Compression, Indexing, and Retrieval for Massive String Data

Jeff Vitter
Texas A&M University and University of Kansas, USA

with coauthors

Wing-Kai Hon
National Tsing Hua University, Taiwan

Rahul Shah
Louisiana State University, USA

(and collaborators Sabrina Chandrasekaran, Yu-Feng Chien, Sheng-Yuan Chiu,
Oğuzhan Külekci, Manish Patil, Sharma Thankachan, & Bojian Xu)

Outline

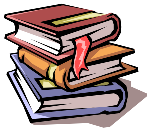
- Background on entropy-compressed data structures
  - Compressed Suffix Arrays
  - FM-index
  - Wavelet tree
  - Using Sparse Suffixes (for external memory)
- Goal is in sight:

  To achieve the efficiency of inverted indexes
  but allow more general patterns (as in suffix trees)
- External Memory Text Indexing with Compression
- Document Retrieval
  - Top-k retrieval (relevance)
  - Searching for Multiple Patterns
- Conclusions and Open Problems

The Attack of Massive Data

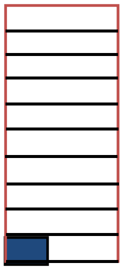
- Lots of massive data sets being generated
  - Web publishing, bioinformatics, XML, e-mail, satellite geographical data
  - IP address information, UPCs, credit cards, ISBN numbers, large inverted files
  - Petabytes of data (10^{15} bytes), soon exabytes (10^{18} bytes)
- Data sets need to be compressed (and are compressible)
  - Mobile devices have limited storage available
  - Search engines use DRAM in place of hard disks
  - Next generation cellular phones will cost # bits transmitted
  - I/O overhead is reduced
  - There is never enough memory!
- **Goal: design data structures to manage massive data sets**
  - Near-minimum amount of space
    - Measure space in *data-aware* way, i.e. in terms of each individual data set
  - Near-optimal query times for powerful queries
  - Efficient in external memory

Analogy to a card catalog in a library

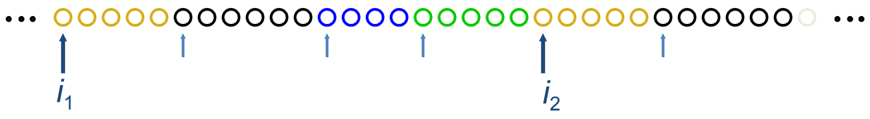


- 10-floor library
- card catalog near front entrance
- indexes books’ titles and authors

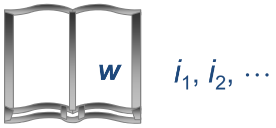
- negligible additional space
- a small card (few bytes) per book
- limited search operations!



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, ⋯



Can be implemented in about 15% of text space using gap encoding of inverted lists.

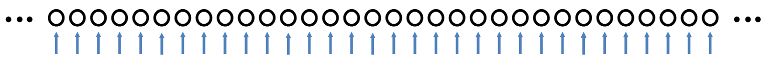
Full-text Indexing
(where pattern P is arbitrary)

Given a text T of n characters from an alphabet Σ, build an index that can answer the following queries for any input pattern P (of length p):

1. **Count** the number of occurrences of P in T;
2. **Report** the locations in T where P occurs.

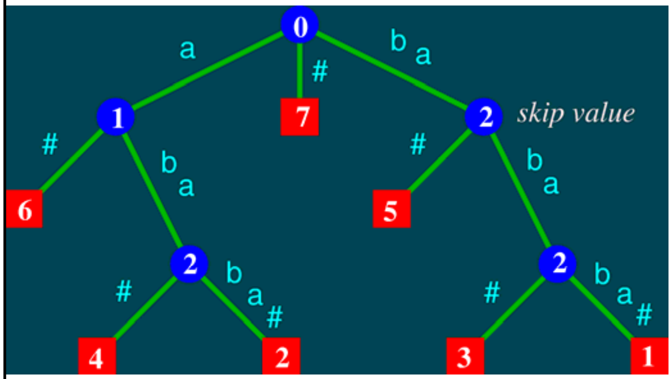
Inverted Indexing Won't Work Well

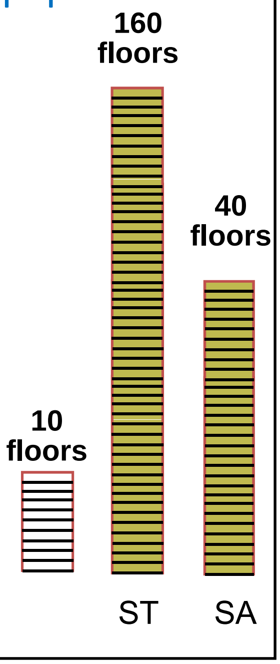
- Not handled efficiently by Google
- Clear notion of word is not always available:
  - Some Eastern languages
  - unknown structure (e.g., DNA sequences)
- Alphabet Σ, text T of size n bytes (i.e., $n \log |\Sigma|$ bits) : each text position is the start of a potential occurrence of P



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, |Σ|=2





- Suffix tree is a compact trie storing the suffixes of the input string (eg., bababa#).
  - Space is O(n log n) bits, ≈ 16 × text size.
- Suffix array is the array of leaf values.
  - Space is O(n log n) bits, ≈ 4 × text size.

Compressed Suffix Array [Grossi, Vitter STOC00]

- Simulate the suffix array-based binary search
- Avoid storing all the suffix array entries.
  Instead generate them on the fly:

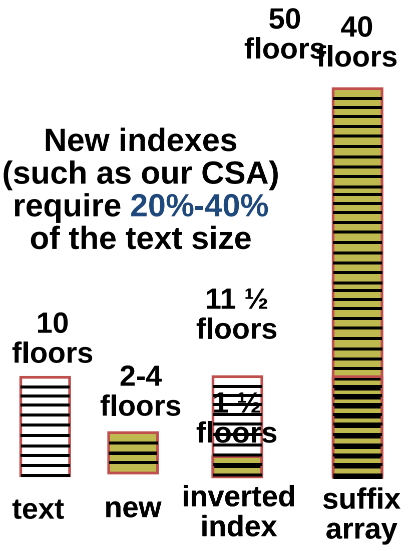
T = a b a a b a b b $

SA =	3	1	4	6	2	5	7	8
Φ =	3	5	6	7	1	4	8	2

Φ is "neighbor function": SA(Φ[i]) = SA[i] + 1

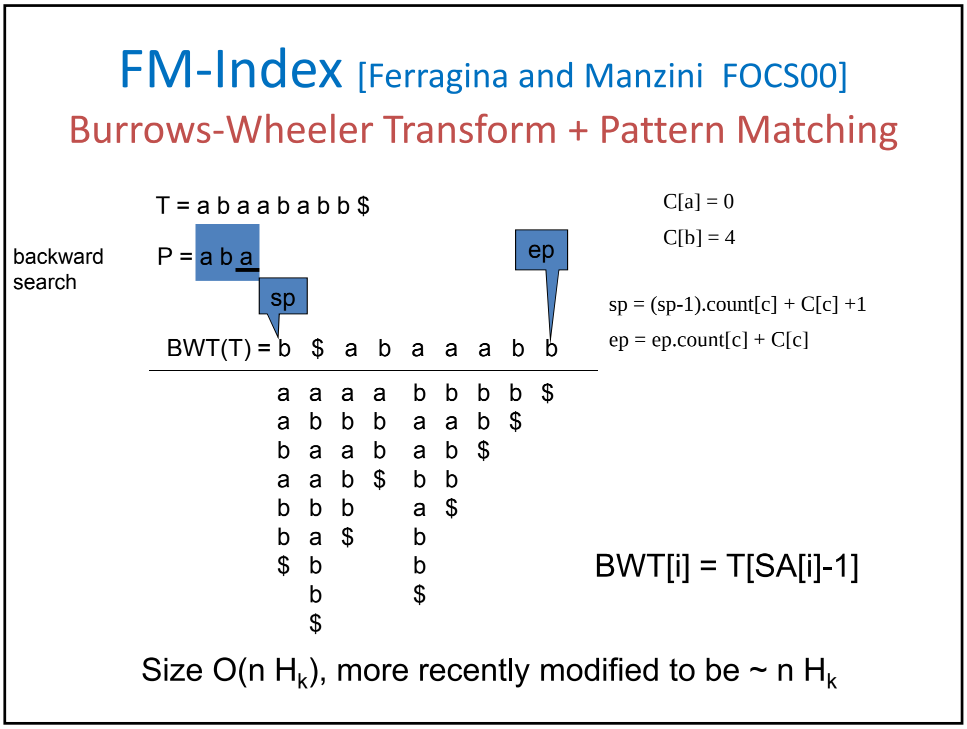
- Φ values are increasing runs, gap compressible
- Recurse to looking only at suffixes in even positions.
- Resulting data structure: first w/ linear size O(n log |Σ|).
- [Sada ISAAC00]: O(n H_0) space, modified it to be self index.

Compressed Suffix Array [Grossi, Gupta, Vitter SODA03]



New indexes (such as our CSA) require 20%-40% of the text size

- O(*p* + polylog(*n*)) search time .
- First index with size equal to text size in *entropy-compressed form* (~ 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).



FM-Index [Ferragina and Manzini FOCS00]

Burrows-Wheeler Transform + Pattern Matching

T = a b a a b a b b $

backward search

P = a b a

sp

ep

BWT(T) = b $ a b a a a b b

```
a  a  a  a   b  b  b  b  $
a  b  b  b   a  a  b  $
b  a  a  b   a  b  $
a  a  b  $   b  b
b  b  b      a  $
b  a  $      b
$  b         b
   b         $
   $
```

C[a] = 0

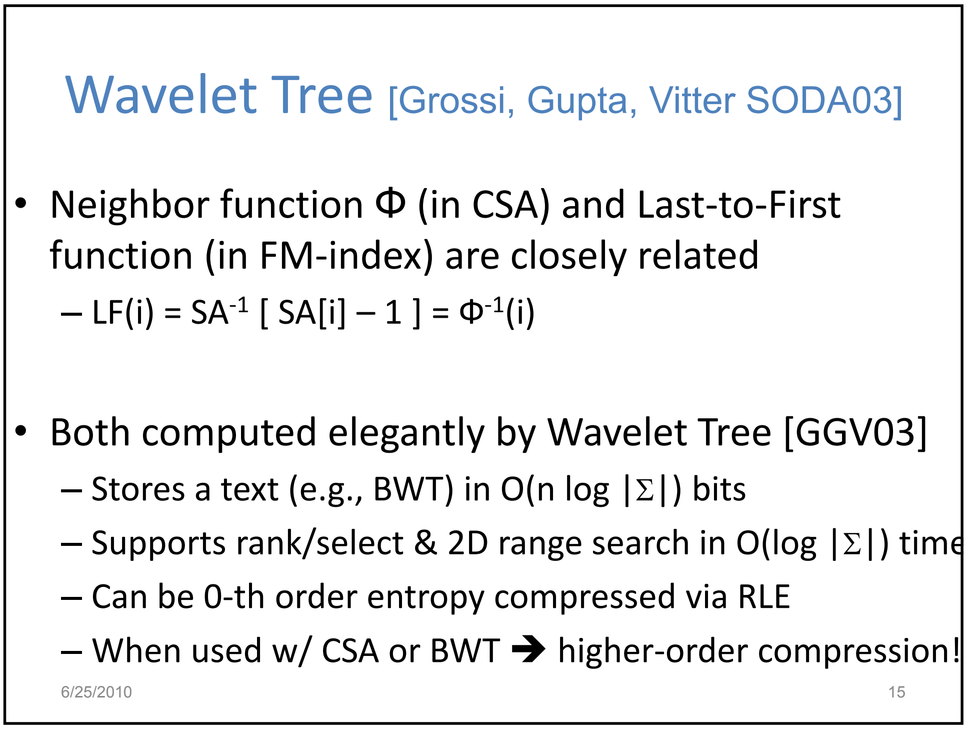
C[b] = 4

sp = (sp-1).count[c] + C[c] +1

ep = ep.count[c] + C[c]

BWT[i] = T[SA[i]-1]

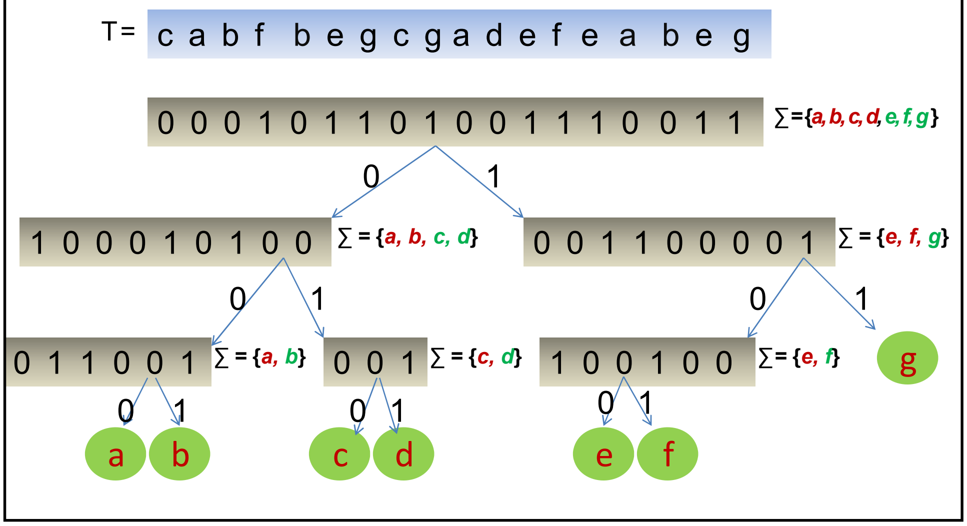
Size O(n H_k), more recently modified to be ~ n H_k



Wavelet Tree [Grossi, Gupta, Vitter SODA03]

- Neighbor function Φ (in CSA) and Last-to-First function (in FM-index) are closely related
  - LF(i) = SA^{-1} [SA[i] – 1] = Φ^{-1}(i)
- Both computed elegantly by Wavelet Tree [GGV03]
  - Stores a text (e.g., BWT) in O(n log |Σ|) bits
  - Supports rank/select & 2D range search in O(log |Σ|) time
  - Can be 0-th order entropy compressed via RLE
  - When used w/ CSA or BWT → 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 Σ={a, b, c, d, e, f, g}

0 1

1 0 0 0 1 0 1 0 0 Σ = {a, b, c, d}

0 0 1 1 0 0 0 0 1 Σ = {e, f, g}

0 1

0 1

0 1 1 0 0 1 Σ = {a, b}

0 0 1 Σ = {c, d}

1 0 0 1 0 0 Σ = {e, f}

g

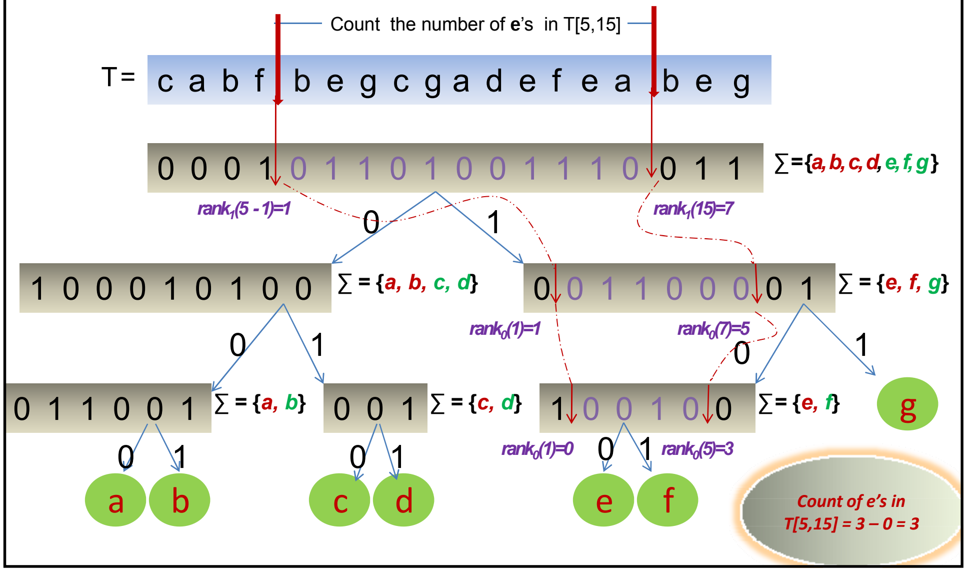
0 1

0 1

0 1

a b c d e f

Wavelet Tree [Grossi, Gupta, Vitter SODA03]



Count the number of **e**'s in T[5,15]

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 Σ={a, b, c, d, e, f, g}

$rank_1$(5 - 1)=1

$rank_1$(15)=7

0 1

1 0 0 0 1 0 1 0 0 Σ = {a, b, c, d}

0 0 1 1 0 0 0 0 1 Σ = {e, f, g}

$rank_0$(1)=1

$rank_0$(7)=5

0 1

0 1

0 1 1 0 0 1 Σ = {a, b}

0 0 1 Σ = {c, d}

1 0 0 1 0 0 Σ = {e, f}

g

$rank_0$(1)=0

$rank_0$(5)=3

0 1

0 1

0 1

a b c d e f

Count of e's in T[5,15] = 3 – 0 = 3

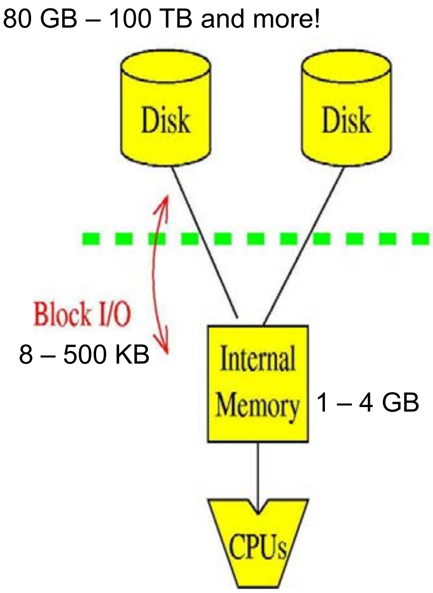
Two Challenges

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

Challenges discussed today:

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

Parallel Disk Model [Vitter, Shriver STOC90, 94]



N = problem size

M = internal memory size

B = disk block size

D = # independent disks

Scan: $O(N/DB)$

Sorting: $O((N/DB) \log_B(N/M))$

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

See [Vitter 08] book for overview

Challenge 1: How to externalize?

Problem:

Compressed Suffix Array and FM-Index
access memory randomly and do not exploit locality
and thus have poor I/O performance!

- 1999: String B-tree
  - Introduced by Ferragina and Grossi [FG99]
  - External memory version of suffix array (B-tree + SA)
  - Optimum I/O bound O(p/B + $\log_B$ n + output/B), where B is block transfer size to/from disk

Linear Space
O(n) words =
O(n log n) bits
☹

- 2008: Geometric Burrows-Wheeler transform
  - index by Chien et al [CHSV08] (using sparse suffixes)
  - ***Can we make it compressed ?***

Succinct Space
O(n log Σ) bits

O($n H_k$) bits?

Index Using Sparse Suffixes [CHSV08]

	d	d	d	d
T =	a a b	a b a	a b a	a b b

Suffixes from blocking boundaries (KU96a)

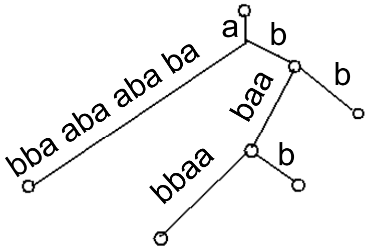
aab aba aba abb

aba aba abb

aba abb

abb

Blocking size
d = ½ $\log_{|\Sigma|}$ n characters
≈ ½ word size
(d = 3 chars. in this example)



Pattern P = ***aaba***

Search for P in sparse suffix tree

Sparse suffix tree(only 4 suffixes)

Index Using Sparse Suffixes [CHSV08]

	d	d	d	d
T =	a a b	a b a	a b a	a b b

d = 0.5 $\log_{|\Sigma|} n$

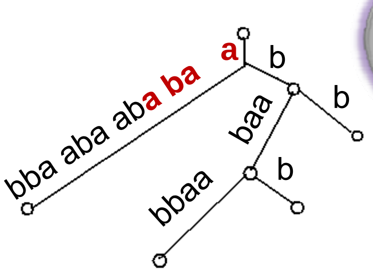
Suffixes from blocking boundaries

aab **a**ba aba abb

aba aba abb

aba abb

abb



Pattern is found at the beginning of the first block
Output size = 1

Pattern P = ***aaba***

Search for P in sparse suffix tree

Sparse suffix tree(only 4 suffixes)

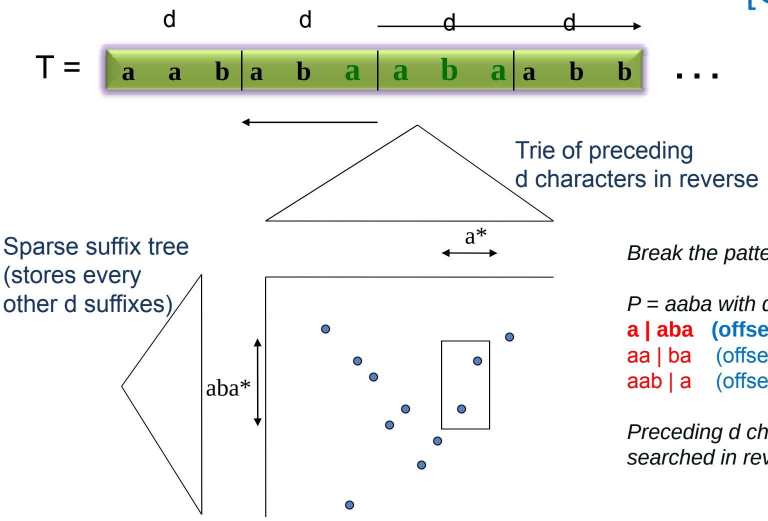
Index Using Sparse Suffixes [CHSV08]

	d	d	d	d
T =	a a b	a b a	a b a	a b b
			d	d

But sparse suffix tree could not capture the second occurrence

Sparse suffix tree will report only those occurrences that start at a blocking boundary

Geometric BWT: Index Using Sparse Suffixes [CHSV08]



Break the pattern into d pairs :

P = aaba with d=3 is broken into:
a | aba **(offset 1)**
aa | ba (offset 2)
aab | a (offset 3)

Preceding d chararacters are searched in reverse

Can use wavelet tree for 2D search.
d chars fit in machine word → d matches take O(p) time

How to search patterns within a block

- Four Russians technique (table lookup)
- If d < $(\log_\Sigma n)/2$,
  # of different short patterns ≤ $\sqrt{n} \log n$
- Construct a generalized suffix tree of unique d-sized blocks
  - Find the set of blocks in which the pattern occurs
  - Next use inverted index which for each unique block, stores the list of positions in T where the block occurs

Index Using Sparse Suffixes [CHSV08]

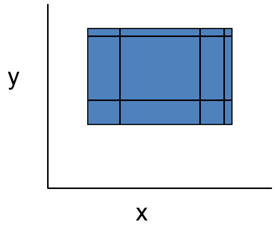
- Pattern matching problem → Orthogonal Range searching query
  - Use R-tree or segment tree or even wavelet tree
- History of 2d range search for exact and approximate pattern matching:
  - Lempel-Ziv parsing, Karkkainen and Ukkonen [KU96b]
  - Amir, Keselman, Landau, Lewenstein[2], Rodeh [AKLLLR99]
  - Grossi and Vitter [GV00]
  - Makinen and Navarro [MK06]
  - Hon, Shah, Vitter [HSV06] etc.
- Externalizing the Index
  - Suffix tree → String B-tree
  - Internal memory range query structure (like wavelet tree) → external memory (linear-space) 4-sided structure
  - **$O(n \log |\Sigma|)$** bits space
  - **$O(p/B + (\log_{|\Sigma|} n)(\log_B n)$ + output $\log_B n)$** I/Os for reporting
  - Other tradeoffs also possible using other range search structures
- Two questions:
  - Can last term in I/O bound be improved to **output/B** ?
  - Can we achieve a compressed version of this Index ?

    i.e., **$O(n \log |\Sigma|)$** → **$O(n H_k)$?**

Points2Text transformation

- Transforms points in 2D to text such that range queries can be answered by using pattern matching queries
- Consider points (x_i, y_i); construct a text block $x_i^R\#y_i\&$ for each point and concatenate them to form text T
- 2D range query on these points can be answered by using $\log^2 n$ pattern matching queries on T
- Given range $[x_{low} .. x_{high}]$, break it up into log n binary ranges, where each range is specified by a prefix bitstring of ≤ log n characters
- → 2D range can be broken into $\log^2 n$ binary 2D ranges
- Each binary range is queried by pattern $x^R\#y$

$T=x_1^R\#y_1\&x_2^R\#y_2\& \ldots\&x_n^R\#y_n\&$



--1D range [7,18] i.e. [00111,10010] can be broken into{00111, 01***, 1000*, 10010}

--2D range ([16,19],[8,15]) → **001#01***

→ pattern 001#01

Existing bounds for range queries

→ Output bounds for polylog queries for text indexing are tight

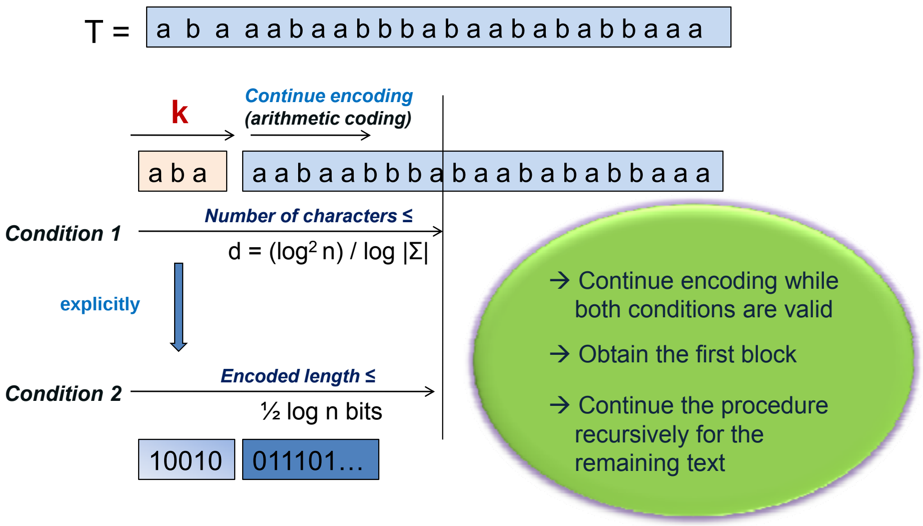
(Alternative is square root time queries with output/B output)

How to Achieve Entropy-Compressed Space ?

- What if we apply entropy compression on each block?
  - Number of blocks is still n / d
  - n / d pointers
  - **$O((n/d) \log (n/d)) = O(n \log |\Sigma|)$ bits**

- How to reduce the number of blocks?
  - Variable blocking factor d (d= # characters per block)
  - Combine as many characters as possible that can be compressed into a machine word
  - d can be very large in case of frequent blocks
  - d offsets → **$O(d_{max}\, p / \log_{|\Sigma|} n + \dots)$ I/Os**

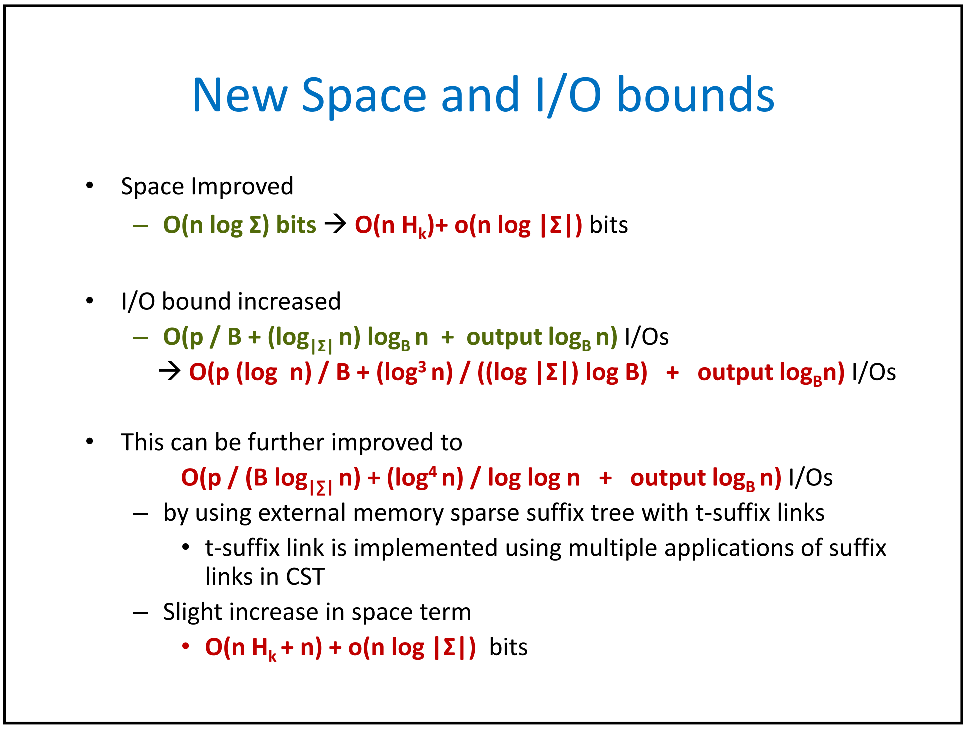
Our O(H_k)-bit Encoding Scheme [HSTV09]


T = a b a a a b a a b b b a b a a b a b a b b a a a
k
Continue encoding (arithmetic coding)
a b a
a a b a a b b b a b a a b a b a b b a a a
Condition 1
Number of characters ≤
d = (log² n) / log |Σ|
explicitly
Condition 2
Encoded length ≤
½ log n bits
10010
011101…

→ Continue encoding while both conditions are valid

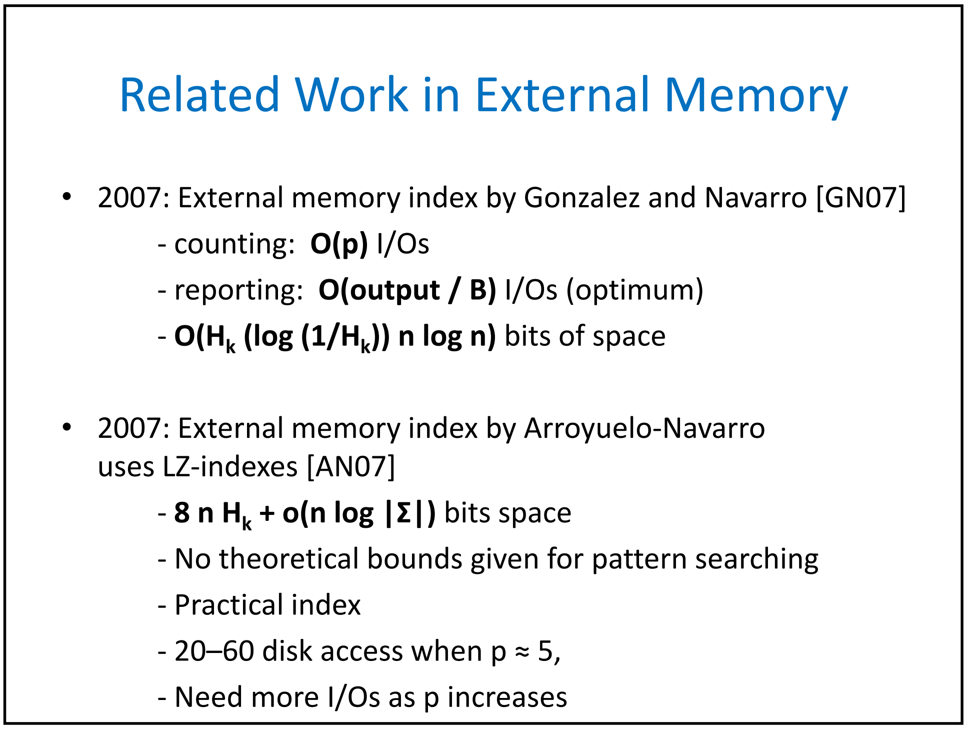
→ Obtain the first block

→ Continue the procedure recursively for the remaining text



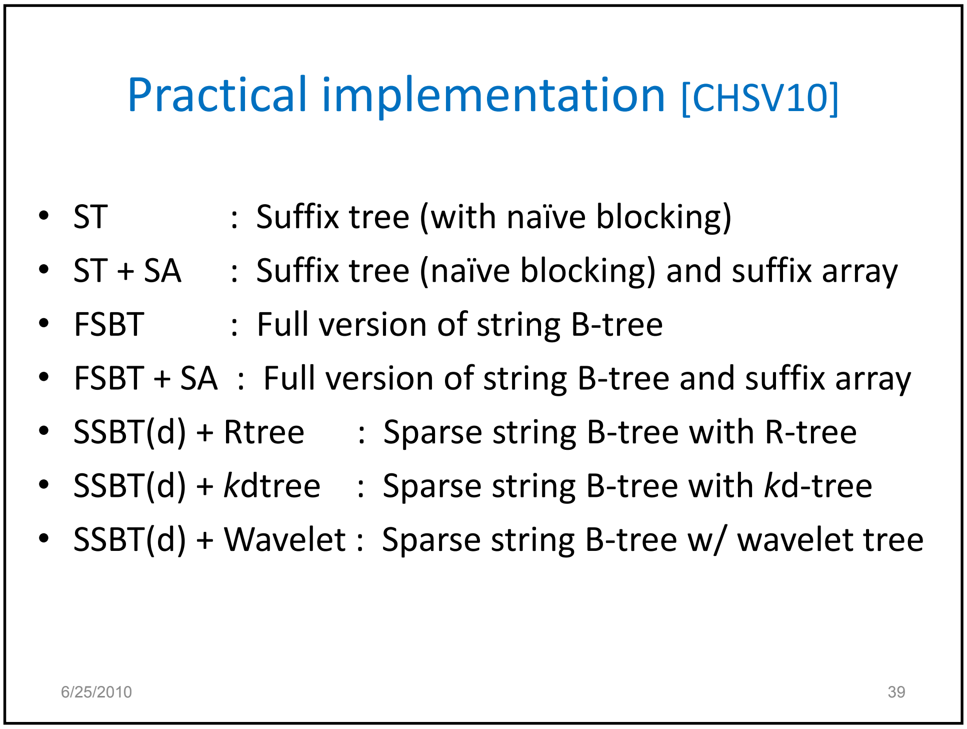
New Space and I/O bounds

- Space Improved
  - $O(n \log \Sigma)$ bits → $O(n H_k) + o(n \log |\Sigma|)$ bits
- I/O bound increased
  - $O(p / B + (\log_{|\Sigma|} n) \log_B n + \text{output} \log_B n)$ I/Os
    → $O(p (\log n) / B + (\log^3 n) / ((\log |\Sigma|) \log B) + \text{output} \log_B n)$ I/Os
- This can be further improved to
  $O(p / (B \log_{|\Sigma|} n) + (\log^4 n) / \log \log n + \text{output} \log_B n)$ I/Os
  - by using external memory sparse suffix tree with t-suffix links
    - t-suffix link is implemented using multiple applications of suffix links in CST
  - Slight increase in space term
    - $O(n H_k + n) + o(n \log |\Sigma|)$ bits



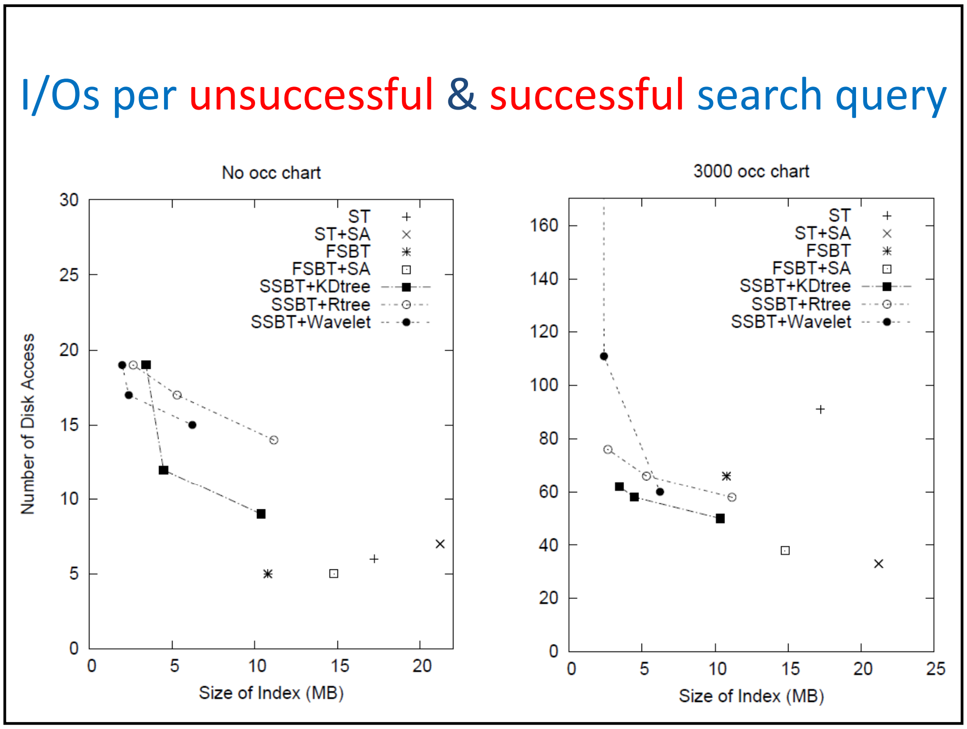
Related Work in External Memory

- 2007: External memory index by Gonzalez and Navarro [GN07]
  - counting: **$O(p)$** I/Os
  - reporting: **$O(\text{output} / B)$** I/Os (optimum)
  - **$O(H_k (\log (1/H_k)) n \log n)$** bits of space
- 2007: External memory index by Arroyuelo-Navarro uses LZ-indexes [AN07]
  - **$8 n H_k + o(n \log |\Sigma|)$** bits space
  - No theoretical bounds given for pattern searching
  - Practical index
  - 20–60 disk access when $p \approx 5$,
  - Need more I/Os as p increases

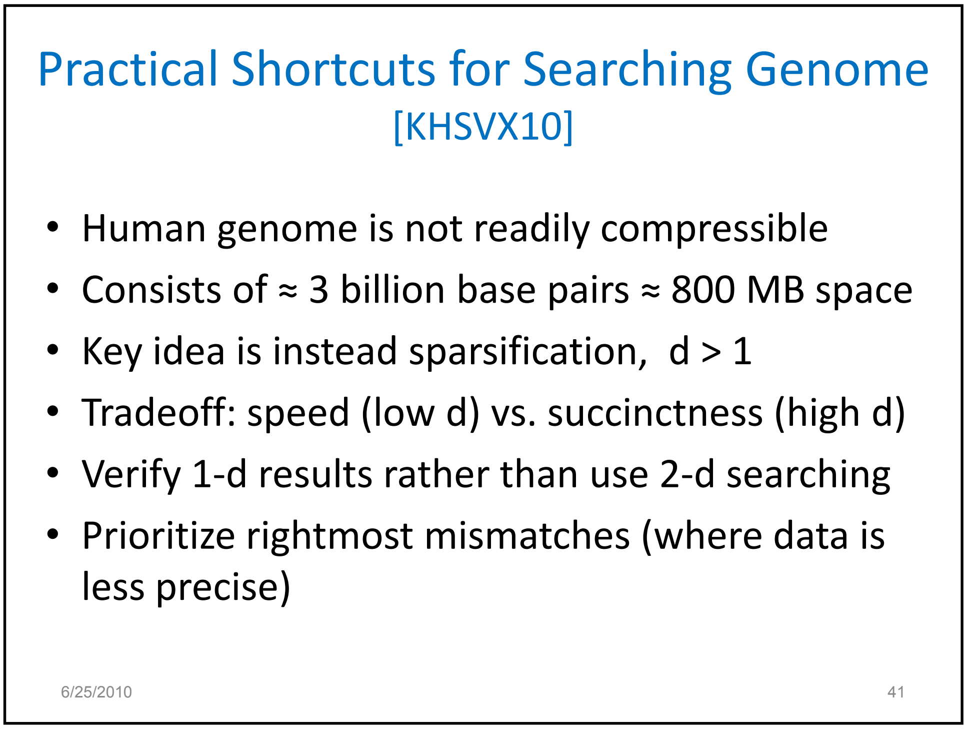


Practical implementation [CHSV10]

- ST : Suffix tree (with naïve blocking)
- ST + SA : Suffix tree (naïve blocking) and suffix array
- FSBT : Full version of string B-tree
- FSBT + SA : Full version of string B-tree and suffix array
- SSBT(d) + Rtree : Sparse string B-tree with R-tree
- SSBT(d) + *k*dtree : Sparse string B-tree with *k*d-tree
- SSBT(d) + Wavelet : Sparse string B-tree w/ wavelet tree

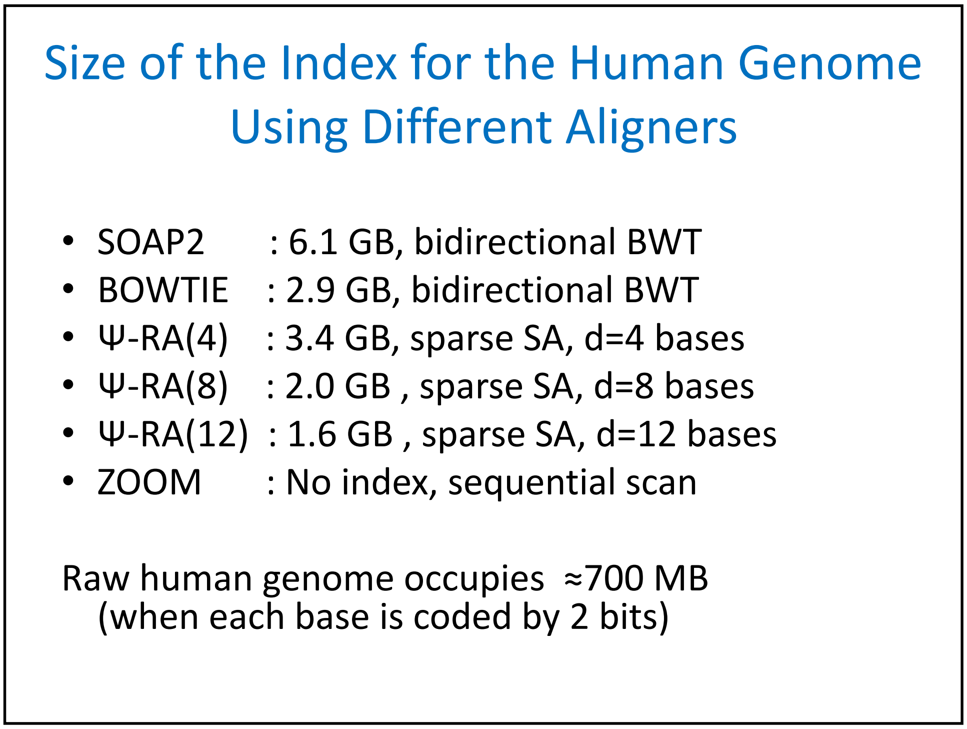


I/Os per unsuccessful & successful search query



Practical Shortcuts for Searching Genome [KHSVX10]

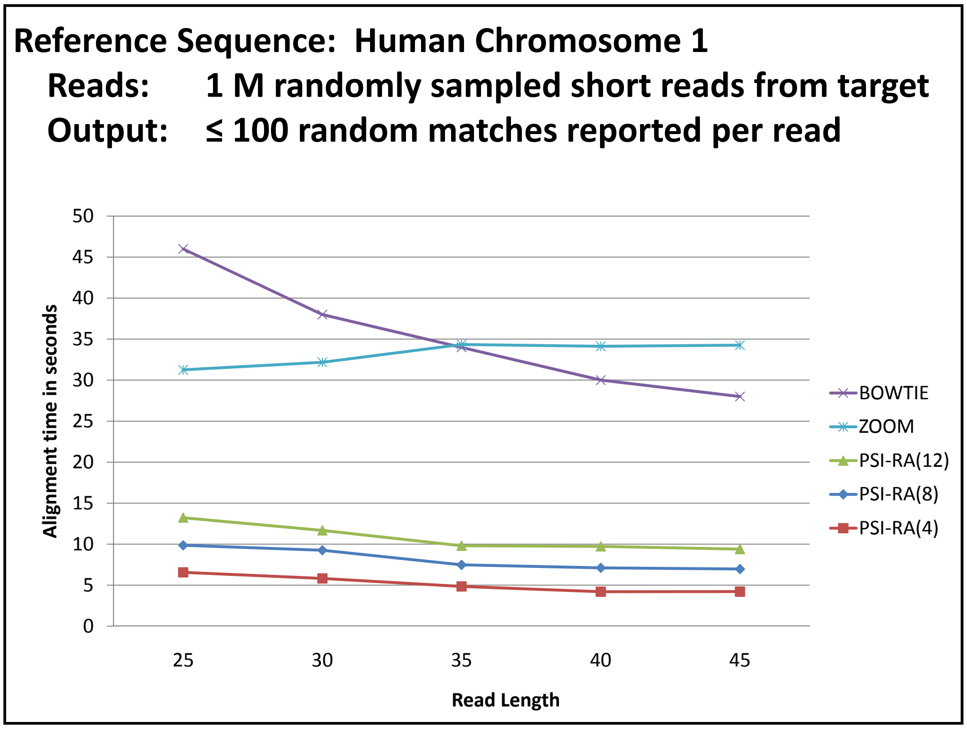
- Human genome is not readily compressible
- Consists of ≈ 3 billion base pairs ≈ 800 MB space
- Key idea is instead sparsification, d > 1
- Tradeoff: speed (low d) vs. succinctness (high d)
- Verify 1-d results rather than use 2-d searching
- Prioritize rightmost mismatches (where data is less precise)



Size of the Index for the Human Genome Using Different Aligners

• SOAP2	: 6.1 GB, bidirectional BWT
• BOWTIE	: 2.9 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
• ZOOM	: No index, sequential scan

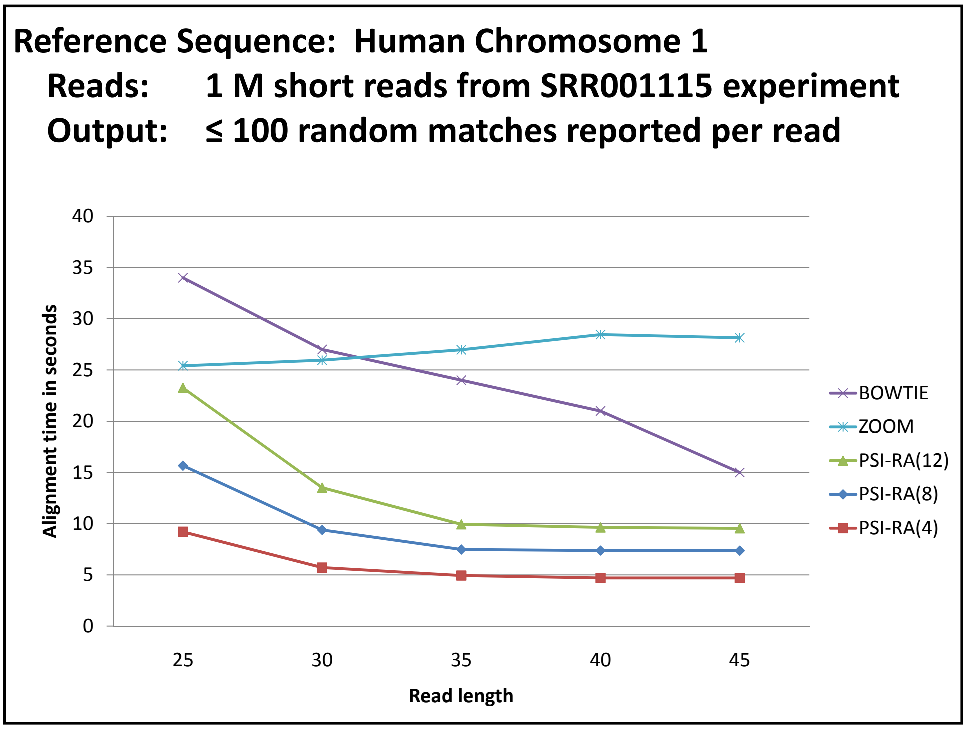
Raw human genome occupies ≈700 MB
(when each base is coded by 2 bits)



Reference Sequence: Human Chromosome 1

Reads: 1 M randomly sampled short reads from target

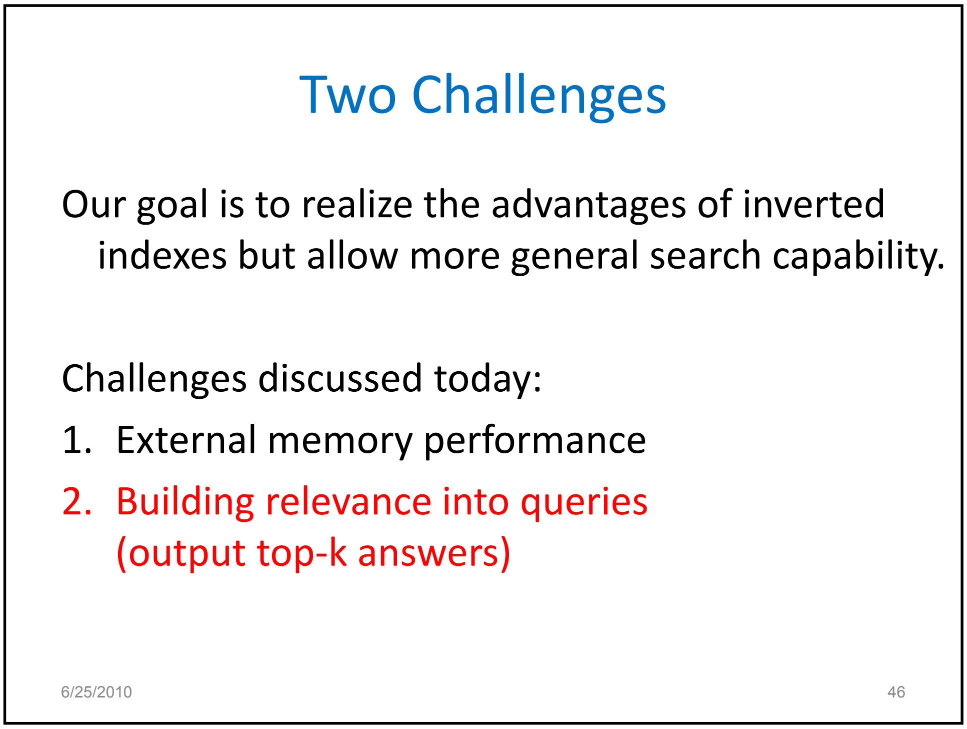
Output: ≤ 100 random matches reported per read



Reference Sequence: Human Chromosome 1

Reads: 1 M short reads from SRR001115 experiment

Output: ≤ 100 random matches reported per read

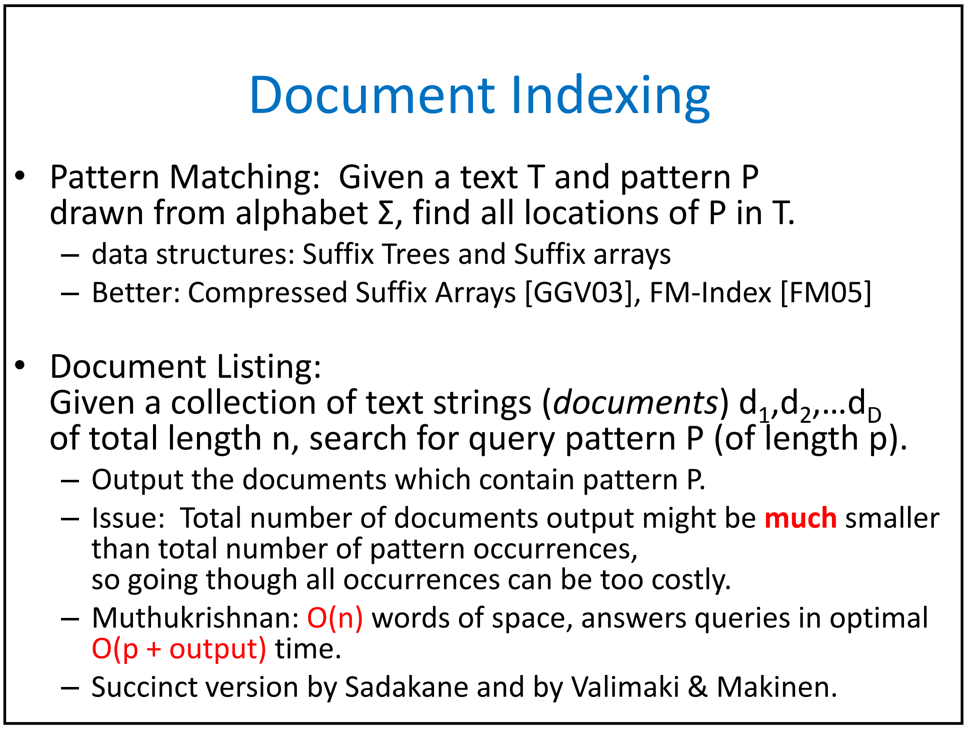


Two Challenges

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

Challenges discussed today:

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

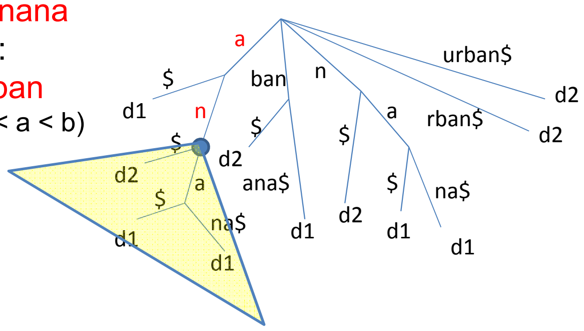


Document Indexing

- Pattern Matching: Given a text T and pattern P drawn from alphabet Σ, find all locations of P in T.
  - data structures: Suffix Trees and Suffix arrays
  - Better: Compressed Suffix Arrays [GGV03], FM-Index [FM05]
- Document Listing:
  Given a collection of text strings (*documents*) $d_1,d_2,\ldots d_D$ of total length n, search for query pattern P (of length p).
  - Output the documents which contain pattern P.
  - Issue: Total number of documents output might be **much** smaller than total number of pattern occurrences,
    so going though all occurrences can be too costly.
  - Muthukrishnan: O(n) words of space, answers queries in optimal O(p + output) time.
  - Succinct version by Sadakane and by Valimaki & Makinen.

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:
  d1 appears twice and d2 appears once in this subtree

Modified Problem—using Relevance

- Instead of listing all documents (strings) in which pattern occurs, list only highly ``relevant” documents.
  - Frequency: where pattern P occurs most frequently.
  - Proximity: where two occurrences of P are close to each other.
  - Importance: where each document has a static weight (e.g., Google’s PageRank).
- Threshold vs. Top-k
  - Thresholding: K-mine and K-repeats problem [Muthu 2002].
  - Top-k: Retrieve only the k most-relevant documents.
    - Intuitive for User

Approaches

- Inverted Indexes
  - Popular in IR community.
  - Do not efficiently answer arbitrary pattern queries.
- Muthukrishnan’s Structures (based on suffix trees)
  - Take O(n log n) words of space for K-mine and K-repeats problem (thresholding) while answering queries in O(p + output) time.
  - Top-k queries require additional overhead.

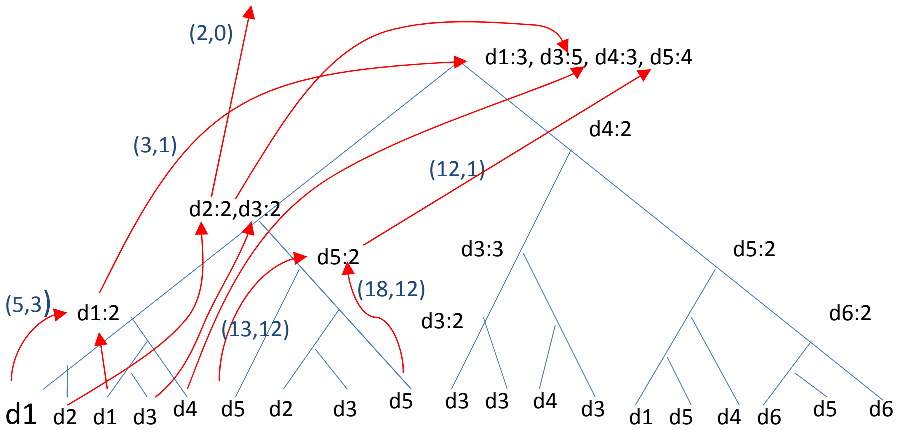
Preliminary : RMQs for top-k on array

- Range Maximum Query: Given an array A and query (i,j), report the maximum of A[i..j]
  - Linear space, linear preprocessing time DS with O(1) query time
- Range threshold: Given an array A, and a query (i,j,K), report all the numbers in A[i..j] which are >= K
  - Can be done using repeated RMQs in O(output) time
- Range top-k: Given an array A, and a query (i,j,k) report top-k highest numbers in A[i..j]
  - Repeated RMQs + heap = O(k log k) time
- Generalization: Given array A, and query specifies set of t ranges $[i_1,j_1]$, $[i_2,j_2]$,…$[i_t,j_t]$
  - Threshold : O(t +output) time; top-k : O(t + k log k) time

Our first framework

- Goal: Use only O(n) words of space.
- For a given query pattern P of length p, each document d gets a score(P, d) based upon the occurrences of P in d.
- Arbitrary score function allowed.
  - Examples: frequency, proximity, Google Page Rank are all captured in this framework.
- Answers the thresholding version in optimal time O(p+ output), improving space bound of Muthukrishnan.
- Answers top-k version (in sorted order) in O(p + k log k) time.
  - Does not need to look at all matching documents!

N-structure Augmenting Suffix Tree

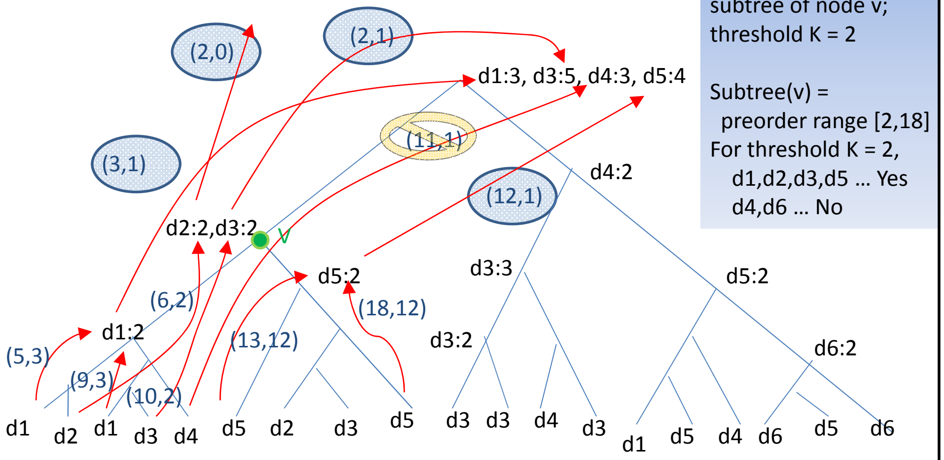


- N-structure N_v: At a node v, store an entry for document d_i if at least two children of v have d_i in their subtrees.
- The score of d_i at node v is # occurrences of d_i in the subtree.
- Link every entry for document d_i to the entry of d_i in its closest ancestor.
- Each link is annotated with preorder numbering of (origin, target).

(2,1,1)-range query

Query pattern P corresponds to the subtree of node v; threshold K = 2

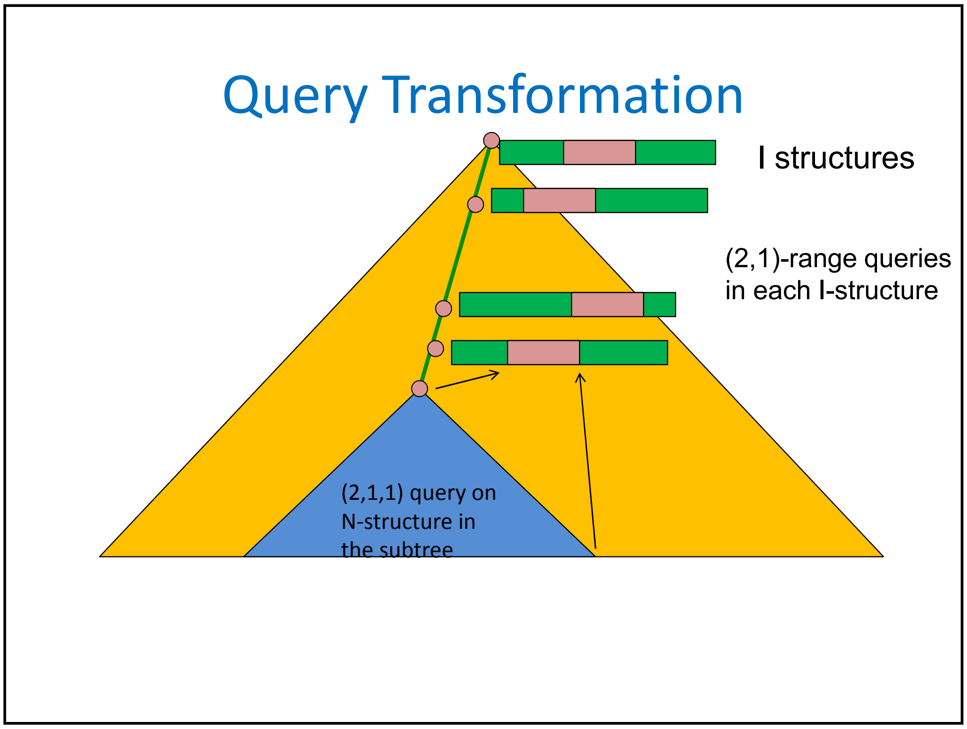
Subtree(v) =
preorder range [2,18]
For threshold K = 2,
d1,d2,d3,d5 … Yes
d4,d6 … No



- If the query matches up to node v in the suffix tree, then we need to focus on all the links with origin in Subtree(v) and target above Subtree(v).
  - This ensures each document is considered only once.
- Among these links, we need the links whose origin score value is greater than threshold K.
- Can be done via a (2,1,1)-query in 3-D

Main Idea !

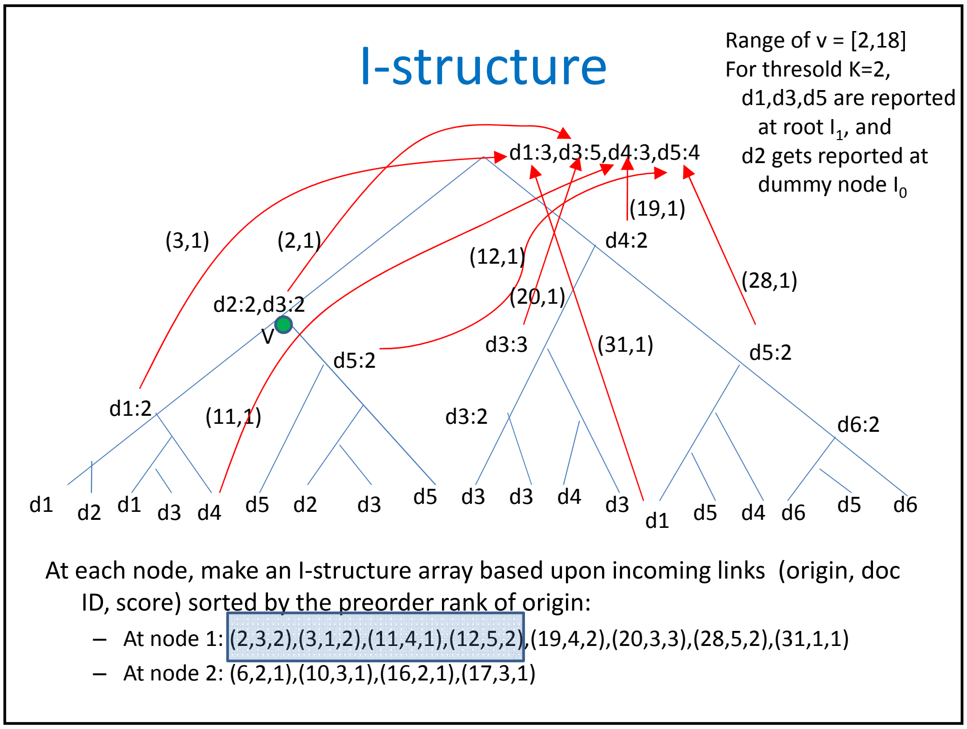
- Each link has 3 attributes (origin, target, origin_score)
- (2,1,1)-range query in 3D
  - Get all links with
    - Origin in [2,18] …. (subtree of v, the range where pattern matches)
    - Target value < 2 …. (enforces uniqueness of each document)
    - Origin score ≥ 2 …. (applies score threshold)
  - Best linear space structure takes O(output × log n) time to answer such a 3D range query—which means O(p + output × logn) time—too costly!
  - Our target is O(p + output) time.
- New Idea: # possible target values ≤ # ancestors of v ≤ p
  - So group the links by their target values and query each relevant group separately via a (2, 1)-range query in 2D.



Query Transformation

I structures

(2,1)-range queries in each I-structure



I-structure

Range of v = [2,18]
For thresold K=2,
d1,d3,d5 are reported at root I_1, and d2 gets reported at dummy node I_0

At each node, make an I-structure array based upon incoming links (origin, doc ID, score) sorted by the preorder rank of origin:

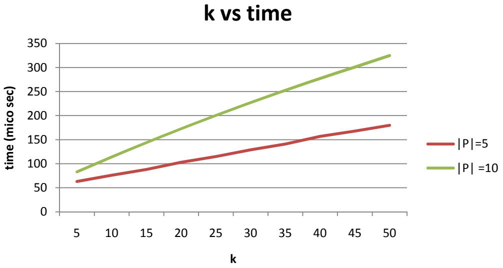
- At node 1: (2,3,2),(3,1,2),(11,4,1),(12,5,2),(19,4,2),(20,3,3),(28,5,2),(31,1,1)
- At node 2: (6,2,1),(10,3,1),(16,2,1),(17,3,1)

Space Analysis

- Number of entries in N-structures is ≤ 2n-1.
- So is the number of links.
- So is the number of entries in I-structures overall.
- Space for RMQ structures is linear in the size of data.
- Thus overall O(n) words of space.

Preliminary Experimental Results

Data size = 16 MB
Index size = 10 × input



Comparing different indexes for P=5 and K=10

Index	Index size	Query time (micro sec)
Wu/Muthu	250 × input	50
Culpepper et al	2.5 × input	700
Ours	10 × input	70

Times will be substantially larger if the data structure does not fit in internal memory.

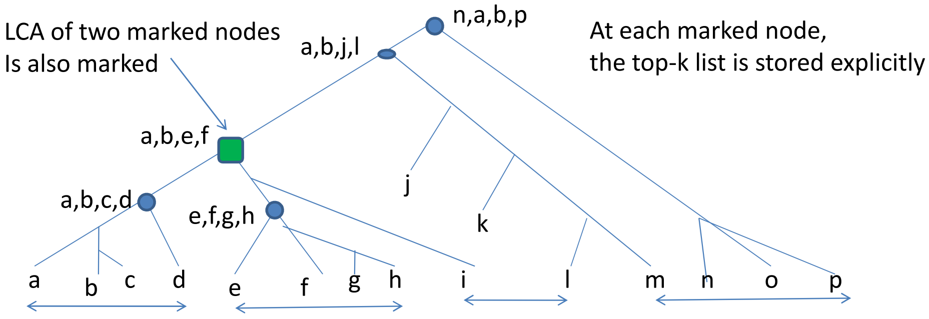
*

Compressed Data Structure

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

Sparsification example

Group consecutive $g = k \times \log^{2+\epsilon} n$ leaves and mark them. We build a CSA on the n/g bottom-level marked nodes.



LCA of two marked nodes Is also marked

At each marked node, the top-k list is stored explicitly

Example: Group size g = 4

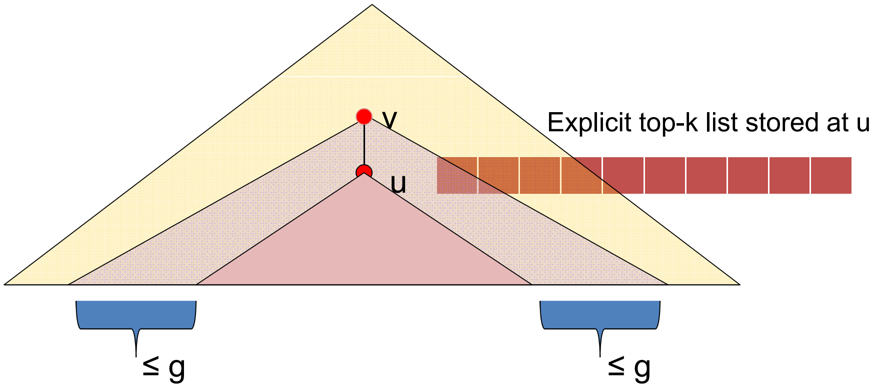
*

Sparsification Framework

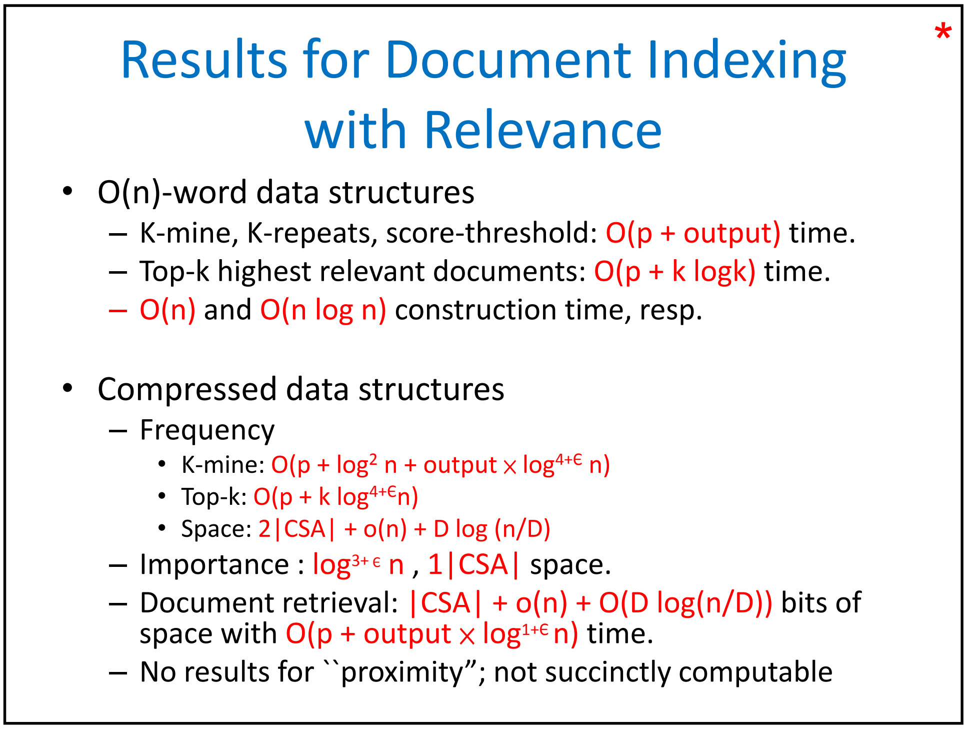
- First assume k (or K) is fixed, let group size $g = k \log^{2+\epsilon} n$.
- Take consecutive g leaves (in Suffix tree) from left to right and make them into groups. Mark the Least Common Ancestor of each group, and also Mark each node that is an LCA of two Marked nodes.
- Store explicit list of top-k highest scoring documents in Subtree(v) at each marked node v.
- Repeat this log n times: k = 1, 2, 4, 8, 16, ….
- Because of the sampling structure, the total space used is

$$O((n / k \log^{2+\epsilon} n) \times k \times \log n \times \log n) \text{ words}$$
$$= O(n / \log^{\epsilon} n) \text{ bits}$$
$$= o(n) \text{ bits}$$

Query Approach



Fringe leaves : ≤ 2g documents are separately queried for their for frequency counts

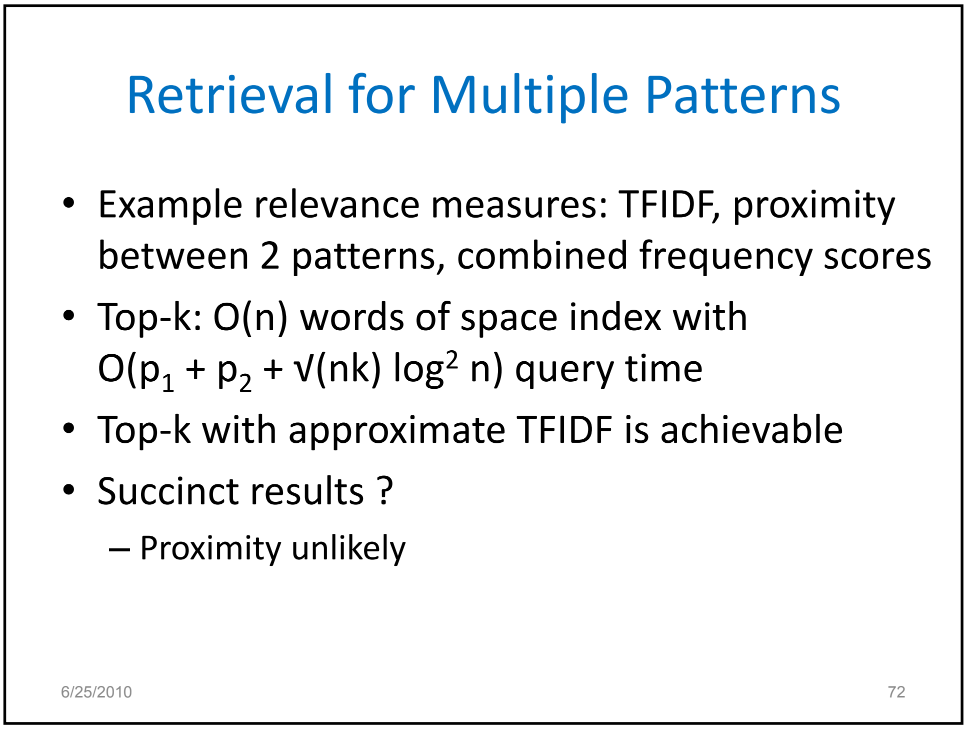


*

Results for Document Indexing with Relevance

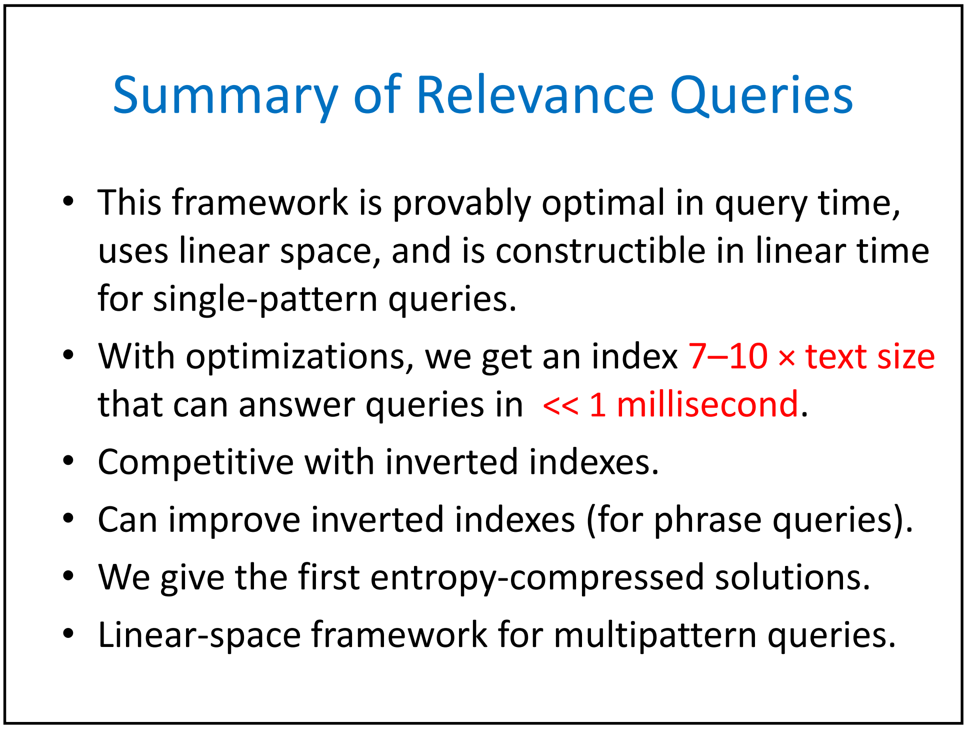
- O(n)-word data structures
  - K-mine, K-repeats, score-threshold: $O(p + \text{output})$ time.
  - Top-k highest relevant documents: $O(p + k \log k)$ time.
  - $O(n)$ and $O(n \log n)$ construction time, resp.

- Compressed data structures
  - Frequency
    - K-mine: $O(p + \log^2 n + \text{output} \times \log^{4+\epsilon} n)$
    - Top-k: $O(p + k \log^{4+\epsilon} n)$
    - Space: $2|CSA| + o(n) + D \log (n/D)$
  - Importance : $\log^{3+\epsilon} n$, $1|CSA|$ space.
  - Document retrieval: $|CSA| + o(n) + O(D \log(n/D))$ bits of space with $O(p + \text{output} \times \log^{1+\epsilon} n)$ time.
  - No results for ``proximity”; not succinctly computable



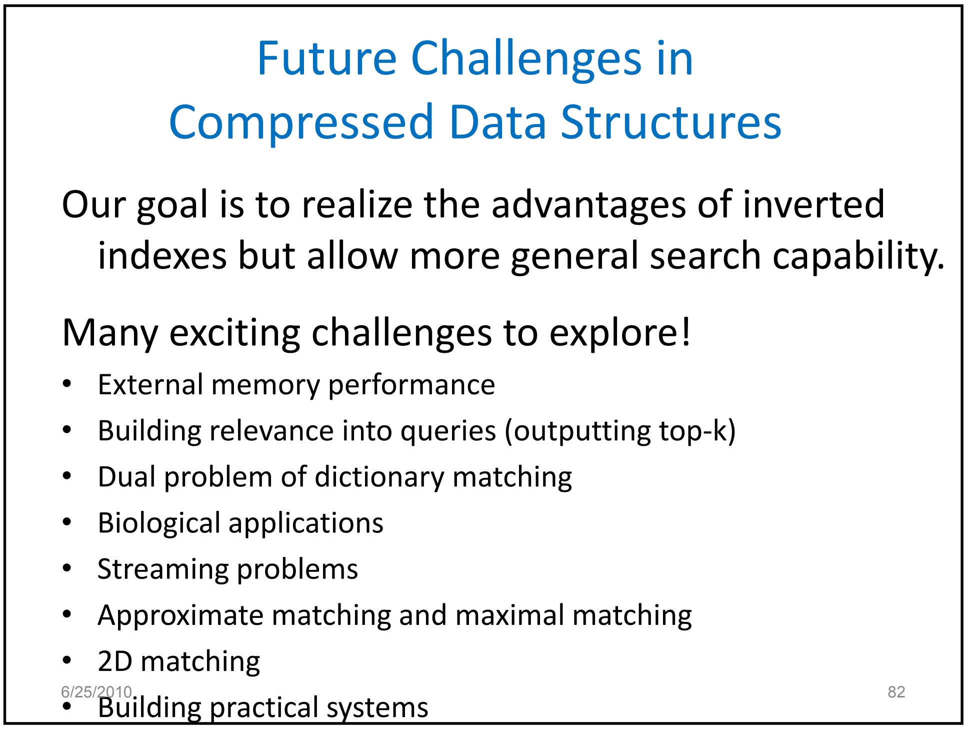
Retrieval for Multiple Patterns

- 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



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.



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!

- External memory performance
- Building relevance into queries (outputting top-k)
- Dual problem of dictionary matching
- Biological applications
- Streaming problems
- Approximate matching and maximal matching
- 2D matching
- Building practical systems