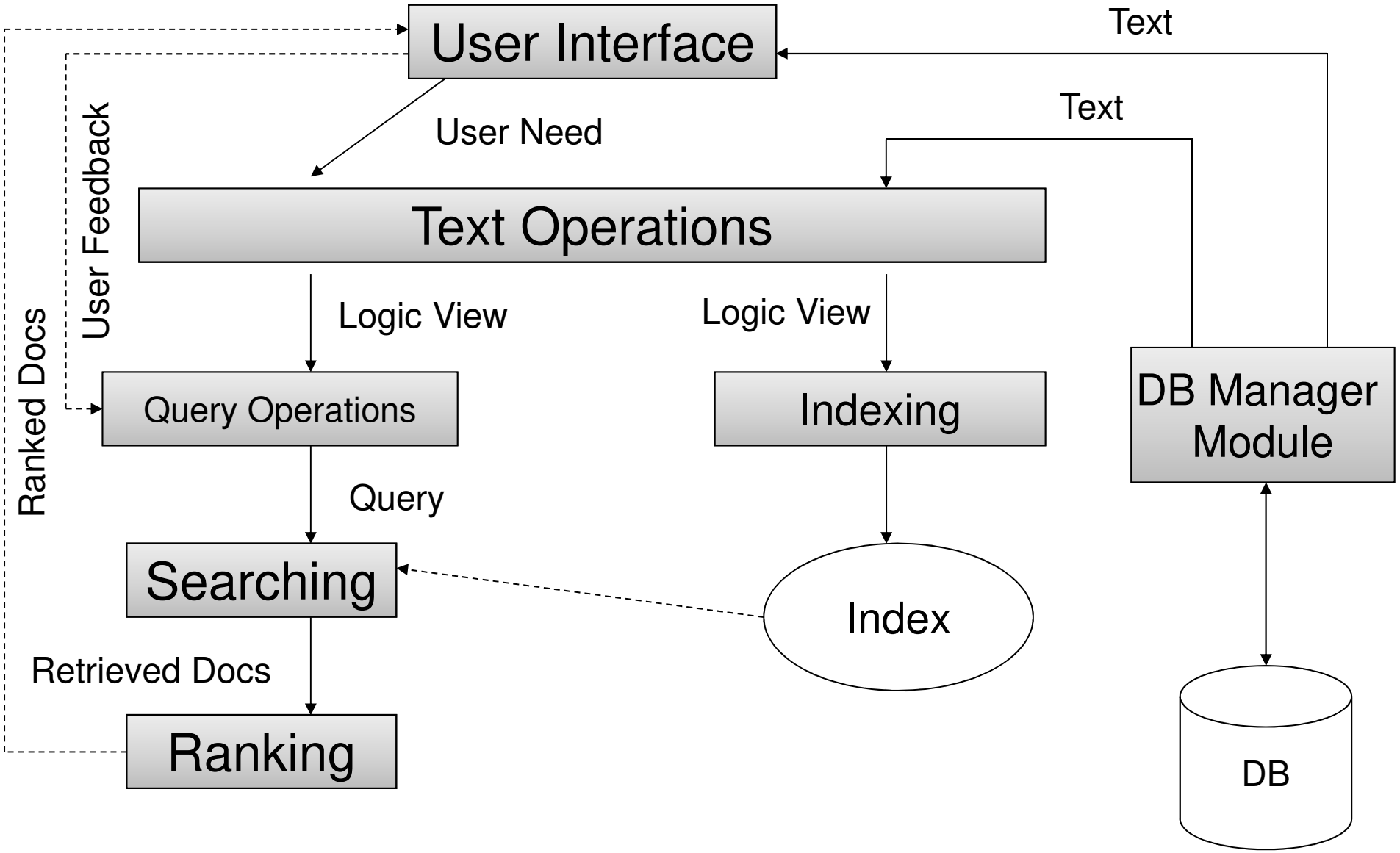
Inverted Indexes

Günther Specht
Eva Zangerle

Summer Term 2019

High-level View of a Textual IR system



Logic View of Documents

- Documents in a collection are usually represented through a set of **index terms** or **keywords**
  - Index term: any word which appears in the text of a document in the collection
  - Assumption: the semantics of the documents and of the user information need can be naturally expressed through sets of index terms (this is a considerable oversimplification of the problem)
- Keywords are:
  - Extracted directly from the text of the document
  - Specified by a human subject (e.g., tags, comments etc.)

- They provide a ***logic view*** *of the document.*
  - Retrieval systems representing a document by its full set of words use a *full text* logical view (or representation) of the documents.
  - With very large collections, the set of representative have to be reduced by means of TEXT OPERATIONS

Indexing Process

1. Define the text data source
   - usually done by the DB manager, which specifies:
     - Documents
     - Operations to be performed on them
     - Content model (i.e., the content structure and what elements can be retrieved)
2. The content operations transform the original documents and generate a ***logical view*** of them
3. An ***index*** of the text is built on logical view
   - The index allows *fast searching* over large volumes of data. Different index structures might be used, but the most popular one is the ***inverted file***
   - The resources (time and storage space) spent on defining the text database and building the index are amortized by querying the retrieval system many times

Retrieval Process

1. The user first specifies a ***user need***
   - User-level *query (*e.g., keywords); might also be done implicitly (RecSys)
2. The user need is parsed and transformed by the same content operations applied to the indexed contents.
3. *Query operations* provide a system representation for the user need as a system-level query
4. The query is processed to obtain the *retrieved documents.*
   - Fast query processing is made possible by the index structure previously built.
5. The retrieved documents are ranked according to a *likelihood* of relevance.
6. The user then examines the set of ranked documents in the search for useful information.
   - he might pinpoint a subset of the documents seen as definitely of interest and initiate a user feedback cycle.

Indexing: Data Structures

Index

- Database index (e.g., B-Tree)
- Avoid linearly scanning texts for each query
- Speed up search

- Required:
  - Content of each document
  - Which document contains which keywords/index terms?
  - Fast lookup

Boolean Term-Document Incidence Matrix

- *Index* documents in advance – simple approach

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	**1**	**1**	**0**	**0**	**0**	**1**
Brutus	**1**	**1**	**0**	**1**	**0**	**0**
Caesar	**1**	**1**	**0**	**1**	**1**	**1**
Calpurnia	**0**	**1**	**0**	**0**	**0**	**0**
Cleopatra	**1**	**0**	**0**	**0**	**0**	**0**
mercy	**1**	**0**	**1**	**1**	**1**	**1**
worser	**1**	**0**	**1**	**1**	**1**	**0**

1 if document contains word, **0** otherwise

- 32 000 distinct terms
- Query: ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***

Incidence Vectors

- Generating a 0/1 vector for each term
  - To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented)
    ➡ bitwise *AND*.

```
      110100
AND   110111
AND   101111
    = 100100
```

Bigger Corpora

- More realistic example
  - Consider ***n*** = 1 mio documents, each with about 1,000 terms
  - Avg. 6 bytes/term including spaces/punctuation
    - 6 GB of data in the documents.
  - Say there are *m* = 500K *distinct* terms among these
  - Building the whole matrix (half a trillion entries) not feasible
    - But matrix is sparse
    - Record only the 1 positions (99.8% are zero)

The central idea for an so-called ***Inverted Index***

Inverted Index

- *Indices* are data structures designed to make search faster
- Inverted Index is the most common data structure for text search
  - Most efficient and flexible structure
  - General name for a class of structures
  - Why inverted?
    - Similar to a *concordance* found in any textbook
    - Documents are associated with words (normally: words part of documents)
    - Words occur redundantly
- Core of all modern web search engines

Example "Collection"

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Four sentences from the Wikipedia entry for *tropical fish* [1]

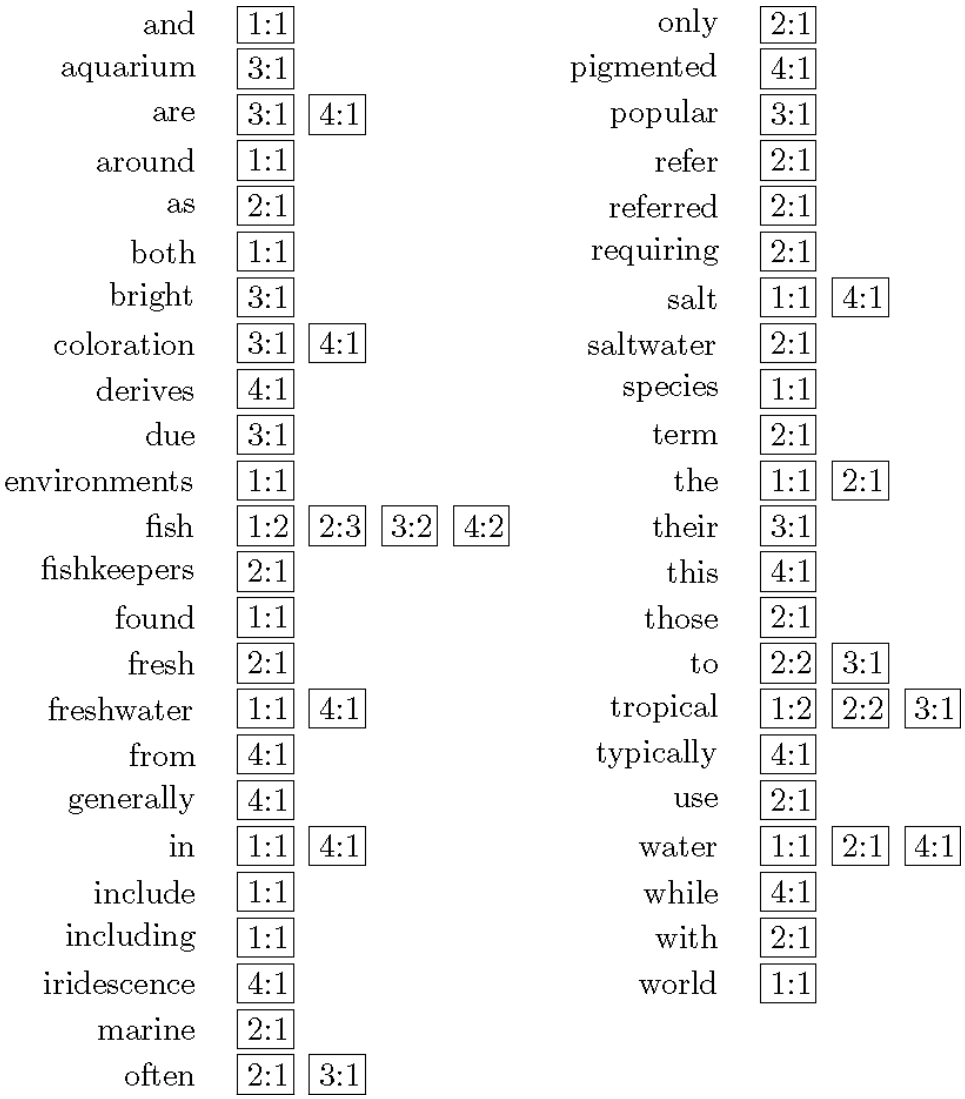
Example Inverted Index

Index sorted by words!

and	1			only	2		
aquarium	3			pigmented	4		
are	3	4		popular	3		
around	1			refer	2		
as	2			referred	2		
both	1			requiring	2		
bright	3			salt	1	4	
coloration	3	4		saltwater	2		
derives	4			species	1		
due	3			term	2		
environments	1			the	1	2	
fish	1	2	3	4	their	3	
fishkeepers	2			this	4		
found	1			those	2		
fresh	2			to	2	3	
freshwater	1	4		tropical	1	2	3
from	4			typically	4		
generally	4			use	2		
in	1	4		water	1	2	4
include	1			while	4		
including	1			with	2		
iridescence	4			world	1		
marine	2						
often	2	3					

Example Inverted Index with Counts

supports better ranking algorithms (see later)



Phrases, Proximity, Zones

- What about phrases?
  - University Innsbruck
- Proximity
  - Find Torvalds NEAR Linux
  - Capture position information in documents
- Zones in documents

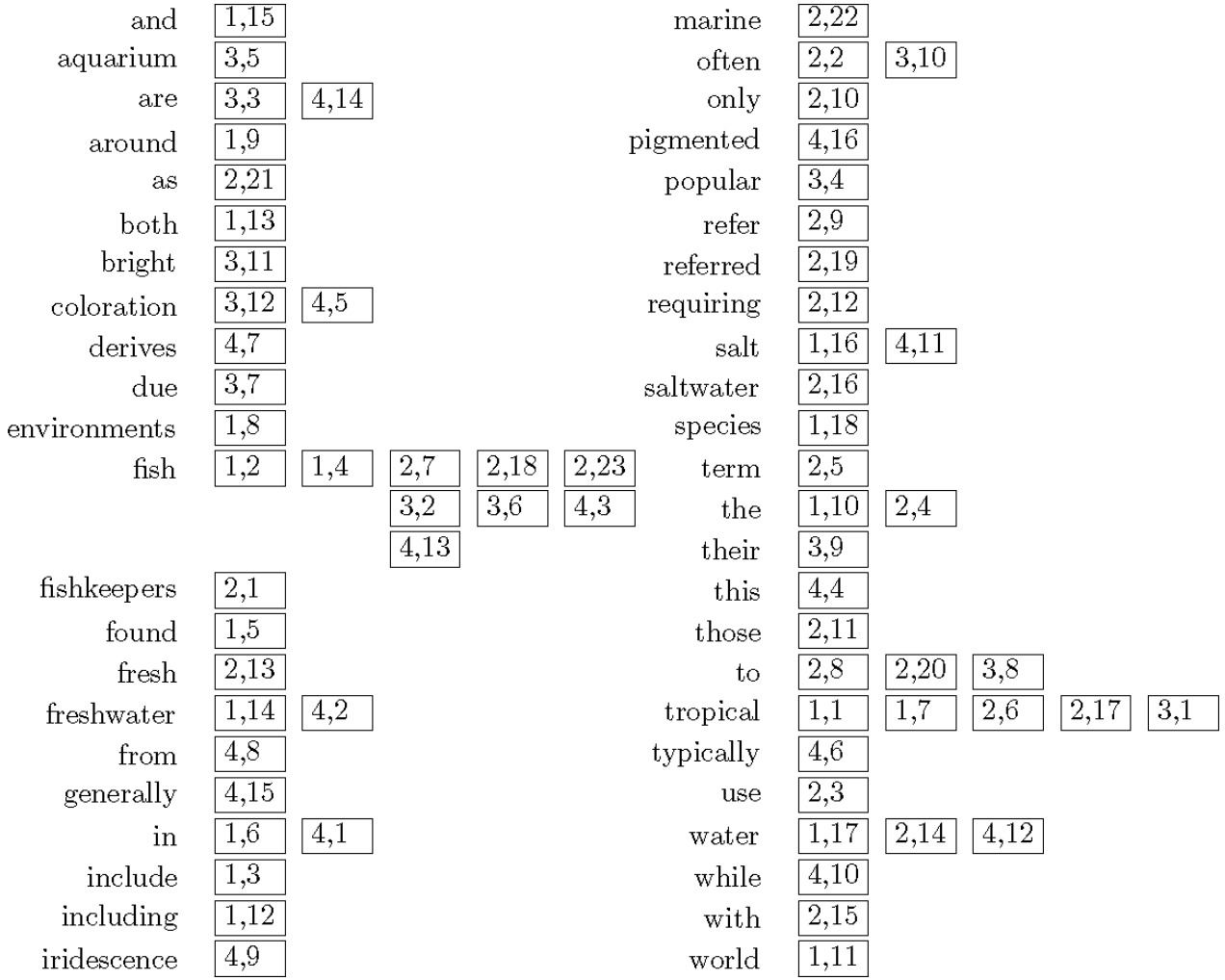
Positional Index required!

Use of Positional Indexes

- For each ***term:*** *store* entries of the form:
  <number of docs containing ***term***;
  doc1: position1, position2 … ;
  doc2: position1, position2 … ;
  etc.>

Inverted Index with Positions

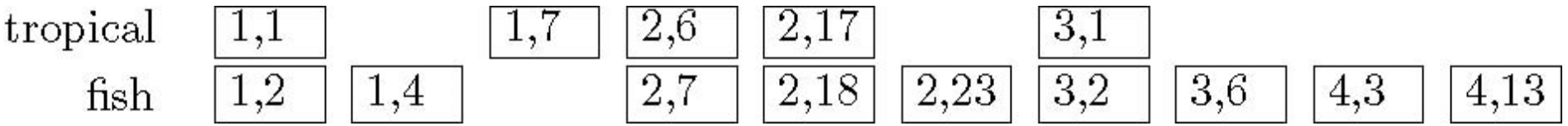
supports proximity matches
Depends on ranking algorithm



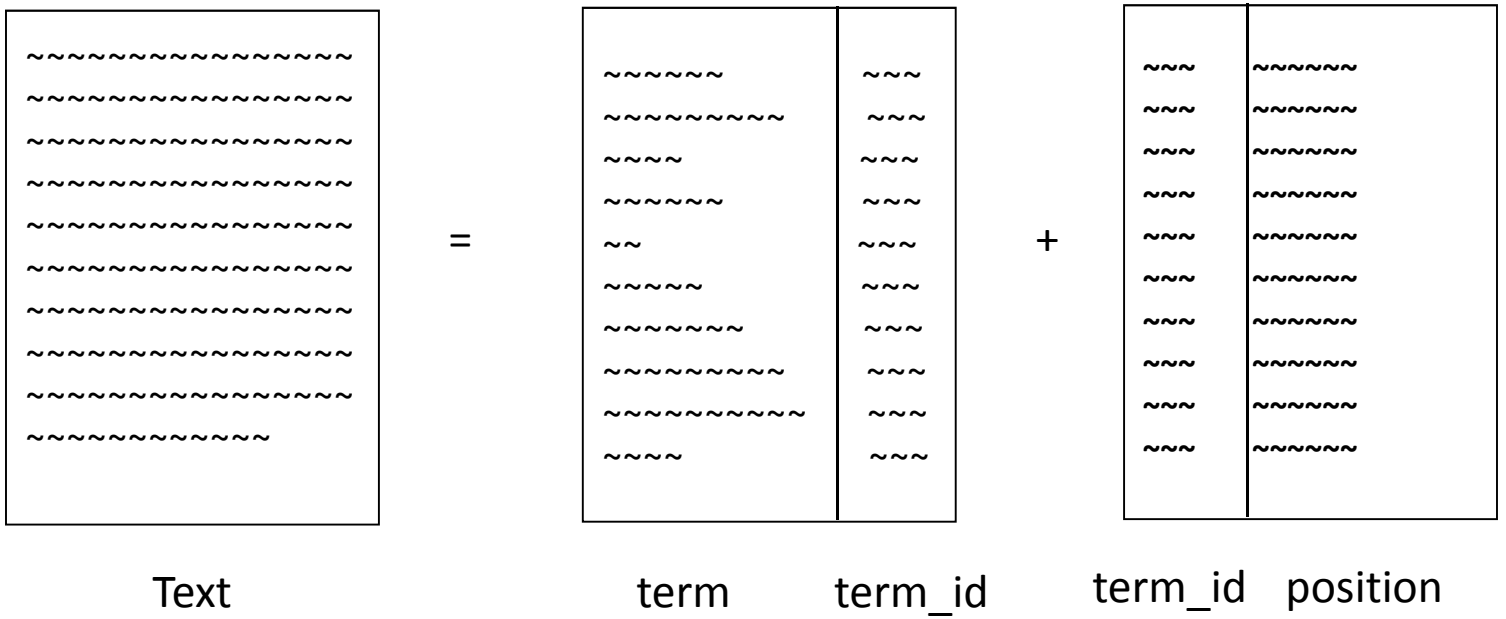
Proximity Matches

- Matching phrases or words within a window
  - e.g., “tropical fish", or “find fish within 5 words of tropic”
- Solution: Word positions in inverted lists
  - Positional index size factor of 2-4 over non-positional index

e.g. for query “`tropical fish`”:



Inverted Index - Truncation



- Truncation at the words‘ ends is enabled (e.g., salt*)
- What about *water?
- A second, inverted index is necessary for truncation at the beginning!
- What about *ltwa*?

Fields and Extents

- Document structure is useful in search
  - *Field* restrictions
    - E.g., date, from, etc.
  - Some fields more important
    - E.g., title
- Use extent lists
  - Represent extents using word positions
  - Inverted list records all extents for a given field type
  - E.g.: *inverted index says us that fish is included in title (title: position 1-3)*

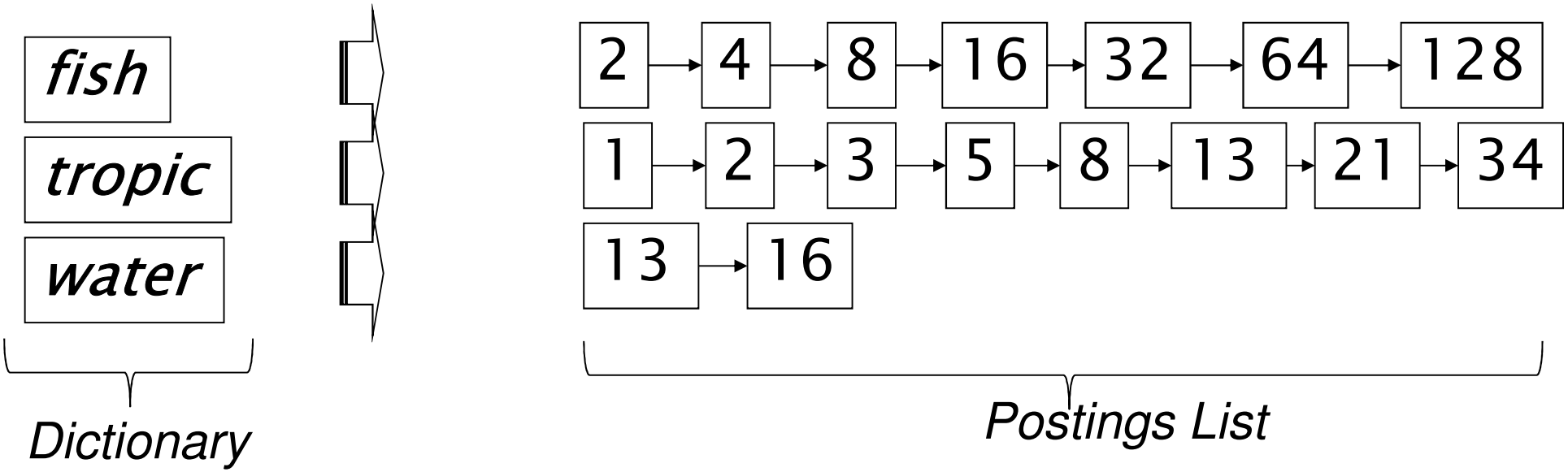
Inverted Index Details

- Each index term in the *dictionary* is associated with an *inverted list*
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each list entry is called a *posting, builds together a postings list* or *inverted list*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given an unique number
  - Lists are usually *document-ordered*

    See later

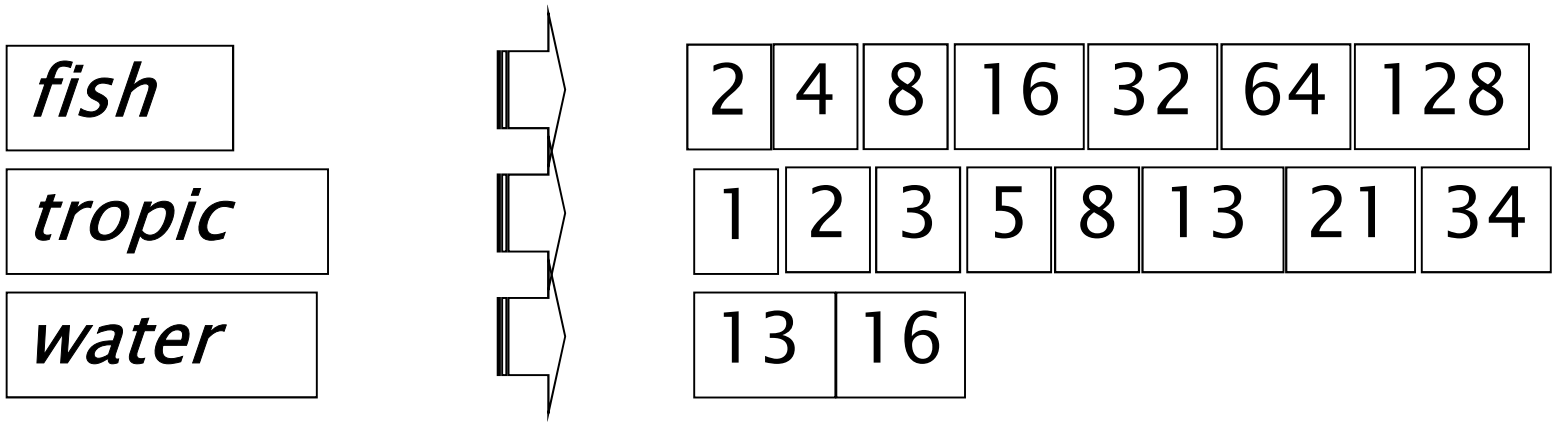
Inverted Index - Data Structure I

- In memory
  - Linked lists generally more preferable than arrays
    - Dynamic space allocation
    - Insertion of terms into documents easy
    - Space overhead of pointers



Inverted Index - Data Structure II

- On disc
  - Compressed, contiguous run of postings
    - No explicit pointers
    - Dictionary in memory
    - Minimize disk seeks to read into memory (see *Hardware basics*)



Query Processing

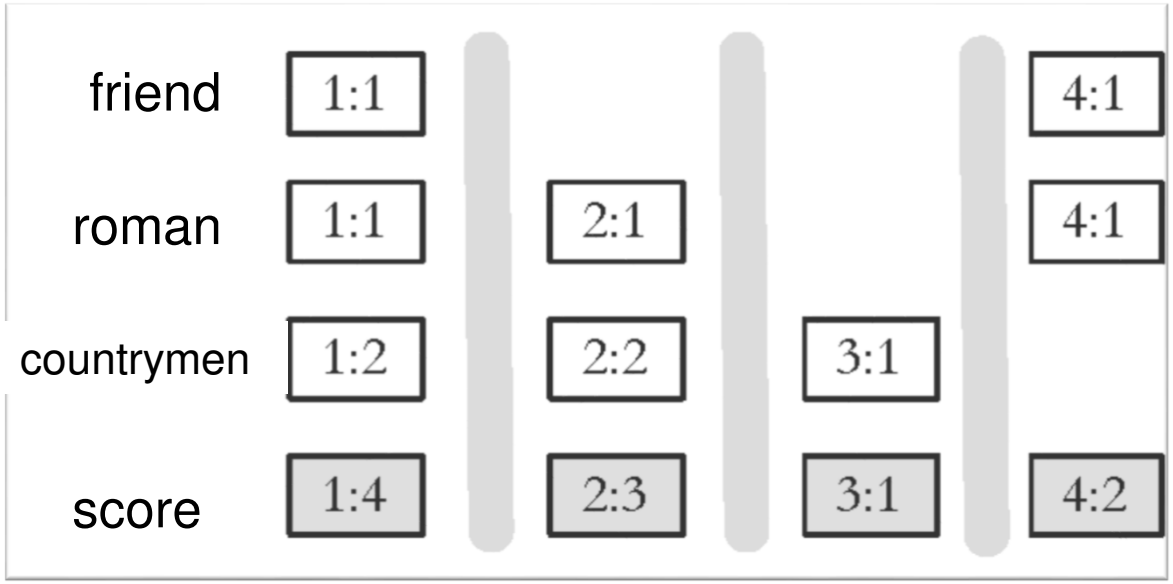
Query Processing

- Document-at-a-time
  - Calculates complete scores for documents, one document at a time
- Term-at-a-time
  - Processes term lists one at a time

- Both simple approaches have optimization techniques that significantly reduce time required to generate scores
  - See later

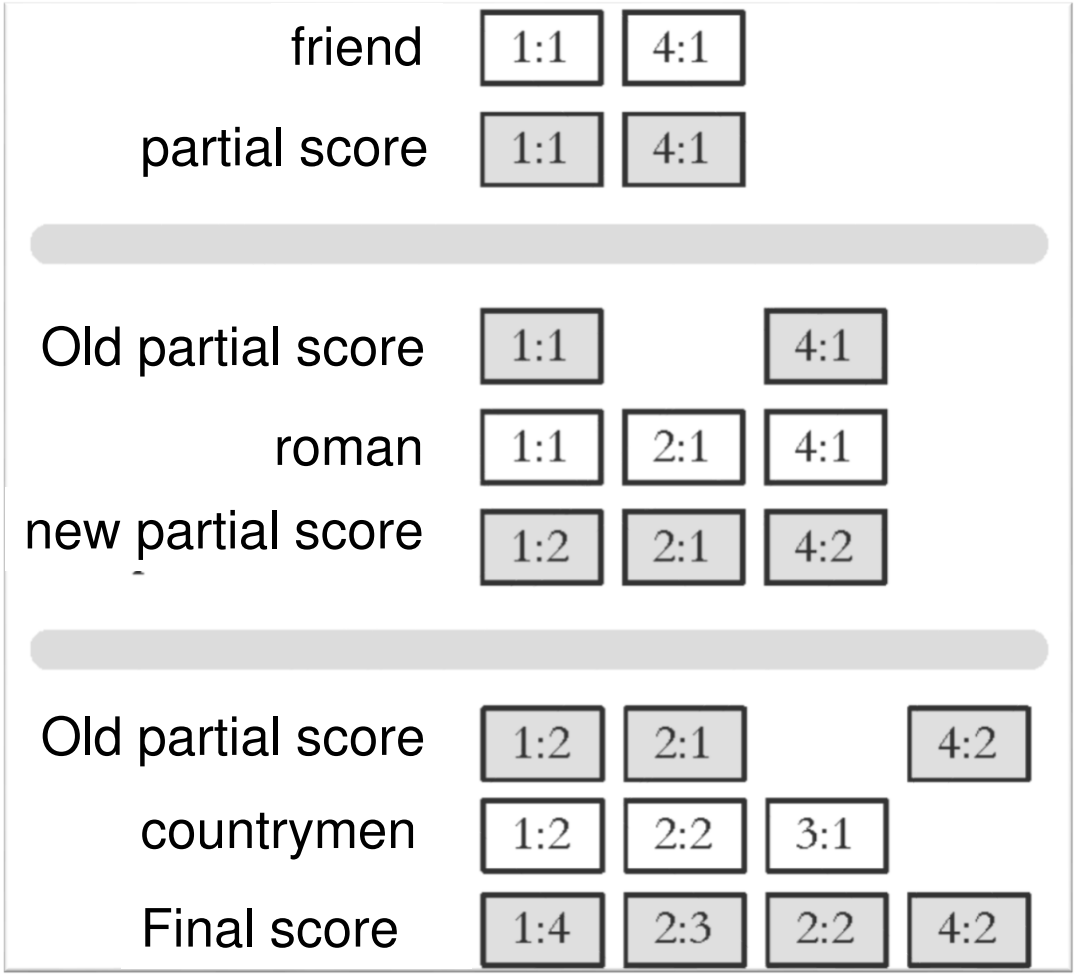
Document-at-a-Time

Query is „Friend roman countrymen“



1) Inverted list for each query word is fetched
   - sorted by docID, all stored in an array
2) At each document, all inverted lists are checked
   - calculate score

Term-at-a-Time



1) Fetch inverted list for every term (same as before)

2) Loop over each list for each search term

-store partial score in a hash table (*accumulators*)

- Additional memory usage
+Minimal disk seeking

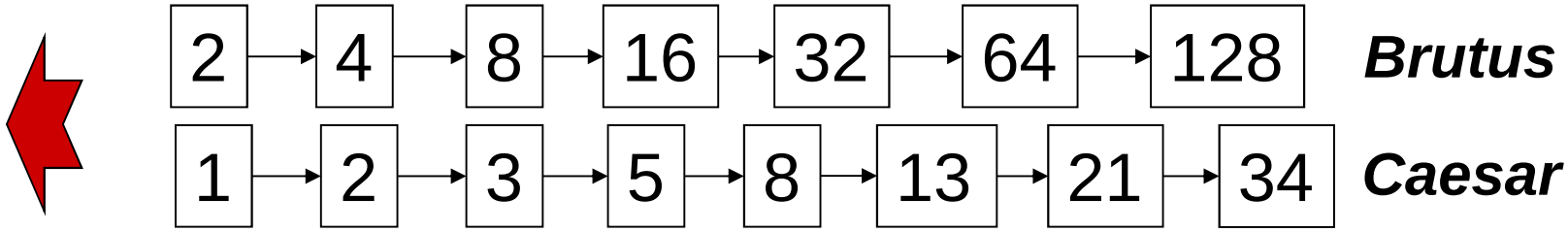
Query Processing: Optimizations

1) Conjunctive Processing

- All query terms must be contained
  - Works best when one term is *rare*

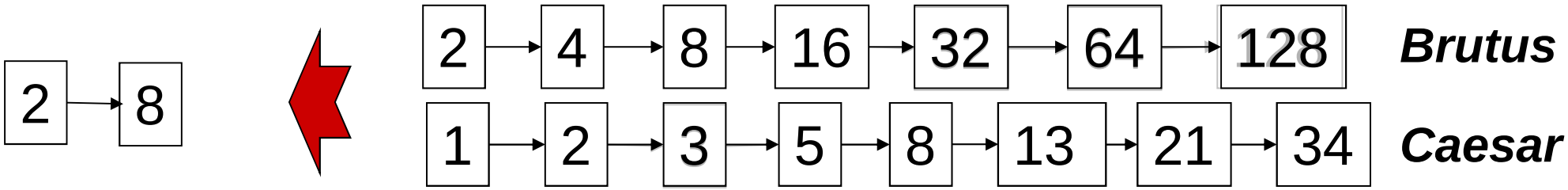
Brutus *AND* ***Caesar***

- Locate ***Brutus*** in the Dictionary;
  - Retrieve its postings.
- Locate *Caesar* in the Dictionary;
  - Retrieve its postings.
- "Merge" the two postings:



The Merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries
- If the list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.
- Crucial: postings sorted by docID

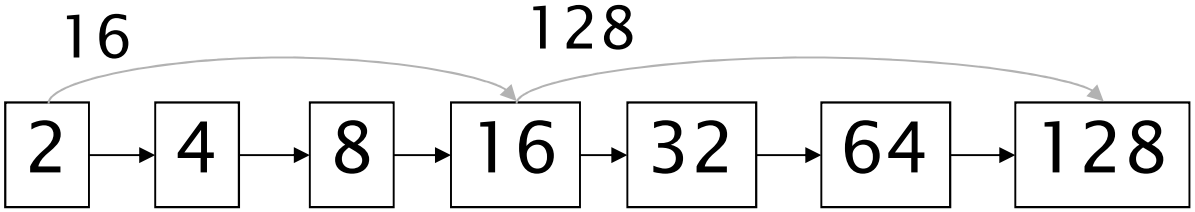


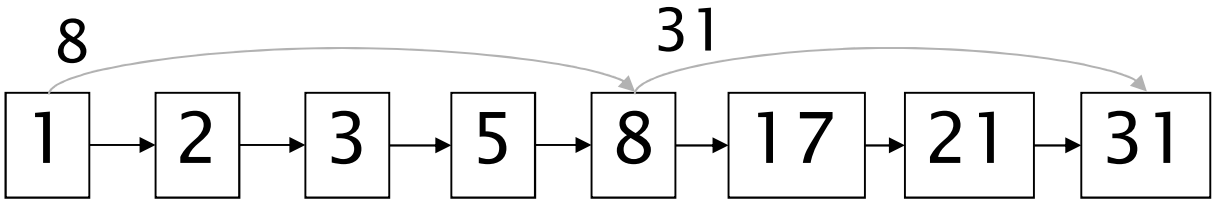
Query Optimization - Skipping

- Search involves comparison of inverted lists of different lengths
  - Can be very inefficient
  - “Skipping” ahead to check document numbers is much better
  - Compression makes this difficult

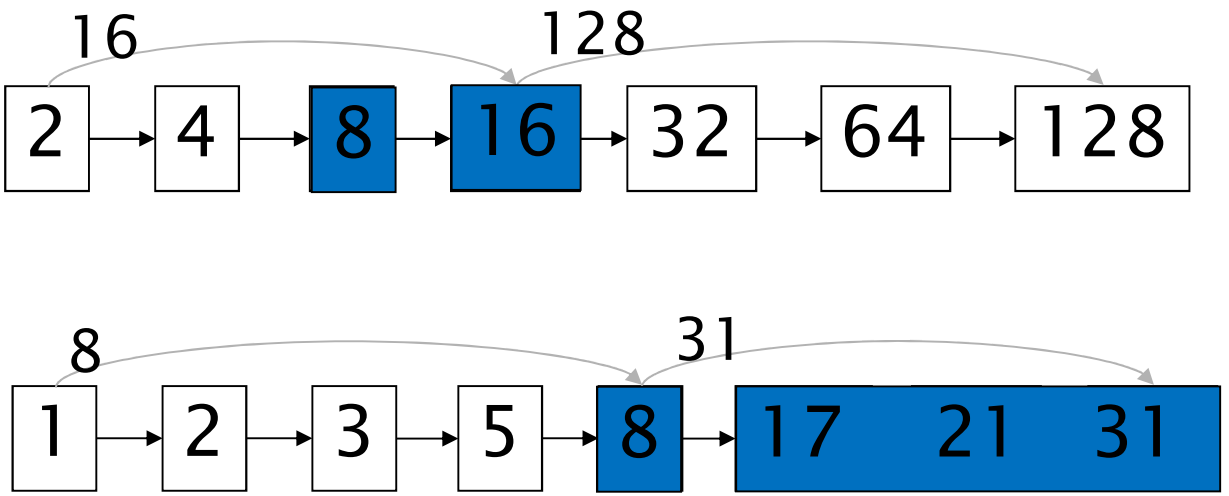
Augment Postings with Skip Pointers

- Why?
  - To skip postings that will not figure in the search results.
- How?
  - Has to be performed at indexing time
  - Where do we place skip pointers?





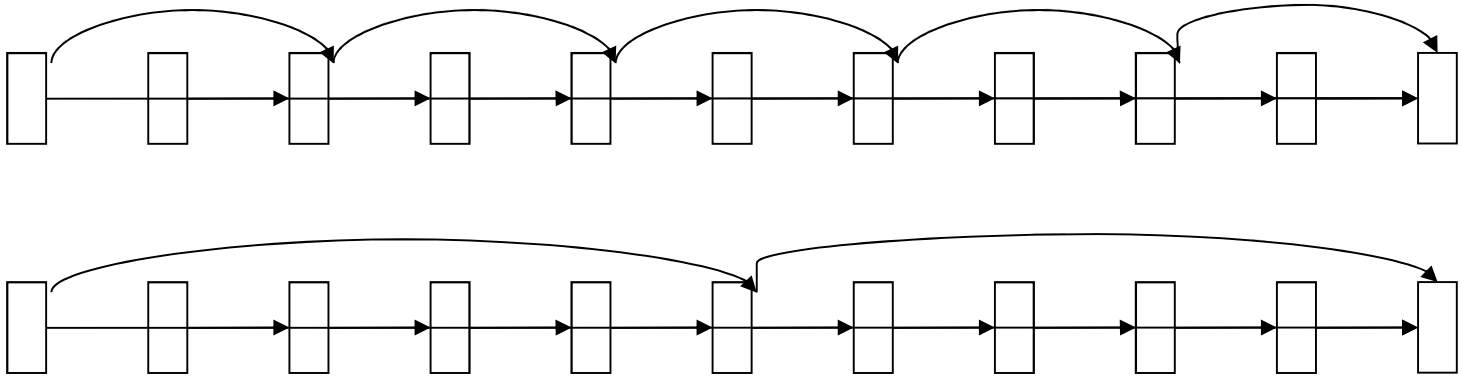
Query Processing with Skip Pointers



- Suppose we've stepped through the lists until we process **8** on each list.
- When we get to **16** on the top list, we see that its successor is **32**.
- But the skip successor of **8** on the lower list is **31**, so we can skip ahead past the intervening postings.

Where do we place skips?

- Tradeoff:
  - More skips → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers; more space required.
  - Fewer skips → few pointer comparison, but then long skip spans ⇒ few successful skips.



Further Query Optimizations

- Threshold methods
  - Minimum score needed
  - Return only documents with a threshold at least T
  - Approximation T‘ <= T
    - Lowest-scoring document currently
- List ordering
  - Until now documents are stored by docID
  - Could by document quality
    - some scores
    - user clicks
  - Not that flexible

Index Construction

Index Construction

- Main index structure
  - Inverted Index
  - How do we construct such an index?
  - Which strategies to use?
    - Limited ressources of a system
    - Hardware constraints

Hardware Basics

- Building an IR system
  - Many decisions are based on the characteristics of the computer hardware:
- Access to memory is faster (transferring vs. accessing a byte on disk)
  - Just a few clock cycles (5 x 10^-9)
  - Keep as much data as possible in memory (*Caching*)

- Seek time (moving disk head)
  - Takes a while to move to the correct part of the disc (5 ms in average)
  - Storage of chunks on disk contiguously
    - 10 MB of data as one chunk: 10*10^6 *2*10^-8 = 20 * 10^-2 = 0.2 sec
    - 10 MB in 100 non contiguous chunks: 0.2 + 100 * (5*10^-3) = 0.7 sec

- Disk I/O is block based (read into the *buffer*)
  - 8KB to 256 KB is common
  - Single byte means reading the entire block
  - Data transfers from disc handled by the system bus (not processor)
  - Processor is able to process data during disk I/O
    - Compression

Build the Index

- To build an index
  1. Collect the documents to be indexed
     - Friends, Romans, countrymen. | So let it be with Caesar…
  2. Tokenize the text, turning each document into a list of tokens (“Tokenizer”)
     - Friends | romans | countrymen | So
  3. Lingustic preprocessing (“Linguistic modules”)
     - friend | roman | countryman | so
  4. Index the documents, create an inverted list consisting of dictionary and postings (“Indexer”)

Simple Index Construction

- Bulding a *basic* inverted index by sort-based indexing:
  - *assumed: First 3 steps (last slide) have already been done*

    1) List of normalized tokens for each document
    - Token + Unique serial number (docID)

    2) Sorting the list (alphabetically)

    3) Organize the docIDs for each term into a posting list, compute statistics

  - *Here: Dictionary + posting list completely in-memory*

Indexer Steps

1) Sequence of (token, Document ID) pairs.

Document 1

I did enact Julius Caesar I was killed i' the Capitol;

Brutus killed me.

Document 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Key Step

2) Inverted file is sorted by terms & docID

sort command in Unix

= Core index step

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

▶

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Merging and Grouping

3) Merging and Grouping of terms; Compute statistics
The result is split into a *Dictionary* file and a *Postings* file.

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Differrent Indexes

- Limiting factor is the memory
  - Posting file of many large collections doesn‘t fit into main memory
  - Compression techniques can help, but do not solve the problem
  - Sorting on disc: no option!
  - *External sorting algorithm* required
    - Uses the disk for intermediate results
    - Sort part („block“) in memory

Blocked Sort-Based Indexing (BSBI)

- Parses documents into termID-docID pairs
  - Term to termID mapping, construction more efficient
- Segment the collection into blocks
- Accumlates the pairs in memory until a block (fixed size) is full („ParseNextBlock“)
  - Block fits into main memory, fast in memory sorting
- Block is inverted and written to disc
  - Sort the pairs
  - Collect all pairs with the same termID into a postings list
- Merge the results of different blocks

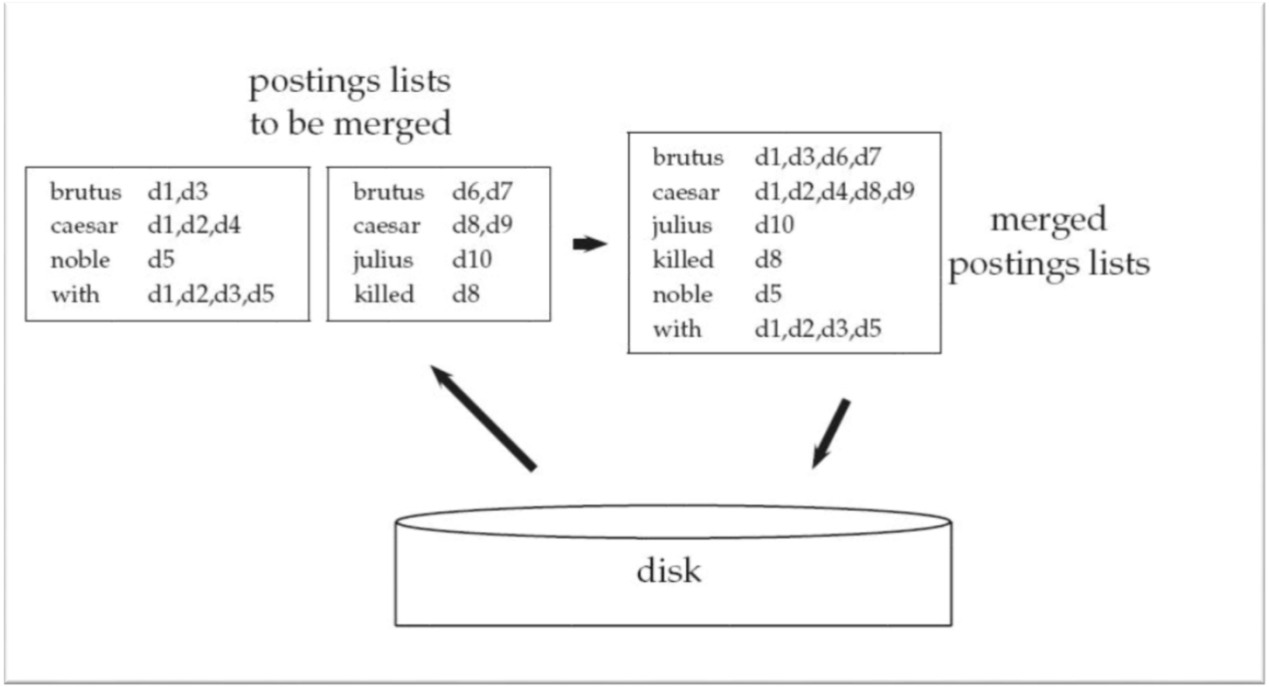
BSBI - Algorithm

```
BSBINDEXCONSTRUCTION()
1  n ← 0
2  while (all documents have not been processed)
3  do n ← n + 1
4     block ← PARSENEXTBLOCK()
5     BSBI-INVERT(block)
6     WRITEBLOCKTODISK(block, f_n)
7  MERGEBLOCKS(f_1, ..., f_n; f_merged)
```

BSBI - Merge

- Open all file blocks *simultaneously (read buffer)*
- Select lowest termID that has not been processed (priority queue), maintaining small *read buffers for all blocks*
- All posting lists for this term are read and merged, written to disk

- BSBI: O(n log n) (for sorting, but dominated by parsing & merging)



Single-Pass-In-Memory Indexing (SPIMI)

- BSBI needs a data structure for mapping terms to termsID
  - Does not fit in memory for very large collections
- SPIMI operates on terms (not termsID)
  - When a new term occurs, added to the dictionary (hash) and a new posting list is created
  - Posting is added directly to its posting list
    - BSBI: collecting and sorting first
  - Some memory is wasted (space for posting list is doubled each time new memory is allocated)
  - Blocks are written to disk (as in BSBI) and merged
- O(n)
  - *Linear scan through* for merging

Distributed Indexing

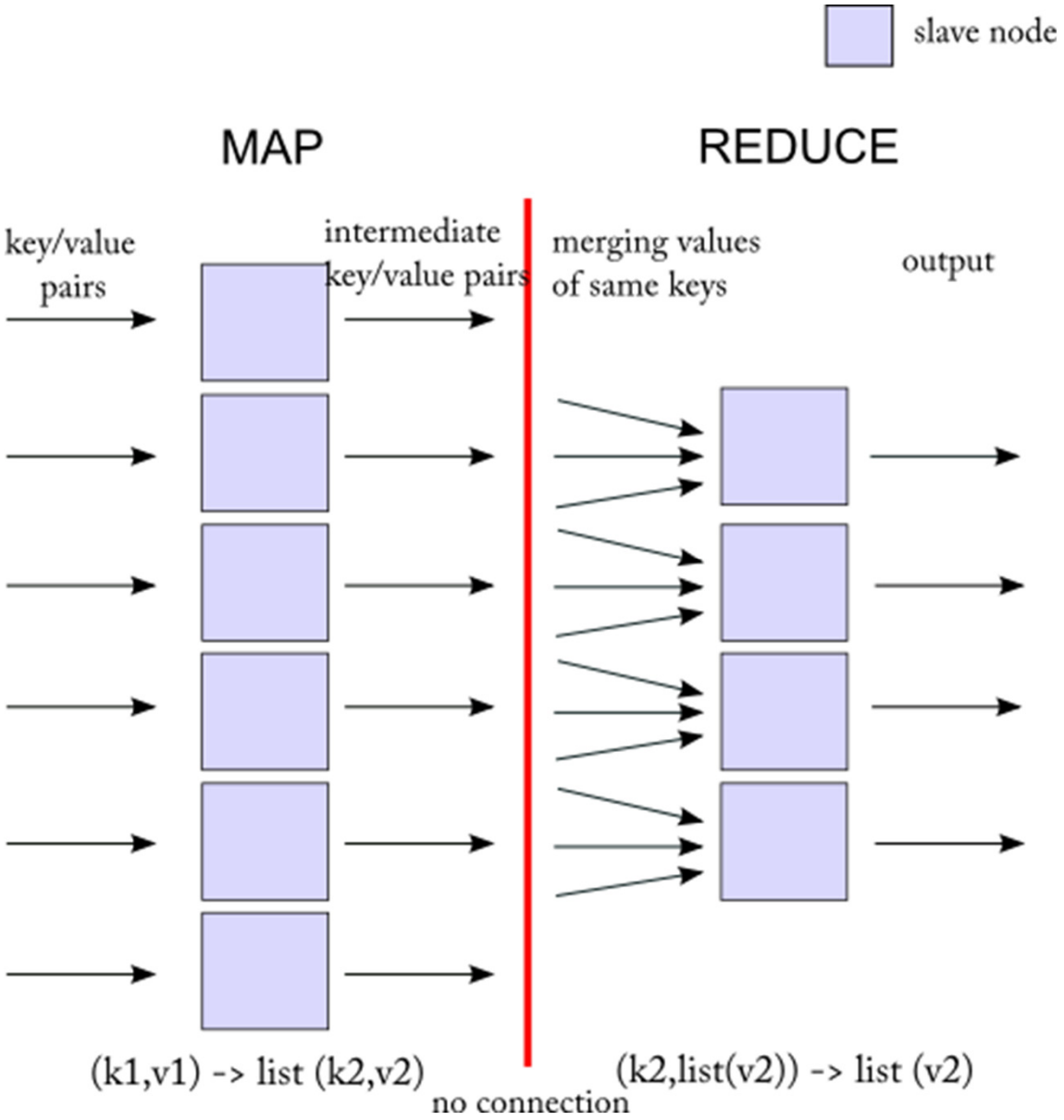
- Collections too large for single machines
  - Particularly true for WWW
- Large numbers of inexpensive servers used rather than larger, more expensive machines
- *Index Construction here*
  - *Instance of MapReduce, a general architecture for distributed computing* designed for indexing and analysis tasks

MapReduce Approach

- „MapReduce: Simplified Data Processing on Large Clusters“ (2004)
  - Hide distribution etc from end user

- Based on two constructs
  - Map and Reduce
    - Primitives presented in functional languages like Lisp
    - Input and output are key/value pairs
      - map (k1,v1) -> list(k2,v2) *(generating a set of intermediate keys)*
      - reduce (k2,list(v2)) -> list(v2) *(merging of same keys)*
    - No side effects allowed -> Parallel execution possible

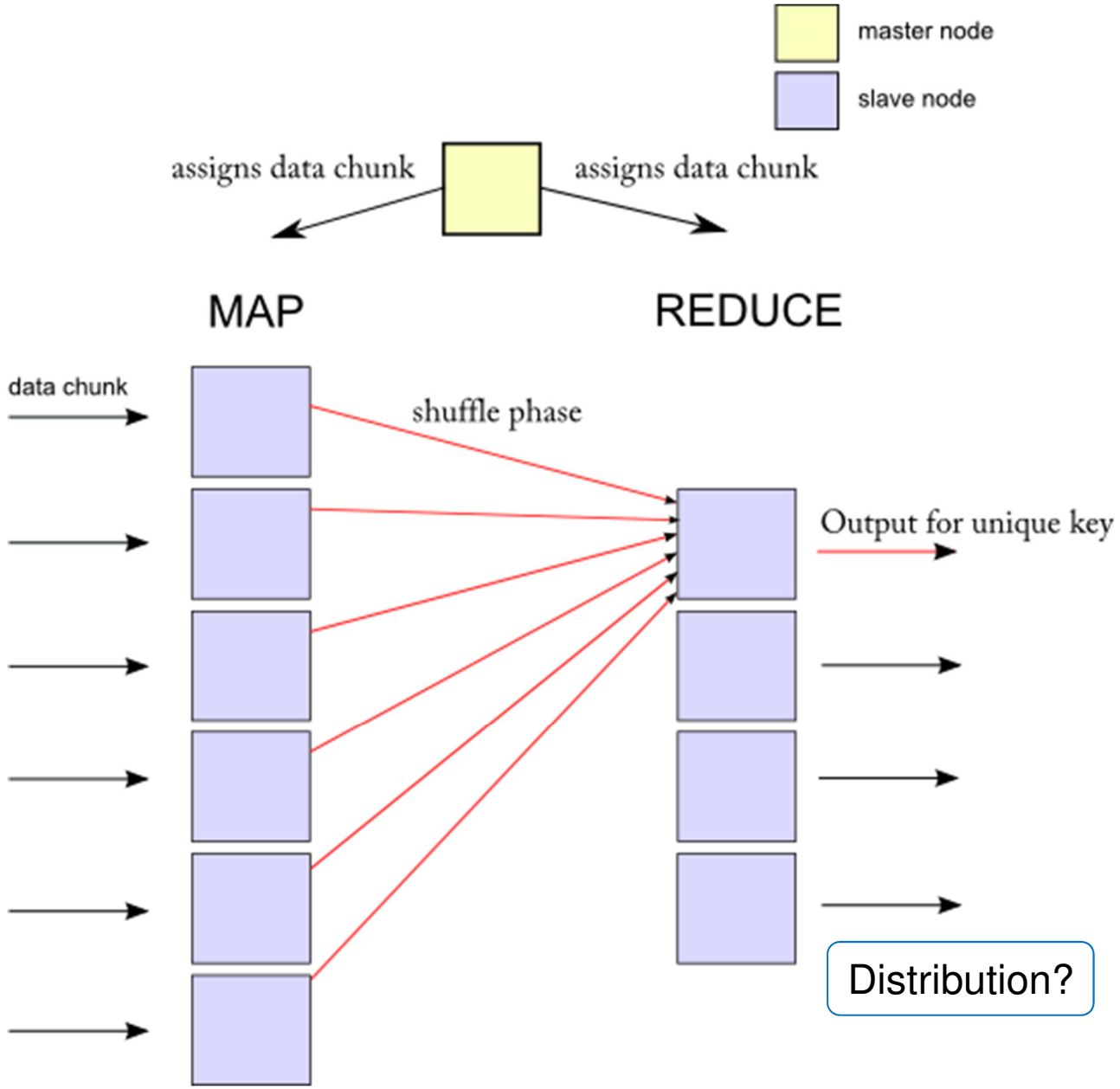
MapReduce Approach



MapReduce Approach

- Maintain a *master* machine directing the job
  - Assigns chunks to map tasks (*map store files to intermediate files*)
  - Scheduling across machines
  - Inter machine communication
    - Notifies reduce task where to find data (RPC)
  - Reassignment of tasks (machine errors)

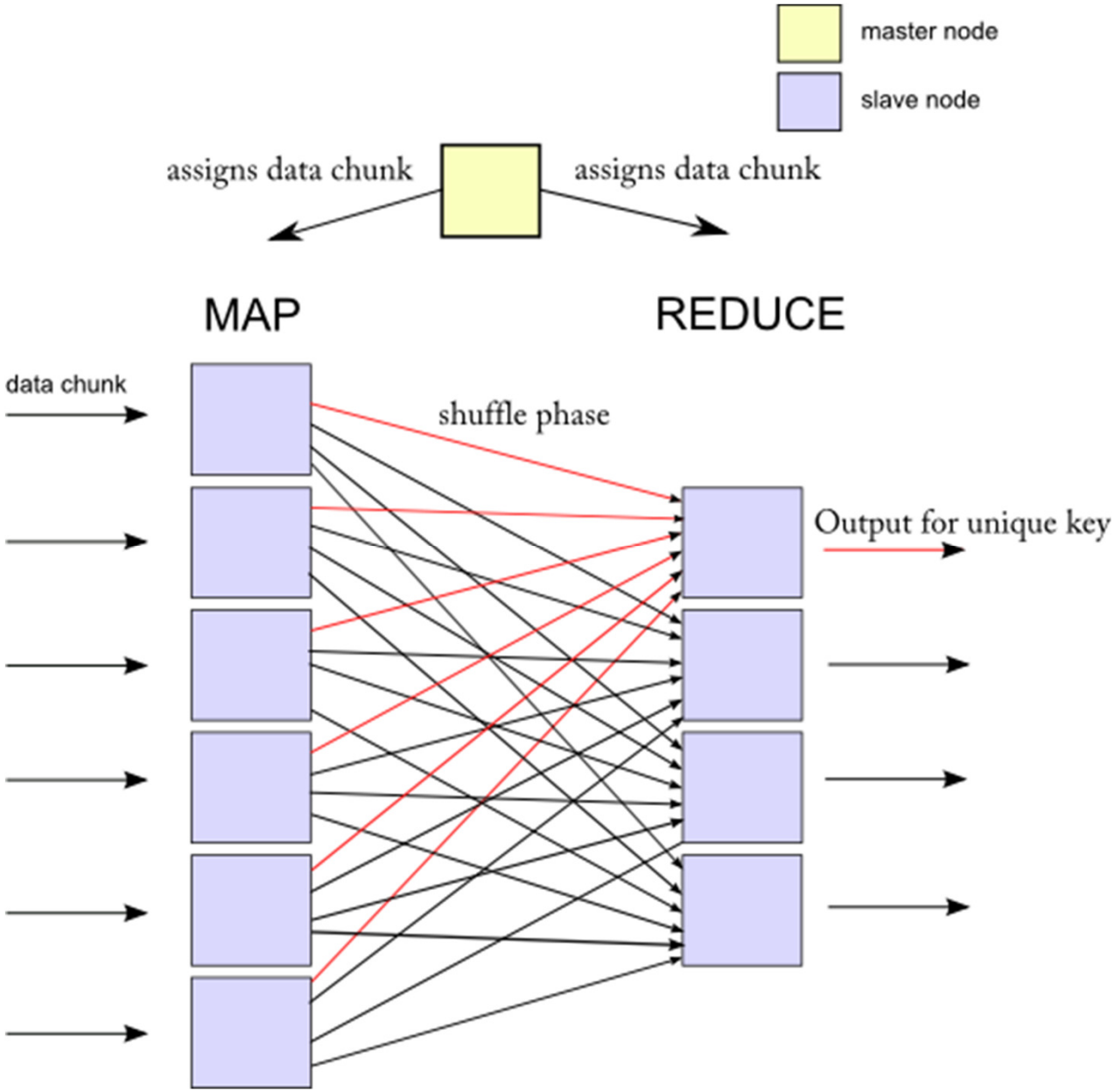
MapReduce Approach



MapReduce Approach

- Distribution
  - Programmer specifies amount of reduce tasks („buckets“)
    - Via XML configuration file
    - Should be set to 2x reduce tasks
      - Depending on hardware
  - Intermediate key space (map results) distributed over partitions
    - e.g. hash(key) **mod** R (R is number of reducer tasks)
    - RPC calls of Reducer to get intermediate data
      - Sorting by intermediate keys

MapReduce Approach

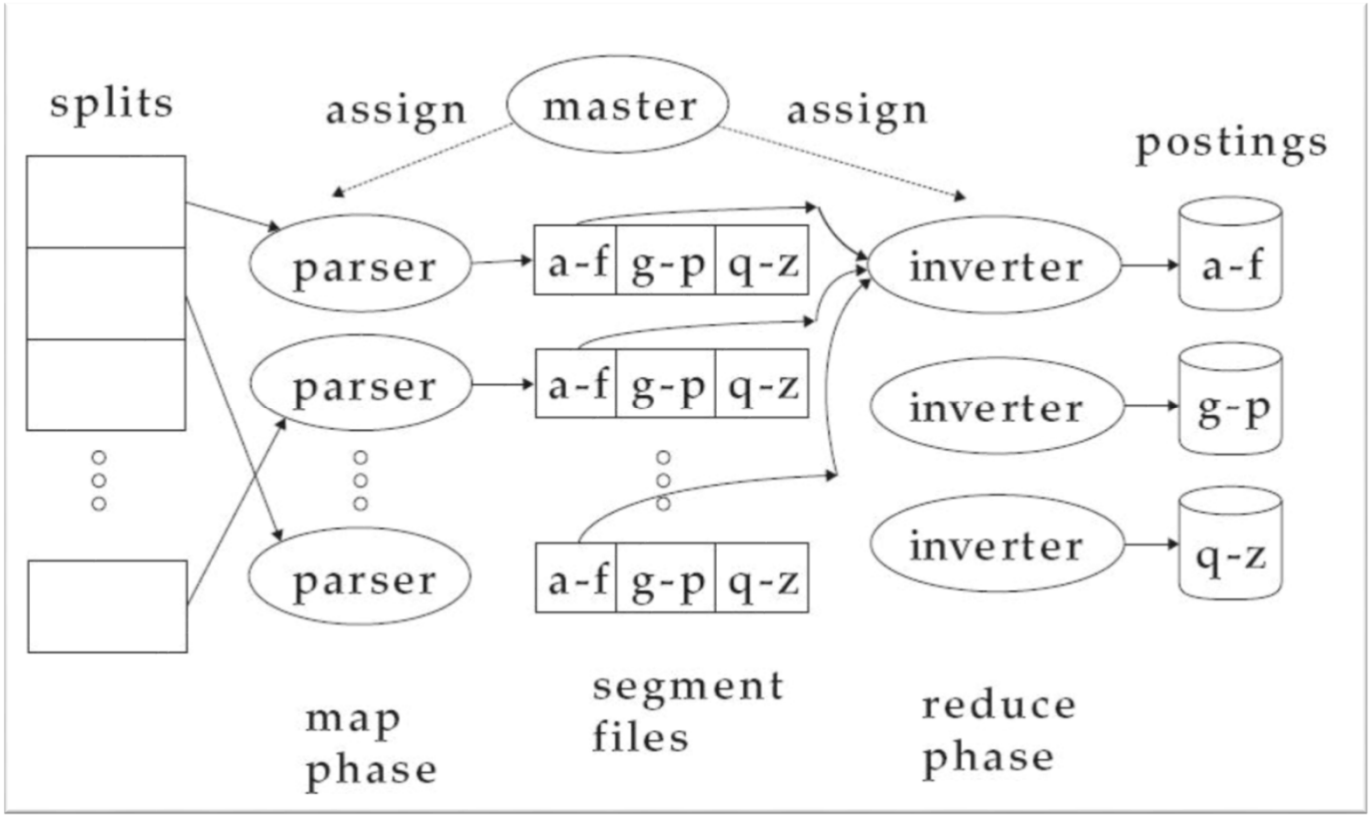


Parsers

- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (termID, docID) pairs
  - Mapping of terms to termsID
  - Like parsing task in BSBI
- Parser writes pairs into *j partitions*
  - Local machine, files called *local intermediate files*
- Each partition is a range of terms' first letters
  - (e.g., ***a-f, g-p, q-z***) – here *j* = 3.

Inverters

- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists



Apache Hadoop

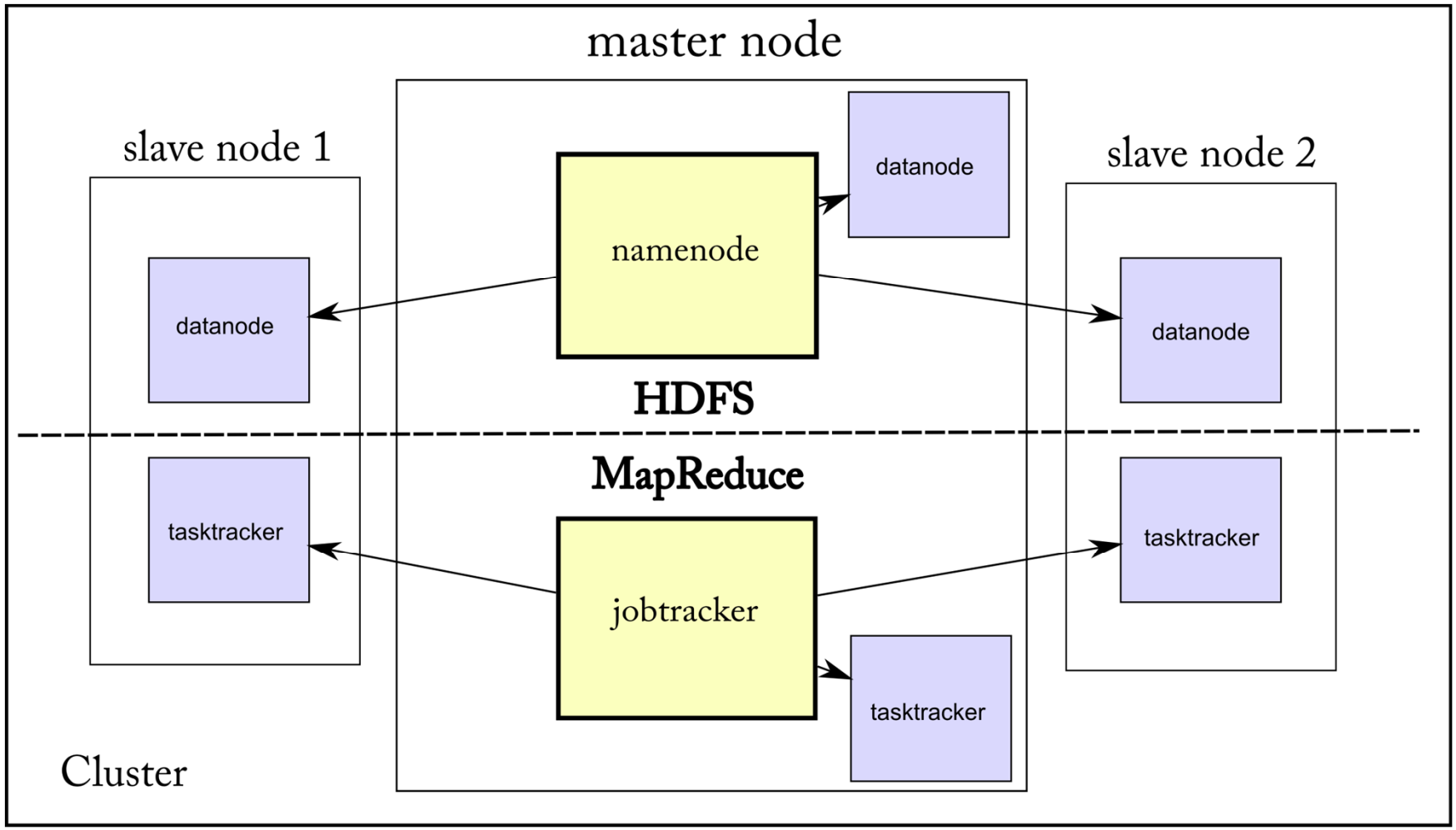
- Open source framework that supports data-intensive distributed applications
  - Yahoo, etc.
- Implements map reduce approach
  - For huge data sets to ensure parallelization and scalability
  - E.g Index creation, word count
  - *Jobtracker (master)* schedules map/reduce jobs to *tasktrackers (workers -> map/reduce tasks)*

Apache Hadoop

- Open source framework that supports data-intensive distributed applications
- Implements map reduce approach
  - For huge data sets to ensure parallelization and scalability
  - E.g Index construction
  - *Jobtracker (master)* schedules map/reduce jobs to *tasktrackers (workers -> map/reduce tasks)*

- In combination with a distributed file system (HDFS)
  - *Namenode (master)* distributes data to *datanodes*
  - Intelligent master necessary to fully utilize system

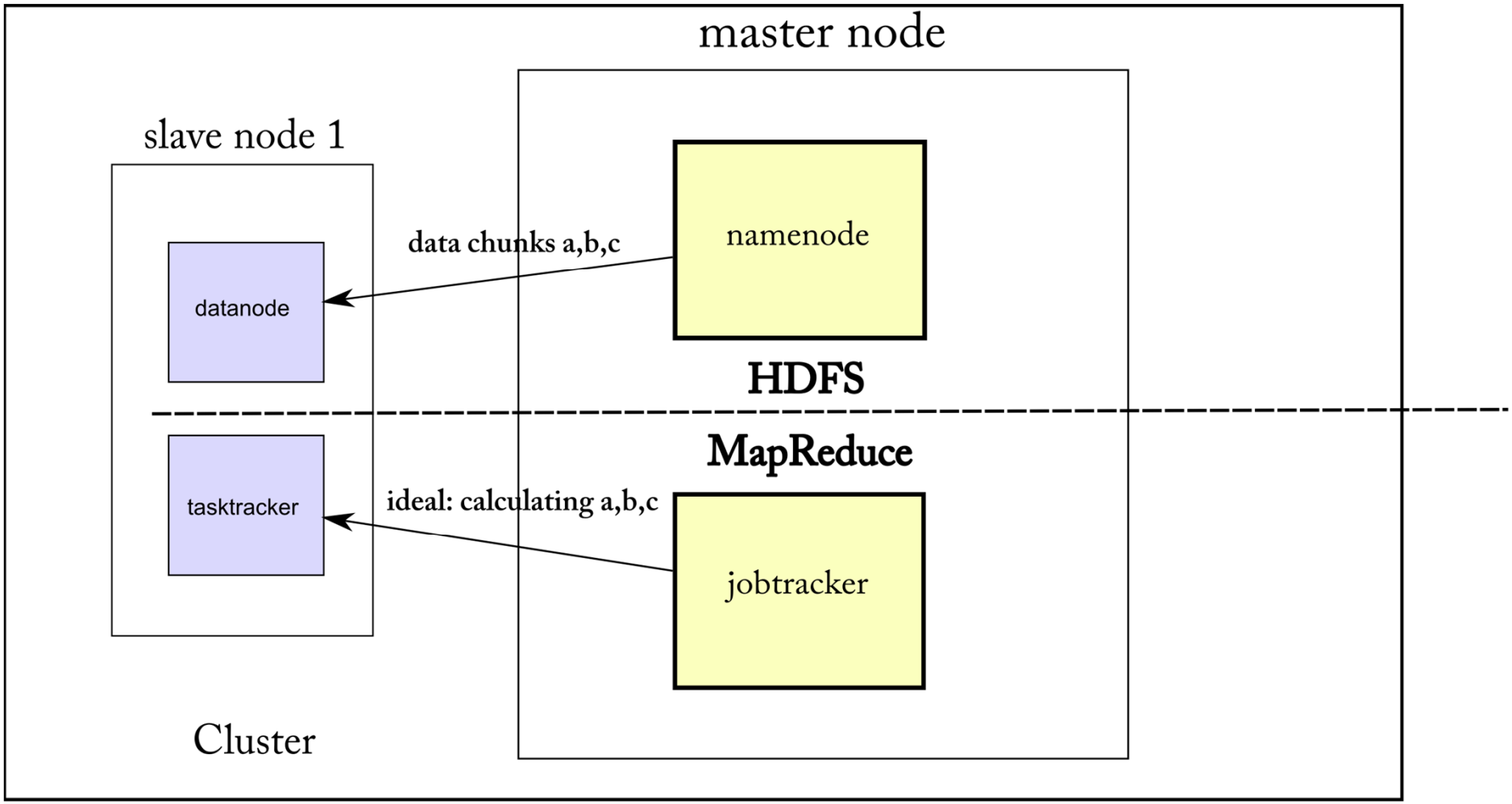
HDFS & MapReduce



HDFS Details

- HDFS
  - Written in Java, needs ssh
  - inspired by GFS
  - Runs on top of the filesystem of the underlying operating system
  - Blocks data into 64 megabyte by default
  - Replication (3 by default)
  - Namenode („master node“) single point of failure
  - Uses Data locality

HDFS Details



Schema for Index Construction in MapReduce

- **Schema of map and reduce functions**
- map: input → list(k, v) reduce: (k,list(v)) → output
- **Instantiation of the schema for index construction**
- map: collection → list(termID, docID)
- reduce: (<termID1, list(docID)>, <termID2, list(docID)>, …) → (postings list1, postings list2, …)

Example for Index Construction

- Map:
- d1 : C came, C c'ed.
- d2 : C died. →
- <C,d1>, <came,d1>, <C,d1>, <c'ed, d1>, <C, d2>, <died,d2>
- Reduce:
- (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c'ed,(d1)>) → (<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c'ed,(d1:1)>)

Dynamic Indexing

- Up to now, we have assumed that collections are static.
- They rarely are:
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- This means that the dictionary and postings lists have to be modified:
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

Simplest Approach

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index

Issues with Main and Auxiliary Indexes

- Problem of frequent merges – you touch stuff a lot
- Poor performance during merge
- Actually:
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for OS.
- Assumption for the rest of the lecture: The index is one big file.
- In reality: Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

Logarithmic Merge

- Maintain a series of indexes, each twice as large as the previous one
  - At any time, some of these powers of 2 are instantiated
- Keep smallest (Z_0) in memory
- Larger ones (I_0, I_1, …) on disk
- If Z_0 gets too big ($> n$), write to disk as I_0
- or merge with I_0 (if I_0 already exists) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1)
- Or merge with I_1 to form Z_2

```
LMERGEADDTOKEN(indexes, Z_0, token)
 1  Z_0 ← MERGE(Z_0, {token})
 2  if |Z_0| = n
 3     then for i ← 0 to ∞
 4          do if I_i ∈ indexes
 5               then Z_{i+1} ← MERGE(I_i, Z_i)
 6                    (Z_{i+1} is a temporary index on disk.)
 7                    indexes ← indexes − {I_i}
 8               else I_i ← Z_i     (Z_i becomes the permanent index I_i.)
 9                    indexes ← indexes ∪ {I_i}
10                    BREAK
11          Z_0 ← ∅
```

```
LOGARITHMICMERGE()
1  Z_0 ← ∅     (Z_0 is the in-memory index.)
2  indexes ← ∅
3  while true
4  do LMERGEADDTOKEN(indexes, Z_0, GETNEXTTOKEN())
```

Logarithmic Merge

- Auxiliary and main index: index construction time is $O(T^2)$ as each posting is touched in each merge.
- Logarithmic merge: Each posting is merged O(log T) times, so complexity is O(T log T)
- So logarithmic merge is much more efficient for index construction
- But query processing now requires the merging of O(log T) indexes
  - Whereas it is O(1) if you just have a main and auxiliary index

Further Issues with Multiple Indexes

- Collection-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
- Will see more such statistics used in results ranking

Dynamic Indexing at Search Engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Sarah Palin, …
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is deleted

Index Compression

Reuters-RCV1 Text Collection

- Messages gathered from Reuters newswire
- 800,000 documents
- 200 tokens per document on average
- 6 bytes per token on average
- 400,000 terms
- 100,000,000 tokens in total

Reuters-RCV1 Text Collection

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enough

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2008 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

Figure taken from Manning, Raghavan, Schütze: Introduction to Information Retrieval, Cambridge University Press, 2008.

Reuters-RCV1 Text Collection

	(distinct) terms			nonpositional postings			tokens (number of position entries in postings)		
	number	delta %	total %	number	delta %	total %	number	delta %	total %
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2	-2	100,680,242	-8	-8	179,158,204	-9	-9
case folding	391,523	-17	-19	96,969,056	-3	-12	179,158,204	0	-9
30 stop words	391,493	0	-19	83,390,443	-14	-24	121,857,825	-31	-38
150 stop words	391,373	0	-19	67,001,847	-30	-39	94,516,599	-47	-52
stemming	322,383	-17	-33	63,812,300	-4	-42	94,516,599	0	-52

30 most common terms account for 30% of all tokens in written text

Important Figures

- Size of term vocabulary (how many distinct terms within index)
- Popularity of terms

- Estimation of index size
- Compression algorithms

Size of Vocabulary

- Grows with size of collection

- Heap‘s law

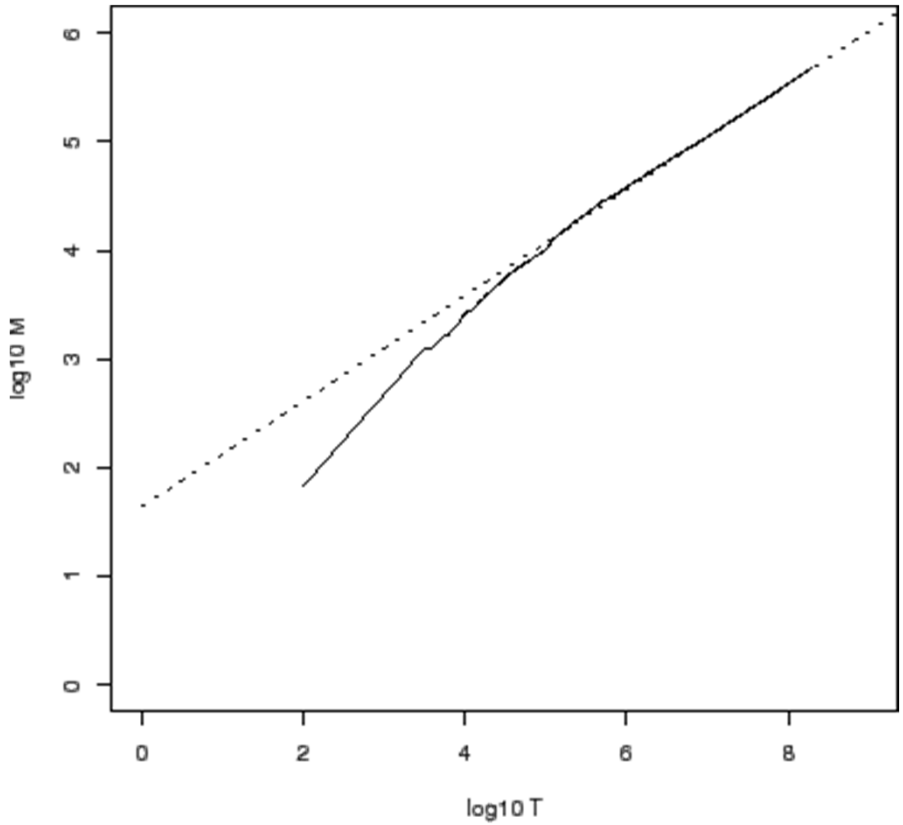
  describes size of vocabulary as a function of the size of the collection

$$M = kT^b$$

  M = size of vocabulary
  T = number of tokens in collection
  k normally between 30 and 100
  b normally approx. 0.5

- Allows estimating size of vocabulary for given T

Heap‘s Law



Heap‘s law for Reuter‘s collection (log/log scale)
with k = 44, b = 0.49

Figure taken from Manning, Raghavan, Schütze: Introduction to Information Retrieval, Cambridge University Press, 2008.

Zipf's Law

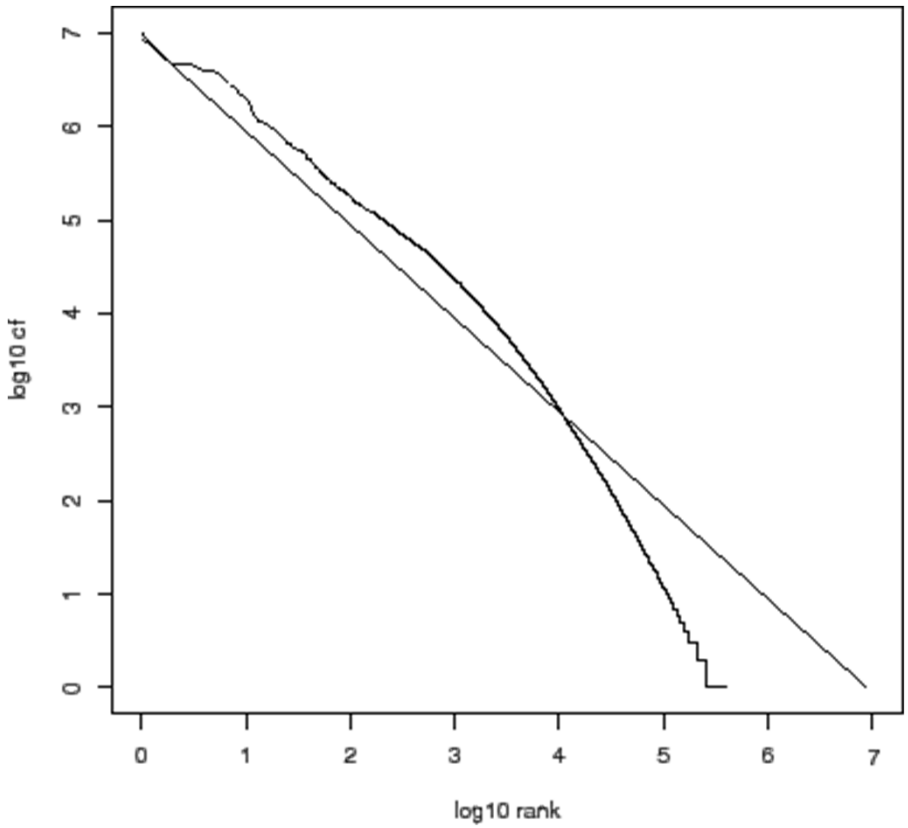
- Estimates frequency distribution of terms
- Many infrequent terms, few terms with a high frequency
- cf_i = collection frequency: number of times term t appears in the collection

- Zipf's Law:
  i^{th} most frequent term has frequency cf_i (collection frequency) which is proportional to 1/i

$$cf_i \propto \frac{1}{i}$$

- Most popular term (in English 'the') appears x times, then 2nd most popular term appears x/2 times within collection, 3rd most popular term appears x/3 times…

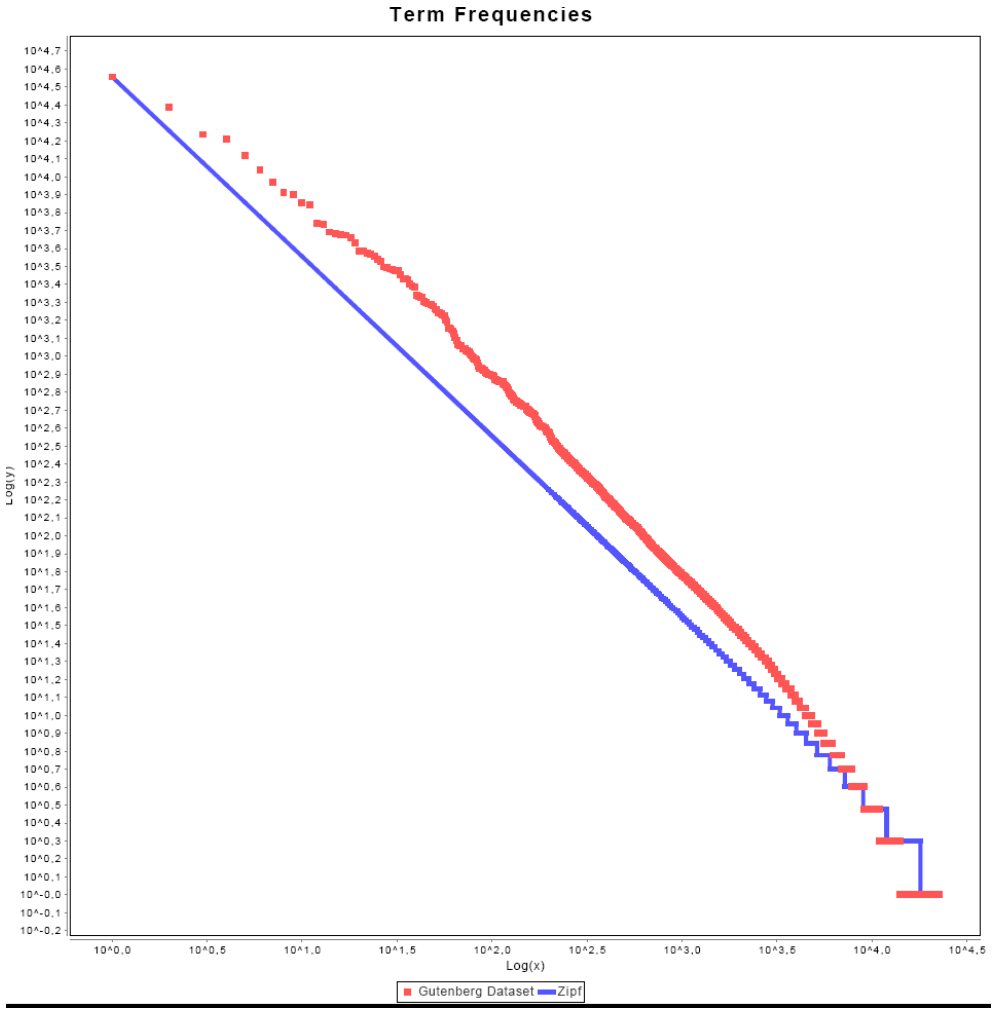
Zipf‘s Law



Zipf‘s law for Reuter‘s collection (log/log scale)

Figure taken from Manning, Raghavan, Schütze: Introduction to Information Retrieval, Cambridge University Press, 2008.

Zipf‘s Law



- 9 Gutenberg books
- 630,000 terms
- 22,400 distinct terms
- most popular term: 'the'

Index Compression

- Full index (including word position, pointer overhead, etc.) sometimes almost as big as data itself

- Compression
  - Less space
  - Fit bigger index in memory (move it up the memory hierarchy)
  - Data is stored closer together -> less seek time
  - Easier to transfer (distributed systems)

- Side note: Compression requires decompression
  Requirements for decompression?

Index Compression

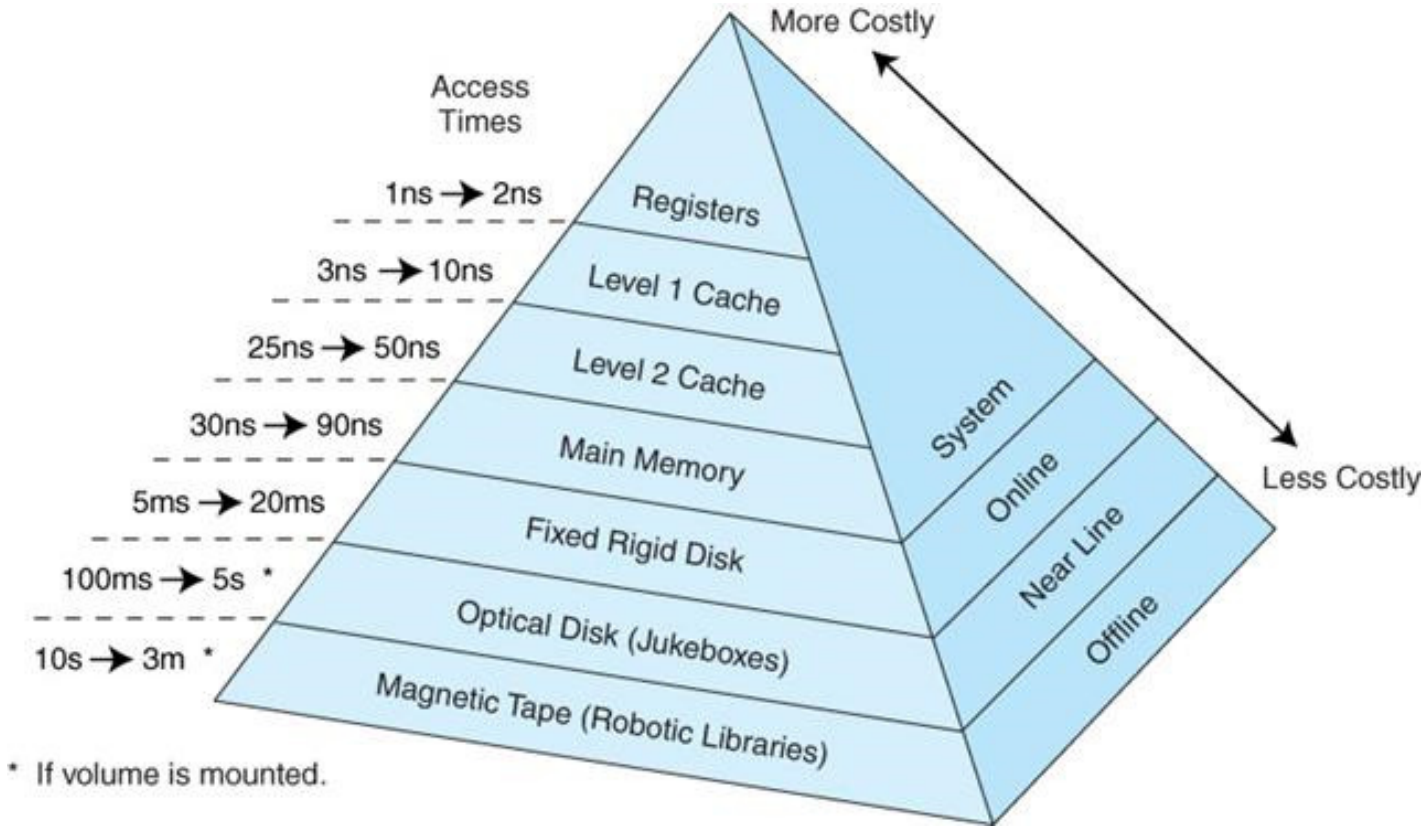


Figure taken from http://tjliu.myweb.hinet.net/COA_CH_6.htm

Index Compression

- Processor can process *p* elements (inverted lists) per second
- Memory can supply processor with *m* elements per second
- Number of elements processed per second *min(m, p)*
- What if *m > p?*
- What if *p > m?*

- Decompression changes these numbers

- Compression ratio mr: r elements can be stored in the same amount of space as 1 uncompressed element
- Decompression factor dp: number of elements processed by the processor per second

Index Compression

- (mr, dp)
  - Normally mr > 1, dp < 1
  - Complicated, space-efficient compression: mr >> 1, dp << 1
  - Optimal compression: mr = dp

- Compression can be lossless or lossy (e.g. even stemming, stopwords, etc. is lossy)
- We only consider lossless compression methods
- Lossy compression only applicable if lost information is not used for search

Index Compression

- Dictionary compression
  - Smaller part of index in terms of storage capacity
- Compression of list of documents containing dictionary entry (postings list)
  - Major part of index in terms of storage capacity

Dictionary Compression

- First attempt of storing index, fixed width

term	frequency	pointer
the	397503	→
a	208484	→
Neurophysiology	2	→

20 bytes 8 bytes 4 bytes

- Reuters: M * (20 + 8 + 4) = 400,000 x 28 = 11.2 MB

Dictionary Compression

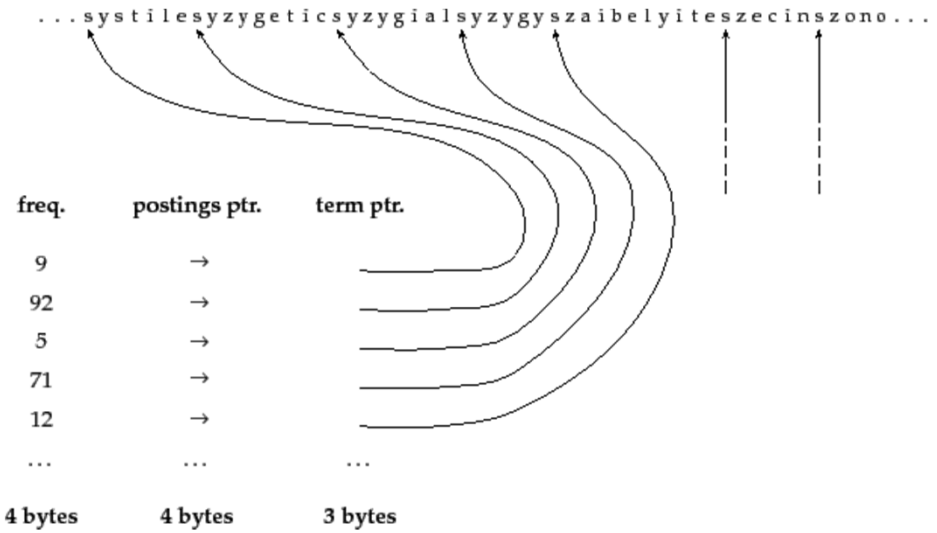
- Allocate 20 bytes per word
- Avg. English word: 8 characters

- What about |word| << 20?
- What about |word| > 20?

- More flexible format required (cf. VARCHAR vs CHAR in databases)

Dictionary Compression

- Storing index as long string



- 400,000 terms * 8 = 3,200,000 position pointers -> 3 bytes
- 4 bytes for frequency and postings pointer each
- 8 bytes for term itself in string (on avg)
- M * (4 + 4 + 3 + 8) = 7.6 MB

Figure taken from Manning, Raghavan, Schütze: Introduction to Information Retrieval, Cambridge University Press, 2008.

Dictionary Compression

- Improvement 1: Blocked Storage
  - Group terms into blocks of k terms
  - Only use pointer to first term in block
  - Add information about term length for faster hops
  - Save storage for k-1 pointers (k=4 → (k-1 * 3 bytes saved)
  - Add 4 bytes for term length
  - k = 4 → save 5 bytes per block
  - 7.1 MB for Reuters
  - Tradeoff: save storage space by increasing k at the cost of fast lookup

Dictionary Compression

- Search within blocked storage compressed data
- Binary search uncompressed (a) vs compressed (b)

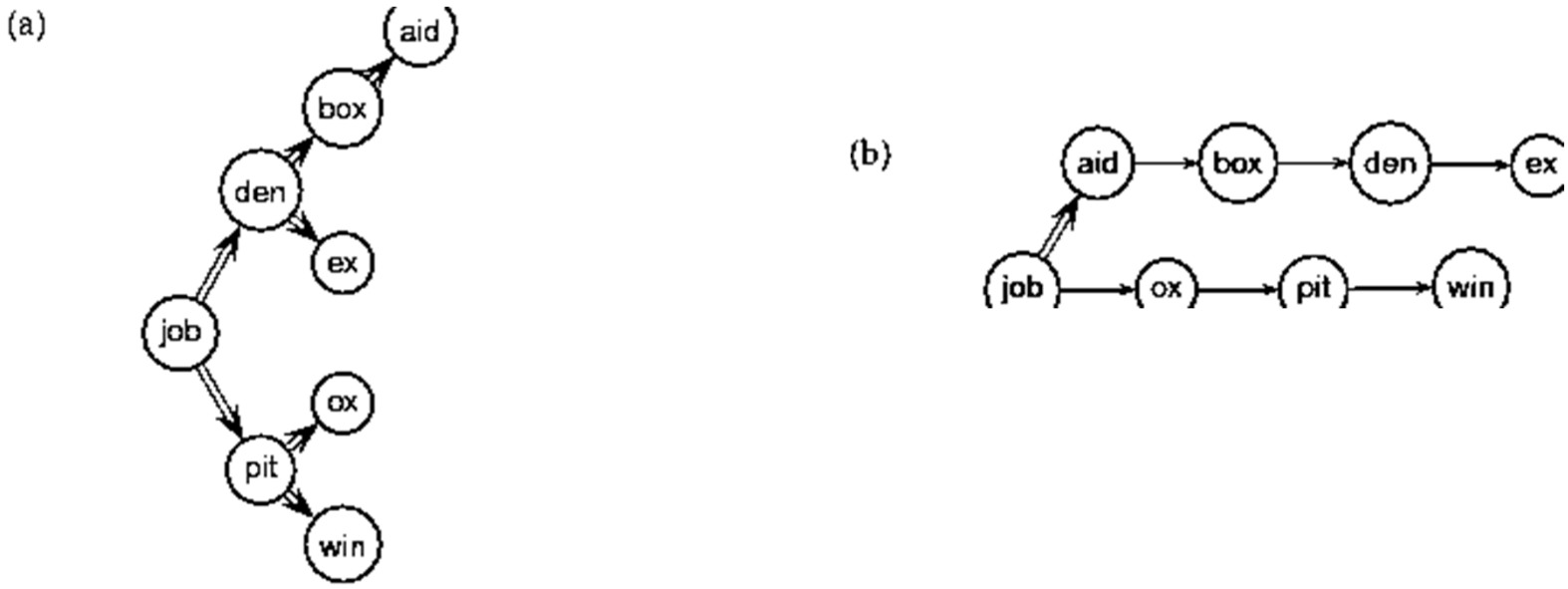


Figure taken from Manning, Raghavan, Schütze: Introduction to Information Retrieval, Cambridge University Press, 2008.

Dictionary Compression

- Improvement 2: Front encoding
  - Exploit common prefixes of strings
  - For blocked storage
  - Use special character for the prefix and delimiter for prefix
  - First byte still represents length of term
  - * marks end of prefix
  - 5.9 MB for Reuters

One block in blocked compression ($k = 4$) ...
8automata**8**automate**9**automatic**10**automation

⇓

...further compressed with front coding.
8automat*a**1**◇e**2** ◇ ic**3**◇ion

Figure taken from Manning, Raghavan, Schütze: Introduction to Information Retrieval, Cambridge University Press, 2008.

Postings Lists Compression

- Text size = 800,000 * 200 * 6 = 960 MB

- 800,000 documents
  - log2 800,000 = 20 bits
  - 3 bytes per document identifier

- Postings list size of 100,000,000,000
- 100,000,000 * 3 = 300 MB (250 MB for 20 bits)

Delta Encoding

- Posting lists → store gaps between document identifiers

	encod	**postings list**								
the	docID	….		283042		283043		283044		283045
	gaps				1		1		1	
computer	docID	…		283047		283154		283159		283202
	gaps				107		5		43	
arachnocentric	docID	25200		500100						
	gaps	25200	248100							

Variable Byte Encoding

- Store offsets of document ids within posting lists to decrease 20 bits for encoding
- Works for frequent terms (size << 20 bits)
- However, not for infrequent terms (20 bits)
- Distribution: Zipfs law
  → Variable encoding required

- First bit used as continuation bit (1 if the byte is the last byte of the document id, 0 else)

Variable Byte Encoding

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Gamma Encoding

- Variable byte encoding on byte level
- Gamma encoding works on bit level
- Simple bit encoding: unary code
  - n represented by n 1s
  - 0 as delimiter
  - 4 = 11110

Gamma Encoding

- Gamma encoding: split representation of a gap into
  - length (unary, of offset)
  - offset (binary, leading 1 removed)
- Encoding: concatenation of length and offset
- e.g. 9 = 111111111 unary = 1001 binary = 1110001 encoded

  offset: 001 (binary, 1001 with leading 1 removed)

  length: 1110 (unary, 111)

Gamma Encoding

number	unary code	offset (binary, leading 1 removed)	length of offset (unary, leading 1 removed)	code
0	0			
1	10		0	0
2	110	0	10	10,0
3	1110	1	10	10,1
4	11110	00	110	110,00
9	1111111110	001	1110	1110,001
13		101	1110	1110,101
24		1000	11110	11110,1000
511		11111111	111111110	111111110,11111111
1025		0000000001	11111111110	11111111110,0000000001

Gamma Encoding

- Decoding Gamma-code:
  - read unary code until 0 (delimiter) → length
  - reconstruct offset by adding leading 1

  - Consider 1110101
  - length: 111 = 3
  - offset: (1)101 = 13

- Reuters collection: 101 MB

Conclusion

Data Structure	MB
Dictionary fixed width	11.2
Dictionary as string	7.6
Dictionary with blocks	7.1
Dictionary with blocks, front encoding	5.9
Text collection	960
Postings lists	250
Postings lists, variable byte encoded	116
Postings lists, Gamma encoded	101

Credits

- Slides partly adapted from
  - Eva Zangerle , DBIS Innsbruck (2014/15)
  - Stefano Ceri, Alessandro Bozzon , Marco Brambilla,Emanuele Della Valle, Piero Fraternali, Silvia Quarteroni: Web Information Retrieval
  - Dietmar Jannach, Markus Zanker, Alexander Felfernig, Gerhard Friedrich: Recommender Systems – An Introduction
  - Günther Specht, DBIS Innsbruck (former lectures)