PGMint and PGEther: Building a Decentralized Source-of-Truth With SQL & Blockchain

Vivek Gopalan

Advisor: Prof. Avi Silberschatz

Department of Computer Science

Yale University

This thesis is submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2021

Abstract:

As use cases emerge for decentralized applications, such as finance, art, publication, and gaming, amongst others, the demand for software developers to build these platforms will increase dramatically. While nascent, infrastructure-as-a-service companies are taking on the challenge associated with developing on top of blockchains like Ethereum. These companies, such as Infura and Alchemy, are focused on providing application developers an easy way to interact with their blockchain, deploy smart contracts, and test behavior in a development environment. Additionally, many common APIs are receiving decentralized equivalents, such as IPFS being the distributed analogue to S3 for object and file storage. As a whole, it appears that there is a secular movement to translate paradigms that application developers are comfortable with in the centralized world to the decentralized.

However, one such space that has yet to be innovated in is that of decentralized relational databases. If these new applications seek to be low latency or high-throughput while still being globally available, they need to have an efficient and query-able source-of-truth. Ledgers and hash trees do not lend themselves to fast data access or complex queries (e.g. joins). Thus, an architecture that can introduce relational data access patterns, while remaining decentralized, can fill this market need. This paper proposes various system designs based on Ethereum and Tendermint, two different blockchain frameworks, as well as an implementation that combines Postgres with Tendermint. The paper also details the performance and scalability of this implementation, benchmarking timings to load a large dataset with up to nine nodes. I additionally discuss methods for achieving strong consistency in this setting.

Section 1 - Introduction:

Some of the myriad reasons why blockchain has been so popular in the past five years include the desirable properties of decentralization, such as geo-distribution of data and global transactions, as well as the prospect of building decentralized applications (DApps). However, these DApps will need a fast, query-able source-of-truth that is also trustless (does not require any trusted third-parties to host and manage data) in order to run efficiently at scale and maintain their truly decentralized nature. This a union of properties that neither cloud-native distributed databases (e.g. Google Spanner, AWS Aurora) nor blockchains (e.g. Bitcoin, Ethereum, Ripple) have on their own.

This paper explores the intersection of blockchains and databases. Specifically, the goal is to build a relational database with blockchain providing the underlying replication and consensus layer. This should not be confused with existing methods that extend query-ability to the blockchain in a read-only manner and instead should be considered in the same space as projects

like BigchainDB[1], MongoDB on top of Tendermint's blockchain framework, and CovenantSQL[2], SQLite on top of a custom implementation of Delegated Proof-of-Stake.

In this paper I will describe two architectures for implementing a decentralized relational database with blockchain providing the underlying replication and consensus layer. Prior to that, I will elaborate on the potential use cases for this technology, the existing market landscape, and required background information on relevant blockchain frameworks (Tendermint & Ethereum) as well as their consensus mechanisms.

Section 1.1 - Motivation:

The proposed architectures in this paper will have a relational query layer, following the wave of NewSQL[3] databases that combine the transactional consistency of an RDBMS with the horizontal scalability of NoSQL databases. This is inspired by architectures such as Spanner[4] and Calvin[5] which offer consistent geo-distributed data at low-latency, perfect for online transaction processing (OLTP) workloads. There is also a secondary, application-developer-centric, reason for this choice. In the status quo, in order to build DApps on popular blockchain networks like Ethereum, one must learn how to utilize the scripting languages that can run on that blockchain (e.g. Solidity & Vyper for Ethereum). However, this is a time-consuming and iterative process with no clear consensus on developer workflow and multiple tech stacks competing for developer adoption. This paper's architectures propose an easy developer transition - by maintaining parity with a popular RDBMS like Postgres, the millions of developers that utilize SQL in their existing stack can easily migrate to the world of DApps with relatively low friction.

There are also a number of other viable use cases for a blockchain-based database:

- Inter-organization data sharing in a trustless setting. This idea has applications in the world of finance, such as with decentralized exchanges.
- Enterprise private blockchain for intra-organization data sharing in a ledgered manner - something akin to Google Sheets but with a query language. This can be useful in cases where data lineage is important, as all transactions are recorded on the blockchain. This is

[1] *BigchainDB GmbH, Berlin, Germany.*, "BigchainDB 2.0 The Blockchain Database"
[2] Wang, Auxten. "CovenantSQL: the SQL Database on Blockchain."
[3] Pavlo, Andrew, and Matthew Aslett. "What's Really New with NewSQL?"
[4] Corbett, James C., et al. "Spanner."
[5] Thomson, Alexander, et al. "Calvin."

one use case of Amazon's Quantum Ledger Database[6], which provides an immutable transaction log to help track and verify changes to data.

- Easy non-fungible token (NFT) creation and issuance using SQL. This would require the architecture to introduce some immutability guarantees that are not possible with a traditional RDBMS. One possible way to imagine NFTs in a relational setting is to think of a relation as defining the attributes of a set of NFTs, including a unique identifier and the current owner. Each record would represent a unique NFT. However, in order to enforce scarcity of supply, after time of table creation and initial insertion, one must not be able to insert or delete any additional records. Furthermore, one must enforce row-level ownership - the only account that should be able to update an individual record is the owner of that record.

Section 2 - Background:

In this section, I will define useful terms and will lay some of the groundwork for my later architectural decisions. For the sake of brevity, I will assume that the reader has knowledge of basic blockchain fundamentals (e.g. mining, block headers & hashing, notion of the longest-chain, etc.). In the event that a consensus algorithm other than Proof-of-Work can be used in relevant blockchain frameworks to this paper (e.g. Tendermint BFT), I will cover the basics in the following subsections.

Section 2.1 - Tendermint:

Tendermint[7], broadly speaking, is a combination of a blockchain consensus engine and an application-blockchain interface that allows an application developer to consistently replicate arbitrary application state across many nodes. Tendermint is also Byzantine Fault Tolerant, a property that is extremely important in trustless decentralized systems as a network needs to be able to tolerate many different failure modes, including bad actors. Tendermint can handle up to one third of nodes in a given network failing. Tendermint's design goal is to move away from monolithic codebases at each node (e.g. Bitcoin) that handle all of networking, transaction pooling and ordering, consensus, unspent transaction outputs, scripting / contract code, identity

[6] "Amazon QLDB." Amazon Quantum Ledger Database
[7] *Tendermint*, tendermint.com/.

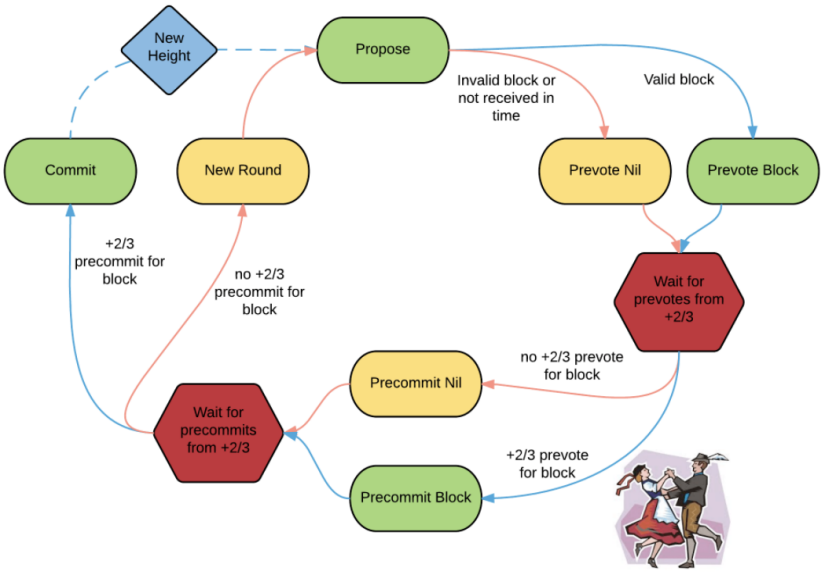
access management, API endpoints, and more[8]. Similar to the movement in web application development from the monolith to microservices, Tendermint seeks a modular framework for handling each of these individual tasks, with the ability to inject custom business logic at any tier (rather than just at the contract / scripting level with a blockchain like Ethereum).

The blockchain consensus engine of Tendermint is called the "Tendermint Core." For the consensus layer, Tendermint runs a BFT consensus[9] protocol that requires no mining. In some ways, it follows very similar paradigms to Byzantine Paxos[10] and Miguel Castro & Barbara Liskov's Practical Byzantine Fault Tolerance Algorithm (PBFT[11]). In a Tendermint network, participants in the consensus protocol are labeled validator nodes. For each new block, a new validator takes its turn as the proposer, a role similar to that of a quorum leader in Paxos. However, the proposer order is determined through a deterministic weighted round robin. This has led some to theorize potential DoS attacks on a Tendermint network where the proposer is attacked in succession.

For a given block height, the proposer packs transactions into a block and broadcasts to the block to the rest of the validators. There can be multiple rounds per block height (as some blocks can fail to be committed), however, for a given round number and block height, no two honest validators will report different proposers. Voting on a block occurs in two stages - pre-voting and pre-committing. For a given block height and round, a validator will wait for a timeout before moving to the next round number. During that time frame, the validator may receive a block from the current proposer. Once it has heard of the block in time, the validator will pre-vote in favor (if block is valid, and node is honest) and broadcast to the other nodes. If two thirds of nodes pre-vote in favor of the block, the consensus algorithm moves to the pre-commit stage. If a node hears from at least two thirds of the nodes pre-voting the block, then that node will pre-commit the block. If it hears at least two thirds of the nodes pre-committing, then the block will be committed. Note that this protocol is tolerant to at most one third of nodes being dishonest. Additionally, Tendermint introduces some locking rules that ensure that a validator cannot commit two different blocks at the same height. Below is a flowchart of the consensus algorithm:

---

[8] "What Is Tendermint?" *Tendermint Core*
[9] *Buchman, Ethan, et al.* "The Latest Gossip on BFT Consensus."
[10] Lamport, Leslie. "Byzantizing Paxos by Refinement."
[11] Castro, Miguel, and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery."



[12]

The generic application layer of Tendermint is called the Application Blockchain Interface (ABCI) which allows for modular transaction processing and validation with underlying user-application code performing these steps. Additionally, there is an invariant posed on custom application code: the state machines that are being replicated must be strictly deterministic. That is, "given the same ordered set of requests, all nodes will compute identical responses."[13] To avoid non-determinism, one must avoid race conditions, rollbacks due to serialization failures, and language specific randomness (e.g. map iteration in Go). In order to enforce that application logic behaves as intended, it should not be exposed externally and must only communicate with the Tendermint Core consensus engine through ABCI connections. Tendermint provides the ability for this to be done in a highly performant way through a built-in application, versus an external application that communicates with Tendermint Core through RPC APIs. With built-in applications, the application logic runs in the same execution environment as the rest of the Tendermint node, decreasing overall transaction latency.

In order to build an application on top of Tendermint, one must implement a series of critical ABCI functions. These broadly fall into the buckets of handling consensus & transactions, checking transactions in the transaction pool, providing user read-only RPC APIs,

[12] "Proposer Selection Procedure in Tendermint." *Tendermint Core*, Interchain GmbH
[13] "Methods and Types." *Tendermint Core*, Interchain GmbH

and handling crash recovery & bootstrapping. For each of these buckets, a Tendermint node has a separate ABCI connection. The methods and their corresponding connections are as follows.

Consensus Connection - responsible for block execution & consensus.

- **InitChain:** This method is called once whenever a new chain is initialized and utilizes the genesis.json and config.toml files (either defined by the network creator or generated through Tendermint scripts) in order to set high level parameters about the network. This includes information about the chain's unique identifier (e.g. to allow private networks), a set of parameters around consensus (e.g. block size, time between blocks), and validator nodes in the network.
- **BeginBlock:** The execution of a given block follows the following method order - BeginBlock → DeliverTx (for all Tx in the Block) → EndBlock → Commit. Transaction order within a block is decided ahead of execution and is ordered based on the timestamp received. An additional account-based nonce can be used in the application layer to ensure that two transactions from the same user are ordered in the order that they were sent, regardless of time received. For a BeginBlock request, the hash, header, and last commit information must be included.
- **DeliverTx:** This performs the brunt of the work in a Tendermint application, executing the entirety of a given transaction. It takes in the bytes of a transaction request and applies a state transition function, emitting any response code, logs, and additional events as a side-effect of execution.
- **EndBlock:** This signals that the transactions within a block have finished and updates some consensus & validator-level parameters.
- **Commit:** This signals that the block should be committed. Locks and flushes the mempool and updates all connection states.

Mempool Connection - responsible for determining transaction validity.

- **CheckTx:** A node's transaction order is determined within its mempool. In order for a transaction to enter the node's mempool, it must be checked via CheckTx, which performs simple validation (e.g. on hashes and signatures as well as any arbitrary application-specific checks).

Info Connection - responsible for responding to user read-only requests.

- **Info:** This method returns information about state (such as block height, latest block hash, and application version) to an end user. During each Commit call (one per block), the latest block hash and block height are updated.
- **Query:** This method is extremely flexible and accepts an arbitrary data field. The only constraint is that this method must not change application state. It can respond with arbitrary information, logging, key-value pairs, and Merkle proofs for the response data.

Snapshot Connection - responsible for recovery.

- **ListSnapshots:** Used for snapshot discovery between peers.
- **LoadSnapshotChunk:** Used to load a specific snapshot from a peer during state synchronization.
- **OfferSnapshot:** Used in bootstrapping another node - send a snapshot to a peer.
- **ApplySnapshotChunk:** Apply a snapshot (versus replaying blocks) in bootstrapping.

For examples of large projects built on top of Tendermint, one can look at the Cosmos Network.

Section 2.2 - Ethereum:

Ethereum can be thought of as a state transition system combined with a consensus layer to agree on the order and contents of the state transitions.[14] In Ethereum, state is represented by accounts: each account has an address, a balance, and a nonce (to ensure transactions are not processed multiple times). A state transition is represented by a transaction of information between accounts. In Ethereum, currency is known as "Ether," which has the lowest denomination of the gwei (10^{-9} Ether).

There are two types of accounts in Ethereum:

- Externally Owned Accounts (EOA), which have a corresponding private key as authentication for transactions. These are accounts that individuals use to interact with the Ethereum network and send/receive transactions.
- Contract Accounts, which have two additional fields. One is the contract's code, which is immutable once the contract is deployed. The other is the contract's persistent storage. To interact with a contract account, one must send a transaction to the account which triggers

[14] Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger."

the execution of contract code, parametrized by the contents of the transaction. Contract code can be written in a few different frameworks, one of which is an object oriented language called Solidity developed by the Ethereum Foundation.

A transaction is a signed packet of data that is sent from an EOA to either another EOA or a Contract Account. A transaction must contain the following fields: the address of the recipient, a signature, the amount of gas to exchange, the number of computational steps involved in the transaction ("startgas"), payment per step ("gasprice"), and an optional data field. The optional data field is what is accessed by a smart contract to parameterize function calls. "startgas" and "gasprice" help protect against DDoS / computational wastage in contract code. Ethereum has fees for storage and computation (e.g. 1 gas per step, 5 gas per byte). This means that an individual pays akin to the serverless model; proportional to what computation and storage they use. If the gas budget provided in the transaction is exceeded by what is consumed by the network, the transaction will terminate when it reaches that threshold.

For an Ethereum state transition, a number of things must occur. A node must validate the number of parameters, the nonce value, subtract the transaction fee from the sender's balance, transfer the amount to the recipient, perform appropriate error checking, and pay the miner their reward, amongst other actions. Transactions are lumped together into blocks and have to be mined. Mining involves solving a cryptographic puzzle where the difficulty is proportional to the current length of the chain. The miner must also perform the above steps for every transaction in the block to accurately represent the state transition. They also must execute the contract code as it is part of the definition of the state transition function. Blocks are important to how participants agree on state and history. Blocks contain a hash that is based on the contents of the previous block in the chain - thus linking blocks together. This means a change in a block in the past will affect blocks in the future (as hashes will now change), thus alerting to fraud.

The final important topic related to Ethereum has to do with smart contracts and code execution. The bytecode for a smart contract can be seen as a stack of instructions that are executed in an infinite loop that iteratively increments the program counter until either the stack is empty or the gas limit has been reached. A contract has two different spaces to store data:

- Memory: Does not persist to the blockchain and can only be used within the scope of a contract's code execution. The cost of memory is approximately 3 gas for 32 bytes.

However, this gas price scales proportionally to the amount of work in a given contract's execution.

- Storage: A persistent key-value storage that is written into the Ethereum blockchain. However, the per-byte cost of Ethereum's persistent storage is far higher than memory and the stack (20000 gas for 32 bytes of storage). The only field size is 32 bytes - there are no smaller denominations. Thus Ethereum optimizes with packing under the hood.

Section 2.4 - the DApp:[15]

DApps are applications with some back-end code execution in a decentralized network as opposed to a central server in the traditional client-server architecture. Additionally, DApps can have front-end code hosted on decentralized file storage such as IPFS[16] or ARWeave[17]. For many in the status quo building on Ethereum, DApp code is deployed as immutable smart contracts, providing trusted code execution for end users. In the case of Tendermint, application developers are building hundreds[18] of DApps that utilize its protocols and highly flexible programming nature. These range in use cases from gaming - with virtual economies emerging on the blockchain - to content creation sites (blogs, forums, crowd-sourcing) and DeFi.

Section 3 - Landscape:

Before I propose any architectures, I will first briefly document the competitive landscape and where existing blockchain databases exist in the broader space.

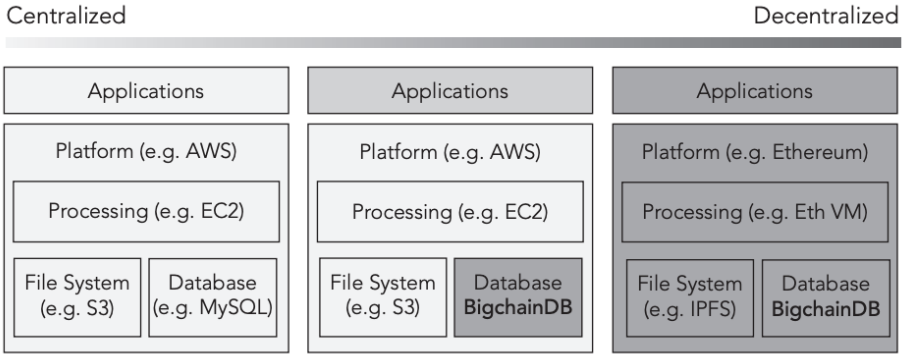
Section 3.1 - BigchainDB:

BigchainDB is a BFT decentralized database that utilizes Tendermint for networking and consensus with each node having a local MongoDB instance. As it uses Tendermint as the underlying layer, BigchainDB can tolerate a third of its nodes failing. However, BigchainDB treats its underlying data as append-only. Once a particular document has been written globally, it cannot be edited or deleted through BigchainDB, though a node could locally change their copy. Additionally, BigchainDB gives the user the ability to create user-defined "assets."

---

[15] Cai, Wei, et al. "Decentralized Applications: The Blockchain-Empowered Software System."
[16] Benet, Juan. "Ipfs-content addressed, versioned, p2p file system."
[17] Williams, Sam, et al. "Arweave: A Protocol for Economically Sustainable Information Permanence."
[18] Asmodat, et al. "List of Projects in Cosmos & Tendermint Ecosystem."

Owner-controlled assets are prevalent in the blockchain world. Within the BigchainDB transaction specification, it allows for a user to create tokens, assign owners to them, and transfer tokens amongst user accounts all through a combination of their own APIs and MongoDB queries. The following image is an example of how BigchainDB can slot itself into application stacks:



[19]

Section 3.2 - BlockchainDB:[20]

BlockchainDB is an attempt to build a database with underlying blockchain-based storage with sharding at its core. Tables are partitioned, and each shard will live on a separate blockchain. Instead of having data replicated to every peer, creating latency due to replication overhead, each shard-blockchain is only replicated to a subset of peers. BlockchainDB provides a simple key-value storage API, with three operations: **get(table, key) → value**, **put(table, key, value)**, and **verify()** which returns the truth over whether the immediately prior transaction was committed. The overall goal here is to provide key-value database abstractions on top of immutable, decentralized storage.

Section 3.3 - Amazon Quantum Ledger Database:[21]

QLDB has use-cases in insurance, supply chain, and banking. It is an append-only immutable ledger that has a query layer on top (PartiQL SQL for relational, nested, and semi-structured data). Within the ledger there exist "tables" which can be understood as JSON (Amazon Ion) document stores. A user can make queries on their ledger to yield historical information, perform joins on current state information, and cryptographically verify transactions.

---

[19] *BigchainDB GmbH, Berlin, Germany.*, "BigchainDB 2.0 The Blockchain Database"
[20] El-Hindi, Muhammad, et al. "BlockchainDB."
[21] Engdahl, Sylvia. "Blogs." *Amazon*, Greenhaven Press/Gale

Section 3.4 - CovenantSQL:[22]

An open-source alternative to Amazon's QLDB with an SQLite query layer. Just like Ethereum, transactions on the network will require gas to compute. Additionally, CovenantSQL relies heavily on a concept known as sidechaining - having smaller blockchains that run in parallel to a main chain that synchronize at regular block intervals and share information between each other to guarantee consistency. In CovenantSQL, each database has its own sidechain.

Section 4 - Architecture & Implementation:

This section will detail an implementation of a blockchain database utilizing Tendermint and Postgres. For all future references, this software will be titled "PGMint." PGMint takes inspiration from the aforementioned BigchainDB project, which combines Tendermint with MongoDB for a highly scalable datastore.

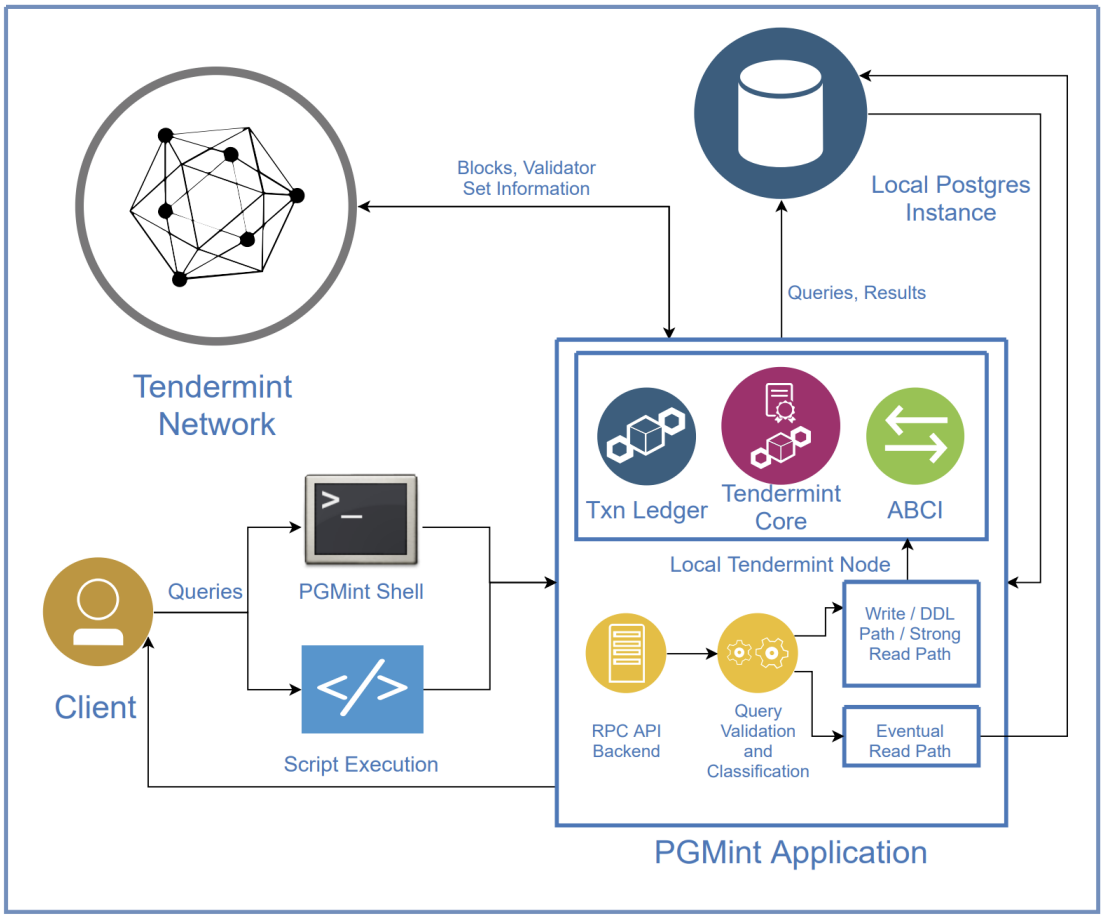
The high-level architecture is that at each PGMint node, there will be a Tendermint validator, a built-in middleware application, and a local Postgres instance. The primary method of interacting with the PGMint database is through JSON-RPC calls to the endpoints provided by Tendermint with the aforementioned connections. The primary API used for reading from the database will be **query** and the primary API used for writing to the database will be **broadcast_tx_{commit | async | sync}**, which return with responses from the CheckTx and DeliverTx ABCI methods. When a user submits the query, the underlying application middleware will first validate that the query is made in the proper API access path.

In the case of a write, Tendermint will add the transaction to the mempool after checking the validity of the SQL statement through custom application logic (CheckTx). The proposer for the current height and round will then pack a series of Tendermint transactions, each of which either represents a single-statement query or a SQL transaction block. If the block is committed through the consensus algorithm, then each node will execute the statement(s) from each transaction in the order that they appear in the block (DeliverTx), serially writing to the local Postgres instance, thus guaranteeing write consistency across multiple nodes.

---

[22] Wang, Auxten. "CovenantSQL: the SQL Database on Blockchain."

In the case of a read (Query API), depending on how the network was initially configured, the read will either be executed on the local Postgres instance (if eventual consistency is all that is required) or sent to the last block's proposer to execute on their local Postgres instance. As a whole, these statements are not written to the blockchain. The client can also submit queries through an interactive shell or upload SQL scripts to a batch processing program that abstracts away JSON-RPC calls to the underlying blockchain database.
An architectural diagram is as follows:



Section 4.1 - Bespoke ABCI Function Implementations:

- **DeliverTx:** This application code executes the query (or SQL transaction block) in the local Postgres instance using the Go SQL interface. It appends the transaction statement to the transaction Merkle tree. It replies with a response containing the statement type and the original query. In a transaction block, reads are ignored.
- **CheckTx:** This uses sqlparser to determine the type of statement for each individual query. It also validates the syntax of the statement and disallows DROPs.

- **Query:** Executes a SELECT either locally or forwards the query to the proposer of the last block. Responds with a byte array of the results that have been marshalled into a JSON.

Section 4.2 - Transaction Properties:

The architecture described above guarantees consistency of writes across all nodes. Consensus is achieved on the order of statements for a given block. Then those statements are executed serially on the local Postgres node. Thus, the states of any two honest nodes at a given block height will be converged.

There are two ways for PGMint to guarantee strong read consistency. The first method is by performing quorum reads. This is a traditional method for achieving strong replica consistency and requires reading from every node and reporting the quorum result (in our case, two thirds of the nodes must be in the quorum for the query to succeed). This is highly inefficient and introduces many additional data round trips, linear with the number of validators in the network. The other method is to query the last block proposer directly. Based on properties of Tendermint's consensus algorithm, the data held by the proposer of the block with the highest height must be guaranteed to be consistent as all transactions up to that block must be delivered, else the block hash would have been rejected. This method drastically improves the performance of the system.

Section 4.3 - Node / Network Startup:

Each node must share the same genesis config file and list of network peers on startup. For adding new nodes to an existing network, Tendermint has protocols for peer discovery. When spinning up a new node, PGMint assumes that a local Postgres server is running on the host and port provided in its configuration data. It will attempt to connect to the **pgmint** database at the given host and port. Then, Tendermint code will discover peers and perform handshakes, retrieving any blocks that the local node may be missing. For each of these local blocks, the underlying application code will execute the transactions serially in order to catch up. If at some point a node goes down, Tendermint will be able to restore node state from a snapshot and then proceed to simply replay transactions that were missed during the outage.

Section 5 - Benchmarks/Performance:

Now that we have discussed the architecture, I will present an implementation of the PGMint design. The source code can be found here: https://github.com/vvkgopalan/pgmint. To run a node locally, perform the following steps:

1. First install Go and Python 3. Then navigate to your **$GOPATH**. Clone the repository in your **$GOPATH**.
2. Navigate to the **src** directory. Here we will initialize our Postgres instance. By default, the host and port that this instance will bind to are **localhost** and port **5432**. We run the two commands in succession:
    a. initdb -D pgdata
    b. pg_ctl -D pgdata -o "-p <port number>" -l logfile start
3. Then we build the source for the ABCI application by running **go build**. This may initially fail due to missing dependencies. Install all required dependencies (can be found in **go.mod**).

To interact with the node directly, please follow the Tendermint RPC reference[23]. For convenience, I have written a shell that behaves very similar to **psql** that abstracts away the underlying **broadcast_tx_*** and **abci_*** JSON-RPC API calls. Here, you can type native Postgres queries and have them propagated to the Tendermint node for validation and execution. To run this shell, run the following command:

python shell.py <src> <n_nodes> <consistency_level>.

In this command, **src** refers to the relative path to your ABCI application source directory, **n_nodes** refers to the number of nodes currently running, and **consistency_level** refers to the tunable read consistency level (one of {"strong" | "eventual"}).

To create local networks for testing purposes, I have developed a automated testnet generator that proceeds through the following workflow:

1. Cleans up any existing Postgres processes and removes data directories. Does the same for Tendermint processes.
2. Copies the source directory **n_nodes** times, building the ABCI application source within each.

[23] *Tendermint RPC Reference*, Interchain GmbH

3. Initializes the validator nodes. Updates config files such that each node is aware of its persistent peers. Updates genesis files such that all nodes share the same information.
4. Assigns non-interfering ports for each node.
5. Spawns an ABCI application process for each node.

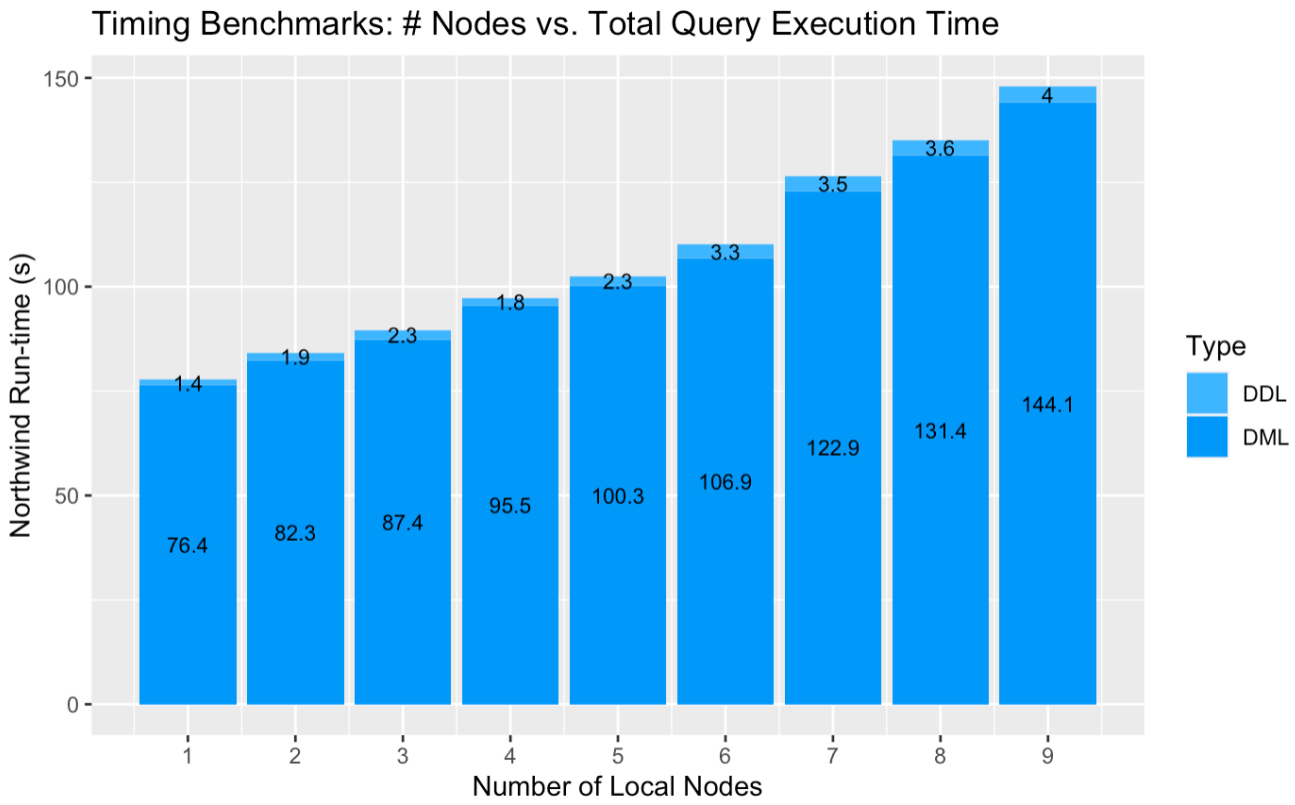
This script can be run through the following command:

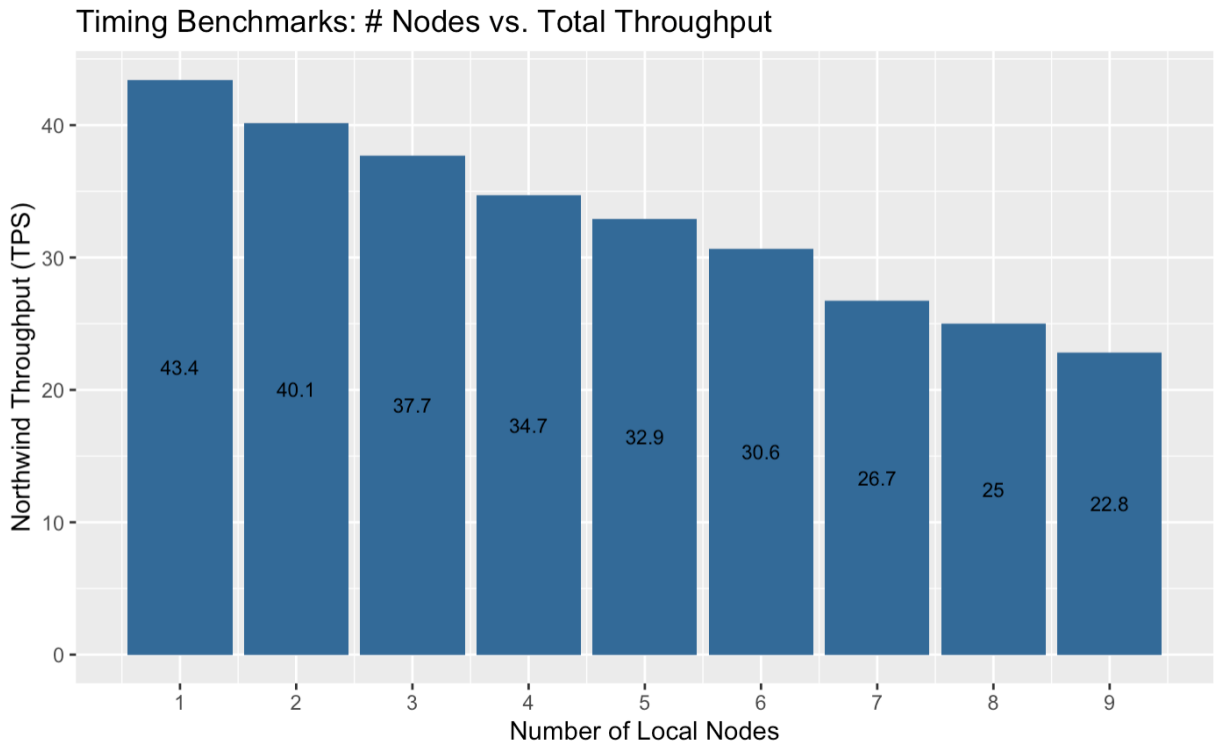
python testnet.py {start | destroy} <src> <n_nodes>

Additionally, to conduct tests with asynchronous RPC calls (e.g. making batches of queries to the database asynchronously), I have written **batch_sql.py**, which reads in a **.sql** file and adds all transactions within it to the Tendermint mempool in order. It then repeatedly queries the node to see whether every transaction was included in the chain, returning only when all transactions have persisted.

To benchmark the performance of this system under various configurations, I have taken the example of the Northwind Sample Database[24]. Using these files, which consist of 3376 DDLs + DMLs, I computed the total execution time as I varied the number of nodes in the network. I ran this test locally in a non-rigorous setting - a 2.8 GHz i7 with 4 physical (8 logical) cores - just to get a sense of relative performance trends.

[24] Yugabyte. “Northwind Sample Database.”


Timing Benchmarks: # Nodes vs. Total Query Execution Time
Northwind Run-time (s)
150
100
50
0
1.4
76.4
1.9
82.3
2.3
87.4
1.8
95.5
2.3
100.3
3.3
106.9
3.5
122.9
3.6
131.4
4
144.1
1
2
3
4
5
6
7
8
9
Number of Local Nodes
Type
DDL
DML


Timing Benchmarks: # Nodes vs. Total Throughput
Northwind Throughput (TPS)
40
30
20
10
0
43.4
40.1
37.7
34.7
32.9
30.6
26.7
25
22.8
1
2
3
4
5
6
7
8
9
Number of Local Nodes

As is evident from the graphs, we find that our network performs slower as we increase the number of validators. This makes a lot of intuitive sense - by increasing the number of nodes, we also increase the communication overhead. Additionally, the block voting cycles will take longer as in order to commit a block, more messages will have to be communicated between peers (quadratic as a function of nodes). Note that this does not measure overall replication time (time it takes for all nodes to reflect changes locally), only the time for all transactions to be committed on-chain and replicated locally on the node I queried.

Section 6 - An Alternative Implementation on Ethereum:

This section will detail a potential implementation utilizing Ethereum and Postgres in conjunction with each other. For all future references, this software will be titled "PGEther." Other entrants in the Blockchain Database space either bring blockchain properties (e.g. ledgering) to an existing database or bring database properties (e.g. query-ability) to the blockchain. PGEther differentiates itself by utilizing the Turing-complete scripting language of Ethereum to provide a database whose state and state transitions live on the blockchain and are replicated locally for querying. The motivation for this specific architecture is similar to that of Consensys Sandcastle, which provides translation middleware between SQL statements and Smart Contract function calls[25]. PGEther further extends this idea, and rather than solely storing contract data in Ethereum storage, we also replicate locally to a Postgres instance, which speeds up data access times and allows us to perform more complex queries on the data as opposed to limiting ourselves to functions available in a smart contract.

The crux of this approach is that an Ethereum blockchain serves as the source of truth for a given database, with an individual relation represented as a smart contract that stores the state of a table as of the most recently mined block. This is an extremely cost-prohibitive approach on the Ethereum mainnet and is completely impractical.

- It costs 20,000 gas to store a 256-bit word. Thus, 1MB of data would be 655,360,000 gas.
- gas ~ gwei (10^{-9} ETH) prices are around 200. Thus, a MB of data would be approximately 131.072 ETH and roughly 235,963 USD.

However, the key is utilizing the Ethereum protocol on private networks (note that private in this case refers to isolated but not necessarily protected). Miners can mine empty blocks on a private

[25] ConsenSys. "Sandcastle Brings SQL to Ethereum Smart Contracts."

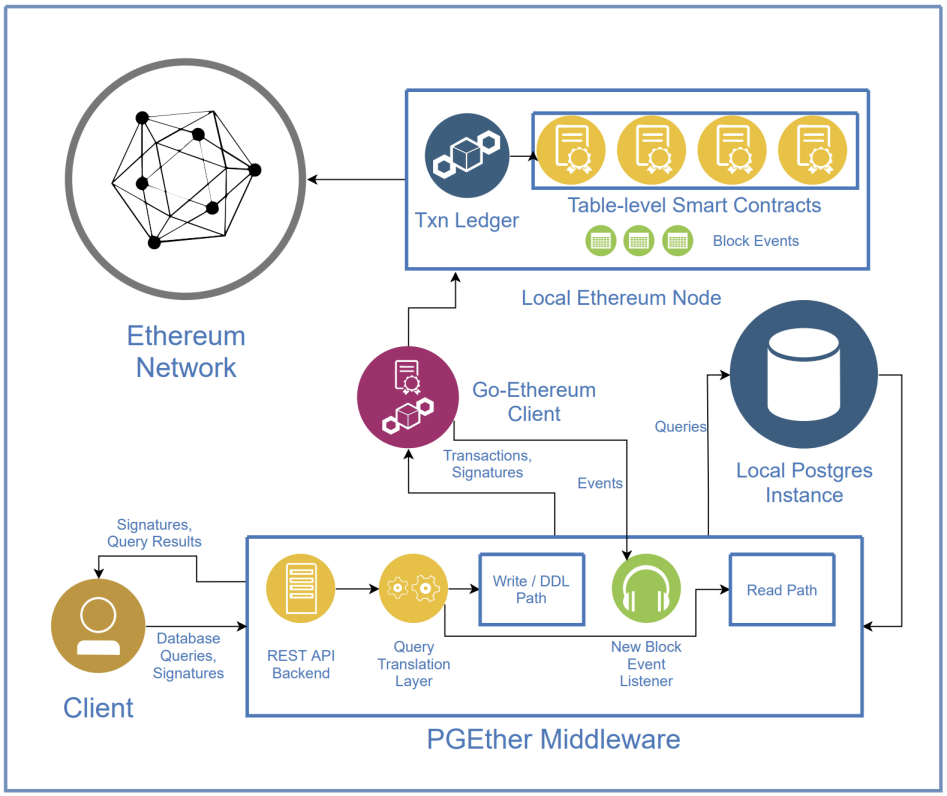
network to receive gas as a reward and can expend that gas to perform database transactions. As the mining difficulty and gas price are proportional to blockchain length, on a new private network these costs will be negligible at first.

The nodes in this blockchain will consist of PGEther-operated miners and validators, though, as it is impossible to discriminate between nodes connecting to a blockchain, other externally owned accounts could theoretically also mine on the network and send transactions. There are safeguards in place to ensure that those transactions are not reflected in the database state unless properly formed.

There are two types of nodes in the network:

- PGEther validator nodes. These nodes take and process queries, generate transactions, and replicate transactions from the blockchain to the local Postgres instance.
- PGEther mining nodes. These nodes execute smart contract transactions and mine blocks. They cannot take user requests.

DApps can send queries to a PGEther validator node through the node's REST API. The node will then execute the proper transaction path for the query. A high-level architectural diagram is as follows:



Section 5.1 - Transaction Properties

Every query (CREATE | SELECT | UPDATE | INSERT | DELETE) is implicitly formed as a single-statement transaction. However, we have differing transaction semantics for reads and writes. PGEther can be most accurately described as a BASE system that is almost entirely in the direction of availability. PGEther offers eventual consistency. A read will not be blocked by a write in progress as the time to mine a transaction sent to a smart contract and add it to the chain far exceeds that of a local write. It does not make sense to block local reads on pending global blockchain transactions. Reads will only be blocked from proceeding in the scenario where the local instance is being updated to reflect a new transaction block.

It's important to also note that a row exists in a table in the local Postgres instance if and only if it has been committed to the blockchain at some block height. For serializability, transactions in a block on the blockchain are linearly serialized in the order that they packed.

Section 5.2 - Identity Access Management & Data Ownership:

Identity access management & securing data access patterns is critical to every application developer, and it is even more so when data is decentralized. For example, take the use-case of inter-organization data sharing (e.g. many pharmaceutical research labs sharing data about their COVID vaccine clinical trials). In this case, there are a couple desired properties:

- The database is secure and can only be read by privileged members of these organizations (context specific need).
- An organization’s records can only be changed by those within the organization.
- A ledger of all changes / interactions with the database must exist (solved inherently by Ethereum).

For now, I will ignore the first need as it does not come up in truly decentralized settings. The problem in question could be solved by some combination of encrypting smart contract data and emitted events, pre-sharing keys, and introducing a centralized authentication layer in order to create a node in a given network.

The second need is far more interesting, and the solution involves having a hierarchical permissioning paradigm, similar to that of RDBMSs. However, we do not permission reads (as the blockchain itself can be read by any individual miner / validator) - we only seek to permission writes.

- We introduce table-level permissioning. When an individual submits a transaction to a table's smart contract through PGEther, the contract checks whether the user's public key is within the set of individuals who have been granted modification access to the table. As a result, PGEther would support the GRANT {INSERT | UPDATE | DELETE} family of queries.
- Each table contract will need to maintain a set of users that will be permissioned for INSERT | UPDATE | DELETE access on the table. Additionally, the table contract will need to maintain the set of individuals with GRANT OPTION (by default the creator of the contract). By default, a table will be able to be modified by any individual with any access type (the contract must maintain a flag for this as well, set at creation time). The rationale behind this is that otherwise, the table owner would need to expend gas to properly permission the table.
- Additionally, we must introduce the contract analogue to Postgres roles. The table contract will maintain a mapping of role name to the users present in the role. PGEther will also need to support the CREATE ROLE and {GRANT | REVOKE} ROLE families of queries.
- To go even deeper, PGEther can support row-level security. That is, if configured to do so, a table will maintain a column of the owners of given row records. By default, the owner will be the one who inserted the record. The owner can be changed through an UPDATE statement. This means that in order to execute an update or deletion of records, a user must both have the proper permissions on the table (if applicable) as well as be the row owner or part of the role that owns the row for every row in the query.

Section 5.3 - DDL Path:

To make sure that an application developer's workflow is unchanged, a user will never be required to write a smart contract. PGEther will translate a DDL to an appropriate smart contract with CRUD operations. Example code can be found here.

- **enable_perms()** - can only be called by table creator
- **get_owner()** - returns address of owner, immutable
- **grant_owner(address)** - can only be granted by table creator
- **revoke_owner(address)** - can only be revoked by table creator
- **grant_write(address)** - can only be called by an existing owner
- **revoke_write(address)** - can only be called by an existing owner
- **perms_enabled()** - returns boolean of whether table-level perms are enabled for table
- **row_perms_enabled()** - returns boolean of whether row ownership contract data exists.
- **get_row_perms(uint256 key)** - return the row-level owner for a key.
- **get_row_perms(uint256[] key)** - overloaded for multiple rows.
- **update_row_perms(uint256 key, address owner)**
- **update_row_perms(uint256[] keys, address[] owners)** - dim of keys and owners must be same
- **update_one(uint256 key, string memory contents, uint most_recent_block, string memory query)** - check whether key exists first (if so emit an error for violating primary key constraint).
- **update_many(uint256[] calldata key, string[] memory contents, uint most_recent_block, string memory query, uint numelem)**
- **insert_one(uint256 key, string memory contents, string memory query)** - check whether key exists first (if so emit an error for violating primary key constraint).
- **insert_many(uint256[] calldata key, string[] memory contents, string memory query, uint numelem)**
- **delete_one(uint256 key, uint most_recent_block, string memory query)** - check whether key exists first (if no emit an error).
- **delete_many(uint256[] key, uint most_recent_block, string memory query)**

The table will be stored as a mapping of bytes to a struct corresponding to each row (the fields of the struct are each attribute). If the table has a joint primary key, the keys must be hashed together. Tables **must** have primary keys - if one is not provided, a record ID column is introduced to provide a uniqueness constraint.

- For grant and revoke owner, the contract maintains a set of owner addresses.
- For grant and revoke write, the contract maintains a set of write access addresses (if perms_enabled() returns true).
- If row_perms_enabled(), the contract maintains a mapping of primary key to row owner.

Section 5.4 - Read Path:

A user can do anything they desire to optimize local reads and writes (e.g. creating indexes). This is not interfered with by PGEther and is encouraged (as it provides non-interference with the blockchain itself). After validating the query as being a SELECT,

PGEther simply forwards the query along to the local instance and returns the resulting table back to the requester. No blockchain transactions are formed. Note that reads can be stale, with the main temporal cost being that of block mining and validation. This is one drawback of the Ethereum-based architecture.

Section 5.5 - Write Path:

The write path is arguably the most complex part of this whole system. For an UPDATE, the workflow is as follows:

- After parsing and validating the query, PGEther fetches the address of the smart contract corresponding to the table to be updated from its existing metadata. If the table does not exist, return an error.
- First, PGEther checks for any new blocks in the chain and makes updates to the local instance accordingly.
- Then PGEther begins a transaction block in the local instance.
    - Then we execute the update, adding a RETURNING * clause to yield the values of the records that are updated. If the update fails, then ROLLBACK and return to the user with error. If the update succeeds, still ROLLBACK the transaction but proceed with the global control flow. The transaction needs to be rolled back as it should not be reflected locally until it is on the blockchain (and consensus on it has been achieved).
    - PGEther constructs a transaction to the appropriate table smart contract with the update data (e.g. the keys to be updated and the new values of their rows). PGEther uses the size of data in the update to make a conservative estimate on Gas cost of the computation. Excess Gas is returned to the sender.
    - PGEther sends the blockchain transaction to the user for the user to sign.
- The signed transaction is then sent to the smart contract to be executed and incorporated into a block. This will execute the **update_one** or **update_many** function within the table's smart contract. There are two sets of checks that will happen for every row:
    - The user issuing the transaction has proper update perms for that row.
    - The integrity constraints are not violated (this must be checked again as the query could be maliciously formed by a node that is not controlled via PGEther).

- If all conditions are met, the smart contract data is appropriately updated.
- The transaction emits an event at end of execution:
    - If the transaction succeeded, it emits the original query string and success status.
    - If the transaction fails, it emits error status.
- Once the entire block has been validated and mined, the local node receives it. The local node then begins a SQL transaction block on the Postgres instance and runs the queries of the transactions that succeeded in that block (in the order that they appear in the block).

DELETEs follow the identical path as updates just with different function calls (and only requires the keys of the deleted records), and INSERTs follow a similar path but do not require the row-level permissions.

While I have yet to implement this particular architecture in full, it is apparent that there is an opportunity for middleware to translate SQL to contract code and that this software can be designed given current blockchain and database technologies.

Section 7 - Conclusion:

With the advent of recent technologies and frameworks for developing decentralized applications, such as Web3.js, Tendermint, Hyperledger Fabric, and many smart contract implementations, it has become apparent that a developer-led revolution is under way. However, it is not fully clear what the winning use-cases will be and where decentralized technologies will disrupt the technology incumbents. Nevertheless, as more use-cases coalesce, new application developers will make their foray into building DApps and discover that many programming paradigms that they were familiar with in the centralized world do not exist when developing on top of blockchains. In order to ease that transition, we must continue to transition infrastructure components that developers are familiar with and reliant on, including the relational database. In this paper we have proposed a handful of designs that incorporate both traditional blockchains and relational databases, offering an expressive query layer on top of the peer-to-peer networking, consensus, and storage protocols of both Tendermint and Ethereum. We show that with one of these designs, it is possible to implement a decentralized database that offers reasonable throughput and consistent data access. Additionally, we demonstrate that this network

can scale linearly as a function of nodes - an 8x increase in nodes only yields a 0.5x decrease in throughput.

For future development, there are a number of pieces that have still yet to be written, namely a database driver to allow for easy programmatic access to PGMint as well as enable testing using more common benchmarking suites (e.g. TPCC). Additionally, I would like to continue to experiment by building a decentralized web application that uses PGMint as the database tier.

Section 8 - Acknowledgements:

I would like to thank Prof. Silberschatz for his guidance and conversations throughout this project. In addition, I would like to offer my appreciation for the Yale Computer Science Department and everything that I have learned from its members over the past four years.

Section 9 - References:

1. Amazon Web Services. “Amazon QLDB.” Amazon Quantum Ledger Database, AWS, aws.amazon.com/qldb/.
2. Asmodat, et al. “List of Projects in Cosmos & Tendermint Ecosystem.” *Cosmos Forum*, 6 Feb. 2018, forum.cosmos.network/t/list-of-projects-in-cosmos-tendermint-ecosystem/243.
3. Benet, Juan. "Ipfs-content addressed, versioned, p2p file system." *arXiv preprint arXiv:1407.3561* (2014).
4. BigchainDB GmbH, Berlin, Germany., “BigchainDB 2.0 The Blockchain Database” White paper, 2018. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf
5. Buchman, Ethan, et al. “The Latest Gossip on BFT Consensus.” https://arxiv.org/pdf/1807.04938.pdf.
6. Cai, Wei, et al. “Decentralized Applications: The Blockchain-Empowered Software System.” *IEEE Access*, vol. 6, 2018, pp. 53019–53033., doi:10.1109/access.2018.2870644.
7. Castro, Miguel, and Barbara Liskov. “Practical Byzantine Fault Tolerance and Proactive Recovery.” *ACM Transactions on Computer Systems*, vol. 20, no. 4, 2002, pp. 398–461., doi:10.1145/571637.571640.
8. ConsenSys. “Sandcastle Brings SQL to Ethereum Smart Contracts.” *Medium*, ConsenSys Media, 3 June 2019, media.consensys.net/sandcastle-brings-sql-to-ethereum-smart-contracts-4addd1b351cd.
9. Corbett, James C., et al. “Spanner.” *ACM Transactions on Computer Systems*, vol. 31, no. 3, 2013, pp. 1–22., doi:10.1145/2491245.
10. El-Hindi, Muhammad, et al. “BlockchainDB.” *Proceedings of the VLDB Endowment*, vol. 12, no. 11, 2019, pp. 1597–1609., doi:10.14778/3342263.3342636.
11. Engdahl, Sylvia. “Blogs.” *Amazon*, Greenhaven Press/Gale, 2008, aws.amazon.com/blogs/aws/now-available-amazon-quantum-ledger-database-qldb/.
12. Lamport, Leslie. “Byzantizing Paxos by Refinement.” Lecture Notes in Computer Science, 2011, pp. 211–224., doi:10.1007/978-3-642-24100-0_22.
13. “Methods and Types.” *Tendermint Core*, Interchain GmbH, docs.tendermint.com/master/spec/abci/abci.html#determinism.

14. Pavlo, Andrew, and Matthew Aslett. “What's Really New with NewSQL?” ACM SIGMOD Record, vol. 45, no. 2, 2016, pp. 45–55., doi:10.1145/3003665.3003674.
15. “Proposer Selection Procedure in Tendermint.” *Tendermint Core*, Interchain GmbH, docs.tendermint.com/v0.32/spec/reactors/consensus/proposer-selection.html.
16. “Tendermint RPC.” *Tendermint Core*, Interchain GmbH, docs.tendermint.com/master/rpc/.
17. Thomson, Alexander, et al. “Calvin.” *Proceedings of the 2012 International Conference on Management of Data - SIGMOD '12*, 2012, doi:10.1145/2213836.2213838.
18. Wang, Auxten. “CovenantSQL: the SQL Database on Blockchain.” *Medium*, Good Audience, 7 Feb. 2019, blog.goodaudience.com/covenantsql-the-sql-database-on-blockchain-db027aaf1e0e
19. “What Is Tendermint?” *Tendermint Core*, Interchain GmbH, docs.tendermint.com/master/introduction/what-is-tendermint.html.
20. Williams, Sam, et al. "Arweave: A Protocol for Economically Sustainable Information Permanence." *arweave.org, Tech. Rep* (2019).
21. Wood, Gavin. "Ethereum: A secure decentralised generalised transaction ledger." *Ethereum project yellow paper* 151.2014 (2014): 1-32.
22. Yugabyte. “Northwind Sample Database.” *YugabyteDB Docs*, docs.yugabyte.com/latest/sample-data/northwind/.