О применении дополнительных индексов часто встречающихся слов для полнотекстового поиска

© А. Б. Веретенников

Уральский федеральный университет,
Екатеринбург
alexander@veretennikov.ru, AlexanderBorisovich@urfu.ru

Аннотация

Разные слова в текстах встречаются с разной частотой. Рассматриваются дополнительные индексы, предназначенные для ускорения выполнения поискового запроса, который включает в себя часто встречающиеся слова. Вводится несколько групп слов, для которых разработаны разные методы. Рассмотрен вопрос оптимизации записи индекса.

1 Введение

Рассматривается задача полнотекстового поиска, то есть поиска фраз или наборов слов в текстах на естественном языке, например, русском или английском. Изучаем задачу поиска с учетом расстояния, в документах искомые слова должны располагаться как можно ближе друг к другу. Эта задача требует сохранения в индексе информации о каждом вхождении в документе каждого слова.

Слова в текстах встречаются с разной частотой. Считается, что распределение частот слов соответствует закону Ципфа [8]. Слова, которые часто используются, могут встречаться в тысячи и более раз чаще, чем редко используемые слова.

Скорость выполнения поискового запроса определяется наиболее часто встречающимися словами, входящими в него.

Разделим слова на группы, на основании их частоты встречаемости. Для разных групп слов определим разные методы их обработки, например, создание дополнительных индексов.

В [10] мы определили три группы слов:

1. «Стоп слова»: и, в, или. Встречаются очень часто и могут вообще не включаться в индекс, поэтому их и можно называть стоп словами. Например, предлоги. Далее будем назвать данные слова стоп словами, даже если в каком-то виде включаем информацию о них в индекс.
2. Часто используемые слова. Встречаются часто, но несут в себе существенный смысл и должны включаться в индекс.
3. Остальные, будем называть их «обычные слова».

Далее учитываем все виды слов при поиске, то есть, для любого слова информация в каком-то виде включается в индекс. В поисковом запросе учитываются все слова.

2 Морфологический анализатор

Морфологический анализатор для каждой словоформы возвращает список номеров базовых форм, где номер это число в промежутке [0, *WordsCount* – 1], где *WordsCount* – число базовых форм в словаре, около 260 тыс. для используемого словаря, описывающего русские и английские слова.

Базовые формы слова также называются леммами, а сам процесс получения набора лемм по словоформе – лемматизацией

С учетом морфологического анализа разделение слов на три группы при использовании анализатора применяется не к исходным словоформам, а к леммам. Соответственно, есть три типа лемм в смысле частоты встречаемости: стоп леммы, часто используемые леммы и остальные.

Если мы упорядочим леммы по частоте встречаемости, по убыванию, то вначале идет группа стоп лемм, затем группа часто используемых лемм, затем остальные леммы. Размеры групп определяются параметрами, в зависимости от задачи и способа создания дополнительных индексов.

3 Инвертированные индексы

Для решения задач полнотекстового поиска применяются инвертированные файлы [4] и их аналоги. Индекс состоит из двух частей.

Инвертированный индекс представляет собой набор записей вида (*ID*, *P*), *ID* – идентификатор документа, *P* – позиция, например порядковый номер, слова в документе. Запись (*ID*, *P*) будем называть записью о вхождении слова в документе или словопозицией. Все записи, соответствующие одной лемме, хранятся последовательно для их быстрого чтения при поиске. Идентификатор

Труды XVIII Международной конференции DAMDID/RCDL'2016 «Аналитика и управление данными в областях с интенсивным использованием данных», Ершово, 11-14 октября 2016

документа *ID* будем считать целым числом, например, номером документа.

Таблица дескрипторов. Содержит для каждой леммы структуру, дескриптор, которая содержит информацию о том, где в инвертированном индексе находятся данные для этой леммы.

Индекс это ассоциативный массив, в качестве ключа может выступать, например, лемма, в качестве значения – список словопозиций.

4 Поиск фраз в текстах

4.1 Существующие методы

В [1, 7] приводятся алгоритмы создания дополнительных индексов для более быстрого поиска фраз. Авторы предлагают две оптимизации:

1. Вводится частичный индекс фраз, в котором ключами являются часто используемые фразы (набор которых определяется часто используемыми запросами пользователей).
2. Вводится индекс пар слов (nextword – индекс). Для пары слов хранится информация об их вхождениях в текстах. При этом фиксируется порядок слов в паре и слова находятся непосредственно рядом друг с другом.

В [7] дается обоснование необходимости учета стоп слов при поиске. Простое устранение стоп слов из индекса и поиска может привести к непредсказуемым результатам, если стоп слово имеет для конкретного случая особый смысл (например, см. пример далее).

При этом на поиск накладываются условия:

1. Учет порядка слов фразы.
2. Отсутствие «лишних» слов в составе фразы в искомых текстах.

То есть ищутся только те тексты, в которых фраза встречается точно так, как она введена пользователем. Например, если в тексте есть фраза «Time and a world by Yes», то при поиске

1. Time and a world Yes,
2. Yes time and a world,

данный текст не будет найден методами [1, 7], по причине наличия «by» в тексте (для обоих запросов) и несовпадении порядка слов (для второго запроса). Примечание. Yes – название группы, «Time and a world» – название произведения.

В [2, 3] дано развитие идей nextword – индекса. Идет переход от пар слов к последовательностям из нескольких слов. Однако указанные недостатки [1, 7] остаются. Таким образом, методы из [1, 2, 3, 7] ограничены в применении.

4.2 Отличие предлагаемых методов от существующих методов поиска фраз

Существенное отличие настоящей работы в том, что мы ищем без учета порядка слов, и поиск осуществляется с учетом того, что в текстах посередине фразы может еще что-то быть. Поэтому мы и говорим о поиске «**наборов слов**» и фраз.

Предлагаемые нами методы предлагают принципиально более качественный уровень решения проблемы и их потенциальная область применения существенно шире. На поисковый запрос не накладывается никаких дополнительных условий. Поиск с использованием дополнительных индексов по своим параметрам идентичен поиску с использованием обычных индексов, но выполняется существенно быстрее. За исключением двух ограничений. Во-первых, слова фразы в искомых текстах должны быть все-таки вблизи, то есть число «лишних» слов между ними должно быть ограничено (допустимое количество настраивается параметром). Во вторых, для запросов, которые состоят *только* из стоп лемм, не допускаем в тексте лишних слов во фразе, но ищем без учета порядка.

5 Обработка стоп слов

5.1 Обработка пар стоп лемм и стоп лемм, располагающихся радом с другим словом

В [10] рассмотрен простой способ обработки стоп лемм. Во первых, создается дополнительный индекс, ключ в котором – пара стоп лемм, для которого в индексе хранятся позиции в документах, где эти стоп леммы находятся непосредственно рядом.

Во вторых, если мы имеем словопозицию некоторого слова, то в ее состав включается информация о находящихся непосредственно рядом с этим словом стоп леммах. Алгоритм кодирования описан в [10].

Данный подход позволяет ускорять поисковые запросы, в которых рядом с обычными словами находятся стоп леммы, а также поисковые запросы, включающие в себя нескольких стоп лемм.

5.2 Обработка фраз, состоящих из стоп слов

Если мы создаем индекс для пар стоп лемм, а в поисковом запросе их больше, то использовать пары неудобно. К примеру, если запрос включает в себя n стоп лемм, то может потребоваться обработать порядка $n*n$ пар стоп лемм.

В [11] рассмотрен индекс, ключ в котором определяется набором стоп лемм. Если в тексте встречается последовательность идущих друг за другом стоп лемм некоторой длины (заданы ограничения снизу и сверху, на длину последовательности), то формируется ключ, на основании этого набора стоп лемм. Словопозиция, соответствующая первой из них, включается в индекс. Набор лемм при создании ключа сортируется, поэтому поиск осуществляется без учета порядка.

Это позволяет ускорить поиск фраз, состоящих только из стоп лемм, длиной более 2-х слов.

Кроме того, для словопозиции произвольного слова рассмотрен вариант хранения в основном индексе информации о леммах стоп слов, которые

находятся на расстоянии от данного слова не более чем *MaxDistance* (параметр, например, 5).

Такие подходы имеют смысл исходя из:

1. Вследствие того, что стоп слова очень часто встречаются, практически для каждой их комбинации есть вхождение в текстах.
2. Существенная часть запросов, в которые входят стоп слова, это составные термины или часто используемые фразы, в которых состав слов зафиксирован.
3. Один из видов запросов, включающих стоп слова, это фрагмент текста, который получен путем копирования фразы из уже существующего текста, для поиска других документов, включающих подобную фразу.

6 Дополнительные индексы часто используемых слов

В [10, 11] расширенный индекс (*w*,*v*) это список вхождений слова *w*, когда в тексте не более чем на расстоянии *ProcessingDistance* от *w* присутствовало слово *v*. С учетом морфологического анализа под *w* и *v* мы понимаем леммы слов. Лемма *w* является часто используемой, лемма *v* – произвольная.

Параметр *ProcessingDistance* может быть задан различным в зависимости от частоты встречаемости леммы *w* в текстах. В экспериментах использовалось значение *ProcessingDistance* от 5 до 7.

Мы считаем, если расстояние между словами меньше или равно *ProcessingDistance*, то слова связаны по смыслу друг с другом, иначе нет.

В [11] описаны примеры применения дополнительных индексов для стоп базовых форм и часто используемых базовых форм для выполнения запросов, включающих разные комбинации разных видов слов.

В [11] также представлен алгоритм создания индекса стоп слов и результаты экспериментов, показывающие, что применение дополнительных индексов позволяет значительно (более чем в 10 раз) ускорить выполнение ряда запросов.

В [12] представлен алгоритм создания расширенных индексов и результаты экспериментов построения таких индексов.

7 Примеры выполнения поискового запроса

Рассмотрим запрос: "every stick has two ends".

В данном запросе stick, end, every – часто используемые слова, остальные слова – стоп слова. Каждое слово имеет одну базовую форму.

Для выполнения запроса с использованием дополнительных индексов требуется прочитать три списка словопозиций:

1. stick – обычный индекс,
2. (stick, end) – расширенный индекс,
3. (stick, every) – расширенный индекс,

При этом информация о стоп леммах has, two будет извлекаться из потока словопозиций stick. Для каждой словопозиции *T* в нем хранится информация о стоп леммах, что находятся в тексте близко от *T* (расстояние не более *MaxDistance,* в словах*)*.

В списке словопозиций (stick, end) элементов значительно меньше, чем в списке словопозиций для леммы (end), за счет чего получаем ускорение при выполнении поискового запроса.

Заметим, что запросы рассматриваются не как фразы, а как наборы слов. К примеру, если в тексте есть «every stick has two ends», то запрос «every stick two ends» также найдет это вхождение.

Рассмотрим запрос: "A thing of beauty is a joy for ever". В данном запросе thing, joy, ever – часто используемые слова, beauty – обычное слово, остальные слова – стоп слова. Каждое слово имеет одну базовую форму.

Для выполнения запроса с использованием дополнительных индексов требуется прочитать четыре списка словопозиций

1. beauty – обычный индекс,
2. (beauty, thing) – расширенный индекс,
3. (beauty, ever) – расширенный индекс,
4. (beauty, joy) – расширенный индекс.

8 Об алгоритме поиска

Мы можем обрабатывать запросы, включающие все виды слов. Запрос это несколько слов. В данном разделе мы рассмотрим одну из подзадач поиска, а именно, обработку поискового запроса, в котором:

1. У слов запроса нет стоп лемм.
2. Хотя бы для одного слова запроса все леммы часто используемые.

Будет предложен не рассмотренный ранее алгоритм поиска. Смысл алгоритма: выберем некое слово *a* в запросе, такое, что все леммы *a* часто используемые, назовем его основным словом запроса. Для каждого другого слова *b* рассмотрим расширенные индексы (*a*, *b*). Расширенный индекс содержит словопозиции слова *a*, когда *b* было близко с *a*. Рассмотрим последовательно каждую позицию слова *a* в текстах, и проверим, находились ли близко с *a* все из требуемых слов запроса.

То есть, если есть словопозиция (*ID1*, *P1*) слова *a*, то для каждого другого слова запроса *b* расширенный индекс (*a*, *b*) должен содержать словопозицию (*ID1*, *P1*). Рассмотрим данный подход с учетом того, что словоформа может иметь несколько лемм.

8.1 Определения

Секцией будем называть объект, который содержит в себе лемму и список словопозиций. Поля секции:

Lemma. Лемма.

Postings. Список словопозиций.

Value. Определяет текущий элемент списка словопозиций. Вначале это первый элемент списка,

затем мы можем последовательно брать следующий элемент списка.

Введем понятие «клетка» поискового запроса. Клеткой будем называть список секций. Поисковый запрос в структурированном виде это список клеток.

Секция является итератором словопозиций, то есть имеет операцию *Next*, возвращающую следующую запись. Реализация итератора включает в себя чтение списка словопозиций из индекса. Вначале итератор соответствует первой записи списка словопозиций. Операция *Next* позволяет перейти к следующей записи.

Клетка в свою очередь является итератором словопозиций. Итераторы секций образуют упорядоченный список. Итератор с минимальным значением текущей словопозиции находится в начале списка и его значение есть значение итератора клетки *Value*. При выполнении *Next* для клетки используем *Next* текущей секции, которая затем перемещается в списке, если ее значение становится больше, чем значение другой секции.

Запрос «двести сорок миль» соответствует трем клеткам: [двести] [сорока, сорок] [миля]. Слово «сорок» имеет две леммы, остальные слова по одной лемме.

Выберем индекс *M*, который соответствует клетке запроса, с минимальной суммарной частотой встречаемости лемм, среди таких клеток, у которых все леммы часто используемые. Для каждой секции *w* клетки *M*, и для каждой секции *v* остальных клеток существует расширенный индекс (*w*, *v*). Этот расширенный индекс определит список словопозиций *Postings* для секции *v*.

Будем считать, что для *v* есть список словопозиций *Postings,* состоящий из записей вида (*P*, *Delta*, *DeltaFlag*), где:

- *P* – позиция (номер) слова *w* в документе,
- *Delta* – расстояние от *w* до *v*,
- *DeltaFlag* – определяет, до или после *w* находится *v* в тексте.

Список *Postings* упорядочен по полю *P*, по возрастанию. В данном списке нет *ID* документа, потому что алгоритм применяется в рамках обработки одного документа. После обработки одного документа списки словопозиций очищаются, и заполняются данными следующего документа, затем поиск повторяется для него.

8.2 Вспомогательные переменные

Введем набор массивов, количество элементов в которых равно *L* = («Число клеток запроса» – 1). Одна ячейка массива соответствует одной клетке запроса, не совпадающей с основной клеткой *M*.

Flags. Битовый массив.

Positions, PositionFlags. Массивы расстояний. Пусть ячейка массива соответствует клетке *T*. В тексте есть лемма *w* клетки *M* и лемма *v* клетки *T*. В ячейке *Positions* хранится расстояние от *v* до *w*, а в *PositionFlags* – до или после *w* находится *v* в тексте.

Lemmas. Каждый элемент массива содержит список идентификаторов лемм клетки запроса.

Дополнительные переменные:

Current. Текущая позиция.

Marked. Счетчик тех клеток запроса *V*, леммы которых были найдены, т. е. выполнялось *V.Value.P* = *Current*.

8.3 Алгоритм

Формируем временную клетку *S*, в которую помещаем все секции клеток, не совпадающих с *M*. Выполняем в цикле:

1. Если *S.Value.P* не равно *Current* то:
   a. Если *Marked* = *L,* то мы нашли искомое. Сохраняем позицию *Current* в результатах поиска.
   b. Очищаем *Flags* (все ячейки теперь равны 0).
   c. Присваиваем *Marked* = 0.
   d. *Current* = *S.Value.P*.
2. Помечаем S.Lemma. Выполняем одно из двух.
   a. Если есть индекс i, такой что Lemmas[i] содержит S.Lemma и Flags[i] = 0, то: меняем Flags[1] на 1, сохраняем S.Value.Delta в Positions[i], PositionFlags[i] = S.Value.DeltaFlag, Marked = Marked + 1.
   b. Если есть индекс i, такой что Lemmas[i] содержит S.Lemma и Flags[i] = 1 и Positions[i] > S.Value.Delta, то: меняем Positions[i] = S.Value.Delta, PositionFlags[i] = S.Value.DeltaFlag.
3. Переход к следующему значению клетки S.

9 Производительность поиска

Эксперименты проведены в соответствии с методикой из [11], но с другими параметрами. Запрос может содержать все виды лемм. Всего 4500 запросов, из них 330 – запросы, которые включают только стоп леммы, 462 – запросы, которые не включают стоп лемм.

Стоп лемм 700, часто используемых лемм 2100, русская и английская морфология.

Проиндексировано 72.5 Гб обычного текста (1 символ – 1 байт). Созданы:

1) Обычный индекс, который для каждой леммы включает все ее словопозиции.

2) Обычный индекс, который для каждой леммы, кроме стоп лемм, включает все ее словопозиции, и дополнительно хранит в составе словопозиции информацию о расположенных близко стоп леммах (*MaxDistance* = 5). Индекс последовательностей стоп лемм (длины последовательностей от 2-х до 5-и), используется для запросов, которые включают только стоп леммы. Расширенные индексы, *ProcessingDistance* от 5 до 7.

Среднее число словопозиций, прочитанных при выполнении запроса: 1) 171 млн. 2) 753 тыс.

Результаты показывают: число прочитанных словопозиций при использовании дополнительных индексов снижено существенно.

10 Структура индекса

Существует два варианта организации инвертированного файла.

1. Использование внешней сортировки слиянием. Вначале тексты читаются, формируются словопозиции и записываются в файл инвертированного индекса в порядке их встречаемости в текстах. Затем, с помощью внешней сортировки файл индекса сортируется таким образом, чтобы словопозиции одной леммы располагались подряд.
2. Легко обновляемые индексы. Словопозиции одной леммы хранятся в наборе блоков. При появлении новых словопозиций этой леммы, идет запись в этот набор блоков, при необходимости добавляются дополнительные блоки. Это позволяет избежать внешней сортировки. См., например, [6].

При реализации дополнительных индексов мы используем легко обновляемые индексы. Детали реализации описаны в [13, 14].

Особенность дополнительных индексов для часто используемых слов заключается в том, что объем данных в индексе, то есть, число словопозиций, для конкретного ключа существенно меньше, чем в обычных индексах.

Это приводит к задаче оптимизации построения индекса, для чего разрабатывается новая подсистема ввода-вывода.

Заметим, что теоретически, для создания дополнительных индексов можно использовать и обычные инвертированные индексы, с использованием внешней сортировки. Но этот вопрос выходит за рамки текущей работы.

11 Новая подсистема ввода-вывода

11.1 Логические файлы

Определяем файл, как объект, поддерживающий операции Запись(*O*, *S*, *B*) и Чтение(*O*, *S*, *B*), где *O* – смещение (адрес) относительно начала файла для записи или чтения, *S* – размер блока данных в байтах, *B* – данные.

На основании файлов операционной системы мы можем создавать логические файлы. Мы имеем дерево, в котором узел – это файл, а его потомки – файлы, от которых он зависит, назовем их файлами-компонентами или дочерними файлами. При записи в файл запись осуществляется в файлы более низкого уровня. Рассмотрим модель подсистемы ввода-вывода, которая состоит из четырех уровней.

Приложение
Распределенные файлы и другие виды логических файлов
Модульный маршрутизатор
Операционная система

Уровень приложения – в нашем случае это компонент системы, который осуществляет запись индекса.

Распределенный файл это особый вид логического файла, предназначенный для оптимизации записи в индекс.

Модульный маршрутизатор, это компонент, который при открытии файла по его имени или другим параметрам, определяет, какой вид логического файла нужно создать.

Какие-то части приложения могут напрямую осуществлять запись в файлы операционной системы, а другие части могут использовать распределенные файлы.

11.2. Распределенные файлы

Распределенный файл формируется на базе нескольких файлов. У каждого файла есть поле *TotalSize* – размер файла.

Распределенный файл представляет собой последовательность блоков разного размера. Размер блоков определяется приложением. Конкретный блок храниться в одном из дочерних файлов.

В составе распределенного файла существует таблица. Ее предназначение, преобразование адреса распределенного файла в адрес одного из дочерних файлов. Таблицу реализуем в виде АВЛ дерева [9]. Ключ в таблице – адрес в распределенном файле, значение – запись, определяющая блок данных, назовем ее описателем блока. Поля описателя блока:

1) *Start* – адрес в распределенном файле.
2) *Size* – размер блока.
3) *Position* – адрес в файле-компоненте.
4) *Index* – номер файла-компонента, в котором хранится блок, определяет файл-компонент.
5) *Next*, *Prev* – переменные для организации циклического списка.

Мы предусматриваем 3 вида файлов-компонентов.

1) Файл для малых операций ввода-вывода, ФМО.
2) Файл для обычных операций ввода-вывода, ФОО.
3) Файл метаданных, ФМ.

Блоки данных сохраняются в ФМО или ФОО. Файл-компонент определяется на этапе первой записи конкретного блока, т. е. записи, которая осуществляется в конец распределенного файла.

АВЛ дерево поддерживает операцию *LowerBound*(*V*), которая возвращает описатель *R*, ключ которого *R.Start* <= *V* ("<=" – меньше или равно), и при этом не существует записи *R2*, *R2.Start* <= *V* и *R2.Start* > *R.Start*.

11.3 Процедура записи в распределенный файл

Осуществляем запись блока (*O*, *S*, *B*) в распределенный файл *D*.

Вначале мы определяем, требуется ли записать блок в конец файла, т. е. будет ли это новый блок. В случае, если часть блока должна обновить существующие данные, а часть выходит за пределы существующих данных, блок делится на два блока по границе существующих данных.

Т. е. если $O = D.TotalSize$, значит это новый блок. Иначе, по крайней мере, часть блока, должна обновить существующие данные.

Если запись осуществляется в существующие данные ($O < D.TotalSize$ и $(O + S) <= D.TotalSize$), то по таблице определяем описатели блоков и запись осуществляется в соответствующие им файлы-компоненты.

Если $O < D.TotalSize$ и $(O + S) > D.TotalSize$, то нужно разбить блок на 2 блока, для каждого из которых применяется один из предыдущих случаев.

Если запись осуществляется в конец файла, то мы по размеру блока определяем целевой файл-компонент. Далее мы создаем для блока описатель в таблице и сохраняем данные в конец файла-компонента.

Для файла обычных операций ввода-вывода мы не предусматриваем никакой особой логики.

11.4 Файл для малых операций ввода-вывода

Файл для малых операций разделяется на плоскости. Плоскость – это блок большого размера, например, 8-32 Мб. Размер плоскости фиксирован. Поля записи *Next* и *Prev* описателя предназначены для организации циклического списка. В этом циклическом списке хранятся описатели, соответствующие одной плоскости.

Будем использовать идеи из [5].

Выделим буфер оперативной памяти, размер которого совпадает с размером плоскости. Данный буфер соответствует активной плоскости.

Как мы увидим далее, плоскости могут освобождаться, данные их них перемещаться в другое место. Организован список свободных плоскостей.

При инициализации буфера активной плоскости активная плоскость определяется путем извлечения из списка свободных плоскостей элемента. Если список пуст, то адрес активной плоскости определяется адресом конца файла, то есть мы добавляем в конец файла новую плоскость.

Если мы записываем в файл новый блок, то записываем его в буфер активной плоскости. Записи блоков в буфер осуществляются последовательно.

Если в буфере активной плоскости недостаточно места для записи нового блока, то мы записываем активную плоскость на диск, по ее адресу, и осуществляем повторную инициализацию буфера и выбор новой активной плоскости.

Если мы должны обновить существующий блок, описатель *R*, то возможны два варианта:

1. $S = R.Size$. В этом случае мы осуществляем запись данных в буфер активной плоскости, обновляем в описателе поле *Position*. Описатель теперь соответствует новой плоскости. Удаляем его из циклического списка старой плоскости и помещаем в циклический список новой плоскости.
2. S не равно $R.Size$. В этом случае запись осуществляем в файл-компонент по адресу $R.Position + (O – R.Start)$. То есть оптимизация записи не применяется в случае частичного обновления блока.

При записи новых данных запись будет преимущественно осуществляться в конец файла.

11.5 Ограничения на уровне приложения

Описанная логика предназначена для оптимизации осуществления малых операций записи. При этом на уровень приложения накладываются ограничения:

1. Уровень приложения должен оперировать блоками данных. Блок данных должен иметь логический смысл на уровне приложения. Приложение должно осуществлять атомарную запись блоков. То есть, если есть какой-то блок данных на уровне приложения, то запись его должна осуществляться за одну операцию ввода вывода. Не должно быть такого, что вначале мы пишем часть, например, половину блока, а затем остальную часть.

   Это достаточно серьезное ограничение, так как приложение в обычном случае, если пишет данные блока за несколько операций ввода вывода, но подряд, то это не несет каких-либо негативных последствий. В нашем же случае, если запись осуществляется в распределенный файл, то части блока могут попасть в разные плоскости. Блок будет фрагментирован, что далее, при чтении приведет к падению производительности.
2. Если приложение ранее осуществляло операцию (O, S, B), то повторные операции должны обновлять данный блок также целиком, то есть, если есть повторная операция записи ($O1$, $S1$, $B1$), где $O1 >= O$, $O1 < (O+S)$, то $O1 = O$, $S1 = S$. Это правило может нарушаться, но в случае нарушения оптимизации записи не будет.

11.6 Процесс контроля заполнения плоскости

При записи (O, S, B) в ФМО, при обновлении существующих данных мы можем теперь осуществлять запись данных в конец активной плоскости. Получается, в той плоскости, где ранее были данные блока, появляется неиспользуемое пространство.

Организуем процедуру контроля заполнения плоскостей. В текущей реализации мы определяем переменную, в которой сохраняется общий объем записанных данных. Когда значение переменной

преодолевает заданный порог, запускается процедура контроля свободного места. Перебираем все плоскости, анализируем, насколько они заполнены. Объем плоскости фиксирован. Объем используемых данных определяется суммированием поля *Size* описателей этой плоскости, что можно сделать, так все описатели плоскости в одном циклическом списке. Если плоскость заполнена менее, чем наполовину (задаем параметром необходимый уровень наполненности плоскости), то переносим все данные ее заполненных блоков в активную плоскость. Текущую плоскость объявляем свободной. Описатели перенесенных блоков обновляются.

12 Преимущества новой подсистемы ввода-вывода

12.1 Замена случайного ввода вывода последовательным

Позволяет вместо случайного ввода-вывода, состоящего из малых операций записи осуществлять, в определенных случаях, последовательный ввод вывод.

Недостаток заключается в наличии процедуры контроля свободного места. За счет чего могут возникнуть дополнительные операции ввода-вывода, однако, это тоже будет последовательный ввод-вывод. За счет замены случайного ввода-вывода на последовательный ввод-вывод улучшается производительность.

Операции записи большими блоками и операции чтения не оптимизируются.

12.2 Оптимизация для SSD.

Для SSD массивный случайный ввод-вывод, состоящий из большого числа малых операций записи, может негативно влиять на производительность. Именно такой вид ввода-вывода мы заменяем последовательным.

12.3 Возможность организации многоуровневого хранения для создания индекса

Метод предусматривает возможность размещать файлы для разных видов операций на разных носителях. Например, случайный ввод-вывод можно перенести на SSD, с учетом пункта 12.2.

Заметим, что в настоящее время в системах хранения предусматривается автоматическое многоуровневое хранение данных. Данные, к которым обращаются чаще, перемещаются на более быстрые носители. Однако система хранения не знает об особенностях приложения и за счет ручного управления хранением данных можно повысить производительность.

12.4 Сохранение производительности поиска

При условии выполнения ограничений, накладываемых на уровень приложения, производительность поиска не ухудшится.

12.5 Распараллеливание ввода-вывода

На построенной модели можно организовать хранение индекса на нескольких носителях и распараллеливание ввода-вывода на них.

13 Применение новой модели ввода вывода к построению индекса

13.1 Организация инвертированного индекса

Как было сказано ранее, мы используем способ создания индекса, описанный в [13, 14].

Инвертированный индекс представляет собой файл, который разбит на блоки фиксированного размера – кластеры. Цепочка кластеров или поток – это набор кластеров, в котором хранятся словопозиции. В простейшем случае словопозиции одной леммы хранятся в одном потоке.

Базовая идея заключается в том, что мы создаем один кластер для словопозиций леммы и записываем их в него последовательно. Когда свободное место в кластере заканчивается, создается новый кластер, в текущем кластере прописывается ссылка (номер кластера) на новый кластер.

Необходимо, чтобы блоки одной леммы располагались преимущественно подряд для снижения числа операций ввода-вывода при поиске.

Поэтому, введено понятие блока кластеров, это кластеры, которые идут в файле подряд. Зафиксируем максимальный размер блока кластеров, например, 16 Мб. Рассмотрим блоки кластеров, размером в 1, 2, 4, 8 и т.д. кластеров.

Вначале используется блок, состоящий из одного кластера. Когда свободное место в нем заканчивается, выделяется блок, размером, в два раза больше. Данные из существующего кластера переносятся в половину нового блока. Так делается, до достижения максимального размера блока. После чего мы выделяем уже новый блок, и поток состоит уже из двух блоков. Также есть возможность использования части кластера для одного потока, а другой части для другого.

Нужно, чтобы первоначальная запись блока кластеров осуществлялась атомарно, за одну операцию ввода вывода. Чтобы в распределенном файле был правильно определен целевой файл-компонент и для блока кластеров был создан один описатель.

13.2 Кеш инвертированного индекса

Считаем, что одновременно в памяти мы храним не менее одного кластера для одной леммы. В качестве одного из вариантов выполнения данного условия можно использовать кэш, размер кэша должен быть не меньше, чем число лемм, обрабатываемых на текущий момент, умноженное на размер кластера. Организуем таблицу, ключ в которой – это номер кластера. Значение – структура, содержащая адрес буфера, размером в один кластер,

с данными, и дополнительные поля, например, признак того, что данные изменены.

Поскольку в кэше оперируем единичными кластерами, нужно обеспечить запись данных из кэша в файл таким образом, чтобы кластеры одного блока кластеров сохранялись, например, при выгрузке из кэша, за одну операцию ввода-вывода.

Для этого, в структуру, описывающую кластер в кэше, добавим поле – номер операции ввода вывода.

Введем переменную – счетчик, которая будет увеличиваться на 1 при каждой последующей операции ввода-вывода. Когда осуществляется первоначальная запись блока кластеров, она осуществляется за одну операцию ввода вывода и значение номера операции ввода вывода одинаковое для всех кластеров блока в таблице кэша.

13.3 Учет соединения записей (write coalescing) на уровне распределенного файла

При сбросе кэша на диск, записи отдельных кластеров кэша могут объединяться в одну для оптимизации процесса записи. При этом кластеры могут соответствовать блокам разного вида (ФМО или ФОО). Это нужно учитывать при организации распределенного файла. Если осуществляется запись буфера *B* в распределенный файл и при этом должны быть обновлены несколько блоков распределенного файла, то нужно вначале разделить буфер *B* на части, каждая из которых соответствует одному блоку распределенного файла. Затем для блоков одного вида осуществить обратное соединение записей, на основании частей буфера *B*, которые соответствуют этим блокам, если это возможно.

14 Заключение

Рассмотрены методы создания дополнительных индексов для поиска фраз, включающих часто используемых слова. Эксперименты показывают, что число словопозиций, которые должны быть обработаны при выполнении поискового запроса, может быть снижено более чем в 10 раз за счет применения дополнительных индексов. Дан новый алгоритм поиска для случая запроса, включающего хотя бы одно часто используемое слово, но без стоп слов. Представлена новая подсистема ввода-вывода для оптимизации создания таких индексов.

Литература

[1] Bahle D., Williams H.E., Zobel J.; Efficient Phrase Querying with an Auxiliary Index. In Proc. ACM-SIGIR Conf. on Research and Development in Inform. Retrieval, Finland, 2002, p. 215–221.

[2] Chang M., Chung Keung Poon. Efficient Phrase Querying with Common Phrase Index. ECIR 2006, LNCS 3936, Springer-Verlag Berlin Heidelberg, 2006, p. 61–71.

[3] Shashank Gugnani, Rajendra Kumar Roul. Triple Indexing: An Efficient Technique for Fast Phrase Query Evaluation. International Journal of Computer Applications. Vol 87, No. 13, 2014.

[4] Prywes N. S., Gray H. J; The organization of a Multilist-type associative memory, IEEE Trans. on Communication and Electr., 1963, 68, p. 488–492.

[5] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Trans. Comp. Syst., 10(1), 1992, p. 26–52.

[6] Tomasic A., Garcia-Molina H., Shoens K.; Incremental updates of inverted lists for text document retrieval, In Proc. ACMSIGMOD Int. Conf., Minnesota, 1994, p. 289–300.

[7] Williams H. E., Zobel J., Bahle D.; Fast phrase querying with combined indexes. ACM TOIS. Vol 22, No. 4, 2004, p. 573–594.

[8] George Kingsley Zipf. Relative frequency as a determinant of phonetic change. Harvard Studies in Classical Philology, Vol 40, 1929, p. 1–95.

[9] Адельсон-Вельский Г. М., Ландис Е. М. (1962). Один алгоритм организации информации. Докл. АН. СССР, 146, с. 263-266.

[10] Веретенников А.Б. О поиске фраз и наборов слов в полнотекстовом индексе, Системы управления и информационные технологии, №2.1(48), 2012, с. 125–130.

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

[12] Веретенников А. Б. Создание дополнительных индексов для более быстрого полнотекстового поиска фраз, включающих часто встречающиеся слова. Системы управления и информационные технологии, №1(63), 2016, с. 27–33.

[13] Веретенников А. Б. Создание легко обновляемых текстовых индексов, RCDL'2008. Дубна: ОИЯИ, 2008, с. 149–154.

[14] Веретенников А. Б. Эффективная индексация текстовых документов с использованием CLB-деревьев, Системы управления и информационные технологии, №1.1(35), 2009, с. 134–139.

Using additional indexes of frequently used words for full-text search

Alexander B. Veretennikov

Different words can occur in texts with different frequency. We describe additional indexes, intended for speeding up search in case the search query contains frequently used words. We defined several groups of words and developed different methods for each group. Index writing optimization is given. For the case when search query is a set of frequently used words, a search algorithm is explained.