Building a Distributed Search Engine for Bitcoin Transactions

Maegan Jong
Harvard University
maeganjong@college.harvard.edu

Savvy Raghuvanshi
Harvard University
sraghuvanshi@college.harvard.edu

Albert Zhang
Harvard University
albert_zhang@college.harvard.edu

Abstract

The ability to effectively work with large quantities of data is arguably one of the most important contributions of modern computer science.

We built a distributed search engine to understand the complexities of both search as a problem and distributed systems as a field. Our cluster has a main master node, which sends queries to search the data that is striped across worker nodes. We utilize Bitcoin transaction data since its large volume naturally lends itself to distributed solutions, and we were motivated by the prospect of building a system using real-world data. We sorted our data in multiple ways to index it and thus improve the performance of our search queries.

To understand how our distributed search engine and its optimizations perform compared to other systems, we compared with both our own system, with only one worker node, as well as a single-node database (SQLite). We discovered that our search engine has higher throughput and lower latency relative to an equivalent SQLite database lacking indices, but lower throughput and higher latency on average relative to an indexed SQLite equivalent.

Our code can be found at this public GitHub repository: https://github.com/alzh9000/260r-Final-Project-Search-Engine.

Keywords: Bitcoin, search engine, distributed computing, distributed systems, indexing, database, parser, distributed search engine

ACM Reference Format:
Maegan Jong, Savvy Raghuvanshi, and Albert Zhang. 2022. Building a Distributed Search Engine for Bitcoin Transactions. In *Proceedings of (CS 260r: Topics and Close Readings in Software Systems).* 14 pages.

1 Introduction

We hypothesize that using a distributed system to build a search engine for Bitcoin transaction data would result in improved performance compared to using a centralized system due to the large total size of the data. As such, our introduction is generally split along two fronts, which we unify later: discussing information related to distributed systems and search engines, which is what we are designing and implementing, and information related to Bitcoin transactions, which is our use case that we are designing and implementing for.

To begin, a distributed system is a computer system whose software is run on a network of multiple machines that are orchestrated together such that the whole network functions like a single computer. Then, distributed computing is the field of using distributed systems for computing purposes. In other words, a distributed system can be thought of as an abstraction of a computer that is generally able to provide better scalability, compute power, storage, and redundancy than a single machine due to being comprised of a network of multiple machines, which can also be extended with even more machines, that combined together have potentially much better capabilities than a single machine. For instance, if a distributed system is used for a certain software task, if that task requires more compute power or more storage because more users joined to use that software, then more computers and storage devices can simply be added to the distributed system, whereas this expansion would be much more difficult to perform on a single machine.

However, there are downsides of distributed systems too: generally, they are harder to build, write code for, debug, and maintain than centralized systems. Also, additional effort needs to be put into setting up the network and security for the distributed system compared to for a centralized system. Conceptually, one can think of the trade-offs of a distributed system versus a centralized system as somewhat broadly analogous to the trade-offs of a multi-threaded program versus a single-threaded program. As such, generally, one should not use a distributed system unless the task at hand necessitates it for the aforementioned strengths of distributed systems, like better scaling or storage.

A search engine is a software system designed to enable users to perform searches (also known as queries) for information, generally in a database, relatively quickly. Usually, for a search engine to have good performance, it is necessary to index the database with respect to the types of data users can query for, where indexing is the process of organizing the database to make certain types of queries faster [5]. For instance, if users want to query a tabular database of people's heights for certain heights, one could index the database by sorting all the rows by height, so that when a user looks for the rows corresponding to a certain height, those rows can be found more quickly using binary search (running in $O(\log n)$ time) rather than the naive linear search (running in $O(n)$ time). Note that the process of sorting the rows in this case

CS 260r: Topics and Close Readings in Software Systems, May 14, 2022, Harvard University

takes $O(n \log n)$ time, so there is a tradeoff to this indexing where we do more work upfront (when sorting) in order to do less work later (when performing binary search). If only one search is ever performed then the naive linear search is faster than sorting plus binary search, but as more and more searches are performed, the indexing by sorting strategy wins out because each additional search (done via binary search) takes less time compared to the naive approach.

Thus, we see that the best strategy for indexing depends on the kind of searches that should be made, and that there are often tradeoffs involved when indexing. Another strategy of indexing for a search engine is by building an inverted index, which is a mapping from desired queries (such as keywords) to locations in the database or content, so that when a query is made, the corresponding data in the database can easily be found at the locations given by the mapping [5]. So, for a web search engine whose tabular database contains website links, an inverted index could be a mapping from common search terms (ex: “dog" or “learning to code") to row numbers for corresponding website links in that database. The tradeoff in this case is the extra work required upfront in creating the inverted index, as well as the extra space required to store the inverted index. As such, an inverted index sometimes only contains the locations for popular queries rather than all possible queries, like how the index at the back of a textbook does not list the locations of all the words in the textbook, just the important ones.

As the database that a search engine searches over grows larger, the search engine’s performance suffers because it takes longer to search over greater quantities of data, in addition to the extra storage required to fit the larger database, even with indexing. For instance, an inverted index requires more space and must be updated as its corresponding dataset grows. So, as “big data" has become increasingly popular, distributed search engines have been important for searching over that “big data" because they leverage the vast potential of compute and storage that distributed systems have in order to maintain usable, interactive performance for searching. As such, we turn to discuss the dataset we want to search over for this paper, which is large enough to necessitate a distributed search engine: Bitcoin transaction data.

Bitcoin is a both a digital currency and a decentralized blockchain through which the Bitcoin cryptocurrency is transacted, and this blockchain is a distributed ledger where all Bitcoin transactions are publicly stored in blocks [12]. As such, it is possible to inspect and analyze the Bitcoin blockchain for information on Bitcoin transactions, which are transfers of Bitcoin from one wallet address (the sender) to another wallet address (the receiver). For each transaction, the associated version number, flag (which represents whether there is witness data), in-counter (which represents the number of inputs), list of inputs (which represent the addresses that are sending the Bitcoin), out-counter (which represents the number of outputs), list of outputs (which represent the addresses that are receiving the Bitcoin), list of witnesses (which represent signatures that show that the transaction is valid and authenticated), and lock_time are stored with the transaction inside a Bitcoin block [3]. This information is stored in a binary format inside the blocks, so in order for a human to be able to read, understand, and use this data, it is necessary to parse that binary.

Since Bitcoin’s inception in 2008, its usage has greatly increased in popularity, with hundreds of thousands Bitcoin transactions being added via blocks to the blockchain each day, and there being hundreds of millions of Bitcoin transactions in total stored on the blockchain ledger, which makes up over 400 gigabytes of binary data. One way to access this data is to set up a full node for the Bitcoin network, through which one can download the entire Bitcoin ledger, including the binary data on the transactions. As Bitcoin’s usage grew, analyzing and searching through this transaction data became increasingly important, such as for criminal justice and investigation related purposes since research has shown that Bitcoin has been used for crimes including ransomware, money laundering, illegal online purchases, and more [13]. So, being able to search through Bitcoin transaction data for any information useful to investigations into these matters is helpful for law enforcement and society. In particular, tracing the lineage of a particular transaction, both forwards and backwards, is a well-motivated application and the one that we design our search engine and database around.

In order to be able to effectively search Bitcoin transaction data, we hypothesize that we would benefit from making a search engine distributed over multiple machines. Additionally, because the Bitcoin blockchain is still actively used and has grown more popular over time, the total size of the Bitcoin transaction data is increasing rapidly, so using a distributed system would help with this scaling aspect. Otherwise, if we build a centralized search engine for Bitcoin data, it may become too slow to use in the future when the Bitcoin transaction data is much larger, while scaling is easier for a distributed system. By leveraging the benefits of distributed systems, we can more effectively build a search engine for the massive Bitcoin data.

Our contributions include the following:

- A binary parser for raw Bitcoin transaction data dumped by the BitcoinCore client. This parser can output the resulting data in either our custom data format or into a SQLite database. For our custom format, it can also split the data up among a desired number of worker machines.
- A search engine master, which connects to a provided list of workers and sends queries off to them. While the current master is non-interactive and runs a set of random queries broadcast to all clients in a loop, its

design is flexible enough to permit several searching models, including stragglers and replication.
- An implementation of the search engine worker. This is basically the workhorse of the engine, and runs on its own set of data on a machine. The worker can be replicated as many times as desired, and can be initialized with arbitrary subsets of the data to perform searches on.
- Performance measurements of our system and comparisons with single-machine and SQLite baselines.

The rest of this paper is organized as follows. Section 2 contains a body of related work and background for our project. Section 3 describes the design of our system, including the parser, master, and worker. Section 4 describes an implementation, followed by evaluation in Section 5.

2 Background and Related Work

The direction of our work matters because searching through Bitcoin transaction data has many uses, from gaining information on the purchasing habits of Bitcoin holders and investors, to finding which wallets hold the most Bitcoin or send Bitcoin the most frequently, to tracing transactions of interest in an effort to identify who is involved in them. This last point has been especially important for organizations and individuals, such as the FBI, that are interested in investigating crimes related to Bitcoin. For instance, the FBI shut down the Silk Road, an illegal black market website for purchasing drugs using Bitcoin, and has attempted to trace illicit drug purchases on the website by tracking which wallets are connected, whether directly or indirectly, to Bitcoin transactions on the Silk Road [4]. Although Bitcoin wallets are technically anonymous, unlike a bank account, because in many cases Bitcoin eventually needs to be converted into fiat money to be used in the real world, that conversion can oftentimes reveal the identity of a Bitcoin wallet, which gives more information on that wallet and the other wallets that have transacted with that one. Thus, organizations are interested in tracing transactions from points of suspicion (like an illegal drug purchase on the Silk Road) to points where identities are revealed in order to track the people involved in the original suspicious transactions. This strategy has also been used to track down Bitcoin money laundering criminals, Bitcoin hacks, and Bitcoin Ponzi schemes [4].

Then, as stated earlier, for us to attempt to provide an effective, performant search for the large Bitcoin transaction data, we turn to distributed systems for the scalability and power that we want. Much work has already been done in the area of distributed systems. We focus first on work in the area of distributed search engines, which has seen some interest in the past two decades. Note that since we are interested in a system that is expected to repeatedly perform queries, the general strategy used for this involves indexing the data so that all queries are faster, rather than strategies for just making a single query faster. As such, some systems including MapReduce [7] are not the best for our goal because they generally are not designed for making repeated searches on the same dataset, especially when we know what kind of searches are likely to be made. For instance, MapReduce can greatly speed up searching using grep, which works well for querying a single time but is not optimal for multiple queries. The latency of a MapReduce-based system for our data would be extremely high.

In [10], Kronfol creates a fault-tolerant, adaptive, scalable, distributed search engine that uses a network of nodes which combine storage to cache metadatakeys that are used for indexing content for faster search. Importantly, Kronfol's distributed search engine is able to match a centralized search engine in performance, which shows us that it is possible to create effective distributed search engines in practice. For our distributed search engine, we also use a somewhat similar concept of keys to index the Bitcoin transaction data. There are additional ways to index more effectively too. Zhang and Suel explore a more optimal indexing strategy for a distributed search engine in [14] by grouping index data corresponding to terms that frequently occur together in queries onto the same node to improve overall search speed, since this makes it more likely that a given query would only need to touch one node in the distributed system rather than multiple nodes. However, this strategy likely does not really apply to a Bitcoin transaction search engine since the queries for Bitcoin transaction data are simpler and involve less terms in one single query.

There are several existing papers related to the application of distributed search engines to other fields similar to how we want to apply distributed search to Bitcoin transaction data. In [2], Baeza-Yates discusses how layered caching and online prediction mechanisms can be applied to build a distributed search engine for Internet webpages, which might have some relevancy in building out a more advanced Bitcoin search engine with more complex queries enabled. Distributed search has also been applied to geospatial resource discovery, which Gui et. al pursue in [9] to support the development of geographic science applications, giving users more capabilities for analyzing geospatial data and enhanced performance. This empowering of users is similar to how a Bitcoin transaction search engine would empower users to better explore and analyze Bitcoin transactions.

The above related work is helpful for our ideation and design of building out a distributed search engine, so now we turn to understanding work done relating to our dataset of interest, Bitcoin transactions, to see if there are any insights here that will be helpful to our development. The original Bitcoin whitepaper [12] by Satoshi Nakamoto contains important information on what Bitcoin transactions are and what information they contain, which we use to decide what our search engine should even be capable of searching for.

Work by Zhao and Guan in [15] illuminates how analyzing Bitcoin transactions can give interesting insights, with Zhao and Guan using graph-based methods to investigate the clustering and flow of Bitcoin, especially for transactions related to criminal activity like the stealing of Bitcoin from the Mt. Gox exchange.

There is also additional work related to the analysis of Bitcoin transaction data. [11] dives deep into judging the level of anonymity that Bitcoin provides, especially when mixing services are involved, which help anonymize transactions by obfuscating their origins. Moser finds that, though these mixing services do help anonymize transactions, they are not perfect, and in many cases it still is possible to track down identities through Bitcoin transactions, so we confirm through this that it is worthwhile to be able to search through Bitcoin transaction data. Then, in [6], Conti et al. investigate the economic impact of ransomware campaigns conducted involving Bitcoin and find that the usage of Bitcoin by cybercriminals is massive, with hundreds of millions or even billions of U.S. dollars, depending on Bitcoin's value, worth of Bitcoin being involved in criminal activity. As such, it is important to be able to search Bitcoin transaction data to aid in the relevant investigations.

So, we have presented much of the background for why our work matters by showing related work for both distributed search engines and for Bitcoin transactions. The related work indicates that there both is a good use case for wanting to search Bitcoin transaction data and that using a distributed system for building the search engine is a good candidate for better performance.

3 Design

Our design can approximately be divided into two main parts: the parser and the distributed search engine. We wrote a custom parser in Rust to parse the binary Bitcoin transaction data so we can search through it and use it for parsing into both SQLite databases that we use for baseline comparisons and our own custom data format that our distributed search engine uses. Then, we discuss in more detail our own distributed search engine that we designed and implemented.

3.1 Parser

Before we had any data to parse, we had to download the history of the Bitcoin blockchain. There is over 400GB of data, and Bitcoin runs on a vast peer-to-peer network of computers, most of which have throttled internet connections to keep costs down. The BitcoinCore client (the official Bitcoin implementation) only connects to 10 peers at once, and many of those peers have slow connections. Even if the client connects to a fast peer, in our observation the peer sharply limits or cuts off upload traffic to our node after 1GB of transfer. These design decisions make sense for the Bitcoin network; in practice, nodes are meant to be long-living and exchange little traffic with one another in the steady state, and this traffic is also heavily weighted towards more recent data on the chain. However, for our project, we needed this data fast, so we forked the BitcoinCore client to allow for more peer connections and larger download buffers (to minimize TCP backpressure). This significantly sped up our downloads, and we were able to download all of the data within a day.

We tried several ideas for parsing the binary Bitcoin transaction data before settling on creating our own custom parser in Rust. First, we looked for existing parsers online and found many, several of which were in Python, but they did not fit our purposes because they were either outdated, too complicated, too slow (especially the Python parsers), or too hard to modify to store the parsed data in a convenient format for us. Some of those drawbacks we had to discover the hard way: we had some success getting a couple Python parsers to work, including modifying one substantially to store the parsed data in CSV files, only to discover when testing it on the large raw, binary Bitcoin transaction data files that the code was far too slow to parse all the data that we wanted. The final nail in the coffin for using an existing parser was that the ones we found relied on the raw, binary data files being from an older version of the BitcoinCore library which did not account for differences that some of the newer transaction formats had, whereas we obtained some of the raw, binary data files from a newer version by setting up one of our computers as a full Bitcoin node, giving it access to the entire record of Bitcoin transactions as those raw, binary data files.

Learning from this, we decided to write our own custom parser that would support these newer transaction formats and be performant enough to parse all of the blockchain data. So, when deciding on which programming language to write our parser in, we wanted it to meet these criteria: fast, memory-safe, compiled, and with good bindings to database libraries. We care about the parser being fast so that we can successfully parse the gigabytes of data in a reasonable amount of time. We want the code to be memory-safe and compiled, with a strong type system, so that it is less likely for us to run into bugs and errors when building the parser, plus compiled languages are generally faster too. Finally, because we want to have the parsed data stored in a database like SQLite at least for testing purposes, we want the programming language we choose to have good packages or libraries we can easily use to parse our data directly into a database (rather than first storing the data in some intermediary format, like in CSVs). Based on these factors, we settled on Rust as the programming language of choice, as it fulfills each of these criteria well. The primary downside of using Rust is that our team had not used Rust before, so learning it for this project was an added challenge.

When building our parser, we thought about how Bitcoin transaction data are designed for verification, not analysis,

and so the information stored in that data is mostly just what is necessary for Bitcoin miners to verify transactions. Our parser parses the raw, binary data files and stores the relevant information into three SQLite tables: `blocks`, `transactions`, and `input_output_pairs`. The `blocks` table contains information on each Bitcoin block, which can include many transactions, with each row consisting of the `block_hash`, `version`, `prev_block_id`, `merkle_root`, `unix_time`, `tx_count`, and `height` of that block. The `transactions` table contains information on each Bitcoin transaction, with each row consisting of the `id`, `version`, `block`, and `size` of that transaction. The `input_output_pairs` table contains information on each input (sender) and output (receiver) pair of wallets for a transaction, with each row consisting of the `src_tx`, `src_index`, `value`, `dest_tx`, and `dest_index` of that transaction. Note that the `input_output_pairs` table is composed primarily of data we have derived ourselves from the underlying data, rather than simply being read off the BitcoinCore binary format. In particular, we had to match the outputs of transactions with the inputs of other transactions in memory ourselves.

These SQLite databases are used for our baseline comparison, where we query these databases on a single machine. Each of these columns in these databases represents some piece of information relating to Bitcoin blocks or transactions, which are described in more detail in [8] and [1].

For our purposes, we opted to optimize for the specific use case of tracing the lineage of a particular transaction, given its hash. In particular, we'd like to be able to compute parent and child transactions of a given transaction efficiently and quickly.

In addition to dumping data into SQLite, we wrote our parser to drop data in our own custom data formats — one for each type of data we were storing (`transaction`, `block`, or `input_output_pair`). These were binary files encoded with the `bincode` crate and dumped to disk in order. This allowed us to read the data files from disk linearly and deserialize the data into Rust structures in memory very quickly.

When building our parser, and its associated search engine, we made some different performance tradeoffs than the SQLite authors, and it showed. In particular, we found that SQLite was painfully slow to dump data into. Initial runs of our parser showed around 3 minutes to insert 100MB of data into SQLite. Disabling disk synchronization and moving the Write Ahead Log (WAL) off disk and into memory dropped this insertion time by about a factor of 10, but it was still pretty slow compared to writing data in our custom data format. Both SQLite insertion and our custom data format insertion occurred row by row, though the SQLite insertion had the advantage of occurring with a cached prepared statement. Even when we would write our data to disk, read it from disk, sort it, and write it back to disk, our parser was faster than an unindexed SQLite instance.

Despite the relative speedup of writing to our custom format over SQLite, we still know there is substantial room for speedup in our parser codebase. In particular, the parser code is single-threaded, and computation and data transfer are never overlapped. We read an entire raw binary file from disk into memory, and then parse the file in memory, writing structs to disk as we have fully parsed them. Pipelining data transfer with parsing and data output would improve our parsing speed even more.

Note that while our parser can output data into our custom file format faster than it can into SQLite, this performance benefit comes with an associated cost — when a worker node starts up, it must read all the serialized data in our custom format into memory before being able to accept queries. Therefore, our improved parsing speed comes at a system startup cost. This was a worthwhile tradeoff for us given the constraints of this project and the limited amount of memory on the second author's computer (where the parser code was being run).

Note that our parsing methodology was not without significant flaws — in particular, the requirement that we read all data into memory to operate on it meant that our maximum working set size was limited to around 50GB (including swap) on the machines we had available. Since our indexing strategy (see below) relies on storing some of the data twice (with different sorts), that further limits the amount of data our worker and parser can handle. Furthermore, our parser started crashing on the 374th Bitcoin data file, so we capped our dataset at 373 raw files (i.e. around 32 GB) and worked with this subset of the data. This takes us up to November 2015 on the chain, so our data still covers a substantial period of time and has some real-world applicability.

3.2 Distributed Search Engine

Now, we discuss the design for our distributed search engine. First, we focus on the search engine aspect of it by discussing what kinds of queries it supports, how its data is organized, and what kinds of indexes it has to speed up the queries. Later, we focus on the distributed aspect of it by discussing how we use multiple machines to split up the computation, as well as how those machine communicate and stay organized with one another.

The primary query that our distributed search engine is built for is the query of finding the parent transactions or finding the child transactions of a given transaction. For a transaction T, which has a list of input wallets T_I that send the Bitcoin and a list of output wallets T_O that receive the Bitcoin, the parent transactions of T are the transactions whose output wallets are in T_I, so in the web of all Bitcoin transactions, the parent transactions precede T because the output wallets that receive Bitcoin from the parent transactions then send Bitcoin for T. Analogously, the child transactions of T are the transactions whose input wallets are in T_O, so in the web of all Bitcoin transactions, T precedes its child

transactions because the input wallets that send Bitcoin in the child transactions received Bitcoin from T previously.

The reason why we focus on supporting parent and child transaction search queries is because finding the parent and child transactions of T is crucial in tracing where and how Bitcoin flows from an origin transaction. For instance, if 100 Bitcoin are stolen in a hack in transaction T_H, then investigators would like to find the child transactions of T_H, then the child transactions of those child transactions, and so on through the descendants of T_H, to trace where that 100 Bitcoin is flowing, because eventually it is possible that one of the descendent transactions will reveal one of the actor's identity, such as if a descendent transaction is sending some of the stolen Bitcoin to a bank or Coinbase. On the other hand, by searching for parent transactions, if investigators find a transaction T_A which reveals an actor's identity, the investigators can work backwards from that identity, searching for the parent transactions of T_A, then the parent transactions of those parent transactions, and so on through the ancestors of T_H, until they find relevant evidence that may indicate that that actor was involved in the hack.

Thus, we want to figure out how to index our data effectively to support searching for parent and child transactions quickly. We do this by indexing our custom data format's equivalent of the `input_output_pairs` table, which contains information on each input (sender) and output (receiver) pair of wallets for a transaction. We index this dataset by storing two copies of it: one copy is sorted by the `src_tx` column, which tells us the inputs, while the other copy is sorted by the `dest_tx` column, which tells us the outputs. By doing so, we sacrifice space since now we have twice the amount of data, but doing so enables us to make both parent and child transaction search queries fast. We want one copy to be sorted by the `src_tx` column because this speeds up querying for the child transactions of T since those desired child transactions will have T match their value in the `src_tx` column so we can now use binary search with $O(\log(n))$ runtime for this column to find those desired child transactions rather than the naive linear search with $O(n)$ runtime. Similarly, we want the other copy to be sorted by the `dest_tx` column because this speeds up querying for the parent transactions of T since those desired parent transactions will have T match their value in the `dest_tx` column so we can now use binary search with $O(\log(n))$ runtime for this column to find those desired child transactions rather than the naive linear search with $O(n)$ runtime.

In addition to supporting queries for parent and child transactions, our distributed search engine also supports querying for information on a specific block given the corresponding `block_hash` and querying for information on a specific transaction given the `id` of that transaction. These are our search engine's equivalents of finding a specific row in the `blocks` SQLite database and finding a specific row in the `transactions` SQLite database, respectively. We speed up the `block_hash` queries by indexing our custom data format's equivalent of the `blocks` database by sorting it by the `block_hash` column so we can use binary search by that column, giving runtime speedups as previously discussed. Similarly, we speed up the transaction `id` queries by indexing our custom data format's equivalent of the `transactions` database by sorting it by the `id` column so we can use binary search by that column, giving runtime speedups as previously discussed. Note that for each of our indexes, we are able to generate them using sorting because the columns that we sort all have a numerical representation that enables sorting. In particular, we simply sort by the binary data of each hash.

Now, we discuss how we design the distributed aspect of our distributed search engine. The distributed system we use consists of a master machine and some number of worker machines (for our tests, we use three), where the master is primarily responsible for orchestrating and serving as a communication hub for the worker machines. The workers and master communicate with each other using Remote Procedure Call (RPC). When a user makes a query on our search engine, the master will send requests to the workers that correspond to that query, such as requests along the lines of "return all the children of these transactions" or "return all parents of these transactions." The workers then respond using RPC, and then the master uses those responses to construct and return the results for the user's query. We implement this in Rust using `tarpc`, an RPC framework for Rust, because it has the benefits of not having a separate compilation process and not requiring context switching and code duplication between different languages (Rust and Protobuf). We decided this after experimenting with another RPC framework (`Tonic`) and realizing the additional complexities involved with sending requests using Protobufs. `tarpc` instead defines the schema in Rust code, simplifying the exchange of messages.

We design the workers such that the whole process simply involves our master first sending requests to the workers concurrently, then our master waits for all the responses via RPC, then the master combines the responses. To do so, we split the tables of data we are searching over, which are our custom data format versions of the `blocks`, `transactions`, and `input_output_pairs` SQLite databases after we sort them for indexing purposes as described earlier, between each of the n workers. We split each of rows of these tables between the n workers in a cycle by using using the modulo operator: for each table, worker i of the n total workers gets all of the rows with an index congruent to $i \pmod{n}$. Conceptually, this is as though we assign the 1st row to worker 1, the 2nd row to worker 2, ..., the nth row to worker n, then we cycle back, to assign the $(n+1)$th row to worker 1, the $(n+2)$th row to worker 2, and so on. Additionally, note that this strategy for splitting up the sorted data maintains the sorted ordering of the rows for every worker, which is

important so that every worker can still individually perform binary search on its partition of the data, since its partition is still in sorted order.

By splitting data in the above manner, this has the tradeoffs that we suffer from the fan-out problem of latencies. In essence, these queries, despite being closely grouped, have to be made through separate network requests. In particular, if we have n worker machines, then we have to wait for n queries to complete (simultaneously) and for at least n network roundtrips to different workers to send data. This increases the overall system latency, particularly at the tail (which we see in our evaluation), because we are increasing the number of network connections that need to be made. However, this structure has the advantage that repeated queries of the same few transactions that are clustered together will not overload a single machine because neighboring transactions will be stored on different workers.

With this relatively straightforward partitioning of data, it is clear that if we want to search for a certain query, we just need each worker to perform that search over their own partition of the data, and then we simply combine the results of that search together on the master, filtering out any duplicates, and return them. No additional complex work needs to be done by the master, which is good since the single point of failure has its workload minimized. Additionally, it is clear how our distributed system can work with even just one worker (in which case all rows are put on that single worker and we have a trivial partition) and should in theory become faster as the number of workers n is increased, assuming n is less than the number of rows, since the more workers, the more the data is split up, and thus each individual worker's search procedure will be faster.

Note that our worker design also enables implementation of fancier systems from distributed systems we have studied this semester — in particular, since our master node filters out duplicate data, we can freely replicate data across different workers without any change to the codebase. If we doubled the number of workers, striping data twice instead of once, our system would have much greater reliability without any change to the codebase (at the expense of higher tail latencies and more network traffic). Note also that our choice of RPC library (`tarpc`) and runtime (`tokio`) mean that we can freely combine or discard the results from different workers, cancelling in-progress requests. In our master, results from workers are represented as Rust futures, which can be manipulated easily in the code. For instance, to use the result from the fastest worker, we can use the `tokio::select!` macro (rather than the `tokio::join!` macro currently used to wait for all workers). By combining these macros, we can design a system with arbitrary levels of replication in both data and requests. For example, for each query, we could fire off n pairs of identical requests to $2n$ different workers (two of the same request for each worker), taking the faster result within each pair (and cancelling the requests that were not satisfied in time) but waiting for all pairs to complete before returning a response.

4 Implementation

For the deployment of our distributed search engine and our centralized SQLite databases, we use Microsoft Azure Virtual Machines (VMs) for both. For our centralized SQLite databases, we host, run, and query those on a single Azure VM, while for our distributed search engine, we have separate Azure VMs for each of the master and n workers. To use these VMs, we wrote Bash scripts using the Azure command-line interface (CLI) to spin up our VMs. We also upload our code, files, and data to the VMs accordingly, then run our experiments on the VMs, and then once the experiments are completed, shut down our VMs to avoid incurring any additional charges from Azure. All of the Azure VMs we use are identical, and for our distributed search engine, all of our VMs (master and all workers) are running single-threaded. As such, we hope that using more workers increases our performance because using more workers incurs a greater cost. Even if increasing workers improves performance, we also would need to consider how valuable that performance improvement is compared to the increased costs.

Our code is deployed on `Standard_E4-2ds_v4` Azure VMs running Ubuntu 20.04, which each have 2 virtual CPUs, 32 GiB of RAM, and are located in the same Azure region of `Central US`. We chose these Azure VMs because they strike a good balance between being sufficiently powerful to run out code while also not being too expensive, which seems representative of the same balance that many systems have to achieve in the real world. Our code also runs on MacOS and Windows Subsystem for Linux with Ubuntu 18.04, which we used on our own laptops for local development, debugging, and testing. We realized that there are some complications with Azure regarding resource quotas such that Azure limits the number of CPUs you can utilize depending on your subscription. We prioritized RAM over CPUs after some of our CPU quota increase requests were denied.

As for more specifics of our code implementation, our implementation is built for Rust Version 1.60.0, the latest version at the time of writing, and uses several dependencies to achieve the functionality we wanted from the parser, distributed search engine, and SQLite wrapper code. We use `tokio` to serve as an asynchronous runtime, `serde` for serializing and deserializing Rust data structures, `clap` for parsing command line arguments (which is especially helpful for when we want to run our code on the Azure VM without repeated recompilation), `tarpc` for providing a framework for RPC, and `rusqlite` for interfacing with SQLite. All of these libraries are industry standard and some of the most popular in their categories.

5 Evaluation

5.1 Experimental Setup

For our experimental setup and deployment, our distributed search engine is designed to be run and tested on Azure, where we use Azure VMs to have a consistent platform to run tests on and compare results more fairly. We also run our baseline experiments on Azure. The configuration of these Azure VMs is detailed earlier in the Implementation section.

The baselines that we compare our distributed search engine with are:

- Similar queries performed on a SQLite database with the same data running on a single machine, with indices targeting the queries we are optimizing for,
- The same SQLite database, without any indexing,
- Our distributed system but only running with a single worker.

The indexes we create in SQLite are created with the following commands:

```
CREATE INDEX iopairs_index_src ON
    ↪ input_output_pairs(src_tx);
CREATE INDEX iopairs_index_dest ON
    ↪ input_output_pairs(dest_tx);
```

Because our distributed search engine's capabilities are based off of the queries we can make on the `blocks`, `transactions`, and `input_output_pairs` SQLite tables, we just make the comparisons correspondingly. For instance, we would compare the performance of a parent transaction query on our distributed search engine to the performance of a similar parent transaction query on the `input_output_pairs` SQLite database on a single machine. To make the comparison fairer, we run our baselines using the Rusqlite package, which is a performant wrapper for using SQLite from Rust, and a similar format to the Rust code we use to build our distributed search engine, so that any overheads should apply to both our baseline and distributed search engine in testing. Note that we do not run queries with exactly identical inputs on the SQLite and distributed engines, but rather perform our queries on uniformly sampled random inputs in a loop with many iterations.

For each of our testing configurations, we track throughput and latency. We measure the throughput of a query by timing how long in seconds it takes in total to run I iterations of that query, while timing the whole loop. Then, dividing I by that total time tells us how many iterations of that query can be run in 1 second, which is the throughput. We want higher throughput because that means we can run more queries in the same amount of time, indicating higher performance. Also, since our master and all workers are running single-threaded for our distributed search engine, note that the throughput we are measuring is single-threaded throughput. We measure the latency of a query through the a loop over I iterations of that same query, where we time the latency of each individual query to be the amount of time it takes to get the result for that single query. Thus, in total we will have I latency values for that query, which we then use to compute our latency statistics: the mean, standard deviation, minimum, maximum, 1st quartile, 2nd quartile, 3rd quartile, and other percentiles (in the format pX, which represents the Xth percentile, such as p99.9 representing the 99.9th percentile), over those I latency values. We want lower latency because that means each individual query generally takes less time to run, indicating better performance. Note that because for throughput we time the whole loop, whereas for latency we time each individual iteration in the loop, we only get one value for throughput (though this value should be robust considering it is taken over I iterations), while we get a whole suite of statistics for latency.

5.2 Experimental Results

We conduct our measurements of throughput and latency for four systems: unindexed SQLite, indexed SQLite, our distributed search engine with a single worker, and our distributed search engine with three workers. The unindexed SQLite and indexed SQLite results will serve as a baseline for our distributed search engine, and we experiment with our distributed search engine with a single worker versus three workers to see how the number of workers impacts performance. The results of our experiments split into two parts: throughput results and latency results. We present our throughput results in tables because this is more compact and because we only have 1 measurement/value for throughput (the average throughput), while we present our latency results in plain-text because there are too many statistics/measurements to fit them in a table (doing so would require a 3-dimensional table). Our throughput tables are in 1, 2, 3, and 4. Our latency results are below.

We run all experiments on three different data sizes — small (around 100MB), medium (around 10GB), and large (around 30GB).

5.2.1 Latency for Unindexed SQLite. Here are experimental results for latency, which we provide detailed statistics on:

```
Latency performance results for Unindexed SQLite
    ↪  on Small Data in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 1000
    ↪ iterations complete. Statistics:
Mean latency: 55553638.4 ns
Std deviation: 1290350.5612566285 ns
Min latency: 52723712 ns
p25 latency: 54624255 ns
p50 latency: 55476223 ns
p75 latency: 56459263 ns
```

Throughput for Unindexed SQLite (in queries/second)		
Data Size	Child Transactions	Parent Transactions
Small	17.898	17.829
Medium	0.13571	0.13362
Big	0.01725	0.04574

Table 1. Throughput for Unindexed SQLite (in queries/second)

Throughput for Indexed SQLite (in queries/second)		
Data Size	Child Transactions	Parent Transactions
Small	82135	99694
Medium	28515	37847
Big	2717	6986

Table 2. Throughput for Indexed SQLite (in queries/second)

Throughput for Single Worker Distributed Search Engine (in queries/second)		
Data Size	Child Transactions	Parent Transactions
Small	1488	1443
Medium	1424	1459
Big	Data too large	Data too large

Table 3. Throughput for Single Worker Distributed Search Engine (in queries/second)

Throughput for Three Workers Distributed Search Engine (in queries/second)		
Data Size	Child Transactions	Parent Transactions
Small	1719	1725
Medium	1702	1715
Big	1807	1813

Table 4. Throughput for Three Workers Distributed Search Engine (in queries/second)

```
p90 latency: 56983551 ns
p95 latency: 57376767 ns
p99 latency: 58327039 ns
p99.9 latency: 62226431 ns
p99.99 latency: 69992447 ns
p99.999 latency: 69992447 ns
Max latency: 69992447 ns

parents queries latency test...
parents queries latency test with 1000
    ↪ iterations complete. Statistics:
Mean latency: 55999381.503999986 ns
Std deviation: 1015630.6989443172 ns
Min latency: 54296576 ns
p25 latency: 55476223 ns
p50 latency: 55967743 ns
p75 latency: 56492031 ns
p90 latency: 56852479 ns
p95 latency: 57212927 ns
p99 latency: 58032127 ns
p99.9 latency: 60620799 ns
p99.99 latency: 77856767 ns
p99.999 latency: 77856767 ns
Max latency: 77856767 ns
```

```
Latency performance results for Unindexed SQLite
    ↪ on Medium Data in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 10 iterations
    ↪ complete. Statistics:
Mean latency: 7361003519.999999 ns
Std deviation: 151044451.97259277 ns
Min latency: 7168065536 ns
p25 latency: 7205814271 ns
p50 latency: 7356809215 ns
p75 latency: 7470055423 ns
p90 latency: 7482638335 ns
```

```
p95 latency: 7633633279 ns
p99 latency: 7633633279 ns
p99.9 latency: 7633633279 ns
p99.99 latency: 7633633279 ns
p99.999 latency: 7633633279 ns
Max latency: 7633633279 ns

parents queries latency test...
parents queries latency test with 10 iterations
    ↪ complete. Statistics:
Mean latency: 7491865804.799999 ns
Std deviation: 93619458.49655667 ns
Min latency: 7356809216 ns
p25 latency: 7398752255 ns
p50 latency: 7541358591 ns
p75 latency: 7558135807 ns
p90 latency: 7566524415 ns
p95 latency: 7637827583 ns
p99 latency: 7637827583 ns
p99.9 latency: 7637827583 ns
p99.99 latency: 7637827583 ns
p99.999 latency: 7637827583 ns
Max latency: 7637827583 ns
```

```
Latency performance results for Unindexed SQLite
    ↪  on Big Data in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 1 iterations
    ↪ complete. Statistics:
Mean latency: 49811554304 ns
Std deviation: 0 ns
Min latency: 49794777088 ns
p25 latency: 49828331519 ns
p50 latency: 49828331519 ns
p75 latency: 49828331519 ns
p90 latency: 49828331519 ns
p95 latency: 49828331519 ns
p99 latency: 49828331519 ns
p99.9 latency: 49828331519 ns
p99.99 latency: 49828331519 ns
p99.999 latency: 49828331519 ns
Max latency: 49828331519 ns

parents queries latency test...
parents queries latency test with 1 iterations
    ↪ complete. Statistics:
Mean latency: 21785214976 ns
Std deviation: 0 ns
Min latency: 21776826368 ns
p25 latency: 21793603583 ns
p50 latency: 21793603583 ns
p75 latency: 21793603583 ns
p90 latency: 21793603583 ns
p95 latency: 21793603583 ns
p99 latency: 21793603583 ns
p99.9 latency: 21793603583 ns
p99.99 latency: 21793603583 ns
p99.999 latency: 21793603583 ns
Max latency: 21793603583 ns
```

5.2.2 Latency for Indexed SQLite. Here are experimental results for latency, which we provide detailed statistics on:

```
Latency performance results for Indexed SQLite
    ↪ on Small Data in nanoseconds (ns):

hildren queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 12120.247920000003 ns
Std deviation: 3942.700369997151 ns
Min latency: 7200 ns
p25 latency: 10703 ns
p50 latency: 11503 ns
p75 latency: 13303 ns
p90 latency: 13903 ns
p95 latency: 14503 ns
p99 latency: 15703 ns
p99.9 latency: 64415 ns
p99.99 latency: 176383 ns
p99.999 latency: 246015 ns
Max latency: 246271 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 10009.523639999996 ns
Std deviation: 2752.7551229052597 ns
Min latency: 5900 ns
p25 latency: 8207 ns
p50 latency: 10407 ns
p75 latency: 10807 ns
p90 latency: 11303 ns
p95 latency: 12007 ns
p99 latency: 16103 ns
p99.9 latency: 30015 ns
p99.99 latency: 95743 ns
p99.999 latency: 208127 ns
Max latency: 283391 ns
```

```
Latency performance results for Indexed SQLite
    ↪ on Medium Data in nanoseconds (ns):

Children queries latency test...
```

```
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 36065.73392000003 ns
Std deviation: 63185.21519342133 ns
Min latency: 16400 ns
p25 latency: 29311 ns
p50 latency: 32703 ns
p75 latency: 36127 ns
p90 latency: 39711 ns
p95 latency: 42815 ns
p99 latency: 95935 ns
p99.9 latency: 704511 ns
p99.99 latency: 2003967 ns
p99.999 latency: 8904703 ns
Max latency: 8962047 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 26951.480319999984 ns
Std deviation: 7850.660606680986 ns
Min latency: 9200 ns
p25 latency: 23407 ns
p50 latency: 25615 ns
p75 latency: 28415 ns
p90 latency: 32207 ns
p95 latency: 36031 ns
p99 latency: 55807 ns
p99.9 latency: 105023 ns
p99.99 latency: 213887 ns
p99.999 latency: 316927 ns
Max latency: 349951 ns
```

```
Latency performance results for Indexed SQLite
    ↪ on Big Data in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 27164.44360000002 ns
Std deviation: 83933.05336479556 ns
Min latency: 13400 ns
p25 latency: 21311 ns
p50 latency: 23215 ns
p75 latency: 25007 ns
p90 latency: 27615 ns
p95 latency: 31615 ns
p99 latency: 76927 ns
p99.9 latency: 675839 ns
p99.99 latency: 4263935 ns
p99.999 latency: 8847359 ns
Max latency: 10289151 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 22164.992080000076 ns
Std deviation: 19676.835687195668 ns
Min latency: 8896 ns
p25 latency: 17407 ns
p50 latency: 18415 ns
p75 latency: 20415 ns
p90 latency: 24415 ns
p95 latency: 30911 ns
p99 latency: 149503 ns
p99.9 latency: 246143 ns
p99.99 latency: 314623 ns
p99.999 latency: 384511 ns
Max latency: 446207 ns
```

5.2.3 Latency for Single Worker Distributed Search Engine.

Here are experimental results for latency, which we provide detailed statistics on:

```
Latency performance results for Single Worker
    ↪ Distributed Search Engine on Small Data
    ↪ in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 686646.699520001 ns
Std deviation: 207063.29275492308 ns
Min latency: 610816 ns
p25 latency: 640511 ns
p50 latency: 650239 ns
p75 latency: 675327 ns
p90 latency: 732671 ns
p95 latency: 792575 ns
p99 latency: 1282047 ns
p99.9 latency: 3715071 ns
p99.99 latency: 6115327 ns
p99.999 latency: 10403839 ns
Max latency: 13262847 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 693936.0307200011 ns
Std deviation: 245353.89718492367 ns
Min latency: 605696 ns
p25 latency: 641023 ns
p50 latency: 652287 ns
p75 latency: 674303 ns
p90 latency: 722431 ns
p95 latency: 836607 ns
p99 latency: 1482751 ns
```

```
p99.9 latency: 4276223 ns
p99.99 latency: 7368703 ns
p99.999 latency: 9428991 ns
Max latency: 12410879 ns
```

```
Latency performance results for Single Worker
    ↪ Distributed Search Engine on Medium Data
    ↪  in nanoseconds (ns):

Children queries throughput test...
Children queries throughput test with 100000
    ↪ iterations took: 70.179563266s

Children queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 692554.5625600005 ns
Std deviation: 165571.6104586851 ns
Min latency: 614912 ns
p25 latency: 648703 ns
p50 latency: 659967 ns
p75 latency: 692223 ns
p90 latency: 751615 ns
p95 latency: 803839 ns
p99 latency: 1125375 ns
p99.9 latency: 2975743 ns
p99.99 latency: 5689343 ns
p99.999 latency: 9961471 ns
Max latency: 10977279 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 689333.754880001 ns
Std deviation: 191730.92315412973 ns
Min latency: 605696 ns
p25 latency: 645119 ns
p50 latency: 655871 ns
p75 latency: 685567 ns
p90 latency: 747007 ns
p95 latency: 799231 ns
p99 latency: 1093631 ns
p99.9 latency: 3254271 ns
p99.99 latency: 7778303 ns
p99.999 latency: 10305535 ns
Max latency: 13123583 ns
```

We do not have latency performance results for Single Worker Distributed Search Engine on Big Data because that data size was too large to generate results, as explained later.

5.2.4 Latency for Three Workers Distributed Search Engine. Here are experimental results for latency, which we provide detailed statistics on:

```
Latency performance results for Three Workers
    ↪ Distributed Search Engine on Small Data
    ↪ in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 580845.1532800011 ns
Std deviation: 109200.23061321232 ns
Min latency: 462080 ns
p25 latency: 506879 ns
p50 latency: 565759 ns
p75 latency: 641023 ns
p90 latency: 694783 ns
p95 latency: 726527 ns
p99 latency: 798719 ns
p99.9 latency: 1723391 ns
p99.99 latency: 3592191 ns
p99.999 latency: 5476351 ns
Max latency: 5980159 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 571316.8691199998 ns
Std deviation: 114293.75037534042 ns
Min latency: 457728 ns
p25 latency: 503551 ns
p50 latency: 546303 ns
p75 latency: 624639 ns
p90 latency: 675327 ns
p95 latency: 713215 ns
p99 latency: 772095 ns
p99.9 latency: 1915903 ns
p99.99 latency: 3995647 ns
p99.999 latency: 7053311 ns
Max latency: 7127039 ns
```

```
Latency performance results for Three Workers
    ↪ Distributed Search Engine on Medium Data
    ↪  in nanoseconds (ns):

Children queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 587215.9923200002 ns
Std deviation: 156249.49922683716 ns
Min latency: 460544 ns
p25 latency: 506111 ns
p50 latency: 565759 ns
p75 latency: 645119 ns
p90 latency: 706047 ns
p95 latency: 735231 ns
```

```
p99 latency: 878079 ns
p99.9 latency: 2392063 ns
p99.99 latency: 6385663 ns
p99.999 latency: 8486911 ns
Max latency: 8994815 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 581139.9654399989 ns
Std deviation: 148069.34780097738 ns
Min latency: 463872 ns
p25 latency: 505343 ns
p50 latency: 555007 ns
p75 latency: 639487 ns
p90 latency: 698879 ns
p95 latency: 731135 ns
p99 latency: 821759 ns
p99.9 latency: 2020351 ns
p99.99 latency: 6959103 ns
p99.999 latency: 9379839 ns
Max latency: 9388031 ns
```

```
Latency performance results for Three Workers
    ↪ Distributed Search Engine on Big Data in
    ↪  nanoseconds (ns):

Children queries latency test...
Children queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 555345.8777600001 ns
Std deviation: 151979.4208317316 ns
Min latency: 451840 ns
p25 latency: 485887 ns
p50 latency: 512255 ns
p75 latency: 594431 ns
p90 latency: 666111 ns
p95 latency: 705023 ns
p99 latency: 809983 ns
p99.9 latency: 2134015 ns
p99.99 latency: 6205439 ns
p99.999 latency: 9576447 ns
Max latency: 9879551 ns

parents queries latency test...
parents queries latency test with 100000
    ↪ iterations complete. Statistics:
Mean latency: 544409.8367999993 ns
Std deviation: 129306.43690255893 ns
Min latency: 451584 ns
p25 latency: 484351 ns
p50 latency: 502527 ns
p75 latency: 583679 ns
p90 latency: 646143 ns
p95 latency: 680959 ns
p99 latency: 776703 ns
p99.9 latency: 2101247 ns
p99.99 latency: 5406719 ns
p99.999 latency: 6750207 ns
Max latency: 8269823 ns
```

5.3 Interpretation and Summary

Overall, we are quite happy with the performance of our system, especially given the constraints it was built under. There is plenty of room for future improvement, but we have created a good base and measured a lot of performance results.

One observation that brings us joy is that our system is quite responsive (i.e. has low latencies overall). We accomplished our aim of building a distributed system for searching Bitcoin data with latencies that support interactively searching through the data. Interestingly, but not surprisingly, our system's latency does not increase as the amount of data increases. This tells us two things:

1. For the types of queries we are running, our indexing strategy is pretty good. In particular, each worker node only needs to perform two binary searches to determine all the children or parent transactions of a query hash. Therefore, it would take massive amounts of data on each worker before the actual search became slow due to the $O(\log n)$ runtime of binary search.
2. Our data is simply too small to be brushing up against the limits of our distributed system.

Our system was actually limited by our resource consumption on each individual node; we were unable to run the single-worker system on the Large dataset (around 30GB in size before indexing) because the node was running out of memory while loading all the indexed data, even with 25GB of additional swap. In theory, we could have designed our worker nodes to more effectively use their memory (with `mmap`'d files), but we would have then run up against the memory limits of the sort steps in our parser. We also could implement on-disk sorting of data, which remains future work.

Furthermore, we noticed that there's also an incredibly high variance for our sampled data, both on a single worker and with multiple workers. SQLite is much more consistent in its latencies, even when compared with our system running with a single worker. While we were initially stumped by this observation, we realized that this is because the master and worker live on separate machines, even with a single worker. Therefore, even with just a single worker, our RPC mechanism must still make network round-trips, and our testing VMs have very poor networking, leading to high variance in our latencies.

Overall, our system performed far better than unindexed SQLite, especially for the medium and large data, but slightly slower than indexed SQLite on a single machine. The SQLite implementation also has the advantage of being able to store more data than there is system memory available. However, our (far) better performance than unindexed SQLite, even including network round-trips, indicates that our indexing strategy was effective. In fact, comparing the sizes of the indexed and unindexed SQLite databases with our doubly-sorted and unsorted custom data formats leads us to believe that SQLite is doing the same indexing as we are via sorting under the hood. However, SQLite has the advantage of not paying the cost of the network roundtrip.

6 Discussion and Conclusion

From working on this, we made several realizations as a group. First, we discovered that sometimes it is easier and better to write one's own tool than to find an existing tool online. This was the case for our Bitcoin transaction data parser: at first, we scoured the internet looking for an existing parser, since surely there had to be one we could use, considering how popular Bitcoin is and how long Bitcoin has been used for. However, what we did not realize at the time was that, although Bitcoin has been around for a while, Bitcoin has also been updated, so existing parsers might have been (and indeed were) outdated. Plus, we had a relatively specific use case in mind since we wanted to parse the data directly into either a database or into a custom data format for our distributed search engine, in which case we would have had to modify an existing parser quite significantly anyways. So, we ended up creating our own parser, which ended up not being as hard as we expected, providing a good lesson for us when we want specific tools in the future.

Using Rust was an *interesting* experience. We learned a lot and grew as programmers, but also had our fair share of frustration along the way and lost many hours to fights with the compiler. However, the elegance of the language and our complete lack of crashes and non-deterministic bugs gives us a strong reason to return to the language in the future.

Comparing our results with that of SQLite also tells us a few things — a piece of well-optimized software running on a single machine can be profoundly powerful, and in fact better, than a distributed system for a huge number of use cases, with surprisingly large amounts of data. But using the tool incorrectly (for example, by failing to index for the most common queries) can lead to abysmal (and in fact, unusable) levels of performance.

Another takeaway we learned is that building a distributed system is really complex and difficult and can result in frustrating, confusing bugs. Because we were simulating the distributed system using multiple Azure VMs, we also had to learn how to use Azure and communicate between the VMs, which was quite complicated. Additionally there would sometimes be strange issues where our code would work locally but not when uploaded to and run on Azure. These experiences reinforced the idea that if it is not necessary to use a distributed system, one should generally avoid using one because of the extra overhead complexity involved in building, debugging, and maintaining it. In our case, for this project specifically, we definitely wanted to build out the distributed system because we wanted to test how a distributed search engine would fare against our baselines, but for future projects, now we better know how important it is to think through and debate whether it is truly worth it to use a distributed system.

7 Acknowledgments

We would like to thank Prof. Eddie Kohler for extensive mentorship on our project, as well as financial, moral, and intellectual support.

References

[1] 2021. Transaction. https://en.bitcoin.it/wiki/Transaction
[2] Ricardo Baeza-Yates. 2010. Towards a distributed search engine. In *International Conference on Algorithms and Complexity*. Springer, 1–5.
[3] John Biggs, Hoa Nguyen, and Noelle Acheson. 2013. How do bitcoin transactions work? https://www.coindesk.com/learn/how-do-bitcoin-transactions-work-2/
[4] JOHN BOHANNON. 2016. Why criminals can't hide behind Bitcoin. https://www.science.org/content/article/why-criminals-cant-hide-behind-bitcoin
[5] Stefan Buttcher, Charles LA Clarke, and Gordon V Cormack. 2016. *Information retrieval: Implementing and evaluating search engines*. Mit Press.
[6] Mauro Conti, Ankit Gangwal, and Sushmita Ruj. 2018. On the economic significance of ransomware campaigns: A Bitcoin transactions perspective. *Computers & Security* 79 (2018), 162–189.
[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[8] Joan Antoni Donet Donet, Cristina Pérez-Sola, and Jordi Herrera-Joancomartí. 2014. The bitcoin P2P network. In *International conference on financial cryptography and data security*. Springer, 87–102.
[9] Zhipeng Gui, Chaowei Yang, Jizhe Xia, Kai Liu, Chen Xu, Jing Li, and Peter Lostritto. 2013. A performance, semantic and service quality-enhanced distributed search engine for improving geospatial resource discovery. *International Journal of Geographical Information Science* 27, 6 (2013), 1109–1132.
[10] Amr Z Kronfol. 2002. *FASD: A fault-tolerant, adaptive scalable distributed search engine*. Ph. D. Dissertation. Master's Thesis http://www.cs. princeton. edu/akronfol/fasd.
[11] Malte Moser. 2013. Anonymity of bitcoin transactions. (2013).
[12] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.
[13] Rhyme Upadhyaya and Aruna Jain. 2016. Cyber ethics and cyber crime: A deep dwelved study into legality, ransomware, underground web and bitcoin wallet. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 143–148.
[14] Jiangong Zhang and Torsten Suel. 2007. Optimized inverted list assignment in distributed search engine architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–10.
[15] Chen Zhao and Yong Guan. 2015. A graph-based investigation of bitcoin transactions. In *IFIP International Conference on Digital Forensics*. Springer, 79–95.