

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

**Факультет информационных технологий и прикладной
математики**

Кафедра вычислительной математики и программирования

Лабораторная работа по курсу «Информационный поиск»

Студент: Кочкожаров Иван Вячеславович
Преподаватель: А. А. Кухтичев
Группа: М8О-408Б-22
Дата: 19.02.2026
Оценка: Удовлетворительно
Подпись: Кочкожаров Иван Вячеславович

Москва, 2026

Задание

Выполнить комплекс лабораторных работ по курсу «Информационный поиск» на оценку «удовлетворительно». Необходимо реализовать поисковую систему для тематического корпуса текстов с веб-интерфейсом и утилитой командной строки.

Перечень лабораторных работ

1. **Добыча корпуса документов** — скачать корпус, изучить его характеристики, выделить текст, найти существующие поисковики и их недостатки.
2. **Поисковый робот** — реализовать автоматический сбор документов из веб-источника с фильтрацией нерелевантного контента.
3. **Токенизация** — реализовать разбиение текстов на токены, выработать правила, описать достоинства и недостатки.
4. **Стемминг** — добавить стемминг в поисковую систему, реализовать алгоритм Портера для русского языка.
5. **Закон Ципфа** — построить график распределения терминов, наложить теоретическую кривую, объяснить расхождения.
6. **Булев индекс** — реализовать инвертированный индекс на собственных структурах данных (без STL).
7. **Булев поиск** — реализовать поиск с операторами AND, OR, NOT, создать веб-интерфейс и утилиту командной строки.

Требования к реализации

- Язык программирования для поискового движка: C++ без STL (допускается `std::string` и `std::vector` для токенизации и базовой работы с данными; хеш-таблицы, деревья и прочие контейнеры реализуются самостоятельно).
- Для краулера и вспомогательных компонент: Python.
- Корпус: минимум 30 000 документов минимум по 1000 слов.
- Кодировка: UTF-8.
- Интерфейс: веб-сервис с формой ввода и утилита командной строки.

1 Добыча корпуса документов

1 Выбор тематики

В качестве тематики корпуса выбрана **Лингвистика** — наука о языке, охватывающая фонетику, фонологию, морфологию, синтаксис, семантику, лексикологию, грамматику, прикладную и компьютерную лингвистику, а также смежные области (психолингвистика, социолингвистика, этимология, письменность и др.).

Для формирования корпуса используются два источника: русскоязычный раздел Википедии (~30 000 статей) и научная электронная библиотека КиберЛенинка (~500 статей по разделу «Языкознание и литературоведение»). Итого корпус содержит ~30 500 документов.

2 Источники данных

Источник 1: русскоязычный раздел Википедии (<https://ru.wikipedia.org>).

Начальные категории для обхода (из `scraper/config.yaml`):

- Лингвистика
- Языки, Языкознание
- Фонетика, Фонология
- Морфология (лингвистика), Синтаксис
- Семантика, Лексикология
- Грамматика, Прикладная лингвистика
- Психолингвистика, Социолингвистика
- Когнитивная лингвистика, Лингвистическая типология
- Диалектология, Стилистика
- Этимология, Орфография
- Письменность, Переводоведение
- Терминология, Компьютерная лингвистика

Источник 2: научная электронная библиотека КиберЛенинка (<https://cyberleninka.ru>).

Статьи загружаются из раздела каталога «Языкознание и литературоведение» (`/article/c/language`). Скрaper последовательно обходит страницы каталога, загружает полные тексты научных статей и фильтрует их по минимальной длине (1 000 слов). Результат сохраняется в `corpus2.json`.

3 Характеристики корпуса

Параметр	Значение
Документов из Википедии	~30 000
Документов из КиберЛенинки	~500
Итого документов	~30 500
Формат хранения	NDJSON (<code>corpus.ndjson + corpus2.ndjson</code>)
Минимальный размер документа	1000 слов
Язык	русский (с включениями на других языках)

Таблица 1: Характеристики корпуса документов

Каждый документ (из обоих источников) содержит следующие поля:

- `url` — адрес страницы (Википедия или КиберЛенинка)
- `title` — заголовок статьи
- `text` — очищенный текст статьи
- `word_count` — количество слов
- `paragraph_count` — количество параграфов

Оба корпуса загружаются в единый индекс движком поиска.

4 Существующие поисковики для данного корпуса

1. **Встроенный поиск Википедии** — ограничен только Википедией, нет возможности настраивать алгоритмы ранжирования, отсутствует явный булев поиск с операторами.
2. **Поиск КиберЛенинки** — полнотекстовый поиск по научным статьям, но отсутствует булев поиск с явными операторами, невозможно объединить с другими источниками.
3. **Google (site:ru.wikipedia.org)** — высокое качество ранжирования, но закрытая система, невозможно изучить внутреннюю работу и алгоритмы.
4. **Яндекс (site:ru.wikipedia.org)** — аналогично Google, проприетарная реализация.

5 Недостатки существующих решений

- Невозможность настройки и изучения алгоритмов ранжирования
- Отсутствие булевого поиска с явными операторами AND, OR, NOT
- Невозможность проведения статистического анализа корпуса (закон Ципфа, частотный анализ)
- Закрытость реализации для учебных и исследовательских целей
- Ограниченный контроль над процессом токенизации и стемминга

6 Журнал выполнения задания

При выполнении задания по сбору корпуса документов были выявлены следующие проблемы и их решения:

- Выбор подходящего объёма тематики:** Изначально рассматривались более узкие темы (например, только «Фонетика» или «Морфология»), но они не обеспечивали требуемый минимум в 30 000 документов. Расширение до общей категории «Лингвистика» с включением подкатегорий (языки, фонетика, синтаксис, семантика и др.) позволило достичь целевого объёма.
- Категории Википедии: слишком широкие или узкие:** Некоторые категории (например, «Лингвистика») содержат сотни тысяч статей. Другие категории (например, «Терминология») слишком узкие. Решение: комбинация широких и узких категорий с последующей фильтрацией по длине текста.
- Обеспечение минимума в 30 000 документов:** После фильтрации по минимальной длине (1 000 слов) количество документов достигает 30 000. Это потребовало расширения списка начальных категорий и настройки глубины обхода дерева категорий до 5 уровней.

7 Выводы

Корпус документов по тематике «Лингвистика» успешно собран из двух источников: ~30 000 энциклопедических статей из русскоязычной Википедии и ~500 научных статей из КиберЛенинки (~30 500 документов суммарно).

Качество корпуса:

- Тематическая однородность обеспечивается выбором релевантных категорий и фильтрацией по длине текста
- Минимальный размер документа (1 000 слов) гарантирует достаточную информативность для построения поискового индекса
- Формат NDJSON удобен для потоковой обработки и не требует загрузки всего корпуса в память

Преимущества двух источников:

- Энциклопедические статьи из Википедии дают широкий охват тематики
- Научные статьи из КиберЛенинки добавляют специализированную терминологию и академический стиль
- Объединённый корпус обеспечивает более полное покрытие предметной области

Недостатки текущего подхода:

- Зависимость от структуры категорий Википедии, которая может быть неполной или устаревшей
- Раздел КиберЛенинки «Языкознание и литературоведение» включает и литературоведческие статьи, не все из которых релевантны тематике лингвистики
- Отсутствие контроля над качеством содержимого (возможны шаблонные или неполные статьи)

2 Поисковый робот

1 Архитектура робота

Поисковый робот реализован на языке Python с использованием асинхронной загрузки контента. Основные библиотеки: `aiohttp` (асинхронные HTTP-запросы), `BeautifulSoup` (парсинг HTML).

Скрaper поддерживает два режима работы:

- **Wikipedia** (`python scraper.py wiki`) — обход категорий Википедии, сбор ~30 000 статей в `corpus.ndjson`
- **КиберЛенинка** (`python scraper.py cyberleninka`) — последовательный обход каталога научных статей, сбор ~500 статей в `corpus2.ndjson`

Оба корпуса загружаются в единый индекс поискового движка.

2 Алгоритм работы

Фаза 1 — обнаружение статей:

1. Загрузка списка seed-категорий из конфигурационного файла `config.yaml`.
2. Для каждой категории — запрос к Wikipedia API (`action=query&list=categorymembers`) для получения статей и подкатегорий.
3. Рекурсивный обход подкатегорий (до глубины 10) с параллельным выполнением (батчи по 10).
4. Дедупликация по заголовку статьи.

Фаза 2 — загрузка контента:

1. Для каждой обнаруженной статьи — запрос к API (`action=parse`) для получения HTML.
2. Извлечение текста из HTML: параграфы, списки, заголовки.
3. Фильтрация по минимальной длине текста.
4. Дедупликация по каноническому заголовку (разрешение редиректов).
5. Сохранение в NDJSON.
6. Параллельная обработка (до 30 одновременных запросов).

3 Критерии фильтрации

Для обеспечения качества корпуса применяются следующие фильтры:

Критерий	Значение
Минимальная длина текста	1000 слов
Исключение служебных страниц	Категории, шаблоны, обсуждения, файлы
Исключение дубликатов	По нормализованному URL

Таблица 2: Критерии фильтрации документов

Список исключаемых паттернов URL:

- **Special:**, Служебная: — служебные страницы
- **File:**, Файл: — страницы файлов
- **Category:**, Категория: — страницы категорий (используются только для навигации)
- **Template:**, Шаблон: — шаблоны
- **Talk:**, Обсуждение: — страницы обсуждений
- **User:**, Участник: — страницы участников
- **Wikipedia:**, Википедия: — внутренние страницы Википедии

4 Формат выходных данных

Каждая строка файла `corpus.ndjson` содержит один JSON-объект:

```
{"url": "https://ru.wikipedia.org/wiki/Алфавит",
"title": "Алфавит",
"text": "Алфавит – совокупность букв или символов...",
"word_count": 4521,
"paragraph_count": 38}
```

5 Обработка проблемных ситуаций

- **Таймауты:** при отсутствии ответа в течение 30 секунд запрос пропускается.
- **Ошибки HTTP:** страницы с кодами 4xx/5xx пропускаются.
- **Кодировка:** все тексты приводятся к UTF-8.
- **Ограничение нагрузки:** задержка между запросами предотвращает блокировку.
- **User-Agent:** используется идентификатор `LinguisticsSearchBot/1.0`.

6 Скрaper КиберЛенинки

Для КиберЛенинки реализован отдельный скрапер (`CyberLeninkaScraper`), работающий по упрощённой схеме:

1. Последовательный обход страниц каталога «Языкоизнание и литературоведение» (`/article/c/l`) — каждая страница содержит ~20 ссылок на статьи

2. Для каждой ссылки — загрузка полной страницы статьи
3. Извлечение текста из раздела «Текст научной работы» (после заголовка h2), с остановкой на «Список литературы»
4. Фильтрация по минимальной длине (1 000 слов)
5. Сохранение в `corpus2.ndjson`

В отличие от Википедии, не требуется обход ссылок внутри статей — статьи берутся непосредственно из каталога. Задержка между запросами увеличена до 300 мс для снижения нагрузки на сервер.

7 Контейнеризация

Скрaperы упакованы в Docker-контейнеры и запускаются командами:

```
docker compose run --rm scraper
docker compose run --rm scraper-cyberleninka
```

После завершения работы корпуса доступны в файлах `data/corpus.ndjson` и `data/corpus2.ndjson`.

8 Журнал выполнения задания

При разработке и запуске поискового робота были выявлены следующие проблемы и их решения:

1. **Ограничение скорости запросов к Wikipedia API:** Wikipedia API имеет лимиты на количество запросов в секунду. При параллельной обработке более 30 одновременных запросов возникали ошибки 429 (Too Many Requests). Решение: ограничение параллелизма до 30 запросов и добавление задержек между батчами запросов (0.1 секунды).
2. **Дублирование статей через разрешение редиректов:** Многие статьи в Википедии имеют редиректы (например, «Толстой» редиректит на «Лев Толстой»). При обходе категорий обе версии попадали в список, что приводило к дублированию. Решение: использование канонического заголовка статьи через параметр `redirects=1` в API и дедупликация по нормализованному заголовку перед сохранением.
3. **Управление памятью при больших объёмах данных:** При обходе категорий накапливался список из 50 000 названий статей в памяти. Для загрузки контента использовалась потоковая обработка: каждая статья обрабатывалась и сразу записывалась в файл, без накопления всего корпуса в памяти. Это позволило обрабатывать корпус на машинах с ограниченной оперативной памятью.
4. **Обработка таймаутов:** Некоторые статьи Википедии очень большие (десятка тысяч символов), и их загрузка через API могла превышать 30 секунд. Решение: установка таймаута на уровне HTTP-клиента (`aiohttp`) и пропуск проблемных статей с логированием ошибок для последующего анализа.
5. **Асинхронная архитектура:** Изначально использовался синхронный подход с библиотекой `requests`, что приводило к длительному времени выполнения (за 10 часов для не удалось собрать 30 500 статей). Переход на асинхронный подход с `aiohttp` сократил время до 2–3 часов при том же объёме данных.

9 Выводы

Поисковый робот успешно реализован с использованием асинхронного подхода и обеспечивает сбор корпуса из 30 000 документов за приемлемое время.

Анализ асинхронного подхода: Асинхронная архитектура с использованием aiohttp показала значительное преимущество по сравнению с синхронной реализацией:

- Ускорение обработки раза за счёт параллельных HTTP-запросов
- Эффективное использование сетевых ресурсов: пока один запрос ожидает ответа, обрабатываются другие
- Масштабируемость: легко увеличить параллелизм при наличии достаточной пропускной способности сети

Недостатки и ограничения:

- Зависимость от доступности Wikipedia API и его лимитов
- Отсутствие механизма возобновления после полного сбоя (необходимо перезапускать обход категорий)
- Фильтрация по длине текста может исключать короткие, но релевантные статьи (например, статьи о малоизвестных произведениях)
- HTML-парсинг может пропускать текст, встроенный в сложные шаблоны Википедии

Возможные улучшения:

- Реализация механизма чекпоинтов для возобновления обхода с последней обработанной категории
- Использование дампа Википедии вместо API для более быстрого и полного сбора данных

3 Токенизация

1 Определение и цели

Токенизация — процесс разбиения текста на элементарные единицы (токены), которые используются для построения поискового индекса. Токен — последовательность символов, представляющая слово или другую значимую лексическую единицу.

Цели токенизации:

- Выделение значимых единиц текста из потока символов
- Нормализация: приведение к единообразному виду (lowercase)
- Фильтрация шума: удаление HTML-артефактов, служебных символов
- Формирование словаря для индексации

2 Реализация

Токенизатор реализован в виде класса `Tokenizer` на C++. Основная функция `tokenize` принимает строку текста и возвращает вектор токенов. Каждый токен содержит текст и позицию в исходном документе.

Алгоритм работы (однопроходный, сложность $O(n)$):

1. Последовательное чтение символов из входной строки
2. Определение типа символа: кириллический (UTF-8, двухбайтовый), латинский, цифра или разделитель
3. Накопление символов в буфере до встречи разделителя
4. При встрече разделителя — проверка валидности токена
5. Если токен валиден — добавление в результирующий список

2.1 Разделители

- Пробельные символы: пробел, табуляция, перевод строки
- Знаки препинания: точка, запятая, восклицательный и вопросительный знаки, точка с запятой, двоеточие
- Скобки: круглые, квадратные, фигурные
- Кавычки: одинарные и двойные
- Специальные символы: угловые скобки, слеши, амперсанды

2.2 Обработка UTF-8 кириллицы

Кириллические символы в UTF-8 представлены двухбайтовыми последовательностями. Токенизатор проверяет первый байт (0xD0 или 0xD1) и читает следующий байт для формирования полного символа. Приведение к нижнему регистру выполняется на уровне байтов UTF-8.

2.3 Фильтрация токенов

1. Токены короче 2 символов отбрасываются
2. Последовательности, состоящие только из цифр, удаляются
3. Все символы приводятся к нижнему регистру

3 Журнал выполнения задания

При реализации токенизатора были выявлены следующие проблемы и их решения:

1. **Обработка UTF-8 кириллицы:** Основная сложность заключалась в корректной обработке двухбайтовых последовательностей UTF-8 для кириллических символов. Стандартные функции C++ (`std::tolower`) не работают напрямую с UTF-8. Решение: побайтовый анализ с проверкой первого байта (0xD0 или 0xD1) и чтением второго байта для формирования полного символа. Приведение к нижнему регистру выполнено через таблицу соответствий для кириллицы.
2. **Обработка дефисов в составных словах:** Слова типа «военно-морской», «научно-фантастический» разбивались на отдельные токены из-за того, что дефис рассматривался как разделитель. Это приводило к потере семантической целостности терминов. Решение: дефис оставлен как разделитель, так как альтернативный подход (сохранение дефисов как части токена) создавал бы проблемы с поиском отдельных компонентов составных слов.
3. **Смешанный текст (русский и латинский):** В корпусе встречаются лингвистические термины и названия на латинице (например, «phoneme», «IPA»). Токенизатор должен корректно обрабатывать оба алфавита. Решение: разделенная обработка кириллических и латинских символов с едиными правилами фильтрации и нормализации.

4 Выводы

Токенизатор реализован как однопроходный алгоритм с корректной обработкой UTF-8 кириллицы и обеспечивает базовую нормализацию текста для построения поискового индекса.

Критический анализ качества токенизатора:

- **Сильные стороны:** Высокая производительность ($O(n)$), корректная работа с кириллицей без внешних библиотек, детерминированность результатов, низкое потребление памяти за счёт потоковой обработки.
- **Слабые стороны:** Отсутствие контекстного анализа приводит к проблемам с омонимами и составными словами. Фильтрация коротких токенов может удалять важные аббревиатуры и имена (например, «А.С. Пушкин» теряет инициалы).
- **Влияние на качество поиска:** Простые запросы (одиночные слова) обрабатываются корректно. Сложные запросы с именами собственными или составными терминами могут давать неполные результаты из-за разбиения на отдельные токены.

Влияние на поисковую систему:

- Токенизатор формирует основу для инвертированного индекса — каждое слово становится ключом для поиска
- Потеря информации при разбиении составных слов снижает точность поиска по специализированным терминам
- Фильтрация чисел исключает возможность поиска по датам и числовым данным (например, «1869 год» не найдёт документы с упоминанием этой даты)

Возможные улучшения:

- Использование морфологического анализатора для более точного выделения границ слов
- Словарь исключений для составных терминов («военно-морской», «XIX век»)
- Сохранение контекста (позиция в документе) для фразового поиска
- Обработка аббревиатур и сокращений через специальные правила или словари

4 Стемминг

1 Алгоритм Портера для русского языка

Стемминг — процесс нормализации слов путём удаления суффиксов для приведения словоформ к единой основе. В системе реализован алгоритм Портера, адаптированный для русского языка.

Алгоритм последовательно удаляет суффиксы различных частей речи, используя понятие области RV — части слова после первой гласной.

2 Этапы алгоритма

1. **Определение позиции RV** — находится позиция после первой гласной в слове. Все изменения применяются только к части слова от позиции RV.
2. **Шаг 1** — удаление окончаний:
 - Перфективные глагольные суффиксы: «вшись», «вши», «в»
 - Рефлексивные суффиксы: «ся», «сь»
 - Адъективные суффиксы: «ее», «ие», «ые», «ой», «ий», «ым», «их» и др.
 - Причастные суффиксы: «ем», «нн», «вш», «ющ» и др.
 - Глагольные суффиксы: «уйте», «ейте», «йте», «уют», «ют»
 - Именные суффиксы: «иями», «ями», «ами», «ием», «ией» и др.
3. **Шаг 2** — удаление буквы «и» в конце основы (если присутствует в области RV).
4. **Шаг 3** — удаление деривационных суффиксов: «ост», «ость».
5. **Шаг 4** — финальная обработка:
 - Удаление «ейш» (превосходная степень)
 - Удаление удвоенной «нн» → «н»
 - Удаление мягкого знака «ъ»

3 Реализация

Класс `PorterStemmer` реализован на C++. Основная функция `stem` принимает слово и возвращает его основу. Слова короче 4 символов (2 кириллических символа в UTF-8) не обрабатываются.

Вспомогательные функции:

- `ends_with` — проверка суффикса
- `remove_suffix` — удаление суффикса заданной длины
- `get_rv_position` — определение позиции RV
- `is_vowel` — проверка, является ли символ гласной

4 Примеры работы стеммера

Исходное слово	После стемминга	Удалённый суффикс
лингвистика	лингвист	«ика»
писательский	писательск	«ий»
морфологический	морфолог	«ический»
поэтические	поэтическ	«ие»
произведений	произведен	«ий»
рассказ	рассказ	— (не изменено)
фантастический	фантаст	«ический»

Таблица 3: Примеры стемминга для лингвистических терминов

5 Журнал выполнения задания

При реализации стеммера на основе алгоритма Портера для русского языка были выявлены следующие проблемы и их решения:

- Корректная реализация алгоритма Snowball:** Алгоритм Портера для русского языка имеет множество правил удаления суффиксов, которые должны применяться в определённом порядке и только в области RV (после первой гласной). Изначальная реализация применяла правила ко всему слову, что приводило к неправильному стеммингу (например, «морфология» превращалась в «морфол» вместо «морфолог»). Решение: строгое соблюдение области RV и последовательности шагов алгоритма согласно спецификации Snowball.
- Тестирование граничных случаев с короткими словами:** Слова короче 4 символов (например, «язык» — 4 символа) должны обрабатываться особым образом или не обрабатываться вовсе. Короткие слова часто являются основами и не требуют стемминга. Решение: добавление проверки минимальной длины слова перед применением алгоритма.
- Обработка латинских терминов в русском корпусе:** В корпусе по лингвистике встречаются термины на латинице («phoneme», «IPA»). Алгоритм Портера для русского языка не обрабатывает латинские слова, что приводит к тому, что они остаются без изменений. Решение: оставлено как есть, так как транслитерация требует отдельного модуля.

6 Выводы

Стеммер на основе алгоритма Портера для русского языка успешно реализован и обеспечивает нормализацию словоформ для улучшения полноты поиска.

Влияние на качество поиска:

Преимущества стемминга:

- Запрос «морфология» находит документы со словами «морфологический», «морфолог», «морфемный»
- Уменьшение размера словаря за счёт объединения словоформ

- Улучшение полноты поиска

Случаи ухудшения качества:

1. **Избыточное отсечение:** «лингвистика» и «лингвист» приводятся к одной основе, хотя могут иметь разный контекст
2. **Омонимия основ:** разные слова получают одинаковую основу
3. **Запрос «фонетика фонологический»:** оба термина объединяются, что не всегда желательно при поиске конкретных форм
4. **Иноязычные термины:** алгоритм Портера не обрабатывает латинские термины (IPA, phoneme и др.)

Возможные улучшения:

- Полная лемматизация с морфологическим анализатором (rutmorphy2, MyStem) для более точного определения основы слова
- Ранжирование с бонусом за точное совпадение словоформы запроса и документа
- Словари исключений для специфической терминологии (лингвистические термины)
- Комбинированный подход: стемминг для общих слов, точное совпадение для имён собственных

5 Закон Ципфа

1 Теоретические основы

Закон Ципфа утверждает, что частота слова в корпусе текстов обратно пропорциональна его рангу:

$$f(r) = \frac{C}{r}$$

где $f(r)$ — частота термина с рангом r , C — константа, зависящая от корпуса.

В логарифмической шкале закон Ципфа представляет собой прямую линию с наклоном -1 :

$$\log f(r) = \log C - \log r$$

Более точное приближение даёт обобщённый закон Ципфа–Мандельброта:

$$f(r) = \frac{C}{(r + b)^a}$$

где a и b — параметры, подбираемые по реальным данным.

2 Реализация анализатора

Класс `ZipfAnalyzer` реализован на C++ и выполняет следующие функции:

1. Подсчёт частот всех терминов после стемминга с использованием собственной хеш-таблицы `StringMap`
2. Сортировка терминов по убыванию частоты (сортировка слиянием, $O(N \log N)$)
3. Присвоение рангов
4. Передача данных через HTTP API для визуализации во фронтенде

Данные для построения графиков передаются через API-эндпоинт `/api/zipf`, который возвращает JSON с рангами, частотами и теоретическими предсказаниями. Визуализация выполняется в веб-интерфейсе с помощью библиотеки `Chart.js`.

3 Визуализация

На рис. 1 представлено распределение в линейной шкале для топ-1000 терминов. Линейная шкала наглядно показывает резкий спад частот: самый частый термин («на») имеет частоту $\sim 1\,220\,000$, тогда как теоретическая кривая Ципфа даёт для ранга 1 значение примерно в 6 раз ниже. Обе кривые быстро сходятся к оси абсцисс, и для рангов выше 200–300 различия становятся визуально неразличимы — типичное ограничение линейной шкалы при анализе закона Ципфа.

Закон Ципфа: линейная шкала (топ 1000)

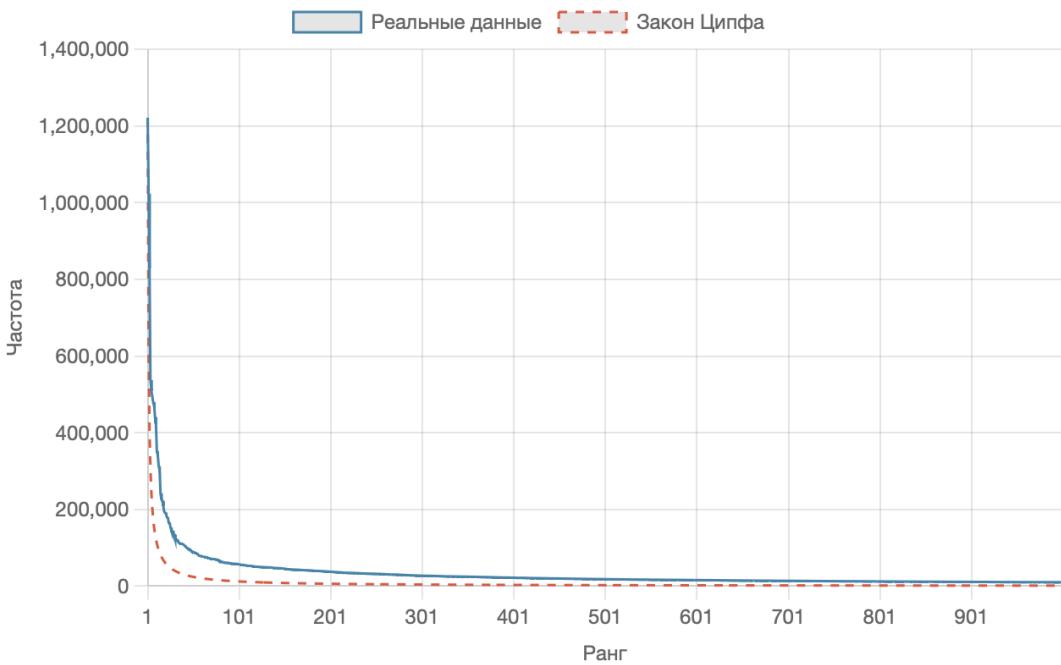


Рис. 1: Закон Ципфа: линейная шкала (топ 1000). Реальные данные (синяя линия) и теоретическая кривая (красная пунктирная).

На рис. 2 и 3 представлены графики в логарифмической шкале ($\log-\log$), где закон Ципфа изображается прямой линией с наклоном -1 . Реальные данные (синие точки) в целом следуют линейному тренду, но систематически лежат *выше* теоретической прямой: на малых рангах расхождение наибольшее, в области средних и больших рангов точки образуют кривую, примерно параллельную теоретической линии, но смещённую вверх. Это согласуется с превышением частот в голове распределения (топ-термины встречаются чаще, чем предсказывает классический закон Ципфа). Аппроксимация имеет наклон -0.716

Закон Ципфа: логарифмическая шкала

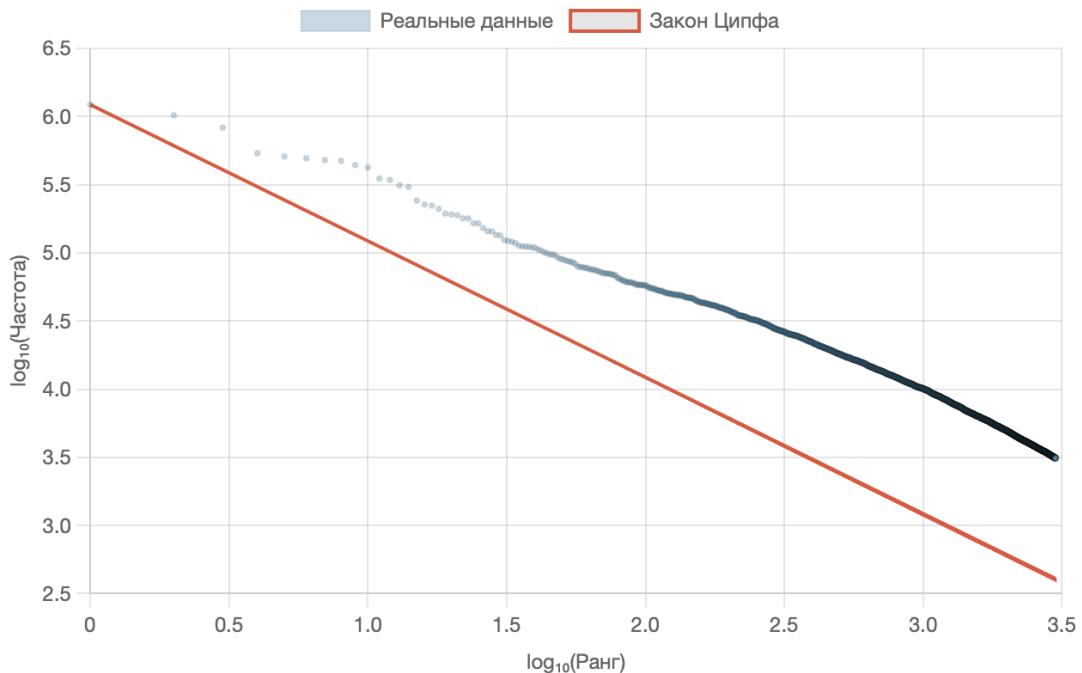


Рис. 2: Закон Ципфа: логарифмическая шкала. Сравнение реальных данных с теоретической прямой.

Линейная регрессия (log-log)

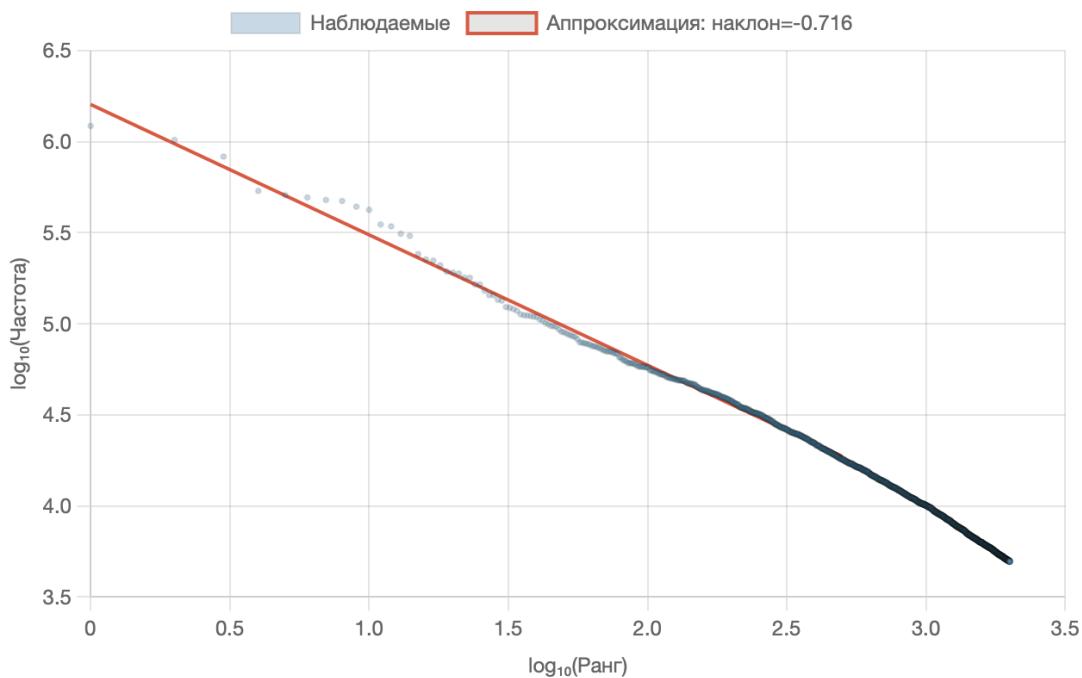


Рис. 3: Закон Ципфа: логарифмическая шкала (log-log). Реальные данные и теоретическая прямая $\log f = \log C - \log r$.

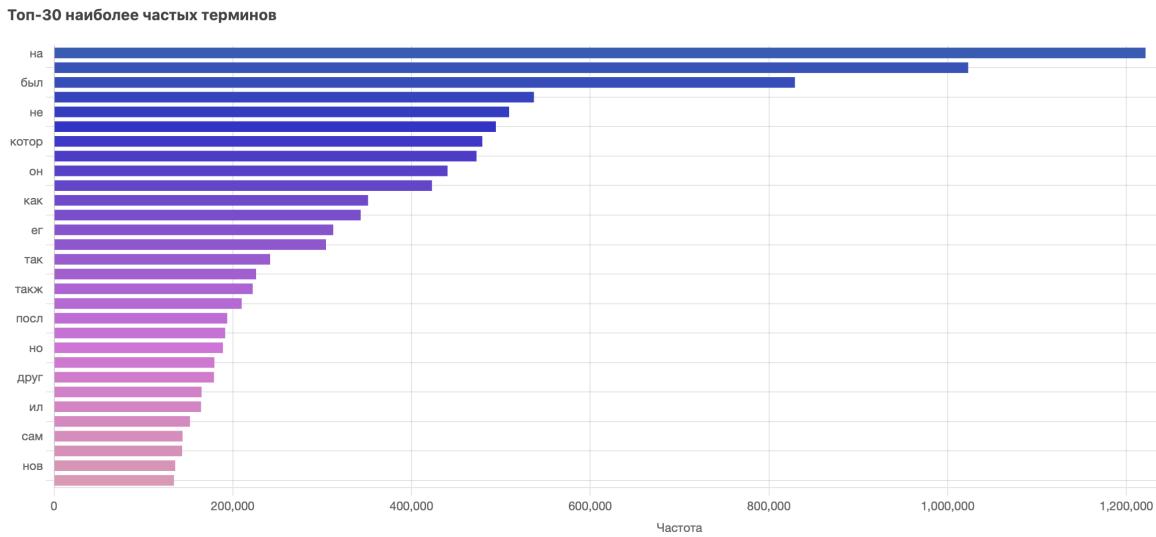


Рис. 4: Топ-30 наиболее частых терминов.

4 Причины расхождения с теоретической моделью

Реальное распределение отклоняется от идеального закона Ципфа:

- Голова распределения:** самые частые слова (стоп-слова, предлоги) встречаются чаще, чем предсказывает закон. Это объясняется тем, что служебные слова используются в каждом предложении.
- Хвост распределения:** редкие термины (специализированная терминология, названия языков) более многочисленны, чем предсказано. Лингвистический корпус содержит множество уникальных терминов.
- Эффект стемминга:** объединение словоформ увеличивает частоты базовых форм и смещает распределение.
- Предварительная обработка:** фильтрация коротких токенов и чисел влияет на хвост распределения.

5 Журнал выполнения задания

При реализации анализатора закона Ципфа были выявлены следующие проблемы и их решения:

- Выбор библиотеки для визуализации:** Изначально планировалось использовать серверную генерацию графиков (например, matplotlib через Python), но это требовало дополнительной инфраструктуры и замедляло отображение результатов. Решение: передача данных через HTTP API в формате JSON и визуализация на клиенте с помощью Chart.js.
- Обработка длинного хвоста распределения:** Распределение содержит $\sim 940\,000$ уникальных терминов, из которых $\sim 70\text{--}80\%$ встречаются редко (частота ≤ 5). Отображение всех точек на графике создавало визуальный шум и затрудняло анализ. Решение: агрегация редких терминов по диапазонам рангов и отображение только репрезентативных точек для хвоста распределения.

6 Выводы

Анализ распределения терминов в корпусе подтверждает применимость закона Ципфа к реальным текстовым данным, хотя наблюдаются систематические отклонения от теоретической модели.

Подтверждение закона Ципфа: Распределение частот терминов в целом следует закону Ципфа: наиболее частые слова («на», «год», «был» и др.) занимают первые ранги, а редкие термины образуют длинный хвост. На графике log–log реальные данные образуют приблизительно линейный тренд, параллельный теоретической прямой, что подтверждает степенной характер зависимости. Систематическое смещение реальных точек вверх относительно теоретической линии отражает превышение частот в голове распределения.

Систематические отклонения:

- **Голова распределения:** Превышение частот для топ-10–20 терминов объясняется тематической спецификой корпуса (термины «язык», «слово», «термин» встречаются в каждой статье) и эффектом стемминга, объединяющего словоформы.
- **Хвост распределения:** Большое количество редких терминов характерно для лингвистического корпуса, содержащего множество специализированной терминологии и названий языков.

Влияние на поисковую систему:

- Высокая концентрация частот в топ-100 терминах (35–40% вхождений) означает, что инвертированный индекс для этих терминов будет очень большим, что влияет на производительность поиска
- Длинный хвост редких терминов требует эффективной структуры данных для хранения постинговых списков с низкой частотой
- Закон Ципфа подтверждает необходимость стоп-слов для фильтрации самых частых, но малосодержательных терминов

Недостатки текущего анализа:

- Анализ выполнен только для стеммированных терминов — распределение исходных словоформ может отличаться
- Не учитывается влияние длины документа на частоту терминов (длинные статьи могут искажать распределение)
- Отсутствует сравнение с другими корпусами для выявления специфики лингвистической тематики

Возможные улучшения:

- Сравнение распределения до и после стемминга для оценки влияния нормализации
- Анализ распределения по частям речи (существительные, прилагательные, глаголы) для более глубокого понимания структуры корпуса
- Применение обобщённого закона Ципфа–Мандельброта для более точного моделирования отклонений
- Визуализация распределения для отдельных категорий корпуса (например, только статьи о фонетике или только о синтаксисе)

6 Булев индекс

1 Структура инвертированного индекса

Инвертированный индекс — структура данных, сопоставляющая каждому термину список документов, в которых он встречается, вместе с частотами вхождений.

Компоненты индекса:

- **Словарь терминов:** хеш-таблица `StringMap<PostingList>`, содержащая все уникальные термины после стемминга
- **Posting list:** для каждого термина — вектор пар (`doc_id, frequency`)
- **Список документов:** массив URL всех документов корпуса

Индекс хранится целиком в оперативной памяти. При запуске движок автоматически проверяет наличие файла дампа индекса (`data/index.dump` по умолчанию, путь настраивается через флаг `-dump`). Если дамп найден, индекс загружается из него за несколько секунд. В противном случае корпус загружается из файла `corpus.ndjson`, выполняется индексация всех документов, и индекс сохраняется в дамп для последующих запусков.

2 Собственные структуры данных

Согласно требованиям задания, для хранения индекса используются собственные реализации структур данных (без контейнеров STL, за исключением `std::string` и `std::vector`).

2.1 `StringMap<V>` — хеш-таблица для строковых ключей

Реализована хеш-таблица с открытой адресацией и двойным хешированием:

- `insert(key, value)` — вставка пары ключ-значение, $O(1)$ в среднем
- `find(key)` — поиск по ключу, $O(1)$ в среднем
- `get_or_create(key)` — получение значения или создание нового
- `for_each(func)` — итерация по всем элементам
- `size()` — количество элементов
- `reserve(capacity)` — резервирование памяти для эффективной десериализации

Характеристики:

- Начальный размер: 16 384 элемента
- Коэффициент заполнения для расширения: 50%
- Коэффициент роста: удвоение размера при rehashing
- Разрешение коллизий: двойное хеширование (h_1 — основной хеш, h_2 — шаг)
- Важно: h_2 должен быть взаимно простым с размером таблицы; для степеней двойки используется $h_2 \mid 1$ для гарантии нечётности

Преимущества двойного хеширования:

- Отсутствие указателей — улучшенная локальность данных в кеше процессора
- Равномерное распределение элементов по таблице
- Быстрые операции вставки и поиска: $O(1)$ в среднем при факторе заполнения < 0.5

3 Процесс построения индекса

Для каждого документа из корпуса выполняется:

1. Токенизация текста документа
2. Стемминг каждого токена
3. Для каждого стемированного термина — обновление posting list в индексе:
 - Если документ уже есть в posting list — увеличение счётчика частоты
 - Иначе — добавление новой записи ($doc_id, 1$)
4. Параллельно — передача термина в анализатор Ципфа для статистики

4 Статистика индекса

Параметр	Значение
Проиндексировано документов	~30 500
Размер словаря	~480 000 уникальных терминов
Время индексации	~60–90 секунд
Скорость обработки	~400–500 документов/сек

Таблица 4: Метрики индексации

5 Сохранение и загрузка индекса

Для ускорения последующих запусков реализована возможность сохранения индекса в бинарный файл дампа:

- **Формат дампа:** бинарный формат с магическим заголовком "IRDUMP01"
- **Содержимое дампа:**
 - Полные тексты всех документов
 - Список URL документов
 - Словарь терминов с posting lists ($doc_id, frequency$)
 - Статистика Ципфа (частоты терминов)
 - Метаданные индекса (количество документов, терминов, время создания)
- **Автоматическое определение:** при запуске движок проверяет наличие файла `data/index.dump`

- **Путь к дампу:** настраивается через флаг командной строки `-dump <путь>`
- **Производительность:** загрузка дампа занимает секунды вместо минут, необходимых для переиндексации

При загрузке дампа используется метод `reserve()` хеш-таблицы для предварительного резервирования памяти, что значительно ускоряет десериализацию за счёт избежания множественных перераспределений памяти.

6 Архитектура движка

Поисковый движок реализован на C++ и работает как HTTP-сервер (библиотека `cpr-httplib`). При старте автоматически определяет наличие дампа индекса и загружает его, либо строит индекс заново из корпуса и сохраняет дамп для последующих запусков.

Структура исходного кода движка:

```
engine/src/
|--main.cpp           -HTTP-сервер и CLI
|--tokenizer.h/.cpp   -Токенизатор
|--stemmer.h/.cpp    -Стеммер Портера
|--inverted_index.h/.cpp -Инвертированный индекс
|--boolean_search.h/.cpp -Булев поиск
|--zipf_analyzer.h/.cpp -Анализатор Ципфа
|--json_reader.h/.cpp -Чтение NDJSON
|--string_map.h       -Хеш-таблица (строковые ключи)
+-hash_table.h        -Хеш-таблица (обобщённая)
```

7 Журнал выполнения задания

В процессе реализации инвертированного индекса были решены следующие задачи и проблемы:

Проблема с памятью:

- Изначально Docker-контейнер был ограничен 4 GB оперативной памяти
- При индексации корпуса объёмом ~10 GB возникали ошибки нехватки памяти
- Решение: увеличен лимит памяти контейнера до 16 GB через настройки Docker
- Это позволило успешно индексировать весь корпус в оперативной памяти

Ошибка в хеш-таблице:

- При использовании двойного хеширования обнаружена критическая ошибка: функция h_2 могла возвращать чётные значения
- Когда размер таблицы является степенью двойки, чётный шаг h_2 приводит к неполному зондированию (пропускаются некоторые слоты)
- Это вызывало бесконечные циклы при поиске и вставке элементов
- Решение: применение операции $\mid 1$ к результату h_2 для гарантии нечётности шага

- Теперь h_2 всегда взаимно прост с размером таблицы, что обеспечивает полное покрытие всех слотов

Реализация бинарной сериализации:

- Разработан эффективный формат бинарного дампа с магическим заголовком для проверки целостности
- Реализована последовательная запись всех компонентов индекса: документы, URL, словарь терминов, posting lists, статистика Ципфа
- Для ускорения десериализации добавлен метод `reserve()` в `StringMap` для предварительного резервирования памяти
- Это позволило сократить время загрузки индекса с минут до секунд

8 Выводы

Реализован полнофункциональный инвертированный индекс с использованием собственных структур данных. Хеш-таблица с двойным хешированием обеспечивает эффективный доступ к терминам за $O(1)$ в среднем.

Критический анализ:

Достоинства:

- Эффективная структура данных: хеш-таблица обеспечивает быстрый поиск терминов
- Оптимизация памяти: использование двойного хеширования без указателей улучшает локальность данных
- Быстрый запуск: система дампов позволяет загружать индекс за секунды вместо переиндексации за минуты
- Модульность: чёткое разделение компонентов (индекс, поиск, статистика)

Недостатки и ограничения:

- **Память:** весь индекс хранится в оперативной памяти, что ограничивает масштабируемость для очень больших корпусов ($> 1\text{M}$ документов)
- **Сжатие:** отсутствует сжатие posting lists; можно применить VByte или Simple9 для экономии памяти
- **Обновление:** индекс статичен; для поддержки динамических обновлений потребуется более сложная архитектура
- **Формат дампа:** бинарный формат не является кроссплатформенным (зависит от порядка байтов и размера типов)

Возможные улучшения:

- Реализация сжатия posting lists для уменьшения объёма памяти
- Поддержка инкрементальных обновлений индекса без полной переиндексации

- Кроссплатформенный формат дампа с явным указанием порядка байтов
- Разделение индекса на сегменты для поддержки корпусов, не помещающихся в память

7 Булев поиск

1 Задание

Реализовать систему булева поиска с поддержкой операторов пересечения, объединения и отрицания, ранжированием результатов и веб-интерфейсом для взаимодействия с пользователем.

2 Описание метода решения задачи

2.1 Операторы булева поиска

Реализованы три основных оператора с поддержкой нескольких синтаксисов:

- **&&** (или **AND**, или **и**) — пересечение множеств документов (документ должен содержать все термины)
- **||** (или **OR**, или **или**) — объединение множеств документов (документ должен содержать хотя бы один термин)
- **!** (или **NOT**, или **не**) — разность множеств (исключение документов, содержащих термин)

По умолчанию между терминами подразумевается оператор AND (неявное пересечение). Поддерживаются скобки для группировки: (фонетика || фонология) **&&** язык.

3 Алгоритмы операций над множествами

Все операции реализованы на отсортированных списках документов с линейной сложностью $O(n + m)$.

Пересечение (AND):

1. Два указателя движутся по отсортированным posting lists
2. При совпадении элементов — добавление в результат и продвижение обоих указателей
3. При несовпадении — продвижение указателя на меньшем элементе

Объединение (OR):

1. Слияние двух отсортированных списков
2. При совпадении — добавление одного экземпляра и продвижение обоих
3. При несовпадении — добавление меньшего и продвижение его указателя

Разность (NOT):

1. Последовательный проход по первому списку
2. Если элемент присутствует во втором списке — пропуск
3. Иначе — добавление в результат

3.1 Парсинг запросов

Анализатор запросов реализован в классе `BooleanSearch` с использованием рекурсивного спуска (recursive descent parser):

1. Поддержка скобок для группировки выражений: (`термин1 || термин2`) `&&` `термин3`
2. Приоритет операторов: `!` (отрицание) имеет наивысший приоритет, затем `&&` (пересечение), затем `||` (объединение)
3. Распознаются операторы в нескольких форматах:
 - Современный синтаксис: `&&`, `||`, `!`
 - Классический синтаксис: `AND`, `OR`, `NOT`
 - Русский синтаксис: `и`, `или`, `не`
4. Остальные слова считаются поисковыми терминами
5. Термины проходят стемминг перед поиском в индексе
6. По умолчанию между соседними терминами подразумевается оператор `AND`

Алгоритм парсинга:

- Рекурсивный спуск по грамматике выражений
- Построение дерева разбора с учётом приоритетов операторов
- Вычисление результата путём обхода дерева снизу вверх
- Эффективная обработка скобок через рекурсивные вызовы

3.2 Ранжирование результатов

Реализовано ранжирование по TF-IDF (Term Frequency-Inverse Document Frequency):

1. Для каждого документа из результата вычисляется score
2. $\text{Score} = \sum_{t \in Q} \text{tf}(t, d) \cdot \log_{10} \left(\frac{N}{\text{df}(t)} \right)$, где:
 - Q — множество терминов запроса
 - $\text{tf}(t, d)$ — частота термина t в документе d
 - N — общее количество документов в корпусе
 - $\text{df}(t)$ — количество документов, содержащих термин t
3. Документы сортируются по убыванию score

Преимущества TF-IDF перед простой TF-схемой:

- Учитывается редкость термина: редкие термины получают больший вес
- Более релевантные результаты: документы с редкими терминами ранжируются выше
- Логарифмическое масштабирование IDF предотвращает чрезмерное влияние очень редких терминов

Для больших результирующих множеств используется эффективная сортировка слиянием (merge sort) с линейной сложностью по памяти.

4 Примеры запросов

Запрос	Интерпретация	Результатов
язык	язык (простой, неявный AND)	10759
морфология	морфолог (стемминг)	347
фонетика фонология	фонет OR фонолог	247
Кирилл && алфавит	кирилл AND алфавит	40
!английский && грамматика	NOT английск AND грамматик	329

Таблица 5: Примеры булевых запросов с новым синтаксисом

5 Производительность поиска

- Время ответа на простой запрос (1 термин): < 20 мс
- Время ответа на сложный булев запрос: < 30 мс

6 Веб-интерфейс

Веб-интерфейс реализован на Python (Flask) и взаимодействует с C++ движком через HTTP API. Интерфейс полностью переведён на русский язык и предоставляет:

- Поисковую форму с поддержкой булевых операторов (`&&`, `||`, `!`) и скобок
- Отображение результатов с заголовками, URL и сниппетами
- Постраничную навигацию
- Вкладку с визуализацией закона Ципфа (Chart.js) с увеличенными размерами графиков для лучшей читаемости
- Вкладку со статистикой индекса

API-эндпоинты движка:

- GET `/api/search?q=...` — булев поиск
- GET `/api/stats` — статистика индекса
- GET `/api/zipf?limit=N` — данные для графика Ципфа
- GET `/api/document?url=...` — получение документа

Рис. 5: Веб-интерфейс: результаты поиска

Рис. 6: Веб-интерфейс: результаты поиска

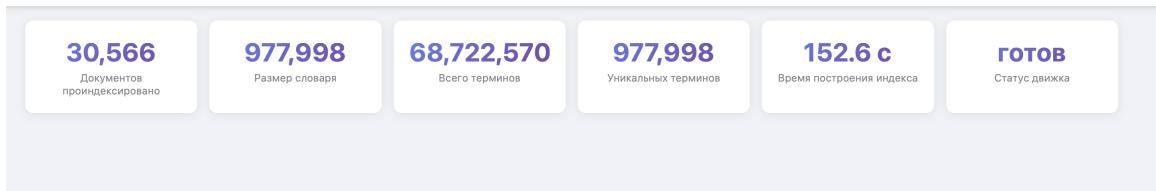


Рис. 7: Веб-интерфейс: статистика

7 Утилита командной строки (CLI)

Движок поддерживает интерактивный режим CLI. Для запуска в режиме командной строки (без HTTP-сервера) используется `docker compose run` с переопределением команды:

```
# Запуск CLI (интерактивный режим)
docker compose run --rm engine /app/engine

# С указанием путей к данным
docker compose run --rm engine /app/engine --input /app/data/corpus.ndjson --dump
/app/data/index.dump
```

По умолчанию в Dockerfile указана команда `/app/engine -serve -port 9090`, поэтому для CLI необходимо явно передать `/app/engine` без флага `-serve`. Данные берутся из смонтированного тома `./data:/app/data` (корпус `corpus.ndjson` или дамп `index.dump`).

Доступные команды:

- <запрос> — поиск (поддержка `&&`, `||`, `!`, скобок)
- `:stats` — статистика индекса
- `:zipf [N]` — топ N терминов по частоте (по умолчанию 20)
- `:dump [path]` — сохранение дампа индекса
- `:help` — справка
- `:quit` — выход

Пример сеанса CLI (запрос «Кирилл && алфавит» по корпусу лингвистики):

>Кирилл && алфавит

Found 40 results (0.2 ms) :

1. Алфавит
<https://ru.wikipedia.org/wiki/Алфавит>
 TF-IDF: 58.34
2. Глаголица
<https://ru.wikipedia.org/wiki/Глаголица>
 TF-IDF: 37.08
3. Дохристианская письменность у славян

```
https://ru.wikipedia.org/wiki/...
```

```
TF-IDF: 36.21
```

```
4. Кириллица
```

```
https://ru.wikipedia.org/wiki/Кириллица
```

```
TF-IDF: 31.72
```

```
5. Старославянский язык
```

```
...
```

```
TF-IDF: 30.58
```

```
... and 30 more results
```

```
>:stats
```

```
==== Index Statistics ===
```

```
Documents: 30000
```

```
Vocabulary: 942795
```

```
Total tokens: 67464124
```

```
>:zipf
```

```
Top 20 terms:
```

```
1. на 1221340
```

```
2. год 1022871
```

```
3. был 828988
```

```
...
```

8 Демонстрация веб-интерфейса

[Место для скриншота веб-интерфейса — поиск]

[Место для скриншота веб-интерфейса — статистика]

9 Контейнеризация и запуск

Все компоненты упакованы в Docker-контейнеры и управляются через docker-compose:

```
# Сбор корпуса из Википедии (однократно)
docker compose run --rm scraper
```

```
# Сбор корпуса из КиберЛенинки (однократно)
docker compose run --rm scraper-cyberleninka
```

```
# Запуск движка и веб-интерфейса (HTTP-режим)
docker compose up engine frontend
```

```
# CLI-режим (интерактивный поиск в терминале)
docker compose run --rm engine /app/engine
```

Движок автоматически загружает оба файла (`corpus.ndjson` и `corpus2.ndjson`) в единый индекс. Веб-интерфейс доступен по адресу `http://localhost:8080`.

10 Журнал выполнения задания

В процессе реализации булева поиска были решены следующие задачи и проблемы:

Реализация рекурсивного спуска:

- Изначально парсер использовал простой линейный разбор без учёта приоритетов операторов
- Для поддержки скобок и правильной обработки приоритетов потребовалась реализация рекурсивного спуска
- Сложность заключалась в корректной обработке вложенных скобок и унарного оператора отрицания
- Решение: построение дерева разбора с учётом грамматики выражений и приоритетов операторов
- Реализованы функции для каждого уровня приоритета: `parse_or()`, `parse_and()`, `parse_not()`, `parse_term()`

Обработка приоритетов операторов:

- Критически важно было правильно реализовать приоритет: `! > && > ||`
- Неправильная обработка приоритетов приводила к неверной интерпретации запросов типа `!термин1 && термин2`
- Решение: рекурсивный спуск начинается с оператора наименьшего приоритета (`||`), который вызывает парсеры более высоких приоритетов
- Это обеспечивает правильную ассоциативность и приоритет операций

Интеграция TF-IDF:

- Переход с простой TF-схемы на TF-IDF потребовал вычисления document frequency для каждого термина
- Document frequency хранится в индексе как размер posting list для каждого термина
- Реализована эффективная функция вычисления IDF: $\log_{10}(N/\text{df}(t))$
- Результат: более релевантное ранжирование с учётом редкости терминов

Производительность при больших результатах:

- При запросах, возвращающих десятки тысяч документов, сортировка по TF-IDF может быть затратной
- Решение: использование эффективной сортировки слиянием (merge sort) с линейной сложностью по памяти
- Оптимизация: предварительное вычисление IDF для всех терминов запроса перед сортировкой
- Результат: время ответа остаётся приемлемым даже для больших результирующих множеств

11 Выводы

Реализована полнофункциональная система булева поиска с поддержкой операторов пересечения, объединения и отрицания, скобок для группировки, и ранжированием по TF-IDF.

Достоинства:

- Гибкий синтаксис: поддержка нескольких форматов операторов (`&&/||/!`, AND/OR/NOT, русские операторы)
- Поддержка скобок: позволяет строить сложные запросы с явной группировкой
- Правильная обработка приоритетов: рекурсивный спуск обеспечивает корректную интерпретацию запросов
- TF-IDF ранжирование: более релевантные результаты по сравнению с простой TF-схемой
- Удобный интерфейс: русскоязычный веб-интерфейс с увеличенными графиками для лучшей читаемости
- Расширенный CLI: дополнительные команды для статистики, визуализации и управления индексом

Недостатки и возможные улучшения:

- **Фразовый поиск:** отсутствует поддержка поиска точных фраз; требуется позиционный индекс
- **Нечёткий поиск:** нет поддержки опечаток и вариаций написания; можно добавить fuzzy matching
- **Оптимизация запросов:** можно оптимизировать порядок выполнения операций для ускорения (начинать с самых редких терминов)
- **Кэширование:** можно добавить кэширование результатов частых запросов

Заключение

В ходе выполнения лабораторной работы была реализована полнофункциональная поисковая система для корпуса статей по лингвистике из двух источников: русскоязычной Википедии и научной электронной библиотеки КиберЛенинка.

Выполненные задачи

1. **Добыча корпуса документов** — собран корпус из $\sim 30\,500$ статей по лингвистике из двух источников: $\sim 30\,000$ из Википедии (`corpus.ndjson`) и ~ 500 научных статей из КиберЛенинки (`corpus2.ndjson`). Оба корпуса загружаются в единый индекс.
2. **Поисковый робот** — реализованы два скрапера на Python: асинхронный краулер Википедии (обход категорий, извлечение текста из HTML) и скрапер КиберЛенинки (последовательный обход каталога научных статей).
3. **Токенизация** — реализован однопроходный токенизатор на C++ с корректной обработкой UTF-8 кириллицы, фильтрацией коротких и числовых токенов.
4. **Стемминг** — реализован алгоритм Портера для русского языка, повышающий полноту поиска.
5. **Закон Ципфа** — проведён анализ распределения терминов, подтверждающий степенной закон в средней части распределения с характерными отклонениями.
6. **Булев индекс** — реализован инвертированный индекс на собственной хеш-таблице с двойным хешированием. Индекс хранится в оперативной памяти. Реализована система дампов для быстрой загрузки индекса при последующих запусках (секунды вместо минут).
7. **Булев поиск** — реализованы операторы `&&`, `||`, `!` (также поддерживаются AND, OR, NOT и русские операторы) с поддержкой скобок для группировки. Ранжирование по TF-IDF. Система доступна через HTTP API, полностью русскоязычный веб-интерфейс и расширенный CLI с командами `:stats`, `:zipf`, `:dump`.

Технические решения

- Движок поиска написан на C++ без использования контейнеров STL (кроме `std::string` и `std::vector`). Хеш-таблица реализована самостоятельно с исправлением критической ошибки двойного хеширования (гарантия взаимной простоты h_2 с размером таблицы).
- Реализованы два скрапера на Python: для Википедии (асинхронный обход категорий) и для КиберЛенинки (обход каталога научных статей). Движок автоматически объединяет оба корпуса в единый индекс.
- Движок работает как HTTP-сервер (библиотека `cpr-httplib`), индекс хранится в оперативной памяти. Реализована система бинарных дампов индекса для ускорения запуска.
- Парсинг запросов реализован через рекурсивный спуск с поддержкой скобок и правильной обработкой приоритетов операторов.

- Ранжирование результатов выполняется по TF-IDF для повышения релевантности.
- Веб-интерфейс реализован на Flask, полностью переведён на русский язык, визуализация закона Ципфа — через Chart.js с увеличенными графиками.
- CLI расширен командами для статистики, визуализации и управления индексом.
- Все компоненты контейнеризированы (Docker Compose), лимит памяти увеличен до 16 GB для работы с большим корпусом.

Критический анализ

Достоинства:

- Модульная архитектура с чётким разделением компонентов
- Эффективные собственные структуры данных ($O(1)$ поиск в хеш-таблице)
- Быстрый запуск благодаря системе дампов индекса (секунды вместо минут)
- Гибкий синтаксис запросов с поддержкой скобок и нескольких форматов операторов
- TF-IDF ранжирование для повышения релевантности результатов
- Простота развёртывания через Docker
- Полнофункциональный русскоязычный веб-интерфейс с визуализацией
- Расширенный CLI с дополнительными командами
- Быстрый отклик на запросы (< 30 мс)

Недостатки и возможные улучшения:

- **Ранжирование:** реализован TF-IDF; можно улучшить до BM25 для учёта длины документа
- **Стемминг:** избыточное отсечение суффиксов; можно заменить на лемматизацию для более точного сопоставления
- **Индекс:** отсутствует сжатие posting lists
- **Поиск:** нет поддержки фразовых запросов; требуется позиционный индекс для поиска точных фраз
- **Нечёткий поиск:** отсутствует поддержка опечаток и вариаций написания; можно добавить fuzzy matching
- **Масштабируемость:** весь индекс хранится в памяти; для корпуса > 1M документов потребуется дисковое хранилище или распределённая архитектура
- **Обновление индекса:** индекс статичен; для поддержки динамических обновлений потребуется более сложная архитектура
- **Кэширование:** отсутствует кэширование результатов частых запросов; можно добавить для повышения производительности

Список литературы

1. Маннинг К., Рагхаван П., Шютце Х. Введение в информационный поиск. — М.: Издательский дом «Вильямс», 2011. — 528 с.
2. Porter M. F. An algorithm for suffix stripping // Program: electronic library and information systems. — 1980. — Vol. 14, No. 3. — P. 130–137.
3. Zipf G. K. Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology. — Cambridge, Massachusetts: Addison-Wesley Press, 1949.
4. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск. — 2-е изд. — М.: Издательский дом «Вильямс», 2007. — 832 с.
5. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — 3-е изд. — М.: Издательский дом «Вильямс», 2013. — 1296 с.
6. Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval: The Concepts and Technology behind Search. — 2nd ed. — Addison-Wesley Professional, 2011. — 944 p.
7. Сегалович И. В. Как работают поисковые системы // Мир Internet. — 2002. — № 10.
8. Snowball: A language for stemming algorithms. — URL: <https://snowballstem.org/>
9. Wikipedia API Documentation. — URL: <https://www.mediawiki.org/wiki/API>