Indexing string dictionaries

Pesaresi seminars, 19/01/2024

Mattia Odorisio, Mariagiovanna Rotundo

The problem

What and Why?

What?

Index a sorted dictionary *D* of **strings** s_1, …, s_n while supporting operations such us:

- **Access**(*i*): retrieve the *i-th* string of the dictionary *D*;
- **Rank**(*P*): find the lexicographic position of *P* in the dictionary *D*.

Why?

Important problem because it appears in **a lot of applications** like, for example, databases, search engine, and bioinformatics tools

Different kind of existing solutions

- **Classic** approaches
- **Learned** approaches

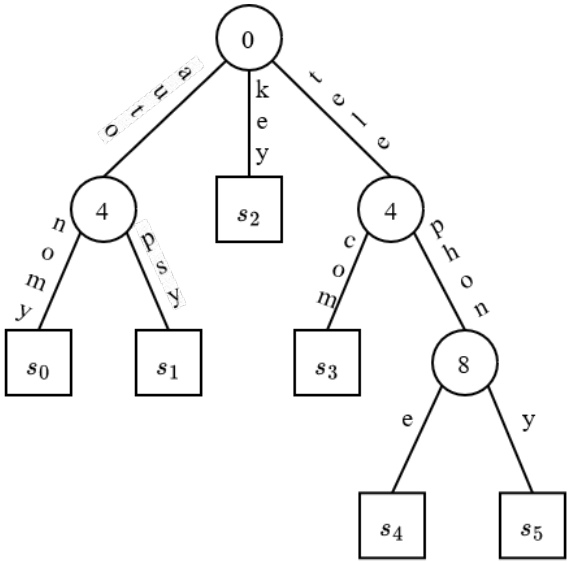
Classic solutions

State-of-the-art solutions

Based on **Tries**:

- Fast Succinct Trie
- Path Decomposed Trie
- Compressed Collapsed Trie
- Patricia Trie

Trie: a multi-way tree whose edges are labeled by characters of the indexed strings



Compacted trie

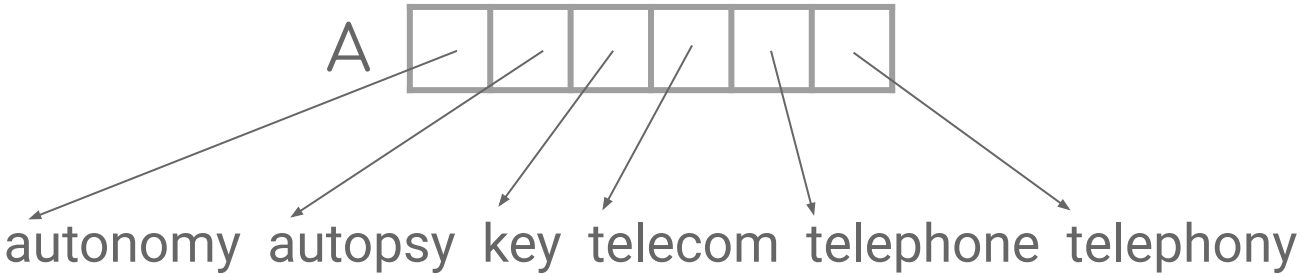
State-of-the-art ideas

Different approaches for these solutions:

- exploit properties of **density** of Tries
- reduce the **height** of the Trie
- compacts subtries
- 2-level approach to reduce the amount of information kept in internal memory

Used in practice solution

Array of string pointers + binary search



So what?

Different solution to index string dictionaries (of different sizes) exist.

Problem

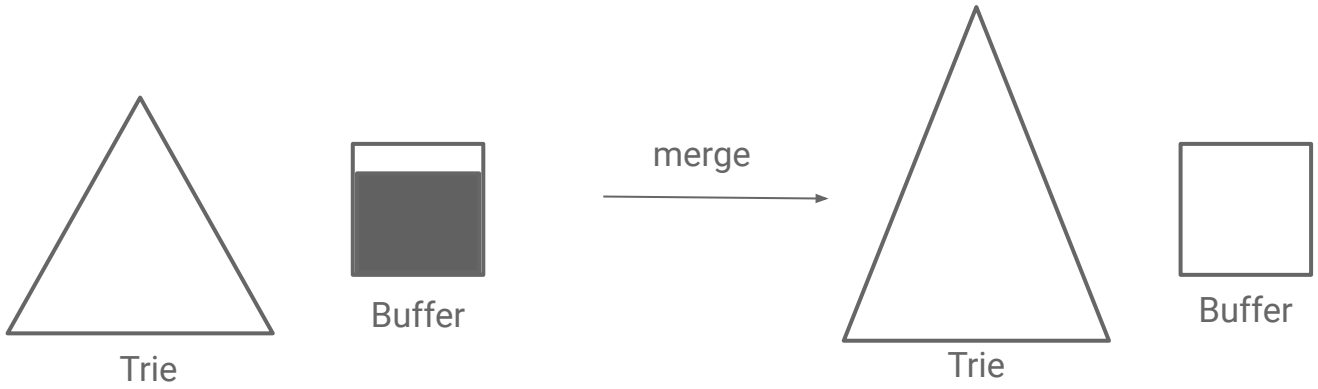
All these solutions are proposed to index **static** string dictionaries.

What for dynamic dictionaries?

OPEN PROBLEM

Used in practice solution

- Buffering
- Frequent **rebuild** of static data structures



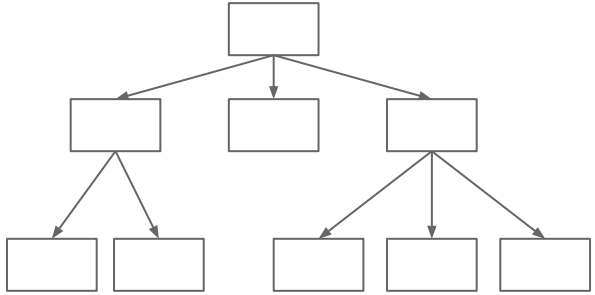
Pointer solutions

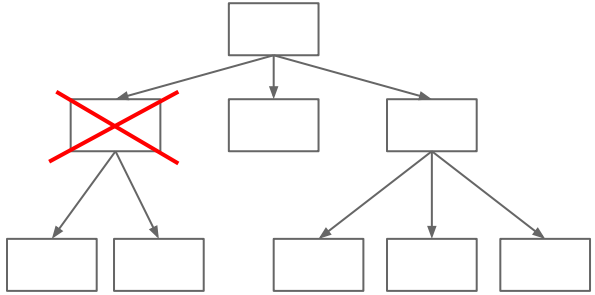
Pro:

- Easy to insert and delete nodes: insert/delete the node and adjust pointers

Cons:

- A lot of space (not succinct)
- Good complexity but random memory accesses





Dynamic Tries

- **HAT-trie**: nontrivial space and construction time bounds
- Trie based on **Bonsai trie**: some operations on the trie are brute force
- **Jansson's dynamic trie**: good in theory but difficult to efficiently implement
- **Ctrie++**: state-of-the-art solution

Dynamic Tries: what if the dataset is very big?

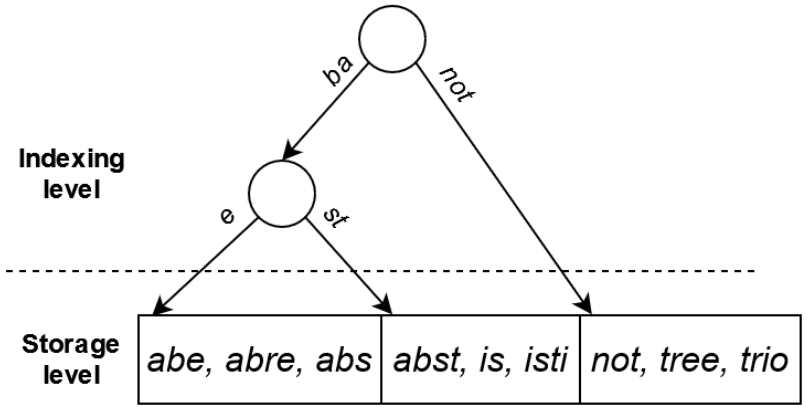
Ctrie++: high **space requirements**. Can require more internal memory than the one available

What can we do?

Static solution for big datasets

A 2-level design scheme:

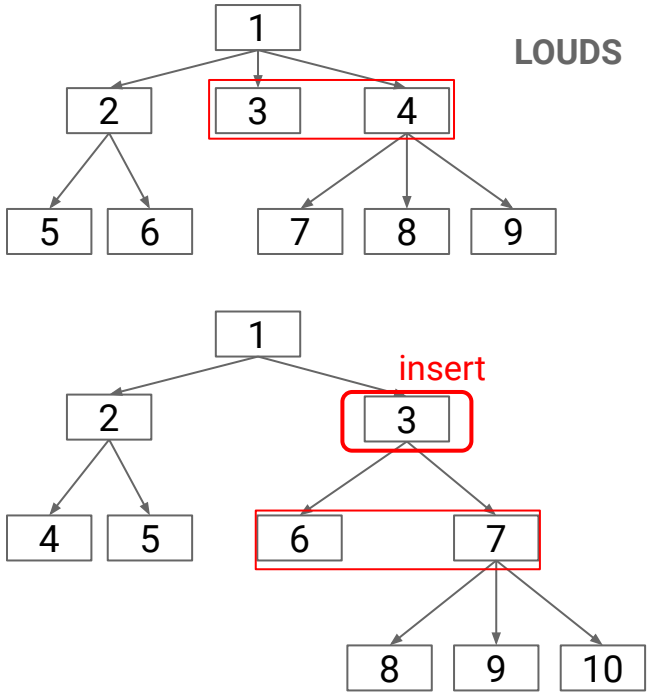
- **Storage level,** where the strings are compressed via Rear Coding and stored in blocks.
- **Indexing level,** where the first string of each block of the storage level is indexed via a succinct Patricia Trie.



Succinct representations of Tries

dynamic succinct encoding for tries:

- LOUDS: insert and deletes of internal nodes not (efficiently) supported
- BP and DFUDS operations in O(log n) time and O(n/logn) bits of extra space



Existing studies

Many studies in literature:

- Dynamic bitvectors
- Dynamic rank and select operations
- Dynamic arrays

Each of this study introduce a slowdown and extra space.

Learned indexes

Learned Data Structures

Data structures enhanced by **machine learning**.

They can reveal and exploit **patterns** and **trends** in the input data for achieving more **efficiency** in **time** and **space**, compared to previously known solutions.

Let’s start from numbers Revisiting this slide…

What?

Index a sorted sequence *S* of **values** $v_1, \ldots, v_n$ while supporting operations such us:

- **Access(*i*)**: retrieve the *i-th* value of the sequence *S*;
- **Rank(*x*)**: Given x ∈ U (universe), return the number of elements in S that are less than or equal to x;

Why?

- Member(x)
- Lookup(x)
- **Predecessor(x)**
- **Range(x, y)**
- Insert(x)
- Delete(x)

Anticipating some questions...

Can we compute all these operation with...?

- Binary search: YES, but it takes O(log n) time
- Hash Tables/Bloom filters etc.: Only some operations, (recall $x \in U$, we cannot solve predecessor and range queries)
- B^+-Tree: YES, but $\Theta(n/B)$ space, and $O(\log_B n)$ I/Os, can we do better?
- Other ideas?

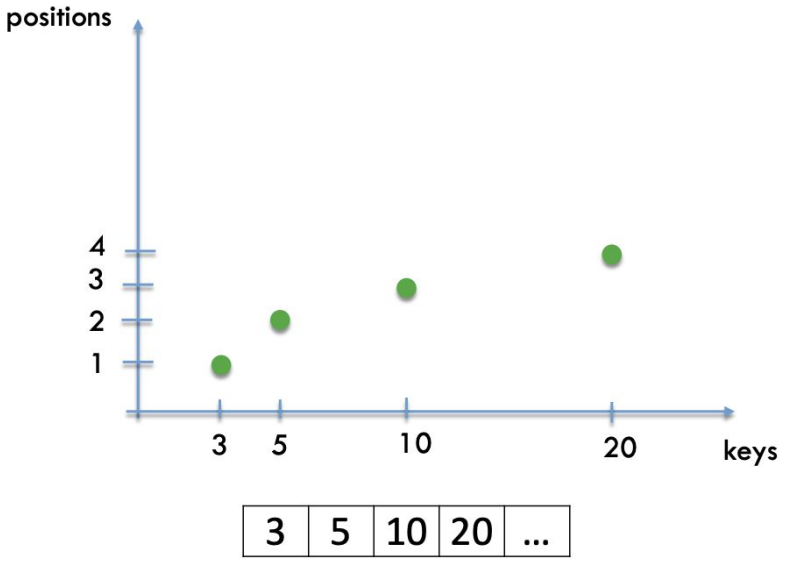
The Rank problem: a geometrical perspective

Let's start from numbers…

Machine Learning?

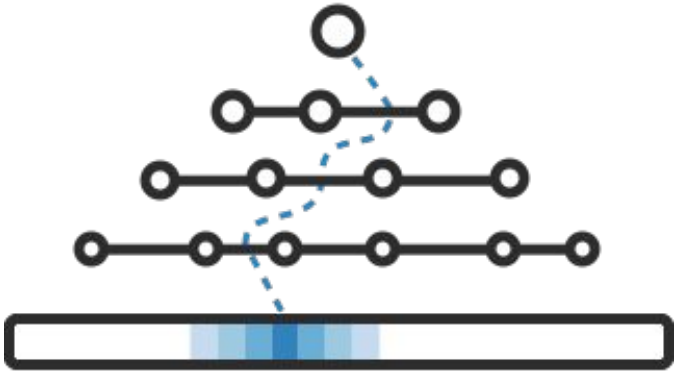
We need a function.

Function: given a value, find its position



So what is a learned index

A **learned index** is a **ML** model, that given a key **x** and a sorted sequence **S**, returns the approximate position of **x** in **S**.



Learned indexes, an example: RMI

RMI: a DAG of models.

Example with three layers, the prediction done by Model 1.1 (root) allows to pick a Model 2.?, that is used to pick 3.? that predicts the rank.

Monotonic? x < y ↔ pred(x) ≤ pred(y) it depends on the models and the implementation.

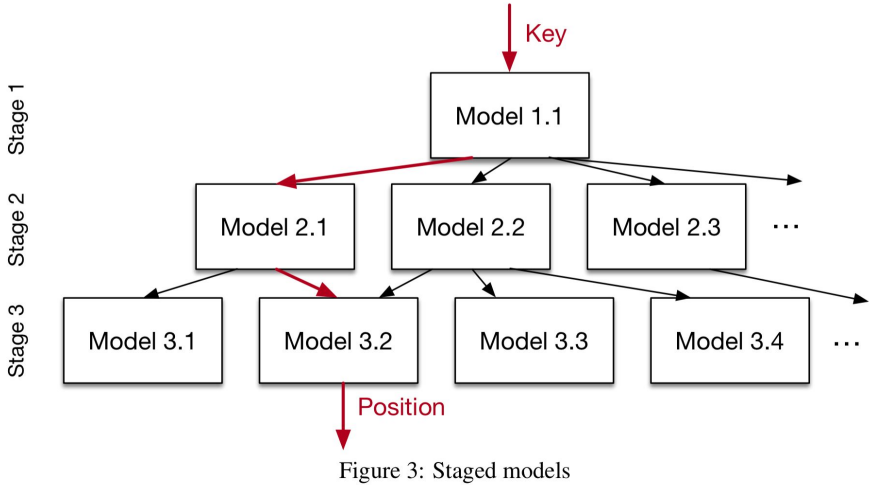


Figure 3: Staged models

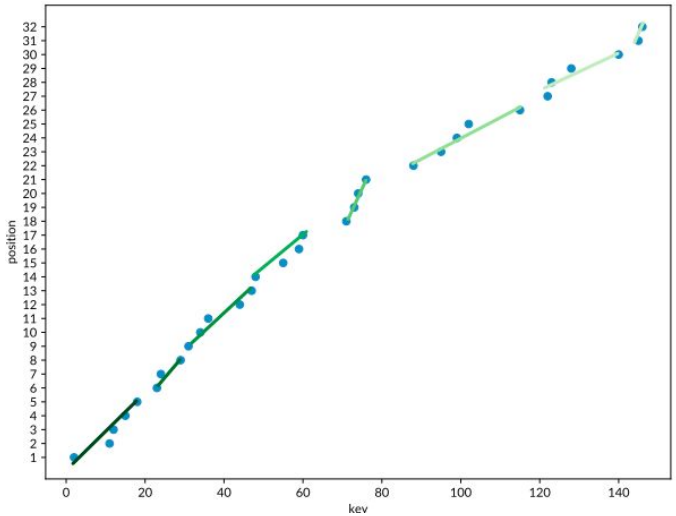
Three-Layer Recursive Model Index

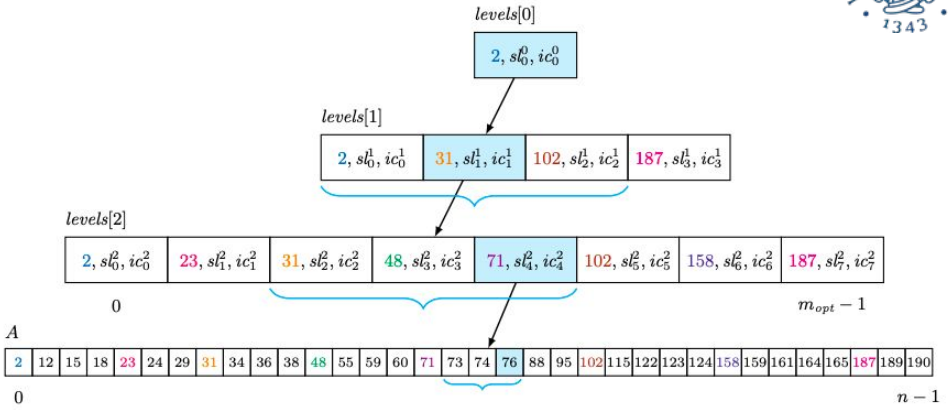
Learned indexes, another example PGM-Index

The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds

Paolo Ferragina
University of Pisa, Italy
paolo.ferragina@unipi.it

Giorgio Vinciguerra
University of Pisa, Italy
giorgio.vinciguerra@phd.unipi.it





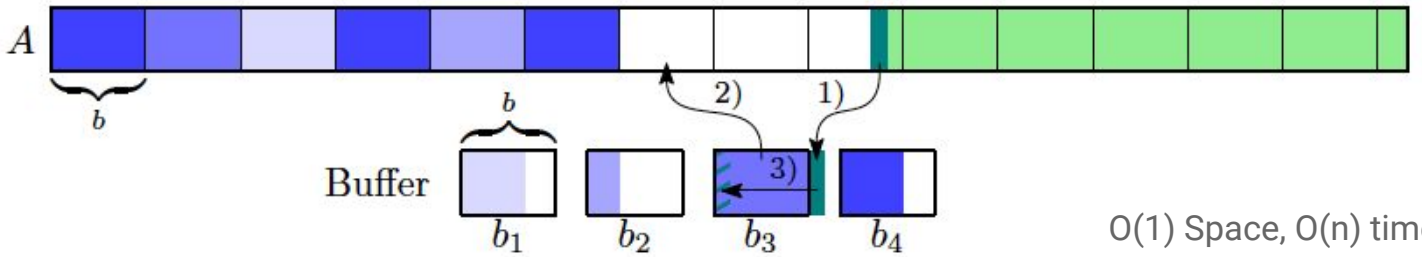
It builds many linear models (**segments**), such that the error is bounded by a threshold ε.
It is recursive: we need to find the right segment at query time..

Another practical application: sorting!

Background: **samplesort** a.k.a. **muti-way quicksort**

Partitioning:

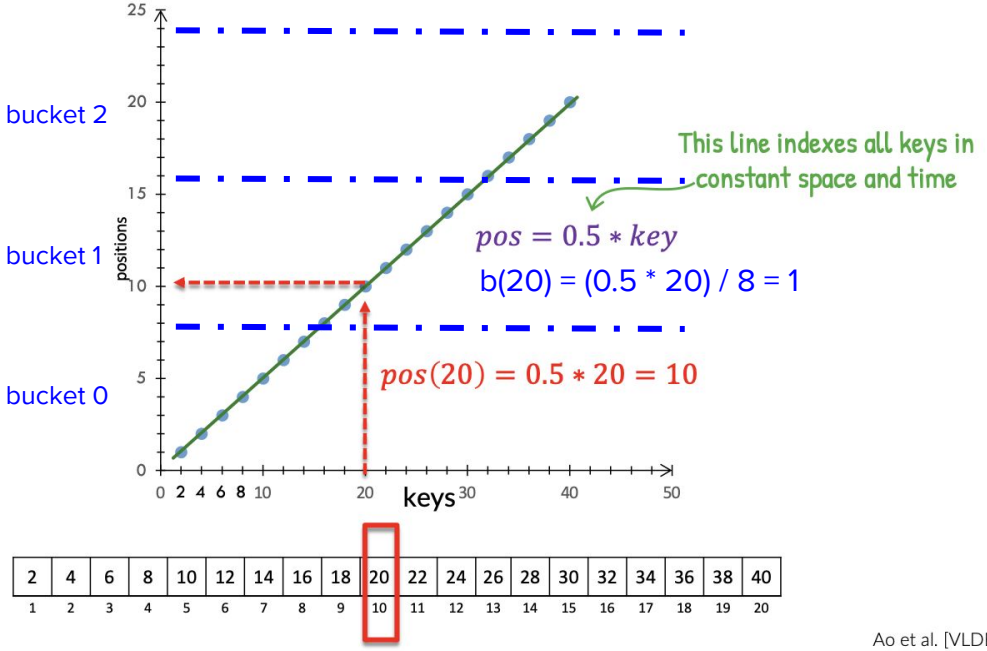
- Allocate fixed size buffers called **fragments**, one for each partition
- Scan the input and classify each item in a **fragment**
- Once a fragment is full, copy it back in the original input array
- **Defragmentation** to get the partitions



O(1) Space, O(n) time, O(n/B) I/Os

Another practical application: sorting!

- **Multi-way Quicksort** where the bucketing is driven by a **learned model [O(1)]**, instead of classical binary-search **[O(log k)]** over a set of pivots.
- Model trained over a small sample of the data (1%).



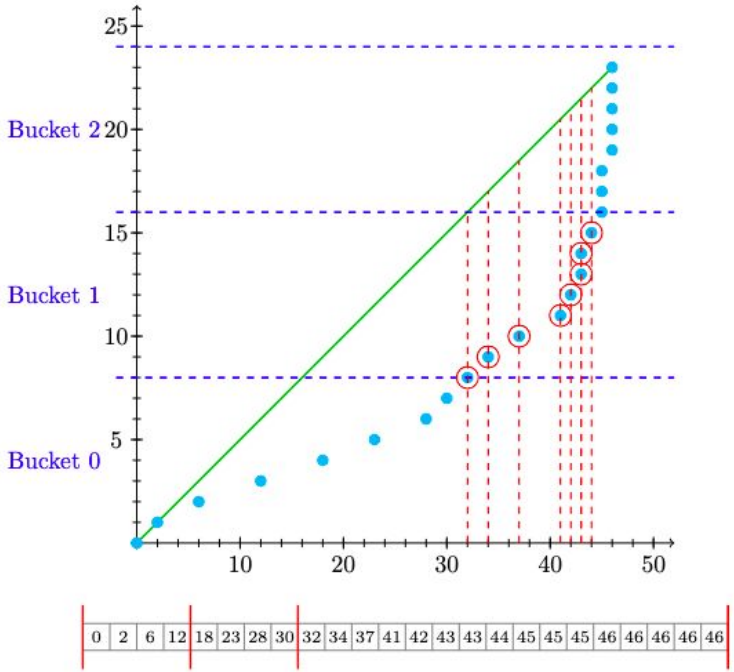
Ao et al. [VLDB

Another practical application: sorting!

The **linear model** is simple but not effective!

An error happens when the linear model assigns an element to an erroneous bucket.

Recall the optimal case of the quicksort is when all the partitions have the same size.



Which is the better learned index for sorting?

Many details have to be considered...

- Training time
- Inference time
- Monotonicity of the index
- How balanced are the partitions created

...led to a new index proposal.

Learned data structures for strings

OPEN PROBLEM

Why is that an open problem?

All the strings can be encoded and treated like numbers, so we can reuse the same indexes.

- Strings have a variable length, complicating both model inference and memory layout.
- Strings are much larger objects which are expensive to store, compare, and manipulate.
- Strings tend to have worse distributional properties than integer keys (common prefixes, substrings etc.).

Anticipating some questions...

Can we simplify the problem? Some ideas:

- Can we discard the least common prefix? YES, of course.
- Can we truncate the string to have a bounded representation? NO, looking only at the first characters is *often* enough to determine the approximate rank. But what happens when the distribution is far from being linear?

Existing proposals

- **RMI for strings:** Truncate the string and treat them as numbers, if the error exceed a given threshold, use a B^+-Tree.

- **RSS:** Compress the strings, starts as a trie, then it switches to a **radix spline model** when the error below a threshold**.**

- **SIndex:** Greedily partitions the input into groups that are approximated with a **linear model** with mean error below a threshold. It uses a **piecewise linear regression model** to route the queried string to the correct group.

What we have seen...

- Classic approaches
  - state-of-the-art and used-in-practice solutions to index **static** string dictionaries
  - **existing** approaches for dynamic string dictionaries and **a possible idea** to index **dictionaries** that are **huge**
- Learned approaches
  - what are learned data structures
  - two **examples**
  - how to use them for **sorting**
  - open problem: **Learned string indexing**

Thank you!