A Reproduction of Succinct Data Store System

Peter Spradling
Stanford University
Stanford, CA
psprads@stanford.edu

Jerry Zhilin Jiang
Stanford University
Stanford, CA
zjiang23@stanford.edu

ABSTRACT

This is a reproduction of Succinct [2], a distributed in-memory data store that utilizes compression to fit more data into memory. Succinct’s compression scheme enables fast queries directly on top of the compressed representation. According to the original authors, Succinct exposes “a minimal, but powerful API,” while requiring much less memory than traditional databases. Below, we attempt to reproduce some of the original paper’s findings and evaluate certain claims. Specifically, we attempt to reproduce the paper’s comparison of Succinct and MongoDB in terms of memory usage, throughput, and latency.

The original paper found that MongoDB’s in memory indices took up 8x more memory than the actual data, whereas Succinct’s total in memory representation was actually smaller than the raw data, leading to dramatically higher memory usage for MongoDB given the same amount of input data. The original paper also found that Succinct displayed measurably better throughput and latency with respect to MongoDB on a workload of 10,000 search requests.

Our reproduction contains mixed results. We were able to reproduce similar Succinct throughput and latency metrics, but due to complications with measuring memory usage in Spark and compiler issues with the C++ version of Succinct, we were not able to obtain a post-compression memory usage metric that we feel confident is correct. Of all our attempts to measure memory usage, our most promising method yielded a metric 28% larger than the original data, which would contradict many of the paper’s results, but as we discuss below, we believe the issue is likely with our measurement.

We also could not verify the MongoDB memory usage metrics. We tested both MongoDB 2.7 and MongoDB 4.0, and like the original paper, we created indexes on every column in the smallkv dataset. Our results show a 4:1 and 3:1 ratio of metadata to data in MongoDB 2.7 and MongoDB 4.0, not the 8:1 ratio that the original authors mentioned.

1 INTRODUCTION

Succinct is a new (2015) in memory distributed data store that keeps data compressed in order to fit more data into memory. Succinct uses a compression scheme that stores data in such a way that you can achieve the benefits of having indexes on your data, while using less data than a simple raw data storage format. Indexes, while very useful for querying, can take up a lot of memory. The more memory taken by indexes, the less memory available to the actual data, making it likely that storing data on disk will be necessary. According to the original Succinct paper, databases like MongoDB rely heavily on indexes, resulting in an 8:1 ratio of index memory to data memory. Our results are shown in Section 4.1.

Succinct’s compression format allows for the following actions: append, delete, extract, count, search, rangesearch, and wildcardsearch. This API is simultaneously incredibly powerful and somewhat restrictive.

To achieve fast query times for the API shown above at a memory cost well below other databases, Succinct makes a number of trade-offs. First, the upfront resource cost of converting data into the Succinct format is extensive in terms of both CPU and memory (see Section 4.2). Also, as mentioned previously, Succinct does not support in-place updates, meaning it is not a general purpose solution. Finally, according to the original authors, Succinct should not be used if the entire data set cannot fit in memory. Once compressed, Succinct’s format makes it easier to fit a large amount of data in memory, but if you cannot afford to scale horizontally to a large enough RAM cluster, Succinct is not recommended.

In this paper, we attempt to reproduce the original paper’s comparison of Succinct to MongoDB. We specifically attempt to reproduce the memory usage comparison metrics, the query latency results, and the search throughput (Workload C) results. We also measure the computational cost to compress data from its raw format to its final fully compressed format and compare that upfront cost to the cost of creating indexes. We end with thoughts on the limitations of Succinct and evaluate its generality.

2 EXPERIMENT SETUP

We created three clusters, one running MongoDB 2.7.2, one running MongoDB 4.0.9, and one running Spark on top of HDFS. The original paper used MongoDB 2.6.4, but since none of Google Cloud Platform’s default OS images continue to support MongoDB 2.6.4, we decided to use the oldest

1Note that in-place updating is missing. The authors mentioned that not supporting in-place updates is potentially a fundamental limitation of Succinct’s data format. Updates can be achieved by deleting and appending.
supported version of MongoDB. MongoDB 3.0.X introduced significant changes to how indexes are stored in memory with the introduction of WiredTiger, so we felt it necessary to compare to a pre-3.0.X version as well as modern MongoDB.

The original paper ran the C++ version of Succinct, but we could not compile the latest master build of the C++ codebase, even after the authors pushed a new build. We opted to use Succinct Spark instead, and despite the differences in the testing environments, our throughput and latency results matched the original paper.

All three clusters were sharded across 10 worker nodes and a client/master node. Each MongoDB node is a n1-standard-4 GCP instance (4 vCPUs, 15 GB memory), with 200GB of hard disk space for sharding/worker VMs, 500GB of hard disk space for client/routing/master VMs, running on the Ubuntu 14.04 operating system. Both MongoDB clusters used hash sharding, which we observed had balanced shard distribution.

We deployed Succinct on a cluster pre-configured by GCP’s Dataproc service, running Spark 2.3.3 and Hadoop 2.9.2 on the Debian 9 operating system. We initially chose GCP’s n1-standard-4 instance type as described above, but would later switch to the n1-highmem-4 (4 vCPUs, 26 GB memory) instance type in order to handle out-of-memory errors during Succinct’s compression stage.

Our reproduction implementation and instructions are made available at: https://github.com/PBS590/CS244proj3.

3 RELATED WORK

As the original authors mention, there has been prior work on compressed suffix arrays and FM-indices in order to allow for fast search with little memory overhead [2]. There are also many column stores that keep data compressed in memory, query on the compressed format if possible, and decompress when the compressed format cannot answer the query in question [1].

Succinct borrows compression techniques from previous papers and extends them using techniques like sampling, delta encoding, and dictionary encoding to reduce memory usage. Succinct also provides a three tiered data store implementation that exposes the Succinct API.

4 RESULTS

4.1 Memory Usage

We measured the memory and storage usage of MongoDB 2.7.2 vs. MongoDB 4.0.9 vs. Succinct by loading 5GB of the SmallKV dataset into the cluster and evaluate the total database size (including any indexes) after the load operation. For MongoDB instances, we measured the total database size of the smallkv collection using the db.smallkv.totalSize() command which takes both data and indexes into consideration.

One important factor in measuring MongoDB’s memory usage is the number of indexes we create. Succinct is designed to support queries on all columns without the need to build a single index, and therefore, the original paper [2] justified creating indexes for every single column in MongoDB to have the same functional capability as Succinct. This assumption might be too punitive toward MongoDB, given that indexes are usually created based on query load. In a production system, depending on the query workload, it might not be necessary to create an index on every column, or even most columns in the database. The SmallKV dataset has 16 columns (fields), and depending on the number of created indexes, the database’s total size can vary substantially (especially in MongoDB 4.0.9). Given that our throughput measurements search on only a single column, we decided...
to measure MongoDB’s memory usage both after creating one index on the searched column and after adding indexes to every single column.

Figure 1 shows memory usage measurement from the original paper (proportionally adapted to our experiment case of 5GB) and from our reproduction measurements.

### 4.1.1 MongoDB v2.7.2

The 2.7.2 version of MongoDB loads data significantly more slowly as compared to the newer version (it took approximately 10 hours to load a 5.0GB chunk of csv data using `mongoimport`), and therefore we decided to load only one chunk of 5.0GB SmallKV data for our experiments.

After loading in a 5.0GB chunk of data, the size of the collection is 16.72GB without indices, and 18.62GB after creating indices on all 16 columns. Curiously, the size of the database did not grow as much as MongoDB 4.0.9 after adding the indexes. This is especially surprising given that MongoDB 4.0.9 is using WiredTiger, which is supposed to compress indexes in memory, and is something we would want to look into given more time.

### 4.1.2 MongoDB v4.0.9

Since version 3.0, MongoDB introduced WiredTiger, which according to MongoDB documentation, compresses indices in RAM. We observed significantly improved memory usage with just the default indexes on the _id field for MongoDB 4.0.9.

After loading in a 5.0GB chunk of data, the size of the collection is 6.98GB without indices, and 18.62GB after creating indices on all 16 columns. As mentioned earlier, this increase in proportional memory usage after the creation of indexes relative to MongoDB 2.7.2 is surprising and something we would like to look into further.

### 4.1.3 Succinct

We tried several methods to measure Succinct’s post compression memory usage. We tried using Spark’s console memory measurements; we tried using GCP’s memory monitoring tools; we tried persisting the Succinct Spark RDD’s memory to disk; we tried using top across our Spark workers; and we tried to configure the C++ Succinct implementation to circumvent the difficulties of measuring memory on top of Spark. Ultimately, we could not observe memory fluctuations we felt comfortable reporting in either Spark’s console, GCP monitoring, or using top. Also, we could not compile the latest version of Succinct C++, and after talking with the original authors, they pushed a new build of Succinct C++, which we still could not compile.

We reached out to the original authors, and they recommended persisting the Succinct RDD memory to disk. This method yielded a memory usage of 6.83 GB, which is larger than the 5.3 GB of raw data. This result would contradict many of the original authors claims, but we do not know if that number includes Spark overhead or other data that should not be counted towards the final compressed size of Succinct’s data representation. We were unable to corroborate this metric using any of the other methods, and so we leave Succinct’s post compression memory usage as an important piece of future work.

### 4.2 Compression Cost

Although we were unable to accurately measure the post compression memory usage of Succinct, we did observe a large spike in memory usage during compression. Using top on each worker node, we observed an increase in memory usage of 71GB across the 10 nodes to compress a total of 5GB. The compression stage took around 258s seconds, fully utilizing two CPU cores on each worker.

Given that memory scarcity is one of the driving reasons to use Succinct, it is a little troubling that the actual compression step requires so much memory. For an already running Succinct cluster where data is incrementally appended, the compression cost is amortized by Succinct’s three-tiered data store (LogStore, SuffixStore, and SuccinctStore). Appended data is incrementally compressed as it travels through the three stores. Our cost measurements are done as if the 5GB went straight from the LogStore to the Succinct Store, however, the stores do not change the fact that under heavy load (exactly when you would hope Succinct would be most memory efficient), Succinct would require large amounts memory to compress the incoming data.

#### 4.2.1 Single-machine Succinct Shell

In preliminary testing, we used a single machine in the cluster to perform the compression. Our tests were limited to smaller data sizes since we used a modified version of the Succinct-Shell program, which allocates the data in a single byte array on the Java heap before compressing, limiting our largest possible run to 200MB.

We observed the following latencies:

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Compression Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1MB</td>
<td>0.4s</td>
</tr>
<tr>
<td>50MB</td>
<td>34.0s</td>
</tr>
<tr>
<td>100MB</td>
<td>72.2s</td>
</tr>
<tr>
<td>200MB</td>
<td>161.0s</td>
</tr>
</tbody>
</table>

Which shows a roughly linear (with some variance) relation between data size and compression time.

#### 4.2.2 10-worker Spark Succinct cluster

After setting up the 10-worker Spark cluster, we experimented loading a 5GB chunk of data into a compressed SuccinctKVRDD.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Compression Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5GB</td>
<td>258.7s</td>
</tr>
</tbody>
</table>
4.3 Throughput

4.3.1 Throughput measurements.
We measured throughput of a specific workload, namely Workload C in the original paper [2] (100% searches, following a Zipf distribution with 0.99 skewness). Our random query workload is generated using the C++ sampling implementation provided by the authors. We evaluate throughputs by measuring the total time taken to finish 10,000 random search queries (\texttt{db.smallkv.find()} for MongoDB, and \texttt{SuccinctKVDD.search()} for Succinct), and use the result to divide the total number of queries to obtain throughput. All queried values are directly sampled from the dataset itself, and therefore are guaranteed to hit.

Figure 2 shows the throughput measurement results of the original paper [2] juxtaposed against our reproduction results. We are able to obtain roughly identical throughput for Succinct, although our measured throughput for MongoDB is significantly lower than the original paper. During our experiments, we observed that repeating exactly the same query for a second time (or beyond) is an order of magnitude faster than running that query for the first time, due to MongoDB caching previous query results. As a workaround we re-sample our query keys every query to ensure MongoDB cannot cache any search results, but this might potentially explain our discrepancy against the original paper.

Another possible factor might be that our MongoDB throughput benchmarking script calls \texttt{db.smallkv.find()} for each of the 10,000 queries individually and sequentially, costing an RTT for each search. Other implementations exist (e.g. batched queries) that could lead to different throughput results.

4.3.2 Impact of MongoDB indexing assumptions.
As described in Section 4.1, the original paper [2] created an index for every single column of the dataset for MongoDB evaluations. This not only leads to larger memory usage, but also takes significant amount of time. We compared the performance with three scenarios: (1) only index
Figure 4: Left: Succinct [2] paper, Figure 13-right, comparing latency among Succinct, MongoDB 2.6.4, and Cassandra. Right: Reproduced evaluations, comparing latency among MongoDB 2.7.2, MongoDB 4.0.9, and Succinct.

the one column being queried; (2) index a total of 10 columns (chosen to roughly match the amount of time Succinct takes to compress the same dataset); (3) index all 16 columns.

Figure 3 shows that although Succinct has a higher throughput and a shorter preparation time compared to 16-column-indexed MongoDB, its preparation time (spent on compression) is significantly higher than 1-column-indexed MongoDB, meaning that Succinct will need to process many queries to amortize its large upfront cost when compared against the 1-column-indexed MongoDB. In the 10-column-indexed MongoDB scenario, it would take Succinct roughly 3,000-6,000 queries to overtake MongoDB in terms of progress.

4.4 Latency

We measure query latency of Workload C (using the same 5GB subset of SmallKV) across different systems by executing the workload sequentially. For MongoDB, we time every single individual query to obtain a series of data points, and take the average as our final reading; for Succinct, due to the nature of our implementation, we measure the total time spent going through the whole workload sequentially, and divide it with the number of queries to calculate the average latency as our final reading. This was simply due to time constraints, and for future work, we would certainly measure individual Succinct query latency similar to how we measured the MongoDB latency.

Figure 4 shows latency measurements of the original paper [2] vs. our reproduction. We can see that Succinct achieves significantly lower average search latency over MongoDB, and the two versions of MongoDB have similar latency values. The relationship between MongoDB and Succinct latency values we observe is consistent with the original paper, although we observe lower latency for both MongoDB and Succinct as compared to the original measurements. We expected absolute latency to be lower given our dataset is only 5GB, vs. 17GB in the original paper.

5 TAKEAWAYS

Succinct is technically very impressive and implements many compression techniques from academic literature, however, some of the assumptions and results in the original paper seem extreme compared to our reproduction. For example, MongoDB does not seem nearly as memory wasteful as the original paper shows, and we found Succinct very memory wasteful if you consider the memory needed during the compression stage.

Given its limited API, its requirement to fit in memory, and the memory, CPU, and latency costs of compression, Succinct seems like a rather niche choice of data store. One must on the one hand care to optimize average memory usage, while on the other, be willing to greatly expand memory usage to perform compression.

The authors have been able to build a document store, key value store, and NoSQL store on top of Succinct, which speaks to the power of their implementation, and the throughput and latency results relative to MongoDB seem very promising. However, Succinct adopters must sacrifice flexibility to realize those gains. For a dataset that doesn’t change often and must support many text search queries, Succinct is a very promising potential solution. Once you start to relax some of those assumptions, it becomes more difficult to evaluate vs. a more popular and supported database like MongoDB.

6 LIMITATIONS AND FUTURE WORK

Limited by the short time frame (5 weeks) of this reproduction project, as well as our initial unfamiliarity with the tech
stack, our reproduction only covers a subset of the original paper, and some of our measurements can be augmented for higher accuracy. For example, our measurements on Succinct are done on its Spark implementation, as we attempted to use the C++ implementation but could not get it to successfully compile under our benchmarking environment. If given more time and resources, the following extensions can be added for more complete and accurate reproduction of the original paper:

- Find a way to more accurately measure Succinct-Spark’s post-compression memory usage
- Evaluate Succinct performance using its C++ implementation
- Experiment with other workloads used in the original paper
- Experiment with workloads from YCSB [3] to show whether the authors’ workload designs are generalizable
- Reproduce the same experiments on Cassandra, HyperDex, and DB-X
- Reproduce the same experiments on AWS and observe any potential differences

REFERENCES

