Prologo

Paolo Ferragina
Dipartimento di Informatica, Università di Pisa

Pre-history of string processing!
- Collection of strings
  - Documents
  - Books
  - Emails
  - Source code
  - DNA sequences
  - ...

An XML excerpt
```xml
<book>
  <author>Donald E. Knuth</author>
  <title>The Textbook</title>
  <publisher>Addison-Wesley</publisher>
  <year>1986</year>
</book>

<article>
  <author>Donald E. Knuth</author>
  <author>Ronald W. Moore</author>
  <title>An Analysis of Alpha-Beta Pruning</title>
  <journal>Artificial Intelligence</journal>
  <year>1975</year>
</article>
```

The Query-Log graph
- QueryLog (Yahoo! dataset, 2005)
  - #links: 70 Mil
  - #nodes: 50 Mil
  - Dictionary of URLs: 24 Mil, 56.3 avg/chars, 1.6Gb
  - Dictionary of terms: 44 Mil, 7.3 avg/chars, 307Mb
  - Dictionary of Inlinks: 2.6Gb
In all cases...

- Some structure: relation among items
  - Trees, (hyper-)graphs, ...
- Some data: (meta-)information about the items
  - Labels on nodes and/or edges
- Various operations to be supported
  - Given node $u$
    - Retrieve its label, $Fw(u)$, $Bw(u)$, ...
  - Given an edge $(u, i)$
    - Check its existence, Retrieve its label, ...
  - Given a string $p$:
    - search for all nodes/edges whose label includes $p$
    - search for adjacent nodes whose label equals $p$

Large space
(I/O, cache, compression,...)

Id \Rightarrow String

Index

Conclusions

- Systems should **automatically compress** data whenever the benefits of storing or transmitting the compressed data outweigh the costs
  - It's time to "teach" systems how to do this
Conclusions

- Systems should **automatically compress** data whenever the **benefits** of storing or transmitting the compressed data outweigh the **costs**

- It’s time to “teach” systems how to do this

**Data Compression**

- **Random Access**
- **Search**

**Seven years ago...**

*Now, J. ACM 05*

**FOCS 2000**

The 41st Annual Symposium on Foundations of Computer Science

Opportunistic Data Structures with Applications

P. Ferragina, G. Manzini

Nowadays several papers: theory & experiments (see Navarro-Makinen’s survey)
Our starting point was...

Ken Church (AT&T, 1995) said "If I compress the Suffix Array with Gzip I do not save anything. But the underlying text is compressible... What's going on?"

Practitioners use many "squeezing heuristics" that compress data and still support fast access to them.

Can we "automate" and "guarantee" the process?

---

In these lectures....

A path consisting of five steps
1) The problem
2) What practitioners do and why they did not use "theory"
3) What theoreticians then did
4) Experiments
5) The moral ;-)!

At the end, hopefully, you'll bring at home:
- Algorithmic tools to compress & index data
- Data aware measures to evaluate them
- Algorithmic reductions: Theorists and practitioners love them!
- **No ultimate receipts!!**

---

A basic problem

Given a dictionary D of strings, of variable length, compress them in a way that we can efficiently support \( id \rightarrow \text{string} \)

- Hash Table
  - Need D to avoid false-positive and for \( id \rightarrow \text{string} \)
- (Minimal) ordered perfect hashing
  - Need D for \( id \rightarrow \text{string} \), or check
- (Compacted) Trie
  - Need D for edge match

Yet the dictionary D needs to be stored
- its space is not negligible
- I/O- or cache-misses in retrieval
Front-coding

Practitioners use the following approach:
- Sort the dictionary strings
- Strip-off the shared prefixes [e.g. host reversal?]
- Introduce some bucketing, to ensure fast random access

Locality-preserving FC

A simple incremental encoding algorithm [where $c = 2/(c-2)$]
- Assume to have FC($S_1, \ldots, S_n$)
- Given $S_a$, we proceed backward for $X = c \cdot |S_a|$ chars in FC
  - If $S_a$ is decoded, then we add FC($S_a$) else we add $S_a$

Locality-preserving FC

Drop bucketing + optimal string decompression
- Compress D up to $(1+c)$ FC$(D)$ bits
- Decompress any string $S$ in $1+|S|/c$ time

A simple incremental encoding algorithm [where $c = 2/(c-2)$]
I. Assume to have FC($S_1, \ldots, S_n$)
II. Given $S_a$, we proceed backward for $X = c \cdot |S_a|$ chars in FC
- Two cases
  - $X = c \cdot |S_a|$
  - $S_a$ FC-coded
    - $c$ copied
  - $S_a$ copied

Random access to LPFC

We call $C$ the LPFC-string, $n = \#\text{strings in } C$, $m = \text{total length of } C$

How do we Random Access the compressed $C$?
- Get($i$): return the position of the $i$-th string in $C$ (id->string)
- Previous($i$), Next($i$): return the position of the string preceding or following character $C[i]$

Classical answers: ;)
- Pointers to positions of copied-strings in $C$
  - Space is $O(n \log m)$ bits
  - Access time is $O(1) + O(|S_i|)$
- Some form of bucketing... Trade-off
  - Space is $O(nr/\log m)$ bits
  - Access time is $O(|S_i|)$

No trade-off!
Re-phrasing our problem

C is the LSPC-string, n = #strings in C, m = total length of C
Support the following operations on C:
- Get(i): return the position of the i-th string in C
- Previous(i), Next(i): return the position of the string prec/following C[i]

Proper encoding:

C = 0 http://beckmate.com/Alt_Natural/ 33 Applied.html 34 roman.html 38 html 38 sc_Art.html ...
B = 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...

- Rank_i(x) = number of 1s in B[i..x]
- Select_i(y) = position of the y-th 1 in B

- Get(i) = Select_i(i)
- Previous(i) = Select_i(Rank_i(i) - 1)
- Next(i) = Select_i(Rank_i(i) + 1)

Look at them as pointerless data structures

A basic problem!

Select(3) = 8

B = 0010100101010111110000010101010111000...

- Rank_i(7) = 4
- Select_i(4) = 51

A basic problem!

Select(3) = 8

B = 0010100101010111110000010101010111000...

- Rank_i(7) = 4
- Select_i(4) = 51

Given an integer set, we set B as its characteristic vector
- pred(x) = Select_i(Rank_i(x - 1))

LBs can be inherited [Paterson-Thomas, 1984]

A basic problem!

Select(3) = 8

B = 0010100101010111110000010101010111000...

- Rank_i(7) = 4
- Select_i(4) = 51

\[ m = \lceil \log n \rceil \]
\[ n = \#1s \]

The Bit-Vector Index

Goal. B is read-only, and the additional index takes o(m) bits.

\[
\begin{aligned}
\text{Rank} & \quad \text{Z} \\
\text{(absolute) Rank}_i & \quad \text{z} \quad \text{b} \\
\text{(bucket-relative) Rank}_i & \quad \text{z} \quad \text{b} \\
\text{Black pos} & \quad \#1 \\
\end{aligned}
\]

- Setting Z = poly(log m) and z = (1/2) log m:
  - Space is \(|B| + (m/Z) \log m + (m/z) \log Z + o(m)
  - \(|B| + O(m \log m / \log m) \) bits
  - Rank time is O(1)
  - The term \( o(m) \) is crucial in practice

\[ \Omega \]
The Bit-Vector Index

- Sparse case: If $r > k^2$, store explicitly the position of the k 1s
- Dense case: $k \leq r \leq k^2$, recurse... One level is enough!!
- still need a table of size $o(m)$.

Setting $k = \text{polylog } m$
- Space is $m + o(m)$, and B is not touched!
- Select time is $O(1)$

There exists a Bit-Vector Index taking $|B| + o(|B|)$ bits and constant time for rank/select. B is read-only!

Compressed String Storage

Paolo Ferragina
Dipartimento di Informatica, Università di Pisa

The empirical entropy $H_0$

$$H_0(S) = -\sum_i \left(\frac{m_i}{m}\right) \log_2 \left(\frac{m_i}{m}\right)$$

- $m_{H_0}$ is the best you can hope for a memoryless compressor
- We know that Huffman or Arithmetic come close to this bound

$H_3$ cannot distinguish between $A^B$ and a random with $x A$ and $y B$

We get a better compression using a codeword that depends on the k symbols preceding the one to be compressed
The empirical entropy $H_k$

$H_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma} |S(\omega)| \cdot H_k(S(\omega))$

- $S(\omega)$ = string of symbols that follow the substring $\omega$ in $S$

Example: Given $S =$ "mississippi", we have $S(\text{"y"}) = \text{s}$

Follow $\approx$ Precede

How much is "operational"? 

Entropy-bounded string storage

Goal. Given a string $S[1,m]$ drawn from an alphabet $\Sigma$ of size $\sigma$
- encode $S$ within $mH_2(S) + \alpha(m \log \sigma)$ bits, with $k \leq \ldots$
- extract any substring of $L$ symbols in optimal $\Theta(L / \log m)$ time

This encoding fully-replaces $S$ in the RAM model!

Two corollaries
- Compressed Rank/Select data structures
  - B was read-only in the simplest R/S scheme
  - We get $|B|H_2(B) + \omega(|B|)$ bits and R/S in $O(1)$ time
- Compressed Front-Coding + random access
  - Promising: FC+Gzip saves 16% over gzip on uk-2002

Bounding $|V|$ in terms of $H_k(S)$

- Introduce the statistical encoder $E_\alpha(S)$:
  - Compute $F(i) = \text{freq of } S[i]$ within its $k$-th order context $S[i-k+1, i+1]$
  - Encode every block $B[1,b]$ of $S$ as follows
    1) Write $B[1,k]$ explicitly
    2) Encode $B[k+1, b]$ by Arithmetic using the $k$-th order frequencies
  - Some algebra $\Rightarrow (m/b) \cdot (k \log \sigma) + mH_2(S) + 2 (m/b)$ bits

- $E_\alpha(S)$ is worse than our encoding $V$
  - $E_\alpha$ assigns unique $cw$ to blocks
  - These $cw$ are a subset of $\{0,1\}^*$
  - Our $cw$ are the shortest of $\{0,1\}^*$

Golden rule of data compression
$|V| \leq |E_\alpha(S)| \leq |S|H_2(S) + \alpha(|S| \log \sigma)$ bits
Part #2: Take-home Msg

- Given a binary string $B$, we can
  - Store $B$ in $|B|$ $H_2(B) + \alpha(|B|)$ bits
  - Support Rank & Select in constant time
  - Access any substring of $B$ in optimal time

- Given a string $S$ on $\Sigma$, we can
  - Store $S$ in $|S|H_2(S) + \alpha(|S| \log |\Sigma|)$ bits, where $k \leq \alpha \log |\Sigma| |S|$
  - Access any substring of $S$ in optimal time

Exp: $10^7$ select/sec, $10^9$ rank/sec

(Compressed)
String Indexing

Paolo Ferragina
Dipartimento di Informatica, Università di Pisa

What do we mean by “Indexing”?

- Word-based indexes, here a notion of "word" must be devised!
  - Inverted files, Signature files, Bitmaps

- Full-text indexes, no constraint on text and queries!
  - Suffix Array, Suffix Tree, String B-tree...

The Problem

Given a text $T$, we wish to devise a (compressed) representation for $T$ that efficiently supports the following operations:
- $\text{Count}(P)$: How many times string $P$ occurs in $T$ as a substring?
- $\text{Locate}(P)$: List the positions of the occurrences of $P$ in $T$?
- $\text{Visualized}(i)$: Print $T(i)$

- Time-efficient solutions, but not compressed
  - Suffix Arrays, Suffix Trees, ...
  - Many others...

- Space-efficient solutions, but not time efficient
  - ZGrep: uncompress and then grep it
  - CGrep, NGrep: pattern-matching over compressed text

Paolo Ferragina, Università di Pisa
The Suffix Array

Prop 1. All suffixes of \( T \) having prefix \( P \) are contiguous.
Prop 2. Starting position is the lexicographic one of \( P \).

\[ \text{(SA) space} \]

\[ \text{Suffix pointer} \]

\[ \text{Suffix Array} \]
- \( \text{SA}: (N \log_2 N) \) bits
- \( \text{Text} \ T: N \) chars
- \( \text{In practice, a total of 5N bytes} \)

Searching a pattern

Indirect binary search on \( SA \): \( O(p) \) time per suffix cmp

List of the occurrences

Suffix Array search
- \( O(\log_2 N) \) binary-search steps
- Each step takes \( O(p) \) char cmp
- Overall, \( O(\log_2 N \cdot p) \) time

where \( N < \sum < S \)
**Text mining**

$Lcp[i,N-1]$ stores the LCP length between suffixes adjacent in SA

---

**What about space occupancy?**

$T = \text{mississippi}\#$

\[ SA + T \text{ take } \Theta(N \log_2 N) \text{ bits} \]

---

**The Burrows-Wheeler Transform (1994)**

Take the text $T = \text{mississippi}\#$

<table>
<thead>
<tr>
<th>Text</th>
<th>$L$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mississippi#</td>
<td>i</td>
</tr>
<tr>
<td>isissippi#</td>
<td>p</td>
</tr>
<tr>
<td>sissippi#</td>
<td>p</td>
</tr>
<tr>
<td>ississippi#</td>
<td>m</td>
</tr>
<tr>
<td>sissippi#</td>
<td>s</td>
</tr>
<tr>
<td>iissippi#</td>
<td>i</td>
</tr>
<tr>
<td>sissippi#</td>
<td>s</td>
</tr>
<tr>
<td>iissippi#</td>
<td>l</td>
</tr>
<tr>
<td>sissippi#</td>
<td>i</td>
</tr>
<tr>
<td>iissippi#</td>
<td>i</td>
</tr>
</tbody>
</table>

Sort the rows
A famous example

### sorted rotations

<table>
<thead>
<tr>
<th>char</th>
<th>z to decompress. It achieves compression</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>to perform only comparisons to a depth</td>
</tr>
<tr>
<td>x</td>
<td>x transformation. This section describes</td>
</tr>
<tr>
<td>z</td>
<td>x transformation. We use the example and</td>
</tr>
<tr>
<td>t</td>
<td>x treat the right-hand side as the most</td>
</tr>
<tr>
<td>a</td>
<td>x tree for each kbyte input block, etc.</td>
</tr>
<tr>
<td>x</td>
<td>x tree in the output stream, then encodes</td>
</tr>
<tr>
<td>t</td>
<td>x turn, set $z[i]$ to the</td>
</tr>
<tr>
<td>a</td>
<td>x turn, set $z[i]$ to the</td>
</tr>
<tr>
<td>x</td>
<td>x unusual data. Like the algorithm of Man</td>
</tr>
<tr>
<td>a</td>
<td>x use a single set of probabilities table</td>
</tr>
<tr>
<td>x</td>
<td>x using the positions of the suffix in</td>
</tr>
<tr>
<td>i</td>
<td>x value at a given point in the vector $z$</td>
</tr>
<tr>
<td>e</td>
<td>x we present modifications that improve t</td>
</tr>
<tr>
<td>e</td>
<td>x when the block size is quite large. No</td>
</tr>
<tr>
<td>i</td>
<td>x which codes have not been seen in</td>
</tr>
<tr>
<td>e</td>
<td>x with $\delta$ appear in the</td>
</tr>
</tbody>
</table>
Compressing L seems promising...

Key observation:
- L is locally homogeneous

→ L is highly compressible

Algorithm Bzip:
1. Move-to-Front coding of L
2. Run-Length coding
3. Statistical coder

Bzip vs. Gzip: 20% vs. 33%, but it is slower in (de)compression!

Why it works...

Key observation:
- L is locally homogeneous

→ L is highly compressible

Each piece is a context

Compress pieces up to their H0(T)

MTF + RLE avoids the need to partition BWT

An encoding example

T = mississippi
L = ppppppppmmiii

Mtf = 020030000003002000000100000
Mtf = 0300400000040004000000200000
RLE0 = 02131303131310110

Arithmetic/Huffman su +1 simboli....

Be back on indexing: BWT ⊆ SA

L includes SA and T. Can we search within L?
**Implement the LF-mapping**

[Ferragina-Marzini]

- **F start**
  - 1
  - 2
  - m
  - p
  - 7
  - 9

The oracle

**Rank** $s_9 = 3$

We need Generalized R&S

---

**Substring search in $T$ (Count the pattern occurrences)**

- **First step**
  - available $i$

- **Inductive step** (even $fr, j$ for $P[j+1]$)
  1. Take $c = P[j]$.
  2. Find the first $c$ in $L[i][P[j+1]]$
  3. Check if $c$ is in $L[i][P[j+1]]$
  4. If $c$ is not mapped, check if $F$.

Rank is enough

---

**Rank and Select on strings**

- If $\Sigma$ is small (i.e. constant)
  - Build binary Rank data structure per symbol of $\Sigma$
  - Rank takes $O(1)$ time and entropybounded space.

- If $\Sigma$ is large (words?)
  - Need a smarter solution: Wavelet Tree data structure

Another step of reduction:

>> Reduce Rank & Select over arbitrary strings to Rank & Select over binary strings

Binary R/S are key tools

>> tons of papers <<

---

**The FM-index**

- **The result (on small alphabets):**
  - Count($P$): $O(p)$ time
  - Locate($P$): $O(\text{occ log}^m N)$ time
  - Visualize($i$, $L$): $O(L \cdot \log^m N)$ time
  - Space occupancy: $O(N f_0(T)) + o(N)$ bits $\rightarrow o(N)$ if $T$ compressible

- New concept: The FM-index is an opportunistic data structure

Survey of Navarro-Makinen contains many compressed index variants
Is this a technological breakthrough?

The question then was...

- Engineered implementations
- Flexible API to allow reuse and development
- Framework for extensive testing

---

Some figures over hundreds of MBs of data:

- \( \text{Count}(P) \) takes 5 \( \mu \text{sec/char} \), \( \approx 42\% \) space
- \( \text{Extract} \) takes 20 \( \mu \text{sec/char} \)
- \( \text{Locate}(P) \) takes 50 \( \mu \text{sec/occ} \), +10\% space

Trade-off is possible!!!
We need your applications...

(Compressed) Tree Indexing

Paolo Ferragina
Dipartimento di Informatica, Università di Pisa

Part #5: Take-home msg...

Data type: Text

This is a powerful paradigm to design compressed indexes:
1. Transform the input in few arrays
2. Index (Compress) the arrays to support rank/select ops

Where we are...

A data structure is "opportunist" if it indexes a text T within compressed
space and supports three kinds of queries:
- Count(P): Count the occurrences of P occurs in T
- Locate(P): List the occurrences of P in T
- Display(i,j): Print T[i..j]

Key tools: Burrows-Wheeler Transform + Suffix Array
Key idea: reduce P’s queries to few rank/select queries on BWT(T)
Space complexity: function the k-th order empirical entropy of T
Another data format: XML

```
<dblp>
  <book>
    <author>Donald E. Knuth</author>
    <title>The TeXbook</title>
    <publisher>Addison-Wesley</publisher>
    <year>1986</year>
  </book>
  <article>
    <author>Donald E. Knuth</author>
    <author>Ronald W. Moore</author>
    <title>An Analysis of Alpha-Beta Pruning</title>
    <pages>293-326</pages>
    <year>1975</year>
    <volume>6</volume>
    <journal>Artificial Intelligence</journal>
  </article>
</dblp>
```

A key concern: Verbosity...

The problem, in practice...

We wish to devise a (compressed) representation for \( T \) that efficiently supports the following operations:
- Navigational operations: parent(\( u \)), child(\( u, i \)), child(\( u, i, c \))
- Subpath searches over a sequence of \( k \) labels
- Content searches: subpath search + substring

- XML-aware compressors (like XMll, XmPpm, ScmPpm, ...) need the whole decompression for navigation and search
- XML-queriable compressors (like XPress, XGrind, XQzip, ...) achieve poor compression and need the scan of the whole (compressed) file

Theory?

XML-native search engines need this tool as a core block for query optimization and (compressed) storage of information
### A transform for labeled trees

[Farinoga et al. 2005]

**XBW-transform** on trees ∼ **BW-transform** on strings

The **XBW-transform** linearizes T in 2 arrays such that:

- the compression of T reduces to the compression of these two arrays (e.g. gzip, bzip2, ppm, ...)
- the indexing of T reduces to implement generalized rank/select over these two arrays

**Rank&Select are again crucial**

---

### The XBW-Transform

**Step 1:**
Visit the tree in pre-order. For each node, write down its label and the labels on its upward path.

**Permutation of tree nodes**

**Upward labeled paths**

---

### The XBW-Transform

**Step 2:**
Stably sort according to $S_2$.

**Upward labeled paths**

---

### The XBW-Transform

**Key fact:**
Nodes correspond to items in $<S_{ext}, S_2>$

**XBW** can be built and inverted in optimal $O(n)$ time.

**XBW takes optimal $t$ log $|I_t| + t$ bits.**
XBW is navigational

- Rank and select data structures on $S_0$ and $S_1$
- The array $C$ of $|I|$ integers

XBW

Subpath search in XBW

- Inductive step:
- Rank the next char in $I[I+1]$ (i.e., $D$)
- Search for the first and last 'D' in $S_f[I]$ and $S_l[I]$ (jump to their children)

XBW-index

XBzipIndex: XBW + FM-index

- Under patenting by Pisa + Rutgers

- DBLP: 1.75 bytes/node, Pathways: 0.31 bytes/nod, News: 3.91 bytes/nod

- Upto 36% improvement in compression ratio
- Query (counting) time = 8 ms, Navigation time = 3 ms
**Part #6: Take-home msg...**

This is a powerful paradigm to design compact:

1. Transform the input in few arrays
2. Index it (compress) the arrays to support

**Data type**

- Text
- More ops
- More experiments and Applications
- Other data types: 2D, Labeled graphs

---

**I/O issues**

Paolo Ferragina
Dipartimento di Informatica, Università di Pisa

---

**What about I/O-issues ?**

B-tree is ubiquitous in large-scale applications:
- Atomic keys: integers, reals, ...
- Prefix B-tree: bounded length keys (≤ 255 chars)

String B-tree = B-tree + Patricia Trie

- Unbounded length keys
- I/O-optimal prefix searches
- Efficient string updates
- Guaranteed optimal page fill ratio

They are not opportunistic [Bender et al -> FC]

---

**The B-tree**

Search(P):
- O(g/B log2 B) I/Os
- O(log B) I/Os

P[I,p] pattern to search

- O(g/B log2 B) I/Os
- O(log gB) levels
On small sets...

Scan FC(D):
- If P[x]=1, then x++ else jump
- Compare P and S[x] \rightarrow Max_lcp
- If P[Max_lcp+1]=0 go left, else go right, until [L] \leq Max_lcp

Init x = 1
Correct

4 is the candidate position, Max_lcp = 3

Time is \#D + |P| + |FC(D)|
Just S[x] needs to be decoded!

On larger sets...

Patricia Trie
Space = O(|FD|) words

Search(P):
- Phase 1: tree navigation
- Phase 2: Compute LCP
- Phase 3: tree navigation

Two-phase search:
P = SCACSCAC

1 string checked
Space PT = MD

The String B-tree

Succinct PT \rightarrow smaller height in practice
...not opportunistic \Theta(|D| \log |D|) bits

Search:
- O(g/B) I/Os
- O(c log(B)) I/Os

It is dynamic...

O(g/B) I/Os
O(log\_\_n) levels

Cachegon, partial of PT