String algorithms and data structures

(or, tips and tricks for index design)

Paolo Ferragina

Università di Pisa, Italy

ferragina@di.unipi.it

An overview

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

Why string data are interesting ?

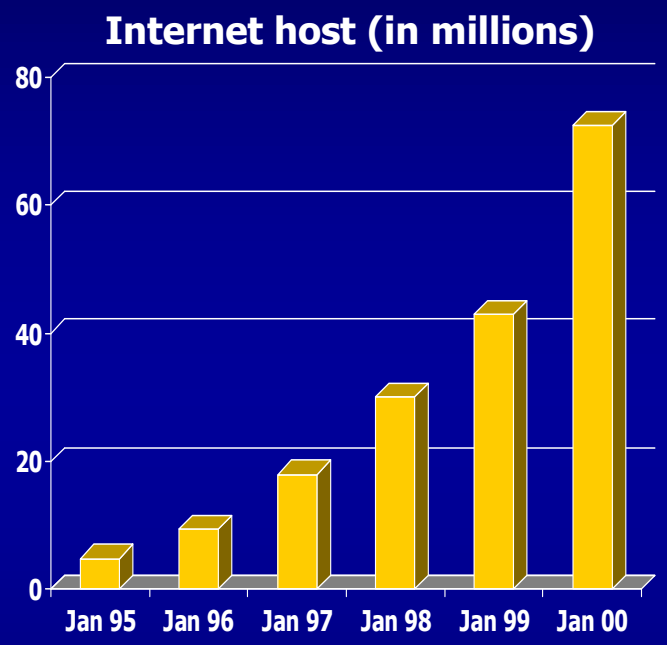
They are ubiquitous:

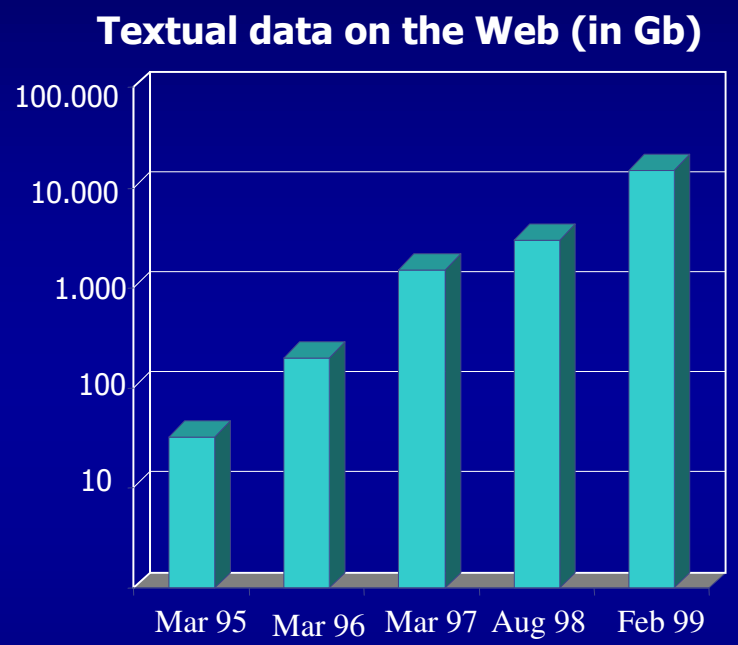
- Digital libraries and product catalogues
- Electronic white and yellow pages
- Specialized information sources (e.g. Genomic or Patent dbs)
- Web pages repositories
- Private information dbs
- ...

String collections are growing at a staggering rate:

- ...more than 10Tb of textual data in the web
- ...more than 15Gb of base pairs in the genomic dbs

Some figures





"Surface" Web: about 25÷50 Tb

- 2.5 billions of documents (7.3 millions per day)

"Deep" Web: about 7.500 Tb

- 4.200 Tb of interesting textual data

Mailing List: about 675 Tb (every year)

- 30 millions of msg per day, within 150,000 mailing lists

XML data storage (W3C project since '96)

An XML document is a simple piece of text containing some mark-up that is self-describing, follows some ground rules and is easily readable by humans and computers.

➔ It is text based and platform independent ➔

Great opportunity for IR…

Queries might exploit the tag structure to refine, rank and specialize the retrieval of the answers. For example:

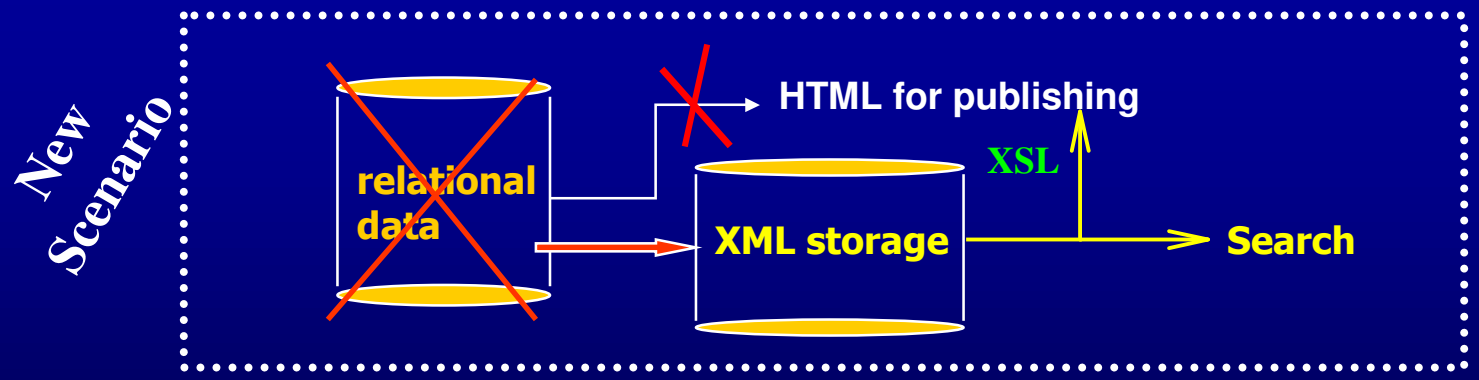
- Proximity may exploit tag nesting

  <author> John Red </author><author> Jan Green </author>

- Word disambiguation may exploit tag names

  <author> Brown … </author> <university> Brown … </university>

  <color> Brown … </color> <horse> Brown … </horse>



☑ XML structure is usually represented as a set of paths (strings?!?)

☑ XML queries are turned into string queries: /book/author/firstname/paolo

The need for an "index"

Brute-force scanning is not a viable approach:

- Fast single searches
- Multiple simple searches for complex queries

The American Heritage Dictionary defines index as follows

Anything that serves to guide, point out or otherwise facilitate reference, as:

(a) An alphabetized listing of names, places, and subjects included in a printed work that gives for each item the page on which it may be found;

(b) A series of notches cut into the edges of a book for easy access to chapters or other divisions;

(c) Any table, file or catalogue.

In computer science an index is a persistent data structure that allows to focus the search for a query string (or a set of them) on a provably small portion of the data collection.

What else ?

The index is a basic block of any IR system.

An IR system also encompasses:

- IR models
- Ranking algorithms
- Query languages and operations
- User-feedback models and interfaces
- Security and access control management
- ...

We will concentrate only on "index design" !!

Goals of the Course

- Learn about:
  - Model and framework for evaluating string data structures and algorithms on massive data sets
    - » External-memory model
    - » Evaluate the complexity of Construction and Query operations
  - Practical and theoretical foundations of index design
    - » The I/O-subsystem and other memory levels
    - » Types of queries and indexed data
    - » Space vs. time trade-off
    - » String transactions and index caching

    } **Dichotomy between**
    - **Word-based indexes**
    - **Full-text indexes**
  - Engineering and experiments on interesting indexes
    - » Inverted list vs. Suffix array, Suffix tree and String B-tree
    - » How to choreograph compression and indexing: the new frontier !

😐 MORAL: No clear winner among these data structures !!

Model and Framework

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

Why do we care of disks ?

In the last decade

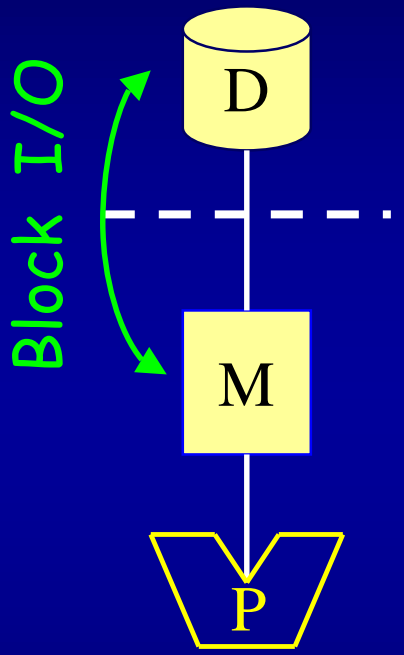
- Disk performance + 20% per year } Mechanical device
- Memory performance + 40% per year } Electronic devices
- Processor performance + 55% per year }

Current performance

Bandwidth	Disk SCSI	10 ÷ 80 Mb/s	3 ÷ 10 Mb/s in practice
	Disk ATA/EIDE	3 ÷ 33 Mb/s	
	Rambus memory	2 Gb/s	
Access time	Disk	≈ 7 millisecs	
	Memory	20 ÷ 90 nanosecs	
	Processor	few Ghz	

significant GAP between memory vs. disk performance

The I/O–model [Aggarwal–Vitter '88]



Model parameters

K= # strings in Δ's collection
N = total # of characters in strings
B = # chars per disk page
M = # chars fitting in internal memory

Model refinement

To take care of disk seek and bandwidth, we sometime distinguish between:

- Bulk I/Os: fetching cM contiguous data
- Random I/Os: any other type of I/O

Algorithmic complexity is therefore evaluated as:

- Number of random and bulk I/Os
- Internal running time (CPU time)
- Number of disk pages occupied by the index or during algorithm execution

Two families of indexes

Types of data

Linguistic or tokenizable text
Raw sequence of characters or bytes

DNA sequences
Audio-video files
Executables

Types of query

Word-based query
Character-based query

Exact word
Word prefix or suffix
Phrase

Arbitrary substring
Complex matches

Two indexing approaches :

- Word-based indexes, here a concept of "word" must be devised !
  - » Inverted files, Signature files or Bitmaps.
- Full-text indexes, no constraint on text and queries !
  - » Suffix Array, Suffix tree, Hybrid indexes, or String B-tree.

Word-based indexes

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

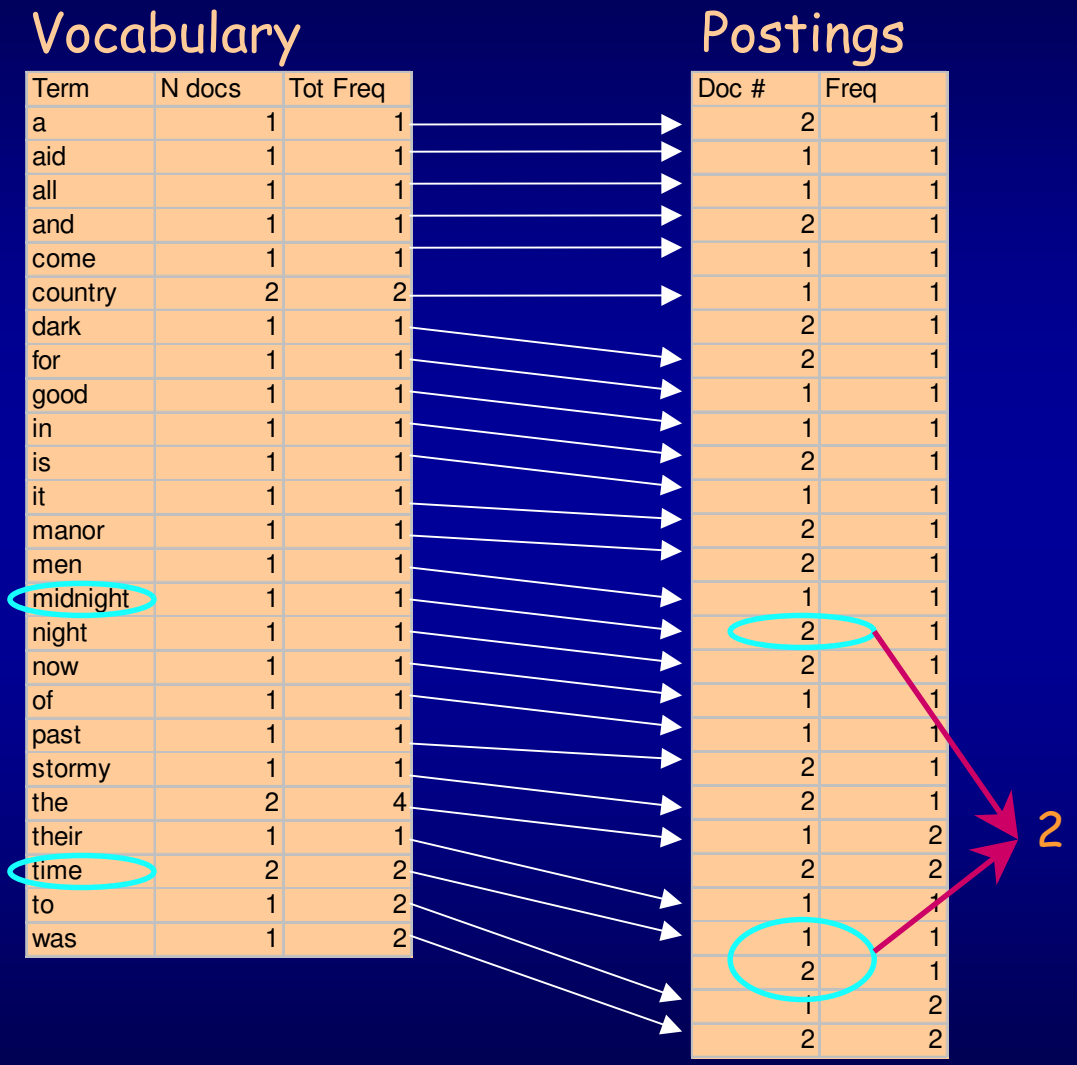
Inverted files (or lists)

Doc #1

Now is the time for all good men to come to the aid of their country

Doc #2

It was a dark and stormy night in the country manor. The time was past midnight



☑ Query answering is a two-phase process: midnight AND time

Some thoughts on the Vocabulary

- Concept of "word" must be devised
  - It depends on the underlying application
  - Some squeezing: normal form, stop words, stemming, ...
- Its size is usually small
  - Heaps' Law says $V = O(N^{\beta})$, where N is the collection size
  - β is practically between 0.4 and 0.6
- Implementation
  - Array: Simple and space succinct, but slow queries
  - Hash table: fast exact searches
  - Trie: fast prefix searches, but it is more complicated
  - Full-text index ?!? Fast complex searches.
- Compression ? Yes, speedup factor of two on scanning !!
  - Helps caching and prefetching
  - Reduces amount of processed data

Some thoughts on the Postings

- Granularity or accurancy in word location:
  - Coarse-grained: keep document numbers
  - Moderate-grained: keep the numbers of the text blocks
  - Fine-grained: keep word or sentence numbers

Space around 60%
Fast queries and precision

Space less than 20%
Slow queries: Post-filtering

- An orthogonal approach to space saving: Gap coding !!
  - Sort the postings for increasing document, block or term number
  - Store the differences between adjacent posting values (gaps)
  - Use variable-length encodings for gaps: γ-code, Golomb, ...

Continuation bit: given bin(x) = 10100|1000001 → 100 10100 01000001

It is byte-aligned, tagged, and self-synchronizing
Very fast decoding and small space overhead (~ 10%)

A generalization: Glimpse [Wu-Manber, 94]

- Text collection divided into blocks of fixed size b
  - A block may span two or more documents
  - Postings = block numbers

Fine-graned ← b → Coarse-grained

- Two types of space savings
  - Multiple occurrences in a block are represented only once
  - The number of blocks may be set to be small

  ⇨ Postings list is small, about 5% of the collection size

  ⇨ Under IR laws, space and query time are o(n) for a proper b

- Query answering is a three-phase process:
  - Query is matched against the vocabulary: word matchings
  - Postings lists of searched words are combined: candidate blocks
  - Candidate blocks are examined to filter out the false matches

Vocabulary turns complex text searches into exact block searches

Full-scan or succinct index ?

Other issues and research topics...

- Index construction
  - Create doc-term pairs < d,t > sorted by increasing d;
  - Mergesort on the second component t;
  - Build Postings lists from adjacent pairs with equal t.

  ⇨ In-place block permuting for page-contiguous postings lists.
- Document numbering
  - Locality in the postings lists improves their gap-coding
  - Passive exploitation: Integer coding algorithms
  - Active exploitation: Reordering of doc numbers [Blelloch *et al.*, 02]
- XML "native" indexing
  - Tags and attributes indexed as terms of a proper vocabulary
  - Tag nesting coded as set of nested grid intervals

  ⇨ Structural queries turned into boolean and geometric queries !

☑ Our project: XCDE Library, compression + indexing for XML !!

DBMS and XML

- Main idea:
  - Represent the document tree via tuples or set of objects;
  - *Select-from-where* clause to navigate into the tree;
  - Query engine use standard *join* and *scan;*
  - Some additional indexes for special accesses;

- **Advantages:**
  - **Standard DB engines can be used without migration;**
  - **OO easily holds a tree structure;**
  - **Query language is well known: SQL or OQL;**
  - **Query optimiser well tuned;**

DBMS and XML (2 of 2)

- General disadvantages:
  - Query navigation is costly, simulated via many joins;
  - Query optimiser looses knowledge on XML nature of the document;
  - Fields in tables or OO should be small;
  - Need extra indexes for managing effective *path* queries
- **Disadvantages in the relational case:** **(Oracle 8i/9i)**
  - **Impose a rigid and regular structure via tables;**
  - **Number of tables is high and much space is wasted;**
  - **Do exist translation methods but error-prone and DTD is needed.**
- **Disadvantages in the OO case:** **(Lore at Stanford university)**
  - **Objects are space expensive, many OO features unused;**
  - **Management of large objects is costly, hence search is slow.**

XML native storage

The literature offers various proposals:

- Xset, Bus: build a DOM tree in main memory at query time;
- XYZ-find: B-tree for storing pairs <path,word>;
- Fabric: Patricia tree for indexing *all* possible paths;
- Natix: DOM tree is partitioned into disk pages (see e.g. Xyleme);
- TReSy: String B-tree → large space occupancy;
- Some commercial products: Tamino,… (no details !)

Three interesting issues…

1. Space occupancy is usually not evaluated (surely it is ≥ 3) !
2. **Data structures and algorithms forget *known* results !**
3. **No software in the form of a library for public use !**

XCDE Library: Requirements

- XML documents may be:
  - strongly textual (e.g. linguistic texts);
  - only well-formed and may occur without a DTD;
  - arbitrarily nested and complicated in their tag structure;
  - retrievable in their original form (for XSL, browsers,...).
- **The library should offer:**
  1. **Minimal space occupancy (Doc + Index ~ original doc size);**
     ⇒ space critical applications: e.g. e-books, Tablets, PDAs !
  2. **State-of-the-art algorithms and data structures;**
  3. **XML native storage for full control of the performance;**
  4. **Flexibility for extensions and software development.**

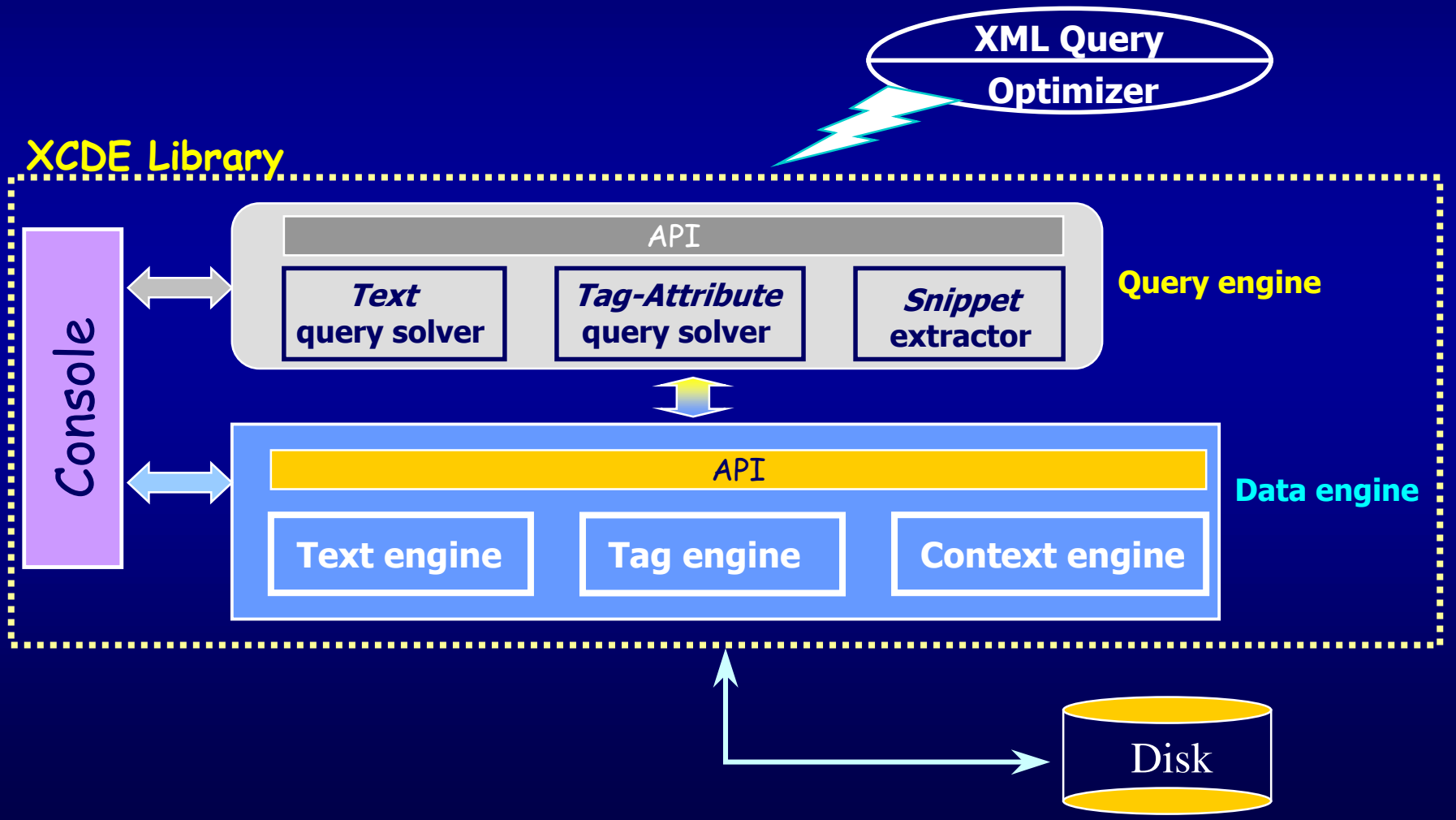
XCDE Library: Design Choices

- Single document indexing:
  - Simple software architecture;
  - Customizable indexing on each file (they are heterogeneous);
  - Ease of management, update and distribution;
  - *Light* internal index or *Blocking via XML tagging* to speed up query;

- **Full-control over the document content:**
  - **Approximate or Regexp match on text or attribute names and values;**
  - **Partial path queries, e.g. //root_tag//tag1//tag2, with distance;**

- **Well-formed snippet extraction:**
  - **for rendering via XSL, Braille, Voice, OEB e-books, …**

XCDE Library: The structure



Full–text indexes

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

The prologue

Their need is pervasive:

- Raw data: DNA sequences, Audio-Video files, ...
- Linguistic texts: data mining, statistics, ...
- Vocabulary for Inverted Lists
- Xpath queries on XML documents
- Intrusion detection, Anti-viruses, ...

Four classes of indexes:

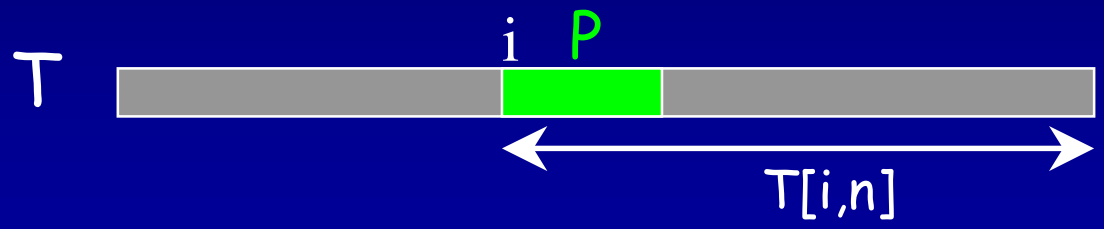
1. Suffix array or Suffix tree
2. Two-level indexes: Suffix array + in-memory Supra-index
3. B-tree based data structures: Prefix B-tree
4. String B-tree: B-tree + Patricia trie

Our lecture consists of a tour through these tools !!

Basic notation and facts

Pattern P[1,p] occurs at position i of T[1,n]

iff P[1,p] is a prefix of the suffix T[i,n]



Occurrences of P in T = All suffixes of T having P as a prefix

T = This is a visual example
This is a visual example
This is a visual example

} 3,6,12

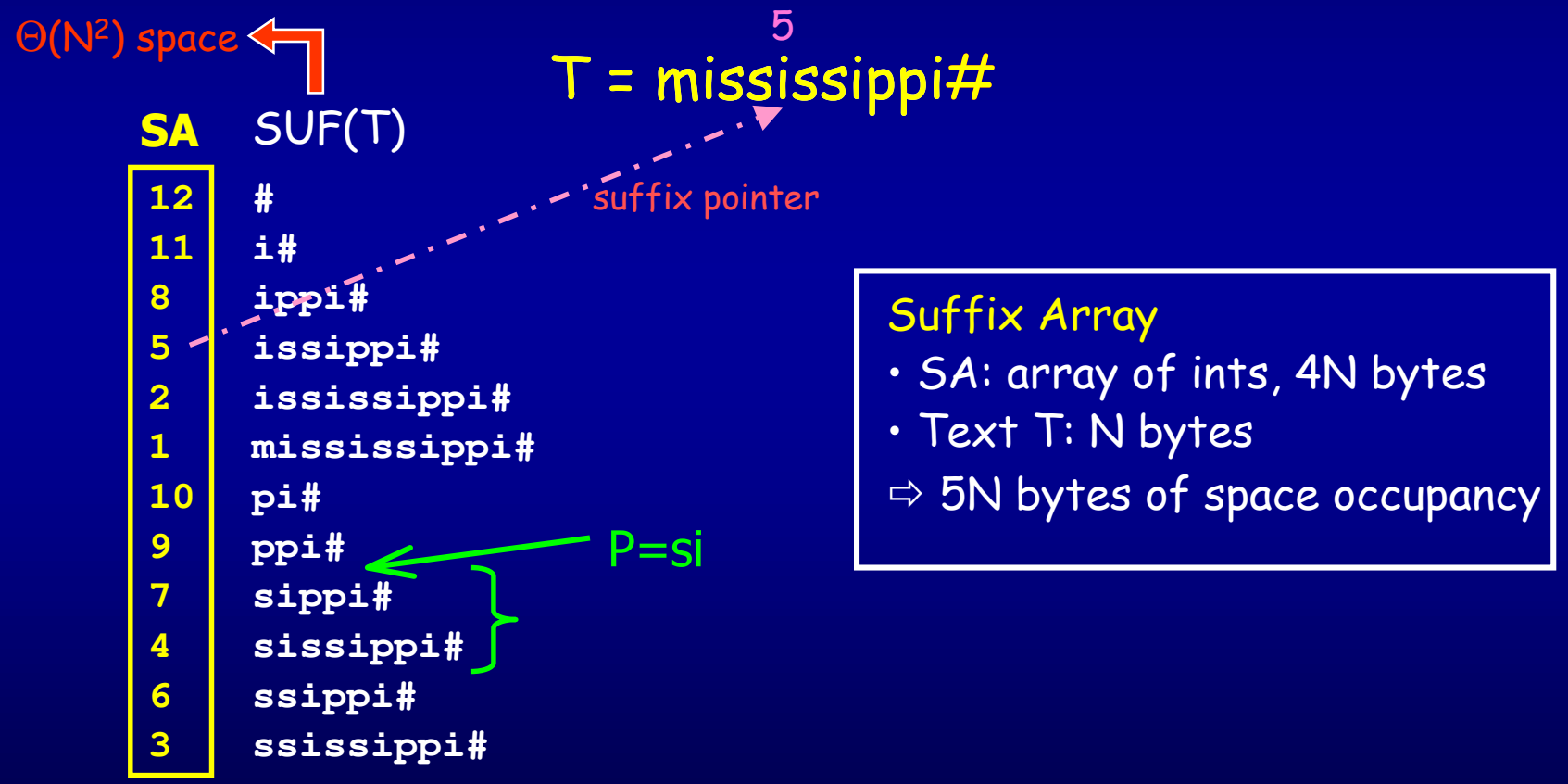
SUF(T) = Sorted set of suffixes of T

SUF(Δ) = Sorted set of suffixes of all texts in Δ

Two key properties [Manber-Myers, 90]

Prop 1. All suffixes in SUF(T) having prefix P are contiguous.

Prop 2. Starting position is the lexicographic one of P.



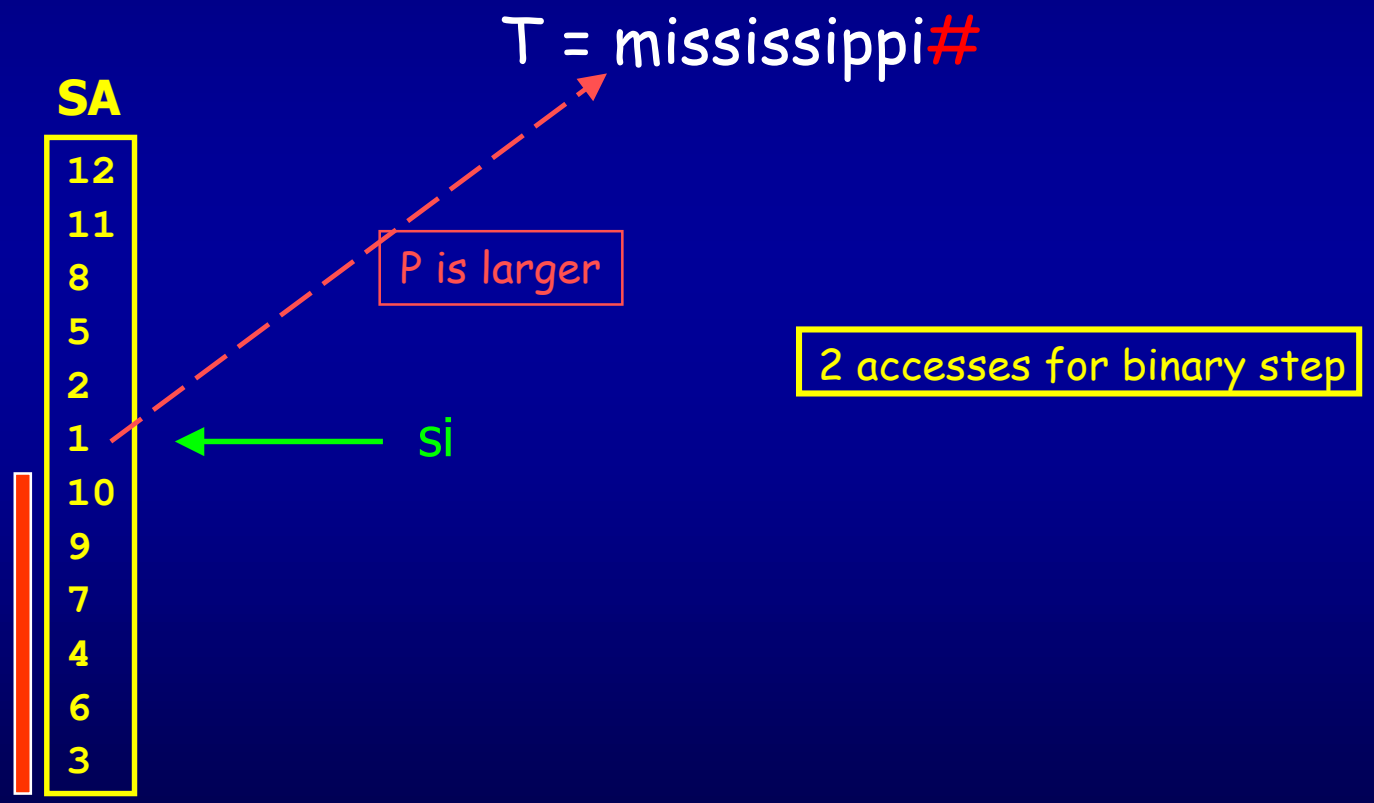
Suffix Array

- SA: array of ints, 4N bytes
- Text T: N bytes

⇨ 5N bytes of space occupancy

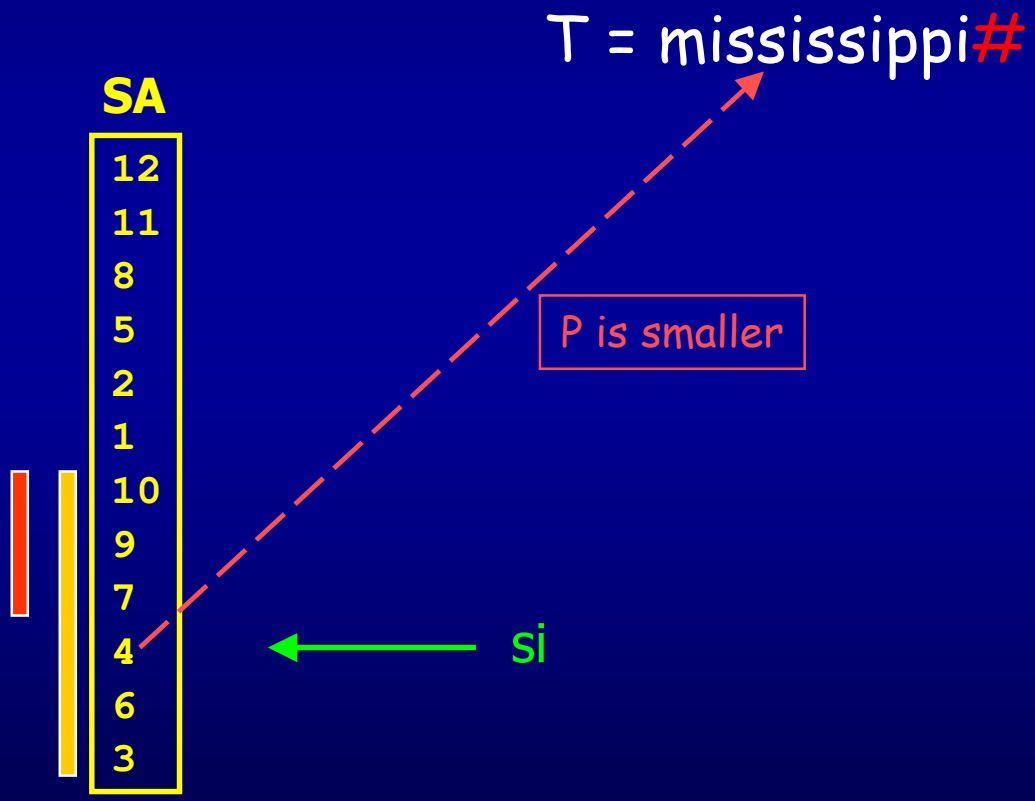
Searching in Suffix Array [Manber-Myers, 90]

Indirected binary search on *SA*: $O(p \log_2 N)$ time



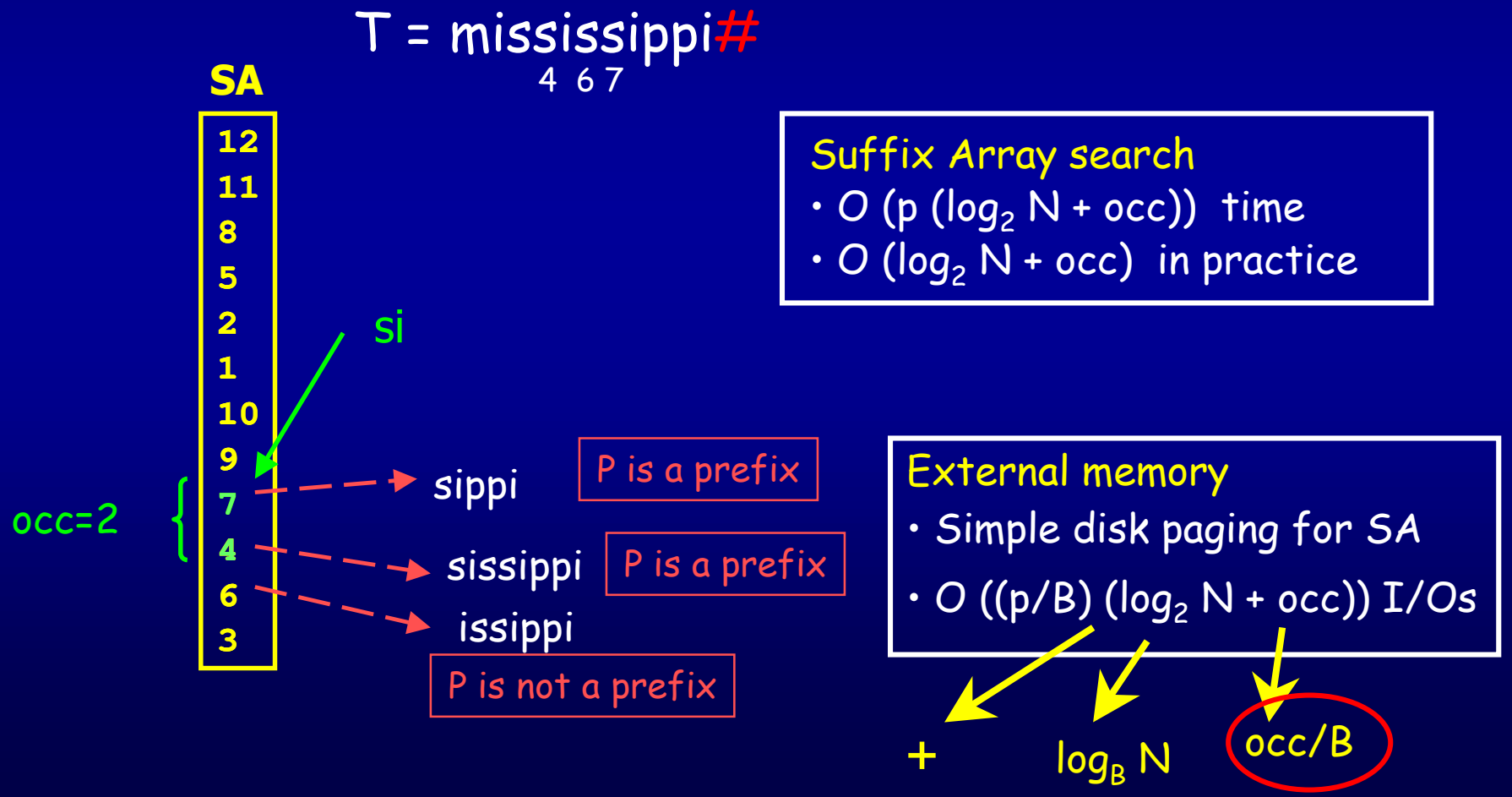
Searching in Suffix Array [Manber-Myers, 90]

Indirected binary search on *SA*: $O(p \log_2 N)$ time



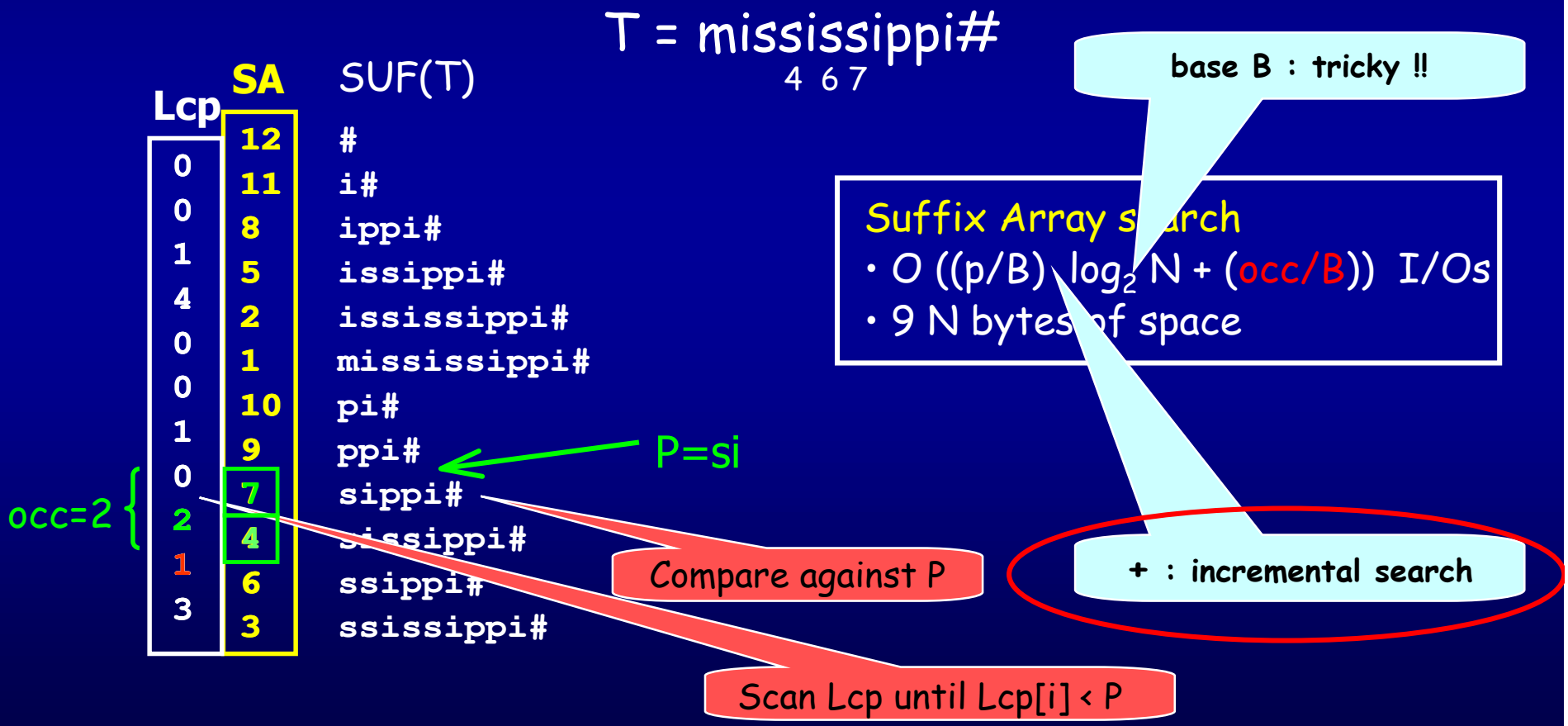
Listing the occurrences [Manber–Myers, 90]

Brute-force comparison: $O(p \times occ)$ time



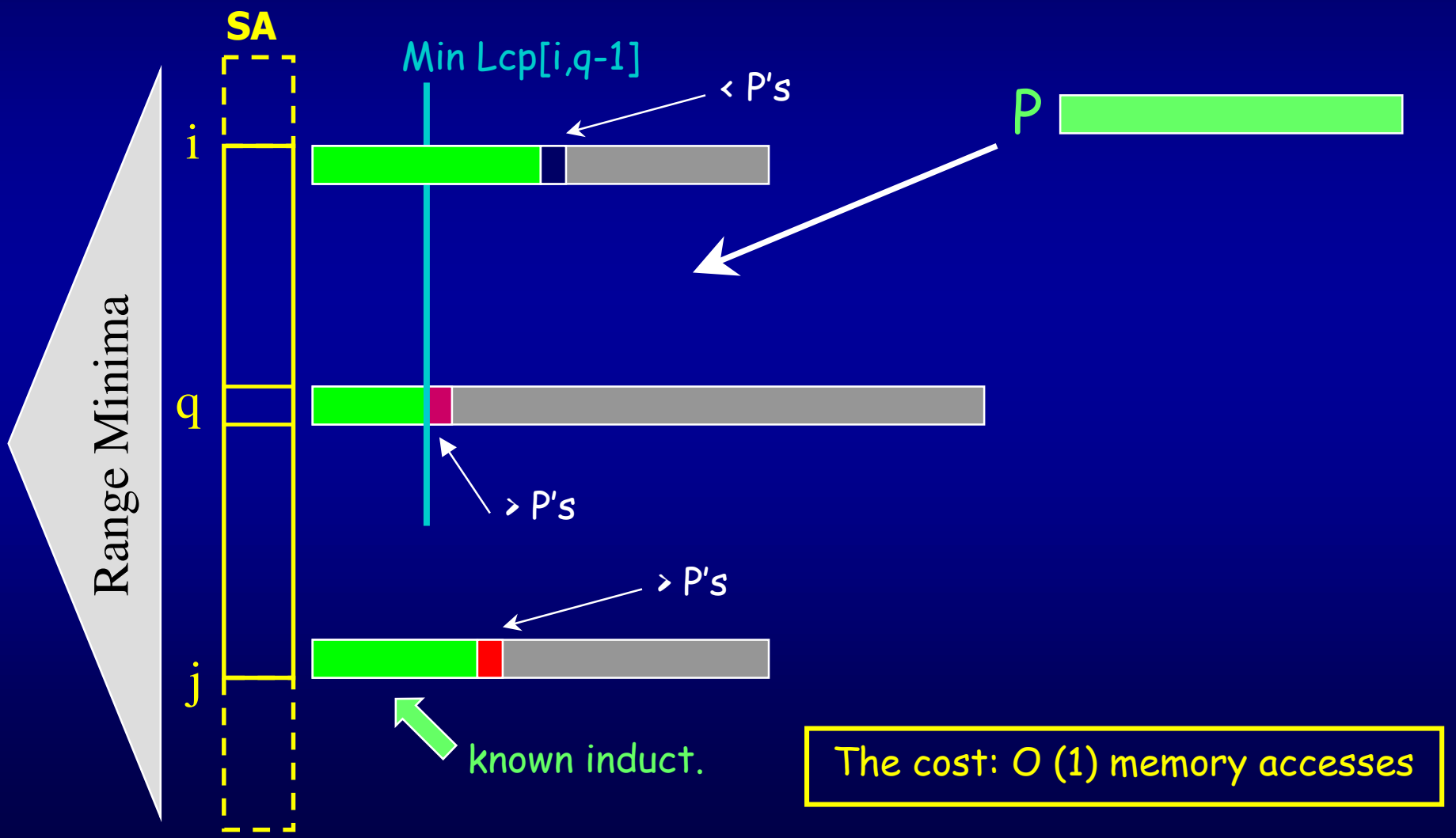
Output-sensitive retrieval

Lcp[1,n-1] stores the longest-common-prefix between suffixes adjacent in SA



Incremental search (case 1)

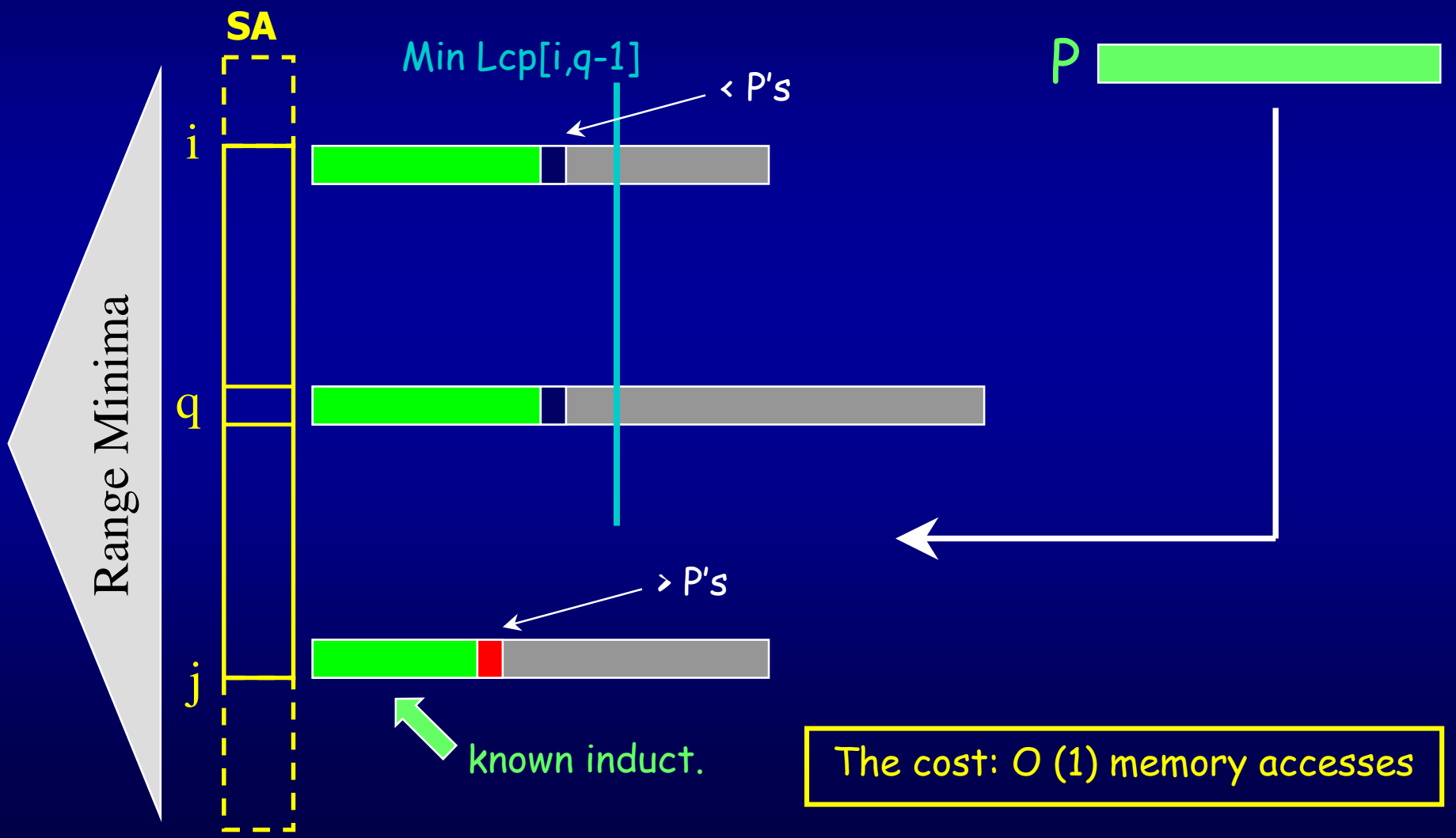
Incremental search using the LCP array: no rescanning of pattern chars



The cost: O (1) memory accesses

Incremental search (case 2)

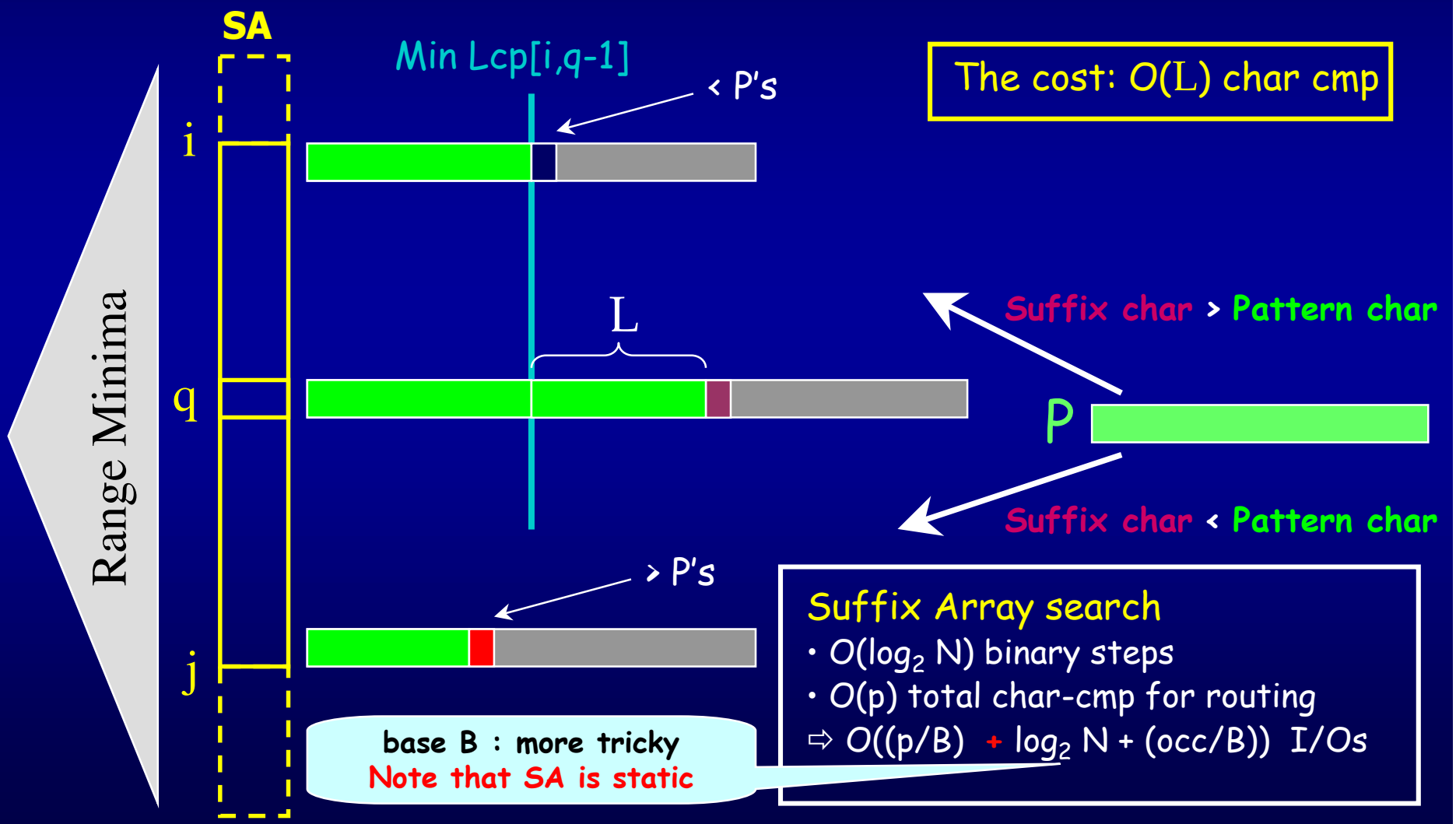
Incremental search using the LCP array: no rescanning of pattern chars



The cost: O (1) memory accesses

Incremental search (case 3)

Incremental search using the LCP array: no rescanning of pattern chars



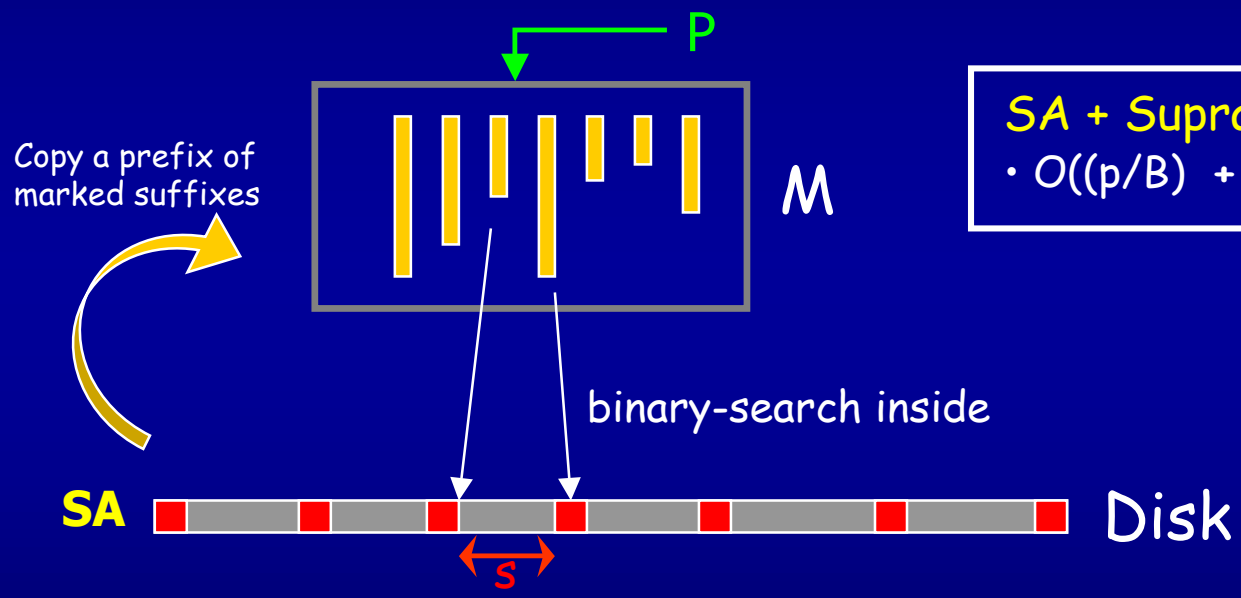
The cost: O(L) char cmp

Suffix Array search
- $O(\log_2 N)$ binary steps
- $O(p)$ total char-cmp for routing

⇨ $O((p/B) + \log_2 N + (occ/B))$ I/Os

Hybrid Index

Exploit internal memory: sample the suffix array and copy something in memory



SA + Supra-index

- $O((p/B) + \log_2 (N/s) + (occ/B))$ I/Os

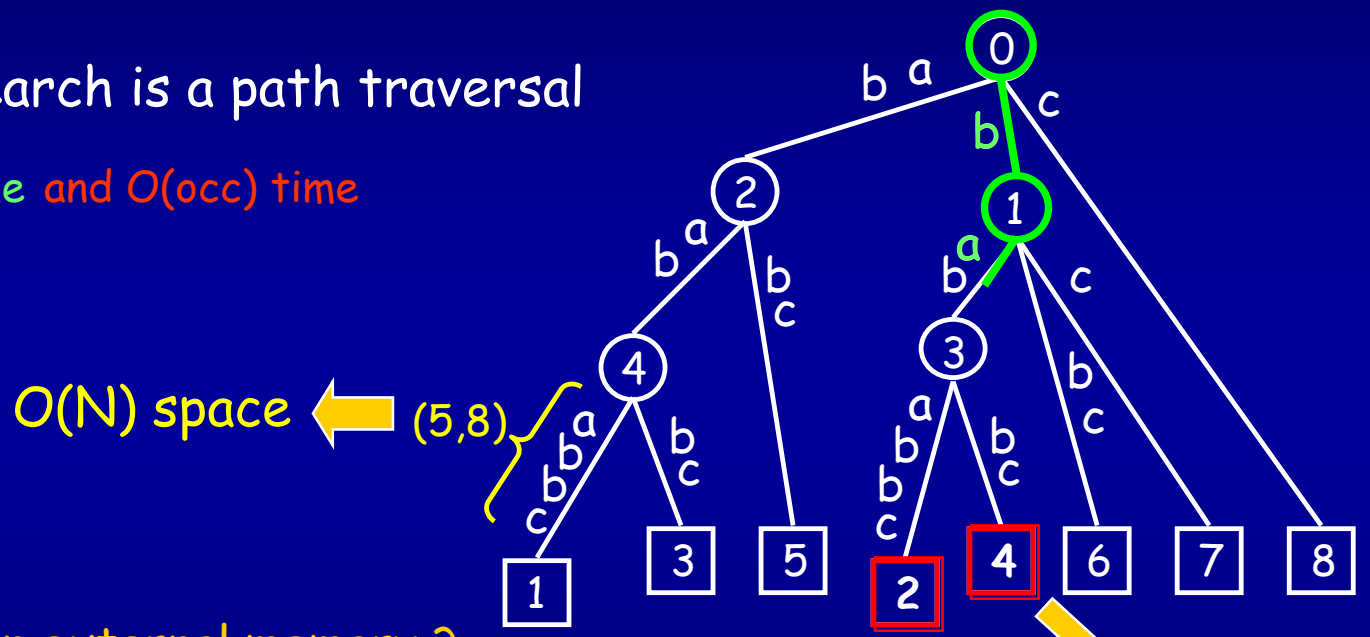
☑ Parameter s depends on M and influences both performance and space !!

The suffix tree [McCreight, '76]

It is a compacted trie built on all text suffixes

P = ba → Search is a path traversal

↳ O(p) time and O(occ) time



O(N) space

What about ST in external memory ?

- Unbalanced tree topology
- Dinamicity

} Packing ?!

- Large space ~ 15N

} CPAT tree ~ 5N on average

T = abababbca

1 3 5 7 9

Ω(p) I/Os

Ω(occ) I/Os??

No (p/B), possibly no (occ/B), mainly static and space costly

The String B-tree

(An I/O-efficient full-text index !!)

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

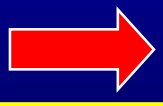
The prologue

We are left with many open issues:

- Suffix Array: dinamicity
- Suffix tree: difficult packing and $\Omega(p)$ I/Os
- Hybrid: Heuristic tuning of the performance

B-tree is ubiquitous in large-scale applications:

- Atomic keys: integers, reals, ...
- Prefix B-tree: bounded length keys ($\leq$ 255 chars)

Suffix trees + B-trees ?  String B-tree [Ferragina-Grossi, 95]

☑ Index unbounded length keys
☑ Good worst-case I/O-bounds in search and update
☑ Guaranteed optimal page-fill ratio

Some considerations

Strings have arbitrary length:

- Disk page cannot ensure the storage of $\Theta(B)$ strings
- M may be unable to store even one single string

String storage:

- Pointers allow to fit $\Theta(B)$ strings per disk page
- String comparison needs disk access and may be expensive

String pointers organization seen so far:

❶ Suffix array: simple but static and not optimal

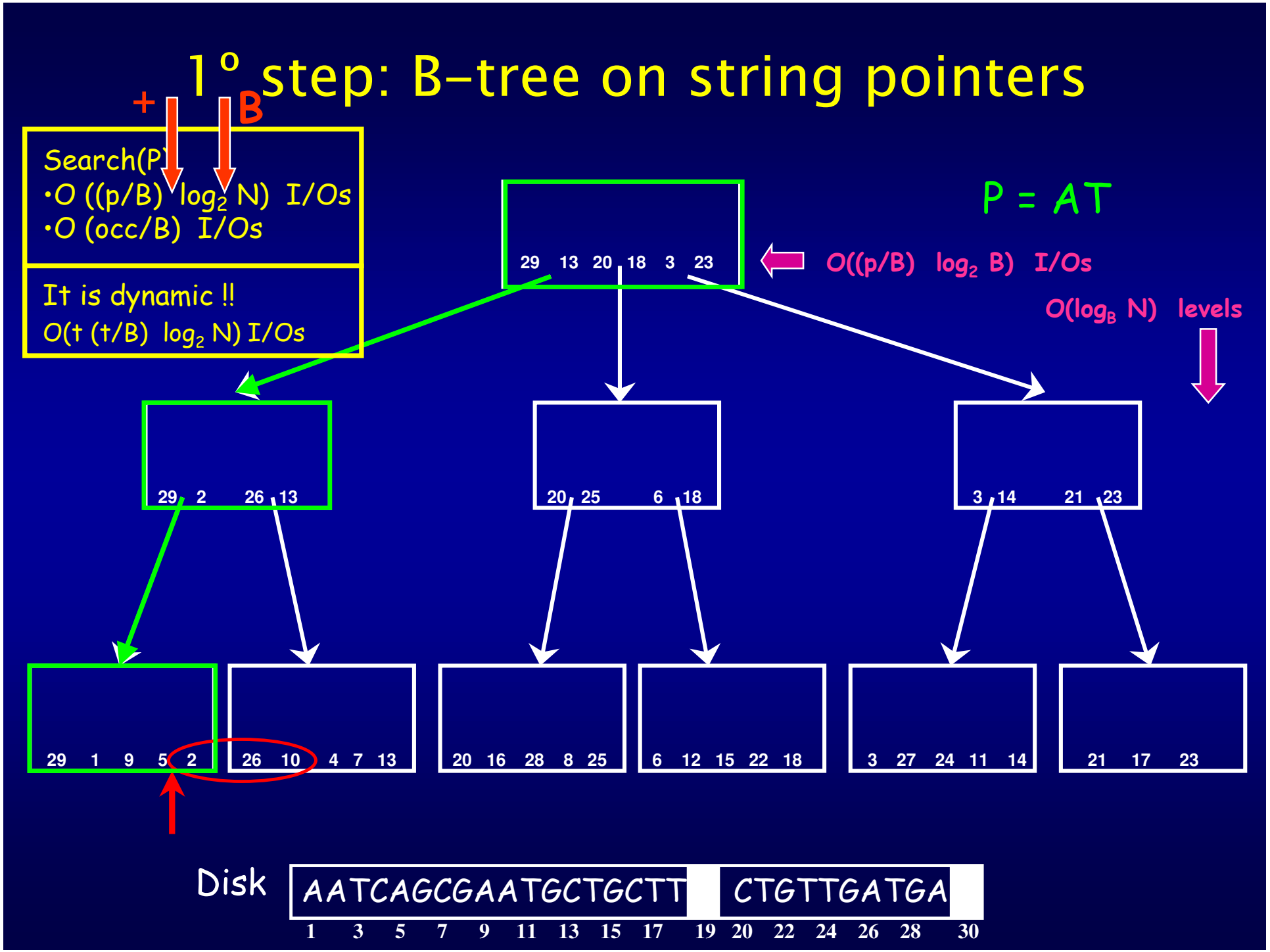
❷ Patricia trie: sophisticated and much efficient (optimal ?)

Recall the problem: Δ is a text collection

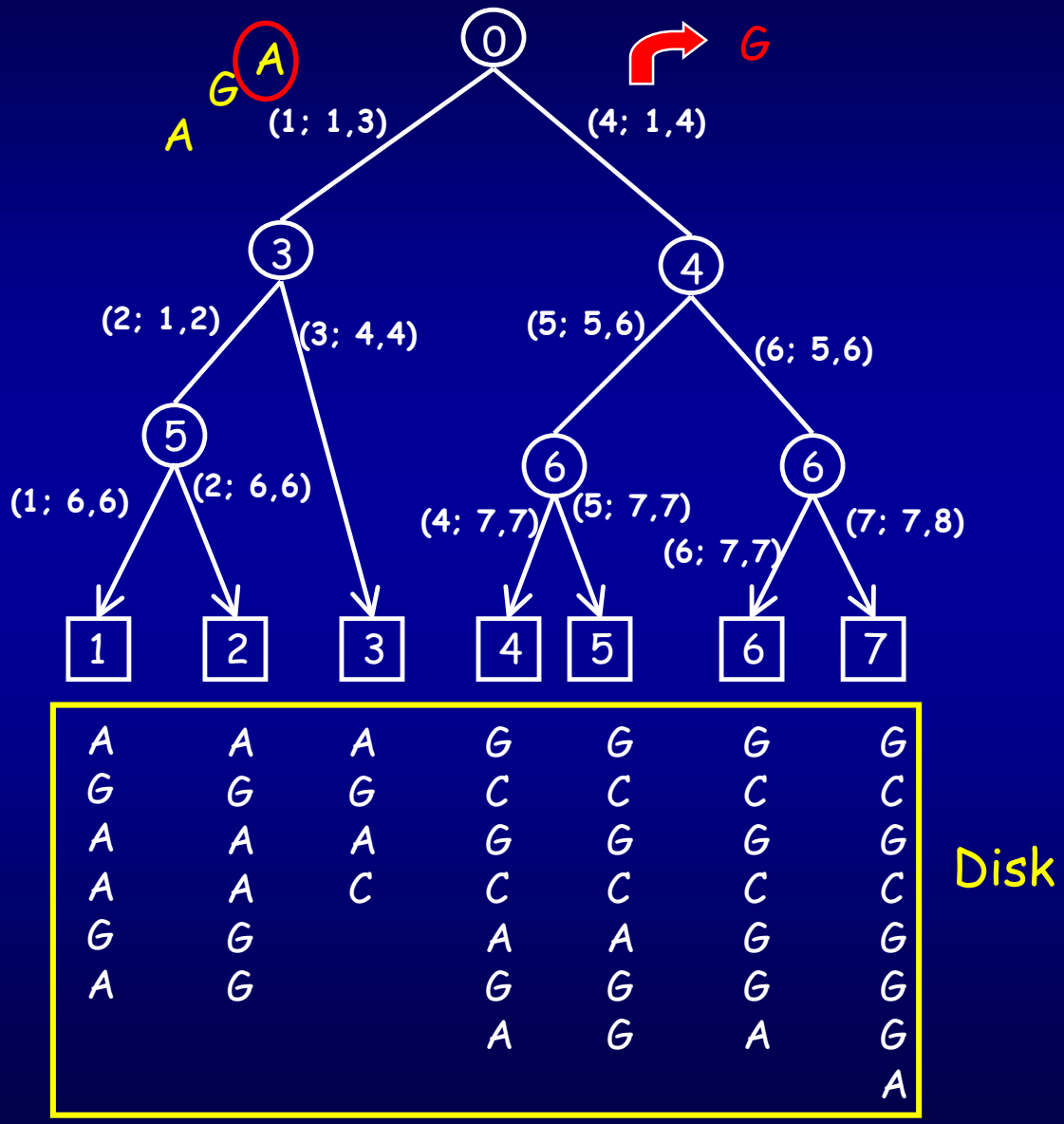
✓ Search(P[1,p]): retrieve all occurrences of P in Δ's texts

✓ Update(T[1,t]): insert or delete a text T from Δ

1° step: B-tree on string pointers



2° step: The Patricia trie

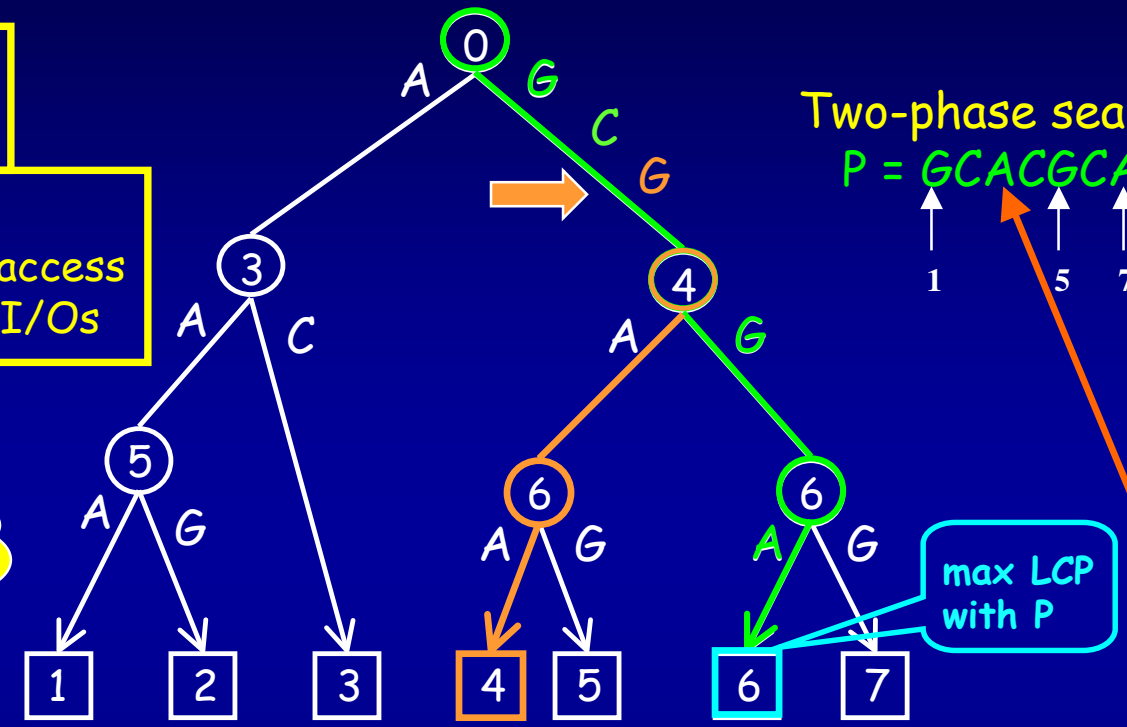


2° step: The Patricia trie

Space PT
- O(k) , no O(N)

Search(P):
- First phase: no string access
- Second phase: O(p/B) I/Os

Just one string is checked !!



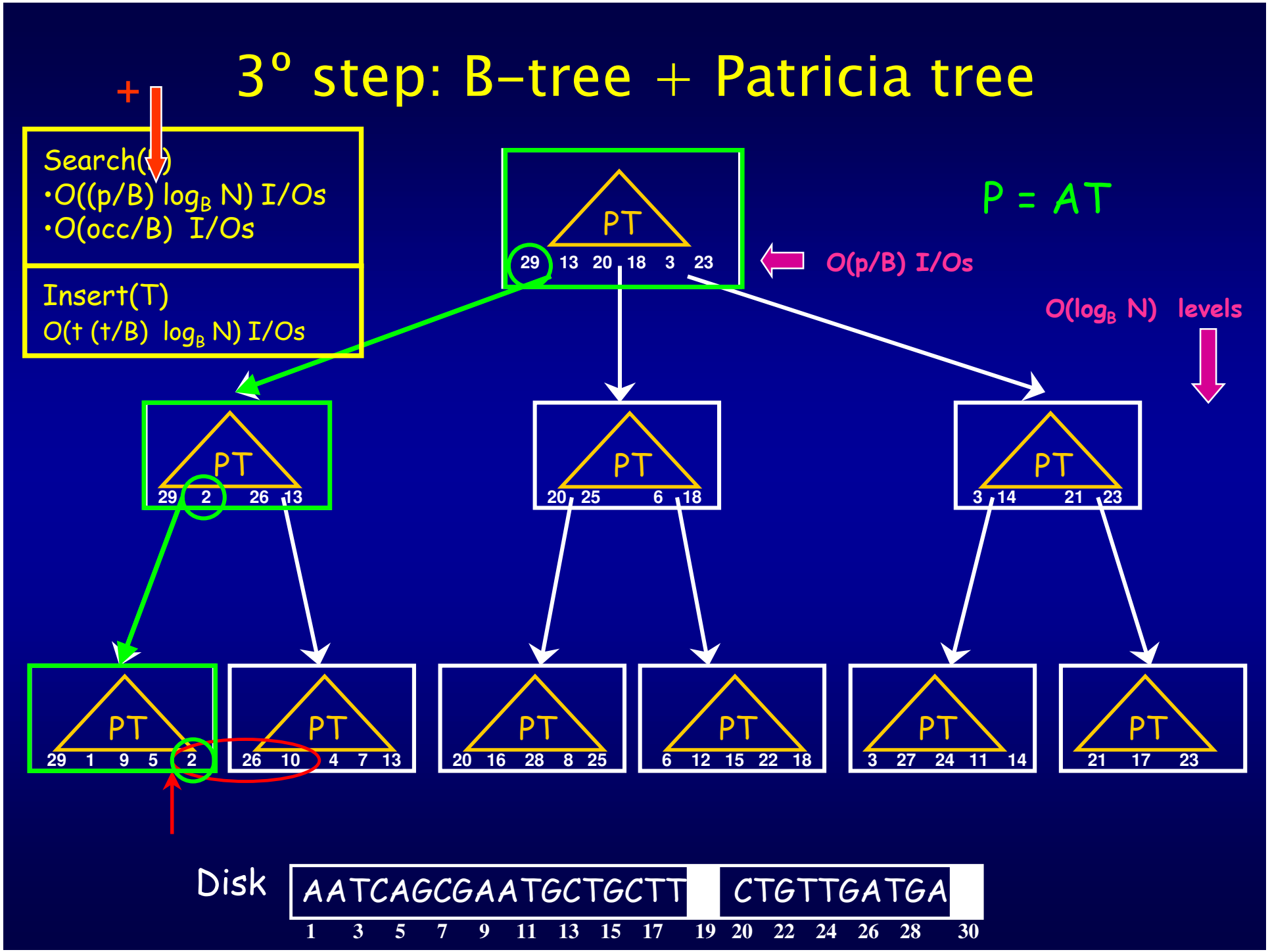
Two-phase search:
P = GCACGCAC

Disk

A	A	A	G	G	G	G
G	G	G	C	C	C	C
A	A	A	G	G	G	G
A	A	C	C	C	C	C
G	G		A	A	G	G
A	G		G	G	G	G
			A	G	A	G
						A

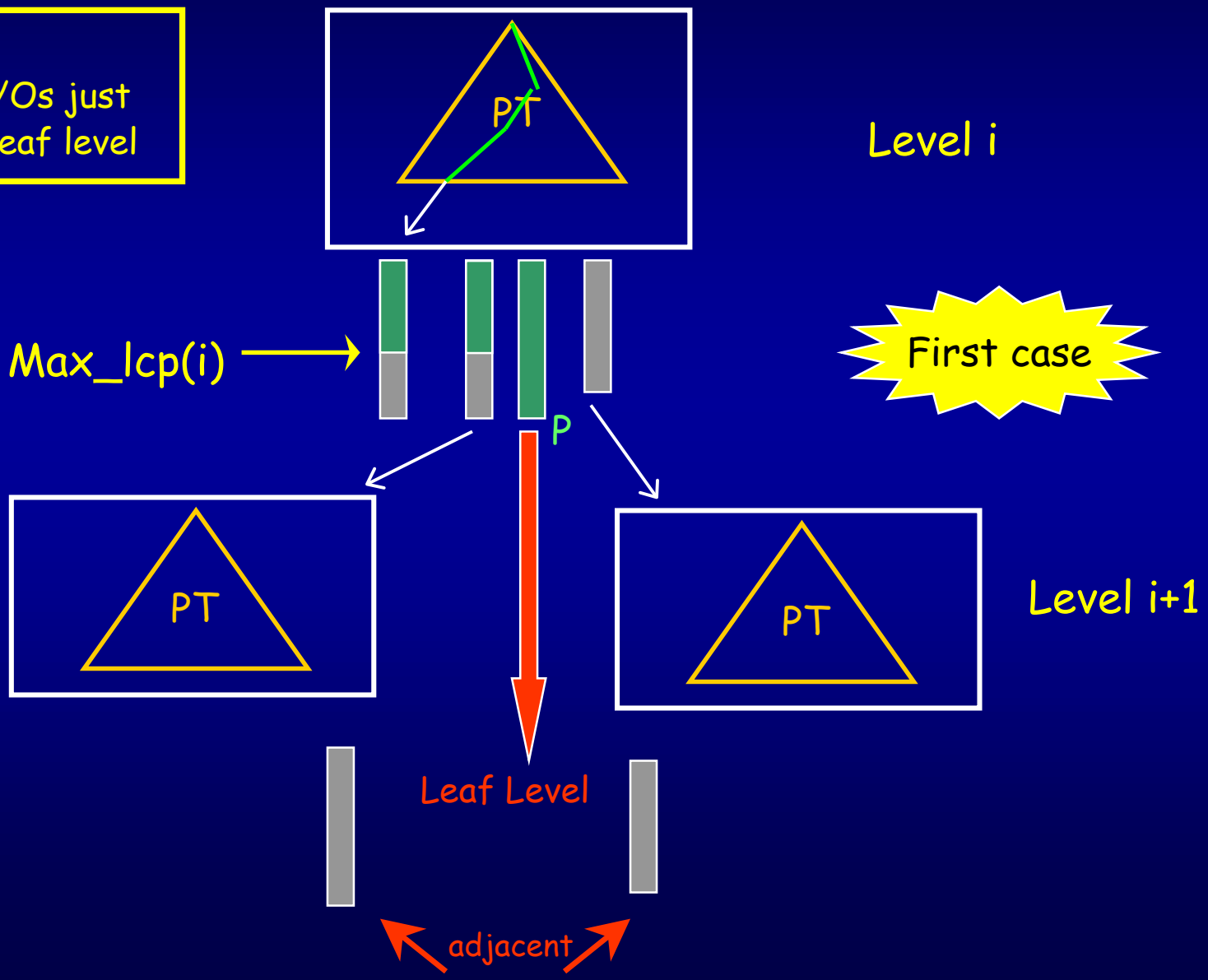
P's position

mismatch


3° step: B-tree + Patricia tree
+
Search(T)
•O((p/B) log_B N) I/Os
•O(occ/B) I/Os
Insert(T)
O(t (t/B) log_B N) I/Os
P = AT
PT
29 13 20 18 3 23
O(p/B) I/Os
O(log_B N) levels
PT
29 2 26 13
PT
20 25 6 18
PT
3 14 21 23
PT
29 1 9 5 2
PT
26 10 4 7 13
PT
20 16 28 8 25
PT
6 12 15 22 18
PT
3 27 24 11 14
PT
21 17 23
Disk
AATCAGCGAATGCTGCTT
CTGTTGATGA
1 3 5 7 9 11 13 15 17 19 20 22 24 26 28 30

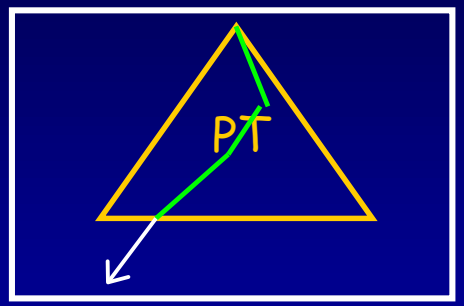
4° step: Incremental Search

Search(P)
• $O(\log_B N)$ I/Os just to go to the leaf level



4° step: Incremental Search

Search(P)
- $O(p/B + \log_B N)$ I/Os
- $O(occ/B)$ I/Os



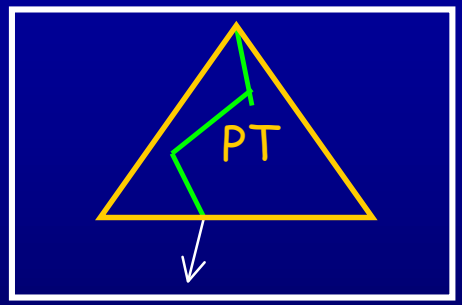
Level i

Second case

Max_lcp(i)

Inductive Step

P

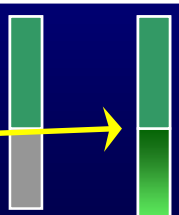


No rescanning

Level i+1

i-th step:
$O((lcp_{i+1} - lcp_i)/B + 1)$ I/Os

skip Max_lcp(i)



Max_lcp(i+1)

In summary

String B-tree performance: [Ferragina-Grossi, 95]

- Search(P) takes $O(p/B + \log_B N + occ/B)$ I/Os
- Update(T) takes $O(t \log_B N)$ I/Os
- Space is $\Theta(N/B)$ disk pages

Using the String B-tree in internal memory:

- Search(P) takes $O(p + \log_2 N + occ)$ time
- Update(T) takes $O(t \log_2 N)$ time
- Space is $\Theta(N)$ bytes
  - ➢ It is a sort of dynamic suffix array

Many other applications:

- String sorting [Arge *et al.*, 97]
- Dictionary matching [Ferragina *et al.*, 97]
- Multi-dim string queries [Jagadish *et al.*, 00]

Algorithmic Engineering

(Are String B-trees appealing in practice ?)

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

Preliminary considerations

Given a String B-tree node π, we define:

- S_π = set of all strings stored at node π
- b = maximum size of S_π

An interesting property:

- H grows as $\log_b N$, and does not depend on Δ's structure
- b is related to the space occupancy of PT_π, and $b < B$

☑ The larger is b, the faster are search and update operations

☑ Our Goal: Squeeze PT_π as much as possible

PT_π implementation

Node π actually contains (let $k=|S_\pi|$):

- PT_π = Patricia trie indexing the k strings of S_π
- The pointers to the k/2 children of π ⟸ 3 or 4 bytes
- Some auxiliary and bookeping information ⟸ negligible

If the strings are binary then PT_π constists of:

- k leaves, pointing to S_π's strings
- (k-1) internal nodes, each storing an integer value
- (2k-1) arcs, each storing one single char

Implementing PT_π takes: [Ferragina-Grossi, 96]

- 12k bytes, via a pointer-based solution
- 9k bytes via a proper encoding of the binary-tree structure

Some details and results

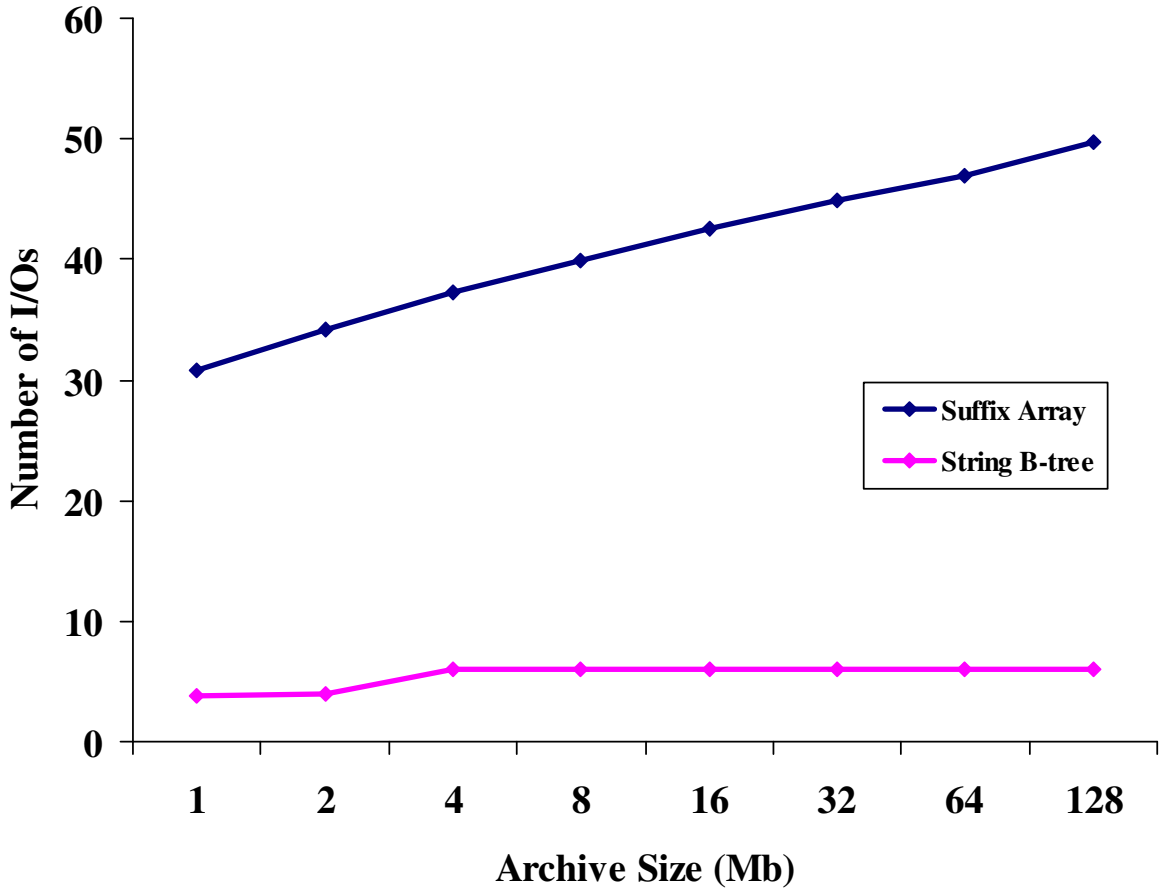
Experiments have shown that: [Ferragina-Grossi, 96]

- Search(P)
  - It takes about 2H disk accesses (as the worst-case bound)
  - It is 10 times faster than Suffix Array search
  - Comparable to Suffix Tree search
- Insert(T), via a batched insertion
  - It is 5 times faster than UNIX Prefix B-trees
  - Better page-fill ratio than Suffix trees

Two limitations:

- Space usage of 9N is too much
- The update ops are CPU-bounded

An experiment



A new proposal

Implementing the node π:

- String pointers and child pointers in 4 bytes
- Integers in the nodes of PT_π stored via Continuation Bit
  - Experiments showed that 90% are very small → 1 byte

» How do we implement PT_π ?!

- Should be space succinct and allow basic navigational ops

Some results on the succinct coding of binary trees:

- Optimal k+o(k) bits and basic navigational ops [Jacobson, 89]
- 2k+o(k) bits and more navigational ops [Munro *et al.*, 99]

Two specialties of our context:

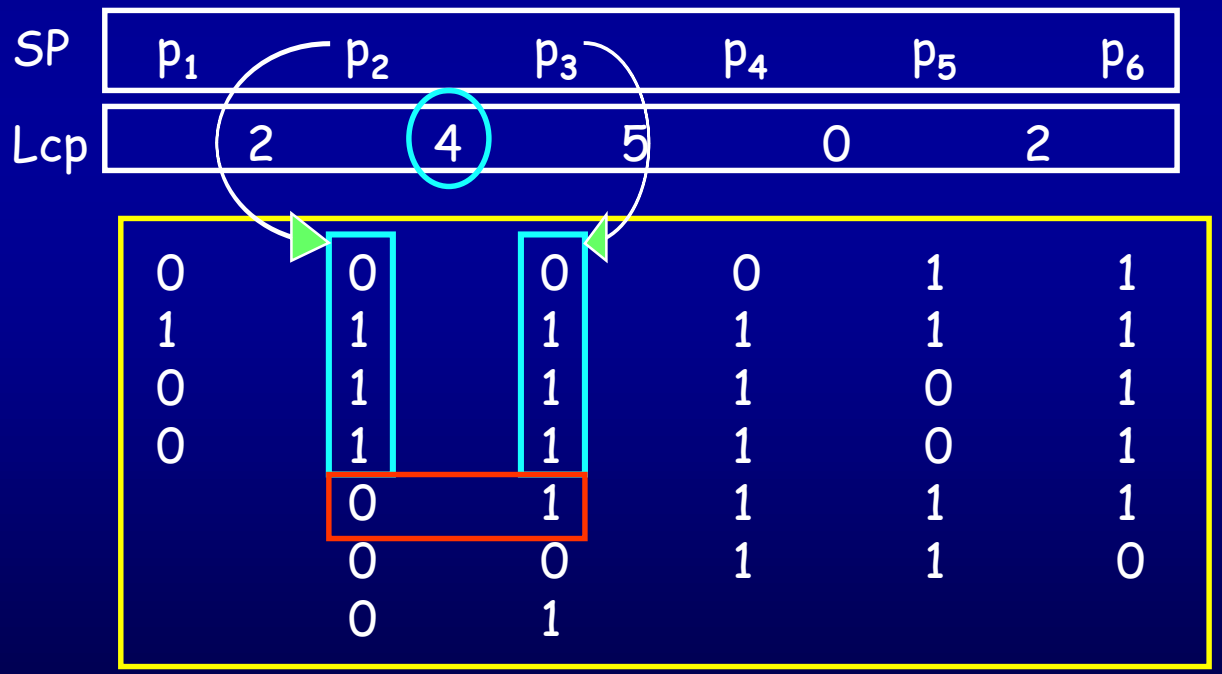
- PT_π is small, about a thousands of strings
- Navigational ops = downward traversal
- CPU-time is not the only resource, 1 I/O is surely paied

PT_π's topology may be dropped !!

[Ferguson, 92]

Take the in-order visit of PT_π :

- SP[1,k] array of pointers to S_π's strings (ie. PT_π leaves)
- Lcp[1,k-1] array of LCPs between strings adjacent in SP



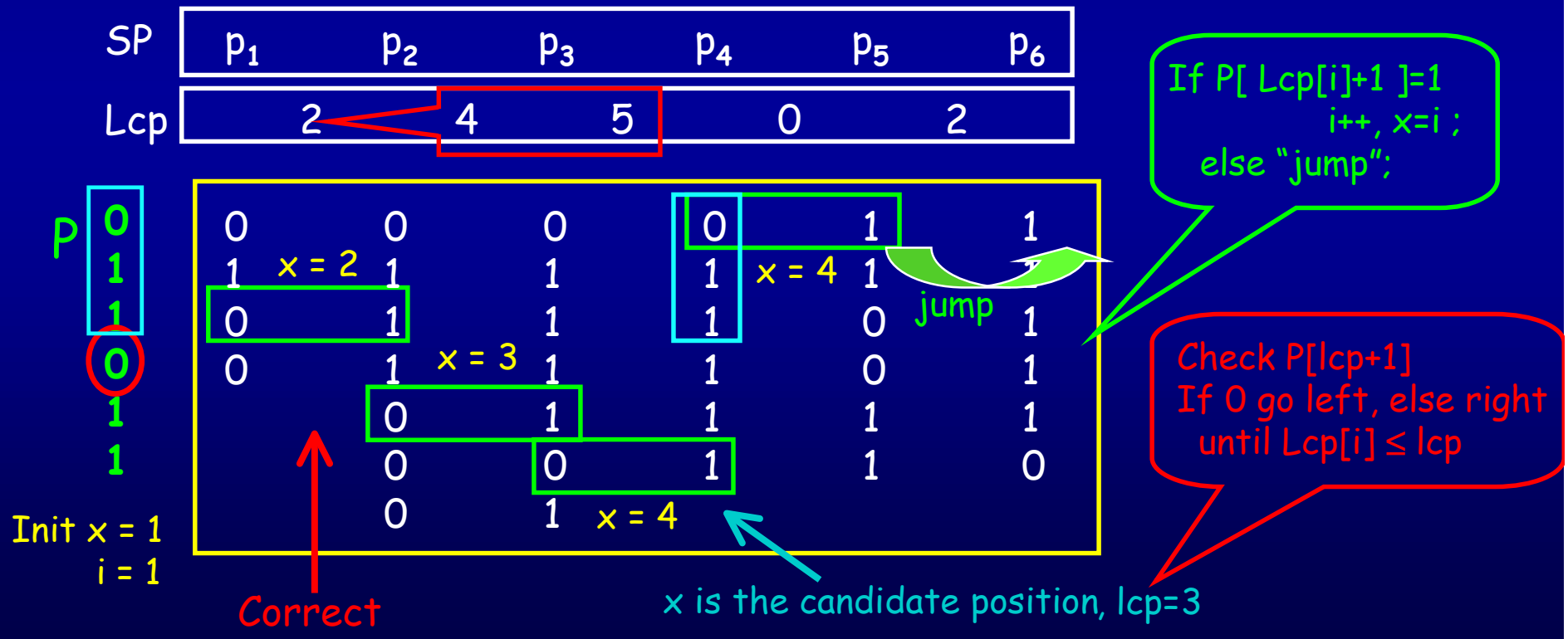
S_π's strings on Disk

PT_π's topology may be dropped !!

[Ferguson, 92]

Take the in-order visit of PT_π :

- SP[1,k] array of pointers to S_π's strings (ie. PT_π leaves)
- Lcp[1,k-1] array of LCPs between strings adjacent in SP



In summary

Node π contains (let $k=|S_\pi|$):

- A pointer array SP[1,k]
- An integer array Lcp[1,k-1], stored by Continuation Bit

Searching P's position among S_π's strings:

- 1 I/O to fetch the disk page containing node π
- 2 array scans: O(p+k) chars and integer comparisons
- 1 string access to the candidate string, O(p/B) I/Os

Since k is about a thousands of strings:

- The I/O to fetch the disk page takes 5,000 μs
- The two array scans are very fast: 200 μs (cache prefetching)
  - » The string access might deploy "incremental search"

☑ Same I/O-bounds as before, and about 5N bytes of space in practice !!

Research Issues

- Provide a public implementation of String B-trees
  - ✓ Refer to Berkeley-DB for the API
- Xpath queries: How to index a labeled tree for path queries ?
  - ✓ /doc/author/name/*paolo*
- Multi-dimensional substring queries: multi-field record search
  - ✓ May we plug Geometric data structures in String B-trees ?
- Stream of queries, possibly biased: String B-tree is not optimal
  - ✓ May we devise a self-adjusting index ? [Sleator-Tarjan, 85]
- Cache-oblivious tries: No explicit paramerization on B
  - ✓ String B-tree are balanced but B-dependant !

Index Construction

(Building a full-text index is a challenging task !)

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

Some considerations

We have already shown that the Suffix Array SA and the corresponding LCP array suffice to build the String B-tree

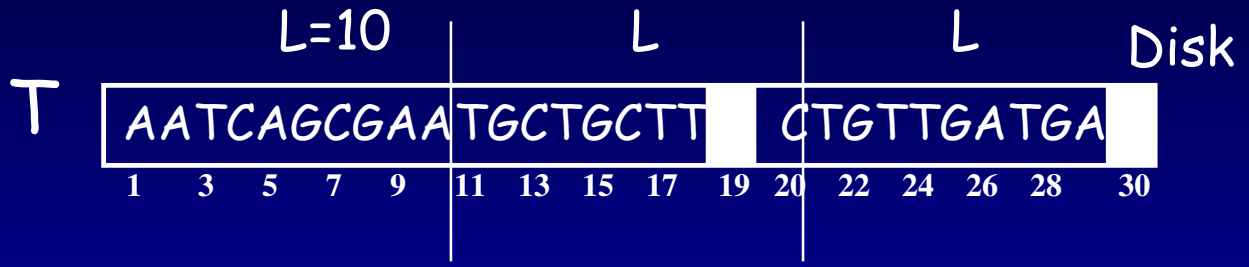
How do we build the arrays SA and Lcp ?

- In-memory algorithms are inefficient
- Naming + Ext_Sort efficient but space consuming [Crauser *et al.*, 00]
- ∃ theoretically optimal algorithm, but complicated and space costly [Ferragina *et al.*, 98]

There exists an algorithm which is [BaezaYates *et al.*, 92]

- Theoretically unacceptable: cubic I/O complexity
- Practically very appealing for performance and space occupancy
- Its asymptotics can be improved with some tricks [Crauser *et al.*, 00]

Suffix Array merge (first step)



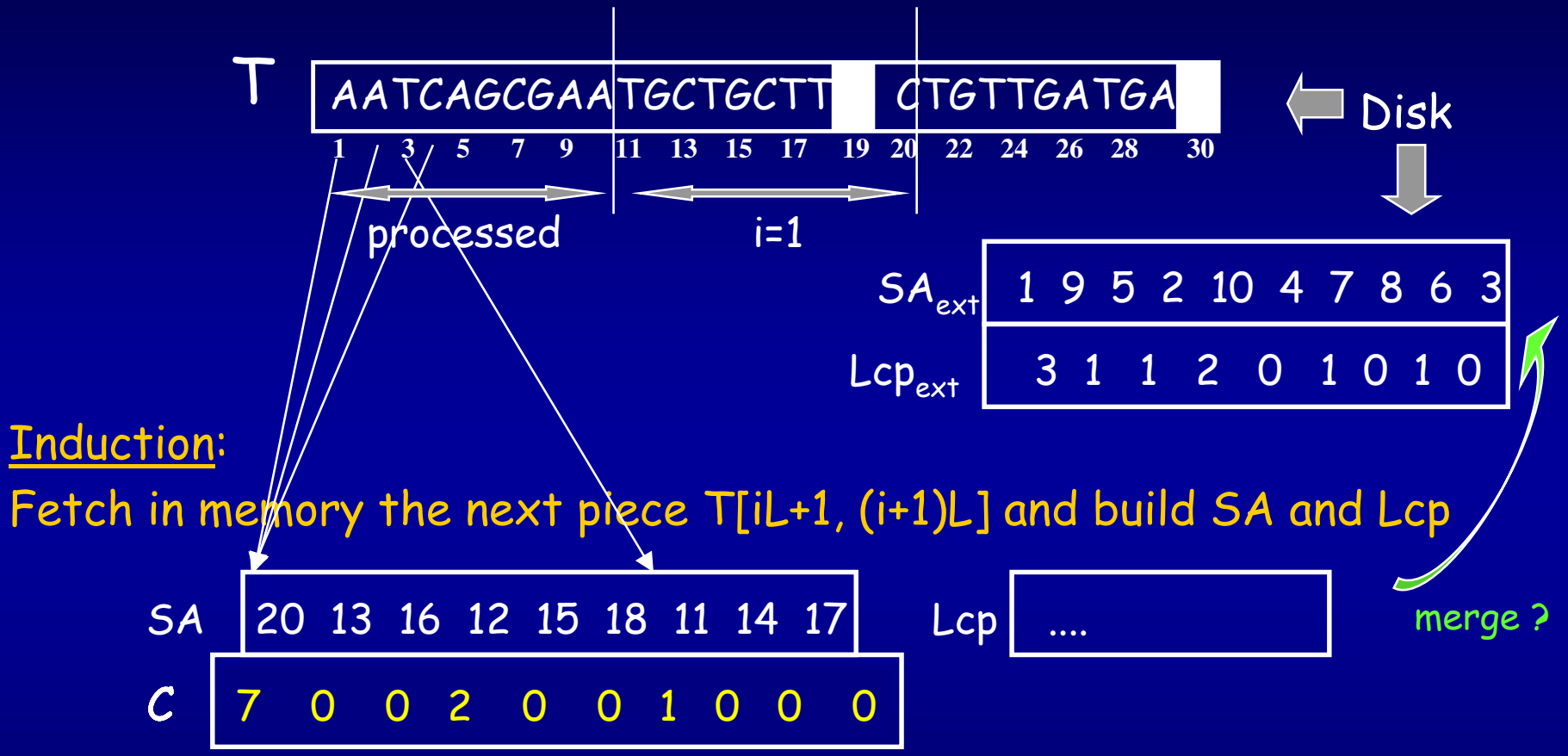
Fetch in memory the first piece T[1,L] and build SA and Lcp for the suffixes which start at positions 1..L

Possibly some extra I/Os are needed, cmp 1st and 9th suffix

Induction: We have SA_{ext} and Lcp_{ext} for the suffixes starting inside T[1,iL], we extend this to the suffixes starting in T[iL+1, (i+1)L]

We aim at executing mainly bulk I/Os

Suffix Array merge (inductive step)

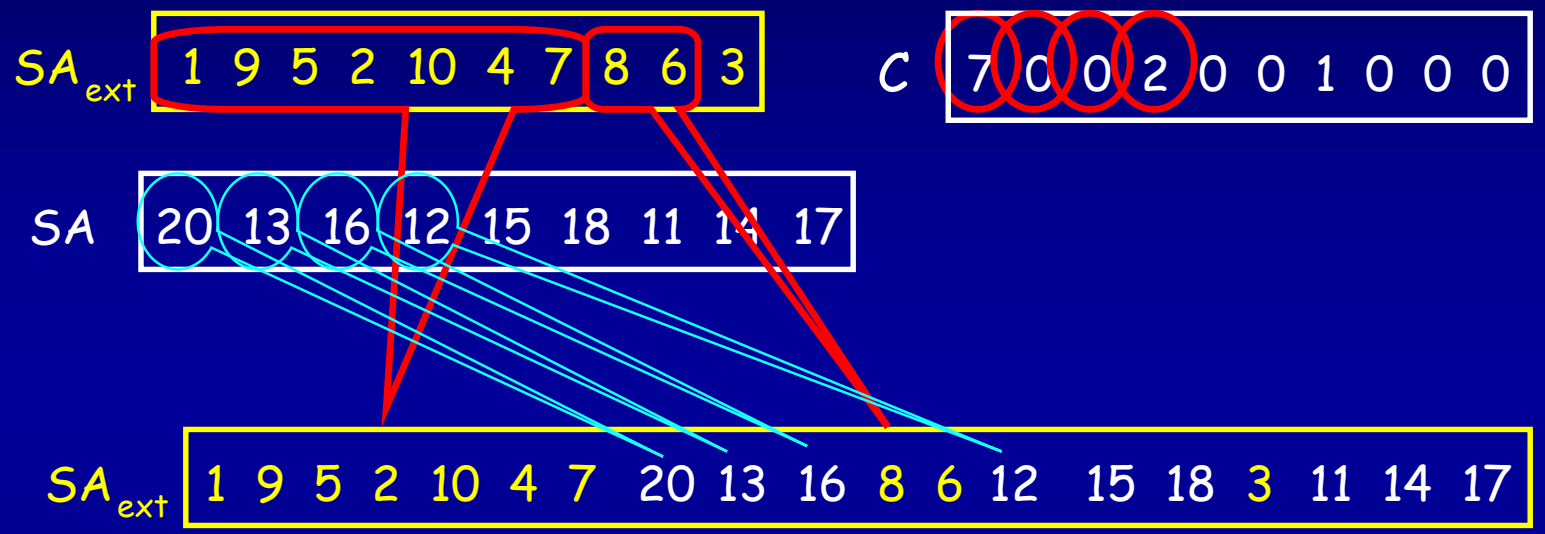


Scan T[1,iL] on disk and compute an in-memory "counting" array *C*

- Search within SA the position of each suffix starting into T[1,iL]
  - » This takes O(iL/B) I/Os [actually bulk I/Os]

Suffix Array merge (inductive step)

Merge SA_{ext} and SA by using the array C, via a disk scan



In the worst-case it is a cubic bound !!

The I/O-complexity of the i-th step is:

- Fetching T[iL+1, (i+1)L] takes $O(L/B)$ I/Os (bulk I/Os)
- Building SA and LCP takes practically no I/Os (or few randoms)
- Computing C via a scan of T[1,iL] takes $O(iL/B)$ I/Os (bulk I/Os)
- Merging SA_{ext}[1,iL] and SA[1,L] via C[1,L+1] takes $O(iL/B)$ I/Os (bulk I/Os)

➢ Overall the algorithm executes $O(N^2/M^2)$ I/Os in practice, mainly bulk I/Os.

String Sorting

(Sorting strings is similar to sorting suffixes ?)

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

On the nature of string sorting

In internal memory, we know an optimal bound:

- Via a compacted trie we get $\Theta(K \log_2 K + N)$ time
- Lower bound comes from the "sorting of K elements"

In external memory, we would expect to achieve:

$$\Theta((K/B) \log_{M/B} (K/B) + (N/B)) \quad I/Os$$

but,

- String B-trees allow to achieve $O(K \log_B K + (N/B))$ I/Os
- Three-way quicksort gets $O(K \log_2 K + N)$ I/Os [Bentley-Sedgewick, 97]

The situation is much complicated, the complexity depends on:

- "breaking" strings into chars is allowed
- the string size relative to B

The scenario

Let us define ($K = K_S + K_L$; $N = N_S + N_L$):

- K_S and N_S for strings smaller than B
- K_L and N_L for strings longer than B

If strings are indivisible everywhere (it is optimal):

$$\underbrace{(N_S/B)\log_{M/B}(N_S/B)}_{\text{short}} + \underbrace{K_L \log_{M/B} K_L + (N_L/B)}_{\text{long}}$$

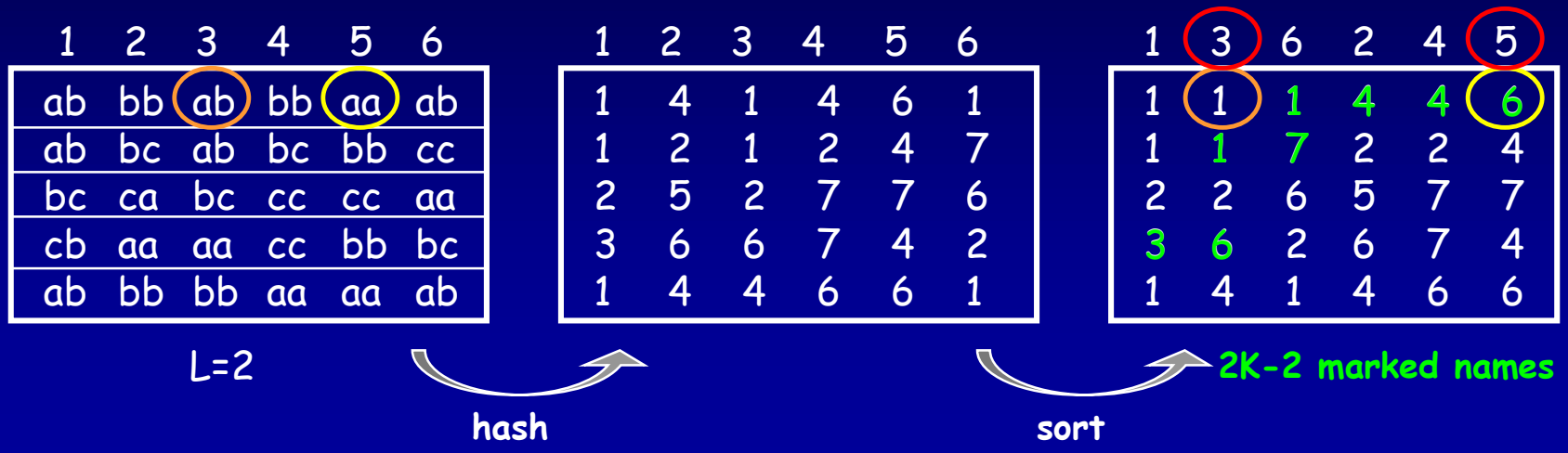
If strings are only indivisible in external memory:

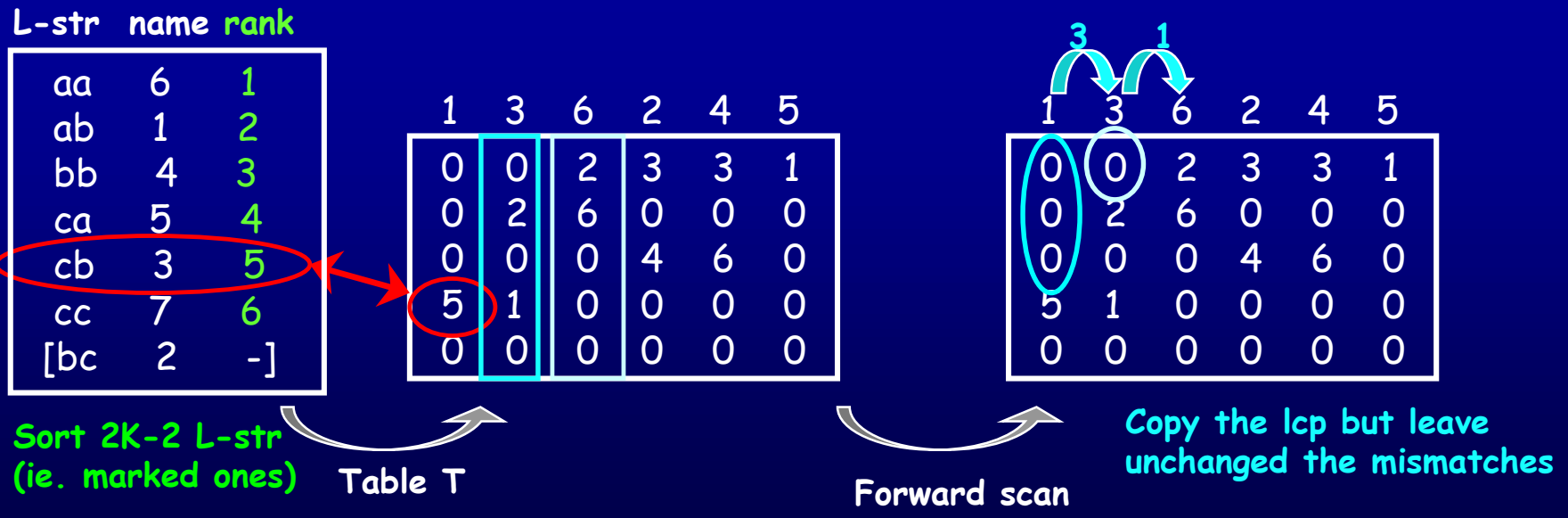
$$\underbrace{\min\{ K_S \log_M K_S , (N_S/B)\log_{M/B}(N_S/B)\}}_{\text{short}} + \underbrace{K_L \log_{M/B} K_L + (N_L/B)}_{\text{long}}$$

If strings may be chopped into pieces: $O(N/B)$ I/Os

- It is a randomized algorithm [Ferragina-Thorup, 97]
- The average string length should be $\Omega((\log_{M/B}(N/B))^2 \log_2 K)$

The randomized algorithm [Ferragina–Thorup, 97]





The randomized algorithm (contd.)

1	2	3	4	5	6
ab	bb	ab	bb	aa	ab
ab	bc	ab	bc	bb	cc
bc	ca	bc	cc	cc	aa
cb	aa	aa	cc	bb	bc
ab	bb	bb	aa	aa	ab

Input

1	3	6	2	4	5
1	1	1	4	4	6
1	1	7	2	2	4
2	2	6	5	7	7
3	6	2	6	7	4
1	4	1	4	6	6

Hashed and sorted strings

1	3	6	2	4	5
0	0	2	3	3	1
0	2	6	0	0	0
0	0	0	4	6	0
5	1	0	0	0	0
0	0	0	0	0	0

Table T after Forward Scan

See the survey

3 1

1	3	6	2	4	5
2	2	2	3	3	1
2	2	6	0	0	0
0	0	0	4	6	0
5	1	0	0	0	0
0	0	0	0	0	0

Backward scan

Copy the lcp but leave unchanged the mismatches

5	3	1	6	2	4
1	2	2	2	3	3
0	2	2	6	0	0
0	0	0	0	4	6
0	1	5	0	0	0
0	0	0	0	0	0

sort

correct

Research issues

- Close the various gaps
  - ✓ Long strings in the case of indivisibility on external memory
  - ✓ Better analysis for the randomized algorithm
- Implement all those algorithms
- What about cache-oblivious string sorting algorithms ?
  - ✓ Most of them are based on tries
  - ✓ Arbitrary length creates a lot of problems
  - ✓ Probably the randomized approach can help in this case too

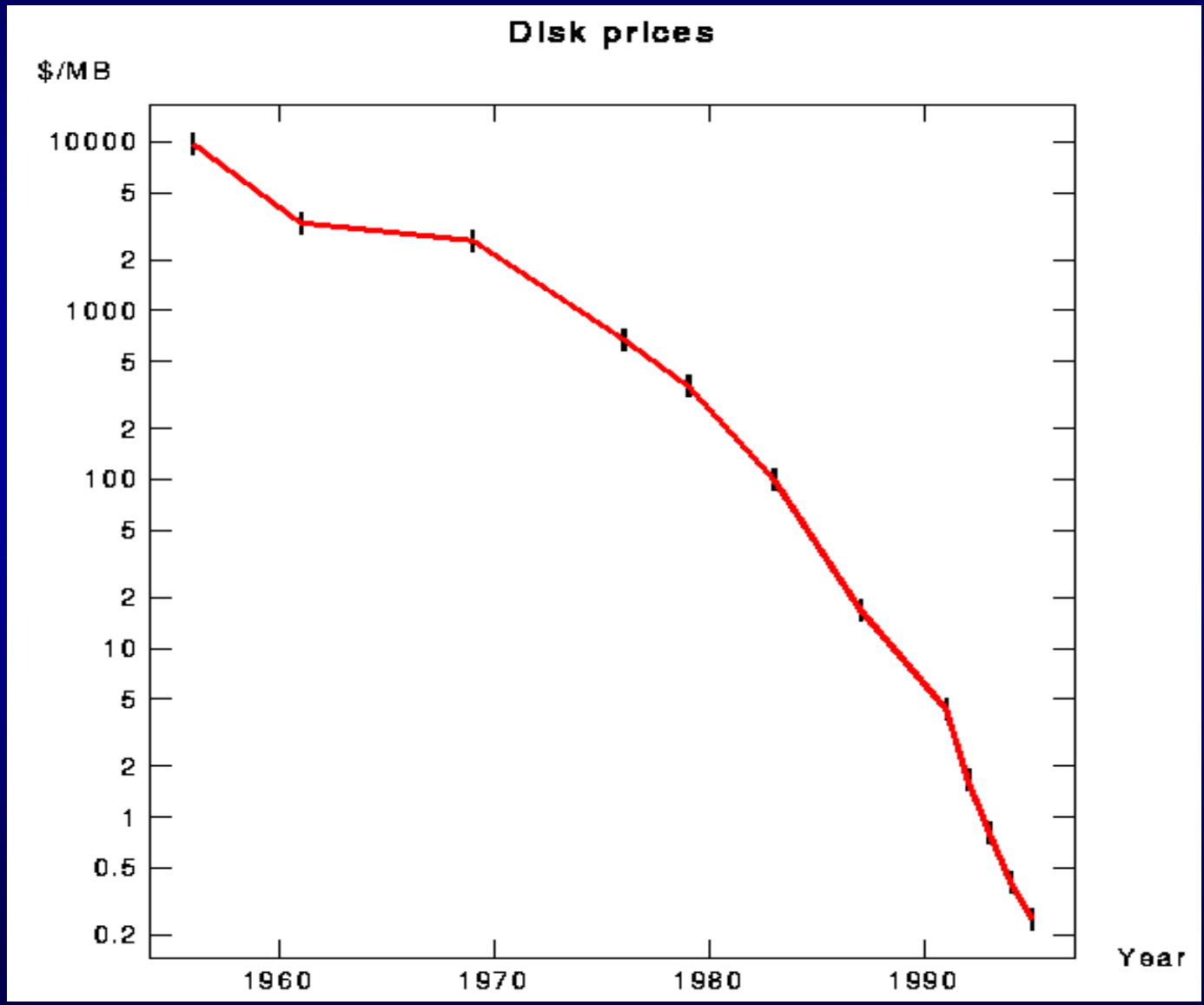
Compressed Indexes

(Is space overhead the tax to pay for using a full-text index ?)

String
algorithms and data structures
(or, tips and tricks for index design)

Paolo Ferragina

Disks are cheaper and cheaper



Why compressing data ?

☺ Compression has two positive effects:

- ✓ Space saving
- ✓ Performance improvement
  - Better use of memory levels close to processor
  - Increased disk and memory bandwidth
  - Reduced (mechanical) seek time
  - » CPU speed makes (de)compression "costless" !!

☑ Well established: "It is more economical to store data in compressed form than uncompressed"

☑ Knuth in the 3rd vol says: "Space optimization is closely related to time optimization in a disk memory system"

☑ IBM released in March 2001 the Memory eXpansion Technology (MXT) plugged into eServers x330

- ✓ Double memory at about same cost and performance

The scenario

Large constants from 5 to 25

Classical full-text indexes use $\Theta(N \log_2 N)$ bits of storage

- Suffix array: $O(p + \log_2 N + occ)$ time
- String B-tree: $O((p/B) + \log_B N + (occ/B))$ I/Os

Succinct suffix trees use $N \log_2 N + \Theta(N)$ bits of storage [Munro et al., 97....]

Really needed ?!

Suffix permutation cannot be any from {1, 2, ..., N}

- ✓ # binary texts = 2^N « N! = # permutations on {1, 2, ..., N}

Compact suffix array uses $\Theta(N)$ bits of storage [Grossi-Vitter, 00]

- ✓ Query time is $O((\log_2 N) + occ (\log_2 N)^{\varepsilon})$ time

May we achieve o(N) on compressible texts ?
As in the case of word-based indexes

The problem

- Input:
  - A constant-sized alphabet Σ
  - An arbitrarily long text T[1,N] over Σ
- Query on an arbitrary string P[1,p]:
  - Count the occurrences of P in T
  - Locate the positions of the occurrences of P in T

Aim at exploiting repetitiveness in the input to squeeze the index !!

Does it exist an "opportunistic index" ?

Example: ... ... +39.050.521232, +39.050.521304, +39.06.5421245, +39.02.342109, +39.012.256312, +39.050.2212764, ...

- count the calls from Rome (+39.06.*)
- locate who called from CS-dept in Pisa (+39.050.22127*)

The FM-index [Ferragina-Manzini, 00]

Bridging data-structure design and compression techniques:

- ✓ Suffix array data structure
- ✓ Burrows-Wheeler Transform → bzip2 compression algorithm (1994)

The theoretical result:

- ✓ Query complexity: $O(p + occ \log^{\varepsilon} N)$ time
- ✓ Space occupancy: $O(N \mathcal{H}_k(T)) + o(N)$ bits → o(N) if T is compressible

k-th order empirical entropy, it may be o(1)

The nice stuff is that this result:

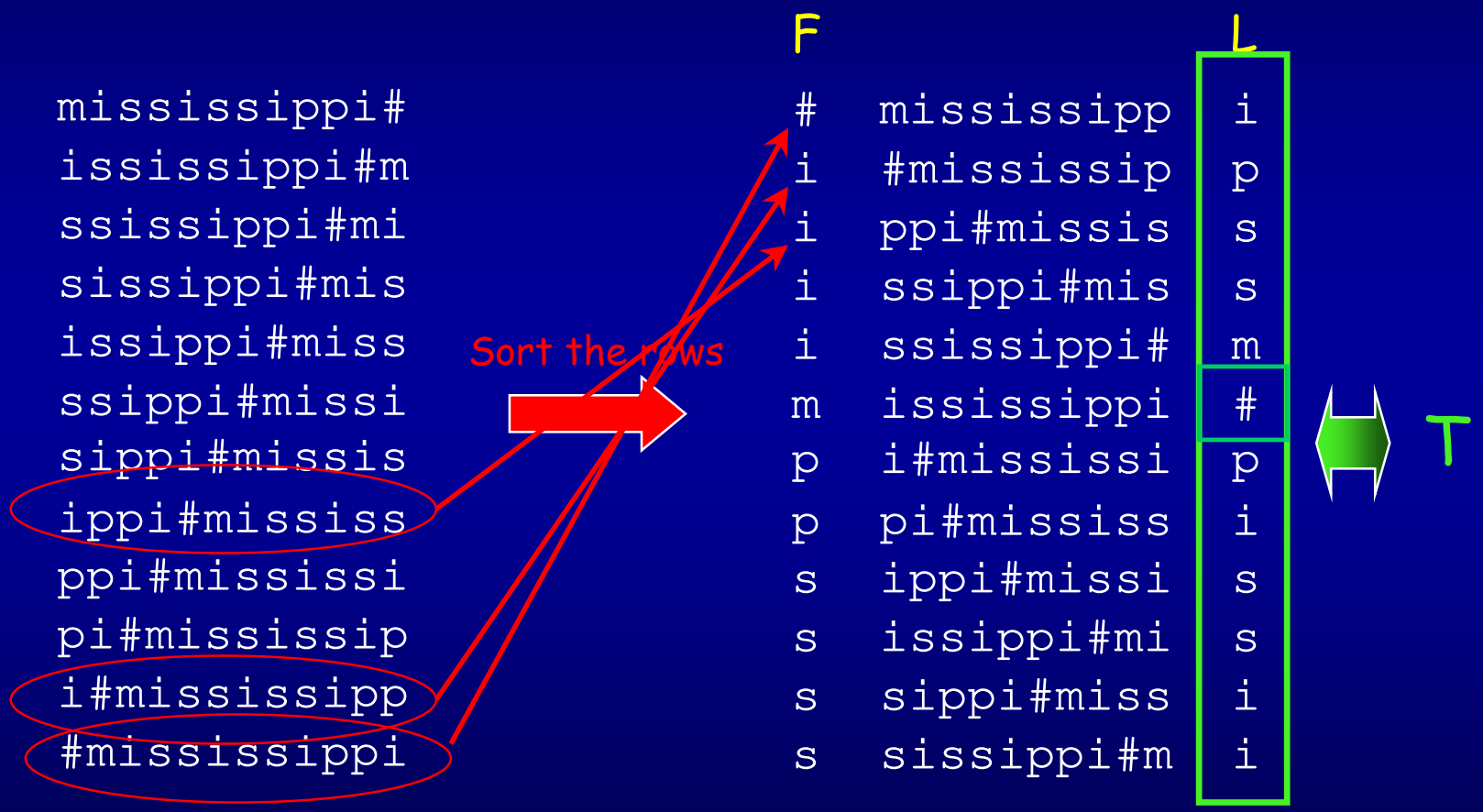
- ✓ is independent on the input source, ie. pointwise on T
- ✓ implicitely shows that Suffix Arrays are "compressible"

In practice, the FM-index is much appealing:

- ✓ Space close to the best known compressors
- ✓ Query time of few millisecs on hundreds of MBs of text

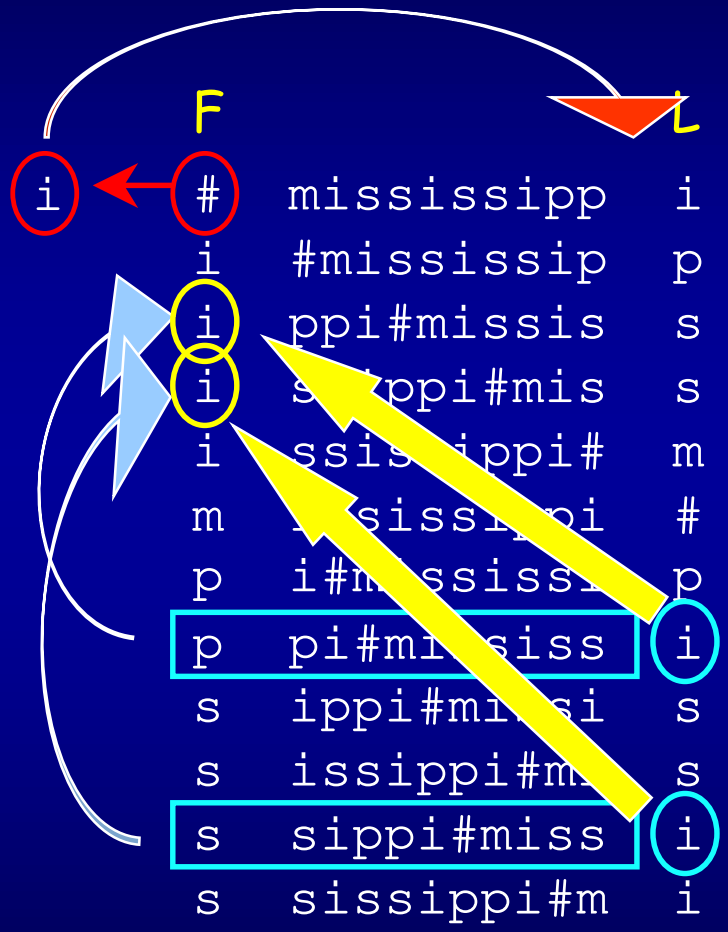
The BW-Transform

Let us given a text T = mississippi#



Every column is a permutation of T, hence also F and L

BWT is invertible



1. L's chars precede F's in T

→ Reconstruct backward T

→ We stop at " i# " soon !!

2. How do we map L's onto F's chars ?

... Need to distinguish equal chars in F...

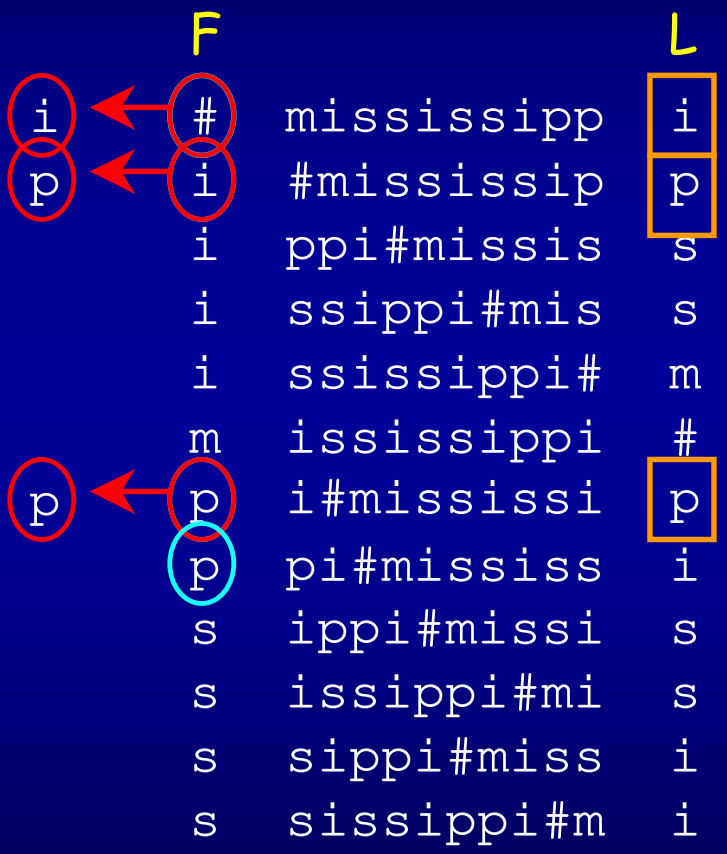
→ Take two equal L's chars

→ Rotate their rows

→ Same relative order !!

3. Hence, i-th "c" in L is the i-th "c" in F

BWT is invertible (contd.)



Two properties:

- L's chars precede F's in T
- i-th "c" in L = i-th "c" in F

Reconstruct T
... in O(N) time

T= ... p p i #

L is highly compressible

F		L
#	mississipp	i
i	#mississip	p
i	ppi#missis	s
i	ssippi#mis	s
i	ssissippi#	m
m	ississippi	#
p	i#mississi	p
p	pi#mississ	i
s	ippi#missi	s
s	issippi#mi	s
s	sippi#miss	i
s	sissippi#m	i

Two observations:

- Equal substr prefix adjacent rows
- Close chars are "similar"

Locality !!

Algorithm Bzip :

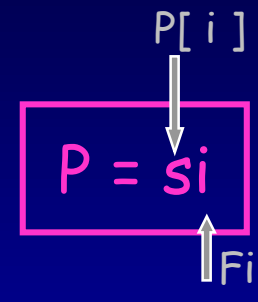
1. Move-to-Front coding of L → L'
2. Run-Length coding of L' → L''
3. Statistical coder on L'': Arithmetic

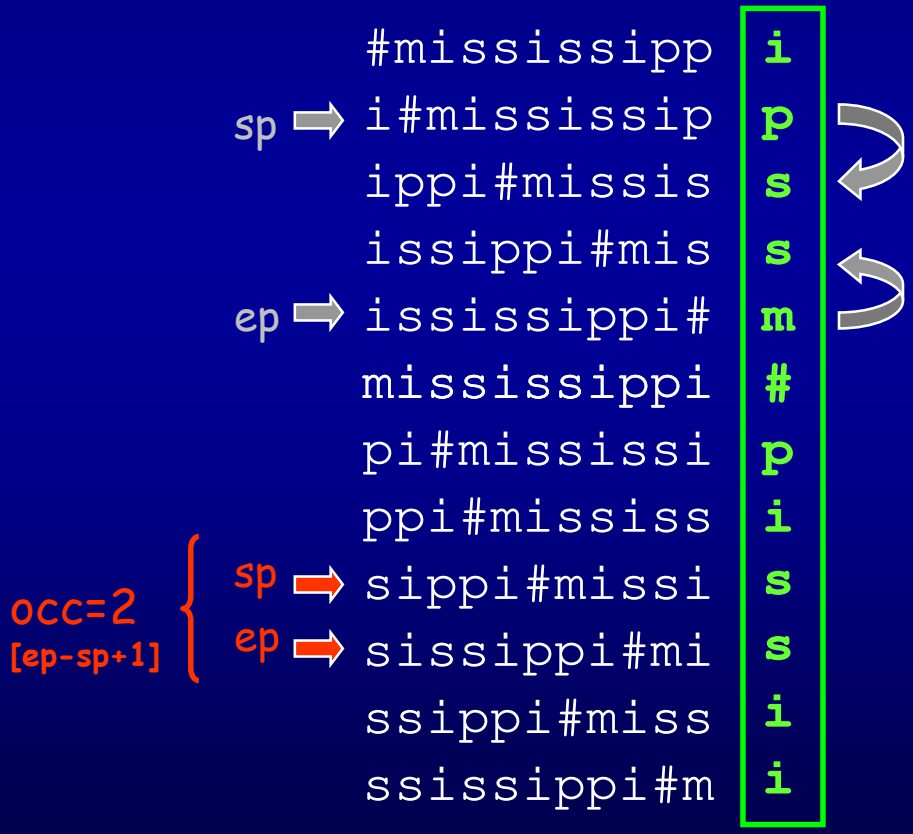
☑ Bzip compresses much better than Gzip, but it slower in (de)compression !!

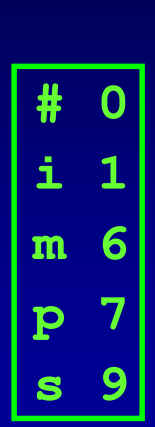
Suffix Array vs. BW-transform

SA	Rotated text	L
12	#mississipp	i
11	**i**#mississip	p
8	**ippi**#missis	s
5	**issippi**#mis	s
2	**ississippi**#	m
1	**mississippi**	#
10	**pi**#mississi	p
9	**ppi**#mississ	i
7	**sippi**#missi	s
4	**sissippi**#mi	s
6	**ssippi**#miss	i
3	**ssissippi**#m	i

Full-text search in L





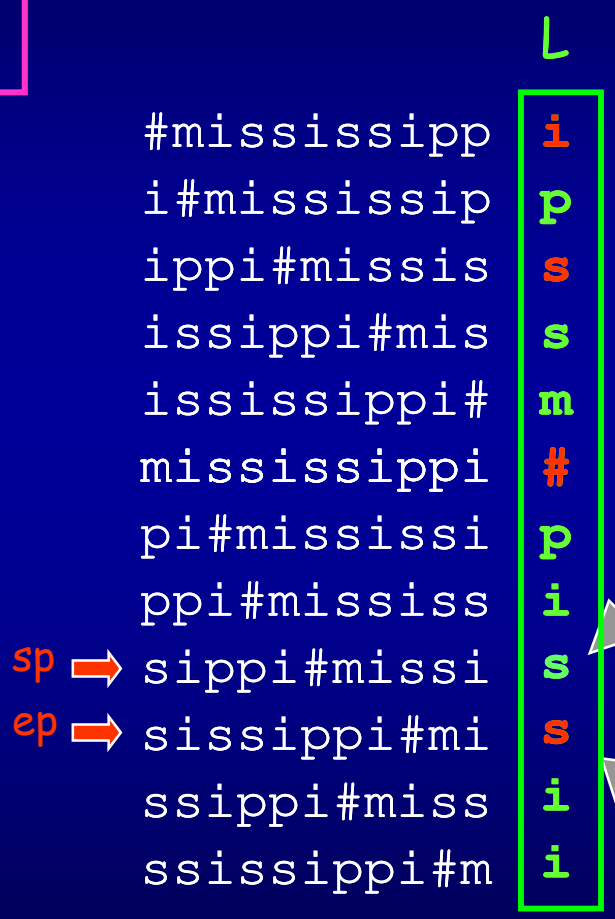


Inductive step: Given sp,ep for P[i+1,p]

1. Take c=P[i]
2. Find first c in L[sp,...]
3. Find last c in L[..., sp]
4. L-to-F mapping of these chars

Locate the occurrences

P = si



T = mississippi#

1 4 8 12

sampling step is 4

In theory, we set to $(\log N)^{\varepsilon}$ to balance space and listing time

For this, we need to go backward:

missississippi#

3 ep's row

From s's position we get 4 + 3 = 7, ok !!

This occurrence is listed immediately !

The FM-index in practice

We developed two tools:

- ✓ Tiny index supports just the counting of the occurrences
- ✓ Fat index supports both count and locate

both of them encapsulate a compressed copy of the text

Collection AP-news 64Mb	Space	Query Time
Tiny index	22 %	2 ms (counting only)
Fat index	35 %	5 ms (locating one occurrence)
Grep on **Gzip**ped files **(Zgrep)**	37 %	6,000 ms

☑ *Lossless fingerprint* : Existential and counting queries fast

Word-based compressed index

What about word-based occurrence of P ?

- ✓ Search for P as a substring of T, using the FM-index
- ✓ For every candidate occurrence, check if it a word-based one

word prefix substring suffix

T = ...␣bzip␣...␣bzip2␣unbzip2␣unbzip␣ ...

P=bzip

...the post-processing phase can be very costly.

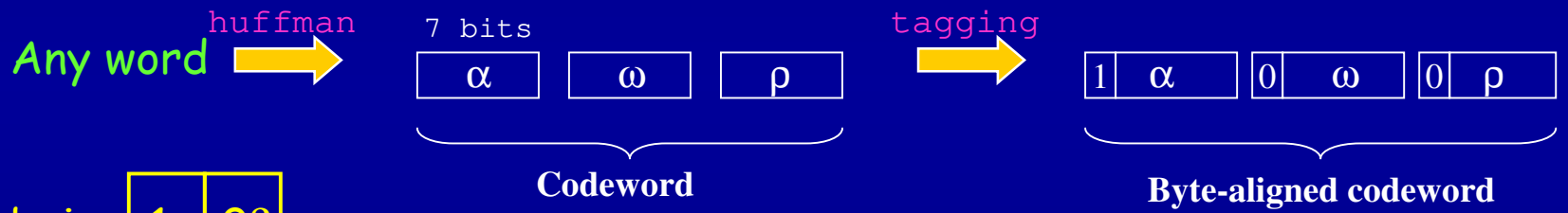
The FM-index can be adapted to be word-based

- ✓ Preprocess T to form a "digested" text DT
- ✓ Build an FM-index over DT
- ✓ Transform any word-based occurrence on T, into a substring occurrence on DT, and solve it using the FM-index built on DT

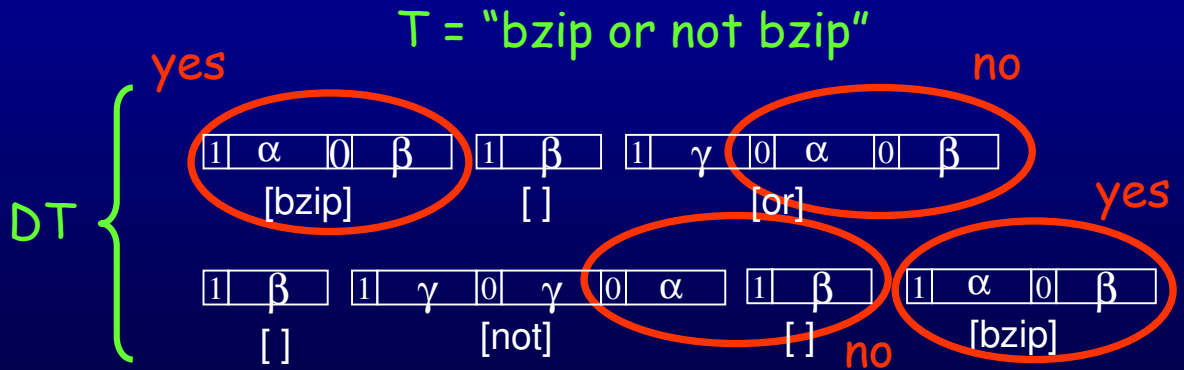
The WFM-index

Variant of Huffman algorithm:

- ✓ Symbols of the huffman tree are the *words* of T
- ✓ The Huffman tree has fan-out *128*
- ✓ Codewords are *byte-aligned* and *tagged*



P= bzip= 1α 0β



WFM-index

1. Dictionary of words
2. Huffman tree
3. FM-index built on DT

space ~ 22 %
word search ~ 4 ms

Research issues

- Achieve O(occ) time in occurrence retrieval
  - ✓ $O(N \mathcal{H}_k(T) \ (\log N)^{\varepsilon}) + o(N)$ bits [Ferragina-Manzini, 01]
- Achieve O(occ/B) I/Os in occurrence retrieval
  - ✓ Known compressed indexes perform random accesses
- Fast constuction algorithms for Suffix Arrays
  - ✓ Bzip compression or FM-index construction
  - ✓ Suffix Tree construction
  - ✓ Clustering of documents []
- Implement the IR-tool: WFM-index + Glimpse
  - ✓ This improves theoretically the Inverted Lists

The end

"By few years, we will be able to store everything"

[Gray, 99]

Plato (in *Phaedrus*) suggested that writing would crate

"forgetfulness in the minds of those who learn to use it"
and "the show of wisdom without the reality".

I hope that this will not occur, again !!