Efficient Query Processing in Web Search Engines

Simon Lia-Jonassen
UiO, 12/04/16
Outline

- Introduction and Motivation
- Query Processing and Optimizations
- Caching
- Parallel and Distributed Query Processing
- Hardware Trends and Their Impact
A4 page is \(~0.005\) cm thick. 3000 drawers \(\times 304.8\) cm/drawer \(\times 200\) pages/cm gives \textbf{182.9 million} pages. A4 page fits \(350 - 400\) words.

Indexed Web contains at least \textbf{4.57 billion} pages. An avg. page is \(478\) words.

Google has more than \(1\) billion global visitors and annually spends billions of USD on data centers.
Web-Search in a Nutshell

- Query processing basics
  - Given an indexed collection of textual documents \( D \) and a textual query \( q \), find the \( k \) documents with largest similarity score.

- Some similarity model alternatives
  - Boolean model.
  - Vector space models, e.g., TFxIDF, Okapi BM25.
  - Term proximity models.
  - Link analysis models, e.g., PageRank, HITS, Salsa.
  - Bayesian and language models.
  - Machine learned models.
Effectiveness (quality and relevance)

- Boolean, not ranked
  - Precision – how many selected items are relevant.
  - Recall – how many relevant items are selected.
  - F1 – weighted harmonic mean of precision and recall.

- Boolean, ranked
  - Precision and recall at 5, 10, …k.
  - Average Precision and Mean Average Precision.

- Graded, ranked
  - Cumulative Gain (CG), Discounted CG (DCG) and Normalized DCG (NDCG).
  - Mean Reciprocal Rank (MRR).
Efficiency (volume, speed and cost)

- Index and Collection Size
  - Number of documents and document size.
  - Index granularity, size and freshness.

- Infrastructure Cost
  - Number of machines, memory/disk size, network speed, etc.
  - Resource utilization and load balancing.

- Queries and Time
  - Latency – time per query.
  - Throughput – queries per time unit.
  - Query volume (offline) and arrival rate (online).
  - Query degradation rate (online).
  - Result freshness (cached results).
D1: Because I'm Slim Shady, yes I'm the real Shady.

D2: All you other Slim Shadys are just imitating,

D3: So won't the real Slim Shady please stand up,

D4: Please stand up, please stand up?

Stopwords: all are because i'm just other please so the up won't yes you

Query Processing with Inverted Index

Inverted Index:
- imitate 1:1
- real 0:1 2:1
- shady 0:2 1:1 2:1
- slim 0:1 1:1 2:1
- stand 2:1 3:2

Query: Real Slim Shady

Results: D1 D3 D2
Query Processing with Inverted Index

We could use $k_1 = 1.2$, $b = 8$, $k_3 = 0.75$, but note that IDFs would be quite strange in this particular example.
Posting Lists (doc. ids and frequencies)

- Usually represented as one or two integer lists ordered by document id.
  - Alternatives:
    - Bitmaps instead of document ids, weights instead of frequencies, or impact ordered lists.

- For better compression document ids are replaced by differences (deltas), and the ids can also be remapped/reordered to minimize resulting deltas.

- Usually accessed via an iterator (get and next) and an auxiliary index/set of pointers can be used to skip forward in the inverted file.

Original postings:
\[
\langle 5, 1 \rangle \langle 8, 1 \rangle \langle 12, 2 \rangle \langle 13, 3 \rangle \langle 15, 1 \rangle \langle 18, 1 \rangle \langle 23, 2 \rangle \langle 28, 1 \rangle \ldots
\]

Encoded postings:
\[
\langle \langle 5, o_2 \rangle \rangle \langle 5, 1 \rangle \langle 3, 1 \rangle \langle 4, 2 \rangle \langle \langle 13, o_3 \rangle \rangle \langle 1, 3 \rangle \langle 2, 1 \rangle \langle 3, 1 \rangle \langle \langle 23, o_4 \rangle \rangle \langle 5, 2 \rangle \langle 5, 1 \rangle \ldots
\]
Posting List Compression

- **Bit-Aligned Methods**
  - E.g., Unary, Elias, Golomb, Rice, Interpolative, etc.
  - Space efficient but require less efficient bit arithmetic.

- **Byte-Aligned Methods**
  - E.g., VByte (VB)
    - 1 : 1000 0001
    - 13 : 1000 1101
    - 133 : 0000 0001 1000 0101
    - 1337 : 0000 1010 1011 1001
  - Fast but less space efficient.

<table>
<thead>
<tr>
<th>Value</th>
<th>Unary</th>
<th>Elias-γ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10 0</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>10 1</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>110 00</td>
</tr>
<tr>
<td>5</td>
<td>11110</td>
<td>110 01</td>
</tr>
<tr>
<td>6</td>
<td>111110</td>
<td>110 10</td>
</tr>
<tr>
<td>7</td>
<td>1111110</td>
<td>110 11</td>
</tr>
<tr>
<td>8</td>
<td>11111110</td>
<td>1110 000</td>
</tr>
<tr>
<td>9</td>
<td>111111110</td>
<td>1110 001</td>
</tr>
<tr>
<td>10</td>
<td>1111111110</td>
<td>1110 010</td>
</tr>
</tbody>
</table>
Posting List Compression

- Word-Aligned Methods
  - E.g., Simple9 (left) and NewPFoR (right)

<table>
<thead>
<tr>
<th>Selector</th>
<th>Number of codes</th>
<th>Code length (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>g</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>h</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>28</td>
</tr>
</tbody>
</table>

- Branch-free and superscalar, can be tuned for compression/speed.
- In the rest, we consider NewPFoR with chunks of 128 deltas followed by 128 frequencies.
  - Alternatives: 32/64 or variable size chunks.
Posting List Layout with Skips

Logical View

Physical View

Current State - docID's:
- SC1#0: 3401, 7293, ..., 4350801
- SC0#0: 273, 553, 831, ..., 3401
- DC#2: 554, 587, 591, ..., 831
Query Evaluation and Pruning

- **Disjunctive (OR) and Conjunctive (AND) Evaluation**
  - OR – The documents that match any query term will be considered.
    - Slower, lower precision, higher recall.
  - AND – Only the documents that match all query terms will be considered.
    - Faster, higher precision, lower recall.

- **Static Pruning – aka Index Pruning**
  - Term Pruning – remove less important terms from the index.
  - Document Pruning – remove less important documents from the index.

- **Dynamic Pruning – aka Early Exit Optimization**
  - Avoid scoring postings for some of the candidates under evaluation.
    - Safe pruning – guarantees the same result as a full evaluation (score, rank or set safe).
    - Unsafe pruning – does not guarantee the same result as a full evaluation.
Term- vs Document-At-A-Time Evaluation

- **Term-At-A-Time (DAAT)**
  - Score all postings in the same list and merge with the partial result set (*accumulators*).
  - Need to iterate only one posting list at a time, but store all possible candidates.

- **Document-At-A-Time (DAAT)**
  - Score all postings for the same document and add to the result head.
  - Need to iterate all posting lists at once, but store only the $k$ best candidates.
Term-At-A-Time Evaluation (TAAT)

- **AND** – Evaluate query in conjunctive mode, starting with shortest posting list.
- **Quit/Continue** – Stop adding new accumulators when a target size is reached.
- **Lester** – Dynamically scale a frequency threshold based on the accumulator set size.
- **MaxScore** – Ignore postings and remove candidates for documents that are guaranteed not to be in the result set (need to track the $k$ best candidates seen so far).

**Some optimizations**
Some optimizations

- And – Don’t score unless all pointers align, skip to the largest document id.
- MaxScore – Stop candidate evaluation when it is guaranteed not to be in the result set.
- WAND – Similar to MaxScore but a bit different (see the next slide).
DAAT MaxScore and WAND

**MaxScore**
- Evaluate lists in maxScore order.
- Don’t advance optional term pointers.
  - Must match at least one term with maxAg >= thres.
- Discard when score + maxAg < thres.

**WAND**
- Evaluate lists in the current document id order.
- Pivot by first pointer with maxAg >= thres.
- If first pointer and pivot are equal, evaluate. Otherwise, skip using the pivot.

- Both methods can be improved with block-wise maximum scores.
Post-processing and Multistep Processing

- **Traditional post-processing**
  - Generate snippets.
  - Retrieve page screenshots.
  - Result clustering and/or faceting.

- **Two-phase processing**
  1. Use a simple ranking model to generate a pool of candidates.
  2. Refine the results using a more expensive model.
     - E.g., machine learned model, phrase matching, etc.

- **Query refinement**
  - Suggest another query based on the found results.
Caching

- Two-level (Saraiva et al.)
  - Result cache and posting list cache.
- Three-level (Long and Suel)
  - Adds intersection cache.
- Five-level (Ozcan et al.)
  - Adds document cache and score cache.
- Other alternatives
  - Projections instead of intersections.
  - Blocks instead of posting lists.
- Static, dynamic and hybrid caches
  - Static – rebuilt once in a while.
  - Dynamic – updated on each access.
  - Static-dynamic – splits the cache into a static part and a dynamic part.
Some of the most common methods

- **LRU** – put newly accessed entries into a queue, discard from the tail.
- **LFU** – each entry has a counter, discard entries with lowest count.
  - Other alternatives: Two-Stage LRU, LRU-2, 2Q, Multi-Queue (MQ), Frequency-Based Replacement (FBR), Adaptive Replacement Cache (ARC).
- **Cost** – evict entries with the lowest processing cost.
- **Landlord**
  - assign $H = \text{cost/size}$ on insertion or hit
    - cost = 1 gives most hits, cost = proc. time gives best avg. latency.
  - evict the one with $\min H$ and subtract it from remaining $H$.
  - Improvement: use $H + \alpha \times \text{remaining } H$ on the first renewal, and $\alpha'$ on the second and subsequent renewals.
Freshness with Infinite Result Cache

- **Motivation**
  - We can cache results infinitely but they will become outdated after a while.

- **A solution and the improvements**
  1. Set a TTL to invalidate the results after a while.
     - Always rerun the expired queries on request.
  2. Refresh cache results when have capacity.
     - May improves average result freshness.
  3. Proactively refresh expensive queries that are likely to be expired at peak-load hours.
     - May improves result freshness, degradation and latency, but determining such queries is really hard.
Offline
- All queries are available at the beginning.
- Query throughput must be maximized, individual query latency is unimportant.
- Specific optimizations
  - Query reordering, clairvoyant intersection and posting list caching, etc.

Online
- Queries arrive at different times and the volume varies during the day.
- Throughput must be sustained and individual query latency minimized.
- Specific optimizations
  - Degradation, result prefetching, micro-batching, etc.
Parallel Query Processing

- **Motivation**
  - Modern CPUs are multi-core and we would like to utilize this in the best possible way.

- **Inter-query concurrency**
  - Assign different queries to different cores.
  - Improves throughput, affects latency.

- **Intra-query concurrency**
  - Assign different blocks to different cores.
  - Improves latency, affects throughput.

- **Some of the main issues**
  - Memory wall (I/O bottleneck).
  - CPU cache related issues
    - Coherence, conflicts, affinity, etc.
  - Amdahl's and Gustafson's laws.
Distributed Query Processing

- **Partitioning and Replication**
  - Add more partitions to scale with the collection size.
  - Add more replicas to scale with the query volume.

- **Clustering**
  - Group similar documents in clusters.
  - Build a different index for each cluster.

- **Tiering**
  - Split the collection into several parts, e.g.
    - 1 mil important documents.
    - 10 mil less important documents.
    - 100 mil not really important documents.
  - Decide when a query should fall through.
Partitioned Query Processing

- **Document-wise partitioning**
  - Each node has a subset of documents.
  - Each query is processed by all of the nodes in parallel, one of the nodes merges the results.
    - Pros: Simple, fast and scalable.
    - Cons: Each query is processed by all of the nodes.
    - Cons: Large number of posting lists to be processed.

- **Term-wise partitioning**
  - Each node has a subset of terms.
  - Posting lists are fetched and sent to a node that processes all of them.
    - Pros: Only a few nodes are involved.
    - Pros: Possibility for inter-query concurrency.
    - Pros: Only one posting list for each term.
    - Cons: Only one node does all processing.
    - Cons: Network load and load balancing are critical.
Partitioned Query Processing

- **Pipelined query processing (on top of TP)**
  - Route a bundle from one node to next, process on each node and extract the results on the last one.
  - **Pros**: The work is divided among the nodes.
  - **Pros**: The network load is reduced.
  - **Cons**: Sequential dependency and load balancing.

- **Possible improvements**
  - Do early exit to reduce the amount of transferred data.
  - Apply fragment based or semi-pipelined processing to reduce query latency.
  - Optimize term assignment to improve network load and load balancing.

- **Other techniques**
  - Hybrid partitioning – divide posting lists into chunks and distribute the chunks.
  - 2D partitioning – apply both document- and term-wise partitioning (m x n).
Multi-Site Query Processing

- **Motivation**
  - Search engines spanning across multiple datacenters open a range of new possibilities.

- **Some of the techniques**
  - User-to-center assignment – choose the nearest datacenter.
  - Corpus partitioning – cluster and assign documents geographically.
  - Partial replication – replicate only certain documents to other datacenters.
  - Reduce costs by forwarding requests to a remote datacenter at peak.
Impact of the Hardware Trends

- **2002**
  - Processor: Intel Pentium 4 (2GHz)
  - Main Memory: 4x512MB
  - Disk: 80GB HD
  - Status: not so fast =(

- **2016**
  - Processor: Intel Core i7 (4x2x4GHz++)
  - Main Memory: 4x8GB+
  - Disk: 512GB SSD
  - Status: fast!

**Comparison**

- **2002** vs **2016**:
  - Processor: 2GHz vs 4x2x4GHz++
  - Memory: 4x512MB vs 4x8GB+
  - Disk: 80GB HD vs 512GB SSD
CPU: From GHz to multi-core

- **Moore’s Law:**
  - \( \sim \) the number of transistors on an IC doubles every two years.
  - Less space, more complexity.
  - Shorter gates, higher clock rate.

- **Strategy of the 80s and 90’s:**
  - Add more complexity!
  - Increase the clock rate!

- **Pollack’s Rule:**
  - The performance increase is \( \sim \) square root of the increased complexity. [Borkar 2007]

- **The Power Wall:**
  - Increasing clock rate and transistor current leakage lead to excess power consumption, while RC delays in signal transmission grow as feature sizes shrink. [Borkar et al. 2005]
Instruction-level parallelism

- Pipeline length: 31 (P4) vs 14 stages (i7).
- Multiple execution units and out-of-order ex.:
  - i7: 2 load/store address, 1 store data, and 3 computational operations can be executed simultaneously.
- Dependences and hazards:
  - Control: branches.
    - Dean 2010: a branch misprediction costs ~5ns
  - Data: output dependence, antidependence (naming).
  - Structural: access to the same physical unit of the processor.
- Simultaneous multi-threading ("Hyper-threading"):
  - Duplicate certain sections of a processor (registers etc., but not execution units).
  - Reduces the impact of cache miss, branch misprediction and data dependency stalls.
  - Drawback: logical processors are most likely to be treated just like physical processors.
## Computer memory hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>Latency</th>
<th>Size</th>
<th>Technology</th>
<th>Managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>&lt;&lt;1ns</td>
<td>1KB</td>
<td>CMOS</td>
<td>Compiler</td>
</tr>
<tr>
<td>L1 Cache (on-chip)</td>
<td>&lt;1ns</td>
<td>4x32KBx2</td>
<td>SRAM</td>
<td>Hardware</td>
</tr>
<tr>
<td>L2 Cache (off-chip)</td>
<td>2.5ns</td>
<td>4x256KB</td>
<td>SRAM</td>
<td>Hardware</td>
</tr>
<tr>
<td>L3 Cache (shared)</td>
<td>5ns</td>
<td>8MB</td>
<td>SRAM</td>
<td>Hardware</td>
</tr>
<tr>
<td>Main Memory</td>
<td>50ns</td>
<td>4x8GB+</td>
<td>DRAM</td>
<td>OS</td>
</tr>
<tr>
<td>Solid-State Drive</td>
<td>&lt;100µs</td>
<td>512GB-</td>
<td>NAND Flash</td>
<td>Hardware/OS/User</td>
</tr>
<tr>
<td>Hard-Disk Drive</td>
<td>3-12ms</td>
<td>1TB+</td>
<td>Magnetic</td>
<td>Hardware/OS/User</td>
</tr>
</tbody>
</table>

(1ms = 1 000 µs = 1 000 000 ns; 1ns = 4 clock cycles at 4GHz or 29.8cm of light travel)
Some of the main challenges of CMP:
- Cache coherence.
- Cache conflicts.
- Cache affinity.

Other important cache-related issues:
- Data size and cache line utilization.  
  - i7 has 64B cache lines.
- Data alignment and padding.
- Cache associativity and replacement.

Additional memory issues:
- A large span of random memory accesses may have additional slowdown due to TLB misses.
- Some of the virtual memory pages can also be swapped out to disk.
Cache- and processor-efficient query processing

- Modern compression methods for IR:
  - S9/S16, PFOR/NewPFD, etc.
  - Fast, superscalar and branch-free.
  - Loops/methods can be generated by a script.

- While compression works on chunks of postings, processing itself remains posting-at-a-time.

- So what about:
  - Branches and loops?
  - Cache utilization?
  - ILP utilization?

- Some interesting alternatives and trade-offs:
  - Term vs document-at-a-time processing.
  - Posting list iteration vs random access.
  - Bitmaps vs posting lists.

code: https://github.com/javasoze/kamikaze
Acknowledgements

- For details and references see Chapter 2 of my PhD thesis:
  - “Efficient Query Processing in Distributed Search Engines” – http://goo.gl/vDNGGb

- Also a great tutorial by Barla Cambazoglu and Ricardo Baeza-Yates:
  - “Scalability And Efficiency Challenges In Large-Scale Web Search Engines” – http://goo.gl/oyWDqU

- A very good book:
  - “Information Retrieval: Implementing and Evaluating Search Engines” – Büttcher et al., The MIT Press, 2010

- Any publication co-authored or even cited by these guys:
Thank you!

We're done.
Questions?

simon.jonassen@gmail.com