

Compressed Data Structures with Relevance

Jeff Vitter

Provost and Executive Vice Chancellor
The University of Kansas, USA

(and collaborators Rahul Shah, Wing-Kai Hon, Roberto Grossi,
Oğuzhan Külekci, Bojian Xu, Manish Patil, Sharma Thankachan,
Sabrina Chandrasekaran, Yu-Feng Chien, Sheng-Yuan Chiu)



Outline

- Generic motivation for Text Indexing and Pattern Search:
To achieve the time and space efficiency of Google (i.e., inverted indexes) but allow more general patterns (as in suffix trees)
- Background on entropy-compressed data structures:
How to work directly on compressed data efficiently.
 - Compressed Suffix Arrays (CSAs)
 - Wavelet tree
- Retrieve the most *relevant* documents
- Achieve efficient performance in *external memory*
- Conclusions and Open Problems

The Attack of Big Data

- Lots of massive data sets being generated
 - Web publishing, bioinformatics, XML, e-mail, satellite data, commerce
 - NASA's Earth Observing System produces Petabytes (10^{15} bytes), soon Exabytes (10^{18} bytes)
 - Yahoo's Hadoop cluster contains 170 petabytes, runs 5M jobs/month
- Data sets are compressible and should be compressed
 - Mobile devices have limited storage available
 - Search engines use DRAM in place of hard disks
 - Next generation cellular phones will charge # bits transmitted
 - There is never enough memory!
 - I/O overhead is reduced
- When the index is external, *minimize I/O!* Examples:
 - Search desktop for phrase in a file.
 - Hum a tune and search iPhone playlist for a match.

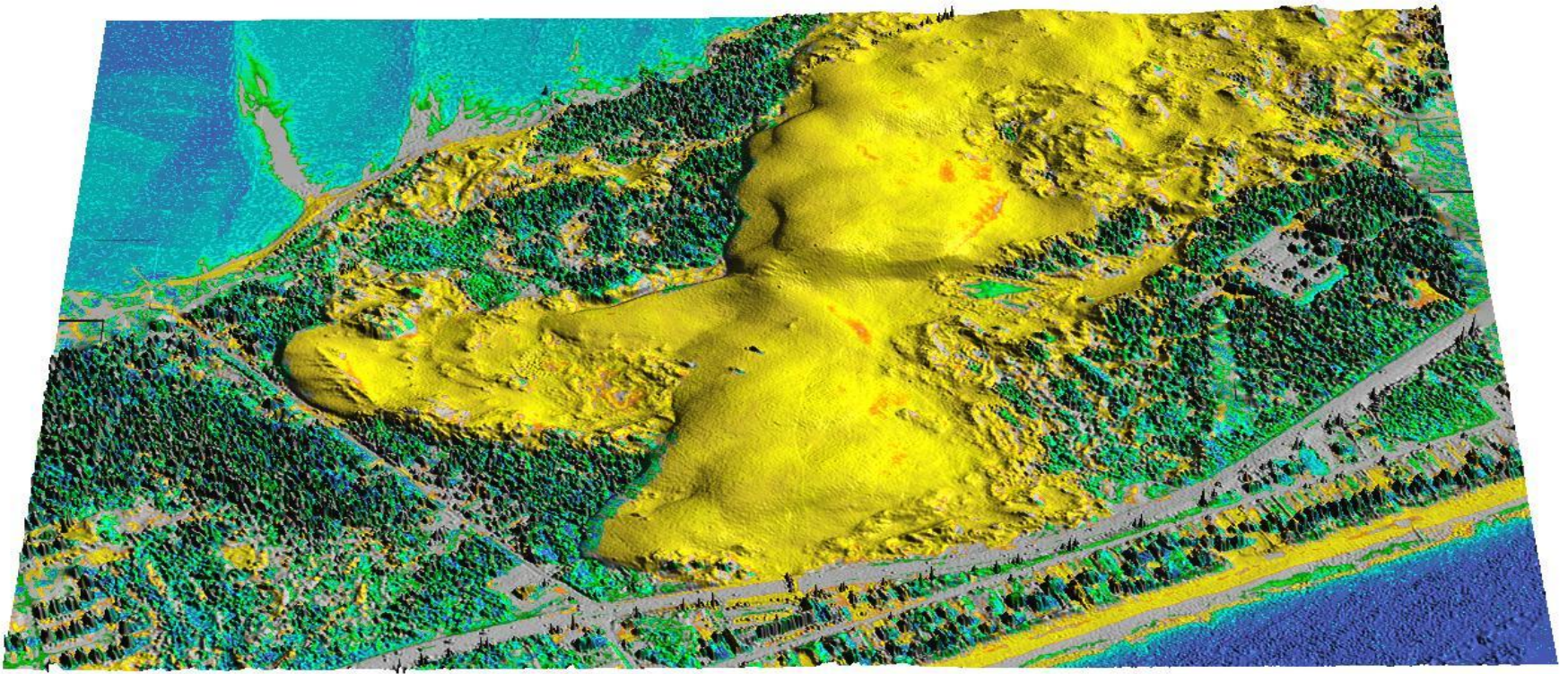
Goal of this talk is . . .

... to design compressed data structures to manage massive data sets

- Use near-minimum amount of space
- Measure space in *data-aware* way, i.e., in terms of each individual data set
- Attain near-optimal query times and I/O bounds
- Focus today is on *relevance* and *external memory*

LIDAR terrain data in external memory

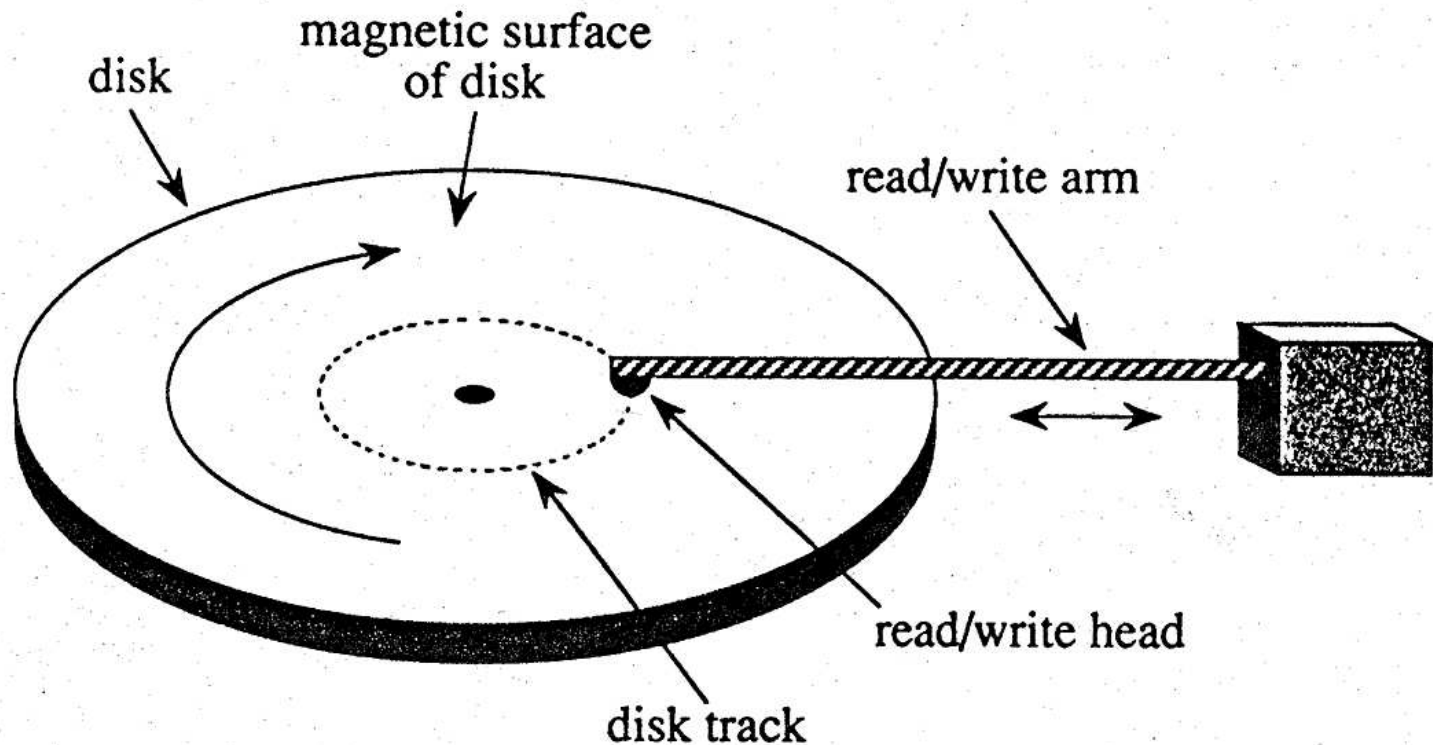
- Problem: Compute how rain flows. Where does it flood?
- Naïve approaches take days to run (and cannot complete for large files).
- Exploiting locality in algorithm design reduces processing time to minutes or hours.



Example: Jockey's ridge (Outer Banks, North Carolina coast, U.S.A.)

Disk Drive Characteristics

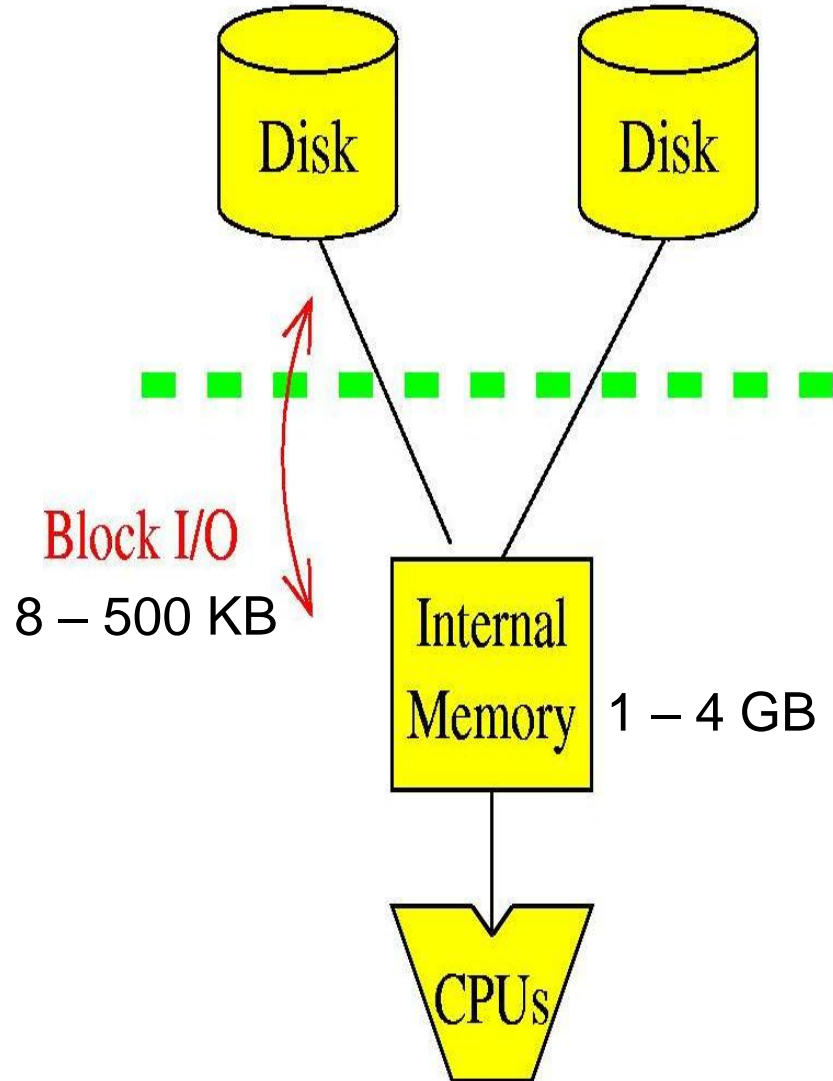
The difference in speed between modern CPUs and disks is analogous to the difference in speed in sharpening a pencil by using a sharpener on one's desk or by taking a hot-air balloon to Siberia and using a sharpener on someone's desk there. -- adapted from Doug Comer



Parallel Disk Model

[Vitter, Shriver STOC90, 94]
[Vitter 08] book for overview

80 GB – 100 TB and more!



N = problem size

M = internal memory size

B = disk block size

D = # independent disks

Scan: $O(N/DB)$ -- “linear”

Sorting: $O((N/DB) \log_{M/B}(N/M))$

Search: $O(\log_{DB} N)$

Full-text Indexing

(where pattern P is arbitrary)

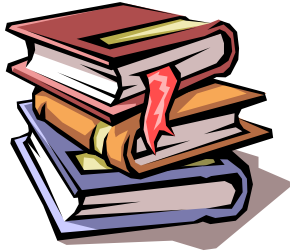
Given a text T of n characters from an alphabet Σ , build an index that can answer the following queries:

For an input pattern P (of length p):

1. **Count** the # locations in T where P occurs;
or
2. **Report** the locations in T where P occurs.

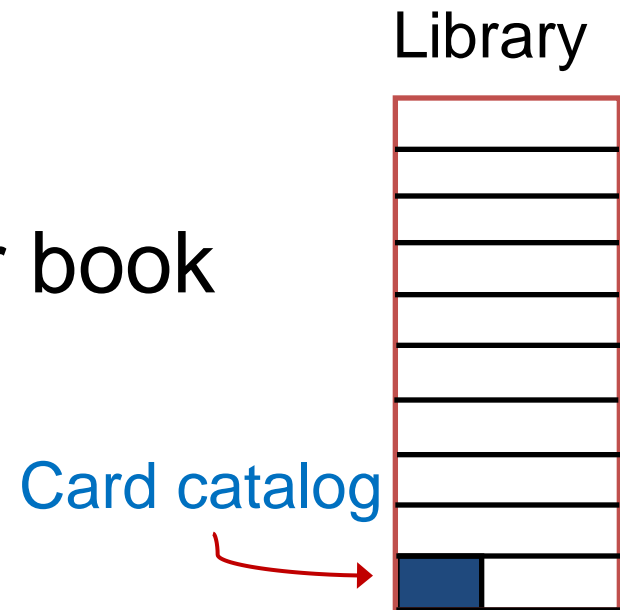
(We will add *relevance* later.)

Analogy to a card catalog in a library



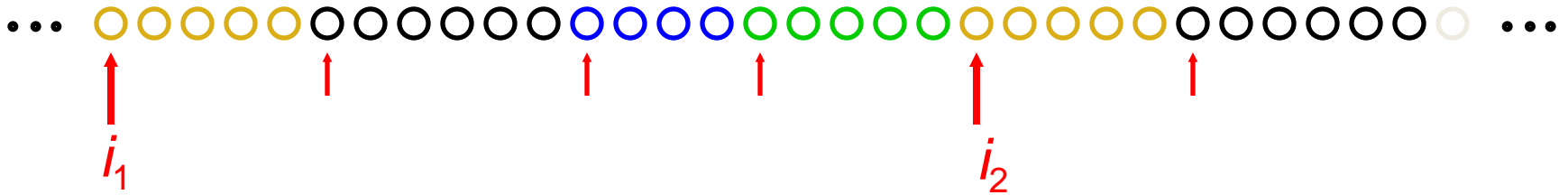
- 10-floor library
- Card catalog near front entrance
- indexes books' titles and authors

- negligible additional space
- a small card (few bytes) per book
- limited search operations!
(only titles and authors)



Word-level indexing (à la Google)

(search for a word using *inverted index*)

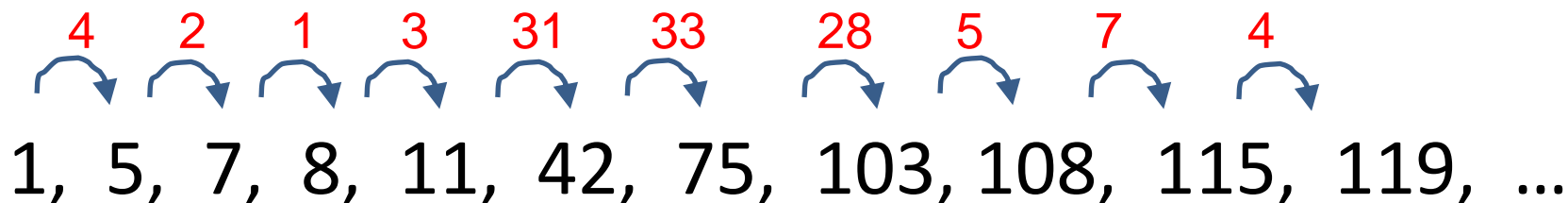


1. Split the text into words.
2. Collect all distinct words in a dictionary.
3. For each word w , store the inverted list of its locations in the text: i_1, i_2, i_3, \dots


$$i_1, i_2, i_3, \dots$$

Can be implemented in practice with $\approx 15\%$ of text space using gap encoding of inverted lists.

Entropy-Compression of Sequences



Encode sequence by encoding the gaps 4, 2, 1, 3, 31, ...

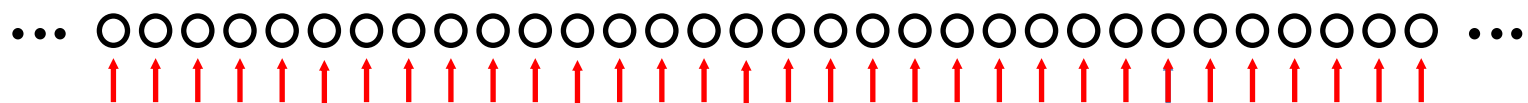
Gap length g can be encoded by delta code in $\approx \log g$ bits, which is roughly the length of g in binary format

Sequence of t items spanning length n takes $\approx t \log (n/t)$ bits
= $t \log (\text{average gap length}) = 10 \log (118/10) \approx 35.6$ bits

This encoding realizes the information-theoretic limit (0^{th} -order entropy) for encoding t numbers in the range $[1, n]$

Inverted Index Can't Do General Search

- Arbitrary phrases not handled by Google (only word search)
- Clear notion of word is not always available:
 - Some Eastern languages
 - unknown structure (e.g., DNA sequences)
- Alphabet Σ , text T of size n bytes (i.e., $n \log |\Sigma|$ bits) :
each text position is the start of a potential occurrence of P



Naive approach: blow-up with $O(n^2)$ words of space

Suffix trees and suffix arrays use $O(n)$ words (i.e., $O(n \log n)$ bits)

Can we do better with linear space $O(n \log |\Sigma|)$ bits?

Or best yet with compressed space $n H_k (1 + o(1))$ bits (where H_k is entropy), which is competitive with inverted indexes?

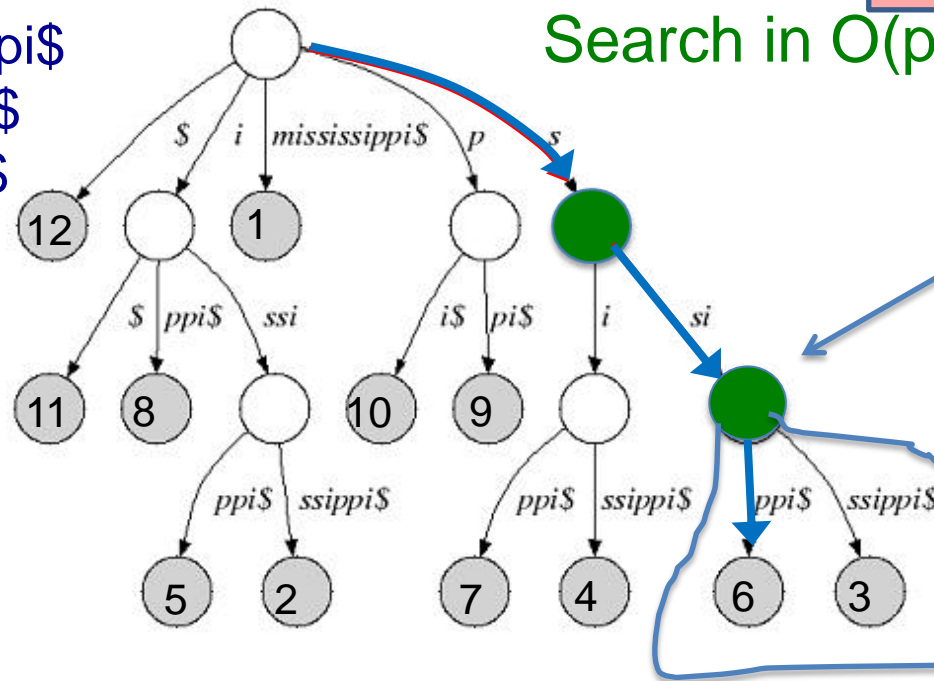
Suffix Tree / Patricia Trie

Text T: mississippi\$

Pattern P = ssi, $|p| = 3$
Search in $O(p)$ time

suffixes

1. mississippi\$
2. ississippi\$
3. ssiissippi\$
4. sissippi\$
5. issippi\$
6. ssippi\$
7. sippi\$
8. ippi\$
9. ppi\$
10. pi\$
11. i\$
12. \$



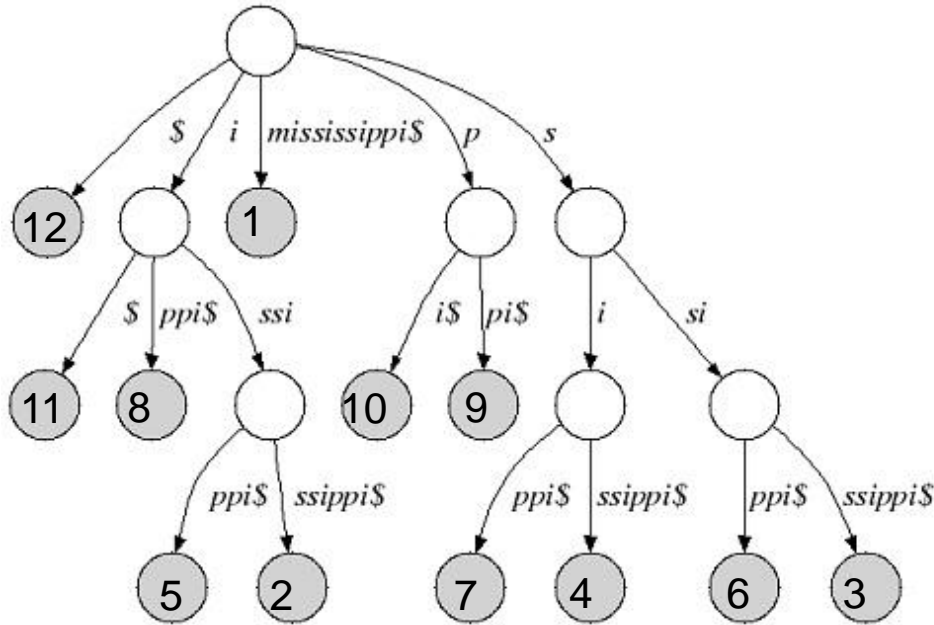
Locus of P

Find the occ
occurrences
in $O(occ)$
time

$O(n)$ words space and optimal $O(p+occ)$ query time

Suffix tree ST / Patricia trie

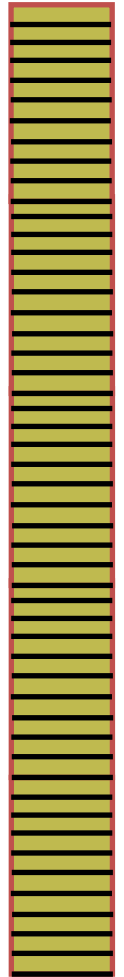
Text T: mississippi\$



Each link and each suffix value requires $O(\log n)$ bits

→ Suffix tree space is $O(n \log n)$ bits
 $\approx 16 \times$ text size in practice.

160
floors



10
floors



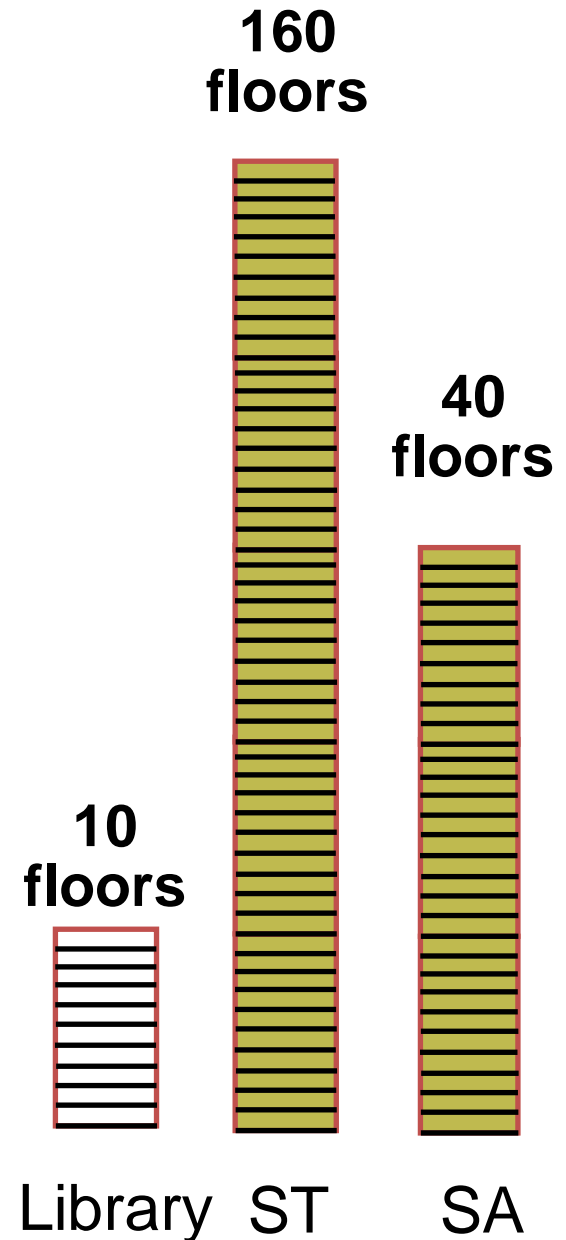
Library

ST

Suffix array SA: sorted list of suffixes

Text T: MISSISSIPPI\$

SUFFIX ARRAY	SORTED SUFFIXES
12	\$
11	I\$
8	IPPI\$
5	ISSIPPI\$
2	ISSISSIPPI\$
1	MISSISSIPPI\$
10	PI\$
9	PPI\$
7	SIPPI\$
4	SISSIPPI\$
6	SSIPPI\$
3	SSISSIPPI\$



Compressed Suffix Array [Grossi & Vitter STOC00] [Grossi, Gupta, Vitter SODA03]

Text T: MISSISSIPPI\$

	SUFFIX ARRAY	SORTED SUFFIXES	NEIGHBOR FUNCTION
1.	12	\$	6
2.	11	I\$	1
3.	8	IPPI\$	8
4.	5	ISSIPPI\$	11
5.	2	ISSISSIPPI\$	12
6.	1	MISSISSIPPI\$	5
7.	10	PI\$	2
8.	9	PPI\$	7
9.	7	SIPPI\$	3
10.	4	SISSIPPI\$	4
11.	6	SSIPPI\$	9
12.	3	SSISSIPPI\$	10

Neighbor function Φ :
 $SA(\Phi[i]) = SA[i] + 1$

We can recreate the entries for the odd-numbered text positions from the even-numbered text positions.

Compressed Suffix Array [Grossi & Vitter STOC00] [Grossi, Gupta, Vitter SODA03]

Text T: ~~M~~ I ~~S~~ S I ~~S~~ S I ~~P~~ P I \$

	SUFFIX ARRAY	SORTED SUFFIXES	NEIGHBOR FUNCTION
1.	12	\$	6
2.	11	I\$	1
3.	8	IPPI\$	8
4.	5	ISSIPPI\$	11
5.	2	ISSISSIPPI\$	12
6.	1	MISSISSIPPI\$	5
7.	10	PI\$	2
8.	9	PPI\$	7
9.	7	SIPPI\$	3
10.	4	SISSIPPI\$	4
11.	6	SSIPPI\$	9
12.	3	SSISSIPPI\$	10

Neighbor function Φ :

$$SA(\Phi[i]) = SA[i] + 1$$

We can recreate the entries for the odd-numbered text positions *from the even-numbered text positions.*

Compressed Suffix Array [Grossi & Vitter STOC00] [Grossi, Gupta, Vitter SODA03]

Text T: ~~M~~ I S S I S S I P P I \$

	SUFFIX ARRAY	SORTED SUFFIXES	NEIGHBOR FUNCTION
1.	12	\$	6
		I\$	1
2.	8	IPPI\$	8
		ISSIPPI\$	11
3.	2	ISSISSIPPI\$	12
		MISSISSIPPI\$	5
4.	10	PI\$	2
		PPI\$	7
		SIPPI\$	3
5.	4	SISSIPPI\$	4
6.	6	SSIPPI\$	9
		SSISSIPPI\$	10

Neighbor function Φ :

$$SA(\Phi[i]) = SA[i] + 1$$

We can recreate the entries for the odd-numbered text positions *from the even-numbered text positions*.

The SA values are all even, so we can “remember” that and cut them in half.

Compressed Suffix Array [Grossi & Vitter STOC00] [Grossi, Gupta, Vitter SODA03]

Text T: ~~M~~ I S S I S S I P P I \$

	SUFFIX ARRAY	SORTED SUFFIXES	NEIGHBOR FUNCTION
1.	6	\$	6
		I\$	1
2.	4	IPPI\$	8
		ISSIPPI\$	11
3.	1	ISSISSIPPI\$	12
		MISSISSIPPI\$	5
4.	5	PI\$	2
		PPI\$	7
		SIPPI\$	3
5.	2	SISSIPPI\$	4
6.	3	SSIPPI\$	9
		SSISSIPPI\$	10

Neighbor function Φ :

$$SA(\Phi[i]) = SA[i] + 1$$

We can recreate the entries for the odd-numbered text positions *from the even-numbered text positions*.

The SA values are all even, so we can “remember” that and cut them in half.

How do we recreate the odd-numbered text positions?

Compressed Suffix Array [Grossi & Vitter STOC00] [Grossi, Gupta, Vitter SODA03]

Text T: ~~M~~ I ~~S~~ S I ~~S~~ S I ~~P~~ P I \$

	SUFFIX ARRAY	SORTED SUFFIXES	NEIGHBOR FUNCTION
1.	12	\$	6
2.	11	I\$	1
3.	8	IPPI\$	8
4.	5	ISSIPPI\$	11
5.	2	ISSISSIPPI\$	12
6.	1	MISSISSIPPI\$	5
7.	10	PI\$	2
8.	9	PPI\$	7
9.	7	SIPPI\$	3
10.	4	SISSIPPI\$	4
11.	6	SSIPPI\$	9
12.	3	SSISSIPPI\$	10

Neighbor function Φ :

$$SA(\Phi[i]) = SA[i] + 1$$

Ex: How to compute $SA[4] = 5$?

$$SA[\Phi[4]] = SA[11] = 6 \quad \text{Voila!}$$

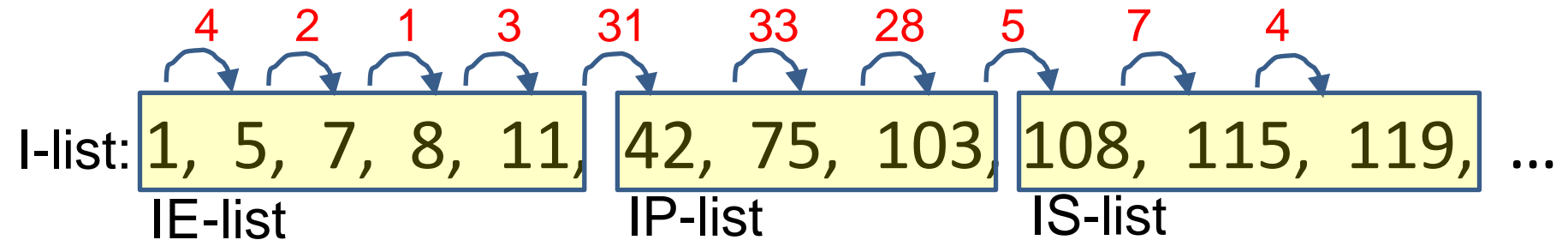
How to compute Φ ?

For each character, let's write the Φ values for its suffixes.

\$:	6	length = 1
I:	1, 8, 11 , 12	length = 4
M:	5	length = 1
P:	2, 7	length = 2
S:	3, 4, 9, 10	length = 4

The 4th smallest neighbor is the 3rd element in the I list: **11**!

Entropy-Compressing Sequences



Encode sequence by encoding the gaps 4, 2, 1, 3, 31, ...

Gap length g can be encoded by delta code in $\approx \log g$ bits, which is roughly the length of g in binary format

Sequence of t items spanning length n takes $\approx t \log (n/t)$ bits
 $= t \log (\text{average gap length}) = 10 \log (118/10) \approx 35.6 \text{ bits}$

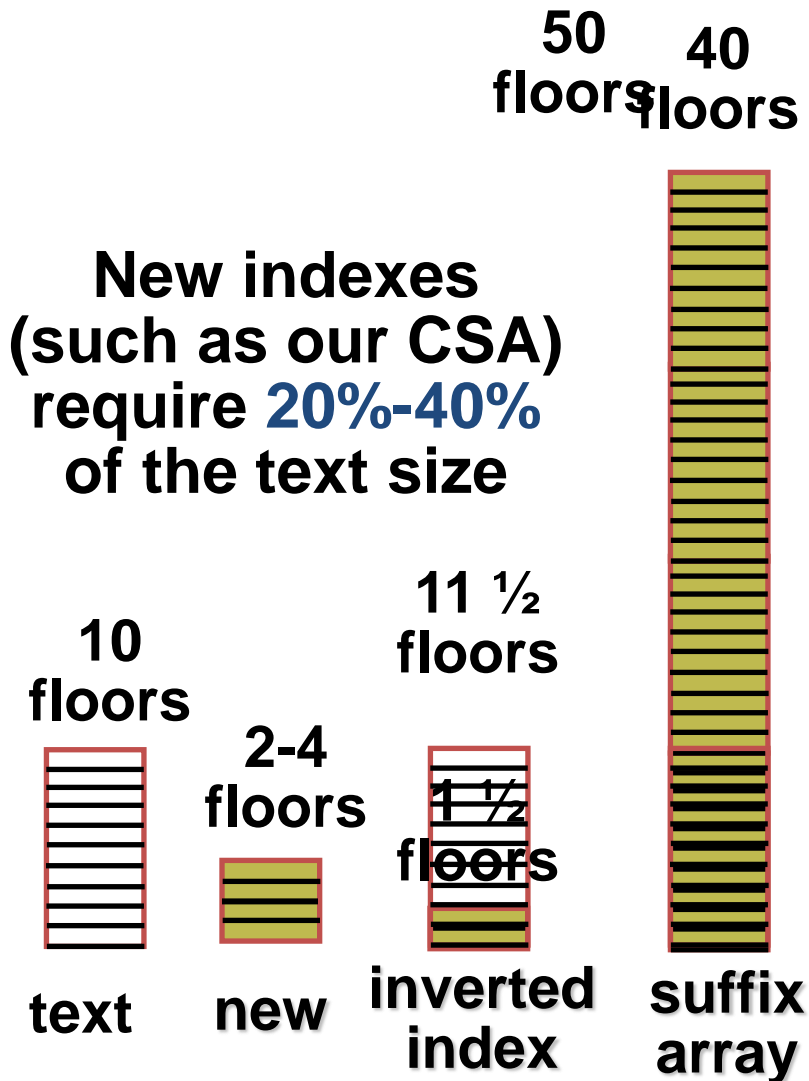
(Hypothetical) Partitioning: $4 \log (10/4) + 3 \log (92/3)$
 $+ 3 \log (16/3) \text{ bits} \approx 27.3 \text{ bits (high-order entropy)}$

Compressed Suffix Array

[Grossi & Vitter STOC00]
[Grossi, Gupta, Vitter
SODA03]

- We recursively halve the compressed suffix array until it is small enough to store explicitly.
- The storage at each level is in high-order entropy-compressed format.
- *Wavelet tree* data structure does necessary computations and in this context achieves high-order entropy.

Compressed Suffix Array [Grossi & Vitter STOC00] [Grossi, Gupta, Vitter SODA03]



- ✓ $O(p + \text{polylog}(n))$ search time.
- ✓ First index with size equal to text size in *entropy-compressed form* ($\sim n H_k$, i.e., w/ mult. constant 1)!
- ✓ Self-indexing text:
no need to keep the text!
- ✓ Any portion of the text can be *decoded* from the index.
- ✓ Decoding is fast and does not require scanning the whole text.
- ✓ Can cut search time further by $\log n$ factor (word size).
- ✓ Similar provable for FM-Index.

Wavelet Tree [Grossi, Gupta, Vitter SODA03]

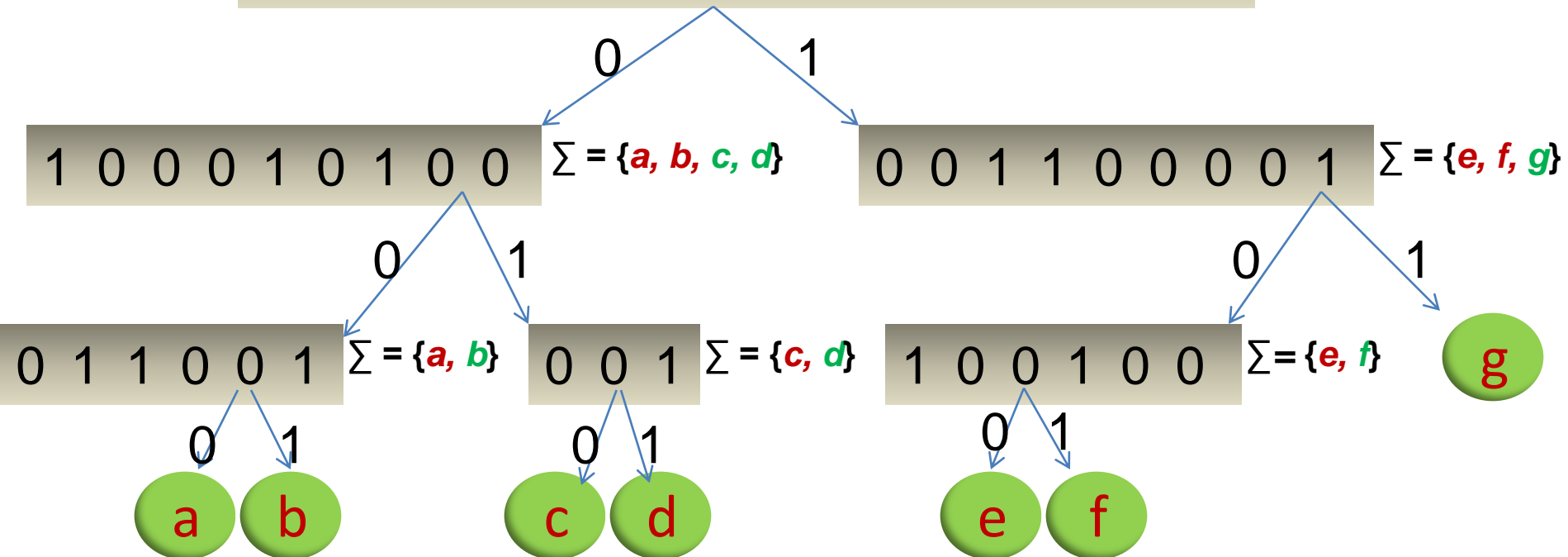
- How to do the gap encoding?
- Neighbor function Φ (in CSA) and Last-to-First function (in FM-index) are closely related
 - $LF(i) = SA^{-1} [SA[i] - 1] = \Phi^{-1}(i)$
- Both computed elegantly by Wavelet Tree [GGV03]
 - Stores a text (e.g., BWT) in $O(n \log |\Sigma|)$ bits
 - Supports rank/select & 2D range search in $O(\log |\Sigma|)$ time
 - Can be 0-th order entropy compressed via RLE (gap)
 - When used w/ CSA or BWT \rightarrow higher-order compression!

Wavelet Tree [Grossi, Gupta, Vitter SODA03]

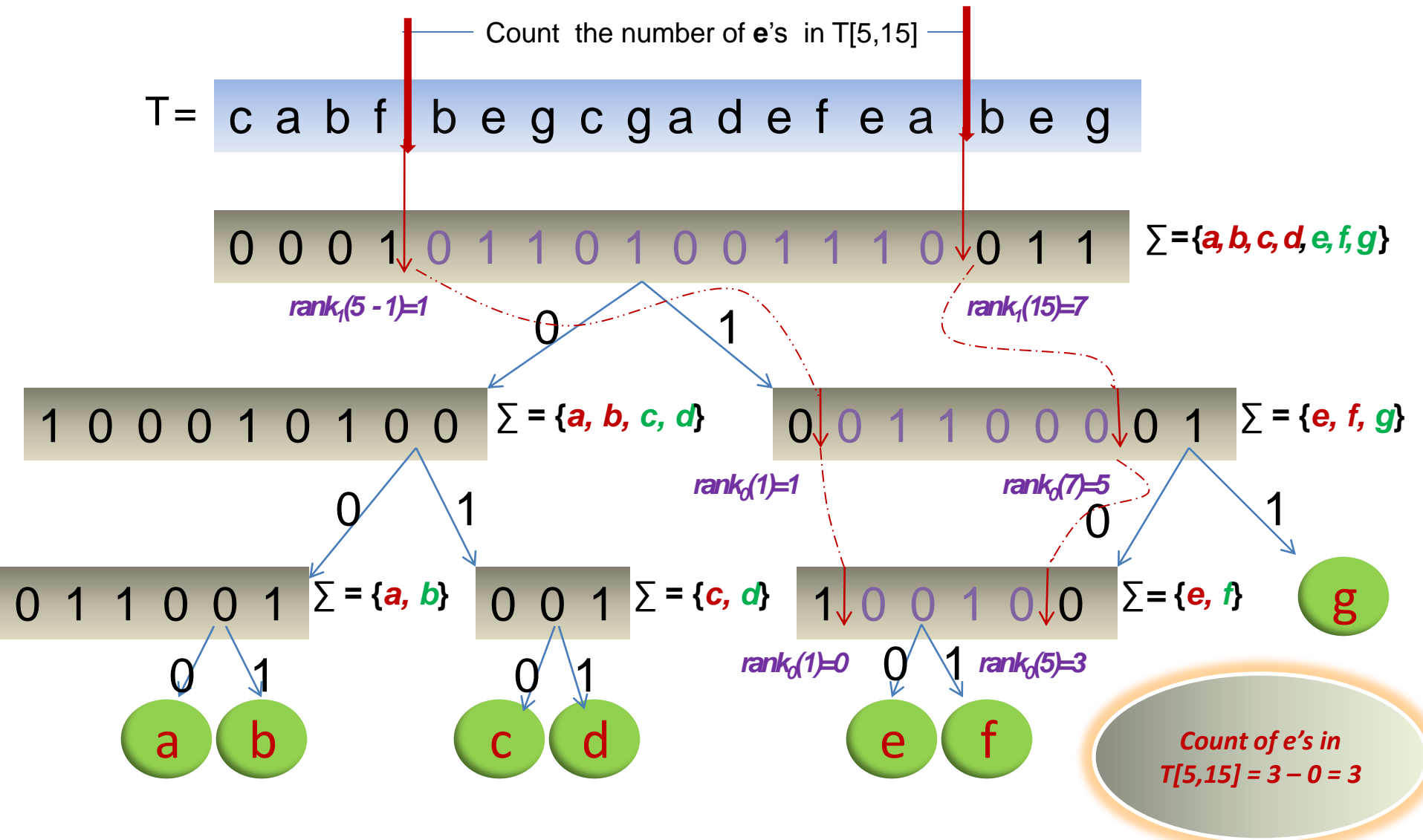
$T =$ c a b f b e g c g a d e f e a b e g

0 0 0 1 0 1 1 0 1 0 0 1 1 1 0 0 1 1

$\Sigma = \{a, b, c, d, e, f, g\}$



Wavelet Tree [Grossi, Gupta, Vitter SODA03]



Two Challenges

Our goal is to realize the advantages of inverted indexes but allow more general search capability.

Challenges discussed today:

1. Building relevance into queries
(output top-k answers)
2. External memory performance

Note: Compressed suffix array (CSA) and FM-index
access memory randomly and do not exploit locality
and thus have poor I/O performance!

Document Indexing

In a collection of text strings (*documents*) d_1, d_2, \dots, d_D of total length n ,
search for query pattern P (of length p).

- Output the IDs for the documents that contain pattern P .
- Issue: # documents output might be **much** smaller than the total number of pattern occurrences, so going through all occurrences can be too costly.
- Muthukrishnan: **$O(n)$** words of space, answers queries in optimal **$O(p + \text{output})$** time.
- Succinct version by Sadakane and by Valimaki & Makinen.

Modified Problem—using Relevance

- Instead of listing all documents (strings) where pattern occurs, list only highly “relevant” documents.
 - Importance: where each document has a static weight (e.g., Google’s PageRank).
 - Frequency: where pattern P occurs most frequently.
 - Proximity: where two occurrences of P are close to each other.
- Threshold K vs. Top-k
 - Threshold K: Retrieve matching documents with score $\geq K$
 - Top-k: Retrieve only the k most-relevant documents.
 - More intuitive for User

Approaches

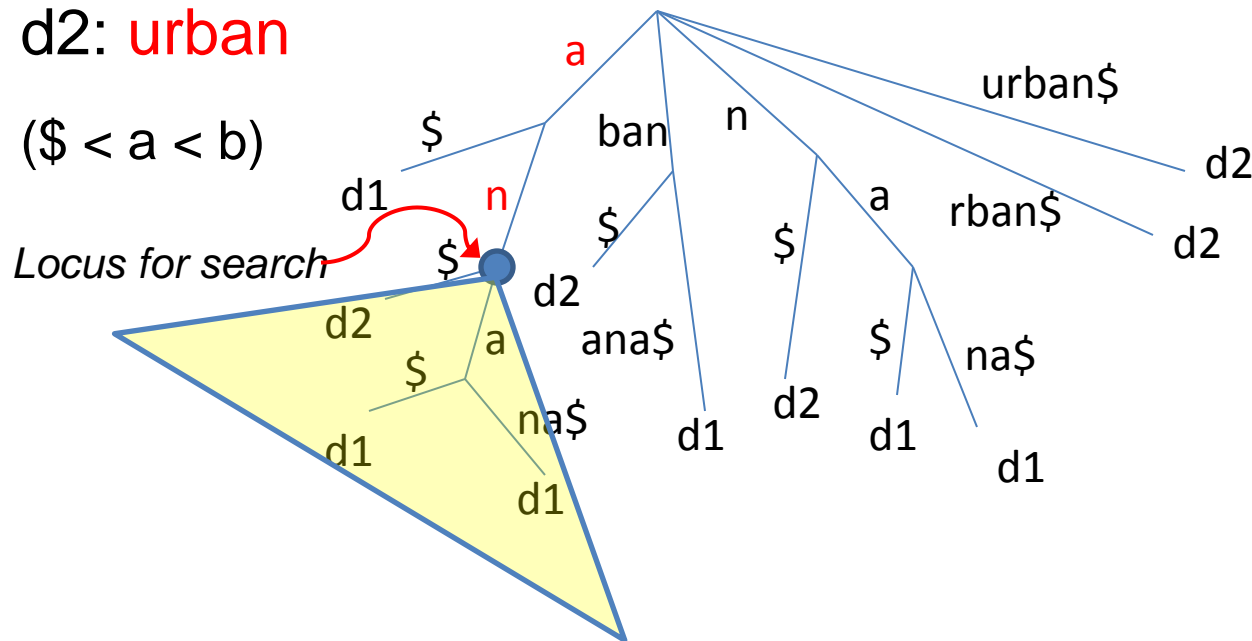
- Inverted Index
 - Popular in IR community.
 - Does not efficiently answer arbitrary pattern queries.
 - Slower
- Muthukrishnan's Structure (based on suffix trees)
 - Takes $O(n \log n)$ words of space for threshold queries while answering queries in $O(p + \text{output})$ time.
 - Top-k queries require additional overhead.
- Let's first improve to $O(n)$ words of space.

Suffix tree based solutions

d1: **banana**

d2: **urban**

(\$ < a < b)

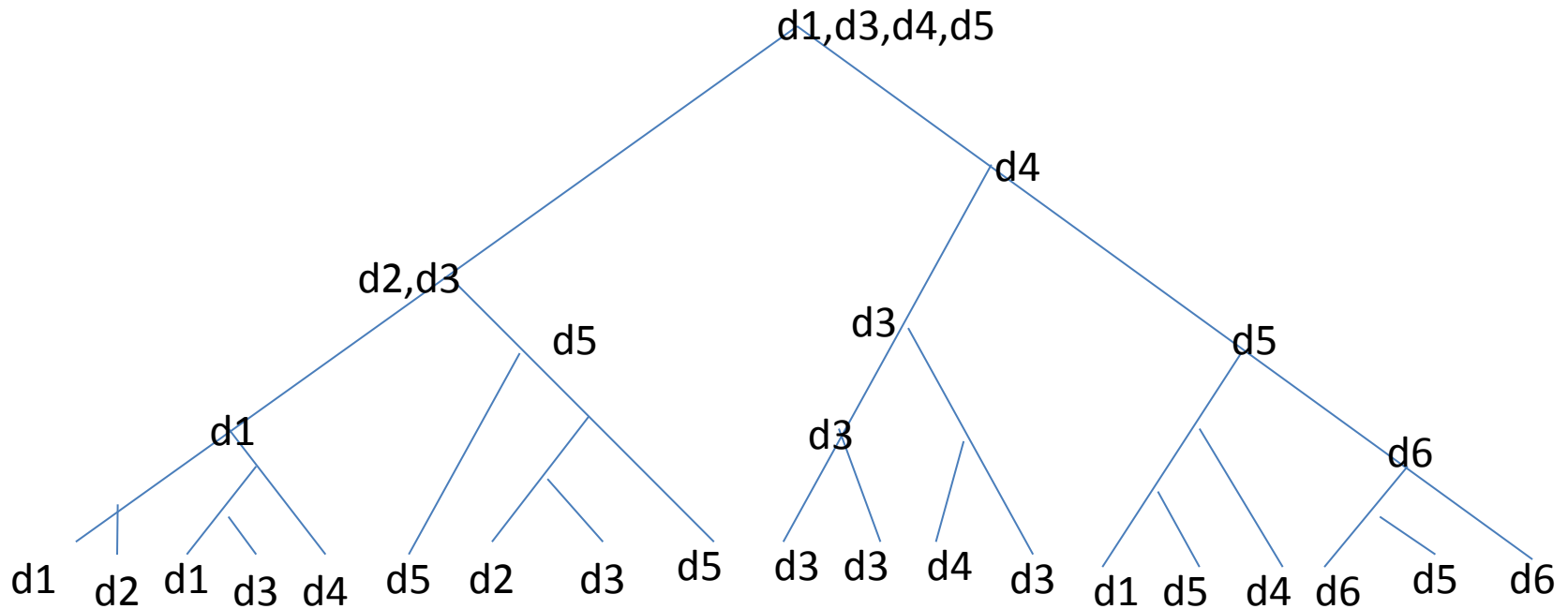


Suffixes:

a\$
an\$
ana\$
anana\$
ban\$
banana\$
n\$
na\$
nana\$
rban\$
urban\$

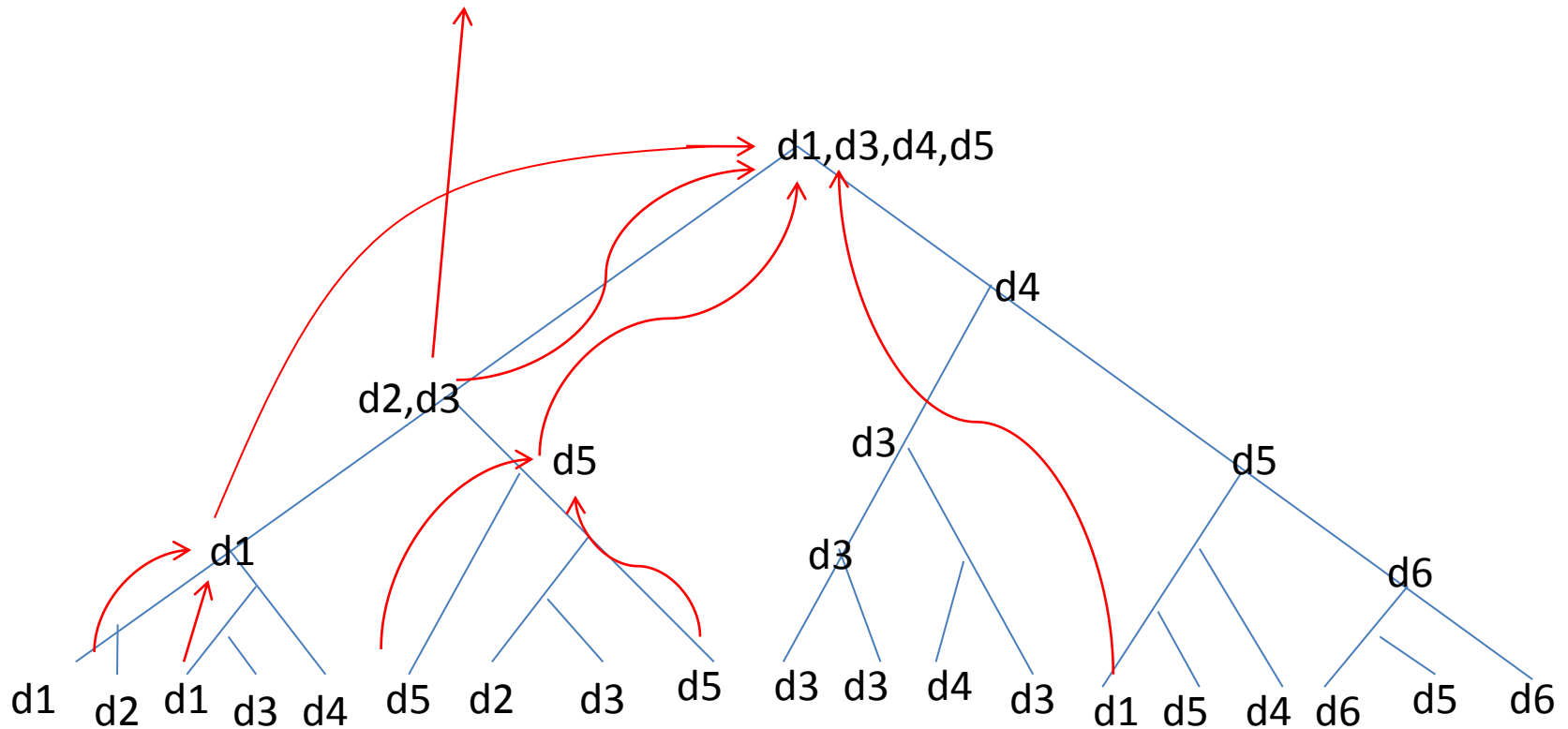
- Example: Search for pattern “**an**”
- We look at the node’s subtree:
Output d1 (in which “**an**” appears twice) and d2.

Tree-link Structure: Focs 2009



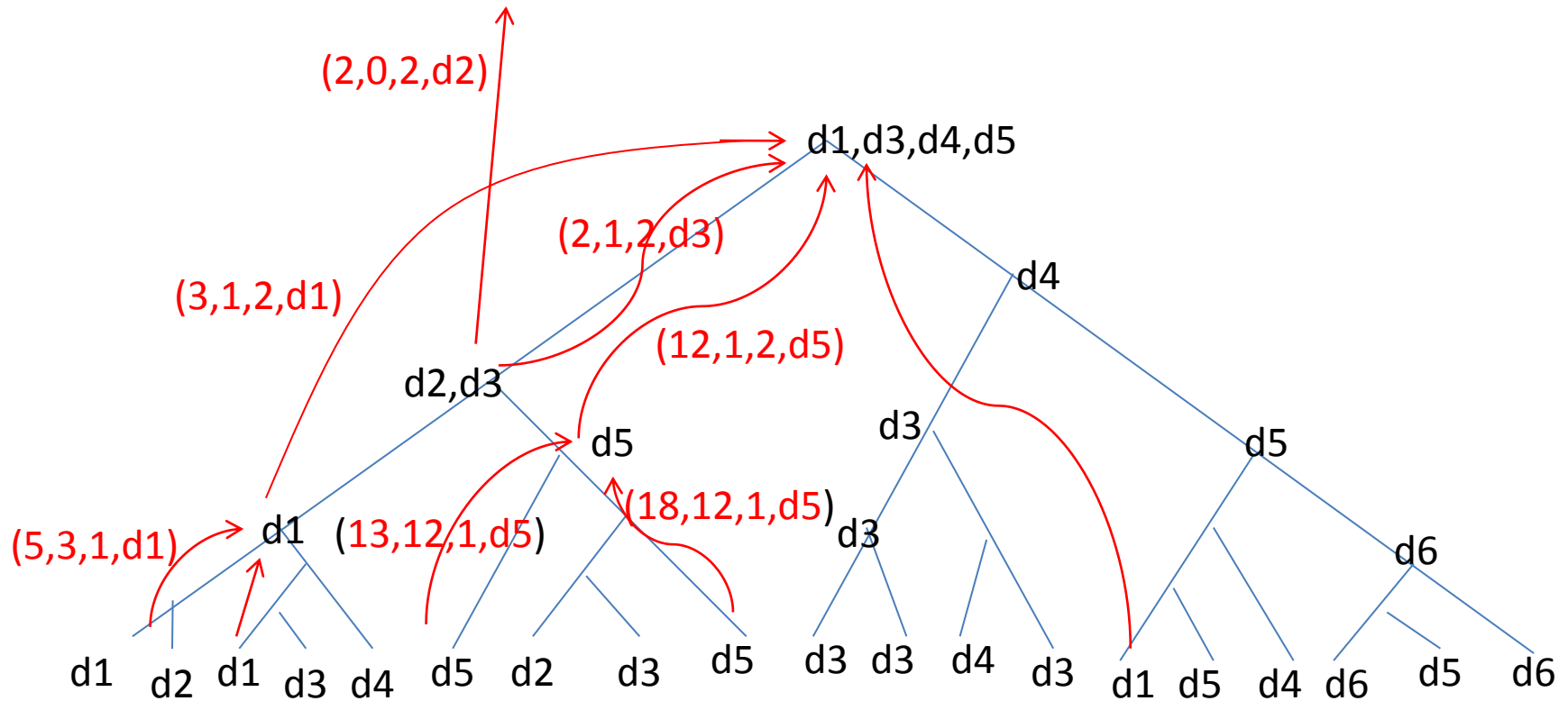
- At each node v , store the ID for document d_i if at least two subtrees of v contain d_i .

Tree-link Structure: Focs 2009



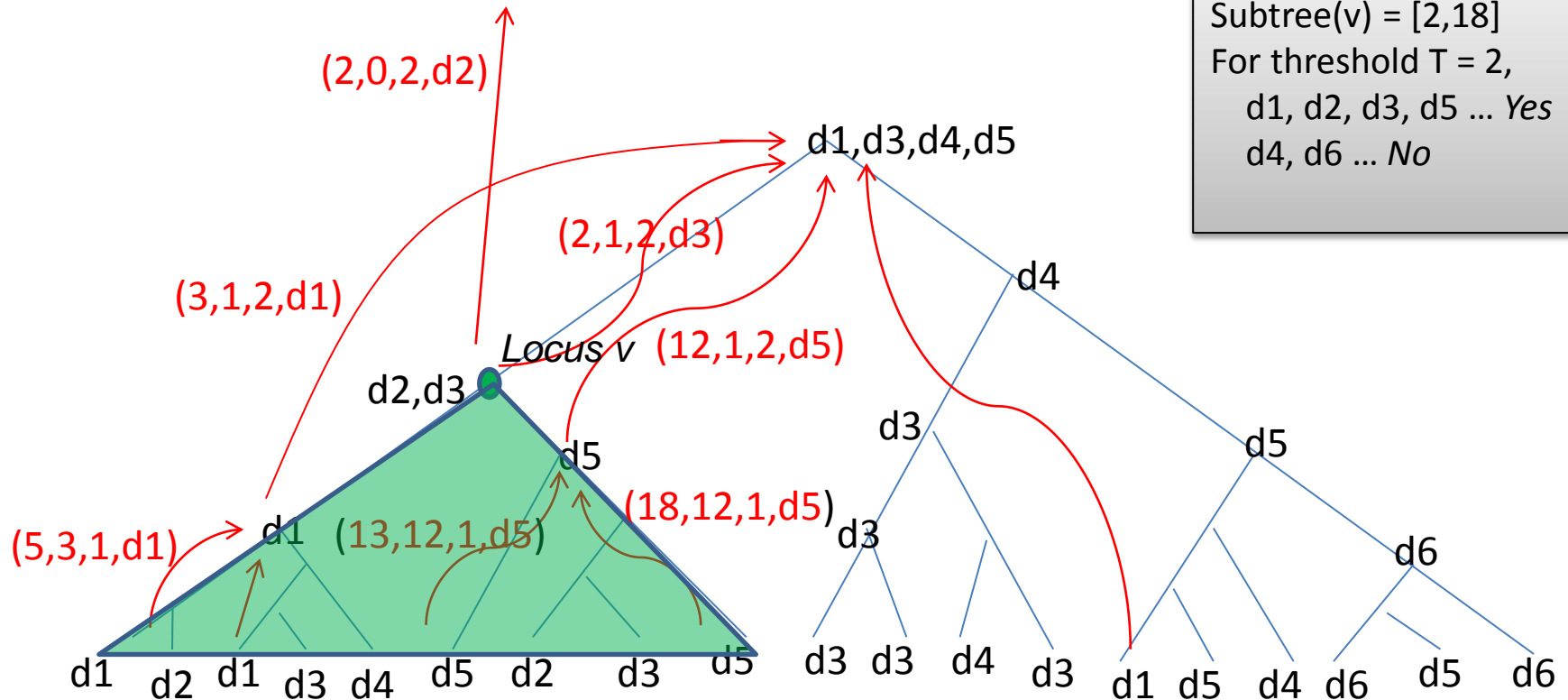
- At each node v , store the ID for document d_i if at least two subtrees of v contain d_i .
- Link every entry for document d_i to the entry of d_i in a closest ancestor node.

Tree-link Structure: Focs 2009



- At each node v , store the ID for document d_i if at least two subtrees of v contain d_i .
- Link every entry for document d_i to the entry of d_i in a closest ancestor node.
- Each link is annotated with **(origin, target, score, doc_id)** where origin and target are the **preorder** numbers for the start node and end node.

Frame query as (2,1,1) range search



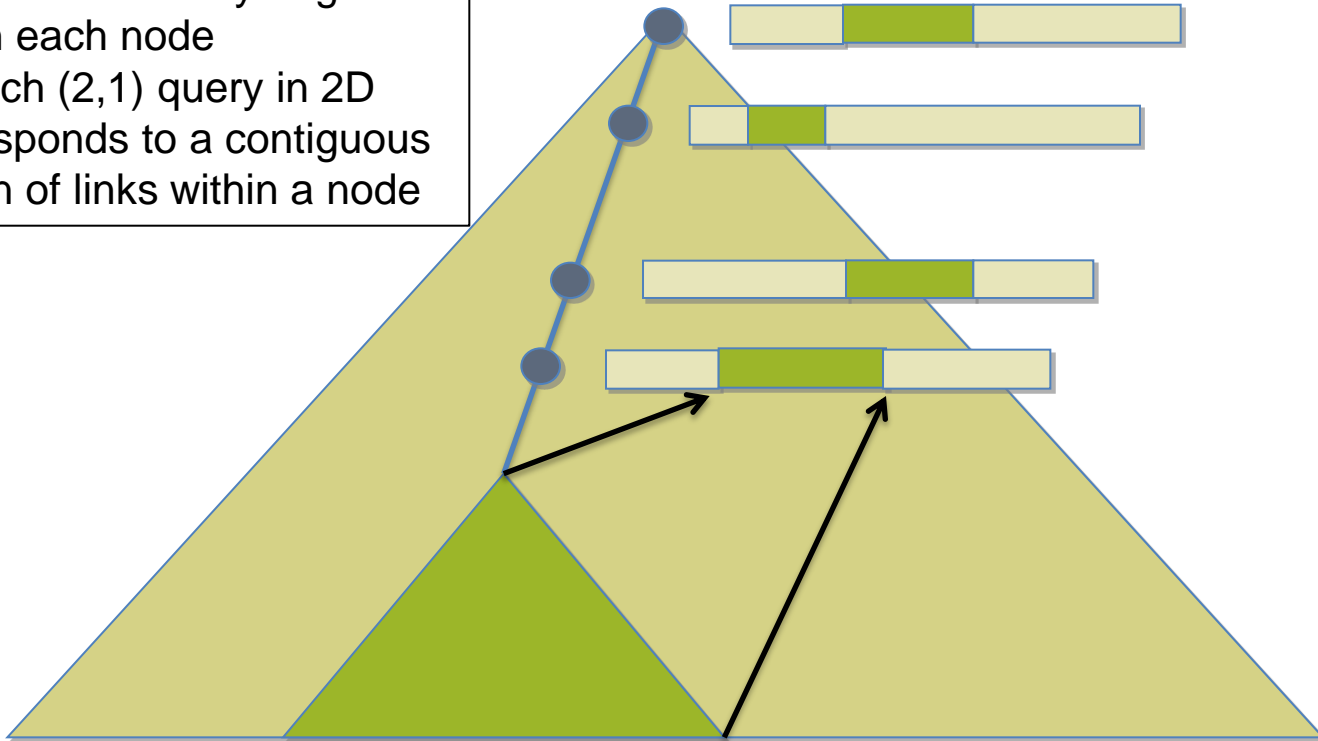
- Let's say we convert top-k into score threshold T
- Then only the links which originate from the subtree of locus node v qualify.
- Exactly one link for each document goes above v from this subtree
- Among such links, we want to those with score $\geq T$.

Main Idea !

- Each link has four attributes: (origin, target, origin_score, doc_id)
- (2,1,1)-range query in 3D
 - In previous example, get all links with
 - Origin in $[2,18]$ (subtree of v , the range where pattern matches)
 - Target value < 2 (enforces target above node v , uniqueness of each document)
 - Origin score ≥ 2 (applies score threshold)
 - Best linear space structure takes $O(\text{output} \times \log n)$ time to answer such a 3D range query — which means $O(p + \text{output} \times \log n)$ time — **too costly!**
 - Our target is $O(p + \text{output})$ time.
- New Idea: **# possible target values \leq # ancestors of $v \leq p$**
 - So group the links by their target values and query each relevant group separately via a (2, 1)-range query in 2D.
 - Each link will be represented as a triplet (origin, origin_score, doc_id).
 - At each target node is a list of all incoming links.

Query Answering

Links are sorted by origins within each node
→ each $(2,1)$ query in 2D corresponds to a contiguous region of links within a node



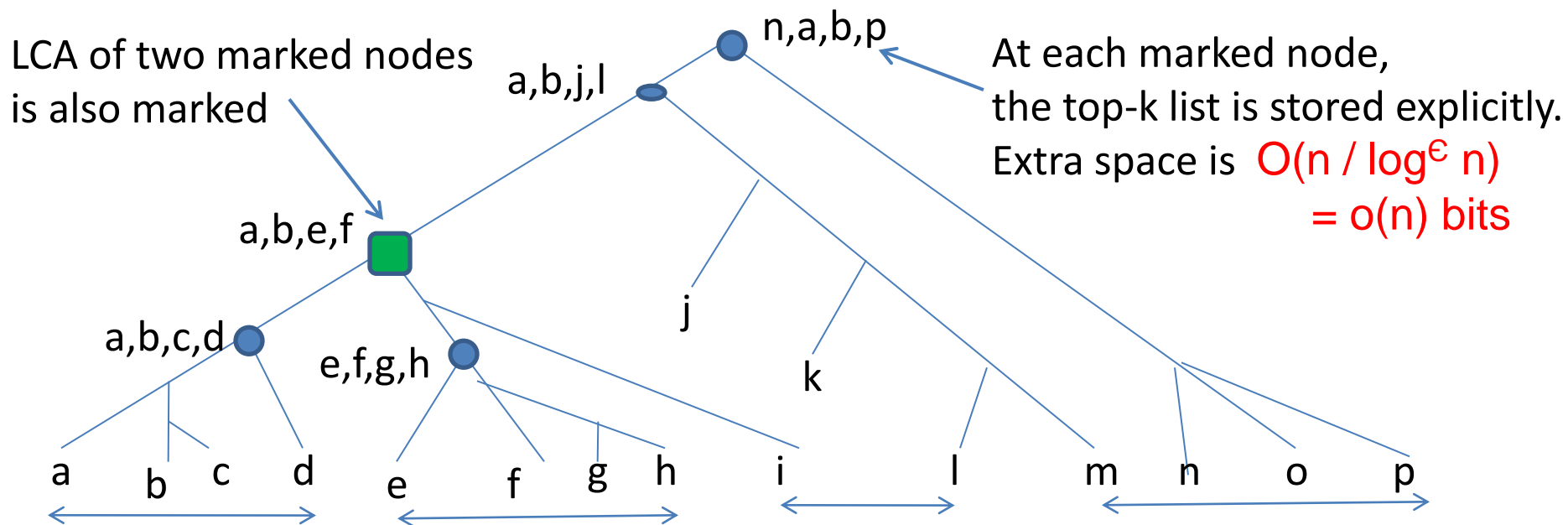
Top- k Retrieval: $O(p \log n + k \log k)$ time
Via fractional cascading: $O(p + k \log k)$ time
Via further techniques: $O(p + k)$ time

Compressed Data Structure

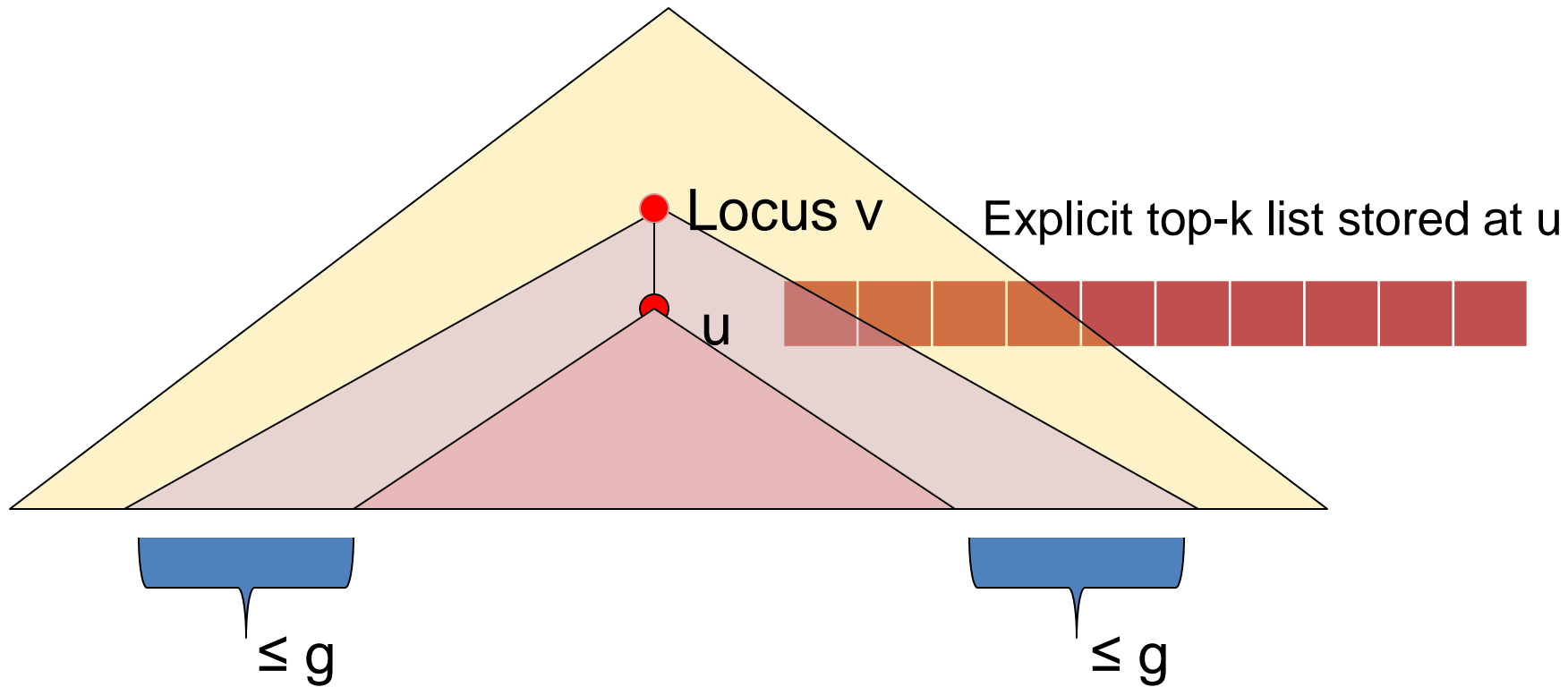
- $O(n)$ words of space in previous solution (i.e., $O(n \log n)$ bits) is MUCH BIGGER than text
- Can we design data structures that take only as much space as a compressed text? And still answer queries efficiently?
- Yes! We show solutions based on sparsification and CSA (compressed suffix array).

Sparsification example

Group consecutive $g = k \times \log^{1+\epsilon} n$ leaves and mark them.
Build a Compressed Suffix Array (CSA) on the n/g bottom-level marked nodes.



Query Approach



Fringe leaves : $\leq 2g$ documents are separately queried for their frequency counts

Results for Document Indexing with Relevance

- $O(n)$ -word data structures
 - K-frequency, K-repeats threshold: $O(p + \text{output})$ time.
 - Top-k highest relevant documents: $O(p + k)$ time.
 - $O(n)$ and $O(n \log n)$ construction time, resp.
- Compressed data structures
 - Frequency
 - Threshold: $O(p + \text{output} \times \log^{1+\epsilon} n)$
 - Top-k: $O(p + k (\log k) (\log^{1+\epsilon} n))$
 - Space: $2 |CSA| + o(n)$
 - Importance: $\log^{1+\epsilon} n$ time per item, $1 |CSA| + o(n)$ space.
 - Document retrieval: Same.
 - No results for “proximity”; not succinctly computable

Summary of Relevance Queries

- This framework is provably optimal in query time, uses linear space, and is constructible in linear time for single-pattern queries.
- With optimizations, we get an index **7–10 × text size** that can answer queries in **<< 1 millisecond**.
- Competitive with inverted indexes.
- Can improve inverted indexes (for phrase queries).
- We give the first entropy-compressed solutions.
- Linear-space framework for multipattern queries.

How To Query Efficiently In External Memory

- Block size B
- Minimize number of block transfers
- Is it possible to get optimal $O(p/B + \log_B N + k/B)$ I/Os ?
 - Sorted ? ... No.
 - Convert top-k queries to threshold queries via sketches
- Not clear ... Last trick requires search of p separate lists (one per ancestor).


If we apply it, we get $O(p \log_B N + k/B)$ I/O ... **Too much!**
- In external memory, 3 constraints can be solved efficiently, but 4 constraints take super-linear space or $O(\sqrt{N/B})$ I/Os

Geometric Problem:


3-Sided Range Searching with Priority

If these 4 constraints were independent, no linear-space external memory structure has desired I/O performance.

But 2 constraints share a common end point, i.e., they are “hinged”

 Jeff: $O(N \log \log N)$ space with $O(\log \log N)$ multiplicative I/Os


- Wavelet tree

 Cheng: $O(N)$ space... $O(\log \log B)$ extra I/Os

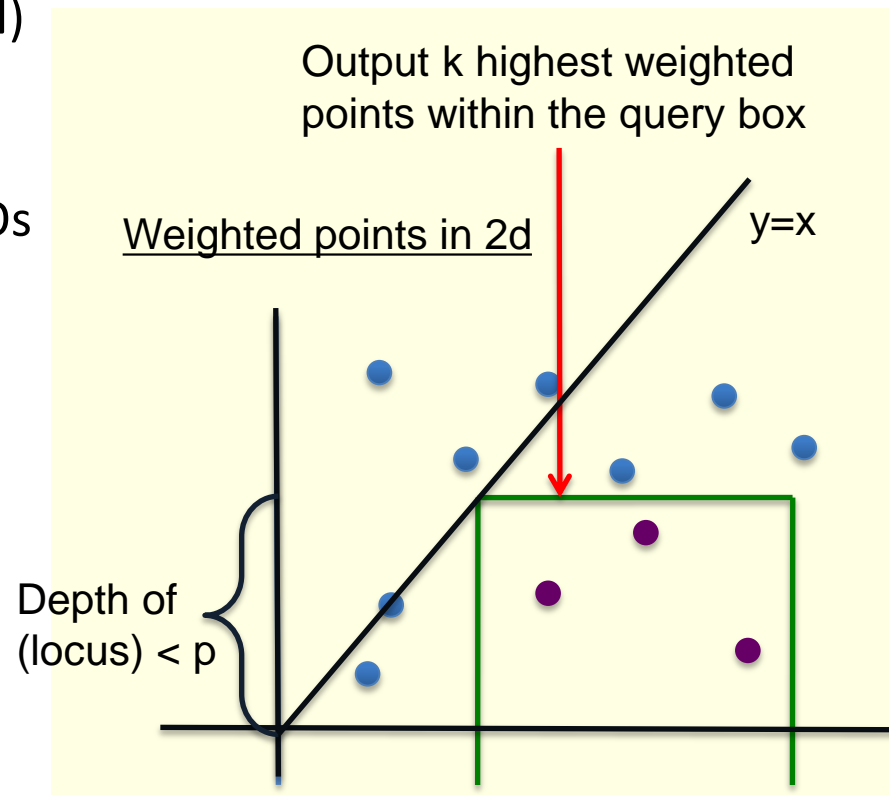
- Using bootstrapping with polynomial reduction

 Sharma: $O(N \log \log \log B)$ space ... optimal # I/Os

- Using bootstrapping with small buckets

 Rahul: $O(N \log^* N)$ space... optimal $O(p/B + \log_B N + k/B)$ I/Os

- Using bootstrapping with logarithmic reduction + recursive bucketing



Practical Index

- We come back to storing (origin, score, doc_id) lists at target nodes
- Index A
Replace GST by the compressed suffix tree (CST)
- Index B
Encode the origin info of the links by run-length (gap) encoding.
Encode the frequency or score by variable-length encoding
- Index C
Drops the document info in each entry
(retrieval can be done on-the-fly by traversing the origins recursively until reaching a leaf node)

Experimental Setup for word-phrase searching

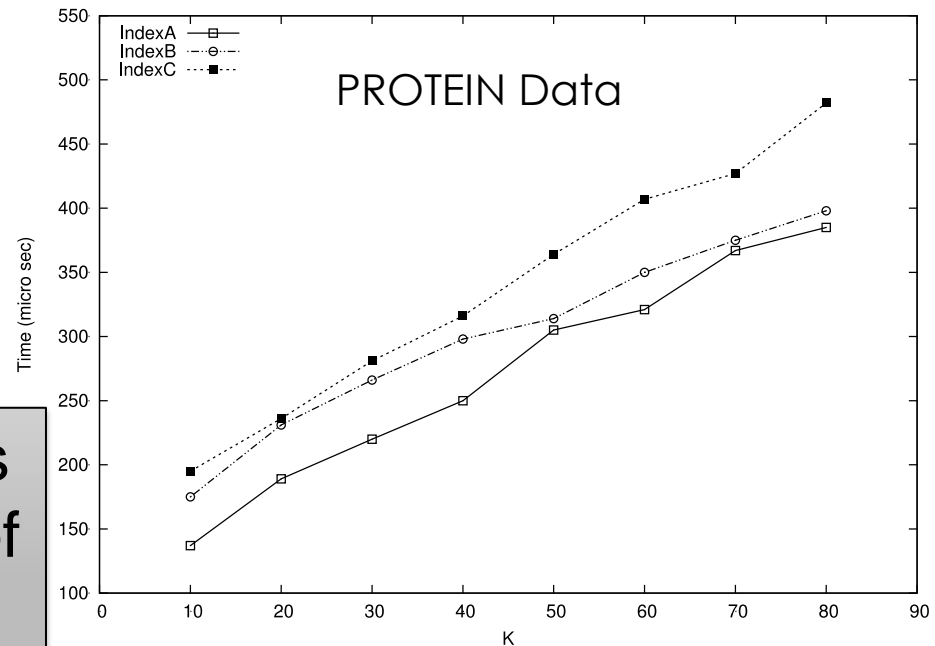
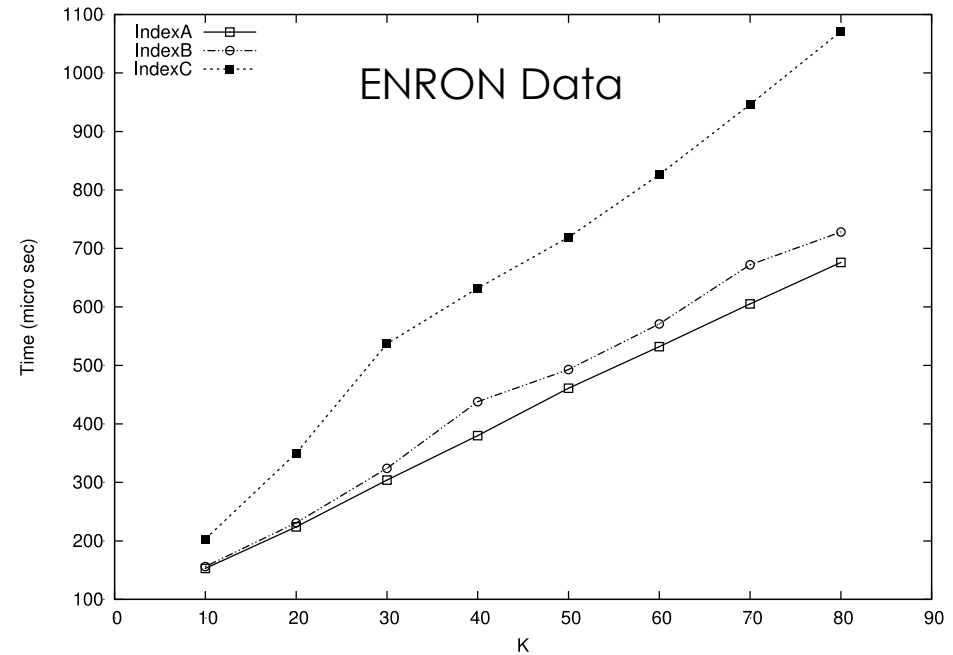
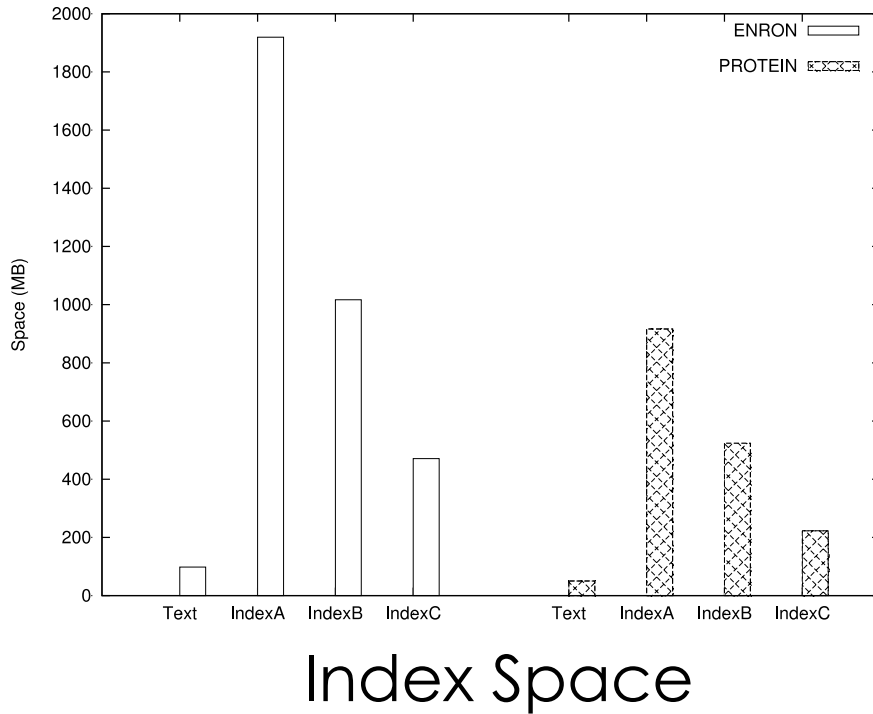
Datasets

1. ENRON:
48,619 email messages from ENRON executives drawn from a dataset prepared by the CALO Project, totaling 100MB
2. PROTEIN:
Concatenation of 141,264 Human and Mouse protein sequences, totaling 60MB

Public Code Libraries

1. Succinct Data Structures Libraries (@ University of Ulm)
2. Compressed Suffix Trees (@ PizzaChili Corpus)

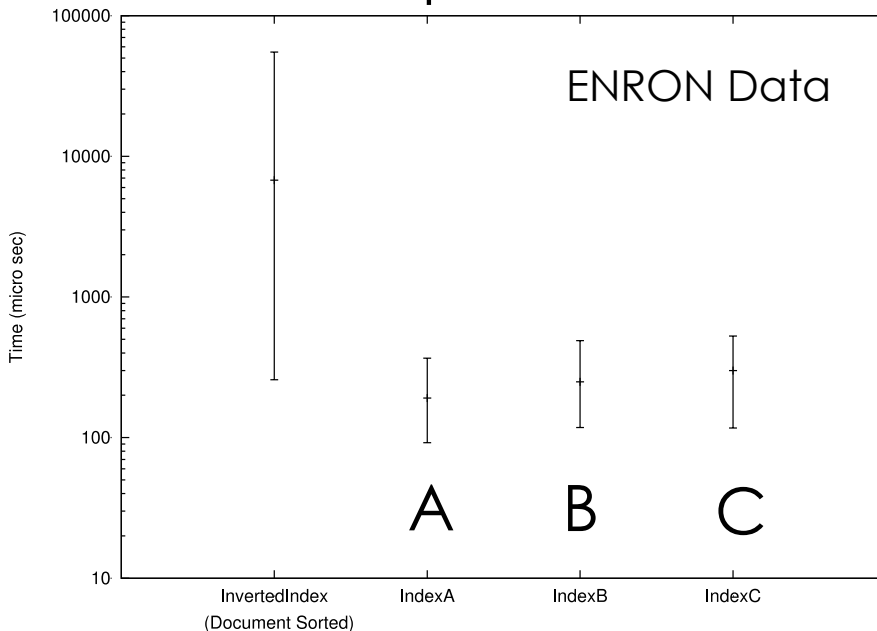
Space vs. Time Tradeoff



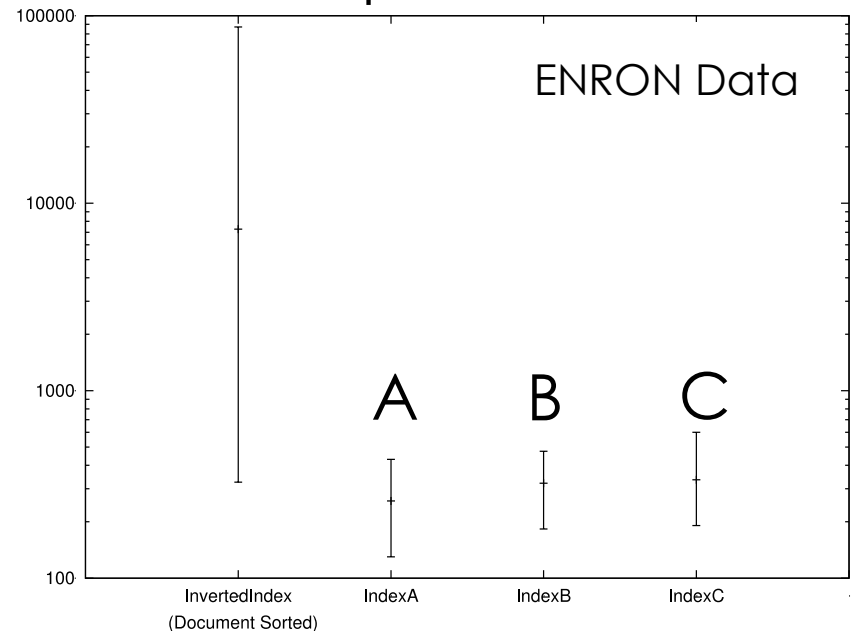
Mean time to report k documents with highest frequency for a set of queries with 3 words

Our Index vs. Inverted Index (Query)

2-word phrase
queries



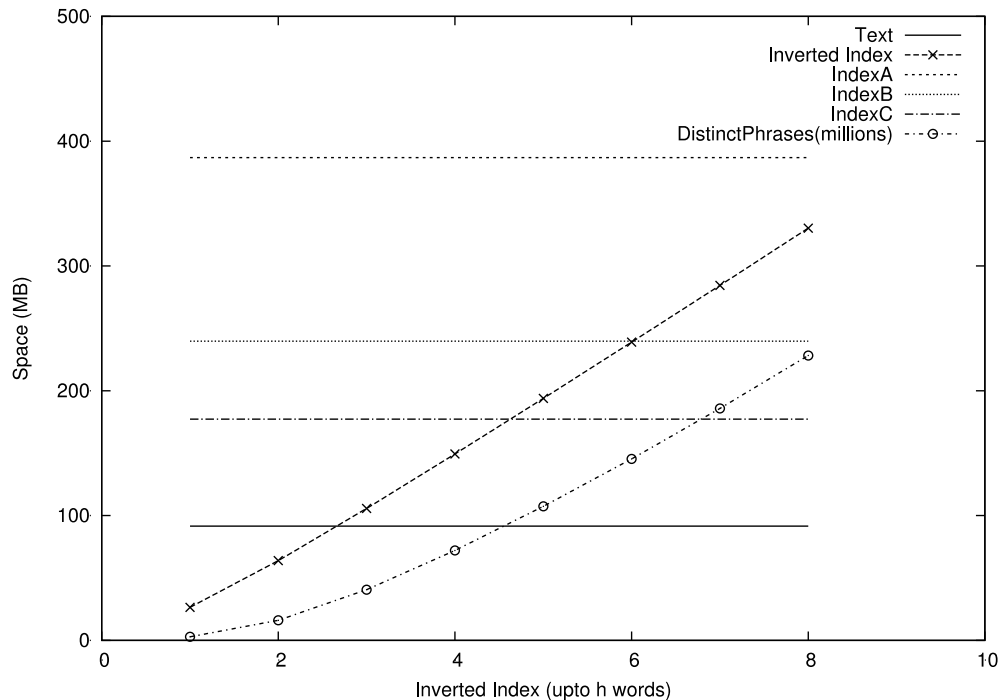
3-word phrase
queries



Time (High, Low, Mean) to report 10 documents with highest frequency for a set of phrase queries with 2 words and 3 words

Our Index vs. Inverted Index (Space)

- To speed up phrase searching, we may store inverted lists of all phrases up to h words \rightarrow the larger the h , the more the index space

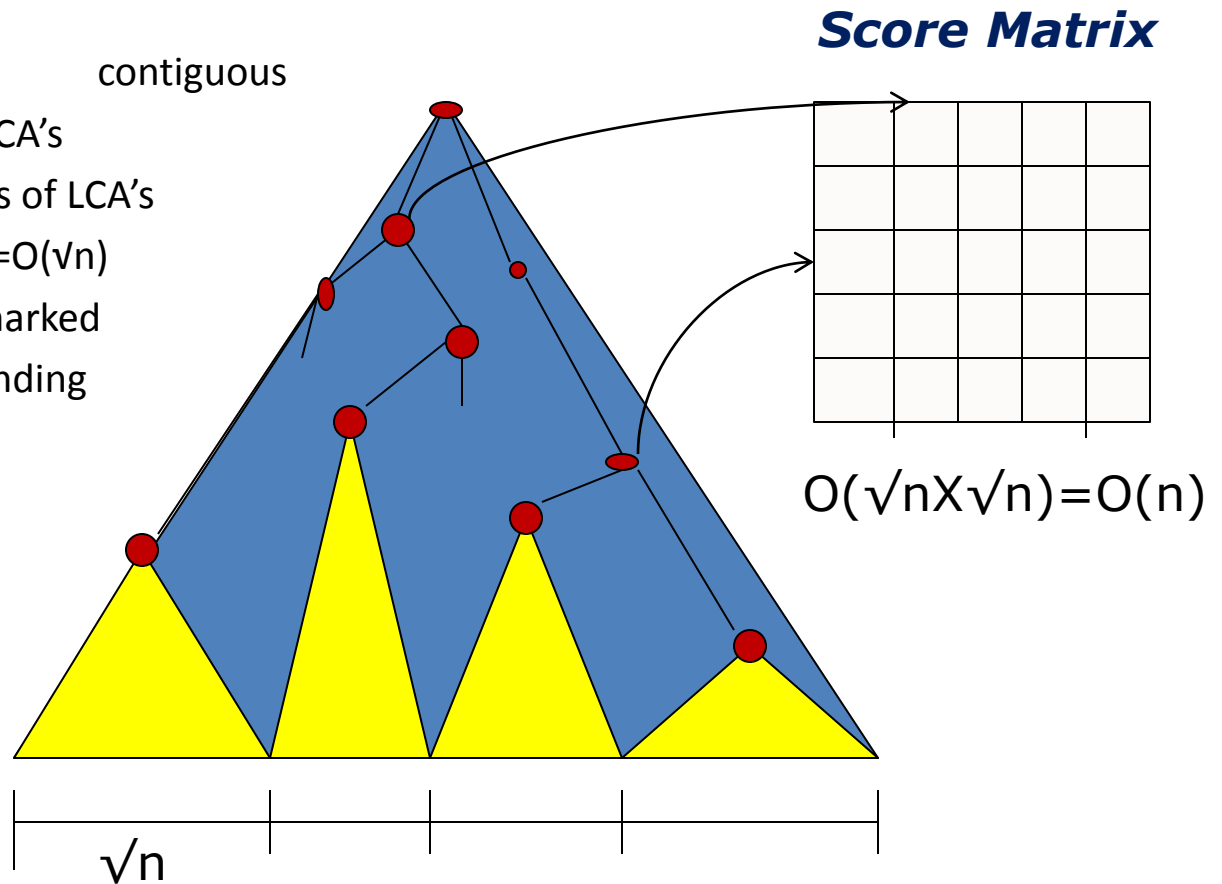


Retrieval for Multiple Patterns

- Example relevance measures: TFIDF, proximity between 2 patterns, combined frequency scores
- Top-k: $O(n)$ words of space index with $O(p_1 + p_2 + \sqrt{nk \log^2 n})$ query time
- Top-k with approximate TFIDF is achievable
- Succinct results ?
 - Proximity unlikely

Any-1 / Top-1 Index

- Build GST on D
- Group every $g (= \sqrt{n})$ contiguous leaves and mark their LCA's
- Mark LCA's of each pairs of LCA's
- #marked nodes $\leq 2n/g = O(\sqrt{n})$
- Between each pair of marked nodes (for the corresponding patterns), store the top score in score matrix



- We also keep individual structures for each document
- Total size $= O(n)$ words

Practical Shortcuts for Searching Genome

[KHSVX10]

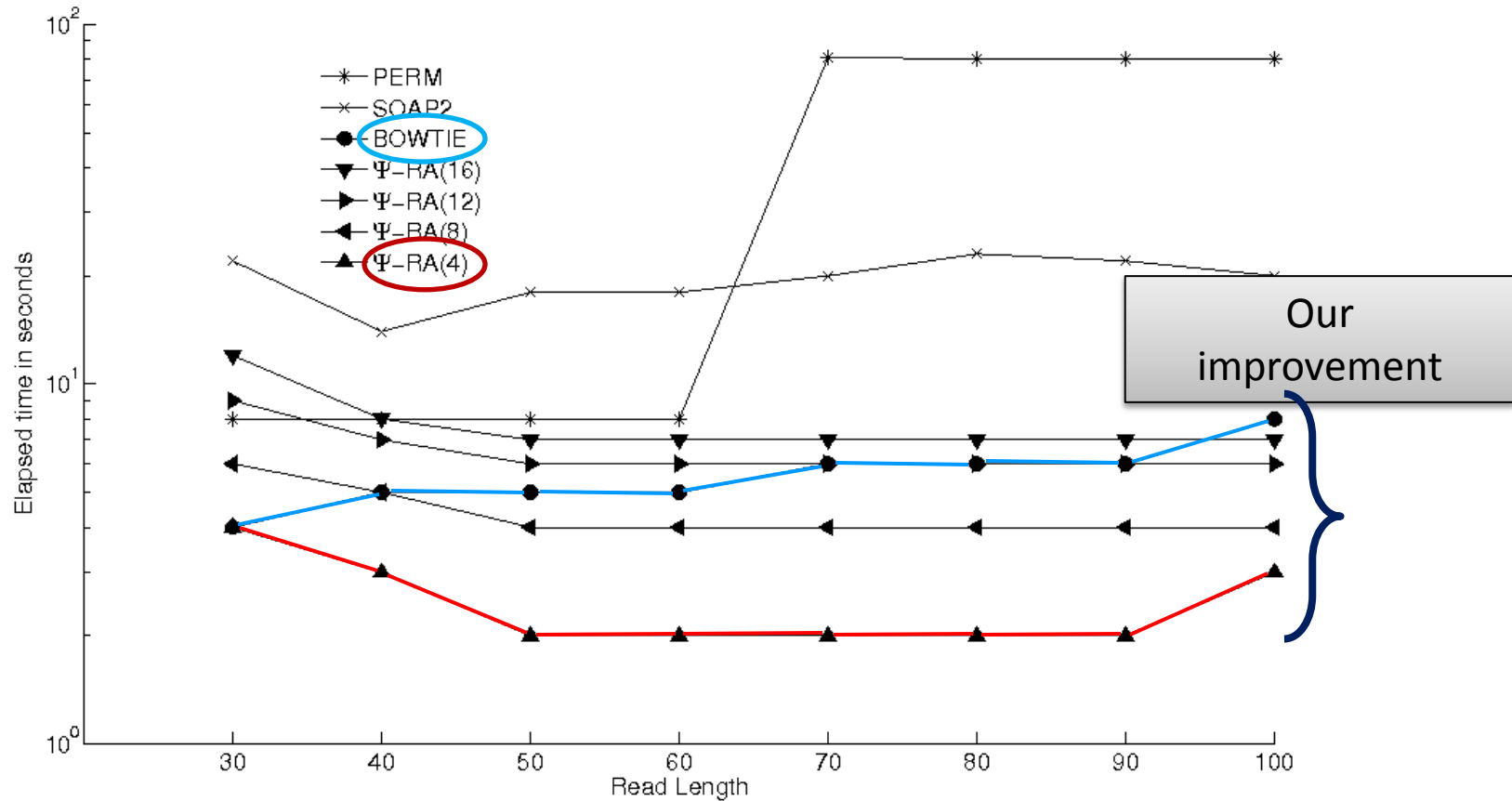
- Human genome is not readily compressible
- Consists of ≈ 3 billion base pairs ≈ 750 MB space
- Key idea is instead Geometric Burrows-Wheeler Transform sparsification, $d > 1$
- Tradeoff: speed (low d) vs. succinctness (high d)
- Verify 1-d results rather than use 2-d searching
- Prioritize rightmost mismatches (where data is less precise)

Size of the Index for the Human Genome Using Different Aligners

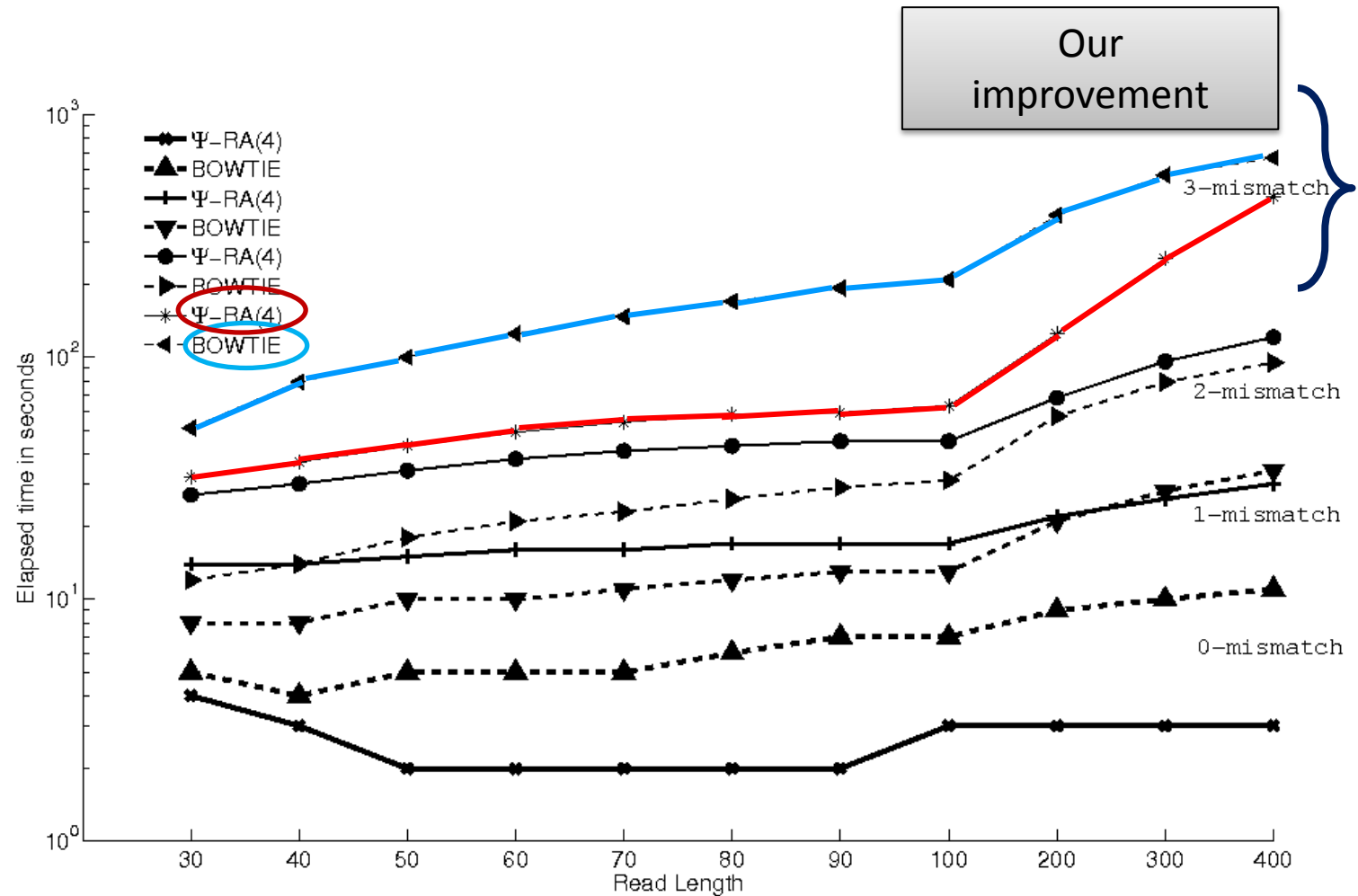
- PerM : 12.4 GB, spaced seeds
- SOAP2 : 6.1 GB, bidirectional BWT
- BOWTIE : 2.3 GB, bidirectional BWT
- Ψ -RA(4) : 3.4 GB, sparse SA, d=4 bases
- Ψ -RA(8) : 2.0 GB, sparse SA, d=8 bases
- Ψ -RA(12) : 1.6 GB, sparse SA, d=12 bases

Raw human genome occupies ≈ 750 MB
(when each base is coded by 2 bits)

Exact Match: Time for 100K “Reads”



Approximate Matching: Time for 100K Reads



Multi-core Performance: Time for 100K Reads

Key Takeaway: Our algorithm is easily parallelizable

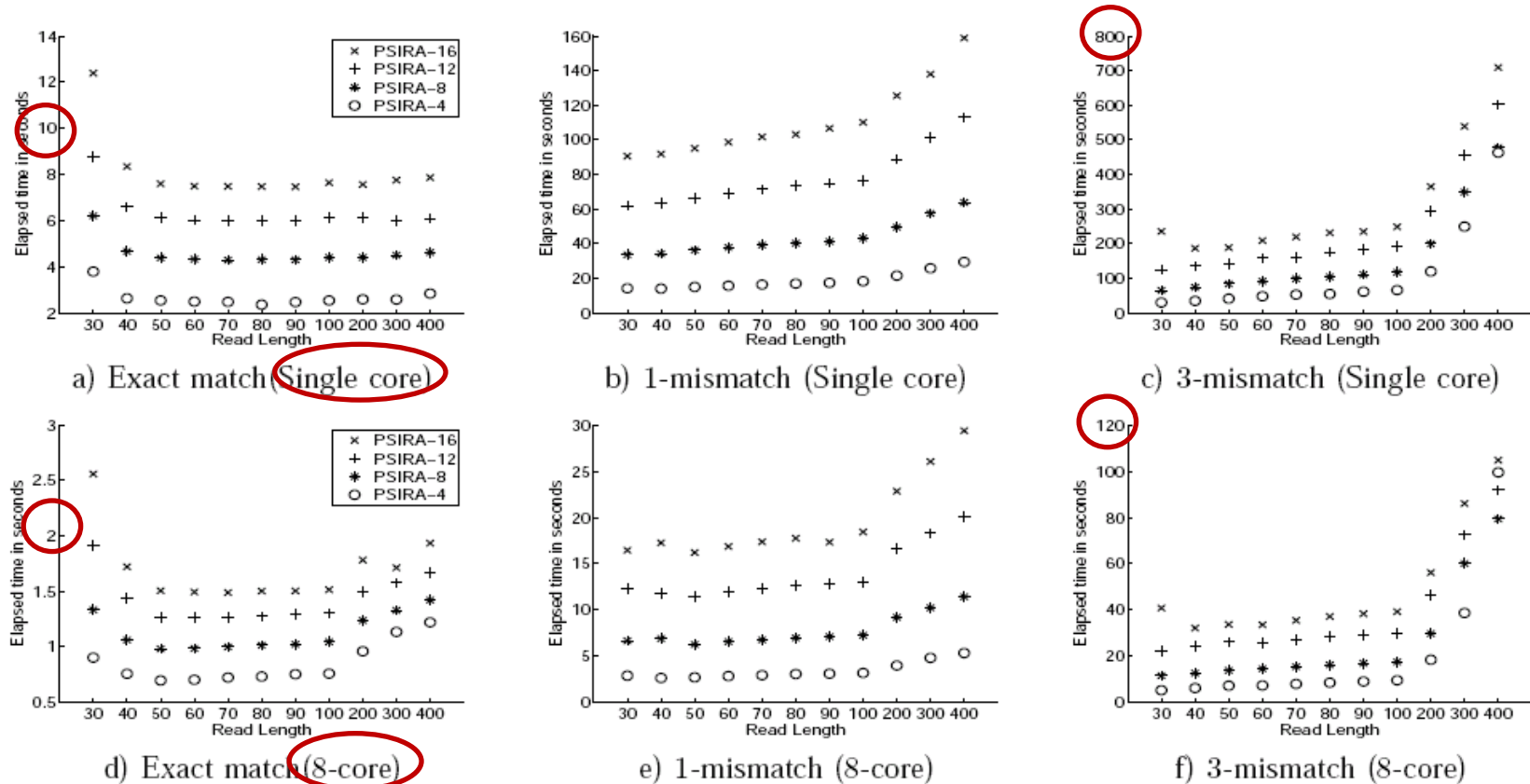


Figure 3. The effect of sparsification factor D on single thread versus eight thread executions.

Future Challenges in Compressed Data Structures

Our goal is to realize the advantages of inverted indexes but allow more general search capability.

Many exciting challenges to explore!

- Computing directly on compressed data. . . in many settings
- External memory performance
- Building relevance into queries (outputting top k)
- Dual problem of dictionary matching
- Biological applications
- Streaming problems
- Approximate matching, maximal matching, 2D matching, . . .
- Building practical systems