



# ТЕХНОСФЕРА

## Современные методы и средства построения систем информационного поиска

### ЛЕКЦИЯ 5: булев индекс и поиск (часть 2)

## Еще раз о сжатии

1. **Simple9**
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

## Дерево запроса

1. Исполнение дерева
2. Парсинг запроса

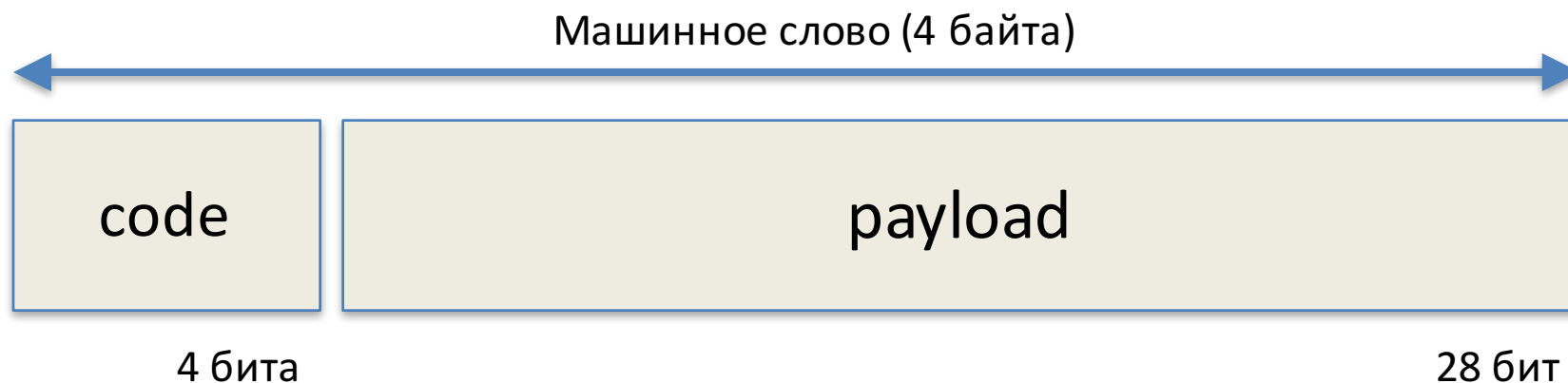
## Общий workflow



# Вспомним сжатие

- VarByte – быстрый, но избыточный
- Fibonacci – компактный, но медленно
- Gamma – избыточны + нечетны по битам

# Simple9



# Simple9: payload

- 1 28-и битное число
- 2 14-и битных числа
- 3 9-и битных числа (и теряем 1 бит)
- 4 7-и битных числа
- 5 5-и битных чисел (и теряем 3 бита)
- 7 4-х битных чисел
- 9 3-х битных чисел (и теряем 1 бит)
- 14 2-х битных чисел
- 28 1 битных чисел

# Simple9

## — Плюсы:

- Очень быстр при распаковке ( $2 \cdot 10^9$  чисел/сек)
- Компактен

## — Минус:

- Избыточен для одного числа

# Когда какое сжатие использовать?

- Сжатие списка документов
- Сжатие вхождений слов в документ

## Еще раз о сжатии

1. Simple9

- 2. Бинарные данные в Python**

## Словарь поиска

1. Представление стоп-слов

2. Хранение словаря

## Дерево запроса

1. Исполнение дерева

2. Парсинг запроса

## Общий workflow





# Бинарное представление данных против текста

- Если много данных, необходимо бинарно
- Меньше размер
- Существенно меньше время на операции
- Фиксированный размер/вычисляемые смещения

# Python и двоичные данные

## **import struct**

- `struct.pack(fmt, v1, v2, ...)`
  - `'h'`, `'i'`, `'q'`
- `struct.unpack(fmt, string)`
- `struct.unpack_from(fmt, buffer[, offset=0])`
- Аналог `sizeof()`: `struct.calcsize(fmt)`

# Python и двоичные данные

## **import array**

- `array.array(typecode)` # аналогично *struct.pack*
- Наполняем с `append()`
- Сохраняем с `tostring/tofile`
- Загружаем с `fromstring/fromfile`

# Хранение бинарных данных

- `array.array('c')` # медленно
- `cStringIO.StringIO()`
- `bytearray`

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

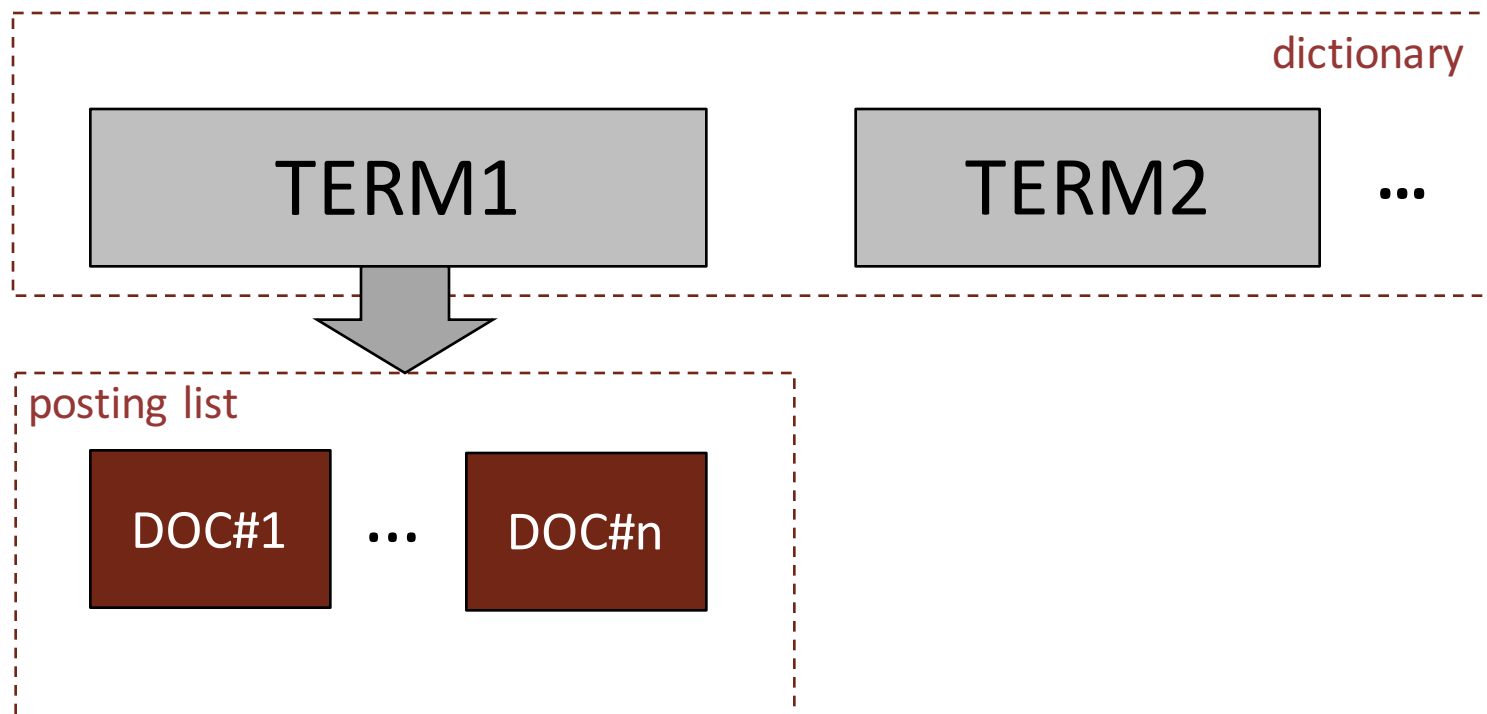
## Дерево запроса

1. Исполнение дерева
2. Парсинг запроса

## Общий workflow



# Словарь



# Как хранить

- Кол-во термов будет расти с ростом текста:
  - ошибочные написания
  - разные тематики: медицина, IT, ...
  - подходы к хранению стоп-слов
- Средняя длинна терма = 7.5 символов
- Но достигает и 20+ символов

# Как искать?

- **Как** искать терм в таком массиве? Т.е. какую структуру данных можно использовать?

Терм	частотность	Координаты блока
a	656 256	32
aachen	65	63216
...		
zulu	342	→



# Как искать?

- **Как** искать терм в таком массиве? Т.е. какую структуру данных можно использовать?

Терм	частотность	Координаты блока
a	656 256	32
aachen	65	63216
...		
zulu	342	→

Основные используемые структуры данных – хэши и деревья

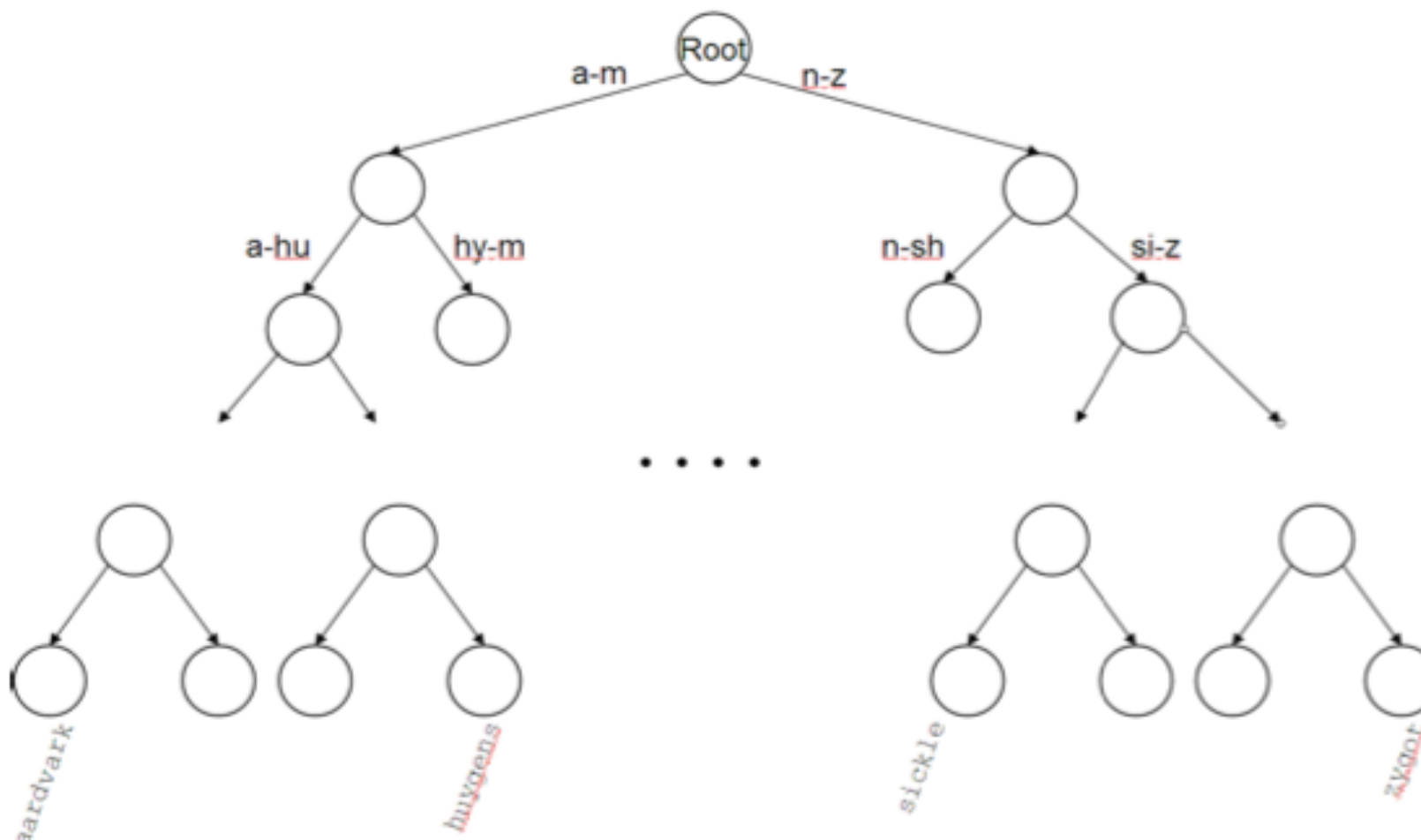
Выбор, в основном, зависит от назначения поиска:

- Надо искать кош\* {ка, ек, ам, ке, ку, ках} – деревья
- Надо искать одно слово - хэш

# Деревья

- Позволяют искать термы с общим префиксом
- Простейшее дерево – бинарное
- Поиск медленнее хэшей,  $O(\log M)$ , где  $M$  – размер словаря

# Бинарные деревья поиска



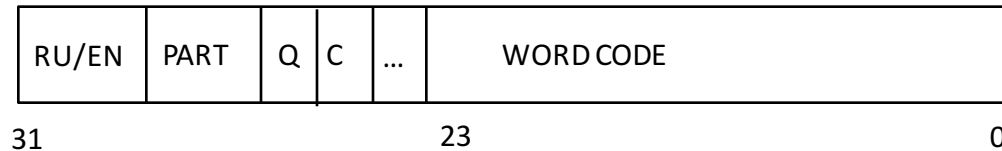
# Хэши

- Представление слова хэшируем в  $x$  постоянной длины
- Боремся с коллизиями
- Или берем  $x=64$  и хорошую хэш-функцию
- **Плюсы:**
  - Время поиска  $O(1)$
- **Минусы:**
  - Элементы хэша никак не связаны
  - Нельзя искать строку по префиксу
- Теоретически, можно сделать морфологическую хэш-функцию

# Морфологический хэш

- Rambler:

Кошку →



- go.mail.ru:

Кошку → “КОШКА” → mmhash64(...)

# Поиск с метасимволами

- Чтобы искать кошк\*
  - Используя В-дерево просто собираем все листья от «КОШК»
- Искать \*ошка
  - Создаем дерево записанное задом-наперед
  - Ищем аналогично первому случаю
- Результат – множество терминов подходящих под маску
- Теперь нужно найти документы содержащие эти термины

# Обработка «\*» внутри термина

- кош\*а
  - Можно поискать в В-дереве префикс и окончание
  - И пересечь 2 множества
  - Довольно расточительно
- Можем использовать k-граммы:
  - «Кошка» → “\$K” “КО” “ОШ” “ШК” “КА” “А\$”
  - Строим индекс по таким биграммам
  - Преобразуем запрос “\$K” AND “КО” AND “ОШ” AND “А\$”
- Можем обрабатывать сложные случаи:
  - к\*ш\*а

- Почему большие поисковики не используют поиск с метасимволами?





- Почему большие поисковики не используют поиск с метасимволами?
- 



- Неудобные запросы
- Очень много слов
- Как исправлять опечатки?

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

- 1. Представление стоп-слов**
2. Хранение словаря

## Дерево запроса

1. Исполнение дерева
2. Парсинг запроса

## Общий workflow



# Что такое стоп-слова

- Слова с наивысшей частотой
- Как правило, не имеют смысла без контекста
- Имеют большое значение для цитат, меньшее – для других запросов
- И, ИЛИ, О, ...
- Также специфические: “купить”, ...

# Проблемы хранения

- Присутствуют в каждом документе
  - Можем создавать инверсный индекс
    - Подходит для булевского поиска
  - Игнорировать
    - Все равно большинство запросов работают
    - Уменьшаем индекс
  - Можем учитывать контекст

# Стоп-слова с контекстом

... частично из-за наследования и полиморфизма ...

⇒ НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Представим контекст специальными биграммами:

- Term1: НАСЛЕДОВАНИЕ → hash(..)
- Term2: НИЕ\_И → hash(..)
- Term3: И\_ПОЛ → hash(..)
- Term4: ПОЛИМОРФИЗМ → hash(..)

# Стоп-слова с контекстом

- Раздуваем словарь т.к. комбинации слов
- Обеспечивают небольшой список вхождений
- Позволяют искать цитаты

# Итог по словарю

- Словарь – адресует блоки индекса
- Обычно строится используя хэши
- Стоп-слова: важны особенно при цитатном поиске
- Размер словаря имеет значение
- Как правило, хранится в памяти (Random access)

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
- 2. Хранение словаря**

## Дерево запроса

1. Исполнение дерева
2. Парсинг запроса

## Общий workflow



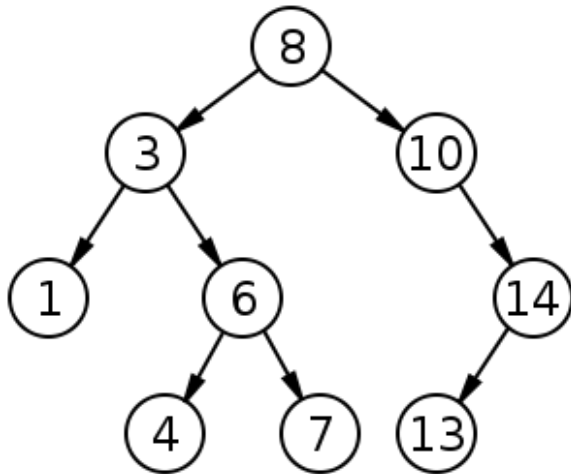


# Python: словари

- Используем обычный `dict()`
- Сериализация?
- **`import pickle`**
  - `pickle.dump(obj, file[, protocol])`
  - `load`

Не компактен и требует полной загрузки

# Сериализация сложных данных



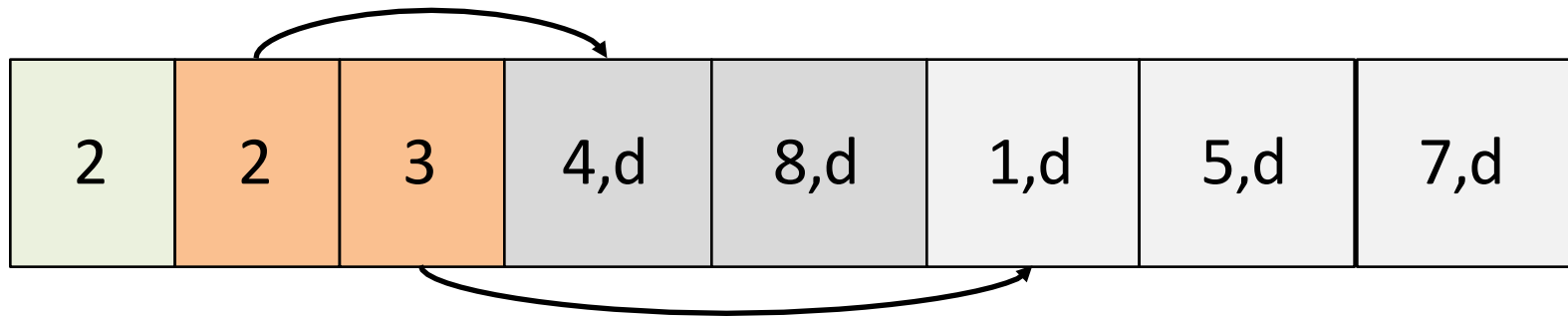
8	3	10	1	6	-1	14	-1	-1	4	7	-1	-1	13	-1
---	---	----	---	---	----	----	----	----	---	---	----	----	----	----

Размер:  $\text{sizeof}(\text{item}) * 2^{\text{depth}-1}$

# Представление хэш-таблицы

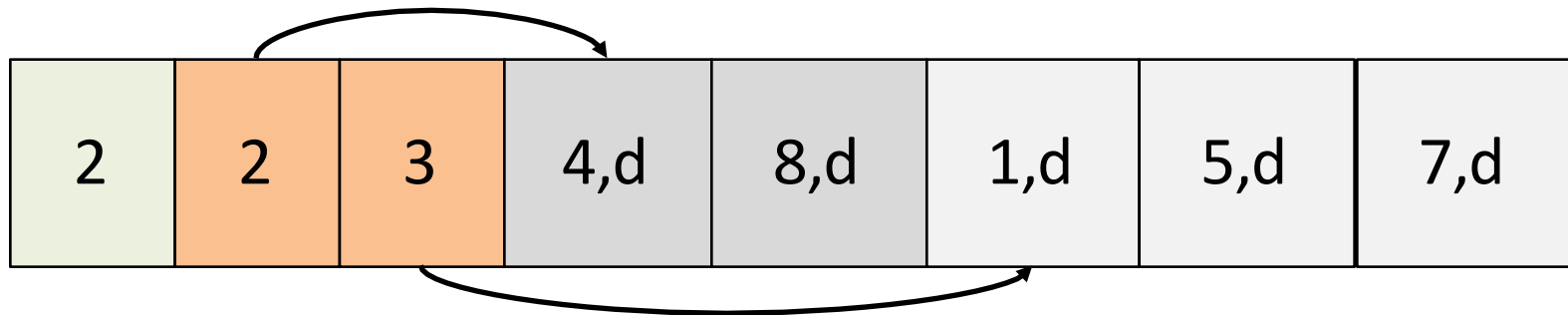
- Считаем, что ключ фиксированного размера
- Хранимые данные - тоже
- Делим данные на корзинки  $\% N$ 
  - Выбираем  $N$  чтобы сгруппировать данные в  $\sim 4K$
  - Сортируем по ключу

# сериализация хэш-таблицы



- 3 входных параметра загрузки:
  - Кол-во корзин (N)
  - Размерность ключа
  - Размерность данных

# сериализация хэш-таблицы



- Поиск:
  - Корзина=hash(word) % N
  - Смещение корзины известно
  - Внутри корзины – бинарным поиском по ключу
  - Возвращаем [d]

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

## Дерево запроса

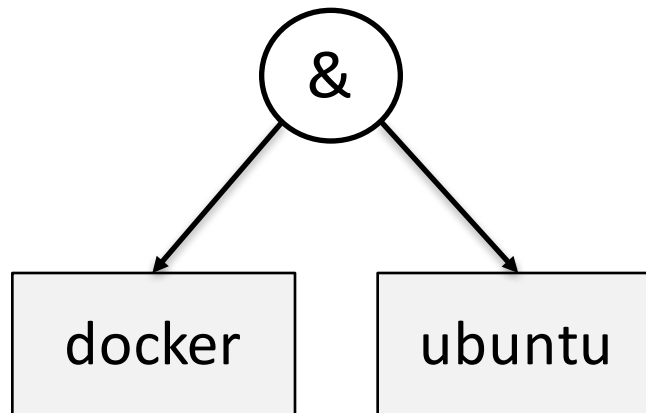
1. Исполнение дерева
2. Парсинг запроса

## Общий workflow



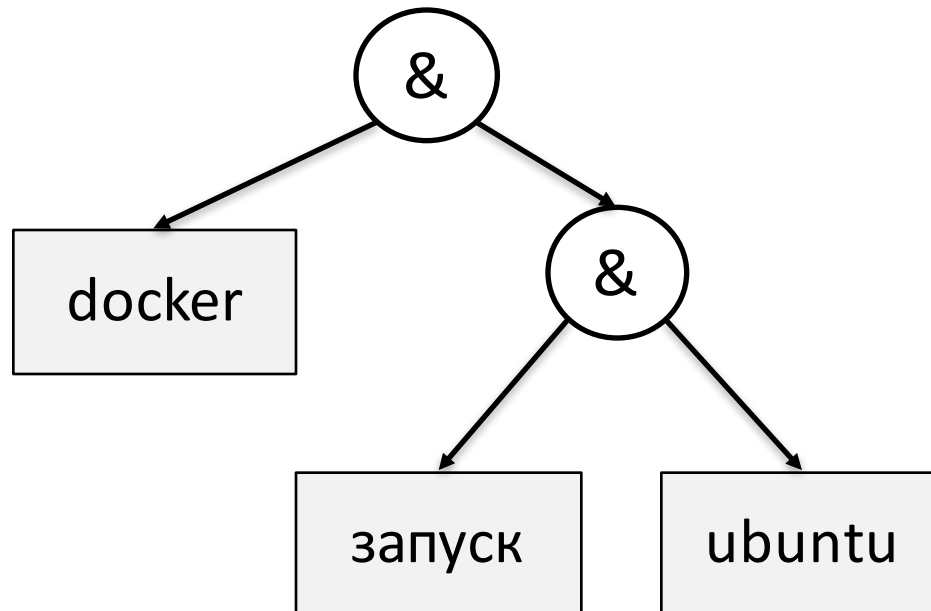
# Представление запроса

- Простой случай: docker ubuntu
- Эквивалентно docker & ubuntu



# Представление запроса

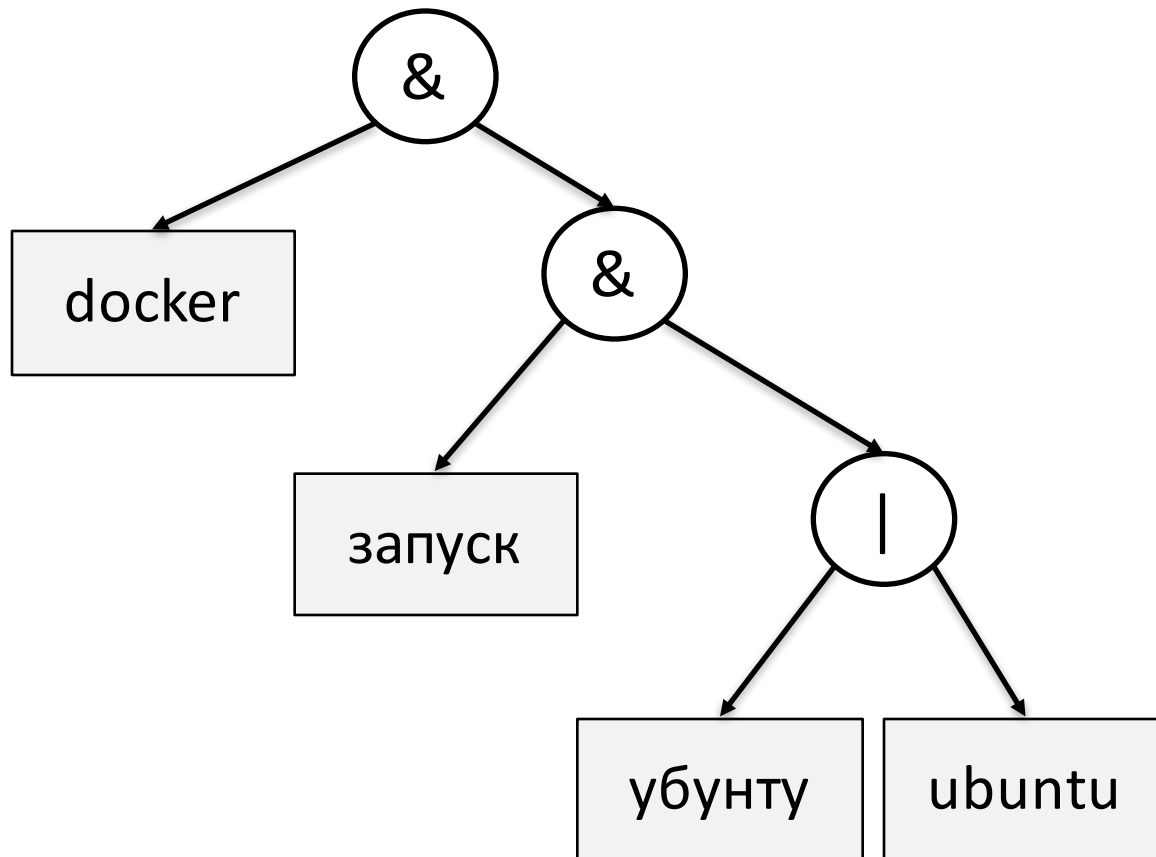
- Расширим: docker & запуск & ubuntu





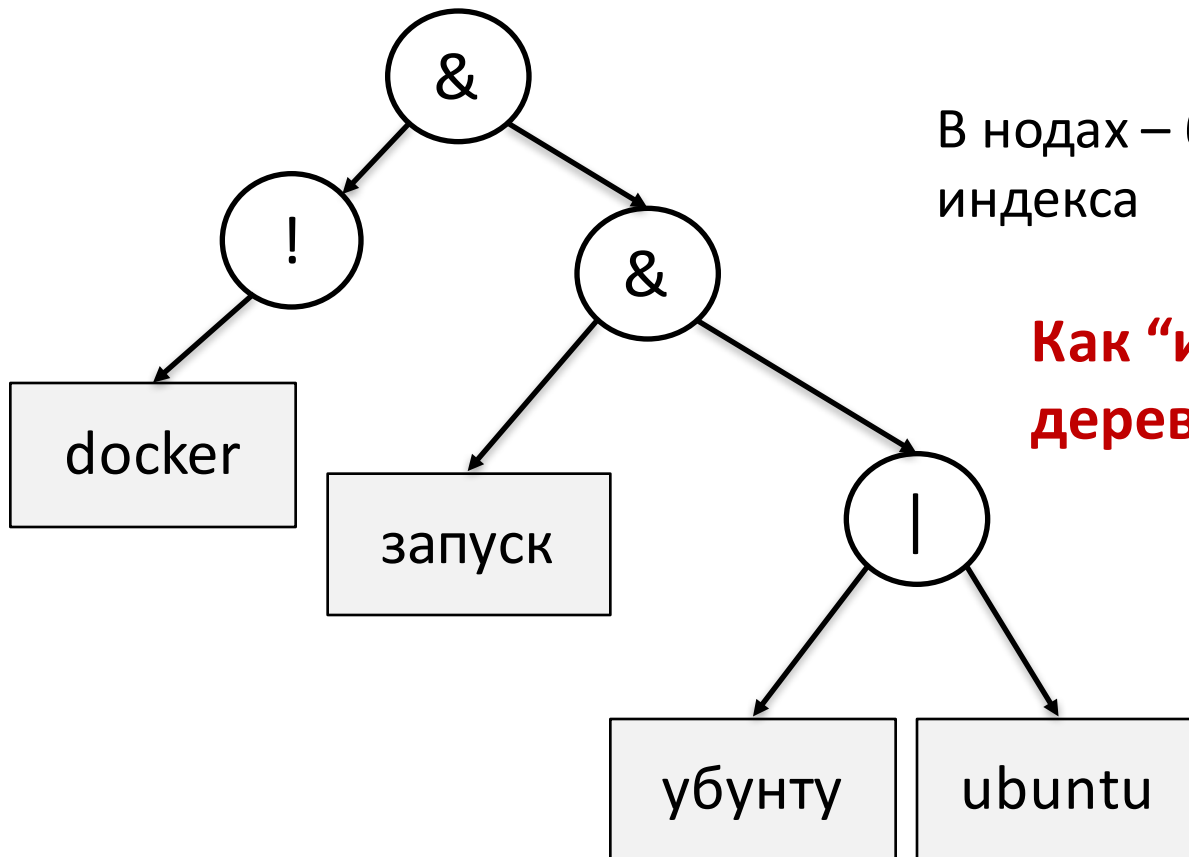
# Представление запроса

- Расширим: `docker & запуск & (ubuntu | убунту)`



# Представление запроса

- Полный: !docker & запуск & (ubuntu | убунту)



В нодах – блоки обратного индекса

**Как “исполнять” такое дерево?**

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

## Дерево запроса

- 1. Исполнение дерева**
2. Парсинг запроса

## Общий workflow



# Исполнение дерева: “в лоб”

- Все термы – set-ы
- Делаем соответствующую операцию рекурсивно:
  - ! “docker” (A)
  - “ubuntu” | убунту” (B)
  - & “запуск” (B)
  - & A
- Представим что у нас  $10^7$  документов
- Отрицание -> |M|

# Исполнение дерева: пошагово

- Считаем что:
  - каждый узел представим потоком
    - `evaluate()`
    - `goto(docid)`
  - doc монотонно возрастает
  - 2 специальных значения: `AlphaID`, `OmegalD`

# Исполнение дерева: алгоритм

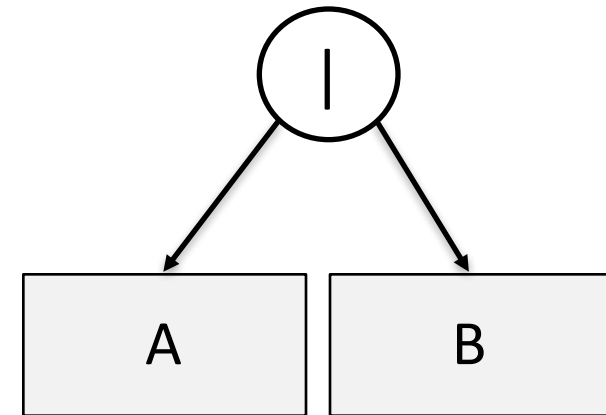
- Алгоритм:
  - Разберем дерево, и свяжем каждый узел классом операций (`get_current`, `goto`)
  - Начнем с `last_doc_id = AlphaID` (`=-1`)
  - На каждом шаге рекурсивно:
    - Перемещаемся на `last_doc_id+1`
    - Вызываем `evaluate()`

# Специфика термов

- Evaluate = current doc id
- Goto(docid) =
  - Пропуск всех  $id < docid$
  - Опционально используя jump tables

# Специфика дизъюнкции

- `goto(docid) =`
  - `A.goto(docid)`
  - `B.goto(docid)`
- `evaluate() =`
  - `a = A.evaluate(), b = B.evaluate()`
  - `(a == OmegalD) => b, (b == OmegalD) => a`
  - `=> min(a, b)`





# Дополнительно для ДЗ

- Конъюнкция
- Отрицание
- Отсутствующие термы (какой случай?)

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

## Дерево запроса

1. Исполнение дерева
- 2. Парсинг запроса**

## Общий workflow



# Разбор запроса

- ОК, знаем как исполнять дерево
- Необходимо сформировать из запроса
- `!docker & запуск & (ubuntu | убунту)`

# Разбор запроса

- !docker & запуск & (ubuntu | убунту)
- Подход:
  - Токенизация
  - Формирование дерева

# Разбор запроса: токенизация

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

# Разбор запроса: дерево

## — Алгоритм:

- Находим самый низкоприоритетный оператор
  - Наиболее внешний, наиболее правый
  - Запомним как token
- Если не нашли, значит результат = терм | None
- Иначе:
  - token.left = рекурсивно слева
  - token.right = рекурсивно справа
  - Результат = является token

# ! docker & запуск & ( ubuntu | убунту )



!docker & запуск

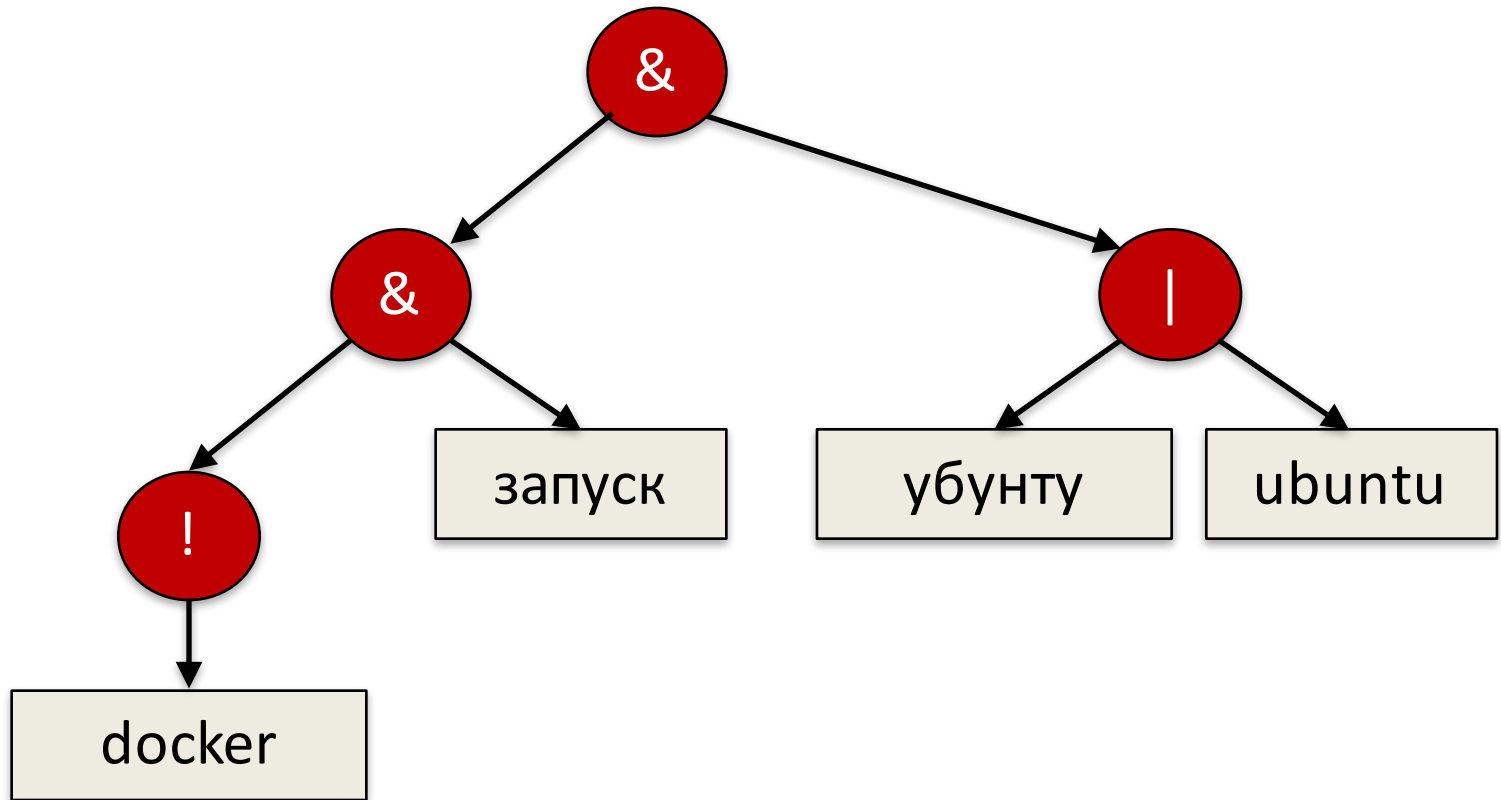
ubuntu | убунту

# ! docker & запуск & ( ubuntu | убунту )





! docker & запуск & ( ubuntu | убунту )



## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

## Дерево запроса

1. Исполнение дерева
2. Парсинг запроса

## Общий workflow



# Общий workflow индексации

- Индексация входных документов [index.sh]
- Оптимизация индекс
- Построение словаря [make\_dict.sh]
- Поиск [search.sh]

# Workflow: индексация

- Наиболее затратна по времени
- А также по RAM
  - Для больших коллекций нельзя держать все

Поэтому:

- Сбрасываем порциями на диск
- Делаем ссылки на блоки ( != словарь )
- Сбрасываем соответствие docid -> url

# Workflow: оптимизация

- Искать среди  $N$  блоков невыгодно:
  - Разрозненные обращения к диску
  - Больше  $N$   $\rightarrow$  больше словарь
  - $-//-$   $\rightarrow$  хуже сжатие

Поэтому:

- Склеиваем блоки: один терм  $\rightarrow$  один блок
- Строим бинарный словарь

# Workflow: поиск

- Загружаем словарь
- Для каждого запроса:
  - Разбираем дерево запроса
  - Ставим в соответствие блоки
  - Пошагово получаем docid
    - Конвертируем docid в URL

## Еще раз о сжатии

1. Simple9
2. Бинарные данные в Python

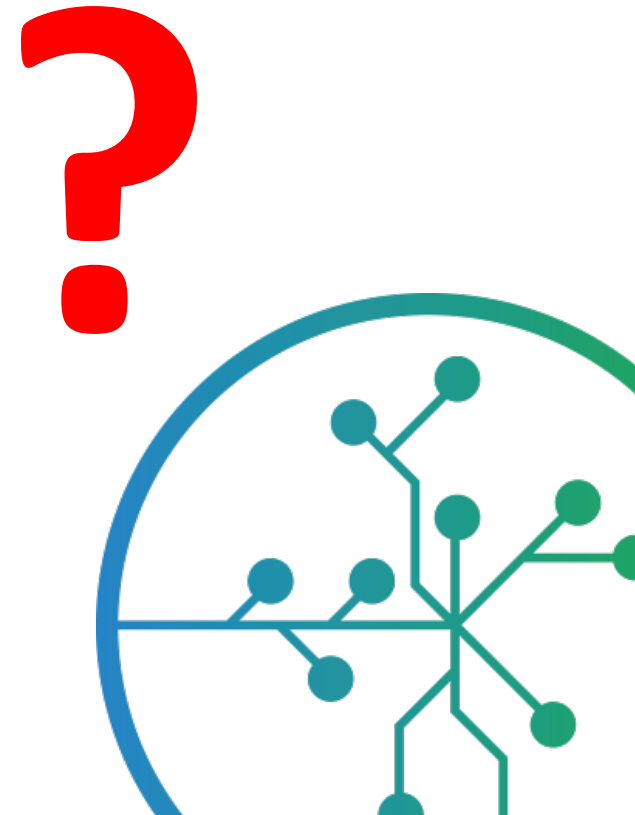
## Словарь поиска

1. Представление стоп-слов
2. Хранение словаря

## Дерево запроса

1. Исполнение дерева
2. Парсинг запроса

## Общий workflow



# Have a nice homework!

