Современные методы улучшения качества поиска

Яндекс
Den Raskovalov
denplusplus@yandex-team.ru
Санкт-Петербург
25.02.2011

Краткая история IR

1950: Боязнь научного отставания от СССР подстегивает работы по построению мехнизированных систем поиска, изобретению индекса цитирования

1975: Salton публикует свои основные работы (TF*IDF)

1992: Первый TREC

~2000: Индустриализация IR с широким распространением web'a и появлением поисковых машин

2003: РОМИП

Состояние IR

- Не наука
- Близко не подошла к пониманию смысла текста
- Роль эвристики велика, как нигде
- Накоплено и развито некоторые техники:
  - Морфология
  - Машинное обучение
  - Обработка огромных объемов косвенных данных (логи запросов -> синонимы)

Как происходит поиск?

- Запрос токенизируется, к словам запроса применяется морфологический анализ, ищутся синонимы
- Из документов индекса отбираются те, которые с большой вероятностью имеют отношение к запросу (фильтрация)
- Для отфильтрованных документов рассчитываются признаки (фичи, features)
- К признакам применяется формула, дающая конечную оценку релевантности

Инвертированный индекс

- Позволяет для данного ключа (слова) найти (проитерироваться) по его вхождениям (позициям) в документы коллекции
- Есть возможность дополнительно хранить информацию про вхождение (сегмент, позицию)

- Получается, что все, что мы знаем про документ, когда считаем релевантность – это позиции слов запроса (мало?)

Инвертированный индекс, структура и реализация

- Минимальная реализация
- Два файла:
  - Key – отсортированный список слов, с указанием, где они хранятся в inv-файле
  - Inv – плоский файл с информацией о позициях
    - Позиции для одного ключа идут подряд
    - Позиции отсортированы по (id документа, позиции в документе)

Читать удобно с помощью memory map

Инвертированный индекс, его построение

- Проблема: индексатор получает документ за документом
- Накапливать инвертированный индекс в памяти map< string, vector<TPosition> >
- Когда памяти перестает хватать, записать порцию инвертированного индекса на диск
- Когда документы кончатся, слить порции инвертированного индекса с диска в один индекс по всем документам

Инвертированный индекс: слияние

- Идея для слияния используется та же, что и для внешней сортировки
  - Завести heap, хранящий итераторы на порции
  - Записывать каждый раз в выходной файл минимальную позицию, пока все не закончатся

  Можно разработать весьма эффективные алгоритмы сжатия инвертированного индекса (дельта-кодирование, префиксное сжатие)

Фильтрация

- Запрос состоит из нескольких слов, какие документы считать найденными?
- Зачем нужна? На самом деле, для оптимизации. Цель – не потерять релевантные документы.
- Те, которые содержат слова запроса. Содержат где?
- Есть текст, есть ссылки на документ (ссылки можно трактовать, как хорошие описания)
- [мой дядя самых чистых правил, когда не в шутку занемог]
- [скачать учебник философия вуз платон и демокрит pdf djvu torrent]
- AND?
- OR?
- Все или почти все слова запроса.

Фильтрация: кворум

Q – запрос
q_i – i-ое слово запроса
$w(q_i)$ – функция веса слова
D - документ

$$\sum_{q_i \in D} w(q_i) > Quorum(Q) \cdot \sum_{q_i \in Q} w(q_i)$$

$$Quorum(Q) = 1 - 0.01^{\left(1/\sqrt{(|Q|-1)}\right)}$$

$$w(q_i) = -\log\left(DocFreq(q_i)/SumOfFreq\right)$$

Фильтрация: идеи

- Слова в заголовках важнее, чем в остальном тексте
- Существительные важнее, чем прилагательные
- Очень редкие слова только мешают (опечатки)
- Стоп-слова (предлоги, союзы) должны иметь нулевой вес
- Иногда очень частые слова очень важны, их нельзя отбрасывать (география) [нотариус москва]

- Решение – выбрать метрику, составить обучающую выборку, произвести машинное обучение

Вопросы?

Ранжирование

- Для того, что работать над качеством ранжирования, надо уметь его измерять (трюизм, да).
- В основе оценки качества работы любого алгоритма лежит сравнение результатов его работы с результатом работы человека.

Ранжирование: тексты

- До появления интернета, был только текст документа
  - TF-IDF (Salton)
  - BM25 (Robertson)
    - $$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})},$$
    - Пенальти длинным документам
    - Пенальти большому числу вхождений слова
  - Морфология
  - Тезаурусы (синонимы)
  - BM25F - Зоны

Ранжирование: ссылки

- С появлением WWW и HTML появляются ссылки. Зачастую они хорошо описывают документ. Сам факт их наличия много говорит о документе:
  - LF-IDF
  - LinkBM25

Ранжирование: PageRank

- Рассмотрим граф. Вершины – страницы интернета, ребро – ссылка.
- Рассмотрим модель “блуждающей обезьянки”.
- PageRank страницы – мера времени, которая обезьянка проводит на странице.
- Не зависящая от запроса мера важности страницы в интернете.

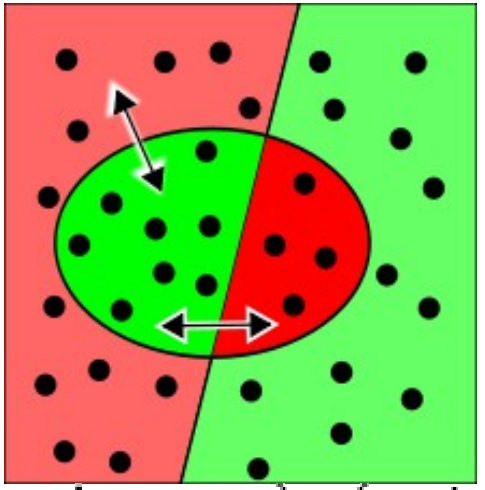
Ранжирование: клики

- Можно собирать реацкию пользователя. По нерелевантному не кликают. На нерелевантном не проводят время.

Ранжирование

- Одна из основных проблем ранжирования: как научиться сочетать столь разнородные сигналы?
- Для того, что работать над качеством ранжирования, надо уметь его измерять (трюизм, да).
- В основе оценки качества работы любого алгоритма лежит сравнение результатов его работы с результатом работы человека.

Метрики: бинарный классификатор



$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Метрики: ранжирование

- Point-wise
  - Например, невязка (сумма квадратов отклонение оценок от предасказанной релевантности)
- Pair-wise
  - Например, число неправильно отранжированных пар
- List-wise
  - Сумма по всем запросам
    - Релевантности первого документа (P1)
    - Релевантности первых десяти документов (P10)
    - NDCG (сумма релевантностей, нормированный на позицию документа)

Метрики ранжирования: pFound

- В основе метрики лежит модель поведения пользователя:
  - Пользователь просматривает выдачу сверху-вниз результат за результатом
  - После просмотра каждого результата пользователь может остановить поиск:
    - Текущий результат решил его поисковую задачу
    - Он отчаялся
  - pFound – это вероятность того, что пользователь нашел
  - pView[i] = pView[i-1]*(1 – pSuccess[i – 1]))*0.85
  - pFound = sum pView[i]*pSuccess[i]
- Приятное свойство: один раз собрав оценки результатов поиска, можно оценивать результаты работы разных алгоритмов.

Проблема недооцеенности

- Оценки быстро устаревают
- Релевантные документы исчезают
- Все классические метрики при этом стремятся к нулю по абсолютному значению и не сохраняют порядок ранжирования систем

Анализ поведения пользователей

- “Счастье”
- Кликовые метрики
- Средняя позиция первого (по времени!) клика
- Доля некликнутых страниц
- Доля длинных кликов
- Общая кликабельность выдачи
- A/B тестирование

A/B тестирование

- Проверка «единственного» небольшого изменения
- Условия: наличие популярной поисковой системы
- Большинство пользователей все еще используют старую систему
- Малый процент (1-2-4-10%) выполняем в новой системе
- Одновременное выполнение
- Анализ затронутых запросов
- «Общие» запросы для стандартной и экспериментальной систем
- Менее точно, низкая разрешающая способность, но надежно и не требует оценок

A/B тест: пример эксперимента

Показатели качества

Результаты предварительного тестирования на 6% России

11.11.2009	контрольная выборка	новые формулы
Доля некликнутых	40.8%	40.6%
Позиция первого клика	2.03	2.04
Время до первого клика	7.33 сек	7.35 сек
Кликов на запрос	1.97	1.96
Доля инверсий	27.7%	28.0%
Доля «длинных» кликов	78.8%	78.9%
Количество запросов	1048212	1031559

Алгоритмы машинного обучения

- Метод ближайшего соседа
- SVM (попытка линейно разделить релевантное и нерелевантное)
- Жадный перебор полиномиальной формулы

- Очень хочется пользоваться градиентным спуском. Но как?

Алгоритмы машинного обучения

- Проблема в том, что pFound недифференцируем. На помощь приходит модель Люка-Плакетта.
- После этого можно применять градиентный спуск.
- Полезный прием: bagging.
- Бустим много раз: используем жадность.
- В качестве базовых примитивов используем “кубики” малой размерности.
- Не забудем про регуляризацию.

Алгоритмы машинного обучения

- В результате получаем MatrixNet.
- Ближайший аналог: TreeNet от господина Фридмана.
- В реальном мире нужно распараллелить на множество машин.

Машинное обучение

- С помощью машинного обучения в Яндексе решается множество задач:
  - Решение, что обходить
  - Решение, что выкладывать
  - Сниппеты
  - Задачи производительности
  - Детекция спама
  - Решение о показе рекламы

Что еще?

- Множество источников (новости, блоги, товары)
- Актуальность
- Свежесть
- Непорнушность
- Дубликаты
- Разнообразие интентов
- Спам
- Подавление спама

Спасибо.

Вопросы?

Современные методы увеличения производительности поиска

Организация разработки в поиске Яндекса

Яндекс
Den Raskovalov
denplusplus@yandex-team.ru
25.02.2011
Санкт-Петербург

Базовый поиск

- Базовый поиск – это программа, которая по поисковому индексу умеет находить самые релевантные документы, строить сниппеты, возвращать дополнительную документную информацию.
- Индекс обычно содержит миллионы документов.
- Реализован, как HTTP-сервер (для отладки подходит браузер).
- В базовом поиске тратятся 95% всех ресурсов при обслуживании запроса.
- Стоимость кластера - десятки миллионов долларов. 1% производительности - сотни тысяч долларов.

Базовый поиск

- Все данные, нужные для поиска, хранятся в оперативной памяти.
- Форматы, используемые в индексе, рассчитаны на прямую совместимость с системой индексации.
- В разработке программы принимали/принимают участие ~500 программистов.
- Чтобы часто тестировать качество/скорость, нужно, чтобы это не требовало человеческих ресурсов.
- Профайлер – ваш лучший друг (vtune, valgrind, callgrind, kcachegrind).
- Некоторые оптимизации не меняют результаты поиска, некоторые – их немного ухудшают.

```
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1124
  e886b0 9557   1.32% |X                of b7 51 0a         movzwl 0xa(%rcx),%edx
  e886b4 3252   0.45% |                66 41 of 6f 44 95 00 movdqa 0x0(%r13,%rdx,4),%xmm0
  e886bb 57960   8.02% |XXXXXXX           of b7 51 08         movzwl 0x8(%rcx),%edx
▶ e886bf 540   0.07% |                66 of 72 f0 05       pslld  $0x5,%xmm0
  e886c4 11707   1.62% |X               66 45 of 6f 4c 95 00 movdqa 0x0(%r13,%rdx,4),%xmm9
  e886cb 31128   4.31% |XXX              of b7 51 06         movzwl 0x6(%rcx),%edx
  e886cf 308   0.04% |                66 41 of 72 f1 04    pslld  $0x4,%xmm9
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1004
  e886d5 6414   0.89% |                66 41 of fe c1       paddd  %xmm9,%xmm0
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1124
  e886da 6969   0.96% |                66 45 of 6f 4c 95 00 movdqa 0x0(%r13,%rdx,4),%xmm9
  e886e1 23915   3.31% |XX               of b7 51 04         movzwl 0x4(%rcx),%edx
  e886e5 243   0.03% |                66 41 of 72 f1 03    pslld  $0x3,%xmm9
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1004
  e886eb 5732   0.79% |                66 41 of fe c1       paddd  %xmm9,%xmm0
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1124
  e886f0 6289   0.87% |                66 45 of 6f 4c 95 00 movdqa 0x0(%r13,%rdx,4),%xmm9
  e886f7 22746   3.15% |XX               of b7 51 02         movzwl 0x2(%rcx),%edx
  e886fb 208   0.03% |                66 41 of 72 f1 02    pslld  $0x2,%xmm9
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1004
  e88701 6064   0.84% |                66 41 of fe c1       paddd  %xmm9,%xmm0
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1124
  e88706 6357   0.88% |                66 45 of 6f 4c 95 00 movdqa 0x0(%r13,%rdx,4),%xmm9
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1004
  e8870d 20585   2.85% |XX               of b7 11            movzwl (%rcx),%edx
../../../kernel/relevfml/mx_calcer.cpp:188
  e88710 267   0.04% |                48 83 c1 0c          add    $0xc,%rcx
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1124
  e88714 2207   0.31% |                66 41 of 72 f1 01    pslld  $0x1,%xmm9
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1004
  e8871a 8174   1.13% |X                66 41 of fe c1       paddd  %xmm9,%xmm0
  e8871f 1812   0.25% |                66 41 of fe 44 95 00 paddd  0x0(%r13,%rdx,4),%xmm0
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:1312
  e88726 25525   3.53% |XXX              66 of c5 d0 00       pextrw $0x0,%xmm0,%edx
/usr/local/lib/gcc44/gcc/x86_64-portbld-freebsd7.2/4.4.3/include/emmintrin.h:986
  e8872b 21340   2.95% |XX               44 of b6 c2          movzbl %dl,%r8d
  e8872f 5343   0.74% |                of b6 d6            movzbl %dh,%edx
  e88732 8224   1.14% |X                66 44 of 6e 14 90    movd   (%rax,%rdx,4),%xmm10
  e88738 160297  22.19% |XXXXXXXXXXXXXXXXXXXX 66 46 of 6e 0c 80    movd   (%rax,%r8,4),%xmm9
```

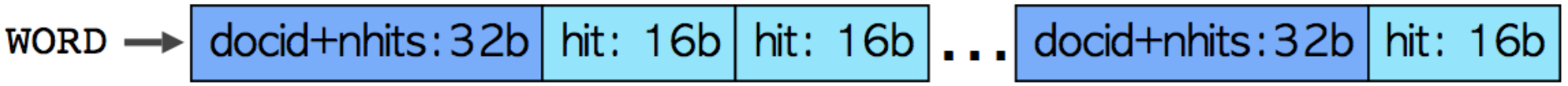
Базовый поиск: тестирование производительности

```
      start time              1314962615638723, Fri Sep  2 15:23:35 2011
end time                  1314963152282997, Fri Sep  2 15:32:32 2011
full time                 536644274, 00:08:56
data readed               628043057
requests                  200000
error requests            0
requests/sec              372.686
avg req. time             53512.823
avg req. len              3140.215
req time std deviation    97724.924

Ok                        200000

L range(865, 2815548)
 h[0]  (0       ..865)      = 1      (0.000%),  200000 (100.000%)
 h[1]  (865     ..141599)   = 184932 (92.466%), 199999 (99.999%)
 h[2]  (141599  ..282333)   = 10828  (5.414%),  15067  (7.533%)
 h[3]  (282333  ..423067)   = 2119   (1.059%),  4239   (2.119%)
 h[4]  (423067  ..563801)   = 788    (0.394%),  2120   (1.060%)
 h[5]  (563801  ..704535)   = 466    (0.233%),  1332   (0.666%)
 h[6]  (704535  ..845269)   = 287    (0.143%),  866    (0.433%)
 h[7]  (845269  ..986004)   = 201    (0.100%),  579    (0.289%)
 h[8]  (986004  ..1126738)  = 131    (0.065%),  378    (0.189%)
 h[9]  (1126738..1267472)  = 102    (0.051%),  247    (0.123%)
 h[10] (1267472..1408206)  = 35     (0.017%),  145    (0.072%)
 h[11] (1408206..1548940)  = 32     (0.016%),  110    (0.055%)
 h[12] (1548940..1689674)  = 18     (0.009%),  78     (0.039%)
 h[13] (1689674..1830408)  = 22     (0.011%),  60     (0.030%)
 h[14] (1830408..1971143)  = 17     (0.008%),  38     (0.019%)
 h[15] (1971143..2111877)  = 4      (0.002%),  21     (0.010%)
 h[16] (2111877..2252611)  = 6      (0.003%),  17     (0.008%)
 h[17] (2252611..2393345)  = 3      (0.001%),  11     (0.005%)
 h[18] (2393345..2534079)  = 5      (0.002%),  8      (0.004%)
 h[19] (2534079..2674813)  = 1      (0.000%),  3      (0.001%)
 h[20] (2674813..2815548)  = 2      (0.001%),  2      (0.001%)
 h[21] (2815548..)         = 0      (0.000%),  0      (0.000%)
```

Оптимизации

- Сжатие инвертированного индекса
- Фильтрация
- Синтаксический колдунщик
- Pruning
- Fastrank
- Параллельное выполнение запроса
- Частичное выполнение MatrixNet'a
- Отсечение по времени выполнения запроса
- Использование SSEk инструкций процессора
- Кодогенерация

- Original encoding ('97) was very simple:



  - hit: position plus attributes (font size, title, etc.)

  - Eventually added skip tables for large posting lists

- Simple, byte aligned format
  - cheap to decode, but not very compact
  - ... required lots of disk bandwidth

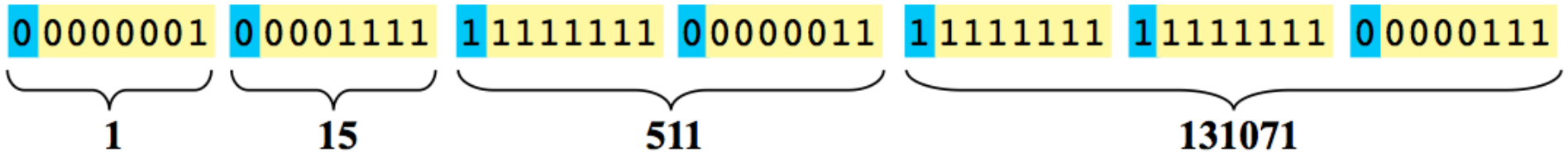
Google

- Bit-level encodings:
  - Unary: *N* '1's followed by a '0'
  - Gamma: $\log_2(N)$ in unary, then floor(log2(*N*)) bits
  - Rice$_K$: floor(N / 2^K) in unary, then N mod 2^K in K bits
    - special case of Golomb codes where base is power of 2
  - Huffman-Int: like Gamma, except $\log_2(N)$ is Huffman coded instead of encoded w/ Unary

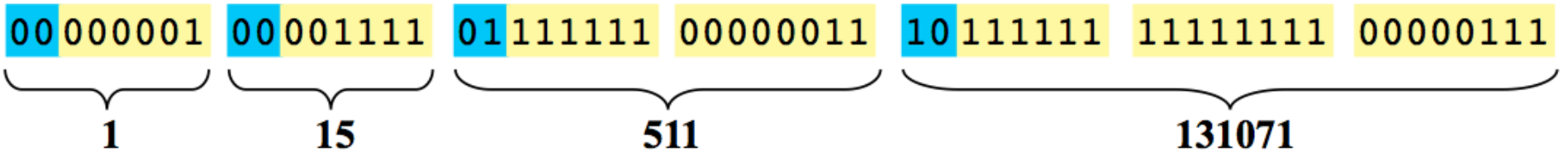
- Byte-aligned encodings:
  - varint: 7 bits per byte with a continuation bit
    - 0-127: 1 byte, 128-4095: 2 bytes, ...
  - ...

Google

- Varint encoding:
  - 7 bits per byte with continuation bit
  - Con: Decoding requires lots of branches/shifts/masks

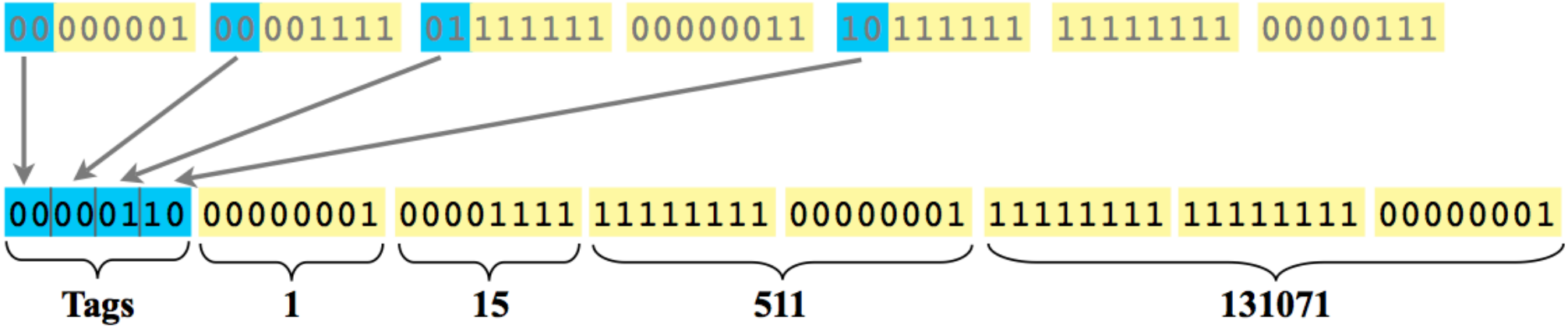


- Idea: Encode byte length as low 2 bits
  - Better: fewer branches, shifts, and masks
  - Con: Limited to 30-bit values, still some shifting to decode



Google

- Idea: encode groups of 4 values in 5-17 bytes
  - Pull out 4 2-bit binary lengths into single byte prefix



- Decode: Load prefix byte and use value to lookup in 256-entry table:

```
                    ...
00000110  ------>   Offsets: +1,+2,+3,+5; Masks: ff, ff, ffff, ffffff
                    ...
```

- Much faster than alternatives:
  - 7-bit-per-byte varint: decode ~180M numbers/second
  - 30-bit Varint w/ 2-bit length: decode ~240M numbers/second
  - Group varint: decode ~400M numbers/second

Google

Сжатие инвертированного индекса

- Алгоритмы компрессии списка позиций хорошо работают для групп близких чисел.
- Хорошо бы, чтобы документы, содержащие слово, шли подряд.
- Решение? Отсортировать по url ;)

Фильтрация

- Пример оптимизации: часто документ не модет пройти фильтрацию, если в нем нет некоторых тяжелых слов:

$$\sum_{q_i \in D} w(q_i) > Quorum(Q) \cdot \sum_{q_i \in Q} w(q_i)$$

$$Quorum(Q) = 1 - 0.01^{(1/\sqrt{(|Q|-1)})}$$

- Фактически внутри запроса появляются слова, которые связаны AND.
- Когда мы итерируемся, если мы можем быстро узнать, какой документ следует за текущим, можно сразу идти на max(next_i)

Синтаксический колдунщик

- [скачать учебник философия для вузов платон и демокрит pdf djvu torrent]
- Можно написать алгоритм, который проанализирует синтаксическую согласованность групп слов внутри запроса
- [“скачать учебник” “философия для вузов” “платон и демокрит” “pdf” “djvu” “torrent”]
- Это уменьшит число найденных документов
- Сейчас эта оптимизация не используется, как сильно ухудшающая качество поиска

Pruning

- Pruning – раннее отсечение запроса
- Если мы нашли 40000 документов, то мы обрываем поиск.
- Хорошо бы, чтобы хорошие документы были в самом начале.
- Пусть качество == вероятность показа
- Как же быть с эффективностью сжатия индекса?
- Разобьем на 16 групп по качеству, внутри каждой отсортируем по URL'у.
- Требует итеративной архитектуры базового поиска. Один за другим перебираем документы, один за другим подвергаем их фильтрации, можем посчитать релевантность для одного документа. Может прервать поиск после любого документа.

Fastrank

- Фичей очень много
- MatrixNet содержит множество деревьев.
- Что же делать?
- Нам нужно вернусть обычно лишь 10 лучших результатов.
- Давайте выделим “легкие” фичи, подберем “легкую” функцию ранжирования.

Fastrank

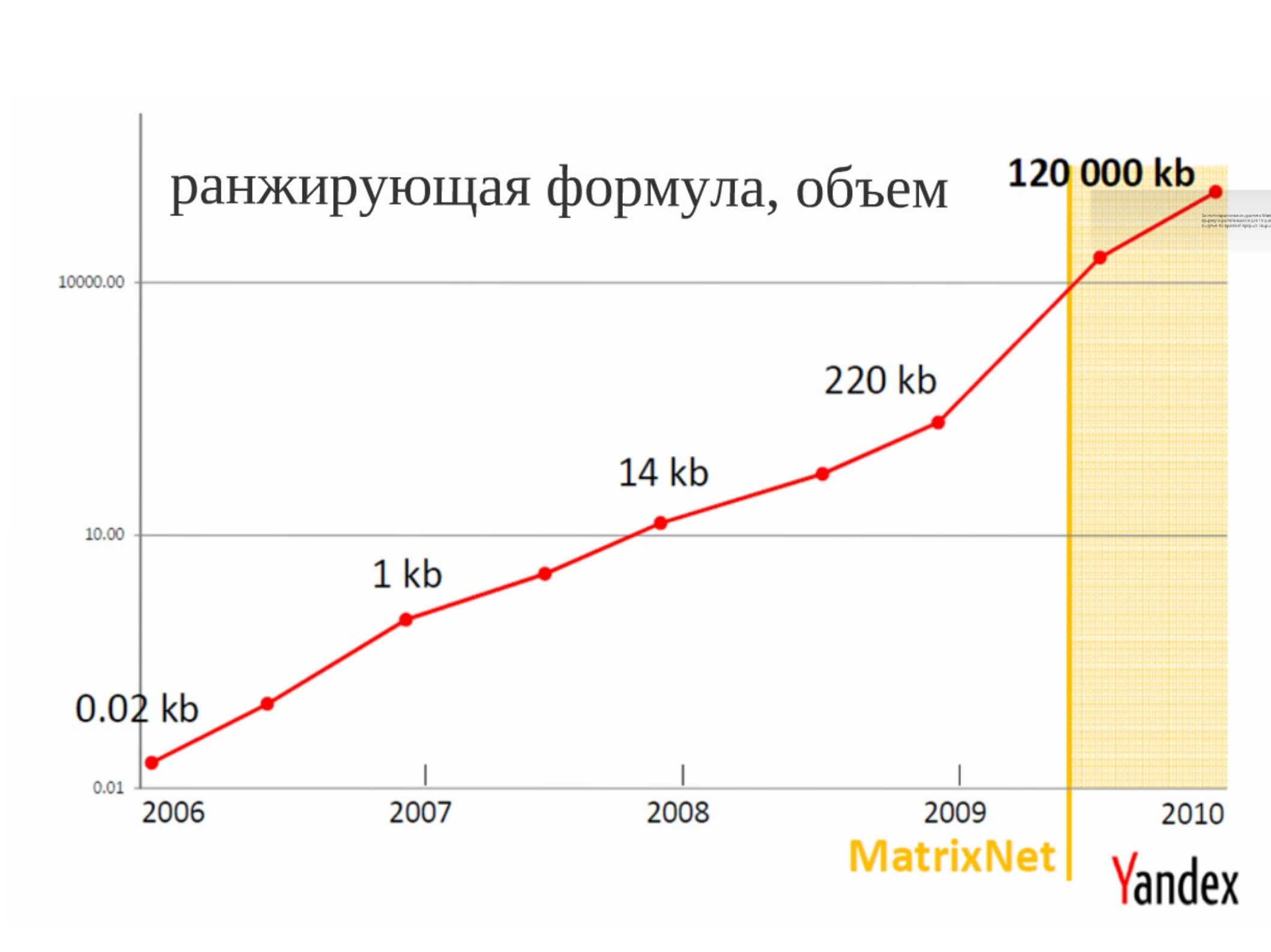
```
if (CurrentDocument.FastRank >
FastRankHeap.Min() || FastRankHeap.Size() <
300) {
  if (FastRankHeap.Size() >= 300)
     FastRankHeap.Pop();
     FastRankHeap.Push(CurrentDocument.FastRank
     );
  PositionCache.Memorize(CurrentDocument);
 }
```

EarlyRank

- Та же идея, только будем оперировать только наличием/отсутствием слов в тексте/ссылках.
- Будем экономить на чтении и распаковке позиций в инвертированном индексе.

Параллельное выполнение запроса

- В типовом современном серверном процессоре 24 ядра.
- Исполнение одного запроса в несколько тредов не увеличиваем емкость (количество запросов, которые можно выполнить за секунду). И даже уменьшает (расходы на слияние).
- Зато позволяет увеличить латентность (среднее время выполнения запроса).
- Можно распараллелить, разбив документы на несколько виртуальных интервалов.
- Можно распараллелить лишь некоторые стадии (после fastrank'a).
- Легкие запросы нет смысла распараллеливать (можно оценить “легкость” прямо во время выполнения запроса).


ранжирующая формула, объем
120 000 kb
10000.00
220 kb
14 kb
10.00
1 kb
0.02 kb
0.01
2006
2007
2008
2009
2010
MatrixNet
Yandex

Частичое выполнение MatrixNet'a

- Результирующая ”формула” MatrixNet'a состоит из тысяч деревьев, каждое из которых дает небольшой вклад в итоговую релевантность. Нет малого числа “очень важных” деревьев.
- Это позволяет применить привычный прием.
- Если базовому поиску нужно вернуть 10 лучших документов, давайте для 300 документов после FastRank посчитаем 1/10 деревьев MatrixNet. И лишь для 30 лучших по “легкому MatrixNet” посчитаем “полный” MatrixNet.

SSE и кодогенерация

- SSE позволяет выполнить одну и ту же операцию (например, сложение, умножение) с векторами из 4/8/16 чисел.

- Именно это и нужно. Можно реорганизовать код так, чтобы вычислять формулы релевантности для “пачек” из 4/8 документов.

Кодогенерация

- Многие данные (формулы) удобнее превратить в код на этапе компиляции.
- Он должен помещаться в кэш комманд процессора.
- Удобно использовать шаблоны C++, для явных указаний, какой вариант чаще всего будет использоваться в production (условные дампы).

```
#include <tmmintrin.h>



__m128i shuffles[256];

//decoded 1G ui32 per second, Intel E5530
inline const ui8 *DecodeSSE(const ui8 *stream, ui32 *res) {
    ui8 symbol = stream[0];
    const __m128i code = _mm_loadu_si128((const __m128i *)(stream + 1));
    const __m128i result = _mm_shuffle_epi8(code, shuffles[symbol]);
    _mm_store_si128((__m128i *)res, result);
    return stream + lengths[symbol];
}
```

Отсечение по времени выполнения запроса

- Хорошо бы, чтобы базовый поиск отвечал всегда за 200ms. Такое ограничение может проистекать из требования, чтобы пользователь всегда получал ответ за 1s.
- Просто ограничить время выполнения запроса на одной машине нельзя? Почему?
- Давайте в процессе исполнения запроса накапливать счетчики:
  - Число скипов
  - Число обработанных позиций
  - Число найденных документов
  - ...
- Построим регрессионную формулу, предсказывающую время выполнения запроса по внутренним статистикам.
- Если формула предсказывает, что мы работаем уже больше 200ms, прервем поиск.

Вопросы?