In-memory processing of big data via succinct data structures

Rajeev Raman

University of Leicester

SDP Workshop, University of Cambridge

Overview

Big Data vs. big data

- Big Data: 10s of TB+.
  - Must be processed in streaming / parallel manner.
- Data mining is often done on big data: 10s-100s of GBs.
  - Graphs with 100s of millions of nodes, protein databases 100s of millions of compounds, 100s of genomes etc.
- Often, we use Big Data techniques to mine big data.
  - Parallelization is *hard* to do well [Canny, Zhao, *KDD'13*].
  - Streaming is inherently limiting.
- Instead of changing the way we *process* the data, why not change the way we *represent* the data?

Processing big data

- Essential that data fits in main memory.
  - Complex memory access patterns: out-of-core $\Rightarrow$ thrashing.
- Data accessed in a complex way is usually represented in a data structure that supports these access patterns.
  - Often data structure is MUCH LARGER than data!
  - Cannot process big data if this is the case.
- Examples:
  - Suffix Tree (text pattern search).
  - Range Tree (geometric search).
  - FP-Tree (frequent pattern matching).
  - Multi-bit Tree (similarity search).
  - DOM Tree (XML processing).

Succinct/Compressed Data Structures

Store data *in memory* in *succinct* or *compressed* format and operate directly on it.

- (Usually) no need to decompress before operating.
- Better use of memory levels close to processor, processor-memory bandwidth.
    - Usually compensates for some overhead in CPU operations.
- **Programs = Algorithms + Data Structures**
    - If compressed data structure implements same/similar ADT to uncompressed data structure, can reuse existing code.

Compression vs. Data Structuring

Answering queries requires an *index* in *in addition to* the data.

Space usage = "space for data" + "space for index".

Index may be larger than the data:

- *Suffix tree:* data structure for indexing a text of n bytes.
  - Supports many indexing and search operations.
  - Careful implementation: $20n$ bytes of index data in worst case [Kurtz, *SPrEx '99*]
- *Range Trees:* data structures for answering 2-D orthogonal range queries on n points.
  - Good worst-case performance but $\Theta(n \log n)$ space.

"Space for Data"

Information-Theoretic Lower Bound

If the object x that you want to represent is drawn from a set S, x must take at least $\log_2 |S|$ bits to represent.

- Example: object x is a binary tree with n nodes.
  - x is from the set S of all binary trees on n nodes.
  - There are $\sim 4^n$ different binary trees on n nodes.
  - Need $\sim \log_2 4^n = 2n$ bits, or 2 bits per node.
  - A normal representation: 2 pointers, or $2 \log_2 n$ bits, per node.

Succinct Data Structuring

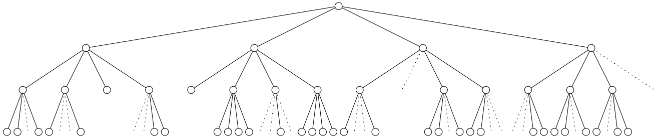
$$\text{Space usage for } x = \underbrace{\text{“space for data”}}_{\text{ITLB for } x} + \underbrace{\text{“space for index”}}_{\text{lower-order term}},$$

and support fast operations on x.

- Not really compression: ITLB applies even to random x.
- Probably over 1000 papers on SDS in algorithms venues.

The “trie” ADT

- Object is a rooted tree with n nodes.
- Each node from a parent to a child is labelled with a *distinct* letter c from an alphabet Σ, where $\Sigma = \{0, \ldots, \sigma - 1\}$.
- All possible children may not be present.
- Represents a collection of strings over Σ.

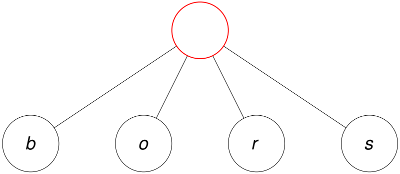


$\Sigma = \{0, 1, 2, 3\}, n = 50$

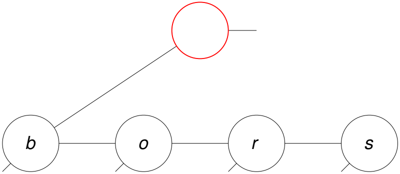
Operations

- $parent(x)$;
- $child(x, c)$;
- $desc(x)$, $nextsib(x)$, $prevsib(x)$, $\ldots$.

Normal Trie Representations

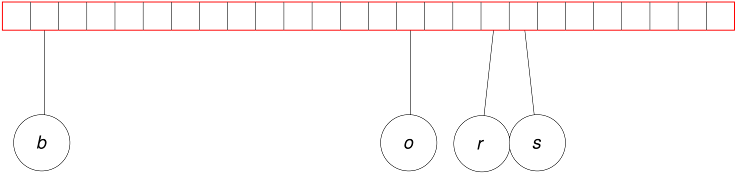


Normal Trie Representations



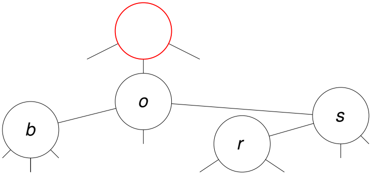
- Each node points to parent, first-child and next-sibling.
  - Space: 3 pointers (192 bits) per node.
  - *child*: $O(\sigma)$ time.

Normal Trie Representations



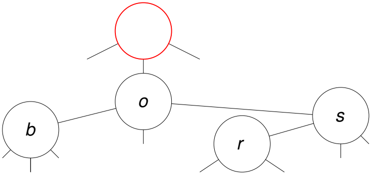
- Each node has array of σ pointers, one to each possible child.
    - Space: $\sigma + 1$ pointers per internal node.
    - *child*: $O(1)$ time.

Normal Trie Representations



- *Ternary search tree* [Bentley/Sedgewick, *SODA'97*]. Siblings arranged in a binary tree.
  - Space: 4 pointers (256 bits) per node.
  - *child*: $O(\lg \sigma)$ time.

Normal Trie Representations



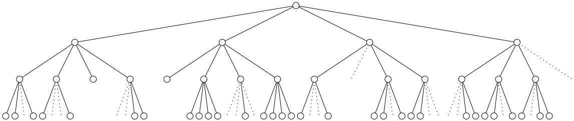
- *Ternary search tree* [Bentley/Sedgewick, *SODA'97*]. Siblings arranged in a binary tree.
  - Space: 4 pointers (256 bits) per node.
  - *child*: $O(\lg \sigma)$ time.

- **ITLB** $= \left\lceil \log_2 \left(\frac{1}{\sigma n+1} \binom{\sigma n+1}{n} \right) \right\rceil \sim n \log_2 \sigma + O(n)$ bits.

▷ One *character* per node.

Succinct Tries

- Output a 1. Then visit each node in level-order and output σ bits that indicate which labels are present. [Jacobson, *FOCS'89*]



1 1111 1111 1111 1011 1110 1101 1001 0000 0011 0000 1111 0010 1111 1001 1101 1100 0011
1101 1011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

- Bit-string is of length $\sigma n + 1$ bits. It has n 1s.
- Its ITLB is $\left\lceil \log_2 \binom{\sigma n+1}{n} \right\rceil \sim n \log_2 \sigma + O(n)$ bits.
- Representation is *static*, but a lot of operations in $O(1)$ time.

Dynamic Tries

- ADT:
  - *parent*(x);
  - *child*(x, c);
  - *add*(x, c);
- Bonsai tree [Darragh et al., *Soft. Prac. Exp'93*],[PR, *SPIRE'15*].
- Data structure: open hash table of $(1+\epsilon)n$ entries.
  - Nodes of trie reside in hash table.
  - ID of a node: location where it resides.
  - ID of child labelled c of x:
    - Create key $\langle x, c\rangle$ and insert.
- Hash table entries only store "quotients", require only $\log_2 \sigma + O(1)$ bits.
- Space usage $(1+\epsilon)n\log_2 \sigma + O(n)$ bits, $O(1)$ time.
- Fast in practice (2-3 times slower than TST).

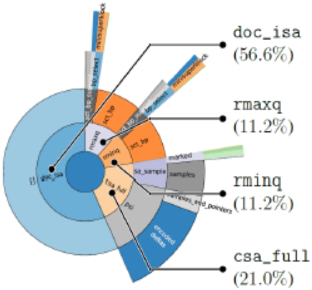
Applications

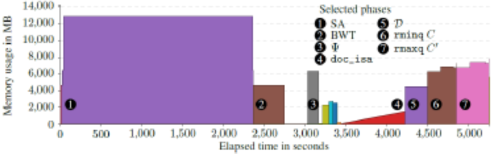
SDS have been applied in a number of domains:

- Information retrieval.
- NGS: Bowtie read aligner.
- Representing XML data:
    - "SiXML" project, XML DOM with order of magnitude less space.
    - Data store for Zorba XQuery processor.
- Many data mining tasks (papers in KDD'14, KDD'16).

Library sdsl-lite

- Comprehensive (but low-level) library. [Gog et al., *SEA '14*]
- Structured to facilitate flexible prototyping of new high-level structures (building upon bases such as bit vectors).
- Robust in terms of scale, handling input sequences of arbitrary length over arbitrary alphabets.
- Serialization to disk and loading, memory usage visualization.





Conclusions

- Use of succinct data structures can allow scalable processing of big data *using existing algorithms*.
  - With machines with 100s of GB RAM, maybe even Big Data can be processed using compressed data structures.
- Many of the basic theoretical foundations have been laid, and succinct data structures have never been easier to use.
- Succinct data structures need to be chosen and used appropriately. Optimized to ADT.
  - Even "simple" operations can't necessarily be added later.
  - E.g. in Bonsai tree, all of a node's descendants' IDs are derived from its ID; can't delete internal nodes cheaply.
  - Single-threaded dynamic SDS much less developed, let alone concurrent SDS.
- Many individual applications, but no complex systems built around SDS.