# Succinct: Enabling Queries on Compressed Data

Rachit Agarwal
UC Berkeley

Anurag Khandelwal
UC Berkeley

Ion Stoica
UC Berkeley

## Abstract

*Succinct* is a data store that enables efficient queries directly on a compressed representation *of the input data*. Succinct uses a compression technique that allows random access into the input, thus enabling efficient storage and retrieval of data. In addition, Succinct natively supports a wide range of queries including count and search of arbitrary strings, range and wildcard queries. What differentiates Succinct from previous techniques is that Succinct supports these queries *without* storing indexes — all the required information is embedded within the compressed representation.

Evaluation on real-world datasets show that Succinct requires an order of magnitude lower memory than systems with similar functionality. Succinct thus pushes more data in memory, and provides low query latency for a larger range of input sizes than existing systems.

## 1 Introduction

High-performance data stores, *e.g.* document stores [1, 6], key-value stores [5,9,23,24,26,38,39,44] and multi-attribute NoSQL stores [3, 19, 21, 25, 35, 50], are the bedrock of modern cloud services. While existing data stores provide efficient abstractions for storing and retrieving data using primary keys, interactive queries on values (or, secondary attributes) remains a challenge.

To support queries on secondary attributes, existing data stores can use two main techniques. At one extreme, systems such as column oriented stores, simply scan the data [10, 36]. However, data scans incur high latency for large data sizes, and have limited throughput since queries typically touch all machines[1]. At the other extreme, one can construct indexes on queried attributes [3, 6, 35]. When stored in-memory, these indexes are not only fast, but can achieve high throughput since it is possible to execute each query on a single machine. The main disadvantage of indexes is their high memory footprint. Evaluation of popular open-source data stores [6,35] using real-world datasets (§6) shows that indexes can be as much as 8× larger than the input data size. Traditional compression techniques can reduce the memory footprint but suffer from degraded throughput since data needs to be decompressed even for simple queries. Thus, existing data stores either resort to using complex memory management techniques for identifying and caching "hot" data [5, 6, 26, 35] or simply executing queries off-disk or off-SSD [25]. In either case, latency and throughput advantages of indexes drop compared to in-memory query execution.

We present Succinct, a distributed data store that operates at a new point in the design space: memory efficiency close to data scans and latency close to indexes. Succinct queries on secondary attributes, however, touch all machines; thus, Succinct may achieve lower throughput than indexes when the latter fits in memory. However, due to its low memory footprint, Succinct is able to store more data in memory, avoiding latency and throughput degradation due to off-disk or off-SSD query execution for a much larger range of input sizes than systems that use indexes.

Succinct achieves the above using two key ideas. First, Succinct stores *an entropy-compressed representation of the input data* that allows random access, enabling efficient storage and retrieval of data. Succinct's data representation natively supports count, search, range and wildcard queries *without* storing indexes — all the required information is embedded within this compressed representation. Second, Succinct *executes queries directly on the compressed representation*, avoiding data scans and decompression. What makes Succinct a unique system is that it not only stores a compressed representation of the input data, but also provides functionality similar to systems that use indexes along with input data.

Specifically, Succinct makes three contributions:

- Enables efficient queries directly on a compressed representation of the input data. Succinct achieves this using (1) a new data structure, in addition to adapting data structures from theory literature [32, 46–48], to compress the input data; and (2) a new query algorithm that executes random access, count,

[1] Most data stores shard data by rows, and one needs to scan all rows. Even if data is sharded by columns, one needs to touch multiple machines to construct the row(s) in the query result.

search, range and wildcard queries directly on the compressed representation (§3). In addition, Succinct provides applications the flexibility to tradeoff memory for faster queries and vice versa (§4).

- Efficiently supports data appends by chaining multiple stores, each making a different tradeoff between write, query and memory efficiency (§4): (1) a small log-structured store optimized for fine-grained appends; (2) an intermediate store optimized for query efficiency while supporting bulk appends; and (3) an immutable store that stores most of the data, and optimizes memory using Succinct's data representation.
- Exposes a minimal, yet powerful, API that operates on flat unstructured files (§2). Using this simple API, we have implemented many powerful abstractions for semi-structured data on top of Succinct including document store (*e.g.*, MongoDB [6]), key-value store (*e.g.*, Dynamo [23]), and multi-attribute NoSQL store (*e.g.*, Cassandra [35]), enabling efficient queries on both primary and secondary attributes.

We evaluate Succinct against MongoDB [6], Cassandra [35], HyperDex [25] and DB-X, an industrial columnar store that supports queries via data scans. Evaluation results show that Succinct requires 10−11× lower memory than data stores that use indexes, while providing similar or stronger functionality. In comparison to traditional compression techniques, Succinct's data representation achieves lower decompression throughput but supports point queries directly on the compressed representation. By pushing more data in memory and by executing queries directly on the compressed representation, Succinct achieves dramatically lower latency and higher throughput (sometimes an order of magnitude or more) compared to above systems even for moderate size datasets.

## 2 Succinct Interface

Succinct exposes a simple interface for storing, retrieving and querying flat (unstructured) files; see Figure 1. We show in §2.1 that this simple interface already allows us to model many powerful abstractions including MongoDB [6], Cassandra [35] and BigTable [19], enabling efficient queries on semi-structured data.

The application submits and compresses a flat `file` using `compress`; once compressed, it can invoke a set of powerful primitives directly on the compressed file. In particular, the application can append new data using `append`, can perform random access using `extract` that returns an uncompressed buffer starting at an arbitrary offset in original `file`, and count number of occurrences of any arbitrary string using `count`.

```
f = compress(file)
append(f, buffer)
buffer = extract(f, offset, len)
cnt = count(f, str)
[offset1, ...] = search(f, str)
[offset1, ...] = rangesearch(f, str1, str2)
[[offset1, len1], ...]
  = wildcardsearch(f, prefix, suffix, dist)
```

**Figure 1:** Interface exposed by Succinct (see §2).

Arguably, the most powerful operation provided by Succinct is `search` which takes as an argument an *arbitrary* string (*i.e.*, not necessarily word-based) and returns offsets of all occurrences in the uncompressed `file`. For example, if `file` contains `abbcdeabczabgz`, invoking `search(f, "ab")` will return offsets `[0, 6, 10]`. While `search` returns an array of offsets, we provide a convenient iterator interface in our implementation. What makes Succinct unique is that `search` not only runs on the *compressed* representation but is also efficient, that is, does not require scanning the `file`.

Succinct provides two other search functions, again on arbitrary input strings. First, `rangesearch` returns the offsets of all strings between `str1` and `str2` in lexicographical order. Second, `wildcardsearch(f, prefix, suffix, dist)` returns an array of tuples. A tuple contains the offset and the length of a string with the given `prefix` and `suffix`, and whose distance between the prefix and suffix does not exceed `dist`, measured in number of input characters. Suppose again that file `f` contains `abbcdeabczabgz`, then `wildcardsearch(f, "ab", "z", 2)` will return tuples `[6, 9]` for `abcz`, and `[10, 13]` for `abgz`. Note that we do not return the tuple corresponding to `abbcdeabcz` as the distance between the prefix and suffix of this string is greater than 2.

### 2.1 Extensions for semi-structured data

Consider a logical collection of records of the form `(key, avpList)`, where `key` is a unique identifier, and `avpList` is a list of attribute value pairs, i.e., `avpList = ((attrName1, value1),... (attrNameN, valueN))`. To enable queries using Succinct API, we encode `avpList` within Succinct data representation; see Figure 2. Specifically, we transform the semi-structured data into a flat file with each attribute value separated by a delimiter unique to that attribute. In addition, Succinct internally stores a mapping from each attribute to the corresponding delimiter, and a mapping from `key` to `offset` into the flat file where corresponding `avpList` is encoded.

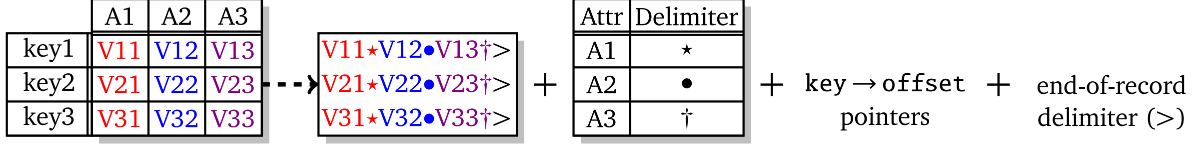



**Figure 2:** Succinct supports queries on semi-structured data by transforming the input data into flat files (see §2.1).

Succinct executes `get` queries using `extract` API along with the `key→offset` pointers, and `put` queries using the `append` API. The `delete` queries are executed lazily, similar to [8,10], using one explicit bit per record which is set upon record deletion; subsequent queries ignore records with set bit. Applications can also query individual attributes; for instance, search for string `val` along attribute `A2` is executed as `search(val•)` using the Succinct API, and returns every `key` whose associated attribute `A2` value matches `val`.

**Flexible schema, record sizes and data types.** Succinct, by mapping semi-structured data into a flat file and by using delimiters, does not impose any restriction on `avpList`. Indeed, Succinct supports single-attribute records (*e.g.*, Dynamo [23]), multiple-attribute records (*e.g.*, BigTable [19]), and even a collection of records with varying number of attributes. Moreover, using its key → offset pointers, Succinct supports the realistic case of records varying from a few bytes to a few kilobytes [17]. Succinct currently supports primitive data types (`strings, integers, floats`), and can be extended to support a variety of data structures and data types including composite types (`arrays, lists, sets`). See [16] for a detailed discussion.

## 3 Querying on Compressed Data

We describe the core techniques used in Succinct. We briefly recall techniques from theory literature that Succinct uses, followed by Succinct's entropy-compressed representation (§3.1) and a new algorithm that operates directly on the compressed representation (§3.2).

**Existing techniques.** Classical search techniques are usually based on tries or suffix trees [13, 49]. While fast, even their *optimized representations* can require 10–20× more memory than the input size [33, 34]. Burrows-Wheeler Transform (BWT) [18] and Suffix arrays [12, 40] are two memory efficient alternatives, but still require 5× more memory than the input size [33]. FM-indexes [27–30] and Compressed Suffix Arrays [31, 32, 46–48] use compressed representation of BWT and suffix arrays, respectively, to further reduce the space requirement. Succinct adapts compressed suffix arrays due to their simplicity and relatively better performance

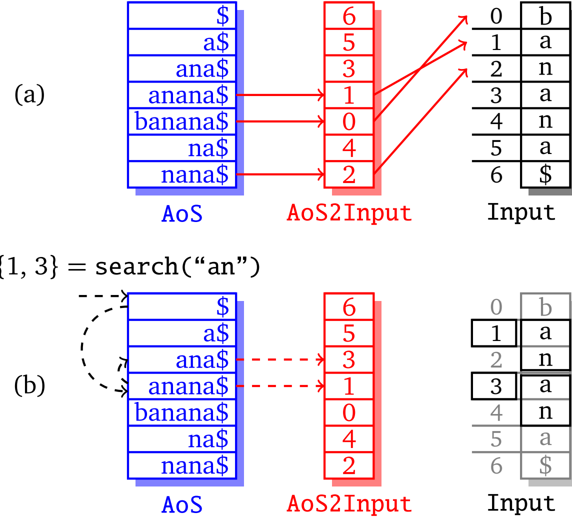



**Figure 3:** An example for input file banana$. AoS stores suffixes in the input in lexicographically sorted order. (a) AoS2Input maps each suffix in AoS to its location in the input (solid arrows). (b) Illustration of search using AoS and AoS2Input (dashed arrows). Suffixes being sorted, AoS allows binary search to find the smallest AoS index whose suffix starts with searched string (in this case "an"); the largest such index is found using another binary search. The result on the original input is showed on the right to aid illustration.

for large datasets. We describe the basic idea behind Compressed Suffix Arrays.

Let Array of Suffixes (AoS) be an array containing all suffixes in the input file in lexicographically sorted order. AoS along with two other arrays, AoS2Input and Input2AoS[2], is sufficient to implement the search and the random access functionality without storing the input file. This is illustrated in Figure 3 and Figure 4.

Note that for a file with $n$ characters, AoS has size $O(n^2)$ bits, while AoS2Input and Input2AoS have size $n\lceil \log n \rceil$ bits since the latter two store integers in range 0 to $n-1$. The space for AoS, AoS2Input and Input2AoS is reduced by storing only a subset of values; the remaining values are computed on the fly using a set of pointers, stored in NextCharIdx array, as illustrated in Figure 5, Figure 6 and Figure 7, respectively.

[2] AoS2Input and Input2AoS, in this paper, are used as convenient names for Suffix array and Inverse Suffix Array, respectively.

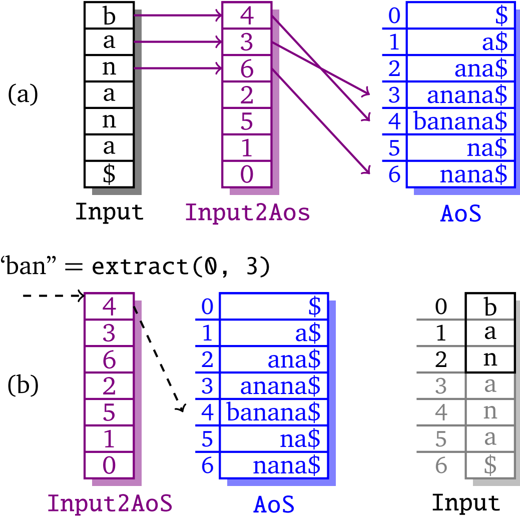



**Figure 4:** (a) The Input2AoS provides the *inverse mapping* of AoS2Input, from each index in the input to the index of the corresponding suffix in AoS (solid arrows). (b) Illustration of extract using AoS and Input2AoS (dashed arrows). The result on the original input is showed on the right to aid illustration.

The NextCharIdx array is compressed using a two-dimensional representation; see Figure 8. Specifically, the NextCharIdx values in each column of the two-dimensional representation constitute an *increasing sequence* of integers[3]. Each column can hence be independently compressed using delta encoding [2, 7, 11].

## 3.1 Succinct data representation

Succinct uses the above data representation with three main differences. We give a high-level description of these differences; see [16] for a detailed discussion.

First, Succinct uses a more space-efficient representation of AoS2Input and Input2AoS by using a sampling by "value" strategy. In particular, for sampling rate $\alpha$, rather than storing values at "indexes" $\{0, \alpha, 2\alpha, \dots\}$ as in Figure 6 and Figure 7, Succinct stores all AoS2Input values that are a multiple of $\alpha$. This allows storing each sampled value `val` as `val`$/\alpha$, leading to a more space-efficient representation. Using $\alpha = 2$ for example of Figure 6, 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.

[3]Proof: Consider two suffixes `cX<cY` in a column (indexed by character "c"). By definition, NextCharIdx values corresponding to `cX` and `cY` store AoS indexes corresponding to suffixes `X` and `Y`. Since `cX<cY` implies `X<Y` and since AoS stores suffixes in sorted order, `NextCharIdx[cX]<NextCharIdx[cY]`; hence the proof.

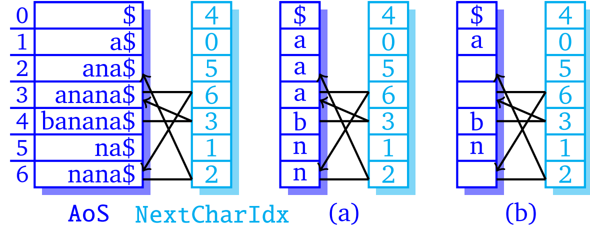



**Figure 5:** Reducing the space usage of AoS: NextCharIdx stores pointers from each suffix $S$ to the suffix $S'$ after removing the first character from $S$. (a) for each suffix in AoS, only the first character is stored. NextCharIdx pointers allow one to reconstruct suffix at any AoS index. For instance, starting from AoS[4] and following pointers, we get the original AoS entry "banana$". (b) Since suffixes are sorted, only the first AoS index at which each character occurs (*e.g.*, `{($,0),(a,1),(b,4),(n,5)}`) need be stored; a binary search can be used to locate character at any index.

Second, Succinct achieves a more space-efficient representation for NextCharIdx using the fact that values in each row of the two-dimensional representation constitute a *contiguous sequence* of integers[4]. Succinct uses its own *Skewed Wavelet Tree* data structure, based on Wavelet Trees [32, 46], to compress each row independently. Skewed Wavelet Trees allow looking up NextCharIdx value at any index without any decompression. The data structure and lookup algorithm are described in detail in [16]. These ideas allow Succinct to achieve 1.25–3× more space-efficient representation compared to existing techniques [7, 11, 31].

Finally, for semi-structured data, Succinct supports dictionary encoding along each attribute to further reduce the memory footprint. This is essentially orthogonal to Succinct's own compression; in particular, Succinct dictionary encodes the data along each attribute before constructing its own data structures.

## 3.2 Queries on compressed data

Succinct executes queries directly on the compressed representation from §3.1. We describe the query algorithm assuming access to uncompressed data structures; as discussed earlier, any value not stored in the compressed representation can be computed on the fly.

Succinct executes an `extract` query as illustrated in Figure 7 on Input2AoS representation from §3.1. A strawman algorithm for `search` would be to perform two binary searches as in Figure 3. However, this algorithm suffers from two inefficiencies. First, it executes binary searches on the entire AoS2Input array; and sec-

[4]Intuitively, any row indexed by `rowID` contains NextCharIdx values that are pointers into suffixes starting with the string `rowID`; since suffixes are sorted, these must be contiguous set of integers.

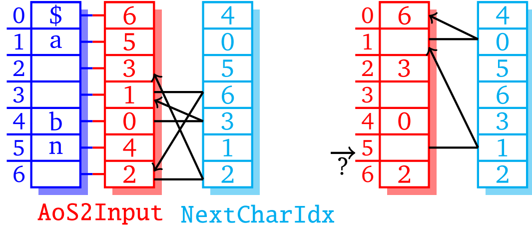



**Figure 6:** Reducing the space usage of AoS2Input. (left) Since AoS2Input stores locations of suffixes in AoS, NextCharIdx maps AoS2Input values to next larger value. That is, NextCharIdx[idx] stores the AoS2Input index that stores AoS2Input[idx]+1[5]; (right) only a few sampled values need be stored; unsampled values can be computed on the fly. For instance, starting AoS2Input[5] and following pointers twice, we get the next larger sampled value 6. Since each pointer increases value by 1, the desired value is $6-2=4$.

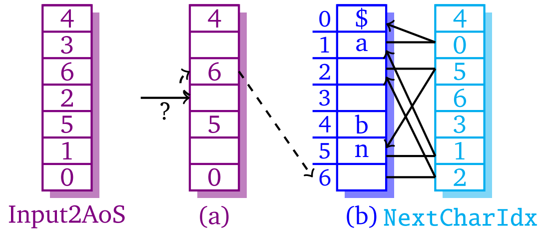



**Figure 7:** Reducing the space usage of Input2AoS. (a) only a few sampled values need be stored; (b) `extract` functionality of Figure 4 is achieved using sampled values and NextCharIdx. For instance, to execute `extract(3, 3)`, we find the next smaller sampled index (Input2AoS[2]) and corresponding suffix (AoS[2]="nana$"). We then remove the first character since the difference between the desired index and the closest sampled index was 1; hence the result "ana$".

ond, each step of the binary search requires computing the suffix at corresponding AoS index for comparison purposes. Succinct uses a query algorithm that overcomes these inefficiencies by aggressively exploiting the two-dimensional NextCharIdx representation.

Recall that the cell `(colID, rowID)` in two-dimensional NextCharIdx representation corresponds to suffixes that have `colID` as the first character and `rowID` as the following $t$ characters. Succinct uses this to perform binary search in cells rather than the entire AoS2Input array. For instance, consider the query `search("anan")`; all occurrences of string "`nan`" are contained in the cell $\langle$`n,an`$\rangle$. To find all occurrences of string `anan`, our algorithm performs a binary search only in the cell $\langle$`a,na`$\rangle$ in the next step. Intuitively, af-

[5]Proof: Let $S$ be a suffix and $S'$ be the suffix after removing first character from $S$. If $S$ starts at location `loc`, then $S'$ starts at `loc+1`. NextCharIdx stores pointers from $S$ to $S'$. Since AoS2Input stores locations of suffixes in input, NextCharIdx maps value `loc` in AoS2Input to AoS2Input index that stores the next larger value (`loc+1`).

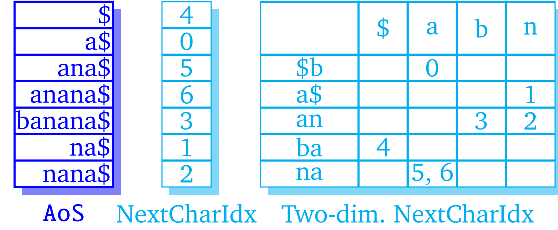



**Figure 8:** Two-dimensional NextCharIdx representation. Columns are indexed by all unique characters and rows are indexed by all unique $t$−length strings in input file, both in sorted order. A value belongs to a cell `(colID, rowID)` if corresponding suffix has `colID` as first character and `rowID` as following $t$ characters. For instance, `NextCharIdx[3]=5` and `NextCharIdx[4]=6` are contained in cell `(a, na)`, since both start with "`a`" and have "`na`" as following two characters.

ter 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 $2(m-t-1)$ binary searches, two per NextCharIdx cell [16], which is far more efficient than executing two binary searches along the entire AoS2Input array for practical values of $m$. In addition, the algorithm does not require computing any of the AoS suffixes during the binary searches. For a 16GB file, Succinct's query algorithm achieves a 2.3× speed-up on an average and 19× speed-up in the best case compared to the strawman algorithm.

**Range and Wildcard Queries.** Succinct implements `rangesearch` and `wildcardsearch` using the `search` algorithm. To implement `rangesearch(f, str1, str2)`, we find the smallest AoS index whose suffix starts with string `str1` and and the largest AoS index whose suffix starts with string `str2`. Since suffixes are sorted, the returned range of indices necessarily contain all strings that are lexicographically contained between `str1` and `str2`. To implement `wildcardsearch(f, prefix, suffix, dist)`, we first find the offsets of all prefix and suffix occurrences, and return all possible combinations such that the difference between the suffix and prefix offsets is positive and no larger than `dist` (after accounting for the prefix length).

## 4 Succinct Multi-store Design

Succinct incorporates its core techniques into a write-friendly multi-store design that chains multiple individual stores each making a different tradeoff between write, query and memory efficiency. This section describes the design and implementation of the individual stores and their synthesis to build Succinct.

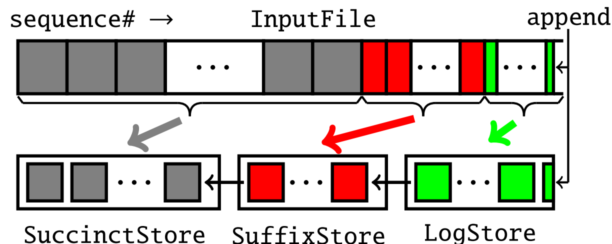



**Figure 9:** Succinct uses a write-optimized LogStore that supports fine-grained appends, a query-optimized SuffixStore that supports bulk appends, and a memory-optimized SuccinctStore. New data is appended to the end of LogStore. The entire data in LogStore and SuffixStore constitutes a single partition 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 | $\approx 0.4\times$ | $\approx 5\times$ | $\approx 9\times$ |

**Succinct design overview.** Succinct chains *three* individual stores as shown in Figure 9; Table 1 summarizes the properties of the individual stores. New data is appended into a write-optimized *LogStore*, that executes queries via in-memory data scans; the queries are further sped up using an inverted index that supports fast fine-grained updates. An intermediate store, *SuffixStore*, supports bulk appends and aggregates larger amounts of data before compression is initiated. Scans at this scale are simply inefficient. SuffixStore thus supports fast queries using *uncompressed* data structures from §3; techniques in place ensure that these data structures do not need to be updated upon bulk appends. SuffixStore raw data is periodically transformed into an immutable entropy-compressed store *SuccinctStore* that supports queries directly on the compressed representation. The average memory footprint of Succinct remains low since most of data is contained in the memory-optimized SuccinctStore.

## 4.1 LogStore

LogStore is a write-optimized store that executes data append via main memory writes, and other queries via data scans. Memory efficiency is not a goal for LogStore since it contains a small fraction of entire dataset.

One choice for LogStore design is to let cores concurrently execute read and write requests on a single shared partition and exploit parallelism by assigning each query to one of the cores. However, concurrent writes scale poorly and require complex techniques for data structure integrity [39,41,42]. Succinct uses an alternative design, partitioning LogStore data into multiple partitions, each containing a small amount of data. However, straightforward partitioning may lead to incorrect results if the query searches for a string that spans two partitionsLogStore thus uses *overlapping partitions*, each annotated with the starting and the ending offset corresponding to the data "owned" by the partition. The overlap size can be configured to expected string search length (default is 1MB). New data is always appended to the most recent partition.

LogStore executes an `extract` request by reading the data starting at the offset specified in the request. While this is fast, executing `search` via data scans can still be slow, requiring tens of milliseconds even for 250MB partition sizes. Succinct avoids scanning the entire partition using an "inverted index" per partition that supports fast updates. This index maps short length (default is three character) strings to their locations in the partition; queries then need to scan characters starting only at these locations. The index is memory inefficient, requiring roughly 8× the size of LogStore data, but has little affect on Succinct's average memory since LogStore itself contains a small fraction of the entire data. The speed-up is significant allowing Succinct to scan, in practice, up to 1GB of data within a millisecond. The index supports fast updates since, upon each write, only locations of short strings in the new data need to be appended to corresponding entries in the index.

## 4.2 SuffixStore

SuffixStore is an intermediate store between LogStore and entropy-compressed SuccinctStore that serves two goals. First, to achieve good compression, SuffixStore accumulates and queries much more data than LogStore before initiating compression. Second, to ensure that LogStore size remains small, SuffixStore supports bulk data appends without updating any existing data.

Unfortunately, LogStore approach of fast data scans with support of inverted index does not scale to data sizes in SuffixStore due to high memory footprint and

data scan latency. SuffixStore thus stores *uncompressed* AoS2Input array (§3) and executes search queries via binary search (Figure 3). SuffixStore avoids storing AoS by storing the original data that allows random access for comparison during binary search, as well as, for extract queries; these queries are fast since AoS2Input is uncompressed. SuffixStore achieves the second goal using 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. AoS2Input is constructed per partition to ensure that bulk appends do not require updating any existing data.

### 4.3 SuccinctStore

SuccinctStore is an immutable store that contains most of the data, and is thus designed for memory efficiency. SuccinctStore uses the entropy-compressed representation from §3.1 and executes queries directly on the compressed representation as described in §3.2. SuccinctStore's design had to resolve two additional challenges.

First, Succinct's memory footprint and query latency depends on multiple tunable parameters (*e.g.*, AoS2Input and Input2AoS sampling rate and string lengths for indexing NextCharIdx rows). While default parameters in SuccinctStore are chosen to operate on a sweet spot between memory and latency, Succinct will lose its advantages if input data is too large to fit in memory even after compression using default parameters. Second, LogStore being extremely small and SuffixStore being latency-optimized makes SuccinctStore a latency bottleneck. Hence, Succinct performance may deteriorate for workloads that are skewed towards particular SuccinctStore partitions.

Succinct resolves both these challenges by enabling applications to tradeoff memory for query latency. Specifically, Succinct enables applications to select AoS2Input and Input2AoS sampling rate; by storing fewer sampled values, lower memory footprint can be achieved at the cost of higher latency (and vice versa). This resolves the first challenge above by reducing the memory footprint of Succinct to avoid answering queries off-disk[6]. This also helps resolving the second challenge by increasing the memory footprint of overloaded partitions, thus disproportionately speeding up these partitions for skewed workloads.

We discuss data transformation from LogStore to SuffixStore and from SuffixStore to SuccinctStore in §5.

[6]Empirically, Succinct can achieve a memory footprint comparable to GZip. When even the GZip-compressed data does not fit in memory, the only option for any system is to answer queries off disk.

## 5 Succinct Implementation

We have implemented three Succinct prototypes along with extensions for semi-structured data (§2.1) — in Java running atop Tachyon [37], in Scala running atop Spark [54], and in C++. We discuss implementation details of the C++ prototype that uses roughly 5,200 lines of code. The high-level architecture of our Succinct prototype is shown in Figure 10. The system consists of a central coordinator and a set of storage servers, one server each for LogStore and SuffixStore, and the remaining servers for SuccinctStore. All servers share a similar architecture modulo the differences in the storage format and query execution, as described in §3.

The coordinator performs two tasks. The first task is *membership management*, which includes maintaining a list of active servers in the system by having each server send periodic heartbeats. The second task is *data management*, which includes maintaining an up-to-date collection of pointers to quickly locate the desired data during query execution. Specifically, the coordinator maintains two set of pointers: one that maps file offsets to partitions that contain the data corresponding to the offsets, and the other one that maps partitions to machines that store those partitions. As discussed in §2.1, an additional set of key → offset pointers are also maintained for supporting queries on semi-structured data.

Clients connect to one of the servers via a light-weight *Query Handler* (QH) interface; the same interface is also used by the server to connect to the coordinator and to other servers in the system. Upon receiving a query from a client, the QH parses the query and identifies whether the query needs to be forwarded to a single server (for `extract` and `append` queries) or to all the other servers (for `count` and `search` queries).

In the case of an `extract` or `append` query, QH needs to identify the server to which the query needs to be forwarded. One way to do this is to forward the query to the coordinator, which can then lookup its sets of pointers and forward the query to the appropriate server. However, this leads to the coordinator becoming a bottleneck. To avoid this, the pointers are cached at each server. Since the number of pointers scales only in the number of partitions and servers, this has minimal impact on Succinct's memory footprint. The coordinator ensures that pointer updates are immediately pushed to each of the servers. Using these pointers, an `extract` query is redirected to the QH of the appropriate machine, which then locates the appropriate partition and extracts the desired data.

In the case of a `search` query, the QH that receives the query from the client forwards the query to all the

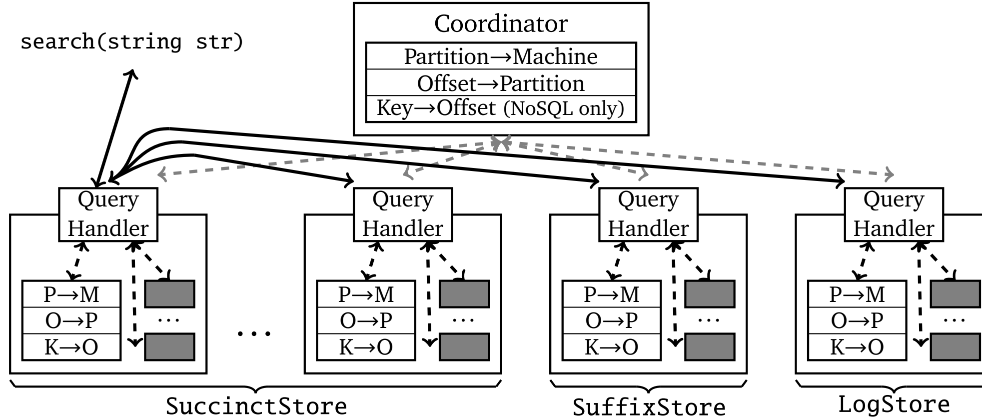



**Figure 10: Succinct system architecture**. Server and coordinator functionalities are described in §5. Each server uses a light-weight Query Handler interface to (1) interact with coordinator; (2) redirect queries to appropriate partitions and/or servers; and (3) local and global result aggregation. P→M, O→P and K→O are the same pointers as stored at the coordinator.

other QHs in the system. In turn, each QH runs multiple tasks to search all local partitions in parallel, then aggregates the results, and sends these results back to the initiator, that is, to the QH that initiated the query (see Figure 10). Finally, the initiator returns the aggregated result to the client. While redirecting queries using QHs reduces the coordinator load, QHs connecting to all other QHs may raise some scalability concerns. However, as discussed earlier, due to its efficient use of memory, Succinct requires many fewer servers than other in-memory data stores, which helps scalability.

**Data transformation between stores.** LogStore aggregates data across multiple partitions before transforming it into a single SuffixStore partition. LogStore is neither memory nor latency constrained; we expect each LogStore partition to be smaller than 250MB even for clusters of machines with 128GB RAM. Thus, AoS2Input for LogStore data can be constructed at LogStore server itself, using an efficient linear-time, linear-memory algorithm [52]. Transforming SuffixStore data into a SuccinctStore partition requires a merge sort of AoS2Input for each of the SuffixStore partitions, scanning the merged array once to construct Input2AoS and NextCharIdx, sampling AoS2Input and Input2AoS, and finally compressing each row of NextCharIdx. Succinct could use a single over-provisioned server for SuffixStore to perform this transformation at the SuffixStore server itself but currently does this in the background.

**Failure tolerance and recovery.** The current Succinct prototype requires manually handling: (1) coordinator failure; (2) data failure and recovery; and (3) adding new servers to an existing cluster. Succinct could use traditional solutions for maintaining multiple coordinator replicas with a consistent view. Data failure and recovery can be achieved using standard replication-based techniques. Finally, since each SuccinctStore contains multiple partitions, adding a new server simply requires moving some partitions from existing servers to the new server and updating pointers at servers. We leave incorporation of these techniques and evaluation of associated overheads to future work.

# 6 Evaluation

We now perform an end-to-end evaluation of Succinct's memory footprint (§6.1), throughput (§6.2) and latency (§6.3).

**Compared systems.** We evaluate Succinct using the NoSQL interface extension (§2.1), since it requires strictly more space and operations than the unstructured file interface. We compare Succinct against several open-source and industrial systems that support search queries: MongoDB [6] and Cassandra [35] using secondary indexes; HyperDex [25] using hyperspace hashing; and an industrial columnar-store DB-X, using in-memory data scans[7].

We configured each of the system for no-failure scenario. For HyperDex, we use the dimensionality as recommended in [25]. For MongoDB and Cassandra, we used the most memory-efficient indexes. These indexes do not support substring searches and wildcard

[7]For HyperDex, we encountered a previously known bug [4] that crashes the system during query execution when inter-machine latencies are highly variable. For DB-X, distributed experiments require access to the industrial version. To that end, we only perform microbenchmarks for HyperDex and DB-X for Workloads A and C.

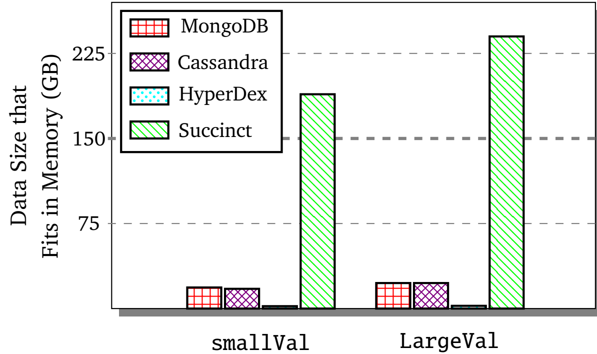



**Figure 11:** Input data size that each system fits in-memory on a distributed cluster with 150GB main memory (thick horizontal line). Succinct pushes 10-11× larger amount of data in memory compared to popular open-source data stores, while providing similar or stronger functionality.

searches. HyperDex and DB-X do not support wildcard searches. Thus, the evaluated systems provide slightly weaker functionality than Succinct. Finally, for Succinct, we disabled dictionary encoding to evaluate the performance of Succinct techniques in isolation.

**Datasets, Workloads and Cluster.** We use two multi-attribute record datasets, one `smallVal` and one `largeVal` from Conviva customers as shown in Table 2. The workloads used in our evaluation are also summarized in Table 2. Our workloads closely follow YCSB workloads; in particular, we used YCSB to generate query keys and corresponding query frequencies, which were then mapped to the queries in our datasets (for each of read, write, and search queries). All our experiments were performed on Amazon EC2 m1.xlarge machines with 15GB RAM and 4 cores, except for DB-X where we used pre-installed r2.2xlarge instances. Each of the system was warmed up for 5 minutes to maximize the amount of data cached in available memory.

## 6.1 Memory Footprint

Figure 11 shows the amount of input data (without indexes) that each system fits across a distributed cluster with 150GB main memory. Succinct supports in-memory queries on data sizes larger than the system RAM; note that Succinct results do *not* use dictionary encoding and also include pointers required for NoSQL interface extensions (§2.1, §5). MongoDB and Cassandra fit roughly 10–11× less data than Succinct due to storing secondary indexes along with the input data. HyperDex not only stores large metadata but also avoids touching multiple machines by storing a copy of the entire record with each subspace, thus fitting up to 126× less data than Succinct.

## 6.2 Throughput

We now evaluate system throughput using a distributed 10 machine Amazon EC2 cluster. Figure 12 shows throughput results for `smallVal` and `LargeVal` datasets across the four workloads from Table 2.

**Workload A.** When MongoDB and Cassandra can fit datasets in memory (17GB for `smallVal` and 23GB for `LargeVal` across a 150GB RAM cluster), Succinct's relative performance depends on record size. For small record sizes, Succinct achieves higher throughput than MongoDB and Cassandra. For MongoDB, the routing server becomes a throughput bottleneck; for Cassandra, the throughput is lower because more queries are executed off-disk. However, when record sizes are large, Succinct achieves slightly lower throughput than MongoDB due to increase in Succinct's `extract` latency.

When MongoDB and Cassandra data does not fit in memory, Succinct achieves better throughput since it performs in-memory operations while MongoDB and Cassandra have to execute some queries off-disk. Moreover, we observe that Succinct achieves consistent performance across data sizes varying from tens of GB to hundreds of GB.

**Workload B.** MongoDB and Succinct observe reduced throughput when a small fraction of queries are append queries. MongoDB throughput reduces since indexes need to be updated upon each write; for Succinct, LogStore writes become a throughput bottleneck. Cassandra being write-optimized observes minimal reduction in throughput. We observe again that, as we increase the data sizes from 17GB to 192GB (for `SmallVal`) and from 23GB to 242GB (for `LargeVal`), Succinct's throughput remains essentially unchanged.

**Workload C.** For search workloads, we expect MongoDB and Cassandra to achieve high throughput due to storing indexes. However, Cassandra requires scanning indexes for search queries leading to low throughput. The case of MongoDB is more interesting. For datasets with fewer number of attributes (`SmallVal` dataset), MongoDB achieves high throughput due to caching being more effective; for `LargeVal` dataset, MongoDB search throughput reduces significantly even when the entire index fits in memory. When MongoDB indexes do not fit in memory, Succinct achieves 13–134× higher throughput since queries are executed in-memory.

As earlier, even with 10× increase in data size (for both `smallVal` and `LargeVal`), Succinct throughput reduces minimally. As a result, Succinct's performance for large datasets is comparable to the performance of MongoDB and Cassandra for much smaller datasets.

**Table 2:** (left) Datasets used in our evaluation; (right) Workloads used in our evaluation. All workloads use a query popularity that follows a Zipf distribution with skewness 0.99, similar to YCSB [20].

| | Size (Bytes) | | #Attr- | #Records |
|---|---|---|---|---|
| | Key | Value | ibutes | (Millions) |
| `smallVal` | 8 | $\approx 140$ | 15 | 123–1393 |
| `LargeVal` | 8 | $\approx 1300$ | 98 | 19–200 |

| | Workload | Remarks |
|---|---|---|
| A | 100% Reads | YCSB workload C |
| B | 95% Reads, 5% appends | YCSB workload D |
| C | 100% Search | - |
| D | 95% Search, 5% appends | YCSB workload E |

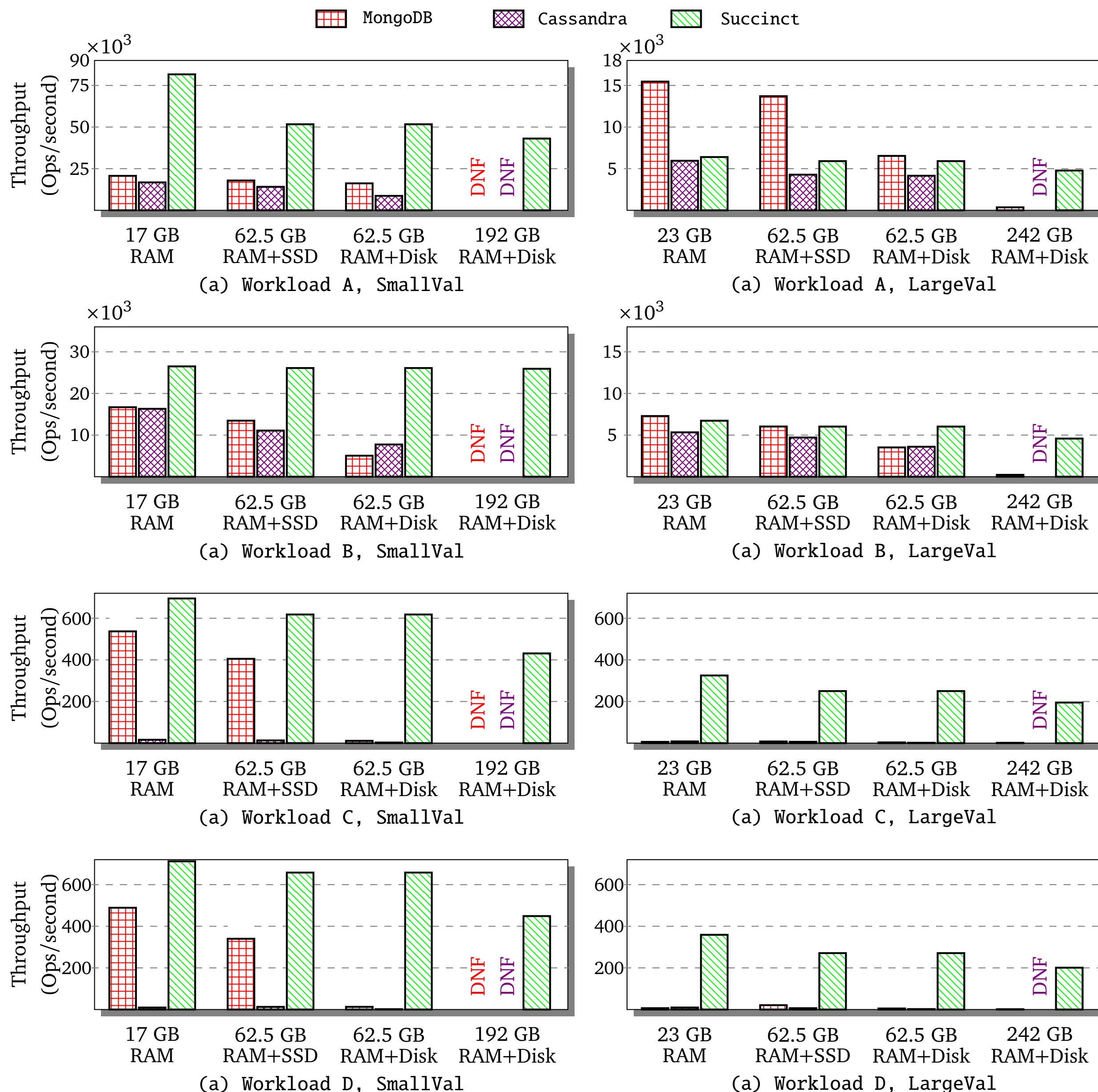



**Figure 12:** Succinct throughput against MongoDB and Cassandra for varying datasets, data sizes and workloads. MongoDB and Cassandra fit 17GB of `SmallVal` dataset and 23GB of `LargeVal` dataset in memory; Succinct fits 192GB and 242GB, respectively. `DNF` denote the experiment did not finish after 100 hours of data loading, mostly due to index construction time. Note that top four figures have different y-scales.

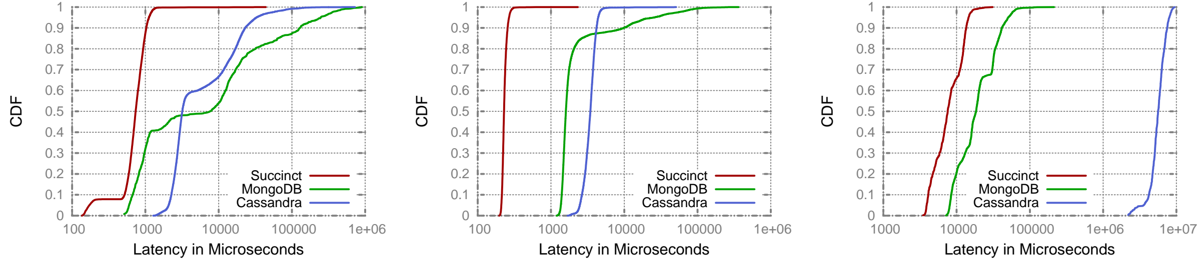



**Figure 13:** Succinct's latency for `get` (left), `put` (center) and `search` (right) against MongoDB and Cassandra for `smallVal` dataset when data and index fits in memory (best case for MongoDB and Cassandra). Discussion in §6.3.

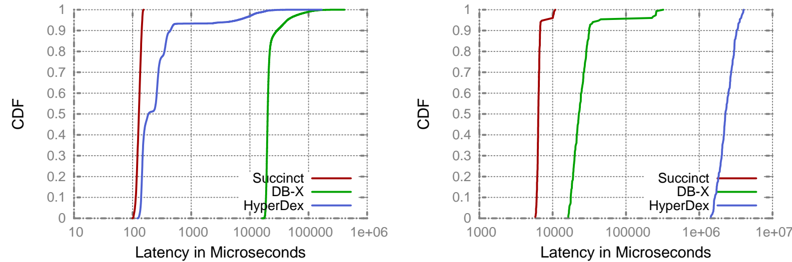



**Figure 14:** Succinct's latency for `get` (left) and `search` (right) against HyperDex and DB-X for `smallVal` 10GB dataset on a single machine. HyperDex uses subspace hashing and DB-X uses in-memory data scans for search. Discussion in §6.3.

**Workload D.** The search throughput for MongoDB and Cassandra becomes even worse as we introduce 5% appends, precisely due to the fact that indexes need to be updated upon each append. Unlike Workload B, Succinct search throughput does not reduce with appends, since writes are no more a bottleneck. As earlier, Succinct's throughput scales well with data size.

Note that the above discussion holds even when MongoDB and Cassandra use SSDs to store the data that does not fit in memory. When such is the case, throughput reduction is lower compared to the case when data is stored on disk; nevertheless, the trends remain unchanged. Specifically, Succinct is able to achieve better or comparable performance than SSD based systems for a much larger range of input values.

## 6.3 Latency

We now compare Succinct's latency against two sets of systems: (1) systems that use indexes to support queries (MongoDB and Cassandra) on a distributed 10 node Amazon EC2 cluster; and (2) systems that perform data scans along with metadata to support queries (HyperDex and DB-X) using a single-machine system. To maintain consistency across all latency experiments, we only evaluate cases where all systems (except for HyperDex) fit the entire data in memory.

**Succinct against Indexes.** Figure 13 shows that Succinct achieves comparable or better latency than MongoDB and Cassandra even when all data fits in memory. Indeed, Succinct's latency will get worse if record sizes are larger. For writes, we note that both MongoDB and Cassandra need to update indexes upon each write, leading to higher latency. For search, MongoDB achieves good latency since MongoDB performs a binary search over an in-memory index, which is similar in complexity to Succinct's search algorithm. Cassandra requires high latencies for search queries due to much less efficient utilization of available memory.

**Succinct against data scans.** Succinct's latency against systems that do not store indexes is compared in Figure 14. HyperDex achieves comparable latency for `get` queries; `search` latencies are higher since due to its high memory footprint, HyperDex is forced to answer most queries off-disk. DB-X being a columnar store is not optimized for `get` queries, thus leading to high latencies. For `search` queries, DB-X despite optimized in-memory data scans is around 10× slower at high percentiles because data scans are inherently slow.

## 6.4 Throughput versus Latency

Figure 15 shows the throughput versus latency results for Succinct, for both `get` and `search` queries

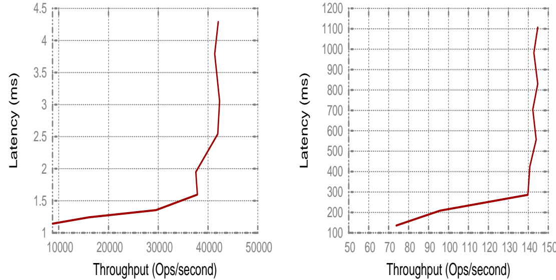



**Figure 15:** Throughput versus latency for Succinct, for `get` (left) and for `search` (right).

for a fully loaded 10 machine cluster with `smallVal` 192GB dataset. The plot shows that Succinct latency and throughput results above are for the case of a fully loaded system.

### 6.5 Sequential Throughput

Our evaluation results for workload A and B used records of sizes at most 1300bytes per query. We now discuss Succinct's performance in terms of throughput for long sequential reads. We ran a simple microbenchmark to evaluate the performance of Succinct over a single `extract` request for varying sizes of reads. Succinct achieves a constant throughput of 13Mbps using a single core single thread implementation, irrespective of the read size; the throughput increases linearly with number of threads and/or cores. This is essentially a tradeoff that Succinct makes for achieving high throughput for short reads and for search queries using a small memory footprint. For applications that require large number of sequential reads, Succinct can overcome this limitation by keeping the original uncompressed data to support sequential reads, of course at the cost of halving the amount of data that Succinct pushes into main memory. The results from Figure 11 show that Succinct will still push 5-5.5× more data than popular open-source systems with similar functionality.

## 7 Related Work

Succinct's goals are related to three key research areas:

**Queries using secondary indexes.** To support point queries, many existing data stores store indexes/metadata [3,6,25,35] in addition to the original data. While indexes achieve low latency and high throughput when they fit in memory, their performance deteriorates significantly when queries are executed off-disk. Succinct requires more than 10× lower memory than systems that store indexes, thus achieving higher throughput and lower latency for a much larger range of input sizes than systems that store indexes.

**Queries using data scans.** Point queries can also be supported using data scans. These are memory efficient but suffer from low latency and throughput for large data sizes. Most related to Succinct is this space are columnar stores [10,15,22,36,51]. The most advanced of these [10] execute queries either by scanning data or by decompressing the data on the fly (if data compressed [14]). As shown in §6, Succinct achieves better latency and throughput by avoiding expensive data scans and decompression.

**Theory techniques.** Compressed indexes has been an active area of research in theoretical computer science since late 90s [27–30,32,46–48]. Succinct adapts data structures from above works, but improves both the memory and the latency by using new techniques (§3). Succinct further resolves several challenges to realize these techniques into a practical data store: (1) efficiently handling updates using a multi-store design; (2) achieving better scalability by carefully exploiting parallelism within and across machines; and (3) enabling queries on semi-structured data by encoding the structure within a flat file.

## 8 Conclusion

In this paper, we have presented Succinct, a distributed data store that supports a wide range of queries while operating at a new point in the design space between data scans (memory-efficient, but high latency and low throughput) and indexes (memory-inefficient, low latency, high throughput). Succinct achieves memory footprint close to that of data scans by storing the input data in an entropy-compressed representation that supports random access, as well as a wide range of analytical queries. When indexes fit in memory, Succinct achieves comparable latency, but lower throughput. However, due to its low memory footprint, Succinct is able to store more data in memory, avoiding latency and throughput reduction due to off-disk or off-SSD query execution for a much larger range of input sizes than systems that use indexes.

## Acknowledgments

This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, C3Energy, Cisco, Cray, Cloudera, EMC, Ericsson, Facebook, Guavus, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata and VMware.

## References


[1] CouchDB. http://couchdb.apache.org.

[2] Delta Encoding. http://en.wikipedia.org/wiki/Delta_encoding.

[3] Elasticsearch. http://www.elasticsearch.org.

[4] Hyperdex Bug. https://groups.google.com/forum/#!msg/hyperdex-discuss/PUIpjMPEiAI/I3ZImpU7OtkJ.

[5] MemCached. http://www.memcached.org.

[6] MongoDB. http://www.mongodb.org.

[7] Pizza&Chili Corpus: Compressed Indexes and their Testbeds. http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array/.

[8] Presto. http://prestodb.io.

[9] Redis. http://www.redis.io.

[10] SAP HANA. http://www.saphana.com/.

[11] SDSL. https://github.com/simongog/sdsl-lite.

[12] Suffix Array. http://en.wikipedia.org/wiki/Suffix_array.

[13] Suffix Tree. http://en.wikipedia.org/wiki/Suffix_tree.

[14] Vertica Does Not Compute on Compressed Data. http://tinyurl.com/l36w8xs.

[15] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*, 2006.

[16] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. In *Technical Report, UC Berkeley*, 2014.

[17] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64, 2012.

[18] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-distributed Database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[22] Daniel J. Abadi and Samuel R. Madden and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *ACM International Conference on Management of Data (SIGMOD)*, 2008.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[24] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[25] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, 2012.

[26] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[27] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2000.

[28] P. Ferragina and G. Manzini. An Experimental Study of a Compressed Index. *Information Sciences*, 135(1):13–28, 2001.

[29] P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[30] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[31] R. Grossi, A. Gupta, and J. S. Vitter. High-order Entropy-compressed Text Indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003.

[32] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[33] W.-K. Hon, T. W. Lam, W.-K. Sung, W.-L. Tse, C.-K. Wong, and S.-M. Yiu. Practical aspects of Compressed Suffix Arrays and FM-Index in Searching DNA Sequences. In *Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALC)*, 2004.

[34] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[35] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[36] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[37] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[38] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[39] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast Inmemory Key-value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[40] U. Manber and G. Myers. Suffix Arrays: A New Method for On-line String Searches. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

[41] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[42] M. M. Michael. High Performance Dynamic Lockfree Hash Tables and List-based Sets. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[43] G. Navarro and E. Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Experimental Algorithms*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. 2012.

[44] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[45] Rodrigo GonzÃąlez and Szymon Grabowski and Veli MÃď'kinen and Gonzalo Navarro. Practical implementation of rank and select queries. In *Workshop on Efficient and Experimental Algorithms (WEA)*, 2005.

[46] K. Sadakane. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array. In *International Conference on Algorithms and Computation (ISAAC)*. 2000.

[47] K. Sadakane. Succinct Representations of Lcp Information and Improvements in the Compressed Suffix Arrays. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002.

[48] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[49] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[50] S. Sivasubramanian. Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service. In *ACM International Conference on Management of Data (SIGMOD)*, 2012.

[51] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *International Conference on Very Large Data Bases (VLDB)*, 2005.

[52] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

[53] S. Vigna. Broadword Implementation of Rank/Select Queries. In *Workshop on Efficient and Experimental Algorithms (WEA)*, 2008.

[54] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[55] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. 2013.

# A Data Structures

We describe the four arrays used in Succinct, show how we achieve a *compressed* representation for each of the arrays and how these arrays are used for performing queries directly on a compressed representation of the data.

`AoS`. `AoS` stores the set of suffixes in a file in lexicographically sorted order, with `AoS[i]` storing the i$^{\text{th}}$ lexicographically smallest suffix.

`AoS2Input`. `AoS2Input` maps the suffixes in `AoS` to corresponding locations in the input. That is, `AoS2Input[i]` stores the location of `AoS[i]` in the input file.

`Input2AoS`. `Input2AoS` maps locations in the input file to indexes into `AoS2Input` that stores these locations. That is, `Input2AoS[loc]` stores an index `idx` such that `AoS2Input[idx] = loc`.

`NextCharIdx`. `NextCharIdx[i]` stores the index into `AoS2Input` that stores `AoS2Input[i]+1`.

## A.1 Compression

Suppose input file contains $n$ `ASCII` characters, and is of size $8n$ bits. Since there are $n$ suffixes varying from length 1 to $n$, `AoS` stores $0.5n(n+1)$ characters requiring $4n(n+1)$ bits. Each of `AoS2Input`, `Input2AoS` and `NextCharIdx` require $\lceil \log n \rceil$ bits for each entry, leading to a total space of $4n(n+1)+3n\lceil \log n \rceil$ bits. These arrays contain a lot of redundancy since the size of input file is just $8n$.

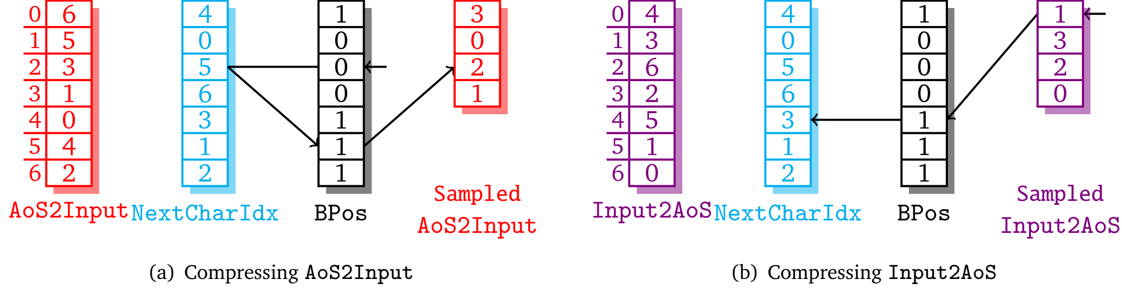
### A.1.1 Compressing `AoS2Input` and `Input2AoS`

In this subsection, we describe the details pertaining to the compression of the `AoS2Input` and `Input2AoS` arrays.

**Sampling.** Succinct employs *sampling* to reduce the storage footprint of the `AoS2Input` and `Input2AoS` arrays. Various sampling techniques have been used in theory [31, 32, 46–48]. Our implementation samples and stores all `AoS2Input` values that are a multiple of a configurable integer parameter $\alpha$, with default being $\lceil \log n \rceil$. Each sampled value `val` is stored as `val`$/\alpha$ in an array called the `SampledAoS2Input`, leading to a more space-efficient representation. Additionally, we store a bitmap `BPos` that marks the positions in `AoS2Input` where the values are sampled. Since `AoS2Input` stores locations of suffixes in input file, `SampledAoS2Input` values correspond to sampled locations. Succinct's compressed version of `Input2AoS` maps the sampled locations in the input file to indexes into `SampledAoS2Input` that store these locations; these indexes are stored in the `SampledInput2AoS` array. Observe that the `SampledInput2AoS` is simply the *inverse mapping* of the `SampledAoS2Input` array. Figure 17 shows an example of sampling for the `AoS2Input` and `Input2AoS` arrays.

**Rank and Select Data Structures.** In order to efficiently translate an index into a sampled array to the corresponding index into the unsampled array (and vice versa), we store *rank* and *select* data structures for the `BPos` bitmap. The rank data structure takes an index `idx` as an input and returns the number of 1's before `idx` in the bitmap, whereas the select data structure takes integer `i` as an input and returns the location of $(i+1)^{th}$ set bit in the bitmap (see Figure 18). Rank and Select are one of the most well-researched data structures [43, 45, 53, 55]. We implemented several of the known algorithms and found the one in [55] to perform the best in practice; we use this implementation in Succinct. In order further reduce storage overheads, we compress the bitmap itself using entropy compression.

Figure 16 illustrates how lookups can be performed on compressed versions of `AoS2Input` and `Input2AoS`




(a) Compressing AoS2Input

(b) Compressing Input2AoS

**Figure 16:** Lookups on AoS2Input and Input2AoS. (a) To find AoS2Input[2], note that BPos[2]=0; following NextCharIdx[2]=5, find that BPos[5]=1, i.e., the value is sampled. rank(5)=2 gives us the corresponding index into the SampledAoS2Input. Multiplying SampledAoS2Input[2]=2 with $\alpha = 2$ and subtracting the number of NextCharIdx hops gives us AoS2Input[2]=2×2-1=3. (b) To lookup Input2AoS[1], we find the smallest multiple of $\alpha = 2$ less than loc=1, i.e., loc1=0. SampledAoS2Input[0] gives us the SampledAoS2Input index corresponding to loc1; we translate the SampledAoS2Input index to the corresponding AoS2Input index from select(1)=4. Finally, we follow loc-loc1=1 NextCharIdx pointers to get the required value Input2AoS[1]=3.

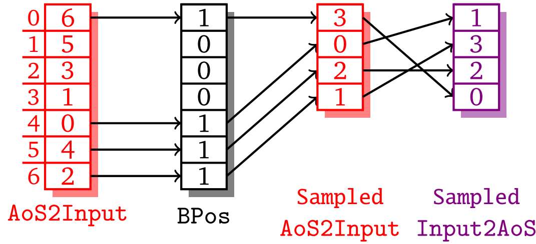



**Figure 17: Sampling AoS2Input and Input2AoS:** We sample AoS values that are multiples of $\alpha$ ($\alpha = 2$ in this example) and mark the positions in BPos. Note that we store each sampled value val as val/$\alpha$ in SampledAoS2Input. SampledInput2AoS maps the sampled locations in the input file to indexes into SampledAoS2Input that store these locations.

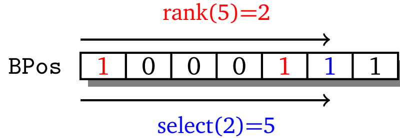



**Figure 18: Rank and Select Data Structures:** rank(i) counts the number of 1's before i, while select(i) gives the position of the $(i+1)^{th}$ 1 in the bitmap.

through examples. We store only the sampled versions of each of the two arrays, along with the BPos bitmap and associated rank/select data structures. The NextCharIdx array along with BPos enables looking up unsampled values in the two sampled arrays.

**Looking up AoS2Input values.** Looking up values in AoS2Input (Algorithm 1) can be broken down into three steps:

1. Follow NextCharIdx pointers until we reach an index idx1 in AoS2Input where the value is sampled; count the #hops it took to get there.
2. Translate the index into AoS2Input to the corresponding index into SampledAoS2Input by consulting the rank data structure (i.e., rank(idx1)).
3. Multiply the corresponding SampledAoS2Input value with $\alpha$ and subtract the #hops to get the AoS2Input value.

To see why Step 1 works, note that NextCharIdx[2] in Figure 16(a) tells us where AoS2Input[2]+1=4 is stored; or, NextCharIdx[i] tells us where AoS2Input[i]+1 is stored in AoS2Input. Hence, to find an unsampled AoS2Input value, we check if AoS2Input at NextCharIdx[i] is sampled by consulting the BPos array. If yes, we know the value AoS2Input[i]+1 and hence, AoS2Input[i]. This process can be repeated until we find a sampled value.

Note that the SampledAoS2Input values must be multiplied by $\alpha$ (Step 3) to get back the corresponding AoS2Input value. Once we obtain the sampled value, we simply subtract the number of additional hops we took to reach the sampled value to get the required AoS2Input value. See Figure 16(a) for an example.

**Looking up Input2AoS values.** Similar to lookups on AoS2Input, lookups on Input2AoS can also be viewed as three steps:

1. Find the largest multiple of $\alpha$ loc1 smaller than or equal to the location loc being looked up; obtain the SampledInput2AoS value idx1 at loc1.
2. Translate this index into SampledAoS2Input to the corresponding index into AoS2Input

**Algorithm 1** Algorithm for `lookupAoS2Input(i)`.

```
1: #hops ← 0
2: idx1 ← i
3: While !BPos[idx1]
4:     idx1 ← lookupNextCharIdx[idx1]
5:     #hops += 1
6: #BitsSet ← rank(BPos, idx1)
7: Val ← SampledAoS2Input[#BitsSet]
8: Return Val ×α - #hops
```

by consulting the select data structure (i.e., `select(idx1)`.

3. Follow (`loc-loc1`) `NextCharIdx` pointers to get the required `Input2AoS` value.

Since `Input2AoS` and `SampledInput2AoS` are inverse mappings of `AoS2Input` and `SampledAoS2Input` respectively, observe that the algorithm for looking up `Input2AoS` values is the *inverse* of the `AoS2Input` lookup algorithm.

Step 1 exploits the fact that given a location `loc` in the file, we know that the largest multiple of $\alpha$ that is smaller than or equal to `loc` (say, `loc1`) must be an `AoS2Input` value that is sampled. The corresponding `SampledInput2AoS` value would give us the `SampledAoS2Input` index for the sampled value.

Once we obtain the corresponding index into `AoS2Input` (Step 2), we observe that we would have to follow (`loc-loc1`) `NextCharIdx` pointers to get from the sampled `AoS2Input` value to the location which we started with (i.e., our query for the `Input2AoS` lookup algorithm). Therefore, it suffices to follow as many `NextCharIdx` pointer to get the corresponding `AoS2Input` index (Step 3), which would give us the required `Input2AoS` value. Figure 16(b) illustrates the lookup algorithm with an example.

**Algorithm 2** Algorithm for `lookupAoS2Input(loc)`.

```
1: loc1 ← α × ⌊i/α⌋
2: idx1 ← lookupInput2AoS[⌊i/α⌋]
3: idx ← select(BPos, idx1)
4: For i=1 to loc-loc1
5:     idx ← lookupNextCharIdx[idx]
6: Return idx
```

### A.1.2 Compressing AoS

The largest of the four arrays is `AoS`. The redundancy in `AoS` is best understood by observing that `AoS[4]` and `AoS[3]` in Figure **??** overlap at "anana$". Indeed, the second character of `AoS[4]` is the first character of `AoS[3]`. Observe that interestingly, the value of `NextCharIdx[4]` is 3. This is not a coincidence; intuitively, since `AoS2Input` stores locations of suffixes in the input, `AoS2Input[4]` and `AoS2Input[4]+1` are the locations of the first and the second characters of `AoS[4]`. Since `NextCharIdx[4]` tells us where `AoS2Input[4]+1` is stored into `AoS2Input`, the first character of `AoS[3]=AoS[NextCharIdx[4]]` is the second character of `AoS[4]`. It follows that we only need to store the first character for each index of `AoS`; the remaining characters can be computed on the fly using `NextCharIdx`. In fact, we can do even better — since `AoS` stores suffixes in sorted order, all we need is each unique character in the input file and the first index into `AoS` at which the suffix starts with the character. This reduces the size of `AoS` from $4n(n+1)$ bits to $512\lceil\log n\rceil$ bits.

Succinct representation of `AoS` stores: (a) all unique characters in the input file in sorted order; and (b) for each character, first `AoS` index with suffix starting with that character.

**Looking up `AoS` values.** Let `characters` be the sorted array of unique characters, and `char-indexes` be the corresponding array of indexes. Observe that binary searching for an index `idx` on `char-indexes` yields the index of the *first* suffix that shares the same starting character as `AoS [i]`; consulting the `characters` array gives us the first character. As described above, the first character of `AoS[NextCharIdx[idx]]` is the second character of `AoS[idx]`. Therefore, to get the second character of `AoS[idx]`, we simply determine `idx1 = NextCharIdx[idx]` and determine the first character of `AoS[idx1]`. If we repeat this process `len` times, we obtain the first `len` bytes of `AoS[idx]`. Algorithm 3 summarizes this process.

**Algorithm 3** Algorithm for lookupAoS(idx, len).

```
1: str ← NULL
2: idx1 ← idx
3: If i=0 to len-1
4:     idx2 = BinSearch(char-indexes, idx1)
5:     str += characters[idx2]
6:     idx1 ← lookupNPA[idx1]
7: Return str
```

### A.1.3 Compressing NextCharIdx

To compress `NextCharIdx`, Succinct exploits two properties identified in theory literature [31,32,46–48]. These properties are based on a two-dimensional representation of `NextCharIdx`, where columns are indexed by all unique characters and rows are indexed by all unique $t$—length strings, both in sorted order. The value `NextCharIdx[idx]` belongs to cell `(rowID,`

columnID) if the corresponding suffix at AoS[idx] starts with character columnID, followed by string rowID.

**Property1: NextCharIdx values in any column form an *increasing* sequence of integers.**

**Property2: NextCharIdx values in any row form a *contiguous* sequence of integers.**

**Skewed Wavelet Trees** We use the same two-dimensional representation of uncompressed NextCharIdx with default value of $t$ set to 3. However, our compressed representation differs from previous ones in terms of a number of optimizations. In particular, the NextCharIdx compression techniques in [31,32,46–48] were focused on providing theoretical guarantees; our implementation uses subtle changes to trade-off theoretical guarantees in a few corner cases to achieve lower memory footprint and query latency.

Recall that the NextCharIdx representation consists of as many rows as the number of unique $t$—length substrings in input file. Each row contains a contiguous (not necessarily sorted) sequence of integers (see Figure 19, top row) distributed across multiple cells. To start with, given an index idx, it is easy to find the row, the cell and the offset into the cell where NextCharIdx[idx] is stored by storing a few small arrays.

Succinct compresses each row independently in a manner that allows looking up the NextCharIdx value at a given offset into a cell. To achieve this, we construct a binary tree over the cells (the leaves of the tree correspond to cells in the row), partitioning the cells at each level such that the number of values in the cells assigned to the left children and to the right children are as close as possible. We use simple heuristics to identify cells that contain very few values (cells 1,2 and 7 in top row each contain a single value) and store these separately; all values in separated cells are put in a dummy cell (last cell in second row of Figure 19). By separating sparse cells and by partitioning cells based on number of values, we attempt to reduce the height of the tree and to ensure that cells with a large number of values have lower depth. These lead to significant reduction in memory footprint *and* latency, at the cost of theoretical guarantees in some corner cases.

Next we construct a bit array for each node of the binary tree as follows. Let S be the set of values contained in the cells being partitioned at the root. We create an array of |S| bits; the $i^{th}$ bit of this array is set if the $i^{th}$ largest value in S is contained in a cell assigned to the right child, and unset otherwise. See Figure 19 and consider the left child of the root of the tree. The root is assigned the first three cells and $S = \{3,5,7,15,11,8,9,2,4\}$; the sorted version of S is $\{2,3,4,5,7,8,9,11,15\}$ with values assigned to the right child being $\{2,4,8,9,11\}$, precisely the set bits in the bit array stored at that root.

In addition to the values in separated cells, Succinct stores (a) cell identifiers at which partitioning occurs at each node in the binary tree (circled values in Figure 19) and (b) a compressed representation of the bit arrays at each node for each level of the tree, together with *rank* and *select* data structures.

Now suppose we want to locate the NextCharIdx value at the second offset in the second cell, which is equal to 9. We first locate the cell by traversing down the tree, comparing the identifier stored at the node to the identifier of the cell being located until we hit a leaf. Once the cell is located in the tree, we start traversing up the tree. At each level, we check if the current node is the left child or the right child of the parent. If it is the right child, we update offset to be the index into parent's bit array where the offset-th 1 (and 0 if it is the left child) lies. Note that these translate into select operations on the bitmap corresponding to the node; we represent searching for the position of the $i^{th}$ 1 as select1, and the $i^{th}$ 0 as select0. Intuitively, offset maintains the order of the desired NextCharIdx value among (the sorted) set of values at the binary tree node. **Since the root of the tree contains the set of contiguous integers (Property 2)**, the final value of offset gives us the desired integer. For our example, offset = 2 at the leaf. Since the second cell is the left child of the parent, we find the location of second '0' in parent's bit array and set offset = 4. At the next level, we find the location of fourth '1' in parent's bit array since the current node is the right child and set offset = 7. At the root, we find the location of the seventh '0', giving us offset = 9 as desired.

# B Query Algorithms

We described how lookups can be performed on the compressed representations of AoS, AoS2Input and Input2AoS using lookups on the NextCharIdx array in §A.1.1); we now describe how we can perform search, count and extract using the compressed arrays.

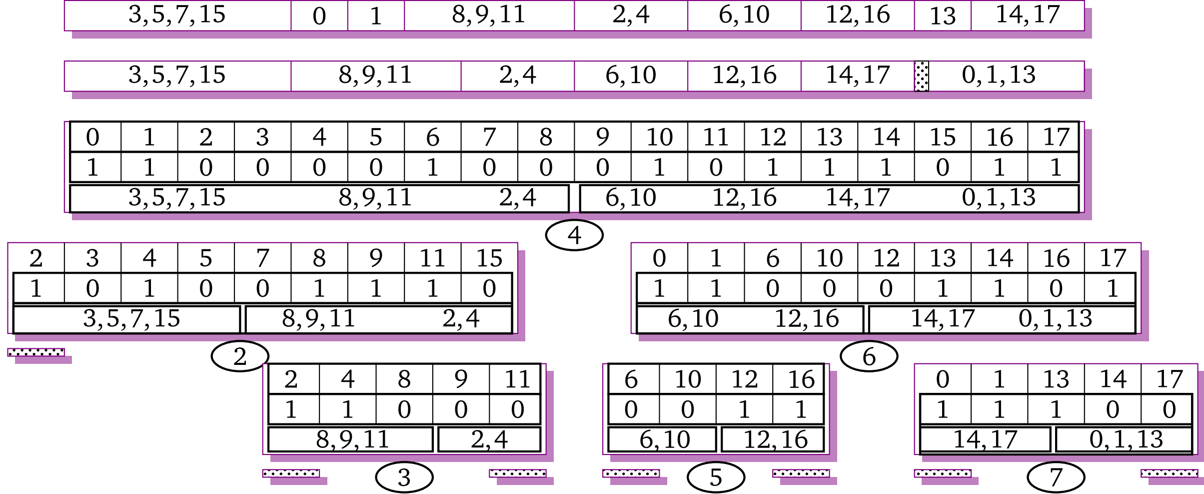



**Figure 19:** Example for constructing the tree for each row of two-dimensional representation of `NextCharIdx`.

## B.1 Random access in Succinct

The `extract` operation forms the basis of random access in Succinct. In order to extract len bytes starting at offset off in the *uncompressed* input file, we first lookup `Input2AoS[off]` to obtain the index `idx` where `AoS2Input[idx]=off`. Once we have this index, we simply need to obtain the first `len` bytes of the suffix at `AoS[idx]`, as described in Algorithm 3.

**Algorithm 4** Algorithm for `extract(off, len)`.

```
1: idx ← lookupInput2AoS(off)
2: str ← lookupAoS (idx, len)
3: Return str
```

## B.2 Counting and Searching

It is simple to see that two binary searches over the `AoS` array for an input string would give us the first and the last suffix that start with a given input string. If the indexes of these suffixes be `idx1` and `idx2` respectively, then the `count` of the string occurrences is simply `(idx2-idx1+1)`, and the `AoS2Input` values `{AoS2Input[idx1], ..., AoS2Input[idx2]}` would give us the locations of each of these occurrences (i.e., the `search` results).

Succinct, however, exploits the two dimensional representation of `NextCharIdx` and the information contained within it obtain the two indexes `idx1` and `idx2` more efficiently. Consider the input string "banana"; recall from §A.1.3 that the `cell=`⟨`n, ana`⟩ contains all `NextCharIdx[idx]` values for which the first character of the suffix at `AoS [idx]` is "`a`" and the following three characters are "`ana`".

Our algorithm aggressively exploits the above interpretation. In particular, the algorithm first finds indexes `idx1` and `idx2` for `NextCharIdx` values that belong to `cell` = ⟨n, ana⟩. In the next step, the algorithm looks in `cell` = ⟨a, nan⟩ and performs a binary search over the `NextCharIdx` values in the cell (they are sorted, not necessarily contiguous) and finds indexes `idx01` and `idx02` such that `NextCharIdx[idx01] = idx1` and `NextCharIdx[idx02] = idx2`. Intuitively, after this round, `idx01` and tt idx02 are indexes into `AoS` for which suffixes start with a and are followed by `nana`. The final binary search is done in `cell` = ⟨b, ana⟩ and the results are the desired indexes. The algorithm for finding `idx1` and `idx2` is depicted in Algorithm 5, while `count` and `search` are summarized in Algorithms 6 and 7.

**Algorithm 5** Algorithm for `findRange(str)`.

```
1: len ← length(str)
2: cell ← ⟨str[len-t-1], str[(len-t)...(len-1)]⟩
3: (idx1, idx2) ← (firstIdx(cell), lastIdx(cell))
4: For i=len-1 to 0
5:     cell ← ⟨str[i-t-1], str[(i-t)...(i-1)]⟩
6:     idx01 ← BinSearch(cell, idx1)
7:     idx02 ← BinSearch(cell, idx2)
8:     (idx1, idx2) ← (idx01, idx02)
9: Return (idx1, idx2)
```

**Algorithm 6** Algorithm for `count(str)`.

```
1: (idx1, idx2) ← findRange(str)
2: cnt ← idx2 - idx1 + 1
3: Return cnt
```

**Algorithm 7** Algorithm for `search(str)`.

```
1: res ← {}
2: (idx1, idx2) ← findRange(str)
3: For i=idx1 to idx2
4:     loc ← lookupAoS2Input[i]
5:     Insert loc in res
6: Return res
```

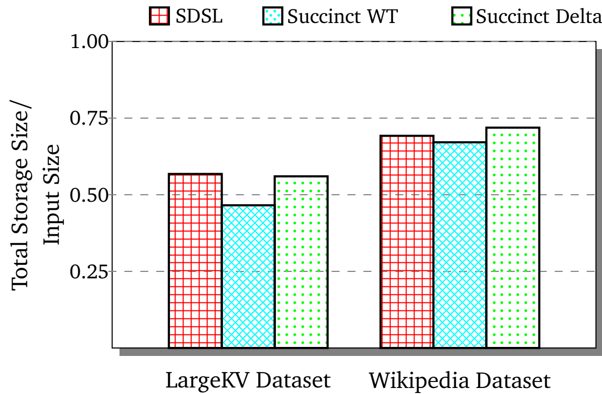



**Figure 20:** Storage footprint for the compared implementations.

# C Comparison against SDSL

We compare the storage footprints and performance of Succinct's `search` operation against SDSL [?]'s implementation of compressed suffix arrays. We consider two implementations for Succinct's `NextCharIdx` array — the first encoded using *Skewed Wavelet Trees* as described in §A.1.3, and the other encoded using sampling with delta encoding, with the delta gaps encoded using the Elias Gamma scheme [?].

**Datasets and Workloads.** We use two datasets: the Wikipedia dataset composed of Wikipedia articles, and the Conviva dataset containing information pertaining to video streams viewed by internet users. We fix the dataset size to 8*GB* for both datasets. The query workload for our evaluation is composed of random 8 byte strings sampled from both datasets. These strings are grouped according to their number of occurrences in the respective dataset, and we report the average `search` latency measurements for each *query bucket* or group.

**Results.** Figure 20 shows the comparison of storage footprints for the different implementations. Succinct's storage optimized skewed wavelet tree implementation achieves 4−20% better compression when compared to SDSL, while the implementation using delta encoded `NextCharIdx` achieves a footprint comparable to or better than the SDSL implementation.

We study the variation of search latency with the number of occurrences of the searched term in Figure 21. While Succinct's `search` latency is 1.4−2.3× lower for the skewed wavelet tree implementation, the implementation employing delta encoding achieves 3−5.6× faster `search` queries when compared to the SDSL implementation, for both datasets.

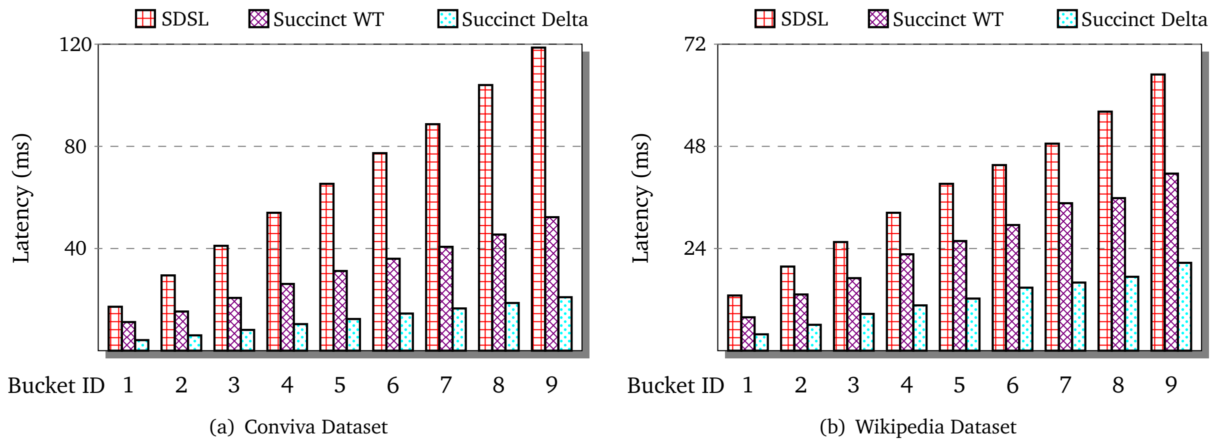



**Figure 21:** Variation of search latency with the number of occurrences; the different buckets IDs correspond to different number of occurrences of the searched term. Bucket ID 1 corresponds to queries with 1000−2000 occurrences, bucket ID 2 corresponds to queries with 2000−3000 occurrences, and so on.