Introduction to Search Engine Technology

Term-at-a-Time and Document-at-a-Time Evaluation

Ronny Lempel

Outbrain

(Some of the following slides are courtesy of Aya Soffer, IBM Haifa Research Lab; and David Carmel, Yahoo! Labs)

Query Evaluation Strategies

- We’ve got an inverted index (Lexicon & postings lists)
- The posting elements in all postings lists are sorted by increasing location (docID, offset)
  - Furthermore, each postings list is contiguous on disk
- Given a query, we need to do the following:
  1. Parse and tokenize it – turn into a list of search terms, taking into account operators (+, -, “..”)
  2. Lookup terms in Lexicon
  3. Get a postings iterator (cursor) per term from inverted index, calculate term weights
  4. Calculate score per matching document
  5. Return top scoring documents

29 November 2015 236620 Search Engine Technology
Query Evaluation Strategies

“Calculate score per matching document”:
- Term-at-a-Time Processing (TAAT): scan postings lists one at a time, maintain a set of potential matching documents along with their partial scores.
- Document-at-a-Time Processing (DAAT): scan postings lists in parallel, identifying at each point the next potential candidate document and scoring it.

Pros and cons will depend primarily on:
- Query semantics (conjunctive vs. disjunctive).
- Allowed operators (e.g., phrase support).
- Ranking logic (e.g., proximity considerations).
- Whether the index is distributed across multiple machines or not, and if distributed—how? (next lecture).

TAAT Conjunctive Query Processing

Boolean conjunctive query:
- For each query term \( t \), locate lexicon entry.
  - Record frequency \( df(t) \) and grab the postings list \( L_t \) of \( t \).
- Identify \( t^* \) - term with smallest frequency (rarest term).
- Iterate through \( L_{t^*} \) (sequential disk read), and set \( C \leftarrow L_{t^*} \).
  - \( C \) is the set of candidates, ordered by increasing docIDs.
- For each remaining term \( t \) in increasing \( df(t) \) order:
  - Merge candidate set \( C \) with current postings list \( L_t \).
    - For each docID \( d \) in \( C \), if \( d \) is not in \( L_t \) then set \( C \leftarrow C \backslash \{d\} \).
  - If \( C=\emptyset \) return, there is no answer.
- For each \( d \) in \( C \), return \( d \).
**TAAT Disjunctive Query Processing**

Boolean disjunctive query:
- For each query term $t$, locate lexicon entry
  - Record frequency $df(t)$ and grab the postings list $L_t$ of $t$
- Identify $t^\star$ - term with highest frequency
- Iterate through $L_{t^\star}$ (sequential disk read), and set $C \leftarrow L_{t^\star}$
  - $C$ is the set of candidates, ordered by increasing docIDs
- For each remaining term $t$ (in arbitrary order):
  - For each docID $d$ in $L_t$, If $d$ is not in $C$ then set $C \leftarrow C \cup \{d\}$
- For each $d$ in $C$, return $d$

---

**TAAT Vector Space Evaluation for Top-r Retrieval**

TF/IDF scoring (cosine similarity measure):
1. Set $A=\emptyset$, an empty set of accumulators
   - Denote by $A_d$ the score accumulator for document $d$
2. For each query term $t$ in $Q$
   - Record $df(t)$ and grab postings list $L_t$
   - Set $idf_t \leftarrow \log(N/df(t))$
   - For each docID $d$ in $L_t$
     - If $A_d$ is not in $A$: $A_d \leftarrow 0; A \leftarrow A \cup \{A_d\}$
     - Update $A_d \leftarrow A_d + idf_t \cdot freq_d(t)$
3. Normalization: for each $A_d$ in $A$, Set $A_d \leftarrow A_d/\|d\|$
   - This normalizes $A_d$ to be proportional to $\cos(Q, d)$
4. Return the $r$ documents with the highest scores in $A$ in decreasing relevance order
**Top-r Document Selection**

How can we efficiently return the r documents with the highest scores in A in decreasing relevance order?

- **Naive method:** sort the set of accumulators
  - If |A|=M, time complexity is O(M logM)
- **Better approach:** since typically r<<M, selecting the r top scores can be done in O(M+r log M) time using a heap:
  1. Heapify the set of M scores (about 2M comparisons) so that the top score is at the root
  2. Repeatedly extract the heap’s root (r times), each time fixing the heap in O(logM)

---

**The Heap Data Structure - Reminder**

- A binary heap is a (mostly full) binary tree with values stored at all leaves and internal nodes, and an ordering rule that requires values to be non-decreasing (alternatively, non-increasing) along each path from a leaf to the root
  - Largest/smallest value is at the root

```
  23
 /   \
17    15
 /  \
17   2  13
/  \
14  8  \\
4    5
```
Extracting the Top-r Elements

- Remove the largest item $r$ times
- Each time:
  - Remove the largest item – the root of the heap
  - Replace it with the last element of the heap (deepest and rightmost leaf)
  - Sift the new root down until restoring order; number of sifts is bounded by height of heap, i.e. $\log($size of heap$)$

Top-r Selection Using a Min-Heap

- The selection problem can be solved by a heap that stores the smallest item at the root: min-heap
- A min-heap of $r$ items is held instead of a max-heap of $M$ – lots of memory is saved, which is always good
- Process the $M$ accumulator values, storing in the min-heap the $r$ largest values seen so far
  - First $r$ values are heapified in $O(r)$ comparisons
  - Replace the smallest value in the min-heap (the $r^{th}$ largest) whenever a larger value is found
- Sort the $r$ highest values in descending order and return the corresponding documents – $O(r \log r)$
Min-Heap Processing - Illustration

<table>
<thead>
<tr>
<th>Processed</th>
<th>Unprocessed</th>
</tr>
</thead>
</table>

- Min-heap of r largest items
- Discard smallest value

Top-r Selection Using a Min-Heap: Complexity Analysis

- Worst case: the scores are already in increasing order
  - Each of the M-r last values is inserted into the heap
  - Furthermore, it percolates to the bottom of the heap
  - Complexity is $O((M-r) \times \log(r))$

- Average case – the scores arrive in a permutation of size M chosen uniformly at random
  - The expected number of times one of the M-r last values is inserted into the heap is $\sim r \times \ln(M/r)$
  - Each insertion costs $O(\log(r))$
  - Complexity is $O(r \times \log(r) \times \log(M/r))$
TAAT: Buckley & Lewit Pruning Process (SIGIR 85)

- For each query term \( t \), compute its maximal score contribution to any document and denote by \( \text{ms}(t) \)
- Sort & scan the terms in descending order of \( \text{ms}(t) \)
- During accumulation, maintain a min-heap of size \( r+1 \)
- After accumulating the contribution of term \( t \):
  - If \( A_r > A_{r+1} + \sum_{k>i} \text{ms}(t_k) \), stop query processing and return the top \( r \) docs
- Lemma: the pruning process returns the same \( r \) docs as the full process (not necessarily in the same order)
- This is a form of “early termination” of the query evaluation process

TAAT and Web Search

- Queries on the Web are typically short (less than 3 words on average)
- Billions of documents
- Implications:
  - Conjunctive queries still provide more than enough recall
  - Proximity of query terms in documents is very important and improves scores over classic TF/IDF
- Web search engines allow exact-phrase queries
- How can proximity considerations and exact-phrase searches be accomplished in term-at-a-time evaluation?
Document-at-a-Time Evaluation

- Will identify matching documents in increasing docID order
- The postings lists of all terms will be *aligned* on each matching document
- Terms within the document can be enumerated in increasing offset order, making it easy to identify terms appearing in proximity
- Main issue: with all postings lists being traversed in parallel rather than sequentially, how can disk I/O be optimized?

Postings API

- Each postings list will be traversed by a cursor
  - The cursor for term t will be denoted \( C_t \)
  - A cursor supports the following operators:
    - `init()` – position at the beginning of the list
    - `next()` – return the position (docID, offset) of the next posting element in the list
    - `nextBeyond(position)` – find the first posting element in whose position is beyond the argument
      - If the cursor is already beyond the given position, it doesn’t move
      - When no more positions are available, the above methods return \( \infty \)
- In the next slide, we assume positions are just docIDs
Zig-Zag Join for Enumerating Candidates in Conjunctive Queries

- For each term $t$: $c(t).init()$
- Repeat
  - $\text{candidate} \leftarrow c(t_0).next()$, $t_{\text{align}} \leftarrow 1$  // toss ahead first term
  - While ($\text{candidate} < \infty$ && $t_{\text{align}} < \text{numTerms}$):
    - $\text{nextDoc} = c(t_{\text{align}}).nextBeyond(\text{candidate}-1)$
    - If ($\text{nextDoc} == \text{candidate}$):
      - $t_{\text{align}} \leftarrow t_{\text{align}}++$
    - else  // nextDoc must be larger than candidate, toss first term
      - $\text{candidate} \leftarrow c(t_0).nextBeyond(\text{nextDoc}-1)$; $t_{\text{align}} \leftarrow 1$
  - If ($t_{\text{align}} == \text{numTerms}$):  // alignment found
    - Score candidate, enter into min-heap
- Until ($\text{candidate} == \infty$)
- Output the top-$r$ documents of min-heap in decreasing score order

Zig-Zag Join, Observations

- To increase the expected location skip per nextBeyond() operation, terms should be ordered from rarest to most frequent – the rarest term “drives” the query!
- Phrase matches can be found similarly, by zig-zagging the phrase components to be found in the correct relative positions
  - We thus build a virtual cursor for a phrase, that exposes the normal postings API to whoever is driving it
- For two terms, reduces to just a simple merge of their respective postings lists
  - Finding common entries in two sorted lists of length $L_1$ and $L_2$ can be done naively in $O(L_1+L_2)$
  - What if $L_1 >> L_2$? Can we improve? What about I/O considerations (sequential is good, random is bad)?
- Consequence: need efficient forward skipping on postings lists!
Efficient Skipping in Postings Lists

- In order to efficiently support skipping (forward) in postings lists, lists are often implemented as B/B⁺ Trees or Skip Lists (adapted to disk I/O)
  - B⁺Tree – A B-Tree whose values are only stored in the leaves (intermediate nodes only hold keys)
    - furthermore, the leaves are laid out sequentially (or chained) to allow for easy iteration
- In a B-Tree implementation, all postings lists can be encoded in a single tree by having the sort key be (termID, location)
- With efficient skipping, the less reading done in DAAT processing as compared with TAAT processing can compensate for the I/O being random access and interleaved rather than sequential

Early Termination in DAAT Evaluation

In certain scoring models, DAAT evaluation schemes support early termination. For example:

- Assume that document identifiers are assigned in decreasing order of some query-independent “static score”
- Suppose that the score of each document is a linear combination of its query-dependent text score and its static score:
  \[ \text{score}(d) = \alpha \times \text{textScore}(d) + (1-\alpha) \times \text{staticScore}(d) \]
- Furthermore, assume that text scores are bounded by some maximum MTS (max. text score)
**Early Termination in DAAT Evaluation**

- Suppose that the score of each document is a linear combination of its query-dependent text score and its static score:
  \[ \text{score}(d) = \alpha \text{textScore}(d) + (1-\alpha) \text{staticScore}(d) \]
- Furthermore, assume that text scores are bounded by some maximum MTS.
- One can terminate evaluation after document \( k \) if the score of the \( r \)'th best document in the min-heap is greater than:
  \[ \alpha \text{MTS} + (1-\alpha) \text{staticScore}(k) \]
- Unlike in TAAT, the \( r \) returned documents will have their correct and final scores and so their relative ordering will be correct
  - Result counting, though, will not be correct
  - Search engine results counts, for all engines, are notoriously unreliable

---

**WAND and the Two-Level Retrieval Process**

[Broder, Carmel, Herscovici, Soffer, Zien 2003]

- Setting: document-at-a-time evaluation of top-\( r \) query in an additive scoring model
  - Score of a document sums over terms and other signals
- Full, exact scoring of a document is expensive
- First level evaluation: quickly establish whether a document merits to be fully evaluated
  - i.e. whether it has any chance of being a top-\( r \) candidate
  - No false negatives: cannot just throw away potential matches
  - As few false positives as possible: don’t want to pay the cost of the expensive full evaluation
- Second level is the costly full evaluation
WAND: Weighted AND

- Let the query contain terms $t_1, \ldots, t_k$
- Let $w_1, \ldots, w_k$ be non-negative term weights
- Let $x_{i,d}$ be a boolean indicator of the existence of $t_i$ in document $d$

$$WAND(q,d,\alpha) = \text{Ind}(\sum_{i=1}^{k} w_i x_{i,d} \geq \alpha)$$

- When $w_1 = \ldots = w_k = 1$, $\alpha = 1$: WAND reduces to OR
- When $w_1 = \ldots = w_k = 1$, $\alpha = k$: WAND reduces to AND

- For a given set of weights, as $\alpha$ is increased, WAND intuitively becomes harder to satisfy
- Can be adapted to include bounded query-independent additive score factors

Two-Level Evaluation Using WAND, View from 30K Feet

1. Set the weight of each term to its maximal possible contribution to the score, and set $\alpha$ to $\varepsilon > 0$
2. Quickly find next document $d$ who may satisfy $WAND(q,d,\alpha)$ – a candidate for full evaluation
3. Fully evaluate $d$, attempt insert into min-heap of size $r$
4. Set $\alpha$ to the min score of the heap
   - Whenever $d$ succeeded in entering heap, $\alpha$ grows
**WAND Flavor of Zig-Zag Join**

- For each term t: ct.init(); candidate ← 0; α ← ε
- Repeat
  - Sort term cursors by increasing position of cursor
  - pivot ← min cursor such that cumulative weighted sum ≥ α
  - If (pivot doesn’t exist or is at ∞) candidate ← ∞
  - If (pivot ≤ candidate):
    - NextBeyond(rarest term preceding or at pivot, candidate)
  - Else // pivot > candidate
    - If (first cursor by order is at pivot): // WAND is true
      - candidate ← pivot
      - Score candidate, enter into min-heap, update α
    - Else NextBeyond(rarest term lagging behind pivot, pivot-1)
- Until (candidate == ∞)
- Output the top-r documents in the min-heap