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 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: _____ _____
(подпись, дата) (И. О. Фамилия)

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка 56 с., 25 рис., 1 табл., 32 источн., 1 прил. ИНДЕКСЫ, B-ДЕРЕВЬЯ, ХЕШ-ИНДЕКСЫ, БИТОВЫЕ ИНДЕКСЫ, ОБУЧЕННЫЕ ИНДЕКСЫ, БАЗЫ ДАННЫХ, СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

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].

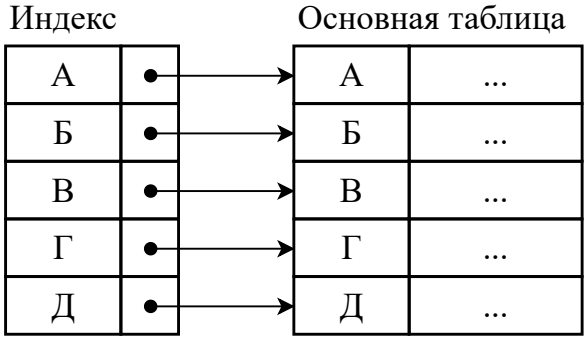

Рисунок 1.1 – Плотный индекс

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].

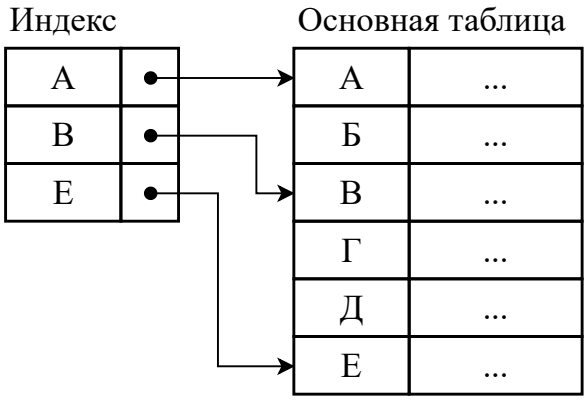

Рисунок 1.2 – Разреженный индекс

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).

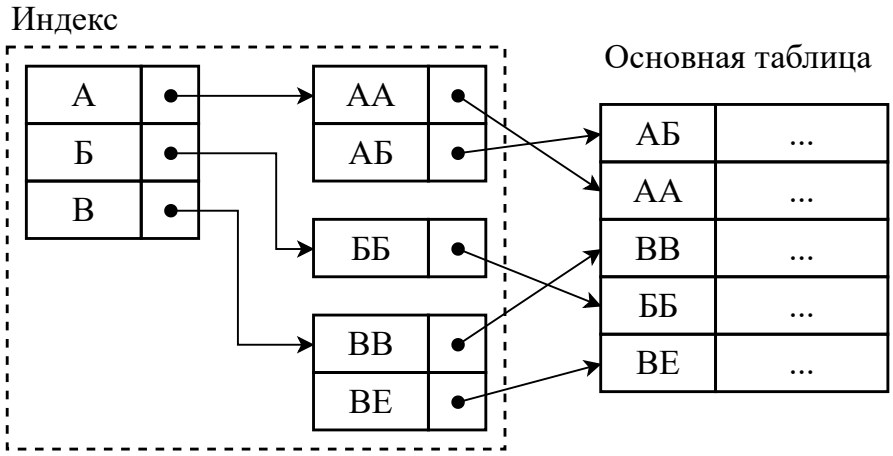

Рисунок 1.3 – Многоуровневый индекс

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 формулой (1.2):

$$\begin{gathered} K_{i-1} < X < K_i, \text{ если } 1 < i < q, \\ X < K_i, \text{ если } i = 1, \\ K_{i-1} < X, \text{ если } i = q; \end{gathered} \tag{1.2}$$

— 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.

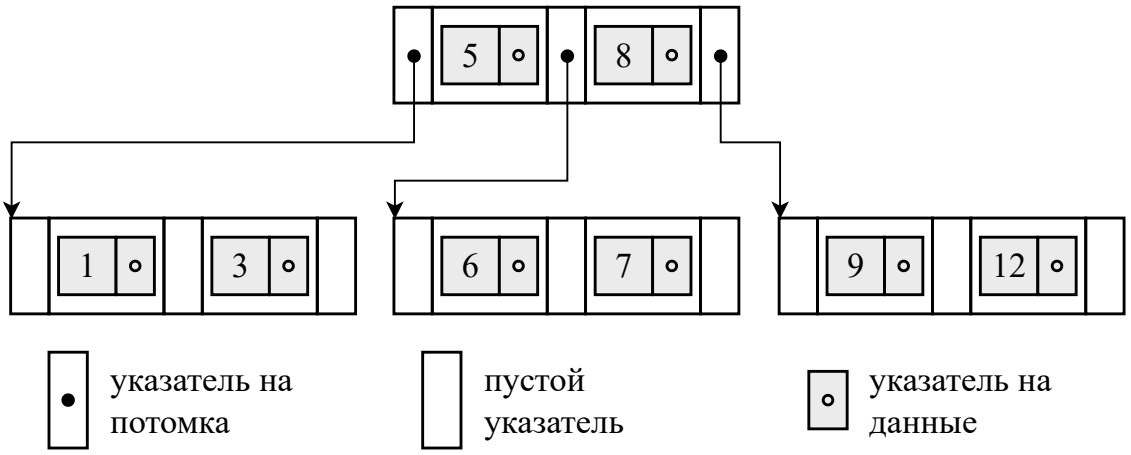

Рисунок 1.4 – Пример B-дерева

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.

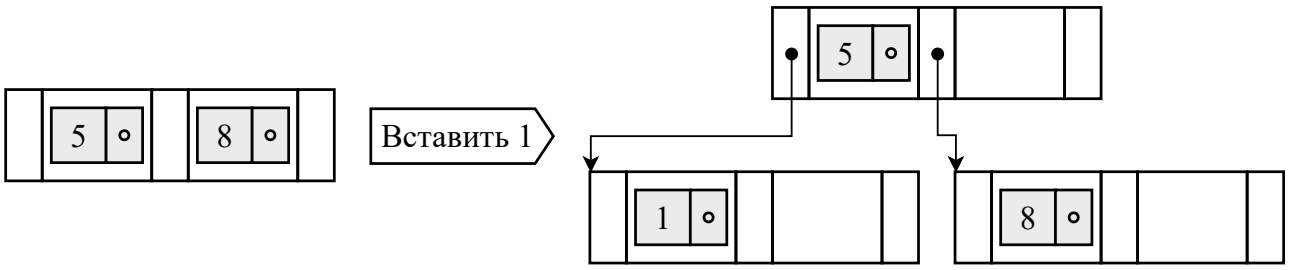

Рисунок 1.5 – Пример вставки в B-дерево при заполненном корне

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

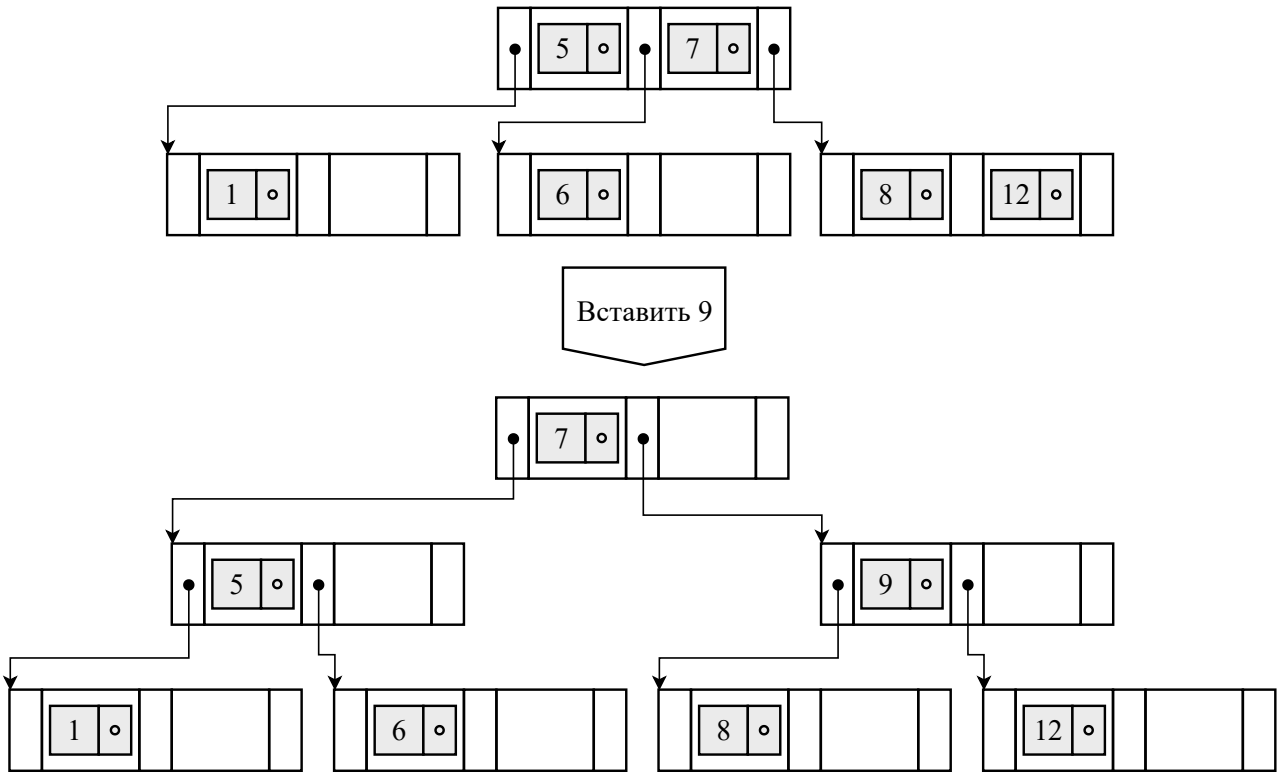

Рисунок 1.6 – Пример вставки в B-дерево при последовательном заполнении узлов разных уровней

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 (1.3):

$$(P_1, K_1, P_2, K_2, \ldots, K_{q-1}, P_q), \tag{1.3}$$

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 формулой (1.2), для B$^+$-деревьев модернизируется в формулу (1.4):

$$\begin{gathered} K_{i-1} < X \leqslant K_i, \text{ если } 1 < i < q, \\ X \leqslant K_i, \text{ если } i = 1, \\ K_{i-1} < X, \text{ если } i = q. \end{gathered} \tag{1.4}$$

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.

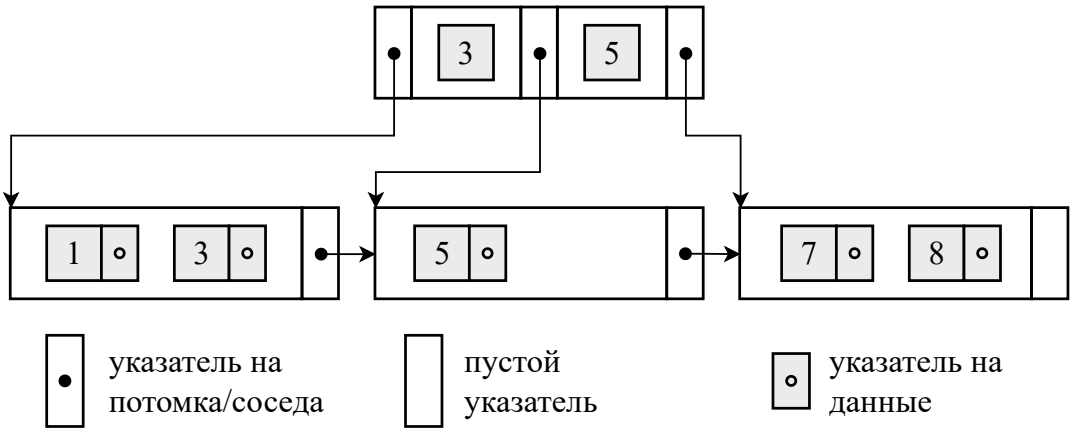

Рисунок 1.7 – Пример B^+-дерева

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 метод открытой адресации при повторной коллизии, возможно включение и третьей хеш-функции.

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 пере-

полнения, приведен на рисунке 1.8.

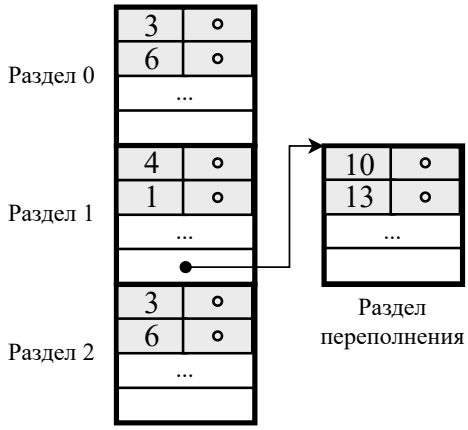

Рисунок 1.8 – Пример структуры хеш-индекса с разрешением коллизий методом цепочек переполнения

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.

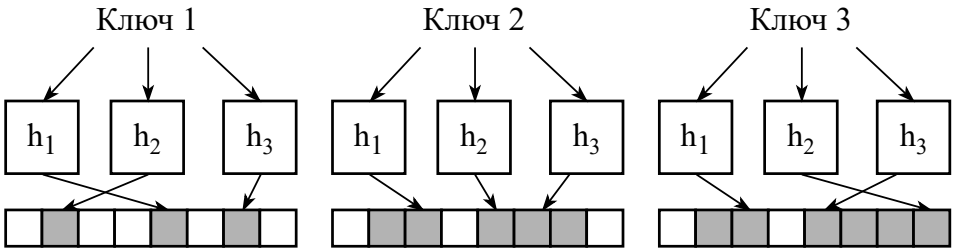

Рисунок 1.9 – Пример построения индекса на основе фильтра Блума

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), при этом возможно достижение временной сложности поиска $O(1)$ [5].

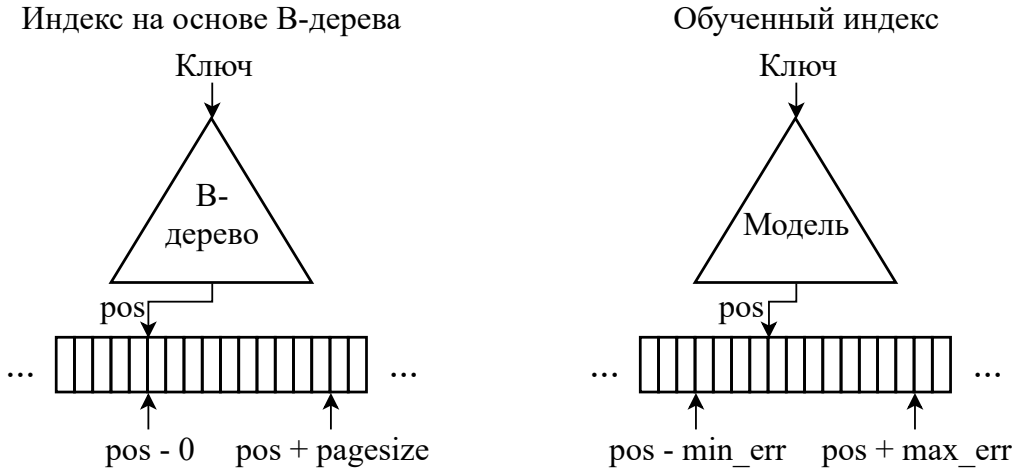

Рисунок 1.10 – Сравнение индексов на основе B-деревьев и обученных индексов

Основываясь на этой идее авторы [5] пришли к наблюдению, что модель

предсказывающая положение заданного ключа внутри отсортированного массива, аппроксимирует функцию распределения, что позволяет находить искомое положение с помощью формулы (1.5):

$$p = F(K) \cdot N, \tag{1.5}$$

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.

Описанные выше индексы были названы обученными, однако по ряду причин [5] время поиска с помощью описанного выше «наивного» подхода превзошло время поиска с помощью B-деревьев, поэтому была предложена 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. В данную иерархию возможно включение различных моделей: например, на верхнем уровне использовать нейронные сети, а на нижних простые линейные регрессионные модели или даже B-деревья.

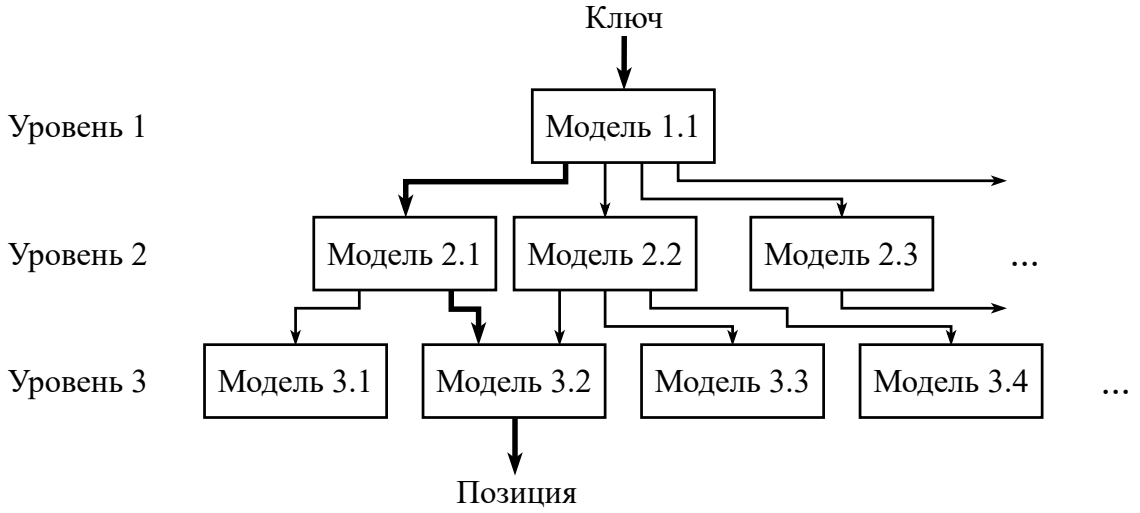

Рисунок 1.11 – Рекурсивная модель индекса

С учетом иерархии временная сложность поиска обученных индексов составляет $O(\log N)$, однако такая модель по сравнению с B-деревьями каждо-

му узлу соотносит большее число элементов, что дает меньшее время поиска. 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 (1.6):

$$h(K) = F(K) \cdot M, \tag{1.6}$$

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).

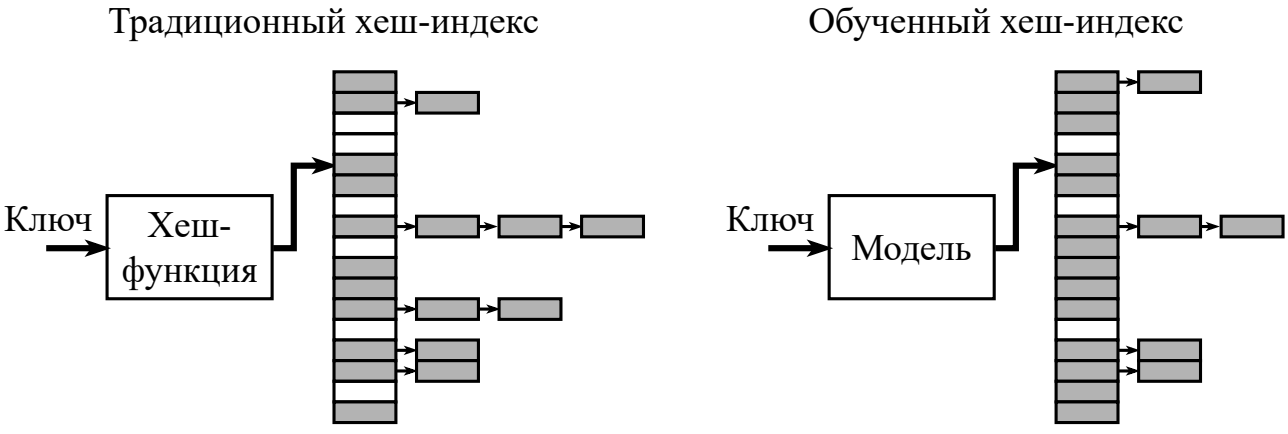

Рисунок 1.12 – Сравнение традиционных и обученных хеш-индексов

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].

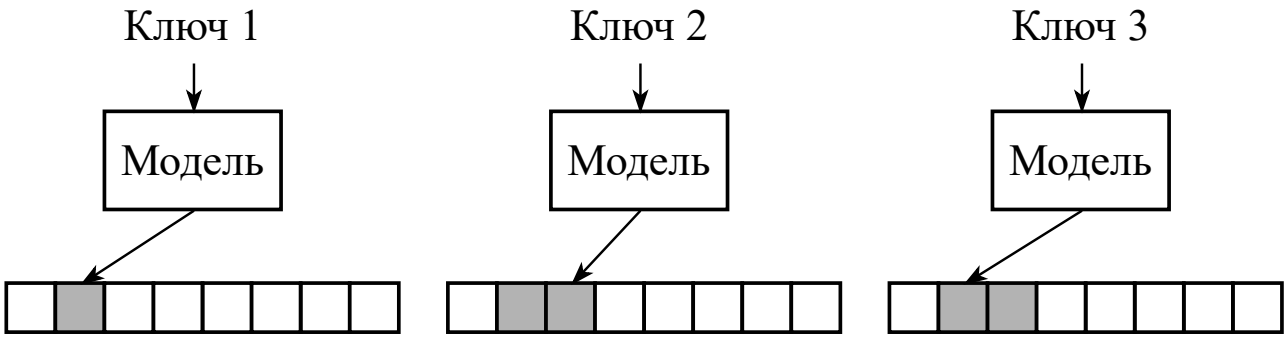

Рисунок 1.13 – Пример построения обученного индекса проверки существования

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		**B-дерево**	**Хеш-таблица**	**Фильтр Блума**	**Обученные индексы**
Временная сложность	поиска	O(log N)	O(1) / O(N)	O(k)	O(1)
	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, искус-

ственная 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.

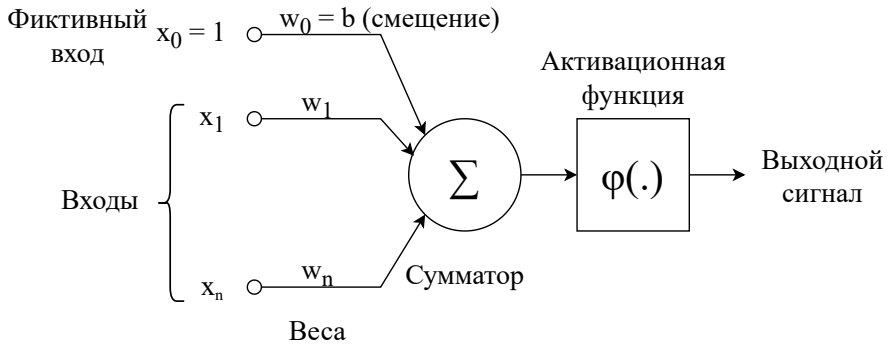

Рисунок 1.14 – Модель нейрона

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.

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

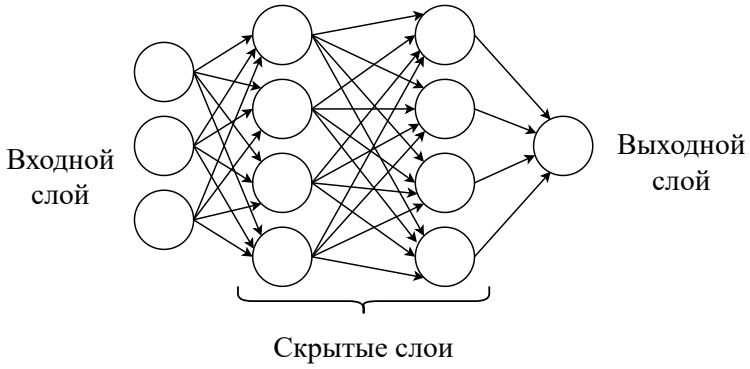

Рисунок 1.15 – Глубокая нейронная сеть

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, аппрокимирующая 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. Выбор архитектур нейронных сетей, используемых в данной работе, подробнее в рассматривается в разделе 2.

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, требующейся для обучения модели;
— 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.

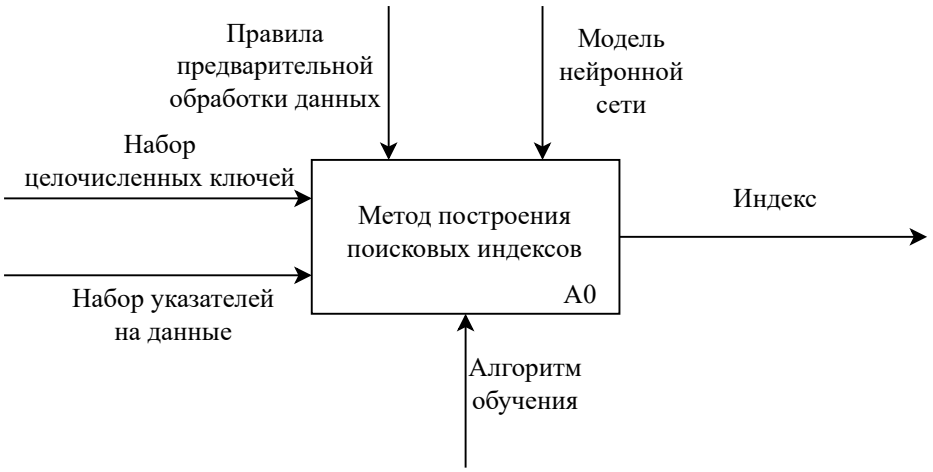

Рисунок 1.16 – Постановка задачи

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. сохранять параметры обученной модели для каждой таблицы с целью возможности выполнять запросы поиска без переобучения;
5. обеспечивать поиск записи (диапазона записей) таблицы по ключу (диапазону ключей) с использованием обученной модели;
6. 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

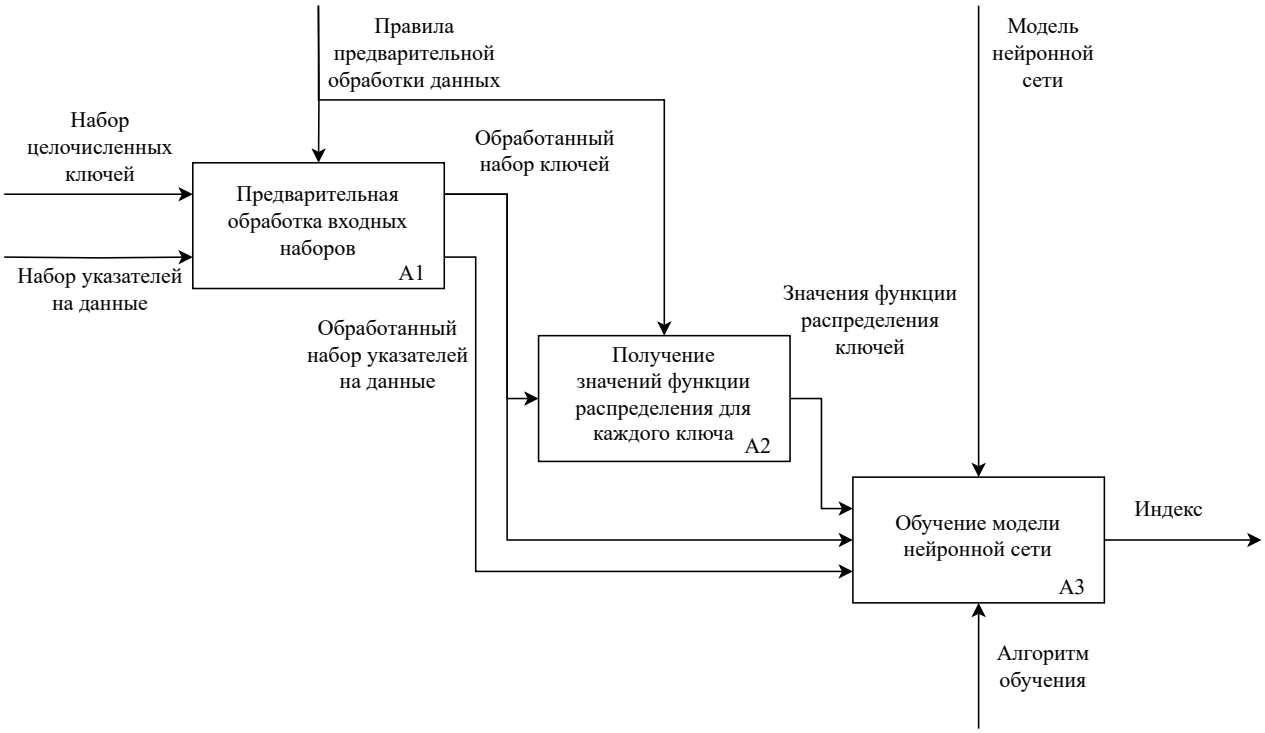

Рисунок 2.1 – Функциональная схема метода построения индекса

этапом выделено получение значений функций распределения для каждого ключа, относящееся к предварительной обработке, но представляющее собой ее 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 соответвующих 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, предствляющая 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.

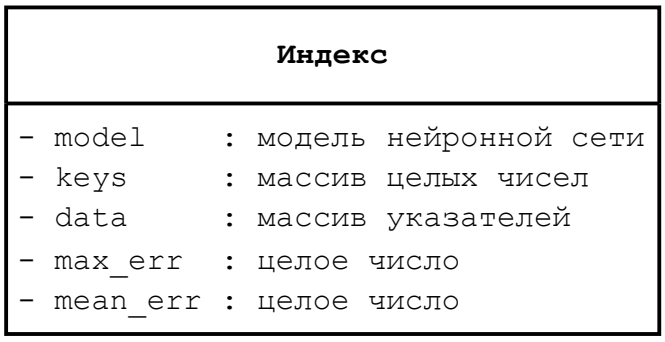

Рисунок 2.2 – Индекс как структура данных

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.

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 дапазон $[0, 1]$, для чего используется метод минимакс-нормализации, при котором нормализованное значение вычисляется по формуле:

$$x_{\text{норм}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}, \tag{2.1}$$

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

x — натуральное значение ключа;

$x_{\min}$, $x_{\max}$ — минимальное и максимальное возможное значение ключа в

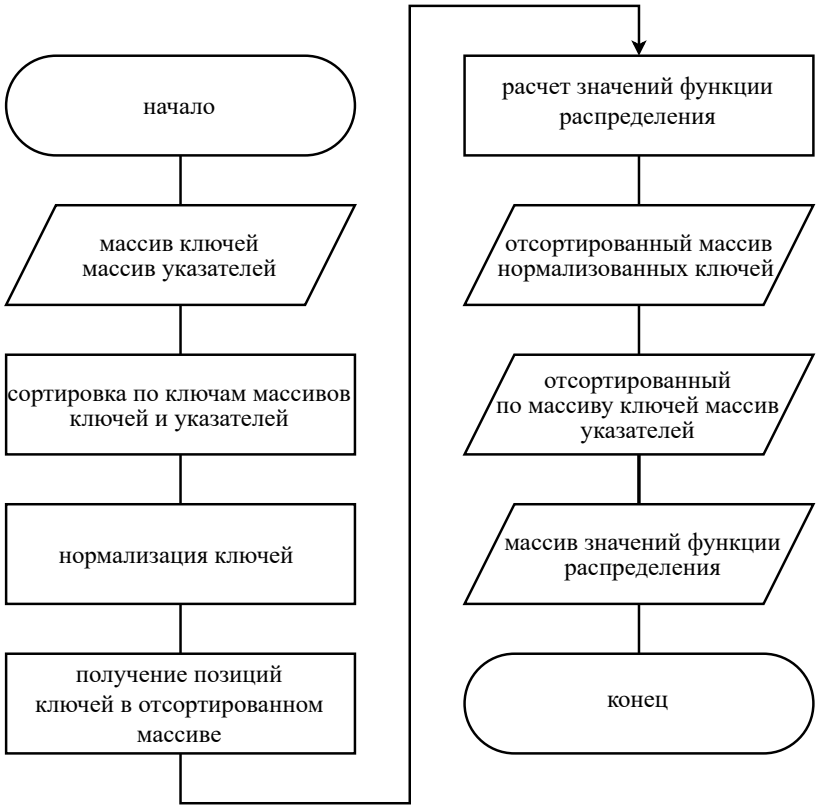

Рисунок 2.3 – Схема алгоритма предварительной обработки данных

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.

```
1 begin
2     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
3     keys ← (keys − keys[0]) / (keys[N−1] − keys[0]);
4     positions ← [0, 1, ..., N − 1];
5     cdf ← positions / (N − 1);
6     return keys, data, cdf;
7 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 фунцкции 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.

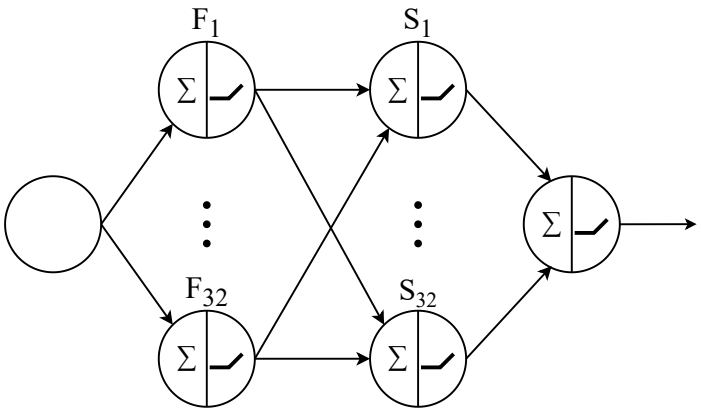

Рисунок 2.4 – Полносвязная нейронная сеть с двумя скрытыми слоями

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.

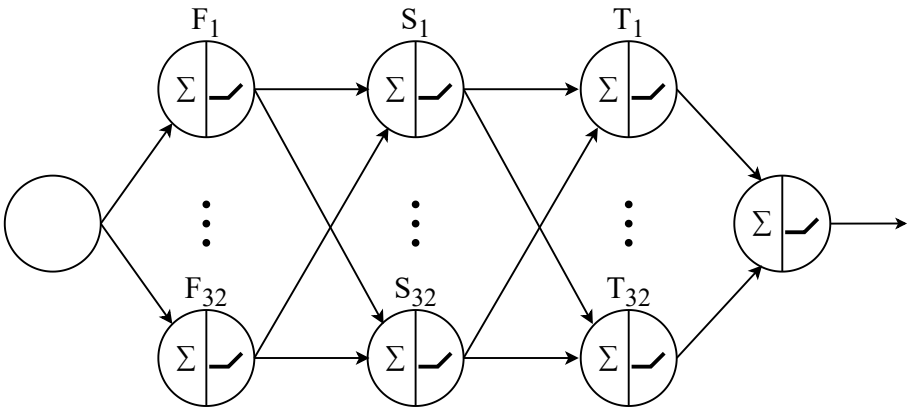

Рисунок 2.5 – Полносвязная нейронная сеть с тремя скрытыми слоями

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 единичиному ключу и по диапазону ключей, то есть принимать диапазоны, представленные в виде операций сравнения `=, >, <, >=, <=`, а `a <= key <= b, ...`, где key — ключ, a, b — границы диапазона.

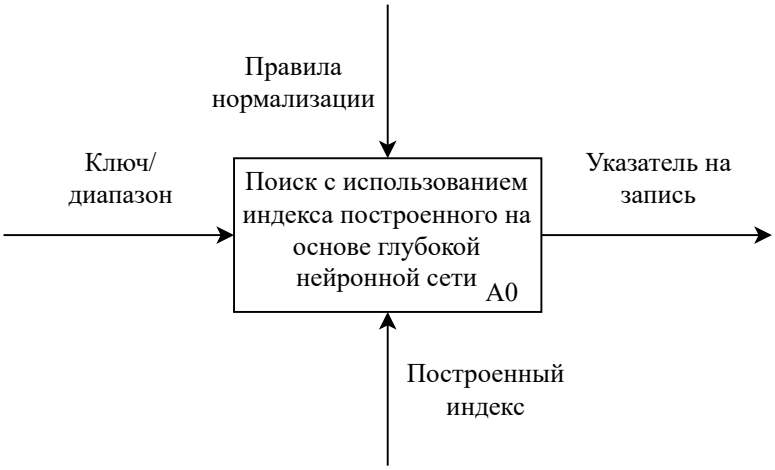

Рисунок 2.6 – Функциональная схема нулевого уровня поиска

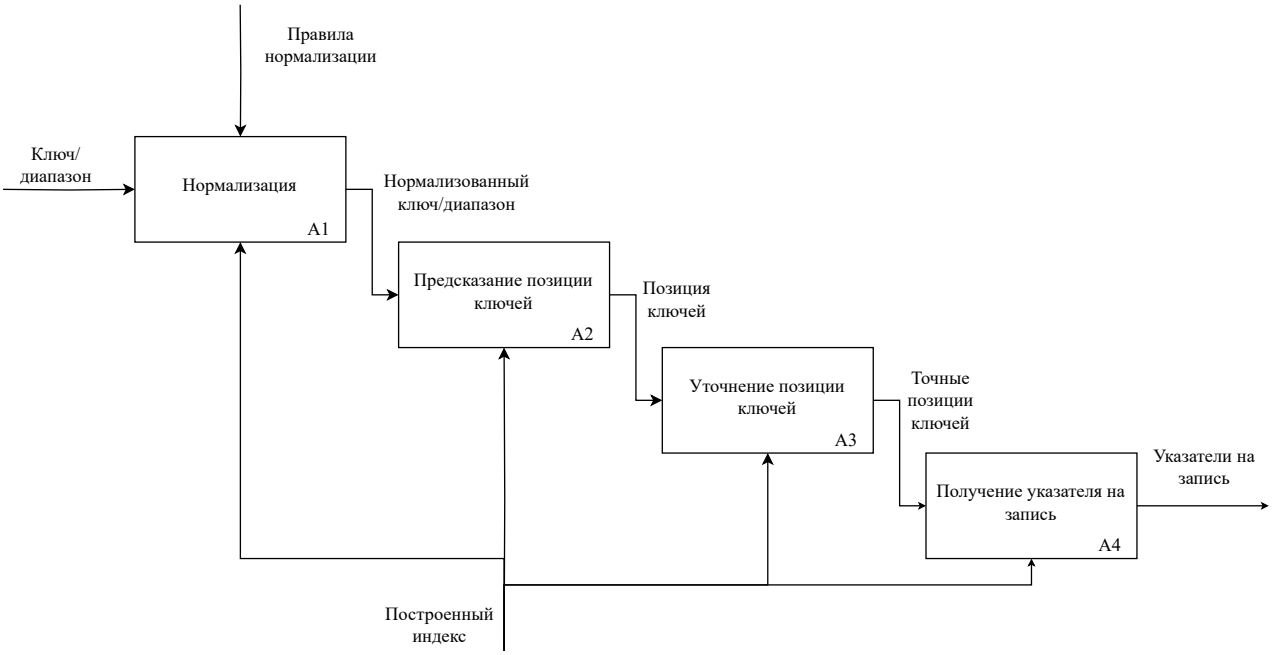

Рисунок 2.7 – Функциональная схема первого уровня поиска

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 истинно-

го 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     else if is_lower then
13         return upper;
14     else
15         return lower;
16     end
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
```

Вставка 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 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 2.8

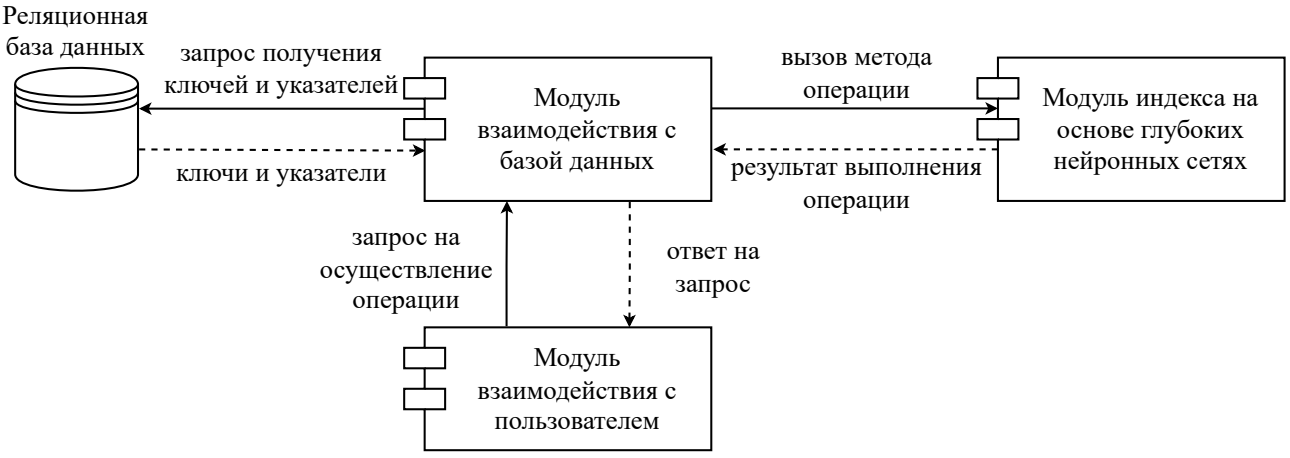

Рисунок 2.8 – Структура программного обеспечения

Как видно из рисунка, программное обеспечение должно включать три модуля: модуль взаимодействия с пользователем, взаимодействия с реляционной 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 $R[a, b]$, фукнция 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 2.9.

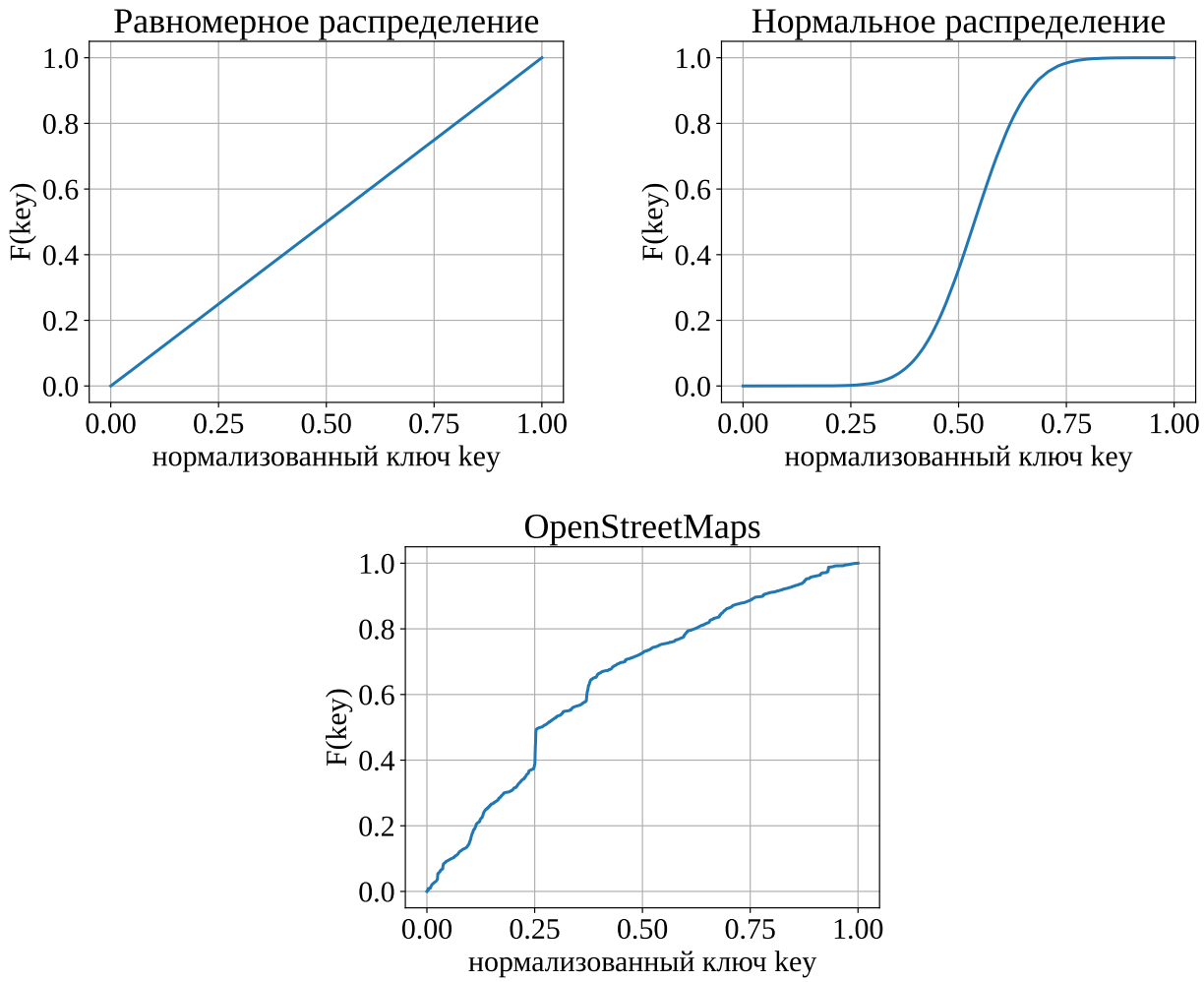

Рисунок 2.9 – Графики функций распределения ключей из наборов данных для обучения и тестирования

3 Технологическая часть

3.1 Выбор средств программной реализации

Для реализации метода построения индекса в реляционной базе данных на основе глубоких нейроных сетей в качестве языка программирования выбран Python 3.10 [27], так как предоставляет широкий выбор библиотек для глубокого обучения и визуализации его результатов. В качестве библиотеки глубокого обучения выбран TensorFlow 2.11.0 [28] и работающий поверх нее высокоуровневый программный интерфейс Keras 2.11.0 [29]. Для работы с массивами данных выбрана библиотека numpy [30].

В качестве реляционной системы управления базами данных выбрана SQLite [31], предоставляющая программный интерфейс виртуальных таблиц, позволяющих релизовать пользовательский поисковый индекс. Виртуальные таблицы являются одним из видов расширений SQLite, программный интерфейс которых предоставляется на языке C [32], 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 [**vtable**] — это объект базы данных, представляющий с точки зрения инструкций SQL обычную таблицу или представле-

ние, но обрабатывающий запросы посредством вызова функций программного интерфейса, которые реализуются пользователем.

Виртальные таблицы являются расширением SQLite, регистрация которых происходит с использованием макросов и фукнции инициализации, представленных на линстинге 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 вирутальной таблицы должен быть зарегистрирован модуль, описывающийся структурой `sqlite3_module`, с помощью функции регистрации `sqlite3_create_module`, подробное описание которых представлено на листинге 3.2

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, вызывющийся при выполнении любого запроса к таблице, который является первым при повторном подключении к базе данных;
  – `xDestroy` — удаление вирутальной таблицы при выполнеивыполнении запроса, представленного на листинге 3.4;
  – `xDisconnect` — удаление поключения к виртуальной таблице.

Листинг 3.2 – Структура и функции для 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,
                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*/
);
```

Листинг 3.3 – Запрос на создание виртуальной таблицы

```
CREATE VIRTUAL TABLE <имя_таблицы> USING <имя_модуля>(arg1, ...);
```

Листинг 3.4 – Запрос на удаление виртуальной таблицы

```
DROP TABLE <имя_таблицы>;
```

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 вирутальной таблицы

```
1 struct sqlite3_vtab {
2     const sqlite3_module *pModule; /* модуль таблицы */
3     int nRef; /* число ссылок, инициализирующееся ядром SQLite */
4     char *zErrMsg; /* для передачи сообщений об ошибках ядру */
5 };
```

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

Листинг 3.6 – Структура курсора

```
1 struct sqlite3_vtab_cursor {
2   sqlite3_vtab *pVtab; /* указатель на виртуальную таблицу */
3 };
```

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_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 программиный интерфейс приведен на листинге 3.9.

Обработка параметров фильтрации приведена на листинге 3.10. Получение массива подходящих строк и проход для их вывода приведены на листингах 3.11, 3.12 соответственно.

3.2.4 Реализация индекса

...

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 3.13.

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));

    if (!vtab)
        return SQLITE_NOMEM;

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

    char *sql_template = get_create_table_query_by_args(argc,
       argv);

    const char *vTableName = argv[2];
    const char *rTableName = sqlite3_mprintf("r%s", vTableName);

    char *vSqlQuery = sqlite3_mprintf(sql_template, vTableName);
    char *rSqlQuery = sqlite3_mprintf(sql_template, rTableName);

    int rc = sqlite3_declare_vtab(db, vSqlQuery);

    if (!rc)
        rc = sqlite3_exec(db, rSqlQuery, NULL, NULL, errMsg);

    sqlite3_free(sql_template);
    sqlite3_free(vSqlQuery);
    sqlite3_free(rSqlQuery);

    rc = initPythonIndex(db, rTableName, "fcnn2", vtab);

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

    return rc;
}
```

Листинг 3.9 – Инициализация индекса

```
int initPythonIndex(sqlite3 *db,
                    const char *const tableName,
                    const char *const modelName,
                    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));
    PyObject* builder = PyObject_CallObject(builderClassName,
       pyModelName);
    PyObject* lindex = PyObject_CallMethod(builder, "build", NULL
       );
    PyObject* keys = PyList_New(0);
    PyObject* rows = PyList_New(0);

    int i = 0;
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        int key = sqlite3_column_int(stmt, 1);
        int64_t rowid = sqlite3_column_int64(stmt, 0);

        PyList_Append(keys, PyLong_FromLong(key));
        PyList_Append(rows, PyLong_FromLong(rowid));

        i++;
    }

    if (i) {
        PyObject* train = PyUnicode_FromString("train");
        PyObject* check = PyObject_CallMethodObjArgs(lindex,
           train, keys, rows, NULL);
        Py_DECREF(check);
        Py_DECREF(train);
    }

    vTab->lindex = lindex;

    Py_DECREF(keys);
    /* ... */
    sqlite3_finalize(stmt);

    return SQLITE_OK;
}
```

Листинг 3.10 – Обработка параметров фильтрации запроса

```
int lindexBestIndex(sqlite3_vtab *tab,
                    sqlite3_index_info *pIndexInfo)
{
    if (pIndexInfo->nConstraint > 0) {
        for (int i = 0; i < pIndexInfo->nConstraint; i++) {
            if (pIndexInfo->aConstraint[i].usable
                && pIndexInfo->aConstraint[i].op ==
                   SQLITE_INDEX_CONSTRAINT_EQ) {
                pIndexInfo->aConstraintUsage[i].argvIndex = i+1;
                pIndexInfo->aConstraintUsage[i].omit = 1;
            }
        }
    }
    return SQLITE_OK;
}
```

Листинг 3.11 – Выбор строк, удовлетворяющих фильтру

```
int lindexFilter(sqlite3_vtab_cursor *cur,
                 int idxNum,
                 const char *idxStr,
                 int argc,
                 sqlite3_value **argv)
{
    import_array()
    lindex_vtab *lTab = (lindex_vtab*)cur->pVtab;

    PyObject* keys = PyList_New(0);
    PyList_Append(keys, PyLong_FromLong(sqlite3_value_int(argv[i
       ])));

    PyObject* find = PyUnicode_FromString("find");
    PyObject* rowids = PyObject_CallMethodObjArgs(lTab->lindex,
       find, keys, NULL);
    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 3.12 – Реализация работы курсора

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

    sqlite3_reset(lTab->stmt);
    sqlite3_clear_bindings(lTab->stmt);
    int64_t rowid = *(int64_t *)PyArray_ITER_DATA(pCur->iter);
    sqlite3_bind_int64(lTab->stmt, 1, rowid);
    sqlite3_step(lTab->stmt);

    PyArray_ITER_NEXT(pCur->iter);

    return SQLITE_OK;
}

int lindexColumn(sqlite3_vtab_cursor *cur,
                 sqlite3_context *ctx,
                 int i)
{
    lindex_vtab *lTab = (lindex_vtab*)cur->pVtab;
    int columnValue = sqlite3_column_int(lTab->stmt, i);
    sqlite3_result_int(ctx, columnValue);

    return SQLITE_OK;
}

int lindexEof(sqlite3_vtab_cursor *cur)
{
    lindex_cursor *pCur = (lindex_cursor*)cur;
    return !PyArray_ITER_NOTDONE(pCur->iter);
}
```

Листинг 3.13 – 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 3.14. Также для штатной работы программного обеспечения требуется прописать путь к модулям, реализованным на языке Python, что приведено на том же листинге.

Листинг 3.14 – Подготовка для работы модулей Python

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

3.4 Взаимодействие с программным обеспечением

Взаимодействие с программным обеспечением происходит через командную строку sqlite3. Пример работы представлен на листинге **??**.

3.5 Результаты тестирования

???

Листинг 3.15 – Пример работы программного обеспечения

```
$ sqlite3 test.db
sqlite> .load ./lindex
sqlite> create table maps(keys INTEGER);
sqlite> .mode csv
sqlite> .import osm100000.csv maps;
sqlite> create virtual table virtmaps using lindex(0, "FCNN2");
Epoch 1/30
1001/1001 [====================] - 1s 768us/step - loss: 0.0011
...
sqlite> select * from virtmaps where keys = 232131750;
keys
232131750
```

4 Исследовательская часть

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Коптенок Е. В.* Исследование способов ускорения поисковых запросов в базах данных / Е. В. Коптенок, М. А. Подвесовская, Т. М. Хвостенко, А. В. Кузин // Вестник образовательного консорциума среднерусский университет. Информационные технологии. — 2019. — № 1(13). — С. 24—27.
2. *Reinsel D.*, *Gantz J.*, *Rydning J.* The Digitization of the World From Edge to Core // IDC White Paper. — 2018.
3. *Носова Т. Н.*, *Калугина О. Б.* 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 / T. Kraska, A. Beutel, E. H. Chi [и др.] // Proceedings of the 2018 International Conference on Management of Data. — 2018. — С. 489—504.
6. *Григорьев Ю. А.*, *Плутенко А. Д.*, *Плужникова О. Ю.* Реляционные базы данных и системы NoSQL: учебное пособие. // Благовещенск : Амурский гос. ун-т. — 2018. — С. 424.
7. *Silberschatz A.*, *Korth H. F.*, *Sudarshan S.* 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 / В. И. Халимон, Г. А. Мамаева, А. Ю. Рогов, В. Н. Чепикова // С-Пб.: СПбГТИ(ТУ). — 2017. — С. 117.

12. *Lemahieu W.*, *Broucke S. vanden*, *Baesens B.* 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.*, *Major C.* 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.*, *Тараканов О. В.* Базы данных // М. : ИНФРА-М. — 2018. — С. 304.

19. *Wu J.* Updatable Learned Index with Precise Positions / J. Wu, Y. Zhang, S. Chen [и др.] // Proceedings of the VLDB Endowment. — 2021. — Т. 14(8). — С. 1276—1288.

20. *Ding J.* ALEX: An Updatable Adaptive Learned Index / J. Ding, U. F. Minhas, H. Zhang [и др.] // 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) / B. Lu, J. Ding, E. Lo [и др.] // Proceedings of the VLDB Endowment. — 2022. — Т. 15(3). — С. 597—610.

22. *Ferragina P.*, *Vinciguerra G.* 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.*, *Бенджио И.*, *Курвилль А.* 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. TensorFlow [Электронный ресурс]. — URL: `https://www.tensorflow.org/`. — (Дата обращения: 17.05.2023).

29. Keras: Deep Learning for humans [Электронный ресурс]. — URL: `https://keras.io/`. — (Дата обращения: 17.05.2023).

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

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

32. 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).