Использование префиксного дерева для хранения и поиска строк во внешней памяти

Таранов И. С.
epsilon@ispras.ru

Аннотация. Поиск среди больших объёмов текстовых данных, хотя и изучается в computer science давно, не теряет своей актуальности. В работе представлена структура данных для поиска и эффективного хранения во внешней памяти массивов текстовых строк, реализованная для поддержки индексов в XML СУБД Sedna. Описываются алгоритмы для вставки, удаления и поиска строк переменной длинны в префиксных деревьях, хранимых на дисках. Мы также сравниваем нашу реализацию с существующей реализацией B-дерева. В работе показано, что в некоторых случаях предложенная структура данных занимает в несколько раз меньше места во внешней памяти при той же скорости поиска.

Ключевые слова: словарные структуры данных; префиксные деревья; B-деревья; индексы в СУБД

1. Введение

Проблемы реализации структур данных, реализующих основные словарные операции (добавление, удаление и поиск элемента), хотя и изучаются в computer science очень давно, тем не менее, не теряют своей актуальности. В данной работе описана структура данных, предназначенная для хранения множества строковых ключей во внешней памяти, приведено её сравнение с B-деревьями, и описаны основные алгоритмы работы с ней. Предложенная структура данных в некоторых случаях позволяет значительно сократить объём индекса во внешней памяти.

Задача разработки специальной структуры данных встала в ходе работы над XML-СУБД Sedna [1, 2]. Данная XML-СУБД поддерживает индексы по значению узлов над XML-документом. В результате к структуре данных, используемой для поддержки индексов, предъявляются следующие требования:

1. Возможность хранить ключи произвольно большой длины. Необходимость этого связана с подходом к хранению XML, где все

значения при отсутствии предписывающей схемы являются строками, при таком хранении нельзя заранее ограничить длину строки. Одним из примеров длинных строк являются URI, поиск по которым зачастую необходим, но которые при этом могут быть достаточно длинными.

2. Возможность хранить мультимножество пар "ключ/значение". В произвольном XML-документе при произвольном индексе могут существовать одинаковые ключи с разным значением и одинаковые значения с одинаковыми ключами. Причём при обновлении документа (например, удалении узлов) может быть удалена только одна из одинаковых пар "ключ/значение".
3. Эффективный поиск пары "ключ/значение" необходим для быстрого удаления пары из индекса при обновлении документа.

2. Обзор существующих работ

Стандартной структурой данных для создания такого индекса является *B-дерево* [3] и его варианты [4, 5, 6] [1]. B-деревья очень широко применяются в базах данных для задач индексации [8]. Хранение ключей произвольной длины в B-деревьях хотя и возможно, однако представляет ряд проблем. Во-первых, достаточно сложно реализовать эффективное хранение ключей переменной длины. На практике обычно в узле дерева хранят только ограниченную часть ключа, а остаток хранят в отдельных *страницах переполнения (overflow pages)* [9, 10]. Такой подход достаточно эффективен в тех случаях, когда ключи короткие или различаются в первых символах. Во-вторых, в случаях, когда одному ключу соответствует множество различных значений, сложно реализовать достаточно эффективный поиск по паре "ключ/значение". Часто B-деревья не предусматривают хранения мультимножества ключей. Для поставленной же задачи характерно наличие дубликатов пар "ключ/значение". Для возможности хранения одинаковых логических ключей в качестве физического ключа используется конкатенация пары "ключ/значение" [11][2] .

В статье предложен подход к организации индексов над строками во внешней памяти, который удовлетворяет всем вышеописанным требованиям. Его

[1] Во всей статье под B-деревьями на самом деле подразумеваются B^+деревья [7].

[2] Здесь мы будем понимаем под физическим ключом тот ключ, который действительно используется при сравнении и поиске в описываемой структуре данных. Под логическим ключом понимается ключ, который передаётся интерфейсу работы с данной структурой данных.

можно рассматривать как расширение структуры данных, известной как trie [12], которую в дальнейшем будем называть *префиксное дерево*[3].

Идея использования данной структуры для реализации индексов не нова. В работе [14] предложена структура данных S(b)-tree, которая представляет собой разновидность B-дерева, в узлах которого находится бинарная *"Патриция" (patricia tree)* [15]. Особенностью S(b)-tree является то, что в узлах хранятся не ключи как таковые, а количество пропускаемых при сравнении бит. В процессе поиска искомую строку, возможно, придётся сравнивать со строкой, хранящейся во внешней памяти. Однако само по себе такое сравнение не представляет больших проблем. Для поставленной задачи проблемой было бы хранить все строки-ключи в отдельном месте. S(b)-tree позиционируется как структура данных для поддержки полнотекстового индекса и достаточно хорошо справляется с этой задачей [16].

Наиболее похожей на описанную в работе структуру данных является предложенная в работе B-trie [17]. За основу в этой работе была взята реализация префиксного дерева от тех же разработчиков, которая предусматривает эффективное использование процессорного кеша [18]. В обеих структурах предложено эффективное для этого разбиение префиксного дерева на блоки (buckets).

Кроме того, существуют совсем простые реализации подобного подхода, такие как Index Fabric [19], который является тем же B-деревом, в узлах которого хранение и поиск ключей осуществляется с помощью префиксного дерева.

3. Описание предложенной структуры

В статье предложена структура данных для хранения множества строк *K*, которую в дальнейшем мы будем называть **BST** (**B**lock **S**tring **T**rie), и над которой определены следующие (словарные) операции:

- *insert*(*s*) - добавить строку *s* в множество *K*;
- *delete*(*s*) - удалить строку *s* из множества *K*;
- *find*(*s*) - найти все строки в *K* с префиксом *s*, включая саму строку *s*.

Данная структура данных, как видно, сама по себе не предусматривает хранения пар "ключ/значение". Учитывая наши требования, мы будем хранить значения, как части физически сохраняемого ключа. Рассмотрим пример. Пусть надо сохранить пару (k, v). Для этого мы строим строковый ключ $k' = k + c + string(v)$, где под c понимается символ, которого нет в алфавите символов логических ключей (так, для текстовых строк можно

[3] В русском переводе [13] для перевода термина trie используется слово "луч". Также в литературе встречается слово "бор".

использовать нулевой символ), а $string(v)$ — любое строковое представление значения. Если надо найти все значения, соответствующие некоторому ключу k, необходимо вызвать $find(k + c)$.

В нашей структуре данных мы разделяем несколько уровней объединения вершин дерева. Внутренние вершины дерева объединяются в *ветки*, а ветки целиком хранятся в страницах внешней памяти. Ниже мы опишем последовательно эти уровни.

3.1. Префиксное дерево

Описываемая структура данных представляет собой *корневое дерево T*, хранящее множество ключей *K* и устроенное следующим образом:

1. Каждая вершина x содержит следующие поля: префикс $prefix(x)$ (возможно, пустой), массив $E(x)$ из $n(x)$ исходящих из неё рёбер, помеченных *различными* символами (упорядоченный в лексикографическом порядке), а также некоторые флаги (булевские переменные), например $final(x)$, который определяет, соответствует ли этой вершине какой-либо ключ. Флаги вершины будут описываться по мере их введения.
2. Ребра $e = (x, L_i(x))$ (будем обозначать $L_i(x)$ вершину, в которую ведёт i-е ребро из массива $E(x)$) помечены символами $c(e)$. В данном случае мы не отличаем отдельные символы от строк и будем считать их строками единичной длины.
3. Любой путь $S(x_1, x_n) = x_1 e_1 x_2 e_2 \text{K}\ e_{n-1} x_n$ в дереве задаёт строку s, которая получается из этого пути S конкатенацией строк $s = prefix(x_1) + c(e_1) + prefix(x_2) + c(e_2) + \text{K} + c(e_{n-1}) + prefix(x_n)$ Строка s, заданная путём S, принадлежит хранимому в дереве множеству ключей K в том и только в том случае, если конечная вершина x_n пути S помечена флагом $final(x_n)$.

Отметим, что в общем случае некоторому множеству ключей K может соответствовать более одного дерева T. Это объясняется тем, что в дереве, в котором есть хотя бы одна вершина, из которой исходят не все возможные рёбра, можно добавить вершину x' с произвольным префиксом, не помеченную $final(x')$. Полученное дерево будет задавать то же самое множество ключей K. Поэтому введём дополнительное свойство,

выполнения которого требовать мы не будем, позволяя реализовать тем самым *отложенное удаление* (которое будет обсуждаться ниже):

1. Любая вершина $x \in T$, для которой $n(x) \le 1$, помечена как $final(x)$. Т.е. (единственному) пути, ведущему от корня к такой вершине, соответствует ключ множества K.
2. Заметим, что при выполнении последнего свойства множество K однозначно задаёт дерево T. Доказательство этого факта не представляет сложности, однако выходит за рамки данной статьи.

Дерево T, для которого выполняется последнее свойство, будем называть *минимальным*. Кроме того, вершину x, для которой выполняется (не выполняется) условие $n(x) \le 1 \Rightarrow final(x)$, будем называть *неизбыточной* (*избыточной*). В дальнейшем (если не оговорено обратного) будем рассматривать только минимальные деревья.

Заметим также, что для хранения мультимножества достаточно ввести дополнительный параметр вершины $count(x) \ge 1$, определённый только для вершин, для которых установлен флаг $final(x)$.

3.2. Разделение на страницы

Описанная в предыдущем разделе структура удобна для хранения и поиска строковых ключей в оперативной памяти. Но для использования её с большими объёмами данных во внешней памяти необходимо эффективно распределять её на страницы фиксированной длины. Заметим, что это важно не только для структур данных, предназначенных непосредственно для хранения во внешней памяти (например для поддержания индексов баз данных), но также для структур данных в оперативной памяти [13].

Для такого разделения выделим особый тип *ссылочных вершин*, которые не хранят префиксов и ссылок на другие вершины, а хранят лишь ссылку на другой узел, находящийся в другой странице. Такие вершины пометим флагом $external(x)$, каждая из которых ссылается на некоторую вершину $y = J(x)$ такую, что $external(x) \Rightarrow not\ external(J(x))$. Введём также понятие ветки. Назовём *веткой корневое дерево* B такое, что *конечные вершины всех путей от корня к листовым вершинам* в нём помечены либо $final(x)$, либо $external(x)$. Если в дереве T нет ссылочных вершин, то оно состоит из одной ветки.

Заметим, что любое дерево T может произвольным образом быть разбито на ветки путём вставки перед любой вершиной новой ссылочной вершины. Ссылочная вершина разбивает ветку на две. Если ветки считать узлами, то они

также образуют дерево T_b. Отметим также, что ссылочные вершины никак не влияют на минимальность дерева *по определению* и не являются избыточными.

Страницы могут хранить одинаковый объём данных *W* байт[4]. В странице могут храниться одна или несколько веток исходного дерева, причём в одной странице могут храниться только ветки с общим прямым предком *P*(*B*). Последнее условие необходимо, во-первых, для обеспечения локальности изменений в дереве [20], а, во-вторых, для обеспечения эффективных блокировок на уровне страниц. Из этого также следует, что на странице, в которой хранится корневая ветка, других веток быть не может.

4. Алгоритмы

Алгоритмы изменения дерева должны обеспечивать, во-первых, локальность изменений (ограниченное число изменяемых страниц), во-вторых, компромисс между оптимальным заполнением страниц и высотой дерева.

4.1. Поиск пути

Алгоритм BST-Search(r, k) возвращает вершину x_{n+1} такую, что:

- строка $s(x_n)$, задаваемая путём $S(r, x_n)$, если такая существует, является префиксом искомой либо совпадает с ней,
- искомая строка k является префиксом строки $s(x_{n+1})$ либо совпадает с ней.

Т.е. выполняется неравенство: $s(x_n) \le k \le s(x_{n+1})$, где под операцией $A \le B$ понимается то, что A является префиксом B или совпадает с ней. Процедура получает на вход указатель x на корневой узел поддерева, а также строку k. Промежуточные результаты поиска сохраняются в стек S. В процедуре также используется 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 NIL, если такого ребра нет. Такую функцию можно реализовать, например, бинарным поиском, т.к. массив рёбер упорядочен. Также используется функция $Cut(p, s)$, которая возвращает строку, получающуюся из s удалением префикса p.

Алгоритм достаточно компактен, поэтому приведём его здесь:

[4] В СУБД Sedna, для которой реализована описываемая структура, размер страницы по умолчанию равен 64K.

```
1.  BST-Search(x, k, S)
2.  Push(S, x)
3.  if external(x) then
4.      Disk-Read(J(x))
5.      return BST-Search(J(x), k, S)
6.  endif
7.  if not Is-Prefix( prefix(x), k)) then
8.      if Is-Prefix(k, prefix(x)) then
9.          return x
10.     else
11.         return NIL
12.     endif
13. else
14.     s ← Cut( prefix(x), k)
15.     if Empty(s) then
16.         return x
17.     elseif L(x, s[1]) = NIL
18.         return NIL
19.     else
20.         return BST-Search(L(x, s[1]), Cut(s[1], s), S)
21.     endif
22. endif
```

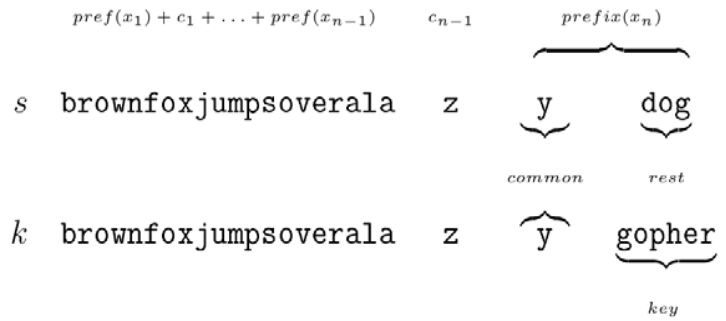
Процедура возвращает такой узел x, что все пути, ведущие из корня в $final$-вершины и содержащие x, задают строки, для которых искомая строка является префиксом, либо NIL, если такого узла нет. Таким образом, нам остаётся найти все эти пути (обходом поддерева от возвращённой вершины).

4.2. Вставка

Процедура добавления строки устроена таким образом, что она сначала строит структуру, описывающую изменение страницы, в которую надо вставить новые вершины. Может случиться, что для вставки нет места в странице. В этом случае процедура вызывает расширение дерева на необходимую величину, и вставка не происходит. Она должна быть вызвана ещё раз. Заметим также, что корень дерева может измениться в процессе работы процедуры.

Сама процедура добавления достаточно проста, однако она слишком велика, чтобы приводить её здесь. Опишем только основной подход.

Начинается вставка с поиска пути по дереву к ключу k, который надо вставить. Алгоритм практически идентичен алгоритму поиска пути, только нас интересует лишь путь S, получающийся в результате. Кроме того, нам понадобятся три дополнительные строки, получаемые из найденного пути и строки k: $common$, $rest$ и key. Строятся они следующим образом. Возьмём строку s, которая получилась из строки, соответствующей найденному пути S удалением последнего префикса. Она, очевидно, является префиксом добавляемой строки k (либо совпадает с ней). Будем рассматривать строку k, которая получилась из k удалением этого префикса s. Тогда $common$ — это наибольший общий префикс строк k и $p = prefix(S[n])$ (где $S[n]$ — последняя вершина пути S). $rest$ — это остаток от строки p, полученный удалением префикса $common$, и key — остаток от k, полученный удалением префикса $common$. Все три строки удобно считать одновременно, и их значение иллюстрирует следующая схема:



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*.

Основной момент, который связан с разделением на блоки, заключается в том, что в случае 4 и 5, если ветка, в которую мы производим добавление, имеет дочерние ветки, мы создаём новую вершину с *key* в новой ветке. Для этого мы находим среди дочерних веток ту, на странице которой осталось больше всего места. Это самая дорогая из всех приведённых операций, т.к. требует в худшем случае чтение такого количества страниц, которое равно количеству различных возможных дочерних веток. На практике это неприемлемо. Поэтому в нашей реализации мы ограничиваем количество просматриваемых страниц некоторой константой *D* (в реализации равна 2). В случае, если среди первых просмотренных *D* страниц не нашлось той, в которую помещается добавляемая вершина, пытаемся расширить самую занятую из просмотренных. При таком подходе затрагивается не более *D*+2 страниц.

4.3. Разделение страниц

В отличие от B-деревьев, предложенные нами BST-деревья не сбалансированы. Заметим, что так как минимальное дерево T полностью определяется множеством ключей K, которые оно хранит, мы не можем никак повлиять на его высоту. Под высотой BST-дерева подразумевается высота дерева его веток, т.к. именно этим определяется количество блоков, которые необходимо прочитать, чтобы найти вершину в худшем случае. Кроме того, нашей задачей является избежать большого числа “полупустых” блоков.

Процедура выделения места в странице подразумевает разделение страницы и вызывается только в том случае, если на странице не хватает места для вставки нового узла. Мы используем два очень простых алгоритма разделения страниц.

Первый из них вызывается в случае, если на странице расположено несколько веток. В этом случае мы разделяем ветки страницы 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. Т.е. эти наборы должны разделять ветки примерно пополам по их общему размеру. Это можно сделать, например, с помощью жадного алгоритма. Каждый из этих наборов записывается в отдельную страницу, а ссылки обновляются. Данная операция затрагивает ровно три блока.

Второй алгоритм вызывается в случае, если на странице, в которую происходит добавление, осталась только одна ветка. В этом случае мы

выделяем корень P_p этой ветки следующим образом. Находим первую вершину от корня, которая содержит $N > 1$ ссылок. У этой вершины есть N дочерних вершин, которые будут корнями N новых веток, оставшихся на данной странице. Корневое множество вершин P_p мы удаляем из данной страницы и целиком переносим в ветку-предок. Чтобы гарантировать возможную вставку в исходную страницу, применяем к ней первый алгоритм разделения. Если исходная ветка являлась корневой, то множество вершин P_p помещается в новую страницу. Может оказаться, что в странице, хранящей ветку-предок, не хватит места для вставки P_p. В этом случае вызывается процедура выделения свободного места для страницы предка. Данная процедура затрагивает ровно три страницы без учёта возможного рекурсивного вызова.

Проблемой могут оказаться строки, по размеру превышающие h страниц, где h — высота дерева в страницах. В этом случае мы можем не найти в ветке вершину, которую можно “перенести” в верхнюю ветку. Тогда, в нашей реализации, строка разбивается на две (возможно нарушая минимальность дерева), и листовую часть строки представляется правильным вынести в отдельную страницу. Это практически является аналогом страниц переполнения для B-деревьев, но не будет приводить к постоянному чтению этой страницы, т.к. любая исходная строка сравнивается в нашем дереве не более одного раза.

4.4. Удаление

Следуя примеру большинства работ по B-деревьям, мы вводим процедуру отложенного удаления [20, 21, 22]. Данный подход широко используется в базах данных, т.к. процедура удаления для B-деревьев может быть сложнее процедуры вставки. Отложенное (lazy) удаление позволяет, во-первых, значительно упростить само удаление; во-вторых, ускорить обновления базы данных. Удаление заключается в простом снятии флага *final* с найденного узла. После удаления узла таким образом мы получаем *избыточный* узел, и, следовательно, неминимальное дерево.

При таком подходе необходима процедура минимизации дерева (или, возможно, отдельной ветки). Она заключается в удалении всех избыточных вершин. Избыточные вершины 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 префиксов через символ ребра.

В результате применения этой операции не остаётся избыточных вершин. В результате такой операции может получится так, что в некоторой ветке не останется вершин, либо останется единственная *external*-вершина. Такую ветку необходимо удалить, а (единственную) дочернюю ветку перенести вверх на данную страницу. Может оказаться так, что страницу необходимо будет для этого разделить. Таким образом минимизация одной ветки затрагивает не более четырёх страниц (три страницы может затронуть разделение, и с одной страницы мы переносим ветку)

5. Эксперименты

В качестве набора данных для тестирования мы использовали базу данных DBLP [23]. Тестирование производилось с использованием СУБД Sedna. При тестировании индексировались публикации по идентификатору (ID) и по двум типам URI-ссылок (EE и URL), которые есть в базе DBLP. Всего индексировалось 861473 публикации. Результаты тестирования показаны в следующей таблице:

Набор данных	Объем данных	Время поиска (сек.)	
		Большой буфер	Маленький буфер
BST			
DBLP ID	27Mb	6.0	6.2
DBLP EE	17Mb	15.9	16.0
DBLP URL	31Mb	15.7	17.0
B-дерево			
DBLP ID	31Mb	6.0	6.0
DBLP EE	44Mb	16.0	16.0
DBLP URL	46Mb	16.0	17.0

Каждая серия поисковых запросов выполнялась в двух условиях: при большом буфере оперативной памяти (большая часть дерева помещалась в памяти) и при маленьком буфере (в памяти помещались всего 32 страницы). В тестах производился поиск 10% всех возможных значений каждого набора в случайном порядке. Как видно, предложенные деревья BST показывают практически одинаковую производительность по сравнению с B-деревьями, однако занимают в некоторых случаях гораздо меньше места за счёт сжатия ссылок. Кроме того, из результатов можно сделать вывод о том, что количество сравнений строк не оказывает существенного влияния на производительность. За счёт того, что они занимают меньше места, BST-деревья медленнее растут в глубину, что может в некоторых случаях серъёзно повлиять на производительность. Однако для того, чтобы это продемонстрировать, нужно генерировать большие искусственные наборы данных.

Кроме того, было произведено сравнение скорости вставки в B-деревья и BST-деревья. Различие скорости вставки было также практически неразличимо для двух типов деревьев, поэтому мы не приводим здесь его результатов.

В работе [17] очень похожая структура данных сравнивается с различными реализациями B-деревьев (их префиксным вариантом и Berkeley B-Tree). Описанное в работе B-дерево принципиально не отличается от B-деревьев, используемых в СУБД Sedna (со всеми его особенностями). Результаты этой работы также показывают, что префиксные деревья во внешней памяти и B-деревьях отличаются по скорости поиска незначительно. Кроме того, как и отмечалось в данной работе, видно, что скорость поиска в хранимых вариантах префиксных деревьев в большей степени зависит от размера буфера оперативной памяти. Основным преимуществом нашей структуры данных по сравнению с предложенной в работе [17] является то, что нам удалось достичь гораздо более значительного сжатия похожих ключей за счёт хранения общих префиксов. В итоге это может означать более медленный рост дерева в высоту.

6. Заключение

Структура данных BST была реализована в качестве возможной структуры данных для индексов в СУБД Sedna. Существует ещё много возможностей её оптимизации (в основном, алгоритмов разделения), которые необходимо дополнительно рассмотреть в будущем. В настоящее время проводится достаточно много работ, связанных с использованием префиксных деревьев.

Различные эксперименты с разными наборами данных подтверждают, что структура данных BST показывает производительность не хуже, чем у B-деревьев. Поэтому она может использоваться как полная их замена для строковых ключей. При этом, в случае, если строки достаточно длинные и по своей природе могут иметь длинные общие префиксы, можно достичь значительного сжатия индекса. Например, значительного выигрыша можно достичь при хранении индекса по нумерующим числам в XML-документах [24] и быстрого поиска по ним, например для поддержки блокировок [25].

Однако стоит отметить, что реализация BST достаточно сложна, и при выборе структуры данных для каких-то задач в общем случае есть смысл использовать хорошо проверенные и более надёжные реализации B-деревьев.

Литература

[1] Andrey Fomichev and Maxim Grinev and Sergei D. Kuznetsov. Sedna: A Native XML DBMS. SOFSEM, pages 272-281, 2006.

[2] Ilya Taranov et al. Sedna: native XML database management system (internals overview). *SIGMOD Conference*, pages 1037-1046, 2010.

[3] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173-189, 1972.

[4] Rudolf Bayer and Karl Unterauer. Prefix B-Trees. *ACM Trans. Database Syst.*, 2(1):11-26, 1977.

[5] Nikolaus Walczuch and Herbert Hoeger. Using individual prefixes in B^+-trees. *Journal of Systems and Software*, 47(1):45-51, 1999.

[6] Ratko Orlandic and Hosam M. Mahmoud. Storage Overhead of O-Trees, B-Trees and Prefix B-Trees: A Comparative Analysis. *Int. J. Found. Comput. Sci.*, 7(3):209-226, 1996.

[7] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121-137, 1979.

[8] Eugene Inseok Chong et al. A mapping mechanism to support bitmap index and other auxiliary structures on tables stored as primary B^+-trees. *SIGMOD Record*, 32(2):78-88, 2003.

[9] Ricardo A. Baeza-Yates. An Adaptive Overflow Technique for B-trees. *EDBT*, pages 16-28, 1990.

[10] B. Srinivasan. An Adaptive Overflow Technique to Defer Splitting in B-Trees. *Comput. J.*, 34(5):397-405, 1991.

[11] SQLLite File Format: B-Tree Structures. http://www.sqlite.org/fileformat.html#btree_structures

[12] Walter A. Burkhard. Hashing and Trie Algorithms for Partial Match Retrieval. *ACM Trans. Database Syst.*, 1(2):175-187, 1976.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[14] Paolo Ferragina and Roberto Grossi. The String B-tree: A New Data Structure for String Search in External Memory and Its Applications. *J. ACM*, 46(2):236-280, 1999.

[15] Wojciech Szpankowski. Patricia Tries Again Revisited. *J. ACM*, 37(4):691-711, 1990.

[16] Joong Chae Na and Kunsoo Park. Simple Implementation of String B-Trees. *SPIRE*, pages 214-215, 2004.

[17] Nikolas Askitis and Justin Zobel. B-tries for disk-based string management. *VLDB J.*, 18(1):157-179, 2009.

[18] Steffen Heinz and Justin Zobel and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192-223, 2002.

[19] Neal Sample and Brian F. Cooper and Michael J. Franklin and Gsli R. Hjaltason and Moshe Shadmon and Levy Cohe. Managing Complex and Varied Data with the IndexFabric(tm). *ICDE*, pages 492-493, 2002.

[20] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[21] Theodore Johnson and Dennis Shasha. B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than Merge-at-Half. *J. Comput. Syst. Sci.*, 47(1):45-76, 1993.

[22] Garcia-Molina, Hector and Ullman, Jeffrey D. and Widom, Jennifer. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[23] Michael Ley. Die Trierer Informatik-Bibliographie DBLP. *GI Jahrestagung*, pages 257-266, 1997.

[24] N.A. Aznauryan, S.D. Kuznetsov, L.G. Novak, and M.N. Grinev. SLS: A numbering scheme for large XML documents, *Programming and Computer Software*, vol. 32, Jan. 2006, pp. 8-18.

[25] Peter Pleshachkov and Petr Chardin and Sergei D. Kuznetsov. A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. *ADBIS 2005*, Tallinn, Estonia, September 12-15, 2005, pp. 268-282