Государственное образовательное учреждение
высшего профессионального образования
«Уральский государственный университет им. А.М. Горького»

На правах рукописи

Веретенников Александр Борисович

ПРОГРАММНЫЙ КОМПЛЕКС И ЭФФЕКТИВНЫЕ МЕТОДЫ ОРГАНИЗАЦИИ И ИНДЕКСАЦИИ БОЛЬШИХ МАССИВОВ ТЕКСТОВ

05.13.18 – математическое моделирование, численные методы и комплексы программ

Диссертация на соискание ученой степени кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук
Пименов Владимир Германович

Екатеринбург — 2009

Оглавление

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 информация в Интернете исчисляется уже терабайтами.

Одновременно активно развиваются корпоративные системы, в которых хранятся огромные объемы информации. Постоянно создаются новые системы, идет развитие таких отраслей, как Enterprise Content Management. На сегодняшний день существует общая тенденция к переводу процессов документооборота из бумажного в электронный вид. Как правило, задачи поиска в подобных системах всегда весьма актуальны.

В этой ситуации существенно возрос интерес к классификации больших массивов документов и быстрому поиску в них нужной информации. См., например, [19, 32].

Очевидно, что обеспечение быстрого поиска в подобных массивах должно являться комбинацией методов внешнего поиска (в дисковых файлах) и внутреннего поиска (в оперативной памяти). 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 индексирования данных во внешней памяти обычно используются либо инвертированные файлы, Н. Прайвс (N. S. Prywes) и Г. Грей (H. J. Gray) [30], либо B-деревья и их вариации, предложенные Р. Байером (R. Bayer), Э. Мак-Крейтом (E. M. McCreight) [16, 17] и М. Кауфманом (M. Kaufman) [не опубликовано]. Инвертированные файлы применяются в таких поисковых системах, как Яndex и Google. B-деревья часто применяются при выполнении поисковых запросов в базах данных.

С другой стороны, для быстрого поиска во внутренней памяти чаще всего применяются цифровые деревья Patricia [29], суффиксные деревья [28, 31], суффиксные массивы [23, 27] и тернарные деревья поиска [18]. Представляется логичным строить новые комбинированные структуры на основе уже апробированных структур. Первой из известных нам попыток создания такой комбинированной структуры были String B-деревья – комбинация B-дерева и Patricia, предложенные П. Феррагина (P. Ferragina) и Р. Гросси (R., Grossi) в [21, 22].

В диссертации рассматриваются задачи поиска в текстах, написанных на естественных языках. В [3, 5, 6, 9–12] автором была описана новая структура данных, CLB-дерево, предназначенная для поиска в большом объеме электронных документов, написанных на естественном языке. Новая структура обладает рядом преимуществ по сравнению с другими структурами данных.

Большинство современных поисковых систем выполняют поиск по ключевым словам. Можно выделить несколько основных задач, решаемых этими системами.

Три задачи поиска, в порядке возрастания сложности

1. Поиск всех документов, содержащих каждое из заданных слов.
2. Точный поиск фразы, нахождение всех документов, включающих в себя подряд все указанные слова (иначе говоря, между искомыми словами в тексте нет других слов):
   a. С учетом порядка искомых слов
   b. Без учета порядка искомых слов
3. Поиск с учетом расстояния, задача похожа на задачу п. 2, только в тексте допускается наличие других слов между искомыми словами. Результатом поиска является набор фраз текста, в которых встречается как искомые слова, так и возможно другие слова. При этом в найденной фразе не должно содержаться фразы меньшего размера, удовлетворяющей условиям поиска.

Разработанный комплекс программ и структура данных CLB-дерево позволяют решать эффективно все эти задачи. Достаточно давно используемые для поиска в текстах инвертированные файлы имеют один существенный недостаток – их сложно обновлять. Остальные упомянутые структуры менее применимы для решения рассматриваемых задач. Вместе с тем, следует сказать, что массивы документов часто изменяются, и задача эффективного обновления индекса весьма актуальна.

Цель диссертации состоит в создании комплекса программ, предназначенного для эффективного создания индекса и поиска в большом массиве текстовых данных. Основной задачей является разработка такого комплекса, который будет обладать возможностью быстрого добавления новых данных в индекс. Для решения данной задачи требуется разработать новые структуры данных и алгоритмы.

Методика исследования. В соответствии с концепциями математического моделирования исследование разбито на три этапа: модель –

алгоритм – программный комплекс.

На первом этапе построена модель исследуемого объекта – большого набора текстов, с учетом анализа морфологии языка. Построенная модель отражает важнейшие свойства исследуемого объекта, связи между его составляющими частями и позволяет исследовать объект и связанные с ним научные и технические проблемы теоретическими методами. Построенная модель является фундаментом для дальнейшего исследования с использованием современной технологии математического моделирования.

На втором этапе для решения поставленной задачи разработана новая структура данных, CLB-дерево, и широкий набор алгоритмов для работы с ней. Выявленные в результате анализа модели свойства текстов позволили еще в начале разработки определить наиболее важные требования к структуре данных и алгоритмам, а также понять за счет применения каких подходов можно достичь максимальной эффективности при решении поставленных проблем.

На третьем этапе разработан комплекс проблемно-ориентированных программ, предназначенный для решения задачи. После чего были проведены разнообразные эксперименты с целью подтверждения эффективности созданного комплекса программ, а также разработанных структур данных и алгоритмов.

Исследование основано на современных методах программирования, таких, в частности, как структурное, объектно-ориентированное программирование, активное применение шаблонов и моделей. Систематически применяются понятия и методы теории вычислительной сложности, методов построения и анализа алгоритмов.

Научная новизна. Разработана новая структура данных CLB-

дерево. CLB-дерево – это структура данных, по скорости поиска не уступающая инвертированным файлам и их аналогам, но в отличие от них в CLB-дерево можно легко и быстро добавлять новые данные. При поиске в текстах с помощью CLB-дерева учитывается морфология языка этих текстов. Эффективность CLB-дерева доказана в ряде теорем. Разработан комплекс программ, на практике показывающий преимущества использования CLB-дерева.

Теоретическая и практическая ценность. Разработанные в диссертации структуры данных позволяют эффективно строить быстро обновляемый индекс, предназначенный для поиска в большом массиве текстовых данных. Разработанные методы, структуры данных и алгоритмы реализованы в комплексе программ.

Основные результаты.

1) Разработана структура данных CLB-дерево.
2) Разработаны эффективные алгоритмы создания CLB-дерева на основании большого массива текстовых документов. Основное преимущество CLB-дерева заключается в том, что в CLB-дерево можно легко добавлять новые данные, при этом скорость поиска такая же, как у инвертированных файлов.
3) Получены теоретические оценки затрат ресурсов на добавление данных в CLB-дерево, поиск данных в CLB-дереве и хранение CLB-дерева. Данные оценки позволяют заранее предсказать, сколько времени займут создание индекса, поиск и сколько места во внешней памяти может потребоваться для хранения индекса.
4) Разработан программный комплекс, позволяющий строить индекс на основании большого массива текстовых документов. Разработанный комплекс имеет модульную архитектуру и широкие возможности кон-

фигурации. Реализован интерфейс для использования программного комплекса в других продуктах.

5) Проведены эксперименты подтверждающие эффективность разработанных структур данных и алгоритмов, как в 32-битной среде, так и в 64-битной среде.
6) Проведены сравнительные эксперименты с инвертированными файлами по созданию индекса и поиску. Эксперименты показывают преимущество CLB-дерева при создании индекса, а также то, что скорость поиска в CLB-дереве такая же, как и при использовании инвертированных файлов.
7) Проведены сравнительные эксперименты с рядом широко используемых программных комплексов, предназначенных для решения рассматриваемых задач. Проведенные эксперименты показывают преимущество по скорости создания индекса, основанного на CLB-дереве, по сравнению с аналогами.

Структура и объем работы. Диссертация состоит из введения, 4-х глав и списка литературы. Главы разбиты на параграфы, нумерация глав и параграфов в работе сквозная. Нумерация формул и утверждений в работе двойная: первый индекс – номер параграфа, второй индекс – порядковый номер формулы или утверждения внутри параграфа. Общий объем работы составляет 150 страниц, библиография содержит 33 наименования.

Апробация работы. Результаты диссертации докладывались и обсуждались на:

— Международной алгебраической конференции: К 100-летию со дня рождения П. Г. Конторовича и 70-летию Л. Н. Шеврина (Екатеринбург, 2005);

— 37-й Региональной молодежной конференции «Проблемы теоретической и прикладной математики» (Екатеринбург, 2006);

— 39-й Региональной молодежной конференция «Проблемы теоретической и прикладной математики» (Екатеринбург, 2008);

— Электронные библиотеки: перспективные методы и технологии, электронные коллекции. Десятая Всероссийская научная конференция «RCDL'2008» (Дубна: ОИЯИ, 2008);

— Третьей международной научной конференция «Информационно-математические технологии в экономике, технике и образовании» (Екатеринбург, 2008);

— 40-й Всероссийской молодежной конференции «Проблемы теоретической и прикладной математики» (Екатеринбург, 2009);

— Межвузовской научной конференции по проблемам информатики «СПИСОК 2009». (Екатеринбург 2009);

— Научном семинаре «Системный семинар» в Уральском государственном университете (Екатеринбург, 2005, 2008).

Публикации. Основные результаты диссертации опубликованы в работах [3–12]. Результаты, вошедшие в диссертацию получены автором самостоятельно. Работы [10, 12] опубликованы в ведущих рецензируемых научных журналах.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Во введении обоснована актуальность темы исследований, сформулирована цель диссертационной работы и пути ее достижения, отмечена новизна и практическое значение работы.

В первой главе построены модели исследуемого объекта, дается обзор существующих структур данных, алгоритмов, применяющихся при

работе с внешней памятью, и систем.

Сначала мы строим базовую модель текстов. Затем путем ее усложнения строится модель, учитывающая морфологию языка. Далее построена модель базы данных поисковой системы, которая определяет, каким образом должны хранится данные для решения поставленных задач поиска. В завершение дана модель программного комплекса поисковой системы. После описания моделей приведены ряд часто используемых структур данных, которые могут быть использованы для построения комплекса программ. Описана структура несколько поисковых систем, применяющихся для поиска в большом наборе текстовых данных.

Во второй главе дается описание разработанных автором данной работы структур данных, алгоритмов, и ряд теорем, которые доказывают эффективность разработанных алгоритмов.

Дается описание CLB-дерева, приведен основной алгоритм эффективного создания индекса, приведен алгоритм поиска.

Каждой базовой форме слова сопоставляется цепочка связанных блоков, в которой сохраняются данные о вождениях соответствующего слова в документах. Основные моменты: предотвращение фрагментации данных для последующего быстрого поиска, эффективное использование дискового пространства, сокращение количества обращений к внешней памяти для повышения производительности, методы организации кэша.

Предлагается также новый подход для обработки наиболее часто встречающихся слов, который позволяет более эффективно выполнять точный поиск фраз, содержащих такие слова. Рассмотрены схемы кодирования. Доказываются следующие основные теоретические результаты.

Введем обозначения:

Пусть d – доля известных слов в текстах, R – текущее количество

записей о вхождении в CLB-дереве.

K – размер кластера, H – размер страницы используемого B-дерева.

Теорема 1. Для вставки N записей о вхождении в CLB-дерево достаточно $O(d \cdot N/K + (1-d) \cdot N \cdot (1 + log_H((1-d) \cdot (R+N))))$ обращений к внешней памяти.

Теорема 2. Для поиска набора слов из N-элементов в CLB-дереве достаточно $O(N \cdot (log_H((1-d) \cdot R) + occ/K))$ обращений к внешней памяти (здесь occ – количество вхождений данных слов в текстах).

Теорема 3. Размер файла с кластерами для CLB-дерева составляет $O(S)$, где S – суммарный объем сохраненных в индексе данных.

На практике можно использовать не строгую оценку объема файла с кластерами: $d \cdot S \cdot 2 + (1-d) \cdot S \cdot 4$. Следует отметить, что данная оценка, хотя и не является характеристикой худшего случая, но весьма близка к нему, а в среднем в проведенных экспериментах получается меньший объем файла.

В третьей главе дается описание разработанного программного комплекса и приводятся результаты экспериментов. подтверждающие теоретические результаты и возможность применения данной структуры и алгоритмов на практике.

Рассматривается архитектура комплекса. Описываются возможности конфигурации.

Отдельно проведены сравнительные исследования производительности в 32-битной и 64-битной среде. Дается сравнение скорости работы алгоритма создания индекса с другими структурами данных. Приводится сравнение скорости работы с рядом широко используемых программных комплексов других разработчиков, предназначенных для решения рассматриваемой задачи.

В четвертой главе рассмотрены вспомогательные компоненты и особенности реализации, в частности подсистема управления выделением памяти и подсистема интерфейса пользователя, на базе WindowSystemObject (WSO) – специальной библиотеки для создания интерфейсов пользователя, также разработанной автором работы.

Использование морфологии. Текущая реализация CLB-дерева использует морфологический анализатор.

В качестве морфологического анализатора может использоваться анализатор, созданный Ю. С. Лукачом [14]. Морфологический анализатор содержит около 3,5 млн. словоформ русского языка, образованных от 205 тысяч базовых форм. Для каждого слова, известного анализатору он возвращает одну или несколько базовых форм. Слова, хранящиеся в словаре морфологического анализатора, составляют более 90% всех слов в типичных документах.

Также может использоваться широко известный морфологический анализатор [33]. По своему объему он сопоставим с [14]. Кроме этого, [33] также поддерживает английский язык и при работе с англоязычными текстами используются словари этого анализатора.

Разработанный комплекс программ может применяться для поиска в сети Интернет, базах данных, системах документооборота, больших коллекциях электронных документов. Комплекс программ будет весьма полезен при организации поиска в быстро обновляющихся массивах текстовых данных, например электронных газет и журналов.

Глава 1

Формальные модели и обзор существующих структур данных и алгоритмов

1.1. Формальная модель текста

Для разработки эффективного программного комплекса требуется разработать адекватную исследуемой проблеме модель набора текстов, следуя общим принципам математического моделирования (см., например, [26]). Нужно выделить наиболее важные свойства текстов и проигнорировать те свойства, которые нас не интересуют. Текст является с одной стороны простым объектом, например, можно считать его последовательностью символов – наиболее простое представление. С другой стороны, чем больше углубляться в изучение объекта, тем больше возникает проблем при построении модели текста.

Представление текста, от простого к сложному:

1. Последовательность символов.
2. Выделив отдельные слова, можно рассмотреть текст как последовательность слов.

3. Можно разбить текст на предложения и абзацы.

4. Применить анализ морфологии языка.

5. Если это HTML-текст, то можно выделить связи между разными текстами на основании наличия гиперссылок.

6. Для обычных текстов можно определить взаимосвязи на основании нахождения определенных слов, например, имен собственных или специфических терминов в разных документах.

7. Синтаксический анализ текста, определение частей речи, грамматических зависимостей между словами.

8. Анализ смыслового содержания текста.

При переходе от одного представления к другому усиливается сложность формализации и разработки модели. В последнем пункте создать модель уже представляется достаточно затруднительным, с учетом того, что в тексте может быть отображено все многообразие мира. Для решения поставленной задачи поиска нам достаточно пунктов 2 – 4.

Обрабатываемый текст является последовательностью предложений. Каждое предложение является последовательностью слов. Таким образом, каждому слову можно сопоставить в соответствие запись со следующими полями:

1. $WORD\_ID$ – идентификатор слова.

2. $SEQUENCE\_ID$ – идентификатор предложения. Например, порядковый номер предложения.

3. $NUMBER$ – порядковый номер слова в тексте.

4. $SEQUENCE\_NUMBER$ – порядковый номер слова в предложении.

5. $DOCUMENT\_ID$ – идентификатор документа.

Обрабатываемый набор текстов можно представить как набор подобных записей. Это позволит искать в наборе текстов как отдельные слова, так и фразы. Недостаток данной модели заключается в том, что поиск с учетом морфологии языка здесь достаточно сложен.

1.2. Формальная модель текста с учетом морфологии

Обрабатываемый текст является последовательностью предложений. Каждое предложение является последовательностью слов. В случае, если слово входит в словарь морфологического анализатора, то анализатор возвращает идентификатор слова в словаре и набор идентификаторов базовых форм для данного слова. В случае, если слово не входит в словарь морфологического анализатора, то требуется сохранять его в отдельном словаре, в этом случае можно считать, что слову соответствует идентификатор в данном словаре, у него одна базовая форма и идентификатор этой базовой формы совпадает с идентификатором слова. Таким образом, каждому слову можно поставить в соответствие запись со следующими полями:

1. $KNOWN$ – признак. 1 – слово входит в словарь морфологического анализатора, 0 – не входит.

2. $WORD\_ID$ – идентификатор слова.

3. $BASE\_IDS$ – набор идентификаторов базовых форм слова.

4. $SEQUENCE\_ID$ – идентификатор предложения. Например, порядковый номер предложения.

5. $NUMBER$ – порядковый номер слова в тексте.

6. $SEQUENCE\_NUMBER$ – порядковый номер слова в предложении.

7. $DOCUMENT\_ID$ – идентификатор документа.

Данную запись далее будем назвать записью о вхождении слова в документе. Текст после морфологического анализа можно представить как набор подобных записей. База данных, содержащая подобные записи, позволяет выполнять различные запросы. Например, можно для заданного слова получить набор всех его вхождений в обработанных документах. Если мы ищем некоторую фразу, то получив для каждого слова фразы набор его вхождений, можно определить, в каких документах встречается фраза целиком. Для хранения подобных записей можно использовать различные структуры данных.

1.3. Формальная модель базы данных поисковой системы

В простейшем случае база данных поисковой системы состоит из следующих компонентов:

1. Список документов – файл, содержащий описания обработанных документов.

2. Репозитарий – файл, содержащий текст обработанных документов.

3. Индекс – сохраненный в каком-то виде массив записей о вхождении слов.

4. Словарь для слов, не входящих в словарь морфологического анализатора.

В списке документов сохраняется информация об обработанных документах.

Например, для каждого документа можно сохранять следующую информацию:

1. $DOCUMENT\_ID$ – идентификатор документа, в простейшем случае номер документа.

2. $NAME$ – полное имя документа.

3. $SIZE$ – размер документа.

4. $REPOSITORY\_ID$ – идентификатор для записи о документе в репозитарии.

5. $CHARSET$ – кодировка документа.

6. $FORMAT$ – формат документа.

Можно также сохранять самую различную вспомогательную информацию в зависимости от решаемой задачи.

В репозитарии сохраняется текст документов. Репозитарий предназначен для быстрого извлечения фрагмента текста, содержащего результат поискового запроса.

Можно считать, что репозитарий хранит в себе записи со следующими полями:

1. $REPOSITARY\_ID$ – идентификатор документа в репозитарии.

2. $TEXT$ – текст документа.

Дополнительно в репозитарии может хранится самая различная информация, например, контрольная сумма текста.

При хранении текста в репозитарии имеет смысл использовать алгоритмы сжатия. В этом случае дополнительные данные могут хранится или в репозитарии, или в списке документов.

1.4. Формальная модель программного комплекса поисковой системы

Поисковая система может включать в себя следующие компоненты:

1. Источник данных – модуль, отвечающий за получение обрабатываемых текстов и информации о них. В случае веб-поиска в данный модуль может включаться модуль, который загружает документы из сети Интернет. В случае различных текстовых коллекций данный модуль читает документы прямо с устройства внешней памяти, где они хранятся. Могут быть и другие источники данных, например, почтовые сообщения из некоторой системы или реляционные базы данных.

2. Модуль определения формата файла – данный модуль, получая исходный файл, может определять его формат и извлекать из него текст. Например, на вход данному модулю могут поступать файлы в форматах PDF, HTML, XML, а на выходе мы получаем обычный текст, который передается сканеру текста.

Если на вход данному модулю подается некоторый файл-контейнер, например, архив, то он может распаковывать его и извлекать из него вложенные файлы.

В случае веб-поиска модуль может обрабатывать гиперссылки в документах и передавать их источнику данных, обновляя его базу данных.

При необходимости можно встроить в систему модуль распознавания текста и вызывать его при обработке файлов-изображений на данном этапе.

3. Сканер текста – модуль, который осуществляет выделение в тексте предложений и слов. Сканер текста может вызывать модуль распознавания кодировки для определения кодировки текста. Слова он передает морфологическому анализатору. Текст файла передается в модуль репозитария. Полученные на основании работы морфологического анализатора записи о вхождении передаются модулю создания индекса.

4. Модуль распознавания кодировки осуществляет автоматическое распознавание кодировки. При этом он может использовать морфологический анализатор.

5. Модуль создания индекса сохраняет записи о вхождении слов в индексе.

6. Модуль репозитария сохраняет текст документа в репозитарии. Текст может обрабатываться перед сохранением модулем сжатия.

7. Модуль сжатия текста.

8. Модуль морфологического анализа.

9. Модуль поиска осуществляет поиск в полученном индексе.

1.5. Терминология

Строка – упорядоченная последовательность символов некоторого алфавита. Символы в данном тексте нумеруются, начиная с 1. S_i или $S[i]$ — символ строки S с номером i. Для строки S определена функция $length(S)$ или $|S|$ — длина строки. По умолчанию между строками определено отношение порядка, строки упорядочиваются лексикографически.

Подстрока строки S, начинающаяся i-го символа и заканчивающаяся j-м символом определяется как $A = S_i, S_{i+1}, \ldots, S_j$ и обозначается $S[i, j]$.

Префикс строки S – это подстрока строки S, начинающаяся с 1-го символа. Суффикс строки S – это подстрока строки S, заканчивающаяся на последнем символе строки S.

Поиск вхождения строки A в строке S понимается как нахождение такого числа q в S, что $S[q, q + |A| - 1] = A$. Может требоваться как нахождение первого вхождения (минимального числа q), так и всех вхождений.

1.6. Модель внешней памяти

Под внешней памятью автор понимает такие устройства, как жесткие диски, оптические диски (CD-ROM, DVD-ROM и т. д.), дискеты (как правило имеют малый размер и используются только для переноса данных между разными ЭВМ), флэш-диски и т. д. В основном для решения описываемых в работе задач используются жесткие диски, вследствие того, что на данный момент они имеют лучшие характеристики, а также

того, что другие виды внешней памяти имеют ряд ограничений: на CD-ROM и DVD-диски сложнее осуществлять запись, обычные флеш-карты и USB-диски, как правило, имеют фиксированное количество циклов перезаписи. Большой производительности также можно достичь, используя диски SSD (Solid State Drive).

Предполагается, что внешняя память поддерживает операции следующих типов: чтение блока некоторого размера в оперативную память и запись блока некоторого размера из оперативной памяти по внешнюю память. Время операции равно $C + O(S)$, где C – время операции, которое не зависит от размера блока, и примерно одинаковое для любых операций одного типа, а S – размер блока. Таким образом, если размер блока достаточно велик, то можно считать, что время операции линейно зависит от размера блока, так как C становиться мало по сравнению с $O(S)$. Если блок мал, то возможно, что C будет играть существенную роль и даже быть больше, чем $O(S)$.

Во многих алгоритмах размеры блоков выбираются кратными размеру сектора диска. Кроме того, некоторые операционные системы поддерживает определенные разновидности операций только с блоками, кратными сектору диска (например, ОС Microsoft Windows поддерживает чтение и запись без промежуточного кэширования только с такими блоками, а операции с кэшированием – с блоками любого размера). Также обычно на устройстве внешней памяти существует какая-либо файловая система, размер блока рекомендуется выбирать кратным размеру кластера этой файловой системы. Последняя рекомендация не является обязательной.

1.7. B-деревья

B-деревья были впервые упомянуты в 1972 г. Р. Байером, Э. Мак-Крейтом [16] и независимо от них М. Кауфманом [не опубликовано]. Также B-деревья упомянуты в [13]. Предназначены B-деревья для решения задачи, часто возникающей в компьютерных науках, эта задача имеет две разновидности.

Первый вариант: пусть у нас есть некоторое множество S элементов определенной природы, и мы хотим решать задачу принадлежности заданного элемента множества S некоторому заданному подмножеству Q.

Второй вариант заключается в том, что элемент множества S разделен на две части, называемые ключ и значение, и требуется найти в заданном подмножестве Q элемент с заданным ключом.

Для решения этой задачи существует много структур данных: сбалансированные бинарные деревья (красно-черные и АВЛ-деревья [1]), 2-3 деревья, treap [8, 15], хеш-таблицы и т. д. Как правило, эти структуры непосредственно не применимы при использовании их во внешней памяти. Не вдаваясь в подробности организации 2-3 деревьев, можно отметить, что по своей структуре B-деревья в значительной мере похожи на них.

Эти задачи похожи и могут быть сведены одна к другой. Первая задача может рассматриваться как вторая, если считать что значение элемента имеет нулевой размер. Вторая задача может рассматриваться как первая, если не разделять элемент на ключ и значение и рассматривать его целиком. B-дерево можно использовать для организации дополнительного словаря для слов, не входящих в словарь морфологического анализатора. См. раздел 1.3.

Теоретически B-дерево можно использовать для создания индекса,

если в качестве ключа взять идентификатор базовой формы слова $BASE\_IDS$, а в качестве значения – остальную часть записи о вхождении слова. Иными словами, для каждого элемента из $BASE\_IDS$ нужно вставлять в B-дерево пару из этого элемента и записи о вхождении слова, исключая из нее $BASE\_IDS$. Однако, в данном случае при создании индекса количество операций записи будет весьма велико, поэтому данный подход не применим при большом объеме исходных данных.

1.7.1. B-дерево при фиксированном размере элементов

Рассмотрим первую задачу. B-дерево состоит из некоторого количества узлов, каждый из которых храниться в одной странице внешней памяти. Страница это блок внешней памяти выбранного фиксированного размера H. Обычно размер страницы выбирают в диапазоне 4-32 килобайта. Узел, не имеющий потомков, называется листом, а имеющих их – промежуточным узлом. Для элементов должно быть определено отношение порядка.

При этом выполняются следующие условия:

* В каждом узле храниться не более m элементов.
* Каждый узел, за исключением корневого узла, хранит не менее $m/2$ элементов.
* Все листья находятся на одном уровне.
* Промежуточный узел с k элементами имеет $k + 1$ потомков.
* Корневой узел, если не является листом, то имеет по крайней мере двух потомков.

Число m выбирается исходя из размера элемента (он предполагается фиксированным) и размера страницы H, так чтобы в страницу H входило как можно больше элементов.

Элементы в узлах хранятся в порядке возрастания. Пусть у нас есть множество элементов $D_1, \ldots D_s$, упорядоченные по возрастанию. На рисунке 1.1 отображено соответствующее B-дерево.

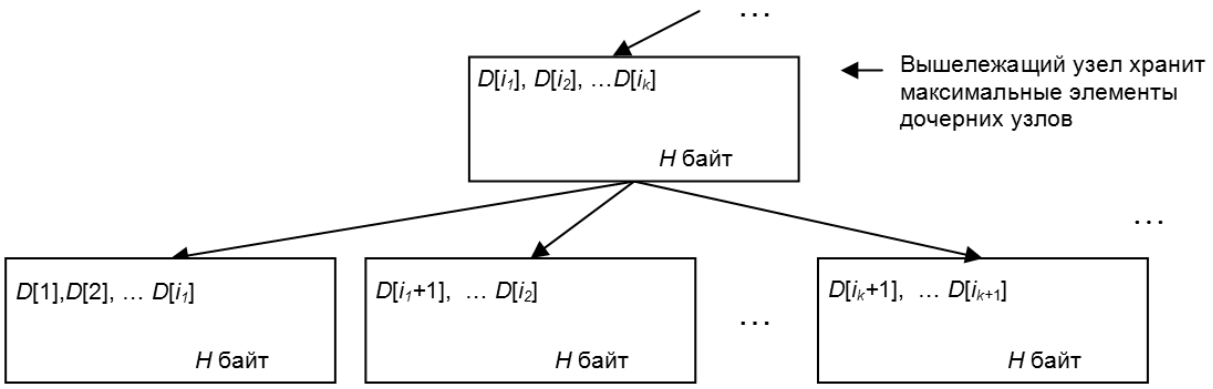

Рис. 1.1: Структура B-дерева

Каждый промежуточный узел хранит в себе максимальные элементы всех своих дочерних узлов, кроме последнего, и указатели на каждый из своих дочерних узлов. Эта структура позволяет легко осуществлять поиск.

Для B-дерева справедливо следующее утверждение:

Утверждение 1 *Высота B-дерева равна* $O(log_H S)$.

1.7.2. Поиск элемента в B-дереве

Поиск элемента V в B-дереве:

1. P – корень дерева.

2. Если у узла P нет потомков, то ищем V в наборе элементов P и возвращаем результат, иначе переходим на следующий шаг.

3. $E_1, ...E_k$ – элементы P, а $P_1...P_{k+1}$ – потомки P, при этом E_j – это максимальный элемент у P_j. Ищем такой номер i, чтобы V был меньше или равен E_i. Если такого i нет, т. е. V больше всех элементов P, то присваиваем $P = P_{k+1}$, иначе присваиваем $P = P_i$. Переходим на шаг 2.

Вычислительная сложность поиска, т. е. число обращений к внешней памяти, равна высоте дерева: $O(log_H S)$ где S – количество элементов в дереве. При достаточно большом размере страницы высота дерева редко превосходит 3 – 4.

Таким образом, справедливо следующее утверждение:

Утверждение 2 *Поиск в B-дереве требует $O(log_H S)$ чтений страниц из внешней памяти.*

1.7.3. Вставка элемента в B-дерево.

В процессе вставки в B-дерево нужно найти лист, где мог бы находится элемент с помощью алгоритма поиска и вставить элемент в этот лист. При этом в найденном листе может не оказаться требуемого места для вставки элемента, т. е. там может уже быть m элементов. В этом случае лист разделяется на два листа, в каждый из которых помещается от $m/2$ до m элементов. Максимальные элементы листьев должны быть помещены в вышележащий узел, что может привести к разделению вышележащего узла и т. д. Даже если лист не требуется разделять на две части, может потребоваться обновить вышележащий узел, если максимальный элемент листа изменился.

Утверждение 3 *Вставка элемента в B-дерево также требует* $O(log_H S)$ *обращений к внешней памяти.*

Можно также реализовать процедуру удаления элемента из B-дерева за $O(log_H S)$ обращений к внешней памяти.

При реализации B-дерева для решения второй задачи, т. е. когда элемент множества S разделен на две части, называемые ключ и значение и требуется найти в заданном подмножестве Q элемент с заданным ключом, можно просто сохранять в каждом узле в качестве элемента сразу и ключ, и значение. Однако значение требуется на самом деле хранить только в листьях B-дерева. Это приводит к разделению узлов дерева на два класса и введению вместо числа m чисел m_1 и m_2, так как в листья и в промежуточные узлы могут входить различное количество элементов. При этом алгоритмы изменятся незначительно и их вычислительная сложность не изменится. Даже при решении первой задачи, т. е. когда элемент не разделяется на ключ и значение, целесообразно разделять узлы дерева на два класса. В промежуточных узлах требуется хранить, помимо элементов, указатели на дочерние узлы, в листьях же хранить указатели не требуется, следовательно, в листьях может поместиться больше элементов, чем в промежуточных узлах.

1.7.4. B-дерево при произвольном размере элементов

Заметим, что мы намеренно избегали использования переменной m в алгоритме вставки. Там эта переменная присутствует только в скобках в виде комментария. По сути условия для B-дерева можно переформулировать без использования m:

* В каждом узле должно храниться максимальное количество элементов, которое помещается в страницу.

* Каждый узел, за исключением корневого узла, заполнен по крайне мере наполовину.

* Все листья находятся на одном уровне.

* Промежуточный узел имеет на единицу больше потомков, чем количество хранимых в нем элементов.

* Корневой узел, если он не является листом, имеет по крайней мере двух потомков.

При этом также можно отказаться от того, что элементы имеют фиксированный размер. Элементы в B-дереве при подобном определении могут иметь произвольный размер, только он должен быть ограничен некоторым числом, значительно меньшим, чем половина H. В этом случае сохранятся все оценки работы алгоритмов. Например, B-дерево может быть использовано для хранения набора слов естественного языка, например, русского. Такие слова имеют переменный размер, но, как правило, не очень большой.

Если размер элементов не может быть ограничен некоторым числом, то требуется хранить элементы отдельно, а в B-дереве хранить указатели на элементы вместо самих элементов. При этом во время поиска или вставки требуется определять, больше или нет заданный элемент некоторых элементов в B-дереве, а для этого нужно извлекать элементы, указатели которых хранятся в B-дереве оттуда, где они хранятся. Если сами элементы хранятся в отдельных файлах, то для извлечения элементов из этих файлов требуются отдельные операции внешней памяти.

Примером подобных структур данных является String-B дерево, описанное в разделе 1.10. В нем хранятся текстовые строки произвольной

длины.

1.7.5. B-деревья при заполнении узлов на 2/3

Почему мы требуем заполнения узлов только наполовину? Можно ввести требование заполнения узлов на 2/3. В этом случае требуется изменить алгоритмы вставки и удаления. Например, в том случае, когда элементы не помещаются в страницу, при вставке требуется смотреть на какую либо страницу рядом, но находящуюся на том же уровне. И если у обеих страниц в сумме есть место для хранения элементов общего набора, то мы должны сохранять набор в них. Если общий набор не помещается в двух страницах, то при добавлении новой страницы он может быть разделен так, чтобы каждая страница была заполнена на 2/3. Эта техника начинает работать на этапе, когда у B-дерева есть по крайней мере 3 страницы (корень и два листа). При использовании этого подхода процедуры вставки и удаления усложняются и требуют больше операций внешней памяти, но происходит выигрыш по размеру дерева на диске и, возможно, скорости поиска соответственно.

1.7.6. Кэширование

При обращении к B-дереву целесообразно держать в оперативной памяти несколько верхних уровней. Например, для хранения первых двух уровней требуется незначительный объем памяти, а для хранения только первого уровня требуется всего H байт.

Хорошо работает также схема, когда в оперативной памяти хранить-ся в виде очереди определенное фиксированное количество страниц. При этом, как только происходит обращение к странице в оперативной памяти, эта страница переходит в начало очереди, а как только требуется в

кэш поместить страницу, которой нет в кэше, то из очереди удаляется последняя страница. Наиболее часто доступ происходит к страницам дерева, расположенным на первых уровнях, таким образом эти страницы будут постоянно находиться в начале очереди.

1.8. Инвертированные файлы

1.8.1. Идея инвертированных файлов

Инвертированные файлы являются одним из основных инструментов для поиска в текстах на естественных языках. Общая идея заключается в следующем: для каждого слова текста создается список, в котором храниться информация обо всех вхождениях данного слова во всех документах. В зависимости от задачи эта информация храниться с точностью до позиции слова в тексте, номера слова в тексте, номера абзаца, страницы, или просто номера документа. При поиске определенного слова нужная информация просто выдается из этого списка. При поиске набора слов или фразы обрабатываются несколько списков. Инвертированные файлы позволяют решать все три указанные во введении задачи поиска.

Как правило, для организации инвертированных файлов используется внешняя сортировка слиянием, в результате получившиеся файлы практически невозможно обновлять, если они построены на основе большого объема данных.

Обычно для организации индекса, в частности для хранения списка записей о вхождении слов (см. раздел 1.2), используются именно инвертированные файлы.

1.8.2. Внешняя сортировка слиянием

Пусть дан файл с содержащимися в нем записями, между которыми определено отношение порядка. Требуется получить новый файл, в котором содержаться те же записи, но отсортированные по возрастанию.

Выберем некоторое число X.

1. Читаем последовательно X элементов из файла, сортируем их в памяти и записываем в новый временный файл. Делаем это до тех пор, пока файл не кончиться. После этого старый файл удаляем. В результате получается файл, состоящий из групп отсортированных элементов. Этот шаг можно опустить, если считать $X = 1$, но выбор большего числа X значительно ускоряет алгоритм.

2. Пока X меньше чем размер файла присваиваем $i = 0$ и выполняем шаг 3.

3. Читаем элементы у групп с номерами i и $i + 1$. В каждый момент у нас есть текущий элемент у каждой из групп. Выбирая из этих элементов минимальный, мы записываем его в новый файл и читаем следующий элемент группы, которой указанный элемент соответствовал. Действуем так, пока у нас есть элементы в обоих группах. Далее дописываем элементы оставшейся группы в файл. В результате в новом файле создается группа с отсортированными элементами обоих старых групп. Увеличиваем i на 2. Если мы не прочитали весь файл, переходим на шаг 3, иначе удаляем старый файл, увеличиваем X в два раза и переходим на шаг 2.

Этот процесс отображен на рисунке 1.2.

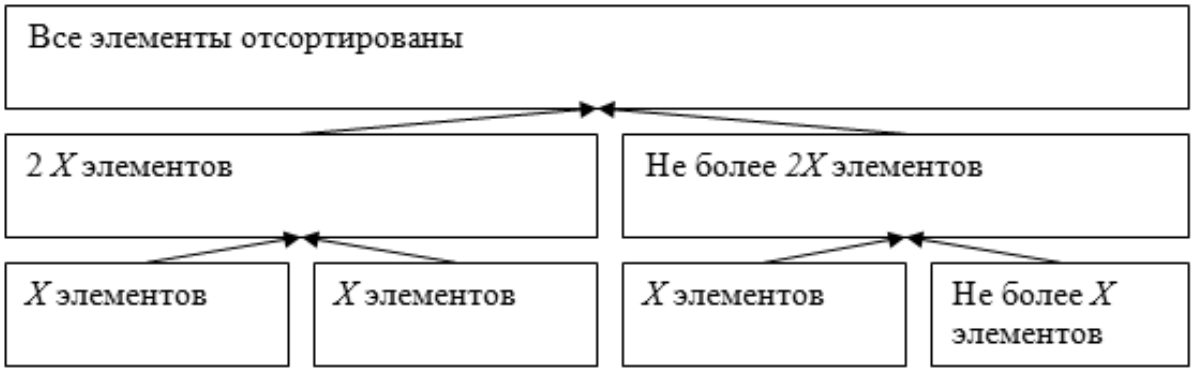

Рис. 1.2: Внешняя сортировка слиянием

1.8.3. Создание инвертированного файла

1. Чтение исходных документов, для каждого вхождения слова запись во временный файл позицию слова.

2. Внешняя сортировка временного файла.

3. Запись инвертированного файла на основе отсортированного временного файла.

В результате получается файл, в котором для каждого слова хранится подряд информация обо всех вхождениях данного слова во всех документах. Также предусматривается создание оглавления, содержащего список слов и соответствующие смещения в инвертированном файле для нахождения информации о конкретном слове. После этого информацию из файла легко извлекать и тем самым осуществлять поиск.

Значительное время в создании инвертированного файла занимает внешняя сортировка. Для нее требуется $O(log_2(N/X))$ проходов по файлу, где N – общее количество записей в файле. В частности, поэтому следует выбирать число X побольше.

В принципе, можно объединять группы не по 2, а брать сразу большее количество групп, например k групп. В этом случае количество прохо-

дов по файлу будет $O(log_k(N/X))$. Но вследствие ограничений устройств внешней памяти большое число k брать нельзя.

Инвертированные файлы и их аналоги используются в современных поисковых системах, например, в Google и Яndex.

Для создания подобного индекса можно использовать B-дерево, взяв в качестве элемента дерева запись из слова и информации о его вхождении. Отношение порядка между элементами B-дерева определяется на основании слова. Следует отметить, что при добавлении в B-дерево N элементов, количество обращений к внешней памяти будет $O(N \times log_H(S+N))$, где S – текущее количество записей в B-дереве. При этом, обращения будут осуществляться в разные места диска. Запись в разные места диска осуществляется существенно медленнее, чем последовательная запись. В результате для создания индекса с помощью B-дерева требуется большое время.

В отличие от B-деревьев, при создании инвертированных файлов операции обращения к внешней памяти осуществляются последовательно. Поэтому инвертированные файлы создаются быстрее. Однако, для инвертированных файлов стоимость добавления N записей будет $O(N+S)$, где S–текущее количество записей в инвертированном файле. Т. е. для добавления данных практически требуется перезаписать весь индекс.

Мы имеем две структуры данных, со своими достоинствами и недостатками.

Инвертированные файлы можно создавать быстрее чем B-деревья, но для добавления даже небольших данных требуется переписать весь индекс. B-дерево создается гораздо дольше (на больших объемах исходных данных его практически невозможно использовать), но его легко обновлять.

Возникает естественное желание получить структуру данных, обладающую достоинствами обеих упомянутых структур. Требуется разработать новую структуру данных, которая по скорости создания индекса и поиска не уступает инвертированным файлам и при этом обладает возможностью быстрого обновления.

1.9. Суффиксные массивы

Суффиксные массивы применяются при поиске в неформатированных текстах. Текст понимается как строка и требуется найти все вхождения заданной строки в тексте. Далее $S = S_1, ..., S_n$ – текст, а $W_1, ..., W_p$ искомая строка.

Суффиксный массив – это отсортированный список суффиксов текста. Поиск в таком массиве осуществляется при помощи бинарного поиска. Вычислительная сложность поиска составляет $O(plog_2n)$. При помощи сохранения информации о длинах общих префиксов некоторых суффиксов можно разработать алгоритм с вычислительной сложностью $O(p + log_2n)$.

Суффиксный массив может быть реализован во внешней памяти. Для этого текст S сохраняется в файле. Каждый суффикс текста S определяется одним числом – смещением в этом файле, относительно начала файла. Суффиксный массив в результате – набор чисел. При реализации в оперативной памяти используется аналогичный подход, только вместо файла рассматривается буфер в памяти, в который помещается текст S. Далее файл разделяется на страницы фиксированного размера H. Соответственно для бинарного поиска требуется осуществить $O(log_2n)$ чтений страниц внешней памяти.

Обычно строка S из n символов занимает в памяти n байт, а суффиксный массив $4n$ байт. Поиск строки W в суффиксном массиве A, построенном на основе строки S. Элементы массива нумеруются от 0.

1. Если $W \leq S[0]$, то выход из алгоритма, результат поиска 0.
2. Если $W > S[n-1]$, то выход из алгоритма, результат поиска n.
3. $L = 0, R = n - 1$.
4. Пока $R - L > 1$ выполнять
   (a) $M = (L + R)/2$
   (b) Если $W \leq S[M]$, то $R = M$, иначе $L = M$
5. Выход из алгоритма, результат поиска R.

Сложность данного алгоритма $O(plog_2n)$, так как $O(log_2n)$ – число итераций пункта 4. На каждой итерации требуется не более p сравнений между символами строк для сравнения двух строк.

1.10. String B-деревья

String 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. Длины строк ничем не ограничиваются. Строки могут быть на любом алфавите.

Далее изображены три строки неограниченной длины, записанные в файле большого размера и разделенные символом с кодом 0.

п	а	р	о	в	о	з	**0**	п	а	р	о	х	о	д	**0**	с	а	м	о	л	е	т	**0**

String-B дерево позволяет найти строки с заданным префиксом в наборе строк, а также найти все вхождения данной строки в какую-либо строку заданного набора строк. Эта задача сводится к предыдущей путем помещения в дерево всех суффиксов каждой строки исходного набора.

Дерево является сбалансированным, каждый его узел хранится в отдельной странице во внешней памяти. В листах дерева хранится заданный набор строк, упорядоченный в лексикографическом порядке, причем хранятся не сами строки, а указатели на них. Отдельное хранение строк позволяет работать со строками практически 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 храниться в одной странице внешней памяти.

Дерево, построенное на основе некоторого набора строк D, выглядит следующим образом. D упорядочивается в соответствии с лексикографическим порядком 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. Следует отметить, что в отличие от описанного ранее B-дерева, в промежуточных узлах хранится по два элемента для каждого из нижележащих узлов.

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 храниться в текущем узле. Поскольку 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 дерево PATRICIA [29]. С его 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, обычно берется символ с кодом 0, за счет чего любая строка может только совпадать полностью с некоторой другой строкой. 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.

Алгоритм поиска строки S в String-B дереве:

1. $P =$ корневой узел
2. Загружаем узел P из файла
3. Если $S \leq L[P]$ выводим все строки; выход
4. Если $S > R[P]$ выход, строки нет в дереве
5. $j = Search(S, P)$;

6. Если P – лист выводим все строки дерева, начинающиеся со строки в P с номером j, имеющие S своим префиксом; выход.

7. Иначе, если P – не лист $P[j] = R[T]$ для некоторого узла T то полагаем $P = T$, загружаем узел P из внешней памяти и переходим на шаг 5.

8. Если $R[j] = L[T]$ для некоторого узла T, то полагаем P равным самому левому нижележащему листу для узла T, и выводим все строки дерева, начинающиеся с $L[P]$, имеющие S своим префиксом; выход.

При поиске в дереве поиск в наборе строк узла P осуществляется с помощью процедуры $Search$. Можно, например, реализовать бинарный поиск, но при этом будет произведено во время работы процедуры $O(log_2 m)$ доступов к внешней памяти, так как в узле дерева хранятся не сами строки, а указатели на них. Феррагина и Гросси предлагают хранить в узлах дерева дополнительную структуру – PATRICIA [29], за счет чего можно уменьшить число доступов к внешней памяти, которые требуются процедуре $Search$ до 1.

Утверждение 4 *Поиск строки S в дереве потребует $O((length(S)+occ(S))/m + log_m k)$ где $length(S)$ – длина строки, $occ(S)$ – количество искомых вхождений строки S в дереве, k – количество строк в D. Кроме $O((length(S)+occ(S))/m+log_m k)$ потребуются occ дисковых операций для загрузки occ строк из внешней памяти, так как они хранятся в дереве в виде указателей.*

Утверждение 5 *Объем внешней памяти для хранения дерева равен $O(k/m)$ дисковых страниц. При этом также требуется отдельно*

хранить массив строк D в последовательном файле или в каком-либо другом виде.

1.11. Google

Поисковая система Google – одна из самых известных поисковых систем. В этом разделе будет описано ее примерное строение. Описание дано на основе информации приведенной в [20] (2000 г.).

Состав системы:

1. Лексикон – словарь из 14 миллионов слов, примерно 256 мегабайт в оперативной памяти, без учета новых слов, найденных в документах в процессе сканирования.

2. Индекс документов хранит описание документов.

3. Набор хранилищ (всего 64), каждое из которых соответствует некоторому подмножеству слов. Если документ хранит в себе слово, соответствующее некоторому хранилищу, то в хранилище записывается идентификатор документа, а за ним список всех позиций слова в документе. Каждое вхождение описывается 2-мя байтами. При этом для хранения позиции слова используется 12 бит, и все позиции, большие 4096, хранятся как 4096. Хранилища отсортированы, и для каждого слова в лексиконе храниться указатель на то место в соответствующем хранилище, где начинается информация о слове. Это сделано примерно так, как в инвертированных файлах.

   Есть два набора хранилищ, первый хранит информацию об авторах документов и ключевых словах, второй набор хранит информацию о текстах документов. Таким образом, поиск вначале производится

по авторам и ключевым словам, а только потом по всему объему текстов.

4. Система рангов. В Google реализована система подсчета рангов документов для того, чтобы выдавать пользователю более подходящие документы. Также присутствует система распознавания похожих документов.

5. Отдельные модули осуществляют поиск документов в Интернет.

6. Отдельные модули осуществляют сортировку хранилищ.

7. Репозитарий хранит в себе все найденные документы в сжатом виде.

В [20] дано описание производительности системы:

Объем документов (веб-страниц)	147,8 Гб
Репозитарий (сжатый)	53,5 Гб
Короткий индекс (первый набор хранилищ)	4,1 Гб
Полный индекс (второй набор хранилищ)	37,2 Гб
Лексикон	293 Мб
Индекс документов	9,7 Гб
Временные данные гиперссылок	6,6 Гб
База данных ссылок	3,9 Гб
Всего без учета репозитария	55,2 Гб
Всего с учетом репозитария	108,7 Гб
Число веб-страниц	24 миллиона

Скорость поиска	1-10 секунд
Сортировка	24 часа на 4-х ЭВМ параллельно

На сегодняшний день Google состоит из большого числа параллельно работающих машин.

1.12. Яndex

Яndex – одна из самых развитых систем поиска в сети Интернет, в большей степени нацеленная на поиск среди русскоязычных сайтов. Описываемая в данном разделе информация взята с сайта www.yandex.ru.

Для реализации индексов в Яndex используются инвертированные файлы. В Яndex реализован поиск по ключевым словам в полнотекстовом индексе с учетом морфологии языка. В системе реализованы модуль анализа морфологии, модули индексирования и поиска, сетевой «паук» для загрузки данных из Интернет, модуль вычисления релевантности и др. Модуль морфологического анализа поддерживает построение гипотез для слов, не входящих в словарь анализатора. Объем словаря примерно 90 тыс. слов. Есть словари для русского и английского языка. Поддерживается поиск с учетом логических операторов и поиск с учетом близости между словами.

Создаваемый индекс составляет около 1/3 объема текста (без учета графических файлов, тегов и пр.). Запись информации о вхождении слова сохраняется с точностью до позиции в тексте. Скорость индексации — 10-30 Мб/минуту на компьютерах класса Pentium II/III.

Данные приведены по состоянию на 5 марта 2002 года.

Глава 2

CLB-Деревья

В этой главе мы описываем новую структуру данных, разработанную автором, которая предназначена для поиска в большом объеме электронных документов, написанных на естественном языке [3,5,6,10–12]. Новая структура, которую мы назвали CLB-дерево, обладает рядом преимуществ по сравнению с другими структурами данных.

Достаточно давно используемые для поиска в текстах инвертированные файлы [30] имеют один существенный недостаток – их сложно обновлять. CLB-дерево – это структура данных, по скорости поиска не уступающая инвертированным файлам и их аналогам, но, в отличие от них, в CLB-дерево можно легко и быстро добавлять новые данные.

B-деревья и их вариации [16, 17] сложно использовать для поиска в естественных текстах большого объема из-за большого числа требуемых обращений к внешней памяти. String-B деревья [21, 22] предназначены в основном для поиска в неформатированных текстах, например, в текстовых представлениях молекул ДНК, и также тоже требуют большого числа обращений к внешней памяти. Суффиксные массивы [23, 27] при реализации по внешней памяти по производительности примерно эквивалентны B-деревьям.

CLB-дерево предназначено для поиска слов или словосочетаний в большом объеме текстовых файлов. При этом учитывается морфология языка этих текстов.

В качестве морфологического анализатора может использоваться анализатор, созданный Ю. С. Лукачом [14]. Морфологический анализатор содержит около 3,5 млн. словоформ русского языка, образованных от 205 тысяч базовых форм. Для каждого слова, известного анализатору он возвращает одну или несколько базовых форм. Слова, хранящиеся в словаре морфологического анализатора, составляют более 90% всех слов в типичных документах.

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 [14], построенная на базе алгоритма пошагового построения МДКА [25].

Также может использоваться широко известный морфологический анализатор [33]. По своему объему он сопоставим с [14]. Кроме этого, [33] также поддерживает английский язык и при работе с англоязычными текстами используются словари этого анализатора. Автором диссертации разработан морфологический анализатор, использующий словари [33], который значительно превосходит по скорости [33].

Исходные текстовые файлы, для которых строится CLB-дерево, далее будем называть документами. Под известными словами мы будем понимать слова, входящие в словарь морфологического анализатора, а под неизвестными – те слова, которые не входят в этот словарь. Известные слова могут иметь несколько базовых форм. Неизвестные слова всегда имеют одну базовую форму – само слово. Таким образом, каждому слову

сопоставляется набор его базовых форм (далее просто форм).

CLB-дерево расположено во внешней памяти, что приводит к ряду сложностей, так как внешняя память обладает некоторыми особенностями. Предполагается, что внешняя память разбита на страницы фиксированного объема. Внешняя память поддерживает операции двух типов: чтение страницы в оперативную память и запись страницы из оперативной памяти. По быстродействию внешняя память значительно уступает оперативной памяти компьютера, и поэтому основным критерием работоспособности алгоритма в данном случае является максимальное число операций с внешней памятью, которое ему требуется. Также следует отметить, что, как правило, чтение или запись нескольких страниц, расположенных последовательно, производится гораздо быстрее, чем чтение или запись страниц, расположенных не последовательно. Таким образом, время выполнения для одного типа операции внешней памяти предполагается равным $C + O(S)$, где C – постоянное время для всех операций этого типа, а S – размер данных.

2.1. Базовая идея CLB-дерева

В [12] автором диссертации описана начальная идея CLB-деревьев.

Предполагалось, что для каждой формы слова создается цепочка кластеров, состоящая из блоков фиксированного размера K, каждый из которых имеет ссылку на следующий блок цепочки. Каждая форма каждого слова текста сохранялась в B-дереве вместе с указателем на первый кластер цепочки. В цепочке кластеров для данной формы слова сохраняется информация обо всех вхождения данной формы в тексте. Например, для каждого вхождения каждого слова сохраняется номер или иденти-

фикатор соответствующего документа и позиция слова в этом документе. При этом для формы известного слова в памяти сохраняется последний кластер соответствующей цепочки, и добавление новой информации о соответствующем слове производится в этот кластер.

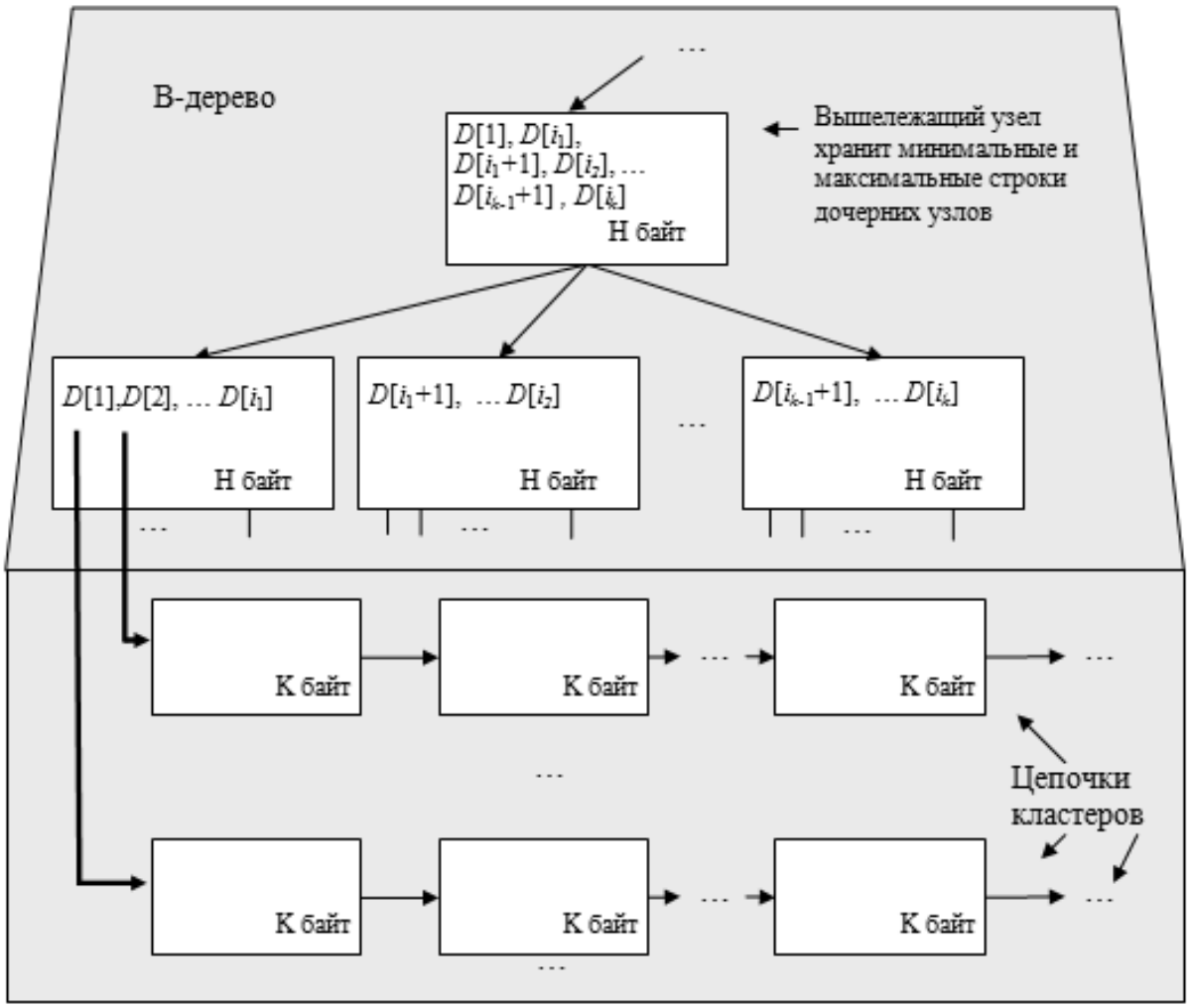

Рис. 2.1: Структура CLB-дерева вначале. Массив D в дереве составлен из всех форм слов (неизвестных и известных) данного текста, и храниться в B-дереве (в верхней части рисунка). Каждой форме каждого слова текста сопоставлена цепочка кластеров, содержащая информацию обо всех вхождениях данной формы в тексте

Таким образом, у нас есть файл, состоящий из блоков заданного раз-

мера. Несколько блоков объединяется в список: в каждом блоке из данного списка сохраняется ссылка на следующий блок.

При создании индекса, когда слово встречается первый раз, мы помещаем его в B-дерево и создаем список, состоящий из одного блока, в который записываем информацию о вхождении данного слова в тексте. Если слово встречается повторно, информация о следующем вхождении добавляется в данный блок. Если свободное место в блоке заканчивается, мы создаем новый пустой блок (увеличивая файл в размере) и прописываем на него ссылку в текущем блоке, и далее информация добавляется в новый блок. Таким образом, мы достигаем возможности легкого обновления индекса.

В [12] были даны теоретические оценки для количества дисковых операций при создании индекса и поиске с его помощью.

Однако, данный подход обладает рядом недостатков. В частности, блоки располагаются не последовательно, что замедляет скорость их чтения при поиске. Данная проблема была решена за счет разработки более эффективных алгоритмов, описание которых дано далее. Кроме того, некоторые слова редко встречаются в текстах. Соответственно, размер информации о вхождениях для таких слов меньше, чем размер одного блока. Следовательно, при описанной организации индекса у нас возникают частично заполненные блоки.

Пусть, например, некоторое слово встречается всего один раз. Информация об одном вхождении занимает несколько байт, например, номер документа 4 байта и позиция слова в документе – еще 4 байта, итого 8. При определенных оптимизациях информация об одном вхождении занимает 1 – 3 байта. Для хранения информации о вхождении данного слова выделяется блок в индексе, размер блока обычно 4 – 16 Кб. В ито-

ге место на диске расходуется неэффективно. Данная проблема также была решена.

2.2. Структура CLB-дерева, описание основного алгоритма создания индекса

2.2.1. Компоненты CLB-дерева

Каждой базовой форме слова соответствует некоторая цепочка блоков, которые назовем кластерами. В этой цепочке содержится описание всех вхождений данной базовой формы во всех документах. Каждый кластер имеет фиксированный размер K. Каждая цепочка кластеров описывается небольшой структурой-указателем.

CLB-дерево состоит из следующих компонентов:

1. Таблица указателей для базовых форм известных словоформ.
2. Словарь неизвестных словоформ – B-дерево, хранящее неизвестные словоформы и указатели для них.
3. Файл с цепочками кластеров.

Указатели для базовых форм известных словоформ хранятся в таблице – массиве, находящемся в оперативной памяти. Базовые формы неизвестных словоформ и указатели для них хранятся в B-дереве, оно расположено во внешней памяти и разбито на страницы фиксированного размера H.

2.2.2. Организация данных для эффективного чтения

Зафиксируем некоторое число c и соответствующее ему число $BLOCK\_L = 2^c$. Цель алгоритма заключается в том, чтобы организо-

вать хранение данных таким образом, чтобы список блоков был разбит на группы, и все блоки, входящие в заданную группу, были расположены в файле последовательно.

Например, если у нас есть 25 блоков и $BLOCK\_L = 8$, мы разбиваем 25 блоков на группы следующих размеров: 8, 8, 8, 1.

Тем самым весь список блоков разделяется на группы по $BLOCK\_L$ блоков, за исключением последней группы, в которой может быть меньше чем $BLOCK\_L$ блоков, т. к. длина списка может не делится нацело на $BLOCK\_L$.

При проведении экспериментов размеры блоков брались 4 – 16 килобайт. Число $BLOCK\_L$ было выбрано таким образом, чтобы суммарная длина группы последовательно располагающихся блоков была не меньше 8 Мб.

В результате скорость считывания данных из списка блоков практически совпадает со скоростью последовательного чтения данных.

Итак, кластеры объединяются в блоки кластеров. Кластеры в блоке располагаются последовательно. Максимальная длина блока $BLOCK\_L$ является числом степени 2. Последний кластер в блоке хранит в себе указатель на первый кластер следующего блока, если текущий блок не последний. Все блоки, кроме последнего, имеют длину $BLOCK\_L$.

2.2.3. Эффективное заполнение блоков

Для многих базовых форм слов объем информации, который следует хранить в цепочке, мал (меньше половины K). Для этого случая существует отдельный список кластеров. Каждый из таких кластеров делится на определенное количество равных частей, и для конкретной базовой

формы слова используется только одна из этих частей. В этом случае мы считаем, что цепочка кластеров состоит из соответствующей части указанного кластера. Остальные части могут использоваться другими базовыми формами слов, для которых также объем информации мал.

2.2.4. Поиск словоформы и извлечение информации о ней

1. Создается пустой список указателей.
2. Если словоформа известна, то морфологический анализатор выдает список номеров ее базовых форм. Каждый номер соответствует ячейке таблицы для базовых форм известных словоформ. Из этой ячейки, если она не пуста, извлекается указатель и добавляется в список указателей.
3. Если это неизвестная словоформа, то она ищется в B-дереве и, если она там есть, возвращается ее указатель и добавляется в список указателей.
4. Для каждого указателя из списка определяется цепочка кластеров, и считываются все данные из нее.

Это позволяет реализовать различные алгоритмы поиска. По сути, с точки зрения поиска CLB-дерево предлагает те же возможности, которые есть у инвертированных файлов.

2.2.5. Кэширование для слов, входящих в словарь морфологического анализатора

Очевидно, что если мы будем каждый раз при добавлении информации о вхождения слова записывать данные в файл, то мы будем создавать индекс очень долго.

Основная оптимизация осуществляется за счет использования морфологии языка. Слова, входящие в состав данного анализатора составляют 80-90слов в обычных тестах (например, художественная литература, новости). В словарь анализатора [14] входит 3,5 млн. словоформ русского языка, образованных от 205 тыс. базовых форм.

Словарь анализатора позволяет по словоформе получить набор ее базовых форм. Для большинства словоформ существует только одна базовая форма, однако для многих словоформ существует несколько базовых форм (например, форма «суда» может быть образована как от слова «суд», так и от слова «судно»). В данной работе автор отождествляет термины «слово» и «словоформа». Там, где это необходимо, используется термин «базовая форма слова».

При сохранении данных о вхождении слова в текст слово сначала приводится к своей базовой форме. Если базовых форм несколько, то информация добавляется для всех базовых форм.

Соответственно для каждой базовой формы слова существует список блоков, для хранения информации о вхождении данной базовой формы в текстах.

При определенном размере блока мы можем сохранять в оперативной памяти последний блок для каждого из этих списков блоков. Этот блок сохраняется на диске, только когда он становится полностью заполненным.

Данный подход можно применять только для слов, входящих в словарь морфологического анализатора, т. к. мы знаем количество базовых форм и можем заранее создать кэш требуемого размера.

Слов, не входящих в словарь анализатора, может быть гораздо больше, например, в одной из используемых автором текстовых тестовых

коллекций было 65 млн. различных слов, не входящих в словарь морфологического анализатора. О том, как обрабатывать такие слова, описано далее.

2.2.6. Кэширование для слов, не входящих в словарь морфологического анализатора

Подход, описанный в предыдущем подразделе, мы не можем использовать для слов, не входящих в словарь анализатора. В данном случае можно реализовать некие стандартные стратегии кэширования, например, при необходимости добавления нового блока в кэш выгружать из кэша блок, запись в который не производилась дольше всего.

Автором также был реализован подход, описанный далее. Слова, не входящие в состав морфологического анализатора, как правило, встречаются очень редко. Соответственно, суммарный объем информации об их вхождениях достаточно мал.

Предлагается сохранять информацию о вхождениях для нескольких слов в одном списке блоков. При этом, при записи информации о вхождении вместе с ней сохраняется тег, определяющий, к какому слову она относится.

Данный метод значительно сокращает количество операций записи при создании индекса, при этом эффективность поиска снижается незначительно.

Например, пусть в одном списке блоков сохраняется информация о вхождениях слов: ааа, ббб, ввв. Присвоим этим словам номера 1, 2, 3. Если мы ищем слово ааа, то мы считываем весь этот список блоков. При чтении из списка блоков информации о каждом вхождении считываем также и тег, соответствующий данному вхождению. Рассматриваем

только те записи, у которых тег равен 1.

При создании индекса контролируется, сколько раз уже встретилось слово. Если оказывается, что слово встречается слишком часто, то для него создается новый список блоков, в который переносится информация о вхождениях данного слова, далее в данном списке блоков будет сохранена информация только об этом слове.

Автором был проведен следующий эксперимент. Было зафиксировано число 4096, и для слов, число вхождений которых превышало данное число, создавались отдельные списки блоков. Для тех слов, число вхождений которых меньше или равно 4096, в один список блоков помещалась информация сразу о 32 различных словах. Обрабатывалось 35 Гб текстов, содержащих 69 млн. различных слов. В результате оказалось, что максимальное количество записей в списке блоков, для списков, содержащих несколько слов, было равно 19 431.

Данное число показывает, что эффективность не пострадала, т. к. объем указанных списков блоков не превышает нескольких десятков килобайт (т. к. одна запись занимает примерно 2-4 байта). Слов, которые встречаются больше чем 4096 раз, оказалось всего 19 593.

Для повышения быстродействия также используется алгоритм вставки пакета словоформ в B-дерево, подробно описанный в [12].

2.2.7. Общая структура системы индексирования и поиска

Общая структура системы индексирования и поиска отображена на рис. 2.2. В верхней части отображено CLB-дерево и его компоненты. Более подробно CLB-дерево и все его компоненты отображено на рис. 2.3.

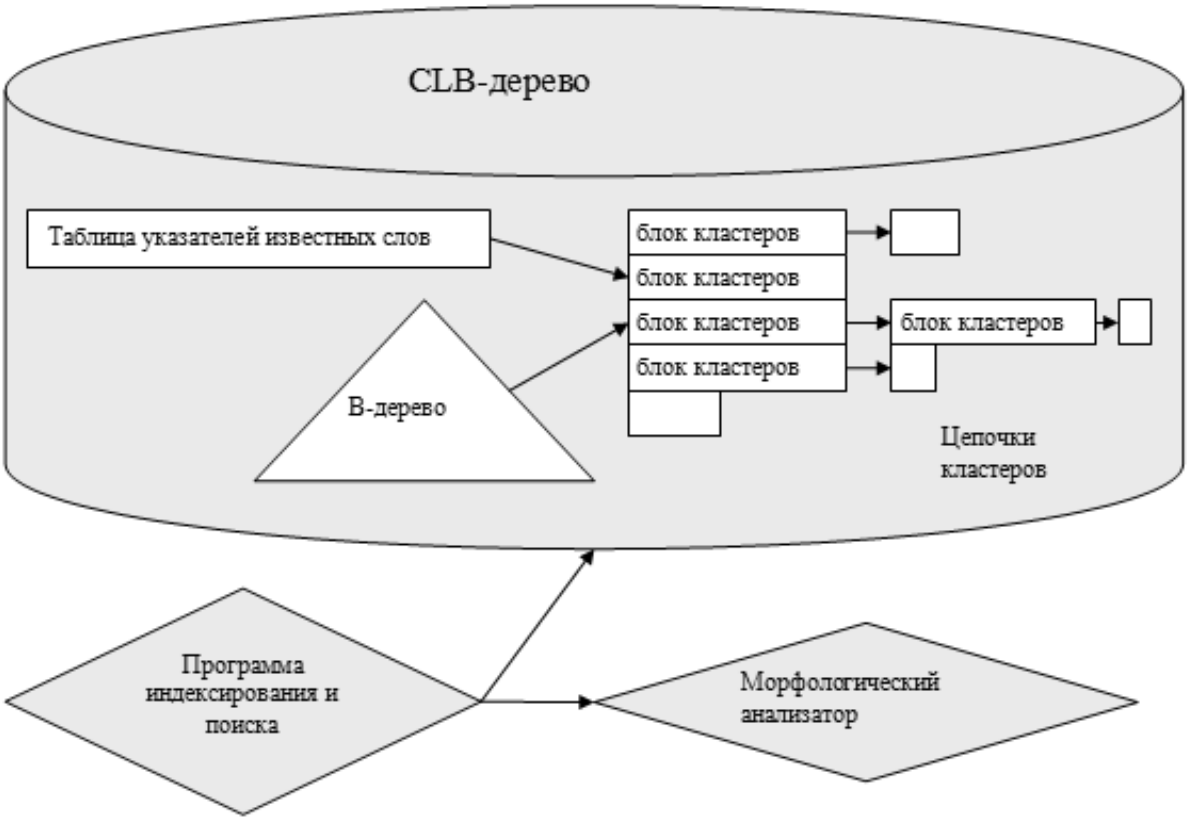

Рис. 2.2: Структура системы индексирования и поиска

2.2.8. Создание индекса

Описанная в [12] структура позволяет использовать только кластеры небольшого размера (в [12] – 512 байт), т. к. нужно хранить в оперативной памяти столько кластеров, сколько имеется базовых форм известных слов. Малый размер кластера приводит к большому числу дисковых операций. Увеличение размера кластера требует большого объема оперативной памяти.

Выход в том, чтобы разделить все базовые формы слов на группы, и записывать группы по отдельности. Определяем число K_GROUPS – количество групп, исходя из размера доступной оперативной памяти.

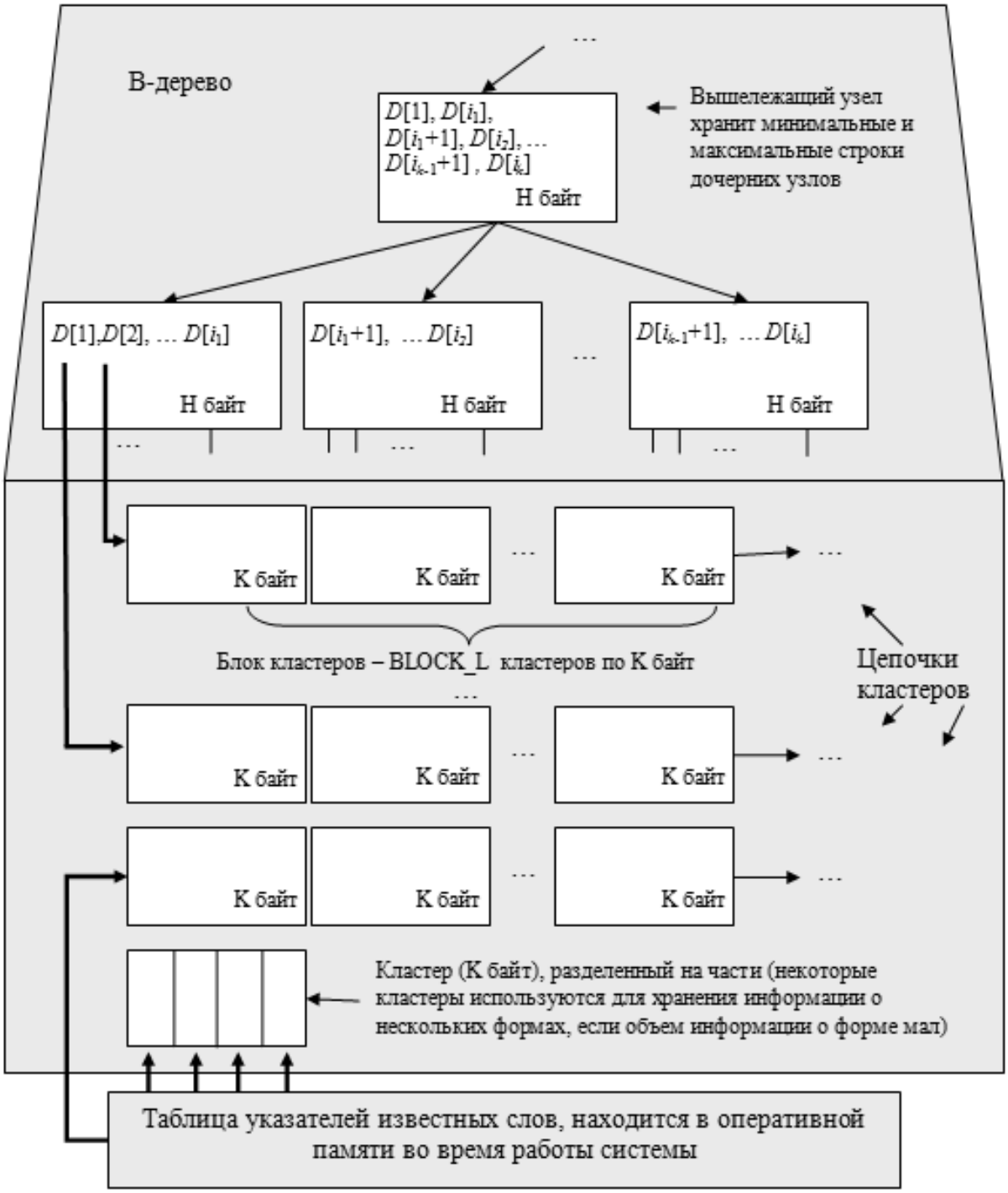

Рис. 2.3: Структура CLB-дерева. Массив D в дереве составлен из всех неизвестных слов данного текста, и храниться в B-дереве (в верхней части рисунка). Каждой форме каждого слова текста сопоставлена цепочка кластеров, содержащая информацию обо всех вхождениях данной формы в тексте

Пусть K_FORMS – количество базовых форм известных слов (на данный момент около 200000), тогда:

1. число кластеров в кэше $GROUP\_SIZE$ = объем кэша / K.
2. $K\_GROUPS = K\_FORMS/GROUP\_SIZE$, с округлением в большую сторону.

Будем использовать кэш и для неизвестных словоформ. Задаем число U_GROUPS – число групп для неизвестных словоформ. Номер группы здесь определим с помощью хеш- функции $HASH$, которая, например, суммирует все коды символов словоформы и делит сумму на U_GROUPS.

Всего групп $GROUPS = K\_GROUPS + U\_GROUPS$. Введем $GROUPS$ временных файлов. Первые K_GROUPS файлов для известных словоформ, остальные для неизвестных словоформ, нумерация идет от нуля.

При чтении текстов информация о словах в текстах сохраняется во временных файлах. Информация для каждой группы базовых форм слов сохраняется в своем временном файле. После завершения чтения каждая группа обрабатывается и сохраняется в индексе отдельно, с использованием своего кэша.

Алгоритм создания индекса

Чтение файлов. Для каждой прочитанной словоформы $WORD$:

1. Если это известная словоформа, анализатор возвращает список номеров базовых форм словоформы.
   (a) Для каждого номера N
   (b) $M = N/GROUP\_SIZE$

(c) В M-й временный файл записывается номер N, и информация о вхождении словоформы.

2. Если это неизвестная словоформа

   (a) $M = HASH(WORD)$

   (b) В $(M + K\_GROUPS)$-й временный файл записывается словоформа и информация о ее вхождении.

После чтения файлов производится запись в индекс в два этапа. Используются две процедуры. Процедура $InsertRecord$ применяется при первом добавлении данной базовой формы слова в индекс, создается новая цепочка кластеров. Процедура $AppendRecord$ применяется, когда данная базовая форма уже добавлялась в индекс, для нее уже существует цепочка кластеров.

Алгоритм записи в индекс. Первый этап. Запись в индекс информации об известных словоформах.

1. Для каждого i от 0 до $K\_GROUPS - 1$ выполняем шаги 2 – 3.

2. Для каждого номера формы N из i-го временного файла выполняем 3.

3. Если N-я ячейка таблицы пуста, то вызов $InsertRecord$, ее результат записываем в ячейку таблицы, иначе вызов $AppendRecord$ с указателем из ячейки и запись нового значения в ячейку.

Алгоритм записи в индекс. Второй этап. Запись в индекс информации об неизвестных словоформах.

1. Для каждого i от K_GROUPS до $GROUPS - 1$ выполняем шаги 2 — 4.

2. Пока все слова из i-го временного файла не прочитаны, выполняем шаги 3 — 4.

3. Чтение MAX_APPEND слов (или всех слов, если их меньше) и формирование пакета из слов. Число MAX_APPEND выбирается как можно больше, алгоритм вставки пакета слов в B-дерево описан в [12].

4. Вставка пакета в B-дерево, в процессе вставки для каждого слова один из двух вариантов:

   (a) если слова нет в B-дереве, то вызов $InsertRecord$, вставка слова вместе с указателем в B-дерево.

   (b) если слово есть в B-дереве, то вызов $AppendRecord$, обновление указателя в B-дереве.

Вторая проблема, которая усугубляется при увеличении размера кластера: в [12] для каждой базовой формы слова используется по крайней мере один кластер. Это приводит к пустотам в файле с кластерами. Будем использовать некоторые кластеры для хранения в них информации о нескольких базовых формах. Выберем число $PARTS$, являющееся степенью 2, и введем массив списков $PART\_LST$ из $log_2 PARTS$ элементов (нумерация от 0). Каждый список хранит в себе номера кластеров. Список с номером i хранит в себе кластеры, разбитые на 2^{i+1} частей (номер 0 соответствует двум частям, $log_2 PARTS - 1$ соответствует $PARTS$ частям). Каждая часть соответствует некоторой отдельной базовой форме слова. В конце кластера выделяем область, назовем ее таблицей частей, для хранения информации о том, какие части кластера заполнены. Размер этой области $PARTS/8$ байт, с округлением в большую сторону

(чтобы хранить $PARTS$ бит). Остальная область кластера делится на равные части. В таких кластерах не хранятся какие-либо указатели на следующие кластеры.

Поскольку запись информации о словах происходит по группам, нужно, чтобы для каждой группы массив $PART\_LST$ был свой. Перед шагом 2 в двух вышеописанных алгоритмов необходимо сменить текущий массив $PART\_LST$ на тот массив, который соответствует текущей группе слов. Вводятся списки свободных кластеров $FREE$, количество этих списков $max(log_2(BLOCK\_L), 1)$, нумерация от 0. Нулевой список хранит номера единичных свободных кластеров, i-й список хранит номера первого кластера для групп длины 2^i подряд расположенных кластеров.

Структура указателя на цепочку кластеров (далее указатель):

$FIRST$	Номер первого кластера
$LAST$	Номер последнего кластера
NUM	Номер кластера в блоке
$LASTDATA$	Размер данных в последнем кластере
$PART$	Номер части
$PARTCOUNT$	Количество частей в текущем кластере

На языке C++ это выглядит примерно так:

```
struct POINTER
{
        short NUM;
        unsigned int LASTDATA;
        struct
        {
                CLUSTERPOINTER FIRST;
```

```
                union
                {
                        CLUSTERPOINTER LAST;
                        struct
                        {
                                unsigned short PARTCOUNT;
                                unsigned short PART;
                        };
                };
        };
}
```

Поле NUM – номер текущего кластера в блоке, считается с 1. Если NUM равно 0, то информация о слове храниться в кластере, разделенном на части.

Процедура $AllocCluster$**():**

Процедура выдает номер свободного кластера.

Если $FREE[0]$ пуст, то добавление кластера в конец файла и N = номер добавленного кластера, иначе получение номера N из списка $FREE[0]$ и удаление его из этого списка. Возвращаем N в качестве результата.

Процедура $InsertRecord$**(Указатель** PTR**, добавляемые данные** $Data$**):**

Процедура создает цепочку кластеров и помещает в нее некоторую информацию о найденной форме слова. Процедура заполняет передаваемый ей указатель на создаваемую цепочку кластеров. В скобках после названия процедуры описаны ее аргументы.

1. $NEWSIZE$ = <Размер $Data$>

2. Если $NEWSIZE$ больше, чем размер частей у кластеров, имеющих две части (что маловероятно), то переход на шаг 2a, иначе на шаг 3

   (a) $N = AllocCluster()$

   (b) Запись $Data$ в кластер N, $PTR.FIRST = N$, $PTR.LAST = N$, $PTR.NUM = 1$, переход на шаг 13.

3. Определяем, на сколько частей должен делиться кластер, чтобы размер одной части был больше или равен $NEWSIZE$, в переменную $PARTCOUNT$ помещаем количество частей.

4. $INDEX = (log_2 PARTCOUNT) - 1$

5. Если $PART\_LST[INDEX]$ пуст, переход на шаг 5a, иначе на шаг 6

   (a) $N = AllocCluster()$

   (b) Добавление N в $PART\_LST[INDEX]$ и запись в таблицу частей этого кластера нулевых значений для всех частей. Переход на шаг 6.

6. Получение номера P кластера из $PART\_LST[INDEX]$.

7. $PTR.FIRST = P, PTR.NUM = 0$

8. Определение номера $PART$ свободной части по таблице частей кластера P.

9. $PTR.PART = PART, PTR.PARTCOUNT = PARTCOUNT$

10. Запись значения 1 в таблицу частей кластера P для части $PART$.

11. Если все части P заполнены, удаляем P из списка $PART\_LST[INDEX]$.

12. Запись $Data$ в кластер P по смещению $PART\times$ <размер части>.

13. $PTR.LASTDATA = NEWSIZE$.

Процедура *AppendRecord*(**Указатель** PTR**, добавляемые данные** $Data$)**:**

Процедура добавляет в существующую цепочку кластеров информацию о найденной форме слова.

$NEWSIZE = PTR.LASTDATA +$ <Размер $Data$>

Если $PTR.NUM = 0$, то информация сейчас храниться в кластере, разбитом на части, тогда выполняем процедуру 1, иначе выполняем процедуру 2; обе процедуры имеют те же параметры, что и *AppendRecord*.

Процедура 1

Данная процедура соответствует добавлению информации в кластер, если на данный момент кластер разделен на части.

1. Определяем $INDEX = (log_2 PTR.PARTCOUNT) - 1$

2. Если $NEWSIZE$ меньше или равен чем <размер части>, то переход на шаг 2а, иначе на шаг 3:

   (а) Добавление $Data$ в кластер $PTR.FIRST$ по смещению $PTR.PART\times$<размер части>$+PTR.LASTDATA$, переход на шаг 12.

3. Читаем информацию о слове, хранящуюся в кластере в промежуточный буфер $BUFFER$

4. Устанавливаем значение 0 в таблице частей кластера $PTR.FIRST$, освобождая часть.

5. Если раньше все части кластера были заполнены, т. е. сейчас свободна одна часть, добавляем $PTR.FIRST$ в $PART\_LST[INDEX]$.

6. Если все части кластера свободны, удаляем его из $PART\_LST[INDEX]$ и помещаем его в $FREE[0]$.

7. Если $NEWSIZE$ больше, чем размер частей у кластеров, имеющих две части, то переход на шаг 8, иначе на шаг 7a

   (a) Определяем, на сколько частей должен делиться кластер, чтобы размер одной части был больше или равен $NEWSIZE$, в переменную $PARTCOUNT$ помещаем количество частей.

   (b) $INDEX = (log_2 PARTCOUNT) - 1$

   (c) Если список $PART\_LST[INDEX]$ не пуст, то переход на шаг 7d, иначе на шаг 7(c)i

      i. $N = AllocCluster()$

      ii. Добавление N в $PART\_LST[INDEX]$ и запись в таблицу частей этого кластера нулевых значений для всех частей.

      iii. Переход на шаг 7d

   (d) Получение номера кластера P из $PART\_LST[INDEX]$.

   (e) Определение номера $PART$ свободной части по таблице частей кластера P.

   (f) Запись значения 1 в таблицу частей кластера P для части $PART$.

   (g) Если все части P заполнены, удаляем P из $PART\_LST[INDEX]$.

   (h) $PTR.PART = PART$, $PTR.PARTCOUNT = PARTCOUNT$, $PTR.FIRST = P$.

(i) Запись данных из буфера $BUFFER$ в кластер P.

(j) Добавление $Data$ в кластер P.

(k) Переход на шаг 12.

8. $N = AllocCluster()$.

9. Запись данных из буфера $BUFFER$ в кластер N.

10. Добавление $Data$ в кластер N.

11. $PTR.FIRST = N$, $PTR.LAST = N$, $PTR.NUM = 1$.

12. $PTR.LASTDATA = NEWSIZE$.

Процедура 2

Эта процедура обеспечивает добавление информации в цепочку кластеров, когда кластеры используются уже целиком для одной цепочки.

1. K' – Размер данных в кластере. Если $NUM = BLOCK\_L$, то $K' = K -$ <размер указателя на кластер>, иначе $K' = K$ (если кластер является последним кластером в блоке, имеющим длину $BLOCK\_L$, то сохраняем в нем указатель на первый кластер следующего блока).

2. Если $NEWSIZE \leq K'$, то добавление новой информации в текущий кластер $PTR.LAST$, устанавливаем $PTR.LASTDATA = NEWSIZE$, выход.

3. $NEWSIZE = NEWSIZE - K'$, $LASTSIZE =$ <Размер добавляемых данных> $- NEWSIZE$. Запись первых $LASTSIZE$ байт $Data$ в кластер $PTR.LAST$.

4. Если $PTR.NUM = BLOCK\_L$, переходим на шаг 4a, иначе на шаг 4b.

   (a) Добавляем в конец файла $BLOCK\_L$ кластеров, N устанавливаем на первый из них, далее эти кластеры будут зарезервированы для этой цепочки. $PTR.LAST = N$. $PTR.NUM = 1$. Переходим на шаг 6.

   (b) Если $PTR.NUM$ является степенью 2, а также в цепочке меньше $BLOCK\_L$ кластеров, то переходим на шаг 4(b)i, иначе: $N = PTR.LAST + 1$, $PTR.LAST = N$ и переходим на шаг 5.

      i. Если $PTR.NUM$ не равен $BLOCK\_L/2$, то переходим на шаг 4(b)ii, иначе на шаг 4(b)iv.

      ii. $i = log_2(PTR.NUM \times 2)$

      iii. Если в списке $FREE[i]$ есть элемент, то извлекаем его из списка и помещаем в N, иначе добавляем $PTR.NUM \times 2$ кластеров в конец файла и устанавливаем N на первый из добавленных кластеров, далее добавленные кластеры будут принадлежать текущей цепочке кластеров. Переходим на шаг 4c.

      iv. Добавляем в конец файла $BLOCK\_L$ кластеров, N устанавливаем на первый из них, далее эти кластеры будут зарезервированы для данной цепочки. Переходим на шаг 4c.

   (c) Читаем $PTR.NUM$ кластеров в промежуточный буфер $TEMP$, начиная с кластера $PTR.FIRST$.

   (d) $PTR.FIRST = N$

   (e) Запись буфера $TEMP$ в $PTR.NUM$ кластеров, начиная с кластера $PTR.FIRST$.

(f) $PTR.LAST = N + PTR.NUM$.

5. $PTR.NUM = PTR.NUM + 1$.

6. $PTR.LASTDATA = NEWSIZE$.

7. Добавление оставшихся $NEWSIZE$ байт $Data$ в кластер $PTR.LAST$.

2.3. Теоретическое обоснование производительности

Введем обозначения:

Пусть d – доля известных слов в текстах, R – текущее количество записей о вхождении в CLB-дереве.

K_FORMS – количество базовых форм в словаре (для текущего словаря 200 тыс.).

K – размер кластера.

Под вставкой слова в CLB-дерево автор понимает добавление информации о вхождении данного слова в текст в CLB-дерево. Информация о вхождении может включать в себе, например, идентификатор документа и позицию слова в документе. Следует учесть, что в CLB-дерево мы добавляем информацию о вхождениях для каждой базовой формы слова. Некоторые слова имеют несколько базовых форм. Если у нас было изначально N слов в тексте, то суммарное количество их базовых форм равно $N' = N \times KF$, где $KF > 1$. Для используемых словарей русского языка $KF \approx 0.25$, поэтому автор считает, что данный коэффициент не влияет на асимптотику.

Теорема 2.1. *Для вставки N записей о вхождении в CLB-дерево достаточно* $O(d \cdot N/K + (1-d) \cdot N \cdot (1 + log_H((1-d) \cdot (R+N))))$

обращений к внешней памяти.

Формулировка данного утверждения была дана в [10]. Доказательство теоремы следует из описанного в [10] алгоритма индексирования.

Первое слагаемое определяется вставкой записей известных слов, второе – записей неизвестных слов. Значение $log_H((1-d)\cdot(R+N))$ – это высота B-дерева. Вставка каждого неизвестного слова требует $O(1)$ обращений к внешней памяти для записи в цепочку кластеров и $O(log_H((1-d)\cdot(R+N)))$ для записи в B-дерево. Эта теорема показывает, что в CLB-дерево можно легко добавлять новые данные.

Далее кратко приведен алгоритм добавления данных в индекс, подробно описанный в [10]:

1. Добавление в индекс данных известных слов.
2. Добавление в индекс данных неизвестных слов.

Процесс добавления записи о вхождении слова включает в себя:

1. Получение указателя на цепочку кластеров для данной базовой формы слова.
2. Добавление данных в цепочку кластеров.
3. Сохранение обновленного указателя.

Рассмотрим более подробно пункт 2, соответствующий добавлению данных в цепочку кластеров.

Случай А. Пока суммарный размер информации о вхождениях достаточно мал, данные сохраняются в указателе.

Указатель – некоторая структура фиксированного размера, имеющая ряд полей. Часть из полей может использоваться для организации цепочек кластеров, например, номер первого кластера цепочки, номер последнего кластера цепочки, в случае Б и В. Одновременно, те же поля могут использоваться для хранения данных в них в случае А. Структура указателя более подробно описана в [10].

Случай Б. Пока суммарный размер информации достаточно мал, данные сохраняются в кластерах, разделенных на части.

Выберем число $PARTS$, являющееся степенью 2, и введем $log_2 PARTS$ списков кластеров. Список с номером i хранит в себе кластеры, разбитые на 2^{i+1} частей (номер 0 соответствует двум частям, $log_2 PARTS - 1$ соответствует $PARTS$ частям). В конце кластера выделяем область, назовем ее таблицей частей, для хранения информации о том, какие части кластера заполнены. Размер этой области $PARTS\_TABLE\_SIZE = PARTS/8$ байт, с округлением в большую сторону (чтобы хранить $PARTS$ бит). Остальная область кластера делится на равные части. При этом для заданной базовой формы слова используется только одна часть кластера, остальные части могут использоваться для других базовых форм других слов. Как только данные перестают помещаться в часть кластера, разбитого на две части (т. е. их размер больше, чем $(K - PARTS\_TABLE\_SIZE)/2$), мы переходим к случаю В.

Случай В. Данные сохраняются в цепочке из нескольких кластеров. Последний кластер хранится в оперативной памяти. Как только последний кластер полностью заполняется, в цепочку добавляется новый кластер.

Для предотвращения фрагментации вводится число $BLOCK\_L$. Если количество кластеров в цепочке C является степенью 2, и меньше чем

$BLOCK\_L$, то при добавлении нового кластера в цепочку осуществляется выделение на диске области памяти из $2 \cdot C$ последовательно расположенных кластеров и копирование текущих данных в первую половину новой области, т. е. C операций записи. Остальные кластеры считаются зарезервированными для данной цепочки кластеров. Если в дальнейшем кратно $BLOCK\_L$, то при добавлении в цепочку нового кластера сразу выделяется область из $BLOCK\_L$ кластеров, запись осуществляется в первый кластер новой области, остальные кластеры области резервируются для данной цепочки. Если C – другое число, то мы используем зарезервированные ранее для данной цепочки кластеры. Описанный метод подробно описан в [5, 10].

Рассмотрим процесс добавления данных известных слов. При добавлении в индекс известного слова вначале осуществляется получение указателя на цепочку кластеров PTR. Этот указатель хранится в таблице в оперативной памяти и таким образом его получение не требует обращений к внешней памяти. При записи известных слов следует учитывать, что для каждой базовой формы известного слова последний кластер цепочки хранится в оперативной памяти.

Без ограничения общности будем считать, что размер информации, которую мы сохраняем в индексе при добавлении одной записи о вхождении равен максимум 8 байт (4 байта – ID файла, 4 байта – позиция слова в файле). Это гораздо меньше K (типичные значения 4 Кб, 16 Кб, 32 Кб).

Рассмотрим случай А. В данном случае не требуется обращений к внешней памяти при записи данных об известных словах, т. к. данные указатели целиком хранятся в таблице в оперативной памяти.

Рассмотрим случай Б. При этом, как было указано, последние кла-

стеры цепочек хранятся в оперативной памяти. Это означает, что для случая Б текущий кластер также хранится в оперативной памяти как первый и последний кластер для цепочки кластеров по крайней мере для одной базовой формы слова. Таким образом можно считать, что данные кластеры хранятся в оперативной памяти на протяжении всего процесса добавления данных и сохраняются только после завершения этого процесса. Количество кластеров, разделенных на части не более K_FORMS, т. к. по крайней мере данные об одной базовой форме слова должны быть сохранены в каждом таком кластере. Следовательно операций записи для таких кластеров также не более K_FORMS.

Рассмотрим случай В. Как было сказано, последний кластер каждой цепочки хранится в оперативной памяти и, соответственно, сохраняется во внешней памяти в двух подслучаях: во-первых, если данный кластер заполнен и мы добавляем в цепочку новый кластер, во-вторых, если мы завершили добавление данных и сохраняем все хранящиеся в оперативной памяти кластеры во внешней памяти. Количество операций во втором подслучае не больше K_FORMS. Для первого подслучая количество операций $(N \times 8)/K$, где $N \times 8$ – максимальный объем добавляемых данных.

Также следует учесть, что, если количество кластеров в цепочке C является степенью 2, и меньше чем $BLOCK\_L$, то при добавлении нового кластера в цепочку осуществляется выделение на области памяти из $2 \cdot C$ последовательно расположенных кластеров и копирование текущих данных в первую половину новой области, т. е. C операций записи, как это описано в [10]. Пусть у нас в текущем блоке R кластеров. Количество операций перезаписи составляло $1 + 2 + 4 + 8 + ... + R/2 < R$. Таким образом, процедуры переноса данных в новую область увеличивают об-

щее число обращений к внешней памяти не более чем в два раза. Если количество кластеров в цепочке больше чем $BLOCK\_L$, то подобные переносы данных больше не производятся.

Без ограничения общности будем считать, что K_FORMS – это константа, значительно меньшая чем число N, как это всегда бывает в реальных приложениях. Соответственно, число обращений к внешней памяти при записи известных слов составляет $O(d \cdot (K\_FORMS + N/K)) = O(d \cdot (N/K))$, т. к. K_FORMS можно пренебречь.

Рассмотрим добавление неизвестных слов в индекс. При получении указателя на цепочку кластеров PTR для неизвестного слова требуется найти данное слово в B-дереве. Это требует $O(log_H((1-d) \cdot (R+N))))$ обращений к внешней памяти. Процедура сохранения обновленного указателя в B-дереве требует также $O(log_H((1-d) \cdot (R+N))))$ обращений к внешней памяти. При добавлении данных об одном вхождении неизвестного слова требуется одна операция записи во внешнюю память. Если мы учитываем переносы данных, как описано в пункте B, то они увеличивают количество операций не более чем в 2 раза. Соответственно получаем оценку $O((1-d) \cdot N \cdot (1 + log_H((1-d) \cdot (R+N))))$.

Теорема доказана.

Следует отметить, что на практике мы можем получить лучшие результаты при добавлении неизвестных слов в индекс, используя стандартные схемы организации кэша.

Теорема 2.2. *Для поиска набора слов из N-элементов в CLB-дереве достаточно $O(N \cdot (log_H((1-d) \cdot R) + occ/K))$ обращений к внешней памяти (здесь occ – количество вхождений данных слов в текстах).*

Здесь $O(N \cdot log_H((1-d) \cdot R))$ – число обращений к внешней памяти, требуемое для чтения информации из B-дерева для неизвестных слов,

которые могут быть среди искомых слов. Это слагаемое мало и на практике его можно не учитывать. Основное время занимает $O(occ/K)$ обращений к внешней памяти, для чтения информации обо всех вхождениях словоформы в текстах.

Теорема доказана.

Следует отметить, что как описано в пункте В, кластеры по сути практически располагаются последовательно. Следовательно, поиск с помощью CLB-дерева имеет такую же эффективность, как и с помощью инвертированных файлов.

Теорема 2.3. *Размер файла с кластерами для CLB-дерева составляет $O(S)$, где S – суммарный объем сохраненных в индексе данных.*

Доказательство

Данная теорема позволяет оценить максимальный размер файла с кластерами для CLB-дерева. Оценим, сколько занимают кластеры для известных слов.

В случае Б количество кластеров, разделенных на части для известных слов, не более K_FORMS.

Рассмотрим случай В. Если мы рассмотрим цепочку кластеров, то последний кластер цепочки заполнен как минимум наполовину, т. к. иначе использовались бы кластеры, разделенные на части. Более точно, размер данных в таких кластерах $DATA\_SIZE > (K - PARTS\_TABLE\_SIZE)/2)$, где $K/8 \leq PARTS\_TABLE\_SIZE \leq (K/8 + 1)$, и соответственно $DATA\_SIZE > (K - K/8 - 1)/2) = 7/16K - 1/2$. Однако разница между 2 и 7/16 не существенна с точки зрения асимптотики.

Рассмотрим процесс переноса данных из блока в блок. Если количество кластеров в цепочке C является степенью 2, и меньше чем

$BLOCK\_L$, то при добавлении нового кластера в цепочку осуществляется выделение на диске области памяти, равной $2 \times C$ и копирование текущих данных в первую половину новой области. В итоге половина кластеров в блоке автоматически становится полностью заполненной. Если равно $BLOCK\_L$, то мы выделяем для данной цепочки сразу $BLOCK\_L$ кластеров, однако, в данном случае у нас уже есть $BLOCK\_L$ заполненных кластеров, т. е. половина кластеров цепочки заполнена. Следовательно, в худшем случае размер области, используемой для хранения цепочек кластеров известных слов, не более чем в два раза больше суммарного размера добавленных данных.

Теперь оценим объем, необходимый для хранения данных неизвестных слов. Рассмотрим случай Б. В худшем случае у нас каждое неизвестное слово встретится один раз, при этом информация об его вхождении займет часть в кластере, разделенном на части. За счет выбора параметра – максимального количества частей в кластере, мы можем сделать размер данной части сколь угодно малым. например, 1 байт. Размер таблицы частей при этом равен $K/8$.

Данные переносятся из части в большую часть, если они в нее не помещаются. Следовательно, если в части есть данные, то можно считать, что часть заполнена по крайней мере наполовину. Новый кластер, разделенный на части, добавляется только, если все кластеры с такими размерами частей заполнены. Следовательно, число ни разу не задействованных частей равно количеству частей для одного кластера данного типа. Если же часть когда-то использовалась, то данные из нее были перенесены в другое место. Т. о., объем, составленный всеми свободными частями не более чем размер добавленных в индекс данных. В итоге, количество свободного места в кластерах разделенных на части не более

3/4 от их суммарного объема.

Более подробно. В кластерах разделенных на части хранятся данные для некоторых базовых форм слов. Рассмотрим одну базовую форму. Информация о ее вхождениях хранится в одной части некоторого кластера. Пусть размер данной части X. Для простоты будем считать, что если мы переносим данные из одной части в часть более большего размера, то старую часть больше не используем – это худший случай. Для данной базовой формы часть размера X заполнена на половину. Объем свободного места в данной части не более $X/2$. При этом ранее были использованы части размером $X/2$, $X/4$, ... $X/8$, ... 1, и все они теперь свободны. Т. е. объем свободного места в худшем случае равен $X/2 + X/2 + X/4 + X/8 + ... + 1 \leq X/2 + X = 1.5X$. Суммарный объем соответственно $2X$ – в 4 раза больше добавленных данных.

В случае, если ранее данные для базовой формы хранились в кластере, разделенном на части, но были перенесены в целый кластер, рассматривается аналогично.

Случай B рассматривается для неизвестных слов аналогично рассмотренному случаю B для известных слов.

Доказательство завершено.

Таким образом, мы имеем теоретические оценки затрат ресурсов на добавление данных в CLB-дерево, поиск данных в CLB-дереве и хранение CLB-дерева.

2.4. Замечания

В разделе 2.9 описана реализация B-дерева с использованием тернарных деревьев для организации хранения строк в узле B-дерева. Там же

описана пакетная процедура вставки, позволяющая одновременно добавлять в B-дерево много строк (слов). Однако, использование тернарных деревьев не является обязательным для реализации CLB-дерева. Можно использовать B- деревья независимо от того, как организовано хранение строк в узлах дерева.

Даже использование B-дерева не является обязательным. При наличии достаточного количества оперативной памяти можно использовать структуры данных, предназначенные для оперативной памяти, например, 2-3 деревья, АВЛ-деревья, хеш-таблицы и т. д. Достаточно, чтобы структура данных поддерживала операцию добавления ключа (слова) и его значения (указателя на цепочку кластеров, который описывается далее) и получения значения по ключу.

В описанном алгоритме каждой базовой форме слова соответствует одна цепочка кластеров. Можно использовать другой вариант, когда вместо одной цепочки используются две. В первой из них храниться вся информация о слове без указателей на первый кластер следующего блока, а во второй – номера первых кластеров каждого блока и указатели на кластеры для описания второй цепочки (каждый кластер второй цепочки должен хранить в себе указатель на следующий кластер). Ясно, что если информации для конкретного слова занимает менее чем $BLOCK\_L$ кластеров, то требуется только одна цепочка, так как все кластеры расположены последовательно друг за другом и следующий кластер и так известен.

2.5. Поиск

В этом разделе описано, как можно использовать CLB-дерево для поиска в текстах. Под поиском в текстах можно рассматривать различные задачи, в зависимости от того, что искать и зачем. В зависимости от задачи можно хранить в CLB-дереве различную информацию о словах.

Исходные данные: набор слов S, например, «Шла Саша по шоссе», этот набор может кодироваться в виде строки, в которой слова разделены символом «пробел».

Три задачи поиска, в порядке возрастания сложности

1. Поиск всех документов, содержащих каждое из заданных слов.
2. Точный поиск фразы, нахождение всех документов, включающих в себя подряд все указанные слова (т. е. между искомыми словами в тексте нет других слов):
   (a) с учетом порядка искомых слов;
   (b) без учета порядка искомых слов.
3. Поиск с учетом расстояния, задача похожа на задачу п. 2, только в тексте допускается наличие других слов между искомыми словами. Результатом поиска является набор фраз текста, в которых встречается как искомые слова, так и возможно другие слова. При этом в найденной фразе не должно содержаться фразы меньшего размера, удовлетворяющей условиям поиска.

CLB-дерево позволяет решать все эти задачи.

В процессе сканирования документов каждому из них сопоставляется номер. При этом, чем позже просканирован документ, тем больший

он имеет номер. Для каждого слова мы храним номер его документа и позицию в документе, например, два 32-битных целых числа. Позиция в документе может быть сохранена как порядковый номер первого символа слова или как порядковый номер самого слова. Имена документов вместе с дополнительной информацией о них хранятся в отдельном файле.

Задача 2 сводится к 3, если использовать в качестве позиции слова номер слова в документе; в таком случае разница между позициями первого и последнего слов в найденной фразе должна быть равна количеству слов в искомой фразе минус 1. Учет порядка слов легко реализовать.

Задача 1 также сводится к задаче 3 путем вывода только одного найденного результата для каждого документа. Задача 1 также легко решается, если хранить в индексе только одну запись для каждого конкретного слова в документе, а не хранить описание каждого вхождения данного слова в данный документ.

Рассмотрим задачу 3.

Запись о результате имеет следующее строение:

Поле	Обозначение	Описание
Файл	$File$	Тот документ, в котором расположены все слова из S
Начальная позиция	$Start$	Для каждого слова из S определяется позиция его в документе. Данное поле хранит в себе минимум из этих позиций.
Конечная позиция	End	Данное поле хранит в себе максимум из позиций найденных слов

Естественное условие заключается в упорядочении полученного набо-

ра записей по значению $End - Start$, т. е. по расстоянию между первым и последним из найденных слов.

Каждую цепочку кластеров мы рассматриваем как набор записей. Вследствие организации CLB-дерева и алгоритма индексирования получается: если запись B расположена после записи A, то либо номер файла B больше номера файла A, либо номера файлов совпадают, но позиция в документе у B больше чем у A, т. е. можно считать, что эти записи упорядочены по возрастанию. За счет этого упорядочения можно рассмотреть различные схемы кодирования, позволяющие уменьшить требуемый объем памяти для хранения информации о словах. Пример: если мы имеем два 32-битных числа, то их можно рассматривать вместе, как одно 64-битное число; если при этом сравнивать записи как числа, то упорядочение по возрастанию будет сохранено. Ясно, что хранить нужно 64-битное число не целиком в виде 8 байт, следует хранить только значащие разряды. Пусть дан набор чисел $A, B, C, D, ...$, вместо хранения в цепочке кластеров чисел $A, B, C, D, ...$ целиком, можно хранить числа $a = A, b = B - A, c = C - B, d = D - C, ...$ при этом, читая цепочку сначала, можно восстановить все числа $A, B, C, D, ...$ Но числа $a, b, c, d, ...$ «лучше» чем числа $A, B, C, D, ...$ так как они меньше их, поэтому у них меньше значащих разрядов. Обычно для хранения записи в таком случае требуется 3 – 4 байта.

Далее, не вдаваясь в подробности, считаем, что для каждого слова в определены следующие операции:

Процедура $FIND$**(слово** w**)**

Процедура ищет слово в CLB-дереве и возвращает логическое значение, есть ли оно в нем. Если слово есть в CLB-дереве, то процедура также определяет все цепочки кластеров, которые соответствуют этому

слову и возвращает список описателей этих цепочек. Будем считать, что описатель – это некоторая запись некоторого типа $HANDLE$. Если слова нет в CLB-дереве, то возвращается пустой список. Каждый описатель определяет текущую запись, которую мы читаем из цепочки, вначале это первая запись цепочки.

Процедура $GET(HANDLE\ h)$

Процедура возвращает номер файла и позицию слова в документе, которые хранятся в текущей записи.

Процедура $NEXT(HANDLE\ h)$

Процедура осуществляет переход к следующей записи в цепочке.

В терминах этих процедур описан алгоритм поиска:

Если какой либо список пуст, то какого-то слова вообще нет в CLB-дереве и ничего не найдено.

Дан набор слов S, всего слов в нем $|S|$, $S[i]$ – i-е слово.

Вначале для простоты рассмотрим случай, когда в каждом списке есть ровно 1 элемент.

1. Пусть $List$ – список списков, в котором $|S|$ элементов. Здесь и далее $First(x)$ – первый элемент списка x, $Last(x)$ – последний элемент списка .

2. Цикл по переменной i от 0 до $|S|-1$: $L = FIND(S[i])$. Если L пуст, то поиск неудачен, выход. Добавляем L в $List$.

3. Сортируем список по возрастанию, в зависимости от значения $GET()$ у элемента списка $List$. Здесь и далее считаем, что процедура GET определена не только для описателя типа $HANDLE$, но и для списка таких описателей, в последнем случае просто используется первый элемент списка как исходный параметр для процедуры.

4. Если номера файлов у $GET(First(List))$ и $GET(Last(List))$ совпадают, это значит, что все слова фразы находятся в одном документе, добавляем в результат поиска запись с полями: $File$ – номер файла, $Start$ – позиция слова в документе, определяемая $GET(First(List))$, End – позиция слова в документе, определяемая $GET(Last(List))$.

5. Вызов процедуры $NEXT(First(List))$, если больше нет записей в цепочке, то выход, поиск завершен, весь индекс просмотрен.

6. Возможно, список $List$ теперь неупорядочен, сортируем его заново, так как ранее список был упорядочен, то теперь сортировка линейная, нужно найти место только для одного элемента.

7. Переход на шаг 4.

Общий случай отличается только тем, что необходимо вначале отсортировать каждый полученный список L на шаге 2 по возрастанию относительно значения $GET()$, чтобы первый элемент списка имел минимальное значение, а также требуется сортировать заново список $First(List)$ на шаге 5 после вызова процедуры $NEXT$.

2.6. Кодирование позиций слов

Для поиска требуется для каждого слова записывать информацию о нем. В зависимости от типа поиска требуется сохранять различную информацию.

Тип поиска	Вид сохраняемой позиции	Точность позиции
1	Сохранение позиции слова с точностью до идентификатора документа, сохранение для каждого документа одной записи для каждой конкретной формы в документе	Файл
2a, 2b, 3	Сохранение для каждого вхождения каждой формы номера соответствующего слова в документе	Номер слова
3	Сохранение для каждого вхождения каждой формы номера первого символа соответствующего слова в документе	Номер символа

Чтобы решать одновременно оптимальным образом все три задачи поиска, предлагается сохранять позиции не в одной цепочке кластеров для каждой формы, а в двух. В первой цепочке кластеров сохраняется для каждого документа, в котором встретилась форма, идентификатор документа и количество вхождений формы в документе. Во второй цепочке сохраняются позиции форм в документе. Если во второй цепочке сохраняются номера слов, то оптимальным образом будут решаться все три задачи поиска. При этом для решения первой задачи требуется прочитать только первую цепочку кластеров, а для решения остальных двух задач – обе цепочки. При сохранении номеров слов может потребоваться при выводе результатов поиска переходить от номера слова к номеру его первого символа, например, чтобы определить место найденных слов в документе и извлечь фрагмент текста для просмотра. Для этого требу-

ется создавать дополнительные таблицы.

Для таблицы фиксируется некоторое число S. Далее для каждого файла создается таблица – массив чисел. В i-й ячейке таблицы сохраняется количество слов в части файла, которая начинается с $S \times i$-го символа и заканчивается на $S \times (i+1)$-м символе. Номер символа в данном случае и число i номеруются от 0. Для нахождения позиции слова с точностью до символа требуется пройти по таблице, суммируя ее элементы, до тех пор, пока сумма не станет больше или равна номеру слова, или таблица не закончится. В результате определится число i, такое, что данное слово будет находиться в файле в промежутке $[S \times i, S \times (i+1)]$. Для определения точной позиции следует прочитать соответствующую область из файла и прочитать слова в ней. Номер первого слова в области определяется по таблице. Соответственно читая подряд слова в области, мы знаем для каждого из них и номер слова, и его позицию с точностью до символа. Среди слов в прочитанной области находится и искомое слово. Таким образом, мы находим его позицию с точностью до символа.

Для реализации подобного поиска требуется одна операция для чтения области размера не более S из файла и, возможно, одна операция для чтения таблицы. Однако в зависимости от числа файлов и размера оперативной памяти указанные таблицы могут быть полностью помещены в оперативную память.

Чтение области из файла будет осуществляться медленно, если мы будем открывать конкретный файл и читать из него, так как может потребоваться дополнительно читать данные с диска для открытия файла вследствие существующей организации файловой системы. Для того, чтобы ускорить этот процесс, следует создать репозитарий – отдельный

файл, в котором будут храниться все проиндексированные документы. Репозитарий также полезен и по другим причинам, описанным далее в разделе 2.8.

2.7. Обработка наиболее часто встречающихся слов

Существуют такие слова, как союзы, предлоги, местоимения и некоторые другие, которые чрезвычайно часто встречаются в текстах.

Пример: на основе анализа коллекции текстовых документов общим объемом 8 914 155 363 байт (8,3 Гб) с использованием морфологического анализатора были получены следующие результаты:

1. Всего слов – 1 374 343 611.
2. Известных слов – 985 480 558 (71%).
3. Неизвестных слов – 388 863 053 (29%).
4. Всего форм слов известных слов – 1 238 201 706 (125% от количества слов).
5. Различных форм известных слов – 162 807.

128 наиболее часто встречающиеся формы:

и, в, он, не, я, что, на, быть, с, этот, оно, тот, то, а, она, это, весь, они, кака, как, к, все, всё, но, его, у, по, ты, мы, из, вы, така, свой, о, за, который, мочь, от, так, же, сказать, было, бы, человек, один, её, ё, себя, своя, когда, только, ещё, уж, такой, длить, для, сам, если, мой, их, знать, какой, узкий, иль, или, уже, кто, да, чтобы, время, до, хотеть, рука, другой, говорить, самый, нет, ни, большой, э, вот, своё, стать, деть, б, чем, эта, может, даже, год, дело, наш, глаз, есть, где, видеть, другая, хороший,

во, под, ли, быль, ю, том, раз, два, день, ж, жизнь, е, со, голова, очень, теперь, там, ничто, одна, того, должен, первый, такое, слово, мыть, будет, лицо, друг, пот, переть.

Частоты данных форм в процентах

3.025%, 2.298%, 1.767%, 1.609%, 1.432%, 1.284%, 1.274%, 1.084%, 1.083%, 0.999%, 0.988%, 0.925%, 0.747%, 0.745%, 0.731%, 0.697%, 0.680%, 0.674%, 0.645%, 0.595%, 0.565%, 0.519%, 0.518%, 0.506%, 0.498%, 0.447%, 0.433%, 0.416%, 0.406%, 0.395%, 0.390%, 0.372%, 0.364%, 0.357%, 0.357%, 0.338%, 0.309%, 0.308%, 0.301%, 0.289%, 0.289%, 0.267%, 0.246%, 0.245%, 0.239%, 0.233%, 0.228%, 0.228%, 0.226%, 0.217%, 0.216%, 0.212%, 0.211%, 0.209%, 0.203%, 0.203%, 0.199%, 0.196%, 0.193%, 0.191%, 0.185%, 0.180%, 0.176%, 0.174%, 0.172%, 0.167%, 0.165%, 0.163%, 0.163%, 0.162%, 0.160%, 0.158%, 0.156%, 0.154%, 0.152%, 0.148%, 0.141%, 0.139%, 0.139%, 0.135%, 0.135%, 0.134%, 0.133%, 0.126%, 0.125%, 0.123%, 0.122%, 0.122%, 0.121%, 0.120%, 0.119%, 0.118%, 0.115%, 0.114%, 0.113%, 0.113%, 0.112%, 0.111%, 0.110%, 0.110%, 0.107%, 0.107%, 0.107%, 0.104%, 0.103%, 0.103%, 0.103%, 0.101%, 0.100%, 0.100%, 0.099%, 0.098%, 0.098%, 0.097%, 0.097%, 0.096%, 0.094%, 0.092%, 0.091%, 0.091%, 0.091%, 0.090%, 0.089%, 0.089%, 0.089%, 0.088%, 0.088%, 0.087%

Суммарная частота данных форм 44%

Как видно, указанные 128 слов составляют более чем значительную часть текстов. Уже поиск фразы из 6 первых указанных слов потребует прохода по 1/10 части индекса.

Множество таких слов обозначим как WS, также будем называть далее такие слова как WS-слова. Многие поисковые системы имеют список WS-слов и обрабатывают их иначе, чем обычные слова. Такие слова также называются стоп-словами.

Три подхода к таким словам:

1. Не сохранять такие слова в индексе и соответственно не учитывать их при поиске.

2. Сохранять только информацию о файле для данных слов, т. е. только одну запись о слове на документ. Это хорошо работает, особенно если документы имеют большой размер, но при этом поиск таких слов не даст существенного результата, так как, например, список документов, содержащих слово «и», вряд ли интересен пользователю.

3. Новая техника, описанная далее, позволяющая легко решать задачу точного поиска для фраз, содержащих WS-слова, а также с некоторыми ограничениями задачу поиска с учетом расстояния.

В этом разделе описывается особая техника, позволяющая более эффективно искать фразы, включающие часто встречающиеся слова. Эта техника присутствует в нашей реализации CLB-дерева и может также применяться и в инвертированных файлах. Описываемый подход полностью применим к точному поиску фразы (как с учетом порядка слов, так и без учета порядка).

Следует отметить, что данный метод имеет преимущество перед подходом 1, так как, если мы не храним стоп-слова в индексе, то мы не можем их искать, и следует соответственно выбирать множество стоп-слов не очень большим. При сохранении стоп-слов мы можем выбрать в качестве стоп-слов произвольно большое количество слов (следует учитывать только, что увеличение количества стоп-слов приводит к увеличению индекса и некоторым издержкам при кодировании), при этом

поиск с данными словами будут осуществляться быстрее. При исследовании метода брались в качестве стоп-слов первые 128 наиболее часто встречающихся слов.

Несколько менее он применим к поиску с учетом расстояния. Если искомая фраза содержит в себе WS-слово, то будут найдены только такие вхождения слов фразы в текстах, в которых каждое WS-слово находится непосредственно рядом с каким либо другим словом фразы.

Например, дан текст: «Шла Саша по шоссе и сосала сушку». В данном тексте существуют два слова из WS, это «по» и «и».

При поиске с учетом расстояния будут следующие результаты:

* Фраза «Саша по шоссе и» будет найдена.

* Фраза «Саша шоссе и» также будет.

* Фраза «по и» не будет найдена.

* Фраза «Саша шоссе» будет найдена.

Однако при поиске WS-слов именно точный поиск наиболее важен, а поиск с учетом расстояния имеет скорее чисто теоретический интерес. Для реализации подобного поиска требуется следующее:

Для каждой упорядоченной пары WS-слов создается своя цепочка кластеров, содержащая все вхождения данной пары в тексте в указанном порядке. Для каждого слова не из WS, помимо позиции в его цепочки кластеров, записывается информация о том, какие именно WS-слова в тексте находятся рядом с данным словом, а также то, перед данным словом или после него они находятся. Информация о парах WS-слов может храниться в отдельном файле с кластерами.

2.7.1. Алгоритм поиска

Простейший алгоритм поиска заключается в модификации базового алгоритма поиска. Для этого для каждой формы каждого искомого слова нужно сформировать в памяти массив, содержащий все интересующие нас вхождения данной формы в тексте. При этом у стоп-слов в массиве будут храниться информация о конкретном вхождении данного стоп-слова, только если это слово в тексте находилось рядом с каким либо другим словом из искомого набора.

Если слово не является стоп-словом, то мы загружаем в оперативную память для каждой формы слова соответствующую цепочку кластеров. Если слово w является стоп-словом, то нужно сформировать несколько массивов:

1. Для каждого другого (в данном случае имеющего другой номер в искомом наборе) стоп-слова v из искомого набора слов читается цепочка кластеров для пары слов (w, v) и (v, w).

2. В процесс чтения не стоп-слов необходимо внести изменения, чтобы для каждого стоп-слова w и каждого не стоп-слова v получить массив вхождений слова w в текстах, при этом только таких, когда слово w находилось рядом со словом v. Можно считать, что цепочка кластеров для слова v представляет собой набор записей вида:

ID документа	Позиция данной формы слова в документе	Информация о находящихся рядом с данным словом v каких-либо стоп-слов, включая информацию о том, до или после данного слова они находятся. Например, для каждого стоп-слова, находящегося рядом со словом v записывается идентификатор стоп-слова, бит, определяющий до или после v это стоп-слово находится.

Таким образом, необходимая информация для слова w может быть извлечена из цепочек кластеров для слова v. Каждая цепочка кластеров для слова v порождает виртуальную цепочку кластеров для слова w, содержащую все вхождения слова w в текстах рядом с данной формой слова v.

В результате для каждого слова из исходного набора слов будет получен набор массивов. Каждый массив содержит в себе записи вида <Идентификатор документа, Позиция в документе> упорядоченные по возрастанию. В данном случае можно применить алгоритм поиска, описанный в разделе 2.5.

2.8. Репозитарий

Репозитарий – это файл, содержащий в себе все проиндексированные документы. Применение репозитария:

* Определение позиции слова с точностью до символа, если позиция сохранена с точностью до номера слова (см. раздел 2.6).

* Многие поисковые системы выводят в результатах поиска несколько строк из документа в том месте, где обнаружилось вхождение указанных для поиска данных. Репозитарий незаменим в этом случае, так как чтение этих строк из исходного файла занимает гораздо больше времени, чем чтение из репозитария, вследствие того, что требуется открыть требуемый файл, на что нужно как затратить время процессора, так и произвести дополнительные чтения данных из внешней памяти, которые требуются из-за организации файловой системы.

* Если есть репозитарий, то индекс может быть в любое время создан заново, без чтения заново исходных файлов.

Репозитарий может быть как сжатым, так и не сжатым. Первый способ предпочтительнее, так как репозитарий занимает меньше места, но в этом случае индекс создается немного дольше, чем во втором случае, так как сжатие данных требует определенного времени.

Для сжатия данных автор применяет свободно распространяемую библиотеку ZLIB или собственную реализацию алгоритма Хаффмана. В результате данные сжимаются примерно в два раза. Тексты могут быть сжаты и сильнее, но в данном случае требуется также поддержка чтения из любой части сжатого файла. Для этого исходный документ разбивается на страницы фиксированного объема, и каждая страница сжимается отдельно. Также создается дополнительная таблица, хранящая сжатые размеры получившихся страниц. Эта таблица может храниться там же, где и таблица для определения позиции слова по номеру слова (см. раздел 2.6) и при определенных условиях может храниться в оперативной памяти при работе системы.

Созданная таблица размеров сжатых страниц позволяет определять то место в репозитарии, где находится требуемая страница исходного документа, и легко извлекать ее за одну операцию чтения из внешней памяти. При этом время распаковки мало, если выбрать достаточно малым размер страницы. Обычно это 4 – 8 килобайт.

2.9. B-дерево с использованием тернарных деревьев

Мы храним имена документов в отдельном файле, и каждое имя имеет указатель в этом файле; если число документов не очень велико мы можем хранить этот файл в оперативной памяти. Каждое неизвестное слово мы храним в B-дереве, вместе с указателем на цепочку кластеров, соответствующую этому слову. 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. 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 могут быть различны, не следует ограничивать число строк в листе некоторым числом, так как за счет меньших 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. Вместе со словами в B-дереве хранятся и указатели на цепочки кластеров. В данном разделе природа указате-

лей нас не интересует. Будем предполагать только, что указатели имеют фиксированный размер. Следует отметить, что строки в B-дереве, соответствующие исходным строкам, не совпадают с исходными строками. При добавлении строки в B-дерево она дополняется терминальным нулем. Добавление терминального нуля гарантирует, что никакая строка не будет являться префиксом любой другой строки, что является необходимым условием для работоспособности алгоритмов.

Для CLB-дерева не допускается наличие одинаковых строк в B-дереве. При наличии строки в B-дереве добавление новой информации будет происходить в уже существующую цепочку кластеров и новых строк в B-дерево добавляться не будет. Однако в B-дерево можно помещать и одинаковые строки. Для этого строка при помещении дополняется не только терминальным нулем, но и уникальным счетчиком. За счет этого, хотя исходные строки и одинаковы, в B-дереве они оказываются разными. Для извлечения всех указателей одинаковых строк, извлекаются указатели всех строк в B-дереве, которые имеют префикс, идентичный искомой строке, а сразу за этим префиксом данных строк в B-дереве должен находиться терминальный ноль.

Алгоритмы вставки и поиска будут одинаковыми для обоих случаев.

Каждый лист B-дерева хранит некоторый набор строк в виде тернарного дерева и для каждой строки – указатель на цепочку кластеров. Каждый промежуточный узел хранит набор минимальных и максимальных строк 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, обычно берутся числа из промежутка от 4 до 32 Кб). Мы ор-

ганизуем хранение дерева так, что корень дерева храниться в нулевой странице.

Тернарное дерево, подробно описанное ниже, 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 далее мы обозначим $S(T)$ для тернарного дерева T, будем также считать что $T[i]$ означает i-ю строку в $S(T)$.

Далее приводим алгоритм поиска строки s длины l в B-дереве, построенного на основе набора строк D. Этот алгоритм является модификацией алгоритма, описанного в [22]. Алгоритм поиска строки s позволяет определить все строки из D (и связанную с ними информацию), префикс которых равен s. Для нас важнее определить все слова, помещенные в дерево, которые равны s: поскольку 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 $T[i-1] < s <= T[i]$ (если $i = 0$, то s меньше всех строк набора, если же i равно количеству строк, то s больше всех строк набора). Эта процедура описана подробно в пункте, посвященном тернарным деревьям.

1. $Node = 0$.

2. Повторяем шаги 2a – 2f в цикле:

   (a) Загружаем страницу внешней памяти с номером $Node$ в блок оперативной памяти.

(b) Читаем из загруженной страницы тернарное дерево T.

(c) $i = Search(T, s, l)$.

(d) Если мы находимся в листе B-дерева, то поиск прекращается, называем строку $T[i]$ текущей и переходим на шаг 3.

(e) 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.

(f) 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 2a.

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 $O(log_H|D|)$, здесь и далее $|D|$ означает количество строк в D. Для поиска слова в B-дереве требуется $O(log_H|D| + occ/H)$, 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.

Вставка в B-дерево осуществляется следующим образом:

1. Спуск по дереву для поиска листа, в котором должна находиться вставляемая строка.

2. Вставка строки 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.

Когда мы стали создавать индексы документов, то столкнулись со следующей проблемой: для вставки слова в B-дерево требуется $O(log_H|D|)$ обращений к внешней памяти в худшем случае. При таком количестве обращений скорость создания индексов нас не устраивала. Чтобы увеличить быстродействие, необходимо несколько слов добавлять в дерево за одно обращение к внешней памяти. Мы приходим к тому, чтобы добавлять в дерево одновременно много слов, при этом мы получаем, что некоторые слова попадают при добавлении в дерево в одну и ту же страницу внешней памяти.

В [22] 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 за один раз 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, каждая из которых описывает

один узел B-дерева, появившихся в результате вставки, возможно, в результате разделения узла $Node$ на части. Каждая запись в списке 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 4a 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 2a – 2c:
   (a) Переносим страницу 0 в конец файла и изменяем номер узла первой записи в $List$.
   (b) Формируем тернарное дерево T, вставляя в него все строки из $List$.
   (c) $InsertTree(0, T, List)$.

Процедура $Insert(Node, Strings, Count, List)$ является модификацией упомянутого выше алгоритма, описанного в [22]:

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 3a – 3b:

   (a) 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.

   (b) $InsertTree(Node, T, List)$.

4. Если это промежуточный узел дерева, то выполняем шаги 4a – 4c:

   (a) 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.

   (b) Для каждого набора строк:

      i. Вызываем процедуру $Insert$, которая возвращает список $List$.

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

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

iv. 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).

(c) $InsertTree(Node, T, List)$.

В следующей процедуре 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 $ExtractFirst(T, l, r)$ 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 $X = ExtractFirst(T, l, r)$.
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 ограничивались одновременной вставкой одного миллиона слов. Чтобы минимизировать число обращений к внешней памяти необходимо, чтобы словоформы неизвестных слов одной группы находились рядом в дереве. Для этого мы добавляем к словам определенной группы, в начало символ с кодом, равным номеру данной группы, чтобы отделить слова одной группы от слов другой группы (разбиение неизвестных слов на группы при создании CLB-дерева описывается в разделе 2.2.8). Вставка и поиск одного слова в B-дереве на основе тернарных деревьев имеют такую же вычислительную сложность, как при вставке слова в обычное B-дерево, но пакетная вставка большого набора слов значительно уменьшает суммарное число операций внешней памяти.

2.9.1. Тернарные деревья

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, 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 дерева длины $l+1$, 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:

   (a) Если $C < SplitChar(P)$, то:

   Если у P есть левый узел, то $P = Left(P)$ и переходим на шаг 2a, 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.

(b) Если $C > SplitChar(P)$, то:

Если у P есть правый узел, то $P = Right(P)$ и переходим на шаг 2a, 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.

(c) $i = i + 1$

(d) $C = S[i]$

(e) $P = Middle(P)$

(f) 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 $Middle$.

2.9.2. 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.
   (a) 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.
   (b) 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.
   (c) $P = Middle(P)$.
   (d) Если P – лист, то требуемое число равно номеру листа P и выход.
   (e) $i = i + 1$.
   (f) Если $i \geq l$, то требуемое число равно номеру самого левого нижележащего листа P и выход.
   (g) $C = s[i]$.

2.9.3. 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:

   (a) $X = Left(U)$.

   (b) Если у U есть родитель, то заменяем в родителе U узел U на $Right(U)$, иначе корнем дерева делаем $Right(U)$.

   (c) U полагаем равным $Right(U)$.

   (d) Пока у U есть левый узел, делаем U равным$Left(U)$.

   (e) $Left(U)$ полагаем равным X.

   (f) Выход.

4. Если у U есть левый узел, то:

   (a) Если у U есть родитель, то заменяем в родителе U узел U на $Left(U)$, иначе корнем дерева делаем $Left(U)$.

   (b) Выход.

5. Если у U есть правый узел, то:

   (a) Если у U есть родитель, то заменяем в родителе U узел U на $Right(U)$, иначе корень дерева делаем равным $Right(U)$.

   (b) Выход.

6. $P = U$, переходим на шаг 3.

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

2.9.4. Разделение

Следующий алгоритм отделяет от дерева T часть, содержащую все строки от 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. На этом этапе в отделяемом дереве в пути от корня к самому правому листу могли появиться узлы, у которых нет среднего узла, но у них тогда есть либо левый, либо правый узел. Такие узлы не несут никакой информации, и мы их удаляем, при этом нижележащее поддерево переносим к их родительскому узлу (если родительского узла нет, то все отделяемое дерево состоит из указанного ранее поддерева).

4. Если в оставшемся дереве сложилась такая ситуация, как в шаге 3, относительно его пути от корня к самому левому листу, то производятся аналогичные действия.

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 выше описанных алгоритмах можно при изменении дерева корректировать его размер. Процедура $ExtractFirst(T, l, r)$ 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 меньше 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 $ExtractFirstByLeaf$.

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: $Save(Right(P), Pointer)$.

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.

Глава 3

Программный комплекс и результаты экспериментов

3.1. Система индексирования и поиска на базе CLB-дерева

Индекс для поиска на основе CLB-дерева состоит из следующих компонентов:

1. CLB-дерево, сохранено в следующих файлах
   (a) Файл, содержащий B-дерево
   (b) Файл, содержащий таблицу указателей известных слов и вспомогательную информацию о CLB-дереве.
   (c) Файл с цепочками кластеров
   (d) Файл, содержащий список свободных кластеров
2. Файл, содержащий описания документов и вспомогательную информацию для каждого документа – список документов.
3. Файл, содержащий общие настройки индекса.

4. Репозитарий, содержащий текст всех проиндексированные документов (см. раздел 2.8). Компонент не обязателен и может отсутствовать.

5. Отдельное CLB-дерево, содержащее информацию о WS-словах. (см. раздел 2.7). Компонент не обязателен и может отсутствовать.

3.1.1. Структура списка документов

Список документов это набор записей, каждая из которых содержит:

1. Имя документа

2. Кодировка документа, формат документа (текст, HTML, RTF и т. д.), и возможно другая информация о документе.

3. Таблица для определения позиции слова с точностью до символа (см. раздел 2.6). Компонент не обязателен и может отсутствовать.

4. Таблица сжатых размеров страниц документов в репозитарии документов (см. раздел 2.8). Компонент не обязателен и может отсутствовать, если не создается репозитарий в сжатом виде.

3.1.2. Описание возможностей разработанной системы

Автором разработана библиотека для создания индексов и поиска в текстах, в которой реализована описанная структура данных и алгоритмы.

Возможности:

1. Библиотека может индексировать файлы в различных форматах, например RTF, PDF, CHM, HTML, DJVU, DOC (Microsoft Word) и кодировках, например UNICODE, UTF-8, CP-1251, ASCII, KOI-8.

2. Поддерживается обработка архивов форматов ZIP, CAB, RAR, 7Z, ARJ, TAR, и др.

3. Поиск с учетом морфологии языка.

4. Возможность сохранения полной информации о проиндексированных текстах, что позволяет осуществлять поиск и просмотр документов, даже если исходные документы недоступны.

Системные требования

1. x86 совместимый процессор.

2. 512 МБ оперативной памяти.

Рекомендуемые системные требования

1. x86 совместимый процессор.

2. 1 Гб оперативной памяти.

Требования к программному обеспечению

1. Операционная система Windows 2000/XP/2003/Vista/2008 и следующих версий.

2. Windows Script Host Версии 5.0 (Microsoft Internet Explorer 5.0 или выше).

Интерфейс пользователя поддерживает русский и английский язык.

Примечание. Модули поддержки форматов PDF, DJVU, CHM, и архивов реализованы с помощью сторонних бесплатных библиотек. Модули поддержки RTF, HTML, DOC и кодировок реализованы автором диссертации.

Библиотека реализована в виде COM сервера для операционных систем Windows. Основная часть кода написана на языке C++ и независима от конкретной операционной системы и технологии COM. Данный COM сервер можно использовать из любых языков программирования и систем, поддерживающих COM интерфейсы автоматизации, например, из скриптовых языков (JScript, VBScript, Perl, Python...).

Состав библиотеки

1. Ядро, осуществляет создание индекса и поиск.
2. Модуль поддержки морфологии.
3. Модуль распознавания кодировки. При распознавании кодировки также учитывается морфология.
4. Модуль поддержки форматов файлов. Поддержка форматов файлов и архивов реализована с помощью подключаемых дополнительных модулей, которые могут быть реализованы в виде динамических библиотек или написаны на Java. Модуль поддержки форматов файлов реализован в виде отдельного процесса для повышения надежности системы.
5. Модуль атрибутов документов, для сохранения описания документов.
6. Модуль репозитария, для сохранения текстов документов. Создается для того, чтобы при поиске можно было быстро получать фрагмент текста, содержащий найденную фразу. Тексты в репозитарии могут сохраняться с использованием различных алгоритмов сжатия.

7. Модуль COM осуществляет доступ к остальным модулям извне с помощью COM, что позволяет использовать библиотеку в различных языках программирования.

Реализованные алгоритмы не слишком требовательны к ресурсам компьютера. Для создания индекса достаточно иметь 300-400 мегабайт свободной оперативной памяти.

Автором проводились эксперименты по созданию индексов на машине с оперативной памятью размером 512 мб.

3.1.3. Создание индекса

Основная часть программы реализована в виде COM сервера. Оконный интерфейс реализуется с помощью COM сервера WSO [4], который разработан автором диссертации.

Для создания индекса необходимо создать конфигурационный файл индекса. (см. раздел 3.1.4).

После этого можно воспользоваться файлом make.js для запуска скрипта индексирования. После запуска отобразится основное окно.

На первой странице можно указать различные параметры индексирования:

* Максимальный размер – можно указать максимальный размер файлов, который следует проиндексировать, при достижении данного размера индексирование останавливается.

* Буфер – размер буфера для индексирования, рекомендуется не менее 80 мегабайт.

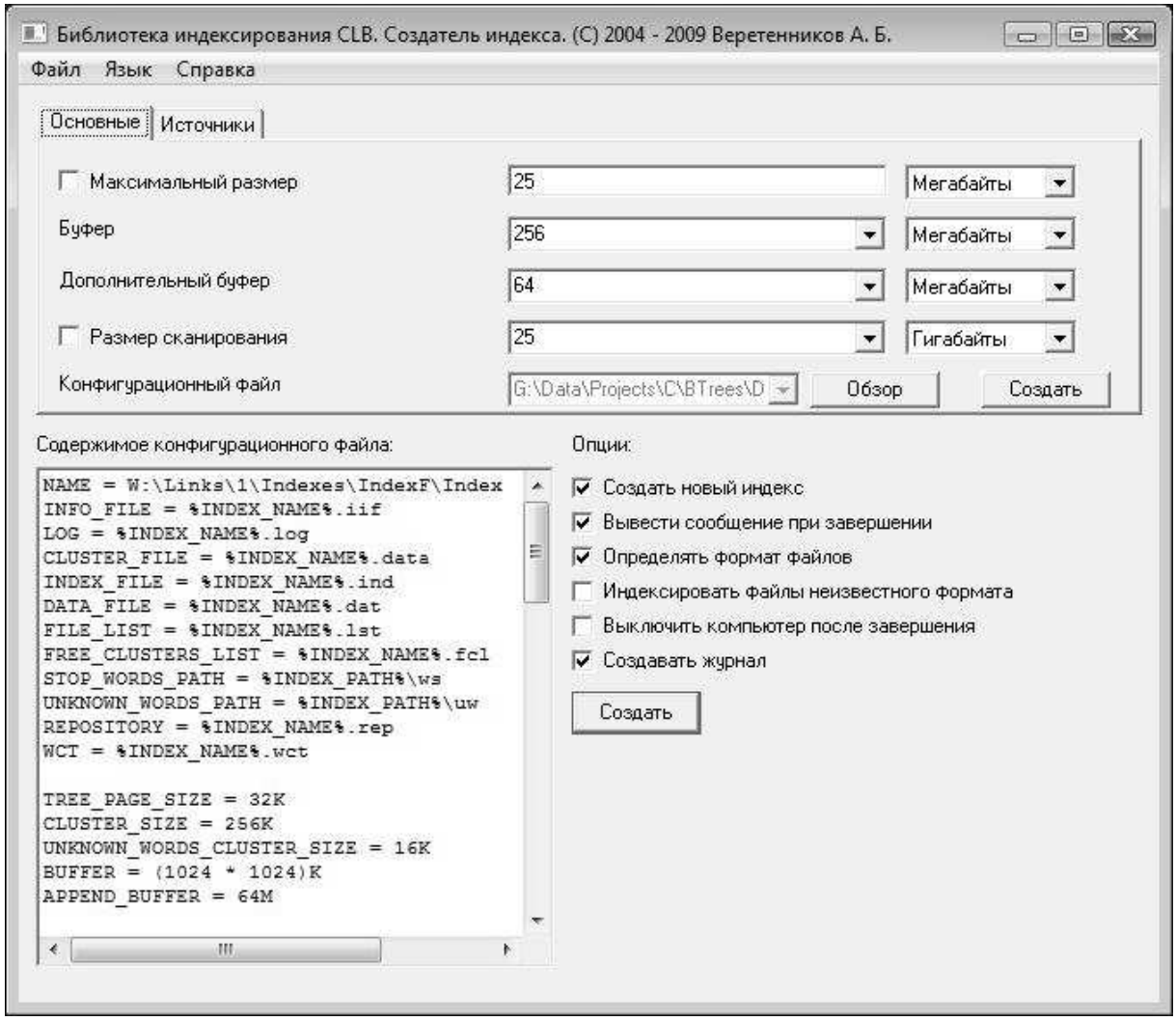

Рис. 3.1: Создание индекса. Основное окно

* Дополнительный буфер – размер вспомогательного буфера, рекомендуется 16-32 мегабайт.

* Размер сканирования – размер итерации. В процессе индексирования указанное число байт считывается во временные файлы, затем из временных файлов информация переносится в индекс, затем процесс повторяется. Чем больше данный параметр – тем больше быстродействие. Рекомендуется не включать данный переключатель, в этом случае будет осуществлена только одна итерация. Размер вре-

менных файлов равен 70-90% от исходных файлов.

* Конфигурационный файл – требуется указать конфигурационный файл.

* Создать новый индекс – создается новый индекс; если существуют какие-то файлы, то они удаляются. Если данный переключатель отключить и если индекс уже существует, то новые данные будут добавлены в уже существующий индекс.

* Вывести сообщение при завершении – после завершения работы будет выведено сообщение об этом.

* Определять формат файлов – включает режим распознавания кодировки файлов, с использованием анализа морфологии.

* Индексировать файлы неизвестного формата – разрешает индексирование файлов, кодировка которых не распознана. Учитывается только если включена опция «Определять формат файлов».

* Выключить компьютер после завершения – завершить работу компьютера после завершения индексирования.

* Создавать журнал – включить создание журнала. В данный журнал сохраняется расширенная диагностическая информация.

На второй странице «Источники» можно указать каталоги и файлы, которые следует проиндексировать.

Можно добавлять в список как каталоги, так и отдельные файлы, например архивы.

3.1.4. Конфигурационный файл индекса

Индекс создается на основании конфигурационного файла индекса.

Основные настройки

1. NAME – базовое имя индекса.
2. LOG – журнал индекса – в него записывается различная диагностическая информация.
3. INFO_FILE, CLUSTER_FILE, INDEX_FILE, DATA_FILE, FILE_LIST – месторасположение файлов индекса.
4. STOP_WORDS_PATH – путь для файлов индекса стоп-слов.
5. UNKNOWN_WORDS_PATH – путь для файлов индекса неизвестных слов.
6. REPOSITORY – путь для файла репозитария, в этом файле сохраняется текст проиндексированных файлов, репозитарий используется для при поиске для извлечения строки текста, включающей в себя искомые слова или фразы. Репозитарий может создаваться с использованием сжатия, см. параметр REPOSITORY_COMPRESSION.
7. TREE_PAGE_SIZE – размер страницы B-дерева, обычно 4K.
8. CLUSTER_SIZE – размер кластера, обычно 16K.
9. UNKNOWN_WORDS_CLUSTER_SIZE – размер кластера для индекса неизвестных слов, обычно 512.
10. BUFFER – размер буфера, рекомендуется не менее 80 мегабайт. При создании индекса размер буфера определяется как максимум значения данного параметра и параметра «Буфер», выбранного пользователем в окне создания индекса.

11. APPEND_BUFFER – размер дополнительного буфера, рекомендуется 16-32 мегабайт. При создании индекса размер дополнительного буфера определяется как максимум значения данного параметра и параметра «Дополнительный буфер», выбранного пользователем в окне создания индекса.

12. STORE_STOP_WORDS – параметр контролирует индексирование стоп-слов, если его значение FALSE – стоп-слова не индексируются.

13. STORE_STOP_WORDS_ONE_PER_FILE – параметр контролирует индексирование стоп-слов, если его значение TRUE – для стоп-слов сохраняется информация только по первом его вхождении в документе. Если этот параметр равен TRUE – то параметр STORE_STOP_WORDS не учитывается.

14. MAKE_STOP_WORDS_INDEXES – параметр разрешает создание дополнительных индексов для стоп-слов.

15. DATA_DETAILS – определяет тип сохранения информации в индексе, возможные значения:

    * FILE – сохранение в индексе для конкретного слова только информации о том, в какие файлы оно входит.
    * POSITION – сохранение в индексе для конкретного слова информации о том, в какие файлы оно входит и позиции этого слова в файлах (в байтах).
    * WORD – сохранение в индексе для конкретного слова. информации о том, в какие файлы оно входит и позиции этого слова в файлах (в виде порядкового номера слова).

    Рекомендуемый параметр WORD.

16. TEMP_PATH – путь для временных файлов, на соответствующем диске должно быть достаточное количество свободного места.

17. REPOSITORY_COMPRESSION –

    Параметр, определяющий тип сжатия репозитария: 0 – без сжатия 1 – 9 сжатие ZIP, соответственно указанное число является уровнем сжатия, 1 – наименьшее сжатие, 9 – наилучшее сжатие. 128 сжатие по алгоритму Хаффмана, данный алгоритм сжимает тексты не так хорошо как ZIP, но зато гораздо быстрее.

18. UNKNOWN_WORDS_PARTS – параметр, для индексирования неизвестных слов, обычно 32, изменять не рекомендуется.

19. SEQUENCE_REPOSITORY – файл с индексом последовательностей.

20. SIMULAR_INDEX – файл индекса похожих документов [7].

21. SEQUENCE_BUFFER_SIZE – размер буфера для поиска похожих документов, рекомендуется не менее 512 мегабайт.

22. USE_TRANSFORMATION_SERVICE – запуск дополнительных модулей в отдельном процессе. Рекомендуется указывать TRUE.

23. FORMATS – настройка дополнительных модулей поддержки форматов файлов. Названия используемых модулей записываются через запятую.

24. ARCHIVE_FORMATS – настройка дополнительных модулей поддержки форматов архивов. Названия модулей записываются через запятую.

25. SKIPPED_FILES_LIST – файл для записи списка пропущенных при создании индекса файлов. Если настройка не указана, файл со списком пропущенных файлов не создается. Это обычный текстовый файл, по которому можно быстро просмотреть, какие файлы были пропущены при создании индекса.

26. ENABLE_INDEX_INIT – используется для тестирования. Также как ENABLE_INDEXING запрещает создание индекса.

27. UNKNOWN_WORDS_BUFFER_SIZE – размер вспомогательного буфера, изменять в меньшую сторону не рекомендуется. 64M

28. DISABLE_CLUSTER_RESERVING – по умолчанию FALSE. Применяется для тестирования.

29. GROUP_BUFFER_SIZE – размер буфера группы слов, по умолчанию 1M. Изменять не рекомендуется.

30. SINGLE_FILE_UNKNOWN_WORDS – по умолчанию FALSE. Если имеет значение TRUE, то для неизвестных слов создается один индекс. Иначе, создается один индекс для каждой группы неизвестных слов (их всего UNKNOWN_WORDS_PARTS).

31. FILELIST_MAX_MEMORY_SIZE – размер кэша для описаний документов. Используется при открытии индекса для поиска. Значение по умолчанию 0 – нет ограничений на размер кэша.

3.1.5. Поиск

Для поиска можно использовать скрипт find.js.

Основное окно поиска можно увидеть на рисунке 3.2.

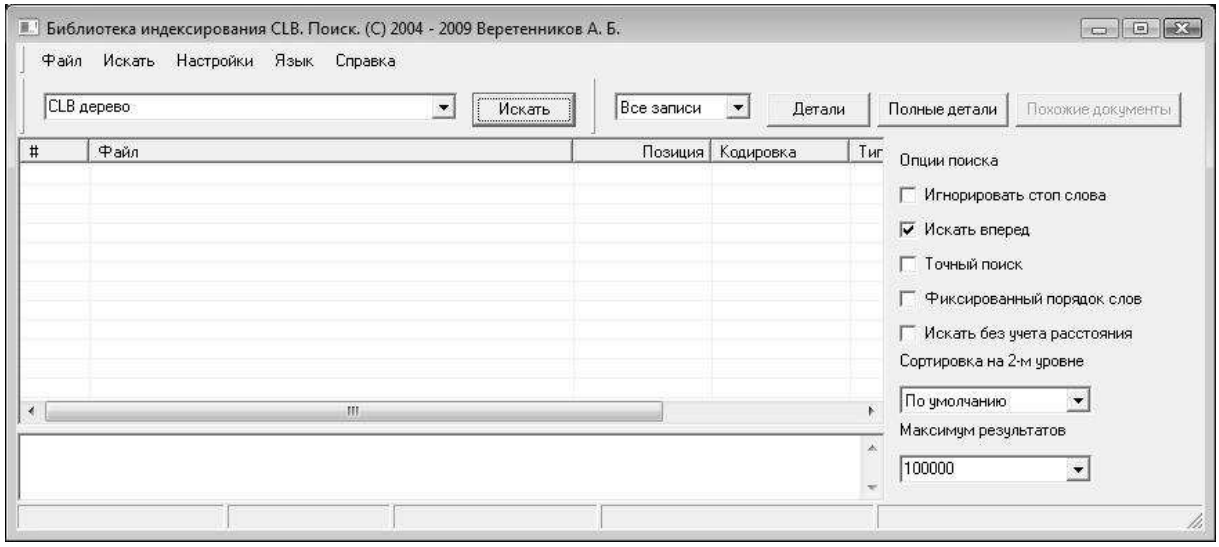

Рис. 3.2: Поиск

Для поиска следует ввести искомые слова или фразу и нажать на кнопку «Искать».

Опции поиска:

1. Игнорировать стоп-слова.
2. Искать вперед – записи отсортированы в том порядке, в котором были проиндексированы соответствующие файлы, если переключатель не включен – в обратном порядке.
3. Точный поиск – точный поиск фразы.
4. Фиксированный порядок слов.
5. Искать без учета расстояния.
6. Переключатель «Все записи / Одна запись на файл – определяет способ вывода записей: если выбран режим «Все записи» выводятся все найденные записи, иначе – для каждого документа выводятся только первые найденные записи. Чтобы посмотреть остальные

вхождения фразы в данном файле, нужно выбрать файл в таблице и нажать кнопку «Детали» или «Полные детали»

7. Сортировка на втором уровне – два значения: по умолчанию / позиция в документе. Если выбран режим «Позиция в документе», найденные записи сортируются по номеру их первого слова.

8. Максимум результатов – максимальное количество записей, которое возвращает запрос.

Результаты поиска отображаются в виде таблицы. Каждая строка таблицы содержит в себе имя файла, который содержит требуемые слова, информацию о местонахождении данных слов в файле и информацию об этом файле, в частности кодировку файла, формат файла, идентификатор файла.

Внизу экрана располагается поле для отображения фрагмента файла. При выборе строки таблицы в указанном поле отображается фрагмент файла, содержащий искомые слова. В данном фрагменте найденная фраза выделяется цветом.

3.1.6. Журнал индекса

В журнал индекса сохраняется различная статистическая информация, в частности о времени создания индекса, количестве и объеме обработанных файлов, количестве файлов определенного формата, доля известных слов, количество обработанных базовых форм слов, время работы различных компонентов программного комплекса и т. д.

3.1.7. Настройки библиотеки

Настройки библиотеки хранятся в файле index.cfg. Данный файл располагается в основном каталоге библиотеки рядом с файлом index.dll. Возможные настройки:

1. JAVA_HOME – путь к JAVA машине (jvm.dll).
2. JAVA_CLASSPATH – дополнительные пути для поиска классов JAVA. Пути перечисляются через «;».
3. TRANSFORMATION_SERVICE_TIMEOUT – таймаут для сервиса поддержки дополнительный форматов документов. Задается в миллисекундах. Если в течении данного промежутка времени модуль дополнительных форматов не отвечает на запросы, он автоматически перезапускается.
4. DEBUG – включение режима отладки – запись дополнительной отладочной информации в log файл.
5. LOG_PATH – путь для сохранения отладочной информации.
6. MAX_LOG_FILE_SIZE – максимальный размер файла журнала диагностической информации.
7. MAX_LOG_TOTAL_SIZE – максимальный размер всех файлов журнала диагностической информации.
8. JAVA_MIN_MEMORY – минимальное количество памяти, используемое виртуальной машиной JAVA.
9. JAVA_MAX_MEMORY – максимальное количество памяти, используемое виртуальной машиной JAVA.

10. JAVA_STACK_SIZE – размер стека для виртуальной машины JAVA.

11. MORPHO – настройка морфологии, через запятую перечисляются файлы словарей.

3.1.8. Модуль поддержки форматов

Загрузка дополнительных модулей может осуществляться как в основном процессе, так и в отдельном процессе – модуле поддержки форматов. Использование модуля поддержки форматов или сервиса дополнительных модулей контролируется в конфигурационном файле индекса параметром USE_TRANSFORMATION_SERVICE.

На схеме 3.3 отображена работа модуля поддержки форматов.

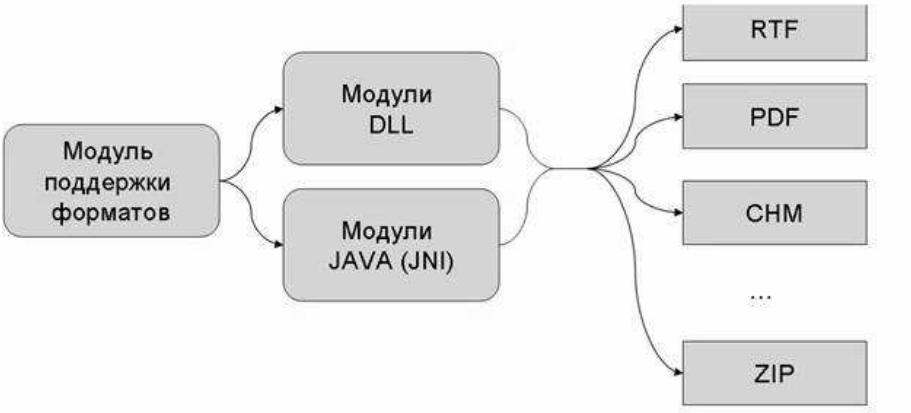

Рис. 3.3: Работа модуля поддержки форматов

Модуль поддержки форматов реализует уровень изоляции дополнительных модулей от ядра библиотеки.

При сбое в одном из дополнительных модулей модуль дополнительных форматов перезапускается автоматически. Таким образом, процесс

создания индекса не прерывается.

Модуль дополнительных форматов перезапускается автоматически, если он не отвечает на запросы в течение заданного промежутка времени, определяемого в конфигурационном файле библиотеки параметром TRANSFORMATION_SERVICE_TIMEOUT.

3.1.9. Внутреннее устройство библиотеки

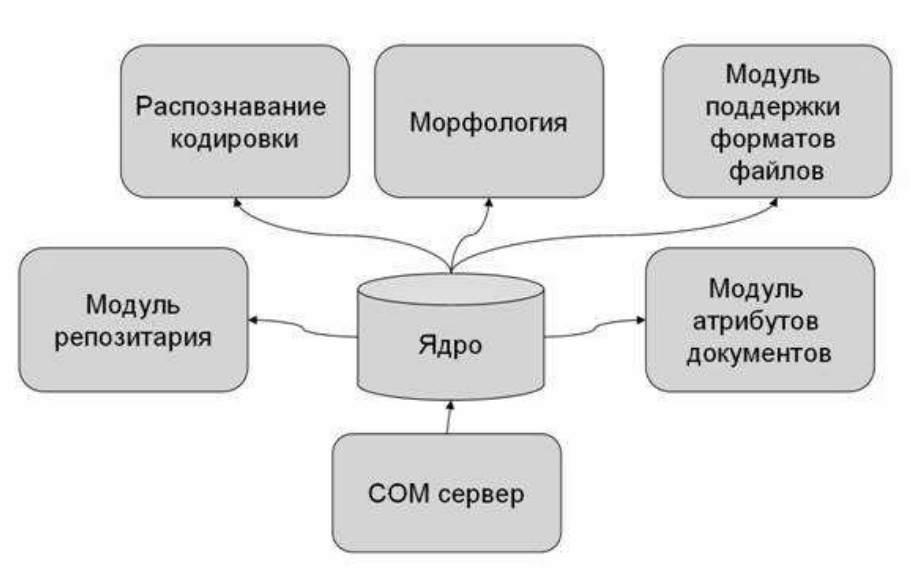

Рис. 3.4: Внутреннее устройство библиотеки

Библиотека состоит из нескольких модулей:

* Ядро – осуществляет создание индекса и поиск.

* Модуль поддержки морфологии.

* Модуль распознавания кодировки.

* Модуль поддержки форматов файлов.

* Модуль атрибутов документов сохраняет в файле метаданные документов: имя файла, формат и т. д.

* Модуль репозитария сохраняет текст документов и позволяет извлекать фрагменты текста при поиске, для отображения результатов поиска.

* Модуль COM позволяет обращаться к библиотеке через COM.

3.2. Результаты экспериментов

3.2.1. Исследование производительности базовой структуры

В [12] были описаны эксперименты измерения временных характеристик построения CLB-дерева в зависимости от объема исходных данных в диапазоне от нуля до 2 Гб. Производилась обработка художественной литературы на русском языке. Было показано, что при добавлении данных свыше 1 Гб зависимость времени от размера исходных данных становится линейной.

3.2.2. Сравнение с инвертированными файлами

Описание тестовых данных

Было произведено сравнение по скорости создания индекса. Обрабатывались текстовые документы, общий объем 35,2 Гб, 191 074 файла. Все файлы были в кодировке Windows-1251 (CP1251). Язык документов – русский. Все файлы представляли собой обычный текст.

Описание конфигурации оборудования

Эксперименты проводились на следующей конфигурации:

Процессор: Intel Core 2 Duo E6700, 2.66 GHz, кэш: L1 Data – 2 x 32 кб, L1 inst. 2 x 32 кб, L2 - 4096 кб.

Оперативная память: 4 Гб, DDR2 800.

Жесткий диск: Seagate Barracuda 7200.10, 7200 RPM, кэш 16 мб., объем 750 Гб. FSB 1066 MHz.

Создание индекса

Создание инвертированного файла: время 9 часов, размер 40 Гб.

Создание CLB индекса: время 3 часа, 32 мин., размер 24 Гб.

Добавление в индекс одного файла среднего размера

Был проведен замер скорости добавления в индекс одного документа, размер документа 1,2 мб.

Время добавления одного документа 1,2 мб. для CLB индекса: 9 мин.

Время добавления одного документа 1,2 мб. в инвертированный файл: 57 мин.

Добавление в индекс одного файла малого размера

Был проведен замер скорости добавления в индекс одного документа, размер документа 534 байта.

Время добавления одного документа размером 534 байта для CLB индекса: 22 с.

Время добавления одного документа размером 534 байта в инвертированный файл: 57 мин (т. е. такое же, как при размере файла 1,2 мб.).

Поиск

Время поиска в инвертированном файле и CLB-индексе практически совпадают.

Выводы

Проведенные эксперименты показывают высокую эффективность CLB индекса при добавлении в него данных небольшого размера.

3.2.3. Сравнение с существующими разработками

Описание тестовых данных

Было произведено сравнение по скорости создания индекса и размеру индекса. Использовались те же документы, которые указаны в разделе 3.2.2.

Описание конфигурации оборудования

Эксперименты проводились на следующей конфигурации:

Процессор: Intel Pentium 4, 3.0 GHz, кэш: L1 Data - 16 кб, L1 trace - 12 Kuops, L2 - 2048 кб.

Оперативная память: 4 Гб, DDR2 533.

Жесткий диск: Seagate Barracuda 7200.8, 7200 RPM, кэш 8 мб., объем 200 Гб.

FSB: 800 MHz.

SearchInform Desktop

http://www.searchinform.com

Размер индекса 16,15 Гб.

Время создания 9 часов.

Архивариус 3000

http://www.likasoft.com/

Размер индекса 24,83 Гб.

Время создания 6 часов 46 мин.

Google Desktop

www.google.com

Размер индекса 5 Гб

Время создания 31 час 25 минуты

Создание CLB индекса

Размер индекса 26,2 Гб.

Время создания 5 часов 49 мин.

Выводы

Эксперименты показывают высокую скорость создания CLB индекса.

3.2.4. Сравнение эффективности CLB-дерева в 32-битных и 64-битных архитектурах

В настоящее время активно развивается разработка приложений в 64-битных архитектурах. В частности, процессоры Intel и AMD уже поддерживают архитектуру EM64T. Описываемый программный комплекс реализован как под 32-битную архитектуру, так и под 64-битную архитектуру [9]. В данном разделе рассмотрены вопросы связанные с вопросом перехода с 32- битной архитектуры на 64-битную и приведены результаты экспериментов сравнения производительности в 32-битной и 64-битной среде.

Основные преимущества 64-битной архитектуры.

Основным преимуществом данной архитектуры очевидно является объем адресуемой оперативной памяти. В 32-битной архитектуре максимальный объем адресного пространства составляет 4 Гб. При этом обычно операционная система резервирует часть этого пространства под собственные нужды и приложению остается например 2 Гб. На сегодняшний момент подобный объем оперативной памяти часто недостаточен для решения ряда задач.

Следует также отметить, что при ограничении в 2 Гб мы обычно не можем задействовать все 2 Гб, т. к. при выделении блоков памяти память может стать фрагментированной. Как правило в случае невозможности выделить блок памяти возбуждается исключение. Корректная обработка данной ситуации и восстановление работы программы могут быть до-

статочно сложны. Некоторые системы для решения данной проблемы не используют больше 1,5 Гб оперативной памяти, оставляя таким образом некоторый запас для надежности.

Эксперименты проводились на следующей конфигурации:

Процессор: Intel Core 2 Duo E6700, 2.66 GHz, кэш: L1 Data – 2 x 32 кб, L1 inst. 2 x 32 кб, L2 - 4096 кб.

Оперативная память: 8 Гб, DDR2 800.

Жесткий диск: Seagate Barracuda 7200.10, 7200 RPM, кэш 16 мб., объем 750 Гб. FSB 1066 MHz.

Операционная система: Microsoft Windows 2008 Enterprise x64 Edition with Service Pack 1.

Следует отметить, что данная операционная система и процессор позволяют выполнять как 32-, так и 64-битные приложения. Архитектура системы

Исследуемые алгоритмы оформлены в виде библиотеки. Библиотека может индексировать файлы в различных форматах, например RTF, PDF, CHM, HTML, DJVU и кодировках, например UNICODE, UTF-8, CP-1251, ASCII, KOI-8. Поддерживается обработка архивов форматов ZIP, CAB, RAR, 7Z, ARJ, TAR, и др. Библиотека реализована в виде COM-сервера для операционных систем Windows.

Состав библиотеки

1. Ядро, осуществляет создание индекса и поиск.
2. Модуль поддержки морфологии.
3. Модуль распознавания кодировки. При распознавании кодировки также учитывается морфология.

4. Модуль поддержки форматов файлов. Поддержка форматов файлов и архивов реализована с помощью подключаемых дополнительных модулей, которые могут быть реализованы в виде динамических библиотек или написаны на Java. Модуль поддержки форматов файлов реализован в виде отдельного процесса для повышения надежности системы.

5. Модуль атрибутов документов, для сохранения описания документов.

6. Модуль репозитария, для сохранения текстов документов. Создается для того, чтобы при поиске можно было быстро получать фрагмент текста, содержащий найденную фразу. Тексты в репозитарии могут сохраняться с использованием различных алгоритмов сжатия.

7. Модуль COM осуществляет доступ к остальным модулям извне с помощью COM, что позволяет использовать библиотеку в различных языках программирования.

8. Модуль создания индекса похожих документов.

Библиотека написана на языке C++. Для компиляции использовался компилятор из Microsoft Visual Studio 2008. COM-сервер реализован с помощью библиотеки ATL. Также используется библиотека STL.

Модуль 4 организован в виде отдельного исполняемого файла, который вызывает отдельные подмодули для каждого формата файла. Модуль 2 выделен в отдельную динамическую библиотеку. Остальные модули скомпилированы вместе в одну динамическую библиотеку.

На 64-битную архитектуру были переведены все модули, кроме модуля поддержки форматов файлов и его подмодулей. Модуль поддержки

форматов реализован в виде отдельного процесса, и таким образом его можно использовать и в 32-, и в 64-битном приложении.

Далее указано, какие ресурсы потребляет каждый модуль в процессе работы. Выделено два типа ресурсов: 1 - вычислительные ресурсы (процессор и обращения к оперативной памяти), 2 - внешняя память (чтение и запись на жесткий диск).

Название модуля	Потребление ресурсов процессора и оперативной памяти	Использование жесткого диска
Ядро	Среднее	Большое
Модуль распознавания кодировки	Незначительно	Отсутствует
Модуль поддержки морфологии	Большое	Отсутствует
Модуль поддержки форматов файлов	Большое	Среднее
Модуль атрибутов документов	Незначительно	Незначительно
Модуль репозитария	В зависимости от используемых алгоритмов сжатия	Большое
Создание индекса похожих документов	Большое	Среднее

Рассмотренные вопросы перехода на 64-битную архитектуру:

1. Библиотека реализована в виде COM-сервера, рассмотрен вопрос перевода COM- сервера с 32-битной на 64-битную архитектуру.

2. Вопросы связанные с размером указателя и числовых типов данных, которые могли изменится с переходом на 64-битную архитектуру.

3. Модуль 4 остался без изменений. Рассмотрены вопросы корректной передачи данных между основным процессом и модулем 4.

Перевод COM-сервера на 64-битную архитектуру.

Перевод COM-сервера с 32-битной на 64-битную архитектуру не вызвал никаких проблем. Для этого потребовалось только изменить настройки компилятора. Размеры типов данных

Размер указателя естественно меняется с 32 бита на 64 бита. Теоретически могут возникать проблемы, если используется преобразование указателя в числовой тип и обратно. Таких проблем обнаружено не было.

Размеры числовых типов могут варьироваться в зависимости от компилятора. В используемом компиляторе числовые типы short, int, long, long long и их беззнаковые аналоги имеют одинаковый размер как на 32-битной архитектуре, так и на 64-битной архитектуре (соответственно 16, 32, 32, 64 бита).

Эксперименты

Были проведены эксперименты создания индекса.

Параметры создания индекса, которые влияют на производительность:

1. H – размер страницы B-дерева.

2. K – размер кластера для слов, входящих в словарь морфологического анализатора.

3. KU – размер кластера для слов, не входящих в словарь морфологического анализатора.

Далее кратко приведен алгоритм создания индекса.

1. Чтение файлов и запись данных во временные файлы (предварительная обработка). Этап включает в себя морфологический анализ, анализ кодировки файлов и запись данных в репозитарий.

2. Запись данных из временных файлов в индекс.

   (a) Добавление в индекс слов, входящих в словарь морфологического анализатора.

   (b) Добавление в индекс слов, не входящих в словарь морфологического анализатора.

   (c) Добавление в индекс информации о часто используемых словах.

3. Создание индекса похожих документов.

1. Индексирование набора текстовых документов.

Общий размер документов: 86 Гб, известных слов 90%. Все файлы представляли собой обычный текст.

H = 32 Кб, K = 256 Кб, KU = 16 Кб, размер кэша 1 Гб, размер буфера для индекса похожих документов 1 Гб.

Получившийся объем индекса 56 Гб.

Репозитарий, индекс похожих документов и индекс часто используемых слов не создавались.

	32	64
Общее время создания индекса	4 час 16 мин	4 час 11 мин

Распознавание кодировки	3 мин 27 сек	3 мин 8 сек
Морфологический анализ	49 мин 36 сек	51 мин 45 сек
Чтение файлов	2 час 50 мин	2 час 44 мин
Время добавления в индекс слов, входящих в словарь морфологического анализатора	54 мин	52 мин
Время добавления в индекс слов, не входящих в словарь морфологического анализатора	31 мин	34 мин
Суммарное время добавления данных в индекс	1 час 25 мин	1 час 26 мин

2. Индексирование набора текстовых документов.

Набор файлов и параметры, аналогичный предыдущему.

На этот раз создавался репозитарий, индекс похожих документов и индекс часто используемых слов. При создании репозитария использовался алгоритм Хаффмана для сжатия данных.

Суммарный объем индекса 139 Гб, из них 57 Гб - репозитарий.

	32	64
Общее время создания индекса	7 час 43 мин	7 час 41 мин
Распознавание кодировки	3 мин 20 сек	3 мин 42 сек
Морфологический анализ	50 мин 36 сек	55 мин 07 сек
Чтение файлов	3 час 58 мин	3 час 44 мин
Создание индекса похожих документов	1 час 17 мин	1 час 33 мин

Время добавления в индекс слов, входящих в словарь морфологического анализатора	1 час 17 мин	1 час 12 мин
Время добавления в индекс слов, не входящих в словарь морфологического анализатора	44 мин	48 мин
Время записи данных об часто используемых словах	26 мин	23 мин
Суммарное время добавления данных в индекс	2 часа 27 мин	2 час 23 мин

3. Индексирование набора текстовых документов.

Как в предыдущем случае, только размер кеша - 2 Гб. Случай 32-битной архитектуры не исследовался.

	32	64
Общее время создания индекса	-	7 час 23 мин
Распознавание кодировки	-	3 мин 26 сек
Морфологический анализ	-	52 мин 50 сек
Чтение файлов	-	3 час 45 мин
Создание индекса похожих документов	-	1 час 27 мин
Время добавления в индекс слов, входящих в словарь морфологического анализатора	-	1 час 10 мин

Время добавления в индекс слов, не входящих в словарь морфологического анализатора	-	47 мин
Время записи данных об часто используемых словах	-	23 мин
Суммарное время добавления данных в индекс	-	2 часа 20 мин

4. Индексирование файлов в архиве.

Индексировались файлы в архивах формата rar, без сжатия. Суммарный размер архивов 110 Гб. Файлы были в различных форматах, в основном текстовые файлы, HTML и RTF. Создавался репозитарий, индекс похожих документов и индекс часто используемых слов. При создании репозитария использовался алгоритм Хаффмана для сжатия данных.

Суммарный объем индекса 142 Гб, из них 58 Гб - репозитарий.

	32	64
Общее время создания индекса	9 час 23 мин	9 час 03 мин
Распознавание кодировки	4 мин 13 сек	4 мин 06 сек
Морфологический анализ	53 мин 11 сек	55 мин 43 сек
Чтение файлов	5 час 39 мин	5 час 06 мин
Создание индекса похожих документов	1 час 20 мин	1 час 28 мин
Время добавления в индекс слов, входящих в словарь морфологического анализатора	1 час 10 мин	1 час 10 мин

Время добавления в индекс слов, не входящих в словарь морфологического анализатора	49 мин	54 мин
Время записи данных об часто используемых словах	24 мин	22 мин
Суммарное время добавления данных в индекс	2 час 23 мин	2 час 26 мин

Проведенные эксперименты показали, что время работы системы в 32-х битном случае и 64-х битном случае примерно одинаково.

Следует также обратить внимание на эксперименты 3 и 2, которые показывают, что в текущей реализации алгоритмов увеличение размера основного буфера в 2 раза не дает существенного увеличения производительности.

3.2.5. Эксперименты поиска

Был проведен ряд экспериментов поиска различных слов и словосочетаний. При указанном объеме максимальное время поиска равнялось примерно одной секунде как при поиске фраз без стоп-слов, так и при поиске фраз со стоп-словами.

Глава 4

Вспомогательные компоненты

4.1. Оптимизация выделения динамической памяти

Реализация CLB–дерева выполнена на языке программирования C++. Для ускорения работы системы был применен новый метод оптимизации выделения динамической памяти.

Управление выделением динамической памяти является одной из ключевых задач программирования. При реализации многих структур данных, как-то: очереди, списки, различные древовидные структуры, активно используется динамическую память. Вместе с тем хорошо известно, что функции языка C $malloc/free$ (и соответственно операции C++ $new/delete$) в силу универсального характера их реализации очень медленно работают при выделении/освобождении большого количества малых по размеру блоков памяти. Эта проблема присутствует и во многих других языках программирования и средах. Необходимо отметить, что упомянутые структуры требуют многократного выделения небольших блоков памяти. Это наводит на мысль о необходимости разработки функций управления памяти, которые эффективно работают с блоками небольшого размера.

А. Александреску в гл. 4 [2] дал вариант решения подобной задачи, но описанный им алгоритм иногда может работать неэффективно. В данной работе описан другой алгоритм, который решает задачу более эффективно и обладает гарантированной производительностью.

Предлагаемый алгоритм при выделении блока, меньшего некоторого фиксированного числа (например 1 Кб), всегда выделяет и освобождает память за небольшое константное число операций, то есть имеет для таких блоков вычислительную сложность $O(1)$. Для блоков большего размера он просто передает управление обычным функциям.

Предполагается, что операционная система предоставляет нам функции управления памяти, которые работают для любых размеров блоков, но возможно медленно. С их помощью мы выделяем блок большого размера. Далее наша функция для выделения блока малого размера использует часть выделенного большого блока.

Реализация алгоритма автора статьи была выполнена на языке C++, при этом были учтены следующие аспекты:

1) Какие функции операционной системы использовать для выделения больших блоков: есть стандартные функции языка C $malloc$ и $free$, но операционная система может предлагать свои функции.

2) Выделение памяти может происходить как в однопоточных программах, так и в многопоточных.

Основная идея быстрого выделения памяти заключается, как и в [2], в реализации выделения блоков памяти фиксированного размера, но предлагаемый алгоритм абсолютно иной.

Пусть у нас есть функции операционной системы для выделения блоков некоторого размера $BlockSize$, которые мы называем страницами, организуем хранение маленьких блоков в этих страницах. Страница опи-

сывается структурой $LIST$. Все страницы хранятся в виде циклического списка. Свободные блоки хранятся в виде общего для всех страниц циклического списка, при этом для каждой страницы определены указатели $FreeFirst$ и $FreeLast$, которые указывают на ту часть общего циклического списка маленьких блоков, которая относиться к конкретной странице, свободные блоки для конкретной страницы располагаются в списке последовательно. Также для страницы определен счетчик $Count$ – число выделенных блоков. Когда мы говорим, что работаем со списком страницы, это означает, что мы работаем с частью общего списка используя $FreeFirst$ и $FreeLast$.

Пусть мы организуем выделение памяти для блоков имеющих некоторый размер S. Каждый блок будет описываться структурой

```
struct ITEM {
  union { char Data[S];
    struct {ITEM *Next; ITEM *Prev; }; };
  LIST *Block; };
```

Когда блок заполнен, данные его хранятся в поле $Data$, когда он свободен, поля $Next$ и $Prev$ используются для организации циклического списка. Указатель $Next$ указывает на следующий элемент списка, а $Prev$ на предыдущий элемент списка. Поле $Block$ указывает на заголовок страницы, который располагается в ее начале, остальная часть страницы заполнена блоками (далее будет отмечено, что поле $Block$ не обязательно).

Общее состояние системы описывается переменными: $BlockSize$ – размер страницы; $FirstFreeItemSize$ – вспомогательная переменная; $list$ – указатель на последний из созданных элементов списка страниц; $freeitem$ – указатель на один из элементов списка свободных блоков.

Зачем нужна переменная $FirstFreeItemSize$? $FirstFreeItemSize$ – это число блоков, свободных и еще не выделявшихся, в странице $list$, то есть в последней выделенной странице. Использование этого параметра позволяет не добавлять все свободные блоки страницы в список свободных блоков при выделении новой страницы. Всего блоков у страницы ($BlockSize$ – размер($LIST$)) / размер($ITEM$). В список свободных блоков помещается только один блок – с номером 0, этот блок мы переносим в переменную $freeitem$. При выделении памяти блоки будут выделяться из страницы $list$ пока $FirstFreeItemSize > 1$. При этом нулевой блок, помещенный в переменную $freeitem$, будет выделен последним.

Выделение блока:

1. Переменная X обозначает выделяемый блок

2. Если список свободных блоков не пуст (указатель $freeitem$ не нулевой), то переходим на шаг 2.1, иначе на шаг 3.

2.1. Получаем страницу P блока $freeitem$

2.2. Если $FirstFreeItemSize > 1$, то выполняем шаг 2.2.1, иначе шаг 2.2.2.

2.2.1. Уменьшаем $FirstFreeItemSize$ на 1 и устанавливаем в переменную X не блок $freeitem$, а блок страницы с номером $FirstFreeItemSize$. Заполняем переменную $Block$ у X. Переходим на шаг 6.

2.2.2. Устанавливаем X равным $freeitem$. Удаляем $freeitem$ из списка свободных блоков страницы P (меняем ее указатели $FreeFirst$, $FreeLast$). Удаляем $freeitem$ из общего списка свободных блоков, в частности устанавливаем переменную $freeitem$ на следующий элемент в списке (или обнуляем $freeitem$ если список состоял из одного элемента). Переходим на шаг 6.

3. Список свободных блоков пуст, выделяем новую страницу, помещаем ее в список страниц, устанавливаем $freeitem$ на нулевой блок страницы.

4. Устанавливаем значение переменной $FirstFreeItemSize$ равным числу блоков в странице, значение переменной $Count$ равным 0. Помещаем нулевой блок страницы в список свободных блоков (также меняем указатели $FreeFirst$ и $FreeLast$ у страницы).

5. Уменьшаем $FirstFreeItemSize$ на 1 и устанавливаем в переменную X не блок $freeitem$, а блок новой страницы с номером $FirstFreeItemSize$. Заполняем переменную $Block$ у блока.

6. Увеличиваем счетчик $Count$ у страницы блока X и возвращаем указатель на поле $Data$ блока X.

Освобождение блока:

1. По адресу определяем указатель на структуру $ITEM$ блока, получаем страницу блока P.

2. Уменьшаем счетчик $Count$ у P.

3. Если Count равен 0, то нужно освободить страницу, удалив также все свободные блоки этой страницы из списка свободных блоков. Переходим на шаг 3.1, иначе на шаг 4.

3.1. Удаляем страницу P из списка страниц.

3.2. Если $freeitem$ равен значению поля $FreeFirst$ страницы P, то устанавливаем значение $FirstFreeItemSize$ равным 1 и $freeitem$ на свободный блок другой страницы (если он есть, то он найдется как следующий за блоком $FreeLast$ страницы P блок в списке свободных блоков)

3.3. Для удаления всех свободных блоков страницы P из списка свободных блоков достаточно изменить указатели у следующего блока за $FreeLast$ и предыдущего блока у $FreeFirst$.

3.4. Выход из процедуры

4. Страница не удаляется из памяти, нужно передать освобождаемый блок в список свободных блоков и обновить указатели $FreeFirst$ и $FreeLast$ у страницы P.

Подобная организация позволяет организовать выделение и освобождение блоков за константное $O(1)$ число операций (без учета выделения страниц, используя функции операционной системы). Все операции ограничены простой арифметикой. Этот алгоритм был реализован в виде класса $CConstantMemoryManager$.

Для реализации выделения блоков размера менее или равных выбранного значения $SMALL\_SIZE$ необходимо создать $SMALL\_SIZE$ объектов, подобных $CConstantMemoryManager$. При этом указатель на конкретный экземпляр соответствующего блоку $ITEM$ класса $CConstantMemoryManager$ нужно поместить в структуру $LIST$.

Следует немного модифицировать структуру $ITEM$:

```
struct ITEM {
  LIST *Block;
  union { char Data[1];
    struct { ITEM *Next; ITEM *Prev; }; }; };
```

При выделении блока мы возвращаем поле $Data$, а при освобождении блока смотрим на память перед ним и получаем указатель $Block$ на структуру $LIST$. Если выделяется блок размером более, чем $SMALL\_SIZE$, то для различия блоков следует выделить блок большего размера, чем указано, на размер указателя и поместить в начало 0 (то есть обнулить указатель $Block$). Впоследствии можно таким образом различать большие и маленькие блоки.

Каким выбрать размер страницы? Для организации выделения па-

мяти по небольшим страницам, например по 4Кб, можно использовать уже описанную схему $CConstantMemoryManager$, а уже здесь использовать большие страницы, например 1 – 10 Мб. Также полезно при освобождении страницы удерживать одну страницу в оперативной памяти и использовать ее при необходимости повторно. Модель выделения страниц оформлена как отдельный класс.

Можно избавиться и от указателя $Block$ в структуре $ITEM$ за счет выравнивания страниц на определенную границу, но в таком случае при освобождении блока нужно будет указывать его размер (только для того, чтобы различать, больше он чем $SMALL\_SIZE$ или нет).

В итоге созданы классы $CExtendedConstantMemoryManager$, имеющий методы $Alloc(size\_t\ Size)$ и $Free(void\ *A)$, и $CExtendedConstantMemoryManagerAlignment$, имеющий метод $Alloc(size\_t\ Size)$ и $Free(void\ *A, size\_t\ Size)$. Для использования их с контейнерами STL созданы вспомогательные классы.

Реализованный алгоритм был сравнен с алгоритмами библиотеки Loki [24], разработанной А. Александреску, и работал более, чем на 30% быстрее, и в 5-7 раз быстрее стандартных функций. При этом выяснилось, что при работе с большим числом блоков (более 10 миллионов) алгоритм Loki начинает работать весьма медленно, а описанный нами алгоритм сохраняет свою эффективность. При тестировании использовался компилятор Microsoft Visual C++ 7.1.

4.2. WindowSystemObject

На сегодняшний день широкое развитие получили различные сценарные (скриптовые) языки, такие как JScript, VBScript, Perl. Эти языки

обладают большими возможностями и могут применяться для решения самых различных задач: системное администрирование, обработка текстов и т. д. Самая простая операционная система использует конфигурационные файлы, являющиеся простейшим видом сценария, например пакетные bat-файлы в DOS, cmd-файлы Windows NT, командные процессоры Un*x. Windows Script Host (WSH) в Windows развивает эту идею, позволяя создавать пакетные файлы на языках JScript или VBScript. При создании программ на сценарных языках отсутствуют проблемы, связанные с управлением памятью, созданием и управлением COM объектами.

К сожалению, сценарные языки не имеют встроенных средств для создания оконных интерфейсов, для вывода результатов работы программы используется в основном консольные интерфейсы. Как правило, средства для создания оконных интерфейсов исчерпываются только функциями для создания типового окна для вывода сообщения с несколькими кнопками «OK», «Отмена»... Некоторые сценарные среды используют библиотеку Tcl/Tk, однако ее возможности сильно ограничены.

Это потребовало создания дополнительного инструментария для решения подобных задач.

Разработанная автором библиотека WindowSystemObject (WSO) [4] предназначена для обеспечения доступа к оконной подсистеме Windows на базе архитектуры COM, для создания оконных интерфейсов в программах, написанных на сценарных языках, а также в программах, написанных на других языках и в других системах программирования.

WSO распространяется бесплатно и доступно по адресу

http://cs.usu.edu.ru/home/abv/.

Основные достоинства WSO:

* WSO обеспечивает полный доступ ко всем возможностям оконной системы, включая рисование в окнах, поддержку всех встроенных управляющих элементов Windows и встроенных диалоговых окон.

* WSO поддерживает использование любых элементов ActiveX, таких как Internet Explorer или Windows Media Player.

* WSO поддерживает работу со всеми популярными форматами графических файлов.

* WSO доступен из любого языка программирования, поддерживающего COM- интерфейсы автоматизации.

* Доступ к WSO осуществляется с помощью интуитивно понятной объектной модели. Программисты, знакомые с оконным программированием для Windows, быстро обнаружат, что ничего нового им осваивать не нужно.

* WSO обеспечивает полную поддержку обработки событий от оконных элементов. Разработана трехуровневая модель задания обработчиков событий.

* WSO работает в любой современной версии Windows, а именно: 98SE, ME, NT 4, 2000, XP, 2003, Vista, 2008.

Большое количество примеров использования WSO подтверждает универсальность разработанного программного обеспечения. Написаны примеры использования WSO в JScript, VBScript, Perl, Pascal (Delphi), Java, Python, HTML, WSH, WSC.

При создании интерфейса необходимо не только создать форму и на ней кнопку, но жизненно важно определить действия, которые будут

происходить при нажатии на эту кнопку. В WSO существует развитая система задания обработчиков различных событий, предусматривающая три уровня:

* Базовый уровень.

* Уровень непосредственного определения.

* Уровень списков обработчиков.

Эти три уровня позволяют задавать обработчики событий для WSO почти в любом языке программирования.

Литература

[1] *Адельсон-Вельский Г. М., Ландис Е. М.* Один алгоритм организации информации // *Докл. АН. СССР.* — 1962. — Vol. 146. — Pp. 263–266.

[2] *Александреску А.* Современное проектирование на C++. — М.: Вильямс, 2002. — 335 с.

[3] *Веретенников А. Б.* Новый подход к быстрому выделению памяти в программах на c++ // *Проблемы теоретической и прикладной математики: Труды 37-й Региональной молодежной конференции. Екатеринбург: УрО РАН.* — 2006. — С. 413–417.

[4] *Веретенников А. Б.* Библиотека для создания оконных интерфейсов на любых скриптовых языках в операционной системе windows // *Информационно-математические технологии в экономике, технике и образовании: Тезисы докладов Третьей международной научной конференции. Екатеринбург: УГТУ-УПИ.* — 2008. — С. 220–221.

[5] *Веретенников А. Б.* Создание легко обновляемых текстовых индексов // *Электронные библиотеки: перспективные методы и технологии, электронные коллекции: Труды Десятой Всероссийской научной конференции «RCDL'2008». Дубна: ОИЯИ.* — 2008. — С. 149–154.

[6] *Веретенников А. Б.* Эффективное создание текстовых индексов // *Проблемы теоретической и прикладной математики: Труды 39-й Всероссийской молодежной конференции. Екатеринбург: УрО РАН.* — 2008. — С. 348–350.

[7] *Веретенников А. Б.* Гибкий подход к проблеме поиска похожих документов // *Проблемы теоретической и прикладной математики: Труды 40-й Всероссийской молодежной конференции. Екатеринбург: УрО РАН.* — 2009. — С. 392–396.

[8] *Веретенников А. Б.* Размышление об использовании treap при работе с неравномерно распределенными данными // *Материалы межвузовской научной конференции по проблемам информатики «СПИСОК 2009».* — 2009. — С. 14–18.

[9] *Веретенников А. Б.* Сравнение эффективности clb-дерева в 32-битных и 64-битных архитектурах // *Материалы межвузовской научной конференции по проблемам информатики «СПИСОК 2009».* — 2009. — С. 7–13.

[10] *Веретенников А. Б.* Эффективная индексация текстовых документов с использованием clb-деревьев // *Системы управления и информационные технологии.* — 2009. — Т. 1.1, № 35. — С. 134–139.

[11] *Веретенников А. Б., Лукач Ю. С.* Clb-деревья: новый способ индексации больших массивов текстов // *Международная алгебраическая конференция: К 100-летию со дня рождения П. Г. Конторовича и 70-летию Л. Н. Шеврина. Тез. докл. Екатеринбург: Изд-во Урал. ун-та.* — 2005. — С. 173–175.

[12] *Веретенников А. Б., Лукач Ю. С.* Еще один способ индексации больших массивов текстов // *Известия Уральского государственного университета. Серия «Компьютерные науки»*. — 2006. — № 43. — С. 103–122.

[13] *Кнут Д. Э.* Искусство программирования. Том 3. Сортировка и поиск. — Москва, Санкт-Петербург, Киев: Вильямс, 2000. — 822 с.

[14] *Лукач Ю. С.* Быстрый морфологический анализ флективных языков // *Международная алгебраическая конференция: К 100-летию со дня рождения П. Г. Конторовича и 70-летию Л. Н. Шеврина. Тез. докл. Екатеринбург: Изд-во Урал. ун-та.* — 2005. — Pp. 182–183.

[15] *Aragon C. R., Seidel R.* Randomized search trees // *Foundations of Computer Science, 30th Annual Symposium.* — 1989. — Pp. 540–545.

[16] *Bayer R., McCreight E. M.* Organization and maintenance of large ordered indexes // *Acta Informatica.* — 1972. — Vol. 1, no. 3. — Pp. 173–189.

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

[18] *Benteley J. N., Sedgewick R.* Fast algorithms for sorting and searching strings // *8th ACM-SIAM Symposium on Discrete Algorithms.* — 1997.

[19] *Berry M. W., Browne M.* Understanding Search Engines: Mathematical Modeling and Text Retrieval. — 2 edition. — Philadelphia: Society for Industrial and Applied Mathematics, 2005. — 117 pp.

[20] *Brin S., Page L.* The anatomy of a large-scale hypertextual web search engine // *Computer Science Department, Stanford University, Stanford.* — 2000.

[21] *Ferragina P., Grossi R.* An experimental study of sb-trees // *7th ACM-SIAM symposium on Discrete Algorithms.* — 1996.

[22] *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, no. 2. — Pp. 236–280.

[23] *Gonnet G. H., Baeza-Yates R. A., Snider T.* Linear pattern matching algorithm // *Information Retrieval: Data Structures and Algorithms. Prentice-Hall.* — 1992. — Pp. 66–82.

[24] *http://www.awl.com/cseng/titles/0 201-70431-5.*

[25] Incremental construction of minimal acyclic finite state automata / J. Daciuk, M. S., B. Watson, R. Watson // *Computational Linguistics.* — 2000. — Vol. 26, no. 1. — Pp. 3–16.

[26] *Lancaster P.* Mathematics: Models of the Real World. — Englewood Cliffs, New Jersey: Prentice-Hall, 1976. — 164 pp.

[27] *Manber U., Myers G.* Suffix arrays: a new method for on-line string searches // *SIAM Journal on Computing.* — 1993. — Vol. 22, no. 5. — Pp. 935–948.

[28] *McCreight E. M.* A space-economical suffix tree construction algorithm // *Journal of the ACM.* — 1976. — Vol. 23, no. 2. — Pp. 262–272.

[29] *Morrison D. R.* Patricia: Practical algorithm to retrieve information coded in alphanumeric // *Journal of ACM.* — 1968. — no. 15. — Pp. 514–534.

[30] *Prywes N. S., Gray H. J.* The organization of a multilist-type associative memory // *IEEE Trans. on Communication and Electronics.* — 1963. — no. 68. — Pp. 488–492.

[31] *Weiner P.* Linear pattern matching algorithm // *IEEE Symp. on Switching and Automata Theory.* — 1973. — Pp. 1–11.

[32] *Witten I. H., Moffat A., Bell T. C.* Managing Gigabytes: Compressing and Indexing Documents and Images. — 2 edition. — San Diego, CA: Academic Press, 1999. — 519 pp.

[33] *www.aot.ru.*