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 НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

НА ТЕМУ:

«Классификация методов построения индексов в базах данных»

Студент: ИУ7-73Б _____ М. Д. Маслова
(группа) (подпись, дата) (И. О. Фамилия)

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

2022 г.

РЕФЕРАТ

Расчетно-пояснительная записка 26 с., 16 рис., 0 табл., 19 источн., 1 прил.

ИНДЕКСЫ, B-ДЕРЕВЬЯ, ХЕШ-ИНДЕКСЫ, БИТОВЫЕ ИНДЕКСЫ, ОБУЧЕННЫЕ ИНДЕКСЫ, БАЗЫ ДАННЫХ, СИСТЕМЫ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ

Объектом исследования является построение индексов в базах данных.

Цель работы — классификация методов построения индексов в базах данных.

В разделе 1 рассмотрено понятие индекса в базах данных и его основные свойства, а также описаны типы индексов.

В разделе 2 проведен обзор методов построения индексов на основе B-деревьев, хеш-таблиц и битовых карт, а также соответствующих обученных индексов.

В разделе 3 приведены критерии оценки качества описанных методов и проведено сравнение по этим критериям.

что-то про результат

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На протяжении последнего десятилетия происходит автоматизация все большего числа сфер человеческой деятельности [1]. Это приводит к тому, что с каждым годом производится все больше данных. Так, по исследованию компании IDC (International Data Corporation), занимающейся изучением мирового рынка 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 используют такие структуры, как деревья поиска, хеш-таблицы и битовые карты [4]. На основе данных методов проводятся исследования по разработке новых для уменьшения времени поиска и затрат на перестроение индекса при изменении данных, а также сокращения дополнительно используемой памяти. Одно из таких исследований [5] было проведено в 2018 году, авторы которого, опираясь на идею, что обычные индексы не учитывают распределение данных, предложили новый вид индексов, основанный на машинном обучении, и назвали их обученные индексы (learned indexes). За последние пять лет было проведено множество исследований [6–9] по совершенствованию обученных индексов в плане поддержки операций и улучшению производительности, поэтому в данной работе в сравнение к базовым приводятся методы построения обученных индексов.

Целью данной работы является ***классификация методов построения индексов в базах данных***.

Для достижения поставленной цели требуется решить следующие задачи:

— провести анализ предметной области: дать основные определение, описать свойства индексов и их типы;
— описать методы построения индексов в базах данных;
— предложить и обосновать критерии оценки качества описанных методов и сравнить методы по предложенным критериям оценки.

1 Анализ предметной области

1.1 Основные определения

Индекс — это некоторая структура, обеспечивающая быстрый поиск записей в базе данных [10]. Индекс определяет соответствие значения атрибута или набора атрибутов — *ключа поиска* — конкретной записи с местоположением этой записи [11]. Это соответствие организуется с помощью индексных записей. Каждая из них соответствует записи в *индексируемой таблице* — таблице, по которой строится индекс — и содержит два поля: идентификатор записи или указатель на нее, а также значение индексированного поля в этой записи [12].

Индексы могут использоваться для поиска по конкретному значению или диапазону значений, а также для проверки существования элемента в таблице, однако обеспечение уменьшения времени доступа к записям в общем случае достигается за счет [11]:

— 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 таблицы, что необходимо учитывать[13].

Таким образом, индекс может описываться [11]:

— *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*, занимаемая индексной структурой.

1.2 Типы индексов

Индексы могут быть:

— 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 [13].

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 [11].

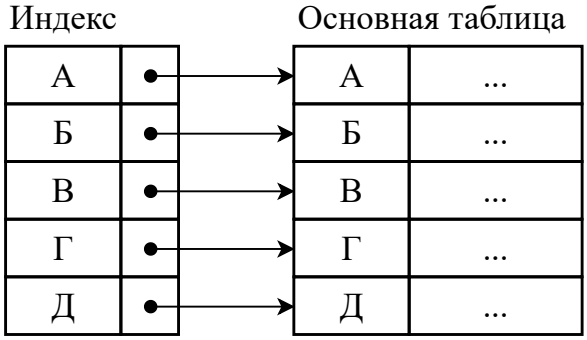

Рисунок 1.1 – Плотный индекс

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 [11].

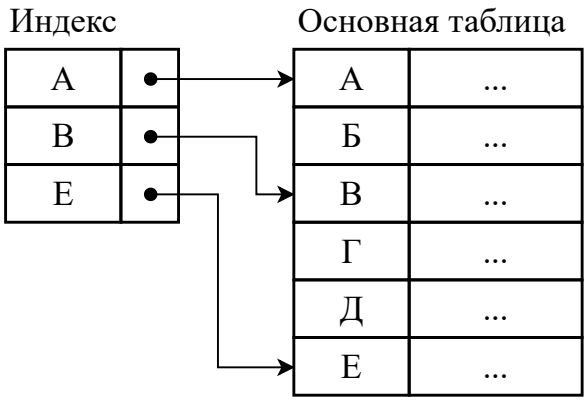

Рисунок 1.2 – Разреженный индекс

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 [11].

Одноуровневые индексы ссылаются на данные таблице, индексы же верхнего уровня *многоуровневой* структуры ссылают на индексы нижестоящего уровня [11] (рисунок 1.3).

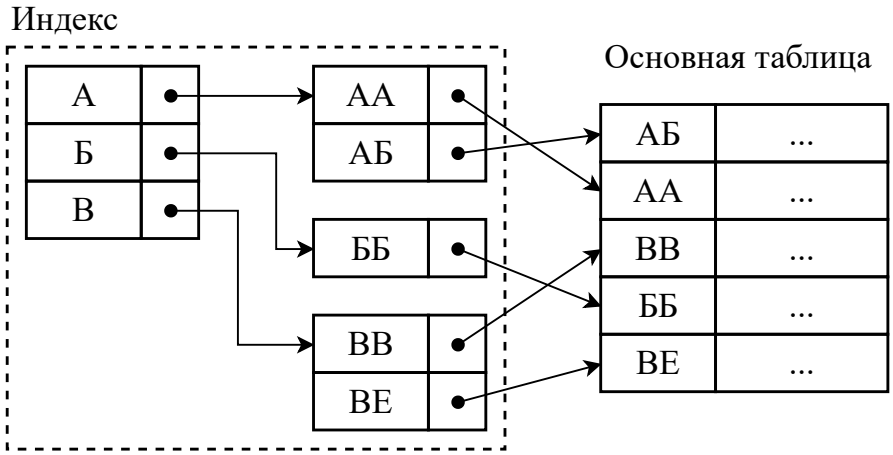

Рисунок 1.3 – Многоуровневый индекс

2 Описание существующих методов построения индексов

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.

2.1 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 [14].

В данном случае *сбалансированными деревьями* называют такие деревья, что длины любых двух путей от корня до листьев одинаковы [15]. *Сильноветвящимися* же являются деревья, каждый узел которых ссылается на большое число потомков [16]. Эти условия обеспечивают минимальную высоту дерева для быстрого поиска и свободное пространство в узла 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 [14].

2.1.1 B-деревья

B-дерево — это сбалансированная, сильноветвящаяся древовидная, работающая с отсортированными значениями структура данных, операции вставки и удаления в которой не изменяют ее свойств [17]. Все свойства данной структуры поддерживаются путем сохранения в узлах положений для включения новых элементов [18]. Это осуществляется за счет свойств узлов, которые определяются порядком B-дерева m.

2.1.1.1 Описание структуры

B-деревом порядка m [14, 18] называется дерево поиска, такое что:

— каждый узел имеет формат, описывающийся формулой (2.1):

$$(P_1, (K_1, Pr_1), P_2, (K_2, Pr_2), \ldots, (K_{q-1}, Pr_{q-1}), P_q), \tag{2.1}$$

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 (2.2):

$$\begin{gathered} K_{i-1} < X < K_i, \text{ если } 1 < i < q, \\ X < K_i, \text{ если } i = 1, \\ K_{i-1} < X, \text{ если } i = q; \end{gathered} \tag{2.2}$$

— 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 [14].

Пример B-дерева представлен на рисунке 2.1.

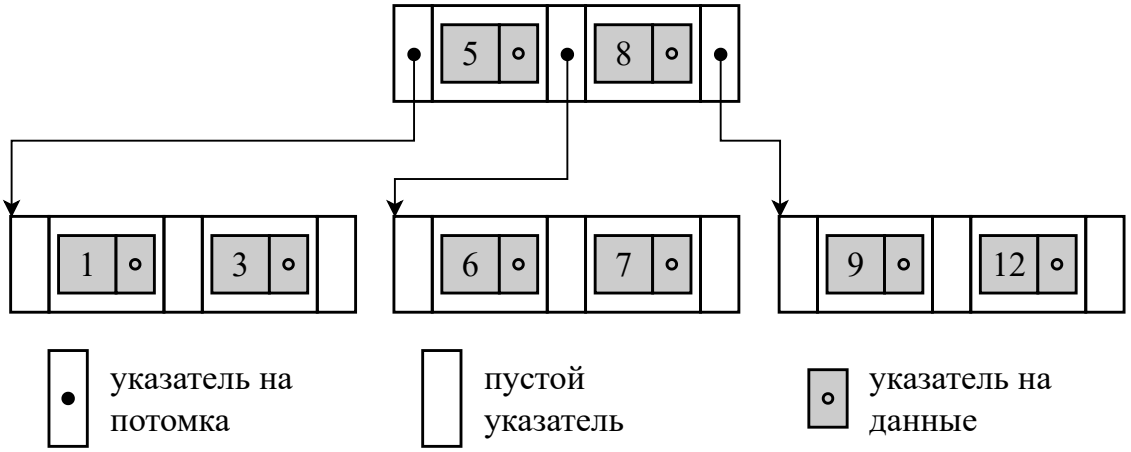

Рисунок 2.1 – Пример B-дерева

2.1.1.2 Принцип построения

Построение B-дерева [19] 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 2.2). Два созданных узла становятся потомками нового корня.

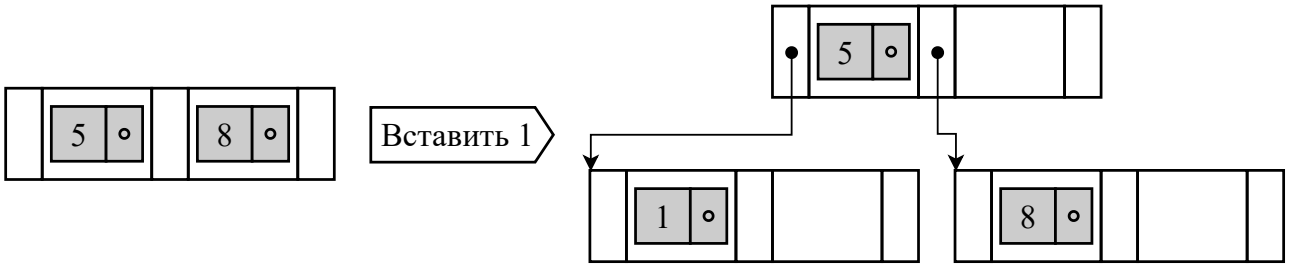

Рисунок 2.2 – Пример вставки в B-дерево при заполненном корне

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 2.3). Фактически дерево строится последовательным выполнением операций вставки.

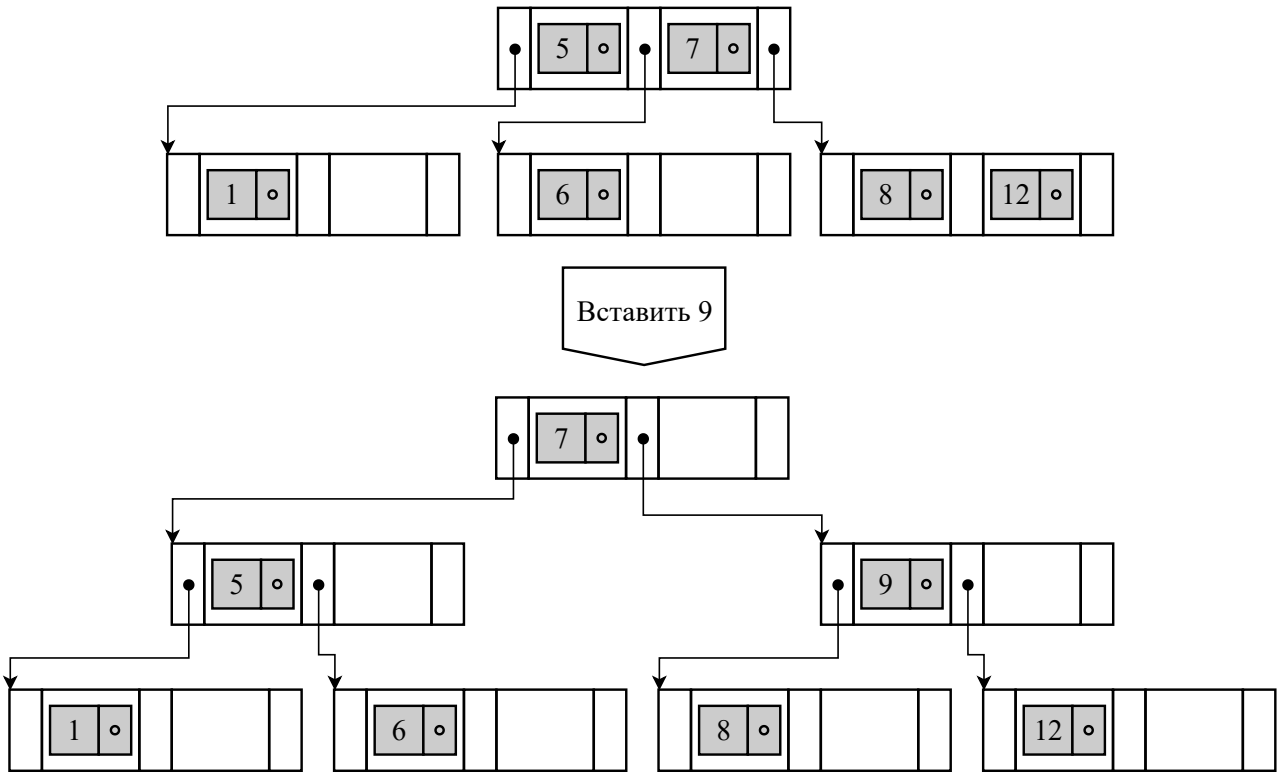

Рисунок 2.3 – Пример вставки в B-дерево при последовательном заполнении узлов разных уровней

2.1.1.3 Операции поиска, вставки и удаления элементов

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 соответвующему указателю Pr_i. Если значение не найдено, происходит переход к поддереву по указателю P_i, соответвующему 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 [14].

Выполнение операции вставки в B-дерево описано выше при описании алгоритма построения.

Удаление значений основано на той же идее. Нужное значение удаляется из узла, в котором оно находится, и если количество значений в узле становится меньше половины максимально возможного количества значений, то узел объединяется с соседними узлами, что также может распространяться вплоть до корня. ***расписать бы по-хорошему + чем не угодили b-деревья***

2.1.2 B$^+$-деревья

Структура B$^+$-дерева аналогична структуре B-дерева за исключением двух моментов. Во-первых, внутренние узлы не содержат указателей на записи, в них хранятся только значения ключей, то есть внутренние узлы имеют формат, описывающийся формулой (2.3):

$$(P_1, K_1, P_2, K_2, \ldots, K_{q-1}, P_q), \tag{2.3}$$

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 (2.2), для B$^+$-деревьев модернизируется в формулу (2.4):

$$\begin{gathered} K_{i-1} < X \leqslant K_i, \text{ если } 1 < i < q, \\ X \leqslant K_i, \text{ если } i = 1, \\ K_{i-1} < X, \text{ если } i = q. \end{gathered} \tag{2.4}$$

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 листы 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 2.4.

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 даёт возможность быстрее обрабатывать запросы на поиск в диапазоне. Операции вставки и удаления элементов в B$^+$-дерево аналогичны соответствующим операциям

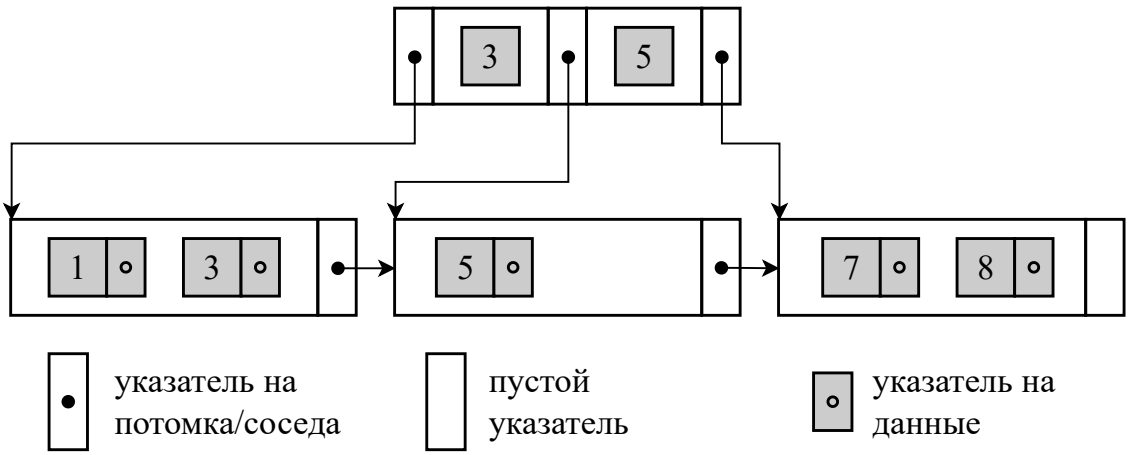

Рисунок 2.4 – Пример B^+-дерева

на B-дереве, ***за исключением***. Из-за большей скорости поиска по сравнению с B-деревьями и аналогичных операций B^+-деревья частно называют просто B-деревьями, подменяя исходный термин.

2.1.3 Обученные индексы

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 (рисунок 2.5).

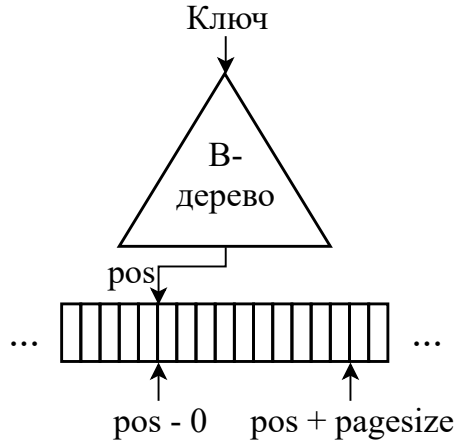

Рисунок 2.5 – B-деревья

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 2.6).

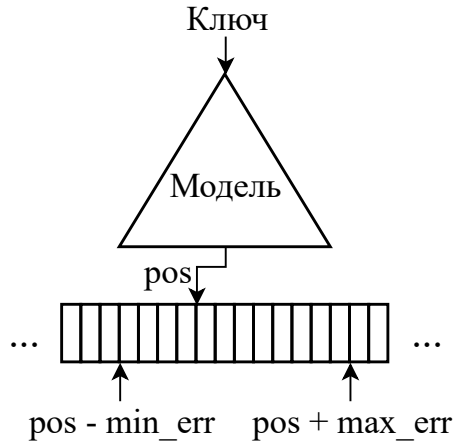

Рисунок 2.6 – Обученный индекс

При этом будет учитываться распределение данных (путем cdf) Так для предсказания можно представлять Range Index Models как модели функции распределения (рисунок 2.7):

$$\text{position} = F(key) \cdot N, \tag{2.5}$$

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

N — количество ключей.

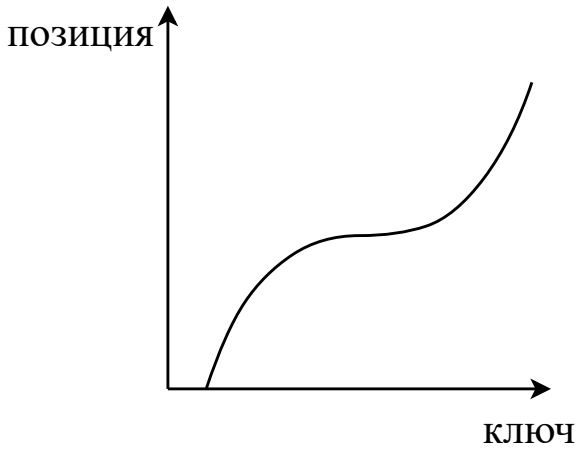

Рисунок 2.7 – Индекс как функция распределения

Можно построить индексы на основе рекурсивной модели (рисунок 2.8), в которой строится иерархия моделей из n уровней. Каждая модель на вход получает ключ, на основе которого выбирает модель на следующем уровне. Модели последнего этапа предсказывают положение записи.

Можно использовать различные модели: например, на верхнем использо-

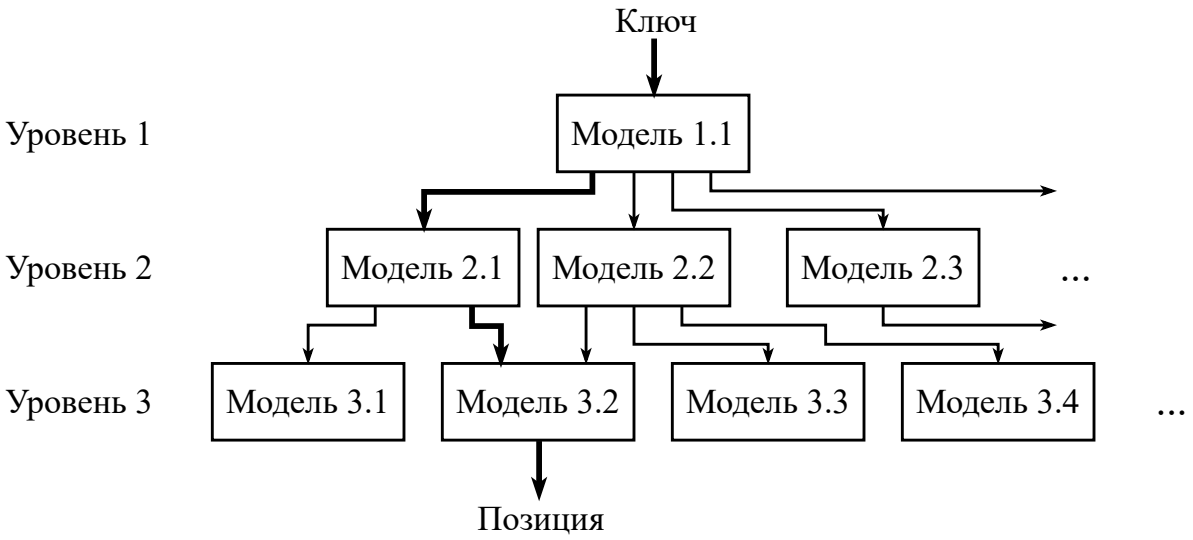

Рисунок 2.8 – Рекурсивная модель индекса

вать нейронные сети, а на нижних простые линейные регрессионные модели или даже простые B-деревья.

2.2 Индексы на основе хеш-таблиц

ship

Хэширование - это широко используемый метод построения индексов в основной памяти; такие индексы могут быть временно созданы для обработки операции объединения (как мы увидим в разделе 15.5.5). или может быть постоянной структурой в базе данных основной памяти. Хеширование также использовалось как способ организации записей в файле, хотя организация хэш-файлов используется не очень широко. Первоначально мы рассматриваем только хеш-индексы в памяти, а позже в этом разделе рассмотрим хеширование на основе диска.

В нашем описании хеширования мы будем использовать термин "корзина"для обозначения единицы хранения, в которой может храниться одна или несколько записей. Для хеш-индексов в памяти корзина может представлять собой связанный список записей индекса. Для индексов, основанных на диске, корзина будет представлять собой связанный список дисковых блоков. В организации хэш-файлов вместо указателей на записи в сегментах хранятся фактические записи; такие структуры имеют смысл только для данных, находящихся на диске. Остальная часть нашего описания не зависит от того, хранятся ли в сегментах указатели на записи или фактические записи.

Формально, пусть K обозначает набор всех значений ключа поиска, и пусть B обозначает набор всех адресов корзины. Хэш-функция h - это функция от K до B. Пусть h обозначает хэш-функцию. С хеш-индексами в памяти набор сегментов представляет собой просто массив указателей, с i-м сегментом со смещением i. Каждый указатель хранит заголовок связанного списка , содержащего записи в этом сегменте. Чтобы вставить запись с ключом поиска K2, мы вычисляем h(K2), который дает адрес корзины для этой записи. Мы добавляем индексную запись для записи в список со смещением i. Обратите внимание, что существуют другие варианты хэш-индексов, которые по-разному обрабатывают случай нескольких записей в корзине; форма, описанная здесь, является наиболее широко используемым вариантом и называется цепочкой переполнения.

Хэш-индексация с использованием цепочки переполнения также называется закрытой адресацией (или, реже, закрытым хешированием). Альтернативная схема хэширования, называемая открытой адресацией , используется в некоторых приложениях, но не подходит для большинства приложений индексирования баз данных, поскольку открытая адресация не поддерживает удаления эффективно. Мы не рассматриваем это дальше. Хэш-индексы эффективно поддерживают запросы на равенство по поисковым ключам. Чтобы выполнить поиск по значению ключа поиска Kj, мы просто вычисляем h(K,), затем выполняем поиск в корзине с этим адресом. Предположим, что два ключа поиска, K5 и K7, имеют одинаковое хэш-значение; что то есть h(K5) = h(K7). Если мы выполним поиск по K5, корзина h(K5) содержит записи со значениями ключа поиска K и записи со значениями ключа поиска K7. Таким образом, мы должны проверить значение ключа поиска для каждой записи в корзине, чтобы убедиться, что это та запись, которая нам нужна.

В отличие от индексов B +-дерева, хэш-индексы не поддерживают запросы диапазона; например, запрос, который хочет получить все значения ключа поиска v, такие, что l S v u, не может быть эффективно выполнен с использованием хэш-индекса.

Удаление столь же просто. Если значение ключа поиска записи, подлежащей удалению, равно K3, мы вычисляем h(K2), затем выполняем поиск соответствующей корзины для этой записи и удаляем запись из корзины. При представлении связанного списка удаление из связанного списка происходит

просто.

В хеш-индексе на основе диска, когда мы вставляем запись, мы определяем местоположение корзины с помощью хэширование по ключу поиска, как описано ранее. Предположим на данный момент, что в корзине есть место для хранения записи. Затем запись сохраняется в этом ведре. Если в ведре недостаточно места, говорят, что происходит переполнение ведра. Мы справляемся с переполнением ведра с помощью переливных ведер. Если запись должна быть вставлена в корзину b, a b уже заполнена, система предоставляет корзину переполнения для b и вставляет запись в корзину переполнения. Если переливное ведро также заполнено, система предоставляет другое переливное ведро, и так далее. Все переливные ведра данного ведра соединены вместе в связанном список, как показано на рисунке 14.25. При цепочке переполнения, учитывая ключ поиска k, алгоритм поиска должен затем выполнять поиск не только в сегменте h(k), но и в сегментах переполнения, связанных с сегментом h(k).

Переполнение корзины может произойти, если для заданного количества записей недостаточно корзин. Если количество индексируемых записей известно заранее, может быть выделено необходимое количество сегментов; вскоре мы увидим, как справляться с ситуациями , когда количество записей становится значительно больше, чем первоначально предполагалось- я устал. Переполнение корзины также может произойти, если некоторым корзинам назначено больше записей, чем другим, что приводит к переполнению одной корзины, даже если в других корзинах все еще много свободного места.

Такой перекос в распределении записей может возникнуть, если несколько записей могут иметь один и тот же ключ поиска. Но даже если на ключ поиска приходится только одна запись, может возникнуть перекос, если выбранная хэш-функция приводит к неравномерному распределению ключей поиска. Вероятность возникновения этой проблемы можно свести к минимуму, тщательно выбирая хэш-функции, чтобы гарантировать равномерное и случайное распределение ключей по сегментам. Тем не менее, может возникнуть некоторый перекос.

Чтобы уменьшить вероятность переполнения корзины, количество корзин выбирается равным (n,/ f,) * (1 + d), где n обозначает количество записей, f обозначает количество записей в корзине, d - коэффициент искажения, обычно

около 0,2. При коэффициенте помадки 0,2 около 20 процентов пространства в ведрах будет пустым. Но преимущество заключается в том, что вероятность переполнения уменьшается. Несмотря на выделение на несколько сегментов больше, чем требуется, переполнение сегмента все равно может произойти, особенно если количество записей превысит первоначально ожидаемое.

Хэш-индексация, как описано выше, где количество сегментов фиксируется при создании индекса, называется статическим хешированием. Одна из проблем со статическим хешированием заключается в том, что нам нужно знать, сколько записей будет сохранено в индексе. Если со временем добавляется большое количество записей, в результате чего получается гораздо больше записей, чем сегментов, поисковым системам придется выполнять поиск по большому количеству записей, хранящихся в одном сегменте или в одном или нескольких переполненных сегментах, и, таким образом, они станут неэффективными.

Чтобы справиться с этой проблемой, хэш-индекс может быть перестроен с увеличенным количеством ковши. Например, если количество записей становится в два раза больше количества сегментов, индекс может быть перестроен с вдвое большим количеством сегментов, чем раньше. Однако перестройка индекса имеет тот недостаток, что это может занять много времени, если отношения большие, что приводит к нарушению нормальной обработки. Было предложено несколько схем, которые позволяют увеличивать количество сегментов более поэтапным образом. Такие схемы называются методами динамического хеширования; метод линейного хеширования и метод расширяемого хеширования являются двумя такими схемами; более подробную информацию об этих схемах см. в разделе 24.5. методы.

Лошадь

Одной из простейших реализаций индекса является хэш-карта. Хэш -карта - это набор сегментов, содержащих результаты хэш-функции , примененной к ключу. Этот хэш указывает на местоположение, где можно найти записи. Хэш-карта пригодна только для поиска по одному ключу, поскольку сканирование диапазона было бы непомерно дорогим. Кроме того, хэш должен помещаться в память для обеспечения производительности. С учетом этих предостережений хэш-карты обеспечивают отличный доступ для конкретных случаев использования, для которых они работают.

Альтернативным способом организации индексов является хеширование. Идея этого подхода заключается в применении к значению ключа поиска некоторой функции свертки, называющейся *хеш-функцией*, по определенному алгоритму вырабатывающей значение меньшего размера. Значение хеш-функции от ключа используется как адрес в таблице, содержащей ключи и записи, называющейся *хеш-таблицей*. Хеш-функция должна иметь равномерное распределение ее значений для уменьшения числа коллизий — ситуаций, когда применительно к нескольким поступающим ключа образуется одно и то же значение свертки.

Что есть хеш-таблица в принципе как структура

2.2.1 Хеш-индексы

Как она используется для индексов.

Бакеты, статическое и динамическое хеширование

2.2.1.1 Описание структуры

Бакеты здесь???

2.2.1.2 Принцип построения

разрешение коллизий
«совершенное хеширование»
статическое/динамическое хеширование

2.2.1.3 Операции поиска, вставки и удаления элементов

Ключ -> хеш-функция -> бакет -> поиск (-> вставка/удаление)

2.2.2 Обученные хеш-индексы

Хеш-индексы можно рассматривать как модель сопоставления ключа позиции искомой записи в неупорядоченном массиве.

Функция распределения вероятностей распределения ключей может использоваться как хеш-функция. Для этого функция распределения масштабируется на размер хеш-таблицы M и для поиска положения записи аналогично случаю с B-деревьями используется формула:

$$h(K) = F(K) \cdot M, \tag{2.6}$$

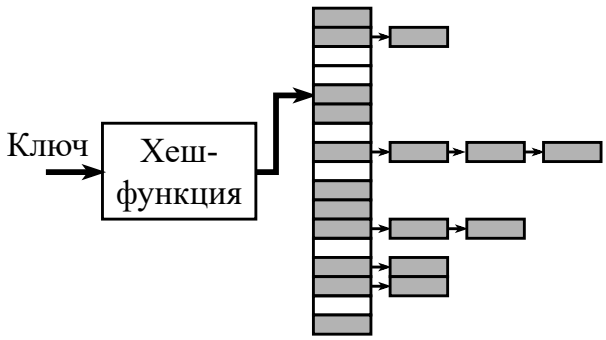

Рисунок 2.9 – Хеш-индекс

где K — ключ.

Обученные хеш-индекс учитывает эмпирическое распределение ключей, что позволяет уменьшать колличество коллизий по сравнению с обычными хеш-таблицами.

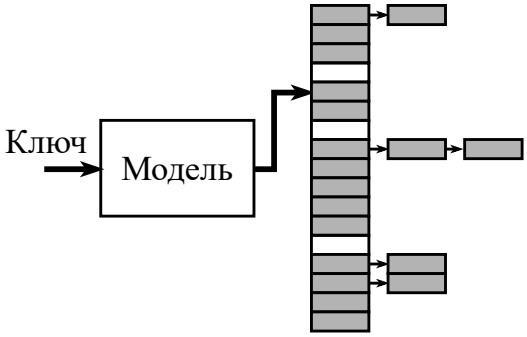

Рисунок 2.10 – Обученный хеш-индекс

2.3 Индексы на основе битовых карт

2.3.1 Фильтр Блума

Данные индексы можно рассматривать как модель проверки существования записи в массиве данных.

Фильтр Блума — алгоритм используемый для проверки существования записи.

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

возвращенных позиций, равен нулю, то ключ не входит во множество. Из этого следует, что данный алгоритм гарантирует отсутсвие ложноотрицательных результатов.

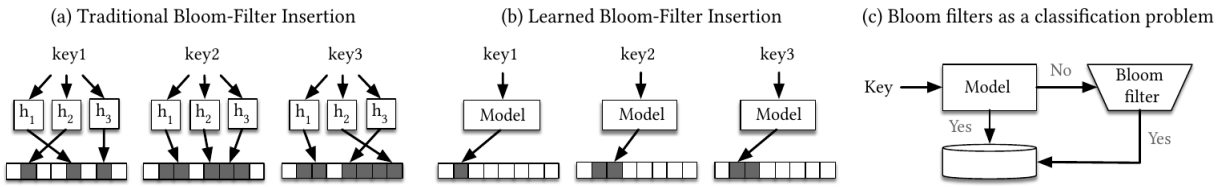

Рисунок 2.11 – Bitmap-индексы

Может со 100%-ной вероятностью сказать, что элемент отсутсвует в наборе, но то, что элемент присутсвует в наборе, со 100%-ной вероятностью он сказать не может (возможны ложноположительные результаты)

2.3.2 Обученные индексы

В случае индексов существования необходимо обучить функцию таким образом, чтобы среди возвращенных значений для множества ключей были коллизии, аналогично для множества неключей, но при этом не было коллизий возвращенных значений для ключей и неключей.

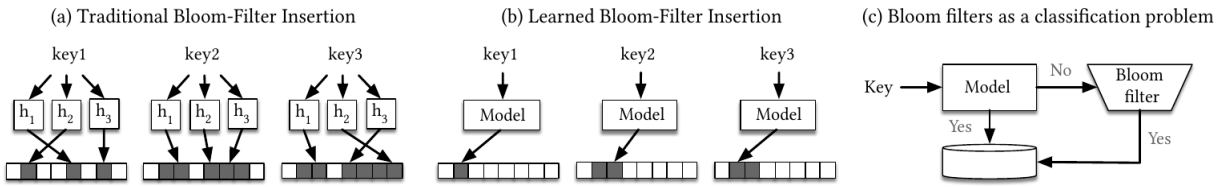

Рисунок 2.12 – Обученные bitmap-индексы

В отличие от оригинального фильтра Блума, где $FNR = 0$, $FPR = const$, где const выбрано априори, при обучении достигается заданное значение FPR при $FNR = 0$ на реальных запросах.

3 Классификация существующих методов

		LIPP	ALEX	PGM	Learned	B+Tree
Lookup	Complexity	$O(\log N)$	$O(\log N + \log m)$ [1]	$O(\log^2 N)$	$O(\log N)$	$O(\log N)$
	Latency [6]	24.23ns	68.92ns	151.53ns	139.09ns	237.94ns
Insert	Complexity	$O(\log^2 N)$	$O(\log^2 N + \log m)$ [2]	$O(\log^2 N + \log N)$ [3]	—	$O(\log N)$
	Latency [6]	70.93ns	204.94ns	217.17ns	—	1114.19ns
Search Range (Leaf)		$O(1)$ [4]	$O(m)$	$O(\epsilon)$ [5]	$O(\epsilon)$	$O(m)$
Search Range (Non-Leaf)		$O(1)$	$O(1)$	$O(\epsilon)$	$O(1)$	$O(m)$

Рисунок 3.1 – Классификация методов по сложности чтения-записи ;)

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. 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. — N 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. — N 1(38). — С. 63–67.
4. DAMA-DMBOK : Свод знаний по управлению данными. — 2-е изд. — М. : Олимп-Бизнес, 2020. — 828 с.
5. The Case for Learned Index Structures / T. Kraska [et al.] // Proceedings of the 2018 International Conference on Management of Data. — SIGMOD'18, June 10–15, 2018, Houston, TX, USA, 2018. — P. 489–504.
6. ALEX: An Updatable Adaptive Learned Index / J. Ding [et al.] // Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. — 2020. — P. 969–984.
7. APEX: A High-Performance Learned Index on Persistent Memory (Extended Version) / B. Lu [et al.] // Proceedings of the VLDB Endowment. — 2022. — Vol. 15(3). — P. 597–610.
8. Updatable Learned Index with Precise Positions / J. Wu [et al.] // Proceedings of the VLDB Endowment. — 2021. — Vol. 14(8). — P. 1276–1288.
9. *Ferragina P.*, *Vinciguerra G.* The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds // PVLDB. — 2020. — Vol. 13(8). — P. 1162–1175.
10. *Григорьев Ю. А.*, *Плутенко А. Д.*, *Плужникова О. Ю.* Реляционные базы данных и системы NoSQL: учебное пособие. — Благовещенск : Амурский гос. ун-т, 2018. — 424 с.

11. *Silberschatz A.*, *Korth H. F.*, *Sudarshan S.* Database System Concepts. — New York : McGraw-Hill, 2020. — 1344 p.

12. *Эдвард Сьоре*. Проектирование и реализация систем управления базами данных. — М. : ДМК Пресс, 2021. — 466 с.

13. *Осипов Д. Л.* Технологии проектирования баз данных. — М. : ДМК Пресс, 2019. — 498 с.

14. *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. — 1843 p.

15. Encyclopedia of Database Systems / ed. by L. Liu, M. T. Özsu. — New York : Springer New York, 2018. — 4866 p.

16. *Mannino M. V.* Database Design, Application Development, and Administration. — Chicago : Chicago Business Press, 2019. — 873 p.

17. *Campbell L.*, *Major C.* Database Reliability Engineering : Designing and Operating Resilient Database Systems. — Sebastopol : O'Reilly Media, 2018. — 560 p.

18. *Gupta G. K.* Database Management Systems. — Chennai : McGraw Hill Education (India) Private Limited, 2018. — 432 p.

19. *Petrov A.* Database Internals : A Deep Dive into How Distributed Data Systems Work. — Sebastopol : O'Reilly Media, 2019. — 370 p.