Note of Succinct: Enabling Queries on Compressed Data

Cheng Zhao

August 27, 2017

1 Motivation

Evaluation of popular open-source data stores (MongoDB, Cassandra) using real-world datasets shows that indexes can be much as 8x larger than the input data size. Existing data stores either **resort** to using complex memory management techniques for identifying and caching "hot" data or **simply executing queries off-disk or off-SSD**. In either case, latency and throughput advantages of indexes drop compared to in-memory query execution.

2 Succinct Overview

- a distributed data store
- operates at a new point in the design space
- memory efficiency close to data scans and latency close to indexes
- is able to store more data in memory, avoiding latency and throughput degradation due to off-disk or off-SSD query execution

Contributions:

- Enables efficient queries directly on a compressed representation of the input data
  - a new data structure
  - a new query algorithm
- Efficiently supports data appends by chaining multiple stores
  - a small log-structure store optimized for fine-grained appends
  - an intermediate store optimized for query efficiency while supporting bulk appends
  - an immutable store that stores most of the data
- Exposes a minimal, yet powerful, API that operates on flat unstructured files

2.1 Succinct Interface

For string `abbcdeabczabgz`,

- `search(f, "ab")` returns `[0, 6, 10]`.
- `wildcardsearch(f, prefix="ab", suffix="z", dist=2)` returns `[6, 9]` for `abcz`, `[10, 13]` for `abgz`.

For semi-structured data (KV), like figure 1, transforming the input data into flat files.

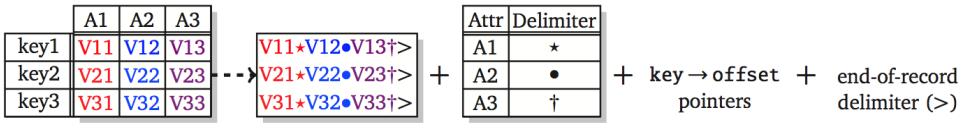


Figure 1: Succinct supports queries on semi-structured data by transforming the input data into flat files

3 Querying on Compressed Data

3.1 Compressed Suffix array and Succinct data representation

Suffix array Wikipedia

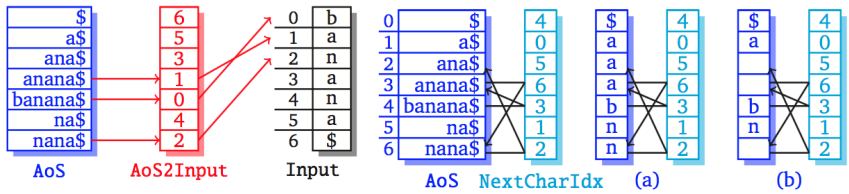
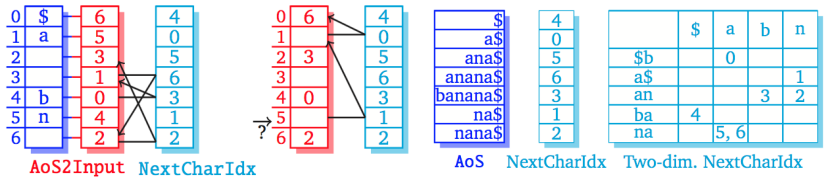


Figure 2

Figure 2 (a) shows that AoS stores the first char of each suffix only. Since suffixes are sorted, only the first AoS index at which each character occurs (e.g., $\{(\$, 0), (a, 1), (b, 4), (n, 5)\}$) need be stored. Succinct uses *Skewed Wavelet Tree* to compress each row independently.

Succinct uses sampling by "value" strategy. For sampling rate α, Succinct stores all AoS2Input values that are a multiple of α. This allows storing each sampled value `val` as `val`$/\alpha$, leading to a more space-efficient representation. Using α `= 2` for example of Figure 3a, for instance, the sampled

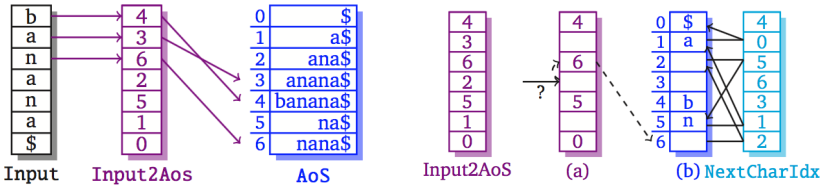
AoS2Input values are `{6,0,4,2}`, which can be stored as `{3,0,2,1}`. Sampled Input2AoS then becomes `{1,3,2,0}` with i-th value being the index into sampled AoS2Input where i is stored. Succinct stores a small amount of additional information to locate sampled AoS2Input indexes.



(a) (b)

Figure 3: 3a for reducing the space usage of AoS2Input. 3b for two-dimensional NextCharIdx representation.

For instance, consider the query `search(anan)`; all occurrences of string `"nan"` are contained in the cell `<n,an>`. To find all occurrences of string `anan`, our algorithm performs a binary search only in the cell `<a,na>` in the next step. After this step, the algorithm has the indexes for which suffixes start with `"a"` and are followed by `"nan"`, the desired string. For a string of length m, the above algorithm performs $2(mt1)$ binary searches, two per NextCharIdx cell.



(a) (b)

Figure 4: 4a for The Input2AoS provides the inverse mapping of AoS2Input. 4b for reducing the space usage of Input2AoS.

`extract` functionality is shown in Figure 4. For instance, to execute `extract(3, 3)`, we find the next smaller sampled index `(Input2AoS[2])` and corresponding suffix `(AoS[2]="nana$")`. We then remove the first character since the difference between the desired index and the closest sampled index was 1; hence the result `"ana$"`.

4 Succinct Store

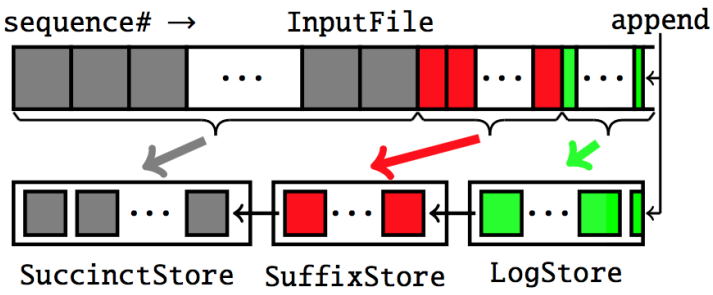


Figure 9: Succinct uses a write-optimized LogStore that supports fine-grained appends, a query-optimized SuffixStore that supports bulk appends, and a memory-optimized SuccinctStore. New data is appended to the end of LogStore. The entire data in LogStore and SuffixStore constitutes a single partition of SuccinctStore. The properties of each of the stores are summarized in Table 1.

Table 1: Properties of individual stores. Data size estimated for 1TB original uncompressed data on a 10 machine 64GB RAM cluster. Memory estimated based on evaluation (§6).

	Succinct Store	**Suffix Store**	**Log Store**
Stores	Comp. Data (§3.1)	Data + AoS2Input	Data + Inv. Index
Appends	-	Bulk	Fine
Queries	§3.2	Index	Scans+ Inv. Index
#Machines	$n-2$	1	1
%Data(est.)	$> 99.98\%$	$< 0.016\%$	$< 0.001\%$
Memory	$\approx 0.4\times$	$\approx 5\times$	$\approx 9\times$

Figure 5

4.1 LogStore

Succinct partitioning LogStore data into multiple partitions, each containing a small amount of data. LogStore executes an `extract` request by reading the data starting at the offset specified in the request. While executing `search` uses an "inverted index" per partition that supports fast updates.

4.2 SuffixStore

- SuffixStore accumulates and queries much more data than LogStore before initiating compression
- SuffixStore supports bulk data appends without updating any existing data

SuffixStore stores uncompressed AoS2Input array and executes search queries via binary search. SuffixStore achieves the second goal us- ing excessive partitioning, with overlapping partitions similar to LogStore. Bulk appends from LogStore are executed at partition granularity, with the entire LogStore data constituting a single partition of SuffixStore.

4.3 SuccinctStore

SuccinctStore is designed for memory efficiency. It uses the **entropy-compressed** representation. Two challenges:

- 1. Succinct will lose its advantages if input data is too large to fit in memory even after compression using default parameters because of the multiple tunable fixed parameters (*e.g.*, AoS2Input and Input2AoS sampling rate and string lengths for indexing NextCharIdx rows). [**Solution:** Enables applications to select AoS2Input and Input2AoS sampling rate, but it will cost higher latency.]
- 2. Succinct performance may deteriorate for workloads that are skewed towards particular SuccinctStore partitions. [**Solution:** Increasing the memory footprint of over- loaded partitions, thus disproportionately speeding up these partitions for skewed workloads.]