Indexing

Index Organization, Construction, Temporal Query Processing
Inverted Index

- Inverted Indexing basics revisited
- Indexing Static Collections
  - Dictionaries
  - Forward Index
- Inverted Index Organisation
- Scalable Indexing
- Indexing Temporal Collections
Why do we index text collections?

How do we index documents?
  
  What are the data structures?
  
  What are the design decisions for organising the index?

How do we index huge collections?

How do we index temporal collections?
Text Collections and Indexing

- Why do we index text collections?
  - Efficient document retrieval

- How do we index documents?

  - What are the data structures?
    - lexicon, inverted lists

  - What are the design decisions for organising the index?
    - document order, score order

- How do we index huge collections?
  - distributed indexing, term/doc partitioning

- How do we index temporal collections?
  - index maintenance strategies
Terminology Recap

terms, documents, collection

information retrieval

Lexicon

stemming, stop-word rem

queries, results

index, lexicon, posting, posting list
Lexicon or Dictionary

- Maintains **statistics** and **information** about the indexed unit (word, n-gram etc)

  < hannover ; location: 82271; tid:12 ; df:23, ...  >

- **Posting list location** - for posting list retrieval

- **Term identifier** - for term lookups, matching and range queries

- **Document frequency** and associated **statistics** - for ranking

- **Data Structures for Lexicon**
  
  - Hash-based Lexicon
  
  - B+-Tree based Lexicon
Hash-Based Lexicon

Figure 4.2

Dictionary data structure based on a hash table with $2^{10} = 1024$ entries (data extracted from the schema-independent index for TREC45). Terms with the same hash value are arranged in a linked list (chaining). Each term descriptor contains the term itself, the position of the term's postings list, and a pointer to the next entry in the linked list.

Storing each term in a fixed-size memory region of 20 bytes wastes 10.8 bytes per term on average (internal fragmentation). One way to eliminate the internal fragmentation is to not store the index terms themselves in the array, but only pointers to them. For example, the search engine could maintain a primary dictionary array, containing 32-bit pointers into a secondary array. The secondary array then contains the actual dictionary entries, consisting of the terms themselves and the corresponding pointers into the postings file. This way of organizing the search engine's dictionary data is shown in Figure 4.3. It is sometimes referred to as the dictionary-as-a-string approach, because there are no explicit delimiters between two consecutive dictionary entries; the secondary array can be thought of as a long, uninterrupted string.

For the GOV2 collection, the dictionary-as-a-string approach, compared to the dictionary layout shown in Figure 4.1, reduces the dictionary's storage requirement by 10$^{8} - 4 = 6$ bytes per entry. Here the term 4 stems from the pointer overhead in the primary array; the term 10 corresponds to the complete elimination of internal fragmentation.

It is worth pointing out that the term strings stored in the secondary array do not require an explicit termination symbol (e.g., the "\0" character), because the length of each term in the dictionary is implicitly given by the pointers in the primary array. For example, by looking at the pointers for "shakespeare" and "shakespearean" in Figure 4.3, we know that the dictionary entry for "shakespeare" requires $16629970 - 16629951 = 19$ bytes in total: 11 bytes for the term plus 8 bytes for the 64-bit file pointer into the postings file.
Hash-Based Lexicon

- Constant lookups based on a Hash table
- Entire Lexicon loaded to the memory
Hash-Based Lexicon

- Constant lookups based on a Hash table
- Entire Lexicon loaded to the memory
- Updates difficult
- Range Searches, Matching, Substring queries not supported
B+-Tree or Sort-based Lexicon

- **B+-Tree**: Leaf nodes additionally linked for efficient range search
- Supports lookups in $O(\log n)$ and range searches in $O(\log n + k)$
- Vocabulary dynamics (i.e., new or removed terms) no problem
- Works on *secondary storage*
• Mapping of doc-ids to term-ids in the same order

1: “what does the fox say?”

1: 124 53 1 49935 100

• Efficient retrieval of terms from (already parsed) text
• snippet generation
• proximity features for proximity-aware ranking
• per-doc term distribution for query expansions
Inverted index is a collection of posting lists

Posting contains **document identifiers** (as integers) along with **scores** (integers or doubles) and possibly **positions** (as integers)

Postings list can be organised according to

- **document identifiers** - document ordering
- **scores** - Impact ordering

What are the merits of these orderings?
Index Organisation

**Document Ordering**

- Based on faster intersections
- High compression of index using gap encoding of dids
- Easily updatable

**Score/Impact Ordering**

- Based on processing Top-k results fast
- Low compression ratio
- Difficult to update

Index organisation depends on query processing style.
Inverted Index Construction

- We are given a set of documents $D$, where each document $d$ is considered as a bag of terms

- Inverted Lists are created by a process termed as Inversion

- **Memory-based** Inversion
  - Takes place entirely in-memory
  - For small collections, where the index + lexicon fits in memory

- **Disk-based** Inversion
  - Sort-based inversion vs Merge-based inversion
Memory-based Inversion

- A **dictionary** is required that allows efficient single-term lookup and insertion operations.

- An extensible (i.e., dynamic) **list data structure** is needed that is used to store the postings for each.

**1:** “what does the fox say?”

**2:** “the fox jumped over the fence”

doc: [term, positions]

1: [the, <1,5>]

[term, positions]

….. [the, <3>] [fox, <4>] …..

….. [the, <1,5>] [fox, <2>] …..

[term, posting list]

“the”: [1, <3>] ….[2, <1,5>]
Sort-based Inversion

- Input Collection D $\gg$ memory size $M$

- Inversion can be seen as a sort operation on the term identifiers

- This method is based on **external sort** over data which does not fit into the memory
  - Read data of size $M$ into memory, sort them and write back to disk
  - Multiway merge of $D/M$ sorted lists to create index

- Shortcomings
  - Dictionary might not fit in-memory
  - Large memory requirements due to intermediate data
Exercise 1: Analysis of Sort-based Inversion

Simple Computational Model

Total number of postings = N
Number of postings which fit in memory = M
Cost of disk read/write of a posting = c

- What is the estimated cost of sort-based Inversion in terms of N, M and c?

- How does the cost compare with in-memory sort-based inversion (assuming we had enough memory or N > M)?
Generalisation of in-memory indexing

Reads input collection to create an in-memory index of size $M$ and write it to disk to create **partial indexes** with local lexicons

Compression in posting lists in partial indexes

Multiway Merge of corresponding lists from the partial indexes to create one consolidated index
Map-Reduce crash course

- Programming paradigm for distributed data processing
- Improves overall throughput by parallelising loading of data
- Data is partitioned into the nodes which process the data in the following phases
  - **Map**: Generates (key, value) pairs
  - **Shuffle**: Shuffles the pairs over the network to the reducers
  - **Reduce**: Operates on all values for the same key are
Map-Reduce Example: Word Count

1: “what does the fox say?”
   Mapper - 1
   - what: 1
   - does: 1
   - the: 1
   - fox: 1
   - say: 1

2: “the fox jumped over the fence”
   Mapper - 2
   - jumped: 1
   - over: 1
   - the: 2
   - fox: 1
   - fence: 1

Shuffle + Sort

Reducers aggregate freq.

Reducer - 1
- does: 1
- fence: 1
- jumped: 1
- fox: 1
- over: 1
- the: 1
- say: 1
- what: 1

Reducer - 2
- jumped: 1
- over: 1
- the: 2
- fox: 1
- fence: 1
Exercise 2: Index Construction using Map-Reduce

✨ How would you build the inverted index using Map-reduce?

✨ What are the key-value pairs as defined by the Mapper?

✨ What does the reducer do with the values of the same key?
Temporal Collections and Queries

- Temporal collections have temporal information
  - Publication times - News Articles
  - Valid times - Wikipedia articles, Web archive versions
  - Temporal references - time mentions in text
- Time-travel Queries: Retrieve all documents relevant to the text and time
  - Point in time queries: *house of cards @ 02/03/2013*
  - Time-interval queries: *game of thrones @ 2011 - 2014*
Given a versioned collection of documents with valid time intervals

and a time-travel text query: \textit{game of thrones @ 2011 - 2014}

We want to retrieve documents containing terms “game”, “thrones” and valid between 2011 - 2014

\textit{How do we efficiently retrieve these documents?}
Time-Travel Index

- Inverted index processes keyword queries
- Intersection of posting lists for processing queries
- Versions have valid time intervals
- Augment postings with valid time intervals
- Post filtering after standard query processing
Inverted index processes keyword queries

Intersection of posting lists for processing queries

Versions have valid time intervals

Augment postings with valid time intervals

Post filtering after standard query processing
Time-Travel Index

- Inverted index processes keyword queries
- Intersection of posting lists for processing queries
- Versions have valid time intervals
- Augment postings with valid time intervals
- Post filtering after standard query processing
• Inverted index processes keyword queries

• Intersection of posting lists for processing queries

• Versions have valid time intervals

• Augment postings with valid time intervals

• Post filtering after standard query processing
• If documents are points in time how would you organize the index?

• What is the problem if the documents are associated with time intervals?

• Query processing expensive due to wasted accesses
• Each interval represents a document in the posting list

• Data Model: Each document is associated with a time interval

• Query Model: Queries are associated with a time interval \([tb, te]\)
  
  • point in time queries: when begin time = end time
  
  • time interval queries
Challenges in Indexing Time

- We would want to avoid unwanted or wasted access to posting lists

- Typically only access those postings that are relevant or a few more (bounded loss)

- Dealing with time points easy, akin to range queries (sorting acc to begin time and range search)
Challenges in Indexing Time

- We would want to avoid unwanted or wasted access to posting lists.
- Typically only access those postings that are relevant or a few more (bounded loss).
- Dealing with time points easy, akin to range queries (sorting acc to begin time and range search).
Index List Partitioning

- **Vertically Partition** the temporal space and each partition

- Now multiple posting lists per term, each with a valid time interval

- Limits index access, introduces replication
- Dictionary or Lexicon should contain partitioning information.
- For each temporal query, select a subset of affected partitions and only read them.
- Filter postings which do not overlap with query time interval.

<table>
<thead>
<tr>
<th>term</th>
<th>partition</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>hannover</td>
<td>[t1 - t5)</td>
<td>12646</td>
</tr>
<tr>
<td>hannover</td>
<td>[t5 - t7)</td>
<td>12673</td>
</tr>
<tr>
<td>hannover</td>
<td>[t7 - t25)</td>
<td>13446</td>
</tr>
<tr>
<td>hannover</td>
<td>[t25 - t43)</td>
<td>15324</td>
</tr>
</tbody>
</table>

“hannover”
Optimal Approaches

• **Performance Optimal Approach**

  - Keeps one posting list per every elementary time interval,
  - Achieves optimal performance but large space overhead

• **Space Optimal Approach**

  - No replication of postings, no blowup, sub-optimal performance
Partitioning Strategies

- Given an input sequence of intervals how do we partition them into sublists?

- Space Bound Materialization Approach: We have a limited budget for space, need to maximize our performance

- Performance Guarantee Approach: For any query we need a guarantee on the performance loss, need to minimize blowup

Trade-off size and performance
Partitioning Strategies

- Given an input sequence of intervals how do we partition them into sublists?

- Space Bound Materialization Approach: We have a limited budget for space, need to maximize our performance

- Performance Guarantee Approach: For any query we need a guarantee on the performance loss, need to minimize blowup

Trade-off size and performance
Space Bound Approach

• Minimize expected number of postings read for a time-point query, while ensuring that the index contains at most \( \kappa \) times the optimal number of postings

• Optimal solution computable in \( O(|S| \times n^2) \) time and \( O(|S| \times n) \) space using dynamic programming over prefix subproblems \([t_1, t_k)\) and space bounds \( s \leq \kappa \cdot |Lv| \)

Space budget = \( \frac{1}{3} \) \cdot (optimal number of postings)
Space Bound Approach

- Minimize expected number of postings read for a time-point query, while ensuring that the index contains at most $\kappa$ times the optimal number of postings.

- Optimal solution computable in $O(|S| \times n^2)$ time and $O(|S| \times n)$ space using dynamic programming over prefix subproblems $[t_1, t_k)$ and space bounds $s \leq \kappa \cdot |L(v)|$.

Space budget = $1/3 \cdot$ (optimal number of postings)
Performance Guarantee

- Minimize total number of postings kept in the index, while guaranteeing that for any time-point query the number of postings read is at most a factor $\gamma$ worse than optimal

- Optimal solution computable in time $O(|L_v| + n^2)$ and space $O(n^2)$ using dynamic programming over prefix subproblems $[t_1, t_k)$

$\text{performance guarantee} = \gamma \times \text{(number of optimal results at that time)}$
Performance Guarantee

- Minimize total number of postings kept in the index, while guaranteeing that for any time-point query the number of postings read is at most a factor $\gamma$ worse than optimal.

- Optimal solution computable in time $O(|Lv| + n^2)$ and space $O(n^2)$ using dynamic programming over prefix subproblems $[t_l, t_k)$.

performance guarantee = $\gamma$ (times the number of optimal results at that time)
Exercise

- Performance Guarantee with $\gamma = 2$ (read at max twice the number of postings than optimal)

- Space bound approach with $\kappa = 1.33$ (1/3 more than overall space)
Can we partition a posting list without replicating postings?

- Index size blowup due to replication of postings across slices
- Query processing inefficient if replicated postings are accessed multiple times
Can we partition a posting list without replicating postings?

- Index size blowup due to replication of postings across slices
- Query processing inefficient if replicated postings are accessed multiple times
Can we partition a posting list without replicating postings?

- Index size blowup due to replication of postings across slices.
- Query processing inefficient if replicated postings are accessed multiple times.
Index Sharding

- Partition documents in each posting list into sublists called shards
- Contents of each shard disjoint - no replication, no index blowup
- Postings stored in begin time order
- Access structure over each shard for efficient query processing
Index Sharding

- All shards for a given query term are accessed
- **Open-skip-scan** on each shard assisted by impact lists
- Result list constructed by merging results from each shard
Index Sharding - Impact Lists

- **Open** - Each shard of a query term opened for access
- **Skip** - Given a query begin time seek to appropriate offset
- **Scan** - Read while postings still have overlap with query time interval

**Impact list**

<table>
<thead>
<tr>
<th>( t_1 )</th>
<th>( t_{12} )</th>
<th>( t_{52} )</th>
<th>( t_{100} )</th>
<th>( t_{115} )</th>
<th>( t_{150} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>( \leq t_e )</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

1 \( \leq t_e \)

read until begin time \( < t_e \)
Index Sharding - Impact Lists

- **Open** - Each shard of a query term opened for access
- **Skip** - Given a query begin time seek to appropriate offset
- **Scan** - Read while postings still have overlap with query time interval

**Impact list**

<table>
<thead>
<tr>
<th>Time</th>
<th>Shard</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>1</td>
</tr>
<tr>
<td>t₁₂</td>
<td>2</td>
</tr>
<tr>
<td>t₅₂</td>
<td>5</td>
</tr>
<tr>
<td>t₁₀₀</td>
<td>4</td>
</tr>
<tr>
<td>t₁₁₅</td>
<td>5</td>
</tr>
<tr>
<td>t₁₅₀</td>
<td>5</td>
</tr>
</tbody>
</table>

Seek to shard offset for tᵦ

Read until begin time < tₑ
Index Sharding - Impact Lists

- **Open** - Each shard of a query term opened for access
- **Skip** - Given a query begin time seek to appropriate offset
- **Scan** - Read while postings still have overlap with query time interval

**Impact list**

- \( t_1 \)
- \( t_{12} \)
- \( t_{52} \)
- \( t_{100} \)
- \( t_{115} \)
- \( t_{150} \)
Index Sharding - Impact Lists

- **Open** - Each shard of a query term opened for access
- **Skip** - Given a query begin time seek to appropriate offset
- **Scan** - Read while postings still have overlap with query time interval

**Impact list**

```
<table>
<thead>
<tr>
<th></th>
<th>t_1</th>
<th>t_12</th>
<th>t_52</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

seek to shard offset for t_b

```

```
<table>
<thead>
<tr>
<th></th>
<th>t_100</th>
<th>t_115</th>
<th>t_150</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

read until begin time < t_e

```

```
<table>
<thead>
<tr>
<th></th>
<th>l</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

Avishek Anand
Index Sharding - Impact Lists

- **Open** - Each shard of a query term opened for access
- **Skip** - Given a query begin time seek to appropriate offset
- **Scan** - Read while postings still have overlap with query time interval

Impact list

- \( t_1 \) - \( \text{Read until begin time} < t_e \)
- \( t_{12} \)
- \( t_{52} \)
- \( t_{100} \)
- \( t_{115} \)
- \( t_{150} \)
Index Sharding - Impact Lists

- **Open** - Each shard of a query term opened for access
- **Skip** - Given a query begin time seek to appropriate offset
- **Scan** - Read while postings still have overlap with query time interval

Impact list
Index Sharding - Staircase Property

- **Wasted reads** are processed but do not overlap with the query time interval

- **Staircase property** in a shard
  - Intervals arranged in begin time order
  - No interval completely subsumes another interval

- Eliminates **wasted reads**
Index Sharding - Staircase Property

- **Wasted reads** are processed but do not overlap with the query time interval
- **Staircase property** in a shard
  - Intervals arranged in begin time order
  - No interval completely subsumes another interval
- Eliminates **wasted reads**
Index Sharding - Staircase Property

- **Wasted reads** are processed but do not overlap with the query time interval
- **Staircase property** in a shard
  - Intervals arranged in begin time order
  - No interval completely subsumes another interval
- Eliminates **wasted reads**
Index Sharding - Staircase Property

- **Wasted reads** are processed but do not overlap with the query time interval
- **Staircase property** in a shard
  - Intervals arranged in begin time order
  - No interval completely subsumes another interval
- Eliminates **wasted reads**
Index Sharding - Idealized Sharding

- Staircase property eliminates sequential accesses of postings non-overlapping with query time interval
- Minimizing number of shards is essential in minimizing number of random accesses
- **Input**: Set of postings/intervals corresponding to a postings list
- **Problem Statement**: Minimize the number of shards where each shard exhibits the **staircase property**

Greedy Algorithm exists which is proven to be optimal
Index Sharding - Idealized Sharding

**Input:**

- Postings arrive in begin time order
- For each posting choose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$
Index Sharding - Idealized Sharding

**Input:**

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$
Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$

Shard 1
Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$
Index Sharding - Idealized Sharding

Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Runtime complexity $O(n \log n)$

Shard 1

- $p_1$
- $p_2$
- $p_3$
- $p_4$
- $p_5$

Shard 2

- $p_2$

$p_1$ $p_2$ $p_3$ $p_4$ $p_5$
Index Sharding - Idealized Sharding

Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Append posting to the end of chosen shard

Shard 1

- Runtime complexity $O(n \log n)$

Shard 2
Index Sharding - Idealized Sharding

Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$
Index Sharding - Idealized Sharding

**Input:**

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$
Index Sharding - Idealized Sharding

Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Runtime complexity $O(n \log n)$

Shard 1

Shard 2
Index Sharding - Idealized Sharding

Input:

- Postings arrive in begin time order
- For each posting chose a shard which
  - does not violate staircase prop.
  - has min. end time difference
- Append posting to the end of chosen shard
- Runtime complexity $O(n \log n)$
Exercise

- What is the Idealized sharding for the given input?

- What are Impact lists for each idealized shard?

- What is the worst case (in the number of shards) example for idealised sharding?
Index Sharding - Challenges

• Idealized Sharding

  __________
  __________
  __________

• No wasted reads
• Many shards
• QP suffers. Why?

• Random accesses (RA) are typically much more expensive than sequential accesses (SA)
Index Sharding - Challenges

Idealized Sharding

- No wasted reads
- Many shards
- QP suffers. Why?

More shards the more the random accesses to disk

- Random accesses (RA) are typically much more expensive than sequential accesses (SA)
- Allow wasted reads to balance SA and RA
Index Sharding - Challenges

- Random accesses (RA) are typically much more expensive than sequential accesses (SA)
Index Sharding - Challenges

- Random accesses (RA) are typically much more expensive than sequential accesses (SA)

- Allow wasted reads to balance SA and RA
Bounded Subsumption

• Balancing sequential and random accesses

• **Bounded subsumption:** no more than $\eta$ wasted reads for any query begin time

• **Bounded Subsumption Problem:** Minimize number of shards s.t. each shard has bounded subsumption
Bounded Subsumption

- Balancing sequential and random accesses

- **Bounded subsumption**: no more than $\eta$ wasted reads for any query begin time

- **Bounded Subsumption Problem**: Minimize number of shards s.t. each shard has bounded subsumption
Bounded Subsumption

- Balancing sequential and random accesses
- **Bounded subsumption**: no more than $\eta$ wasted reads for any query begin time

Can we create shards solving the bounded subsumption problem?
Incremental Sharding

- Algorithm assigns incoming posting to a shard
- Posting inserted into shard buffer maintaining begin time order
- Top posting popped and appended to the shard end
Incremental Sharding

- Algorithm assigns incoming posting to a shard
- Posting inserted into shard buffer maintaining begin time order
- Top posting popped and appended to the shard end
Incremental Sharding

- Algorithm assigns incoming posting to a shard
- Posting inserted into shard buffer maintaining begin time order
- Top posting popped and appended to the shard end
Incremental Sharding

- Algorithm assigns incoming posting to a shard
- Posting inserted into shard buffer maintaining begin time order
- Top posting popped and appended to the shard end
Incremental Sharding

- Algorithm assigns incoming posting to a shard
- Posting inserted into shard buffer maintaining begin time order
- Top posting popped and appended to the shard end
Putting Things Together

- Start with initial posting list and partition them vertically or horizontally
- Given a query time interval for each term
  - Access either subset of entire lists or part of all lists
- Intersect or Union the results for all query terms
Open Source Full-text Indexing Software

- Lemur
- elasticsearch
- MG4J
- Terrier
- Lucene
- Apache Solr
- Zettair
References

✦ Information retrieval:  (http://www.ir.uwaterloo.ca/book/)
  Stefan Büttcher, Google Inc. , Charles L. A. Clarke, Univ. of Waterloo,
  Gordon V. Cormack, Univ. of Waterloo

✦ Managing Gigabytes: by Justin Zobel, Alistair Moffat, Ian Witten

✦ Indexing Methods for Web Archives: Avishek Anand