Segmented Inverted Indices

- Large-scale search engines have huge indices, which are distributed across multiple machines
  - Addresses mainly data scale; usage scale (query throughput) is mostly addressed by replication
- Two basic architectures:
  - Local index organization - index partitioned by documents. Each machine inverts a disjoint set of documents
  - Global index organization - index partitioned by terms. Each machine holds postings lists for a disjoint set of terms
- Query processing becomes a distributed task, where the choice of the partitioning scheme influences the query processing algorithm
**Segmented Inverted Indices**

- **Doc 1**: A, B, C
- **Doc 2**: A, B, D
- **Doc 3**: A, C, D
- **Doc 4**: B, C, D

**Global index organization**: index partitioned by terms.

**Segment 1**
- A: 1, 2, 3
- B: 1, 2, 4

**Segment 2**
- C: 1, 3, 4
- D: 2, 3, 4

---

**Global Index - Preprocessing**

- Multiple crawler nodes bring in new content which needs to be parsed and tokenized.
- Parsing and tokenization is CPU bound, so to keep up with many crawlers one needs multiple P/T servers.
- The tokens of each parsed document are distributed to the various index segments by some function $f: \{\text{terms}\} \rightarrow \{\text{segments}\}$. 

\[ \{t_1, t_2, t_3, t_4, t_5\} \rightarrow \{\text{Index Segment}\} \]

All-to-all communication according to $f(\text{term})$. 

---

"Web" Crawler P/T P/T P/T All-to-all communication according to $f(\text{term})$ Index Segment Index Segment Index Segment Index Segment 

$\{t_1, t_2, t_3\}$ $\{t_4, t_5\}$
Global Index - Runtime

- Given a query \( q = \{ t_1, t_2, \ldots, t_k \} \), a centralized server (QI – Query Integrator) computes \( f(t_i) \) for all \( i = 1, \ldots, k \).
  - That determines the set of servers that need to be involved in query evaluation, \( S_1, \ldots, S_m \) (certainly \( m \leq k \)).

- Invariably, accumulators for the partial result sets stemming from each server will need to be passed around in the system.
  - Hence term-at-a-time evaluation fits this partitioning scheme.

Global Index – Runtime (cont.)

Multiple query partitioning schemes:

- Serial: QI sends the query to \( S_1 \), gets back the accumulators, forwards them to \( S_2 \), then to \( S_3 \), etc. When \( S_m \) returns the accumulators, they can be finalized (normalized and ranked, summarized and site collapsed).

- Parallel: QI sends the query to \( S_1, \ldots, S_m \), receives all \( m \) sets of accumulators, and aggregates them.

- Pipeline: upon finishing its computation, \( S_1 \) forwards the accumulators directly to \( S_2 \), which aggregates its stuff and forwards to \( S_3 \), etc. \( S_m \) returns the aggregates to QI for finalization.
Global Index – Serial Query Evaluation

- To minimize inter-machine communication, the order in which the servers are contacted depends on the frequencies of the terms which they serve.
  - This either requires a global lexicon at the QI, or a preliminary round of communication in which the QI queries the servers for DF values.

Global Index – Pipelined Query Evaluation

- Less communication than in the serial approach.
  - But servers and LAN need to be more complex.
Global Index – Parallel Query Evaluation

- For conjunctive queries, this scheme may increase inter-machine communication but lower latency; for disjunctive queries, it decreases inter-machine communication.
- In either case, this scheme increases the computational load on the QI (can be addressed via replication).

The Function $f: \{\text{terms}\} \rightarrow \{\text{segments}\}$

- Ideally, the function spreads terms so that the query load on each index partition is approximately equal:
  - Load depends primarily on number of queries requesting each term, and on length of each term’s postings list.
  - In a periodic batch-build mode, we can assume we know those parameters – e.g. from the previous generation of the index and some analysis of the query log.
- Load balancing the servers is theoretically NPC even for two servers (equivalent to partitioning a set of integers into sets of equal sums).
- In practice, due to the power-law distributions of both terms in natural language and terms in queries, there are enough “small blocks to play with” so that balancing can be done.
  - But with such a fine-grained $f()$, lookup may be costly (involve disk I/O).
Additional Optimization Schemes

- Add postings lists of common query terms to all segments, increasing the chances of a single server being able to serve a query all by itself.
  - Also helps in load balancing single term queries.
- Refinement: identify pairs of frequently co-occurring terms in queries, and adopt one of the following policies:
  - Assign such term pairs to the same server.
  - Cross-replicate the postings lists of the pair on both servers.
  - Create a postings list representing the intersection of the two terms.
- In either case, each server will probably set aside some small percentage of its capacity for replicated postings lists.
- Computing f(\cdot) and planning query evaluation is now more complex since f(\cdot) is 1-to-many and there are multiple possible evaluation plans.

Segmented Inverted Indices

Local index organization:
index partitioned by documents
Local Index - Preprocessing

- Crawlers bring in new documents, which are sent to the various segments according to a function $g: \{\text{documents}\} \rightarrow \{\text{segments}\}$.
- Each segment parses, tokenizes, inverts and serves its own (disjoint) set of documents.

The Function $g: \{\text{documents}\} \rightarrow \{\text{segments}\}$

- Ideally, the function spreads documents so that the query load on each index partition is approximately equal.
  - Query latency depends on the latency of the slowest partition.
  - Load depends on the number of documents in each segment that match the query, and on the size of the index on each segment.

- An easy way to achieve good load balancing is by distributing documents uniformly at random between the segments.
  - E.g., by basing $g()$ on the hash value of each URL.

- Next slide assumes that documents are assigned uniformly at random (u.a.r) to segments.
Local Index - Runtime

To guarantee consistent rankings across the segments, the QI either maintains a global lexicon with document frequencies or performs a preliminary round of communication to determine the DFs of all terms.

Tuning of Retrieval Parameters

- The user expects n results back; the QI asks each of the m segments to retrieve its top-k results.
- What is the relationship between n and k? Does m play a part?
- If k≥n, the QI can guarantee that the km results it collects from the segments will indeed contain the true top-n results.
  - Also, the QI can effectively prefetch additional results and cache them for potential follow-up queries by the user.
  - However, especially for large values of m, this seems excessive.
- If k<n, there is a probability (that depends on m) that some of the true top-n results will not be obtained by the QI.
  - Can this probability be computed, and a “reasonable” value of k determined?
**k<n: Problem Demonstration**

For \( m=5, n = 10 \) – what happens if we set \( k=3 \) and collect 15 results?

![Diagram of top three results from each of the five segments](image)

Fetching the top three results from each of the five segments fails to collect all top 10 results!

---

**Additional Load Balancing and Scale-Out by Replication**

- In either index partitioning scheme, a single array of partitions cannot handle the query loads seen by today’s Web search engines.
- Solution: replicate the array (including QI nodes), effectively creating a matrix of servers; Furthermore, place replicas in different geo regions.
  - This achieves resiliency against inevitable hardware and network failures, as well as reduces latency experienced by users.
  - However, maintaining consistent replicas is difficult.
- Load balancing software directs work to the least-loaded copy of each server/QI, also taking into account network latency.
Tiering of Documents

- Each tier is a distributed index
- Orthogonal to the question of distribution scheme

Fall-Through Logic

Continue to query the next tier if:
- Number of results so far is below threshold
- Score of n'th result so far is below threshold
- Number of different sites represented in results is below a threshold
- ...
### Possible Tier Definitions

- Pages with high static scores
  - Title, heading and anchor-text tokens of other (crawled) pages

- Pages with medium static scores (body tokens)

- Pages with low static scores (body tokens)
  - Anchor-only documents
  - Pages that we suspect are outdated

### Tiering: Engineering Tradeoffs

- The higher the tier, the larger the query rate it experiences
- Hence, higher tiers should sustain higher throughputs

Need to determine:

1. Size of index on each node of tier $j$ (the smaller the index, the higher its throughput)
2. Size of a single tier-$j$ replica (fan out of QI)
   - Together with (1), determines size of tier-$j$
3. Replication factor of tier $j$
Comparing between Global and Local Index Architectures

- Inter-machine communication: just a few machines involved per query in a global index, as opposed to all machines in a local index
  - Large QI fan-out in the local scheme introduces large latency variance
- Communication between the QI and each machine is smaller in local indices, since only small result sets are passed rather than potentially large postings list
- Evaluation scheme: global indices favor term-at-a-time evaluation; local indices can use either TAAT of document-at-a-time evaluation
- Load balancing: simpler in the local index scheme
- Due to the advantages of document-at-a-time evaluations for Web search, and the simplicity of the local index organization, large Web search engines predominantly used the local index partitioning scheme in the first decade of the 21st century

Revisiting the Tradeoffs Between Local and Global Segmented Indices

- Observation: off-page ranking indicators for head Web content is very dynamic (links, anchors, tweets/likes/shares, clicks). However, for tail content, off-page indicators are few and much more stable
- Pain points in very large locally-distributed indexes:
  - The large fan-out of the QI introduces large variance in response time from the segments
  - One way to control latency is to hold the local indices in RAM
  - ...but this limits the number of documents each node can serve and increases hardware costs
- Economic drivers force Web-scale engines to use hard drives for some portions of the index – can’t just use RAM
Hardware Trends

- Hard disk drives costs are low – around $0.05 per GB (2013) – and are expected to continue decreasing in the foreseeable future.
- Solid state drives (flash), once costing x120 the cost of HDD, is now priced at x25 and is much faster.
- 10Gbit networks essentially enables remote storage to be as fast as local storage, and allows for larger inter-machine communication.
- The above trends have shifted performance tradeoffs and previous best practices should be reexamined.

Indexing in Maguru

- The following are defined as indexing units ("atoms") in Maguru:
  1. All n-grams up to a certain n
  2. Certain n-grams beyond length n
  3. Tuples: certain pairs of non-adjacent closely co-occurring words
- The selection process in 2,3 depends on corpus and query log statistics.
- Each atom’s postings list is composed of \{docid, score\} elements.
- Atoms are indexed on hard drives in a global distribution scheme.
  - Reminder: this applies to tail content; head content is still locally distributed.
- Maguru’s lexicon is hash-based and is spread across multiple layers of the storage hierarchy (RAM, SSD, HDD).
**Implications of Indexing Units**

- Having rich indexing units has the following consequences:
  - Many postings lists are short and can be read very quickly
  - The number of machines involved in evaluating queries is small; in particular, many queries may be evaluated by a single node
- Still, some postings lists are very long, so:
  - Tiering by score into three tiers is performed
  - The first two tiers always fit on single machines, but 3rd tier may not
  - 3rd tier can be distributed across multiple nodes in the segment
  - Query evaluation will cascade to use lower tiers if not enough results were found when using higher tiers
- Consequently: lower query latency & higher throughput, better scalability as the collection grows and QPS (queries/second) grows

---

**Optimizing the Partitioning of Data**

- Maguru’s indexing scheme attempts to partition data given access patterns, performance needs, and economic drivers
- Partitioning of data is called “sharding”
- Smart sharding is key to many “Big Data” applications
- See also “Sharding Social Networks”; Duong, Goel, Hofman and Vassilvitskii, WSDM’2013
Multi-Collection Federation

Multi-Collection Federation: Architecture

query → Query Interpretation → Source Selection & Dispatching

Web → Images → Weather → News → Video → Ads

Blending and Page Composition → Results page