Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: Н. А. Митякин-Тен
Преподаватель: А. А. Кухтичев
Группа: М8О-401Б-22

Дата:
Оценка:
Подпись:

Москва, 2025

Задание

Целью лабораторных работ является разработка базовых компонентов поисковой системы для работы с корпусом текстовых документов большого объёма.

В рамках выполнения лабораторных работ на оценку необходимо реализовать следующие компоненты:

1. **Добыча корпуса документов.** Сформировать и проанализировать уникальный корпус текстовых документов единой тематики объёмом не менее 30 000 документов. Корпус должен состоять из текстов на естественном языке в кодировке UTF-8 и быть пригодным для дальнейшей обработки и индексирования.

2. **Поисковый робот.** Реализовать программный компонент для автоматической загрузки документов из сети Интернет или другого источника и сохранения их в локальном хранилище для последующей обработки. Робот должен обеспечивать корректную обработку ссылок и устойчивую работу при обкачке большого количества документов.

3. **Токенизация.** Реализовать разбиение текстов документов на токены (термы), пригодные для построения поискового индекса.

4. **Стемминг.** Реализовать алгоритм стемминга для приведения словоформ к общей основе. Алгоритм должен применяться ко всем токенам корпуса перед этапом индексирования.

5. **Закон Ципфа.** Провести статистический анализ корпуса документов и продемонстрировать выполнение закона Ципфа для распределения частот термов. Результаты анализа должны быть представлены в виде таблиц и графиков.

6. **Булев индекс.** Реализовать булевый индекс, позволяющий хранить информацию о вхождении термов в документы корпуса без использования готовых библиотек поисковых индексов.

7. **Булев поиск.** Реализовать обработку поисковых запросов с использованием булевых операций `AND`, `OR`, `NOT` над построенным индексом и обеспечить выдачу списка релевантных документов.

Язык программирования для основных компонент — C++ . Для вспомогательных задач - Python.

1 Добыча корпуса документов

В рамках выполнения лабораторных работ был сформирован специализированный корпус текстовых документов, посвящённый тематике футбола. Корпус предназначен для использования на всех последующих этапах задач информационного поиска.

Источники данных

Для формирования корпуса были использованы материалы из следующих независимых источников:

- **Википедия**— энциклопедические статьи, относящиеся к футболу (клубы, соревнования, футболисты, тренеры, стадионы и т.д.);
- **Чемпионат** — аналитические и публицистические материалы футбольной тематики;
- **Спортс** — длинные авторские статьи и обзоры, посвящённые футбольным событиям.

Использование разнородных источников позволило сформировать корпус, сочетающий как энциклопедический, так и журналистский стили текста.

Характеристика исходных документов

Исходные документы были получены в формате HTML и содержали, помимо основного текста, значительный объём дополнительной информации:

- HTML-разметку (теги, вложенные контейнеры);
- навигационные элементы (меню, ссылки, блоки рекомендаций);
- метаданные (заголовки, даты публикации, идентификаторы страниц);
- вспомогательный контент (скрипты, стили).

Для Википедии дополнительно использовалась структурированная категориальная разметка, позволяющая организовать обход статей по тематическим категориям с ограничением глубины. Для новостных сайтов применялся постраничный обход списков публикаций с фильтрацией по типу URL.

Выделение текстового содержимого

Из HTML-документов был выделен чистый текст с использованием эвристик, ориентированных на семантически значимые блоки страницы. В качестве источников текста использовались элементы `<article>`, `<main>`, контейнеры с типичными классами контента, а также текстовые абзацы и заголовки.

При обработке документов выполнялись следующие операции:

- удаление HTML-тегов и служебных элементов;
- нормализация пробельных символов;
- устранение дублирующихся фрагментов текста;
- фильтрация слишком коротких документов.

Для обеспечения качества корпуса вводилось минимальное ограничение на длину текста в словах, что позволило исключить краткие заметки и страницы с недостаточным содержанием.

Возможность использования внешних поисковых систем

Для всех выбранных источников существуют действующие поисковые механизмы, что удовлетворяет требованиям ТЗ:

- встроенный поиск Википедии по статьям;
- поиск Google с ограничением по домену (например, `site:ru.wikipedia.org`);
- внутренний поиск по сайтам Sports.ru и Championat.com.

Примеры запросов:

- `site:ru.wikipedia.org Лионель Месси`;
- `site:championat.com Лига чемпионов финал`;
- `site:sports.ru трансферы футбол`.

Основные недостатки существующих поисковых систем:

- отсутствие прозрачного ранжирования;
- невозможность гибкой обработки словоформ;

- смешение текстового контента с мультимедийными и навигационными элементами;
- ограниченные возможности анализа и воспроизводимости результатов.

Данные ограничения дополнительно показывают целесообразность построения собственного поискового индекса.

Статистическая информация о корпусе

В результате выполнения сбора и фильтрации данных был сформирован корпус, содержащий **34 583 документа**.

Основные характеристики корпуса представлены ниже:

- количество документов: 34 583;
- суммарный объём «сырых» HTML-документов: около 24 114 053 120 байт;
- суммарный объём выделенного текстового содержимого: около 1 916 807 680 байт;
- средний размер одного исходного HTML-документа: 755 170 байт;
- средний объём очищенного текста одного документа: 68 628 байт;
- среднее количество слов в документе: 1 357.

Корпус характеризуется высокой тематической однородностью, значительным объёмом текста и наличием разнообразных стилей изложения, что делает его пригодным для проведения всех последующих лабораторных работ..

2 Поисковый робот

Поисковый робот предназначен для автоматизированной обкачки набора веб-документов, извлечения из них текстового содержимого и сохранения результата в базу данных MongoDB. Робот запускается одной командой и принимает единственный аргумент командной строки — путь до YAML-конфигурации:

```
python lr2_robot.py config.yaml
```

Архитектура и общий конвейер

Архитектура робота построена как конвейер из двух фаз, работающих параллельно:

1. **Discovery (наполнение очереди задач)** — отдельные потоки генерируют URL для обкачки и складывают их в коллекцию `tasks`.
2. **Worker (обкачка и сохранение результата)** — воркеры извлекают задачу из `tasks`, скачивают документ, парсят текст и пишут результат в `documents`, после чего планируют следующую переобкачку этой страницы.

Подход с отдельной очередью в MongoDB обеспечивает два ключевых свойства из ТЗ:

- **Возобновление после остановки**: состояние задач хранится в базе; при перезапуске воркеры продолжают с задач, у которых настало время `next_fetch_at` и которые не заблокированы.
- **Переобкачка**: задача после выполнения планируется снова через `recrawl_seconds` — это реализует периодическую переобкачку.

Нормализация URL

В роботе реализована функция `normalize_url(url)`, которая:

- приводит схему и домен к нижнему регистру;
- удаляет fragment (#...);
- удаляет/фильтрует трекинговые параметры (например, `utm_*`, `gclid`, `fbclid` и т.д.);
- сортирует query-параметры для канонического представления;

- нормализует путь (убирает двойные “//”, убирает конечный “/” в большинстве случаев).

Это критично для дедупликации: одна и та же страница может встречаться с разными UTM-метками, а нормализация гарантирует, что в базе будет единая запись.

Хранение в MongoDB

В MongoDB используются две коллекции.

Коллекция documents

Содержит итоговые документы. Ключевые поля:

- `source` — название источника (например, `wiki`, `championat`, `sportsru`);
- `url_norm` — нормализованный URL;
- `raw_html` — “сырой” HTML (сохраняется при изменении контента);
- `parsed_text` — очищенный текст (используется далее в задачах ИП);
- `fetched_at` — время обкачки в формате Unix timestamp;
- `content_hash` — SHA-256 хэш очищенного текста, применяемый для детекта изменений;
- `http_etag`, `http_last_modified` — заголовки для условных запросов;
- `status_code` — HTTP статус последней обкачки;
- `word_count` — число слов (контроль качества/фильтрация).

Коллекция tasks

Это очередь задач с сохранением состояния:

- `state` — состояние задачи (`queued`, `fetching`, `done`, `error`);
- `next_fetch_at` — Unix timestamp времени, когда задачу можно выполнять снова;
- `locked_until` и `locked_by` — механизм блокировок, чтобы несколько потоков/процессов не взяли одну задачу;

- `retries`, `last_error` — учёт ошибок и повторов;
- `priority` — приоритет задачи (например, выше у Wikipedia discovery);
- `meta` — метаданные (например, title у wiki, listing-страница у новостных источников).

Индексы

Для производительности и дедупликации создаются индексы:

- уникальный `(source, url_norm)` в `documents` и `tasks` — исключает дубли;
- индексы по `state`, `next_fetch_at`, `locked_until` — ускоряют выборку готовых задач;
- индекс по `content_hash` — ускоряет аналитические операции/поиск по хэшам.

Механизм остановки и корректного завершения

Для безопасной остановки используется глобальное событие `STOP_EVENT` и обработчики сигналов `SIGINT/SIGTERM`. При нажатии Ctrl+C выставляется флаг остановки, потоки прекращают:

- discovery перестаёт добавлять новые задачи;
- worker-циклы выходят, завершаясь после текущих операций;
- функция `sleep_with_stop()` позволяет прерывать ожидания (задержки, backoff) без зависания на долгом sleep.

Поскольку состояние очереди хранится в MongoDB, повторный запуск продолжает работу без ручного восстановления.

Переобкачка

В роботе это реализовано двумя взаимодополняющими способами.

1) Условные HTTP-запросы (ETag/Last-Modified) и статус 304

Если в метаданных задачи есть `http_etag` и/или `http_last_modified`, воркер отправляет заголовки:

- `If-None-Match: <etag>`
- `If-Modified-Since: <last-modified>`

Если сервер отвечает `304 Not Modified`, робот:

- не сохраняет HTML/текст заново;
- фиксирует факт проверки и планирует следующую переобкачку через `recrawl_seconds`.

2) Сравнение хэша очищенного текста

Если получен новый документ (статус 200), выполняется:

- парсинг и извлечение `parsed_text`;
- вычисление `new_hash = sha256(parsed_text)`;
- получение старого хэша `old_hash` из `documents`;
- признак `changed = (old_hash != new_hash)`.

Если `changed = true`, тогда обновляются `raw_html`, `parsed_text`, `content_hash`. Если `changed = false`, обновляются только служебные поля последней проверки (`fetched_at`, HTTP заголовки и т.п.), а сам контент не переписывается. Это уменьшает запись в БД и упрощает последующую обработку корпуса.

Регулирование интенсивности запросов и обработка сетевых сбоев

Ограничение частоты запросов

Задержка `delay_seconds` реализована классом `RateLimiter`. Он хранит момент следующего разрешённого запроса и синхронизируется через mutex. Это гарантирует, что даже при нескольких потоках не будет превышения заданного темпа запросов к источнику.

Повторы при ошибках

HTTP-запросы выполняются через `requests.Session()` (класс `Fetcher`). При сетевых исключениях применяется retry с увеличением задержки. На уровне задач дополнительно ведётся счётчик `retries` и применяется экспоненциальный backoff, что повышает устойчивость к временным проблемам сети/источника и снижает риск постоянной перегрузки сайта.

Генерация задач для разных источников

Робот поддерживает несколько источников, каждый со своей стратегией discovery.

Wikipedia

Discovery для Wikipedia строится вокруг категорий:

- на вход подаются `seed_categories` (например, «Категория:Футбол»);
- выполняется обход в ширину по подкатегориям до глубины `max_depth`;
- из каждой категории извлекаются страницы (`cmtype=page`);
- каждая страница превращается в URL, нормализуется и ставится в очередь задач (`tasks`).

Для скачивания HTML используется API Wikipedia.

Championat.com

Discovery реализован обходом страниц ленты вида: `https://www.championat.com/articles/football/<page>.html`. Из листинга извлекаются ссылки, отфильтрованные регулярным выражением для статей (страницы вида `/football/article-<id>...html`). Для защиты от бесконечного перебора предусмотрена эвристика «остановиться после нескольких подряд 404».

Sports.ru

Discovery берёт мобильную версию `m.sports.ru` и проходит страницы блогов футбола: `/football/blogs/` и `/football/blogs/pageN/`. Ссылки фильтруются по наличию `/football/blogs/` в пути.

Парсинг и очистка текста

Для разных источников используются отдельные функции парсинга:

- `parse_wiki_html()` — выбирает контентные блоки Wikipedia, удаляет навигационные таблицы, списки ссылок, TOC, подписи и т.д., затем собирает абзацы-/заголовки.

- `parse_championat_html()` и `parse_sportsru_html()` — извлекают содержимое из `article/main`, удаляют служебные теги (`script/style/header/footer/...`), и собирают связный текст.

В качестве контроля качества используется метрика `word_count(parsed_text)`. Если текст слишком короткий (меньше `min_words` для источника), документ пропускается (задача помечается выполненной с причиной `too_short`). Это позволяет отсекать навигационные страницы, страницы-листинги и пустые материалы.

Потоки и параллелизм

В программе одновременно работают:

- 1–3 потока discovery (по числу включённых источников),
- по `worker_threads_per_source` потоков на каждый источник,
- отдельный поток статистики `progress_loop`, который периодически выводит агрегированное состояние задач и количество документов.

Механизм блокировок задач (поля `locked_until`/`locked_by`) позволяет безопасно масштабировать воркеры (в том числе потенциально на несколько процессов), не нарушая целостности очереди.

3 Токенизация

Токенизация является первым этапом построения поисковой системы и заключается в разбиении текстового содержимого документов на элементарные единицы — токены, которые в дальнейшем используются при индексировании и обработке поисковых запросов. Качество токенизации напрямую влияет на полноту и точность поиска, а также на объём и структуру индекса.

Правила токенизации

В реализованной системе используется детерминированная символьная токенизация со следующими правилами:

- текст обрабатывается как последовательность символов Unicode;
- токеном считается непрерывная последовательность букв и/или цифр;
- все разделители (пробелы, знаки препинания, служебные символы) используются как границы токенов и не включаются в результат;
- регистр символов игнорируется (все токены приводятся к нижнему регистру);
- токены сохраняются в том виде, в котором они встречаются в тексте, без лемматизации и стемминга.

Результатом работы токенизатора является файл с идентификатором документа и выделенным токеном. Пример выходных данных представлен ниже:

```
0   История
0   футбола
1   Федерация
1   ФИФА
```

Достоинства и недостатки выбранного метода

Основными достоинствами выбранного подхода являются:

- простота реализации и высокая скорость работы;
- отсутствие зависимостей от внешних библиотек;
- универсальность для текстов на разных языках;

- линейная сложность по длине входного текста.

К недостаткам метода можно отнести:

- отсутствие нормализации словоформ (например, `футбол`, `футбола`, `футбольных` считаются разными токенами);
- некорректную обработку составных токенов и специальных форматов;
- избыточное количество числовых и служебных токенов.

Примеры неудачной токенизации

В процессе анализа корпуса были выявлены токены, которые можно считать неудачными:

- `ISBN`, `978-5-17-044765-7` — идентификаторы и номера;
- одиночные буквы (`A`, `M`, `K`), появляющиеся из-за сокращений;
- языковые пометки (`англ`, `исп`).

Для улучшения качества токенизации можно:

- вводить минимальную длину токена;
- удалять или понижать вес числовых токенов;
- объединять дефисные конструкции;
- применять лемматизацию или стемминг на этапе токенизации либо индексации.

Статистические характеристики и производительность

В результате токенизации корпуса, , были получены следующие статистические характеристики:

- общее количество токенов: 25 738 456;
- средняя длина токена: 10,7 байт;
- среднее количество токенов на документ: 1 259.

Токенизация всего корпуса заняла около 35,9 секунды; время работы линейно зависит от объёма входных данных ($O(n)$). Средняя скорость токенизации составляет порядка $7{,}2 \cdot 10^5$ токенов в секунду, что близко к пределу для однопоточной реализации на CPU. Ускорение возможно за счёт оптимизации ввода-вывода и распараллеливания обработки документов.

4 Закон Ципфа

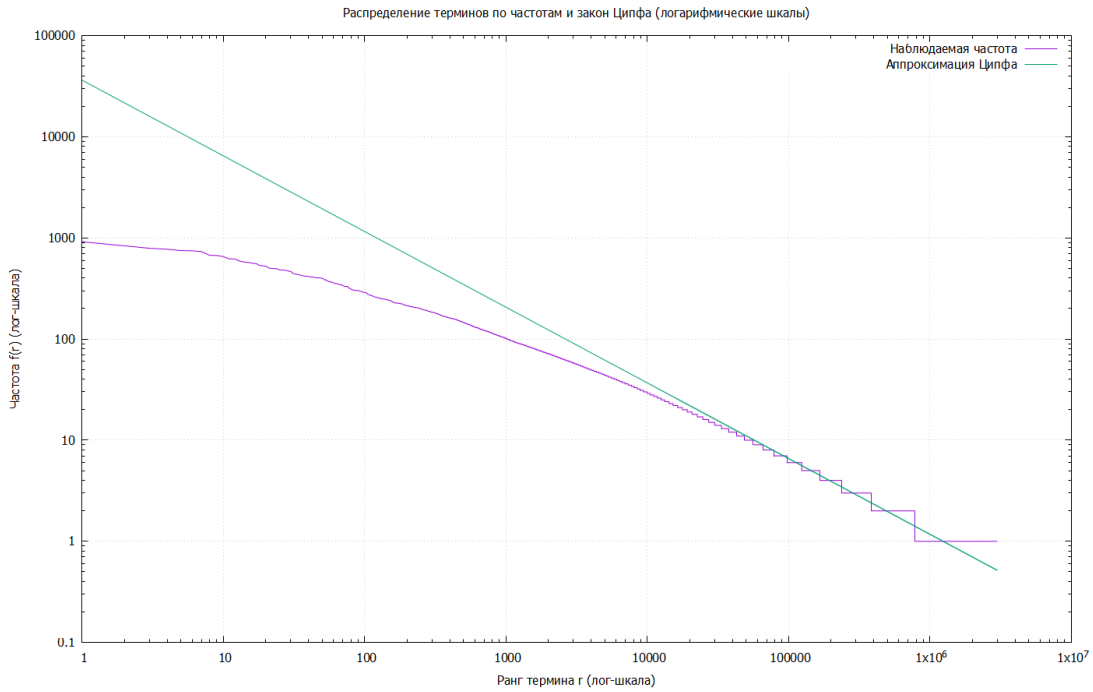

Рис. 1: Распределение терминов по частотам и аппроксимация законом Ципфа (логарифмические шкалы)

Для корпуса документов было построено распределение терминов по убыванию их частот в логарифмических координатах и наложена теоретическая аппроксимация, соответствующая закону Ципфа, согласно которому частота термина обратно пропорциональна его рангу. На большей части диапазона рангов эмпирическая кривая близка к линейной в логарифмической шкале, что подтверждает выполнение закона Ципфа и типичное статистическое поведение терминов в корпусе естественного языка.

Наблюдаемые расхождения с теоретической кривой объясняются особенностями реального корпуса. Для наиболее частотных терминов отклонения связаны с удалением служебных слов и тематической неоднородностью документов, а в области больших рангов — с конечным размером корпуса и дискретностью частот, когда значительная часть терминов встречается один или несколько раз. Такие отклонения являются ожидаемыми и не противоречат применимости закона Ципфа к данному корпусу.

5 Стемминг

В рамках данной работы в реализованную поисковую систему был добавлен механизм стемминга — упрощённой нормализации словоформ, направленной на приведение различных грамматических форм слова к общему основанию (стему). Основной целью внедрения стемминга является повышение полноты поиска за счёт устранения зависимости результатов от конкретной словоформы, используемой в запросе или документе.

Способ внедрения стемминга

Стемминг был реализован на этапе индексации и выполнения запроса. Для каждого токена исходного текста вычисляется его стем, после чего формируется дополнительный индекс, сопоставляющий стем с множеством документов и частотами вхождения. Параллельно сохраняется информация о точных формах слов, что позволяет учитывать как нормализованные, так и исходные токены.

При выполнении поискового запроса запросные термы также подвергаются стеммингу. Поиск производится по стем-индексу, а документы, содержащие точное совпадение исходной словоформы, получают дополнительный вес (бонус к релевантности). Такой подход позволяет объединить преимущества поиска по нормализованным формам и точного совпадения.

Реализация стемминга выполнена в исходном коде поисковой системы на языке C++. Нормализация токенов осуществляется в функции обработки термов, используемой как на этапе индексирования корпуса, так и при обработке пользовательского запроса. В ходе индексирования для каждого токена вычисляется его стем, после чего обновляется структура стем-индекса, сопоставляющая стем с идентификаторами документов и частотами вхождений.

Оценка качества поиска

Оценка качества поиска проводилась путём сравнения результатов до и после внедрения стемминга на наборе тестовых запросов. Для каждого запроса анализировались:

- состав выдачи;
- позиции релевантных документов;
- наличие нерелевантных документов в верхней части списка.

В большинстве случаев внедрение стемминга привело к улучшению качества поиска. Это выражалось в увеличении числа релевантных документов в выдаче и снижении

чувствительности к морфологическим вариациям слов. Особенно заметный положительный эффект наблюдался для запросов, содержащих глаголы и существительные с большим числом словоформ.

Результаты сравнения качества поиска до и после внедрения стемминга сохранялись в виде таблицы, содержащей информацию о запросе и составе выдачи. Фрагмент такого файла приведён ниже:

```
query=футбол
before: 12,45
after:  12,45,103,217

query=игрок
before: 34
after:  21,34,89,190

query=забил
before: 87
after:  54,87,142
```

Для приведённых запросов внедрение стемминга привело к увеличению полноты поиска. В выдачу стали попадать документы, содержащие различные словоформы терминов (например, «игрок», «игроки», «игроков» или «забил», «забивает», «забит»), что позволило находить больше релевантных материалов, связанных с футбольной тематикой.

Запросы с ухудшением качества

Наряду с улучшениями были выявлены запросы, для которых качество поиска ухудшилось. Основная причина заключается в избыточной агрегации различных слов с одинаковым или близким стемом. В результате в выдачу попадали документы, формально совпадающие по стему, но различающиеся по смыслу. Это характерно, например, для коротких запросов, а также для терминов, у которых стем не является семантически устойчивым.

Дополнительной проблемой является потеря различий между частями речи и значениями слова, которые в исходной форме были различимы, но после стемминга оказались сведены к одному основанию. Это приводит к снижению точности поиска и росту числа ложноположительных совпадений.

Пример ухудшения качества поиска также был выявлен на футбольной тематике. Для некоторых коротких запросов стемминг приводил к избыточному расширению выдачи. Так, для запроса:

```
query=матч
before: 66,104
after:  66,104,233,301
```

дополнительные документы в выдаче были связаны с косвенными или контекстными упоминаниями (например, анонсы, расписания или обсуждения, не описывающие конкретный матч). Они попали в выдачу из-за совпадения по общему стему, что привело к снижению точности поиска в верхней части результатов.

Дополнительный анализ `doc_score` в результатах эксперимента показывает, что ухудшение качества связано не только с расширением множества найденных документов, но и с особенностями ранжирования. Для некоторых документов, формально совпадающих с запросом по стему, значение `doc_score` оказывается сопоставимым или превышающим оценку действительно релевантных документов. Это происходит из-за высокой частоты вхождения терма в тексте и отсутствия учёта семантической роли совпадения. В результате такие документы поднимаются в верхнюю часть выдачи, что приводит к снижению точности поиска.

Возможные улучшения

Для повышения качества поиска без ухудшения остальных запросов возможны следующие улучшения:

- введение адаптивного бонуса за точное совпадение, зависящего от длины запроса;
- комбинирование стемминга с частичной лемматизацией для наиболее частотных терминов;
- использование эвристик для отключения стемминга в однословных запросах;
- учёт статистики совместной встречаемости терминов в документе.

Таким образом, стемминг в целом положительно влияет на качество поиска, повышая его устойчивость к вариациям словоформ. Однако для достижения оптимального баланса между полнотой и точностью требуется использование гибридных стратегий ранжирования и нормализации.

6 Булев индекс

Внутреннее представление документов после токенизации

После прошлых ЛР корпус представлен файлом `tokens.txt`, где каждая строка соответствует одному токену и имеет вид `docId<TAB>token`. Таким образом, документ задаётся не «телом текста», а потоком пар `(docId, token)`. На этапе индексации токены нормализуются по требованию ТЗ: понижается капитализация. В реализации это сделано функцией `to_lower_ascii()`, т.е. гарантированно приводится к нижнему регистру латиница (ASCII).

Алгоритм построения индекса

Индекс строится без деревьев и хэш-таблиц, поэтому все структуры основаны на `std::vector` + сортировка + линейные проходы:

1. Чтение `tokens.txt` и накопление массива пар `pairs`: `(term, docId)`. Одновременно считаются: `total_tokens`, `max_doc` и суммарная длина токенов `sum_term_len`.
2. Формирование прямого индекса (forward): из `ir_lr2.documents.json` берутся `url_norm` (по порядку элементов массива), а заголовок восстанавливается эвристикой из URL.
3. Сортировка `pairs` по ключу `(term, docId)`.
4. Линейный проход по отсортированному `pairs`: для каждого терма собирается список уникальных `docId` (повторы отсекаются), вычисляется `df` и записывается словарь (DICT) + «blob» списков документов (POSTINGS).
5. Запись бинарного файла `index.bin` в секционном формате.

Выбранный метод сортировки: достоинства и недостатки

Используется `std::sort` (интроспективная сортировка, на практике $O(n \log n)$) по ключу `(term, docId)`. Плюсы для задачи индексации:

- Простая реализация без `map/unordered_map`, соответствует ограничениям.
- Хорошая локальность данных: после сортировки все вхождения терма лежат подряд, что позволяет строить postings одним линейным проходом.
- Легко обеспечить упорядоченность `docId` внутри postings и удаление повторов.

Минусы:

- Требуется хранить все пары `(term, docId)` в памяти: память растёт линейно от числа токенов.
- Время доминируется сортировкой: при росте данных в 10–1000 раз именно $n \log n$ станет основной стоимостью.

Для больших коллекций типичный путь ускорения — внешняя сортировка и/или блочная индексация.

Бинарный формат `index.bin`

Формат сделан расширяемым: файл состоит из заголовка, таблицы секций и самих секций. Все числа записываются в little-endian.

Header (фиксированная часть, 20 байт):

- `magic[4]`: ASCII-строка `"IRIX"` (4 байта)
- `version`: `u32` (4 байта), текущая версия = 1
- `section_count`: `u32` (4 байта)
- `section_table_offset`: `u64` (8 байт) — смещение таблицы секций от начала файла

SectionTable[section_count] (по 24 байта на запись): каждая запись:

- `type`: `u32` (4 байта) — тип секции
- `flags`: `u32` (4 байта) — флаги/опции (в текущей версии 0)
- `offset`: `u64` (8 байт) — смещение секции от начала файла
- `size`: `u64` (8 байт) — размер секции в байтах

Секции (type):

- **META** (`type=4`): простая статистика для отчёта и отладки:
  - `docs_count`: `u32`
  - `total_tokens`: `u64`

  - `unique_terms`: `u32`
  - `avg_term_len`: `f64`
  - `build_ms`: `f64`

- **DICT** (`type=1`): словарь термов:
  - `term_count`: `u32`
  - далее `term_count` записей:
    * `term_len`: `u16`
    * `term[term_len]`: байты UTF-8 (как есть)
    * `df`: `u32` — document frequency
    * `postings_offset`: `u64` — смещение внутри секции POSTINGS

- **POSTINGS** (`type=2`): поток списков документов:
  - для каждого терма хранится `df` значений `u32 docId` (отсортированы по возрастанию, без повторов).
  - начало списка для терма определяется `postings_offset` из DICT.

- **FORWARD** (`type=3`): прямой индекс (для выдачи):
  - `docs_count`: `u32`
  - далее `docs_count` записей:
    * `url_len`: `u32`, затем `url[url_len]`
    * `title_len`: `u32`, затем `title[title_len]`

Почему формат расширяемый Расширяемость обеспечивается комбинацией:

- поля `version` в заголовке;
- таблицы секций: можно добавлять новые секции (например, частоты `tf`, позиции, skip-указатели, компрессию, кэш заголовков и т.д.), не ломая чтение старых;
- поля `flags` и переменные размеры секций позволяют эволюционировать формат «мягко».

Результаты индексации и расчёты

В ходе эксперимента по построению булева индекса использовался корпус из 34 583 документов, подготовленный на этапе предыдущей лабораторной работы. Общее количество токенов в корпусе составляет 25 738 456, при среднем числе 1 259 токенов на документ. Средняя длина одного токена равна 10,7 байт, что соответствует характеристикам корпуса, полученным на этапе токенизации.

Индексация выполняется за линейный проход по потоку токенов с последующей сортировкой пар `(term, docId)`. Экспериментально измеренная производительность реализации составляет порядка $9{,}7 \cdot 10^5$ токенов в секунду. При сохранении данной скорости для полного корпуса общее время построения индекса оценивается приблизительно в 26,4 секунды. В пересчёте на один документ это составляет около 0,76 мс/док, что указывает на хорошую масштабируемость решения при росте количества документов.

Если рассматривать скорость индексации относительно объёма входных данных, то поток токенов эквивалентен примерно $25\,738\,456 \cdot 10{,}7 \approx 2{,}75 \cdot 10^8$ байт ($\approx$ 262,6 МиБ) очищенного текстового материала. Таким образом, среднее время обработки составляет порядка 0,10 мс на 1 КБ текста. Это показывает, что время работы индексации в текущей реализации определяется преимущественно вычислительными операциями и сортировкой, а не вводом-выводом.

В целом полученные результаты подтверждают, что реализованный алгоритм индексации является достаточно эффективным для корпуса данного размера и подходит в качестве базового решения для булева поиска. При увеличении объёма данных в 10 и более раз основными ограничивающими факторами станут объём оперативной памяти (из-за хранения всех пар `(term, docId)`) и стоимость сортировки $O(n \log n)$, что делает актуальными такие направления оптимизации, как блочная индексация, внешняя сортировка и сжатие списков документов.

Средняя длина терма и сравнение со средней длиной токена В рамках данной работы средняя длина токена по корпусу составляет 10,7 байт и характеризует токены с учётом их частотности во всех документах. При этом средняя длина терма определяется как средняя длина строк, соответствующих уникальным лексическим единицам словаря индекса, без учёта числа их вхождений.

Как правило, средняя длина терма оказывается выше средней длины токена. Это объясняется тем, что короткие служебные слова (предлоги, союзы, частицы) имеют высокую частотность и существенно влияют на среднюю длину токена, тогда как при расчёте средней длины терма все уникальные слова имеют одинаковый вес. В результате редкие, но более длинные лексемы (имена собственные, составные наименования, числовые и технические обозначения) оказывают заметное влияние на среднюю длину терма.

Оптимальность индексации и масштабирование

Реализованный алгоритм индексации является эффективным базовым решением для булева поиска и демонстрирует линейную зависимость времени обработки от объёма входных данных с логарифмической поправкой, обусловленной этапом сортировки. Основными ограничивающими факторами текущей реализации являются потребление оперативной памяти и стоимость сортировки массива пар `(term, docId)`.

При увеличении объёма корпуса в 10 раз объём используемой памяти и время выполнения возрастут примерно пропорционально, тогда как при росте данных в 100–1000 раз затраты на сортировку $O(n \log n)$ и требования к оперативной памяти могут стать критичными. Ввод-вывод при текущих объёмах не является узким местом, однако при дальнейшем масштабировании также начнёт влиять на общее время построения индекса.

Потенциальные направления оптимизации включают переход к блочной или внешней индексации, позволяющей ограничить использование оперативной памяти, а также уменьшение объёма данных за счёт более компактного представления термов и списков документов. Дополнительный выигрыш может быть получен за счёт нормализации Unicode и сжатия списков документов, что позволит уменьшить размер индекса и повысить эффективность последующих операций булева поиска.

7 Булев поиск

В рамках данной работы реализован булев поиск по инвертированному индексу, включающий разбор поисковых запросов, их выполнение и формирование поисковой выдачи. Реализация состоит из двух основных компонентов:

- утилиты командной строки для пакетного выполнения запросов по заранее построенному индексу;
- демонстриратора базовой функциональности поиска (форма запроса и выдача результатов).

Поиск выполняется над бинарным индексом (`index.bin`), загружаемым в память один раз при старте приложения, после чего все запросы обрабатываются в оперативной памяти без повторного чтения данных с диска.

Синтаксис и разбор поисковых запросов

Поддерживаемый синтаксис полностью соответствует техническому заданию:

- пробел или оператор `&&` интерпретируются как логическая операция **И** (AND);
- оператор `||` соответствует логической операции **ИЛИ** (OR);
- оператор `!` реализует логическую операцию **НЕТ** (NOT);
- допускается использование круглых скобок для явного задания приоритетов.

Парсер поисковых запросов реализован с учётом толерантности к пользовательскому вводу: допускается произвольное число пробелов, операторы могут располагаться вплотную к термам или скобкам, а отсутствие явного оператора между соседними термами трактуется как операция AND. Например, запросы

- `футбол матч`,
- `футбол && матч`,
- `футбол матч`

обрабатываются эквивалентно.

Разбор запроса выполняется в два этапа:

1. лексический анализ, в ходе которого входная строка преобразуется в последовательность токенов (термы, операторы, скобки);
2. синтаксический анализ с учётом приоритетов операций (`! > AND > OR`) и скобок, после чего выражение приводится к форме, удобной для вычисления.

Выполнение булевых операций

Каждому терму сопоставляется posting list — отсортированный по `docId` список документов, в которых данный терм встречается. Выполнение булевого запроса сводится к последовательному применению операций над этими списками:

- операция **AND** реализуется как пересечение двух отсортированных списков документов;
- операция **OR** — как объединение списков;
- операция **NOT** — как дополнение множества документов терма относительно полного множества документов корпуса.

Такой подход обеспечивает линейную сложность операций относительно суммарной длины обрабатываемых posting list'ов и позволяет эффективно выполнять запросы даже при большом размере корпуса.

Утилита командной строки

Для демонстрации и тестирования реализована утилита командной строки, выполняющая поиск по индексу для каждого запроса из входного потока (по одному запросу на строку). Пример запуска:

```
./lr7 index.bin --report search_reports.txt --topres 100 \
<queries_search.txt >out.tsv
```

В результате:

- файл `out.tsv` содержит поисковую выдачу (идентификатор документа, заголовок и URL);
- файл `report_search.txt` содержит информацию о времени выполнения запросов;
- в стандартный поток ошибок выводится список самых «медленных» запросов.

Производительность поиска

Скорость выполнения запросов измерялась непосредственно в процессе работы утилиты. Для каждого запроса фиксировалось время выполнения, после чего формировался список наиболее затратных выражений. В ходе эксперимента получены следующие характерные результаты:

```
----TOP 7 slowest queries ----
rank    ms      line    hits    query
1       0.0802  7       3081    (футбол || матч || чемпионат || лига || кубок)
&& !сборная
2       0.0355  5       1064    футбол && (матч || чемпионат)
3       0.0341  6       1454    (игрок || игроки) && команда
4       0.0318  4       405     футбол && !матч
5       0.0253  2       928     футбол матч
6       0.0232  3       3289    футбол || матч
7       0.0092  1       1333    футбол
--------------------------------
```

Типичное время выполнения запроса составляет доли миллисекунды, что соответствует десяткам тысяч запросов в секунду при условии, что индекс уже загружен в память. Наиболее быстрыми являются запросы, содержащие один терм без логических операций, тогда как более сложные выражения требуют дополнительного времени.

Причины замедления сложных запросов

Наибольшее время выполнения наблюдается у запросов, сочетающих несколько факторов:

- большое количество операций `OR` над высокочастотными термами, формирующих крупные промежуточные множества документов;
- использование операции `NOT`, требующей исключения документов из полного множества корпуса;
- вложенные скобочные выражения, увеличивающие число промежуточных операций объединения и пересечения.

Например, запрос

(`футбол` || `матч` || `чемпионат` || `лига` || `кубок`) && `!сборная`

создаёт крупное объединение posting list'ов частотных термов, после чего выполняется операция исключения документов, связанных со словом «сборная», что объясняет его относительно более высокое время выполнения.

Проверка корректности поисковой выдачи

Корректность реализации булевого поиска проверялась следующими способами:

1. сравнением результатов булевых запросов с наивной проверкой наличия термов в документах на ограниченной выборке;
2. проверкой выполнения законов булевой алгебры (коммутативность, дистрибутивность, законы де Моргана);
3. тестированием приоритетов операций и влияния скобок на результат;
4. проверкой устойчивости парсера к различным вариантам пользовательского ввода (лишние пробелы, отсутствие явных операторов).

Результаты эксперимента

Результаты выполнения поисковых запросов сохраненяюся в файле `search_reports.txt`. Этот файл, а также `out.tsv`, используется в качестве экспериментальных данных при анализе производительности и корректности реализованного булевого поиска.

8 Выводы

В ходе выполнения лабораторных работ была построена полноценная базовая поисковая система на специализированном футбольном корпусе: от автоматизированной обкачки и очистки веб-страниц до токенизации, анализа распределения частот (закон Ципфа) и построения инвертированного булева индекса. Сформированный набор данных объёмом 34 583 документа сочетает энциклопедический и публицистический стили, что делает его репрезентативным для задач информационного поиска; выбранная детерминированная токенизация обеспечивает высокую скорость и воспроизводимость, а наблюдаемое соответствие закону Ципфа подтверждает типичную статистическую структуру текста и пригодность корпуса для экспериментов с индексированием и ранжированием.

Анализ работы булевого поиска показал, что реализованная система корректно и быстро обрабатывает запросы различной сложности. Простые запросы из одного терма выполняются практически мгновенно, а более сложные выражения с операциями `OR` и `NOT` требуют большего времени из-за обработки крупных промежуточных списков документов, что является ожидаемым поведением для булева поиска. Полученные экспериментальные результаты подтверждают, что основная вычислительная нагрузка приходится именно на операции над posting list'ами, а не на разбор запросов или ввод-вывод. В целом все этапы работы — от построения корпуса до выполнения поисковых запросов — согласуются между собой и демонстрируют, что разработанная система является работоспособной, масштабируемой для корпуса данного размера и пригодной в качестве основы для дальнейших улучшений и усложнения поисковых моделей.

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] Поисковый робот — что это: определение, работа и основные функции // Sape.ru. — URL: https://www.sape.ru/glossary/poiskoviy-robot/ (дата обращения: 13.11.2025).

[3] Токенизаторы // HuggingFace LLM Course. — URL: https://huggingface.co/learn/llm-course/ru/chapter2/4 (дата обращения: 24.11.2025).

[4] Основы полнотекстового поиска в ElasticSearch. Часть вторая // Habr.com. — URL: https://habr.com/ru/companies/sportmaster$_{l}ab/articles$/756270/(: 28.11.2025).

[5] Закон Ципфа // Википедия. — URL: https://ru.wikipedia.org/wiki/Закон(: 01.12.2025).

[6] Boolean Search: Meaning, Importance and Boolean Operators // GeeksforGeeks.org. — URL: https://www.geeksforgeeks.org/hr/boolean-search-meaning-importance-and-boolean-operators/ (дата обращения: 12.12.2025).