Effizienzaspekte von Information Retrieval

Ralf Schenkel

This talk includes material from

- Andreas Broschart
- Debapriyo Majumdar
- Martin Theobald
- Gerhard Weikum

- and myself ;-)

Outline

- Efficient Centralized Query Processing
  - **Introduction**
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - Non-Traditional Top-K Processing
- Efficient Precomputation
  - The Map-Reduce Framework
- Efficient Distributed Query Processing

Part 1 – Efficient Query Processing

General Data Model:

- Set of data items D („documents“)
- Set of attributes T („terms“)
- Each item d:
  - Set of attribute values $s_t(d)$ for all $t \in T$ („term frequencies“, „scores“)
  - Importance weight w(d) („page rank“)

Any implementations of values and weights suitable that satisfy
$0 \leq s_t(d) \leq 1$ and $0 \leq w(d) \leq MAX_WEIGHT$

(Simple) Query Model

Input: query q={t1…tn} on D

„efficient query processing“ on the Web

Output: subset $R \subseteq D$ of items where

$F(s_{t1}(d),\ldots,s_{tn}(d)) \geq \theta$ „score of d for q“

for some (monotonous) function F

and some threshold θ

More fancy query models:
Term weights, mandatory terms, negative terms, phrases

Common Instances of this Model

- **Boolean** (unranked) **queries** and scores: $s_t(d)=1$ iff d contains t, 0 else
  - conjunctive Boolean: „efficient and effective“ $F(x1\ldots xn)=x1\cdot\ldots\cdot xn=\min(x1\ldots xn)$
  - disjunctive Boolean: „efficient or effective“ $F(x1\ldots xn)=x1+\ldots+xn$ or $F(x1\ldots xn)=\max(x1\ldots xn)$
  - Threshold $\theta=1$ in both cases

Common Instances of this Model

- **Ranked queries**:
  - $s_t(d) \sim$ importance of t in d,
    importance of t in D,
    features of D (like length), …
  - most frequent implementation of F:
    F(x1…xn)=x1+…+xn (summation)
  - Threshold θ = score of kth result in score order

tf*idf,
BM25 Okapi

Focus of part 1:
„Find the k results with highest aggregated score"

Part 1 – Efficient Query Processing

Different aspects of efficiency:

1. **user-oriented**: minimize query answer time
2. **system-oriented**: maximize query throughput
3. **resource-oriented**: minimize disk accesses, memory footprint, CPU cycles, energy consumption, …

Difficult to optimize 1+2 together; combine goals:
Maximize throughput such that query answer time is below 0.1s for 95% of queries

Part 1 – Efficient Query Processing

Different aspects of efficiency:

1. **user-oriented**: minimize query answer time
2. **system-oriented**: maximize query throughput
3. **resource-oriented**: minimize disk accesses, memory footprint, CPU cycles, energy consumption, …

Focus of part 1:
Resource-oriented optimization to reduce answer time

Part 1 – Efficient Query Processing

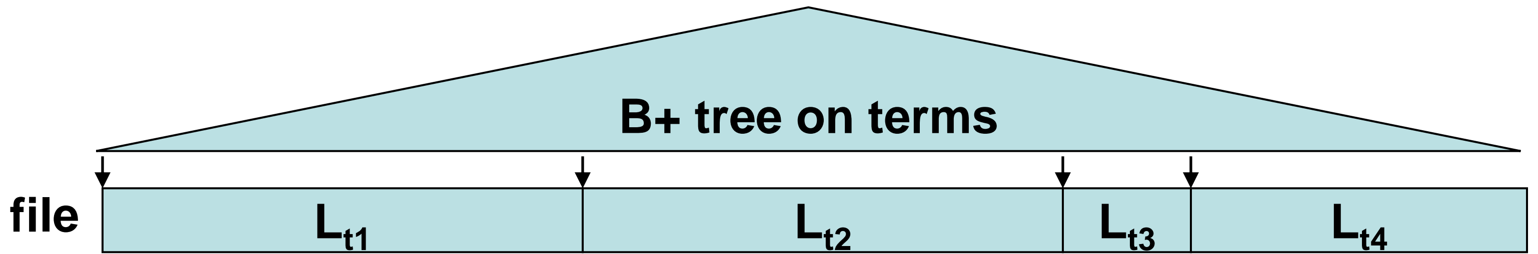
Fundamental data structure: **Inverted List**

- Inverted List L(t) for a term t consists of sequence of tuples **(d,payload)**
- each d contains term t
- payload is additional information
  - $s_t(d)$
  - frequency of t in d
  - positions of t in d (for phrases)

Order of tuples depends on **processing strategy**

Inverted Lists

- Implementation usually as compressed file with all inverted lists for a collection plus access index



- Alternative implementation (simpler, but slower): use big database table with index on t (plus additional columns, depending on sort order)

Index Compression

Why?

- Smaller index, may fit in memory
- Faster list access when stored on disk

Comes with two kinds of execution cost:

- Compression effort at indexing time
- Decompression effort at query time

Important to keep this low

Compression/Performance Tradeoff

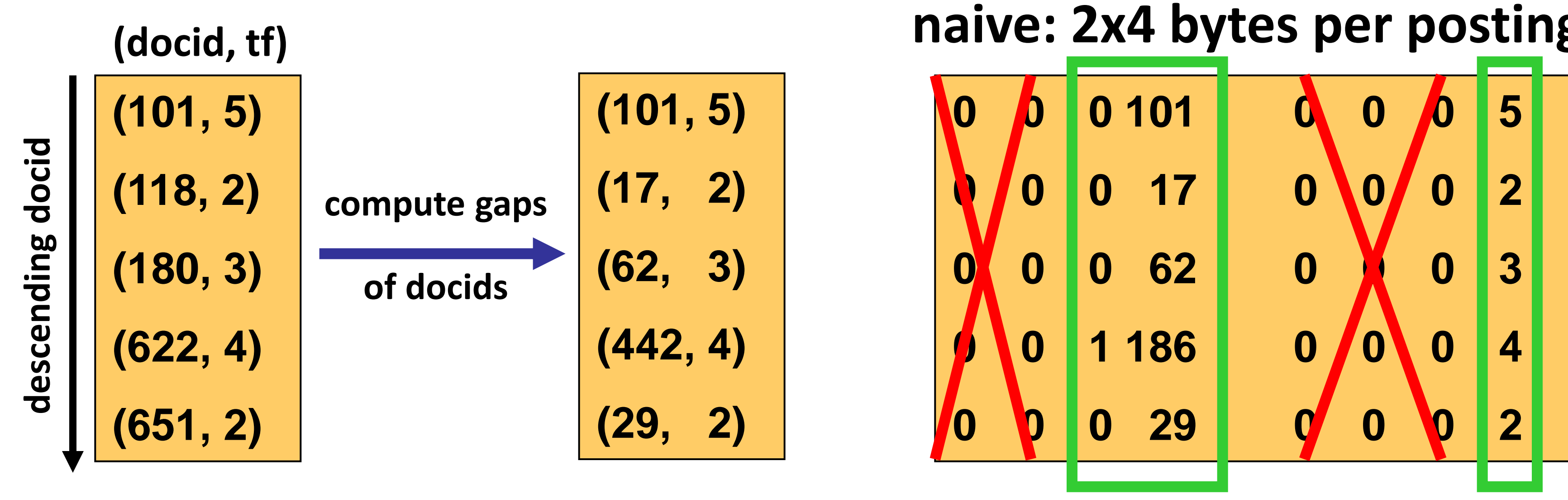
When does it pay off to compress?

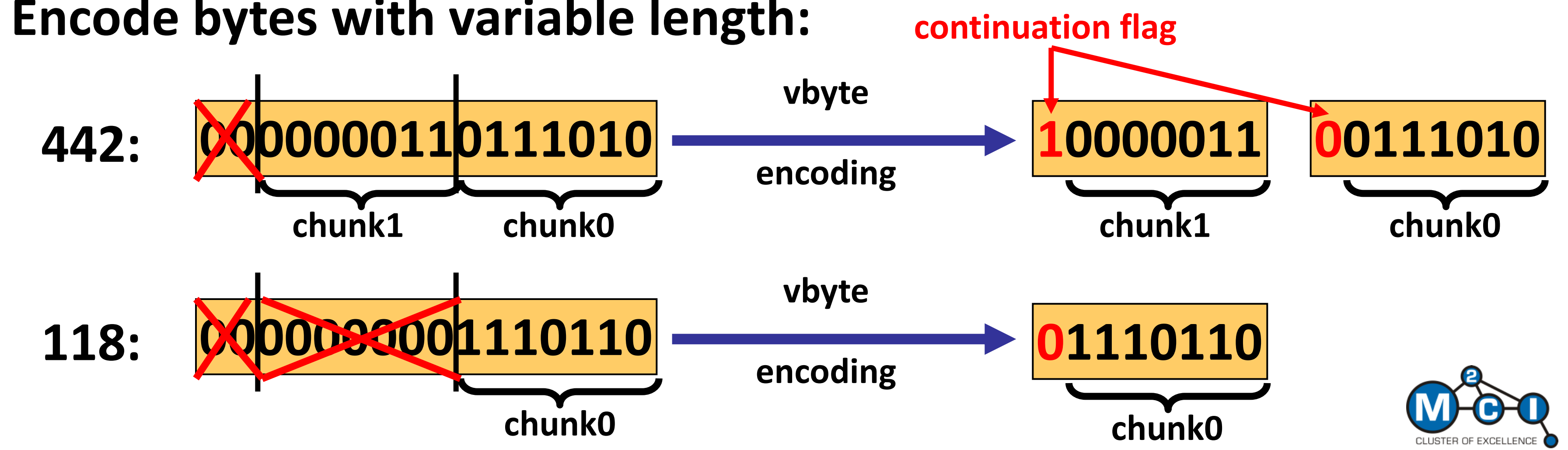
Method that saves b bits per posting must decompress posting in ≤ b ns

Rules out many effective bit-based methods (Huffman Coding, gzip, γ codes, δ codes, ω codes, Golomb/Rice codes, …)

[BCC-6]

Common method: Δ encoding & vbytes





Outline

- Efficient Query Processing
  - Introduction
  - **Basic Top-k Algorithms**
  - Scheduling 1x1
  - Approximation Algorithms
  - Non-Traditional Top-K Processing
- Efficient Precomputation
- Efficient Distributed Query Processing

Three main Classes of Algorithms

- Term-at-a-time (**TAAT**)
- Document-at-a-time (**DAAT**)
- Score-at-a-time (**SAAT**)

Term-at-a-Time Processing

- Lists sorted by d (technically, by d's unique ID)
- Lists read one after the other
- Partial scores of results maintained
- Implemented by 2-way merge join with skipping
- Results sorted by score to get top-k results

increasing ID

D1: 0.15
D2: 0.51
D3: 0.01
D4: 0.77
D8: 0.99

Pros:
- Independent of payload
- Best for Boolean queries

Cons:
- Needs to load and consider complete lists
- Requires |D| intermediate variables
- Requires sorting (|D|·log |D| time)

[Anh et al., CIKM 06]

CLUSTER OF EXCELLENCE

Conjunctive Term-at-a-time Processing

List 1

D1: 0.15
D2: 0.51
D3: 0.01
D4: 0.77
D8: 0.99

→

RSet 1

D1: 0.15
D2: 0.51
D3: 0.01
D4: 0.77
D8: 0.99

|x|

D1: 0.31
D4: 0.02
D7: 0.41
D9: 0.63

→

RSet 2

D1: 0.46
D4: 0.79

|x|

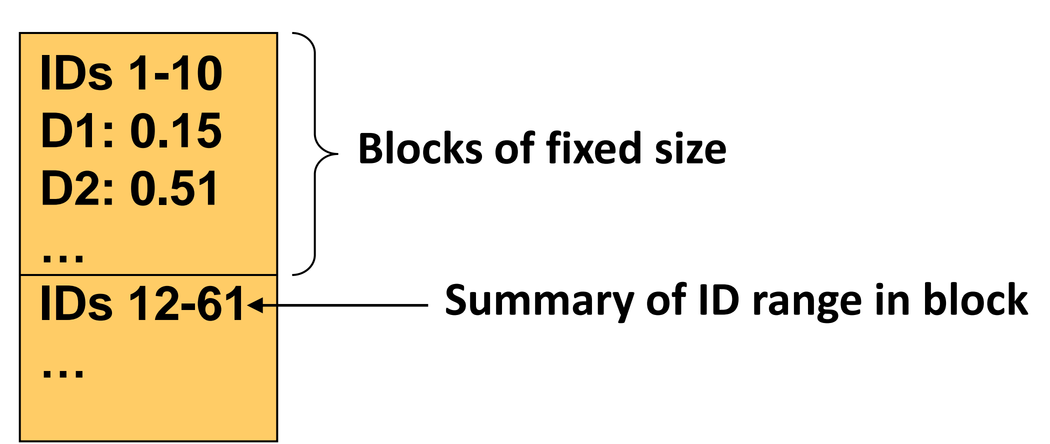
D1: 0.05
D2: 0.71
D3: 0.88
D5: 0.16

↓

RSet 3

D1: 0.51

Even more skipping with block-structured lists:



Layered (or Impact-Ordered) Indexes

List 1

4 Layers of List 1

4 Layers of List 2

sort by descending score

max=31.7

max=20.5

max=12.8

max=7.2

sort by descending doc

max=25.3

max=10.5

max=4.3

max=2.2

Current top-k results
+ candidates

disjunctive query with both lists

[Strohman & Croft, SIGIR 07]

same idea as [Bast et al, VLDB 06]

Document-at-a-Time Processing

- Lists sorted by d (technically, by d's unique ID)
- joined by n-way merge join
- Top-k results computed on the fly

increasing ID

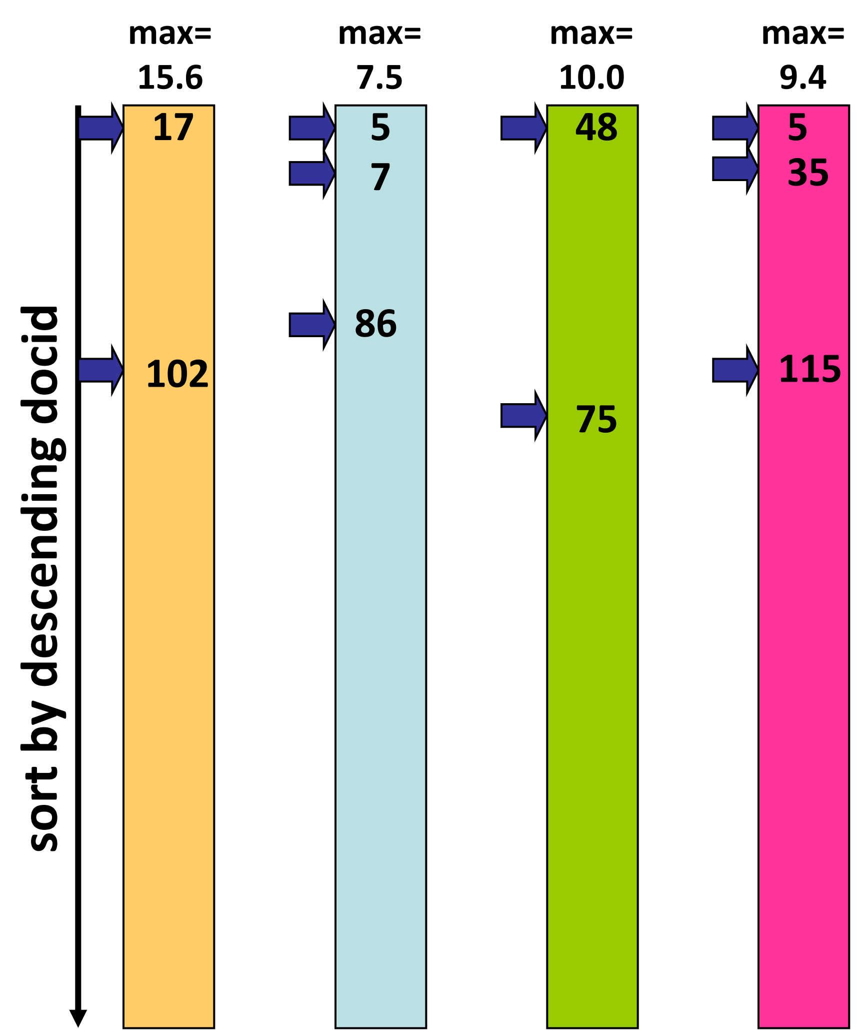
D1: 0.15
D2: 0.51
D3: 0.01
D4: 0.77
D8: 0.99

Pros:
- Can be very efficiency implemented
- Simple data structures
- Independent of payload
- Requires k intermediate variables

Cons:
- Needs to load and consider complete lists

[Anh et al., CIKM 06]

Efficient DAAT: WAND

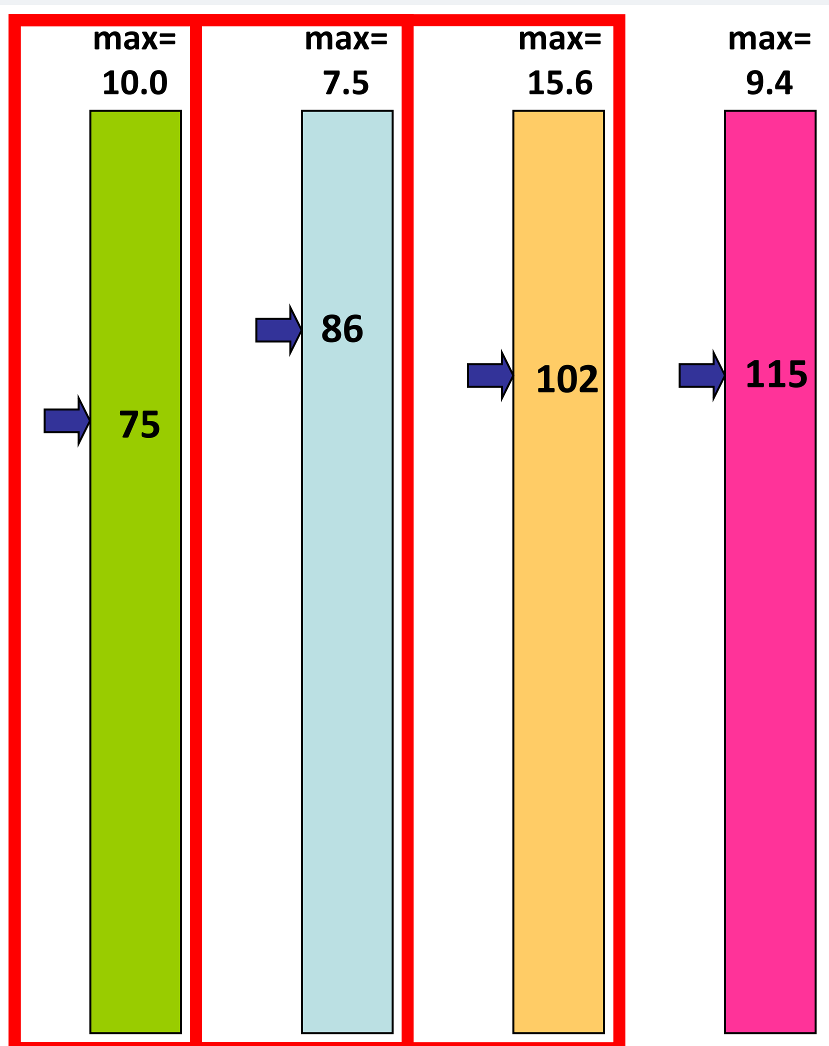


current top-1

doc 42: 15.7
doc 5: 26.6

[Broder et al., CIKM 03]

Efficient DAAT: WAND



current top-1

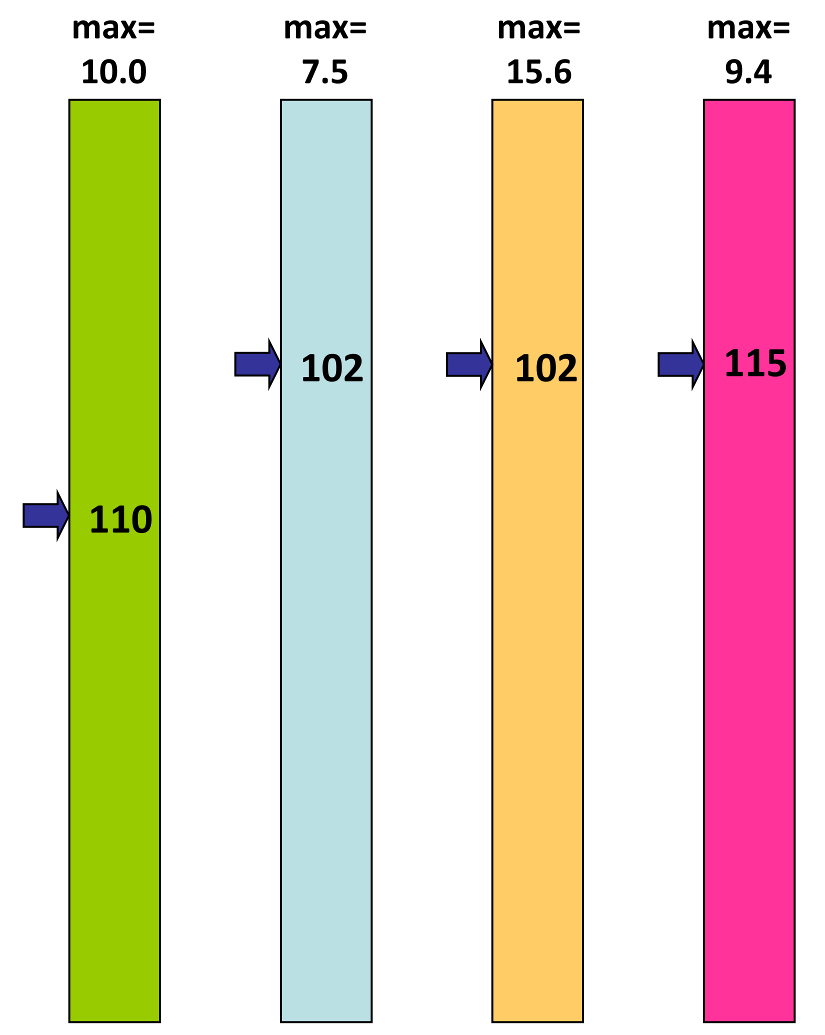
doc 42: 26.6

max=

Select enough lists to improve on current top-1 score

[Broder et al., CIKM 03]

Efficient DAAT: WAND



current top-1

doc 42: 26.6

Score this document, replace top-1 if possible, resort lists, …

Improvement:
consider per-list blocks & per-block max score [Ding&Suel, SIGIR 11]

[Broder et al., CIKM 03]

Score-at-a-Time Processing

Goal:

Avoid reading of complete lists (millions of entries)

Observation:

„Good“ results have high scores

⇒Order lists by descending scores

⇒Have „intelligent“ algorithm with early stopping

[Anh et al., CIKM 06]

List Access Modes

Factors of disk access cost:

Seek time, rotational delay, transfer time

- **Sequential** (sorted)
  - Access tuples in list order
  - Seek time & rotational delay amortized over many accesses
- **Random**
  - Look up list entry for specific item
  - Pay full cost (plus lookup cost for tuple position) for each access
  - 10-1000 times more expensive than sequential acc.

Family of Threshold Algorithms

- State-of-the-art algorithm for top-k processing
- Independently developed by different groups:
  - Fagin et al. [Fagin03]
  - Güntzer et al. [Güntzer01]
  - Nepal et al. [Nepal99]

Sorted-Access-Only (NRA) Baseline

- Interleaved scans of index lists (round-robin)
- Maintain current high score bound $high_i$ for list i
- Maintain, for each seen item d:
  - dimensions $E(d)$ where d has been seen
  - worstscore(d), bestscore(d): score bounds for d

  Updated whenever d is seen or $high_i$ changes, $i \notin E(d)$
- k items with best worstscores are current top-k; smallest worstscore in top-k: mink
- Prune item d whenever $\sum_{i \in E(d)} s_i(d) + \sum_{i \notin E(d)} high_i \le \text{mink}$
- Stop when no candidates left and $\sum high_i \le \text{mink}$ ("virtual document check")

[Fagin et al., JCSS 03], [Güntzer et al., ITCC 01], [Nepal et al., ICDE 99]

Example: Top-1 for 2-term query

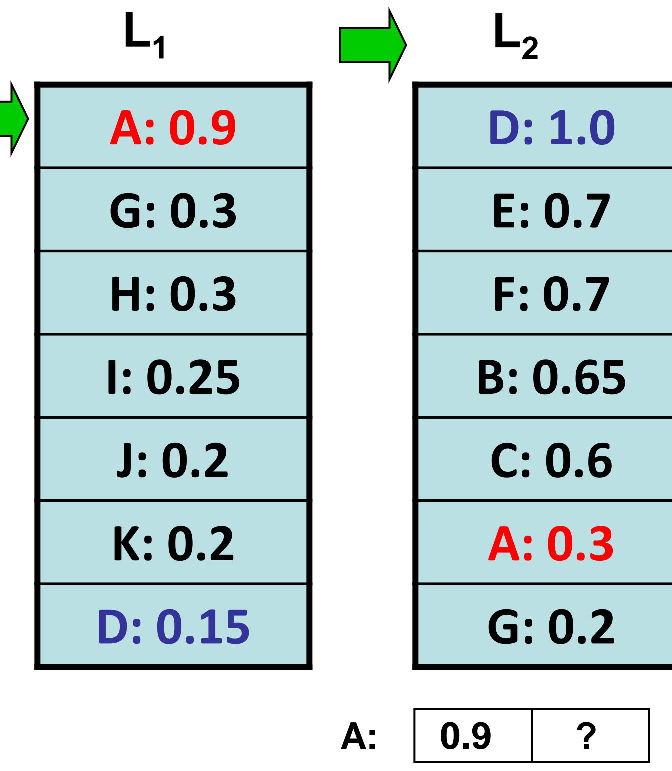
L_1	L_2
A: 0.9	D: 1.0
G: 0.3	E: 0.7
H: 0.3	F: 0.7
I: 0.25	B: 0.65
J: 0.2	C: 0.6
K: 0.2	A: 0.3
D: 0.15	G: 0.2

top-1 item

min-k:

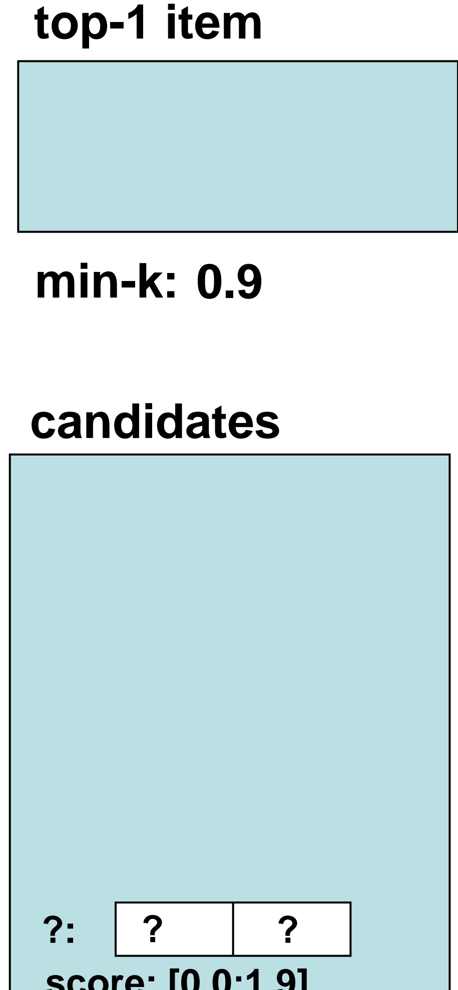
candidates

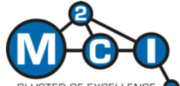
Example: Top-1 for 2-term query



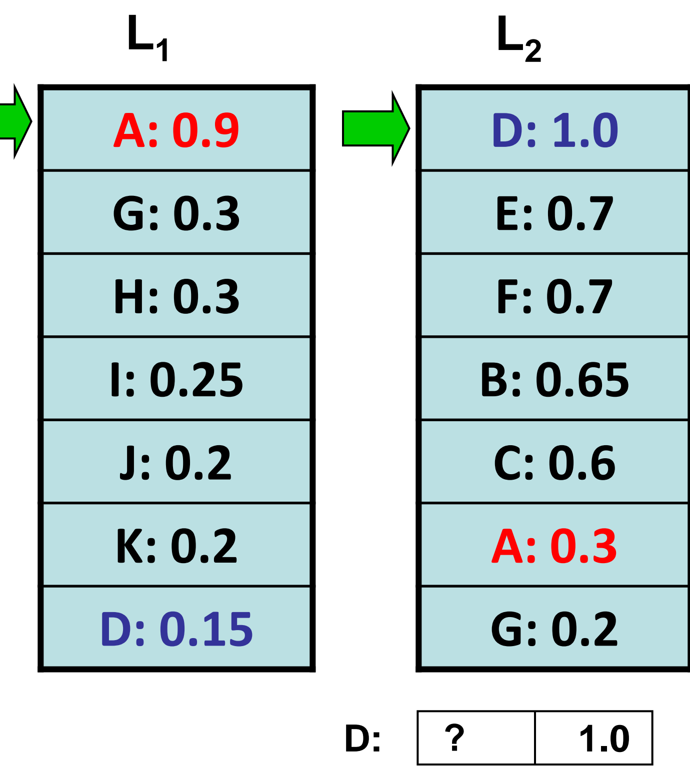
A: 0.9 | ?

score: [0.9;1.9]





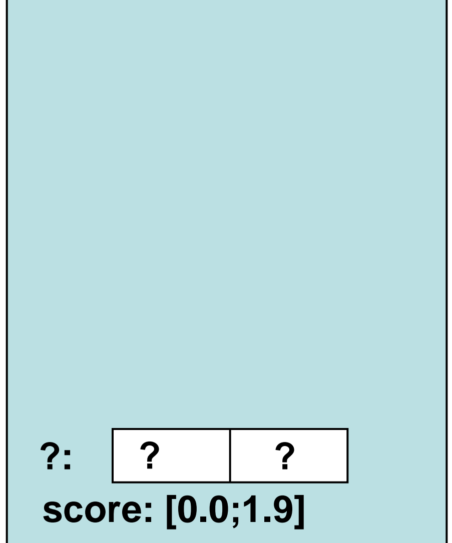
Example: Top-1 for 2-term query



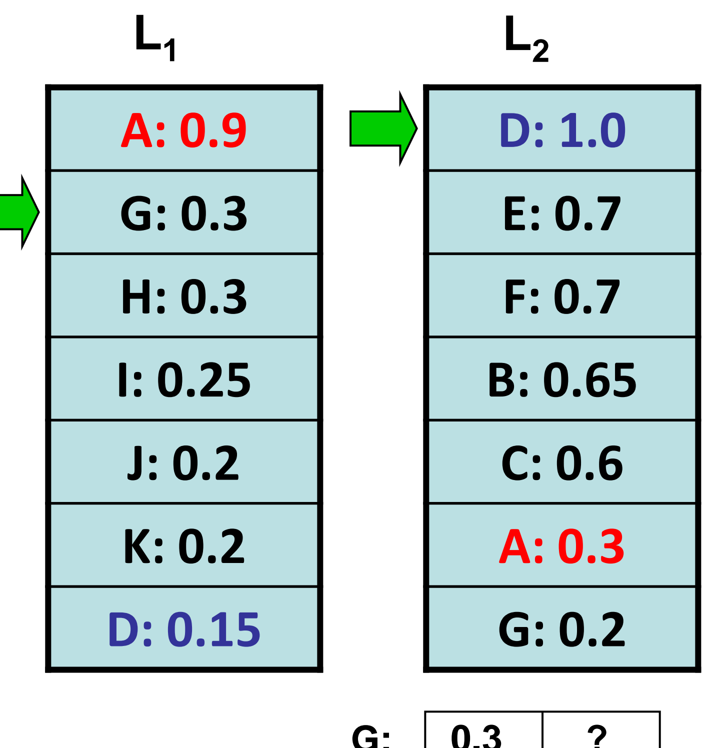
A: 0.9 ?
score: [0.9;1.9]

min-k: 1.0

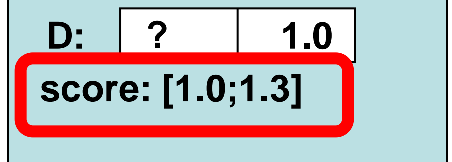
candidates



Example: Top-1 for 2-term query

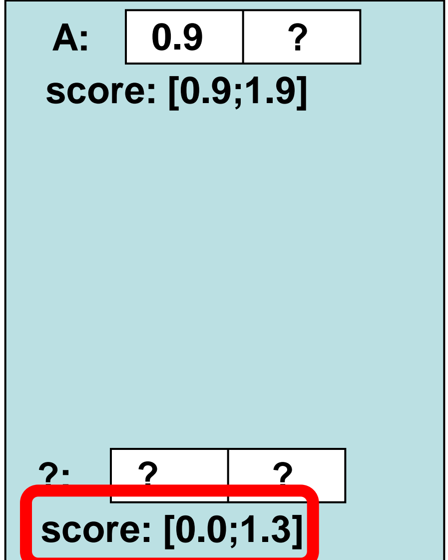


G: 0.3 ?
score: [0.3;1.3]

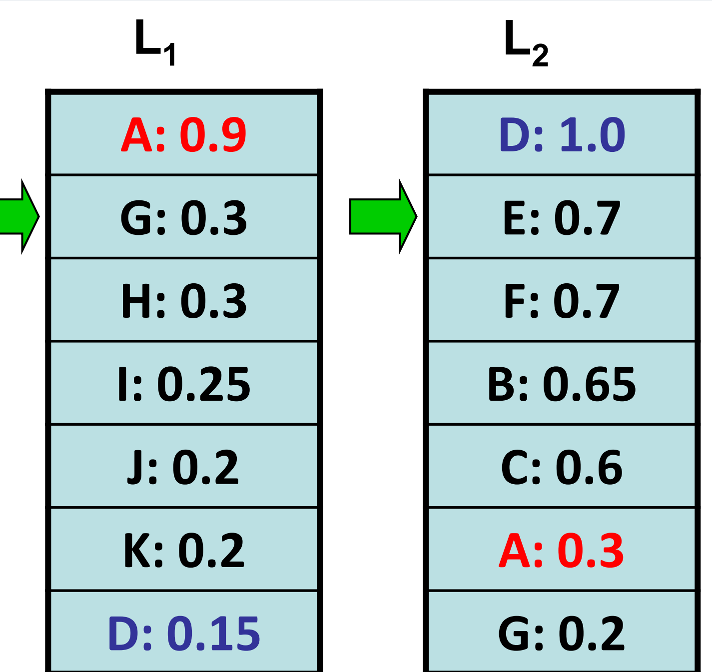


min-k: 1.0

candidates



Example: Top-1 for 2-term query

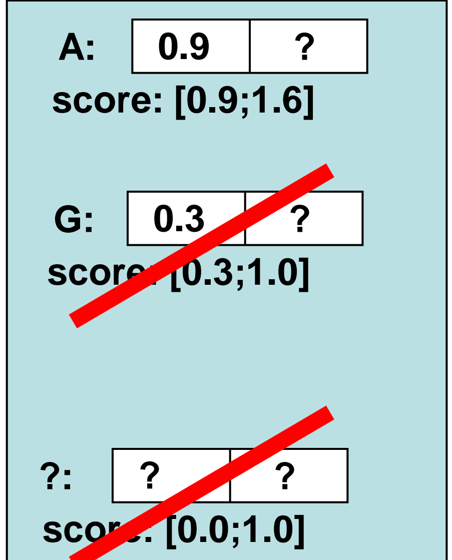


top-1 item

D: ? 1.0
score: [1.0;1.3]

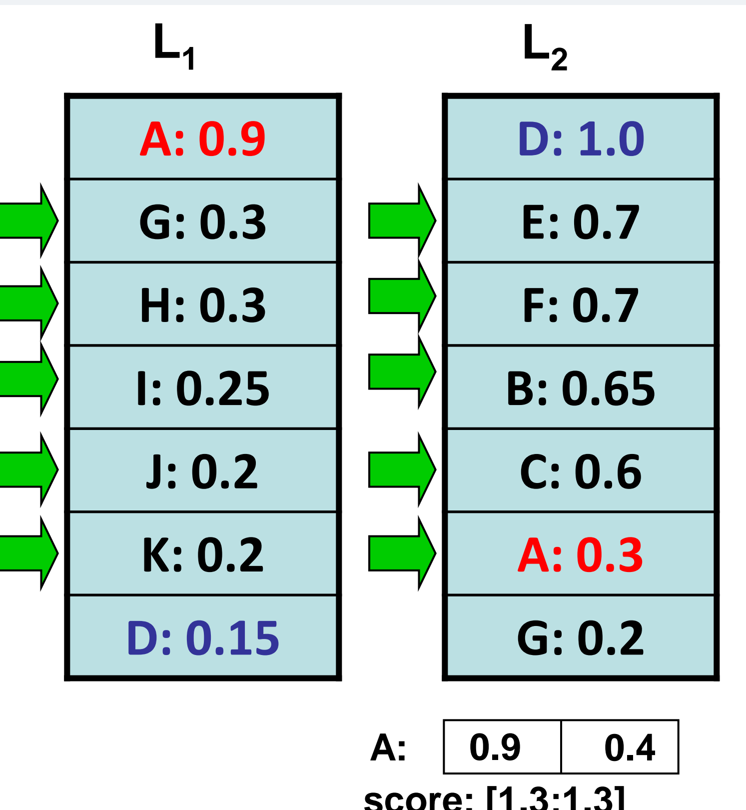
min-k: 1.0

candidates



No more new candidates considered

Example: Top-1 for 2-term query

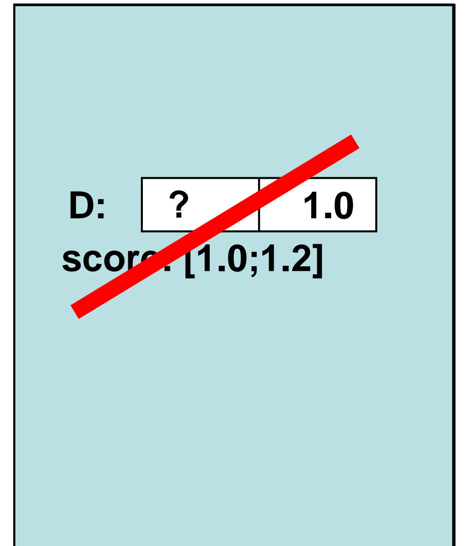


A: 0.9 | 0.4
score: [1.3;1.3]

top-1 item

min-k: 1.3

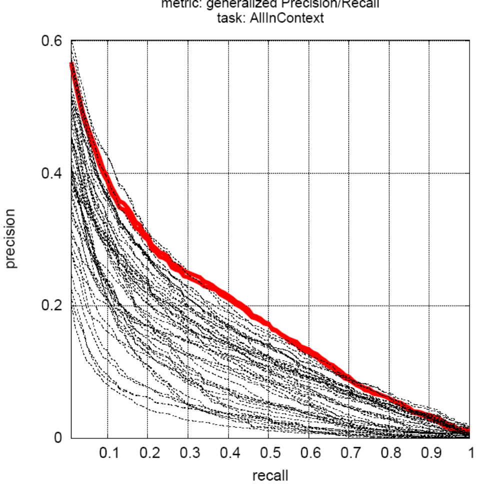
candidates



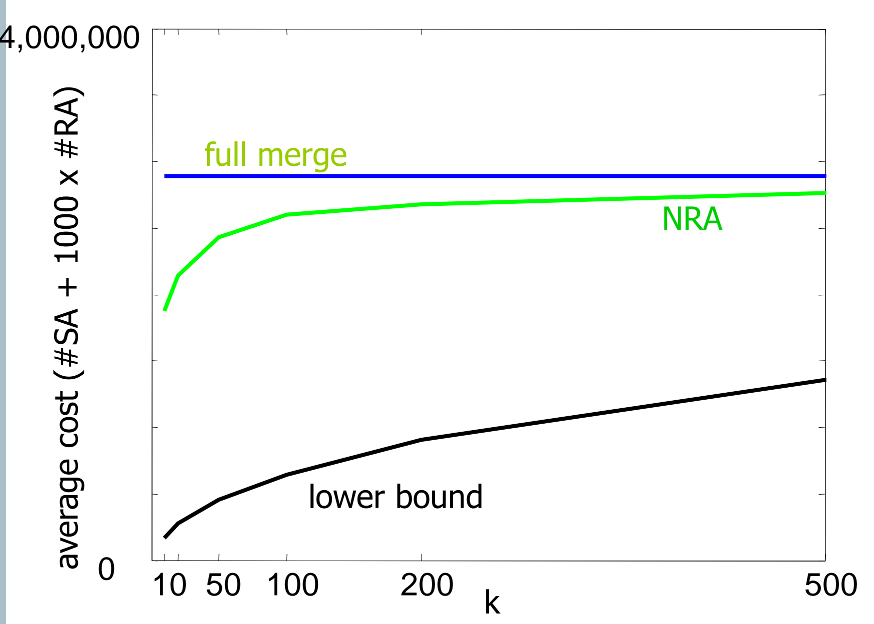
Algorithm safely terminates after 12 SA

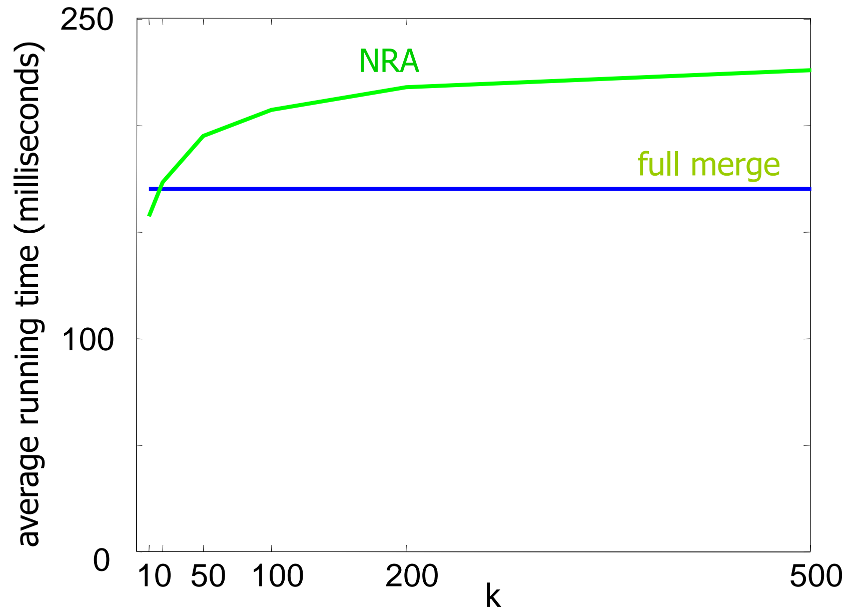
Background: TREC Benchmark Collection

- TREC Terabyte collection:
  ~24 million docs from .gov domain,
  ~420GB (unpacked) size
- 200 keyword topics from TREC Terabyte 2004/5
- Quality measures:
  - Precision at several cutoffs
  - Mean average precision (MAP)
- Performance measures:
  - Number of (sequential, random) accesses
  - Weighted cost C(factor)= #SA + factor · #RA
  - Wall-clock answer time



Experiments: NRA





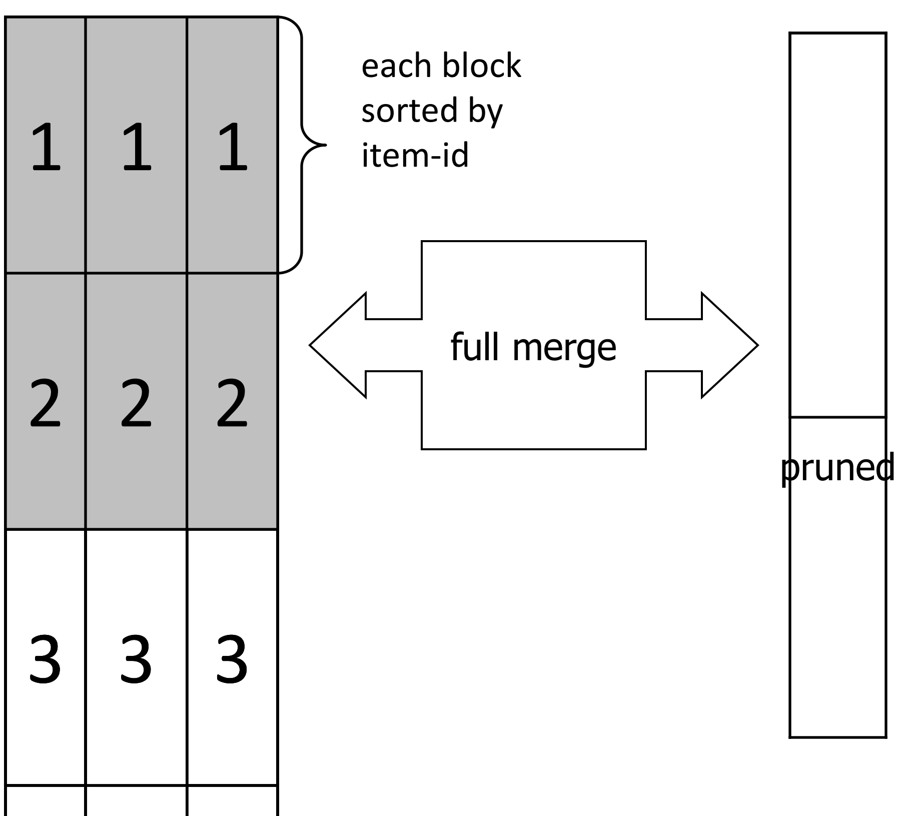
Improving Sorted Access

- Reduce overhead:
  - Prune candidates not after every step, but after a batch of steps (100-10000)
- Improve List Structure
- Improve List Selection

[Bast et al., VLDB 06]

Inverted block-index

Lists are first sorted by score



Top-k algorithm with block-index

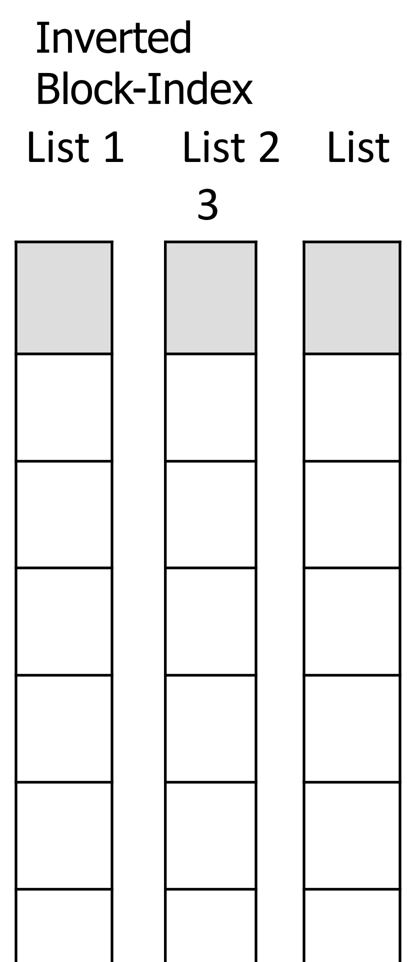
full-merge

pruned

blocks are sorted by item ids, efficiently merged by full-merge!

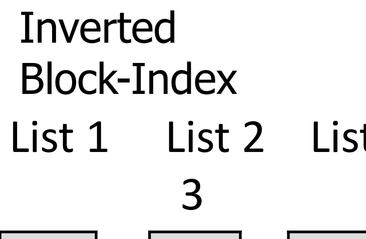
and so on…

Sorted access scheduling



General Paradigm

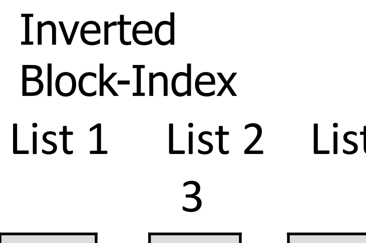
Sorted access scheduling



General Paradigm

- We assign benefits to every block of each list
- Optimization problem
  - Goal: choose a total of 3 blocks from any of the lists such that the total benefit is maximized
  - This problem is NP-Hard, the well known Knapsack problem reduces to it
  - But, since the number of blocks to choose and number of lists to choose from are very small, we can solve it exactly by enumerating all possibilities
  - We choose the schedule with maximum benefit, and continue to next round

Sorted access scheduling

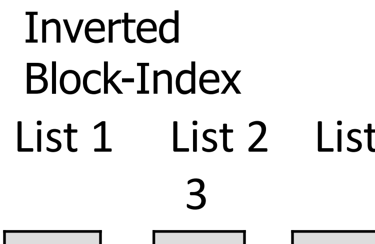


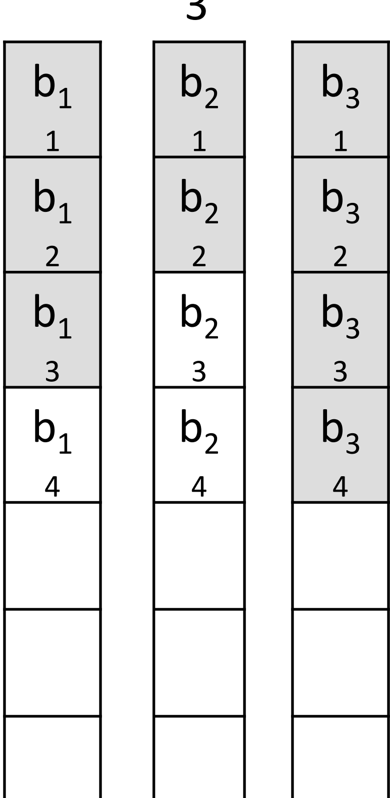
List 1	List 2	List 3
$b_{1,1}$	$b_{2,1}$	$b_{3,1}$
$b_{1,2}$	$b_{2,2}$	$b_{3,2}$
$b_{1,3}$	$b_{2,3}$	$b_{3,3}$
$b_{1,4}$	$b_{2,4}$	$b_{3,4}$

General Paradigm

- We assign benefits to every block of each list
- Optimization problem
  - Goal: choose a total of 3 blocks from any of the lists such that the total benefit is maximized
  - This problem is NP-Hard, the well known Knapsack problem reduces to it
  - But, since the number of blocks to choose and number of lists to choose from are very small, we can solve it exactly by enumerating all possibilities
  - We choose the schedule with maximum benefit, and continue to next round

Sorted access scheduling

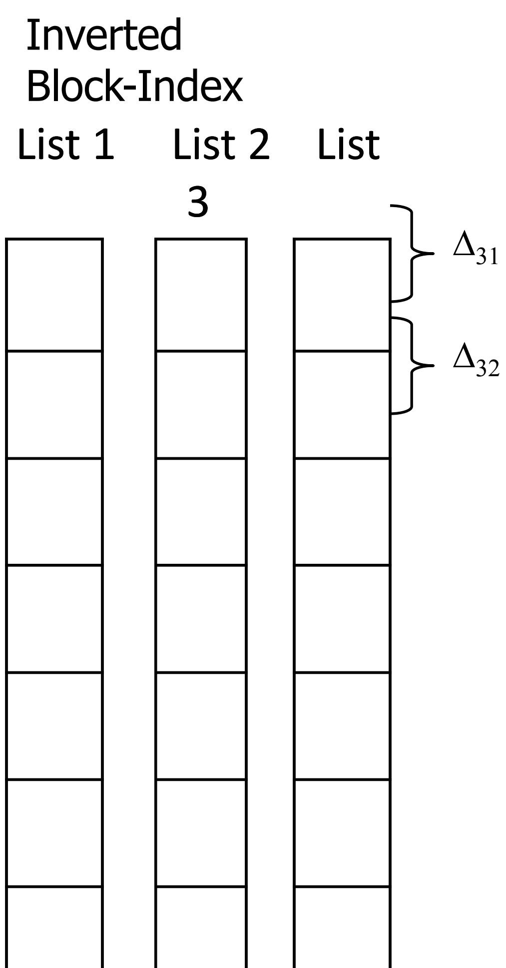




General Paradigm

- We assign benefits to every block of each list
- Optimization problem
  - Goal: choose a total of 3 blocks from any of the lists such that the total benefit is maximized
  - This problem is NP-Hard, the well known Knapsack problem reduces to it
  - But, since the number of blocks to choose and number of lists to choose from are very small, we can solve it exactly by enumerating all possibilities
  - We choose the schedule with maximum benefit, and continue to next round

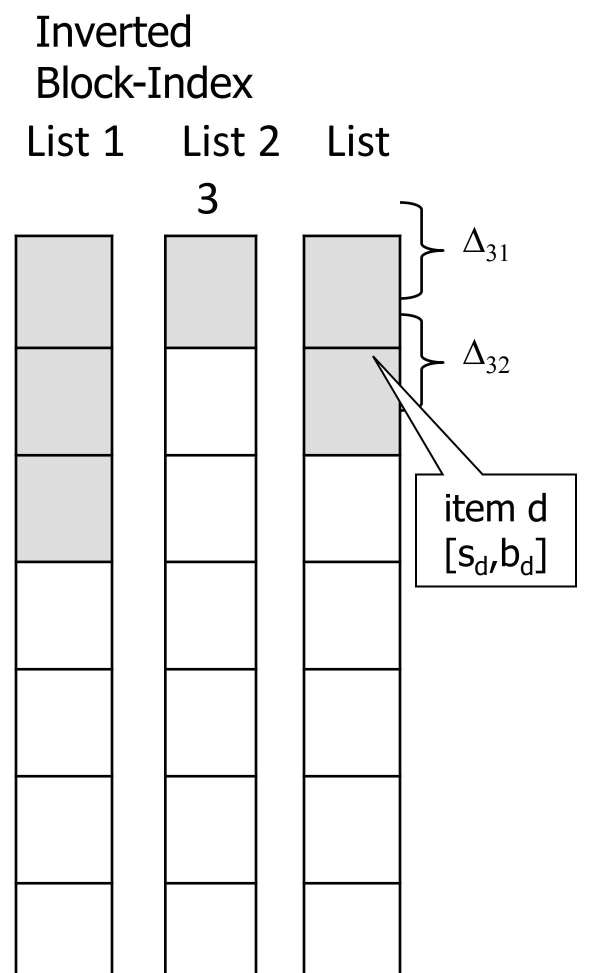
Sorted access scheduling



Knapsack for Score Reduction (KSR)

- Pre-compute score reduction Δ_{ij} of every block of each list : (max-score of the block – min-score of the block)

Sorted access scheduling



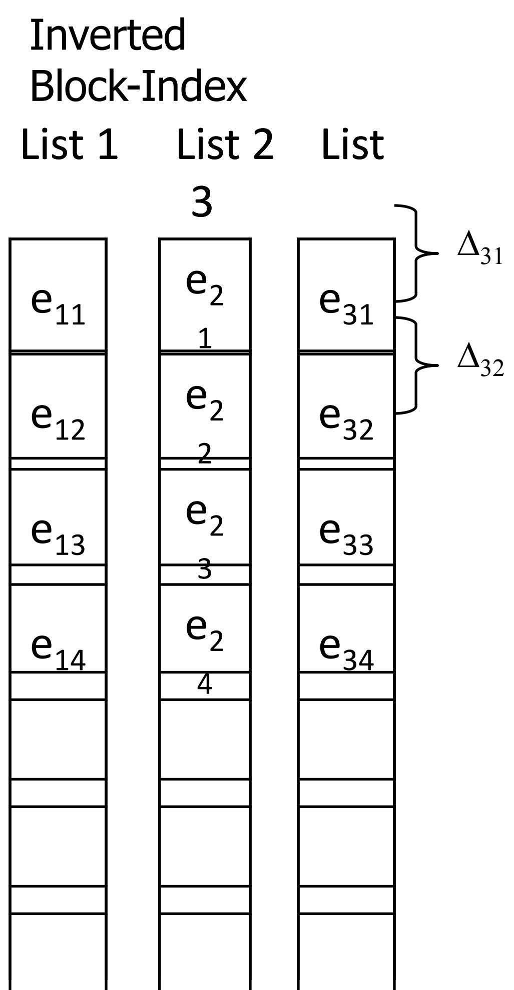
Knapsack for Score Reduction (KSR)

- Pre-compute score reduction Δ_{ij} of every block of each list : (max-score of the block – min-score of the block)
- Candidate item d is already seen in list 3. If we scan list 3 further, score s_d and best-score b_d of d do not change
- In list 2, d is not yet seen. If we scan one block from list 2
  - with high probability d will not be not found in that block: best-score b_d of d decreases by Δ_{22}
- Benefit of block B in list i

$\sum_d \Delta_B (1 - Pr[d \text{ found in } B])$ » $\sum_d \Delta_B$

sum taken over all candidates d not yet seen in list i

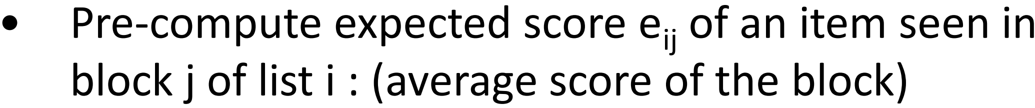
Sorted access scheduling



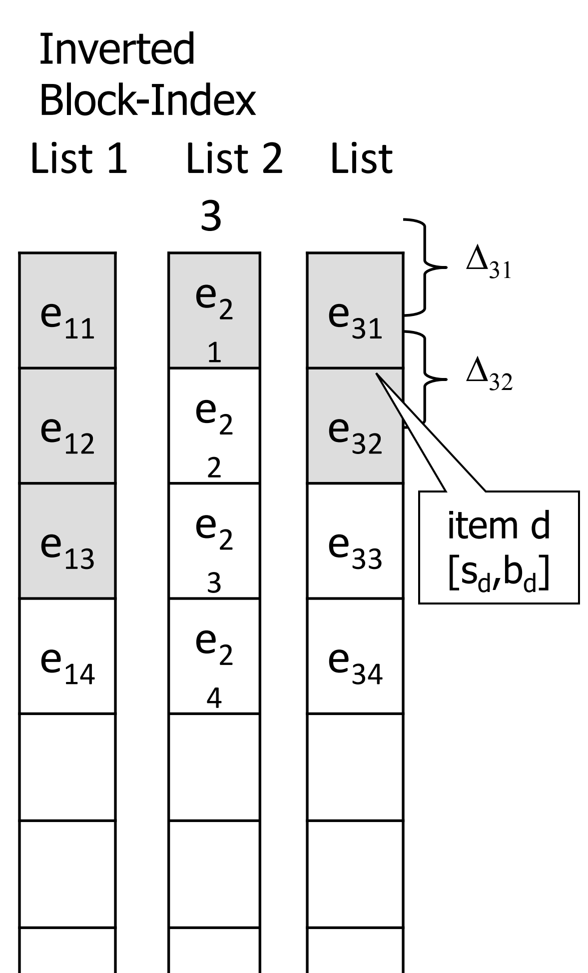
Knapsack for Benefit Aggregation (KBA)

- Pre-compute expected score e_{ij} of an item seen in block j of list i : (average score of the block)
- Pre-compute score reduction Δ_{ij} of every block of each list : (max-score of the block – min-score of the block)

Sorted access scheduling



Knapsack for Benefit Aggregation (KBA)



- Pre-compute expected score e_{ij} of an item seen in block j of list i : (average score of the block)
- Pre-compute score reduction Δ_{ij} of every block of each list : (max-score of the block – min-score of the block)
- Candidate item d is already seen in list 3. If we scan list 3 further, score s_d and best-score b_d of d do not change
- In list 2, d is not yet seen. If we scan one block from list 2
  - either d is found in that block: score s_d of d increases, expected increase = e_{22}
  - or d is not found in that block: best-score b_d of d decreases by Δ_{22}
- Benefit of block B in list i

$$\Sigma_d\, e_B \Pr[d \text{ found in } B] + \Delta_B (1 - \Pr[d \text{ found in } B])$$

The sum is taken over all candidates d not yet seen in list i

Random Accesses

Two main purposes for random accesses:

- Can speed up execution
- Some predicates cannot be read from sorted lists („X and not Y“) => expensive predicates

Scheduling problem:

- **When** perform RA for **which** item to **which** list?

Random Access Scheduling – When

- **Immediately** when an item is seen (TA)
  - Scores always correct
  - No need for score bounds & candidates
  - Most RA are wasted (items seen again later)
  - Really slow if RA are expensive
- **Balanced**: after C sorted accesses, do 1 RA (Combined Algorithm, CA)
  - Faster than TA
  - Most RA are still wasted

Random Access Scheduling – When

- **LAST heuristics:** switch from SA to RA when
  - All possible candidates have been seen
  - expected future cost for RA is below the cost already spent for SA
    - Cost spent for SA: known by bookkeeping
    - (simplified) cost expected for RA:

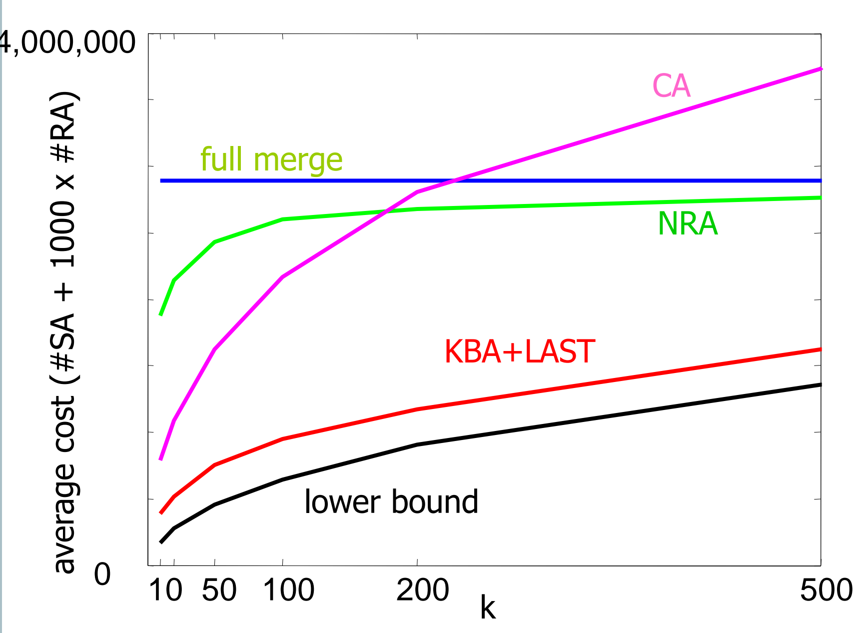
$$\sum_{d} \sum_{i \notin E(d)} C$$

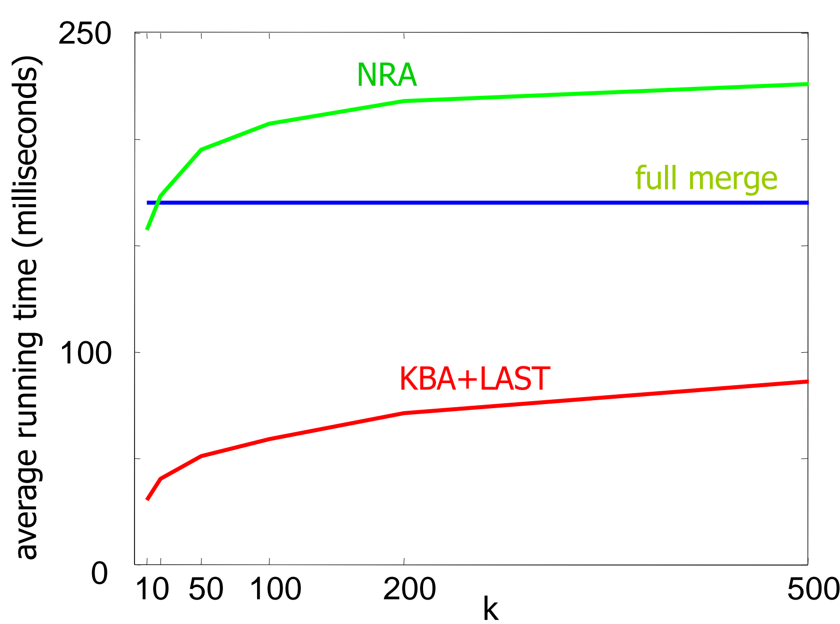
Rationale behind this:
Do expensive RA as late as possible
to avoid wasting them

Experiments: TREC

TREC Terabyte benchmark collection

- over 25 million documents, 426 GB raw data
- 50 queries from TREC 2005 adhoc task





Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - **Approximation Algorithms**
  - Non-Traditional Top-K Processing
- Efficient Precomputation
- Efficient Distributed Query Processing

Rationale for Approximation Algorithms

- Scoring functions are (well-founded) heuristics, not the gold standard
- Users don't care about the exact top-k results, but about **relevant** results
- Many relevant results beyond the top-k
- Often one relevant result is enough

Threshold algorithms may be overly conservative

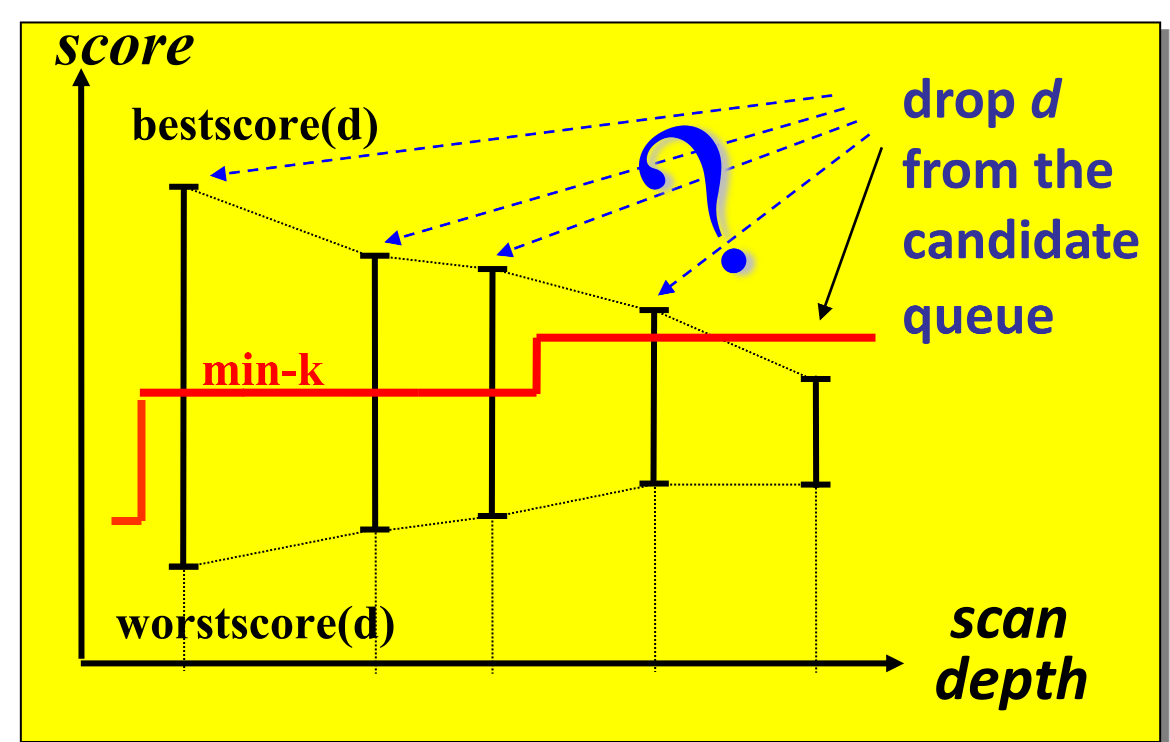
Evolution of a Candidate's Score

TA family of algorithms based on invariant (with sum as aggr)

$$\underbrace{\sum_{i \in E(d)} s_i(d)}_{\text{worstscore(d)}} \le s(d) \le \underbrace{\sum_{i \in E(d)} s_i(d) + \sum_{i \notin E(d)} high_i}_{\text{bestscore(d)}}$$

- Worst- and best-scores slowly converge to final score
- Add *d* to top-k result, if *worstscore(d) > min-k*
- Drop *d* only if *bestscore(d) < min-k*, otherwise keep it in candidate queue

→ **Overly conservative threshold & long sequential index scans**



- **Approximate top-k**
  "What is the **probability** that *d* qualifies for the top-k ?"

Probabilistic Guarantees

TA family of algorithms based on invariant (with sum as aggr)

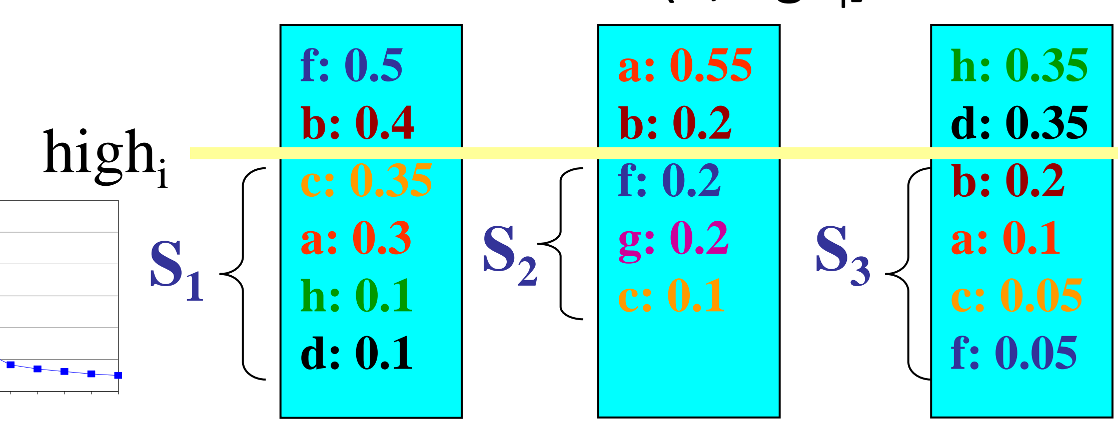
$$\sum_{i \in E(d)} s_i(d) \leq s(d) \leq \sum_{i \in E(d)} s_i(d) + \sum_{i \notin E(d)} high_i$$

Relaxed into probabilistic invariant

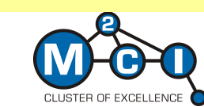
$$p(d) := P[s(d) > \delta] = P[\sum_{i \in E(d)} s_i(d) + \sum_{i \notin E(d)} S_i > threshold]$$

$$= P[\sum_{i \notin E(d)} S_i > threshold - \sum_{i \in E(d)} s_i(d)] =: P[\sum_{i \notin E(d)} S_i > \delta'] \leq \varepsilon$$

where the random variable S_i has some (postulated and/or estimated) distribution in the interval (0,$high_i$]

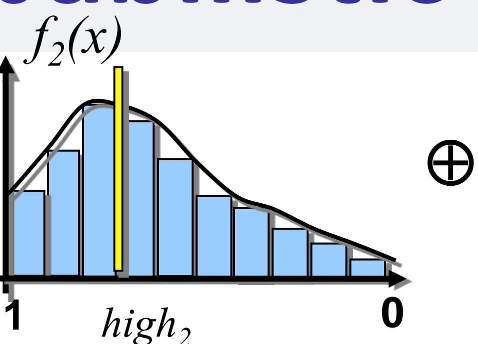


- *Discard candidates with p(d) ≤ ε*
- Exit index scan when candidate list empty

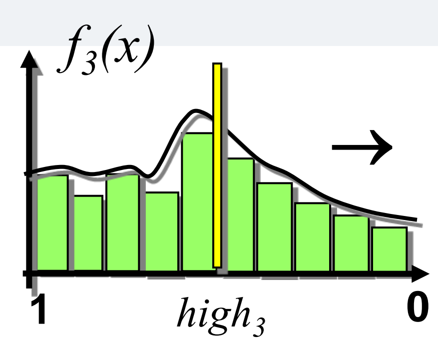


Probabilistic Threshold Test

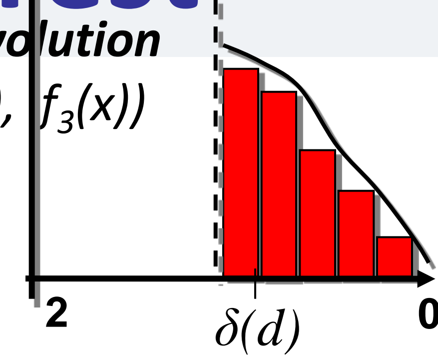
cand item d with
$2 \notin E(d)$,
$3 \notin E(d)$



⊕



→



- fitting *Poisson* distribution (or Poisson mixture)
  - over equidistant values:
  - easy and exact convolution

$$P[d = v_j] = e^{-\alpha_i} \frac{\alpha_i^{\,j-1}}{(j-1)!}$$

- distribution approximated by *histograms:*
  - precomputed for each dimension
  - dynamic convolution at query-execution time

engineering-wise histograms work best!

with *independent* Si's or with *correlated* Si's

Probabilistic Guarantees:
E[relative precision @ k] = 1-ε
E[relative recall @ k] = 1-ε

Results for .Gov Queries

on .GOV corpus from TREC-12 Web track:

1.25 Mio. docs (html, pdf, etc.)

50 keyword queries, e.g.:

- *„Lewis Clark expedition“,*
- *„juvenile delinquency“,*
- *„legalization Marihuana“,*
- *„air bag safety reducing injuries death facts“*

	NRA	Prob-Top-k
#sorted accesses	2,263,652	527,980
elapsed time [s]	148.7	15.9
max queue size	10849	400
relative recall	1	0.69
rank distance	0	39.5
score error	0	0.031

.Gov Expanded Queries

on .GOV corpus with query expansion based on WordNet synonyms:

50 keyword queries, e.g.:

- **„juvenile delinquency *youth minor crime law jurisdiction offense prevention*“,**
- **„legalization marijuana *cannabis drug soft leaves plant smoked chewed euphoric abuse substance possession control pot grass dope weed smoke*“**

	NRA	Prob-Top-k
#sorted accesses	22,403,490	18,287,636
elapsed time [s]	7908	1066
max queue size	70896	400
relative recall	1	0.88
rank distance	0	14.5
score error	0	0.035

Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - **Non-Traditional Top-K Processing**
    - **Query Expansion**
    - Proximity-Aware Retrieval
    - Top-k with Constrained Budget
- Efficient Precomputation
- Efficient Distributed Query Processing

Including Term Expansion

Problem: Users use different terms for similar things

$\Rightarrow$ poor recall (missing relevant results)

Example:
MPI, MPII, MPI-INF, MPI-CS, Max-Planck-Institut, D5, AG5, DB&IS, MMCI, UdS, Saarland University, …

Solution:

1. Define notion of **similar terms**
2. **Expand** queries with similar terms
3. **Modify** scoring function for expanded queries

[Theobald et al., SIGIR 05]

Heuristics for finding similar terms

Co-Occurrence heuristics:

Terms t_1 and t_2 ***similar*** if they occur (almost) always together

$$sim(t_1, t_2) = \frac{2 \cdot |docs(t_1) \cap docs(t_2)|}{|docs(t_1)| + |docs(t_2)|}$$

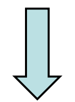
Specialization heuristics:

Term t_2 ***specialization*** of t_1 if t_1 occurs (almost) whenever t_2 occurs

$$sim(t_1, t_2) = P[t_1 | t_2] = \frac{|docs(t_1) \cap docs(t_2)|}{|docs(t_2)|}$$

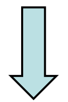
Ontology-Based Query Expansion

Similarity conditions like
~Professor ~course ~IR



disambiguation

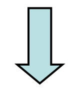
Query expansion



δ-exp(x)={w | sim(x,w)>δ}

Weighted expanded query

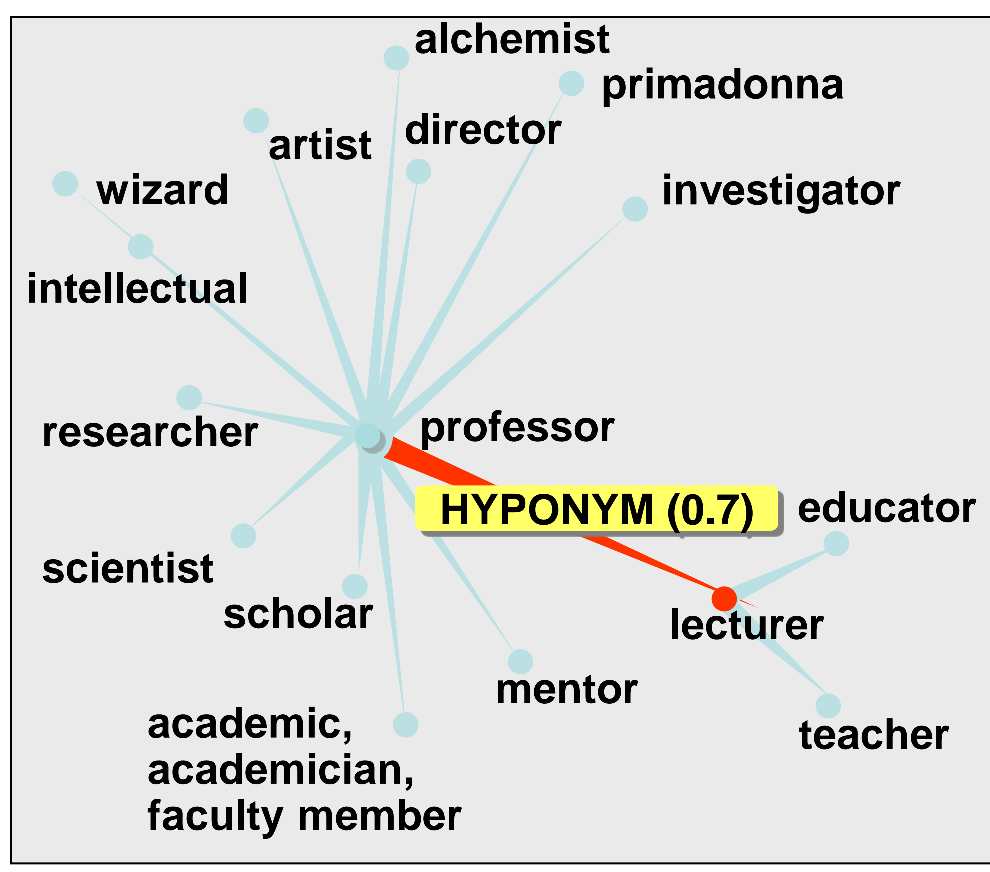
Example:
(professor lecturer(0.7) scholar(0.6) ...)
(course class(1.0) seminar(0.84) ...)
(„IR" „Web search" (0.653) ...)



better recall, but possibly worse precision (due to topic drift)

Efficient top-k search with dynamic expansion

Thesaurus/Ontology:
concepts, relationships, glosses from WordNet, Gazetteers, Web forms & tables, Wikipedia



relationships quantified by statistical correlation measures

Scoring Expanded Queries

Naive approach:

For query term t, add similar terms t‘ with sim(t,t‘)>δ to query

But:

„transport ... ane ...“

Result quality drops due to topic drift

Better: auto-tuning incremental expansion [SIGIR’05]

For query term t, consider only expansion with

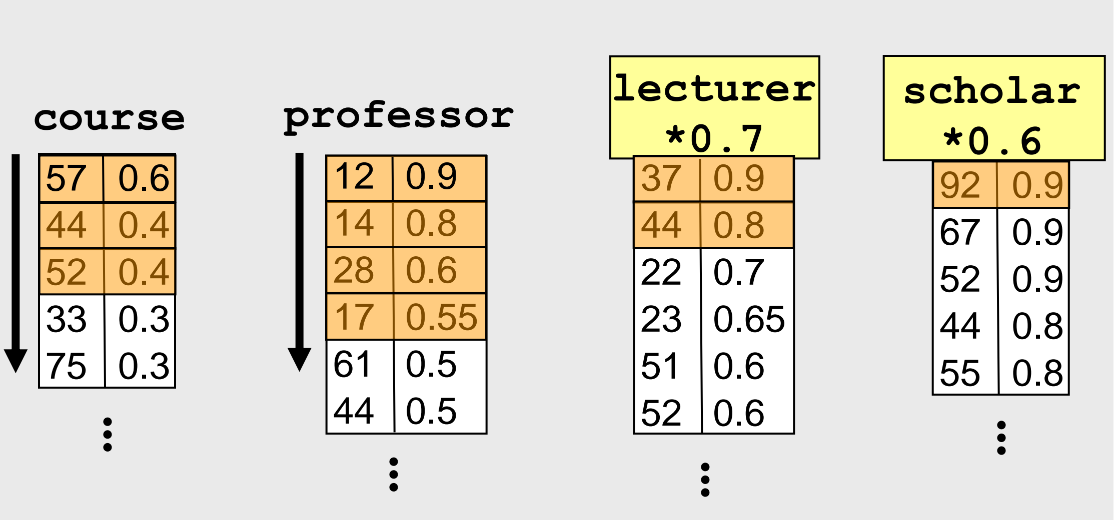
highest combined score per item

$$\bar{s}_t(i) = \max_{t' \in T} sim(t,t') \cdot s_{t'}(i)$$

Incremental Query Expansion

Consider expandable content condition **Professor**
with score $\max_{t \in T} \{ sim(Professor,t)*s_t(i) \}$

Dynamic query expansion with incremental, on-demand merging of additional index lists



+ much more efficient than threshold-based expansion
+ no threshold tuning
+ better recall, no topic drift

Effectiveness of ~~Incremental Expansion~~ Approximation

Rprec, title-only
Rprec, static expansion, T&D&WN
Rprec, incr. merge, T&D&WN
P@10, title-only
P@10, static expansion, T&D&WN
P@10, incr. merge, T&D&WN
MAP, title-only
MAP, static expansion, T&D&WN
MAP, incr. merge, T&D&WN

1.0 0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1 0.0

0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 ε

Figure 5: Precision as a function of ϵ - incremental merge vs. static expansion

Efficiency of Incremental Expansion

50,000,000
40,000,000
30,000,000
20,000,000
10,000,000
0

Sorted IOs, static expansion, ε = 0.0
Sorted IOs, incr. merge, ε = 0.0
Sorted IOs, incr. merge, ε = 0.1
Random IOs, static expansion, ε = 0.0
Random IOs, incr. merge , ε = 0.0
Random IOs, incr. merge , ε = 0.1

1.00 0.30 0.10 0.03 0.01 θ

Figure 8: Efficiency as a function of θ

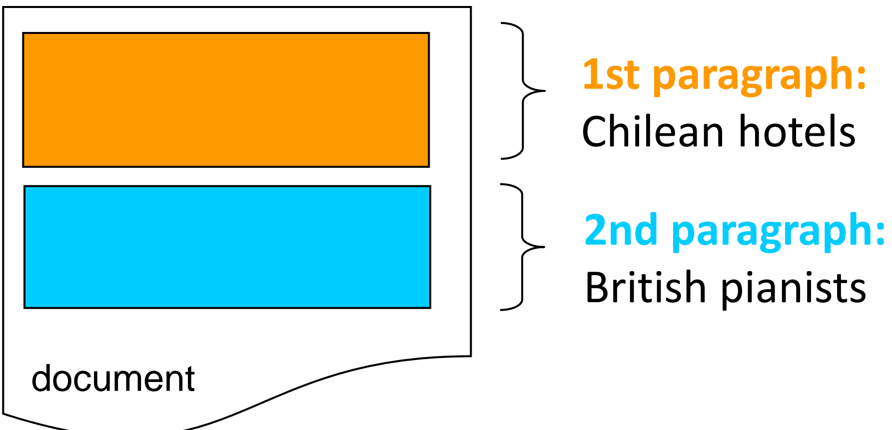
Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - **Non-Traditional Top-K Processing**
    - Query Expansion
    - **Proximity-Aware Retrieval**
    - Top-k with Constrained Budget
- Efficient Precomputation
- Efficient Distributed Query Processing

Motivation for Text Proximity Scoring

„Bag of words“ without term proximity sometimes yields unsatisfactory results

Example: query: *Chilean pianists*



1st paragraph:
Chilean hotels

2nd paragraph:
British pianists

All query terms individually important, but appear in different paragraphs.

Phrase queries can avoid such bad results.
But: prevent also many potentially good results.

[Schenkel et al., SPIRE 07], [Broschart et al., Grundlagen von Datenbanken 08]

Motivation for Text Proximity Scoring

document

1st paragraph:
Chilean hotels

Chilean hotels …
Chilean hotels usually offer …
Chilean specialty …

2nd paragraph:
British pianists

British **pianists …**
The **pianist** performed …
The live show of the **pianist** …

Idea of proximity scores:
Reward occurrences of different query terms in close proximity

What is the best proximity score?

#1 for all test beds: Büttcher et al.

P@10

0,64 0,62 0,60 0,58 0,56 0,54 0,52 0,50 0,48 0,46 0,44 0,42 0,40 0,38 0,36 0,34 0,32 0,30 0,28 0,26 0,24 0,22

TERABYTE WEB ROBUST INEX

test bed

BM25 ES Büttcher Rasolofo Tao,Zhai Song De Kretser,Moffat LM Zhao,Yun Lv,Zhai Cummins,O'Riordan

Proximity-enhanced scoring

Büttcher et al., SIGIR 2006:

linear combination of content score and

BM25-style proximity score

$$score_{Büttcher}(d,q) = \underbrace{score_{BM25}(d,q)}_{\text{content score}} + \underbrace{\sum_{t \in q} \min\{1, idf(t)\} \frac{acc(d,t) \cdot (k_1+1)}{acc(d,t)+K}}_{\text{proximity score}}$$

$$K = k_1 \cdot [(1-b) + b \cdot \frac{dl(d)}{avgdl}], \text{where } k_1 = 1.2, b = 0.5$$

$$acc(d,t) = \sum_{t \in q} \frac{idf(t')}{(pos(t')-pos(t))^2}; t' \text{ adjacent query term } \neq t \text{ in } d$$

Example: Computation of acc

It[1] took[2] the[3] **sea[4]** a[5] thousand[6] **years,[7]**
A[8] thousand[9] **years[10]** to[11] trace[12]
The[13] granite[14] features[15] of[16] this[17] **cliff,[18]**
In[19] crag[20] and[21] scarp[22] and[23] base.[24]

Query: {**sea**, **years**, **cliff**}

$$acc(d,\text{sea}) = \frac{idf(\text{years})}{(7-4)^2}$$

$$acc(d,\text{years}) = \frac{idf(\text{sea})}{(7-4)^2} + \frac{idf(\text{cliff})}{(18-10)^2}$$

$$acc(d,\text{cliff}) = \frac{idf(\text{years})}{(18-10)^2}$$

That's great, but...

Experiments on TREC collection:

approach	P@10	MAP@1000	P@10 stemmed	MAP@1000 stemmed
Büttcher	0.567	0.194183	0.602	0.232466
BM25	0.532	0.184334	0.568	0.215673

Implementation of this score in a top-k-style engine with precomputed inverted lists?

Towards an efficient implementation

Problem: acc(d,t) based on **adjacent** query terms

$$acc(d,t) = \sum_{t' \in q} \frac{idf(t')}{(pos(t') - pos(t))^2}$$; t' **adjacent** query term $\neq t$ in d

But: queries not known at index build time

=> we need a query-independent index!

Solution:

$$acc'(d,t) = \sum_{t' \in q} idf(t') \underbrace{\frac{1}{(pos(t') - pos(t))^2}}_{acc(d,t,t')}$$; t' **every** query term $\neq t$ in d

Build inverted list with $acc(d,t,t')$ for all term pairs

Document length (in K) does not fit in this framework

=> drop document length (set b=0)

Index Structures and Results

descending score

TL(hotel)	TL(california)	PXL (california, hotel)	CL (california, hotel)
d73:9.3	d68:9.1	d7:3.0	d7:(3,8.6,4.5)
d61:7.2	d7:8.6	d68:0.7	d68:(0.7,9.1,1.5)
d78:5.0	d73:5.9	d61:0.5	d61:(0.5,3.0,7.2)
d7:4.5	d8:4.6	d10:0.2	d10:(0.2,2.0,1.7)
⋮	⋮	⋮	⋮

TL lists: $d : score_{BM25}(d, t)$

PXL and CL lists: $d : (acc(d, t_i, t_j), score_{BM25}(d, t_i), score_{BM25}(d, t_j))$

	Run	k=10		k=50		k=100	
		P@k	t[ms]	P@k	t[ms]	P@k	t[ms]
TopX	TL	0,57	5.220	0,45	6.868	0,38	8.827
TopX	TL+PXL	0,61	6.266	0,48	11.127	0,40	15.524
TopX	TL+KL	0,61	821	0,48	1.651	0,40	2.042

What about the index size?

Construction of query-independent index failed (too slow!)*

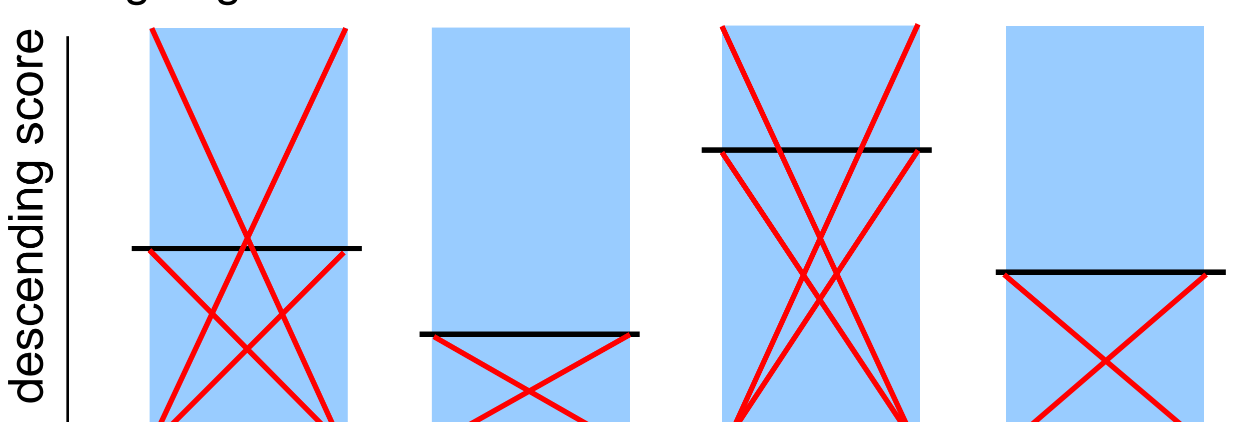
randomly sampled 1,500,000 term pairs:

1.2% nonempty proximity lists

index/limit	unpruned size(#tuples)	required space
TL	$3.191 \cdot 10^{9}$	47.5 GB
PXL/CL (estimated!)	$1.410 \cdot 10^{12}$	20.5 TB / 41.0 TB

keeping all proximity lists: infeasible

Pruning might be the solution:



horizontal pruning and

vertical pruning

*we fixed that now

Different horizontal pruning methods

- limit distance of term occurrences
- limit proximity score
- limit list size to a constant (from 500 to 3,000 tuples)
- Carmel et al. [SIGIR 2001]: static index pruning
  drop index entries having scores below ε·top-k score
- combinations (e.g., limit list size + static index pruning)

Horizontal pruning helps a lot

Index size: in million tuples (estimated)

index/limit	500	1000	1500	2000	2500	3000	unpruned
TL	295	355	402	442	472	496	3,191
PXL/CL (est.)	368,761	435,326	481,949	515,079	542,611	566,277	1,410,238
PXL/CL, score≥0.01 (est.)	23,050	28,855	34,023	38,985	42,085	45,186	87,049

Index size: in bytes (estimated)

index/limit	500	unpruned
PXL (est.)	5.4 TB	20.5 TB
CL (est.)	10.7 TB	41.0 TB
PXL, score≥0.01 (est.)	343.5 GB	1.3 TB
CL, score≥0.01 (est.)	686.9 GB	2.5 TB

Index size of (real) file-based index

Index/max. Länge	3.000	ungekürzt
TL	2,9 GB	34 GB
PXL(window≤5)	133 GB	311 GB
KL(window≤5)	266 GB	622 GB

Top-10 retrieval: unpruned vs. pruned lists

Configuration	P@10	Cost(1000)	index size(est.)
BM25(=TL)	0.56	1,956,193,840	47.5 GB
TL(2000 tuples)	0.34	9,303,808	6.6 GB
TL+CL	0.60	187,999,568	41.0 TB
TL+CL(2000 tuples)	0.60	25,971,904	15.0 TB
TL+CL(ϵ=0.025)	0.60	73,744,304	
TL+CL(ϵ=0.1)	0.60	84,484,976	n/a
TL+CL(ϵ=0.2)	0.58	105,584,992	
TL+CL(500; ϵ=0.025)	0.54	6,628,528	n/a
TL+CL(2000; ϵ=0.025)	0.60	20,377,904	
TL+CL(500; score$\geq$0.01)	0.58	6,931,904	691.3 GB
TL+CL(1000; score$\geq$0.01)	0.60	12,763,376	865.3 GB
TL+CL(1500; score$\geq$0.01)	0.61	18,117,552	1.0 TB
TL+CL(2000; score$\geq$0.01)	0.61	22,734,544	1.1 TB

Query Processing with Merge Joins

No need for top-k overhead!

TL(hotel)	TL(california)
d73:9,3	d68:9,1
d61:7,2	d7:8,6
d78:5,0	d73:5,9
d7:4,5	d8:4,6
⋮	⋮

descending score

threshold

top-k results

sort

merge join

TL(hotel)	TL(california)
d7:4,5	d7:8,6
d61:7,2	d8:4,6
d73:9,3	d68:9,1
d78:5,0	d73:5,9

increasing id

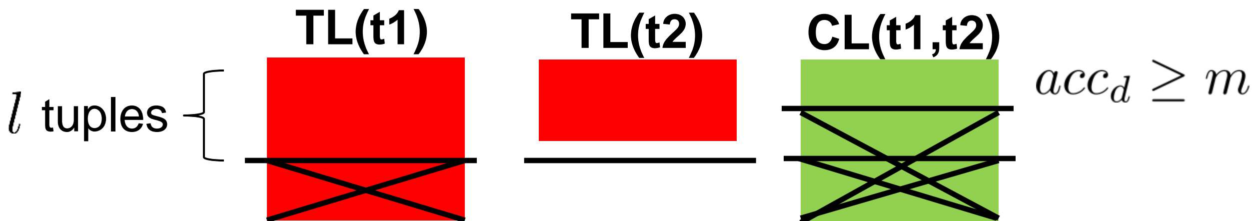
prune and resort index lists

Evaluation

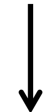
Static index pruning for TL+CL

Our pruning approach

- keep all pair lists (more precise: CLs)
- tune list length l and
- minimum acc_d-score m and text window size W=10 for CLs



- **Apply compression** to docid-ordered index lists:
  - docid values: delta-encoding + v-byte encoding
  - scores: v-byte encoding (normalization ⇒ ≤ 2bytes each)



$I(C, l, m)$: index for collection C with TLs and CLs cut after l entries and only keeping CL tuples with $acc_d \geq m$ (and text window W=10)

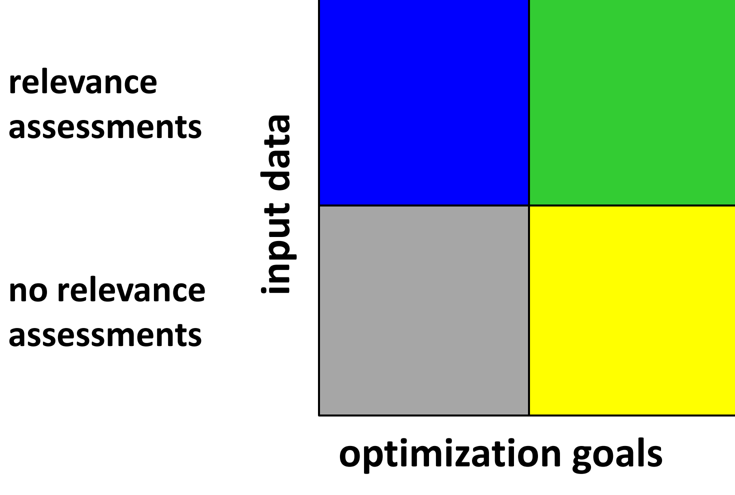
Index tuning

Two optimization goals:

- effectiveness-oriented index tuning:
  best retrieval quality within index size constraint (then minimize size)
- efficiency-oriented index tuning:
  at least BM25 quality and query processing as fast as possible.

Available input data:

- absolute index quality tuning:
  we have relevance assessments
- relative index quality tuning:
  we do not have relevance assessments



[Broschart2012]

Index quality measures

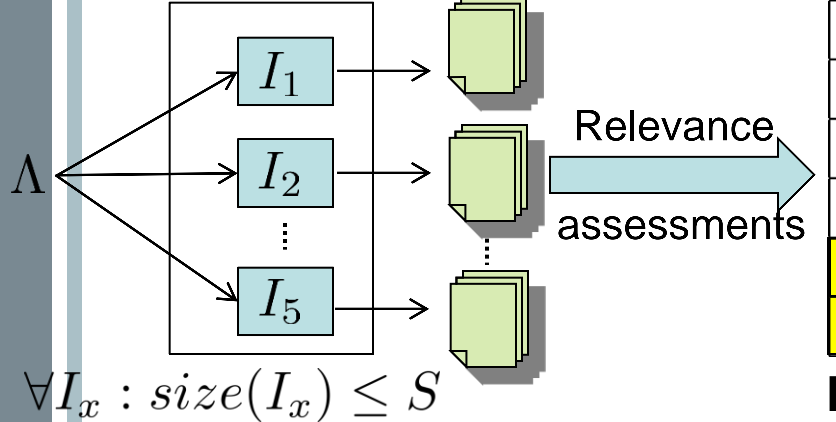
Goal: choose pruning parameters l and m for a given collection C, an upper limit S for the index size, and a result cardinality k s.t. the index quality measure $M(C, l, m, k)$ is maximized.

Absolute index quality tuning:
input: training topics Λ + their **relevance assessments**
$p_\Lambda[k; I]$: average quality of top-*k* results (e.g., P@*k*) over Λ on index I

effectiveness-oriented: maximize $M(C, l, m, k) = p_\Lambda[k; I(C, l, m)]$

Example ($S = 150GB, k = 10$)

Top-*k* results of I_x



$index$	l	m	$size(index)$	$p_\Lambda[k; index]$
I_1	210	0.1	110GB	0.50
I_2	310	0.1	120GB	0.51
I_3	410	0.15	130GB	0.51
I_4	510	0.1	140GB	0.52
I_5	610	0.1	150GB	0.52

Maximize precision for all feasible indexes:
equally high precision of I_4 **and** I_5
Pick index with smaller index size I_4

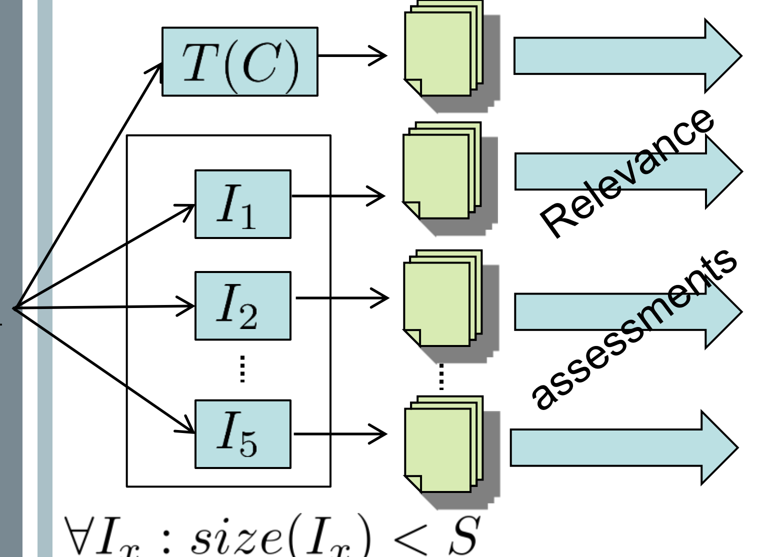
Index quality measures

Goal: choose pruning parameters l and m for a given collection C, an upper limit S for the index size, and a result cardinality k s.t. the index quality measure $M(C,l,m,k)$ is maximized.

Absolute index quality tuning:
input: training topics Λ + their **relevance assessments**
$p_\Lambda[k;I]$: average quality of top-*k* results (e.g., P@*k*) over Λ on index I

efficiency-oriented: maximize $M(C,l,m,k) = \begin{cases} \frac{1}{l} & \text{if } \frac{p_\Lambda[k;I(C,l,m)]}{p_\Lambda[k;T(C)]} \geq 1 \\ 0 & \text{else} \end{cases}$

Example ($S = 150GB, k = 10$**)**



$p_\Lambda[k;T(C)] = 0.51$

index	l	m	$size(index)$	$p_\Lambda[k;index]$
I_1	210	0.1	110GB	0.50
I_2	310	0.1	120GB	0.51
I_3	410	0.15	130GB	0.51
I_4	510	0.1	140GB	0.52
I_5	610	0.1	150GB	0.52

$Candidates = \{I_x : \frac{p_\Lambda[k;I_x]}{p_\Lambda[k;T(C)]} \geq 1\}$

Lowest length for I_2 = maximal index quality

Warm cache comparison to BMW

50,000 queries from TREC Terabyte Efficiency Track 2005: compare fastest index (l,m)= (310,0.05) (efficiency-oriented index tuning) to state-of the art DAAT-algorithm BMW. Use LRU cache of varying size.

$\bar{l}$	$\overline{m}$	cache size[MB]	cache hit ratio [bytes]	cache hit ratio [#lists]	#non-cached lists	$\varnothing t_{warm}$ [ms]
310	0.05	8	28.98%	29.29%	161,393	39.85
310	0.05	16	37.05%	37.08%	143,613	36.04
310	0.05	32	44.36%	43.89%	128,069	32.70
310	0.05	64	50.54%	49.39%	115,525	29.67
310	0.05	1024	54.44%	52.77%	107,801	28.92

k	index	cache size[MB]	cache hit ratio [bytes]	cache hit ratio [#lists]	#non-cached lists	#read blocks	$\varnothing t_{warm}$[ms]
10	$T(C)_{BMW}$	64	3.62%	2.15%	115,938	397,735,689	204.63
100	$T(C)_{BMW}$	64	3.62%	2.15%	115,938	573,001,133	259.01
10	$T(C)_{BMW}$	1024	46.39%	26.58%	86,990	397,735,689	173.16
10	$I'(C)_{BMW}$	1024	44.02%	14.57%	197,638	312,500,850	206.36

Speedup of our approach: factor 7 for top-10, factor 9 for top-100 retrieval (cache hit ratio 50% vs less than 4%)
Index size: (310,0.05): 94.9GB, $T(C)_{BMW}$: 10.5GB, $I'(C)_{BMW}$: 221.0GB

Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - **Non-Traditional Top-K Processing**
    - Query Expansion
    - Proximity-Aware Retrieval
    - **Top-k with Constrained Budget**
- Efficient Precomputation
- Efficient Distributed Query Processing

Dual Optimization Problem

So far:

Minimize answer time for optimal results

But: this may take too long (several seconds).

Now:

Maximize result quality for given answer time (or processing cost)

User-oriented efficiency measure

[Shmueli-Scheuer et al., ICDE 09]

Classes of Top-k Algorithms

- **Budget-Keeping Algorithms**:
  Execution cost never exceeds predefined limit
- **Budget-Oblivious Algorithms**:
  Scheduler does not know cost limit
  (**Anytime**-algorithms)
- **Budget-Aware Algorithms**:
  Scheduler knows cost limit in advance, optimizes for result quality when limit is hit

Measuring Result Quality

Gold standard:

Results R_{opt} of top-k algorithm with unlimited budget

Goal:

Optimize relative overlap of results R with R_{opt}

Traces for a query

Trace: sequence of steps performed by an algorithm

- Sequential scan in a list (cost 1)
- Random access to a previously read item (cost C)

Cost of a trace: sum of cost of its steps

Results of a trace: Results of a top-k algorithm performing the steps of the trace in this order

Optimization problem

Given a query with the corresponding lists, find a trace with cost ≤ B with a result that maximizes relative overlap with R_{opt}

This is a nontrivial problem.

	L_1		L_2
1	s:0.95		a:1.00
2	u:0.93		b:0.90
3	t:0.92		c:0.85
4	d:0.90		d:0.80
5	x:0.50		t:0.60
6	y:0.40		e:0.40
7	⋮		⋮

Fig. 2. Prefixes of two lists

C=3 (cost for random access)

Final top-2 result: {d,t}

Correct result requires at least **budget 9 (4SA in L1, then 2RA to L2 for d and t)**

For precision 0.5, we need at least budget 6 (t)

TA: budget 12 to find {t}, budget 16 to find {d,t}

NRA with round-robin: 8 steps to find {d}, 10 steps for {d,t}

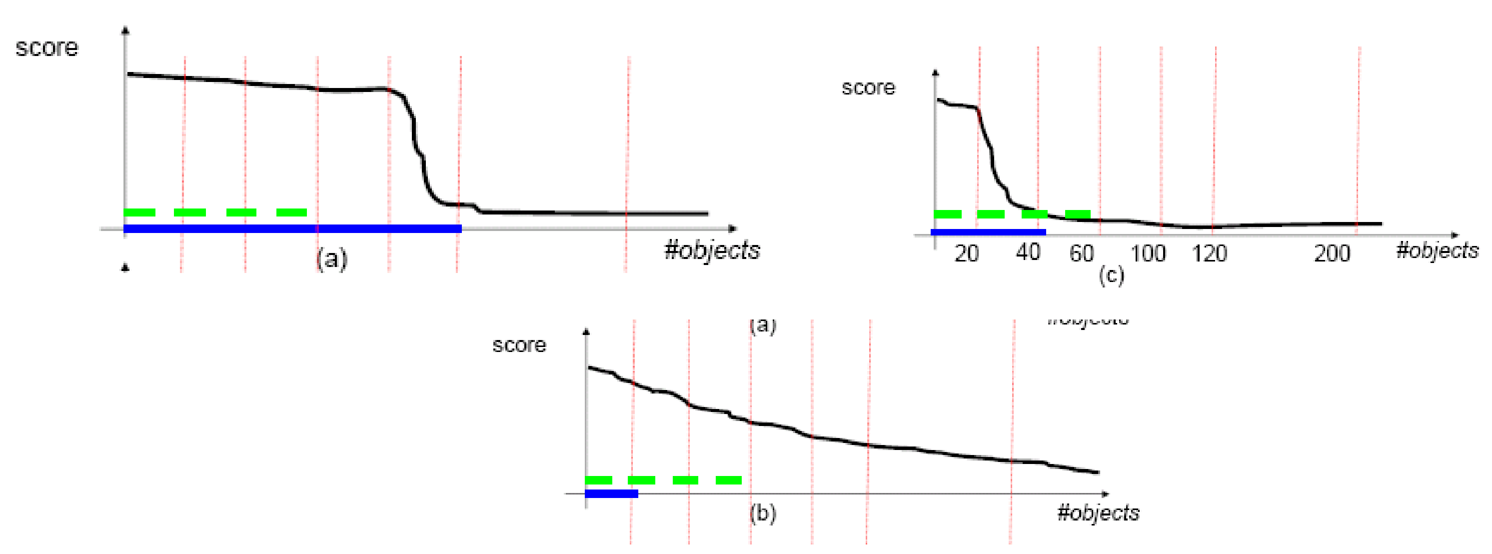
Results depend on clever scheduling of SA and RA

Heuristics for SA scheduling 1

Two execution phases (without sharp transition):

- *Gathering*: Find good candidate items (with high scores) that may be in final top-k
- *Reducing*: Decide for k results in the final top-k (reducing score bounds by dropping list high score bounds)

Rule of thumb: Mediocre scores don't help



Heuristics for SA Scheduling 2

Schedule batches of size *b* (*b*<<budget)

Utility functions for performing *x* scans on list *i*:

- Based on average score

$$util_{as}(L_i, x) = \frac{1}{x} \cdot \sum_{j=pos_i}^{pos_i+x} score_i(j). \qquad (1)$$

- Based on score drop

$$util_{sr}(L_i, x) = high_i - score_i(pos_i + x). \qquad (2)$$

Heuristics for SA Scheduling 3

Combined utility for optimization:

$$util(L_i, x) = \alpha \cdot util_{as}(L_i, x) + (1 - \alpha) \cdot util_{sr}(L_i, x). \quad (3)$$

where α depends on the phase:

$$\alpha = \begin{cases} 1, & \text{if less than k different items have been seen} \\ \frac{1}{|cand.\ set|} \sum_{c \in cand.\ set} p_k(c), & \text{else} \end{cases}$$

$p_k(c)$: probability that c will move to the top-k

Heuristics for SA Scheduling 4

Fair scheduling of the next b accesses:

Assign to each list L_i a number SA_{Li} of SA

$$SA_{L_i} = b \times \frac{util(L_i, b)}{\sum_{j=1}^{m} util(L_i, b)} \qquad (6)$$

More complex (and more effective) heuristics

Experiments: SA Scheduling

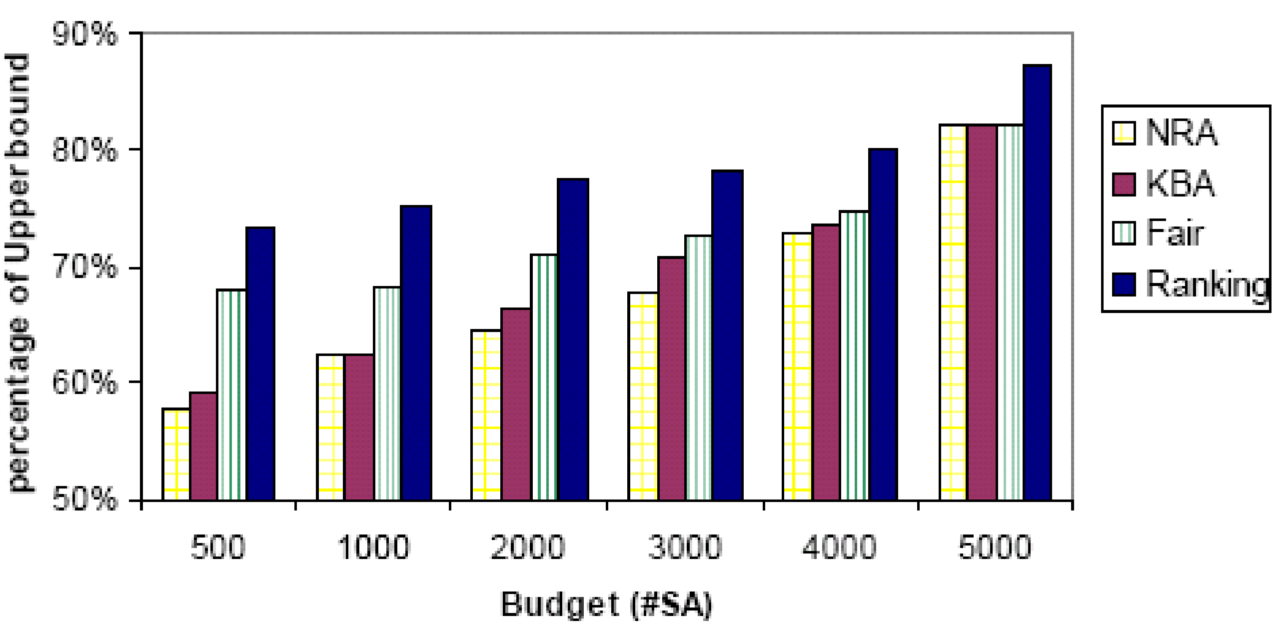


Fig. 6. TREC: average percentage of optimal precision, k=100, varying budgets

RA Scheduling is a Lot More Difficult

Key questions to answer:

- When to switch from SA to RA?
  - Need to have seen „enough" items
  - Need to have „enough" budget left
- Which items to access?
  - Goal: RA only for „good" items, not to eliminate candidates

Some results, but far from real understanding

Part 1 – Uncovered Issues

- Inverted file organization, ~~compression~~, …
- Caching of (partial) results
- Hardware issues
  - Multicore CPUs
  - Memory hierarchies (CPU caches, flash disks)
  - Nonstandard hardware (FPGA, GPU)
- Parallel and Distributed Retrieval
  - ~~Distribute & replicate lists over different machines~~
  - Query distributed data over the Web
- XML retrieval
- …

Part 1 – Summary

- Top-k processing central part of search engines
- Basic problem well understood in the literature
- Good engineering can make the difference

- Many interesting problems still out there
  - Heuristics are good, but guarantees would be better.

Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - Non-Traditional Top-K Processing
- Efficient Precomputation
  - **The Map-Reduce Framework**
- Efficient Distributed Query Processing

Web-Scale Computation

Many problems cannot be easily scaled to the Web ~~(about 20TB per Google crawl)~~

(commoncrawl.org: 5 billion pages, 60TB)

- Document inversion
- PageRank etc. computation
- Web log mining
- Host statistics
  - Term distribution per host
  - Accesses per host

Motivation

Precomputation on 20TB of data?

Easy, we have ~~paris~~ ~~titan:~~ himalia:

25,000€ $\Rightarrow$ 1625

Dell PowerEdge R810
Preis ab .... 22.498,02 €
zzgl. MwSt. und Versand

- Dell PowerEdge R810

	27.09.2012 04:57:33 Central Standard Time			
nmer	80852 Retail rc1213273			
mer / Beschreibung	Produktcode	Qty	SKU	ID
810 Rack Chassis, Up to 6x 2.5"	554566	1	[210-35881]	1
E7-4820, 8C, 2.00GHz, 18M Cache, W TDP, Turbo, HT, DDR3-980MHz	550295	1	[213-13409]	146
Prozessor: E7-4820, 8C, 2.00GHz, 18M Cache, W TDP, Turbo, HT, DDR3-980MHz	552485	1	[374-14140]	2
cher: y for 2 or 4 CPUs, DDR3, 1066MHz ad Ranked LV RDIMMs)	569502	1	[370-20832]	3

ALDI SÜD

MEDION® empfiehlt Windows® 7.

MEDION® www.medion.de

MULTIMEDIA-PC
MEDION® AKOYA® E4065 D
Ab jetzt im Handel

3 JAHRE GARANTIE

NAVIGATION
Ausstattung
Prozessor
Software
Design
Windows 7
Glossar
Impressum
Zum Produktfilm

Original Windows® 7 Home Premium 64 Bit

Neuester AMD Quad-Core A8-5500 Accelerated Prozessor

Brillante AMD Radeon™ HD 7560D DirectX® 11 Grafik

Großer 4 GB DDR3 SDRAM Arbeitsspeicher mit 1600 MHz

je 399,-

Kaufen Sie einen Windows 7-PC und erhalten Sie Windows 8 Pro für nur 14,99 €. Dieses Angebot ist ab dem 2. Juni 2012 bis zum 31. Januar 2013 gültig. Ausführliche Informationen finden Sie unter „windowsupgradeoffer.com"

Large Clusters of Commodity Hardware

- Thousands of off-the-shelf networked PCs
- Hardware failures (of single machines) common
- Harddrive failures common

- Distributed Programs to exploit full power (RPC, CORBA, MPI, WebServices, REST, ...?)

MapReduce Features

- Complete solution for distributed computing
- **Simple**, but powerful **interface**
- Implementation within **hours**, not weeks
- **Detects** machine failures and **redistributes** work
- **Avoids** data loss due to harddisk failures (together with distributed file system)

Widely used at Google for daily business
(2 mio MapReduce jobs in Sep 07 on 15TB each, 400s each)

[Dean et al., CACM 51(1), 2008]

MapReduce by Example

Problem: Compute document frequencies

- Input: data with keys (docs with docids/urls)
- Output: aggregated data (terms with counts)

Solved by two functions (provided by user):

- **MAP**: partition input data by output key (term)
- **REDUCE**: aggregate data for each output key

Automatically executed in a distributed fashion

MapReduce by Example

```
map(String key, String value)
  // key: document name
  // value: document content
  for each term in value:
    EmitIntermediate(term,1);


reduce(String key, Iterator values)
  // key: term
  // values: list of counts
  int result=0;
  for each v in values:
    result:=result+value;
  Emit(term,result);
```

Architecture

User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1

Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

taken from [Dean et al., CACM 51(1), 2008]

Architecture

- Dedicated **master process** identifies worker processes/machines for map and reduce
- Master **partitions** input file into M partitions
- Partitions **assigned** to map workers
- Map workers **output to R files** on local hard disks (by hash code), master notified
- Each reduce worker **reads** one output file from the map workers (by RPC) & **sorts** them (many output keys per file!)
- Each reduce worker **aggregates** data per key

M2CI CLUSTER OF EXCELLENCE

Failure Handling

- Master monitors workers
- On worker failure:
  - All MAP tasks marked failed and submitted to other workers (including finished ones – data on local hard disk!)
  - All active REDUCE tasks resubmitted to other workers
  - Requires idempotence of operations (workers could just be slow, not failed)

Application Example: PageRank

- Definition of PageRank

$$PR(v) = \varepsilon \sum_{(u,v)\in E} \frac{PR(u)}{\text{outdeg}(u)} + (1-\varepsilon)$$

- Computed through power iteration:
  values in step i computed from values in step i-1 and graph structure
- Highly local computation: requires only old pageranks from incident nodes

PageRank in MapReduce

Map: distribute PageRank "credit" to link targets

Reduce: gather up PageRank "credit" from multiple sources to compute new PageRank value

Iterate until convergence

[Picture probably courtesy of Jimmy Lin or Christophe Bisciglia et al.]

Initial Step

MAP:

(url, content)

↓

(url, (initial pagerank, list(linked urls)))

REDUCE:

Passes input tuples to output without change

Iteration Steps

MAP:

(url, (PR, list(n linked urls)))

↓

(linked url 1, PR/n), …, (linked url n, PR/n),

(url, list(n linked urls)

REDUCE:

(url, PR1),…,(url, PRx) , (url, list(linked urls))

↓

(url, (PR', list(linked urls)))

Termination

Terminate when values are stable

(determined by central component)

Implementations freely available

- PIG (Yahoo)
  http://research.yahoo.com/node/90
- Hadoop (Apache)
  http://hadoop.apache.org/
- DryadLinq (Microsoft)
  http://research.microsoft.com/research/sv/DryadLINQ/

Pig Latin vs. SQL

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10^6;
output = FOREACH big_groups GENERATE
category, AVG(good_urls.pagerank);
```

Part 2 – Summary

- MapReduce is a powerful framework for distributed computing
- Exploits potential of commodity hardware
- Hadoopify your applications!
- But: Does not solve everything

Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - Non-Traditional Top-K Processing
- Efficient Precomputation
- Efficient Distributed Query Processing
  - **Term-based partitions**
  - Document-based partitions

Simplified Preprocessing Procedure

1. For each document d in the collection
   for each term t contained in d
   emit tuple (t,d,score(d,t)) to temp storage
2. Group tuples by term
3. Build inverted lists for each term

Map phase of a MapReduce job

Reduce phase of a MapReduce job

Natural steps to exploit distribution: 1+3

- Assign subcollections to different machines and parse documents at these machines
- Two alternatives:
  - Generate local index for each subcollection
  - Combine all temp data, partition it by term to different machines, create global inverted list for each term

Two distribution models

Index can be distributed in two ways:

- Partitioned by terms (complete index lists at different machines);
  often the outcome of index creation
- Partitioned by documents (subcollections with their own indexes at different machines);
  often caused by natural distribution of data

Result of horizontal partitioning of table, can be seen as „distributed database“ with one logical table:

T

t1	d1	score(d1,t1)
t2	d1	score(d1,t2)
t1	d2	score(d2,t1)
t2	d2	score(d2,t2)
t1	d3	score(d3,t1)

→

T1

t1	d1	score(d1,t1)
t1	d2	score(d2,t1)
t1	d3	score(d3,t1)

T2

t2	d2	score(d2,t2)
t2	d1	score(d1,t2)

or

D1

t1	d1	score(d1,t1)
t2	d1	score(d1,t2)

D2

t1	d2	score(d2,t1)
t2	d2	score(d2,t2)
t1	d3	score(d3,t1)

QP for Term-Based Partitions

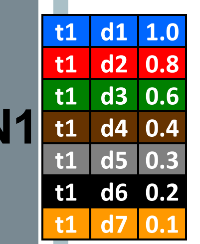
Can we apply straight-forward techniques from distributed databases?

Assume query with 3 terms at 3 (different) nodes, compute top-1
- Ship all to one node (here, the query initiator Q)
- (Semi-)Join at Q, sum scores, project terms away, sort by score

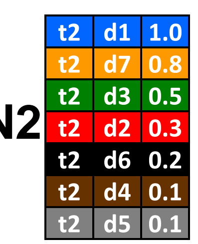
Q |x| |x| =

d1	2.1
d3	1.4
d2	1.3
d6	1.0
d7	0.9
d4	0.7
d5	0.7

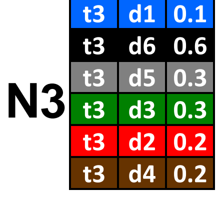
cost: 3 messages, 20 attribute values

N1


t1	d1	1.0
t1	d2	0.8
t1	d3	0.6
t1	d4	0.4
t1	d5	0.3
t1	d6	0.2
t1	d7	0.1

N2


t2	d1	1.0
t2	d7	0.8
t2	d3	0.5
t2	d2	0.3
t2	d6	0.2
t2	d4	0.1
t2	d5	0.1

N3


t3	d1	0.1
t3	d6	0.6
t3	d5	0.3
t3	d3	0.3
t3	d2	0.2
t3	d4	0.2

Can we do better?

Basic distributed top-k algorithm: TPUT

Assume query with m terms at m (different) nodes

Three phases, driven by query initiator Q:

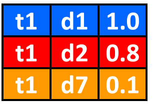
1. Collect top-k entries from all lists at Q, join and sort them by score, denote score of current top-k by mink
2. Collect all entries with score at least mink/m from all lists at Q, recompute current top-k and mink, prune candidates
3. Get missing scores for all remaining candidates

(Easy) Theorem:
Step 2 does not miss any final top-k results, TPUT is correct.

TPUT for the example

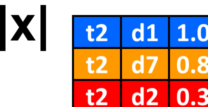
- Each node ships top-1 local entry to Q
- (Semi-)Join at Q, sum scores, project terms away, sort by score
- Each node ships entries with score ≥mink/3 to Q
- Update scores at Q
- Get missing scores for remaining candidates, update scores

Q



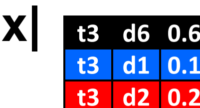
t1	d1	1.0
t1	d2	0.8
t1	d7	0.1

|x|



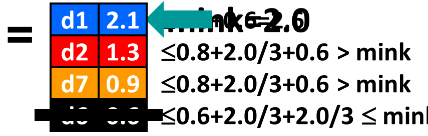
t2	d1	1.0
t2	d7	0.8
t2	d2	0.3

|x|



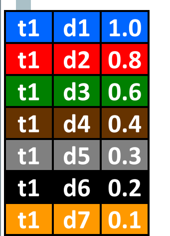
t3	d6	0.6
t3	d1	0.1
t3	d2	0.2

=



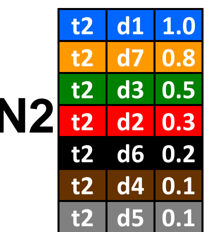
d1	2.1	mink=2.0
d2	1.3	≤0.8+2.0/3+0.6 > mink
d7	0.9	≤0.8+2.0/3+0.6 > mink
[illegible]	[illegible]	≤0.6+2.0/3+2.0/3 ≤ mink

N1



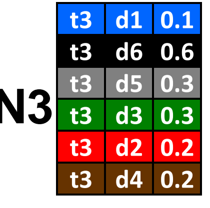
t1	d1	1.0
t1	d2	0.8
t1	d3	0.6
t1	d4	0.4
t1	d5	0.3
t1	d6	0.2
t1	d7	0.1

N2



t2	d1	1.0
t2	d7	0.8
t2	d3	0.5
t2	d2	0.3
t2	d6	0.2
t2	d4	0.1
t2	d5	0.1

N3



t3	d1	0.1
t3	d6	0.6
t3	d5	0.3
t3	d3	0.3
t3	d2	0.2
t3	d4	0.2

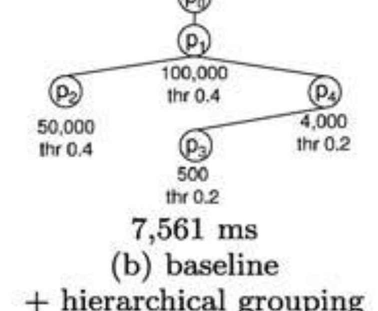
cost: 9 messages, 9 attribute values

Improvements for TPUT

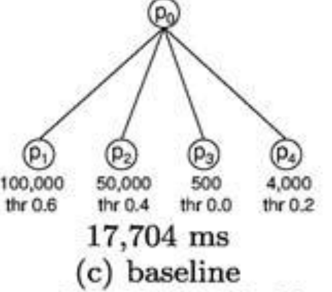
- **Locality**: Execute query at node with longest list, send only result to Q
- **Approximation**: Drop last phase, drop some lists (but: which lists?)
- Hierarchically **group** operators
- Distribute mink threshold **not uniformly**, but in a way that minimizes (estimated) number of values to

p0
p1 100,000 thr 0.3
p2 50,000 thr 0.3
p3 500 thr 0.3
p4 4,000 thr 0.3

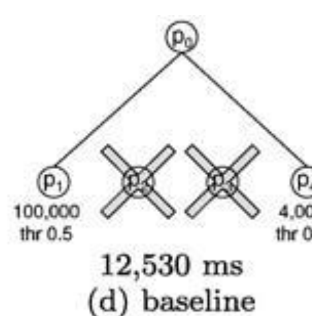
22,545 ms
(a) baseline



7,561 ms
(b) baseline
+ hierarchical grouping



17,704 ms
(c) baseline
+ adaptive thresholds



12,530 ms
(d) baseline
+ sampling

[From Neumann et al., Distributed top-k aggregation queries at large, 2009]

M2CI CLUSTER OF EXCELLENCE

Outline

- Efficient Query Processing
  - Introduction
  - Basic Top-k Algorithms
  - Scheduling 1x1
  - Approximation Algorithms
  - Non-Traditional Top-K Processing
- Efficient Precomputation
- Efficient Distributed Query Processing
  - Term-based partitions
  - **Document-based partitions**

2 important use cases for doc-based p.

- **Distributed indexing (always „cooperative“)**
  – **scale out indexing** by distributing existing collection over many nodes
  – Keep index at each node (plus optional extra nodes)
  $\Rightarrow$ **scale out query processing** as well (indexes in memory)
- **Federation of independent search engines**
  - Document partitions built independently (crawlers, digital libraries, archives)
  - Local indexes built independently
  - Perform federated queries over all search engines (examples: excite.com, metager.de)

2 important use cases for doc-based p.

- **Distributed indexing (always „cooperative“)**
  - **scale out indexing** by distributing existing collection over many nodes
  - Keep index at each node (plus optional extra nodes)
  - ⇒ **scale out query processing** as well (indexes in memory)
- **Federation of independent search engines**

Important distinction:

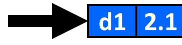
- ***cooperative sources*** *provide details about scores (implementations, parameters, statistics, …) and allow partial access to their collection (e.g., for computing new statistics)*
- ***uncooperative sources*** *provide only a query-based interface and no access to internal operations (only sampling possible)*

QP for doc-based partitions

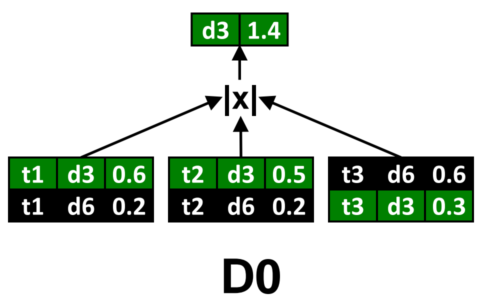
- Documents distributed over multiple servers (may or may not include duplicates)
- Straight-forward top-k query processing:
  - Submit top-k query to all servers
  - Collect results at dedicated machine & combine to overall top-k

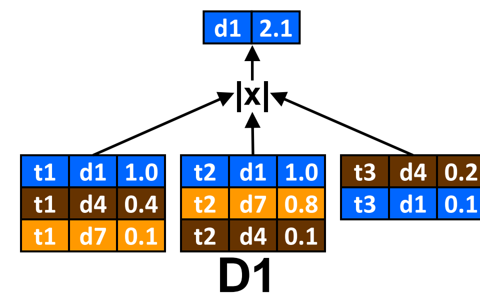
Two questions:

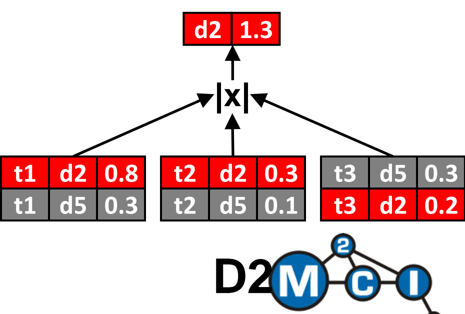
1. **Is this always correct?**
2. **Is this always efficient?**



cost: 3 messages, 3 attribute values







Correctness under score equivalence

- **Correctness**:
  combination of local top-k results identical to top-k result in unpartitioned collection
- straight-forward for **partitioned database**
- Not necessarily true for **distributed search engine**:
  - Indexes may be build locally
  - Scores may be **computed locally** (with local document frequencies!)
- **Solution**: Make sure that local and global scores are *equivalent* (e.g., keep global document freq.)
- Additional complication: Local optimizations (pruning of entries with low scores, …)

What if score equivalence is impossible?

- Scores may be **incomparable** (different scoring models or even **no scores** at all, e.g., Google)
- **Result Merging** (or Fusion) in such settings:
  - **Round Robin**:
    - Order sources by expected usefulness S_i (see later)
    - Pick result 1 from source 1, result 1 from source 2, etc.
  - Use **source-normalized scores**:
    - Normalize scores for all docs from a source to [1.0;0.0]
    - Multiply scores by expected usefulness to get sourced-normalized scores
    - Rank documents in order of sourced-normalized score
  - Use **machine learning** to predict scores:
    - Collect samples of each collection in central place
    - Learn correllation of result scores on centralized sample and in each collection (from large training set of queries)

M2CI CLUSTER OF EXCELLENCE

Improving Efficiency: Two Paths

- Reduce **number of results per partition**
  - For global top-k, usually local top-k‘ sufficient (with k‘<<k)
  - But: safe choice of local k‘ difficult (depends on scores of local results), estimation based on local score distribution (done at central node!)
  - **Approximate**, not exact query results
- Reduce **number of partitions** accessed
  - Many partitions have no (or hardly any) good results (esp. in federations over multiple domains)
  - Preselect a few good partitions for querying based on expected usefulness („collection selection problem“)
  - **Approximate**, not exact results

Collection Selection: CORI

Basic approach:

Rank collections similar to documents for a query

$$p(t \mid db_i) = b + (1-b) \cdot \underbrace{T_i(t)}_{\text{Importance of term t in collection } db_i} \cdot \underbrace{I(t)}_{\text{Importance of term t for collection selection}}$$

Belief in db_i for term t

Importance of term t in collection db_i

Importance of term t for collection selection

$$T_i(t) = \frac{df_i(t)}{df_i(t) + 50 + 150 * \frac{cw_i}{avg(cw)}}$$

$$I(t) = \frac{\log \frac{|DB| + 0.5}{cf(t)}}{\log |DB| + 1.0}$$

$df_i(t)$: number of documents in db_i that contain term t
$cw_i(t)$: number of words in db_i
avg(cw): average number of words in collections
|DB|: number of collections
cf(t): number of collections that contain term t

Given query Q={t1,…,tn}, rank sources by average belief

[Callan et al., 1995]

Extensions for Collection Selection

- Consider **size of collection**:
  larger collection should give more „good" results [Si, 2006]
- Consider **overlap of collections**:
  if two collections are largely overlapping, consider only one of them („diversity" of collections) [Bender et al., 2005]
- Estimate **usefulness** of collection from **sample**:
  Estimate number of relevant results from subcollection from scores of docs in sample (e.g., ratio of documents in sample with score>threshold, divided by sample ratio) [Si&Callan, 2003]

Upwards Down: Document Allocation

- **Goal**: Create partitions such that „usually“ a small number of collections is sufficient per query
- Options for document allocation:
  - Randomized
  - Group documents per source (e.g., Web server)
  - Group documents per topic:
    - Build coherent clusters of subset of documents
    - Assign remaining docs to clusters
    - Each cluster forms a partition

Example for Document Allocation

- TREC ClueWeb09-CatA collection (~500M docs)
- 50 standard benchmark queries
- Sample size 1%, 1000 partitions

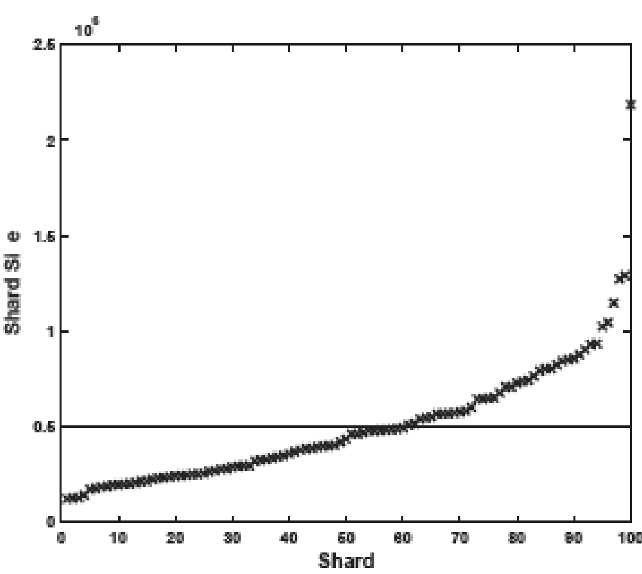


Figure 3: Distribution of shard sizes for the topic-based shards of the Clue-CatB dataset.

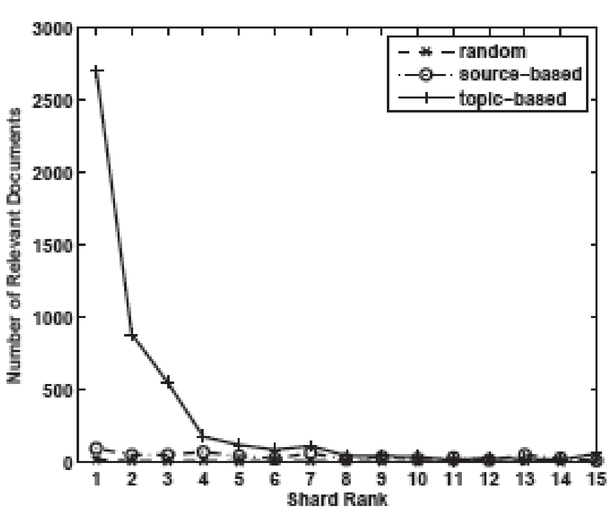


Figure 6: Distribution of relevant documents across top shards for the three allocation policies for the Clue-CatA-Eng dataset. Total number of relevant documents: 5684

figures from [Kulkarni and Callan, 2010]

Summary: Distributed IR

- Techniques in general very similar to DDBS
- Main techniques:
  - Distributed top-k
  - Collection selection
- Approximative variants very common (unlike in distributed databases, but may have applications there as well)

References 1 – Efficient Centralized QP

[Anh01] V. N. Anh et al: Vector-Space Ranking with Effective Early Termination. *SIGIR* 2001

[Anh06] V. N. Anh et al: Pruned query evaluation using pre-computed impacts. *SIGIR* 2006

[Anh06a] V. N. Anh et al: Pruning strategies for mixed-mode querying. *CIKM* 2006

[Arai07] B. Arai et al. Anytime measures for top-k algorithms. *VLDB* 2007.

[Bast06] H. Bast et al. IO-Top-k: Index-access optimized top-k query processing. *VLDB* 2006.

[Broschart12]A. Broschart et al. High-performance processing of text queries with tunable pruned term and term pair indexes, *ACM Trans. Information Syst., 30(1):5, 2012*

[Buckley85] C. Buckley et al. Optimization of inverted vector searches. *SIGIR* 1985.

[Chang02] K. C.-C. Chang et al. Minimal probing: Supporting expensive predicates for top-k queries. *SIGMOD* 2002.

[Fagin03] R. Fagin et al. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[Güntzer01] U. Güntzer et al. Towards efficient multifeature queries in heterogeneous environments. *ITCC* 2001.

[Ilyas08] Ihab F. Ilyas et al. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 2008.

[Marian04] A. Marian et al. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[Moffat96] A. Moffat et al. Self-indexing inverted files for fast text retrieval. ACM Trans. Inf. Syst., 14(4):349–379, 1996.

[Nepal99] S. Nepal et al. Query processing issues in image (multimedia) databases. *ICDE* 1999.

[Schenkel07] R. Schenkel et al. Efficient Text Proximity Search, *SPIRE* 2007.

[Shmueli09] M. Shmueli-Scheuer et al. Best-effort top-k query processing under budgetary constraints. *ICDE* 2009.

[Theobald04] M. Theobald et al. Top-k query evaluation with probabilistic guarantees. *VLDB* 2004.

[Theobald05]M. Theobald et al: Efficient and self-tuning incremental query expansion for top-k query processing. *SIGIR* 2005.

[Zobel06] J. Zobel et al. Inverted files for text search engines. *ACM Computing Surveys* 38(2), 2006.

[Strohman07] T. Strohman et al. Efficient document retrieval in main memory. SIGIR, 2007.

[Broder03] A. Broder et al. Efficient query evaluation using a two-level retrieval process. CIKM, 2003.

[Ding11] S. Ding et al. Faster top-k document retrieval using block-max indexes. SIGIR, 2011

[Ding11a] S. Ding et al. Batch query processing for web search engines. WSDM, 2011.

[BCC-4] Stefan Büttcher, Charles L.A. Clarke, Gordon V. Cormack: Information Retrieval, chapter 6 (index compression). MIT Press, 2010

References 2 – Efficient Precomputation

[Dean04] J. Dean et al. MapReduce: Simplified Data Processing on Large Clusters. *OSDI* 2004.

[DeWitt08] D. DeWitt et al. MapReduce: A major step backwards. The Database Column, January 17, 2008 http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html

[He08] B. He et al.Mars: A MapReduce Framework on Graphics Processors. *PACT* 2008.

[Olston08] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. *SIGMOD* 2008.

[Ranger07] C. Ranger. et al. Evaluating MapReduce for Multi-core and Multiprocessor Systems. *HPCA* 2007.

[Witten99] I. Witten et al. Managing Gigabytes. Morgan Kaufman, 1999.

[Yang07] H. Yang et a. Map-reduce-merge: simplified relational data processing on large clusters. *SIGMOD* 2007.

References 3 – Efficient Distributed QP

- Stefan Büttcher, Charles L.A. Clarke, Gordon V. Cormack: *Information Retrieval.* MIT Press, 2010
  - Chapter 4 for inverted indexes
  - Chapters 8-11 for scoring models
  - Chapter 14 for parallel IR
- Ihab F. Ilyas, George Beskales, Mohamed A. Soliman: *A survey of top-k query processing techniques in relational database systems*. ACM Comput. Surv. 40(4), 2008
- P. Cao, Z. Wang: *Efficient top-k query calculation in distributed networks*. PODC, pp. 206–215, 2004
- H. Yu, H.G. Li, P.Wu, D. Agrawal, A.E. Abbadi: *Efficient processing of distributed top-k queries*. DEXA, pp. 65–74, 2005
- Thomas Neumann, Matthias Bender, Sebastian Michel, Ralf Schenkel, Peter Triantafillou, Gerhard Weikum: *Distributed top-k queries at large*. Distributed and Parallel Databases 26:3-27, 2009
- James P. Callan, Zhihong Lu, W. Bruce Croft: Searching Distributed Collections With Inference Networks. SIGIR, 1995.
- Luo Si and Jamie Callan: *Relevant Document Distribution Estimation Method for Resource Selection*. SIGIR, 2003.
- Luo Si: *Federated Search of Text Search Engines in Uncooperative Environments*. PhD Thesis, CMU, 2006. http://www.lti.cs.cmu.edu/Research/Thesis/LuoSi06.pdf
- Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, Christian Zimmer: *Improving Collection Selection with Overlap Awareness in P2P Search Engines*. SIGIR, 2005
- Anagha Kulkarni and Jamie Callan: *Document Allocation Policies for Selective Searching of Distributed Indexes.* CIKM, 2010.
- Richard McCreadie, Craig Macdonald, Iadh Ounis: MapReduce indexing strategies: Studying scalability and efficiency. IP&M 48(5):873-888, September 2012