Building in Parallel a Term-Partitioned Global Inverted File Index

Progetto del corso di Calcolo Parallelo
AA 2005-06

Salvatore Orlando

Indexing a collection of Web pages

- **Simple Boolean queries involving relationships between terms and documents**
  - **e.g. the queries supported by a Web Search Engine**
- ***Index***
  - **a data structure used to increase system performance for *word-based* queries**
- **Inversion**
  - M document files, containing N distinct terms
  - Documents identified by $docID_i$, and terms identified by $termID_j$
    - *Doc:* $\{docID_i\} \rightarrow \{termID_j\}$
  - Needed to *invert* such files, by determining function:
    - *Index*: $\{termID_j\} \rightarrow \{docID_i\}$

Tokenization/parsing

- Filter away tags
- Tokens defined as non-empty sequence of chars excluding spaces and
- punctuation
- Token represented by a suitable integer, TermID, typically 32 bits
- Optional: stemming/conflation of words

- Result
  - each document (identified by a DocID) transformed into a sequence of integers (TermID, pos)

Example of tokenization/parsing

- Original documents
  - File_foo: “My care is loss of care with old care done”
  - File_boo: “Your care is gain of care with new care won”
- Replaced by:
  - d1:
    - t1 t2 t3 t4 t5 t2 t6 t7 t2 t8
  - d2
    - t9 t2 t3 t10 t5 t2 t6 t11 t2 t12

- Typical parameters for a document collection of 5 GB
  - Total text size B 5 × GB
  - Number of docs N 5 × M
  - Number of distinct words n 1 × M
  - Total number of words F 800 × M

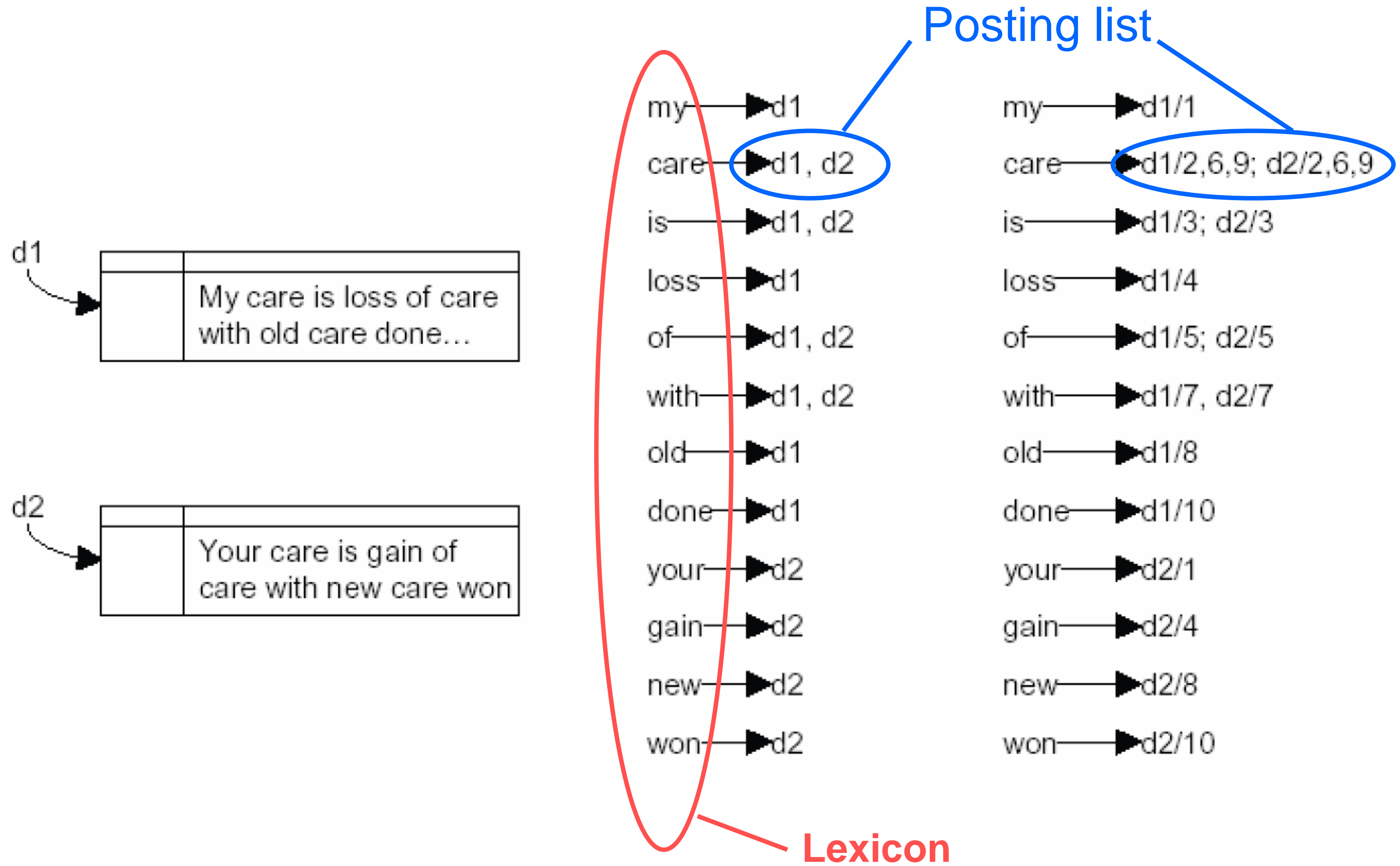
  - About 7 bytes/chars per word

Example of tokenization/parsing

- **d1:**
  - **t1 t2 t3 t4 t5 t2 t6 t7 t2 t8**
- **d2**
  - **t9 t2 t3 t10 t5 t2 t6 t11 t2 t12**

Doc	Term, Pos
1	1,1
1	2,2
1	3,3
1	4,4
1	5,5
1	2,6
1	6,7
1	7,8
1	8,10
2	9,1
2	2,2
2	3,3
2	10,4
2	5,5
2	2,6
2	6,7
2	11,8
2	2,9
2	12,10

Inversion: a complex task



Lexicon

- Every query goes to the Lexicon first
  - keep in memory or a separate file
  - search should be fast
  - support prefix or pattern matching
- Each entry in the lexicon/vocabulary has
  - the word
  - a pointer into the postings structure
  - word metadata (e.g., the document frequency of the words)

Posting

- **Postings**
  - **Addresses of words in text**

- **A posting within each list usually holds**
  - **document ID**
  - **count in the document**
  - **positions within the document**

- **Posting lists usually stored on disk**

- **For this project, we will consider posting lists containing DocIDs only**

Memory-based inversion

- **Informal outline**
  - **use a dynamic dictionary data structure (B-tree, hash table) to record distinct terms, with a linked list of nodes storing doc identifiers, etc.**
  - **once all documents have been processed, the dictionary is traversed, and all the terms and corresponding lists are written**

- **Problem**
  - **use of too much memory**
  - **impossible for very large collections**

Sort-based inversion

- **Main problems with the memory-based discussed so far**
  - **require too much memory**
  - **use data access sequence that is random, preventing an efficient mapping from memory onto disk**

- **Note that**
  - **for large disk files, sequential access is the only efficient processing mode, since transfer rates are usually high and random seeks are time-consuming**

- **For large volumes of data, the use of disk is inescapable**
  - **inversion should always perform sequential processing**
    $\Rightarrow$ ***sort-based inversion***

Parsing documents

- */* Parsing */*
- For each document Doc_i in the collection, 1 < i <<< N
  - (a) Read and parse Doc_i
  - (b) For each index term $t \in Doc_i$
    - Write record (t, i) to a temporary file
    - Usually also *frequency*, *positions* of t in the document, etc. are stored with (t, i)

- Usually t is represented by its term identifier in the lexicon S
- So, S should be searched for term t, and t should be added to S if necessary ….

- In the following we suppose to have an already pre-computed S, and thus to know all the possible terms, and identifiers associated with them …

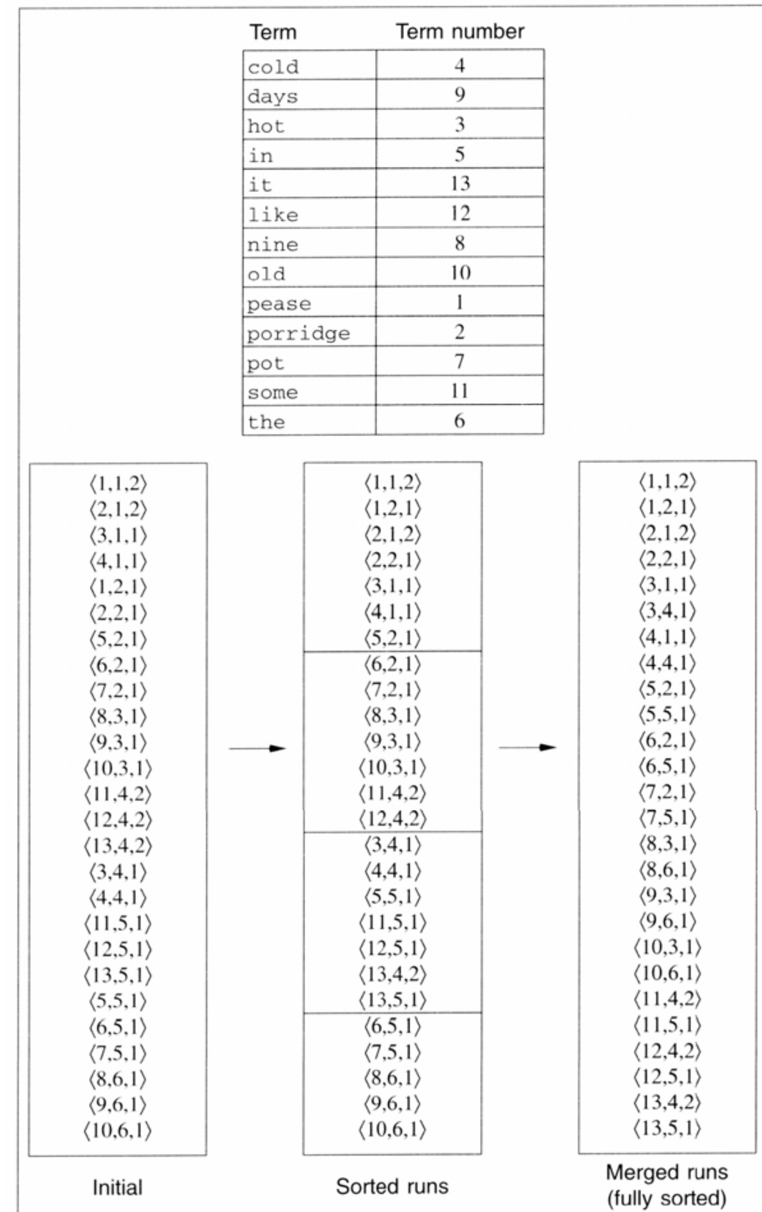
Internal sorting to make runs and run merging

- /* Building sorted runs */
- Let k be the number of records that can be held in memory
  - (a) Read k records *(t, i)* from the temporary file
  - (b) Sort them in memory into non-decreasing *t* order, and for equal values of *t*, non-decreasing *i* order
  - (c) Write the sorted run back to the temporary file
  - (d) Repeat until there are no more runs to be sorted
- /* Merging */
- Pairwise merge runs in the temporary file until it is one sorted run

- *Multiway merge*
  - in this case, a heap data structure (priority queue) must be exploited
  - HEAP: logically a binary tree, with the smallest value stored at the root of the tree
  - it is efficient, since it can be used to repetetetly find the smallest pair *(t, i)*, and efficiently insert a new pair that replaces the smallest one

Sort-based inversion: an example

Document	Text
1	Pease porridge hot, pease porridge cold
2	Pease porridge in the pot
3	Nine days old
4	Some like it hot, some like it cold
5	Some like it in the pot
6	Nine days old



Build the inverted file from the merged runs

- /* Output inverted file */
- For each term *t* read from the file containing the “merged runs”
  - (a) start a new inverted file entry
  - (b) read all pair *(t, i)* from the temporary file, and form the posting list for term *t*
    - Note that all pairs *(t, i)* appear in consecutive positions
    - if *(t, i)* appears before *(t, j)*, then $i < j$
  - (c) if required, compress the posting list
  - (d) append this posting list to the inverted file

Partitioning the inverted files

- **The inverted file can be distributed across different parallel nodes**
  - ***parallel* query engine of the *web search engine***

- **Two different partitioning strategies can be devised**
- ***Term partitioning***
  - **Horizontally partitioning the whole inverted index with respect to the lexicon**
  - **Each query server stores the inverted lists associated with only a subset of the index terms.**
- ***Document partitioning***
  - **Vertical partitioning of the inverted index**
  - **Each query server becomes responsible for a disjoint subset of the whole document collection**

- **The last approach is easier**
  - **the construction of an IF index partition can be carried out locally and independently by each node, using a distinct *document partition* of the whole collection.**

Term vs. Document Partitioning

my → d1
care → d1, d2
is → d1, d2
loss → d1
of → d1, d2
with → d1, d2
old → d1
done → d1
your → d2
gain → d2
new → d2
won → d2

Term Partitioning

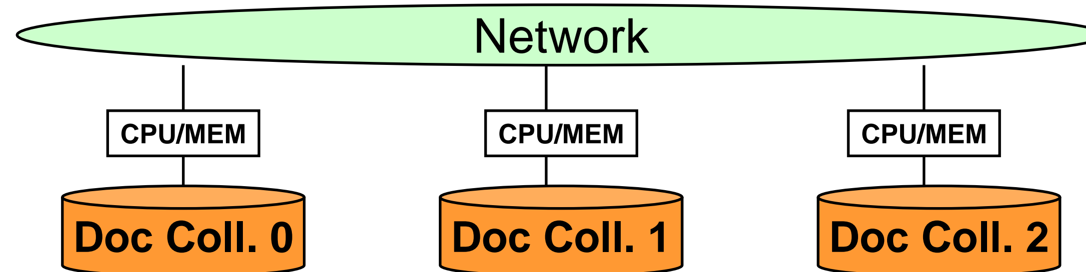
Node 0	Node 1
my → d1	old → d1
care → d1,d2	done → d1
is → d1, d2	your → d2
loss → d1	gain → d2
of → d1,d2	new → d2
with → d1,d2	won → d2

Document Partitioning

Node 0	Node 1
my → d1	my →
care → d1	care → d2
is → d1	is → d2
loss → d1	loss →
of → d1	of → d2
with → d1	with → d2
old → d1	old →
done → d1	done →
your →	your → d2
gain →	gain → d2
new →	new → d2
won →	won → d2

Project assignment

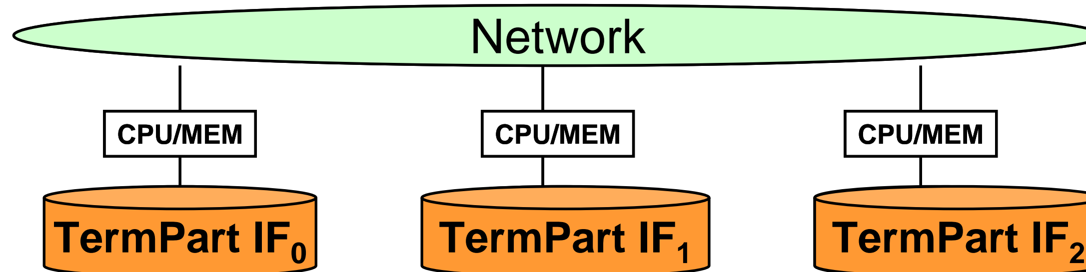
– A distinct doc collection is stored on each node N_i



– The collection has been parsed, and thus transformed into a set of pairs
  - Each node has a single file containing unordered pairs
    – **<TermID, DocID>**
  - TermID are *global identifiers*
    – **if different nodes store two pairs <x, y> and <x, z>, both the pairs refer to the same term x**

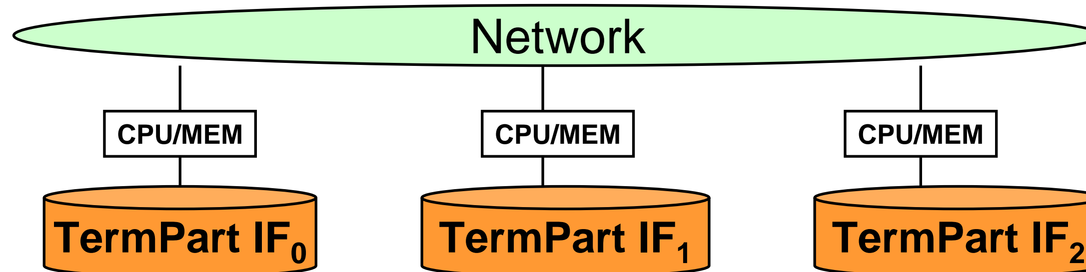
– Problem to solve:
  - build in parallel the global IF index, and partition it according to a ***term partitioning*** approach

Project assignment



- **Problem to solve**:
  - build in parallel the global IF index, and partition it according to a ***term partitioning*** approach
- Given:
  - Total number of Terms is **M**
  - Total number of Nodes is **N**
  - Total number of Docs is **D**
  - The number of docs per node is ***approximately*** **D/N**
- **Issues**:
  - Order the **termIDs**
  - At the end: M/N posting lists must be stored on the each node
    - **the lists corresponding to the first M/N TermIDs stored on Node 0 (TermPart$_0$)**
    - **the lists corresponding to the next M/N TermIDs stored on Node 1 (TermPart$_1$)**
    - **and so on …**
  - the **DocIDs** are local identifiers …
    - **they must be transformed first into global identifiers DocID', ranging from 1 to D**
  - the pairs **<TermID, DocID'>** must be globally sorted before building the posting lists
    - **non-decreasing TermID order, and for equal values of TermID, non-decreasing DocID' order**

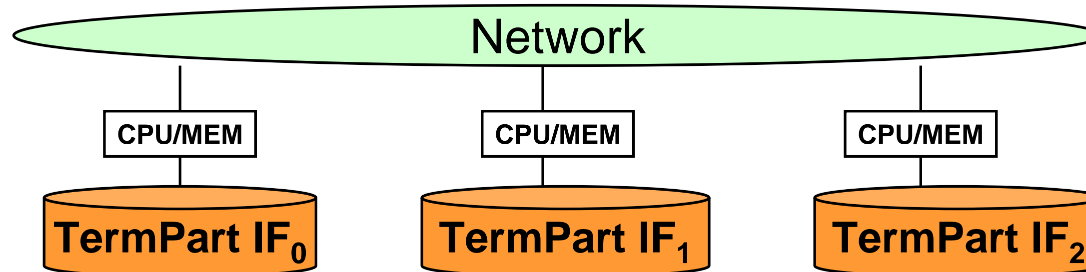
Project assignment



- **Problem to solve**:
  - build in parallel the global IF index, and partition it according to a ***term partitioning*** approach

- Issues:
  - First, each node has to *order* its own *run*, using sort-based if the file is too large
  - DocIDs must be made global (transformed into DocID')
    - **Communication**
  - TermIDs must be partitioned
    - **decision depending on the TermIDs actually present in the global collection**
    - **Communication**
  - Each node has to send part of its run to a given node
    - **All to all communication**
    - **ring based vs. other communication patterns**
    - **Pipelining: messages to the same nodes must partitioned in blocks and sent as soon as a block is filled**
  - Each node receives from multiple node
    - **multi-way merging to generate posting lists**
    - **it can not start the multiway merge till it has received from each other node a portion of their sorted runs**

Project assignment



- **Problem to solve**:
  - build in parallel the global IF index, and partition it according to a ***term partitioning*** approach

- Evaluation:
  - No input is provided
    - **generate synthetic files (ascii or binary) of pairs, with TermID and DocID falling in specific ranges**
  - The tests have to evaluate completion time, by varying
    - **communication patters and primitives used**
    - **message blocking**
    - **methods of merging**
    - **etc. etc.**