Министерство образования и науки РФ
федеральное государственное автономное образовательное учреждение высшего образования
«Омский государственный технический университет»

Факультет (институт) *Информационных технологий и компьютерных систем*

Кафедра *Информатики и вычислительной техники*

# КУРСОВОЙ ПРОЕКТ

по дисциплине ***Проектная деятельность***

на тему Принципы построения поисковых механизмов в серверных web-приложениях

**Пояснительная записка**

Шифр проекта **020-КП-09.03.01-ИВТ-129-24-ПЗ**

Студента (ки) **Шмидта Антона Владиславовича**
фамилия, имя, отчество полностью

Курс ***2*** Группа **ИВТ**-244

Направление (специальность) *09.03.01 – Информатика и вычислительная техника*
код, наименование

Руководитель *ассистент*
ученая степень, звание

*Симоненок А.Е.*
фамилия, инициалы

Выполнил (а) _____
дата, подпись студента (ки)

К защите _____
дата, подпись руководителя

| Выполнение и подготовка к защите КП (КР) | Защита КП (КР) | Итоговый рейтинг |
|---|---|---|
| | | |

Проект (работа) защищен (а) с оценкой _____

Омск 2026

**Оглавление**

Введение.....3

Основные компоненты и этапы работы поисковых механизмов.....4

Модели ранжирования и оценка релевантности.....6

Эволюция подходов к ранжированию: от простейших метрик к сложным системам .....6

Okapi BM25 — текущий золотой стандарт ранжирования.....7

Практическое значение параметров $k_1$ и b.....8

Дополнительные механизмы повышения качества ранжирования.....8

Современные многоступенчатые архитектуры ранжирования.....9

Масштабируемость и отказоустойчивость: шардирование и репликация.....10

Шардирование: принцип разделения данных.....10

Репликация: обеспечение надёжности и масштабирования чтения.....11

Типичные стратегии репликации в продакшене:.....12

Реализация в Elasticsearch и OpenSearch.....12

Интеграция поискового механизма с серверным web-приложением.....14

Основные способы взаимодействия приложения с поисковым движком.....14

Паттерны индексации данных.....14

Near Real-Time возможности современных движков.....16

Интеграция поиска на стороне фронтенда через API.....17

Дополнительные возможности современных поисковых систем.....18

Заключение.....20

Список литературы.....21

## Введение

В эпоху цифровой трансформации серверные web-приложения обрабатывают колоссальные объёмы информации: от каталогов интернет-магазинов и корпоративных баз знаний до пользовательского контента в социальных сетях и системах электронного документооборота. Поисковые механизмы в таких системах превратились из вспомогательной функции в один из определяющих факторов успеха продукта. Пользователь ожидает получить точные, релевантные и быстрые результаты за доли секунды, независимо от того, ищет ли он товар среди миллионов позиций или документ в архиве предприятия.

Серверная реализация поиска принципиально отличается от клиентской: она работает с большими данными, использует мощные вычислительные ресурсы серверов, обеспечивает безопасность и конфиденциальность, а также позволяет применять сложные алгоритмы индексации и ранжирования. Основные принципы построения таких механизмов сформировались на стыке информационного поиска, систем управления базами данных и распределённых вычислений.

Ключевыми технологиями стали библиотека Apache Lucene (основа большинства современных решений), а также построенные на ней Elasticsearch и Apache Solr. Эти инструменты обеспечивают полнотекстовый поиск, масштабируемость через шардирование и репликацию, гибкое ранжирование и множество дополнительных возможностей. В настоящей работе рассматриваются фундаментальные принципы организации поиска на серверной стороне, этапы обработки данных, модели ранжирования, вопросы масштабируемости, безопасности и интеграции с веб-приложениями.

## Основные компоненты и этапы работы поисковых механизмов

Любая современная поисковая система в серверном web-приложении проходит через несколько последовательных этапов: сбор данных (crawling / ingestion), анализ и нормализация текста, построение индекса, обработка запроса, ранжирование и выдача результатов.

Первый этап — сбор и поступление данных. В отличие от классических поисковиков интернета, где работает краулер, в большинстве серверных приложений данные поступают либо напрямую из базы данных (при добавлении/обновлении записей), либо через очередь сообщений (Kafka, RabbitMQ), либо в режиме реального времени (change data capture). Например, в интернет-магазине при добавлении нового товара бэкенд-логика отправляет JSON-документ в поисковый движок практически мгновенно.

Следующий критически важный этап — анализ текста и токенизация. Текст разбивается на токены (слова или их значимые части), приводится к нижнему регистру, удаляются стоп-слова («и», «в», «на» и т.п.), применяется стемминг или лемматизация. Для русского языка особенно важна качественная лемматизация, поскольку слова изменяются по падежам, числам и родам гораздо активнее, чем в английском. Современные анализаторы (например, в Elasticsearch russian analyzer) используют словари и правила морфологии, чтобы привести «книгами», «книге», «книга» к одной лемме «книга».

Центральным элементом всей архитектуры является инвертированный индекс — структура данных, в корне отличающаяся от традиционных реляционных таблиц. Вместо хранения документов и последовательного поиска по ним, инвертированный индекс хранит для каждого уникального термина (слова после нормализации) список документов, в которых этот термин встречается, а также позиции вхождений и дополнительную статистику (частота, длина документа и др.). Именно эта структура позволяет выполнять поиск за миллисекунды даже по миллиардам документов.

На практике индекс состоит из неизменяемых сегментов. Каждый новый пакет документов создаёт новый сегмент, который периодически сливается (merge) с

существующими для оптимизации. Apache Lucene (и построенные на ней системы) хранят сегменты как набор файлов в файловой системе: словарь терминов (terms dictionary), постинг-листы (posting lists), нормализованные длины документов и т.д.

После индексации начинается этап обработки пользовательского запроса. Запрос проходит те же этапы анализа, что и документы: токенизируется, нормализуется, иногда расширяется синонимами или исправлением опечаток (fuzzy search). Затем движок находит все документы, содержащие хотя бы один из терминов запроса (или комбинацию по логике AND/OR/NOT), и передаёт их в этап ранжирования.

## Модели ранжирования и оценка релевантности

Просто найти документы, которые содержат хотя бы некоторые слова из запроса пользователя, совершенно недостаточно для создания действительно полезной и качественной поисковой системы. Пользователь ожидает, что на первых позициях выдачи окажутся именно те результаты, которые наиболее точно, полно и удобно отвечают на его информационную задачу. Именно поэтому этап ранжирования (ranking) считается одним из самых сложных, наукоёмких и критичных этапов в архитектуре любых современных поисковых механизмов, работающих на серверной стороне web-приложений.

## Эволюция подходов к ранжированию: от простейших метрик к сложным системам

Самой первой и долгое время основной моделью оценки релевантности была TF-IDF (Term Frequency — Inverse Document Frequency). Эта метрика оценивает важность каждого слова (термина) в конкретном документе относительно всей коллекции документов.

TF (частота термина) измеряет, насколько часто искомое слово встречается именно в этом документе. Чем чаще — тем выше считается значимость термина для данного текста.

IDF (обратная документная частота) показывает, насколько редко слово встречается во всей коллекции в целом. Редкие термины получают высокий вес, потому что они более информативны и специфичны.

Классическая формула выглядит так:

$$score(t, d) = TF(t, d) \times \log(N / df(t)),$$

где N — общее количество документов в коллекции, df(t) — количество документов, в которых встречается термин t.

TF-IDF была очень успешной в 1990-е и начале 2000-х годов благодаря своей простоте, скорости вычисления и отсутствию необходимости в обучении. Однако по мере роста объёмов данных и повышения требований пользователей стали очевидны её фундаментальные недостатки:

• отсутствие механизма насыщения частоты — документ со 100 повторениями ключевого слова получал почти в 100 раз больший вес, чем документ с одним точным упоминанием (что поощряло спам и переоптимизацию);

• отсутствие нормализации длины документа — длинные тексты искусственно выигрывали за счёт большего количества возможных вхождений;

• крайне слабая работа с многословными запросами, синонимами, порядком слов и семантикой;

• чрезмерная чувствительность к очень редким терминам.

Именно эти проблемы привели к созданию более совершенной модели.

## Okapi BM25 — текущий золотой стандарт ранжирования

В середине 1990-х годов в рамках проекта Okapi (City University London) была разработана вероятностная модель Okapi BM25 (Best Matching 25 — 25-я версия улучшений), которая к настоящему моменту стала абсолютным стандартом де-факто во всех серьёзных поисковых системах мира: Google, Яндекс, Bing, Baidu, Elasticsearch, Apache Solr, OpenSearch, Azure Cognitive Search и многих других.

BM25 сохраняет общую идею TF × IDF, но радикально улучшает обе компоненты.

Усовершенствованная TF-часть вводит насыщение и нормализацию длины документа:

$$TF\_BM25(t, d) = \frac{(f(t,d) \times (k_1 + 1))}{(f(t,d) + k_1 \times (1 - b + b \times (dl / avgdl)))}$$

где:

- $f(t,d)$ — частота термина в документе
- $dl$ — длина документа (в терминах)
- $avgdl$ — средняя длина документа по коллекции
- $k_1$ — коэффициент насыщения частоты (обычно 1.2–2.0)
- $b$ — коэффициент влияния длины документа (обычно 0.75)

IDF-часть также уточнена:

$IDF(t) = \log( (N - df(t) + 0.5) / (df(t) + 0.5) )$

Такая форма даёт более стабильные результаты на коллекциях любого размера и предотвращает крайние значения.

### Практическое значение параметров $k_1$ и b

Параметры $k_1$ и b — это единственные свободные настройки базовой BM25, которые разработчики могут и должны подбирать под свою коллекцию.

Наиболее часто используемые и проверенные десятилетиями значения:

$k_1 = 1.2$

$b = 0.75$

Они считаются «золотым стандартом» и используются по умолчанию почти во всех движках.

Типичные корректировки в зависимости от характера данных:

- Короткие тексты (заголовки, описания товаров, твиты, новости) → $k_1 = 0.8–1.0$ (быстрое насыщение)
- Длинные тексты (статьи, научные работы, техническая документация) → $k_1 = 1.6–2.0$, $b = 0.6–0.7$
- Сильно неоднородная коллекция → лучше оставить дефолтные значения и проводить A/B-тестирование
- Много спама и переоптимизированного контента → снижать $k_1$ до 0.9–1.1

### Дополнительные механизмы повышения качества ранжирования

В реальных высоконагруженных системах чистый BM25 практически никогда не используется в одиночку. Над базовой моделью накладывается многослойная система бустов и модификаторов:

- Буст по важности полей

  Заголовок (title) обычно в 3–10 раз важнее основного текста (body). Анкоры ссылок, H1–H3, мета-описание, alt-тексты изображений получают повышенные коэффициенты.
- Буст по свежести

Новые документы получают значительный бонус. Чаще всего используется экспоненциальное затухание:

freshness_boost = base × exp(-возраст_в_днях / период_полураспада)

- Буст по популярности

  Учитываются просмотры, лайки, репосты, комментарии, CTR в выдаче. Особенно важен для социальных сетей, интернет-магазинов, видеохостингов.

- Персонализация

  Учёт предыдущих запросов, кликов, геолокации, демографии пользователя.

**Современные многоступенчатые архитектуры ранжирования**

В 2025–2026 годах типичная архитектура ранжирования в крупных системах выглядит как многоуровневая воронка:

1. Быстрый грубый отбор (1000–20 000 кандидатов) — BM25 + фильтры + векторный поиск (dense retrieval на эмбеддингах)
2. Средний этап — лёгкие нейронные cross-encoder модели → топ-200–500
3. Финальное ранжирование — тяжёлые трансформерные модели (типа BERT, ColBERT, monoT5 и их наследников), которые глубоко понимают семантику запроса и документа

Такая схема позволяет сочетать высокую скорость классических статистических методов с превосходной точностью современных нейросетей.

## Масштабируемость и отказоустойчивость: шардирование и репликация

Когда объём индексируемых данных превышает физические возможности одного сервера — будь то ограничения по объёму диска, оперативной памяти, процессорной мощности или пропускной способности ввода-вывода — дальнейшее вертикальное масштабирование (покупка более мощного железа) становится неэффективным и дорогостоящим. В этот момент единственным рациональным решением является переход к горизонтальному масштабированию: добавление новых серверов (нод) в распределённый кластер. Два ключевых механизма, которые позволяют реализовать такое масштабирование в современных поисковых системах, — это шардирование и репликация.

### Шардирование: принцип разделения данных

Шардирование (sharding) представляет собой процесс логического и физического разбиения единого индекса на несколько независимых частей — шардов (shards). Каждый шард — это самодостаточный фрагмент индекса, который содержит свою долю всех документов коллекции, свой словарь терминов, свои постинг-листы и другие вспомогательные структуры. Шарды полностью независимы друг от друга и могут размещаться на разных физических серверах или даже в разных дата-центрах.

Основные преимущества шардирования:

- Распределение объёма данных по нескольким машинам, чтобы избежать исчерпания ресурсов на одной ноде.
- Параллельное выполнение поисковых запросов — каждый шард обрабатывает свою часть данных одновременно.
- Линейный рост производительности: при добавлении новых нод кластер становится быстрее пропорционально количеству добавленных ресурсов (при условии правильной балансировки).
- Возможность работать с очень большими коллекциями — от десятков терабайт до петабайт данных.

При индексации каждый новый документ направляется в конкретный шард по

определённому правилу маршрутизации. Чаще всего используется хеш-функция от уникального идентификатора документа (_id), что обеспечивает равномерное распределение. В некоторых случаях применяется маршрутизация по значению определённого поля (routing), например по категории товара или по региону пользователя — это позволяет локализовать запросы и повысить скорость.

При выполнении поискового запроса процесс происходит в две фазы:

- Scatter (рассеивание) — координатор кластера отправляет запрос параллельно на все шарды.
- Gather (сбор) — каждая нода возвращает свои топ-результаты, координатор объединяет их, выполняет финальное ранжирование и возвращает пользователю отсортированную выдачу.

Выбор количества первичных шардов — один из самых ответственных архитектурных решений. Количество первичных шардов задаётся при создании индекса и практически не подлежит изменению без полной переиндексации данных. Рекомендации на 2025–2026 годы выглядят примерно так:

- Небольшие системы (до 50 ГБ) — 3–5 первичных шардов
- Средние системы (50 ГБ – 1 ТБ) — 5–15 шардов
- Крупные системы (1–50 ТБ) — 15–50 шардов
- Очень крупные системы (50+ ТБ) — 50–200+ шардов

Слишком малое количество шардов ограничивает будущую масштабируемость, слишком большое — создаёт избыточный overhead на управление тысячами мелких сегментов.

**Репликация: обеспечение надёжности и масштабирования чтения**

Репликация — это создание одной или нескольких идентичных копий каждого первичного шарда. Эти копии называются репликами (replicas).

Реплики выполняют две основные функции:

- Повышение отказоустойчивости

  Если нода с первичным шардом выходит из строя (аппаратный отказ, перезагрузка ОС, проблемы с сетью), кластер автоматически выбирает одну

из реплик и повышает её до статуса primary (процесс называется failover). В современных движках этот переход занимает от нескольких секунд до десятков секунд. После восстановления старой ноды реплики синхронизируются, и шард может вернуться на исходное место.

- Масштабирование нагрузки на чтение

  Поисковые запросы могут выполняться как на первичных шардах, так и на любых репликах. Чем больше реплик — тем больше параллельных запросов может обработать кластер без перегрузки первичных шардов. Это особенно важно в системах с высоким соотношением чтение/запись (поиск >> индексация).

**Типичные стратегии репликации в продакшене:**

- 0 реплик — только для разработки или очень маленьких тестовых систем (нет отказоустойчивости)
- 1 реплика — минимально приемлемый уровень для большинства приложений (всего 2 копии данных)
- 2 реплики — стандарт для высоконагруженных систем (всего 3 копии)
- 3+ реплик — для систем с жёсткими требованиями к доступности (99.99%+ uptime)

**Реализация в Elasticsearch и OpenSearch**

Ведущие распределённые поисковые движки (Elasticsearch, OpenSearch) делают шардирование и репликацию максимально прозрачными и автоматическими:

- Кластер сам распределяет первичные шарды и реплики по всем доступным нодам с учётом баланса нагрузки.
- При добавлении новой ноды происходит автоматический rebalancing — шарды плавно перемещаются без остановки сервиса.
- Поддерживается динамическое изменение количества реплик (можно увеличить с 1 до 3 «на лету»).

- Автоматический failover и failback.
- Разделение ролей нод: master-eligible, data, coordinating, ingest и др., что позволяет строить гибкие топологии кластера.

### Интеграция поискового механизма с серверным web-приложением

В большинстве современных серверных web-приложений поисковый движок (Elasticsearch, OpenSearch, Apache Solr, Typesense и др.) не встраивается напрямую в код основного приложения, а работает как самостоятельный распределённый сервис. Такая архитектура позволяет независимо масштабировать поиск, обновлять его без перезапуска основного приложения и использовать специализированные инструменты для индексации и ранжирования. Общение между основным бэкендом и поисковым движком происходит либо по протоколу HTTP (через REST API), либо с помощью официальных клиентских библиотек для конкретного языка программирования.

### Основные способы взаимодействия приложения с поисковым движком

Для большинства языков и фреймворков существуют удобные официальные или хорошо поддерживаемые клиенты:

- Python → elasticsearch-py / opensearch-py
- Java / Kotlin → официальный Java High Level REST Client или новый Elasticsearch Java API Client
- Node.js → @elastic/elasticsearch
- PHP → elasticsearch/elasticsearch
- Go → elastic/go-elasticsearch
- Ruby → elasticsearch-ruby

Использование клиентской библиотеки предпочтительнее прямых HTTP-запросов, поскольку она абстрагирует низкоуровневые детали (аутентификацию, retry-логику, пул соединений, сериализацию, обработку ошибок), а также предоставляет типизированные методы для наиболее частых операций: index, update, delete, search, bulk и т. д.

### Паттерны индексации данных

Существует несколько основных стратегий синхронизации данных между основной базой данных приложения и поисковым индексом. Выбор паттерна

зависит от требований к актуальности данных, допустимой задержки, объёма изменений и надёжности системы.

1. Синхронная индексация

При создании, обновлении или удалении сущности в основной реляционной базе данных (PostgreSQL, MySQL, MariaDB и т.п.) приложение сразу же в том же транзакционном контексте или сразу после успешного коммита отправляет соответствующий документ в поисковый движок.

Примерный жизненный цикл:

- Пользователь создаёт новый товар → сохраняется в PostgreSQL → после commit вызывается client.index(...) в Elasticsearch
- Преимущества: простота реализации, данные в поиске появляются мгновенно (или почти мгновенно)
- Недостатки: если поисковый движок временно недоступен, операция сохранения может завершиться ошибкой или зависнуть; увеличивается время ответа пользователю; при высокой нагрузке создаётся дополнительная нагрузка на основной запрос

Этот подход подходит для приложений с невысокой нагрузкой или когда критична 100%-ная консистентность между БД и поиском.

2. Асинхронная индексация через очередь сообщений

Самый популярный и рекомендуемый паттерн для средних и крупных систем.

После успешного сохранения в основную БД приложение публикует событие в очередь сообщений (RabbitMQ, Kafka, Redis Streams, AWS SQS и др.). Отдельный потребитель (worker, background job) читает события из очереди и выполняет операции индексации в поисковом движке.

Преимущества:

- Отказоустойчивость: если поисковый движок недоступен, события накапливаются в очереди и будут обработаны позже
- Разделение ответственности: основной запрос пользователя не блокируется
- Легко масштабировать: можно запустить несколько воркеров-потребителей
- Возможность реализации retry-механизмов, dead letter queue и мониторинга

отставания

Типичный стек: PostgreSQL → транзакционный outbox или триггер → Kafka / RabbitMQ → consumer (например, на Celery, Sidekiq, Laravel Queue, Spring Kafka) → Elasticsearch.

3. Change Data Capture (CDC) — автоматическое отслеживание изменений

Наиболее продвинутый и надёжный способ для высоконагруженных систем, где важна максимальная консистентность без вмешательства в бизнес-логику приложения.

Инструменты CDC (Debezium, Maxwell, pglogical, Oracle GoldenGate и др.) отслеживают изменения в журналах транзакций базы данных (WAL в PostgreSQL, binlog в MySQL) и публикуют их как поток событий в Kafka или другую шину. Затем коннектор (например, Debezium Elasticsearch Sink Connector) автоматически преобразует эти события в операции индексации/обновления/удаления в поисковом движке.

Преимущества:

- Полностью декларативный подход — не нужно писать код индексации в приложении
- Гарантированная доставка и порядок событий
- Поддержка схематизированных изменений (schema evolution)
- Минимальная нагрузка на основную БД

Этот паттерн широко используется в крупных e-commerce, финтехе и enterprise-системах.

**Near Real-Time возможности современных движков**

В Elasticsearch и OpenSearch реализован принцип near real-time (NRT) поиска: после отправки документа в индекс он становится видимым для поиска не мгновенно, а через очень короткий интервал — обычно 1–2 секунды (контролируется параметром index.refresh_interval). Это компромисс между скоростью индексации и производительностью: частые refresh'ы замедляют запись, редкие — увеличивают задержку видимости данных.

Для сценариев, где нужна мгновенная видимость (например, добавление комментария в соцсети), можно использовать параметр refresh=true в запросе индексации — в этом случае данные сразу попадут в поиск, но это значительно снижает пропускную способность записи.

### Интеграция поиска на стороне фронтенда через API

Поисковый движок предоставляет REST API, через которое фронтенд (или мобильное приложение) получает результаты. Основные возможности, которые обычно реализуются:

- Автодополнение (completion suggester, search-as-you-type, prefix/phrase suggester) — быстрые подсказки при вводе текста
- Пагинация (from/size или search_after) — эффективная постраничная выдача больших результатов
- Фасетная навигация (aggregations) — динамические фильтры по категориям, брендам, ценовым диапазонам, рейтингу, цвету и т.д. с подсчётом количества документов в каждом бакете
- Подсветка совпадений (highlighting) — выделение жирным тех фрагментов текста, где найдены термины запроса
- Сортировка по релевантности, цене, дате, популярности
- Фильтры (must/must_not/should) — по диапазонам, геолокации, булевым условиям

Типичный ответ от API содержит не только список документов, но и метаданные: общее количество совпадений, время выполнения запроса, агрегации для фасетов, подсвеченные фрагменты.

**Дополнительные возможности современных поисковых систем**

Современные поисковые движки, такие как Elasticsearch, OpenSearch, Apache Solr и Typesense, давно вышли за рамки простого полнотекстового поиска по ключевым словам и предлагают богатый набор функций, которые превращают поиск в мощный аналитический и пользовательский инструмент. Эти возможности позволяют не просто находить документы, а предоставлять пользователям удобную, интуитивную и интеллектуальную навигацию по большим объёмам данных.

Одной из самых востребованных функций стал фасетный поиск (faceted search) вместе с агрегациями. Он позволяет в реальном времени подсчитывать количество документов, соответствующих различным значениям полей: например, сколько товаров в категории «Смартфоны», сколько из них стоят от 30 до 50 тысяч рублей, сколько произведены в Китае или имеют рейтинг выше 4.5 звёзд. Эти подсчёты отображаются как динамические фильтры, которые пользователь может применять последовательно, сужая результаты поиска без перезагрузки страницы. Агрегации также используются для построения гистограмм цен, распределения по датам или географическим регионам.

Не менее важной стала поддержка геопоиска. Движки позволяют индексировать координаты (широта и долгота), выполнять поиск по радиусу («найди все кафе в радиусе 5 км от моей позиции»), сортировать результаты по расстоянию от заданной точки или применять геофильтры в комбинации с другими условиями. Это критически важно для приложений доставки, карт, локального бизнеса и туристических сервисов.

Современные системы отлично работают с вложенными документами и массивами объектов. Например, в одном документе «заказ» можно хранить массив товаров со своими характеристиками, ценами и количеством, и при этом искать по любому вложенному полю, подсчитывать агрегации по вложенным элементам или применять фильтры на уровне вложенных объектов.

С 2023–2025 годов бурно развивается векторный поиск и гибридный поиск. Благодаря интеграции моделей машинного обучения (BERT, Sentence Transformers,

ColBERT, E5 и др.) документы и запросы преобразуются в плотные векторы (эмбеддинги), после чего поиск выполняется не по точному совпадению слов, а по семантической близости. Это позволяет находить документы, которые по смыслу соответствуют запросу, даже если в них нет ни одного общего слова: «как быстро похудеть» → статьи про «дефицит калорий и тренировки». Гибридный поиск сочетает классический BM25 с векторным поиском, применяя веса к каждому типу и объединяя результаты для максимальной точности и полноты.

Дополнительно движки предоставляют мощные инструменты работы с текстом: автоматическую поддержку синонимов (можно задать, что «смартфон», «телефон», «мобильник» и «гаджет» — эквивалентны), фильтрацию стоп-слов, разбиение на n-граммы и особенно edge n-gram для реализации быстрого и точного автодополнения при вводе текста (search-as-you-type). Например, при вводе «самс» система мгновенно предложит «Samsung Galaxy», «Samsung S24», «Samsung Watch».

Наконец, современные движки позволяют применять сложные фильтры практически по любым типам данных: по диапазонам дат (созданные после 2024 года), числовым интервалам (цена от 10 000 до 50 000), булевым условиям (в наличии = true, акция = false), а также комбинировать их с полнотекстовым поиском, геофильтрами и векторными условиями в одном запросе.

Все эти функции вместе делают поиск не просто инструментом нахождения текста, а полноценной системой навигации, анализа и персонализированного взаимодействия с данными в серверных web-приложениях.

## Заключение

Принципы построения поисковых механизмов в серверных web-приложениях представляют собой синтез классической теории информационного поиска и современных технологий распределённых систем. Инвертированный индекс остаётся сердцем любой эффективной системы, модель BM25 — наиболее универсальной базовой формулой релевантности, а шардирование с репликацией — ключом к масштабируемости и надёжности.

В 2025–2026 годах лидерами остаются Elasticsearch и Solr, построенные на Apache Lucene. Они позволяют создавать поисковые решения практически любой сложности — от простого поиска по каталогу товаров до сложных корпоративных систем с персонализацией, семантикой и обработкой миллиардов документов.

Понимание этих принципов необходимо любому разработчику серверных web-приложений, поскольку качественный поиск напрямую влияет на пользовательский опыт, конверсию и общую удовлетворённость продуктом.

## Список литературы


1. Устройство поисковых систем: базовый поиск и инвертированный индекс // Хабр. URL: https://habr.com/ru/articles/545634/ (дата обращения: 11.01.2026).
2. Погружение в недра Apache Lucene: архитектура индекса, выполнение поиска и репликация данных // Хабр. URL: https://habr.com/ru/articles/852666/ (дата обращения: 11.01.2026).
3. BM25 — как работает алгоритм релевантности в поиске и SEO // Spirit Digital. URL: https://spiritdigital.ru/chto-takoe-bm25 (дата обращения: 11.01.2026).
4. Полнотекстовый поиск // Википедия. URL: https://ru.wikipedia.org/wiki/Полнотекстовый_поиск (дата обращения: 11.01.2026).
5. Шардирование и репликация в Elasticsearch // Yandex Cloud Документация. URL: https://cloud.yandex.ru/docs/managed-elasticsearch/concepts/scalability-and-resilience (дата обращения: 11.01.2026).
6. Делаем поиск в веб-приложении с нуля // Хабр. URL: https://habr.com/ru/companies/joom/articles/526550/ (дата обращения: 11.01.2026).
7. Elasticsearch: что это, как работает умный поиск и чем он полезен // Korus Consulting. URL: https://omni.korusconsulting.ru/blog/elasticsearch-kak-rabotaet-umnyy-poisk-i-chem-on-polezen-b2b-prodazham-/ (дата обращения: 11.01.2026).
8. Поисковая система // Википедия. URL: https://ru.wikipedia.org/wiki/Поисковая_система (дата обращения: 11.01.2026).