Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Информационный поиск»

Студент: А. Р. Прокофьев
Преподаватель: А. А. Кухтичев
Группа: М8О-401Б-22

Дата: 29.12.2025
Оценка:
Подпись:

Москва, 2025

1 Задание

Разработать поисковую систему по собственному корпусу документов морской тематики, включающую:

- поискового робота (crawler) для сбора документов из открытых источников;
- хранение документов в базе данных;
- индексатор с токенизацией и стеммингом;
- построение булевого индекса;
- реализацию булевого поиска с поддержкой операторов AND, OR, NOT и скобок;
- экспериментальный анализ свойств корпуса (в том числе закон Ципфа).

2 Корпус и источники данных

Корпус собран по морской тематике и состоит из двух независимых источников. Я специально выбрал морскую тематику, потому что по ней легко найти много текстов, но при этом термины достаточно специфичные (встречаются слова про суда, порты, океан, офшор, экологию и т.д.), что удобно для проверки поиска и индексации.

- **Wikipedia EN** — статьи английской Википедии из тематических категорий, связанных с морем, океанографией, судоходством, морской экологией и портовой инфраструктурой. Этот источник в основном даёт “энциклопедические” тексты: они обычно длинные, структурированные, содержат определения, перечисления, факты и ссылки на связанные понятия. Это полезно, потому что в таких текстах часто встречаются устойчивые термины и много разных словоформ.
- **MarineLink** — новостные публикации портала MarineLink (раздел Maritime News), посвящённые отраслевым событиям: суда, офшор, порты, логистика, судостроение, безопасность, экология. Этот источник даёт более “живые” новости: тексты короче, часто содержат имена компаний, названия судов, даты, географические объекты и отраслевую лексику. В таких документах обычно меньше “общих” слов и больше специфики.

Корпус является тематически однородным (везде морская тематика), но источники различаются по стилю и структуре. Википедия чаще даёт подробные описания и справочную информацию, а новости MarineLink — события и контекст вокруг них.

Это удобно для лабораторных работ, потому что можно проверить, как система работает на документах разного типа: где-то текст длинный и насыщенный терминами, а где-то короткий, но содержит много конкретных сущностей.

Также важно, что источники независимые: это снижает риск того, что корпус получится “однообразным”. В дальнейшем это помогает лучше увидеть проблемы токенизации (например, дефисы, сокращения, имена), стемминга (словоформы и редкие слова) и булевого поиска (когда нужный термин встречается в разных стилях текста).

3 Формат сырых документов и выделение текста

Сырые документы сохраняются в виде отдельных файлов в формате JSON (один файл — один документ). Такой формат я выбрал, потому что он простой, удобен для отладки и легко читается любым скриптом. При необходимости можно быстро посмотреть любой документ руками и понять, что именно скачалось.

Каждый документ содержит основные поля:

- `source` — источник (wikipedia_en / marinelink), чтобы потом можно было разделять документы по источникам и анализировать их отдельно;
- `url` — нормализованный URL. Нормализация нужна, чтобы избежать дублей (например, если одна и та же страница встречается с разными параметрами или с якорями);
- `title` — заголовок страницы, он полезен как мета-информация и для отображения в поисковой выдаче;
- `raw_html` — исходный HTML-код страницы, то есть “как скачали” (это нужно для воспроизводимости и для того, чтобы при необходимости можно было заново выделить текст другими правилами);
- служебные поля (например, время обкачки в Unix timestamp, идентификатор документа, иногда дополнительные заголовки), которые помогают понимать, когда документ был скачан и можно ли его переобкачивать.

Дальше для лабораторных работ по индексации и поиску нужен именно текст, поэтому из HTML выделяется plain-text. Я делал это отдельным шагом, чтобы:

- отделить сбор данных от обработки текста;
- получить единый формат документов для всех следующих лабораторных;

- уменьшить объём данных (в HTML много лишнего, а для поиска нужен текст).

При извлечении текста применялись базовые правила очистки:

- удаляются теги `script/style/noscript`, чтобы в тексте не было кода, стилей и других технических вставок;
- для Wikipedia извлекается основной контент статьи (без навигации, меню и служебных блоков);
- для MarineLink извлекается заголовок и основной текст новости, а также убираются элементы шаблона сайта (например, блоки рекомендаций и похожих новостей).

Важно, что извлечённый текст сохраняется отдельно как `.txt` в кодировке UTF-8 (один файл — один документ). Это удобно по нескольким причинам:

- дальше можно обрабатывать документы потоково (читать по одному файлу), не загружая всё в память;
- легко пересобирать индекс или пересчитывать статистики, не обращаясь заново к “сырым” HTML;
- текстовые файлы проще использовать для отладки токенизации/стемминга, потому что видно исходный материал.

Таким образом, получилось два уровня данных: “сырой” слой (HTML в JSON) для воспроизводимости и “чистый” слой (plain-text), который используется в последующих лабораторных работах.

4 Статистика корпуса

Характеристика	**Wikipedia EN**	**MarineLink**
Количество документов	32 991	5 151
Размер сырых данных (HTML)	2.95 GB	368.6 MB
Размер выделенного текста	544.8 MB	27.8 MB
Средний размер сырого документа	93.7 KB	73.3 KB
Средний размер текста	16.9 KB	5.53 KB
Коэффициент “сжатия” (text/raw)	18.0%	7.55%

Таблица 1: Статистика корпуса по источникам

Итого в корпусе 38 142 документа общим объёмом 3.31 GB сырых HTML-данных, из которых извлечено 572.6 MB текста.

5 Существующие поисковики и примеры запросов

Для выбранных источников доступны внешние поисковые системы:

- встроенный поиск Wikipedia (по статьям en.wikipedia.org);
- поиск Google с ограничениями `site:en.wikipedia.org` и `site:marinelink.com`.

Примеры запросов и наблюдаемые недостатки выдачи:

1. **Запрос:** `site:en.wikipedia.org ocean pollution microplastics`
   Недостатки: в выдаче часто доминируют общие статьи (“Ocean”, “Pollution”) без ранжирования по тематике; встречаются страницы-списки и служебные страницы, которые хуже подходят как ответы.
2. **Запрос:** `site:marinelink.com offshore wind installation vessel`
   Недостатки: результаты могут содержать несколько очень похожих новостей, а также материалы не из раздела Maritime News; часть результатов зависит от SEO и даты публикации, а не от точного соответствия запросу.
3. **Запрос:** `site:en.wikipedia.org tension-leg platform offshore`
   Недостатки: при наличии редких терминов выдача может включать нерелевантные страницы, где термины встречаются только в ссылках/примечаниях; иногда полезные статьи находятся глубже из-за особенностей ранжирования.
4. **Запрос:** `site:marinelink.com tanker accident investigation`
   Недостатки: результаты чувствительны к формам слов (accident/accidents), иногда лучше работает поиск по точной фразе; также встречаются страницы с неполным контентом (короткие анонсы).

Вывод: существующие поисковики позволяют подтвердить пригодность корпуса (по обоим источникам есть доступный поиск), однако выдача не всегда удовлетворяет требованиям по точности и контролю логики поиска, что мотивирует реализацию собственной поисковой системы.

6 Краткое описание решения

Поисковая система реализована как набор отдельных программ (утилит) командной строки и простой веб-интерфейс. Такой подход я выбрал специально: каждую часть можно запускать отдельно, проверять результат и при необходимости переделывать, не ломая весь проект.

- **Lab01:** загрузка корпуса из двух источников (Wikipedia EN, MarineLink). На этом этапе я сохраняю “сырые” страницы (HTML) и отдельно выделенный текст. Это нужно, чтобы можно было в любой момент перепроверить качество извлечения текста или повторно обработать данные другими правилами.

- **Lab02:** поисковый робот (crawler), который умеет скачивать документы по конфигу и складывать их в базу данных SQLite. Робот работает с задержкой между запросами, чтобы не перегружать сайт. Также его можно остановить и потом продолжить работу с того места, где он остановился. Для переобкачки предусмотрена проверка изменения документа, чтобы не скачивать одно и то же бесконечно.

- **Lab03–Lab04:** токенизация и стемминг (реализованы на C++). Эти утилиты читают тексты документов и разбивают их на токены по простым правилам. Для стемминга применяется сведение словоформ к основе, чтобы поиск был менее чувствителен к разным формам слова. Дополнительно программы выводят статистику (сколько токенов получилось, средняя длина токена, время работы и скорость).

- **Lab05:** закон Ципфа. На основе частотного словаря строится график распределения частот терминов по рангу в логарифмической шкале и сравнивается с моделью $f(r) = C/r$. В отчёте отдельно описываются причины расхождений (тематика корпуса, смешивание источников, редкие термины и т.д.).

- **Lab07:** построение булевого инвертированного индекса. Индекс хранит для каждого термина список документов, где этот термин встречается. Индекс сохраняется на диск, чтобы поиск мог работать без повторной обработки текста.

- **Lab08:** булев поиск с поддержкой AND/OR/NOT и скобок. Поиск реализован как CLI-утилита (удобно для тестов) и как веб-страница (удобно показывать результат). В веб-интерфейсе выдача группируется по источникам (Wikipedia слева, MarineLink справа), чтобы было проще сравнивать результаты.

Обработка данных организована потоково: документы читаются по одному файлу, а индекс строится без необходимости загружать весь корпус в оперативную память. Это позволяет применять решение к корпусам порядка десятков тысяч документов и выше, при условии что хватает места на диске для сохранения исходных данных и индекса.

7 Лабораторная работа 2. Поисковый робот

1 Цель работы

Цель лабораторной — реализовать поискового робота (crawler), который обходит страницы из заданных источников, скачивает HTML и сохраняет документы в базу данных. Робот должен уметь:

- запускаться с единственным аргументом — путь до YAML-конфига;
- сохранять в БД нормализованный URL, сырой HTML, источник документа и время обкачки (Unix timestamp);
- корректно продолжать работу после остановки;
- периодически переобкачивать документы и обновлять их только если содержимое изменилось.

2 Формат конфига (YAML)

Робот запускается командой вида:

```
python robot.py config.yaml
```

В конфиге есть как минимум две секции:

- `db` — настройки базы данных (например, путь к SQLite-файлу);
- `logic` — настройки логики робота (задержка между запросами, параметры переобкачки и т.п.).

Также в конфиге можно задавать дополнительные параметры, которые нужны для работы: стартовые URL, лимиты, таймауты, user-agent и т.д. Я вынес это в конфиг, чтобы можно было менять поведение робота без изменения кода.

3 Хранение данных

В качестве БД я использовал SQLite, потому что она не требует отдельного сервера и хранится в одном файле, что удобно для запуска на разных машинах. Для объёма уровня десятков тысяч документов этого достаточно.

В базе данных используется две основные таблицы:

- `docs` — таблица уже скачанных документов;

- `frontier` — очередь URL на скачивание и переобкачку.

В таблице `docs` сохраняются поля:

- `url` — нормализованный URL;
- `source` — источник (wikipedia_en / marinelink);
- `html` — сырой HTML страницы;
- `fetched_at` — время обкачки в формате Unix timestamp;
- дополнительные поля для проверки изменений (например, `etag`, `last_modified`, `content_hash`).

4 Нормализация URL

Перед добавлением URL в очередь я нормализую ссылки: убираю якорь (часть после #), стараюсь удалять очевидные трекинговые параметры (например, `utm_*`) и привожу ссылку к единому виду. Это нужно, чтобы одна и та же страница не попадала в базу несколько раз под разными вариантами URL.

5 Остановка и продолжение работы

Робота можно остановить в любой момент (например, Ctrl+C). Поскольку данные пишутся в SQLite по мере работы, состояние не теряется: скачанные документы уже лежат в `docs`, а оставшиеся URL находятся в `frontier`.

При повторном запуске робот продолжает работу: он берёт следующий URL из `frontier` и продолжает обход. Таким образом, не нужно начинать скачивание заново.

6 Переобкачка документов

Чтобы робот мог обновлять документы, я сделал простую переобкачку. Для каждого документа хранится информация, по которой можно понять, изменился он или нет:

- если сервер отдаёт `ETag` или `Last-Modified`, они сохраняются и используются при следующей проверке;
- дополнительно (или вместо этого) считается хеш от HTML, чтобы надёжно определить изменение.

Если документ не изменился, то HTML не перезаписывается, а робот просто фиксирует, что проверка выполнена. Если документ изменился, то в базе обновляются HTML и время обкачки.

7 Итог

В результате получился робот, который скачивает документы из открытых источников, сохраняет “сырые” страницы в базу данных, умеет продолжать работу после остановки и поддерживает переобкачку документов только при изменениях. Это даёт основу для дальнейших лабораторных работ: выделение текста, индексация и поиск.

8 Лабораторная работа 3. Токенизация

1 Правила токенизации

Токенизация сделана максимально простой, чтобы её было легко проверить и чтобы она работала быстро на большом количестве текстов. По сути программа проходит по строке и выделяет “слова” как непрерывные последовательности допустимых символов.

Используются такие правила:

- весь текст переводится в нижний регистр, чтобы `Sea` и `sea` считались одним и тем же;
- токеном считается последовательность букв латиницы и цифр (`a-z, 0-9`);
- все остальные символы считаются разделителями: пробелы, знаки препинания, скобки, кавычки, служебные символы и т.д.;
- токены короче 2 символов отбрасываются, так как они чаще всего создают шум (например, одиночные буквы, обрывки слов и т.п.).

Такой подход хорошо подходит именно для лабораторных работ, потому что он предсказуемый: если посмотреть на текст, можно легко понять, какие токены получится на выходе.

2 Достоинства и недостатки

Плюсы выбранного метода:

- реализация простая и быстрая, её удобно применять в потоковой обработке (файл за файлом);
- после выделения текста из HTML остаётся мало мусора, и токенизация даёт достаточно чистые термы;

- получается воспроизводимый словарь терминов, который удобно использовать дальше при построении индекса.

Минусы и ограничения:

- составные слова и термины с дефисами распадаются на части (например, `deep-sea` превращается в *deep* и *sea*), из-за этого иногда теряется смысл именно как "единого термина";
- аббревиатуры и маркировки обрабатываются не идеально (например, в морской тематике встречаются названия типа `IMO-1234567` или `AHTS`, где дефисы и формат важны);
- числовые токены иногда создают шум (годы, номера, размеры), но полностью выбрасывать числа тоже нельзя, потому что иногда они несут смысл.

Примеры неудачных случаев:

- *"co2"* и *"co-2"* превращаются в разные последовательности, хотя по смыслу это одно и то же;
- *"shipowner's"* $\rightarrow$ *shipowner*, *s* (лишний токен *s* не несёт смысла).

Как можно было бы улучшить правила в будущем:

- добавить отдельное правило для апострофа, чтобы `'s` отрезалось корректно;
- разрешить дефис внутри токена для некоторых случаев (например, если вокруг дефиса буквы);
- сделать фильтрацию чисел по простым эвристикам (например, выбрасывать слишком длинные числа или номера без букв).

3 Статистика

После токенизации по всему корпусу получаются базовые характеристики, которые нужны для последующих лабораторных работ:

- общее количество токенов (сколько всего слов получилось);
- средняя длина токена (примерно показывает "среднюю длину слова" в корпусе).

Для корпуса морской тематики значения находятся в ожидаемом диапазоне: средняя длина токена получается около 5 символов, а общее количество токенов — десятки миллионов. Эти значения логично зависят от того, насколько длинные документы, и от того, насколько сильно в текстах встречаются специальные термины и имена собственные.

9 Лабораторная работа 4. Стемминг

1 Метод

Для английского языка я добавил стемминг — это упрощённое приведение слов к основе. Идея в том, что разные формы одного слова должны восприниматься как “почти одно и то же”. Это полезно для поиска, потому что пользователь может ввести слово в одной форме, а в документах оно встречается в другой.

Стемминг сделан на основе набора правил (суффиксные преобразования). Примерно это выглядит так: *shipping* → *ship*, *pollution* → *pollut*, *fishing* → *fish* и т.п. То есть отрезаются частые окончания и суффиксы, которые обычно не меняют смысл, но меняют форму слова.

2 Влияние на словарь и поиск

После добавления стемминга я заметил два основных эффекта:

- **словарь становится меньше**, потому что разные словоформы склеиваются в одну “основу”;
- **поиск становится более гибким**: запрос по одному слову чаще находит документы, даже если в тексте другая форма (например, *ship* и *shipping*).

При этом у метода есть минус: иногда разные слова могут случайно стать одинаковыми после стемминга. Это происходит потому, что стемминг не понимает смысл, он просто применяет правила к окончаниям. В редких случаях это может давать лишние документы в выдаче (то есть снижать точность), но для уровня “удовлетворительно” такой подход считается нормальным.

3 Производительность

Стемминг почти всегда работает медленнее чистой токенизации, потому что после выделения токена нужно ещё прогнать его через набор правил и сделать несколько проверок суффиксов. Поэтому время обработки увеличивается, а скорость (в KB/s) падает.

Ниже приведены итоговые показатели для корпуса, чтобы можно было сравнить токенизацию и стемминг на одинаковом объёме текста.

Tokenizer (Lab03)

- files: 38142

- input_kb: 424176
- total_tokens: 59284975
- avg_token_len: 5.36174
- time_s: 3.771
- speed_kb_s: 112487.3

Stemming (Lab04)

- files: 38142
- input_kb: 424176
- total_tokens: 59284975
- avg_token_len: 5.06492
- time_s: 6.415
- speed_kb_s: 66124.7

Из сравнения видно, что стемминг уменьшает среднюю длину токена (часть окончаний отрезается) и заметно увеличивает время обработки. При этом производительности всё равно хватает для обработки десятков тысяч документов и дальнейшей потоковой индексации, то есть практическое применение на корпусе такого размера остаётся удобным.

10 Лабораторная работа 5. Закон Ципфа

1 Построение распределения

Для проверки закона Ципфа я сначала строю частотный словарь по всему корпусу. То есть для каждого термина считаю, сколько раз он встретился в документах (после токенизации, а при необходимости и после стемминга).

Дальше действия простые:

1. Собираю пары `термин` → `частота`.
2. Сортирую термины по убыванию частоты.
3. Самому частому термину присваиваю ранг 1, следующему ранг 2 и так далее.

4. Строю график “частота от ранга” в логарифмической шкале (log-log), потому что обычный график получается слишком “скошенным” и по нему трудно сравнивать.

Такой график удобен тем, что если распределение близко к Ципфу, то в log-log масштабе получается почти прямая линия на заметном участке.

2 Сравнение с законом Ципфа

Закон Ципфа в простом виде говорит, что частота термина примерно обратно пропорциональна его рангу:

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

где r — ранг термина, $f(r)$ — частота, C — некоторая константа.

На практике идеального совпадения не бывает, и это нормально. В моём корпусе я заметил такие причины отклонений:

- **верх графика (самые частые слова)** портится тем, что корпус тематический: кроме обычных частотных слов появляются “доменные” слова, которые встречаются часто именно из-за темы (например, ship/port/marine и т.п.). Также влияет то, что Wikipedia и MarineLink по объёму разные.
- **хвост графика (редкие слова)** получается более “тяжёлым”, потому что в морской тематике много редких сущностей: названия судов, имена людей, географические точки, аббревиатуры, коды и т.д. Они часто встречаются один-два раза и сильно увеличивают хвост.
- **правила токенизации** тоже влияют: я отбрасываю токены длиной 1 символ, и из-за этого меняется распределение самых частых “мусорных” токенов. Также дефисы/апострофы могут разбивать слова на части, и это добавляет редких токенов.

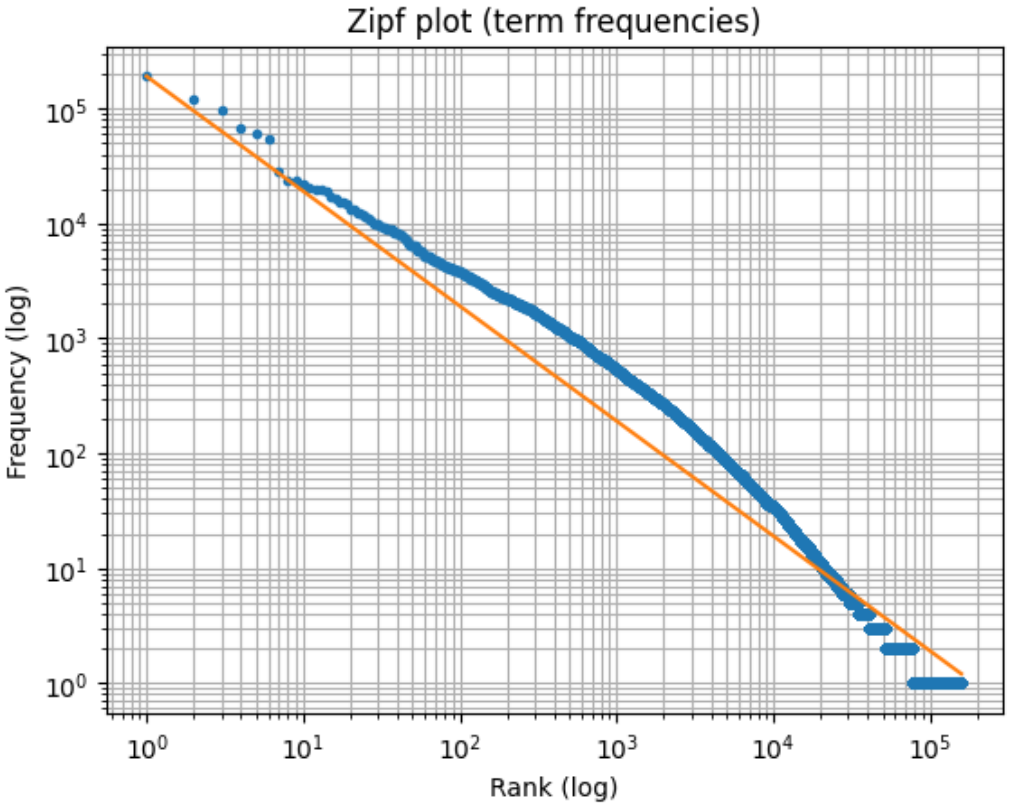

Рис. 1: Распределение терминов по частотам и закон Ципфа (log-log)

В целом график получается близок к прямой линии в log-log масштабе, то есть закон Ципфа для корпуса в общем работает. Отклонения в начале и в хвосте объясняются тематикой корпуса, смешиванием двух разных источников и особенностями токенизации.

11 Лабораторная работа 7. Булев индекс

1 Построение инвертированного индекса

Для булевого поиска нужен инвертированный индекс. Идея простая: вместо того чтобы каждый раз пробегать по всем документам, мы заранее строим структуру вида “термин → список документов”, где этот термин встречается.

Построение индекса я делаю так:

- по очереди читаю каждый текстовый файл документа;

- прогоняю текст через токенизацию и стемминг, чтобы дальше работать с едиными терминами;
- для каждого полученного термина добавляю текущий `docID` в список документов этого термина.

Важный момент: внутри одного документа одно и то же слово может встречаться много раз, но для булевого индекса это не важно. Поэтому я добавляю `docID` в postings list без повторов внутри документа. Это уменьшает размер индекса и ускоряет операции пересечения/объединения при поиске.

2 Хранение

Индекс сохраняется на диск, чтобы потом поиск работал быстро и не требовал пересборки.

Файлы индекса разделены на несколько частей:

- `docs.tsv` — таблица соответствия `docID` и документа. Я храню там источник и путь к текстовому файлу. Это нужно для выдачи результатов: поиск возвращает docID, а по нему можно понять, какой это документ.
- `dict.tsv` — словарь терминов. Для каждого термина хранится, где именно в бинарном файле начинаются его postings и какой длины этот список. Так поиск может быстро открыть нужный участок `postings.bin`.
- `postings.bin` — бинарный файл со списками docID. Списки лежат в отсортированном виде, потому что булевы операции (AND/OR) проще и быстрее делать именно на отсортированных списках.

Такое разбиение удобно тем, что словарь можно читать отдельно, а большие списки документов хранить компактно в бинарном виде. Это также упрощает масштабирование: даже если документов станет больше, поиск всё равно открывает только те postings, которые нужны под конкретный запрос.

3 Замечания по масштабированию

Построение индекса выполняется потоково: документы обрабатываются по одному, поэтому не нужно загружать весь корпус в память. Основная память уходит на словарь терминов и рабочие структуры при сборке. В целом подход подходит для десятков тысяч документов, а при аккуратной реализации и для большего объёма, если хватает места на диске под индекс.

12 Лабораторная работа 8. Булев поиск

1 Логика поиска

В этой лабораторной я реализовал булев поиск по уже построенному инвертированному индексу. Смысл булевого поиска в том, что пользователь задаёт логическое выражение, а система возвращает документы, которые этому выражению удовлетворяют.

Поддерживаются операторы:

- **AND** — пересечение множеств документов (оба слова должны быть в документе);
- **OR** — объединение (достаточно, чтобы было хотя бы одно слово);
- **NOT** — исключение (убираем документы, где встречается слово справа);
- **скобки** — позволяют явно задать порядок вычисления.

Чтобы запрос работал правильно, его нужно разобрать (распарсить). Я делаю это так:

- сначала строка запроса превращается в список токенов запроса (слова, AND/OR/NOT, скобки);
- дальше выражение приводится к удобному виду для вычисления (например, в ОПН/постфиксную форму);
- затем выражение вычисляется над postings lists.

Операции AND/OR выполняются на отсортированных списках `docID` с помощью простого двухуказательного алгоритма: это быстро и не требует лишней памяти. Для NOT берётся множество всех документов и из него вычитается список документов термина справа.

Примеры запросов:

- `offshore AND sea` — документы, где встречаются оба термина;
- `sponge OR star` — документы, где есть хотя бы один из терминов;
- `(sponge AND star) OR (offshore AND sea)` — комбинированный запрос со скобками;
- `ship AND NOT war` — пример запроса с исключением (NOT).

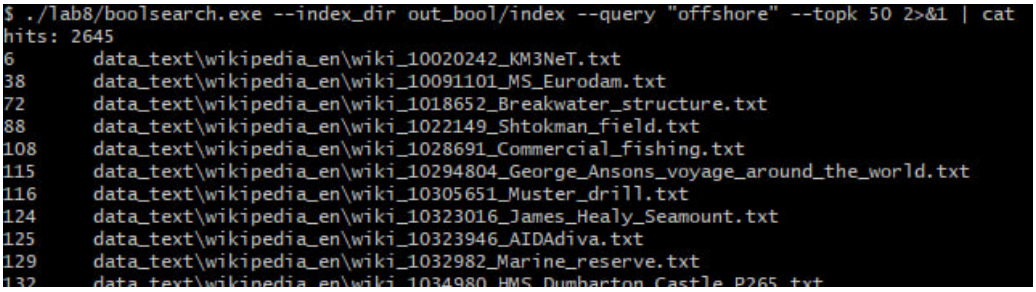

Рис. 2: Пример поиска

Важно, что такие запросы удобно использовать для проверки индекса: результат должен быть логически понятен (например, AND даёт меньше документов, чем OR).

2 Интерфейс

Чтобы было удобно и тестировать, и показывать работу, я сделал два интерфейса:

- **Утилита командной строки** `boolsearch` — принимает запрос и выводит количество найденных документов и список результатов. Этот вариант удобен для отладки и для автотестов, потому что вывод легко сравнивать.
- **Веб-интерфейс (Flask)** — простая HTML-страница с формой поиска. Вводится запрос, затем показывается выдача.

В веб-интерфейсе я дополнительно разделил результаты по источникам: Wikipedia показывается в одной колонке, MarineLink — в другой. Это удобно, потому что источники сильно разные по стилю, и можно быстро увидеть, откуда берутся найденные документы.

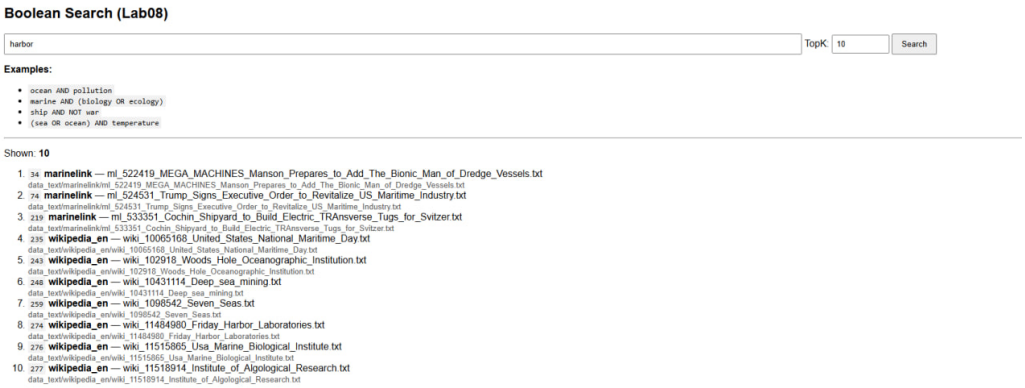

Рис. 3: Веб-интерфейс: пример запроса и выдачи (пример 1)

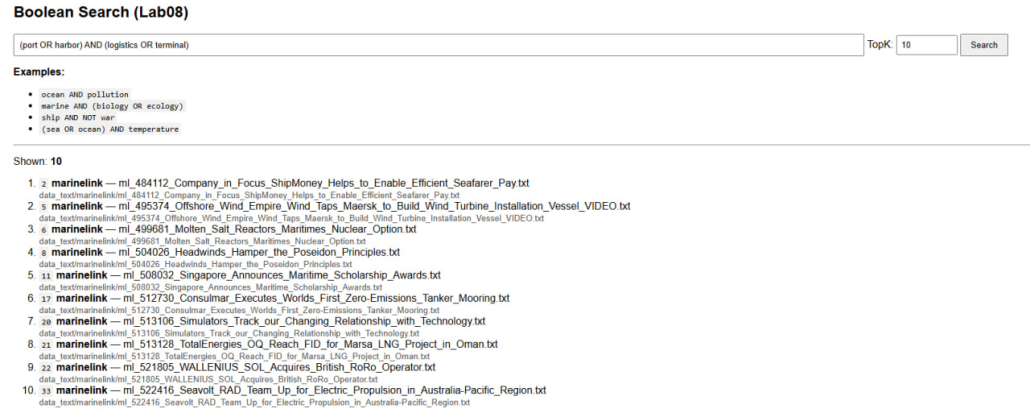

Рис. 4: Веб-интерфейс: пример запроса и выдачи (пример 2)

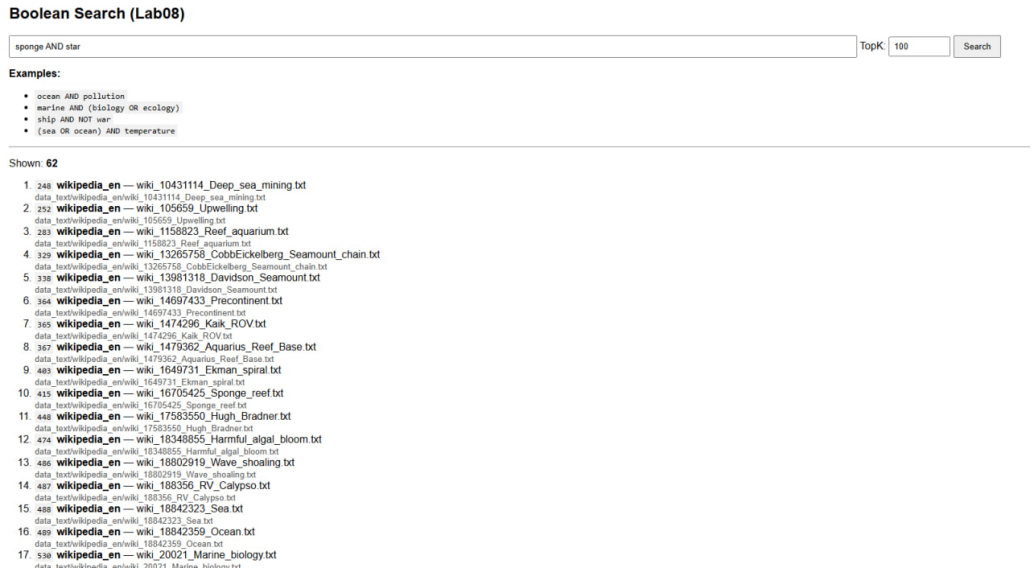

Рис. 5: Веб-интерфейс: пример запроса и выдачи со скобками (пример 3)

13 План тестирования

Ниже приведён план тестирования, который я использовал, чтобы убедиться, что каждая часть проекта работает правильно. Идея простая: после каждого шага должен быть понятный результат (файлы/вывод/записи в БД), который можно быстро проверить.

1 Lab01

- **Проверка структуры данных.** Проверить, что после скачивания появились папки для обоих источников и в них лежат файлы (сырые документы и отдельно тексты).
- **Проверка двух источников.** Убедиться, что в корпусе реально присутствуют документы из Wikipedia и из MarineLink (например, по названиям папок или по полю `source` в JSON).
- **Проверка выделенного текста.** Открыть несколько случайных `.txt` файлов и убедиться, что это действительно текст статьи/новости, а не мусор (меню, реклама, пустые строки).
- **Проверка минимальной длины.** Проверить, что тексты не пустые и проходят минимальный порог длины (если используется фильтр по символам).

2 Lab02

- **Проверка первого запуска.** Запустить робота на небольшом количестве URL и проверить, что в SQLite создалась база и таблицы, и что появились записи о документах.
- **Проверка корректности записи.** Проверить несколько записей в БД: что сохраняются url, source, время обкачки и html.
- **Остановка и продолжение.** Во время работы остановить робота (Ctrl+C), затем запустить снова и убедиться, что он продолжает работу, а не начинает заново.
- **Проверка повторной обкачки.** Запустить робота повторно и убедиться, что уже скачанные страницы не перекачиваются бесконечно, а обновляются только при необходимости (если документ изменился).
- **Проверка устойчивости.** Убедиться, что при сетевых ошибках робот не падает полностью, а пропускает проблемные URL или пытается повторить позже.

3 Lab03–Lab05

- **Проверка количества файлов.** Запустить токенизацию и убедиться, что программа обработала все файлы из `data_text` (количество обработанных файлов совпадает с количеством текстовых документов).

- **Проверка статистики.** Проверить, что в выводе есть осмысленные значения: общее число токенов больше нуля, средняя длина токена выглядит адекватно, время работы и скорость рассчитаны.
- **Сравнение токенизации и стемминга.** Запустить стемминг и убедиться, что средняя длина токена немного уменьшилась, а время стало больше (это ожидаемо).
- **Построение Ципфа.** Построить частотное распределение терминов и график в log-log масштабе. Проверить, что график выглядит как типичная "почти прямая линия" на среднем участке и не превращается в случайный шум.
- **Визуальная проверка.** На всякий случай открыть пару частотных слов вручную (например, топ-20) и убедиться, что там находятся действительно часто встречающиеся термины, а не мусор.

4 Lab07–Lab08

- **Проверка сборки индекса.** После построения индекса проверить наличие файлов `docs.tsv`, `dict.tsv`, `postings.bin` и что они не нулевого размера.
- **Проверка простых запросов.** Протестировать несколько запросов без скобок и убедиться, что поиск даёт ожидаемый результат (например, AND даёт меньше результатов, чем OR).
- **Проверка скобок и приоритетов.** Сравнить результаты для `a AND b OR c` и `a AND (b OR c)`, чтобы убедиться, что скобки реально влияют на порядок вычисления.
- **Проверка NOT.** Протестировать запрос вида `ship AND NOT war` и убедиться, что документы со словом `war` исключаются.
- **Пустая выдача.** Проверить запрос, который почти наверняка не встретится, и убедиться, что программа корректно сообщает 0 результатов и не падает.
- **CLI vs Web.** Для одинакового запроса сравнить результаты командной строки и веб-интерфейса (по количеству hits и по нескольким первым документам).

14 Выводы

В ходе выполнения лабораторных работ я собрал корпус на морскую тематику из двух источников (Wikipedia EN и MarineLink) и сделал простой поисковик по этому корпусу. Система умеет загружать документы, сохранять их, выделять из HTML

чистый текст, делать токенизацию и стемминг, строить булев инвертированный индекс и выполнять булев поиск с операторами AND/OR/NOT и со скобками. Также я сделал веб-страницу для поиска, чтобы было удобнее демонстрировать работу не только через консоль.

Главный плюс того, что получилось — проект можно запускать по шагам: сначала собрать корпус, потом выделить текст, потом построить индекс и уже после этого выполнять запросы. Это удобно для тестирования и для того, чтобы не переделывать всё сразу при мелких изменениях.

Минусы текущей версии:

- **Нет ранжирования.** Результаты поиска не сортируются по релевантности, поэтому в выдаче могут быть документы, которые формально подходят под запрос, но не являются лучшими ответами.
- **Дисбаланс источников.** Wikipedia значительно больше по количеству документов, из-за этого в выдаче она часто доминирует, и результаты из MarineLink могут появляться ниже (особенно если брать небольшой топ выдачи).
- **Простая токенизация.** Простые правила токенизации иногда дают неудачные токены (дефисы, апострофы, аббревиатуры), это может немного ухудшать поиск по некоторым запросам.
- **Стемминг не идеален.** Иногда разные слова могут сводиться к одному стему или наоборот, что может давать лишние совпадения.

Если улучшать систему дальше, то логичнее всего добавить ранжирование (например, TF-IDF или BM25), чтобы результаты были более полезными по порядку. Также можно улучшить обработку запросов: поддержать фразы, сделать более аккуратную токенизацию для дефисов и апострофов, а ещё добавить небольшие бонусы за точное совпадение слова без стемминга. В целом текущая версия закрывает требования для уровня “удовлетворительно” и показывает рабочий конвейер от скачивания документов до поиска по индексу.

Список литературы

[1] Manning C. D., Raghavan P., Schütze H. *Introduction to Information Retrieval.* — Cambridge University Press, 2008.

[2] Маннинг К., Рагхаван П., Шютце Х. *Введение в информационный поиск.* — М.: Издательский дом «Вильямс», 2011. — 528 с. (ISBN 978-5-8459-1623-4).

[3] Wikipedia contributors. *Wikipedia*. — Online encyclopedia. (дата обращения: 29 декабря 2025 г.).

[4] MarineLink. *MarineLink: Maritime News*. — Online. (дата обращения: 29 декабря 2025 г.).

[5] Pallets Projects. *Flask Documentation*. — Online. (дата обращения: 29 декабря 2025 г.).

[6] SQLite authors. *SQLite Documentation*. — Online. (дата обращения: 29 декабря 2025 г.).

[7] ГОСТ Р 7.0.5–2008. Библиографическая ссылка. Общие требования и правила составления. — (дата обращения: 29 декабря 2025 г.).