A Succinct Reproduction

Iskandar Pashayev and Haiyin Wang

ABSTRACT
In this report we discuss the results and experience of our reproduction of a distributed data storage system, called Succinct, proposed at NSDI 2015. The paper, titled "Succinct: Enabling Queries on Compressed Data" and written by Agarwal, Khandelwal, and Stoica[2], aims to solve the problem of index data structures in distributed NoSQL storage systems evicting "real" data from memory, thereby actually hurting database operation throughput in distributed databases storing massive data sets. Succinct improves performance in these settings by querying on a compressed representation of the data without any indexes. We describe the design of the system and the experiments in the original paper that supported the improved throughput, and then present our reproduction results of these experiments.

1 INTRODUCTION
Modern NoSQL data stores use indexes to speed up database operations such as search. However, a big concern with indexes is that they consume too much memory, and thus push actual data out of memory and into storage. The extra storage I/O cost then actually makes database operations more costly than had there been no index. This issue is especially relevant in the case of storing massive data sets over a cluster of machines where data querying times are especially long. Indexes over data partitions consume an overwhelming amount of memory. In 2015, Agarwal, Khandelwal, and Stoica develop a new distributed data storage system, called Succinct, that avoids the construction of large indexes and yet supports efficient database operations[2].

1.1 Succinct Design
The key difference between Succinct and other NoSQL data stores is that Succinct enables query operations directly over a compressed representation of the data. This enables Succinct to maximize the amount of data that can be stored in memory by removing indexes entirely and by compressing data. This drastically increases database operation throughput and latency for massive data sets. The workhorse data structure that enables this kind of a storage system is the suffix array[7]. Even for large data sets however, suffix arrays can be prohibitively large due to storing every possible suffix of a record. Thus, the authors use a compressed representation of suffix arrays, that maps suffixes to indexes (and vice-versa), and a random subset of this compressed suffix array. Succinct also uses auxiliary data structures, such as an array of pointers that maps each suffix to the location of the next suffix (i.e. the suffix beginning at the second character in the current suffix) in the compressed suffix array. The latter data structure, referred to as NextCharIdx allows for efficient searching by directly following paths to records satisfying the queried value during the suffix array traversal.

Succinct chains the following three separate data stores for an overall data storage system.

- **LogStore** - Contains a small fraction of the entire dataset that handles small appends in main memory.
- **SuffixStore** - Accumulates bulk appends before initiating compression.
- **SuccinctStore** - Contains a majority of the data in a compressed representation using the aforementioned suffix array (related) data structures.

Each of the three data stores is itself a chain of partitions, and can live in different locations throughout a cluster of machines. Altogether, the system opens an API with extract, search, and append functions that can be used for implementing key database operations for reading, querying, and inserting multi-attribute data.

1.2 Key Experiments
The key experiments that highlight Succinct’s performance advantage run over a 10 machine EC2 cluster. There is 150 GB RAM across the entire cluster and each machine has four cores. The authors use two datasets, one with a small number of attributes and one with a large number of attributes, called "smallkv" and "largekv". The smallkv data set ranges from 17 GB to 192 GB and the largekv data set ranges from 23 GB to 242 GB. The authors shared with us the smallkv data set (which we use later in our experiments), which is represented as a list of text 16-attribute values.

There are two main figures that illustrate the performance improvement of Succinct. Figure 11 shows the difference in data footprint across Succinct, MongoDB (indexed), Cassandra (indexed), and HyperDex (columnar) storage systems. This figure shows that across the aggregate 150 GB memory, both Mongo and Cassandra can fit about 17 GB of smallkv and 23 GB of largekv. The difference in memory with the aggregate memory availability is occupied by the indexes. HyperDex performs even worse as nearly the full 150GB memory is stored by non-index metadata. Succinct’s lack of indices and compressed representation allows it to load approximately 172 GB of the data set into memory.
<table>
<thead>
<tr>
<th>Workload</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>100% reads</td>
</tr>
<tr>
<td>B</td>
<td>95% reads and 5% appends</td>
</tr>
<tr>
<td>C</td>
<td>100% search</td>
</tr>
<tr>
<td>D</td>
<td>95% search and 5% appends</td>
</tr>
</tbody>
</table>

Table 1: Workloads for Figure 12 used in the original paper for comparing database operation throughput across different distributed stores.

Figure 12 is a set of charts that compares database operation throughput across Succinct, MongoDB, and Cassandra, and across different workloads. These are summarized in Table 1. The figure shows that for smallkv, Succinct achieves much higher throughput (in terms of operations per second) across fixed amounts of "real" data loaded into the storage system. For largekv, Succinct lags behind in the read-only workload to MongoDB, but dominates MongoDB in the search workload. The measurements are at 17 GB (of original data set), the amount of real data required to saturate MongoDB and Cassandra, 62.5 GB, where MongoDB and Cassandra push data to storage but Succinct still maintains it in memory, and 192 GB, where Succinct needs to push data to storage. The authors were unable to measure throughput at 192 GB of data for MongoDB and Cassandra since the loading process was taking in excess of 100 hours.

We aimed to reproduce these two figures. Due to time limitations and infrastructure complexity, we were only able to consider Workloads A and C in our reproduction. We provide a brief overview of related work in distributed storage system design in §2. We present our experiment setup in §3 and discuss our findings in §4.

2 RELATED WORK

There is work on two extremes of the spectrum. One end concerns itself with work on using and optimizing indexes in distributed NoSQL databases[3, 8, 9]. These index data structures significantly increase database operation throughput when all data fits in memory. The work on the other end concerns itself with a system that compresses and densely packs column values in a contiguous manner, referred to as columnar stores[1, 5, 6]. Since they must scan the data, they remove all index space overhead at the cost of lower operation throughput. Succinct is a novel system that aims to bridge the gap between these extremes along the overhead-throughput spectrum.

3 EXPERIMENTS AND METHODOLOGY

We compared a clustered Succinct data store implemented over Apache Spark to a MongoDB cluster. We decided to focus on the comparison to MongoDB since (we believe) it is the most popular NoSQL database out of the three used in the paper. Both clusters were deployed on AWS. We utilize the smallkv key value dataset from the original Succinct paper for our experiments, provided to us by the authors. The following sections describe our methods for conducting the experiment and present the results. See §4 for a discussion of our results. We also note that since these experiments require setting up multiple instances with several configuration processes (in both AWS, Mongo, and Spark) and measurements for which no tool exists, there is some amount of manual command entry. We do our best to present these in a precise and transparent manner. Our collection of scripts for these experiments can be found at https://github.com/isapshayevis/cs244-final-project.

3.1 MongoDB

The mongo cluster consists of a collection of AWS instances: 10 shard servers, a configuration server, a query router, and a client conducting database operations. Our client and config servers were m5.xlarge instances (4-core CPU and 16 GB RAM). For the shard servers however, we measured space usage and throughput using both m5.xlarge (4 cores/16GB RAM) and t2.medium (2 cores/4GB RAM) instances. For more on this, see §5 Discussion. Lastly, we also use a networked filesystem (AWS Elastic File System, or EFS) to flexibly buffer data for database loads. In our experiments, the client and the query router run on the same machine. The configuration server manages the metadata for the sharded database.

The following list declares the sequence of steps used to set up the cluster and make measurements. Note that we assume the instances have been started and have been given permission to communicate with each other and access a shared EFS. We assume that ssh connections to all the servers have been started. The screen facility significantly increases the ease of navigating this process and allows us to safely close ssh connections to servers without terminating running processes. Lastly, the commands are run from the mongo directory of our git repository.

3.1.1 Preliminaries.

(1) Mount the EFS filesystem:
    ./scripts/mount-efs
(2) Download the test dataset, which we call 'smallkv' :
    ./scripts/download-smallkv
(3) Clean the smallkv dataset:
    ./scripts/clean-data
(4) Generate random keys for the workloads:
    ./scripts/get-random-points

3.1.2 Start the cluster.

We denote the configuration server as ConfigSvr, shard
server $X$ as ShardSvrX, and the client server as Client. In our case, $X$ ranges from 1 to 10. We also denote the private subnet IP addresses of these machines with an "IP_" prefix. Our sharded database is called benchmarks and the collection is called smallkv.

(1) From ConfigSvr, start mongod:
   ```
   sudo mongod --config config/ConfigSvr.cfg
   ```
(2) From each ShardSvrX, start mongod:
   ```
   sudo mongod --config config/ShardSvrX.cfg
   ```
(3) Connect to ConfigSvr and initiate the replica set:
   ```
   mongo --host <IP_ConfigSvr> --port 27019
   ```
   This can be done from any instance. When you enter the mongo shell, execute `rs.initiate()`. You can then leave the shell: Ctrl-c
(4) Connect to each ShardSvrX and initiate its corresponding replica set:
   ```
   mongo --host <IP_ShardSvrX> --port 27018
   ```
   This can also be done from any instance. Again, initiate the replica set via an `rs.initiate()` call and then leave the shell.
(5) From the Client, start a mongos process:
   ```
   sudo mongos --config config/Mongos.cfg
   ```
(6) In another Client terminal, enter the mongo shell by just executing mongo.
(7) For each ShardSvrX, add the shard by executing:
   ```
   sh.addShard("rsShardX/IP_ShardSvrX:27018")
   ```
   Note: change the X in rsShardX to match ShardSvrX.
(8) Enable sharding for benchmarks:
   ```
   sh.enableSharding("benchmarks")
   ```
(9) Enable sharding for smallkv:
   ```
   sh.shardCollection("benchmarks.smallkv", {"a" : 1})
   ```
   Note: the second parameter here specifies a sharding key, and the direction to shard in. A value of 1 means that shard keys are increasing, whereas a value of -1 means that shard keys are decreasing. The shard key is some field name in the collection, and should be uniformly distributed over its range. It is important that values in this key field are uniformly distributed over the range in order to produce evenly split shards. In our case, the sharding field is simply labeled "a:"

3.1.3 Measuring Memory Usage and Throughput.

Out of necessity, we describe this section in a more open manner. In our correspondence with the authors, the space usage is measured by gradually loading the database with chunks and measuring the total space taken by the stored data. When the total space taken exceeds the amount of aggregate RAM, we stop the insertion process. This amount is the amount of raw data that can be stored in memory. Any further insertions increases the index size and results in raw data being pushed to storage.

Our throughput tests emulate two workloads in the original paper: Workload A (100% reads) and Workload C (100% search). Throughout any point of this incremental chunk loading, the reader is welcome to perform the throughput tests. We perform our first tests after loading two 12.5GB chunks. See §5 Discussion for more details. Workload A performs 10,000 (random-access) reads. Workload C performs 10,000 searches, where we query the database for 10,000 random, indexed values. To load a chunk, execute:

   ```
   mongoimport -d benchmark -c smallkv --type csv --file ~/smallkv/12.5GB/data_0.csv -f a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p
   ```
   numInsertionWorkers 6

   To check how much total space the database is occupying, run the following command: mongo benchmarks --eval '"db.smallkv.totalSize()'. Lastly, you can execute the different workloads by calling time ./workload-a and time ./workload-c in the shell.

3.2 Succinct

To recreate the Succinct measurements we utilized an AWS cluster consisting of 10 worker instances with a single master instance with each instance provisioned with 20GB of SSD storage. This cluster was loaded with Apache Spark and the Hadoop File System through a cluster launch script. Succinct was then loaded into the system through a published spark package from the Succinct authors[Ref HERE]. We describe the steps to set up and run the cluster as well as how to collect the memory usage and throughput experimental values.

3.2.1 Preliminaries.

While we utilize the spark ec2 installation script for the general setup of our cluster with password less ssh, we additionally mount a SSD and a HDD to be utilized in our measurements.

(1) First acquire all required AWS credentials and configuration options to be used in the cluster set up script.
(2) Setup the cluster using the spark-ec2 script which sets up a cluster with specified parameters and an installation of the required base packages for Spark and HDFS. ./spark_ec2.py
(3) SSH into the master instance to configure the mounted SSD volume of the instance. To utilize the SSD volumes for your HDFS instead of the default machine memory volume you will need to set the data node directory parameter in the hdfs-site.xml file and restart the system. This will allow for the writing of larger compressed files for the experiment to disk for repeat reference.
(4) By default the script sets up an ephemeral HDFS. To persist the data, run the `stop-dfs.sh` and `stop-yarn.sh` scripts in the ephemeral directory and then run the equivalent start scripts in the persistent folder. Note: by default the HDFS will be available on port 9000 for the ephemeral system and 9010 for the persistent system.

3.2.2 Starting Succinct.
We utilize a Scala Spark program for our experiment with direct loading of the smallkv dataset from the original author’s s3 bucket. The loaded spark program will compress the given input files into a Succinct Key Value Store which we utilize to perform our measurement tasks on. Our experiment did not write out intermediate compressed files to storage due to constraints on our instance size and time. Instead data was loaded directly into memory from S3 and throughput and memory measurements were directly made on the cached data.

(1) Before beginning, compile the experiment’s SuccinctApp. Scala files from the test repository and scp it into the master instance of your cluster. The implementation of Succinct functionality in the program lies in this file.

(2) SSH into your master instance and begin the spark program utilizing the command below. Ensure that the succinct package is explicitly notated both in the command line as well as in the source dependency of the spark program
```bash
./spark/bin/spark-submit --packages amplab:succinct:0.1.7 jarfilehere.jar
```

(3) The program will load and you can review the cluster’s workload at http://[your-master-instance-dns]:8080

3.2.3 Measuring Memory Usage and Throughput.
To measure the memory usage of the compressed data by Succinct, we utilized the Spark web console that is instantiated by our cluster start up script. It records the total memory usage of each job in an application and by monitoring the outbound memory from the Succinct KVRDD calls made during compression we were able to record an accurate memory footprint of the compressed data. In the provided spark application code, we perform the experimental measurements.

In order to measure the throughput of Succinct we implemented a similar timed workload as mentioned in the MongoDB section. Workload A (100% reads) and Workload C (100% searches) as mentioned in the original paper were recreated. After compressing the input data, the provided program creates 10000 random indexes to read from and 10000 random attribute values to search on and measures the time required by the application to complete the workload. As this application is run over Spark, the provided workload is heavily parallelized and the resulting throughput values support this observation.

3.3 Results
Figure 1 shows the maximum amount of data (excluding indexes and other metadata) we were able to insert into memory in a sharded mongo database and in the Succinct Spark Cluster. This figure aims to achieve Figure 11 from the original paper. Table 2 shows Workload A and Workload C throughput for a Mongo cluster with 150 GB aggregate memory on datasets that fit comfortably in memory. Table 4 shows throughput measurements for the workloads on a Mongo cluster with 40 GB aggregate memory. We discuss why we did both of these measurements in §4 Discussion.

4 DISCUSSION
4.1 Mongo
4.1.1 Results. Our MongoDB measurements are drastically different: our experiments show that we are able to load significantly more data into MongoDB before saturating the aggregate memory in the cluster. Furthermore, our throughput measurements are much lower than the ones presented in the original paper. We suspect that this is because of a new WiredTiger storage engine that MongoDB officially adopted in December 2015 with the version 3.2 release, prior to the original publication of this paper [10, 11]. This particular storage engine utilizes a new compression technique for the data and the indexes, both in and out of memory. Based on our results, it appears that this particular technique trades lower operation throughput for a large reduction in index space.

4.1.2 Run-time and Challenges. The biggest wrench thrown at us in this portion of the experiment was the realization that the current iteration of Mongo had a far more space-efficient representation of the indexes. Our initial cluster configuration conditioned on the expectation that with 160 GB aggregate RAM, approximately 17 GB of the data set will fit in memory due to large indexes, like in the original paper. Table 2 at the end of this paper shows that this is not the case.
As we likely would have had to insert approximately 150 GB of data just to saturate the aggregate 160 GB memory in the m5.xlarge cluster, we realized that this computation would an excessively long time. Fortunately, AWS is flexible in that it allows us to downgrade instances. Thus, after loading 40 GB of data, we powered down the shard servers, and changed them to t2.medium instances. Each t2.medium has 4 GB RAM. This way, we would have already saturated the 40 GB aggregate memory. After downgrading the instances, we repeated our measurements to ensure that they had not changed too much. You can compare the difference by comparing the 37.5 GB data set size throughput measures in Tables 2 and 4. After safely dodging this bullet, we resumed the insertion process to measure throughput when the index data structures push the real data to storage. Table 4 also includes measurements with data that scales beyond main memory.

4.2 Succinct

4.2.1 Results. Our experimental results indicate that we were successful in deriving compressed file sizes utilizing Succinct which verifies the reproduction ability of memory footprint measurements from the original paper. However, as we utilized the Spark package for Succinct we saw differing throughput measurements due to the usage of Spark and its optimized parallelism. The original paper utilized a C++ implementation of Succinct in a simpler cluster setup which explains the difference in our throughput measurements.

We opted not to use the C++ implementation as we intended on extending the original work with a more recent and publicly available offering of Succinct. The Spark package was authored by the original work’s authors shortly after they released the first Succinct implementation utilized in the paper. Moreover, our usage of the public Spark Succinct package demonstrates an extension of the original work into testing Succinct’s viability in a more accessible and public facing package.

4.2.2 Run-time and Challenges.

Overall, there was a steep learning curve to many of the components required for our experiments. Specific to Succinct, was the challenge of understanding the provisioning and scaling of Spark clusters. Even with the helpful cluster set up script mentioned, the setup of HDFS to support the writing of the compressed data required a lot of initial time. Furthermore, while we show Succinct’s data compression to do well, the compression steps require a significant amount of memory to perform which lead to several re configurations of our cluster setups to support the experiments.

Minor challenges included incorrect documentation, compatibility issues, and little reference material presented additional hurdles to achieving our measurement system.

4.3 Benchmarking

The authors of the paper used YCSB for generating queries for the throughput results. YCSB is a standard industry tool for benchmarking a variety of NoSQL databases[4]. It measures a variety of statistics by loading standard data into a database. Measuring performance on a custom dataset requires the definition of a new Java class and a new workload template. For simplicity of experiment reproduction, we chose not to use YCSB for our measurements. Instead, we wrote a python script to generate random query indexes, and timed simple bash script.

5 CONCLUSION

Succinct represents a recent approach to finding an efficient middle ground between compressing data to minimize storage size and overhead and maintaining relatively competitive throughput operations. It achieves this through its unique three-chain data store which allows for queries to be conducted directly over compressed data. Through our experiments we attempted to extend the work conducted by the original paper by recreating the original measurements with more recent implementations of the database systems. This includes the measurement of the released Spark package for Succinct which allows for heightened parallelism and for a new storage engine backing MongoDB. We find that Succinct has the potential higher performance due to the parallelism afforded by Spark while maintaining its compression performance and that MongoDB has made strides in minimizing its storage overhead with its new engine. In the short time from the original paper, it is apparent that these systems have already experienced significant advancements. From our results we can infer an advancing landscape for efficient systems achieving a balance between compression and performance.

REFERENCES

Table 2: Mongo measurements taken on a cluster provisioned with 10 EC2 m5.xlarge instances (160 GB aggregate memory). Both data sizes fit comfortably in memory. Since the indexes were vastly more space-efficient than in the original paper, we paused execution and downgraded instances to not have to load more than 100 gigabytes before saturating memory. Workload A consists of 100% reads and Workload B consists of 100% searches.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Total Storage Required</th>
<th>Workload A Throughput</th>
<th>Workload B Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 GB</td>
<td>28 GB</td>
<td>29,851 ops/sec</td>
<td>20.6 ops/sec</td>
</tr>
<tr>
<td>37.5 GB</td>
<td>40.3 GB</td>
<td>49,948 ops/sec</td>
<td>20.4 ops/sec</td>
</tr>
</tbody>
</table>

Table 3: Mongo measurements taken on a cluster provisioned with 10 EC2 t2.medium instances (40 GB aggregate memory). Measurements in the first row completely saturate memory. Subsequent rows are measurements taken with data pushed to SSD. Workload A consists of 100% reads and Workload C consists of 100% searches.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Total Storage Required</th>
<th>Workload A Throughput</th>
<th>Workload C Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.5 GB</td>
<td>40.3 GB</td>
<td>45,662 ops/sec</td>
<td>20.3 ops/sec</td>
</tr>
<tr>
<td>48.6 GB</td>
<td>52.2 GB</td>
<td>27,020 ops/sec</td>
<td>20.1 ops/sec</td>
</tr>
<tr>
<td>60.3 GB</td>
<td>64.7 GB</td>
<td>23,201 ops/sec</td>
<td>20.0 ops/sec</td>
</tr>
<tr>
<td>71.3 GB</td>
<td>76.6 GB</td>
<td>29,322 ops/sec</td>
<td>20.0 ops/sec</td>
</tr>
</tbody>
</table>

Table 4: Succinct measurements taken on a cluster provisioned with 10 EC2 m4.xlarge instances (150 GB aggregate memory). Measurements are shown to have extremely high throughput. This is likely due to the high performance parallelism afforded by Spark.

<table>
<thead>
<tr>
<th>Data Size</th>
<th>Total Storage Required</th>
<th>Workload A Throughput</th>
<th>Workload C Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.2 GB</td>
<td>28 GB</td>
<td>250 k ops/sec</td>
<td>200 k ops/sec</td>
</tr>
<tr>
<td>60 GB</td>
<td>48 GB</td>
<td>200 k ops/sec</td>
<td>160 k ops/sec</td>
</tr>
</tbody>
</table>
