Министерство образования и науки российской федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования

**«Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина »**

**Институт математики и компьютерных наук
Департамент математики, механики и компьютерных наук
Кафедра алгебры и дискретной математики**

Распределенная отказоустойчивая система запуска заданий и ее использование для автоматического построения индексов для полнотекстового поиска

Допустить к защите

Зав. кафедрой:
д. ф.-м. н., проф.
Волков М. В.

Выпускная квалификационная работа
студента 4 курса
Вяцкова М. В.

Научный руководитель:
м.н.с.
Хворост А.А.

Екатеринбург
2016

Реферат

Выпускная квалификационная работа по теме «Распределенная отказоустойчивая система запуска заданий и ее использование для автоматического построения индексов для полнотекстового поиска» содержит 32 страницы текстового документа, 12 использованных источников и 3 изображения.

ПОЛНОТЕКСТОВЫЙ ПОИСК, РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, INFORMATION RETRIEVAL, LUCENE, ELASTICSEARCH.

В данной работе описывается процесс проектирования и реализации системы инструментов для создания индексов для полнотекстового поиска. Кроме того, вместе с каждым созданным индексом предоставляется механизм для непрерывной индексации данных из внешнего первичного источника. Обязательным требованием к создаваемой системе было, чтобы все компоненты обладали свойствами надежности, отказоустойчивости и масштабируемости.

В процессе проектирования оказалось, что не существует готового решения для задачи индексации, которое бы обладало необходимыми свойствами. По этой причине было принято решение самостоятельно разработать такую систему, удовлетворяющую всем поставленным требованиям и, кроме всего прочего, расширяемую и переиспользуемую.

Результатом работы является надежная и отказоустойчивая инфраструктура для встраивания полнотекстового поиска в существующие программные продукты.

ОГЛАВЛЕНИЕ

Введение

В любой достаточно востребованной современной системе генерируется и хранится огромное количество данных. Кроме того, почти всегда необходимо иметь возможность быстро получать доступ к необходимой в данный момент информации. Чтобы обеспечить подобную функциональность, необходимо использование различных алгоритмов и структур данных.

Структуры данных, предназначенные для выполнения поисковых запросов, обычно называются поисковыми индексами, а объекты, по которым осуществляется поиск — документами. Область научного знания, занимающаяся вопросами эффективного поиска называется Information Retrieval (IR).

Особенно важным случаем является так называемый полнотекстовый поиск, когда документы содержат текст на естественном языке. Естественный язык — используемый человеком для общения, например русский или английский. Важным этот случай является из-за большого количество задач, в которых он встречается. Поиск на форуме, в почтовом ящике, в интернете. Все эти задачи требуют реализации полнотекстового поиска.

Помимо всего прочего, большинство существующих систем, взаимодействующих с пользователем, постоянно пополняются данными. Ежесекундно приходят миллионы писем, публикуются тысячи статей и создаются сотни станиц в интернете. Для того, чтобы индексы содержали актуальные данные, необходим непрерывный процесс индексации — переноса новой информации в индекс.

В данной работе описывается процесс создания системы инструментов для внедрения полнотекстового поиска с налаженным процессом индексации в конечное приложение.

Глава 1. Постановка задачи

На момент постановки задачи не существовало известного автору универсального, масштабируемого и отказоустойчивого решения для внедрения полнотекстового поиска с налаженным процессом индексации в существующее приложение, не существует и до сих пор. В результате работы хотелось насколько это возможно приблизится к появлению такого продукта.

Первая проблема касается почти всех существующих решений и состоит в том, что настройка и обслуживание требует достаточно много ручной работы. Например, популярный набор инструментов ELK требует самостоятельной настройки и отслеживания работоспособности каждого индексатора. Будучи весьма удобным в использовании для одного приложения, данный инструмент порождает множество проблем при масштабировании в 10, 100 раз.

Вторая проблема заключается в отсутствии у многих решений способности автоматически горизонтально масштабировать индексы. Отсутствие масштабирования ограничивает сверху размер поисковых структур и запасы по производительности при неизменных доступных вычислительных мощностях. Индекс размером в 20ТБ проблематично даже физически разместить на одном сервере, не говоря уж о манипуляциях над таким объемом данных. В это же время кластерные решения обрабатывают и большие структуры.

Третья проблема, свойственная некоторым из существующих решений, состоит в навязывании определенной платформы для разработки конечного продукта. Однако задача часто состоит во внедрении полнотекстового поиска в уже существующее приложение и навязывание платформы недопустимо.

Таким образом, необходимо было создать решение, которое обладало бы большой степенью автоматизации, хорошо масштабировалось бы на большие объемы данных и большое количество использований и было бы дружелюбно к приложению на любой платформе. При этом получившаяся система должна была быть отказоустойчивой и достаточно производительной.

Глава 2. Распределенные системы

Введем базовые понятия из области распределенных систем

Глава 2..1 Модели косистентности

Прежде всего необходимо ввести понятие "истории как некоторой последовательности событий, упорядоченной в абсолютном времени, которые совершались с системой. При этом события не происходят мгновенно, каждое событие имеет время вызова (invocation) и время завершения (response).

Для примера упростим на время модель выше и скажем, что операции все-таки происходят мгновенно. Теперь рассмотрим логический поток управления в программе, который работает с одной переменной. Историей в данном случае будет например последовательность операций

$$(write(3), read(3), write(5))$$

где $write(x)$ обозначает запись в переменную числа x, а $read(x)$ означает чтение из переменной числа x. Также историей будет последовательность

$$(write(10), read(42))$$

Несмотря на то, что формально последний пример является историей, интуитивно мы понимает, что такого не могло произойти. Множество допустимых, корректных историй, заданное явно или с помощью правил, называется моделью консистентности. Чем больше возможных историй допускает модель, тем она слабее, и наоборот, чем меньше возможных историй допускает модель, тем она строже (или сильнее, что означает то же самое). Введем несколько самых строгих прикладных моделей консистентности.

Глава 2..1.1 Линеаризумость

Допустим, что в системе есть единственное глобальное состояние, с которым взаимодействует каждый процесс. Кроме того, все операции, допустимые в системе, являются атомарными, то есть происходят не одновременно с другими операциями. Кроме того, логично будет допустить, что операции выполняются не раньше, чем она была вызваны и не позже, чем был получен ответ. Истории, допустимые в такой модели называются линеаризуемыми. Можно представить, что все операции выполнялись мгновенно в какой-то момент времени между своим вызовом и завершением, образуя линейную историю атомарных изменений единого состояния.

Из таких временных рамок следует, например, что прочитав состояние после успешно завершенной записи, мы обнаружим не более старое состояние, чем было записано. Такие ограничения являются естественными для понимания и поэтому линеаризуемость выбрана за основу многих параллельных и распределенных программных конструкций. Например, volatile переменные в Java, атомы в Clojure и переменные в javascript являются линеаризуемыми.

На самом деле "единое глобальное состояние"не обязательно должно быть расположено на одном вычислительном узле. Более того, операции не обязательно должны быть атомарными. Состояние может быть распределено на многих узлах, его изменение может состоять из нескольких шагов. Важно лишь то, чтобы извне история применения операций казалась эквивалентной истории атомарных изменений единого состояния. Часто, линеаризуемая система состоит из более мелких частей, каждая из которых линеаризуема и свою очередь состоит из линеаризуемых частей, и так далее до линеаризуемых на аппаратном уровне примитивов.

Глава 2..1.2 Сериализуемость

Сериализуемость во многом похожа на линеаризуемость, но в данном случае нет ограничения на единое состояние. Важно лишь то, чтобы история

операций была эквивалентна некоторому последовательному атомарному выполнению. Кроме того, времена вызова и завершения не обязаны коррелировать со временем действительного выполнения.

С одной стороны, данная модель консистентности достаточно слабая, так как разрешает множество перестановок операций, с другой стороны, она отсекает большой класс историй за счет требования линейности. Для примера, история операций

$$(read(3), writeif(2,1), writeif(3,2), write(1))$$

допускает единственный возможный порядок выполнения (операция $writeif(x,y)$ записывает в состояние x, если в данный момент оно y).

Недостатком сериализуемости самой по себе является то, что из-за отсутствия ограничения на время выполнения операции, теоретически может произойти так, что какое-то изменение будет бесконечно откладываться на будущее и никогда не исполнится. На практике это приведет к фактической потери транзакции, что, конечно же, мало когда допустимо.

Глава 2..1.3 Строгая сериализуемость

Чаще, когда говорят о желаемой модели консистентности, имеют в виде строгую сериализуемость. Сохраняя все свойства сериализуемости, она также накладывает ограничение на временные рамки выполнения операций, обеспечивая важное почти во всех приложениях свойство гарантии прогресса. Таким образом, все успешно дошедшие до системы операции будут выполнены за конечное время.

Что интересно, на практике нередко ограничиваются более слабой моделью консистентности и то, что подразумевается под сериализуемостью на самом деле не дает таких же сильных гарантий.

Глава 2..2 CAP-теорема

В 2000 году на конференции PODC Эриком Брювером была выдвинута так называемая гипотеза Брювера, утверждающая, что невозможно создать систему, которая была бы одновременно консистентная (C, consistency), доступна (A, availability) и устойчива к разделению сети (P, partition tolerance)[2]. Позже, в 2002 была выпущена работа за авторством Сета Гильберта и Нэнси Линч[3], в которой формально было сформулирована и доказана следующая теорема:

Теорема 1 *В условиях асинхронной сети невозможно создать объект, поддерживающий операции чтения и записи, который бы гарантировал следующие свойства:*

- *Доступность*
- *Атомарная консистентность*

для всех совершенных запросов (включая те, в которых сообщение было потеряно).

Которая сейчас и называется CAP-теоремой. Для понимания, как это соотносится с исходными понятиями, необходимо формально ввести, что такое доступность, консистентность и устойчивость к разделению сети.

Глава 2..2.1 Консистентность

Атомарная консистеность понимается как линеаризуемость, упомянутая выше, то есть множество допустимых историй, каждую из которых можно представить в виде последовательности атомарных изменений единого состояния, причем все изменения происходили мгновенно в момент времени между вызовом и завершением.

Глава 2..2.2 Доступность

Доступность распределенной системы означает, что на любой запрос, полученный работающим узлом в кластере должен быть сгенерирован ответ за конечное время. Это означает, что алгоритм, положенный в основу работы распределенной системы не должен допускать бесконечно долгого ожидания результата. Сообщение об ошибке не считается ответом, то есть система, которая может бесконечно долго отвечать ошибкой на запрос не является доступной в данном смысле.

Глава 2..2.3 Устойчивость к разделению сети/Асинхронная сеть

Важно понимать, что данный пункт не накладывает ограничений на алгоритм в основе распределенной системы. P в аббревиатуре CAP хоть и расшифровывается как устойчивость к разделению сети, но на самом деле означает то, что в сети допускается любое количество потерянных и задержанных сообщений между узлами.

Легче всего понять это ограничение, если заметить, что почти любая CP система также является CA, так как в условии отсутствия задержек и потерь сообщений несложно добиться доступности. Если бы P было исключительно свойством алгоритма, то данный факт эффективно показывал бы наличие CAP системы, что противоречит теореме.

Также немаловажным выводом является то, что алгоритм, лежащий в основе P системы может рассчитывать только на полученные сообщение и локальные для узла параметры.

Глава 2..3 В применении к задаче

Исходя из вышесказанного, можно понять, что каждый раз при проектировании системы нам необходимо выбирать, какому классу будет принадлежать наша система, CP или AP. CA не является сколько-нибудь значимым вариантом, так как в реальных системах неизменно случаются разрывы в

сети, отказы вычислительных узлов и тому подобное.

Так как приложения в основном имеют дело со сгенерированными пользователем данными, крайне нежелательно было бы их потерять. Отсюда естественным выводом будет всегда в данной работе отдавать предпочтение CP системам.

Существуют сферы и приложения, для которых необходимы AP системы, например часто таковыми являются распределенные системы кеширования. Однако в данном случае подобные возможности не столь ценны.

Глава 3. Полнотекстовый поиск

Глава 3..1 Введение

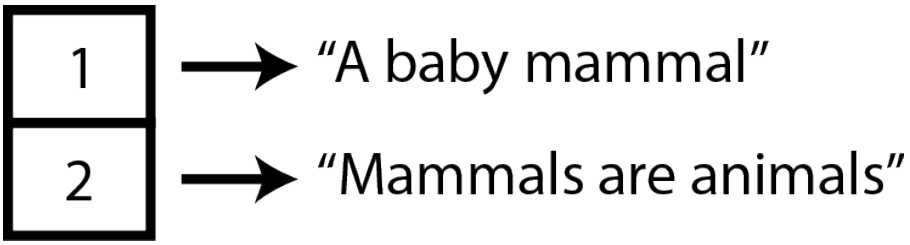
Оговоримся заранее, что интересующие нас данные представлены в виде **документов**, каждый из которых имеет набор **полей**, некоторых свойств, значения которых принадлежат некоторому домену. Для удобства легче всего считать, что все поля — байтовые строки. Это представление удобно тем, что естественно для данных, представимых внутри компьютера и кроме того, лексикографический порядок байтовых массивов соответствует естественному порядку чисел, дат и лексикографическому порядку строк например в представлении UTF-16. Также удобно считать, что одному и тому же полю может соответствовать несколько значений. Пара (поле, значение) называется **термом** и обладает семантикой слова в тексте.

Любой документ можно привести к такой форме. Даты например при этом переводятся в числа, а числа в байтовые строки фиксированной длины, соответствующей размерности числа. Процесс приведения текста чуть более сложен и включает в себя две стадии — токенизирование, когда текст превращается в набор слов-токенов, каждый из которых затем анализируется, то есть приводится к нормальной форме. Для естественных языков это может быть например выделение корня из слова. Морфологический анализ слов является отдельной задачей, которая используется при построении полнотекстовых индексов, но которую мы не будем подробно затрагивать далее в тексте.

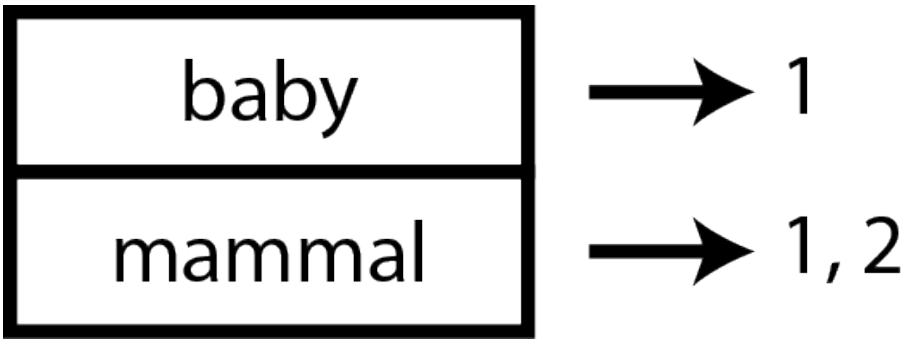
После того, как каждый документ приведен к форме набора термов, можно составить так называемые прямой и обратный индексы. Прямой индекс, который мы можем во многих формах наблюдать в реальности, отображает документ в виде какого-то уникального идентификатора, например абстрактного номера, в набор термов. Обратный индекс нужен для противоположной задачи, отобразить терм в набор документов, в которых он встречается. После того, как мы построили эти индексы, мы может обрабатывать

запросы, включающие в себя термы, их отрицание и логические связки.

Рис. 1, Пример прямого и обратного индексов



(a) Прямой индекс



(b) Часть обратного индекса

Такой набор возможностей уже позволяет достаточно полно обрабатывать специальные и полнотекстовые запросы, то есть запросы, также сформулированные на естественном языке. С такими запросами необходимо проделать ту же процедуру, что и с исходными документами, преобразовав их к набору термов, затем по каждому терму получить множество документов, которые соответствуют терму, и после этого применить некоторые теоретико-множественные операции к получившимся наборам документов.

Отдельная задача состоит в ранжировании выборки, то есть сортировке формально подходящих документов по убыванию релевантности документа запросу. В данной области опять же существует множество подходов. Обычно все подходы сводятся к подсчету некоторой численной метрики для каждого документа, зависящей от параметров запроса и индекса. По этой метрике и производится сортировка. Примерами может служить метрика TF-IDF, берущая свое начало в работах Питера Луна[4] и Карена Джоунса[5], или PageRank[6], разработанная на заре поискового движка Google, особенная

тем, что использует информацию, не связанную напрямую с содержанием текста.

Полнотекстовые запросы и запросы, вручную собранные из термов и логических связок не исчерпывают возможности поисковых систем. Также существуют нечеткие поисковые запросы, которые могут находить по похожим термам; поисковые запросы, сформулированные на языке регулярных выражений; поисковые запросы по фразе, когда порядок слов и то, что они идут в тексте последовательно, имеет значение. Каждый перечисленный вид поисковых запросов является по отдельности большой задачей достойной отдельного исследования.

Важным термином, связанным с полнотекстовым поиском, является так называемый Real Time поиск и Near Real Time (NRT) поиск. Данные термины связаны с тем, что если у нас есть непрерывный процесс генерации и соответственно индексации новых данных, то может быть задержка между тем, как мы проиндексировали документ в системе и тем, как мы теоретически можем его получить в составе поисковой выдачи. Это может быть связано с техническими особенностями построения индекса. Real Time поиск гарантирует, что такой задержки не существует и после того, как мы получили так или иначе подтверждение о том, что запрос на индексацию завершился успешно, документ гарантированно можно будет найти соответствующим поисковым запросом. NRT формально не определяется и предполагает, что задержка, хотя и существует, достаточно мала и незаметна с точки зрения пользователя. Свойство RT или NRT очень желательны в системе, так как обеспечивает незамедлительную обратную связь пользователю на его действия над данными в системе.

Глава 3..2 Масштабируемость

Иногда данных в системе настолько много, что они физически не вмещаются на одно устройство, или поток запросов становится настолько высок, что один сервер не справляется с такой нагрузкой, в этом случае применяют

стратегии горизонтального масштабирования, а именно **шардинг** и **репликацию**.

Шардинг — разделение данных на непересекающиеся части, каждая из которых работает независимо, чаще всего независимо по вычислительным ресурсам. Каждый шард представляет собой отдельный индекс, к которому применяется запрос. Результатом применения запроса к шарду — некоторая выборка документов. После того, как запрос выполнился на каждом шарде, результаты объединяются в конечную выборку, которая и является ответом на запрос к шардированному индексу. Чаще всего объединение происходит по метрике релевантности, которая была рассмотрена выше, вследствие чего, при необходимости получения запроса размера N при K шардах, нельзя обойтись без того, чтобы не запросить N наиболее релевантных документов с каждого шарда, так как мы не знаем, какие из них в итоге попадут в выборку. Это в K раз увеличивает количество документов, которые будут извлечены из индекса, переданы по сети и агрегированы на сервере, обслуживающем клиентский запрос.

Таким образом, шардинг позволяет разделить индекс на части меньше размера, каждая из которых является подъемной для одного вычислительного узла. Это позволяет строить индексы колоссальных размеров. В свою очередь данный процесс отрицательно сказывается на производительности системы при обращении к ней пользователей.

Чтобы производить шардинг, каждый документ также должен обладать некоторым внешним по отношению к индексу параметром, в некоторых источниках упоминающимся как роутинг, по которому происходит выбор, в какой шард попадет этот документ. Роутинг не обязательно должен быть уникальным для каждого документа, но он должен обладать двумя важными свойствами. Во-первых, роутинг должен быть равномерно распределен по документам, так как иначе создается дизбаланс в нагрузке и в какой-то момент может получиться так, что при обработке запроса все шарды, кроме одного, самого большого, простаивают. Во-вторых, роутинг не должен ме-

няться при изменении документа, так как иначе изменение может оказаться некорректным.

Репликация — копирование данных на независимые вычислительные мощности. Зачастую применяется в совокупности с шардированием, когда исходных индекс разбивается на несколько независимых частей, каждая из которых реплицируется. Репликация позволяет достигнуть двух важных вещей. Во-первых, используя ту или иную технологию распределения нагрузки между репликами, мы можем получить улучшения производительности системы, за счет того, что запросы к одному и тому же индексу исполняются на разных вычислительных мощностях. Это, например, может компенсировать проседание производительности при шардировании. Во-вторых, репликация данных позволяет обеспечить большую степень отказоустойчивости, об этом будет подробно рассказано ниже.

Однако, с увеличением производительности при поиске происходит уменьшение производительности при индексации, так как нам необходимо каждый раз записывать новые данные не в один индекс, а в каждую реплику, которых может быть 2 и более. Кроме того, наличие реплик может привести к проблемам, известным в области распределенных вычислений как stale reads, когда запросы на разные реплики дают разные результаты, так как запись прошла на одной реплике, но не прошла на другой. Об этом также будет сказано ниже.

Глава 3..3 Отказоустойчивость

В реальном мире компьютеры, на которых запускается программное обеспечение зачастую отказывает в силу тех или иных причин, по мере роста и масштабирования системы вероятность того, что в данный промежуток времени произойдет сбой, растет очень быстро, поэтому важно, чтобы система сохраняла работоспособность даже при условии, что произошел сбой, возможно даже не единственный.

Основная мера по предотвращению такой ситуации — создание распре-

деленной системы без единой точки отказа, когда каждый функциональный узел дублируется на независимых вычислительных машинах. Таким образом, чтобы соответствовать стандартам качества и отказоустойчивости, необходимо, чтобы система, выполняющая поисковые запросы, была распределенной и умела реагировать на экстренные ситуации.

Если шард поискового индекса будет существовать в кластере в единственном экземпляре, то об отказоустойчивости не может идти речи. Выход из строя узла, на котором расположен шард приведет к невозможности обслуживания запросов к данному индексу. Таким образом, краеугольный камень работы поискового движка в условиях распределенности — репликация данных.

Из-за появления структуры данных, распределенной на несколько компьютеров в сети, возникает необходимость в гарантии консистентности системы в смысле корректности ее поведения с точки зрения внешнего наблюдателя. Как мы обсуждали ранее, для нашего применения система должна быть CP, то есть быть устойчива к разрывам связи в сети и обеспечивать линеаризуемость операций над данными.

Глава 3..4 Существующие решения

В предыдущих параграфах мы рассмотрели несколько задач, каждая из которых является предметом пристального изучения исследователей в области компьютерных наук, занимая все время на ее изучение и аккуратную реализацию. Кроме того, это не все задачи, связанные с создание собственной системы для построения полнотекстовых индексов. Среди тех, которые не были упомянуты ранее можно вспомнить алгоритмы сжатия и хранения данных для поиска или алгоритмы для построения и оптимизации поисковых запросов. Ввиду такой ситуации видится невозможным создавать собственное решение для полнотекстового поиска, так как это займет слишком большой объем времени и сил при итоговых результатах, несопоставимых с уже существующими.

Вследствие таких рассуждений было решено для задачи построения индексов и осуществления полнотекстового поиска использовать готовые инструменты. Крайне желательным качеством готового программного обеспечение является его открытость. Это позволяет лучше понимать устройство системы, в первую очередь для оперативного решения возникающих проблем. Таким образом, выбор пал на уже реализованные, зарекомендовавшие себя ранее движки для полнотекстового поиска с открытым исходным кодом. В результате анализа существующих аналогов, были выбраны и рассмотрены следующие альтернативы.

Apache Lucene[7] реализовывает функционал по работе с полнотекстовыми индексами, но не предоставляет интерфейса для внешних сервисов и встроенных возможностей по созданию распределенной системы.

Sphinx, кроме реализации полнотекстовых индексов и работы с ними, предоставляет также внешний интерфейс для приложений, которые будут использовать этот движок. Однако работа по распределению индексов на несколько компьютеров не автоматизирована, хоть инструменты для этого и есть.

Последние два решения, Apache Solr и Elasticsearch, очень похожи между собой. Оба предоставляют богатый внешний интерфейс для конечного приложения. Оба автоматически управляют распределенными индексами и предоставляют инструменты для горизонтального масштабирования.

Рассмотрим каждое решение подробно.

Глава 3..4.1 Apache Lucene

Lucene — фреймворк, созданный на платформе Java, для создания индексов для полнотекстового поиска и последующего поиска информации в них. Данный фреймворк решает задачи, связанные с построением индексов как структур данных в памяти и на постоянном носителе; предварительным анализом текста и приведением его к нормальной форме, упомянутой выше, необходимой для организации поиска; выполнением поисковых запросов, в

том числе переписывание их для лучшей производительности и релевантности и многие другие.

Однако сконцентрировавшись на задачах полнотекстового поиска, данный фреймворк не предусматривает создание распределенной системы хранения индексной информации исключительно с помощью средств самого фреймворка. Используя данное решение, задачи горизонтального масштабирования, предоставления интерфейса внешнему пользователю и индексирования данных в контексте данного проекта должны быть решены программистом самостоятельно.

Несмотря на то, что многие задачи оказываются покрыты функциональностью данного решения, не меньшее количество задач остается нерешенными. Среди них одни из самых сложных, связанные с надежностью и консистентностью поисковых индексов. По данным причинам было решено отказаться от Lucene в чистом виде.

Глава 3..4.2 Sphinx

Sphinx — готовая платформа для полнотекстового поиска, написанна на C++, используя в своей основе существующие базы данных и обладающая SQL-синтаксисом запросов, что очень удобно для встраивания в процессы, уже использующие подобный синтаксис для осуществления запросов к массивам данных. Данное решение обладает за счет своей архитектуры высокой производительностью как при индексации документов, так и при поиске. Для приложения, где производительность важнее всего, Sphinx среди всех остальных решений подойдет больше всего.

Однако как решение существующей задачи Sphinx подходил плохо, так как обладал скудным функционалом для создания распределенных индексов, требуя при этом большое количество ручной работы, от чего изначально хотелось избавится.

Глава 3..4.3 Elasticsearch

Elasticsearch — тоже готовая платформа, основанная на Lucene. Данное решение, полагаясь на качество построения индексов и поиска по ним, обеспечивающееся уже существующей библиотекой, концентрируется на обеспечении распределенной системы с готовым сетевым интерфейсом, решающее такие задачи, как репликация данных, поддержка синхронизации данных между репликами, обработка разделений сети и так далее.

Elasticsearch поддерживает дополнительные уровни абстракции, необходимые для оперирования большими объемами данных на множестве вычислительных узлов, и выполняет операции, необходимые для поддержки подобных абстракций и операций над ними. Например, ключевым понятием в данной системе является индекс, который состоит из некоторого количества шардов. Каждая реплика шарда назначается на некоторый узел в кластере Elasticsearch, после чего средствами Lucene на этом узле создается уже физический индекс. В этом индексе в дальнейшем содержатся документы, и в нем же происходит поиск.

По архитектуре Elasticsearch представляет собой систему с единым мастером, который выбирается с помощью протокола распределенного консенсуса Zen, реализованного специально для данного решения. После того, как мастер был выбран, только он может производить мутации состояния, такие как создание новых и удаление старых индексов, назначение шардов на узлы, добавление узлов в кластер и исключение узлов из кластера и так далее. Если мастер еще не выбран, то по-умолчанию допускаются запросы на чтение, но не на запись, что, впрочем, можно изменить с помощью конфигурации. В случае возникновения разделения в сети, мастер не выбирается, если только в текущей доле не находится достаточно оперирующих узлов, которые могут стать мастером.

Очень важно особенностью Elasticsearch является богатый специальный язык для написания запросов к индексам, поддерживающий вкладывающиеся запросы, логические связки и множество типов самих запросов, дубли-

рующих и дополняющих функциональность Lucene. Также, кроме основной способности находить набор наиболее релевантных документов среди всех, добавленных в индекс, Elasticsearch отличается большим набором функций для агрегации данных и нахождение различных метрик и статистики по данным, что открывает богатые возможности по анализу данных и подготовке отчетов по большим объемам информации.

Покрывая множество задач, связанных с полнотекстовым поиском в распределенной системе, Elasticsearch имеет некоторое количество недостатков, как то недостаточная надежность, сырость протокола распределенной синхронизации, отсутствие API для управления задачами и так далее. Однако положительные стороны в достаточной мере оправдывают меры предосторожности, необходимость в грамотной эксплуатации и тщательном мониторинге системы, чтобы потенциально выбрать данное решение.

Глава 3..4.4 Apache Solr

Во многом схоже с Elasticsearch, Solr является распределенной системой, которая в своей основе для полнотекстового поиска использует библиотеку Lucene. Главным отличием от конкурента в положительную сторону заключается в протоколе синхронизации, который используется в Solr. В выборе мастера Solr полагается на другой продукт от Apache Foundation, Zookeeper, который исключительно фокусируется на создании системы для распределенных блокировок, за счет чего имеет большую степень надежности в случаях неполадок и возникновения разделения сети.

В остальном архитектура Solr очень похожа внешне на архитектуру Elasticsearch, за исключением может быть не такого гибкого и развитого языка запросов и агрегаций. Однако, тогда как Elasticsearch поставляет внешний интерфейс поверх HTTP+Json, в Solr этим не ограничивается и транспорт гораздо более разнообразен, что позволяет встроиться в большинство существующих систем.

Глава 3..5 Выбор решения

В силу факторов, указанных выше, Sphinx и чистый Lucene не рассматривались, как возможные варианты при реализации финальной системы. Вследствие этого выбор сузился до одного из двух вариантов, очень похожих между собой, Solr или Elasticsearch. Для того, чтобы сделать осмысленный вывод, был произведен анализ различных аспектов этих двух решений, включающий такие вопросы, как производительность поиска и индексации, язык запросов, его удобство в использовании и совместимость с существующими системами, требование к содержанию и поддержке и так далее.

В результате анализа было принято решение использовать Elasticsearch, так как его возможности и удобство использования и эксплуатации перекрывали преимущества конкурента[10]. Кроме того, чаще всего в приложениях одновременно с поиском осуществляется активная индексация новых данных, а в подобных условиях известно, что Solr проигрывает по производительности поиска и индексации[11], когда как в обратной ситуации Solr был бы предпочтительным выбором по параметру быстродействия[12].

Глава 4. Индексация

Задачей, не связанной напрямую с полнотекстовым поиском и решением для его осуществления, является индексация, а именно транспорт данных из первичного источника в поисковые индексы. Так как индексы являются специальной структурой данных, они не обязаны давать доступ к первичным данным, перед тем, как те были проанализированы, приведены к набору термов и в таком виде сохранены. Кроме того, хранение данных в системах, которые не гарантируют линеаризуемость чревато потерями данных, что недопустимо для многих реальных приложений.

Однако если есть отдельно система надежного хранения данных и отдельно полнотекстовые индексы, необходимо, чтобы, во-первых, данные, попадая в первичный источник, попадали также в индексы. И кроме того, при сбое в системе хранения индексов, необходимо, чтобы данные синхронизировались и за конечное время оказывалось, что все документы, находящиеся в первичном хранилище, проиндексированы и доступны для поиска.

Глава 4..1 Индексатор

Индексатор — программа, внешняя по отношению к первичному хранилищу данных и поисковым индексам, которая выполняет функции описанные выше, то есть синхронизирует данные между указанными двумя. Существует два способа организации процесса индексации, в первом случае, работа индексатора организована в бесконечном цикле, на каждой итерации которого происходит работа, во втором случае создается система, основанная на событиях и их обработке. Однако во втором случае работа с ошибками индексации становится сложнее, так как необходимо генерировать отдельные события. Также необходимо организовать первичный перенос данных, уже находящихся в первичном источнике на момент начала работы индексатора.

Глава 4..1.1 Цикл индексации

В общем случае итерация цикла индексации организована следующим образом:

1. Прочитать набор документов из первичного источника
2. Преобразовать исходные документы
3. Отправить запрос на индексацию преобразованных документов
4. Приостановить выполнение на некоторое время

Шаг преобразования необходим для общности. Например, он может включать формирования запроса не на добавление документа в индекс, а на изменение существующего. Кроме того, не всегда есть потребность во всех данных, возможно в индекс будет отправлять какой-то срез. Также возможны ситуации, когда шаг преобразования не востребован и данные, прочитанные из первичного источника данных, сразу же отправляются в индексы, что является вполне корректным сценарием поведения.

Глава 4..1.2 Состояние

Индексатор не может существовать сам по себе, у него должна присутствовать внешняя часть, которая будет работать независимо от работы самого индексатора. Она будет содержать информацию о текущем прогрессе, то есть о том, какие документы уже прочитаны и проиндексированы, а какие документы еще подлежат обработке. Данная внешняя часть может быть реализована как файл в файловой системе, документ в базе данных, область памяти программы, запущенной независимо и любым другим способом, который обеспечивает независимость данного состояния от процесса индексации.

Наличие состояния продиктовано необходимость допустить возможность завершения программы индексатора, как в силу всевозможных ошибок и отказов, так и в силу намеренных действий. Если состояние будет целиком

частью индексатора, то при подобном перезапуске не будет доступна информация, какие документы уже были проиндексированы, а какие еще только следует прочитать из первичного источника. Таким образом у индексатора появляется внешний интерфейс, который диктует ему входные данные и сохраняет прогресс. Подобный инструмент является первым шагом к достижению надежности системы индексации.

Глава 4..1.3 Модульность и унификация

Так как задач по индексации существует большое количество, то возникает необходимость в ускорении разработки за счет унифицирования узлов и переиспользования уже использованной логики. Например, одни и те же данные по способу организации могут быть проиндексированы одним и тем же индексатором, по-разному сконфигурированным. Однако редко бывает так, что данные полностью совпадают по формату и логике обработке. Из архитектуры рабочего цикла индексатора следует наличие трех независимых компонент, на которые этот индексатор можно разделить — компонента для чтения, компонента для преобразования документов и компонента для составления запроса на индексацию для конкретных документов. Используя такую модульность и возможность независимо конфигурировать разные компоненты, мы получаем гораздо больший простор для переиспользования логики. Например, есть модуль для чтения документов из базы данных, конфигурируя его именем коллекции, из которой брать данные, и типом данных, можно менять приложения получающего индексатора, причем радикально.

Глава 4..1.4 Отказоусточивость и надежность

Держа в уме тот факт, что существует большое количество ситуаций, в которых программное обеспечение выходит из строя, нельзя не обращать внимание на корректную обработку подобных случаев. Самым простейшим способом реагировать на ситуации в случае отказов можно автоматически перезапуская процесс индексации, например средствами операционной системы

или специализированных программ. Однако это, во-первых, не спасает от отказов на уровне вычислительного узла, когда операционная система и все программы прикладного уровня перестают функционировать, во-вторых, не обеспечивает достаточной информативностью в случае ошибки, которая приводит к постоянному перезапуску индексатора.

Ввиду вышесказанного возникает необходимость создания специализированной распределенной системы запуска индексаторов, которая позволила бы выполнить требования по отказоустойчивости и надежности системы в целом. Как и в любой распределенной системе от нее требуется решения определенного класса задач, в том числе в данном случае, наличие линеаризуемости и устойчивости к разделению сети.

Глава 4..1.5 Существующие решения

Индексаторы, будучи очень сильно завязанными на конкретные типы данных и бизнес-процессы, чаще всего оставляются на откуп разработчиком конкретного продуктового решения, однако существуют некоторые универсальные программы с готовым набором конфигурируемых модулей, которые позволяют ускорить разработку и облегчить введение полнотекстового поиска в эксплуатацию.

Одним из самых известных решений является Logstash. Он в точности реализует схему, указанную выше, обладая тремя независимыми конфигурируемыми компонентами. Любую из них чаще всего можно реализовать с помощью настройки готовых модулей. Этот продукт активно продвигается создателями Elasticsearch, он действительно позволяет просто и быстро настроить даже не очень простой процесс непрерывной индексации данных их произвольного источника в полнотекстовый индекс. Главным его недостатком является то, что он не реализует логику, связанную с решением задачи распределения процесса индексации на несколько компьютеров. Logstash представляет собой программу, которая работает, будучи вручную запущена в операционной системе. Это четко следует принципу разделения ответ-

ственности между программами, но делает выбор в стороны данного решения невозможным.

Также существует множество решений, предложенных сообществом, для решения задачи индексации, но нет сопоставимого Logstash по расширяемости и гибкости, и тем более нет такого, который был, согласуясь с платформой разработки программного обеспечения, используемой в компании, заинтересованной в решении рассматриваемой задачи, предоставлял одновременно как богатые возможности по конфигурации и настройке, так и решение задачи распределения процесса индексации на множество вычислительных узлов.

Глава 4..1.6 Требования к решению

После безуспешной попытки найти решение среди существующих на данный момент, было очевидно, что необходимо создать свой продукт. Для того, чтобы определить, что необходимо проделать, было выдвинуто несколько требований, которым должно удовлетворять решение, чтобы можно быть считать его готовым к использованию в реальном окружении. Требования включали, но не ограничивались следующими пунктами:

- Модульность
- Наличие готовых компонентов
- Расширяемость
- Гибкость, высокая степень конфигурируемости
- Надежность, отказоустойчивость

В результате анализа возможного решения было выделено две независимые задачи, которые в сумме составили бы систему, удовлетворяющую вышеизложенным требованиям. Первая компонента, плагин, представляет собой некоторый скелет логики по индексации, хорошо расширяемый за счет

модулей, в том числе некоторых готовых. Плагины при этом имеют некоторый внешний интерфейс, унифицирующий работу с состоянием, запуск и остановку. Вторая компонента представляет собой распределенную систему для запуска плагинов, доставляя до них существующее состояние, сохраняющее прогресс и отслеживающая ошибки и отказы, происходящие с отдельно взятыми экземплярами.

Глава 4..2 Плагины

Глава 4..2.1 Компоненты

Плагин конфигурируется следующими компонентами, каждая из которых имеет свою зону ответственности и не зависит напрямую от других, и в частности может быть отдельно сконфигурирована:

- Reader
- Primary Key Extractor
- Document Factory
- Merger

Reader Данная компонента отвечает за чтение данных из первичного источника, как то например из базы данных. В рамках платформы, предоставляемой конечному пользователю системы, в частности, предлагается готовая вариация, читающая данные из локально разработанной системы хранения информации. Так как чтение напрямую связано с внутренним состоянием индексатора, то именно в этой части сконцентрирована работа с ним. Кроме интерфейса для чтения, Reader дает возможность установить состояние в данное или считать текущее, чтобы отслеживать и контролировать осуществляемый прогресс. Заметим также, что данный интерфейс обеспечивает чтение **входных данных**, назначение которых будет определено позднее.

Primary Key Extractor Для того, чтобы однозначно определить, к какому документу относится прочитанная часть входных данных, необходим некоторый уникальный идентификатор. Данный интерфейс определяет логику по извлечению подобного индексатора, предоставляя возможность по входным данным получить строку, уникальную для каждого документа в итоговом индексе, доступном для поиска.

Document Factory После того, как для входных данных был определен уникальный идентификатор, по нему происходит поиск в индексе, на случай, если такой документ уже существует и доступен для поиска. Если документа не обнаружилось, то необходимо инициализировать его из входных данных. Логика по подобной инициализации и заключена в данном интерфейсе.

Merger Если документ уже был найден в индексе, то входные данные заключают в себе изменения, которые необходимо применить к уже существующему документу. Логика по тому, как документу и входных данным сопоставить документ, отражающий внесенные правки и заключается в данной компоненте.

Глава 4..2.2 Принцип работы

Плагин заключает в себе два логических потока выполнения, первый занимается непрерывным чтением данных из источника и добавлением их в очередь, второй осуществляет функцию индексатора, преобразуя и индексируя прочитанные в первом потоке данные.

Логика выполнения потока чтения грубо может быть представлена в виде следующего алгоритма

1. Прочитать объект из источника
2. Добавить в формирующееся множество исходных данных

3. Если размер текущего множества меньше фиксированной величины, вернуться на шаг 1

4. Иначе, зафиксировать текущее состояние Reader'а и добавить в очередь на индексацию пару из текущего состояния и множества исходных данных

5. Обнулить множество исходных данных и перейти на шаг 1

Размер множества исходных данных ограничивается некоторой величиной, которая конфигурируется отдельно для каждого плагина в системе, ровно, как и максимальный размер очереди, при превышении которого поток чтения будет приостановлен до момента, когда освободится место. Данная системы была введена для ускорения процесса индексации, так как процесс чтения является достаточно времязатратным и при этом независимым.

Логика потока выполнения индексации в свою очередь может быть описана следующим образом

1. Взять множество исходных данных и состояние из очереди

2. Получить идентификатор каждого объекта исходных данных

3. Получить из индекса документы для каждого идентификатора

4. Пройтись по всем объектам исходных данных в порядке их чтения и либо создать новый документ, либо применить изменение к уже существующему

5. Проиндексировать получившиеся документы

6. Для всех объектов исходных данных, принадлежащих документам, которые не проиндексировались в силу какой-либо ошибки, повторить шаги начиная с 3

7. Считать, что были проиндексированы документы до состояния, взятого ранее из очереди

Кроме того, существует ограничение на количество попыток проиндексировать документы, которые вызывают ошибку, чтобы не возникло опасности возникновения бесконечного цикла без какого-либо прогресса. Функции взаимодействия с документами и объектами исходных данных реализованны с помощью вышеописанных компонент.

За счет того, что состояние сохраняется после того, как все документы, прочитанные до перехода в это состояние, были успешно проиндексированы, мы можем сказать, что невозможно получить извне состояние более новое, чем то, которое соответствует по меньшей мере не более новым данным, чем были на самом деле проиндексированы.

Глава 4..3 Система запуска плагинов

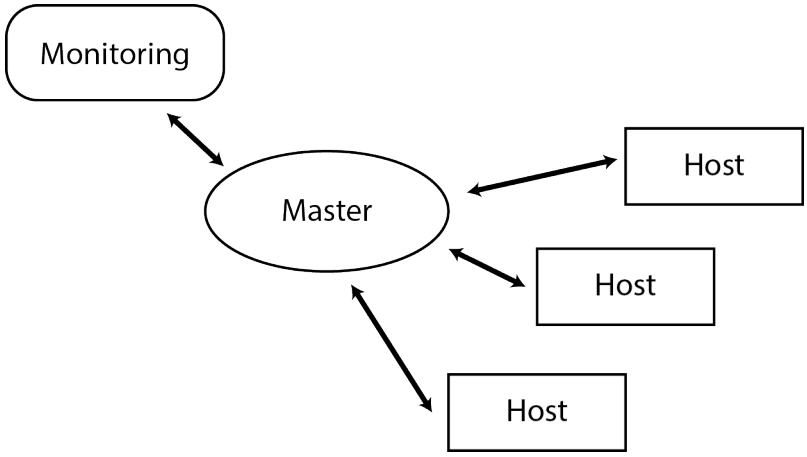
Распределенная система запуска плагинов также разбивается на несколько функциональных компонент, каждая из которых занимает свою роль в итоговой картине. Три роли охватывают поверхностно картину системы со стороны, а именно

- Master — часть, которая отвечает за хранение метаданных о существующих плагинах, распределение плагинов по существующим вычислительным мощностям, обработку и хранение ошибок
- Host — часть, напрямую взаимодействующая с плагинами. Хост получает информацию от мастера, какие плагины необходимо запустить, стартует их и следит за последующим выполнением
- Monitoring — интерфейс для администратора данной системы, который позволяет получить доступ к созданию новых плагинов, конфигурированию существующих и анализу произошедших событий в кластере

Кроме перечисленных компонентов, для корректного функционирования системы необходимо также несколько внешних систем, как то сервис распределенных блокировок и внешняя система хранения данных для исходного

кода плагинов и система хранения данных для состояния мастера. Общая схема системы в целом выглядит следующим образом:

Рис. 2, **Схема кластера системы запуска индексаторов**



Далее в отдельности рассмотрим каждую компоненту системы

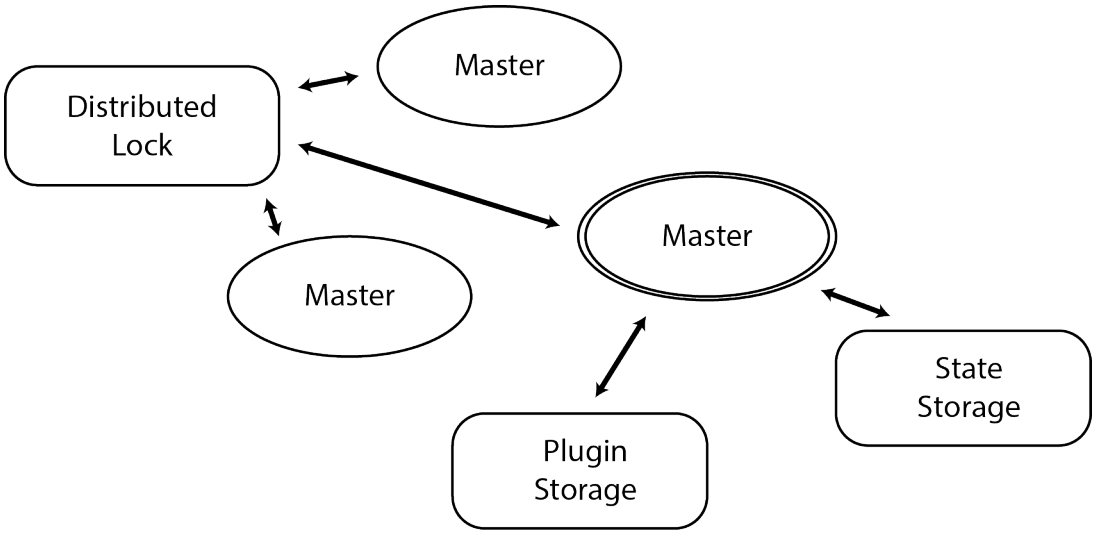
Глава 4..3.1 Мастер

Мастер представляет собой мета-состояние кластера и содержит в себе информацию о каждом существующем плагине, его состоянии, конфигурации, случившихся ошибках и поддерживает внешний интерфейс, доступный конечному пользователю, для изменения этого состояния. Для того, чтобы мастер не являлся единой точкой отказа в системе, одновременно существует несколько запущенных мастеров. С помощью внешнего механизма распределенных блокировок мастера выбирают одного, который единственный имеет после этого возможность изменять состояние.

Кроме того, перед тем, как изменение будет применено к состоянию, оно записывается в лог операций. Каждая операция надежно сохраняется в хранилище, для которого доказано свойство консистентности.

Иногда может произойти так, что мастер, который в данный момент держит единоличное право на оперирование с состоянием, выйдет из строя. Тогда с помощью все того же внешнего механизма распределенной блокировки выбирается новый мастер, который занимает место вышедшего из строя.

Рис. 3, **Схема устройства мастера (кластера мастеров)**



Таким образом, даже при множественном отказе мастера, покуда существует хотя бы один, успешно функционирующий и покуда функционирует система распределенных блокировок, рассматриваемая система будет функционировать.

Для оперирования плагинами в мастере есть механизм аренд, который выдает конкретному хосту уникальное в данный момент право на владение плагином, то есть на его запуск. Для поддержания аренд хосты периодически отсылают мастеру сообщения со своим текущим состоянием. На основании этих сообщений возможны изменения решений о выдаче аренд. Также из сообщений от хоста берется некоторая информация для сохранения в текущее состояние мастера. Если на протяжении продолжительного времени от хоста не приходит сообщение, то мастер считает этот хост вышедшим из строя и назначает плагины, назначенные на этот хост, на другие хосты.

Глава 4..3.2 Хост

Задача хоста — периодически информировать мастера о своем состоянии, получать от него список индексаторов, которые еще не были запущены и запускать их. Кроме того, хост также отправляет мастеру информацию о том, какие плагины запущенны в данный момент, их состояние и различные

метрики, а также ошибки, которые произошли в процессе работы. Если хост получает информацию о том, что у него не запущен плагин, который назначен на него в состоянии мастера, он сначала скачивает исполняемый код из выделенного хранилища, внешнего по отношению к рассматриваемой системе, затем запускает цикл исполнения скачанного кода, перезапуская плагин по мере надобности, если ошибка привела к его немедленной остановке.

Немаловажной задачей хоста является запись метрик во внешнее хранилище для мониторинга и построения аналитики. Для расследования аномальных ситуаций администратором необходимо понимать текущее состояние ресурсов, выделенных операционной системой на задачу хоста, а также иметь перед глазами историю изменения потребления данных ресурсов на протяжении некоторого времени, для осуществления правильного в текущий момент действия, направленного на стабилизацию ситуации и нормализации работы системы.

Глава 4..3.3 Мониторинг

Одним из важных компонентов системы является мониторинг, который содержит информацию о текущем состоянии кластера и позволяет через него осуществлять манипуляции с плагинами, их конфигурациями и состояниями. Мониторинг по сути является промежуточным пунктом между администратором, который контролирует работоспособность системы и мастером, управляющим плагинами в автоматическом режиме.

Заключение

В рамках данной работы было проанализировано множество задач, которые необходимо решить для успешного создания системы, удовлетворяющей выставленным требованиям. В результате исходная задача была декомпозирована до двух независимых компонентов, каждая из которых была независимо решена.

Был сделан вывод о необходимости использования готового решения для создания и управления распределенными поисковыми индексами, проведено сравнение существующих доступных вариантов и выбран наиболее подходящий с точки зрения функциональности и производительности.

Анализ существующих решений для индексации данных показал, что подходящего под все требования готового решения не существует, или не известно автору. По этой причине было решено создать собственный продукт для обеспечения полнотекстовых индексов непрерывным процессом индексации. Был выдвинут список требований и в соответствии с ним реализован программный продукт.

На данный момент система успешно внедрена и эксплуатируется более, чем 20 приложениями одновременно. В различных индексах расположено уже более 18 ТБ данных, к которым суммарно ежедневно обращаются более миллиона раз.

Дальнейшее развитие данной работы видится в повышении надежности сохранности данных путем введения механизмов отката процесса индексации и введением большего количества критериев для мониторинга системы. Также одним из направлений будущего развития может стать развитие пользовательского интерфейса и процессов введения в эксплуатацию новых индексаторов для достижения большей степени удобства и большей степени автоматизации в процессе администрирования системы.

Источники и литература

1. Kingsbury K., Strong consistency models [Электронный ресурс] // Aphyr URL: https://aphyr.com/posts/313-strong-consistency-models (дата обращения: 17.04.2016)

2. Brewer E., Towards Robust Towards Robust Distributed Systems [Электронный ресурс] // PODC Keynote URL: http://people.eecs.berkeley.edu/ brewer/cs262b-2004/PODC-keynote.pdf (дата обращения: 11.03.2016)

3. Gilbert S., Lynch N., Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services // ACM SIGACT News. – 2002. – Volume 33 Issue 2, June 2002. – сс. 51-59

4. Luhn H. P., A Statistical Approach to Mechanized Encoding and Searching of Literary Information // IBM Journal of Research and Development. – 1957. – Volume 1 Issue 4, October 1957. – сс. 309-317

5. Jones K. S., A statistical interpretation of term specificity and its application in retrieval // Document retrieval systems. – 1988. – сс. 132-142

6. Brin S., Page L., The anatomy of a large-scale hypertextual Web search engine // WWW7 Proceedings of the seventh international conference on World Wide Web 7. – 1998 – сс. 107-117

7. Białecki A., Muir R., Ingersoll G., Apache Lucene 4
SIGIR 2012 Workshop on Open Source Information Retrieval URL: http://opensearchlab.otago.ac.nz/paper_10.pdf (дата обращения: 22.05.2016)

8. McCandless M., Hatcher E., Gospodnetić O., Lucene in Action, 2-е изд. – Manning, 2010 – 488 с.

9. Gheorghe R., Matthew Hinman L., Russo R., Elasticsearch in Action, – Manning, 2015 – 496 с.

10. Apache Solr vs Elasticsearch [Электронный ресурс] // Kelvin Tan URL: http://solr-vs-elasticsearch.com/ (дата обращения: 23.05.2016)

11. Elasticsearch vs. Solr performance: round 2 [Электронный ресурс] // flax URL: http://www.flax.co.uk/blog/2015/12/02/elasticsearch-vs-solr-performance-round-2/ (дата обращения: 23.05.2016)

12. Realtime Search: Solr vs Elasticsearch [Электронный ресурс] // Socialcast URL: http://blog.socialcast.com/realtime-search-solr-vs-elasticsearch/ (дата обращения: 23.05.2016)