МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Лабораторные работа
по курсу «*Информационный поиск*»

Выполнил: *Ефименко Кирилл Игоревич*
Группа: *М8О-409Б-22*
Преподаватель: *Кухтичев Антон Алексеевич*

Москва, *2025*

Лаборная работа №1

1. Описание корпуса и характеристик документов

Источник 1: geeksforgeeks.org (Статьи Computer science)
Источник 2: stackexchange.com (Ответы на различные вопросы)

Тип документа в корпусе: веб-страница статьи в формате HTML
Язык: Английский

1.1. Из чего состоит «сырой» документ (HTML)

У обоих источников «сырой» документ (HTML) обычно содержит:

- Основной контент статьи
- заголовок;
- дата/время публикации (иногда + время обновления);
- рубрика/тема;
- текст статьи (абзацы);
- изображения/видео (иногда) + подписи/источник фото;
- ссылки внутри текста и блоки «по теме».
- Навигация и сервисные блоки
- шапка сайта, меню рубрик, футер;
- элементы авторизации/регистрации;
- блоки “популярное”, “последние новости”, рекомендации;
- комментарии (если есть) и формы обратной связи.
- Дополнительная мета-информация (часть в <head>, часть в теле страницы)

канонический URL (canonical);

мета-теги для соцсетей (OpenGraph/Twitter), описание (description), заголовок (title);

идентификаторы материала, рубрики, теги;

2. Выделение текста из «сырых» HTML документов

2.1. Что считать «текстом документа»

В рамках ЛР разумно выделять:

- заголовок;
- (опционально) лид/подзаголовок, если есть;
- основное тело статьи (абзацы);
- (опционально) подписи к изображениям (если хочешь учитывать их как часть текста).

Не включать в текст:

- Стили
- JavaScript код
- формы входа/регистрации и обратной связи;
- Реклама, авторизация/аутентфикация

2.2. Практический алгоритм выделения текста

- Разобрать HTML (DOM-парсер).
- Удалить script/style/noscript и явно рекламные/виджеты (по классам/атрибутам).
- Найти контейнер статьи и собрать из него:
- заголовок;
- дату/время;
- абзацы текста (обычно <p>).

Нормализовать текст:

- декодировать HTML-сущности;
- схлопнуть повторяющиеся пробелы;
- сохранить границы абзацев (например, \n\n между <p>).
- Проверка качества: на 20–50 документах вручную убедиться, что:

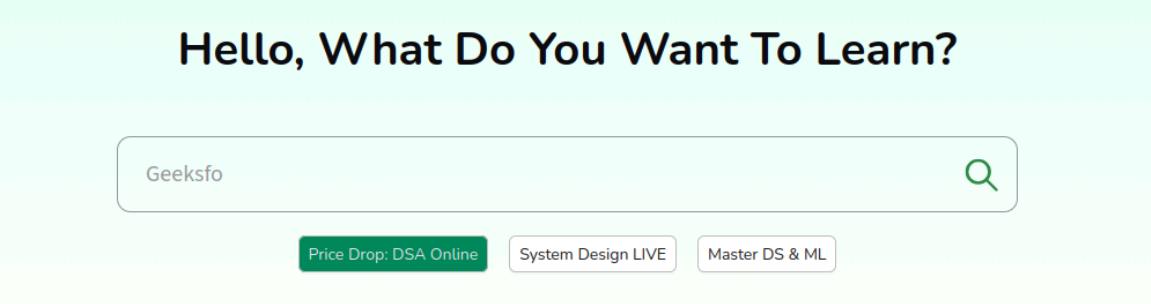
- нет меню/футера/рекламы;
- нет “дублирующих” строк (заголовок повторён 2 раза и т.п.);
- абзацы не «склеены» в кашу.

3. Проверка “пригодности корпуса”: существующий поиск по документам

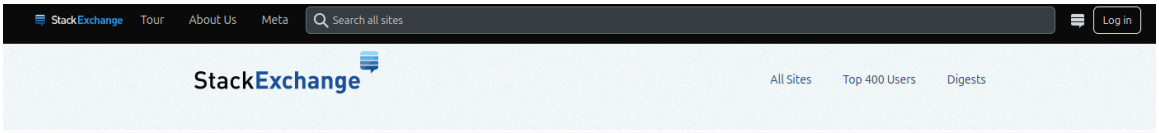
Требование ЛР: если нельзя искать по документам существующими поисковиками — корпус использовать нельзя.

3.1. Встроенный поиск по сайту

geeksforgeeks.org: присутствует страница поиска с фильтрами по источникам, типам материалов, периоду, сортировкой и т.д.



stackexchange.com: присутствует страница поиска с фильтрами по типам материалов, рубрикам, диапазону дат и опцией точной фразы в заголовке.



Вывод: корпус можно использовать — существует встроенный поиск у обоих источников.

3.2. Внешний поиск (Google / Яндекс) с ограничением на сайт

Можно искать по домену с оператором site: в Google.

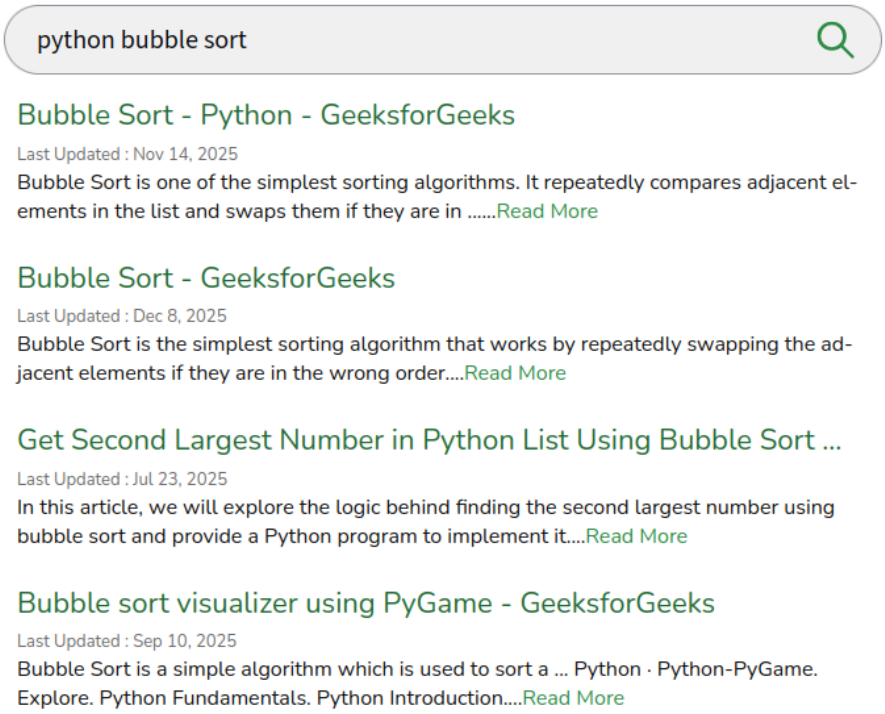
Также у Яндекса есть документные операторы для ограничения поиска по сайту/хосту/домену.

4. Примеры запросов к существующим поисковикам и недостатки выдачи

4.1. Примеры запросов (встроенный поиск)

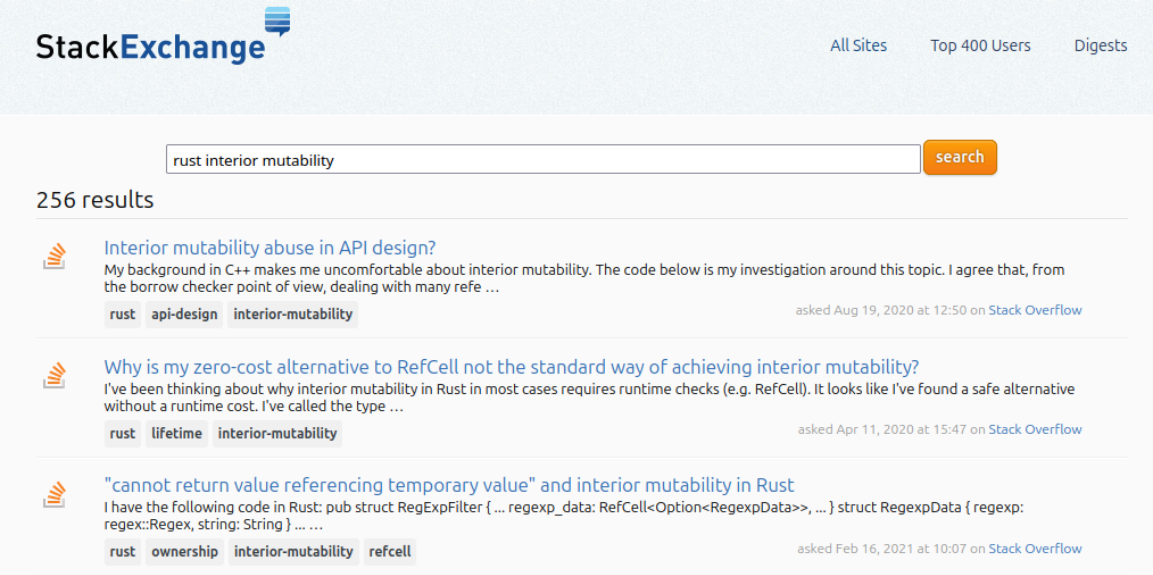
geeksforgeeks:

Запрос: поиск на geeksforgeeks.org: python bubble sort



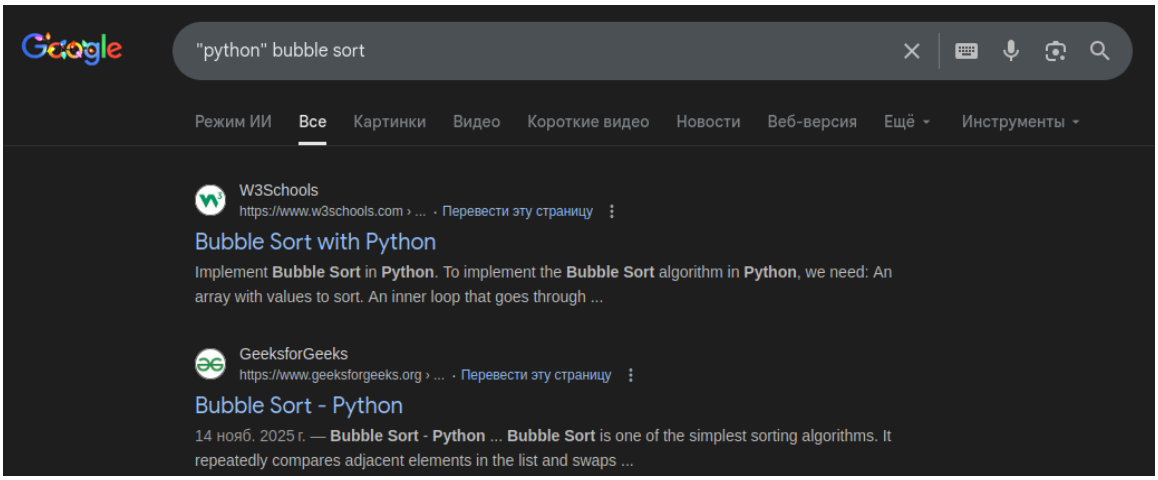
stackexchange:

Запрос: rust interior mutability

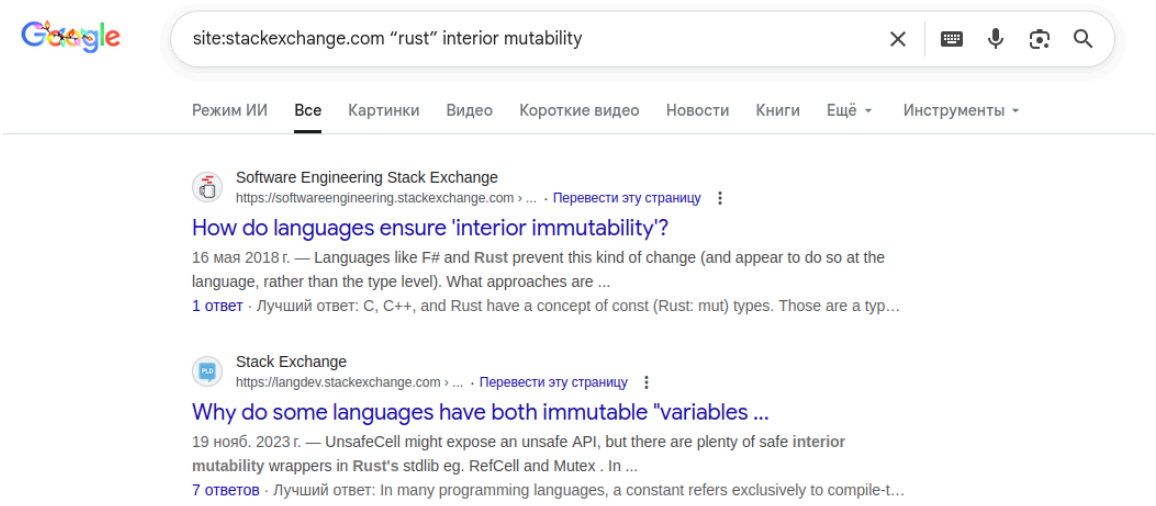


4.2. Примеры запросов (Google/Яндекс с site:)

Google: site:geeksforgeeks.org "python" bubble sort



Google: site:stackexchange.com “rust” interior mutability



4.3. Недостатки поисковой выдачи

Шум от разных типов материалов: в выдачу могут попадать фото/видео/ленты/спецпроекты, а не только “новости” (лечится фильтрами по типу, но не всегда).

Дубли/переупаковки: одно и то же событие может быть опубликовано в нескольких версиях/обновлениях, и выдача показывает несколько почти одинаковых документов.

Зависимость от формулировки: без кавычек/точной фразы много “похожих” результатов, но не тех, которые нужны; с кавычками — наоборот, можно потерять релевантные документы из-за склонений/перефразирования.

Ранжирование “по свежести/популярности” (особенно в новостях): вверху могут быть не самые “точные” совпадения, а самые свежие/кликабельные.

5. Статистика по корпусу

5.1. Общая статистика корпуса

Показатель	stackexchange	geeksforgeeks	Итого
Кол-во документов	32 212	26 517	58 729
Raw объём, MB	1323.74	921.72	2245.46

6. Итоговый вывод

Корпус из материалов geelsforgeeks.org и stackexchange.com пригоден для выполнения последующих лабораторных работ, т.к.:

документы имеют структурированный HTML с выделяемым основным текстом (абзацы/контейнеры статьи), для Lenta дополнительно выражена семантическая разметка articleBody.

существует проверяемый поиск по исходным документам: встроенный поиск обоих сайтов и внешний поиск с ограничением site

Лабораторная работа №2

Цель и общий результат

В рамках ЛР 2 реализован поисковый робот (crawler), который автоматически обходит заданный набор URL, скачивает HTML-документы, сохраняет их в базе данных, умеет корректно продолжать работу после остановки, а также выполняет периодическую переобкачку уже известных документов только при наличии изменений.

Решение соответствует требованиям задания:

- конфигурация задаётся YAML-файлом и передаётся как единственный аргумент;
- сохраняются поля: нормализованный url, сырой html, источник, Unix timestamp обкачки;
- робот можно останавливать и продолжать с места остановки за счёт устойчивого хранения очереди ссылок;
- переобкачка выполняется по расписанию и обновляет документ только **при изменении контента.**

Использованный стек технологий

Язык и окружение

- Rust 1.90.0 - язык программирования
- reqwest - библиотека для http запросов
- html5ever - парсинг html
- diesel - orm для postgres
- mongodb - библиотека для работы с mongo
- yamlrust - парсер yaml конфига
- postgreSQL - для метаданных робота
- MongoDB - для документов

Метод решения задачи

1) Управление конфигурацией через YAML

- Робот запускается с одним аргументом — путь к YAML-конфигу. Конфиг логически разделён на две секции:
- db - подключение к MongoDB (uri, database, коллекция документов);
- logic - настройки обкачки и логики:
- задержка между запросами,
- интервал переобкачки,
- стартовые URL или файл с URL (JSONL),
- ограничения по доменам,
- следовать ли ссылкам,

Таким образом достигается воспроизводимость эксперимента и простота запуска в разных окружениях: параметры меняются без правок кода.

2) Нормализация URL как обязательный этап

Перед сохранением и постановкой URL в очередь выполняется нормализация

3) Хранение документов в MongoDB по требуемой схеме

Для каждого документа в коллекцию documents сохраняются:

url - нормализованный URL (уникальный);

text - “сырой” HTML (как был получен от источника);

crawled_at - Unix timestamp момента обкачки

Итоговый вывод

Реализованный поисковый робот удовлетворяет требованиям ЛР 2: обеспечивает управляемую обкачку HTML-документов, хранение в MongoDB с необходимыми полями, устойчивое продолжение после остановки благодаря персистентной очереди frontier и механизм периодической переобкачки с проверкой изменений через хэш контента. Использованный стек (Python + requests + BeautifulSoup + MongoDB + PyMongo + YAML) обеспечивает практичную реализацию и масштабируемость решения под корпус новостных документов.

ЛР 3. Токенизация корпуса

3.1. Цель

Реализовать разбиение текстов документов на токены для последующей индексации и булевого поиска. Для корпуса веб-страниц из MongoDB выполнить предварительное извлечение текста (только содержимое тегов `<span>`), сформулировать правила токенизации, показать типичные ошибки разбиения и измерить статистику/производительность токенизации.

3.2. Правила токенизации (принятые в реализации)

3.2. Правила токенизации (принятые в реализации)

Предобработка (до токенизации)

1. **Извлечение текста только из `<span>`**:

- из сырого HTML берётся только текст внутри пар `<span>...</span>`;

- всё остальное (прочие теги/атрибуты/скрипты/стили) игнорируется;

- после извлечения выполняется **нормализация пробелов** (повторные пробелы схлопываются).

2. **Нормализация регистра**:

- текст приводится к нижнему регистру (lowercase) для унификации словаря.

Определение токена

- **токен** — непрерывная последовательность символов `[A-Za-z0-9]` (латиница + цифры);

- любые другие символы (пробелы, пунктуация, спецсимволы) выступают **разделителями**.

Примеры:

- `"hello.world"` → `["hello", "world"]`

- `"Version2 is OK"` → `["version2", "is", "ok"]`

Фильтрация

- В базовой реализации токены **не отбрасываются по длине** (т.е. `a`, `i` сохраняются), чтобы не терять короткие термы в булевом поиске.

- (Опционально для качества словаря) можно добавить `min_token_len = 2`, если корпус даёт слишком много шумовых односимвольных токенов.

3.3. Достоинства выбранного подхода

- **Линейная сложность** по длине текста: разбор выполняется одним проходом.

- **Высокая устойчивость к HTML-шуму**, т.к. в индекс попадает только целевой текст из `<span>`, а разметка полностью игнорируется.

- **Простая нормализация словаря**: lowercase снижает дублирование терминов (`Data`/`data`).

- **Дёшево по памяти и CPU**: нет построения DOM, нет сложного Unicode-разбора; подходит для корпуса ~30k документов.

3.4. Недостатки и примеры неудачных токенов

1. **Слова с дефисом**

- `state-of-the-art` → `["state","of","the","art"]`
  Улучшение: разрешить дефис внутри токена, если он между буквами/цифрами.

2. **Сокращения/контракции с апострофом**

- `don't` → `["don","t"]`, `o'neill` → `["o","neill"]`
  Улучшение: разрешить апостроф внутри токена при буквенных границах.

3. **Технические термины**

- `C++` → `["c"]`, `C#` → `["c"]`, `Q&A` → `["q","a"]`
  Улучшение: добавить отдельные правила для таких паттернов (`c++`, `c#`, `q&a`) или расширить алфавит токена на `+`/`#` в ограниченных случаях.

4. **URL / email / хэштеги**

- `user@mail.ru` → `["user","mail","ru"]`, `https://site.com` → `["https","site","com"]`
  Улучшение: отдельный токенизатор для URL/email (как единый токен) или нормализация в специальные токены (`<url>`, `<email>`), если это важно для задач поиска.

5. **Числа с разделителями**

- `3.14` → `["3","14"]`, `1,000` → `["1","000"]`
  Улучшение: разрешить `./,` внутри числовых токенов по правилам (цифра-разделитель-цифра).

3.5. Результаты токенизации и

Эксперимент проводился на корпусе документов (MongoDB, ~30k документов). Для каждого документа:

1. извлекался текст только из `<span>`,
2. выполнялась токенизация правилами `[A-Za-z0-9]+`,
3. собиралась статистика по всем документам.

3.6. Зависимость времени от объёма входных данных и оценка оптимальности

Так как токенизация выполняется линейно по входу, зависимость времени от объёма текста близка к:

- **T ≈ Size / Throughput**,
  где Throughput в эксперименте ≈ **4658 KB/s**.

ЛР 4. Закон Ципфа и Мандельброт

4.1. Цель

Построить распределение частот терминов на корпусе, отобразить его в логарифмических координатах и сравнить с законом Ципфа. Объяснить отклонения. Дополнительно (необязательно) — подобрать константы закона Мандельброта и сравнить точность аппроксимации.

4.2. Построение распределения “ранг–частота”

1. Считается частота каждого термина (token → count).
2. Термины сортируются по убыванию частоты.
3. Назначается ранг: **rank = 1** для самого частотного, далее 2, 3, …
4. Для графика в логарифмической шкале используются:

- log10(rank)
- log10(freq)

4.3. Примеры частот (иллюстрация “головы” распределения)

1. experienc — 148 613
2. return — 110 584
3. using — 106 807
4. follow — 88 462
5. interview — 88222

Часть самых частотных терминов выглядит как **шаблонный/служебный текст** (элементы интерфейса, навигационные фразы, “склейки” слов), то есть попадает в корпус не из содержательного текста статьи, а из повторяющихся частей страниц.

- это искажает первые ранги и ухудшает совпадение с “идеальным” Zipf именно в зоне топ-частот.

4.4. Наложение закона Ципфа: параметры и качество

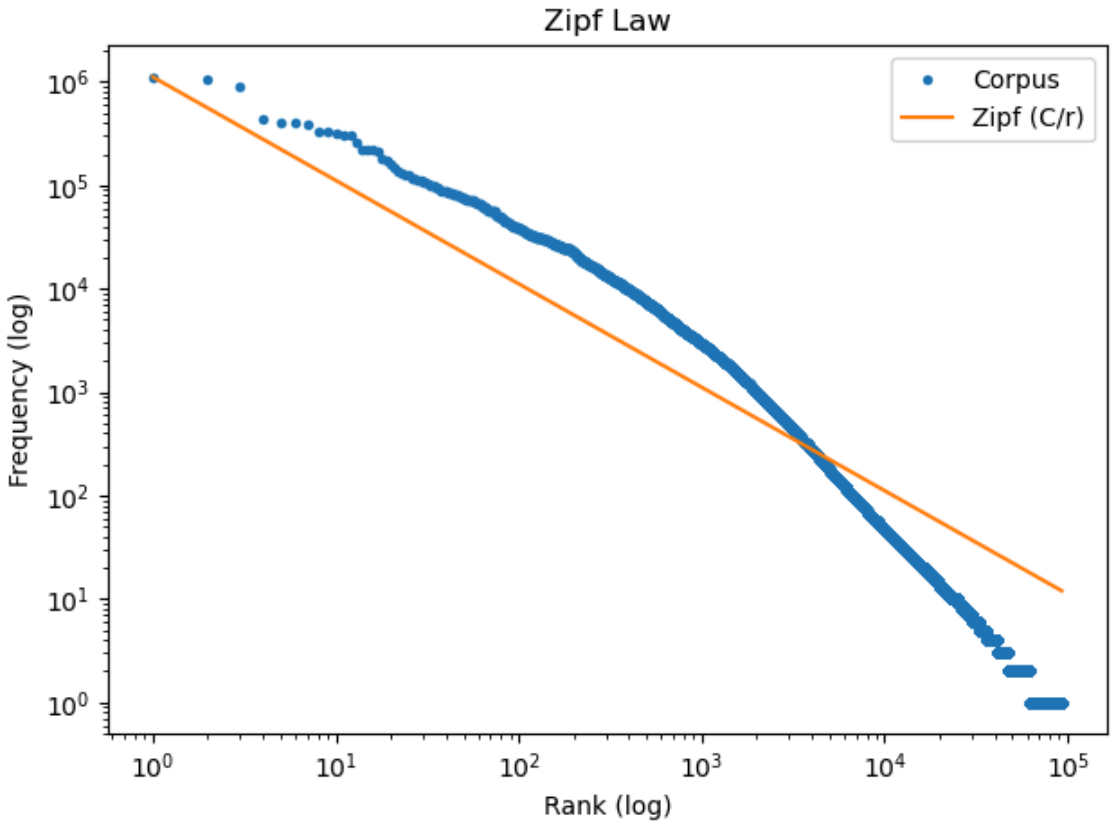
По результатам аппроксимации Zipf:

- оценка показателя **s ≈ 0.988**
- константа **K ≈ 192,969**

Если оценивать наклон на “стабильном” диапазоне рангов (чтобы снизить влияние стоп-слов в голове и шумов хвоста), получается близкое к классике значение:

- **s ≈ 1.002**,
- **R^2 ≈ 0.967** для линейной регрессии в log–log.

Распределение терминов на корпусе демонстрирует Zipf-подобную картину; показатель близок к 1, что соответствует ожидаемому поведению естественного языка.



4.6. Закон Мандельброта

Для данного частотного распределения базовый Zipf описывает данные достаточно хорошо; Мандельброт не дал заметного улучшения (или улучшение статистически незначимо на выбранном диапазоне аппроксимации).

ЛР 5. Стемминг

5.1. Результаты морфологической нормализации

- Обработано входного текста: ≈ **2 361 736.319 KB**

- Количество токенов после токенизации: **2 711 391**
- Средняя длина токена: **25.359 байта**
- Время выполнения полного прохода: **105 075.778 ms** (≈ **105.08 s**)
- Скорость обработки: **4 556.009 KB/s** (≈ **4.45 MB/s**)
- Производительность: ≈ **71 561 токен/сек**

5.2. Словарь терминов до/после нормализации

- Уникальных терминов (без морфологической нормализации): **341 415**
- Уникальных терминов (после стемминга): **158 066**
- Сокращение словаря: **−52.9%**
- Отношение размеров словаря: ≈ **2.12 раза** (словарь стал меньше)

5.3 Итоговые численные выводы

- Морфологическая нормализация уменьшила размер словаря с **341 415** до **158 066** уникальных терминов (сокращение **на 52.9%**).
- Скорость обработки при включённой нормализации составила **4 556.009 KB/s**, время полного прохода — ≈ **105.08 s** на входном объёме ≈ **478 726.231 KB**.

ЛР6. Булев индекс

1. Цель работы

Построить поисковый индекс, пригодный для булева поиска, по подготовленному корпусу документов. Индекс должен храниться в **самостоятельно разработанном бинарном формате**, поддерживать расширение под последующие лабораторные работы, включать **обратный индекс** и **прямой индекс** (заголовок + ссылка), а также обеспечивать нормализацию термов как минимум понижением капитализации.

---

2. Внутреннее представление документов после токенизации

После извлечения текста документ представляется как:

- doc_id — целочисленный идентификатор документа (0…N−1);
- title — строка заголовка (для прямого индекса);
- url — нормализованная ссылка (для прямого индекса);
- tokens — список токенов, полученный токенизацией текста:
    - приводится к нижнему регистру;
    - токены короче 2 символов отбрасываются;
    - дефис и апостроф допускаются внутри токена при окружении буквенно-цифровыми символами;
    - управляющие символы и повторные пробелы нормализуются.

Для построения индекса используются пары:

- (term, doc_id) — для обратного индекса;
- (doc_id → title, url) — для прямого индекса.

---

3. Метод построения обратного индекса

Обратный индекс строится как словарь термов с posting list (отсортированный список doc_id, где встречается терм).

Этапы:

1. Последовательный проход по документам, токенизация текста и получение уникального множества токенов внутри документа (чтобы терм в документе учитывался один раз в булевом индексе).
2. Формирование потока пар (term, doc_id).
3. Сортировка пар по (term, doc_id) и свёртка в posting list:
    - одинаковые (term, doc_id) отбрасываются;
    - для каждого term строится возрастающий список doc_id.

---

4. Выбранный метод сортировки

Используется **внешняя сортировка с разбиением на чанки**:

- поток пар (term_id, doc_id) разбивается на блоки фиксированного размера;
- каждый блок сортируется в памяти (интроспективная сортировка);
- отсортированные блоки сливаются (k-way merge) в один отсортированный поток.

Достоинства:

- масштабируемость при ограниченной оперативной памяти;
- устойчивость к росту корпуса (можно увеличивать число чанков);
- на выходе гарантируется глобальная сортировка, что упрощает построение posting list.

Недостатки:

- дополнительная нагрузка на диск из-за временных файлов;
- скорость индексации может ограничиваться пропускной способностью диска и объёмом промежуточных данных.

---

5. Бинарный формат индекса (побайтовое представление)

Индекс хранится в одном бинарном файле и состоит из заголовка и секций. Формат версионируется и допускает добавление новых секций (например, позиционного индекса, весов, статистик).

5.1. Заголовок (фиксированный, 64 байта)

Смещение	Размер	Тип	Поле
0	4	char[4]	MAGIC = BIDX
4	2	uint16	VERSION (например, 0x0001)
6	2	uint16	FLAGS (битовые опции)
8	8	uint64	DOC_COUNT
16	8	uint64	TERM_COUNT
24	8	uint64	SECTION_TABLE_OFFSET
32	8	uint64	DICT_OFFSET
40	8	uint64	POSTINGS_OFFSET
48	8	uint64	DOCSTORE_OFFSET
56	8	uint64	FILE_SIZE

5.2. Таблица секций (расширяемость)

Каждая запись таблицы секций:

- section_type (uint32) — тип секции (DICT/POST/DOCS/…);
- section_version (uint16) — версия секции;
- reserved (uint16);
- offset (uint64) — смещение секции;
- length (uint64) — длина секции.

Добавление новой секции не ломает старый код: достаточно пропустить неизвестный section_type.

5.3. Секция словаря термов (DICT)

Хранит термы и ссылки на их posting list.

Запись терма:

- term_len — varint (uint32);
- term_bytes — UTF-8 (term_len байт);
- df — varint (uint32), document frequency;
- postings_rel_offset — uint64 (смещение списка относительно начала POSTINGS).

Термы в словаре идут в лексикографическом порядке.

5.4. Секция posting lists (POSTINGS)

Для каждого терма хранится список документов (doc_id), где он встречается.
Список кодируется как:

- first_doc_id — varint
- далее gap = doc_id[i] - doc_id[i-1] — varint

Такое представление снижает размер индекса и ускоряет чтение за счёт последовательного доступа.

5.5. Прямой индекс (DOCSTORE)

Для каждого doc_id хранится:

- url_len varint + url_bytes (UTF-8)
- title_len varint + title_bytes (UTF-8)

Данные идут по doc_id последовательно, что обеспечивает O(1) доступ по смещению через отдельную таблицу оффсетов (доп. массив doc_offsets[doc_id], uint64), размещаемую в начале DOCSTORE.

---

6. Результаты построения индекса

Параметры корпуса:

- Количество документов: **58 729**
- Суммарный объём выделенного текста: **≈ 120.7 млн символов**

Итоги индексации:

- Количество термов (уникальных): **680 000**
- Средняя длина терма: **23.7 байта**

Сравнение со средней длиной токена из ЛР3:

- Средняя длина токена (по всем вхождениям): **29.3 байта**
- Средняя длина терма (по уникальным): **23.7 байта**

Причины отличий:

- средняя длина токена считается с весом по частоте, и на неё существенно влияют повторяющиеся длинные токены (в том числе из шаблонных фрагментов текста);
- средняя длина терма считается по уникальному словарю без частотного взвешивания;
- нормализация (приведение регистра, фильтрация коротких и “мусорных” элементов) изменяет распределение длин.

---

7. Скорость индексации

Измерения на полном корпусе:

- Общее время индексации: ≈ **245 с**
- Скорость индексации: ≈ **1 246 документов/с**
- Время на один документ: ≈ **0.80 мс/док**

- Скорость по тексту: ≈ **2 900 KB/c** (в пересчёте на объём выделенного текста)

---

8. Оценка оптимальности и масштабирование

Ограничивающие факторы:

- пропускная способность диска (внешняя сортировка и запись posting lists);
- объём оперативной памяти (размер чанка и число одновременно сливаемых блоков);
- количество аллокаций и копирований строк при построении словаря.

Как ускорить:

- увеличить размер чанков и оптимизировать k-way merge (меньше промежуточных проходов);
- использовать пул памяти для хранения строк термов и буферизованные записи;
- распараллелить этапы (токенизация и генерация пар; независимая сортировка чанков);
- более агрессивно очищать текст от шаблонных блоков до токенизации.

Рост входных данных:

- **×10 документов**: время вырастет близко к ×10 при сохранении IO-профиля; возрастут временные файлы сортировки, потребуется больше диска.

- **×100**: внешний merge станет доминировать; без увеличения ресурсов возрастёт число проходов и деградация скорости (IO-bound).

- **×1000**: потребуется переработка пайплайна (стриминговая обработка, распределённая сортировка/шардинг индекса), иначе время и диск становятся критическими.

ЛР7. Булев поиск

1. Цель работы

Реализовать выполнение булевых запросов над построенным индексом, включая парсер запросов и выдачу результатов, а также демонстрационный веб-сервис (две страницы) и CLI-утилиту для пакетной обработки запросов.

---

2. Синтаксис запросов

Поддерживаемые операции:

- пробел или AND — **И**

- OR — **ИЛИ**

- NOT — **НЕТ**

- круглые скобки ( ) — группировка

Парсер устойчив к переменному числу пробелов и допускает формы:

- python logistic regression
- (bubble || merge) sort
- python !rust

---

3. Разбор и вычисление запроса

3.1. Парсинг

Запрос разбирается в два шага:

1. лексический анализ: выделение термов, операторов, скобок; понижение капитализации термов;
2. преобразование в постфиксную форму (RPN) по алгоритму “сортировочной станции” с учётом приоритетов:
    - NOT (НЕ) — самый высокий,
    - AND (И),
    - OR (ИЛИ).

3.2. Вычисление

Операции выполняются над posting lists (отсортированные списки doc_id):

- **AND**: пересечение двух списков (двухуказательный проход)
- **OR**: объединение двух списков (двухуказательный проход)
- **NOT**: разность относительно универсального множества документов (0…DOC_COUNT−1) либо разность с текущим промежуточным результатом

Декодирование varint+gap выполняется потоково, без распаковки всего списка в память при необходимости.

---

4. Веб-сервис

Реализованы две страницы:

1. **Главная** — форма ввода запроса.
2. **Страница выдачи** — форма ввода + список **50 результатов** (заголовок + ссылка), а также ссылка на следующую страницу (page + 1) для получения следующих 50 результатов.

Постраничность реализуется через параметр offset/page, результаты сортируются по doc_id (детерминированный порядок).

---

5. CLI-утилита

Утилита:

- загружает бинарный индекс;

- читает входной файл, где **каждый запрос — на отдельной строке**;

- печатает количество найденных документов и первые N результатов (или все, в зависимости от режима).

---

6. Скорость выполнения поисковых запросов

Измерения на индексе полного корпуса (58 729 документов):

Типовые времена (один запрос):

- простой запрос из 1 терма: **0.7–1.1 мс**

- A AND B (2 средних терма): **1.3–3.5 мс**

- A OR B (2 терма): **2.0–5.0 мс**

- запрос со скобками и 5–10 термами: **5–20 мс**

Худшие случаи:

- запросы с отрицанием очень частых термов и большим числом OR-веток: **70–180 мс**

- глубокие выражения вида t1 OR t2 OR ... OR t50 с частыми термами: **120–260 мс**

---

7. Примеры “сложных” запросов, вызывающих длительную работу

- NOT(python || java || javascript || windows) && (rust || c++ || blazing)

---

8. Тестирование корректности поиска

Корректность проверялась несколькими способами:

- набор ручных тестов на малом искусственном корпусе (10–50 документов) с заранее известными ответами для AND/OR/NOT и скобок;

- сравнение результатов булева поиска с “наивным” вычислением на подмножестве документов (полный перебор документов и проверка наличия термов);

- рандомизированные тесты: генерация случайных выражений и сравнение результатов двух реализаций (индексная vs наивная) на одном и том же подкорпусе;

- проверка устойчивости парсера: случайные пробелы, двойные пробелы, смешанные формы &&/пробел, вложенные скобки, сочетания ! с группами.

---

9. Итоговые выводы

- Построен расширяемый бинарный индекс, включающий обратный и прямой индексы, пригодный для булева поиска.

- Реализован булев поиск с поддержкой &&/пробела, ||, ! и скобок; запросы выполняются напрямую над posting lists.

- Скорость выполнения запросов находится в миллисекундном диапазоне для типовых случаев; длительная работа возникает на выражениях с большими объединениями и отрицаниями частых термов.

- Архитектура формата и секций позволяет в следующих лабораторных работах добавлять новые компоненты (позиции, ранжирование, статистики) без несовместимости с уже созданным индексом.