Санкт-Петербургский государственный университет

Математико-механический факультет

Кафедра системного программирования

Евстифеев Сергей Викторович

«Оптимизация выполнения запросов по диапазонам для библиотек индексации, построенных на основе обратного индекса»

Дипломная работа

Допущен к защите
Зав. кафедрой,
доктором физ.-мат. наук,
профессором Тереховым А.Н.

Научный руководитель:
Разработчик ПО, «Ланит-Терком»
Пименов А. А.

Рецензент:
Доктор физ.-мат. наук, профессор
Новиков Б.А.

Санкт-Петербург

2010

Saint Petersburg State University

Faculty of Mathematics and Mechanics

Department of Software Engineering

Evstifeev Sergey Victorovich

«Range queries optimization for IR libraries based on inverted index»

Graduate Paper

Admitted to proof
Head of the Chair:
Dr. of Phys. and Math. Sci., Professor
Terekhov A.N.

Scientific advisor:
Software Developer, “Lanit-Terkom”,
Alexander A. Pimenov

Reviewer:
Dr. of Sci., Professor, Boris A. Novikov

Saint Petersburg

2010

Оглавление

1. Введение

Поисковые системы, основанные на библиотеках индексации в настоящее время используются практически повсеместно: для осуществления Web-поиска, поиска по архивам документов, а также для других специфических задач. Отличительной чертой таких систем является возможность осуществления поиска по неструктурированным или слабоструктурированным массивам данных, таких как множества разнородных документов в различных форматах. Благодаря отсутствию заранее предопределенной структуры данных и возможности неполного соответствия для поиска по запросу, поисковые системы представляют собой удобное средство, предоставляющее возможности постоянного хранения и быстрого поиска по хранимым документам. Библиотеки индексации позволяют связывать с каждым индексируемым документом набор полей, по которым возможно осуществлять поиск. Простейшими примерами служат поля «имя документа» и «содержимое». Кроме того, в таких полях возможно хранить численные даннные, а также дату и время. Существует множество сценариев использования информации, хранимой в этих полях. К примеру, информация о дате публикации статьи может требоваться для создания списка статей, опубликованных за прошедший месяц, кроме того широкое применение такие данные имеют для семантического поиска и анализа (так как, как правило, имеет смысл не абсолютная величина результатов анализа, а изменение показателей за выбранный временной период). Однако структура обратного индекса не оптимизирована для запросов по диапазонам. Несмотря на то, что на данный момент существует решение, значительно ускоряющее производительность таких запросов, существуют ситуации, в которых этот алгоритм можно улучшить, так как он выбирает заведомо неоптимальный вариант. Задачей работы является исследование возможности и реализация оптимизации существующего алгоритма поиска

по диапазонам для библиотек индексации, построенных на основе обратного индекса.

2. Библиотека индексации Lucene

Lucene представляет собой библиотеку поиска информации, полностью написанную на языке Java. Поиск информации включает в себя поиск документов, поиск информации, содержащийся в документах, а также мета-данных, относящихся к документам. Lucene обладает следующими преимуществами:

1) Полнота функциональности. Все основные необходимые функции осуществления поиска, хранения и создания индекса, а также средства для обновления индекса (что немаловажно) присутствуют в Lucene;
2) Масштабируемость;
3) Открытый исходный код (лицензия Apache 2.0)

Среди альтернатив Lucene в сфере поисковых систем, помимо закрытых продуктов, можно назвать Egothor, однако, судя по всему, этот проект в данный момент либо полностью закрыт, либо заморожен. Таким образом, Lucene является единственной системой, позволяющей реализовать функциональность полнотекстового поиска с использованием свободного программного обеспечения. Кроме того, стоит отметить, что Lucene активно используется в значительных масштабах: более 300 больших компаний и проектов используют Lucene, среди которых такие как: AOL, Apple, Wikipedia, IBM, LinkedIn, Wolfram Research, SourceForge.net и MIT OpenCourseware.

Стоит отметить, что Lucene не является готовой системой Web-поиска. Библиотека включает в себя простой и гибкий Application Programming Interface (API), который позволяет эффективно осуществлять индексирование и поиск.

3. Структура библиотек индексации

Каким образом можно осуществить поиск файлов, содержащих определенное слово или фразу? Тривиальным подходом будет: сканировать все файлы по порядку и проверять их на наличие искомых слов. Однако такой подход не подойдет для больших наборов файлов или случаев, когда отдельные файлы слишком велики. Современные поисковые приложения построены на основе структуры, называемой индексом. Поисковая система сперва сканирует набор документов и конвертирует его в формат, который позволит осуществлять по ним быстрый поиск. Этот процесс называется индексированием, а результирующая структура данных – индексом. Можно рассматривать индекс как структуру данных, позволяющую осуществлять быстрый прямой доступ к отдельным словам, хранимых в ней. В случае Lucene, индекс – специальная структура данных, как правило, хранимая на жестком диске в виде набора файлов индекса.

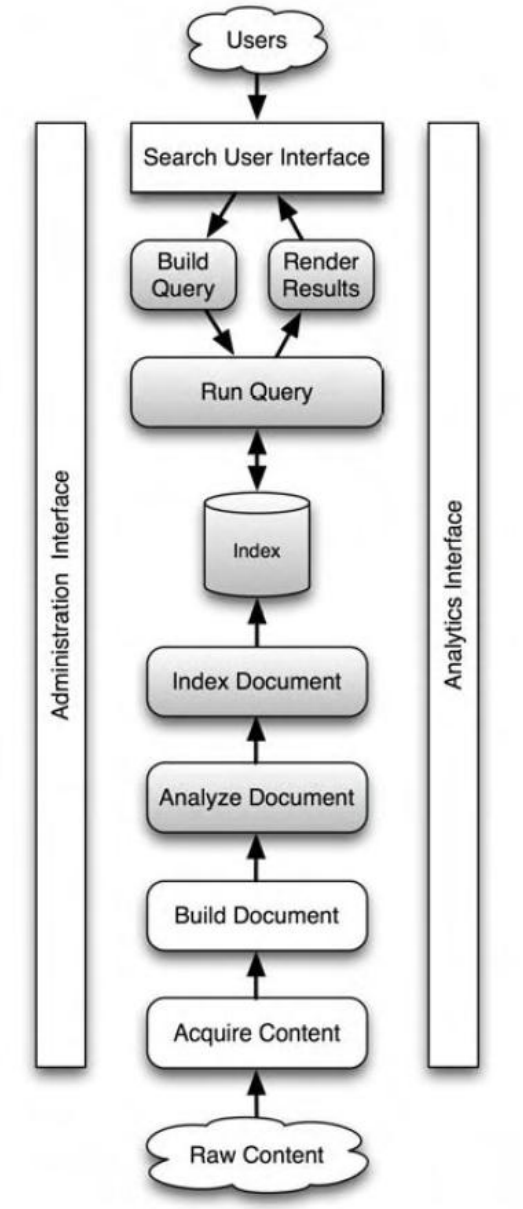

Рисунок 1. Структура поискового приложения, серым цветом выделены участки, за осуществление которых отвечает Lucene

3.1. Получение данных

Рассмотрим более детально пропроцесс индексации. Первым этапом, в соответствии с Рис.1, является получение данных. Эта задача, т.е. сбор файлов для последующего индексирования, может быть как достаточно тривиальной в случае, если хранилищем данных является директория на жестком диске или хорошо структурированная база данных, – так и невероятно сложной в случае, если данные расположены в различных местах и слабо структурированы. Также, значительной проблемой в случае больших

наборов данных является реализация данного этапа таким образом, что повторному сканированию будут подвергнуты только документы, которые были изменены со времени прошедшего запуска. Кроме того, приложение, реализующее этот этап, может быть «живым», то есть постоянно контролировать появление новых или измененных данных и загружать их, как только они становятся доступными.

Стоит отметить, что Lucene, являясь лишь ядром, служащим для создания индекса и осуществления поиска, не включает в себя компоненты для получения данных. Эта задача должна осуществляться самим пользовательским приложением, либо можно использовать стороннее программное обеспечение для реализации этого этапа. Среди доступных приложений с открытым исходным кодом отметим Nutch, Heritrix, Grub, Apache Droids, Aperture.

3.2. Построение внутренних представлений документов

На данном этапе происходит преобразование данных, собранных на предыдущем шаге в отдельные элементы, которые будем условно называть документами. Именно на этом шаге необходимо извлечь текстовую информацию из исходных данных. В случае, если данные были изначально представлены в текстовой форме, это не является проблемой. Однако значительное количество текстовых документов представлено в бинарной форме, например такие форматы как: PDF, Microsoft Office, Open Office и так далее. Еще одним примером класса документов, нуждающихся в специальной обработке, являются документы, построенные с использованием специальной разметки, такие как RDF, XML, HTML. Мета-данные, например, дата, время, расположение, также можно выделить на этом этапе при помощи семантического анализа исходных данных.

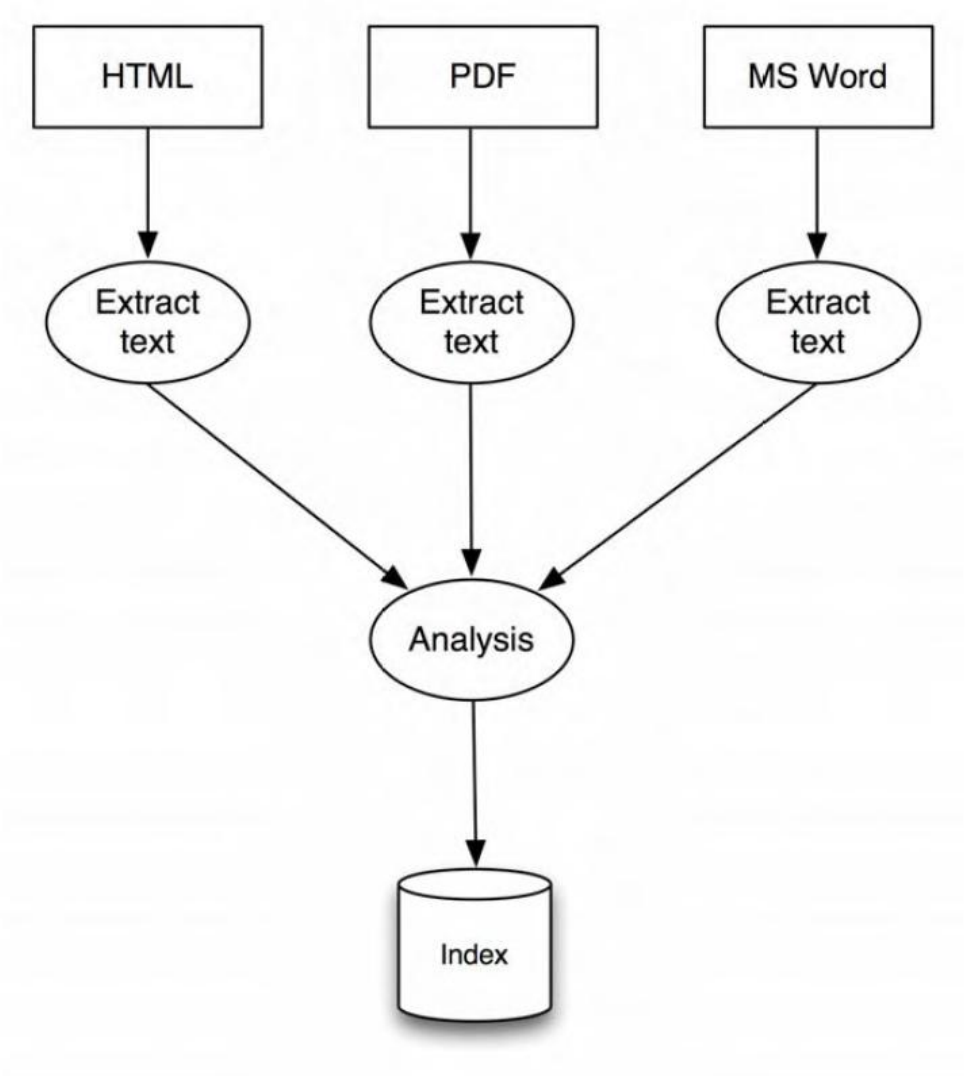

Рисунок 2. Построение документов

Как можно видеть из Рис.1, Lucene не осуществляет построение документов, так как этот шаг должен быть осуществлен ad hoc для каждого приложения, хотя в библиотеку включен соответствующий API. Однако существует проект-спутник Lucene, Tika, который специально создан для реализации описанного этапа.

3.3. Анализ документа

Перед тем, как осуществлять индексацию документа, необходимо разделить текст на отдельные атомарные элементы, которые называются токенами. В первом приближении токен – это просто слово, а на данном

этапе осуществляется разбиение текстовых полей документа на отдельные токены. Именно на этом шаге необходимо определить, как работать с составными словами, нужно ли исправлять орфографические ошибки, как обрабатывать множественное число существительных и текст в верхнем регистре. Для этой задачи Lucene содержит широкий спектр встроенных анализаторов, позволяющих полностью контролировать процесс анализа документа.

3.4. Индексирование документа

На данном шаге текущий документ добавляется в индекс. Lucene использует структуру, называемую «обратный индекс». Эта структура позволяет одновременно и эффективно использовать дисковое пространство, и осуществлять быстрый поиск по ключевым словам. При этом токены используются как центральные объекты индекса, они и являются ключевыми словами, и по ним же осуществляется поиск. Lucene хранит информацию в таком виде, чтобы наиболее удобно было ответить не на вопрос «Какой текст находится в этом документе?», а «Какие документы содержат этот текст?».

3.5. Гибкая схема

В отличие от баз данных, в Lucene отсутствует понятие предопределенной глобальной схемы. Таким образом, каждый очередной документ, добавляемый в индекс, может полностью отличаться по своей структруре от предыдущих: с произвольной структурой полей и содержимого. Такой подход позволяет применять итеративный подход к созданию индекса. Для того, чтобы произвести переиндексацию, не обязательно создавать индекс заново, можно просто начать связывать дополнительные поля с вновь добавляемыми документами, одновременно

производя переиндексацию уже добавленного набора документов. Кроме того, можно добавлять «мета-документы», не появляющиеся в результатах поиска, но содержащие мета-данные об индексе, такие как, к примеру, время последнего обновления индекса.

3.6. Сегментированная структура индекса

Одной из сильных сторон Lucene является возможность инкрементальной индексации, что является одним из требований к современной поисковой библиотеке. В то время как многим поисковым библиотекам для этого требуется переиндексировать весь объем документов, Lucene использует структуру, позволяющую производить регулярное добавление новых документов. При этом как только очередной документ оказывается добавленным в индекс, по нему сразу же можно осуществлять поиск.

В Lucene индекс состоит из одного или нескольких сегментов, каждый из которых сам по себе представляет собой полноценных индекс. Каждый раз при выполнении поиска, сканирование сегментов осуществляется по-отдельности, а результаты объединяются. При каждой новой индексации будет создан новый сегмент, в который попадут вновь добавленные документы. Со временем, сегментов становится много, и необходимо объединить их для оптимизации процесса поиска. Эта процедура осуществляется в соответствии с заданными настройками, задающими необходимые условия для объединения.

4. Сравнение поисковых систем и баз данных

Поисковые системы структурно во многом схожи с системами для работы с базами данных. В обоих случаях документы содержатся в хранилище и поддерживается индекс. Запросы обрабатываются при помощи индекса, и на его основе определяются подходящие значения, которые в конечном итоге возвращаются пользователю. Однако при этом существует и множество отличий. К примеру, базы данных должны иметь возможность обрабатывать сколь угодно сложные запросы, в то время как подавляющее большинство запросов к поисковым системам представляют собой списки термов (так называемые *bag-of-word*) и фразы.

Несмотря на то, что поисковые системы не несут затрат, связанных с такими операциями, как, к примеру, слияние таблиц, существует и множество факторов, препятствующих быстрому ответу на запрос: терм, содержащийся в запросе, может встречаться в большом количестве документов, а каждый документ, в свою очередь, содержит значительное количество термов. Все эти дополнительные сложности привели к созданию множества специальных алгоритмов и структур данных, таких как структура текстовых индексов, алгоритмы построения индекса и алгоритмы обработки текстовых запросов[1]. Среди требований к поисковым системам можно перечислить следующие:

1) Быстрая и эффективная обработка запросов;
2) Использование текстовых функций, таких как схожесть термов для улучшения эффективности;
3) Использование возможностей гиперссылок, таких как anchor-ы, и URL-термы;

[1] [ZM06] Параграф «Introduction»

4) Минимизация использования ресурсов, таких как жесткий диск, основная память и каналы передачи данных;

5) Возможность масштабирования для больших наборов данных;

6) Обработка изменений в хранимом наборе документов;

7) Обеспечение продвинутых функций, таких как бинарное ограничение по термам и запросы по фразам целиком;

Внутренняя структура индекса Lucene спроектирована таким образом, чтобы ускорить наиболее частые запросы к поисковой системе: запросы типа *bag-of-words*. Индекс чаще всего хранится на жестком диске (однако существует возможность создания структуры индекса и в оперативной памяти, что используется для тестирования, а также для ускорения работы в случае большой нагрузки и небольшой величины индекса) в виде набора файлов, каждый из которых содержит информацию об отдельных компонентах индекса. В рамках данной работы интерес представляет так называемый *словарь термов*, представленный двумя файлами:

1) .tis файл, содержащий набор всех хранимых термов, отсортированный в лексикографическом порядке сначала по имени поля, соответствующему терму, а затем по содержимому терма, в соответствии с кодами символов в кодировке UTF-16. Также в этом файле хранится связанная с каждым термом информация, как в текстовом виде, так и в виде указателей на позиции элементов в других файлах индекса;

2) .tii файл, представляющий собой индекс термов, облегчающий поиск по набору термов из предыдущего файла. Файл предназначен для полного считывания в основную память и обеспечения прямого доступа к элементам “tis” файла.

5. Постановка задачи

Поисковые библиотеки, использующие структуру обратного индекса, оптимизированы для выполнения запросов по набору термов. На языке запросов к реляционным базам данных подобные запросы можно записать следующим образом:

```
SELECT [columns]

FROM [table] T

WHERE ${field}='term1' OR ... OR ${field}='termN'
```

Достаточно объединить результаты запросов поиска документов, содержащих каждый из термов. Благодаря использованию обратного индекса, каждый из этих запросов выполняется очень быстро.

Однако при попытке осуществления поиска всех значений поля, лежащих в заданном диапазоне, возникает проблема.

```
SELECT [columns]

FROM [table] T

WHERE ${field} BETWEEN rangeMin AND rangeMax
```

Поисковые библиотеки обрабатывают такой код, преобразуя диапазон в набор всех термов, содержащихся в нем, и заменяя конструкцию “`BETWEEN…AND`“ на набор конструкций “`OR`”. Рассмотрим пример:

```
SELECT [columns]

FROM [table] T

WHERE ${field} BETWEEN 1 AND 10
```

В случае, если поле `${field}` может принимать только целые значения, запрос будет преобразован в следующую конструкцию:

```
SELECT [columns]
FROM [table] T
WHERE (${field} = 1 OR
 ${field} = 2 OR
 ${field} = 3 OR
 ${field} = 4 OR
 ...
 ${field} = 10 OR
)
```

С первого взгляда, такой подход может показаться удобным, однако принимая во внимание то, что каждый из элементов конструкции “OR” порождает отдельный поисковой запрос, можно понять, что время, требующееся на выполнение поиска по диапазону оказывается довольно значительным. Проблема полностью раскрывается, если рассматривать поиск по диапазону, в случае, когда поле `${field}` может принимать вещественные значения. В этом случае, раскрытие даже незначительного по абсолютной величине диапазона порождает невероятно большие конструкции “OR”, при этом количество термов в конструкции прямо пропорционально количеству значащих битов того типа данных, к которому принадлежит поле. Такая дискретизация диапазона приводит к тому, что выполнение запросов по диапазонам занимает значительное время и очень интенсивно использует ресурсы системы. В связи с этим, с точки зрения архитетектуры промышленных поисковых систем имеет смысл не включать запросы по диапазонам в список доступных широкому кругу пользователей поисковых конструкций. Для выполнения подобных запросов можно перевести имеющиеся документы в базу данных – в этом случае запросы по

диапазонам будут осуществляться эффективно. Однако, существует ряд ситуаций, когда использование поисковой библиотеки предпочтительнее. Одной из значительных причин может являться гибкость схемы данных, описанная ранее в работе. В то же время, запросы по диапазонам могут являться абсолютно необходимой функцией поисковых систем, используемых для обработки массивов научных или коммерческих данных. Возникает необходимость эффективного выполнения подобных запросов.

6. Существующие решения

Во-первых, можно использовать кэширование результатов запросов (как на стороне пользователя, так и на стороне поисковой системы). Этот подход необходимо использовать, однако он имеет очевидные недостатки. Кроме того, это более высокоуровневое решение, не улучшающее сам алгоритм, и в некоторых ситуациях не уменьшающее время запроса (которое, являясь значительным для запросов по диапазону, будет неприемлемым с точки зрения требований к эффективности работы системы).

Библиотеки индексации возвращают результат запроса в виде отсортированного по релевантности списка документов. Это удобно, если необходимо представить данный результат конечному пользователю: документы, которые наилучшим образом соответствуют запросу, с большей вероятностью будут подходить пользователю, и должны быть представлены в начале списка результатов в случае приложения с визуальным пользовательским интерфейсом. Однако в большинстве случаев для запроса по диапазонам такая статистика не нужна: пользователя интересует фильтрация документов, соответствующих диапазону, и информация о релевантности того или иного документа не требуется[2]. Для таких запросов Lucene предоставляет возможность фильтрации по диапазону, которая выполняется за меньшее время, чем обычный запрос.

Следующее решение действительно значительным образом оптимизирует выполнение запросов по диапазону. Алгоритм описан в [SD08], в главе «2.2 Optimized Range Queries». В данном случае используется подход, основанный на структуре, схожей с префиксным деревом. Для этого необходимо изначально проиндексировать все поля специальным образом. К примеру, требуется осуществлять поиск по датам следующего вида:

---

[2] http://wiki.apache.org/lucene-java/DateRangeQueries
Параграф «Using a Filter Instead»

CCYYMMDD. В этом случае необходимо произвести индексацию данного поля в виде набора префиксов:

```
C CC CCY CCYY CCYYMM CCYYMMD CCYYMMDD
```

Теперь можно использовать наименьший префикс для поиска по диапазонам, например для поиска даты после 1990-го года достаточно осуществить поиск по терму «199». Или, к примеру, для поиска по диапазону, начинающемуся с января 2007 года и заканчивающемуся январем 2008 года включительно подойдет запрос «2007 OR 200801». Данную схему можно обобщить на целые и вещественные числа. Легко представить алгоритм с помощью деревьев:

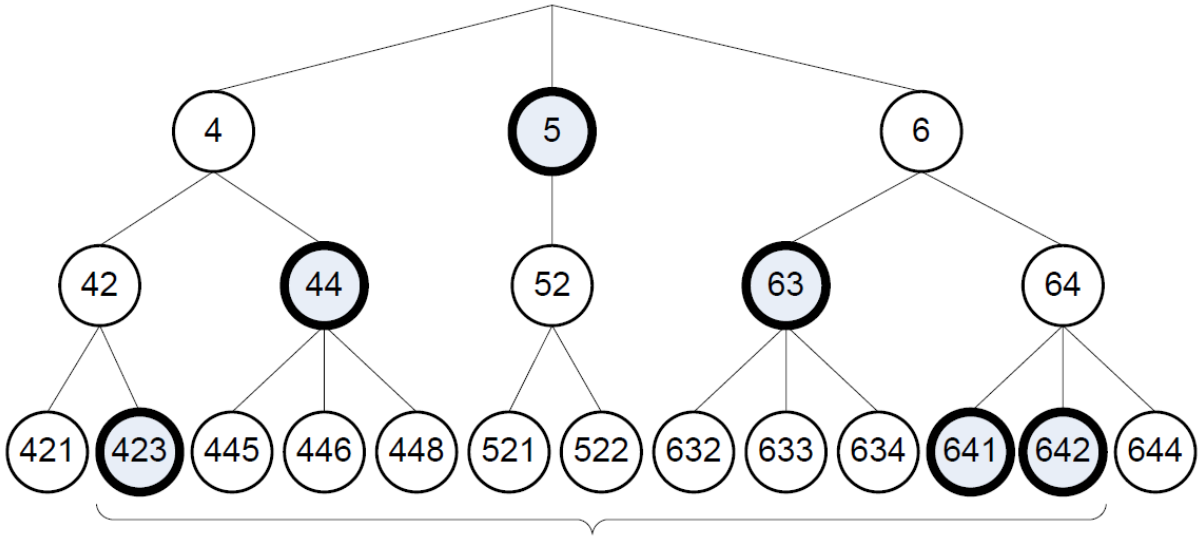

Рисунок 3. Поиск по диапазону ["623" TO "642"].

Для примера рассмотрим реализацию поиска термов между «423» и «642». Необходимо заметить, что в реальности префиксная индексация происходит по элементам двоичного представления чисел. Но для наглядности примера, предположим, что используется десятичное представление. Итак, вместо того, чтобы перечислять все листья дерева, принадлежащие диапазону, можно искать соответствие для термов, приходящихся на центр диапазона, с наименьшей степенью точности, поиск

вблизи границы же приходится осуществлять более точно. В данном случае достаточно произвести поиск по терму «5», чтобы найти все записи, начинающиеся с «5» («521», «522»). То же самое для «44»: «445», «446», «448». Таким образом, рассмотренный поисковой запрос сводится к нахождению элементов, содержащих в индексе «423», «44», «5», «63», «641», или «642». Так как каждый из термов порождает отдельный поисковой запрос, с целью оптимизации по времени необходимо стремиться к уменьшению количества термов в запросе. Предложенный в работе и описанный в следующей главе алгоритм позволяет уменьшить количество термов для некоторых диапазонов.

Такой подход значительно ускоряет выполнение запросов по диапазонам и кроме того позволяет оценить количество термов, в которые преобразуется любой диапазон, сверху (в зависимости от количества значимых битов в типе данных, по которому осуществляется поиск).

Рассмотрим алгоритм более детально. Числа представляются в двоичном виде. Будем рассматривать целые числа, на остальные типы алгоритм можно с легкостью обобщить. Рассматриваем двоичное представление чисел. Первым шагом алгоритма является определение старшего различного бита в числах `min`, `max`. Верхние биты можно на данном этапе откинуть – они будут использованы только при окончании работы алгоритма в качестве битовой маски, которую нужно будет наложить на все результирующие диапазоны. К примеру:

```
min = 110001

max = 110100
```

В данном случае старшим по номеру справа отличающимся битом будет 3й. Можно откинуть старшие биты и сохранить их в виде маски:

```
mask = 110000
```

Затем осуществляется отбрасывание старших битов (операция ‘~’ – побитовое отрицание, ‘&’ – побитовое «И»):

```
min = min & ~mask

max = max & ~mask
```

В результате получаем (для удобства чтения числа выровнены по наибольшему количеству ненулевых битов):

```
min = 001

max = 100
```

Этот диапазон необходимо разбить таким образом, чтобы использовать как можно меньше термов в запросе по префиксам. В данном случае очевидно, что нужно использовать префиксы: «001», «01» и «100». За исключением одного специального случая, описанного далее, разбиение можно выполнить отдельно для правого и левого поддеревьев (минимум и максимум находятся в различных поддеревьях по тому преобразованию, которое было произведено: бит, с номером старшего бита максимума равен нулю для минимума).

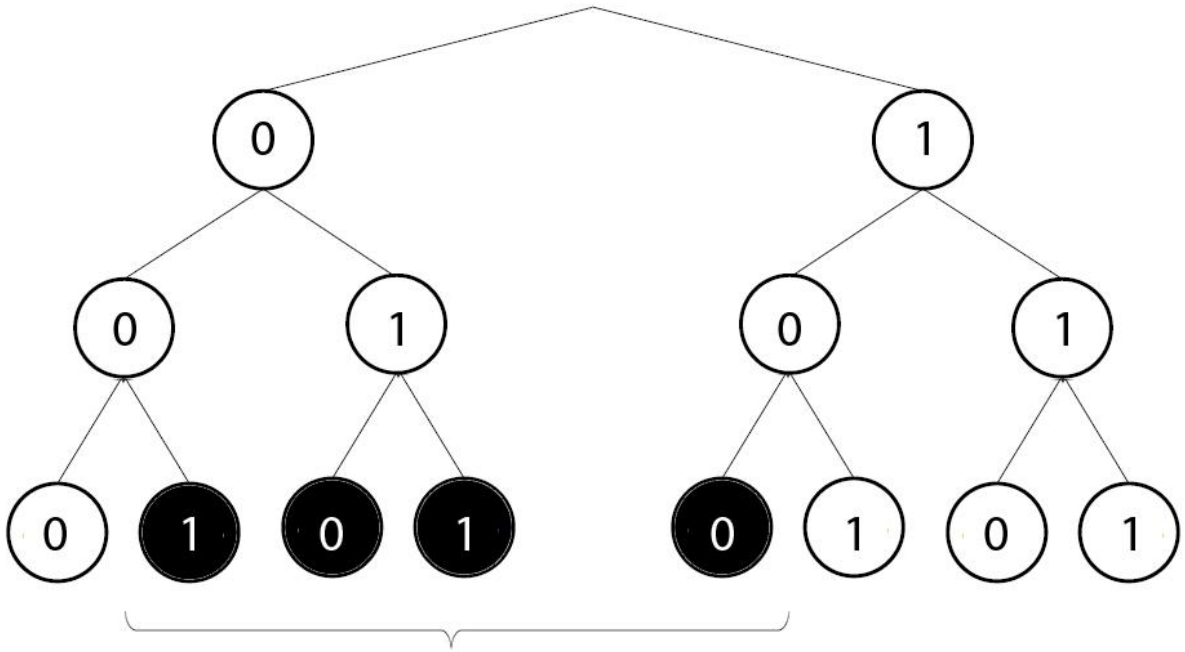

Рисунок 4. Поиск по диапазону при отбрасывании общего для минимума и максимума префикса (маски)

Особым случаем является только ситуация, в которой минимум и максимум являются соответственно крайним левым и крайним правым элементами поддеревьев. То есть:

```
min = [prefix]0...0

max = [prefix]1...1
```

В такой ситуации весь диапазон покрывается единым префиксом (обозначен `[prefix]` выше), то есть поддеревья можно объединить. За исключением этого случая можно рассматривать две независимые задачи: разбиение правого и левого поддеревьев на наименьшее количество диапазонов, покрываемых префиксами. Для удобства будем рассматривать задачу разбиения правого поддерева, решение для левого поддерева получается простым изменением входных значений. Если обратиться к рисунку, то задача для левого поддерева преобразуется в соответствующую задачу для правого поддерева при «перекрашивании» белых листьев в

черные, а черных – в белые, и отражении поддерева относительно вертикальной оси.

Таким образом задача сводится к построению по заданному диапазону `[0, N]` наименьшего множества покрывающих его префиксов. В качестве примера рассмотрим разбиение диапазона `[0, 10]`.

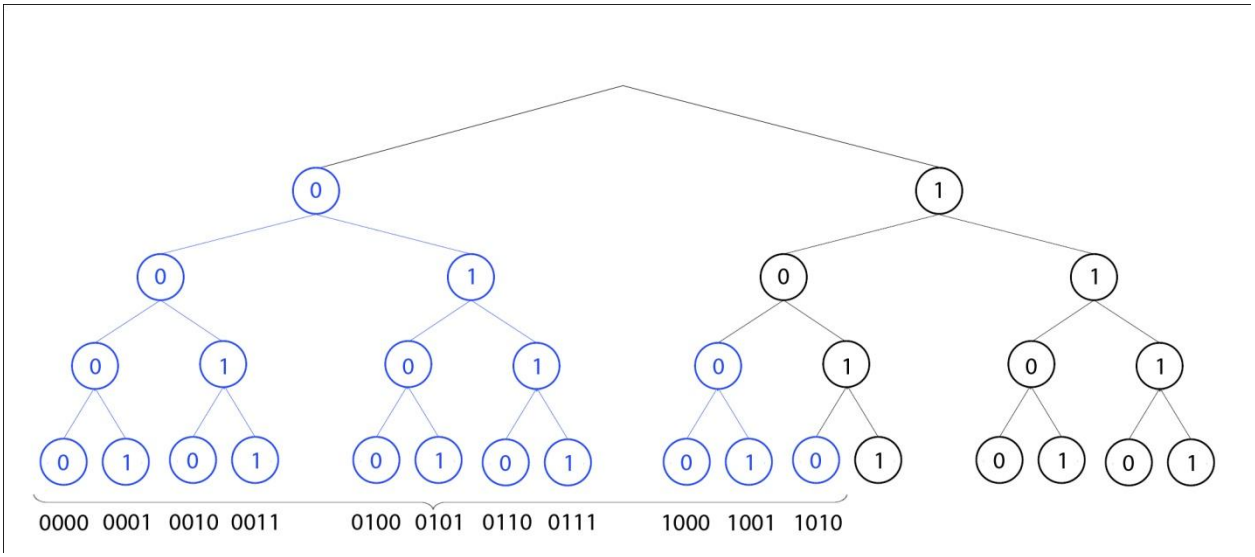

Рисунок 5. Покрытие диапазона [0,9] двоичными поддеревьями, соответствующими префиксам

Заметим, что каждому префиксу однозначно соответствует двоичное дерево, все листья которого будут найдены при выполнении поиска по этому префиксу. Будем говорить, что бинарное дерево (или соответствующий префикс) *покрывает диапазон* `[a, b]`, если все элементы из этого диапазона, и только они, будут найдены при осуществлении запроса по данному префиксу. Используя это понятие, можно определить задачу как построение минимального множества поддеревьев, покрывающих заданный диапазон. У данной задачи существует очевидное решение: выбираем наибольшее слева дерево, покрывающее диапазон `[0, n1]`, содержащийся в `[0, N]`, затем наибольшее поддерево, покрывающее диапазон, принадлежащий `[n1+1, N]`, и так далее. Получаемые на каждом шаге деревья будут иметь уменьшающуюся глубину. Пример покрытия приведен

на *Рисунке 5*. Затем нужно дополнить полученные диапазоны отброшенными ранее битами маски слева. Это и есть множество диапазонов, по которым будет осуществлен запрос.

Поиск по диапазону, покрываемому каждым таким деревом может быть осуществлен достаточно быстро благодаря индексированию по префиксам. В результате задача сводится к разбиению диапазона на непересекающиеся поддиапазоны, объединение которых дает исходный диапазон (такое преобразование называется *query rewriting*, это предварительная обработка запроса для выполнения его наиболее эффективным образом), при этом по каждому из этих диапазонов можно достатончо быстро осуществить поиск. В итоге достаточно просто объеденить данные, получаемые при запросах по каждому из поддиапазонов для получения результата запроса по исходному диапазону.

Такой подход позволяет значительно ускорить поиск по диапазонам. Кроме того немаловажным фактом является то, что типа данных с заданным количеством значащих битов существует не очень большая верхняя граница (прямо-пропорционально зависящая от количества битов) количества термов, в которые будет преобразован запрос после query rewrite. Это позволяет значительно эффективнее использовать память. Так, простейшие реализации запроса по диапазону зачастую приводили к исключению `TooManyClauses Exception`, так как для выполнения запроса требовалось слишком много памяти, так как запрос преобразовывался в слишком большое количество термов.

Отметим еще один факт: количество деревьев, необходимых для такого вида покрытия диапазона `[0, N]`, равно количеству единиц в двоичном представлении числа `N+1`. Данное замечание позволяет с легкостью вычислить количество деревьев в покрытии при использовании данного алгоритма даже для больших чисел. Это легко доказать по определению

алгоритма: на каждом шаге выбирается дерево, покрывающее диапазон из 2^k элементов, где k – максимальная степень двойки, такая что $2^k <$ `N+1`. Затем происходит уменьшение `N`: `N = N -` 2^k. Отсюда легко увидеть, что числа `k1,...,kN` являются ни чем иным, как позициями, на которых стоят единицы в числе `N+1` (прибавление единицы происходит из-за того, что нумерация начинается на с нуля, а с единицы).

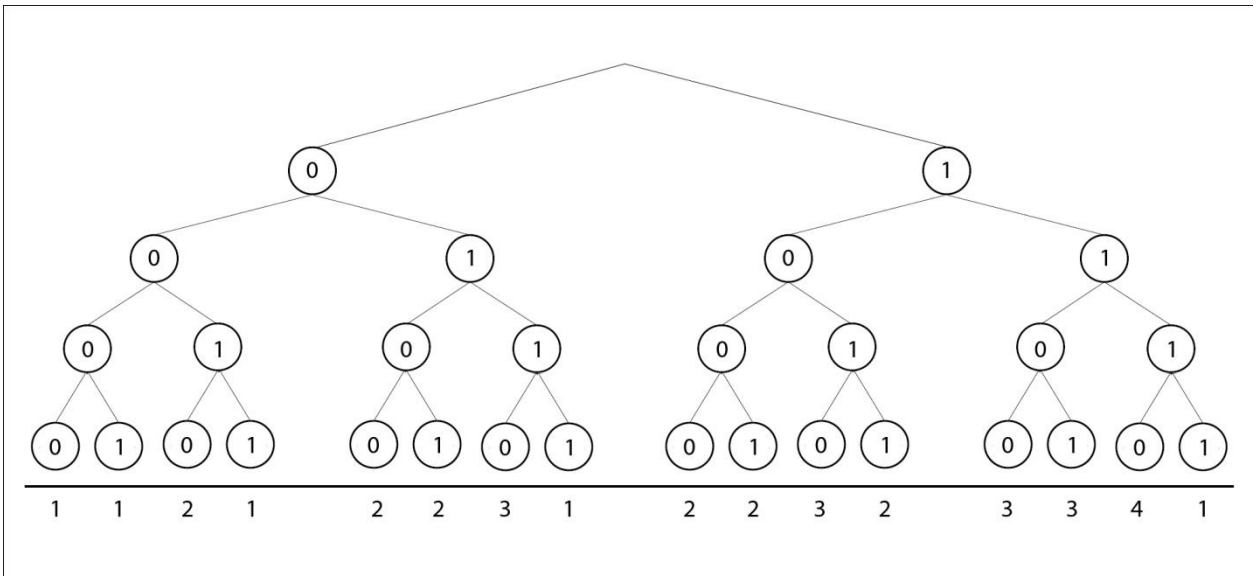

Рисунок 6. Количество поддеревьев, покрывающих диапазоны [0, n], где n < 16

7. Предложение по оптимизации

В рамках дипломной работы был разработан алгоритм, описание которого представлено в данной главе.

Нетрудно заметить, что поиск по диапазону $[0,\ 2^k - 2]$ порождает большое количество деревьев (k деревьев). Однако этот диапазон можно покрыть всего двумя деревьями, если включить в число возможных операций вычисление разности диапазонов: $[0,\ 2^k - 2] = [0,\ 2^k - 1] \ / \ [2^k - 1, 2^k - 1]$. Таким образом можно рассмотреть способы покрытия, уменьшающие общее количество деревьев в покрытии при допущении возможности вычитания. Заметим, что при этом нужно сделать дополнительное предположение, которое позволит избежать искажения результатов запроса: значение поля, по которому производится поиск, должно быть единственно для каждого документа (то есть отображение документ → поле обладает свойством функциональности).

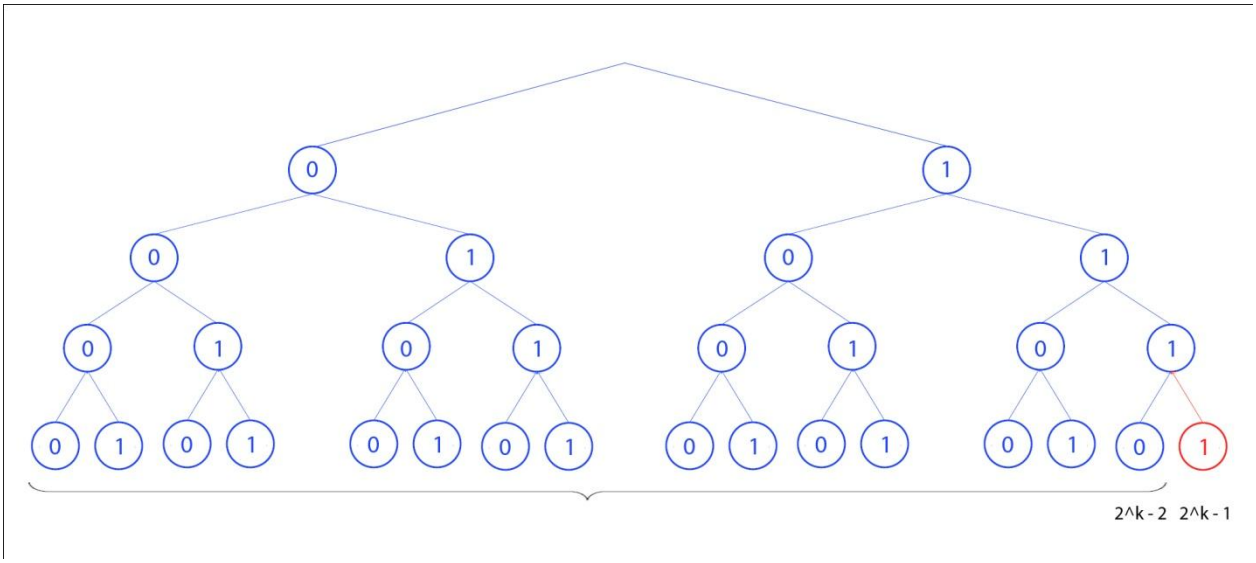

Рисунок 7. Реализация поиска по диапазону [0, $2^k - 2$] с использованием вычитания

Иначе возможна следующая ситуация: во время исключения документов, принадлежащих диапазону, дополняющему заданный до $[0,\ 2^k - 1]$, будут

откинуты документы, содержащие одновременно еще и значения этого поля, принадлежащие исходному диапазону. В итоге произойдет потеря части документов в результатах запроса.

Существует закономерность в последовательности чисел на *Рисунке 6*; видно, что при переходе от интервала $[0, 2^k - 1]$ к $[0, 2^{k+1} - 1]$ числа соответствующие элементам в диапазоне $[2^k, 2^{k+1} - 1]$, за исключением $2^{k+1} - 1$ (этому элементу соответствует число 1) получаются прибавлением единицы к элементам диапазона $[0, 2^k - 1]$. Логично, что при движении вправо, в среднем числа будут увеличиваться. Алгоритм построен на этом наблюдении. Обозначим количество деревьев, покрывающих интервал [0,N] в соответствии с описанным выше алгоритмом как trees(N). Пусть высота дерева, в котором находится элемент N, равна m. Тогда в первом приближении алгоритм выглядит следующим образом:

1) Вычисление trees(N), trees($2^m - 1 - N$)

2) Сравнение значений. Если trees(N)< trees($2^m - 1 - N$), поиск выгоднее производить в соответствии с исходным алгоритмом. Если же:

   trees(N)= trees($2^m - 1 - N$) + A

   То выбор запроса осуществляется в зависимости от значения A. Если число A достаточно велико, то имеет смысл произвести 2 запроса: по диапазонам X=[0, $2^m - 1$] и Y=[N, $2^m - 1$] и затем вернуть разность query(X) / query(Y) в качестве результата исходного запроса, где query(X) – результат поиска по диапазону X. Число A должно быть достаточно велико, чтобы выигрыш в уменьшении числа диапазонов не перекрывался дополнительным временем, необходимым для вычисления разности множеств результатов query(…). Таким образом число A необходимо выбрать эмпирически.

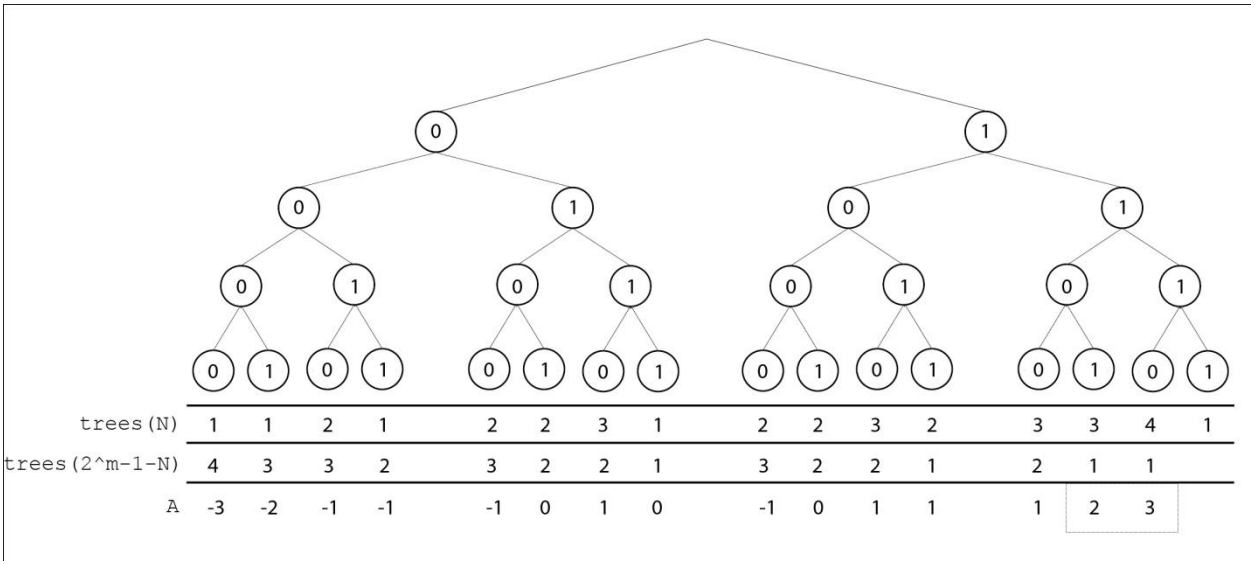

Рисунок 8. Расчет значения A для промежутка [0,15]. При росте длины промежутка выгодно использовать оптимизацию для значений, близких к правой границе.

В соответствии с описанной выше закономерностью видно, что в среднем функция `trees(N)` растет пропорционально `log(N)`. Таким образом, логично ожидать наибольшего выигрыша по производительности на конце диапазона. На *Рисунке 8* представлено изменение параметра `A` при росте числа `N`. Как уже было сказано выше, при применини алгоритма на скорость работы действуют два разнонаправленных эффекта:

1) Выигрыш от уменьшения количества термов в запросе (пропорционален параметру `A`);

2) Уменьшение производительности в результате затрат, связанных с операцией вычитания множеств. Затраты на данный шаг прямо пропорциональны количеству элементов в множестве `query(Y)`;

В результате данных наблюдений можно сделать выводы об области применимости описанного алгоритма. Итак, алгоритм представляется в виде последовательности шагов:

1) Для заданного диапазона произвести вычисление `A = trees(`$2^m - 1 - N$`) - trees(N);`

2) Вычислить значение $2^m - 1 - N$, которое прямо пропорционально количеству элементов в `query(Y);`

3) На основании данных параметров сделать вывод о целесообразности применения предложенной оптимизации;

4) При принятии решения об использовании стандартного алгоритма, результат не требует последующей обработки. Произвести возврат значений, полученных в результате работы стандартного алгоритма;

5) При принятии решения об использовании измененного алгоритма, необходимо вернуть в качестве результата разность диапазонов (границы диапазонов были описаны ранее);

В ходе дипломной работы был написан код, осуществляющий выполнение алгоритма, в зависимости от задаваемых параметров.

В рамках работы были проведены тестовые запуски алгоритма на созданном индексе из 500000 документов, проиндексированных таким образом, что значения, соответствующие полю равномерно распределены в интервале Δ=`[0, 2000000]`. Необходимо учитывать, что в данном случае сбор статистики происходит с погрешностью, вызываемой операциями с жестким диском.

Было произведено тестирование алгоритма на наборах диапазонов следующим образом:

1) Случайные диапазоны из интервала Δ

2) Диапазоны вида `[0, `$2^k - 2$`]`

3) Диапазоны вида `[0, `$2^k - 2 - a$`]`, где `a > 0` невелико

В результате запусков на компьютере Intel Celeron 1.86GHz, 2GB RAM была получена следующая статистика:

1) Для соотношения времени работы оптимизированного алгоритма к времени работы исходного в зависимости от параметра `A` на случайных наборах данных (равномерно распределенных в интервале Δ) получено соотношение, представленное на графике:

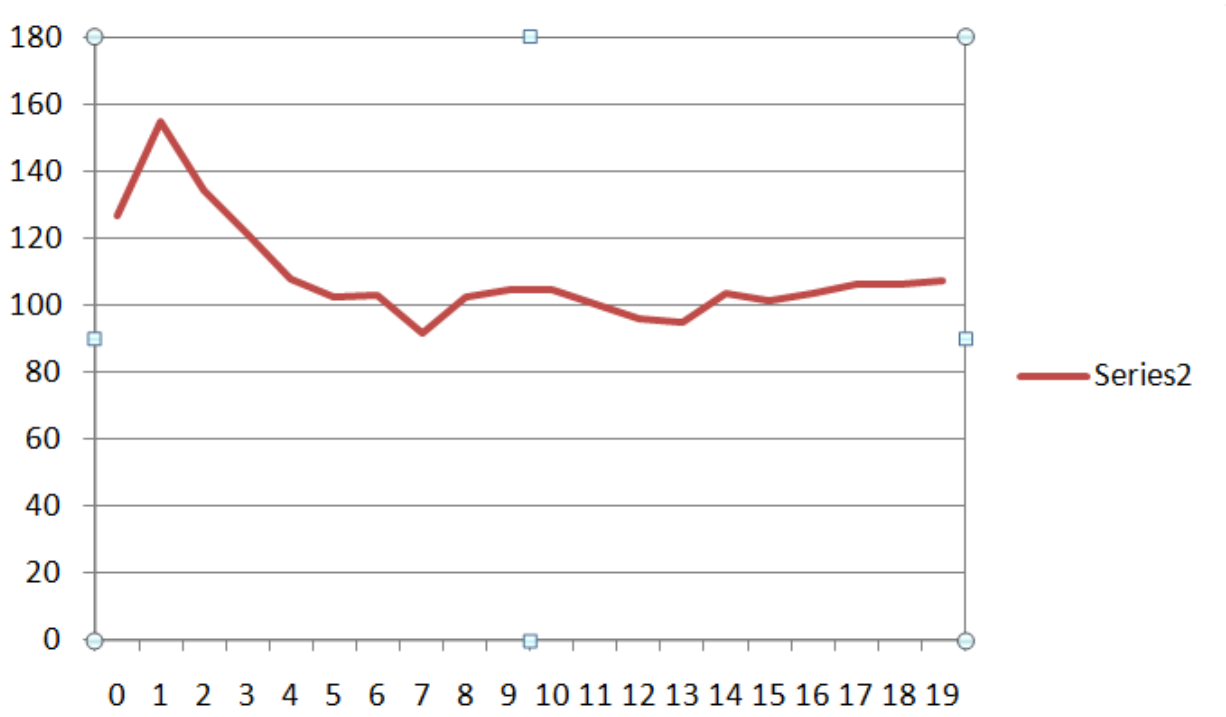

Рисунок 9. Соотношение в % по времени работы оптимизированного и исходного алгоритмов в зависимости от параметра A. По оси абсцисс значения A, по оси ординат – процентное соотношение времени работы. Видно, что имеет смысл выбирать A не меньше 5.

Видно, что необходимо выбирать `A` не меньшим 5, иначе выигрыш от уменьшения количества диапазонов нивелируется временем, затрачиваемым на вычитание диапазонов. Кроме того, этот график показывает, что при адекватных значениях параметра `A` в среднем алгоритм работает за то же время, что и исходный.

2) Теперь проверим работу алгоритма на диапазонах, при поиске по которым ожидается максимальный выигрыш по производительности:

[0, $2^k - 2$]. Запуск был произведен для k, принимающего значения из [5, 20].

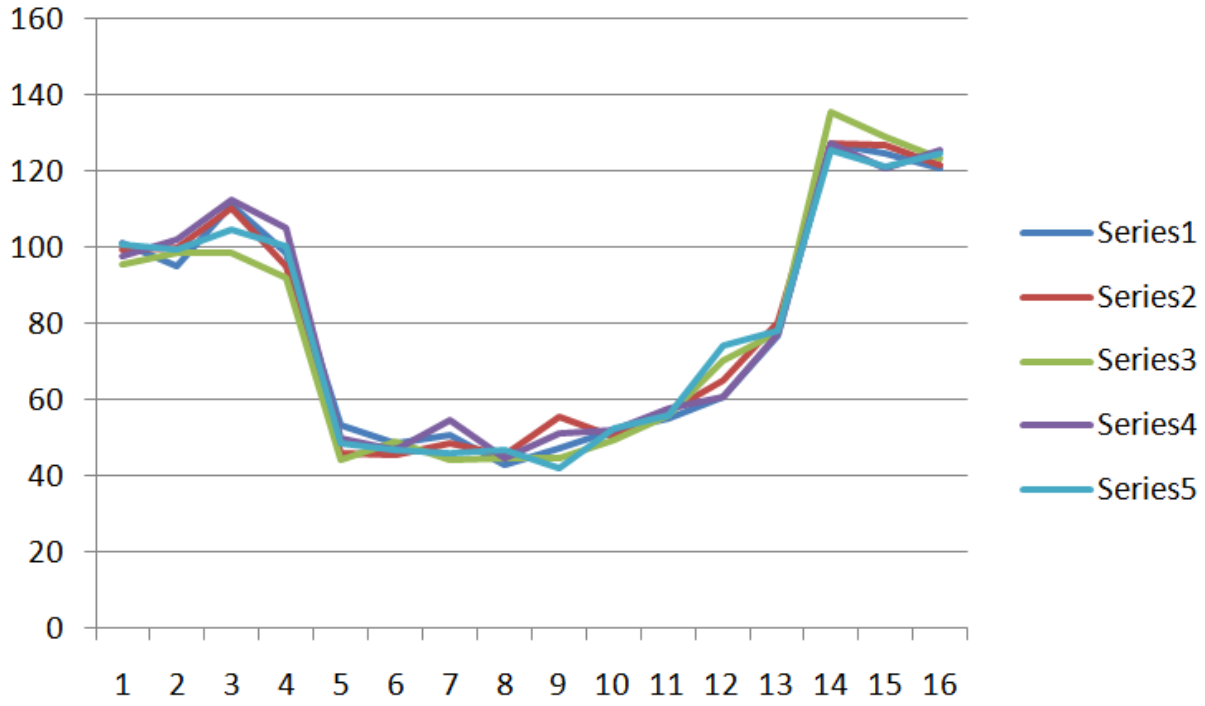

Рисунок 10. Соотношение времени работы оптимизированного и исходного алгоритмов на диапазонах вида [0, $2^k - 2$] за 5 запусков. По оси абсцисс значения k, по оси ординат – процентное соотношение времени работы. Видно, что в середине интервала достигается уменьшение времени работы около 50%.

Таким образом на специфических интервалах алгоритм оправдывает ожидания, сокращая время работы до 50%.

3) Рассмотрим поиск по диапазонам близким к предыдущему классу диапазонов [0, $2^k - 2 - a$], где a небольшое положительное число.

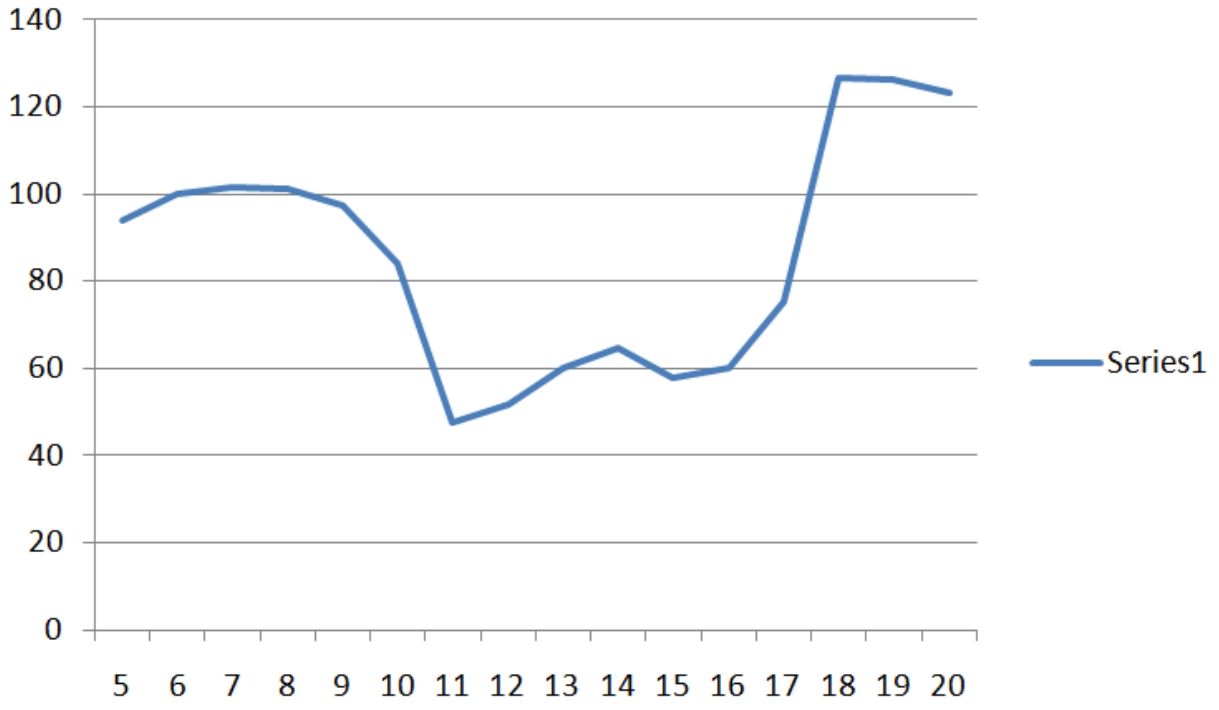

Рисунок 11. Соотношение времени работы оптимизированного и исходного алгоритмов на диапазонах вида [0, $2^k - 2 - a$]. По оси абсцисс значения k, по оси ординат – процентное соотношение времени работы, усредненное по a из интервала `[1,5]`.

Таким образом, алгоритм уменьшает время работы для специфических диапазонов, описанных выше (при поиске по которым время работы исходного алгоритма велико относительно времени работы исходного алгоритма в среднем по диапазонам из интервала Δ) и близких к ним. Однако за счет сравнительной малости множества диапазонов, на которых будет достигнуто ускорение, относительно общего числа диапазонов из интервала Δ, ускорение в среднем практически не заметно. Поэтому, и так как необходима предварительная калибрация параметров алгоритма (например, значения `A`) применимость алгоритма должна быть изучена отдельно для каждой конкретной совокупности индекса и аппаратного обеспечения для работы с ним. Тем не менее, учитывая что алгоритм производит оптимизацию «в худшем случае», его использование обоснованно для поисковых систем, так как пользователь ожидает получения ответа на запрос в течение короткого промежутка времени. Таким образом, описанная

оптимизация «сглаживает» время работы алгоритма, делая его более предсказуемым.

8. Заключение

В ходе дипломной работы было осуществлено:

1) Описан оптимизированный алгоритм и его дополнительные параметры;
2) Предложенный алгоритм был реализован;
3) Был произведен набор статистических запусков оптимизированного алгоритма на наборе из 500 тыс. документов, произведен анализ полученных результатов и сделано заключение о сфере его применимости;

Актуальность работы объясняется распространенностью библиотеки индексации Lucene. Кроме того, возможно применение того же алгоритма для реализации запросов по диапазону для любой библиотеки индексации, основанной на обратном индексе.

Кроме того, важно то, что алгоритм производит уменьшение времени работы алгоритма в худшем случае. В сочетании с предметной областью это дает весьма положительный эффект: пользователи поисковых систем ожидают получение результатов запроса в течение достаточно небольшого промежутка времени. В случае, если на некоторых входных данных работа системы будет занимать значительное время, пользователи предпочтут использовать альтернативные программные продукты. Таким образом, предложенная оптимизация позволяет сделать работу поисковой библиотеки более предсказуемой, что положительно сказывается и на общем удовлетворении пользователей результатами работы системы, и на возможности оценки времени работы различных составных задач.

Дополнительным значительным плюсом предложенного подхода является то, что для его применения в случае Lucene не требуется изменять

используемую сборку библиотеки. Программа модифицирует входные данные запроса и обрабатывает результат таким образом, чтобы:

1) Обеспечить лучшее время работы библиотеки индексации, благодаря знаниям о ее внутренней структуре. То есть при использовании алгоритма Lucene считается «белым ящиком».

2) Результаты выполнения запроса не отличаются от результатов выполнения исходного запроса благодаря исключению лишних элементов.

   Таким образом, при конечном использовании отстутствует необходимость в приостановке работы back end системы. Достаточно лишь изменить процесс обработки запросов и результатов на стороне клиента, либо на стороне back end, но без остановки работы поисковой библиотеки.

Список использованной литературы

[EG04] Erik Hatcher and Otis Gospodnetić. Lucene in Action. Manning Publications (December 1, 2004)

[ZM06] Justin Zobel, Alistair Moffat. Inverted files for text search engines. ACM Computing Surveys (CSUR), Volume 38, Issue 2 (2006)

[SD08] Uwe Schindler, Michael Diepenbroek. Generic XML-based Framework for Metadata Portals. Center for Marine Environmental Sciences (MARUM), University of Bremen, 2008

[LW] Lucene-java Wiki. http://wiki.apache.org/lucene-java/DateRangeQueries

[DZ06] Deng Peng Zhou, Delve inside the Lucene indexing mechanism, Shanghai Jiaotong University, 2006

[SO01] Steven J. Owens, Lucene Tutorial, 2001.
http://www.darksleep.com/lucene/

[CP90] Doug Cutting, J. Pedersen, Optimizations for Dynamic Inverted Index Maintenance, Proceedings of SIGIR'90, 1990