Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Отчет по лабораторным работам по курсу "Информационный поиск"

Студент: Н. А. Медведев
Преподаватель: А. А. Кухтичев
Группа: М8О-401Б-22
Дата: 26.12.2025
Оценка:
Подпись:

Москва, 2025

Содержание

1 Добыча корпуса документов

1.1 Выбор источников и цель корпуса

Для формирования корпуса использовались два открытых русскоязычных источника:

- **ru.wikisource.org (Викиисточник)** — коллекция текстов (произведения, документы), доступная через MediaWiki API;
- **lib.ru (Библиотека Мошкова)** — архив художественных и публицистических текстов, организованный как набор HTML-страниц с каталогами и произведениями.

Целью было собрать корпус порядка **35 000 документов**: **26 000** из Викиисточника и **9 000** из lib.ru. Такая комбинация источников обеспечивает разнообразие жанров и стилей при сохранении единого языка (русский), что удобно для последующих этапов (токенизация, стемминг, закон Ципфа и булев поиск).

1.2 Конфигурация сбора

Сбор настраивается через YAML-файл. Ключевые параметры:

- подключение к MongoDB: `uri = mongodb://localhost:27017`, база `lab_corpus`;
- User-Agent: `NikitaLabCrawler/1.0 (edu; contact: ...)`;
- параллелизм: `threads = 6`;
- задержка между запросами: `delay_seconds = 0.5`;
- таймаут: `timeout_seconds = 20`;
- число повторов при ошибках: `max_retries = 3`;
- целевые лимиты: `wikisource_target = 26000`, `libru_target = 9000`.

1.3 Формирование списка URL (seed)

Перед загрузкой документов в очередь добавляются стартовые URL (seed) для каждого источника.

Wikisource (MediaWiki API). Для Викиисточника используется механизм `list=allpages` через API `https://ru.wikisource.org/w/api.php`. Скрипт последовательно выгружает названия страниц пространства имён `apnamespace=0` и конструирует URL вида:

$$\texttt{https://ru.wikisource.org/wiki/} < \mathit{Title} >$$

Полученные URL добавляются в очередь до достижения заданного лимита (в реализации — до `wikisource_target * 2`, чтобы компенсировать возможные отсеивания и ошибки).

lib.ru (список стартовых страниц). Для lib.ru используется фиксированный список стартовых страниц (главная, каталоги, буквенные индексы, страницы авторов). Далее, в процессе обхода, из HTML дополнительно извлекаются ссылки на другие страницы домена `lib.ru`.

1.4 Сохранение в MongoDB

Корпус хранится в MongoDB и состоит из двух основных коллекций:

- `queue` — очередь заданий на обход страниц;
- `documents` — сырые загруженные документы (HTML) и метаданные.

Каждый документ в `documents` содержит, как минимум:

- `url`, `url_norm` (нормализованный URL без фрагмента);
- `source` (например, `wikisource_ru` или `libru`);
- `fetched_at` (UNIX-время загрузки);
- `raw_html` (исходная HTML-страница);
- `etag`, `last_modified` (если отдал сервер);
- `content_hash` (SHA-256 от HTML, используется для контроля изменений).

1.5 Очистка HTML и формирование коллекции с текстом

После загрузки документов выполняется отдельный этап очистки HTML-страниц скриптом `clean_to_mongo.py`. Он читает `documents` и записывает результат в `documents_clean`.

Очистка реализована через `BeautifulSoup` и включает:

- удаление служебных блоков: `script`, `style`, `noscript`, `header`, `footer`, `nav`, `aside`, `form`;
- удаление HTML-комментариев;
- извлечение текстового содержимого с разделителем пробелом;
- нормализацию пробельных символов (`\s+` → пробел).

Итоговая коллекция `documents_clean` хранит поле `clean_text`, пригодное для токенизации и построения индексов.

1.6 Индексация и уникальность

Для предотвращения дублей применяется нормализация URL и уникальные индексы по `url_norm`. В `documents_clean` создаются индексы:

- уникальный индекс по `url_norm`;
- индекс по `source`.

Таким образом, на выходе получается корпус текстов, унифицированный по структуре хранения и пригодный для дальнейших этапов информационного поиска. Выгруженная коллекция documents в формате csv:
`https://drive.google.com/file/d/1ogOoUKZrLalVgb3ritZefvXTcZ13YnHn/view?usp=sharing`

2 Поисковый робот

2.1 Общая схема

Поисковый робот реализован в виде многопоточного краулера `multi_scraler.py`, который:

1. формирует очередь URL (seed) для выбранных источников;
2. извлекает из очереди следующую задачу, учитывая приоритеты и лимиты по источникам;
3. загружает страницу по HTTP;
4. сохраняет HTML и метаданные в MongoDB;
5. при необходимости добавляет новые URL в очередь (для lib.ru);
6. повторяет процесс до достижения целевого размера корпуса.

2.2 Нормализация URL

Для контроля дублей и устойчивости к различным представлениям одного и того же адреса используется функция нормализации:

- удаление фрагмента (`#...`) через `urldefrag`;
- приведение схемы и домена к нижнему регистру;
- сохранение пути и query-части;
- сборка нормализованного URL `url_norm`.

Именно `url_norm` используется как ключ в очереди и коллекции документов.

2.3 Структура очереди и статусы задач

Очередь хранится в коллекции `queue`. Запись очереди содержит:

- `url`, `url_norm`, `source`;
- `priority` (меньше — важнее; для `wikisource_ru` установлен приоритет 1, для `libru` — 2);
- `status`: `pending`, `in_progress`, `done`, `error`;
- `attempts` (число попыток);
- `next_fetch_at` (время, когда задачу можно брать снова);
- `updated_at`.

Алгоритм обработки ошибок устроен так:

- при временной ошибке задача возвращается в `pending` с увеличенным `attempts` и сдвигом `next_fetch_at` (retry);
- при превышении `max_retries` задача фиксируется в статус `error`.

2.4 Выбор следующей задачи

Выбор URL для скачивания выполняется функцией `get_next_job()`. Логика учитывает:

- **лимит документов по источнику** (краулер стремится набрать заданные `wikisource_target` и `libru_target`);
- **приоритет** (сначала обрабатываются задачи с меньшим `priority`);
- **время доступности** (берутся только задания, у которых `next_fetch_at` уже наступил).

При взятии задачи она атомарно переводится в `in_progress` через `find_one_and_update`, что предотвращает гонки между потоками.

2.5 HTTP-загрузка и кеширование

Загрузка выполняется через `requests.Session`. Если документ уже встречался, используются условные заголовки:

- `If-None-Match` (ETag),
- `If-Modified-Since` (Last-Modified).

Обработка ответов:

- **304 Not Modified**: обновляется только `fetched_at`, задача помечается как `done`;
- **200 OK**: сохраняется HTML, вычисляется `content_hash` (SHA-256), при изменении контента обновляется запись документа;
- **прочие коды**: выполняется повтор или перевод в `error` при превышении лимита попыток.

2.6 Извлечение новых ссылок (lib.ru)

Для источника `libru` реализовано расширение фронтира: после успешной загрузки страницы из HTML извлекаются ссылки из атрибутов `href="..."`. Далее:

- ссылки приводятся к абсолютному виду через `urljoin`;
- отбрасываются `mailto:`, `javascript:` и фрагменты;
- в очередь добавляются только URL домена `lib.ru`;
- уникальность обеспечивается индексом `url_norm`.

2.7 Многопоточность и уважение к источникам

Краулер использует пул потоков (`ThreadPoolExecutor`) с числом воркеров `threads=6`. Чтобы не перегружать источники, введена задержка `delay_seconds=0.5` после обработки каждой задачи, а также таймауты и ограничение числа повторов.

2.8 Индексы MongoDB

Для производительности и предотвращения дублей создаются индексы:

- `documents.url_norm` (unique);
- `queue.url_norm` (unique);
- составные индексы в `queue` по `status`, `next_fetch_at`, `priority`, `source`.

2.9 Результат работы

В результате работы робота формируется коллекция `documents` с HTML и метаданными, а после очистки — коллекция `documents_clean` с полем `clean_text`, являющимся основой для токенизации, построения статистик и индексов в последующих лабораторных работах.

3 Стемминг

3.1 Назначение стемминга

Стемминг (stemming) применяется для приведения словоформ к общей основе (стему), чтобы уменьшить размер словаря и повысить полноту поиска. В русском языке одна и та же лексема может встречаться в большом числе форм (падежи, числа, времена, суффиксы), поэтому без нормализации индекс разрастается, а релевантные документы могут не находиться из-за несовпадения словоформ.

В рамках работы стемминг выполняется **после токенизации** и используется при построении статистик и индекса: в индекс попадает уже нормализованная форма терма.

3.2 Реализация

Стеммер реализован функцией `stem_ru(token)`, которая выполняет упрощённое эвристическое усечение суффиксов (rule-based suffix stripping). Подход не является полноценным морфологическим анализом: алгоритм не определяет часть речи и не использует словари, а действует по набору заранее заданных правил.

Алгоритм включает три основные стадии:

1. **Нормализация символа «ё».** Для стабильности индекса выполняется замена «ё/Ё» на «е/Е» (функция `normalize_yo`). Это снижает число дублей, когда один и тот же термин встречается в тексте в двух написаниях.

2. **Отсечение коротких токенов.** Для коротких слов стемминг часто приводит к переусечению и потере смысла, поэтому введено ограничение: токены короче 8 байт (UTF-8) возвращаются без изменений (функция `too_short_for_stem`).

3. **Поиск и удаление суффикса.** Используется список распространённых русских суффиксов/окончаний для существительных, прилагательных и глаголов. Список сортируется по убыванию длины, поэтому сначала пытаемся удалить наиболее длинные суффиксы (например, `«ностями»`, `«аться»`), и только затем короткие (например, `«а»`, `«е»`). Если токен оканчивается на суффикс и после удаления остаётся **минимум 4 символа основы** (проверка `token.size() > s.size() + 4`), суффикс удаляется, после чего цикл завершается.

3.3 Дополнительная нормализация

После удаления суффикса выполняется обработка мягкого знака: если основа оканчивается на «ь», он удаляется (функция `strip_soft_sign`). Это дополнительно уменьшает вариативность (например, *«дверь»* и производные формы чаще приводятся к общей основе).

3.4 Пример работы (типовые случаи)

Типовые эффекты стемминга:

- глагольные окончания: *«делается»* → *«дела»* (удаление `«ется»`);
- абстрактные существительные: *«сущность»*, *«сущностью»* → *«сущн»*/*«сущност»* (удаление `«остью»`/`«ность»` в зависимости от формы);
- прилагательные: *«большими»* → *«больш»* (удаление `«ими»`).

Следует учитывать, что из-за эвристического характера алгоритма возможны переусечения: стеммер не гарантирует получение леммы, а выдаёт удобную для индексации основу.

3.5 Достоинства и ограничения

Достоинства:

- высокая скорость (операции сравнения суффиксов и усечение строки);
- отсутствие словарей и внешних зависимостей;
- заметное уменьшение числа уникальных токенов в словаре.

Ограничения:

- алгоритм не учитывает морфологию и часть речи, поэтому возможны ошибки нормализации;
- правило «не стеммить короткие токены» снижает риск ошибок, но часть вариативности остаётся;
- усечение работает по байтам UTF-8, что требует аккуратности при подборе порогов (используется консервативное ограничение, чтобы не разрушать слово).

4 Закон Ципфа

4.1 Суть закона Ципфа для текстов

Для естественных языков характерно неравномерное распределение частот слов: небольшое число термов встречается очень часто, а подавляющее большинство — редко. Это эмпирическое наблюдение формализуется **законом Ципфа**:

$$f(r) \approx \frac{C}{r^s},$$

где r — ранг слова (1 для самого частотного), $f(r)$ — частота слова с рангом r, C — константа, s — параметр (обычно близок к 1 для больших корпусов).

Проверка закона Ципфа в лабораторной работе позволяет:

- оценить качество и "естественность" собранного корпуса;
- увидеть вклад частотных служебных слов и наличие длинного "хвоста" редких термов;
- подготовить данные для анализа словаря и индекса (объём, разреженность).

4.2 Формирование распределения частот

Для построения распределения используется коллекция `documents_clean`, содержащая поле `clean_text` — очищенный текст документа. Из MongoDB выгружается только это поле (projection), что уменьшает сетевые и памятные затраты:

```
projection: {clean_text: 1, source: 1}
```

Также применяется фильтр: берутся только документы, где `clean_text` существует и не пустой.

4.2.1 Токенизация для подсчёта частот

Важно отметить, что токенизация в данной части используется **только для подсчёта частот**, а не как финальный модуль поиска. Функция `tokenize_and_count()` последовательно сканирует строку UTF-8 и выделяет токены, состоящие из:

- латиницы/цифр (ASCII),
- кириллицы (двухбайтовые UTF-8 символы диапазонов `D0 xx` и `D1 xx`).

Дополнительные правила:

- **ASCII приводится к нижнему регистру** (опция `ascii_to_lower`);
- **короткие токены отбрасываются** (минимум `min_token_bytes = 2`);
- **дефис внутри слова сохраняется**, если по обе стороны стоят буквенно-цифровые символы (`keep_hyphen_inside = true`).

Для каждого выделенного токена увеличивается счётчик в хэш-таблице:

```
freqs[token] += 1
```

В результате получается отображение `term` → `frequency`.

4.3 Построение файла `zipf.csv`

После обработки всех документов значения частот собираются в массив, сортируются по убыванию, после чего формируется CSV-файл вида:

`rank,freq`

$$(1, f_1), (2, f_2), \ldots$$

где $f_1 \geq f_2 \geq \ldots$.

Таким образом, в `zipf.csv` хранится уже не список термов, а **упорядоченное распределение частот**, достаточное для проверки закона Ципфа и аппроксимации параметров C и s.

4.4 Аппроксимация параметров закона Ципфа

Для оценки параметров модели используется логарифмирование исходной формулы:

$$f(r) = \frac{C}{r^s} \quad \Rightarrow \quad \ln f = \ln C - s \ln r.$$

Обозначим:

$$x = \ln r, \quad y = \ln f,$$

тогда получаем линейную зависимость:

$$y = ax + b, \quad \text{где } a = -s,\ b = \ln C.$$

В Python-скрипте параметры a и b находятся методом наименьших квадратов через `numpy.polyfit(x, y, 1)`. Затем вычисляются:

$$s = -a, \quad C = e^b.$$

4.5 Анализ распределения частот

На рисунке представлено распределение частот терминов корпуса в логарифмических координатах, а также аппроксимация законом Ципфа. По оси абсцисс отложен ранг терма, по оси ординат — его частота появления в корпусе.

Экспериментальные точки образуют характерную убывающую кривую с почти линейным участком в log–log масштабе, что свидетельствует о выполнении закона Ципфа для данного корпуса. Наибольшие частоты соответствуют небольшому числу термов, тогда как подавляющее большинство слов имеет низкую частоту встречаемости, формируя длинный «хвост» распределения.

Аппроксимирующая кривая хорошо согласуется с экспериментальными данными на среднем диапазоне рангов, который является наиболее информативным для анализа корпуса. Отклонения в области малых рангов объясняются высокой долей служебных и общеязыковых слов, а расхождения в области больших рангов — наличием редких и уникальных термов, чувствительных к особенностям токенизации и очистки текста.

Оценка параметра степени показывает значение $s \approx 1.19$, что близко к типичным значениям для корпусов естественного языка. Это подтверждает, что собранный корпус обладает естественной статистической структурой и подходит для задач информационного поиска и построения текстовых индексов.

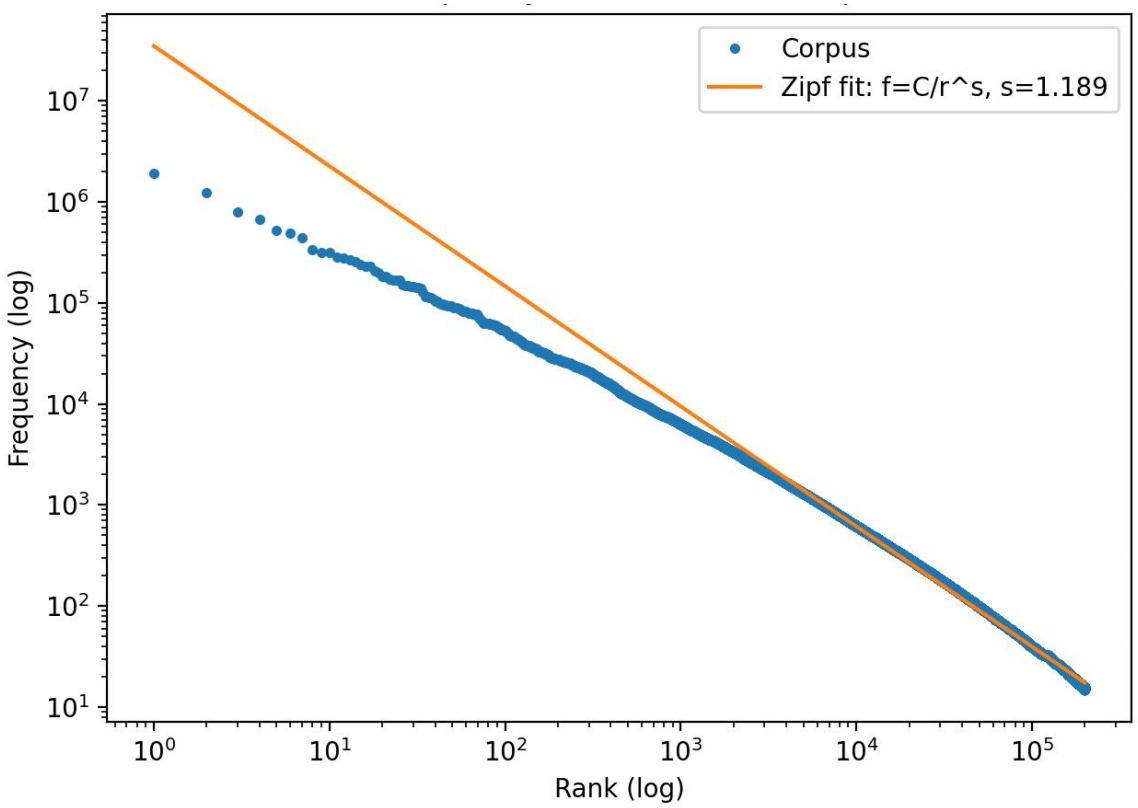

Рис. 1: Распределение частот терминов корпуса и аппроксимация законом Ципфа

5 Булев индекс и поиск

5.1 Нулевая модель булевого поиска: реализация через собственные структуры

Реализованный булев поиск относится к нулевой (базовой) модели информационного поиска, в которой документ либо удовлетворяет запросу, либо нет, без вычисления весов и ранжирования. При этом основное внимание уделяется не теоретической формулировке модели, а эффективности реализации операций над множествами документов.

В основе поиска лежит собственная реализация булевого индекса и операций над постинг-листами, без использования стандартных контейнеров множеств (`std::set`, `std::unordered_set`). Вместо этого используются:

- хэш-таблица `unordered_map` для отображения термов в постинг-листы;
- упорядоченные векторы идентификаторов документов (`vector<uint32_t>`).

5.2 Организация постинг-листов

Каждому терму соответствует постинг-лист — отсортированный массив идентификаторов документов. Отсортированность достигается естественным образом за счёт последовательной индексации документов, где `doc_id` монотонно возрастает. Это позволяет:

- избежать дополнительной сортировки после построения индекса;
- эффективно применять линейные алгоритмы пересечения и объединения.

При добавлении термов выполняется проверка последнего элемента постинг-листа, что гарантирует отсутствие дубликатов документов внутри одного списка.

5.3 Модель вычисления запроса

Поиск выполняется методом `search_and(query)` и интерпретирует запрос как конъюнкцию слов. Для каждого токена запроса формируется **множество документов**, в которое включаются:

- документы, содержащие точную форму токена;
- документы, содержащие его стем.

Объединение этих двух множеств реализуется функцией `postings_union`, работающей напрямую по отсортированным массивам. Таким образом, для одного терма формируется результирующее множество документов без использования абстрактных структур множеств.

Далее для всех термов запроса применяется последовательное пересечение множеств (`postings_intersect`). Алгоритм использует два указателя и выполняется за линейное время от суммарной длины списков.

5.4 Инкрементальное пересечение и досрочное завершение

Результат поиска формируется инкрементально:

- для первого терма инициализируется текущее множество документов;
- для каждого следующего терма выполняется пересечение с текущим результатом;
- если на любом шаге результат становится пустым, вычисление прекращается.

Такой подход минимизирует количество операций и особенно эффективен для запросов, содержащих редкие термы.

5.5 Преимущества выбранного подхода

Выбранная реализация нулевого булевого поиска обладает следующими свойствами:

- минимальные накладные расходы по памяти по сравнению с `std::set`;
- предсказуемая временная сложность операций AND/OR;
- контроль над внутренним представлением данных и алгоритмами;
- простота отладки и расширения (например, добавление OR/NOT или парсинга выражений).

6 Как работает поиск в приложении

6.1 Режим работы программы

Поисковое приложение реализовано как консольная утилита, работающая в интерактивном режиме: после запуска система строит булев индекс по документам из MongoDB, затем принимает поисковые запросы пользователя и выводит результаты.

Параметры подключения и построения индекса передаются через аргументы командной строки:

- URI MongoDB;

- имя базы данных;
- имя коллекции (как правило, `documents_clean`);
- опционально — ограничение `limit` на число индексируемых документов (для ускорения тестов).

После построения индекса программа печатает краткую сводку:

`docs = ..., terms = ...`

что позволяет контролировать размер индексируемого корпуса и словаря.

6.2 Обработка пользовательского запроса

Программа запрашивает строку запроса в цикле:

- пустая строка завершает работу;
- непустая строка обрабатывается поисковым движком.

Запрос передаётся в метод `engine.search_and(q)`. Данный метод реализует булев поиск в нулевой модели: запрос интерпретируется как конъюнкция (AND) всех слов, присутствующих в строке. На выходе получается список `doc_id` — внутренних идентификаторов документов, удовлетворяющих запросу.

6.3 Формирование и вывод результатов

Для каждого найденного `doc_id` система восстанавливает метаданные документа из массива `meta`:

- `Mongo _id` (идентификатор документа в базе);
- `title` (если присутствует);
- `source` (источник: `wikisource_ru` / `libru`);
- `url` (исходный адрес страницы).

Для повышения наглядности результатов дополнительно выводится **фрагмент текста (snippet)**. Если MongoDB-`_id` известен, вызывается метод загрузчика `fetch_snippet_by_oid_hex(mo 200)`, который извлекает из базы короткую текстовую выдержку длиной до 200 символов. Сниппет позволяет быстро понять контекст совпадения без открытия документа целиком.

Для удобства просмотра вывод ограничен первыми 10 документами, при этом дополнительно печатается сообщение о наличии оставшихся результатов:

`... and N more`

6.4 Демонстрация работы

На рисунках ниже приведены примеры работы системы: построение индекса, ввод запросов и вывод найденных документов с метаданными и сниппетами.

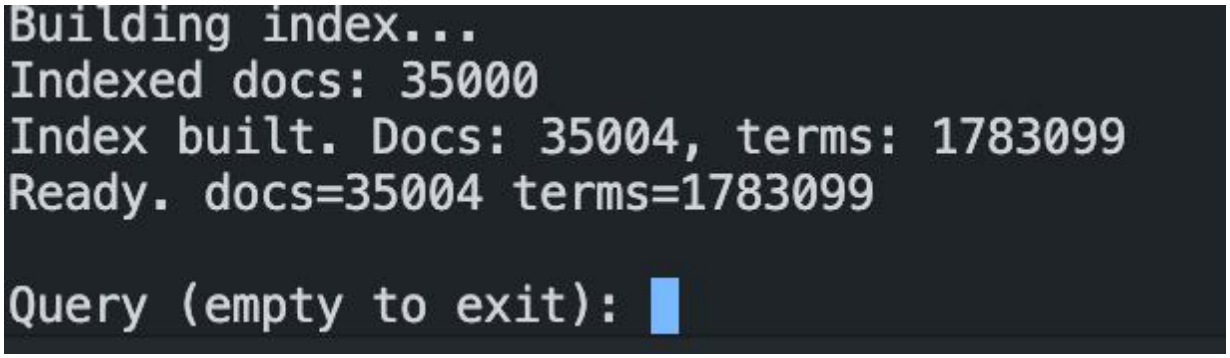

Рис. 2: Пример выполнения запроса и вывод результатов

Рис. 3: Пример выполнения запроса и вывод результатов

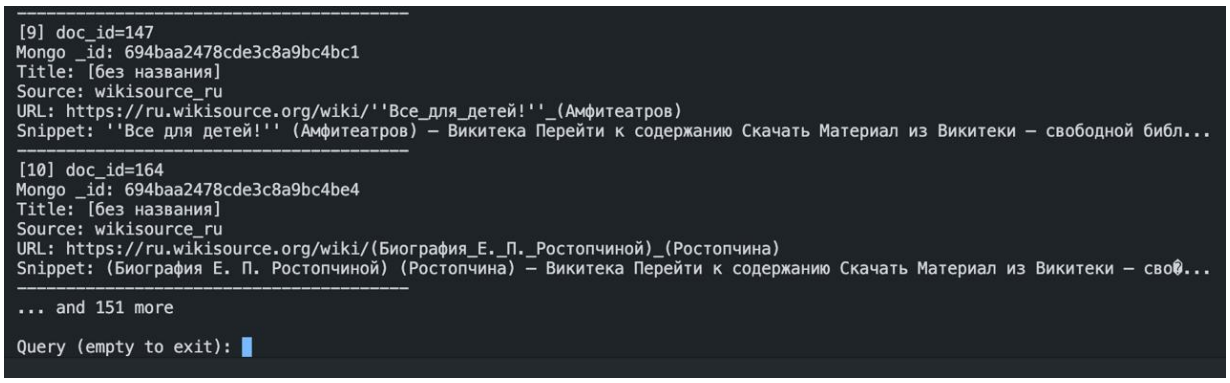

Рис. 4: Пример выполнения запроса и вывод результатов

7 Выводы

В ходе выполнения лабораторных работ был реализован полный базовый конвейер информационного поиска: от сбора и очистки текстового корпуса до построения булевого индекса и выполнения поисковых запросов.

Был разработан многопоточный поисковый робот для сбора документов из открытых источников, реализованы механизмы нормализации URL, управления очередью и повторных загрузок, а также этап очистки HTML и выделения чистого текстового содержимого.

Для корпуса были выполнены токенизация и стемминг русскоязычных текстов, что позволило снизить размер словаря и повысить полноту поиска. Анализ распределения частот терминов показал соответствие корпуса закону Ципфа, что подтверждает его естественную статистическую структуру.

На основе корпуса был построен булев индекс с использованием собственных структур данных и эффективных алгоритмов работы с постинг-листами. Реализована нулевая модель булевого поиска, в которой запрос интерпретируется как конъюнкция термов, а поиск осуществляется за счёт операций пересечения и объединения множеств документов.

Реализованная система демонстрирует корректную работу и может служить основой для дальнейшего развития, включая поддержку сложных булевых выражений, ранжирование результатов и расширенные методы обработки естественного языка.

8 Список литературы

1. Солтон Дж., Макгилл М. Введение в современный информационный поиск. — М.: Мир, 1983. — 416 с.

2. Кузнецов С. Д. Основы информационного поиска. — М.: Физматлит, 2009. — 320 с.

3. Мэннинг К., Рагхаван П., Шютце Х. Введение в информационный поиск. — М.: Диалектика, 2011. — 528 с.