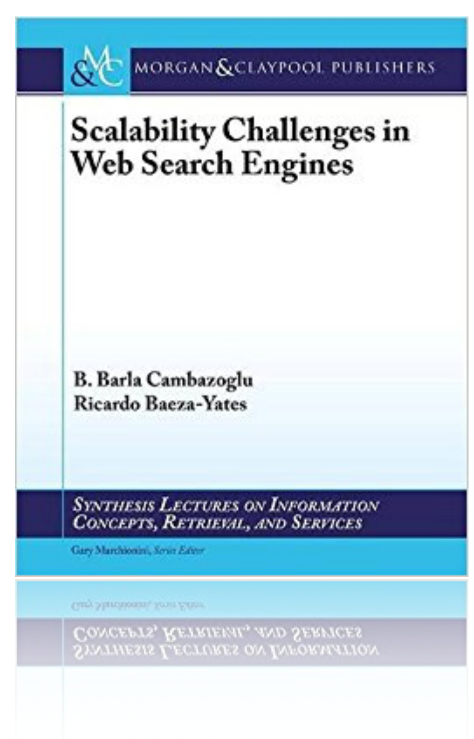
Disclaimer

- These slides have been prepared by B. Barla Cambazoglu, Director of Applied Science, NTENT Inc.
- Used in several IR tutorials and schools by Ricardo Baeza-Yates and B. Barla Cambazoglu.
- Used with permission, please do not redistribute in any form.
- Please refer to:



Indexing

The indexing system performs several tasks

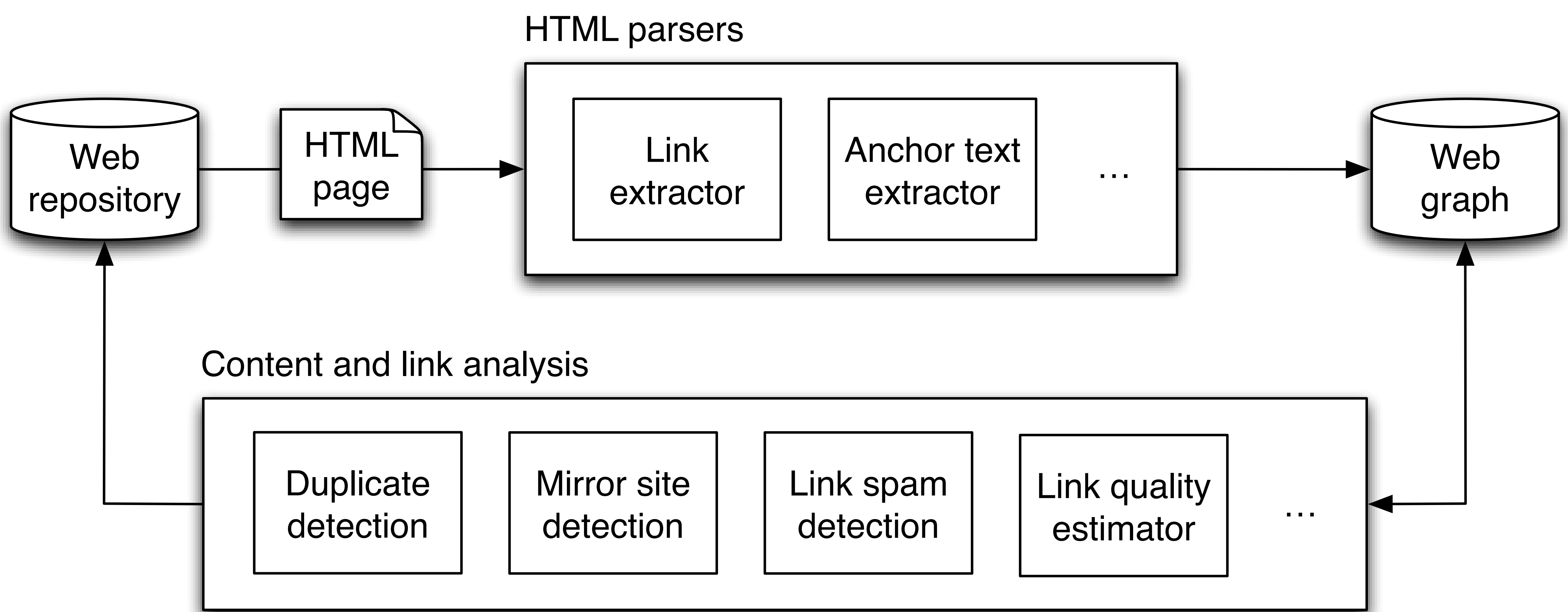
- performs information extraction, filtering, and classification on downloaded web pages
- provides meta-data, metrics, and other kinds of feedback to the crawling and query processing systems
- converts the pages in the web repository into appropriate index structures that facilitate searching the textual content of pages.

Document Processing Pipeline

- A typical indexing system involves various document processing pipelines, each performing different normalization or extraction tasks on web pages.

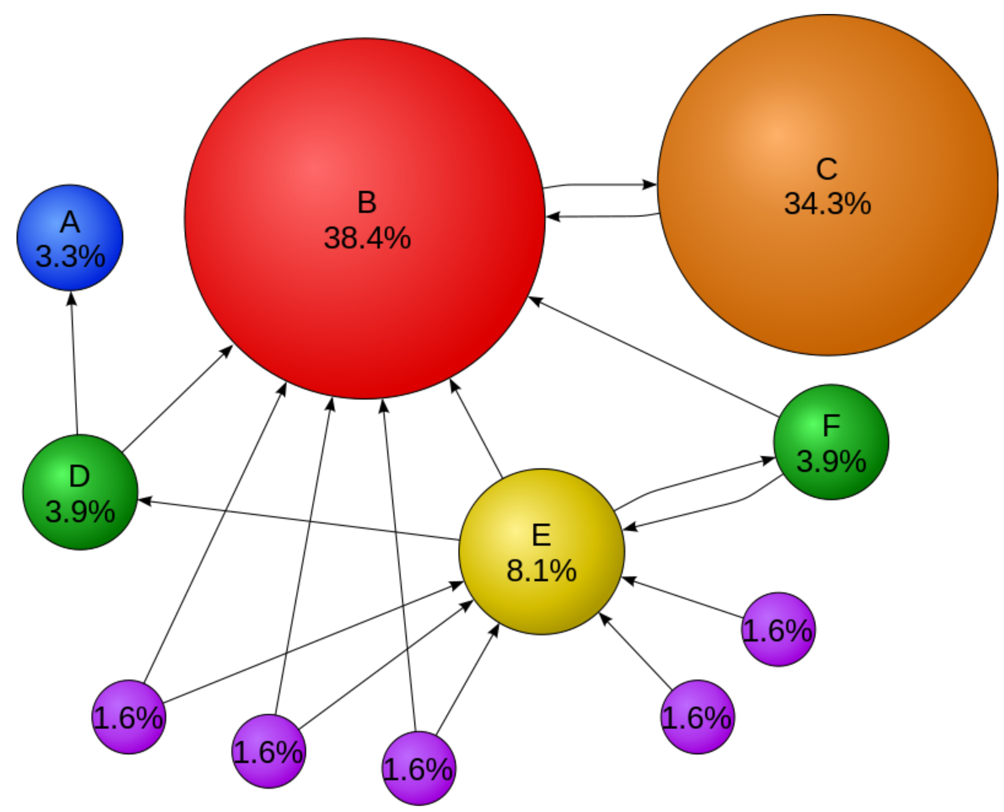
- Common data structures generated by these pipelines are
  - web graph
  - page attribute file
  - inverted index

Web Graph


HTML parsers
Web repository
HTML page
Link extractor
Anchor text extractor
…
Web graph
Content and link analysis
Duplicate detection
Mirror site detection
Link spam detection
Link quality estimator
…

Web Graph

- Web graph
  - node: attributes about the page
    - URL
    - inbound/outbound links
    - geographical region
    - language
  - edges: attributes about the links
    - anchor text



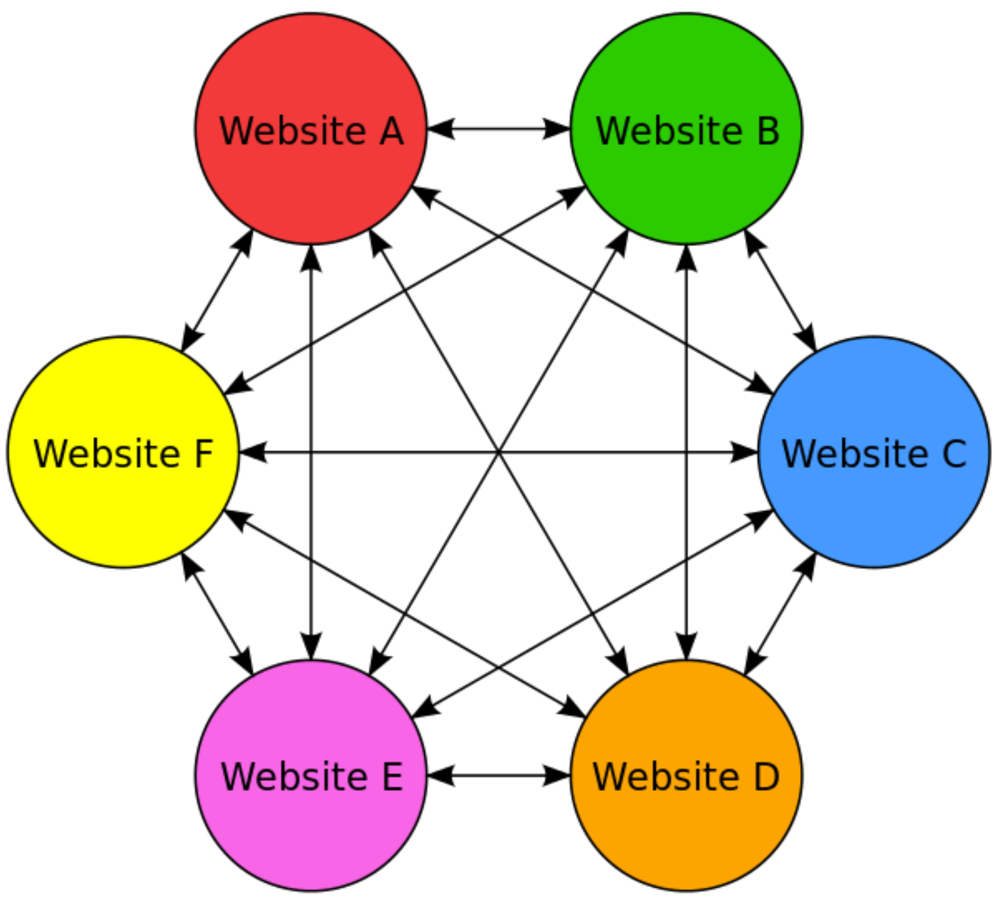
- Built at different granularities
  - page-level: duplicate detection
  - host-level: host quality estimation
  - site-level: mirror site detection

Link Analysis

- PageRank: A link analysis algorithm that assigns a weight to each web page indicating its importance.
- Iterative process that converges to a unique solution.
- Weight of a page is proportional to
  - number of inbound links of the page
  - weight of linking pages
- Other algorithms
  - HITS
  - TrustRank

Spam Detection

- Types of spam: link spam, content spam, cloaking/redirection spam, click spam.



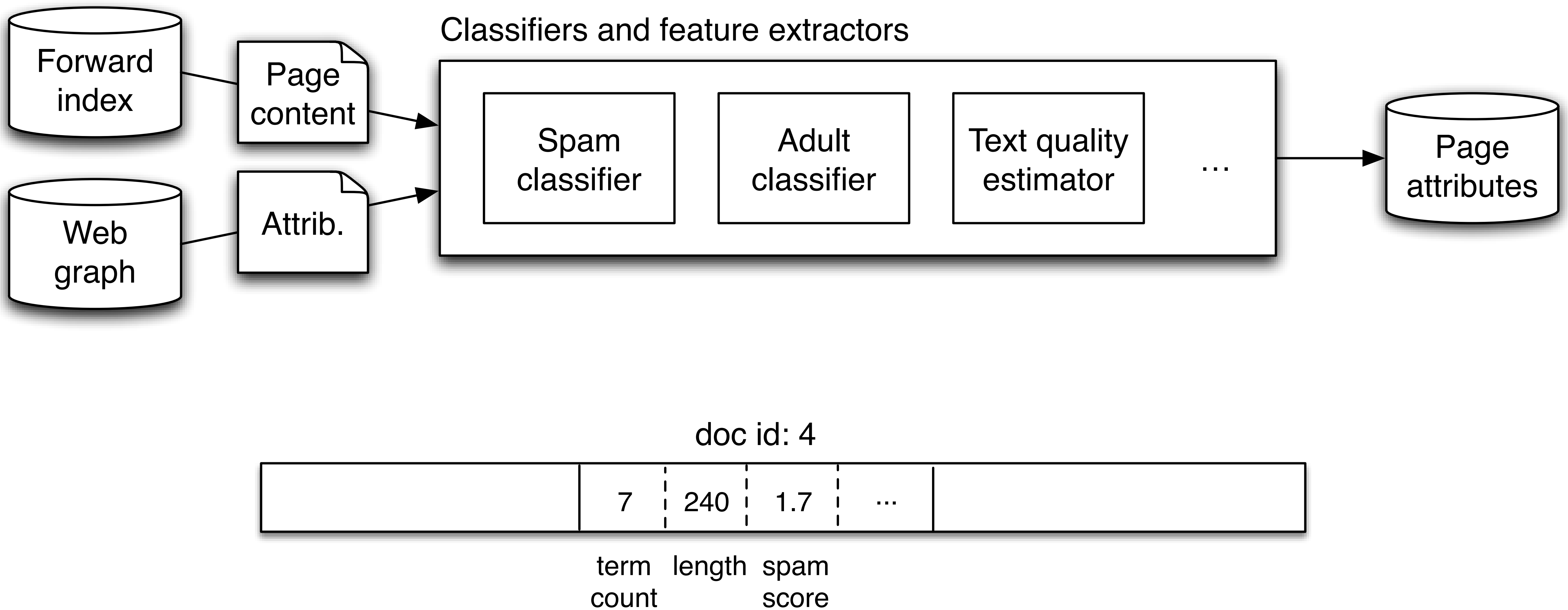
Duplicate Page Detection

- Detecting pages that have duplicate content
  - exact duplicates
    - comparing hash values
  - near duplicates
    - shingles
    - locality sensitive hashing

P1: A B C D E F → 79, 189, 44, 14, 99 → H1 = {14, 44, 79} → J(H1,H2) = 4/6

P2: A B C X D E F → 79, 189, 84, 68, 14, 99 → H2 = {14, 68, 79}

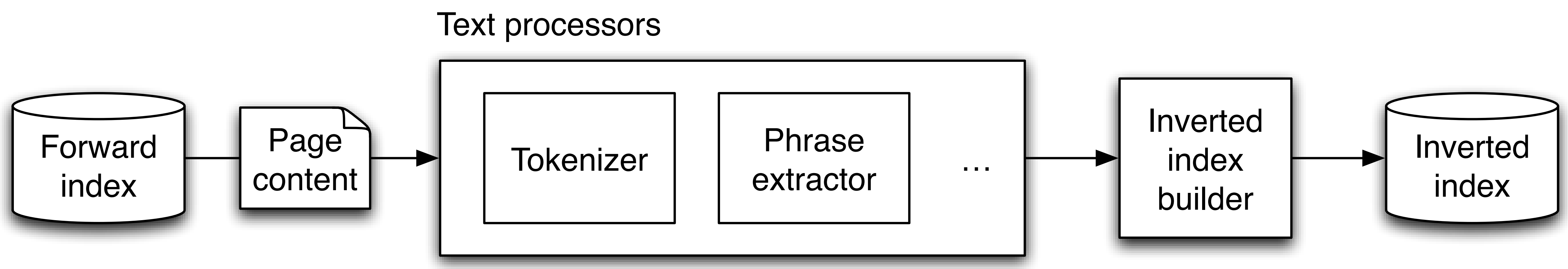
Page Attribute File



Query-Independent Features

Feature	Source	Description
Content spam	Page content	Score indicating the likelihood that the page content is spam
Text quality	Page content	Score combining various text quality features (e.g., readability)
Link quality	Web graph	Page importance estimated based on page's link structure
CTR	Query logs	Observed click-through rate of the page in search results (if available)
Dwell time	Query logs	Average time spent by the users on the page
Page load time	Web server	Average time it takes to receive the page from the server
URL depth	URL string	Number of slashes in the absolute path of the URL

Inverted Index



- Text processing may involve
  - tokenization
  - stopword removal
  - case conversion
  - stemming
- Example
  - original text: *Living in America*
  - applying all: *liv america*
  - in practice: *living in america*

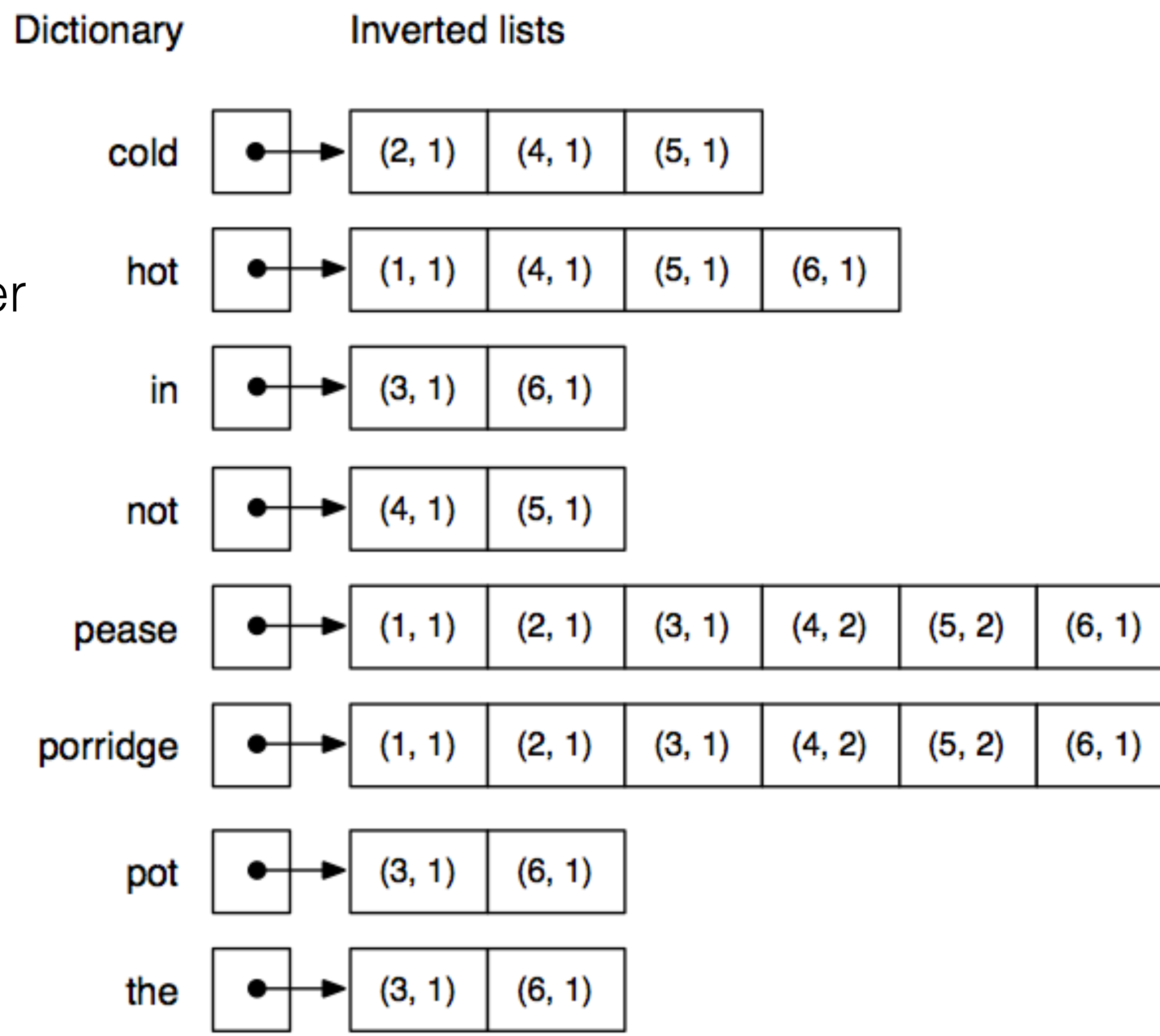
Sample Document Collection

Doc id	Text content
1	pease porridge hot
2	pease porridge cold
3	pease porridge in the pot
4	pease porridge hot, pease porridge not cold
5	pease porridge cold, pease porridge not hot
6	pease porridge hot in the pot

Inverted Index

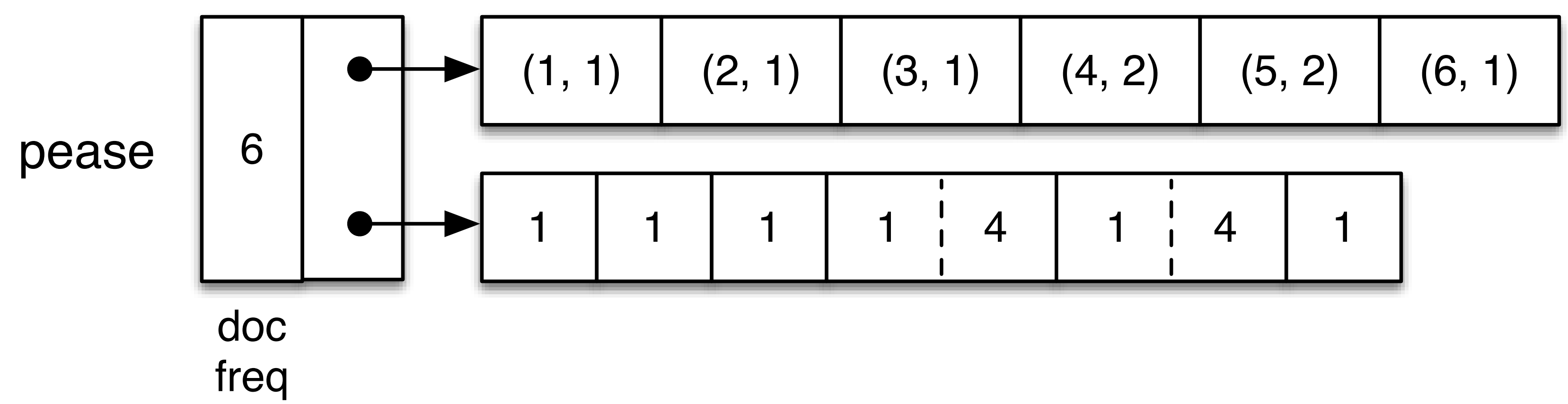
- An inverted index is a representation for the document collection over which user queries are evaluated.

- It has two parts
  - a vocabulary index (dictionary)
  - inverted lists
    - document id
    - term information

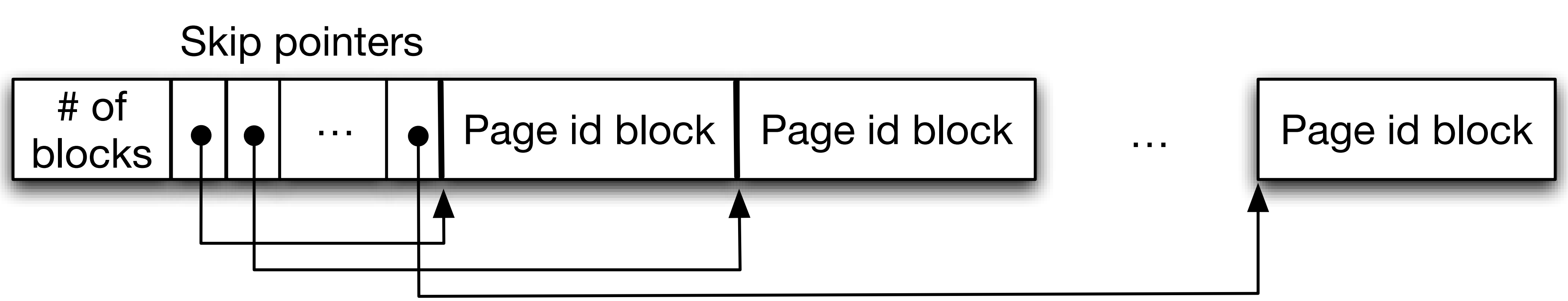


Inverted Index

- Extensions
  - position lists: list of all positions a term occurs in a document



  - skipping



  - title, body, header, anchor text (inbound, outbound links)

Success Measure

- Quality measures
  - spam rate: fraction of spam pages in the index
  - duplicate rate: fraction of near duplicate web pages in the index
- Performance measures
  - compactness: size of the index in bytes
  - deployment cost: effort needed to create and deploy a new inverted index from scratch
  - update cost: time and space overhead of updating a document entry in the index

Compression

- Benefits
  - reduced space consumption
  - reduced transfer costs
  - increased posting list cache hit rate

- Gap encoding
  - original: $17\quad 18\quad 28\quad 40\quad 44\quad 47\quad 56\quad 58$
  - gap encoded: $17\quad 1\quad \underbrace{10\quad 12\quad 4\quad 3\quad 9\quad 2}_{\text{gaps}}$

Compression Algorithms

Compression algorithm	Input sequence	Output	Parameters	Encoded values
Unary	gaps	bit-aligned	non-parametric	individual values
Gamma	gaps	bit-aligned	non-parametric	individual values
Delta	gaps	bit-aligned	non-parametric	individual values
Variable byte	gaps	byte-aligned	non-parametric	individual values
Golomb	gaps	bit-aligned	parametric	individual values
Simple-9	gaps	word-aligned	parametric	blocks of values
PForDelta	gaps	bit-aligned	parametric	blocks of values
Binary interpolation	monotonic sequences	bit-aligned	parametric	bisections
Elias-Fano	monotonic sequences	bit-aligned	parametric	entire sequence

Docid Reordering

- Goal: reassign document identifiers so that we obtain many small d-gaps, facilitating compression.

Id mapping:

1 → 1
2 → 9
3 → 2
4 → 7
5 → 8
6 → 3
7 → 5
8 → 6
9 → 4

Original lists:

L1: 1, 3, 6, 8, 9 L2: 2, 4, 5, 6, 9 L3: 3, 6, 7, 9

Original d-gaps:

L1: 2, 3, 2, 1 L2: 2, 1, 1, 3 L3: 3, 1, 2

Reordered lists:

L1: 1, 2, 3, 4, 6 L2: 3, 4, 7, 8, 9 L3: 2, 3, 4, 5

New d-gaps:

L1: 1, 1, 1, 2 L2: 1, 3, 1, 1 L3: 1, 1, 1

Docid Reordering

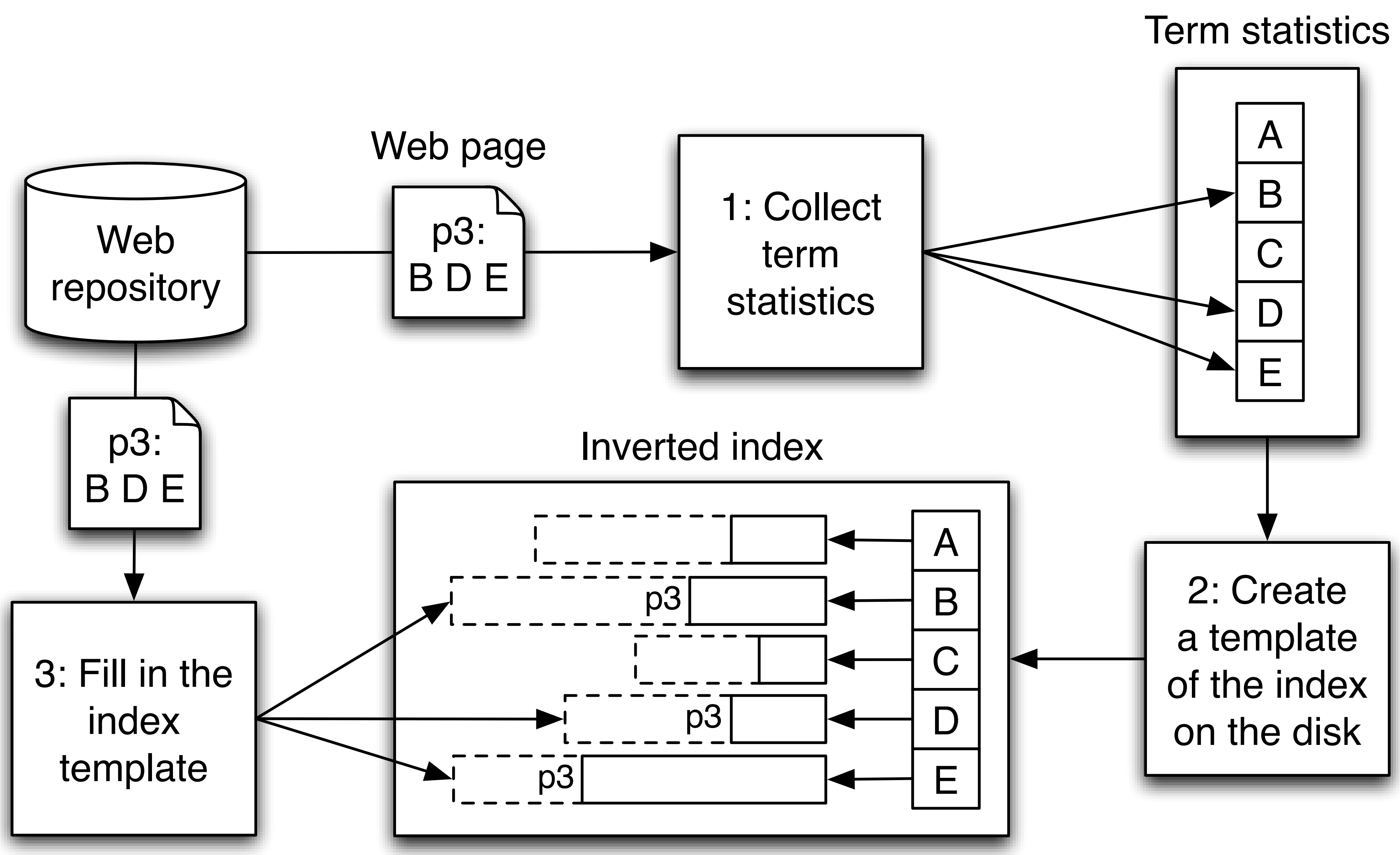
- Techniques
  - traversal of document similarity graph
    - formulated as the traveling salesman problem
  - clustering similar documents
    - assigns nearby ids to documents in the same cluster
  - sorting URLs alphabetically and assigning ids in that order
    - idea: pages from the same site have high textual overlap
    - simple yet effective
    - only applicable to web page collections

Index Construction

- Equivalent to computing the transpose of a matrix.
- In-memory techniques do not work well with web-scale data.
- Techniques
  - two-phase
  - one-phase

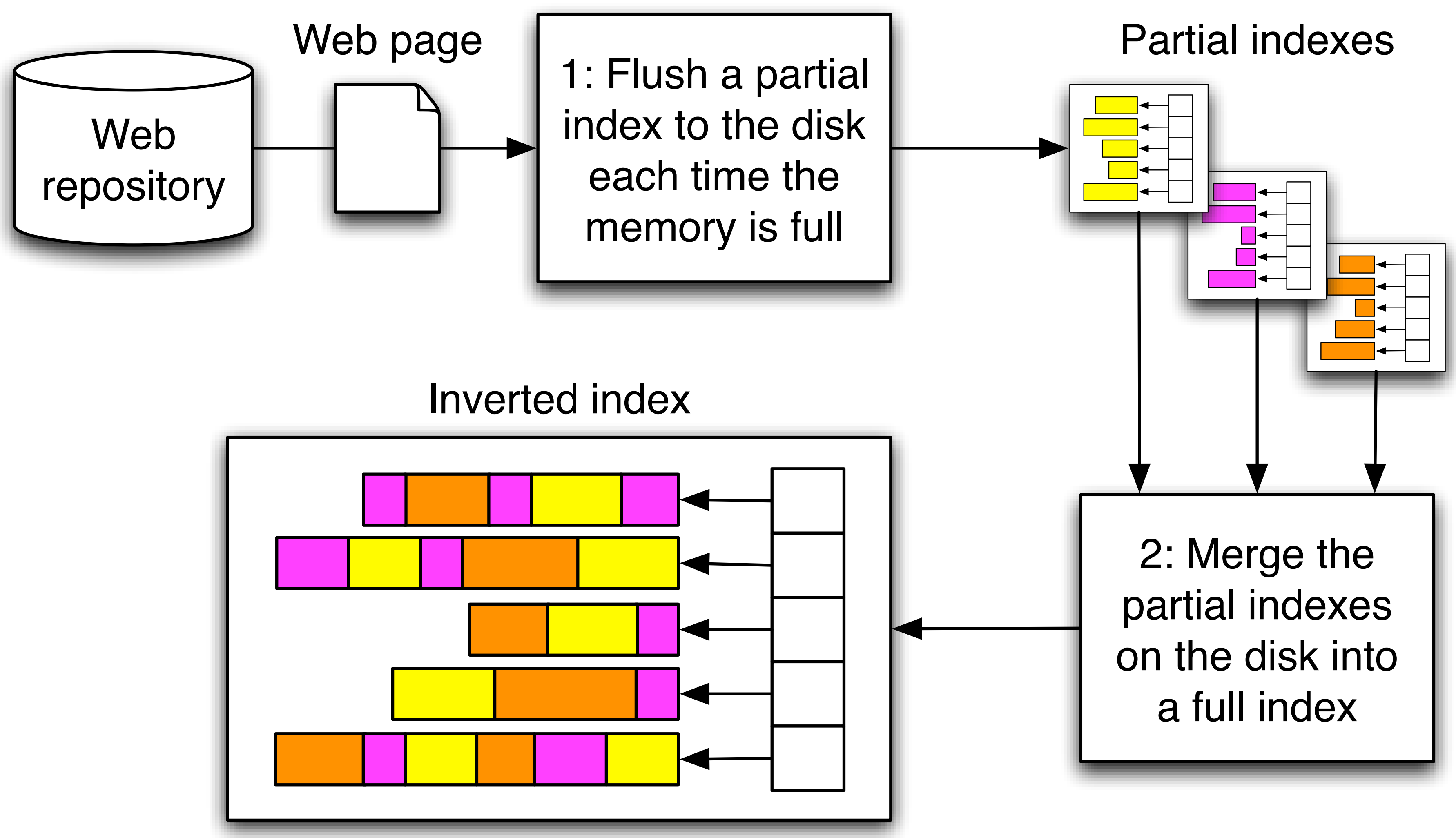
Two Phase

- First phase: read the collection and allocate a skeleton for the index.
- Second phase: fill the posting lists.



One Phase

- Keep reading documents and building an in-memory index.
- Each time the memory is full, flush the index to the disk.
- Merge all on-disk indexes into a single index in a final step.



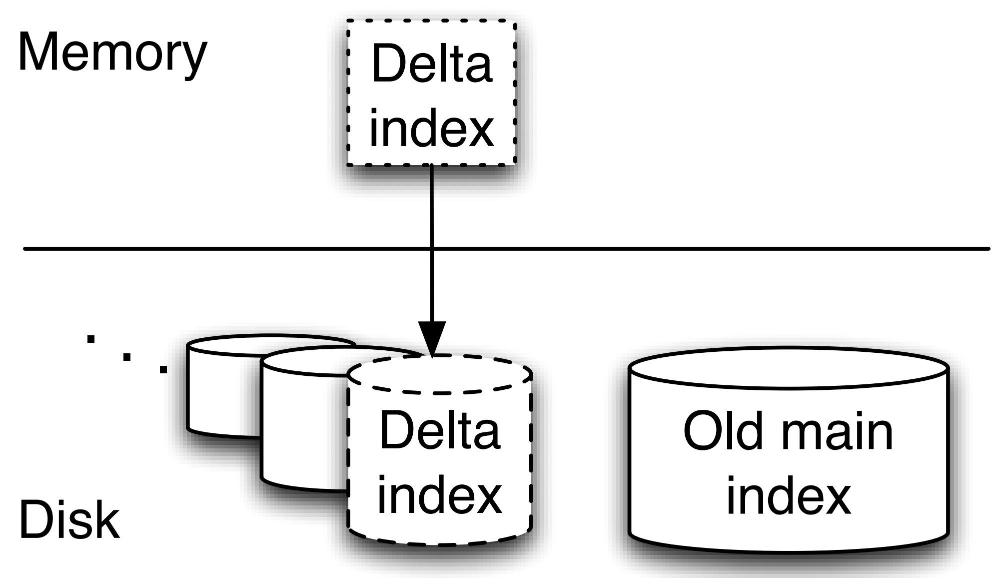
Index Maintenance

- Grow a new (delta) index in the memory; each time the memory is full, flush the in-memory index to disk.

- Techniques
  - no merge
  - incremental update
  - immediate merge
  - lazy merge

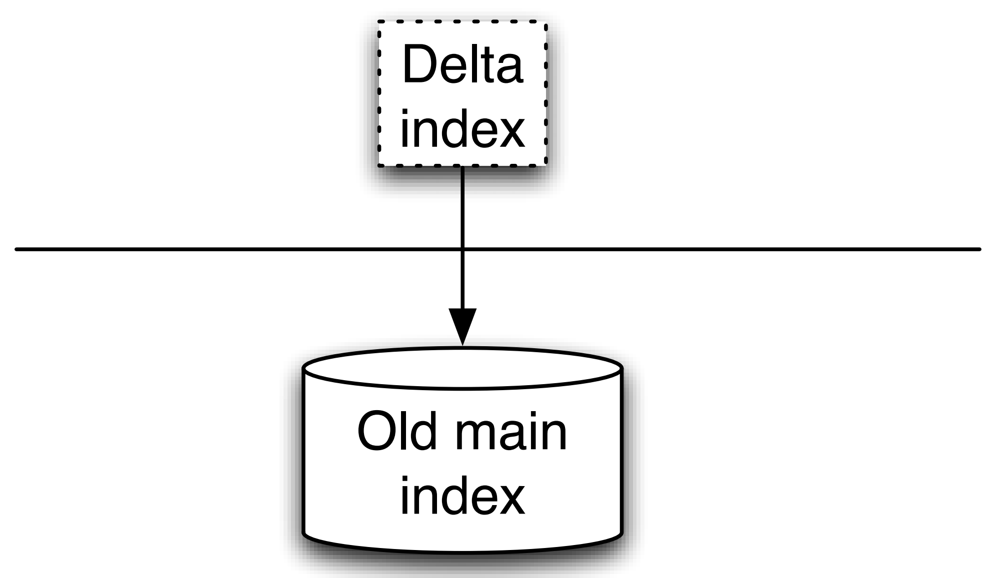
No Merge

- Flushed index is written to disk as a separate index.
- Increases fragmentation and query processing time.
- Eventually requires merging all on-disk indexes or rebuilding.



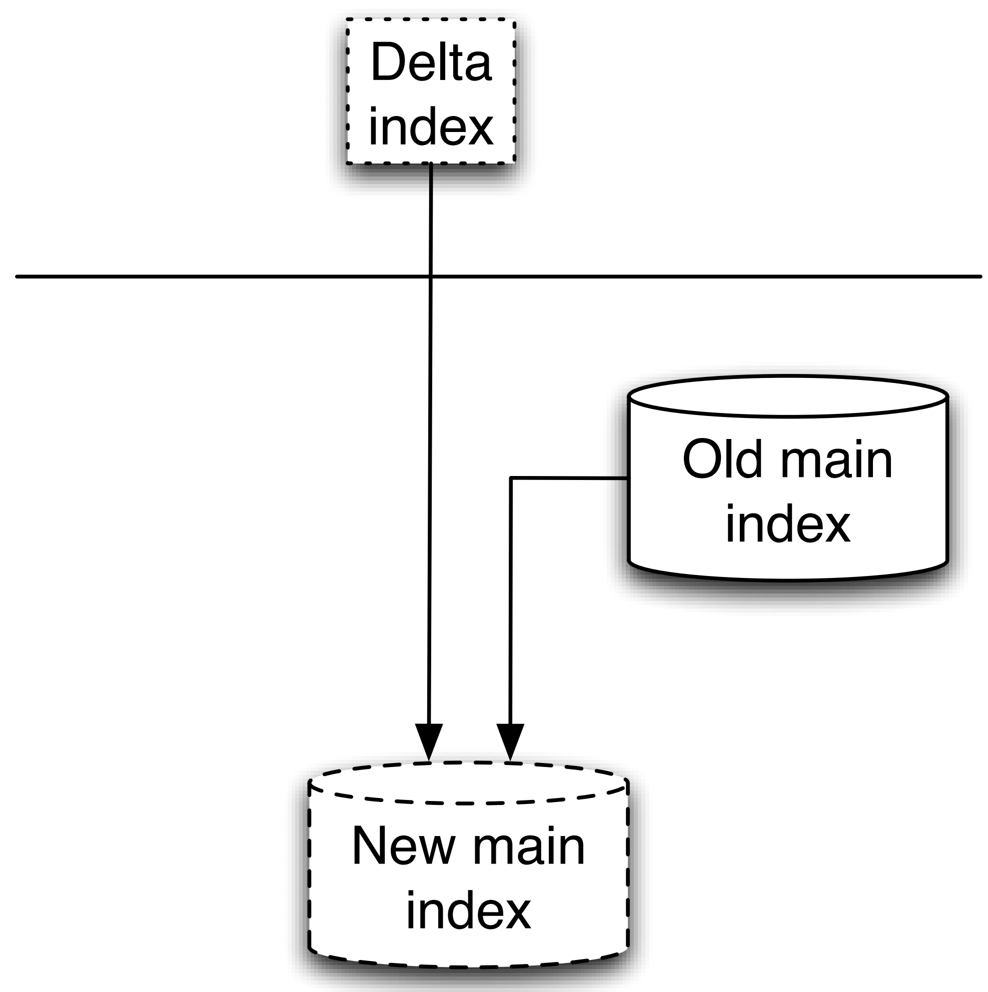
Incremental Update

- Each inverted list contains additional empty space at the end.
- New documents are appended to the empty space in the list.
- If the extra space allocated in an inverted list is full.
  - inverted list may be reallocated on disk
  - inverted list is maintained in multiple fragments on disk



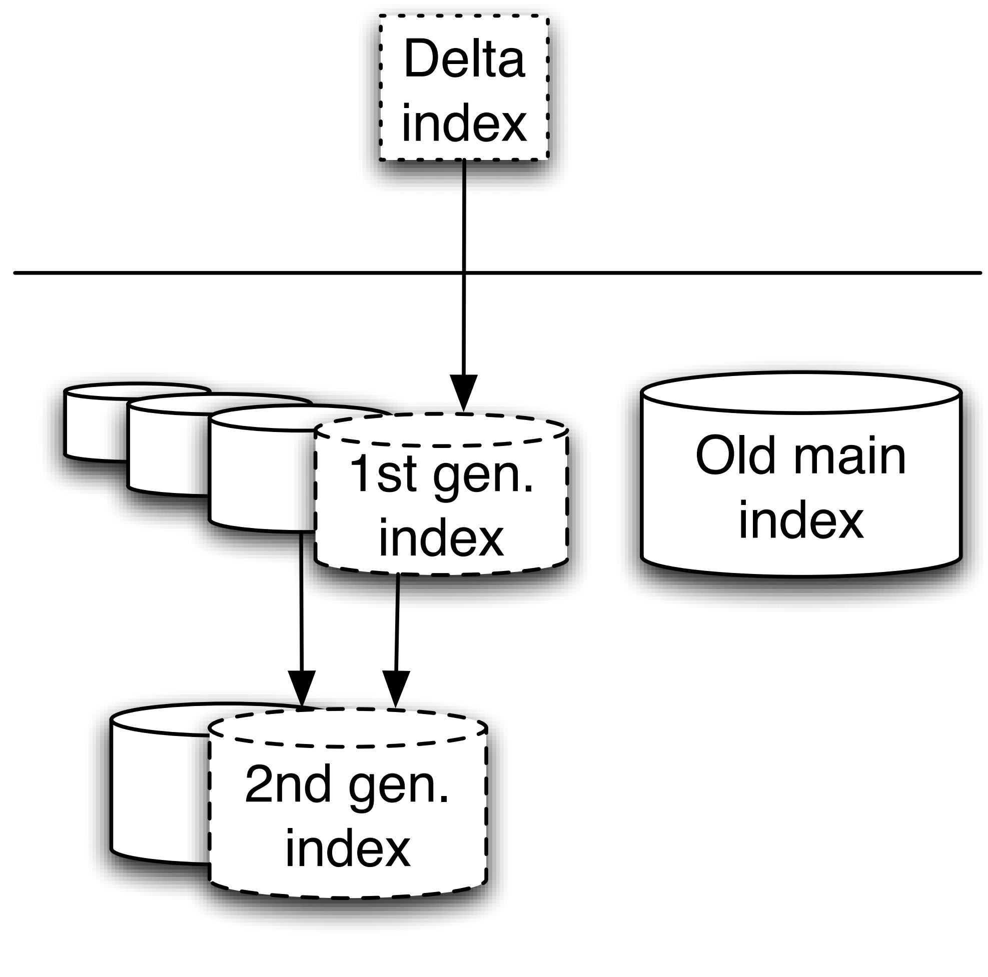
Immediate Merge

- Delta index is immediately merged to the old index and written to a new location on disk.
- Only one copy of the index is maintained on disk.



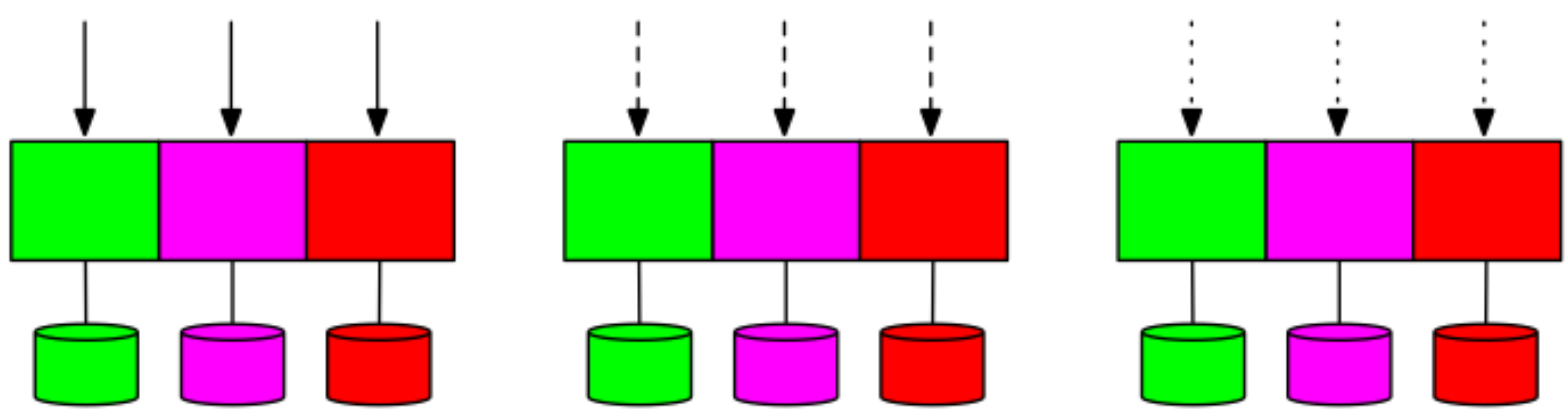
Lazy Merge

- Maintains multiple generations of the index on disk.
- Index generations are lazily merged.



Inverted Index Partitioning/Replication

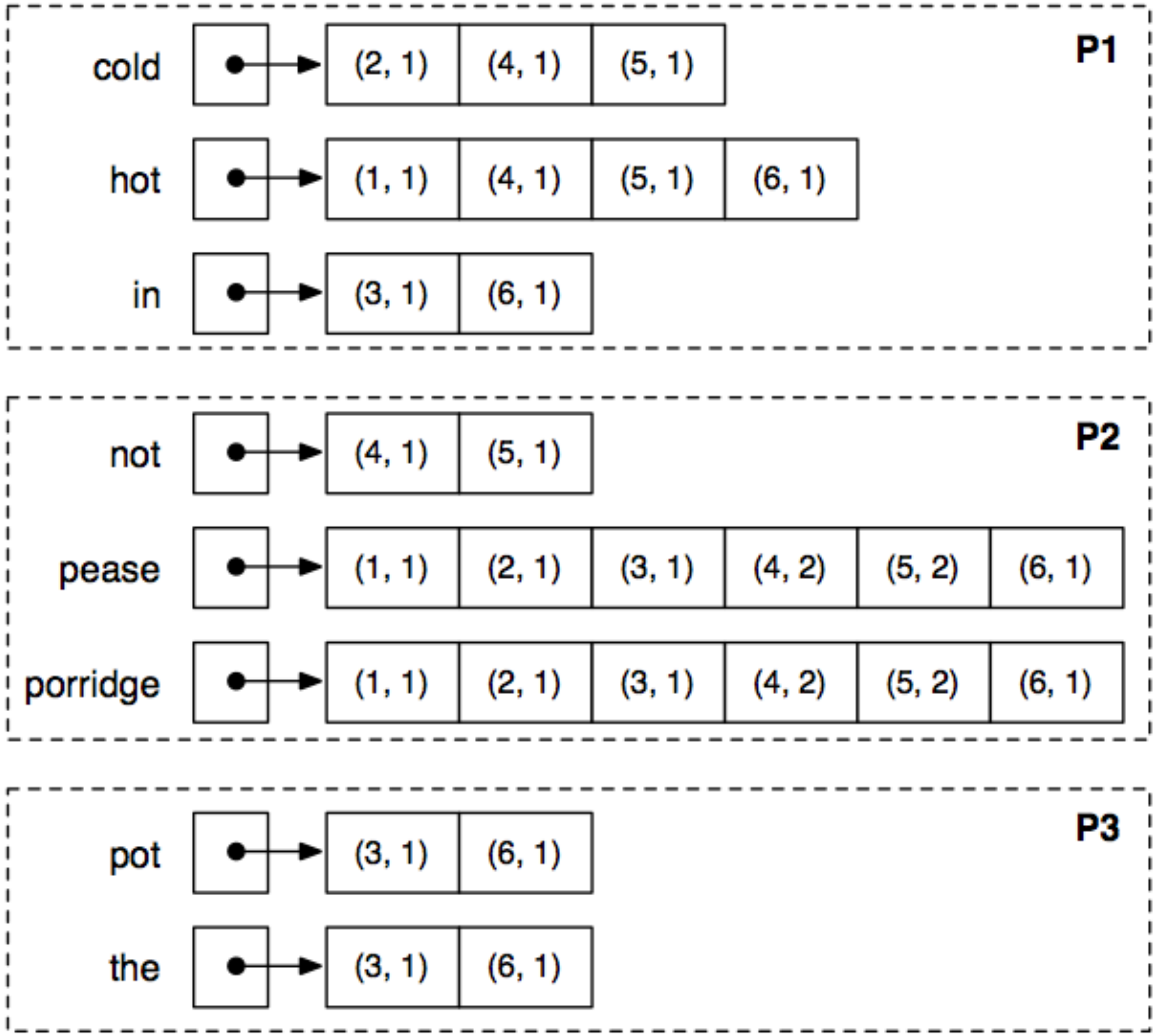
- In practice, the inverted index is
  - partitioned on thousands of computers in a large search cluster
    - reduces query response times
    - allows scaling with increasing collection size
  - replicated on tens of search clusters
    - increases query processing throughput
    - allows scaling with increasing query volume
    - provides fault tolerance



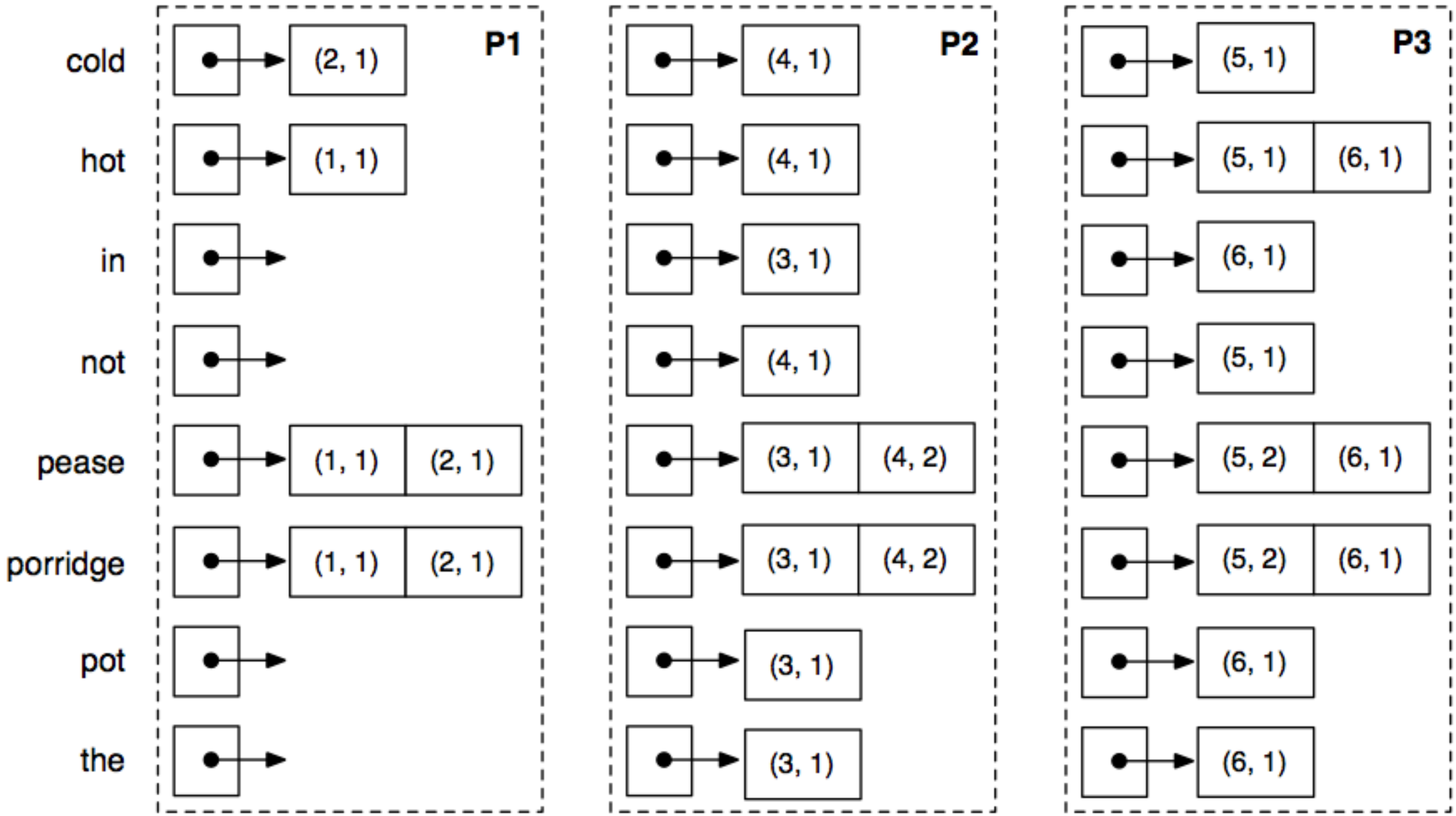
Inverted Index Partitioning

- Two alternatives for partitioning an inverted index
  - term-based partitioning
    - T inverted lists are distributed across P processors
    - each processor is responsible for processing the postings of a mutually disjoint subset of inverted lists assigned to itself
    - single disk access per query term
  - document-based partitioning
    - N documents are distributed across P processors
    - each processor is responsible for processing the postings of a mutually disjoint subset of documents assigned to itself
    - multiple (parallel) disk accesses per query term

Term-Based Index Partitioning



Document-Based Index Partitioning



Comparison of Index Partitioning Approaches

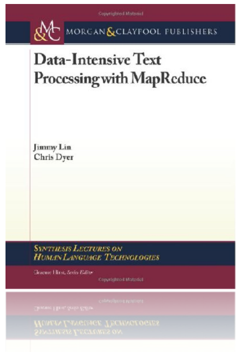
	Document-based	Term-based
Space consumption	Higher	Lower
Number of disk accesses	Higher	Lower
Concurrency	Lower	Higher
Computational load imbalance	Lower	Higher
Max. posting list I/O time	Lower	Higher
Cost of index building	Lower	Higher
Maintenance cost	Lower	Higher

Inverted Index Partitioning

- In practice, document-based partitioning is used
  - easier to build and maintain
  - low inter-query-processing concurrency, but good load balance
  - low query processing time
  - high throughput is achieved by replication
  - more fault tolerant
- Hybrid techniques are possible (e.g., term partitioning inside a document sub-collection).

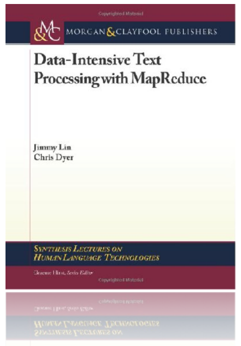
Indexing with MapReduce

- **Map over documents**
  - Emit *term* as key, *(docno, tf)* posting as value
  - Emit other information as necessary (e.g., term position)
- **Group postings by term**
  - Sort the postings (by docid)
  - Write postings to disk



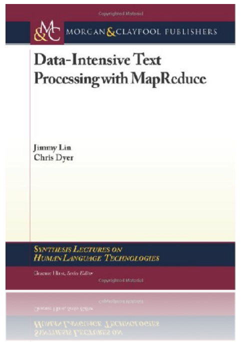
Indexing with MapReduce (I)

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray                ▷ histogram to hold term frequencies
        for all term t ∈ doc d do               ▷ processes the doc, e.g., tokenization and stopword removal
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(term t, posting ⟨n, H{t}⟩)     ▷ emits individual postings

class Reducer
    method Reduce(term t, postings [⟨n1, f1⟩ . . .])
        P ← new List
        for all ⟨n, f⟩ ∈ postings [⟨n1, f1⟩ . . .] do
            P.Append(⟨n, f⟩)                    ▷ appends postings unsorted
        P.Sort()                                ▷ sorts for compression
        Emit(term t, postingsList P)
```



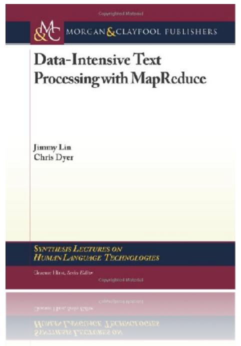
Indexing with MapReduce (II)

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do                      ▷ builds a histogram of term frequencies
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(tuple ⟨t, n⟩, tf H{t})                ▷ emits individual postings, with a tuple as the key

class Partitioner
    method Partition(tuple ⟨t, n⟩, tf f)
        return Hash(t) mod NumOfReducers              ▷ keys of same term are sent to same reducer

class Reducer
    method Initialize
        t_prev ← ∅
        P ← new PostingsList
    method Reduce(tuple ⟨t, n⟩, tf [f])
        if t ≠ t_prev ∧ t_prev ≠ ∅ then
            Emit(term t, postings P)                  ▷ emits postings list of term t_prev
            P.Reset()
        P.Append(⟨n, f⟩)                              ▷ appends postings in sorted order
        t_prev ← t
    method Close
        Emit(term t, postings P)                      ▷ emits last postings list from this reducer
```



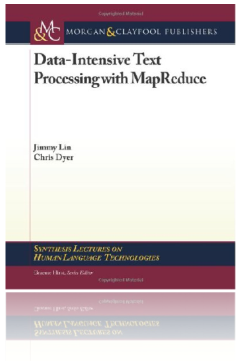
Indexing with MapReduce (III)

```
class Mapper
    method Initialize
        M ← new AssociativeArray                          ▷ holds partial lists of postings
    method Map(docid n, doc d)
        H ← new AssociativeArray                          ▷ builds a histogram of term frequencies
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            M{t}.Add(posting ⟨n, H{t}⟩)                   ▷ adds a posting to partial postings lists
        if MemoryFull() then
            Flush()
    method Flush                     ▷ flushes partial lists of postings as intermediate output
        for all term t ∈ M do
            P ← SortAndEncodePostings(M{t})
            Emit(term t, postingsList P)
        M.Clear()
    method Close
        Flush()
```



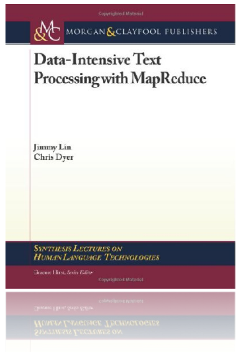
Indexing with MapReduce (III)

```
class Reducer
    method Reduce(term t, postingsLists [P_1, P_2, ...])
        P_f ← new List                                   ▷ temporarily stores partial lists of postings
        R ← new List                                     ▷ stores merged partial lists of postings
        for all P ∈ postingsLists [P_1, P_2, ...] do
            P_f.Add(P)
            if MemoryNearlyFull() then
                R.Add(MergeLists(P_f))
                P_f.Clear()
        R.Add(MergeLists(P_f))
        Emit(term t, postingsList MergeLists(R))         ▷ emits fully merged postings list of term t
```



Pagerank

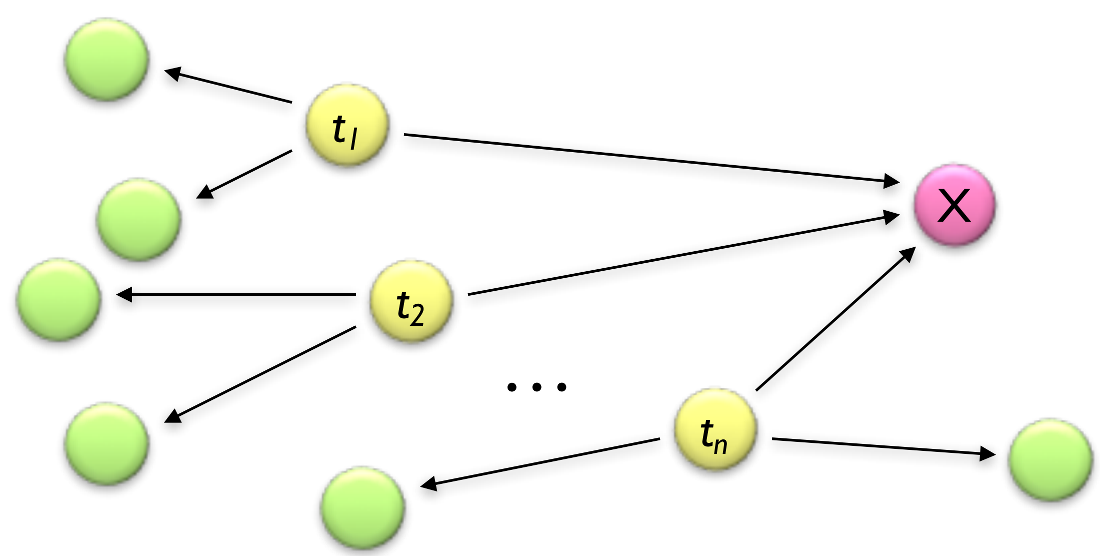
- **Random Surfers**
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- **Pagerank**
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- **Web Ranking**
  - One of thousands of features used in web search

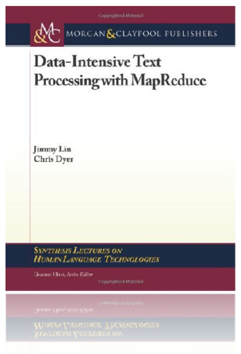


Definition

- Given page *x* with inlinks t_1, ..., t_n, where
  - *C(t)* is the out-degree of link *t*
  - *α* is probability of random jump
  - *N* is the total number of nodes in the graph

$$PR(x) = \alpha \left(\frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$



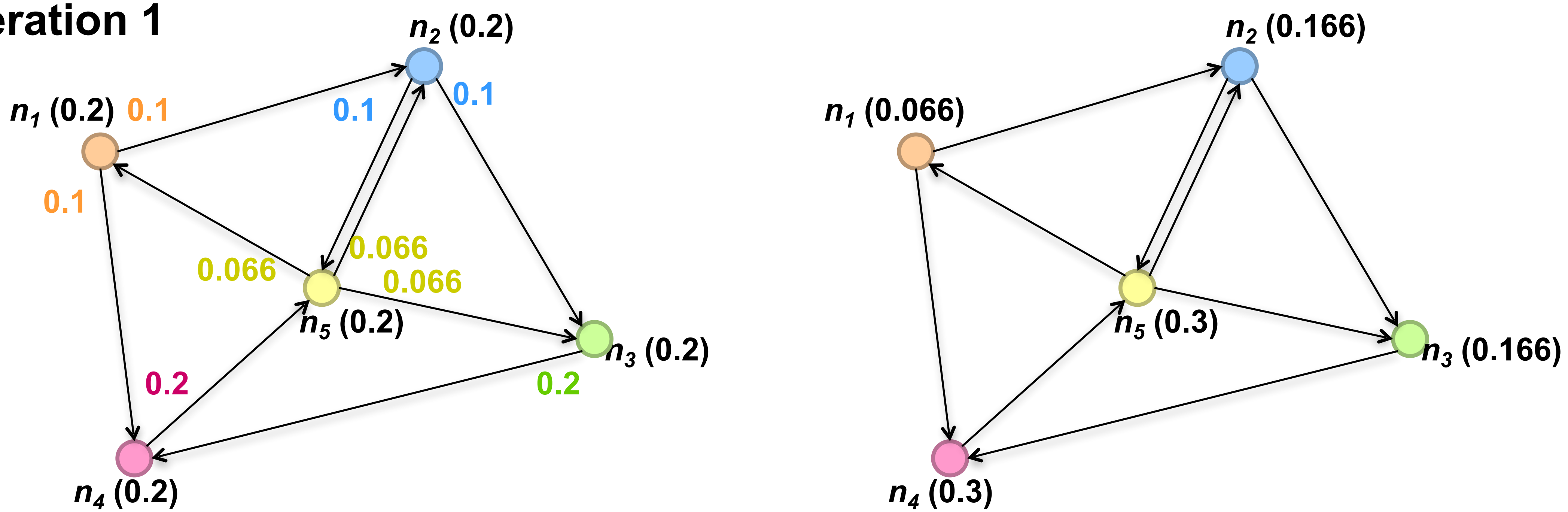


Algorithm Sketch

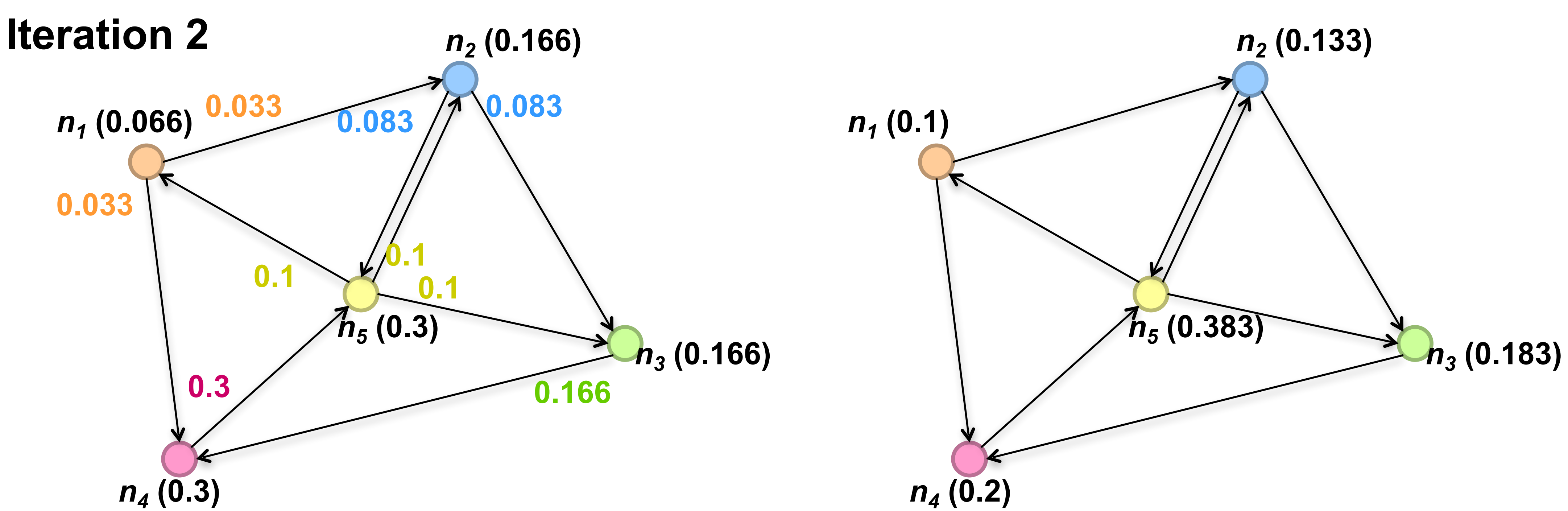
- Start with seed PR_i values
- Each page distributes PR_i mass to all pages it links to
- Each target page adds up mass from in-bound links to compute $PR_i + 1$
- Iterate until values converge

Simplified Algorithm Example (I)

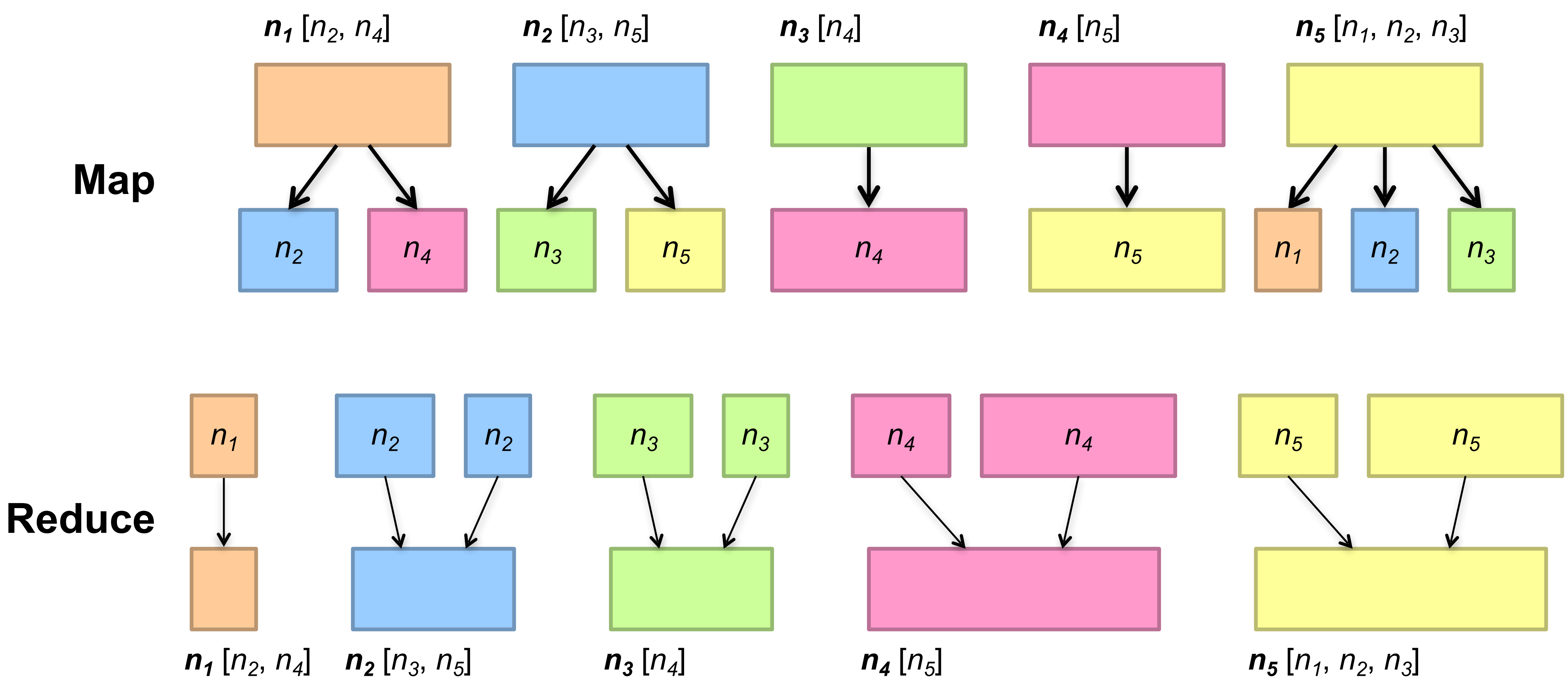
Iteration 1



Simplified Algorithm Example (II)



Pagerank in MapReduce (I)



Pagerank in MapReduce (II)

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sums incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```