Index Construction

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa

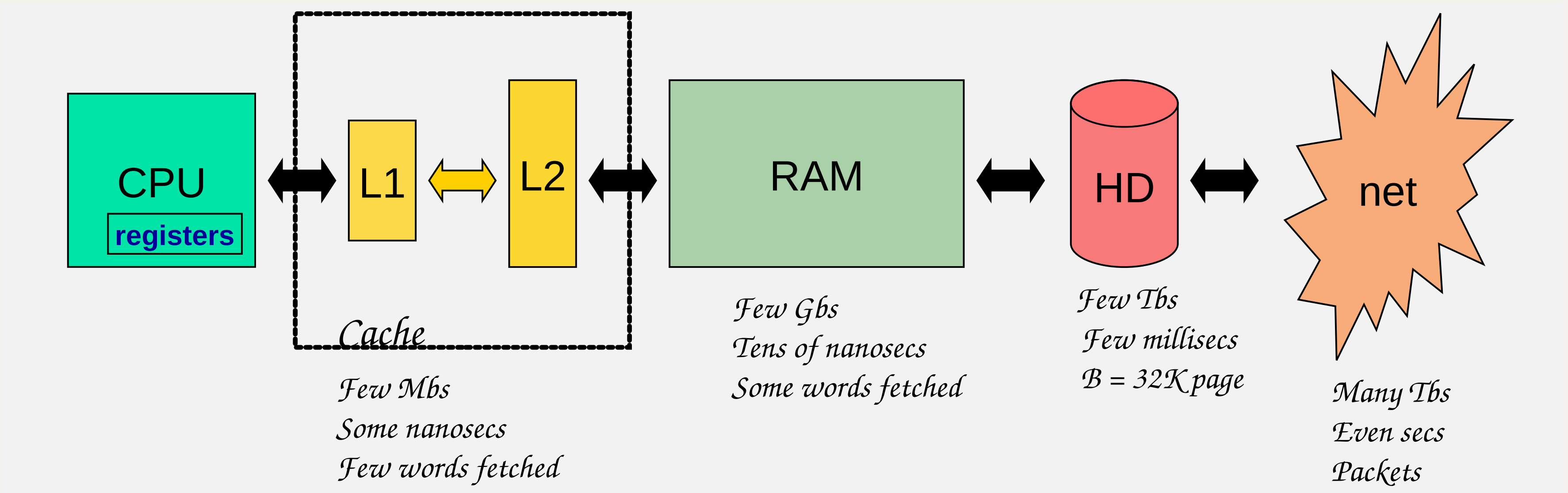
Basics

Brutus	→	1	2	4	11	31	45	173	174	
Caesar	→	1	2	4	5	6	16	57	132	...
Calpurnia	→	2	31	54	101					
⋮										

dictionary

postings

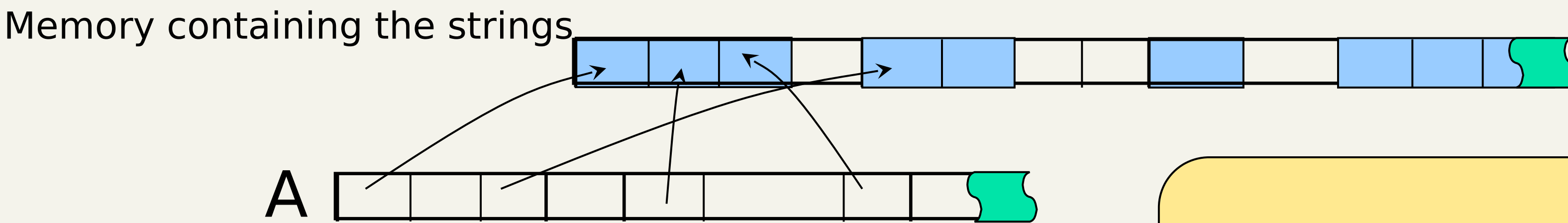
The memory hierarchy



Spatial locality or Temporal locality

Keep attention on disk...

- If sorting needs to manage <u>**strings**</u>



You sort A
Not the strings

Key observations:

- Array A is an "array of *pointers* to objects"
- For each object-to-object comparison A[i] *vs* A[j]:
  - 2 **random** accesses to 2 memory locations A[i] and A[j]
  - $\Theta(n \log n)$ **random** memory accesses (I/Os ??)

Again caching helps, but how much ?
Strings → IDs

SPIMI: Single-pass in-memory indexing

- Key idea **#1**: Generate separate dictionaries for each block of docs (No need for term → termID)
- Key idea **#2**: Accumulate postings in lists as they occur in each block of docs (in internal memory).

- Generate an inverted index for each block.
  - More space for postings available
  - Compression is possible

- What about one big index ?
  - Easy append with 1 file per posting (docID are increasing within a block)
  - But we have possibly many blocks to manage… (next!)

How do we:

- Find in dict ? …time issue…
- AddTo dict + posting? …space issues …
- Postings' size ? doubling
- Dictionary size ? … in-memory issues …

SPIMI-Invert

```
SPIMI-Invert(token_stream)
  output_file = NewFile()
  dictionary = NewHash()
  while (free memory available)
  do token ← next(token_stream)
     if term(token) ∉ dictionary
       then postings_list = AddToDictionary(dictionary, term(token))
       else postings_list = GetPostingsList(dictionary, term(token))
     if full(postings_list)
       then postings_list = DoublePostingsList(dictionary, term(token))
     AddToPostingsList(postings_list, docID(token))
  sorted_terms ← SortTerms(dictionary)
  WriteBlockToDisk(sorted_terms, dictionary, output_file)
  return output_file
```

What about one single index?

Doc 1

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

Doc 2

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

Term	docID
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	docID
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

Some issues

- Assign TermID
  - (1 pass)
- Create pairs <termID, docID>
  - (1 pass)
- Sort pairs by TermID
  - This is a **stable** sort

Term	docID
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	docID
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

Sorting on disk

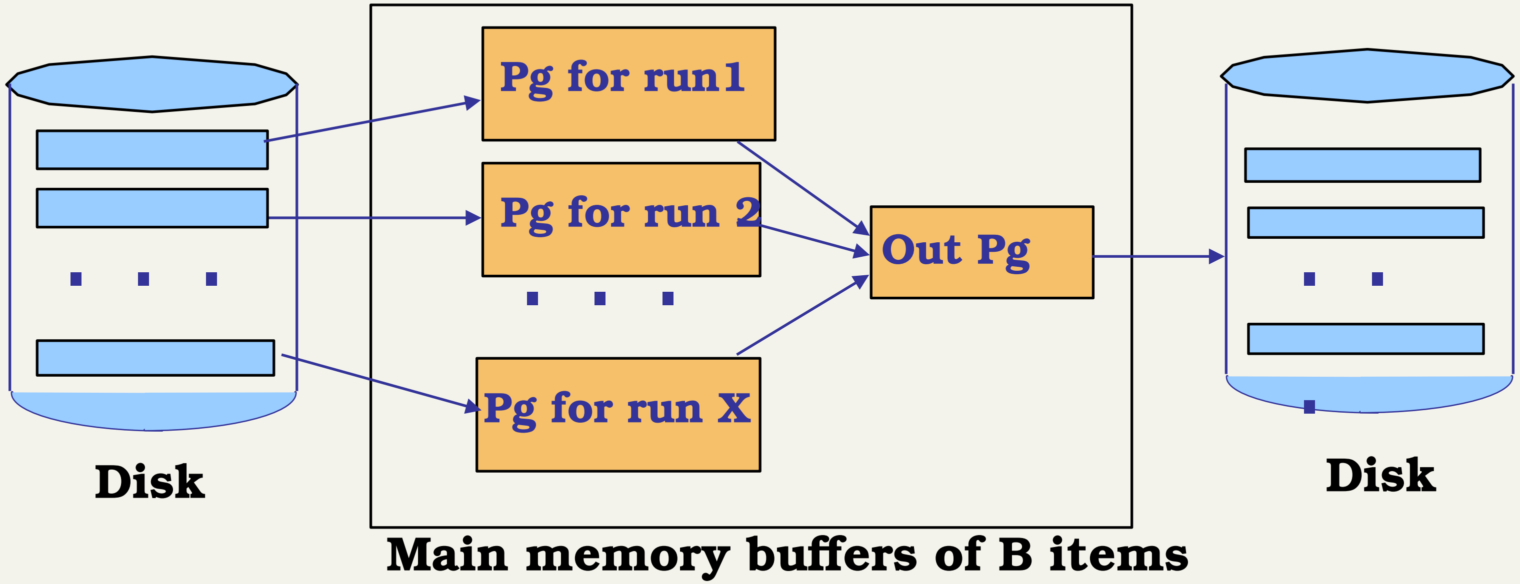
- **multi-way merge-sort**

  aka BSBI: Blocked sort-based Indexing

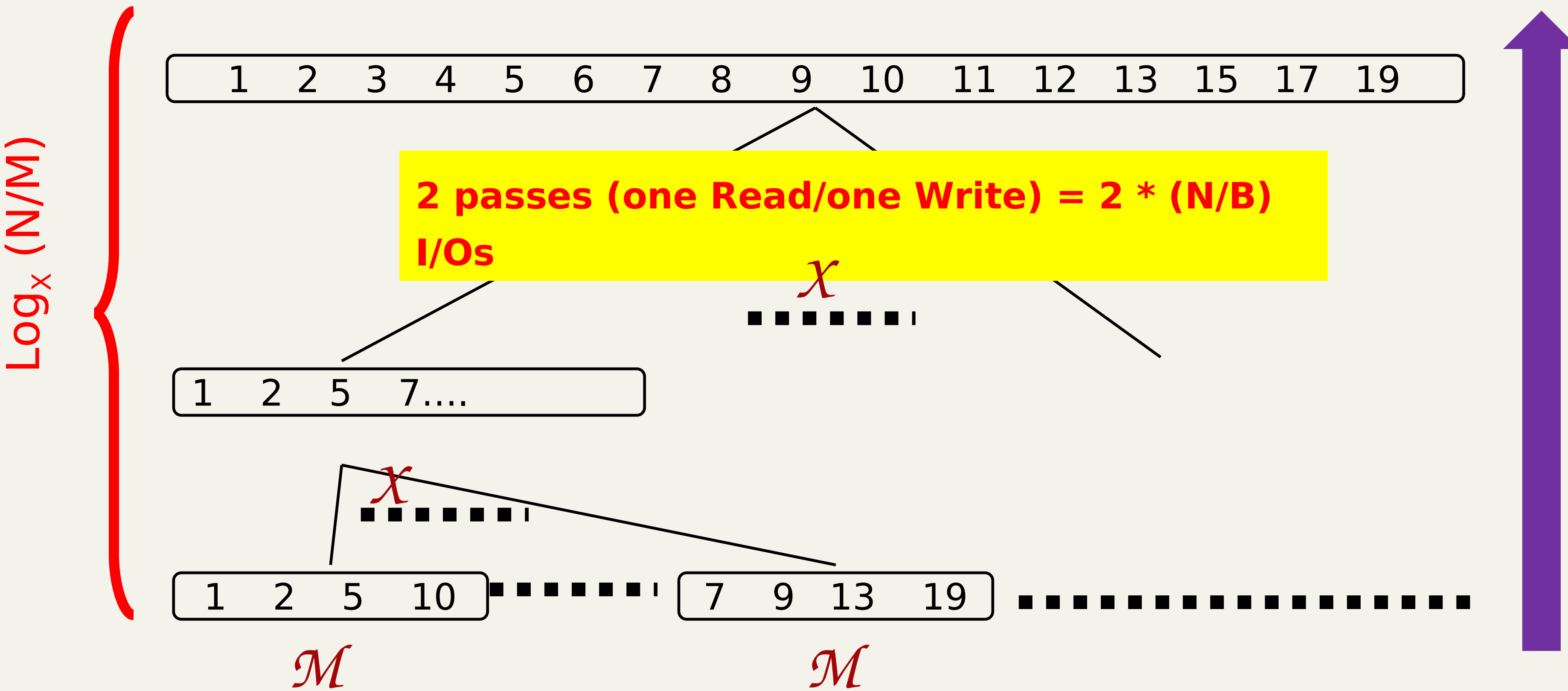
  - Mapping term → termID
    - to be kept in memory for constructing the pairs
    - Needs **two passes**, unless you use hashing and thus some probability of collision.

Multi-way Merge-Sort

- Sort *N* items with main-memory M and disk pages B:
  - Pass 1: Produce (N/M) sorted runs.
  - Pass i: merge X = *M/B-1* runs → $\log_X N/M$ passes



How it works



N/M runs, each sorted in internal memory = 2 (N/B) I/Os

— I/O-cost for X-way merge is ≈ 2 (N/B) I/Os per level

Cost of Multi-way Merge-Sort

- Number of passes = $\log_X N/M \cong \log_{M/B} (N/M)$
- Total I/O-cost is $\Theta((N/B) \log_{M/B} N/M)$ I/Os

In practice

- **M/B ≈ 10^5** → #passes = **1** → few mins

Tuning depends on disk features

✓ Large fan-out (M/B) decreases #passes

✓ Compression would decrease the cost of a pass!

Distributed indexing

- For web-scale indexing: must use a **distributed** computing cluster of PCs
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

Distributed indexing

- Maintain a *master* machine directing the indexing job – considered “safe”.
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns tasks to idle machines
- Other machines can play many roles during the computation

Parallel tasks

- We will use two sets of parallel tasks
  - Parsers and Inverters
- Break the document collection in two ways:

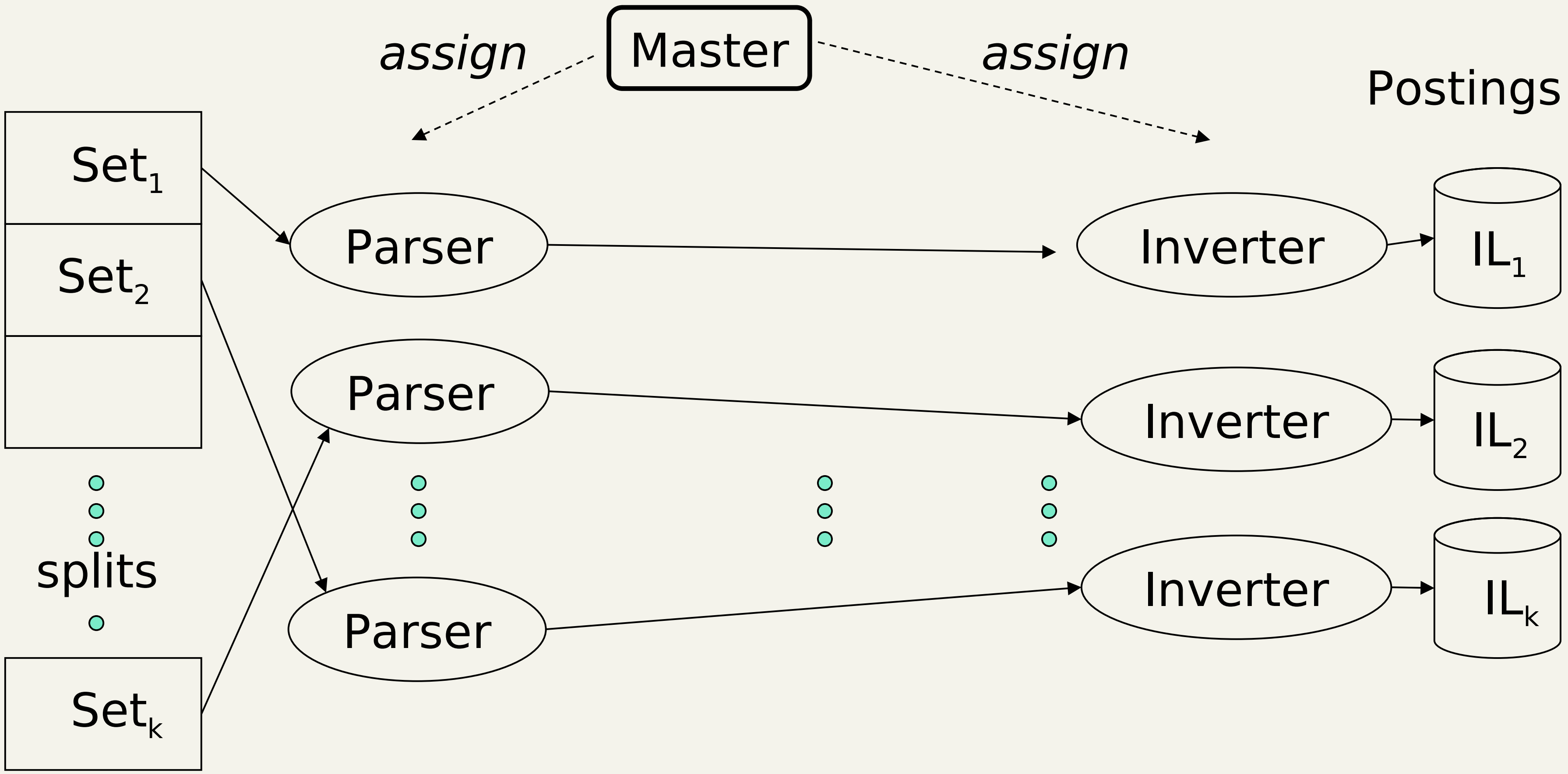
- **Term-based partition**

  one machine handles a subrange of terms

- **Doc-based partition**

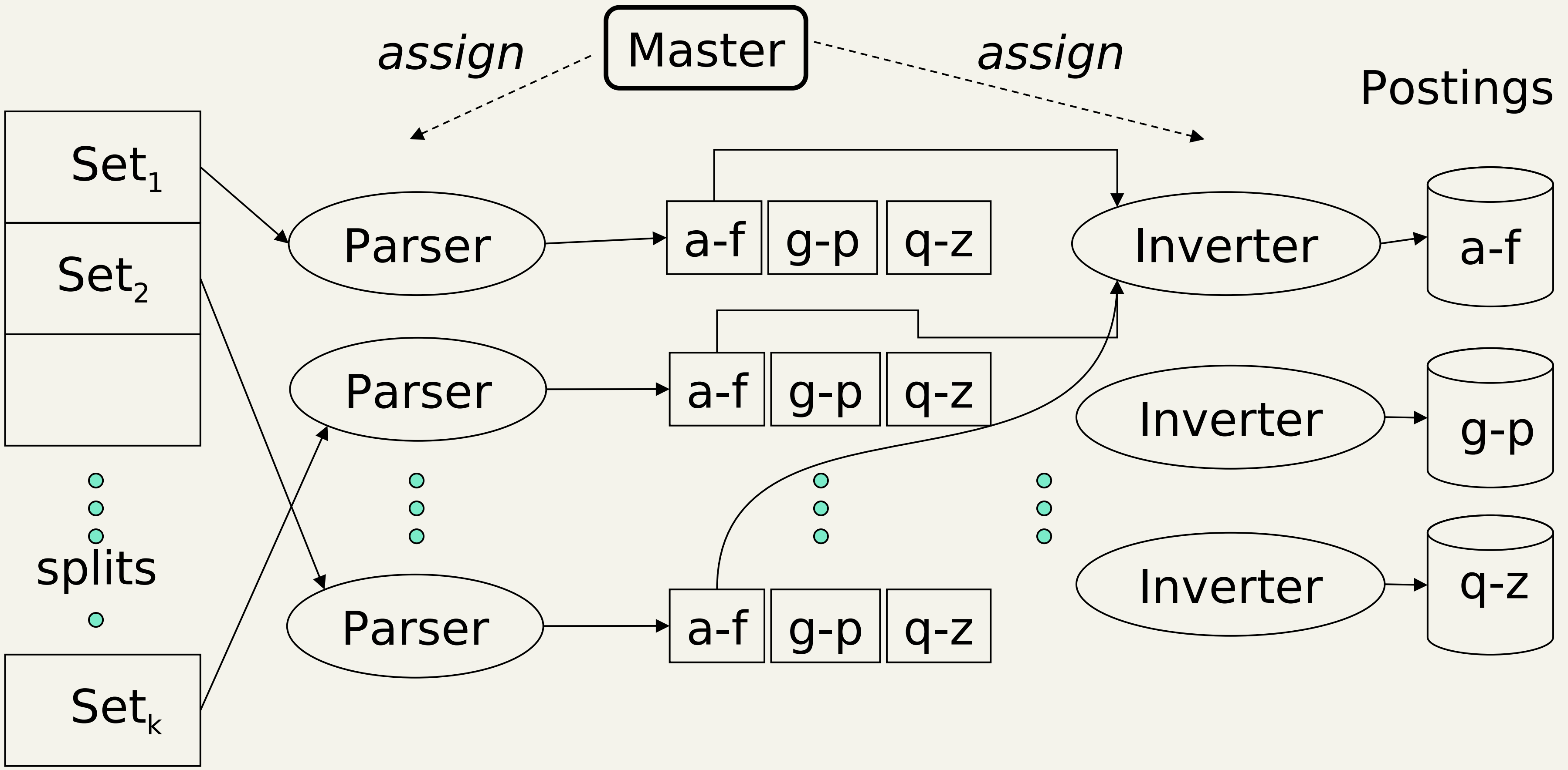
  one machine handles a subrange of documents

Data flow: **doc-based** partitioning



Each query-term goes to many machines

Data flow: **term-based** partitioning

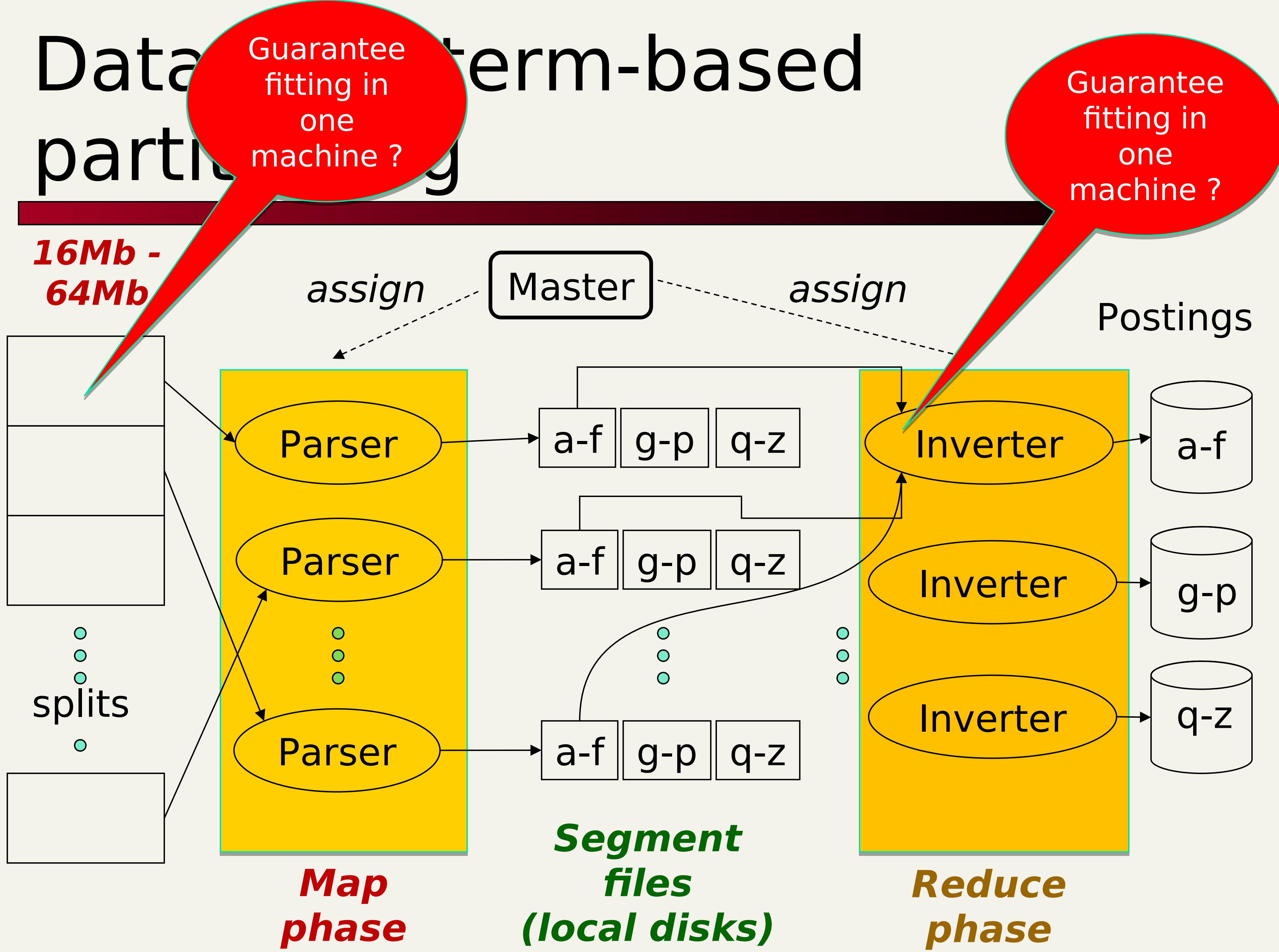


Each query-term goes to one machine

MapReduce

- This is
  - a robust and conceptually simple framework for distributed computing
  - … without having to write code for the distribution part.
- Google indexing system (ca. 2002) consists of a number of phases, each implemented in MapReduce.

Data … term-based parti… g



Dynamic indexing

- Up to now, we have assumed **static** collections.

- Now more frequently occurs that:
  - Documents come in over time
  - Documents are deleted and modified

- And this induces:
  - Postings updates for terms already in dictionary
  - New terms added/deleted to/from dictionary

Simplest approach

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, and merge the results

- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter search results (i.e. docs) by the invalidation bit-vector

- Periodically, re-index into one main index

Issues with 2 indexes

- Poor performance
  - 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 [new docIDs are greater].
  - But this needs a lot of files – inefficient for O/S.

- **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: M, 2^1 M , 2^2 M , 2^3 M , ...
- Keep a small index (Z) in memory (of size M)
- Store $I_{0,}$ I_1, I_2, ... on disk (sizes M , 2M , 4M,...)
- If Z gets too big (= *M*), write to disk as I_0

  or merge with I_0 (if I_0 already exists)
- Either write Z + I_0 to disk as I_1 (if no I_1)

  or merge with I_1 to form I_2, and so on
- etc.

indexes = logarithmic

Some analysis (C = total collection size)

- **Auxiliary and main index**: Each text participates to at most (C/M) mergings because we have 1 merge of the two indexes (small and large) every M-size document insertions.

- **Logarithmic merge**: Each text participates to no more than log (C/M) mergings because at each merge the text moves to a next index and they are at most log (C/M).

Web search engines

- Most search engines now support dynamic indexing
  - News items, blogs, new topical web pages
- But (sometimes/typically) they also periodically reconstruct the index
  - Query processing is then switched to the new index, and the old index is then deleted