САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Разработка алгоритмов обеспечения качества распределенного поискового робота для сети интернет

Дипломная работа студента 545 группы

Волкова Сергея Андреевича

Научный руководитель ..... ассистент кафедры АСОИУ СПбГЭТУ ЛЭТИ

Выговский Л.С

/подпись/

Рецензент ..... ст.преп. Луцив Д.В

/подпись/

“Допустить к защите” ..... д.ф.-м.н., проф. Терехов А.Н.

заведующий кафедрой, /подпись/

Санкт-Петербург

2011

SAINT PETERSBURG STATE UNIVERSITY

Mathematics & Mechanics Faculty

Software Engineering

Development of quality assurance algorithms for the distributed web crawler

by

Sergey Volkov

Master's thesis

Supervisor	.....	L.S Vygovsky
Reviewer	.....	Senior Lect. D.V Luciv
"Approved by"	.....	Professor A. N. Terekhov
Head of Department,		

Saint Petersburg

2011

Оглавление

Введение **3**

1 Обзор области **5**

1.1 Web Search 5

1.2 Поиск по новостям 6

2 Постановка Задачи **7**

2.0.1 Условия 7

2.0.2 Требования 7

3 Обзор средств **9**

3.1 Сравнение open source поисковых роботов 9

3.1.1 Описание роботов 10

3.1.2 Выбор 11

3.2 Архитектура Nutch 11

3.2.1 Основные Этапы 12

3.2.2 Система Плагинов 13

4 Алгоритмы **15**

4.1 Ранжирование 15

4.1.1 OPIC 16

4.1.2 Предложенный алгоритм 16

4.1.3 Особенности реализации 17

4.1.4 Результат 18

4.2 Работа с индексом 18

4.2.1 Lucene 19

4.2.2 Алгоритмы 20

4.2.3 Особенности реализации 20

4.2.4 Результат 21
4.3 Раннее удаление дубликатов 22
4.3.1 Реализация в Nutch 22
4.3.2 Алгоритм 22
4.3.3 Сравнение Key Value СУБД 22
4.4 Черные списки 24
4.4.1 Фильтры 25
4.4.2 Создание фильтров 25
4.4.3 Получение статистики по crawldb 26
4.4.4 Реализация 27
4.4.5 Результаты 29
4.5 Генерация фильтров 29
4.5.1 Алгоритм 29
4.5.2 Реализация 30
4.5.3 Результат 30

Заключение **31**

Введение

— В то время, когда наши корабли бороздят просторы Вселенной...

На текущем этапе развития, когда общество осуществляет переход от постиндустриальной эпохи к информационной, требования к системам хранения и обработки информации непрерывно растут. Традиционные подходы не справляются с ростом количетва данных. Трудно оценить общий объем данных, однако, по оценкам IDC (International Data Corporation) в данный момент хранится порядка $1.8 \cdot 10^{21}$ байт, что в 10 раз больше чем в 2006 году.

К значительному количеству данных можно получить доступ через Всемирную Паутину (WWW). При таких объемах остро стоит задача организации эффективного поиска. Уже в 2009 году Google Search обработал более 109,5 миллионов сайтов, и более 10^{12} уникальных URL. На данный момент их индекс содержит $4 \cdot 10^{10}$ документов.

Одной из специфических областей поиска является поиск по новостным ресурсам. Для документов с новостных сайтов характерна привязка к дате, региону и тематике. Таким образом, такие документы легко классифицировать, что позволяет производить более качественный поиск и анализ. В качественном инструменте для анализа СМИ заинтереснованы различные консалтинговые и PR агенства, пресс-службы, маркетинговые отделы крупных компаний.

Одна из задач поисковой системы - нахождение и загрузка документов (Web crawling), за которую отвечает поисковый робот (Spider, Crawler). Web crawling весьма ресурсоемкий процесс. Основные проблемы связаны с большим количеством данных, остутсвием контроля над данными, постоянным изменением структуры ресурсов, динамическим созданием страниц и низким качеством некоторых ресурсов. Од-

нако, специализация на определенной узкой области web позволяет существенно повысить производительность web crawler'a.

Конечной целью работы является создание системы способной эффективно индексировать новости в рунете.

Глава 1

Обзор области

1.1 Web Search

Поисковая система — система, разработанная для поиска информации в WWW. Результаты поиска которой, как правило, представлены в виде списка "попадений". Информация может состоять из веб страниц, изображений, мультимедийных данных. Одной из первых поисковых систем стал проект Archie, разработанный в 1990 году студентами McGill University. Программа скачивала списки файлов с открытых FTP серверов, и добавляла их в базу с возможностью поиска по названию.

Поисковая система состоит из трех основных компонент:

- поисковый робот — программа, предназначенная для перебора докуметнов и занесения данных о них в базу;
- индексатор — программа, создающая на основе полученных с помощью робота данных индекс;
- поисковик — программа, осуществляющая поиск в полученном индексе на основе поискового запроса.

В условиях постоянно расширяющегося и изменяющегося WWW, непрерывно возрастают требования к поисковым системам.

Системы общего поиска нацелены на охват большей части данных доступных в WWW. Такие системы предназначены для поиска наиболее релевантных документов относящихся к объекту поиска.

Системы тематического поиска более разнообразны, и требования к ним более специфичны. Например Google Microblogging Search Engine, ориентированный на поиск по записям в микроблогах, где крайне важна задержка между созданием записи, и ее попадением в индекс.

1.2 Поиск по новостям

Основные источники новостей в WWW — это электронные СМИ и блоги. По данным liveinternet на 2008 год, рунет насчитывает 4392 сайта СМИ, а число блогов значительно больше - по данным Яндекс за 2009 год в русскоязычной блогосфере насчитывается порядка 840000 активных блогов, на которых ежедневно публикуется порядка 300000 постов.[1] Очевидно, за прошедшее время количество таких сайтов значительно увеличилось. За сутки каждое из подобный изданий публикует до 100 документов (lenta.ru). Таким образом, можно говорить о десятках миллионов создаваемых документов в год.

Под новостью понимается документ содержащий текст, заголовок и дату. Для СМИ и блогов характерно:

- большое количество посторонних страниц, не содержащих новостей;
- схожая структура (как именования url, так и самого html);
- наличие rss ленты.

К новостным поисковым системам предъявляются следущие требования:

- минимальное время между публикацией статьи на новостном ресурсе и ее предоставление в поисковой выдаче;
- поиск должен осуществлять не по всей HTML–странице, а только по ее существенным частям.

[1] http://mediarevolution.ru/audience/1962.html

Глава 2

Постановка Задачи

Конечной целью работы является создание поискового робота способного эффективно индексировать новости в рунете.

Поисковый робот (Web crawler) — программа для поиска веб-страниц в сети[1]. Грубо говоря поисковый робот начинает с URL для начальной страницы p_0. Он скачивает p_0, выделяет все URL которые в ней находятся, и добавляет их в очередь URL (*crawling frontier*). Затем робот в некотором порядке выбирает URL из очереди и повторяет процесс.

Каждая скачанная страница передается клиенту, котрый затем создает индекс по страницам.

2.0.1 Условия

Важным фактором, влияющим на качество поиска, является идентификация страницы содержащей новость и выделение её содержательной части. В данной работе предполагается наличие базы данных, содержащей правила на основе регулярных выражений, которые по URL определяют содержит ли данная страница новость, а так же правил по которым из веб-страницы выделяется содержательная часть.

Далее под *документом* понимаются выделенные по этим правилам данные.

2.0.2 Требования

- поддержка десятков миллионов документов;
- скорость роста базы документов — более 50 тысяч документов в день;

- попадение в индекс документов из RSS лент не позже чем через 12 часов после их публикации;
- попадение в индекс старых документов из архива — некоторые документы могут находиться достаточно “глубоко“ (например что-бы получить новости месячной давности на ресурсе fontanka.ru необходимо сделать 5 переходов).
- отсутствие дубликатов в пределах одного домена — под дубликатом понимается документ у которого совпадает URL или текст с другим документом.

В качестве основного показателя эффективности используется количество полученных документов за сутки. Поскольку объем данных непрерывно увеличиватся, неизбежна деградация производительности. Это связано в первую очередь с постоянным ростом размера базы ссылок. Поэтому также в качестве метрики используется произведение количества полученных за сутки документов к общему числу документов. То есть если количество вычислительных мощностей растет прямопропорционально числу документов в индексе, скорость получения новых документов должна хотя бы не падать, что подразумевает хорошую горизонтальную масштабируемость всех ключевых компонент системы.

Глава 3

Обзор средств

Большинство популярных поисковых сервисов предоставляют возможность поиска по новостям (Google, Yandex, Yahoo!), однако, они пользуются закрытыми алгоритмами и не предоставляют доступа непосредственно к индексу

3.1 Сравнение open source поисковых роботов

Существует достаточно много open source поисковых роботов. Для успешного решения задачи робот должен справлятся с нагрузкой (50000 документов в день, база ссылок порядка 10^9 и порядка 10^7 документов в индексе), быть легко изменяем и расширяем. Поскольку предполагается коммерческое использование робота не протяжении долгого времени, проект должен быть достаточно зрелым и развивающимся.

Метрики

- язык
- поддержка robots.txt
- распределенность системы
- тип хранения индекса
- тип хранилища url
- поддержка

Роботы

- DataparkSearch — поисковая система разработаная для поиска по локальным файлам, группам сайтов и интранету
- AspSeek — поисковая система оптимизированная для работы с многими сайтами, и средней загрузкой — до нескольких миллионов страниц
- Nutch — поисковая система основанная на Lucene
- Hounder — поисковая система онованная на nutch
- Heritix — еще одна java поисковая система

Сравнение

3.1.1 Описание роботов

DataparkSearch [1] — предназначен для работы с небольшой группой сайтов или интранета, написан на языке C. Состоит из двух частей — индексатора и CGI фронтенда. DataparkSearch отделился в 2003 году от MnoGoSearch. Имеет встроенные парсеры для html, xml, есть возможность написания собственных парсеров для других форматов. Данные по ссылкам хрянятся в SQL базе данных. Можно запустить сразу несколько процессов индексации работающих с одной базой. Данные по документам могут храниться как в бд, так и в собственном формате на диске (cache mode), который эффективно работает с несколькими миллионами документов.

AspSeek [2] — поисковая система написанная на C++ и оптимизированная для работы с множеством сайтов. Состоит из индексирующего робота, поискового демона и CGI фронтенда. Данные поискового сервера хранятся в SQL базе данных и бинарных файлах (delta files), рассчитан для работы с несколькими миллионами документов.

Nutch [3] — поисковый робот написанный на java, работающий поверх системы Hadoop[4]. Изначально Nutch разрабатывался в рамках проекта Lucene[5], однако в 2005 году отделился как отдельный проект. Благодаря работе поверх Hadoop обладает хорошей

[1] http://www.dataparksearch.org/
[2] http://www.aspseek.org/
[3] http://nutch.apache.org/
[4] http://hadoop.apache.org/
[5] http://lucene.apache.org/

маштабируемостью (до 100 машин в кластере). Nutch отличается гибкой системой плагинов, через которые осуществляется поддрежка множества протоколов (http, ftp, file) и форматов (от html до msexcel и swf).

Hounder [6] — поисковая система на Java, робот которой основан на Nutch. Из дополнительного функционала следует отметить фильтр Байеса для разбиения документов по категориям.

3.1.2 Выбор

В качестве основы системы был выбран Nutch, так как он полностью удовлетворяет требованиям.

- Нагрузка — Nutch использовался в качестве основы для Sapphire Web Crawler[7], с помощью которого было скачано более 10^9 документов со средней скростью в 431 документ в секунду.
- Расширямеость — благодаря модульности и гибкой системе плагинов можно достаточно легко изменять поведение системы.
- Поддержка — проект разрабатывается более 7 лет, текущая стабильная версия проекта 1.2 была выпущена в сентябре 2010. Проект поддерживается “Yahoo! Research Labs”.

3.2 Архитектура Nutch

Высокая масштабируемость робота достигается за счет работы поверх MapReduce фреймворка Hadoop[3]. Hadoop на данный момент представляет набор подпроектов Apache Software Foundation, среди которых находятся Hadoop MapReduce и HDFS.

MapReduce — модель программирования для обработки больших объемов данных, впревые опубликованная[2] Google в 2004 году. При данном подходе логика программы реализуется в функциях *map*, которая преобразует пары ключ/значение

[6] http://hounder.org/

[7] http://boston.lti.cs.cmu.edu/crawler/index.html

в набор промежуточных пар ключ/значение, и *reduce*, которая обрабатывает все значения связанные с одним промежуточным ключом 3.1.

$$\begin{aligned} map &: \langle key_{in}, value_{in} \rangle \rightarrow \langle key_{int}, value_{int} \rangle^* \\ reduce &: \langle key_{int}, value_{int}^+ \rangle \rightarrow \langle key_{out}, value_{out} \rangle^* \end{aligned} \tag{3.1}$$

Написанная таким образом программа может автоматически параллельно выполняться на кластере машин, программное обеспечение которых брало бы на себя распределение данных, управление выполнением задач, поддержку отказов и управление взаимодействием между узлами кластера.

HDFS (Hadoop Distributed File System) — распределенная система хранения данных, основанная на модели GFS[4] — распределенной файловой системы используемой Google. HDFS предназначена для:

- больших файлов — имеются ввиду файлы от нескольких сотен мегабайт до нескольких терабайт;
- потокового доступа к данным — предпологается что данные записываются один раз, и программа в процессе работы использует большую часть из записанного набора данных;
- дешевого оборудования — риск отказа оборудования достаточно высок.

Файл в HDFS представляет из себя последовательность достаточно больших блоков (по умолчанию 64Mb), которые в нескольких экземплярах (обычно используется 3 реплики) хранятся на различных узлах —*DataNode*. Последовательность блоков в файле и их расположение управляется через *NameNode*. Таким образом нагрузка на передачу и запись данных распределяется между набором *DataNode*, а структура папок и файлов находится в *NameNode*.

3.2.1 Основные Этапы

Nutch является средством инкрементальной сборки, на каждом этапе выполняются следующие действия:

- *inject* — добавление списка URL в базу ссылок *crawldb* (используется при инициализации сборки или при добавлении новых доменов);

- *generate* — из *crawldb* выбирается фиксированное число ссылок для их последующего скачивания;
- *fetch* — скачивание документов по выбранным ссылкам;
- *parse* — разбор документов и выделение их них ссылок;
- *invertlinks* — обновляение базы обратных ссылок;
- *index* — создание индекса по сегменту;
- *merge index* — объединение индекса по сегменту с основным;
- *update* — обновление *crawldb*

3.2.2 Система Плагинов

Особого внимания заслуживает система плагинов в Nutch, через которую реализована основная функциональность. Плагины выполняют разбор документов, индексацию, поиск, ранжирование, фильтрацию ссылок и.т.д.

Каждый плагин предоставляет одно или несколько *расширений* (*extensions*) для *точек расширений* (*extension points*), причем сами по себе точки расширений определены в плагине.

Таблица 3.1: Сравнение поисковых роботов.

Название	Язык	Распределенность	robots.txt	Индекс	Хранилище url	Документы
DataparkSearch	C		+	SQL database/собственный формат	SQL database	10^6
AspSeek	C++	??	+	SQL database	SQL database	10^6
Nutch	Java	+	+	Lucene index	распределенный файл	10^9
Hounder	Java	+	+	Lucene index	распределенный файл	???

Глава 4

Алгоритмы

Поскольку WWW представляет из себя крайне сложную среду, которая постоянно развивается, предпочтение было отдано практическому подходу:

1. запустить систему без модификаций;
2. найти самое слабое место в системе;
3. принять меры по его устранению;
4. перейти к пункту *2*.

4.1 Ранжирование

При работе системы в стандартной конфигурации было замечено, что только из 10% скачиваемых веб-страниц выделяются документы. Это происходит из-за неоптимального упорядочивания ссылок из *crawldb*.

Определение порядка выбора URL для скачивания существенно сказывается на эффективности работы робота.[1][5][6] Порядок неважен только в том случае, если робот нацелен на одноразовое скачивание всего Web, и нагрузка создаваемая роботом на целевые сайты не важна, так как тогда каждая известная URL будет в конце концов загружена. Однако большинство роботов не способно посетить каждый URL по трем основным причинам:

- Ограничение по ресурсам — размер хранилища, ширина канала, CPU time для обработки страниц.
- Сбор документов занимает время, поэтому в определенный момент робот вынужден заново посещать некоторые страницы для нахождения изменений.

- Динамическое создание страниц — сейчас большинство сайтов разадают не статический контент, а создают его динамически при помощи скриптов обрабатывающих URL и возвращающих результат, таким образом количество страниц на сайте может быть неограничено.

Во всех остальных случаях существенно что бы робот сначала посещал “важные” страницы. Ранжирование отвечает за определенеие того, на сколько URL “важна”. В Nutch порядок выбора URL определяется в плагинах подключенных к точке расширения *ScoringFilter*, в результате работы которых каждой URL сопоставляется *score*. На каждой итерации для скачивания выбираются URL с наибольшим score.

В стандартной конфигурации Nutch для выбора ссылок используется *OPIC Score* плагин.

4.1.1 OPIC

OPIC — On-Line Page Importance Computation[6]. Данный алгоритм расчитывает важность страницы на основе важности страниц на него ссылающихся. В отличии от off-line методов, когда сначала скачивается часть Web, а потом на основе полученного графа ссылок вычисляется важность страниц, этот метод позволяет работать когда большая часть сетевого графа еще не известна.

В Nutch реализована упрощенная версия данного алгоритма, при котором в момент скачивания страницы к score каждой из ссылок с нее (*outlink*) добавляется S_0/n где S_0 — *score* скачанной страницы, а n — число исходящих ссылок. Данный способ плохо подходит для поиска новостей, поскольку для новостей основным признаком “важности” веб-страницы является присутствие в нем текста новости, что на практике не связано с количеством входящих ссылок.

Поскольку в системе идентификация новостей производится только по URL возникает желание скачивать только те URL, которые подходят под правла.

4.1.2 Предложенный алгоритм

Так как в Nutch можно использовать одновременно сразу несколько ScoringFilter’ов (полученные из различных алгоритмов *score* просто перемножаются), вместо изменения стандартного метода следует создать отдельный плагин, который всем новым поступающим в систему URL которые подходят под правила разбора устанавливать $score = s_{fit}$ и s_n для остальных.

4.1.3 Особенности реализации

Данная функциональность была реализована при помощи двух плагинов. Первый проверяет является ли URL "полезной", и добавляет необходимую информацию в *crawldatum* (в *crawldb* хранятся записи вида $\langle URL, crawldatum \rangle$, где *crawldatum* — различная дополнительная информация об URL), а второй выставляет ранг согласно *crawldatum* и настроек приложения. Такое разбиение было сделано потому что информация о том, что url является "полезной", представляет собой интерес и без использования данного ранжирования (например, это было использовано при получении статистики).

SkaiScoringMeta

SkaiScoringMeta — плагин являющийся расширением к точке ScoringFilter, выделяющий "полезные" URL. Для хранения дополнительной информации в *crawldatum* используется поле *metaData*, которое представляет из себя набор пар вида $\langle key, value \rangle$ Для сохранения индикатора "полезности" в мета данные *crawldatum* добавляется пара $\langle skai.scoring.fit, true \rangle$.

Для определения "полезности" URL используется кэш с регулярными выражениями, загружаемыми из базы данных при инициализации. Поскольку таких регулярных выражений может быть много (хотя бы по одному на сайт) и они разбиты по доменам, для сокращения времени проверки каждой URL используется хэш вида $\langle domainname, regex^{+} \rangle$, таким образом время проверки URL практически не зависит от числа доменов.

Так же, для уменьшения числа проверок, метаданные добавляются не в момент первого обнаружения ссылки системой (сразу после разбора веб-страницы), а только в момент добавления уникальной URL в *crawldb*, что уменьшает число проверок приблизительно в 10 раз, так как в среднем только 9% исходящих со страницы ссылок не были ранее известны.

SkaiScoring

SkaiScoring — плагин являющийся расширением к точке ScoringFilter, выставляющий *score*. Для ссылок только что попавших в систему, зависимости от наличия мета тега, выставляет значения *score* либо s_{fit}, либо s_n, где коэффициенты s_{fit} и s_n получаются из файла конфигурации Nutch.

4.1.4 Результат

При использовании стандартных ScoringFilter'ов в среднем за цикл, из всех скачиваемых ссылок "полезных" скачивалось порядка 10% (В качестве пространства поиска использовалось 40 крупнейших СМИ, на каждой итерации выбиралось 20000 ссылок, "полезными" признавались непосредственно новости). А при использовании данных плагинов уже на ранней стадии сборки порядка 80% ссылок были "полезными" (Рис. 4.1). Благодаря предложенным оптимизациям время цикла практически не изменилось по сравнению с временем работы в стандартной конфигурации. Таким образом было получено увеличение эффективности в восемь раз по сравнению с базовой реализацией Nutch.

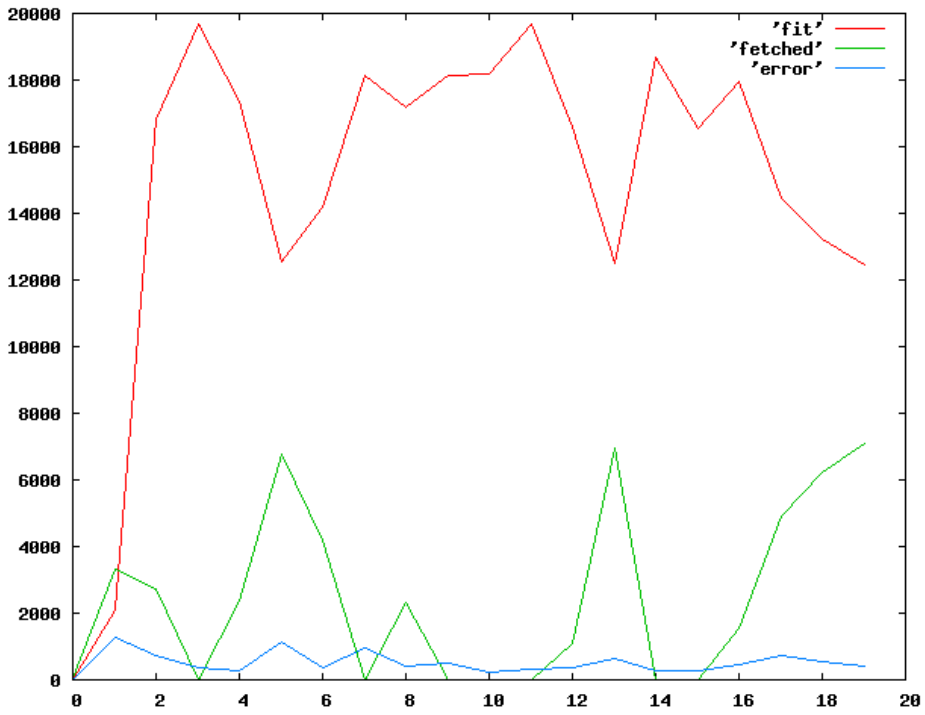

Рис. 4.1: График зависимости числа ссылок от номера итерации. fit—число полезных ссылок, fetched—число скачанных ссылок не являющихся полезными, error—число ошибок при скачивании

4.2 Работа с индексом

При росте размера индекса прамопропорционально растет время используемое для слияния полученного из сегмента индекса с основным. Этот этап не работает через MapReduce и фактически выполняется на одной машине, что приводит к простою

кластера. Для индекса содержащего 2 миллиона документов (70GB) только копирование индекса из HDFS в локальную файловую систему и обратно занимает больше двух часов (при скорости локальной сети 20Mb/s), а общее время работы достигает 11 часов (2×2.5 GHz 2007 Xeon, 1.7GB RAM, скорость последовательного доступа к диску 50MB/s). При размере *crawldb* в 80 миллионов ссылок и генерации сегмента в 80 тысяч URL на слияние индека уходит 68% от общего времени цикла (на кластере из трех узлов).

4.2.1 Lucene

Lucene — библиотека полнотекстовго поиска. Индекс Lucene построен по принципу обратного файла (инвертированного индекса). Сам индекс состоит из сегметнов, каждый из которых грубо говоря представляет отсортированный список термов. Для ускорения поиска сегменты "склеиваются" в один. Операция "склеивания" представляет из себя этап слияния из сортировки слиянием, и производится за время прямопропорциональное сумме размеров склеиваемых сегментов. По-умолчанию Nutch хранит индекс в одном сегменте, к которому на каждой итерации "приклеиваются" новые сегменты индекса, таким образом за каждую итерацию весь индекс перезаписывается.

Было рассмотрено несколько подходов к решению данной проблемы:

- **Асинхронное обновление** — выделяется отдельный поток, который находит все новые сегменты, скачивает их и независимо от процесса основной сборки и обновляет индекс. При этом время на слияние индексов не влияет на скорость скачивания документов, однако, уже при размере индекса в 5.3 миллионов документов индекс будет обновляться только один раз за сутки, из-за чего новости за последние 30 часов никогда не появятся в индексе (время на слияние+время цикла).

- **Изменение политики слияния** — хранение индекса в одном сегменте требует перезаписи всего инедса при каждом обновлении. При этом если вообще не "склеивать" сегменты, существенно падает время поиска в индексе, поскольку приходится искать терм в большом числе словарей. Необходимо разработать алгоритм который позволял бы контролировать число сегментов, при этом не перезаписывая основную часть индекса.

Приблизительно, время выполнения запроса можно оценить следующим образом: $t_q = \sum_{s \in S} td_s + ti$, где td_s это поиск термов в словаре сегмента s, а ti — время на пересечение списков документов соответствующих термам. Причем ti прямопропорционально количеству документов соответсвующих термам, а $td_s = O(\log n_s)$, где n_s — размер словаря сегмента, который практически не зависит от размера сегмента, и можно считать константой. Само время "склеивания" сегментов прямопропорционально их размеру, из чего следует что эффективнее всего склеивать между собой самые маленькие сегменты.

4.2.2 Алгоритмы

Асинхронное обновление

Поскольку появление новых сегментов происходит один раз за несколько часов, было решено создать процесс, который через определенные промежутки времени проверяет наличие новых сегментов и скачивает их стандартными средствами Hadoop в локальную файловую систему клента работающего с индексом, после чего сообщает клиенту о необходимости подключить новые сегменты.

Политика слияния

Индексам присваивается некий уровень l, так же известно максимальное количество сегментов на каждом уровне N_l. Всем только что скачанным сегментам присваивается $l = 0$, в тот момент когда количество индексов на каком-то из уровней превышает N_l они "склеиваются" в один сегмент уровня $l + 1$.

4.2.3 Особенности реализации

Асинхронное обновление

Как только заканчивается индексация нового сегмента, к имени сегмента добавляется префикс *ready*. Процесс просматривает папку с сегментами в HDFS и при наличии сегментов с такими префисками скачивает их и убирает префикс, после чего сообщает клиенту HTTP запросом о необходимости подключить новый сегмент. При таком подходе аварийное завершение процесса обновления не влечет никаких последсвий — достаточно просто заново его запустить, и отключение основного цикла сборки

никак не сказывается на процесс обновления (кроме того, что не появляется новых сегментов индекса).

Политика слияния

Индексы были разбиты на следующие уровни:

1. new — новые сегменты;
2. daily — сегменты за день;
3. weeky — сегменты за неделю;
4. monthly — сегменты за 4 недели;
5. 1-level — сегменты за 4 месяца (4×4 недели);
6. k-level — сегменты получаемые при слиянии четырех k-1-level.

В конце каждого дня все новые индексы объединяются в дневные, как только их набирается 7 — они склеиваются в недельные и.т.д. Сама функциональность была реализована как *Groovy* скрипт.

4.2.4 Результат

Благодаря асинхронному обновлению время цикла уменьшилось на 68%, однако возрасла нагрузка на сервер осуществляющим поиск, так как слияние индексов стало происходить на нем, однако, за счет новой политики слияния, каждый документ в индексе перезаписывается не более 5 раз за год, что дает в сумме не более 80 часов в год при конечном индексе в 10 миллионов документов.

4.3 Раннее удаление дубликатов

Для выдачи качественных результатов поиска важно что бы в индексе не было дубликаов. Под дубликатом понимаются:

- web-страницы с одинаковым URL;
- web-страницы с одинаковым текстом и доменом.

4.3.1 Реализация в Nutch

В Nutch удаление дубликатов реализовано в виде серии MapReduce задач и работает достаточно быстро — на дедубликацию как правило уходит не больше 3% от общего времени. Однако у этого метода есть ряд существенных недостатков.

- Дедубликация происходит уже после индексации документов. При этом дубликаты тоже индексируются, что занимает существенно большее время (порядка 10% от общего времени цикла).
- Для эффективной работы MapReduce желательно что бы индекс хранился в распределенной файловой системе, а для быстрого поиска по индексу необходимо наличие копии в локальной файловой системе. Таким образом нужно постоянно хранить согласованные копии индекса в двух файловых системах.
- Для удаления дубликатов достаточно знать только набор md5 хэшей и URL, а приходится читать весь индекс.

4.3.2 Алгоритм

Поскольку для удаления дубликатов достаточно небольшого количества данных (URL + md5 для каждого документа) было решено использовать Key Value базу данных. После получения текста статьи проверяется есть ли в базе данных такой хэш или URL, если есть, то страница отбрасывается, иначе в базу добавляются данные о новой странице. Поскольку сервер с базой данных должен справляться с нагрузкой с целого кластера был произведен анализ различных реализаций Key Value СУБД.

4.3.3 Сравнение Key Value СУБД

Были рассмотрены следующие реализации:

- Memcached[1] — система кэширования данных разработанная для ускорения веб приложений. Изначально была создана для LiveJournal в 2003 году. Данные хранятся только в памяти, что позволяет осуществлять быстрый доступ, но при этом необходимо отдельно заботится о сохранении данных на диск.

- MongoDB[2] — документо-ориентированная Key Value СУБД. MondoDB позиционируют себя как промежуточное звено между простейшими Key Value хранилищами и реляционными СУБД.

- Project Voldemort[3] — распределенное отказоустойчивое Key Value хранилище. Одним из приемуществ данной системы является отсутсвие единой точки отказа (Single point of failure). Project Voldemort используется в качестве хранилища данных в LinkedIn[4].

- Tokyo Cabinet[5] — встроенное Key Value хранилище. Поддерживает как хранение данных только в памяти, так и на диске. Для удаленного доступа используется Tokyo Tyrant[6]. Протокол Tokyo Tyrant почти полностью совместим с Memcached.

Сравнение данных СУБД представлено в таблице 4.1.

Таблица 4.1: Сравнение Key Value СУБД.

Название	Memcached	MongoDb	Project Voldemort	Tokyo Cabinet
Операции put/ms	4.03	6.65	1.21	3.25
Операции get/ms	4.54	3.05	1.01	4.39
Устойчивость	-	+	+	+
Распределенность	+	+	+	-
Репликация	-	+	+	-
Модель данных	bin	object	object	bin

[1]http://memcached.org/
[2]http://mongodb.org/
[3]http://project-voldemort.com/
[4]http://www.linkedin.com/
[5]http://fallabs.com/tokyocabinet/
[6]http://fallabs.com/tokyotyrant/

Измерение производительности

Так как время выполнения put и get запросов является критическим для нашей системы, было решено разработать утилиту[7] измеряющую производительность различных СУБД на данных, с которыми потом будет осуществляться работа. Результаты данных тестов представлены в строчках put и get таблицы 4.1.

Тестирование происходило по следующему сценарию:

1. в базу добавляются все URL из рабочего индекса (5688210 ссылок);
2. проверяется наличие всех URL из индекса по одному из сегметнов (52583 ссылок).

Добавление и проверка производятся в 10 потоков. Сервер и клиент находятся на разных хостах (1.2 GHz 2007 Xeon, 1.7GB RAM, скорость последовательного доступа к диску 50MB/s), время ping между хостами — 0.6ms.

4.4 Черные списки

При долгой работе сборки значительно увеличивается размер базы ссылок crawldb.

Crawldb используется практически на всех этапах, время которое тратится на обработку crawldb линейно зависит от её размера. Время на выполнения всего цикла зависит от размера crawldb и количества ссылок выбираемых для скачки. Хотя само время скачивания очень сильно зависит от канала и сайтов с которых идет скачка, его можно представить в виде:

$$t_c = n_c * c_1 + n_g * c_2$$

Где n_c - число записей crawldb, n_g - число выбираемых ссылок, а t_c общее время цикла. В реальных условиях $c_1/c_2 \approx 0.0013$, таким образом при $n_c = 1000000$ и $n_g = 20000$ на работу с crawldb тратится порядка 30% времени.

Значительная часть ссылок хранимых в базе не представляют никакого интереса. Например, не имеет смысла хранить ссылки на документы неподдерживаемого формата (*.mp3, *.avi, *.jpg), или все ссылки на раздел с форумом. За отсечение ненужных URL в Nutch отвечает система URL фильтров.

[7] http://github.com/volkov/kvstorage-test

4.4.1 Фильтры

Фильтрация ссылок в Nutch реализована через плагины с точкой расширения URLFilter. Фильтры используются в момент добавления ссылки в базу ссылок и во время её обновления. В Nutch реализовано несколько плагинов для фильтрации:

- PrefixURLFilter — фильтрация по префиксу URL. Служит, как правило, для ограничения сборки определенными доменами.
- SuffixURLFilter — фильтрация по суффиксу URL. Используются для ограничения форматов файлов.
- RegexURLFilter — фильтрация по регулярному выражению.

Каждый из плагинов допускает “пропускающие” и “исключающие” правила, которые берутся из файлов конфигурации.

4.4.2 Создание фильтров

Фильтры ограничивающие домены и форматы тривиально создаются из списка доменов и расширений поддерживаемых форматов. Это дает самое общее ограничение области ссылок, при этом еще остается большое число ссылок не являющихся “полезными”. Например, если мы хотим скачивать только новости, то нам совершенно не интересно знать что на в том же домене располагается форум, или какие-либо статьи. Так же часто присутствует большое число технических ссылок, например некоторые сайты делают внешние ссылки как редиректы с собственного домена. Основную сложность для ручного создания фильтров представляют как раз технические разделы сайта.

Необходимо создать метод с низкой вероятностью ошибок первого рода (отклонение разделов с полезной информации), эффективно отсекающий нежелательные URL. Проблема заключается в том, что необходимо оставить не только все “полезные ссылки”, но и те страницы без которых не все “полезные” ссылки достижимы из корневой страницы домена.

Рассмотрим в качестве примера сайт lenta.ru. Все новости находятся по ссылкам вида 4.1, а архив новостей, без которого мы не сможем получить все новости, находится в разделе 4.2 который не должен попасть под фильтры.

$$http://lenta\.ru/news/\d\{4\}/\d\{2\}/\d\{2\}/\w+/ \tag{4.1}$$

$$http://lenta\.ru/\d\{4\}/\d\{2\}/\d\{2\}/ \quad (4.2)$$

Формальные требования к автоматической генерации:

- ни одна “полезная” ссылка не должна попадать под фильтры;
- после применения фильтров все “полезные” ссылки должны быть достижимы из корневой ссылки домена;
- значительное число остальных ссылок должно попасть под фильтры.

Задачу построения фильтров можно разбить на две части:

- получение из crawldb данных в удобной для анализа форме;
- непосредственно анализ выделенных данных.

4.4.3 Получение статистики по crawldb

Для анализа crawldb в nutch используется утилита readdb которая умеет получать статистику по базе. Наибольший интерес представляет такой предоставляемый ею набор метрик, как число ссылок определенного статуса, разбитые по доменам. Что бы стало возможным нахождение бесполезных разделов, необходимо реализовать возможность получения статистики не только по доменам, но и по крупным их частям.

Основная идея заключается в том, что бы получить записи вида $\langle domain, prefix, metrics\rangle$ где:

- domain — домен к которому относится запись.
- prefix — префикс url к которому относится запись.
- metrics — набор пар вида $\langle state, count\rangle$, где $count$ — это число документов с данным префиксом на данном домене, в состоянии $state$ (например: новая ссылка, скачанная ссылка, “полезная ссылка”).

По подобным записям в дальнейшем достаточно легко делать выводы о разделах ресурсов.

Алгоритм

Поскольку данная задача работает с crawldb, размер которой может быть очень большим, алгоритм необходимо представить в виде MapReduce задачи.

Map На этапе map по url и $crawldatum$ получается множество пар $\langle prefix, info\rangle+$, где $prefix$ это префикс url вместе с доменом, а $info$ это пара вида $\langle state, 1\rangle$

$$\langle url, crawldatum\rangle \to \langle prefix, info\rangle+$$

Сначала из url неким способом получается множество префиксов S_p, например:

$$http://lenta.ru/2010/05/09/ \to$$

$$http://lenta.ru/2010/05/09,$$

$$http://lenta.ru/2010/05/,$$

$$http://lenta.ru/2010/,$$

$$http://lenta.ru/;$$

Далее в зависимости от свойств url описанных в $crawldatum$ создается множество метрик S_m, например если ссылка была "полезной" но не была еще скачана, то

$$S_m = \{\langle fit, 1\rangle, \langle unfetched, 1\rangle\}$$

Затем все пары из $S_p \times S_m$ попадают в выходной поток. Перед этапом reduce промежуточные записи сортируются, и важно что бы их было не слишком много, то есть url следует разбивать на наименьшее число префиксов, но так, что бы основные разделы домена могли быть сопоставлены какому-нибудь префиксу, а множество метрик содержало только одно значение (это верно если url не может находится сразу в нескольких состояниях).

Reduce На этапе reduce метрики префиксов суммируются и отбрасываются незначительные префиксы

$$\langle prefix, \langle state, 1\rangle+\rangle \to \langle prefix, metricvector\rangle$$

$Metricvector$ представляет из себя вектор из пар $\langle state, nurl\rangle$, где $nurl$ — число url с состоянием $state$. Незначительными признаются метрики где сумма $nurl$ достаточно мала. Такие префиксы отбрасываются, а остальные сохраняются в базу данных.

4.4.4 Реализация

В ходе реализации был изменен класс CrawlDbReader, отвечающий за получение статистики, написаны классы для map и reduce.

CrawlDbExtendedStatMapper

CrawlDbExtendedStatMapper — класс имплементирующий стандартный интерфейс Hadoop — Mapper, который служит для создания собственных этапов map. Для разбиения url на префиксы был использован UrlSplitter, который разбивал url на не более чем n_s префиксов по символу "/", так же как и в примере с lenta.ru. Таким образом $|S_p| \leqslant n_s$. В качестве метрик использовались следующие показатели:

- unfetched —документ по ссылке не был скачан;
- fetched — документ по ссылке была скачана;
- fit — ссылка была признана "полезной".

Первые два показателя брались непосредственно из статуса *crawldatum*, а последняя из мета данных, которые были проставлены с помощью SkaiScoringMeta. Таким образом $|S_m| \leqslant 2$. В результате, число записей попадающих в выходной поток $n \leqslant 2 \cdot n_s \cdot n_{url}$, где n_{url} — число записей в crawldb.

CrawlDbExtendedStatReducer

CrawlDbExtendedStatReducer — класс имплементирующий стандартный интерфейс Hadoop — Reducer, который служит для создания собственных этапов reduce.

На этапе reduce суммируются все показатели для префикса и если $n_{unfetched} + n_{fetched} < n_s$ префикс отбрасывается. Для совместимости со стандартным CrawlDbReader'ом и уменьшения нагрузки на базу данных, префиксы не сразу добавляются в базу данных, а выводятся в файл.

Добавление в базу данных

После отработки MapReduce работы, её результат считывается из файла, после чего все записи добавляются в базу данных одним запросом. Так же в базу передается время в которое статистика была создана.

Получать статистику можно достаточно редко — например один раз в 5-20 циклов, таким образом, время на сбор статистики не оказывает существенного влияния на эффективность.

4.4.5 Результаты

Для crawldb с 10 000 000 url время получения статистики составляет порядка 15% от общего времени работы цикла (для скачивания выбирается 20 000 ссылок). Таким образом, при создании статистики каждый двадцатый цикл, потеря времени составляет порядка 0.75%, которой можно пренебречь.

4.5 Генерация фильтров

При создании алгоритма были сделаны следующие предположения:

- все “полезные” можно ограничить некотором числом разделов, не зависящем от числа документов. (под разделом понимается некий префикс url)
- архив документов находится в определенном разделе (под архивом понимаются документы с ссылками на “полезные” документы)
- архив достижим из корневого документа
- документов архива меньше чем “полезных”

4.5.1 Алгоритм

Сам алгоритм, пользуясь некоторой эвристикой, признает некоторые префиксы ненужными, из которых потом создаются фильтры. Алгоритм выбирающий префиксы согласно сделанным предположениям выглядит так:

1. Выбирается вся статистика для конкретного домена.
2. Домены для которых $u_d < n_d$ далее не рассматриваются.
   - u_d — число “полезных” ссылок во всем домене
   - n_d — пороговое значение, введенное для того, что бы не начать создавать фильтры до того, как будут получены ссылки на архив.
3. По префиксам для которых $u_p = 0$ и $s_p > u_d$ создаются исключающие правила.
   - u_p — число “полезных” ссылок с данным префиксом
   - s_p — общее число известных ссылок с данным префиксом

   Сам по себе префикс уже представляет из себя префиксный фильтр.

4.5.2 Реализация

Данная функциональность была реализована в виде отдельного приложения запускающегося сразу после создания статистики. Поскольку пользовательские фильтры были организованы на базе RegexURLFilter, для простоты управления автоматически созданные фильтры были тоже реализованы регулярными выражениями.

Была предусмотрена возможность отключения конкретного фильтра таким образом, что бы он не создавался заново. Для этого фильтр не удаляется из базы данных, а становится не активен.

4.5.3 Результат

Качество результатов работы данного алгоритма достаточно сложно оценить, однако на тестовых данных автоматические фильтры достаточно успешно находили бесполезные разделы. Предполагается что существенное увеличение производительности будет получено на позднем этапе сборки - когда большая часть “полезных” документов уже будет скачена.

Заключение

Результаты

- Сделан сравнительный анализ различных open source поисковые роботы (DataparkSearch, AppSeek, mnlGoSearch, Nutch, Hounder, Heritix) и выбрать наиболее подходящий для решения задачи
- Изменено поведение ядра nutch для более эффективной работы с индексом большого объема.
- Проанализированы различные key-value хранилища (Memcached, MongoDb, Project Voldemort, Tokyo Cabinet) и выбрано MongoDb в качестве хранилища для системы удаления дубликатов из индекса
- Разработан и реализован плагин к Nutch для раннего удаления дубликатов
- Разработан и реализован плагин для более эффективного ранжирования ссылок для новостных сайтов
- Разработана и реализована система для автоматического создания url фильтров
- Измененный поисковый робот протестирован на реальных данных.

Литература

[1] Junghoo Cho, Hector Garcia-Molina, Lawrence Page: Efficient Crawling Through URL Ordering, 1998.

[2] Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, 2004.

[3] Tom White: Hadoop: The Definitive Guide, 2009

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System, 2003.

[5] Martin Ester, Matthias Groß, Hans-Peter Kriegel: Focused Web Crawling: A Generic Framework for Specifying the User Interest and for Adaptive Crawling Strategies, 2001.

[6] Serge Abiteboul, Mihai Preda, Grégory Cobena: Adaptive On-Line Page Importance Computation, 2003

[7] Пименов Александр, Hadoop Nutch и Lucene v3, 2009.

[8] D Cutting, Nutch: an Open-Source Platform for Web Search

[9] E Hatcher, O Gospodnetic, Lucene in action

[10] R Khare, D Cutting, K Sitaker, A Rifkin, Nutch: A flexible and scalable open-source web search engine

[11] T White, Hadoop: The Definitive Guide