Московский государственный технический
университет имени Н. Э. Баумана

Факультет «Информатика и системы управления»
Кафедра «Программное обеспечение ЭВМ
и информационные технологии»

Расчётно-пояснительная записка
к курсовому проекту по базам данных на тему

Полнотекстовой поиск в сети Интернет

Исполнитель: студент ИУ7-61 Павелко П. Ю. _____

Руководитель: преподаватель ИУ7 Ломовской И. В. _____

Москва, 2016

Содержание

Введение

В данной работе рассматриваются системы полнотекстового поиска, которые позволяют искать слова в большом наборе документов и сортируют результаты поиска по релевантности найденных документов запросу. Алгоритмы полнотекстового поиска относятся к числу важнейших среди алгоритмов коллективного разума.

Информационный поиск — это очень широкая область с долгой историей. В этой работе затрагиваются лишь немногие ключевые идеи, но тем не менее дающие значительные результаты. Хотя в центре внимания будут алгоритмы поиска и ранжирования, а не требования к инфраструктуре, необходимой для индексирования больших участков Всемирной паутины, созданная в данной работе система должна быть способной индексировать до миллиона страниц в сутки, сохраняя при этом относительно небольшое время поиска.

В данной работе рассматриваются все основные этапы: обход страниц, выделение содержимого, индексирование документов и непосредственно поиск по собранному индексу согласно различным критериям для ранжирования.

Целью данной работы является разработка и реализация информационной системы для сбора информации в сети Интернет и последующего поиска по ней.

В рамках работы должны быть решены следующие задачи:

а) Анализ предметной области;

б) Разработка индекса — базы данных для хранения информации;

в) Разработка поискового робота — программы для сбора информации в сети Интернет;

г) Разработка поисковика — программы для поиска релевантных запросу страниц в индексе;

д) Разработка поисковой страницы — веб-приложения для осуществления запросов;

1 Аналитический раздел

Поисковая система состоит из двух основных компонентов: поискового робота, осуществляющего сбор данных в сети Интернет, и поисковика, осуществляющего поиск по индексу релевантных запросу документов и их ранжирование.

Поисковой робот (рис. 1.1) находит страницы в сети, используя ссылки с уже проиндексированных страниц. Начальный набор страниц задаётся пользователем, как и конфигурация робота (глубина обхода и другие).

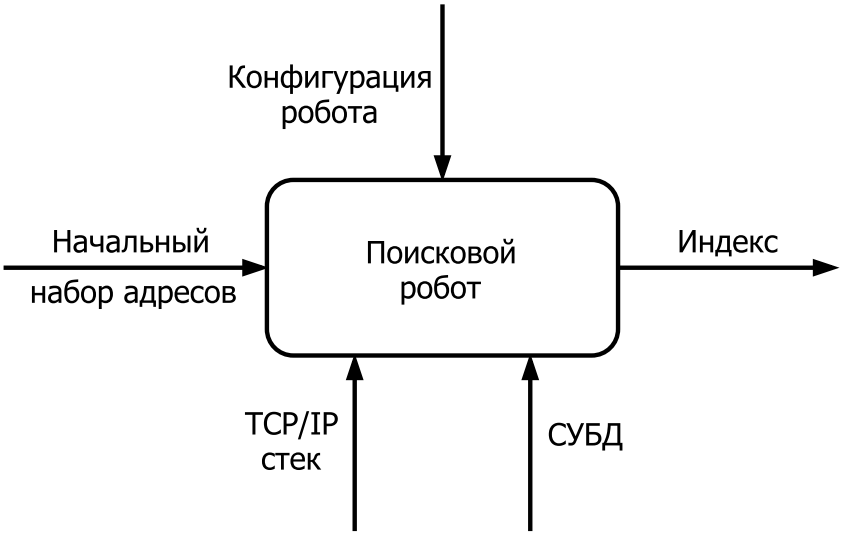

Рисунок 1.1 — IDEF0-A0 поискового робота.

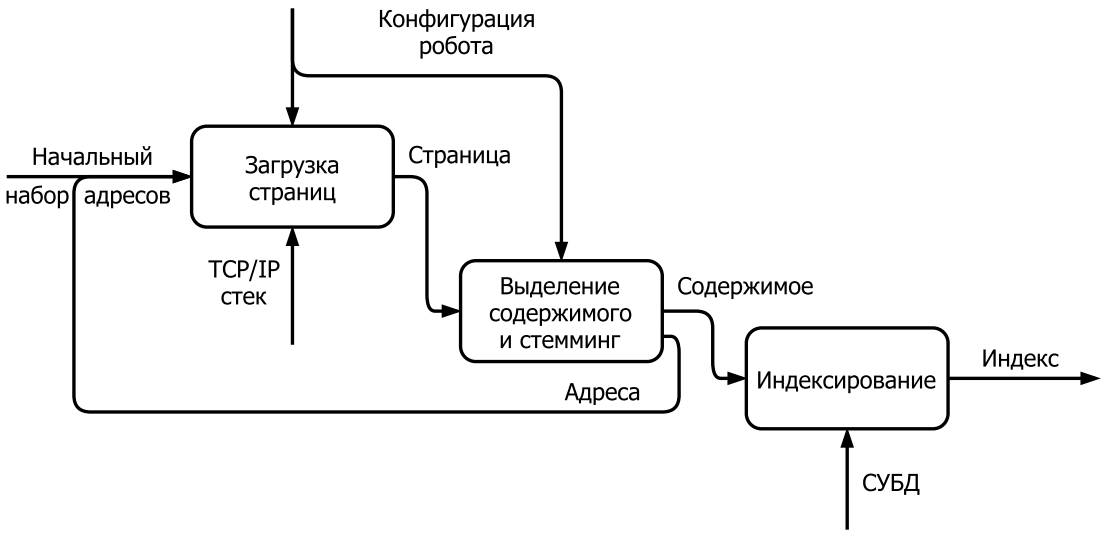

Рисунок 1.2 — IDEF0-A1 поискового робота.

Для достижения своей цели поисковому роботу необходимо получить страницу (стек TCP/IP), выделить основное содержимое, разобрать его на слова, выделив основу слова (стемминг), после чего сохранить в БД в форме, удобной для дальнейшего поиска (рис. 1.2).

Поисковик (рис. 1.3) по заданному запросу находит все релевантные документы в собранном индексе и возвращает пользователю отсортированный набор документов, соответствующих запросу.

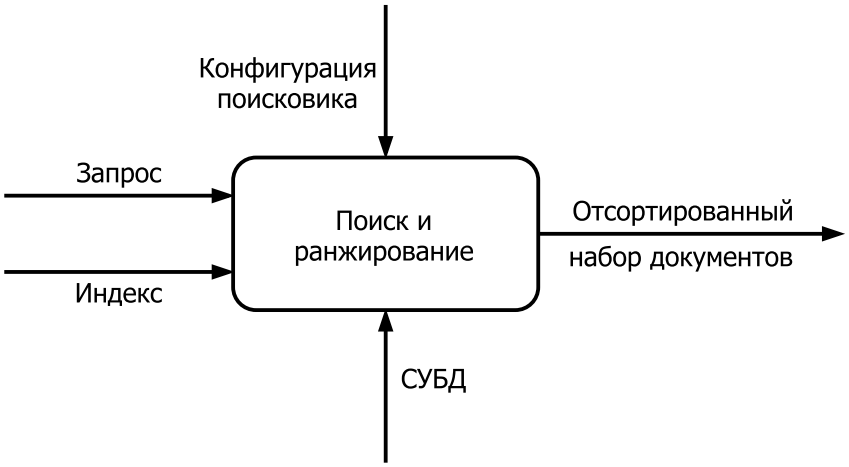

Рисунок 1.3 — IDEF0-A0 поисковика.

Поскольку именно поиск документов является главной целью, система должна разрабатываться с учётом тех характеристик документов, которые участвуют в ранжировании, так как от этого зависит способ сбора и хранения данных.

Перед проектированием базы данных необходимо рассмотреть предметную область (рис. 1.4). В целом предметная область определяется через две основные сущности: страницу и слово. Страницы могут ссылаться друг на друга посредством гиперссылок, в тексте которых содержатся определённые слова.

Краткое описание атрибутов:

а) URL — адрес страницы;

б) Ключ — агрессивно нормализованный адрес URL (см. 1.2.3.6);

в) PR — рейтинг страницы (см. 1.1.2.2);

г) Заголовок — совокупность содержимого всех заголовков на странице;

д) Длина в словах — количество слов в странице;

е) Основа — нормализованная форма слова (см. 1.2.3.5);

ж) Встречаемость — количество страниц, включающих данное слово;

з) IDF — производная от встречаемости характеристика (см. 1.1.1.1);

и) Количество — количество определённого слова в странице;

к) Положение — смещение первого вхождения слова (см. 1.1.1.3);

л) TF — частота вхождения слова (см. 1.1.1.1);

м) BM25 — производная характеристика TF и IDF (см. 1.1.1.2);

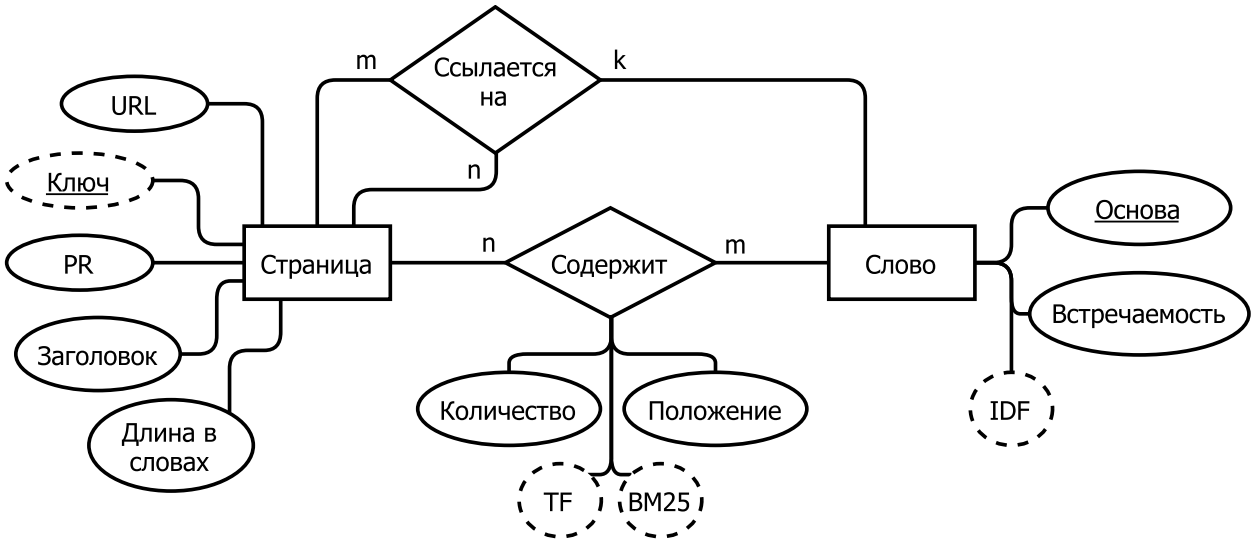

Рисунок 1.4 — Концептуальная схема предметной области.

Производные атрибуты (TF, IDF, BM25, ...) рассматриваются далее в этом разделе, как и причины их возникновения.

1.1 Ранжирование

Данный раздел посвящён разработке алгоритма сортировки документов и рассматривается первым, поскольку именно от способа поиска и сортировки документов зависит структура базы данных индекса.

Для начала определимся с некоторыми понятиями:

Запрос — набор слов, заданных пользователем.

Релевантный документ — документ, соответствующий запросу.

Ранжирование — упорядочивание документов по релевантности.

Фактор (признак) — характеристика свойств документа, запроса или их взаимоотношение.

Виды факторов:

а) Статические факторы — зависящие только от документа;
б) Факторы, зависящие только от запроса;
в) Динамические факторы — зависящие и от документа, и от запроса.

Функция ранжирования — алгоритм, выполняющий ранжирование, используя один или несколько факторов. Функция ранжирования сама по себе так же является фактором и может быть комбинацией других функций ранжирования, используя, например, средневзвешенную сумму или обученную модель регрессии [1].

Так же факторы условно можно разделить на следующие классы:

а) По содержимому: наличие слов из запроса, их частоты, положения;
б) Ссылочные: популярность страницы, цитируемость;
в) Метрики, основанные на ответной реакции пользователей.

В данной работе внимание уделено первым двум классам факторов. Класс ответной реакции, функции ранжирования которого часто машинно-обучены, не рассматриваются, так как для получения пригодных результатов требуется достаточно длительное живое обучение, которые в рамках работы не достижимо.

Далее будут рассмотрены оставшиеся два класса: по содержимому и ссылочные. После чего будет выведена итоговая функция ранжирования, которая будет объединять рассмотренные.

1.1.1 Ранжирование по содержимому

Данный класс основывается на информации, которую можно получить непосредственно из документа и запроса:

— Общее количество слов в документе (длина документа);
— Частота вхождения запрошенных слов в документ;
— Положение запрошенных слов в документе;
— Расстояние между запрошенными словами в документе;
— Входят ли запрошенные слова в заголовки.

1.1.1.1 TF-IDF

Простейшим динамическим фактором является количество вхождений слова q запроса Q ($q \in Q$) в документ d: n_q^d.

Однако использование данного фактора приводит к тому, что более длинные страницы оказываются более релевантными, даже если относительная доля искомого слова невелика. Поэтому имеет смысл говорить о частоте слова.

Частота слова (от англ. TF — term frequency) — другой простейший динамический фактор, определяемый как

$$tf(q,d) = \frac{n_q^d}{|d|}, \tag{1.1}$$

где $|d|$ — длина документа (количество слов в нём).

Тогда функцию ранжирования на основе только этого фактора можно задать как

$$score^{tf}(d,Q) = \sum_{q \in Q} tf(q,d). \tag{1.2}$$

Данный фактор позволяет ранжировать страницы исходя из того, сколько раз на ней встретилось искомое слово. Так, выполняя поиск по слову «python», пользователь ожидает увидеть документы, где это слово встречается часто, а не документ о музыканте, который где-то в конце упомянул, что у него дома живёт питон.

Однако данный фактор обладает существенным недостатком: в случае, если в запрос входят популярные (часто встречаемые) слова, то большая часть веса будет приходится именно на них. Для решения этой проблемы вводят обратную частоту документа.

Обратная частота документа (от англ. IDF — inverse document frequency) — инверсия частоты, с которой слово встречается в документах. Учёт IDF уменьшает вес широкоупотребительных слов.

$$idf(q) = \log \frac{N}{N_q}, \tag{1.3}$$

где N — количество документов в коллекции, $N = |D|$;

N_q — количество документов, содержащих слово q, $N_q = |\{d : d \ni q\}|$.

Объединяя (1.1) и (1.3), получаем следующую функцию ранжирования на основе TF-IDF:

$$score^{tfidf}(d, Q) = \sum_{q \in Q} tf(q, d) \cdot idf(q, D). \tag{1.4}$$

Теперь вес некоторого слова пропорционален количеству употребления этого слова в документе, и обратно пропорционален частоте употребления слова в других документах коллекции.

1.1.1.2 BM25

BM25 — TF-IDF-подобная функция ранжирования, имеющая лучшую вероятностную интерпретацию [2]:

$$score^{bm25}(d, Q) = \sum_{q \in Q} idf(q) \cdot \frac{n_q^d \cdot (k+1)}{n_q^d + k \cdot \left(1 - b + b \cdot \frac{|d|}{avg(|d|)}\right)} \tag{1.5}$$

где k и b — свободные коэффициенты, $b \in [0..1]$ и $k \geqslant 0$;
$avg(|d|)$ — средняя длина документов в коллекции.

В данном методе часто используют «сглаженные» варианты idf, например:

$$idf(q) = \log \frac{N - N_q + 0.5}{N_q + 0.5}. \tag{1.6}$$

Вышеуказанная формула IDF имеет следующую особенность. Для слов, входящих в более чем половину документов из коллекции, значение IDF отрицательно. Таким образом, при наличии любых двух почти идентичных документов, в одном из которых есть слово, а в другом — нет, второй может получить большую оценку. Простейшим решением является игнорирование отрицательных слагаемых в сумме, что эквивалентно игнорированию соответствующих высокочастотных слов:

$$idf(q) = \max\left(\log \frac{N - N_q + 0.5}{N_q + 0.5}, 0\right). \tag{1.7}$$

Существуют модификации метода, позволяющие уменьшить влияние длины документа для очень больших документов: BM25l [3] и BM25+

[4], однако в данной работе эта проблема решается восстановлением выбросов (раздел 1.1.3.2).

Для объединения значений BM25 по разным полям документа (текст, заголовки, входящие ссылки и др.) часто используют различные вариации метода BM25F [2]. Однако в данной работе BM25 является лишь одной из многих частей итоговой функции ранжирования и проблема объединения решена нормализацией каждой части в отдельности (раздел 1.1.3.1).

1.1.1.3 Положение в документе

Обычно, если страница релевантна поисковому слову, то это слово расположено близко к началу страницы, быть может, даже находится в заголовке [1]. Чтобы воспользоваться этим наблюдением, поисковая система может приписывать результату больший ранг, если поисковое слово встречается в начале документа:

$$score^{pos}(d, Q) = \sum_{q \in Q} p_q^d, \tag{1.8}$$

где p_q^d — порядковый номер первого вхождения слова q в документ d.

1.1.1.4 Расстояние между словами

Если запрос содержит несколько слов, то часто бывает полезно ранжировать результаты в зависимости от того, насколько близко друг к другу встречаются поисковые слова. Как правило, вводя запрос из нескольких слов, человек хочет найти документы, в которых эти слова концептуально связаны [1].

$$score^{dist}(d, Q) = \sum_{\substack{i<j \\ q_i, q_j \in Q}} |p_{q_j}^d - p_{q_i}^d|. \tag{1.9}$$

Эту же функцию ранжирования можно использовать при поиске точного совпадения (фраза, заключённая в кавычки), указав для неё большой вес.

Однако данная функция ранжирования обладает существенным недостатком: это единственная функция из рассмотренных, которая тре-

бует хранения в индексе всех вхождений слов в документ, что многократно увеличивает индекс и, как следствие, время запроса. Поэтому в данной работе эта функция не используется.

1.1.2 Ссылочное ранжирование

Все обсуждавшиеся до сих пор функции ранжирования были основаны на содержимом документа. Хотя многие поисковые системы до сих пор работают таким образом, часто результаты можно улучшить, приняв во внимание, что сказано об этом документе в других. Это особенно полезно при индексировании документов сомнительного содержания или таких, которые могли быть созданы спамерами, поскольку маловероятно, что на такие документы ссылаются релевантные.

1.1.2.1 Простой подсчёт ссылок

Простейший способ работы с внешними ссылками заключается в простом подсчёте их количества. Так обычно оцениваются научные работы: считается, что их значимость тем выше, чем чаще их цитируют.

$$score^{inb}(d) = \left|\{(d_i, d) \in L\}\right|, \tag{1.10}$$

где L — мн-во всех ссылок (d_i, d_j).

1.1.2.2 PageRank

Этот алгоритм приписывает каждому документу ранг, оценивающий его значимость. Значимость документа вычисляется исходя из значимости ссылающихся на него документов и общего количества ссылок, имеющихся на каждом из них [1]:

$$pr(d) = 0.15 + 0.85 \cdot \left(\sum_{(d_i,d)\in L} \frac{pr(d_i)}{|L_{d_i}|} \right), \tag{1.11}$$

где L_d — мн-во всех ссылок с документа d: $L_d = \{(d, d_j) \in L\}$.

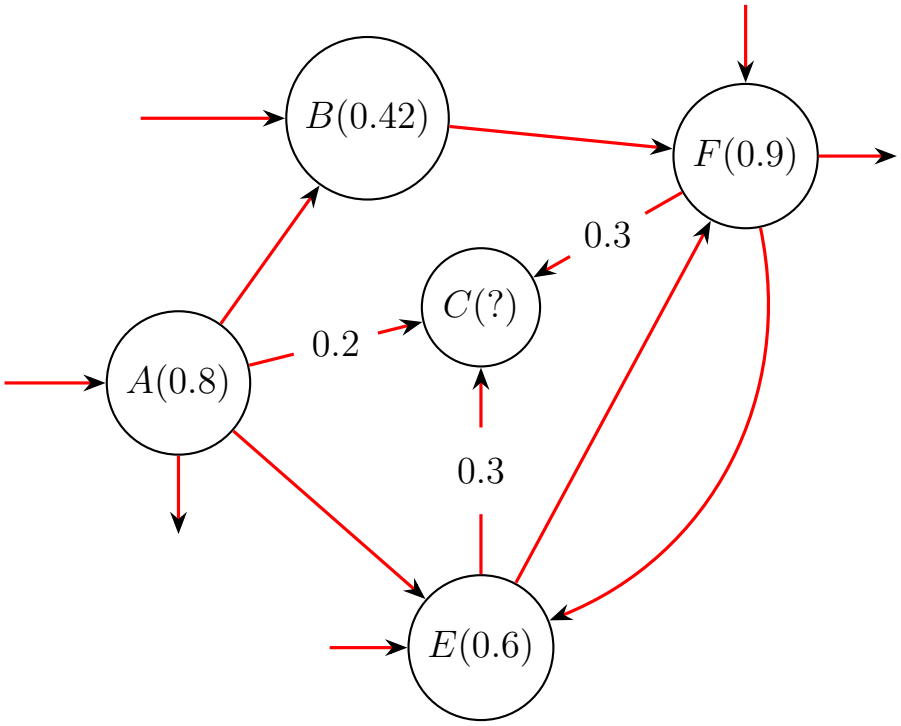

Рисунок 1.5 — Вычисление ранга PageRank документа C.

Например, PageRank страницы C (рис. 1.5) вычисляется как

$$\begin{aligned} pr(C) &= 0.15 + 0.85 \cdot \left(\frac{pr(A)}{|L_A|} + \frac{pr(F)}{|L_F|} + \frac{pr(E)}{|L_E|} \right) \\ &= 0.15 + 0.85 \cdot \left(\frac{0.8}{4} + \frac{0.9}{3} + \frac{0.6}{2} \right) \\ &= 0.15 + 0.85 \cdot 0.8 = 0.83. \end{aligned} \tag{1.12}$$

Увы, имеется небольшая ловушка — в данном примере для всех документов, ссылающихся на C, уже вычислен ранг, что является тривиальным случаем. Необходимо вычислить ранги для множества документов, ранги которых ещё не известны.

Решение состоит в том, чтобы присвоить всем документам произвольный начальный ранг (отличный от нуля, например 1) и провести несколько итераций. Количество необходимых итераций зависит от числа страниц и в нашем случае (количество страниц до миллиона) 20-30 должно быть достаточно. Google, например, просчитывает ранги для своего индекса приблизительно за 100 итераций.

Функция ранжирования в данном случае предельно проста:

$$score^{pr}(d) = pr(d). \tag{1.13}$$

1.1.2.3 Использование текста ссылки

Ещё один полезный способ ранжирования результатов — использование текста ссылок на документ при определении степени её релевантности запросу. Часто удаётся получить более качественную информацию из того, что сказано в ссылках, ведущих на документ, чем из самого документа.

Таким образом, учитывается ранг источников тех ссылок, которые ведут на оцениваемую страницу и содержат слова из запроса:

$$score^{ref}(d, Q) = \sum_{\substack{(d_i,d,t)\in L \\ q\in Q\cap t}} pr(d_i), \tag{1.14}$$

где t в (d_i, d, t) — текст ссылки (d_i, d).

1.1.3 Объединение функций ранжирования

Теперь, имея все необходимые функции ранжирования, необходимо получить итоговую функцию, которая будет являться средневзвешенной суммой нормализованных функций ранжирования.

1.1.3.1 Нормализация

Чтобы сравнивать результаты, получаемые различными функциями ранжирования, необходимо как-то нормализовать их, то есть привести к одному и тому же диапазону и направлению: от 0 (наихудший результат) до 1 (наилучший результат):

$$norm(s, s_{min}, s_{max}) = \begin{cases} \frac{s-s_{min}}{s_{max}-s_{min}}, & s_{max} > s_{min} \\ 1, \text{ иначе} \end{cases}, \tag{1.15}$$

где s — результат функции ранжирования;
s_{min} — минимальное значение функции ранжирования;
s_{max} — максимальное значение.

В случае, когда $s_{min} = s_{max}$, то есть значение функции для всех документов одинаково, будем считать, что все документы получили максимальную оценку.

1.1.3.2 Обработка выбросов

Выброс — результат измерения, выделяющийся из общей выборки. Например, среди документов, полученных по запросу «ecmascript», встречается спецификация, а так как ссылок, включающих искомое слово, на неё ведёт много, то и значение $score^{ref}$ для данного документа будет сильно выше других документов. Это приведёт к дискредитации $score^{ref}$ — остальные документы будут иметь низкие показатели.

Таким образом, необходимо задать такой диапазон $[s'_{min}, s'_{max}]$, который не будет покидать значение функции ранжирования. Тогда исправленный результат вычисляется как

$$s' = \min(s'_{max}, \max(s, s'_{min})), \tag{1.16}$$

где s — результат функции ранжирования.

Простейший метод определения такого диапазона основан на межквартильном расстоянии: выбросами считается всё, что не попадает в диапазон

$$\left[(s_{25} - 1.5 \cdot (s_{75} - s_{25})), (s_{75} + 1.5 \cdot (s_{75} - s_{25}))\right], \tag{1.17}$$

где s_{25} — 0.25-квантиль;
s_{75} — 0.75-квантиль.

Однако использование такого критерия приводит к тому, что по некоторым функциям ранжирования s_{min} или s_{max} не достижимы (максимум или минимум внутри диапазона, то есть $s'_{min} < s_{min}$ или $s_{max} < s'_{max}$). Поэтому имеет смысл в качестве границ диапазона брать

$$\begin{aligned} s'_{min} &= \max(s_{min}, s_{25} - 1.5 \cdot (s_{75} - s_{25})), \\ s'_{max} &= \min(s_{max}, s_{75} + 1.5 \cdot (s_{75} - s_{25})). \end{aligned} \tag{1.18}$$

1.1.3.3 Итоговая функция ранжирования

Теперь, если функцию нормировки (1.15) переписать с учётом исправления выбросов:

$$norm(s) = \begin{cases} 1, & s \geqslant s'_{max} \\ 0, & s < s'_{min} \\ \frac{s - s'_{min}}{s'_{max} - s'_{min}}, & \text{иначе} \end{cases}, \tag{1.19}$$

то итоговую формулу для функции ранжирования можно задать как средневзвешенную сумму всех нормализованных функций:

$$score(d, Q) = \frac{\sum w_{sc} \cdot norm(score^{sc}(d, Q))}{\sum w_{sc}}, \tag{1.20}$$

где w_{sc} — вес функции ранжирования sc.

В данной работе используются следующие (рассмотренные выше) функции ранжирования: $score^{bm25}$ (отдельно для всего текста и для заголовков), $score^{pos}$, $score^{pr}$, $score^{ref}$.

1.2 Сбор информации

Сбор информации является важной задачей в системах полнотекстового поиска, потому что именно качество собранной информации главным образом влияет на репрезентативность выборки, которую получит пользователь системы, совершающий запрос.

Данная задача порождает множество проблем, таких как: огромное количество заведомо нерелевантных страниц, ограничительная пропускная способность каналов, слабость серверов и пр.

В данном разделе приводится описание возникающих проблем и способы борьбы с ними.

1.2.1 Фильтрация посещённых страниц

Во время обхода важно отбрасывать ссылки, ведущие на страницы, которые уже были посещены или находятся в очереди на обработку. Очевидное на первый взгляд решение проблемы — использование ассоци-

ативных массивов (причём скорее всего именно хеш-таблиц) — довольно требовательно к памяти.

Поэтому для фильтрации посещённых страниц применяется фильтр Блума.

1.2.1.1 Фильтр Блума

Фильтр Блума — вероятностная структура данных, позволяющая компактно хранить множество элементов и проверять принадлежность заданного элемента к множеству. При этом существует вероятность получить ложноположительный результат, то есть ситуацию, когда элемента в множестве нет, но фильтр сообщает, что элемент есть.

Фильтр Блума может использовать любой объём памяти, заранее заданный пользователем, причём чем он больше, тем меньше вероятность ложного срабатывания. Поддерживается операция добавления новых элементов в множество, но не удаления существующих (если только не используется модификация со счётчиками).

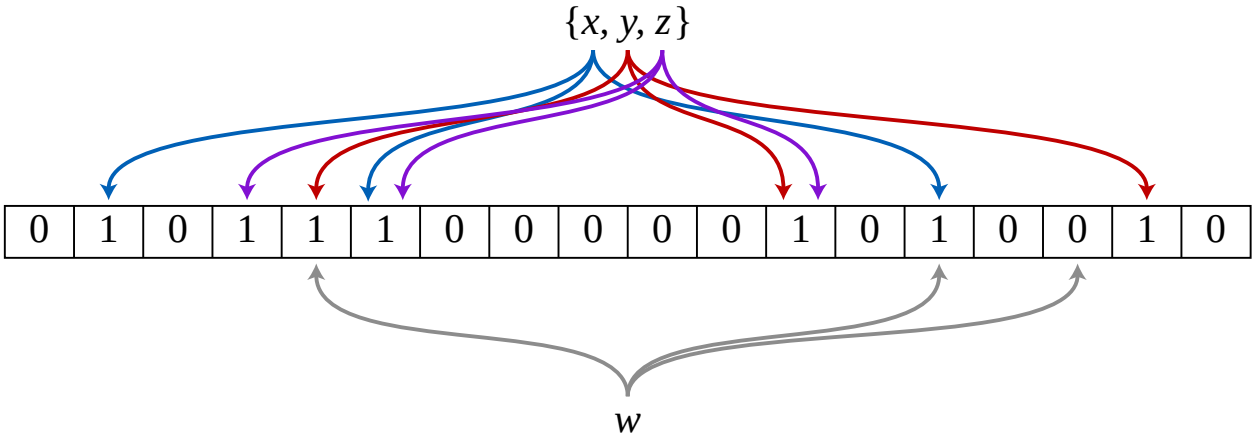

Рисунок 1.6 — Пример фильтра Блума с $m = 18$ и $k = 3$.

Фильтр Блума представляет собой битовый массив из m бит. Изначально, когда структура данных хранит пустое множество, все m бит обнулены. Пользователь должен определить k независимых хеш-функций h_i, отображающих каждый элемент в одну из m позиций битового массива достаточно равномерным образом.

Для добавления элемента e необходимо записать единицы на каждую из позиций $h_i(e)$ битового массива.

Для проверки принадлежности элемента e к множеству хранимых элементов, необходимо проверить состояние битов h_i. Если хотя бы один из них равен нулю, элемент не может принадлежать множеству (иначе бы при его добавлении все эти биты были установлены). Если все они равны единице, то структура данных сообщает, что e принадлежит множеству. При этом может возникнуть две ситуации: либо элемент действительно принадлежит к множеству, либо все эти биты оказались установлены по случайности при добавлении других элементов, что и является источником ложных срабатываний в этой структуре данных.

Согласно [5], для заданных n — числа ожидаемых элементов в множестве и p — максимальной вероятности ложноположительного срабатывания возможно вычислить оптимальный размер m как

$$m = -\frac{n \ln p}{(\ln 2)^2}, \tag{1.21}$$

и оптимальное количество хеш-функций как

$$k = \frac{m}{n} \ln 2. \tag{1.22}$$

1.2.1.2 Параметры фильтра Блума

Количество максимально ожидаемых страниц очевидным образом зависит от глубины обхода:

$$n = l^{d-1}, \tag{1.23}$$

где d — максимальная глубина обхода;

l — среднее количество ссылок на странице.

Так, при $l = 100$, $d = 4$ и $p = 0.00001$ (ожидается не более одного ложноположительного срабатывания на 10 миллионов) потребуется всего ≈ 2.86МБ данных.

1.2.2 Стандарт исключений для роботов

Когда владельцы сайта хотят ограничить доступ к определённым страницам для поисковых ботов, они используют для этого файл `robots.txt`, находящийся в корне сайта (то есть по пути `/robots.txt`).

Это может быть полезно для ограничения частоты запросов со стороны роботов, запрета индексирования динамический и служебных страниц.

Формат файла имеет следующий вид:

```
<поле>:<необязательный пробел><значение><необязательный пробел>
```

Стоит заметить, что вместо пробела могут использоваться также другие пробельные символы, а `поле` является регистронезависимым.

Основные используемые поля:

— `user-agent`: стандартное поле, используется для задания имени бота, которому адресованы последующие правила, либо `*`, если всем;

— `disallow`: стандартное поле, используется для задания пути, индексация которого запрещена. В пути разрешены два спец-символа: `*` (любое количество любых символов) и `$` (конец пути). Причём путь, к примеру, `/path/to/file` действует аналогично `/path/to/file*`;

— `crawl-delay`: нестандартное поле, используется для задания минимальной задержки в секундах между запросами со стороны робота.

1.2.3 Разбор страницы

После загрузки страницы необходимо подготовить её к индексации — выделить полезную информацию: определить заголовок, слова для индексации, ссылки для загрузки.

1.2.3.1 Декодирование мнемоник HTML

Мнемоника — это конструкция SGML, которая ссылается на символ из набора символов текстового файла. В HTML предопределено большое количество спецсимволов. Чтобы вставить определённый символ в разметку, нужно вставить определенную ссылку-мнемонику в HTML-структуру.

Мнемоника имеет вид `&...;`. Так, например, буква «q» может представляться как `&#113;`, а «—» как `&mdash;`.

Необходимость в декодировании символов-мнемоник исходит из нескольких причин:

а) Мнемоники встречаются в заголовках страницы, который будет отображаться пользователю при поиске;

б) Должны индексироваться не мнемоники, а только их значения. Так, мнемоника `&empty;` («∅») не должна индексироваться как слово «empty».

1.2.3.2 Выделение содержимого

Далеко не вся информация на странице ценна для поиска. Например, информация, размещённая в подвале и по сторонам страницы несёт обычно служебную информацию, которая может быть полезна при обходе страниц (так как может содержать полезные ссылки), но бесполезна конечному пользователю. Поэтому ставится вопрос об ограничении индексации исключительно полезными частями страницы. Для этого необходимы методы выделения такой информации, которые в основном базируются на эвристики и неплохо работают во многих случаях [6].

Для получения основного содержимого во многих случаях достаточно сохранять только текстовые элементы HTML, то есть блоки текста, которые не прерывались разметкой, которые имеют более чем десяток слов. Люди выбирают один из двух типов текста для двух различных мотивов написания текста: «навигационного» и «информационного».

Для элементов навигации обычно применяется текст, состоящий из нескольких слов (например, «STOP», «Прочтите это», «Нажмите здесь»), в то время как для основого содержимого используется много слов.

В то время как это разделение работает во множестве случаев, все становится сложнее с заголовками, короткими предложениями, отказами от ответственности, авторскими правами и другими колонтитулами.

Есть более сложные стратегии, а также функции, которые помогают отделять основное содержание от шаблонного [7]:

а) Рассмотренная выше плотность HTML-тегов;

б) Ссылочная плотность (количество слов внутри ссылок по сравнению с общим количеством слов в блоке);

в) Связь текущего блока с контекстом (с предыдущими и следующими блоками);

г) DOM-структура документа (`<article>`, `<section>` и другие);

д) Визуальное изображение страницы (например, большие изображения, окружённые текстом);

е) Различные машинно-обученные алгоритмы.

Результирующий алгоритм, используя комбинацию этих факторов, рекурсивно оценивает все узлы документа, находя наиболее похожие на основное содержимое.

1.2.3.3 Выделение ссылок

Ссылки на странице могут быть обнаружены во многих местах: `<a>`, `<link>`, `<script>` и др. Но для поискового бота по (x)html интерес представляют только ссылки в `<a>`, причём далеко не все. Для уменьшения количества бесполезных запросов, связанных, например, с неиспользуемым языком, которые скорее всего всё равно будут выявлены на этапе получения заголовков (HTTP-заголовок `content-language`), можно попробовать угадать по адресу ссылки релевантность.

Рассмотрим несколько методов фильтрации ссылок:

а) По наличию букв из неиспользуемых алфавитов в пути ссылки. Действительно, если ссылка содержит несколько таких символов, то скорее всего представленная информация на странице имеет тот же язык;

б) Использование чёрного или белого списков расширений, которые позволяет откидывать страницы с заведомо нерелевантной информацией, например `.jpg`. Чёрный список задаётся списком запрещённых разрешений, в том время как белый список — разрешённых, например `.html`, `.asp` и `.php`;

в) Фильтрация доменов, так же основанная на чёрном или белом списках;

г) Фильтрация поддоменов, основанная на некоторой эвристике. Например, поддомены `git.` и `svn.` часто содержат соответствующие репозитории, индексировать которые в большинстве случаев не имеет смысла.

Кроме того, ссылки, заданные через `<a>` могут иметь атрибут `rel="nofollow"`, который используется для того, чтобы запретить учи-

тывать данную ссылку поисковыми ботами при индексации страницы и передавать по ней вес данной страницы при расчёте PageRank. Чаще всего таким образом помечаются рекламные ссылки. Несмотря на то, что в индекс данные ссылки не попадают, включать их в оборот всё же имеет смысл.

1.2.3.4 Стоп-слова

Существование заведомо высокочастотных слов (таких как «an», «and», «и», «как» и др.) ведёт к раздуванию индекса и, как следствие, замедлению работы поиска. При этом их вклад в ранжирование минимален. Поэтому разумно отказаться от индексирования таких слов вообще, путём занесения их в списки так называемых стоп-слов.

1.2.3.5 Стемминг

Стемминг —процесс нормализации слов путём выделения основы. Это позволяет учитывать морфологически близкие слова как формы одно и того же слова (например, «connect» и «connected» или «чистый» и «чистая»), что сильно улучшает ранжирование и упрощает поиск.

Наиболее известен алгоритм стемминга Портера. Алгоритм не использует баз основ слов, а работает, последовательно применяя ряд правил отсечения окончаний и суффиксов [8].

Рассмотрим версию алгоритма для русского языка [9].

Во-первых, в слове выделяются три зоны:

RV — область слова после первой гласной. Может быть пустой, если гласных в слове нет;

R1 — область слова после первого сочетания «гласная-согласная»;

R2 — область R1 после первого сочетания «гласная-согласная».

Далее выделяются группы окончаний слов:

— Совершенного герундия («в», «вши», «вшись» после «а» или «я»; «ив», «ивши», «ившись», «ыв», «ывши», «ывшись»);

— Прилагательных («ее», «ие», «ые», «ое», «ими», «ыми», «ей», «ий», «ый», «ой», «ем», «им», «ым», «ом», «его», «ого», «ему», «ому», «их», «ых», «ую», «юю», «ая», «яя», «ою», «ею»);

— Причастных («ем», «нн», «вш», «ющ», «щ» после а и я; «ивш», «ывш», «ующ»);

— Возвратных («ся», «сь»);

— Глагольных («ла», «на», «ете», «йте», «ли», «й», «л», «ем», «н», «ло», «но», «ет», «ют», «ны», «ть», «ешь», «нно» после а или я; «ила», «ыла», «ена», «ейте», «уйте», «ите», «или», «ыли», «ей», «уй», «ил», «ыл», «им», «ым», «ен», «ило», «ыло», «ено», «ят», «ует», «уют», «ит», «ыт», «ены», «ить», «ыть», «ишь», «ую», «ю»);

— Существительных («а», «ев», «ов», «ие», «ье», «е», «иями», «ями», «ами», «еи», «ии», «и», «ией», «ей», «ой», «ий», «й», «иям», «ям», «ием», «ем», «ам», «ом», «о», «у», «ах», «иях», «ях», «ы», «ь», «ию», «ью», «ю», «ия», «ья», «я»);

— Превосходных («ейш», «ейше»);

— Словообразовательных («ост», «ость»);

— Адъективированных (определяется как прилагательное или причастие + прилагательное окончание).

При поиске окончания из всех возможных выбирается наиболее длинное. Например, в слове «величие» выбираем окончание «ие», а не «е».

Все проверки производятся над областью RV. Так, при проверке на совершенный герундий предшествующие буквы «а» и «я» также должны быть внутри RV. Буквы перед RV не участвуют в проверках вообще.

а) Найти окончание совершенного герундия. Если оно существует — удалить его и завершить этот шаг. Иначе, удаляем возвратное окончание, если существует. Затем по порядку удаляется адъективированное окончание, глагольное окончание, окончание существительного. Как только одно из них найдено — шаг завершается;

б) Если слово оканчивается на «и» — удалить «и»;

в) Если в R2 найдётся словообразоваельное окончание — удалить его.

г) Возможен один из трёх вариантов:

1) Если слово оканчивается на «нн» — удалить последнюю букву;

2) Если слово оканчивается на превосходное окончание — удаляем его и снова проверяем «нн»;

3) Если слово оканчивается на «ь» — удалить его.

Аналогично существуют версии алгоритма для многих европейских языков, однако в данной работе используется только русский и английский языки.

1.2.3.6 URL нормализация

Различные URL могут указывать на одну и ту же страницу, но отличаться при этом незначительно.

URL нормализация — процесс приведения URL к каноничному виду, путём применения различных правил трансляции адресов. Не все правила дают строго эквивалентные URL, однако на практике эти эвристики работают неплохо [10].

Ключ страницы — своего рода хеш, получаемый в результате агрессивной нормализации, то есть применения всех правил, указанных в данном разделе.

Относительно безопасные преобразования:

а) Конвертация в нижний регистр компонентов схемы и хоста: `HTTP://www.Example.com/` в `http://www.example.com/`;

б) Удаление относительных каталогов, сегментов-точек: `http://example.com/../a/b/../c/./d` в `http://example.com/a/c/d`;

в) Удаление фрагментов: `http://example.com/bar.html#section1` в `http://example.com/bar.html`;

г) Удаление конечного слеша: `http://example.com/foo/` в `https://example.com/foo`;

д) Удаление порта по умолчанию: 80 для http и 443 для https `http://example.com:80` в `http://example.com`;

е) Перевод IDN в unicode: `http://xn--e1afmkfd.xn--80akhbyknj4f/` в `http://пример.испытание/`.

Преобразования, которые можно использовать для проверки эквивалентности двух страниц, но не при запросе:

а) Удаление головного индекса (`index.html`, `default.aspx` и другие): `http://example.com/index.html` в `http://www.example.com/`;

б) Удаление дублированных слешей: `http://example.com/foo//bar.html` в `http://example.com/foo/bar.html`;

в) Удаление `www.`: `http://www.example.com/` в `http://example.com/`;

г) Сокращение идентификаторов протокола: `https://example.com` в `http://example.com`;

д) Конвертация в нижний регистр всего URL: `HTTP://example.COM/TEST.html` в `http://example.com/test.html`;

Кроме того, так как для работы с динамическими страницами (их URL чаще всего содержат поисковую компоненту) требуется наличие отлаженного и надёжного метода определения дубликатов страницы, в данной работе было решено отказаться от подобных ссылок — к ссылке добавляется атрибут `rel="nofollow"`, а запрос отбрасывается: `http://example.com/?id=123&q=test` в `http://example.com/`.

2 Конструкторский раздел

2.1 База данных

Рассмотрим процесс перехода от ER-диаграммы (рис. 1.4) к схеме базы данных (рис. 2.1):

а) В силу регламентированности структуры данных имеет смысл использовать реляционную схему базы данных;

б) Сущность «слово» естественным образом переходит в таблицу `word`, причём без сохранения производных атрибутов (IDF), в силу тривиальности их расчёта;

в) Сущность «страница» переходит в две таблицы: `page` и `indexed`, так как все атрибуты кроме «URL» и «ключа» могут быть получены только после разбора страницы, а «URL» и «ключ» получены и из ссылок на другие страницы. На практике количество неразобранных страниц на несколько порядков больше проиндексированных, а значит использование только одной таблице ведёт к неоправданному расходу памяти;

г) Связь многие-ко-многим «содержит» переходит в таблицу `location` вместе с непроизводными атрибутами;

д) Связь многие-ко-многим «ссылается на» переходит в таблицы `link` и `linkword`. Данная денормализация вызвана необходимостью поисковой оптимизации (см. 2.4.1.1);

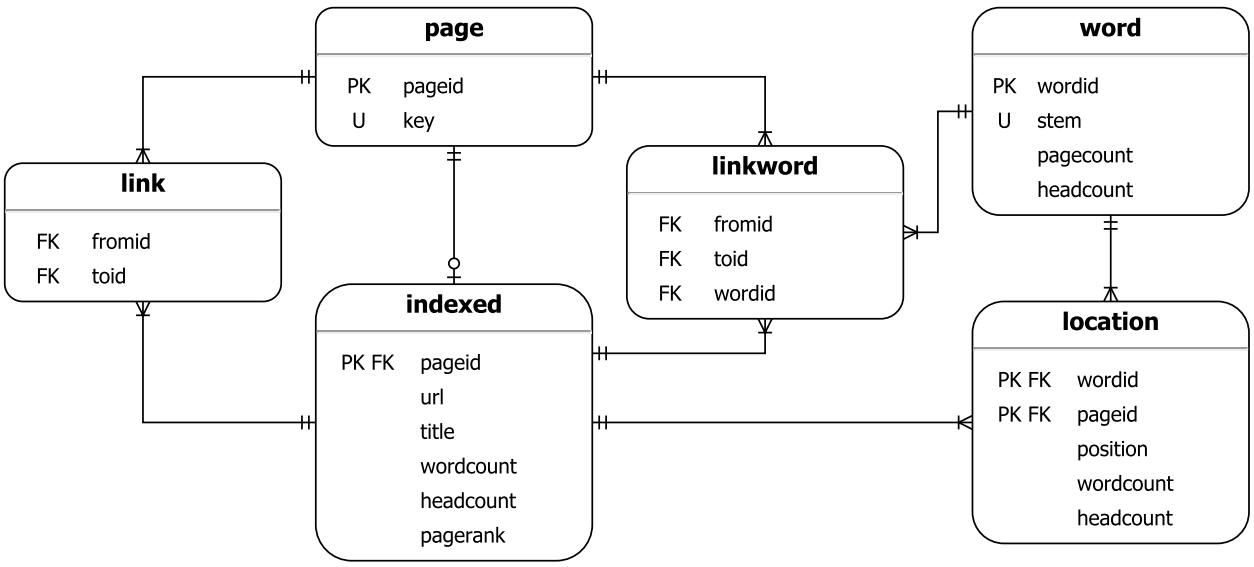

Рисунок 2.1 — Схема базы данных.

Рассмотрим подробнее каждую таблицу:

`page(pageid, key)`

Таблица содержит информацию о страницах, которые когда-либо встречались поисковым роботом. Сюда входят как посещённые страницы, так и ещё не обработанные страницы, ссылка на которые когда-либо встречалась. Поле `key` содержит уникальный ключ страницы, получающийся в результате агрессивной нормализации URL (см. раздел 1.2.3.6).

`word(wordid, stem, pagecount, headcount)`

Информация о словах, когда либо индексировавшихся роботом. Поле `stem` содержит уникальную основу слова (см. раздел 1.2.3.5), `pagecount` — количество проиндексированных страниц, содержащих данное слово в любой форме, а `headcount` — количество проиндексированных страниц, в заголовках которых встретилось данное слово (см. раздел 1.1.1.1).

`indexed(pageid, url, title, wordcount, headcount, pagerank)`

Таблица всех проиндексированных страниц, содержащая URL-адрес `url`, заголовок страницы `title`, количество слов в документе `wordcount`, количество слов в заголовках документа `headcount` и ранг страницы `pagerank` (см. раздел 1.1.2.2). Заголовок страницы не обязательно будет совпадать со значением `<title>`, особенно если последнего нет или он пустой. URL хранится в безопасно нормализованной форме (см. раздел 1.2.3.6), чтобы пользователь мог воспользоваться оригинальным URL.

`location(wordid, pageid, position, wordcount, headcount)`

Инвертированный индекс, содержащий информацию о каждом слове на странице: позиция первого вхождения `position` (см. раздел 1.1.1.3), количество вхождений в текст `wordcount` и в заголовки `headcount` (см. раздел 1.1.1.2).

`link(fromid, toid)`

Таблица ссылок, связывающая однозначно проиндексированную страницу с идентификатором `fromid` и страницу (необязательно проиндексированную) с идентификатором `toid`. Не содержит ссылок с атрибутом `rel="nofollow"` (см. раздел 1.2.3.3).

`linkword(fromid, toid, wordid)`

Таблица слов в ссылках между `fromid` и `toid`. Совокупность пар (`fromid`, `toid`) является подмножеством отношения `link`, однако замена этой пары на идентификатор `link` ведёт к усложнению поиска и невозможности некоторых оптимизаций (см. раздел 2.4.1.1). Не содержит ссылок с атрибутом `rel="nofollow"` и не относящихся к основному содержимому (см. раздел 1.2.3.2).

2.2 Поисковой робот

2.2.1 Архитектура робота

Упрощённо процесс работы такого робота состоит из следующих этапов (рис. 2.2):

а) Всё начинается с того, что пользователь роботом [1] добавляет в очередь [2] какой-то начальный набор адресов, с которого и начнётся обход;

б) Загрузчик страниц [3] получает из очереди [2] очередной адрес и производит соответствующий HTTP-запрос через сеть Интернет [4]. Чтобы избежать DNS-запросов перед загрузкой каждой страницы, IP-адрес домена кешируется;

в) В случае успешного ответа загрузчик [3] передаёт страницу в экстрактор информации [4];

г) Экстрактор информации [5] извлекает из страницы ссылки и полезную текстовую информацию. После чего ссылки добавляются в очередь [2], а текстовой документ отправляется в индексатор [6];

д) Индексатор [6] подготавливает документ и формирует запросы на сохранение в базу данных [7].

Несколько уточнений по работе поискового робота:

а) Связь между компонентами осуществляется асинхронно;

б) Используется несколько загрузчиков страниц, работающих параллельно;

в) Очередь адресов представляет собой комплексную структуру (о чём далее).

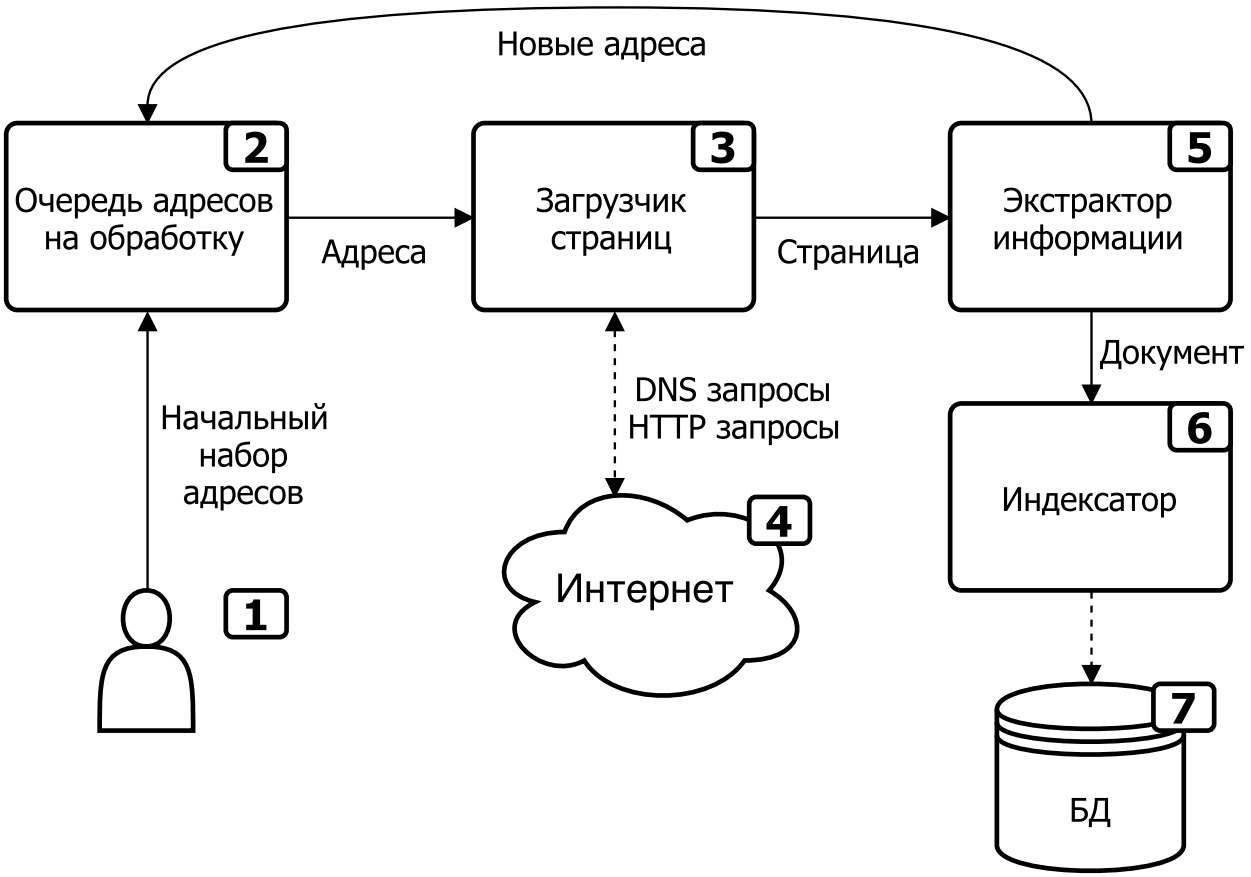

Рисунок 2.2 — Архитектура поискового робота.

2.2.2 Очередь адресов

Адреса, принадлежащие одному домену имеют общую информацию, такую как набор правил, заданного в `robots.txt` (см. раздел 1.2.2) и IP-адрес, кеширование которого позволяет уменьшить количество DNS-запросов. Поэтому необходимо группировать страницы в домены, которые и будут содержаться в очереди.

Используется очередь доменов с приоритетом, основанная на бинарной куче. В качестве приоритета задаётся время пробуждения, которое получается как сумма времени предыдущего запроса и времени, необходимого для ожидания, которое может быть получено либо из `robots.txt`, если владелец сайта явно ограничил частоту запросов, либо из соображений длительности отклика сервера: при сильных нагрузках сервера отвечают значительно реже.

Удаление доменов из очереди происходит после определённого (наперёд заданного) времени бездействия в порядке обычной обработки очереди.

Кроме того, внутри каждого домена существует очередь с приоритетом из страниц. Здесь, в качестве приоритета приняты штрафные очки страницы, которые суммируются из следующих факторов:

— Была ли ссылка на эту страницу в основном содержимом;

— Содержала ли ссылка текст;

— Текущая глубина обхода;

— Содержала ли ссылка атрибут `rel="nofollow"` (или был он добавлен в результате обработки динамической ссылки, см. раздел 1.2.3.6);

2.3 Постобработка

После сбора коллекции документов необходима некоторая обработка данных для вычисления PageRank (см. раздел 1.1.2.2) и BM25 (см. раздел 1.1.1.2).

2.3.1 Вычисление PageRank

Для правильного вычисления ранга страницы нужно иметь сеть, обладающую только внутренними ссылками, то есть такую, где все ссылки указывают только на проиндексированные страницы. Для этого создаётся и заполняется временная таблица `inboundlink` (рис. 2.3), основываясь на таблице `link` (листинг 2.1).

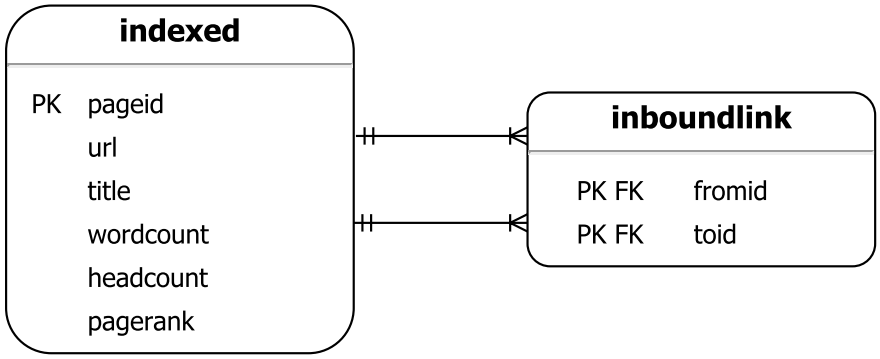

Рисунок 2.3 — Концептуальная ER-диаграмма связей внутри коллекции.

```
select fromid, toid
from indexed ifr, indexed ito join link
on fromid = ifr.pageid and toid = ito.pageid;
```

Листинг 2.1 — Выборка внутренних для коллекции ссылок.

После чего можно приступать непосредственно к вычислению PageRank. Сначала необходимо задать базовый ранг страницы, например 1, и подсчитать количество ссылок на странице (листинг 2.2).

```
select pageid, count(toid), 1.
from indexed left join inboundlink on pageid = fromid
group by pageid;
```

Листинг 2.2 — Инициализация процесса вычисления PageRank.

Значения на очередной итерации могут быть вычислены, используя 1.11, как показано на листинге 2.3. Функция `total` эквивалентна функции `sum` из стандарта SQL, за исключением того, что на пустой выборке возвращает `0`, а не `NULL`.

```
select src.pageid, src.linkcount, .15 + .85 * total(fr.pagerank/fr.linkcount)
from src left join inboundlink on src.pageid = toid
         left join src fr on fr.pageid = fromid
group by src.pageid;
```

Листинг 2.3 — Итерация процесса вычисления PageRank.

2.3.2 Общая информация

Помимо ранга страницы необходимо найти некоторую общую информацию, такую как общее количество документов в коллекции, средняя длина в словах документа и средняя длина в словах в заголовках документа (листинг 2.4).

```
select count(*), avg(wordcount), avg(headcount) from indexed;
```

Листинг 2.4 — Нахождение общей информации.

Кроме того, самое время указать СУБД на необходимость заново проанализировать базу данных (команда `analyze`), что может быть полезно для внутренней оптимизации запросов к базе данных.

2.4 Поиск

Для получения списка релевантных документов и всей необходимой для их ранжирования информации формируется один большой запрос, зависящий от количества запрашиваемых слов и их идентификаторов. Дан-

ный запрос является узким местом при поиске и должен быть максимально оптимизирован.

Пример упрощённого сформированного запроса двух слов с идентификаторами 42 и 146 представлен на листинге 2.5.

```
select
  idx.pageid,                    -- идентификатор страницы
  idx.wordcount,                 -- количество слов на странице
  idx.headcount,                 -- количество слов в заголовках страницы
  idx.pagerank,                  -- ранг страницы
  total(fromidx.pagerank),       -- реферальный ранг страницы
  l0.position + l1.position,     -- близость слов к началу
  l0.wordcount, l0.headcount,    -- частотность первого слова
  l1.wordcount, l1.headcount     -- частотность второго слова

from location l0
join location l1 using (pageid)
join indexed idx using (pageid)
left join linkword lw on lw.wordid in (42, 146)
                     and idx.pageid = toid
left join indexed fromidx on fromidx.pageid = fromid

where l0.wordid = 42 and l1.wordid = 146
group by l0.pageid
```

Листинг 2.5 — Запрос релевантных документов.

В данном запросе в результирующий набор попадают страницы, которые обладают всеми запрошенными словами. Кроме того, для каждой страницы просчитывается суммарный реферальный ранг страницы.

2.4.1 Оптимизация запроса

Так как данный запрос является ядром поиска и при этом наиболее узким местом, то необходимо максимально оптимизировать его. Рассмотрим несколько техник.

2.4.1.1 Оптимизация перебора

Рассмотрим план выполнения запроса (листинг 2.5) и попробуем оптимизировать его (листинг 2.6).

```
SEARCH TABLE location AS l0 USING PRIMARY KEY (wordid=?)
SEARCH TABLE location AS l1 USING PRIMARY KEY (wordid=? AND pageid=?)
```

```
SEARCH TABLE indexed AS idx USING PRIMARY KEY (pageid=?)
SCAN TABLE linkword AS lw
SEARCH TABLE indexed AS fromidx USING PRIMARY KEY (pageid=?)
```

Листинг 2.6 — Неоптимальный план выполнения запроса.

`SCAN TABLE` в плане запроса указывает на полное сканирование таблицы `linkword`. Для ускорения поиска требуется создать дополнительный индекс (листинг 2.7).

```
create index wordidtoididx on linkword(wordid, toid);
```

Листинг 2.7 — Создание индекса.

Теперь запрос будет выполнен максимально оптимизированным образом (листинг 2.8): со сложностью $O(\log n)$ вместо $O(n)$ при поиске по таблице `linkword` в силу использования B/B+-деревьев внутри движка запросов СУБД.

```
SEARCH TABLE location AS l0 USING PRIMARY KEY (wordid=?)
SEARCH TABLE location AS l1 USING PRIMARY KEY (wordid=? AND pageid=?)
SEARCH TABLE indexed AS idx USING PRIMARY KEY (pageid=?)
SEARCH TABLE linkword AS lw USING INDEX wordidtoididx (wordid=? AND toid=?)
EXECUTE LIST SUBQUERY 1
SEARCH TABLE indexed AS fromidx USING PRIMARY KEY (pageid=?)
```

Листинг 2.8 — Оптимальный план выполнения запроса.

2.4.1.2 Предварительная сортировка слов

При выполнения запроса (листинг 2.5) большую роль играет последовательность слов. Действительно, если первое слово встречается чаще, чем последующие, то интерпретатору запроса придётся проверить больше документов. Сортировка слов по возрастанию их частоты позволяет отбрасывать нерелевантные документы раньше.

2.5 Сервер и поисковая страница

Для того, чтобы пользователь смог воспользоваться графическим интерфейсом поисковой системы, необходим работающий веб-сервер, который отдаст поисковую страницу в ответ на запрос. Поскольку запросы могут выполняться продолжительное время, веб-сервер работает в асинхрон-

ном режиме, а все запросы к базе данных осуществляются в отдельном потоке.

Поисковый запрос передаётся методом `GET` с передачей параметров запроса (сам текст запроса и количество страниц, которые следует пропустить после ранжирования). Чтобы текст запроса был передан без искажений производится кодирование текста как компонента URI, путём замены всех символов, за исключением латинских букв, цифр и знаков «-», «_», «.», «!», « », «*», «'», «(», «)». Например, пользователь может произвести запрос с текстом «Python&language», что выделит новую пару ключ-значение из-за использования амперсанда. Поэтому вместо одного параметра «Python&language» будут получены два и запрос будет произведён только для «Python».

3 Технологический раздел

3.1 Используемые технологии

Язык программирования

Поставленная задача требует много работы с I/O: загрузка файлов и работа с базой данных. Вычислений производится относительно немного (стемминг и выделение основного содержимого), поэтому очевиден выбор асинхронной модели. Относительно низкоуровневые варианты (Си + libuv/libev/libevent или rust + mio) не дадут выигрыша из-за малого количества вычислений. Первоначально синхронные python и lua имеют синхронные обёртки баз данных, что усложняет их встраивание в существующие асинхронные фреймворки (в случае с python это asyncio). С другой стороны, есть первоначально асинхронный и при этом достаточно популярный сегодня node.js.

Дополнительным преимуществом использования node.js является единый язык программирования (javascript) на клиенте и сервере. Таким образом, в качестве ЯП был выбран EcmaScript 2015 — будущая версия ЯП JavaScript. Однако его поддерживают пока далеко не все браузеры, поэтому исходный код для поисковой страницы транслируется компилятором babel в Javascript 1.5 — предыдущую версию языка.

База данных

В качество базы данных было решено взять небольшую, но при этом достаточно функциональную и быструю, СУБД SQLite, которая размещает базу данных в одном единственном файле (не считая временных файлов журнала). Однако запросы специально составлялись наиболее переносимым образом, чтобы программу можно было легко адаптировать к другим СУБД (например, PostgreSQL). Главным преимуществом использования встраиваемой базы данных является просто установки и настройки: нет необходимость запускать сервер СУБД, поскольку логика его работы «встраивается» в приложение.

Используемые библиотеки

В среде node.js-разработки общепринят UNIX-подход: множество небольших библиотек, каждая из которых решает только одну задачу, но эффективно. Стандартный пакетный менеджер позволяет легко устанавливать все зависимости одной командой.

bloomfilter — эффективная реализация фильтра Блума;
sqlite3 — асинхронный интерфейс к sqlite;
entities — обнаружение и замена мнемоник (X)HTML;
htmlparser2 — высокопроизводительный SAX-парсер (X)HTML;
koa — небольшой асинхронный веб-фреймворк;
natural — работа с естественными языками;
priorityqueuejs — высокопроизводительная реализация пирамиды;
readabilitySAX — выделение основного содержимого;
request — упрощение запросов;
yargs — построение командных интерфейсов;

Установка зависимостей производится командой `npm install`, а полная сборка проекта по команде `npm run build`.

Окружение разработчика

В качестве редактора кода был использован Vim на ОС Arch Linux. Сборка проекта осуществляется пакетным менеджером npm — родным инструментом для разработчиков на JS. В качестве системы контроля версий использовался git.

Тестирование

Для большей части модулей, входящих в программу, реализовано модульное тестирование — тестирование отдельного модуля для проверки корректности его работы в штатных и исключительных ситуациях. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии.

Тесты представлены в директории `test/*`. Тестирование запускается по команде `npm test`.

3.2 Пользовательский интерфейс

Интерфейс командной строки

Управление большей частью системы осуществляется через интерфейс командной строки. Входная точка представлена на листинге 3.1.

```
Usage: cli <command> ... (see -h)

Commands:
  crawl      browse WWW and index information
  pagerank   precalculate pagerank
  search     request for indexed information
  server     start the web server

Options:
  -d, --database  specify path to database          [string] [default: "se.db"]
  -h, --help      show help                                             [boolean]
```

Листинг 3.1 — Интерфейс командной строки.

Поисковая страница

ОБНОВИТЬ КАРТИНКИ И ЛИСТИНГИ ПОСЛЕ НАБОРА НОВОЙ БАЗЫ И ПОДБОРА ВЕСОВ

Поисковая страница позволяет пользователю осуществлять запросы и получать ранжированный набор результатов (рис. 3.1). Для переключения между блоками страниц предусмотрен специальный переключатель (рис. 3.2) для запроса дополнительных страниц.

Все поисковые запросы со страницы выполняются без перезагрузки самой страницы.

Поисковой робот

Данная команда запускает поискового робота, осуществляющего сбор и индексацию информации с сайтов сети (листинг 3.2). Процесс работы робота показан на листинге 3.3.

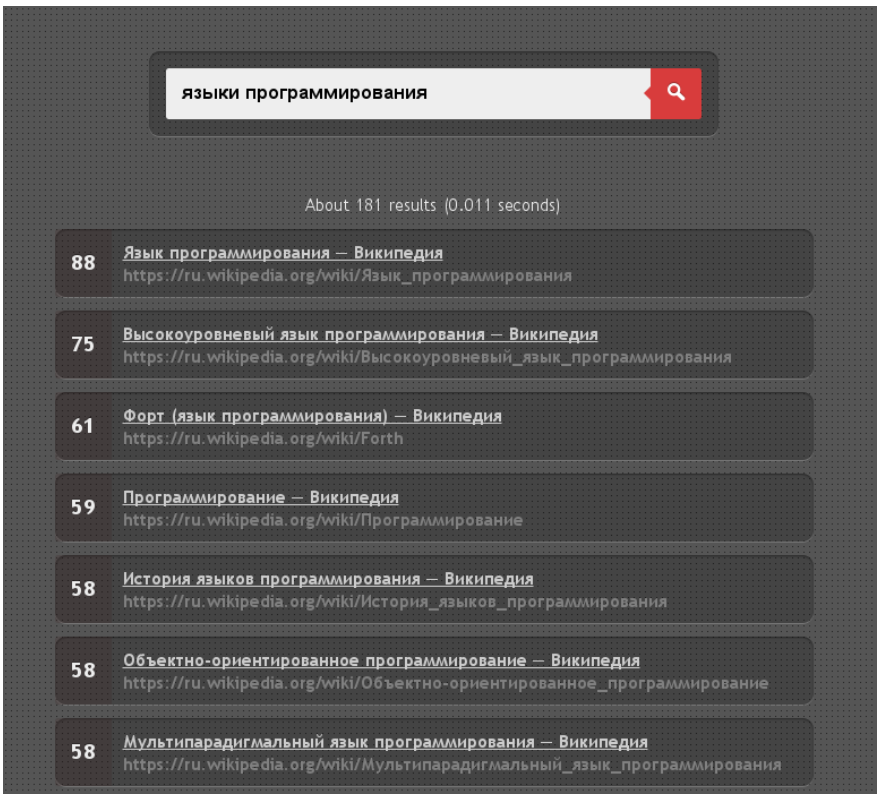

Рисунок 3.1 — Поисковая страница.

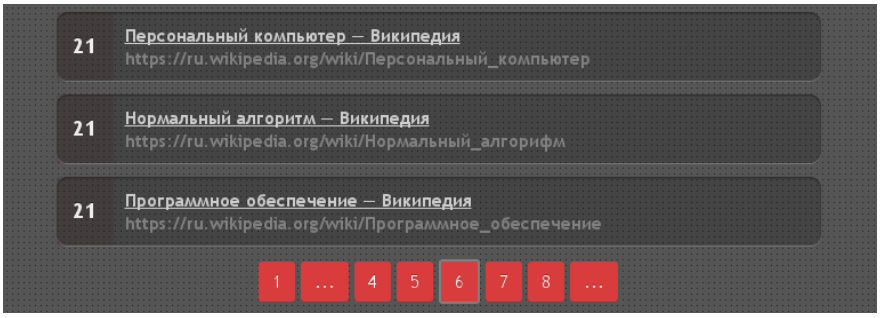

Рисунок 3.2 — Переключение групп страниц выдачи.

```
Usage: cli crawl [options] <urls...>

Options:
  -d, --database          specify path to database    [string] [default: "se.db"]
  -h, --help              show help                                    [boolean]
  -m, --max-depth         how far the crawler can go    [number] [default: 4]
  -t, --timeout           a waiting time for a response [number] [default: 15]
  -s, --max-size          max file size to process      [number] [default: 16]
  -r, --relax-time        how long to hold an empty domain [number] [default: 10]
  -g, --no-guessing       don't guess link relevant by url             [boolean]
  -i, --ignore-nofollow   ignore rel="nofollow"                        [boolean]
  -l, --link-stem-limit   limit stems per link          [number] [default: 10]
```

Листинг 3.2 — Интерфейс командной строки: поисковой робот.

```
D: 78523    I: 77890 (51210/h)    S: 1:32    Q: 4960|32329
[D] http://www.nature.com/nature/journal/v454/n7204/full/nature07130.html
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Downloaded: 78523
Indexed: 77890
Spent: 1:32
```

Листинг 3.3 — Интерфейс командной строки: поисковой робот.

Постобработка

Данная команда предназначена для запуска процесса постобработки — вычисления PageRank проиндексированных страниц и некоторой статистики (листинги 3.4 и 3.5).

```
Usage: cli pagerank [options]

Options:
  -d, --database    specify path to database        [string] [default: "se.db"]
  -h, --help        show help                                        [boolean]
  -i, --iterations  the number of iterations          [number] [default: 30]
```

Листинг 3.4 — Интерфейс командной строки: постобработка.

```
collecting inbound links
collecting initial data
iteration #0
iteration #1
...
iteration #28
iteration #29
filling index
updating info
analyzing tables
done
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Spent: 37m 53s
```

Листинг 3.5 — Интерфейс командной строки: постобработка.

Сервер

Данная команда запускает сервер, задачей которого является передача поисковой страницы пользователю с последующими принятием и обработкой поисковых запросов (листинги 3.6 и 3.7).

```
Usage: cli server [options]

Options:
  -d, --database  specify path to database          [string] [default: "se.db"]
  -h, --help      show help                                            [boolean]
  -p, --port      specify the port                  [number] [default: 3000]
  -l, --limit     pages per request limit           [number] [default: 15]
```

Листинг 3.6 — Интерфейс командной строки: сервер.

```
GET / - 7ms
GET /style.css - 2ms
GET /index.js - 1ms
GET /favicon.ico - 1ms
GET /search?q=python%20%D1%8F%D0%B7%D1%8B%D0%BA&o=0 - 23ms
```

Листинг 3.7 — Интерфейс командной строки: сервер.

Поиск

Данная команда предназначена для проведения поиска и обладает большими возможностями по сравнению с поисковой страницей (листинги 3.8 и 3.9).

```
Usage: cli search [options] <query>

Options:
  -d, --database  specify path to database          [string] [default: "se.db"]
  -h, --help      show help                                            [boolean]
  -l, --limit     the number of pages               [number] [default: 10]
  -o, --offset    the number of skip pages          [number] [default: 0]
  -v, --verbose   provide more useful info                             [count]
```

Листинг 3.8 — Интерфейс командной строки: поиск.

```
[88] Язык программирования — Википедия | https://ru.wikipedia.org/wikiЯзык/...
[75] Высокоуровневый язык программирования — Википедия | https://ru.wikiped...
[61] Форт язык( программирования) — Википедия | https://ru.wikipedia.org/wi...
[59] Программирование — Википедия | https://ru.wikipedia.org/wikiПрограмми/...
[58] История языков программирования — Википедия | https://ru.wikipedia.org...
```

```
[58] Объектноориентированное– программирование — Википедия | https://ru.wik...
[58] Мультипарадигмальный язык программирования — Википедия | https://ru.wi...
[58] Мультипарадигмальный язык программирования — Википедия | https://ru.wi...
[53] Язык ассемблера — Википедия | https://ru.wikipedia.org/wikiЯзык/ассем_...
[50] Парадигма программирования — Википедия | https://ru.wikipedia.org/wiki...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  About 181 results (0.013 seconds)
```

Листинг 3.9 — Интерфейс командной строки: поиск.

Данный команда позволяет контролировать детальность вывода аргументами `-v` и `-vv` (листинг 3.10).

```
[88] Язык программирования — Википедия | https://ru.wikipedia.org/wikiЯзык/...
     scores: wbm=0.91 hbm=0.99 cnt=0.64 pos=1.00 ref=1.00 pr=0.49
     words: 2402   heads: 11   total pos: 3   PR: 1.01   ref PR: 40.29

[75] Высокоуровневый язык программирования — Википедия | https://ru.wiped...
     scores: wbm=0.90 hbm=1.00 cnt=0.12 pos=1.00 ref=0.91 pr=0.10
     words: 469   heads: 4   total pos: 5   PR: 0.33   ref PR: 11.86

[59] Программирование — Википедия | https://ru.wikipedia.org/wikiПрограмми/...
     scores: wbm=0.91 hbm=0.62 cnt=0.16 pos=0.94 ref=0.52 pr=0.29
     words: 628   heads: 2   total pos: 81   PR: 0.67   ref PR: 6.90

[58] История языков программирования — Википедия | https://ru.wikipedia.org...
     scores: wbm=0.86 hbm=1.00 cnt=0.25 pos=1.00 ref=0.02 pr=0.01
     words: 948   heads: 4   total pos: 5   PR: 0.17   ref PR: 0.35

[58] Объектноориентированное– программирование — Википедия | https://ru.wik...
     scores: wbm=0.78 hbm=0.52 cnt=1.00 pos=0.91 ref=0.16 pr=0.24
     words: 3794   heads: 4   total pos: 121   PR: 0.58   ref PR: 2.25
  ...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  About 181 results (0.025 seconds)
```

Листинг 3.10 — Интерфейс командной строки: поиск.

Заключение

В ходе выполнения работы был реализован программный продукт, полностью отвечающий требованиям, изложенным в техническом задании, а именно поисковая система для сбора информации с сети Интернет и выполнения полнотекстового поиска по ней, предоставляющая интерфейс командой строки и графический интерфейс пользоватя для выполнения запросов.

В течение выполнения проекта я провёл анализ существующих методов, позволяющих решить поставленные задачи, на основе проведённого анализа выбрал те методы, который наиболее подходит для решения задачи, изучил пути оптимизации и ускорения выбранного метода, разработал свой алгоритм решения задачи и программу.

Список использованных источников

1. *Segaran, Toby.* Programming Collective Intelligence / Toby Segaran. — O'Reilly, 2007.

2. *Robertson, Stephen.* The Probabilistic Relevance Framework: BM25 and Beyond / Stephen Robertson, Hugo Zaragoza // *Found. Trends Inf. Retr.* — 2009. — apr. — Vol. 3, no. 4. — Pp. 333–389.

3. *Lv, Yuanhua.* Lower-bounding Term Frequency Normalization / Yuanhua Lv, ChengXiang Zhai // Proceedings of the 20th ACM International Conference on Information and Knowledge Management. — ACM, 2011. — Pp. 7–16.

4. *Lv, Yuanhua.* When Documents Are Very Long, BM25 Fails! / Yuanhua Lv, ChengXiang Zhai // Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval. — ACM, 2011. — Pp. 1103–1104.

5. *Broder, Andrei.* Network Applications of Bloom Filters: A Survey / Andrei Broder, Michael Mitzenmacher, Andrei Broder I Michael Mitzenmacher // Internet Mathematics. — 2002. — Pp. 636–646.

6. *Pomikálek, Jan.* Removing boilerplate and duplicate content from web corpora: Ph.D. thesis / Masaryk university, Faculty of informatics, Brno, Czech Republic. — 2011.

7. *Kohlschütter, Christian.* Boilerplate Detection Using Shallow Text Features / Christian Kohlschütter, Peter Fankhauser, Wolfgang Nejdl // Proceedings of the Third ACM International Conference on Web Search and Data Mining. — ACM, 2010. — Pp. 441–450.

8. *Porter, M. F.* Readings in Information Retrieval / M. F. Porter. — Morgan Kaufmann Publishers Inc., 1997. — Pp. 313–316.

9. *Porter, M. F.* Russian stemming algorithm. — 2007. `http://snowball.tartarus.org/algorithms/russian/stemmer.html`.

10. *Pant, Gautam.* Crawling the Web / Gautam Pant, Padmini Srinivasan, Filippo Menczer // In Web Dynamics: Adapting to Change in Content, Size, Topology and Use. Edited by M. Levene and A. Poulovassilis. — Springer-Verlag, 2004. — Pp. 153–178.