Информационные технологии

УДК 004.91+519.256

А. Б. Веретенников, Ю. С. Лукач

CLB-деревья: новый способ индексации больших массивов текстов

Введение

В последние годы резко повысилась актуальность обработки больших объемов разнородной текстовой информации. В первую очередь это связано с лавинообразным развитием Интернета, которое привело к появлению громадного количества текстов, представленных самым различным образом: в виде обычных текстов, HTML- и XML-документов, сообщений электронной почты и пр. В частности, юридическая, патентная и новостная информации в Интернете исчисляются уже терабайтами. В этой ситуации существенно возрос интерес к классификации подобных массивов документов и быстрому поиску в них нужной информации. Под поиском слов в массиве текстов в дальнейшем мы будем понимать нахождение имен всех документов, содержащих искомые слова, и списка позиций найденных слов в этих документах.

Очевидно, что для обеспечения быстрого поиска в подобных массивах необходимо комбинировать методы внешнего и внутреннего поиска. Структуры данных для каждого из этих видов поиска изучены достаточно хорошо, однако их эффективные комбинации начали рассматриваться в литературе только в последние годы.

Хорошо известно, что для хранения и индексации данных во внешней памяти обычно используются либо инвертированные файлы [1], либо B-деревья и их вариации [2, 3]. С другой стороны, для быстрого поиска во внутренней памяти чаще всего применяются цифровые деревья типа Patricia [4], суффиксные деревья [5, 6], суффиксные массивы [7, 8] и тернарные деревья поиска [9]. Представляется логичным строить новые комбинированные структуры на основе уже апробированных. Первой из известных нам попыток создания такой комбинированной структуры были строковые B-деревья – комбинация

B-дерева и структуры Patricia, предложенная П. Феррагина и Р. Гросси в работах [10] и [11].

Поэтому мы начинаем с краткого рассмотрения строковых B-деревьев, а затем предлагаем новую структуру – CLB-дерево (от Cluster List + B-дерево), представляющую собой комбинацию B-дерева и тернарных деревьев с хранением информации о словах в цепочках связанных блоков – кластеров. Постановка задачи и реализация морфологического анализатора выполнена Ю. С. Лукачом, разработка концепции CLB-дерева является совместной, реализация CLB-дерева выполнена А. Б. Веретенниковым.

Хорошо известно, что инвертированные файлы обеспечивают быстрый поиск в индексированном массиве данных, но требуют большого времени на перестроение. С другой стороны, B-деревья легко модифицируются, но проигрывают инвертированным файлам в скорости поиска. Поэтому CLB-деревья изначально проектировались так, чтобы обеспечить поиск со скоростью не ниже, чем в инвертированных файлах, и модификацию за время, сравнимое с модификацией B-деревьев. Второй особенностью CLB-деревьев является то, что они предназначены для обработки текстов на естественных языках. Поэтому они учитывают морфологию этих языков, что дает двойной выигрыш. Во-первых, мы получаем возможность поиска слова по любой из его морфологических форм; во-вторых, за счет хранения только базовых форм слов мы можем хранить весь словарь языка в оперативной памяти, что резко ускоряет поиск нужной страницы данных во внешней памяти.

CLB-дерево предназначается как для поиска отдельных слов, так и для поиска словосочетаний в текстах. В данной статье рассматриваются только общая организация CLB-дерева и применение его для поиска отдельных слов. Более подробное описание управления памятью, схемы кэширования данных и результаты тестов в сравнении с другими известными системами будут приведены в отдельной работе.

1. Строковые B-деревья

Строковое B-дерево предназначено для поиска заданной строки текста в некотором наборе строк. Строки этого набора хранятся во внешней памяти, и каждая из них имеет некоторый указатель фиксированного размера, по которому ее можно извлечь в оперативную память. Дерево является сбалансированным, каждый его узел хранится в отдельной странице во внешней памяти. В листах дерева хранится заданный набор строк в лексикографическом порядке, причем хранятся не сами строки, а указатели на них. Отдельное хранение строк позволяет работать со строками неограниченной длины. В каждом узле дерева, за исключением корня дерева, хранятся от $[m/2]$ до

m указателей на строки, где m – некоторое число, зависящее от размера страницы внешней памяти. Каждый узел дерева хранится в одной странице внешней памяти.

Дерево, построенное на основе некоторого набора строк D, выглядит следующим образом. D лексикографически упорядочивается по возрастанию и разбивается на части таким образом, что в каждой части получается от $[m/2]$ до m строк. Каждая часть помещается в отдельный лист дерева в виде набора указателей. В промежуточный узел дерева помещаются указатели наименьшей и наибольшей строк и номер страницы внешней памяти для каждого дочернего узла.

Подобная организация позволяет искать заданную строку путем последовательного спуска от корня дерева к листьям. Чтобы определить тот нижележащий узел, к которому следует спуститься необходимо определить, на каком месте (относительно лексикографического порядка) должна находиться искомая строка в том наборе строк, который хранится в текущем узле. Так как в узле хранятся не сами строки, а указатели на них, для сравнения искомой строки с какой-либо строкой в узле необходимо загрузить строку из внешней памяти. Для организации хранения строк внутри узла дерева используется Patricia. С ее помощью для определения позиции строки в узле требуется загрузить во внешнюю память только одну строку.

При этом требуется, чтобы среди всех строк, хранящихся в дереве, ни одна не являлась префиксом некоторой другой строки. Это обеспечивается добавлением в конце каждой строки уникального символа (далее называемого терминальным), не встречающегося в документах; в качестве терминального обычно используется символ с кодом 0. Путем добавления еще и уникального счетчика достигается различие всех строк, что позволяет хранить в дереве одинаковые первоначально строки. Для спуска по дереву во всех узлах дерева, за исключением листьев, хранятся номера страниц внешней памяти, соответствующих дочерним узлам.

2. Организация CLB-дерева

Мы храним имена документов в отдельном файле, и каждое имя имеет указатель в этом файле; если число документов не очень велико, то мы можем хранить этот файл в оперативной памяти. Каждое слово мы храним в B-дереве вместе с указателем на имя его документа и позицией этого слова в документе.

Так как слова обычно имеют не очень большую длину, нет смысла хранить в B-дереве указатели на них, а сами слова отдельно. В отличие от строковых B-деревьев для хранения строк в узле B-дерева мы используем тернарные

деревья. Использование тернарных деревьев является удобным, но не единственно возможным выбором – для хранения строк в узле B-дерева можно использовать и другие структуры. Для поиска заданного слова мы находим его в B-дереве и получаем тот документ, в котором оно находится, и его позицию в этом документе. За счет терминального нуля и счетчика в конце слова мы храним в дереве одинаковые слова.

Поскольку в электронных документах могут встречаться ошибки, мы ограничиваем длину слова некоторым числом и пропускаем последовательности символов, длина которых превышает это число. Вследствие этого ограничения мы можем гарантировать, что в узел B-дерева помещается не менее некоторого фиксированного количества строк.

Так как длины слов различны, не следует ограничивать число строк в листе: за счет коротких слов мы можем поместить в узел B-дерева большее количество слов. Далее мы считаем, что любой узел B-дерева, за исключением корня, заполнен по крайней мере наполовину.

Каждый лист B-дерева хранит некоторый набор слов в виде тернарного дерева и для каждого слова – указатель на имя документа и позицию слова в документе. Каждый промежуточный узел хранит набор минимальных и максимальных слов и номера страниц для нижележащих узлов. B-дерево хранится в отдельном файле во внешней памяти, который разбит на страницы фиксированного размера H (обычно страницы внешней памяти равны 512 байт, размер страницы файла можно выбирать кратным этому значению, мы используем числа из промежутка от 4 до 32 Кб). Хранение дерева организуется так, что корень дерева находится в нулевой странице.

Тернарное дерево, подробно описанное в разделе 4, строится на основании набора строк. В результате получается дерево, каждый лист которого соответствует некоторой строке исходного набора. При обходе листьев дерева слева направо мы получаем исходный набор строк, упорядоченный по возрастанию, этот набор строк мы обозначим как $S(T)$ для тернарного дерева T; будем также считать, что $T[i]$ обозначает i-ю строку в $S(T)$.

Приведем алгоритм поиска слова s длины ℓ в B-дереве, построенного на основе набора строк D. Этот алгоритм является модификацией алгоритма, описанного в [10]. Алгоритм поиска строки s позволяет определить все строки из D (и связанную с ними информацию), префикс которых равен s. Нам же нужно определить все слова, помещенные в дерево, которые равны s: так как помещаемые слова дополняются терминальным символом, то необходимо проверить, является ли ℓ-й символ найденной строки терминальным. В алгоритме используется процедура $Search$, которая ищет в тернарном дереве T строку s и возвращает ее индекс i в $S(T)$, т. е. такое число i, что $T[i-1] < s \leq T[i]$ (если $i = 0$, то s меньше всех строк набора; если i равно

количеству строк, то s больше всех строк набора). Эта процедура описана подробно в разделе 4, посвященном тернарным деревьям.

Алгоритм поиска слова s длины ℓ в B-дереве:

1. $Node = 0$.
2. Повторяем шаги 2а–2е в цикле:
   а) загружаем страницу внешней памяти с номером $Node$ в блок оперативной памяти;
   б) читаем из загруженной страницы тернарное дерево T;
   в) $i = Search(T, s, \ell)$;
   г) если мы находимся в листе B-дерева, то поиск прекращается, называем строку $T[i]$ текущей и переходим на шаг 3;
   д) если i четно, то строка с номером i дерева T является минимальной строкой некоторого нижележащего узла. Присваиваем номер этого узла переменной $Node$, переходим к самому левому нижележащему листу узла $Node$, делаем его минимальную строку текущей и переходим на шаг 3;
   е) если i нечетно, то строка с номером i дерева T является максимальной строкой некоторого нижележащего узла. Заносим номер этого узла в $Node$ и переходим снова на шаг 2а.
3. Текущая строка является первой строкой из D, которая имеет самый длинный общий префикс со строкой s среди всех строк из D. Пока ℓ-й символ текущей строки дерева терминальный, добавляем в результат информацию, связанную со строкой, и переходим к следующей строке дерева (возможно, будет необходимо перейти к следующему листу B-дерева).

Высота B-дерева, построенного на основе набора строк D, равна

$$O(\log_m |D|) = O(\log_H |D|),$$

здесь и далее $|D|$ означает количество строк в D. Для поиска слова в B-дереве требуется

$$O(\log_H |D| + occ/m) = O(\log_H |D| + occ/H),$$

где occ – количество вхождений данного слова в B-дерево. Первое слагаемое относится к спуску по дереву в процессе поиска, второе – к последовательному просмотру листьев B-дерева в процессе перебора подходящих строк.

3. Создание индекса

Вставка в B-дерево осуществляется следующим образом:

1. Спускаемся по дереву для поиска листа, в котором должна находиться вставляемая строка.
2. Вставляем строку в тернарное дерево листа.
3. Если тернарное дерево не помещается в одной странице, то оно делится примерно пополам и сохраняется в двух страницах (т. е. узел B-дерева разделяется на два узла), иначе в одной. Далее возможно потребуется обновить вышележащие узлы, если произошло разделение узла B-дерева или изменились минимальная или максимальная строки в текущем узле.

При индексации документов мы столкнулись со следующей проблемой: вставка слова в B-дерево требует $O(\log_H |D|)$ обращений к внешней памяти в худшем случае. При таком количестве обращений скорость создания индексов недостаточно высока. Чтобы увеличить быстродействие, необходимо добавлять в дерево несколько слов за одно обращение к внешней памяти.

Поэтому было принято решение хранить в дереве каждое слово только один раз вместе со ссылкой на информацию о том, где оно встречается, которую мы храним отдельно. При этом мы дополняем слово только терминальным символом, а уникальный счетчик не требуется.

3.1. Хранение информации о каждом слове

Информация обо всех словах хранится в одном файле. Информация о конкретном слове хранится в списке связанных блоков фиксированного размера K, которые мы называем кластерами. При вставке слова, которое еще не встречалось, мы увеличиваем файл, выделяя пространство для нового кластера. В каждом кластере содержится номер следующего кластера цепочки или специальное обозначение конца цепочки. Если слово уже встречалось, то оно есть в дереве и информация о нем добавляется в последний кластер цепочки; если в нем не хватает места, то цепочка увеличивается и запись производится в новый кластер. При этом соответственно изменяется указатель на следующий кластер в том кластере, который был ранее последним в цепочке. Для извлечения информации о слове мы последовательно просматриваем цепочку, начиная с ее начала. Таким образом, для поиска и добавления слова мы должны в B-дереве хранить вместе со словом номер первого кластера, номер последнего кластера и объем занятого пространства в последнем кластере. При подобной схеме для вставки N слов потребуется $O(N)$ обращений

к внешней памяти, но это число можно значительно уменьшить, как будет показано далее. Полученную структуру мы называем CLB-деревом (рис. 1). Массив длины D составлен из всех слов данного текста. Он упорядочен по возрастанию, в нем нет одинаковых слов, каждое слово представлено цепочкой кластеров, в которой хранится информация обо всех вхождениях данного слова в текст.

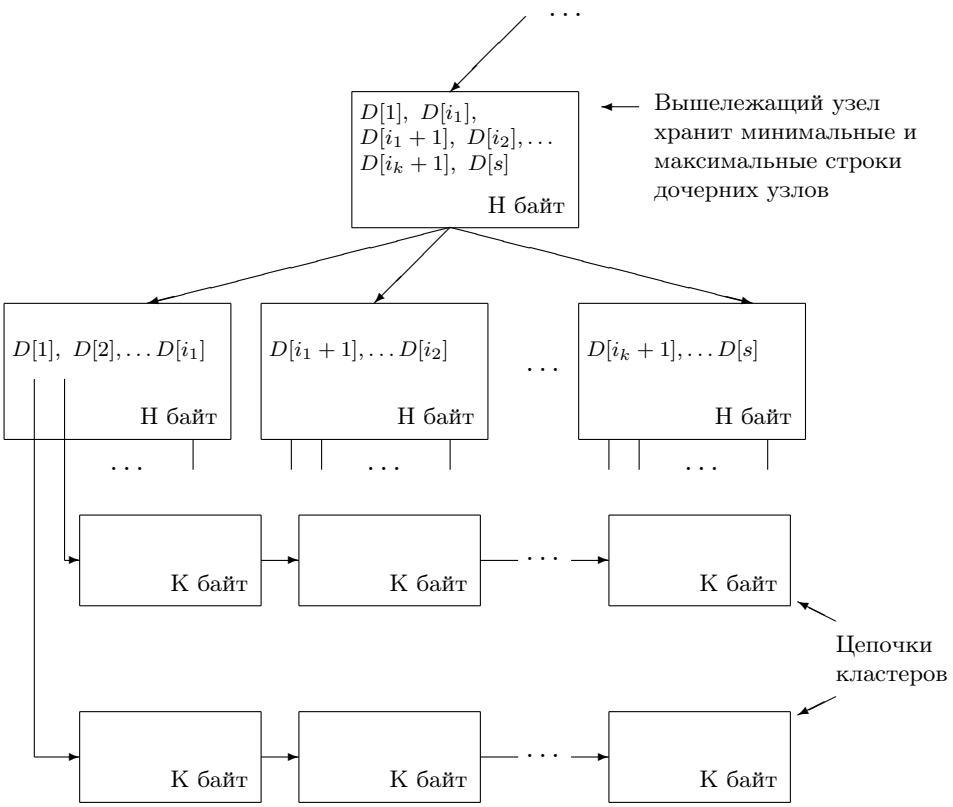

Рис. 1. Структура CLB-дерева

Количество обращений к внешней памяти уменьшится, если мы будем хранить в оперативной памяти последние кластеры цепочек наиболее часто встречающихся слов. Однако слов достаточно много, и в оперативной памяти, как правило, мы можем хранить кластеры лишь малой их части. За счет чего можно достичь нужного результата? Только в том случае, если мы будем хранить в оперативной памяти последние кластеры большинства слов, мы сможем добиться того, чтобы за одно обращение к внешней памяти записывать информацию о более чем одном слове документа. Мы достигаем этого путем перехода к хранению в дереве базовых (словарных) форм

для тех слов, которые есть в словаре морфологического анализатора. Слова, хранящиеся в словаре, составляют более 90 % всех слов в обычных документах, а соответствующих базовых форм для этих слов гораздо меньше, чем самих слов. Разработанный одним из авторов морфологический анализатор содержит около 3,5 млн словоформ русского языка, образованных от 205 тыс. базовых форм. При этом таблицы анализатора хранятся в виде минимального детерминированного конечного автомата (МДКА), состоящего из 172 тыс. узлов и 414 тыс. дуг, который занимает на диске менее 15 Мб. Для построения автомата использована оригинальная техника [12], основанная на базе алгоритма последовательного построения МДКА [13].

При сегодняшнем развитии вычислительной техники можно хранить в памяти последние кластеры для известных базовых форм. Слова, которых нет в словаре, будем хранить в том виде, в котором они встречаются в тексте.

Пусть q – это минимальное помещающееся в кластер число записей, каждая из которых хранит информацию об одном слове, а d – доля всех слов в индексируемых документах, словоформы которых есть в словаре. Тогда общее число обращений к внешней памяти при вставке N слов будет равно

$$O(dN/q + (1-d)N) = O(dN/K + (1-d)N).$$

Таким образом решается задача добавления информации о слове в цепочку кластеров. Но как добавлять слова в B-дерево, совершая минимум обращений к внешней памяти? Мы приходим к тому, чтобы добавлять в дерево одновременно несколько слов, при этом некоторые слова попадают в одну и ту же страницу внешней памяти.

3.2. Добавление слов в B-дерево

В [10] описан алгоритм добавления в дерево одновременно некоторого количества строк, но он ограничивает количество добавляемых строк числом m, которое не очень велико. Далее мы предлагаем алгоритм, позволяющий добавлять одновременно любое количество строк.

Процедура $AppendPacket$ добавляет в B-дерево набор строк $Strings$ в количестве $Count$. В ней используется процедура $Insert$, которая вставляет набор строк $Strings$ в указанный узел $Node$. Эта процедура возвращает список, состоящий из новых записей, появившихся в результате вставки (возможно, в результате разделения узла $Node$ на части). Каждая из этих записей описывает один узел B-дерева и хранит в себе следующие поля: минимальную строку L, максимальную строку R и номер соответствующего узла $Node$.

Каждая строка описывается записью, имеющей следующие поля: $String$ – сама строка, $Length$ – ее длина, $Data$ – информация о строке, которую мы добавляем, $Next$ – указатель на запись следующей строки. Иными словами,

строки представлены в виде односвязного списка, который является удобной структурой для построения наборов строк в шаге 4а процедуры $Insert$ (можно использовать те же записи, изменяя только указатели $Next$).

Процедура $AppendPacket(Strings, Count)$:

1. $Insert(0, Strings, Count, List)$.
2. Пока количество элементов в $List$ больше одного, выполняем шаги 2а–2в:
   а) переносим страницу 0 в конец файла и изменяем номер узла первой записи в $List$;
   б) формируем тернарное дерево T, вставляя в него все строки из $List$;
   в) $InsertTree(0, T, List)$.

Процедура $Insert(Node, Strings, Count, List)$ является модификацией упомянутого выше алгоритма, описанного в [10]:

1. Загружаем в память страницу узла $Node$.
2. Читаем из загруженной страницы тернарное дерево T.
3. Если узел $Node$ соответствует листу B-дерева, то выполняем шаги 3а–3б:
   а) вставляем все строки набора $Strings$ в загруженное тернарное дерево T, при этом для каждой строки:
   – если такой строки не было в дереве, то создаем новый кластер, добавляя его в конец файла, и записываем в него информацию о слове;
   – в противном случае: если в последнем кластере цепочки есть место для информации о слове, добавляем информацию в этот кластер; если нет, то увеличиваем цепочку, добавляя еще один кластер (и изменяем указатель на следующий кластер для того кластера, который был последним в цепочке ранее), записываем в добавленный кластер информацию о слове;
   б) $InsertTree(Node, T, List)$.
4. Если это промежуточный узел дерева, то выполняем шаги 4а–4в:

а) определяем для каждой строки набора $Strings$ тот нижележащий узел, в который она должна попадать, и формируем на основании этого для каждого нижележащего узла вставляемые наборы строк;

б) для каждого набора строк:

– вызываем процедуру $Insert$, которая возвращает список $List$;

– если минимальная строка нижележащего узла изменилась, вставляем в T новую минимальную строку (поле L первой записи в списке $List$) и удаляем старую;

– если максимальная строка нижележащего узла изменилась (поле R последней записи в списке $List$), вставляем в T новую максимальную строку и удаляем старую;

– если $List$ имеет длину больше чем один, то добавляем все промежуточные строки (поле R первой записи, поле L последней записи и все поля L и R остальных записей, если длина списка больше двух);

в) $InsertTree(Node, T, List)$.

В следующей процедуре $InsertTree(Node, T, List)$ используется процедура $ExtractFirst$, которая в случае, если тернарное дерево не помещается в одной странице внешней памяти, разделяет его на два дерева и возвращает дерево, которое помещается в страницу. При этом оставшееся дерево будет занимать не менее половины страницы.

Процедура $ExtractFirst(T, \ell, r)$ также возвращает самый левый лист отделенного дерева ℓ, его самый правый лист r и оставляет в T оставшееся дерево. Эта процедура подробно описана в разделе 4.

Процедура $InsertTree(Node, T, List)$:

Пока указатель T не является пустым деревом:

1. Вызываем процедуру $X = ExtractFirst(T, \ell, r)$.
2. Записываем дерево X в буфер.
3. Записываем буфер в страницу узла $Node$.
4. Добавляем в результат запись с узлом $Node$, полем L, равным ℓ, и полем R, равным r.
5. Делаем $Node$ равным общему количеству страниц, т. е. следующая страница будет записана в конец файла.

Число вставляемых слов ограничено только доступной оперативной памятью, так как необходимо хранить в ней вставляемые слова и информацию о них. При тестировании мы вставляли за один раз миллион слов. Для минимизации числа обращений к внешней памяти необходимо, чтобы словоформы известных слов находились рядом в дереве. Для этого в начало слов, которых нет в словаре, мы добавляем символ с кодом 0, чтобы отделить их от известных словоформ. Таким образом, известные словоформы занимают в B-дереве не более $2C/m$ страниц, где C – число известных базовых слов. В настоящий момент словарь содержит около 205 тыс. слов.

Вставка N слов в B-дерево (при $N \geq C$) потребует

$$O(2C/m + (1-d)N \log_H(C + (1-d)|D|))$$

обращений к внешней памяти, высота дерева равна

$$O(\log_H(C + (1-d)|D|)).$$

Первое слагаемое соответствует вставке в дерево известных слов, а второе – неизвестных слов. Вставка N слов в CLB-дерево потребует

$$\begin{aligned} O(2C/m + (1-d)N \log_H(C + (1 - |D|) + dN/K + (1-d)N) = \\ = O(2C/m + dN/K + (1-d)N(1 + \log_H(C + (1-d)|D|))). \end{aligned}$$

Первые два слагаемых в сумме соответствуют вставке известных слов, а последнее – вставке неизвестных. Значительный выигрыш мы получаем, когда N много больше C. Если считать C константой, то вставка потребует

$$O(dN/K + (1-d)N(1 + \log_H((1-d)|D|)))$$

обращений к внешней памяти. Поиск слова в CLB-дереве потребует

$$O(\log_H(C + (1-d)|D|) + occ/K)$$

обращений к внешней памяти. Если считать C константой, то будет

$$O(\log_H((1-d)|D|) + occ/K)$$

обращений к внешней памяти. При этом в отличие от простого B-дерева, где стоимость поиска была $O(\log_H |D| + occ/H)$, здесь кластеры хранят в себе только информацию о словах. В то же время листы B-дерева хранят в себе дополнительные структуры, например тернарные деревья, т. е. кластеры более плотно заполнены информацией о словах.

3.3. Дополнительные улучшения

Когда мы вставляем N слов в CLB-дерево, количество кластеров увеличивается (не менее чем на dN/q) за счет добавления известных слов, плюс те кластеры, которые возникают при вставке слов, которых еще нет в дереве. При добавлении большого количества слов происходит увеличение размера файла и добавление некоторого числа кластеров в конец файла. Если во время вставки слов сначала определить, в какие кластеры они попадут и какие кластеры будут добавлены, то можно произвести некоторую оптимизацию. Добавив слова в уже существующие кластеры, можно выделить блок памяти для всех новых кластеров, которые добавляются в конец файла, и сначала записать всю информацию, которая должна быть в новых кластерах, в этот блок, а затем записать этот блок во внешнюю память за одно обращение. Быстродействие при этом повысится, так как современные операционные системы оптимизируют запись нескольких кластеров, проводимую одной файловой операцией.

При поиске слова и извлечении информации о нем требуется прочитать всю цепочку кластеров. Если мы так организуем цепочку кластеров, что некоторые группы кластеров будут идти подряд, то мы сможем считывать во время поиска сразу группу кластеров. Введем некоторое число G – количество кластеров, которые будут записываться последовательно, назовем эти кластеры блоком кластеров. При добавлении кластера в цепочку мы будем резервировать следующие за ним кластеры, чтобы впоследствии записывать информацию о слове в них. При добавлении кластера в цепочку обозначим через N число кластеров в последнем блоке, это число хранится вместе со словом в B-дереве. Если $N = G$, то начинаем новый блок, присваиваем N значение 1. Иначе, если N является степенью 2, то находим то место файла, где есть $2N$ свободных кластеров, и переносим N последних кластеров в начало этого места, в следующий за ними кластер мы добавляем информацию о слове, а остальные $N - 1$ кластеров объявляем зарезервированными. Мы также организуем $\log_2 G$ списков, в каждом из которых храним начальные кластеры освобождающихся фрагментов, чтобы в дальнейшем их использовать (первый список для фрагментов из одного кластера, второй – из двух, третий – из четырех и т. д.). Перенос кластеров можно реализовывать, только если в цепочке менее G кластеров. Если в цепочке есть уже G кластеров, то при начале блока мы можем сразу резервировать G новых кластеров. В результате мы получаем увеличение использующегося дискового пространства не более чем в два раза. В проведенном нами эксперименте, описанном в конце статьи, K было равно 512 байт (стандартный размер сектора диска), G бралось равным 2048 байт, т. е. один блок кластеров равнялся 1 Мб. За счет единовременной записи кластеров значительно (в несколько раз для боль-

ших цепочек) возрастает скорость чтения цепочки, тогда как из-за переноса скорость создания индекса замедляется в среднем на четверть.

4. Тернарные деревья

4.1. Общее описание

Тернарные деревья – одна из структур для поиска строки в наборе строк. Каждый промежуточный узел P тернарного дерева имеет трех потомков, обозначим их так: $Left(P)$ – левый узел, $Middle(P)$ – средний узел, $Right(P)$ – правый узел, – и хранит в себе символ $SplitChar(P)$. Родительский узел для узла P обозначим $Parent(P)$. Каждый лист дерева хранит информацию о соответствующей строке. Каждый узел дерева либо промежуточный, либо лист, поэтому при реализации можно использовать для узла дерева одну структуру, в которой указатели на дочерние узлы и информация о строке находятся на одном месте в памяти. Поиск строки в дереве начинается в корне и далее строка просматривается символ за символом. Если текущий символ строки меньше $SplitChar$ у текущего узла дерева, то мы переходим к узлу $Left$, если больше – то к узлу $Right$ (если требуемых узлов нет, значит, строки нет в дереве), если равен $SplitChar$ – то к узлу $Middle$, и только в последнем случае переходим к следующему символу строки.

Дерево на основе одной строки s длины ℓ создается в виде цепочки узлов длины $\ell + 1$, i-й промежуточный узел хранит в себе i-й символ строки, последний узел – лист, который хранит информацию о строке, каждый узел является узлом $Middle$ для предыдущего узла цепочки.

Процедура $InsertTernaryTree(T, s)$ осуществляет вставку строки s в непустое тернарное дерево T:

1. $C = s[0], P$ – корень дерева, $i = 0$.
2. Повторяем следующие шаги в цикле:
   а) если $C < SplitChar(P)$, то:
   если у P есть левый узел, то $P = Left(P)$ и переходим на шаг 2а, иначе создаем цепочку узлов, как при создании дерева на основе одной строки, но начинаем с i-го символа строки s. Первый узел цепочки назначаем левым узлом для P. Выход;
   б) если $C > SplitChar(P)$, то:
   если у P есть правый узел, то $P = Right(P)$ и переходим на шаг 2а, иначе создаем цепочку узлов, как при создании дерева на основе

одной строки, но начинаем с i-го символа строки s. Первый узел цепочки назначаем правым узлом для P. Выход;

в) $i = i + 1$;

г) $C = S[i]$;

д) $P = Middle(P)$;

е) если P – лист, то поскольку все строки имеют в конце терминальный символ, тот случай, что строка, соответствующая P, является собственным префиксом s, исключается (строка s уже есть в дереве).

Заметим, что при таком алгоритме вставки у любого промежуточного узла дерева есть узел $Middle$.

4.2. Поиск в дереве

Можно считать, что дерево T соответствует некоторому набору строк $S(T)$, причем этот набор упорядочен по возрастанию и n-й лист дерева, при обходе листов слева, соответствует n-й строке в упорядоченном наборе строк. Через $T[i]$ обозначаем i-ю строку в $S(T)$. Нам необходимо определить для данной строки s длины ℓ число i, удовлетворяющее условию: $T[i-1] < s \leq T[i]$ (если $i = 0$, то s меньше всех строк набора, если i равно количеству строк, то s больше всех строк набора).

Процедура $Search(T, s)$:

1. $C = s[0], P$ – корень дерева, $i = 0$.

2. Повторяем следующие шаги в цикле:

а) если $C < SplitChar(P)$, то:
если у P есть левый узел, то $P = Left(P)$, иначе требуемое число равно номеру самого левого нижележащего листа у P. Выход;

б) если $C > SplitChar(P)$, то:
если у P есть правый узел, то $P = Right(P)$, иначе требуемое число равно увеличенному на единицу номеру самого правого нижележащего листа P. Выход;

в) $P = Middle(P)$;

г) если P – лист, то требуемое число равно номеру листа P. Выход;

д) $i = i + 1$;

е) если $i \geq \ell$, то требуемое число равно номеру самого левого нижележащего листа P. Выход;

ж) $C = s[i]$.

4.3. Удаление

Во время вставки строк в B-дерево возможно потребуется удаление строки из дерева. Процедура $RemoveTernaryTree(T, s)$:

1. Пусть P – лист дерева T, соответствующий строке s, $U = Parent(P)$.
2. Если P равен левому или правому узлу U, то удаляем его из узла U вместе со всеми нижележащими узлами. Выход.
3. Если U имеет и левый и правый узел, то:
   а) $X = Left(U)$;
   б) если у U есть родитель, то заменяем в родителе U узел U на $Right(U)$, иначе корнем дерева делаем $Right(U)$;
   в) U полагаем равным $Right(U)$;
   г) пока у U есть левый узел, делаем U равным$Left(U)$;
   д) $Left(U)$ полагаем равным X;
   е) выход.
4. Если у U есть левый узел, то:
   а) если у U есть родитель, то заменяем в родителе U узел U на $Left(U)$, иначе корнем дерева делаем $Left(U)$;
   б) выход.
5. Если у U есть правый узел, то:
   а) если у U есть родитель, то заменяем в родителе U узел U на $Right(U)$, иначе корень дерева делаем равным $Right(U)$;
   б) выход.
6. $P = U$, переходим на шаг 3.

В процессе выполнения этой процедуры мы удаляем из памяти неиспользуемые более узлы.

4.4. Разделение

Следующий алгоритм отделяет от дерева T часть, содержащую все строки – с первой до строки заданного листа $Leaf$.

Процедура $ExtractFirstByLeaf(T, Leaf)$:

1. Поднимаемся наверх от $Leaf$ до тех пор, пока не встретим узел, у которого родительский узел имеет кроме текущего узла еще и узел справа, обозначаем этот узел справа через P; если мы не встретим такого узла, то возвращаем дерево целиком, т. е. отделяем от T все дерево.

2. Далее поднимаемся наверх, дублируя все проходимые узлы, помещая один из узлов в отделяемое дерево, а другой – в оставшееся дерево, при этом обнуляем у узла, помещаемого в отделяемое дерево, все указатели, ведущие в оставшееся дерево, а для узла, помещаемого в оставшееся дерево, выполняем обратное действие.

3. После шага 2 в отделяемом дереве на пути от корня к самому правому листу могли появиться узлы, у которых нет среднего, но есть либо левый, либо правый узел. Такие узлы не несут никакой информации, и мы их удаляем, при этом нижележащее поддерево переносим к родительскому узлу (если родительского узла нет, то все отделяемое дерево состоит из указанного ранее поддерева).

4. Если в оставшемся дереве на пути от корня к самому левому листу также появились узлы, у которых нет среднего узла, то совершаем действия, аналогичные шагу 3.

Следующую процедуру $ExtractFirst$ легко реализовать, если мы на протяжении изменения дерева T храним в отдельной переменной размер $size(T)$, который потребуется для сохранения T во внешней памяти. Во всех вышеописанных алгоритмах можно при изменении дерева корректировать его размер.

Процедура $ExtractFirst(T, \ell, r)$ возвращает отделенное дерево, его самый левый лист ℓ, самый правый лист r; T после ее выполнения содержит оставшееся после отделения дерево.

1. Если $size(T) < H$, то результат процедуры – все дерево T и выход.

2. Если $size(T) < 3H/2$, то $S = size(T)/2$, иначе $S = H$.

3. Начиная с самого левого нижележащего листа дерева двигаемся направо по листам дерева, до тех пор пока поддерево дерева T, содержащее пройденные листья, имеет размер во внешней памяти, меньший или равный S.

4. Пусть $Leaf$ – последний пройденный узел. Теперь используем процедуру $ExtractFirstByLeaf(T, Leaf)$ для отделения от дерева его части. Полученное дерево, возможно, будет иметь размер, меньший S, за счет шагов 3 и 4 в процедуре $ExtractFirstByLeaf$.

4.5. Хранение тернарных деревьев

Сохранение дерева в блоке памяти размером H производится следующей рекурсивной процедурой. Она вызывается для корня дерева и изменяет указатель *Pointer*, который указывает на начало блока, а после вызова процедуры – на незанятую часть блока.

Процедура $Save(P, Pointer)$:

1. Если P – лист, сохраняем в памяти по адресу $Pointer$ информацию о строке, хранящейся в листе. Выход.
2. Сохраняем $SplitChar(P)$ в памяти по адресу $Pointer$ и изменяем $Pointer$.
3. Если у P есть левый узел: $Save(Left(P), Pointer)$.
4. $Save(Middle(P), Pointer)$.
5. Если у P есть правый узел: $Save(Right(P), Pointer)$.

4.6. Оптимизация хранения дерева

При сохранении дочерних узлов можно также сохранять в блоке смещения, которые потребуются для перехода от текущего узла к соответствующему дочернему узлу. Как правило, для них достаточно двух байт. Потребуется сохранить смещения для всех дочерних узлов, за исключением одного, располагающегося сразу после текущего узла. Впоследствии при поиске строк в B-дереве можно будет только читать из памяти блок и не создавать в памяти дополнительных структур, а просто перемещаться по этому блоку в процессе поиска строк.

5. Вычислительный эксперимент

Для исследования временных характеристик CLB-деревьев нами было измерено время построения CLB-дерева в зависимости от объема исходных данных в диапазоне от нуля до 2 Гб. Производилась обработка произведений художественной литературы на русском языке. При этом количество известных словоформ в данных текстах составило 94 %. При создании индекса мы брали $H = 4096$ байт, $K = 512$ байт. Эксперимент проводился на компьютере с процессором Intel Celeron с тактовой частотой 1,1 Гц и оперативной

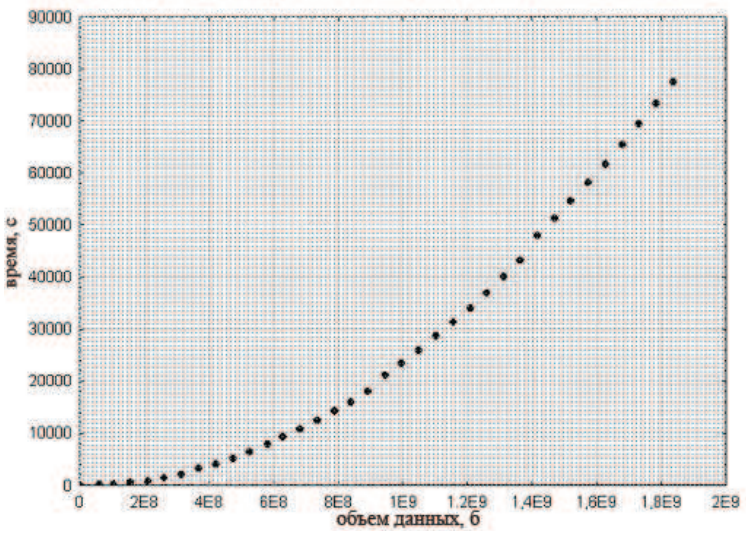

Рис. 2. Зависимость времени построения CLB-дерева от размера входных данных

памятью 384 Мб, работающем под управлением Microsoft Windows XP. Результирующий график приведен на рис. 2. зависимость времени от размера становится линейной, что подтверждается регрессией, приведенной на рис. 3.

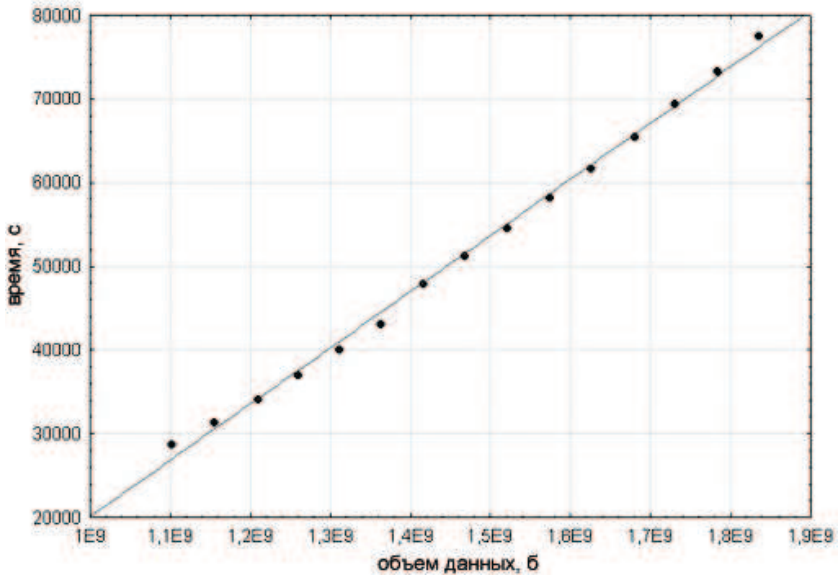

Рис. 3. Линейная регрессия данных измерений в диапазоне от 1 до 2 Гб

Литература

1. Prywes N. S., Gray H. J. The organization of a Multilist-type associative memory // IEEE Trans. on Communication and Electronics. 1963. № 68. P. 488–492.
2. Bayer R., McCreight E. Organization and maintenance of large ordered indexes // Acta Informatica. 1972. Vol. 1, № 3. P. 173–189.
3. Bayer R., Unterauer K. Prefix B-trees // ACM Trans. Database Syst. 1977. Vol. 2, № 1. P. 11–26.
4. Morrison D. R. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric // Journal of the ACM. 1968. Vol. 15, № 4. P. 514–534.
5. McCreight E. M. A space-economical suffix tree construction algorithm // Journal of the ACM. 1976. Vol. 23, № 2. P. 262–272.
6. Weiner P. Linear pattern matching algorithm // Proc. 14th Annual IEEE Symposium on Switching and Automata Theory. Washington, 1973. P. 1–11.
7. Gonnet G., Baeza-Yates R., Snider T. New indices for text: pat trees // Information Retrieval: Data Structure and Algorithms. New Jersey, 1992. P. 66–82.
8. Manber U., Myers G. Suffix arrays: a new method for on-line string searches // SIAM Journal on Computing. 1993. Vol. 22, № 5. P. 935–948.
9. Bentley J. N., Sedgewick R. Fast algorithms for sorting and searching strings // Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms. San Francisco, 1998. P. 360–369.
10. Ferragina P., Grossi R. The string B-tree: a new data structure for string search in external memory and its applications // Journal of the ACM. 1999. Vol. 46, №. 2. P. 236–280.
11. Ferragina P., Grossi R. An experimental study of SB-trees // Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms. Philadelphia, 1996. P. 373–397.
12. Лукач Ю. С. Быстрый морфологический анализ флективных языков // К 100-летию со дня рождения П. Г. Конторовича и 70-летию Л. Н. Шеврина: Междунар. алгебраич. конф.: Тез. докл. Екатеринбург, 2005. С. 182–183.
13. Daciuk J., Mihov S., Watson B., Watson R. Incremental construction of minimal acyclic finite state automata // Computational Linguistics. 2000. Vol. 26, № 1. P. 3–16.