

# Efficient Indexing of Versioned Document Sequences

Michael Herscovici  
Ronny Lempel  
Sivan Yogev

IBM Haifa Research Lab

## Motivation

- Many information systems save multiple versions of documents
  - Content management systems
  - Version control systems (CVS, CMVC, ClearCase)
  - Wikis
  - Backup and archiving solutions
  - In a sense, e-mail threads
- Searching over such data is possible by naively indexing each version of each document separately
- Goal: exploit the inherent redundancy that is present in the document versions for building more compact indices
  - Not at the expense of any retrieval capabilities, though.

## Talk Outline

- Related Work
- Mechanics of indexing version sequences
- What impacts the index size?
- Optimal alignment of version sequences
- Experimental results
- Additional implementation issues
- Conclusions

3

## Related Work - Stringology

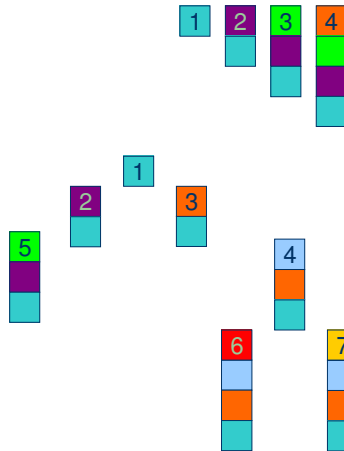
The following are efficiently solvable problems in stringology:

- **Edit/Levenstein distance**: given two strings  $s_1$  and  $s_2$ , find the smallest number of insert/delete/substitute transformations on  $s_1$  that produce  $s_2$
- **Longest common subsequence (LCS)**: given two strings  $s_1$  and  $s_2$ , find their longest common subsequence
- **Example**:  $s_1 = A B C D E F$ ,  $s_2 = A B X E F Y$ 
  - Edit distance is 3:
    - Replace C by X (producing ABXDEF)
    - Erase D (producing ABXEF)
    - Insert Y after F (producing  $s_2$ )
  - LCS is "A B E F"

4

## Related Work – Indexing Shared Content

- Consider a mail thread where each reply or forward of a note doesn't change the replied/forwarded content, but just appends to it (non-interleaving content):
- Regular (linear) threads
  - Each message contains the full text of all previous messages in the thread
- Conjoined (tree-like) thread sets
  - Discussions may split at any point, spinning off sub-threads.
  - Obviously, linear threads are a special and simple instance of a conjoined thread-set.



5

## Related Work – Indexing Shared Content

- Recently published IBM paper: can index each piece of content in the thread (each box) just once, without re-indexing any “quoted” text, producing a much more compact index without losing any retrieval capabilities.
- Idea: “share” the indexed tokens of each node in the tree (each message in the thread) with all nodes beneath it (any downstream message)
- But if the quoted messages are modified, or the added text is interleaved within the quoted text, can't use the method
- Also, indexing method relies on ordering the documents of the thread in a particular order, so method is suitable for batch indexing but not for incremental indexing
- See Broder et al. “Indexing Shared Content in Information Retrieval Systems”, EDBT 2006

6

## Our Problem - Running Example

Assume the following strings (documents):

1. A B C D E F
2. A B X E F Y
3. X C D E F Y
4. Z B X C D F Y

Each symbol represents a token

Each string contains distinct symbols just for ease of presentation

The following is a super-sequence of the strings (not necessarily unique or "optimal"):

Z A B X C D E F Y

7

## Alignment Matrix

- We build a matrix whose first line (line 0) is the super-sequence, with a column per symbol
- Every subsequent line  $j$  is a binary representation of the  $j^{\text{th}}$  string – one can reproduce the  $j^{\text{th}}$  string by taking the symbols of the super-sequence that correspond to the columns having '1' in them.
- As a reminder, these were the strings:
  1. A B C D E F
  2. A B X E F Y
  3. X C D E F Y
  4. Z B X C D F Y

Z	A	B	X	C	D	E	F	Y
0	1	1	0	1	1	1	1	0
0	1	1	1	0	0	1	1	1
0	0	0	1	1	1	1	1	1
1	0	1	1	1	1	0	1	1

8

## Alignment Matrix – Runs of 1

- We now examine the runs of '1' in each column of the matrix.
- Note that some columns contain more than one run of '1's
- Since the matrix has four rows, there are  $1+2+3+4=10$  runs possible: [1:1], [1:2], [2:2], [1:3], [2:3], [3:3], [1:4], [2:4], [3:4], [4:4]
- The above ordering of the runs will be the one we will use – runs are sorted by primarily their end-point, with secondary sort being their start point.

Z	A	B	X	C	D	E	F	Y
0	1	1	0	1	1	1	1	0
0	1	1	1	0	0	1	1	1
0	0	0	1	1	1	1	1	1
1	0	1	1	1	1	0	1	1

[4:4] [1:2] [1:2] [2:4] [1:1] [1:1] [1:3] [1:4] [2:4]  
 [4:4] [3:4] [3:4]

9

## From Runs to Virtual Documents

- So there are 10 possible runs of 1 in this matrix.
- We build a virtual document corresponding to each run of 1
- Virtual document  $[i,j]$  will contain the symbols corresponding to columns containing the run  $[i,j]$

[1:1]	C D	[3:3]	
[1:2]	A B	[1:4]	F
[2:2]		[2:4]	X Y
[1:3]	E	[3:4]	C D
[2:3]		[4:4]	Z B

Z	A	B	X	C	D	E	F	Y
0	1	1	0	1	1	1	1	0
0	1	1	1	0	0	1	1	1
0	0	0	1	1	1	1	1	1
1	0	1	1	1	1	0	1	1

[4:4] [1:2] [1:2] [2:4] [1:1] [1:1] [1:3] [1:4] [2:4]  
 [4:4] [3:4] [3:4]

10

## From Runs to Virtual Documents

- The search engine will index the virtual documents
- Note that the total number of "tokens" to be indexed is equal to the number of runs of 1 in the alignment
- The naïve index will have a number of tokens that simply equals the number of 1s in the matrix

[1:1]	C D	[3:3]	
[1:2]	A B	[1:4]	F
[2:2]		[2:4]	X Y
[1:3]	E	[3:4]	C D
[2:3]		[4:4]	Z B

Z	A	B	X	C	D	E	F	Y
0	1	1	0	1	1	1	1	0
0	1	1	1	0	0	1	1	1
0	0	0	1	1	1	1	1	1
1	0	1	1	1	1	0	1	1

[4:4] [1:2] [1:2] [2:4] [1:1] [1:1] [1:3] [1:4] [2:4]  
[4:4] [3:4] [3:4]

11

## From Virtual Documents to Inverted Index

- We invert the virtual documents, some of which may be empty, in the normal manner
  - It is possible to avoid indexing empty documents, not included in the current work.

[1:1]	C D	[3:3]	
[1:2]	A B	[1:4]	F
[2:2]		[2:4]	X Y
[1:3]	E	[3:4]	C D
[2:3]		[4:4]	Z B

Docs:      1      2      3      4      5      6      7      8      9      10  
C D A B  E   F X Y C D Z B

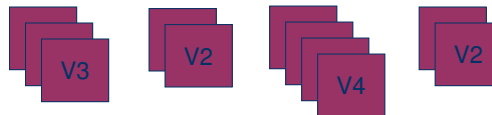
Postings lists:      A → 2                      B → 2, 10                      C → 1, 9  
                             D → 1, 9                      E → 4                              F → 7  
                             X → 8                              Y → 8                              Z → 10

12

## Multiple Versioned Groups

- In practice, the index will include virtual documents from multiple groups of versioned documents.
- Each group will be translated into the virtual document representation that corresponds to the alignment of its documents, as demonstrated before

Four real groups  
with total 11 docs:



22 virtual docs:

6 virtual docs

3 virtual docs

10 virtual docs

3 virtual docs

13

## Auxiliary Predicates per Virtual Doc

Four real groups  
with total 11 docs:



22 virtual docs:

6 virtual docs

3 virtual docs

10 virtual docs

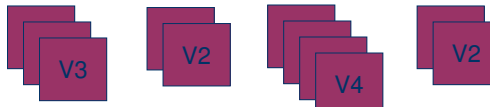
3 virtual docs

- We also need four auxiliary predicates per virtual document id:
  - From(j): the first row of the runs of 1s represented by j
  - To(j): the last row of the run of 1s represented by j
  - Root(j): the docid of the first virtual document in j's group
  - Last(j): the docid of the last virtual document in j's group
- We can calculate the four predicates in  $O(1)$  using two integer arrays (each having an entry per virtual document)

14

## Auxiliary Predicates per Virtual Doc

Four real groups:



22 virtual docs:



- The predicates:

From:	1	1	2	1	2	3	1	1	2	1	1	2	1	2	3	1	2	3	4	1	1	2
To:	1	2	2	3	3	3	1	2	2	1	2	2	3	3	3	4	4	4	4	1	2	2
Root:	1	1	1	1	1	1	7	7	7	10	10	10	10	10	10	10	10	10	10	20	20	20
Last:	6	6	6	6	6	6	9	9	9	19	19	19	19	19	19	19	19	19	19	22	22	22

15

## Auxiliary Predicates per Virtual Doc

From:	1	1	2	1	2	3	1	1	2	1	1	2	1	2	3	1	2	3	4	1	1	2
To:	1	2	2	3	3	3	1	2	2	1	2	2	3	3	3	4	4	4	4	1	2	2
Root:	1	1	1	1	1	1	7	7	7	10	10	10	10	10	10	10	10	10	10	20	20	20
Last:	6	6	6	6	6	6	9	9	9	19	19	19	19	19	19	19	19	19	19	22	22	22

- We can collapse the last two predicates into a single array which holds the root predicate except for the root documents themselves, where it holds the last predicate:

6	1	1	1	1	1	1	9	7	7	19	10	10	10	10	10	10	10	10	10	22	20	20
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

- Since  $from(j) = j - root(j) - to(j) * [to(j) - 1] / 2 + 1$ , we don't need to store the *from* predicate if we have access to the *root* and *to* predicates

16

## Index Representation and Query Evaluation

- The index representation allows easy support for queries such as  $+A +B -C$ , i.e. find (virtual) documents containing all of a required set of terms and none of a forbidden set of terms
- We deal with negated (forbidden) terms by wrapping them with a virtual cursor, that uses the underlying physical cursor to return the next (maximal) interval where the negated term doesn't appear in.
  - Thus, the query above is transformed into  $+A +B +(NegatedCursor(C))$
- High level algorithm:
  - Candidate  $\leftarrow 0$  // the candidate document number for a match
  - Position the iterators of all query terms at the beginning of the postings lists
  - While (Candidate  $\neq \infty$ ):
    1. Candidate  $\leftarrow nextCandidate(candidate)$   
// find document containing all required terms
    2. Score and Output candidate

17

## Primitives on Postings Lists, Predicates and Document Offsets

- The primitives to use on postings lists:
  - A *next(term, doc-num)* primitive, which advances the iterator for *term* to the first document whose number is greater than *doc-num* (and returns that number)
    - If no such next document exists, a value of  $\infty$  is returned
  - A *current(term)* primitive, which returns the current position (virtual document id) of the iterator for *term*.
- In addition:
  - $d \rightarrow root$ ,  $d \rightarrow from$ ,  $d \rightarrow to$  and  $d \rightarrow last$  will denote the root, from and to values corresponding to a virtual document id *d*.
  - We use a function *Location(root, from, to)* that calculates the ID of a virtual document corresponding to the range [from,to] given the beginning root:
    - $Location = root + (from-1) + \frac{1}{2}(to-1)*to$
  - Given two virtual docids  $d_1$  and  $d_2$ , *intersect( $d_1, d_2$ )* returns the docid of the range defined by the intersection of their two ranges, or  $\infty$  if the ranges of  $d_1$  and  $d_2$  do not intersect.

18

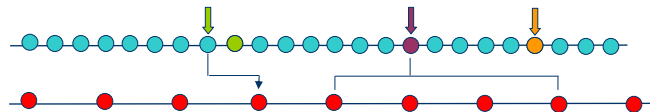
## Example: +A +B -C (Step 1)

- High level algorithm:
  - Candidate  $\leftarrow 0$  // the candidate document number for a match
  - Position the iterators of all query terms at the beginning of the postings lists
  - While (Candidate  $\neq \infty$ ):
    - // find document containing all required terms (some of which may be virtual)
    - 1. Candidate  $\leftarrow$  nextCandidate(candidate)
    - 2. Score and Output candidate
- So how do we find a virtual document satisfying the query whose index is greater than a given value of Candidate?
- We use a zig-zag join procedure on the iterators of A, B and the negation of C
  - We advance lagging cursors to runs (intervals) that overlap with that of the advanced cursor, i.e. to runs that end **at or beyond** where the run of the advanced term **starts**.
  - Basically, we apply simple interval algebra, with caution...

19

## Interval Algebra with Virtual Documents

- Assume the leading cursor is on a virtual document representing an interval [from,to] in some group



- All virtual documents **before** [1,from] of the same group represent intervals that do not intersect with the leading cursor's range (ending at [from-1,from-1])
- All virtual documents in the range [1,from]...[to,to] of the same group represent intervals that surely intersect with the leading's cursor range
- All virtual documents beyond [to,to] will either:
  1. Not intersect at all with the leading cursor's range
  2. Intersect with the suffix of the leading cursor's range
  - Furthermore, if we advance a lagging cursor and it hits a non-intersecting range, it is guaranteed to not intersect with the leading cursor's range later
    - So we can switch the leading cursor

20

## Next Candidate Method

```
NextCandidate(loc) {
  // position the first term beyond the latest document in the range of loc
  d ← next(t1, Location(loc→root, loc→to, loc→to))
  align ← 2 // which is the next term we should align?
  while ( align ≠ n+1 && d ≠ ∞ ) {
    // throw the next term to (or beyond) the beginning of the interesting range
    temp ← next (talign, Location(d→root, 1, d→from) -1)
    // d→from ≤ d→to ≤ temp→to
    if ( temp→root = d→root && temp→from ≤ d→to ) {
      d ← intersection (temp, d) // same root, max from, min to
      align++ // move to align next term
    }
    else { // need to restart - reposition interesting range according to temp
      d ← next(t1, Location(temp→root, 1, temp→from) -1 )
      align ← 2
    }
  }
  return d; // first next (third line) guarantees that loc always advances
}
```

21

## Next Method for Negated Term

- As mentioned, we wrap negated (forbidden) terms with a virtual cursor, that uses the underlying cursor to return the next (maximal) interval where the negated term doesn't appear in.
- Assumptions:
  - The wrapper remembers the *last* position to which the underlying cursor was advanced
  - The next method of the wrapper is always called with a range of the form [X, X]
- Recall that we can identify, for each group, the number of the last physical document in the group (i.e. the largest "to" value of any range in that group)
  - We have that information in the auxiliary predicate tables

22

## Next Method for Negated Term

```
Next( t = -c, loc) {
  // invariant: loc→from equals loc→to
  if ( last ≤ loc):
    last ← next(c, loc)
    target = loc+1
  // we now know that last→to is at or beyond target→to, and target→from=1
  if ( last = ∞ || last → root > target → root ):
    // can return the interval from target→to until the end of the group
    return Location(target→root, target→to, to(target→last) )
  // we now know that the groups of last and target are the same
  if ( last → from > target → to ):
    // the prefix of the target range is legal - return the max interval with that prefix
    return Location(target→root, target→to, last→from-1)
  // we now know that the forbidden term disqualifies the prefix of the target range
  // apply tail recursion
  return next( t, Location(target → root, last→to, last→to))
}
```

23

## Index Size Analysis

Four factors influence the size of the inverted index in our scheme:

1. Lexicon size
  - No change as compared to naïve indexing
2. Number of posting elements
  - This scheme reduces that number from the number of 1s in the alignment matrix to the number of runs of 1 in that matrix
3. Compression of postings lists
  - The use of virtual documents increases the document space and the gaps between postings elements, therefore incurring some overhead as compared with naïve indexing
4. Our scheme also requires the two predicate arrays per virtual document – a little more overhead

24

## Back to the String Alignment Problem

- Index size depends on the sum, over all columns of the alignment matrix, of the number of runs of 1.
- The optimization problem:
  - Given a set of strings, find an alignment matrix whose sum of runs of 1 in its columns is minimal
- The following problems are NP-Hard:
  - “Shortest Common Super-Sequence”: given a set of strings, find the smallest alignment matrix (i.e. the matrix with the fewest columns).
  - “Consecutive Blocks Minimization”: given a set of strings, their super-sequence, and the mapping of each string to the super-sequence, i.e. given a set of binary row-vectors – order them in a matrix so that the number of runs is minimal.

25

## Optimizing the Alignment Matrix

- Let's assume that the string versions were generated serially (no branches).
- Intuition suggests that the rows of the alignment matrix should be ordered by the version creation order.
- The modified optimization problem:
  - Given an ordered set of strings, find an alignment matrix whose sum of runs of 1 in its columns is minimal
- Theorem 1: the following greedy algorithm produces an optimal alignment matrix of an ordered set of strings:
  - Take string #1, and write a row of 1s of the same length in the matrix
  - For all  $j=2, \dots, n$ :
    - Compute the LCS of strings  $j$  and  $j-1$ , inserting new columns into the matrix for all symbols in string  $j$  that are inserted relative to string  $j-1$

26

## Greedy Algorithm example

A	B	C	D	E	F					
1	1	1	1	1	1					

ABXEFY

A	B	C	D	X	E	F	Y			
1	1	1	1	0	1	1	0			
1	1	0	0	1	1	1	1			

XCDEFY

A	B	C	D	X	C	D	E	F	Y	
1	1	1	1	0	0	0	1	1	0	
1	1	0	0	1	0	0	1	1	1	
0	0	0	0	1	1	1	1	1	1	

ZBXCDFY

27

## Greedy Algorithm example

A	B	C	D	Z	B	X	C	D	E	F	Y
1	1	1	1	0	0	0	0	0	1	1	0
1	1	0	0	0	0	1	0	0	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	0	1	1

28

## Optimizing the Alignment Matrix

- Theorem 1: the following greedy algorithm produces an optimal alignment matrix of an ordered set of strings:
  - Take string #1, and write a row of 1s of the same length in the matrix
  - For all  $j=2, \dots, n$ :
    - Compute the LCS of strings  $j$  and  $j-1$ , inserting new columns into the matrix for all symbols in string  $j$  that are inserted relative to string  $j-1$
- Proof sketch: counting the number of runs of 1 by row – every 1 in every row starts a run unless immediately below a 1 in the row above
  - Number of 1s in row  $j$  that can be immediately below 1s in row  $j-1$  is exactly  $\text{LCS}(s_j, s_{j-1})$ , so can't do better than the greedy policy above

29

## Theoretical Justification to Sequentially Ordering the Strings

- We intuitively ordered the strings in the alignment matrix corresponding to the evolution of the sequence of versions. Does that make sense from a theoretical point of view?
- Theorem 2: let there be version sequence of  $n$  strings,  $s_1, \dots, s_n$  such that for all  $j > 1$ ,  $\text{lcs}(s_j, s_{j-1}) \geq \text{lcs}(s_j, s_{j-2}) \geq \dots \geq \text{lcs}(s_j, s_1)$ . Then, aligning the strings in the natural order is optimal.
  - Proof: by induction on the number of sequences (not straightforward)
- The theorem above intuitively means that if the distance from the original version keeps growing, aligning the versions in the order in which they were created is optimal.

30

## Scoring Documents

- So far, we've only discussed how matching documents are identified – not how they are scored
- Assume a virtual document corresponding to interval [from,to] has been identified as relevant
  - Initialize to-from+1 accumulators – one for each physical document in the matching range
  - Set iterators for all terms to virtual document <1, from> of that group, and iterate through all occurrences until virtual document <to, to> of that group
  - Per occurrence of a term in virtual document <i, j> in that range, add the term's weight to the corresponding accumulators
- Once all matching physical documents in a group have been identified, decide which to return:
  - Time dependent: the earlier or latest matching version
  - Score dependent: the highest scoring version
  - Maybe return all the versions

31

## Maintaining Proximity-Based Retrieval

- Search engines associate inner-document locations with each indexed token; these location represent adjacencies of the tokens in the document
  - Enables exact-phrase searching
  - Enables proximity-based scoring (boosting of documents where query terms appear close to each other)
  - Typically, phrase matching and proximity-based scoring do not cross sentence boundaries
- In our scheme, the columns of the alignment matrix are assigned increasing locations
  - The virtual documents associate each column's location with the tokens derived from the runs of 1s in that column
  - However, this means that neighboring words of a physical document may not be assigned consecutive locations, thus disrupting proximity search
- Solution: perform alignment at the sentence level
  - On the one hand, a change in a single word of a sentence will require the re-indexing of the entire sentence in some new virtual document
  - On the other hand, working on sentences means that the alignment phase can run much faster, since the sequences to align become shorter

32

## Experimental Results

- Downloaded two (small) versioned corpora:
  - 222 Wikipedia entries, corresponding to countries
  - MediaWiki PHP source-code classes
- Up to 20 versions of each document set were downloaded
- Indexing was done using Lucene 1.9.1, with documents (real and virtual) tokenized with Lucene's StandardTokenizer
- Two ratios were measured:
  - Alignment ratio: the ratio between the total number of tokens in the virtual documents, and the corresponding number in the original documents
  - Index ratio: the ratio between the size of the Lucene index on the virtual documents and the size of the index on the full documents

33

## Experimental Results

- For both repositories, the compact index was less than 20% the size of the original index
- Other experiments showed a very strong linear correlation between the two ratios, with the index ratio proportional to about 1.15 times the alignment ratio

Repository	#version sets	#docs	#virtual docs	Alignment ratio	Index Ratio
Wikipedia	222	4323	45138	0.0744	0.1332
MediaWiki	142	2055	19144	0.0859	0.1964

34

## Conclusions and Future Work

- Contributions of the work:
  - Tapping multiple sequence alignment for efficient indexing of documents with largely overlapping content
  - Optimizing the alignment for the linear model of version evolution
- Future work:
  - Extend to document version trees (e.g. ClearCase branches, general email threads)
  - The presented method is appropriate for batch indexing. What about incremental indexing?
    - In archiving solutions, lack of incremental capabilities may not be a big deal