Chapter V:
Indexing & Searching

Information Retrieval & Data Mining
Universität des Saarlandes, Saarbrücken
Winter Semester 2011/12

Chapter V: Indexing & Searching*

V.1 Indexing & Query processing

Inverted indexes, B^+-trees, merging vs. hashing,

Map-Reduce & distribution, index caching

V.2 Compression

Dictionary-based vs. variable-length encoding,

Gamma encoding, S16, P-for-Delta

V.3 Top-k Query Processing

Heuristic top-k approaches, Fagin's family of threshold-algorithms,

IO-Top-k, Top-k with incremental merging, and others

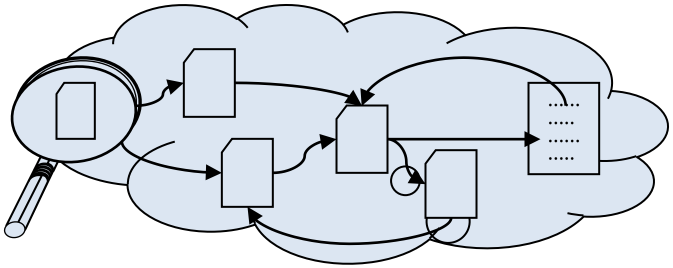
V.4 Efficient Similarity Search

High-dimensional similarity search, SpotSigs algorithm,

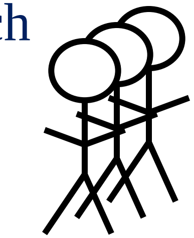
Min-Hashing & Locality Sensitive Hashing (LSH)

*mostly following Chapters 4 & 5 from **Manning/Raghavan/Schütze** and Chapter 9 from **Baeza-Yates/Ribeiro-Neto** with additions from recent research papers

V.1 Indexing



- Web, intranet, digital libraries, desktop search
- Unstructured/semistructured data

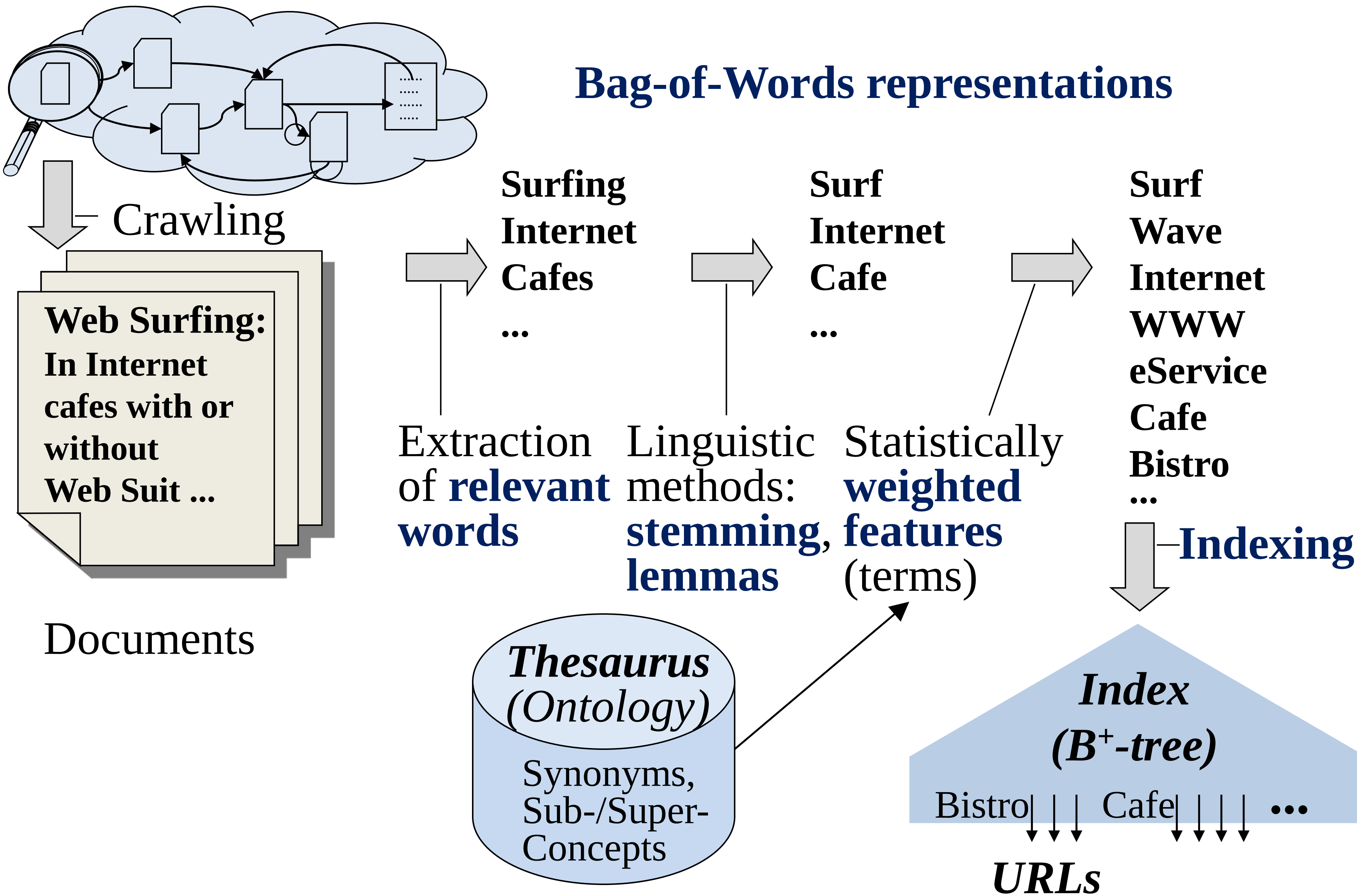


crawl → **extract & clean** → **index** → **search** → **rank** → **present**

- **crawl**: strategies for crawl schedule and priority queue for crawl frontier
- **extract & clean**: handle dynamic pages, detect duplicates, detect spam
- **index**: build and analyze Web graph, index all tokens or word stems
- **search**: fast top-k queries, query logging, auto-completion
- **rank**: scoring function over many data and context criteria
- **present**: GUI, user guidance, personalization

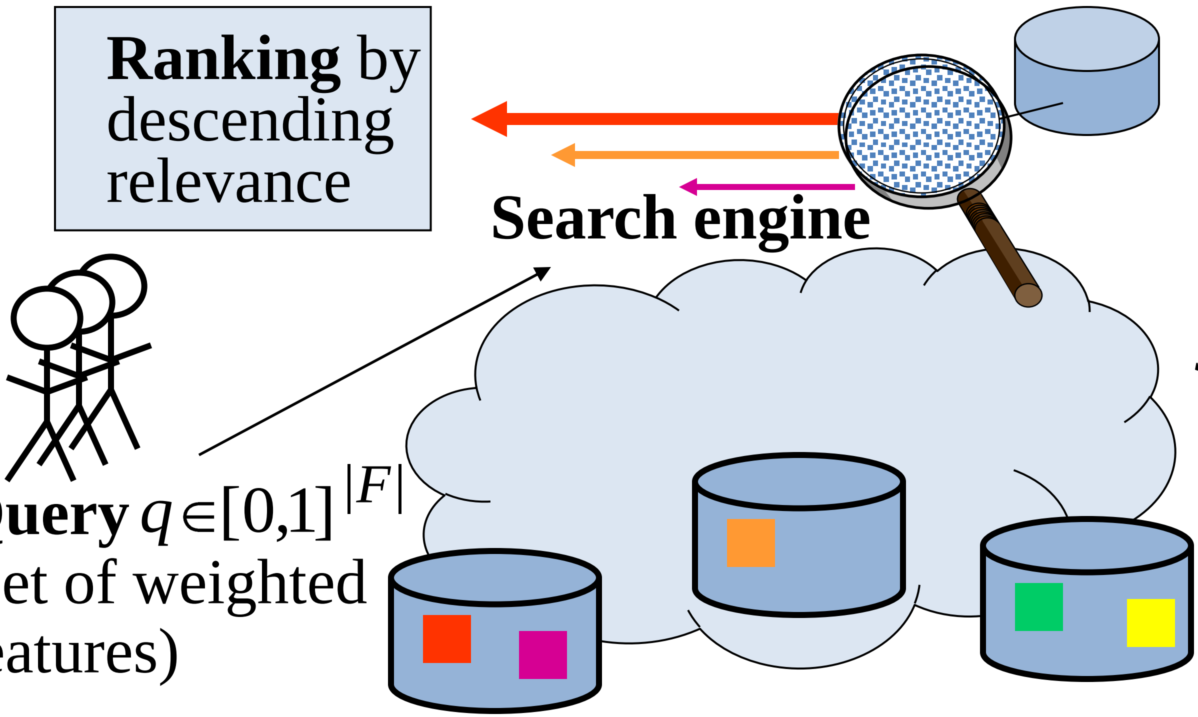
Server farms with **10 000's** (2002) – **100,000's** (2010) computers, distributed/replicated data in high-performance file system (**GFS**,**HDFS**,…), massive parallelism for query processing (**MapReduce**, **Hadoop**,…)

Content Gathering and Indexing



Vector Space Model for Relevance Ranking

Ranking by descending relevance



Query $q \in [0,1]^{|F|}$ (set of weighted features)

Documents are **feature vectors (bags of words)**

Similarity metric:
(e.g., Cosine measure)

$$sim(d_i, q) := \frac{\sum_{j=1}^{|F|} d_{ij}\, q_j}{\sqrt{\sum_{j=1}^{|F|} d_{ij}^2 \sum_{j=1}^{|F|} q_j^2}}$$

$$\text{with } d_i \in [0,1]^{|F|}$$

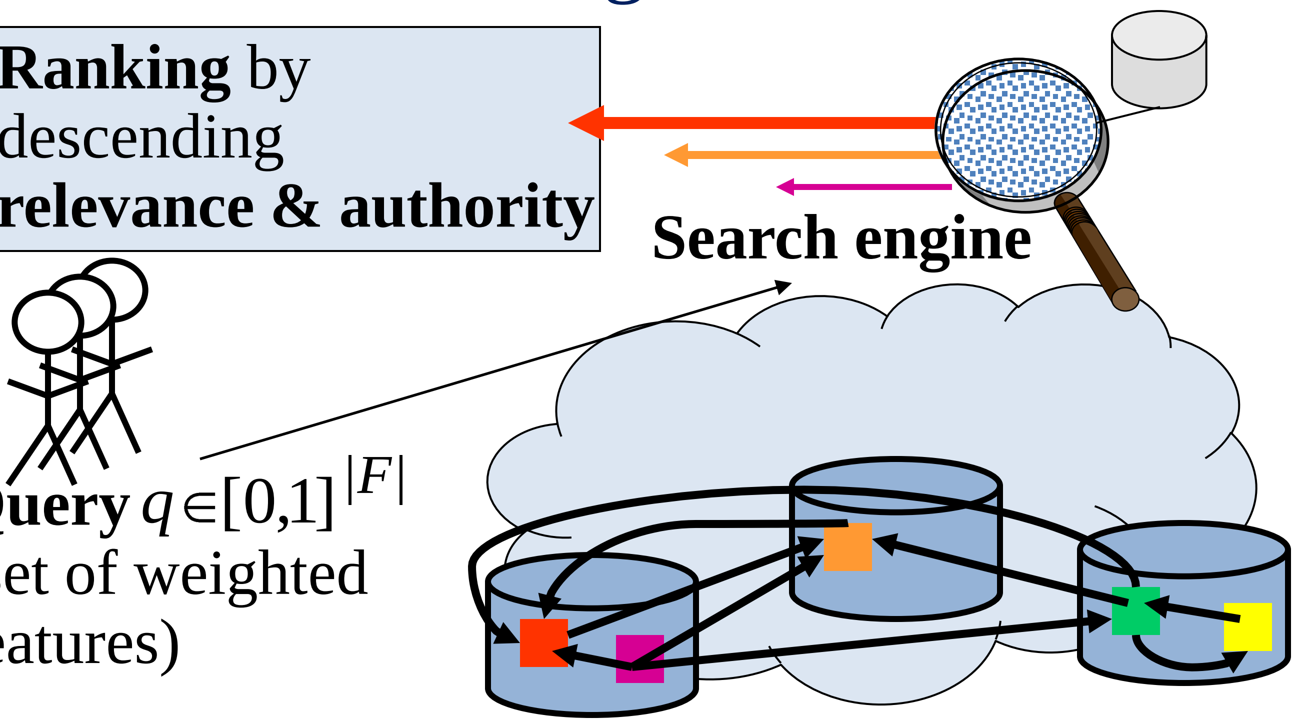
e.g., using: $d_{ij} := w_{ij} / \sqrt{\sum_k w_{ik}^2}$

$$w_{ij} := \log\left(1 + \frac{freq(f_j, d_i)}{\max_k \; freq(f_k, d_i)}\right) \log \frac{\#docs}{\#docs \; with \; f_i}$$

Using, e.g., tf*idf as weights

Combined Ranking with Content & Links Structure

Ranking by descending **relevance & authority**



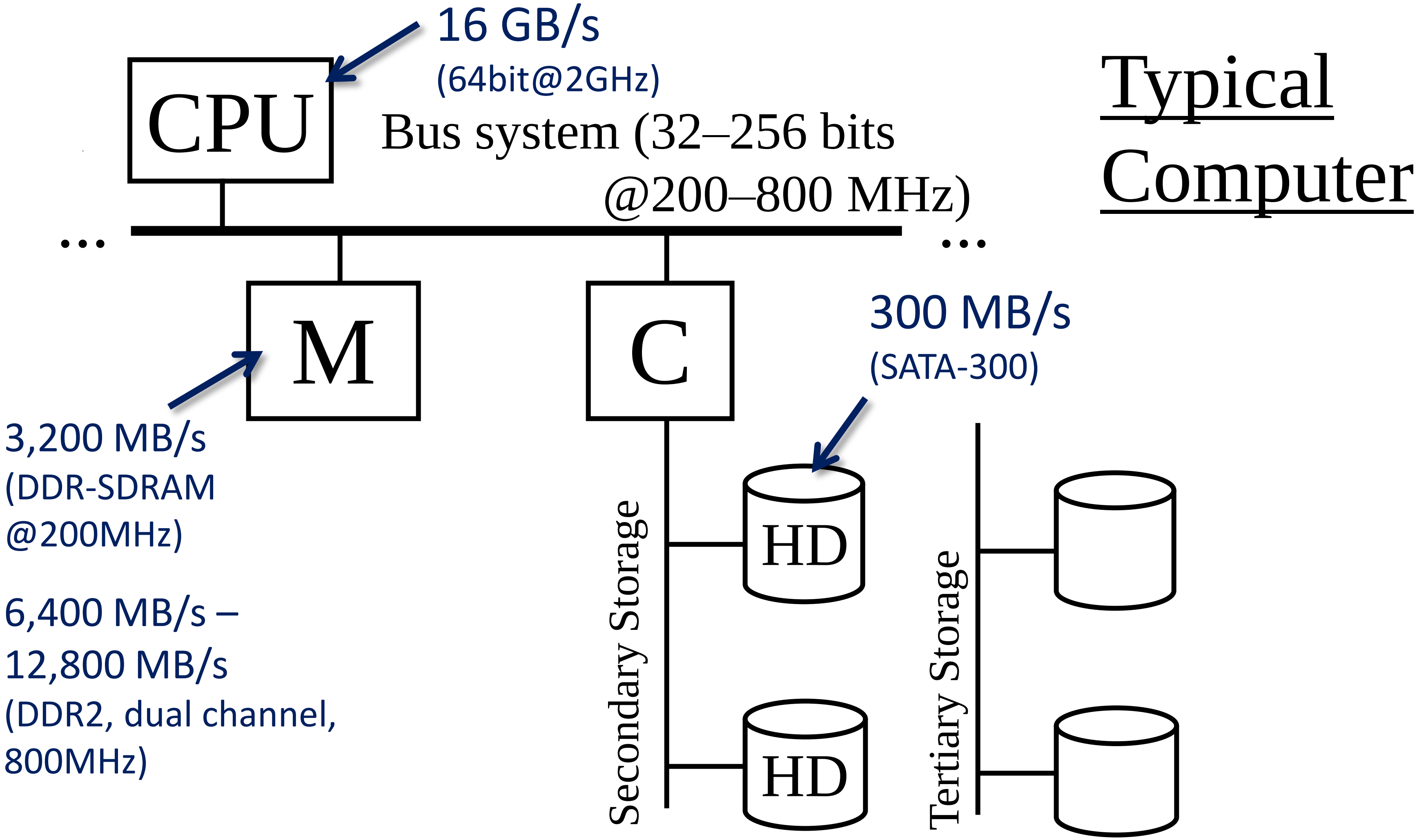
Query $q \in [0,1]^{|F|}$ (set of weighted features)

<u>Ranking functions:</u>

- **Low-dimensional queries (ad-hoc ranking, Web search):**
  BM25(F), authority scores, recency, document structure, etc.
- **High-dimensional queries (similarity search):**
  Cosine, Jaccard, Hamming on bitwise signatures, etc.

+ Dozens of more features employed by various search engines

Digression: Basic Hardware Considerations



TransferRate = **width** (number of bits) x **clock rate** x $\underbrace{\textbf{data per clock}}_{\text{typically 1}}$ / 8 **(bytes/sec)**

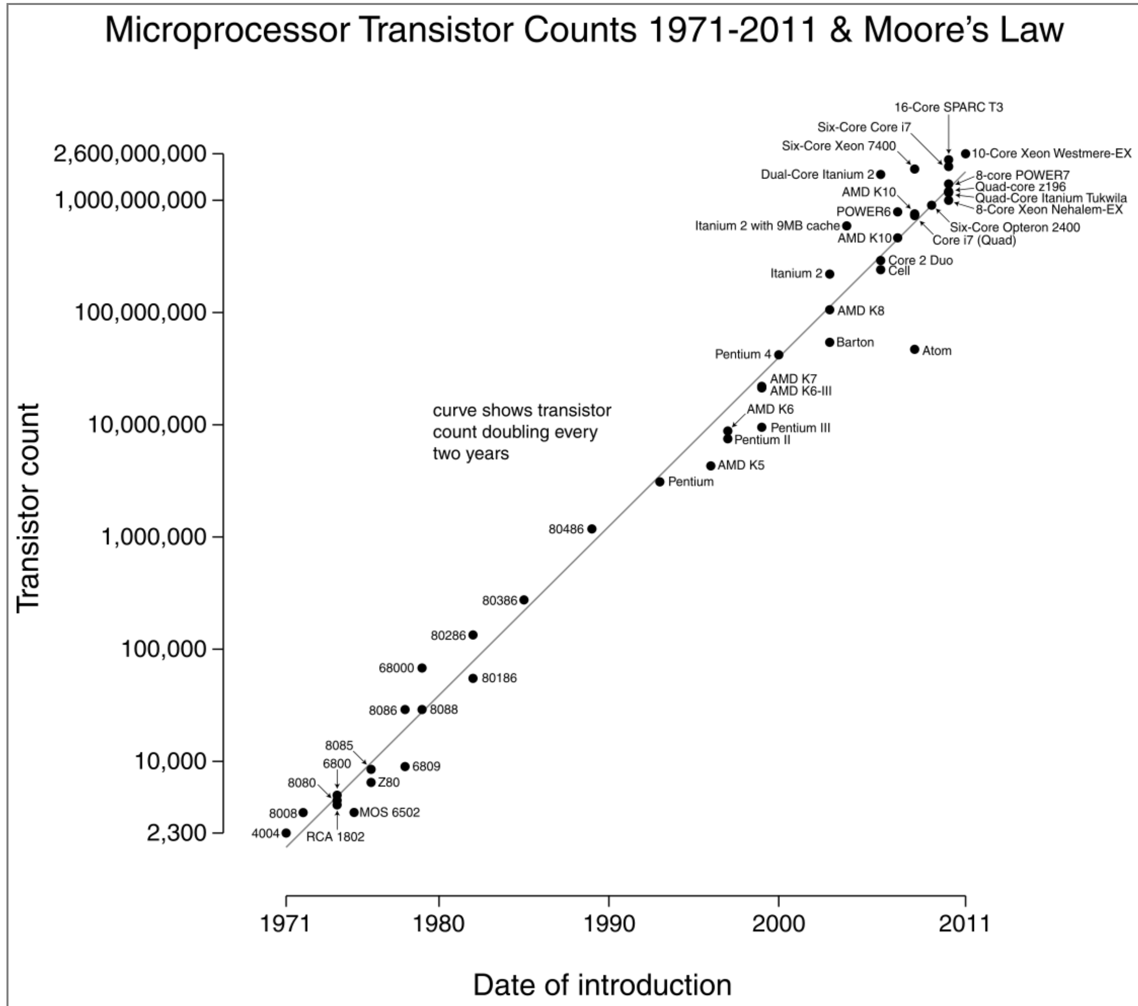
Moore's Law

Gordon Moore (Intel) anno 1965:

"The density of integrated circuits (transistors) will double every 18 months!"

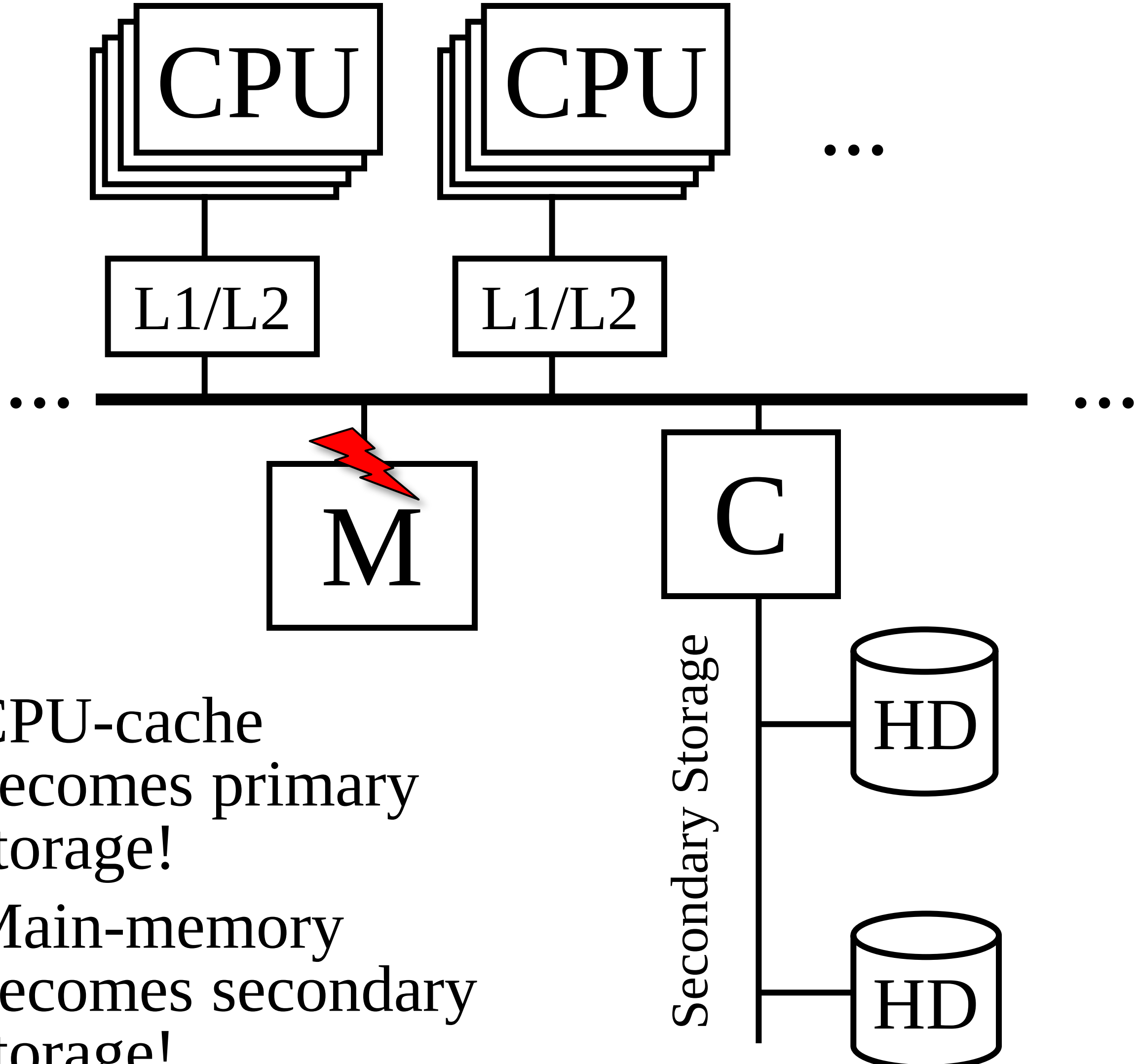
→ Has often been generalized to clock rates of CPUs, disk & memory sizes, etc.

→ **Still holds today for integrated circuits!**



Source: http://en.wikipedia.org/wiki/Moore%27s_law

More Modern View on Hardware



Multi-core-multi-CPU Computer

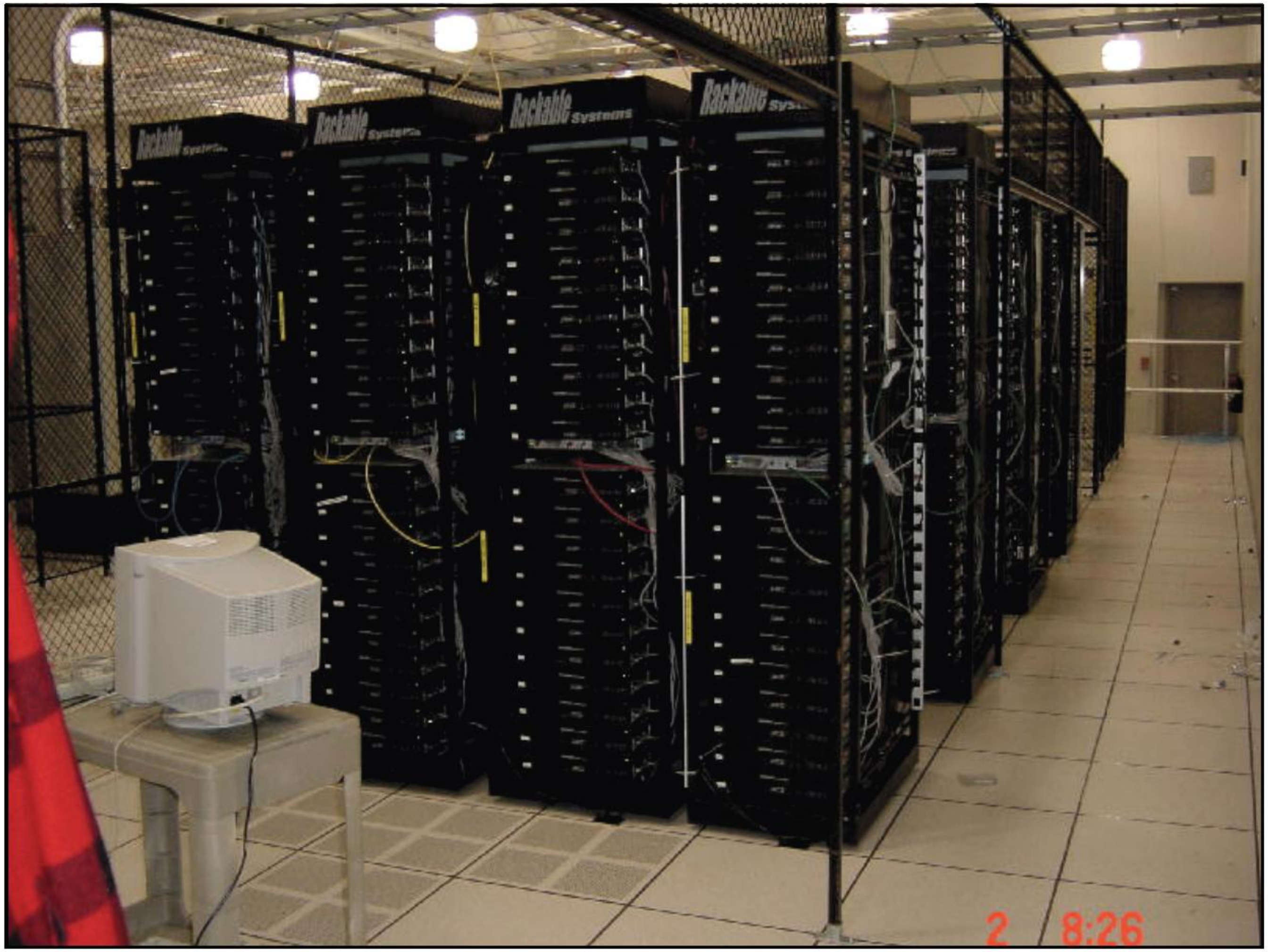
- CPU-cache becomes primary storage!
- Main-memory becomes secondary storage!

CPU-to-L1-Cache:
3-5 cycles initial latency, then "burst" mode

CPU-to-L2-Cache:
15-20 cycles latency

CPU-to-Main-Memory:
~200 cycles latency

Data Centers



Google Data Center anno 2004

Source: J. Dean: WSDM 2009 Keynote

Different Query Types

Conjunctive queries:
all words in $q = q_1 \ldots q_k$ required

Disjunctive ("andish") queries:
subset of q words qualifies,
more of q yields higher score

Mixed-mode queries and **negations**:
$q = q_1\ q_2\ q_3\ +q_4\ +q_5\ -q_6$

Find relevant docs
by list processing
on inverted indexes

Including variant:
- scan & merge
  only subset of q_i lists
- lookup long
  or negated q_i lists
  only for best result
  candidates

Phrase queries and **proximity** queries:
$q =$ "$q_1\ q_2\ q_3$" $q_4\ q_5 \ldots$

Vague-match (approximate) queries
with tolerance to spelling variants

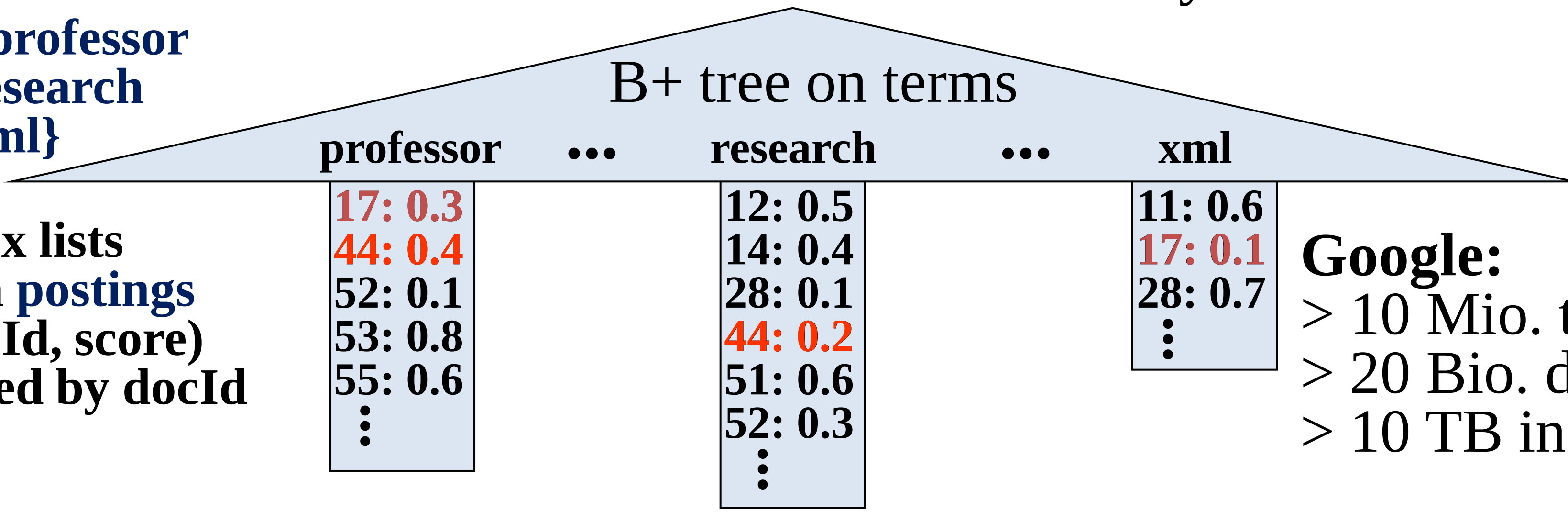
Structured queries and **XML-IR**
//article[about(.//title, "Harry Potter")]//sec

see Chapter III.5

Indexing with Inverted Lists

Vector space model suggests **term-document matrix**,
but data is sparse and queries are even very sparse.
→ Better use **inverted index lists** with terms as keys for B+ tree.

q: {professor research xml}



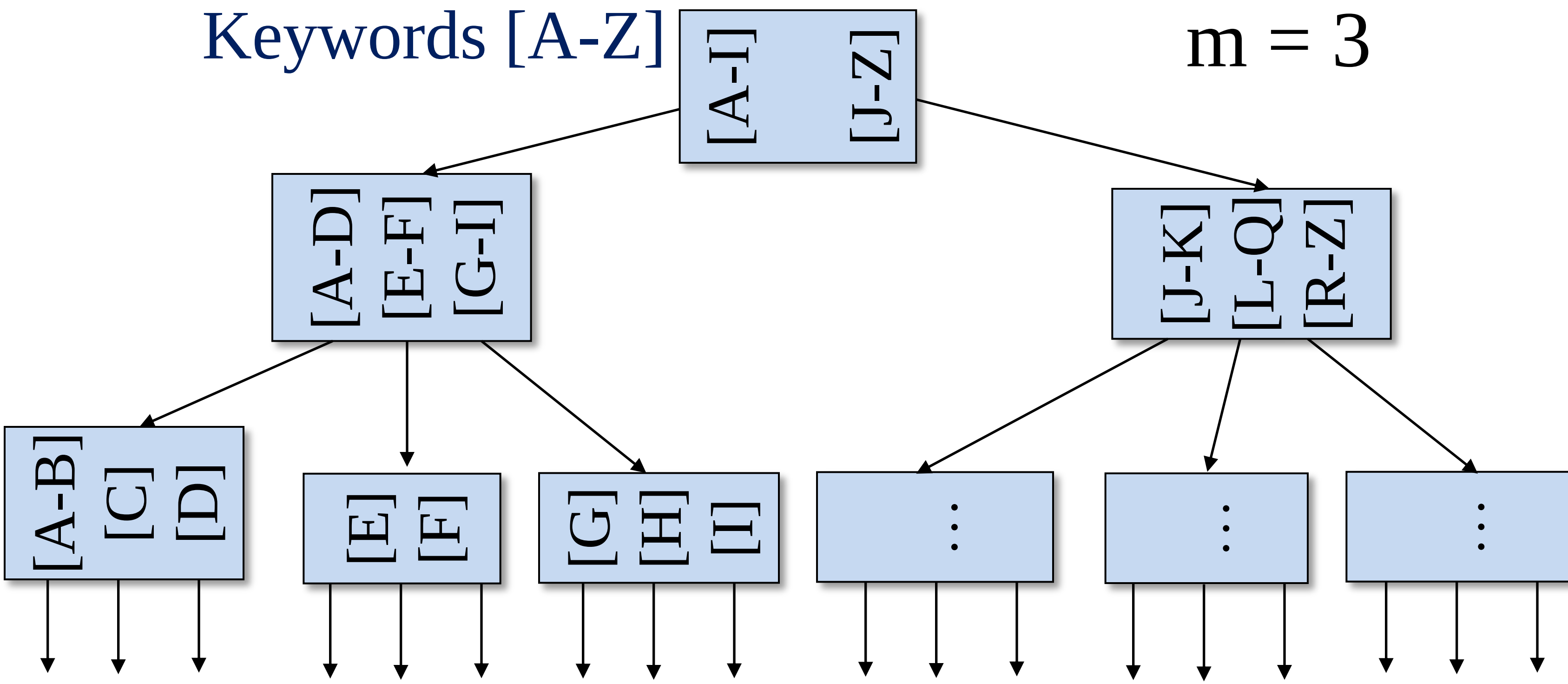
index lists with postings (docId, score) sorted by docId

Google:
> 10 Mio. terms
> 20 Bio. docs
> 10 TB index

terms can be full words, word stems, word pairs, substrings, N-grams, etc.
(whatever "dictionary terms" we prefer for the application)

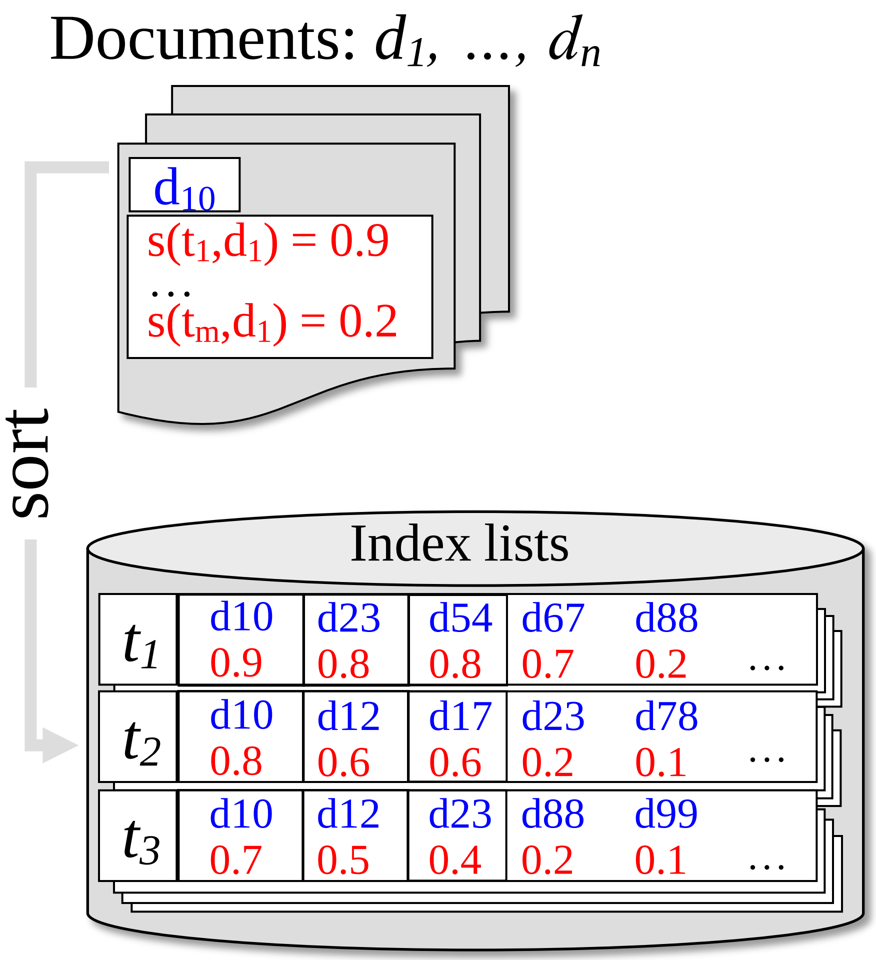
- Index-list entries in **docId order** for fast Boolean operations
- Many techniques for excellent **compression** of index lists
- Additional **position index** needed for phrases, proximity, etc. (or other pre-computed data structures)

B+-Tree Index for Term Dictionary



- **B-tree:** balanced tree with internal nodes of $\leq$m fan-out
- **B$^+$-tree:** leaf nodes additionally linked via pointers for efficient range scans
- For **term dictionary:** Leaf entries point to inverted list entries on local disk and/or node in compute cluster

Inverted Index for Posting Lists



Index-list entries usually stored in ascending order of docId (for efficient **merge joins**)

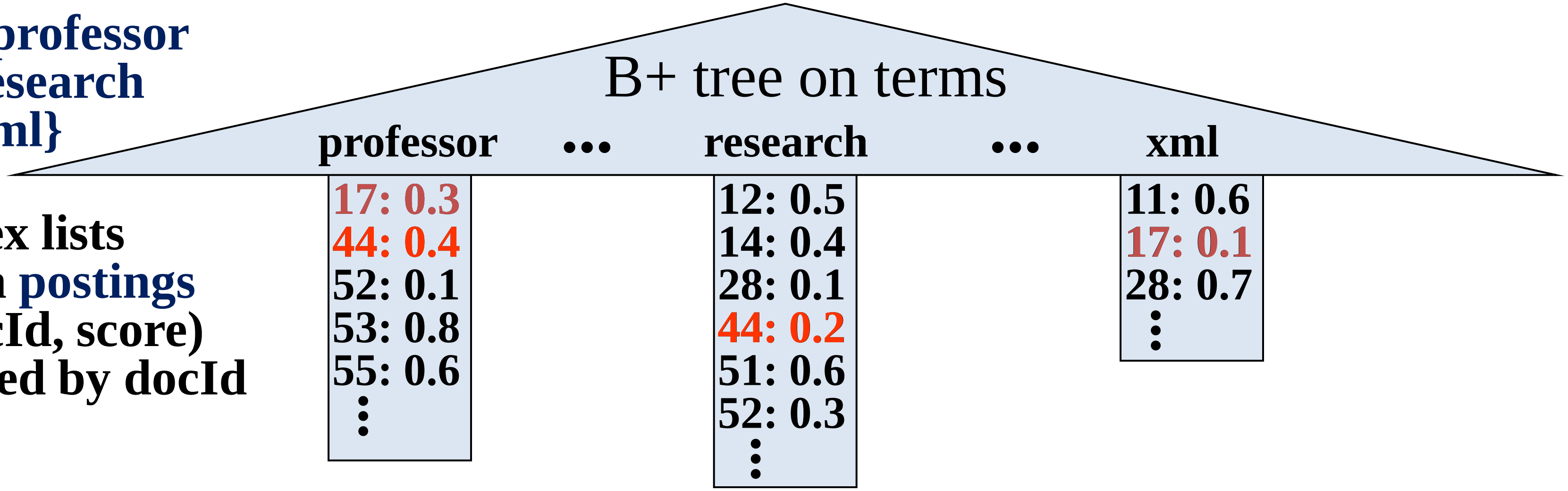
<u>or</u>

in descending order of per-term score (**impact-ordered lists** for top-k style pruning).

Usually compressed and divided into block sizes which are convenient for disk operations.

Query Processing on Inverted Lists

q: {professor research xml}



index lists with postings (docId, score) sorted by docId

<u>Given:</u> query $q = t_1\ t_2 \dots t_z$ with z (conjunctive) keywords
similarity scoring function ***score(q,d)*** for docs $d \in D$, e.g.: $\vec{q} \cdot \vec{d}$
with precomputed scores (index weights) $s_i(d)$ for which $q_i \neq 0$

<u>Find:</u> top-k results for ***score(q,d) =aggr{s_i(d)}*** (e.g.: $\Sigma_{i \in q}\ s_i(d)$)

Join-then-sort algorithm:

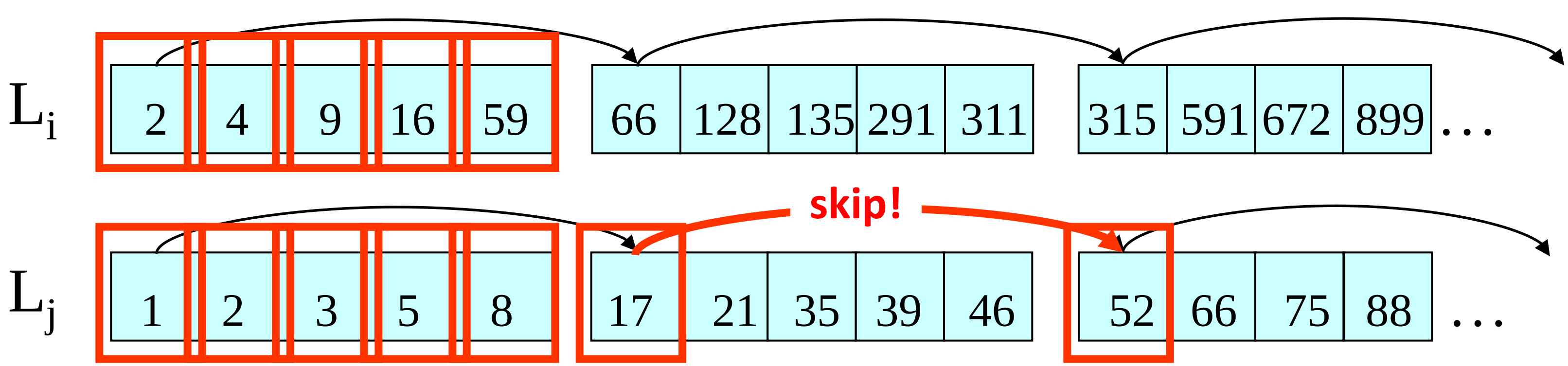
top-k (
$\sigma[term=t_1]\ (index)\ \bowtie_{DocId}$
$\sigma[term=t_2]\ (index)\ \bowtie_{DocId}$
...
$\bowtie_{DocId}$
$\sigma[term=t_z]\ (index)$
order by s desc)

Index List Processing by Merge Join

Keep L(i) in **ascending order of doc ids**.
Delta encoding: compress L_i by actually storing the gaps between successive doc ids (or using some more sophisticated prefix-free code).

QP may start with those L_i lists that are **short and have high idf**.
→ Candidates need to be looked up in other lists L_j.

To avoid having to uncompress the entire list L_j, L_j is encoded into **groups** (i.e., blocks) of **compressed entries** with a **skip pointer** at the start of each block → sqrt(n) evenly spaced skip pointers for list of length n.



Index List Processing by Hash Join

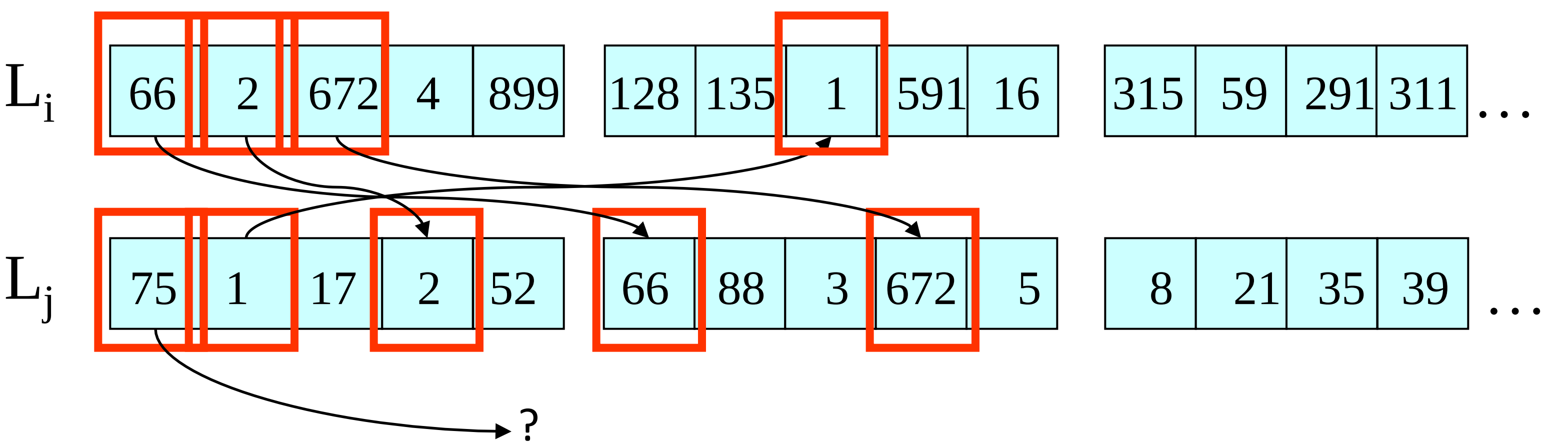
Keep L_i in **ascending order of scores** (e.g., TF*IDF).

Delta Encoding: compress L_i by storing the gaps between successive scores (often combined with variable-length encoding).

QP may start with those L_i lists that are **short and have high scores**, schedule may vary adaptively to scores.

→ Candidates can **immediately be looked up** in other lists L_j.

→ Can **aggregate candidate scores** on-the-fly.



Index Construction and Updates

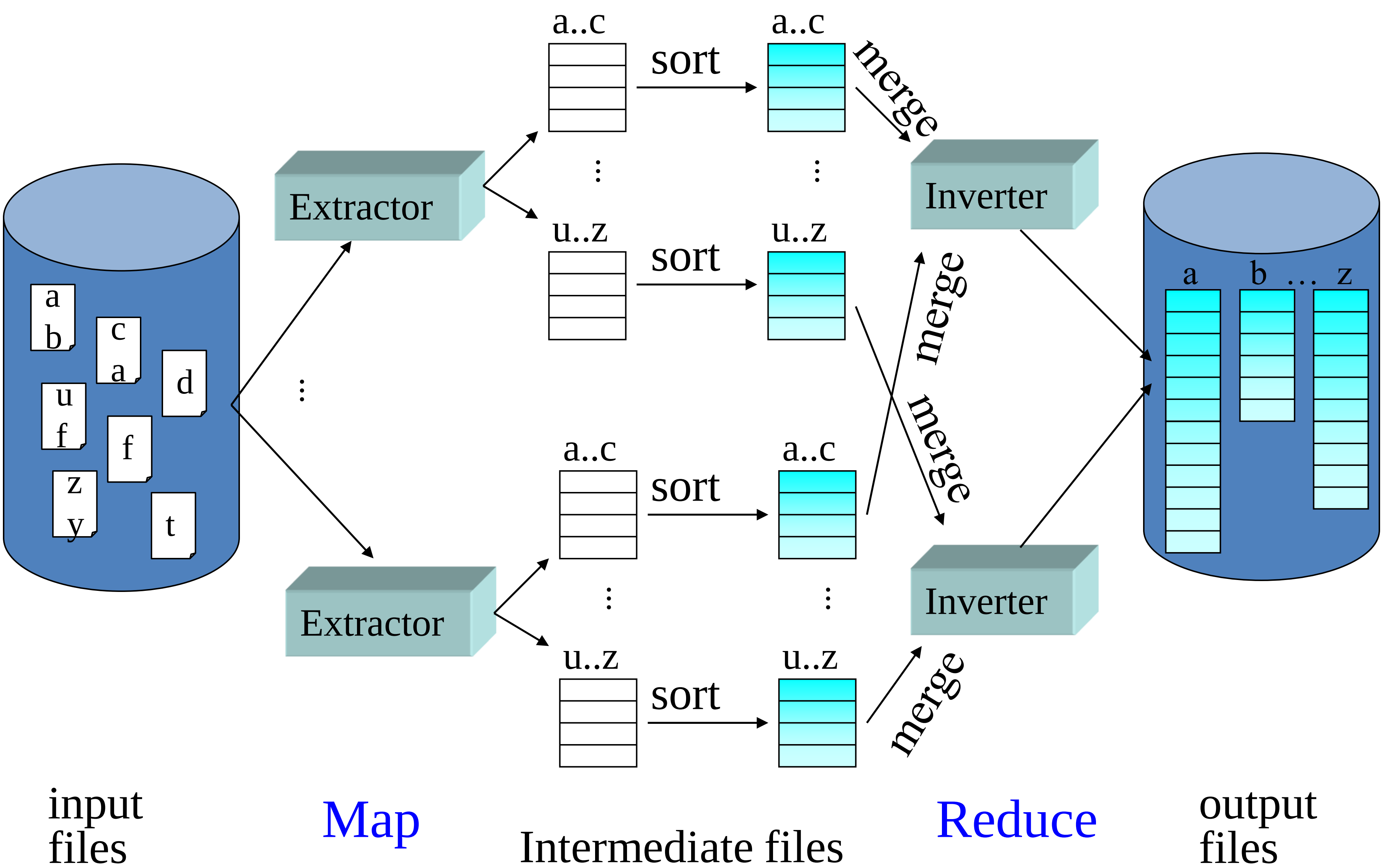
Index construction:

- 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 Parallelism 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: $K \times V \rightarrow (L \times W)^*$

$(k1, v1) \quad \mapsto \quad (l1,w1), (l2,w2), \ldots$

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

$l1, (x1, x2, \ldots) \mapsto \quad y1, y2, \ldots$

Examples:

- **Index building**: K=docIds, V=contents, L=termIds, W=docIds
- **Click log analysis**: K=logs, V=clicks, L=URLs, W=counts
- **Web graph reversal**: K=docIds, V=(s,t) outlinks, L=t, W=(t,s) inlinks

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 [<n1,f1>, <n2,f2>, …])
     P ← new List<posting>
     For posting <n, f> ∈ postings [<n1,f1>, <n2,f2>, …] 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

Challenge: Petabyte-Sort

Jim Gray benchmark:

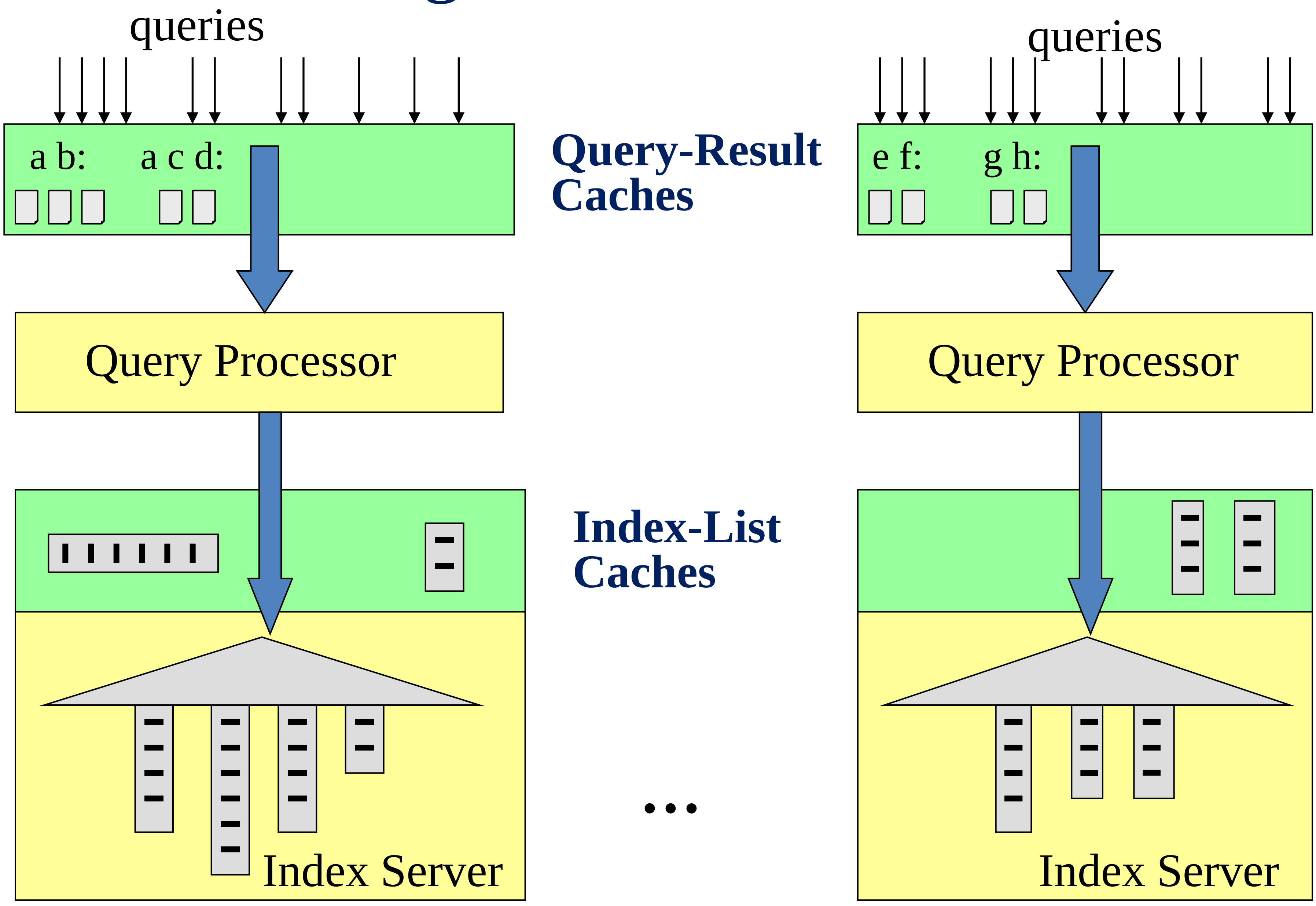
- Sort large amounts of 100-byte records (10 first bytes are keys)
- Minute-Sort: sort as many records as possible in under a minute
- Gray-Sort: must sort at least 100 TB, must run at least 1 hour

May 2011: Yahoo sorts 1 TB in 62 seconds and 1 PB in 16:15 hours on Hadoop

(http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_sorts_a_petabyte_in_162/)

Nov. 2008: Google sorts 1 TB in 68 seconds and 1 PB in 6:02 hours on MapReduce (using 4,000 computers with 48,000 hard drives)
(http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html)

Index Caching



Caching Strategies

What is cached?

- **index lists** for individual terms
- entire **query results**
- postings for **multi-term intersections**

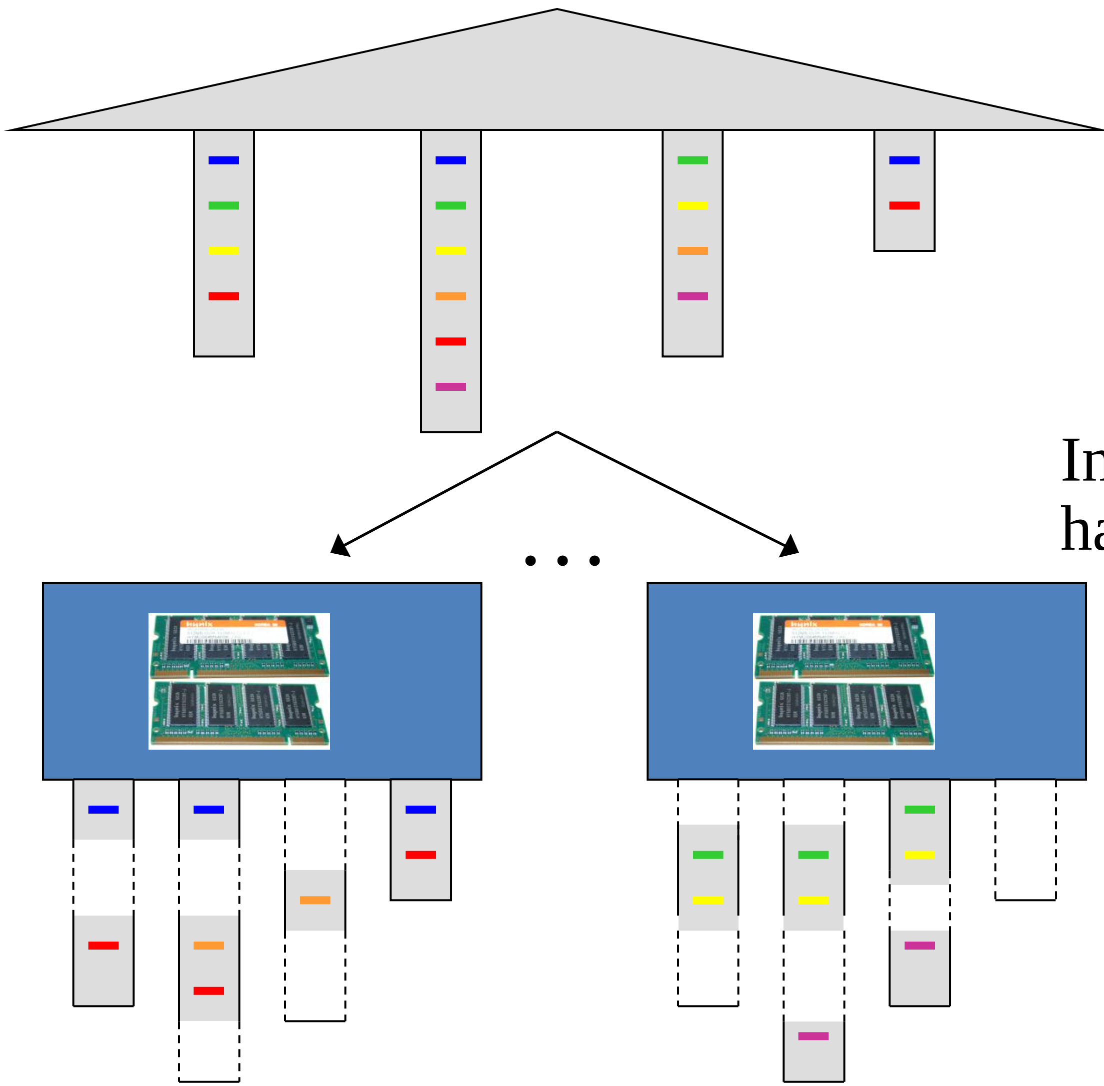
Where is an item cached?

- in RAM of responsible server-farm node
- in front-end accelerators or proxy servers
- as replicas in RAM of all (many) server-farm

When are cached items dropped?

- estimate for each item: **temperature = access-rate / size**
- when space is needed, drop item with lowest temperature
  Landlord algorithm [Cao/Irani 1997, Young 1998], generalizes LRU-k [O'Neil 1993]
- prefetch item if its predicted temperature is higher than
  the temperature of the corresponding replacement victims

Distributed Indexing: Doc Partitioning



Index-list entries are hashed onto nodes by docId.

Each complete query is run on each node; results are merged.

→ Perfect load balance, embarrasingly scalable, easy maintenance.

Data, Workload & Cost Parameters

- 20 Bio. Web pages, 100 terms each → 2 x 10^{12} index entries
- 10 Mio. distinct terms → 2 x 10^5 entries per index list
- 5 Bytes (amortized) per entry → 1 MB per index list, 10 TB total

- Query throughput: typical 1,000 q/s; peak: 10,000 q/s
- Response time: all queries in ≤ 100 ms
- Reliability & availability: 10-fold redundancy

- Execution cost per query:
  - 1 ms initial latency + 1 ms per 1,000 index entries
  - 2 terms per query

- Cost per PC (4 GB RAM): $ 1,000
- Cost per disk (1 TB): $ 500 with 5 ms per RA, 20 MB/s for SA's

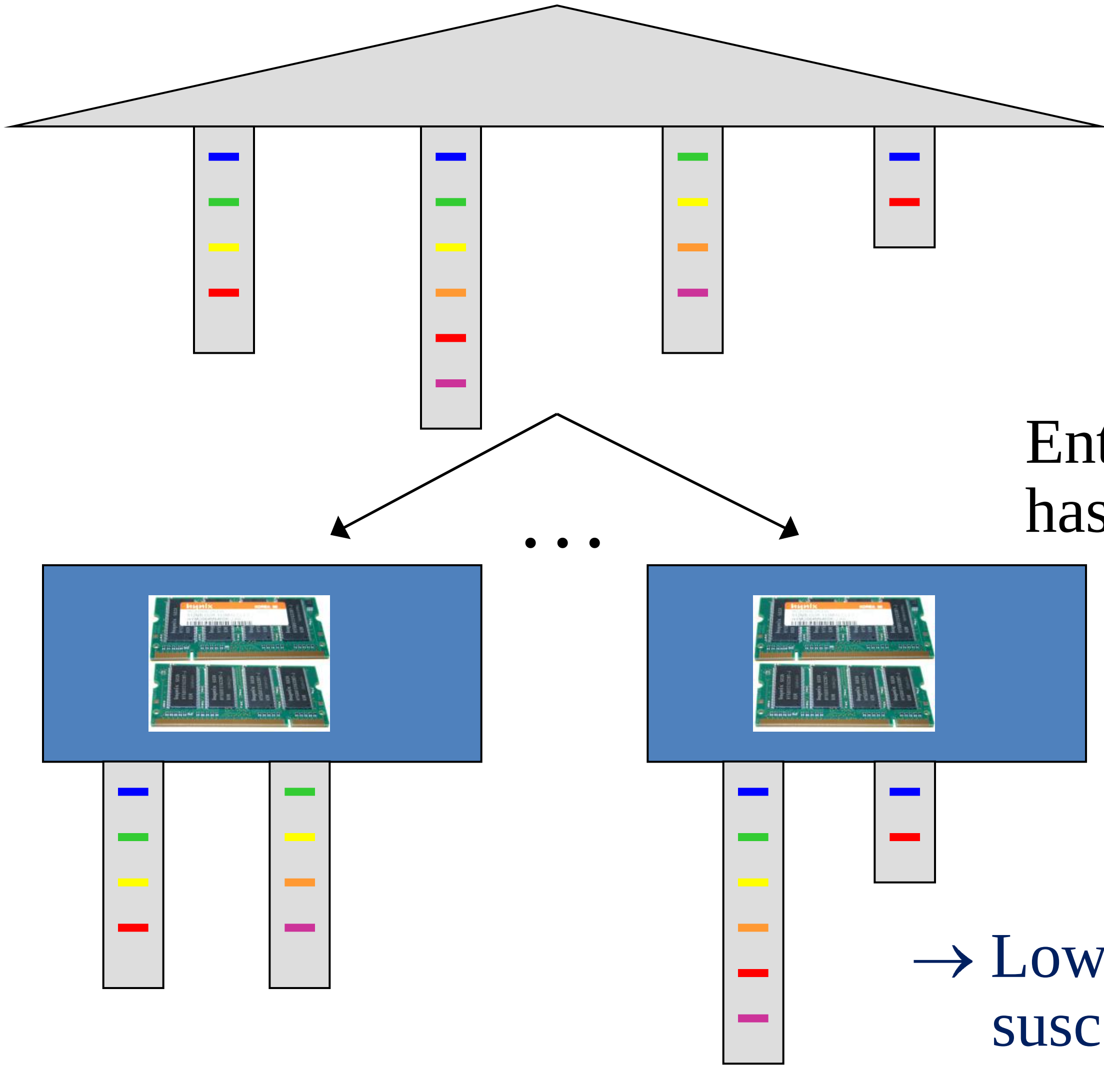
Back-of-the-Envelope Cost Model for Document-Partitioned Index (in RAM)

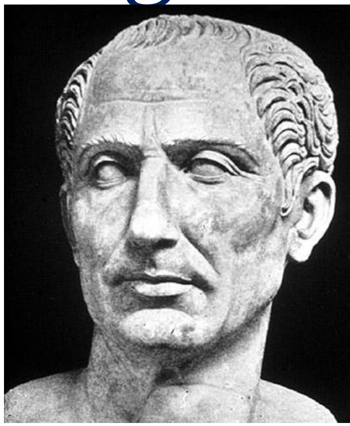
- 3,000 computers for one copy of index = 1 cluster
  - 3,000 x 4 GB RAM = 12 TB
    (10 TB total index size + workspace RAM)

- Query Processing:
  - Each query executed by all 3,000 computers in parallel:
    1 ms + (2 x 200 ms / 3000) ≈ 1 ms
    → each cluster can sustain ~1,000 queries / s

- 10 clusters = 30,000 computers to sustain peak load and guarantee reliability/availability
  → $ 30 Mio = 30,000 x $1,000 (no “big” disks)

Distributed Indexing: Term Partitioning



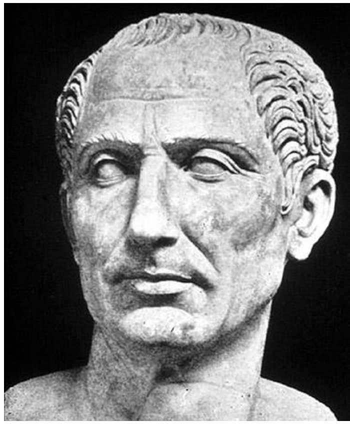


Entire index lists are hashed onto nodes by termId.

Queries are routed to nodes with relevant terms.

→ Lower resource consumption, susceptible to imbalance (because of data or load skew), index maintenance non-trivial.

Back-of-the-Envelope Cost Model for Term-Partitioned Index (on Disk)



- 10 nodes, each with 1 TB disk, hold entire index
- Execution time:
  max (1 MB / 20 MB/s, 1 ms + 200 ms)
  - but limited throughput:
  - 5 q/s per node for 1-term queries
- Need 200 nodes = 1 cluster
  to sustain 1,000 q/s with 1-term queries
  or 500 q/s with 2-term queries
- Need 20 clusters for peak load and reliability/availability
  4,000 computers → \$ 6 Mio = 4,000 x (\$1,000 + \$500)

saves money & energy
but faces challenge of update costs & load balance