописанные свойства индексов верны и в случае их применения к реляционным базам данных, так при построении нескольких индексов следует помнить о необходимости их изменения при обновлении соответствующей таблицы, иначе при неконтролируемом создании многих индексов может произойти потеря скорости работы с данными, а не ее увеличение.

1.3 Типы индексов

В общем случае, в том числе и в реляционных базах данных, индексы делятся на [7]:

— кластеризованные и некластеризованные;
— плотные и разреженные;
— одноуровневые и многоуровневые;
— а также иметь в своей основе различные структуры, что описывается в следующем разделе, так как исследуется в данной работе.

В кластеризованных индексах логический порядок ключей определяет физическое расположение записей, а так как строки в таблице могут быть упорядочены только в одном порядке, то кластеризованный индекс может быть только один на таблицу. Логический порядок некластеризованных индексов не влияет на физический, и индекс содержит указатели на записи таблицы [9].

Плотные индексы (рисунок 1.1) содержат ключ поиска и указатель на первую запись с заданным ключом поиска. При этом в кластеризованных индексах другие записи с заданным ключом будут лежать сразу после первой записи, так как записи в таких файлах отсортированы по тому же ключу. Плотные некластеризованные индексы должны содержать список указателей на каждую запись с заданным ключом поиска [7].

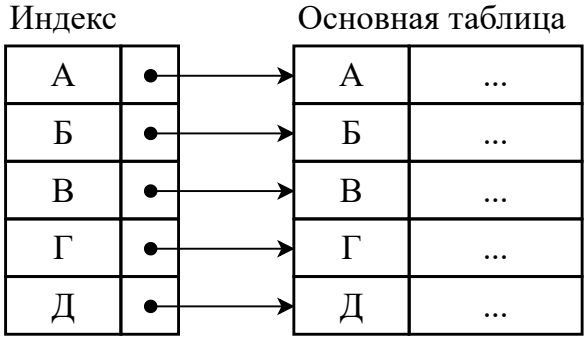

Рисунок 1.1 – Плотный индекс

В разреженных индексах (рисунок 1.2) записи содержат только некоторые значения ключа поиска, а для доступа к элементу отношения ищется запись индекса с наибольшим меньшим или равным значением ключа поиска, происходит переход по указателю на первую запись по найденному ключу и далее по указателям в файле происходит поиск заданной записи. Таким образом,

разреженные индексы могут быть построены только на отсортированных последовательностях записей, иначе хранения только некоторых ключей поиска будет недостаточно, так как будет неизвестно, после записи, с каким ключом будет лежать необходимый элемент отношения [7].

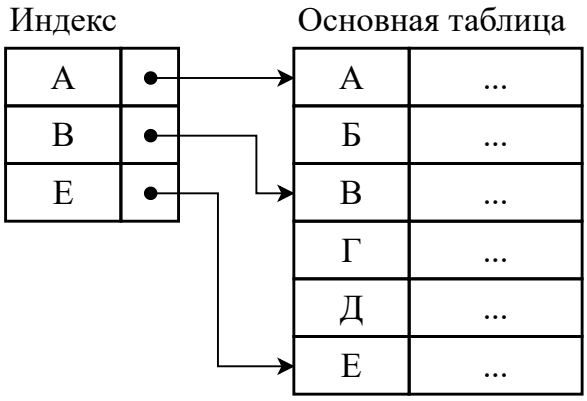

Рисунок 1.2 – Разреженный индекс

Поиск с помощью плотных индексов быстрее, так как указатель в записи индекса сразу приводит к необходимым записям. Однако разреженные индексы требуют меньше дополнительной памяти и сокращают время поддержания структуры индекса в актуальном состоянии при вставке или удалении [7].

Одноуровневые индексы ссылаются на данные в таблице, индексы же верхнего уровня многоуровневой структуры ссылаются на индексы нижестоящего уровня [7] (рисунок 1.3).

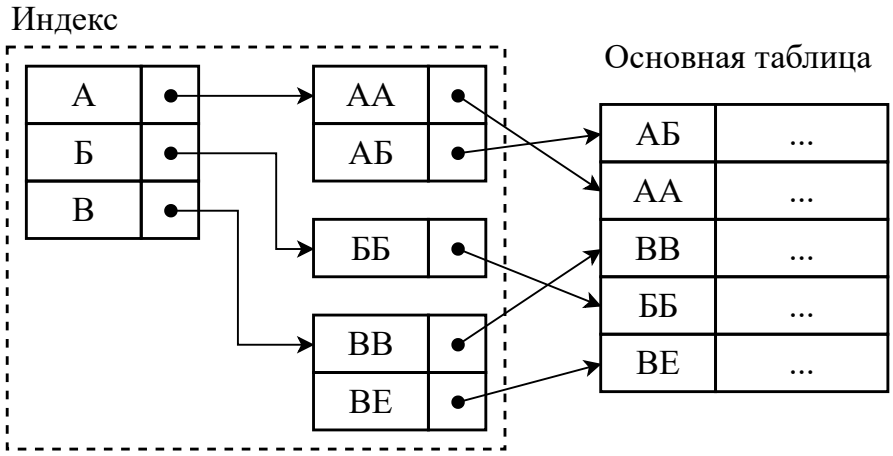

Рисунок 1.3 – Многоуровневый индекс

1.4 Построение индексов на основе базовых структур

Как было сказано выше индексы обеспечивают быстрый поиск записей, поэтому в их основе лежат структуры, предназначенные для решения этой задачи. По данным структурам индексы подразделяются на

— индексы на основе деревьев поиска,
— индексы на основе хеш-таблиц,
— индексы на основе битовых карт.

При этом каждой из представленных групп индексов соответствует своя подзадача, которую решают данные индексы, что подробнее описано далее.

1.4.1 Индексы на основе деревьев поиска

Дерево поиска — иерархическая структура, используемая для поиска записей, которая осуществляет работу с отсортированными значениями ключей и в которой каждый переход на более низкий уровень иерархии уменьшает интервал поиска. При использовании деревьев поиска для построения индексов необходимо учитывать, что требуется обеспечить как ускорение поиска данных, так и уменьшение затрат на обновление индекса при вставках и удалениях. По этим причинам при решении задачи поиска в базах данных используют сбалансированные сильноветвящиеся деревья [12].

В данном случае сбалансированными деревьями называют такие деревья, что длины любых двух путей от корня до листьев одинаковы [13]. Сильноветвящимися же являются деревья, каждый узел которых ссылается на большое число потомков [14]. Эти условия обеспечивают минимальную высоту дерева для быстрого поиска и свободное пространство в узлах для внесения изменений в базу данных без необходимости изменения индекса при каждой операции.

Наиболее используемыми деревьями поиска, имеющими описанные свойства, являются B-деревья и их разновидность — B^+-деревья [12].

1.4.1.1 B-деревья

B-дерево — это сбалансированная, сильноветвящаяся древовидная, работающая с отсортированными значениями структура данных, операции вставки и удаления в которой не изменяют ее свойств [15]. Все свойства данной структуры поддерживаются путем сохранения в узлах положений для включения

новых элементов [16]. Это осуществляется за счет свойств узлов, которые определяются порядком B-дерева m.

B-деревом порядка m [12, 16] называется дерево поиска, такое что:

— каждый узел имеет формат, описывающийся формулой (1.1):

$$(P_1, (K_1, Pr_1), P_2, (K_2, Pr_2), \ldots, (K_{q-1}, Pr_{q-1}), P_q), \tag{1.1}$$

где $q \leqslant m$,

P_i — указатель на i-ого потомка в случае внутреннего узла или пустой указатель в случае внешнего (листа),

K_i — ключи поиска,

Pr_i — указатель на запись, соответствующую ключу поиска K_i;

— для каждого узла выполняется $K_1 < K_2 < \cdots < K_q$;

— для каждого ключа поиска X потомка, лежащего по указателю P_i выполняются условия, описывающиеся формулами (1.2)-(1.4):

$$K_{i-1} < X < K_i, \text{ если } 1 < i < q, \tag{1.2}$$

$$X < K_i, \text{ если } i = 1, \tag{1.3}$$

$$K_{i-1} < X, \text{ если } i = q; \tag{1.4}$$

— каждый узел содержит не более $m - 1$ ключей поиска или, что то же самое, имеет не более m потомков;

— каждый узел за исключением корня содержит не менее $\lceil m/2 \rceil - 1$ ключей поиска, или, что то же самое, имеет не менее $\lceil m/2 \rceil$ потомков;

— корень может содержать минимум один ключ, либо, что то же самое, иметь минимум два потомка;

— каждый узел за исключением листьев, содержащий $q-1$ ключей, имеет q потомков;

— все листья находятся на одном и том же уровне.

В случае с индексами к каждому ключу поиска во всех узлах добавляется указатель на запись, соответствующую этому ключу. Другими словами, каждый узел содержит набор указателей, ссылающихся на дочерние узлы, и набор пар, каждая из которых состоит из ключа поиска и указателя, ссылающегося на данные. При этом записи с данными хранятся отдельно и частью B-дерева не являются [12].

Пример B-дерева представлен на рисунке 1.4.

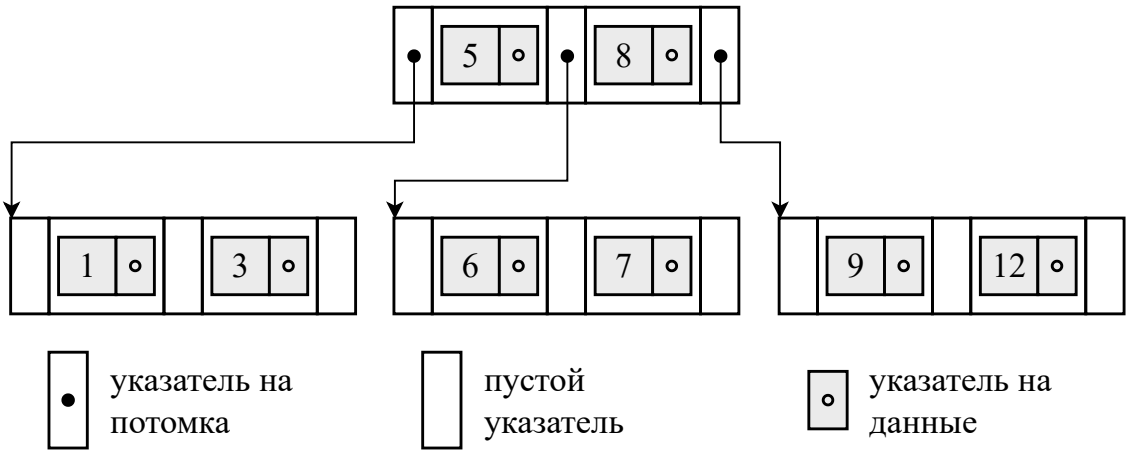

Рисунок 1.4 – Пример B-дерева

Построение B-дерева [17] начинается с создания корневого узла. В него происходит вставка до полного заполнения, то есть до того момента, пока все $q - 1$ позиций не будут заняты. При вставке q-ого значения создается новый корень, в который переносится только медиана значений, старый корень разделяется на два узла, между которыми равномерно распределяются оставшиеся значения (рисунок 1.5). Два созданных узла становятся потомками нового корня.

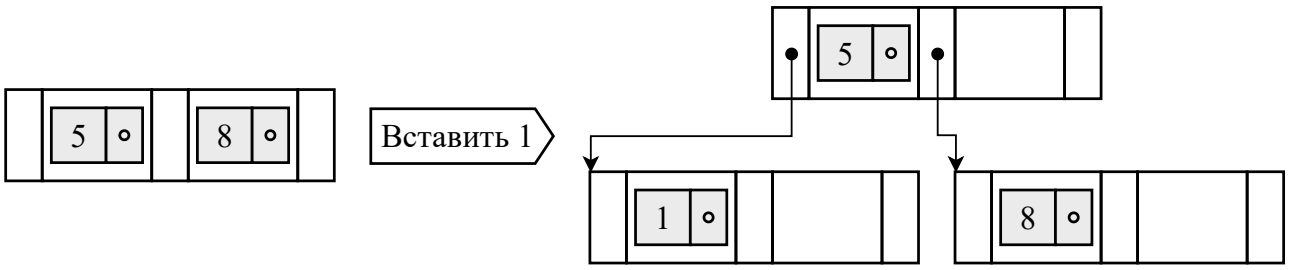

Рисунок 1.5 – Пример вставки в B-дерево при заполненном корне

Когда некорневой узел заполнен и в него должен быть вставлен новый ключ, этот узел разделяется на два узла на том же уровне, а средняя запись перемещается в родительский узел вместе с двумя указателями на новые разделенные узлы. Если родительский узел заполнен, он также разделяется. Разделение может распространяться вплоть до корневого узла, при разделении которого создается новый уровень (рисунок 1.6). Фактически дерево строится последовательным выполнением операций вставки.

Удаление значений основано на той же идее. Нужное значение удаляется из узла, в котором оно находится, и если количество значений в узле становится

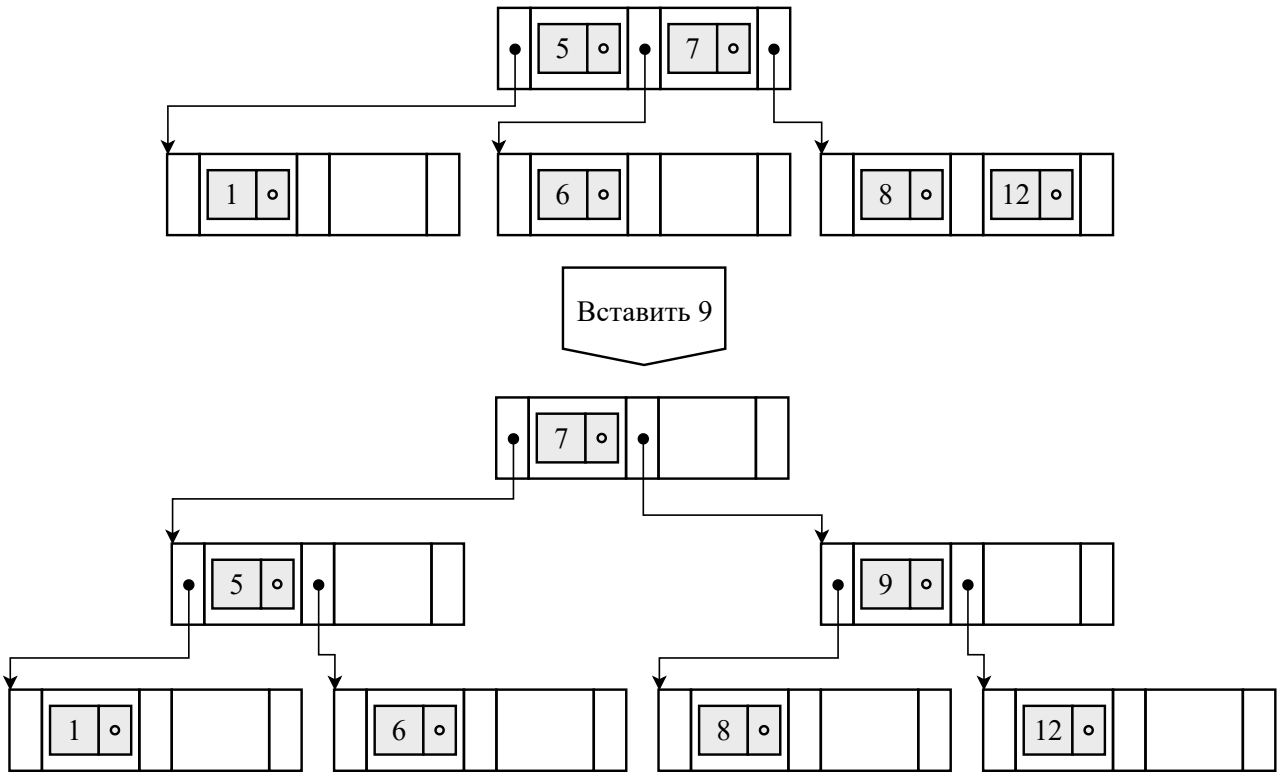

Рисунок 1.6 – Пример вставки в B-дерево при последовательном заполнении узлов разных уровней

меньше половины максимально возможного количества значений, то узел объединяется с соседними узлами, что также может распространяться вплоть до корня [18].

Поиск в B-дереве начинается с корня. Если искомое ключевое значение X найдено в узле, то есть какой либо ключ K_i в нем равен X, то доступ к нужной записи осуществляется по соответствующему указателю Pr_i. Если значение не найдено, происходит переход к поддереву по указателю P_i, соответствующему наименьшему значению i, такому, что $X < K_i$. Если X больше K_i для любого значения $i \in \overline{1, q-1}$, то переход осуществляется по указателю P_q. Далее действия повторяются для того поддерева, к которому произошел переход, до тех пор, пока не будет найдено нужное значение или не будет достигнут конец листового узла, что означает отсутствие искомого ключа [12].

Таким образом, можно выделить следующие основные свойства B-деревьев [18]:

— ключи и указатели на данные хранятся во всех узлах дерева;
— как следствие первого свойства, поиск различных ключей может выполняться проходом по разному числу узлов, но максимальная длина

пути равна высоте дереве, что дает временную сложность поиска в среднем случае — $O(\log N)$;

— операции вставки и удаления также имеют временную сложность в среднем случае — $O(\log N)$;

— в случае поиска по ключам, принадлежащим некоторому диапазону, требуется переход от дочерних узлов к родительским и наоборот, что является главным недостатком B-дерева.

1.4.1.2 B$^+$-деревья

Для устранения недостатков B-деревьев были введены B$^+$-деревья [18], структура которых аналогична структуре B-дерева за исключением двух моментов [12]. Во-первых, внутренние узлы не содержат указателей на записи, в них хранятся только значения ключей, то есть внутренние узлы имеют формат, описывающийся формулой (1.6):

$$(P_1, K_1, P_2, K_2, \ldots, K_{q-1}, P_q), \tag{1.5}$$

где $q \leqslant m$,

P_i — указатель на i-ого потомка,

K_i — ключи поиска.

Указатели на данные содержатся только в листьях. При этом каждый ключ, содержащийся во внутренних узлах, встречается в каком-либо листе, то есть условие, представленное формулами (1.2)-(1.4), для B$^+$-деревьев модернизируется в формулы (1.6)-(1.8):

$$K_{i-1} < X \leqslant K_i, \text{ если } 1 < i < q, \tag{1.6}$$

$$X \leqslant K_i, \text{ если } i = 1, \tag{1.7}$$

$$K_{i-1} < X, \text{ если } i = q. \tag{1.8}$$

Все остальные свойства B-дерева порядка m верны и для B$^+$-дерева.

Во-вторых, каждый листовой узел содержит только пары (ключ, указатель на данные) и не содержит указателей на потомков, так как при любых операциях лист не может стать внутренним узлом, а также структура внешнего узла отличается от структуры внутренних. При этом в конец каждого листа добавляется указатель на следующий лист.

Пример B^+-дерева приведен на рисунке 1.7.

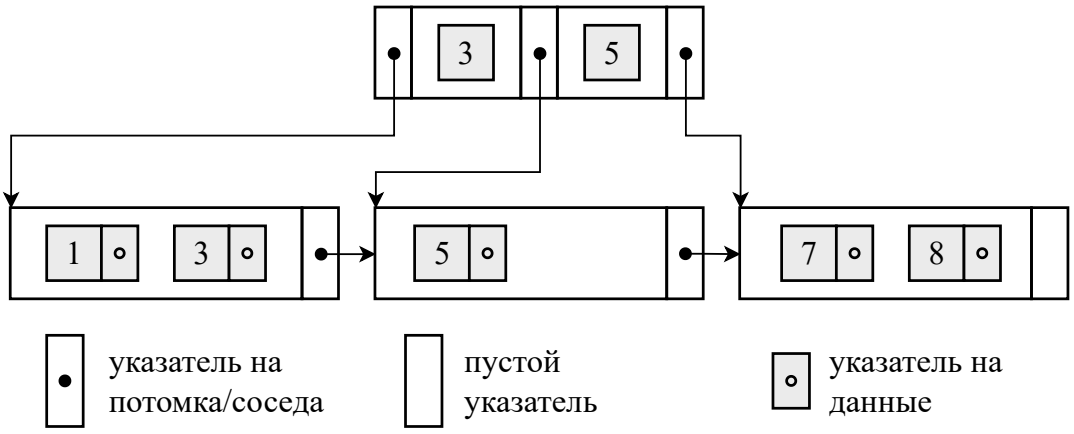

Рисунок 1.7 – Пример B^+-дерева

В силу того, что листы имеют структуру, они могут иметь порядок отличный от порядка внутренних узлов, что позволяет уменьшить высоту дерева, а следовательно и количество блоков памяти, к которым необходимо обратиться, что позволяет сократить время поиска. Наличие же во внешних узлах всех ключей и указателей на соседние листы, предоставляет новый способ обхода дерева — последовательно по листам, что дает возможность быстрее обрабатывать запросы на поиск в диапазоне [16]. Операции вставки и удаления элементов в B^+-дерево аналогичны соответствующим операциям на B-дереве, то есть сложности всех операций над B^+-деревом остаются такими же как в B-дереве. Из-за большей скорости поиска по сравнению с B-деревьями и аналогичных операций B^+-деревья часто называют просто B-деревьями, подменяя исходный термин.

1.4.2 Индексы на основе хеш-таблиц

Альтернативным способом построения индексов является хеширование. Идея этого подхода заключается в применении к значению ключа поиска некоторой функции свертки, называющейся хеш-функцией, по определенному алгоритму вырабатывающей значение, определяющее адрес в таблице, содержащей ключи и записи или указатели на записи, называющейся хеш-таблицей [9]. Следует учитывать, что разные ключи могут быть преобразованы хеш-функцией в одно и то же значение. Такая ситуация называется коллизией и дожна быть каким-либо способом разрешена.

К хеш-функциям предъявляется ряд требований [16, 18]:

— значения, получаемые в результате применения хеш-функции к ключу должны принадлежать диапазону значений, определяющему действительные адреса в хеш-таблице;
— значения хеш-функции должны быть равномерно распределены для уменьшения числа коллизий;
— хеш-функция должна при одном и том же входном значении выдавать одно и то же выходное.

Для построения индекса на основе хеш-таблиц выбирается единица хранения, именуемая бакетом (англ. *bucket* — корзина) [18] или хеш-разделом [6], которая может содержать одну или несколько индексных записей, при этом их количество фиксировано [7].

Первоначально создается некоторое количество бакетов, которые и составляют хеш-таблицу. Хеш-функция, получая на вход ключ, отображает его в номер хеш-раздела в таблице. В случае, если раздел не заполнен, запись, состоящая из ключа и указателя на данные, помещается в него. Если же в разделе нет места для вставки новой записи, то есть возникает коллизия и необходимо найти новое место для вставки [18]. Процесс поиска такого места называется разрешением коллизии и может выполняться [16]:

— методом открытой адресации, при котором ищется первая свободная позиция в последующих незаполненных хеш-разделах;
— методом цепочек переполнения, заключающийся в создании хеш-разделы переполнения, к каждому из которых, включая раздел в хеш-таблице, добавляется указатель на следующих раздел, что создает связный список, относящийся к одному значению хеш-функции;
— методом двойного хеширования, при получении коллизии в результате применении первой хеш-функции используется вторая и, возможно, третья, а также метод открытой адресации при повторной коллизии.

Операции поиска, вставки и удаления в хеш-индексе зависят от используемого метода разрешения коллизий. Простейшим в этом плане и рассматривающимся далее при описании обученных хеш-индексов является метод цепочек переполнения, в котором поиск, вставка и удаление являются операциями над связным списком [16].

Пример хеш-индекса с разрешением коллизий по методу цепочек переполнения, приведен на рисунке 1.8.

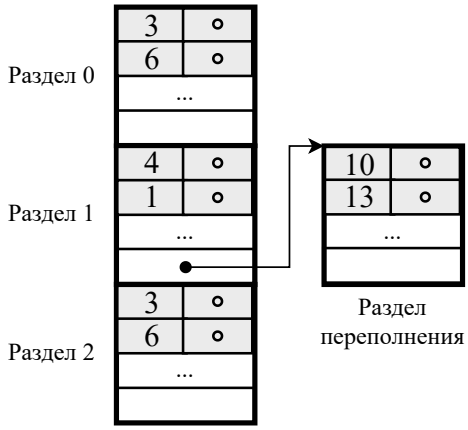

Рисунок 1.8 – Пример структуры хеш-индекса с разрешением коллизий методом цепочек переполнения

Хеш-индексы обеспечивают временную сложность каждой операции в среднем и лучшем случае — $O(1)$, в худшем случае — $O(N)$ [14]. При этом хеш-индексы, в отличие индексов, на основе деревьев поиска, используются только для поиска единичных ключей и не предназначены для поиска диапазонов.

Существование коллизий подчеркивает потенциальную проблему использования хеш-индексов [14]. Так при заполнении хеш-таблицы более чем на 70% возникающие коллизии увеличивают время поиска в 1.5-2.5 раза при любом из методов разрешения коллизий [16].

Для предотвращения появления коллизий используют также методы динамического хеширования, при котором при вставках и удалениях размер исходной хеш-таблицы увеличивается или уменьшается соответственно. К таким методам относят:

— расширяемое хеширование (*extendible hashing*) [13], представляющее значение хеш-функции как битовую строку и использующее из нее количество бит, необходимое для однозначной идентификации текущего количества записей в таблице;
— и линейное хеширование (*linear hashing*) [13], при необходимости изменяющее размер таблицы на один хеш-раздел.

Однако они также имеют недостатки. При расширяемом хешировании из-за увеличения числа учитывающихся разрядов в битовой строке размер таблицы каждый раз увеличивается в два раза, что может не оправдаться в случае, если дальнейших вставок в таблицу не произойдет. При этом линейное хеширование не исключает создание разделов переполнения [13].

1.4.3 Индексы на основе битовых карт

Индексы на основе битовых карт хранят данные в виде битовых массивов. Обход индекса осуществляется путем выполнения побитовых логических операций над битовыми картами [15]. Данные индексы используются, когда атрибут имеет небольшое количество значений, так как, чем больше записей соответствуют значению одной и той же битовой карте, тем меньше их требуется, тем меньше размер индекса [18]. За счет этого свойства данные индексы могут использоваться для проверки существования записи с заданным ключом в наборе данных.

Одним из индексов, использующимся для проверки существования записи, является индекс на основе фильтра Блума.

Фильтр Блума использует массив бит размером m и k хеш-функций, каждая из которых сопоставляет ключ с одну из m позиций. Для добавления элемента в множество существующих значений ключ подается на вход каждой хеш-функции, возвращающих позицию бита, который должен быть установлен в единицу. Для проверки принадлежности ключа множеству, ключ также подается на вход k хеш-функций. Если какой-либо бит, соответствующий одной из возращенных позиций, равен нулю, то ключ не входит во множество. Из этого следует, что данный алгоритм гарантирует отсутствие ложноотрицательных результатов, то есть, если по результату работы алгоритма ключ не существует в исходном наборе данных, то он на самом деле отсутствует, если же по результату работы алгоритма ключ существует, то он может как и принадлежать множеству ключей исходного набора, так и не принадлежать ему. Временная сложность поиска для индекса на основе фильтра Блума — $O(k)$, где k — количество используемых хеш-функций [7].

Пример построения индекса на основе фильтра Блума приведен на рисунке 1.9, где h_1, h_2, h_3 — хеш-функции.

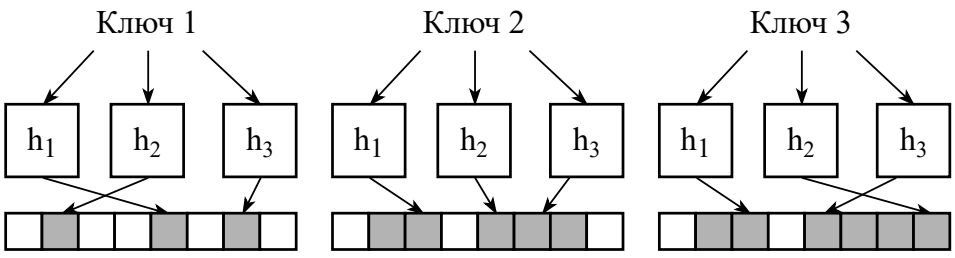

Рисунок 1.9 – Пример построения индекса на основе фильтра Блума

1.5 Применение методов машинного обучения к построению индексов

1.5.1 Обученные индексы поиска в диапазоне

Индексы на основе B-деревьев можно рассматривать как модель сопоставления ключа с позицией искомой записи в отсортированном массиве, или в терминах машинного обучения, как дерево принятия решения. Такие индексы сопоставляют ключ положению записи с минимальной ошибкой, равной нулю, и максимальной ошибкой, равной размеру страницы, гарантируя, что искомое значение принадлежит указанному диапазону. Поэтому B-дерево может быть заменено на какую-либо модель машинного обучения, включая нейронные сети, при условии, что эта модель будет также гарантировать принадлежность записи некоторому диапазону (рисунок 1.10). Такие индексы, в которых предсказывается положения ключа или диапазона ключей с помощью модели машинного обучения, называются обученными (*learned index*).

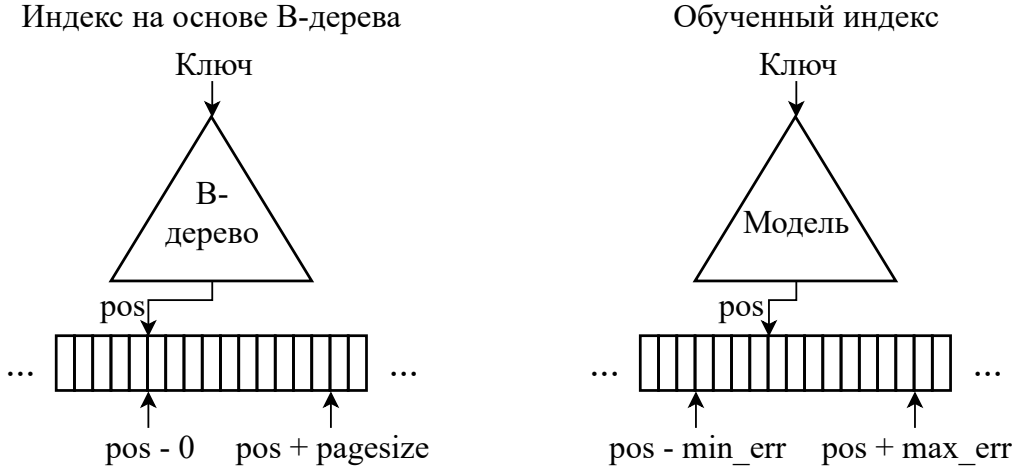

Рисунок 1.10 – Сравнение индексов на основе B-деревьев и обученных индексов

Возможность применения моделей машинного обучения опеспечивается тем, что они, предсказывая положение заданного ключа внутри отсортированного массива, аппроксимируют функцию распределения [5], что позволяет находить искомое положение ключа с помощью формулы (1.9):

$$p = F(K) \cdot N, \tag{1.9}$$

где p — искомая позиция;

K — ключ поиска;

$F(K)$ — функция распределения, дающая оценку вероятности обнаружения ключа, меньшего или равного ключу поиска K, то есть $P(X < K)$;

N — количество ключей.

То есть поиск положения ключа с помощью обученной модели представляет собой вычисление значения некоторой функции, что на первый взгляд может обеспечить достижение временной сложности поиска $O(1)$. Однако следует учитывать, что предсказание происходит с некоторой ошибкой, из-за чего возникает необходимость решения, так называемой, задачи «последней мили», заключающейся в уточнении найденной позиции ключа [5].

Для уточнения могут использоваться различные модели, например, линейные регрессии при линейности отдельных участков распределения, или B-деревья. Также для различных промежутков ключей могут быть использованы разные модели. В таком случае стоится используется рекурсивная модель индекса (рисунок 1.11), в которой строится иерархия моделей из n уровней. Каждая модель на вход получает ключ, на основе которого выбирает модель на следующем уровне, а модели последнего этапа предсказывают положение записи.

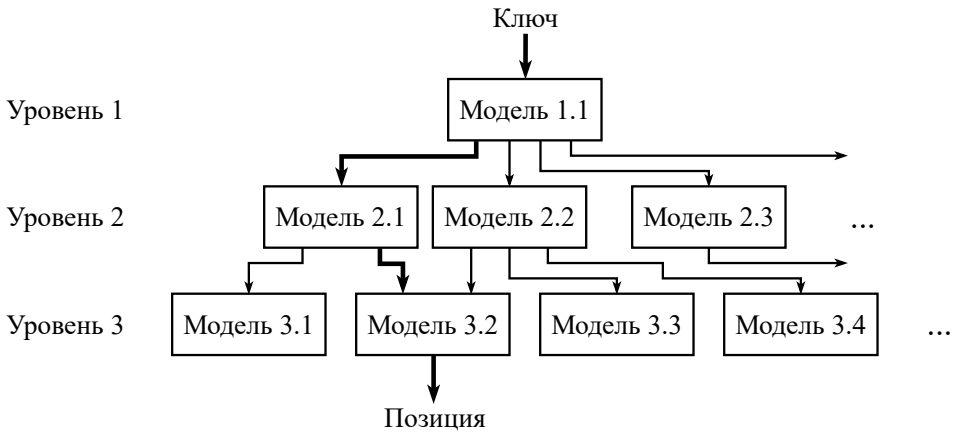

Рисунок 1.11 – Рекурсивная модель индекса

Таким образом, в лучшем случае для произвольного распределения временная сложность поиска обученных индексов будет составлять $O(n)$. Однако по сравнению с B-деревьями обученные индексы позволяют сократить число узлов в иерархии, что дает меньшее время поиска, а также уменьшить число дополнительных указателей для поддержания связей различных частей структуры, что позволяет сократить занимаемую индексом память. Ее сокращение в

том числе происходит и за счет постоянного не зависящего от числа ключей размера моделей машинного обучения на верхних уровнях.

Недостатком такого подхода является невозможность осуществления операций вставки и удаления без переобучения модели [19]. Однако дальнейшие исследования [19—22] решили эту проблему.

1.5.2 Обученные хеш-индексы

Традиционные хеш-индексы, описанные выше, могут быть рассмотрены как модели сопоставления ключа позиции искомой записи в неупорядоченном массиве, а следовательно могут быть заменены моделями машинного обучения. Обученные хеш-индексы [5] основаны на предположении, что модели машинного обучения, учитывающие распределение ключей, могут без увеличения размеров хеш-таблицы уменьшить количество коллизий. Для этого функция распределения ключей F масштабируется на размер хеш-таблицы M, а в качестве хеш-функции используется выражение, описывающееся формулой (1.10):

$$h(K) = F(K) \cdot M, \tag{1.10}$$

где K — ключ.

Таким образом, обученный хеш-индекс учитывает эмпирическое распределение ключей, что позволяет уменьшать количество коллизий по сравнению с обычными хеш-таблицами (рисунок 1.12).

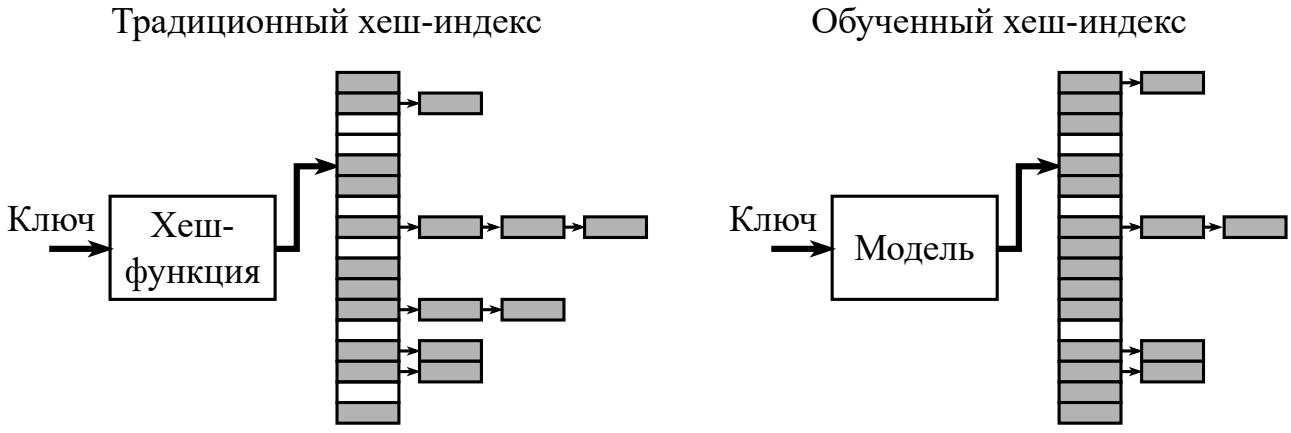

Рисунок 1.12 – Сравнение традиционных и обученных хеш-индексов

Так как обученный индекс уменьшает, но не предотвращает появление коллизий, его временные сложности совпадают со сложностями традиционного хеш-индекса.

1.5.3 Обученные индексы проверки существования

Обученные индексы для проверки наличия ключа в наборе данных преследуют цель уменьшения размера индекса, для этого в результате обучения должна получаться такая функция, которая относит ключи к одному как можно меньшему набору бит, а не ключи — к другому, не пересекающемуся с первым (рисунок 1.13). То есть, если провести аналогию с индексами на основе хеш-таблиц, где требуется сокращение числа коллизий, то в данному случае требуется идеальной будет функция, которая дает коллизии для каждого ключа с каждым, для каждого не ключа с каждым не ключом, и не дает ни одной коллизии какого-либо ключа с каким-либо не ключом [5].

Основным отличием дающим преимущество обученным индексам в данном случае является то, что для традиционного фильтра Блума число ложноотрицательных результатов, равное нулю, и число ложноположительных результатов, равное константе, выбираются априори, а для обученных индексов проверки существования для нулевого числа ложноотрицательных результатов, достигается заданное число ложноположительных результатов на реальных запросах. При этом для поиска достигается сложность $O(1)$, так как происходит только вычисление значения функции [5].

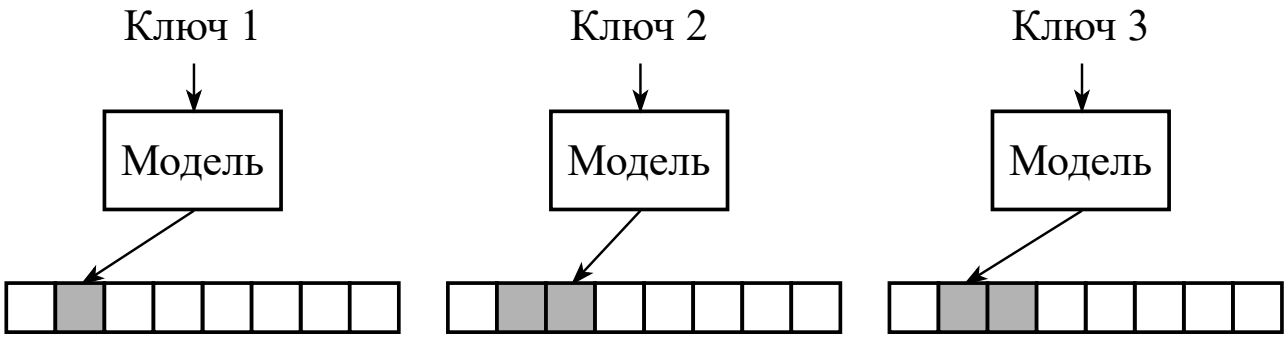

Рисунок 1.13 – Пример построения обученного индекса проверки существования

1.6 Сравнение описанных методов

Выше были описаны структуры данных, которым соответствуют методы построения индексов. Основными характеристиками, определяющими эффективность данных методов, являются время поиска и вставки, память, занимаемая структурой и возможность выполнения подзадач. На основе этого были выделены следующие критерии для оценки качества описанных методов:

— временная сложность поиска;
— временная сложность вставки;
— память под структуру;
— возможность поиска в диапазоне;
— возможность поиска единичных ключей;
— возможность проверки существования.

В таблице 1.1 приведены результаты сравнения рассмотренных методов. Обученные индексы в таблице представлены единым методом в силу того, что в основе каждого из них лежит та или иная модель машинного обучения.

Таблица 1.1 – Сравнение методов построения индексов

Метод		**Классические индексы**			**Обученные индексы**
		B-дерево	**Хеш-таблица**	**Фильтр Блума**	
Временная сложность	поиска	O(log N)	O(1) / O(N)	O(k)	O(1) / O(N)
	вставки	O(log N)	O(1) / O(N)	O(k)	(*)
Память		Высокая	Средняя	Низкая	Средняя
Поиск в диапазоне		+	-	-	+
Поиск единичного ключа		+	+	-	+
Проверка существования		+	+	+	+

$(*)$ — вставка в обученный индекс требует переобучения, сложность которого зависит от архитектуры используемой модели машинного обучения.

По приведенной таблице можно сделать вывод, что обученные индексы являются наиболее универсальным средством в плане существующих подзадач, как и B-деревья, однако по сравнению с ними имеют возможность сокращения времени поиска и используемой индексом памяти.

1.7 Нейронные сети и построение индексов

1.7.1 Понятие нейронных сетей

Обученные индексы, преимущества которых были описаны в предыдущем пункте, как уже было сказано, в своей основе содержат методы машинного обучения, одним из которых являются нейронный сети.

Нейронные сети [23], или искусственные нейронные сети представляют собой математическую модель, основанную на сетях нервных клеток живого организма — их организации и функционировании. Как и биологическая, искусственная нейронная сеть состоит из связанных между собой узлов, называемых нейронами, в которых происходит обработка информации с помощью трех элементов (рисунок 1.14):

— синапсов или связей, характеризующихся весами, с помощью которых происходит изменение входных сигналов;
— сумматора, складывающего входные взвешенные входные сигналы, сюда же включаются смещения, отражающие увеличение или уменьшение выходного сигнала и обычно включающиеся в матрицу весов, путем добавления фиктивного входного сигнала всегда равного единице;
— функции активации, определяющая значение выходного сигнала по выходу сумматора.

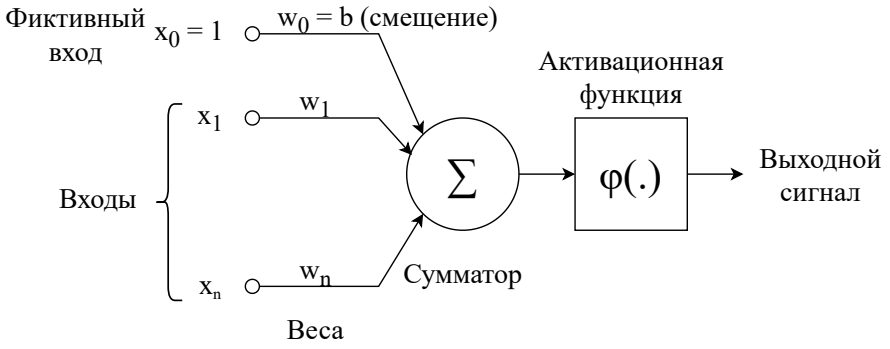

Рисунок 1.14 – Модель нейрона

Нейроны объединяются в слои, которые делятся на:

— входной слой, принимающий входные сигналы;
— выходной слой, выдающий прогнозируемые значения;

— скрытые слои, располагающиеся между входным и выходным слоем и выполняющие обработку.

Число скрытых слоев и число нейронов в каждом из них определяют сложность модели. В простейшем случае скрытые слои могут отсутствовать. Нейронные сети же с двумя и более скрытыми слоями называют глубокими [24]. Пример нейронной сети, являющейся глубокой, приведен на рисунке 1.15.

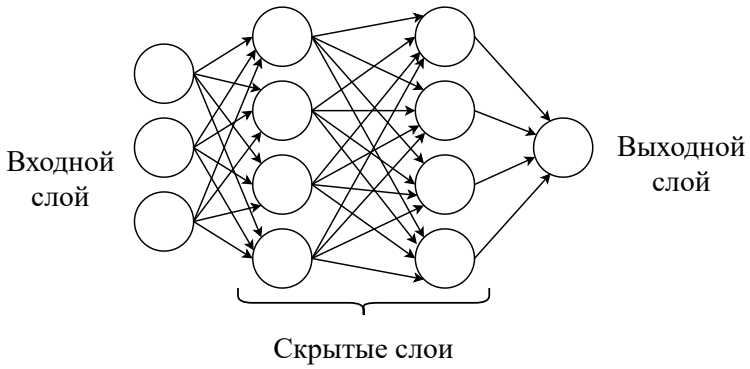

Рисунок 1.15 – Глубокая нейронная сеть

Обучение нейронной сети происходит за счет изменения ее параметров на основе большого количества обучающих данных. В процессе обучения сеть изменяет веса, минимизируя ошибку, определяющейся функцией потерь, между предсказанными и реальными значениями. Обычно обучение происходит путем прямого распространения сигнала через сеть и обратного распространения ошибки, в ходе которого корректируются веса и смещения слоев.

Определение характеристик нейронной сети: числа слоев, количества нейронов в каждом из них, активационных функций, функции потерь и т. д. — является нетривиальным и зависит от решаемой задачи.

1.7.2 Применение нейронных сетей к построению индексов

Как уже было сказано выше, индекс представляет собой модель, предсказывающая по переданному ей в качестве входных данных ключу его положение. В случае, если массив данных, по которому происходит поиск, отсортирован, описанная модель аппроксимирует функцию распределения ключей $F(K)$. При этом искусственную нейронную сеть, обучаемую согласно алгоритму обратного распространения, можно рассматривать как практический механизм реализации нелинейного отображения «вход-выход» общего вида [25], то есть как аппроксиматор функций.

Таким образом нейронная сеть, аппроксимирующая функцию распределения ключей, может быть применена в качестве поискового индекса. Архитектура такой сети должна иметь один нейрон на входном слое, соответствующий ключу, подаваемому в качестве входных данных, и один нейрон на выходом слое, соответствующий позиции или значению функции распределения в качестве выходных данных. При этом конфигурация скрытых слоев может быть различной.

Для построения индекса на основе нейронной сети необходимо обучить описанную модель, для чего требуется исходный набор ключей, по которому вычисляются значения функции распределения. На основании значений ключей и соответствующих им значений функций распределения происходит обучение модели, которая используется как индекс.

1.8 Постановка задачи

На основе рассуждений, представленных в данном разделе, можно сделать вывод, что для построения индекса на основе нейронной сети требуется:

— исходный набор ключей в качестве входных данных;
— правила их предварительной обработки;
— модель нейронной сети в качестве основы будущего индекса;
— алгоритм обучения нейронной сети, результатом работы которого является обученная модуль, представляющая собой индекс.

Формально данная задача может быть описана с помощью IDEF0-диаграммы нулевого уровня, представленной на рисунке 1.16.

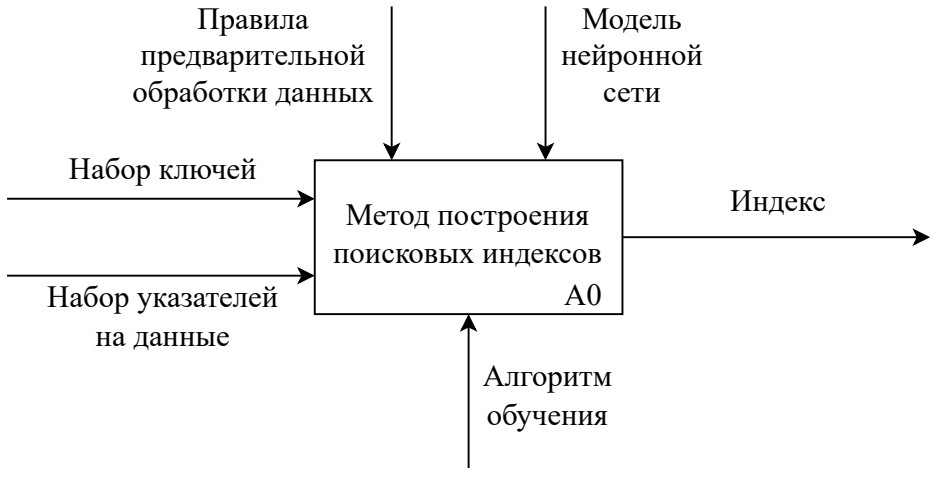

Рисунок 1.16 – Постановка задачи

Вывод

В данном разделе был проведен анализ предметной области, описаны особенности индексов в реляционных базах данных. Также было представлено описание построения индексов на основе базовых структур и применение методов машинного обучения к построению индексов. Проведено сравнение индексов на основе B-деревьев, хеш-таблиц и битовых карт с индексами на основе моделей машинного обучения. Описано применение нейронных сетей к построению индексов. Была описана формальная постановка задачи в виде IDEF0-диаграммы.

2 Конструкторская часть

2.1 Требования и ограничения метода

Метод построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей (далее – метод построения индексов) должен:

1. получать из таблицы реляционной базы данных набор ключей и набор соответствующих указателей на записи в индексируемой таблице реляционной базы данных или иных значений, выполняющих роль указателей;
2. выполнять предварительную обработку полученных наборов, такую, как их совместную сортировку по значениям ключей, получение позиций ключей в отсортированном виде и нормализацию ключей и позиций;
3. обучать модель нейронной сети на подготовленных набора ключей и позиций;
4. обеспечивать поиск записи (диапазона записей) таблицы по ключу (диапазону ключей) с использованием обученной модели;
5. обеспечивать корректность операции поиска после вставки/удаления новых записей путем переобучения модели;

На разрабатываемый метод накладываются следующие ограничения:

— в качестве ключей на вход принимаются целые числа для исключения решения дополнительной задачи преобразования входных данных;
— ключи во входном наборе уникальны.

2.2 Особенности метода построения индекса

2.2.1 Общее описание метода построения индекса

Основные этапы метода построения индекса приведены на функциональной декомпозиции метода на рисунке 2.1.

На вход методу подается набор уникальных целочисленных ключей, которые перед обучением модели глубокой нейронной сети проходят предварительную обработку по определенным правилам, описанным далее. Отдельным этапом выделено получение значений функций распределения для каждого ключа, относящееся к предварительной обработке, но представляющее собой ее

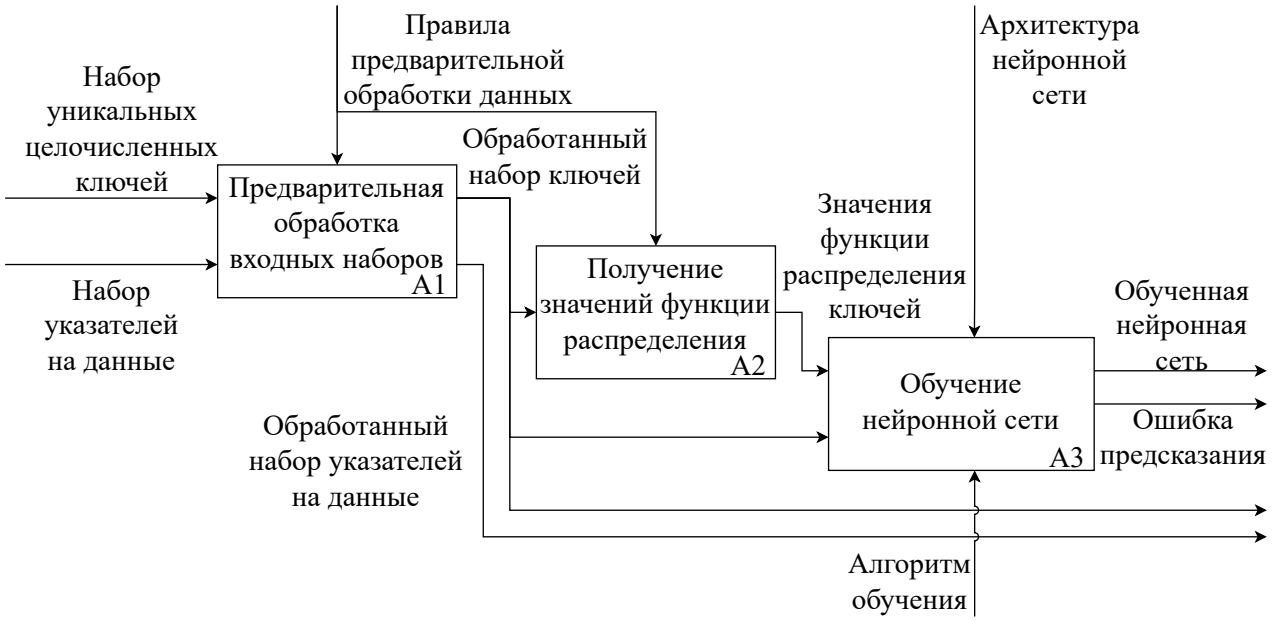

Рисунок 2.1 – Функциональная схема метода построения индекса

ключевой момент. Полученные после первых двух этапов обработанные ключи и соответствующие значения функций используются для обучение модели глубокой нейронной сети в качестве признаков и меток соответственно.

Ключевым моментом метода является представление в отсортированном (по ключам) виде наборов ключей и набора соответствующих указателей на данные. Именно отсортированный вид позволяет использовать закономерность распределения ключей по позициям для обучения модели, предсказывать позиции ключей и уточнять их.

Результатом работы метода является структура данных, представляющая собой индекс на основе глубокой нейронной сети и имеющая следующие поля:

— отсортированный массив ключей, поданных на вход;
— отсортированный по значениям ключей массив указателей на данные, соответствующие ключам;
— модель обученной глубокой нейронной сети, с помощью которой будет предсказываться положение ключа в отсортированном массиве;
— средняя и максимальная абсолютные ошибки предсказания позиции ключа, для ее уточнения и возврата верного указателя на данные.

Краткое описания индекса, являющегося результатом работы метода, как структуры данных представлено на рисунке 2.2.

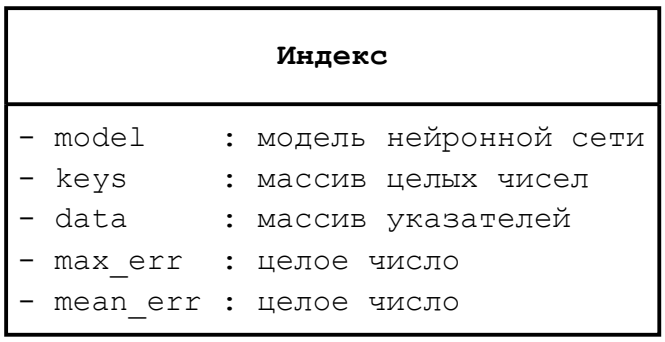

Рисунок 2.2 – Индекс как структура данных

Подробное описание каждого этапа приведено в следующих пунктах данного подраздела.

2.2.2 Предварительная обработка данных

Разрабатываемый метод построения индекса предполагает предварительную обработку набора целочисленных ключей, схема алгоритма которой представлена на рисунке 2.3.

Рисунок 2.3 – Схема алгоритма предварительной обработки данных

На вход подаются согласованные массивы ключей и указателей на данные, то есть считается, что ключ, стоящем на первой позиции в массиве ключей, идентифицирует данные по указателю, стоящем на первой позиции в массиве указателей; ключ, стоящей на второй, — указатель, стоящий на второй, и так далее. С учетом этого происходит согласованная сортировка двух массивов по значениям ключей.

Далее для последующего обучения модели глубокой нейронной сети производится нормализация ключей, выступающих в качестве входных данных сети, в диапазон $[0, 1]$, для чего используется метод минимакс-нормализации, при котором нормализованное значение вычисляется по формуле:

$$x_{\text{норм}} = \frac{x - x_{\text{мин}}}{x_{\text{макс}} - x_{\text{мин}}}, \tag{2.1}$$

где $x_{\text{норм}}$ — нормализованное значение ключа;

x — натуральное значение ключа;

$x_{\text{мин}},\ x_{\text{макс}}$ — минимальное и максимальное возможное значение ключа в наборе соответственно.

Далее полученный набор ключей размечается путем вычисления для каждого ключа K значения функции распределения F по его позиции P в отсортированном массиве и количества индексируемых ключей N с помощью формулы:

$$F(K) = \frac{P}{N}. \tag{2.2}$$

На этом предварительная обработка завершается и полученные отсортированные массивы ключей и указателей, а также соответствующие значения функции распределения передаются в качестве входных данных на этап обучения модели глубокой нейронной сети.

Полное описание алгоритма предварительной обработки представлено на листинге 2.1.

Листинг 2.1 — Предварительная обработка данных

Вход:
$keys$: массив целочисленных ключей;
$data$: массив указателей на данные, соответствующие ключам;
N : длина массивов.
Выход:
$keys$: отсортированный массив нормализованных ключей;
$data$: отсортированный по массиву ключей массив указателей;
cdf : массив значений функции распределения.

begin
  сортировать $keys$ и $data$ по $keys$;
  ▷ здесь и далее под операцией к вектору (массиву) и числу понимается
  ▷ применение данной операции с данным числом к каждому элементу вектора
  $keys \leftarrow \frac{keys - keys[0]}{keys[N-1] - keys[0]}$;
  $positions \leftarrow [0, 1, \ldots, N-1]$;
  $cdf \leftarrow \frac{positions}{N-1}$;
  return $keys$, $data$, cdf;
end

2.2.3 Разработка архитектуры глубокой нейронной сети

Полученные на этапе предварительной обработки массивы ключей и соответствующих им значений функций распределения используются для обучения глубокой нейронной сети. Массив данных, хранимый в индексе, используется для возврата нужных данных при поиске, но не для обучения.

Задача построения индекса на основе нейронной сети сводится к задаче аппроксимации функции распределения, для решения которой подходят полносвязные нейронные сети.

За основу архитектуры глубокой нейронной сети принята архитектура из исследования [5], которая представлена на рисунке 2.4. Это полносвязная нейронная сеть с двумя скрытыми слоями по 32 нейрона. В качестве функции активации в каждом нейроне скрытых слоев используется линейный выпрямитель или ReLU (*Rectified Linear Unit*), значение которой вычисляется по формуле (2.3).

$$f(x) = \max(0, x). \tag{2.3}$$

Активационная функция выходного слоя является линейной.

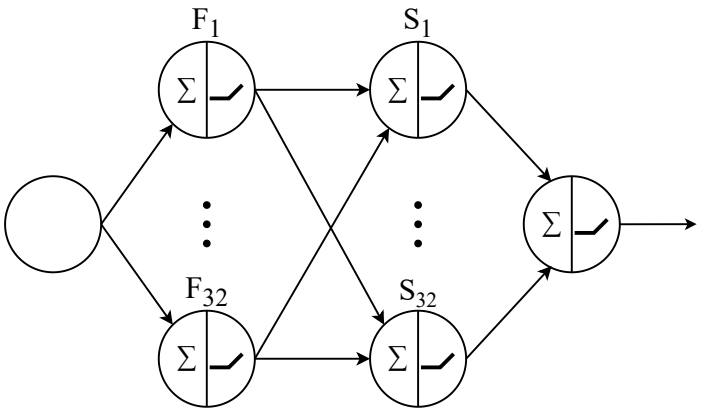

Рисунок 2.4 – Полносвязная нейронная сеть с двумя скрытыми слоями

Для исследования возможности увеличения точности предсказания, и как следствие уменьшение времени поиска, в качестве модели глубокой нейронной сети, представляющей основу индекса, используется полносвязная нейронная сеть с тремя слоями, представленная на рисунке 2.5. Число нейронов в слоях и активационные функции приняты такими же, как в случае глубокой нейронной сети с двумя скрытыми слоями.

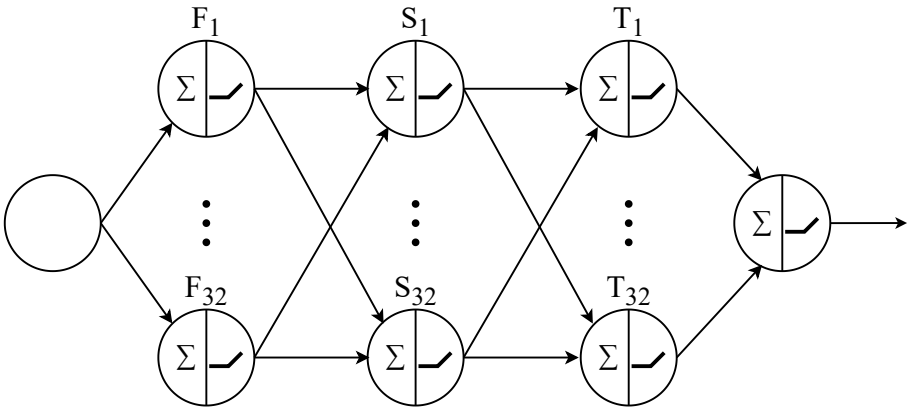

Рисунок 2.5 – Полносвязная нейронная сеть с тремя скрытыми слоями

Обучение обоих моделей глубокой нейронной сети начинается с инициализации весов случайно сгенерированными значениями по распределению $U(-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}})$. Собственно обучение проводится методом стохастического градиентного спуска с оптимизацией в качестве функции потерь среднеквадратической ошибки (*MSE — mean squared error*). Описание алгоритма обучения приведено на листинге 2.2.

Листинг 2.2 — Алгоритм обучения глубокой нейронной сети на основе градиентного спуска

Вход:
$keys$: массив нормализованных ключей;
cdf : массив значений функции распределения для каждого ключа;
N : длина массивов;
$epochs$: количество эпох;
$alpha$: скорость обучения.
Выход:
$model$: обученная модель.

begin
  $model \leftarrow$ структура модели;
  ▷ вектор смещений включей в матрицу весов
  $model.weights \leftarrow U(-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}})$; ▷ случайные значения из распределения
  for $epoch \leftarrow 1$ **to** $epochs$ **do**
    согласованно перемешать $keys$ и cdf;
    for $i \leftarrow 0$ **to** $N - 1$ **do**
      $\hat{y} = model.predict(key[0])$; ▷ прямой проход
      $mse = (\hat{y} - cdf[i])^2$;
      $gradients \leftarrow backward\_pass(model, mse)$; ▷ обратный проход
      $model.weights \leftarrow model.weights - alpha \cdot gradients$
    end
  end
end

Так как в случае поискового индекса глубокая нейронная сеть будет предсказывать положения только тех ключей, на которых она обучалась, явления переобучения нейронной сети является положительным, поэтому в качестве одного батча выступает одна пара (ключ; значение функции распределения), что также отражено на листинге выше.

2.3 Разработка алгоритмов поиска и вставки

Основной операцией, выполняемой с помощью индекса, является поиск, функциональная схема выполнения которого представлена на рисунках 2.6-2.7.

Разрабатываемый алгоритм поиска должен поддерживать операцию поиска по единичному ключу и по диапазону ключей, то есть принимать диапазоны, представленные в виде операций сравнения `=, >, <, >=, <=`, а также `a <= key <= b`, где key — ключ, a, b — границы диапазона.

Рисунок 2.6 – Функциональная схема нулевого уровня поиска

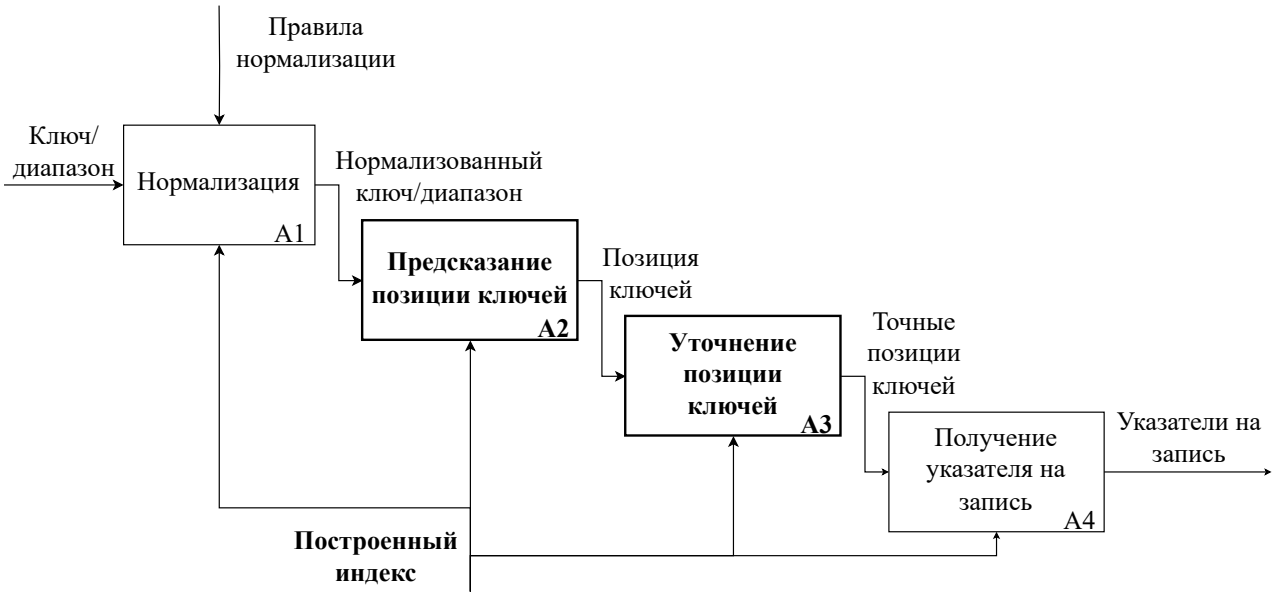

Рисунок 2.7 – Функциональная схема первого уровня поиска

В случае ограничения с одной стороны производится поиск позиции одного ключа, со значением соответствующим значению границы, с двух сторон — двух ключей. По найденным позициям и позициям первого и последнего ключа формируется диапазон позиций в соответствии с заданным ограничением. По найденному диапазону позиций выполняется получение указателей на нужные записи.

Получаемые по результатам поиска с помощью модели глубокой нейронной сети позиции требуют уточнения, происходящего за счет получаемого в результате обучения модели среднего и максимального отклонения от истинного расположения. Так ключи в индексе отсортированы применяется алгоритм бинарного поиска. Подробный алгоритм поиска представлен на листинге 2.3.

Листинг 2.3 — Алгоритм поиска

Вход:

$keys$: кортеж двух значений ключей, задающие ограничения поиска;

$constraints$: ограничение границ (включая (0)/не включая (1));

$index$: построенные индекс.

Выход:

$data$: массив указателей на записи.

```
1  function clarify(key, limit, is_lower, constraint, index)
2      if is_lower и limit == None then
3          return 0;
4      else if limit == None then
5          return index.keys.size − 1;
6      end
7      bin_lower ← max(limit − index.error, 0);
8      bin_upper ← min(limit + index.error, index.keys.size − 1);
       ▷ бинарный поиск, но возвращаются итоговые нижняя и верхняя границы
9      lower, upper ← binary_search(key, bin_lower, bin_upper);
10     if lower == upper then
11         return lower − (−1)^(is_lower) · constraint;
12     end
13     if is_lower then
14         return upper;
15     end
16     return lower;
17 end
18 begin
       ▷ при передаче None возвращается None
19     limits ← index.normalize(keys);
20     positions_limits ← index.predict(limits);
       ▷ lower, upper — нижняя и верхняя границы позиций ключей в диапазоне
21     lower ← clarify(key[0], positions_limits[0], True, constraints[0], index);
22     upper ← clarify(key[1], positions_limits[1], False, constraints[1], index);
23     return index.data[lower : upper]
24 end
```

Функциональная схема вставки приведена на рисунке 2.8.

Она осуществляется путем комбинирования алгоритма поиска и построения: сначала происходит поиск позиции, куда должна произойти вставка, далее новые ключ и указатель помещаются в данную позицию отсортированных массивов и в конце происходит дообучение модели на новом наборе данных, то

Рисунок 2.8 – Функциональная схема вставки

есть обучение модели происходит с уже имеющихся весов. Алгоритм вставки представлен на листинге 2.4.

Листинг 2.4 — Алгоритм вставки

Вход:

key : новый ключ;

$data$: указатель, соответствующий ключу;

$index$: индекс.

Выход:

$index$: переобученный индекс.

```
1 begin
2     position ← index.search(key);          ▷ уточненная позиция
3     insert(index.keys, key, position);
4     insert(index.data, data, position);
5     index.train();
6 end
```

2.4 Разработка архитектуры программного обеспечения

Схема архитектуры программного обеспечения, реализующего метод построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей, представлена на рисунке 2.9

Как видно из рисунка, программное обеспечение должно включать три модуля: модуль взаимодействия с пользователем, взаимодействия с реляцион-

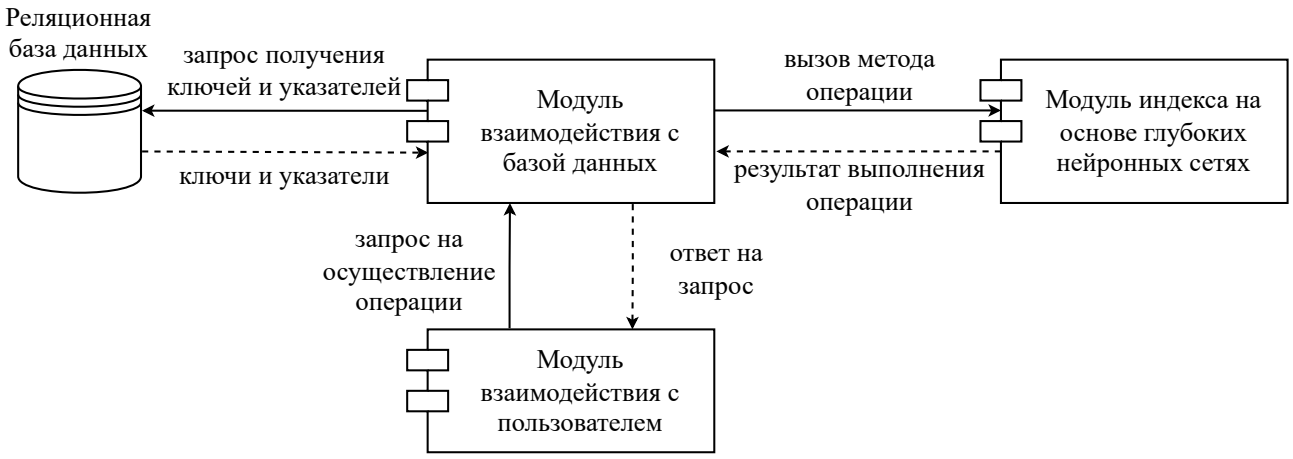

Рисунок 2.9 – Структура программного обеспечения

ной базой данных и модуля, реализующего индекс.

2.5 Данные для обучения и тестирования индекса

Так как в основе индекса на основе глубоких нейронных сетей лежит аппроксимация функции распределения ключей, работу разработанного метода необходимо протестировать на различных законах, перечисленных далее.

— Равномерный закон $R[a, b]$, функция распределения которого описывается формулой (2.4).

$$F(x) = \begin{cases} 0, & \text{если } x < a, \\ \frac{x-a}{b-a}, & \text{если } a \leq x \leq b, \\ 1, & \text{если } x > b. \end{cases} \tag{2.4}$$

Нормализованные ключей лежат в диапазоне $[0, 1]$, значения функции распределения за пределами этого диапазона не представляют интереса для построения индекса, поэтому можно считать, что функция имеет вид, представленный формулой (2.5).

$$F(x) = x, \ x \in [0, 1]. \tag{2.5}$$

— Нормальный закон $N(\mu, \sigma^2)$, функция распределения которого описывается формулой (2.6).

$$F(x) = \int_{-\infty}^{x} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(t-\mu)^2}{2\sigma^2}\right) dt. \tag{2.6}$$

Для тестирования метода построения по заданным законам генерируются значения ключей, по совокупности которых формируется эмпирическая функция распределения, как это было описано выше.

Также для проверки работы метода требуется оценить его работоспособность на реальных данных, в качестве которых выбраны уникальные идентификаторы элементов из открытого набора географических данных *OpenStreetMap*, или *OSM* [26], функция распределения которых имеет более сложный вид, чем основные законы распределения.

Графики функций распределения каждого из набора входных ключей, представлены на рисунке 2.10.

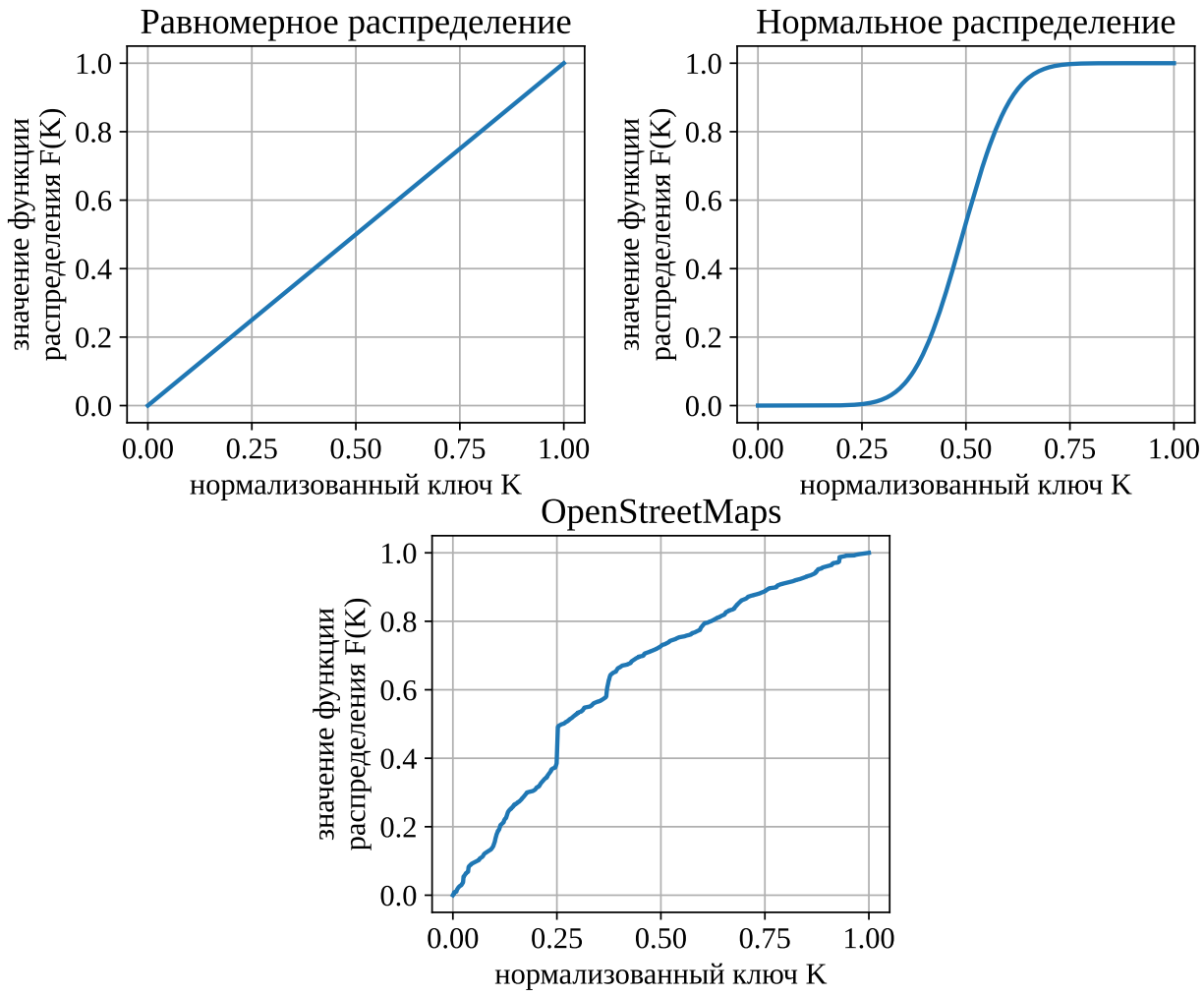

Рисунок 2.10 – Графики функций распределения ключей из наборов данных для обучения и тестирования

Вывод

В данном разделе были представлены требования к разрабатываемому методу, описаны алгоритмы построения индекса, осуществления операций поиска и вставки с помощью построенного индекса, представлена архитектура глубокой нейронной сети, а также данные для тестирования и обучения.

3 Технологическая часть

3.1 Выбор средств программной реализации

Для реализации метода построения индекса в реляционной базе данных на основе глубоких нейронных сетей в качестве языка программирования выбран Python 3.10 [27]. В качестве библиотеки глубокого обучения выбран Pytorch 2.0.0 [28]. Для работы с массивами данных выбрана библиотека numpy [29].

В качестве реляционной системы управления базами данных выбрана SQLite [30], предоставляющая программный интерфейс виртуальных таблиц, позволяющих реализовать пользовательский поисковый индекс. Виртуальные таблицы являются одним из видов расширений SQLite, программный интерфейс которых предоставляется на языке C [31], который и выбран в качестве языка программирования для взаимодействия с реляционной базой данных.

Для обеспечения взаимодействия между компонентом работы с базой данных и компонентом, непосредственно реализующий индекс, используются библиотеки языка C `Python.h` для работы с объектами языка Python и `numpy/arrayobject.h`, предоставляющая программный интерфейс для работы с numpy-массивами, которые являются основным типом данных, через который происходит взаимодействие модулей.

3.2 Реализация программного обеспечения

3.2.1 Форматы входных и выходных данных

Основой для построения индекса в качестве входных выступают данные из таблицы реляционной базы данных SQLite. Требованием к таблице является наличие атрибута целочисленного типа (`INTEGER`) с уникальными значениями (`UNIQUE`).

Для создания индекса в аргументах соответствующего запроса должен присутствовать идентификатор столбца, удовлетворяющего требованию описанному выше, и строковое имя модели индекса, на основе которой он строится (`FCNN2` — для модели с двумя скрытыми слоями, `FCNN3` — с тремя).

В качестве входных данных компонента, реализующего индекса, является набор значений столбца, идентификатор которого был указан в аргументах

запроса на создание индекса, и идентификаторы строк `ROWID` индексируемой таблицы.

Выходным данных является указатель на объект, представляющий индекс на основе глубокой нейронной сети, обученный на предоставленных входных данных.

Формат входных и выходных данных операций с индексом (поиска и вставки) описан в следующем пункте.

3.2.2 Поддерживаемые виды запросов

Разработанное программное обеспечения предоставляет возможность работы только с запросами фильтрации, представленными оператором `WHERE` следующими со следующими условиями:

— `column operator value`,
  где `column` — имя проиндексированного столбца,
  `operator` — одна из операций сравнения: `=, <, >, <=, >=`,
  `value` — некоторое целочисленное значение.
— `column BETWEEN value1 AND value2`,
  где `column` — имя проиндексированного столбца,
  `value1, value2` — целочисленные значения, представляющие нижнюю и верхнюю границы диапазона.

Выходным значением из модуля на языке Python, реализующего индекс, по данным запросам является numpy-массив с соответствующими запросу значениями ROWID, а результатом работы программного обеспечения — набор записей таблицы с ROWID из представленного массива.

Для вставки поддерживается стандартный запрос `INSERT`, результатом которого является индекс с переобученной на новых данных моделью. Запросы удаления и изменения не поддерживаются, так как являются вторичными для оценки работы метода.

3.2.3 Программный интерфейс виртуальных таблиц

Виртуальные таблицы SQLite [32] — это объект базы данных, представляющий с точки зрения инструкций SQL обычную таблицу или представление, но обрабатывающий запросы посредством вызова функций программного интерфейса, которые реализуются пользователем.

Виртуальные таблицы являются расширением SQLite, регистрация которых происходит с использованием макросов и функции инициализации, представленных на листинге 3.1.

Листинг 3.1 — Инициализация расширения

```
#include <sqlite3ext.h>

SQLITE_EXTENSION_INIT1

int sqlite3_extension_init(sqlite3 *db,
                           char **pzErrMsg,
                           const sqlite3_api_routines *pApi)
{
  int rc = SQLITE_OK;
  SQLITE_EXTENSION_INIT2(pApi);

  /* инициализация расширения */

  return rc;
}
```

При инициализация расширения виртуальной таблицы должен быть зарегистрирован модуль, описывающийся структурой `sqlite3_module`, с помощью функции регистрации `sqlite3_create_module`, подробное описание которых представлено на листинге 3.2

Листинг 3.2 — Структура и функция регистрации модуля виртуальной таблицы

```
struct sqlite3_module {
    int iVersion;
    int (*xCreate)(sqlite3*, void *pAux,
                int argc, char *const*argv,
                sqlite3_vtab **ppVTab,
                char **pzErr);
    int (*xConnect)(sqlite3*, void *pAux,
                int argc, char *const*argv,
                sqlite3_vtab **ppVTab,
                char **pzErr);
    int (*xBestIndex)(sqlite3_vtab *pVTab, sqlite3_index_info*);
    int (*xDisconnect)(sqlite3_vtab *pVTab);
    int (*xDestroy)(sqlite3_vtab *pVTab);
    int (*xOpen)(sqlite3_vtab *pVTab,
```

Продолжение листинга 3.2

```
                    sqlite3_vtab_cursor **ppCursor);
    int (*xClose)(sqlite3_vtab_cursor*);
    int (*xFilter)(sqlite3_vtab_cursor*,
                    int idxNum, const char *idxStr,
                    int argc, sqlite3_value **argv);
    int (*xNext)(sqlite3_vtab_cursor*);
    int (*xEof)(sqlite3_vtab_cursor*);
    int (*xColumn)(sqlite3_vtab_cursor*, sqlite3_context*, int);
    int (*xRowid)(sqlite3_vtab_cursor*, sqlite_int64 *pRowid);
    int (*xUpdate)(sqlite3_vtab *, int,
                    sqlite3_value **, sqlite_int64 *);
    /* представлены указатели на реализующиеся методы */
    /* при инициализации структуры, */
    /* остальные поля принимают значение 0 */
};

int sqlite3_create_module(
    sqlite3 *db,           /* соединение для регистрации модуля */
    const char *zName,       /* имя модуля */
    const sqlite3_module *, /* ссылка на структуру модуля */
    void *,                  /* данные для xCreate/xConnect*/
);
```

Методы, сигнатуры которых представлены на листинге 3.2, можно разделить на две группы.

Методы для взаимодействие с таблицей, как с некоторым объектом, к которым относятся:

— `xCreate` — создание виртуальной таблицы, при выполнении соответствующего запроса, представленного на листинге 3.3;
— `xConnect` — подключение к виртуальной таблице, вырывающийся при выполнении любого запроса к таблице, который является первым при повторном подключении к базе данных;
— `xDestroy` — удаление виртуальной таблицы при выполнении запроса, представленного на листинге 3.4;
— `xDisconnect` — удаление подключения к виртуальной таблице.

Листинг 3.3 — Запрос на создание виртуальной таблицы

```
CREATE VIRTUAL TABLE <table_name> USING <module_name>(arg1, ...);
```

Листинг 3.4 — Запрос на удаление виртуальной таблицы

```
DROP TABLE <table_name>;
```

Данные методы работают со структурой `sqlite3_vtab`, представленной на листинге 3.5. Для реализации нужных функциональностей указатель на данную структуру включается в пользовательскую, с которой уже работают представленные методы посредством преобразования типов. Это дает возможность передавать между методами вирутальной таблицы нужные данные.

Листинг 3.5 — Структура виртуальной таблицы

```
struct sqlite3_vtab {
    const sqlite3_module *pModule; /* модуль таблицы */
    int nRef; /* число ссылок, инициализирующееся ядром SQLite */
    char *zErrMsg; /* для передачи сообщений об ошибках ядру */
};
```

Методы прохода по записям таблицы, использующие для этого структуру курсора `sqlite3_vtab_cursor`, представленную на листинге 3.6, над которой также реализуют обертку для хранения необходимых для обработки переменных.

Листинг 3.6 — Структура курсора

```
struct sqlite3_vtab_cursor {
  sqlite3_vtab *pVtab; /* указатель на виртуальную таблицу */
};
```

Данная группа представлена методом обработки вставки, удаления и изменения записи (последний) и методами для прохода по записям таблицы при поиске:

— `xOpen` — создание и инициализации структуры курсора;
— `xBestIndex` — получение параметров фильтрации и выбор лучшего индекса для обработки запроса;
— `xFilter` — получение соответствующих параметрам фильтрации записей, установка курсора на первую из них;
— `xEof` — проверка окончания списка выбранных записей;
— `xNext` — переход к следующей записи;

— `xColumn` — обработка столбца записи;

— `xClose` — удаление структуры курсора;

— `xUpdate` — реализация запросов вставки, удаления и изменения.

Для реализации метода построения индекса используются оберточные структуры для виртуальной таблицы и курсора, представленные на листинге 3.7.

Листинг 3.7 — Пользовательские структуры виртуальной таблицы и курсора

```
typedef struct lindex_vtab {
    sqlite3_vtab base;  /* основа виртуальной таблицы */
    sqlite3 *db;        /* подключение к базе данных */
    sqlite3_stmt *stmt; /* инструкция доступа к записи по ROWID*/
    PyObject *lindex;   /* собственно объект индекса */
} lindex_vtab;

typedef struct lindex_cursor {
    sqlite3_vtab_cursor base; /* базовая структура курсора */
    PyObject *rowids;         /* массив выбранных ROWID */
    PyArrayIterObject *iter;  /* итератор по массиву ROWID */
} lindex_cursor;
```

На листинге 3.8 представлена реализация метода создания индекса, реализованного в качестве `xCreate`. Код инициализации и запуска обучение индекса в Python через программный интерфейс приведен на листинге 3.9.

Листинг 3.8 — Создание индекса

```
int lindexCreate(sqlite3 *db, void *pAux,
                 const int argc, const char *const *argv,
                 sqlite3_vtab **ppVtab, char **errMsg)
{
    lindex_vtab *vtab = sqlite3_malloc(sizeof(lindex_vtab));

    memset(vtab, 0, sizeof(*vtab));
    *ppVtab = &vtab->base;

    const char *vTableName = argv[2];
    const char *rTableName = vTableName + 4;

    char *sql_template = "SELECT sql FROM sqlite_master WHERE
    ↪  type='table' AND name='%s';";
```

Продолжение листинга 3.8

```
    char *schemaQuery = sqlite3_mprintf(sql_template, rTableName);

    char* messaggeError;
    char vSqlQuery[10000];
    strcpy(vSqlQuery, vTableName);
    sqlite3_exec(db, schemaQuery, callback, vSqlQuery, &messaggeError);
    char *resVSqlQuery = sqlite3_mprintf("%s;", vSqlQuery);

    sqlite3_declare_vtab(db, resVSqlQuery);

    sqlite3_free(schemaQuery);
    sqlite3_free(resVSqlQuery);

    long column_index = strtol(argv[3], NULL, 10);
    const char *model = argv[4];

    initPythonIndex(db, rTableName, model, column_index, vtab);

    char* result_query = sqlite3_mprintf("SELECT * FROM %s WHERE ROWID =
    ↪  ?;", rTableName);
    sqlite3_prepare_v2(db, result_query, -1, &vtab->stmt, NULL);

    return SQLITE_OK;
}
```

Листинг 3.9 — Инициализация индекса

```
int initPythonIndex(sqlite3 *db,
                    const char *const tableName,
                    const char *const modelName,
                    const long column_index,
                    lindex_vtab *vTab) {
    char* query = sqlite3_mprintf("SELECT ROWID, * FROM %s", tableName);

    sqlite3_stmt* stmt;
    sqlite3_prepare_v2(db, query, -1, &stmt, NULL);
    sqlite3_free(query);

    PyObject* builderModule = PyImport_ImportModule("indexes.builder");

    PyObject* builderClassName = PyObject_GetAttrString(builderModule,
    ↪  "LindexBuilder");
    PyObject* pyModelName = PyTuple_Pack(1,
    ↪  PyUnicode_FromString(modelName));
```

Продолжение листинга 3.9

```
16      PyObject* builder = PyObject_CallObject(builderClassName,
        ↪  pyModelName);
17
18      PyObject* lindex = PyObject_CallMethod(builder, "build", NULL);
19      PyObject* keys = PyList_New(0);
20      PyObject* rows = PyList_New(0);
21
22      int i = 0;
23      while (sqlite3_step(stmt) == SQLITE_ROW) {
24          int64_t key = sqlite3_column_int64(stmt, column_index);
25          int64_t rowid = sqlite3_column_int64(stmt, 0);
26
27          PyList_Append(keys, PyLong_FromLong(key));
28          PyList_Append(rows, PyLong_FromLong(rowid));
29
30          i++;
31      }
32
33      if (i) {
34          PyObject* train = PyUnicode_FromString("train");
35          PyObject* check = PyObject_CallMethodObjArgs(lindex, train,
            ↪  keys, rows, NULL);
36      }
37
38      vTab->lindex = lindex;
39      vTab->db = db;
40
41      Py_DECREF(keys);
42      /* ... */
43      sqlite3_finalize(stmt);
44
45      return SQLITE_OK;
46  }
```

Обработка параметров фильтрации приведена, получение массива подходящий строк и проход для их вывода приведены в приложении А на листингах А.1, А.2 и А.3 соответственно.

Реализация вставки в виртуальную таблицу приведена на листинге 3.10.

Листинг 3.10 — Реализация вставки в виртуальную таблицу

```
1   int lindexUpdate(sqlite3_vtab *pVTab,
2                    int argc, sqlite3_value **argv,
3                    sqlite_int64 *pRowid) {
```

Продолжение листинга 3.10

```
    if (!(argc > 1 && sqlite3_value_type(argv[0]) == SQLITE_NULL))
        return SQLITE_CONSTRAINT;

    lindex_vtab *lTab = (lindex_vtab*)pVTab;
    sqlite3 *db = lTab->db;
    int64_t column = sqlite3_value_int64(argv[2]);
    char *query = sqlite3_mprintf("INSERT INTO maps VALUES(%d);",
    ↪  column);
    sqlite3_exec(db, query, 0, 0, 0);
    sqlite3_int64 lastRowID = sqlite3_last_insert_rowid(db);

    PyObject* insert = PyUnicode_FromString("insert");
    PyObject* key = PyLong_FromLongLong(column);
    PyObject* data = PyLong_FromLongLong(lastRowID);

    PyObject_CallMethodObjArgs(lTab->lindex, insert, key, data, NULL);

    return SQLITE_OK;
}
```

3.2.4 Реализация индекса

Основой индекса является модель нейронной сети, реализация инициализации, построения и обучения которой, представлена на листинге 3.11.

Листинг 3.11 — Реализация модели нейронной сети

```
import torch
import numpy as np
from torch.utils.data import TensorDataset, DataLoader
from indexes.models.abs_model import AbstractModel

class PTModel(AbstractModel):

    def __init__(self, layers_num):
        self.pt_model = torch.nn.Sequential(
            torch.nn.Linear(1, 32),
            torch.nn.ReLU(),
            *([torch.nn.Linear(32, 32), torch.nn.ReLU()] * layers_num),
            torch.nn.Linear(32, 1)
        )

    def build(self):
        self.loss_function = torch.nn.MSELoss()
```

Продолжение листинга 3.11

```
        self.optimizer = torch.optim.SGD(self.pt_model.parameters(),
        ↪  lr=1e-2)

    def train(self, keys, positions):
        self.N = len(keys)
        input_tensor = torch.from_numpy(keys).unsqueeze(1).float()
        output_tensor = torch.from_numpy(positions).unsqueeze(1).float()

        dataset = TensorDataset(input_tensor, output_tensor)

        batch_size = 32
        dataloader = DataLoader(dataset, batch_size=batch_size,
        ↪  shuffle=True)

        num_epochs = 30
        for epoch in range(num_epochs):
            mean_ae = 0
            max_ae = 0

            epoch_loss = 0.0
            for i, (batch_inputs, batch_outputs) in
            ↪  enumerate(dataloader):
                self.optimizer.zero_grad()
                predictions = self.pt_model(batch_inputs)
                loss = self.loss_function(predictions, batch_outputs)
                loss.backward()
                self.optimizer.step()

                epoch_loss += loss.item()
                print(f"{i}/{len(dataloader)}, Loss: {epoch_loss / (i +
                ↪  1):.3e}", end='\r')

            epoch_loss /= len(dataset)
            print(f"Epoch [{epoch+1}/{num_epochs}], Loss:
            ↪  {epoch_loss:.3e}")

            if epoch_loss < 1e-5:
                break
```

Реализация класса разработанного обученного индекса на основе глубокой нейронной сети со всеми этапами и операциями: инициализацией, нормализацией, обучением, предсказанием, уточнением поиском и вставкой — представлена в приложению Б на листинге Б.1.

3.3 Сборка программного обеспечения

При сборке расширения SQLite компиляция и линковка происходит с флагами, обычно используемыми при сборке динамический библиотек: флаг `-fPIC` при компиляции в объектные файлы для создание позиционно-независимого кода и флаг `-shared` для получения файла динамической библиотеки. Дополнительными флагами при линковке являются флаги подключения библиотек `-lsqlite3` и `-lpython3.10`. Также при компиляции требуется указание путей к заголовочным файлам `Python.h` и `numpy/arrayobject.h`. Их автоматическое получения, а также ключевые моменты сборки приведены на листинге 3.12.

Листинг 3.12 — Ключевые моменты сборки программного обеспечения

```
PATHFLAG := -I$(INCDIR)
CFLAGS := -std=c99 $(PATHFLAG) -fPIC
ADD_LIBS :=  -lsqlite3 -lpython3.10
PYTHON_PATH:= $(shell pkg-config --cflags --libs python3)
NUMPY_PATH := -I$(shell pip show numpy
                          | grep -oP "(?<=Location: ).*"$\
                          | awk '{$$1=$$1};1')/numpy/core/include

SRCS := $(wildcard $(SRCDIR)*.c)
OBGS := $(patsubst $(SRCDIR)%.c,$(OUTDIR)%.o,$(SRCS))

.PHONY : clean build

build: lindex.so

%.so : $(OBGS)
        @mkdir -p $(@D)
        $(CC) $(ADD_LIBS) -shared $^ -o $@

$(OUTDIR)%.o : %.c
        @mkdir -p $(@D)
        $(CC) $(CFLAGS) $(PYTHON_PATH) $(NUMPY_PATH) -c $< -o $@
```

Для работы компонента индекса, реализованного на Python, требуется установка зависимостей из уже сформированного файла `requirements.txt` путем выполения команды, представленной на листинге 3.13. Также для штатной работы программного обеспечения требуется прописать путь к модулям, реализованным на языке Python, что приведено на том же листинге.

Листинг 3.13 — Подготовка для работы модулей Python

```
$ pip install -r requirements.txt
$ export PYTHONPATH=<путь_к_модулям>:$PYTHONPATH
```

3.4 Результаты тестирования

После разработки программного обеспечение было проведно автоматическое тестирование модуля индекса на основе глубоких нейронных сетей. Были использованы следующие классы эквивалентности:

— поиск существующего единичного ключа;
— поиск наименьшего и наибольшего в наборе ключей;
— поиск несуществующего ключа;
— поиск ключей по условию < с существующим ключом в качестве границы;
— поиск ключей по условию < с несуществующим ключом в качестве границы;
— два предыдущих класса по условиям >, <=, >=;
— поиск по диапазону с двумя границами;
— поиск по диапазону с двумя границами, в который попадают все ключи;
— поиск по диапазону с двумя границами, в который не попадает ни один ключ.

Результаты тестирования представлены на листинге 3.14

Листинг 3.14 — Результаты автоматического тестирования

```
======== short test summary info ================
PASSED tests.py::TestLindex::test_train
PASSED tests.py::TestLindex::test_middle
PASSED tests.py::TestLindex::test_first
PASSED tests.py::TestLindex::test_last
PASSED tests.py::TestLindex::test_not_exist
PASSED tests.py::TestLindex::test_range_lw_in
PASSED tests.py::TestLindex::test_range_lw_out
PASSED tests.py::TestLindex::test_range_gr_in
PASSED tests.py::TestLindex::test_range_gr_out
PASSED tests.py::TestLindex::test_range_le_in
PASSED tests.py::TestLindex::test_range_le_out
```

Продолжение листинга 3.14

```
PASSED tests.py::TestLindex::test_range_ge_in
PASSED tests.py::TestLindex::test_range_ge_out
PASSED tests.py::TestLindex::test_range_gle
PASSED tests.py::TestLindex::test_range_gle_all
PASSED tests.py::TestLindex::test_range_gle_none
PASSED tests.py::TestLindex::test_insert
======= 17 passed, 1 warning in 4.36s ===========
```

Также было проведено ручное интеграционное тестирование программного обеспечения, взаимодействие с которым происходит через командную строку sqlite3. Пример работы и тестирования представлен на листинге 3.15.

Листинг 3.15 — Пример работы программного обеспечения

```
$ sqlite3 test.db
sqlite> .load ./lindex
sqlite> create table maps(key INTEGER UNIQUE);
sqlite> .mode csv
sqlite> .import osm10.csv maps
sqlite> select * from maps;
5694768947
1000
4577603404
8742104813
2577217863
3465205493
10920113439
814309230
6943212874
1766254734
sqlite> create virtual table virtmaps using lindex(1, fcnn2);
Epoch [1/30], Loss: 1.963e-01
...
sqlite> select * from virtmaps where key = 3465205493;
3465205493
sqlite> select * from virtmaps where key > 3465205493;
4577603404
5694768947
6943212874
8742104813
10920113439
sqlite> select * from virtmaps where key < 3465205493;
1000
814309230
```

Продолжение листинга 3.15

```
1766254734
2577217863
sqlite> select * from virtmaps where key <= 3465205493;
1000
814309230
1766254734
2577217863
3465205493
sqlite> select * from virtmaps where key >= 3465205493;
3465205493
4577603404
5694768947
6943212874
8742104813
10920113439
sqlite> select * from virtmaps where key between 3465205493 and
↪  5694768947 ;
3465205493
4577603404
5694768947
sqlite> insert into virtmaps values(1);
Epoch [1/30], Loss: 5.933e-02
...
sqlite> select * from virtmaps where key = 1;
1
```

Вывод

В данном разделе был проведен выбор средств программной реализации, описан программный интерфейс виртуальных таблиц SQLite, приведены реализации модуля взаимодействия с реляционной базой данных и модуля индекса на основе глубоких нейронных сетей, также были описаны сборка и тестирование разработанного программного обеспечения.

4 Исследовательская часть

4.1 Предмет исследования

Характеристиками, определяющими эффективность индекса являются:

— время выполнения основных операций:
  — построения;
  — поиска;
  — вставки (являющейся комбинацией первых двух операций);
— память, занимаемая индексом.

Так как метод основан на использовании глубокой нейронной сети, характеристикой также является средняя абсолютная ошибка предсказания позиции ключей, получаемая в ходе обучения.

Предполагается зависимость описанных характеристик от объема индексируемых данных, а также от распределения ключей, поэтому исследование проводится на различном количестве ключей для распределений, описанных в подразделе 2.5: равномерного, нормального и распределения реальных данных OpenStreetMap. Также проводится сравнение времени поиска при моделях с различным числом скрытых слоев, описанных в подразделе 2.2.3, и с реализацией классического индекса в SQLite.

4.2 Зависимость времени построения индекса от количества ключей

На рисунке 4.1 приведен график зависимости времени построения индекса от количества ключей в индексируемом наборе OpenStreetMap при различных количествах скрытых слоев в модели в сравнении с классическим индексом SQLite.

Из приведенных графиков можно сделать вывод, что добавление дополнительного слоя в модель увеличивает время обучения, а следовательно и построения. По сравнению с индексом на основе модели с двумя скрытыми слоями индекс на основе модели с тремя увеличивает время построения на 10%. Время построения же классического индекса в SQLite в 25 раз ниже построения реализованного индекса.

По полученным данным исследования строится линейная регрессионная

модель, откуда вытекает линейная зависимость времени построения индекса от количества ключей в наборе, что объясняется достижением необходимой точности модели за 1-2 эпохи обучения при каждом размере данных, за которые происходит проход по всем значениям ключей. Аналогичная зависимость наблюдается и у классического индекса.

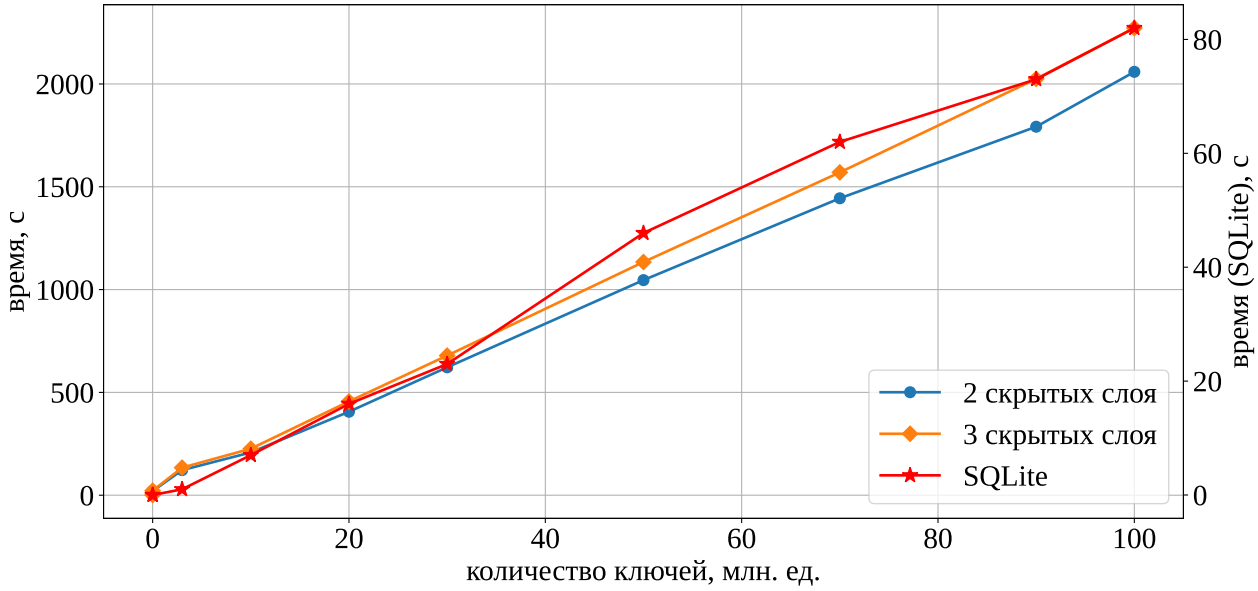

Рисунок 4.1 – График зависимости времени построения индекса от количества ключей (модели, OpenStreetMap)

4.3 Исследование времени поиска

На время поиска с использованием индекса, построенного с помощью разработанного метода, должна оказывать влияние абсолютная ошибка предсказания позиции ключа моделью глубокой нейронной сети, так как она определяет диапазон, в котором осуществляется уточнение с помощью бинарного поиска, поэтому в дополнение к исследованию зависимости времени поиска от количества ключей было также проведено исследование зависимости средней абсолютной ошибки от количества ключей.

На рисунка 4.2 и 4.3 приведены графики зависимостей средней абсолютно ошибки и времени поиска от количества ключей при различных распределениях с использованием модели с двумя скрытыми слоями.

По графикам на данных, распределенных по равномерному закону, функция распределения которых является линейной, имеет наименьшее значение абсолютной ошибки и, как следствие, меньшее время поиска. Наибольшая

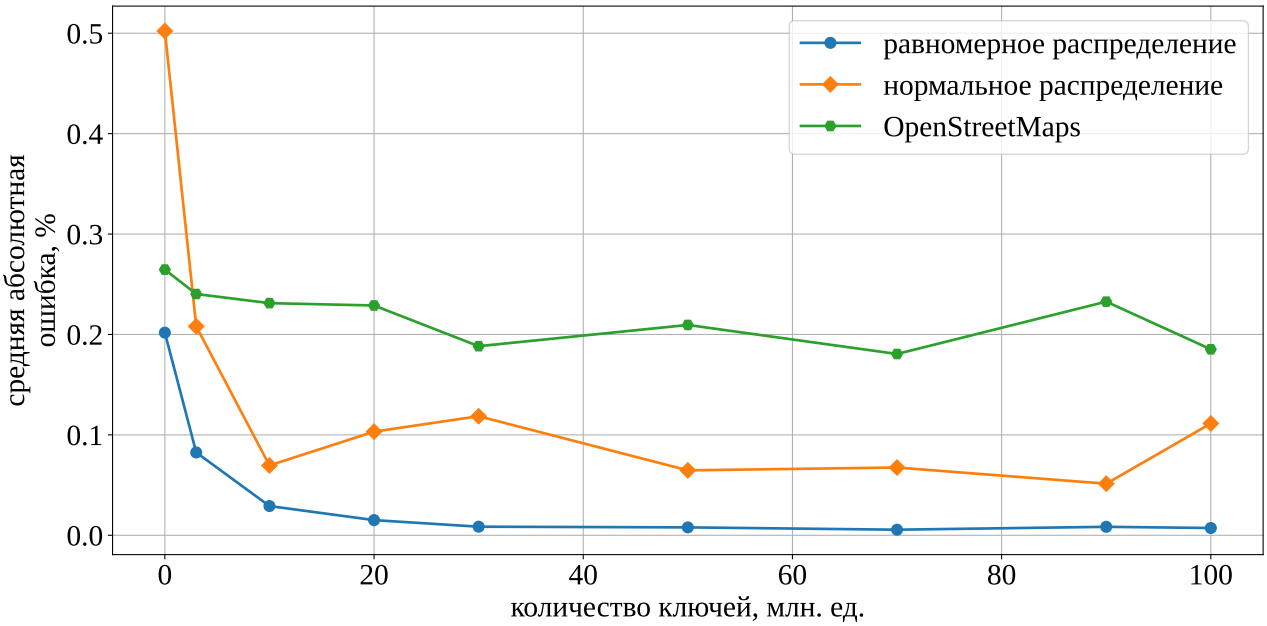

Рисунок 4.2 – График зависимости средней абсолютной ошибки от количества ключей (распределения, 2 скрытых слоя)

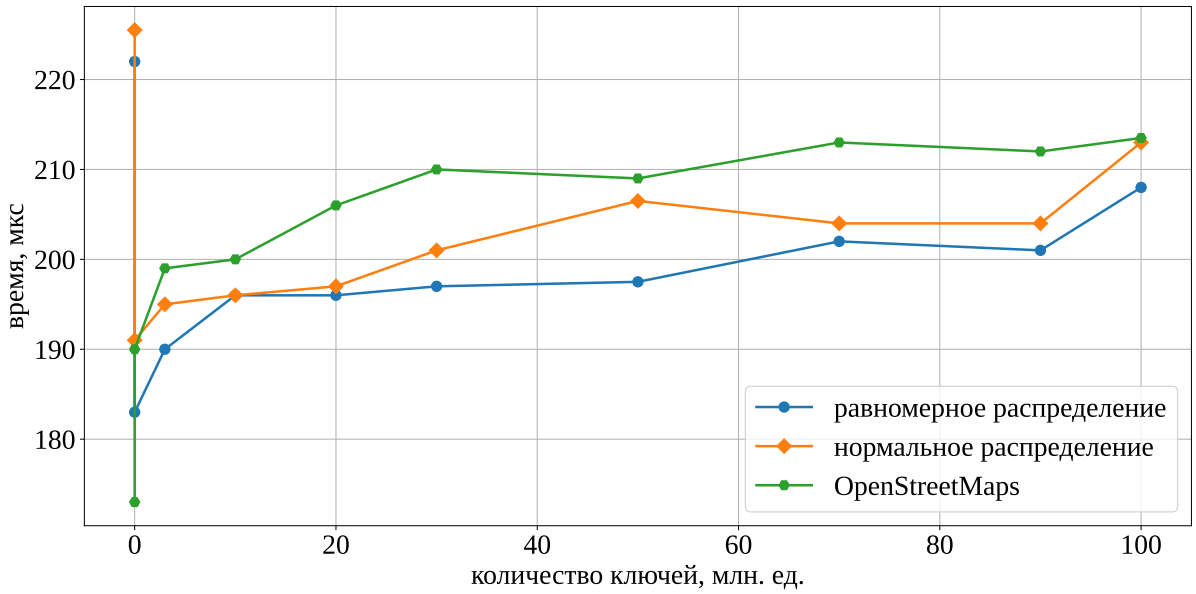

Рисунок 4.3 – График зависимости времени поиска от количества ключей (распределения, 2 скрытых слоя)

абсолютная ошибка и время поиска наблюдаются на реальных данных, так как график их функция распределения имеет более сложный вид нежели классические распределения. Однако при росте ошибки на 0.2% относительно равномерного распределения, время поиска на реальных данных увеличивается в среднем на 6%.

При этом при увеличении числа ключей отношение средней абсолютной ошибке к количеству ключей стремится к некотором постоянному значению,

то есть диапазон бинарного поиска будет составлять некоторую постоянную часть от числа ключей, то есть будет линейно расти.

На рисунка 4.4 и 4.5 приведены графики зависимостей средней абсолютно ошибки и времени поиска от количества ключей при реальных данных с использованием моделей с двумя и тремя слоями.

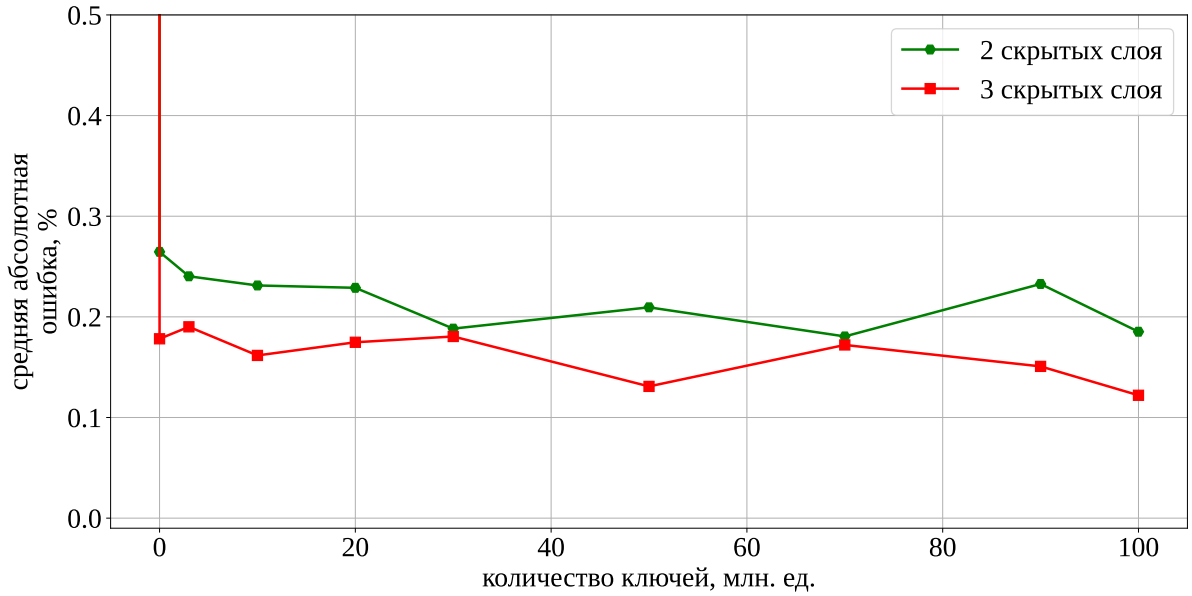

Рисунок 4.4 – График зависимости средней абсолютной ошибки от количества ключей (модели, OpenStreetMap)

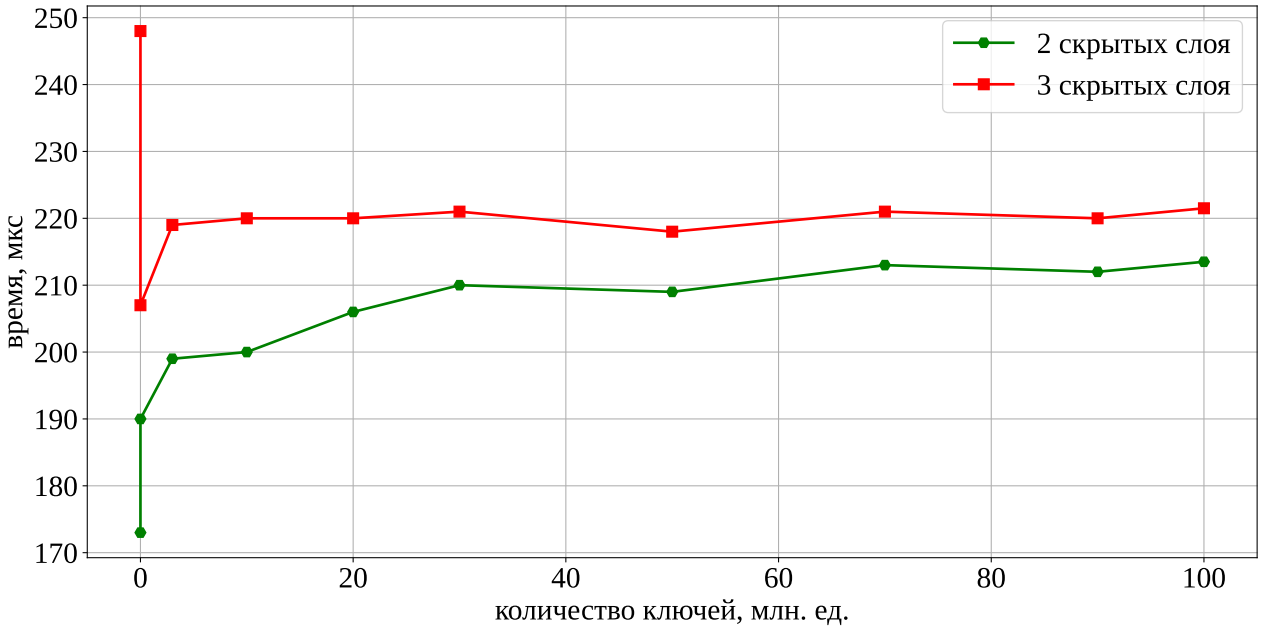

Рисунок 4.5 – График зависимости времени поиска от количества ключей (модели, OpenStreetMap)

Таким образом, добавлении третьего слоя оказалось неоправданным,

достигаемое уменьшение средней абсолютной ошибки на 0.06% и уменьшение времени бинарного поиска превысилось временем, затрачиваемым на подсчет дополнительных коэффициентов третьего слоя при предсказании.

На рисунке 4.6 представлен график зависимости времени поиска и его этапов (предсказания и уточнения) от числа ключей на реальных данных с использованием модели с двумя скрытыми слоями.

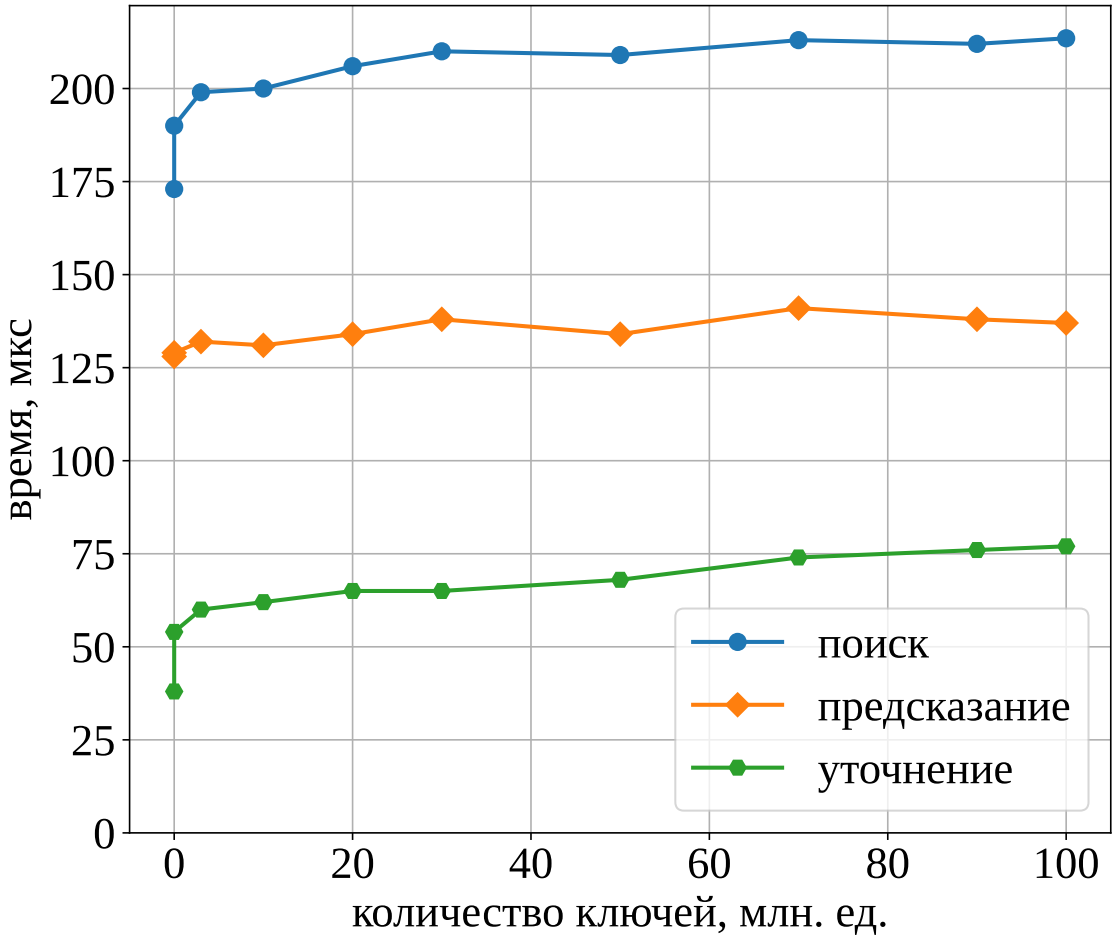

Рисунок 4.6 – График зависимости времени поиска, предсказания и уточнения от количества ключей (OpenStreetMap, 2 скрытых слоя)

Таким образом, предсказание выполняется за постоянное время, которое больше времени выполнения бинарного поиска с временной сложностью $O(logN)$. Таким образом, сложность поиска с использованием индекса, построенного разработанным методом, $O(logN)$. При этом коэффициент при N равен величине отношения средней абсолютной ошибки к числу ключей, которая составляет 0.2%.

Для оценки худшего случая построим нормированное распределение абсолютной ошибки, которое представлено на рисунке 4.7.

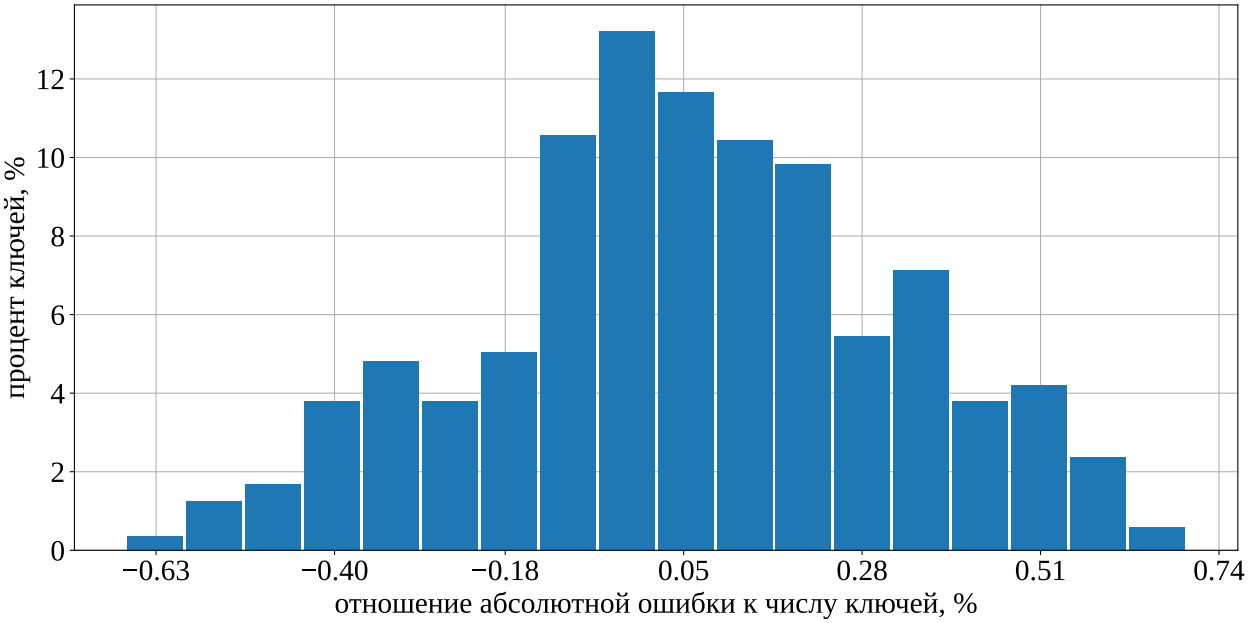

Рисунок 4.7 – Распределение абсолютной ошибки предсказания позиции

Из графика можно сделать вывод, что максимальная абсолютная ошибка составляет 0.6% от числа ключей, то есть в худшем случае бинарный поиск будет осуществляться в диапазоне $0.006N$.

На рисунке 4.8 представлен график зависимости времени поиска от числа ключей на реальных данных с использованием модели с двумя скрытыми слоями в сравнении с временем поиска без индекса в SQLite и с классическим индексом SQLite.

Таким образом, время поиска с использованием обоих индексов в среднем 17500 раз меньше, чем без индекса. При этом на взятых количествах ключей время поиска с индексом SQLite меньше, чем время поиска с использованием разработанного индекса, однако время поиска с использованием классического индекса SQLite растет быстрее времени поиска с использованием разработанного индекса при одинаковом проценте увеличения числа ключей. Так, при увеличении числа ключей в 2 раза время поиска SQLite увеличивается в 1.25 раза, а время поиска разработанного индекса в 1.03 раза.

4.4 Исследование времени вставки

На рисунке 4.9 представлен график зависимости времени вставки от числа ключей с использованием разработанного индекса и классического индекса SQLite.

Вставка представляет собой комбинацию поиска и построения индекса,

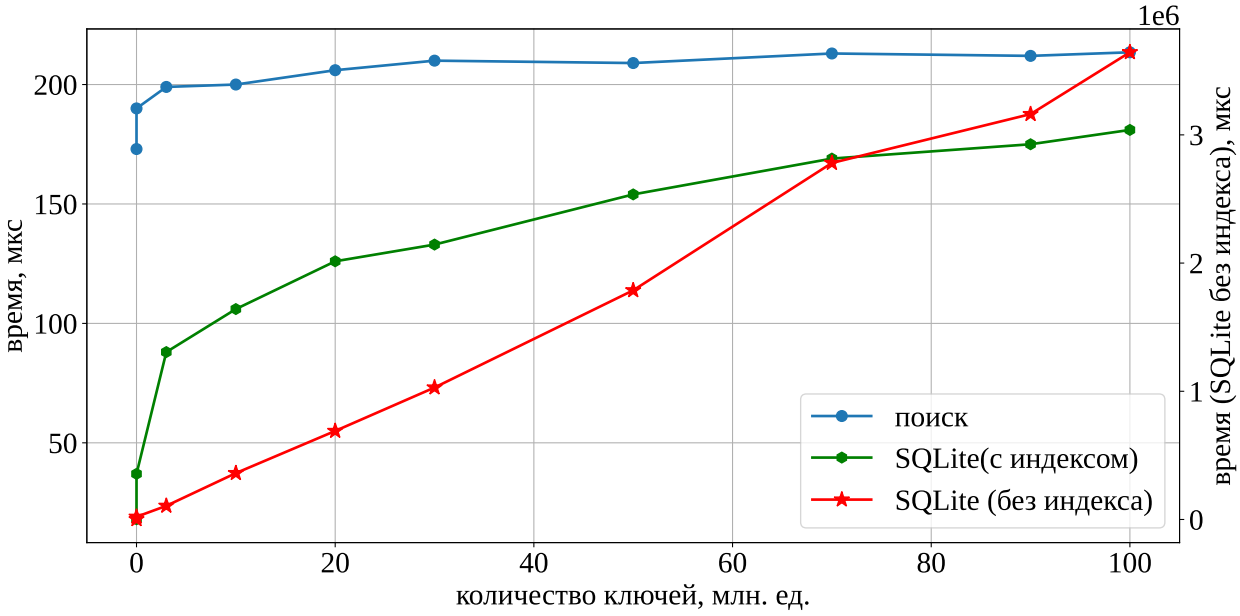

Рисунок 4.8 – Зависимость времени поиска от числа ключей с помощью разработанного индекса, без индекса в SQLite и с классическим индексом SQLite

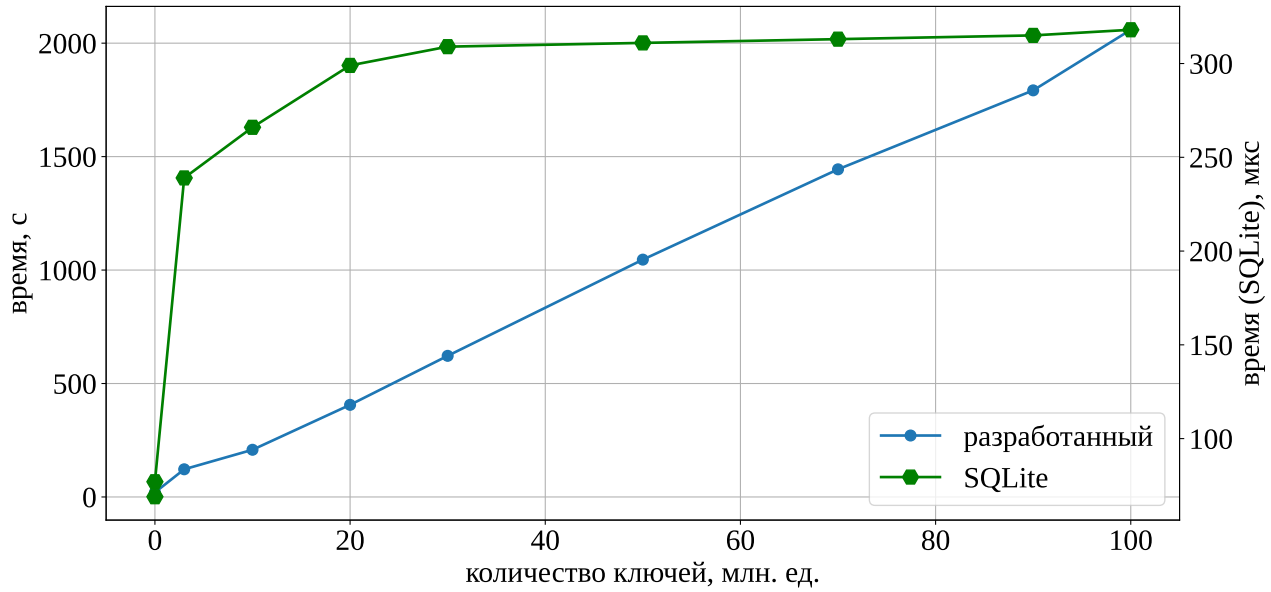

Рисунок 4.9 – График зависимости времени вставки от количества ключей (реальные данные, 2 скрытых слоя)

поэтому зависимость для разработанного индекса является линейкой, как у построения. При этом время вставки в разработанный индекс в 6.7 раза больше классического при максимальном количестве ключей.

4.5 Исследование памяти, используемой индексом

На рисунке 4.10 представлен график зависимости размера индекса от количества ключей.

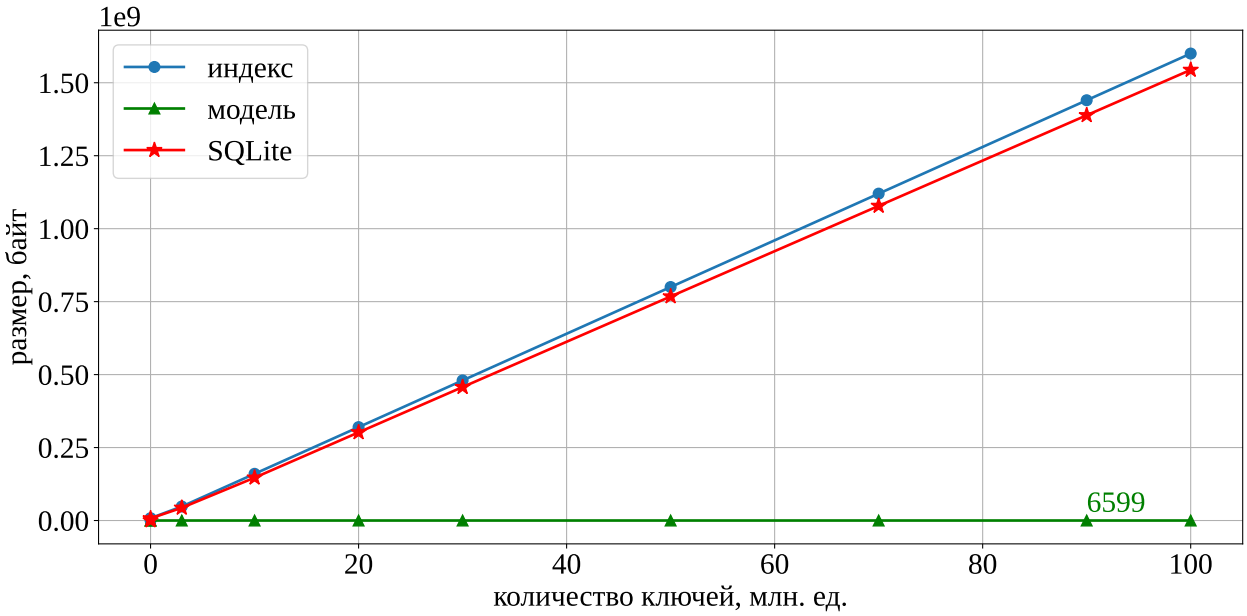

Рисунок 4.10 – График зависимости дополнительной памяти, занимаемой индесом, от количества ключей

Индекс, построенный разработанным методом, занимает место только под хранение модели, имеющей постоянный размер, который не зависит от числа ключей, и под два массива, память под который увеличивается линейно. То есть, если учитывать только память, выделяемую на поддержку структуры индекса, разработанный индекс имеет постоянный размер, соответствующий размеру модели нейронной сети, в отличие от традиционных индексов, в которых выделяется память под указатели для поддержки структуры, число которых зависит от числа ключей.

4.6 Результаты исследования

В ходе исследования были сделаны следующие выводы:

— временная сложность построения $O(N)$;

— средняя абсолютная ошибка предсказания модели стремится к некоторому постоянному значению по отношению к общему числу ключей, что дает линейную зависимость размера диапазона для бинарного поиска от числа ключей, причем с малым коэффициентом, равным 0.002

— в среднем случае, и 0.006 — в худшем для реальных данных;
- разработанный индекс имеет временную сложность поиска $O(\log N)$;
- временная сложность вставки составляет $O(N)$;
- разработанный индекс затрачивает на поддержание своей структуры не зависящее от числа данных количество памяти;
- с учетом хранимых ключей и указателей, память, затрачиваемая индексом, линейно растет при увеличении числа индексируемых ключей;
- увеличение точность путем добавления дополнительных слоев в нейронную сеть не оправданно из-за роста затрат на вычисления дополнительных коэффициентов третьего слоя;
- индекс быстрее работает на классических распределениях.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был разработан метод построения поисковых индексов в реляционной базе данных на основе глубоких нейронных сетей, для чего были выполнены следующие задачи:

- проведен анализ известных методов построения индексов;
- приведено описание построения индексов с помощью нейронных сетей;
- разработан метод построения индексов в реляционной базе данных на основе глубоких нейронных сетей;
- разработано программное обеспечение, реализующее данный метод;
- проведено исследование (по времени и памяти) операций поиска и вставки с использованием индекса, построенного разработанным методом, при различных объемах данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Коптенок Е. В.* Исследование способов ускорения поисковых запросов в базах данных // Вестник образовательного консорциума среднерусский университет. Информационные технологии. — 2019. — № 1(13). — С. 24—27.
2. *Reinsel D.* The Digitization of the World From Edge to Core // IDC White Paper. — 2018.
3. *Носова Т. Н.* Использование алгоритма битовых шкал для увеличения эффективности поисковых запросов, обрабатывающих данные с низкой избирательностью // Электротехнические системы и комплексы. — 2018. — № 1(38). — С. 63—67.
4. DAMA-DMBOK : Свод знаний по управлению данными // 2-е изд. — М. : Олимп-Бизнес. — 2020. — С. 828.
5. *Kraska T.* The Case for Learned Index Structures // Proceedings of the 2018 International Conference on Management of Data. — 2018. — С. 489—504.
6. *Григорьев Ю. А.* Реляционные базы данных и системы NoSQL: учебное пособие. // Благовещенск : Амурский гос. ун-т. — 2018. — С. 424.
7. *Silberschatz A.* Database System Concepts // New York : McGraw-Hill. — 2020. — С. 1344.
8. *Эдвард Сьоре*. Проектирование и реализация систем управления базами данных // М. : ДМК Пресс. — 2021. — С. 466.
9. *Осипов Д. Л.* Технологии проектирования баз данных // М. : ДМК Пресс. — 2019. — С. 498.
10. *Akhtar A.* Popularity Ranking of Database Management Systems // ArXiv. — 2023.
11. *Халимон В. И.* Базы данных // С-Пб.: СПбГТИ(ТУ). — 2017. — С. 117.
12. *Lemahieu W.* Principles of database management : the practical guide to storing, managing and analyzing big and small data // Cambridge : Cambridge University Press. — 2018. — С. 1843.
13. Encyclopedia of Database Systems / ed. by L. Ling, M. T. Özsu // New York : Springer New York. — 2018. — С. 4866.

14. *Mannino M. V.* Database Design, Application Development, and Administration // Chicago : Chicago Business Press. — 2019. — С. 873.

15. *Campbell L.* Database Reliability Engineering : Designing and Operating Resilient Database Systems // Sebastopol : O'Reilly Media. — 2018. — С. 560.

16. *Gupta G. K.* Database Management Systems // Chennai : McGraw Hill Education (India) Private Limited. — 2018. — С. 432.

17. *Petrov A.* Database Internals : A Deep Dive into How Distributed Data Systems Work // Sebastopol : O'Reilly Media. — 2019. — С. 370.

18. *Шустова Л. И.* Базы данных // М. : ИНФРА-М. — 2018. — С. 304.

19. *Wu J.* Updatable Learned Index with Precise Positions // Proceedings of the VLDB Endowment. — 2021. — Т. 14(8). — С. 1276—1288.

20. *Ding J.* ALEX: An Updatable Adaptive Learned Index // Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. — 2020. — С. 969—984.

21. *Lu B.* APEX: A High-Performance Learned Index on Persistent Memory (Extended Version) // Proceedings of the VLDB Endowment. — 2022. — Т. 15(3). — С. 597—610.

22. *Ferragina P.* The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds // PVLDB. — 2020. — Т. 13(8). — С. 1162—1175.

23. *Хайкин С.* Нейронные сети: полный курс, 2-е издание // М. : Издательский дом «Вильямс». — 2008. — С. 1103.

24. *Гудфеллоу Я.* Глубокое обучение. // М.: ДМКПресс. — 2018. — С. 652.

25. *Калистратов Т. А.* Методы и алгоритмы создания структуры нейронной сети в контексте универсальной аппроксимации функций // Вестник российских университетов. Математика. — 2014. — № 6. — С. 1845—1848.

26. OpenStreetMap [Электронный ресурс]. — Режим доступа URL: `https://www.openstreetmap.org`. — (Дата обращения: 17.05.2023).

27. Welcome to Python [Электронный ресурс]. — URL: `https://www.python.org`. — (Дата обращения: 17.05.2023).

28. PyTorch [Электронный ресурс]. — URL: https://pytorch.org/. — (Дата обращения: 17.05.2023).

29. NumPy [Электронный ресурс]. — URL: https://numpy.org/. — (Дата обращения: 17.05.2023).

30. SQLite Documentation[Электронный ресурс]. — URL: https://www.sqlite.org/docs.html. — (Дата обращения: 17.05.2023).

31. The GNU C Reference Manual [Электронный ресурс]. — Режим доступа URL: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html. — (Дата обращения: 17.05.2023).

32. The Virtual Table Mechanism Of SQLite [Электронный ресурс]. — Режим доступа URL: https://www.sqlite.org/vtab.html. — (Дата обращения: 17.05.2023).

ПРИЛОЖЕНИЕ А

Реализация курсора виртуальной таблицы

Листинг А.1 — Обработка параметров фильтрации запроса

```
int lindexBestIndex(sqlite3_vtab *tab,
                    sqlite3_index_info *pIndexInfo)
{
    if (pIndexInfo->nConstraint == 1)
    {
        if (pIndexInfo->aConstraint[0].usable)
        {
            int mode;
            switch (pIndexInfo->aConstraint[0].op)
            {
                case SQLITE_INDEX_CONSTRAINT_EQ:
                    mode = 0;
                    break;
                case SQLITE_INDEX_CONSTRAINT_GT:
                    mode = 11;
                    break;
                case SQLITE_INDEX_CONSTRAINT_GE:
                    mode = 10;
                    break;
                case SQLITE_INDEX_CONSTRAINT_LT:
                    mode = 21;
                    break;
                case SQLITE_INDEX_CONSTRAINT_LE:
                    mode = 20;
                    break;
                default:
                    return SQLITE_CONSTRAINT;
            }
            pIndexInfo->idxNum = mode;
            pIndexInfo->aConstraintUsage[0].argvIndex = 1;
            pIndexInfo->aConstraintUsage[0].omit = 1;

            return SQLITE_OK;
        }

        return SQLITE_CONSTRAINT;
    }

    if (pIndexInfo->nConstraint == 2)
    {
        for (int i = 0; i < pIndexInfo->nConstraint; i++)
```

Продолжение листинга А.1

```
42          {
43              switch (pIndexInfo->aConstraint[i].op)
44              {
45                  case SQLITE_INDEX_CONSTRAINT_LE:
46                      break;
47                  case SQLITE_INDEX_CONSTRAINT_GE:
48                      break;
49                  default:
50                      return SQLITE_CONSTRAINT;
51              }
52
53              pIndexInfo->aConstraintUsage[i].argvIndex = i + 1;
54              pIndexInfo->aConstraintUsage[i].omit = 1;
55          }
56
57          pIndexInfo->idxNum = 30;
58
59          return SQLITE_OK;
60      }
61
62      return SQLITE_CONSTRAINT;
63  }
```

Листинг А.2 — Выбор строк, удовлетворяющих фильтру

```
1   int lindexFilter(sqlite3_vtab_cursor *cur, int idxNum,
2                    const char *idxStr, int argc,
3                    sqlite3_value **argv) {
4       import_array()
5       lindex_vtab *lTab = (lindex_vtab*)cur->pVtab;
6
7       PyObject* keys = PyList_New(0);
8
9       for (int i = 0; i < argc; ++i) {
10          int64_t value = (int64_t)sqlite3_value_int64(argv[i]);
11          PyList_Append(keys, PyLong_FromLong(value));
12      }
13
14      PyObject* tuple_rowids;
15
16      if (!idxNum) {
17          PyObject* find = PyUnicode_FromString("find");
18          tuple_rowids = PyObject_CallMethodObjArgs(lTab->lindex, find,
            ↪  keys, NULL);
19      }
```

Продолжение листинга А.2

```
    else {
        PyObject* constraints = PyList_New(0);
        PyObject* noneObj = Py_None;

        for (int i = 0; i < argc; ++i) {
            PyList_Append(constraints, PyLong_FromLong(idxNum % 10));
        }

        if (idxNum / 10 != 3) {
            Py_INCREF(noneObj);
            PyList_Insert(keys, idxNum / 10 % 2, noneObj);

            Py_INCREF(noneObj);
            PyList_Insert(constraints, idxNum / 10 % 2, noneObj);
        }

        PyObject* prange= PyUnicode_FromString("predict_range");
        tuple_rowids = PyObject_CallMethodObjArgs(lTab->lindex, prange,
                keys, constraints, NULL);
    }

    PyObject* rowids;

    int tmp;
    PyArg_ParseTuple(tuple_rowids, "Oi", &rowids, &tmp);
    npy_intp size = PyArray_SIZE(rowids);
    PyArrayIterObject *iter = (PyArrayIterObject
    ↪  *)PyArray_IterNew(rowids);

    lindex_cursor *pCur = (lindex_cursor*)cur;
    pCur->rowids = rowids;
    pCur->iter = iter;

    int64_t rowid = *(int64_t *)PyArray_ITER_DATA(pCur->iter);
    sqlite3_bind_int64(lTab->stmt, 1, rowid);
    sqlite3_step(lTab->stmt);

    return SQLITE_OK;
}
```

Листинг А.3 — Реализация работы курсора

```
int lindexNext(sqlite3_vtab_cursor *cur) {
    lindex_cursor *pCur = (lindex_cursor*)cur;
    lindex_vtab *lTab = (lindex_vtab*)cur->pVtab;
```

Продолжение листинга А.3

```
4
5       PyArray_ITER_NEXT(pCur->iter);
6
7       sqlite3_reset(lTab->stmt);
8       sqlite3_clear_bindings(lTab->stmt);
9
10      int64_t rowid = *(int64_t *)PyArray_ITER_DATA(pCur->iter);
11      sqlite3_bind_int64(lTab->stmt, 1, rowid);
12      sqlite3_step(lTab->stmt);
13
14      return SQLITE_OK;
15  }
16
17  int lindexColumn(sqlite3_vtab_cursor *cur,
18                   sqlite3_context *ctx,
19                   int i)
20  {
21      lindex_vtab *lTab = (lindex_vtab*)cur->pVtab;
22      int64_t columnValue = sqlite3_column_int64(lTab->stmt, i);
23      sqlite3_result_int64(ctx, columnValue);
24
25      return SQLITE_OK;
26  }
27
28  int lindexEof(sqlite3_vtab_cursor *cur)
29  {
30      lindex_cursor *pCur = (lindex_cursor*)cur;
31      return !PyArray_ITER_NOTDONE(pCur->iter);
32  }
```

ПРИЛОЖЕНИЕ Б

Реализация модуля индекса

Листинг Б.1 — Класс индекса на основе глубоких нейронных сетей

```
import sys
import os
import pickle

import numpy as np
import matplotlib.pyplot as plt

from time import process_time_ns

from indexes.models.abs_model import AbstractModel

from utils.timer import timer

class Lindex:
    def __init__(self, model: AbstractModel):
        self.model = model
        self.model.build()

        self.trained = False

        self.max_abs_err = 0
        self.mean_abs_err = 0

    def _normalize(self, keys):
        if keys.size == 0:
            return

        min_key = self.keys[0]
        max_key = self.keys[-1]
        return (keys - min_key) / (max_key - min_key)

    def _init_for_train(self, keys: list[int], data: list[any]):
        sort_indexes = np.argsort(keys)

        self.N = len(keys)
        size = self.N
        self.keys = np.array(keys)[sort_indexes]
        self.norm_keys = self._normalize(self.keys)
        self.data = np.array(data)[sort_indexes]
        self.positions = np.arange(0, self.N) / (self.N - 1)

```

Продолжение листинга Б.1

```
    def _true_train(self):
        self.model.train(self.norm_keys, self.positions)

    def train(self, keys: list[int], data: list[any]):
        self._init_for_train(keys, data)

        self._true_train()

        self.mean_abs_err = self.model.get_mean_abs_err()
        self.max_abs_err = self.model.get_max_abs_err()

        self.trained = True

    def _predict(self, keys):
        if not self.trained:
            return None

        keys = self._normalize(keys)
        pposition = self.model(keys)
        return np.around(pposition * self.N).astype(int).reshape(-1)

    def _clarify(self, keys, positions):
        def clarify_one(key, position):
            position = max(min(position, self.N - 1), 0)

            if self.keys[position] == key:
                return position

            low = max(position - self.mean_abs_err, 0)
            high = min(position + self.mean_abs_err, self.N - 1)

            if not (self.keys[low] < key < self.keys[high]):
                low = max(position - self.max_abs_err, 0)
                high = min(position + self.max_abs_err, self.N - 1)

            while low <= high:
                mid = (low + high) // 2
                if self.keys[mid] == key:
                    return mid
                elif self.keys[mid] < key:
                    low = mid + 1
                else:
                    high = mid - 1

            return -1
```

Продолжение листинга Б.1

```
87
88          vec_clarify = np.vectorize(clarify_one)
89          positions = vec_clarify(keys, positions)
90          return positions[positions >= 0]
91
92      def find(self, keys):
93          if not self.trained or not keys:
94              return None
95
96          keys = np.array(keys)
97          positions, predict_time = self._predict(keys)
98          positions, clarify_time = self._clarify(keys, positions)
99
100         return self.data[positions]
101
102     def insert(self, key, data):
103         index = np.searchsorted(self.keys, key)
104         self.keys = np.insert(self.keys, index, key)
105         self.data = np.insert(self.data, index, data)
106         self.N = len(self.keys)
107
108         self._true_train()
109
110     def _range_binsearch(self, key, lower, upper):
111         while lower <= upper:
112             mid = (lower + upper) // 2
113             if self.keys[mid] == key:
114                 return mid, mid
115             elif self.keys[mid] < key:
116                 lower = mid + 1
117             else:
118                 upper = mid - 1
119
120         return min(lower, upper), max(lower, upper)
121
122     def _range_clarify(self, key, limit, is_lower, constraint):
123         if is_lower and np.isnan(limit):
124             return 0
125         elif np.isnan(limit):
126             return self.N - 1
127
128         key = int(key)
129         limit = int(limit)
130
131         limit = max(min(limit, self.N - 1), 0)
```

Продолжение листинга Б.1

```
132
133         lower, upper = limit, limit
134
135         if self.keys[limit] != key:
136             bin_lower = max(limit - self.mean_abs_err, 0)
137             bin_upper = min(limit + self.mean_abs_err, self.N - 1)
138
139             if not (self.keys[bin_lower] < key < self.keys[bin_upper]):
140                 bin_lower = max(limit - self.max_abs_err, 0)
141                 bin_upper = min(limit + self.max_abs_err, self.N - 1)
142
143             lower, upper = self._range_binsearch(key, bin_lower,
                ↪  bin_upper)
144
145         if lower == upper:
146             return lower - (-1) ** int(is_lower) * constraint
147
148         if is_lower:
149             return upper
150
151         return lower
152
153     def predict_range(self, keys_range, constraint):
154         if keys_range[0] is not None and keys_range[1] is not None:
155             if keys_range[0] > keys_range[1]:
156                 keys_range[0], keys_range[1] = keys_range[1],
                    ↪  keys_range[0]
157                 constraint[0], constraint[1] = constraint[1],
                    ↪  constraint[0]
158
159         keys_range = np.array([np.nan if x is None else x for x in
            ↪  keys_range])
160
161         positions_limits, _ = self._predict(keys_range)
162         positions_limits = positions_limits.astype(float)
163         positions_limits[np.isnan(keys_range)] = np.nan
164
165         lower = self._range_clarify(keys_range[0], positions_limits[0],
            ↪  True,
166                                     constraint[0])
167         upper = self._range_clarify(keys_range[1], positions_limits[1],
            ↪  False,
168                                     constraint[1])
169
170         return self.data[lower:upper + 1]
```

ПРИЛОЖЕНИЕ В