Chapter IR:IV

IV. Indexes

- Inverted Indexes
- Query Processing
- Index Construction
- Index Compression

Index Construction

Inversion and Indexing

Turning a document d into a set of pairs of terms and postings is called inversion. Doing the same for a document collection D is called indexing.

An in-memory term frequency indexing algorithm:

```
    procedure INDEX(D):
 1: index ← hashmap()
 2: for all documents d ∈ D do
 3:     termsequence ← parse(d)
 4:     terms ← set()
 5:     frequencies ← hashmap()
 6:     for all t ∈ termsequence do
 7:         add(terms, t)
 8:         if t not exists in frequencies then add(frequencies, t, 1)
 9:         else add(frequencies, t, get(frequencies, t) + 1)
10:     for all t ∈ terms do
11:         if t not exists in index then add(index, t, list())
12:         postlist ← get(result, t)
13:         posting ← (identifier(d), get(frequencies, t))
14:         add(postlist, posting)
15: return index
```

Index Construction

Index Merging

If the document collection D does not fit into main memory, indexing is done iteratively, sharding the document collection and merging the shard indexes similar to an external merge sort:

1. The INDEX procedure runs until main memory is full.
2. The postlists are written to disk in alphabetical order of terms.
3. Steps 1 and 2 are repeated until D processed.
4. All k postlist files created are read concurrently, performing a k-way merge.

Index Construction

Index Merging

T	**Postings**
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

Index Construction

Index Merging

T	**Postings**
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	**Postings**
⋮	
t_i	

Index Construction

Index Merging

T	**Postings**
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$

Index Construction

Index Merging

T	**Postings**
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $4,9$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$

Index Construction

Index Merging

T	**Postings**
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$

Index Construction

Index Merging

T	Postings
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	Postings
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …

Index Construction

Index Merging

T	**Postings**
⋮	
t_i	$4,9$ $19,7$ $23,5$ $28,6$ $50,6$ …
t_j	$1,1$ $3,5$ $51,5$ $60,5$ $71,3$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $8,2$ $16,1$ $41,8$ $77,8$ …
t_j	$2,3$ $5,2$ $8,17$ $41,6$ $77,2$ …
⋮	

T	**Postings**
⋮	
t_i	$2,4$ $4,9$ $8,2$ $16,1$ $19,7$ $23,5$ $28,6$ $41,8$ $50,6$ $77,8$ …
t_j	$1,1$ $2,3$ $3,5$ $5,2$ $8,17$ $41,6$ $51,5$ $60,5$ $71,3$ $77,2$ …
⋮	

Remarks:

- Alphabetical ordering of intermediary postlist files ensures that the index can be read sequentially, albeit concurrently, during merging. Compare to document-at-a-time scoring.
- If a term appears in only one of the indexes, it will be added to the merged index.
- Postings with skip pointers can be pre-determined before merging a postlist so that appropriate space can be allocated immediately, but the actual skip pointers need to be recomputed after the postlist is merged.
- The number k of intermediary postlist files that can be read concurrently without causing too much seeking overhead depends on the underlying hardware (e.g., k is smaller for spinning hard disks than for solid state disks). In case k is too large, the intermediary postlist files are merged in multiple passes, $k' < k$ at a time, until all are merged.

Index Construction

Distributed Indexing

If neither the document collection D, nor its index can be stored on a single machine, indexing must be performed distributed across a computer cluster.

Many cluster computing frameworks exist nowadays; the NoSQL movement, and ultimately the Big Data hype, was kicked off by Google's MapReduce [Dean 2004].

From a developer perspective, data processing with MapReduce boils down to implementing two procedures:

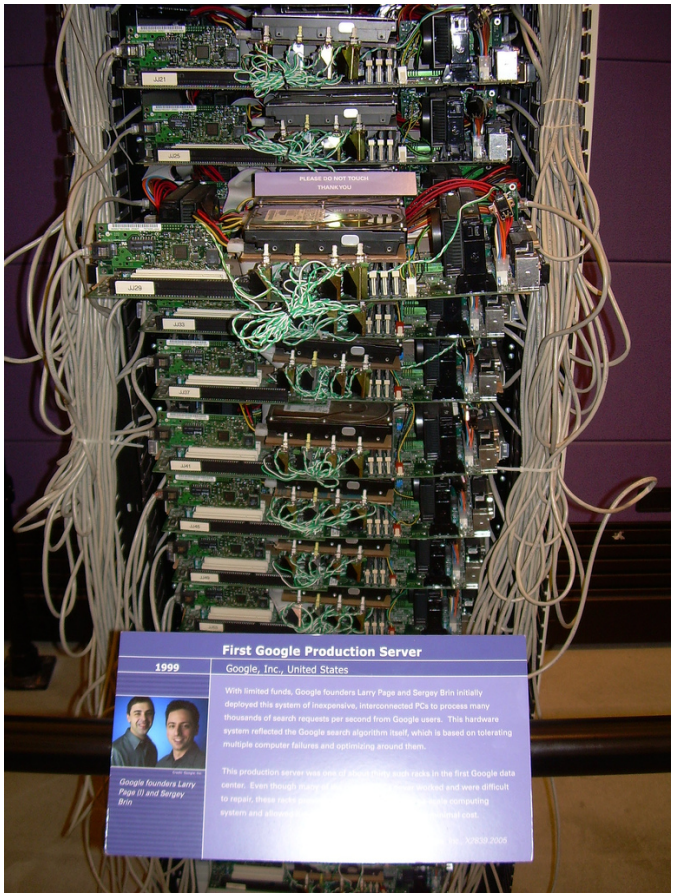
- Map: Given a key-value pair as input, it outputs a list of key-values pairs.
- Reduce: Given a key and the list of values output by map under that key, it outputs a key-value pair.

Example:

- InversionMapper: Given a pair (i, d_i), where i is the document identifier of document d_i as input, output a list of pairs (t, i) for all unique $t \in d_i$.
- DFReducer: Given $(t, [\ldots, i, \ldots])$ as input, output $(t, |[\ldots, i, \ldots]|)$.

Remarks:

- Computer clusters are often built from inexpensive commodity hardware. In the early days, desktop computers were used as Beowulf clusters, or dismantled and stacked. Google 1997 and 1999:



- The key contributions of the MapReduce framework are not the actual map and reduce functions, but the scalability and fault-tolerance achieved for a variety of applications by optimizing the execution engine [Wikipedia].
- This framework is best-suited for problems that are embarrassingly parallel.
- The most widespread open source implementation is found in Apache Hadoop.

Index Construction

Distributed Indexing

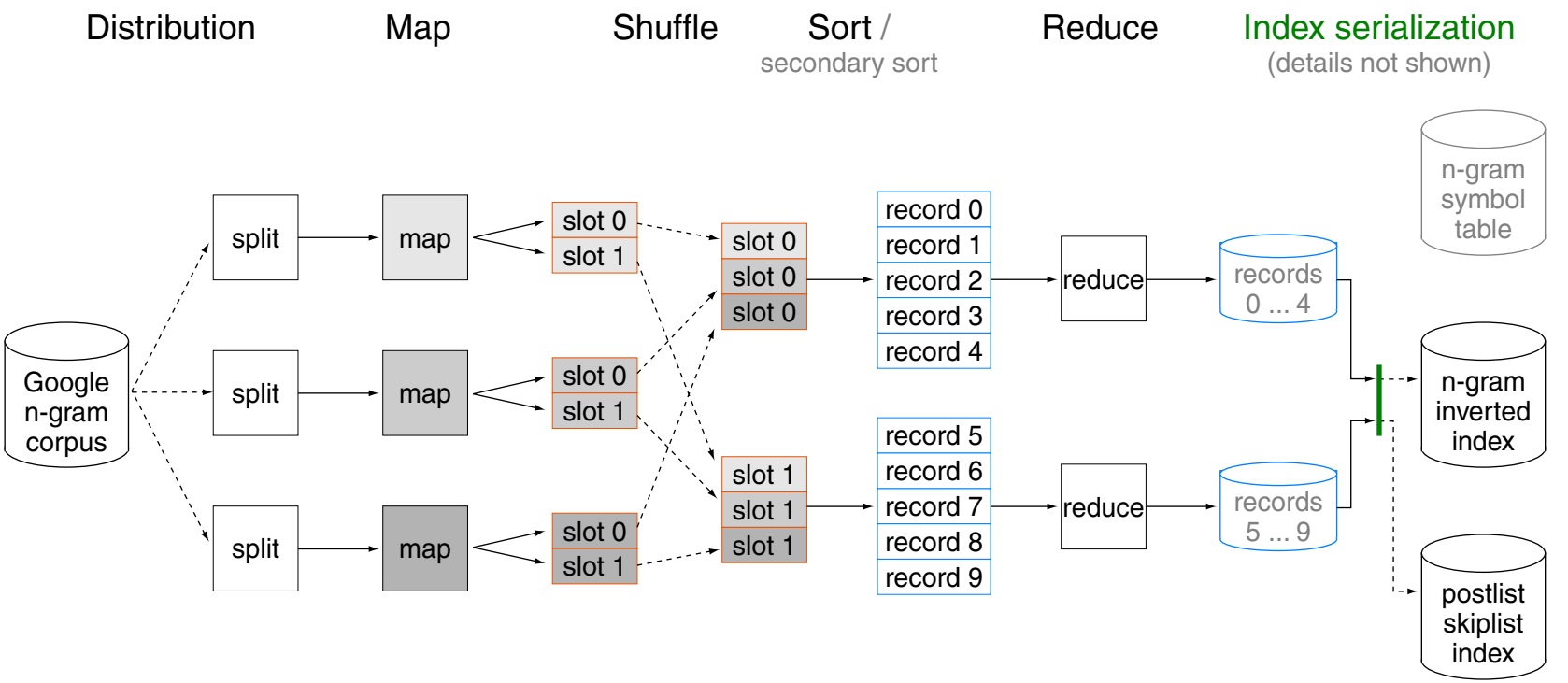
Presuming the document collection is stored in a distributed document storage across the cluster, the execution of a MapReduce job divides into three basic phases:

- Map phase
  The map function is called in parallel on all cluster nodes and fed chunks of the data. Its output is recorded locally on each cluster node.
- Shuffle phase
  The output is transferred to a random cluster node chosen using a hash function, so that the same key is always transferred to the same cluster node. Once all data belonging to a key are on the same node, the values are sorted.
- Reduce phase
  The reduce function is called in parallel on all cluster nodes and fed the sorted lists, recording their output.

The map and reduce functions are idempotent: they are reexecuted in case of failures. To make optimal use of available resources, the framework may execute the same task more than once on different machines, retaining the first output that emerges (so-called speculative execution).

Index Construction

Distributed Indexing: Example Netspeak [www.netspeak.org]



Input records

(line number, line)

(1, `hello world` 712963 4)
(2, `hello kitty` 1468134 5)

Intermediate records

((word, len, pos), (freq, id))

((`hello`, 2, 0), (712963, 4))
((`world`, 2, 1), (712963, 4))
((`hello`, 2, 0), (1468134, 5))
((`kitty`, 2, 1), (1468134, 5))

Sorted records

((word, len, pos), ((freq, id) ... (freq, id)))

((`hello`, 2, 0), ((1468134, 5),(712963, 4)))
((`world`, 2, 1), ((712963, 4)))
((`kitty`, 2, 1), ((1468134, 5)))

Output records

(reduced sorted records)

Cluster-wide data copy
Local data copy

Index Construction

Index Updates

Document collections grow and change. Therefore, the index must be updated. The following strategies are applied:

- Index merging
  When new documents arrive in large numbers at a time, they are indexed and then the existing index is merged with the new one.
- Result merging
  When new documents arrive in small numbers at a time, a separate, small index is maintained and updated. Queries are processed against both the existing index and the small one containing the new arrivals, fusing the results.
- Deletions list
  Deletions are recorded in a deletions list, and deleted documents are removed from search results before results are shown.

Modifications are done by inserting a new document, and deleting the previous version.