

# Информационный ПОИСК

Индексация и булев поиск



Игорь Поляков

# Немного о себе

- Игорь Поляков
- Старший программист в Группе Инфраструктуры Поиска
- Работал над поисковыми подсказками для RuStore
- Работаю над поиском по ВК Видео

# План лекции

Термины, определения
Булев поиск
Обратный индекс
Препроцессинг документов
Структура обратного индекса и его эффективность
Построение больших индексов
Сжатие индекса
Оценка качества булевой модели
Домашнее задание

# Информационный поиск, модели поиска

- Что такое информационный поиск?
- Информационный поиск (ИП) — это процесс поиска в большой коллекции (хранящейся, как правило, в памяти компьютера) некоего неструктурированного материала (обычно документа), удовлетворяющего информационные потребности (Manning, 2011)
- Булева модель (Boolean Retrieval Model, BIR)
- Модели поиска с ранжированием (Ranked Retrieval Models)

# Документы

- Будем рассматривать только текстовые документы
- Коллекция документов (или текстовый корпус, особ. в лингвистике)
  - У каждого документа уникальный DocID из [1, ...)
  - Может быть большой: >10 млрд. документов весь Рунет

# Поисковые запросы

- Пользователь выражает свою информационную потребность в виде поискового запроса
- Средняя длина поискового запроса ~2.5 слова
- Запросы могут:
  - Быть неоднозначны (напр. *машина*) и не-грамматичны (напр. *вконтакте вход моя страница*)
  - Часто содержат ошибки и опечатки (*однакласники*)

# Слова

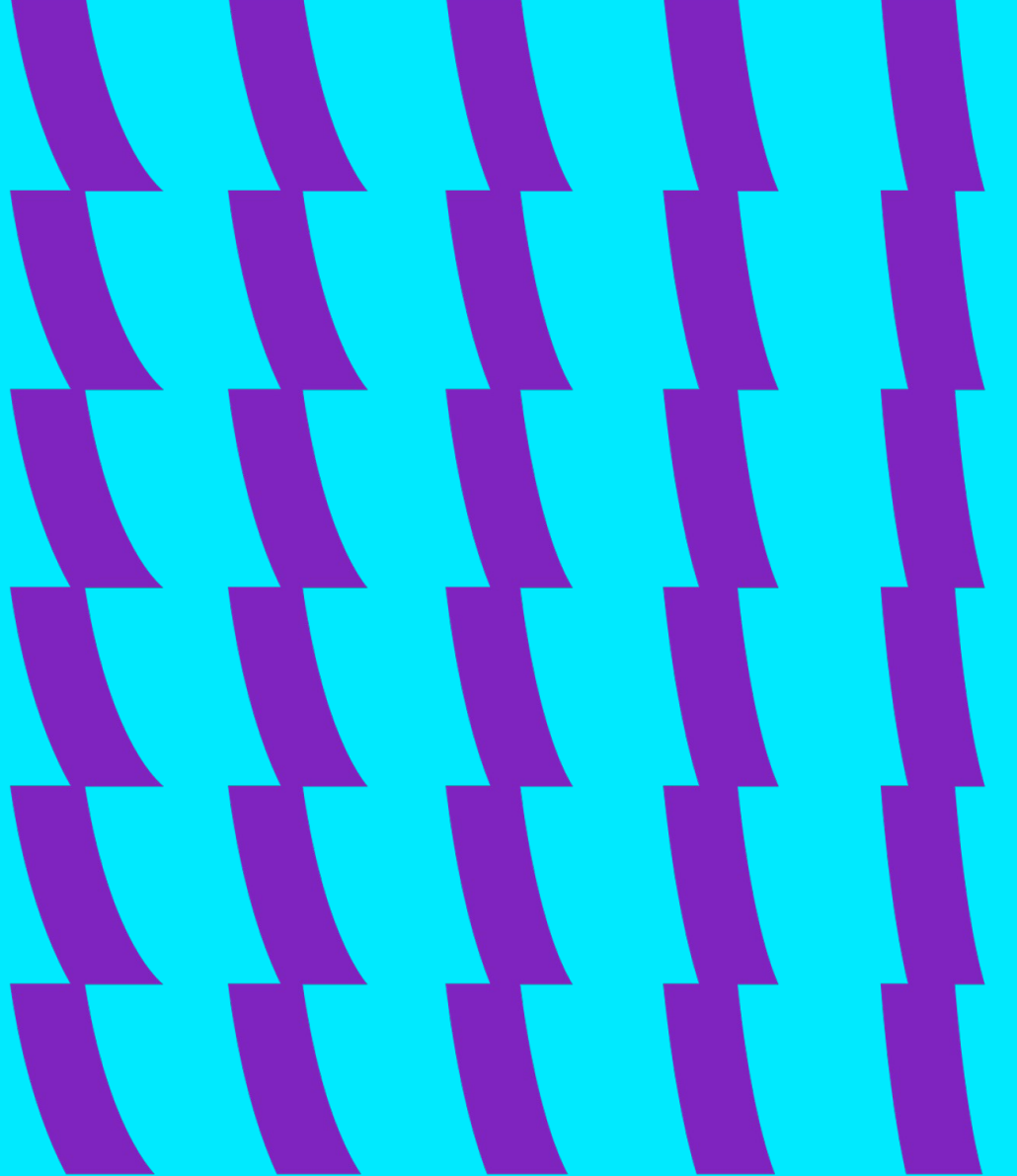
- Документы и запросы состоят из слов
- Но со словами есть проблемы
- Одно и то же слово может быть написано по-разному: МГУ и мгу
- Слова имеют формы и окончания
- По запросу *университеты* мы должны находить документы со словом *университет*
- Понятие слова довольно неопределенно
  - C++ – это слово?
- У слова может быть несколько значений
  - Омонимы: машина

# Термины

- Термины (terms) — базовые словоподобные единицы, на которые мы разбиваем документы и запросы
- Термины можно понимать как нормализованные слова: Москва → москва, университеты → университет, МГУ → мгу
  - Нижний регистр
  - Именительный падеж, единственное число и т.п.
  - Подробности позже
- Документы и запросы — последовательность терминов
  - Московский государственный университет → [московский, государственный, университет]



# Булев поиск



# Булева модель

- Моделируем запросы с помощью логических операций (кроме терминов добавляем операции):
  - конъюнкция (И):  $A \&\& B$
  - дизъюнкция (ИЛИ):  $A \parallel B$
  - отрицание (НЕ):  $\sim A$
- НЕ-запросы (отрицание)
  - Классическая булева модель допускает возможность использования оператора НЕ
  - Редко используется в современных поисковых системах т.к. сложно реализовать эффективно
  - В дальнейшем НЕ-запросы мы рассматривать НЕ будем

# И-запросы (конъюнкция терминов)

- Многословные запрос – это просто конъюнкция нескольких терминов
  - Московский государственный университет → московский && государственный && университет
- Такие запросы очень просто интерпретировать:
  - Документ должен содержать ВСЕ термины: И московский, И государственный, И университет

# ИЛИ-запросы (дизъюнкция терминов)

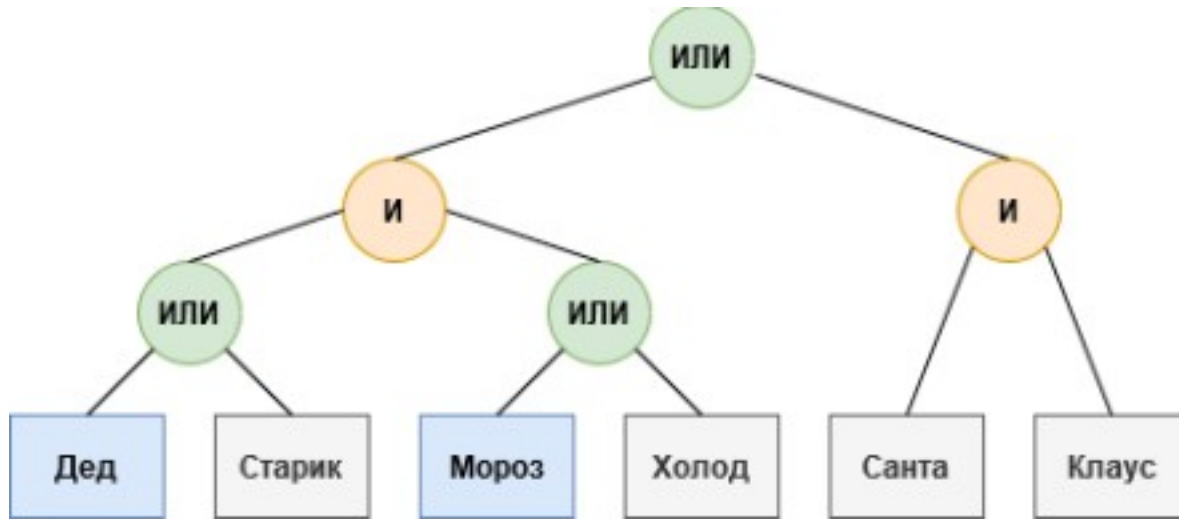
- По запросу университет мы хотим находить документы со словом *институт*
- Достигается с помощью расширения запроса синонимами
- Например:
  - Синонимы: {университет, институт}
  - *университет итмо* → *(университет || институт) && итмо*
- Либо пользователь может вручную указать “ИЛИ” в запросе, например: *кеды || кроссовки*
- Дизъюнкцию тоже просто интерпретировать:
  - Документ должен содержать ХОТЯ БЫ ОДИН из терминов: ИЛИ *университет*, ИЛИ *институт*

# Многословные синонимы

- Фраза из N слов может быть синонимом другой фразы из M слов, при этом отдельные слова могут не являться синонимами друг друга
- Примеры:
  - {дед мороз, санта клаус}
  - {япония, страна восходящего солнца}
- Запросы с многословными синонимами также представляются с помощью операторов ИЛИ и И
  - дед мороз → (дед && мороз) || (санта && клаус)

# Дерево запроса

- В общем случае запрос можно представить в виде дерева: узлы-операции и листья-термины
- Синонимы: {дед мороз, санта клаус}, {дед, старик}, {мороз, холод}



- Запрос: дед мороз  $\rightarrow ((\text{дед} \parallel \text{старик}) \&\& (\text{мороз} \parallel \text{холод})) \parallel (\text{санта} \&\& \text{клаус})$

# Тестовая коллекция документов

- Будем работать с 3 документами:
  - 1) Московский физико-технический институт
  - 2) Московский государственный университет
  - 3) Университет ИТМО
- Разобьем на термины:
  - 1) [московский, физика, технический, институт]
  - 2) [московский, государственный, университет]
  - 3) [университет, итмо]

# Матрица термин-документ

- Представим коллекцию документов в виде матрицы

	№1 МФТИ	№2 МГУ	№3 ИТМО
государственный	0	1	0
институт	1	0	0
ИТМО	0	0	1
московский	1	1	0
технический	1	0	0
физика	1	0	0
университет	0	1	1

- $M_{ij}=1$  если в  $j$ -м документе есть  $i$ -е слово



# Обработка И-запросов

- С помощью матрицы термин-документ легко найти множество документов, удовлетворяющих запросу-конъюнкции
- Запрос: *московский && университет*
- Применяем логическую операцию И к строкам
  - *московский* → 110
  - *Университет* → 011<sup>&</sup>
  - *<И>* = 010
- 010 соответствует 2-му столбцу: №2 <МГУ>

# Обработка ИЛИ-запросов

- Аналогично обрабатываем запросы-дизъюнкции
- Запрос: *московский && (институт || университет)*
- Применяем логические операции:
  - *институт* → 100     ||
  - *университет* → 011
  - <ИЛИ> = 111
  - *Московский* → 110     &&
  - <И> = 110
- 110 соответствует 1-му и 2-му столбцам: {№1<МФТИ>, №2 <МГУ>}

# Булева модель: достоинства и недостатки

- Достоинства:
  - Простота
  - Пользователь получает полный контроль над результатами поиска
  - Идеальная область применения: пользователи-эксперты, которые работают с небольшой коллекцией документов
- Недостатки:
  - “Обычные” пользователи не любят писать сложные логические выражения
  - Булева модель плохо работает для больших коллекций

# Булева модель: избыточная полнота и точность

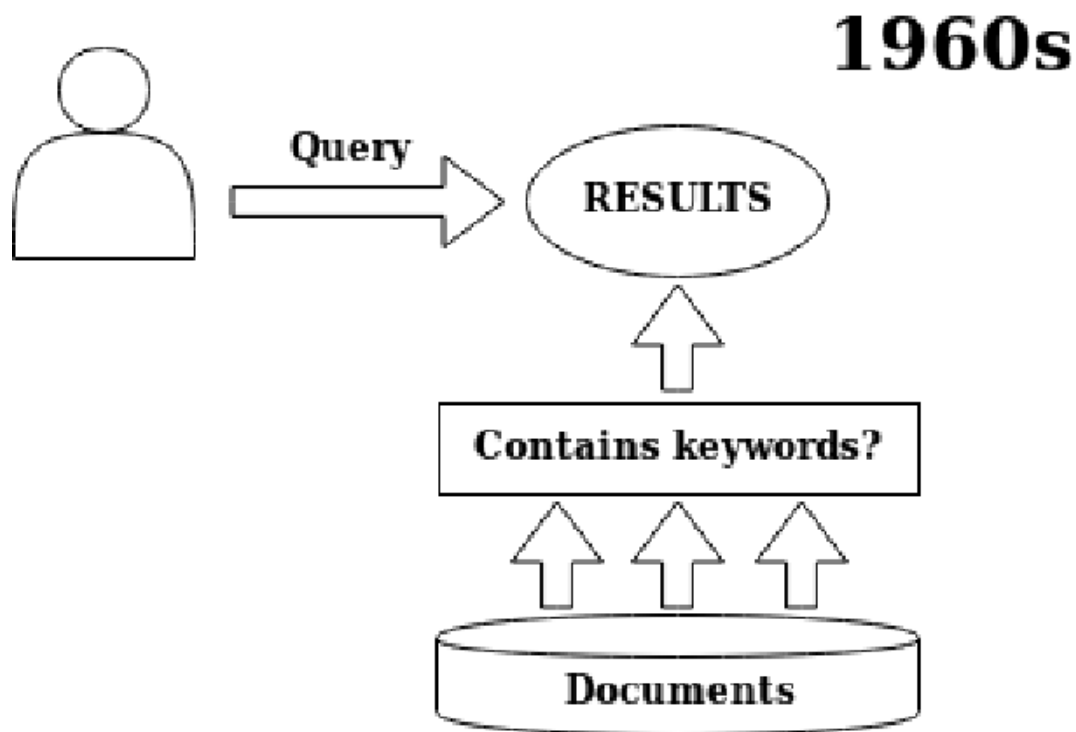
- Булевы запросы являются слишком “мягкими” и находят большое число малорелевантных документов в больших коллекциях
  - Пример: запрос *мгу* в веб-поиске (миллионы документов в коллекции, но пользователей интересуют только несколько самых релевантных)
- Верно и обратное: булевы запросы являются слишком “жесткими”
  - По запросу *лучшие университеты россии* мы не найдем документ *московский государственный университет*, если в нем нет слова *россия*
  - На практике пользователю очень сложно подобрать “руками” нужный баланс между точностью и полнотой

# Современная поисковая система

- Что нужно пользователю?
- Запросы в свободной форме
  - Без логических операторов
  - Смысл запроса!
- Ранжирование
  - Топ-10 из миллионов с ключевыми словами
- Скорость!
  - Время ответа  $< 1\text{с}$
- Под капотом любого современного поисковика работает булев поиск
- Чтобы это понять, сделаем небольшой экскурс в эволюцию поисковых систем

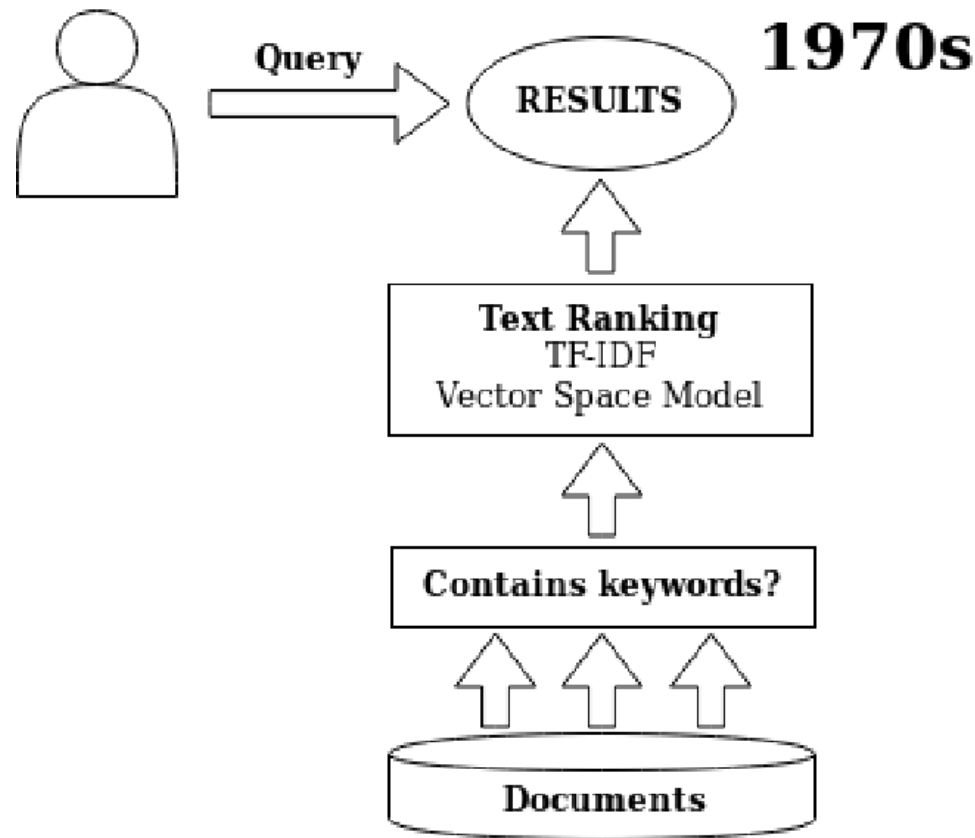
# Эволюция поисковых систем: 60-е

- Первые поисковики
- «Фильтрация» по ключевым словам
- Ранжирования еще нет
- Булева модель в чистом виде!



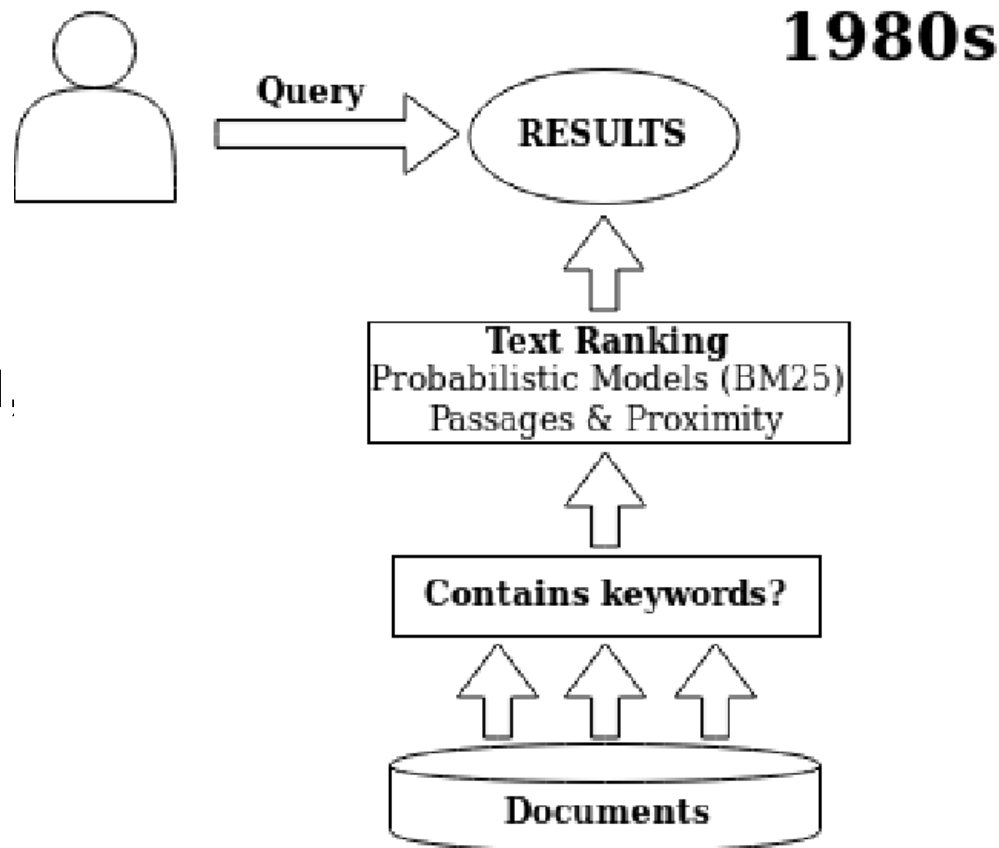
# Эволюция поисковых систем: 70-е

- Размер коллекций растет
- Примитивное ранжирование
  - Только по тексту
  - Кол-во слов в документе
  - Важность слов



# Эволюция поисковых систем: 80-е

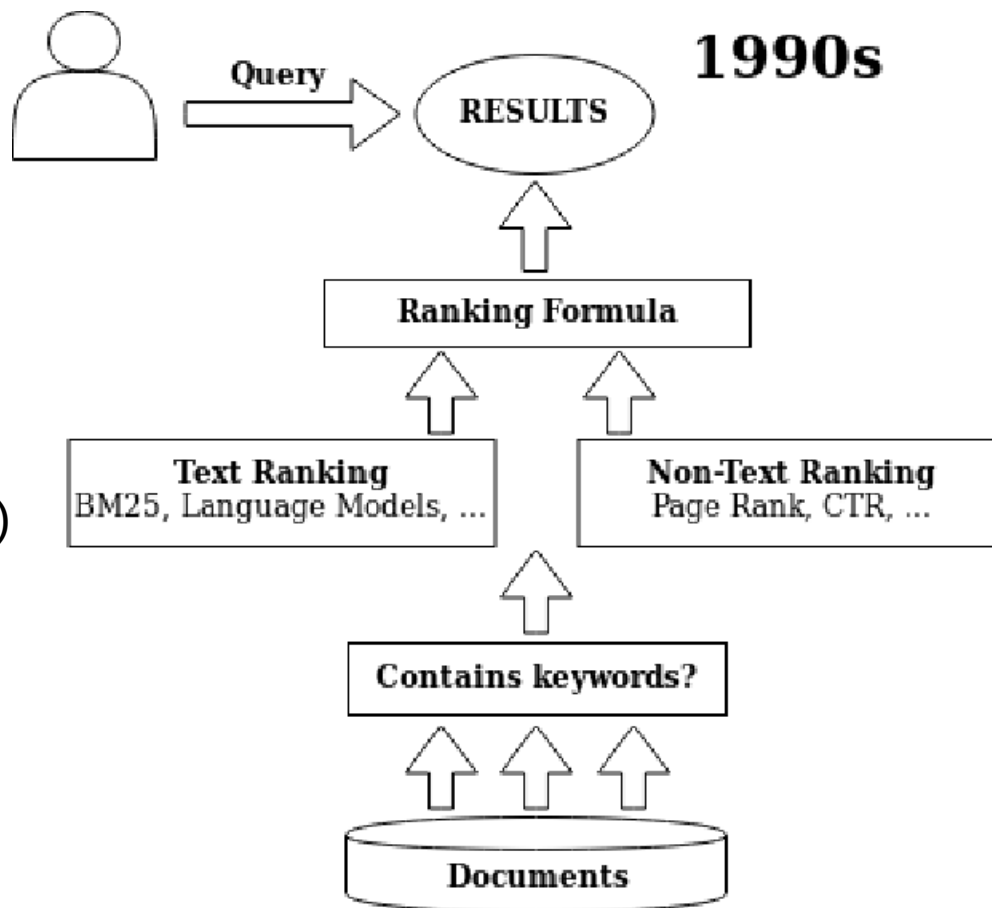
- Ранжирование становится лучше
- Все еще по тексту
  - Вероятностные модели, BM25
  - Близость и порядок слов





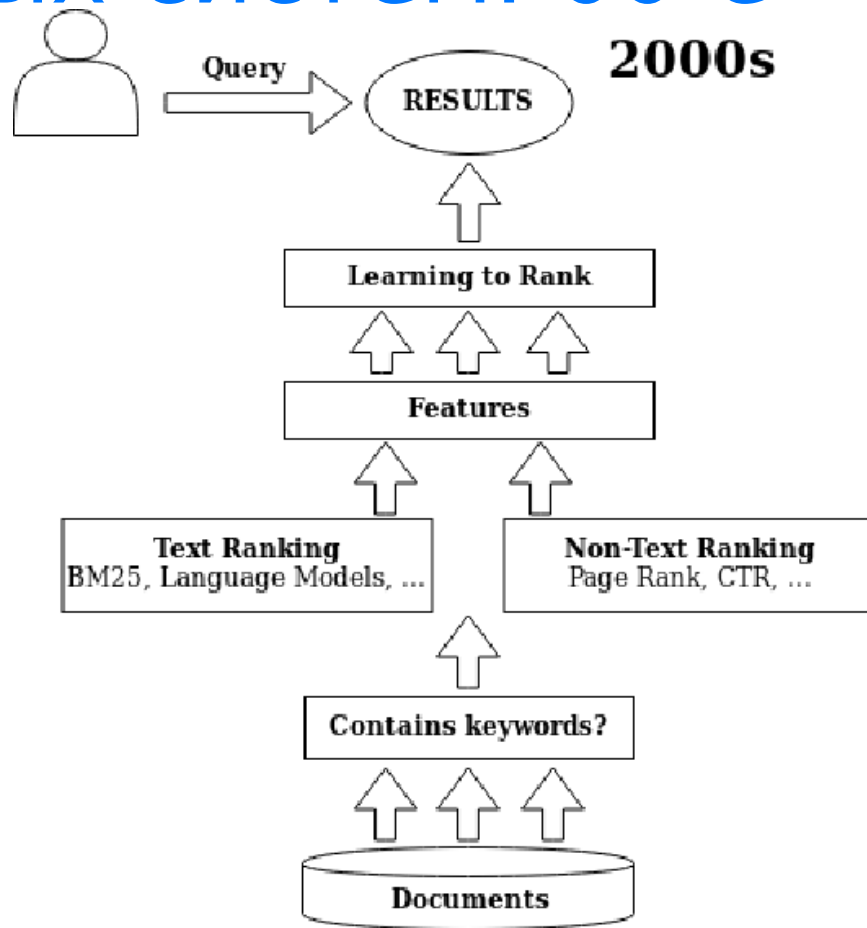
# Эволюция поисковых систем: 90-е

- Веб-поиск
- Не-текстовое ранжирование
  - Page Rank
  - Кликовая статистика (CTR)
- «Формула» ранжирования
  - $\text{Rank} = F(\text{BM25}, \text{PageRank}, \text{CTR}, \dots)$
  - Подбирается вручную
- Возникают понятия спама, SEO, ...



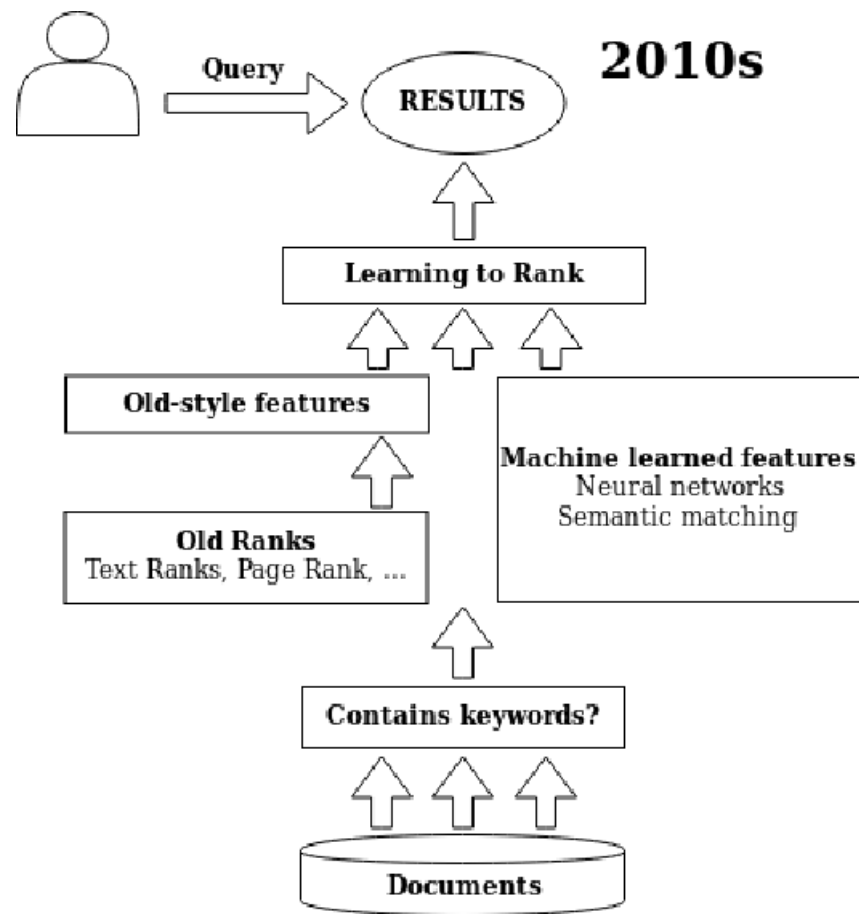
# Эволюция поисковых систем: 00-е

- Машинное обучение
  - Старые ранки → признаки
- Learning to Rank
  - Градиентный бустинг
  - Оптимизируем метрики ранж-ия (LambdaRank)
  - Тысячи признаков
- Большие данные
  - Миллиарды документов в коллекции
  - Терабайты статистики



# Эволюция поисковых систем: 10-е

- Машинное-обученные признаки
- Нейронные сети
- Глубокое обучение
- Семантика
- Пытаемся «понять» запросы
- Learning to Match

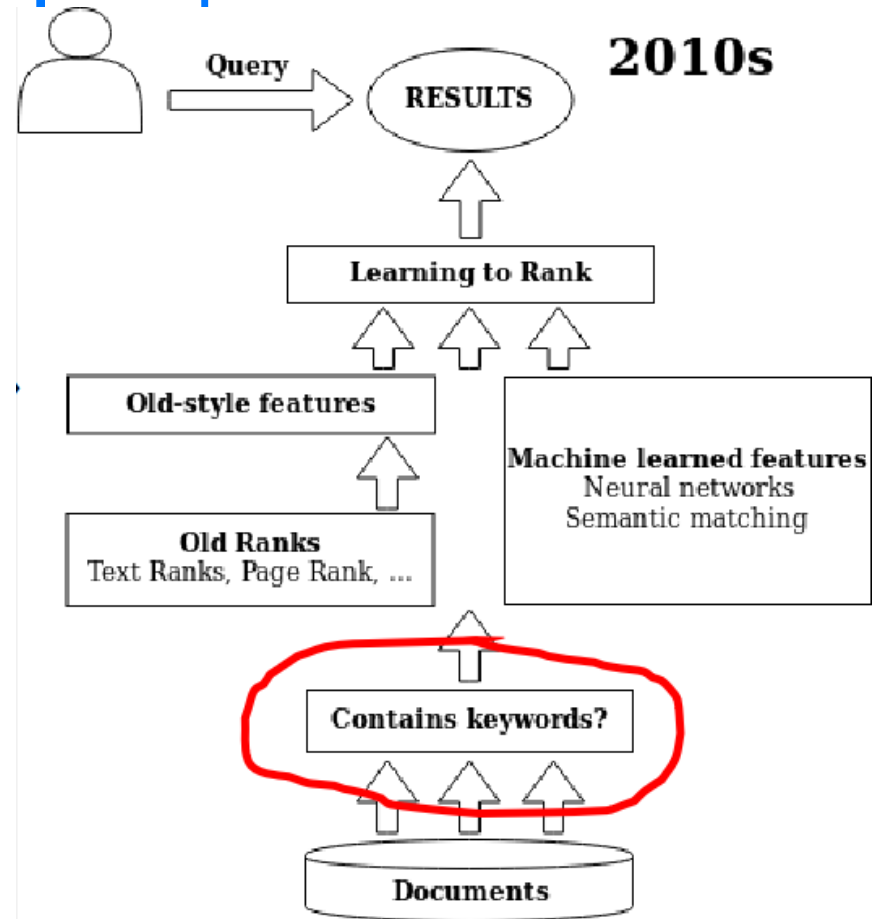


# Булев поиск сегодня


- “Мягкая” булева модель
  - На практике, в современных поисковиках работает «мягкий» вариант булевой модели
  - Оператор «мягкого» И – требуем, чтобы документ содержал не все, а только самые важные (про важность – в следующих лекциях) из слов запроса

# Слой фильтрации

- Архитектура «слоеного пирога»
- Новые технологии – надстройка над старыми
- Нижний слой «фильтрации»
  - Пропускает документы с ключевыми словами
  - Реализует булеву модель
  - До сих пор работает в каждом поисковике!



# Обратный индекс

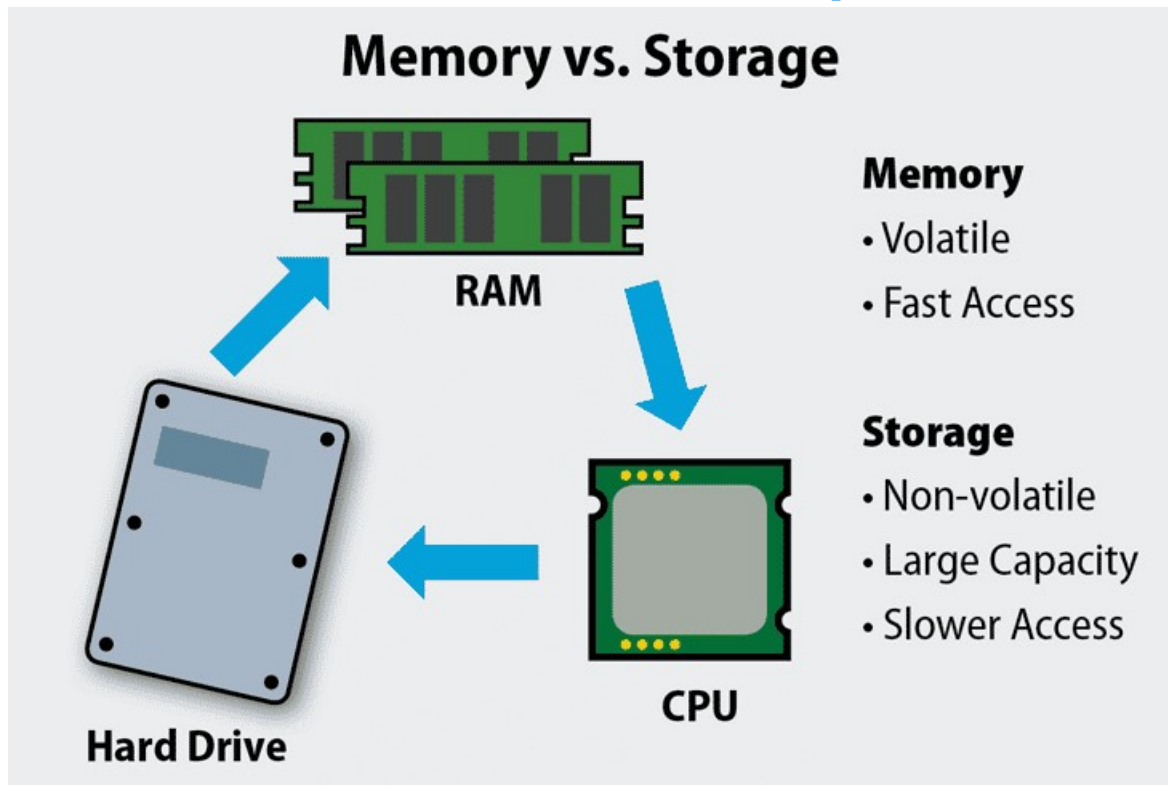
The background of the slide features a solid blue field overlaid with a pattern of thick, bright green curved lines. These lines are arranged in a series of parallel, slightly wavy bands that sweep across the frame from the bottom left towards the top right, creating a sense of dynamic movement.

# Как реализовать булеву модель?

- Существует эффективная реализация булевой модели на основе обратного индекса
- Но сначала:
  - Вспомним как работает современное компьютерное железо
  - Рассмотрим несколько наивных и неэффективных решений

# Современный компьютер

- Нам будут интересны только:
  - Процессор (CPU)
  - Оперативная память (RAM)
  - Внешняя память (HDD, SSD)





# Оперативная память (RAM): достоинства и недостатки

- + Быстрый произвольный доступ, типичное время:  
< 100нс
- + Очень надежна
- Малый объем: обычно  
< 512 Гб/сервер
- Данные не  
сохраняются при  
перезагрузке
- Дорогая!

# Жесткие диски (HDD): достоинства и недостатки

- + Большой объем: обычно > 8 Тб
- + Данные сохраняются при перезагрузке
- + Легко объединяются в массивы большого объема (RAID)
- + Дешевые!
- Медленный последовательный доступ
- Крайне медленный произвольный доступ (позиционирование головки), типичное время до 10 мс
- Быстро и часто ломаются

# Твердотельные накопители (SSD): достоинства и недостатки

- + Средний объем: обычно ~ 1 Тб
- + Данные сохраняются при перезагрузке
- + Быстрый (относительно HDD) случайный доступ: ~100 мкс
- + А у NVMe до 10 мкс!
- + Часто используются как кэш для данных на HDD
- Все еще значительно медленнее RAM
- Ограниченный срок службы
- Дорогие (относительно HDD)

# Иерархия памяти

- Типичные времена ожидания для разных типов оперативной и долговременной памяти

Память	Время ожидания	
L1 cache	0.5ns	
L2 cache	7ns	
RAM	100ns	200x L1 cache
Compress 1 KB (snappy)	3 $\mu$ s	Fast!
Network (send 1 KB over 1Gbps)	10 $\mu$ s	
SSD (read 4 KB randomly)	150 $\mu$ s	~1 GB/s
HDD seek	10 ms	
HDD (read 1 MB sequentially)	20 ms	

# Память — самый ценный ресурс!

- Памяти всегда мало
- Единственная возможная стратегия:
  - Большие «холодные» данные лежат на HDD
  - Маленькие «горячие» данные грузим в RAM
  - Используем SSD для «теплых» данных или под кэш

# Наивный подход к обработке булевых запросов

- Сколько памяти надо для хранения матрицы термин-документ?
- Типичная маленькая коллекция
  - 1 млн документов, 1 млрд терминов (~1 млн уникальных)
  - $\#(\text{Элементов матрицы Т-Д}) = 1 \text{ млн док-тов} * 1 \text{ млн уникал терминов} = 1 \text{ трлн}$
  - Потребуется 128 Гб из расчета 1 бит/элемент – это 1 мощный сервер
- Не будет работать в масштабах веб-поиска
  - 10 млрд док-тов в Рунете:  $> 1 \text{ PB RAM!!!}$

# Почему просто не использовать `grep`?

- Храним документы как текстовые файлы на диске
- Запрос: *московский университет*
- Делаем *grep университет \*.txt | grep московский*
- Типичная маленькая коллекция
  - 1 млн документов \* 1 млрд терминов \* 10 байт/термин = 10 Гб на диске
  - Типичная скорость последовательного чтения с HDD - ~150 Мб/с
  - `grep` с HDD займет > 1 минуты
  - Не будет работать даже для маленькой базы!

# Рабочее решение: обратный индекс

- Основная идея: матрица термин-документ **ОЧЕНЬ** разреженная
- Большинство слов встречаются только в очень небольшом числе документов
- Будем хранить только ненулевые элементы!
- Отображение: термин → список из ID документов, содержащих этот термин



# Обратный индекс: пример

- *государственный* → [2]
- *институт* → [1]
- *итмо* → [3]
- *московский* → [1, 2]
- *технический* → [1]
- *физика* → [1]
- *университет* → [2, 3]

Документы:

1. <МФТИ>
2. <МГУ>
3. <ИТМО>

- **Словопозиция** (posting) – элемент списка с информацией о наличии термина в док-те (в нашем примере просто DocID)
- **Словарь** – структура данных для хранения терминов

# Сколько нужно памяти под обратный индекс

- Точное потребление памяти зависит от распределения терминов
- Типичный случай: размер индекса  $\approx 10\%$  от размера коллекции
- Большая коллекция веб-масштаба
  - $10 \text{ млрд док-тов} * 1000 \text{ терминов/док-т} * 10 \text{ байт/термин} = 100 \text{ Тб}$
  - $\text{Размер индекса} = 0.1 * 100 \text{ Тб} = 10 \text{ Тб}$
  - Немало, но гораздо лучше чем было в наивном варианте

# Как хранить мультитерабайтный индекс?

- Предположим, что мы хотим полностью загрузить индекс в память
  - У нас нет машины с 10 Тб памяти ☹️
  - Но мы можем «порезать» (шардировать) индекс по кластеру
  - $10 \text{ Тб} / 48 \text{ Гб/сервер} \approx 200 \text{ серверов}$
  - На практике еще меньше т.к. существуют эффективные алгоритмы сжатия

# Как построить обратный индекс

- Начнем с простейшего случая
- Индекс полностью помещается в память на 1 машине
- Построение индекса также полностью происходит в памяти

# Построение обратного индекса на Python

- Используем словарь из множеств (dict of sets) для сбора информации по каждому термину

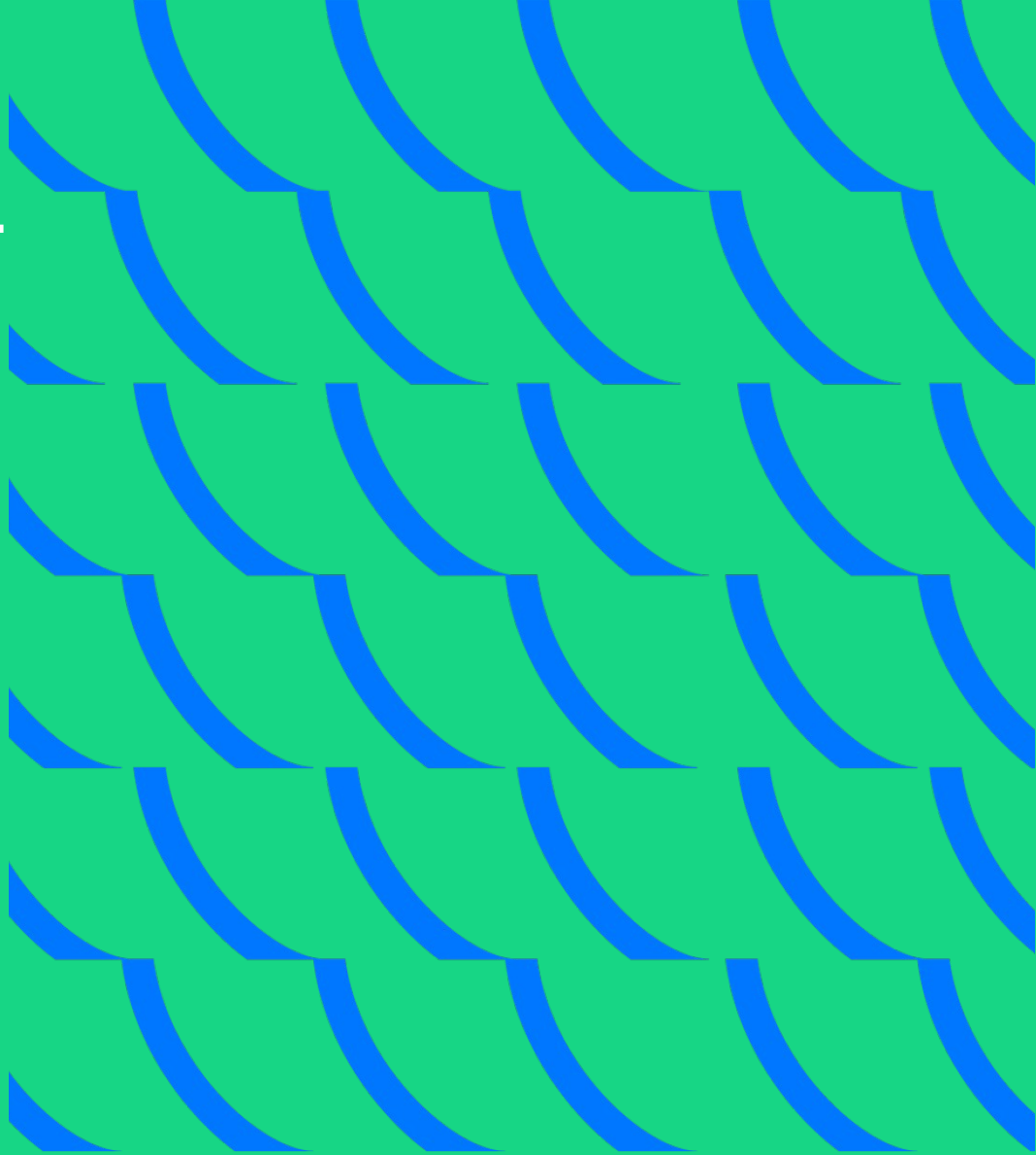
```
1  def build_index(docs):
2      inverted_index = {}
3      next_doc_id = 1
4      for doc in docs:
5          for word in doc:
6              postings = inverted_index.setdefault(word, set())
7              postings.add(next_doc_id)
8              next_doc_id += 1
9      return inverted_index
10
11  inverted_index = build_index([
12      ["moscow", "institute", "of", "physics", "and", "technology"],
13      ["moscow", "state", "university"],
14      ["itmo", "university"]
15  ])
16  print(inverted_index)
```

# Обработка запроса с помощью обратного индекса на Python

- Ищем в словаре множества словопозиций для каждого из терминов запроса
- Пересекаем множества с помощью оператора &

```
18 def search(query, inverted_index):
19     results = None
20     for word in query:
21         postings = inverted_index.get(word, set())
22         results = results & postings if results is not None else postings
23     return results
24
25 results = search(["moscow", "university"], inverted_index)
26 print(results)
```

# Препроцессинг документов



# Предобработка документов

- Мы рассматривали запросы и документы как последовательности терминов
- Все еще остаются открытыми вопросы:
  - Как именно происходит нормализация слов
  - Как превратить “сырой” документ в последовательность терминов?
- Конвейер из 3-х этапов
  - 1) Преобразование
  - 2) Токенизация
  - 3) Нормализация



# Этап №1: преобразование документа

- Поисковый робот качает документы в разных форматах
  - Текстовые (HTML) или бинарные (DOC, PDF, ...)
  - Кодировка тоже бывает разной: UTF-8, CP1251, ...
- На входе “сырые” бинарные документы
- Парсим разные форматы и достаем текст
- Определяем исходную кодировку текста и конвертируем его во внутреннюю кодировку поисковой системы (напр. UTF-8)
- На выходе: текст во внутренней кодировке

# Этап №2: токенизация

- Разбиваем текст на **лексемы (токены)**
- **Лексема** – это экземпляр последовательности символов, объединенных в семантическую единицу для обработки (Manning)
- Лексемы, как правило, соответствуют словам
- На входе: Московский физико-технический институт (МФТИ)
- Бьем по пробелам, удаляем знаки пунктуации
- На выходе список лексем: [Московский, физико,технический, институт, МФТИ]
- Много пограничных случаев: как быть с C++?
- Зависит от языка → нужен детектор языка!
- Например, в китайской письменности нет пробелов – токенизация может быть разной в зависимости от контекста

# Этап №2: токенизация — разные языки

- Китайский язык необычен тем, что у каждого символа есть смысл, но это не значит, что каждый символ (иероглиф) может быть воспринят как слово.
- Кроме того, значение слова не обязательно связано со значением иероглифов, составляющих его.
- Например, китайское слово кризис 危機 состоит из иероглифа “опасность” и иероглифа “возможность”. Однако второй иероглиф имеет значение ближе к “важный момент”, и значит “машина” в других контекстах.
- Этот же символ присутствует как первый иероглиф в слове “аэропорт” 機場, в котором второй иероглиф значит “поле”.
- Соединение частей слова правильным образом – очень важно, т.к. аэропорт это не “машинное поле”. Свобода 自由 - это не синоним для “само причина”.
- Многие слова в китайском состоят из 2 иероглифов, но не все. Поэтому для китайского хорошо работает биграмный индекс

# Этап №2: токенизация - пограничные случаи

- Все ли лексемы мы хотим добавлять в обратный индекс?
- E-mails
- URLs
- IP адреса
- Трекинг номера
- Финансовые суммы
- Числа
- Сильно раздувают размер словаря
- Можно исключать, но теряем возможность искать по ним

## Этап №3: нормализация

- Приведение лексем к нормальной форме
- На входе список лексем: [*Московский, физико, технический, институт, МФТИ*]
- Приводим к нижнему регистру
- Нормализуем: лексема → нормальная форма (стемминг или лемматизация, см. дальше)
- Удаляем стоп-слова (самые частотные, однобуквенные, ...)
- На выходе список терминов: [*московский, физика, технический, институт, мфти*]

# Нормализация с помощью стемминга

- Стемминг – приближенный эмпирический процесс, в ходе которого от слов отбрасываются окончания в расчете на то, что это себя оправдывает (Manning)
- Например:
  - fishing → fish
  - fished → fish
  - fisher → fish

# Стемминг: плюсы и минусы

- + Прост в реализации
- + Работает быстро
- + Хорошие результаты для “простых” языков (английский)
- Много коллизий у разных по смыслу слов (напр. стеммер Портера приводит universe и university к univers)
- Плохо работает для «сложных» языков, в которых у слова может быть множество форм (русский, финский, ...)

# Лемматизация

- **Лемматизация** – это точный процесс с использованием лексикона и морфологического анализа слов, в результате которого удаляются только флективные окончания и возвращается основная форма слова, называемая леммой (Manning)
- Например:
  - рыбаки → рыбак
  - рыбаками → рыбак
  - рыбаков → рыбак
- Плюс: хорошо работает для русского и других “сложных” языков
- Минусы:
  - Сложна в реализации, требует морфологического анализа слова
  - Мало качественных библиотек
  - Работает медленно
  - Для “простых” языков показывает результаты не сильно лучше стемминга
  - Бывают ошибки: Киев → Кий, Меган → Мигать



# Стемминг или лемматизация?

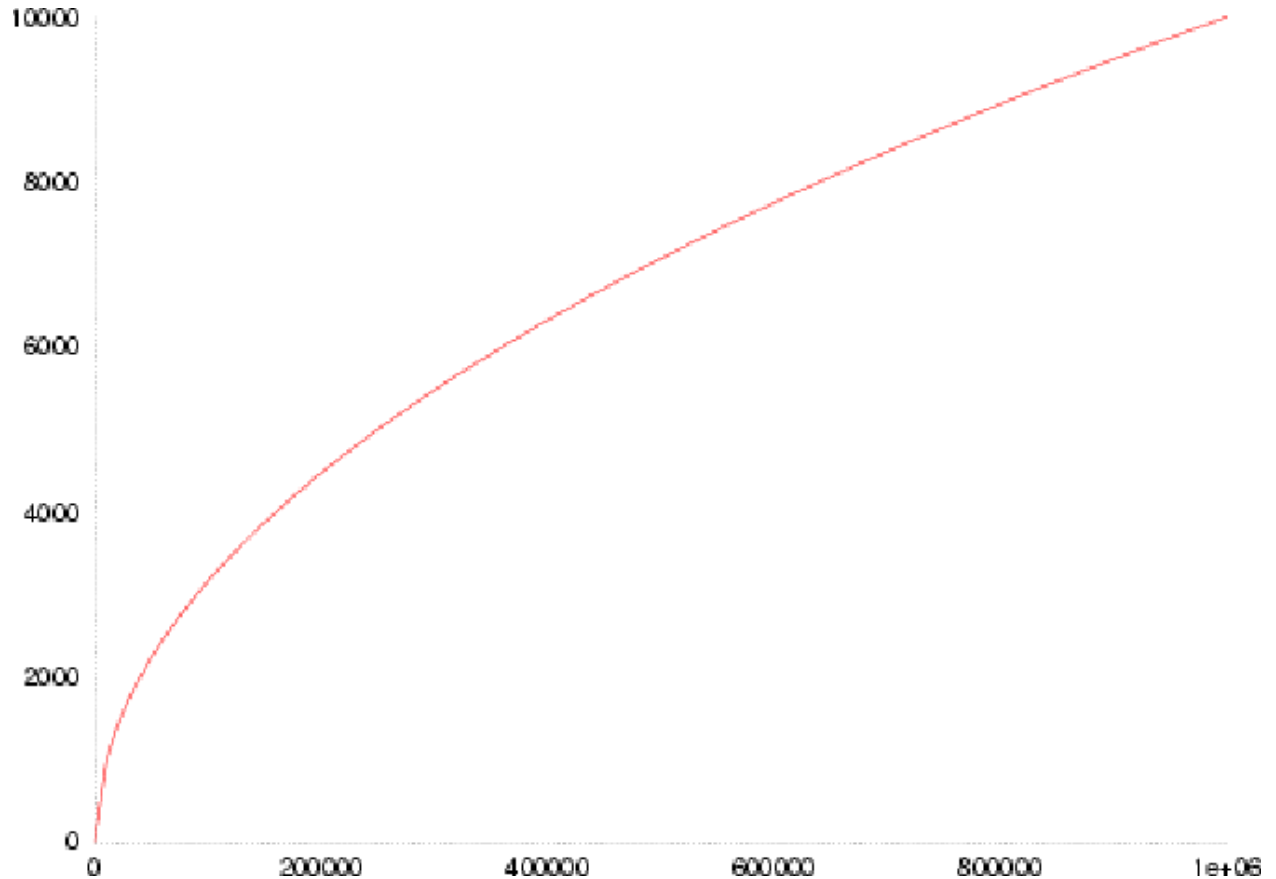
- Для английского языка лучше начинать со стемминга, и только при необходимости переходить к лемматизации
- Для русского языка необходимо использовать лемматизацию
- Все большие поисковые машины используют лемматизацию («морфология»)

# Предобработка запросов

- Симметричной предобработке подвергаются и запросы
- Очень важно, чтобы токенизация и нормализация для запросов и документов были идентичны

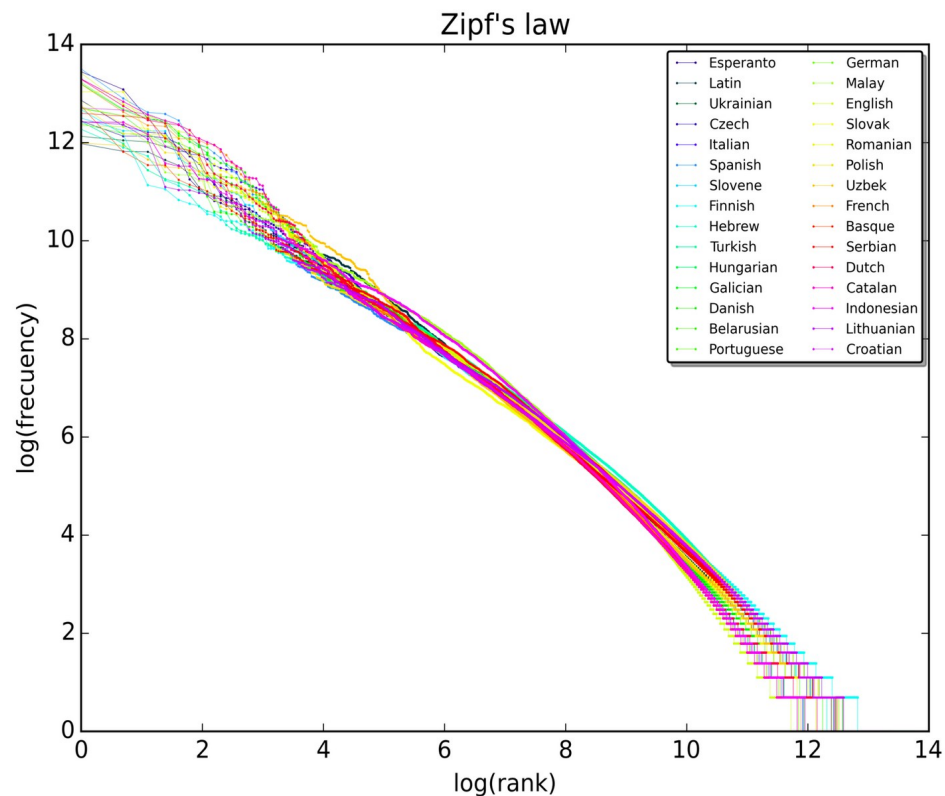
# Закон Хипса (Heaps' Law)

- Размер словаря  $|V|$  как функция количества лексем в коллекции  $T$
- $|V| = kT^b$ , где  $30 \leq k \leq 100$ ,  $b \approx 0.5$



# Закон Ципфа (Zipf' Law)

- Пусть  $cf_i$  – частота  $i$ -го по распространенности термина в коллекции
- $cf_i \propto 1/i$  или  $cf_i = ci^{-k}$  где  $k = 1$  (степенной закон)



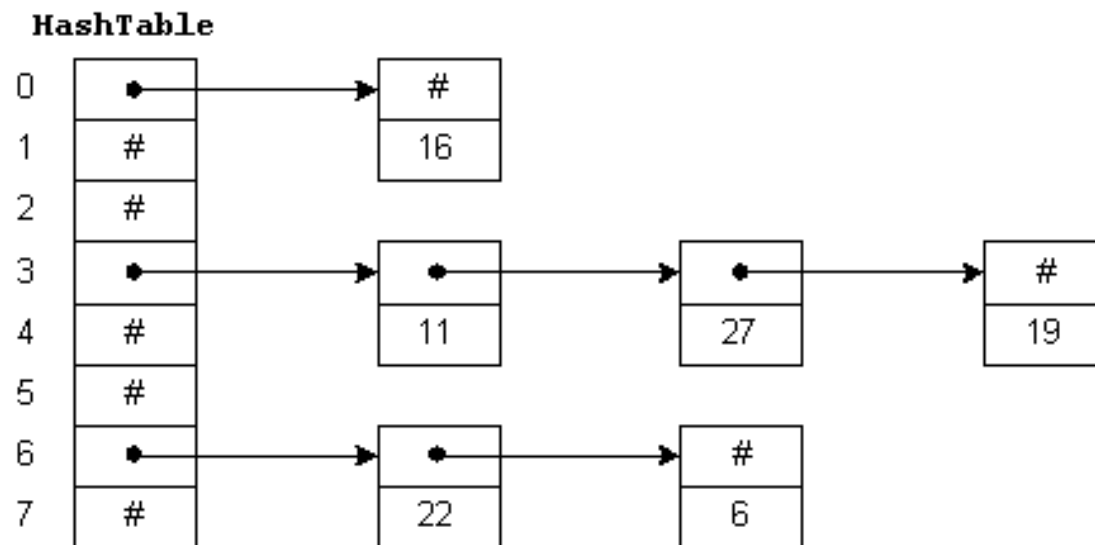
# Структура обратного индекса и его эффективность

# Какую структуру данных выбрать для словаря?

- Возможны несколько вариантов
  - Хэш-таблица
  - Дерево поиска (сбалансированное дерево поиска, В-дерево, ...)
  - Отсортированный список или массив
- Важный момент: словарь почти всегда помещается в память
  - Даже для веб-поиска размер словаря  $|V| < 100$  млн терминов

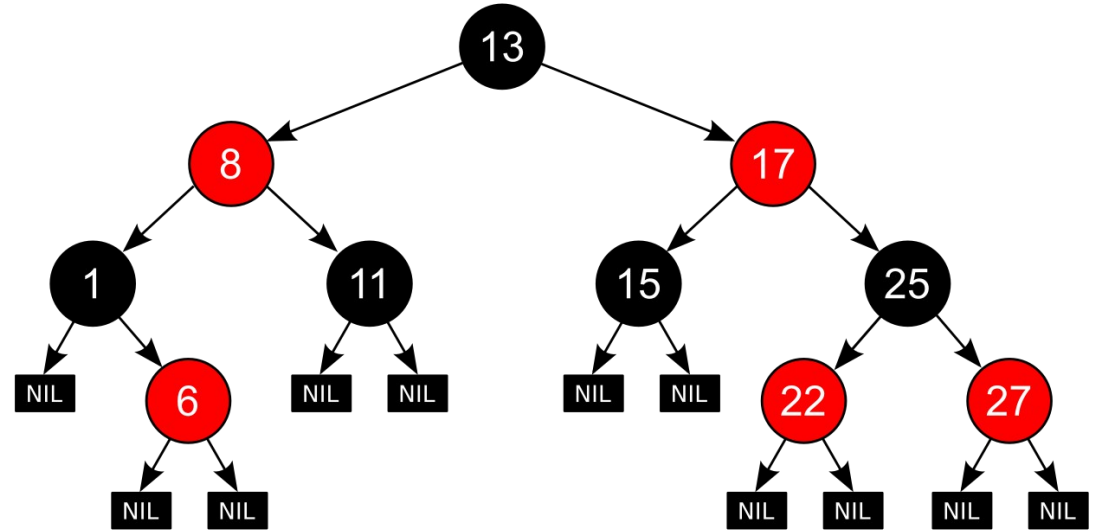
# Словарь на основе хэш-таблицы

- dict в Python использует хэш-таблицу «под капотом»
- Память используется не оптимально
- Хорошее решение для небольших индексов
- Реализовывали ли хэш-таблицы? Какая сложность вставки и поиска?
- Быстрые вставка и поиск за  $O(1)$



# Словарь на основе дерева поиска

- Вставка и поиск за  $O(\log(N))$
- Упорядочены: возможен поиск по префиксу!
- Много памяти расходуется на указатели
- «Рыхлые»: недружелюбны к кэшу ЦПУ
- Плохое решение если не нужен порядок



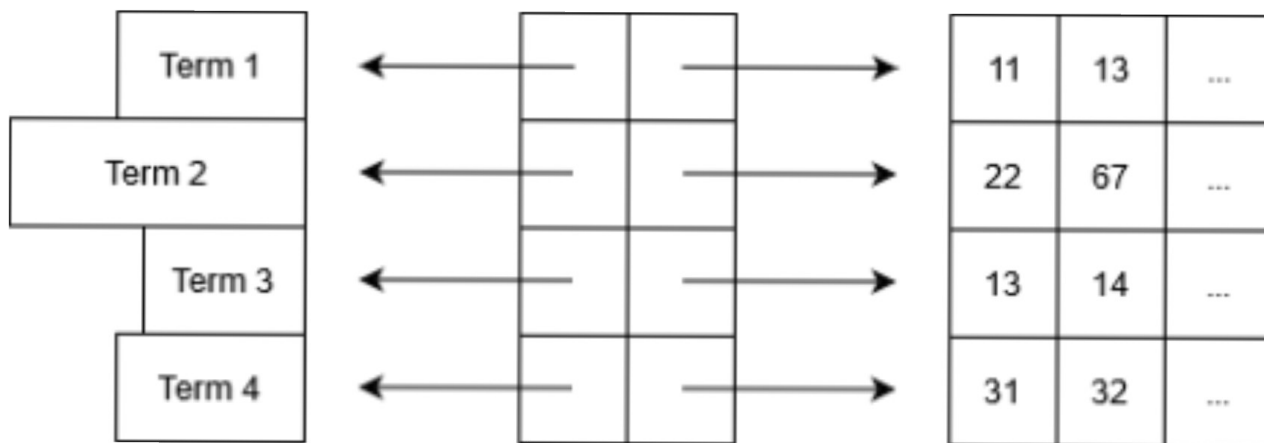


# Словарь на основе отсортированного списка

- Динамический список с амортизированной стоимостью вставки  $O(1)$
- Сортируем 1 раз, в конце процесса индексации перед началом обработки запросов
- Ищем термины бинарным поиском за  $O(\log(N))$
- Главная проблема: все термины разной длины!
- Мы не можем хранить такой список в непрерывном блоке памяти (массиве в языке C)

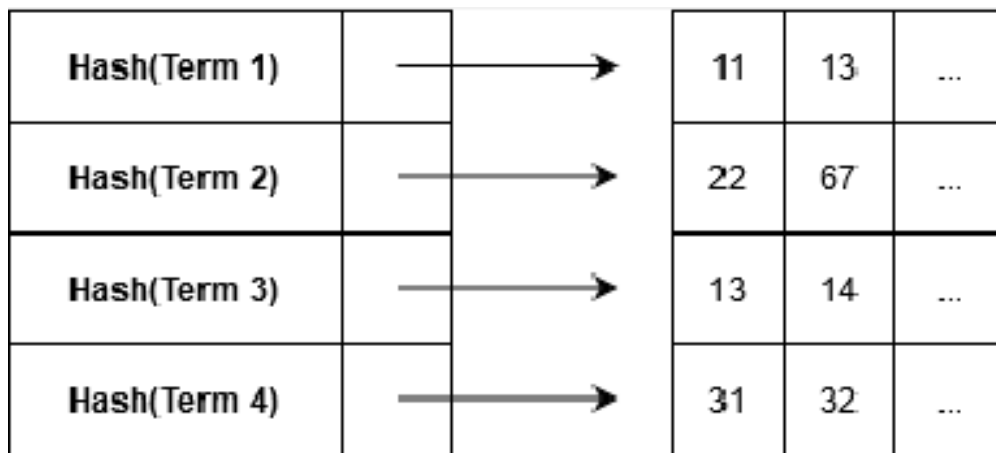
# Динамический список в памяти (на примере list из Python)

- Нам понадобится динамический список наподобие list из Python
- Массив из указателей на текст термина и список словопозиций
- «Рыхлый», неоптимальный по кэшу и памяти



# Динамический массив в памяти

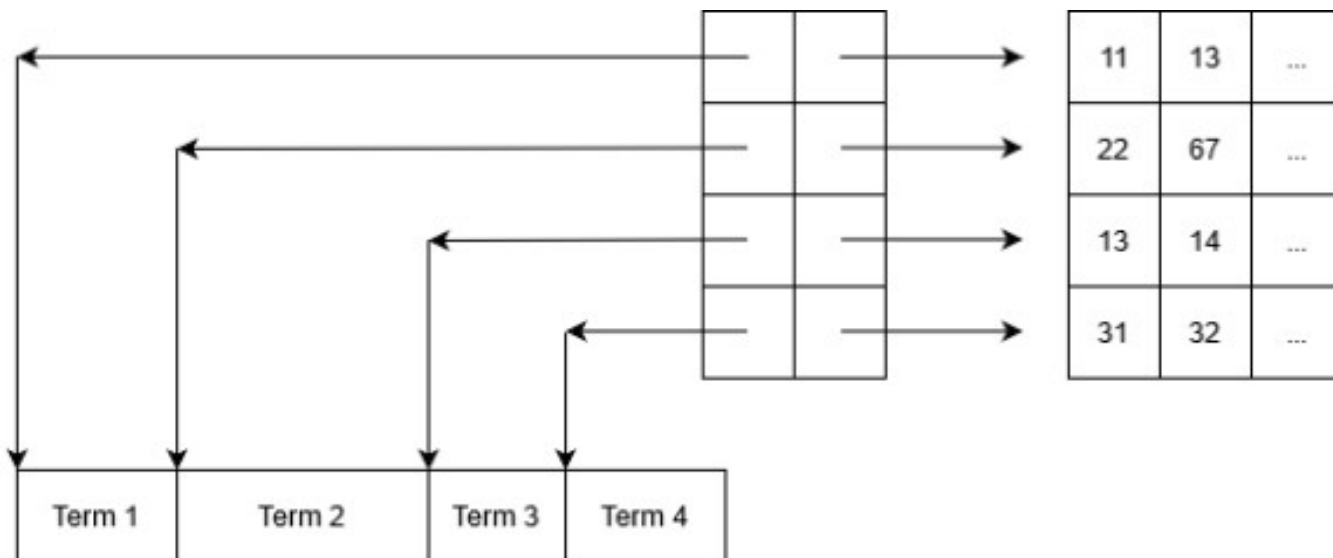
- Основная идея: надо хранить термины в непрерывном блоке памяти
- Не list из Python, а массив из C/C++!
- Можем пренебречь коллизиями и хранить хэши терминов!



- Возможно ли обойтись без коллизий? (ответ на след. слайде)

# Словарь как строка

- Можно хранить тексты терминов подряд в одном длинном блоке памяти
- В другом блоке храним указатели на начало терминов и списки словопозиций
- Отличное решение для больших индексов

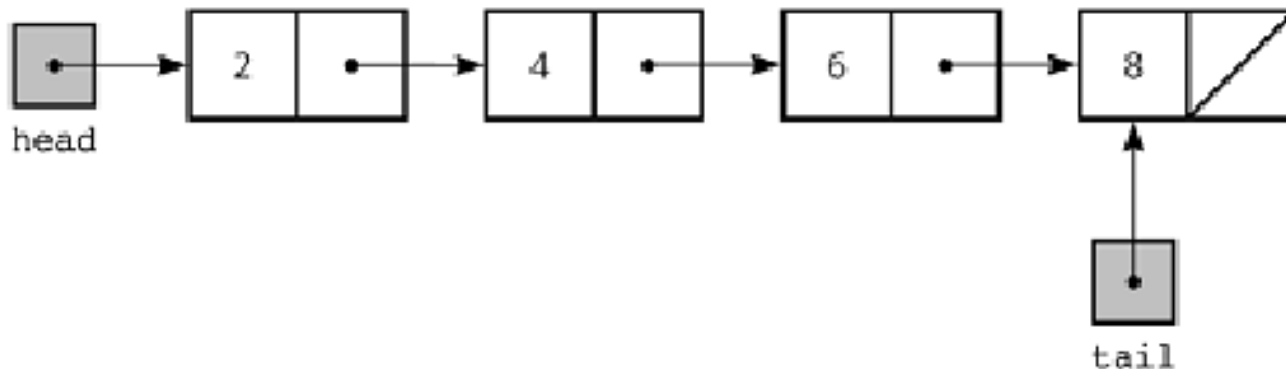


# Какую структуру данных выбрать для списка словопозиций?

- Возможны несколько вариантов
  - Связанный список
  - Хэш-таблица
  - Отсортированный массив
- Выбранная структура данных должна обеспечивать эффективное пересечение/слияние списков для многословных запросов

# Списки словопозиций на основе СВЯЗАННЫХ СПИСКОВ

- Простейшая структура данных
- Теряем много памяти под указатели
- Плохо используем кэш ЦПУ
- Почти всегда плохое решение на современном железе



# Списки словопозиций на основе хэш таблиц

- Решение из нашего примера на Python
- Быстрые вставка и поиск за  $O(1)$
- Быстрое пересечение за  $O(N)$ : бежим по первой хэш-таблице и ищем элементы во второй
- Не совсем оптимально по памяти
- Отличное решение для небольших коллекций
- Не самое лучшее решение для больших!

# Списки словопозиций на основе отсортированного массива

- Динамический массив из DocID: list в Python или std::vector в C++
- Быстрая вставка в конец с амортизированной стоимостью  $O(1)$
- Перед использованием сортируем за  $O(N \cdot \log(N))$
- Сортированные массивы можно эффективно пересекать за  $O(N)$  времени и  $O(1)$  памяти
  - Подробнее позже
- Сортировку можно делать на этапе построения индекса!
- Лучшее решение для больших коллекций

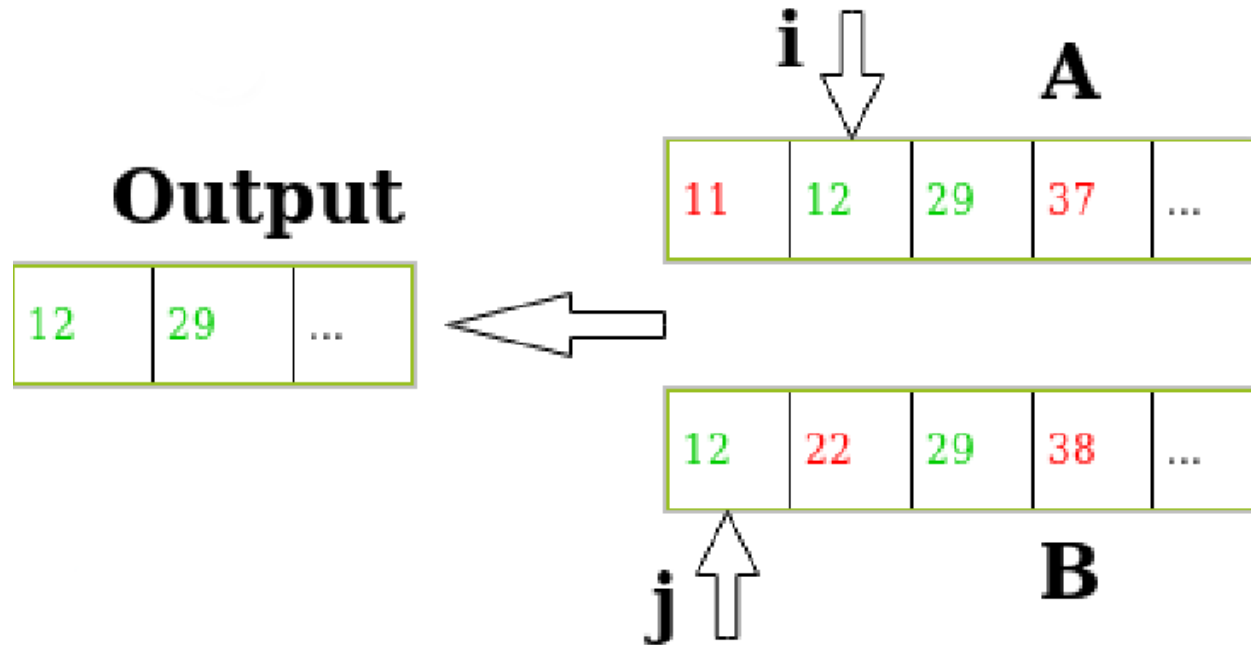


# Пересечение списков словопозиций

- Для обработки булевых запросов надо пересекать или сливать связанные с терминами списки словопозиций
- Простейший случай: 2-словные запросы
  - И-запросы: пересечение 2-х сортированных списков
  - ИЛИ-запросы: слияние 2-х сортированных списков
- Эффективный алгоритм:  $O(N_1 + N_2)$  по времени и  $O(1)$  по памяти

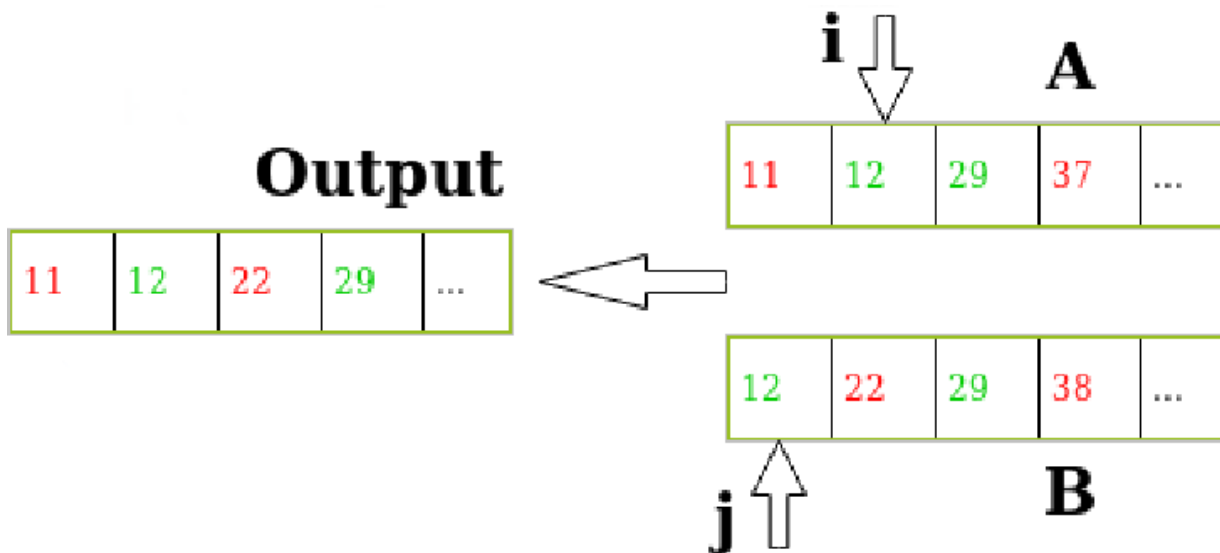
# Пересечение 2-х отсортированных СПИСКОВ

- И-запросы (конъюнкция)
- 2 индекса:  $i$  и  $j$
- $A[i] < B[j] \rightarrow i++$
- $A[i] > B[j] \rightarrow j++$
- $A[i] == B[j] \rightarrow$ 
  - `result.append(A[i])`
  - $i++, j++$



# Слияние 2-х отсортированных списков

- ИЛИ-запросы (дизъюнкция)
- 2 индекса:  $i$  и  $j$
- $A[i] < B[j] \rightarrow$ 
  - `result.append(A[i])`
  - $i++$
- $A[i] > B[j] \rightarrow$ 
  - `result.append(B[j])`
  - $j++$
- $A[i] == B[j] \rightarrow$ 
  - `result.append(B[i])`
  - $i++, j++$

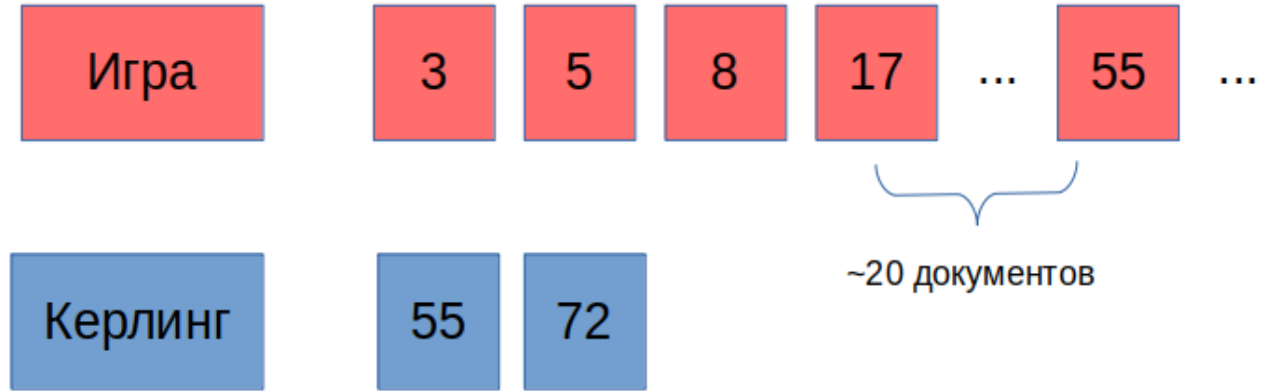


# Эффективное пересечение списков словопозиций

- Рассмотрим запрос  
*игра керлинг*

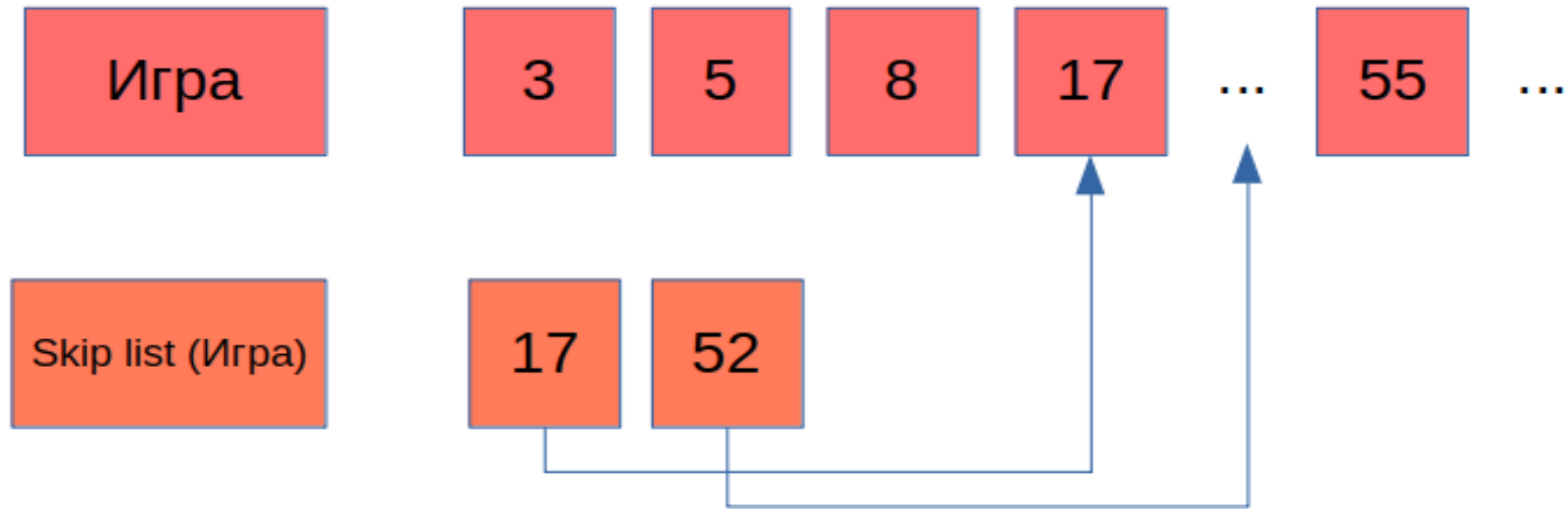
- $|P(\text{Игра})| = 1000 \times |P(\text{Керлинг})|$

- Идеи?



- Поиск произвольными перемещениями не эффективен для современных CPU

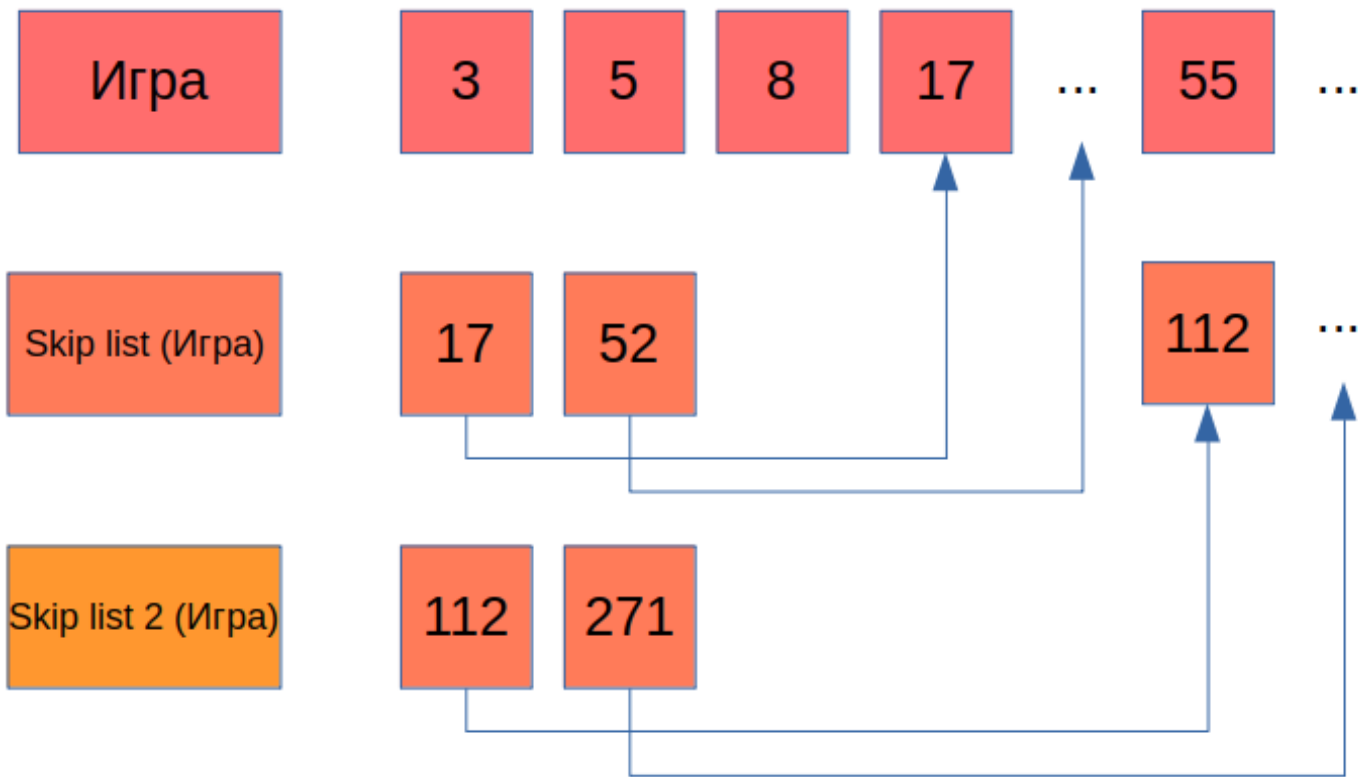
# Эффективное пересечение списков. Skip-lists



- Что если и этого мало?

# Эффективное пересечение списков. More skip-lists

- Можем точно знать, куда именно прыгать для сжатых списков
- *Каждый уровень skip-листа это массив, а значит cache friendly*

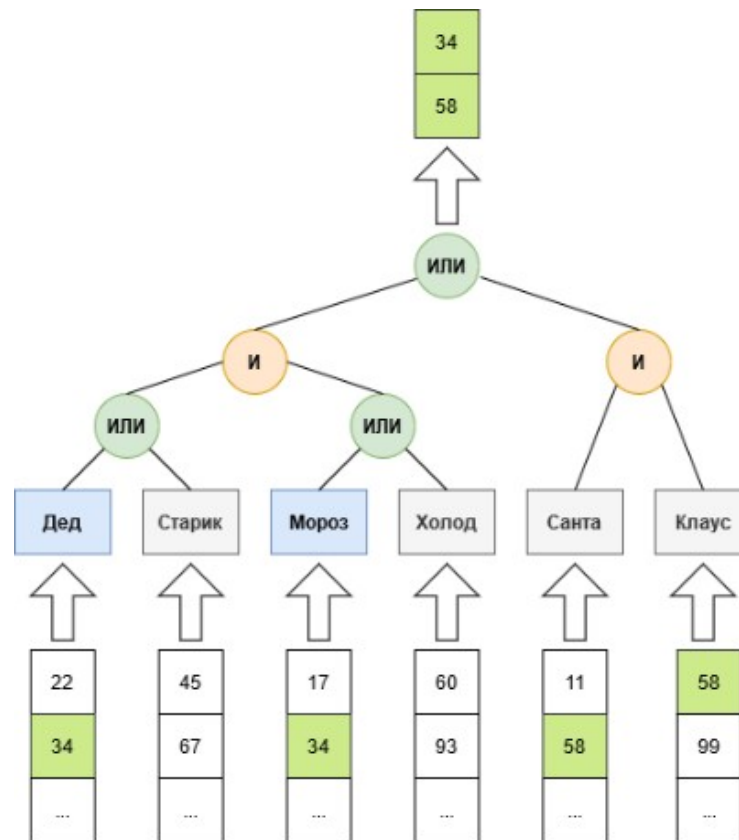


# Стоп-слова

- Проблема: есть слова, которые встречаются почти в каждом документе
- Очень длинные списки словопозиций → очень медленное пересечение
- Возможное решение: списки стоп-слов
- Составляем список самых частотных слов: и, в, на, с, ...
- Правило 30: топ-30 самых частотных терминов порождают 30% лексем
- Просто удаляем такие слова из индекса
- Качество поиска практически не снижается

# Обработка сложного дерева запроса

- С каждым термином-листом связан свой список документов
- Документы «текут» вверх по дереву, в каждом узле происходит пересечение или слияние дочерних списков





# Фразовые запросы

- Иногда И- и ИЛИ-запросов недостаточно
- Пользователи ожидают от современных поисковых систем возможность подавать фразовые запросы, напр. «*московский университет*»
- Слова в кавычках должны содержаться в документе как упорядоченная последовательность
- Документ *московский государственный университет* по такому запросу находиться не должен!

# Запросы с ограничениями на близость слов

- Иногда требуется найти только те документы, в которых слова запроса находятся рядом
- «Рядом» можно определить строго: запросы удовлетворяет только те документы, в которых  $distance(term\_i, term\_j) \leq K$  для любых  $i$  и  $j$
- где  $distance(term\_i, term\_j)$  можно понимать как минимальное число слов в документе между терминами  $i$  и  $j$

# Расширенная булева модель

- Для обработки фразовых запросов и запросов с ограничениями на близость слов необходимо учитывать позиции слов в документе
- Такая модель иногда называется **расширенной булевой моделью**
- Будем использовать **координатный обратный индекс** — это обратный индекс, дополненный информацией о позициях (координатах) слов в документах

# Координатный обратный индекс: пример

- Словопозиции теперь содержат списки позиций термина в документе

- государственный → [2:[2]]
- институт → [1:[4]]
- ИТМО → [3:[2]]
- московский → [1:[1], 2:[1]]
- технический → [1:[3]]
- физика → [1:[2]]
- университет → [2:[3], 3:[1]]

Документы:

1. Московский  
физико-  
технический институт
2. Московский  
государственный  
университет
3. Университет ИТМО

# Координатный обратный индекс гораздо больше обычного

- Позиция каждого слова из документа должна быть упомянута в списке позиций для соответствующего термина
- Размер индекса теперь пропорционален полному числу терминов в коллекции (а не числу уникальных терминов, как раньше)
- Размер такого индекса сравним в размером самой коллекции!
- И на порядок больше размера обычного индекса
- Памяти нужно еще больше
- VK хранит списки DocID в памяти, а списки позиций на SSD

# Координатный обратный индекс необходим для ранжирования

- Модели поиска с ранжированием должны учитывать, что:
- Документ, в котором слова запроса найдены в начале, как правило более релевантен запросу, чем документ со словами в середине или конце
- Документ, в котором слова запроса идут подряд (или почти) и в таком же порядке (или почти) как правило более релевантен, чем документ, в котором эти же слова находятся далеко друг от друга

# Обработка запросов с ограничениями на близость и порядок слов (в т.ч. фраз)

- Ищем и пересекаем/сливаем списки словопозиций так же, как и раньше
- Для каждого документа, который нашелся бы обычному булевому запросу:
  - Получаем списки позиций
  - Проверяем, насколько он удовлетворяет ограничениям на порядок и близость слов
  - Если да, то добавляем в список результатов поиска

# Подведем промежуточные итоги

- Мы реализовали булеву модель поиска с помощью обратного индекса, и рассмотрели некоторые ее расширения
- Знаем, как строить обратный индекс, когда он еще помещается в память
- Все еще не знаем как строить большие обратные индексы, которые не помещаются в память



# Построение больших индексов

# Построение больших индексов

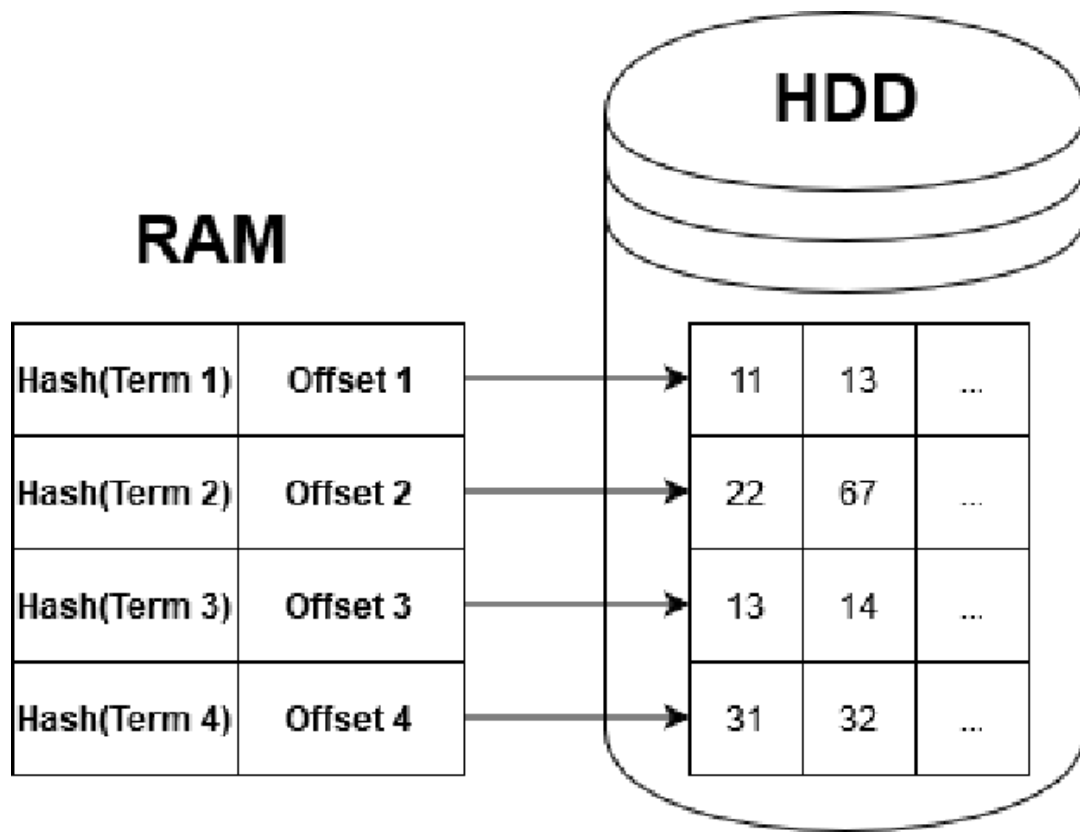
- Наши примеры обработки запроса и построения индекса на Python работают только в тех случаях, когда индекс помещается в память
- На практике это, как правило, не так
  - обратный индекс по Рунету занимает ~50 Тб
- Перейдем к рассмотрению алгоритмов построения таких индексов

# Данные бывают большие и очень большие

- Будем различать 2 основных случая
  - Индекс не помещается в память, но все еще помещается на жесткий диск на одной машине
    - Индекс размером до нескольких терабайт
    - Обрабатываем запросы без загрузки индекса в память
    - Применяем алгоритмы индексации во внешней памяти
  - Очень большой индекс
    - Индекс размером десятки терабайт и больше
    - Обрабатываем запросы на поисковом кластере
    - Применяем распределенные алгоритмы индексации на кластере

# Индекс со списком словопозиций на диске

- Словарь с оффсетами в памяти, списки словопозиций в файле

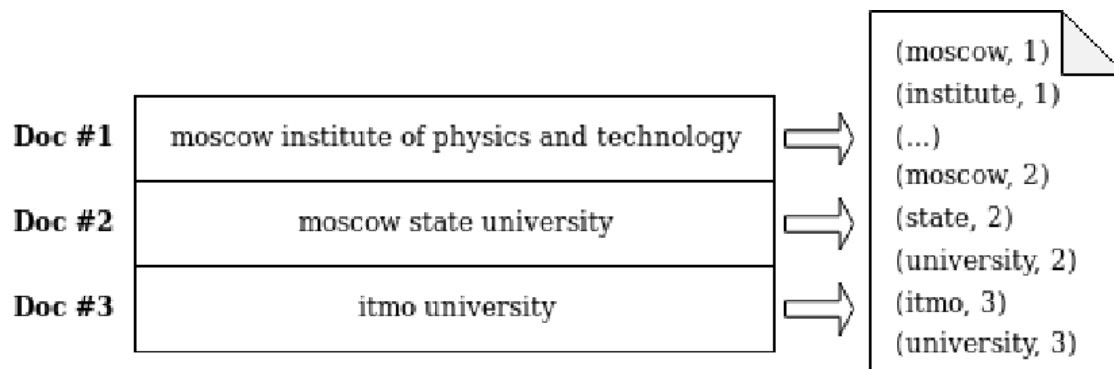


# Индексирование, основанное на сортировке

- Индекс все еще помещается на жестком диске
- Решение основано на применении операций сортировки и слияния с использованием внешней памяти
- Состоит из 3-х этапов (детали на след. слайдах):
  - MAP: документ → список пар (термин, DocID)
  - SORT: сортируем пары на диске, в качестве ключа используем термин
  - REDUCE: группируем DocID'ы одного и того же термина в списки словопозиций

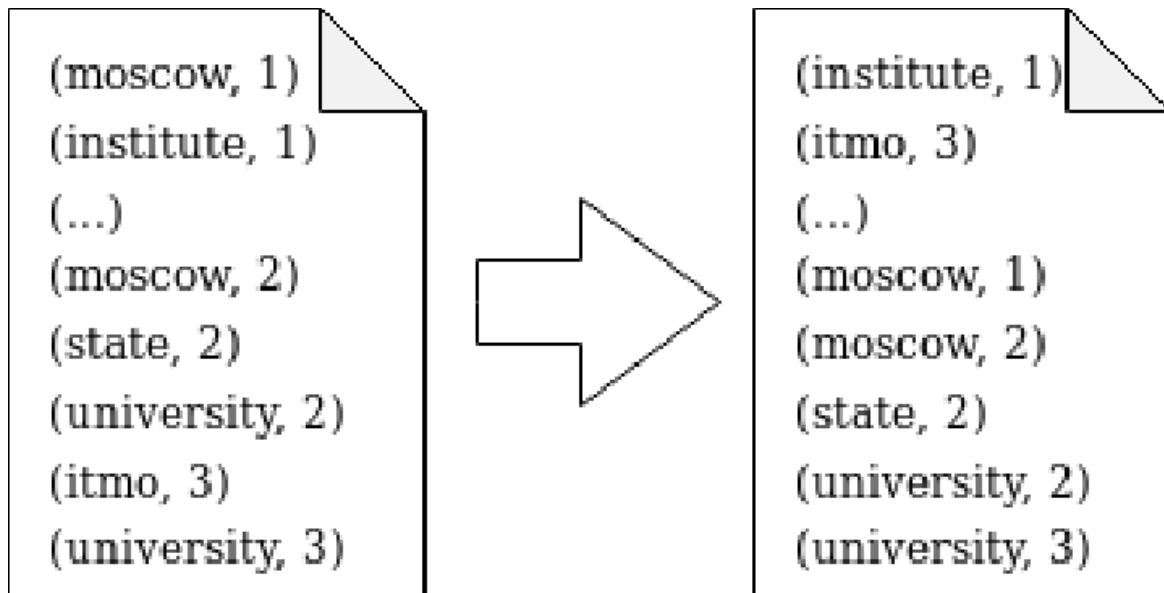
# Этап #1: MAP

- Для каждого документа из коллекции
  - Загружаем документ в память
  - Токенизируем и нормализуем
  - Для каждого термина порождаем пару (термин, DocID)
  - Сохраняем все пары на жесткий диск в один большой файл



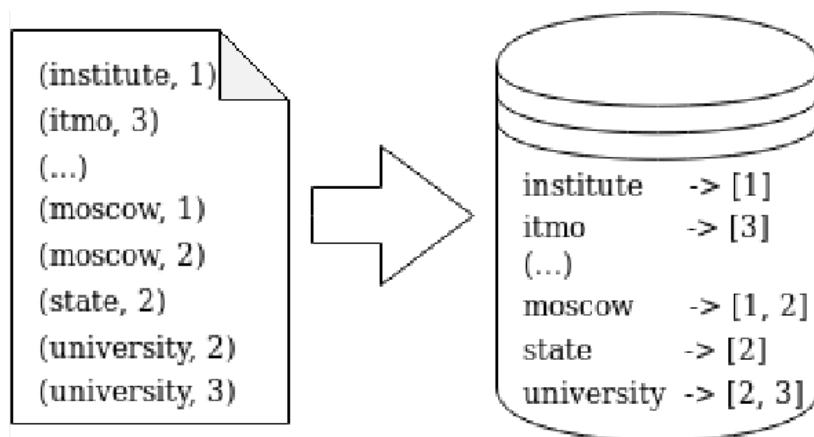
## Этап #2: SORT

- Сортируем большой файл с парами на диске
- Как? Размер файла > размера доступной памяти! Есть ли идеи?
- Вернемся к этому вопросу чуть позже



# Этап #3: REDUCE

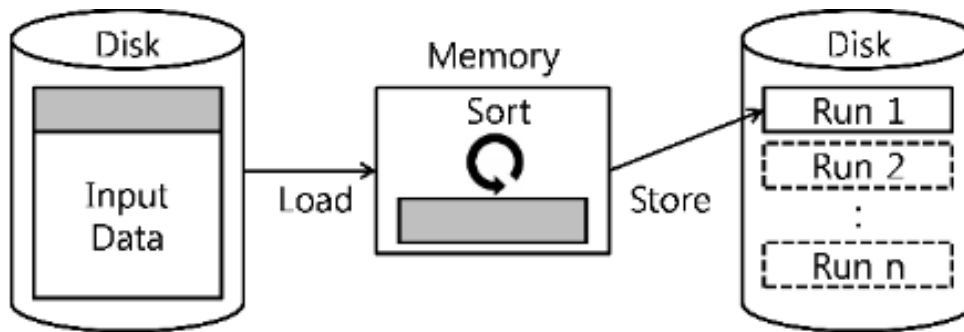
- Для каждого термина
  - Читаем из файла все связанные с ним пары
  - Формируем список словопозиций
  - Сохраняем термин и список словопозиций обратно на диск в файл обратного индекса
- Эффективно по памяти:  $O(L)$  где  $L$  – длина списка словопозиций



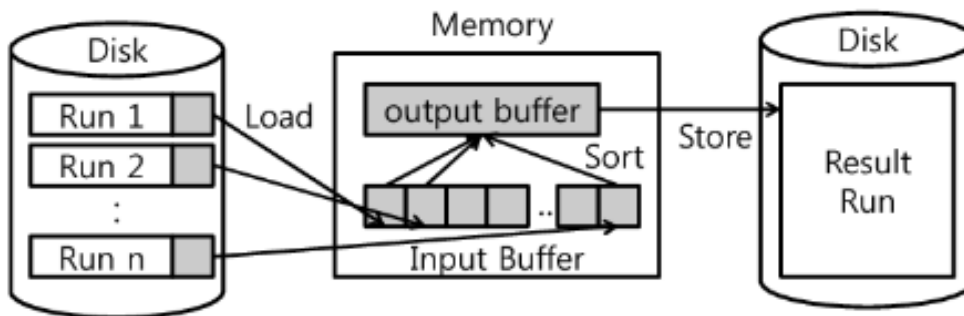


# Сортируем большой файл во внешней памяти

- Внешняя сортировка слиянием
- Разбиваем файл на блоки размером с доступную память
- Сортируем каждый блок в памяти
- Сливаем отсортированные блоки



(a) Run formation phase



(b) Merge phase

# Утилита sort

- На практике для сортировки больших файлов во внешней памяти удобно использовать утилиту sort
- Файл \$FILE с парами (term, count)
- Сортируем по термину: *sort -k1,1 \$FILE*
- При желании, всю индексацию можно реализовать используя только утилиты UNIX: sort, uniq, awk

# Обработка запросов на кластере

- Самый тяжелый случай: индекс не помещается на одной машине
- Решение: **шардирование**
- Разбиваем большой индекс на блоки-шарды
- Каждый шард храним на отдельной машине в кластере
- Возможны 2 способа разбиения: по терминам и по документам

# “Нарезка” индекса по терминам

- В каждый шард попадает только часть обратного индекса для какого-то диапазона терминов
- Термины [А-К]
  - абажур → [771, ...]
  - аббат → [608, ...]
  - абзац → [133, ...]
  - ...
- Термины [Л-Я]
  - лабаз → [905, ...]
  - лабиринт → [108, ...]
  - лаборант → [297, ...]
  - ...

# “Нарезка” индекса по документам

- Более сбалансированная схема
- В VK индекс нарезается так:  $\text{shard} = \text{hash}(\text{DocURL}) \% (\text{общее кол-во шардов})$
- Современные поисковики используют поддокументное шардирование

## Шард №1

- абажур → [771, ...]
- аббат → [608, ...]
- абзац → [133, ...]
- ...

## Шард №2

- абажур → [1502, ...]
- аббат → [2913, ...]
- абзац → [96, ...]
- ...

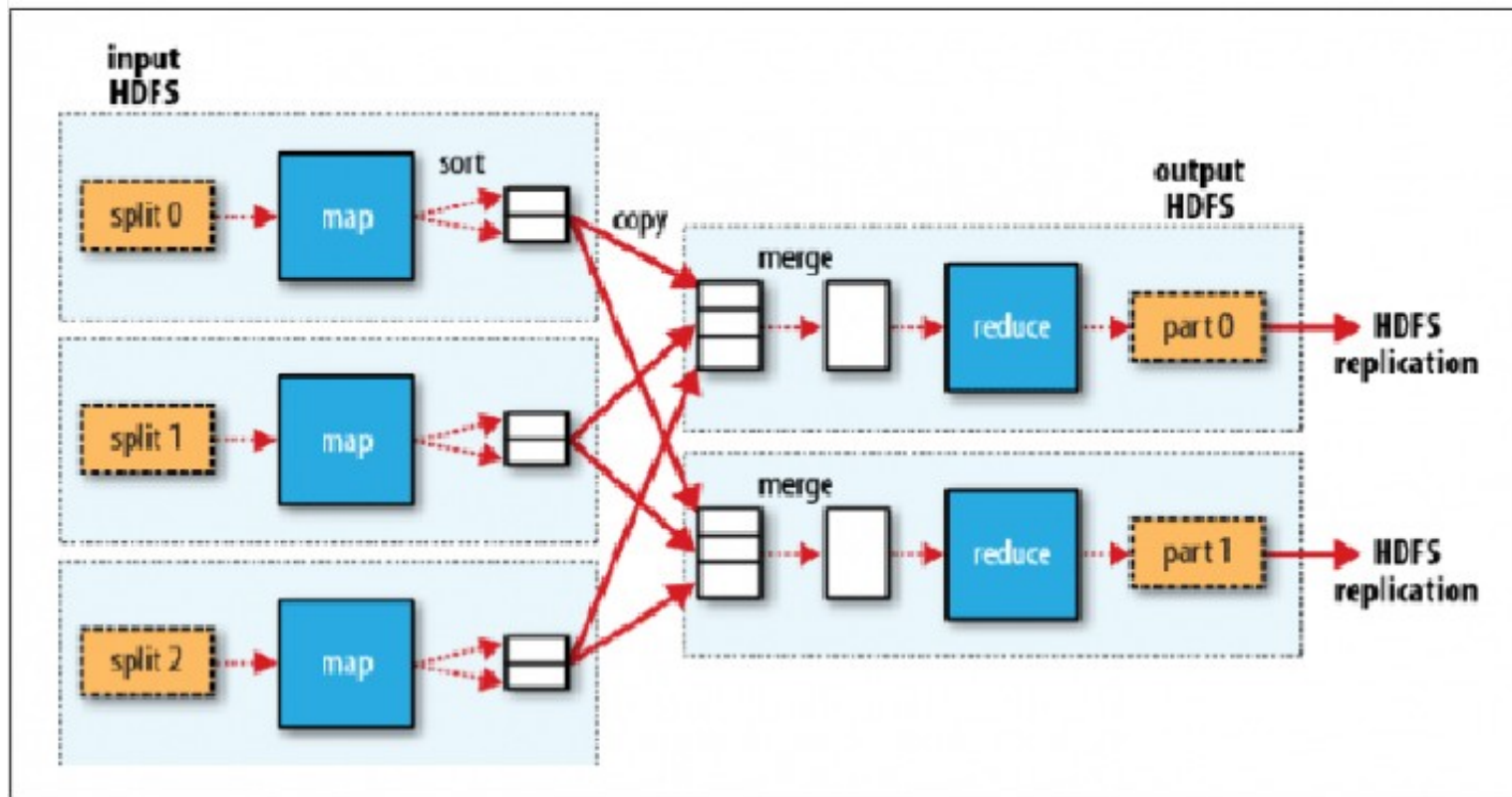
## Шард №3

- абажур → [771, ...]
- аббат → [608, ...]
- Абзац → [133, ...]
- ...

# Распределенная индексация

- Самый тяжелый случай: индекс не помещается даже на диске
- Индексирование, основанное на сортировке – это один из вариантов парадигмы MapReduce
- Нам понадобится кластер с установленной реализацией MapReduce, напр. Hadoop
- Алгоритм остается прежним, только:
  - Этапы MAP и REDUCE выполняются параллельно в узлах кластера
  - Этап SORT скрыт от пользователя «под капотом» платформы

# Общая схема работы кластера Hadoop



# Реальный коллекции документов ПОСТОЯННО МЕНЯЮТСЯ

- Например, веб-страницы постоянно создаются, изменяются и удаляются
- Пользователи ожидают от поисковой системы
  - свежесть – по запросу *выборы* надо находить сегодняшние документы
  - актуальность – по запросу *спартак зенит* надо в первую очередь находить документы про самый последний матч

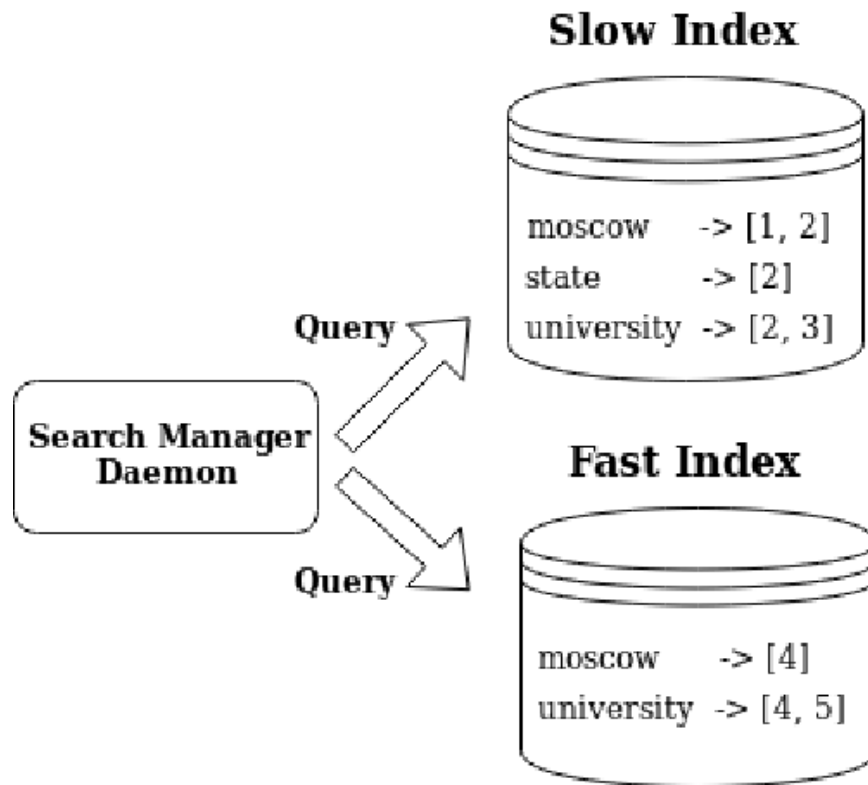


# Нужен механизм эффективного обновления индекса

- Мы должны уметь:
  - Добавлять в индекс новые документы
  - Заменять старые версии документов на новые
  - Удалять «мертвые» документы (Ошибка 404, ...)
- Простейшее решение: просто перестроить индекс с нуля
  - В поиске VK большой индекс обновляется каждые 2 недели
- Будем терять самые свежие документы

# “Быстрые” индексы

- «Быстрый» индекс
- Содержит только свежие документы
- Помещается в памяти → можно обновлять «на лету»!
- Поисковый запрос идет сразу на оба индекса
- Документ с одним и тем же URL найден в обоих? Берем тот, который из «быстрого»



The background is a solid purple color. It features several thin, light green lines: a vertical line running down the center, and two large, sweeping curves that start from the top left and bottom left, arching towards the right side of the frame.

Сжатие индекса

# Зачем сжимать?

- Экономим место
  - Особенно если RAM
- Больше помещается в память
  - Быстрее передача данных
  - {Прочитать сжатое, распаковать} может быть быстрее чем {прочитать несжатое}
  - Больше можно закешировать
- И мы говорим здесь не про gzip/rar/etc (десятки мб/сек), а действительно быстрые алгоритмы сжатия/распаковки, например LZO/LZ4 (гб/сек)

# Сжатие с потерями

- Пример не из поиска: Jpeg
- Понижение капитализации, стоп-слова, морф. нормализация – может рассматриваться как сжатие с потерями
- Удаление координат для позиций, которые вряд ли будут вверху на ранжировании

# Какие части индекса можно сжимать?

- Полезно сжимать
  - Словарь
  - Списки словопозиций
  - Списки вхождений в словопозициях
- Мы будем рассматривать только списки словопозиций в не-координатном индексе, т.е. отсортированные списки DocID
  - москва → [..., 283047, 283154, ...]
  - ...

# Как сжимать списки DocID?

- Ключевая идея: храним промежутки (gaps) между последовательными DocID вместо самих DocID!
  - москва → [..., 283047, 283154, 283159, 283202, ...]
  - москва → [..., ..., 107, 5, 43, ...]
- Для частотных слов промежутки меньше
- Как следствие, промежутки распределены неравномерно: гораздо больше маленьких, чем больших
  - Маленькие промежутки можно кодировать меньшим числом байт (т.н. **кодирование с переменной длиной**) Какие есть идеи?

# Кодирование с переменной длиной (variable byte encoding)

- Очень похоже на UTF-8
- 1-й бит каждого байта служит маркером конца кода
  - 1 – код заканчивается этим байтом, 0 – продолжается
- Последние 7-бит – полезная нагрузка, которая хранит части промежутка

DocIDs	824	829	215406
Gaps		5	214577
VarByte code	00000110 10111000	10000101	00001101 00001100 10110001



# Свойства кодирования с переменной длиной

- Очень просто в реализации
- Уменьшает размер обратного индекса на 50%!
- Можем кодировать блоками больше или меньше байта (напр. 32 бита или 4 бита)
- Меньше размер блока → более экономно расходуется память
- На практике 1-байтовые коды почти оптимальны

# Унарное кодирование

- Работает на битовом уровне
- Очень неэффективно само по себе, но используется в эффективном гамма-коде Элиаса
- Унарный код числа N: 111...(повторяется N раз)0
- Например: 5 → 111110

# Гамма-код Элиаса

- Работает на битовом уровне
- Представляем промежуток в виде **длины** и **смещения**
- Для промежутка  $G$ :
  - Смещение  $\text{Offset}(G)$  – это  $G$  в двоичном коде без первой 1
  - Длина  $\text{Length}(G)$  – это длина смещения в унарном коде
  - Гамма-код  $\text{Gamma}(G)$  – конкатенация длины и смещения

# Гамма-код Элиаса: пример

- Посчитаем гамма-код для  $G=13$ 
  - $\text{Offset}(13) = \text{Offset}(1101b) = 101$
  - $\text{Length}(13) = 3 \text{ бита} = 1110$
  - $\text{Gamma}(13) = 1110101$  (конкатенация длины и смещения)

# Гамма-код Элиаса: свойства

- Можно показать, что гамма-код Элиаса почти оптимален
- Уменьшение размера индекса может достигать 75%!
- Для произвольной последовательности гамма-кодов всегда существует уникальный результат декодирования
  - При хранении и передаче кодов между ними не нужны разделители (если можем прочесть всю последовательность от начала)
- Универсальные: эффективны для произвольного распределения промежутков
- Нет каких-то дополнительных параметров, которые надо было бы настраивать под распределение

# Оценка качества булевой модели

# Оценка качества булевой модели

- В булевой модели документ или удовлетворяет запросу, или нет
- По сути, это классификатор
- Каждому документу можно приписать метку класса:
  - 1 – релевантный
  - 0 – нерелевантный
- Большой дисбаланс классов: релевантных документов гораздо меньше чем нерелевантных

# Матрица ошибок

- Результаты классификации удобно представить в виде матрицы ошибок
- $y$  – реальное соответствие документа запросу
- $h(x)$  – результат работы классификатора

	$y = 1$	$y = 0$
$h(x) = 1$	True Positive (TP)	False Positive (FP)
$h(x) = 0$	False Negative (FN)	True Negative (TN)

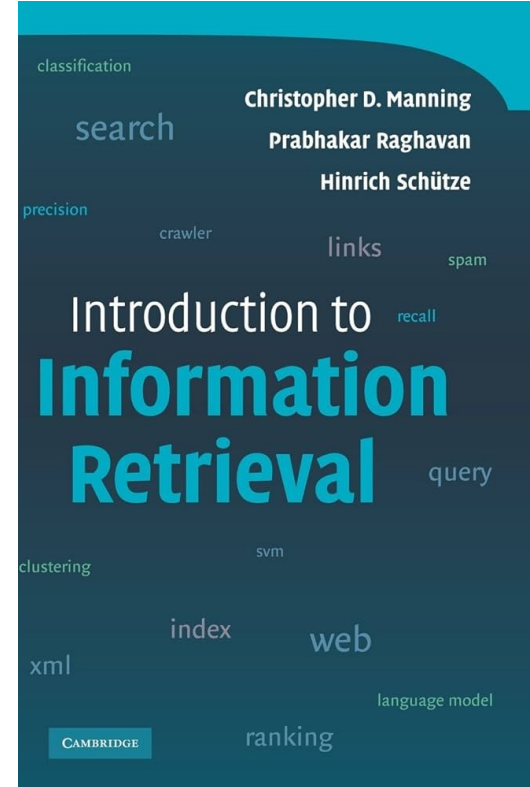


# Точность и полнота

- Посчитаем TP, FP и FN
- Точность:  $precision = \frac{TP}{TP+FP}$
- Полнота:  $recall = \frac{TP}{TP+FN}$
- $F_\beta$ -мера выражает баланс между точностью и полнотой
- $F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall}$
- В домашнем задании мы будем использовать  $F_1$ -меру (т.е.  $\beta = 1$ )

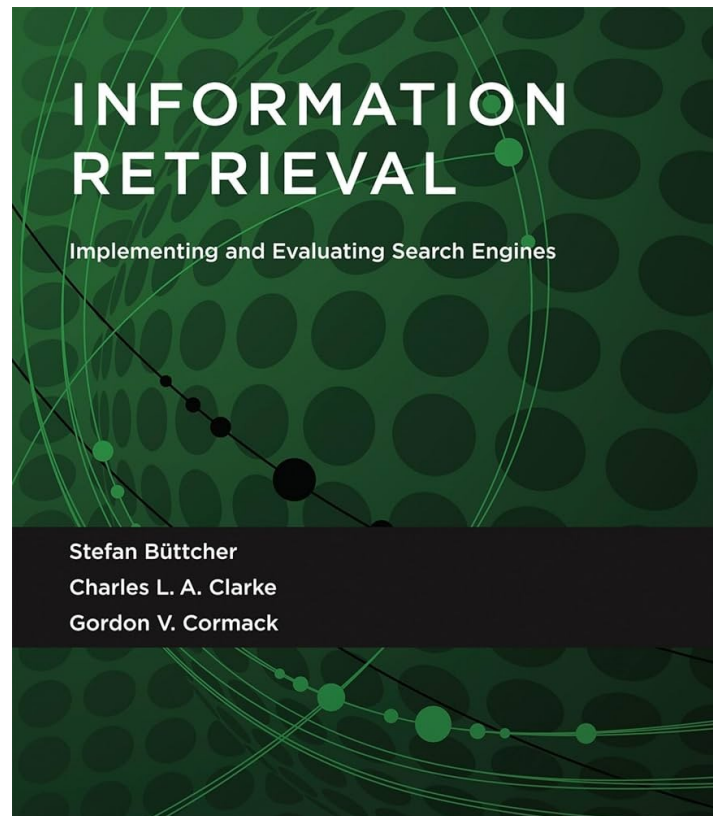
# Что можно почитать

- **Introduction to Information Retrieval**  
Christopher D. Manning et al. (2008)
- Есть на русском!
- Английская версия доступна бесплатно
- Немного устарела
- Не очень строгое изложение



# Лучший продвинутый учебник

- Лучший продвинутый учебник
- **Information Retrieval: Implementing and Evaluating Search Engines** Stefan Büttcher et al. (2016)
- Хорошо дополняет Маннинга



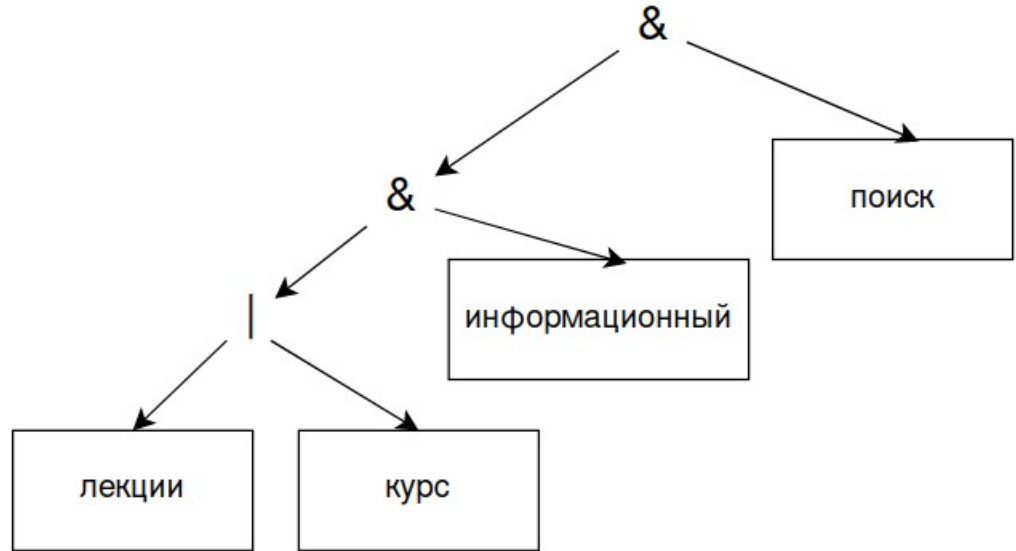
# Домашнее задание

# Домашнее задание

- Дана тестовая коллекция из небольшого количества документов и запросов
- Тексты запросов и документов уже лемматизированы
- Необходимо реализовать булеву модель и для каждого запроса найти множество удовлетворяющих ему документов
- Для сдачи задания  $F_1$ -мера результата должна попадать в заданный допустимый интервал

# Дерево запроса

- Вспоминаем парсер арифметических выражений.  
Реализовывали такой?
- Например: *(лекции | курс) информационный поиск*



# Дерево запроса: токенизация

- Все токены `r'\w+|[\(\)& \!]`
- Разбиваем на 3 класса:
  - Скобки
  - Операторы
  - Термы

# Дерево запроса: алгоритм разбора

- Находим оператор с минимальным приоритетом
  - Наиболее внешний, наиболее правый
  - Запомним как token
- Если не нашли, значит результат = термин или None
- Иначе
  - token.left = рекурсивно слева
  - token.right = рекурсивно справа
  - результат = token



# Исполнение дерева

- Вспоминаем исполнение дерева
- С каждым термином-листом связан свой список документов
- Документы «текут» вверх по дереву, в каждом узле происходит пересечение или слияние дочерних списков

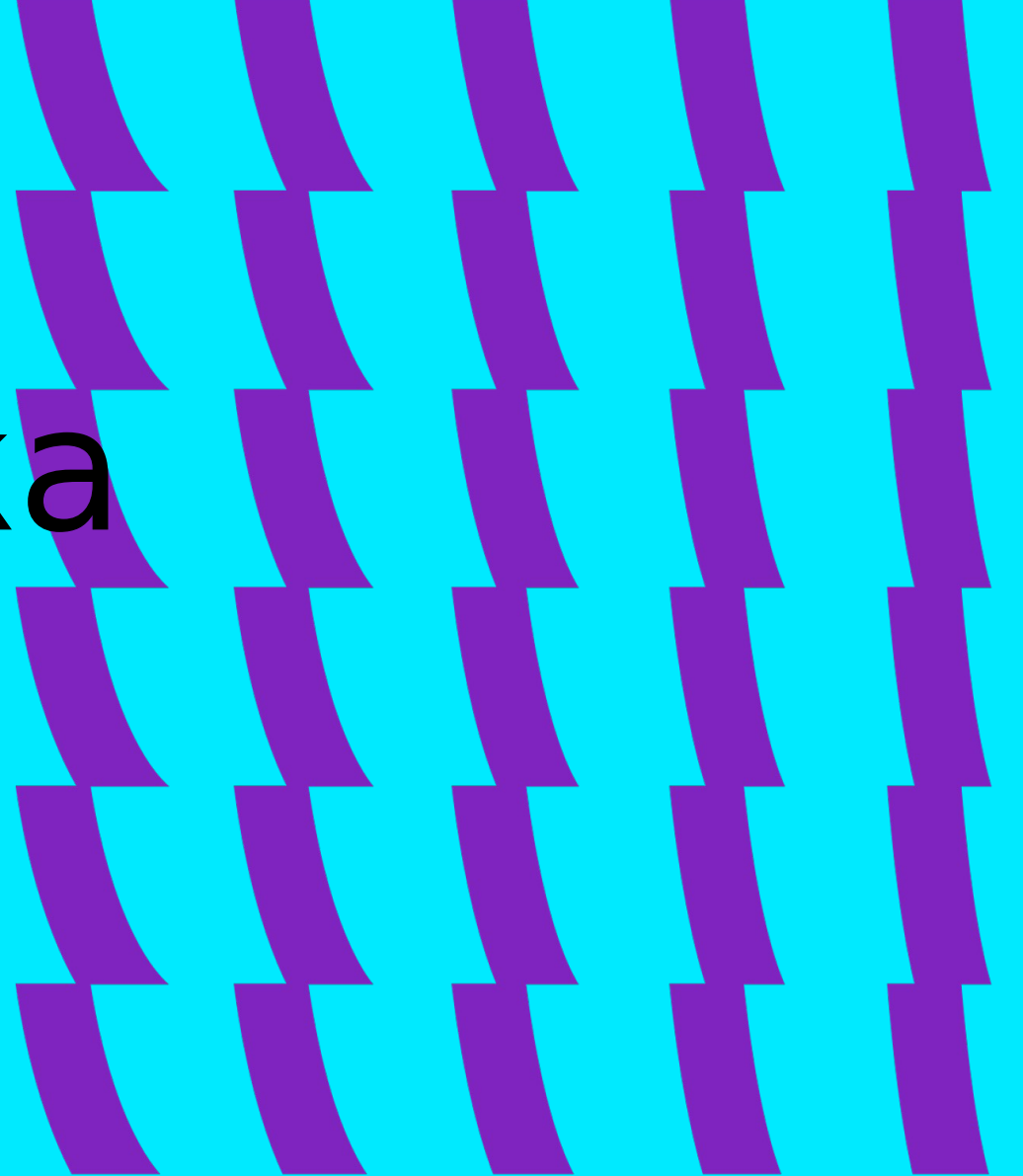
# Ссылки на домашнее задание

- Постройте обратный индекс для набора документов, сделайте поисковый движок и посмотрите насколько успешно находятся документы для разных запросов.
- Соревнование Kaggle <https://www.kaggle.com/t/5b8e2f30d5c840a79a8b037fb3c7bddc>
- Забрать скрипт для заполнения кодом и отправить решение здесь <https://github.com/agcr/vk-ir-course-fall-2024/blob/main/homeworks/03-indexing/solution.py>
- Здесь подробно описаны требования к коду <https://github.com/agcr/vk-ir-course-fall-2024/tree/main/homeworks/03-indexing>
- Сохраните лучший результат соревнования до окончания проверки
- Дополнительные баллы за экономное использование памяти и др.

The background is a solid purple color. It features several thin, light green lines: a vertical line running through the center, and two large, sweeping curves that intersect the vertical line, creating a stylized, abstract shape resembling a cross or a four-leaf clover.

Вопросы?

# Семинар: Библиотека whoosh



# Всем спасибо!

- Лекция, семинар: Игорь Поляков
- Домашнее задание: Андрей Кривой, Тимур Мубаракшин
- Организация: Андрей Кривой, Федор Петрайкин