CryptoSql: Exploring viability of blockchain-based databases

Compatibility layer between MySQL and Ethereum

Dillon Newell

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering, University of Toronto, Toronto, Ontario, M5S 3G4

Abstract

Most companies employ some sort of relational database that interacts with front-end and back-end applications. In this article we explore the viability of storing databases on a blockchain instead of locally on-disk or with a cloud provider. Since the consuming applications can be highly specific for the choice of database, we introduce CryptoSQL as a compatibility layer that accepts connections from MySQL clients and stores the data on the Ethereum blockchain. We show the first implementation is viable under some niche scenarios and suggests exciting future results with future implementations.

1. Introduction

Relational Database Management Systems (RDBMS) have become widespread across almost every industry. There are many varieties of such systems, but they generally share characteristics such as SQL-based interaction, disk storage backing, and memory caching for performance.

There are currently two main ways to use an RDBMS server: on a dedicated local server, or through a cloud provider such as Amazon or Microsoft. These methods have their drawbacks:

- Local storage is susceptible to corruption and/or loss if the database is not properly replicated. This places a responsibility on the company to add multiple disks for backup, and in large scales requires geographical redundancy. With the amount of hardware required this can require a large initial investment.
- Cloud storage solves most of the problems with local storage. The provider manages redundancy and provides options to have the data stored many places around the world simultaneously. However, even though there is no up-front hardware cost, the provider charges fees on a subscription and/or per-use basis. It also requires the company place trust in the provider.

Both methods have their downsides, and the right one to use depends on many factors. However, some of the problems bear resemblance to the problems in currencies that the Bitcoin blockchain was introduced to solve. The following are some solutions that a blockchain such as Ethereum would have:

- Blockchains are generally designed to be untrusted, so no trust needs to be placed in a third party with storing your data
- Blockchains are replicated around the world by thousands of different nodes
- Data stored on the blockchain is permanently stored and easily retrieved
- There is no requirement to have the data permanently stored on replicated disks

This suggests that an alternative RDBMS which uses a blockchain as its storage could be a useful solution for some companies that have priority with these problems. However, applications that consume the database are frequently highly specific for their RDBMS interface. One of the most popular solutions is MySQL, an open-source RDBMS used by high-profile companies such as Facebook [1] and Twitter [2]. Therefore, in the following we propose CryptoSQL, a blockchain-backed database that accepts native MySql client connections and queries.

2 Comparisons and expectations

Expanding on the problems listed above, discussed in this section are key metrics that could influence a company's decision on which database solution to use, and the expected results after analysis. Cost and performance are measured empirically in Section 4, and the rest are discussed in Section 5.

2.1 Performance

Many usage scenarios require high-performance databases. Traditional RDBMSs usually store data on a disk as soon as it is received [3], and cache it in memory for high reading speed [4]. There is also a considerable amount of optimization done within the server.

We define two categories for a database's performance. Online performance is the read and write speeds that require accessing the storage backing, i.e. the disk or blockchain. Offline performance consists of the operations that do not require access to the backing. Generally loading and committing databases are online, and everything else is offline.

Both offline and online performances are expected to be the best with local servers. Cloud servers will have slightly higher offline performance (note "offline" operations are still online as they are sent to the provider), and similar online performance (note loading/committing databases happens internally at the provider). Blockchain backing can potentially have similar offline performance to local servers but exceptionally higher online. Loading data requires making a request to a node and getting transaction data and committing requires sending a transaction that has to be mined.

2.2 Cost

Cost is also a very important metric for companies. We split cost into three categories: up-front, subscription, and per-use. Subscription refers to over-time payments required to keep the server running and per-use is the cost of updating/retrieving data.

Local servers require by far the most up-front costs, with cloud requiring none. CryptoSQL requires a locally running server to translate MySQL commands and cache data, so some hardware is required (but not more, since replication is done by the blockchain. Subscription costs are very high for cloud providers and close to negligible for the others. In this context it would refer to the electricity costs associated with running the hardware. Per-use cost is negligible with local servers and high for the cloud. Blockchain usage costs are high for writing data (requires a transaction to be mined) but zero for retrieval.

2.3 Integrity

The first of the theoretically analyzed metrics is integrity, which refers to how well the data is expected to persist without loss or alteration. Integrity is expected to be low on local servers, but with the possibility for upscaling based on company investments. Cloud storage has very high integrity, but if the company does not trust the provider to manage their data this could lower. Blockchain has excellent integrity that only depends on it still having full nodes running.

2.4 Availability

Availability is how infrequently the data becomes temporarily unavailable, e.g. the server goes down. The results expected are in line with those of integrity and for the same reasons.

2.5 Privacy

Privacy is expected to be highest on local servers, where the company has full control. Cloud privacy depends on provider trust. Blockchain privacy does not require trust, but data stored is accessible by anyone. However, with proper encryption this could reach levels of local storage.

To summarize, blockchain storage is expected to be a viable option for companies that don't have the means to replicate their data around the globe or need to commit data often, but do need high integrity and availability.

3 Methods

Specifications on hardware and software is listed in Appendix 1. Amazon's RDS was used for the cloud provider, with instance type db.t2.small.

3.1 Server software

CryptoSQL works by running a server on a local machine that can accept MySQL queries and respond to them by interacting with the blockchain. The server does not need to run on the same machine running the application, nor even the same network.

The software is programmed in C# language using the .NET Core runtime, which is a high performing solution that can run on all major operating systems [5]. It is organized into three main projects: CryptoSql.MySql accepts client connections and converts them to a neutral form, CryptoSql.Blockchain handles connections to the blockchain, and CryptoSql is the executable project that bridges these two. The blockchain project is currently implemented as CryptoSql.Blockchain.Ethereum, but in the future other implementations could be added to test other blockchains. CryptoSQL was designed to be expandable, so addition of other blockchains and RDBMS interfaces can be made quickly.

Prior to launching the server, a geth Ethereum client is opened. CryptoSQL launches and automatically connects to geth, then listens for incoming MySQL connections.

More details on CryptoSQL internals can be found in Appendix 2.1.

3.2 Testing methodology

3.2.1 Performance

Performance is mainly measured with timed operations. Exact commands issued are referenced in Appendix 2.2.

Using a command-line MySQL client (which reports durations of operations), timings were recorded for multiple queries. To start offline performance, queries were made to add 600 rows into a 5-column table (200 at a time). Then operations that require the database to search and update the rows were timed. Finally, the time to empty a table is recorded.

The columns consist of two integers and three strings. To populate the data, random names were generated and used for two string columns, random addresses were used for the third, and random integers on the range $(0,65534)$ for the integer columns.

For online performance, time to commit and load data from a cold boot are recorded. Load data is only recorded for the blockchain backing since the MySQL client does not report timings.

3.2.1 Cost

For local servers, upfront is calculated as the hardware purchase price. Usage is calculated using the decrease on the drive's lifespan, however negligible. Subscription is the electricity costs associated with total hardware. Exact calculations are shown in Appendix 2.3.1.

For cloud servers, upfront is 0, and usage and subscriptions are read from the RDS pricing agreements (see Appendix 2.3.2).

For blockchain, upfront is again the cost of hardware, usage is the blockchain committing costs, and subscription is electricity (see Appendix 2.3.3).

4 Empirical results

For both sections, the resources required are recorded as follows:

- geth: 800MB of RAM, little to no CPU after starting up
- MySQL server: 50MB RAM, CPU spikes when processing queries
- CryptoSQL: 300MB RAM, CPU spikes when processing queries
    - 95% of the memory is boilerplate due to the .NET Core runtime, the usage change is hardly noticeable during operation

4.1 Performance

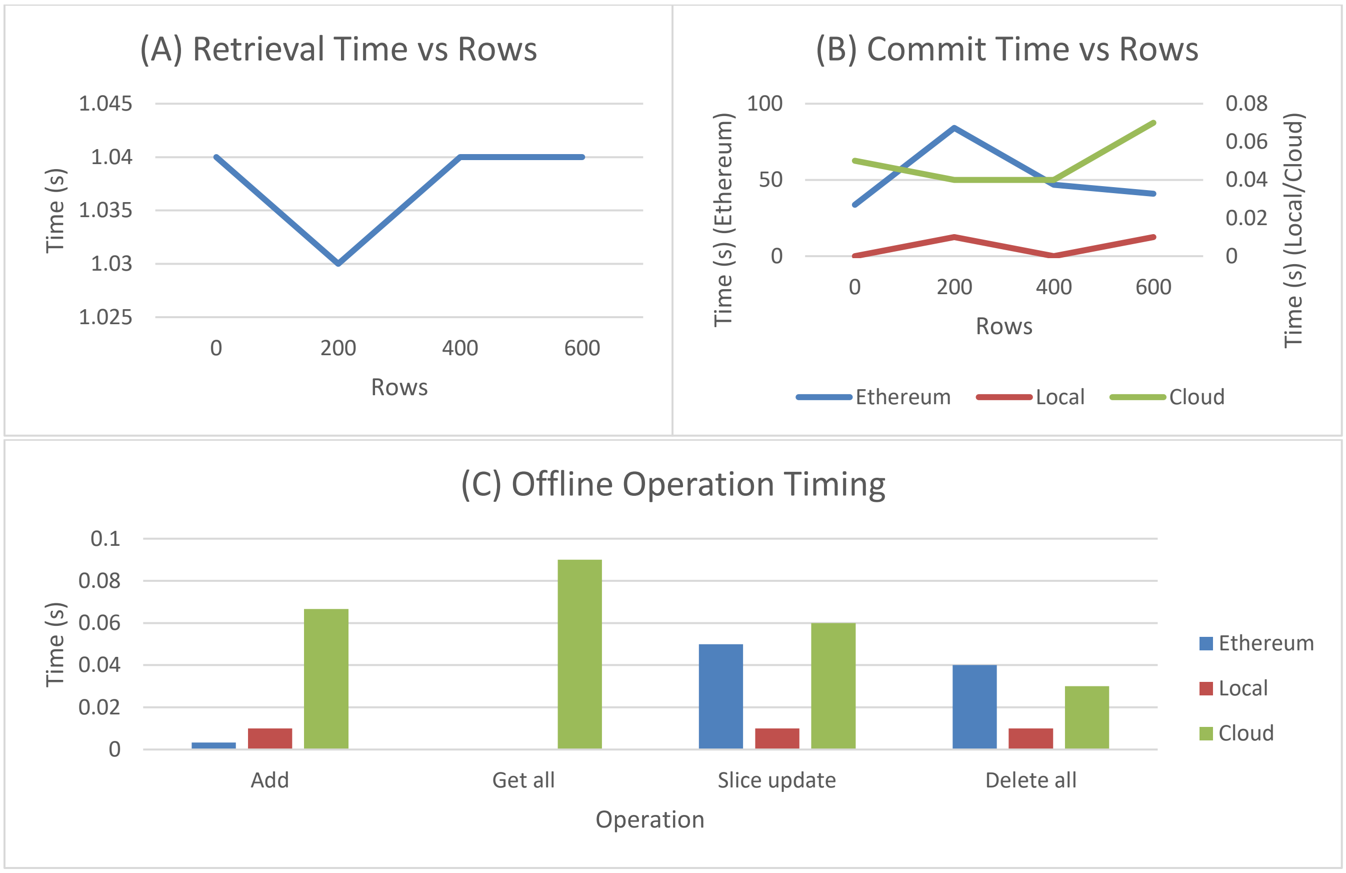

Figure 1: Recorded metrics of performance. (A) The time CryptoSQL takes to read data from the blockchain compared to row count. (B) The time the servers take to commit data compared to row count. Note that Ethereum storage is on the left axis and local/cloud on the right for visibility. (C) Comparison of various offline operation timings.

The results are generally in line with the expectations. Online operations are exceptionally slow on CryptoSQL, beaten by local/cloud by up to a factor of 10^4 for committing data. Reading data is much better, averaging around 1.04s regardless of row count. If the Ethereum client already contains the transaction data needed, that drops down to about the same speed as the local server (however this is infrequent, and usually only after it has been loaded recently). Interestingly, neither reading nor writing data appears to correlate with row count, suggesting that the overhead is mainly from the blockchain pool priority and not amount of data processed. At 600 rows we are very near to the maximum allowed per-transaction value.

Offline operation for CryptoSQL is also admirable. Get all rows and add rows align with local server performance. As expected, the cloud server has the highest offline operation times because the data must transmit over the Internet. However, CryptoSQL does sub-par on slice updating and delete all. This is likely due to the heavy optimization MySQL has done internally over the years. Given enough time, it is conceivable CryptoSQL could be optimized to handle the queries much faster.

These results indicate that CryptoSQL can only be a viable database solution for companies that do not need to commit data frequently.

4.2 Costs

In the costs section, we have provided an alternative low-spec price point for hardware needed to run the server. These specs are listed in Appendix 2.3.1 and correspond closer to what the tested cloud tier is using. There is also a higher cloud tier for one test, that closer aligns to the tested server hardware (listed in Appendix 2.3.2).

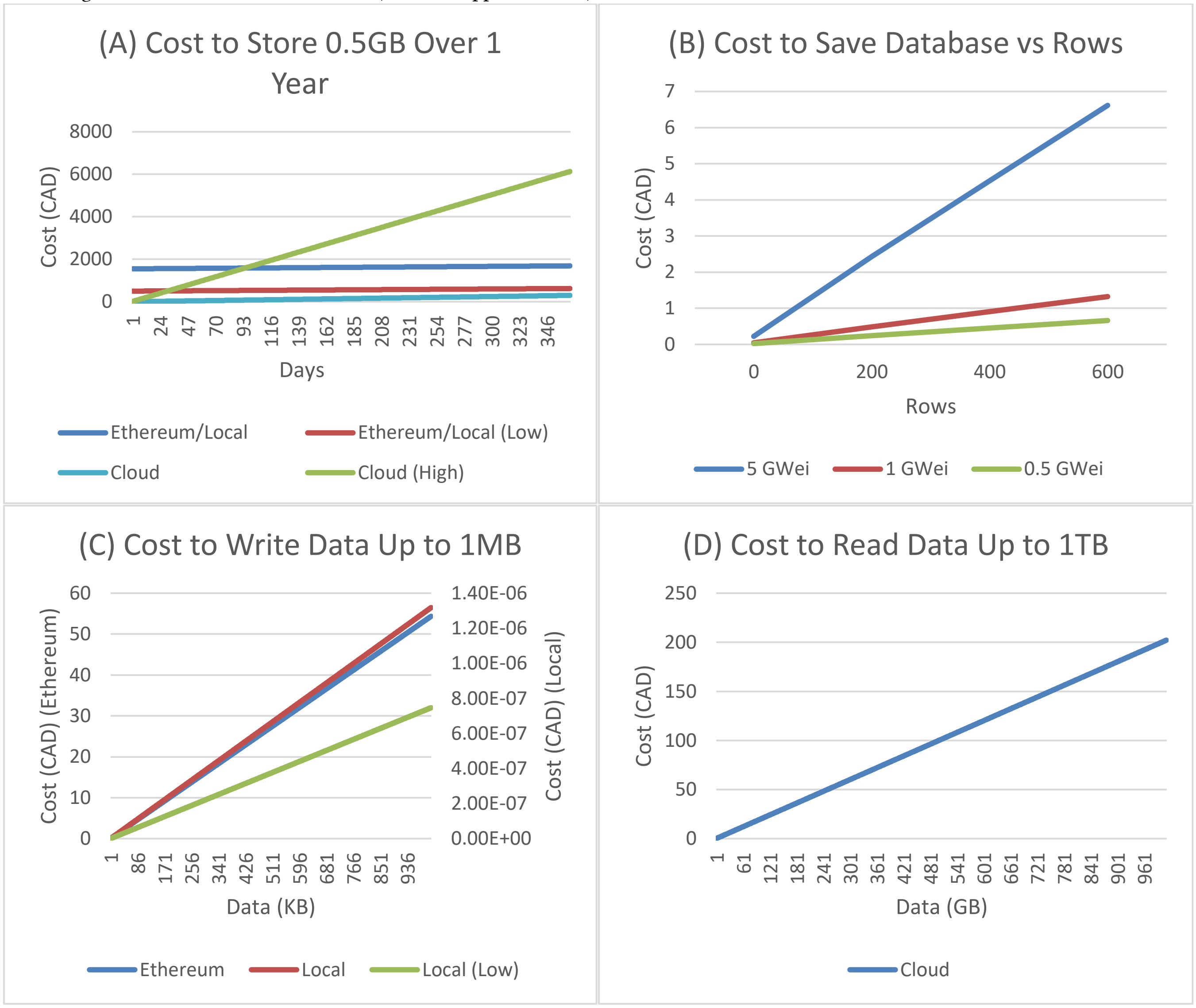

Figure 2: Analysis of cost metrics. (A) The cost to consecutively leave 0.5GB of data in the database over a year. (B) The cost in GWei required to save to the blockchain based on number of rows. (C) The cost to write data to the server, from 1KB to 1024KB. Note cloud is missing because writing data is free up to 10TB, and Ethereum is on a different axis than locals to increase visibility. Further note the cost is the same for Ethereum no matter which computer is used (D) Cost to read data up to 1TB. Note cloud is the only server that has a cost for reading data.

The cost to keep stored data is a major bonus for blockchain-based storage. There are no perpetual fees like with cloud providers, but there is still the resiliency of the blockchain. When using the higher tier Amazon RDS instance, the hardware investment of the high-end CryptoSQL setup is cheaper after around 3 months. The lower tier RDS instance passes the low tier hardware at about 450 days. The blockchain and local servers can also be turned off, so electricity is not charged if the data is not needed for times. This cannot be done on the cloud.

The cost to save the database to the blockchain is a linear function to the data size. If data is repeated often, compression can help mitigate the costs. However, since both the Ethereum price and minimum gas level fluctuate often, it can be problematic to the point where companies are deciding the best times to commit data.

As expected, the cost to commit data to the blockchain is enormous compared to local and cloud. Note while the Ethereum and local lines align, Ethereum is on a scale ~10^7 the magnitude. The cost to store even just 1MB gets prohibitive. Writing to the cloud is free up to 10TB.

Another thing to note for these costs is that this assumes no undertaken effort for replication and/or redundancy on the local server.

Finally, reading data is only costly on the cloud. Focusing on how the blockchain reading costs scale upward is important for the high-read, low-write usage niche CryptoSQL is looking to fill. All these results corroborate that a database which is committed once and read frequently may be better off on the blockchain.

5 Theoretical analysis

5.1 Integrity

Arguably the most important of the three theoretical analyses, the need for integrity can vary greatly by company. Consider a company that re-populates their database every day, such as a maintained list of active users. Loss of data will not be a problem because it was on its way to being replaced. However, some companies require serious integrity, such as banks holding client records. To these companies, data replication is at the top of their list.

As explained in the introduction, local storage has its problems with integrity unless you invest heavily in hardware around the world. Cloud providers can abstract this, but charge a hefty premium for redundancy (the subscription costs in section 4.2 are triple if using a multi-region database). Blockchains solve this by incentivizing thousands of people around the world to maintain it. For as long as users are incentivized to maintain the chain, and it is safe from attacks, there is no risk of data loss. Furthermore, if the blockchain does decline it would be a slow process, likely giving enough time to pull the data out. This contrasts with the other solutions, where data loss is usually sudden.

We conclude that a blockchain provides a best-case scenario for integrity, provided the company pays attention to the chain's status.

5.2 Availability

Availability goes close in hand with integrity. Local servers that crash or lose power can take the database offline completely if the company has not setup distributed computing properly. Cloud providers are designed for high availability, but occasionally can go offline and cause massive outages across the Internet.

Blockchain availability is again based on the number of nodes running. If you can connect to a node, you can get a list of previous transactions. There are even services such as Etherscan [7] that allow you to access transactions without setting up a client.

The conclusion is the same, a blockchain provides the best availability provided the company pays attention to it.

5.3 Privacy

Privacy is a big problem on the blockchain, as everything on it is publicly accessible. Users can combat this with anonymity, however a company storing data in plaintext could be susceptible to any number of bad actors scraping for it. This is not a problem at all with local servers, nor cloud servers if you can trust the provider.

The only real answer to keeping a database private on the blockchain is to encrypt it. That comes with its own set of problems however, as every encryption is susceptible to being broken at some point. While standards like AES-256 are certainly adequate for today, every transaction made on the blockchain is there forever. If at some point down the line AES-256 is vulnerable, changing to a new standard does not protect the old records.

Other methods can help mitigate privacy concerns, such as cycling through many different wallet addresses and using different cyphers for every transaction. However, these have their own drawbacks, particularly they could cause a loss of the database if the host computer is corrupted and no backups were made.

In conclusion, blockchain storage has the worst privacy concerns out of the three. Therefore, it is not a viable option for companies that require their data be kept secret. However, for non-private storage, the public accessibility of the blockchain has some benefits, discussed in Section 7.

6 Limitations

- Databases are limited to 32KB. Even though the hard limit on transaction data is 100KB, the default Ethereum client geth does not accept higher than 32KB [6], and so most nodes will reject the transaction. While 32KB can comfortably fit a 5x600 table, usages may require many thousands of rows, so this is not adequate. However, it can be fixed by switching from the monolithic provider to a more flexible one, described in Section 7. This is why CryptoSQL was designed to be expandable, it was unlikely that the first solution would be the best. New providers can be added in with much less effort now.
- Running CryptoSQL requires an Ethereum client. A more adaptable solution could allow read-only access without the need for a client, described in Section 7. This way light clients such as mobile phones could consume databases.
- CryptoSQL only supports a small subset of MySQL commands. Only those needed for this research were included.
- CryptoSQL still requires some amount of local storage. Namely, the transactions to the databases need to be stored. This does not add a huge risk for data loss however, since one could look through their account transactions on Etherscan [7] to find the hash of their last transaction. The whole database can be accessed from any computer provided the transaction is input to the config file.

- Future CryptoSQL implementations may have similar RAM requirements to local servers to maintain peak performance, although they do not require the same redundancy or disk requirements.

8 Future work

- Support arbitrarily sized databases. One way to solve this would be to have a distributed database that is contained over many transactions. This is an idea I wanted to implement but could not figure out a reliable way until late in the project. The proposed implementation is described in Appendix 3.1.
- There is much optimization that can be done internally. Instead of converting queries to objects to have them converted to different objects, deferred execution can be used.
- Supporting more MySQL commands can make integration easier.
- Support for more data types.
- Explore contract storage.
- A "light" version could be made specifically for client apps to consume. Typically, client apps get their data from company servers, and those servers do all the interface with the database. However, the organization of the blockchain could allow a client access only to their own data directly from their device (even a phone) without the need for an Ethereum client. Transactions can be retrieved from third-party block explorers such as Etherscan [7]. This saves the company from having to process as many requests.

9 Conclusion

A solution such as CryptoSQL is certainly niche, but it does have its usage scenarios. If a company is some of the following, then it is likely a good fit:

- Does not need to write data often
- Wants the data to be highly available and read from any computer around the globe with just a transaction hash
- Has a need for high integrity and/or availability
- Does not need privacy

Situations of read-only data could include postings by companies that hang around for a couple of weeks. For example, a weekly or monthly challenge board posted for a video game, which could fit in the couple hundred lines the implementation supports. With a client-side reader, the data could be read off the blockchain without contacting company servers.

The early results look good, but more importantly signify potential of future work on the project. Many of the limitations can be tightened up and the results will likely get better.

References

[1] Om Malik. 2008. Facebook's Insatiable Hunger for Hardware. Retrieved April 15, 2018 from https://gigaom.com/2008/04/25/facebooks-insatiable-hunger-for-hardware/.
[2] Jeremy Cole. 2011. Big and Small Data at @Twitter. Video. Retrieved April 15, 2018 from https://www.youtube.com/watch?v=5cKTP36HVgI.
[3] MySQL. 2018. START TRANSACTION, COMMIT, and ROLLBACK Syntax. In MySQL 5.7 Reference Manual. 13.3.1. Retrieved April 15, 2018 from https://dev.mysql.com/doc/refman/5.7/en/commit.html.
[4] MySQL. 2018. The InnoDB Buffer Pool. In MySQL 5.7 Reference Manual. 14.9.2.1. Retrieved April 15, 2018 from https://dev.mysql.com/doc/refman/5.5/en/innodb-buffer-pool.html.
[5] Microsoft. 2016. .NET Core Guide. Retrieved April 15, 2018 from https://docs.microsoft.com/en-us/dotnet/core/index.
[6] Ethereum. 2018. tx_pool.go. In go-ethereum. Retrieved April 15, 2018 from https://github.com/ethereum/go-ethereum/blob/ba1030b6b84f810c04a82221a1b1c0a3dbf499a8/core/tx_pool.go#L557.
[7] Etherscan. 2018. The Ethereum Block Explorer. Retrieved April 15, 2018 from https://etherscan.io/.
[8] Microsoft. 2016. Windows Subsystem for Linux Overview. Retrieved April 16, 2019 from https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/.
[9] Microsoft. 2007. DataSet Class. Retrieved from https://msdn.microsoft.com/en-us/library/system.data.dataset(v=vs.110).aspx.
[10] Nethereum. 2018. Nethereum. Retrieved from http://nethereum.com/.
[11] Dillon Newell. 2018. CryptoSQL. Retrieved from https://github.com/DillonN/CryptoSQL.
[12] PCPartPicker. 2018. PCPartPicker. Retrieved from https://ca.pcpartpicker.com/.
[13] Toronto Sun. 2016. Higher electricity rates kick in for Ontario. Retrieved from http://torontosun.com/2016/05/01/higher-electricity-rates-kick-in-for-ontario/wcm/c38aa84c-46b8-4edc-b5c1-ec39802d6f1f.
[14] Amazon. 2018. Amazon RDS for MySQL Pricing. Retrieved from https://aws.amazon.com/rds/mysql/pricing/.
[15] Gentelella. 2018. ETH Gas Station. Retrieved from https://www.ethgasstation.info/.
[16] CoinGecko. 2018. Ethereum Price Chat (ETH/CAD). Retrieved from https://www.coingecko.com/en/price_charts/ethereum/cad.
[17] Dr. Gavin Wood. 2018. Ethereum: A Secure Decentralised Generalised Transaction Ledger Byzantium Version c0c3b5d. Retrieved from https://ethereum.github.io/yellowpaper/paper.pdf.

Appendix 1 – Specifications

Appendix 1.1 – Hardware

All tests were performed on a desktop with the following relevant specifications:

- Intel Core i7-8700k @5.0GHz (6 cores/12 threads)
- 32GB DDR4 RAM @2400MHz
- 500GB Samsung 960 Evo M.2 NVMe SSD
- Gigabyte Aorus Gaming 7 Motherboard

Appendix 1.2 – Software

- Windows 10 x64 Education build 1803
    - Visual Studio 2017.5 IDE for development
        - Runs CryptoSQL in debug mode
        - C# 7.2
        - .NET Core 2.0
- Ubuntu 16.04.3 LTS running on Windows Subsystem for Linux (lxss)
    - MariaDB for MySQL server and client, version 10.0.34
    - geth version 1.8.4

The Ubuntu lxss is not a virtual machine, it runs natively through the Windows kernel. Performance is higher than a VM, and at or approaching native [8].

Appendix 2 – Methods specifics

Appendix 2.1 – CryptoSQL internals

The bridge program starts by initializing both the MySQL connector and Ethereum provider. Then, the MySQL connector is told to start listening for connections over a TCP port. When a connection is made and queries are sent, the MySQL project converts it into a neutral language and sends it to the bridge, which then calls the correct function of the provider.

The reason for conversion to a neutral language is so that other RDBMSs can be supported in the future. The bridge also maintains a list of providers, and forwards commands to every one of them. That way if other providers are added, they can be tested without having to issue commands multiple times.

The monolithic provider holds all data in a System.Data.DataSet object, a built in class to .NET Core that handles an in-memory database [9]. This means we get existing optimization and features. It supports serialization, so when the time comes to commit the database, it is compressed with gzip to a byte array and sent in an Ethereum transaction.

Sending the transaction blocks the client until it is mined and a receipt is returned. The receipt includes the transaction hash, which is the new location for that database. CryptoSQL saves the hash to its config file, so that on next launch when a user selects that database it knows where to look.

CryptoSQL communicates with geth through Nethereum, a library that wraps Ethereum functions into .NET calls [10].

CryptoSQL supports several features not directly related to testing, such as estimating gas cost of a potential commit and the barebones for an exchange rate system to show costs in CAD. Expandability support is extensively used, with interfaces and abstract classes that can be inherited by future providers.

Source code can be viewed on GitHub [11].

Appendix 2.2 – Issued commands

Ethereum client is run with the command “geth --rpc --rinkeby”. The rpc switch tells geth to bind an HTTP interface, and rinkeby tells it to use the Rinkeby testnet.

The GitHub repo has the specific MySQL commands listed in the order they are executed, in the directory CryptoSql.MySql\Readme.md.

Appendix 2.3 – Cost Calculations

All costs are in Canadian Dollars. Hardware prices are market values obtained from PCPartPicker [12].

Appendix 2.3.1 – Local server

The following hardware costs are for models identical to the test hardware used. Similar performance could likely be obtained with cheaper hardware, this is explored but not tested.

- Samsung 960 Evo 500GB = $280
- Intel Core i7-8700k = $440
- 32GB DDR4 2400MHz = $428
- Gigabyte Aorus Gaming 7 = $280

These hardware costs do not affect performance, so they are the reasonable minimum required:

- 450W Power Supply = $87

- Case = $35

Total = $1550
Estimated average power usage = 80W

For proposed hardware that could get similar performance:

- Samsung 860 Evo 250GB = $119
- Intel Core i3-8100 = $144
- 8GB DDR4 2400MHz = $75
- LGA1151 Motherboard = $40
- 450W Power Supply = $87
- Case = $35

Total = $500
Estimated average power usage = 70W

Note the big power usage hitters (namely CPU) are mostly idle so power remains low.

For the subscription (uptime) costs we simply calculate the per hour cost of running the hardware.

$$s = P \times C \div 1000$$

With s the subscription cost, P the power usage, and C the cost per kilowatt-hour

With a value of $0.20/kWh (average Ontario price with fees [13]):
As tested = $0.016/hr
Minimal = $0.014/hr

We estimate usage based on drive depreciation by the total writes rating. Read costs are considered free since no read limits are given on the warranty.

$u = \frac{V}{R}$ where u is the price in dollars per gigabyte, V is the value of the drive to start, and R is the rating.

For the 960 Evo rated to 200TB, cost = $0.0014/GB
For the 860 Evo rated to 150TB, cost = $0.0008/GB

Appendix 2.3.2 – Amazon cloud

Values are used for the US East region. The RDS used is db.t2.small which has 1 vCPU and 2GB of RAM.

Startup costs are 0.
Subscription cost as tested are $0.034/hr for single region and $0.068 for multi-AZ, plus a storage cost of $0.115 per GB-month. When using an instance close to tested local/blockchain hardware, subscription costs are $0.70/hr (db.m4.2xlarge) [14].
Write usage is very low, free up to 10TB where it goes to $0.09/GB. Read usage is $0.02/GB [14].

Appendix 2.3.3 – Blockchain

Startup and subscription costs are the same as with local server in Appendix 2.2.2.1.

Usage costs are 0 for reading and vary for writing depending on gas price and Ether value. As of writing this is at 1 GWei [15] and 1 ETH = $772 [16].
Gas required to send a transaction are described in the Ethereum yellow paper [17]:
$w = 21000 + 4z + 68n$ where w is the write cost, z is the number of empty bytes, and n is the number of nonempty bytes.
If we assume all bytes are nonempty, we can say that the cost per gigabyte is ~68 billion gas. Since 1GWei = 0.000000001 ETH, this is around 68 ETH = ~$52000!
However, the transaction is limited to 7 million gas, or around 100KB. So the most we can ever spend on one transaction is 0.007 ETH = $5.40. Furthermore, there is an unpublished limit of 32KB in the geth client [6].

Appendix 3 Future projects

Appendix 3.1 Distributed transactional storage

The current monolithic provider has to fit the whole database in one transaction. It would be possible to split up the bytes among several transactions, but this is unideal since it could quickly require sending dozens of transactions to update one row.

A better solution is to create a distribution of transactions that are linked to each other. Here is the overall architecture:

- One master transaction represents the database and stores links to every table. This transaction is saved by CryptoSQL on disk so that it can find it on load, and so other computers can access it. The data may be preceded by a special character to enable automatic finding of the most recent copy if the disk file is lost.
- The master transaction contains a list of tables it defines, each table is represented by a transaction hash.
- Each table transaction contains a short header that describes the column count/types. After that, it contains a list of transactions that link to row data transactions.
- Row data transactions contain the row content, up to the extent that the transaction limit allows. When one set is full, another is added to the table transaction.

The drawbacks:

- Updating one row requires sending 3 transactions sequentially (roughly 2min total).
- Retrieving data requires requesting 3 transactions sequentially (roughly 3s total).

The benefits:

- With continued nesting, there is no limit to the size of the database.
- 3 nesting levels can store a considerable number of rows (quantified below).
- Read permission can be granted to light consumers (e.g. mobile) for single sets of rows. This could be useful e.g. for a game, when the user logs in to check their stats, the company's server simply sends them the 32-byte transaction address containing only their info and the rest of the computation is done client-side, saving the company money.

How much data can we store at 3 nesting levels?

If we consider the amount of transactions the "indexer" transactions can hold, and assume the row sets hold ~400 rows (remember monolithic provider could hold 600):

Each transaction address is 32 bytes.

Each transaction can hold 32KB.

An efficient transaction that holds nothing but key-value pairs for transactions can hold $\frac{32\times1024}{x+32}$ of them, where x is the byte length of the key. We want to maximize this by minimizing x, but it obviously cannot 0 (then we don't have a key!) so we try 1 and get $\frac{32\times1024}{33} = 992.96$. However, a 1-byte key is not enough to index 990 values since it only goes up to 255. So, with a 2-byte key (65536 keys) we can hold just over 960 transactions. That means our database master transaction can hold 960 tables.

For the table transactions, they don't need indexers, so we can get (assuming 32-byte header for definitions) $\frac{32\times1024-32}{32} = 1023$ transactions that point to row sets.

Therefore, with 400 rows in a set, we can get $960\ \times 1023\ \times 400\ =\ 392832000$ rows. For comparison, assuming a monolithic provider can store 600 rows per transaction, that would require spreading over 654720 transactions to store. Since monolithic requires sending the whole thing for one row update, that 3 transaction update is looking good.

Taking this one step further, at 4 nesting levels we could have over 400 billion rows. Although the cost required to fill up 3 nesting levels is astronomical so it's not likely this is needed.

Bonuses:

- A "light" version of distributed transactions could be used where the master transaction is a table definition, that way up to $1023 \times 400\ =\ 409200$ rows can be managed with a nesting level of two. The downside of course being that the CryptoSQL host is now responsible for keeping many more transaction hashes, and it would be harder to work back in the case of a failure.
- The row sets could be partitioned, a feature that MySQL supports to quicken searching. E.g. the first row set transaction would be responsible for A-D, then E-H, etc., allowing for efficient alphabetical searching.

To conclude, distributed transactions can provide a feasible solution to the 32KB data limit. By doubling the number of transactions you have to update for changing/adding a row, you get access to ~682 times as many rows. This makes a better solution than monolithic as soon as the first transaction limit is reached.