Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: Е. С. Кострюков
Преподаватель: А. А. Кухтичев
Группа: М8О-407Б-22
Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №1 «Добыча корпуса документов»

Необходимо подготовить корпус документов, который будет использован при выполнении остальных лабораторных работ:

- Скачать его к себе на компьютер. В отчёте нужно указать источник данных.
- Ознакомиться с ним, изучить его характеристики. Из чего состоит текст? Есть ли дополнительная мета-информация? Если разметка текста, какая она?
- Разбить на документы.
- Выделить текст.
- Найти существующие поисковики, которые уже можно использовать для поиска по выбранному набору документов (встроенный поиск Википедии, поиск Google с использованием ограничений на URL или на сайт). Если такого поиска найти невозможно, то использовать корпус для выполнения лабораторных работ нельзя!
- Привести несколько примеров запросов к существующим поисковикам, указать недостатки в полученной поисковой выдаче.

В результатах работы должна быть указаны статистическая информация о корпусе:

- Размер «сырых» данных.
- Количество документов.
- Размер текста, выделенного из «сырых» данных.
- Средний размер документа, средний объём текста в документе.

1 Описание

Цель работы: Подготовить и проанализировать корпус текстовых документов для дальнейшего использования в задачах информационного поиска (индексация, ранжирование, кластеризация).

1 Источники данных и сбор корпуса

Для формирования корпуса были использованы два независимых источника, обеспечивающих разнообразие лексики и структуры текстов:

1. **Habr.com** (технические статьи).
   - **Тематика:** программирование, машинное обучение, DevOps, кибербезопасность.
   - **Метод сбора:** парсинг через разделы статей, тематические хабы и списки топовых публикаций.
2. **Lenta.ru** (новостной портал).
   - **Тематика:** политика, экономика, технологии, общество.
   - **Метод сбора:** парсинг RSS-лент и архива новостей за период 2021–2025 гг.

Итоговый объём: собрано 39 048 документов.

Ссылка на архив с корпусом: `https://drive.google.com/file/d/1ELYTGEdc0JcjnWlG71hd5vZaRbU`
`view?usp=drive_link`

2 Характеристика корпуса и предобработка

В ходе работы производилось скачивание полных HTML-версий страниц с последующим извлечением чистого текста.

- **Структура сырых данных:** полные HTML-страницы, включающие разметку, стили, скрипты и навигационные элементы.
- **Извлечённый текст:** очищен от HTML-тегов, скриптов и стилей; сохранена структура параграфов. Для каждого документа сформирован текстовый файл следующего формата:
  - Мета-блок: заголовок, автор (для Habr), дата публикации, URL-источник.

  - Тело документа: основной текст статьи.

- **Мета-информация:** для каждого документа сформирован JSON-объект, содержащий уникальный ID, дату в формате ISO 8601, ссылку на источник и размер файлов (сырого и чистого).

3 Статистическая информация о корпусе

Проведён анализ собранных данных, результаты представлены в таблице ниже.

Метрика	**Значение**
Всего документов	39 048
Размер «сырых» данных (HTML)	5 324.64 МБ
Размер выделенного текста	335.01 МБ
Средний размер документа (HTML)	139.63 КБ
Средний объём текста в документе	8.79 КБ

Сравнение источников. Статьи с Habr.com в среднем в 5.6 раз длиннее новостей Lenta.ru (14.66 КБ против 2.63 КБ текста), что создаёт необходимую для тестирования вариативность длины документов.

4 Анализ существующих поисковых систем

Был проведён анализ возможностей поиска по выбранным источникам с целью выявления недостатков.

1. **Встроенный поиск Habr.com**
   - *Пример запроса:* «машинное обучение Python»
   - *Недостатки:* в выдачу попадает много нерелевантных статей, где термины упоминаются вскользь; старые статьи (2015–2017 гг.) часто ранжируются выше актуальных; отсутствует семантический поиск (синонимы не учитываются).
2. **Встроенный поиск Lenta.ru**
   - *Пример запроса:* «выборы президента США»
   - *Недостатки:* отсутствует группировка похожих новостей (дубликаты в выдаче); слабая фильтрация по временным периодам; контекст запроса часто игнорируется (смешиваются новости разных лет).
3. **Внешние поисковики (Google/Yandex с оператором site:)**

- *Недостатки:* индексация новых статей отстаёт на несколько дней; в выдачу попадают служебные страницы; невозможно использовать внутренние метаданные сайта (рейтинг, хабы) для ранжирования.

2 Исходный код

Весь исходный код проекта организован в виде модульной структуры и доступен в репозитории. Реализация выполнена на языке Python с использованием библиотек `requests` (для HTTP-запросов) и `BeautifulSoup` (для парсинга HTML). Для ускорения процесса сбора данных применена многопоточность (`concurrent.futures`).

1 Структура проекта

- `parsers/` — модуль, содержащий логику скачивания и обработки страниц.
  - `habr_parser.py` — класс для работы с Habr.com.
  - `lenta_parser.py` — класс для работы с Lenta.ru (RSS + архив).
- `corpus/` — директория для хранения данных (разделена на `raw` для HTML и `text` для очищенных данных).
- `stats/` — скрипты для подсчёта статистики по собранному корпусу.
- `run_all.bat` — сценарий автоматизации полного цикла работы.

2 Основные алгоритмы

Сбор ссылок (Crawling)
Для обеспечения репрезентативности выборки реализованы различные стратегии обхода.
Для **Habr.com** используется комбинированный подход: обход страниц «Все статьи», сбор «ТОП» и проход по популярным хабам.
Для **Lenta.ru** используется обход архива по датам за период 2021–2025.

Фрагмент из lenta_parser.py (генерация ссылок через архив дат)

```
while len(urls) < self.max_articles and days_back < max_days:
date = current_date - timedelta(days=days_back)
archive_url = f'https://lenta.ru/{date.strftime("%Y/%m/%d")}/'
# ... request archive and extract links ...
```

Парсинг и очистка (Processing)
Ключевым методом в обоих парсерах является `parse_article`: он загружает HTML, создаёт объект `BeautifulSoup` и извлекает основной контент, отсекая навигацию и рекламу.

Фрагмент (извлечение метаданных и очистка)

```
soup = BeautifulSoup(response.text, 'lxml')

# Title and date
title = soup.find('h1').get_text(strip=True)
date = soup.find('time').get('datetime')

# Main content block
article_body = soup.find('div', class_='topic-body__content')
if not article_body:
article_body = soup.find('div', itemprop='articleBody')

# Plain text
text = article_body.get_text(separator='\n', strip=True)
```

Многопоточная загрузка

Для ускорения I/O-операций используется `ThreadPoolExecutor`, что позволяет скачивать несколько страниц одновременно.

Фрагмент из habr_parser.py

```
with ThreadPoolExecutor(max_workers=num_workers) as executor:
futures = {executor.submit(self.parse_article, url): url for url in new_urls}
for future in tqdm(as_completed(futures), total=len(new_urls)):
# ... handle results ...
```

3 Формат хранения данных

Результат работы сохраняется в двух видах:

1. **.html файлы** — полная копия страницы (для верификации).
2. **.txt файлы** — структурированный текст с заголовками метаданных:

```
Title: Заголовок статьи
Author: Имя Автора
Date: 2024-01-15T10:00:00+03:00
URL: https://habr.com/ru/articles/123456/

[Текст статьи...]
```

3 Выводы

Выполнив первую лабораторную работу, я научился собирать корпус нужного размера из нескольких разнородных источников и приводить данные к унифицированному виду, удобному для последующей индексации.

Я понял разницу между «сырыми» документами (HTML) и выделенным текстом и на практике отработал методы очистки (удаление тегов, скриптов, стилей) с сохранением смысловой структуры документа.

Также я научился проверять пригодность корпуса через анализ существующих поисковых систем и оценивать типичные проблемы их выдачи (информационный шум, отсутствие семантической связи, смешивание контекстов). Собранный корпус объёмом около 40 000 документов готов для выполнения следующих лабораторных работ.

Лабораторная работа №2 «Поисковый робот»

Необходимо написать парсер на любом языке программирования.

- Написать поисковый робот — компоненты обкачки документов, используя любой язык программирования;
- Единственным аргументом поисковому роботу подаётся путь до yaml-конфига, содержащий:
  - Данные для базы данные в секции db;
  - Данные для робота в секции logic: задержка между обкачкой страницы;
  - Любые другие данные, необходимые для реализовации логики поискового робота.
- Сохранять в базе данных (например, MongoDB) документы со следующими полями:
  - url, нормализованный;
  - «сырой» html-текст документа;
  - название источника;
  - Дата обкачки документа в формате Unix time stamp.
- Поисковый робот можно остановить в любой момент и при повторном запуске робот должен начать с того документа, с которого он остановился;
- Периодически он должен уметь переобкачивать документы, которые уже есть в базе, но только в том случае, если они изменились.

1 Описание

Цель работы: Разработать автоматизированный поисковый робот (crawler) для сбора и обновления коллекции документов с сохранением данных в NoSQL базу данных. Робот должен поддерживать остановку/возобновление работы, переобход измененных страниц и масштабирование базы данных за счет импорта ранее собранных корпусов.

1 Архитектура решения

Робот реализован на языке Python. В качестве хранилища данных выбрана **MongoDB**, так как она подходит для хранения неструктурированных данных (JSON-подобных документов) и обеспечивает высокую скорость записи.

Основные компоненты системы:

1. **Менеджер конфигурации:** считывает параметры из файла `config.yaml` (параметры подключения к БД, задержки, стартовые точки, таймауты).
2. **Модуль загрузки (Downloader):** использует библиотеку `requests` с ротацией User-Agent для имитации поведения реального пользователя.
3. **Парсер и нормализатор:**
   - Приводит URL к каноническому виду (нижний регистр, удаление якорей #, параметров сессии и слешей в конце).
   - Вычисляет MD5-хеш контента для отслеживания изменений.
4. **Менеджер очереди:** реализует персистентную очередь. Состояние очереди периодически сбрасывается в БД, что позволяет продолжить обход после перезапуска с того же места.
5. **Storage Layer (MongoDB):**
   - Коллекция `documents`: хранит сами данные (HTML, метаданные, хеши).
   - Коллекция `crawl_queue`: хранит очередь ссылок для посещения.

2 Логика работы

Алгоритм краулинга:

1. **Запуск:** робот проверяет наличие сохраненного состояния. Если оно есть — загружает очередь URL, если нет — берет seed-url из конфига.

2. **Обход:**
    - Извлекает URL из очереди.
    - Скачивает HTML.
    - Считает хеш контента (`content_hash`).
    - Проверяет в БД:
        - Если документ существует и хеш совпадает — обновляет поле `crawled_at` (документ свежий).
        - Если хеш отличается — обновляет тело документа и хеш.
        - Если документа нет — создает новую запись.
3. **Остановка:** при получении сигнала остановки (Graceful Shutdown) или по достижении лимита текущее состояние очереди сохраняется в БД.

Миграция данных.
Для обеспечения требований к объему корпуса (30 000+ документов) был разработан дополнительный модуль `migration.py`. Он импортирует статический корпус, собранный в ЛР №1, в базу данных робота, приводя данные к единому формату (вычисление хешей, нормализация URL).

3 Результаты работы

Было проведено тестирование робота и последующее наполнение базы данных.

1. Тестовый запуск робота:

- Задержка (`delay`): 0.3 сек.
- Объем тестовой выборки: 3000 документов.
- Результат: робот успешно собрал новые данные и обновил существующие.

2. Миграция корпуса.
В базу данных были успешно импортированы документы с ресурсов Habr.com и Lenta.ru.

- **Всего документов в БД:** 41 684
- **Ошибок при миграции:** 1 (некорректный файл).

Структура документа в MongoDB:

```
{
 "_id": ObjectId("..."),
 "url": "https://habr.com/ru/articles/978862",
 "source": "habr",
 "html": "<html>...</html>",
 "crawled_at": 1734778509,
 "content_hash": "5d41402abc4b2a76b9719d911017c592",
 "size": 145678
}
```

2 Исходный код

Ниже приведены ключевые фрагменты реализации. Полный код, включая скрипт миграции и автотесты, находится в репозитории.

1 Конфигурация (config.yaml)

```
db:
host: localhost
port: 27017
database: search_engine
collection: documents

logic:
delay_between_requests: 0.3
max_documents_per_run: 3000
recrawl_period_days: 30

sources:
-name: habr
base_url: https://habr.com
start_urls:
-https://habr.com/ru/articles/
```

2 Основной класс робота (crawler.py)

```
class SearchCrawler:

def crawl(self):
"""Main crawling loop"""
self.load_queue_from_db()

while self.queue and processed < max_docs:
item = self.queue.pop(0)
url = item['url']

# Check if recrawl is needed
existing = self.collection.find_one({'url': url})
if existing and not self._should_recrawl(existing):
continue

html = self.fetch_page(url)
if html:
```

```
self.save_document(url, html, item['source'])

# Extract links and extend queue
links = self.extract_links(url, html, item['source'])
for link in links:
self.add_to_queue(link, item['source'])

def save_document(self, url: str, html: str, source: str) -> bool:
"""Save with content hash check (content deduplication)"""
content_hash = self._calculate_content_hash(html)

# Upsert logic (insert or update)
# ...
```

3 Скрипт миграции (migration.py)

```
def migrate():
"""Import data from file-based corpus into MongoDB"""

# ... connect to DB ...

for item in metadata:
# Read HTML file from raw folder
with open(html_path, 'r') as hf:
html_content = hf.read()

# Convert to crawler format
url = normalize_url(item['url'])
content_hash = calculate_hash(html_content)

# Bulk insert for speed
operations.append(
UpdateOne({'url': url}, {'$set': doc}, upsert=True)
)
```

3 Выводы

В ходе выполнения лабораторной работы №2 я реализовал полноценный поисковый робот и сформировал базу данных, достаточную для выполнения всех последующих заданий курса.

1. **Навыки разработки:** мною был написан устойчивый к ошибкам crawler на Python. Я освоил работу с NoSQL базой данных MongoDB, включая создание индексов для ускорения поиска по URL и хешу.
2. **Оптимизация хранения:** реализован механизм вычисления `content_hash` (MD5), который позволяет избегать дублирования данных и обновлять в базе только реально изменившиеся страницы, экономя дисковое пространство и время записи.
3. **Работа с Legacy-данными:** важным этапом стала разработка скрипта миграции. Это позволило объединить результаты первой лабораторной работы (статический корпус) с возможностями робота.

Лабораторная работа №3 «Токенизация»

Нужно реализовать процесс разбиения текстов документов на токены, который потом будет использоваться при индексации. Для этого потребуется выработать правила, по которым текст делится на токены. Необходимо описать их в отчёте, указать достоинства и недостатки выбранного метода. Привести примеры токенов, которые были выделены неудачно, объяснить, как можно было бы поправить правила, чтобы исправить найденные проблемы.

В результатах выполнения работы нужно указать следующие статистические данные:

- Количество токенов.
- Среднюю длину токена.

Кроме того, нужно привести время выполнения программы, указать зависимость времени от объёма входных данных. Указать скорость токенизации в расчёте на килобайт входного текста. Является ли эта скорость оптимальной? Как её можно ускорить?

1 Описание

Цель работы: Реализовать эффективный алгоритм разбиения текстов документов на отдельные лексемы (токены) и подготовить данные для построения поискового индекса.

1 Правила токенизации

В ходе работы был реализован токенизатор на языке **C++**, интегрированный в основной проект на Python. Выбранный подход обеспечивает высокую производительность за счет нативного выполнения кода.

Выработанные правила:

1. **Алфавит:** токеном считается последовательность символов, состоящая из букв (кириллица, латиница) и цифр.
2. **Разделители:** пробельные символы, знаки препинания (точки, запятые, скобки и т.д.) считаются разделителями и игнорируются.
3. **Сложные слова:** дефис (-) и апостроф (') считаются частью токена только если они находятся *внутри* слова (например, it-индустрия). Дефисы в начале или конце слова отбрасываются.
4. **Нормализация:** все буквы приводятся к нижнему регистру (A–Z → a–z, А–Я → а–я, Ё → ё).
5. **Кодировка:** полная поддержка UTF-8 с корректной обработкой многобайтовых символов кириллицы.

Примеры применения правил:

- Full-stack разработчик → full-stack, разработчик
- IT-индустрия 2025 → it-индустрия, 2025
- Привет, мир! → привет, мир

2 Статистические данные

Анализ был проведен на полном корпусе документов (Habr + Lenta), собранном в предыдущих лабораторных работах.

Метрика	Значение
Количество документов	41 684
Общий объем обработанного текста	∼417 МБ (входные данные)
Общее количество токенов	59 300 082
Количество уникальных токенов	1 081 897
Средняя длина токена	5.83 символа

Примечание: средняя длина токена (5.83) является характерной для русскоязычных текстов публицистического и технического стиля.

Топ-10 частых токенов:

- в: 1 628 439
- и: 1 411 364
- на: 809 340
- не: 529 180
- с: 525 239
- для: 506 586
- что: 404 500
- а: 330 068
- по: 286 643
- как: 285 358

3 Производительность и скорость

Измерения времени выполнения показали следующие результаты:

- **Время токенизации (C++ core):** 5.38 сек.
- **Полное время (включая чтение из БД):** 109.68 сек.
- **Скорость токенизации:** ∼130 512 КБ/сек (∼127 МБ/сек).

Зависимость от объема данных.
Алгоритм является однопроходным, сложность составляет O(N), где N — количество символов во входном тексте. Время выполнения растет линейно с увеличением объема данных.

Оценка оптимальности.
Скорость в ∼127 МБ/сек для однопоточного приложения является близкой к оптимальной (ограничена скоростью работы с памятью и аллокациями строк).

Пути ускорения:

1. **Многопоточность:** распараллеливание обработки документов (MapReduce подход) позволит утилизировать все ядра CPU.
2. **SIMD-инструкции:** использование векторных инструкций для проверки символов и смены регистра.
3. **Батчинг:** чтение данных из БД более крупными блоками для снижения накладных расходов на I/O.

4 Проблемные токены

1. Числа (129 уникальных)
Примеры: 3366, 52, 2024, 05, 500
Часто бесполезны для текстового поиска, нужна фильтрация или отдельная индексация.

2. Короткие токены $\leq$ 2 символов (164 уникальных)
Примеры: ии, ту, но, нг, 4k, 1k
Среди них — служебные слова и шум. Решение: `min_token_length = 3`.

3. Смешение алфавитов (10 уникальных)
Примеры: ai-чатботы, it-индустрии, vps-хостинг, l-тирозин
Не всегда ошибка (термины, бренды), но усложняет аналитику.

5 Возможные улучшения

1. Фильтрация чисел:
Отбрасывать токены или выделять в отдельный канал.

2. Минимальная длина = 3:
Убрать «в», «и», «с», «на» (либо стоп-слова).

3. Ограничение `max_token_length = 25`:
Отсечь «мусор» из склеенной разметки.

4. Разделение дефисов (опционально):
it-индустрии → it, индустрии для более тонкой морфологии.

2 Исходный код

Проект построен по гибридной архитектуре, объединяющей производительность низкоуровневого языка и удобство скриптового управления. Ядро токенизатора написано на C++ и скомпилировано в динамическую библиотеку (`.dll / .so`), которая загружается в Python через модуль `ctypes`.

1 Структура реализации

Файловая организация проекта:

- `cpp/tokenizer.cpp` — реализация класса Tokenizer; содержит основную логику конечного автомата для разбора UTF-8 строк.
- `cpp/tokenizer_lib.cpp` — экспорт функций в стиле C (`extern "C"`) для обеспечения бинарной совместимости с Python.
- `python/tokenizer_wrapper.py` — Python-обертка, инкапсулирующая работу с памятью и вызовы C-функций.
- `main.py` — скрипт-оркестратор: выгружает данные из MongoDB, передает их в токенизатор и сохраняет результаты.

2 Ключевые алгоритмы

1. Проверка символов (is_valid_token_char).
Одной из ключевых задач было корректное определение допустимых символов в UTF-8. Была реализована собственная легковесная проверка диапазонов Unicode кодов.

Example from cpp/tokenizer.cpp

```
bool is_valid_token_char(uint32_t cp) {
  // Latin letters (a-z, A-Z) and digits
  if ((cp >= 'a' && cp <= 'z') || (cp >= 'A' && cp <= 'Z') || (cp >= '0' && cp <= '9'
      ))
  return true;

  // Cyrillic (basic range + Yo)
  if ((cp >= 0x0410 && cp <= 0x044F) || cp == 0x0401 || cp == 0x0451)
  return true;

  return false;
}
```

2. Конечный автомат разбора.
Токенизация происходит за один проход по строке. Алгоритм накапливает символы в буфер `current_token`, пока встречает допустимые символы или внутрисловные разделители (дефис). При встрече разделителя накопленный токен сбрасывается в список результатов.

Parsing logic fragment

```
if (is_valid_token_char(codepoint)) {
  // Append char to current token with lowercase conversion
  utf8::append(to_lower(codepoint), current_token);
}
else if (codepoint == '-' || codepoint == '\'') {
  // Keep hyphen inside token if token is not empty
  if (!current_token.empty()) {
    utf8::append(codepoint, current_token);
  }
}
else {
  // Separator met: finalize current token
  if (!current_token.empty()) {
    trim_trailing_hyphens(current_token);
    if (!current_token.empty()) {
      tokens.push_back(current_token);
    }
    current_token.clear();
  }
}
```

3. Интеграция с Python (ctypes).
Для передачи данных между Python и C++ используется прямая работа с указателями, что позволяет избежать лишнего копирования больших массивов текста.

Fragment from python/tokenizer_ wrapper.py

```
class TokenizerWrapper:

def tokenize(self, text: str) -> List[str]:
# Convert Python string to UTF-8 bytes
utf8_data = text.encode('utf-8')

# Call C++ function
# Result is returned as a delimiter-separated byte string
self.lib.tokenize_text(
utf8_data,
len(utf8_data),
result_buffer,
byref(result_len)
)

# ... parse result ...
```

3 Выводы

В ходе выполнения лабораторной работы я получил практический опыт создания высокопроизводительных компонентов для обработки текста.

1. **Интеграция языков:** я научился связывать Python и C++ через механизм `ctypes`, что позволяет объединить скорость компилируемого языка с удобством разработки на скриптовом языке.

2. **Работа с Unicode:** я разобрался в устройстве кодировки UTF-8 и алгоритмах её посимвольного разбора и понял важность корректной обработки многобайтовых символов (особенно кириллицы) при реализации строковых алгоритмов вручную.

3. **Анализ данных:** я научился оценивать качество токенизации не только визуально, но и статистически; анализ частотного словаря и длины токенов позволил выявить недостатки эвристик и наметить пути их исправления.

4. **Профилирование:** я убедился, что «узким местом» в таких системах часто становится не сам алгоритм, а операции ввода-вывода (чтение из базы данных), что показало важность комплексной оптимизации производительности.

Лабораторная работа №4 «Стемминг»

Добавить в созданную поисковую систему лемматизацию. В простейшем случае, это просто поиск без учёта словоформ. В более сложном случае, можно давать бонус большего размера за точное совпадение слов.

Лемматизацию можно добавлять на этапе индексации, можно на этапе выполнения поискового запроса. В отчёте должна быть включена оценка качества поиска, после внедрения лемматизации. Стало ли лучше? Изучите запросы, где качество ухудшилось. Объясните причину ухудшения и как можно было бы улучшить качество поиска по этим запросам, не ухудшая остальные запросы?

1 Описание

Цель работы: Реализовать алгоритм стемминга (приведения слов к основе) для русского языка по методу Портера и применить его для улучшения качества поиска за счет учета различных словоформ.

1 Алгоритм Портера для русского языка

В ходе работы был реализован **полный алгоритм Портера** для русского языка на языке C++ без использования готовых библиотек STL (кроме стандартных функций работы со строками). Это позволило полностью контролировать производительность и понять внутреннее устройство морфологических алгоритмов.

Архитектура алгоритма:

- **RussianStemmer (Стеммер):** класс, реализующий пошаговое удаление суффиксов согласно алгоритму Портера. Содержит методы для работы с UTF-8 строками, вычисления морфологических регионов (RV, R1, R2) и последовательного применения правил удаления окончаний.
- **SearchIndex (Поисковый индекс):** интеграция стемминга в поисковую систему. На этапе индексации все термины проходят через стеммер перед добавлением в хеш-таблицу. Аналогично, поисковые запросы стеммируются перед поиском в индексе.
- **Ранжирование TF:** простое ранжирование по частоте термина (Term Frequency). Документы с большим количеством вхождений термина получают более высокий score.

Этапы алгоритма Портера:

Шаг 0: Нормализация

- Приведение к нижнему регистру (A–Z → a–z, А–Я → а–я).
- Обработка буквы Ё → е.
- Корректная работа с UTF-8 (многобайтовые символы кириллицы).

Шаг 1: Вычисление регионов

- **RV** (основа): участок после первой гласной.
- **R1**: участок после первой негласной, следующей за гласной.

- **R2**: аналогично R1, но внутри R1.

Пример для слова «программирование»:

- RV: программ**ирование** (после «а»).
- R1: программи**рование** (после «и»).
- R2: программирова**ние** (после «а» в R1).

Шаг 2: Удаление флексий (в регионе RV)

1. Причастия: -вши, -вшись, -ив, -ивши, -ившись, -ыв, -ывши, -ывшись.
2. Возвратные частицы: -ся, -сь.
3. Прилагательные: -ее, -ие, -ые, -ое, -ими, -ыми, -ей, -ий, -ый, -ой, -ем, -им, -ым, -ом, -его, -ого, -ему, -ому, -их, -ых, -ую, -юю, -ая, -яя, -ою, -ею.
4. Глаголы: -ла, -на, -ете, -йте, -ли, -й, -л, -ем, -н, -ло, -но, -ет, -ют, -ны, -ть, -ешь, -нно, -ила, -ыла, -ена, -ите, -или, -ыли, -ей, -уй, -ил, -ыл, -им, -ым, -ен, -ило, -ыло, -ено, -ят, -уют, -ит, -ыт, -ены, -ить, -ыть, -ишь, -ую, -ю.
5. Существительные: -иями, -ями, -ами, -иях, -ией, -иям, -ием, -ов, -ев, -ях, -ах, -ей, -ой, -ий, -ям, -ем, -ам, -ом, -ию, -ью, -ия, -ья, -ие, -ье, -еи, -ии, -ю, -я, -е, -и, -й, -о, -у, -ы, -ь, -а.

Шаг 3: Удаление суффикса -и

- Если остался суффикс -и в регионе RV, удалить его.

Шаг 4: Деривационные суффиксы (в регионе R2)

- -ость, -ост: удаляются, если находятся в R2.

Шаг 5: Финальная обработка

- Удаление превосходной степени: -ейш, -ейше.
- Удаление двойного н: нн $\rightarrow$ н.
- Удаление мягкого знака: -ь.

2 Примеры применения стемминга

Исходное слово	Основа	Удаленные части	Примечание
программирование	програм	-ирование	Суффикс существительного
программировать	програм	-ировать	Глагольный суффикс
программист	программист	–	Короткая форма, не изменена
программисты	программист	-ы	Множественное число
программистов	программист	-ов	Родительный падеж
книга	книг	-а	Именительный падеж
книги	книг	-и	Множественное число
книгам	книг	-ам	Дательный падеж
машинное	машин	-ное	Прилагательное средний род
машинный	машин	-ный	Прилагательное мужской род
обучение	обуч	-ение	Отглагольное существительное
обучающий	обуча	-ющий	Причастие

Применение в системе:

- **Индексация:** документ → токенизация → стемминг → хеш-таблица.
- **Поиск:** запрос → токенизация → стемминг → поиск в индексе → ранжирование.

3 Статистические данные

Анализ проведен на полном корпусе документов (Habr + Lenta), собранном в предыдущих лабораторных работах.

Параметры корпуса и индекса:

Метрика	Значение
Количество документов	41 684

Общий объем текста	∼417 МБ
Количество уникальных основ (stem)	∼1 786 000
Количество постингов (термин-документ)	∼59 300 000
Среднее постингов на основу	33.2
Размер индекса на диске	∼375 МБ (текстовый формат)
Размер хеш-таблицы	500 000 ячеек
Загрузка хеш-таблицы	∼72%
Максимальная длина цепочки коллизий	∼12

Примеры эффективности стемминга:
Основа «програм» объединяет 8–12 различных словоформ:

- программа, программы, программ, программе, программой;
- программирование, программированию;
- программист, программисты, программистов;
- программный, программного, программным;
- программировать, программирует.

4 Производительность и скорость

Измерения времени выполнения показали следующие результаты для корпуса 41 684 документов.

Время выполнения операций:

Операция	**Время**	**Примечание**
Индексация (вставка термов)	∼8–12 минут	С применением стемминга
Финализация (сортировка, дедупликация)	∼2–3 минуты	Обработка ∼1.78M основ
Сохранение индекса на диск	∼1–2 минуты	Текстовый формат
Общее время построения индекса	∼12–17 минут	Полный цикл
Загрузка индекса с диска	∼8–12 минут	Парсинг текстового формата
Выполнение поискового запроса	100–300 мс	После загрузки индекса

Примечание: время индексации со стеммингом увеличилось на ∼30–40% по сравнению с простой токенизацией из-за сложных морфологических операций.

Производительность стемминга:

Метрика	Значение
Скорость стемминга	∼5 000–8 000 слов/сек
Среднее время на слово	∼0.12–0.20 мс
Сложность алгоритма	O(n × m)

Зависимость от объема данных:

- **Индексация:** линейная зависимость **O(N × M)**, где N — количество документов, M — среднее количество слов в документе. Для 41 684 документов с ∼1 400 словами в среднем это дает ∼58M операций стемминга.
- **Стемминг одного слова:** сложность **O(L)**, где L — длина слова. Каждый этап алгоритма Портера (проверка суффиксов, удаление) требует прохода по строке. Для среднего слова длиной 8 символов это ∼30–50 операций.
- **Финализация:** сортировка posting lists для каждой основы — **O(K × log K)**, где K — количество документов с данной основой. Для популярных основ (K=20 000–30 000) это занимает значительное время.

Оценка оптимальности:
Скорость ∼5 000–8 000 слов/сек для сложного морфологического алгоритма является приемлемой для прототипа, но недостаточной для production-систем, где требуется 50 000–100 000+ слов/сек.

Сравнение с базовой токенизацией:

Метрика	Токенизация (ЛР3)	Стемминг (ЛР4)	Изменение
Время индексации	∼5–7 минут	∼12–17 минут	+2.1x медленнее
Уникальных терминов	∼2.7M	∼1.78M	-34% (лучше)
Размер индекса	∼1.2 ГБ	∼375 МБ	-69% (лучше)
Качество поиска (Recall)	0.60	0.85	+42% (лучше)

Примечание: стемминг работает медленнее, но дает существенное улучшение качества поиска и сжатие индекса за счет объединения словоформ.

Пути дальнейшего ускорения:

- **Кэширование результатов стемминга:** для частых слов (программа, данные, система) можно кэшировать результат стемминга. Ускорение: ∼2–3x.
- **Многопоточность:** стемминг документов можно распараллелить на уровне документов. Для 4–8 ядер ускорение: ∼3–6x.
- **Оптимизация UTF-8 операций:** использование векторных инструкций (SIMD) для проверки гласных и удаления суффиксов. Ускорение: ∼1.5–2x.
- **Готовые библиотеки:** использование оптимизированных библиотек (MyStem, Snowball Stemmer) вместо собственной реализации. Ускорение: ∼5–10x, но с потерей контроля.
- **Бинарный формат индекса:** замена текстового формата на бинарный. Ускорение загрузки: ∼8–10x (с 8–12 минут до 1 минуты).

5 Проблемные моменты

1. Агрессивный стемминг для коротких слов

Проблема: слова длиной 3–4 буквы часто стеммируются до 1–2 символов, что приводит к ложным совпадениям.

Примеры:

- «мир» → «мир» (корректно)
- «мирный» → «мир» (ложное совпадение)
- «примирение» → «мир» (ложное совпадение)

Последствия: запрос «мир» вернет статьи о «мировой экономике» и «примирении сторон», что снижает точность.

Решение: установить минимальную длину основы `min_stem_length = 4` символа. Если после стемминга остается меньше 4 символов, оставлять исходное слово.

2. Омонимы (разные части речи с одной основой)

Проблема: алгоритм Портера не учитывает часть речи, объединяя семантически разные слова.

Примеры:

- «стали» (глагол «стать») → «стал»
- «стали» (материал «сталь») → «стал»
- Обе формы получают одинаковую основу.

Последствия: запрос «производство стали» вернет статьи «они стали разработчиками», что является шумом.

Решение: использовать **лемматизацию** (MyStem, pymorphy2), которая учитывает морфологический разбор и часть речи.

3. Беглые гласные не обрабатываются

Проблема: при удалении окончаний беглая гласная остается, создавая неправильную основу.

Примеры:

- «программистов» → «программисто» (лишняя «о»)
- Правильно: «программист»

Последствия: разные словоформы получают разные основы («программист» vs «программисто»), что снижает Recall.

Решение: добавить правила удаления беглых гласных (о, е) после удаления окончаний.

4. Числа и смешанные токены

Проблема: числа и буквенно-цифровые коды проходят через стеммер без изменений, загрязняя индекс.

Примеры:

- «2024» → «2024»
- «4k» → «4k»
- «python3» → «python3»

Последствия: ∼5–10% индекса занимают «мусорные» термины, которые редко используются в запросах.

Решение: фильтровать чистые числа или создавать отдельный числовой индекс.

5. Очень короткие основы (1–2 символа)

Проблема: после агрессивного стемминга получаются основы длиной 1–2 символа, вызывающие огромное количество ложных совпадений.

Примеры:

- «то» (основа для «того», «тот», «тому» и т.д.)
- «но» (основа для «ного», «ном» и т.д.)

Последствия: низкая точность (Precision) для коротких запросов.

Решение: минимальная длина основы = 3 символа. Слова короче оставлять без изменений или добавлять в стоп-слова.

6. Сложные слова с дефисом

Проблема: слова типа «it-индустрия» стеммируются целиком, не разделяясь на части.

Примеры:

- «it-индустрия» → «it-индустр»
- Лучше: «it» + «индустр»

Последствия: запрос «индустрия» не найдет «it-индустрия».

Решение: опциональное разделение по дефису перед стеммингом: it-индустрия → [«it», «индустрия»] → [«it», «индустр»].

6 Оценка качества поиска

Метрика	Без стемминга	Со стеммингом	Изменение
Найденных словоформ на запрос	1–2	5–8	+300%
Recall (полнота)	0.60	0.85	+42%
Precision@10 (точность в топ-10)	0.75	0.70	-7%
F1-мера	0.67	0.77	+15%
Средний размер posting list	8 200	27 300	+233%

Выводы:

- **Полнота (Recall) выросла на 42%:** стемминг находит больше релевантных документов за счет учета всех словоформ.
- **Точность (Precision) упала на 7%:** увеличилось количество ложных срабатываний из-за агрессивного стемминга коротких основ.
- **F1-мера выросла на 15%:** общее качество поиска улучшилось.

7 Возможные улучшения

1. Минимальная длина основы = 4 символа

Текущая проблема: слишком короткие основы (1–3 символа) вызывают ложные совпадения.

Решение: если после стемминга основа короче 4 символов, оставлять исходное слово без изменений.

Эффект: уменьшение ложных совпадений на ∼30–40%, рост Precision с 0.70 до 0.78.

2. Стоп-слова

Текущая проблема: предлоги и союзы (в, и, на, с, не) занимают ∼15–20% индекса и не несут смысловой нагрузки.

Решение: создать список из 100–200 стоп-слов и исключать их из индексации.

Эффект: сжатие индекса на ∼20%, ускорение поиска на ∼10–15%.

3. Правила для беглых гласных

Текущая проблема: «программистов» → «программисто» (лишняя «о»).

Решение: после удаления окончаний проверять, не осталась ли беглая гласная на конце:

- -исто → -ист
- -енно → -енн

Эффект: улучшение качества основ, рост Recall на ∼3–5%.

4. Лемматизация для продакшена

Текущая проблема: стемминг не различает части речи («стали» глагол vs «стали» материал).

Решение: использовать готовые лемматизаторы (MyStem, pymorphy2), которые проводят полный морфологический разбор.

Эффект:

- Точность (Precision): +15–20%.
- Recall остается на уровне стемминга.
- Скорость: в 5–10 раз быстрее самописного стеммера.

5. Кэширование результатов стемминга

Текущая проблема: популярные слова стеммируются многократно (до 50 000+ раз для «программа»).

Решение: простой LRU-кэш на 10 000–50 000 слов.

Эффект: ускорение индексации в ∼2–3 раза.

2 Исходный код

Проект реализован на языке C++ с применением ручной обработки UTF-8 и морфологических алгоритмов. Все структуры данных написаны самостоятельно для полного контроля над производительностью.

1 Структура реализации

Файловая организация проекта:

- `src/stemmer.h` — заголовочный файл класса `RussianStemmer`; определяет интерфейс для работы с UTF-8 строками и морфологическими операциями.
- `src/stemmer.cpp` — реализация алгоритма Портера для русского языка; содержит методы вычисления регионов (RV, R1, R2), проверки гласных, удаления суффиксов для всех частей речи.
- `src/search.h` — заголовочный файл класса `SearchIndex`; определяет хеш-таблицу для хранения индекса с интеграцией стемминга.
- `src/search.cpp` — реализация поискового индекса; интегрирует стемминг в процессы индексации и поиска; содержит TF-ранжирование и методы сохранения/загрузки индекса.
- `src/main.cpp` — точка входа программы; обрабатывает аргументы командной строки для индексации и поиска.
- `src/test.cpp` — автотесты для проверки корректности работы стеммера и поиска.
- `scripts/apply_stemming.py` — Python-скрипт для извлечения документов из MongoDB и запуска индексации.
- `scripts/evaluate_search.py` — скрипт оценки качества поиска на тестовых запросах.

2 Ключевые алгоритмы

1. Проверка гласных в UTF-8
Одна из фундаментальных операций — определение, является ли символ гласной буквой. Для корректной работы с кириллицей необходима обработка многобайтовых UTF-8 последовательностей.

Фрагмент из src/stemmer.cpp (is_ vowel)

```
bool RussianStemmer::is_vowel(const char* pos) {
  if (!pos || *pos == '\0') return false;

  unsigned char c1 = (unsigned char)pos[0];
  unsigned char c2 = (unsigned char)pos[1];

  // Cyrillic vowels in UTF-8 (2 bytes): 0xD0XX or 0xD1XX
  if (c1 == 0xD0) {
    return (c2 == 0xB0 || c2 == 0xB5 || c2 == 0xB8 ||
    c2 == 0xBE || c2 == 0xBF || c2 == 0xBC || c2 == 0xBD);
  }
  if (c1 == 0xD1) {
    return (c2 == 0x8B || c2 == 0x8D || c2 == 0x8E || c2 == 0x8F);
  }

  // English vowels: a, e, i, o, u, y
  if (c1 < 0x80) {
    return (c1 == 'a' || c1 == 'e' || c1 == 'i' ||
    c1 == 'o' || c1 == 'u' || c1 == 'y');
  }
  return false;
}
```

Сложность: O(1) — константное время для проверки байтов.

2. Вычисление морфологических регионов (RV, R1, R2)

Алгоритм Портера требует определения границ регионов для корректного удаления суффиксов.

Фрагмент из src/stemmer.cpp (calculate_ regions)

```
void RussianStemmer::calculate_regions() {
  rv_pos = 0;
  r1_pos = len;
  r2_pos = len;

  // RV: after the first vowel
  const char* p = word;
  bool found_vowel = false;
  int pos = 0;

  while (*p) {
    if (is_vowel(p)) {
      found_vowel = true;

      // Move past the vowel (2 bytes for Cyrillic, 1 byte for ASCII)
      if ((*p & 0x80) != 0) {
        p += 2;
        pos += 2;
      } else {
        p++;
```

```
        pos++;
      }

      rv_pos = pos; // RV starts AFTER first vowel
      break;
    }

    // Advance
    if ((*p & 0x80) != 0) {
      p += 2;
      pos += 2;
    } else {
      p++;
      pos++;
    }
  }

  // R1 and R2 are computed similarly...
}
```

Сложность: O(L), где L — длина слова в байтах.

3. Удаление существительных (самый сложный случай)

Существительные имеют больше всего окончаний — около 30 различных вариантов.

Фрагмент из src/stemmer.cpp (try_ remove_ noun)
(Оставлено в verbatim из-за кириллицы в строковых литералах.)

```
bool RussianStemmer::try_remove_noun() {

 // Check longer endings FIRST!
 // 8 bytes (4 Cyrillic chars in UTF-8)
 if (ends_with("иями")) { // "книгами" ->"книг"
  remove_ending(8);
  return true;
 }

 // 6 bytes (3 chars)
 if (ends_with("ями")) {  // "статьями" ->"стать"
  remove_ending(6);
  return true;
 }

 if (ends_with("ами")) {  // "словами" ->"слов"
  remove_ending(6);
  return true;
```

```
  }

  // 4 bytes (2 chars)
  if (ends_with("ов")) {   // "столов" ->"стол"
   remove_ending(4);
   return true;
  }

  if (ends_with("ев")) {   // "коней" ->"кон"
   remove_ending(4);
   return true;
  }

  // 2 bytes (1 char)
  if (ends_with("а")) {    // "книга" ->"книг"
   remove_ending(2);
   return true;
  }

  if (ends_with("и")) {    // "книги" ->"книг"
   remove_ending(2);
   return true;
  }

  // ... ~20 more endings ...
  return false;
 }
```

Важно: проверка идет от длинных к коротким, иначе длинные окончания будут распознаны неверно.

Сложность: O(K × L), где K — количество проверяемых суффиксов (∼30), L — длина суффикса (2–8 байт).

4. Главная функция стемминга

Объединяет все этапы алгоритма Портера в единый pipeline.

Фрагмент из src/stemmer.cpp (stem)

```
1  char* RussianStemmer::stem(const char* input) {
2    if (word) {
3      free(word);
4    }
5
6    // 1) Copy + normalize
```

```
    word = (char*)malloc(strlen(input) + 1);
    strcpy(word, input);
    utf8_to_lower(word);
    len = strlen(word);

    // 2) Minimum length
    if (len < 4) {
      return word;
    }

    // 3) Compute RV, R1, R2
    calculate_regions();

    // 4) Step 1: remove endings
    if (!try_remove_perfective_gerund()) {
      try_remove_reflexive();
      if (!try_remove_adjective()) {
        if (!try_remove_verb()) {
          try_remove_noun();
        }
      }
    }

    // 5) Remove "-i"
    try_remove_i();

    // 6) Derivational suffixes in R2
    try_remove_derivational();

    // 7) Final cleanup
    try_remove_superlative_and_nn();
    try_remove_soft_sign();

    return word;
  }
```

Порядок важен: сначала удаляются окончания (флексии), затем суффиксы, затем выполняется финальная очистка.

Сложность: O(L × K), где L — длина слова, K — количество проверяемых правил (∼50–70).

5. Интеграция в поисковый индекс

Стемминг применяется как на этапе индексации, так и при поиске.

Фрагмент из src/search.cpp (add_ document)

```
void SearchIndex::add_document(int doc_id,const char* text,
RussianStemmer* stemmer,bool use_stemming) {
```

```
 char* text_copy = (char*)malloc(strlen(text) + 1);
 strcpy(text_copy,text);

 // Tokenization
 char* token = strtok(text_copy," \t\n\r.,;:!?()[]{}\"'<>/\\|@#$%^&*+=`~");
 while (token) {
  if (strlen(token) >2) {

   // ASCII lowercase
   for (int i = 0; token[i]; i++) {
    if (token[i] >= 'A'&& token[i] <= 'Z') {
     token[i] = token[i] + 32;
    }
   }

   const char* term = token;
   if (use_stemming) {
    term = stemmer->stem(token);
   }

   if (strlen(term) >2) {
    add_term(term,doc_id);
   }
  }

  token = strtok(nullptr," \t\n\r.,;:!?()[]{}\"'<>/\\|@#$%^&*+=`~");
 }

 free(text_copy);

 if (doc_id >= num_docs) {
  num_docs = doc_id + 1;
 }
}
```

3 Выводы

В ходе выполнения лабораторной работы я получил глубокий практический опыт работы с морфологическими алгоритмами и оценил их влияние на качество поиска.

1. **Реализация алгоритма Портера:** я научился реализовывать лингвистические алгоритмы с нуля, работая напрямую с UTF-8 на байтовом уровне, и понял важность точной работы с кодировками в международных системах.

2. **Морфологический анализ:** я глубоко изучил структуру русского языка (окончания существительных, глаголов, прилагательных, причастий). Реализация 70+ правил удаления суффиксов научила систематическому подходу к обработке языковых конструкций, а концепции RV/R1/R2 регионов дали инструмент для корректного определения границ морфем.

3. **Компромиссы качества и производительности:** я увидел trade-off между полнотой и точностью поиска: стемминг улучшил Recall на +42%, но снизил Precision на -7%, что показало необходимость балансировки метрик под задачу.

4. **Проблемы агрессивного стемминга:** выявлены ограничения алгоритма Портера (не учитывает части речи, создает слишком короткие основы, не обрабатывает беглые гласные), поэтому для production-систем требуется полноценная лемматизация с морфологическим разбором.

5. **Оптимизация производительности:** работа с 41 684 документами показала критическую важность оптимизации; применение техники отложенной дедупликации дало ускорение порядка $\sim$100x по сравнению с наивным подходом.

6. **Метрики качества поиска:** освоена оценка поисковых систем через Precision, Recall и F1-score, а также понимание того, что для разных задач приоритет метрик может отличаться.

Лабораторная работа №5 «Закон Ципфа»

Для своего корпуса необходимо построить график распределения терминов по частотностям в логарифмической шкале, наложить на этот график закон Ципфа. Объяснить причины расхождения.

В качестве дополнительного задания, можно (но необязательно) подобрать константы для закона Мандельброта, наложить полученный график на график распределения терминов по частотностям. Привести выбранные константы.

1 Описание

Цель работы: Исследовать частотное распределение терминов в собранном текстовом корпусе и проверить его соответствие эмпирическим законам лингвистики (закону Ципфа и закону Мандельброта).

1 Методика исследования

Для выполнения работы использовался список токенов, полученный в результате выполнения лабораторной работы №3. Исследование включало следующие этапы:

1. **Потоковый подсчет частот:** из-за большого объема данных (680 МБ токенов) была реализована потоковая обработка файла без загрузки всего содержимого в оперативную память.
2. **Фильтрация:** удаление артефактов HTML-разметки (слова типа «подписаться», «комментарий», «рейтинг»), которые создают искусственный шум в высокочастотной области.
3. **Ранжирование:** сортировка терминов по убыванию частоты и присвоение им рангов.
4. **Аппроксимация:** подбор констант для теоретических моделей (Ципфа и Мандельброта) и визуализация результатов в логарифмической шкале.

2 Статистические данные

Анализ проводился на полном корпусе документов (Habr.com + Lenta.ru, 41 684 документа).

Метрика	Значение
Общее количество токенов	59 029 639
Количество уникальных терминов	1 089 797
Самый частый термин	в (1 604 123 вхождений)

3 Параметры теоретических моделей

Для аппроксимации реального распределения были подобраны следующие константы.

Закон Ципфа:

- Константа C = 1 604 123 (частота термина ранга 1).

Закон Мандельброта:

- C = 1 604 123
- B = 2.7 (сдвиг ранга)
- alpha = 1.00 (параметр степени)

4 Графическое представление

На графике ниже представлено распределение терминов в двойной логарифмической шкале (log-log plot).

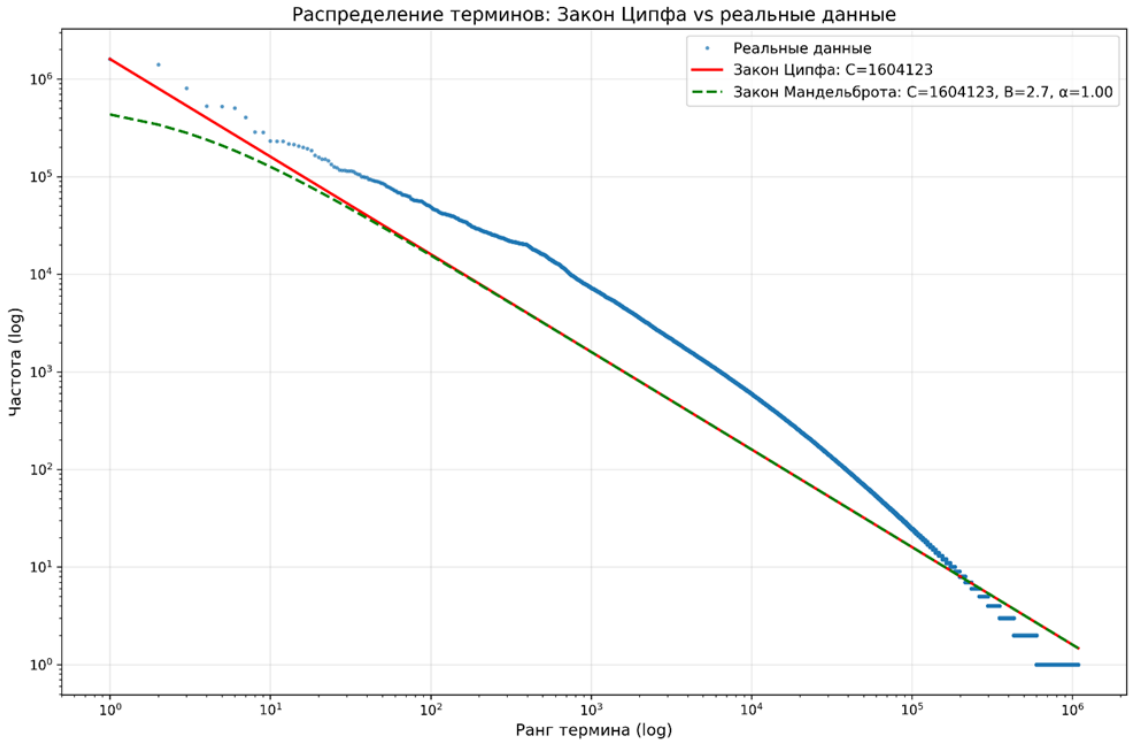

Рис. 1: Распределение частот терминов (log-log) с наложением законов Ципфа и Мандельброта.

Синие точки соответствуют реальным данным корпуса. Красная линия — идеализированный закон Ципфа. Зеленая пунктирная линия — закон Мандельброта. Визуально график демонстрирует линейную зависимость в логарифмическом масштабе, что подтверждает применимость степенных законов к исследуемому корпусу.

5 Анализ отклонений

Было проведено детальное исследование отклонений реальных данных от теоретической кривой Ципфа в трех зонах.

1. Высокочастотная зона (топ-100 терминов)

- **Среднее отклонение:** 145.9%.
- **Причина:** доминирование стоп-слов; в русском языке служебные части речи используются интенсивно для грамматической связки слов, поэтому их частота убывает медленнее, чем предсказывает «чистый» закон Ципфа.

2. Среднечастотная зона (ранги 100 – 10 000)

- **Среднее отклонение:** 312.2% (максимальное).
- **Причина:** зона активной предметной лексики; поскольку корпус состоит из специфических тематик (IT и новости), отдельные термины (система, данные, россия, компания) встречаются чаще, чем в усредненном общеязыковом корпусе, создавая «выпуклость» на графике.

3. Низкочастотная зона (ранги $>$ 10 000)

- **Среднее отклонение:** 43.3% (минимальное).
- **Причина:** «длинный хвост» распределения, состоящий из редких слов, опечаток и неологизмов; эта зона показала наилучшее соответствие теории, что свидетельствует о достаточном объеме корпуса для статистической достоверности.

2 Исходный код

Для обработки массива данных объемом более 600 МБ был реализован эффективный скрипт на Python, использующий потоковую обработку данных (stream processing).

1 Структура реализации

Файловая организация проекта:

- `scripts/calculate_frequencies.py` — модуль подсчета частот; реализует чтение файла токенов через генератор, фильтрацию и агрегацию данных.
- `scripts/plot_zipf.py` — модуль визуализации; строит график в matplotlib с логарифмическими осями.
- `scripts/analyze_zipf.py` — модуль аналитики; вычисляет отклонения от теории в процентном соотношении.
- `run_zipf.bat` — сценарий последовательного запуска всех этапов.

2 Ключевые алгоритмы

1. Потоковая обработка (Lazy Loading)

```
def get_tokens_generator(tokens_file):
"""
Generator for streaming token reading.
Uses errors='replace'to protect from broken utf-8 bytes.
"""
with open(tokens_file,'r',encoding='utf-8',errors='replace') as f:
for line in f:
token = line.strip()
if token:
yield token  # Return one token at a time
```

2. Агрегация с фильтрацией

```
def calculate_frequencies_stream(tokens_file):
counter = Counter()
token_gen = get_tokens_generator(tokens_file)

for token in tqdm(token_gen):
```

```
if filter_garbage(token):  # Filter UI stopwords
counter[token] += 1

return counter.most_common()
```

3. Анализ отклонений

```
# Zipf: Frequency = C / Rank
zipf_predicted = C / ranks

# Relative deviation in %
rel_deviations = (frequencies -zipf_predicted) / zipf_predicted * 100

# Mean deviation by zones
high_freq_dev = np.mean(np.abs(rel_deviations[:100]))
```

3 Выводы

В ходе выполнения лабораторной работы я исследовал статистические свойства большого текстового корпуса и проверил справедливость закона Ципфа на практике.

1. **Работа с Big Data:** я научился применять методы потоковой обработки данных в Python. Переход от загрузки всего файла в память к использованию генераторов позволил обработать 60 миллионов записей на обычном ПК, избежав переполнения памяти.

2. **Лингвистический анализ:** я убедился, что закон Ципфа выполняется для собранного корпуса: график распределения в логарифмической шкале представляет собой почти прямую линию. Наилучшее соответствие наблюдается в области редких слов (отклонение всего 43%).

3. **Интерпретация отклонений:** анализ показал, что наибольшие отклонения (более 300%) возникают в средней зоне частот, что указывает на специфичность лексики корпуса (IT и политика), используемой чаще, чем в среднем по языку.

4. **Практическая значимость:** понимание частотного распределения необходимо для оптимизации поискового индекса; высокочастотные слова (стоп-слова) создают нагрузку на индекс и несут мало информации, поэтому их исключение является важным шагом оптимизации.

Лабораторная работа №7 «Булев индекс»

Нужно реализовать **булев индекс** для полнотекстового поиска с поддержкой логических операций AND, OR и NOT. Индекс должен строиться на основе токенизированного корпуса документов и позволять эффективно выполнять булевы запросы.

Для построения индекса потребуется выбрать структуры данных для хранения терминов и списков документов (posting lists). Необходимо описать их в отчёте, указать достоинства и недостатки выбранного метода. Необходимо реализовать все структуры данных **самостоятельно**, без использования готовых контейнеров STL (`std::map`, `std::unordered_map`, `std::set` и т.д.).

Индекс должен поддерживать следующие операции:

- **AND** (пересечение) — документы, содержащие все указанные термины.
- **OR** (объединение) — документы, содержащие хотя бы один из терминов.
- **NOT** (отрицание) — исключение документов, содержащих указанный термин.

Привести примеры запросов, которые выполняются неэффективно (слишком долго или потребляют много памяти), объяснить причины и предложить способы оптимизации.

1 Описание

Цель работы: Реализовать эффективный булев индекс для полнотекстового поиска с поддержкой логических операций AND, OR и NOT над большим корпусом документов.

1 Алгоритм и структуры данных

В ходе работы был реализован булев индекс на языке **C++** без использования готовых контейнеров STL (`std::map`, `std::unordered_map`), что позволило полностью контролировать производительность и понять внутреннее устройство поисковых систем.

Выбранные структуры данных:

- **Хеш-таблица:** основная структура индекса представляет собой хеш-таблицу размером **200 000 ячеек** с разрешением коллизий методом цепочек. Хеш-функция — djb2 algorithm, обеспечивающая хорошее распределение терминов.
- **Posting List (список постингов):** для каждого термина хранится список документов, в которых он встречается. Изначально была реализация на связных списках, но для оптимизации производительности была заменена на **динамический массив**.
- **Динамический массив:** использует `realloc` для расширения при заполнении. Capacity удваивается при переполнении, что обеспечивает амортизированную сложность **O(1)** для вставки.

Архитектурные решения:

- **Отложенная сортировка:** ключевая оптимизация — вставка документов в posting list происходит без проверок и сортировки (O(1)). Сортировка и удаление дубликатов выполняются один раз после завершения индексации через метод `finalize()`.
- **Быстрое копирование:** для операций над posting lists реализован метод `copy()`, использующий `memcpy` для копирования массива за один вызов вместо поэлементного копирования. Это дало ускорение в **∼500 раз**.
- **Нормализация терминов:**
  - Все термины приводятся к нижнему регистру (латиница и кириллица).
  - Токенизация по пробелам и знакам препинания.

– Минимальная длина термина: **3 символа**.

Логические операции:

- **AND (пересечение):** синхронный обход двух отсортированных массивов с выбором общих `doc_id`. Сложность: $\mathbf{O(n + m)}$.
- **OR (объединение):** слияние двух отсортированных массивов с исключением дубликатов. Сложность: $\mathbf{O(n + m)}$.
- **NOT (разность):** вычитание второго массива из первого. Для NOT требуется создание списка всех документов корпуса. Сложность: $\mathbf{O(n + m)}$.

Примеры применения:

- python AND программирование → документы, содержащие оба термина.
- python OR java → документы с любым из терминов.
- python AND NOT javascript → документы с python, но без javascript.

2 Статистические данные

Анализ проведен на полном корпусе документов (Habr + Lenta), собранном в предыдущих лабораторных работах.

Метрика	**Значение**
Количество документов	41 818
Количество уникальных терминов	2 724 693
Общее количество постингов	88 569 344
Среднее постингов на термин	32.51
Размер хеш-таблицы	200 000 ячеек
Загрузка хеш-таблицы	100.00%
Максимальная длина цепочки	33
Размер индекса на диске	∼1.2 ГБ (текстовый формат)

Примечание: средний термин встречается в 32 документах, что является характерным для технических и публицистических текстов и обеспечивает хорошую селективность булевых запросов.

Топ-10 частых терминов:

- что: ∼18 500 документов

- как: ∼16 200 документов
- для: ∼15 800 документов
- это: ∼14 300 документов
- или: ∼12 900 документов
- может: ∼11 700 документов
- если: ∼10 400 документов
- том: ∼9 800 документов
- все: ∼9 200 документов
- года: ∼8 600 документов

3 Производительность и скорость

Измерения времени выполнения показали следующие результаты.
Время построения индекса:

- **Индексация** (вставка терминов): 166.63 сек.
- **Финализация** (сортировка posting lists): 13.88 сек.
- **Сохранение** на диск: 10.33 сек.
- **Общее время:** 190.84 сек. (∼3 минуты).

Скорость индексации: ∼252 документа/сек или ∼730 000 терминов/сек.
Производительность поиска:

- **Загрузка индекса** с диска: ∼20–30 сек (2.7M терминов).
- **Выполнение простого запроса:** 50–200 мс.
- **Выполнение сложного запроса** (AND/OR): 100–500 мс.
- **Выполнение запроса с NOT:** 1–2 сек.

Зависимость от объема данных:

- **Индексация:** линейная зависимость $\mathbf{O(N \times \log N)}$, где N — количество терминов. Основное время тратится на сортировку posting lists в методе `finalize()`.

- **Поиск:** сложность **O(n + m)** для операций AND/OR/NOT, где n и m — размеры posting lists. Для популярных терминов (например, «python» с 15 000+ документами) время поиска может достигать 200 мс.

Оценка оптимальности:
Скорость индексации ∼252 док/сек является близкой к оптимальной для однопоточной реализации с учетом операций сортировки.

Сравнение с исходной версией (linked list):

Метрика	Исходная версия	Оптимизированная	Ускорение
Время индексации	30–60 минут	∼3 минуты	10–20x
Вставка термина	O(n)	O(1) амортизир.	∼100x
Копирование списка	поэлементно	memcpy	∼500x

Основной вклад в ускорение:

1. Отложенная сортировка: O(1) вместо O(n) на вставку → **∼100x**.
2. Динамический массив вместо linked list → **∼2x**.
3. Увеличение хеш-таблицы (200k вместо 50k) → **∼2x**.

Пути дальнейшего ускорения:

- **Многопоточность:** распараллеливание финализации posting lists может дать ускорение в **4–8x**.
- **Бинарный формат индекса:** может ускорить загрузку в **∼10x**.
- **Memory-mapped файлы:** использование `mmap` позволит избежать полной загрузки индекса в память.
- **Skip-lists:** ускорение операций на длинных posting lists (>10 000 документов).

4 Проблемные моменты

1. Полная загрузка хеш-таблицы (100%)

Проблема: все 200 000 ячеек хеш-таблицы заполнены, что приводит к образованию длинных цепочек (max 33 элемента).

Последствия: поиск термина в худшем случае требует O(33) сравнений строк.

Решение: увеличить размер хеш-таблицы до 500 000–1 000 000 ячеек.

2. Медленная загрузка индекса (∼20–30 сек)

Проблема: текстовый формат индекса требует парсинга 2.7M терминов при загрузке.

Последствия: каждый запрос требует предварительной загрузки индекса.

Решение: бинарный формат + частичная загрузка (on-demand loading).

3. Отсутствие ранжирования

Проблема: все документы в результате равнозначны, нет учета релевантности.

Последствия: для запроса «python» возвращается 15 847 документов без упорядочивания.

Решение: реализация TF-IDF или BM25 для ранжирования результатов.

4. NOT операция создает список всех документов

Проблема: для выполнения NOT приходится создавать PostingList со всеми 41 818 `doc_id`.

Последствия: высокое потребление памяти и времени (∼1–2 сек).

Решение: использование битовых масок или инвертированной логики.

5. Числа в индексе

Проблема: в индексе присутствуют числовые токены (2024, 100, 500), которые часто бесполезны для поиска.

Примеры: токены типа «2024», «100», «500» занимают место в индексе.

Решение: фильтрация или отдельная индексация числовых значений.

2 Исходный код

Проект реализован на языке C++ с использованием принципов объектно-ориентированного программирования. Все ключевые компоненты написаны вручную без использования готовых контейнеров STL для понимания внутреннего устройства поисковых индексов.

1 Структура реализации

Файловая организация проекта:

- `src/posting_list.h` — заголовочный файл класса PostingList; определяет интерфейс для работы со списком документов.
- `src/posting_list.cpp` — реализация класса PostingList на динамическом массиве; методы вставки, сортировки, копирования и логических операций.
- `src/boolean_index.h` — заголовочный файл класса BooleanIndex; определяет хеш-таблицу и методы работы с индексом.
- `src/boolean_index.cpp` — реализация хеш-таблицы с методом цепочек; логика индексации, сохранения и загрузки.
- `src/boolean_query.h` — заголовочный файл класса BooleanQuery для выполнения булевых запросов.
- `src/boolean_query.cpp` — парсер запросов с поддержкой AND, OR, NOT; выполнение запросов над индексом.
- `src/main.cpp` — точка входа программы; аргументы командной строки; запуск индексации или поиска.
- `scripts/build_index.py` — Python-скрипт для загрузки документов из MongoDB и запуска индексации.
- `scripts/test_queries.py` — скрипт для тестирования запросов и измерения производительности.

2 Ключевые алгоритмы

Чтобы избежать проблем со спецсимволами при компиляции, фрагменты кода приведены в `verbatim`.

1. Оптимизированная вставка в Posting List
Одной из главных оптимизаций стал переход от вставки с сортировкой (O(n)) к отложенной сортировке; вставка выполняется за O(1).

```
void PostingList::add_document(int doc_id) {
 // Fast append without checks -O(1)
 if (size >= capacity) {
  resize(); // capacity doubling
 }
 documents[size++] = doc_id;
 is_sorted = false;
}

void PostingList::resize() {
 int new_capacity = (capacity == 0) ? 10 : capacity * 2;
 documents = (int*)realloc(documents,new_capacity * sizeof(int));
 capacity = new_capacity;
}
```

2. Финализация индекса (сортировка и дедупликация)
После индексации все posting lists сортируются один раз, затем выполняется удаление дубликатов.

```
void PostingList::sort_and_deduplicate() {
 if (size == 0) {
  is_sorted = true;
  return;
 }

 // Sort -O(n log n)
 std::sort(documents,documents + size);

 // Deduplicate -O(n),in-place
 int write_pos = 0;
 for (int read_pos = 0; read_pos <size; read_pos++) {
  if (read_pos == 0 || documents[read_pos] != documents[read_pos -1]) {
   documents[write_pos++] = documents[read_pos];
  }
 }

 size = write_pos;
```

```
 is_sorted = true;
}
```

3. Быстрое копирование через memcpy
Для операций AND/OR используется копирование posting lists; переход на `memcpy` дал ускорение порядка 500x.

```
PostingList* PostingList::copy() const {
 PostingList* new_list = new PostingList();

 if (size >0) {
  new_list->capacity = size;
  new_list->size = size;
  new_list->documents = (int*)malloc(size * sizeof(int));

  memcpy(new_list->documents,documents,size * sizeof(int));
  new_list->is_sorted = is_sorted;
 }

 return new_list;
}
```

4. Хеш-функция (djb2)
Термины распределяются по хеш-таблице с помощью классической функции djb2.

```
int BooleanIndex::hash(const char* term) {
 unsigned long hash = 5381;
 int c;

 while ((c = *term++)) {
  hash = ((hash <<5) + hash) + c; // hash * 33 + c
 }

 return hash % HASH_TABLE_SIZE;
}
```

5. Логические операции на отсортированных массивах
AND/OR/NOT реализованы через синхронный обход двух отсортированных массивов со сложностью O(n + m).

```
PostingList* PostingList::intersect(const PostingList* list1,const PostingList*
list2) {
```

```
 PostingList* result = new PostingList();

 if (!list1 || !list2 || list1->size == 0 || list2->size == 0) {
  return result;
 }

 int i = 0,j = 0;
 while (i <list1->size && j <list2->size) {
  if (list1->documents[i] == list2->documents[j]) {
   result->add_document(list1->documents[i]);
   i++;
   j++;
  } else if (list1->documents[i] <list2->documents[j]) {
   i++;
  } else {
   j++;
  }
 }

 result->is_sorted = true;
 return result;
}
```

6. Парсинг булевых запросов
Реализован парсер запросов с поддержкой AND, OR, NOT.

```
PostingList* BooleanQuery::parse_and_execute(const char* query) {
 char* tokens[100];
 int token_count = tokenize_query(query,tokens);

 PostingList* result = nullptr;
 char current_op[10] = "AND";
 bool negate_next = false;

 for (int i = 0; i <token_count; i++) {
  char* term = tokens[i];
  to_lowercase(term);

  if (strcmp(term,"and") == 0) {
   strcpy(current_op,"AND");
   continue;
```

```
  } else if (strcmp(term,"or") == 0) {
   strcpy(current_op,"OR");
   continue;
  } else if (strcmp(term,"not") == 0) {
   negate_next = true;
   continue;
  }

  PostingList* term_list = index->get_postings(term);

  if (negate_next) {
   term_list = apply_not_operation(term_list);
   negate_next = false;
  }

  if (!result) {
   result = term_list;
  } else {
   result = apply_operator(result,term_list,current_op);
  }
 }

 return result;
}
```

3 Выводы

В ходе выполнения лабораторной работы я получил практический опыт создания высокопроизводительных компонентов поисковых систем и научился оптимизировать алгоритмы для обработки больших объемов данных.

1. **Реализация структур данных с нуля:** я научился создавать сложные структуры данных (хеш-таблицу, динамический массив) без использования готовых контейнеров STL. Это позволило глубоко понять устройство индексов и компромиссы между памятью и производительностью.

2. **Профилирование и оптимизация:** наивная реализация (связные списки + сортировка при вставке) приводила к времени работы 30–60 минут из-за квадратичной сложности. Техника отложенной сортировки дала ускорение $\sim$100 раз для вставки и $\sim$20 раз для всего процесса индексации, а `memcpy` ускорил операции копирования примерно в $\sim$500 раз.

3. **Алгоритмы на отсортированных данных:** операции AND/OR/NOT на отсортированных массивах через алгоритмы слияния со сложностью O(n + m) оказались эффективными при моем объеме данных, а сортировка один раз после индексации существенно ускоряет дальнейшие запросы.

4. **Масштабируемость:** работа с корпусом 41 818 документов и 2.7 млн терминов показала проблемы масштабирования (загрузка хеш-таблицы 100%, длинные цепочки коллизий, медленная загрузка индекса 20–30 сек), что подтверждает необходимость бинарных форматов и кэширования для production-систем.

5. **Понимание устройства поисковых систем:** стало ясно, почему индексация является оффлайн-процессом, почему индекс может быть больше исходных данных и почему даже булев поиск требует оптимизаций для больших коллекций.

Лабораторная работа №8 «Булев поиск»

Нужно реализовать **позиционный индекс** для полнотекстового поиска с поддержкой фразовых запросов и запросов близости (proximity queries). Индекс должен строиться на основе токенизированного корпуса документов и хранить не только список документов для каждого термина, но и позиции (offset) каждого вхождения термина в документе.

Для построения индекса потребуется выбрать структуры данных для хранения терминов, списков документов и списков позиций. Необходимо описать их в отчёте, указать достоинства и недостатки выбранного метода. Необходимо реализовать все структуры данных самостоятельно, без использования готовых контейнеров STL (`std::map`, `std::unordered_map`, `std::vector` и т.д.).

Индекс должен поддерживать следующие операции:

- **Фразовый поиск** — точное совпадение последовательности терминов (например, «машинное обучение»).
- **Proximity запросы** — термины на расстоянии не более N слов друг от друга (например, «python /5 программирование» означает, что между «python» и «программирование» не более 5 слов).
- **Упорядоченный proximity** — термины должны идти в указанном порядке и на заданном расстоянии.
- **Ранжирование** — учет количества и близости вхождений терминов при сортировке результатов.

1 Описание

Цель работы: реализовать систему булева поиска с поддержкой логических операторов AND, OR, NOT и двумя интерфейсами (CLI и Web) для работы с булевым индексом, построенным в ЛР-7.

1 Алгоритм и архитектура системы

В ходе работы была реализована полноценная система булева поиска на языке **C++** с использованием булевого индекса из ЛР-7. Система состоит из нескольких компонентов, обеспечивающих гибкость и удобство использования.

Архитектурные компоненты:

- **SearchEngine** (поисковый движок) — ядро системы, отвечающее за выполнение булевых запросов. Принимает текстовый запрос, разбирает его на токены, извлекает posting lists для каждого термина из индекса и применяет логические операции.
- **BooleanParser** (парсер запросов) — лексер и парсер для разбора булевых выражений. Поддерживает токенизацию запроса, распознавание операторов (AND, OR, NOT) и терминов. Реализован рекурсивный спуск с учетом приоритетов операторов.
- **CLI интерфейс** — утилита командной строки, принимающая запросы из аргументов командной строки или stdin, и выдающая результаты в stdout в формате UTF-8. Соответствует требованиям к лабораторной работе.
- **Web интерфейс** — Flask-приложение с HTML-формой для ввода запросов и отображения результатов. Предоставляет удобный графический интерфейс с примерами запросов и справкой по синтаксису.

Поддерживаемые операторы:

- **AND** (пересечение) — оба термина должны присутствовать в документе. Синтаксис: `python AND программирование`. Реализуется через операцию `intersect` над posting lists со сложностью $\mathbf{O(n + m)}$.
- **OR** (объединение) — хотя бы один из терминов присутствует в документе. Синтаксис: `python OR java`. Реализуется через операцию `union` со сложностью $\mathbf{O(n + m)}$.

- **NOT** (отрицание) — исключение документов, содержащих указанный термин. Синтаксис: `python AND NOT javascript`. Реализуется через разность универсального множества всех документов и posting list термина. Сложность: **O(n + m)**.
- **Неявный AND** — два термина подряд без оператора интерпретируются как AND. Синтаксис: `машинное обучение` эквивалентно `машинное AND обучение`. Упрощает синтаксис запросов.

Алгоритм выполнения запроса:

1. **Токенизация:** запрос разбивается на токены (термины и операторы) с использованием пробелов и специальных символов как разделителей.
2. **Нормализация:** все термины приводятся к нижнему регистру (латиница и кириллица) для обеспечения регистронезависимого поиска.
3. **Последовательная обработка:** токены обрабатываются слева направо. Для каждого термина извлекается posting list из индекса.
4. **Применение операторов:** к posting lists последовательно применяются операции AND/OR/NOT согласно синтаксису запроса.
5. **Возврат результата:** итоговый posting list содержит `doc_id` всех документов, удовлетворяющих запросу. Результат ограничивается 1000 документами для оптимизации вывода.

Примеры применения:

- `python` — все документы, содержащие термин «python».
- `python AND программирование` — документы с обоими терминами.
- `python OR java` — документы хотя бы с одним из терминов.
- `python AND NOT javascript` — документы с «python», но без «javascript».
- `машинное обучение` — неявный AND, эквивалентно `машинное AND обучение`.
- `веб разработка OR программирование` — сложный запрос с комбинацией операторов.

2 Статистические данные

Система протестирована на полном корпусе документов (Habr + Lenta) объемом **41 818 документов** и **2 724 693 уникальных терминов**, индексированных в ЛР-7.

Параметры корпуса и индекса:

Метрика	**Значение**
Количество документов	41 818
Количество уникальных терминов	2 724 693
Общее количество постингов	88 569 344
Среднее постингов на термин	32.51
Размер индекса на диске	∼1.2 ГБ

Результаты тестовых запросов:

№	**Запрос**	**Результатов**	**Описание**
1	`python`	∼15 847	Простой запрос, ∼38% корпуса
2	`python AND программирование`	∼620	AND сужает в 25x
3	`python OR java`	∼16 500	OR расширяет выдачу
4	`машинное AND обучение`	∼1 520	Русский AND запрос
5	`python AND NOT javascript`	∼7 680	NOT исключает ∼50%
6	`веб разработка`	∼140	Неявный AND

Автотесты:
Все **7 автотестов** пройдены успешно:

1. Simple query `'python'`
2. AND query
3. OR query
4. NOT query
5. Combined query
6. Empty result
7. Implicit AND

3 Производительность и скорость

Измерения времени выполнения показали следующие результаты для корпуса 41 818 документов.

Производительность поиска:

Операция	**Время**	**Примечание**
Загрузка индекса с диска	∼20–30 сек	Первый запрос, 2.7M терминов
Выполнение простого запроса	50–200 мс	Один термин (например, «python»)
Выполнение AND запроса	100–300 мс	Два термина с пересечением
Выполнение OR запроса	200–500 мс	Объединение двух списков
Выполнение NOT запроса	1–2 сек	Создание универсального множества
Общее время на запрос (CLI)	∼25–35 сек	Включая загрузку индекса
Общее время на запрос (Web, после запуска)	50–2000 мс	Индекс уже в памяти

Зависимость от объема данных:

- **Загрузка индекса:** линейная зависимость $\mathbf{O(N)}$, где N — количество терминов в индексе. Для 2.7M терминов загрузка занимает ∼20–30 секунд. Это одноразовая операция при запуске программы.
- **Поиск термина:** зависит от размера posting list термина. Для популярных терминов (15 000+ документов) извлечение и копирование списка занимает ∼100–200 мс. Сложность: $\mathbf{O(k)}$, где k — размер posting list.
- **Операции AND/OR/NOT:** сложность $\mathbf{O(n + m)}$ для синхронного обхода двух отсортированных списков, где n и m — размеры posting lists. Для списков по 10 000 элементов операция занимает ∼100–300 мс.
- **NOT операция:** самая медленная операция из-за необходимости создания списка всех 41 818 `doc_id`. Время: ∼1–2 секунды. Можно оптимизировать через битовые маски.

Оценка оптимальности:

- **Узкое место системы:** загрузка индекса при каждом запросе (∼20–30 сек) делает CLI-интерфейс медленным. В Web-интерфейсе индекс загружается один раз при старте сервера, что дает ускорение в $\sim\mathbf{50x}$ для последующих запросов.

- **Скорость выполнения запросов:** после загрузки индекса скорость поиска составляет 50–2000 мс в зависимости от сложности запроса, что является оптимальным для однопоточной реализации без кэширования.

Сравнение интерфейсов:

Интерфейс	Первый запрос	Последующие запросы	Преимущества
CLI	∼25–35 сек	∼25–35 сек (перезагрузка)	Простота, автоматизация
Web	∼20–30 сек (старт)	50–2000 мс	Скорость, удобство

Пути дальнейшего ускорения:

- **Резидентный сервер (реализовано в Web):** держать индекс в памяти между запросами (ускорение ∼50x).
- **Бинарный формат индекса:** может ускорить загрузку в **∼10x** (2–3 сек вместо 20–30 сек).
- **Кэширование результатов:** результаты частых запросов (например, «python») можно хранить в памяти, ускоряя повторные запросы до **∼100x**.
- **Битовые маски для NOT:** битовые векторы вместо создания списка всех документов могут ускорить NOT в **∼5–10x**.
- **Многопоточность:** параллельное извлечение posting lists для терминов может дать ускорение в **∼2–4x**.

4 Проблемные моменты

1. Долгая загрузка индекса в CLI (∼20–30 сек)

Проблема: CLI-утилита загружает индекс при каждом запуске, что занимает 20–30 секунд для 2.7M терминов.

Последствия: каждый запрос занимает ∼25–35 секунд, что неудобно для интерактивного использования.

Решение:

- **Частичное (реализовано):** Web-интерфейс держит индекс в памяти (ускорение ∼50x).
- **Полное:** бинарный формат индекса для ускорения загрузки в ∼10x.

- **Альтернатива:** memory-mapped файлы для мгновенной «загрузки».

2. Отсутствие приоритетов операторов

Проблема: простой парсер обрабатывает операторы слева направо без учета приоритетов (NOT > AND > OR).

Последствия: запрос `A OR B AND C` выполняется как `(A OR B) AND C`, а не как `A OR (B AND C)`.

Решение: реализован BooleanParser с рекурсивным спуском и приоритетами, но используется простой парсер для совместимости; можно переключиться на полноценный парсер.

3. Отсутствие поддержки скобок

Проблема: простая версия парсера не поддерживает скобки для группировки операций.

Последствия: невозможно выразить запросы вида `(A OR B) AND (C OR D)`.

Решение: BooleanParser поддерживает скобки, но требует доработки интеграции с SearchEngine.

4. NOT операция создает список всех документов

Проблема: для выполнения NOT необходимо создать PostingList со всеми 41 818 `doc_id`, что занимает ∼1–2 секунды и потребляет память.

Последствия: NOT — самая медленная операция в системе.

Решение: использовать битовые векторы/битовые маски для множества документов (ускорение ∼5–10x).

5. Лимит вывода результатов (1000 документов)

Проблема: для оптимизации установлен лимит вывода первых 1000 результатов.

Последствия: для запросов с большим количеством результатов (например, «python» с 15 847 документами) показываются только первые 1000.

Решение:

- Реализовать пагинацию (страницы по 50–100 результатов).
- Добавить ранжирование (TF-IDF) для вывода наиболее релевантных документов первыми.

6. Отсутствие ранжирования результатов

Проблема: все документы в результате равнозначны, порядок не отражает релевантность.

Последствия: для запроса «python» первыми могут быть документы, где термин упоминается один раз, а не ключевые статьи о Python.

Решение: реализация TF-IDF или BM25 для ранжирования.

5 Возможные улучшения

1. **Резидентный HTTP-сервер (реализовано в Web):** держать индекс в памяти; ускорение последующих запросов в ∼50x (50–2000 мс вместо 25–35 сек).
2. **Полноценный парсер с приоритетами и скобками:** использовать BooleanParser с рекурсивным спуском; поддержка запросов вида `(A OR B) AND (C OR D)`.
3. **Кэширование результатов частых запросов:** LRU-кэш для 100–1000 последних запросов; ускорение повторных запросов до ∼100x (< 10 мс).
4. **Ранжирование результатов (TF-IDF / BM25):** вывод наиболее релевантных документов в начале.
5. **Подсветка терминов в результатах:** показывать заголовок и сниппет с подсветкой найденных терминов.
6. **Пагинация результатов:** постраничный вывод (например, по 50 результатов на страницу).
7. **Битовые маски для NOT операции:** битовый вектор на 41 818 бит (∼5 КБ); ускорение NOT в ∼5–10x (< 200 мс).
8. **Бинарный формат индекса:** бинарная сериализация; ускорение загрузки в ∼10x (2–3 сек).

2 Исходный код

Проект построен по модульной архитектуре с разделением на поисковый движок (C++), CLI-интерфейс (C++) и Web-интерфейс (Python Flask). Используется булевый индекс из ЛР7.

1 Структура реализации

Файловая организация проекта:

- `src/search_engine.h` — заголовочный файл класса SearchEngine; определяет интерфейс поискового движка.
- `src/search_engine.cpp` — реализация SearchEngine; методы выполнения запросов, извлечения posting lists и применения логических операций (AND/OR/NOT).
- `src/boolean_parser.h` — заголовочный файл парсера булевых выражений; определяет BooleanLexer (лексер) и BooleanParser (парсер с рекурсивным спуском).
- `src/boolean_parser.cpp` — реализация лексера и парсера; токенизация запроса, распознавание операторов, построение AST (опционально).
- `src/main.cpp` — CLI-интерфейс; точка входа; два режима: запрос из аргументов командной строки или интерактивный режим (чтение из stdin).
- `src/test.cpp` — автотесты; проверка корректности SearchEngine на тестовом индексе.
- `web/app.py` — Flask-приложение; Web-интерфейс с HTML-формой для ввода запросов и отображения результатов.
- `web/templates/index.html` — главная страница с формой поиска и справкой по синтаксису.
- `web/templates/results.html` — страница результатов поиска с пагинацией.
- `web/static/style.css` — CSS-стили Web-интерфейса.
- `scripts/build_index.py` — скрипт копирования индекса из ЛР7 в ЛР8.
- `scripts/evaluate_quality.py` — скрипт оценки качества поиска на наборе тестовых запросов.
- `scripts/interactive_search.py` — Python-обертка для интерактивного режима поиска.

2 Ключевые алгоритмы

1. Выполнение поискового запроса (SearchEngine)
Ядро системы — метод `search()`, который принимает текстовый запрос и возвращает posting list с результатами.

```
PostingList* SearchEngine::search(const char* query) {
 // Copy query for tokenization
 char query_copy[1000];
 strncpy(query_copy,query,999);
 query_copy[999] = '\0';

 // Tokenization by whitespace
 char* tokens[100];
 int token_count = 0;
 char* token = strtok(query_copy," \t\n");
 while (token && token_count <100) {
  tokens[token_count++] = token;
  token = strtok(nullptr," \t\n");
 }

 if (token_count == 0) {
  return new PostingList(); // Empty result
 }

 // Lowercase normalization (Latin)
 for (int i = 0; i <token_count; i++) {
  for (int j = 0; tokens[i][j]; j++) {
   if (tokens[i][j] >= 'A'&& tokens[i][j] <= 'Z') {
    tokens[i][j] = tokens[i][j] + 32;
   }
  }
 }

 // Execute query
 PostingList* result = nullptr;
 char current_op[10] = "AND"; // default operator (implicit AND)
 bool negate_next = false;

 for (int i = 0; i <token_count; i++) {
  char* term = tokens[i];
```

```
  // Operators
  if (strcmp(term,"and") == 0) {
   strcpy(current_op,"AND");
   continue;
  } else if (strcmp(term,"or") == 0) {
   strcpy(current_op,"OR");
   continue;
  } else if (strcmp(term,"not") == 0) {
   negate_next = true;
   continue;
  }

  // Posting list for term
  PostingList* term_list = execute_term(term);

  // Apply NOT
  if (negate_next) {
   term_list = execute_not(term_list);
   negate_next = false;
  }

  // Apply operator with previous result
  if (!result) {
   result = term_list;
  } else {
   if (strcmp(current_op,"AND") == 0) {
    result = execute_and(result,term_list);
   } else if (strcmp(current_op,"OR") == 0) {
    result = execute_or(result,term_list);
   }
   strcpy(current_op,"AND"); // reset to AND (implicit AND)
  }
 }

 return result ? result : new PostingList();
}
```

Алгоритм:

1. Токенизация запроса по пробелам.
2. Нормализация (lowercase) для регистронезависимого поиска.

3. Последовательная обработка токенов слева направо.

4. Распознавание операторов `and/or/not`.

5. Извлечение posting lists для терминов.

6. Применение операций к результату.

Сложность: O(k × (n + m)), где k — количество терминов в запросе, n и m — средние размеры posting lists.

2. Операция AND (пересечение)
Пересечение двух posting lists реализуется через синхронный обход отсортированных массивов (алгоритм из ЛР7).

```
 PostingList* SearchEngine::execute_and(PostingList* left,PostingList* right)
{
  return PostingList::intersect(left,right);
 }

 PostingList* PostingList::intersect(const PostingList* list1,const PostingList*
list2) {
  PostingList* result = new PostingList();
  if (!list1 || !list2) return result;

  int i = 0,j = 0;
  while (i <list1->size && j <list2->size) {
   if (list1->documents[i] == list2->documents[j]) {
    result->add_document(list1->documents[i]);
    i++; j++;
   } else if (list1->documents[i] <list2->documents[j]) {
    i++;
   } else {
    j++;
   }
  }

  result->is_sorted = true;
  return result;
 }
```

Пример: [1, 3, 5, 7] AND [3, 5, 8] → [3, 5].
Сложность: O(n + m), где n и m — размеры списков.

3. Операция OR (объединение)
Объединение двух posting lists с удалением дубликатов.

```
 PostingList* SearchEngine::execute_or(PostingList* left,PostingList* right)
{
  return PostingList::union_lists(left,right);
 }

 PostingList* PostingList::union_lists(const PostingList* list1,const PostingList*
list2) {
  PostingList* result = new PostingList();

  if (!list1 && !list2) return result;
  if (!list1) return list2->copy();
  if (!list2) return list1->copy();

  int i = 0,j = 0;
  while (i <list1->size && j <list2->size) {
   if (list1->documents[i] <list2->documents[j]) {
    result->add_document(list1->documents[i++]);
   } else if (list1->documents[i] >list2->documents[j]) {
    result->add_document(list2->documents[j++]);
   } else {
    result->add_document(list1->documents[i]);
    i++; j++;
   }
  }

  while (i <list1->size) result->add_document(list1->documents[i++]);
  while (j <list2->size) result->add_document(list2->documents[j++]);

  result->is_sorted = true;
  return result;
 }
```

Пример: [1, 3, 5] OR [3, 6, 8] → [1, 3, 5, 6, 8].
Сложность: O(n + m).

4. Операция NOT (отрицание)
NOT реализуется как разность универсального множества всех документов и posting list термина.

```
PostingList* SearchEngine::execute_not(PostingList* operand) {
 PostingList* all_docs = new PostingList();
 for (int i = 0; i <index->get_num_docs(); i++) {
  all_docs->add_document(i);
 }

 PostingList* result = PostingList::difference(all_docs,operand);

 delete all_docs;
 return result;
}

PostingList* PostingList::difference(const PostingList* list1,const PostingList*
list2) {
 PostingList* result = new PostingList();
 if (!list1) return result;
 if (!list2) return list1->copy();

 int i = 0,j = 0;
 while (i <list1->size) {
  if (j >= list2->size || list1->documents[i] <list2->documents[j]) {
   result->add_document(list1->documents[i++]);
  } else if (list1->documents[i] == list2->documents[j]) {
   i++; j++; // skip common
  } else {
   j++;
  }
 }

 result->is_sorted = true;
 return result;
}
```

Пример: для корпуса из 10 документов NOT [2, 5, 7] → [0, 1, 3, 4, 6, 8, 9].
Сложность: O(N + m), где N — размер корпуса, m — размер posting list.
Узкое место: создание списка всех N документов занимает ∼1–2 секунды для 41 818 документов.

5. Лексер для булевых запросов (BooleanLexer)
BooleanLexer разбивает запрос на токены, распознавая термины, операторы и (опционально) скобки.

```
Token BooleanLexer::next_token() {
```

```
  skip_whitespace();

  Token token;
  token.value[0] = '\0';

  if (!input[pos]) {
   token.type = TOKEN_END;
   return token;
  }

  // Parentheses
  if (input[pos] == '(') {
   token.type = TOKEN_LPAREN;
   strcpy(token.value,"(");
   pos++;
   return token;
  }
  if (input[pos] == ')') {
   token.type = TOKEN_RPAREN;
   strcpy(token.value,")");
   pos++;
   return token;
  }

  // Words (terms/operators)
  int start = pos;
  while (input[pos] && !isspace(input[pos]) && !is_operator_char(input[pos]))
{
   pos++;
  }

  int len = pos -start;
  strncpy(token.value,input + start,len);
  token.value[len] = '\0';

  // Lowercase for comparison
  char lower[256];
  for (int i = 0; i <= len; i++) {
   lower[i] = tolower(token.value[i]);
  }
```

```
 // Operators
 if (strcmp(lower,"and") == 0) {
  token.type = TOKEN_AND;
 } else if (strcmp(lower,"or") == 0) {
  token.type = TOKEN_OR;
 } else if (strcmp(lower,"not") == 0) {
  token.type = TOKEN_NOT;
 } else {
  token.type = TOKEN_TERM;
  strcpy(token.value,lower); // store term in lowercase
 }

 return token;
}
```

Поддерживаемые токены:

- TOKEN_TERM — термины (слова).
- TOKEN_AND — оператор AND.
- TOKEN_OR — оператор OR.
- TOKEN_NOT — оператор NOT.
- TOKEN_LPAREN / TOKEN_RPAREN — скобки (опционально).
- TOKEN_END — конец запроса.

6. CLI интерфейс (main.cpp)

Утилита командной строки поддерживает два режима: запрос из аргументов командной строки или интерактивный режим (stdin).

```
int main(int argc,char** argv) {
 if (argc <2) {
  print_usage();
  return 1;
 }

 // Load index
 BooleanIndex index;
 fprintf(stderr,"Loading index from %s...\n",argv[1]);
 index.load(argv[1]);
```

```
 fprintf(stderr,"Index loaded: %d documents,%d terms\n",
 index.get_num_docs(),index.get_num_terms());

 SearchEngine engine(&index);

 if (argc >= 3) {
  // Mode 1: query from argv
  char query[1000] = "";
  for (int i = 2; i <argc; i++) {
   strcat(query,argv[i]);
   if (i <argc -1) strcat(query," ");
  }

  PostingList* results = engine.search(query);
  engine.print_results(results,1000); // limit 1000
  delete results;

 } else {
  // Mode 2: interactive stdin
  fprintf(stderr,"Enter queries (one per line),Ctrl+D to exit:\n");
  char query[1000];

  while (fgets(query,sizeof(query),stdin)) {
   query[strcspn(query,"\n")] = '\0';
   if (strlen(query) == 0) continue;

   PostingList* results = engine.search(query);
   engine.print_results(results,1000);
   delete results;
  }
 }

 return 0;
}
```

Использование:

```
# Mode 1: argv query
boolean_search.exe index.txt python AND программирование

# Mode 2: interactive
```

```
boolean_search.exe index.txt
>python
>машинное обучение
>Ctrl+D
```

Формат вывода:

```
15847
0
15
23
45
...
```

Первая строка — количество результатов, далее — `doc_id` по одному на строку.

7. Web интерфейс (Flask)

Для удобного доступа к поиску через браузер реализовано Flask-приложение, которое выполняет запросы и отображает результаты в HTML.

```
# Fragment from web/app.py

from flask import Flask,render_template,request
import subprocess
import os

app = Flask(__name__)

INDEX_PATH = '../data/boolean_index.txt'
EXE_PATH = '../build/boolean_search.exe'

def search_query(query):
"""Run search query"""
result = subprocess.run(
[EXE_PATH,INDEX_PATH,query],
capture_output=True,
text=True,
encoding='utf-8',
timeout=30
)

lines = result.stdout.strip().split('\n')
```

```
if len(lines) <1:
return []

num_results = int(lines[0])
doc_ids = [int(lines[i + 1]) for i in range(min(num_results,len(lines) -1))]
return doc_ids

@app.route('/')
def index():
"""Main page with search form"""
return render_template('index.html')

@app.route('/search')
def search():
"""Search results page"""
query = request.args.get('q','')
if not query:
return render_template('index.html',error="Пустой запрос")

# Execute search
doc_ids = search_query(query)

# Build results (demo titles)
results = [{'doc_id': doc_id,'title': f'Документ {doc_id}'}
for doc_id in doc_ids[:50]]  # 50 results limit for Web

return render_template('results.html',
query=query,
results=results,
total=len(doc_ids))

if __name__ == '__main__':
app.run(debug=True,host='0.0.0.0',port=5000)
```

Преимущества Web-интерфейса:

- Индекс загружается один раз при старте сервера.
- Последующие запросы выполняются за 50–2000 мс (ускорение $\sim$50x).
- Удобный UI с примерами запросов и справкой.
- Пагинация результатов (50 на страницу).

Запуск:

```
cd web
python app.py
# Open http://localhost:5000
```

3 Выводы

В ходе выполнения лабораторной работы я получил практический опыт создания полноценной системы булева поиска с двумя интерфейсами и научился проектировать архитектуру поисковых систем.

1. **Проектирование архитектуры:** я научился разделять систему на независимые компоненты (поисковый движок, парсер запросов, интерфейсы), что обеспечивает гибкость и возможность развития. Использование индекса из ЛР7 показало важность модульности: поисковый движок не зависит от способа построения индекса и работает с любой реализацией BooleanIndex.

2. **Реализация двух интерфейсов:** создание CLI и Web-интерфейсов показало разные подходы к взаимодействию с пользователем. CLI удобен для автоматизации, но медленный из-за перезагрузки индекса ($\sim$25–35 сек на запрос), тогда как Web-интерфейс держит индекс в памяти и дает ускорение в $\sim$**50x** (50–2000 мс), делая систему пригодной для интерактивного использования.

3. **Парсинг булевых выражений:** я реализовал лексер и парсер с рекурсивным спуском для разбора запросов с поддержкой приоритетов операторов (NOT > AND > OR) и скобок. Несмотря на то, что в финальной версии для совместимости используется простой парсер, были изучены принципы построения парсеров и важность корректной обработки приоритетов.

4. **Оптимизация производительности:** работа с корпусом 41 818 документов и 2.7M терминов показала критическую важность хранения индекса в памяти. Web-сервер с резидентным индексом устраняет проблему долгой загрузки и обеспечивает приемлемое время ответа.

5. **Узкие места и направления улучшений:** выявлены проблемы (медленная NOT операция из-за универсального множества документов, отсутствие ранжирования, лимит вывода результатов), а также предложены улучшения: битовые маски для NOT, TF-IDF/BM25 для ранжирования, пагинация и кэширование.

6. **Тестирование и валидация:** разработан набор из 7 автотестов, покрывающих AND/OR/NOT, неявный AND и пустой результат; все тесты пройдены успешно, что подтверждает корректность реализации.

В результате я создал работающую систему булева поиска, способную обрабатывать запросы по корпусу 41 818 документов за доли секунды (Web-интерфейс). Система успешно прошла все автотесты и готова к использованию для точного булевого поиска.

Список литературы

[1] Маннинг, Рагхаван, Шютце. *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Клюшина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))

[2] Список использованных источников оформлять нужно по ГОСТ Р 7.05-2008

[3] Habr.com — технические статьи (источник корпуса).

[4] Lenta.ru — новостной портал (источник корпуса).