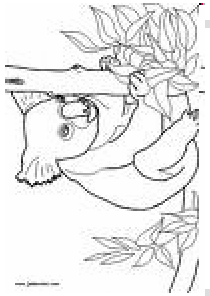
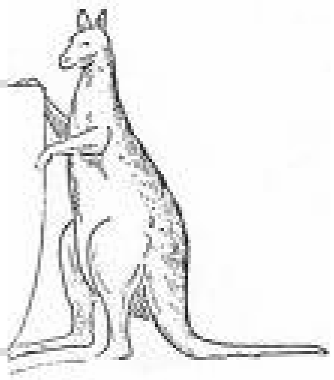


3rd Workshop on Compression, Text, and Algorithms

Melbourne. November 13, 2008



Re-pair compression of inverted indexes

Francisco Claude

Antonio Fariña

Gonzalo Navarro

University of Waterloo

Laboratorio de Bases de Datos

fcfm

Outline

- **I**ntroduction
- **D**ealing with inverted lists
- **R**e-pair and inverted lists
  - Structure
  - Intersection algorithms
- Experimental Results
- Conclusions

Introduction

Text Retrieval Scenario

- Very large text collections of data
  - Need of TR
  - Techniques aiming at:
    - Reducing space needs: compression
    - Improving efficiency of retrieval

- Searches
    - Online searches imply a sequential scan
      - Over compressed text? → **0,3** · |T|
    - Indexed searches.
      - Inverted indexes, suffix arrays, …

Inverted Indexes and their variants (for Nat. Lang.)

Vocabulary	Posting Lists		
SPIRE	0	103	
workshop	147	277	
string	58	159	399
retrieval	92	313	
processing	65	166	406
information	80	302	
Europe	476		
even	486		

Indexed text

Doc1

SPIRE 2008 is the 15th Annual Edition of the Symposium on **string processing** and **information retrieval**. **SPIRE** has its origins in the South American **workshop** on **string processing** which was first held in Belo Horizonte (Brazil, 1993).

Doc2

Starting in 1998, the focus of the **workshop** was broadened to include **information retrieval** due to its increasing relevance and its inter-relationship with the area of **string processing**. In addition, since 2000, SPIRE venue has been in **Europe** in **even** years.

Searches

Word → fetching the posting of the word

Phrase → intersection of posting lists

Space-time trade-off

Granularity:

- Full-positional information of words
- *Doc/Block-addressing*

Compression

- Indexed text (+- 30% ratio)
  - *Huffman, Dense Coding…*
- ***Posting lists!!***

Introduction

Compression of Posting-Lists

- *Compression usually rely on two main features*
  - *Postings lists contain increasing values*
  - *Gaps between them are smaller in the longer lists*

  - *Keep gaps instead of absolute values*
  - *Compress those gaps with a variable-length representation*

	1	2	3	4	5	6	7	8	9	10	11	12	
Original Posting list	4	10	15	25	29	40	46	54	57	70	79	81	
Gaps	4	6	5	10	4	11	6	8	3	13	9	2	
Var-length coded	c4	c6	c5	c10	c4	c11	c6	c8	c3	c13	c9	c2	*Full decompression*

Absolute sampling + Var-length coded

4				29				57				← *Random access*
	c6	c5	c10		c11	c6	c8		c13	c9	c2	*Partial decompression*

Introduction

Posting lists: Compression & Intersection

- *But at the end… it is a typical case of* ***trading space/time***
  - *Space*
  - *Fetching time → a fast <u>decoding</u> algorithm is mandatory*
  - *Intersection time → <u>fast access</u> to the compressed representation is needed*

- *Re-pair*
  - *Is a grammar-based compression technique with:*
    - *Fast decompression.*
    - *Allows fast access to the compressed data (even in 2^{ary} memory)*
    - *Obtains good compression*
  - *We show that… with posting lists*
    - *It gives a competitive space/time tradeoff*

Outline

- **I**ntroduction
- **D**ealing with inverted lists
- **R**e-pair and inverted lists
  - Structure
  - Intersection algorithms
- Experimental Results
- Conclusions

Dealing with inverted lists

Intersection algorithms

- *Intersection of two inverted lists N and M*
  - ***Merge-wise** intersection*
    - *Traversing both lists in parallel*
    - *Best choice if both lists have similar length: |N| <= 20|M|*
    - *Can be done along with decoding*
  - ***Set-vs-set** approach*
    - *The elements of the smallest list are searched for in the longest list.*
    - *Different search options:*
      - *Sequential search*
      - ***Binary search***
      - ***Exponential search***

Requires random access to the longest inverted list

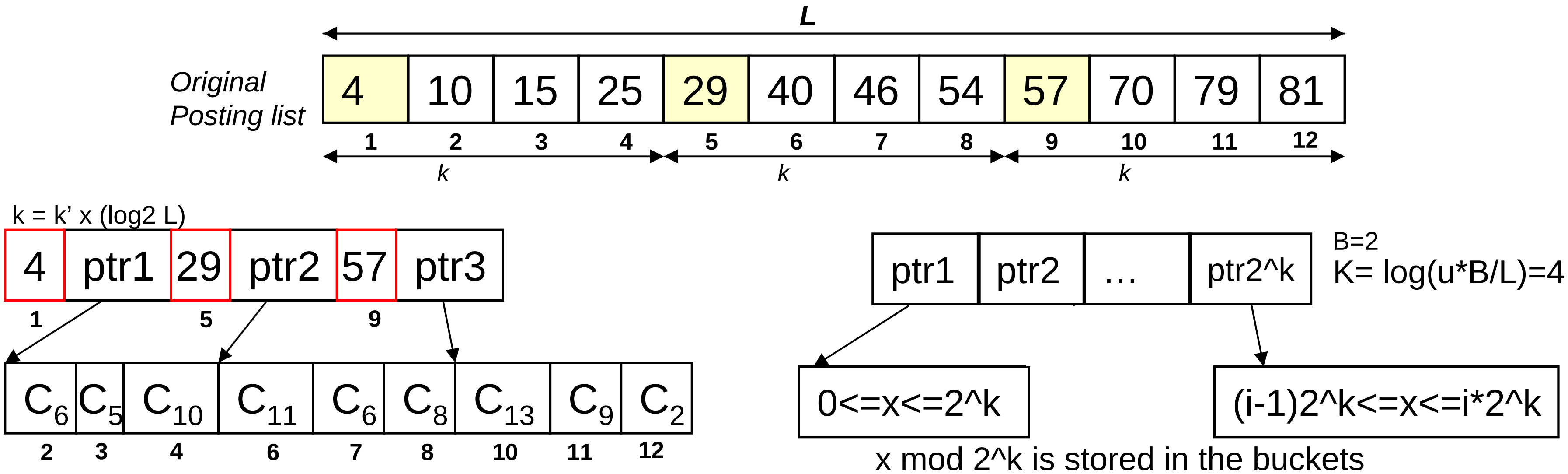
  - *Others…*

Dealing with inverted lists

Data structures

– *Variable-length encoding of gaps (golomb codes, bytecodes,…)*

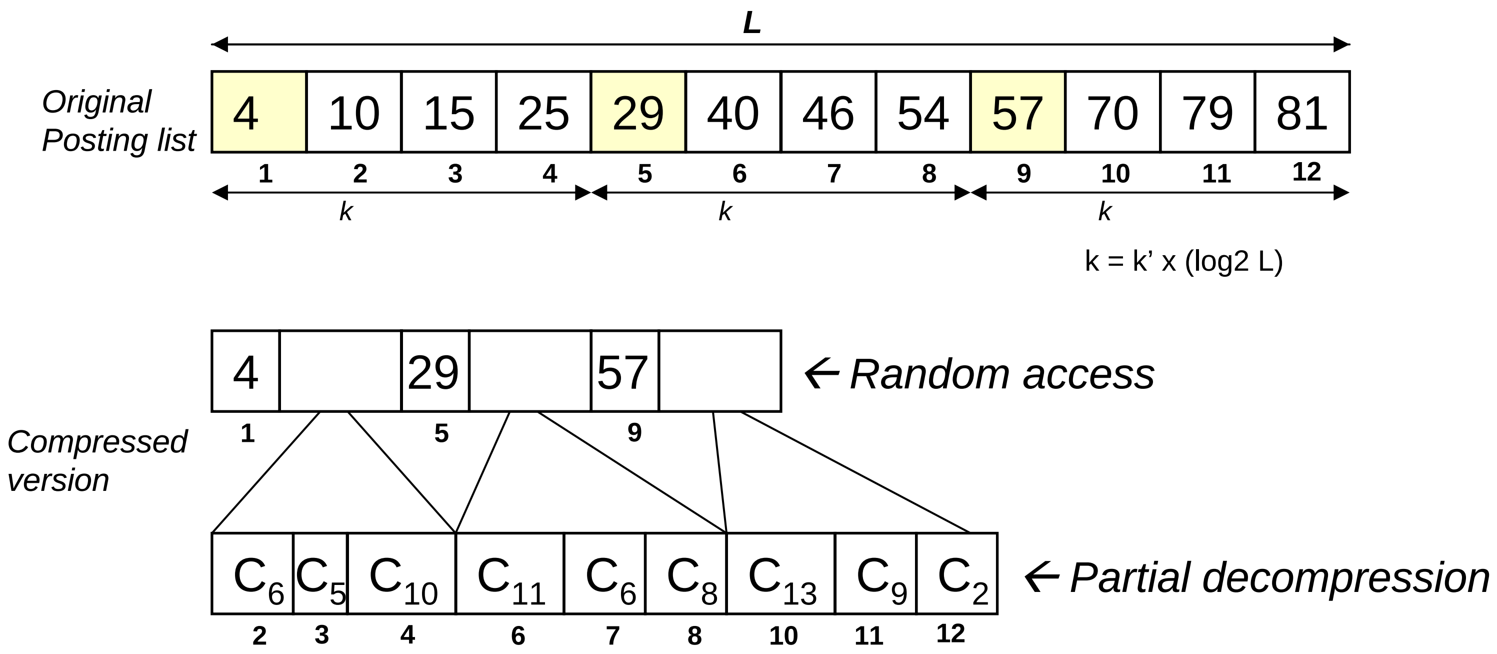
- *merge-type algorithms are still possible*
- *Algorithms that need random access to elements require using a 2-level data structure*
  - *A top-level array indexes the compressed sequence in the bottom level*
- *2 different choices:*
  - ***Sampling at regular intervals** of the list [Moffat-Culpepper '07]*
    *Search is needed in the top-level array → buckets of the same size**
  - ***Sampling regularly at domain values** [Sanders-Trasier '07]*
    Values belong to a bucket depending on their most significative bits → buckets of different size



Introduction

Compression of Posting-Lists + sampling

- *Gap encoding*
- *Variable-length coding of gaps (golomb codes, bytecodes,…)*
  - *Avoiding full decompression → 2-level structure.*
  - *[Moffat&Culpepper'07], [Sanders et al'07] …*



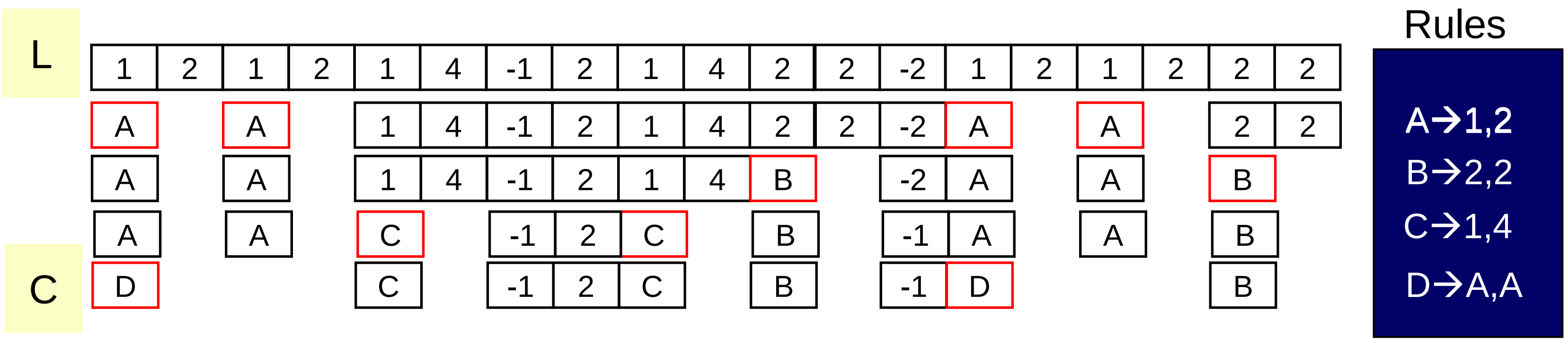
Outline

- **I**ntroduction
- **D**ealing with inverted lists
- **R**e-pair and inverted lists
  - Structure
  - Intersection algorithms
- Experimental Results
- Conclusions

Repair of inverted lists

Repair: The compression algorithm

- Steps:
  - Find the most frequent pair ***ab*** of symbols in L
  - Replace all the occurrences of ab by S in L
    - Add a rule S→ab to a dictionary R (***S*** not appearing before)
  - Iterate until every pair in L appears only once



- The compressed sequence **<u>C</u>** keeps phrases:
  - Of length 1 if the symbol is a terminal one
  - Of length >1 for the new added symbols (phrase)

Repair of inverted lists

Repair: Compressing the dictionary

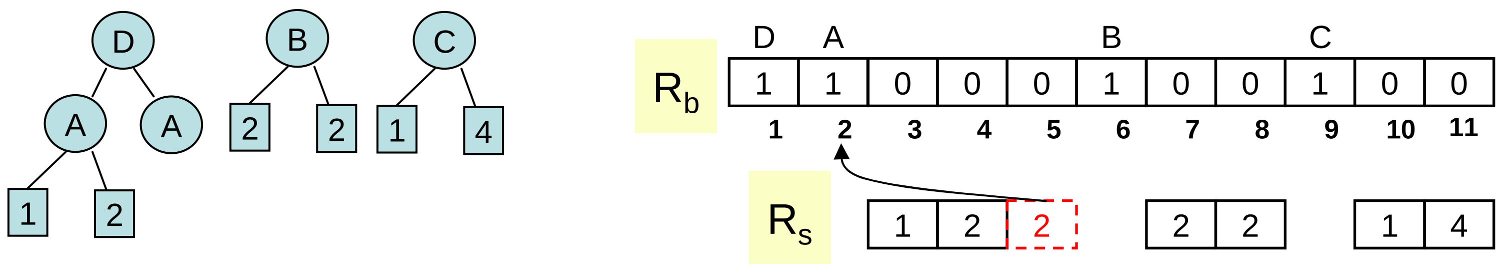
- Rules are represented as a set of trees and each tree as:
  - Rb = Bitmap representing the <u>Tree shape in preorder</u> {0=leaf, 1=internal}
    - The value of the i-th leaf in Rb is found at Rs[rank0(Rb,i)]
    - Non-terminals are shifted by μ to differentiate against terminals (μ = max terminal value)
    - Expanding non-terminals implies→
      - Traversing Rb and extract the leaf values until processing more 0's than 1's
      - Non-terminals are recursively expanded
      - Phrase is expanded in optimal time (time proportional to its length)
  - Rs = Sequence of <u>leaf nodes</u>
    - Non terminals are represented by the starting position of their tree in Rb

A→1,2

B→2,2

C→1,4

D→A,A

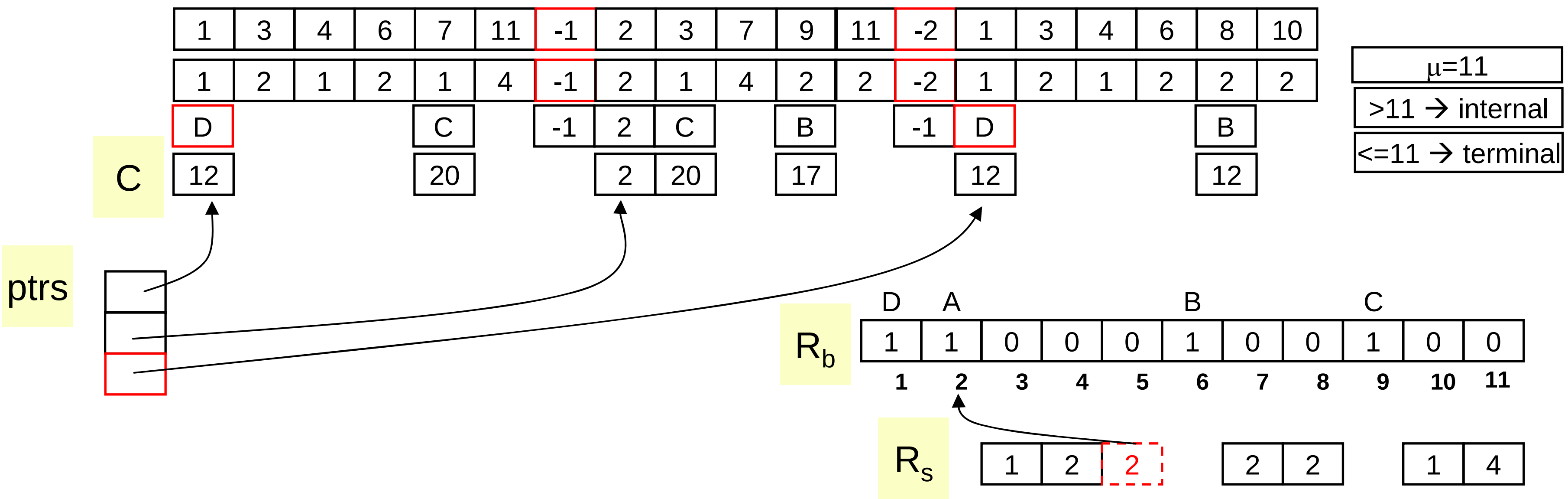


Example: Expanding ("A") → $R_s[Rank_0(R_b,2)]$ $R_s[Rank_0(R_b,3)]$ → 1,2

Repair of inverted lists

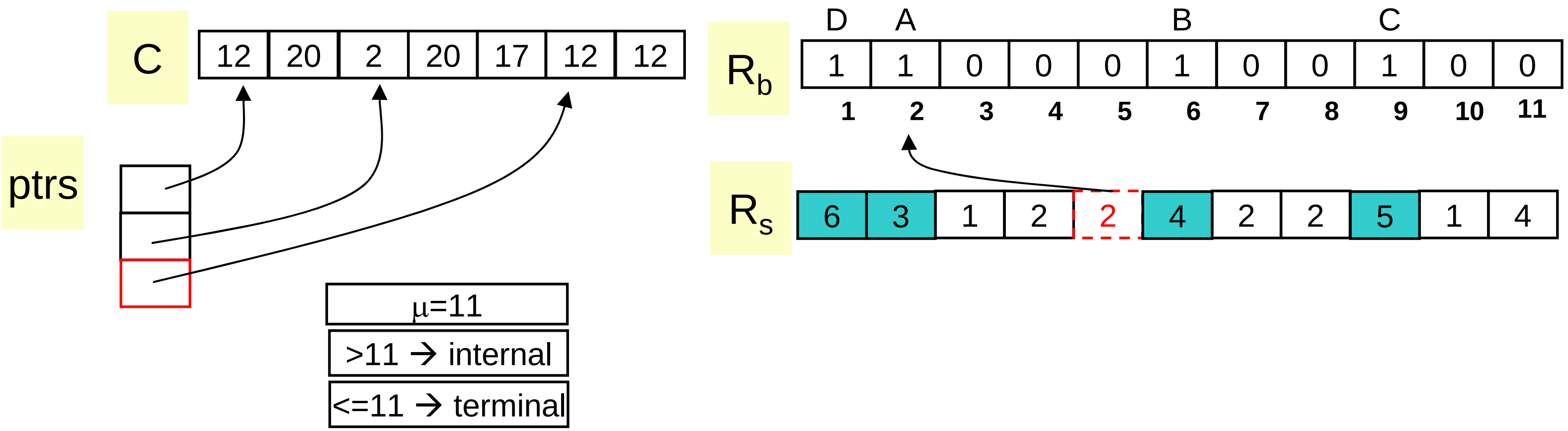
Application to inverted lists

- Differentially encode the inverted lists
  $<p_1, p_2, p_3, \dots p_k> \rightarrow <p_1, p_2\text{-}p_1, p_3\text{-}p_2, \dots p_k\text{-}p_{k-1}>$
- Apply Re-pair to the concatenation of all the lists
  - Ensuring no phrase spans more than 1 list (use of artificial symbol)
  - Storing also Re-pair dictionary
    - Terminal symbols store themselves their differential value
  - Keeping a pointer of each vocabulary entry to its first occ in C



Searching: skipping data

- Some skipping data (for non-terminals) can be added to R_s
  - Avoids expanding some non-terminals
  - Adding data aligned with R_b
    - $Rank_0$ is no longer needed for expanding a symbol
    - So adds some data into R_b but saves "rank structures" → similar space requirements
  - Searching for a given value in a list:
    - Scanning of the list [summing values] until exceeding the value sought
      - If we reach a terminal → we are done (add its value)
      - If we reach a non-terminal → skipping data indicates if it has to be:
        - » expanded (currValue + skip_data > value sought) or,
        - » just skipped (currValue + skip_data < value sought)

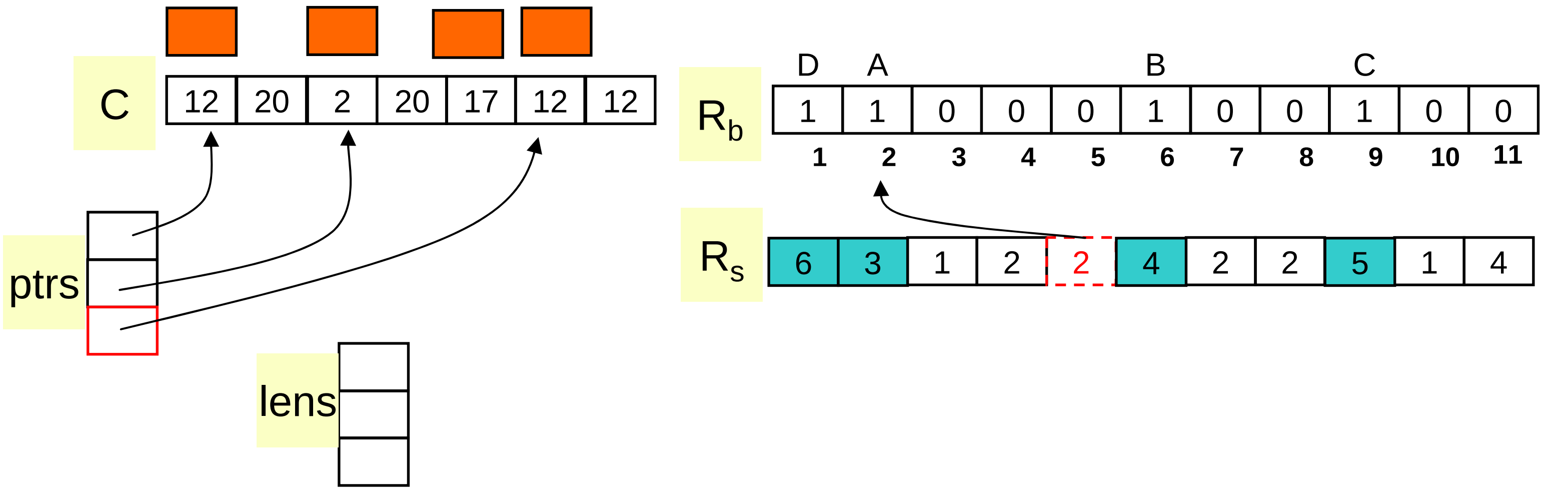


Repair of inverted lists

Searching: intersection algorithms proposed

- Intersection of 2 lists:
  - We sort them by its uncompressed length (so that lenght is also kept)
  - Apply an intersection algorithm (currently svs+seq-search)
    - Using skipping data
      - As show before.
    - Using skipping data PLUS

      sampling at regular intervals in the sequence of phrases (C)



Outline

- **I**ntroduction
- **D**ealing with inverted lists
- **R**e-pair and inverted lists
  - Structure
  - Intersection algorithms
- Experimental Results
- Conclusions

Experimental Results

Framework used

- *Corpus* ***<u>FT91 to FT94</u>****: TREC4*
  - *495Mb English text (indexed in lowercase)*
  - *210,138 documents (2.4Kb on average)*
  - *Doc-oriented index → 50,285,802 entries*

- *Intel Core2duo[T8300]@2.4GHz, 4GB, 3MB-L2cache, Ubuntu, gcc (-O9)*

- *Comparing.*
  - ***Repair vs***
    - ***[Sanders, Trasier, 2007 → lookup structure, parameter B]***
    - ***[Moffat, Culpepper, 2007 → 2-levels, parameter K]***
    - *Merge-wise version*
  - *Bottom-layer → using bytecodes*

- *Showing…*
  - *<u>Space</u> needed by the structures*
  - *<u>Intersection time of 2 lists</u>*

Experimental Results: Espace usage

Len of lists

Ptr to posting lists

Sampling structs

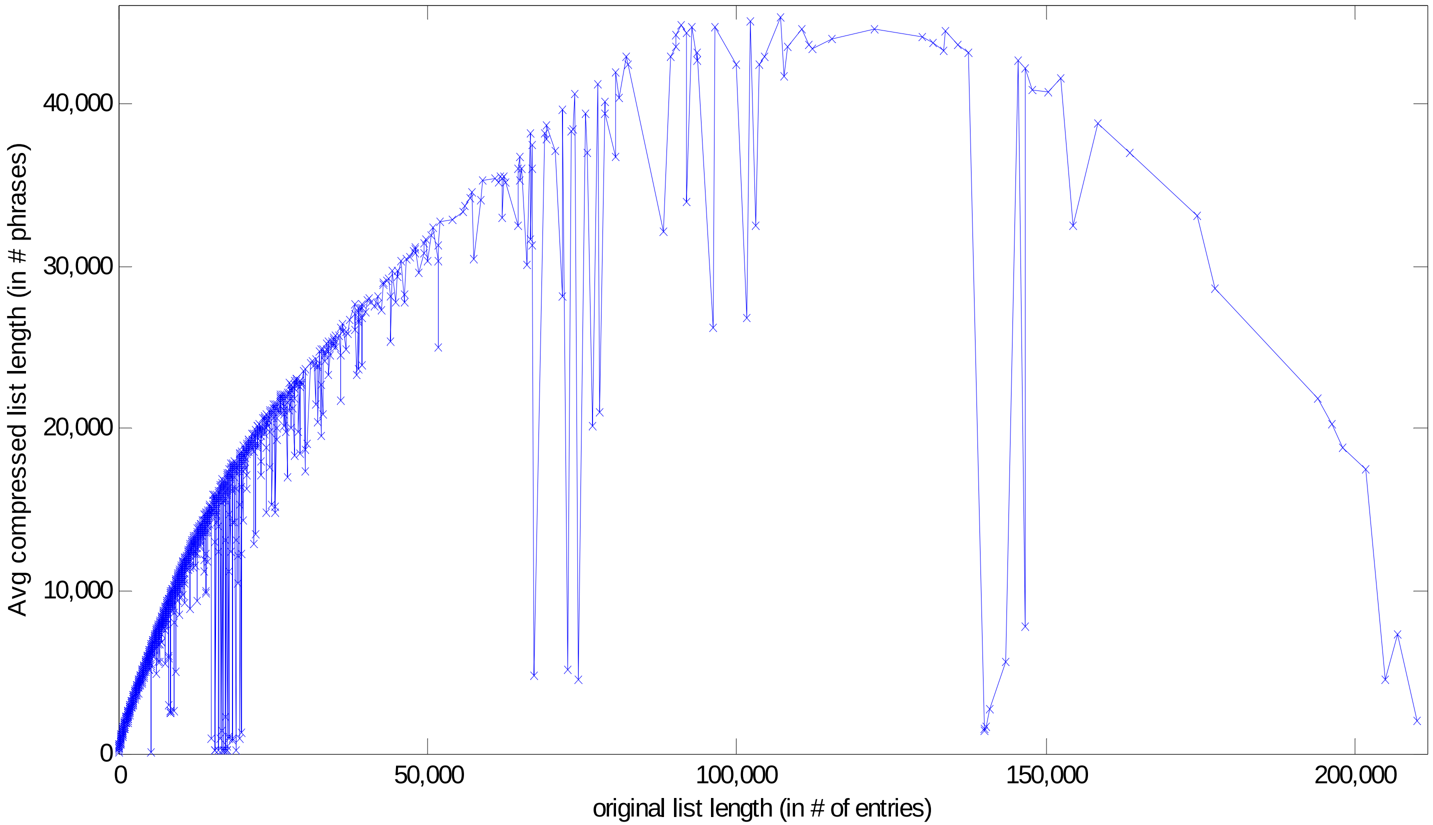
Method	Vocabulary	Extra data	Inverted lists	Total
repair	2,699,656	(dict) 594,467	50,586,760	53,880,883
$(k = 1)$	3,955,308	+ 1,606,640	50,586,760	56,744,691
merge	1,569,568	—	59,406,108	60,975,676
bin,seq $(k = 2)$	4,456,556	8,606,784	55,668,305	68,732,645
$(k = 32)$	4,269,208	1,629,688	57,912,204	63,811,100
lookup $(B = 8)$	4,457,556	8,467,733	58,970,767	71,896,056
$(B = 64)$	4,269,208	1,170,400	59,355,998	64,795,606

-

+

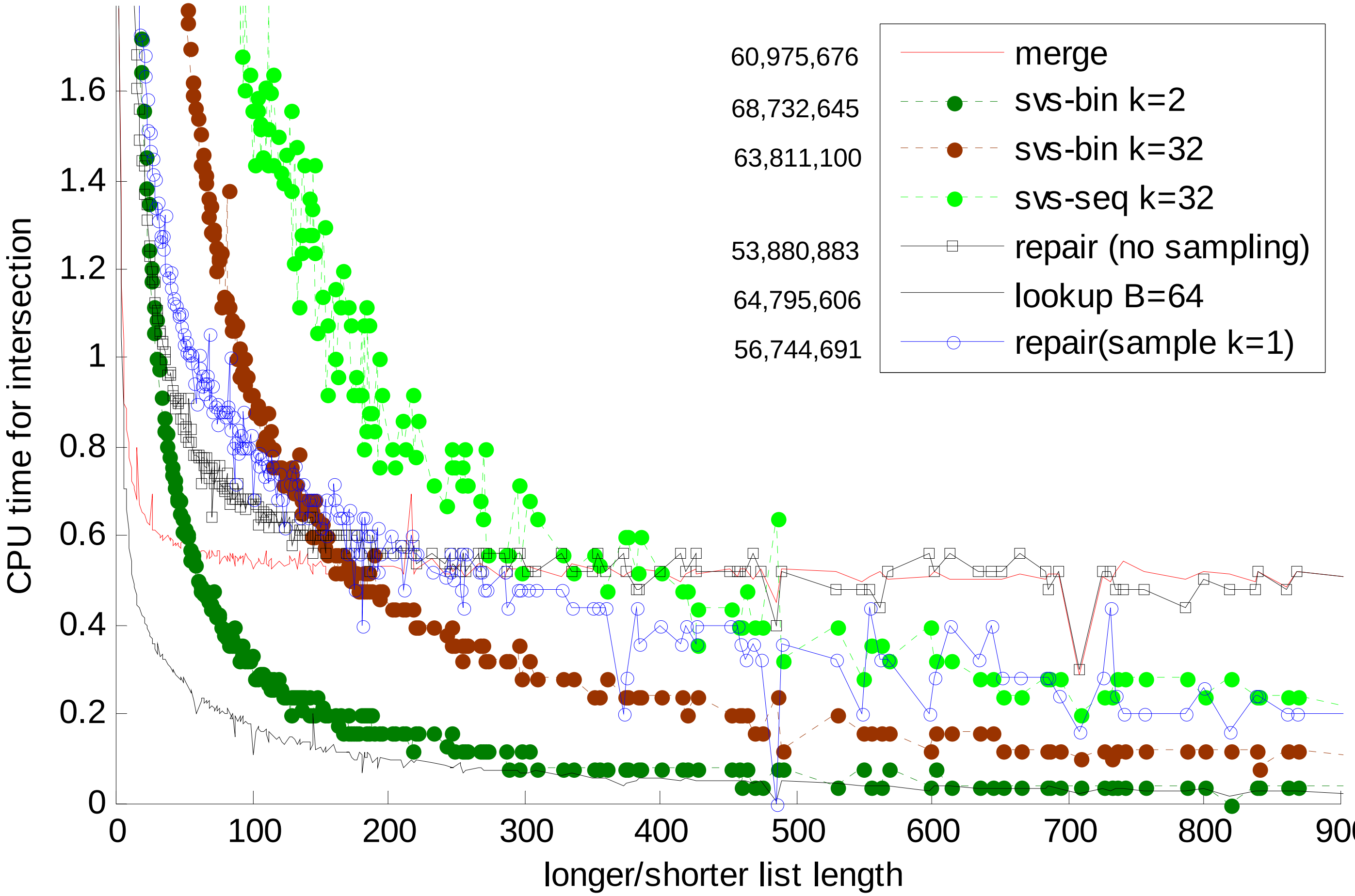
– **Repair with/without sampling (including skipping data)**
  - **Dictionary size is negligible → It fits in RAM**
  - ***Size:***
    - Around 10% of the original text
    - Around 25% of a representation of the inverted lists with 32-bit integers

Experimental Results: curiosities of lists lengths



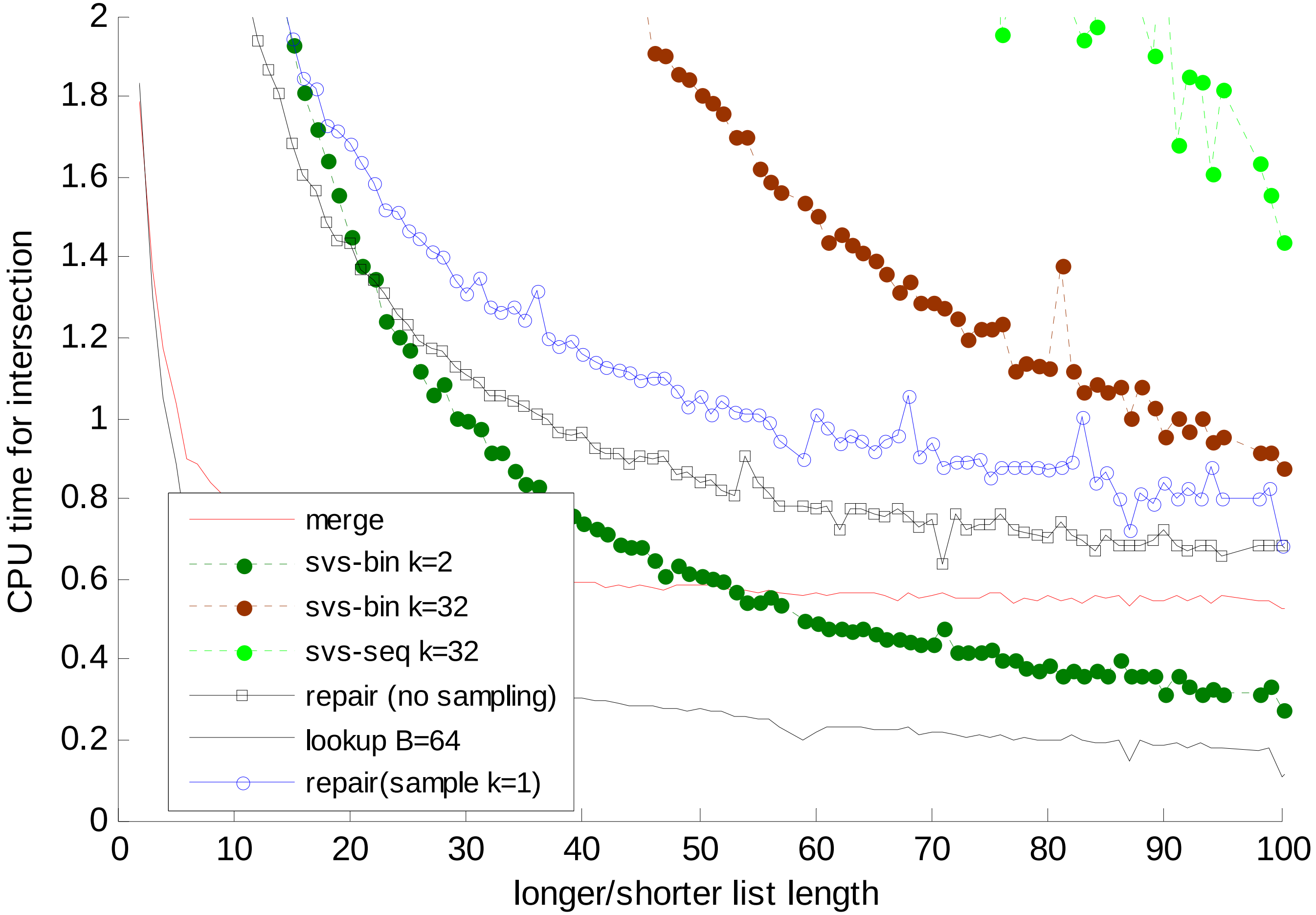
– The longer lists involve much more repetitions → they compress better

Experimental results: List intersection (doc size 2.4kb)



- Lookup → faster but more space
- Merge → good if similar list lengths
- Svs → good choice. Improves results as lengths vary
- Repair→
  - good compression. Sampling good for x>120
  - Performing similar to bc+svs (with large sampling values)

Experimental results: List intersection (zoomed)



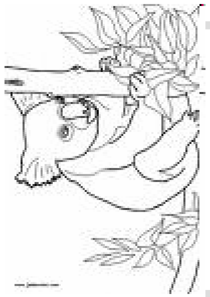
Outline

- **I**ntroduction
- **D**ealing with inverted lists
- **R**e-pair and inverted lists
  - Structure
  - Intersection algorithms
- Experimental Results
- Conclusions

Conclusions

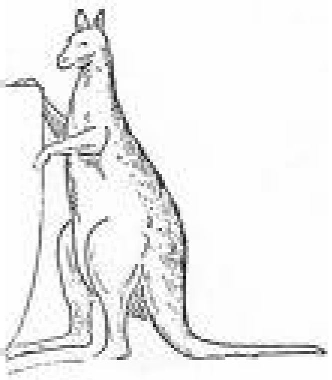
- Re-pair on inverted lists (document-addressing)
  - Better compression than those techniques using bytecodes
    - More space for sampling can be wasted
  - Implicit skipping data
  - Good space/time tradeoff
    - Expecting good performance in 2^{ary} memory
      - Dictionary can be kept in RAM (it is very small)

- Future work
  - Trying other representations/search algorithms:: More experiments!!
  - Dealing with word-addressing indexes
  - New dictionary representation allowing improved descending of the parse tree → important for searches

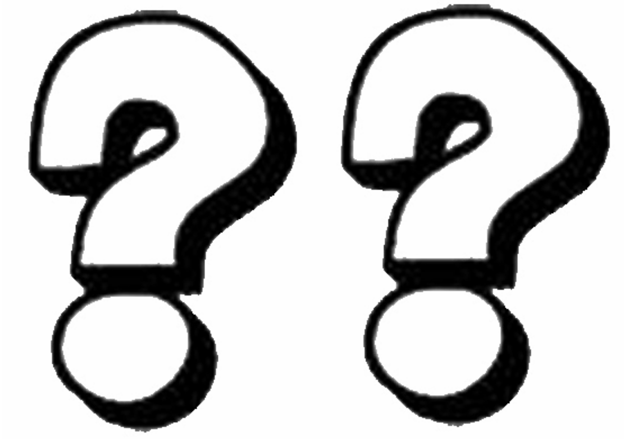


3rd Workshop on Compression, Text, and Algorithms

Melbourne. November 13, 2008



Re-pair compression of inverted indexes



Francisco Claude

Antonio Fariña

Gonzalo Navarro