Distributed Indexing for Semantic Search

Peter Mika

Researcher, Data Architect

Yahoo! Research

MapReduce

- MapReduce programming model
  - Map: convert input into (k,v) pairs, apply some transformation
  - Reduce: collect all values for the same key: (k, {v1,v2…}), apply some computation, output result
- Parallel computation and distributed data
  - Map and Reduce tasks run in parallel
    - Each mapper works on a part of the input
    - Each reducer works on a subset of the keys
  - Reduce starts only after all Mappers are finished
  - Distributed file system
- Open source implementation
  - Hadoop (Java)

Text indexing using MapReduce

- Information Retrieval relies on inverted indices of text
  - A map from terms to documents which contain that term
  - Queries are resolved by looking up each query term and merging the resulting document sets
- MapReduce is the perfect model for building inverted indices
  - Map creates (term, {doc1}) pairs
  - Reduce collects all docs for the same term: (term, {doc1, doc2…}
  - Skew is a known issue: reducers have uneven load
  - Sub-indices are merged afterwards (inexpensive)
  - See C. D. Manning, P. Raghavan, and H. Schütze. Introduction to Information Retrieval. Cambridge University Press, 2008.
- Implementations for building Lucene indices using Hadoop
  - Katta project

RDF indexing using MapReduce

- RDF data has a much richer structure
  - More expressive queries require more sophisticated indices
- Differences in semantic search literature as to what expressivity is required
  - Current, web search keyword queries have very limited structure (see Pound et al. WWW2010)
  - End users are unlikely to type in SPARQL queries
- Queries on property values are required in almost all cases
  - **foaf_name:Peter foaf_name:Mika**
  - **foaf_name:"Peter Mika"**
  - **foaf_name:"Peter Mika" foaf_age:32**

Post-fixing

- This can be achieved without any special index structure by ”post-fixing”
  - Instead of the term **Peter** index the term **Peter#foaf_name**
  - Prefix queries are needed to search only for Peter
- ✓ There is less skew
- ✗ Dictionary is number of unique terms per property
  - It works well when the number of properties are small
    - Example: NER indexing with a small number of types
  - RDF has large number of properties: dictionary explodes
    - **the_name**, **the_title**, **the_address, the_org**, ….

Horizontal indexing

- Two fields (indices): one for terms, one for properties
- For each term, store the property on the same position in the property index
  - Positions are required even without phrase queries
- Query engine needs to support the alignment operator
  - E.g. MG4J: **”Peter^property:foaf_name Mika^property:foaf:name”**

✓ Dictionary is number of unique terms + number of properties

▯ Occurrences is number of tokens * 2

✗ As much skew as in normal text indexing

Field	p1	p2	p3	p4
token	peter	mika	32	barcelona
property	foaf:name	foaf:name	foaf:age	vcard:location

Table 1: Horizontal indexing of RDF data

Vertical indexing

- One field (index) per property
- Positions are not required
  - But useful for phrase queries
- Query engine needs to support fields
  - E.g. MG4J: **“foaf_name:Peter foaf_name:Mika”**
- Dictionary is number of unique terms
- Occurrences is number of tokens
- Less skew

✗ Number of fields is a problem for merging, query performance

Field	p1	p2	p3	p4
foaf:name	peter	mika		
foaf:age	32			
vcard:location	barcelona			

Table 2: Vertical indexing of RDF data

Implementation

- Reality is more complex than the textbooks…
  - Hashed subject URIs using MG4J’s Minimal Perfect Hash
    - hash function occupies only 307MB
    - Fits in memory for now, only required for map
  - Implemented fields, positions: keys are (field, term) pairs, values are (docid, position) pairs
  - For each term, documents need to be indexed in increasing order of docid
    - Secondary sort by making value part of the key
    - Increased amount of data

Implementation

- Document frequency needs to be known when starting to write out the posting list
  - Introduced dummy occurrences, one for each document
  - Set docid to -1 to make sure dummy occurrences come first
- Memory problems with unbalanced executions (too many documents for a single term)
- Memory problems with large number of indices (index caching)
- Trade-offs between how much memory is left for our job vs. memory for the system itself

Evaluation

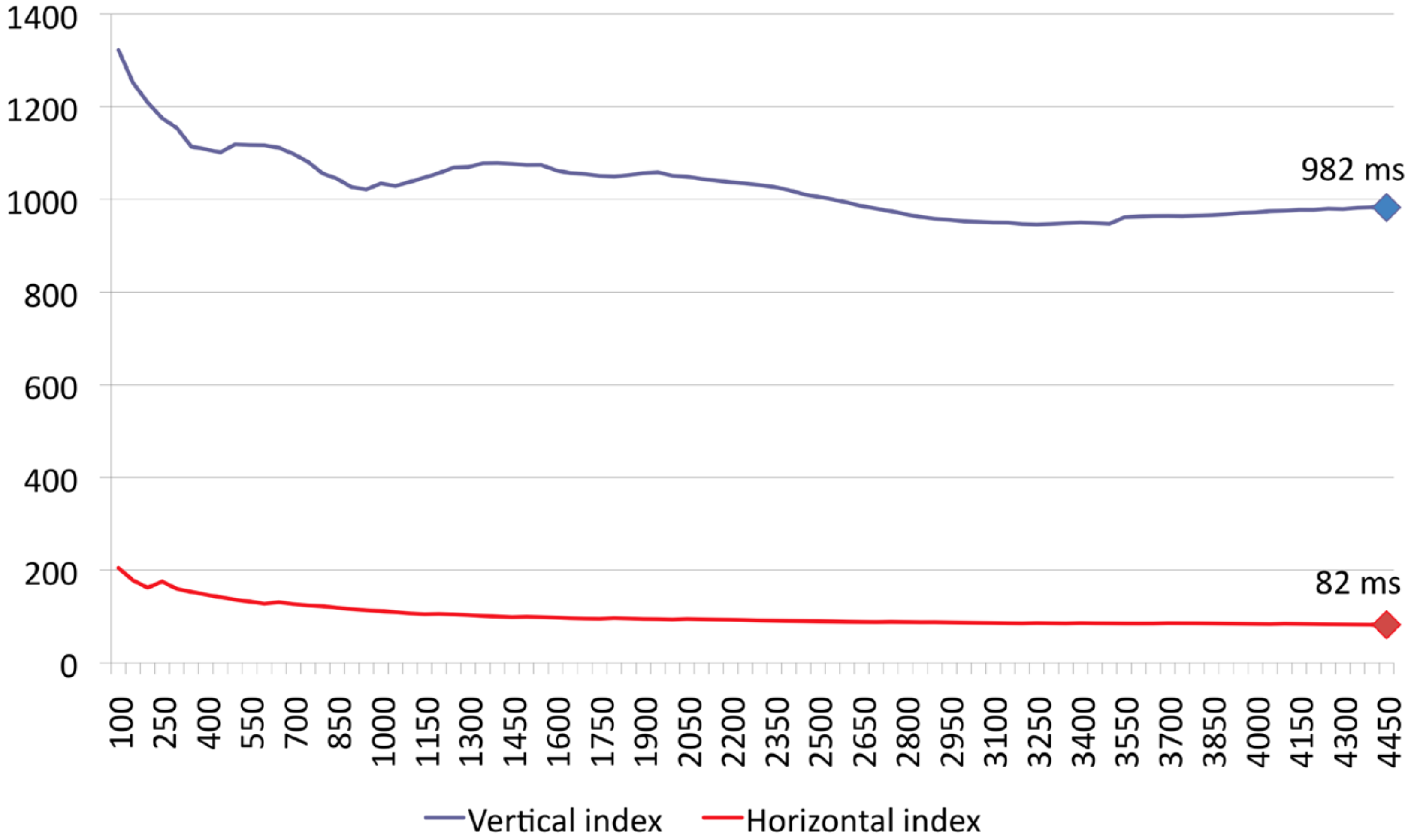
- BTC 2009 dataset (as used in Entity Search Track)
  - 114,530,196 URIs
  - 273,922,563 triples
  - 2,931,625,024 / 1,438,318,071 occurrences (horiz/vert)
- We index subject URIs, not document URLs
  - Collected triples by subject URI using Hadoop
- Only datatype-properties indexed
- Pre-selected 300 datatype-properties for vertical indexing
  - Problem: frequent but “useless” properties such as foaf:mbox_sha1sum or image:height
  - TODO: likelihood-to-match
- Measured indexing cost and query performance
  - TODO: factor in the cost of merging

Indexing cost

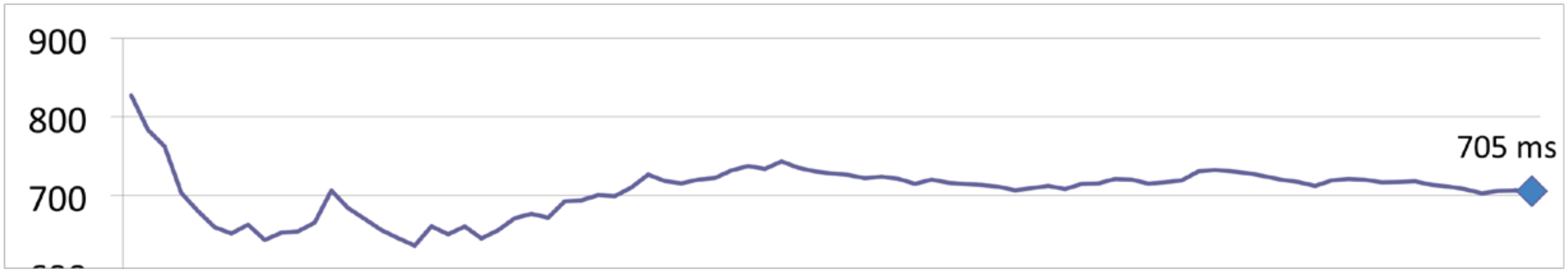
- Number of mappers depends on input size
  - More data, more machines needed for full parallelism
- Number of reducers can be chosen based on a trade-off
  - Too many: very small indices
  - Too few: little parallelization
- Single machine indexing would have taken days
- Real time can be significantly improved by reducing dictionary
- Vertical indexing is less efficient, though not much

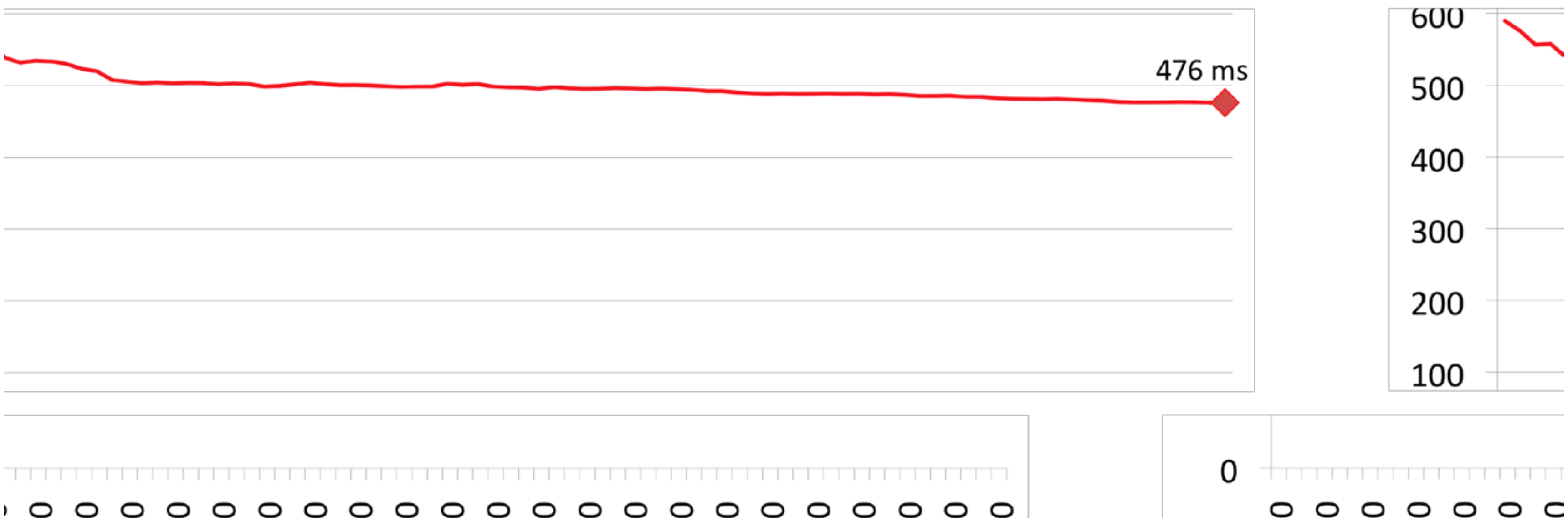
Metric	Horizontal	Vertical
Real time	3h 18m	4h 51m
Maps	2,000	2,000
Time per map	166s	581s
Map output records	$4.018 * 10^9$	$2.627 * 10^9$
Map output size	144 GB	68 GB
Reduces	20	20
Time per reduce	8371s	14987s
Reduce shuffle bytes	42 GB	28 GB

Keyword query performance (4450 unique queries)



Unigram field query performance (4450 unique queries)





Conclusions

- Horizontal index seems to be more efficient for both keyword queries and field restricts
  - verify under more ideal setting
  - verify under different memory constraints
  - check multi-word queries, joins on different properties
  - different datasets: microformats, RDFa vs. Linked Data
- Further efficiency improvements
  - Language detection
  - Local sort