Московский Государственный Университет имени М.В.Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Системного Программирования

Дипломная работа

Реализация хранимых в блоках префиксных деревьев в XML-СУБД

Автор::
гр. 527
Борисенко Олег Дмитриевич

Научный руководитель:
ак. РАН проф. Иванников Виктор Петрович,
м. н. с. Таранов Илья Сергеевич

Москва, 2011

Аннотация

Реализация хранимых в блоках префиксных деревьев в XML-СУБД

Борисенко Олег Дмитриевич

В этой работе рассмотрены особенности префиксных деревьев в сравнении с другими типами деревьев. Также была разработана и реализована структура данных (BST) в СУБД Sedna, представляющая собой один из видов префиксных деревьев. В ходе работы произведено исследование разработанной структуры данных, обоснованы ключевые характеристики и проведён ряд испытаний производительности реализованной структуры данных в сравнении с Б+-деревом, реализованных в СУБД Sedna.

Abstract

Implementing block-stored prefix trees into XML-DBMS

Oleg Borisenko

In this paper we revisit main distinguishing features of prefix trees in comparison with other tree types. We have developed and implemented prefix datastructure «BST» into Sedna DBMS. Also there were main charactertics of BST analized and benchmark in comparison with B+-tree was carried out.

Содержание

1 Введение

В современных базах данных необходимо хранить большое количество данных. При этом, кроме необходимости просто хранить данные, существует необходимость организовывать эффективный поиск по этим данным. Для организации эффективного поиска по данным в современных СУБД используются вспомогательные структуры, называемые *индексами*. Целью этой работы является разработка и реализация в рамках СУБД Sedna структуры данных для хранения индексов, позволяющей производить эффективный поиск и компактное хранение.

При увеличении объёма хранимых данных, увеличиваются требования к памяти, и возникает потребность в обработке данных во внешней памяти. Работа с внешней памятью обладает рядом специфических особенностей, вследствие которых необходимо очень аккуратно подходить к вопросу выбора структуры данных для хранения информации. Существует обширный класс структур данных, которые хорошо подходят для этой цели.

При выборе структуры данных для хранения информации также необходимо понимать, что у каждой структуры данных существуют свои ограничения, которые могут сказываться на предполагаемом применении. В общем случае, как правило, используются Б-деревья или их вариации; существуют канонические надёжные реализации этой структуры данных. Также в этой роли могут использоваться деревья поиска и их вариации, а также префиксные деревья.

Уместным было бы описать, что собой представляет понятие *дерево*. *Связным графом* называется граф, все вершины в котором связаны, то есть для каждых двух вершин существует простая цепь, соединяющая эти вершины. В общем смысле *дерево* представляет собой связный граф, не содержащий циклов. В контексте работы здесь и далее для удобства описания будет подразумеваться ориентированный граф. *Ориентированным графом* $G = (V, E)$ называется пара множеств, где V — множество вершин (узлов), E — множество дуг (ориентированных рёбер). Вершина графа, в которую не ведут дуги, называется *корневым узлом дерева*; такая вершина в корневых деревьях одна. Вершины, из которых не исходит ни одной дуги, называются концевыми вершинами или *листьями*. *Степенью узла* называется количество исходящих дуг из этого узла. *Уровнем узла* будем называть длину пути от корневого узла до этого узла; уровень корневого узла равен нулю. *Детьми* некоего узла будем называть узлы, в которые

непосредственно входят дуги из этого узла. *Родителем* узла будем называть вершину, из которой в этот узел непосредственно входят дуги. *Братьями* или *братскими узлами* будем называть детей одного родителя.

В рамках этой работы под деревьями будут подразумеваться корневые деревья с описанной выше структурой. В общих чертах специфика работы с внешней памятью проявляется в нижеследующих моментах. Наиболее распространёнными представителями внешней памяти в контексте работы с базами данных на текущий момент являются *накопители на жёстких магнитных дисках*. В контексте работы с накопителями на жёстких магнитных дисках наиболее важными являются две особенности, которые являются существенными для эффективной работы с данными: секторная адресная организация и время считывания данных.

Большинство накопителей на жёстких магнитных дисках[1] устроены так, что время считывания данных напрямую зависит от физического расположения считываемых данных на диске. Время считывания данных зависит от того, насколько быстро механизм жёсткого диска может позиционировать считывающую головку над нужной дорожкой, и от скорости чтения. Таким образом, если считываемые данные располагаются физически последовательно, время их считывания минимально; время чтения последовательных данных и произвольно расположенных данных может отличаться на несколько порядков в зависимости от объёма считываемых данных.

Сектор является минимальной адресуемой областью данных на жёстком диске и считывается целиком. Размер сектора на текущий момент составляет от 512 байт до 4096 байт в зависимости от носителя. Таким образом, даже если для какой-то операции необходимо считать лишь 100 байт данных, это займёт ровно столько же, сколько и чтение всего сектора в этой области диска. В ходе работы будет использоваться понятие *блок*. *Блоком* мы будем называть непрерывный участок в памяти (внешней или оперативной) фиксированной величины. Как правило, его размер берется кратным максимально допустимому сектору для носителя информации для обеспечения автоматического выравнивания данных в памяти, но в теории он может быть любым.

[1] В этой работе мы не будем рассматривать специфические виды накопителей (к примеру, гибридные диски производства компание Seagate серии Momentus), так как принципы их работы до сих пор не раскрыты компаниями-производителями до конца и в силу их крайней нераспространённости. Также мы не будем рассматривать зависимости от алгоритмов выталкивания данных из кэшей, так как они варьируются в зависимости от произодителя устройства.

Стандартный размер блока в используемой СУБД Sedna равен 64 килобайтам. Отдельно стоит подчеркнуть, что все расчёты количества задействованных в операциях блоков будут вестись именно относительно описанных блоков, а не относительно физических секторов НМЖД.

Из написанного в последних трёх абзацах напрямую вытекает несколько соображений об эффективном использовании этих конструктивных особенностей:

- Необходимо максимально заполнять блоки, причём заполнять так, чтобы минимизировать количество блоков, задействованных при одном запросе. Это позволит наиболее оптимально использовать особенности секторной организации НМЖД.

- При организации хранимой информации крайне желательно использовать такие структуры данных, которые бы за счёт своего устройства располагали данные так, чтобы большинство операций происходило на последовательных наборах секторов[2], так как в противном случае ощутимая часть времени выполнения запроса может быть потрачена на перемещение магнитных головок, а не на непосредственно операции чтения.

Актуальность данной темы диктуется современными темпами роста количества хранимых данных и необходимостью поиска информации. Необходимость хранить огромное количество данных подталкивает к поиску способов хранить данные наиболее оптимально по занимаемому пространству без потерь производительности. В этой работе описывается устройство и реализация структуры данных, которая в некоторых случаях существенно превосходит Б+ деревья в эффективности использования занимаемого пространства без потерь в производительности. Кроме того, в некоторых случаях необходимо хранить информацию в виде строк произвольно большой длины, которые не всегда можно заранее ограничить[3]. Для реализации подобного свойства в Б-деревьях как правило используют так называемые «страницы переполнения», но такой подход эффективен не всегда. Описываемая в данной работе структура данных поддерживает хранение строк произвольно большой длины без дополнительных ухищрений и потерь.

[2]К примеру, если мы хотим найти все слова, начинающиеся на «пример», поиск будет наиболее быстр, если слова «пример», «примерный», «примерочная», «примерять» находятся в одном или хотя бы в последовательных секторах

[3]В качестве примера можно привести URI — поиск по них зачастую необходим, при этом длина подобных строк может быть непредсказуемо большой

В своей работе мы основываемся на многочисленных современных статьях, посвященных структурам данных; их полный список приведен в списке используемой литературы. Далее следует обзор ряда структур данных, которые принципиально подходят для хранения в блоках.

1.1 Обзор структур, подходящих для хранения в блоках

1.1.1 Классическое Б-дерево

Классическое Б-дерево[1] представляет собой сбалансированное сильно ветвистое дерево (см. рис. 1). Каждому узлу дерева как правило ставится в соответствие блок внешней памяти. Б-дерево состоит из страниц, каждая страница содержит ключи, упорядоченные по возрастанию. Каждый ключ имеет двух потомков — левого и правого. Правый потомок ключа является левым потомком следующего ключа. Левый потомок ключа содержит только ключи, меньшие данного, правый потомок — только большие. Для большей ясности мы в дальнейшем будем сопровождать описания структур некоторыми характерными иллюстрациями. Степенью Б-дерева называют максимальное коли-

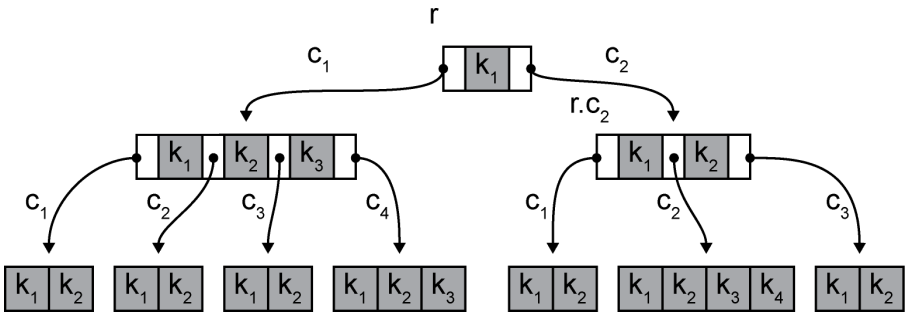


Рис. 1: Устройство Б-дерева

чество детей для каждого узла. B-деревья должны удовлетворять следующим требованиям:

- Каждый узел имеет не больше m детей, где m – степень дерева
- Каждый узел, кроме корневого узла и листьев имеет не меньше $m/2$ детей
- Корневой узел имеет не меньше двух детей (кроме тривиального случая)
- Все листья находятся на одном уровне; в них хранится информация

- Внутренние узлы с k детьми содержат k-1 ключ

Поиск в Б-дереве осуществляется так:

1. Начиная с корневого узла на каждом уровне ищется нужный указатель на следующего ребёнка
2. Переход по ссылке на следующего ребёнка
3. Повторять до тех пор, пока не достигнем листа
4. Поиск внутри узла осуществляется любым алгоритмом (чаще всего бинарным поиском)

Все операции вставки начинаются с листьев, алгоритм таков (см. рис.2):

- Найти с помощью указанного выше алгоритма положение, где должен быть добавлен элемент.
- Если найденный узел содержит меньше максимума допустимых элементов, то вставить элемент туда
- Если узел полон, то его нужно разделить на два узла следующим образом (см. рис.3):
    1. Находится средний элемент в листе, в который будет происходить добавление
    2. Значения, которые меньше среднего значения перемещаются в новую левую ветвь
    3. Значения, которые больше среднего значения перемещаются в новую правую ветвь
    4. Среднее значение перемещается в родительский узел изначального листа
    5. Если родительский узел переполнен, он делится таким же образом
    6. Если узел не имеет родителей, создаётся новый корневой узел

Удаление элемента производится по-разному в зависимости от того, из какого узла оно производится. При удалении ребалансировка происходит автоматически.

Удаление из листового узла происходит так:

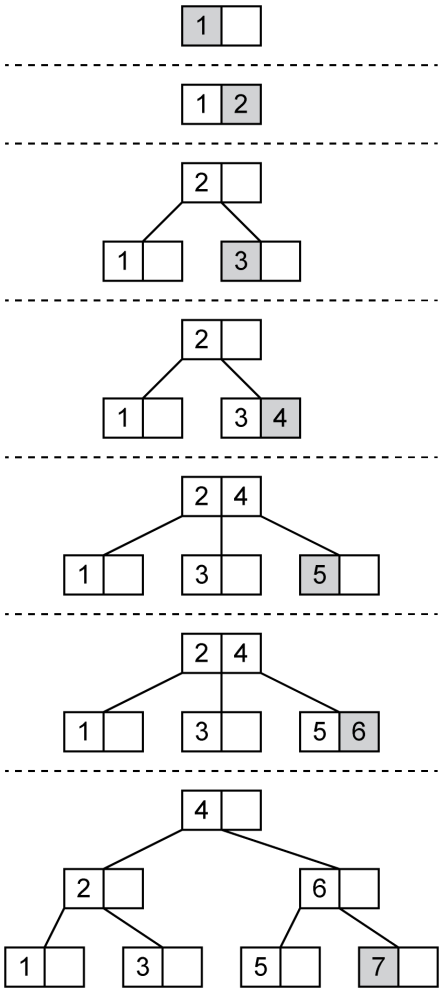


Рис. 2: Вставка в Б-дерево

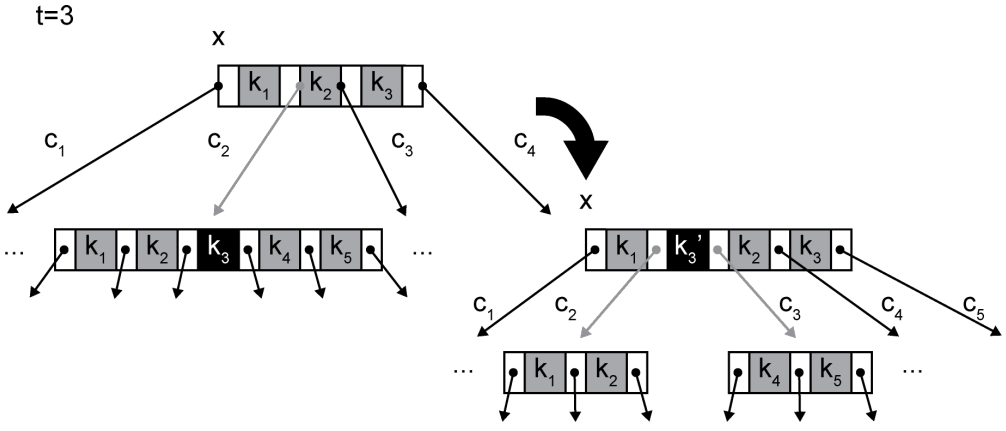


Рис. 3: Разделение Б-дерева

1. Найти значение для удаления
2. Если значение находится в листе, оно может быть просто удалено
3. Если значений становится слишком мало (меньше $m/2$), нужно проверить, можно ли слить братские узлы. Если нельзя, произойдёт обмен между братьями
4. Если удаление произошло из правой ветви, необходимо получить максимальное значение левой ветви
5. В обратной ситуации, соответственно, необходимо перенести минимальное значение из правой ветви

Удаление из внутреннего узла производится следующим образом:

1. Выбрать новый разделитель (наибольший элемент в левом поддереве или наименьший элемент в правом)
2. Удалить его из листа, в котором он находится, и заменить им элемент, который необходимо удалить
3. Теперь задача равносильна предыдущей

Итак, это был обзор первого кандидата. Теоретические выводы в контексте задачи хранения в блоках таковы:

- Из сбалансированности дерева следует существование гарантированного худшего времени доступа к любому узлу.
- Время доступа к любому узлу примерно одинаково вне зависимости от хранимых данных.
- За счет ограничений на емкость узла как сверху, так и снизу, данный тип дерева хорошо приспособлен для хранения в блоках.
- Задача хранения строк неопределённой длины не укладывается в каноническую модель; для реализации хранения строк неопределённой длины необходимо расширять структуру данных и использовать страницы переполнения.

1.1.2 Некоторые распространенные вариации Б-деревьев

Б+ деревья в каноническом виде по организации являются почти точной копией Б-деревьев за исключением двух отличий[1]: каждый лист хранит ссылку на ближайший братский узел (смотреть рис.4) по порядку обхода. Кроме того, в листьях хранятся все ключи. Это может давать выигрыш по времени в обходе вершин. Эта структура данных будет подробно рассмотрена в разделе, посвящённом существующим решениям рассматриваемой задачи.

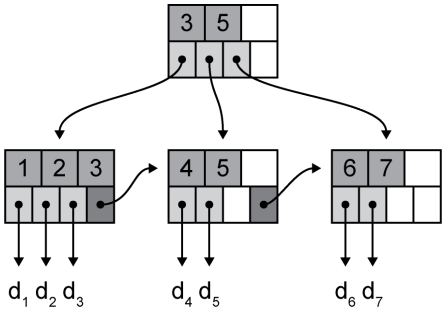


Рис. 4: Пример Б+ дерева

Префиксные Б+ деревья R. Bayer и K. Unterauer[2] предложили комбинированную структуру. Структура представляет собой модификацию Б+ дерева (см. рис.5). В этой модификации предлагается хранить вместо ключей-разделителей различающиеся префиксы хранимых слов. Все остальное, в том числе и алгоритмы, совпадает с Б+ деревьями.

Б* деревья отличаются от Б-деревьев только тем, что в них увеличена минимальная емкость узла с половины до двух третей. Это дает выигрыш в занимаемом месте, но в то же время, увеличивает нагрузку на узлы. В связи с этим несколько отличается алгоритм разделения узлов, но в целом алгоритмы сходны с Б-деревом.

1.1.3 Структуры типа «бор»

Русскоязычный термин «бор» был впервые употреблён в переводе книги Дональда Кнута «Искусство программирования»[3]. Бором называют множество структур, объединяющее префиксные и суффиксные деревья. Префиксные (суффиксные) деревья — это

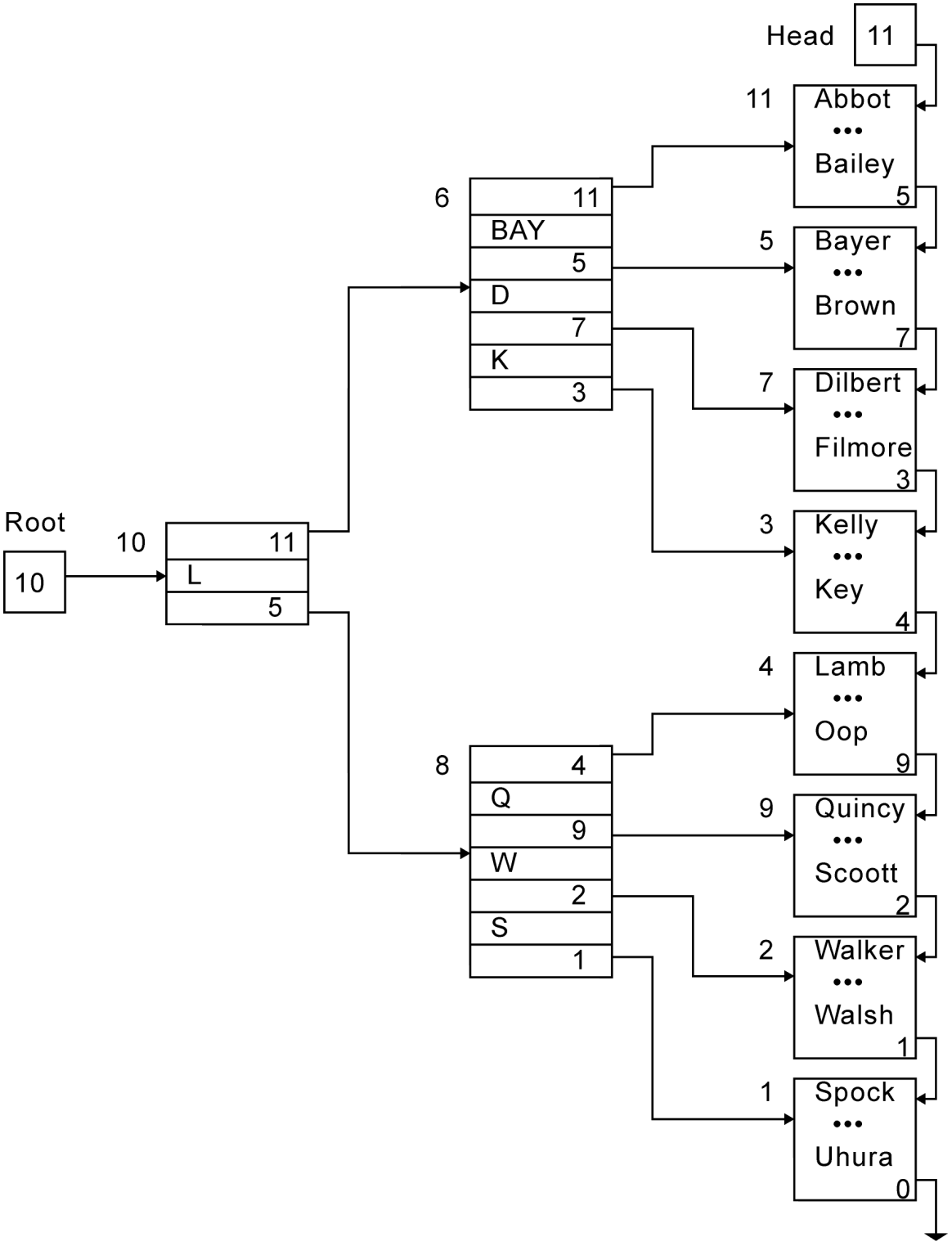


Рис. 5: Пример префиксного Б+ дерева

деревья, в которых строки группируются по общему префиксу (суффиксу). Для лучшего понимания, простейшую модель префиксного дерева можно представить следующим образом (см. рис.6): будем хранить в каждом не листовом узле массив, состоящий из n ссылок, где n — количество букв в алфавите языка, для которого мы составляем дерево. Каждая ссылка ведет на следующую букву, присутствующую в каком-либо из префиксов слов. В таком случае количество уровней дерева будет на единицу больше длины самого длинного хранимого слова. Понятно, что эта простейшая модель очень требовательна к ресурсам, так как занимает неоправданно много места в памяти. В связи с этим были разработаны различные методы оптимизации подобных структур данных. Возможные пути оптимизации таковы[4]:

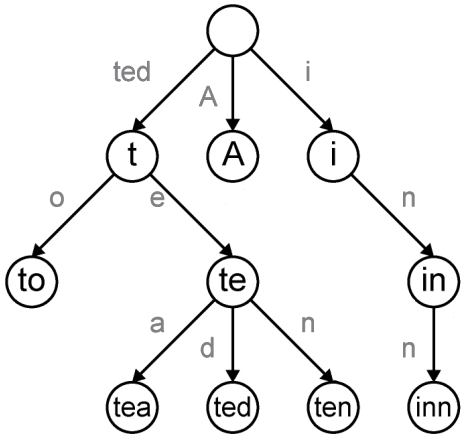


Рис. 6: Пример простейшего префиксного дерева

- Группирование частей дерева – ведёт к усложнению структуры данных и к увеличению времени доступа. Реализации, использующие этот метод:
  - Компактный бор — группируются все рёбра, которые идут только к одному листу
  - Бор Патрисии — группируются все избыточные пути, а не только те, которые ведут к листьям[4]. К бору Патрисии мы вернёмся чуть позже.
- Удаление лишних указателей

[4]Некоторые из приведенных примеров будут рассматриваться подробнее в дальнейшем

– Брайндэйз бор — в узлах хранятся не массивы, а связные списки с ненулевыми указателями.

– Тернарный бор Бентли — используется массив, указывающий на связные списки, которые в свою очередь указывают на двоичные деревья поиска. Такая модель выигрывает в занимаемом месте у простейшей модели, обладает меньшим временем доступа, чем у предыдущего варианта.

– Burst trie[5][5] — хранит строки в блоках фиксированного размера. Переполнение блока приводит к созданию нового родительского узла[5], который представлен в виде массива указателей, по одному на каждую букву алфавита. Строки оригинального блока распределяются по почти всем новым блокам. При хранении строк в блоках фиксированного размера и при создании только одного узла при переполнении можно выиграть в затратах на хранение данных, при этом это практически не отражается на скорости доступа к данным.

Бор Патрисии Бор Патрисии[6] (radix tree, Patricia trie, crit bit tree) —- структура данных, основанная на префиксных деревьях и предназначенная для хранения наборов строк. В отличие от обычных префиксных деревьев, узлы бора Патрисии помечены последовательностью символов, а не единичными символами. Это могут быть символьные строки, битовые строки (например, IP-адреса) и вообще любые последовательности, для которых возможно установить лексикографический порядок. Иногда названия «radix tree» и «crit bit tree» применяются только по отношению к деревьям, которые хранят целые числа, а «бор Патрисии» упоминается в контексте более общих данных, но сама организация данных в этих случаях идентична.

Бор Патрисии легче всего представить (см. рис.7) как оптимизированное по пространству префиксное дерево, в котором внутренние узлы, обладающие единственным потомком, объединяются с этим потомком. Как следствие, каждый внутренний узел обладает как минимум двумя потомками. В отличие от обычных префиксных деревьев,

[5]Название приводится в таком виде по двум причинам: во-первых, не удалось найти ни одного аналога или перевода этого термина в русскоязычной литературе, во-вторых, для простоты восприятия, т.к. в дальнейшем будут использоваться струтуры под названиями «B-trie», «SBTrie» (String Based Trie) и «BST», которые является наследниками идеи устройства Burst trie, но является совершенно иными структурами данных.

каждый узел может быть помечен как последовательностью символов, так и единичным символом. Наихудшая сложность поиска, вставки, удаления, нахождения предшественника, нахождения последующего элемента для всех операций одинакова и равна $O(k)$, где k — самая длинная строка в множестве хранимых строк.

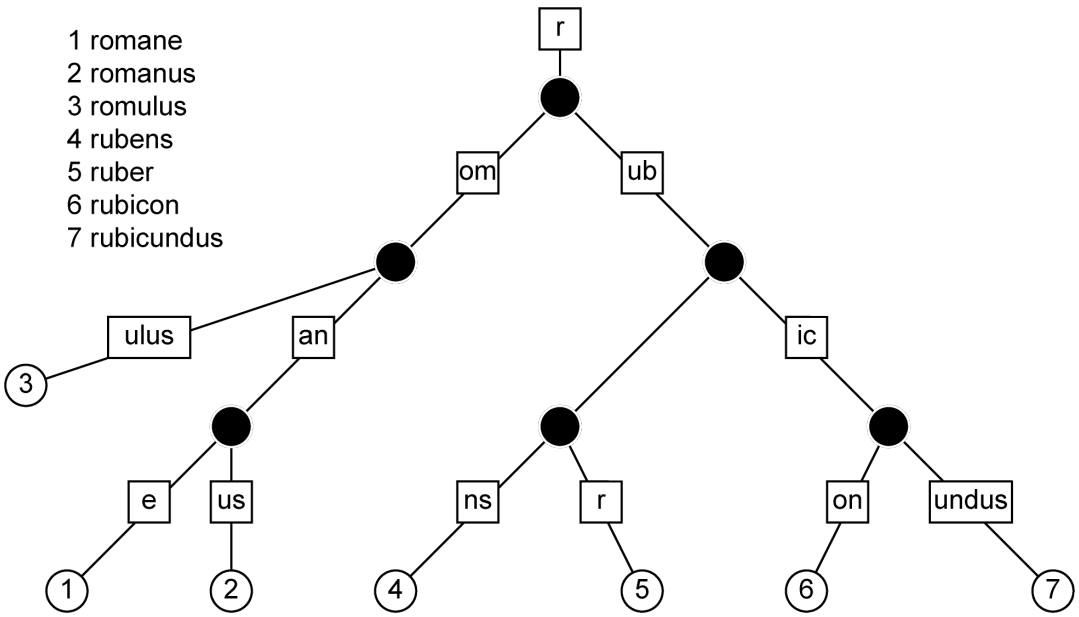


Рис. 7: Пример бора Патрисии

Поиск Поиск производится путем обхода дерева по буквам, входящим в запрос

Вставка

1. Находим место, где должен был бы находиться вставляемый элемент.
2. Либо добавляем исходящее ребро, помеченное оставшимися буквами добавляемого слова, либо, если исходящее ребро с частью оставшегося префикса уже существует, мы разделяем его на два, одно из которых содержит общий префикс, а второе — остаток добавляемого слова.

Удаление

1. Найти удаляемый элемент

2. Удалить лист с искомым элементом

3. Если у родительского элемента остаётся только один потомок, родитель объединяется с этим потомком.

Также доступны операции поиска предшественника и последующего элемента. В первом случае поиск сводится к поиску наиболее длинной строки, меньшей заданной; в качестве результата выдаётся родитель, идущий первым в лексикографическом порядке. Во втором случае требуется поиск наименьшей строки, большей, чем заданная строка, и в качестве результата выдаётся первый потомок узла в лексикографическом порядке.

Сравнение с другими структурами Будем считать, что ключи обладают длиной k, и структуры данных содержат n элементов. В отличие от сбалансированных деревьев, бор Патрисии позволяет производить поиск, вставку и удаление со сложностью $O(k)$ против $O(logk)$ у сбалансированных деревьев. На первый взгляд, это не является преимуществом, так как обычно $k > logk$, но в сбалансированном дереве каждое строковое сравнение происходит со сложностью $O(k)$ в худшем случае, и это даёт худшие результаты при обработке длинных общих префиксов. При этом в боре все сравнения занимают одинаковое время, но требуется m сравнений для нахождения строки длины m. Бор Патрисии может осуществлять эти действия с меньшим количеством сравнений и требует меньше узлов. Бор Патрисии, с другой стороны, обладает общими недостатками с префиксными деревьями, откуда следует то, что данная структура подходит не для любых наборов данных, в то время как Б-деревья, к примеру, безразличны к хранимой информации, если её можно представить в виде линейно-упорядоченного множества.

Б-бор Структура Burst trie на данный момент является одной из самых быстрых структур для хранения строк в памяти[5], но она не может быть напрямую размечена во внешней памяти из-за методов работы с блоками данных. Переполнение блока влечет за собой создание большого количества новых блоков фиксированного размера, что значит, что в отсутствие носителя данных с ассоциативным доступом, каждое переполнение блока будет вызывать огромное количество последовательных перезаписей блоков, а это недопустимое расточительство в случае использования НМЖД. Вследствие этого, была создана новая структура данных, лишенная этого недостатка и при

этом обладающая всеми достоинствами Burst trie. Эта структура называется Б-бор[7] (см. рис.8), и сейчас мы ее рассмотрим.

Б-бор — несбалансированная префиксная структура для хранения на диске, заимствующая идею эффективности по занимаемому пространству у Burst trie. Строки хранятся в блоках фиксированного размера. Каждый узел хранится в отдельном блоке, так что понятия «узел» и «блок» в данном случае идентичны. При переполнении количество вновь создаваемых узлов удерживается в разумных рамках, в отличие от burst trie. Авторы этой структуры данных предлагают алгоритм, называемый «разделением б-бора». Главная идея этого алгоритма заключается в том, что каждый узел хранит по одному символу строки из каждого блока. Как только все строки в блоке имеют повторяющийся префикс, он удаляется из каждой строки, а дальнейшее разделение этого блока приведет к созданию нового родительского узла. Такие блоки мы назовём «чистыми». В случае, если у множества строк в блоке разные префиксы, к ним ведут разные пути в дереве. Такие блоки мы назовём «гибридными», и их разделение не вызовет создания нового узла в дереве. В обоих случаях, каждый блок представляет собой скопление строк с общим префиксом. В сравнении с Б+ деревьями есть потенциальный недостаток: загруженность блоков может быть неравномерной.

Для ускорения операций вводится используется хэш-таблица, которая хранит введенные запросы для поиска.

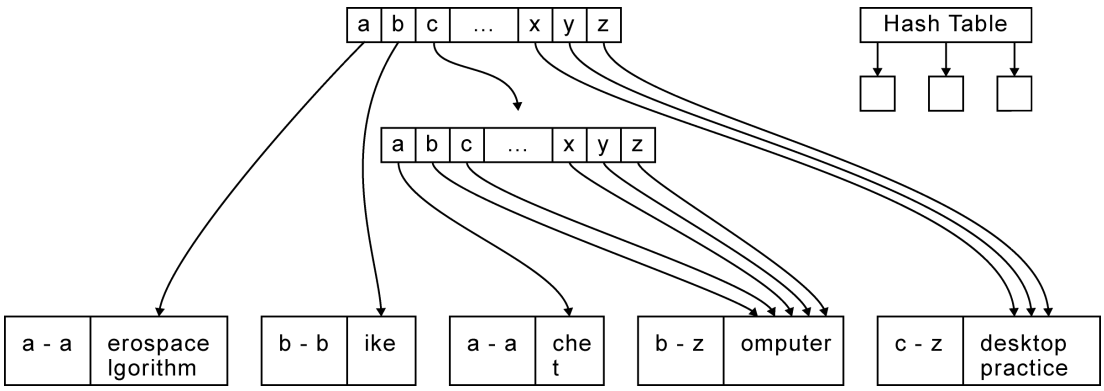


Рис. 8: Пример б-бора

Инициализация На момент инициализации б-бор имеет один гибридный узел и его родителя. При разделении гибридного узла, создаётся братский узел, и бор рас-

тёт горизонтально. Деление гибридного блока приводит к созданию чистого. Деление чистого блока приводит к созданию родительского элемента.

Поиск При поиске используется строка запроса, на выходе получаем указатель на искомый узел, на его родительский узел (если он есть) и остаток строки запроса, не поглощённый во время обхода. Если строка при обходе поглощена полностью, то она хранится в хэш-таблице. Поиск происходит следующим образом. Каждая буква запроса сопоставляется с хранимым узлом в префиксе. Если запрос длиннее префикса, то мы переходим в одного из детей и так до тех пор, пока не достигнем листа, или пока строка запроса не поглотится. Если строка поглотилась, то результаты получаются из хэш-таблицы.

Вставка Сначала происходит поиск вставляемого элемента, если такого элемента в боре не было, вставляется остаток строки запроса в тот блок, на котором завершился поиск. Вставка происходит либо в чистый блок (если такой есть), либо в гибридный (если чистого блока не оказалось). Если блок переполнился во время вставки, происходит разделение. Если нет — вставка завершена.

Удаление Удаление в данном случае реализуется ленивым алгоритмом. Происходит поиск элемента, который необходимо удалить. Если строка запроса поглощается полностью, то из найденной точки удаляется символ конца строки, и строка запроса удаляется из хэш-таблицы. Если во время поиска мы получаем нулевой указатель, то просто удаляем строку запроса из хэш-таблицы. Если от строки запроса к моменту достижения узла остаётся суффикс, мы удаляем его из найденного блока, и блок дефрагментируется.

Разделение Разделение узла производится в случае, если блок переполнен. Ищется примерная середина блока и делится в соответствии с лексикографическим порядком слов, в ней хранящихся, после чего происходит рост вширь. Вследствие такого алгоритма могут образоваться недогруженные блоки, но пустых получиться не может.

2 Постановка задачи

Задачи данной работы заключаются в следующем:

1. Построить и описать структуру данных, основанную на идее префиксного дерева, обладающую следующими свойствами:
   - Структура должна поддерживать поиск по строкам произвольно большой длины и их хранение
   - Структура должна обеспечивать хранение данных в блоках
   - Структура должна позволять хранить мультимножество пар «ключ/значение» и производить поиск по ключу и по паре «ключ/значение»
2. Произвести сравнение между реализованной структурой и существующей реализацией Б+-дерева по следующим параметрам:
   - Время, затрачиваемое на добавление и поиск одинаковых данных в обе структуры
   - Объем занимаемого пространства обеими структурами при добавлении одинаковых данных
   - Для оценки возможностей построенной структуры необходимо провести тестирование на нескольких наборах данных, а именно:
     - Набор множества URL
     - Набор множества URI
     - Набор строк-идентификаторов некоторого множества статей
     - Набор случайных строк

3 Обзор существующих решений

Как правило, для организации быстрого поиска в базах данных используются вспомогательные структуры, называемые «индексами». Строго говоря, существует множество разнообразных техник организации индексов, но идея у всех одна — обеспечить более быстрый поиск по базе данных за счёт уменьшения пространства поиска. Мы будем использовать индексы по значениям.

Как уже было отмечено во введении, одной из наиболее распространённых структур данных для хранения индекса является Б-дерево и его модификации[8]. Это связано с тем, что эта структура хорошо изучена и имеет ряд ценных свойств: прогнозируемые наибольшую и наименьшую высоту, сбалансированность и организацию, позволяющую хранить дерево в блоках. В СУБД Sedna[9] используется собственная реализация Б+-дерева, и именно с этой версией дерева будет сравниваться построенная в работе структура данных. В этой части работы будет произведён подробный обзор использующегося с СУБД Sedna Б+-дерева и будут описаны принципы сравнения характеристик Б+-дерева и строкового префиксного дерева.

3.1 Б+-дерево в СУБД Sedna

Б+-дерево представляет собой сбалансированное сильно ветвистое дерево. Каждому узлу дерева соответствует блок памяти фиксированного размера. Б+-дерево состоит из страниц (узлов), каждая страница содержит ключи, упорядоченные по возрастанию. Ключи могут представлять собой любые символьные строки. Каждый ключ имеет двух потомков — левого и правого. Левый потомок ключа содержит только меньшие или равные ключи, правый потомок содержит большие ключи. Правый потомок ключа является левым потомком следующего ключа. Б+-дерево, реализованное в СУБД Sedna, обладает следующими свойствами:

- Ограничения на количество хранимых в блоке узлов задаются непосредственно размером блока.
- Все листовые узлы находятся на одном уровне; в них содержатся все ключи, встречающиеся в дереве
- Внутренние узлы с k детьми содержат $k - 1$ ключ

- Поиск всегда заканчивается в листовой вершине
- Листовые вершины связаны ссылками на братские узлы
- Ключи представляют собой символьные строки, в качестве значений используются указатели на данные, соответствующие значениям
- В случае переполнения количества значений в узле, последнее значение содержит ссылку на другой блок, в котором находится продолжение массива указателей на значения

Однако в данной реализации дерева нет строгого ограничения на заполненность узлов по нижней границе. В соответствии со статьёй, посвящённой этому вопросу, снятие ограничения на нижнюю границу заполненности дерева положительно влияет на производительность и слабо влияет на занимаемое пространство[10]. Поиск в Б+-дереве осуществляется так:

1. Начиная с корневого узла на каждом уровне производится поиск[6] указателя, ссылающегося на следующее поддерево
2. Производится переход по ссылке на следующего ребёнка
3. Первые два шага повторяются до тех пор, пока не поиск не достигает листа

Все операции вставки начинаются с листьев, алгоритм таков:

1. Найти с помощью указанного выше алгоритма положение, где должен был бы находиться добавляемый элемент
2. Если найденный узел содержит меньше максимума допустимых элементов, то элемент вставляется в найденное место
3. Если узел полон, то его нужно разделить на два узла. Алгоритм разделения приведён дальше.

Разделение узла дерева происходит следующим образом:

[6]На основе алгоритма бинарного поиска

1. В узле, который необходимо разделить, ищется точка разбиения; она соответствует середине узла
2. Выделяется новый блок
3. Значения, которые больше серединного ключа, перемещаются в новый правый лист
4. Самый маленький ключ нового листа вставляется в родительский узел вместе с указателем на новый лист
5. В последний элемент «старого» блока прописывается ссылка на самый маленький ключ отделившегося листа
6. Если родительский узел переполнен, он делится таким же образом
7. Если узел не имеет родителей, создаётся новый корневой узел

Удаление элемента производится по-разному в зависимости от того, из какого узла оно производится. При удалении ребалансировка происходит автоматически. Удаление из листового узла происходит так:

1. Производится поиск удаляемого элемента
2. Производится расчёт занятого места в блоке после удаления элемента
   - Если после удаления элемента лист останется заполненным хотя бы наполовину, то элемент просто удаляется, и процедура удаления на этом завершается
   - Если после удаления элемента лист стал бы недозаполненным до половины размера блока, то необходимо предпринять дополнительные действия
3. Если лист становится недозаполненным до половины размера блока, то проверяется заполненность соседних братских листовых узлов.
4. Если это возможно, происходит слияние недозаполненного листьев с одним из братских листьев. Если же слияние невозможно, ничего не происходит[7]
5. В случае слияния двух листьев удаляется один из указателей в родительской вершине (в зависимости от того, с каким из братьев слился данный блок)

[7] это является следствием свойства данной реализации, упомянутого в начале раздела

3.2 Методика и критерии сравнения

Поскольку одной из задач работы является сравнения построенной структуры с реализованным в СУБД Sedna Б+-деревом, необходимо обозначить методику и критерии сравнения. В ходе работы была реализована структура префиксного строкового дерева как структура для хранения индексов в СУБД Sedna. Б+-дерево было уже реализовано в том же качестве. Также интепретатор команд СУБД Sedna был преобразован таким образом, чтобы принимать тип структуры, используемой для построения индексов, как один из параметров команды построения индекса. Таким образом, обеспечивается возможность явного сравнения. Для сравнения были подготовлены несколько наборов данных. В качестве набора данных для тестирования использовалась база данных DBLP Computer Science Bibliography[11]. Тестирование проводилось с использованием СУБД Sedna. Наборы данных формировались посредством создания индексов; публикации индексировались по идентификатору (key) и по двум типам URI-ссылок (EE и URL). Всего индексировалось 939613 публикаций. Также был создан ещё один документ, призванный продемонстрировать то, что BST подходит не для любых данных. Для генерации использовался инструмент XBench[12]. Индексирование проводилось по полю Q, которое содержит в себе наборы случайных сгенерированных строк. Теперь непосредственно об оценке результатов. Необходимо сравнить результаты производительности и занимаемого пространства для каждой структуры, а именно:

- Оценка производительности каждой операции будет измеряться в милисекундах. Измерения времени будут производиться в ОС Linux на компьютере со следующей аппаратной конфигурацией: Intel Core i7 930, 6GB DDR3 RAM, Ubuntu 10.10 x86_64.

- Оценка занимаемого пространства будет выражаться в количестве занятых структурой блоков и в количестве занятых мегабайт[8]. Эта величина будет измеряться для каждого индекса непосредственно после его создания.

- Производительность операции добавления элементов в структуры будет оцениваться через операцию создания индекса. При создании индексов одних и тех же

[8]Численное выражение в занятых мегабайтах будет приводиться исключительно из соображений простоты восприятия — эта величина всегда будет равна количеству занятых блоков помноженных на размер блока.

документов в обе структуры будут происходить вставки одних и тех же элементов и в том же количестве.

- Производительность операции поиска будет оцениваться через время, требуемое для поиска 10% публикаций в случайном порядке в созданных индексах. Список искомых публикаций был сформирован заранее, чтобы подобный тест можно было повторить для обеих структур.

- Производительность операции удаления оцениваться не будет в связи с тем, что удаление в обеих структурах реализовано при помощи «ленивых алгоритмов», и скорость их реального выполнения измерению не поддаётся.

Каждый из тестов будет производиться отдельно для выполнения в оперативной памяти и для внешней памяти. В связи с критичностью вопроса занимаемого пространства и производительности необходимо усовершенствовать используемые структуры данных. Тесты показывают, что реализованная структура данных имеет преимущества в занимаемом пространстве при идентичных наборах данных без потерь в производительности. Кроме того, создание новой структуры данных диктуется необходимостью хранить строки произвольного размера, так как их хранение реализуется в Б+-деревьях через «блоки переполнения»[13], а эта техника обладает определенными недостатками.

4 Исследование и построение решения задачи

В данной работе представлена структура для хранения множества строк, основанная на префиксных деревьях. Множество хранимых строк в дальнейших описаниях мы будем обозначать символом K. Построенную стуктуру в дальнейшем мы будем называть префиксным строковым деревом или BST (Block String Trie)[14]. Над данной структурой данных определены следующие словарные операции:

- Добавление строки s в множество K
- Поиск всех строк с префиксом s, входящих в множество K
- Удаление строки s из множества K

4.1 Задача поиска по ключу и по паре «ключ/значение»

Поскольку в работе предполагается построение структуры данных для использования в СУБД, необходимо уточнить, как именно обеспечиваются возможность для хранения пар «ключ/значение» и возможность поиска значений по ключу или по паре «ключ/значение». Подобная возможность обеспечивается тем, что подобные пары можно представить в виде $s = k + c + v$, где k является ключом, v — любым строковым представлением значения, а в качестве c испольуется символ, которого нет в алфавите символов ключей[9]. Отсюда также вытекает и возможность поиска всех значений, соответствующих некоторому ключу: для этого необходимо найти все строки с префиксом $k + c$. Таким образом задачи хранения и поиска по паре «ключ/значение» и поиска по ключу сводятся к задаче хранения и поиска строк.

4.2 Префиксное дерево

Структура данных, которой посвящена данная работа, представляет собой корневое дерево T, хранящее множество строк K. Наиболее близкими родственниками этой структуры являются описанные во введении структуры «burst trie»[5] и «b-trie»[7]. Итак, BST устроена следующим образом:

1. Каждая вершина x дерева T содержит несколько полей:

[9] Для текстовых строк можно использовать нулевой символ

- Префикс $prefix(x)$ (возможно, пустой)
- Упорядоченный в лексикографическом порядке массив $E(x)$ исходящих из неё ребёр, помеченных различными символами
- Служебные двоичные флаги (необходимые для описания структуры флаги будут описаны по мере их введения)

2. Рёбра $e = (x, L_i(x))$, где $L_i(x)$ это вершина, в которую ведёт i-е ребро из массива $E(x)$. Каждое ребро помечено символом $c(e)$. В данном случае мы считаем под «симоволом» строку единичной длины.

3. Любой путь $S(x_1, x_n) = x_1e_1x_2e_2 \dots e_{n-1}x_n$ в описанном дереве задаёт строку s, получающуюся из пути $S(x_1, x_n)$ конкатенацией входящих в него строк $s = prefix(x_1) + c(e_1) + prefix(x_2) + c(e_2) + \dots + c(e_{n-1}) + prefix(x_n)$. Также введём первый служебный флаг: $final(x)$ определяет соответствует ли данной вершине какой-либо из ключей множества строк K. Таким образом, строка s, заданная путём $S(x_1, x_n)$, принадлежит к множеству строк K в том и только в том случае, если вершина x_n помечена флагом $final(x_n)$

Из последнего пункта вытекает тот факт, что в общем случае некоторому множеству строк K может соответствовать более одного дерева — для этого к дереву T, соответствующему множеству строк K, нужно добавить не помеченную флагом $final(x')$ вершину x' с произвольным префиксом. Тогда полученное и изначальное деревья будут задавать один и тот же набор строк K. По этой причине было бы логично ввести дополнительное свойство, которое охватывает подобные ситуации:

4. Любая вершина $x \in T$, для которой $n(x) \leq 1$, помечена как $final(x)$. Деревья, для которых выполняется это свойство, будем называть *минимальными*. Вершины x, для которых выполняется (не выполняется) условие $n(x) \leq 1 \Rightarrow final(x)$, мы будем называть *неизбыточными/избыточными*.

Отметим, что при выполнении последнего свойства множество K однозначно задаёт дерево T. Докажем это.

Теорема 1. *Множеству строк K соответствует одно и только одно дерево T, удовлетворяющее свойствам 1–4.*

Доказательство. Допустим, что это утверждение ложно, и множеству строк K соответствует не единственное дерево T. Это подразумевает под собой два случая: что множеству строк K не соответствует ни одного дерева или что множеству K соответствует более одного дерева. Первый случай мы рассматривать не будем формально, поскольку то, что любому множеству K соответствует хотя бы одно такое дерево, следует из построения.

Итак, допустим, что множеству строк K соответствует более одного дерева T, удовлетворяющего свойствам 1–4. Рассмотрим два дерева, соответствующие множеству K. Из допущения следует, что в множестве K существует строка k, для которой существуют пути $S(x_1^1, x_n^1)$ и $S(x_1^2, x_m^2)$ в деревьях T и T' соответственно такие, что:

$$k = prefix(x_1^1) + c(e_1^1) + prefix(x_2^1) + c(e_2^1) + \ldots + c(e_{n-1}^1) + prefix(x_n^1) = \\ prefix(x_1^2) + c(e_1^2) + prefix(x_2^2) + c(e_2^2) + \ldots + c(e_{m-1}^2) + prefix(x_m^2) \quad (1)$$

Для такой строки k выполнение равенства (1) означает то, что либо среди вершин x_i деревьев T и T' существует хотя бы одна вершина пара вершин[10] x_i^1 и x_i^2, для которых $prefix(x_i^1) \neq prefix(x_i^2)$, либо существует хотя бы одна пара рёбер e_i^1 и e_i^2 такие, что $c(e_i^1) \neq c(e_i^2)$, либо, что существует пара соответствующих вершин x_i^1 и x_i^2 таких, что у одной вершины стоит флаг $final(x_i^{1|2})$, а у другой нет. Но первые два случая противоречат построению, а третий случай вступает в противоречие с четвёртым пунктом свойств дерева. Значит, деревья T и T' совпадают, и следовательно существует лишь одно дерево, соответствующее множеству строк K. □

В дальнейшем мы будем рассматривать только минимальные деревья, если не будет явного указания об обратном. Тем не менее, в реализации структуры мы не будем требовать выполнение свойства (4) для реализации операции «отложенного удаления»[11]

[10] Если в одном из деревьев существует вершина x_i, а во втором не существует, это также означает, что они не совпадают

[11] Об «отложенном удалении» будет рассказано позже в разделе описания алгоритмов.

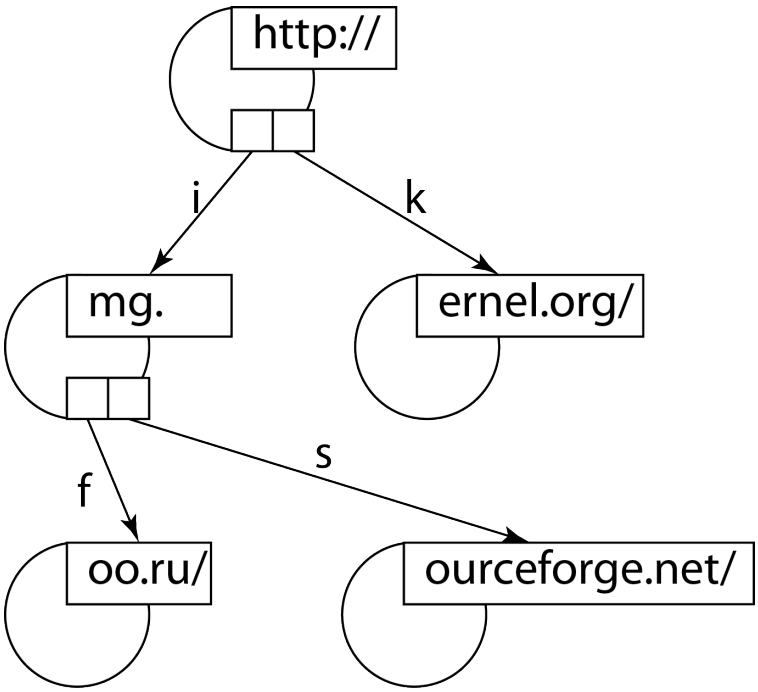


Рис. 9: Пример префиксного строкового дерева без разбиения на блоки

4.3 Разделение префиксного дерева на блоки

Описанная в предыдущем разделе структура хорошо подходит для хранения и поиска строковых ключей в оперативной памяти. Но в связи с необходимостью её использования для больших объёмов данных во внешней памяти, нужно эффективно разбивать её на блоки фиксированного размера.

Прежде всего, введём особый тип вершин, которые не хранят префиксов и ссылок на другие вершины, а хранят только ссылку на другой узел, находящийся в другом блоке. Такие вершины мы будем называть «ссылочными» и помечать служебным флагом $external(x)$. Следует особо отметить, что узлы, на которые ссылается ссылочная вершина, в свою очередь, не являются ссылочными. Кроме того, введём понятие «ветка».

«Веткой» называется корневое дерево B такое, что конечные вершины всех путей от корня к листовым вершинам в нём помечены либо флагом $final(x)$, либо $external(x)$. Если в дереве T нет ссылочных вершин, то оно состоит из одной ветки. Понятие «ветки» является одним из ключевых в вопросе разбиения дерева на блоки. Любое дерево T может быть произвольным образом разбито на ветки путём вставки перед любой

вершиной новой ссылочной вершины.

Блоки хранят в себе одинаковый объём данных W байт[12]. Распределение веток по блокам организовано таким образом, что в в одном блоке могут храниться одна или несколько веток исходного дерева. Все хранящиеся в блоке ветки обладают общим прямым предком. Эти условия необходимы для обеспечения локальности изменений в дереве и для обеспечения блокировок на уровне блоков. Кроме того, из этих условий следует, что в блоке, содержащем корневую ветку, других веток нет.

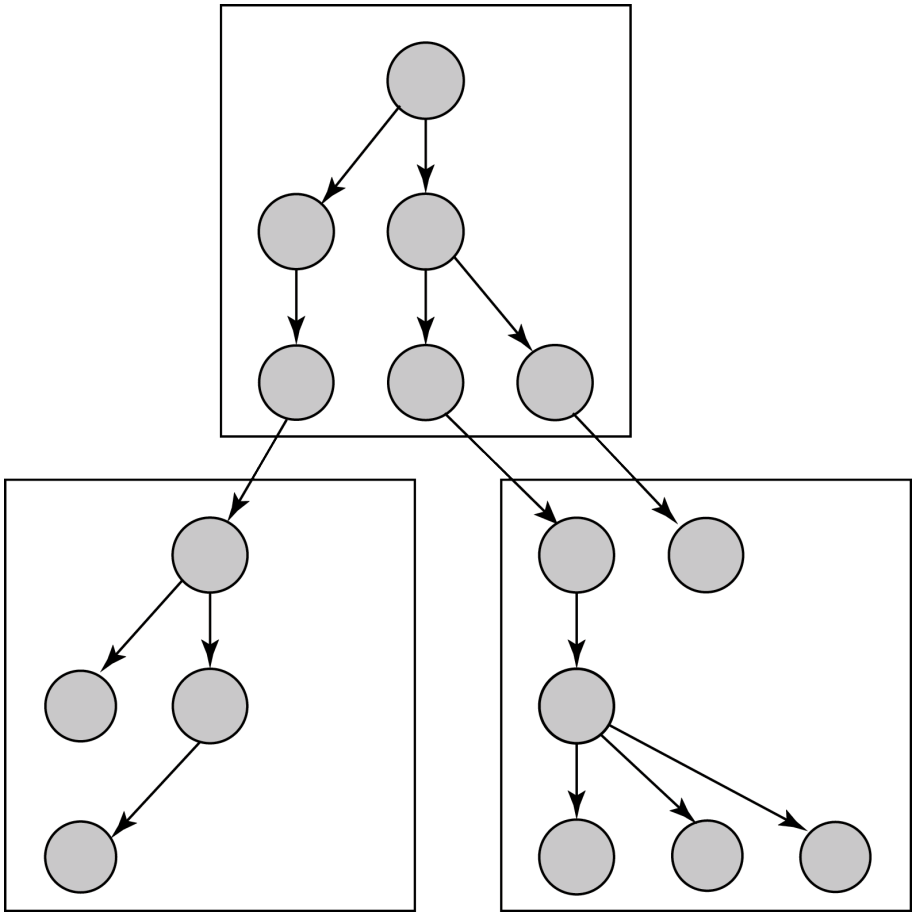


Рис. 10: Пример разбиения на блоки префиксного строкового дерева

[12]В СУБД Sedna, на которой производилось тестирование, размер блока по умолчанию равен 64кб.

4.4 Алгоритмы

4.4.1 Поиск

Алгоритм поиска по префиксному строковому дереву требует два входных параметра: указатель r на узел, с которого необходимо начинать поиск, и строку k, которую предполагается найти. В качестве возвращаемого значения алгоритм $BST - Search(r, k)$ возвращает узел x_{n+1} такой, что:

1. Строка $s(x_n)$ задаваемая путём $S(r, x_n)$, если такая существует, является префиксом искомой строки k или совпадает с ней.

2. Искомая строка k является префиксом строки $s(x_{n+1})$ либо совпадает с ней.

Таким образом, если под отношением $A \leq B$ понимать то, что строка A является префиксом строки B или совпадает с ней, то выполняется неравенство $s(x_n) \leq k \leq s(x_{n+1})$.

Поиск по дереву начинается с корня дерева. Функция принимает на вход указатель x на корневой узел поддерева и искомую строку k, промежуточные результаты поиска сохраняются в стеке S. Кроме того, также присутствует вспомогательная функция $y = L(x, c)$, которая находит среди исходящих рёбер вершины x ребро, помеченное символом c, и возвращает узел y, в который оно ведёт, либо NULL, если такого ребра нет. Эта функция реализована на основе алгоритма бинарного поиска, так как массив рёбер упорядочен. Также используется функция $Cut(p, s)$, которая возвращает строку, получающуюся удалением префикса p из строки s.

В конечном счёте, алгоритм поиска можно записать в компактном виде, и он таков:

```
BST-Search(x, k, S)
 1: Push(S, x)
 2: if external(x) then
 3:    Disk-Read(J(x))
 4:    return BST-Search(J(x), k, S)
 5: end if
 6: if not Is-Prefix(prefix(x), k)) then
 7:    if Is-Prefix(k, prefix(x)) then
 8:       return x
 9:    else
10:       return NIL
11:    end if
```

```
12: else
13:   s ← Cut(prefix(x), k)
14:   if Empty(s) then
15:     return x
16:   else if L(x, s[1]) = NIL then
17:     return NIL
18:   else
19:     return BST-Search(L(x, s[1]), Cut(s[1], s), S)
20:   end if
21: end if
```

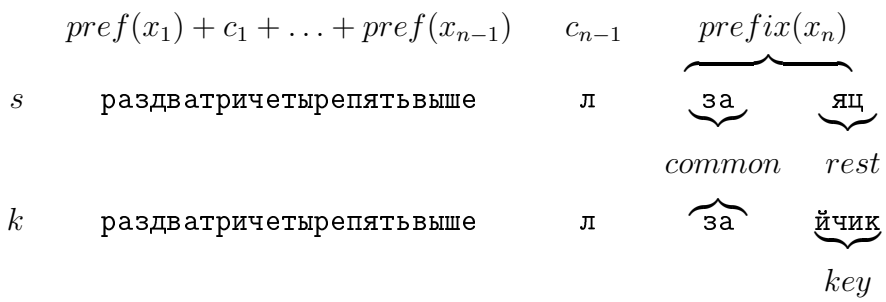
Функция поиска возвращает такой узел x, что все пути, ведущие из корня в final-вершины и содержащие x, задают строки, для которых искомая строка является префиксом, либо функция возвращает NULL, если такого узла нет. Таким образом, остаётся найти все эти пути с помощью обхода поддерева от возвращенной вершины.

4.4.2 Вставка

Перед началом операции вставки происходит поиск блока, в который будет происходить вставка, и строится структура, описывающая изменение блока. В случае, если в блоке недостаточно места для вставки, запускается функция разделения блока, о которой будет рассказано в следующем подразделе, и функция вставки запускается снова только после завершения процесса разделения. Итак, опишем алгоритм.

Вставка начинается с поиска пути в дереве к строке k, которую необходимо вставить. Алгоритм поиска пути идентичен описанному в прошлом подразделе за исключением того, что в данном случае нас интересует лишь путь S, получающийся в результате. Кроме того, нам протребуются три дополнительные строки, получаемые из найденного пути и вставляемой строки k: *common*, *rest* и *key*. Построение этих строк происходит следующим образом. Прежде всего, мы находим строку s', получающуюся удалением последнего префикса из строки, соответствующей найденному пути S. Из устройства процедуры поиска очевидно, что эта строка является префиксом добавляемой строки k или совпадает с ней. Теперь рассмотрим строку k', получающуюся из k удалением префикса s'. В качестве строки *common* берётся наибольший общий префикс строк k' и

$p = prefix(S[n])$, где $S[n]$ — последняя вершина пути S. В свою очередь строками $rest$ и key являются строки p и k' без префикса $common$ соответственно. Идея становится ясна по следующей схеме:



Алгоритм вставки рассматривает пять случаев:

1. В дереве нет узлов. В этом случае нам надо выделить страницу, на которой расположить единственный новый узел дерева.

2. $rest$ и key — пустые строки. В этом случае нам достаточно пометить последнюю вершину пути как $final$ (в случае минимального дерева она уже будет помечена как $final$).

3. key — пустая строка, $rest$ — непустая. В этом случае нам достаточно разбить последнюю вершину пути на две, одна их которых будет содержать префикс $common$ и будет помечена как $final$.

4. key — непустая строка, $rest$ — пустая. В этом случае нам надо добавить дополнительную вершину с префиксом key, дочернюю по отношению к последней вершине пути.

5. key и $rest$ — суть непустые строки. В этом случае последняя вершина x_n разбивается на три: одну с префиксом $common$, с двумя исходящими из неё — x_n, префиксом которой становится $rest$, и final-вершину с префиксом key.

Главный интерес в плане разделения на блоки представляют пункты 4) и 5), поскольку в случае, если ветка, в которую производится добавление, имеет дочерние ветки, мы

создаём новую вершину с key в новой ветке. Для этого мы находим среди дочерних веток ту, у которой в блоке содержащем её больше всего свободного места. Теоретически это очень дорогая операция, поскольку она затрагивает все блоки, в которых хранятся дочерние ветки. Таких блоков не больше количества дочерних веток. Но на практике подобный подход неприемлем, поскольку помимо компактности хранения, также должно быть минимизировано количество блоков, к которым происходят обращения. В связи с этим был найден компромисс, который на практике позволил удовлетворить обоим требованиям: мы ограничиваем количество просматриваемых блоков некоторой константой D[13]. В случае, если в одном из D блоков достаточно места для добавляемой вершины, вершина добавляется в наиболее свободный из них, а в случае, если среди D блоков не нашлось блока с достаточным количеством места, то запускается процедура разделения блока, после выполнения которой добавление запускается вновь. При таком подходе затрагивается не более $D + 2$ блоков[14].

4.4.3 Разделение блоков

Одним из ключевых отличий BST от Б-деревьев является то, что они не являются сбалансированными.Поскольку минимальное дерево T однозначно определяется множеством хранимых строк K (по теореме 1), на высоту дерева повлиять в сторону уменьшения невозможно. Тем не менее, существуют исследования, которые показывают, что сбалансированность дерева в общем случае не очень сильно влияет на производительность операций в дереве[7]. Под высотой BST-дерева подразумевается высота дерева его меток, так как именно ей характеризуется количество веток, которые необходимо прочитать, чтобы найти вершину в худшем случае. Для уменьшения высоты дерева в блоках, необходим механизм, который бы обеспечивал высокую заполненность блоков. Первой частью этого механизма является эффективный алгоритм вставки элементов, второй же его частью является алгоритм разделения блоков, описанный ниже.

Процедура выделения места в блоке подразумевает разделение блока и вызывает-

[13]В реализации константа D=2

[14]Без учёта возможной операции разделения

ся только в том случае, если в блоке не хватает места для вставки нового узла. Мы используем две различных стратегии разделения блока.

Первый алгоритм является наиболее предпочтительным, поскольку он ведёт к разделению блока вширь, не влияя на высоту дерева. Алгоритм занимается обработкой ситуации, когда в одном блоке содержится несколько веток. В этом случае мы разделяем ветки странице на два непересекающихся набора P_1 и P_2, такие, что $|\sum_{w \in P_1} w(B) - \sum_{w \in P_2} w(B)|$ минимальна по всем возможным разделениям P_1 и P_2. Таким образом, эти наборы должны разделять ветки примерно пополам по их общему размеру. Один из этих наборов остаётся в изначальном блоке, а второй набор веток записывается в новый блок, после чего ссылки родительского узла на ветки обновляются. Этот алгоритм затрагивает ровно три блока — один блок, содержащий родительскую вершину[15], и два блока, содержащие ветки (старый и новый).

Второй алгоритм работает в случае, если в блоке, в который происходит добавление, находится только одна ветка. В этом случае, мы находим первую вершину x от корня ветки, которая содержит $N > 1$ ссылок. Это означает, что у этой вершины есть N дочерних вершин, которые будут корнями N новых веток. Далее мы переносим все вершины от корня ветки до найденной вершины x в ветку-предок, содержащуюся в другом блоке[16]. Для того, чтобы гарантировать возможную вставку и чтобы ограничить рост дерева в высоту, оставшиеся N получившихся веток мы разделяем на два блока по первому алгоритму. Если исходная ветка являлась корнем всего дерева, то множество вершин переносится не в страницу, содержащую ветку-предок (так как её не существует), а в новый блок. Именно в этот, и только в этот момент дерево растёт в высоту. Данный алгоритм разделения затрагивает ровно три блока, причём один из этих блоков содержит ветку-предок, а два блока содержат свежесозданные ветки.

[15]Это существенный факт, который сыграет свою роль при подсчёте общего количества затрагиваемых блоков при разделении

[16]Строго говоря, нет никаких гарантий, что в странице, содержащей ветку-предок, достаточно места для переноса этого множества вершин, но мы корректно преодолеваем это возможное затруднение, об этом написано далее

Итак, единственной ситуацией, которую осталось обработать, является случай, когда в блоке содержится лишь одна ветка, и первый алгоритм неприменим, и одновременно с этим в блоке, содержащем ветку-предок, недостаточно места для того, чтобы перенести множество вершин из ветки, содержащейся в блоке, который мы пытаемся разделить. Поскольку возникновение подобной ситуации нельзя предотвратить заранее, мы прибегаем к дополнительным мерам.

Мы строим путь, ведущий к ветке, в которую предполагается совершить вставку, причём в ходе построения этого пути мы собираем некоторые вспомогательные данные, и всё это происходит ещё на этапе поиска пути в операции вставки нового элемента. В частности, мы собираем данные о том, какие блоки задействованы в этом пути, сколько этот блок содержит веток, сколько места в нём занимают множества корневых вершин, описанные во втором алгоритме, если ветка в блоке всего одна, сколько свободного места остаётся в блоке. Эти данные не вычисляются каждый раз заново, они записываются в блоки при изменениях в дереве и хранятся в качестве служебной информации. Таким образом, мы получаем некоторый путь, содержащий данные о всей цепочке от корня дерева до блока, в который производится вставка и который нам необходимо разделить. Благодаря этому пути мы заранее знаем, сколько веток в каждом блоке, сколько в них занимает корневое множество вершин и сколько места свободно, и именно эти знания позволяют заранее обрабатывать разные ситуации.

Дальнейшая процедура проста: начиная с последнего блока в этом пути по направлению к блоку, содержащему корень дерева, мы производим поиск блоков, которые необходимо разделить для того, чтобы в конечном блоке хватило места для вставки нового элемента, и делим их до тех пор, пока места не будет достаточно. За счёт введения этой вспомогательной структуры и подобного алгоритма, нам удаётся уменьшить количество блоков затрагиваемых при делении, в том числе и за счёт того, что и в первом, и во втором алгоритме затрагивается по три блока, причём в каждой из этих троек по сути затрагивается блок, стоящий на предыдущей позиции в найденном пути. Таким образом, в худшем случае всей процедурой деления блоков затрагивается $2h + 1$ блоков, где h — высота дерева.

Кроме того, мы провели довольно обширное тестирование[17] на большом количестве данных, которое указывает на то, что количество блоков, загруженных менее, чем на 30 процентов, при различных средних длинах вставляемых строк в количестве от 1000 строк и до $16 * 10^9$ строк, никогда не превышает 0.2 процентов от общего количества блоков.

4.4.4 Удаление

Следуя примеру большинства работ, посвящённых Б-деревьям, мы вводим операцию[18] «отложенного удаления»[15]. Данный подход широко распространён в применении к базам данных, поскольку процедура удаления для Б-деревьев может быть сложнее процедуры вставки. Отложенное удаление позволяет, во-первых, значительно упростить саму процедуру удаления, а во-вторых, значительно ускорить обновления базы данных.

В разработанной структуре BST удаление заключается в простом снятии флага $final$ с найденного узла. Таким образом, после удаления узла мы получаем избыточный узел и, как следствие, неминимальное дерево. При таком подходе необходима процедура минимизации дерева (или отдельной ветки). Эта процедура заключается в удалении всех избыточных вершин, которые в свою очередь, бывают двух типов:

1. Вершины, которые не помечены как $final$, и у которых нет исходящих рёбер. Такие вершины можно просто удалить.

2. Вершины, которые не помечены как $final$, из которых есть ровно одно исходящее ребро. В этом случае вершины достаточно объединить конкатенацией префикс + символ ребра + префикс.

В результате применения этой операции не остаётся избыточных вершин. В результате такой операции может получится так, что в некоторой ветке не останется вершин, либо

[17] Тестирование производилось на финальном этапе реализации структуры. В качестве строк брались случайно генерируемые цепочки символов со средней длиной от 5 символов в строке до 150. Количество вставленных сгенерированных строк составляло $16*10^9$, измерения загруженности блоков происходили на каждом 10 000 шаге.

[18] Отложенное удаление также известно под назнванием «lazy removal»

останется единственная *external*-вершина. Такую ветку необходимо удалить, а (единственную) дочернюю ветку перенести вверх в данный блок. Может оказаться так, что блок необходимо будет для этого разделить. Таким образом минимизация одной ветки затрагивает не более четырёх блоков — три блока может затронуть разделение конкретного блока, и из одного блока мы переносим ветку.

5 Описание практической части

5.1 Использованный инструментарий

Описываемая в работе структура реализована на языке Си и может быть использована отдельно в любых возможных приложениях. При этом структура написана полностью совместимой с СУБД Sedna, как того требует цель работы. Использование языка Си обусловлено несколькими причинами:

- Язык Си является высокоуровневым языком, но при этом обладает всеми возможностями для явной работы с памятью, и кроме того, этот язык является очень распространённым.
- Языки C/C++ используются для реализации большинства реляционных СУБД, поэтому выбор языка Си является также и шагом к обеспечению совместимости между реализованной структурой и другими её применениями. В связи с тем, что структура реализована без зависимостей от СУБД Sedna, то, что при написании был выбран язык Си, позволяет использовать реализацию как библиотеку ко всем СУБД, написанным на языках C/C++, без каких-либо серьёзных модификаций, равно как и к СУБД, написанным на других языках, при внесении определённых правок.

Несмотря на возможность использовать реализацию отдельно от СУБД Sedna, в рамках дипломной работы была обеспечена полная совместимость с ней. СУБД Sedna выбрана по причине того, что описываемая структура данных имеет большой потенциал для работы с XML документами, при этом СУБД Sedna является фактически уникальной среди прочих прирождённых XML-СУБД по своей производительности и в ряде тестов превосходит все прочие прирождённые XML-СУБД на 2-3 порядка по производительности[16]. Кроме того, в СУБД Sedna является наиболее полной по набору возможностей среди конкурентов. Эти причины подталкивают нас к реализации новых алгоритмов именно для неё как для самой продвинутой прирождённой XML-СУБД.

5.2 Архитектура

Общая архитектура изображена на схеме. В работе доработаны и использованы модули СУБД Sedna, отвечающие за обработку запросов и добавлена функциональность, позволяющая строить индексы не только с использованием структуры Б+ дерева, но и разработанной и реализованной структуры BST. О том, как именно это работает, написано в следующем подразделе, здесь же изображена схема, показывающая устройство реализованного программного средства.

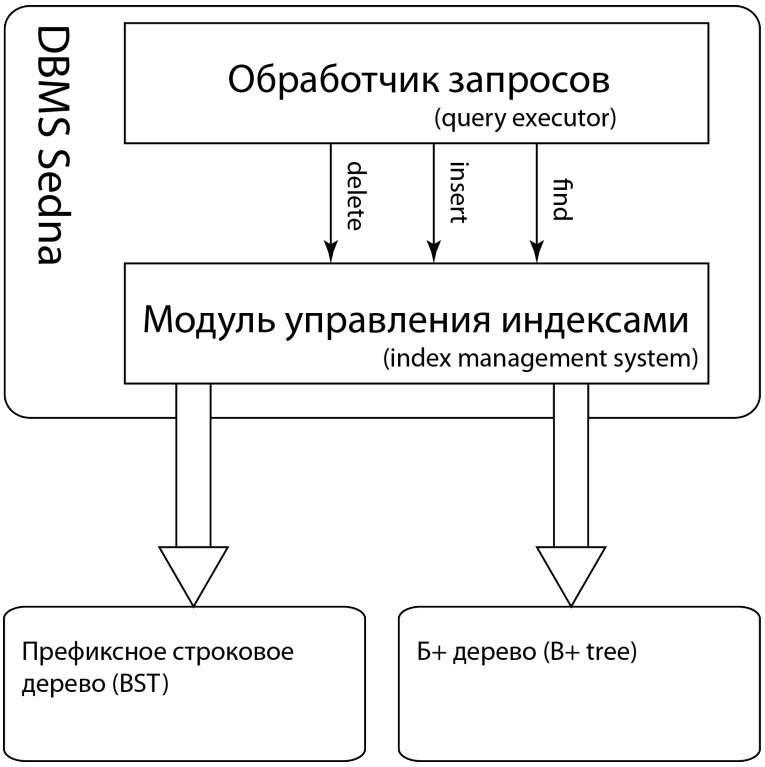


Рис. 11: Архитектура реализованного программного средства

5.3 Схема работы

Всё, что будет перечислено далее относится к работе над уже загруженным XML-документом, потому что процесс загрузки документа в базу данных не имеет отношения к работе и происходит обыкновенным образом.

Итак, обработчик запросов модифицирован таким образом, чтобы можно было выбирать, с какой структурой данных будет происходить работа. Это существенно для процесса построения индекса над документом. Запросы на построение индексов после модификации обработчика имеют подобный вид[19]:

Для Б+ дерева:

```
> create index "urlbtree[1]" on doc("dblp.xml")/descendant::inproceedings
        by ./url as xs:string using "btree"&
```

Для BST:

```
> create index "urlbstrie[1]" on doc("dblp.xml")/descendant::inproceedings
        by ./url as xs:string using "bstrie"&
```

Эти запросы передаются модулю управления индексами, который в свою очередь проводит над каждым ключом/значением следующую операцию: в конец каждого ключа вставляется нулевой символ (его нет в алфавите ключей) и присоединяет к получившейся строке строку значения. После этого эта строка передаётся программе, реализующей BST для вставки в дерево.

В случае поиска всё происходит по той же схеме, кроме того, что не нужно указывать тип хранящей данные структуры, так как индексы и так знают о своём устройстве. Поэтому мы приведем лишь строку пример запроса на поиск:

Поиск по индексу:

```
> index-scan("eebstrie[1]","http://www.aclweb.org/anthology/D07-1066","EQ")&
```

Реализованная программа имеет набор функций, который описывает функции взаимодействия с системой в понятном виде и реализует необходимые вызовы внутренних функций для обеспечения работы алгоритмов: функции добавления строки, функцию

[19]В качестве примера дана команда построения индекса над документом DBLP по полю URL, вложенному в INPROCEEDINGS

поиска строки, функцию удаления строки. Функция поиска строки вызывает функцию поиска, описанную в разделе алгоритмов с дальнейшим обходом поддеревьев, если ищется не конкретное значение, а все значения с заданным префиксом.

Функция вставки строки вызывает функцию поиска добавляемого элемента и таким образом находит место, в которое должна произойти вставка, при этом строится описанный в разделе с алгоритмами метапуть. В случае если для вставки необходимо больше места, чем свободно в блоке, запускается функция деления блоков. Эта функция подробно описана в разделе описания алгоритмов.

5.4 Характеристики функционирования и тесты

Характеристики функционирования реализованной структуры включают в себя пространство, занимаемое множеством добавленных строк, время, затрачиваемое на добавление строк и время, уходящее на поиск в структуре. Помимо измерений непосредственных характеристик функционирования реализованного префиксного строкового дерева, также были произведены измерения на тех же наборах данных и для реализованного в СУБД Sedna Б+ дерева. Для сравнения были подготовлены несколько наборов данных. В качестве набора данных для тестирования использовалась база данных DBLP Computer Science Bibliography[11]. Тестирование проводилось с использованием СУБД Sedna. Наборы данных формировались посредством создания индексов; публикации индексировались по идентификатору (key) и по двум типам URI-ссылок (EE и URL). Всего индексировалось 939613 публикаций.

Кроме того, был подготовлен тест, который представляет собой пример неудачных данных для создания индекса с помощью BST. В качестве индексируемого документа был взят автоматически сгенерированный документ, содержащий в себе случайные строки. Документ был сгенерирован с помощью инструмента XBench[12].

Для измерения характеристик производительности и занимаемого пространства для каждой структуры использовался следующий подход:

- Оценка производительности каждой операции будет измеряться в секундах. Изме-

рения времени были производены в ОС Linux на компьютере со следующей аппаратной конфигурацией: Intel Core i7 930, 6GB DDR3 RAM, Ubuntu 10.10 x86_64.

- Оценка занимаемого пространства измеряется в количестве занятых структурой блоков и в количестве занятых мегабайт[20].

- Производительность операции добавления элементов в структуры будет оцениваться через операцию создания индекса. При создании индексов одних и тех же документов в обе структуры происходили вставки одних и тех же элементов и в том же количестве, так что эти данные отражают соответствие между производительностью предложенных структур.

- Производительность операций поиска измерялась не для каждого запроса в отдельности, а для большого количества запросов, поскольку этот метод обладает большей точностью. Для каждого созданного индекса производился 93961 запрос (это 10% всех записей), каждый из запросов представляет собой некий случайных элемент, присутствующий в созданном индексе. Для индекса xbench производилось 42420 запросов, это также 10% от присутствующих в индексе записей.

- Каждый тест был повторён трижды — по одному разу для буфров с размерами в 640 килобайт, 100 мегабайт и 2 гигабайта соотвественно. Тесты, проведенные с данными размерами буферов, наглядно показывают, как работают структуры данных в случае работы только с внешней памятью, со стандартным буфером в оперативной памяти и в случае, когда вся база данных вместе с индексами помещается в оперативную память.

[20] Численное выражение в занятых мегабайтах будет приводиться исключительно из соображений простоты восприятия — эта величина всегда будет равна количеству занятых блоков помноженных на размер блока.

Таблица 1: Сравнение характеристик префиксного строкового дерева и Б+ дерева

Набор данных	Объём данных (мб)		Объём данных (блоки)		Время создания индекса (сек.)						Время поиска (сек.)					
					буфер 640кб		буфер 100мб		буфер 2гб		буфер 640кб		буфер 100мб		буфер 2гб	
Intel Core i7 930, 6GB DDR3 RAM, Ubuntu 10.10 x86_64																
DBLP KEY	29	33	464	528	22.7	37.6	14.9	21.3	14.5	21.3	859	936	216	221	101	96
DBLP EE	18.6	47.1	298	752	50.9	54.3	12.1	20.6	11.7	18.1	596	587	170	175	77	81
DBLP URL	33.1	49	531	784	23.9	37.6	15.7	24.4	15.1	23.9	816	736	228	235	175	184
XBench	77.5	64.1	1240	1026	489	470	10.7	8.1	8.5	7.2	230.1	226.8	75.2	74.8	73.1	72.8
	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево	BST-дерево	Б+ дерево

6 Заключение

В ходе проведённой работы была разработана, проанализирована и реализована структура для поиска и хранения строк произвольно большой длины. Данная структура хорошо подходит для хранения индексов и поиска по ключам большой длины, обладающим схожей структурой, и позволяет компактно хранить данные без существенных потерь в скорости поиска по ним. Выигрыш в занимаемом пространстве характеризуется устройством структуры. Также разработанная структура поддерживает хранение данных в блоках фиксированного размера.

Для разработанной структуры реализованы эффективные алгоритмы поиска, вставки и удаления, а также обеспечивается совместимость с XML-СУБД Sedna. Кроме того, было произведено тестирование реализованной структуры на нескольких наборах данных по разным параметрам. Тесты показывают, что структура хорошо подходит для использования с индексами по полям, содержащим ссылки или строки, в которых могут присутствовать одинаковые префиксы. Кроме того, тесты показывают, что структура показывает неплохие результаты и в общем случае.

В дальнейшем структуру можно использовать в любых приложениях, а не только в рамках СУБД Sedna.

Список литературы

[1] *Comer D.* The ubiquitous b-tree // *ACM Computing Surveys.* — 1979. — Vol. 11, no. 2. — Pp. 121–137. http://portal.acm.org/citation.cfm?doid=356770.356776.

[2] *Bayer R., Unterauer K.* Prefix b-trees // *ACM Trans. Database Syst.* — 1977. — Vol. 2, no. 1. — Pp. 11–26.

[3] *Кнут* Искусство программирования. — 2 изд. — Вильямс, 2004. — Т. 3. — С. 300–303.

[4] *Szpankowski W.* Patricia tries again revisited // *J. ACM.* — 1990. — Vol. 37, no. 4. — Pp. 691–711.

[5] *Heinz S., Williams H., Zobel J.* Burst tries: A fast, efficient data structure for string keys: Tech. rep. — GPO Box 2476V, Melbourne 3001, Australia: School of Computer Science and Information Technology, RMIT University, 2002.

[6] *Chan Y.-K., Chang C.-C., Shen J.-J.* A compact patricia trie for a large set of keys. // ISDB'02. — 2002. — Pp. 31–36.

[7] *Askitis N., Zobel J.* B-tries for disk-based string management // *The VLDB Journal.* — 2009. — Vol. 18. — Pp. 157–179.

[8] *Bayer R., Bayer R., Mccreight E.* Organization and maintenance of large ordered indexes // *Organization.* — 1972. — Vol. 1. — Pp. 173–189.

[9] Sedna: native xml database management system (internals overview) / I. Taranov, I. Shcheklein, A. Kalinin et al. // SIGMOD Conference / Ed. by A. K. Elmagarmid, D. Agrawal. — ACM, 2010. — Pp. 1037–1046.

[10] A mapping mechanism to support bitmap index and other auxiliary structures on tables stored as primary b^+-trees / E. I. Chong, J. Srinivasan, S. Das et al. // *SIGMOD Record.* — 2003. — Vol. 32, no. 2. — Pp. 78–88.

[11] *Ley M.* Die trierer informatik-bibliographie dblp // GI Jahrestagung. — 1997. — Pp. 257–266. `http://dblp.uni-trier.de/`.

[12] *B. B. Yao M. T. , Khandelwal N.* Xbench benchmark and performance testing of xml dbmss // Proceedings of 20th International Conference on Data Engineering. — 2004. — Pp. 621–632.

[13] *Baeza-Yates R. A.* An adaptive overflow technique for b-trees // EDBT. — 1990. — Pp. 16–28.

[14] *Таранов* Использование префиксного дерева для хранения и поиска строк во внешней памяти // Труды Института системного программирования РАН. — 2011. — Pp. 283–296.

[15] *Johnson T., Shasha D.* B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half // *J. Comput. Syst. Sci.* — 1993. — Vol. 47, no. 1. — Pp. 45–76.

[16] *Hall D.* — An XML-based Database of Molecular Pathways. — Master's thesis, Linkoepings Universitet, 2005.