

# Information Retrieval and Data Mining

Summer Semester 2015  
TU Kaiserslautern

Prof. Dr.-Ing. Sebastian Michel  
Databases and Information Systems  
Group (AG DBIS)

<http://dbis.informatik.uni-kl.de/>

# Announcements

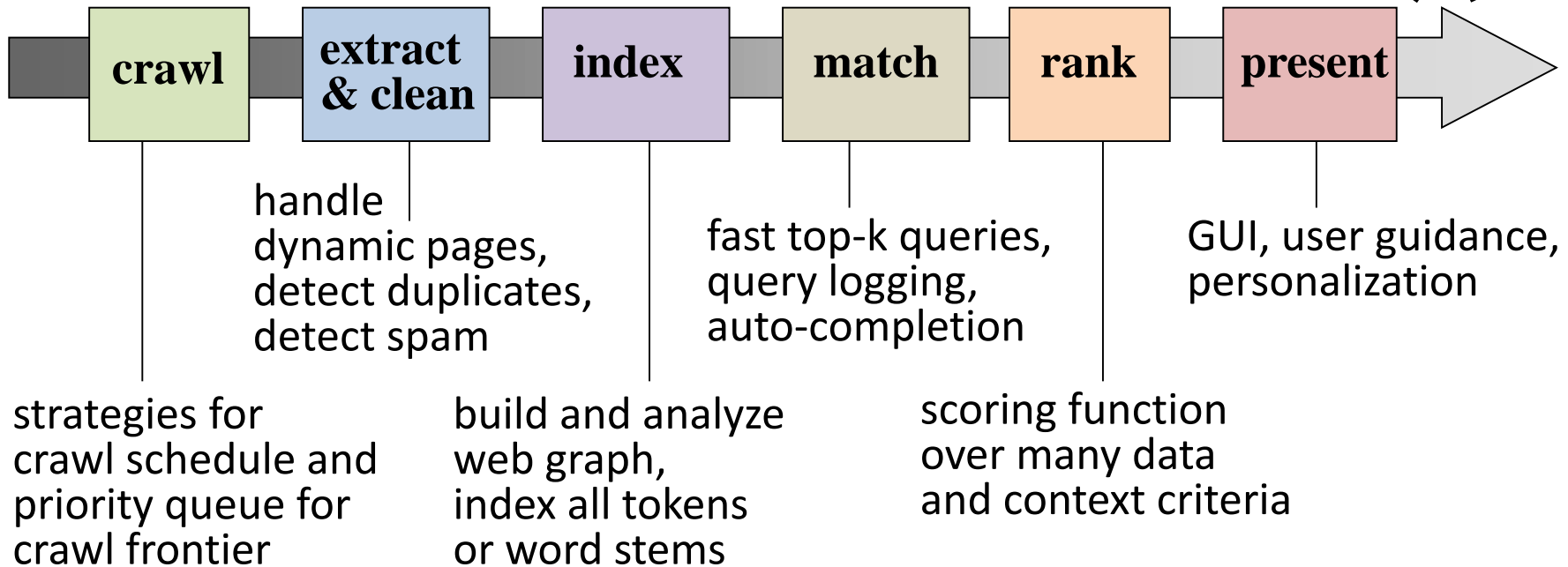
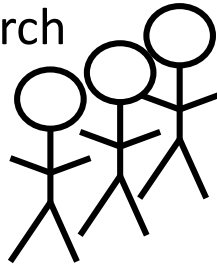
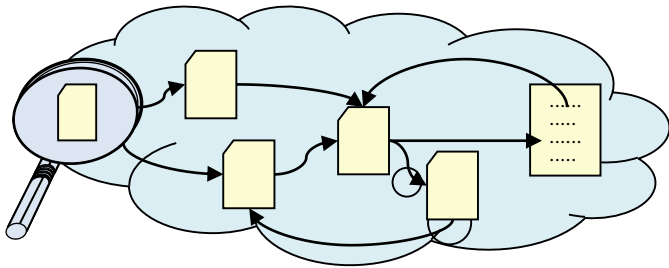
- **Doodle link for exam registration posted**
- Just place your name ONCE (the DDM lecture has its own Doodle). Write down date/time before clicking the save button.
- **First-come, first-served.**
- Read instructions on website.
- **Exam dates are on 22., 23., 24., 27., 28. July**
- **Re-exams in late September / early October.**
- **More exam dates possible in beginning of 2016.**

# Announcements (2)

- **Additional exercise sheet** for upcoming content. Not mandatory, no bonus points.
- **Question & Answer Session** roughly 1 week before exams start; in the exercise slot and room.
- **The department's lecture evaluation has started. Please participate and give feedback!**
- You can also win something. More info under <https://vlu.cs.uni-kl.de/>

# Information Retrieval in a Nutshell

- Web, intranet, digital libraries, desktop search
- Unstructured/semi-structured data



**Server farms** with **10 000's** (2002) – **100,000's** (2010) computers, distributed/replicated data in high-performance file system (**GFS, HDFS,...**), massive parallelism for query processing (**MapReduce, Hadoop,...**)

# Chapter III: Indexing & Searching

- III.1 **Indexing**

Dictionary, Inverted Index, Forward Index, Partitioning, Caching

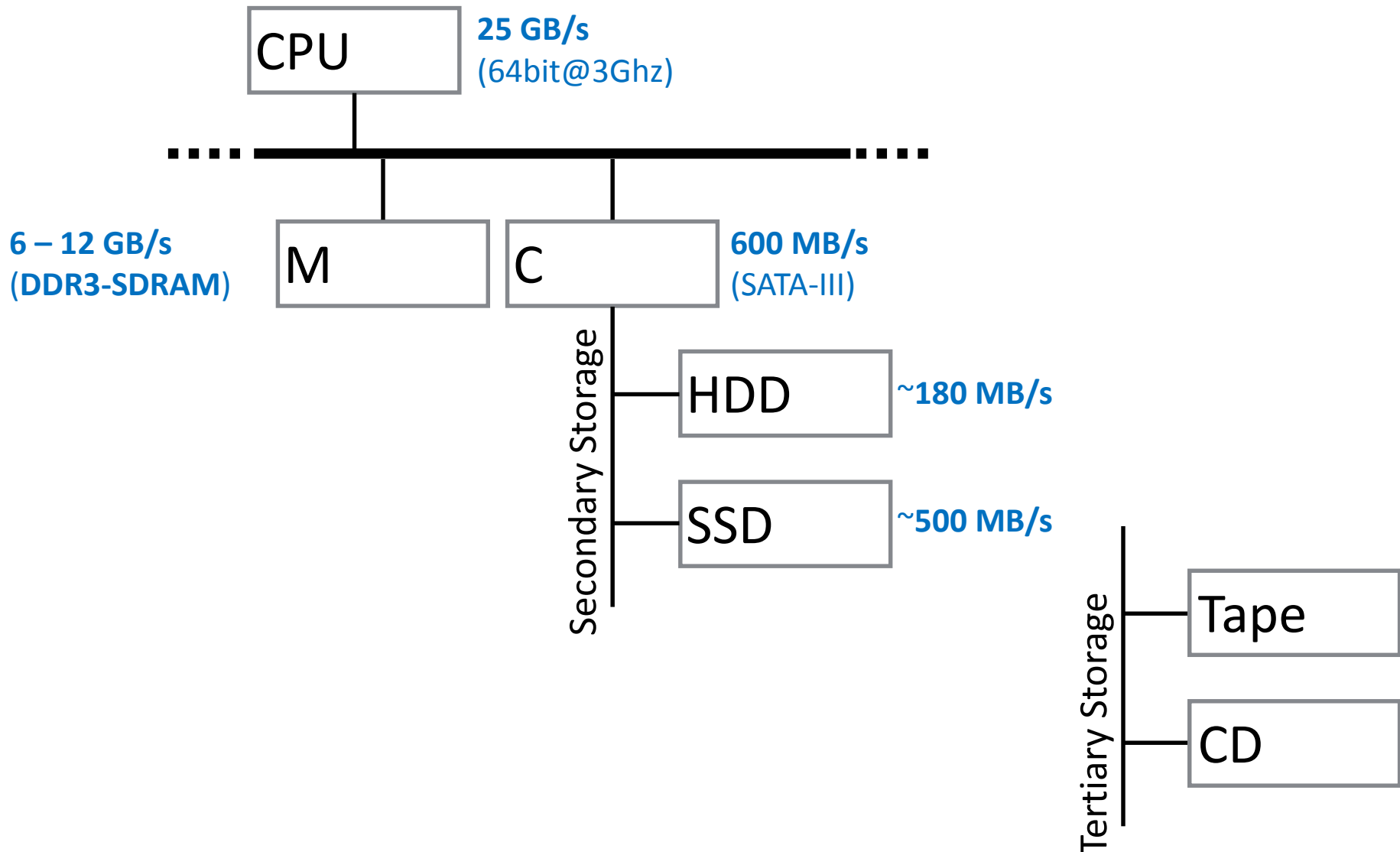
- III.2 **Compression**

Huffman Coding, Ziv-Lempel, Variable-Byte Encoding, Gap Encoding, Gamma Encoding

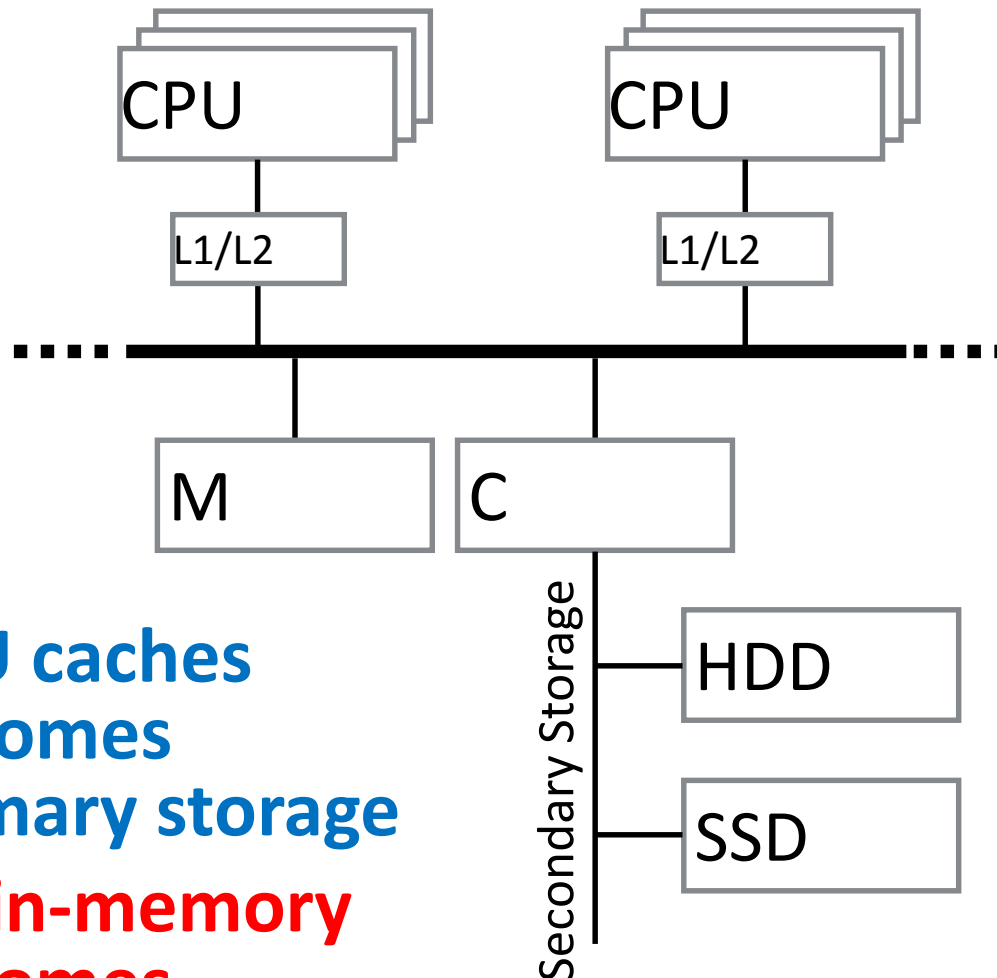
- III.3 **Query Processing**

Term-at-a-Time, Document-at-a-Time, Quit & Continue, WAND, Fagin's TA

# Traditional View on Hardware



# More Modern View on Hardware



- **CPU caches becomes primary storage**
- **Main-memory becomes secondary storage**

- CPU-to-L1:  
~3-5 cycles
- CPU-to-L2:  
~15-20 cycles
- CPU-to-M:  
~200 cycles

# Cost Numbers (=>Complex Cost Model)

- L1 cache reference 0.5 ns
- L2 cache reference 7 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10,000 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- **Disk seek 10,000,000 ns**
- Read 1 MB sequentially from network 10,000,000 ns
- **Read 1 MB sequentially from disk 30,000,000 ns**
- Send packet CA->Netherlands->CA 150,000,000 ns

$$1\text{ns} = 10^{-6}\text{ ms}$$

Numbers source: Jeff Dean



# Random Access vs. Sequential Access

- **Locality matters** across all levels of the memory hierarchy
- **Typical latencies** of performing a random access:
  - Main memory:  $10^{-8}$  s ( $\sim 95$  MB/s assuming one byte is read)
  - Solid state drive:  $10^{-5}$  s ( $\sim 0.9$  MB/s assuming one byte is read)
  - Hard disk drive:  $10^{-2}$  s ( $\sim 0.09$  KB/s assuming one byte is read)
- **High transfer rates only achievable by sequential accesses**, i.e., by reading data that is stored contiguously, e.g., on disk



©brutalSoCal@flickr

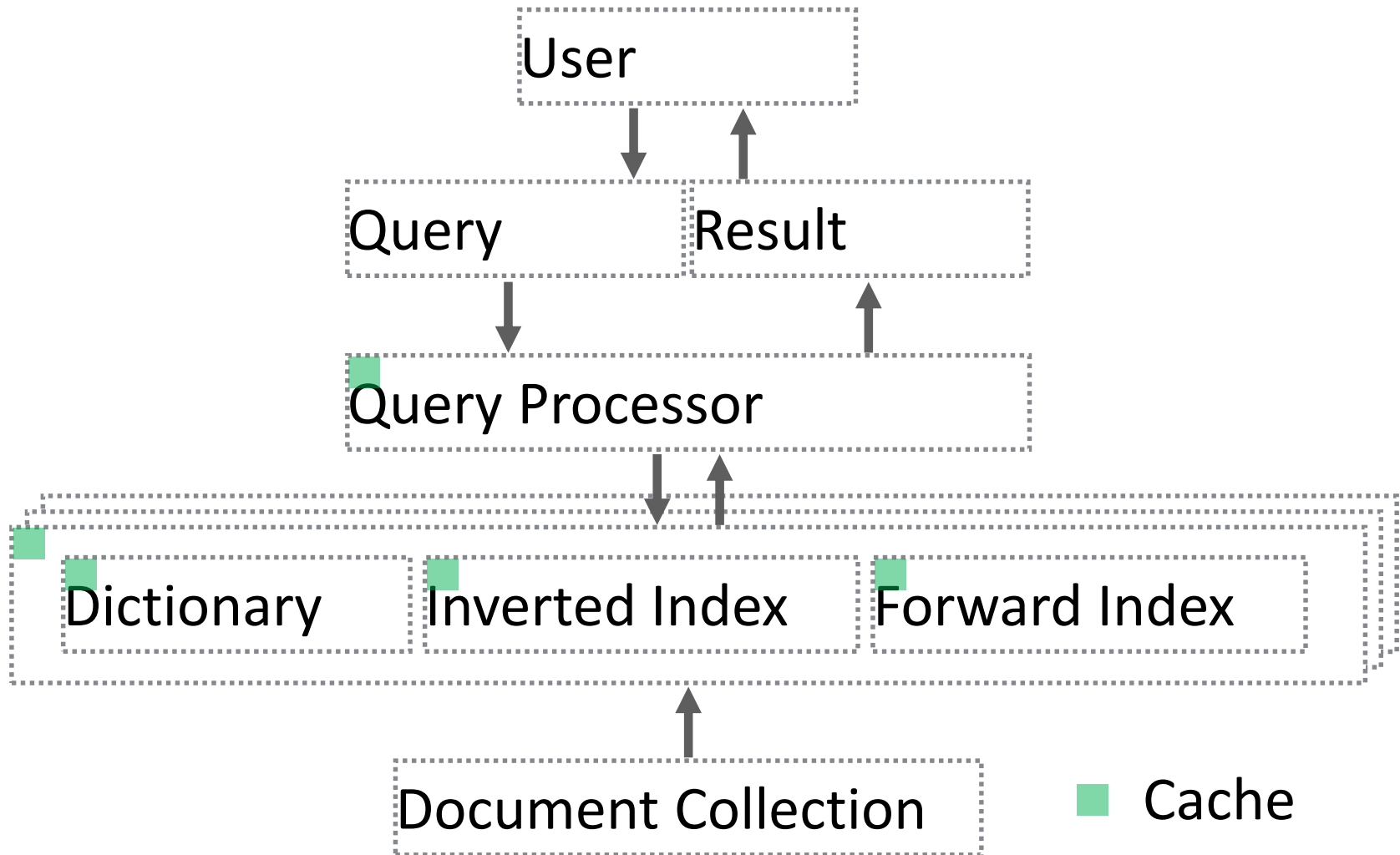


© Andreas@flickr

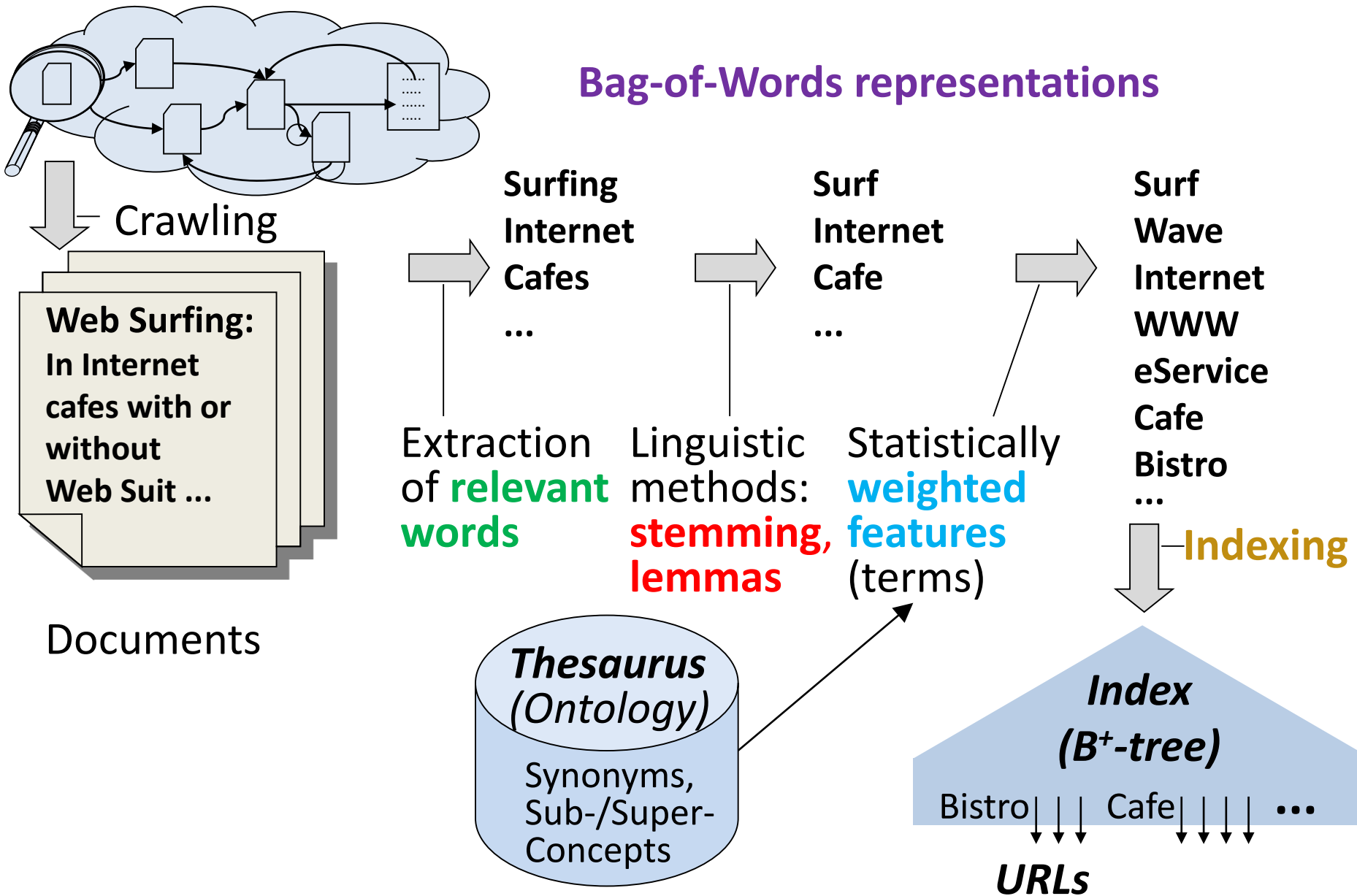


© Uncle Saitful@flickr

# Overview of Modern IR System



# Content Gathering and Indexing



# III.1 Indexing

1. **Dictionary**
2. **Inverted Index**
3. **Forward Index**
4. **Partitioning**
5. **Caching**

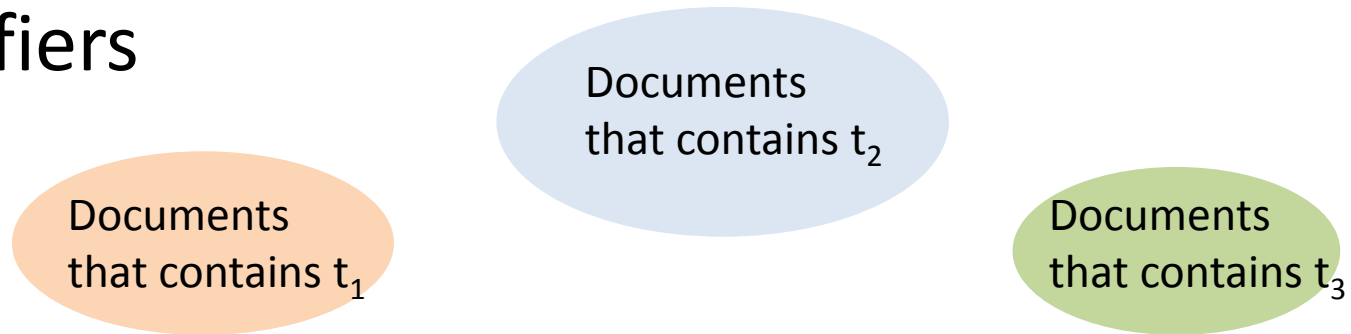
Based on MRS Chapters 2, 3, 4 and RBY Chapter 9

# Problem Statement

- Have seen various ranking models
- Assume we have now “crawled” lots of documents from the Web
- **How can we organize documents and associated information like  $tf*idf$  scores?**
- **How can we efficiently process user queries?**

# Simple (High-Level) Solution

- For each term we store the set of document identifiers



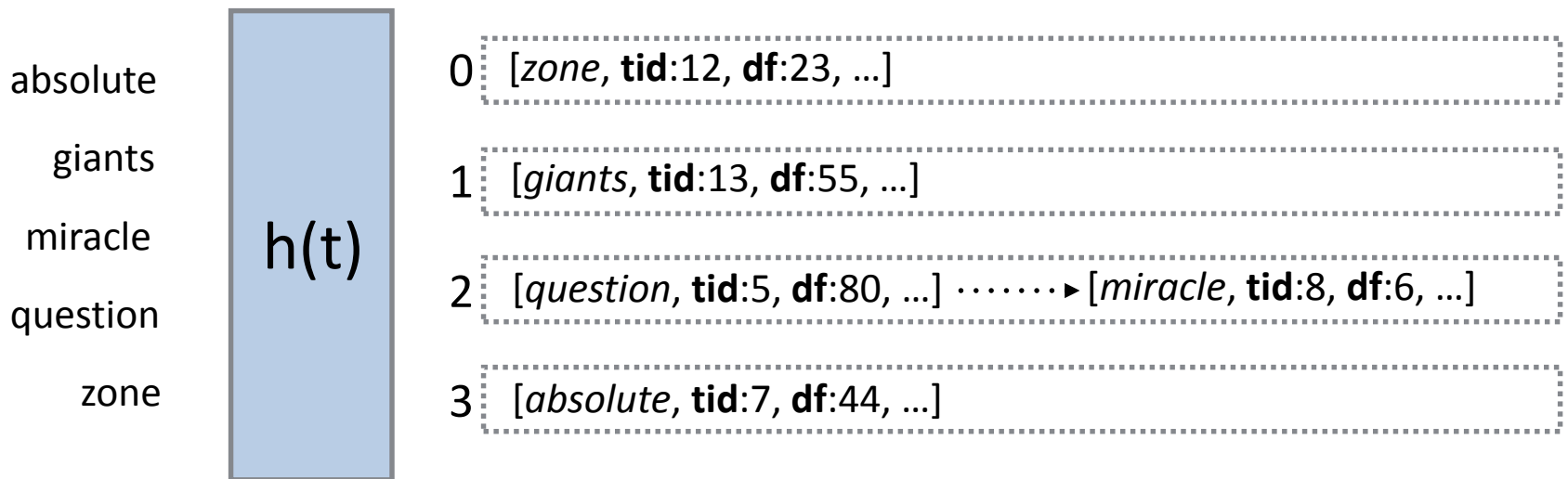
- Query  $q = t_1$  *and*  $t_2$  *and*  $t_3$  is posted
  - How do we find the sets we have to look at?
  - How do we compute the set of documents that answer?
  - Can we also rank documents?
  - Where are the “tf\*idf” scores stored?

# 1. Dictionary

- **Dictionary maintains information about terms**, e.g.:
  - unique term identifier (e.g., house → 3,141)
  - location of corresponding posting list on disk or in memory
  - statistics such as document frequency and collection frequency
- **Operations supported by the dictionary**
  - lookups by term
  - range searches (e.g., for prefix and suffix queries like hous\* and \*ing)
  - substring matching (e.g., for wildcard queries like ho\*e\*lly)
  - lookups by term identifier

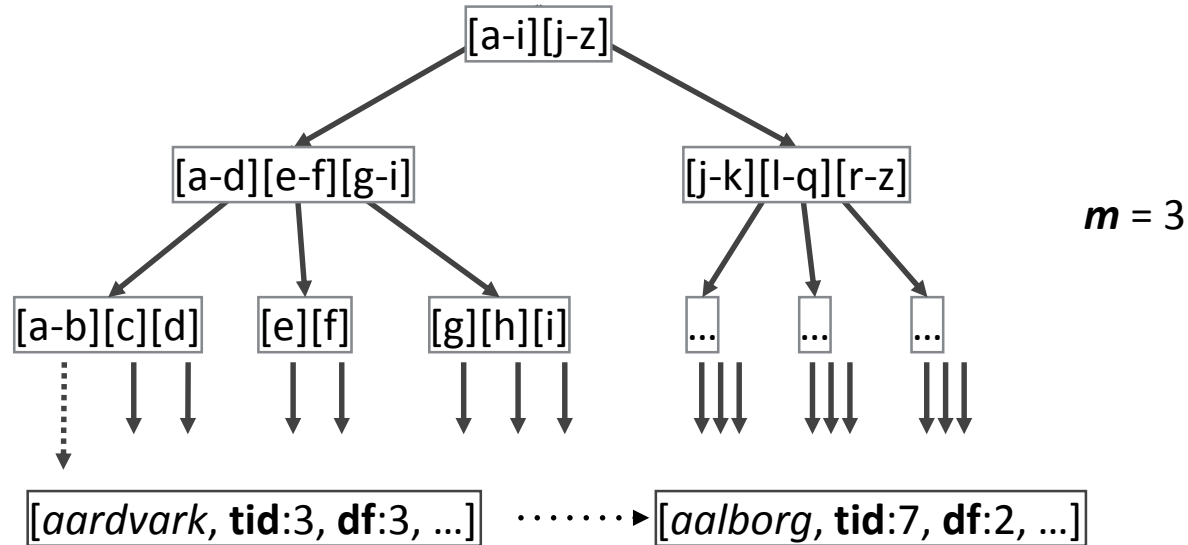
# Hash-Based Dictionary

- Supports lookups in  $O(1)$  but no other operations
- Vocabulary dynamics (i.e., new or removed terms) problematic
- Works best in main memory





# B+-Tree-Based Dictionary



- **Balanced** tree with **internal nodes having fan-out  $m$**
- 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**

# Extended Lookups

- Given the term

**absolute**

- How can we achieve searching for wildcard queries like

**abs\*** or **abs\*lute** ?

- What do we have to index in the dictionary?

# Permuterm Index

- **Indexes all permutations of each term** with delimiter symbol \$

absolute .....►  
absolute\$  
bsolute\$a  
solute\$ab  
olute\$abs  
lute\$abso  
ute\$absol  
te\$absolu  
e\$absolut

- Supports arbitrary wildcard queries (e.g., ho\*e\*lly is mapped to **prefix query** lly\$ho\* with **post-filtering of matching terms**)
- Works on-top of dictionary supporting range searches
- Space blowup proportional to average term length

# k-Gram Index

- **Indexes all k-grams** for each term with delimiter symbol \$

absolute .....►

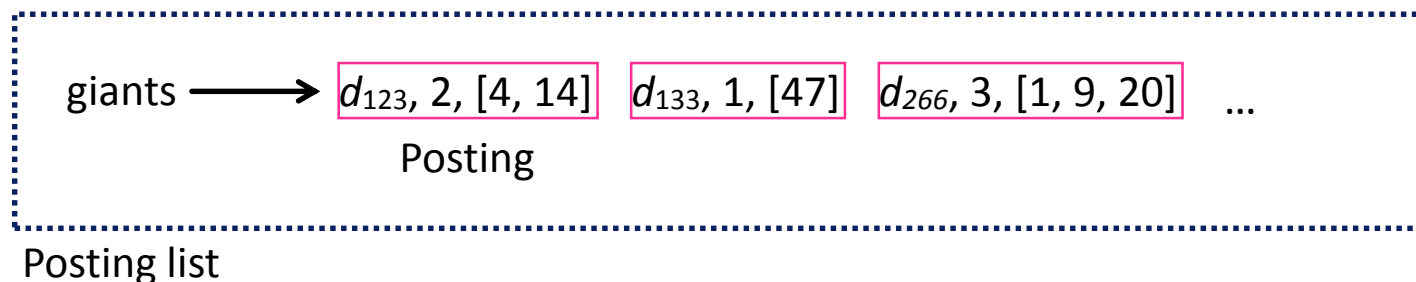
\$ab  
abs  
bso  
sol  
olu  
lut  
ute  
te\$

**k = 3**

- **Supports arbitrary wildcard queries** (e.g., ho\*e\*lly is mapped to lookups \$ho, lly, ly\$ with **intersection and post-filtering of terms**)
- Works on-top of dictionary supporting lookups
- Space blowup proportional to parameter k

# 2. Inverted Index

- **Inverted index keeps a posting list for each term**, which usually reside on secondary storage, with **each posting capturing information about term's occurrences in a specific document**
  - **document identifier** (e.g.,  $d_{123}$ ,  $d_{234}$ , ...)
  - **term frequency** (e.g.,  $tf(\text{house}, d_{123}) = 2$ ,  $tf(\text{house}, d_{234}) = 4$ )
  - **score impacts** (e.g.,  $tf(\text{house}, d_{123}) * idf(\text{house}) = 3.75$ )
  - **offsets** (i.e., absolute positions at which the term occurs in the document)



- Posting lists are **usually compressed** for time and space efficiency

# Posting Payloads

- Posting payloads depend on the kind of queries and the retrieval models to be supported

- **document identifier** (always required, sufficient for Boolean retrieval)

$d_{123}$

- **term frequency** (for ranked retrieval, possibly different retrieval models)

$d_{123}, 2$

- **score impacts** (if the retrieval model has been fixed)

$d_{123}, 3.75$

- **offsets** (for proximity constraints or phrase queries)

$d_{123}, 2, [4, 14]$

# Posting-List Order

- Posting-list order depends on the kinds of queries to be supported
- **Document-ordered posting lists** for more efficient intersections  
(e.g., required for Boolean queries and phrase queries)

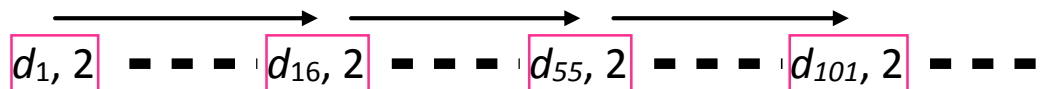
$d_{123}, 2, [4, 14]$   $d_{133}, 1, [47]$   $d_{266}, 3, [1, 9, 20]$  . . . . .

- **Impact-ordered posting lists** for more efficient top-k queries  
(i.e., terminate query processing as soon as top-k results known)

$d_{231}, 1.0$   $d_{12}, 0.9$   $d_{662}, 0.8$   $d_3, 0.5$  . . . . .

# Skip Pointers

- Posting lists can be equipped with additional structure
- Skip pointers allow “**fast forwarding**” in a posting list
  - common **heuristic**: evenly spaced at  $\sqrt{df(\text{term})}$
  - **can be embedded into postings or kept together in posting-list header**





# Index Construction and Updates

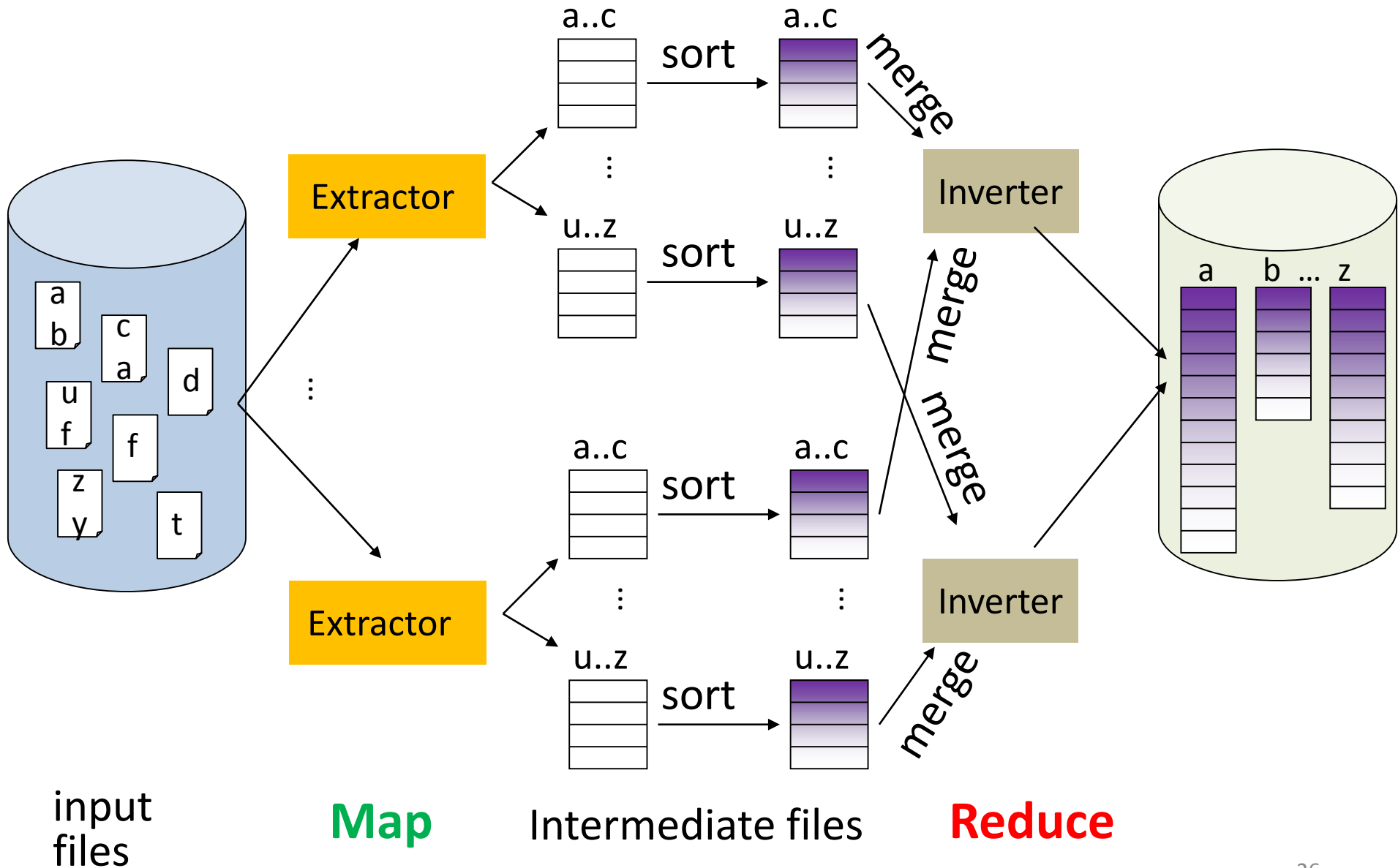
## Index construction:

- **extract** (docId, termId, score) triples from docs
  - can be partitioned & parallelized
  - scores need idf (estimates)
- **sort** entries termId (primary) and docId (secondary)
  - disk-based merge sort (build runs, write to temp, merge runs)
  - can be partitioned & parallelized
- **load** index from sorted file(s), using large batches for disk I/O,
  - compress sorted entries (delta-encoding, etc.)
  - create dictionary entries for fast access during query processing

## Index updating:

- collect large batches of updates in separate file(s)
- periodically sort these files and merge them with index lists

# Map-Reduce for Index Building



# Map-Reduce Parallelism

**Programming paradigm and infrastructure for scalable, highly parallel data analytics.**

- can run on 1000's of computers
- with built-in load balancing & fault-tolerance (automatic scheduling & restart of worker processes)

Easy programming with key-value pairs:

**Map** function:  $K \times V \rightarrow (L \times W)^*$

$(k1, v1) \quad | \rightarrow \quad (l1, w1), (l2, w2), \dots$

**Reduce** function:  $L \times W^* \rightarrow W^*$

$l1, (x1, x2, \dots) \quad | \rightarrow \quad y1, y2, \dots$

**If we want to build an inverted index we have:**

$K = \text{docIds}, V = \text{contents}, L = \text{termIds}, W = \text{docIds}$

# Map-Reduce Example for Inverted Index Construction

**class Mapper**

**procedure** MAP(docId n, doc d)

H ← new Map<term, int>

**For** term t ∈ doc d **do** // local tf aggregation

H(t) ← H(t) + 1

**For** term t ∈ H d **do** // emit reducer job, e.g., using hash of term t

EMIT(term t, new posting <docId n, H(t)>)

**class Reducer**

**procedure** REDUCE(term t, postings [<n<sub>1</sub>,f<sub>1</sub>>, <n<sub>2</sub>,f<sub>2</sub>>, ...])

P ← new List<posting>

**For** posting <n, f> ∈ postings [<n<sub>1</sub>,f<sub>1</sub>>, <n<sub>2</sub>,f<sub>2</sub>>, ...] **do** // global idf aggregation

P.APPEND(<n,f>)

SORT(P) // sort all postings hashed to this reducer by <term, docId || score>

EMIT(term t, postings P) // emit sorted inverted lists for each term

source: Lin & Dyer (Maryland U): Data Intensive Text Processing with MapReduce

# 3. Forward Index

- **Forward index maintains information about documents**
  - compact representation of content (e.g., as sequence of term identifiers)
  - document length

$d_{123}$  the giants played a fantastic season. it is not clear ...

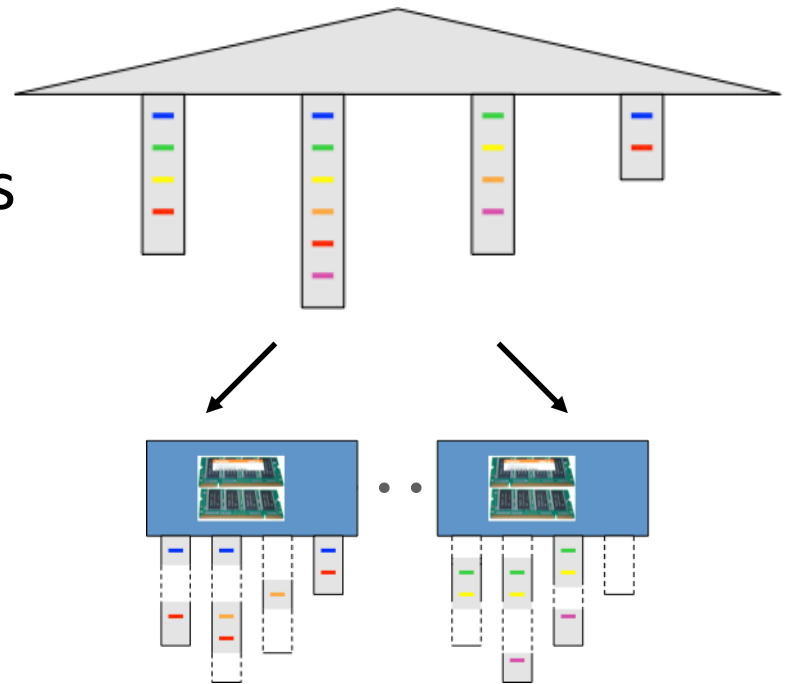


$d_{123}$  **dl:**428 **content:**< 1, 222, 127, 3, 897, 233, 0, 12, 6, 7, 123, ... >

- **Forward index can be used for** tasks, e.g.:
  - **result-snippet generation** (i.e., show context of query terms)
  - **computation of proximity features** for advanced ranking (e.g., width of smallest window that contains all query terms)

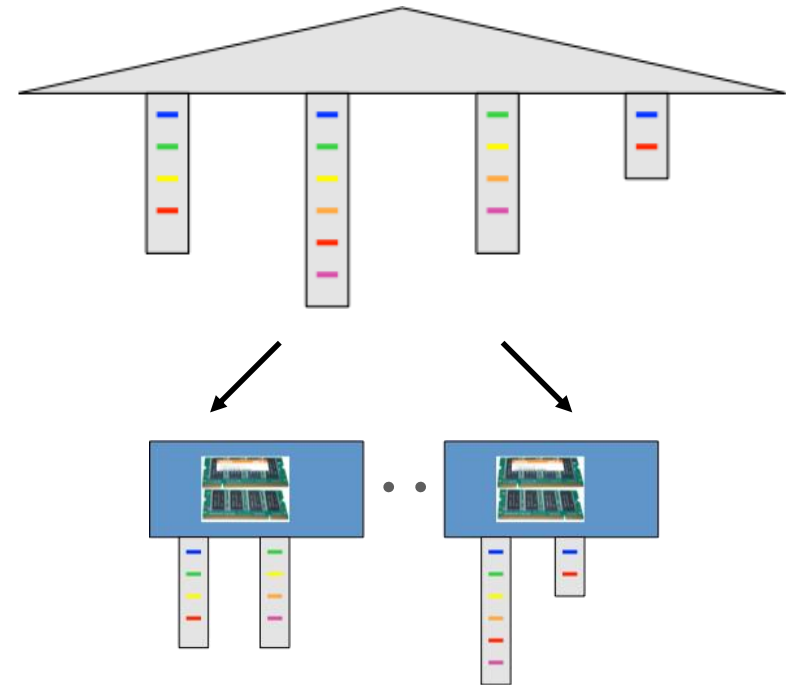
# 4. Partitioning

- Document-partitioned inverted index
  - each compute node indexes a **subset of the document collection**
  - each query is processed by **every compute node**
  - **perfect load balance, embarrassingly scalable, easy maintenance**



# Partitioning (cont'd)

- **Term-partitioned inverted index**
  - each compute **node holds posting lists for a subset of terms**
  - **queries are routed** to compute nodes with relevant terms
  - **lower resource consumption, susceptible to imbalance** (because of skew in the data or query workload), **index maintenance non-trivial**



# Back-of-the-Envelope Cost

## Comparison: Cost Assumptions

- **20 billion web pages**, 100 terms each  $\rightarrow 2 \times 10^{12}$  postings
- **10 million distinct terms**  $\rightarrow 2 \times 10^5$  entries per posting list
- **5 bytes per posting**  $\rightarrow 1$  MB per posting list, 10 TB total
  
- Query throughput: typical 1,000 q/s; peak 10,000 q/s
- Response time: all queries in  $\leq 100$  ms
- Reliability and redundancy: 10-fold redundancy
  
- **Execution cost per query:**
  - 1 ms initial latency + 1 ms per 1,000 postings
  - 2 terms per query
  
- Cost per compute node (4 GB RAM): \$ 1,000
- Cost per disk (1 TB): \$ 500 with 5 ms per RA, 20 MB/s for SAs



# Back-of-the-Envelope Cost Comparison (cont'd)

- **Document-partitioned inverted index in RAM**
- 3,000 compute nodes to hold one copy of the index in RAM
  - 3,000 x 4 GB RAM = 12 TB (10 TB total index size + workspace RAM)
- Query processing:
  - each query executed on 3,000 computers in parallel:  
 $1 \text{ ms} + (2 \times 200 \text{ ms} / 3,000) \approx 1 \text{ ms}$
  - each cluster can sustain  $\sim 1,000$  q/s
- 10 clusters = 30,000 compute nodes to sustain peak load and guarantee reliability & availability
- \$ 30 million = 30,000 x \$ 1,000 (no “big” disks)

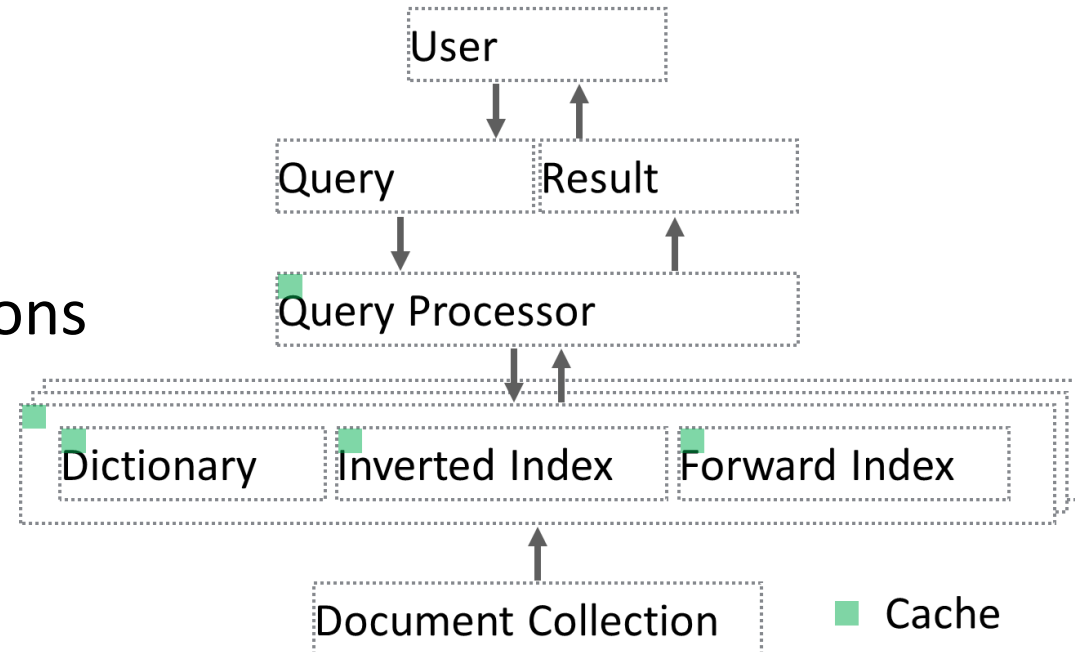
# Back-of-the-Envelope Cost Comparison (cont'd)

- **Term-partitioned inverted index on disk**
- 10 compute nodes each with 1 TB disk to hold entire index
- **Query processing:**
  - $\max(1 \text{ MB} / 20 \text{ MB/s}, 1 \text{ ms} + 200 \text{ ms})$
  - limited throughput: 5 q/s per compute node for 1-term queries
- 1 cluster = 400 nodes to sustain 1,000 q/s for 2-term queries
- 10 clusters = 4,000 nodes to sustain peak load and guarantee reliability & availability
- \$ 6 million =  $4,000 \times (\$ 1,000 + \$ 500)$

# 5. Caching

- **What is cached?**

- Query results
- Posting lists
- Posting-list intersections
- Documents
- Snippets



- **Where is it cached?**

- in RAM of responsible compute node
- in dedicated front-end accelerators or proxy nodes
- in RAM of all (many) compute nodes

# Caching Strategies

- **Least recently used (LRU)**
  - when space is needed, evict the item that was least recently used
- **Least frequently used (LFU)**
  - when space is needed, evict the item that was least frequently used
- **Cost-aware (Landlord algorithm)**
  - estimate for each item: **temperature = access-rate / cost**
  - when space is needed, **evict item with lowest temperature**
  - prefetch item if its predicted temperature is higher than the temperature of the corresponding replacement victims
  - Full details: [Cao and Irani '97][Young '02]

# Caching Effectiveness

- Query frequencies follow Zipf distribution ( $s \approx 1$ )
- [Baeza-Yates et al. '07] analyzed one-year query log of Yahoo!
  - 88% of queries are issued only once
  - account for 44% of overall query volume
  - **query-result caching achieves cache-hit ratios < 50% in practice**

# Summary of III.1

- **Dictionary**  
holds information about terms
- **Inverted Index**  
holds information about word occurrences in documents
- **Forward Index**  
holds compact representations of documents
- **Partitioning**  
distribute inverted index by-document or by-term
- **Caching**  
query results, posting lists, posting-list intersection, etc.

# Additional Literature for III.1

- R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri: The Impact of Caching on Search Engines, SIGIR 2007
- S. Brin and L. Page: The anatomy of a large-scale hypertextual Web search engine, Computer Networks 30:107-117, 1998
- P. Cao and S. Irani: Cost-Aware WWW Proxy Caching Algorithms, USENIX 1997
- R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, O. Ulusoy: A five-level static cache architecture for web search engines, IP&M 48(5):828-840, 2012
- N. E. Young: On-Line File Caching, Algorithmica 33(3):371-383, 2002
- J. Zobel and A. Moffat: Inverted Files for Text Search Engines, ACM Computing Surveys 38(2):6, 2006

# III.2 Compression

1. **Huffman Coding**
2. **Ziv-Lempel Compression**
3. **Variable-Byte Encoding**
4. **Gamma Encoding**
5. **Gap Encoding**
6. **Run-Length Encoding**

Based on MRS Chapter 5 and RBY Chapter 9



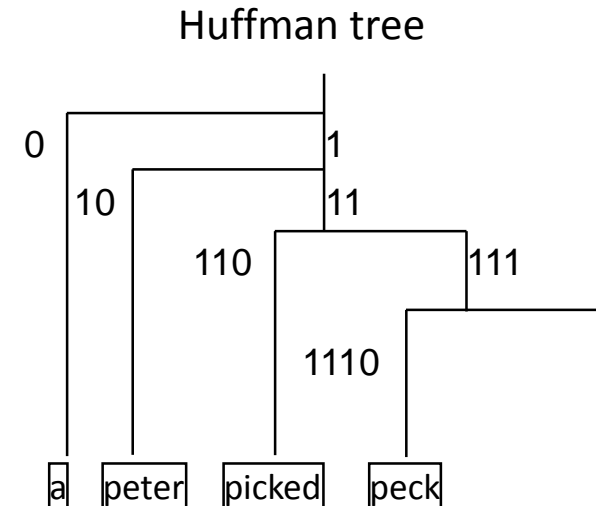
# Why Compression?

- **Zipf's law and Heaps' law suggest opportunities for compression**  
due to frequent terms or terms occurring repeatedly in documents
- **Compression of posting lists** is attractive for several reasons
  - **reduced space consumption** on disk or in main memory
  - **faster query processing**, since reading and decompressing data is nowadays often faster than reading uncompressed data
  - improved **cache effectiveness**, **since more posting lists fit into cache**

# 1. Huffman Coding

- **Variable-length unary code based on frequency analysis** of the underlying distribution of symbols (e.g., terms) in a text
- **Key idea:** Choose shortest unary code for most frequent symbol

Symbol $x$	Frequency $f(x)$	Huffman Encoding
<i>a</i>	0.8	0
<i>peter</i>	0.1	10
<i>picked</i>	0.07	110
<i>peck</i>	0.03	1110



# Entropy

- Let  $f(x)$  be the probability (or relative frequency) of the symbol  $x$  in some text  $d$ . The **entropy** of the text (or the underlying probability distribution) is defined as

$$H(d) = \sum_x f(x) \log_2 \frac{1}{f(x)}$$

- The entropy  $H(d)$  is a lower bound on the average (i.e., expected) number of bits per symbol needed with optimal compression.
- Huffman codes come close to the optimum  $H(d)$

# 2. Ziv-Lempel Compression

- LZ77 (Adaptive Dictionary) and further variants:
  - Scan text and identify in a lookahead window the longest string that occurs repeatedly and is contained in backwards window
  - Replace this string by a pointer to its previous occurrence
- Encode text into list of triples  $\langle \text{back}, \text{count}, \text{new} \rangle$  where
  - back is the backward distance to a prior occurrence of the string that starts at the current position
  - count is the length of this repeated string
  - new is the next symbol that follows the repeated string
- Triples themselves can be further encoded (with variable length)

# Ziv-Lempel Compression (Example)

- **Example:** peter\_piper\_picked\_a\_peck\_of\_pickled\_peppers

$\langle 0, 0, p \rangle$	for character 1:	p
$\langle 0, 0, e \rangle$	for character 2:	e
$\langle 0, 0, t \rangle$	for character 3:	t
$\langle -2, 1, r \rangle$	for characters 4-5:	er
$\langle 0, 0, \_ \rangle$	for character 6:	_
$\langle -6, 1, i \rangle$	for characters 7-8:	pi
$\langle -8, 2, r \rangle$	for characters 9-11:	per
$\langle -6, 3, c \rangle$	for characters 12-13:	_pic
$\langle 0, 0, k \rangle$	for character 16	k
$\langle -7, 1, d \rangle$	for characters 17-18	ed

...

- Great for text but not appropriate for compressing posting lists

# 3. Variable-Byte Encoding

- 32-bit binary code represents 12,038 using 4 bytes as

00000000 00000000 00101111 00000110

- **Variable-byte encoding** (aka. 7-bit encoding) uses one bit per byte as a continuation bit indicating whether the current number expands into the next bytes
- Variable-byte encoding represents 12,038 using only 2 bytes as

01011110 10000110

1 continuation bit

7 data bits

- **Byte-aligned**, i.e., each number corresponds to sequence of bytes

# 4. Gamma Encoding

- **Gamma ( $\gamma$ ) encoding** represents an integer  $x$  as
  - length =  $\text{floor}(\log_2 x)$  in **unary**
  - offset =  $x - 2^{\text{length}}$  in **binary**
- results in  $(1 + \log_2 x + \log_2 x)$  bits for integer  $x$
- **Not byte-aligned**, i.e., needs to be packed into bytes or words
- **Useful when distribution of numbers is not known ahead of time**  
or **when small numbers (e.g., gaps, tf) are frequent**

# Gamma Encoding (Examples)

<b>x</b>	<b>Gamma Encoding</b>	
$1 = 2^0$	u:0	
$4 = 2^2$	u:110	b:00
$24 = 2^4 + 2^3$	u:11110	b:1000
$131 = 2^7 + 3$	u:11111110	b:0000011

**Decoding:** Consider 111101000..... Read from left side until first 0, here, 11110, which tells offset=4 bits. The next 4 bits are 1000 which is 8 in decimal.  $8 + 2^4 = 24$



# 5. Golomb/Rice Encoding

- For **tunable parameter M**, split the number  $x$  into
  - **quotient**  $q = \text{floor}(x / M)$  stored in unary code (using  $q + 1$  bits)
  - **remainder**  $r = (x \bmod M)$  stored in binary code
- If  $M$  chosen as  $2^n$  then  $r$  needs  $\log_2(M)$  bits (**Rice encoding**)
- Otherwise for  $b = \text{ceil}(\log_2(M))$ 
  - If  $r < 2^b - M$  then  $r$  is stored in binary code using  $b - 1$  bits
  - Otherwise  $r + 2^b - M$  is stored in binary code using  $b$  bits
- **Not byte-aligned**, i.e., needs to be packed into bytes or words
- **Useful when distribution of numbers is known ahead of time**

# Golomb/Rice Encoding (Examples)

## Golomb Encoding ( $M = 10$ , $b = 4$ )

<b>x</b>	<b>q</b>	<b>bits(q)</b>	<b>r</b>	<b>bits(r)</b>
0	0	u:0	0	b:000
33	3	u:1110	3	b:011
57	5	u:111110	7	b:1101
99	9	u:1111111110	9	b:1111

# 6. Gap Encoding

- Variable-byte encoding, Gamma encoding, and Golomb/Rice encoding **represent smaller numbers using fewer bytes**
- **Note:** Posting lists contain sequences of increasing integers
  - document identifiers of postings in document-ordered posting list
  - offsets in posting payload if phrase queries need to be supported
- **Gap encoding** (aka. d-gaps) represents sequences of increasing integers as their first element followed by gaps  
<7, 12, 20, 25, 33, 78, ... >      .....▶      <7, 5, 8, 5, 8, 45, ... >

# 7. Run-Length Encoding

- Run-length encoding (e.g., used in early image formats like PCX) targets **sequences of integers having long runs of the same number** (i.e., **many repetitions** of that number in a row)
- Run-length encoding represents integer sequences as **(number, frequency) pairs**

$\langle 7, 7, 7, 8, 8, 1, 1, 1, 1, \dots \rangle \quad \dots \quad \langle (7, 3), (8, 2), (1, 4), \dots \rangle$

# Open Source Search Engines

- Apache Lucene / Apache Solr
  - implemented in Java, widely used in practice
  - <http://lucene.apache.org/core/> <http://lucene.apache.org/solr/>
- Indri
  - implemented in C++, academic IR system developed at CMU & U Mass
  - <http://www.lemurproject.org>
- Terrier
  - implemented in Java, academic IR system developed at U Glasgow
  - <http://terrier.org/>
- MG4J
  - implemented in Java, academic IR system developed at U Milano
  - <http://mg4j.dsi.unimi.it>

# Summary of III.2

- **Compression**  
is essential for performance in modern IR systems
- **Ziv-Lempel compression**  
as a dictionary-based encoding scheme that is great for text
- **Variable-byte encoding**  
as a byte-aligned non-parameterized encoding
- **Gamma encoding and Golomb/Rice encoding**  
as bit-aligned non-parameterized/parameterized encodings
- **Gap encoding and Run-length encoding**  
for transforming integer sequences

# Additional Literature for III.2

- S. Brin and L. Page: The anatomy of a large-scale hypertextual Web search engine, Computer Networks 30:107-117, 1998
- J. Dean: Challenges in Building Large-Scale Information Retrieval Systems, WSDM 2009, [http://videolectures.net/wsdm09\\_dean\\_cblirs/](http://videolectures.net/wsdm09_dean_cblirs/)
- A. Moffat and L. Stuiver: Binary Interpolative Coding for Effective Index Compression, Inf. Retr. 3(1): 25-47 (2000)
- H. Yan, S. Ding, T. Suel: Compressing Term Positions in Web Indexes, SIGIR 2009
- H. Yan, S. Ding, T. Suel: Inverted index compression and query processing with optimized document ordering, WWW 2009
- I. Witten, A. Moffat, and T. Bell: Managing Gigabytes (2nd Edition), Morgan Kaufmann, 1999
- J. Zhang, X. Long, T. Suel: Performance of compressed inverted list caching in search engines, WWW 2008
- M. Zukowski, S. Héman, N. Nes, P. A. Boncz: Super-Scalar RAM-CPU Cache Compression, ICDE 2006