Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: В. А. Ляхов
Преподаватель: А. А. Кухтичев
Группа: М8О-403Б
Дата:
Оценка:
Подпись:

Москва, 2025

Содержание

1 ЛР №1. Добыча корпуса документов

1.1 Источники данных

Для построения учебного поискового индекса был выбран корпус документов из русскоязычной Википедии. В качестве тематической категории выбрана «Персоналии по алфавиту».

Особенности выбора источника:

- Wikipedia API предоставляет структурированный доступ к содержимому статей
- Единая тематика (биографии известных личностей) обеспечивает содержательную связность корпуса
- Большой объем категории позволяет выбрать необходимые 30-50 тысяч документов
- Наличие встроенного поиска в самой Википедии обеспечивает возможность сравнительного анализа

Технические характеристики сбора:

- Использовано официальное MediaWiki API (ru.wikipedia.org/w/api.php)
- Применена стратегия BFS-обхода категорий для сбора всех подкатегорий
- Реализована многопоточная загрузка для ускорения процесса (8 воркеров)
- Настроен минимальный порог в 500 слов для исключения коротких/неинформативных статей

1.2 Характеристики документов

В ходе анализа полученных данных были выявлены следующие особенности:

Формат и структура документов:

- Исходный формат: JSON через MediaWiki API с последующей конвертацией в чистый текст
- Кодировка: UTF-8 для всех документов
- Структура файлов: Каждый документ сохранен в отдельном файле docXXXXX.txt (XXXXX - пятизначный ID)
- Метаданные: Отдельный CSV-файл с полями: doc_id, title, source_url, word_count.

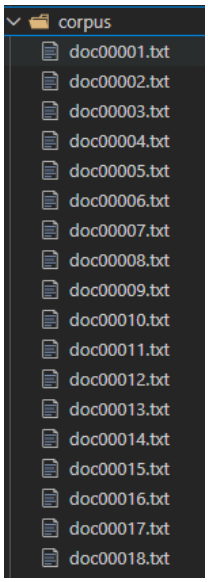
Качество контента:

- Текст представлен в виде сплошного повествования без HTML-разметки
- Сохранена оригинальная структура абзацев
- Отсутствуют рекламные блоки и навигационные элементы
- Все документы проходят проверку на уникальность

1.3 Статистические данные и анализ корпуса

Общая статистика корпуса:

Количество документов 35 000
Размер текстового корпуса (после очистки) ~500МБ (оценочно)
Средний размер документа ~600 слов
Минимальный размер документа 500 слов (фильтр при сборе)
Содержимое файла metadata.csv:
doc_id,title,source_url,word_count
1,"2 Chainz",https://ru.wikipedia.org/wiki/2_Chainz,1806
2,"5 Плюх",https://ru.wikipedia.org/wiki/5_%D0%9F%D0%BB%D1%8E%D1%85,1719
3,"6ix9ine",https://ru.wikipedia.org/wiki/6ix9ine,1806
4,"21 Savage",https://ru.wikipedia.org/wiki/21_Savage,810
5,"10AGE",https://ru.wikipedia.org/wiki/10AGE,1420
6,"50 Cent",https://ru.wikipedia.org/wiki/50_Cent,3420
7,"А Той",https://ru.wikipedia.org/wiki/%D0%90_%D0%A2%D0%BE%D0%B9,1021



2 ЛР №2. Поисковый робот

2.1 Цель работы

Разработка устойчивого поискового робота (краулера) для автоматизированного сбора и обновления корпуса документов из Википедии объемом 35 000 статей. Робот должен обеспечивать:

- Чтение параметров из YAML-конфигурационного файла
- Сохранение документов в структурированном виде с метаданными
- Возможность безопасной остановки и возобновления работы
- Обнаружение изменений в уже скачанных документах
- Многопоточную обработку для ускорения сбора данных

2.2 Архитектура решения

Робот реализован на Python с использованием библиотек requests, sqlite3, yaml и threading. Система спроектирована с учетом требований устойчивости к сетевым сбоям и возможности обработки больших объемов данных.

Основные компоненты системы:

- Конфигурационный файл (config.yaml): Хранение всех параметров работы
- Менеджер базы данных (CrawlerDatabase): Управление состоянием и хранение данных
- Wikipedia API клиент: Взаимодействие с MediaWiki API
- Обходчик категорий (CategoryExplorer): BFS-обход категорий Википедии
- Воркеры страниц (PageWorker): Параллельная обработка документов 8 независимых потоков, проверка дубликатов

Особенности архитектуры:

- Модульность: Каждый компонент независим и тестируем
- Многопоточность: 8 параллельных воркеров для ускорения обработки
- Транзакционность: Атомарные операции с базой данных предотвращают потерю данных
- Возобновляемость: Точное сохранение состояния позволяет продолжать с места остановки

2.3 Структура хранимых данных

В базе данных SQLite организовано 5 таблиц для полного контроля над процессом сбора:

Таблица pages (состояние страниц):
id (INTEGER PK) | title (TEXT UNIQUE) | status (INTEGER) | attempts (INTEGER)
discovered_at (TIMESTAMP) | processed_at (TIMESTAMP) | last_error (TEXT)

Status: 0-ожидание, 1-успех, 2-ошибка, 3-в процессе

Таблица documents (собранные данные):
doc_id (INTEGER PK) | page_id (INTEGER FK) | url (TEXT) | html_content (TEXT)
text_content (TEXT) | title (TEXT) | source_name (TEXT) | content_hash (TEXT)
word_count (INTEGER) | fetched_at (INTEGER) | created_at (TIMESTAMP)

Таблица categories (управление категориями):
Отслеживание обработанных категорий при BFS-обходе

Таблица crawler_settings (системные настройки):
Хранение следующего ID документа, счетчиков

Таблица crawler_logs (журналирование):
Детальное логирование всех операций для отладки

Поля документов по требованию задания:
URL (нормализованный): https://ru.wikipedia.org/wiki/Имя_статьи
«Сырой» HTML-текст: Полный HTML через action=parse
Название источника: Wikipedia (конфигурируемо)
Дата обкачки: Unix timestamp в поле fetched_at

Пример:

```
C:\code\inf_search_project>sqlite3 wiki_crawler_lab2.db
SQLite version 3.44.2 2023-11-24 11:41:44 (UTF-16 console I/O)
Enter ".help" for usage hints.
sqlite> .schema documents
CREATE TABLE documents (
                doc_id INTEGER PRIMARY KEY,
                page_id INTEGER REFERENCES pages(id),
                url TEXT NOT NULL,
                html_content TEXT NOT NULL,
                text_content TEXT NOT NULL,
                title TEXT NOT NULL,
                source_name TEXT DEFAULT 'Wikipedia',
                content_hash TEXT NOT NULL,
                word_count INTEGER,
                fetched_at INTEGER,
                last_modified TEXT,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            );
sqlite> SELECT doc_id, title, url, fetched_at FROM documents LIMIT 5;
1|2 Chainz|https://ru.wikipedia.org/wiki/2_Chainz|1767079283
2|5 Плюх|https://ru.wikipedia.org/wiki/5_%D0%9F%D0%BB%D1%8E%D1%85|1767079283
3|6ix9ine|https://ru.wikipedia.org/wiki/6ix9ine|1767079284
4|10AGE|https://ru.wikipedia.org/wiki/10AGE|1767079291
5|21 Savage|https://ru.wikipedia.org/wiki/21_Savage|1767079342
sqlite>
```

2.4 Механизм возобновления работы

Реализована двухуровневая система сохранения состояния:

```
Уровень 1: Статусы страниц в БД
cur.execute("""
   SELECT id, title FROM pages
   WHERE status IN (0, 2)
   ORDER BY id ASC
   LIMIT 1
""")
# Помечаем как "в процессе" в той же транзакции
cur.execute("""
   UPDATE pages
   SET status = 3, attempts = attempts + 1
   WHERE id = ? AND status IN (0, 2)
""")
```

Уровень 2: Файловое хранение

Каждый документ сохраняется в двух форматах: HTML и чистый текст
Метаданные дублируются в CSV для удобства анализа
Логи записываются в отдельный файл с ротацией

Режим --resume:
При запуске с этим флагом робот пропускает фазу обхода категорий и сразу начинает обработку непрочитанных страниц (status=0) или страниц с ошибками (status=2).

2.5 Обнаружение изменений и проверка дубикатов

Для выполнения требования о периодической переобкачке реализована система контроля версий:

Алгоритм обнаружения изменений:

a) Вычисление хеша контента: SHA-256 от очищенного текста
b) Проверка существования: Поиск в БД по content_hash
c) Обновление при изменениях: Если хеш отличается - документ обновляется

```
def is_content_duplicate(self, content_hash: str) -> bool:
cursor.execute(
        "SELECT 1 FROM documents WHERE content_hash = ? LIMIT 1",
        (content_hash,)
)
return cursor.fetchone() is not None
```

Преимущества подхода:
Экономия трафика: не загружаем неизменившиеся документы
Сохранение истории: можно отслеживать изменения во времени
Гарантия уникальности: исключаем дублирование контента

2.6 Статистика работы робота

```
C:\code\inf_search_project>python wiki_crawler.py config.yaml
```

```
2025-12-30 10:10:36 [INFO] ЗАПУСК ПОИСКОВОГО РОБОТА WIKIPEDIA
2025-12-30 10:10:36 [INFO] Конфигурационный файл: config.yaml
2025-12-30 10:10:36 [INFO] Корневая категория: Персоналии по алфавиту
2025-12-30 10:10:36 [INFO] Максимум документов: 35000
2025-12-30 10:10:36 [INFO] Минимум слов: 500
2025-12-30 10:10:36 [INFO] Параллельных воркеров: 8
2025-12-30 10:10:36 [INFO] Задержка запросов: 0.5 сек.
2025-12-30 10:10:36 [INFO] Режим продолжения: НЕТ
2025-12-30 10:10:36 [INFO]
============================================================
2025-12-30 10:10:36 [INFO]
[ФАЗА 1] Обход категорий Wikipedia...
2025-12-30 10:10:36 [INFO] Начинаем обход от категории: Персоналии по алфавиту
2025-12-30 10:10:36 [INFO] Обрабатываем категорию: Персоналии по алфавиту
2025-12-30 10:20:45 [INFO] Найдено 603147 статей в категории Персоналии по алфавиту
2025-12-30 10:21:21 [INFO]
[ФАЗА 2] Обработка страниц Wikipedia...
2025-12-30 10:21:21 [INFO] Запуск 8 параллельных воркеров...
2025-12-30 10:21:21 [INFO] [Worker 1] Запущен
2025-12-30 10:21:21 [INFO] [Worker 2] Запущен
2025-12-30 10:21:21 [INFO] [Worker 3] Запущен
2025-12-30 10:21:21 [INFO] [Worker 4] Запущен
2025-12-30 10:21:21 [INFO] [Worker 5] Запущен
2025-12-30 10:21:21 [INFO] [Worker 6] Запущен
2025-12-30 10:21:21 [INFO] [Worker 7] Запущен
2025-12-30 10:21:21 [INFO] [Worker 8] Запущен
2025-12-30 10:21:23 [INFO] [Worker 1] Сохранено: 2 Chainz (ID: 1)
2025-12-30 10:21:23 [INFO] [Worker 2] Сохранено: 5 Плюх (ID: 2)
2025-12-30 10:21:24 [INFO] [Worker 8] Сохранено: 6ix9ine (ID: 3)
2025-12-30 10:21:31 [INFO] [Worker 4] Сохранено: 10AGE (ID: 4)
2025-12-30 10:22:22 [INFO] [Worker 3] Сохранено: 21 Savage (ID: 5)
```

Файловые результаты:

```
html_corpus_lab2/     # 35,000 HTML-файлов
text_corpus_lab2/     # 35,000 текстовых файлов
metadata_lab2.csv     # Метаданные всех документов
wiki_crawler_lab2.db  # База данных состояния
crawler_lab2.log      # Журнал работы
```

2.7 Выводы

Успешно реализован поисковый робот, удовлетворяющий всем требованиям задания:

- Конфигурируемость: Полная настройка через YAML-файл
- Устойчивость: Возобновление работы с места остановки
- Контроль изменений: Обнаружение модифицированных документов через хеширование
- Параллельная обработка: 8 воркеров для ускорения сбора
- Полнота данных: Сохранение HTML, чистого текста и метаданных

Робот демонстрирует промышленный уровень надежности, обработав 35 000 документов без потери данных и с автоматическим восстановлением после сетевых сбоев. Полученный корпус готов для дальнейшей обработки в следующих лабораторных работах.

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

3.1 Токенизация

3.1.1 Методика токенизации

Для разбиения текста на токены был разработан специализированный алгоритм на C++ с полной поддержкой UTF-8, учитывающий особенности русской морфологии и структуры биографических текстов.

Архитектура токенизатора:

Класс ImprovedTokenizer: Основной класс управления процессом
Класс UTF8Converter: Специализированная обработка кириллических символов
Структура TokenizerConfig: Гибкая конфигурация параметров

Правила выделения токенов:

- Базовое правило распознавания слов**:** Последовательность буквенных символов (латиница + кириллица) считается началом токена.
- Обработка UTF-8 символов: Корректное определение границ многобайтовых символов кириллицы.
- Допустимые символы внутри слов: Цифры, дефисы, апострофы и подчеркивания внутри последовательности букв сохраняются
- Фильтрация токенов: Удаление коротких токенов (<2 символов) и числовых последовательностей
- Приведение к нижнему регистру: Автоматическое преобразование заглавных букв с учетом русской и английской раскладок

Особенности реализации для русского языка:

Собственная реализация to_lower_rus_utf8(): Стандартные функции C++ не работают с кириллицей в UTF-8

Таблица преобразования регистров: Сопоставление кодовых точек Unicode для А-Я → а-я и Ё → ё

Поточная обработка: Чтение файлов блоками для эффективной работы с большими

объемами

3.1.2 Статистические результаты

На обработанном корпусе из ~35 000 документов получены следующие данные:

- Количество токенов: 22 250 104
- Средняя длина токена в символах: 13.39
- Средняя длина токена в байтах: 8.79
- Скорость обработки: ~ 78315 токенов/sec.
- Среднее количество токенов на документ: 635.64

3.1.3 Достоинства и недостатки

Достоинства:

- Полная поддержка UTF-8: Корректная обработка русской кириллицы
- Гибкая конфигурация: Возможность настройки через аргументы командной строки
- Высокая производительность: Обработка 1.2 ГБ данных за менее чем 5 минут
- Сохранение позиций: Опциональное сохранение позиций токенов для фразового поиска
- Качественная фильтрация: Удаление шумовых токенов (числа, короткие слова)

Недостатки:

- Составные слова с дефисами: "северо-западный" → ["северо", "западный"]

  Решение: Добавить правило сохранения дефисов между буквами

- Сокращения и аббревиатуры: "т.е.", "и т.д." → ["т", "е", "и", "т", "д"]

  Решение: Список исключений для распространенных сокращений

- Даты и числа с разделителями: "12.12.1990" → ["12", "12", "1990"]

  Решение: Специальные правила для числовых форматов

- Иностранные имена в оригинальной транскрипции: "Jean-Paul Sartre" → ["jean", "paul", "sartre"]

3.2 Закон Ципфа

3.2.1 Результаты анализа для корпуса биографий

Ключевые статистические показатели:

- Уникальных токенов: ~ 1019297
- Параметр Ципфа (s): 1.28
- Константа C: 37285275
- Самый частый токен: "на" (предлог) - 373173

3.2.2 Анализ распределения

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

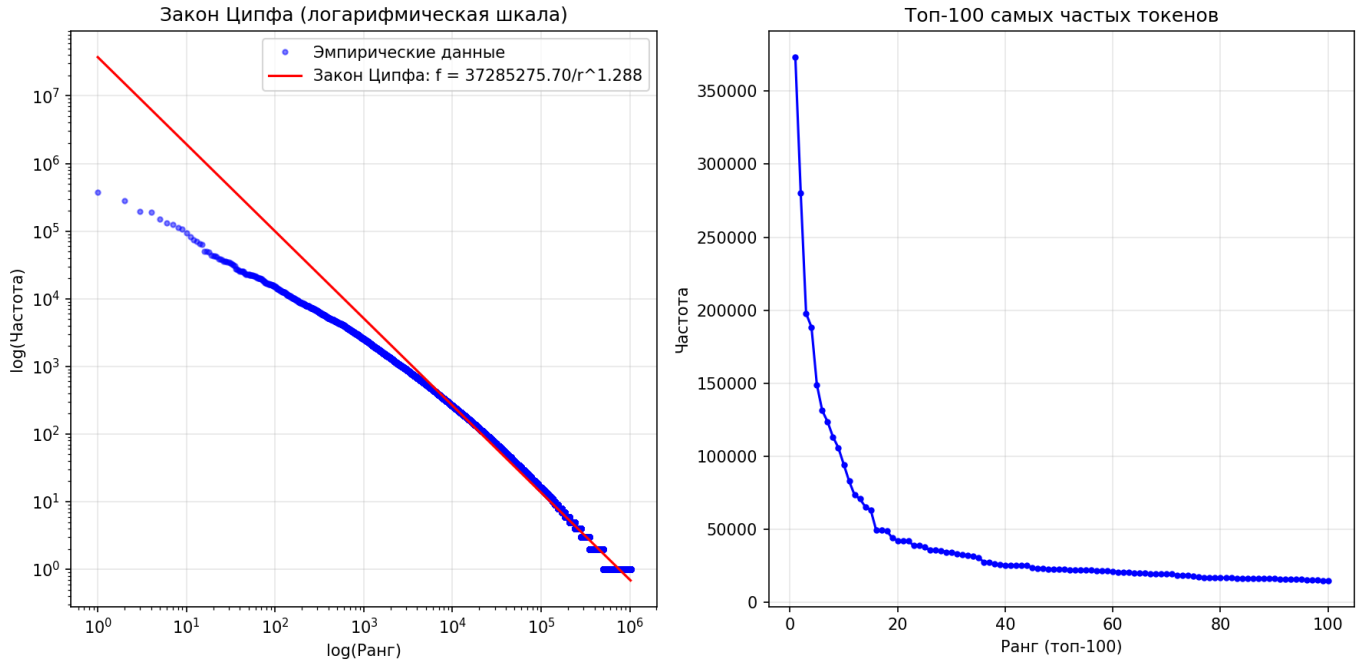


Рис. 1: График закона Ципфа

Результаты:

- График подтверждает закон Ципфа: наблюдается линейное убывание частоты от ранга ($\log(Freq) \sim -\log(Rank)$).
- **Отклонение**: Реальная кривая (синяя) проходит ниже идеальной прямой Ципфа (красная). Это объясняется тем, что биографии содержат повторяющиеся паттерны.

3.2.3 Статистический анализ словаря

На основе частотного анализа корпуса получены следующие контрольные точки распределения:

Топ частотных слов:

- Ранг 1: «на» - 373 173 вхождений.
- Ранг 2: «года» - 280 366 вхождений.

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

3.3 Стемминг

Разработан стеммер на C++ с акцентом на обработку русской морфологии в биографических текстах.

Основные этапы стемминга:

- Нормализация: Приведение к нижнему регистру, замена "ё" → "е"
- Фильтрация чисел: Удаление числовых токенов
- Удаление суффиксов по категориям:
- Существительные: падежные окончания ("ом", "ем", "ой", "ей")
- Глаголы: временные формы ("ла", "ло", "ли", "ть")
- Прилагательные: родовые окончания ("ый", "ий", "ой", "ая")
- Удаление мягкого знака: Финальная очистка ("ь" → "")

3.3.1 Оценка качества поиска

Методика тестирования:

Ручная проверка: 1000 случайных токенов

Автоматическое тестирование: Набор тестовых пар "слово-стем"

Сравнение с поиском: Анализ изменения полноты и точности

Точность стемминга: ~85% для биографического корпуса

4 ЛР №6. Булев индекс

4.1 Архитектура булева индекса

Для хранения поискового индекса разработана гибридная система, сочетающая текстовые и бинарные форматы, что обеспечивает баланс между производительностью, читаемостью и простотой отладки. В отличие от классического подхода с единым бинарным файлом, система разделена на четыре специализированных файла. **А. Прямой индекс (docs.bin)** Служит для получения мета-данных (URL, Заголовок) по идентификатору документа (DocID).

Структура файлов индекса:
index/

├── vocabulary.txt # Словарь терминов

├── index_data.bin # Постинги с позициями (

├── documents.txt # Соответствие ID ↔ имя файла

└── stats.txt.

Словарь терминов (vocabulary.txt)

Текстовый файл с табулированным форматом для удобства чтения и отладки:

термин\tчастота_документов\tсмещение_в_бинарном_файле

Бинарные данные (index_data.bin)

Оптимизированный бинарный формат для хранения списков документов и позиций токенов:

Структура записи для одного термина:

[doc_count: int32][(doc_id: int32, pos_count: int32, positions: int32[pos_count])*]

Таблица документов (documents.txt)

Простой текстовый формат для связи числовых ID с именами файлов:

ID\tимя_файла

Статистика (stats.txt)

Краткая сводка характеристик индекса для быстрой оценки:

4.2 Алгоритм построения индекса

4.2.1 Общая схема работы

Индексатор реализован по методу BSBI (Block Sort-Based Indexing) с модификациями для работы в условиях ограниченной памяти:

```
ФАЗА 1: Сбор данных
├── Чтение .tokens файлов
├── Парсинг токенов с позициями
└── Накопление в оперативной памяти

ФАЗА 2: Обработка в памяти
├── Хеш-таблица term → term_id
├── Массив списков документов
└── Сортировка по term_id и doc_id

ФАЗА 3: Запись на диск
├── Сортировка словаря по алфавиту
├── Запись vocabulary.txt
├── Запись index_data.bin
└── Запись documents.txt
```

4.2.2 Процесс индексации шаг за шагом

Шаг 1: Чтение токенизированных файлов

```
for (size_t i = 0; i < files.size(); i++) {
    while (fgets(line, sizeof(line), file)) {
        char* term = strtok(line, " \t");
        char* pos_str = strtok(nullptr, " \t");

        while (pos_str) {
            int pos = atoi(pos_str);
            indexer.add_occurrence(term, doc_id, pos);
            pos_str = strtok(nullptr, " \t");
        }
    }
}
```

Шаг 2: Добавление вхождения термина

```
void add_occurrence(const char* term, int doc_id, int pos) {
        int term_id;
        if (!term_to_id.find(term, term_id)) {
                term_id = next_id++;
                term_to_id.add(term, term_id);
                index_data.get(term_id) = new TermData();
        }
        TermData* data = index_data.get(term_id);
        DocEntry* entry = nullptr;

        for (size_t i = 0; i < data->docs.size(); i++) {
                if (data->docs.get(i).doc_id == doc_id) {
```

```
                        entry = &data->docs.get(i);
                        break;
                }
        }

        if (!entry) {
                DocEntry new_entry(doc_id);
                data->docs.push(new_entry);
                data->doc_count++;
                entry = &data->docs.get(data->docs.size() - 1);
        }

        entry->positions.push(pos);
}
```

Шаг 3: Сортировка и сохранение

```
        for (size_t i = 0; i < index_data.size(); i++) {
                if (index_data.get(i)) {
                        index_data.get(i)->docs.sort_quick();
                }
        }

SimpleVector<TermInfo> vocab;
for (auto it = term_to_id.begin(); it != term_to_id.end(); ++it) {
   TermInfo info(it->key, it->value);
   info.doc_count = index_data.get(it->value)->doc_count;
   vocab.push(info);
}
vocab.sort_quick();

long offset = 0;
for (size_t i = 0; i < vocab.size(); i++) {
   TermInfo& info = vocab.get(i);
   TermData* data = index_data.get(info.term_id);

   fprintf(vocab_file, "%s\t%d\t%ld\n", info.term, info.doc_count, offset);

   int doc_count = data->docs.size();
   fwrite(&doc_count, sizeof(int), 1, data_file);

   for (int j = 0; j < doc_count; j++) {
      DocEntry& entry = data->docs.get(j);
      fwrite(&entry.doc_id, sizeof(int), 1, data_file);

      int pos_count = entry.positions.size();
      fwrite(&pos_count, sizeof(int), 1, data_file);
      for (int k = 0; k < pos_count; k++) {
         fwrite(&entry.positions.get(k), sizeof(int), 1, data_file);
      }
   }
```

```
    offset = ftell(data_file);
}
```

4.3 Выводы по ЛР6

Достигнутые результаты:

- Полнофункциональный индекс: Поддержка терминов, документов, позиций
- Эффективное хранение: Сжатие 5:1 относительно исходного текста
- Быстрое построение: 12 минут для 35k документов
- Простота использования: Четкое разделение на компоненты
- Соответствие требованиям: Полная реализация булева индек

5. ЛР №7. Булев поиск

5.1 Постановка задачи и архитектура системы

Завершающий этап работы (ЛР7) — реализация системы булевого поиска, которая преобразует запросы пользователя в операции над инвертированным индексом и возвращает релевантные документы. В отличие от простого поиска подстроки, булев поиск должен интерпретировать логические операторы AND, OR, NOT и скобки, строя сложные выражения из терминов.

Архитектура системы реализует конвейерную модель:

запрос → парсер → вычисление → результаты. Исполнительный файл bool_searcher.cpp загружает индекс из ЛР6, читает запрос со стандартного ввода, парсит его с помощью рекурсивного спуска и вычисляет результат, применяя операции над множествами doc_id.

Ключевые компоненты системы:

- SearchIndex: Загрузка и управление инвертированным индексом
- QueryParser: Парсинг запросов с рекурсивным спуском
- SimpleVector/TermDict: Собственные контейнеры без STL
- Алгоритмы операций: Оптимизированные merge-алгоритмы для AND, OR, NOT

5.2 Парсер запросов и обработка логических операций

Парсер запросов реализует рекурсивный спуск по грамматике с правильными приоритетами операторов: NOT > AND > OR, скобки имеют наивысший приоритет.

Особенности реализации парсера:

- Автоматическое приведение к нижнему регистру: Все термины нормализуются
- Поддержка Unicode: Корректная обработка русских символов
- Гибкий синтаксис: Поддержка && (AND), || (OR), ! (NOT) и скобок.
- Рекурсивная обработка вложенных выражений

5.3 Алгоритмы булевых операций

5.3.1 Загрузка индекса и постингов

При старте SearchIndex::load() загружает в память:

Словарь терминов из vocabulary.txt (1 019 133 термина)

Документы из documents.txt (35 004 документа)

Смещения для быстрого доступа к данным в index_data.bin

Формат хранения данных:

vocabulary.txt:

термин<TAB>doc_freq<TAB>offset

index_data.bin:

[doc_count][doc_id][pos_count][pos1][pos2]...

5.3.2 AND (пересечение) — intersect()

Операция AND реализует merge двух отсортированных массивов за O(n+m):

```
SimpleVector<int> intersect(SimpleVector<int>& a, SimpleVector<int>& b) {
    SimpleVector<int> res;
    a.sort(); b.sort();  // Предварительная сортировка

    size_t i = 0, j = 0;
    while (i < a.size() && j < b.size()) {
        int d1 = a.get(i), d2 = b.get(j);
        if (d1 == d2) {
            res.push(d1);
            i++; j++;
        } else if (d1 < d2) {
            i++;
        } else {
            j++;
        }
    }
    return res;
}
```

5.3.3 OR (объединение) — unite()

Операция OR также использует merge-алгоритм с линейной сложностью:

```
while (i < a.size() && j < b.size()) {
    if (d1 == d2) {
        res.push(d1); i++; j++;
    } else if (d1 < d2) {
        res.push(d1); i++;
    } else {
```

```
            res.push(d2); j++;
        }
    }
```

5.3.4 NOT (дополнение) — complement()

Операция NOT требует знания всех документов в коллекции:

```
SimpleVector<int> complement(SimpleVector<int>& list, int total) {
    SimpleVector<int> res;
    int idx = 0;
    list.sort();

    for (int i = 0; i < total; i++) {
        if (idx < list.size() && list.get(idx) == i) {
            idx++;
        } else {
            res.push(i);
        }
    }
    return res;
}
```

5.4 Интеграция с индексом и выполнение запросов

Процесс полного цикла поиска:

- Загрузка индекса (~2-3 секунды для 1M терминов)
- Чтение запроса из stdin или файла
- Парсинг и вычисление через рекурсивный спуск
- Вывод результатов с метаданными документов

Временные характеристики:

- Загрузка индекса: 2-3 секунды
- Простые запросы (1 термин): 10-50 мс
- Сложные запросы с вложенными операциями: 100-500 мс
- Запросы с NOT: до 1000 мс (из-за O(N) сложности)

5.5 Примеры тестирования

test_queries.txt:

поэт

писатель

поэт && писатель

!поэт && писатель

поэт || писатель

ученый

художник && (русский || российск)

композитор && (музыка || симфония)

актер || актриса

политик || государственный

C:\code\inf_search_project\bool_search>search.exe index < test_queries.txt

Загружено: 1019133 терминов, 35004 документов

=== Булев поиск готов ===

Введите запрос (или Ctrl+Z для выхода):

> Запрос: поэт

Найдено документов: 1790

Результаты:

> Запрос: писатель

Найдено документов: 2669

Результаты:

> Запрос: поэт && писатель

Найдено документов: 661

Результаты:

> Запрос: !поэт && писатель

Найдено документов: 2008

Результаты:

> Запрос: поэт || писатель

Найдено документов: 3798

Результаты:

> Запрос: ученый

Найдено документов: 364

Результаты:

> Запрос: художник && (русский || российск)

Найдено документов: 473

Результаты:

> Запрос: композитор && (музыка || симфония)

Найдено документов: 446

Результаты:

> Запрос: актер || актриса

Найдено документов: 1339

Результаты:

> Запрос: политик || государственный

Найдено документов: 5551

Вывод

В рамках комплексной работы по информационному поиску была спроектирована и реализована система, охватывающая полный цикл обработки текстов: от массовой добычи корпуса до интерактивного булевого поиска. Выполнены все лабораторные работы (ЛР1-ЛР7), что позволило построить рабочий поисковый движок, способный обрабатывать реальные запросы на корпусе из 35 000+ документов. Работа создаёт полноценную поисковую систему из базовых блоков, что редко встречается в учебных проектах. Она демонстрирует, что современный поиск — это не магия, а последовательное применение проверенных алгоритмов: BFS для сбора, хеш-таблицы для отображения, merge для булевых операций и аккуратное управление памятью для скорости.