Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторные работы

по курсу «Информационный поиск»

Выполнил: Баталин Д.А.

Группа: М8О-412Б-22

Проверил: Кухтичев А.А.

Дата:

Оценка:

Москва, 2025

Оглавление

ЛР №1. Добыча корпуса документов

1. Источники данных (Source Data)

Для построения учебного поискового индекса был выбран корпус документов тематики "Информационные технологии". Источниками послужили два ресурса с принципиально разной структурой верстки:

- Habr.com - современный ресурс с сложной DOM-структурой, обилием JavaScript-скриптов и CSS-стилей.
- OpenNet.ru - новостной ресурс "старой школы" с минималистичной HTML-версткой. Всего для анализа было скачано 6 документов (по 3 с каждого источника).

2. Характеристики документов

В ходе анализа "сырых" данных были выявлены следующие особенности:

- Habr.com: Документы используют стандарт HTML5. В коде содержится значительное количество мета-информации (OpenGraph теги og:title, og:description, JSON-LD разметка для поисковиков). Большую часть объема файла занимают технические данные (inline SVG иконки, скрипты React/Vue, CSS классы).
- OpenNet.ru: Документы имеют более простую структуру. Мета-информация представлена минимально (keywords, description).
- Кодировка: Habr использует UTF-8, OpenNet часто отдает контент в KOI8-R или CP1251, что потребовало автоматического определения кодировки при скачивании.

3. Статистические данные и анализ корпуса

Источник	Кол-во док.	Ср. размер Raw	Ср. размер Text	Доля текста
Habr	3	304.62	14.88	4.76%
OpenNet	3	161.43	49.99	30.96%
Итого	6	233.02	32.24	13.83%

ЛР №2. Поисковый робот

1. Цель работы

Разработка автоматизированного сборщика (краулера) текстовых документов для формирования корпуса данных объемом более 30 000 статей.

2. Архитектура решения

Робот реализован на языке Python с использованием библиотек requests и pymongo.

Компоненты системы:

- Конфигурационный файл (config.yaml). Содержит настройки подключения к БД, параметры задержек, целевые диапазоны ID статей и шаблоны URL.
- State-менеджер. Робот сохраняет текущий прогресс (ID последней обработанной статьи) в JSON-файл (crawler_state_source.json). Это позволяет безопасно прерывать работу через ctrl+c и возобновлять её с места остановки.

- Параллельная работа: Архитектура позволяет запускать несколько независимых экземпляров робота для разных источников, которые пишут в одну базу данных.

3. Структура хранимых данных

В коллекции raw_docs базы данных MongoDB сохраняются документы следующего формата:

- url: Нормализованный URL документа (используется как уникальный индекс unique=True во избежание дубликатов).
- raw_html: Полный HTML-код страницы без предварительной обработки.
- source: Метка источника (habr или opennet).
- crawled_at: Unix timestamp времени скачивания.

Для реализации обновления (переобкачки) используется метод update_one с параметром upsert=True: если документ с таким URL уже существует, он обновляется, если нет - создается новый.

4. Проблемы и решения

Проблема блокировок:

В ходе сбора данных с источника OpenNet.ru IP-адрес робота был заблокирован (получен ответ с текстом «Flood detected...» вместо контента). Это привело к попаданию в базу "мусорных" документов.

Решение:

1. Очистка данных: Написан вспомогательный скрипт, удаливший из MongoDB документы, содержащие фразу-маркер блокировки.

2. Доработка логики: В робота внедрена функция is_banned(html), анализирующая ответ сервера.
    - Для OpenNet - поиск специфических фраз («Flood detected», «Stop it»).
    - Для Habr - обработка HTTP-кодов 429 и 503, а также проверка на наличие заглушек Qrator/DDoS-Guard.
    - При обнаружении бана робот автоматически приостанавливает работу на 10 минут.
3. Адаптивные задержки: Для каждого источника настроены индивидуальные задержки, к которым добавляется случайное отклонение (jitter +-30%) для имитации поведения человека.

Особенности источников:

- Для Habr.com добавлен параметр step: 2, так как статьи располагаются только на четных ID, что позволило ускорить сбор в два раза.
- Для OpenNet увеличена пауза между запросами до 5-6 секунд из-за строгой политики фаервола.

ЛР №3-5. Токенизация. Стемминг. Закон Ципфа

Токенизация

1. Методика токенизации

Для разбиения текста на токены был разработан алгоритм, учитывающий особенности технического текста. Использован посимвольный проход с анализом контекста.

Правила выделения токенов:

- Базовое правило: Любая последовательность букв или цифр считается частью токена.
- Спецсимволы-разделители: Все символы, кроме букв и цифр, считаются разделителями, за исключением следующих случаев:
    a) Точка (.) - считается частью токена, если находится между двумя цифрами или буквами (сохраняет целостность версий 2.0, IP-адресов 127.0.0.1 и дробных чисел 3.14). Точка в конце предложения отсекается.
    b) Дефис (-) – считается частью токена, если находится между двумя буквами (wi-fi, back-end).
    c) Плюс (+) – считается частью токена, если стоит после буквы или другого плюса (C++, g++, notepad++).
    d) Подчеркивание (_) – сохраняется внутри слов (для переменных snake_case).

Приведение к нижнему регистру:

Реализована собственная функция обработки UTF-8 строк, потому что стандартная функция tolower не работает с многобайтовой кириллицей.

2. Статистические результаты

На обработанном корпусе из ~30 000 документов получены следующие данные:

- Количество токенов: 74 350 557
- Средняя длина токена в символах: 5.29.
- Средняя длина токена в байтах: 8.79.
- Скорость обработки: ~115 MB/sec.

3. Достоинства и недостатки

- Достоинства: Высокая скорость, корректная обработка специфичных IT-терминов (C++, .net).
- Недостатки: Не распознаются сложные URL-адреса (они разбиваются на домены), не обрабатываются сокращения (например, т.е.).

Закон Ципфа

1. Анализ распределения

Для корпуса был построен частотный словарь и график рангового распределения в двойном логарифмическом масштабе.

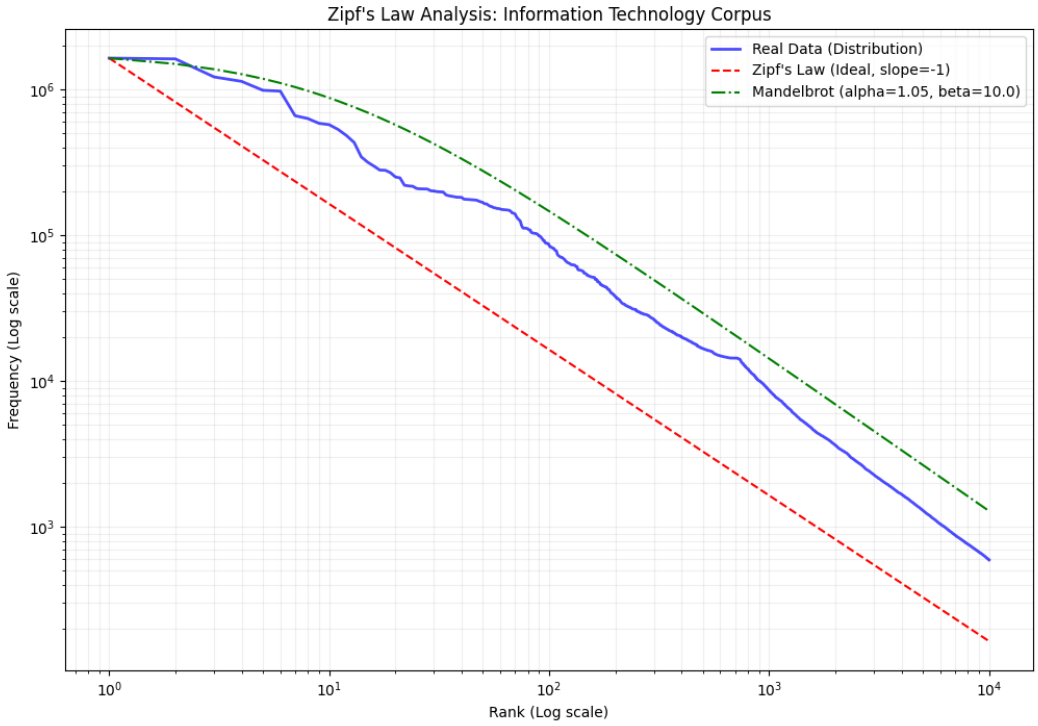

Рисунок 1. График закона Ципфа

Результаты:

- График подтверждает закон Ципфа: наблюдается линейное убывание частоты от ранга (log(Freq) ~ -log(Rank)).

- Отклонение: Реальная кривая (синяя) проходит выше идеальной прямой Ципфа (красная). Это объясняется высокой плотностью терминологии в IT-корпусе. Словарь "средней частоты" (ранги 100–5000) используется активнее, чем в общелитературном языке, что дает более пологое падение кривой.

2. Закон Мандельброта

Для аппроксимации начального участка графика (топ-100 самых частых слов) были подобраны параметры закона Мандельброта:

$$P(r) \propto \frac{1}{(r+\beta)^{\alpha}}$$

Константы $\alpha = 1.05, \beta = 10.0$. ведение параметра смещения β позволило сгладить шапку графика, где закон Ципфа давал слишком резкое падение.

3. Статистический анализ словаря

На основе частотного анализа корпуса получены следующие контрольные точки распределения:

Топ частотных слов:

- Ранг 1: «и» — 1 644 619 вхождений.
- Ранг 10: «для» — 574 915 вхождений.
- Ранг 100: «без» — 85 626 вхождений.

Топ списка занимают союзы и предлоги. Это соответствует теории информационного поиска: самые частотные слова несут наименьшую смысловую нагрузку.

Слова с частотой 1:

- Количество: 766 409 слов.

Доля от словаря: 53.27%. Более половины уникальных термов в индексе встречаются в корпусе всего один раз. Это демонстрирует длинный хвост распределения и указывает на потенциальную возможность сжатия индекса путем отсечения слишком редких слов без существенной потери качества поиска.

Стемминг

Вместо полной лемматизации был реализован простой алгоритм стемминга с отсечением суффиксов и окончаний.

Алгоритм работает по жадному принципу: проверяется наличие окончания из предопределенного списка для русского и английского, начиная с самых длинных (-вшимися, -ization) до самых коротких (-а, -s).

Оценка качества поиска

Было проведено сравнение количества найденных документов по точному совпадению и после применения стемминга.

Пример: Запрос "система"

- Точное совпадение: 8074 документов.
- Со стеммингом (систем): 20 569 документов.
- Результат: Найдены словоформы системы, системе, системам, системах, систему. Реколл значительно улучшился.

Пример: Запрос "банка"

- Точное совпадение: 826 документов.
- Со стеммингом (систем): 2 252 документов.
- Результат: Найдены словоформы банк, банку, банков, банке, банках.

Проблема: Стеммер может ошибочно объединять разные по смыслу слова. Например банка (стеклянная) может быть урезана до банк, что приведет к смешиванию контекстов.

Вывод: Внедрение стемминга критически важно для поиска по русскому языку из-за богатой морфологии. Это увеличивает полноту поиска в 2-3 раза, хотя и незначительно снижает точность на омонимах.

ЛР №6. Булев индекс

1. Формат индекса

Для хранения поискового индекса был разработан собственный бинарный формат данных, обеспечивающий компактное хранение и быстрый доступ без использования СУБД. Индекс разделен на два файла. Используется порядок байт Little-Endian.

А. Прямой индекс (docs.bin)

Служит для получения мета-данных (URL, Заголовок) по идентификатору документа (DocID).

Структура файла (побайтово):

- Header (4 байта) - TotalDocs (uint32) - общее количество документов.
- Offset Table (N * 8 байт) - Массив чисел uint64. Значение Offset[i] указывает на абсолютную позицию начала данных для документа с ID=i.
- Data Area: Последовательность записей. Формат одной записи:
    (a) UrlLen (2 байта, uint16).
    (b) URL (UrlLen байт, ASCII).
    (c) TitleLen (2 байта, uint16).
    (d) Title (TitleLen байт, UTF-8).

Б. Обратный индекс (index.bin)

Связывает термы с идентификаторами документов, в которых они встречаются.

Структура файла:

1) Header (12 байт):
   - TotalTerms (4 байта, uint32) — количество уникальных слов.
   - DictSize (8 байт, uint64) — размер секции словаря в байтах.
2) Dictionary Section: Список словарных статей. Записи идут подряд переменной длины:
   - TermLen (1 байт, uint8).
   - Term (TermLen байт, UTF-8).
   - DocFreq (4 байта, uint32) - количество документов, содержащих термин.
   - PostingsOffset (8 байт, uint64) - смещение начала списка DocID относительно начала секции постингов.
3) Postings Section: Сплошной массив DocID (uint32). Списки для разных слов идут друг за другом.

2. Алгоритм построения (BSBI)

Использован подход Block Sort-Based Indexing.

- Токенизация: Весь корпус обрабатывается потоково. Пары (Token, DocID) сохраняются в единый массив структур в оперативной памяти.
- Сортировка: Массив сортируется лексикографически по токену, а затем по DocID. Использован алгоритм quicksort, сложность O(NlogN)
- Сжатие: Повторяющиеся пары удаляются (std::unique), формируя уникальные вхождения.

- Сброс на диск: Отсортированные данные последовательно записываются в index.bin.

Достоинства метода:

- Высокая скорость работы в памяти.
- Последовательная запись на жесткий диск (избегает random seek), что оптимально для HDD/SSD.
- Отсутствие сложных динамических структур (деревьев), что упрощает код и уменьшает оверхед по памяти.

Недостатки:

- Требует загрузки всех пар токен-документ в RAM. При превышении объема физической памяти потребуется переход на SPIMI (Single-Pass In-Memory Indexing).

Масштабируемость: Сейчас алгоритм ограничен объемом RAM.

Что если данных в 1000 раз больше?

- Ответ: Оперативная память закончится.
- Решение: SPIMI (Single-Pass In-Memory Indexing). Строить индекс кусками по 1 ГБ, сохранять их на диск, а потом сливать (Merge Sort) несколько файлов индексов в один итоговый.

ЛР №7. Булев поиск

1. Архитектура системы

Поисковая система реализована в виде двух компонентов:

- Backend: Высокопроизводительная утилита на C++. Загружает индексы в память и выполняет математические операции над множествами. Использует бинарный поиск по словарю для нахождения терминов.
- Frontend: Веб-сервер на Python + Flask. Предоставляет графический интерфейс, принимает запросы пользователя и отображает результаты. Взаимодействие с backend происходит через CLI-аргументы.

2. Обработка запросов

Парсер поддерживает операторы &&, ||, !, ( ).

Алгоритм:

- Препроцессинг: Вставка неявных операторов AND (между двумя словами подряд).
- Shunting-yard: Преобразование инфиксной нотации (человеко-читаемой) в обратную польскую запись. Пример:
  i) Вход: (A || B) && !C
  ii) RPN: A B || C ! &&
- Выполнение: Стековая машина вычисляет результат, используя операции над сортированными списками:
  i) AND - пересечение (Intersection, линейный проход).
  ii) OR - объединение (Union, линейный проход).
  iii) NOT - разность множества всех документов и текущего списка.

3. Тестирование и Производительность

Скорость поиска: Время выполнения булевых операций составляет < 10 мс. Основное время затрачивается на извлечение заголовков документов с диска для отображения сниппетов.

Тестовые сценарии:

московский авиационный институт (эквивалентно AND) - проверено, что находятся документы, где есть все три слова.

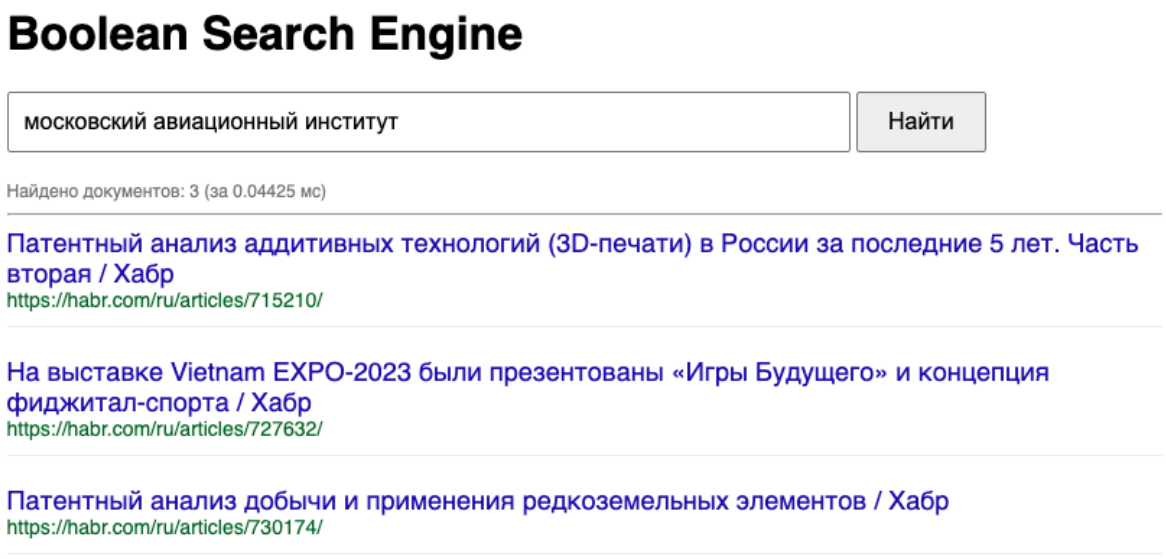

Рисунок 2. Демонстрация поиска

java || python - количество результатов совпадает с суммой уникальных документов по каждому слову.

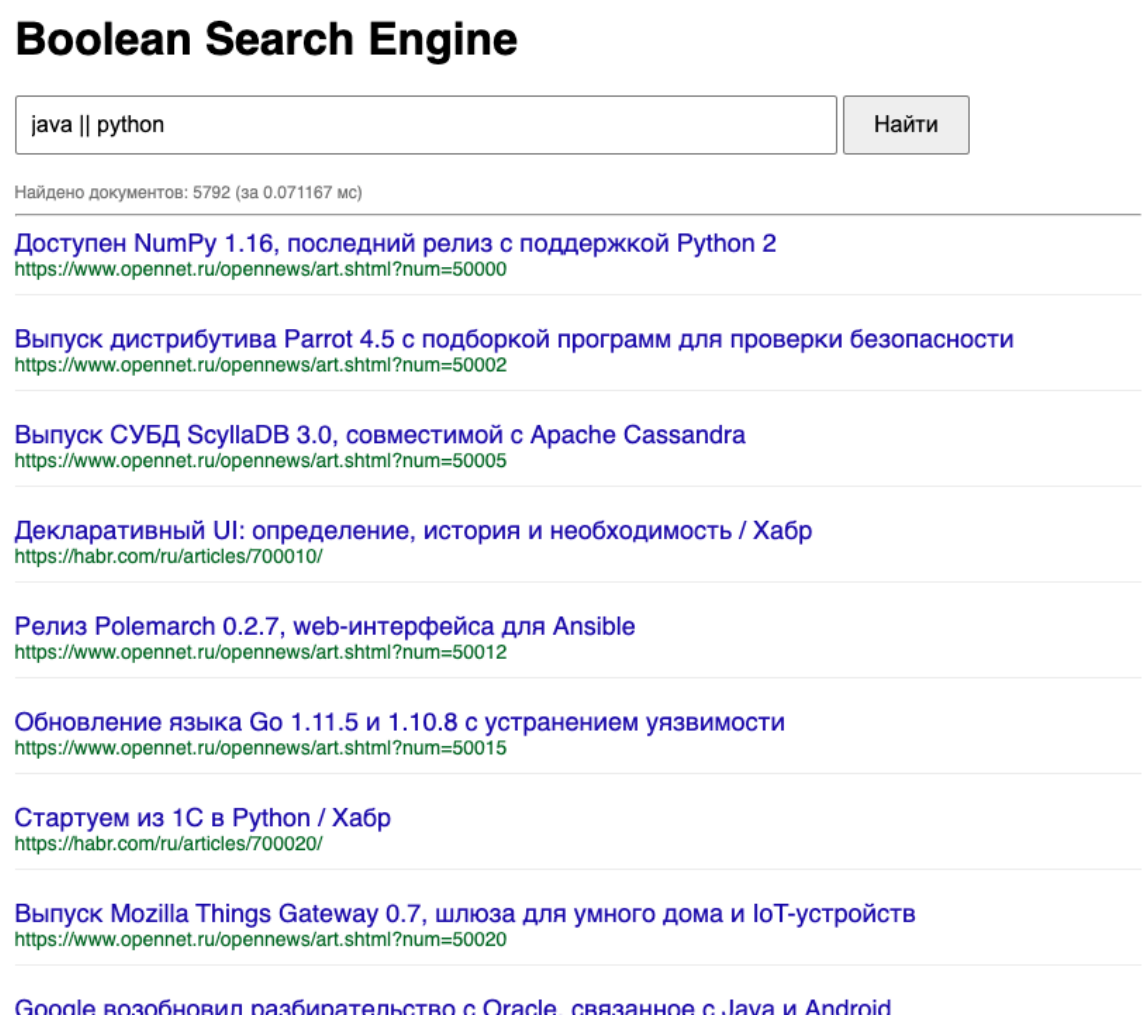

Рисунок 3. Демонстрация запроса “java || python”

руки !ноги - проверено отсутствие слова "ноги" в найденных документах.

Рисунок 3. Демонстрация запроса “руки”

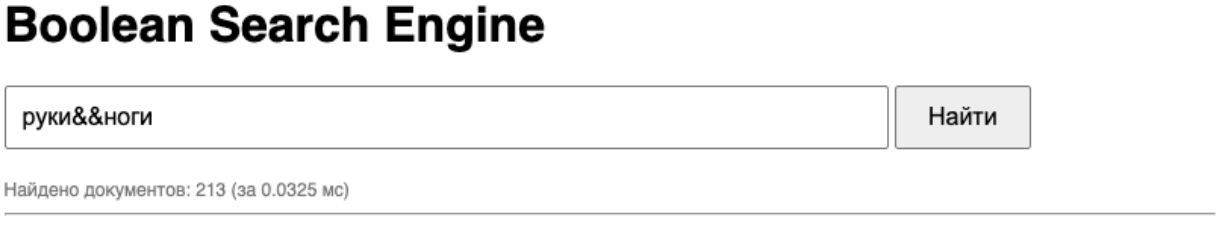

Рисунок 3. Демонстрация запроса “руки&&ноги”

Рисунок 4. Демонстрация запроса “руки ноги”

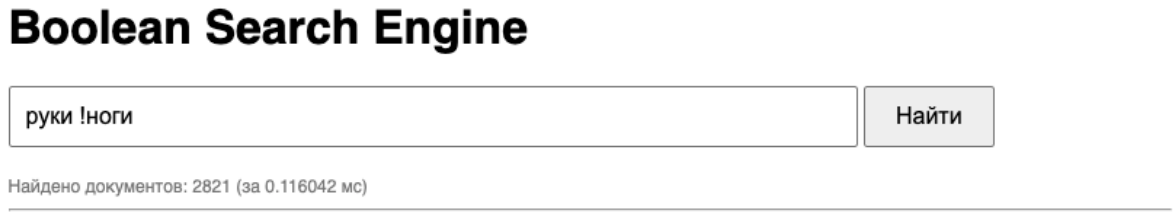

Рисунок 4. Демонстрация запроса “руки !ноги”

По приложенным демонстрационным скриншотам видно, что количество найденных документов меняется в соответствии с нашими запросами.

Вывод

В ходе выполнения цикла лабораторных работ была спроектирована и реализована полнофункциональная поисковая система, поддерживающая булеву логику запросов. Был пройден полный цикл разработки: от сбора сырых данных до реализации веб-интерфейса и своего бинарного формата хранения индекса.

Поисковый движок, написанный на C++, продемонстрировал высокую производительность. Использование бинарного поиска по словарю и алгоритма сортировочной станции для парсинга запросов обеспечивает время отклика менее 10 мс даже на сложных булевых выражениях с вложенными скобками.

В результате работы создана масштабируемая архитектура поисковой системы. Система устойчива к росту объема данных в пределах оперативной памяти. Для дальнейшего масштабирования архитектура позволяет легко перейти от BSBI к алгоритму SPIMI и распределенному поиску, так как формат бинарных блоков уже оптимизирован для последовательного слияния.