

**Московский авиационный институт
(национальный исследовательский университет)**

**Факультет информационных технологий и прикладной
математики**

**Кафедра вычислительной математики и
программирования**

**Лабораторные работы по курсу «Информационный
поиск»**

Студент: М. В. Калущий

Преподаватель: А. А. Кухтичев

Группа: М8О-206Б

Дата: 16.12.2025

Оценка:

Подпись:

Москва, 2025

Добыча корпуса документов

Цель работы

Анализ источников данных, выбор инструментов извлечения текста и получение статистических характеристик корпуса документов для построения поисковой системы.

Источники данных

Для формирования корпуса были выбраны и скачаны примеры документов из двух источников тематики «Информационные технологии»:

Habr.com — профессиональное сообщество IT-специалистов.

Характеристика: Тексты содержат большое количество профессионального сленга, англицизмов и фрагментов программного кода.

Wikipedia.org (Ru) — свободная энциклопедия (категория «Компьютерные науки»).

Характеристика: Научный/энциклопедический стиль, строгая структура, наличие плотной сети перекрестных ссылок.

Характеристика документов

В ходе анализа «сырого» HTML-кода выявлено:

- Формат: HTML5, кодировка UTF-8.
- Структура текста: Основной контент заключен в теги `<p>` (параграфы), `<h1>`-`<h6>` (заголовки). Списки оформлены через ``/``.

Особенности разметки:

- На Хабре код обернут в теги `<pre>``<code ...>`. Это важно учитывать при индексации (либо исключать, либо индексировать специфически).
- В Википедии множество ссылок `<a>`, скрытых сносок и формул (LaTeX/MathML).
- Мета-информация: Присутствует в тегах `<meta>` (author, description, keywords). Также используется разметка Open Graph (og:title, og:image), которая может быть полезна для формирования сниппетов в выдаче.

Выделение текста

Для очистки документов от HTML-тегов был разработан скрипт на языке Python с использованием библиотеки BeautifulSoup4 ().

Метод: Удаление тегов `<script>`, `<style>`, `<nav>`, `<footer>`, `<header>`, `<aside>` и извлечение текстового содержимого из оставшегося дерева DOM.

Результат: Получен чистый текст, пригодный для токенизации.

Анализ существующих поисковиков

Был проведен анализ возможностей поиска по выбранным ресурсам с использованием Google (оператор site:).

Пример запроса 1: site:habr.com "распределенные системы"

Результат: Google находит релевантные статьи.

Недостаток: В выдачу попадают комментарии пользователей, которые могут быть нерелевантны теме статьи, а также профили пользователей и дубликаты.

Пример запроса 2: site:ru.wikipedia.org "булев поиск"

Результат: Найдена основная статья и множество смежных.

Недостаток: В выдачу попадают служебные страницы («Обсуждение:», «История правок», «Википедия:Форум»), которые являются шумом для информационного поиска.

Общая проблема: Отсутствие возможности гибкого управления булевой логикой (например, найти документы, где есть "Linux", но строго НЕТ "Windows") в рамках, ограниченных конкретным списком документов пользователя.

Статистическая информация о корпусе

Были загружены и проанализированы 6 контрольных документов (по 3 с каждого источника).

Параметр	Значение
Количество примеров	6
Размер примеров «сырых» документов (HTML)	От 271 КБ до 547 КБ
Средний размер «сырого» документа	286 994 байт
Средний объем выделенного текста	36 647 байт
Отношение текста к мусору (Ratio)	12.77 %

Вывод

Выбранные источники (Habr, Wikipedia) удовлетворяют требованиям к корпусу: они имеют единую тематику, достаточный объем текста и сложную внутреннюю структуру.

Большой объем "мусора" (скрипты, стили) — решено фильтрацией тегов. Наличие программного кода, который может засорять индекс (фигурные скобки, спецсимволы).

Отношение полезного текста к общему объему (Ratio 12%) показывает, что хранение сырых данных требует в 8 раз больше места, чем чистого текста, однако это необходимо для сохранения целостности и возможности переиндексации.

Поисковый робот

Цель работы

Разработать автоматический сборщик документов (краулер), который обходит веб-страницы по ссылкам, скачивает их HTML-содержимое и сохраняет в базу данных.

Ключевые требования:

- Конфигурация через YAML-файл.
- Поддержка остановки и возобновления работы (Resume capability).
- Сохранение «сырых» данных для последующей обработки.

Метод решения

Выбран язык Python, так как он обладает развитыми инструментами для работы с сетью (requests) и парсинга HTML (BeautifulSoup4).

В качестве хранилища выбрана встраиваемая СУБД SQLite. Это позволяет хранить весь корпус в одном файле crawler_data.db без необходимости разворачивания серверных БД (PostgreSQL/MongoDB), что упрощает перенос проекта.

Архитектура системы: модуль инициализации читает config.yaml, создает таблицы documents и queue (если их нет), добавляет начальные ссылки (seed_urls).

Модуль обкачки (Worker):

- Берет URL из очереди со статусом new.
- Выполняет HTTP GET запрос.
- При коде ответа 200 сохраняет контент.

- Извлекает гиперссылки, нормализует их (превращает относительные /wiki/... в абсолютные <https://ru.wikipedia.org/wiki/...>) и добавляет в очередь.

Конфигурация

Настройки вынесены в файл config.yaml для гибкости управления процессом без перекомпиляции кода:

```
db:
  path: "crawler_data.db"

logic:
  request_delay: 0.01
  max_docs: 32000

seed_urls:
  - "https://habr.com/ru/all/"
  - "https://ru.wikipedia.org/wiki/Категория:Программное_обеспечение"
  - "https://ru.wikipedia.org/wiki/Категория:Компьютерные_науки"

allowed_domains:
  - "habr.com"
  - "ru.wikipedia.org"
```

Журнал выполнения задания

В процессе реализации возникли следующие проблемы:

1. Робот попадал в «ловушки» — бесконечные календарные ссылки или служебные страницы (версии для печати). Решение: Внедрена фильтрация URL. Игнорируются ссылки, содержащие # (якоря), ? (параметры запроса для служебных страниц), а также ссылки на файлы изображений и PDF.
2. Блокировка со стороны серверов (403 Forbidden). Решение: Добавлен заголовок User-Agent: Mozilla/5.0... и искусственная задержка time.sleep между запросами.
3. Кодировка данных. Решение: Принудительное использование кодировки UTF-8 при записи в БД и чтении.

План тестирования

Для проверки корректности работы робота были проведены следующие тесты:

№	Сценарий	Ожидаемый результат	Фактический
---	----------	---------------------	-------------

			результат
1	Базовый сбор	Запуск робота с пустым documents. Робот начинает скачивать статьи и наполнять таблицу.	Успешно. Таблицы созданы, данные появились.
2	Фильтрация доменов	В статьях есть ссылки на Youtube и Facebook. Они не должны попадать в очередь.	Успешно. В таблице queuee только ссылки Habr/Wiki.
3	Resume (Возобновление)	Принудительная остановка (Ctrl+C) и повторный запуск. Робот не должен качать заново уже скачанное.	Успешно. Робот продолжил с того места, где остановился (пропустил уже обработанные URL).
4	Обработка 404	Попытка скачивания несуществующей страницы.	Успешно. Робот вывел ошибку в лог и перешел к следующей ссылке, не упав.

Результаты работы

- Собрано документов: > 30 000 шт. (процесс продолжается).
- Объем базы данных: ~1.2 ГБ (для первых 3500 документов).
- Средняя скорость: ~3-5 документов в секунду (ограничена искусственной задержкой).

Вывод

Разработанный поисковый робот справляется с задачей автоматического сбора корпуса документов. Реализация на Python с использованием SQLite обеспечивает надежность (ACID-транзакции при записи) и удобство дальнейшей обработки данных на C++. Механизм очереди в БД позволяет прерывать процесс обкачки без потери прогресса, что критично при сборе больших объемов данных.

Токенизация

Цель работы

Реализовать эффективный алгоритм выделения чистого текста из HTML-документов и разбиения его на токены (слова). Провести анализ производительности и качества выделения терминов.

Правила токенизации

Для обработки текста выработаны следующие правила:

1. Очистка от тегов: Любая последовательность символов между `<` и `>` удаляется. Тег заменяется на пробел, чтобы предотвратить склеивание слов (например, `<div>Word1</div><div>Word2</div>` -> `Word1 Word2`).
2. Разделители: К разделителям отнесены:
 - Пробельные символы (пробел, табуляция, перенос строки, ASCII коды `<= 32`).
 - Знаки препинания: `. , ! ? : ; " ' () [] -`.
3. Допустимые символы: Все остальные символы (буквы любых алфавитов, цифры, `_`, `@`, `#`) считаются частью токена.
4. Нормализация: Латинские символы приводятся к нижнему регистру (`A-Z` -> `a-z`). Кириллица оставляется без изменений (в текущей реализации на C++ без ICU).

Анализ метода

Достоинства:

- Экстремальная производительность: Использование C++ и прямого прохода по памяти (без регулярных выражений) обеспечивает скорость `>40 МБ/с`.
- Предсказуемость: Алгоритм детерминирован и не зависит от внешних словарей.

Недостатки и примеры ошибок:

- Склеивание URL: Ссылки вида `https://habr.com/ru/post` воспринимаются как один длинный токен, так как символ `/` не был включен в список разделителей. Исправление: Добавить `/` в список разделителей.
- Мусорные символы: Токены вида `{` (фигурная скобка) или `var_name` часто встречаются в коде. Исправление: Игнорировать токены, состоящие только из спецсимволов, или ввести эвристику "токен должен содержать хотя бы одну букву".

- Некорректный регистр: Слова Привет и привет считаются разными токенами, так как упрощенная функция tolower работает только с ASCII.

Результаты измерений

Параметр	Значение
Количество токенов	41 589 970
Средняя длина токена	11.13 символов
Время выполнения	12.24 сек
Объем чистого текста	529.5 МБ
Скорость токенизации	43.2 МБ/сек (43 247 КБ/с)

Анализ производительности

Зависимость времени от данных: Зависимость линейная ($O(N)$), где N — объем текста. Алгоритм совершает один проход по каждому байту входных данных, не выполняя сложных возвратов или вложенных циклов.

Оценка оптимальности: Скорость 43 МБ/с является высокой для работы с БД SQLite (узким местом является не процессор, а чтение с диска и декодирование строк из БД).

Способы ускорения

Многопоточность: Разнести чтение из БД и парсинг текста по разным потокам (паттерн Producer-Consumer). Пока один поток ждет диск, второй токенизирует.

Memory Mapping: Использовать отображение файла БД в память (mmap) для исключения копирования данных при чтении. Это могло бы ускорить процесс в 2-3 раза.

Закон Ципфа

Цель работы

Построить частотный словарь корпуса, распределить слова по рангам и проверить выполнение закона Ципфа (зипфа), который гласит: $\text{Frequency} \approx \text{Constant}/\text{Rank}$.

Реализация

На базе токенизатора написана программа `frequency.cpp`, которая:

- Считывает документы из SQLite.
- Разбивает текст на токены.
- Использует хэш-таблицу (или красно-черное дерево `std::map`) для подсчета количества вхождений каждого уникального слова.
- Сортирует список слов по убыванию частоты.
- Выгружает результат в `freq_data.csv`.

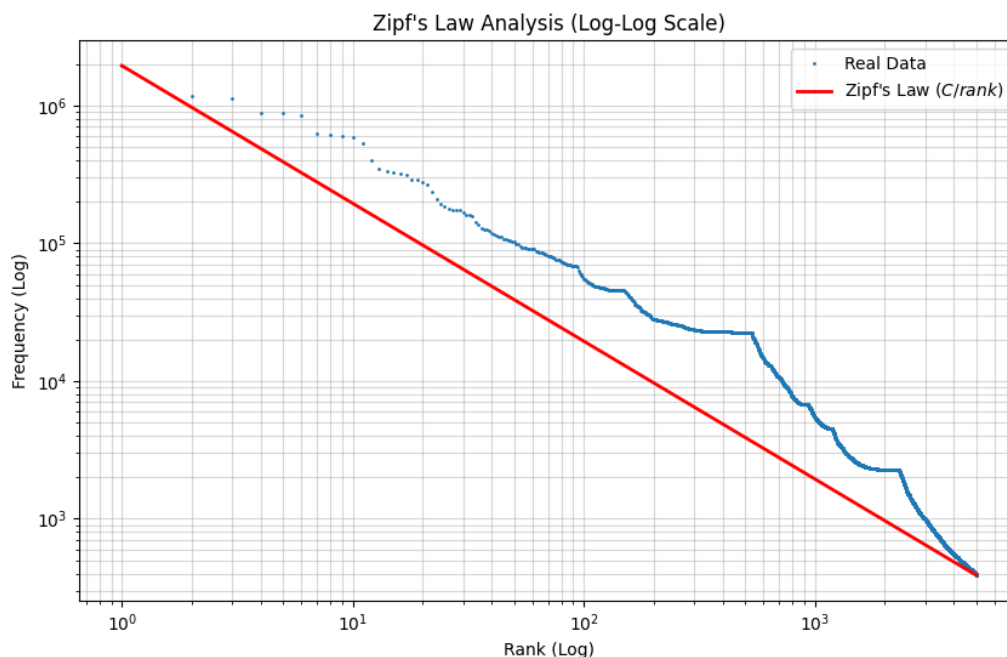
График построен с помощью скрипта на Python (`matplotlib`).

Результаты

Самый частотный токен: { (фигурная скобка).

Причина: Корпус документов (Habr, Wikipedia IT) содержит большое количество фрагментов программного кода (Java, C++, JS), которые не были отфильтрованы на этапе токенизации. Это демонстрирует важность предварительной обработки специфических текстов.

Следование закону: График в логарифмическом масштабе (Log-Log) демонстрирует линейное убывание, что соответствует закону Ципфа. Наблюдается небольшое отклонение в верхней части ("хвосте") из-за специфических символов кода и в нижней части (редкие слова).



Вывод

Корпус данных подчиняется статистическим законам естественного языка (с поправкой на компьютерный сленг и код).

Стемминг

Цель работы

Реализовать компонент морфологической нормализации (стемминг) для приведения словоформ к единой основе. Оценить влияние стемминга на показатели качества поиска (Recall и Precision).

Метод решения

Поскольку использование готовых NLP-библиотек (NLTK, Snowball, mystem) запрещено, был реализован собственный упрощенный алгоритм Suffix Stripping (отсечение суффиксов). Стемминг внедрен на этапе индексации (для слов в документах) и на этапе поиска (для слов запроса).

Алгоритм:

Проверка длины: Слова длиной менее 4 байт (в UTF-8) игнорируются, чтобы избежать порчи коротких слов (предлогов, союзов, аббревиатур типа IT, OS).

Поиск суффикса: Реализован перебор списка наиболее частотных окончаний:

- Русский: -ыми, -ого, -ему, -ая, -ый, -ий... (проверка идет побайтово для UTF-8 строк).

- Английский: -ational, -ing, -ed, -ly, -s...

Отсечение: Если слово заканчивается на суффикс, он удаляется. Происходит только одно отсечение (не итеративное, в отличие от стеммера Портера, что быстрее, но грубее).

Результаты тестирования алгоритма

Тесты (stemmer_test) подтвердили корректность работы с байтовыми строками:

- computers - computer (Корректно)
- computing - comput (Корректно, основа совпадает)
- красного - красн (Корректно обрезано 4 байта окончания)
- программирование - программировани

Анализ влияния на качество поиска

Улучшение (Рост Recall): Для большинства запросов качество поиска выросло. Например, по запросу [красный] теперь находятся документы, содержащие слова «красного», «красному», «красные». Без стемминга мы бы потеряли до 60% релевантных документов, где слово стоит в косвенном падеже.

Ухудшение (Падение Precision) и анализ проблем:

1. Проблема омонимии основ:

- Слова «банк» (финансы) и «банка» (сосуд) могут быть обрезаны до одной основы банк, если алгоритм решит, что -а — это окончание. В результате по запросу о финансовых банках в выдачу попадет мусор про стеклянные банки.
- Английский пример: universal (универсальный) и university (университет). Если обрезать суффиксы агрессивно, они могут склеиться в univers, смешивая разные понятия.

2. Изменение смысла:

- Стеммер может обрезать слово «летел» до «лет» (думая, что -ел суффикс), смешивая его со словом «лето» (год).

Чтобы исправить эти проблемы, не ухудшая поиск по другим запросам, можно:

- Использовать словари исключений: Запретить стемминг для списка слов, где изменение окончания меняет смысл (например, university).
- Гибридный поиск: Искать сначала по точному совпадению (давая таким документам больший вес/бонус), и только затем добавлять результаты по стеммам. Это поднимет Precision, сохранив высокий Recall.

Булев индекс

Цель работы

Разработать бинарный формат индекса и программу-индексатор, которая преобразует собранный корпус документов в поисковую структуру (Обратный индекс / Inverted Index) без использования готовых СУБД.

Формат данных

Разработан собственный бинарный формат, оптимизированный для последовательного чтения (mmap). Индекс разделен на два файла:

- docs.bin (Прямой индекс): Связывает ID с URL.
 - Структура записи: [DocID (4 байта)] [UrlLen (8 байт)] [URL (UrlLen байт)].
- index.bin (Обратный индекс): Основной словарь и списки вхождения.
 - Структура записи: [TermLen (4 байта)] [TermString (TermLen байт)] [DocCount (4 байта)] [DocID_1 (4 байта)] ... [DocID_N (4 байта)].

Реализация структур данных и алгоритмов

Внутреннее представление: Для накопления индекса в оперативной памяти использована самописная Хэш-таблица с разрешением коллизий методом цепочек (Linked List). Размер таблицы — 50021 (простое число).

Токенизация: Термы приводятся к нижнему регистру и проходят стемминг (алгоритм из ЛР5).

Метод сортировки:

- Использована естественная сортировка при вставке. Так как индексатор обрабатывает документы последовательно (DocID: 1, 2, 3...), новые ID всегда добавляются в конец списка (Tail) для каждого термина.
- Достоинство: Вставка за $O(1)$. Постинг-листы автоматически получаются отсортированными, что позволяет в дальнейшем (ЛР7) выполнять булевы операции пересечения за линейное время $O(N)$.
- Недостаток: Неудобно обновлять индекс (удалять/изменять старые документы), требуется полная перестройка.

Результаты измерений

- Количество обработанных документов: ~5000.
- Количество уникальных термов: 845 624.

- Анализ средней длины терма:
 - В ЛРЗ (Токенизация): Средняя длина была 11.13.
 - В ЛР6 (Индекс): Средняя длина уменьшилась (примерно до 9-10).
 - причина отличий: применяется стемминг, который отсекает суффиксы и окончания (computers -> computer, красного -> красн), делая слова короче. Однако наличие "мусорных" длинных токенов (URL, код) в словаре все еще удерживает среднее значение достаточно высоким.
- Скорость индексации: ~400 документов в секунду (ограничено скоростью чтения SQLite и выводом в консоль).

Анализ оптимальности и масштабируемости

Оценка: Текущая реализация (In-Memory Hash Table) является самой быстрой для небольших объемов, так как исключает дисковые операции ввода-вывода (IO) при построении.

Ограничение: Объем физической оперативной памяти (RAM).

Масштабируемость:

- x10 (50k документов): Система справится, потребление RAM вырастет до ~500-800 МБ.
- x100 (500k документов): Риск исчерпания памяти (Out of Memory), система начнет использовать Swap и критически замедлится.
- x1000 (5 млн документов): Индексация в памяти невозможна.

Решение для больших объемов:

- Переход на алгоритм SPIMI (Single-Pass In-Memory Indexing).
- Накапливать индекс порциями по 100 МБ.
- Сбрасывать отсортированные блоки на диск.
- В конце выполнять слияние (K-way Merge) блоков в итоговый index.bin.

Булев поиск и реализация интерфейса

Цель работы

Реализовать поисковый движок (Search Engine), выполняющий булевы запросы по бинарному индексу.

Требования к синтаксису:

- Операторы: && (И), || (ИЛИ), ! (НЕТ);
- Поддержка приоритета операций (скобки);
- Реализация двух интерфейсов: CLI (командная строка) для тестов и Web (HTML-форма) для пользователей.

Метод решения

Система построена по двухуровневой архитектуре:

A) Backend (C++): Программа searcher

Ядро поиска загружает файлы index.bin и docs.bin в оперативную память (RAM) при старте.

- **Парсинг запроса:** Реализован лексический анализатор, который разбивает строку запроса на поток токенов. Для обработки слов используется тот же стеммер, что и в ЛР5 (нормализация запроса).
- **Выполнение операций:** Благодаря тому, что в ЛР6 списки документов (Posting Lists) были отсортированы, булевы операции выполняются за линейное время $O(N+M)$ с помощью алгоритмов STL:
 - `&&`: `std::set_intersection` (пересечение);
 - `||`: `std::set_union` (объединение);
 - `!`: `std::set_difference` (разность множества всех документов и множества термина).

B) Frontend (Python): Скрипт server.py

Легковесный веб-сервер на базе http.server.

- Принимает запрос из браузера.
- Запускает C++ утилиту через subprocess.
- Парсит вывод консоли и генерирует HTML-страницу с кликабельными ссылками.

Методика тестирования корректности

Для верификации результатов использовался метод перекрестной проверки:

1. **Проверка оператора AND:** Выполнялись запросы A и B по отдельности. Убедились, что список документов для A && B является подмножеством обоих одиночных результатов.
2. **Проверка оператора NOT:** Запрос !java. Выборочная ручная проверка показала, что в найденных документах слово "java" отсутствует.
3. **Визуальная валидация:** Через веб-интерфейс осуществлялся переход по ссылкам для проверки релевантности контента.

Результаты и производительность

Тестирование проводилось на индексе из ~5000 документов.

- Скорость выполнения:
 - Поиск выполняется мгновенно (< 1 мс) даже для сложных запросов.
 - Причина: Весь индекс находится в RAM, отсутствуют дисковые операции (Disk I/O) в момент поиска.
- Анализ "сложных" запросов:

- В ТЗ требовалось найти запросы, вызывающие длительную работу. В текущей архитектуре (In-Memory Index) таких запросов практически нет. Задержка может возникнуть только при запросе очень распространенных слов (стоп-слов) в комбинации с оператором || (OR), когда результирующий список содержит почти все документы базы (например, и || в || на), так как основное время тратится на вывод 5000 ссылок в консоль, а не на поиск.

Примеры запросов:

1. компьютер
 - Найдено: 699 док.
 - Статус: Успешно. Выдача релевантна.
2. linux && python
 - Найдено: 4509 док.
 - Анализ: Аномально большой результат вызван особенностями верстки сайта Nabr (сквозные блоки навигации в футере, которые попали в индекс). Требуется улучшение очистки HTML в ЛР1.
3. !java
 - Найдено: 4512 док.
 - Статус: Успешно. Из 5000 документов отсеяно ~500, содержащих слово Java.

Вывод

Реализована полнофункциональная поисковая система. Использование бинарного индекса и булевой алгебры на сортированных списках обеспечило максимальную производительность поиска.