Introduction to the Social Web
Content Search, Recommendation and Exploration

Sihem Amer-Yahia
Behrooz Omidvar-Tehrani
CNRS/LIG
Nov 19th, 2014
Course Outline

• Nov 19th, 2014
  – Brief history of Recommendations: 1 hour
  – Blending Search and Recommendation: 2 hours
    • Hotlists Recommendation on Delicious
    • Top-k Algorithms
    • Network-aware Search on Collaborative Tagging Sites

• Nov 26th, 2014
  – User Studies on the Social Web: 1 hour
    • Group Recommendation
  – Social Content Exploration: 2 hours
Top-k Algorithms with Applications to Collaborative Tagging Sites

• Top-k processing
  – What it is and why we need it
  – Fagin algorithm (FA)
  – Threshold algorithm (TA)
  – No random access algorithm (NRA)
What is *top-k* processing?

- **Find $k$ items that best answer a user’s query**
  - As a set, as a sorted list, as a sorted list with scores
  - Usually from among $N$ items, where $N >> k$

- **Application domains**
  - Web search & other document retrieval / ranking tasks
    - Find documents about “Massachusetts election health care”
  - Search over multimedia repositories
    - Find red images that show a palm tree and a sunset
  - Search over structured datasets with user-defined preferences
    - Find large apartments in a good school district in Brooklyn
  - Many others….

- **Compared to SQL querying**
  - Relevance *to a degree*, not Boolean
  - Return only the best items, not all items
  - Quality of an item is expressed by a score
Why do we care about top-k processing?

- Many practical applications

- Representative of many data management problems
  - Solid application scenarios, and new emerging every day
    - We’ll talk about how top-k is used for social search
  - Variety of algorithmic approaches
  - Explores the *trade-off between run-time performance and space overhead*
Ranking functions

• Ranking (scoring) functions are used to compute the score of an item.
• Item $R(x_1, ..., x_m)$, where $x_i$ are the *ranking attributes*, e.g. degree of redness in an image, square footage of a house, number of times “Massachusetts” occurs in the text, etc.
• $\text{score}(R) = g\left(f_1(x_1), ..., f_m(x_m)\right)$
  - where $f_i$ are *monotone functions*, e.g. $f(x) = 2 \times x$
  - $g$ is a *monotone aggregation function*, e.g. sum, average, max
  - e.g. $\text{score}(i) = 2 \times \text{sq. ft.} + 3 \times \text{quality of school district}$

**Definition**

A function $f$ is monotone if

$f(x) \leq f(y)$ whenever $x \leq y$

An aggregation function $g$ is monotone if

$g(R) \leq g(R')$ whenever $R. x_i \leq R'. x_i$, for all $i$
Top-K Algorithm Performance

• **Execution time**
  – Sequential access (SA)
    • accessing items in order, e.g. by reading from a cursor
    • similar concept to a sequential disk read, where seek time is amortized over multiple accesses
  – Random access (RA)
    • accessing items out of order, e.g. a primary key lookup
    • similar to a random disk read
    • typically more expensive than an SA (even orders of magnitude), sometimes impossible
  – Why not use wall clock time?

• **Buffer size**
  – How much state do we have to keep during computation
  – Is the size bounded by some constant (e.g. $k$), or is it linear in the size of the dataset ($N$)? (recall that $k << N$)
Naïve Computation of top-k Answers

• Algorithm
  – Compute the score of each item
  – Sort items in decreasing order of score
  – Return $k$ items with the highest score

• Example 1 (on the board & next slide):
  $R \ (\text{id}, \text{annual income, net worth})$
  $\text{score}(r) = r.\text{income} + r.\text{net worth}$

• Properties of naïve solution
  – Advantage - simple
  – Disadvantage - unacceptable run-time performance when $N$ is high

• Idea: throw space at the problem
  – pre-compute inverted lists for components of the score
  – aggregate partial scores at run-time
Example 1

| id  | income (K$) | net worth (K$) | score  
= income + net worth |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁</td>
<td>150</td>
<td>350</td>
<td>500</td>
</tr>
<tr>
<td>r₂</td>
<td>150</td>
<td>425</td>
<td>575</td>
</tr>
<tr>
<td>r₃</td>
<td>125</td>
<td>450</td>
<td>575</td>
</tr>
<tr>
<td>r₄</td>
<td>100</td>
<td>450</td>
<td>550</td>
</tr>
<tr>
<td>r₅</td>
<td>100</td>
<td>200</td>
<td>300</td>
</tr>
<tr>
<td>r₆</td>
<td>80</td>
<td>100</td>
<td>180</td>
</tr>
<tr>
<td>r₇</td>
<td>75</td>
<td>500</td>
<td>575</td>
</tr>
<tr>
<td>r₈</td>
<td>75</td>
<td>50</td>
<td>125</td>
</tr>
<tr>
<td>r₉</td>
<td>50</td>
<td>300</td>
<td>350</td>
</tr>
<tr>
<td>r₁₀</td>
<td>50</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>
The Basic Indexing Structure (sorted): Inverted List

<table>
<thead>
<tr>
<th>id</th>
<th>income</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>150</td>
</tr>
<tr>
<td>$r_2$</td>
<td>150</td>
</tr>
<tr>
<td>$r_3$</td>
<td>125</td>
</tr>
<tr>
<td>$r_4$</td>
<td>100</td>
</tr>
<tr>
<td>$r_5$</td>
<td>100</td>
</tr>
<tr>
<td>$r_6$</td>
<td>80</td>
</tr>
<tr>
<td>$r_7$</td>
<td>75</td>
</tr>
<tr>
<td>$r_8$</td>
<td>75</td>
</tr>
<tr>
<td>$r_9$</td>
<td>50</td>
</tr>
<tr>
<td>$r_{10}$</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>net worth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_7$</td>
<td>500</td>
</tr>
<tr>
<td>$r_3$</td>
<td>450</td>
</tr>
<tr>
<td>$r_4$</td>
<td>450</td>
</tr>
<tr>
<td>$r_2$</td>
<td>425</td>
</tr>
<tr>
<td>$r_1$</td>
<td>350</td>
</tr>
<tr>
<td>$r_9$</td>
<td>300</td>
</tr>
<tr>
<td>$r_5$</td>
<td>200</td>
</tr>
<tr>
<td>$r_6$</td>
<td>100</td>
</tr>
<tr>
<td>$r_8$</td>
<td>50</td>
</tr>
<tr>
<td>$r_{10}$</td>
<td>50</td>
</tr>
</tbody>
</table>
Fagin Algorithm (FA)

- **Algorithm**
  - Access all lists sequentially (SA), in parallel
  - STOP once $k$ items have been seen in all lists
  - Compute scores of incomplete items by performing a random access (RA)
  - Sort on score, return the best $k$ items

- **Work out Example 1 on the board for $k = 3$**

- **Performances**
  - Number of accesses: 7 SA + 3 RA
  - Size of buffer: 5 objects in buffer

- **Is this algorithm correct?**
Threshold Algorithm (TA)

• **Algorithm**
  – Access all lists sequentially (SA), in parallel
  – After each cursor move
    • Compute the score of the item $r$ under the cursor with random accesses (RA)
    • Record $r$ in the buffer if
      – (i) buffer size < $k$
      – (ii) $r$ ‘s score > $k^{th}$ score, remove $k^{th}$ item from buffer
    • Update the *threshold* $\theta = \sum$ current list scores
    • STOP when $k^{th}$ score $\geq \theta$
  – Return the $k$ items currently in the buffer

• **Work out Example 1 on the board for $k = 3$**

• **Performance**
  – 4 SA + 4 RA; #RA = #SA * (m-1), where m is the number of lists
Comparison between FA and TA

• Theorem: \# SA in TA \leq \# SA in FA

• Theorem: TA requires only bounded buffers, FA buffers are unbounded
What if we couldn’t do random accesses?

- Sometimes it suffices to output the top-k as a set
- Sometimes we can get away with outputting top-k in sorted order, but with no scores
No Random Access Algorithm (NRA)

**Algorithm**
- Access all lists sequentially (SA), in parallel
- After each cursor move compute
  - Worst-case score $W(r)$, best-case score $B(r)$ for each seen $r$
  - Sort all seen items on $W(r)$, breaking ties by $B(r)$
  - $\theta = \sum$ current list scores (this is the best-case score of any unseen object)
  - STOP when $W(r)$ of $k^{th}$ object $\geq \theta$
- If random accesses are possible, compute complete scores of the top-$K$ items
- Return the top-$k$ items

**Work out Example 1 on the board for $k = 3$**

**Performance**
- 13 SA + 0 RA
- Optimal performance if no RAs are allowed
- In reality, computation may be slow -- re-sorting potentially large buffers at each step