УДК 519.683.5 DOI: 10.14529/cmse180105

ПРИМЕНЕНИЕ ТРЕХКОМПОНЕНТНЫХ КЛЮЧЕЙ ДЛЯ ПОЛНОТЕКСТОВОГО ПОИСКА С УЧЕТОМ РАССТОЯНИЯ С ГАРАНТИРОВАННЫМ ВРЕМЕНЕМ ОТКЛИКА

© 2018 А.Б. Веретенников

Уральский федеральный университет
(620083 Екатеринбург, пр. им. В.И. Ленина, д. 51)
E-mail: alexander@veretennikov.ru

Поступила в редакцию: 28.11.2017

Рассматриваются задачи поиска фраз и наборов слов в большом объеме текстов. В результате поиска получаем список документов, содержащих заданные слова, при этом документы, где слова располагаются ближе друг к другу, считаются более релевантными. Поскольку эта задача требует сохранения в индексе информации о каждом вхождении каждого слова в текстах, запросы, включающие часто встречающиеся слова, требуют для своего выполнения длительного времени. В некоторых поисковых системах предлагается ввести список стоп слов, которые не учитываются при поиске, но этот подход снижает качество поиска. В данной работе при поиске обрабатываются все слова и применяются дополнительные индексы. С помощью дополнительных индексов время выполнения поискового запроса, включающего часто встречающиеся слова, может быть снижено в десятки раз. Разработан новый вид индекса с трехкомпонентными ключами. Приведены алгоритмы поиска и результаты экспериментов поиска в сравнении с обычными индексами. Эксперименты показывают, что при применении разработанных индексов для определенного класса запросов, состоящих из самых часто встречающихся слов, скорость поиска возрастает более чем в 90 раз.

Ключевые слова: полнотекстовый поиск, поисковые системы, инвертированные файлы, дополнительные индексы, поиск с учетом близости слов.

ОБРАЗЕЦ ЦИТИРОВАНИЯ

Веретенников А.Б. Применение трехкомпонентных ключей для полнотекстового поиска с учетом расстояния с гарантированным временем отклика // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. Т. 7, № 1. С. 60–77. DOI: 10.14529/cmse180105.

Введение

Задачей полнотекстового поиска является поиск документов, содержащих заданные слова. Пользователь получает список документов и информацию о том, где в них располагаются слова запроса. Результат поиска — это набор записей, каждая из которых содержит идентификатор документа, название документа, и другую информацию о нем, а также позицию начала и позицию конца фрагмента текста, содержащего искомые слова.

Если слова запроса располагаются в документе вблизи, то такие записи можно считать более релевантными и поместить в начало списка результатов. Чем ближе слова запроса в документе друг к другу, тем большее значение имеет эта запись в отношении заданного поискового запроса. Если есть две записи, где в соответствующих документах искомые слова располагаются друг от друга на, примерно, одинаковом расстоянии, то можно применить отдельную функцию релевантности, для того, чтобы определить, какая из них более важна.

Эта задача (Задача 1) требует для своего решения сохранения в индексе информации о каждом вхождении каждого слова. Поэтому, время выполнения запроса зависит от суммарного объема текстов (например, в байтах). Кроме того, чем чаще встречаются слова запроса в текстах, тем дольше осуществляется поиск. Возникают ситуации, когда в целом поиск работает быстро, но при обработке поискового запроса, включающего часто встречающиеся слова, время обработки запроса существенно возрастает и становится неприемлемым с точки зрения пользователя.

Цель и задачи исследования: разработать методы повышения скорости поиска в случае наличия в составе поискового запроса часто встречающихся слов, разработать структуру данных, позволяющую выполнять любой запрос быстро.

Статья организована следующим образом. В разделе 1 рассмотрено: допустимое время обработки поискового запроса, факторы, определяющие время поиска, преимущества и недостатки методов повышения производительности поиска — введение списка стоп слов, упорядочение по важности, ускорение поиска при помощи дополнительных индексов. Раздел 2 посвящен постановке задачи поиска с учетом расстояния и сведения ее к двум задачам, которые можно решать более эффективно. В разделе 3 сформулирован разработанный автором метод организации дополнительных индексов, применяемый для решения подзадачи поиска запросов, все слова которых выбраны из подмножества самых часто встречающихся слов. В разделе 4 предложен алгоритм поиска. В разделе 5 приводятся результаты экспериментов, позволяющие оценить эффективность метода и тем самым обосновать полезность его применения. В заключении кратко рассмотрены полученные результаты и направления дальнейших исследований.

1. Обзор работ по теме исследования

1.1. Скорость выполнения поискового запроса

В [1] даны оценки желаемого времени отклика в информационных системах. Поисковый запрос, состоящий из нескольких слов, можно рассмотреть как вариант простого запроса информации. Для этого случая дана рекомендация максимального времени выполнения запроса — 2 секунды. При этом указано, что если время ожидания превышает одну секунду, возникает вероятность того, что непрерывность мышления будет нарушена (т. е. пользователь начнет думать о чем-то другом). Таким образом, при максимальном времени выполнения запроса 2 секунды, можем рассмотреть время выполнения запроса, не более 1 секунды, как рекомендуемое.

Для полнотекстового поиска применяются инвертированные файлы [2] и их аналоги [3, 4]. Скорость поиска зависит от того, с какой частотой слова запроса встречаются в текстах. Считается, что распределение частот встречаемости слов соответствует закону Ципфа [5]. Запросы, включающие часто встречающиеся слова, могут выполняться существенно дольше, чем запросы, состоящие из редко используемых слов, так при поиске с учетом расстояния скорость выполнения запроса линейно зависит от числа вхождений слов запроса в индексируемых текстах.

В качестве варианта решения проблемы можно игнорировать «проблемные» слова при поиске. Вводится список стоп слов, включающий в себя самые часто встречающиеся слова, и эти слова пропускаются при поиске. В английском языке такие слова, как «to, be, not, or, who, you, are», а в русском, «и, я, кто, где», обычно включаются в список стоп слов. Это ведет к потере качества поиска. В [6] делается вывод, что нужно учитывать

все слова, авторы утверждают, что пропуск часто встречающихся слов может привести к непредсказуемым эффектам. Часто встречающиеся слово может иметь особый смысл в конкретном документе или в составе фразы, предложения. Мы приведем пример: «who are you who», «time and a word yes». Часть слов в этих запросах имеет особый смысл, например Who, Yes — наименования исполнителей («Who are you» и «Time and a word» — наименования произведений). Если поисковый запрос состоит полностью из пропускаемых слов (например, «to be or not to be» или «from me to you»), то он вообще не может быть выполнен.

1.2. Упорядочение по важности

Инвертированный файл или инвертированный индекс — это ассоциативный массив. В качестве ключа может выступать, например, слово, базовая форма слова, набор слов, набор базовых форм слова. Например, в [6, 7] ключ — пара слов или даже фраза. Значение ключа — это список записей вида (ID, P). ID — это идентификатор документа. P — позиция ключа в документе, например, порядковый номер слова в документе. Запись (ID, P) будем называть записью о вхождении слова в документе или словопозицией. ID будем считать целым числом, например, номером документа. Определим, что словопозиция A меньше словопозиции B, если $A.ID < B.ID$ или ($A.ID = B.ID$ и $A.P < B.P$). Инвертированный файл состоит из файла данных и словаря. Файл данных хранит списки словопозиций. Словарь хранит ключи и для каждого ключа дескриптор — структура, содержит информацию о том, где в файле данных располагаются словопозиции этого ключа.

В [8–10] предлагается отсортировать данные в индексе в порядке релевантности документов и прерывать поиск, если прочитано много данных. То есть, список словопозиций для каждого ключа упорядочен таким образом, что словопозиции более релевантных документов находятся в начале списка. При поиске чтение списка словопозиций можно прервать (early termination), как только все релевантные документы будут обработаны. Критерий остановки поиска можно задавать некими пороговыми параметрами. В [10] учитывают при сортировке списка словопозиций результаты обработки пользовательских поисковых запросов. Если какие-то документы часто при поиске отображаются в начале списка результатов, то данные этих документов размещаются в начале списка словопозиций.

Однако нет никакой гарантии, что в том документе, который функцией релевантности оценен не высоко, не окажется рядом располагающихся требуемых слов или фразы. Сортировка данных в индексе с учетом релевантности не позволяет учитывать расстояние в документе между словами произвольного поискового запроса. Есть риск пропуска требуемых документов при поиске. В [8] обосновывают подход тем, что при его применении результаты поиска выглядят достаточно хорошо (demonstrably «good enough», то есть, пользователь не способен обнаружить проблему), а скорость поиска возрастает. В [9] предлагают ориентироваться на уровень загрузки системы, и только при высокой загрузке применять прерывание поиска. В ряде систем подобные доводы могут быть применимы, но правильнее считывать все данные из индекса, а уже потом сортировать результаты по релевантности.

1.3. Дополнительные индексы

Применение дополнительных индексов исследовалось в [6, 7] для решения задачи поиска фраз. Авторы вводят два вида дополнительных индексов. Во-первых, nextword индекс, где ключ — пара слов, значение — список мест в текстах, где второе слово располагается прямо за первым. Фраза при поиске разделяется на набор пар слов. Во-вторых, частичный индекс фраз, ключ — фраза, значение — список вхождений фразы в текстах. Данный индекс применяется для фраз, включающих часто встречающиеся слова, так как для них nextword индекс работает медленно. Искомая фраза в тексте должна полностью, вплоть до порядка слов, совпадать с поисковым запросом. Это не позволяет применять методы [6, 7] для рассматриваемой задачи поиска с учетом расстояния. Требуется разработать другие виды дополнительных индексов.

В [11] создаются дополнительные индексы, ключ в которых — пара слов. Ключ формируется для такого вхождения двух слов в тексте, где эти слова располагаются близко друг к другу (авторы рассматривают расстояния в 1, 2, и 3). Для каждого такого ключа хранятся данные, соответствующие вхождениям слов ключа в текстах и требуемые для расчета значения функции релевантности. Авторы [11] при выполнении запроса вначале используют индекс пар слов, который им позволяет сформировать первоначальный список документов, для каждого из которых рассчитана релевантность в соответствие с их функцией. Далее, анализируется обычный индекс для поиска дополнительных результатов. Если первоначальный список уже содержит высоко релевантные записи, то чтение обычного индекса можно сократить частично или полностью. То есть, дополнительные индексы применяются как дополнительный компонент для более оптимального прерывания поиска (early termination) в обычном индексе.

Существенным недостатком работы [11] является то, что основной алгоритм и структуры данных применяются только для запросов, состоящих из ровно двух слов. Вопросы, связанные с выполнением более длинных запросов, в [11] не решены. Можно предположить, что с усложнением функции релевантности реализовывать логику раннего прерывания поиска (early termination) для сложных запросов затруднительно.

Можно также отметить, что индексы, где ключом является пара слов, как правило не применимы при обработке запросов, включающих самые часто используемые слова. Если мы рассмотрим, к примеру, пару слов «you who», то число вхождений этой пары в текстах будет настолько большим, что использование индекса с таким ключом не будет иметь смысла, с точки зрения производительности, при поиске с учетом расстояния, если поисковый запрос состоит более чем из двух слов.

2. Постановка задачи

2.1. Разделение Задачи 1 на подзадачи

На основании рассмотренного выше можем сформулировать следующие утверждения, которые будем учитывать при решении задачи:

- Нужно учитывать все слова при поиске.
- Методы раннего прерывания поиска (early termination) не позволяют учитывать расстояние между словами запроса в тексте.
- Поиск должен осуществляться быстро, в пределах 1–2 с, что трудно достичь, если в составе запроса присутствуют самые часто встречающиеся слова.

Задача 1. Поисковый запрос — набор слов. Результат поиска — список записей (ID, P, E), где ID — идентификатор документа, P — начало фрагмента, содержащего слова запроса, E — конец фрагмента. Документы, где искомые слова располагаются близко друг к другу, являются более релевантными по отношению к заданному поисковому запросу. Поэтому результаты сортируются в соответствии с длиной фрагмента $(E - P + 1)$ по возрастанию. Затем применяется дополнительная функция релевантности для сортировки записей с одинаковой длиной фрагмента.

В более формализованном случае, записи могут сортироваться в соответствии с одной функцией релевантности, вида $S = SR{+}IR{+}TP$ (детали реализации и параметры опустим). Где SR – статический ранг документа ID, независящий от поискового запроса, например PageRank. IR позволяет учитывать слова запроса, например, BM25. TP учитывает расстояние между словами запроса в тексте документа для конкретной записи (ID, P, E). Пример см. в [11]. Обычно считают, что TP определяется на основании предположения, что важность связи между словами обратно пропорциональна квадрату расстояния между ними в тексте [11] (на основании этого также можно ввести пороговое значение расстояния, до которого связь между словами важна и TP имеет высокое значение, после которого значение TP мало по сравнению с SR и IR). Поэтому два приведенных подхода сортировки записей дают похожие результаты.

Если слова располагаются в тексте близко друг к другу, то предполагаем, что они связаны друг с другом по смыслу, являются частью некого предложения или высказывания (поэтому, чем ближе слова друг к другу в тексте, тем такие документы более важны). Можно считать, что они сильно связаны друг с другом. Если же слова располагаются на значительном расстоянии в тексте, то они могут быть связаны по тематике или предметной области, но расстояние между ними уже не имеет существенного значения. Данный случай будем обозначать как слабую связь. Т. е. существует некое расстояние $MaxDistance$, если слова располагаются друг от друга на расстоянии не более $MaxDistance$, то связь между ними сильная и это расстояние нужно учитывать. Иначе связь слабая, и расстояние учитывать не нужно. Значение $MaxDistance$ задаем как параметр построения индекса.

Поэтому Задача 1 может быть сведена к Задачам 2 и 3.

Задача 2. Задача полнотекстового поиска с учетом близости слов: ищем только те документы, в которых слова запроса располагаются близко друг к другу, на расстоянии, не превышающем заданного параметра $MaxDistance$.

Задача 3. Задача полнотекстового поиска без учета расстояния: поиск документов, в которых искомые слова могут находиться где угодно.

Задача 2 решается более эффективно с помощью использования дополнительных индексов. Задача 2 более общая, чем поиск фраз, и методы из [6, 7] здесь не применимы. Достаточно наличия хотя бы одного лишнего слова в тексте между искомыми словами, и методами [6, 7] мы не найдем данный документ. Например, если в тексте мы имеем «Time and a word by Yes», а поисковый запрос «Time and a word Yes». Для эффективного решения Задачи 2 автором были разработаны другие подходы [12, 13].

Для решения Задачи 3, в отличие от Задачи 1, для конкретного слова нужно сохранять только информацию о первых его вхождениях в документах. Что соответствует информации о том, есть ли данное слово в документе или нет. Время выполнения запроса зависит от количества документов. Если индексируемые документы достаточно велики, (например, это книги, статьи, объемом от 10–20 страниц), то запрос выполняется на порядок

быстрее, чем в случае Задачи 1. Кроме того, искать в условиях Задачи 3 не нужно, если все слова запроса являются часто встречающимися или стоп словами, так как по отдельности такие слова существенного смысла не несут.

Если вначале осуществим поиск в условиях Задачи 2 с применением дополнительных индексов, а затем, при необходимости, в условиях Задачи 3, то получим результаты, эквивалентные поиску в условиях Задачи 1. Ускорением поиска в условиях Задачи 2 мы обеспечиваем ускорение поиска в общем случае. Методы [12, 13] позволяют искать в условиях Задачи 2, в десятки раз быстрее, чем в общем случае, то есть в условиях Задачи 1.

3. Дополнительные индексы с трехкомпонентными ключами

3.1. Виды слов и морфологический анализатор

Разделим слова на три группы, по их частоте встречаемости [12]:

1) «стоп слова»: и, в, или. Встречаются очень часто и могут, в некоторых поисковых системах, вообще не включаться в индекс, поэтому их и можно называть стоп словами;
2) часто используемые слова;
3) остальные, будем называть их «обычные слова».

В поисковом запросе учитываются все слова, включая стоп слова.

При создании индекса используем морфологический анализатор. Для каждой словоформы, входящей в словарь анализатора он возвращает список номеров базовых форм слов. Номер базовой формы это число в диапазоне от 0 до $WordsCount - 1$, где $WordsCount$ — число различных базовых форм (около 260 тыс. для используемого анализатора). Если слово не входит в словарь анализатора, считаем, что его базовая форма совпадает с самим словом. Термины слово и словоформа считаем взаимозаменяемыми. Базовые формы слова также называются леммами, а сам процесс получения набора лемм по словоформе — лемматизацией. С учетом морфологического анализа деление слов на три группы применяем не к исходным словоформам, а к леммам. Имеем три типа лемм в смысле частоты встречаемости: стоп леммы, часто используемые леммы и остальные.

FL — список известных лемм, отсортированный в порядке убывания частоты их появления в текстах. Введем параметр $WsCount$, например, $WsCount = 700$. Будем также называть порядковый номер леммы в FL-списке — FL-номером, нумерация с нуля. Пусть первые $WsCount$ элементов списка FL, то есть, первые $WsCount$ самые часто встречаемые леммы, это стоп леммы. Далее, следующие 500–5000 — часто используемые леммы. Количество стоп и часто используемых лемм зависит от числа поддерживаемых анализатором языков, в данной работе их два, русский и английский.

3.2. Виды запросов

Поисковый запрос это набор слов. Каждое слово имеет список лемм. Поисковый запрос после применения лемматизации назовем лемматизированным поисковым запросом. Лемматизированный поисковый запрос — это набор списков лемм. Будем называть один список лемм в составе лемматизированного поискового запроса — клеткой.

Нужно обеспечить, чтобы в каждой клетке запроса присутствовали только леммы одного вида [13] (стоп леммы, часто используемые или обычные). Например, запрос «солнце село». Слово «село» имеет две леммы: «село» — часто используемая, «сесть» — стоп лемма. После лемматизации имеем запрос из двух клеток: [солнце] [село, сесть]. Он преобразуется в два поисковых запроса, [солнце] [село] и [солнце] [сесть].

Методы из [13] эффективно решают Задачу 2 за исключением случая, когда все леммы запроса — стоп леммы. В данной работе предлагается решение Задачи 2 для этого случая. Эффективное решение этого случая важно, так как остальные случаи уже решены в [12, 13].

3.3. Расширенный индекс стоп лемм

Для двух словопозиции A и B расстояние между ними определим как $|A.P - B.P|$, если $A.ID = B.ID$. Если $A.ID \neq B.ID$, то расстояние неопределено. Рассмотрим пример: «ведь за полночь уже, разбудим». Т. к. у каждого слова есть номер, скажем: слова «ведь» и «разбудим» в тексте на расстоянии 4 друг от друга. Т. к. у каждого слова есть одна или несколько лемм, скажем: леммы «ведь» и «разбудить» в тексте на расстоянии 4 друг от друга. Леммы «полночь» и «узкий» на расстоянии 1 друг от друга. Слово «уже» имеет три леммы, у остальных слов одна лемма: [ведь], [за], [полночь], [уж, уже, узкий], [разбудить]. Или, [242], [44], [—1], [71, 88, 86], [—1], если заменим леммы на их номера в FL-списке для стоп лемм, и оставим (-1) для остальных.

Расширенный индекс стоп лемм (f, s, t) — это список словопозиций стоп леммы f, таких, что в тексте не более чем на расстоянии $MaxDistance$ от f располагались стоп леммы s и t. Значения f, s и t — номера соответствующих стоп лемм в FL-списке. $MaxDistance$ — заданный параметр, например, 5. Словопозиции одного ключа сохраняем по возрастанию.

Если храним расширенный индекс (f, s, t), то хранить расширенный индекс, с другим порядком компонент, например, (f, t, s) или (s, t, f) — избыточно. Данные (f, t, s), (s, t, f) и других комбинаций порядка f, s, t восстанавливаются по данным (f, s, t). Будем создавать для трех стоп лемм f, s, t, только такой расширенный индекс (f, s, t), что $f \leq s \leq t$. Что означает: f встречается в тексте не реже, чем s, а s встречается в тексте не реже, чем t. В составе словопозиции сохраняем расстояние от s до f и от t до f. Нужно также указывать для s и t, была ли соответствующая лемма в тексте до или после f. Если s была после f в тексте, сохраняем положительное число, иначе отрицательное. Аналогично для t.

Пример 1. Рассмотрим текст $ID1$: скажи мне, кто твой самый близкий друг. Леммы слов запроса: [сказать], [я], [кто], [твой], [самый], [близкий], [друг]. Примеры словопозиций:
Ключ (я, самый, твой) —> $(4, 100, 236)$: словопозиция $(ID1, 1, 3, 2)$, где 1 — номер леммы «я» в тексте, начиная с нуля, 3 — расстояние между леммами «самый» и «я», 2 — расстояние между «твой» и «я», $ID1$ — идентификатор документа, (4, 100, 236) — FL-номера лемм.
Ключ (я, сказать, друг) —> $(4, 58, 170)$: словопозиция $(ID1, 1, -1, 5)$. Расстояние (-1) между «сказать» и «я», т. к. лемма «сказать» в тексте присутствует до леммы «я», 5 — расстояние между «я» и «друг».

Один индекс (f, s, t) представляет собой список записей вида $(ID, P, Delta1, Delta2)$, каждая из которых соответствует вхождению в документе лемм f, s, t, где: ID — идентификатор документа, P — позиция леммы f в документе, то есть, номер слова, $Delta1$ — расстояние от f до s в документе, $Delta2$ — расстояние от f до t в документе.

Словопозиции в индексе (f, s, t) упорядочены по возрастанию. Введем понятие объекта-ленты или итератора словопозиций. Данный объект имеет операцию $Next$, возвращающую следующую запись и поле $Value$, которое содержит текущую запись. Записи, которые выдает итератор, упорядочены, то есть следующая запись должна быть не меньше предыдущей.

4. Поиск

4.1. Поисковый запрос в структурированном виде

Входным параметром поиска является массив стоп лемм. Каждую лемму заменяем на ее номер в FL-списке, получая массив номеров V. Каждый элемент массива V представляет собой лемму, закодированную в виде числа. Нумерация элементов с нуля. В качестве $V.Count$ обозначим количество элементов массива, в качестве $V[x]$ — элемент массива с номером x, нумерация с 0. Введем переменные: $Marked$ — число лемм запроса, обнаруженных на заданной позиции. ITL — список итераторов словопозиций.

Определим аккумулятор позиций, как объект соответствующий одной лемме. Аккумулятор позиций содержит в себе информацию о лемме и вспомогательные поля, требуемые для поиска. Аккумуляторы нужны, чтобы на заданной позиции текста накопить данные о вхождениях в тексте вблизи нее требуемых лемм и определить фрагмент текста минимальной длины, в котором присутствуют все леммы запроса в нужном количестве. Аккумулятор позиций имеет следующие поля:
$Form$: номер леммы в FL-списке.
$Index$: локальный идентификатор (номер) леммы или индекс аккумулятора.
$Count$: счетчик встречаемости леммы в запросе поиска.
$Marked$: счетчик найденных вхождений леммы.
$Positions$: массив позиций найденных лемм, в этом массиве $Count$ элементов.
$PosTable$: битовый массив позиций найденных лемм, представлен двумя целыми 64-битными числами. В текущей реализации, без ограничения общности, $MaxDistance < 64$.

Поисковый запрос в структурированном виде — это список аккумуляторов позиций.

$AccCt$ — количество аккумуляторов позиций, инициализируем его значением 0.

4.2. Алгоритм поиска

Запрос — несколько лемм. Выберем одну лемму запроса f.

Рассмотрим набор индексов вида (f, x, y), где x, y — выбираются из множества остальных лемм запроса. Каждый из этих индексов содержит словопозиции вхождения леммы f в документах. Рассмотрим в документе фиксированную позицию (ID, P) вхождения леммы f. Требуется проверить с помощью этих индексов, что рядом с f присутствовали все остальные леммы запроса, в том количестве, в котором они присутствуют в запросе.

Один аккумулятор позиций S соответствует одной лемме. Если S соответствует лемме g, отличной от f, то его поле $S.Count$ равно количеству встречаемости этой леммы g в запросе V. Если S соответствует лемме f, то $S.Count$ равно количеству встречаемости f в запросе V, минус 1, так как одно вхождение леммы f соответствует самой позиции P.

Читая последовательно словопозиции из индекса (f, x, y), вида $(ID', P', Delta1, Delta2)$, такие что $ID' = ID$ и $P' = P$, отмечаем в аккумуляторах позиций для x и y наличие вхождения их лемм в тексте. Накапливаем число вхождений в поле $Marked$ аккумуляторов. При этом, для любого аккумулятора S, не увеличиваем значение $S.Marked$ больше, если достигнуто $S.Marked = S.Count$.

Искомый запрос присутствует в тексте, если сумма значений $Marked$ всех аккумуляторов позиций равна $(V.Count - 1)$. Глобальную переменную $Marked$ будем

использовать для суммы значений $Marked$ аккумуляторов. Когда увеличиваем $S.Marked$ аккумулятора S, также увеличиваем значение глобальной переменной $Marked$.

Дополнительно аккумулятор S используется для отслеживания самых близких к P позиций леммы $S.Form$. Для этого используется поле $S.Positions$. Поле $S.PosTable$ используется, чтобы исключить дубли позиций леммы $S.Form$, если они вдруг будут обнаружены.

Поскольку позиция леммы f фиксирована и равна P, то позиция другой леммы g (где g — это x или y) определяется расстоянием $Delta$ от P. Это положительное число, если g располагается после f и отрицательное, если g располагается до f. По модулю значение $Delta$ ограничено, так как g располагается близко к f. Поле $S.PosTable$ состоит из двух 64-битных чисел. Одно из них используется в случае $Delta > 0$, другое, если $Delta < 0$. Если мы обрабатываем вхождение леммы g, то выставляем в 1 бит с номером $|Delta|$ в соответствующем числе. Если этот бит был уже выставлен, обрабатываемое вхождение игнорируется. Далее алгоритм поиска структурируем и опишем в виде нескольких процедур.

4.3. Выбор набора индексов

Выберем лемму R запроса V, которая чаще всего встречается в текстах. Обозначим в качестве основного индекса $Main$ индекс первого вхождения этой леммы в V, в частности, $R = V[Main]$. Ключ индекса представляет собой тройку (f, s, t), где f, s, t — номера лемм в FL-списке. В качестве f для всех индексов берем R, т. е. элемент запроса, лемма которого чаще всего встречается в текстах. В качестве s и t нужно выбрать другие леммы, входящие в запрос, таким образом, чтобы все леммы запроса были учтены. Функция $NextSearchIndex(Current)$ возвращает индекс следующего рассматриваемого элемента:

1) Увеличиваем $Current$ на 1.
2) Если $Current$ равен $Main$, то увеличиваем $Current$ на 1.
3) Если $Current \geq V.Count$, присваиваем $Current = 0$ (циклический сдвиг).
4) Если $Current$ равен $Main$, то увеличиваем $Current$ на 1.
5) Возвращаем $Current$ в качестве результата.

Таким образом, с помощью этой функции мы последовательно и циклически перебираем леммы запроса, при этом пропускаем индекс $Main$.

4.4. Выбор индексов, формирование списка аккумуляторов

Предполагаем, есть процедура $AddAcc(Form)$: если аккумулятор S с условием $S.Form = Form$ уже создан, увеличиваем его поле $S.Count$ на 1, иначе создаем новый аккумулятор ($S.Form = Form$, $S.Index = AccCt$, $S.Count = 1$) и увеличиваем $AccCt$ на 1.

Выбор индексов и формирование списка аккумуляторов выполняется в следующем фрагменте псевдокода. Присваиваем $Start = (-1)$. Выполняем в цикле:

1) Присваиваем $A = NextSearchIndex(Start)$.
2) Если $A < Start$, то выходим из цикла. Все леммы запроса учтены, так как мы циклически снова перешли через начало массива запроса V.
3) Присваиваем $B = NextSearchIndex(A)$.
4) Если $A > Start$, то выполняем $AddAcc(V[A])$.
5) Если $B > Start$, то выполняем $AddAcc(V[B])$.
6) Присваиваем $A1 = min(V[A], V[B])$, $B1 = max(V[A], V[B])$.

7) Присваиваем VA = значение поля $Index$ аккумулятора $A1$ ($VA = S.Index$ аккумулятора S с условием $S.Form = A1$).
8) Присваиваем VB = значение поля $Index$ аккумулятора $B1$.
9) Определяем ключ индекса (f, s, t), где $f = R$, $s = A1$, $t = B1$. Формируем итератор IT, с полями $IT.VA = VA$, $IT.VB = VB$, ключ индекса = (f, s, t), помещаем его в ITL.
11) Если $B < Start$, выходим из цикла, иначе: присваиваем $Start = B$, идем на шаг 1.

Для каждого аккумулятора S инициализируем массив $S.Positions$ размером $S.Count$.

Если просуммируем $Count$ у всех аккумуляторов, то получим значение $(V.Count - 1)$.

Пример 2. Рассмотрим запрос «who are you who». В нашем словаре слово «are» имеет две леммы, «are» и «be», имеем лемматизированный запрос [who], [are, be], [you], [who], который преобразуем в два подзапроса, которые выполняем независимо.

Первый подзапрос [who], [are], [you], [who] —> [293], [268], [47], [293]. Лемма «you» имеет минимальный номер в FL-списке — 47, поэтому ее индекс 2 — основной. Действуя циклически, вначале выбираем леммы «who», «are» и вместе с леммой «you» формируем ключ (you, are, who) = (47, 268, 293). Далее пропускаем «you» и выбираем «who» (позиция 3), а затем снова «who» (позиция 0), формируем ключ (you, who, who) = (47, 293, 293). Имеем два аккумулятора: $(Form = 293, Index = 0, Count = 2)$, $(Form = 268, Index = 1, Count = 1)$. Второй подзапрос рассматривается аналогично.

4.5. Процедура поиска

На входе имеем список итераторов ITL. UML диаграмма поиска приведена на рис. 1. В цикле выполняются две операции. Вначале $Equalize$ — применяется для того, чтобы во всех итераторах перейти в одному документу. Значения $Value.ID$ у всех итераторов теперь одинаковые, то есть, соответствуют одному документу, или же поиск завершается, если все данные итераторов обработаны. Далее вызывается процедура $SearchInDoc(Value.ID)$ где выполняется поиск в этом документе.

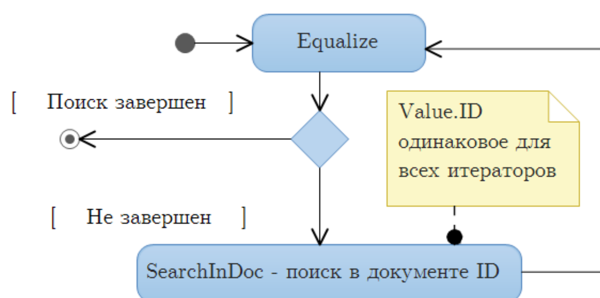

Рис. 1. Процедура поиска

Процедура выравнивания $Equalize$ реализуется в виде цикла:

1) Если значение $Value.ID$ для всех итераторов одинаковое, выходим из процедуры.
2) Выбираем такой итератор IT, значение $IT.Value.ID$ которого минимально среди всех итераторов. Выполняем $IT.Next$. Если перебрали все записи итератора IT, то выходим из поиска. Иначе, переходим на пункт 1, то есть к началу цикла.

Процедуру $Equalize$ можно реализовывать используя две бинарные кучи [14], вычислительная сложность итерации цикла будет $O(log(\text{количество итераторов}))$.

4.6. Поиск в документе ID, процедура $SearchInDoc(ID)$

При входе в эту процедуру значения $Value.ID$ у всех итераторов одинаковые и равны ID. UML диаграмма процедуры $SearchInDoc$ приведена на рис. 2. Предполагаем, есть процедура $InitAccumulators$, которая очищает аккумуляторы при переходе к следующей позиции текста: для каждого аккумулятора S присваиваем $S.Marked = 0$, очищаем $S.PosTable$, присваиваем $Marked = 0$.

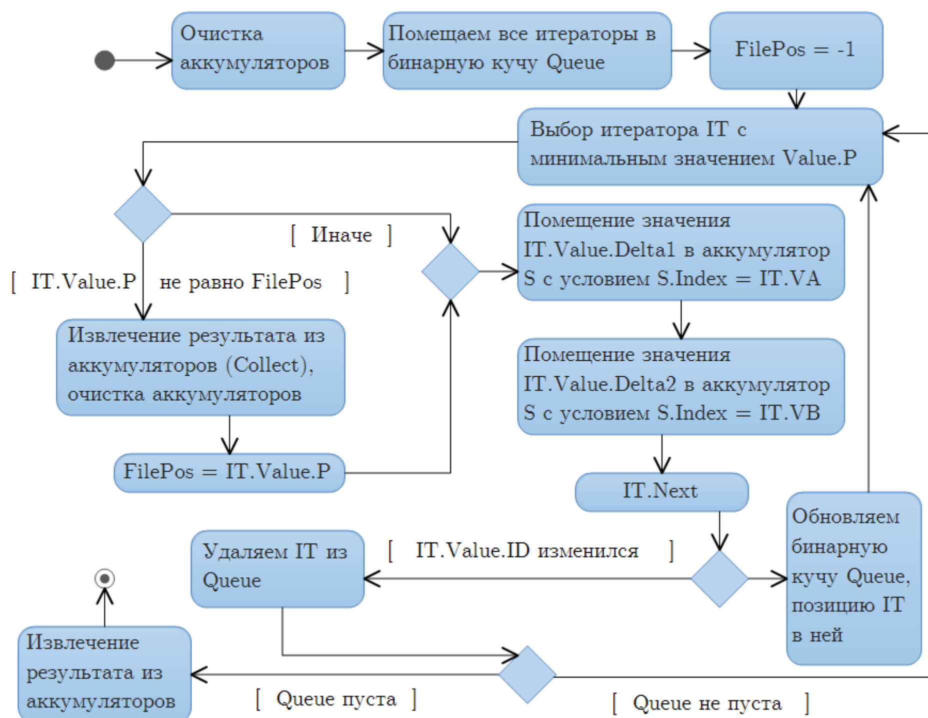

Рис. 2. SearchInDoc(ID)

Вначале вызываем процедуру $InitAccumulators$. Формируем бинарную кучу [14] $Queue$. Для каждого итератора указатель на него помещаем в $Queue$. В качестве операции «$A < B$» для этой бинарной кучи используем $A.Value.P < B.Value.P$. Вершина кучи соответствует итератору с минимальным значением $Value.P$.

Присваиваем $FilePos = -1$. Переменная $FilePos$ соответствует позиции в документе первого компонента ключа индекса, которую сейчас обрабатываем.

Выполняем в цикле, пока $Queue$ не пуста:

1) Выбираем итератор IT с минимальным значением $Value.P$.
2) Если $IT.Value.P$ не равно $FilePos$, то выполняем следующие подшаги (2.a — 2.b):

   a) Запускаем процедуру сборки результата $Collect$.
   b) Вызываем $InitAccumulators$. Присваиваем: $FilePos = IT.Value.P$. Идем на шаг 3.

3) Берем аккумулятор S с индексом $S.Index = IT.VA$, обозначим $Delta = IT.Value.Delta1$. Выполняем один из следующих подшагов (3.a — 3.c):
   a) Если в $S.PosTable$ значение бита, соответствующего $Delta$, равно 1, переходим к следующему пункту 4.
   b) Иначе, если $S.Marked < S.Count$, помещаем в $S.Positions[S.Marked]$ значение $Delta$, увеличиваем $S.Marked$ на 1, увеличиваем $Marked$ на 1. Бит $Delta$ в $S.PosTable$ присваиваем в 1. Переходим к следующему пункту 4.

c) Иначе, обозначим $DeltaAbs = |Delta|$. Бит $Delta$ в $S.PosTable$ присваиваем в 1. Перебираем последовательно значения в $S.Positions$, если какое-то значение по модулю превышает $DeltaAbs$, то заменяем его на $Delta$ и переходим к следующему пункту 4 (т. е. находим и обрабатываем только первое такое значение). Цель — хранение в $S.Positions$ позиций лемм, наиболее близких к $FilePos$.

4) Берем аккумулятор S с индексом $S.Index = IT.VB$, $Delta = IT.Value.Delta2$. Действуем как в пункте 3 (в пункте 3 обрабатываем $IT.VA$, в пункте 4 обрабатываем $IT.VB$).

5) Вызываем $IT.Next$. Выполняем далее одно из трех действий (5.a — 5.c):
   a) Если все записи IT прочитаны, удаляем IT из $Queue$.
   b) Иначе, если $IT.Value.ID$ не равно ID, значит, мы перешли к следующему документу в этом итераторе IT, удаляем IT из $Queue$.
   c) Иначе, обновляем бинарную кучу $Queue$, с учетом возможного изменения порядка элементов после изменения $IT.Value$ на шаге 5 (чтобы вершиной кучи стал итератор, с минимальным значением $Value.P$).

После выхода из цикла еще раз вызываем процедуру сборки результата $Collect$.

4.7. Процедура сборки результата $Collect$

Если $(V.Count - 1)$ не равно $Marked$ — выход из процедуры. Какая-то лемма запроса не найдена на текущей позиции.

Перебираем все аккумуляторы позиций и значения в массивах $Positions$. Вычисляем минимальное $DeltaMin$ и максимальное $DeltaMax$ значения. Добавляем в список результатов запись: $(ID, FilePos - DeltaMin, FilePos + DeltaMax + 1)$ — в документе ID обнаружен поисковый запрос V, позиция первого слова фрагмента текста, содержащего V, равна $FilePos - DeltaMin$, позиция последнего слова фрагмента равна $FilePos + DeltaMax$.

5. Вычислительные эксперименты

5.1. Методика проведения эксперимента

Для экспериментов использовались те же тексты, что и в [13]. Проиндексировано 195 тыс. файлов, объем текста 71,5 Гб, файлы представляли собой обычный текст, однобайтовая кодировка, по стилю, художественная литература, в основном русский язык. Запросы также состоят из слов русского языка.

Для экспериментов поиска случайным образом выбран один из документов коллекции, из слов которого будем формировать тестовые запросы. В эксперименте поиска выберем из документа набор запросов поиска. Далее произведем поиск каждого выбранного поискового запроса. При поиске для каждого используемого ключа читаем в индексе все его словопозиции (т. е. если даже нашли искомый набор слов, все равно читаем все данные в индексе до конца). Далее опишем формирование запросов поиска на заданной позиции.

Параметры: $Step$ — длина пропуска, $Count$ — количество пропусков, Max — максимальное число слов запроса. Если мы находимся на определенной позиции $FilePos$ в тексте, то поисковый запрос выбирается из текста следующим образом:

1) Включаем в запрос слово, которое располагается на позиции $P = FilePos$.
2) Если выбранное число слов меньше или равно $Count$, то пропускаем $Step$ слов $(P = P + Step + 1)$, иначе переходим на одно слово вперед $(P = P + 1)$.

3) Включаем в запрос текущее слово. Если в запросе менее Max слов, идем на шаг 2. Например ($Step = 1, Count = 2, Max = 3$) — берем в тексте текущее слово, пропускаем одно слово, затем берем следующее слово, пропускаем одно слово, берем еще одно слово. ($Step = 2, Count = 1, Max = 3$) — берем в тексте первое слово, пропускаем два слова, далее берем два слова подряд.

Задавая $Step > 0$ проверяем, что работает не только точный поиск, но и поиск с учетом расстояния. То есть, если $Step = 2$, $Count = 1$, то слова поискового запроса в тексте имеют между собой «вставку» из двух «посторонних» слов. Для выбора запросов были использованы следующие параметры: $Par1 = (Step = 0, Count = 0, Max = 3, 4, 5)$ — соответствует точному поиску фраз, длины от 3-х до 5-и, $Par2 = (Step = 1, Count = 1, Max = 3, 4)$, $Par3 = (Step = 1, Count = 2, Max = 3)$, $Par4 = (Step = 2, Count = 1, Max = 3)$.

Для каждого запроса применяется дополнительная фильтрация в соответствии с критерием фильтрации. Позиция выбора запроса $FilePos$ вначале совпадает с началом файла (первое слово файла). На этой позиции формируем 7 запросов, используя настройки $Par1$, $Par2$, $Par3$, $Par4$. Запросы, удовлетворяющие критерию фильтрации, добавляем в набор запросов. Затем увеличиваем $FilePos$ на 1. Количество обрабатываемых позиций определяется параметром $MaxSearch$.

Сформированный набор запросов применяем к разным видам индекса. Основной измеряемый параметр, от которого зависит скорость поиска: количество прочитанных словопозиций при выполнении одного поискового запроса. Выполняя запросы, измеряем среднее значение этого параметра. Для того чтобы получить усредненные результаты каждый запрос выполняем три раза. Преимущество подхода в том, что проверяется: индекс построен корректно. Так как выбираем запросы из уже проиндексированного документа, то мы их точно должны найти. Проверяем, что в результатах поиска присутствует запись, соответствующая тому документу, из которого выбрали запрос, и месту в нем. Получаемые поисковые запросы разнообразны и включают в себя большое количество различных слов.

5.2. Результаты и их обсуждение

Виды индексов для экспериментов поиска:

Idx1. Обычный индекс. Содержит для каждой леммы, включая стоп леммы, список всех ее словопозиций. Данный индекс представляет собой обычный инвертированный файл.

Idx2. Индекс трехкомпонентных ключей. Стоп лемм 700 (524 ru, 176 en), $MaxDistance = 5$. Размер файлов словопозиций трехкомпонентного индекса, 622 Гб. Это существенно превышает размеры индекса последовательностей стоп лемм из [13], 95,4 Гб.

Критерий фильтрации: пропускаем запрос, если в нем есть хотя бы одно слово, имеющее не стоп лемму. Все запросы должны состоять только из стоп лемм. $MaxSearch = 500$.

Для экспериментов поиска использовалось оборудование: CPU — Intel(R) Core(TM) i7 CPU 920 @ 2,67 GHz, HDD — Seagate 7200 RPM, оперативная память — 24 Гб. Выполнено 975 запросов. Выполнение запросов осуществлялось в одном программном потоке.

Среднее число обработанных словопозиций на один запрос поиска: Idx1: 193 млн., Idx2: 756 тыс. Среднее время выполнения поискового запроса: Idx1: 31,27 с, Idx2: 0,33 с. Средний объем прочитанных данных на один запрос: Idx1: 745 Мб, Idx2: 8,45 Мб.

Полное время эксперимента: Idx1: 8 час, 59 мин, Idx2: 6 мин, 42 с.

При поиске в Idx2 среднее время выполнения запроса в 94,7 раза меньше, чем в Idx1.

Среднее число обработанных словопозиций на один запрос при поиске в Idx2 меньше в 255 раз, по сравнению с Idx1. Средний объем прочитанных данных на один запрос при поиске в Idx2 меньше в 88 раз, по сравнению с Idx1. Соотношение меньше, чем соотношение для числа словопозиций, потому что словопозиция в индексе трехкомпонентных ключей содержит дополнительные данные (расстояния между компонентами ключа в тексте).

На рис. 3 отображено среднее время поиска для Idx1 и Idx2 (слева), и средний объем прочитанных данных на один запрос (справа).

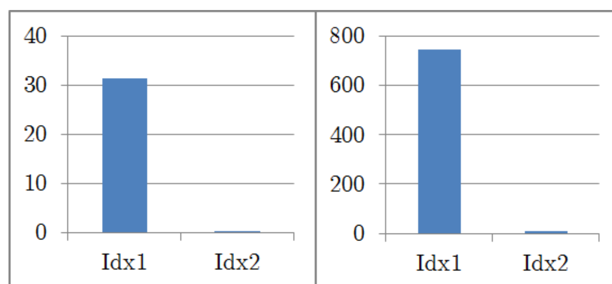

Рис. 3. Среднее время поиска (с) и средний объем прочитанных данных на запрос (Мб)

Заключение

Поисковые запросы, содержащие часто встречающиеся слова, могут выполняться долго, в этом заключается решаемая проблема. Применяя дополнительные индексы, время выполнения запроса можно существенно сократить. Для ускорения выполнения запросов, состоящих из самых часто встречающихся слов, разработаны индексы с трехкомпонентным ключом. По результатам экспериментов, время выполнения таких запросов сокращается в 94,7 раза. Среднее время выполнения таких запросов с использованием обычного индекса, 31,27 секунды, неприемлемо для пользователя, так как превышает 2 секунды (эта временная граница задана исходя из [1]). При этом объем текстовой коллекции 71,5 Гб не является чрезмерным. Среднее время поиска при использовании дополнительных индексов 0,33 секунды является удовлетворительным. В этом смысле говорим, что рассмотренный подход позволяет добиться гарантированного уровня производительности.

В качестве итогового вывода отметим, что с решением текущей задачи получаем методологию поиска, позволяющую решать задачу поиска с учетом близости, для запросов, состоящих из любых лемм, более эффективно, чем с помощью обычных индексов. Если запрос состоит из самых часто встречающихся слов (стоп слова), то применяется новый алгоритм, а для запросов любого другого вида задача уже решена в [12, 13].

Разработанный метод позволяет выполнять более широкий класс запросов, чем [6, 7], где дополнительные индексы применяются только для поиска фраз. Метод обеспечивает больший уровень производительности, чем другие подходы. Например, в [11], таблица 5-2, показано максимальное ускорение выполнения запроса, в 5 раз (в этой таблице даны времена выполнения поисковых запросов для разных видов индекса, максимальное, без оптимизаций, 158 мс, минимальное, со всеми оптимизациями, 32 мс). Результаты в [15] выглядят многообещающими, но в этой работе авторы исключают стоп слова из поиска и индекса. Поэтому возникают сомнения о применимости метода [15] при обработке запросов, включающих часто встречающиеся слова.

Параметр $MaxDistance$ выбран равным 5. Есть резерв для увеличения данного параметра. Если при значении 5, имеем ускорение в 94,7 раза по времени при обработке запросов, состоящих из стоп лемм, то можем увеличить $MaxDistance$, при потере ускорения, скажем, до 50 раз, в этом случае. Определение оптимального, с точки зрения качества поиска, значения параметра $MaxDistance$ – одно из направлений дальнейших исследований. При формировании ключа (f, s, t) используем условие $f \leq s \leq t$, что соответствует текущей реализации. Можно наоборот рассмотреть все ключи с условием $f \geq s \geq t$, при построении индекса, в этом случае поиск может быть более эффективным.

Литература

1. Miller R.B. Response Time in Man-Computer Conversational Transactions // Proceedings: AFIPS Fall Joint Computer Conference. San Francisco, California, December 09–11, 1968. Vol. 33. P. 267–277. DOI: 10.1145/1476589.1476628.

2. Zobel J., Moffat A. Inverted Files for Text Search Engines // ACM Computing Surveys, 2006. Vol. 38, No. 2. Article 6. DOI: 10.1145/1132956.1132959.

3. Tomasic A., Garcia-Molina H., Shoens K. Incremental Updates of Inverted Lists for Text Document Retrieval // SIGMOD '94 Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. Minneapolis, Minnesota, May 24–27 1994. P. 289–300. DOI: 10.1145/191839.191896.

4. Brown E.W., Callan J.P., Croft W.B. Fast Incremental Indexing for Full-Text Information Retrieval // VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases. Santiago de Chile, Chile, September 12–15, 1994. P. 192–202.

5. Zipf G. Relative Frequency as a Determinant of Phonetic Change // Harvard Studies in Classical Philology. 1929. Vol. 40. P. 1–95. DOI: 10.2307/408772.

6. Williams H.E., Zobel J., Bahle D. Fast Phrase Querying with Combined Indexes // ACM Transactions on Information Systems (TOIS). 2004. Vol. 22, No. 4. P. 573–594. DOI: 10.1145/1028099.1028102.

7. Bahle D., Williams H.E., Zobel J. Efficient Phrase Querying with an Auxiliary Index // SIGIR '02 Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Tampere, Finland, August 11–15, 2002, P. 215–221. DOI: 10.1145/564376.564415.

8. Anh V.N., Moffat A. Pruned Query Evaluation Using Pre-computed Impacts // SIGIR '06 Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Seattle, Washington, USA, August 06–11, 2006. P. 372–379. ACM Press. DOI: 10.1145/1148170.1148235.

9. Anh V.N., de Kretser O., Moffat A. Vector-Space Ranking with Effective Early Termination // SIGIR '01 Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. New Orleans, Louisiana, USA, September 9–12, 2001. P. 35–42. DOI: 10.1145/383952.383957.

10. Garcia S, Williams H.E., Cannane A. Access-Ordered Indexes // ACSC '04 Proceedings of the 27th Australasian Conference on Computer Science. Dunedin, New Zealand, January 18–22, 2004. P. 7–14.

11. Yan H., Shi S., Zhang F., Suel T., Wen J.-R. Efficient Term Proximity Search with Term-Pair Indexes // CIKM '10 Proceedings of the 19th ACM International Conference on Information and Knowledge Management. Toronto, ON, Canada, October 26–30, 2010. P. 1229–1238. DOI: 10.1145/1871437.1871593.

12. Веретенников А.Б. Использование дополнительных индексов для более быстрого полнотекстового поиска фраз, включающих часто встречающиеся слова // Системы управления и информационные технологии. 2013. № 2(52). С. 61–66.

13. Веретенников А.Б. Эффективный полнотекстовый поиск с использованием дополнительных индексов часто встречающихся слов // Системы управления и информационные технологии. 2016. № 4(66). С. 52–60.

14. Williams J.W.J. Algorithm 232 — Heapsort // Communications of the ACM. 1964. Vol. 7, No. 6. P. 347–348.

15. Schenkel R., Broschart A., Hwang S., Theobald M., Weikum G. Efficient Text Proximity Search // String Processing and Information Retrieval. 14th International Symposium. SPIRE 2007. Lecture Notes in Computer Science. Santiago de Chile, Chile, October 29–31, 2007. Vol. 4726. Springer, Berlin, Heidelberg. P. 287–299. DOI: 10.1007/978-3-540-75530-2_26.

Веретенников Александр Борисович, к.ф.-м.н., кафедра вычислительной математики и компьютерных наук, Уральский федеральный университет (Екатеринбург, Российская Федерация)

DOI: 10.14529/cmse180105

PROXIMITY FULL-TEXT SEARCH WITH RESPONSE TIME GUARANTEE BY MEANS OF THREE COMPONENT KEYS

© 2018 A.B. Veretennikov

Ural Federal University (pr. Lenina 51, Yekaterinburg, 620083 Russia)

E-mail: alexander@veretennikov.ru

Received: 28.11.2017

Searches for phrases and word sets in large text arrays by means of additional indexes are considered. A search result is a list of documents that contain specified words. A document which contains the query words near each other is more important. Such a tack required to store one posting per any word occurrence in a document. Some search systems use a list of stop words and exclude any information about a stop word from the index thus reducing search quality. In our paper we store information about all words to ensure search quality and build additional indexes for most frequently used words. Use of the additional indexes may reduce the query processing time by an order of magnitude and more in comparison with standard indexes. A new three component key based index has described. Results of search experiments are given and new search algorithm is provided. The results of the experiments shows 90 times improvement of search time for a class of queries containing most frequently used words in comparison with default inverted file.

Keywords: full-text search, search engines, inverted files, additional indexes, proximity search.

FOR CITATION

Veretennikov A.B. Proximity full-text search with response time guarantee by means of three component keys. *Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering.* 2018. vol. 7, no. 1. pp. 60–77. (in Russian) DOI: 10.14529/cmse180105.

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References

1. Miller R.B. Response Time in Man-Computer Conversational Transactions // Proceedings: AFIPS Fall Joint Computer Conference. San Francisco, California, December 09–11, 1968. vol. 33. pp. 267–277. DOI: 10.1145/1476589.1476628.

2. Zobel J., Moffat A. Inverted Files for Text Search Engines // ACM Computing Surveys, 2006, 38(2), Article 6. DOI: 10.1145/1132956.1132959.

3. Tomasic A., Garcia-Molina H., Shoens K. Incremental Updates of Inverted Lists for Text Document Retrieval // SIGMOD '94 Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. Minneapolis, Minnesota, May 24–27, 1994. pp. 289–300. DOI: 10.1145/191839.191896.

4. Brown E.W., Callan J.P., Croft W.B. Fast Incremental Indexing for Full-Text Information Retrieval // VLDB '94 Proceedings of the 20th International Conference on Very Large Data Bases. Santiago de Chile, Chile, September 12—15, 1994. pp. 192–202.

5. Zipf G. Relative Frequency as a Determinant of Phonetic Change // Harvard Studies in Classical Philology. 1929. vol. 40. pp. 1–95. DOI: 10.2307/408772.

6. Williams H.E., Zobel J., Bahle D. Fast Phrase Querying with Combined Indexes // ACM Transactions on Information Systems (TOIS). 2004. vol. 22, no. 4. pp. 573–594. DOI: 10.1145/1028099.1028102.

7. Bahle D., Williams H.E., Zobel J. Efficient Phrase Querying with an Auxiliary Index // SIGIR '02 Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Tampere, Finland, August 11–15, 2002, pp. 215–221. DOI: 10.1145/564376.564415.

8. Anh V.N., Moffat A. Pruned Query Evaluation Using Pre-computed Impacts // SIGIR '06 Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Seattle, Washington, USA, August 06–11, 2006. pp. 372–379. ACM Press. DOI: 10.1145/1148170.1148235.

9. Anh V.N., de Kretser O., Moffat A. Vector-Space Ranking with Effective Early Termination // SIGIR '01 Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. New Orleans, Louisiana, USA, September 9–12, 2001. pp. 35–42. DOI: 10.1145/383952.383957.

10. Garcia S, Williams H.E., Cannane A. Access-Ordered Indexes // ACSC '04 Proceedings of the 27th Australasian Conference on Computer Science. Dunedin, New Zealand, January 18–22, 2004. pp. 7–14.

11. Yan H., Shi S., Zhang F., Suel T., Wen J.-R. Efficient Term Proximity Search with Term-Pair Indexes // CIKM '10 Proceedings of the 19th ACM International Conference on Information and Knowledge Management. Toronto, ON, Canada, October 26–30, 2010. pp. 1229–1238. DOI: 10.1145/1871437.1871593.

12. Veretennikov A.B. Using Additional Indexes for Fast Full-Text Searching Phrases that Contains Frequently Used Words // *Sistemy upravleniya i informatsionnye tekhnologii* [Control Systems and Information Technologies]. 2013. vol. 52, no. 2. pp. 61–66.

13. Veretennikov A.B. Efficient Full-Text Search by Means of Additional Indexes of Frequently Used Words // *Sistemy upravleniya i informatsionnye tekhnologii* [Control Systems and Information Technologies]. 2016. vol. 66, no. 4. pp. 52–60.

14. Williams J.W.J. Algorithm 232 — Heapsort // Communications of the ACM. 1964. vol. 7, no. 6. pp. 347–348.

15. Schenkel R., Broschart A., Hwang S., Theobald M., Weikum G. Efficient Text Proximity Search // String Processing and Information Retrieval. 14th International Symposium. SPIRE 2007. Lecture Notes in Computer Science. Santiago de Chile, Chile, October 29–31, 2007. vol. 4726. Springer, Berlin, Heidelberg. pp. 287–299. DOI: 10.1007/978-3-540-75530-2_26.