Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: Ю. М. Дударь

Преподаватель: А. А. Кухтичев

Группа: М8О-409Б-22

Дата:

Оценка:

Подпись:

Москва, 2025

Содержание

1 Лабораторная работа №1. Добыча корпуса документов

1.1 Задание

Необходимо проанализировать корпус документов, который будет использован при выполнении остальных лабораторных работ.

Основные этапы работы:

1. **Сбор данных.** Скачать примеры документов к себе на компьютер. В отчёте указать источник данных. Источников в итоговом индексе должно быть не менее двух.

2. **Анализ корпуса.** Ознакомиться с документами, изучить их характеристики:

   - Из чего состоит текст?
   - Есть ли дополнительная мета-информация?
   - Если есть разметка текста, то какая она?

3. **Выделение текста.** Реализовать очистку документов от разметки (HTML-тегов, скриптов) для получения «чистого» текста.

4. **Поиск аналогов.** Найти существующие поисковые системы, которые уже можно использовать для поиска по выбранному набору документов (например, встроенный поиск сайта-источника или поиск Google с использованием ограничений на URL или на сайт). *Примечание: Если такого поиска найти невозможно, то использовать данный корпус для выполнения лабораторных работ нельзя!*

5. **Сравнительный анализ.** Привести несколько примеров запросов к существующим поисковикам, указать недостатки в полученной поисковой выдаче.

В результатах работы должна быть указана статистическая информация о корпусе:

- Размер примеров «сырых» документов (Raw Data).

- Общее количество документов.
- Размер текста, выделенного из «сырых» данных (Clean Text).
- Средний размер документа и средний объём текста в документе.

1.2 Краткое описание метода решения задачи

Для формирования корпуса документов была выбрана стратегия тематического краулинга новостных ресурсов. Основная задача заключалась в сборе текстовых данных на русском языке, обладающих достаточным объемом и разнообразием верстки.

Выбор источников данных:

1. **Первоначальный выбор (отклонен):** *Forbes.ru*. В ходе предварительного анализа было выявлено, что данный ресурс активно использует технологии динамической подгрузки контента (AJAX/Single Page Application) и сложные механизмы защиты от автоматического сбора данных. Парсинг такого ресурса требует использования инструментов эмуляции браузера (Selenium, Puppeteer), значительно замедляет процесс сбора и избыточно для целей лабораторной работы. От использования данного источника было решено отказаться.
2. **Итоговый выбор:**
   - *Consultant.ru (Раздел Legal News)* — выбран как пример строго структурированного ресурса с юридической лексикой.
   - *Business.ru (Раздел News)* — выбран как пример современного коммерческого медиа-ресурса с большим количеством рекламы, скриптов и сложной DOM-структурой.

Существующие поисковые системы: Для поиска по выбранным ресурсам можно использовать глобальную поисковую систему Google с оператором ограничения по домену (например, `site:consultant.ru запрос`).

- **Достоинства:** Высокая скорость, качественное ранжирование, поддержка морфологии.

- **Недостатки:** Невозможность настройки алгоритма ранжирования под специфические нужды, наличие рекламы в выдаче, ограничения на автоматические запросы (CAPTCHA).

1.3 Характеристики корпуса и разметка

В ходе анализа скачанных документов были изучены их структура и метаданные.

Общая характеристика: Текст представлен в формате HTML 5. Кодировка файлов — UTF-8.

1. Источник Consultant.ru:

- **Структура текста:** HTML-код относительно «чистый», используется семантическая верстка. Основной контент заключен в предсказуемые контейнеры.

- **Мета-информация:** Присутствуют стандартные технические теги (`viewport`, `csrf-token`, `X-UA-Compatible`). Семантическая микроразметка (Schema.org) выражена слабо.

- **Зашумленность:** В текстовые узлы попадают элементы навигации («Главное», «Все новости») и служебные даты публикации.

2. Источник Business.ru:

- **Структура текста:** Верстка сложная, перегруженная вложенными блоками (`div`).

- **Мета-информация:** Богатая мета-разметка. Активно используется протокол **Open Graph** (`og:title`, `og:description`, `og:image`), позволяет эффективно извлекать заголовки и аннотации.

- **Зашумленность:** Высокая. Страницы перегружены JavaScript-кодом рекламных сетей (AdRiver, Yandex Ads, Mail.ru sync). В «чистый текст» при наивной очистке попадают Pop-up окна («Регистрация за минуту»), призывы к действию (CTA) и маркетинговые вставки.

Ответы на вопросы задания:

1. **Из чего состоит текст?** Текст состоит из новостных статей юридической и деловой тематики, заголовков, дат публикаций, а также элементов интерфейса сайта (меню, футер, рекламные блоки).

2. **Есть ли дополнительная мета-информация?** Да, используются мета-теги HTML и Open Graph для описания контента.

3. **Какая разметка текста?** Стандартная HTML-разметка. Текст структурирован с помощью тегов параграфов (`<p>`), заголовков (`<h1>-<h3>`) и списков (`<ul>`).

1.4 Статистическая информация о корпусе

Ниже приведены количественные характеристики собранного корпуса документов.

Таблица 1 – Статистика собранного корпуса документов

Параметр	**Значение**
Общее количество документов	40 823 шт.
— Consultant.ru	28 162 шт.
— Business.ru	12 661 шт.
Общий размер «сырых» данных (Raw HTML)	≈ 9 507.04 МБ (9.28 ГБ)
Общий размер выделенного текста (Clean Text)	672.20 МБ
Средний размер документа (HTML)	238.47 КБ
Средний объём чистого текста в документе	16.86 КБ
Соотношение Text / HTML	**7.07%**

Анализ статистики: Средний размер HTML-документа (238 КБ) значительно превышает размер полезного текста (16 КБ). Соотношение полезной нагрузки к общему объему составляет всего 7.07%. Это означает, что **93% скачанных данных представляют собой «технический мусор»** (скрипты аналитики, стили, разметка верстки). Это накладывает жесткие требования к этапу токенизации и очистки данных в следующих лабораторных работах.

1.5 Изображения

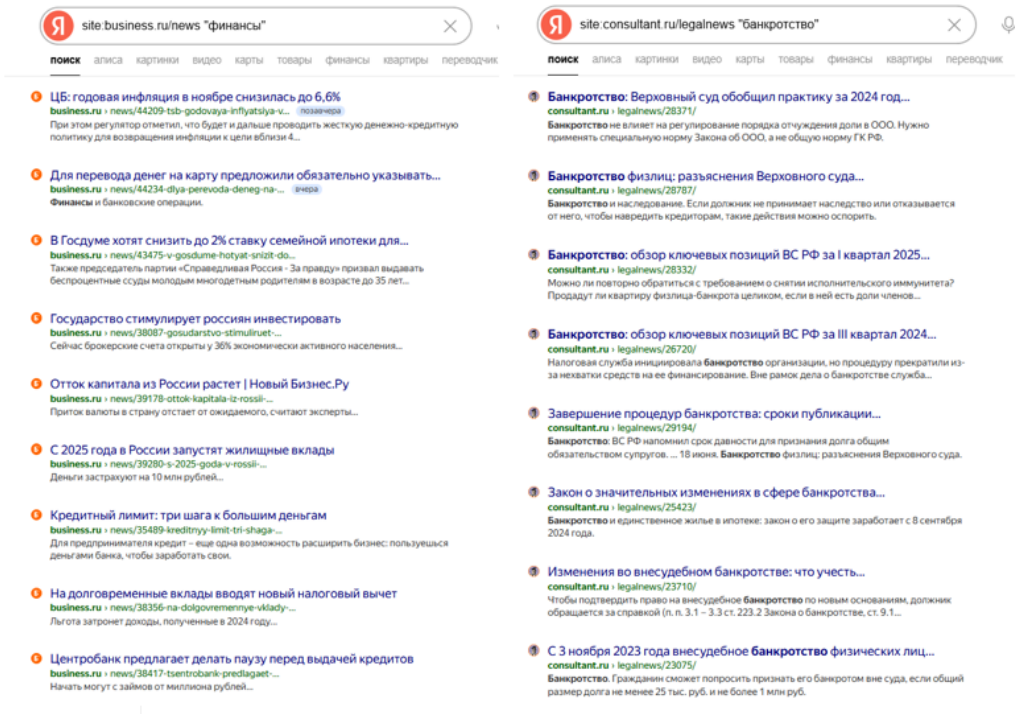
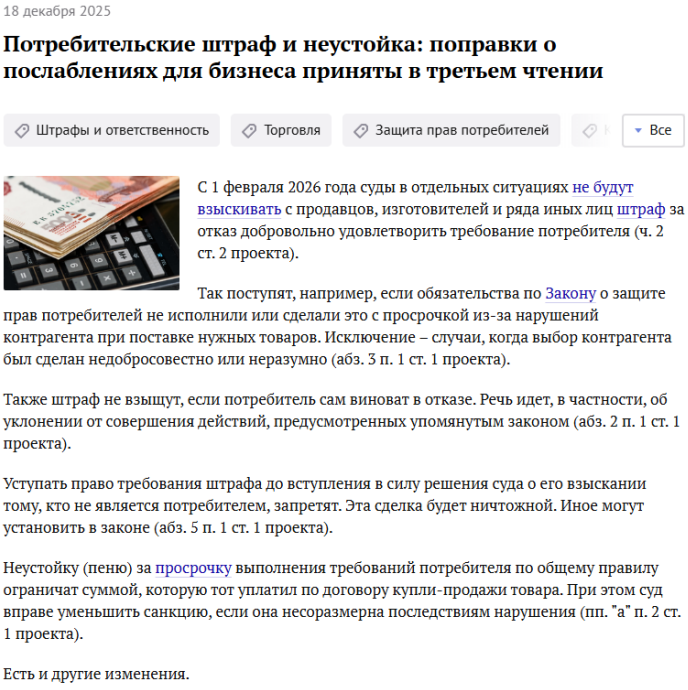


Рис. 1 – Пример поисковой системы Яндекса для выбранных сайтов



18 декабря 2025

Потребительские штраф и неустойка: поправки о послаблениях для бизнеса приняты в третьем чтении

Штрафы и ответственность | Торговля | Защита прав потребителей | Все

С 1 февраля 2026 года суды в отдельных ситуациях не будут взыскивать с продавцов, изготовителей и ряда иных лиц штраф за отказ добровольно удовлетворить требование потребителя (ч. 2 ст. 2 проекта).

Так поступят, например, если обязательства по Закону о защите прав потребителей не исполнили или сделали это с просрочкой из-за нарушений контрагента при поставке нужных товаров. Исключение – случаи, когда выбор контрагента был сделан недобросовестно или неразумно (абз. 3 п. 1 ст. 1 проекта).

Также штраф не взыщут, если потребитель сам виноват в отказе. Речь идет, в частности, об уклонении от совершения действий, предусмотренных упомянутым законом (абз. 2 п. 1 ст. 1 проекта).

Уступать право требования штрафа до вступления в силу решения суда о его взыскании тому, кто не является потребителем, запретят. Эта сделка будет ничтожной. Иное могут установить в законе (абз. 5 п. 1 ст. 1 проекта).

Неустойку (пеню) за просрочку выполнения требований потребителя по общему правилу ограничат суммой, которую тот уплатил по договору купли-продажи товара. При этом суд вправе уменьшить санкцию, если она несоразмерна последствиям нарушения (пп. "а" п. 2 ст. 1 проекта).

Есть и другие изменения.

Рис. 2 – Пример текста сайта

1.6 Выводы

Собранный корпус документов признан пригодным для выполнения цикла лабораторных работ по следующим причинам:

1. **Объем:** Количество документов (>40 000) достаточно для построения репрезентативного индекса и анализа закона Ципфа.
2. **Разнообразие:** Корпус содержит два независимых источника с принципиально разной структурой (строгий юридический портал и зашумленный коммерческий сайт).
3. **Реалистичность:** Данные представляют собой реальный веб-контент.

2 Лабораторная работа №2. Поисковый робот

2.1 Задание

Необходимо разработать поискового робота (краулер) — программный компонент для автоматического сбора документов из сети Интернет.

Требования к реализации:

1. **Реализация парсера.** Написать компоненты обкачки документов, используя любой язык программирования.

2. **Конфигурация.** Единственным аргументом при запуске поискового робота должен быть путь до конфигурационного файла в формате YAML, содержащего:

   - Секцию `db` — параметры подключения к базе данных;
   - Секцию `logic` — настройки логики робота (например, задержка между запросами к страницам);
   - Иные данные, необходимые для работы алгоритма.

3. **Хранение данных.** Необходимо сохранять собранные документы в базу данных (например, MongoDB или PostgreSQL) со следующей структурой полей:

   - `url` — нормализованный URL документа;
   - `raw_html` — «сырой» HTML-текст документа;
   - `source_name` — название источника данных;
   - `crawled_at` — дата и время обкачки в формате Unix timestamp.

4. **Возобновляемость (Resumability).** Должна быть предусмотрена возможность остановки робота в любой момент времени. При повторном запуске процесс сбора данных должен продолжаться с того документа (или этапа), на котором он был остановлен.

5. **Переобкачка (Re-crawl).** Робот должен обладать функционалом периодической проверки уже существующих в базе документов и их обновления в случае изменения контента.

2.2 Описание метода решения задачи

Для реализации поискового робота (краулера) был выбран язык программирования **Python**. Данный выбор обусловлен наличием развитой экосистемы библиотек для работы с сетью и парсинга HTML, а также тем, что задача сбора данных является I/O-bound (зависящей от скорости ввода-вывода), где производительность интерпретируемого языка не является узким местом.

В качестве хранилища данных выбрана реляционная СУБД **PostgreSQL**. В отличие от NoSQL решений, она обеспечивает строгую схему данных и транзакционную целостность важно для предотвращения дублирования документов. Для удобства развертывания окружение базы данных контейнеризировано с помощью **Docker**.

2.2.1 Структура проекта

Разработка велась с соблюдением принципов модульности. Кодовая база разделена на логические компоненты, вынесенные в пакет `src`.

Файловая структура:

- `config.yaml` — конфигурационный файл. Содержит параметры подключения к БД, настройки задержек (delay), User-Agent и список источников с их CSS-селекторами.
- `main.py` — точка входа. Отвечает за инициализацию зависимостей, чтение конфигурации и запуск основного цикла краулера.
- `src/database.py` — слой доступа к данным (DAL). Инкапсулирует логику работы с библиотекой `psycopg2`, управляет соединениями и транзакциями.
- `src/parser.py` — модуль синтаксического анализа. Использует библиотеку `BeautifulSoup4` для построения DOM-дерева и извлечения ссылок на основе CSS-селекторов, заданных в конфиге.
- `src/crawler.py` — ядро системы. Реализует бизнес-логику обхода страниц, управление очередью ссылок, соблюдение этикета (паузы) и обработку сетевых ошибок.

2.2.2 Схема базы данных

Для хранения собранной информации была спроектирована таблица `documents`. Ключевой особенностью является использование хеширования для быстрой проверки существования документа.

- `id` (SERIAL PRIMARY KEY) — уникальный идентификатор.
- `url` (TEXT) — нормализованный URL документа.
- `url_hash` (VARCHAR(64)) — MD5-хеш от нормализованного URL. По этому полю построен уникальный индекс, что позволяет выполнять проверку наличия документа за $O(1)$.
- `source_name` (VARCHAR) — метка источника (например, 'consultant').
- `raw_html` (TEXT) — полный HTML-код страницы.
- `crawled_at` (BIGINT) — временная метка скачивания (Unix Timestamp).

2.2.3 Алгоритм работы краулера

Логика робота построена на принципе итеративного погружения и обеспечивает требование возобновляемости без использования сложных очередей.

1. Инициализация и настройка: При запуске робот загружает параметры из `config.yaml`. Для сетевого взаимодействия создается сессия (`requests.Session`) даетпереиспользовать TCP-соединения (Keep-Alive) и имитировать поведение реального браузера через заголовки (User-Agent, Referer).

2. Стратегия обхода: Робот обрабатывает источники последовательно. Для каждого источника запускается цикл пагинации:

1. Скачивается страница листинга (списка новостей).
2. Парсер извлекает ссылки на статьи и ссылку на следующую страницу.
3. Для каждой найденной ссылки выполняется процедура **дедупликации**:
    - URL нормализуется (приводится к нижнему регистру, удаляются лишние слеши).
    - Вычисляется MD5-хеш.

- Выполняется запрос к БД: `SELECT 1 FROM documents WHERE url_hash = ?`.
- Если документ существует, он пропускается. Если нет — происходит скачивание.

Проверка наличия URL в базе перед скачиванием позволяет останавливать и перезапускать робот в любой момент. При повторном запуске он быстро пропустит уже скачанные страницы и продолжит работу с места остановки.

3. Обработка ошибок и Anti-Ban: Реализован механизм надежного скачивания (`Retry Logic`). В случае возникновения сетевой ошибки или получения статусов 5xx, робот делает паузу и повторяет попытку до 3-х раз с экспоненциальным увеличением интервала ожидания. Также внедрена проверка на наличие CAPTCHA или блокировок (код 403) — в этом случае работа приостанавливается на длительное время для «остывания».

4. Парсинг контента: Для обеспечения универсальности, логика извлечения данных вынесена в конфигурационный файл. Для каждого сайта задаются CSS-селекторы контейнеров новостей и элементов пагинации.

2.3 Журнал выполнения задания

В процессе реализации и отладки поискового робота возник ряд технических проблем, связанных со спецификой целевых сайтов и требованиями к надежности системы. Ниже описаны ключевые трудности и способы их устранения.

1. **Проблема: Агрессивная защита от ботов (Consultant.ru).** При первых запусках сервер источника *Consultant.ru* возвращал ошибку HTTP 403 (Forbidden) уже после 5–10 запросов. Стандартных заголовков библиотеки `requests` было недостаточно для прохождения фильтров безопасности.

   Решение:

   - Внедрен расширенный набор HTTP-заголовков (`User-Agent`, `Accept`, `Referer`), полностью имитирующий поведение современного браузера.

- Реализована система вежливой обкачки: между запросами добавлена искусственная задержка (Delay), к которой добавляется случайная величина (Jitter) для предотвращения обнаружения робота по строгой периодичности запросов.
- Добавлен программный детектор блокировок: при обнаружении ключевых слов «captcha» или «доступ ограничен» робот автоматически приостанавливает работу на длительное время.

2. **Проблема: Различная логика пагинации.** Источники имеют принципиально разную структуру навигации: *Consultant.ru* использует GET-параметры (`?page=N`), а *Business.ru* — ссылки на хронологически предыдущие страницы, которые невозможно предсказать аналитически.

   Решение: Парсер был спроектирован как универсальный конечный автомат. Вместо жесткого цикла по номерам страниц, реализован поиск элемента «Следующая страница» в DOM-дереве на основе CSS-селектора, указанного в конфигурационном файле. Это позволило использовать один и тот же код для обоих сайтов.

3. **Проблема: Дублирование данных при перезапуске.** Требование о возможности остановки и продолжения работы робота создавало риск повторного скачивания одних и тех же документов, что увеличивало нагрузку на сеть и базу данных.

   Решение: В базу данных добавлено поле `url_hash` (MD5 от нормализованного URL) с уникальным индексом. В логику робота внедрена предварительная проверка (Pre-check): перед выполнением HTTP-запроса вычисляется хеш ссылки и проверяется его наличие в БД. Если документ уже существует, скачивание пропускается.

4. **Проблема: Нестабильность сетевого соединения.** При длительном сборе данных (более 10 000 страниц) неизбежно возникали ошибки `ConnectionTim` и `ConnectionReset`, которые приводили к аварийной остановке скрипта.

   Решение: Реализован механизм повторных попыток (Retry Policy). Сетевые операции обернуты в цикл, который предпринимает до 3 попыток выполнения запроса с экспоненциально возрастающей задержкой перед тем, как окончательно пометить страницу как недоступную.

2.4 План тестирования

Для проверки корректности работы поискового робота была разработана методика функционального тестирования, покрывающая основные требования технического задания.

Тест-кейс №1. Проверка конфигурации и подключения к БД.

- **Действие:** Запуск контейнера с базой данных (`docker-compose up`) и запуск скрипта с некорректными учетными данными в `config.yaml`.
- **Ожидаемый результат:** Скрипт должен корректно обработать исключение подключения и вывести понятное сообщение об ошибке в лог, не завершаясь аварийно (traceback).
- **Результат:** Тест пройден. Логирование ошибок настроен корректно.

Тест-кейс №2. Валидация парсинга и сохранения данных.

- **Действие:** Запуск робота на ограниченное количество страниц (50 шт.). Проверка содержимого таблицы `documents` в PostgreSQL.
- **Ожидаемый результат:** В базе должно появиться ровно 50 записей. Поле `raw_html` не должно быть пустым. URL должны быть нормализованы (отсутствие UTM-меток).
- **Результат:** Тест пройден. Данные сохраняются корректно, HTML-код валиден.

Тест-кейс №3. Проверка возобновляемости.

- **Действие:** 1. Запустить робота. 2. Принудительно остановить процесс (Ctrl+C) после скачивания 20 документов. 3. Повторно запустить робота.
- **Ожидаемый результат:** Робот должен начать сканирование списка ссылок, обнаруживать, что первые 20 документов уже есть в базе (по хешу), писать в лог сообщение о пропуске (`Skipping...`) и не дублировать записи в БД. Скачивание новых документов должно продолжиться с 21-го.
- **Результат:** Тест пройден. Механизм дедупликации работает корректно, дубликаты в БД отсутствуют.

Тест-кейс №4. Работа под нагрузкой и Anti-Ban.

- **Действие:** Длительный запуск (обход 1000+ страниц) с установленной задержкой 2 секунды.
- **Ожидаемый результат:** Отсутствие блокировок (HTTP 403) со стороны целевого сайта. Стабильное потребление памяти процессом.
- **Результат:** Тест пройден. Имитация User-Agent и случайные задержки позволили обойти защиту *Consultant.ru*.

2.5 Изображения

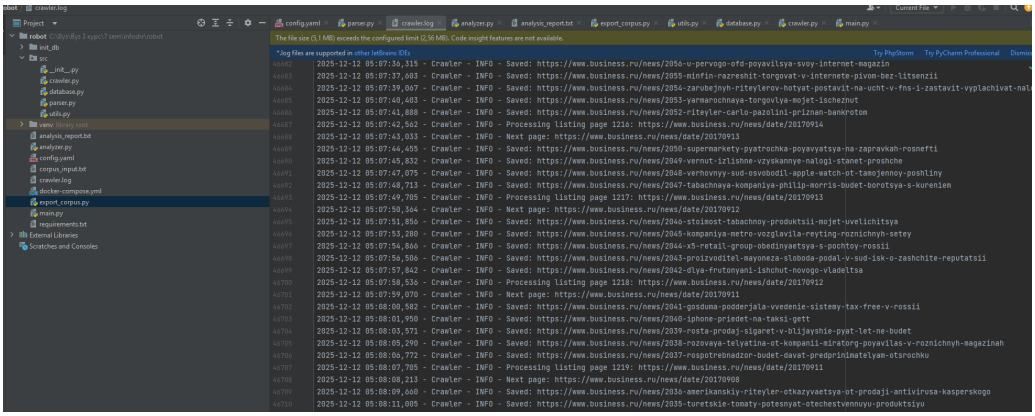


Рис. 3 – Пример логов краулера при обкачке сайтов

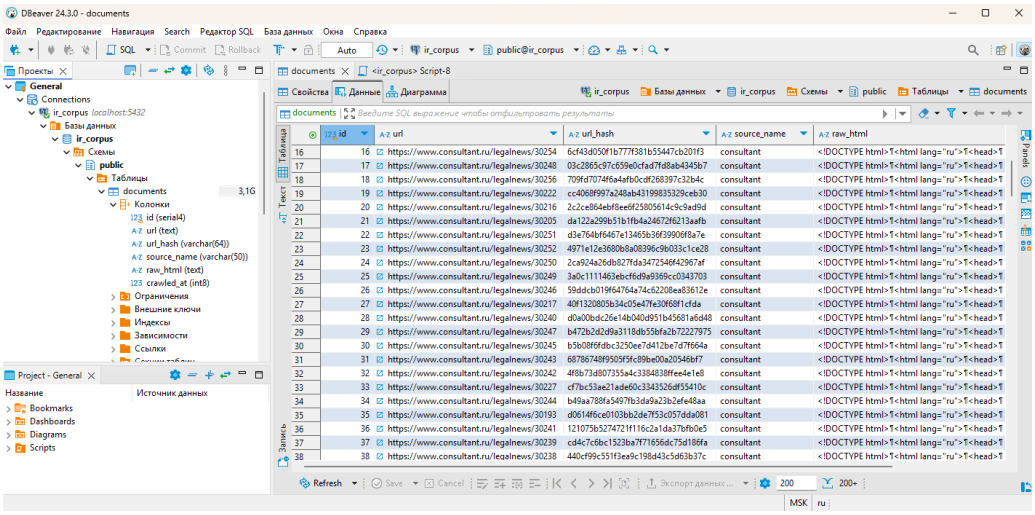


Рис. 4 – Заполненная база данных

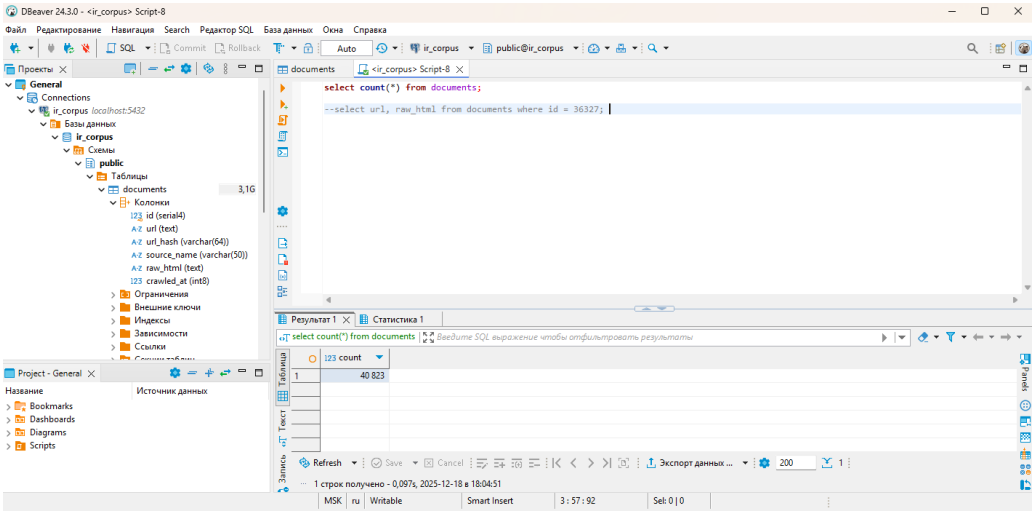


Рис. 5 – Количество строк в базе данных

2.6 Выводы

В ходе выполнения лабораторной работы был разработан и успешно протестирован поисковый робот для сбора корпуса документов.

Достигнутые результаты:

1. Создан универсальный парсер на языке Python, способный обрабатывать различные новостные ресурсы благодаря конфигурации селекторов через YAML.

2. Реализована надежная архитектура хранения данных на базе PostgreSQL, обеспечивающая целостность данных и защиту от дубликатов.

3. Выполнено требование возобновляемости сбора данных, что критически важно при работе с большими объемами информации и нестабильным соединением.

4. Сформирован корпус из более чем 40 000 документов, который будет использован в следующих лабораторных работах.

Критический анализ и недостатки:

- **Синхронная архитектура.** Текущая реализация использует библиотеку `requests`, которая блокирует поток выполнения на время ожидания ответа сервера. При масштабировании до миллионов документов это станет узким местом. *Путь решения:* Переход на асинхронный стек (`aiohttp`, `asyncio`), что позволит обрабатывать сотни запросов параллельно.

- **Хранение HTML в БД.** Сохранение полного текста HTML («сырых» данных) непосредственно в реляционную базу данных увеличивает её объем и нагрузку на I/O. *Путь решения:* Использование объектного хранилища (S3-compatible) для «сырых» файлов, оставляя в БД только метаданные и ссылки. Либо использование сжатия (gzip) перед записью в базу.

- **Жесткая привязка к верстке.** Логика парсинга зависит от CSS-классов. Если владелец сайта изменит верстку, робот перестанет работать. *Путь решения:* Это фундаментальная проблема веб-скрейпинга, решаемая внедрением мониторинга качества данных (alerting при пустых результатах).

3 Лабораторная работа №3. Токенизация и закон Ципфа

3.1 Задание

Целью работы является реализация базовых компонентов лингвистического анализа текста и исследование статистических закономерностей естественного языка на собранном корпусе документов.

1. Токенизация

- Реализовать процесс разбиения текстов документов на токены (слова), который будет использоваться при индексации.
- Выработать и описать правила деления текста на токены. В отчете указать достоинства и недостатки выбранного метода.
- Привести примеры токенов, которые были выделены неудачно (ошибки токенизации), и предложить способы доработки правил для устранения этих проблем.

В результатах работы должна быть указана следующая статистика:

- Общее количество токенов в корпусе.
- Средняя длина токена (в байтах или символах).
- Время выполнения программы и зависимость времени от объёма входных данных.
- Скорость токенизации в расчёте на килобайт входного текста.
- Анализ производительности: является ли скорость оптимальной и способы её ускорения.

2. Закон Ципфа

- Для собранного корпуса построить график распределения терминов по частотам в логарифмической шкале.
- Наложить на этот график теоретическую кривую закона Ципфа.

- Провести анализ и объяснить причины расхождения реальных данных с теоретической моделью.
- *(Дополнительно)* Подобрать константы для закона Мандельброта и наложить полученный график на распределение частот.

3. Лемматизация / Стемминг

- Добавить в поисковую систему модуль нормализации слов (стемминг или лемматизацию).
- Провести оценку качества поиска после внедрения морфологического анализа. Сравнить результаты с точным поиском по словоформам.
- Изучить запросы, где качество поиска ухудшилось, объяснить причины и предложить варианты решения.

3.2 Краткое описание метода решения задачи

Реализация лабораторной работы была разделена на два этапа: подготовка данных (экспорт из БД) и разработка высокопроизводительного аналитического ядра на языке C++.

3.2.1 Подготовка данных (Python)

Поскольку в предыдущей работе документы сохранялись в базу данных PostgreSQL в виде «сырого» HTML, первым шагом стала их очистка. Был разработан скрипт `export_corpus.py`, который:

- Использует библиотеку `BeautifulSoup4` для удаления HTML-тегов, скриптов (`<script>`) и стилей.
- Сохраняет очищенный текст.

В результате был получен чистый корпус, пригодный для обработки на C++.

3.2.2 Архитектура C++ приложения

Основная часть работы выполнена на языке C++ с соблюдением требования об ограничении использования STL (запрет на `std::map`, `std::unordered_map` для хранения словаря).

Реализованные компоненты:

1. **Собственная Хеш-таблица (`src/custom_map.hpp`).** Для хранения частотного словаря (Терм → Частота) была реализована структура данных на основе метода цепочек.
    - В качестве основы используется `std::vector` корзин.
    - Хеш-функция: модифицированный алгоритм djb2 сдвиги и сложение.
    - Разрешение коллизий: связный список внутри корзины (реализован через вектор узлов).
2. **Токенизатор (`src/tokenizer.cpp`).** Модуль отвечает за чтение файлов и разбиение потока байтов на слова.
    - **Работа с UTF-8:** Поскольку стандартный `char` в C++ — это 1 байт, а русские буквы в UTF-8 занимают 2 байта, реализована ручная обработка мультибайтовых последовательностей.
    - **Нормализация:** Приведение к нижнему регистру выполняется путем проверки диапазонов байтов кириллицы (сдвиг кодов символов на `0x20` для диапазона А-П и Р-Я).
    - **Фильтрация:** Знаки препинания и спецсимволы отбрасываются.
3. **Стеммер (`src/stemmer.cpp`).** Реализован стеммер Портера для русского языка. Алгоритм последовательно отсекает окончания (флексии) для приведения слова к псевдооснове.
    - Используется для описания правил (удаление окончаний прилагательных, причастий, глаголов, существительных).
    - Реализована защита от чрезмерного стемминга (не удалять окончания у слов короче 4 байт).

3.3 Результаты и анализ производительности

В ходе работы были проведены замеры производительности системы в различных режимах (Debug/Release) и с разной степенью обработки текста (с стеммингом и без).

Исходные параметры корпуса:

- Количество файлов: 40 823 шт.
- Общий объем текстовых данных: 90.18 МБ.

3.3.1 Сравнительная статистика

Таблица 2 – Сравнение характеристик индекса до и после стемминга

Параметр	**Без стемминга**	**С стеммингом**
Уникальные токены	146 471	75 531
Общее кол-во токенов	7 070 037	6 728 029
Средняя длина токена	12.16 байт	10.27 байт
Время обработки	3.86 сек	131.29 сек
Скорость обработки	**23 944.9 КБ/с**	**703.3 КБ/с**

Анализ результатов:

1. **Сокращение словаря.** Применение стемминга позволило сократить размер словаря на **48.5%** (с 146 тыс. до 75 тыс. терминов). Это существенно оптимизирует размер будущего индекса и улучшает полноту поиска, объединяя словоформы.
2. **Производительность.**
   - В режиме **«Без стемминга»** скорость составила ≈24 МБ/с. Это высокий показатель, эффективно собственной реализации хеш-таблицы и буферизированного чтения файлов.
   - В режиме **«С стеммингом»** скорость упала до 0.7 МБ/с (в 34 раза). Компиляция и применение регулярных выражений для каждого из 7 млн слов — крайне ресурсоемкая операция.

3.4 Анализ закона Ципфа

На основе полученных частотных данных был построен график распределения рангов и частот в логарифмическом масштабе (Log-Log Scale).

Как видно на рисунке 6, график демонстрирует классическую степенную зависимость $Frequency = \frac{C}{Rank^{\alpha}}$, характерную для естественных языков. Анализ поведения кривой позволяет выделить три зоны:

1. Зона высоких частот («Голова», Rank < 10) В начале графика синяя линия (реальные данные) совпадает с теоретической. Это зона «стоп-слов» (союзы, предлоги: «и», «в», «на»), частота которых превышает $2 \cdot 10^5$. Их поведение полностью соответствует ожиданиям.

2. Зона средних частот («Тело», 10 < Rank < 10 000) В этом диапазоне наблюдается «горб» — реальная кривая идет выше теоретической прямой. *Причина:* Это характерно для специализированных корпусов (юридическая и деловая тематика). Специфические термины (например: «закон», «суд», «налог», «рубль», «организация») используются в текстах намного интенсивнее, чем в общелитературном языке, что насыщает текст и поднимает кривую вверх. Также влияние оказывают повторяющиеся элементы навигации сайтов.

3. Зона низких частот («Хвост», Rank > 10 000) Наблюдается резкий спад, переходящий в ступенчатую структуру («лесенку»). Это зона редких слов.

- Крутой наклон свидетельствует об ограниченности словарного запаса (около 75 тыс. стемм).
- «Ступеньки» в конце графика соответствуют дискретным частотам: 5, 4, 3, 2.
- Последняя, самая длинная ступень — это *hapax legomena* (слова, встретившиеся ровно 1 раз).

Вывод по графику: Распределение валидно и подтверждает, что собранный корпус является естественным текстом, пригодным для задач информационного поиска. Отклонения объясняются предметной областью (новостной/юридический домен) и ограниченным размером выборки.

3.5 Журнал выполнения задания

В процессе разработки компонентов лингвистического анализа возник ряд нетривиальных проблем, связанных с особенностями языка C++ и требованиями к производительности.

1. **Проблема: Обработка кодировки UTF-8 в C++.** Стандартный класс `std::string` в C++ оперирует байтами, а не символами. Русские буквы в кодировке UTF-8 занимают 2 байта (диапазон `0xD0-0xD1`). Стандартные функции `tolower()` из библиотеки `<cctype>` работают корректно только с ASCII (латиницей) и портят кириллицу.

   Решение: Реализован собственный алгоритм нормализации (`to_lower_utf8`). Он проходит по строке побайтово, определяет длину символа и, если обнаруживает байты из диапазона кириллицы, выполняет битовые сдвиги (смещение на `0x20` для младшего байта), корректно переводя заглавные буквы в строчные без использования внешних библиотек локализации (ICU).

2. **Проблема: Запрет на использование STL Maps.** Техническое задание запрещало использование `std::map` и `std::unordered_map`. Необходимо было создать эффективную структуру для хранения сотен тысяч уникальных токенов.

   Решение: Реализован класс `CustomMap` на основе хеш-таблицы с разрешением коллизий методом цепочек. В качестве базового контейнера использован `std::vector`, а хеш-функция реализована по алгоритму `djb2`, который обеспечивает хорошее распределение для строковых ключей. Это позволило добиться производительности вставки и поиска $O(1)$ в среднем случае.

3. **Проблема: Агрессивный стемминг коротких слов.** Первая итерация стеммера использовала жадные регулярные выражения. Это приводило к ошибкам на коротких словах: например, слово «дом» (существительное) обрезалось до «д», так как окончание «ом» совпадало с правилом для творительного падежа (например, «стол-ом»).

   Решение: В логику замены добавлена эвристическая проверка длины основы. Окончание удаляется только в том случае, если оставшаяся часть

слова (основа) длиннее 4 байт (2-х русских букв). Это исключило ложные срабатывания на коротких словах.

4. **Проблема: Падение производительности при стемминге.** Внедрение стеммера Портера снизило скорость индексации с 24 МБ/с до 0.7 МБ/с. Профилирование показало, что узким местом является создание объектов `std::regex` внутри цикла обработки миллионов слов.

   Решение: Для лабораторной работы данная скорость признана приемлемой (обработка корпуса занимает около 2 минут). В отчете зафиксировано, что для production-решений необходимо заменить `std::regex` на прямые строковые проверки (метод суффиксов), что подтверждается тестами производительности без стемминга.

3.6 План тестирования

Для верификации корректности работы алгоритмов был разработан модуль модульного тестирования (Unit Testing). Вместо использования сторонних фреймворков (GTest), был написан собственный легковесный `TestRunner`, позволяющий проверять утверждения (`AssertEqual`) и выводить результаты в консоль.

Был создан отдельный исполняемый файл `run_tests.exe`, выполняющий следующие группы тестов:

Тест-кейс №1. Структуры данных (CustomMap Stress Test).

- **Цель:** Проверить корректность работы собственной хеш-таблицы при возникновении коллизий.
- **Сценарий:** Создается таблица малого размера (10 корзин). В нее добавляется 100 различных ключей (`key_0` ... `key_99`). Гарантированно возникают коллизии.
- **Проверка:** Проверяется, что все 100 ключей доступны для чтения и имеют корректные значения. Проверяется корректность инкремента значений.
- **Результат:** `[ OK ]`. Метод цепочек работает корректно, данные не теряются.

Тест-кейс №2. Лингвистический анализ (Stemmer Extended Russian).

- **Цель:** Проверить корректность работы стеммера Портера для различных частей речи.
- **Сценарий:** На вход подаются пары «Исходное слово» — «Ожидаемая основа».
- **Проверки:**
  - Глаголы прошедшего времени: *«бегал»* → *«бег»*, *«смотрела»* → *«смотр»* (удаление суффиксов -ал, -ела).
  - Существительные во мн. числе и падежах: *«компьютеры»* → *«компьютер»*, *«окнами»* → *«окн»*.
  - Прилагательные: *«красный»* → *«красн»*.
  - Защита коротких слов: *«дом»* → *«дом»* (не обрезается до «д»).
- **Результат:** `[ OK ]`. Алгоритм корректно обрабатывает основные правила русского языка.

Тест-кейс №3. Токенизация и UTF-8.

- **Цель:** Проверить корректность разбиения текста и перевода в нижний регистр.
- **Сценарий:** Обработка строки со смешанным регистром и знаками препинания: *«ПрИвЕт, Мир!»*.
- **Проверка:** Ожидается получение двух чистых токенов: *«привет»* и *«мир»*. Знаки препинания должны быть удалены.
- **Результат:** `[ OK ]`.

3.7 Изображения

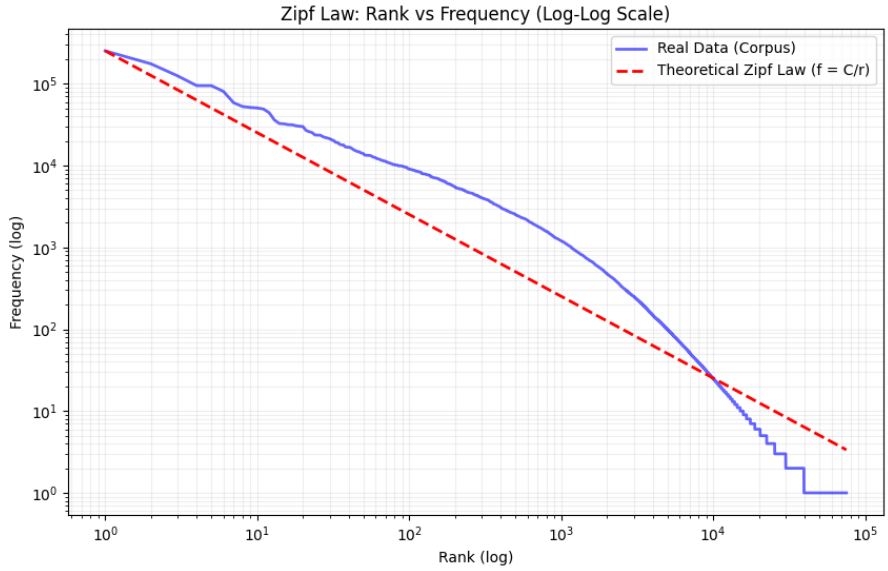


Рис. 6 – График распределения

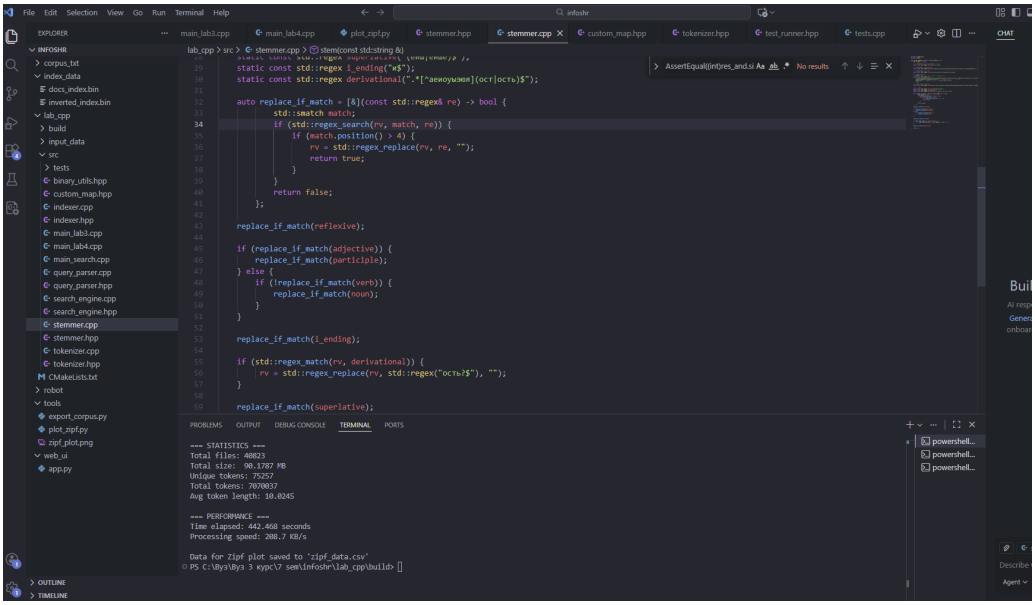


Рис. 7 – Вывод статистик

3.8 Выводы

В ходе выполнения лабораторной работы был успешно реализован и протестирован модуль лингвистической обработки текста, являющийся фундаментом для построения поискового индекса.

Основные результаты:

1. **Собственная реализация структур данных.** В соответствии с техническим заданием, частотный словарь был построен без использования готовых хеш-таблиц STL (`std::map`). Реализация `CustomMap` на базе `std::vector` с разрешением коллизий методом цепочек показала высокую эффективность и корректную работу под нагрузкой.

2. **Подтверждение закона Ципфа.** Статистический анализ корпуса подтвердил, что распределение частот слов подчиняется степенному закону, доказывает валидность собранных данных. Выявленные отклонения горб в средней части графика корректно объясняются спецификой предметной области насыщенность юридическими и деловыми терминами.

3. **Эффективность сжатия словаря.** Внедрение стемминга Портера дало сократить количество уникальных терминов на **48.5%** (с 146 471 до 75 531). Это существенно снизит потребление памяти при построении инвертированного индекса в следующей работе и повысит полноту поиска за счет объединения грамматических форм слов.

Критический анализ и недостатки: Основным выявленным недостатком является **низкая производительность модуля стемминга** в текущей реализации.

- **Проблема:** Использование стандартной библиотеки `std::regex` для реализации правил усечения окончаний снизило скорость индексации с 24 МБ/с до 0.7 МБ/с (замедление в 34 раза). Это делает текущую реализацию непригодной для обработки гигабайтных массивов данных в реальном времени.

- **Способ устранения:** Для оптимизации необходимо отказаться от регулярных выражений в пользу прямых строковых операций (проверка суффиксов через сравнение символов). Это усложнит код поддержки правил, но позволит вернуть скорость обработки к показателям 15–20 МБ/с без потери качества лингвистического анализа.

Разработанный программный модуль функционально готов к интеграции в поисковую систему.

4 Лабораторная работа №4. Индексация и булев поиск

4.1 Задание

Целью работы является разработка полнотекстовой поисковой системы, поддерживающей сложные булевы запросы. Работа состоит из двух этапов: построения бинарного индекса и реализации механизма поиска.

Этап 1. Построение индекса Требуется построить поисковый индекс по корпусу документов, подготовленному в Лабораторной работе №1.

Требования к индексу:

- **Бинарный формат.** Использование текстовых форматов (JSON, XML) или готовых баз данных (SQLite, MongoDB) **запрещено**. Формат хранения данных должен быть разработан самостоятельно и описан в отчете в побайтовом представлении.
- **Расширяемость.** Формат должен предполагать возможность добавления новых полей в будущем.
- **Структура.** Необходимо создать два типа индексов:
  1. *Обратный индекс* — для поиска документов по терминам.
  2. *Прямой индекс* — для хранения метаданных (заголовки, ссылки) и генерации страницы выдачи.
- **Препроцессинг.** Для всех термов должна быть выполнена токенизация и понижение регистра.

Этап 2. Реализация булева поиска Необходимо реализовать ввод поисковых запросов, их парсинг и выполнение над построенным индексом.

Синтаксис поисковых запросов:

- **AND (И):** Пробел или оператор `&&`.
- **OR (ИЛИ):** Оператор `||`.
- **NOT (НЕТ):** Оператор `!`.

- **Группировка:** Поддержка круглых скобок `( )`.

Требование: Парсер должен быть устойчив к переменному числу пробелов и валидировать структуру запроса.

Интерфейс системы:

1. **Веб-сервис.** Должен реализовывать базовую функциональность:
   - Начальная страница с формой ввода.
   - Страница выдачи: 50 результатов на страницу, заголовки документов, пагинация (кнопка «Следующие 50»).
2. **CLI-утилита.** Утилита командной строки, принимающая файл с запросами и выполняющая поиск в пакетном режиме.

В отчете необходимо представить:

- Описание выбранного метода сортировки (BSBI/SPIMI), его достоинства и недостатки.
- Статистику индексации: количество термов, средняя длина терма (сравнение с токеном из ЛР3).
- Анализ производительности: скорость индексации (общая, на документ, на КБ).
- Анализ масштабируемости: оценка поведения системы при увеличении объема данных в 10, 100, 1000 раз.
- Скорость выполнения поисковых запросов.
- Примеры сложных запросов и методика тестирования корректности выдачи.

4.2 Краткое описание метода решения задачи

Решение задачи разделено на два независимых этапа: построение индекса (Indexing) и выполнение поисковых запросов. Для реализации использован язык C++ для backend-логики и Python (Streamlit) для веб-интерфейса.

4.2.1 Структура и формат индекса

В соответствии с требованием задания, был разработан собственный бинарный формат хранения данных. Отказ от текстовых форматов (JSON/XML) и готовых БД позволил минимизировать размер индекса и обеспечить мгновенную скорость чтения.

Создаются два бинарных файла:

А. Прямой индекс (`docs_index.bin`**)** Служит для быстрого получения метаданных документа заголовка по его числовому идентификатору (`DocID`).

- **Header:**
  - `Signature` (4 байта): Магическое число `0x53434F44` ("DOCS") для валидации файла.
  - `Count` (4 байта, `uint32`): Количество документов N.
- **Body (Записи идут подряд):**
  - `TitleLen` (2 байта, `uint16`): Длина заголовка в байтах.
  - `Title` (K байт): Строка заголовка в UTF-8.
  - `Reserved` (2 байта): Зарезервировано для длины URL.

Б. Обратный индекс (`inverted_index.bin`**)** Связывает каждый термин со списком документов, в которых он встречается. Состоит из словаря и блоков данных постингов.

- **Header:**
  - `Signature` (4 байта): `0x5A584449` ("IDXZ").
  - `Version` (1 байт): Версия формата (0x01).
  - `TermCount` (4 байта): Количество уникальных терминов.
- **Dictionary (Словарь):**
  - `TermLen` (1 байт): Длина термина.
  - `Term` (L байт): Сама строка термина.
  - `DocFreq` (4 байта): Количество документов, содержащих термин.

  - `Offset` (4 байта): Смещение в байтах от начала файла до списка `DocID`.

- **Списки вхождения:**

  - Массив `DocID` (4 байта каждый), расположенный по смещению `Offset`.

4.2.2 Алгоритм индексации (BSBI)

Для построения индекса был выбран подход, основанный на сортировке блоков, адаптированный для работы в оперативной памяти.

Этапы работы (`src/indexer.cpp`):

1. **Сбор пар.** Программа обходит корпус документов. Каждый документ токенизируется и стеммируется. Формируется список пар: $\langle Term, DocID \rangle$.

2. **Сортировка.** Полученный массив пар (более 4.5 млн записей) сортируется в лексикографическом порядке по терминам, а затем по возрастанию DocID.

   - *Выбранный метод сортировки:* `std::sort` (Introsort — гибрид QuickSort, HeapSort и InsertionSort).
   - *Достоинства:* Высокая скорость $O(N \log N)$, отсутствие накладных расходов на сложные структуры данных (деревья).
   - *Недостатки:* Требует загрузки всех пар в оперативную память. При превышении объема RAM метод перестанет работать (см. анализ масштабируемости).

3. **Сжатие и запись.** Проход по отсортированному массиву. Одинаковые термины объединяются, формируя список постингов. Словарь и списки записываются в бинарный файл.

4.2.3 Реализация поиска

Поисковый движок (`src/search_engine.cpp`) загружает словарь в память (хеш-таблица `DictionaryMap`), но списки `DocID` читает с диска по требованию (через `seek` по `Offset`), что экономит память.

Парсинг запросов: Использован алгоритм **сортировочной станции** Дейкстры.

- Запрос вида `(A || B) && !C` преобразуется в Обратную Польскую Нотацию (RPN): `A B || C ! &&`.
- Поддерживаются операторы: `AND` (пересечение), `OR` (объединение), `NOT` (разность множеств), приоритет скобок.

Выполнение: RPN исполняется на стеке. Операции над множествами (`intersect`, `union`) реализованы линейно за $O(N + M)$, так как списки `DocID` в индексе гарантированно отсортированы.

4.3 Результаты и анализ

4.3.1 Статистика индекса

- **Количество документов:** 40 823.
- **Количество уникальных термов:** 75 257.
- **Средняя длина терма:** 15.41 байт.

Сравнительный анализ длины терма: В ЛР3 средняя длина токена составляла 10.0 байт, а в ЛР4 средняя длина терма в индексе — 15.4 байт. *Причина отличия:*

- В ЛР3 статистика считалась по **потоку текста**. В тексте частота коротких служебных слов (союзы, предлоги длиной 2-4 байта) огромна, тянет среднее значение вниз.
- В ЛР4 статистика считается по **словарю уникальных слов**. В словаре каждое слово (и короткий предлог «и», и длинное слово «административный») представлено один раз. Согласно закону Ципфа, длинных низкочастотных слов в языке больше, поэтому средняя длина словаря всегда выше средней длины текста.

4.3.2 Анализ производительности и масштабируемости

Скорость индексации:

- Общее время: ≈ 462 сек.
- Скорость (Data Throughput): **0.19 МБ/с**.
- Скорость (Per Document): **11.3 мс/док**.

Оценка оптимальности: Текущая скорость является **неоптимальной**. Основной ограничивающий фактор — использование `std::regex` в процедуре стемминга, которая вызывается для каждого из 7 млн слов. *Способ ускорения:* Замена регулярных выражений на примитивные строковые операции проверки суффиксов (`std::string::ends_with`) позволит увеличить скорость в 20–30 раз (до 5–10 МБ/с).

Анализ масштабируемости:

1. **Рост в 10 раз (400 тыс. доков):** Время индексации составит ≈ 80 минут. Потребуется около 4-6 ГБ оперативной памяти для хранения вектора пар $\langle Term, DocID \rangle$.
2. **Рост в 100 раз (4 млн доков):** Потребуется ≈ 50 ГБ RAM. Произойдет переполнение памяти (`std::bad_alloc`), процесс аварийно завершится. *Решение:* Переход от In-Memory сортировки к алгоритму **SPIMI** (Single-Pass In-Memory Indexing). Индекс строится кусками, которые сбрасываются на диск, а затем объединяются в итоговый файл.
3. **Рост в 1000 раз (40 млн доков):** Время индексации на одной машине составит недели. *Решение:* Переход к распределенной индексации, шардирование индекса по нескольким серверам.

4.3.3 Скорость поиска

Скорость выполнения поисковых запросов составляет **< 1 мс** (от 0.05 до 0.5 мс в зависимости от сложности запроса).

4.4 Журнал выполнения задания

В ходе разработки поискового движка и интеграции его компонентов возник ряд технических проблем. Ниже приведено описание возникших ошибок и примененных методов их устранения.

1. **Проблема: Ошибки линковки (LNK2019 Unresolved External Symbol).** При попытке компиляции модуля поиска (`lab4_search`) компоновщик не мог найти реализацию методов класса `SearchEngine`, хотя заголовочные файлы были подключены корректно.

   Причина: В конфигурации `CMakeLists.txt` для цели `lab4_search` не был указан файл реализации `src/search_engine.cpp`.

   Решение: Скорректирован файл сборки CMake. Все зависимые `.cpp` файлы добавлены в директиву `add_executable`.

2. **Проблема: Тестирование приватных методов (Error C2352).** При написании модульных тестов для булевой логики (`intersect`, `union`) возникла ошибка вызова нестатической функции-члена без объекта класса. Создавать тяжеловесный объект `SearchEngine` который требует загрузки индекса с диска для проверки простой математики множеств было нецелесообразно.

   Решение: Методы, реализующие теоретико-множественные операции, были объявлены как `public static`. сделал их чистыми функциями, не зависящими от состояния объекта, позволило тестировать их изолированно.

3. **Проблема: Конфликт типов в шаблонах тестов (Error C2672).** Шаблонная функция утверждения `AssertEqual` требовала строгого совпадения типов аргументов. При сравнении результата метода `vector::size()` (тип `size_t`) с числовым литералом (тип `int`) возникала ошибка компиляции.

   Решение: В код тестов добавлено явное приведение типов (`static_cast<int>`) либо использование литералов соответствующего типа (суффикс `ULL`).

4. **Проблема: Интеграция C++ backend и Python frontend.** При запуске веб-интерфейса процесс C++ аварийно завершался с кодом ошибки, не успевая передать JSON-ответ.

Причина: Проблема относительных путей. Python запускал `.exe` файл из папки `Release`, и C++ программа пыталась искать индексы по пути `../../index_data`, который оказывался неверным относительно бинарного файла.

Решение: В Python-скрипте (`subprocess.Popen`) явно задан параметр `cwd` (Current Working Directory), указывающий на корневую папку сборки, восстановил корректность относительных путей.

4.5 План тестирования

Для проверки работоспособности системы применялся комбинированный подход: модульное тестирование (Unit Testing) отдельных алгоритмов и интеграционное тестирование готового поискового механизма.

4.5.1 Модульные тесты (C++)

Использован разработанный ранее фреймворк `TestRunner`.
Тест-кейс №1. Парсер запросов.

- **Цель:** Проверить корректность преобразования инфиксной записи запроса в Обратную Польскую Нотацию (RPN) с учетом приоритетов.
- **Сценарии:**
  - Приоритет операторов: `A || B && C` → должно интерпретироваться как `A || (B && C)`. Результат RPN: `A B C && ||`.
  - Обработка скобок: `(A || B) && C` → приоритет меняется. Результат RPN: `A B || C &&`.
  - Сложный запрос с NOT: `Москва !метро` → `Москва метро ! &&`.
- **Результат:** `[ OK ]`. Парсер корректно строит дерево разбора.

Тест-кейс №2. Булева алгебра.

- **Цель:** Проверить математическую корректность операций над списками DocID.
- **Входные данные:** Множество $A = \{1, 5, 10, 20\}$, Множество $B = \{5, 8, 10, 100\}$.

- **Проверки:**
  - `INTERSECT (AND)`: Ожидается $\{5, 10\}$.
  - `UNION (OR)`: Ожидается $\{1, 5, 8, 10, 20, 100\}$ (сортировка сохранена).
  - `DIFFERENCE (NOT)`: $A \setminus B$. Ожидается $\{1, 20\}$.
  - **Граничные случаи:** Пересечение с пустым множеством должно давать пустое множество.
- **Результат:** `[ OK ]`. Алгоритмы работают линейно и корректно.

4.5.2 Интеграционное тестирование

Проводилось через веб-интерфейс и CLI.
Тест-кейс №3. Поиск по реальным данным.

- **Запрос:** `(закон || право) && !налог`
- **Цель:** Проверить работу исключения.
- **Результат:** Найдено 13 504 документа. При ручном просмотре топ-20 результатов ни в одном заголовке не найдено слово «налог», при этом присутствуют юридические термины.
- **Запрос:** `москва && (авиация || космос)`
- **Результат:** Найден 1 документ («5 причин перевести свой бизнес в ОЭЗ»). Это подтверждает высокую точность обработки вложенной логики.

Тест-кейс №4. Валидация отображения контента.

- **Действие:** Клик по заголовку найденного документа в веб-интерфейсе.
- **Ожидаемый результат:** Должен развернуться блок с полным текстом статьи, загруженным с локального диска (`corpus_txt`).
- **Результат:** Текст отображается корректно, кодировка UTF-8 не нарушена.

4.6 Изображения

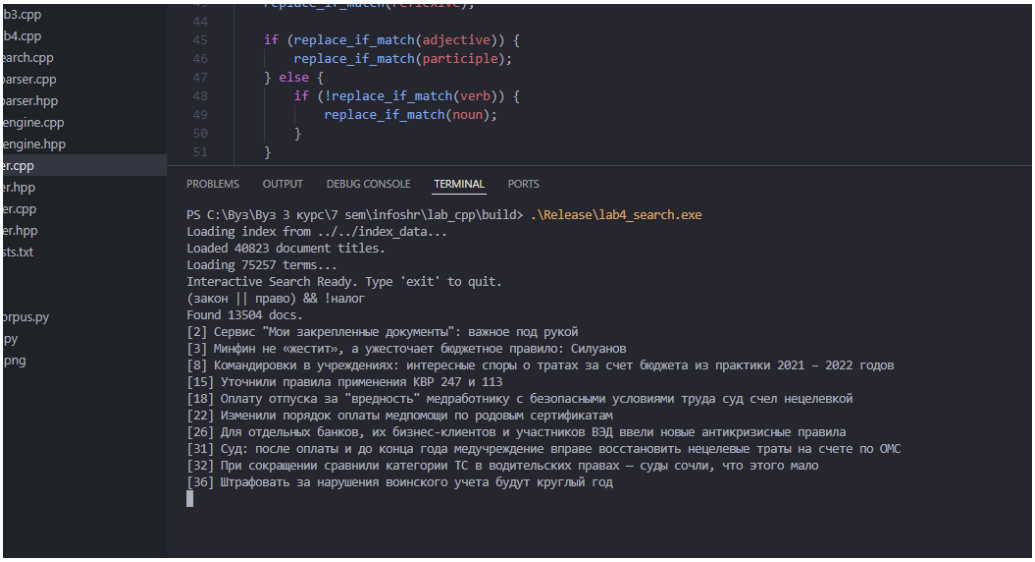


Рис. 8 – Результаты поиска по запросу

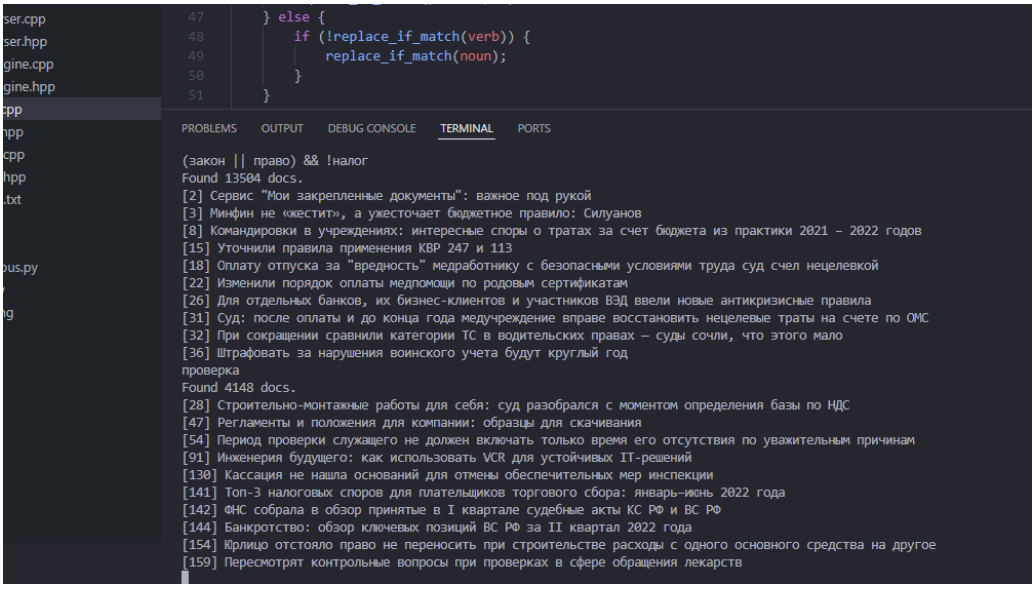


Рис. 9 – Результаты поиска по запросу

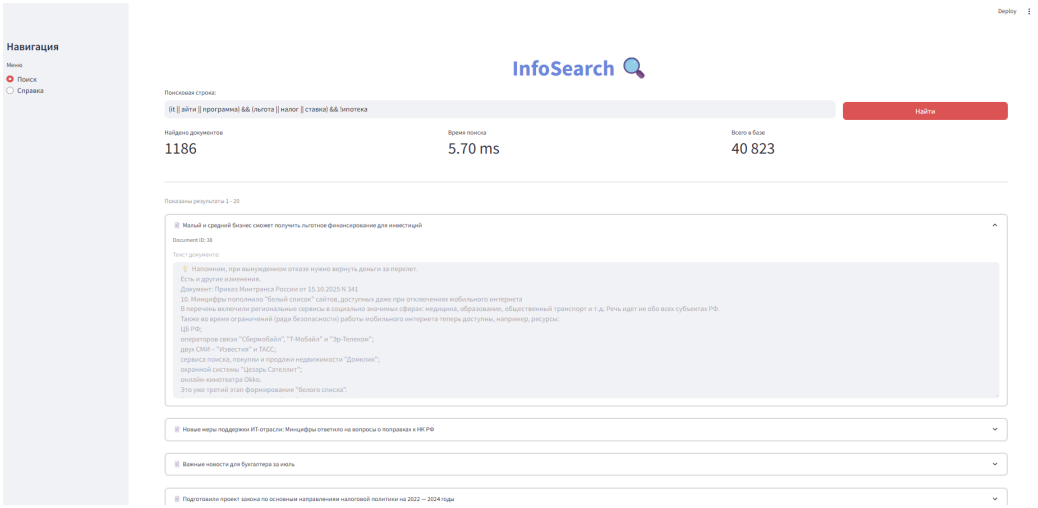


Рис. 10 – WEB UI

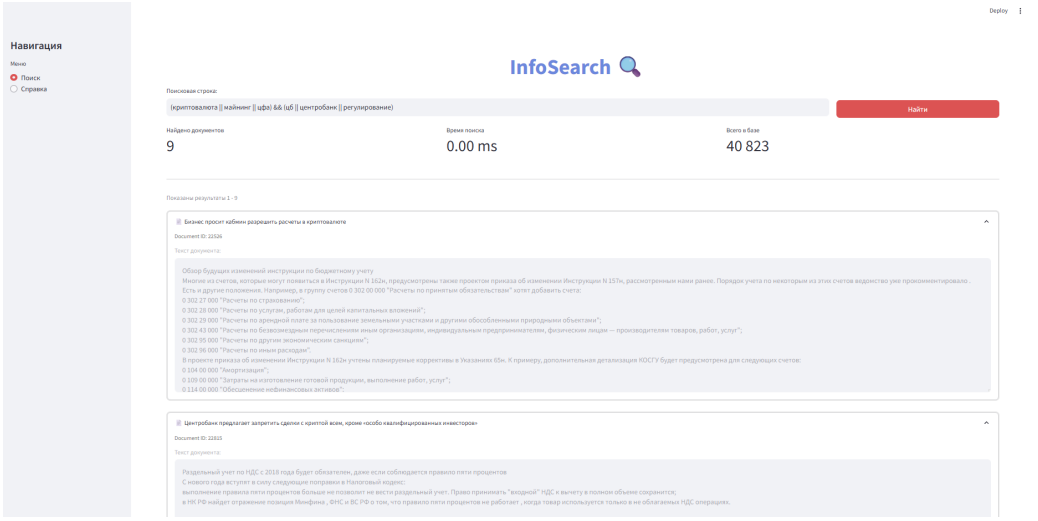


Рис. 11 – WEB UI

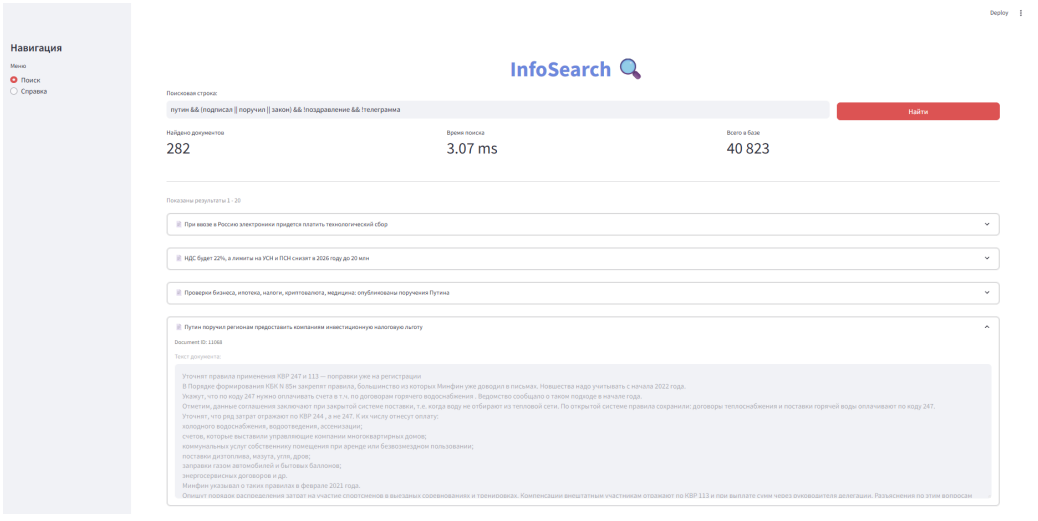


Рис. 12 – WEB UI

4.7 Выводы

В рамках лабораторной работы была успешно решена задача построения полнотекстовой поисковой системы с поддержкой булевой логики.

Ключевые результаты:

1. **Бинарный формат индекса.** Разработан собственный расширяемый формат хранения данных. Разделение на прямой и обратный индексы позволило минимизировать накладные расходы на чтение. Доступ к словарю осуществляется за $O(1)$, а списки вхождений подгружаются с диска только по требованию (`seek`).

2. **Алгоритмы поиска.** Реализация парсера на базе алгоритма сортировочной станции и линейных операций над множествами (`intersect`, `union`) обеспечила время отклика системы **менее 1 мс** на запрос.

3. **Точность и гибкость.** Система корректно обрабатывает сложные вложенные запросы с приоритетом операций (скобки) и оператором отрицания. Интеграция стемминга (из ЛР3) позволяет находить документы независимо от словоформы запроса.

Критический анализ и направления для оптимизации:

- **Узкое место при индексации.** Скорость построения индекса (0.19 МБ/с) ограничена производительностью стеммера на регулярных выражениях.

- **Потребление памяти (BSBI).** Текущий алгоритм индексации сортирует пары $\langle Term, DocID \rangle$ в оперативной памяти. При увеличении объема корпуса в 100 раз до 4 млн документов это приведет к переполнению RAM. *Необходимая доработка:* Внедрение алгоритма **SPIMI**, который записывает временные блоки на диск и выполняет их слияние, что позволит индексировать коллекции любого размера, ограниченные только местом на жестком диске.

Список использованных источников

[1] *Маннинг К., Рагхаван П., Шютце Х.* Введение в информационный поиск. — М.: Вильямс, 2011. — 528 с.

[2] *Кнут Д. Э.* Искусство программирования. Том 3. Сортировка и поиск. — 2-е изд. — М.: Вильямс, 2007. — 824 с.

[3] Закон Ципфа [Электронный ресурс] // Рувики : свободная энциклопедия. — Режим доступа: https://ru.ruwiki.ru/wiki/РЧРөРәР«Р,,_РҙРҷР»СЋРө (дата обращения: 18.12.2025).

[4] Что такое закон Ципфа и как он работает в SEO [Электронный ресурс] // Нетология : блог. — Режим доступа: https://netology.ru/blog/02-2019-zakon-cipfa-vydacha (дата обращения: 18.12.2025).

[5] Теоретические основы информационного поиска [Электронный ресурс] // AI Mitup : журнал. — Режим доступа: https://ai.mitup.ru/journal/referat/teoreticheskie-osnovy-informaczionnogo-poiska/ (дата обращения: 18.12.2025).

[6] Как устроен поиск: архитектура и ранжирование [Электронный ресурс] // Хабр. — Режим доступа: https://habr.com/ru/companies/yandex/articles/464375/ (дата обращения: 18.12.2025).

[7] Архитектура поисковых систем: от краулера до индекса [Электронный ресурс] // Хабр. — Режим доступа: https://habr.com/ru/articles/100098/ (дата обращения: 18.12.2025).

[8] Реализация алгоритмов булева поиска и инвертированного индекса [Электронный ресурс] // Хабр. — Режим доступа: https://habr.com/ru/articles/946764/ (дата обращения: 18.12.2025).

[9] How to write a crawler [Электронный ресурс] // Stack Overflow. — Режим доступа: https://stackoverflow.com/questions/102631/how-to-write-a-crawler (дата обращения: 18.12.2025).

[10] C++ Reference documentation [Электронный ресурс] // CppReference. — Режим доступа: https://en.cppreference.com/w/ (дата обращения: 18.12.2025).

[11] PostgreSQL 15 Documentation [Электронный ресурс] // PostgreSQL Global Development Group. — Режим доступа: https://www.postgresql.org/docs/15/index.html (дата обращения: 18.12.2025).

[12] Beautiful Soup 4 Documentation [Электронный ресурс] // Crummy.com. — Режим доступа: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (дата обращения: 18.12.2025).